Avant-propos

Ce polycopié contient un ensemble d'exercices de systèmes informatiques, avec leur solution, pour la valeur d'Architecture des machines et systèmes informatiques, partie systèmes. Ces exercices proviennent du livre "Les systèmes informatiques" que j'ai fait paraître chez Dunod en 1990, ainsi que des sujets d'examens qui ont eu lieu à Paris.

Christian Carrez

Professeur des Universités

Table des matières

1

Introduction

1.1. Évolution des performances

[d'après Brinch Hansen (73) complété par Krakowiak (85)] *Le but de cet exercice est de mettre en évidence, sur un système simplifié à l'extrême, l'influence de l'évolution historique des systèmes d'exploitation sur quelques grandeurs caractéristiques de leurs performances.*

On considère un ordinateur dont les organes périphériques sont un lecteur de cartes (1000 cartes/minute) et une imprimante (1000 lignes/minutes). Un “travail moyen” est ainsi défini:

- lire 300 cartes,
- utiliser le processeur pendant 1 minute,
- imprimer 500 lignes.

On suppose que tous les travaux soumis par les usagers ont des caractéristiques identiques à celles de ce travail moyen. On définit deux mesures des performances du système:

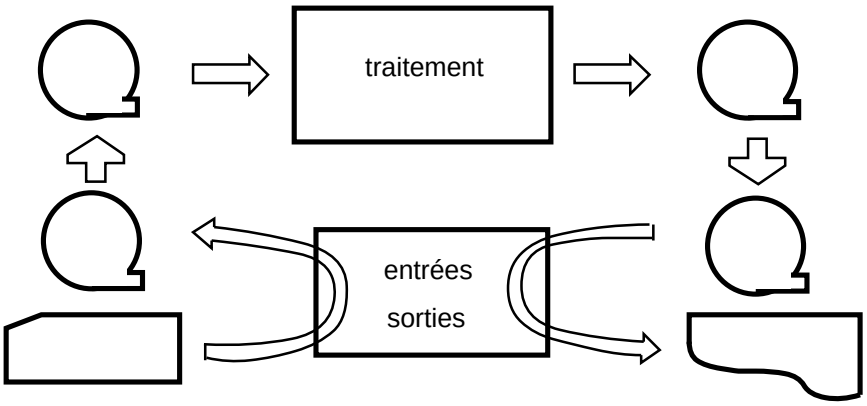
- le débit moyen *D* des travaux: nombre de travaux exécutés en une heure.
- le rendement η de l'unité centrale: fraction du temps total d'utilisation de l'unité centrale pendant lequel elle exécute du travail utile (autre que la gestion des périphériques).

A- On suppose d'abord que les périphériques sont gérés par l'unité centrale. Calculer η et *D* dans les hypothèses de fonctionnement suivantes:

A.1- Le système est exploité en porte ouverte; la durée d'une session est limitée à 15 minutes. On suppose qu'un usager a besoin de 4 minutes pour corriger son programme au vu des résultats, et faire une nouvelle soumission.

A.2- Le système est exploité avec un moniteur d'enchaînement séquentiel des travaux.

B- On suppose maintenant que les périphériques sont gérés par un ordinateur séparé, qui constitue une bande magnétique d'entrée à partir des cartes et liste sur imprimante le contenu d'une bande magnétique de sortie. L'ordinateur est alimenté par la bande magnétique d'entrée et produit la bande de sortie; on néglige la durée de lecture et d'écriture des bandes. Le temps de transfert des bandes d'un ordinateur à l'autre est de 5 minutes dans chaque sens; on suppose qu'une bande regroupe une fournée de 50 travaux (voir schéma).



B.1- On suppose que le rythme de soumission des travaux est suffisant pour occuper l'ordinateur central à plein temps. Calculer les valeurs de η et *D.*

B.2- Établir la planification de la construction des trains de travaux et calculer le temps d'attente moyen d'un usager (temps entre la soumission du travail et la réception des résultats). On admettra que les travaux arrivent à un rythme régulier, que le temps de construction d'une fournée (préparation du train de cartes) est de 10 minutes et que le temps de distribution des résultats d'une fournée (découpage et tri des listings) est de 10 minutes également.

C- Les périphériques sont maintenant gérés par un canal d'entrée-sortie. Le système est monoprogrammé, et le moniteur d'enchaînement permet à l'unité centrale d'exécuter le traitement d'un travail parallèlement à la lecture du suivant et à l'impression du précédent. Calculer dans ces conditions η et *D*. Même question si le travail moyen lit 1200 cartes et imprime 1500 lignes pour 1 minute de l'unité centrale.

D- Les entrées-sorties sont maintenant gérées avec tampons sur disque (spoule de lecture et d'impression). Le travail moyen est celui défini en 3 (1200 cartes, 1 minute et 1500 lignes).

D.1- On suppose qu'une carte et une ligne d'impression occupent respectivement 80 et 100 octets. Quelle est la taille minimale nécessaire des tampons de lecture et d'impression sur disque pour que l'unité centrale soit utilisée à son rendement maximal? Quel est alors le débit des travaux?

D.2- Le rythme d'arrivée des travaux et la taille du tampon de lecture sont ceux calculés en D.1, et la taille du tampon d'impression sur disque est de 2 Méga-octets. Quel est le rendement de l'unité centrale?

Solution de l’exercice 1.1

1.1.1. Question A

Notons que le temps de lecture de 300 cartes est de 0.3 mn, et le temps d'impression de 500 lignes est de 0.5 mn.

1.1.1.1. Question A.1

Le temps pour faire un passage est donc de 0.3 + 1 + 0.5 = 1.8 minutes. Comme entre deux passages l'usager a besoin de 4 mn pour corriger, le nombre de passages pour 15 minutes est au plus n tel que 1.8 * n + 4 * (n - 1) ≤ 15. En prenant n = 3, la durée de sa session sera de 13.4 mn. Il s'ensuit que η = 3 / 15 = 0.2, et *D* = 3 * 4 = 12

1.1.1.2. Question A.2

Lorsque le système est exploité avec un moniteur d'enchaînement des travaux, le temps de passage est le même, mais il n'y a pas d'attente entre deux passages. Il s'ensuit que *D* = 60 / 1.8 = 33, et η = 33 / 60 = 0.55.

1.1.2. Question B

1.1.2.1. Question B.1

Le débit maximum est limité à 60 travaux à l'heure, par l'unité de traitement. Le débit maximum d'entrée des cartes, est de 60 / 0.3 = 200 travaux à l'heure. Le débit maximum d'impression est de 60 / 0.5 = 120 travaux à l'heure. D'où $D = 60$, et $\eta = 60 / 60 = 1.0$. Ces valeurs ne pourront en fait être atteintes que si la planification est correcte.

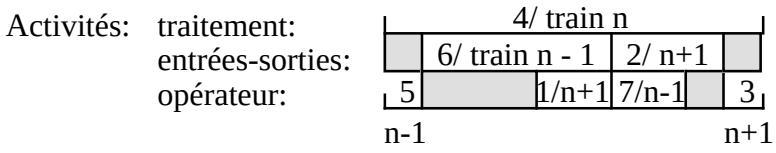
1.1.2.2. Question B.2

La planification doit tenir compte du fait que l'opérateur ne peut faire qu'une chose à la fois, comme d'ailleurs l'ordinateur d'entrées sorties. Par ailleurs elle doit respecter l'ordre suivant pour un train:

1 préparation du train de cartes 10 minutes
2 lecture de 50 travaux. 15 minutes
3 transfert de la bande vers l'ordinateur central, 5 minutes
4 exécution de ces 50 travaux, 50 minutes
5 transfert de la bande d'impression 5 minutes
6 impression des 50 travaux 25 minutes
7 distribution des résultats 10 minutes

L'opération 4 du train n est effectuée en même temps que les opérations suivantes, dans l'ordre:

5 étape n - 1
6 étape n - 1 et 1 de l'étape n + 1
7 étape n - 1 et 2 étape n + 1
3 étape n + 1



Il s'ensuit que le temps d'attente moyen est:

10 + 15 + 5 + 50 + 5 + 25 + 10 = 120 soit 2 heures.

1.1.3. Question C

La lecture du travail n + 1 demande 0.3 mn, ce qui est inférieur au temps de traitement, qui est de 1 mn. L'impression du travail n - 1 demande 0.5 mn, ce qui est encore inférieur au temps de traitement. D'où $D = 60$, et $\eta = 1.0$.

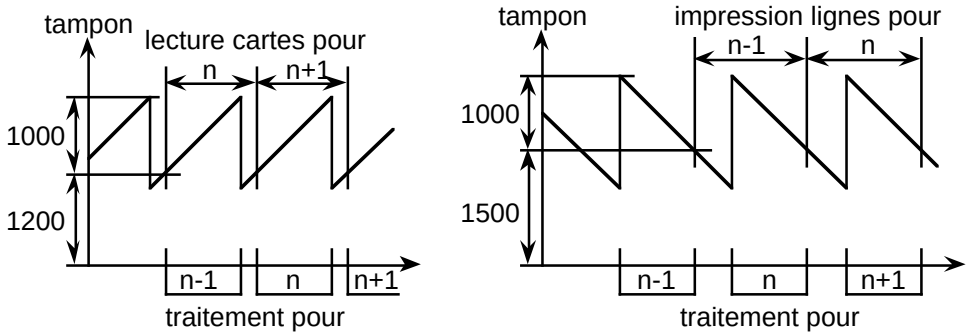
Si on a 1200 cartes à lire, le temps de lecture du travail n + 1 devient de 1.2mn, ce qui est cette fois supérieur au temps de traitement. De même, si on a 1500 lignes, le temps d'écriture du travail n - 1 devient 1.5mn, ce qui est également supérieur au temps de traitement. Le plus grand des 3 est le temps d'impression. On a donc $D = 60 / 1.5 = 40$, et $\eta = 40 / 60 = 0.67$.

1.1.4. Question D

1.1.4.1. Question D.1

L'unité centrale doit attendre d'avoir un travail complet pour s'exécuter. Comme c'est l'unité la plus rapide (cf. C), il faut faire en sorte que le lecteur travaille à sa vitesse maximum, sans être ralenti par l'unité centrale, qui n'aurait pas vidé le tampon de lecture. Comme on ne peut préjuger de la façon dont elle vide ce tampon, on doit considérer qu'elle libère l'espace occupé par les cartes d'un travail lorsque ce travail est terminé. Pendant ce temps, le lecteur peut lire 1000 cartes, il faut donc un tampon d'au moins 1000 + 1200 = 2200 cartes, soit 176000 octets. Pour que l'unité centrale ne soit pas freinée par l'impression, il faut qu'il y ait toujours au moins un tampon de 1500 lignes

disponible au début d'un travail, c'est à dire toutes les 1.2 minutes, puisque c'est le temps mis pour lire un travail. Dans ce cas, puisque l'impression n'a pu libérer que 1200 lignes du tampon pendant ce temps, il faut que le tampon augmente de 300 lignes toutes les 1.2 minutes! Il faut un tampon infini. Le débit cependant reste limité au débit de l'imprimante.



Si on essaie maintenant de faire en sorte que seule l'impression ne soit pas ralentie, il faut qu'à la fin de l'impression du travail n, le traitement du travail n+1 soit terminé. Il faut donc qu'il ait commencé au plus tard 1 minute avant, et qu'il n'ait pas été ralenti par manque de place dans le tampon d'impression. Au moment du début d'un travail, il faut donc qu'il y ait 1000 + 1500 lignes disponibles dans le tampon, qui doit donc faire 250000 octets. Notons que dans ce cas, le tampon de lecture peut être diminué, puisque l'unité centrale étant inactive pendant 0.5 minutes, il faut simplement que 700 cartes aient été lues à ce moment, c'est-à-dire lorsque le traitement libère les 1200 cartes dont il avait besoin. Le tampon de lecture doit alors faire 1900 cartes, soit 152000 octets.

1.1.4.2. Question D.2

Avec un rythme maximal d'arrivée des travaux en entrée, c'est-à-dire, saturant le lecteur de cartes, et donc un tampon d'entrée de 176000 octets, l'unité centrale pourra effectuer un travail toutes les 1.2 minutes, tant que le tampon de sortie ne sera pas saturé. A chaque fois, 300 lignes supplémentaires resteront à imprimer. Comme le tampon de sortie peut contenir 20000 lignes, après 66 travaux, il sera saturé. On peut donc en conclure que pendant les 79 premières minutes, le rendement $\eta = 1 / 1.2 = 0.83$. Il passera alors à $1 / 1.5 = 0.67$ tant que le rythme d'arrivée se maintiendra à une moyenne de 40 par heure. Il restera alors 20 minutes d'impression, pendant lesquelles le rendement sera nul.

Il faut noter que ceci montre que ces tampons ont un second rôle qui est d'amortir les variations du rythme d'arrivée des travaux, ainsi que les variations de leur comportement.

1.2. Amorçage d'un ordinateur

A- Sur un ordinateur, lors de sa mise en route, le microprogramme exécute la séquence suivante:

- lecture d'un numéro de périphérique sur le panneau de contrôle,
- lecture d'un bloc de 1024 octets depuis ce périphérique, vers l'adresse hexadécimale 400 en mémoire, (secteur 0, piste 0, face 0 s'il s'agit d'un disque),
- exécution de l'instruction située à l'adresse hexadécimale 400.

Proposer un schéma pour l'amorce programmée qui figure dans un tel bloc pour un disque.

Indications: L'amorce doit lire sur disque des informations à charger, assurer le chargement et le lancement du programme ainsi chargé en mémoire, à partir d'une table d'implantation décrivant l'organisation.

B- Les compatibles PC sont équipés de microprocesseurs de la famille 8088/286. Lors de la mise sous tension, ces microprocesseurs forcent le compteur ordinal à une valeur prédéfinie (0FFFF0 en hexadécimal), et exécutent donc l'instruction située à cet endroit en mémoire.

B.1- Expliquer comment, à votre avis, les constructeurs réussissent à obtenir que lors de la mise en route par l'utilisateur, le compatible PC charge automatiquement le système MS-DOS depuis une disquette ou depuis le disque dur.

B.2- La solution adoptée pour résoudre B.1, ne permettrait-elle pas d'éviter de charger MS-DOS. Quel est l'intérêt alors de ce chargement?

Solution de l’exercice 1.2

1.2.1. Question A

L’amorce programmée doit contenir les informations suivantes:

1) La localisation sur disque des informations à charger, éventuellement complétées des adresses en mémoire centrale où ces informations doivent être placées:

- numéros de piste, face, secteur de début de zone sur disque,
- nombre de secteurs de la zone
- adresse mémoire centrale où la zone doit être chargée.

2) L'adresse mémoire centrale de la première instruction à exécuter du programme ainsi chargé. On peut imaginer que la dernière entrée de la table est distinguée par un nombre de secteurs nul, et que dans ce cas l'adresse mémoire correspond à cette adresse de lancement du programme.

Le programme amorce se décompose de la façon suivante:

```
initialisation des registres du processeur
pour toutes les entrées de la table faire
     si nombre de secteurs non nul alors lecture sur disque
     sinon saut à adresse mémoire
     finsi
fait
```

Il faut noter que ce programme doit être court, puisque l'ensemble doit tenir sur un bloc de 1Ko. Par ailleurs on ne dispose alors d'aucune fonctionnalité. En particulier la procédure de lecture en général fera un traitement d'erreur rudimentaire, par exemple en envoyant sur la console un message court, et en reprenant le programme amorce au début. Remarquons que ce programme amorce peut être utilisé pour charger non pas le programme final (système par exemple), mais pour charger un second programme d'amorçage plus élaboré, et donc de taille plus importante, qui assurera lui-même le chargement du programme système.

1.2.2. Question B

1.2.2.1. Question B.1

Puisque le microprocesseur exécute une instruction située à une adresse donnée en mémoire, il faut que cette instruction soit déjà dans la mémoire centrale lors de la mise en route. La partie de mémoire qui la contient doit être non volatile, c'est-à-dire doit persister à une coupure de courant, et ne doit pas pouvoir être modifiée pour que l'usager retrouve toujours son ordinateur personnel dans le même état lorsqu'il met le courant. C'est pourquoi on utilise une mémoire morte “ROM” (ou *Read Only Memory*). Dans cette mémoire est inscrit un programme d'amorçage équivalent à ce qui est microprogrammé sur les gros ordinateurs. Cependant ce programme peut être déjà plus performant, comme par exemple, en recherchant sur un ensemble de périphériques, comme les lecteurs de disquettes ou les disques durs, le premier qui contient une amorce programmée (secteur de boot dans la terminologie anglaise) qui chargera alors le système MS-DOS lui-même.

Le programme d'amorce en ROM pourrait être le suivant:

```
pour tous les périphériques faire
   lire un bloc;
   si bloc lu correctement alors
          si bloc d'amorce alors
               exécuter le bloc;       { amorce programmée }
          finsi;
   finsi;
fait;
```

1.2.2.2. Question B.2

Il serait tout à fait possible de mettre MS-DOS directement dans cette mémoire ROM. Il suffirait que sa taille soit suffisante. C'est d'ailleurs ce que fait Atari sur les ordinateurs personnels de la famille ST qui contiennent le “TOS” en mémoire ROM. Cependant, cela ne permet pas de changer facilement de version, et de faire évoluer ainsi le système lui-même, puisqu'il faut alors changer la mémoire morte. En général, ces machines compensent cet inconvénient en conservant la possibilité d'amorcer l'ordinateur à partir d'un périphérique. Évidemment on perd alors l'intérêt du système en ROM.

1.3. Petit système temps-réel

[Krakowiak (85)] Un ordinateur est utilisé pour le prélèvement de mesures. Il exécute périodiquement (avec une période T) un cycle de mesure:

- prélèvement de mesure sur les capteurs (durée: *tmes*)
- traitement des mesures (durée: *tcalc*)
- vidage sur disque des résultats (durée: *tvid*)

Les trois opérations ci-dessus doivent être exécutées en séquence pour chaque cycle. On dispose d'une horloge qui provoque une interruption au passage du compteur à zéro. On demande d'écrire les programmes de ce système (programme principal, traitement des interruptions) dans les quatre cas suivants, en formulant dans chaque cas, s'il y a lieu, la condition de possibilité du traitement en fonction des durées des différentes opérations.

A- Les capteurs et le contrôleur du disque sont commandés par l'unité centrale. L'opération de prélèvement est lancée par l'unité centrale qui doit ensuite prendre les données au fur et à mesure que les capteurs successifs les fournissent, au rythme imposé par ces capteurs. L'écriture sur disque est lancée par l'unité centrale qui doit ensuite fournir les données au fur et à mesure des demandes du disque. Dans chaque cas, l'ordinateur est suffisamment rapide pour satisfaire le rythme imposé par les capteurs ou le disque, mais ne peut faire autre chose en même temps. Les cycles successifs de mesure doivent être exécutés en séquence, et un travail de fond est exécuté pendant les temps morts.

B- Les capteurs sont toujours commandés par l'unité centrale comme en A, mais le disque est commandé par accès direct mémoire. L'ordinateur lance l'opération d'écriture sur disque en précisant l'adresse mémoire où se trouvent les données à écrire. Il peut par ailleurs tester l'état du disque pour savoir si l'opération est terminée. Le vidage du cycle i peut donc s'exécuter en parallèle avec le prélèvement et le traitement du cycle i + 1.

C- Les capteurs sont commandés eux aussi par accès direct mémoire indépendant de celui du disque. L'ordinateur lance l'opération de prélèvement en précisant l'adresse mémoire où doivent être mémorisés les résultats de mesure. Il peut par ailleurs tester l'état du prélèvement pour savoir si l'opération est terminée. Le prélèvement du cycle i peut donc être simultané au traitement du cycle i - 1 et au vidage du cycle i - 2.

D- La fin de l'opération disque, comme la fin de l'opération de prélèvement, déclenchent une interruption. Comment organisez-vous l'ensemble?

Solution de l’exercice 1.3

1.3.1. Question A

Le programme principal est un travail de fond quelconque, indépendant des mesures. A chaque interruption d'horloge, le programme suivant est lancé:

```
sauver état processeur (travail de fond)
lancer la commande de prélèvement
tant que toutes les mesures ne sont pas obtenues faire
      tant que mesure indisponible faire fait;   { attente }
      prendre la mesure et mettre en mémoire
fait;
traiter les mesures
lancer la commande d'écriture disque
tant que tout n'est pas écrit faire
      tant que le contrôleur n'est pas prêt faire fait; { attente }
      transférer la donnée suivante de la mémoire vers le contrôleur
fait;
restituer l'état du processeur (travail de fond)
```

L'ensemble du travail se déroulant séquentiellement, il est nécessaire que l'on ait:

$$tmes + tcalc + tvid < T$$

Si cette condition n'est pas vérifiée, une interruption d'horloge surviendra alors que le traitement correspondant à la précédente n'est pas terminé, ce qui pose des problèmes dits de réentrance. On peut les éviter en “masquant” les interruptions d'horloge au début du programme précédent, et en les démasquant à la fin. Le non respect de la condition précédente entraînera simplement la perte de certaines interruptions, sans perturber le déroulement du programme lui-même.

1.3.2. Question B

Dans ce cas, dès le lancement de l'opération de vidage sur disque, il est possible de restaurer l'état du processeur, l'exécution de l'opération proprement dite se déroulant alors en parallèle. Lors de l'étape suivante, avant de lancer la nouvelle opération de vidage, il faut s'assurer que la précédente est bien terminée. Le programme principal reste un travail de fond quelconque. Le programme sous interruption d'horloge est donné ci-après. Le bon fonctionnement est obtenu cette fois sous deux conditions:

$$tmes + tcalc < T$$

$$tvid < T$$

```
masquer l'interruption d'horloge
sauver état processeur (travail de fond)
lancer la commande de prélèvement
tant que toutes les mesures ne sont pas obtenues faire
      tant que mesure indisponible faire fait;   { attente }
      prendre la mesure et mettre en mémoire
fait;
traiter les mesures
tant que contrôleur disque occupé faire fait;    { attente }
lancer la commande d'écriture disque
restituer l'état du processeur (travail de fond)
démasquer l'interruption d'horloge
```

1.3.3. Question C

Cette fois, pour obtenir un prélèvement régulier des mesures, la commande de prélèvement doit être lancée à chaque interruption d'horloge, mais il ne faut pas attendre qu'elle soit terminée. Le travail de fond consiste à attendre qu'un prélèvement soit terminé pour effectuer le traitement sur les mesures correspondantes, puis écrire sur disque les résultats. Pour que l'on puisse éventuellement relancer un prélèvement le plus tôt possible sans ennui, il faut que le traitement lui-même soit précédé d'un rangement dans une zone de travail des mesures, libérant ainsi la zone de prélèvement pour le cycle suivant. La “synchronisation” entre le prélèvement et le traitement est obtenue par masquage de l'interruption d'horloge.

traitement sur interruption d'horloge:

```
masquer l'interruption d'horloge
sauver état processeur (travail de fond)
lancer la commande de prélèvement
restituer l'état du processeur (travail de fond)
```

travail de fond:

```
tant que vrai faire                { boucle infinie }
      tant que prélèvement non terminé faire fait;    { attente }
      prendre les mesures et les ranger en zone de travail
      démasquer l'interruption d'horloge
      traiter les mesures
     tant que contrôleur disque occupé faire fait;     { attente }
      lancer la commande d'écriture disque
fait;
```

Le bon fonctionnement est obtenu cette fois sous trois conditions:

$$tmes < T \qquad tcalc < T \qquad tvid < T$$

1.3.4. Question D

On peut alors éviter les attentes actives qui existaient dans la solution précédente. Le déroulement séquentiel des opérations implique que l'interruption signalant la fin du prélèvement entraîne l'exécution du traitement, sous réserve que le traitement des mesures précédentes soit terminé. L'écriture sur disque des résultats ne peut avoir lieu que si l'écriture précédente est terminée. L'interruption de fin d'écriture doit donc déclencher l'écriture suivante, sous réserve que le traitement correspondant soit terminé.

traitement sur interruption d'horloge :

```
masquer l'interruption d'horloge
sauver état processeur (travail de fond)
lancer la commande de prélèvement
restituer l'état du processeur (travail de fond)
```

traitement sur interruption de fin de prélèvement:

```
masquer l'interruption de prélèvement
sauver état processeur (travail de fond)
prendre les mesures et les ranger en zone de travail
démasquer l'interruption d'horloge
traiter les mesures
démasquer l'interruption disque
restituer l'état du processeur (travail de fond)
```

traitement sur interruption de fin d'écriture disque:

```
masquer l'interruption disque
sauver état processeur (travail de fond)
ranger les résultats dans le tampon d'écriture
lancer la commande d'écriture disque
démasquer l'interruption de prélèvement
restituer l'état du processeur (travail de fond)
```

initialisation:

```
masquer l'interruption disque
lancer la commande d'écriture disque           { écriture bidon }
démasquer l'interruption d'horloge
démasquer l'interruption de prélèvement
```

L'intérêt essentiel de cette solution est évidemment de récupérer les temps d'inactivité du processeur pour un travail de fond indépendant et quelconque. Les contraintes temporelles sont évidemment les mêmes que ci-dessus. Le respect de ces contraintes nécessite souvent de dimensionner le processeur de façon à être en deçà des limites. Cette solution permet alors de récupérer la puissance de calcul inutilisée.

1.4. Petit système d'exploitation

[Krakowiak (85)] *Ce problème est consacré à la réalisation d'un petit système doté d'un moniteur d'enchaînement de travaux analogue à celui étudié dans le problème 1. Le contexte de définition du travail est très simplifié, par rapport à la réalité, pour permettre d'aborder facilement quelques points de cette réalisation. L'utilisation de cartes peut paraître archaïque de nos jours, mais permet ici de prendre en compte simplement l'asynchronisme des périphériques.*

L'ordinateur comporte, outre le processeur et sa mémoire, un lecteur de cartes, une imprimante et un terminal pour l'opérateur. Un travail est constitué d'une paquet de cartes qui a la structure suivante:

- une carte *JOB <nom de travail> <tmax> <nlmax>,
- la suite des cartes contenant le texte source du programme,
- une carte *DATA,
- la suite des cartes contenant les données,
- une carte *FIN.

Les programmes sont écrits dans un langage unique. Ils sont compilés en binaire absolu en mémoire, et sont exécutés immédiatement. Le système d'exploitation, compilateur compris, est entièrement résident en mémoire. Il connaît l'adresse de lancement du compilateur, ainsi que l'adresse de lancement d'un programme compilé, qui est toujours la même.

Les cartes de commandes sont caractérisées par un "*" en première colonne. Une carte comporte 80 caractères. Les paramètres figurant sur la carte **JOB* spécifient:

- le nom utilisé pour identifier le travail, et qui sera imprimé en tête des listes de sorties,
- une durée maximale d'utilisation de l'unité centrale,
- un nombre maximal de lignes à imprimer.

Un train de travaux est constitué d'une suite de travaux, terminée par une carte **FIN* supplémentaire (le train se termine donc par deux cartes **FIN* consécutives). L'opérateur prépare le train de travaux et lance son exécution. Il communique avec le système à l'aide de son terminal, et dispose de trois commandes, qu'il active en frappant le premier caractère de la commande:

LANCER : lancer l'exécution d'un train de travaux prêt dans le lecteur de cartes,
STOP : interrompre l'exécution immédiatement (arrêt d'urgence),
TERMINER : interrompre l'exécution après la fin du travail en cours.

Le système affiche sur le terminal de l'opérateur le contenu des cartes **JOB* au fur et à mesure de leur lecture.

Le système assure une fonction élémentaire de comptabilité: il imprime à la fin de chaque travail le temps total d'unité centrale passé, le nombre de cartes lues, le nombre de lignes imprimées.

Le système fournit aux utilisateurs trois appels au superviseur:

SVC lire : lire une carte,
SVC écrire : imprimer une ligne,
SVC fin : fin d'exécution, retour au système.

Ces instructions *SVC* ne sont pas directement écrites par l'utilisateur mais insérées par le compilateur dans le programme objet lorsque nécessaire.

Le pilotage des périphériques est assuré par le processeur. Chaque périphérique envoie une interruption à la fin du transfert d'une unité (une carte pour le lecteur de cartes, une ligne complète pour l'imprimante, un caractère pour le terminal de l'opérateur); un code identifie le périphérique émetteur.

Les opérations sur les périphériques sont réalisées de la façon suivante:

- lecteur de cartes ou imprimante, on dispose des opérations:

  - `lancer_opération (périphérique, tampon)` le périphérique détermine le sens du transfert, et la longueur. Le transfert est réalisé en DMA. À la fin du transfert, le périphérique déclenche une interruption.
  - `opération_correcte (périphérique)` le périphérique retourne un booléen qui est `vrai` si la dernière opération sur ce périphérique s'est déroulée correctement.
- terminal opérateur, on dispose des opérations suivantes:
  - Lorsqu'un caractère est reçu en provenance du terminal, une interruption `entrée_terminal` est émise vers le processeur.
  - `lire_terminal` demande à l'interface du terminal de délivrer le dernier caractère reçu du terminal.
  - `écrire_terminal (caractère)` demande à l'interface du terminal d'envoyer le `caractère` au terminal.
  - Lorsque l'envoi d'un caractère vers le terminal est achevé, une interruption `sortie_terminal` est émise vers le processeur.

Note: les interruptions de fin d'opération sont toujours émises au bout d'un temps fini, inférieur à 0.5 seconde, même en cas d'erreur.

Une horloge est disponible, avec interruption au passage à zéro. Le processeur a un mode maître et un mode esclave, les interruptions d'horloge n'étant acceptées qu'en mode esclave. La zone mémoire utilisée par le système est protégée; une tentative d'accès dans cette zone par un programme en mode esclave provoque un déroutement.

Toute interruption, appel au superviseur ou déroutement entraîne le passage en mode maître. Le retour normal au programme remet le mode esclave. Par ailleurs, deux opérations sont disponibles (en mode maître) pour lancer ou arrêter un programme ou le compilateur, et qui peuvent se décrire de la façon suivante:

```
type t_code = (       { code détermine la façon dont le programme est arrêté: }
                NORMAL,           { par appel SVC fin }
                ERR_TEMPS,             { par durée maximum atteinte }
                ERR_LIGNES,       { par nombre de lignes atteint }
                ERR_TRAP,              { par déroutement }
                ERR_STOP )             { par décision de l'opérateur }
opération lancer_prog (e : adresse) : t_code;
début
     sauver_état_moniteur;
     aller à e;                        { passage en mode esclave }
fin;
opération arrêt_prog (code : t_code);
début
     restaurer_état_moniteur;       { restitue l'adresse d'appel de lancer_prog }
     retourner (code);
fin;
```

A- Détailler les procédures, internes au système d'exploitation, qui gèrent les opérations sur chacun des périphériques.

A.1- `lire_carte (var tampon : chaîne [80])` lance la lecture d'une carte dans le `tampon`, attend la fin de l'opération et la relance éventuellement en cas d'erreur.

A.2- `imprimer (tampon : chaîne [132])` attend que la ligne précédente soit imprimée sans erreur et lance l'impression d'une nouvelle ligne depuis le `tampon`.

A.3- `envoi_term (tampon : chaîne [], long : entier)` assure l'envoi de la `chaîne` constituée de `long` caractères au terminal.

B- En utilisant les procédures ci-dessus, détailler les appels au superviseur proposés (SVC). On veillera à garantir qu'une erreur dans un programme d'utilisateur ne mette pas en jeu l'intégrité du système, et on assurera la comptabilité minimale demandée.

C- Les commandes LANCER et TERMINER de l'opérateur doivent être effectivement prises en compte uniquement lorsque le moniteur est en train de s'exécuter. Ceci se fera simplement dans l'algorithme du moniteur par consultation périodique de la présence d'une commande. Par contre la

commande STOP doit interrompre le programme (ou le compilateur) s'il est en cours d'exécution, sans perturber le fonctionnement du moniteur si c'est celui-ci qui s'exécute.

C.1- Proposer une solution pour cette prise en compte de la commande STOP lorsque le programme est en cours.

Idée: faire faire la consultation lors des appels au superviseur, des interruptions de fin de transfert ou lors des interruptions d'horloge.

C.2- Détailler le traitement de toutes les interruptions, en utilisant éventuellement ce qui a été défini, ainsi que le déroutement et les appels au superviseur (SVC).

D- Définir le traitement du moniteur d'enchaînement des travaux, dans ce contexte.

Solution de l'exercice 1.4

1.4.1. Question A

1.4.1.1. Question A.1

Il faut lancer l'opération de lecture d'une carte, et attendre l'interruption qui signale la fin de cette opération asynchrone. En fait, on peut mettre à `vrai` un booléen avant le lancement de l'opération, l'interruption entraînant sa remise à `faux`. On peut alors la relancer si elle est incorrecte.

```
var lecture_en_cours : booléen;
procédure lire_carte (var tampon : chaîne [80]);
début
     répéter
          lecture_en_cours := vrai;      { pour savoir quand c'est terminé }
          lancer_opération (lecteur, tampon);        { opération asynchrone }
          tant que lecture_en_cours faire fait; { attente }
     jusqu'à  opération_correcte (lecteur);
fin;
procédure interruption_carte;
début
     lecture_en_cours := faux;             { indication de fin d'opération }
fin;
```

1.4.1.2. Question A.2

Contrairement à la lecture de carte, où le programme attend le contenu de la carte pour continuer, l'impression d'une ligne peut se faire de façon autonome, pourvu que son exécution soit garantie. Cette garantie ne peut être obtenue que si la ligne à imprimer est mémorisée pour pouvoir relancer son impression si l'opération n'a pas été correcte. Il s'ensuit que cette relance doit être alors assurée par le sous-programme d'interruption lui-même. Par ailleurs une demande d'impression doit attendre que la précédente soit terminée. Ici encore un booléen sera mis à `vrai` lors de la demande, et remis à `faux` lorsqu'elle sera exécutée.

```
var impression_en_cours : booléen;
   tamp_loc : chaîne[132];
procédure imprimer (tampon : chaîne [132]);
début
     tant que impression_en_cours faire fait;     { attente fin de la précédente
}
     tamp_loc := tampon;
     impression_en_cours := vrai;           { indication opération en cours }
     lancer_opération (imprimante, tamp_loc);         { sans attente de fin }
fin;
procédure interruption_imprimante;
début
     si opération_correcte (imprimante) alors
          impression_en_cours := faux;         { autorisation de la suivante }
     sinon
          lancer_opération (imprimante, tamp_loc);    { sans attente de fin }
     finsi;
fin;
```

1.4.1.3. Question A.3

Comme pour l'impression d'une ligne, l'envoi au terminal d'un ensemble de caractères, peut se faire en même temps que la poursuite du travail. Il suffit de mettre les caractères à envoyer dans un tampon local. Notons qu'ici, la difficulté n'est pas liée aux erreurs éventuelles, puisqu'elles ne sont pas détectées, mais au fait que le transfert des caractères est programmé, c'est-à-dire, que le logiciel doit les envoyer un par un. Ce transfert peut être dirigé par l'interruption `sortie_terminal`. En particulier, c'est l'interruption de fin de transfert du dernier caractère qui libère le terminal, et autorise la prise en compte d'une nouvelle demande.

```
var tamp_term_loc : chaîne [80];      { caractères à émettre }
   long_loc, ind_term : entier;
   term_occupé : booléen;             { il y en a encore }
procédure envoi_term (tampon : chaîne [], long : entier);
début
     tant que term_occupé faire fait;  { attente de la fin du précédent }
     tamp_term_loc := tampon;
     long_loc := long;
     term_occupé := vrai;                    { initialisation }
     ind_term := 1;
     écrire_terminal ( tamp_term_loc [0] );      { sortie du premier caractère }
fin;                                    { les suivants sur interruption }
procédure interruption_sortie_terminal;
début
     si ind_term < long_loc alors             { non fin de sortie en cours }
          écrire_terminal ( tamp_term_loc [ind_term] );
          ind_term := ind_term + 1;
     sinon
          term_occupé := faux;                { fin de l'envoi au terminal }
   finsi;
fin;
```

1.4.2. Question B

Lors d'un appel au superviseur, il faut d'abord déterminer duquel il s'agit, et appeler la procédure correspondante.

```
procédure appel_SVC (nature : (lire, écrire, fin) );
début
     cas nature dans
     lire: lecture_contrôlée ( tamp_utilisateur, code_retour );
     écrire: impression_contrôlée ( ligne_utilisateur );
     fin: arrêt_prog ( NORMAL );
     fincas;
fin;
```

La lecture des cartes doit être contrôlée de façon à éviter de délivrer à un programme une carte de commande. S'il n'y a plus de carte de données à lire il faut lui retourner le code de fin de fichier. Cela implique qu'il est possible de lire une carte de commande lors d'une demande du programme, et qu'il faut la mémoriser jusqu'à la demande de cette carte par le moniteur. Par ailleurs, pour faire la comptabilité, il faut compter les cartes effectivement lues pour le JOB en cours.

```
var commande_lue : booléen;  { une carte de commande est présente,
                          initialisé à faux par le moniteur d'enchaînement }
   tampon_carte : chaîne [80];        { tampon de carte de commande }
   nb_cartes : entier;                { nombre de cartes lues pour ce Job }
procédure lecture_contrôlée ( var t : chaîne[80]; var code : entier );
début
     si non commande_lue alors
          lire_carte (tampon_carte) ;
          nb_cartes := nb_cartes + 1;
          commande_lue := (tampon_carte [0] = '*');
     finsi;
     si commande_lue alors code := fin_fichier;
     sinon t := tampon_carte ; code := OK; finsi;
fin;
```

L'impression des lignes doit aussi être contrôlée de façon à comptabiliser leur nombre et arrêter le programme s'il dépasse la limite qu'il a annoncée. Notons que nous ne compterons pas les lignes imprimées par le moniteur.

```
var nb_lignes, nlmax : entier;
procédure impression_contrôlée ( ligne : chaîne[132] );
début
     si nb_ligne ≥ nlmax alors arrêt_prog (ERR_LIGNES) finsi;
     nb_ligne := nb_ligne + 1 ;
     imprimer (ligne);
fin;
```

1.4.3. Question C

1.4.3.1. Question C.1

Une commande de l'opérateur consiste en l'un des trois caractères 'L', 'S' ou 'T'. Le sous-programme d'interruption correspondant à `entrée_terminal` peut donc consister simplement en la lecture du caractère fourni, et en sa mémorisation dans une variable globale s'il est l'un des trois. Il suffit alors de comparer sa valeur à 'S' en différents endroits pour garantir sa prise en compte effective:

- interruption du lecteur de cartes: il faut arrêter la lecture, même si elle est avec erreur.
- interruption de l'imprimante: il faut remettre à `faux` le booléen `impression_en_cours` sans consulter le résultat de l'opération, de façon à libérer l'imprimante, même s'il y a erreur.
- interruption de sortie de terminal: il n'est pas nécessaire de faire quelque chose, puisqu'il n'y a pas de danger d'avoir une boucle infinie.
- interruption d'horloge: comme elles ne peuvent survenir que si le processeur est en mode esclave, on est certain que le programme ou le compilateur est actif à ce moment. Il peut donc être arrêté pour donner le contrôle au moniteur.
- appels au superviseur: comme ils ne sont utilisés que par le programme ou le compilateur, on peut également les arrêter pour donner le contrôle au moniteur. Le test doit ici être fait avant l'exécution des opérations de lecture ou d'écriture, ainsi qu'après, car ces opérations peuvent être longues, à cause des boucles en cas d'erreur. Par contre il est inutile de le faire en cas de fin d'exécution.

1.4.3.2. Question C.2

Le traitement des interruptions est défini comme suit:

```
var k_term : caractère;               { dernier caractère de commande reçu }
procédure interruption_entrée_terminal;
var c : caractère;
début
     c := lire_terminal;
     si c = 'L' ou c = 'S' ou c = 'T' alors
          k_term := c ;               { simple mémorisation }
     finsi;
fin;

procédure déroutement;
début
     arrêt_prog ( ERR_TRAP );
fin;

var temps, tmax : entier;
procédure interruption_horloge;
début
     sauver_contexte;
     temps := temps + 1;
     si temps = tmax alors arrêt_prog (ERR_TEMPS) finsi;
     si k_term = 'S' alors arrêt_prog (ERR_STOP) finsi;
     restitution_contexte;
fin;
```

```
procédure interruption_périphérique;
début
     sauver_contexte;
     cas demandeur dans
     carte: interruption_carte;
     imprimante: interruption_imprimante;
     entrée_terminal: interruption_entrée_terminal;
     sortie_terminal: interruption_sortie_terminal;
     fincas;
     restitution_contexte;
fin;
```

Les procédures de lecture de cartes doivent être modifiées comme suit:

```
var lecture_en_cours, fin_lecture : booléen;
procédure lire_carte (var tampon : chaîne [80]);
début
     répéter
          lecture_en_cours := vrai;      { pour savoir quand c'est terminé }
          lancer_opération (lecteur, tampon);
          tant que lecture_en_cours faire fait; { attente }
     jusqu'à  fin_lecture;
fin;

procédure interruption_carte;
début
     fin_lecture := (k_term = 'S' ou opération_correcte (lecteur));
     lecture_en_cours := faux;         { indication de fin d'opération }
fin;
```

Les procédures d'impression sont modifiées comme suit (seule la procédure d'interruption est en fait modifiée):

```
var impression_en_cours : booléen;
   tamp_loc : chaîne[132];

procédure imprimer (tampon : chaîne [132]);
début
     tant que impression_en_cours faire fait;     { attente fin de la précédente
}
     tamp_loc := tampon;
     impression_en_cours := vrai;           { indication opération en cours }
     lancer_opération (imprimante, tamp_loc);         { sans attente de fin }
fin;

procédure interruption_imprimante;
début
     si k_term = 'S' ou opération_correcte (imprimante) alors
          impression_en_cours := faux;         { autorisation de la suivante }
     sinon
          lancer_opération (imprimante, tamp_loc);    { sans attente de fin }
     finsi;
fin;
```

Le traitement des appels au superviseur doit être modifié comme suit:

```
procédure appel_SVC (nature : (lire, écrire, fin) );
début
     si nature = fin alors arrêt_prog ( NORMAL ); finsi;
     si k_term = 'S' alors arrêt_prog ( ERR_STOP ); finsi;
     cas nature dans
     lire: lecture_contrôlée ( tamp_utilisateur, code_retour );
     écrire: impression_contrôlée ( ligne_utilisateur );
     fincas;
     si k_term = 'S' alors arrêt_prog ( ERR_STOP ); finsi;
fin;
```

1.4.4. Question D

Le moniteur d'enchaînement des travaux comporte d'abord une procédure de recherche de carte de commande. Comme cette procédure parcourt les cartes qui ne contiennent pas de '*' en première colonne, elle doit vérifier que l'opérateur ne demande pas l'arrêt immédiat du travail. Nous la

décomposons d'abord en une procédure de lecture d'une carte et de consultation, et une procédure de recherche proprement dite.

```
procédure lecture_une_carte;
début
     lire_carte (tampon_carte);
     commande_lue := (tampon_carte [0] = '*');
     si k_term = 'S' alors état := arrêt;
                       envoi_term ("STOP", 4);
     finsi;
fin;

procédure recherche_carte_commande ;
début
     tant que non commande_lue et état ≠ arrêt faire
          lecture_une_carte;
     fait;
fin;
```

Le moniteur d'enchaînement des travaux est donné page suivante. Il doit prendre en compte les commandes 'L' et 'T' de l'opérateur, et assurer le lancement du compilateur puis du programme sauf si la compilation ne s'est pas terminée normalement.

1.4.5. Remarques

Le système que nous venons de définir se compose de trois couches:

- couche haute: moniteur d'enchaînement, et contrôle des entrées-sorties demandées par les programmes utilisateurs.
- couche moyenne: procédures de gestion des périphériques, avec reprise en cas d'erreur,
- couche basse: le traitement des interruptions de périphériques et d'horloge, le traitement des déroutements.

Les procédures de gestion du lecteur de cartes et de l'imprimante sont réparties sur les trois couches. Les procédures de gestion du terminal sont réparties sur les deux couches basse et moyenne. Les SVC, les déroutements, le débordement du temps sont transmis à la couche haute, qui prend les décisions de poursuite ou d'arrêt du travail. Le déroulement d'un tel ensemble est essentiellement séquentiel, les interruptions étant en général le résultat d'une commande préalable. Cependant, ceci n'est pas le cas pour l'interruption d'horloge, et pour l'interruption dûe à la frappe d'un caractère au clavier. L'interruption d'horloge ne peut survenir que lorsque le compilateur ou le programme utilisateur sont en train de s'exécuter. La décision d'arrêter alors le programme est identique à l'appel au superviseur de fin de traitement. Par contre, la frappe d'un caractère au clavier peut survenir à tout moment, et doit être pris en compte en particulier lorsque le moniteur d'enchaînement est en train de travailler. L'idée a été de faire une gestion des entrées clavier rudimentaire, puisque le sous-programme d'interruption correspondant assure simplement la mémorisation du caractère frappé. Lorsque la machine est en mode esclave, (compilation ou exécution), le sous-programme d'interruption d'horloge et les appels au superviseur effectuent chacun la consultation nécessaire. Lorsque la machine est en mode maître, (moniteur d'enchaînement), c'est le moniteur lui-même qui assure cette consultation dans son algorithme. Dans tous les cas, les sous-programmes d'interruption des périphériques complètent la consultation pour éviter les blocages.

La présence d'interruptions dans un système entraîne un comportement non déterministe. La synchronisation entre les procédures des différentes couches est obtenue simplement ici par des variables communes, souvent des booléens. Ce n'est pas toujours possible ainsi.

Noter les différentes “attentes actives” du système, ce qui n'est pas trop gênant ici, car il n'y a rien d'autre à faire, mais qu'il faudrait traiter autrement avec la multiprogrammation.

```
programme moniteur_enchaînement ;
début
   état := arrêt;
   répéter
      tant que état = arrêt faire
               si k_term = 'L' alors état := normal; finsi;
      fait;                                          { attente opérateur }
      envoi_term ("LANCER", 6);
      commande_lue := faux;
      recherche_carte_commande;
      tant que état = normal faire
          cas tampon_carte[1..3] dans
          'JOB': début
                 envoi_term (tampon_carte);
                 imprimer (tampon_carte);
                 initialiser_comptabilité;
                 code_retour := lancer_prog (ad_compil);
                 si code_retour = NORMAL alors
                    recherche_carte_commande;
                    si état ≠ arrêt alors
                         commande_lue := ( tampon_carte[1..4] ≠ 'DATA' );
                         code_retour := lancer_prog (ad_prog);
                    finsi;
                 finsi;
                 imprimer_comptabilité ( code_retour );
                 si k_term = 'T' alors état := arrêt;
                                    envoi_term ("TERMINER", 8);
                 finsi;
               fin;
          'FIN': début
                 lecture_une_carte;
                 si état ≠ arrêt et tampon[0..3] = '*FIN' alors
                     état := arrêt;
                     envoi_term ("FIN", 3);
                 finsi;
               fin;
          autre: commande_lue := faux;
          fincas;
          recherche_carte_commande;
      fait;
   finrépéter;
fin;
```

2

Chaîne de production de programmes

2.1. Petit compilateur

Le but de cet exercice est l'étude des composantes d'un compilateur.

Nous définissons un petit langage simple, qui va nous servir d'exemple:

```
<instruction> ::= <identificateur> = <expression>
<expression> ::= <facteur> { + | - } <expression> | <facteur>
<facteur> ::= <terme> { * | / } <facteur> | <terme>
<terme> ::= <identificateur> | <nombre> | ( <expression> )
```

Les identificateurs sont des unités lexicales constituées soit d'une lettre unique, soit d'une lettre suivie d'un chiffre. Les nombres sont également des unités lexicales constituées de chiffres; on ne prend en compte que des entiers positifs ou nuls. Ces unités lexicales peuvent donc être décrites par les règles suivantes:

```
<identificateur> ::= <lettre> | <lettre> <chiffre>
<nombre> ::= <chiffre><nombre> | <chiffre>
```

Les espaces ne peuvent se trouver à l'intérieur d'une unité lexicale, et n'ont pas de signification. Une instruction doit se trouver toute entière sur une même ligne, et une ligne ne peut contenir qu'une seule instruction.

A- L'analyse lexicale consiste à reconnaître la suite des unités lexicales d'une ligne, et à fournir la suite des codes syntaxiques de ces unités lexicales. Elle doit donc fournir une suite d'entiers correspondant à ces codes, de la façon suivante:

- Pour un nombre, le code est la valeur de l'entier correspondant.
- Pour un identificateur, le code est l'entier négatif, compris entre -286 et -1, ayant pour valeur `- (numéro_de_lettre + 26 * (chiffre + 1))`.
- Pour un caractère spécial, le code associé est défini par la table suivante:

caractère	=	(	)	+	-	*	/	K_fin_ligne
code	-287	-288	-289	-290	-291	-292	-293	-294

A.1- Donner le résultat fourni par l'analyseur lexical pour les lignes suivantes:

```
A1  =  ( A * 100 ) + 2 * B
A1= 42 23 * C
```

A.2- Compléter la procédure d'analyse lexicale d'une ligne définie ci-dessous par la définition des procédures `début_ligne`, `cas_lettre`, `cas_chiffre`, `cas_code` et `fin_ligne`.

```
procédure analyse_lexicale (var tab: tableau [1..80] de entier);
var oper : tableau [1..8] de char := " =()+-*/";
début début_ligne;
       lire (c);
       tantque c ≠ K_fin_ligne faire
          si c dans ['A'..'Z'] alors cas_lettre;
          sinsi c dans ['0'..'9'] alors cas_chiffre;
          sinon j := 0;
               pour k := 1 jusqua 8 faire
                    si c = oper[k] alors j := k finsi;
               fait;
               cas_code (j);
          finsi
          lire (c);
       fait;
       fin_ligne;
fin;
```

B- L'analyse syntaxique consiste à vérifier la concordance de la suite des codes syntaxiques telle qu'elle résulte de l'analyse lexicale avec les règles de syntaxe du langage. Pour cela, on peut définir les fonctions `instruction`, `expression`, `facteur` et `terme`, à résultat booléen, vérifiant la concordance avec chacune des règles. Par exemple voici la fonction `instruction`:

```
fonction instruction (var i: entier): booléen;
début instruction := faux ;
       si tab[i] < 0 et tab[i] > -287 alors
          i := i + 1;              { ok, c'est un identificateur }
          si tab[i] = -287 alors
               i := i + 1;     { ok, bien suivi de = }
               instruction := expression (i);
          finsi
       finsi
fin;
```

La fonction d'analyse syntaxique peut alors s'écrire:

```
fonction analyse_syntaxique (tab: tableau [1..80] de entier): booléen;
var i : entier;
début i := 1;
       analyse_syntaxique := faux;
       si instruction (i) et alors tab[i] = -294 alors
          analyse_syntaxique := vrai; { ok, fin de l'instruction }
       finsi
fin;
```

B.1- Définir les fonctions `expression`, `facteur` et `terme`.

B.2.- Donner le résultat de la fonction `analyse_syntaxique` dans les deux cas suivants:

```
A1  =     ( A * 100 ) + 2 * B
A1= 42 23 * C
```

C- L'analyse sémantique consiste à déterminer la sémantique d'une phrase syntaxiquement correcte, et à en donner une autre forme, dans un langage plus proche de la machine.

C.1- Donner la transformation en chaîne postfixée de la ligne:

```
A1 = ( A * 100 ) + 2 * B
```

C.2- Donner les grandes lignes des algorithmes de transformation en chaîne postfixée. On pourra se baser sur la méthode utilisée en B.

Solution de l'exercice 2.1

2.1.1. Question A

2.1.1.1. Question A.1

Le résultat de l'analyse lexicale des phrases est le suivant:

`A1  = ( A * 100 ) + 2 * B`	-53, -287, -288, -1, -292, 100, -289, -290, 2, -292, -2, -294

`A1= 42 23 * C`	-53, -287, 42, 23, -292, -3, -294

2.1.1.2. Question A.2

Certaines unités lexicales sont représentées par plusieurs caractères (identificateurs et nombres). Deux indicateurs, `identif_en_cours` et `chiffre_en_cours` vont être utilisés pour mémoriser le fait que l'on a commencé ou non l'analyse lexicale d'une telle unité. L'indice `i` repère le dernier code produit.

```
procédure début_ligne;
début i := 0;   { initialisation }
       identif_en_cours := faux;
       chiffre_en_cours := faux;
fin;

procédure cas_lettre;
début i := i + 1;    { début d'identificateur }
       tab[i] := 'A' - c - 1;
       identif_en_cours := vrai;
       chiffre_en_cours := faux;
fin;

procédure cas_chiffre;
début si identif_en_cours alors      { fin d'identificateur }
         tab[i] := tab[i] + 26 * ('0' - c - 1);
         identif_en_cours := faux;
       sinsi chiffre_en_cours alors { suite de nombre }
         tab[i] := tab[i] * 10 + c - '0';
       sinon i := i + 1;             { début de nombre }
             tab[i] := c - '0';
             chiffre_en_cours := vrai;
       finsi
fin;

procédure cas_code (j : entier);
début si j = 0 alors imprimer ("erreur lexicale: ", c);
       sinsi j > 1 alors i := i + 1;     { ce n'est pas un espace }
                         tab[i] := -(285 + j);
       finsi;
       identif_en_cours := faux;
       chiffre_en_cours := faux;
fin;

procédure fin_ligne;
début i := i + 1;
       tab[i] := -294;
fin;
```

2.1.2. Question B

2.1.2.1. Question B.1

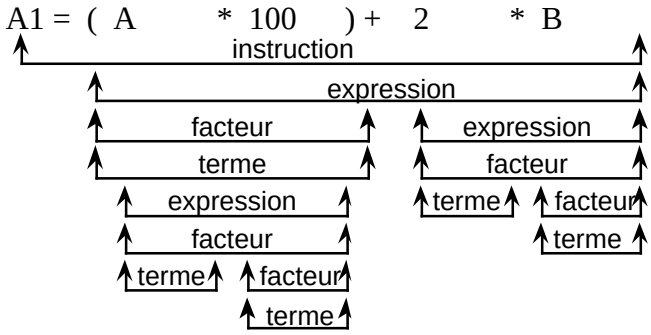
Les fonctions complémentaires d'analyse syntaxique se décrivent comme suit:

```
fonction expression (var i : entier): booléen;
début expression := faux;
       si facteur (i) alors
          si tab[i] = -290 ou tab[i] = -291 alors
               i := i + 1; expression := expression (i);
          sinon expression := vrai;
          finsi
       finsi;
fin;
```

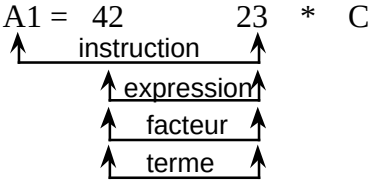
```
fonction facteur (var i : entier): booléen;
début facteur := faux;
       si terme (i) alors
          si tab[i] = -292 ou tab[i] = -293 alors
                i := i + 1; facteur := facteur (i);
          sinon facteur := vrai;
          finsi
       finsi;
fin;
fonction terme (var i : entier): booléen;
début terme := faux;
       si tab[i] >= -286 alors i := i + 1; terme := vrai;
       sinsi tab[i] = -288 alors
          i := i + 1;
          si expression (i) et alors tab[i] = -289 alors
               i := i + 1; terme := vrai;
          finsi
       finsi;
fin;
```

2.1.2.2. Question B.2

Le résultat de l'analyse syntaxique sur les exemples sont les suivants. Pour chaque unité syntaxique, la flèche gauche repère la position où l'on est dans la suite des codes syntaxiques au début de l'analyse de cette unité, et la flèche droite repère la position où l'on est à la fin de cette analyse.



C'est une instruction correcte.



C'est une instruction erronée.

2.1.3. Question C

2.1.3.1. Question C.1

La chaîne postfixée de `A1 = ( A * 100 ) + 2 * B` est:

-53, -1, 100, -292, 2, -2, -292, -290, -287

2.1.3.2. Question C.2

La structure en procédure de la question B peut être reprise, mais cette fois, on sait que la syntaxe est correcte. Il s'agit maintenant de transformer la suite d'unités syntaxiques en chaîne postfixée, en tenant compte de la façon dont doit être transformée chaque entité. Les procédures ont maintenant deux paramètres en **`var`**, l'un pour repérer la position dans la suite d'unités syntaxiques, l'autre pour repérer la position dans la chaîne produite.

```
procédure instruction (var i, j: entier);
début postfixée[j] := tab[i];        { identificateur }
       i := i + 2;              { saut du symbole = }
       j := j + 1;
       expression (i, j);
       postfixée[j] := -287;    { mise du symbole = }
       j := j + 1;
fin;
```

```
procédure expression (var i, j : entier);
var oper : entier;
début facteur (i, j);
       si tab[i] = -290 ou tab[i] = -291 alors
          oper := tab[i];
          i := i + 1;
          expression (i, j);
          postfixée[j] := oper;
          j := j + 1;
       finsi;
fin;
```

```
procédure facteur (var i , j: entier);
var oper : entier;
début terme (i, j);
       si tab[i] = -292 ou tab[i] = -293 alors
          oper := tab[i];
          i := i + 1;
          facteur (i, j);
          postfixée[j] := oper;
          j := j + 1;
       finsi;
fin;
```

```
procédure terme (var i, j : entier);
début si tab[i] >= -286 alors
          postfixée[j] := tab[i];  { identificateur ou nombre }
          i := i + 1;
          j := j + 1;
       sinon    { c'est un ( }
          i := i + 1;
          expression (i, j);
          i := i + 1;    { sauter le ) }
       finsi;
fin;
```

2.2. Autre petit compilateur

On considère un petit langage simple défini comme suit:

```
<instruction> ::= <affectation> | <conditionnelle>
<conditionnelle> ::= ? <condition> : <affectation>
<condition> ::= <expression> {=|#} <expression>
<affectation> ::= <identificateur> = <expression>
<expression> ::= <facteur> {+|-} <expression> | <facteur>
<facteur> ::= <terme> {*|/} <facteur> | <terme>
<terme> ::= <identificateur> | <nombre> | ( <expression> )
```

Les identificateurs sont des unités lexicales constituées soit d'une lettre unique, soit d'une lettre suivie d'un chiffre. Les nombres sont également des unités lexicales constituées de chiffres; on ne prend en compte que des entiers positifs ou nuls. Ces unités lexicales peuvent donc être définies par les règles suivantes:

```
<identificateur> ::= <lettre> | <lettre><chiffre>
<nombre> ::= <chiffre><nombre> | <chiffre>
```

Les espaces ne peuvent se trouver à l'intérieur d'une unité lexicale, et n'ont pas de signification. Une instruction doit se trouver toute entière sur une seule ligne, et une ligne ne peut contenir qu'une seule instruction.

A- L'analyse lexicale consiste à reconnaître la suite des unités lexicales d'une ligne, et à fournir la suite des codes syntaxiques de ces unités lexicales. Elle doit donc fournir une suite d'entiers correspondant à ces codes, de la façon suivante:

- Pour un nombre, le code est la valeur de l'entier correspondant.
- Pour un identificateur, le code est l'entier négatif, compris entre -286 et -1, ayant pour valeur -`numéro_de_lettre` si l'identificateur est composé d'une seule lettre, et -`(numéro_de_lettre+26*(chiffre+1))`, s'il est composé d'une lettre et d'un chiffre.
- Pour un caractère spécial, le code associé est défini par la table suivante:

caractère	=	(	)	+	-	*
code	-287	-288	-289	-290	-291	-292
caractère	/	?	:	#	`K_fin_ligne`	
code	-293	-294	-295	-296	-297	

A.1- Donner le résultat fourni par l'analyseur lexical pour les lignes suivantes:

A1 = A * 100 + 2*B

A1 = A + 100 * B-C

? A1 = B : C = D

A.2- Donner le résultat fourni par l'analyseur lexical pour les lignes suivantes:

?A=3

AB=32

A.3- Que doit indiquer l'analyseur lexical pour la ligne suivante:

A = 2 ! 3

B- L'analyse syntaxique consiste à vérifier la concordance de la suite des codes syntaxiques telle qu'elle résulte de l'analyse lexicale avec les règles de syntaxe du langage. Elle permet aussi de retrouver la structure syntaxique du "programme".

B.1- Donner la structure syntaxique pour chacune des lignes suivantes:

A1 = A * 100 + 2*B

A1 = A + 100 * B-C

? A1 = B : C = D

B.2- Expliquer pourquoi chacune des lignes suivantes est syntaxiquement erronée:

?A=3

AB=32

B.3- Pourquoi n'y a-t-il pas lieu de faire l'analyse syntaxique de la ligne suivante:

A = 2 ! 3

Solution de l’exercice 2.2

2.2.1. Question A

2.2.1.1. Question A.1

Le résultat de l'analyse lexicale est le suivant:

Pour :	Le codage obtenu est :
A1 = A * 100 + 2*B	-53, -287, -1, -292, 100, -290, 2, -292, -2, -297
A1 = A + 100 * B-C	-53, -287, -1, -290, 100, -292, -2, -291, -3, -297
? A1 = B : C = D	-294, -53, -287, -2, -295, -3, -287, -4, -297

2.2.1.2. Question A.2

Le résultat de l'analyse lexicale est le suivant:

Pour :	Le codage obtenu est :
?A=3	-294, -1, -287, 3, -297
AB=32	-1, -2, -287, 32, -297

Comme un identificateur est constitué d'une lettre seule, ou d'une lettre suivie d'un chiffre, la présence ici de deux lettres qui se suivent est interprété, lexicalement, comme une succession de deux identificateurs.

2.2.1.3. Question A.3

Lors de l'analyse de la phrase "A = 2 ! 3", on constate la présence d'un caractère qui ne fait partie d'aucun symbole de notre langage. Il s'agit donc d'une erreur lexicale.

2.2.2. Question B

2.2.2.1. Question B.1

Retrouver la structure syntaxique consiste à retrouver les règles de productions qui ont été utilisées pour construire la phrase. Évidemment, on peut utiliser les algorithmes donnés dans le problème 17.1 du livre, adaptés en fonction des quelques modifications apportées au langage. C'est ce que fera l'analyseur syntaxique. On peut aussi raisonner de façon apparemment plus intuitive, mais en fait analogue à l'algorithme de l'analyseur, en essayant de "deviner" les règles utilisées.

Pour A1 = A * 100 + 2*B, on cherche à développer `<instruction>`, en utilisant l'une des règles possibles, qu'il faut essayer successivement, en cas de refus, jusqu'à trouver le bon choix. On développe donc

```
1    <instruction> ::= <affectation>
2    <affectation> ::= <identificateur> = <expression>
```

qui permet d'accepter "A1 =". Ensuite, on utilise successivement les règles

```
3    <expression> ::= <facteur> {+|-} <expression> | <facteur>
4    <facteur> ::= <terme> {*|/} <facteur> | <terme>
5    <terme> ::= <identificateur> | <nombre> | ( <expression> )
```

permettant d'accepter "A". Le retour sur la règle 4 définissant `<facteur>` conduit ensuite à accepter "*", et à développer les règles

```
6    <facteur> ::= <terme> {*|/} <facteur> | <terme>
7    <terme> ::= <identificateur> | <nombre> | ( <expression> )
```

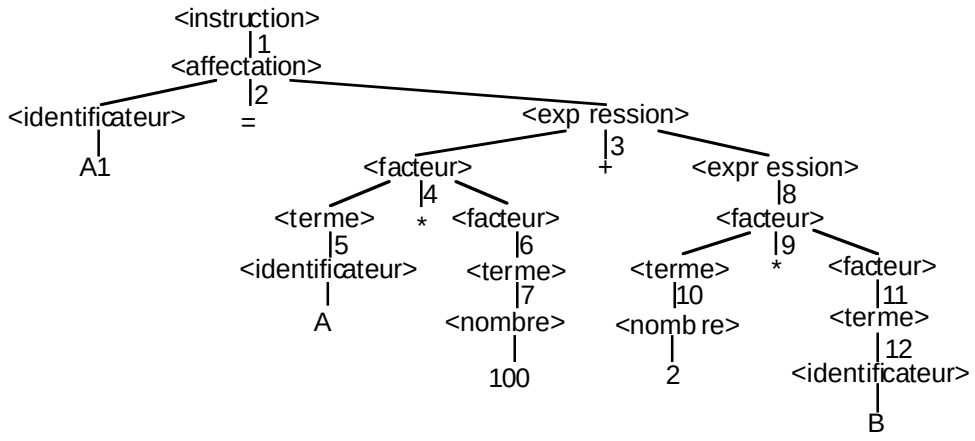
qui conclut sur le nombre 100. Cette fois le retour sur la règle 6 définissant `<facteur>` conduit à la terminaison de cette règle, puis à la terminaison de la règle 4, permettant alors l'acceptation du "+". On développe alors les règles

```
8    <expression> ::= <facteur> {+|-} <expression> | <facteur>
9    <facteur> ::= <terme> {*|/} <facteur> | <terme>
10   <terme> ::= <identificateur> | <nombre> | ( <expression> )
```

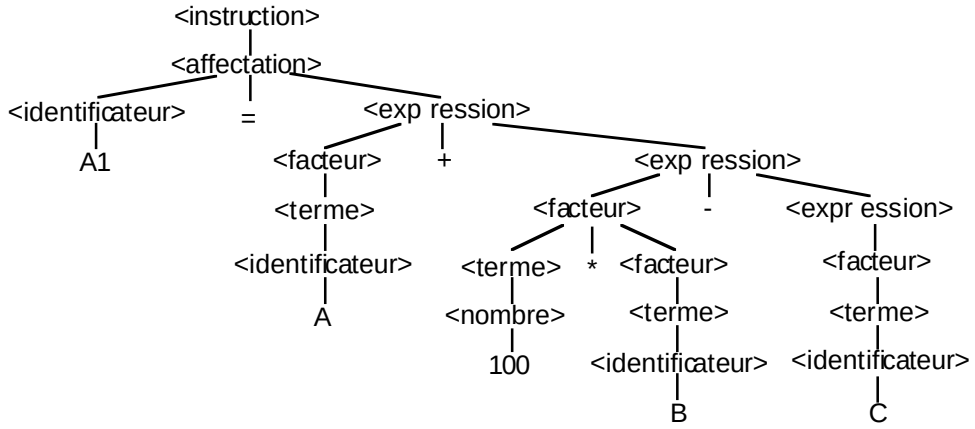
permettant d'accepter "2". Le retour sur la règle 9 définissant `<facteur>` conduit ensuite à accepter "*", et à développer les règles

```
11   <facteur> ::= <terme> {*|/} <facteur> | <terme>
12   <terme> ::= <identificateur> | <nombre> | ( <expression> )
```

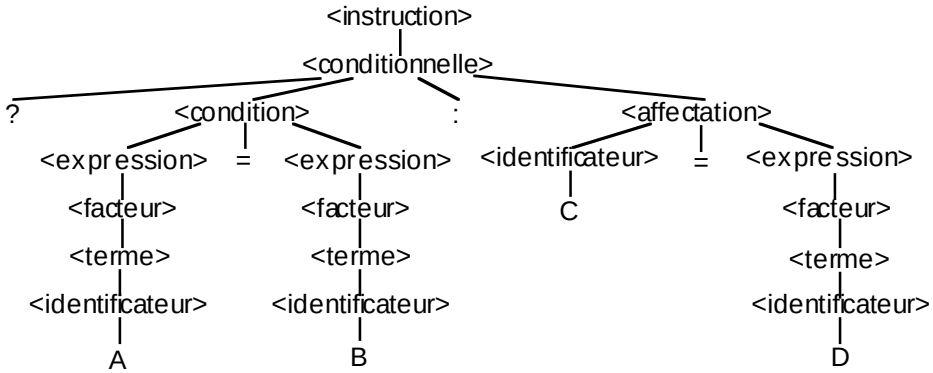
qui conclut sur l'identificateur B. Le retour sur la règle 11 conduit à la terminer, impliquant la terminaison de la règle 9. Le retour sur la règle 8 conduit également à la terminer, impliquant la terminaison des règles 3 puis 2 puis 1, et ensuite à l'acceptation de `K_fin_ligne`, et l'acceptation finale de la phrase. La structure syntaxique est donc:



Pour A1 = A + 100 * B-C, nous obtenons

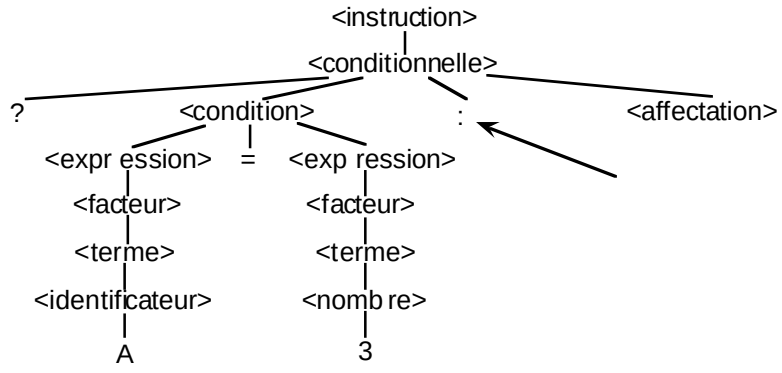


Pour ? A1 = B : C = D, nous obtenons



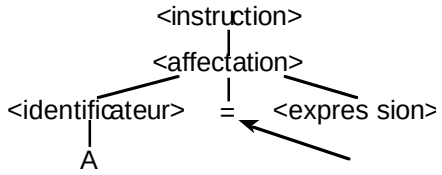
2.2.2.2. Question B.2

Pour la phrase ?A=3, l'analyseur syntaxique commence à construire la structure suivante



Lorsqu'il vérifie que le symbole est ':', il trouve en fait `K_fin_ligne`. Comme il n'a pas d'autre choix possible, il indique une erreur syntaxique.

Pour la phrase AB=32, rappelons d'abord que l'analyseur lexical l'a transformée en suite de symboles codés -1, -2, -287, 32, -297. Ceci montre bien la présence de deux identificateurs qui se suivent. L'analyseur syntaxique commence à construire la structure suivante



Lorsqu'il vérifie que le symbole est '=', il trouve en fait l'identificateur 'B'. Comme il n'a pas d'autre choix possible, il indique une erreur syntaxique.

2.2.2.3. Question B.3

La phrase A = 2 ! 3 étant erronée lexicalement, il est inutile d'en faire l'analyse syntaxique. De fait, le caractère '!' ne correspondant à aucun symbole, ne peut être codé.

Notons que dans certains cas, pour permettre de trouver le maximum d'erreurs le plus vite possible, l'analyseur lexical ignorera ce caractère après avoir signalé l'erreur. Il transmettra à l'analyseur syntaxique la suite codée, comme s'il n'existait pas.

2.3. Expressions dans un tableur

Un tableur est un logiciel qui permet la manipulation d'un ensemble de valeurs organisées sous la forme d'un tableau de lignes et de colonnes. Nous considèrerons ici qu'il y a 256 lignes et 256 colonnes. Chaque case du tableau peut contenir soit une chaîne de caractères, soit un nombre, soit une formule. Une formule décrit un calcul qui permet d'obtenir la valeur de la case correspondante. Ce calcul peut utiliser des valeurs d'autres cases du tableau, des nombres ou des fonctions spécifiques prédéfinies. Nous nous intéressons ici à l'analyse lexicale et syntaxique du langage qui permet de définir le contenu d'une case, sous la forme d'une phrase. Ce langage est défini comme suit:

```
<phrase> ::= <formule> | <nombre signé> | <texte>
<formule> ::= = <expression>
<expression> ::= <terme> {+|-|*|/} <expression> | <terme>
<terme> ::= <référence> | <nombre signé> | ( <expression> )
<nombre signé> ::= <nombre> | {+|-} <nombre>
```

Un texte est une unité lexicale, constituée d'une suite quelconque de caractères qui n'est pas un nombre signé et qui ne commence pas par le signe "=". Les nombres sont des unités lexicales constituées de chiffres. Les références désignent une case du tableau. Ce sont des unités lexicales constituées d'une ou deux lettres suivies d'un à trois chiffres. Le groupe de lettres repère la colonne (A pour la colonne 1, Z pour la colonne 26, AA pour la colonne 27, AZ pour la colonne 52, etc...) et le groupe de chiffres repère la ligne. Ces unités lexicales peuvent être définies par les règles suivantes, où l'exposant indique le nombre de fois où l'élément correspondant peut être pris:

```
<nombre> ::= <chiffre> <nombre> | <chiffre>
<référence> ::= {<lettre>}1,2 {<chiffre>}1,3
```

Les espaces ne peuvent se trouver à l'intérieur d'une unité lexicale, et n'ont pas de signification. Une phrase doit se trouver toute entière sur une seule ligne, et une ligne ne peut contenir qu'une seule phrase.

A- On peut noter qu'un texte est défini par exclusion (toute phrase qui n'est pas lexicalement un nombre et qui ne commence pas par "="). Comment faudra-t-il prendre en compte la règle définissant une `phrase`?

B- Une référence désigne une case du tableau. Ce tableau contient 256 lignes ou colonnes. Certaines combinaisons conformes à la règle définissant les références ne sont donc pas

acceptables. A quel moment doit-on faire le contrôle de la validité d'une référence (lexical, syntaxique ou après)? Justifiez votre raisonnement.

C- On s'intéresse maintenant à l'analyse lexicale des formules. Même si vous savez, évidemment, que l'analyseur lexical fait un codage des différents symboles, nous ne le ferons pas ici pour simplifier. A la place, vous présenterez le découpage en unités lexicales en enfermant celles-ci dans des rectangles.

C.1- Donner le découpage en unités lexicales des phrases suivantes, en justifiant votre raisonnement.

```
= A11*B2+3
= A2*-5
= 5*(B234-A122)
= A1  2 + B1234
= 23C2
```

C.2- Que doit indiquer l'analyseur lexical pour les phrases suivantes, en justifiant votre raisonnement.

```
= AAA2
= A12  $ 45
```

D- On s'intéresse maintenant à l'analyse syntaxique.

D.1- En vous servant du découpage en unités lexicales fait en C.1, donner la structure syntaxique des phrases suivantes, en justifiant votre raisonnement.

```
= A11*B2+3
= A2*-5
= 5*(B234-A122)
```

D.2- En vous servant du découpage en unités lexicales fait en C.1, expliquer pourquoi chacune des phrases suivantes est syntaxiquement erronée.

```
= A1  2 + B1234
= 23C2
```

E- L'analyse sémantique consiste à déterminer à partir de l'arbre syntaxique la suite des opérations à effectuer sur les données. On peut simuler ce comportement en faisant les calculs directement sur l'arbre. En supposant que A11 vaut 5, B2 vaut 2, A2 vaut 6, B234 vaut 42 et A122 vaut 38, vous porterez sur chaque nœud des arbres syntaxiques corrects de la question D les valeurs correspondants à ces calculs..

Solution de l'exercice 2.3

2.3.1. Question A

La définition de l'unité lexicale `texte` présente une certaine forme de dépendance du contexte. Ainsi une suite d'unités lexicales peut être également une unité lexicale `texte` unique. Par exemple, `A23+5`, est une suite de 3 unités lexicales si elle est précédée de "=", mais peut aussi être une seule unité lexicale. Ce problème est assez analogue à celui des commentaires dans un langage de programmation. Il est réglé dans ce cas par la présence d'un ou plusieurs caractères précis commençant le commentaire ("--" par exemple en Ada). Il est clair ici que si le premier caractère non espace est "=", il s'agit d'une formule et non d'un texte. De même si ce premier caractère n'est pas un chiffre ni "+", ni "-", alors il s'agit d'un texte. Dans tous les autres cas, le premier caractère ne permet pas de savoir ce qu'il en est. Deux solutions entre autres peuvent être envisagées:

1. On effectue une analyse lexicale a priori de la phrase complète.
   - Si une erreur se présente, et que la première unité lexicale est "=", alors il s'agit bien d'une erreur lexicale, sinon il s'agit d'un texte.
   - Si aucune erreur lexicale ne se présente, et que la première unité lexicale est "=", alors il s'agit d'une formule lexicalement correcte, dont il faut faire l'analyse syntaxique.

- Si aucune erreur lexicale se présente, et que la première unité lexicale n'est pas "=", alors il faut faire l'analyse syntaxique en tant que nombre signé. L'erreur syntaxique signifie que l'ensemble de la phrase est un texte.

2. L'autre méthode consiste à contrôler le premier caractère de la phrase.
   - S'il s'agit de "=", il s'agit d'une formule, dont on peut faire l'analyse lexicale puis syntaxique, etc...
   - S'il s'agit d'un chiffre ou de "+" ou de "-", on vérifie alors lexicalement qu'il s'agit d'un nombre signé, ce qui est relativement simple ici puisque alors il ne doit y avoir que des chiffres derrière. Notons que dans la réalité, la structure lexicale des nombres est plus complexe (partie décimale et exposant), mais il s'agit toujours de contrôler une unité lexicale précise, sans avoir à faire une analyse lexicale complète de la phrase.
   - Dans tous les autres cas, il s'agit d'une unité lexicale `texte`.

2.3.2. Question B

L'analyse lexicale d'une référence doit au moins vérifier qu'elle est composée d'une ou deux lettres et de 1 à 3 chiffres. Ce peut être la seule vérification. Il est évident que ZZ ne repère pas une colonne du tableau, bien qu'il n'y ait que deux lettres. De même 342 ne repère pas une ligne du tableau. On peut considérer que cette vérification est d'ordre sémantique, et doit donc être faite ultérieurement. Inversement on peut considérer que cette vérification est d'ordre lexical, et qu'une référence ne peut pas être n'importe quel groupe de deux lettres et n'importe quel groupe de 3 chiffres. La description des groupes acceptables est faisable, mais est assez lourd à faire suivant nos règles habituelles, puisqu'il faudrait lister un grand nombre de cas. Si la règle définissant la structure lexicale d'une référence ne dit pas tout, néanmoins, ZZ342 ne devrait pas être pris lexicalement comme une référence!

Peut-on faire le contrôle au moment de l'analyse lexicale? La réponse est oui, et de façon simple: il suffit de déterminer le numéro de colonne et de ligne durant l'analyse lexicale, en vérifiant que ces numéros sont inférieurs ou égaux à 256. On peut penser d'ailleurs que le codage des unités lexicales reprendrait ce principe.

2.3.3. Question C

2.3.3.1. Question C.1

Les unités lexicales sont constituées des caractères individuels suivants:

```
= + - * / ( )
```

ainsi que des nombres et des références.

Le début d'une référence est une lettre alors que le début d'un nombre est un chiffre. Ceci implique que la recherche du début d'une unité lexicale fixera en même temps ce qu'elle peut être. La fin d'un nombre est déterminé par tout caractère différent d'un chiffre. La fin d'une référence est indiquée par un caractère différent d'un chiffre après avoir rencontré au moins un chiffre, ou le fait d'avoir eu 3 chiffres. Evidemment l'absence de chiffres, ou la présence de plus de deux lettres est une erreur lexicale, de même le débordement du numéro de colonne ou de ligne, conformément à ce qui a été dit dans la question B.

`=` `A11` `*` `B2` `+` `3`

La terminaison de la première référence est provoquée par "`*`", et celle de la seconde est provoquée par "`+`"

`=` `A2` `*` `-` `5`

La terminaison de la première référence est provoquée par "`*`", et celle du nombre est provoquée par la fin de la phrase.

`=` `5` `*` `(` `B234` `-` `A122` `)`

La terminaison de la première référence est provoquée par "`-`", et celle de la seconde est provoquée par "`)`".

`=` `A1` `2` `+` `B123` `4`

La terminaison de la première référence est provoquée par l'espace, comme celle du nombre 2. Celle de la deuxième référence est provoquée par la rencontre du troisième chiffre, ce qui implique que le quatrième chiffre commence une nouvelle unité lexicale qui se termine par la fin de la phrase.

`=` `23` `C2`

La deuxième unité lexicale commence par un chiffre, c'est donc un nombre dont la terminaison est provoquée par la rencontre d'une lettre. Celle-ci commence une nouvelle unité lexicale qui ne peut être qu'une référence qui se termine par la fin de la phrase.

2.3.3.1. Question C.2

Pour la phrase `=AAA2`, l'analyseur lexical doit indiquer une erreur sur le troisième `A`. En effet le premier indique le commencement d'une référence qui ne peut comporter que deux lettres au plus, suivies de chiffres. La troisième lettre est refusée, puisqu'il ne peut y avoir à cet endroit qu'un chiffre.

Pour la phrase `=A12 $ 45`, l'analyseur lexical doit indiquer une erreur sur le caractère "`$`", puisque ce caractère ne fait pas partie de ceux qui entrent dans la composition des unités lexicales. Notons que sans le "`=`", la phrase sera interprétée comme du texte, et il n'y aura pas d'erreur.

2.3.4. Question D

2.3.4.1. Question D.1

Retrouver la structure syntaxique consiste à retrouver les règles de productions qui ont été utilisées pour construire la phrase. Évidemment, on peut utiliser les algorithmes donnés dans le problème 2.1 du livre, adaptés en fonction des quelques modifications apportées au langage. C'est ce que fera l'analyseur syntaxique. On peut aussi raisonner de façon apparemment plus intuitive, mais en fait analogue à l'algorithme de l'analyseur, en essayant de "deviner" les règles utilisées.

Pour `=` `A11` `*` `B2` `+` `3` , on cherche à développer `<formule>`, en utilisant la seule règle possible. On développe donc

```
1    <formule> ::= = <expression>
```

qui permet d'accepter "`=`". Ensuite, on utilise successivement les règles

```
2    <expression> ::= <terme> {+|-|*|/} <expression> | <terme>
3    <terme> ::= <référence> | <nombre signé> | ( <expression> )
```

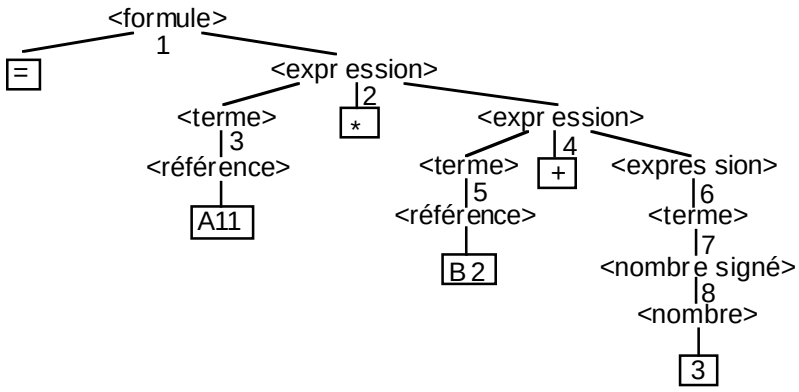
permettant d'accepter "`A11`" comme référence. Le retour sur la règle 2 définissant `<expression>` conduit ensuite à accepter "`*`", et à développer les règles

```
4    <expression> ::= <terme> {+|-|*|/} <expression> | <terme>
5    <terme> ::= <référence> | <nombre signé> | ( <expression> )
```

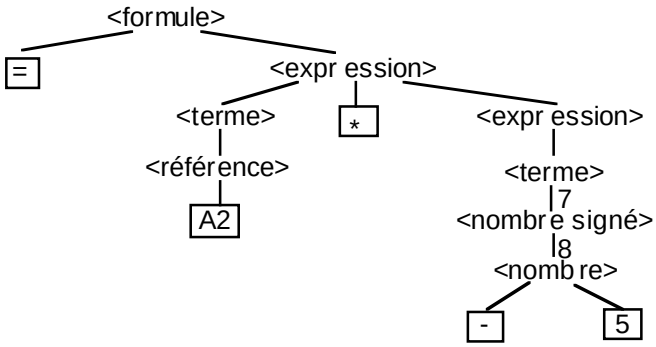
qui conclut sur la référence "`B2`". Le retour sur la règle 4 définissant `<expression>` conduit à accepter "`+`", et à développer alors les règles

```
6    <expression> ::= <terme> {+|-|*|/} <expression> | <terme>
7    <terme> ::= <référence> | <nombre signé> | ( <expression> )
8    <nombre signé> ::= <nombre> | {+|-} <nombre>
```

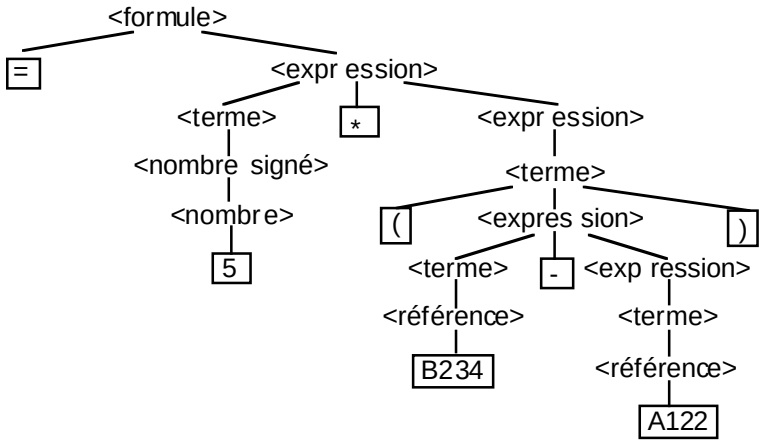
permettant d'accepter "`3`". Le retour sur la règle 6 conduit à la terminer, impliquant la terminaison de la règle 4, puis 2 puis 1, et ensuite à l'acceptation finale de la formule. La structure syntaxique est donc:



Pour [=] [A2] [*] [-] [5] , nous obtenons

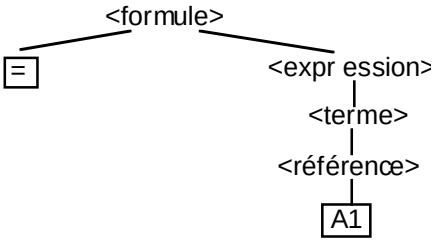


Pour [=] [5] [*] [(] [B234] [-] [A122] [)] , nous obtenons



2.3.4.2. Question D.2

Pour la phrase [=] [A1] [2] [+] [B123] [4] , l'analyseur syntaxique commence à construire la structure suivante

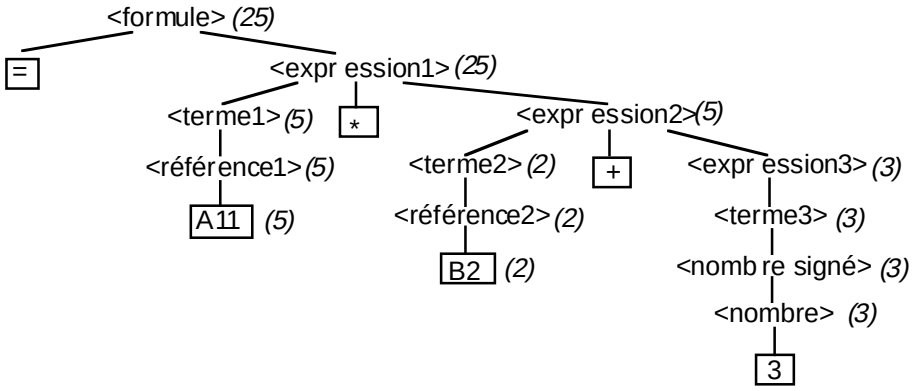


Comme il constate qu'il y a encore des symboles qui suivent, il indique une erreur syntaxique.

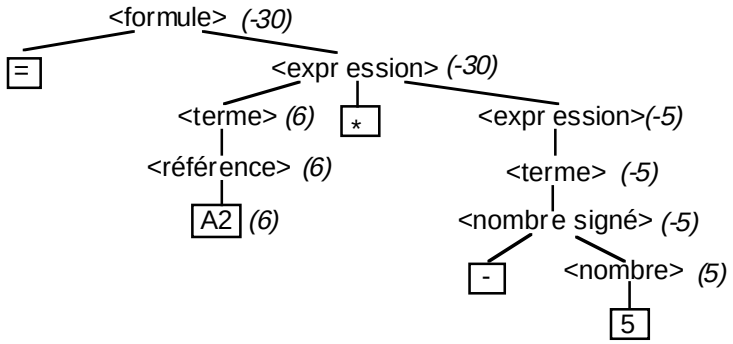
Pour la phrase [=] [23] [C2] , l'analyseur syntaxique commence à construire la structure analogue se terminant sur le nombre 23. Comme il y a encore des symboles qui suivent, il indique une erreur syntaxique.

2.3.5. Question E

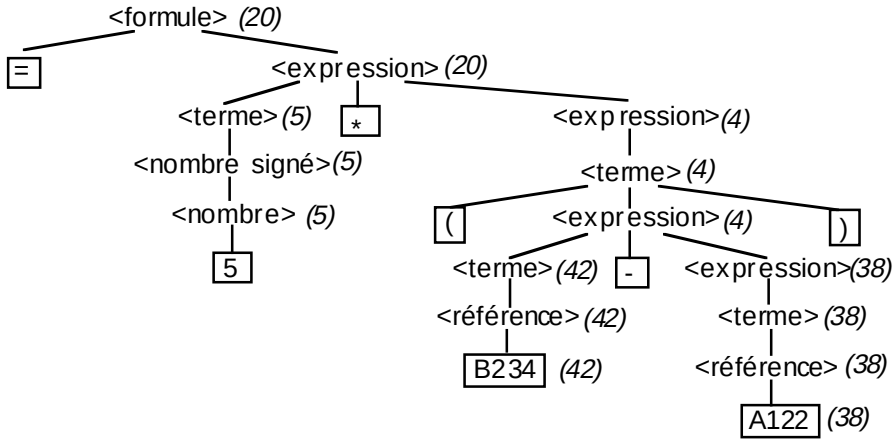
Les valeurs sont données aux opérandes qui sont aux feuilles de l'arbre, et remontent vers la racine, en se combinant par les opérateurs rencontrés. Ainsi, pour la formule = `A11*B2+3`, la valeur 5 de A11 est donc la valeur de la référence1, donc du terme1, opérande gauche de l'expression1. La valeur 2 de B2 est donc la valeur de référence2, donc du terme2, opérande gauche de l'expression2. La valeur 3 est la valeur du nombre, donc du nombre signé, donc du terme3 et enfin de l'expression3, opérande droit de l'expression2. On combine alors les opérandes et l'opérateur de l'expression 2 qui vaut donc 5. On combine ensuite les opérandes et l'opérateur de l'expression1 qui vaut 25, valeur de la formule.



Pour la formule = `A2*-5`, on obtient l'arbre suivant :



Pour la formule = `5*(B234-A122),` on obtient l'arbre suivant :



Les autres expressions proposées sont syntaxiquement erronées. Aucun calcul n'est donc effectué.

2.4. Langage de commande

Dans cet exercice, on étudie brièvement la structure d'un langage de commande. Ce langage est défini comme suit :

```
<suite commandes> ::= <commande> | <commande> ; <suite commandes>
<commande> ::= <mot> | <mot> <suite paramètres>
<suite paramètres> ::= <mot> | <mot> <suite paramètres>
```

Un mot est une unité lexicale constituée d'une suite de caractère de longueur quelconque (>0), ne contenant pas d'espace, tabulation, fin de ligne ou point-virgule, sauf s'il commence par le caractère « ' » auquel cas le mot se termine sur le caractère « ' » suivant, quels que soient les caractères situés entre les deux. Dans ce cas, le mot lui-même ne contient pas les caractères « ' » de début et de fin.

A– Donner le découpage en mots et commandes des lignes suivantes, en justifiant votre réponse :

```
cc -c toto.c
ld toto.o -o toto; rm toto.o
truc c'est le schmilblick
alias virus 'bidule toto; machin'
```

B– L'interpréteur de commande reconnaît la commande interne `alias`, qui a deux arguments A1 et A2, et qui ajoute le couple <A1, A2> à une table de correspondance des alias. Cette table conserve donc les associations entre les arguments des commandes `alias` déjà exécutées. Après avoir effectué le découpage en mot d'une commande quelconque, l'interpréteur recherche le premier mot de la commande dans cette table (première composante d'un couple), et s'il le trouve, le remplace, dans la commande en cours, par la valeur associée (deuxième composante du couple). Expliquer l'intérêt de ce mécanisme, et décrire son effet sur la ligne suivante :

```
virus alpha
```

Solution de l'exercice 2.4

2.4.1. Question A

Il y a peu de difficulté sur le découpage des mots. En fait, c'est en général l'espace ou la fin de ligne qui détermine la fin d'un mot, sauf si le mot commence avec le caractère « ' ». Le seul caractère particulier étant lui-même unité lexicale est le « ; ». Le découpage en mot est la suivante :

`cc` `-c` `toto.c` Il n'y a qu'une seule commande avec deux paramètres.

`ld` `toto.o` `-o` `toto` `;` `rm` `toto.o` Il y a ici deux commandes, l'une `ld` avec trois paramètres, et l'autre `rm` avec un paramètre.

`truc` `c'est` `le` `schmilblick` Il n'y a qu'une seule commande avec trois paramètres. Le caractère « ' » présent dans `c'est`, n'étant pas en début de mot est considéré, comme un caractère ordinaire.

`alias` `virus` `bidule toto; machin` Il y a une seule commande, avec deux paramètres. Notons que le caractère « ' » étant en début de mot implique que la fin du mot est le caractère « ' » suivant. Ces deux caractères spéciaux ont été enlevés du mot, puisqu'ils n'en font pas partie. Ce ne sont pas non plus des symboles du langage, mais des délimiteurs lexicaux, comme l'espace ou la fin de la ligne. Par ailleurs, le « ; » est considéré ici comme un caractère quelconque faisant partie du mot, et non symbole du langage.

2.4.1. Question A

Le mécanisme d'alias est en fait un mécanisme d'abréviation (voir 15.2.4). Sur la commande `virus alpha`, il va y avoir substitution du mot virus par le mot associé, pour donner la suite de commandes :

```
bidule toto; machin alpha
```

Ceci sera donc interprété comme une suite de deux commandes : `bidule toto` suivie de `machin alpha`.

2.5. Pile d'exécution

Le but de ce problème est l'étude de la pile d'exécution d'un programme.

Un programme séquentiel est composé d'un ensemble de procédures qui s'appellent mutuellement. A chacune de ces procédures est associée une suite d'instructions, encore appelée le code de la procédure. Appelons *activité* le phénomène résultant de l'exécution ininterrompue d'une procédure unique. L'exécution d'un programme est donc une suite d'activités. Nous appelons *contexte* d'une activité l'ensemble des informations accessibles au processeur au cours de cette activité. Le contexte comprend donc un contexte du processeur (registres) et un contexte en mémoire (code et données). Le passage d'une activité à une autre est réalisé par des instructions spéciales, l'appel et le retour de procédure, qui réalisent une commutation du contexte.

Lors de l'appel d'une procédure *q* par une procédure *p*, il faut préparer les paramètres transmis à *q* par *p*, sauver le contexte de *p* pour le retrouver au retour, et remplacer ce contexte par celui de *q*. Au retour, il faut préparer les paramètres transmis par *q* à *p*, et restaurer le contexte de *p*. Le contexte de *q* est perdu. Cette sauvegarde et cette restauration utilisent une pile. Nous supposerons que les paramètres sont transmis par valeur; un résultat unique est rendu au retour. Les procédures peuvent être appelées récursivement directement ou indirectement.

A- Justifier l'utilisation d'une pile pour cette sauvegarde et cette restitution.

B- Comment passer les paramètres et le résultat.

C- Où peut-on mettre les locaux de la procédure appelée?

D- Décrire de façon détaillée les opérations d'appel et de retour, en précisant à chaque fois ce qui est fait par la procédure appelante et par la procédure appelée.

E- Pensez-vous que cette pile puisse être utile dans un contexte de multiprogrammation? d'appels au système? d'interruptions?

Solution de l’exercice 2.5

Remarque: S'il n'y a pas de récursivité, on peut réserver des emplacements fixes pour les contextes et paramètres des différentes procédures du programme. La pile n'est pas alors nécessaire. Cependant ceci conduit à réserver statiquement l'espace mémoire utilisé par les différentes procédures, même lorsqu'elles ne sont pas en cours d'exécution.

2.5.1. Question A

Le retour de procédure se fait en ordre inverse des appels. La restauration doit donc se faire en ordre inverse des sauvegardes. C'est bien la structure de pile qui offre cet ordre.

2.5.2. Question B

Les paramètres doivent être mémorisés quelque part par la procédure appelante, et pouvoir être accédés par la procédure appelée. La pile peut servir de zone d'échange puisque les deux procédures connaissent le sommet de pile au moment de l'appel. Par ailleurs, comme la procédure appelée normalement les utilise pendant toute la durée de son activité, c'est-à-dire jusqu'au retour, ils peuvent rester dans la pile jusqu'à ce moment là.

Un raisonnement analogue pour le résultat montre que celui-ci doit être également conservé dans la pile pendant toute la durée de l'exécution de la procédure, pour être transmis à l'appelante lors du retour.

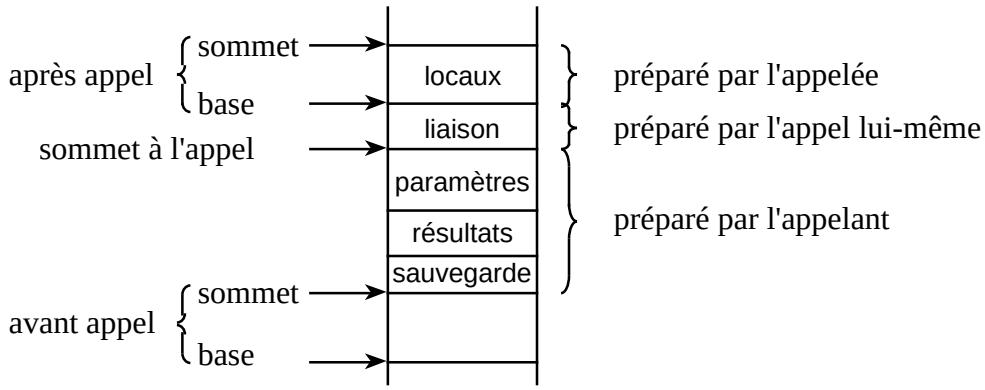
2.5.3. Question C

Les locaux de la procédure doivent être créés lors du début de l'activité correspondante, et détruits lors de la fin de l'activité. Si on admet les appels récursifs, ces locaux ne peuvent être à une adresse fixe en mémoire, puisqu'une même procédure peut donner lieu à plusieurs activités qui doivent coexister ensembles, avec chacune des locaux différents. Leur création/destruction doit donc être

dynamique. Comme la création correspond à l'appel et la destruction au retour, ils sont détruits en ordre inverse de leur création, toutes procédures confondues. On peut donc utiliser la pile pour cela.

2.5.4. Question D

L'évolution de la pile au cours d'un appel de procédure peut être décrit par le schéma donné ci après.



Comme les objets locaux sont à un emplacement variable de la pile, il est nécessaire de repérer la zone où ils se trouvent, pour une exécution donnée de la procédure. Dans le schéma ci-dessus, ce rôle est assuré par ce que nous avons dénoté *base*. En général, il s'agit d'un registre que les compilateurs réservent à cet effet, par convention. Nous laissons à la procédure appelée le soin de ranger ce registre (et de le restituer à la fin); certaines procédures n'ayant pas de locaux, peuvent ne pas s'en préoccuper, pourvu que le registre ait, au moment du retour, la même valeur qu'au moment de l'appel.

procédure appelante:

```
tant que il faut sauvegarder faire
     empiler (ce qu'il faut)
fait
réserver place résultat en pile
tant que il y a des paramètres faire
     empiler (valeur du paramètre)
fait
appel_proc
supprimer les paramètres de la pile, si nécessaire
dépiler (résultat)
tant que il faut restaurer faire
     dépiler (ce qu'il faut)
fait
```

procédure appelée:

```
empiler (base)
base := sommet
réservation des locaux { sommet := sommet - taille }
corps de la procédure
supprimer les locaux { sommet := sommet + taille }
dépiler (base)
retour_proc
```

2.5.5. Question E

Dans un contexte de multiprogrammation, lors de l'interruption d'un programme, il faut évidemment sauvegarder le contexte en cours. Ceci peut être fait dans la pile. Cependant, le système peut décider de réactiver un programme différent de celui qu'il vient d'interrompre. Dans ce cas, on ne peut plus rien dire sur l'ordre des retours vis-à-vis des appels. Ce que l'on peut affirmer, néanmoins, c'est que pour un programme donné, lorsqu'il sera réactivé, il faudra restituer le contexte qu'il avait lorsqu'il avait été suspendu. Il faut donc attribuer une pile par programme.

Lors d'un appel système ou d'une interruption, le système doit sauvegarder le contexte du programme dans sa pile, et mémoriser les registres `base` et `sommet` dans un emplacement réservé au programme. Il peut alors prendre une pile qui lui est propre pour continuer le travail. Lorsqu'il veut relancer un programme, il restitue d'abord les registres `base` et `sommet` du programme, et restaure ensuite le contexte à partir de cette pile.

2.6. Étude du chargeur et de l'éditeur de liens

Le but de cet exercice est d'approfondir le fonctionnement du chargeur et de l'éditeur de liens, déjà vu dans la deuxième partie.

Le résultat d'une compilation ou d'un assemblage est un fichier dont les enregistrements décrivent non seulement la traduction du module source dans le langage de la machine, mais aussi les informations nécessaires pour relier ce module aux autres modules qui constitueront un programme. Ces enregistrements peuvent prendre diverses formes, suivant qu'ils contiennent le code ou les données du module, ou servent à l'établissements des liaisons intermodules. Pour fixer les idées, nous définissons le type de ces enregistrements dans une forme Pascal.

```
t_nature = (entête, lien_à_satisfaire, lien_utilisable, code_absolu,
          code_à_translater, code_avec_l_a_s, adresse_lancement, fin_module);
t_enregistrement = article
     variante nature : t_nature dans
     entête:              ( identificateur_module : t_identificateur;
                            taille : entier );
     lien_à_satisfaire:   ( identificateur_l_a_s : t_identificateur );
     lien_utilisable:     ( identificateur_l_u : t_identificateur;
                            valeur_l_u : entier );
     code_absolu, code_à_translater:
                          ( adresse_code : entier;
                            valeur_code : entier );
     code_avec_l_a_s:     ( adr_code_l_a_s : entier;
                            numéro_l_a_s : entier;
                            valeur_code_l_a_s : entier );
     adresse_lancement:   ( adr_lanc : entier );
     fin_module:          ( ) ;
     finarticle;
```

Remarques:

3. La forme présentée par les modules objets, en réalité, est en général plus complexe, mais plus condensée.
4. Le champ `taille` de l'enregistrement `entête` précise le nombre d'emplacements occupés par le module.
5. Les liens à satisfaire sont implicitement numérotés, à partir de 1, suivant l'ordre d'apparition, dans le module, des enregistrements `lien_à_satisfaire` qui les contiennent.

On suppose que l'on dispose d'une procédure `lire_enrg` qui assure la lecture d'un enregistrement du fichier dans la variable `enrg` de type `t_enregistrement` défini ci-dessus. On suppose, pour simplifier que dans le cas où on a une liste de modules, cette procédure `lire_enrg` sait où trouver successivement les modules de la liste, et qu'elle est complétée par la fonction à résultat booléen `il_y_a_des_modules`, ainsi que par la procédure `premier_module` qui prépare le positionnement sur le premier enregistrement du premier module de la liste. Par ailleurs, on dispose d'une procédure `écrire_enrg` pour écrire la variable `enrg` sur un fichier.

A- Pour fixer les idées, supposons que notre machine ait une mémoire de 10000 emplacements pouvant contenir chacun un entier, qui peut représenter une donnée ou une instruction. Elle dispose de 5 instructions qui sont codées comme suit:

```
LOAD  <adresse>    10000 + adresse    met MEMOIRE[adresse] dans l'accumulateur
ADD   <constante>  20000 + constante  ajoute la constante à l'accumulateur
MUL   <constante>  30000 + constante  multiplie l'accumulateur par la constante
STORE <adresse>    40000 + adresse    met l'accumulateur dans MEMOIRE[adresse]
JMP   <adresse>    50000 + adresse    aller à l'instruction située à adresse
```

On considère le module suivant:

```
      NOM     ESSAI  nom du module
      PUBLIC  INCR
      EXTERN  SUITE
A:    WORD    =25     réservation d'un emplacement pour A initialisé à 25
B:    WORD    1       réservation d'un emplacement pour B non initialisé
INCR: LOAD    A
      MUL     10
      ADD     1
      STORE   B
      JMP     SUITE
      FIN
```

A.1- Donner la suite d'enregistrements que doit produire l'assembleur pour ce module.

A.2- Lors d'une édition de liens d'un programme, le module `ESSAI` est placé à l'emplacement relatif `123` dans le programme exécutable. `SUITE` est un lien utilisable dans un autre module, à l'emplacement relatif `8`. Cet autre module est placé immédiatement derrière le module `ESSAI` dans le programme exécutable. Donner la suite des enregistrements produits par l'éditeur de liens en provenance du module `ESSAI`.

A.3- Le chargeur place le programme à l'emplacement `1042` de la mémoire. Donner le contenu des emplacements de la mémoire qui contiennent ce module.

B- La structure de ces enregistrements est utilisable aussi bien dans les modules objets, entre les compilateurs et l'éditeur de liens, que dans les programmes exécutables, entre l'éditeur de liens et le chargeur. Expliquer pourquoi. Quels enregistrements ne doivent pas se trouver dans un programme exécutable?

C- On se propose de construire l'éditeur de liens en deux passages: le premier au cours duquel on fixe la carte d'implantation des modules, et où on construit la table des liens, le second passage au cours duquel on effectue la translation et où on établit les liens entre les modules. On dispose, pour la gestion de cette table, de trois procédures:

procédure `entrer (identificateur : t_identificateur; valeur : entier);` ajoute une entrée dans la table,

fonction `recherche (identificateur : t_identificateur;` **var** `valeur : entier): booléen;` recherche une entrée dans la table, retourne *vrai* si cette entrée a été trouvée, et met dans *valeur* celle qui était mémorisée dans la table pour cet identificateur; retourne *faux* si l'entrée n'existe pas.

procédure `modifier (identificateur : t_identificateur; valeur : entier );` change la partie valeur de l'entrée de la table associée à l'identificateur donné en paramètre.

C.1- Définir la procédure de premier passage pour l'ensemble des modules à relier en supposant que l'on dispose d'une procédure `lire_liens_module` qui traite les liens d'un module.

C.2- Définir la procédure `lire_liens_module` qui constitue la table des liens.

C.3- Définir la procédure de second passage pour l'ensemble des modules à relier en supposant que l'on dispose d'une procédure `traiter_module` qui produit les enregistrements relatifs à un module.

C.4- Définir la procédure `traiter_module`.

D- Définir la procédure du chargeur, dont la spécification, i.e. l'en-tête, est:

```
     procédure charger ( nom : t_identificateur;
                         adresse_chargement, taille : entier);
```

qui vérifie la concordance du nom avec celui du module exécutable, contrôle que la zone mémoire définie par les paramètres est compatible avec la définition du module et en assure le chargement en mémoire à l'adresse fournie en paramètre, la translation et le lancement.

Solution de l'exercice 2.6

2.6.1. Question A

2.6.1.1. Question A.1

Le module objet doit se présenter sous la forme suivante:

```
<    entête                 ESSAI     7                  >
<    lien_utilisable        INCR      2                  >
<    lien_à_satisfaire      SUITE                        >
<    code_absolu            0         25                 >
<    code_à_translater      2         10000              >
<    code_absolu            3         30010              >
<    code_absolu            4         20001              >
<    code_à_translater      5         40001              >
<    code_avec_l_a_s        6         1         50000    >
<    fin_module                                          >
```

Remarquons qu'il n'y a pas d'enregistrement pour l'emplacement correspondant à l'objet `B`. On pourrait en mettre un si on voulait lui donner une valeur initiale.

2.6.1.2. Question A.2

Lors de l'édition de liens, le module étant placé à l'emplacement 123, il faut ajouter cette valeur aux adresses internes. Comme il est de taille 7, le module placé derrière lui sera placé à l'emplacement relatif 130. Comme `SUITE` est un lien utilisable défini comme emplacement relatif 8 de ce module, il est donc à l'emplacement relatif 138 du programme. Les enregistrements produits par l'éditeur de liens pour ce module sont donc les suivants:

```
<    code_absolu            123         25             >
<    code_à_translater      125       10123            >
<    code_absolu            126       30010            >
<    code_absolu            127       20001            >
<    code_à_translater      128       40124            >
<    code_à_translater      129       50138            >
```

2.6.1.3. Question A.3

Les emplacements occupés par ce module après chargement sont définis comme suit:

```
1165         25
1166          ?
1167      11165
1168      30010
1169      20001
1170      41166
1171      51180
```

Le programme étant en 1042, le module commence en 1042+123 = 1165. Par ailleurs, le chargeur ajoute 1042 aux emplacements contenant un code à translater. L'emplacement associé à `B` a un contenu indéterminé, puisqu'il n'est pas initialisé.

2.6.2. Question B

La structure de `t_enregistrement` convient évidemment pour les modules objets, puisqu'elle permet d'une part la définition des liens à satisfaire et des liens utilisables, d'autre part la translation de code et les liaisons vers d'autres modules par les enregistrements `code_avec_l_a_s`. Notons que les liens à satisfaire étant repérés dans ce cas par leur numéro d'ordre, l'enregistrement `lien_à_satisfaire` correspondant au $n^{ième}$ lien à satisfaire doit se trouver avant les enregistrements `code_avec_l_a_s` mentionnant ce numéro n.

Pour un module exécutable, la structure convient aussi, mais certains enregistrements ne devront pas se trouver dans un tel module. En effet, un module exécutable ne peut contenir que les enregistrements `entête`, `code_absolu`, `code_à_translater`, `adresse_lancement`, ou

`fin_module`. On peut admettre aussi les enregistrements `lien_utilisable`, mais les enregistrements `lien_à_satisfaire` et `code_avec_l_a_s` sont interdits.

2.6.3. Question C

2.6.3.1. Question C.1

Le premier passage doit construire la carte d'implantation des modules, en fixant l'adresse origine de chaque module. C'est au cours de ce premier passage que l'on fixe les valeurs des liens utilisables de l'ensemble des modules. On peut aussi prendre en compte les liens à satisfaire, de façon à permettre ensuite de compléter la liste des modules avec d'autres provenant de bibliothèques, qui contiennent ces liens comme liens utilisables, s'ils n'ont pas été trouvés dans la liste initiale des modules. Nous identifierons les liens à satisfaire par une valeur égale à -1. Le programme principal est le suivant:

```
procédure premier_passage ;
var adr_implantation;
début adr_implantation := 0;
       premier_module;
       tantque il_y_a_des_modules faire
          lire_liens_module ( adr_implantation );
       fait;
       tantque il_y_a_des_entrées_de_la_table_avec_valeur_négative faire
          si un_module_bibliothèque_contient_un_tel_lien_utilisable alors
             ajouter_ce_module_à_la_liste;
             lire_liens_module ( adr_implantation );
          finsi;
       fait;
fin;
```

2.6.3.2. Question C.2

Pour faciliter l'écriture de la procédure de lecture des liens d'un module, on définit deux procédures qui effectuent l'adjonction d'un lien dans la table, `ajout_l_u` et `ajout_l_a_s`. Par ailleurs, nous utiliserons une procédure erreur qui imprime le message fourni en paramètre, et arrête le programme.

```
procédure ajout_l_u ( id : t_identificateur; val : entier );
var valeur : entier;
début si recherche ( id, valeur ) alors
          si valeur = -1 alors         { identificateur rencontré comme l_a_s }
             modifier ( id, val );
          sinon
             erreur ("double définition"); {identificateur rencontré comme l_u }
          finsi;
       sinon entrer ( id, val );        { identificateur non encore rencontré }
       finsi;
fin;
```

```
procédure ajout_l_a_s ( id : t_identificateur );
var valeur : entier;
début si non recherche ( id, valeur ) alors entrer ( id, -1 ); finsi;
fin;
```

```
procédure lire_liens_module ( var adresse : entier );
var adr_impl : entier;
début lire_enrg;
       si enrg.nature <> entête alors erreur ( "pas en début de module" );
       sinon
          adr_impl := adresse;
          adresse := adresse + enrg.taille;
          ajout_l_u ( enrg.identificateur_module, adr_impl );
          tantque enrg.nature <> fin_module faire
             lire_enrg;
             si enrg.nature = lien_à_satisfaire alors
                 ajout_l_a_s ( enrg.identificateur_l_a_s );
             sinon si enrg.nature = lien_utilisable alors
                 ajout_l_u(enrg.identificateur_l_u,enrg.valeur_l_u + adr_impl);
                 finsi;
             finsi;
          fait;
       finsi;
fin;
```

2.6.3.3. Question C.3

Le deuxième passage a pour but de faire un seul module exécutable avec l'ensemble des modules de la liste augmentée éventuellement de ceux provenant de la bibliothèque. Les enregistrements relatifs à chacun des modules sont produits par la procédure `traiter_module`. Il s'agit ici de produire le premier enregistrement, d'exécuter cette procédure sur chacun des modules et de produire le dernier enregistrement.

```
procédure deuxième_passage ;
début enrg.nature := entête;
       enrg.identificateur_module := nom_du_programme;
       enrg.taille := taille_du_programme; { adr_implantation fin de passage 1 }
       écrire_enrg;
       premier_module;
       tantque il_y_a_des_modules faire traiter_module; fait;
       enrg.nature := fin_module;
       écrire_enrg;
fin;
```

2.6.3.4. Question C.4

Le deuxième passage doit effectuer la translation des modules et établir les liaisons intermodules, en utilisant la table des liens construite au premier passage. Le traitement des enregistrements `code_avec_l_a_s` nécessite de disposer d'une table locale qui associe à chaque numéro de lien à satisfaire sa valeur provenant de la table globale. Lors de la recherche de ce lien dans la table globale, il doit normalement s'y trouver, puisque le premier passage l'y a mis. Il peut cependant ne pas être défini, si on n'a pas trouvé de lien utilisable correspondant.

La procédure de traitement d'un module est décomposée en un ensemble de procédures, chacune d'elles correspondant au traitement à faire pour chaque enregistrement du module, suivant sa nature.

```
var table_liens_à_satisfaire : tableau [ 0..500 ] de entier;
   nb_las, base_impl : entier;
procédure traiter_lien_à_satisfaire;
var valeur : entier;
début si nb_las = 500 alors erreur("trop de liens à satisfaire dans module");
       sinon
          nb_las := nb_las + 1;
          si recherche ( enrg.identificateur_l_a_s, valeur ) alors
            si valeur = -1 alors
               erreur ( "identificateur non défini", enrg.identificateur_l_a_s );
            finsi;
          sinon erreur ( "anomalie de fonctionnement" );
          finsi;
          table_liens_à_satisfaire [ nb_las ] := valeur;
       finsi;
fin;
```

```
procédure traiter_code_absolu;
début enrg.adresse_code := enrg.adresse_code + base_impl;
       écrire_enrg;
fin;
procédure traiter_code_à_translater;
début enrg.adresse_code := enrg.adresse_code + base_impl;
       enrg.valeur_code := enrg.valeur_code + base_impl;
       écrire_enrg;
fin;
procédure traiter_code_avec_l_a_s;
début enrg.adresse_code := enrg.adr_code_l_a_s + base_impl;
       si enrg.numéro_l_a_s > nb_las alors erreur ( "module incorrect" );
       sinon enrg.valeur_code := enrg.valeur_code_l_a_s +
                              table_liens_à_satisfaire [ enrg.numéro_l_a_s ];
       finsi;
       enrg.nature := code_à_translater;
       écrire_enrg;
fin;
procédure traiter_adresse_lancement;
début enrg.adr_lanc := enrg.adr_lanc + base_impl;
      écrire_enrg;
fin;
procédure traiter_module;
début lire_enrg;
       si enrg.nature <> entête alors erreur ( "pas en début de module" );
       sinsi recherche ( enrg.identificateur_module , base_impl ) alors
          nb_las := 0;
          tantque enrg.nature <> fin_module faire
               lire_enrg;
               cas enrg.nature dans
                    lien_à_satisfaire: traiter_lien_à_satisfaire;
                    code_absolu: traiter_code_absolu;
                    code_à_translater: traiter_code_à_translater;
                    code_avec_las: traiter_code_avec_las;
                    adresse_lancement: traiter_adresse_lancement;
                    lien_utilisable, fin_module : ;
                    autre : erreur ( "module incorrect" );
               fincas;
          fait;
       sinon erreur ( "module non rencontré en premier passage" );
      finsi;
fin;
```

2.6.4. Question D

La procédure ne présente pas de difficulté particulière. On peut faire une procédure interne `ranger` qui assure le contrôle des adresses et le rangement en mémoire des valeurs successives, éventuellement translatées. Cette vérification doit souvent être faite, car le chargeur lui-même peut parfois accéder à toute la mémoire, sans protection particulière. Il faut aussi contrôler l'absence des enregistrements interdits. Enfin pour pouvoir faire le lancement du programme, il faut contrôler la présence de l'enregistrement correspondant, et vérifier son paramètre.

```
procédure charger(nom: t_identificateur; adresse_chargement, taille: entier);
  var adresse_init : entier;
       adr_lanc_prés : booléen;
  procédure ranger ( adr , val : entier );
  début   si adr < taille alors mémoire [ adresse_chargement + adr ] := val;
          sinon erreur ( "module incorrect" );
          finsi
  fin;
{ corps de la procédure charger }
début lire_enrg;
     si enrg.nature <> entête ou enrg.identificateur_module <> nom
        ou enrg.taille > taille alors erreur ( "paramètres incompatibles" );
     sinon
          adr_lanc_prés := faux ;
          tantque enrg.nature <> fin_module faire
               lire_enrg;
```

```
                cas enrg.nature dans
                     code_absolu: ranger ( enrg.adresse_code, enrg.valeur_code );
                     code_à_translater: ranger ( enrg.adresse_code,
                                                 enrg.valeur_code + adresse_chargement );
                     adresse_lancement:
                          si enrg.adr_lanc < taille alors
                               adresse_init := enrg.adr_lanc + adresse_chargement;
                          adr_lanc_prés := vrai;
                          sinon erreur ( "module incorrect" );
                          finsi;
                     fin_module, lien_utilisable: ;
                     autre : erreur ( "module non conforme" );
                fincas;
           fait;
           si adr_lanc_prés alors compteur_ordinal := adresse_init;
           sinon erreur ( "programme sans adresse de lancement" );
           finsi;
     finsi;
fin;
```

2.7. Références croisées

Nous étudions la réalisation d'un outil de production des références croisées entre modules objets.

Dans le problème sur l'édition de liens, nous avons défini une structure d'enregistrements, dans une forme Pascal, pour les modules objets, et qui est rappelée ci-dessous:

```
t_nature = (entête, lien_à_satisfaire, lien_utilisable, code_absolu,
          code_à_translater, code_avec_l_a_s, adresse_lancement, fin_module);
t_enregistrement = article
     variante nature : t_nature dans
     entête:              ( identificateur_module : t_identificateur;
                            taille : entier );
     lien_à_satisfaire:   ( identificateur_l_a_s : t_identificateur );
     lien_utilisable:     ( identificateur_l_u : t_identificateur;
                            valeur_l_u : entier );
     code_absolu, code_à_translater:
                          ( adresse_code : entier;
                            valeur_code : entier );
     code_avec_l_a_s:     ( adr_code_l_a_s : entier;
                            numéro_l_a_s : entier;
                            valeur_code_l_a_s : entier );
     adresse_lancement:   ( adr_lanc : entier );
     fin_module:          ( ) ;
     finarticle;
```

Comme dans ce problème précédent, nous supposons que l'on dispose d'une procédure `lire_enrg` qui assure la lecture d'un enregistrement du fichier dans la variable `enrg` de type `t_enregistrement` défini ci-dessus. On suppose, pour simplifier, que dans le cas où on a une liste de modules, cette procédure `lire_enrg` sait où trouver successivement les modules de la liste, et qu'elle est complétée par la fonction à résultat booléen `il_y_a_des_modules`, ainsi que par la procédure `premier_module`.

Le but de ce problème est, ici, l'étude d'un outil d'impression des références croisées entre des modules, et qui permet de connaître, pour chaque identificateur exporté par un module, les modules qui l'importent. On rappelle qu'un identificateur exporté correspond à un *lien utilisable*, et qu'un identificateur importé correspond à un *lien à satisfaire*.

A- Expliquez, en une 1/2 page, l'intérêt d'un tel outil pour le programmeur.

B- Parmi les enregistrements d'un module, quels sont ceux qui sont nécessaires pour cet outil, et pourquoi?

C- Proposer un programme qui, pour un identificateur donné, imprime la liste des modules qui importent ou exportent cet identificateur.

D- Compléter le programme ci-dessus pour imprimer, pour chaque module qui importe l'identificateur, la liste des adresses relatives dans le module des emplacements où il est utilisé.

E- Décrire brièvement (1/2 page) comment il faudrait concevoir un programme qui imprime, pour tous les identificateurs exportés par les modules d'une liste, les noms des modules de cette liste qui importent cet identificateur.

Solution de l'exercice 2.7

2.7.1. Question A

Un outil d'impression des références croisées entre des modules permet de connaître qui utilise quoi.

- Si un identificateur exporté par un module désigne une variable, on peut ainsi connaître la liste des modules qui accèdent directement à cette variable. En phase de mise au point d'une application, par exemple lorsque la valeur n'est pas conforme à ce que elle devrait être, on peut savoir quels sont les modules qui peuvent être en cause. En cas de changement d'interprétation du contenu de cette variable, on peut savoir quels modules sont concernés par ce changement.
- Si un identificateur exporté par un module désigne un sous-programme, on peut ainsi connaître la liste des modules où se trouvent les appels à ce sous-programme. En phase de mise au point d'une application, par exemple lorsque l'on constate que les variables permanentes internes au module ne sont pas conformes, il faut déterminer quels sont les appels précédents qui ont conduit à cette non-conformité; l'outil permet alors de savoir quels sont les modules qui peuvent être en cause. En cas de changement de la spécification (ou du fonctionnement) de ce sous-programme, on peut savoir quels modules sont concernés par ce changement.
- Comme il permet de connaître les relations entre les modules, il peut être utilisé pour améliorer le découpage entre les modules, en regroupant ensemble les parties de modules ou les modules entiers qui ont de fortes interactions.
- Il peut être utilisé pour faciliter la construction des bibliothèques de modules lorsque l'ordre des modules est important pour l'éditeur de liens, et constater quels sont les modules qui doivent être après d'autres modules pour garantir leur inclusion.

2.7.2. Question B

Les enregistrements d'un module qui sont nécessaires sont les suivants:

- `entête` permet de connaître le nom du module,
- `lien_à_satisfaire` désigne un identificateur importé par le module,
- `lien_utilisable` désigne un identificateur exporté par le module,
- `fin_module` permet de savoir que l'on a fini de traiter un module.

L'enregistrement `code_avec_l_a_s` est utile pour la question D pour savoir à quel endroit d'un module un lien importé est utilisé par ce module.

Les autres enregistrements ne sont pas utiles.

2.7.3. Question C

Le programme pourrait être le suivant:

```
procédure références_croisées ;
var nom_module, identifcateur: t_identificateur;
procédure traiter_lien_à_satisfaire;
début
     si identificateur = enrg.identificateur_l_a_s alors
          imprimer ("module avec importation: ", nom_module );
     finsi;
fin;
procédure traiter_lien_utilisable;
début
     si identificateur = enrg.identificateur_l_u alors
          imprimer ("module avec exportation: ", nom_module );
     finsi;
fin;
```

```
{ début du programme }
début
     saisir (identificateur);
     premier_module;
     tantque il_y_a_des_modules faire
          lire_enrg;
          si enrg.nature <> entête alors erreur ("pas en début de module");
          finsi;
          nom_module := enrg.identificateur_module;
          tantque enrg.nature <> fin_module faire
               lire_enrg;
               cas enrg.nature dans
                    lien_à_satisfaire: traiter_lien_à_satisfaire;
                    lien_utilisable: traiter_lien_utilisable;
                    code_absolu, code_à_translater, code_avec_las,
                    adresse_lancement, fin_module: ;
                    autre: erreur ("module incorrect");
               fincas;
          fait;
      fait;
fin;
```

2.7.4. Question D

La modification consiste à prendre en compte les enregistrements de type `code_avec_l_a_s` pour imprimer la valeur de `adr_code_l_a_s` lorsqu'il s'agit du numéro de lien à satisfaire correspondant à l'identificateur.

```
procédure références_croisées_et_places ;
var nom_module, identifcateur: t_identificateur;
   nb_las, num_lien: entier;
procédure traiter_lien_à_satisfaire;
début nb_las := nb_las + 1;
       si identificateur = enrg.identificateur_l_a_s alors
          num_lien := nb_las;
          imprimer ("module avec importation: ", nom_module );
       finsi;
fin;
procédure traiter_lien_utilisable;
début si identificateur = enrg.identificateur_l_u alors
          imprimer ("module avec exportation: ", nom_module );
       finsi;
fin;
procédure traiter_code_avec_las;
début si num_lien <> 0 et enrg.numéro_l_a_s = num_lien alors
          imprimer ("lien importé, utilisé en: ", adr_code_l_a_s );
       finsi;
fin;
{ début du programme }
début saisir (identificateur);
       premier_module;
       tantque il_y_a_des_modules faire
          lire_enrg;
          si enrg.nature <> entête alors erreur ("pas en début de module");
          finsi;
          nom_module := enrg.identificateur_module;
          nb_las := 0;
          num_lien := 0;
          tantque enrg.nature <> fin_module faire
               lire_enrg;
               cas enrg.nature dans
                 lien_à_satisfaire: traiter_lien_à_satisfaire;
                 lien_utilisable: traiter_lien_utilisable;
                 code_avec_las: traiter_code_avec_las;
                 code_absolu, code_à_translater, adresse_lancement, fin_module: ;
                 autre: erreur ("module incorrect");
               fincas;
          fait;
     fait;
fin;
```

2.7.5. Question E

Contrairement aux questions précédentes, on ne connaît plus un identificateur unique, mais le travail doit être fait pour tous les identificateurs que l'on trouve comme *lien utilisable* dans un module quelconque. L'idée est alors d'utiliser une table, comme cela a été fait dans l'éditeur de lien. Cette table doit permettre d'associer à chaque identificateur présent la liste des modules qui l'exportent (il peut y avoir plusieurs modules qui exportent le même lien), et la liste des modules qui l'importent. Cette table doit être créée vide à l'initialisation. Chaque fois que l'on rencontre un lien dans un module, il faut rechercher l'identificateur correspondant dans la table, et l'ajouter s'il n'y est pas. Si le lien est utilisable, le nom du module est ajouté à la liste des modules exportant qui lui est associée. S'il est à satisfaire, le nom du module est ajouté à la liste associée des modules qui importent.

Lorsque le traitement de la liste des modules est terminé, il suffit d'imprimer le contenu de cette table.

Si l'on veut fournir les mêmes informations que dans la question D (mais ce n'était pas demandé), il faut de plus pour chacun des modules qui importent, associer la liste des adresses dans le code où le lien à satisfaire est utilisé.

2.8. Exemple d'édition de liens

Il s'agit essentiellement de jouer le rôle de l'éditeur de liens pour calculer les implantations des modules et les valeurs des liens utilisables.

On dispose d'un ensemble de modules définis comme suit:

```
module PROGRAMME    taille:        332
                    liens à satisfaire:     OUVRIR
                                            LIRE
                                            FERMER
                                            EDITER
                    adresse lancement:      133
module ETIQUETTE    taille:        128
                    liens utilisables:      NOM         0
                                            SOCIETE    32
                                            ADRESSE    64
                                            CODEPOST   96
                                            VILLE     101
module LECTURE      taille:        840
                    liens utilisables:      OUVRIR      0
                                            LIRE      340
                                            FERMER    732
                    liens à satisfaire:     NOM
                                            SOCIETE
                                            ADRESSE
                                            CODEPOST
                                            VILLE
module IMPRESSION   taille:        212
                    liens utilisables:      IMPRIMER    0
module EDITION      taille:        642
                    liens utilisables:      EDITER      0
                    liens à satisfaire:     NOM
                                            SOCIETE
                                            ADRESSE
                                            CODEPOST
                                            VILLE
                                            IMPRIMER
```

A- On effectue l'édition de liens des modules `PROGRAMME`, `ETIQUETTE`, `LECTURE`, `IMPRESSION` et `EDITION`. Donner, en justifiant brièvement votre réponse:

- les adresses d'implantations de ces modules,
- la taille totale du programme résultant,
- la table des liens après le premier passage,

- l'adresse de lancement du programme résultant.

B- On désire mettre les modules `ETIQUETTE`, `LECTURE`, `IMPRESSION` et `EDITION` dans une bibliothèque de nom `UTIL`. Dans quel ordre doit-on les mettre, pour obtenir l'édition de liens correcte, en fournissant le module `PROGRAMME` et la bibliothèque `UTIL` à l'éditeur de liens, sachant que ce dernier parcourt séquentiellement la bibliothèque, pour savoir s'il doit incorporer ou non les modules successifs qu'il rencontre. Expliquer le fonctionnement.

Solution de l'exercice 2.8

2.8.1. Question A

L'adresse d'implantation d'un module est 0 pour le premier, et pour les autres, le premier emplacement laissé libre par le module qui le précède, c'est-à-dire, la somme entre son adresse d'implantation et sa taille:

```
PROGRAMME        0
ETIQUETTE      332
LECTURE        460
IMPRESSION    1300
EDITION       1512
```

La taille totale du programme est la somme des tailles de tous les modules, ou encore l'adresse du premier emplacement laissé libre par le dernier module, c'est-à-dire, la somme entre son adresse d'implantation et sa taille, soit 2154.

La table des liens contient tous les identificateurs présents dans les modules, en tant que lien utilisable ou lien à satisfaire, ainsi que éventuellement les noms des modules. De plus, la table associe à tout identificateur rencontré comme lien utilisable dans un module, leur adresse dans le programme, obtenue en ajoutant à leur adresse dans ce module l'adresse d'implantation du module. Enfin aux noms de modules, la table associe l'adresse d'implantation du module.

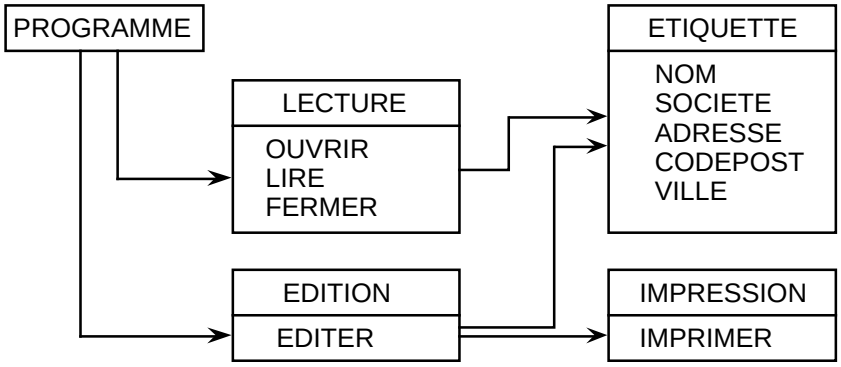
```
*  PROGRAMME        0
   OUVRIR         460
   LIRE           800
   FERMER        1192
   EDITER        1512
*  ETIQUETTE      332
   NOM            332
   SOCIETE        364
   ADRESSE        396
   CODEPOST       428
   VILLE          433
*  LECTURE        460
*  IMPRESSION    1300
   IMPRIMER      1300
*  EDITION       1512
```

Les lignes marquées par un "*" ne sont pas toujours dans la table, suivant que l'on utilise ou non la table pour mémoriser les adresses d'implantations des modules.

L'adresse de lancement est la valeur trouvée dans l'un des modules augmentée de l'adresse d'implantation de ce module: 133 + 0 = 133**.**

2.8.2. Question B

On a le graphe des références suivant:



L'éditeur de liens parcours séquentiellement la bibliothèque pour savoir s'il doit ou non incorporer les modules successifs. Au cours de ce parcours, pour chaque module en bibliothèque, il regarde donc si ce module, par ses liens utilisables, peut satisfaire un lien de la table, encore non satisfait. Si c'est le cas, il ajoute le module à la liste, et effectue le traitement du premier passage sur ce module, c'est-à-dire, définit son implantation, et ajoute ses liens à la table. Si ce module contient un lien à satisfaire qui n'était pas encore dans la table, il doit pouvoir être satisfait par un module qui le suit dans la bibliothèque. Le graphe de la figure montre alors que l'on doit respecter les ordres suivants:

```
LECTURE    précède    ETIQUETTE
EDITION    précède    ETIQUETTE
EDITION    précède    IMPRESSION
```

Les ordres possibles sont donc:

```
LECTURE EDITION ETIQUETTE IMPRESSION
LECTURE EDITION IMPRESSION ETIQUETTE
EDITION LECTURE ETIQUETTE IMPRESSION
EDITION LECTURE IMPRESSION ETIQUETTE
EDITION IMPRESSION LECTURE ETIQUETTE
```

2.9. Mise en œuvre de l'éditeur de liens

On dispose d'un système doté d'une mémoire virtuelle paginée de 16 Mo. par processus.

A- L'éditeur de liens de ce système ne propose pas le mécanisme de recouvrement. Donnez votre avis sur ce choix.

B- Le chargeur de ce système n'utilise aucune information de translation des programmes exécutables. Comment expliquez-vous ce choix?

C- L'éditeur de liens dispose de plusieurs options. Les énoncés ci-dessous sont extraits de la documentation.

-r génère les informations de translation dans le fichier résultat de telle sorte qu'il puisse être utilisé dans une autre édition de liens. Dans ce cas, il n'y a pas de diagnostic de lien non résolu.

-s enlève la table des symboles et les informations de translation du fichier résultat.

-x enlève les symboles locaux de la table des symboles du fichier résultat pour n'y conserver que les symboles externes.

-u prend l'argument suivant comme symbole non défini, et le met dans la table des symboles.

-o prend l'argument suivant comme nom du fichier résultat au lieu du nom implicite a.out.

C.1- Comment comprenez-vous les options `-s` et `-x`?

C.2- On dispose d'une bibliothèque `UTIL` contenant, dans l'ordre, les modules suivants:

```
EDITION       liens utilisables:      EDITER
              liens à satisfaire:     NOM, SOCIETE, ADRESSE,
                                      CODEPOST, VILLE, IMPRIMER
IMPRESSION    lien utilisable:        IMPRIMER
LECTURE       liens utilisables:      OUVRIR, LIRE, FERMER
              liens à satisfaire:     NOM, SOCIETE, ADRESSE,
                                      CODEPOST, VILLE, IMPRIMER
ETIQUETTE     liens utilisables:      NOM, SOCIETE, ADRESSE,
                                      CODEPOST, VILLE
```

Quel est, à votre avis, le résultat de la commande:

```
ld -r -u EDITER UTIL -o EDIETIQ
```

Solution de l'exercice 2.9

2.9.1. Question A

Le mécanisme de recouvrement est utilisé pour permettre de réduire la taille effective d'un programme en faisant en sorte que certaines parties puissent se «recouvrir», c'est-à-dire occupent les mêmes emplacements, à des moments différents. C'est donc un mécanisme pour permettre de mettre dans un espace mémoire limité un programme qui est beaucoup plus grand que cet espace. Dans ce système, un processus disposant d'une mémoire virtuelle pouvant atteindre 16 Mo., les concepteurs ont jugé que ceci était largement suffisant pour les besoins des utilisateurs, et qu'il était inutile de prévoir le cas des programmes qui déborderaient cette limite. Notons que la notion même de mémoire virtuelle induit déjà implicitement un «recouvrement» dans la mémoire réelle.

2.9.2. Question B

Chaque processus disposant de sa propre mémoire virtuelle paginée indépendante de celles des autres processus, l'éditeur de liens peut placer les modules du programme où bon lui semble dans cette mémoire virtuelle. Le chargement du programme consistera à mettre le programme dans la mémoire virtuelle du processus là où l'éditeur de liens l'a prévu, sans qu'il y ait translation de ce programme. Notons que les mécanismes de pagination eux-mêmes assurent une translation implicite des adresses en fonction de la case allouée à la page accédée.

2.9.3. Question C

2.9.3.1. Question C.1

Un chargeur n'utilise en général pas les tables de symboles, quels qu'ils soient, mais ces informations peuvent être utilisées par un metteur au point symbolique pour faciliter le travail de l'utilisateur. Ce chargeur n'utilisant aucune information de translation, de telles informations ne peuvent servir qu'à l'éditeur de liens lui-même. Si elles restent dans le fichier résultat, c'est que ce fichier résultat peut éventuellement être réutilisé pour une autre édition de liens ultérieure (il s'agit donc d'une édition de liens partielle). On peut donc en déduire les interprétations suivantes des options:

`-s` L'édition de liens courante crée un programme exécutable complet et opérationnel. L'option permet alors de supprimer du fichier les informations qui ne sont pas utiles au chargement. L'absence de cette option permet de conserver la possibilité de compléter l'édition de liens avec d'autres modules ultérieurement, ou de faire de la mise au point symbolique sur le programme.

`-x` On peut supposer que cette option n'est pas utilisée avec la précédente, puisque la précédente enlève complètement la table des symboles. Son utilisation permet de conserver les symboles externes (liens utilisables et liens à satisfaire), ainsi que les informations de translation. Le fichier résultat peut donc faire l'objet d'une édition de liens ultérieure. Le fait de supprimer les symboles locaux, qui ne servent qu'au metteur au point signifie que l'utilisateur considère que les modules qui font l'objet de cette édition de liens sont opérationnels.

2.9.3.2. Question C.2

La commande

```
ld -r -u EDITER UTIL -o EDIETIQ
```

peut s'interpréter comme suit:

`-r` Il y a production des informations de translation, permettant une édition de liens ultérieure du résultat. L'absence de diagnostic de lien non résolu est naturel dans ce cas, puisque ces liens pourront être satisfaits par cette édition ultérieure.

`-u EDITER` Le symbole `EDITER` est mis dans la table des liens (initialement vide), en tant que lien à satisfaire. Notons qu'il est seul dans la table à ce moment.

`UTIL` Il s'agit d'une bibliothèque. L'éditeur de liens va donc la parcourir, pour déterminer les modules qu'elle contient et qui doivent être ajoutés au résultat. Quelle que soit la méthode utilisée (voir cours), l'éditeur de liens, partant du seul lien `EDITER` de la table, ajoute le module `EDITION` qui le contient comme lien utilisable. Ceci entraîne également l'entrée dans la table des liens à satisfaire `NOM`, `SOCIETE`, `ADRESSE`, `CODEPOST`, `VILLE`, `IMPRIMER`. La présence du lien à satisfaire `IMPRIMER` dans la table entraînera l'adjonction du module `IMPRESSION`, et celle de `NOM`, `SOCIETE`, `ADRESSE`, `CODEPOST`, `VILLE` entraînera celle du module `ETIQUETTE`.

`-o EDIETIQ` Le résultat de l'édition de liens est mis dans le fichier de nom `EDIETIQ`. Ce résultat est donc l'édition de liens des modules `EDITION`, `IMPRESSION` et `ETIQUETTE`. Ce résultat ne constitue pas un programme, mais relie ensemble ces trois modules. La réutilisation de ce résultat dans une édition de liens ultérieure en sera simplifiée d'autant.

2.10. Édition de liens de 5 modules

On dispose de 5 modules compilés, mémorisés dans les fichiers `cmpdisk.o`, `cmpfile.o`, `disk_io.o`, `lsbrk.o`, `printstr.o`. Ces 5 modules constituent la base d'un programme, qui nécessite en plus des modules de bibliothèque.

On dispose d'un premier outil, appelé `listliens`, qui imprime les liens des modules, en indiquant leur type ainsi que éventuellement leur valeur. Le résultat de l'application de cet outil sur les 5 modules est donné en table 1, où `las` signifie lien à satisfaire et `lu` lien utilisable.

On dispose également d'un second outil, appelé `taille`, qui donne la taille d'un module objet, c'est-à-dire le nombre d'emplacements occupés par ce module. Le résultat de l'application de cet outil sur les 5 modules est donné en table 2.

On se propose de faire l'édition des liens des 5 modules dans l'ordre suivant :

```
cmpdisk.o, cmpfile.o, disk_io.o, lsbrk.o, printstr.o.
```

A- Quel est la place et le rôle de l'éditeur de liens dans la chaîne de production de programmes.

B- En supposant que l'éditeur de liens suit l'algorithme présenté dans le livre, définir les adresses d'implantations des modules, en justifiant votre réponse.

C- Donner le contenu de la table des liens après prise en compte des liens des 5 modules (premier passage). Expliquez votre raisonnement.

cmpdisk.o:	las	bloc_transfer_dr	
	las	close_printer	
	las	compare_file	
	lu	compare_hierarch	1862
	las	exit	
	lu	exit_prog	2820
	lu	file_error	422
	las	free	
	las	get_memory	
	lu	main	2460
	las	open_drive	
	las	open_printer	
	las	print_string	
cmpfile.o:	las	bloc_transfer_dr	
	lu	compare_file	200
	las	file_error	
disk_io.o:	lu	bloc_transfer_dr	2118
	las	exit	
	las	exit_prog	
	las	get_memory	
	lu	open_drive	236
	las	print_string	
lsbrk.o:	las	exit_prog	
	lu	get_memory	0
	las	lmalloc	
	las	print_string	
printstr.o:	lu	ask_confirm	758
	lu	close_printer	1028
	las	getchar	
	lu	open_printer	996
	lu	print_string	340
	las	write	

table 1. résultat de "`listliens`"

cmpdisk.o	3376
cmpfile.o	644
disk_io.o	3862
lsbrk.o	82
printstr.o	1196

table 2. résultat de "`taille`"

D- Donner la liste des références croisées en expliquant votre raisonnement.

E- Certains liens ne sont pas définis. Donner lesquels. Comment seront-ils satisfaits?

F- Indiquer les liens qui ne sont à satisfaire dans aucun module. Sachant que le module du fichier `cmpdisk.o` est le programme principal dont l'adresse de lancement est `main`, et que les autres modules sont des sous programmes de service pouvant être utilisés dans d'autres programmes, précisez pour chacun d'eux s'il vous paraît logique de les trouver en tant que lien utilisable.

G- Dans quel ordre faudrait-il mettre les modules dans une bibliothèque, sachant que l'éditeur de lien la parcourt séquentiellement.

H- En vous inspirant de la question F, voyez-vous une utilisation particulière de l'outil `listliens`?

Solution de l'exercice 2.10

2.10.1. Question A

La construction de programmes peut devenir relativement complexe lorsque la taille du programme devient importante. Pour cela, on est amené à découper les programmes en modules. Chacun des modules peut être compilé de façon séparée. L'éditeur de liens a pour but de constituer un programme exécutable à partir d'un ensemble de modules. Le chargeur peut ensuite mettre en mémoire ce programme exécutable pour l'exécuter.

Le rôle de l'éditeur de liens se présente sous plusieurs aspects :

1. Assurer une fonction de placement des modules dans le programme exécutable. Ces modules ont été compilés de façon à pouvoir s'exécuter à un certain emplacement mémoire. Il s'agit de faire en sorte que tous les modules occupent des espaces disjoints à l'exécution (sauf si on désire utiliser des techniques de recouvrement qui ne seront pas abordées ici). La méthode la plus simple consistera à les mettre les uns derrière les autres en tenant compte de leur taille.
2. Translater les modules de façon à permettre leur bon fonctionnement dans l'espace qui leur a été attribué.
3. Relier les modules entre eux. Si l'on désire faire un programme avec un ensemble de modules, c'est que ces modules doivent coopérer pour cette exécution. Ceci veut dire que des sous programmes ou des variables appartenant à un module doivent pouvoir être utilisé par un autre module. Pour cela on a introduit la notion de lien représentant un tel depuis un module vers un autre. La partie du lien qui se trouve dans le module qui cherche à accéder à l'objet s'appelle le lien à satisfaire et la partie du lien qui est situé dans le module qui fourni l'objet s'appelle le lien utilisable. Relier consiste à associer les liens à satisfaire aux liens utilisables correspondants.
4. Compléter la liste des modules avec des modules en bibliothèque. Une liste initiale de modules est fournie à l'éditeur de liens qui relie les liens à satisfaire aux liens utilisables correspondants. Il est possible que certains liens à satisfaire ne correspondent à aucun lien utilisable d'un module de la liste. L'éditeur de liens utilisent ces liens à satisfaire pour rechercher dans les bibliothèques de modules s'il peut en trouver un qui contienne ce lien comme lien utilisable. Lorsqu'il en trouve, il ajoute ce module à la liste.

2.10.2. Question B

En faisant l'édition de liens des 5 modules dans l'ordre indiqué, les modules seront placés dans le programme exécutable dans cet ordre, les uns derrière les autres, en commençant à 0 pour le premier.

`cmpdisk.o`	`0`
`cmpfile.o`	`3376`
`disk_io.o`	`4020`
`lsbrk.o`	`7882`
`printstr.o`	`7964`

carte d'implantation des modules

Les modules de bibliothèques éventuels seront mis après, c'est-à-dire commenceront en 9160.

2.10.3. Question C.

Au cours du premier passage, l'éditeur de lien fixe la carte d'implantation des modules comme indiqué dans la question précédente. En même temps, il prend en compte l'ensemble des liens de ces modules, pour les mettre dans la table des liens. Pour chacun des liens trouvés dans un module, on effectue le traitement suivant :

S'il s'agit d'un lien utilisable, on modifie sa valeur en lui ajoutant l'adresse de début d'implantation du module, et on le recherche dans la table.

- S'il y est comme lien utilisable, il y a double définition et on ne modifie pas la table.
- S'il y est comme lien à satisfaire, on change son état en lien utilisable et on y met sa valeur.
- S'il n'existe pas dans la table, on l'ajoute comme lien utilisable avec sa valeur.

1. S'il s'agit d'un lien à satisfaire, on le recherche dans la table.
   - S'il n'existe pas, il est rajouté comme lien à satisfaire.
   - S'il existe, la table n'est pas modifiée.

Appliquons ce traitement aux liens des modules fournis. Les liens du premier module sont entrés dans la table, soit comme lien à satisfaire, soit comme lien utilisable avec sa valeur (à laquelle on ajoute 0).

```
las     bloc_transfer_dr
las     close_printer
las     compare_file
lu      compare_hierarch        1862
las     exit
lu      exit_prog               2820
lu      file_error              422
```

	las	free	
	las	get_memory	
	lu	main	2460
	las	open_drive	
	las	open_printer	
	las	print_string	
Le deuxième module conduit simplement à la modification du lien `compare_file`, qui devient utilisable, avec la valeur 3376+200=3576. Le troisième module, quant à lui, conduit à la transformation en liens utilisables d'une part de `bloc_transfer_dr` dont la valeur est maintenant 4020+2118=6138 et d'autre part de `open_drive` dont la valeur est maintenant 4020+236=4256.	las	bloc_transfer_dr	6138
	las	close_printer	
	las	compare_file	3576
	lu	compare_hierarch	1862
	las	exit	
	lu	exit_prog	2820
	lu	file_error	422
	las	free	
	las	get_memory	
	lu	main	2460
	las	open_drive	4256
	las	open_printer	
	las	print_string	
Le quatrième module transforme le lien `get_memory` en lien utilisable, avec la valeur 7882+0=7882, et ajoute le lien `lmalloc` comme lien à satisfaire.	las	bloc_transfer_dr	6138
	las	close_printer	
	las	compare_file	3576
	lu	compare_hierarch	1862
	las	exit	
	lu	exit_prog	2820
	lu	file_error	422
	las	free	
	las	get_memory	7882
	lu	main	2460
	las	open_drive	4256
	las	open_printer	
	las	print_string	
	las	lmalloc	

Enfin, le cinquième module conduit à la transformation en liens utilisables de `close_printer` avec la valeur 7964+1028=8992, `open_printer` avec la valeur 7964+996=8960 et `print_string` avec la valeur 7964+340=8304. De plus, sont ajoutés à la table le lien `ask_confirm` comme lien utilisable avec la valeur 7964+758= 8722, et les liens `get_char` et `write` comme liens à satisfaire.	las	bloc_transfer_dr	6138
	las	close_printer	8992
	las	compare_file	3576
	lu	compare_hierarch	1862
	las	exit	
	lu	exit_prog	2820
	lu	file_error	422
	las	free	
	las	get_memory	7882
	lu	main	2460
	las	open_drive	4256
	las	open_printer	8960
	las	print_string	8304
	las	lmalloc	
	lu	ask_confirm	8722
	las	getchar	
	las	write	

2.10.4. Question D.

La liste des références croisées est obtenue en indiquant pour chaque lien, le module dans lequel il est défini en tant que lien utilisable (normalement il y en a au plus un), et la liste des modules dans lequel il apparaît en tant que lien à satisfaire.

lien	utilisable	à satisfaire
`bloc_transfer_dr`	`disk_io.o`	`cmpdisk.o, cmpfile.o`
`close_printer`	`printstr.o`	`cmpdisk.o`
`compare_file`	`cmpfile.o`	`cmpdisk.o`
`compare_hierarch`	`cmpdisk.o`	
`exit`		`cmpdisk.o, disk_io.o`
`exit_prog`	`cmpdisk.o`	`disk_io.o, lsbrk.o`
`file_error`	`cmpdisk.o`	`cmpfile.o`
`free`		`cmpdisk.o`
`get_memory`	`lsbrk.o`	`cmpdisk.o, disk_io.o`
`main`	`cmpdisk.o`	
`open_drive`	`disk_io.o`	`cmpdisk.o`
`open_printer`	`printstr.o`	`cmpdisk.o`
`print_string`	`printstr.o`	`cmpdisk.o, disk_io.o, lsbrk.o`
`lmalloc`		`lsbrk.o`
`ask_confirm`	`printstr.o`	
`getchar`		`printstr.o`
`write`		`printstr.o`

2.10.5. Question E.

Les liens qui restent à satisfaire dans la table des liens après traitement des 5 modules ne sont pas définis, puisqu'ils n'ont pas été trouvés utilisables dans l'un de ces modules. Il s'agit de `exit`, `free`, `lmalloc`, `getchar` et `write`. Ces liens devront être satisfaits au moyen de modules de bibliothèques. De fait, il s'agit de procédures courantes, qui font appel au système :

`exit` permet de terminer l'exécution du programme,

`free` et `lmalloc` servent à la gestion dynamique de la mémoire,

`getchar` et `write` sont des procédures d'entrées sorties.

L'éditeur de liens recherchera les modules de bibliothèque qui contiennent comme lien utilisable ces liens restant à satisfaire dans la table.

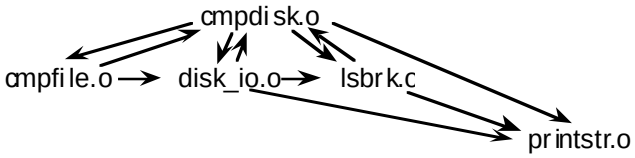
2.10.6. Question F.

La liste des références croisées de la question D montre que `compare_hierarch`, `main` et `ask_confirm` sont définis comme lien utilisables dans un module, mais ne sont mentionnés comme lien à satisfaire dans aucun.

- Le lien `main` étant l'adresse de lancement, il doit être utilisable pour que l'éditeur de lien puisse définir cette adresse de lancement dans le programme exécutable.
- Le lien `ask_confirm` étant dans un module qui peut être utilisé par d'autres programmes, il est probable que ce module offre des possibilités qui ne sont pas utilisées dans ce programme, mais peuvent l'être dans d'autres.
- Le lien `compare_hierarch` se trouve dans le programme principal, sans être utilisé dans un autre module. On peut en déduire que le fait que ce lien soit utilisable est inutile, et devrait être corrigé : le lien devrait rester interne au module `cmpdisk.o`.

2.10.7. Question G.

Le tableau des références croisées permet de construire le graphe des liens entre les modules. On trace un arc d'un module A vers un module B, s'il y a un lien à satisfaire dans A qui est utilisable dans B. Notons que dans ce cas, le module A devrait être placé avant le module B en bibliothèque pour que l'éditeur de liens ajoute B s'il a déjà ajouté A.



On peut constater sur le graphe ainsi obtenu qu'il y a plusieurs cycles. Ainsi le module `cmpdisk.o` doit être placé avant le module `cmpfile.o`, à cause du lien `compare_file`, mais ce module

`cmpfile.o` doit être placé avant `cmpdisk.o` à cause du lien `file_error`. Cependant, le module `cmpdisk.o` est un programme principal; il ne devrait donc pas faire l'objet d'un chargement automatique, mais être le module de départ qui entraîne le chargement des autres. Il doit donc être mis en premier. Le reste du graphe impose alors la séquence complète, qui est donc : `cmpdisk.o`, `cmpfile.o`, `disk_io.o`, `lsbrk.o`, `printstr.o`.

2.10.8. Question H.

Lors de la question F, nous avons pu constater que notre programme principal exportait sans doute inutilement le lien `compare_hierarch`. L'outil `listliens` en listant les liens utilisables et les liens à satisfaire permet au programmeur de vérifier qu'un module exporte (liens utilisables) et importe (liens à satisfaire) juste ce qui doit l'être.

2.11. Appel d'un compilateur C

La notice d'utilisation succincte d'un compilateur C est la suivante :

```
cc compile les fichiers se terminant par '.c', assemble ceux qui se terminent
par '.s' et passe les options non mentionnées ci-dessous à l'éditeur de lien
'ld'.
utilisation : cc [options] fichier ...
options :
   -c compile seulement, sans édition de liens; les fichiers objets ont pour nom
       celui obtenu en remplaçant '.c' ou '.s' par '.o'
   -Dchaine[=valeur] demande au préprocesseur de définir 'chaine' avec 'valeur'
   -Uchaine demande au préprocesseur de retirer toute définition initiale de
       'chaine'
   -Ichemin demande de rechercher les fichiers 'include'dans le répertoire
       'chemin'
   -E demande de lancer seulement le préprocesseur et d'écrire sa sortie sur
       stdout
   -S demande de compiler en langage d'assemblage, sans assembler; les fichiers
       résultats ont pour nom celui obtenu en remplaçant '.c' par '.s'
   -g demande de produire les informations de mise au point
   -O2  demande d'optimiser le code
```

A- Décrire brièvement les étapes essentielles de la production de programme.

B- Dans ce contexte, expliquer l'option `-C` et la commande suivante :

```
cc -c toto.c
```

C- Décrire le rôle d'un préprocesseur et expliquez les options `-D`, `-U`, `-I`. Comment interprétez-vous les deux commandes suivantes :

```
cc -DBUFSIZE=2048 -DUNIX -c schmilblick.c
cc -DBUFSIZE=1024 -DMS_DOS -c schmilblick.c
```

D- Décrire brièvement les fonctions du metteur au point, et expliquer le rôle de l'option `-g` dans la chaîne de production de programme.

E- Quel est le rôle de l'option `-O2`? Expliquer la commande suivante :

```
cc -O2 toto.c -o toto
```

F- Expliquer l'utilité des options `-E` et `-S`.

Solution de l'exercice 2.11.

2.11.1. Question A.

La construction d'un programme se fait souvent de façon modulaire. En effet, dès qu'il est d'une certaine taille, il est nécessaire de le morceler, pour manipuler des entités cohérentes et dont on

maîtrise la complexité. Le découpage modulaire permet aussi la réutilisation des modules dans différents programmes. Ces modules peuvent être écrits dans différents langages (évolués ou d'assemblage). Chacun d'eux doit alors être traduit pour donner un module objet de sémantique équivalente, mais dans un langage intermédiaire proche du langage de la machine. C'est le rôle des compilateurs ou des assembleurs d'assurer cette traduction. Ces modules objets doivent ensuite être regroupés pour constituer un programme exécutable. C'est le rôle de l'éditeur de liens d'une part de translater les modules pour qu'ils puissent résider dans des emplacements de mémoire centrale différents, et d'autre part de résoudre les liens entre les modules, en associant les liens à satisfaire d'un module avec un lien utilisable situé dans un autre module. Enfin, le programme exécutable doit être mis en mémoire centrale pour son exécution. C'est le rôle du chargeur d'assurer cette mise en mémoire (translation finale éventuelle) et de préparer son environnement d'exécution (construction de la machine abstraite).

2.11.2. Question B.

La notice d'utilisation précise dès le début que `cc` assure la compilation des modules écrits en langage C, l'assemblage des modules en langage d'assemblage et l'édition de liens de l'ensemble des modules. En d'autres termes, il s'agit d'un programme qui enchaîne l'ensemble des activités de la chaîne de production de programme, à l'exception du chargement. Cela facilite la construction des programmes constitués de peu de modules.

L'option permet d'empêcher cet enchaînement lorsque le programme final est constitué de plusieurs modules qui sont compilés séparément. Elle est particulièrement utile lorsqu'on utilise des outils automatiques comme le `make` qui détermine les opérations strictement nécessaires à la construction d'un programme en évitant celles qui ne sont pas utiles.

Ainsi la commande `cc -c toto.c` est une demande de compilation sans édition de liens du fichier `toto.c`. Le résultat sera mis dans le fichier `toto.o`.

2.11.3. Question C.

Le rôle du préprocesseur est de faire un traitement sur le texte source lui-même avant la compilation proprement dite. Il permet de paramétrer le texte source, et de fixer les valeurs des paramètres au moment de la compilation. Un même texte source peut alors être utilisé dans diverses situations, comme on le montrera avec les deux commandes. En particulier, le préprocesseur réalise essentiellement trois modifications :

- remplacement des chaînes qui lui ont été définies par la valeur correspondante,
- insertion ou suppression de portions de textes, suivant qu'une chaîne est définie ou non,
- inclusion d'un autre fichier, dans le texte.

L'option `-D` permet précisément de donner une définition à une chaîne, valable pour cette compilation, sans modifier le fichier source lui-même. Evidemment, ceci n'a de sens que si le fichier source fait référence à cette chaîne.

L'option `-U` permet de supprimer une définition présente dans le fichier, pour cette compilation.

L'option `-I` permet de préciser le répertoire où il y a lieu de rechercher les fichiers qui sont mentionnés dans une clause `'#include'` du fichier source. C'est particulièrement utile pour permettre de regrouper dans un répertoire spécifique les fichiers utilisés en inclusion.

Les deux commandes suivantes concernent la compilation du même fichier en langage C.

```
cc -DBUFSIZE=2048 -DUNIX -c schmilblick.c
cc -DBUFSIZE=1024 -DMS_DOS -c schmilblick.c
```

Elles définissent d'une part la chaîne `BUFSIZE` comme une constante avec des valeurs différentes dans les deux cas et d'autre part deux chaînes différentes sans leur donner de valeur. La première définit la chaîne `UNIX`, et la seconde la chaîne `MS_DOS`. Ces deux chaînes seront plutôt utilisées, dans le fichier, pour insérer ou supprimer du texte. Intuitivement, le fichier `schmilblick.c` contient un module qui est paramétré pour être compilé soit pour un système `UNIX`, soit pour un système `MS_DOS`. La première commande produira le module objet adapté à `UNIX`, alors que la seconde

produira celui adapté à `MS_DOS`. Répétons que ceci n'a de sens que si le fichier contient des clauses `'#ifdef UNIX'` ou `'#ifdef MS_DOS'`.

2.11.4. Question D.

Des outils ont été construits pour permettre la mise au point d'un programme. Ces outils sont appelés metteurs au point. Les fonctionnalités essentielles sont les suivantes :

- mettre ou supprimer des points d'arrêt dans le programme en cours de test,
- consulter ou modifier les variables lorsque le programme est arrêté,
- relancer l'exécution du programme après un arrêt,
- exécuter le programme en pas à pas.

Il est plus facile pour l'utilisateur de désigner les points d'arrêt ou les variables sous une forme proche du langage dans lequel il a écrit les modules, c'est-à-dire sous forme symbolique, en utilisant les identificateurs des modules. Pour cela, il faut que le metteur au point puisse faire la transformation entre ces identificateurs et les adresses en mémoire centrale. Ceci nécessite la coopération entre les constituants essentiels de la chaîne de production de programme. L'option `-g` demande au compilateur de rajouter dans le module objet les informations qui sont utiles au metteur au point, comme par exemple, la table des symboles et la table des numéros de lignes. Ces informations seront modifiées par l'éditeur de liens qui tiendra compte des translations effectuées, et les mettra dans le fichier qui contient le programme exécutable, où le metteur au point ira les chercher.

2.11.5. Question E.

L'écriture des modules dans un langage évolué implique une description de haut niveau, éloignée des contraintes matérielles. L'optimisation est une opération effectuée par le compilateur sur le code produit de façon à en améliorer l'efficacité. C'est une opération coûteuse en temps, car elle est complexe. Elle est inutile sur des modules qui ne sont pas complètement au point. Elle peut d'ailleurs perturber la mise au point car elle peut entraîner des déplacements d'instructions et des suppressions de variables. C'est pourquoi elle doit être optionnelle.

La commande `cc -O2 toto.c -o toto` est une demande de compilation du fichier `toto.c`. Le résultat sera mis dans le fichier `toto.o`. Cette compilation doit se faire avec optimisation de code. Ensuite l'éditeur de liens est appelé sur le fichier `toto.o`, avec les options `-o toto`. On peut penser que cette option permet de donner le `toto` au fichier résultat de l'édition de liens.

2.11.6. Question F.

L'option `-E` permet de consulter le résultat du traitement du texte source par le préprocesseur. On peut y voir deux intérêts. D'une part, lorsque le texte source contient des commandes complexes au préprocesseur, cela permet de vérifier leur bonne interprétation. D'autre part, cela permet d'utiliser le préprocesseur lui-même sur des textes quelconques, qui ne sont pas des modules en langages C. Si on désire conserver le résultat du traitement, il suffira alors de rediriger la sortie `stdout` vers un fichier.

L'option `-S` permet d'obtenir le résultat de la compilation en langage d'assemblage. Son utilité est en général réservé à des spécialistes. D'une part, Elle permet de se servir du compilateur comme première étape de préparation d'un module en langage d'assemblage, qui est ensuite adapté et modifié selon les besoins. Si le compilateur est doté d'un bon optimiseur, cette possibilité est souvent d'un intérêt limité. D'autre part elle permet de connaître le code produit par le compilateur sous une forme lisible, ce qui peut s'avérer parfois utile dans des situations de mise au point difficiles.

2.12. Étude du préprocesseur

On dispose d’un certain nombre de fichiers sources, dont des extraits sont donnés en fin d’exercice. Ils définissent une collection de modules qui, avec d’autres modules de bibliothèque, constituent un programme appelé `analyseur`.

A- Comme on le voit, ces fichiers contiennent des directives qui doivent faire l'objet d'un traitement par un préprocesseur[1] avant d'être compilé. Expliquer le fonctionnement des directives `#include` et `#ifdef ... #endif`, dans les fichiers « `.h` ». Donner le résultat du traitement du préprocesseur sur chacun des fichiers « `.c` ».

B- Le langage de programmation, dans lequel sont écrits les modules, permet de distinguer 3 sortes d'objets (sous-programmes ou variables) suivant qu'un « préfixe » est utilisé ou non devant la déclaration:

- sans préfixe, la variable ou le sous-programme est exporté (il est « public »)
- avec le préfixe `extern`, la variable ou le sous-programme est importé (il est « externe »)
- avec le préfixe `static`, la variable ou le sous-programme est privé et n'est pas accessible aux autres modules.

B.1- Pour chacun des trois modules fournis, donner les liens utilisables et les liens à satisfaire.

B.2- Donner la ou les raisons qui expliquent qu'une seule déclaration de sous-programme a été mise dans le fichier `syntax.h`, et non toutes.

C- Pour obtenir le programme `analyseur`, il faut exécuter les commandes suivantes, correspondant à la compilation des modules suivie de l'édition des liens:

```
cc -c lex.c
cc -c syntax.c
cc -c analyseur.c
ld -o analyseur lex.o syntax.o analyseur.o
```

Définir le graphe de dépendance relatif au programme `analyseur`, et construire le fichier `Makefile` minimal correspondant.

[1] Rappel: la notion de préprocesseur a été défini en cours; voir le livre/polycopié page 76.

```
analyseur.h

#ifdef MAIN
 char gLigne [1024];
 int SuiteSymbol [1024];
 int ErrLexicale = 0;
#else
 externe char gLigne [];
 extern int SuiteSymbol [];
 extern int ErrLexicale;
#endif
```

```
lex.h

#ifdef LEX
#else
 extern void AnalyseLexicale (void);
#endif
```

```
syntax.h

#ifdef SYNTAX
#else
 extern int AnalyseSyntaxique (int *);
#endif
```

```
syntax.c

#define SYNTAX 0
#include "analyseur.h"
#include "syntax.h"

int AnalyseSyntaxique (int *i)
{
...
}

static int Instruction (int *i)
{
...
}

static int Expression (int *i)
{
...
}

static int Facteur (int *i)
{
...
}
...
```

```
lex.c

#define LEX 0
#include "analyseur.h"
#include "lex.h"

void AnalyseLexicale ()
{
...
}
```

```
analyseur.c

#define MAIN 0
#include "analyseur.h"
#include "lex.h"
#include "syntax.h"

main ()
{
 for (;;) {
  ...
  AnalyseLexicale ();
  if (!ErrLexicale) {
   ...
   if (AnalyseSyntaxique (&i))
   ...
  }
 }
}
```

Solution de l’exercice 2.12

2.12.1. Question A

La directive `#include` a pour effet d’inclure le fichier désigné en argument dans le texte, à la place de la directive. La directive `#ifdef .(1). #else .(2). #endif` a pour effet de prendre en compte dans le texte résultat, soit les lignes de la partie `(1)`, soit celles de la partie `(2)` selon que l’élément associé à la condition a fit l’objet d’une directive `#define` ou non.

Dans le fichier `lex.c`, qui définit `LEX` sans définir `MAIN`, cela a pour conséquence de prendre la partie `(2)` du fichier `analyseur.h` et la partie `(1)` du fichier `lex.h` qui est vide. On obtient donc :

```
extern char gLigne [] ;
extern int SuiteSymbol [] ;
extern int ErrLexicale ;
```

Dans le fichier `syntax.c`, qui définit `SYNTAX` sans définir `MAIN`, cela a pour conséquence de prendre la partie `(2)` du fichier `analyseur.h` et la partie `(1)` du fichier `syntax.h` qui est vide. On obtient donc :

```
extern char gLigne [] ;
extern int SuiteSymbol [] ;
extern int ErrLexicale ;
```

Dans le fichier `analyseur.c`, qui définit `MAIN` sans définir `LEX` ni `SYNTAX`, cela a pour conséquence de prendre la partie `(1)` du fichier `analyseur.h` et les parties `(2)` des fichier `lex.h` et `syntax.h`. On obtient donc :

```
char gLigne [1024] ;
int SuiteSymbol [1024] ;
int ErrLexicale = 0 ;
extern void AnalyseLexicale (void) ;
extern int AnalyseSyntaxique (int *) ;
```

2.12.2. Question B

2.12.2.1. Question B.1

Les liens utilisables d'un module sont les variables ou sous programmes déclarés sans préfixe particulier, alors que les liens à satisfaire sont ceux préfixés par `extern`. Ceux préfixés par `static` ne sont ni utilisables ni à satisfaire. Le tableau ci-dessous donne les liens utilisables et les liens à satisfaire des trois modules.

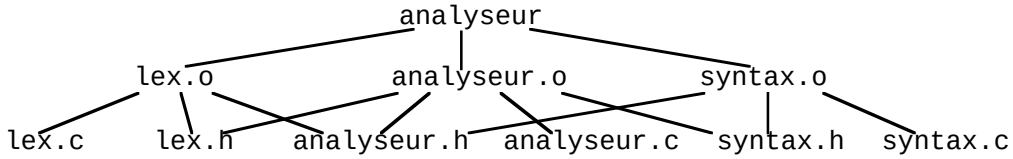
	liens utilisables	liens à satisfaire
lex.o	AnalyseLexicale	gLigne SuiteSymbol ErrLexicale
syntax.o	AnalyseSyntaxique	gLigne SuiteSymbol ErrLexicale
analyseur.o	gLigne SuiteSymbol ErrLexicale main	AnalyseLexicale AnalyseSyntaxique

2.12.2.2. Question B.2

Une seule déclaration a été mise dans `syntax.h`, car c'est en fait le point d'entrée de l'analyse syntaxique. Les autres sous programmes sont appelés par celui-là au fur et à mesure de l'analyse (voir l'exercice 2.1).

2.12.3. Question C

D'après ce que l'on peut déduire des textes sources fournis, il y a un programme principal `analyseur.c` qui pourrait donner lieu au programme exécutable appelé `analyseur`. Ce programme est construit à partir des trois modules objets énoncés ci-dessus. En tenant compte des directives `#include`, qui influent sur le résultat de la compilation, le graphe de dépendance est donc le suivant.



Le fichier `Makefile` le plus simple pourrait être défini comme suit :

```
analyseur : lex.o analyseur.o syntax.o
     ld –o analyseur lex.o syntax.o analyseur.o

lex.o : lex.c lex.h analyseur.h
     cc –c lex.c
syntax.o : syntax.c syntax.h analyseur.h
     cc –c syntax.c
analyseur.o : analyseur.c lex.h syntax.h analyseur.h
     cc –c analyseur.c
```

En tenant compte de la règle de suffixe habituelle, (un « .o » dépend d'un « .c » et la reconstruction fait appel au compilateur C), on pourrait aussi écrire :

```
analyseur : lex.o analyseur.o syntax.o
     ld –o analyseur lex.o syntax.o analyseur.o

lex.o : lex.h analyseur.h
syntax.o : syntax.h analyseur.h
analyseur.o : lex.h syntax.h analyseur.h
```

2.13. Le "make"

Vous recevez les sources du logiciel "`make`". Vous copiez les fichiers dans un répertoire initialement vide, et vous imprimez le contenu de ce répertoire. On restera dans ce répertoire pendant tout l'exercice. Constatant qu'il comporte un fichier de nom `Makefile`, vous imprimez ce fichier. Ces impressions sont données ci-dessous.

A- Construire le graphe de dépendances entre les fichiers.

B- Vous ne disposez pas encore du programme "`make`". Que faut-il faire pour le créer?

C- A la suite de A, vous disposez maintenant de ce programme. Vous en demandez l'exécution. Expliquez ce qui se passe.

D- Vous modifiez le fichier "`basepage.h`", et lancez la commande "`make install`". Énoncez toutes les commandes qui sont exécutées.

E- Vous créez le répertoire "`/usr/local/src/make`", puis lancez la commande "`make backup`". Que se passe-t-il?

F- Vous lancez la commande "`make erase`". Que se passe-t-il? Disposez-vous encore du programme "`make`"?

Contenu du répertoire:

```
-rw------- 1 u           2071 Jun 15 10:01 basepage.h
-rw------- 1 u           6841 Jun 15 10:01 check.c
-rw------- 1 u           3448 Jun 15 10:01 getenv.c
-rw------- 1 u           5416 Jun 15 10:01 getopt.c
-rw------- 1 u           8068 Jun 15 10:01 h.h
-rw------- 1 u          15843 Jun 15 10:01 input.c
-rw------- 1 u          10084 Jun 15 10:01 macro.c
-rw------- 1 u          25598 Jun 15 10:01 main.c
-rw------- 1 u          14933 Jun 15 10:01 make.c
-rw------- 1 u           4238 Jun 15 10:01 make.man
-rw------- 1 u           1176 Jun 15 10:01 Makefile
-rw------- 1 u          38345 Jun 15 10:01 makshell.c
-rw------- 1 u           5792 Jun 15 10:01 reader.c
-rw------- 1 u           3467 Jun 15 10:01 readme
-rw------- 1 u           8507 Jun 15 10:01 rules.c
-rw------- 1 u           1319 Jun 15 10:01 stat.h
-rw------- 1 u           6398 Jun 15 10:01 syserr.c
```

Contenu du fichier `Makefile`[2]:

```
# Makefile for make utility.
#
CFLAGS =
LDFLAGS =
DESTDIR = /usr/bin
WORKDIR = /usr/local/sys/travail
BACKDIR = /usr/local/src/make
PROG = make
```

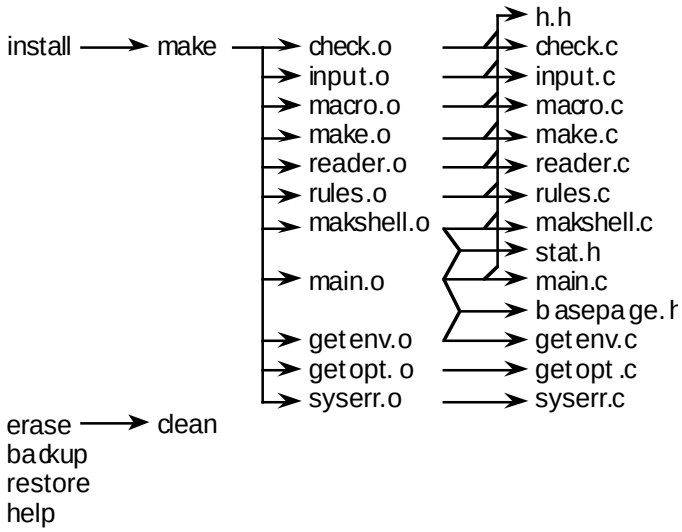
[2] Le caractère @ évite l'impression de la commande lors de son exécution.

```
OBJS = check.o input.o macro.o main.o make.o makshell.o\
          reader.o rules.o
LIBES = getenv.o getopt.o syserr.o
FILES = check.c input.c macro.c main.c make.c makshell.c\
          reader.c rules.c getenv.c getopt.c syserr.c\
          basepage.h h.h stat.h make.man makefile readme
$(PROG): $(OBJS) $(LIBES)
     ld $(LDFLAGS) -o $(PROG) $(OBJS) $(LIBES)

$(OBJS): h.h
getenv.o main.o: basepage.h
main.o makshell.o: stat.h
install: $(PROG)
     cp $(PROG) $(DESTDIR)/$(PROG)
     rm $(PROG)
clean:
     rm $(OBJS) $(LIBES) $(PROG)
erase: clean
     rm $(FILES)
backup:
     cp $(FILES) $(BACKDIR)
restore:
     cd $(BACKDIR)
     cp $(FILES) $(WORKDIR)
     cd $(WORKDIR)
help:
     @echo Makefile options:
     @echo help: Print this message
     @echo default: Create $(PROG)
     @echo install: Move $(PROG) to $(DESTDIR)
     @echo clean: Remove $(PROG) and all .o files
     @echo erase: Erase ALL files
     @echo backup: Copy source files to $(BACKDIR)
     @echo restore: Copy files to $(WORKDIR)
```

Solution de l'exercice 2.13.

2.13.1. Question A

Le graphe de dépendances se déduit du fichier `Makefile` de l'énoncé. Tous les fichiers potentiels doivent être mentionnés dans ce graphe, même s'ils ne sont jamais créés. Il en est ainsi, par exemple, du fichier `install` qui ne sera en fait jamais créé par le programme `make`. Le graphe est donc le suivant:



La règle de dépendance `install: $(PROG)` définit le premier arc du graphe. La règle `$(PROG): $(OBJS) $(LIBES)` définit la dépendance de `make` sur tous les fichiers «`*.o`». La règle implicite sur les suffixes conjointement avec les fichiers du répertoire définit les arc entre les «`*.o`» sur les «`*.c`» correspondants. Enfin les arc vers les «`*.h`» se déduisent des règles:

```
$(OBJS): h.h
getenv.o main.o: basepage.h
main.o makshell.o: stat.h
```

La règle `erase: clean` définit le dernier arc.

2.13.2. Question B

Comme on ne dispose pas encore du programme `make`, il faut en simuler le comportement à la main. Les fichiers «`*.o`» n'existant pas, ils doivent être créés par une commande identique à la commande associée à la règle implicite des suffixes, et appelant le compilateur. Au lieu de lancer ces commandes une à une, on peut utiliser les structures de contrôle du langage de commandes, et donc exécuter:

```
for i in *.c; do cc -c $i; done
```

Lorsque ceci est terminé, on dispose de tous les fichiers «`*.o`» nécessaires, et il reste à exécuter la commande pour créer le programme `make`. Ici encore, on peut utiliser le mécanisme de substitution du langage de commandes, puisque les fichiers «`*.o`» du répertoire sont précisément ceux correspondant aux macros `$(OBJS)` et `$(LIBES)`:

```
ld -o make *.o
```

2.13.3. Question C

Lorsqu'on demande l'exécution du programme `make` sans paramètre, il construit le graphe de dépendances ci-dessus, et fait les vérifications de date en partant du nœud du graphe défini par la première règle du fichier `Makefile`, donc à partir de `make`. Comme ce fichier vient d'être construit, il est plus récent que tous ceux dont il dépend, qui sont eux-mêmes plus récents que ceux dont ils dépendent. Les dates des fichiers étant compatibles avec le graphe de dépendances, il n'y a rien à faire.

2.13.4. Question D

Lorsqu'on lance la commande `make install`, le programme `make` va de nouveau construire le graphe ci-dessus, mais en faisant le contrôle à partir de `install`. Il constatera d'abord que le fichier `install` n'existe pas, et doit donc être construit. Par ailleurs, il constatera également que les fichiers `main.o` et `getenv.o` sont antérieurs au fichier `basepage.h`, puisque ce dernier vient d'être modifié. Il lancera donc les deux commandes:

```
cc -c main.c et cc -c getenv.c
```

Les deux fichiers `main.o` et `getenv.o` étant maintenant postérieurs au fichier `make`, puisqu'ils viennent d'être recréés, le programme lancera la commande de construction du fichier `make`, c'est-à-dire, après remplacement des macros:

```
ld -o make check.o input.o macro.o main.o make.o makshell.o\
reader.o rules.o getenv.o getopt.o syserr.o
```

Enfin, il lancera les commandes de création du fichier `install`, ainsi qu'elles sont définies dans le fichier `Makefile`, après remplacement des macros:

```
cp make /usr/bin/make
rm make
```

Constatons que la première recopie le fichier `make` dans le répertoire `/usr/bin`, et le supprime du répertoire courant. Notons que le fichier `install` n'est toujours pas créé, mais que le programme considère néanmoins, pour cette exécution, qu'il existe maintenant et qu'il est à jour.

2.13.5. Question E

Lors de la commande `make backup`, le programme `make` sera cette fois trouvé par les règles de recherche dans le répertoire `/usr/bin`, pourvu que ces règles de recherche précisent bien ce répertoire. Du fait de la question précédente, il s'agit bien de celui que l'on vient de construire. Il constatera que le fichier `backup` n'existe pas et doit donc être créé par la commande associée définie dans le fichier `Makefile`, et qui est, après remplacement des macros:

```
cp check.c input.c macro.c main.c make.c makshell.c\
     reader.c rules.c getenv.c getopt.c syserr.c basepage.h\
     h.h stat.h make.man makefile readme /usr/local/src/make
```

Cette commande recopie la totalité des fichiers fournis initialement, éventuellement modifiés (pour `basepage.h`) dans le répertoire `/usr/local/src/make` qui vient d'être créé. Notons que les fichiers «`*.o`» ne sont pas recopiés dans ce répertoire.

2.13.6. Question F

Lors de l'exécution de la commande `make erase`, le programme constate qu'il doit créer d'abord le programme `clean`, puisqu'il n'existe pas. Il exécute donc la commande:

```
rm check.o input.o macro.o main.o make.o makshell.o\
     reader.o rules.o getenv.o getopt.o syserr.o make
```

Le fichier `clean` n'est toujours pas créé, mais il considère qu'il existe et est à jour. Il passe donc à la construction du fichier `erase`, en lancant la commande:

```
rm check.c input.c macro.c main.c make.c makshell.c\
     reader.c rules.c getenv.c getopt.c syserr.c basepage.h\
     h.h stat.h make.man makefile readme
```

Le fichier `erase` n'existe toujours pas, mais il considère le travail terminé.

Constatons que la première commande supprime tous les fichiers que l'on a créé au cours de l'exercice. Le fichier `make` n'existait déjà plus dans le répertoire courant depuis la question D. La deuxième commande supprime tous les fichiers qui étaient initialement dans le répertoire, qui est donc vide maintenant. Nous disposons toujours du programme `make`, puisqu'il est dans `/usr/bin` depuis la question D!

2.14. Un make simple

Vous recevez un fichier archive, et vous faites l'extraction de son contenu dans un répertoire initialement vide. Vous affichez alors le répertoire :

```
-rw-r--r--   1         1102 Feb 19  1996 Makefile
-rw-r--r--   1          122 Jun  2  1998 bidule.c
-rw-r--r--   1           30 Feb 20  1996 vol.h
-rw-r--r--   1        23797 Feb 19  1996 vol.c
```

Constatant qu'il contient un fichier `Makefile`, vous en affichez le contenu :

```
CC      = gcc
CFLAGS  =
LDFLAGS =

OBJS    = vol.o
FILES   = Makefile vol.h vol.c

vol:    $(OBJS)
        $(CC) -o vol $(LDFLAGS) $(OBJS)

vol.o : vol.h

vol.tar:  $(FILES)
        ar vol.tar $(FILES)
```

Note : `ar` est un archiveur.

A- Développer les différentes actions possibles liées à ce `Makefile`. Vous préciserez les actions entreprises par les commandes :

```
make vol
```

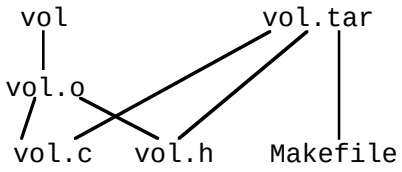
```
make vol.tar
```

B- Tous les fichiers de la distribution initiale vous semblent-ils nécessaires ?

Solution de l'exercice 2.14

2.14.1 Question A

Le graphe de dépendance de ce fichier `Makefile` est relativement simple. La dépendance de `vol.o` sur `vol.c` est obtenue par la règle de suffixe et la présence de `vol.c` dans le répertoire.



La commande `make vol` a pour effet de construire le graphe, en vue de vérifier la cohérence de la portion dont `vol` est racine. On constate que `vol.o` n'existe pas et doit être reconstruit, selon la règle implicite de suffixe, c'est à dire, après application de la substitution de `gcc` à `$(CC)`, par la commande :

```
gcc –c vol.c
```

Il faut ensuite reconstruire `vol`, par la commande associée, qui, après substitutions, devient :

```
gcc –o vol vol.o
```

La commande `make vol.tar` a pour effet de construire le graphe, en vue de vérifier la cohérence de la portion dont `vol.tar` est racine. On constate que `vol.tar` n'existe pas et doit être reconstruit selon la règle associée, qui, après substitutions, devient :

```
ar vol.tar Makefile vol.h vol.c
```

2.14.2 Question B

Dans la distribution le fichier `bidule.c` semble en trop. En effet, il n'apparaît pas dans le fichier `Makefile`, et donc dans le graphe de dépendance associé. Il ne sera donc utilisé par aucune des commandes issues de `make`.

3

Environnement externe

3.1. Comparaisons d'implantation de fichiers

On considère les quatre représentations suivantes de la localisation d'un fichier, représentations vues dans le chapitre 8.

a) Un couple <numéro du premier bloc, nombre de blocs>

b) Un couple <numéro du premier bloc, numéro du dernier bloc> avec chaînage des blocs entre eux.

c) Un ensemble de 24 couples <numéro de bloc, nombre de blocs>

d) Un ensemble de 13 numéros de blocs:
   - 10 numéros de blocs contenant des données,
   - 1 numéro de bloc contenant des numéros de blocs contenant des données,
   - 1 numéro de bloc contenant des numéros de blocs contenant des numéros de blocs contenant des données,
   - 1 numéro de bloc contenant des numéros de blocs contenant des numéros de blocs contenant des numéros de blocs contenant des données,

Donner pour chacune de ces représentations, celles des propositions suivantes qui sont vérifiées, en justifiant brièvement votre réponse.

1. La taille totale du fichier doit être connue lors de sa création.
2. Aucune information de taille n'est nécessaire lors de la création.
3. Il est possible que le fichier ne puisse être créé alors que l'espace libre est supérieur à la taille du fichier.
4. L'accès aléatoire à un bloc quelconque du fichier demande un seul accès disque.
5. L'accès aléatoire à un bloc quelconque du fichier demande au plus 4 accès disque.
6. L'allocation d'espace peut se faire par blocs individuels.

Solution de l’exercice 3.1

La réponse peut se résumer par le tableau suivant:

	1	2	3	4	5	6
a	x		x	x	x	
b		x				x
c			x	x	x	

d		x			x	x

Justification des colonnes 1, 2, 3 et 6:

- Dans le cas *a*, l'espace alloué forme un seul groupe de blocs contigus, sa taille doit donc être connue à la création, il est possible qu'un tel espace n'existe pas à ce moment. Par ailleurs, l'allocation ne peut se faire par blocs individuels, car on ne peut les repérer.
- Le cas *c* est voisin de *a*, si ce n'est que la présence de 24 groupes possibles permet de ne pas connaître exactement la taille du fichier à la création, mais seulement une idée du minimum et du maximum.
- Dans les cas *b* et *d*, les blocs peuvent être ajoutés au fur et à mesure des besoins, suivant une allocation par blocs individuels. On n'a donc besoin d'aucune information sur la taille du fichier lors de la création.

Justification des colonnes 4 et 5:

- Les cas *a* et *c* permettent un calcul de l'adresse physique d'un bloc directement à partir de la représentation de la localisation du fichier. Il y aura donc un seul accès disque, et donc au plus 4.
- Le cas *b* demande à parcourir les liens de chaînage entre blocs. On ne peut donc a priori limiter le nombre des accès disque nécessaires.
- Le cas *d* demande entre 0 et 3 accès, pour parcourir les “blocs contenant des numéros de blocs”, donc au plus 4 accès disque.

3.2. Représentation des fichiers séquentiels

On étudie dans cet exercice diverses représentations d'un fichier dont les enregistrements logiques sont de 50 octets.

A- On dispose d'un lecteur de bandes magnétiques à 1600 BPI (Bytes Per Inch). On suppose que l'espace interbloc est de 3/4 inch. Le temps de lecture ou d'écriture d'un bloc physique de T octets est 10 + 0.008 * T ms.

A.1- On place un enregistrement logique du fichier par bloc physique. Donner le coefficient d'utilisation de la bande ainsi que le temps de lecture/écriture d'un enregistrement logique.

A.2- Pour améliorer cette utilisation, proposer une structure d'un bloc physique permettant le regroupement de plusieurs enregistrements logiques dans un même bloc. Décrire brièvement l'algorithme de lecture, ainsi que l'algorithme d'écriture des enregistrements logiques du fichier.

A.3- Donner, dans les hypothèses de A.2, la taille d'un bloc en fonction du nombre *n* d'enregistrements logiques qu'il contient. Donner le coefficient d'utilisation de la bande et le temps moyen de lecture/écriture d'un enregistrement logique, lorsque $n = 20$ ou $n = 50$.

B- On dispose d'un disque dont les secteurs sont de 1024 octets. On suppose que le temps moyen de lecture/écriture de *s* secteurs consécutifs est 8 + *s* ms (On ne prend pas en compte le mouvement du bras, car on suppose le bras “ bien placé ” au moment de l'opération).

B.1- On place un enregistrement logique du fichier par secteur. Donner le coefficient d'utilisation du disque ainsi que le temps d'exécution de lecture/écriture par enregistrement logique, lors d'un accès séquentiel au fichier.

B.2- On applique la méthode de regroupement vue en A.2. Donner le coefficient d'utilisation de la bande et le temps moyen de lecture/écriture d'un enregistrement logique, lorsque $n = 20$ ou $n = 50$.

B.3- Donner la valeur de *n* qui utilise au mieux les blocs physiques sur 1 secteur; donner dans ce cas le coefficient d'utilisation du disque ainsi que le temps moyen de lecture/écriture d'un enregistrement logique dans ce cas. Même question lorsque les blocs occupent 2, puis 3 secteurs.

C- Le disque comporte 16 secteurs par piste, 20 faces, et 900 cylindres.

C.1- Proposer une numérotation linéaire de l'ensemble des secteurs, ainsi que l'algorithme associé permettant de retrouver à partir d'un numéro virtuel de secteur ses numéros de secteur, face, cylindre.

C.2- L'espace attribué au fichier est une collection d'au plus 24 groupes de secteurs virtuels contigus, le premier groupe étant de taille *s*, et les suivants étant de taille *i*. Proposer une représentation de l'espace alloué à un fichier. Définir les algorithmes de lecture et d'écriture séquentielle d'un enregistrement logique, en supposant les enregistrements regroupés par bloc de 1 secteur comme précisé en B.3.

C.3- Évaluer le nombre d'enregistrements logiques du fichier suivant les valeurs des paramètres *s* et *i*. Donner les critères de choix qui permettraient de déterminer ces paramètres.

C.4- Application numérique, d'une part au cas où l'on sait que le fichier contiendra exactement 500000 enregistrements logiques, d'autre part au cas où le fichier contiendra entre 300000 et 1000000 enregistrements.

Solution du problème 3.2

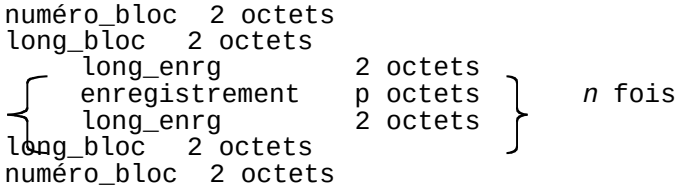
3.2.1. Question A

3.2.1.1. Question A.1

L'espace interbloc représente 3/4 * 1600 octets, soit 1200 octets. L'occupation est donc de 50 / 1250, soit 4 %. Le temps de lecture/écriture est de 10 + 0.008 * 50, soit 10.4 ms.

3.2.1.2. Question A.2

Lorsque les enregistrements sont de taille fixe, la structure la plus simple est de les mettre les uns derrière les autres, à charge pour les sous programmes de lecture et d'écriture de ces enregistrements de découper le bloc physique selon la taille connue des enregistrements logiques. Lorsqu'ils sont de taille variable, il faut connaître la taille de chacun. Le choix d'un caractère spécial comme séparateur n'est pas en général acceptable, car cela implique de l'interdire dans les enregistrements eux-mêmes, ce qui est une contrainte pour l'utilisateur. L'autre solution est de faire précéder chaque enregistrement de sa taille. Dans beaucoup de systèmes, la structure est définie de façon standard et complète, pour permettre de prendre en compte d'une part les enregistrements de taille variable, les blocs de taille variable, ainsi que les opérations de lecture arrière. Une certaine redondance est présente, pour permettre des contrôles. La structure d'un bloc physique pourrait être la suivante:



Pour réaliser les opérations de lecture et d'écriture, on doit disposer d'un tampon de au moins T octets, et de trois variables *numéro* donnant le numéro du bloc en cours, *long* repérant la longueur du bloc courant et *position* repérant la position dans le bloc du prochain enregistrement à lire (ou écrire). Nous ne nous préoccupons ici que de la lecture avant, laissant au lecteur le soin de réaliser l'opération de lecture arrière. Les procédures pourraient être comme suit:

```
procédure lire_enrg(var enregistrement: tableau_octets; var l_enrg: entier);
début si position >= long - 4 alors
          numéro:=numéro+1; lire_bloc; {lecture physique dans le tampon[0..T-1]}
          si tampon[0..1] ≠ numéro alors erreur("numéro de bloc erroné") finsi;
          long := tampon[2..3]; position := 4;
       finsi;
       l_enrg := tampon[position .. position + 1];
       enregistrement := tampon[position + 2 .. position + 1 + l_enrg];
       position := position + 4 + l_enrg;
fin;
```

```
procédure écrire_enrg( enregistrement : tableau_octets; l_enrg : entier);
début si position + l_enrg + 8 > taille_max alors
          numéro := numéro + 1;                { fin de préparation du bloc }
          tampon[0 .. 1] := numéro;
          tampon[2 .. 3] := position + 4;
          tampon[position .. position + 1] := position + 4;
          tampon[position + 2 .. position + 3] := numéro;
          écrire_bloc;                         { écriture physique du tampon }
          position := 4;
       finsi;
       tampon[position .. position + 1] := l_enrg;
       tampon[position + 2 .. position + 1 + l_enrg] := enregistrement;
       tampon[position + 2 + l_enrg .. position + 3 + l_enrg] := l_enrg;
       position := position + 4 + l_enrg;
fin;
```

3.2.1.3. Question A.3

La taille d'un bloc est T = 8 + *n* * (p + 4), soit dans notre cas, T = 8 + *n* * 54. On peut mesurer l'utilisation de deux façons, soit en prenant l'ensemble des informations écrites, soit en ne considérant que les enregistrements seuls. On a donc:

n	T 8+n(4+p)	utilisation T/(T+1200)	util. stricte np/(T+1200)	temps $(10+8 10^{-3} T)/n$
20	1088	48 %	44 %	0.94 ms
50	2708	69 %	64 %	0.63 ms

3.2.2. Question B

3.2.2.1. Question B.1

En prenant 1 secteur par enregistrement logique, le coefficient d'utilisation du disque est de 50 / 1024, soit environ 4.9 %. Le temps d'une opération est de 9 ms par enregistrement logique.

3.2.2.2. Question B.2

Avec la méthode de regroupement définie en A.2, comme il faut un nombre entier de secteurs par bloc, on a donc:

n	T 8+n(4+p)	*s* ⌈T/1024⌉	utilisation T/1024s	util. stricte np/1024s	temps (8+s)/n
20	1088	2	53 %	49 %	0.5 ms
50	2708	3	88 %	81 %	0.22 ms

3.2.2.3. Question B.3

En conservant la méthode de A.2, la valeur optimale de *n* pour un bloc de *s* secteurs est la partie entière de (1024 * *s* - 8) / (p + 4). On a donc:

s	*n* ⌊(1024s-8)/(p+4)⌋	T 8+n(4+p)	utilisation T/1024s	util. stricte np/1024s	temps (8+s)/n
1	18	980	96 %	88 %	0.5 ms
2	37	2006	98 %	90 %	0.27 ms
3	56	3032	98.7 %	91 %	0.2 ms

3.2.3. Question C

3.2.3.1. Question C.1

Une numérotation linéaire possible est nv = ns + 16 * (nf + 20 * nc). Cette numérotation permet de donner des numéros consécutifs à tous les secteurs d'un même cylindre, et donc d'éviter les mouvements de bras entre secteurs virtuels voisins. L'algorithme permettant de retrouver les numéros de secteur, face, cylindre d'un secteur virtuel est le suivant:

ns := nv **mod** 16;

nf := (nv % 16) **mod** 20;
nc := (nv % 16) % 20;

3.2.3.2. Question C.2

L'espace alloué au fichier peut être représenté par la structure suivante:

```
espace = article s, i: entier;                  { taille 1er groupe et suivants }
                  nbg : entier;        { nombre de groupes effectifs du fichier }
                  tab : tableau [0 .. 23] de entier;
                   {tab[i] est le numéro virtuel du premier secteur du groupe i}
         finarticle;
```

Cette structure est contenue dans le descripteur du fichier. Elle est retrouvée depuis le disque lors de l'ouverture du fichier, et sera éventuellement réécrite sur disque en cas de modification (nouvelle allocation). Pour permettre de parcourir le contenu du fichier, on dispose d'une variable supplémentaire *nsf* qui repère le numéro logique du secteur correspondant au bloc actuellement dans le tampon. Ce numéro varie depuis 0 jusqu'à N-1, si N est le nombre total de secteurs du fichier. L'algorithme de lecture d'un enregistrement logique est le même que précédemment. Nous n'avons qu'à préciser l'opération de lecture des blocs physiques.

```
procédure lire_bloc;
début nsf := nsf + 1;
       si nsf < espace.s alors nv := espace.tab[0] + nsf; {premier groupe}
       sinon ng := (nsf - espace.s) % espace.i + 1;      {groupes suivants}
              si ng ≥ espace.nbg alors erreur ("non alloué"); finsi;
              nv := espace.tab[ng] + (nsf - espace.s) mod espace.i;
       finsi;
       ns := nv mod 16; nf := (nv % 16) mod 20; nc := (nv % 16) % 20;
       lire_secteur (tampon, ns, nf, nc);
fin;
```

De même, l'algorithme d'écriture d'un enregistrement logique est le même que précédemment, et c'est même ce qui justifie de prendre la même structure de bloc. Nous n'avons qu'à définir la procédure d'écriture des blocs physiques.

```
procédure écrire_bloc;
début si nsf < espace.s alors nv := espace.tab[0] + nsf;  {premier groupe}
       sinon ng := (nsf - espace.s) % espace.i + 1;       {groupes suivants}
          tantque ng ≥ espace.nbg faire
               si espace.nbg = 24 alors erreur ("trop de groupes"); finsi;
               demande_allocation( espace.tab[espace.nbg], espace.i);
               espace.nbg := espace.nbg + 1;
          fait;
          nv := espace.tab[ng] + (nsf - espace.s) mod espace.i;
       finsi;
       ns := nv mod 16; nf := (nv % 16) mod 20; nc := (nv % 16) % 20;
       écrire_secteur (tampon, ns, nf, nc);
       nsf := nsf + 1;
fin;
```

3.2.3.3. Question C.3

La réponse à B.3 précise qu'il y a 18 enregistrements logiques par secteur. Le nombre de secteurs alloués au fichier est s + (nbg - 1) * i. Donc le nombre d'enregistrements logiques du fichier est 18 * (s + (nbg - 1) * i) au plus, pour un nombre fixé de groupes. Si tous les groupes sont alloués, le nombre d'enregistrements peut atteindre 18 * (s + 23 * i). Par ailleurs, toujours pour un nombre de groupes fixé, il en contient:

nbg	nombre minimal d'enregistrements	nombre maximal d'enregistrements
0	0	0
1	1	18 * s
> 1	18 * (s + (nbg - 2) * i)) + 1	18 * (s + (nbg - 1) * i))

Cependant, le dernier groupe alloué ne sera occupé en moyenne qu'à 50 %. La perte sera donc de 9 * s ou de 9 * i suivant les cas. Ceci permet d'énoncer quelques critères de choix:

a) Si le nombre d'enregistrements logiques du fichier, disons N, est connu lors de la création, et ne doit pas varier, un seul groupe permet de diminuer les mouvements de bras, puisque l'espace alloué est contigu, donc occupe des cylindres consécutifs. Mais encore faut-il trouver un espace de s = N / 18 secteurs consécutifs libres.

b) Si le nombre N n'est connu que par sa limite inférieure N_1 et sa limite supérieure N_2, on doit avoir $s + 23 * i \geq N_2 / 18$. Pour minimiser la perte d'espace pour non utilisation, on peut prendre $i = (N_2 - N_1) / (18 * 24)$ et $s = N_1 / 18 + i$. La perte d'espace sera alors en moyenne équivalente à 9 * i enregistrements logiques du fichier. Mais encore faut-il trouver un espace libre contigu d'au moins s secteurs.

c) Il est toujours possible de diminuer s, pour faciliter l'allocation du premier groupe, mais il faut alors augmenter i en conséquence. Cependant, si l'allocation du premier groupe est impossible dans ce cas, il est inutile de prendre s < i, car l'allocation des groupes suivants sera de toute façon impossible. On peut donc en déduire que $s \geq N_2 / (18 * 24)$. Par ailleurs, on doit avoir $i \geq ((N_2 / 18) - s) / 23$, pour pouvoir ranger les N_2 enregistrements.

3.2.3.4. Question C.4

Dans le cas où N = 500000, de a) ci-dessus, on en déduit que l'on peut prendre s = 500000 / 18 = 27778 secteurs. Ceci représente 10 % de l'espace total du disque, qu'il faut trouver de façon contiguë. Si ceci n'est pas possible, de c) ci-dessus, on en déduit que $s \geq 500000 / (18 * 24) = 1158$ secteurs, et $i \geq (27778 - s) / 23$.

Dans le cas où $N_1 = 300000$ et $N_2 = 1000000$, de b) ci-dessus, on en déduit que l'on peut prendre $i = 700000 / (18 * 24) = 1621$, et $s = 16667 + 1621 = 18288$. Notons que en vertu de c) ci-dessus, on doit avoir $s \geq 2315$, et $i \geq (55560 - s) / 23$.

3.3. Fichiers séquentiels à longueur variable

On désire créer sur disque un fichier séquentiel d'enregistrements logiques, dont la taille est variable, parmi trois tailles possibles :

65% des enregistrements font 200 octets,

25% des enregistrements font 280 octets,

10% des enregistrements font 360 octets.

Les secteurs du disque font 1024 octets, le temps d'accès à s secteurs consécutifs est 8+s millisecondes, pourvu que la suite des opérations permette de négliger les mouvements de bras.

A– Déterminer la taille moyenne des enregistrements du fichier.

B– Donner la structure de bloc physique permettant le regroupement de plusieurs enregistrements logiques.

C– Étant données les caractéristiques du disque, quelle taille de bloc physique vous paraît judicieuse? En déduire l'occupation d'espace disque.

D– À votre avis, quelles sont les conditions qui permettent de négliger les mouvements de bras? Sont-elles remplies ici, et si oui, quel est le temps moyen de lecture ou d'écriture par enregistrements logique?

Solution de l'exercice 3.3

3.3.1. Question A

Si on prend 100 enregistrements, 65 ont une longueur de 200, 25 une longueur de 280 et 10 ont une longueur de 360, le total est donc de 23600, soit une longueur moyenne de 236.

3.3.2. Question B

Nous pouvons reprendre le raisonnement déjà vu dans l'exercice précédent, mais cette fois les enregistrements sont de longueur variable, et il faut connaître la taille de chacun. Le choix d'un caractère spécial comme séparateur n'est pas en général acceptable, car cela implique de l'interdire dans les enregistrements eux-mêmes, ce qui est une contrainte pour l'utilisateur. L'autre solution est de faire précéder chaque enregistrement de sa taille. Dans beaucoup de systèmes, la structure est définie de façon standard et complète, pour permettre de prendre en compte d'une part les enregistrements de taille variable, les blocs de taille variable, ainsi que les opérations de lecture arrière. Une certaine redondance est présente, pour permettre des contrôles. La structure d'un bloc physique pourrait être la suivante:

```
numéro_bloc  2 octets
long_bloc   2 octets
    long_enrg         2 octets
   enregistrement     p octets    } n fois
    long_enrg         2 octets
long_bloc   2 octets
numéro_bloc  2 octets
```

La taille moyenne d'un bloc physique est donc 8 + 240 n.

3.3.3. Question C

Un bloc physique doit occuper un nombre entier de secteurs de 1024 octets. On peut donc étudier combien il est possible de mettre d'enregistrements dans 1, 2, 3 ou 4 secteurs. D'après la question précédente, si s est le nombre de secteur occupés par le bloc physique, n = (1024 s – 8) **div** 240. Le tableau donne les différents cas :

s	n	Taille bloc	Perte en %
1	4	968	5,5
2	8	1928	5,9
3	12	2888	6,0
4	17	4088	0,2

Si on cherche à optimiser la place disque, il semble bien que des blocs physiques de 4 secteurs donnent le minimum de perte. Notons que le nombre d'enregistrements par bloc est un nombre moyen, mais que plus ce nombre augmente, plus la répartition entre les tailles d'enregistrements se rapprocheront de la répartition de l'ensemble du fichier, ce qui milite également pour des blocs de 4 secteurs.

3.3.4. Question D

Si l'allocation des blocs physiques est contiguë et que l'on fait un parcours séquentiel des enregistrements du fichier, il n'y aura pas de mouvement de bras entre deux lectures ou écritures de blocs physiques, et ils peuvent donc être négligés. Dans ce cas, la lecture d'un bloc demandera, pour des blocs de 4 secteurs, 8 + 4 = 12 ms. Notons que ceci veut dire une moyenne de 0,7 ms par enregistrement logique du fichier.

3.4. Étude de la structuration d'un disque

On se propose d'étudier une structuration pour des disques dont la capacité est comprise entre 100 Mo. et 2 Go., les secteurs étant de 4096 octets.

Les machines pour lesquelles ils sont destinés manipulent des octets, des entiers sur 16 bits ou des entiers sur 32 bits.

Les concepteurs du système de gestion de fichiers ont décidé de diviser chaque unité de disque (amovible ou non) de la façon suivante:

- *le descripteur du disque*, qui contient tous les renseignements nécessaires à son identification, les paramètres variables de la structure, et une table donnant accès aux fichiers.

- *la table d'allocation de l'espace*, représentant l'espace libre sur le disque.
- *la zone des fichiers*, qui contient les descripteurs de fichiers et l'espace de données de ces fichiers.

Un fichier est constitué de deux parties:

- *le descripteur de fichier* qui contient toutes les informations nécessaires à sa localisation, et qui seront détaillées ci-dessous.
- *l'ensemble des zones physiques*, qui contiennent les données du fichier. Pour un fichier, toutes les zones physiques du fichier ont la même taille, et sont constituées d'un nombre entier de secteurs contigus. Par contre les différentes zones d'un même fichier ne sont pas nécessairement contiguës.

A. Étude des descripteurs de fichiers.

Les descripteurs de fichiers doivent repérer l'ensemble des zones du fichier. Pour permettre à un fichier de contenir un nombre quelconque de zones, tout en ayant des descripteurs de taille fixe, les descripteurs sont décomposés en une partie principale (obligatoire) et des parties extensions (optionnelles). Les parties extensions éventuelles ont la même structure que les parties principales, les informations inutilisées étant simplement mises à 0 dans ce cas. Par la suite nous appellerons descripteur cette structure, qu'elle soit utilisée comme partie principale ou comme extension. Les descripteurs de fichiers sont regroupés dans des secteurs du disque. Ces secteurs sont chaînés entre eux, pour permettre la recherche des fichiers. Un descripteur de fichier a la structure suivante:

. le *nom* du fichier sur 16 octets,

. des informations diverses sur 12 octets,

. la *longueur du fichier*, c'est-à-dire le nombre total d'octets du fichier,

. le *nombre total de zones* du fichier,

. la *taille des zones* en nombre de secteurs,

. l'“*adresse” de l'extension suivante* (numéro de secteur et position dans le secteur),

. l'“*adresse” de la dernière extension* du fichier,

. la *table des numéros de premiers secteurs* des premières zones.

A.1- En considérant que tout entier sur disque a la même représentation qu'en mémoire centrale, c'est-à-dire 16 ou 32 bits, déterminer la taille en octets nécessaire à la représentation de chacune des informations suivantes:

. un numéro de secteur,

. la longueur du fichier, c'est-à-dire le nombre total d'octets du fichier,

. le nombre total de zones du fichier,

. la taille des zones en nombre de secteurs,

. l'“adresse” d'une extension (numéro de secteur et position dans le secteur).

A.2- Déterminer la taille d'un descripteur en fonction du nombre d'entrées q de la table des premiers secteurs de zones dans un descripteur. En déduire la valeur maximale du nombre r de descripteurs dans les secteurs du disque qui les contiennent, en fonction de q, et la perte qui résulte d'un choix donné de r et q compatibles.

A.3- Étudier en particulier les cas où ce nombre q prend les valeurs 31, 32 et 33. Calculer la perte d'espace dans chaque cas. Commenter le choix de 32.

A.4- Expliquer l'intérêt des différentes informations (autres que diverses) qui sont présentent dans la partie principale d'un descripteur. On distinguera celles qui sont nécessaires et celles qui sont redondantes, et on justifiera cette redondance. Montrer que l'on peut déduire de ces informations le nombre d'extensions du descripteur de fichier.

B. Étude des descripteurs de disque

Nous avons déjà dit que les descripteurs de fichiers étaient regroupés dans des secteurs du disque, chaînés entre eux. Pour minimiser le parcours de ces secteurs lors de la recherche d'un fichier, on partitionne les descripteurs en p ensembles disjoints par un calcul sur le nom du fichier, *H(nom)*, qui délivre un entier entre 1 et p. Le descripteur du disque contient une table qui, pour chacune des valeurs entre 1 et p, indique le numéro du premier secteur contenant les descripteurs des fichiers dont le calcul *H(nom)* donne cette valeur (fonction de hachage). Le descripteur du disque, *qui doit tenir dans un seul secteur du disque*, est donc défini comme suit:

. le *nom* du disque sur 32 octets,

. le *nombre total de secteurs* du disque,

. le *nombre de secteurs occupés*,

. le *nombre de fichiers existants* sur le disque,

. le *nombre de descripteurs de fichiers par secteurs*,

. le *nombre d'entrées de la table de hachage*,

. la *table de hachage*, donnant pour chaque valeur de hachage le numéro du premier secteur des descripteurs de fichiers.

B.1- En considérant que tout entier sur disque a la même représentation qu'en mémoire centrale, c'est-à-dire 16 ou 32 bits, déterminer la taille en octets nécessaire à la représentation de chacune des informations suivantes:

. le nombre de fichiers existants sur le disque,

. le nombre de descripteurs de fichiers par secteurs,

. le nombre d'entrées de la table de hachage,

B.2- Le descripteur du disque devant tenir dans un seul secteur, combien peut-il y avoir au plus d'entrées dans la table de hachage? À quel endroit convient-il de mettre ce secteur sur le disque?

B.3- Expliquer l'intérêt des différentes informations qui sont présentent dans le descripteur du disque. On distinguera celles qui sont nécessaires et celles qui sont redondantes, et on justifiera cette redondance.

B.4- Comment jugez-vous ce système? Quels avantages ou inconvénients y voyez-vous?

B.5- Définir la procédure d'ouverture d'un fichier, ainsi que la procédure de lecture d'un octet quelconque de ce fichier défini par sa position relative au début du fichier.

Solution de l’exercice 3.4

3.4.1. Question A

3.4.1.1. Question A.1

- *numéro de secteur.* Un disque peut contenir jusqu'à $2*2^{30} / 4*2^{10} = 2^{19}$ secteurs. Il faut donc au moins 19 bits pour représenter un numéro de secteur, soit un entier sur 32 bits, ou 4 octets.
- *longueur du fichier.* Un fichier ne peut dépasser la taille maximale d'un disque, donc $2 * 2^{30} = 2^{31}$ Il faut donc des entiers sur 32 bits, soit 4 octets pour représenter le nombre d'octets utile d'un fichier.
- *nombre total de zones et taille des zones.* En l'absence de contrainte, une zone physique d'un fichier doit être d'au moins 1 secteur, mais pouvoir atteindre presque le disque complet (à l'exception des informations structurelles). Un fichier peut donc avoir 2^{19} zones de 1 secteur; il faut donc 32 bits ou 4 octets pour représenter le nombre total de zones du fichier. De même, un fichier peut avoir 1 zone de 2^{19} secteurs; il faut donc 32 bits ou 4 octets pour représenter la taille des zones du fichier. Notons qu'il ne serait pas très contraignant et sans doute assez logique de limiter chacune de ces quantités à 2^{16}, permettant ainsi d'utiliser des entiers sur 16 bits ou 2 octets, puisque ceci nous donne de toute façon 2^{32} secteurs, valeur nettement supérieure au nombre maximal de secteurs d'un disque.

- *adresse d'une extension.* L'adresse d'une partie extension d'un descripteur doit permettre de désigner d'une part le numéro de secteur disque où la partie extension est placée (19 bits), et d'autre part, sa position dans ce secteur. Celle-ci peut être définie simplement par le numéro de son premier octet (12 bits), donnant ainsi un total de 31 bits. Cette position peut aussi être le numéro d'ordre du descripteur dans le secteur, puisque tous les descripteurs sont de même taille. Par exemple, s'il y a au plus 32 descripteurs par secteur, 5 bits suffiront alors pour ce numéro, et une adresse de partie extension nécessitera 24 bits. Enfin, en suivant la politique de représentation des entiers en 16 ou 32 bits, on constate alors qu'il faut 32 bits ou 4 octets dans tous les cas.

3.4.1.2. Question A.2

Pour déterminer le nombre de descripteurs de fichiers par secteur, il faut déterminer la taille d'un descripteur. Nous reprenons les données numériques de la question A.1, et les notons devant les informations du descripteur de fichier:

16	le nom du fichier sur 16 octets,
12	des informations diverses sur 12 octets,
4	le nombre total d'octets du fichier,
4	le nombre total de zones du fichier,
4	la taille des zones en nombre de secteurs,
4	l'“adresse” de l'extension suivante (numéro de secteur et position dans le secteur),
4	l'“adresse” de la dernière extension du fichier,
4*q	la table des numéros de premiers secteurs des premières zones.

Il s'ensuit que le descripteur occupe 48 + 4 * q octets, où q est le nombre d'entrées de la table des premiers secteurs de blocs d'un descripteur. Les secteurs contenant des descripteurs étant chaînés entre eux, le lien de chaînage occupe 4 octets, laissant libres 4092 octets pour r descripteurs, et il s'ensuit que $r \leq 4092 / (48 + 4 * q)$. Pour une valeur de r compatible avec q, la perte est $4092 - r * (48 + 4 * q)$.

3.4.1.3. Question A.3

Nous avons les valeurs suivantes:

q	r maximum	perte
31	23	136
32	23	44
33	22	132

Le choix de 32 est le plus optimal, puisqu'il correspond à la plus petite perte. Cependant, même s'il induisait une perte supérieure aux autres, il pourrait être conservé car il a l'avantage de permettre de représenter, dans chaque descripteur, qu'il soit partie principale ou extension, un nombre de zones qui est une puissance de 2.

3.4.1.4. Question A.4

Le *nom* du fichier est nécessaire pour distinguer les fichiers entre eux, et permet à l'utilisateur de désigner ce fichier par une chaîne de caractères. La *longueur du fichier* N est la seule information de taille qui permette de savoir exactement quels sont les octets de l'espace alloué au fichier qui en font partie. La *taille des zones* T est nécessaire pour déterminer la taille des zones à allouer au fur et à mesure des besoins au fichier; elle permet également de déterminer à quelle zone appartient le $n^{ième}$ secteur du fichier. L'*adresse de l'extension suivante* est nécessaire pour parcourir les extensions du descripteur. La *table des numéros de premiers secteurs* permet de savoir où commence chaque zone sur disque, et donc détermine l'espace alloué au fichier.

Le *nombre total de zones* du fichier est une information redondante, en ce sens qu'elle peut être reconstruite à partir des autres. En effet, le nombre de secteurs S est égal à (N-1)%4096+1, où % désigne la division entière. À partir de la taille des zones T, on peut déduire le nombre de zones B = (S-1)%T+1. Elle évite de refaire ce calcul en particulier pour déterminer la taille de l'espace occupé par le fichier sur le disque. Enfin l'*adresse de la dernière extension* du fichier est également une information redondante, puisqu'elle peut être connue en parcourant les extensions successives.

Elle est utile car l'allocation d'une nouvelle zone lors d'un allongement du fichier, entraîne la modification de cette extension.

Par ailleurs (B-1)%32 nous donne le nombre d'extensions du fichier.

3.4.2. Question B

3.4.2.1. Question B.1

- *nombre de fichiers.* Le nombre maximum de fichiers que l'on peut avoir s'obtient en divisant la taille maximale du disque par la taille minimale occupée par un fichier. Si les fichiers contiennent au moins 1 octet (pas de fichiers vides), l'espace qu'ils occupent est alors au moins de 1 secteur, donc il y en a alors au plus 2^{19}. Si on admet que tous les fichiers peuvent être vides (hypothèse d'école!), leur nombre est limité au nombre de descripteurs pouvant exister, et la taille nécessaire à la représentation du nombre de fichiers existants sur le disque est la même que celle des adresses de descripteurs, calculée ci-dessus. La politique de représentation des entiers conduit donc à prendre 32 bits ou 4 octets. Remarquons que ceci est tout théorique, et que sans doute 16 bits ou 2 octets seraient amplement suffisants. Néanmoins, étant donné que cette valeur ne se retrouve que dans le descripteur du disque, la perte de 2 octets est vraiment négligeable.
- *nombre de descripteurs par secteurs.* Le nombre de descripteurs de fichiers par secteur, dépend de la taille d'un descripteur. Comme un tel descripteur contient au moins 16 octets (le nom), il y en a donc au plus 256 par secteur. Un octet peut suffire. Comme il s'agit d'un entier, nous devons prendre 16 bits ou 2 octets pour sa représentation.
- *nombre d'entrées de la table de hachage.* La table de hachage étant incluse dans le descripteur du disque, lui-même devant tenir dans un seul secteur, le nombre de ses entrées est donc au plus égal à la taille d'un secteur divisé par la taille de l'une de ses entrées. Comme nous avons fixé plus haut à 4 octets la représentation des numéros de secteur du disque, il y a donc au plus 1024 entrées, ce qui nécessite 16 bits ou 2 octets.

3.4.2.2. Question B.2

Nous reprenons les données numériques des questions A.1 et B.1, et les notons devant les informations du descripteur du disque:

32	le nom du disque sur 32 octets,
4	le nombre total de secteurs du disque,
4	le nombre de secteurs occupés,
4	le nombre de fichiers existants sur le disque,
2	le nombre de descripteurs de fichiers par secteurs,
2	le nombre d'entrées de la table de hachage,
4*p	la table des premiers secteurs de descripteurs de fichiers.

Il s'ensuit que le descripteur occupe 48 + 4 * p octets. Comme il doit tenir dans un secteur, il s'ensuit que $p \leq (4096 - 48) / 4 = 1012$.

Le système doit être capable de trouver le descripteur d'un disque quelconque, avant d'avoir aucune information sur sa structure. Ce descripteur doit donc être dans un secteur fixe pour tous les disques. Il est alors logique de le mettre dans le secteur 0, c'est-à-dire, secteur 0 de la face 0 du cylindre 0.

3.4.2.3. Question B.3

Le *nom* du disque permet d'attacher un “label” au volume ainsi structuré. Les utilisateurs pourront utiliser cette chaîne de caractères pour demander au système de contrôler qu'il s'agit du bon volume. Le *nombre total de secteurs* permet de connaître la taille du disque lui-même. Cette information est parfois redondante, si le matériel impose le nombre de secteurs par piste, le nombre de faces et le nombre de cylindres, mais ce n'est pas toujours le cas. Le *nombre de descripteurs de fichiers par secteurs* est nécessaire si le SGF admet que ce nombre varie d'un volume à l'autre. Cette paramétrisation a l'avantage de faciliter l'adaptation de la structure aux besoins des utilisateurs plutôt que de l'imposer en tant que constante interne au système. Le *nombre d'entrées de la table de*

hachage permet d'adapter cette valeur à la capacité du disque et au nombre de fichiers que l'on prévoit d'y mettre.

Le *nombre de secteurs occupés* est une information redondante, puisqu'il suffit de parcourir l'ensemble de tous les descripteurs de fichiers pour la retrouver. Le *nombre de fichiers existants* est également une information redondante, puisque ce nombre peut être obtenu par parcours des descripteurs de fichiers. La conservation de ces valeurs évite ce parcours lorsqu'on désire savoir quel est le taux d'occupation du disque, le nombre de fichiers, la taille moyenne des fichiers, etc...

3.4.2.4. Question B.4

La désignation des fichiers dans ce système est assez pauvre, puisque tous les fichiers sont désignés par un nom sur 16 octets, et qu'il y a un répertoire unique pour l'ensemble des fichiers du disque: il n'y a pas de structure arborescente. Cet inconvénient peut être particulièrement important si les fichiers sont de petite taille, et le disque de grosse capacité (500000 fichiers de 4Ko. par exemple). La présence de la table de hachage peut améliorer nettement l'efficacité de la recherche d'un fichier, puisqu'elle implique un partitionnement de l'ensemble des noms de fichiers en 1000 groupes environ. Cependant, il n'en reste pas moins qu'un tel répertoire plat est désagréable pour les utilisateurs dans leur construction et manipulation des noms de fichiers.

Ce système permet de prendre en compte des fichiers de taille quelconque. Si l'utilisateur fixe correctement la taille des zones de son fichier, et que les allocations contiguës de cette taille sont possibles, le comportement peut se rapprocher de celui des systèmes à nombre fixe maximum de zones, puisque alors son descripteur de fichier peut n'avoir qu'une partie principale, sans extension. Si par contre, l'utilisateur fixe une taille de zones égale à 1 secteur, il a la garantie de pouvoir créer un fichier de taille quelconque, dans la limite de l'espace réellement disponible, au prix d'une certaine perte d'efficacité due au nombre de ses extensions si son fichier est de taille importante. Notons cependant que les fichiers dont la taille est inférieure ou égale à 128 Ko n'ont jamais besoin d'une extension, s'il y a 32 entrées dans la table des numéros de premiers secteurs d'un descripteur.

Notons qu'un disque de 100 Mo offre ici 25000 secteurs, alors qu'un disque de 2 Go en offre 500000. Dans ce dernier cas, l'allocation au niveau du secteur peut être jugée lourde et inefficace. Une méthode courante pour diminuer cette difficulté est de considérer que l'unité minimum d'allocation est de plusieurs secteurs. Le seul changement à apporter à la structure est de mémoriser dans le descripteur du disque cette unité d'allocation en nombre de secteurs. Lors de la création d'un fichier, la taille effective des zones est alors forcée au multiple de cette unité d'allocation immédiatement supérieure ou égale à celle demandée par l'utilisateur, sans qu'il ait à se préoccuper de la valeur de cette unité d'allocation.

3.4.2.5. Question B.5

Donnons tout d'abord les structures manipulées dans un langage “à la Pascal”.

```
type t_descripteur: article     { représentation d'un descripteur }
          nom: chaîne [16];
          info: chaîne [12];
          longueur: entier;
          nb_zone: entier;
          taille_zone: entier;
          ext_suivante, dernière_ext: entier;
          def_zone: tableau [0 .. 31] de entier;
     fin article;
type t_sect_descr: article      { représentation d'un secteur de descripteurs }
          descr: tableau [0 .. 22] de t_descripteur;
          suivant: entier;
     fin article;
```

Nous supposons que la table de hachage du disque est déjà présente en mémoire centrale. La procédure suivante d'ouverture de fichier initialise le descripteur de fichier qui lui est passé en paramètre avec la partie principale si elle a été trouvée, ou avec un descripteur invalide sinon.

```
var tab_hachage: tableau [1 .. p] de entier;         { supposé déja en mémoire }
procédure ouvrir (nom: chaîne [16]; var descr: t_descripteur);
var sect_descr: t_sect_descr;
   s, i: entier;
   non_trouvé: booléen;
début non_trouvé := vrai;
       s := tab_hachage [H (nom)];
       tantque s ≠ 0 et non_trouvé faire
          lire_disque (sect_descr, s);
          i := 0;
          tantque i < 23 et non_trouvé faire
               si sect_descr.descr [i].nom = nom alors non_trouvé := faux
               sinon i := i + 1 finsi;
          fait;
          s := sect_descr.suivant;
       fait;
       si non_trouvé alors descr := descripteur_invalide
       sinon descr := sect_descr.descr [i]; finsi;
fin;
```

Cette procédure pourrait être complétée par la lecture des extensions du fichier. Comme le nombre de zones du fichier est mémorisé dans la partie principale, il serait alors possible de constituer un tableau unique avec tous les numéros de premier secteur de toutes les zones du fichier. Cependant si le fichier fait l'objet d'allongement (écriture au bout), ceci peut conduire à augmenter la taille de cette table au cours de l'exécution du programme, à moins de la surdimensionner initialement pour réduire l'éventualité de cet allongement.

La procédure de lecture d'un octet situé à une position donnée, peut se définir comme suit:

```
procédure lire (descr: t_descripteur; position: entier; var c: octet);
var sect_descr: t_sect_descr;
   sect_donnée: tableau [0 .. 4095] de octets;
   num_ext, num_zone, num_sect, num_oct: entier;
   s, i: entier;
début num_oct := position mod 4096;     { position dans le secteur de données }
  position := position % 4096;          { numéro relatif de secteur du fichier }
  num_sect := position mod descr.taille_zone;{numéro du secteur dans la zone}
  position := position % descr.taille_zone;{numéro relatif de zone du fichier}
  num_zone := position mod 32;       { numéro de zone dans l'extension }
  num_ext := position % 32;
  si num_ext = 0 alors s := descr.def_zone [num_zone]
  sinon { lecture de l'extension si ce n'est pas fait lors de l'ouverture }
     s := descr.ext_suivante;
     tantque num_ext ≠ 0 faire
       i := s mod nb_descr_par_secteur;
       s := s % nb_descr_par_secteur;
       lire_disque (sect_descr, s);
       s := sect_descr.descr [i].ext_suivante;
       num_ext := num_ext - 1;
     fait;
     s := sect_descr.descr [i].def_zone [num_zone];
  finsi;
  lire_disque (sect_donnée, s);
  c := sect_donnée [num_oct];
fin;
```

3.5. Autre structuration d'un disque

On dispose de disques de 1500 cylindres, 12 faces, 16 secteurs de 2 Ko par piste. La vitesse de rotation est de 3600 tours/mn, et le temps moyen de déplacement du bras est de 20 ms. Ce disque comporte de plus 32 cylindres indépendants des précédents, dont chaque piste dispose de sa propre tête, qui est donc fixe, et contient également 16 secteurs de 2 Ko.

Le système implante une technique d'allocation par zone avec extensions: l'espace alloué à un objet externe est constitué d'une zone primaire, et de 0 à 16 zones secondaires.

La notice du système précise que la zone primaire n'est utilisée que pour les fichiers séquentiels indexés, et qu'elle contient alors les tables d’index. S'il y a de la place, la zone primaire est allouée

de préférence dans les cylindres à tête fixe; s'il n'y a pas de place, elle est allouée dans les cylindres à bras mobile. Si cette zone primaire n'est pas suffisante pour contenir ces tables, celles-ci débordent dans les zones extensions. Les zones extensions sont toujours allouées dans les cylindres à bras mobile et, quelle que soit l'organisation, les enregistrements sont rangés dans les zones extensions qui sont allouées au fur et à mesure des besoins.

A- L'ingénieur système a fixé une taille de quantum égale à 32 Ko. Trouvez-vous cette valeur adaptée? Pourquoi faut-il gérer séparément l'espace des cylindres à tête fixe et celui des cylindres à bras mobile? Combien y a-t-il de quanta dans chacun de ces espaces?

B- Lorsqu'on crée un objet externe, on doit préciser quel est son type. Expliquez pourquoi? Dans le cas d'un fichier séquentiel indexé, quelles autres informations vous paraissent devoir également être fournies? Quelles sont celles qui font parties de la définition du lien et celles qui se déduisent du programme source?

C- A votre avis, qu'est-ce qui a justifié de ne pas utiliser la zone primaire pour les fichiers séquentiels ou les fichiers à accès aléatoire par numéro?

D- La notice du système suggère de définir, pour un fichier séquentiel indexé, les valeurs suivantes de taille de partie primaire P et des extensions I:

$$P = 1.4 * \frac{(c+5) * N}{Q} \text{ quantas, et } I = 1.2 * \frac{(e+10) * N}{16 * Q} \text{ quantas}$$

où `c` est la longueur en octets de la clé, `e` la taille moyenne d'un enregistrement, `N` le nombre maximum d'enregistrements du fichier et `Q` la taille d'un quantum. Qu'en pensez-vous?

E- On désire créer un fichier séquentiel indexé d'au plus 300 000 enregistrements, dont les enregistrements font en moyenne 300 octets, et la clé fait 10 octets. Calculez les tailles de parties primaire et extension. Pensez-vous que ces valeurs soient compatibles avec les espaces disques? La création sera-t-elle toujours possible?

F- Un outil d'analyse du système vous informe que d'une part, le fichier créé en E est une indexation sur 3 niveaux et que d'autre part, les deux premiers niveaux d'indexation occupent moins de 100 Ko et le dernier 5 Mo.

F.1- Calculez le temps d'attente d'entrées-sorties pour accéder à un enregistrement de clé donnée, suivant que la partie primaire est dans l'espace des cylindres à tête fixe ou dans l'espace des cylindres à bras mobile.

F.2- Même question si on suppose que le système conserve maintenant les deux premiers niveaux en mémoire centrale.

F.3- Quelles conclusions tirez-vous de ces deux dernières questions?

G- Si on dispose de disques à tête fixe séparés des disques à bras mobile, peut-on appliquer la technique ci-dessus? Quels seront les avantages et les inconvénients?

Solution de l'exercice 3.5

3.5.1. Question A

La taille de 32 Ko pour un quantum correspond exactement à une piste (16 secteurs de 2 Ko). Un quantum multiple d'une piste ou diviseur d'une piste évite qu'une unité d'allocation élémentaire soit à cheval sur deux pistes, et devrait donc minimiser les mouvements de bras pour un fichier donné. Une taille de 4 Ko, 8 Ko ou de 16 Ko permet sans doute d'avoir plus de petits fichiers, mais conduit à un nombre plus grand d'unités allouables. Une taille de 64 Ko ou plus conduirait à une perte place pour ces petits fichiers, mais permettrait un nombre plus faible d'unités allouables. La taille de 32 Ko donne 12 * 1500 = 18000 unités allouables, ce qui paraît être un bon compromis si le disque est destiné à recevoir surtout des fichiers de taille moyenne ou grosse.

Étant donné que les zones primaires sont allouées de préférence dans l'espace des cylindres à tête fixe, contrairement aux zones secondaires, la gestion de cet espace doit être séparé de l'espace des cylindres à bras mobiles.

L'espace à tête fixe contient 12 * 32 = 384 quanta, alors que l'espace à bras mobile en contient 18000.

3.5.2. Question B

Lorsqu'on crée un objet externe, il faut structurer l'espace qui lui est alloué conformément au type de fichier qui est désiré. Tant que cet espace n'est pas structuré, les opérations d'accès ne peuvent mémoriser ou retrouver les informations dans cet objet. Il est donc naturel que le type soit précisée lors de la création. Par ailleurs, dans ce problème, la connaissance du type permet de savoir s'il faut ou non allouer une zone primaire.

Il est évident que lorsque l'on crée un objet externe, il faut définir la taille des zones extensions, ainsi que la taille de la zone primaire pour les fichiers séquentiels indexés. Ces informations doivent faire partie de la définition du lien et ne peuvent être déduites du programme, puisque celui-ci devrait être capable de traiter aussi bien les petits que les gros fichiers.

Si le fichier est séquentiel indexé, cela veut dire que les enregistrements contiennent tous une clé qui doit être mémorisée dans les tables utilisées pour les retrouver. Il est donc nécessaire que le système puisse localiser la clé dans les enregistrements, c'est-à-dire savoir quelle est sa position et quelle est sa longueur. Cette longueur lui permet de plus de structurer l'espace des tables et, en particulier, les blocs physiques qui doivent les contenir. Ces informations peuvent être déduites du programme source qui est censé connaître les informations qu'il manipule.

Il peut être utile de savoir si les enregistrements sont en longueur fixe, et quelle est alors leur longueur, ou si les enregistrements sont en longueur variable, et quelle est alors la longueur maximum. Cette information est utile pour la structuration de l'espace qui doit contenir ces enregistrements et, en particulier, les blocs physiques qui doivent les contenir. Ces informations peuvent souvent être déduites du programme source, sauf s'il ne manipule qu'une partie des informations que doit contenir le fichier au cours du temps; c'est le cas, par exemple, lorsque l'on crée le fichier avec des enregistrements de même taille initiale, mais pouvant être rallongés par la suite.

3.5.3. Question C

Tout d'abord, il est dit que les enregistrements eux-mêmes sont toujours dans les zones extensions, la zone primaire ne pouvant contenir que les tables d’index des fichiers séquentiels indexés. Cela permet une homogénéité avec les fichiers séquentiels et facilitant leur lecture séquentielle. Par ailleurs cela permet de réserver l'espace à tête fixe aux tables d’index pour améliorer les accès.

3.5.4. Question D

On peut déduire des formules que le système réserve 5 octets d'informations internes de gestion (pointeur) pour chaque clé. L'expression `(c + 5)*N` est alors la taille totale en octets pour les tables. En divisant par `Q`, on obtient la taille nécessaire en quanta. Le coefficient de `1.4` permet de prendre en compte un remplissage partiel moyen des blocs physiques à environ 70%, soit par construction pour permettre l'augmentation par la suite du nombre de clés par bloc, soit parce que le système utilise une partie de ces blocs pour leur gestion propre (en-tête de bloc par exemple).

De même on voit que le système réserve 10 octets par enregistrement pour sa gestion interne. La taille minimale en octets est alors de `(e + 10)*N`. En divisant par `Q`, on obtient la taille totale en quanta. En divisant de nouveau par `16`, on obtient la taille minimale de chaque extension. Le coefficient de `1.2` permet de prendre en compte un remplissage partiel moyen de `83`% des blocs disques.

Remarquons que le système, connaissant les valeurs de `c` et de `e` (question B), peut avoir une estimation de la valeur de `N`, et en tenir compte pour structurer les tables de la zone primaire.

3.5.5. Question E

Des valeurs numériques, on déduit P = 193 quanta, et I = 213. Ces valeurs sont compatibles avec les espaces disques, puisque l'espace à tête fixe contient 384 quanta, et l'espace à bras mobile en contient 18000. Comme P est égal à la moitié de l'espace à tête fixe, il n'est pas certain qu'un tel espace existe effectivement de façon contiguë dans les cylindres à tête fixe. S'il n'existe pas, le système recherchera le même espace dans les cylindres à bras mobile. Il est probable qu'il sera toujours capable de faire l'allocation dans ce cas, puisque cela ne représente alors que 1% du total. Si ce n'était pas possible, cela voudrait dire que l'espace libre du disque est très morcelé ou proche de la saturation. Remarquons que l'impossibilité de création ne peut être la conséquence d'une saturation de l'espace à tête fixe, mais de celle de l'espace à bras mobile.

3.5.6. Question F

La recherche d'un enregistrement à partir de sa clé, demande 3 accès disque dans les tables, dus aux 3 niveaux, et un accès disque en zone extension pour obtenir l'enregistrement lui-même.

Commentaire: La proportion, 100 Ko pour les deux premiers niveaux et 5 Mo pour le suivant, correspond à environ 50 clés par table au niveau 2 (5 Mo/100 Ko). S'il en est également ainsi au niveau 1, on voit que chaque table occupe 1 secteur disque (2 Ko) ce qui finalement est assez logique. Notons que l'occupation minimum de 50 clés dans les tables semble être de 50 * (10 + 5) = 750. Dans ce cas les secteurs ne sont occupés qu'à 37%, ce qui laisse une marge appréciable pour des adjonctions. Il est probable que les tables de niveau 3 sont plus occupées, mais on ne peut savoir sans connaître la valeur de `N`. Notons que la zone primaire n'est occupée qu'à 5/6 = 85%. Il semble donc que la création a été faite de façon à garder des possibilités assez larges de modifications sans nécessiter de modifier le nombre de niveaux, et donc avec conservation des performances.

3.5.6.1. Question F.1

Si la zone primaire est dans l'espace à tête fixe, le temps d'attente pour les tables est dû uniquement au délai rotationnel, c'est-à-dire 8.3 ms par accès. Le temps d'attente pour l'accès à l'enregistrement lui-même est dû au mouvement de bras auquel il faut ajouter le délai rotationnel, soit 28.3 ms. Au total, on obtient 3 * 8.3 + 28.3 = 53.2 ms.

Si la zone primaire est dans l'espace à bras mobile, le temps d'attente pour les tables est augmenté du mouvement de bras éventuel, soit 28.3 ms par table. Nous aurons alors un temps total de 4 * 28.3 = 113.2 ms. Cependant, si aucun autre accès pour le compte d'un autre processus ne vient perturber la recherche, les trois accès aux tables ont lieu pour la même zone primaire de 193 pistes, qui occupe donc 193/12 = 16 cylindres consécutifs. Ceci représentant moins de 1% des cylindres, on peut éventuellement négliger les déplacements du bras après le premier accès. On obtient alors 28.3 ms pour l'accès au premier niveau, puis 2 fois 8.3 ms pour les accès aux niveaux suivants et enfin 28.3 ms pour l'accès à l'enregistrement proprement dit, soit un total de 73.2 ms.

3.5.6.2. Question F.2

Dans ce cas, seul le dernier niveau de tables entraîne une attente disque. Si la zone primaire est dans l'espace à tête fixe, le temps d'attente sera de 8.3 + 28.3 = 36.6 ms. Si la zone primaire est dans l'espace à bras mobile, le temps d'attente sera de 28.3 + 28.3 = 56.6 ms.

3.5.6.3. Question F.3

	tête fixe	bras mobile	
		optimal	avec mvt.inter.
tout sur disque	53.2	73.2	113.2
2 niv. en MC	36.6	56.6	

Sur le tableau qui résume les résultats, on constate qu'il y a un rapport de 1 à 3 entre le minimum et le maximum. La présence des tête fixe permet de gagner entre 35 et 53% du temps, que les deux niveaux soient ou non en mémoire centrale. Ce gain est appréciable, mais il n'est pas toujours possible de l'obtenir, car l'allocation dans l'espace à tête fixe peut ne pas être possible. Par ailleurs,

la conservation des deux premiers niveaux en mémoire centrale permet d'obtenir un gain compris entre 31 et 50%. Ce gain est ici encore appréciable, et ne demande que de la place en mémoire centrale. Dans l'état actuel de la technologie, 100 Ko de mémoire centrale coûte moins cher que 384 têtes magnétiques fixes avec leur électronique de commande. On peut en conclure que la première amélioration doit être obtenue par la mise en mémoire des deux premiers niveaux. Si ce n'est pas suffisant, on peut alors envisager la solution des têtes fixes.

Mettre tous les niveaux en mémoire centrale permettrait de ramener le temps d'attente à 28.3 ms. Mais cette fois la taille mémoire centrale nécessaire devient très importante. Néanmoins, dans un système doté de pagination à la demande, il pourrait être intéressant de plaquer l'ensemble des tables dans la mémoire virtuelle du processus, et de laisser faire le mécanisme de remplacement de page.

3.5.7. Question G

Si on dispose d'un disque à tête fixe séparé des disques à bras mobile, il est possible d'appliquer la même technique. Les avantages sont évidemment les avantages de performance qui ont été vus dans la question précédente. Il est probable que dans ce cas la capacité du disque sera plus grande, ce qui devrait permettre plus facilement d'y allouer des zones primaires.

Ceci conduit à des fichiers multi-volumes, puisque le fichier est réparti sur plusieurs disques de natures différentes. Si le système ne supporte pas de tels fichiers, ce sera à l'utilisateur de réaliser une partie du travail de partitionnement du fichier (un fichier de tables et un fichier à accès aléatoire par numéro). En dehors de cet aspect, l'inconvénient essentiel est lié à la difficulté de garantir la cohérence entre données situées sur des volumes différents: sauvegarde et restauration simultanée. Ces inconvénients seront négligeables si le système a été conçu pour supporter les fichiers multi-volumes.

3.6. Stratégies d’allocation par zone

On considère un système de gestion de fichiers qui fait de l’allocation par zone. L’ensemble du disque est constitué de 100 blocs, numérotés de 0 à 99. Trois fichiers existent sur le disque, définis comme suit, le reste de l’espace étant libre.

- F1, Début = bloc 5, Taille = 20 blocs,
- F2, Début = bloc 25, Taille = 5 blocs,
- F3, Début = bloc 50 , Taille = 10 blocs.

Les trois questions sont indépendantes, c’est-à-dire que dans chaque cas, on part de la situation ci-dessus.

A- On veut rajouter 10 blocs au fichier F1. Quelles solutions proposez-vous suivant que l’implantation séquentielle est simple ou avec extensions. Justifiez votre raisonnement et évaluez le coût de ces solutions.

B- On veut créer un fichier F4 de 10 blocs. Où proposez-vous de le mettre, en justifiant votre raisonnement.

C- On veut créer un fichier F4 de 40 blocs. Quelles solutions proposez-vous suivant que l’implantation séquentielle est simple ou avec extensions. Justifiez votre raisonnement et évaluez le coût de ces solutions.

Solution de l’exercice 3.6

3.6.1. Question A

Si l’implantation est séquentielle simple, un fichier ne peut avoir qu’une seule zone. Or le fichier F1 se termine sur le bloc 24 et F2 commence en 25. Il n’y a donc pas d’espace libre permettant de le rallonger sans le déplacer. En général, comme le coût d’un tel déplacement est important, il n’est pas fait automatiquement. Cependant, étudions cette possibilité. Il faut alors trouver un espace

contigu de 30 blocs (20 + 10), ce qui peut se faire en 60, derrière F3. Le déplacement lui-même demande la lecture du fichier et sa réécriture dans la nouvelle zone.

Si l'implantation est séquentielle avec extensions, il faut trouver un espace libre de 10 blocs consécutifs correspondant à une nouvelle zone, puisqu'il n'est pas possible de prolonger la première. On peut allouer en 30, derrière F2, en 40 devant F3, en 60 derrière F3 ou en 90 à la fin. Il est cependant préférable de garder intacte la plus grande zone pour satisfaire des besoins ultérieurs importants. Allouer derrière F2 empêchera ce dernier de s'étendre par prolongement, mais allouer devant F3 empêchera F1 d'étendre éventuellement sa deuxième zone, et l'obligera à recevoir une troisième zone. Enfin, on peut noter que les accès à l'intérieur de F1 risquent d'être plus performant si les zones sont les plus proches possibles, ce qui milite pour une allocation en 30.

3.6.2. Question B

Il faut trouver une zone de 10 blocs libre. Elle peut être soit entre 30 et 49, soit entre 60 et 99. Le raisonnement ci-dessus indique une préférence pour la portion 30..49, de façon à conserver la plus grande portion intacte. Le choix est assez indifférent, même en cas d'implantation séquentielle simple, car il faut choisir lequel de F2 ou de F4 sera autorisé éventuellement à s'étendre sur 10 blocs.

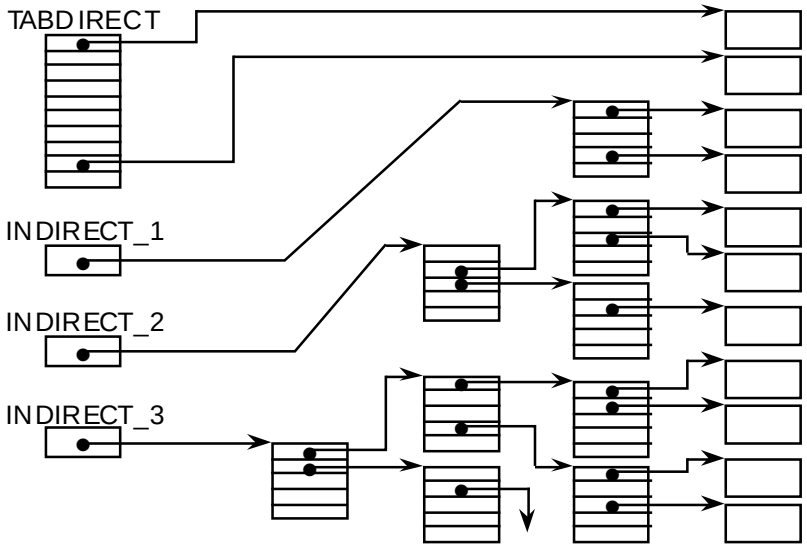
3.6.3. Question C

Il faut trouver cette fois une zone de 40 blocs, ce qui ne peut se faire qu'entre 60 et 99. Notons que si on a une implantation séquentielle avec extensions, il est envisageable de morceler le fichier F4 en plusieurs zones, ce qui n'est pas possible dans le cas de l'implantation séquentielle simple. Cependant, le gain immédiat est nul, et, au contraire, les accès au fichier F4 seront pénalisés dans ce cas, et le gain hypothétique à venir est non prévisible au moment de la création de F4.

3.7. Gestion de fichiers UNIX

Le paragraphe 8.3.2 du cours définit la localisation d'un fichier UNIX par (figure ci-dessous):

- une table, que nous appelons TABDIRECT, donnant les numéros de blocs pour les 10 premiers blocs de l'objet externe,
- un numéro de bloc, que nous appelons INDIRECT_1, qui contiendra les numéros des p blocs suivants de l'objet externe,
- un numéro de bloc, que nous appelons INDIRECT_2, qui contiendra les p numéros de blocs contenant les numéros des p^2 blocs suivants de l'objet externe,
- un numéro de bloc, que nous appelons INDIRECT_3, qui contiendra les p numéros de blocs contenant, au total, p^2 numéros de blocs contenant les numéros des p^3 blocs suivants de l'objet externe.



Nous supposerons que les blocs sont de 1024 octets, et qu'un numéro de bloc occupe 4 octets. Il s'ensuit que p = 256. Par ailleurs, le temps d'accès moyen au disque est de 40 ms.

A- Un processus lit séquentiellement un fichier de 8 Mo, à raison de 256 octets à la fois. Il fait donc 32768 demandes de lecture successives. On suppose qu'il n'y a qu'un seul processus dans le système, et que le système n'utilise pas de tampons de bloc disque, ce qui implique que chaque fois qu'une information située dans un bloc disque est nécessaire, ce bloc doit être lu depuis le disque. Évidemment le descripteur d'un fichier ouvert, c'est-à-dire les informations `TABDIRECT`, `INDIRECT_1`, `INDIRECT_2` et `INDIRECT_3`, restent en mémoire centrale.

A.1- Décrire ce qui se passe lors des deux premières demandes de lecture de 256 octets, puis lors de la 5ième demande.

A.2- Décrire ce qui se passe lors des 41ième et 45ième demandes de lecture de 256 octets.

A.3- Décrire ce qui se passe lors des 1065ième et 1066ième demandes de lecture de 256 octets.

A.4- Décrire ce qui se passe lors des 2089ième et 2090ième demandes de lecture de 256 octets.

A.5- En déduire le nombre total d'accès disque nécessaires et le temps d'attente d'entrées-sorties.

B- Il est dit dans le cours (§8.3.2) qu'Unix utilise une technique de cache des blocs disques. Pour cela, le système dispose d'un tableau de 100 tampons en mémoire centrale, dans lesquels il peut conserver 100 blocs de disque. Lorsque le système a besoin d'un bloc disque, pour lui-même ou pour le compte d'un processus, il recherche dans ces tampons si ce bloc n'est pas déjà en mémoire. S'il n'y est pas, alors, si aucun tampon n'est libre, il commence par en libérer un suivant un algorithme analogue à l'algorithme de remplacement de pages LRU, puis lit le bloc dans un tampon libre et effectue le traitement sur ce bloc. En particulier, lorsqu'un processus demande la lecture ou l'écriture d'octets, le système détermine le bloc disque qui les contient, et lorsque ce bloc est dans un tampon, il réalise le transfert demandé: lecture des octets depuis le tampon ou écriture des octets dans le tampon.

B.1- Les tampons sont dans l'espace du système, et sont partagés entre tous les processus, et non dans l'espace propre de chacun. Expliquez quelles sont à votre avis les raisons de ce choix.

B.2- En supposant les tampons initialement vides, reprendre l'exemple de la question A d'un processus (toujours seul dans le système) lisant séquentiellement un fichier de 8 Mo, à raison de 256 octets à la fois, et décrire ce qui se passe lors des demandes de lecture 1, 2, 5, 41, 45, 1065, 1066, 2089, 2090. En déduire le nombre total d'accès disque nécessaires et le temps d'attente d'entrées-sorties.

C- Le système fonctionne comme en B, et un processus écrit séquentiellement un fichier de 8 Mo, à raison de 256 octets à la fois. On suppose que l'espace disque est déjà complètement alloué au fichier.

C.1- Si on suppose que le système réécrit un tampon sur disque chaque fois qu'il est modifié, évaluez le nombre d'accès disque nécessaires et le temps d'attente d'entrées-sorties.

C.2- Même question qu'en C.1, mais en supposant maintenant qu'un tampon modifié est réécrit sur disque uniquement lorsqu'il y a remplacement. Quels sont les avantages et les inconvénients de cette méthode?

C.3- Expliquer l'intérêt de la primitive système `sync` qui force la réécriture sur disque de tous les tampons modifiés. A votre avis, quand cette primitive doit-elle être appelée?

D- Sur certains systèmes autre que Unix, les contrôleurs disques sont dotés de mémoires caches, de technologie voisine de celle des mémoires centrales, et qui peuvent contenir plusieurs pistes indépendantes. Quelle analogie voyez-vous avec ce qui précède? Si le système implante les objets externes en utilisant une allocation par zone, quel choix de quantum vous paraît alors judicieux?

Solution de l'exercice 3.7.

3.7.1. Question A

3.7.1.1. Question A.1

Lors de la première demande de lecture, le descripteur de fichier contient en `TABDIRECT[0]` le numéro du premier bloc de données. Il faut donc le lire, et transférer les 256 premiers octets (disons ceux de numéro 0 à 255) de ce bloc dans une zone du programme. Lors de la deuxième demande, il s'agit du même bloc, mais comme il n'y a pas de conservation dans un tampon des blocs, il faut le relire, et délivrer les octets 256 à 511. Pour la cinquième demande, il s'agit des octets 0 à 255 du deuxième bloc de données, c'est-à-dire, le bloc de numéro `TABDIRECT[1]`.

3.7.1.2. Question A.2

Les octets de la 41^{ième} demande sont situés dans le 11^{ième} bloc de données. Il faut donc lire le bloc `INDIRECT_1` pour connaître le numéro du bloc de données qui nous intéresse (premier numéro de bloc dans ce bloc), et pouvoir lire celui-ci. Les octets de la 45^{ième} demande sont situés dans le 12^{ième} bloc de données. Il faut donc aussi lire le bloc `INDIRECT_1` pour connaître le numéro du bloc de données qui nous intéresse (deuxième numéro de bloc dans ce bloc), et pouvoir lire celui-ci. Il s'ensuit que chaque demande de lecture, à partir de maintenant entraînera deux accès disque.

3.7.1.3. Question A.3

Les octets de la 1065^{ième} demande sont situés dans le 266^{ième} bloc de données. Il faut donc lire le bloc `INDIRECT_2`, puis celui dont le numéro est le premier dans le bloc venant d'être lu, pour avoir le numéro du bloc de données qui nous intéresse (premier numéro de bloc dans celui venant d'être lu), et lire enfin le bloc de données. Les octets de la 1066^{ième} demande sont situés également dans le 266^{ième} bloc de données. En l'absence de mémorisation, il faudra relire les mêmes blocs que précédemment pour satisfaire la demande. On peut donc en conclure que dorénavant, 3 accès disque seront nécessaires pour satisfaire chaque demande.

3.7.1.4. Question A.4

Les octets de la 2089^{ième} sont situés dans le 522^{ième} bloc de données. Il faut donc lire le bloc `INDIRECT_2`, puis celui dont le numéro est le *deuxième* dans le bloc venant d'être lu, pour avoir le numéro du bloc de données qui nous intéresse (premier numéro de bloc dans celui venant d'être lu), et lire enfin le bloc de données. Les octets de la 2090^{ième} demande sont situés également dans le 522^{ième} bloc de données. En l'absence de mémorisation, il faudra relire les mêmes blocs que précédemment pour satisfaire la demande. On peut donc en conclure que 3 accès disque sont toujours nécessaires pour satisfaire chaque demande.

3.7.1.5. Question A.5

Les 40 premières demandes de lecture demanderont chacune un accès disque. Les 1024 suivantes en demanderont chacune deux. Les demandes restantes (32768 - 1024 - 40 = 31704) en demanderont chacune trois. On a donc un total de 97200 accès disque, et un temps d'attente d'entrées-sorties de 3888 secondes, soit un peu plus d'une heure. Notons que l'indirection de niveau 3 n'est pas utilisée puisque 2 niveaux permettent de mémoriser 65 Mo, alors que nous n'en avons que 8 Mo.

3.7.2. Question B

3.7.2.1. Question B.1

Plusieurs raisons justifient que les tampons soient dans d'espace système, et soient ainsi partagés par tous les processus. Tout d'abord, mentionnons une raison d'économie: s'il fallait réserver des tampons pour chaque processus, il y aurait perte de place pour ceux qui n'accèdent à aucun fichier,

et manque de place pour les autres. Le partage de l'espace permet d'attribuer les tampons lorsque nécessaires.

Cependant, la raison essentielle est liée aux accès simultanés à un même fichier par plusieurs processus. Si plusieurs processus cherchent à lire dans le même fichier, le partage permet de diminuer les accès disque, puisque une lecture d'un bloc dans un tampon partagé pour le compte d'un processus évitera la lecture de ce même bloc pour le compte d'un autre processus. Par ailleurs, si les tampons étaient propres aux processus, une écriture (opération `write`) par un processus sur le fichier, serait en fait une écriture dans ses propres tampons et son effet ne serait visible aux autres processus que plus tard, à un moment imprévisible. La technique de cache des blocs disques doit en fait être «transparente» aux utilisateurs. Les processus manipulent des flots par les opérations `read` et `write`. L'implantation de ces opérations doit respecter la «sémantique» attendue par leur spécification. L'effet sur le contenu du fichier doit être immédiatement visible par les autres processus (la cohérence des données est du ressort de la synchronisation entre les processus).

3.7.2.2. Question B.2

L'utilisation d'un algorithme LRU pour la gestion des tampons signifie que chaque accès à un tampon place celui-ci en tête de liste. Contrairement à la question A, le besoin d'une information située dans un bloc disque ne demande pas d'accès disque, si ce bloc est déjà présent dans un tampon. En reprenant le raisonnement de la question A, on constate que si la première demande, comme la cinquième, demande un accès disque pour obtenir le bloc, par contre la seconde n'en demande pas, puisque le bloc a été conservé dans un tampon. Plus généralement, les 40 premières demandes ne demandent que 10 accès disques pour obtenir les 10 blocs concernés qui tiennent amplement dans les 100 tampons.

Lors de la 41^{ième}demande, il faut lire le bloc pointeur de numéro `INDIRECT_1` dans un tampon, puisqu'une information qui y est contenue (le numéro du bloc de données) est nécessaire. Il faut ensuite lire le bloc de données lui-même. Notons que pour les 3 demandes suivantes, aucun accès n'est nécessaire, puisque les informations sont dans les tampons. Lors de la 45^{ième} demande, le bloc pointeur de numéro `INDIRECT_1` est déjà dans un tampon, et il est alors placé en tête de liste LRU. Le bloc de données est alors lu et placé devant lui dans cette liste. Plus généralement, après chaque demande de lecture (jusqu'à la 1064^{ième} comprise), le bloc de données qui était concerné se trouve en tête de liste LRU, suivi immédiatement par le bloc `INDIRECT_1`. On peut donc en conclure que les 1024 demandes (41 à 1064) nécessitent 257 accès disque (le bloc `INDIRECT_1` et les 256 blocs de données).

Lors de la 1065^{ième} demande, il faut lire le bloc `INDIRECT_2`, puis le bloc pointeur dont le numéro est le premier dans le bloc `INDIRECT_2`, et enfin le bloc de données. Lors de la 1066^{ième} demande, les mêmes blocs sont nécessaires, mais aucun accès n'a lieu, puisqu'ils sont présents dans les tampons. Ici encore, après chaque demande, on constate que le bloc de données est en tête de la liste LRU, suivi du bloc pointeur dont le numéro est le premier du bloc `INDIRECT_2`, suivi du bloc `INDIRECT_2`. Ces blocs pointeurs resteront en mémoire au moins jusqu'à la 2089^{ième} demande. Les demandes 1065 à 2088 demanderont donc 258 accès disques (le bloc `INDIRECT_2`, le premier bloc pointeur et les 256 blocs de données).

Lors de la 2089^{ième} demande, on a besoin du bloc `INDIRECT_2`, toujours présent en mémoire, puis du bloc pointeur dont le numéro est le deuxième dans le bloc `INDIRECT_2`, qui doit donc être lu, et enfin du bloc de données. Ces mêmes blocs seront nécessaires pour la demande 2090. Ici encore, on constate qu'après chaque demande, le bloc de données est en tête de la liste LRU, suivi par le bloc pointeur, suivi par le bloc `INDIRECT_2`. Les 1024 demandes suivantes (2089 à 3112) demanderont donc 257 accès disques (un bloc pointeur et 256 blocs de données), et ainsi de suite.

En résumé, nous avons:

40 demandes avec 10 accès disques

1024 demandes avec 1+256 accès disques

30720 (30*1024) demandes avec 1+30*(1+256) accès disques

984 (32768-40-1024-30720) demandes avec 1+246 accès disques (984/4).

Soit un total de 8225 accès disque, qui se décomposent en 8192 lectures de blocs de données et 33 lectures de blocs pointeurs de blocs. Ceci donne un temps d'attente de 329 secondes, ou 5.5 minutes. Par rapport à la question A, le nombre d'accès disques a été divisé par environ 12! On pouvait s'y attendre: en A, chaque demande en double indirection, c'est-à-dire, les plus nombreuses, demandait la lecture de 3 blocs, alors que maintenant il y a environ une lecture toutes les 4 demandes.

Notons que LRU permet de replacer en début de liste les blocs pointeurs utilisés, ce qui entraîne sur notre exemple que ces blocs ne sont lus qu'une seule fois. Avec l'algorithme FIFO, après avoir lu environ 100 blocs de données repérés par le même bloc pointeur, ce dernier serait chassé de la mémoire pour y être rappelé lors de la demande suivante. Il y aurait donc un peu plus d'accès disque.

3.7.3. Question C

3.7.3.1. Question C.1

Le raisonnement de la question précédente peut s'appliquer également au cas où un processus écrit séquentiellement un fichier. Comme on suppose que l'espace disque est déjà complètement alloué au fichier, lors d'une demande d'écriture de 256 octets, soit le bloc est déjà dans un tampon, soit il s'agit du premier accès à ce bloc, ce qui implique sa lecture préalable puisque la modification est partielle. On peut donc considérer que toute demande d'écriture est identique à une demande de lecture suivie d'une modification du tampon et d'une écriture sur disque du bloc de données. En d'autres termes, on aura le même nombre d'accès disque que précédemment pour les lectures (soit 8225) augmenté des accès disque pour les écritures, soit 32768. Nous avons donc un total de 40993 accès disque, donnant un temps d'attente pour entrées-sorties de 1640 secondes, soit un peu plus de 27 minutes.

3.7.3.2. Question C.2

Si on suppose que les tampons ne sont écrits que lors du remplacement, comme nous avons vu en B qu'un bloc de données n'est lu qu'une seule fois, cela signifie qu'il n'est plus accédé après le premier remplacement dont il fait l'objet (et le seul). Il s'ensuit que chaque bloc de données est écrit exactement une fois. Il n'y a donc plus que 8192 accès disque pour écriture, et donc un total de 16417 accès disque, donnant un temps d'attente de 657 secondes, soit environ 11 minutes.

L'avantage de cette méthode est évidente de par le résultat ci-dessus, puisque le temps d'attente passe de 27 minutes à 11 minutes, soit un gain de 60%. L'inconvénient principal réside dans le risque de ne pas avoir un contenu de fichier cohérent en cas de panne. En effet, si une panne de courant, ou un plantage du système, survient alors que tous les tampons qui ont été modifiés en mémoire centrale n'ont pas encore été réécrits sur disque, il est difficile de savoir dans quel état sont les données. Pour un utilisateur, il peut croire le fichier entièrement modifié, car le processus est terminé, alors qu'il reste une centaine de blocs à réécrire! Si le programme prévoit des affichages intermédiaires pour informer l'utilisateur de l'état d'avancement de l'écriture, on constate que cette information est en fait partielle: les 400 dernières opérations ne sont peut-être pas encore effectives.

Par ailleurs, si tous les processus arrêtent leur activité sur les fichiers, il n'y a plus de remplacement, donc pas d'écriture des derniers blocs actuellement dans les tampons. Si on coupe alors le courant, le contenu de ces derniers blocs sont perdus.

3.7.3.3. Question C.3

La primitive système `sync` garantit, après son exécution, que le disque est cohérent, à ce moment, avec toutes les modifications qui ont été faites par les processus. Il faut donc l'appeler chaque fois que l'on désire avoir cette cohérence. On peut l'appeler périodiquement, par exemple, toutes les 30 secondes, pour se prémunir contre les longues périodes d'inactivités. On doit l'appeler avant d'éteindre le système pour garantir la cohérence des disques avec les dernières modifications. On peut l'appeler lors de la fermeture du fichier (mais c'est souvent fait implicitement par le système).

3.7.4. Question D

Lorsque les contrôleurs disques sont dotés de mémoires caches pouvant contenir plusieurs pistes indépendantes, cela veut dire que toute demande de lecture d'un secteur quelconque entraîne d'abord la lecture de la piste complète qui le contient dans une portion de cette mémoire cache, puis le transfert du (ou des) secteur demandé dans la mémoire centrale. La mémoire cache joue le rôle des tampons des questions B et C, le bloc étant alors l'équivalent d'une piste. Il s'ensuit que la lecture d'un secteur situé dans une piste déjà présente dans la mémoire cache ne nécessite aucun accès disque, mais seulement le transfert des données entre la mémoire cache et la mémoire centrale. Si ce transfert n'est pas instantané, il peut cependant se faire en, par exemple, 200 µs pour 1 Ko.

Pour améliorer la probabilité que, lors d'une demande, le secteur soit présent dans la mémoire cache, il est préférable qu'une piste contienne des informations appartenant à un seul fichier, plutôt qu'à plusieurs fichiers. Ceci peut être obtenu avec une allocation par zone lorsque les zones sont allouées par pistes entières, et donc si le quantum est un nombre entier de pistes. En particulier, on pourra prendre un quantum de une piste, si cela ne conduit pas à un nombre trop important de quanta.

3.8. Mesures sur un système de gestion de fichiers à base de FAT

On dispose d'un ordinateur personnel équipé de deux disques durs identiques d'environ 50 Mo. chacun. Les caractéristiques annoncées par son fabricant sont 800 cylindres, 4 têtes, 34 secteurs par piste, 512 octets par secteur, 3600 tours par minute, et 25 ms de temps moyen de positionnement. La gestion des fichiers est analogue à celle du système MS-DOS. Sur chacun des disques, on a défini une partition d'environ 15 Mo. Rappelons que (voir §11.1):

- Une partition de MS-DOS est une portion de disque dur constituée d'un ensemble de cylindres contigus (ici 220 cylindres), et structurée en volume. Les deux partitions ont reçu comme nom de volume G et H.
- L'implantation des objets sur le volume utilise la technique des blocs chaînés, à l'aide d'une *File Allocation Table* (FAT). Les blocs ou *clusters* sont ici constitués de 2 secteurs.
- Pour des raisons de sécurité, cette table existe en deux exemplaires en début de partition.
- Les objets externes du volume sont organisés en utilisant une arborescence de répertoires.

Lors d'une copie du volume G contenant 587 fichiers répartis sur 44 répertoires et occupant un total de 10.5 Mo. sur le volume H entièrement vide, on a constaté un temps total particulièrement long de 53 minutes. Les voyants montrent une activité négligeable du disque sur lequel se fait la lecture, et très importante de celui sur lequel se fait l'écriture. Le but du problème est de comprendre cette anomalie.

A.1- Combien d'entrées comporte une FAT? Combien de secteurs occupe un exemplaire de FAT?

A.2- Décrire brièvement les étapes de l'ouverture d'un fichier qui n'existe pas et qui doit donc être créé.

A.3- Décrire brièvement les étapes de l'écriture d'un tampon de mémoire centrale de 1 Mo dans un fichier ouvert, qui vient d'être créé (un tel tampon est compatible avec la taille de mémoire centrale qui est de 4 Mo).

B- Après avoir sauvegardé le contenu de G, on vide complètement G et H et on crée un fichier F de 1 Mo sur le volume G. On recopie ce fichier F de G sur H, et on appelle C cette copie. Le programme de copie, dans ce cas, a la forme suivante:

```
ouvrir (f, "G:\F"); /* ouverture du fichier F sur G */
lire (f, Tloc, 1048576); /* lecture de 1 Mo */
fermer (f);
ouvrir (f, "H:\C"); /* ouverture/création fichier C sur H */
écrire (f, Tloc, 1048576); /* écriture de 1 Mo */
fermer (f);
```

On constate, au cours de cette expérience, que les durées respectives de la lecture et de l'écriture sont sensiblement égales, soit 6 secondes chacune, et la durée totale de l'opération est donc de 12 secondes.

B.1- Etant donné la technique utilisée pour l'allocation d'espace aux fichiers, peut-on affirmer que le fichier origine et sa copie sont dans des secteurs consécutifs?

B.2- Pensez-vous que le système en tire profit pour lancer les lectures et écritures disque sur plusieurs secteurs consécutifs?

C- On répète la copie de F sur H, dans des fichiers différents (C1, C2, C3, etc...), en mesurant à chaque fois la durée de cette copie. On constate que chaque copie dure environ 1 seconde de plus que la précédente, c'est-à-dire que la création de la copie C1 dure 13 secondes, celle de C2 dure 14 secondes, celle de C3 dure 15 secondes, etc... Comme on lit toujours le même fichier F sur G, cette augmentation est en fait une augmentation du temps de création et d'écriture de la copie.

C.1- Déterminer ce qui dans la création et l'écriture peut expliquer cet allongement.

C.2- En déduire que l'on peut évaluer le temps en millisecondes d'une copie d'un fichier de L Ko. sur un disque dont T blocs sont déjà occupés par la formule: D = T + 12 * L.

D- En supposant que les 587 fichiers évoqués dans l'introduction sont tous de même taille, et en négligeant le temps de création des 44 répertoires, déduire de C.2 la durée de leur copie de G sur H. A-t-on une explication satisfaisante de l'anomalie évoquée au début?

Solution de l'exercice 3.8

3.8.1. Question A

3.8.1.1. Question A.1

Une partition est de 220 cylindres, chaque cylindre ayant 4 têtes, les pistes ayant 34 secteurs, cela donne donc 220 * 4 * 34 = 29920 secteurs. Les blocs étant de 2 secteurs, il y a donc 14960 blocs allouables. En fait il y en a un peu moins, puisque les deux exemplaires de FAT vont en occuper quelques uns. Un numéro de bloc a besoin de 14 bits, donc 2 octets, et chaque exemplaire de FAT occupe 29920 octets, ce qui demande 59 secteurs. On peut donc voir qu'il n'y a en fait que 14900 blocs allouables.

3.8.1.2. Question A.2

Lors de l'ouverture d'un fichier qui n'existe pas et qui doit être créé, le système exécute les étapes suivantes:

- Localisation du volume, permettant ici de savoir quel est le périphérique concerné, et sur quels cylindres se trouve la partition.
- Parcours des noms successifs de la chaîne d'accès pour descendre dans l'arborescence jusqu'à localiser le dernier répertoire.
- Recherche du nom du fichier dans le répertoire, et en même temps de la première entrée libre du répertoire. Puisque le fichier n'existe pas, le nom n'est pas trouvé.
- Création de l'entrée du répertoire pour ce fichier. Il n'y a pas d'allocation d'espace au fichier qui est vide, c'est-à-dire, que le numéro de son premier bloc indique `EOF`. Cependant, si la première entrée libre trouvée est dans un emplacement non alloué du répertoire, il peut y avoir allocation d'un bloc au répertoire.
- Réécriture sur le disque du secteur du répertoire qui contient cette entrée.
- Conservation du descripteur du fichier vide dans une structure de données relative au flot, et délivrance d'un «identifiant» de cette structure comme résultat de l'opération.

3.8.1.3. Question A.3

Lors de l'écriture d'un tampon de mémoire centrale de 1 Mo dans un fichier ouvert, qui vient d'être créé, il faut d'abord allouer 1024 blocs au fichier puisqu'il est vide. Le système exécute les étapes suivantes:

- Recherche à partir du début de la FAT du premier bloc libre, et mise de son numéro dans le descripteur du fichier.
- Recherche ensuite du bloc libre suivant, chaînage de ce bloc au précédent dans la FAT, et répétition de l'opération jusqu'à avoir alloué les 1024 blocs.
- Recopie des parties de FAT modifiées dans le premier exemplaire, puis le second exemplaire sur disque.
- Écriture du contenu du tampon dans les 1024 blocs ainsi alloués.

De plus, deux remarques peuvent être faites. D'une part, au cours de la recherche des blocs libres, il peut être nécessaire de lire les secteurs de la FAT s'ils ne sont pas résidents en mémoire. D'autre part, le système peut recopier l'entrée du répertoire correspondant au fichier sur disque, puisqu'elle est modifiée (premier bloc et longueur du fichier), ou au contraire attendre que le fichier soit fermé pour le faire, puisque c'est à ce moment seulement que l'on connaît la longueur définitive du fichier.

3.8.2. Question B

3.8.2.1. Question B.1

Puisque le fichier F a été créé sur G vide, toutes les entrées de la FAT étaient libres au moment de l'écriture du contenu de F. L'algorithme précédent d'allocation donnera 1024 blocs successifs. Le fichier F occupera donc bien des secteurs consécutifs. Lors de la copie il en est de même du fichier C créé sur H initialement vide.

3.8.2.2. Question B.2

Les durées respectives de la lecture et de l'écriture étant sensiblement égales, la lecture demande 6 secondes pour 2048 secteurs, soit en moyenne 2.9 ms par secteur. Si le système lançait les lectures et écritures un secteur à la fois, comme ils sont consécutifs, le secteur suivant aurait commencé à passer sous les têtes au moment du lancement de l'opération le concernant, obligeant d'attendre un tour complet pour le lire, soit 16.7 ms. Il est certain donc que le système lance la lecture de plusieurs secteurs à la fois. Si maintenant le système demande la lecture de n secteurs à la fois, cela entraînera la perte d'un tour tous les n secteurs, c'est-à-dire, que la lecture de ces n secteurs demandera 16.7 + n * (16.7 / 34) ms. On a donc l'inéquation:

$$16.7 + n * (16.7 / 34) \leq n * 2.9$$

On en déduit que n est au moins égal à 7. Le système a certainement tenu compte du fait que les données sont dans des secteurs consécutifs.

3.8.3. Question C

3.8.3.1. Question C.1

Considérons les étapes de la création du fichier. Dans notre cas, toutes les copies sont mises dans le répertoire racine, dont le nombre d'entrées est faible. On peut donc penser que les opérations suivantes sont indépendantes de l'occupation du volume: localisation du volume, localisation du dernier répertoire, création de l'entrée du répertoire pour ce fichier, réécriture du secteur du répertoire. La recherche du nom du fichier dans le répertoire est effectivement un peu plus longue à chaque fois, puisqu'il y a une entrée supplémentaire. Cependant, il serait surprenant que cela entraîne un allongement de une seconde par entrée.

Considérons maintenant les étapes de l'écriture d'un tampon de 1 Mo. Lors de la recherche à partir du début de la FAT du premier bloc libre, il faudra parcourir 1024 entrées occupées supplémentaires à chaque fois. Une fois ce premier bloc trouvé, les autres seront immédiatement derrière, comme lors de l'écriture de la copie C. La réécriture des secteurs de FAT modifiés prendra

toujours le même temps, puisqu'il s'agit de réécrire, dans chaque exemplaire de FAT, au plus 5 secteurs consécutifs, pour les 1024 entrées successives modifiées (5 parce que la première entrée n'est pas en début de secteur). Enfin, l'écriture du contenu du tampon dans les 1024 blocs alloués, c'est-à-dire 2048 secteurs consécutifs devrait prendre le même temps si ce n'est le mouvement de bras pour se positionner sur le premier secteur, mais cela devrait prendre environ 25 ms environ.

En conclusion, deux opérations paraissent dépendre du nombre de copies qui ont déjà été faites:

- la recherche dans le répertoire racine, qui demande à parcourir une entrée supplémentaire dans ce répertoire,
- la recherche du premier bloc libre dans la FAT, qui demande à parcourir 1024 entrées supplémentaires de la FAT.

Manifestement c'est cette dernière opération qui est la cause de l'allongement.

3.8.3.2. Question C.2

La question précédente montre que ce système parcourt, en une seconde, 1024 entrées de la FAT lors d'une recherche de premier bloc libre, ce qui donne environ 1 ms par bloc occupé. Si T blocs sont déjà occupés sur un disque, la recherche du premier bloc libre lors de la copie d'un fichier demandera T ms. Par ailleurs, la copie d'un fichier de 1 Mo sur un volume vide demande 12 secondes, donc environ 12 ms par Ko à copier. On obtient bien:

$$D = T + 12 * L$$

3.8.4. Question D

Les 587 fichiers de l'introduction occupent 10.5 Mo. On en déduit donc une taille moyenne de 10.5 * 1024 / 587 = 18.3 Ko par fichier. Si lors de la copie du premier fichier, l'espace sur H est entièrement libre, lors de la copie du $i^{ème}$ fichier, il y a déjà i - 1 fichiers de 18.3 Ko sur H, occupant 18.3 * (i - 1) blocs. La copie de ce fichier demande donc, d'après C.2,

$$D_i = 18.3 * (i - 1) + 12 * 18.3 = 18.3 * (i + 11) \text{ ms}$$

La copie des 587 fichiers demandera donc:

$$D = \sum_{i=1}^{587} 18.3 * (i + 11)$$

$$D = 18.3 * \left(\left(\sum_{i=1}^{587} i \right) + 587 * 11 \right)$$

$$D = 18.3 * \left(\frac{587 * 588}{2} + 587 * 11 \right) \approx 3280_{10}{}^{3}$$

La durée totale estimée est donc d'environ 54.6 minutes, ce qui est à 3% près la durée mesurée. On peut donc penser que l'on a une explication satisfaisante de l'anomalie évoquée au début de l'énoncé.

3.9. Gestion de fichiers par mémoire virtuelle

On dispose d'un système doté d'une mémoire virtuelle paginée. La taille d'une page est de 1 Ko. La pagination est à deux niveaux (§14.3.2), chaque processus ayant sa propre table des hyperpages. La dernière hyperpage de chaque processus est gérée par le système de gestion de fichiers qui peut y mettre des tampons d'entrées sorties fichiers. Lorsqu'un défaut de page se produit dans cette hyperpage, le défaut est traité par le système de gestion de fichiers. Les disques ont des secteurs de 1 Ko. Ils servent à la fois à la mémoire virtuelle et aux fichiers.

Le système de gestion de fichiers a été décomposé en un noyau, qui offre un ensemble minimal de services, et des modules indépendants en charge des opérations d'accès proprement dites. Nous nous intéressons surtout aux services offerts par le noyau, dont les fonctions essentielles sont la

gestion des descripteurs de fichiers, l'allocation (par zone) et la libération d'espace disque, et les entrées-sorties disque couplées avec la gestion d'un ensemble de tampons de mémoire centrale.

Sur chaque volume, une partie de l'espace est réservée pour contenir des descripteurs de fichiers. Un descripteur occupe un secteur, et a pour numéro le numéro relatif du secteur dans la partie réservée aux descripteurs. Un descripteur de fichier a la structure suivante:

- partie gérée par le noyau (tous les entiers sur 32 bits)
  - `tbp`, taille des blocs physiques en nombre de secteurs,
  - `tz`, taille des zones d'allocation en nombre de blocs physiques,
  - `nz`, nombre de zones allouées,
  - `debz`, table des numéros de premier secteur des zones allouées (cette table comporte 200 entrées),
  - `dates` de création, de modification et d'utilisation (chacune sur 4 octets),
  - `protection` (32 octets), utilisé par le noyau pour assurer la protection,
- partie non gérée par le noyau, et à disposition des modules (128 octets).

Les opérations du noyau sont les suivantes:

`créer_descripteur (DCB, d)` Le noyau recherche un descripteur libre, et met son numéro dans `d`. Ensuite il initialise ce descripteur en utilisant les informations trouvées dans le `DCB` (Data Control Bloc), et met à jour le `DCB` de façon à permettre l'accès au fichier ainsi créé.

`lire_descripteur (DCB, d)` Le noyau vérifie que le demandeur est autorisé à accéder au fichier repéré par le descripteur de numéro `d`, et met à jour le `DCB` de façon à permettre l'accès demandé. En particulier, la partie du descripteur non gérée par le noyau est mise dans le `DCB`.

`libérer_descripteur (DCB)` Le noyau met à jour, si nécessaire, le descripteur associé, y compris la partie non gérée par le noyau, et modifie le `DCB` de façon à interdire tout accès par son intermédiaire.

`prendre_bloc (DCB, n, mode)` Le noyau vérifie que l'accès demandé par le `mode` (lecture, écriture ou mise à jour) est autorisé sur ce `DCB`. Si le bloc `n` du fichier n'est pas présent en mémoire centrale, le noyau alloue un tampon en mémoire centrale, et y lit ce bloc depuis le disque sauf si le mode est en écriture. Puis il introduit ce tampon dans l'espace virtuel du processus en mettant à jour les entrées correspondantes de la table des pages de la dernière hyperpage, et retourne l'adresse virtuelle de ce tampon.

`rendre_bloc (DCB, n)` Le noyau vérifie que le bloc `n` du fichier est bien dans l'espace virtuel du processus, et l'enlève de cet espace, en mettant à jour la table des pages de la dernière hyperpage du processus. Le bloc reste en mémoire centrale, tant que l'espace qu'il occupe n'est pas réquisitionné. S'il a été modifié, il sera réécrit sur disque avant cette réquisition.

`sauver_bloc (DCB, n)` L'opération est similaire à `rendre_bloc`, si ce n'est que la réécriture sur disque est immédiate.

A- Expliquez l'utilité des informations de la partie du descripteur gérée par le noyau.

B- Montrer succinctement comment l'on peut réaliser un module qui implante un fichier séquentiel d’enregistrements logiques de taille variable. Quelles informations faudrait-il mettre dans la partie du descripteur qui est à la disposition des modules?

C- Les répertoires ne sont pas gérés par le noyau.

C.1- Montrez comment ils peuvent l'être par un module spécifique? Quelles informations faudrait-il mettre dans la partie de leurs descripteurs qui est à disposition des modules?

C.2- Peut-on imaginer que le fabricant de ce système propose plusieurs organisations possibles des répertoires? Quels en seraient les avantages et les inconvénients?

D- Le paramètre `mode` de l'opération `prendre_bloc` indique le type d'accès désiré sur le tampon. Le système a-t-il les moyens, et comment, de vérifier que le module ou le processus utilisateur n'écrira pas dans un tampon pour lequel il a demandé la lecture seule?

E- Expliquez comment le noyau peut savoir si le tampon est ou non à réécrire lors de l'opération `rendre_bloc`.

F- Si le paramètre `mode` précise en plus, si l'appelant veut l'accès partagé ou exclusif, comment le noyau peut-il réaliser le partage physique du tampon entre plusieurs processus?

G- Le noyau propose en fait trois méthodes pour gérer les tampons de bloc en mémoire centrale, au choix.

G.1- Les blocs, qui sont dans l'espace virtuel d'un processus, restent en mémoire centrale, qui ne peut en contenir qu'un nombre limité. Dans quels cas ceci peut conduire à un interblocage ou à la famine?

G.2- Le noyau dispose d'un espace disque séparé de l'espace des fichiers, comme espace de pagination. Une page, appartenant à un tampon dans l'espace virtuel d'un processus est soit en mémoire, soit dans l'espace de pagination du noyau, qui traite les défauts de page correspondants. Lorsqu'un bloc est rendu par un processus, ses pages peuvent rester en mémoire centrale, mais pas dans l'espace de pagination. Montrez que ceci peut toujours conduire à un interblocage ou à une famine, mais avec une probabilité bien plus faible.

G.3- Le noyau utilise l'espace des fichiers comme espace de pagination. Lorsqu'une page appartenant à un tampon n'est pas présente en mémoire, elle réside sur le disque, à la place correspondante au bloc qui est associé au tampon. Quels sont les avantages et inconvénients de cette méthode, comparée aux deux précédentes?

H- Comment jugez-vous ce système? Quels avantages ou inconvénients y voyez-vous?

Solution de l'exercice 3.9

3.9.1. Question A.

L'information `tpb` permet d'avoir des blocs physiques plus grands qu'un secteur. Les transferts disque font intervenir deux paramètres essentiels, le temps d'accès au premier secteur à transférer, et le nombre total de secteurs à transférer. Le fait de transférer plusieurs secteurs à la fois permet d'amortir le coût du temps d'accès sur plusieurs secteurs au lieu d'un seul. De plus, le bloc physique étant la quantité d'informations minimale transférée en une seule opération, représente l'unité minimale de cohérence. L'information permet donc d'augmenter cette unité minimale de cohérence au delà d'un secteur.

L'information `tz` permet au noyau de savoir quelle est la taille de la zone à allouer, lorsque ceci s'avère nécessaire. D'autre part, elle permet de savoir à quelle zone appartient un bloc de numéro `n`, puisqu'il s'agit de la zone `n`**`div`**`tz`, ainsi que le numéro du bloc dans cette zone, puisqu'il s'agit du bloc `n`**`mod`**`tz`.

L'information `nz` permet de savoir quelles sont les entrées de la table `debz` qui ont une signification, puisqu'il s'agit de celles de numéro inférieur à `nz`. En particulier, lors d'un accès au bloc `n`, le bloc est dans une zone allouée si et seulement si `n`**`div`**`tz < nz`. Remarquons que l'on peut déduire que l'espace total alloué au fichier est `tz*nz*tbp` Ko.

L'information `debz` permet de localiser les secteurs où commence chaque zone du fichier, et donc d'en déduire le secteur où commence un bloc `n` donné, puisqu'il s'agit du secteur `debz[n`**`div`**`tz]+n`**`mod`**`tz`. Par ailleurs, lorsque le fichier est détruit, cette table permet de savoir quelles zones doivent être récupérées.

L'information `dates` permet de savoir si le fichier est à jour, et utile. Certains outils utilisent ces informations (`make`), pour savoir si des fichiers ont besoin d'être reconstruits. Ces informations peuvent être utilisées par les programmes de sauvegarde incrémentale, pour ne sauvegarder que ce

qui a été modifié depuis la dernière sauvegarde. Elles peuvent être utilisées d'autorité pour détruire des fichiers (après sauvegarde éventuelle) qui ne sont pas utilisés pendant une très longue période.

L'information `protection` permet d'assurer une protection du fichier contre les actions intempestives, erronées ou malveillantes des utilisateurs non autorisés. On a peu d'information sur la technique mise en œuvre ici. Le fait de disposer de 32 octets devrait permettre de disposer d'un mot de passe, et d'un début de liste d'accès.

3.9.2. Question B.

Pour implanter un fichier séquentiel d'enregistrements de longueur variable, on peut s'inspirer du du problème 3.2, si on cherche à avoir une certaine compatibilité avec les structures utilisées sur les supports séquentiels. Pour simplifier, nous prenons la structure suivante:

```
long_bloc : 2 octets;
    { long_enrg :      2 octets ; }  répété k fois
    { enregistrement : p octets ; }
```

Certaines informations doivent être retrouvées d'un accès au fichier à l'autre, et qui ne peuvent être déduites de celles qui sont dans la partie du descripteur gérée par le noyau. Ces informations devront donc se trouver dans la partie du descripteur qui est à la disposition du module. Elles se retrouveront automatiquement dans le `DCB`, d'après les opérations `lire_descripteur` et `libérer_descripteur`. Ces informations sont les suivantes:

`type` indiquant qu'il s'agit d'un fichier séquentiel,

`nb_enrg` indiquant le nombre d'enregistrement logique du fichier,

`nb_blocs` indiquant le nombre de blocs effectivement occupés.

Le nombre d'enregistrements logiques est l'information qui permet, lors de la relecture, de savoir quand on a atteint la fin du fichier. Le nombre de blocs occupés est intéressant pour pouvoir reprendre le fichier, en conservant ce qu'il contient déjà, et en lui ajoutant de nouveaux enregistrements. Notons que nous aurons besoin de l'information `tbp` lors des opérations d'écriture, pour savoir quand un bloc est plein.

Définissons d'abord les variables internes au module:

```
var position : entier;  /* position dans bloc courant */
    numéro : entier;    /* numéro du bloc courant */
    ad_tampon : pointeur tampon; /* adresse mémoire tampon courant */
    taille_max : entier; /* taille maximum 1 bloc en octets */
    num_enrg : entier;      /* numéro enregist. courant lecture */
    long : entier;          /* longueur du bloc courant lecture */
```

Les opérations pour la lecture pourraient être:

```
procédure initialisation_lire (d : entier);
début position := 2;
      long := 0; /* variables internes au module */
      numéro := -1;
      num_enrg := 0;
      lire_descripteur (DCB, d);
fin;
procédure lire_enrg (var enregistrement: tableau_octets; var l_enrg: entier);
début si num_enrg ≥ DCB.nb_enrg alors erreur ("fin de fichier"); finsi;
      si position >= long - 4 alors
         si numéro ≥ 0 alors rendre_bloc (DCB, numéro); finsi;
         numéro := numéro + 1;
         ad_tampon := prendre_bloc (DCB, numéro, lecture);
         long := tampon[0 .. 1];
         position := 2;
      finsi;
      l_enrg := tampon[position .. position + 1];
      enregistrement := tampon[position+2 .. position+1+l_enrg];
      position := position + 2 + l_enrg;
      num_enrg := num_enrg + 1;
fin;
```

```
procédure cloture_lecture;
début si numéro ≥ 0 alors rendre_bloc (DCB, numéro); finsi;
       libérer_descripteur (DCB);
fin;
```

Les opérations pour l'écriture pourraient être:

```
procédure initialisation_écrire (d : entier; en_prolongement : booléen);
début lire_descripteur (DCB, d);
       si en_prolongement alors numéro := DCB.nb_blocs; /* blocs déjà écrits */
       sinon numéro := 0; DCB.nb_enrg := 0;/* fichier vide */
       finsi;
       taille_max := DCB.tbp * 1024; /* taille maximum un bloc */
       ad_tampon := prendre_bloc (DCB, numéro, écriture); /*premier bloc vide */
       position := 2;
fin;
procédure écrire_enrg (enregistrement : tableau_octets; l_enrg : entier);
début si position + l_enrg + 2 >= taille_max alors
          tampon[0 .. 1] := position;
          rendre_bloc (DCB, numéro);
          numéro := numéro + 1;
          ad_tampon := prendre_bloc (DCB, numéro, écriture);
          position := 2;
       finsi;
       tampon[position .. position + 1] := l_enrg;
       tampon[position+2 .. position+1+l_enrg] := enregistrement;
       position := position + 2 + l_enrg;
       DCB.nb_enrg := DCB.nb_enrg + 1;
fin;
procédure cloture_écriture;
début si position ≠ 2 alors /* dernier bloc à écrire ? */
          tampon[0 .. 1] := position;
          rendre_bloc (DCB, numéro);
          numéro := numéro + 1; /* un bloc de plus */
       sinon rendre_bloc (DCB, numéro); /* premier bloc vide */
       finsi;
       DCB.nb_blocs := numéro;
       libérer_descripteur (DCB);
fin;
```

3.9.3. Question C

3.9.3.1. Question C.1

Le but des répertoires est d'établir une table de correspondance entre un nom symbolique, qui est une chaîne de caractères, et un descripteur de fichier. Dans ce système, un descripteur de fichier est repéré par un numéro. Un répertoire doit donc mettre en correspondance des chaînes de caractères et des entiers. Il doit être mémorisé sur un support permanent. Il est naturel d'utiliser un fichier pour cela. Nous appelons répertoire désigné par `d` le répertoire contenu dans le fichier de descripteur `d`. Évidemment un même volume peut contenir plusieurs répertoires.

Les opérations essentielles d'un module de gestion des répertoires pourraient être les suivantes:

`recherche (d, nom)` recherche le `nom` dans le répertoire désigné par `d`, et retourne l'entier qui lui est associé.

`créer (d, nom, d')` vérifie que le `nom` n'est pas déjà mémorisé dans le répertoire désigné par `d`, et y ajoute la correspondance entre `nom` et `d'`.

`supprimer (d, nom)` supprime le `nom` dans le répertoire désigné par `d`.

Le descripteur correspondant à un répertoire pourrait contenir par exemple les informations suivantes:

`type` précisant qu'il s'agit d'un répertoire,

`nb_entrées` indiquant combien il y a d'entrées dans ce répertoire,

`nb_blocs` indiquant le nombre de blocs utilisés.

Le schéma simplifié de la procédure `recherche` est donné ci-dessous, en supposant qu'il y a 32 entrées par bloc. On ne prend pas en compte les entrées inutilisées.

```
procédure recherche (d:entier; nom:chaîne) retourne entier;
début lire_descripteur (DCB, d);
       si DCB.type ≠ répertoire alors erreur ("pas un répertoire"); finsi;
       pour i de 0 à DCB.nb_blocs - 1 faire
          ad_tampon := prendre_bloc (DCB, i, lecture);
          pour j de 0 à 31 faire
               si tampon[j].nom = nom alors
                    num_descr := tampon[j].descr;
                    rendre_bloc (DCB, i);
                    libérer_descripteur (DCB);
                    retourner (num_descr);
               finsi;
          fait;
          rendre_bloc (DCB, i);
       fait;
       libérer_descripteur (DCB);
       erreur ("entrée non trouvée");
fin;
```

Un tel module permet, soit de faire un répertoire unique, situé dans un descripteur du volume, par exemple le premier descripteur, soit de faire une arborescence de répertoire sur le volume, dont la racine serait le premier descripteur. Le schéma simplifié de la procédure d'ouverture d'un fichier existant est donné ci-dessous, en supposant qu'on dispose de la procédure `extraire_suivant` qui extrait le prochain `nom` du `chemin_restant`.

```
procédure ouverture (chemin_d'accès : chaîne);
var d : entier;
     chemin_restant, nom : chaîne;
début d := 1; /* racine de l'arborescence */
       chemin_restant := chemin_d'accès;
       tant que chemin_restant ≠ "" faire
          extraire_suivant (chemin_restant, nom);
          d := recherche (d, nom);
       fait;
       lire_descripteur (DCB, d); /* le DCB repère le fichier */
fin;
```

3.9.3.2. Question C.2

La décomposition du système de gestion de fichiers en un noyau offrant les services de base, et des modules utilisant ces services, offre une certaine souplesse. En particulier, la question précédente montre que la structure et l'organisation des répertoires peut effectivement être reporté dans les modules, et ne pas se situer dans le noyau. Il s'ensuit qu'avec un même noyau, il est possible d'avoir plusieurs organisations possibles. Comme les différentes organisations des fichiers de données sont traitées également chacune par un module spécifique, il est possible de changer l'organisation des répertoires sans changer les structures des fichiers de données.

Cette souplesse a cependant des limites. Il est faisable, mais délicat d'avoir plusieurs organisations de répertoires qui coexistent sur un même volume ou dans un même système. En effet, l'information de `type` permet toujours de savoir quelle est l'organisation particulière d'un répertoire et le module qui le gère. Mais cette diversité peut apporter une gène à l'utilisateur par la complexité qu'elle engendre.

3.9.4. Question D

Pour empêcher le module ou le processus d'écrire dans un tampon pour le quel il a demandé la lecture seule, il suffit de positionner correctement les indicateurs de `protection` des entrées de la table des pages qui correspondent aux pages du tampon. S'il y a tentative d'écriture dans ces pages, le matériel provoquera un déroutement, et le système en sera ainsi informé.

3.9.5. Question E

Pour savoir si une page a été modifiée, il suffit de mettre initialement à 0 l'indicateur de `page_modifiée` des entrées de la table des pages qui correspondent aux pages du tampon. Si l'une d'elles est modifiée, son indicateur sera mis à 1 automatiquement par le matériel. Lorsque le processus effectuera l'opération `rendre_bloc`, il suffira de consulter cet indicateur pour toutes les pages du bloc.

3.9.6. Question F

Dans un système paginé, si l'entrée `i` de la table des pages d'un processus `P` repère une case `C` de mémoire centrale, et l'entrée `j` de la table des pages d'un processus `P'` repère également la même case `C`, alors les deux processus `P` et `P'` partagent physiquement toutes les données situées dans la case `C`. Ces données sont vues comme étant dans la page virtuelle `i` par le processus `P`, et dans la page virtuelle `j` par le processus `P'`. Lorsque le noyau décide de faire partager un même tampon par deux processus `P` et `P'`, ce tampon étant situé dans les cases `C1`, `C2`, ..., `Ctbp`, le noyau choisit d'abord les pages consécutives `i..i+tbp-1` pour le processus `P`, et initialise les entrées `i..i+tbp-1` de la table des pages de la dernière hyperpage aux numéros de cases `C1`, `C2`, ..., `Ctbp`, puis il choisit les pages consécutives `j..j+tbp-1` pour le processus `P'`, et initialise les entrées `j..j+tbp-1` de la table des pages de la dernière hyperpage aux numéros de cases `C1`, `C2`, ..., `Ctbp`.

3.9.7. Question G

3.9.7.1. Question G.1

Le noyau dispose d'une taille de mémoire centrale fixe pour y mettre des tampons pour les processus. Si l'ensemble des processus cherchent à disposer en même temps d'un nombre de blocs plus important que ce que peut contenir la mémoire centrale, toutes leurs demandes ne pourront être satisfaites.

- Si on les met en attente, il y a risque d'interblocage, puisque tous les processus peuvent être en attente d'un bloc, alors qu'ils en possèdent déjà, et qu'il n'y a plus d'espace disponible. Chacun attend que l'autre rende les blocs qu'il possède.
- Si on refuse leur demande, en suggérant qu'ils la refassent plus tard, un processus malchanceux peut se voir refuser sa demande de façon répétitive, parce que les autres se sont coalisés pour prendre l'espace disponible dès qu'il est libéré par l'un d'eux. Il est alors en situation de famine.

3.9.7.2. Question G.2

Le problème est en fait le même que le précédent. La seule différence est dans le nombre de ressources dont dispose le noyau. Ici l'espace est limité non plus par la taille de mémoire centrale gérée par le noyau, mais par la taille de l'espace de pagination qu'il gère. Comme en général cet espace est beaucoup plus grand que l'espace de mémoire centrale, la probabilité de le saturer est très faible. Le risque d'interblocage ou de famine a donc beaucoup moins de chance de se produire. Évidemment si la demande est forte, les tampons ne seront pas tous en mémoire centrale, et il y aura des défauts de page pour les amener en mémoire au fur et à mesure des besoins.

3.9.7.3. Question G.3

Lorsque le noyau utilise l'espace de fichier comme espace de pagination, cela implique qu'il n'a plus à allouer d'espace pour les tampons des blocs demandés par les processus. Le noyau a simplement à gérer les défauts de page consécutifs aux accès par les processus aux pages contenant les blocs qu'ils possèdent. Il n'y a plus d'interblocage ni de famine consécutive à la gestion des tampons.

Le travail du noyau est aussi simplifié. Lors de l'opération `prendre_bloc`, il n'est pas nécessaire de lire le bloc physique en mémoire centrale. Le noyau doit simplement choisir un ensemble de pages virtuelles consécutives, et mémoriser la correspondance avec les secteurs du bloc. Les transferts effectifs auront lieu lors des défauts de page ultérieurs provoqués par le processus. De même l'opération `rendre_bloc` consiste simplement en des modifications des entrées correspondantes de la table des pages de la dernière hyperpage. Notons que les transferts disque se faisant alors par le

mécanisme de page à la demande, le paramètre `tbp` n'intervient pas dans les transferts, qui se font toujours un secteur à la fois.

Cette méthode présente l'inconvénient de perdre un certain contrôle de cohérence que pouvaient avoir les processus sur le travail qu'ils effectuaient sur les blocs. En effet, dans les deux méthodes précédentes, le contenu d'un bloc du fichier n'est pas modifié tant que le processus le possède dans son espace virtuel, alors qu'ici, l'algorithme de remplacement de page peut conduire à réécrire une des pages du bloc à tout moment dans le fichier. Lorsqu'un bloc occupe plusieurs pages, il peut même se faire que les modifications de certaines pages du bloc aient été reportées dans les secteurs correspondants, alors que d'autres ne l'ont pas été. En cas de panne, cela veut dire que le contenu du bloc sur disque pourrait être incohérent. Les deux méthodes précédentes évitaient cet inconvénient, puisque le bloc peut être transféré en une seule opération entre l'espace fichier et l'espace des tampons (mémoire centrale pour la méthode G.1 et espace de pagination pour G.2). Ce transfert étant sous la responsabilité du noyau, celui-ci peut appliquer des contrôles, permettant de garantir que le bloc est, dans tous les cas, écrit dans son intégralité.

3.9.8. Question H

Ce système présente dans son noyau des aspects figés. On constate que l'allocation d'espace est par zone. Il en découle, par conséquence, les avantages et inconvénients liés à cette méthode, et déjà énoncés dans le paragraphe 8.2. Comme un fichier peut avoir 200 zones de même taille, on voit que pour les fichiers de taille inférieurs à 200Ko, il suffit de donner la valeur 1 aux paramètres `tbp` et `tz` de ces fichiers pour que l'allocation soit toujours possible pour eux, sauf si tout l'espace est occupé. Sur un site donné, il pourrait être recommandé de faire en sorte que le produit `tbp*tz` soit une valeur fixe, sauf exception. Il s'ensuivrait des allocations en général par zones toujours de même taille, et on se rapprocherait de l'allocation par bloc de taille fixe, tout en ayant exceptionnellement pour les gros fichiers des allocations de taille importante. Mais ces gros fichiers existent souvent sur de longues périodes, sans nécessiter d'allocation nouvelle.

Par ailleurs, la décomposition du système de gestion de fichier en noyau et ensemble de modules présente des avantages du point de vue de la souplesse, comme l'ont montré les questions B et C. Le fabricant peut proposer plusieurs organisations indépendantes, mais aussi permettre à l'utilisateur de construire sa propre organisation de données. Il offre une certaine ouverture: le concepteur d'un logiciel, comme par exemple un système de gestion de base de données, peut adapter une organisation qui lui parait la plus efficace. UNIX offre cette ouverture en présentant les fichiers comme des suites linéaires d'octets adressables, et en reportant au niveau des programmes utilisateurs le soin de construire des structures plus complexes. C'est ce que fait le noyau ici. Cependant, la partie du descripteur qui est à disposition des modules tente d'allier la souplesse et la fiabilité: si un minimum de protection est assurée, qui contrôle que le module est bien autorisé à accéder au contenu du fichier, la correction de l'organisation ne repose plus sur les programmes de l'utilisateur, mais sur le module chargé de ce travail.

3.10. Macintosh: représentation des répertoires

Le but de cet exercice est l'étude d'un système de gestion de fichiers d'un ordinateur individuel. L'évaluation porte sur la compréhension de ce système, qui elle-même résulte de la connaissance du cours.

Chaque volume est structuré en une arborescence de fichiers et de répertoires. Lorsqu'un répertoire ou un fichier est créé, le système lui attribue un numéro d'identification, appelé `Num_ID`, entier sur 32 bits. Ces numéros sont attribués séquentiellement[3] pour un volume, en commençant à 2 pour le répertoire racine du volume. Pour chaque nœud de l'arborescence on définit la clé du nœud obtenue par concaténation d'un numéro et d'une chaîne de caractères. Pour la racine, la clé est constituée du numéro 1 et du nom symbolique du volume (`<1>"Touamotou"` dans l'exemple ci-dessous). Pour les autres nœuds, le numéro est le `Num_ID` du répertoire père du nœud, et la chaîne de caractères est le nom symbolique du nœud dans ce répertoire (`<4>"Dessinateur"` dans l'exemple ci-dessous). Enfin, chaque nœud répertoire contient un nœud particulier que nous appellerons un "lien", dont la

[3] On suppose que 32 bits sont suffisants pour que durant la vie d'un volume il n'y ait pas débordement de ce numéro.

clé est constituée du `Num_ID` du répertoire lui-même et d'une chaîne vide. Notons que c'est la plus petite clé des nœuds fils du répertoire.

Le système maintient un "catalogue" unique par volume, qui est une table associant à chaque clé d'un nœud le descripteur de ce nœud. Ce catalogue a une organisation séquentielle indexée. Les descripteurs peuvent être de 3 types :

- `'r'` Le nœud est un nœud répertoire. Le descripteur contient, en particulier, le `Num_ID` du répertoire.
- `'f'` Le nœud est un nœud fichier. Le descripteur contient, en particulier, le `Num_ID` du fichier.
- `'l'` Le nœud est un nœud lien. Le descripteur contient la clé du répertoire.

Voici un exemple de la partie séquentielle du catalogue du volume `"Touamotou"`. Les points de suspensions indiquent les parties manquantes des descripteurs.

```
    clé            descripteur
<1>"Touamotou"     'r'<2>...
<2>""              'l'<1>"Touamotou"
<2>"Dessins"       'r'<9>...
<2>"Essai"         'f'<3>...
<2>"Logiciels"     'r'<4>...
<2>"Textes"        'r'<8>...
<4>""              'l'<2>"Logiciels"
<4>"Caml"          'f'<5>...
<4>"Dessinateur"   'f'<6>...
<4>"Rédacteur"     'f'<7>...
<8>""              'l'<2>"Textes"
<9>""              'l'<2>"Dessins"
<EOF>
```

A- Dessiner l'arborescence de l'exemple.

B- Que représente pour vous l'opération de montage de volume.

C- Le système fournit, en particulier, les opérations suivantes[4] pour accéder au catalogue (analogues à celles d'un fichier séquentiel indexé) :

- `recherche (clé, enrgt)` localise l'enregistrement ayant la clé donnée et fournit sa valeur.
- `lire (enrgt)` fournit la valeur de l'enregistrement suivant. S'il n'y a plus d'enregistrements, l'opération met à `vrai` la variable globale `fin_fichier`; sinon elle est mise à `faux`.
- `réécrire (enrgt)` modifie l'enregistrement courant, sans modifier sa clé.
- `écrire (enrgt)` ajoute un nouvel enregistrement, en tenant compte de sa clé,
- `supprime (clé)` retire de la structure l'enregistrement ayant la clé donnée.

C.1- Le chemin d'accès à un objet est défini par la concaténation du nom du volume, et des noms des différents répertoires à parcourir, le caractère séparateur entre les noms étant le ':'. Expliquer, en utilisant les opérations ci-dessus, comment on obtient le descripteur d'un fichier donné par son chemin d'accès (On s'intéressera peu à la syntaxe, mais prendra garde à la sémantique).

C.2- Expliquer, en utilisant les opérations ci-dessus, comment on obtient la liste des entrées d'un répertoire donné par son chemin d'accès.

C.3- Expliquer, en utilisant les opérations ci-dessus, comment on peut retrouver le chemin d'accès d'un répertoire d'un volume connaissant son `Num_ID`.

C.4- Expliquer, en utilisant les opérations ci-dessus, comment on peut renommer un objet dans un répertoire. On prendra garde au contenu du nœud lien lorsque l'objet est un répertoire.

C.5- Expliquer, en utilisant les opérations de la méthode d'accès dynamique, comment on peut déplacer une sous arborescence depuis un endroit vers un autre sur le même volume.

D- Quels avantages ou inconvénients voyez-vous à cette structuration en un fichier séquentiel indexé unique, par rapport à des organisations séparées des répertoires du volume.

[4] Vous n'avez pas à les décrire, mais vous pouvez les utiliser.

E- Les descripteurs de fichiers et de répertoires contiennent les dates suivantes :

la date de création de l'objet,

la date de dernière modification,

la date de dernière sauvegarde.

Ces dates sont représentées par un entier sur 32 bits, qui est le nombre de secondes écoulées depuis le 1er Janvier 1904. La date de création est mise par le système lors de la création de l'objet, et ne peut être changée. La date de dernière modification est initialisée par le système à la valeur de la date de création. Elle est mise à jour automatiquement par le système, s'il s'agit d'un répertoire lors d'un renommage, d'une adjonction ou d'une suppression d'une de ses entrées et s'il s'agit d'un fichier lors de sa fermeture s'il a été ouvert en écriture. La date de dernière sauvegarde est initialement nulle et sa mise à jour peut être demandée par une fonction spécifique du système.

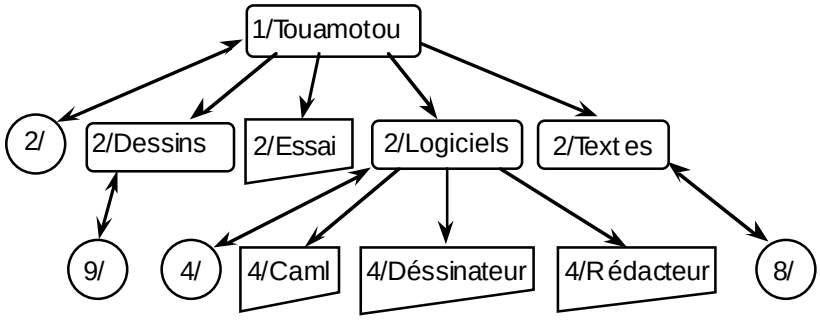
E.1- Rappeler les différents types de sauvegarde et proposer les grandes lignes d'un outil de sauvegarde incrémentale qui ne tient compte que des dates relatives aux fichiers.

E.2- Quelle utilité pourrait-on faire, dans cet outil, des dates relatives aux répertoires.

Solution de l'exercice 3.10.

3.10.1. Question A.

L'arborescence de l'exemple peut se dessiner de la façon suivante. Les rectangles chanfreinés représentent les nœuds répertoires, les ronds représentent les nœuds liens et les trapèzes représentent les nœuds fichiers. A l'intérieur, on a mis les clés des nœuds.



3.10.2. Question B.

L'opération de montage de volume est l'opération qui permet de rendre accessible le contenu d'un volume. Elle va surtout consister en une mise à jour des tables du système pour permettre ces accès. Dans ce cas ci, il s'agit d'associer le nom symbolique du volume au périphérique qui le contient. Il s'agit également de prendre en compte les informations de localisation du fichier séquentiel indexé qui contient le catalogue des objets du volume, et permettre ainsi l'exploration des répertoires. Il est probable que d'autres tables devront être initialisées, par exemple pour permettre l'allocation d'espace sur ce volume. Manifestement ces autres tables sont en dehors de l'exercice.

Lorsque le volume est monté, la désignation des objets situés sur ce volume peut se faire en utilisant le nom symbolique du volume qui identifie, d'une part le périphérique sur lequel se trouve l'arborescence, d'autre part le catalogue du volume qui représente cette arborescence.

3.10.3. Question C.

3.10.3.1. Question C.1.

Globalement, il faut découper le chemin d'accès à l'objet en une suite de nom, chaque nom étant obtenu en recherchant le caractère ':' suivant. Informellement, on suppose que l'on dispose d'une fonction `Nom_Suivant` qui retourne la chaîne de caractères du chemin d'accès qui commence à la position courante et s'arrête avant le prochain caractère ':', ou à la fin de la chaîne. Une telle fonction pourrait être décrite de la façon suivante.

```
Chemin_D_Acces : Chaine; I_Cour : Entier := 0;
fonction Nom_Suivant retourne Chaine est
     Nom_Local : Chaine; I_Local : Entier := 0;
début tantque Chemin_D_Acces(I_Cour) /= Fin_Chaine
       et Chemin_D_Acces(I_Cour) /= ':' faire
          Nom_Local(I_Local) := Chemin_D_Acces(I_Cour);
          I_Local := I_Local + 1; I_Cour := I_Cour + 1;
       fait;
       Nom_Local(I_Local) := Fin_Chaine;
       si Chemin_D_Acces(I_Cour) = ':' alors I_Cour := I_Cour + 1; finsi;
       retourner Nom_Local;
fin Nom_Suivant;
```

Le premier appel de cette fonction retourne le nom symbolique du volume, que le système recherche dans ses tables, et qui permet de localiser l'organisation séquentielle indexée sur laquelle la méthode d'accès sera appliquée.

```
procédure Localiser_Descripteur est
début Cle.Num_ID := 1; Cle.Nom := Nom_Suivant;
       Localiser_Volume (Cle.Nom);
       Recherche (Cle, Enrgt);
       tantque Chemin_D_Acces(I_Cour) /= Fin_Chaine faire
          Cle.Num_ID := Enrgt.Num_ID; Cle.Nom := Nom_Suivant;
          Recherche (Cle, Enrgt);
       fait;
fin Localiser_Descripteur;
```

Dans ce morceau de programme, on utilise deux fois le nom symbolique du volume, une première fois pour localiser le volume lui-même dans les tables du systèmes, et une deuxième fois pour localiser le répertoire racine du volume. Cette deuxième utilisation n'est pas obligatoire, puisque l'on sait que le `Num_ID` du répertoire racine est obligatoirement 2. Cependant cela garantit qu'à la fin, la variable `Enrgt` contient le descripteur d'un nœud quelconque de l'arborescence, qui peut être la racine si le chemin d'accès est simplement constitué du nom du volume.

Notons que l'on pourrait vérifier à chaque étape que le descripteur est bien un descripteur de répertoire. Cependant s'il s'agit d'un fichier, la recherche échouera de toute façon. S'il s'agit d'un lien, le `Num_ID` sera celui du nœud père. Cela veut dire qu'un nom vide dans le chemin d'accès permet de remonter sur le nœud père. Évidemment, sur un nom absolu, partant de la racine, cela n'a pas de sens. Par contre on voit que l'on peut mettre en place simplement des chemins d'accès relatifs à partir d'un répertoire particulier (répertoire de travail par exemple).

3.10.3.2. Question C.2.

L'algorithme de la question précédente fournit dans `Enrgt` le descripteur atteint à la fin de l'exploration du chemin d'accès. S'il s'agit d'un répertoire, le champ `Num_ID` contient le `Num_ID` du répertoire. Le catalogue est un fichier séquentiel indexé. Une telle structure est constituée de deux parties, dont l'une contient les enregistrements en ordre croissant des clés. Il est donc facile de rechercher le nœud lien du répertoire, puis de parcourir séquentiellement les enregistrements jusqu'à en trouver un dont la clé ne contient plus le `Num_ID` du répertoire.

```
procédure Lister_Répertoire est
     Num_Loc : Entier;
début Localiser_Descripteur;
       Num_Loc := Enrgt.Num_ID;
       Cle.Num_ID := Num_Loc; Cle.Nom := ""; Recherche (Cle, Enrgt);
       Lire (Enrgt);
       tantque non Fin_Fichier
       et alors Enrgt.Cle.Num_ID = Num_Loc faire
          Imprimer (Enrgt.Cle.Nom);
          Lire (Enrgt);
       fait;
fin Lister_Répertoire;
```

3.10.3.3. Question C.3.

On a déjà vu dans la question C.1 que le lien permettait de remonter sur le père. En remontant les liens successifs, on pourra donc remonter jusqu'à la racine. Par ailleurs, chaque fois que l'on trouve

un lien, on trouve aussi le nom du répertoire auquel il correspond. Il faut donc concaténer ces noms en ordre inverse où on les trouve (la concaténation est représentée ici par l'opérateur &).

```
fonction Nom_Répertoire (Num_ID : Entier) retourne Chaine est
     Nom_Local : Chaine := "";
début Cle.Num_ID := Num_ID; Cle.Nom := ""; recherche (Cle, Enrgt);
       Nom_Local := Enrgt.Nom_Lien;
       tantque Enrgt.Num_ID /= 1 faire
          Cle.Num_ID := Enrgt.Num_ID; recherche (Cle, Enrgt);
          Nom_Local := Enrgt.Nom_Lien & ':' & Nom_Local;
       fait;
fin Nom_Répertoire;
```

3.10.3.4. Question C.4.

Renommer un objet dans un répertoire signifie changer sa clé dans le catalogue. Sa place dans la suite des enregistrements ordonnés du catalogue risque de changer. Il faut donc rajouter l'enregistrement avec la clé déduite de son nouveau nom et supprimer l'enregistrement précédent. Il est préférable de le faire dans cet ordre, car si une panne survient entre deux, le descripteur ne sera pas perdu mais en deux exemplaires. Évidemment, à la reprise, il faudra faire le ménage! Par ailleurs, si on renomme un répertoire, il faut aussi changer le nom mémorisé dans son lien.

```
procédure Renommer (Nouveau : Chaine) est
début Localiser_Descripteur;
       si Enrgt.Type = 'l' alors Erreur; finsi;
       Enrgt.Cle.Nom := Nouveau;
       Ecrire (Enrgt);
       Supprimer (Cle);
       si Enrgt.Type = 'r' alors
          Cle.Num_ID := Enrgt.Num_ID; Cle.Nom := ""; Recherche (Cle, Enrgt);
          Enrgt.Nom_Lien := Nouveau;
          Réécrire (Enrgt);
       finsi;
fin Renommer;
```

3.10.3.5. Question C.5.

Déplacer une sous arborescence depuis un endroit vers un autre d'un même volume, cela veut dire trouver l'enregistrement correspondant au nœud racine de cette sous arborescence pour en changer la clé, puisque c'est cette clé qui relie le `Num_ID` du répertoire père à ce nœud racine. Cette clé doit être changée en remplaçant le `Num_ID` par celui du nœud répertoire destinataire. Comme on l'a fait dans la question précédente, si le nœud racine est un répertoire (ce pourrait être simplement un fichier), il faut modifier le contenu de son nœud lien. Lors de ce déplacement, une anomalie pourrait se produire qu'il faut empêcher : il ne faut pas que le déplacement puisse raccrocher la racine de la sous arborescence à un nœud qui soit lui-même contenu dans cette arborescence. Pour éviter cela, il suffit de refuser le déplacement si la recherche du répertoire destinataire conduit à passer par le nœud racine de la sous arborescence à déplacer. Une autre vérification à faire est de vérifier que les chemins d'accès sont sur le même volume.

```
procédure Déplacer (Racine : Chaîne; Destinataire : Chaîne) est
     Cle_Locale : Type_Cle;
     Num_ID_Racine : Entier;      -- Num_ID racine déplacée
     Num_ID_Desti : Entier;       -- Num_ID destinataire
début Chemin_D_Acces := Destinataire;
       I_Cour := 0;
       Nom_Volume := Nom_Suivant; -- volume destinataire
       Chemin_D_Acces := Racine;
       I_Cour := 0;
       si Nom_Volume /= Nom_Suivant alors Erreur; finsi;
       I_Cour := 0;          -- même volume
       Localiser_Descripteur;          -- le nœud racine
       si Enrgt.Type = 'l' alors Erreur; finsi;
       Cle_Locale := Enrgt.Cle;   -- son ancienne clé
       Num_ID_Racine:= Enrgt.Num_ID;   -- Num_ID du nœud racine
       Chemin_D_Acces := Destinataire;
       I_Cour := 0;
       Cle.Num_ID := 1; Cle.Nom := Nom_Suivant;
       Localiser_Volume (Cle.Nom);
```

```
        Recherche (Cle, Enrgt);
        tantque Chemin_D_Acces(I_Cour) /= Fin_Chaine faire
           si Enrgt.Num_ID => Num_ID_Racine alors Erreur; finsi;
           Cle.Num_ID:= Enrgt.Num_ID;
           Cle.Nom := Nom_Suivant;
           Recherche (Cle, Enrgt);
        fait;
        si Enrgt.Num_ID = Num_ID_Racine ou Enrgt.Type /= 'r' alors
           Erreur;
        finsi;
        Num_ID_Desti:= Enrgt.Num_ID; -- c'est bien un répertoire
        Recherche (Cle_Locale, Enrgt);  -- reprise racine
        Enrgt.Cle.Num_ID := Num_ID_Desti;    -- changer son père
        Ecrire (Enrgt);            -- création nouveau
        Supprimer (Cle);      -- suppression ancien
        si Enrgt.Type = 'r' alors
           Cle.Num_ID:= Enrgt.Num_ID; Cle.Nom := "";
           Recherche (Cle, Enrgt);
           Enrgt.Num_ID := Num_ID_Desti;     -- contenu du lien
           Réécrire (Enrgt);
        finsi;
fin Déplacer;
```

3.10.4. Question D.

La première remarque est que toutes les opérations habituelles que l'on peut espérer faire sur une arborescence sont bien possibles ici, comme le montrent les questions C.1 à C.5. Le premier avantage que l'on peut voir est le fait d'avoir une seule structure pour l'ensemble de tous les répertoires. Cette structure occupe un espace qui est directement proportionnelle au nombre de nœuds de l'arborescence du volume, quelle que soit la façon dont les nœuds sont situés sur cette arborescence. Ceci n'est pas le cas d'une organisation séparée des répertoires, pour laquelle chaque répertoire occupera un espace minimum sur le disque, même s'il est vide.

On peut penser que l'organisation séparée des répertoires permet des opérations de recherche d'un objet plus rapides que dans l'organisation proposée ici. En effet lors de l'exploration du chemin d'accès, chaque étape consiste en une recherche séquentielle dans les entrées du répertoire, recherche qui fournit le descripteur du répertoire où la recherche est à poursuivre. On peut en déduire, a priori, qu'une étape demandera un ou deux accès disques. Dans le cas du fichier séquentiel indexé unique, chaque étape est une recherche dans les tables d’index, qui peuvent être à plusieurs niveaux, suivant le nombre de nœuds de l'arborescence, et donc demander plusieurs accès disques. A noter cependant que les opérations peuvent conserver dans des tampons en mémoire les blocs disques les plus accédés, limitant ainsi cet inconvénient.

L'organisation proposée ici permet également de parcourir systématiquement l'ensemble des nœuds de l'arborescence, lorsque la structure elle-même n'intervient pas dans le traitement. Il en est ainsi, par exemple, dans les opérations de sauvegarde de la question suivante.

3.10.5. Question E.

3.10.5.1. Question E.1.

On distingue trois types de sauvegardes. La sauvegarde complète consiste en la réalisation d'une copie de l'ensemble des volumes de l'installation. La sauvegarde de reprise consiste à faire la copie d'un volume particulier indépendamment des autres; les sauvegardes de reprise sont donc des vues partielles à différents moments. La sauvegarde incrémentale consiste à ne faire la copie que de ce qui a été modifié depuis la dernière sauvegarde incrémentale ou depuis la dernière sauvegarde de reprise.

Dans notre système, une idée simple pour réaliser une sauvegarde incrémentale est de parcourir séquentiellement les enregistrements du catalogue, et pour chaque enregistrement contenant un descripteur de type fichier dont la date de dernière modification est supérieure à la date de dernière sauvegarde, de faire une sauvegarde de l'objet correspondant. Une fois cette sauvegarde effectuée, il faut mettre à jour la date de dernière sauvegarde. Plus généralement, cette mise à jour de la date de

dernière sauvegarde doit être effectuée pour chaque type de sauvegarde (complète, reprise ou incrémentale).

La sauvegarde individuelle d'un fichier consiste en une copie de son contenu et des informations qui le décrit (date, longueurs, organisation, etc...). Cela implique que dans les sauvegardes successives, on a des copies de différentes versions du même fichier. Une restauration doit permettre de restaurer la dernière version. Il est donc important de déterminer les copies qui correspondent à différentes versions d'un même fichier. Il est clair dans les explications fournies, que l'identification absolue d'un fichier n'est pas son chemin d'accès, car il peut être renommé ou déplacé, mais son `Num_ID` qui ne change pas pendant toute l'existence du fichier sur le volume. On peut donc avoir deux approches : la première consiste à mémoriser chemin d'accès au fichier et `Num_ID` lors de chaque sauvegarde, et la seconde consiste à mémoriser ce chemin d'accès uniquement lors de la première sauvegarde, les sauvegardes suivantes mentionnant seulement le `Num_ID` de l'objet. Dans ce dernier cas, se pose le problème de la mémorisation d'un déplacement ou d'un renommage du fichier, ce qui sera abordé dans la question suivante.

```
procédure Sauvegarde_Incrémentale (Nom_Volume : Chaîne) est
début CLE.Num_ID := 1; Cle.Nom := Nom_Volume;
       Localiser_Volume (Cle.Nom);
       Recherche (Cle, Enrgt);         -- premier enregistrement
       Lire (Enrgt);
       tantque non Fin_Fichier faire
          si Enrgt.Type = 'f' et Enrgt.Date_Modif > Enrgt.Date_Sauv alors
               Sauvegarder (Enrgt);
               Mise_A_Jour_Date_Sauvegarde (Enrgt);
          finsi;
          Lire (Enrgt);
       fait;
fin Sauvegarde_Incrémentale;
```

3.10.5.2. Question E.2.

La date de dernière modification d'un répertoire permet de savoir quand a eu lieu le dernier renommage ou la dernière adjonction ou suppression d'une entrée dans ce répertoire. Ces modifications correspondent à des modifications structurelles de l'arborescence. Au moment de la sauvegarde incrémentale, on ne peut savoir quelles sont ces modifications, mais le résultat. C'est donc ce résultat qu'il faut sauvegarder. Lorsque la date de dernière modification d'un répertoire est postérieure à la date de dernière sauvegarde de ce répertoire, il faut donc sauvegarder la liste des entrées actuelles de ce répertoire suivant un algorithme analogue à celui de la question C.2. Pour chacune de ces entrées, il faut non seulement sauvegarder le nom mais aussi le `Num_ID` de l'entrée. Par contre, il n'est pas nécessaire de sauvegarder à ce moment le contenu de ces entrées, qu'elles correspondent à un fichier ou à un répertoire. Leur sauvegarde éventuelle interviendra lorsqu'on les rencontrera lors du parcours séquentiel du catalogue.

3.11. Macintosh: gestion de l’espace disque

Nous nous intéressons à une gestion particulière de l'espace attribué aux fichiers. L'espace disponible pour les données des fichiers est découpé en blocs (au plus 32768 blocs) de taille fixe, multiple de 512 (taille d'un secteur). Les blocs de cet espace sont numérotés à partir de 0. L'espace libre est représenté par un tableau de bits. Le descripteur de volume contient les informations suivantes :

taille d'un bloc en octets (32 bits),

nombre total de blocs du volume (16 bits),

nombre de blocs libres de l'espace libre (16 bits),

numéro du premier bloc libre (16 bits),

L'espace alloué à un fichier est représenté par un certain nombre de zones, chaque zone étant définie par le numéro de son premier bloc et le nombre de blocs qu'elle contient. Les seize premières zones d'un fichier sont mémorisées dans le descripteur de fichier qui contiennent les informations suivantes :

la longueur physique du fichier en octets,

la longueur logique du fichier en octets,

16 descripteurs de zones (numéro de premier bloc, nombre de blocs).

Lorsque l'on tente d'écrire au delà de la fin physique d'un fichier, le système alloue un nouveau bloc. La recherche de ce bloc libre se fait de préférence au bout de la dernière zone déjà allouée au fichier. Si ce n'est pas possible, une nouvelle zone est ajoutée au fichier (le système peut gérer un nombre quelconque de zones, mais nous supposerons ici qu'il n'y en a jamais plus de 16).

A- Comment peut-on déterminer la taille en nombre d'octets du volume, ainsi que celle de l'espace libre? En supposant qu'un volume contient 1 Go, donner la taille minimale d'un bloc.

B- Quelle différence faites vous entre la longueur physique d'un fichier et sa longueur logique?

C- Expliquer les opérations effectuées pour trouver le secteur qui contient un octet quelconque, dont on connaît le numéro dans l'organisation non structurée.

D- Montrer que, s'il reste de l'espace libre, l'allocation est toujours possible. Evaluer le temps d'une allocation.

E- Cette implantation, faite sur un ordinateur individuel, permet en général d'avoir des zones de grandes tailles. Par contre, si elle est faite sur un ordinateur utilisé simultanément par beaucoup d'utilisateurs, elle conduira à des zones de petites tailles. Expliquer pourquoi.

F- Décrire ce qu'il faut faire lors de la destruction d'un fichier, quant à l'espace qu'il occupe.

G- Quels sont les avantages et les inconvénients de cette implantation.

Solution de l’exercice 3.11

3.11.1. Question A

La taille en nombre d'octets du volume est obtenue en faisant le produit entre les deux valeurs "taille d'un bloc en octets" et "nombre total de blocs du volume" qui sont situés dans le descripteur de volume. Pour connaître la taille de l'espace libre, il suffit de faire le produit entre les deux valeurs "taille d'un bloc en octets" et "nombre de blocs libres de l'espace libre" qui sont situés dans le descripteur de volume.

Si on suppose que le volume contient 1 Go, le nombre total de blocs étant limité à 32768, la taille d'un bloc est donc au moins de 1Go/32768, soit 32768. Notons que un bloc représente 64 secteurs.

3.11.2. Question B

La longueur physique d'un fichier représente la quantité d'espace disque qui lui est alloué. C'est donc le nombre d'octets qu'il occupe réellement sur le disque. La longueur logique d'un fichier représente le nombre d'octets qui contiennent des informations appartenant au fichier. C'est donc le nombre d'octets utiles du fichier. La longueur logique est toujours inférieure ou égale à la longueur physique, et la différence est un espace inutilisé du disque. Ces deux quantités ne sont pas égales, en général. La première est liée à la quantité d'informations à mettre dans le fichier. La seconde tient compte de la méthode de gestion de l'espace disque.

3.11.3. Question C

Notons Z_i la i-ème zone d'un fichier, T_i sa taille en nombre de blocs, et D_i le numéro de son premier bloc. Notons Tb la taille d'un bloc du volume. Rechercher le secteur qui contient l'octet de numéro N du fichier, en supposant que N est inférieur à la longueur logique du fichier, et donc à sa longueur physique, consiste d'abord à rechercher la zone Z_i qui contient cet octet, ainsi que la position relative Pr de l'octet dans la zone. Le secteur est alors celui de numéro $(Pr + D_i * Tb)/512$, la place de l'octet dans le secteur correspondant au reste de cette division.

Pour déterminer la zone, il faut déterminer i tel que :

$$\sum_{p=1}^{p=i-1} T_p \leq N < \sum_{p=1}^{p=i} T_p \text{ , on a alors } Pr = N - \sum_{p=1}^{p=i-1} T_p$$

On peut l'obtenir par le petit programme suivant:

```
Pr := N;                /* position dans le fichier */
I := 0;                     /* première zone */
tantque Pr >= T[I] faire
     Pr := Pr - T[I];   /* position relative dans la suite */
     I := I + 1;
fait;
Ns := (Pr + D[I] * Tb)/512;
```

3.11.4. Question D

L'allocation d'espace se fait lorsque l'on tente d'écrire au delà de la fin physique du fichier, et dans ce cas on alloue un bloc unique. Donc s'il reste de l'espace libre, l'allocation sera possible, puisque n'importe quel bloc peut convenir. Si le système cherche de préférence à allouer un bloc au bout de la dernière zone, il alloue un autre bloc quelconque lorsque ce n'est pas possible. La seule contrainte pourrait être la limitation du nombre de zones d'un fichier. Or il est dit que le système gère un nombre quelconque de zones.

On peut penser que la table de bits représentant l'espace libre est toujours présente en mémoire, et n'est réécrite que de temps en temps lorsqu'elle a été modifiée. Ceci peut poser des problèmes de sécurité, mais nous n'en parlerons pas. Cette table contient au plus 32768 bits, donc 4096 octets, ou 8 secteurs. Ce n'est pas trop encombrant en mémoire centrale.

Lors de l'allocation, il faut d'abord tester si le bloc qui suit la dernière zone est libre. Notons qu'il s'agit du bloc de numéro $D_i + T_i$, si i est le numéro de cette zone. Si ce bloc est libre, il suffit de changer son état à occupé et d'allonger la zone Z_i en incrémentant T_i. Si ce bloc n'est pas libre, on crée une nouvelle zone, de taille 1, qui commence au numéro du premier bloc libre trouvé dans le descripteur du volume. Enfin, il faut mettre à jour la longueur physique du fichier et le nombre de blocs libres de l'espace libre. Toutes ces actions demandent un temps fixe que l'on peut estimer à quelques microsecondes sur les ordinateurs actuels.

Il faut également mettre à jour le numéro du premier bloc libre du descripteur de volume, si c'est ce bloc qui a été alloué. Cela implique de rechercher à partir de ce numéro de bloc (il est inutile de rechercher avant, car il n'y en a pas) le prochain bloc libre. Il est possible que ce soit le suivant. Il est possible que l'on ait besoin d'en parcourir beaucoup avant d'en trouver un de libre. Évidemment, il y en a au pire 32768, si le dernier bloc libre était le premier. Intuitivement, si l'on n'est pas trop près de la saturation, il y a des blocs libres un peu partout dans l'espace, et on ne mettra pas trop de temps pour le trouver. Supposons qu'il soit nécessaire de parcourir 1000 blocs occupés avant d'en trouver un de libre. S'il faut 1 microseconde par bit, cela donne 1 milliseconde. Cependant ceci peut être amélioré. En effet si on représente l'état occupé par un bit à 0, la recherche d'un bloc libre est la recherche d'un bit à 1. Mais un tel bit est alors situé dans un octet qui n'est pas nul, ou encore dans un mot machine qui n'est pas nul. Sur une machine à 32 bits, cela signifie rechercher un mot non nul parmi 32. Une fois ce mot trouvé, on recherche le bit à 1 parmi les 32 du mot. Tout ceci doit pouvoir être fait en au plus 64 microsecondes.

3.11.5. Question E

Sur un ordinateur individuel, il n'y a qu'un seul utilisateur à la fois. Il s'en suit qu'il n'y a en général qu'un seul fichier en cours de création. Bien que l'allocation se fasse bloc par bloc, l'espace alloué au fichier viendra des premières portions d'espace contigu, qui sont libres avant cette création.

Si l'ordinateur est utilisé simultanément par beaucoup d'utilisateurs, cela implique qu'il peut y avoir plusieurs fichiers en cours de création. Le système sera alors amené à allouer de l'espace, à raison de un bloc à la fois, tantôt pour l'un tantôt pour l'autre de ces fichiers. Les blocs ainsi alloués ne seront alors jamais au bout de la dernière zone déjà allouée au fichier, et conduiront à l'ajout d'une nouvelle zone. Notons que ceci pourrait également se faire sur un ordinateur individuel si un même programme créait deux fichiers simultanément, en écrivant tantôt dans l'un tantôt dans l'autre. Un tel programme est cependant assez rare.

3.11.6. Question F

Lors de la destruction d'un fichier, en dehors des actions à faire sur les répertoires, il faut restituer à l'espace libre l'espace qu'il occupait. Ceci se traduit simplement par la remise à l'état libre de chacun des blocs contenus dans chacune des zones du fichier. En même temps, il faut éventuellement remettre à jour le numéro de premier bloc libre du descripteur de volume. Le morceau de programme suivant donne les grandes lignes de ce qu'il faut faire.

```
J := Longueur_Physique / Tb;  /* nombre de blocs du fichier */
I := 0;                                 /* première zone du fichier */
tantque J > 0 faire
     pour K dans D[I] .. D[I] + T[I] - 1 faire
          Etat[K] := Libre;  /* libérer les blocs de la zone */
     fait;
     si D[I] < Premier_Bloc_Libre alors Premier_Bloc_Libre := D[I]; finsi;
     J := J - T[I];
     I := I + 1;
fait;
```

3.11.7. Question G

Les avantages comme les inconvénients ont déjà été évoqués, en grande partie du moins, dans les questions précédentes. On note ainsi que, contrairement aux différentes techniques d'allocation par zone, l'allocation est toujours possible, pourvu qu'il reste de la place, évidemment. En ce sens, le système se rapproche de la technique utilisée dans VMS, lorsque l'utilisateur demande une allocation au mieux. De plus, si le disque comporte de grandes zones libres, le système allouera un nombre réduit de zones au fichier, améliorant les performances par rapport à une allocation par blocs de taille fixe. On a donc un bon compromis entre les deux techniques d'allocation.

L'inconvénient majeur est lié au fait que les zones d'un fichier sont de tailles variables. Si on ne fait que des accès séquentiels sur de tels fichiers, ce n'est pas gênant, puisqu'on va parcourir les zones les unes après les autres en tenant compte de leurs tailles successives. Si on fait de l'accès aléatoire, nous avons vu dans la question C qu'il faudra parcourir la table des descripteurs de zones du fichier, pour déterminer celle qui contient l'information recherchée. Si le fichier contient peu de zones, ce parcours sera très rapide. S'il contient beaucoup de zones, cela peut prendre un peu de temps. Encore faut-il relativiser: s'il y a 50 zones, il faut en parcourir 25 en moyenne, ce qui peut demander 100 microsecondes à raison de 4 microsecondes par zone. Or la question E induit l'idée que même un gros fichier ne comportera qu'un nombre réduit de zones.

Remarque (hors sujet): cet exercice, comme le précédent d'ailleurs, est tiré de la gestion du MacIntosh. Dans le système réel, le descripteur de fichier ne contient que 3 descripteurs de zones, les autres descripteurs de zones éventuels sont rangés dans un B-arbre, tous fichiers confondus. L'expérience montre que très peu de fichiers ont plus de 3 zones, cependant la méthode n'impose pas de véritable limite sur le nombre de zones d'un fichier.

3.12. A propos de MacOS

Vous avez acheté un ensemble d'outils de développement pour votre micro ordinateur personnel (un Macintosh). Vous désirez installer cet ensemble pour vous permettre d'expérimenter et approfondir ce que vous avez appris au CNAM, tout en diminuant vos déplacements. Vous lisez la notice avec attention. Dans le chapitre « Installation », on trouve:

L'espace disque nécessaire à l'installation de cet ensemble d'outils de développement est de 45 Mo sur un volume de 165 Mo. Du fait des techniques d'allocation de MacOS, l'espace nécessaire peut être plus important sur un volume plus gros. Par exemple, il peut atteindre 65 Mo sur un volume de 500 Mo. Il est donc préférable de faire l'installation sur une partition d'au plus 250 Mo.

Vous avez appris que, dans le système MacOS, l'espace disque disponible pour les fichiers est découpé en blocs (au plus 65536 blocs) de taille fixe, multiple de 512 (taille d'un secteur), et que l'espace attribué à un fichier est constitué d'un nombre quelconque de zones, chaque zone contenant un nombre entier de blocs contiguës.

Quelle justification voyez vous au paragraphe de la notice (le texte italique ci-dessus)?

Solution de l'exercice 3.12

Comme le système ne peut gérer plus de 65536 blocs, les gros volumes nécessitent une taille de blocs importante, qui est au moins égale à la taille du volume divisée par 65536, tout en étant un multiple de 512 octets. Avec un volume de 165 Mo, un bloc doit faire au moins 2640 octets pour la première contrainte, qui doit être portée à 3072 pour être un multiple de 512. Avec un volume de 500 Mo, la taille du bloc doit être au moins de 8000 octets soit en fait 8192 octets. Or, chaque fichier se voit attribuer un nombre entier de blocs, et le dernier bloc est plus ou moins rempli, conduisant à une perte moyenne de un demi bloc par fichier. L'ensemble d'outils de développement dont il est question doit comporter un grand nombre de petits fichiers qui voient leur allocation augmenter sans vraiment d'utilité. On peut évaluer cela en remarquant que la notice semble indiquer une perte supplémentaire de 20 Mo lorsque la taille des blocs passe de 3072 à 8192. Si cette perte résulte uniquement de la perte supplémentaire de 5120 octets pour les petits fichiers qui se satisfaisaient d'un bloc de 3 Ko, on peut en déduire qu'il y en a 4096 fichiers. Mais, ce ne sont pas les seuls, puisque, par exemple, un fichier, dont la taille est comprise entre 8 et 9 Ko, reçoit 3 blocs de 3 Ko, ou 2 blocs de 8 Ko, donnant une perte supplémentaire de 7 Ko dans ce cas. Remarquons que la recommandation de faire l'installation dans une partition de 256 Mo permet de limiter les blocs à 4096 octets.

3.13. De HFS à HFS Plus

On s'intéresse ici à la gestion de fichier de MacOS, et à son évolution récente en 8.1. Voici un extrait d'un article de la presse de vulgarisation.

Depuis 1986, la structure des disques durs d'Apple s'appuie sur HFS (Hierarchical File System). Sans entrer dans une description technique, sachez que ce format découpe un disque en 2^{16} blocs indivisibles de taille égale, dits blocs d'allocation. Tout fichier qui occupe même partiellement un bloc d'allocation se l'attribue entièrement, ce qui signifie que la taille d'un bloc d'allocation représente la taille du plus petit fichier possible sur un volume donné.

MacOS 8.1 s'appuie sur HFS Plus, où le nombre de blocs d'allocation d'un volume varie de 2^{16} à 2^{32}. La règle du jeu est fixée comme suit:

Taille du volume	*taille des blocs*
jusqu'à 256 Mo compris	*0,5 Ko*
plus de 256 jusqu'à 512 Mo compris	*1 Ko*
plus de 512 Mo jusqu'à 1 Go compris	*2 Ko*
plus de 1 Go	*4 Ko*

Grâce à HFS Plus, l'utilisateur économise de la place sur son volume, dans une proportion très variable selon la taille du disque, la taille et le nombre de fichiers. En supplément, la barrière des 65536 fichiers sur un volume saute et varie de 65536 sur un volume de 32 Mo à plus de 1000000 sur un volume de plus de 1 Go.

A- Expliquer les conséquences et les limites induites par la contrainte du découpage de HFS en 2^{16} blocs indivisibles de taille égale de l'espace disque. On étudiera en particulier le cas d'un disque de 32 Mo et celui d'un disque de 1 Go.

B- Expliquer les avantages et les inconvénients que vous voyez à la structuration de HFS Plus. On étudiera en particulier le cas d'un disque de 32 Mo et celui d'un disque de 1 Go.

C- L'article signale que des mesures ont donné les gains en espace suivants:

Taille volume	*nombre de fichiers*	*occupation en HFS*	*occupation en HFS+*
500 Mo	*6000*	*340 Mo*	*323 Mo*
1 Go	*15000*	*983,7 Mo*	*879,8 Mo*
2 Go	*22000*	*1,3 Go*	*700 Mo*

Comment expliquez-vous de telles variations de gains suivant la taille?

D- Vous faites des mesures sur votre propre système comportant un disque de 2 Go en HFS, et vous constatez les résultats suivants:

nombre de fichiers	taille réelle	espace occupé
10000	t ≤ 4 Ko	320 Mo
5000	4 < t ≤ 8 Ko	160 Mo
3000	8 < t ≤ 32 Ko, moyenne 20 Ko	96 Mo
2000	32 Ko < t, moyenne 256 Ko	512 Mo

Quel gain espérez-vous obtenir du passage en HFS Plus de votre disque?

E- A votre avis peut-on attendre des changements (perte ou gain) en terme de temps d'accès du changement de structuration.

Solution de l'exercice 3.13

3.13.1. Question A

Nous reprenons ici ce qui a déjà été dit pour l'exercice précédent. Le limite de 2^{16} blocs indivisibles implique que, pour un disque de taille T, le bloc soit au moins de taille T/65536. Cette taille est de 512 octets pour un disque de 32 Mo et de 16 Ko pour un disque de 1 Go. Ceci a deux conséquences. D'une part, tout fichier de moins de 16 Ko recevra de fait 16 Ko sur un disque de 1 Go. Plus généralement, la perte de place étant de 1/2 quantum par objet (voir §8.4.1), plus le quantum est grand, plus la perte est grande. D'autre part, quelle que soit la taille du disque, il y a au plus 65536 blocs, donc au plus 65536 fichiers contenant au moins 1 bloc, limitant de fait le nombre de fichiers d'un volume.

3.13.2. Question B

L'utilisation de HFS Plus pallie les deux inconvénients ci-dessus pour les gros disques et est sans effet sur les disques de 32 Mo (un bloc ne peut être de taille inférieure au secteur, qui contient au moins de 512 octets sur les disques actuels). Pour un disque de 1 Go, la perte d'espace moyenne par objet est divisée par 8 (1 Ko au lieu de 8 Ko) et le nombre de fichiers maximum est multiplié par 8. Par contre, le nombre d'unités allouables augmentant, la taille de la représentation de l'espace libre augmente tout comme la durée d'exécution de l'algorithme d'allocation.

3.13.3. Question C

Analysons l'évolution de l'espace moyen occupé par fichier lors du passage de HFS en HFS Plus dans chacun des trois cas. Il passe de 58 à 55 Ko sur le disque de 500 Mo, de 67,2 à 60 Ko sur celui de 1 Go et de 60,5 à 32,6 Ko sur celui de 2 Go. Ceci veut dire que dans les deux premiers cas les fichiers sont plus gros en moyenne que dans le troisième cas, or le gain est d'autant plus important que les fichiers sont plus petits et le disque plus gros. Pour le disque de 2 Go, le bloc fait 32 Ko en HFS contre 4 Ko en HFS Plus. Comme dans le troisième cas l'espace moyen alloué est de 32,6 Ko, cela veut dire qu'il y a beaucoup de petits fichiers qui bénéficient pleinement d'un espace alloué plus faible. En fait le gain moyen est de 27,9 Ko par fichier, c'est-à-dire que presque tous les fichiers bénéficient pleinement de la réduction de la taille des blocs. Notons que les blocs ont diminués de 28 Ko, et il est étonnant d'approcher cette valeur de si prêt, mais l'article ne donnait aucune information sur la précision des mesures, ce qui nous conduit alors à être prudent dans nos conclusions.

3.13.4. Question D

Les 10000 fichiers de taille inférieure à 4 Ko recevront exactement 4 Ko en HFS Plus au lieu de 32 Ko en HFS. Le gain sera donc de 28 Ko par fichier soit un total de 280 Mo. Les 5000 fichiers dont la taille est comprise entre 4 et 8 Ko, recevront 8 Ko au lieu de 32 Ko, soit un gain de 24 Ko par fichier et donc un total de 120 Mo. Les 3000 fichiers de taille comprise entre 8 et 32 Ko ayant une taille moyenne de 20 Ko peuvent permettre un gain moyen de 12 Ko par fichier, et donc un gain total de 36 Mo. Enfin, les 2000 fichiers de taille supérieure donnaient lieu à une perte moyenne de 1/2 quantum soit 16 Ko par fichier. Avec HFS Plus, cette perte sera ramenée à 2 Ko par fichier, ce qui nous laisse espérer un gain moyen de 14 Ko par fichier, soit un total de 28 Mo. En

rassemblant l'ensemble, on peut donc espérer un gain global de 280 + 120 + 36 + 28 = 464 Mo, sur un total occupé de 1088 Mo, soit prêt de 43%.

3.13.5. Question E

Le fait d'avoir des blocs plus petits peut entraîner une certaine dispersion de l'espace alloué aux gros fichiers et donc une certaine perte de performance quant au temps d'accès, puisqu'il pourrait y avoir plus de mouvement de bras. Notons cependant que un quantum plus petit ne veut pas dire obligatoirement des zones plus petites : cela dépend de l'état de la capacité de trouver de grands espaces contigus au moment de l'allocation des zones.

3.14. Étude simplifiée de NTFS

On étudie brièvement une version simplifiée du système de gestion de fichiers NTFS. Ce système gère des partitions dont la taille peut atteindre 2^{64} octets. L'allocation se fait par blocs de taille fixe, encore appelés *clusters*, les numéros de clusters étant représentés sur 64 bits. L'espace libre est représenté par un fichier *bitmap* qui, à raison de 1 bit par cluster, définit son état d'allocation.

Une table, appelée MFT (Master File Table), contient un enregistrement par fichier existant sur le disque. Cet enregistrement, de la taille d'un cluster, contient les informations suivantes, chacune d'elles étant de taille variable :

- Nom du fichier, informations standards, informations de protection, représentant en moyenne 100 octets,
- Date et taille du fichier, occupant 16 octets
- Données. Il s'agit soit du contenu du fichier, s'il y a suffisamment de place dans l'enregistrement, ou sinon des informations de localisation du contenu sous la forme de la liste des numéros de clusters alloués au fichier.

Cette table est mémorisée dans un fichier (tout est fichier dans NTFS). Une référence à un fichier est constitué d'un couple :

- Numéro de l'enregistrement du fichier dans la MFT (48 bits),
- Numéro de séquence, incrémenté lors de chaque création d'un nouveau fichier qui réutilise ce numéro d'enregistrement (16 bits).

Dans le cas d'un répertoire, les données sont constituées de la table qui associe à un nom symbolique la référence au fichier, ainsi que sa date et sa taille. Ces deux informations sont copiées depuis l'enregistrement du fichier dans la MFT.

A- On gère une partition de 2 Go par ce système, en utilisant une taille de cluster de 4 Ko. Déterminer le nombre de clusters, l'occupation du fichier bitmap, le nombre maximum de fichiers et l'espace perdu en moyenne par fichier. Comparer avec le système de gestion de fichiers utilisant la FAT.

B- Evaluer la taille de fichier qui permet au contenu d'être dans l'enregistrement du fichier dans la MFT. Donner les avantages et inconvénients de cette méthode.

C- Quelle peut être l'utilité du numéro de séquence dans la référence d'un fichier ?

D- La date et la taille d'un fichier sont dupliquées, d'une part dans le répertoire d'accès au fichier, d'autre part dans la MFT. Donner les avantages et les inconvénients de cette redondance.

Solution de l'exercice 3.14

3.14.1. Question A

Le nombre de clusters est obtenu en divisant la taille de la partition par la taille d'un cluster. On obtient 524288 clusters. Comme il faut 1 bit par cluster, cela conduit à une taille du fichier bitmap de 64 Ko. Le descripteur de fichier est mémorisé dans un cluster, ce qui limite à 524288 le nombre de fichiers, si leur contenu peut être tout entier dans ce descripteur. Notons que le descripteur de

fichier est identifié par un numéro sur 48 bits, permettant un nombre de fichiers largement supérieur. Si certains fichiers débordent, l'espace qu'ils occupent limitera ce nombre. En d'autre termes, il n'y a pas de limite dans le nombre de fichiers, si ce n'est l'espace qu'ils occupent. L'espace perdu est en moyenne 1/2 quantum par fichier soit ici 2 Ko.

Comparaison avec le système FAT. Le nombre de clusters est limité à 65536, c'est-à-dire 8 fois moins que pour NTFS. Pour gérer un disque de 2 Go, FAT demande alors une taille de cluster de 32 Ko. La table d'allocation elle-même est cette fois de 128 Ko, donc deux fois plus grande. Le nombre de fichiers avec le système FAT est limité par le nombre de clusters, un fichier non vide devant contenir au moins 1 cluster. Il y a donc 8 fois moins de fichiers dans ce cas. Enfin, l'espace perdu par fichier étant de 1/2 quantum, elle est cette fois de 16 Ko. On peut donc dire que le système FAT n'est pas adapté à la gestion de partition de 2 Go, et que NTFS est beaucoup plus satisfaisant dans ce cas.

3.14.2. Question B

Le descripteur de fichier contient en moyenne 116 octets, en dehors des données propres du fichier. Comme la taille est limité à 4 Ko, il y a de la place pour 3980 octets de données en moyenne. Cela veut dire que les données des petits fichiers seront contenues dans leur descripteur. On peut y voir deux avantages : d'une part, un accès immédiat au contenu du fichier à partir du descripteur au lieu d'une indirection et d'autre part un espace alloué minimum tout en ayant des tailles variables pour les différents champs du descripteur.

3.14.3. Question C

Les fichiers sont identifiés par un couple <ns, nf> où nf est le numéro de l'enregistrement du descripteur du fichier dans la MFT et ns un numéro de séquence incrémenté lors de chaque création d'un nouveau fichier qui réutilise ce numéro d'enregistrement. Il est possible que la référence au fichier soit utilisée à différents endroits sur le disque, comme par exemple dans des répertoires. En cas de destruction du fichier, il ne devrait plus exister de référence vers ce fichier. Cependant, en cas d'anomalie, une telle référence peut encore être présente. Si elle est exploitée avant la réutilisation de l'enregistrement de la table MFT, cela conduira à une erreur, pourvu que l'on ait pris soin de marquer l'enregistrement comme invalide. Si l'enregistrement est réutilisé pour un nouveau fichier, il faut alors pouvoir détecter que l'a référence est ancienne et ne correspond pas au nouveau fichier. Pour cela il suffit que le numéro de séquence soit mémorisé dans l'enregistrement, ce qui permet de vérifier qu'une référence pour cet enregistrement n'est pas périmée.

3.14.4. Question D

La date et la taille d'un fichier sont dupliquées, d'une part dans le répertoire d'accès au fichier, d'autre part dans la MFT. Ces informations font partie du descripteur d'un objet externe et doivent donc se trouver dans la MFT. Par contre, elles ne sont pas nécessaires dans le répertoire d'accès au fichier qui doit contenir surtout la référence au fichier, et permet ainsi de localiser le descripteur. L'avantage de l'avoir en double est de permettre d'en disposer immédiatement lors de l'accès au répertoire sans avoir à consulter le descripteur, par exemple lorsque l'on veut lister le contenu d'un répertoire. L'inconvénient est lié au maintien de la cohérence des données en plusieurs exemplaires. Lors de chaque modification de l'une de ces informations, elle doit être recopiée en différents endroits sur disque. L'utilisation de la mémoire cache et de l'écriture paresseuse (lors de la suppression du cache par exemple) évite une trop grande dégradation des performances. L'utilisation du fichier journal (§11.3.4) permet de garantir que toutes les copies sont bien identiques, même en cas de panne.

3.15. Étude de la sauvegarde des fichiers

Ce problème a pour but l'étude des conditions d'exploitation de la sauvegarde régulière des fichiers disques.

Pour cela on considère une installation équipée de 5 piles de disques de chacune 900 Mo, dont le temps d'accès moyen est de 21 ms, et le débit de 2.5 Mo/sec. Par ailleurs elle dispose de dérouleurs à 6250 BpI, dont le débit est de 500 Ko/sec. Les bandes utilisées font 2400 pieds.

A- On effectue la sauvegarde en utilisant des blocs de 32 Ko, tant en lecture disque qu'en écriture bande. Combien faut-il de bandes par pile, combien au total, quel temps demande cette sauvegarde complète?

B- Devant les temps importants de la sauvegarde complète, on désire faire une sauvegarde incrémentale. Des mesures faites sur le système en cours d'exploitation ont donné les résultats suivants:

. la taille des blocs est de 8 Ko,

. il y a en moyenne 10 accès disque par seconde,

. il y a environ 4 lectures pour 1 écriture.

On décide d'écrire systématiquement sur bande les blocs qui sont écrits sur disque. Déterminer, pour une période de 24 heures, combien de bandes sont nécessaires. Proposer une méthode d'exploitation des bandes de sauvegarde complète et incrémentale pour restaurer une pile de disques endommagée.

C- Un utilisateur fait remarquer qu'un même bloc peut être écrit plusieurs fois dans une même période de 24 heures, aussi, pour réduire encore le nombre de bandes, et améliorer la restauration, on effectue d'autres mesures pour savoir combien de fois en moyenne, un même bloc est écrit dans une période donnée. Les résultats sont les suivants:

période	6 h.	24 h.	2 j.	3 j.	4 j.	5 j.
nombre	3	5	7	8.5	9.5	10

On décide en conséquence de sauvegarder à la fin d'une période de 6 heures les blocs modifiés durant cette période. Déterminer la taille nécessaire, et proposer une exploitation.

D- On décide d'acheter une sixième pile de disques de mêmes caractéristiques, et de l'utiliser pour la sauvegarde incrémentale à la place du dérouleur, en utilisant la méthode de la question C. Tout en conservant la période de 6 heures pour sauvegarder les blocs modifiés pendant la période, on exploite la nature aléatoire du support, pour ne conserver que la dernière version des blocs modifiés, donner la taille de l'occupation disque au bout de 6 h., 24 h., 48 h. et 5 jours. En déduire une exploitation possible.

E- Pour fournir un service complémentaire aux utilisateurs, on décide de faire la sauvegarde toutes les 6 heures, comme en D, mais par fichier modifié, et non plus par bloc, sachant que ceci coûte 50 % d'espace supplémentaire. Quel service peut-on offrir?

Solution du problème 3.15

3.15.1. Question A

L'espace interbloc correspond à 0.75 * 6250 = 4690 octets. En utilisant des blocs de 32 Ko, la bande sera occupée à 32768 / (32768 + 4690) = 87.5 %. Une bande de 2400 pieds peut donc contenir effectivement 158 Mo (on prendra 150 Mo pour simplifier), soit 6 bandes par pile, et 30 bandes pour l'ensemble des disques. La lecture sur le disque pour la sauvegarde, peut être organisée de façon à réduire au minimum les temps d'attente. Il y a cependant au minimum 150 / 2.5 = 60 secondes pour la lecture. En supposant de même que les opérations et le matériel permettent à la bande de ne pas ralentir lors de l'espace interbloc, il faut 180 / 0.5 = 360 secondes pour l'écriture sur bande. L'ordinateur peut éventuellement permettre que les transferts disques et bandes se fassent en même temps, il faut alors 6 mn. par bande (sinon il en faudra 7). Le temps total d'une sauvegarde complète d'un disque est donc de 36 minutes, et de 3 heures pour l'ensembles des 5 disques.

3.15.2. Question B

En 24 heures, il y a 24 * 3600 * 10 = 864000 accès disques, dont 1/5 sont des écritures. Il y a donc 172800 blocs disques de 8 Ko à écrire sur bande. En les regroupant 4 par 4, on aura des blocs de 32 Ko sur bande. L'espace total est de 172800 * 8192, soit 1350 Mo, soit 9 bandes.

Lorsqu'une pile de disques est endommagée, pour la reconstruire, on doit repartir de la dernière sauvegarde complète de cette pile, et utiliser les bandes incrémentales postérieures à cette

sauvegarde pour mettre à jour les blocs disques modifiés depuis cette date. On peut par exemple sauvegarder complètement une pile par jour, par rotation, ce qui demandera 36 minutes et 6 bandes en plus des 9 bandes incrémentales tous les jours (au lieu de 30 bandes et 3 heures!). La restauration d'une pile endommagée peut demander entre 36 minutes (le crash est juste après la sauvegarde) et 36 + 5 * 9 * 6 = 306 minutes, soit 5 heures (le crash est juste au moment où aurait dû commencer la sauvegarde de ce disque).

Notons que chaque bloc bande de 32 Ko contient 4 blocs disque de 8Ko, situés n'importe où. Si les blocs sont équitablement répartis sur les 5 piles, il y aura, par bande, 150000 / (8 * 5) = 3750 blocs de 8 Ko pour cette pile; comme l'écriture aléatoire d'un bloc demande 21 + 3.2 ms, le temps d'écriture disque total par bande de sauvegarde incrémentale est donc 3750 * 24.2 = 90750 ms, soit 1.5 minutes, et donc inférieur à la lecture de la bande elle-même.

3.15.3. Question C

Pour une période de 6 heures, il y a 6 * 3600 * 10 = 216000 accès disque, dont 43200 écritures. En fin de période, puisque les blocs écrits durant la période le sont en moyenne 3 fois, il y a donc en fait 43200 / 3 = 14400 blocs disques à écrire sur bande, soit environ 115 Mo, soit moins de 1 bande par période de 6 heures, soit 4 bandes par jour (au lieu de 9 dans la question précédente). La restauration d'un disque endommagé demandera entre 36 minutes et 36 + 5 * 4 * 5 = 136 minutes soit 2 h. 30 mn. (les bandes étant de 115 Mo, il ne faut plus que 5 minutes pour les lire).

Pour diminuer ce temps de restauration, il faudrait que les bandes utilisées ne contiennent que des informations concernant la pile à restaurer. Ceci n'était pas envisageable en B, car il aurait fallu en permanence 1 dérouleur par pile. Par contre, ici, la sauvegarde incrémentale étant déclenchée périodiquement, peut être faite pile par pile. En moyenne, chaque pile donne lieu à 115 / 5 = 23 Mo de sauvegarde incrémentale toutes les 6 heures. Une bande pourra contenir 6 sauvegardes incrémentales successives d'une même pile de disques. Il faudra donc 5 * 4 / 6 = 3.3 bandes par pile de disques entre deux sauvegardes complètes de cette pile. Si le temps de lecture d'une bande incrémentale demande cette fois 5.5 minutes (138 Mo), tous les blocs contenus sur cette bande sont relatifs à la même pile. Il y aura alors 14400 / 5 * 6 = 17280 blocs de 8 Ko à réécrire sur disque par bande de sauvegarde incrémentale, ce qui demandera 17280 * 24.2 = 418 secondes, soit 7 minutes. La réécriture sur disque des blocs de la sauvegarde incrémentale prendra plus de temps que la lecture de la bande. La restauration demandera donc entre 36 minutes et 36 + 3.3 * 7 = 59 minutes.

3.15.4. Question D

Si on utilise une pile de disques spécialisée pour la sauvegarde incrémentale, lorsqu'on recopie un bloc sur le disque spécialisé, on peut récupérer l'espace occupé éventuellement par une version antérieure de ce bloc. Dans ces conditions, l'espace nécessaire pour une pile donnée, et à un moment donné, correspond au nombre de blocs modifiés depuis la dernière sauvegarde complète de cette pile. En supposant que les opérations disques sont équitablement réparties sur toutes les piles, cet espace est le suivant:

moment	6h.	24 h.	2 j.	3 j.	4 j.	5 j.
écritures	8640	34560	69120	103680	138240	172800
nombre blocs	2880	6912	9874	12198	14552	17280
espace	23 Mo	55 Mo	79 Mo	98 Mo	116 Mo	137 Mo

On peut donc en conclure que l'espace total moyen nécessaire est égal à 55 + 79 + 98 + 116 + 137 = 485 Mo. Le disque supplémentaire est donc amplement suffisant.

L'exploitation pourrait être la suivante:

- toutes les 6 heures, sauvegarde incrémentale:
  - sauvegarde sur le disque spécialisé de tous les blocs modifiés depuis le début de période.
  - récupération de l'espace occupé par les anciennes versions de ces blocs sur le disque spécialisé.
- tous les jours (ou toutes les 4 sauvegardes incrémentales), après une sauvegarde incrémentale, sauvegarde complète de la pile de numéro j **mod** 5, puis suppression sur le disque spécialisé des blocs provenant de cette pile.

Remarque: il peut être judicieux que la sauvegarde complète d'une pile soit non seulement sur bande, mais aussi sur un disque, si l'unité supplémentaire accepte les cartouches amovibles. Cela rallonge éventuellement la sauvegarde de 900 / 2.5 = 360 secondes, soit 6 minutes (peut-être pas si les opérations d'écritures bandes peuvent être simultanées à l'écriture disque). En cas de crash disque, on peut repartir depuis la cartouche, et la recopier sur la pile correspondante, ce qui demandera 12 minutes (6 minutes si l'installation permet la simultanéité des lectures et des écritures sur deux piles différentes), au lieu de 36 minutes à partir de bande. Il faut ensuite mettre à jour les blocs modifiés depuis cette sauvegarde complète en utilisant la sauvegarde incrémentale sur disque spécialisé, ce qui demandera au pire 2 * 24.2 * 17280 = 836 secondes, soit 14 minutes (7 minutes si la simultanéité des lectures et des écritures est possible).

3.15.5. Question E

Si on fait la sauvegarde incrémentale par fichier, et non plus par bloc, cela coûtera 50 % d'espace de plus, et nécessitera donc 730 Mo, ce qui est encore acceptable. L'avantage de faire cette sauvegarde par fichier, est de permettre la récupération non plus par pile, mais par fichier. En effet, le fichier étant identifiable en tant que tel dans les supports de sauvegarde, une telle récupération consiste d'abord à rechercher le fichier sur les sauvegardes incrémentales. S'il est trouvé, la recherche s'arrête. S'il n'est pas trouvé, la recherche se poursuit sur la dernière sauvegarde complète de la pile (ou de toutes les piles). Si on garde plusieurs sauvegardes complètes successives, il est aussi possible de remonter dans ces sauvegardes en commençant aux plus récentes, vers les plus anciennes. C'est pourquoi on distingue souvent 3 niveaux de sauvegardes:

- sauvegarde incrémentale, périodicité 6 heures, conservation 1 semaine.
- sauvegarde partielle (une pile), périodicité 5 jours, conservation 1 mois.
- sauvegarde complète (toutes les piles), périodicité 1 mois, conservation 1 an.

Une telle exploitation permet de fournir à l'utilisateur le service de restauration d'un fichier particulier jusqu'à 1 an après sa destruction sur les disques.

3.16. Sécurité et protection dans Windows NT

On s'intéresse à quelques aspects de la sécurité et de la protection.

Windows NT gère un bit, appelé "archive", pour chaque fichier, qui est mis à 1 automatiquement chaque fois que le fichier est modifié. De plus une fonction système permet de le modifier par programme. Un programme est fourni, qui permet de sauvegarder un ensemble de fichiers[5] selon l'une des trois méthodes suivantes:

- La méthode normale copie les fichiers sans se préoccuper de la valeur de leur bit d'archive, et remet ce bit à 0.
- La méthode incrémentale copie les fichiers dont le bit d'archive est à 1, et remet ce bit à 0.
- La méthode différentielle copie les fichiers dont le bit d'archive est à 1, sans le modifier.

A- Quels avantages ou inconvénients voyez-vous à chacune des trois méthodes?

B- Vous êtes responsable d'une machine équipée de Windows NT, et disposant d'un disque de 10 Go et d'un lecteur de cartouches[6] de 2 Go. Chacun de ces deux périphériques a un débit moyen de 1 Mo/sec, temps d'accès compris. Quelle serait votre stratégie pour définir votre politique de sauvegarde, et garantir un service suffisamment fiable aux utilisateurs de cette machine.

C- Comment pourrait-on implanter chacune des trois méthodes lorsque le système gère une date de dernière modification au lieu d'un bit d'archive.

D- Pour accéder à Windows NT, il faut disposer d'un compte sur la machine. Chaque compte, protégé par un mot de passe, est classé dans un groupe, ce qui confère alors au propriétaire du compte des droits (c'est-à-dire, lui permet d'effectuer certaines opérations) sur la machine et sur les

[5] On ne se préoccupe pas ici de savoir comment on détermine cet ensemble de fichiers à sauvegarder.
[6] Une cartouche est l'équivalent moderne d'une bande magnétique.

fichiers. Windows NT distingue des groupes d'utilisateurs, le groupe des opérateurs de sauvegarde et le groupe des administrateurs[7]. Les administrateurs ont tous les pouvoirs, tandis que les opérateurs de sauvegarde peuvent faire la sauvegarde et la restauration de tous les fichiers, et les utilisateurs ont le droit de lire et d'écrire dans les fichiers attachés à leur groupe.

D.1- Expliquez, en quelques lignes, pourquoi un système doit disposer d'un compte particulier disposant de tous les pouvoirs.

D.2- Quel intérêt voyez-vous au fait de disposer d'un groupe d'administrateurs au lieu d'un seul compte administrateur.

D.3- Expliquez, en quelques lignes, ce qui peut empêcher les sauvegardes et restaurations d'être effectuées par un utilisateur ordinaire.

D.4- Qu'est-ce qui justifie, à votre avis, l'introduction du groupe des opérateurs de sauvegarde en plus du groupe des administrateurs.

Solution de l'exercice 3.16

3.16.1. Question A

Brièvement la première méthode est une sauvegarde totale de l'ensemble des fichiers. Cela donne un état cohérent de ces fichiers, mais est relativement coûteuse en temps.

La deuxième méthode permet de ne sauvegarder que les fichiers qui ont été modifiés depuis la dernière sauvegarde totale ou incrémentale. Elle est beaucoup plus rapide, puisque le minimum est fait. La restauration nécessitera de retrouver dans différents supports la dernière version des objets sauvegardés.

La troisième méthode est très proche de la sauvegarde incrémentale. La non modification du bit d'archive implique que la sauvegarde suivante, quelle qu'elle soit, sauvegardera de nouveau ces fichiers.

Ces différentes méthodes sont utilisées conjointement à différents moments. Notons ST, SI et SD chacune de ces méthodes. Considérons une suite de ces sauvegardes :

$$ST_1, SD_2, SD_3, SD_4, SI_5, SD_6, SD_7, SI_8, SD_9, SD_{10}, ST_{11},$$

Regardons l'ensemble des fichiers sauvegardés lors de chacune de ces sauvegardes, en supposant qu'il n'y a aucune suppression. Nous avons les relations suivantes : $SD_2 \subseteq SD_3 \subseteq SD_4 \subseteq SI_5$, d'une part, ainsi que $SD_6 \subseteq SD_7 \subseteq SI_8$, et enfin $SD_9 \subseteq SD_{10} \subseteq ST_{11}$. On peut voir que toute sauvegarde différentielle peut être supprimée après la sauvegarde suivante, alors que les sauvegardes totales ou incrémentales ne peuvent être supprimées que par une sauvegarde totale. Du point de vue de la restauration consécutive à une panne, il faut remonter dans la suite des sauvegardes jusqu'à la dernière sauvegarde totale, en tenant compte de ces suppressions.

Si on ne fait pas de sauvegarde incrémentale, la restauration consiste à restaurer la dernière sauvegarde totale puis la dernière sauvegarde différentielle qui l'a suivie. En présence de sauvegarde incrémentale, la restauration consiste à restaurer la dernière sauvegarde totale puis les sauvegardes incrémentales qui l'ont suivies (dans l'ordre) et enfin la dernière sauvegarde différentielle éventuelle postérieure à la dernière sauvegarde prise en compte. Dans notre exemple, une panne juste avant ST_{11}, nécessite de prendre dans l'ordre ST_1, SI_5, SI_8, SD_{10}, alors qu'une panne juste avant SI_5, nécessite de prendre dans l'ordre ST_1, SD_4.

3.16.2. Question B

Évaluons rapidement le temps d'une sauvegarde totale lorsque le disque est occupé à 50%, soit 5 Go. Il faut donc 2,5 cartouches pour une sauvegarde totale. En supposant que la lecture disque peut être simultanée avec l'écriture sur la cartouche, il faudra néanmoins 5000 secondes pour une sauvegarde totale, soit environ 1,5 heure.

[7] Il y en a d'autres qui ne nous intéressent pas ici.

On sait que en général, assez peu de données sont modifiées chaque jour. Le bit d'archive permet de savoir celles dont la sauvegarde est en fait nécessaire. La difficulté est dans le choix entre la sauvegarde incrémentale et la sauvegarde différentielle. On peut noter, comme nous l'avons vu dans la question précédente, que les sauvegardes différentielles augmentent en taille à chaque fois jusqu'à ce qu'un autre type de sauvegarde soit effectué. Au contraire les sauvegardes incrémentales devraient sensiblement être de même taille, pourvu qu'elles correspondent à une même période et à une utilisation semblable. Inversement, une restauration n'a à prendre en compte qu'une seule sauvegarde différentielle, alors qu'il faut prendre toutes les sauvegardes incrémentales.

Supposons que 1% des fichiers soient modifiés par période de 24 heures. Une sauvegarde incrémentale journalière prendra donc 1 minute chaque jour, ce qui est négligeable. Une sauvegarde différentielle journalière, demandera 1 mn le premier jour suivant une autre sauvegarde, et augmentera ensuite chaque jour, sans toutefois être proportionnelle au nombre de jours écoulés. Si on reprend les données de l'exercice précédent, cela ne demanderait que 2,5 mn après 5 jours.

Une proposition pourrait être de faire une sauvegarde totale tous les mois. Une sauvegarde incrémentale toutes les semaines, et une sauvegarde différentielle tous les jours, ou à des périodes encore plus rapprochées, comme par exemple toutes les 6 heures. La durée d'une sauvegarde différentielle pourrait d'ailleurs un bon critère pour envisager une sauvegarde incrémentale, et le nombre de sauvegardes incrémentales depuis la dernière sauvegarde totale serait un bon critère pour décider d'une sauvegarde totale.

3.16.3. Question C

Le bit d'archive permet de savoir que le fichier a été modifié depuis la dernière remise à zéro de ce bit, c'est-à-dire, depuis la dernière sauvegarde totale ou incrémentale. Il suffit donc de mémoriser la date de l'une ou l'autre de ces sauvegardes. Le bit d'archive peut alors être simulé par la comparaison de la date de dernière modification avec cette date de dernière sauvegarde.

3.16.4. Question D

3.16.4.1. Question D.1

Certaine opérations, comme la création d'un compte utilisateur ou sa suppression, nécessite des pouvoirs étendus qui ne peuvent être attribués à un utilisateur quelconque. Il en va de même lorsqu'un utilisateur ne respecte pas certaines règles imposées par l'administrateur du système, ce qui nécessite de contraindre et limiter ses actions. Notons enfin que la mise en route du système sur une machine nue implique que quelqu'un dispose de tous les pouvoirs à ce moment. Certains systèmes imposaient que toutes les actions sensibles soient faites avant cette mise en route, ce qui interdit, par exemple, de créer un utilisateur lorsque le système est opérationnel ! Il n'est pas pratique de devoir arrêter le système pour apporter des modifications sensibles, qui ne perturbent cependant pas son fonctionnement.

3.16.4.2. Question D.2

Disposer d'un seul compte administrateur oblige que les opérations qui nécessitent tous les pouvoirs soient effectuées sous ce compte. Pour le système, il s'agit alors d'une seule et même personne, avec la même identification et le même mot de passe. Au contraire, disposer d'un groupe d'administrateur permet de conserver l'identification individuelle de chaque administrateur, avec leur propre mot de passe. Si un administrateur perd la qualité d'administrateur, il suffit de l'enlever du groupe des administrateurs sans perturber les autres et sans devoir changer le mot de passe de l'administrateur unique.

3.16.4.3. Question D.3

La sauvegarde et la restauration demande de pouvoir parcourir une arborescence de fichiers et consulter des indicateurs pour savoir ce qui doit être sauvegarder. De plus, il faut disposer des droits correspondants sur les fichiers concernés. Un utilisateur ordinaire possède en général les accès sur les objets qui lui appartiennent, mais pas sur ceux des autres.

3.16.4.4. Question D.4

Pour pouvoir faire les sauvegardes et les restaurations, il faut disposer de droits particuliers sur les objets externes du système de fichier, mais il n’est pas nécessaire de disposer de tous les droits sur tous les objets du système. Le fait d’introduire un groupe pour les opérateurs de sauvegarde permet que ces opérateurs soient chacun identifiés séparément par le système avec les pouvoirs juste nécessaires pour effectuer ces sauvegardes et ces restaurations. En particulier, on pourra limiter les applications qu’ils sont autorisés à lancer.

3.17. Dates des fichiers

Certains systèmes mémorisent pour chaque fichier ou répertoire les informations suivantes:

- date à la seconde près de la création,
- date à la seconde près de la dernière modification,
- date à la seconde près du dernier accès.

A- Quelle peut être l'utilité de chacune de ces informations pour l'utilisateur? En particulier, on s'intéressera aux cas des fichiers sources, des fichiers résultats de compilation, et des fichiers résultats de l'édition de liens.

B- Quelles procédures automatiques peuvent être appliquées par le système à partir de ces informations?

Solution de l’exercice 3.17

3.17.1. Question A

La date de création permet de savoir à quand remonte la création du fichier, et donc depuis quand ce fichier peut être manipulé. S'il est créé par un programme spécifique, il est ainsi possible de savoir quand ce programme a été exécuté pour la dernière fois. Par exemple, s'il a une structure spécifique, et que cette structure est modifiée à un certain moment, il est possible de savoir s'il a la nouvelle structure ou non. Lorsqu'on a plusieurs copies de sauvegarde d'un même fichier, on peut ainsi savoir quelle est la plus récente.

La date de dernière modification permet de savoir si le contenu du fichier est à jour, c'est-à-dire s'il contient les dernières mises à jour des données. Si on a un état daté du contenu du fichier, on peut savoir si cet état est à jour, ou s'il est périmé. Certains systèmes utilisent cette date, conjointement à une période de rétention T, pour interdire une nouvelle modification du fichier pendant ce temps T qui suit une modification. On évite ainsi d'exécuter plusieurs fois un même traitement de mise à jour.

La date du dernier accès permet de savoir si le fichier est effectivement utilisé, et donc d'éviter de conserver sur support immédiatement accessible des informations qui ne sont pas utilisées, ce qui permet des économies de place sur ce support.

Dans le cas particulier des fichiers qui servent à la construction des programmes, les dates de dernière modification peuvent être utilisées pour savoir si un programme exécutable (résultat d'une édition de liens) a bien été construit avec les dernières modifications des modules sources. Pour cela, il faut que la date de dernière modification du fichier contenant ce programme exécutable soit postérieure à toutes les dates de dernière modification des fichiers contenant les modules objets constituant ce programme. De plus, la date de dernière modification de chacun de ces fichiers doit être également postérieure à celles des fichiers sources qui ont permis sa construction. La prise en compte de ces règles d'antériorité permet donc de décider, après une modification de fichiers sources, si les recompilations et les éditions de liens nécessitées par ces modifications ont été faites ou non. Rappelons que certains systèmes proposent un outil, appelé “make” qui effectue ceci automatiquement. Utilisant un fichier de dépendances explicitant pour chaque programme exécutable quels sont les modules objets qui le composent, et pour chaque module objet quels sont les fichiers sources qui le définissent, cet outil contrôle que les règles d'antériorité sont vérifiées par

les dépendances. Si ce n'est pas le cas, l'outil exécute automatiquement la compilation ou l'édition de liens nécessaire.

3.17.2. Question B

Le système peut tenir compte de la date de dernière modification pour ne sauvegarder que les fichiers qui ont été modifiés depuis la dernière sauvegarde. On obtient ainsi une sauvegarde incrémentale, en général plus rapide et moins encombrante qu'une sauvegarde complète.

Si le système utilise plusieurs types de disques de temps d'accès variable, il peut tenir compte des dates du dernier accès pour déplacer les fichiers les moins utilisés sur les supports les plus lents, et ceux les plus récemment utilisés sur les supports les plus rapides. On pourrait envisager de détruire automatiquement les fichiers très anciens sous réserve qu'ils aient été sauvegardés, cependant cette méthode est rarement admise par les utilisateurs, qui préfèrent décider eux-mêmes de cette destruction; mais on constate bien souvent qu'ils ne le font pas et que l'espace disque est encombré de fichiers inutiles et inutilisés. Il arrive d'ailleurs que les utilisateurs ne se rendent pas compte que certains de leurs fichiers ont ainsi été supprimés après leur sauvegarde par l'équipe système!

4

Environnement physique

4.1. Utilisation de l'horloge physique

Un dispositif extérieur déclenche une interruption toutes les millisecondes. La prise en compte de cette interruption par le système entraîne l'exécution du programme suivant:

```
{ programme sous IT horloge }
cpt_mes := ( cpt_mes + 1 ) mod 1000;
si cpt_mes = 0 alors
     cpt_sec := ( cpt_sec + 1 ) mod 60;
     si cpt_sec = 0 alors
          cpt_min := ( cpt_min + 1 ) mod 60;
          si cpt_min = 0 alors
               cpt_h := ( cpt_h + 1 ) mod 24;
               si cpt_h = 0 alors cpt_j := cpt_j + 1;
               finsi;
          finsi;
     finsi;
finsi;
```

A- Dé1finir la fonction système qui délivre dans un tableau de 5 entiers la date exacte, à la milliseconde près, au moment de l'appel. On prendra garde au fait que l'interruption d'horloge peut survenir pendant l'exécution de cette fonction.

B- Dans le cas d'un système en monoprogrammation, comment vous proposeriez-vous d'utiliser cette fonction système pour optimiser la durée d'exécution d'un programme?

C- Dans le cas d'un système en multiprogrammation, qu'est-ce qui empêche d'utiliser cette fonction pour aider à optimiser la durée d'exécution d'un programme?

Solution de l’exercice 4.1

4.1.1. Question A

L'interruption d'horloge pouvant intervenir à tout moment, si elle est prise en compte pendant l'appel de la fonction système, certains des entiers transmis seront relatifs au temps précédant l'interruption, d'autres à celui qui le suit, et il y aura incohérence dans les données. Pour éviter que l'interruption soit prise en compte pendant l'exécution de la fonction, il suffit de masquer les interruptions au début et de les démasquer à la fin. Si on ne veut pas perdre d'interruption d'horloge, il faut que la durée de ce masquage soit assez bref. Étant donné que cette exécution est plus simple que le programme de prise en compte de l'interruption elle-même, on peut penser que ce sera bien le cas, sinon la machine passerait son temps à mettre à jour les compteurs de l'horloge!

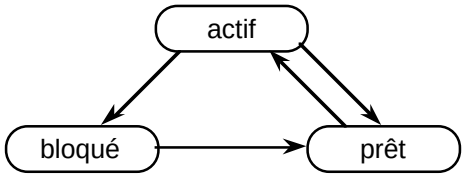
```
fonction temps ( var t: tableau [1..5] de entier);
début
     masquer_IT_horloge;
     t [1] := cpt_mes;
     t [2] := cpt_sec;
     t [3] := cpt_min;
     t [4] := cpt_h;
     t [5] := cpt_j;
     démasquer_IT_horloge;
fin;
```

4.1.2. Question B

Cette fonction permet donc de connaître à tout moment l'heure exacte. Si on l'appelle à différents endroits d'un programme, on peut alors, par différence, calculer le temps écoulé entre deux appels successifs. En monoprogrammation, le programme étant seul en mémoire, ce temps est donc la durée d'exécution (temps Unité Centrale et temps d'entrées-sorties) du programme. En faisant de telles mesures, il est donc possible de savoir quelles parties du programme prennent le plus de temps, et essayer d'optimiser ces parties.

4.1.3. Question C

Dans le cas d'un système en multiprogrammation, un processus passe successivement dans trois états: prêt, élu, bloqué. La mesure de durée obtenue par différence entre deux appels à la fonction temps, mesure en fait le temps pendant lequel le processus a utilisé l'unité centrale, et le temps d'attente des entrées-sorties (bloqué), mais aussi le temps pendant lequel il a été en attente du processeur attribué à un autre processus. Ce dernier temps, imprévisible et sans lien avec le comportement du programme, doit pouvoir être éliminé des mesures, si on désire les utiliser pour savoir les endroits du programme qui sont les plus coûteux. Les mesures importantes pour optimiser la durée d'exécution d'un programme sont en fait le *temps virtuel*, représentant le temps pendant lequel le processus a été actif, et le temps pendant lequel le processus est resté bloqué. Ceux-ci peuvent être mesurés par le système chaque fois qu'il change l'état du processus suivant le graphe ci-dessus.



4.2. Ordonnancement de processus

On considère un système monoprocesseur et les 4 processus P1, P2, P3 et P4 qui effectuent du calcul et des entrées/sorties avec un disque selon les temps donnés ci-dessous :

Processus P1
- Calcul : 3 unités de temps
- E/S : 7 unités de temps
- Calcul : 2 unités de temps
- E/S : 1 unité de temps
- Calcul : 1 unité de temps

Processus P2
- Calcul : 4 unités de temps
- E/S : 2 unités de temps
- Calcul : 3 unités de temps
- E/S : 1 unité de temps
- Calcul : 1 unité de temps

Processus P3
- Calcul : 2 unités de temps
- E/S : 3 unités de temps
- Calcul : 2 unités de temps

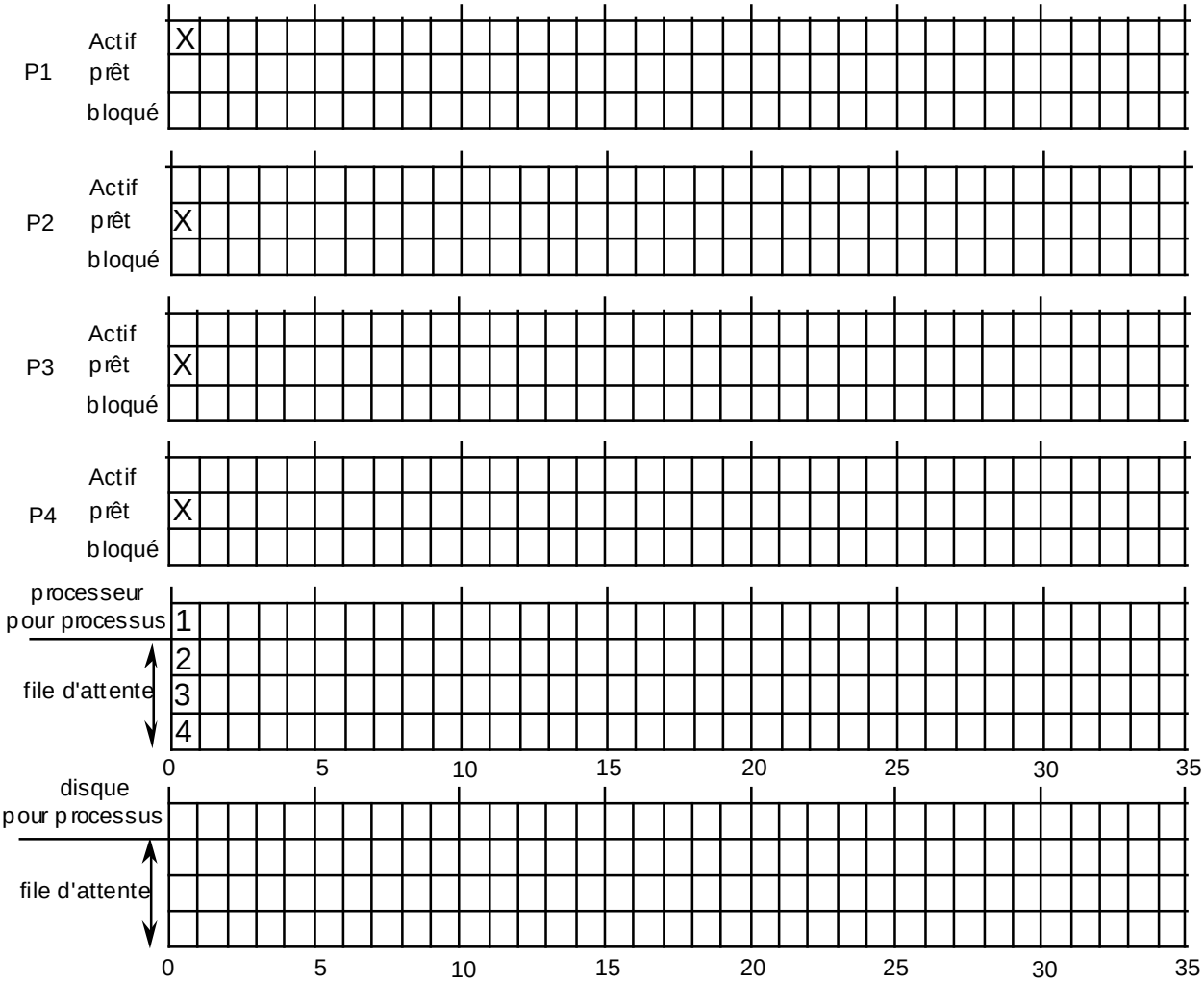
Processus P4
- Calcul : 7 unités de temps

Les trois parties sont indépendantes.

A- On considère que l'ordonnancement sur le processeur se fait selon une politique FIFO : le processus élu à un instant t est celui qui est le plus anciennement dans l'état prêt. Initialement, l'ordre de soumission des processus est P1, puis P2, puis P3, puis P4.

De même, on considère que l'ordre de services des requêtes d'E/S pour le disque se fait selon une politique FIFO.

Sur graphe suivant[8], donnez le chronogramme d'exécution des 4 processus P1, P2, P3 et P4. Vous distinguerez les états des processus : Prêt, Actif et Bloqué et vous indiquerez le contenu des files d'attente des processus (attente processeur et attente du disque[9]). Pour vous guider, la première unité de temps est déjà portée sur le chronogramme. Justifiez votre raisonnement, en expliquant la gestion des files d'attentes et les transitions des processus. Donnez le temps de réponse moyen obtenu.



B- On considère maintenant que l'ordonnancement sur le processeur se fait selon une politique à priorité préemptible : le processus élu à un instant t est celui qui le processus prêt de plus forte priorité. On donne priorité (P1) > priorité (P3) > priorité (P2) > priorité (P4).

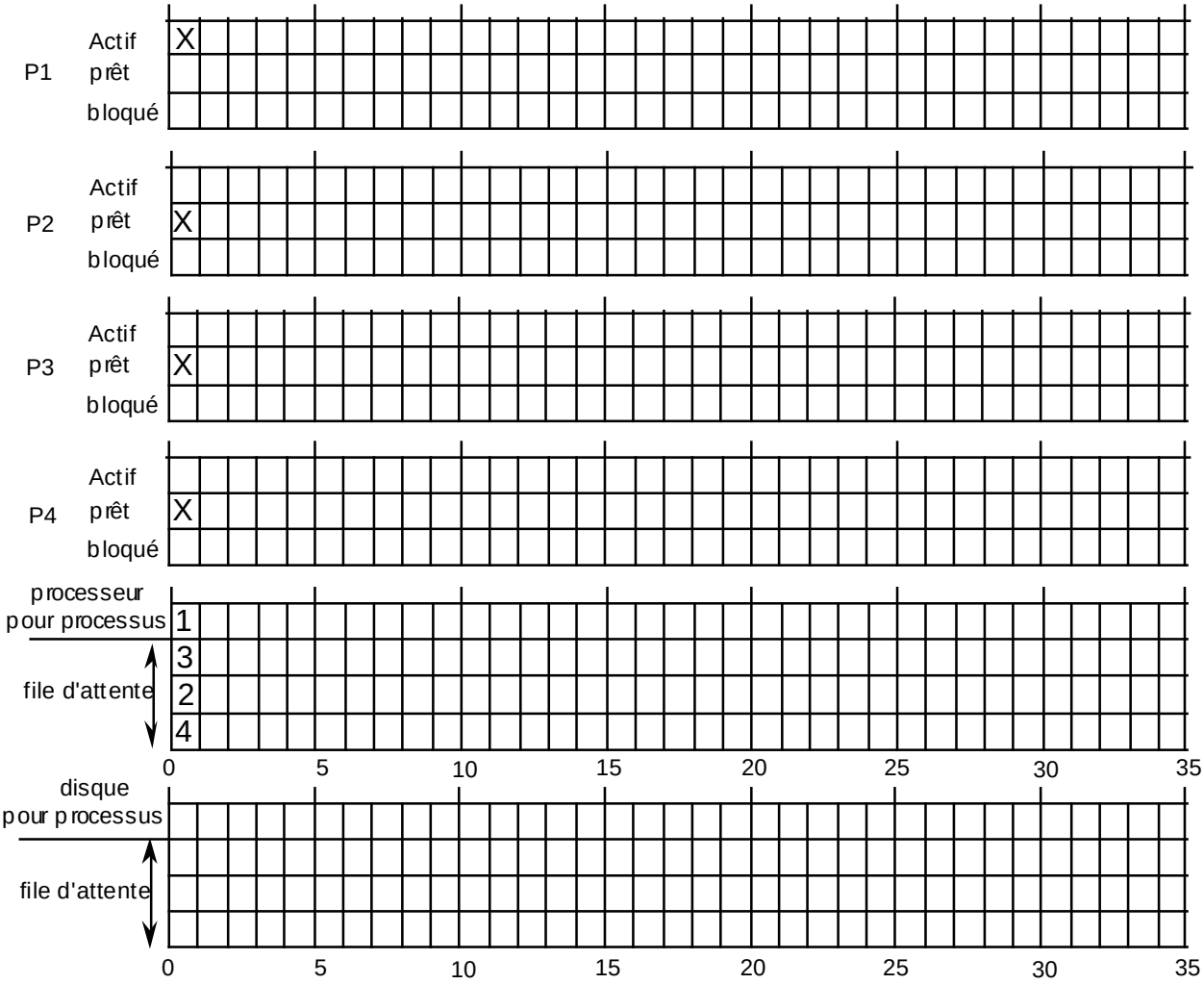
On considère que l'ordre de services des requêtes d'E/S pour le disque se fait toujours selon une politique FIFO.

Sur le graphique suivant, donnez le chronogramme d'exécution des 4 processus P1, P2, P3 et P4. Vous distinguerez les états des processus : Prêt, Actif et Bloqué et vous indiquerez le contenu des files d'attente des processus (attente processeur et attente du disque). Pour vous guider, la première unité de temps est déjà portée sur le chronogramme. Elle diffère du graphique de la question précédente, puisque l'ordre de priorité des processus impose un ordre dans la file d'attente différent.

[8] Note : à chaque instant, la case de la ligne "pour processus" indique le numéro du processus servi par le processeur ou le disque, et les cases des lignes "file d'attente" indiquent les numéros des processus en attente, la tête de file étant dans la case du haut. Ainsi, à l'instant 0, le processus 1 est servi par le processeur, le processus 2 est en tête de file d'attente, suivi du processus 3 puis du processus 4.

[9] Rappelons que le disque en peut exécuter qu'une seule opération à la fois.

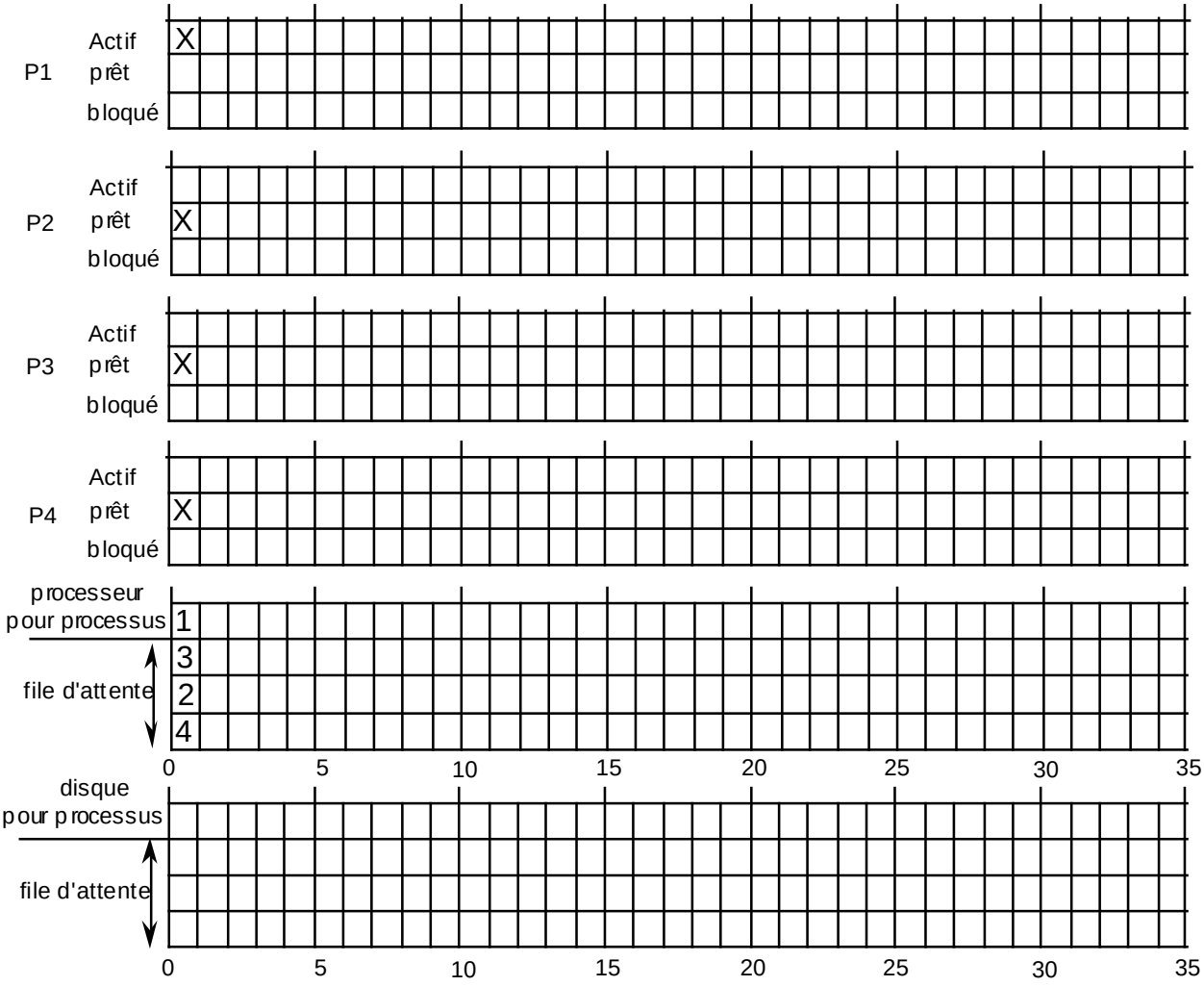
Justifiez votre raisonnement, en expliquant la gestion des files d'attentes et les transitions des processus. Donnez le temps de réponse moyen obtenu.



C- On considère toujours que l'ordonnancement sur le processeur se fait selon une politique à priorité préemptible : l'ordre des priorités des 4 processus reste inchangé.

On considère maintenant que l'ordre de services des requêtes d'E/S pour le disque se fait également selon la priorité des processus : le processus commençant une E/S est celui de plus forte priorité parmi ceux en état d'attente du disque. Une opération d'E/S commencée ne peut pas être préemptée.

Sur graphique suivant, donnez le chronogramme d'exécution des 4 processus P1, P2, P3 et P4. Vous distinguerez les états des processus : Prêt, Actif et Bloqué et vous indiquerez le contenu des files d'attente des processus (attente processeur et attente du disque). Pour vous guider, la première unité de temps est déjà portée sur le chronogramme. Elle est identique à celle du graphique de la question précédente, puisque l'ordre de priorité des processus est le même. Justifiez votre raisonnement, en expliquant la gestion des files d'attentes et les transitions des processus. Donnez le temps de réponse moyen obtenu.



Solution de l'exercice 4.2

4.2.1. Question A

À 0, P1 est actif et obtient le processeur pour 3 UT (fin en 3).

À 3, P1 accède au disque, qui était libre évidemment, pour 7 UT (fin en 10). P2 devient actif et obtient le processeur pour 4 UT (fin en 7).

À 7, P2 passe en tête de file du disque, et P3 devient actif pour 2 UT (fin en 9).

À 9, P3 passe en deuxième de la file disque et P4 devient actif pour 7 UT (fin en 16).

À 10, l'entrée-sortie de P1 se termine et P1 passe en queue de la file du processeur, mais comme elle est vide, il est aussi en tête. L'entrée sortie de P2 commence pour 2 Ut (fin en 12).

À 12, l'entrée-sortie de P2 se termine et P2 passe en queue (en 2) de la file processeur. L'entrée-sortie de P3 commence pour 3 UT (fin en 15).

À 15, l'entrée-sortie de P3 se termine et P3 passe en queue de la file processeur (en 3). La file disque étant vide, le disque devient libre.

À 16, P4 se termine, P1 devient actif et obtient le processeur pour 2UT (fin en 18).

À 18, P1 accède au disque pour 1 Ut (fin en 19) et P2 devient actif pour 3 UT (fin en 21).

À 19, l'entrée-sortie de P1 se termine et P1 passe en queue de la file processeur (en 2).

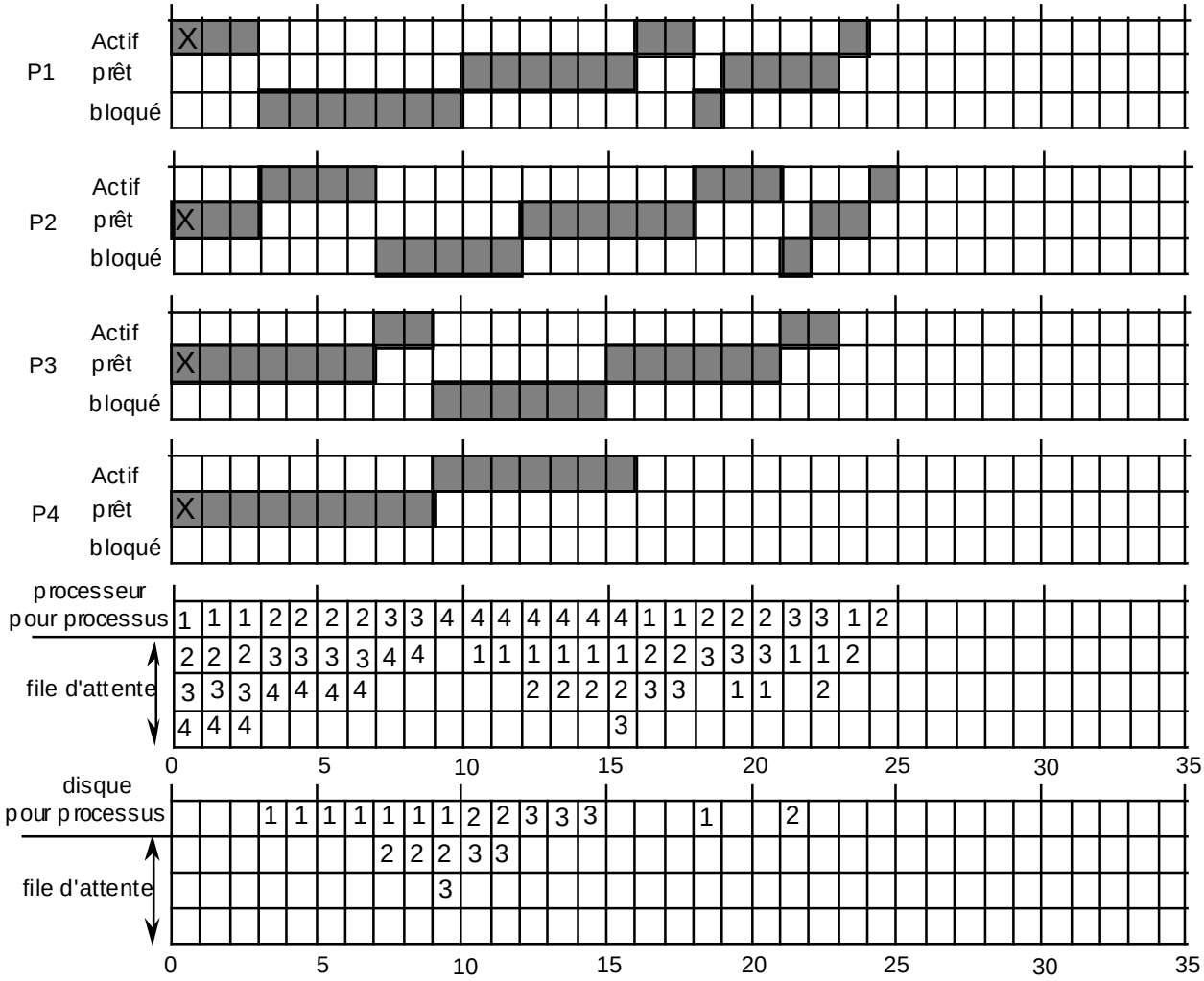
À 21, P2 accède au disque pour 1 UT (fin en 22) et P3 devient actif pour 2 UT (fin en 23).

À 22, l'entrée-sortie de P2 se termine et P2 passe en queue de la file processeur (en 2).

À 23, P3 se termine et P1 devient actif pour 1 UT (fin en 24).

À 24, P1 se termine et P2 devient actif pour 1 UT (fin en 25).

À 25, P2 se termine et il n'y a plus de processus.



Le temps de réponse de P1 est de 24, celui de P2 est de 25, celui de P3 est de 23 et celui de P4 est de 16. Le total est 88, soit une moyenne de 22 UT.

4.2.2. Question B

À 0, P1 est actif et obtient le processeur pour 3 UT (fin en 3).

À 3, P1 accède au disque, qui était libre évidemment, pour 7 UT (fin en 10). P3 devient actif et obtient le processeur pour 2 UT (fin en 5).

À 5, P3 passe en tête de file du disque, et P2 devient actif pour 4 UT (fin en 9).

À 9, P2 passe en deuxième de la file disque et P4 devient actif pour au plus 7 UT (fin ≤ 16).

À 10, l'entrée-sortie de P1 se termine et P1 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 2 UT (fin en 12), et P4 passe en tête de la file du processeur (en 1). L'entrée sortie de P3 commence pour 3 Ut (fin en 13).

À 12, P1 passe en queue de file disque (en 2) et P4 devient actif pour au plus 6 UT (fin ≤ 18).

À 13,l'entrée-sortie de P3 se termine et P3 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 2 UT (fin en 15), et P4 passe en tête de la file processeur. L'entrée-sortie de P2 commence pour 2 UT (fin en 15).

À 15, P3 se termine et l'entrée-sortie de P2 se termine. P2 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour au plus 3 UT (fin ≤ 18), et P4 passe en tête de la file processeur. L'entrée-sortie de P1 commence pour 1 UT (fin en 16).

À 16, l'entrée-sortie de P1 se termine et P1 devient prêt, mais étant de priorité supérieure à celle de P2, devient actif pour 1 UT (fin en 17). P2 est en 1 et P4 en 2 de la file processeur.

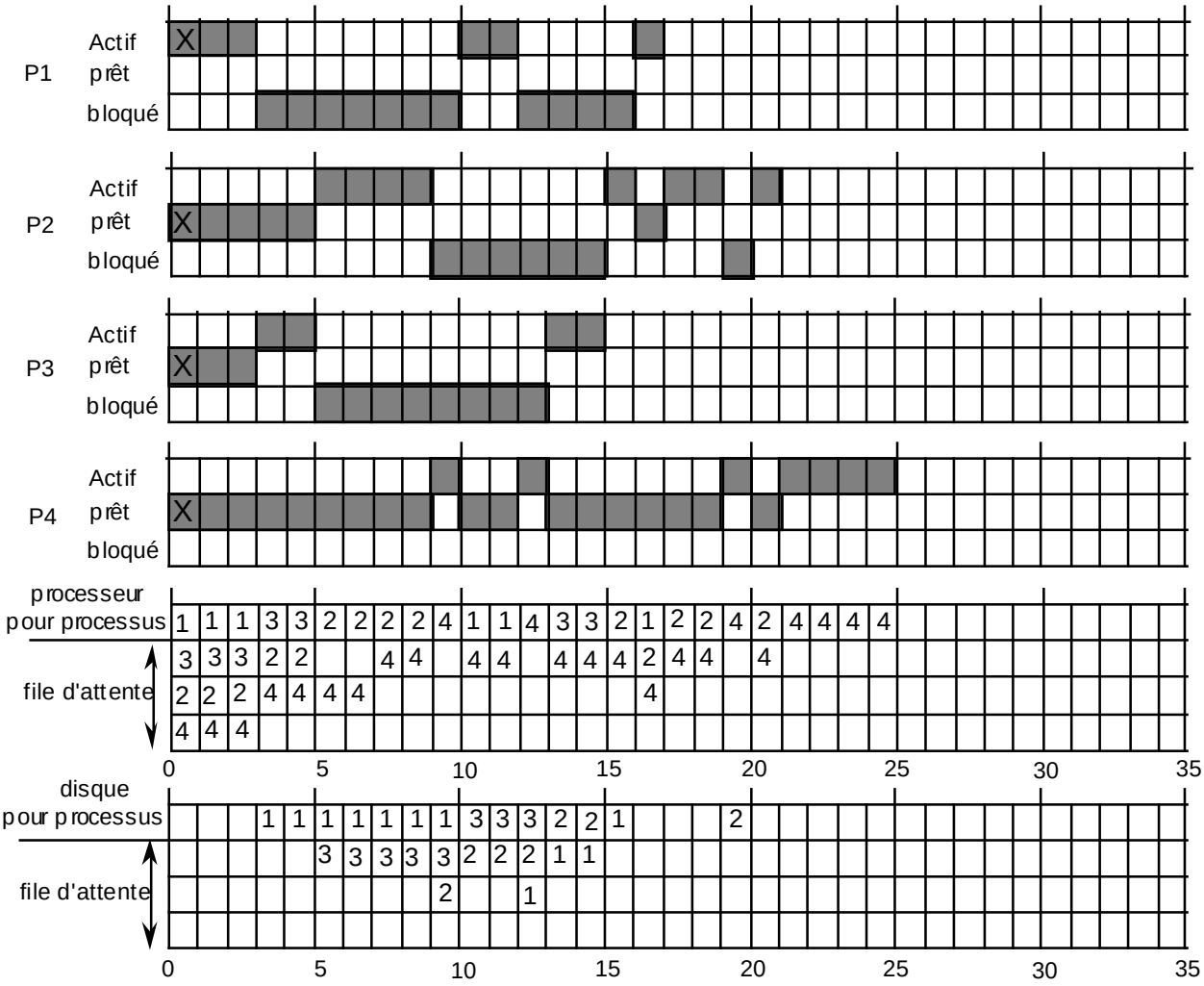
À 17, P1 se termine, P2 devient actif et obtient le processeur pour 2UT (fin en 19).

À 19, P2 accède au disque pour 1 Ut (fin en 20) et P4 devient actif pour au plus 5 UT (fin ≤ 24).

À 20, l'entrée-sortie de P2 se termine et P2 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 1 UT (fin en 21), et P4 passe en tête de la file processeur.

À 21, P2 se termine et P4 devient actif pour 4 UT (fin en 25).

À 25, P4 se termine et il n'y a plus de processus.



Le temps de réponse de P1 est de 17, celui de P2 est de 21, celui de P3 est de 15 et celui de P4 est de 25. Le total est 78, soit une moyenne de 19,5 UT.

4.2.3. Question C

Notons que le début est assez semblable à la question précédente, puisque le seul changement peut intervenir lorsqu'il y a des processus en attente du disque.

À 0, P1 est actif et obtient le processeur pour 3 UT (fin en 3).

À 3, P1 accède au disque, qui était libre évidemment, pour 7 UT (fin en 10). P3 devient actif et obtient le processeur pour 2 UT (fin en 5).

À 5, P3 passe en tête de file du disque, et P2 devient actif pour 4 UT (fin en 9).

À 9, P2 passe en deuxième de la file disque et P4 devient actif pour au plus 7 UT (fin ≤ 16).

À 10, l'entrée-sortie de P1 se termine et P1 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 2 UT (fin en 12), et P4 passe en tête de la file du processeur (en 1). L'entrée sortie de P3 commence pour 3 Ut (fin en 13).

À 12, P1 se bloque en attente du disque, mais étant prioritaire par rapport à P2, il passe en tête de file disque (en 1) et repousse P2 en 2. Notons que, bien évidemment, il n'y a pas préemption de l'entrée-sortie en cours qui doit aller à son terme. P4 devient actif pour au plus 6 UT (fin ≤ 18).

À 13,l'entrée-sortie de P3 se termine et P3 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour au plus 2 UT (fin ≤ 15), et P4 passe en tête de la file processeur. L'entrée-sortie de P1 commence pour 1 UT (fin en 14).

À 14,l'entrée-sortie de P1 se termine et P1 devient prêt, mais étant de priorité supérieure à celle de P3, devient actif pour 1 UT (fin en 15), et P3 passe en tête de la file processeur, repoussant P4 en 2. L'entrée-sortie de P2 commence pour 2 UT (fin en 16).

À 15, P1 se termine et P3 redevient actif pour au plus 1 UT (fin en 16), et P4 passe en tête de la file processeur.

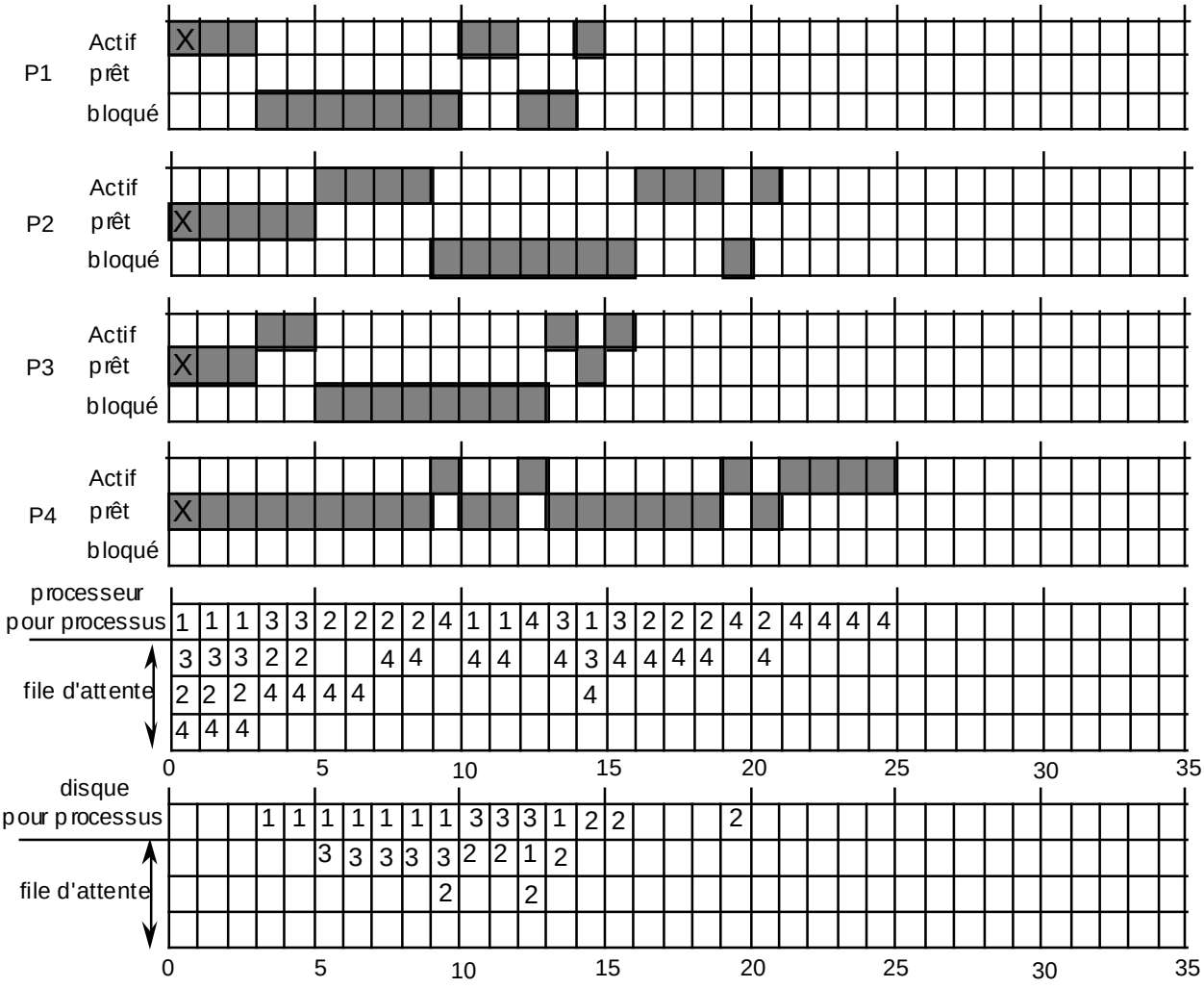
À 16, P3 se termine, l'entrée-sortie de P2 se termine et P2 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 3 UT (fin en 19).

À 19, P2 accède au disque pour 1 Ut (fin en 20) et P4 devient actif pour au plus 5 UT (fin ≤ 24).

À 20, l'entrée-sortie de P2 se termine et P2 devient prêt, mais étant de priorité supérieure à celle de P4, devient actif pour 1 UT (fin en 21), et P4 passe en tête de la file processeur.

À 21, P2 se termine et P4 devient actif pour 4 UT (fin en 25).

À 25, P4 se termine et il n'y a plus de processus.



Le temps de réponse de P1 est de 15, celui de P2 est de 21, celui de P3 est de 16 et celui de P4 est de 25. Le total est 77, soit une moyenne de 19,25 UT.

4.3. Analyse d'un fichier comptabilité

Nous nous plaçons dans un contexte volontairement restreint, pour simplifier l'exercice. Le système est monoprocesseur, et doté d'un seul disque à temps d'accès moyen de 20 ms. Il fonctionne en multiprogrammation, avec 1 Mo disponible pour les processus utilisateurs et uniquement en traitement par lot (les seules ressources prises en compte sont le disque et le processeur).

Il effectue différentes mesures sur les processus, à des fins de comptabilité. Ces mesures sont conservées dans un fichier. Chaque fois qu'un processus utilisateur est détruit, un enregistrement est ajouté au bout du fichier avec les informations décrites ci-dessous (tous les entiers sont supposés être sur 32 bits). *Note : certaines de ces informations ne sont pas utiles dans l'exercice, mais doivent se trouver dans une comptabilité; il ne vous est pas demandé de justifier ce point.*

NU nom de l'utilisateur (identifié au login).

IDP identité du processus : entier attribué au processus par le système lors de sa création.

IC instant de la création : entier représentant le nombre de millisecondes écoulées depuis le début d'activité (voir plus loin).

DV durée de vie du processus : entier représentant le nombre de millisecondes écoulées entre sa création et sa destruction.

MU taille mémoire utilisée : entier représentant le nombre de Ko de l'espace de mémoire centrale alloué au processus. Cet espace est fixe pendant toute son exécution.

TU temps UC consommé : entier représentant le nombre de millisecondes pendant lesquelles le processeur a travaillé pour le compte du processus.

TB temps passé à l'état bloqué : entier représentant le nombre de millisecondes pendant lesquelles le processus a été bloqué en attente d'une ressource autre que le processeur, c'est-à-dire la fin de l'accès disque demandé.

NAS nombre d'appels systèmes effectués par le processus pour des opérations appartenant à une méthode d'accès.

NAD nombre d'accès disque effectués par le système pour le compte du processus.

NOT nombre d'octets transférés depuis ou vers le disque par le système pour le compte du processus.

Pour isoler sur le fichier les périodes d'activités effectives, lors de la création d'un processus utilisateur et s'il n'y en avait pas auparavant, le système écrit un enregistrement particulier qui marque le début d'une période d'activité :

NU a pour valeur "*DEB*"

IDP a pour valeur la date du jour, entier représentant le nombre de jours écoulés depuis le 1er Janvier 1900,

IC a pour valeur le nombre de millisecondes écoulées depuis 0 heure (début de la journée).

De plus, après avoir détruit un processus utilisateur et s'il n'y en a plus, le système écrit un enregistrement particulier qui marque la fin de la période d'activité :

NU a pour valeur "*FIN*",

DV a pour valeur la durée de la période d'activité.

Le tableau suivant est un exemple numérique d'un tel fichier pour une période d'activité. Il sera utilisé dans les applications numériques.

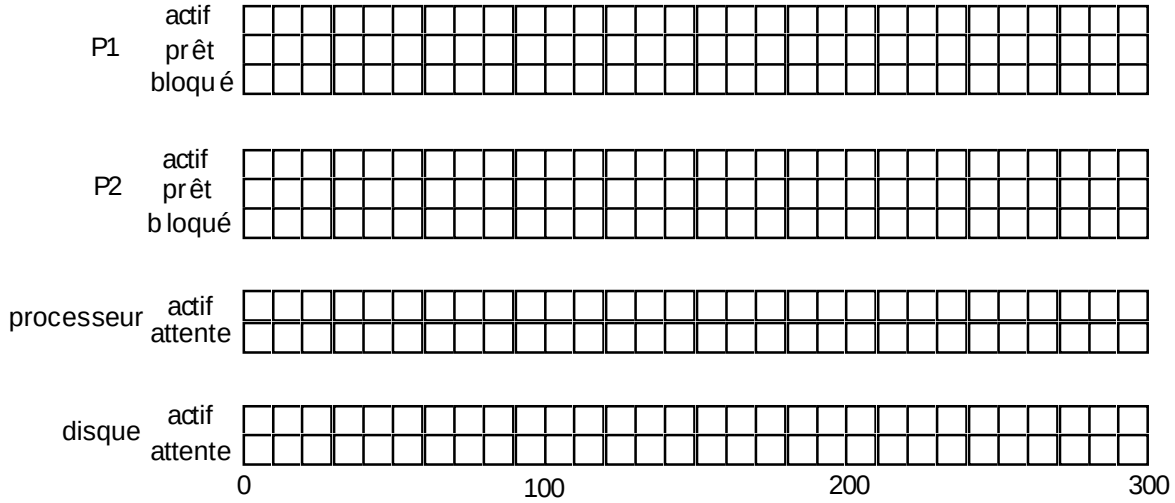
NU	IDP	IC	DV	MU	TU	TB	NAS	NAD	NOT
DEB	33945	360000	0	0	0	0	0	0	0
JEAN	23	10	180	60	40	60	12	2	2048
JEAN	22	0	1820	100	44	990	250	25	51200
PAUL	26	2660	2210	100	50	1550	620	31	31744
PAUL	27	5610	1510	150	40	1050	105	21	21504
JEAN	25	100	7760	400	4920	2180	3180	106	217088
JEAN	24	30	8560	300	2010	3550	2016	168	172032
FIN	0	0	8590	0	0	0	0	0	0

A- Expliquez les différents états d'un processus, et ce qui provoque les transitions entre ces états.

B- On considère deux processus, dont le comportement s'ils étaient seuls serait le suivant :

```
P1   calcul pendant 10 ms
     pour i de 1 à 5 faire
          lecture disque de 1 Ko (20 ms)
          calcul pendant 10 ms
     fait
P2   calcul pendant 10 ms
     lecture disque de 1 Ko (20 ms)
     calcul pendant 10 ms
     lecture disque de 1 Ko (20 ms)
     calcul pendant 30 ms
     lecture disque de 1 Ko (20 ms)
     calcul pendant 10 ms
     lecture disque de 1 Ko (20 ms)
     calcul pendant 40 ms
```

B.1- Il n'y a aucun processus utilisateur. On lance P1, puis on lance P2 10 ms après. Donner la chronologie des événements sur le graphique suivant (10 ms par carreau), en la justifiant.



B.2- En déduire les enregistrements de comptabilité produits par cette période d'activité. Certains champs ne peuvent pas être déterminés. Expliquez pourquoi.

C- Comment peut-on calculer le taux d'activité du processeur pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

D- Comment peut-on calculer le temps moyen d'un accès disque vu des processus? Qu'est-ce qui justifie la différence avec le temps moyen annoncé en début d'exercice? Comment peut-on estimer le taux d'activité du disque pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

E- Comment peut-on calculer le débit moyen du disque pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

F- Comment peut-on calculer le degré de multiprogrammation moyen pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

G- Comment peut-on calculer le temps moyen passé par un processus dans l'état prêt pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

H- Comment peut-on définir et calculer le taux moyen d'occupation mémoire pour une période d'activité? Vous justifierez votre raisonnement. Calculez-le pour l'exemple numérique donné dans le tableau ci-dessus. Commentez.

I- Expliquez en quelques lignes la différence entre les mesures NAS et NAD : à quoi font-elles références?

Solution de l'exercice 4.3

4.3.1. Question A.

Un processus peut être dans trois états :

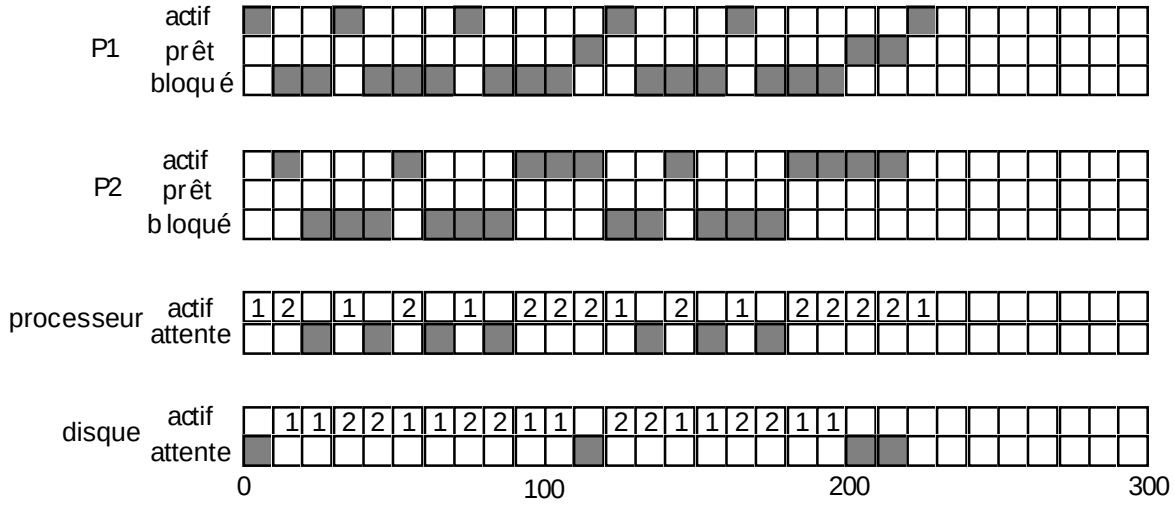
- *actif*, le processeur travaille pour lui,
- *prêt*, il dispose de toutes les ressources dont il a besoin, à l'exclusion du processeur,
- *bloqué*, il lui manque une ressource autre que le processeur pour s'exécuter.

Les transitions actif-prêt sont effectuées par l'allocateur du processeur. La transition actif vers bloqué est la conséquence d'une demande de ressource non disponible. La transition bloqué vers actif est la conséquence de l'allocation des ressources demandées (à l'exclusion du processeur).

4.3.2. Question B.

4.3.2.1. Question B.1.

Les deux processus sont en compétition pour le processeur et pour les accès au disque. Le processeur ne peut travailler que pour un seul processus à la fois, de même que le disque. La chronologie peut se représenter par le graphe suivant :



Au lancement du processus P1, celui-ci est dans l'état actif. Après 10 ms, c'est-à-dire au temps 10 ms, le processeur lance une lecture disque pour le compte de P1, ce qui le bloque. Le disque étant inoccupé, l'opération est lancée immédiatement. Le processus P2 est lancé à ce moment, et devient donc actif. Après 10 ms, c'est-à-dire au temps 20 ms, le processeur doit lancer une opération disque. Comme le disque est occupé, le processus est bloqué en attente du disque. A ce moment, les deux processus sont bloqués, et il n'y pas de processus prêt; le processeur devient libre. Au temps 30 ms, l'opération disque pour le compte de P1 est terminée, ce qui permet le lancement de celle pour le compte de P2, et le déblocage de P1 qui redevient actif. Au temps 40 ms, le processeur doit lancer une opération disque pour le compte de P1. Comme le disque est occupé, cette opération est mise en attente et le processus P1 est bloqué, etc... La suite de la chronologie consiste à suivre l'évolution des processus, en s'assurant que la ressource disque est utilisée par au plus un processus à la fois.

4.3.2.2. Question B.2.

En suivant la chronologie, on constate que P1 commence son exécution au temps 0 et la termine au temps 230 ms. Sa une durée de vie est donc de 230 ms. Par ailleurs, le cumul des durées dans l'état bloqué est de 140 ms. De la même façon, on voit que P2 a une durée de vie de 210 ms et est bloqué pendant 110 ms. Comme P2 se termine avant P1, l'enregistrement concernant P2 précède celui concernant P1. On ne connaît pas l'identité de l'usager qui a lancé les processus, ni l'identité donné à chaque processus par le système. Cependant, on peut penser que le système attribue cette identité en ordre croissant; nous supposerons que P1 reçoit le numéro 1 et P2 reçoit le numéro 2. De plus, on ne connaît pas la taille mémoire allouée à chaque processus ni le nombre d'appels système pour des opérations appartenant aux méthodes d'accès.

NU	IDP	IC	DV	MU	TU	TB	NES	NTD	NOT
DEB	?	?	0	0	0	0	0	0	0
?	2	10	210	?	100	110	?	4	4096
?	1	0	230	?	60	140	?	5	5120
FIN	0	0	230	0	0	0	0	0	0

4.3.3. Question C.

Le taux d'activité processeur est le rapport entre le temps pendant lequel le processeur a travaillé pour le compte des processus et le temps total écoulé. On peut donc écrire :

$$tap = \frac{\sum TU}{FIN.DV}$$

Dans notre application numérique, cela nous donne 82,7%. Notons qu'il s'agit d'un taux tout à fait raisonnable.

4.3.4. Question D.

Puisque la seule ressource prise en compte est le disque, l'état bloqué correspond à l'attente de la fin de l'accès disque demandé. Le temps moyen d'un accès disque vu des processus est donc le cumul des temps passés dans l'état bloqué divisé par le nombre d'accès disque demandés. Il peut se déterminer par la formule suivante :

$$tmad = \frac{\sum TB}{\sum NAD}$$

Pour notre application numérique, un accès disque dure en moyenne 26,6 ms. A priori, les caractéristiques du disque annoncent un temps moyen pour l'accès disque lui-même de 20 ms. L'exemple traité dans la question B montre qu'un processus peut être bloqué parce qu'un autre utilise le disque, ce qui explique la différence. On constate donc que les processus attendent en moyenne 6,6 ms pour la libération du disque et 20 ms d'attente de l'opération elle-même. On peut constater d'ailleurs que le cumul du temps de blocage des processus (9380) dépasse le temps total (8590).

Le taux d'activité du disque est le rapport entre le temps pendant lequel le disque a travaillé pour le compte des processus et le temps total écoulé. Le temps pendant lequel le disque a travaillé pour le compte des processus n'est pas donné explicitement. Cependant on connaît le nombre total d'accès disque ainsi que le temps moyen effectif d'un accès disque, on peut donc estimer ce taux d'activité en supposant que les accès ont duré en moyenne 20 ms. La formule est donc :

$$tad = \frac{20 * \sum NTD}{FIN.DV}$$

Dans notre application numérique, cela nous donne 82,2%. Notons qu'il s'agit d'un taux relativement important, qui explique l'allongement du temps moyen vu par les processus.

4.3.5. Question E.

Le débit moyen du disque est le nombre total d'octets transférés par unité de temps. Il peut donc être calculé par la formule suivante :

$$dd = \frac{\sum NOT}{FIN.DV}$$

Pour notre application numérique, cela donne 56,3 Ko par seconde. Ce n'est pas énorme. Notons que 484 Ko sont transférés en 353 accès, ce qui veut dire qu'il y a transfert de 1,4 Ko par accès. On pourrait augmenter le débit en augmentant le nombre d'octets transférés à chaque accès.

4.3.6. Question F.

Le degré de multiprogrammation moyen se définit comme le nombre moyen de processus présents en mémoire en même temps. Ceci peut dont s'obtenir en divisant le cumul des durées de vie des processus par la durée totale de l'activité. On obtient la formule suivante :

$$dm = \frac{\sum DV}{FIN.DV}$$

Pour notre application numérique, cela donne 2,6. Constatons que les processus identifiés 24 et 25 ont une durée de vie qui couvre presque toute la période d'activité. Le processus 23 a une durée de vie très brève. Les processus 22, 26 et 27 ont des périodes de vie disjointes. Par ailleurs, constatons aussi que ce faible degré de multiprogrammation n'a pas empêché d'avoir un fort taux d'activité processeur. Ceci est dû au fait que le processus 25 a un taux effectif d'attente d'entrées sorties (s'il était seul) qui peut s'évaluer à 106*20/(106*20+4920), soit 30%. Le processus 24 a un taux effectif d'attente d'entrées-sorties de 63%.

4.3.7. Question G.

Le temps moyen passé par un processus dans l'état prêt représente le ralentissement qu'il subit à cause de la compétition sur l'allocation du processeur. L'évaluation du temps passé à l'état prêt pour un processus est la différence entre sa durée de vie et le cumul des temps à l'état actif et bloqué. On a donc TP = DV - TU - TB. Le tableau suivant donne ce temps pour chacun des 6 processus. :

IDP	23	22	26	27	25	24
TP	80	786	610	420	660	3000

Notons que pour 22, 26 et 27, ce temps est beaucoup plus grand que le temps processeur lui-même.

Pour calculer le temps moyen passé par les processus dans l'état prêt, il suffit de cumuler les temps TP et de diviser par le nombre de processus. Dans notre exemple, cela donne 926 ms. L'amélioration du taux d'activité processeur demande la présence d'un processus prêt lorsque le processus actif se bloque, ce qui implique un allongement du temps passé dans l'état prêt.

4.3.8. Question H.

A un instant donné, le taux d'occupation mémoire est le rapport entre la taille de l'espace alloué aux processus et la taille totale de l'espace mémoire. Le taux moyen est la valeur moyenne de ce taux. En d'autres termes, il s'agit de faire la moyenne du taux instantané sur l'ensemble de la période d'activité.

On peut aussi le présenter autrement. On peut considérer que la mémoire est quelque chose qui se loue, et dont le prix de location est proportionnel à la quantité et à la durée de location. Dans ces conditions, le taux moyen d'occupation est le rapport entre le revenu effectif de la location et le revenu maximum que l'on peut obtenir. Il s'agit donc de cumuler les produits entre la quantité louée par le temps de location, et de diviser le tout par le produit entre la quantité maximum à louer par le temps maximum de location.

Quelle que soit l'interprétation donnée ci-dessus, en notant MAXMEM la quantité maximum de mémoire allouable aux processus utilisateurs, le taux moyen d'occupation mémoire peut se calculer par la formule suivante :

$$tom = \frac{\sum MU * DV}{MAXMEM * FIN.DV}$$

Pour notre application numérique, cela donne 71,8%. Remarquons que ce n'est pas un taux très fort. D'ailleurs, le taux instantané ne dépasse jamais 84%. Il est probable que le degré de multiprogrammation pourrait sans doute être encore augmenté. Notons que le taux d'activité processeur étant déjà important, il ne changerait sans doute pas beaucoup si plus de processus étaient en mémoire.

4.3.9. Question I.

La quantité NAS représente le nombre d'appels système appartenant à une méthode d'accès. Ces appels systèmes correspondent à des enregistrements logiques. Les différentes organisation étudiées dans le cours, montrent qu'il y a lieu de distinguer les enregistrements logiques qui sont manipulés par le programme et les enregistrements physiques qui correspondent aux transferts avec le disque. Un accès à un enregistrement logique ne donne pas forcément lieu à un accès disque, le système devant conserver les enregistrements physiques en mémoire centrale pour assurer le groupage et le dégroupage des enregistrements logiques.

Commentaires (hors sujet)

Au fur et à mesure des réponses aux questions, nous avons fait des commentaires sur les résultats de l'application numérique. Nous en avons conclu que le processeur avait un bon taux d'activité, le disque avait un taux d'activité plutôt élevé et un débit pas très important. Enfin le taux d'occupation mémoire ne reflétait pas une mémoire saturée. Il semble donc bien que si la configuration devait être augmentée, ce serait plutôt en nombre de disques, de façon à diminuer les temps de blocage des processus. Par ailleurs le débit pourrait sans doute être amélioré en essayant d'obtenir des transferts élémentaires de plus grosse taille.

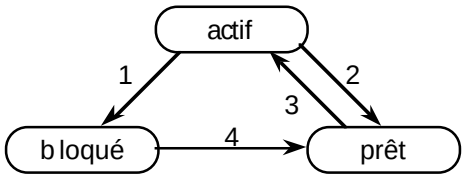
Enonçons quelques indicateurs généraux :

- Une configuration équilibrée et globalement adaptée devrait avoir un taux d'activité processeur et un taux d'occupation mémoire élevés alors que le taux d'activité disque reste moyen.
- Une faiblesse en taille mémoire induit évidemment un fort taux moyen d'occupation mémoire, mais surtout conduira à une faiblesse du degré de multiprogrammation entraînant un taux d'activité processeur faible, par manque de processus prêts, tout en ayant un taux d'activité disque moyen, montrant l'absence de saturation disque.
- Une saturation des disques en terme de performances se concrétisera, en l'absence de mémoire virtuelle, par un fort taux d'activité disque et un taux d'activité processeur faible. Il faut alors augmenter le nombre d'unités et mieux répartir les données sur les disques de façon a diminuer cette saturation.
- Une faiblesse de puissance de processeur se concrétisera par un taux d'activité processeur fort alors que les taux d'activité disque et taux d'occupation mémoire sont faibles. Il ne sert à rien d'augmenter le degré de multi-programmation, puisque le processeur est déjà saturé.

4.4. Exécution de processus en multiprogrammation

On considère un système monoprocesseur dans lequel les processus partagent un disque comme seule ressource (autre que le processeur). Cette ressource n'est accessible qu'en accès exclusif et non requérable, c'est-à-dire qu'une commande disque lancée pour le compte d'un processus se termine normalement avant de pouvoir en lancer une autre. Un processus peut être en exécution, en attente d'entrée-sortie ou en attente du processeur.

A- Expliquer le schéma suivant représentant les états possibles d'un processus et les transitions entre ces états. Expliquer pourquoi certaines transitions ne sont pas possibles.

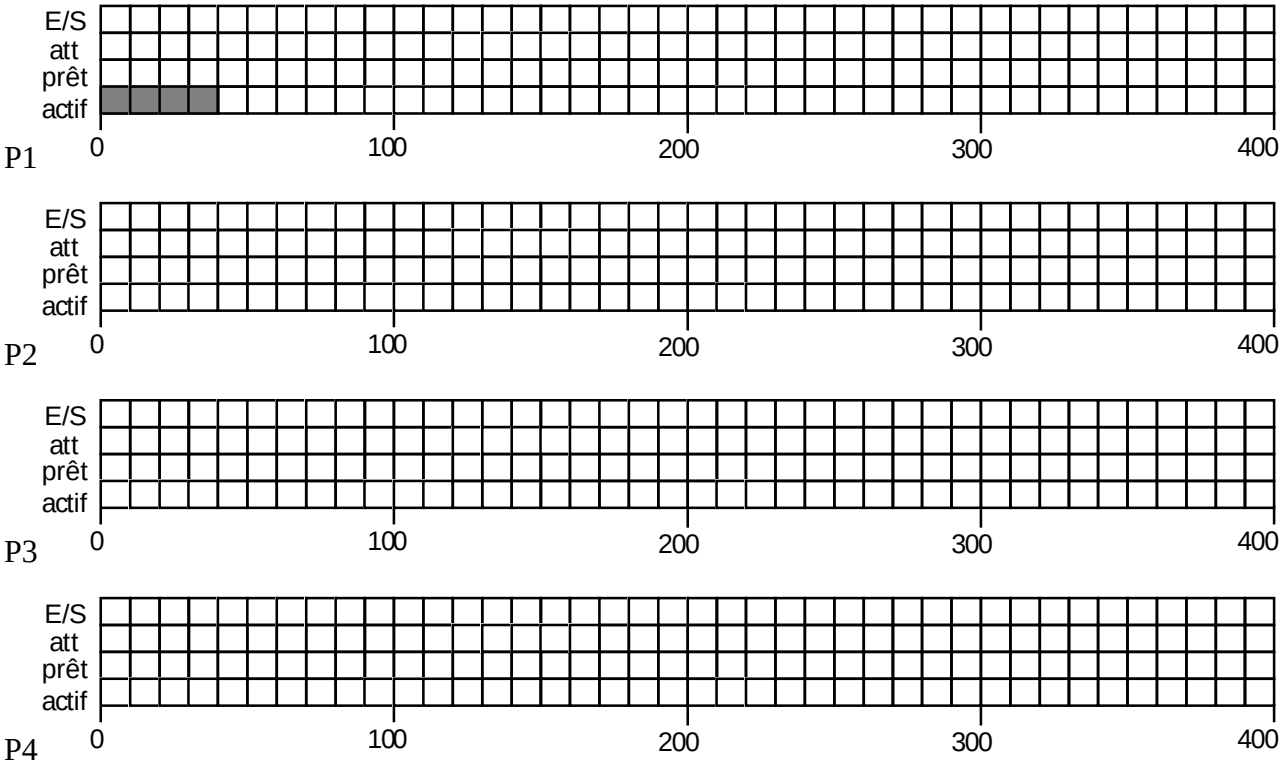


B- En fait l'état bloqué se divise en deux états : attente de la ressource disque et attente de la fin d'exécution de l'opération. Les demandes d'entrées-sorties sont gérées à l'ancienneté, et l'allocation du processeur est faite selon la priorité affectée au processus, et représentée par une valeur entière. Le processus prioritaire est celui qui a la plus grande valeur et si deux processus ont même priorité, c'est le plus ancien dans la file d'attente des processus prêts.

Nous considérons les 4 processus dont le comportement est le suivant (la priorité au démarrage est indiquée entre parenthèses) :

```
P1   (100)     Calcul pendant 40 ms
               Lecture disque pendant 50 ms
               Calcul pendant 30 ms
               Lecture disque pendant 40 ms
               Calcul pendant 20 ms
P2   (99) Calcul pendant 30 ms
               Lecture disque pendant 80 ms
               Calcul pendant 80 ms
               Lecture disque pendant 20 ms
               Calcul pendant 10 ms
P3   (98) Calcul pendant 40 ms
               Lecture disque pendant 40 ms
               Calcul pendant 10 ms
P4   (97) Calcul pendant 80 ms
```

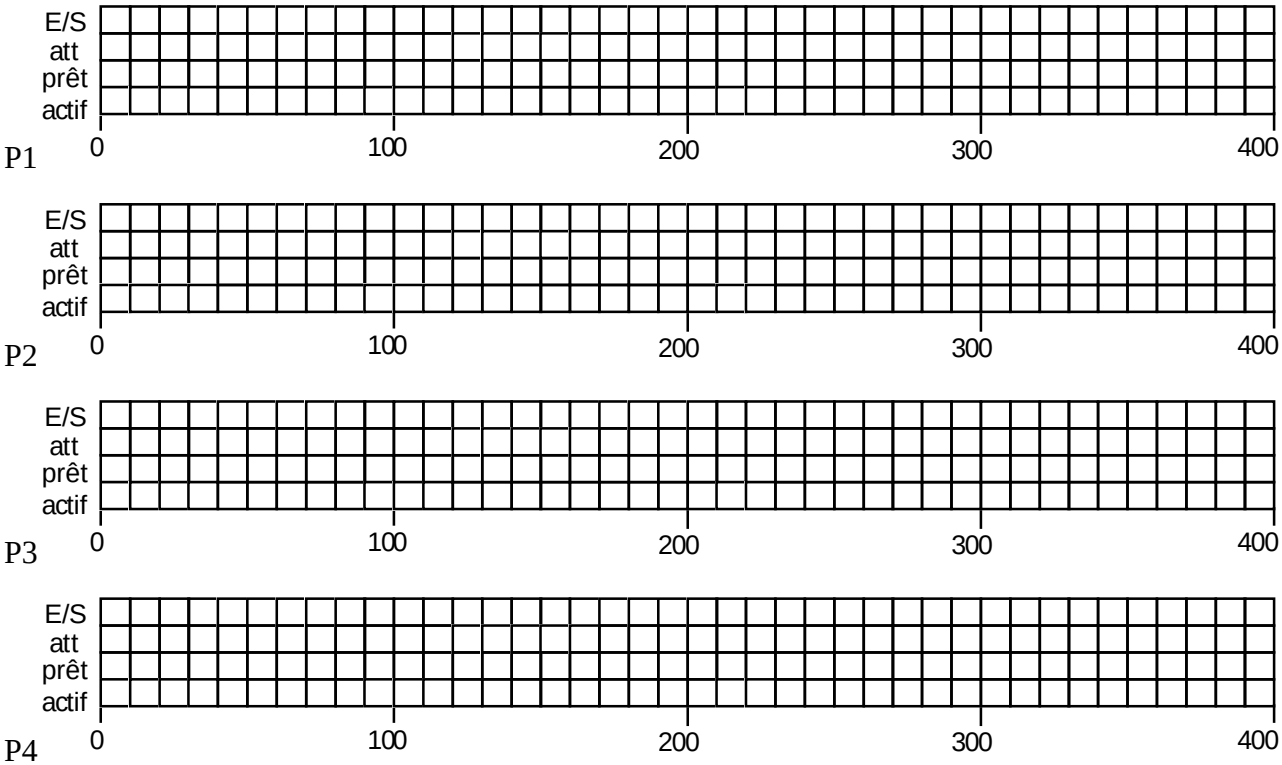
B.1- Les 4 processus sont lancés en même temps et gardent leur priorité initiale pendant toute leur exécution. Établir le chronogramme des 4 processus sur le diagramme suivant. Vous noircirez les cases correspondant à l'état du processus, comme cela a été fait pour le début du processus P1, à titre d'exemple.



B.2- Les 4 processus sont lancés en même temps, mais leur priorité est variable. Chaque fois qu'un processus quelconque quitte l'état bloqué, on recalcule la priorité de chaque processus selon la formule suivante :

Priorité Nouvelle = Priorité Initiale - (Temps processeur utilisé) / 10

Établir le chronogramme des 4 processus sur le diagramme suivant.



C- Donner le temps total de l'exécution de ces 4 processus dans les trois cas suivants :

C1 L'activation des 4 processus est demandée à l'instant initial et ils s'exécutent en monoprogrammation dans l'ordre P1, P2, P3 puis P4,

C2 Gestion de B.1,

C3 Gestion de B.2.

D- Comparer les temps de réponse moyens dans les trois cas C1, C2, C3

E- Donner le taux d'utilisation du processeur dans les 3 cas C1, C2 et C3

Solution de l'exercice 4.4

4.4.1. Question A.

Un processus peut se trouver dans différents états, suivant qu'il dispose de tout ou partie des ressources dont il a besoin pour s'exécuter. On distingue la ressource processeur de l'ensemble des autres ressources. En effet, s'il manque une ressource autre que le processeur, celui-ci ne peut faire évoluer le processus et il est donc inutile que cette ressource chère lui soit allouée. Lorsqu'il manque à un processus une ressource autre que le processeur, il est dans l'état bloqué. Lorsqu'un processus a toutes ses ressources à l'exception du processeur, il est dans l'état prêt. Enfin lorsqu'un processus a toutes ses ressources, y compris le processeur, il est dans l'état actif. L'allocation du processeur consiste à choisir un processus dans l'état prêt, et à lui allouer le processeur, le faisant passer dans l'état actif (transition 3). Un processus actif peut perdre le processeur, et repasser dans l'état prêt lorsque le système désire allouer le processeur à un autre processus (transition 2). Lorsqu'un processus actif demande une ressource qui n'est pas disponible, il passe dans l'état bloqué, et le processeur lui est retiré (transition 1). Lorsque la ressource demandée par un processus

devient disponible, elle peut lui être allouée; le processus a alors toutes ses ressources sauf la ressource processeur et passe donc dans l'état prêt (transition 4).

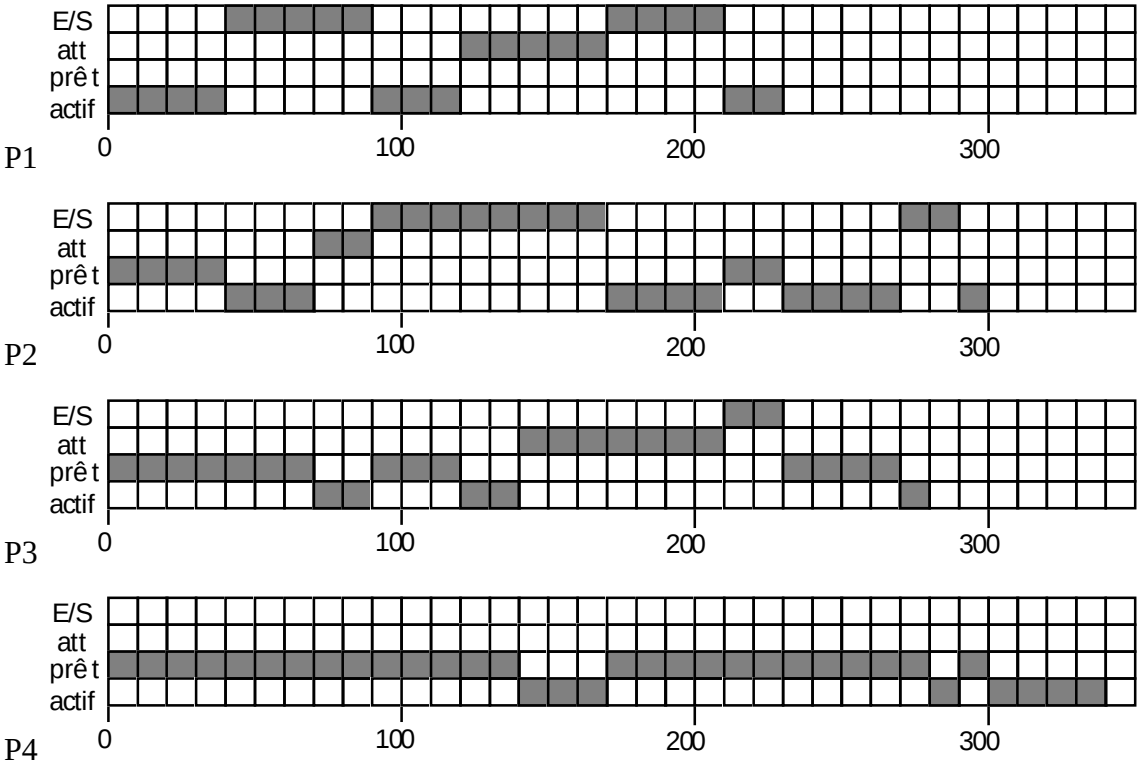
Le processus est dans l'état bloqué lorsqu'il n'a pas toutes ses ressources, à l'exception de la ressource processeur. Pour passer dans cet état, il faut qu'il lui manque une ressource, soit parce qu'on la lui a retiré, ce qui n'est pas prévu ici, soit parce qu'il exprime un nouveau besoin, ce qui implique qu'il dispose du processeur pour exprimer ce besoin, c'est-à-dire qu'il soit actif. En conséquence, la seule transition vers l'état bloqué ne peut venir que de l'état actif. Par ailleurs, un processus quitte l'état bloqué lorsqu'il a toutes ses ressources pour s'exécuter, à l'exception de la ressource processeur, ce qui interdit la transition directe de bloqué à actif.

4.4.2. Question B.

4.4.2.1. Question B.1.

Le chronogramme d'exécution des 4 processus est donné ci-après. Les 4 processus sont initialement prêts. P1 ayant la priorité la plus grande est le premier processus actif jusqu'au temps 40, où son entrée-sortie est lancée, pour une durée de 50, c'est-à-dire jusqu'en 90. P2 devient alors actif. Au temps 70, P2 est bloqué en attente du disque et P3 devient actif.

Au temps 90, la demande d'entrée-sortie de P1 se termine et celui-ci redevient prêt. La priorité de P1 étant maximale, P1 redevient donc actif. En même temps, le disque étant libre, l'entrée-sortie de P2 est lancée, pour une durée de 80, c'est-à-dire jusqu'en 170. Au temps 120, P1 se bloque en attente du disque. P3, de priorité maximale, redevient actif jusqu'au temps 140, où il se bloque en attente du disque (derrière P1). P4 étant le seul processus prêt devient actif.



Au temps 170, l'entrée-sortie de P2 se termine, permettant à ce processus de redevenir prêt, en même temps que l'entrée-sortie suivante de la file, c'est-à-dire celle de P1, est lancée, pour une durée de 40, c'est-à-dire jusqu'en 210. La priorité de P2 étant maximale, P2 devient actif.

Au temps 210, l'entrée-sortie de P1 se termine permettant à ce processus de redevenir prêt, en même temps que l'entrée-sortie de P3 est lancée, pour une durée de 20, c'est-à-dire jusqu'en 230. La priorité de P1 étant maximale, P1 redevient actif jusqu'à son achèvement au temps 230.

Au temps 230, l'entrée-sortie de P3 se termine permettant à ce processus de redevenir prêt. La priorité de P2 étant maximale, P2 redevient actif jusqu'au temps 270, où son entrée-sortie est lancée,

pour une durée de 20, c'est-à-dire jusqu'en 290. P3 devient actif jusqu'à son achèvement en 280. P4 étant seul processus prêt devient actif.

Au temps 290, l'entrée-sortie de P2 se termine permettant à ce processus de redevenir prêt. La priorité de P2 étant maximale, P2 redevient actif jusqu'à son achèvement en 300. P4, seul processus restant, redevient actif jusqu'à son achèvement en 340.

4.4.2.2. Question B.2.

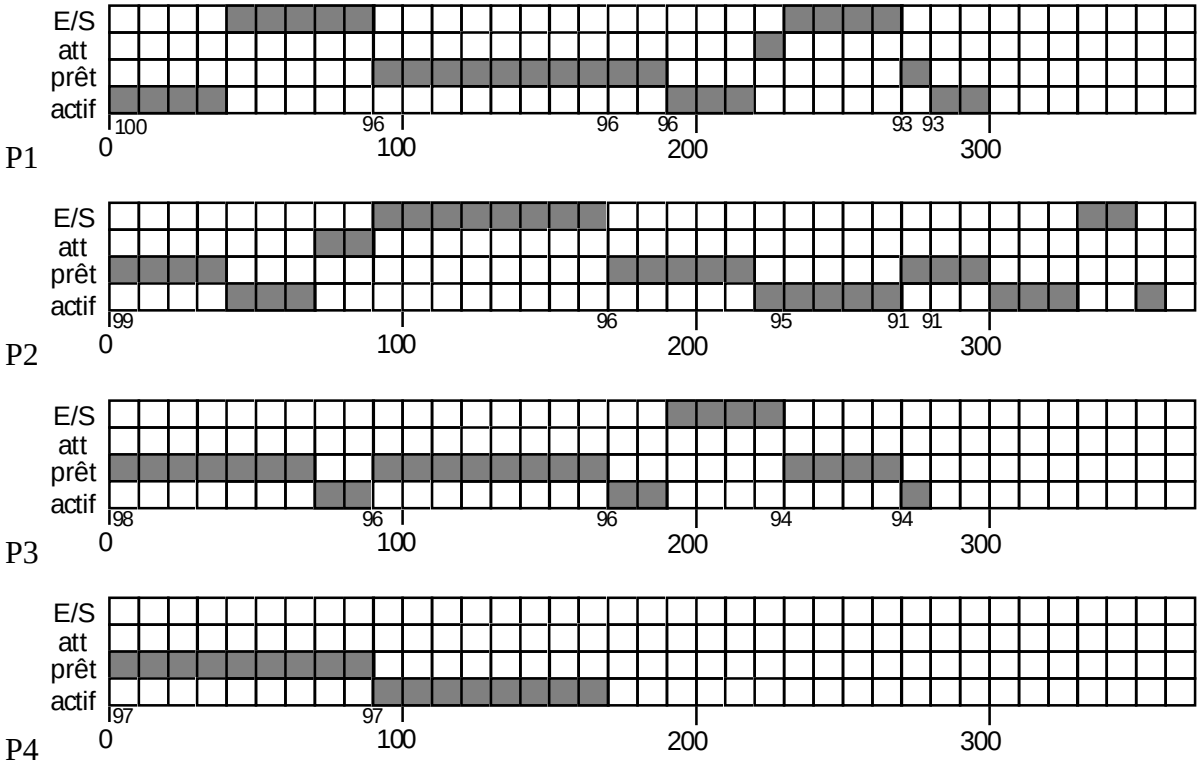
Le chronogramme d'exécution des 4 processus est donné ci-après. Le début est similaire au cas précédent. Les 4 processus sont initialement prêts. P1 ayant la priorité la plus grande est le premier processus actif jusqu'au temps 40, où son entrée-sortie est lancée, pour une durée de 50, c'est-à-dire jusqu'en 90. P2 devient alors actif. Au temps 70, P2 est bloqué en attente du disque et P3 devient actif.

Au temps 90, la demande d'entrée-sortie de P1 se termine et celui-ci redevient prêt. La priorité de P1 est alors 100-4=96, celle de P3 est 98-2=96 et celle de P4 est 97. P4 devient donc actif. En même temps, le disque étant libre, l'entrée-sortie de P2 est lancée, pour une durée de 80, c'est-à-dire jusqu'en 170.

Au temps 170, d'une part P4 est achevé, d'autre part l'entrée-sortie de P2 se termine, et P2 redevient prêt. La priorité de P2 est alors 99-3=96. Les 3 processus P1, P2, P3 ont la même priorité, mais P3 est le plus ancien de la file; il devient actif, jusqu'au temps 190, où son entrée-sortie est lancée, pour une durée de 40, c'est-à-dire jusqu'en 230. Les deux processus prêts ayant même priorité (96) le plus ancien de la file, c'est-à-dire P1, devient actif. Au temps 220, P1 se bloque en attente du disque et P2, seul processus prêt, devient actif.

Au temps 230, l'entrée-sortie de P3 se termine, permettant le lancement de celle de P1, pour une durée de 40, c'est-à-dire jusqu'en 270. P3 redevient actif avec la priorité 98-4=94. La priorité de P2 à ce moment est 99-4=95. Le processus P2 reste actif.

Au temps 270, l'entrée-sortie de P1 se termine et P1 redevient actif. La priorité de P1 est alors 100-7=93, celle de P2 est 99-8=91 et celle de P3 est 98-4=94. P3 devient actif jusqu'à son achèvement en 280. P1 étant alors le plus prioritaire devient actif jusqu'à son achèvement au temps 300. P2 restant seul peut achever ses activités.



4.4.3. Question C.

En monoprogrammation, le temps total est la somme des temps processeurs et entrées-sorties de chaque processus. Or P1 dure 180 ms, P2 dure 220 ms, P3 dure 90 ms et P4 dure 80 ms. Le temps total d'exécution dans le cas C1 est donc de 570 ms.

Dans le cas B1 où l'allocation processeur est à priorité fixe, on constate que le dernier processus à se terminer est P4, donnant une durée totale de 340 ms.

Dans le cas B2 où l'allocation processeur est à priorité variable, on constate que le dernier processus à se terminer est P2, donnant une durée totale de 360 ms.

4.4.4. Question D.

Le temps de réponse pour un processus est le temps qui sépare le moment où le processus est fourni au système du moment de la fin de son exécution. Dans nos hypothèses, les 4 processus sont toujours fournis à l'instant 0, mais ils se terminent plus ou moins tôt.

	C1	C2	C3
P1	180	230	300
P2	400	300	360
P3	490	280	280
P4	570	340	170
moyenne	410	288	278

Constatons sur ce tableau que le temps de réponse moyen est plus fort dans le cas C1 que dans les autres cas. Notons que le temps de réponse pour P4 est nettement amélioré dans la gestion à priorité variable par rapport à la gestion à priorité fixe, au détriment des temps de réponse de P1 et P2.

4.4.5. Question E.

Le taux d'utilisation processeur est le rapport entre la durée d'utilisation effective du processeur par l'un des processus et la durée totale d'exécution. Le temps total d'utilisation du processeur étant de 340 ms, on voit donc que le cas C1 donne un taux d'occupation du processeur de 60%, le cas C2 un taux de 100% et le cas C3 un taux de 94%.

En analysant les chronogrammes, on constate que le cas C2 permet d'obtenir un taux d'occupation de 100% du processeur parce que le processus P4 qui ne fait aucune entrée-sortie a la priorité la plus faible. Il sert donc de "bouche-trou". L'inconvénient est immédiat : c'est l'allongement du temps de réponse pour ce processus.

4.5. Création et synchronisation de processus

Java, le nouveau langage à la mode, propose un mécanisme de création de processus et des mécanismes de synchronisation entre les processus. Nous ne nous attachons pas à la syntaxe de ce langage, mais étudions ici les mécanismes eux-mêmes, dans une syntaxe adaptée. Les réponses n'ont pas besoin d'être longues, mais doivent être précises.

A– Une première structure permet de déclarer un type de processus, sans créer le processus lui-même. Tous les processus créés de ce type ont alors accès à toutes les variables déclarées dans le texte avant la déclaration du type de processus (ces variables sont partagées).

```
processus Nom_Du_Type_De_Processus est
     -- déclarations de variables locales
début
     -- instructions du processus
fin Nom_Du_Type_De_Processus;
```

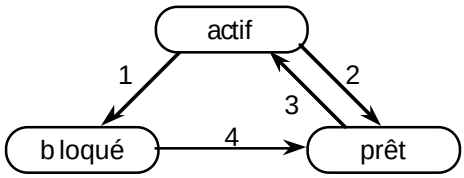
La création d'un processus s'obtient alors comme une simple déclaration de variable. La création ne lance pas le processus, qui reste donc inactif.

```
Nom_De_Type_De_Processus: Mon_Processus;
```

Quatre instructions permettent d'agir sur des processus créés :

```
démarrer Mon_Processus; -- lance effectivement le processus
arrêter Mon_Processus; -- arrête définitivement le processus
Suspendre Mon_Processus; -- le processus ne participe plus à la
                         -- compétition pour l'accès au processeur
Reprendre Mon_Processus; -- le processus participe de nouveau à
                       -- la compétition pour l'accès au processeur
```

Rappelons le graphe d'état des processus.



A.1– Pour chacune des quatre instructions ci-dessus, comment pourrait-on décrire son effet sur l'état du processus? Quelles difficultés cela présente-t-il?

A.2– On ajoute deux états supplémentaires à ce graphe: « bloqué-suspendu » et « suspendu ». Pourquoi sont-ils utiles? Décrire les transitions supplémentaires en précisant les instructions qui les provoquent.

A.3– Pour chacune des quatre instructions, indiquer si elles peuvent être exécutées par le processus lui-même, ou si elles doivent être exécutées par un autre processus que celui sur lequel elles agissent.

B– L'exclusion mutuelle d'accès à un objet est obtenu au moyen d'un nouveau mode de passage de paramètre dans la déclaration d'une procédure :

```
procédure Ma_Procédure(L_Objet: synchronisé Type_D_Objet);
```

L'exécutif associe à chaque objet un verrou. Le compilateur encadre le code de `Ma_Procédure` par les opérations d'obtention et de libération du verrou de `L_Objet`. Pour une procédure, il ne peut y avoir au plus qu'un paramètre ayant ce mode de passage : nous dirons que `Ma_Procédure` est en exclusion mutuelle sur `L_Objet`.

Quelle utilité voyez-vous à ce mécanisme? Quels sont les avantages et inconvénients de cette forme syntaxique d'exclusion mutuelle?

C– A chaque objet, en plus du verrou, on associe une file de processus. Deux instructions permettent de manipuler cette file, à l'intérieur d'une procédure en exclusion mutuelle sur cet objet :

```
attendre; -- l'exécutif libère le verrou sur l'objet, bloque le
          -- processus, et le met dans la file associée à l'objet
notifier; -- s'il y a un processus dans la file, le premier est
          -- extrait de la file, et mis en attente du verrou
```

On considère l'exemple suivant.

```
type Controle_Caddy est article
  Nombre: Entier;
fin article;

procédure Obtenir_Caddy (CC: synchronisé Controle_Caddy) est
début si CC.Nombre = 0 alors attendre; finsi;
       CC.Nombre := CC.Nombre - 1;
fin Obtenir_Caddy;

procédure Rendre_Caddy (CC: synchronisé Controle_Caddy) est
début CC.Nombre := CC.Nombre + 1;
       notifier;
fin Rendre_Caddy;

Les_Caddies : Controle_Caddy := {3};
```

```
processus Client est
début Obtenir_Caddy (Les_Caddies);
       -- Lecture disque -> blocage
       Rendre_Caddy (Les_Caddies);
fin Lecteur;
```

C.1– En supposant que les temps de traitement processeur sont très petit devant le temps d'attente disque, donner la suite des états des processus, de l'objet et de sa file, dans la configuration suivante :

```
C1, C2 : Client; -- les démarrages de processus sont tous effectués avant
                 -- que l'un des processus concerné ne devienne actif
démarrer C1;
démarrer C2;
```

2 pt. C.2– En supposant que les temps de traitement processeur sont très petit devant le temps d'attente disque, donner la suite des états des processus, des composants de l'objet et de sa file, dans la configuration suivante :

```
C1, C2, C3, C4, C5 : Client; -- les démarrages de processus sont tous effectués
                        -- avant que l'un des processus concerné ne devienne actif
démarrer C1;
démarrer C2;
démarrer C3;
démarrer C4;
démarrer C5;
```

Solution de l'exercice 4.5

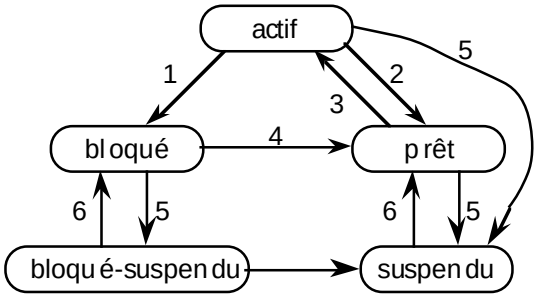
4.5.1. Question A

4.5.1.1. Question A.1

L'opération `démarrer` fait passer le processus dans l'état prêt. L'opération `arrêter` retire le processus des tables du système ; il n'existe plus. L'opération `suspendre` le fait passer dans l'état bloqué s'il n'y était déjà et l'opération `reprendre` le fait passer dans l'état prêt sauf s'il était auparavant dans l'état bloqué et qu'il lui manque d'autres ressources pour s'exécuter.

4.5.1.2. Question A.2

Le processus créé mais non lancé ou arrêté n'a pas à être représenté, car il n'existe pas pour le système. Par contre, un processus suspendu doit être connu, en particulier car il peut posséder des ressources, ou être en attente de ressources. Nous avons vu d'ailleurs, dans la question précédente que le résultat de l'opération `reprendre` devait remettre le processus soit dans l'état bloqué soit dans l'état prêt. Le graphe d'état peut être modifié comme suit.



Les transitions sont le résultat de l'opération `suspendre`, alors que les transitions 6 sont le résultat de l'opération `reprendre`.

4.5.1.3. Question A.3

Les opérations `démarrer` et `reprendre` ne peuvent être exécutées par le processus lui-même, puisque le processus ne peut être actif dans ce cas. Elles sont donc obligatoirement exécutées par un

autre. Par contre, `arrêter` et `suspendre` peuvent être exécutées par le processus lui-même, ou par un autre.

4.5.2. Question B

Le verrou va garantir l'exclusion mutuelle d'accès à `L_Objet`. Cette exclusion mutuelle est nécessaire lorsque deux processus cherchent à modifier en même temps un même objet. L'avantage de cette forme syntaxique est que le compilateur se charge de mettre les opérations de verrouillage et de déverrouillage là où il faut. On a alors une certaine garantie de respect de la règle du jeu. L'inconvénient est une certaine rigidité : le processus obtient l'exclusion mutuelle durant toute l'exécution de la procédure.

4.5.3. Question C

4.5.3.1. Question C.1

Initialement, les deux processus sont dans l'état prêt, C1 étant devant, et le nombre de caddy vaut 3. Le processus C1 devient donc actif. Il fait passer le nombre de caddy à 2, puis se bloque en attente de la fin de sa lecture disque. Le processus C2 devient donc actif, fait passer le nombre de caddy à 1, puis se bloque en attente de la fin de sa lecture disque ; notons que le disque est actuellement occupé par C1. Il n'y a plus de processus prêt, le processeur est inactif. A la fin de la lecture disque pour C1, il y aura lancement de la lecture disque pour C2, et passage à l'état prêt du processus C1. Le processus C1 refait passer le nombre de caddy à 2 et s'arrête. A la fin de la lecture disque pour C2, le processus C2 redevient prêt, donc actif et refait passer le nombre de caddy à 3. Remarquons que l'accès exclusif au nombre de caddy garantit que la valeur de ce nombre est correct en toutes circonstances.

C1	C2	nombre de caddy
prêt	prêt	3
actif		2
bloqué		
	actif	1
	bloqué	
...		
prêt		
actif		2
arrêté		
...		
	prêt	
	actif	3
	arrêté	

4.5.3.2. Question C.2

Initialement, les cinq processus sont dans l'état prêt, dans l'ordre de leur démarrage. Les trois premiers vont passer dans l'état actif, faire diminuer de 1 le nombre de caddy, et se bloquer en attente de leur lecture disque. Les deux derniers vont passer dans l'état actif, constater que le nombre de caddy est nul et entrer dans la file d'attente, ce qui les bloque. Si on suppose que les lectures disque se terminent dans l'ordre de leur lancement, C1 redeviendra prêt, fera passer le nombre de caddy à 1 le nombre de caddy, notifiera le changement, ce qui fera passer C4 dans l'état prêt, et s'arrêtera. C4 passera actif, remettra le nombre de caddy à 0 et se bloquera en attente du disque. Plus tard, C2 agira de même vis-à-vis de C5. Ensuite, successivement, C3, C4 et C5 repasseront prêt, incrémenteront le nombre de caddy et s'arrêteront.

C1	C2	C3	C4	C5	nombre de caddy	file
prêt	prêt	prêt	prêt	prêt	3	Ø
actif					2	
bloqué						
	actif				1	
	bloqué					
		actif			0	

```
                        bloqué
_____
                                  actif
                                  bloqué
C4
                                           actif
                                           bloqué                          C4,
C5
...
     prêt
_____
     actif
                                  prêt                    1
C5
     arrêté
_____
                                  actif
                                                          0
                                  bloqué
_____
...
               prêt
_____
               actif
                                           prêt           1                Ø

               arrêté
_____
                                           actif
                                                          0
                                           bloqué
_____
...
                        prêt
_____
                        actif
                                                          1
                        arrêté
_____
...
                                  prêt
_____
                                  actif
                                                          2
                                  arrêté
_____
...
                                  prêt
_____
                                  actif
                                                          3
                                  arrêté
_____
```

4.6. Synchronisation de lecteurs/rédacteurs

Le but de cet exercice est l'étude d'un problème courant de synchronisation de processus, communément appelé “lecteurs-rédacteurs”. Plusieurs processus, un par terminal, dotés de leur propre espace de mémoire centrale, ont un comportement identique, c'est-à-dire, exécutent le même programme. Celui-ci permet la consultation et la mise à jour d'un fichier aléatoire par numéro (ensemble d'enregistrements logiques auxquels on accède directement par leurs numéros), qui est relié au même fichier externe lors de l'ouverture. Il est schématisé sur la page suivante.

En plus des opérations de lecture et d'écriture mentionnées ci-dessus, le système fournit le type et les opérations suivantes:

- **type** t_mode: (exclusif, partagé);

- **procédure** ouvrir (**var** f: fichier; nom: chaîne; mode: t_mode) recherche le fichier externe de nom nom. S'il ne le trouve pas, le processus est arrêté, et détruit. Sinon, il regarde si un autre processus a ouvert ce fichier. Le processus demandeur est bloqué (mis en file d'attente) si le fichier était déjà ouvert en mode exclusif, ou s'il était déjà ouvert et que la demande courante est en mode exclusif. Dans les autres cas, le fichier est ouvert pour le demandeur, et le système met à jour la variable f.
- **procédure** fermer (**var** f: fichier) rompt la liaison avec le fichier, et met à jour la variable f. Si des processus étaient en attente d'ouverture du fichier, le système examine la demande de celui qui est en tête de la file, pour savoir si les conditions sont maintenant favorables à sa réactivation. De plus, si les conditions sont favorables et que cette demande est en mode partagé, alors il réactive toutes les demandes en mode partagé qui sont dans la file.

```
var f : fichier;
   enrg: t_enregistrement;
   tab_numéro: tableau [1..2] de entier;
début début_prog;
       répéter
          lecture_terminal (type);
          cas type dans
               consultation: début
                     lecture_terminal (tab_numéro);
                     début_consult;
                     pour i := 1 jusqu'à 2 faire
                          lire_enrg (f, enrg, tab_numéro[i]);
                          afficher ("stock = ", enrg.quantite);
                     fait;
                     fin_consult;
               fin;
               mise_à_jour: début
                     lecture_terminal (numéro, variation);
                     début_mise_à_jour;
                     lire_enrg (f, enrg, numéro);
                     enrg.quantité := enrg.quantité + variation;
                     écrire_enrg (f, enrg, numéro);
                     fin_mise_à_jour;
               fin;
               sortie:;
          fincas;
       jusqu'à commande = sortie;
       fin_prog;
fin;
```

Les durées d'exécution des opérations, ainsi que les durées totales d'attente qui en résultent (entrées-sorties) lorsque le processus est seul, sont données par le tableau suivant:

opération	lecture_terminal	ouvrir	fermer	lire_enrg	écrire_enrg	afficher
attente en ms.	10000	300	0	50	50	250
durée UC en ms.	20	30	10	10	50	10

A- On fait l'ouverture du fichier “stock” en mode partagé dans début_prog, et la fermeture dans fin_prog, les autres début/fin étant vides. Montrer que ceci peut conduire à des erreurs dans les valeurs mémorisées dans le fichier.

B- On modifie le mode d'ouverture en mode exclusif. Cela empêche-t-il ces erreurs de se produire? Quelles sont les conséquences de cette modification, pour l'utilisateur assis devant son terminal? Évaluer le nombre de consultations ou de mises à jour par minute.

C- On fait l'ouverture du fichier “stock” dans début_consult et début_mise_à_jour, et la fermeture dans fin_consult et fin_mise_à_jour, tandis que début_prog et fin_prog sont vides. Quels doivent être les modes d'ouvertures dans chacun des cas? Quelles sont les conséquences de ces modifications? On étudiera, en particulier, les différents temps d'attente et UC d'une consultation ou d'une modification, et on tentera d'en déduire le nombre de consultations ou de modifications qui sont possibles par minute. Montrer, sur un exemple, qu'une mise à jour peut être retardée indéfiniment.

D- Pour améliorer le comportement des processus, le système fournit deux opérations supplémentaires (le temps d'exécution de chacune d'elles est de 10 ms. unité centrale uniquement):

- **procédure** `réouvrir (`**var** `f: fichier; mode: t_mode)` le fichier doit avoir fait l'objet d'une fermeture temporaire (voir ci-dessous). Cette fonction a la même fonctionnalité que `ouvrir`, si ce n'est que le fichier `f` est supposé déjà repérer un fichier externe. Le système regarde si un autre processus a ouvert ce fichier. Le processus demandeur est bloqué (mis en file d'attente) si le fichier était déjà ouvert en `mode exclusif`, ou s'il était déjà ouvert et que la demande courante est en `mode exclusif`. Dans les autres cas, le fichier est ouvert pour le demandeur, et le système met à jour la variable `f`.
- **procédure** `fermer_tempo (`**var** `f: fichier)` rompt temporairement la liaison avec le fichier, et met à jour la variable `f`, qui repère toujours le fichier, mais ne permet aucun accès autre que `réouvrir`. Si des processus étaient en attente d'ouverture du fichier, le système examine la demande de celui qui est en tête de la file, pour savoir si les conditions sont maintenant favorables à sa réactivation. De plus, si les conditions sont favorables et que cette demande est en `mode partagé`, alors il réactive toutes les demandes en `mode partagé` qui sont dans la file.

Proposer les procédures de début/fin qui vous paraissent acceptables dans ces conditions. Quelles sont les conséquences sur le comportement des processus? On étudiera, en particulier, les différents temps d'attente et UC d'une consultation ou d'une modification, et on tentera d'en déduire le nombre de consultations ou de modifications qui sont possibles par minute. Montrer que ceci n'empêche toujours pas le retard indéfini des mises à jour.

E- Quelle devrait être la fonctionnalité de `réouvrir` et de `fermer_tempo` pour empêcher le retard indéfini des mises à jour?

Solution de l’exercice 4.6

4.6.1. Question A

Prenons deux processus qui tentent d'effectuer une mise à jour d'un enregistrement de même numéro, par exemple 1, et lui porter une variation de 5. Supposons que initialement la quantité mémorisée dans cet enregistrement soit de 20. Le résultat de l'exécution séquentielle de ces deux processus, c'est-à-dire, l'un derrière l'autre, est égal à 20 + 5 + 5 = 30. Puisque le mode d'ouverture est `partagé`, les deux processus pourront accéder au fichier en même temps. Proposons l'ordonnancement suivant des actions de ces deux processus:

```
processus 1                          processus 2
lire_enrg (f, enrg, numéro);
{ enrg.quantité vaut 20 }
                                     lire_enrg (f, enrg, numéro);
                                     { enrg.quantité vaut 20 }
                                     enrg.quantite := enrg.quantité + variation;
                                     { enrg.quantité vaut 25 }
                                     écrire_enrg (f, enrg, numéro);
                                     { la valeur 25 est écrite dans le fichier }
{ enrg.quantité vaut toujours 20 }
enrg.quantite := enrg.quantité + variation;
{ enrg.quantité vaut 25 }
écrire_enrg (f, enrg, numéro);
{ la valeur 25 est écrite dans le fichier }
```

Le résultat final est alors de 25 au lieu de 30. Il y a bien risque d'erreurs.

4.6.2. Question B

Si on change le mode d'ouverture en `exclusif`, ces problèmes ne pourront plus se produire, puisque dans ce cas les deux processus ne peuvent accéder au fichier en même temps. Le premier obtiendra l'ouverture, le second sera mis en attente jusqu'à ce que le premier ferme le fichier. La conséquence immédiate est que le premier utilisateur qui lancera ce programme ouvrira le fichier en exclusif, empêchant tous les autres processus d'accéder au fichier. Tant que le premier utilisateur n'a pas terminé ses consultations ou ses mises à jour, les autres processus sont bloqués! Cela peut durer des heures.

Le temps total d'une consultation est 2 * 10020 + 2 * (60 + 260) = 20.7 sec, le temps total d'une mise à jour est 2 * 10020 + 60 + 100 = 20.2 sec; il y a donc en moyenne 3 consultations ou mise à jour par minute.

4.6.3. Question C

L'ouverture du fichier dans `début_consult` doit se faire en mode `partagé`. Il n'est pas gênant en effet que plusieurs processus accèdent en même temps au fichier en consultation, puisque alors le fichier n'est pas modifié. Par contre, l'ouverture dans `début_mise_à_jour` doit se faire en mode `exclusif`, pour éviter les problèmes énoncés dans la question A. Ceci a pour conséquence qu'un processus “réserve” le fichier pour la durée d'une consultation ou d'une modification au lieu de la durée totale d'exécution du programme.

Supposons d'abord qu'il n'y ait que des consultations. Le processus répartit son temps de la façon suivante:

attente utilisateur	2 * 10000	= 20 secondes
attente disque	300 + 2 * 50	= 400 ms
attente affichage	2 * 250	= 500 ms
traitement UC	2 * 20 + 30 + 2 * (10 + 10) + 10	= 120 ms
total		= 21 secondes
réservation fichier	2 * (10 + 50 + 250 + 10) + 10	= 650 ms

Le disque limite à 60 / 0.4 = 150 le nombre de consultations par minute, pour l'ensemble des processus, alors qu'un processus ne peut faire au plus que 3 consultations par minute. On peut donc en conclure que, en théorie, 150 / 3 = 50 processus peuvent faire des consultations simultanément. Notons que l'UC est occupée 150 * 120 = 18 secondes par minute.

Supposons maintenant qu'il n'y ait que des mises à jour. Le processus répartit son temps de la façon suivante:

attente utilisateur	2 * 10000	= 20 secondes
attente disque	300 + 2 * 50	= 400 ms
traitement UC	2 * 20 + 30 + 10 + 50 + 10	= 140 ms
total		= 20.5 secondes
réservation fichier	2 * 50 + 10 + 50 + 10	= 170 ms

La réservation du fichier en accès exclusif n'interdit pas les accès disque nécessaires à son ouverture par un autre processus. La limite prépondérante reste alors l'occupation disque, donnant au mieux 150 mises à jour par minute, en acceptant 50 processus. On obtient alors 21 secondes d'UC par minute.

En conclusion la méthode permet d'atteindre *théoriquement* 150 consultations ou mises à jour par minute, à partir de 50 terminaux. Mais la réservation pour consultation par un processus dure 650 ms. Si on suppose, par exemple, que 45 processus se coalisent pour demander des consultations en permanence, chacun d'eux sera candidat toutes les 21 secondes, et le rythme des demandes d'ouverture en mode partagé sera de 1 toutes les 21 / 45 = 0.46 secondes. Ces demandes arriveront donc toujours avant que le processus précédent ait terminé sa consultation. Le fichier restera alors ouvert en mode partagé, empêchant indéfiniment les 5 autres processus de faire des mises à jour.

4.6.4. Question D

```
début_prog:         ouvrir (f, "stock", partagé);
                    fermer_tempo (f);
début_consult:      réouvrir (f, partagé);
fin_consult:        fermer_tempo (f);
début_mise_à_jour:  réouvrir (f, exclusif);
fin_mise_à_jour:    fermer_tempo (f);
fin_prog:           réouvrir (f, partagé);
                    fermer (f);
```

Les changements qui interviennent sont liés à l'absence d'opération disque pour la réouverture.

Supposons d'abord qu'il n'y ait que des consultations. Le processus répartit son temps de la façon suivante:

attente utilisateur	2 * 10000	= 20 secondes
attente disque	2 * 50	= 100 ms
attente affichage	2 * 250	= 500 ms
traitement UC	2 * 20 + 10 + 2 * (10 + 10) + 10	= 100 ms
total		= 20.7 secondes
réservation fichier	2 * (10 + 50 + 250 + 10) + 10	= 650 ms

Le disque limite à 60 / 0.1 = 600 le nombre de consultations par minute, pour l'ensemble des processus, alors qu'un processus ne peut faire au plus que 3 consultations par minute. On peut donc en conclure que, en théorie, 600 / 3 = 200 processus peuvent faire des consultations simultanément. Notons que l'UC est occupée 600 * 100 = 1 minute par minute, donc à 100 %.

Supposons maintenant qu'il n'y ait que des mises à jour. Le processus répartit son temps de la façon suivante:

attente utilisateur	2 * 10000	= 20 secondes
attente disque	2 * 50	= 100 ms
traitement UC	2 * 20 + 10 + 10 + 50 + 10	= 120 ms
total		= 20.2 secondes
réservation fichier	2 * 50 + 10 + 50 + 10	= 170 ms

La limite prépondérante devient ici la réservation du fichier en mode `exclusif`, qui limite à 60 / 0.17 = 352 mises à jour par minute, en acceptant 352 / 3 = 117 processus. On obtient alors 352 * 120 = 42 secondes d'UC par minute, et 352 * 100 = 35 secondes d'occupation disque par minutes.

En conclusion la méthode permet d'atteindre *théoriquement* 600 consultations ou 352 mises à jour par minute, en utilisant un nombre de terminaux compris entre 117 et 200, suivant la proportion de consultations par rapport aux mises à jour. Mais la réservation pour consultation par un processus dure toujours 650 ms. Comme dans la question précédente, une coalition de 45 processus pour demander des consultations en permanence donnera un rythme des demandes d'ouverture en mode partagé de 1 toutes les 0.46 secondes. Ces demandes arriveront donc toujours avant que le processus précédent ait terminé sa consultation. Le fichier restera alors ouvert en mode partagé, empêchant indéfiniment les autres processus de faire des mises à jour. Le problème sera d'autant plus grave que le nombre de processus est plus grand, et survient donc avec une proportion plus faible de processus qui se coalisent.

4.6.5. Question E

Pour corriger cette dernière anomalie, il suffit que `réouvrir` bloque systématiquement le processus demandeur, quel que soit le mode d'ouverture demandé, si la file d'attente n'est pas vide, et mette ce processus en bout de la file. Par ailleurs, l'opération `fermer_tempo` doit être modifiée pour ne pas réactiver les processus de la file qui demandent l'ouverture en mode `partagé` et qui sont derrière un processus qui demande l'ouverture en mode `exclusif`.

```
réouvrir:
     si file [f] non vide
     ou f déja ouvert en exclusif
     ou f déja ouvert en partagé et demande en exclusif alors
          bloquer le demandeur et l'ajouter en bout de file [f]
     finsi;
     autoriser les accès du processus à f

fermer_tempo:
     si file [f] non vide et f n'est plus ouvert alors
          extraire et activer le processus de tête de file [f]
          si demande en partagé alors
               tant que file [f] non vide
               et demande du processus de tête en partagé faire
                    extraire et activer le processus de tête de file [f]
               fait;
          finsi;
     finsi;
```

4.7. Durée de section critique

Le but de ce petit problème est de montrer l'influence de la durée d'une section critique sur les performances globales.

La programmation de la réservation de place d'avions peut s'envisager suivant deux méthodes. Le schéma général du programme serait le suivant:

```
var f : fichier;
   enrg : t_enregistrement;
début répéter
          lecture_terminal ( type, vol, date );
          cas type dans consultation: consulter;
               réservation: réserver;
               sortie:;
          fincas;
       jusqu'à type = sortie;
fin;
```

Pour les deux méthodes, la procédure `consulter` est la suivante:

```
procédure consulter
début début_consult;
       lire_enrg (f, vol, date, enrg);
       places_libres := enrg.nb_place;
       fin_consult;
       afficher ( " places disponibles: ", places_libres);
fin;
```

Dans la première méthode, la procédure `réserver` est la suivante:

```
procédure réserver;
début début_reserv;
       lire_enrg (f, vol, date, enrg);
       si enrg.nb_place > 0 alors
          afficher ( "il y a de la place; en voulez-vous une? " );
          lecture_terminal ( réponse );
          si réponse = "oui" alors
               enrg.nb_place := enrg.nb_place - 1;
               écrire_enrg (f, vol, date, enrg);
          finsi;
       sinon afficher ( "plus de place ");
       finsi;
       fin_reserv;
fin;
```

Dans la deuxième méthode, la procédure `réserver` est la suivante:

```
procédure réserver;
var OK: booléen;
début début_reserv;
       lire_enrg (f, vol, date, enrg);
       si enrg.nb_place > 0 alors
          enrg.nb_place := enrg.nb_place - 1;
          écrire_enrg (f, vol, date, enrg); OK := vrai;
       sinon OK := faux;
       finsi;
       fin_reserv;
       si OK alors afficher ( "place réservée ");
       sinon afficher ( "plus de place ");
       finsi;
fin;
```

Le système fournit, pour chaque fichier ouvert, un mécanisme de verrouillage au moyen de deux opérations. Ces opérations n'ont aucun effet sur les opérations de lecture et d'écriture sur le fichier.

- `procédure verrouiller (f)` met le processus demandeur dans une file associée au fichier s'il y a déjà un processus qui a verrouillé le fichier, sinon pose un verrou sur le fichier.
- `procédure déverrouiller (f)` réactive un des processus de la file associée au fichier s'il y en a et enlève le verrou s'il n'y en a pas.

A- Que doivent être les opérations de `début_reserv` et `fin_reserv` dans chacune des méthodes? Justifiez.

B- Que doivent être les opérations de `début_consult` et `fin_consult` dans chacune des deux méthodes? Justifiez.

C- Comparer les avantages et inconvénients des deux méthodes, et dites celle qui vous semble préférable.

Solution de l'exercice 4.7

4.7.1. Question A

La réservation entraîne une lecture d'un enregistrement pour le modifier éventuellement. Il ne doit pas y avoir plusieurs processus en train de faire la même opération sur le même enregistrement, car le résultat serait erroné. Le verrouillage tel qu'il est proposé permet de garantir qu'un seul processus peut détenir le verrou. Il doit donc y avoir verrouillage du fichier dans la procédure `début_reserv`, et déverrouillage dans `fin_reserv`.

```
début_reserv:       verrouiller (f);
fin_reserv:         déverrouiller (f);
```

4.7.2. Question B

La consultation ne fait que lire un enregistrement unique. Une telle lecture depuis le disque est souvent indivisible (lecture d'un secteur), et il n'est donc pas nécessaire d'utiliser une quelconque contrainte de synchronisation, et `début_consult` et `fin_consult` sont vides. Si une réservation est en cours, et qu'un verrou est posé sur le fichier, on ne peut dire si la consultation lira l'enregistrement avant ou après la modification. Cependant si la lecture a lieu avant, le résultat sera le même que si toute la consultation s'était déroulée avant le verrouillage, alors que si elle a lieu après, le résultat sera le même que si toute la consultation s'était déroulée après le déverrouillage.

4.7.3. Question C

La première méthode est séduisante car le demandeur sait qu'à partir du moment où le système lui dit qu'il y a de la place, il peut la réserver, alors que dans la deuxième méthode, il peut se voir refuser sa réservation, puisque consultation et réservation sont deux opérations séparées. Le gros inconvénient de la première méthode est que le verrouillage dure plus longtemps: il peut même durer un temps assez important puisque le programme attend la réponse du demandeur. Ceci peut durer un temps appréciable si le demandeur n'arrive pas à se décider ou s'il a été accaparé par une autre tâche. La durée de la réservation pouvant atteindre, par exemple 30 secondes, limitera à 2 le nombre de réservations par minute, ce qui est très faible. Il est probable d'ailleurs qu'il n'y aura jamais de consultation, le demandeur préférant faire une réservation puisqu'il peut toujours changer d'avis par la suite. Dans la seconde méthode, la durée de verrouillage est limitée et n'excèdera pas 0.1 secondes (2 accès disque), ce qui donnera 600 réservations par minute. La deuxième méthode est donc de beaucoup préférable, car elle offre une meilleure disponibilité. Son inconvénient essentiel est que si le demandeur décide de réserver après avoir constaté qu'il y avait de la place, sa réservation peut être refusée, si entre temps les places restantes ont été réservées par d'autres. Cet inconvénient est considéré souvent comme peu probable et donc mineur.

4.8. Applications transactionnelles

Une entreprise est organisée en 10 départements. Chaque département gère lui-même ses données relatives aux bons de commandes, aux factures et aux paiements qui le concernent. Pour pallier les problèmes des accès partagés, la gestion informatique est basée sur le schéma de transaction suivant.

```
réservation des données
traitement préliminaire                    10 ms
lecture disque de la donnée concernée
traitement de la donnée                    20 ms
écriture disque de la donnée concernée
fin de traitement                          10 ms
libération des données
```

Les temps indiqués sont les temps d'unité centrale. Le temps moyen d'accès disque est de 30 ms.

A- Lors d'une transaction, la réservation des données porte sur l'ensemble des données de l'entreprise. Déterminer le nombre maximum de transactions par seconde, en justifiant votre réponse.

B- Lors d'une transaction, la réservation des données ne porte plus sur l'ensemble des données de l'entreprise, mais seulement sur celles du département concerné. Toutes les données sont sur un seul disque. Déterminer le nombre maximum de transactions par seconde, en justifiant votre réponse.

C- Lors d'une transaction, la réservation des données porte toujours sur les données du département concerné, mais cette fois l'ensemble des données sont réparties sur deux disques. Déterminer le nombre maximum de transactions par seconde, en justifiant votre réponse.

Solution de l'exercice 4.8

4.8.1. Question A

Notons d'abord que les traitements font une lecture sur disque suivie d'une écriture sur la même donnée qui est donc une donnée partagée qui doit être en accès exclusif durant la transaction.

Chaque transaction réserve les données pendant la durée de son exécution (UC et attente disque), soit 10 + 30 + 20 + 30 + 10 = 100 ms. Il ne peut donc y avoir au plus que 10 transactions par secondes.

4.8.2. Question B

Chaque transaction réserve les données du département pendant 100 ms. Les données des autres départements sont donc libres. Comme il y a 10 départements, il peut y avoir 10 transactions en parallèles qui concourent aux autres ressources : l'UC pour 40 ms et le disque pour 60 ms. La limitation de l'UC est donc de 25 transactions par seconde, mais la limitation du disque est à 17 par seconde. C'est donc le disque qui est la ressource la plus demandée. Il ne peut y avoir au plus que 17 transactions par seconde.

4.8.3. Question C

Si les données sont réparties sur deux disques, chaque disque peut satisfaire 17 transactions par seconde, et l'ensemble des deux disques permettent d'avoir 34 transactions par seconde. Cette fois, c'est l'UC qui est la ressource la plus demandée. Il ne peut y avoir plus de 25 transactions par seconde.

Notons que ces résultats sont « théoriques ». Ils garantissent que ces bornes ne peuvent être dépassées, et non qu'elles sont atteintes. En réalité, les conflits entre les demandes des transactions, le temps processeur pour traiter ces conflits, et la dispersion entre les mesures de temps indiquées comme moyennes entraînent que ces maxima ne seront pas atteint. En fait ces bornes correspondent à la saturation du système, et il faut toujours prendre une marge de sécurité sans espérer des miracles.

4.9. A propos d'interblocage

Dans le service de gestion d'un magasin, un employé est chargé d'enregistrer les commandes des clients dans un fichier COM, et d'éditer les bons de commandes correspondants, sur une imprimante IMP. D'autre part, un processus facturation, lancé périodiquement lit les commandes à facturer dans

COM et édite les factures correspondantes sur l'imprimante IMP. Notons qu'une commande peut concerner plusieurs articles, et donc qu'un bon de commande ou une facture peut comporter plusieurs lignes.

A- Sachant que l'opération de transfert élémentaire (lecture, écriture) concerne une ligne de commande, montrer que l'état de COM peut être incohérent, de même que les impressions sur IMP. En déduire les règles d'accès à COM et à IMP.

B- Afin de résoudre le problème précédent, on fournit deux procédures qui garantissent un accès exclusif à une ressource R:

`réserver(R)` autorise l'accès à `R` par le demandeur si `R` est libre ou bloque le demandeur si `R` est occupée.

`libérer(R)` autorise l'accès à un autre demandeur s'il y en a en attente, sinon indique que `R` est libre.

On propose la solution suivante :

```
processus employé :
début tant qu'il y a des commandes à enregistrer faire
          réserver(COM);
          enregistrer une commande dans COM;
          réserver(IMP);
          éditer un bon de commande sur IMP;
          libérer(COM);
          libérer(IMP);
       fait;
fin;
processus facturation;
début répéter indéfiniment
          réserver (IMP);
          réserver (COM);
          tant qu'il y a des commandes à facturer dans COM faire
               lire la prochaine commande dans COM;
               éditer la facture correspondante sur IMP;
          fait;
          libérer(IMP);
          libérer(COM);
          attendre la prochaine période de facturation;
       fait;
fin;
```

B.1- Montrer que cette programmation permet d'éditer de manière consécutive, toutes les factures pour une période donnée.

B.2- Donner une définition de l'interblocage. Montrer que la programmation risque de conduire à un interblocage.

B.3- Modifier le processus de facturation pour supprimer le risque d'interblocage.

Solution de l'exercice 4.9

4.9.1. Question A

Lorsque l'employé saisit une commande, l'écriture sur disque d'une ligne à la fois, implique que, à un instant donné, le disque ne contient qu'une partie de la commande. Si le processus de facturation est lancé, il ne trouvera pas toutes les lignes de la commande pour éditer la facture, qui sera donc partielle. Au moment où on crée une commande, il faut donc interdire au processus de facturation d'accéder aux commandes.

Par ailleurs, l'employé édite les commandes saisies sur la même imprimante que le processus de facturation. Ces éditions se font ligne par ligne, mais toutes les lignes concernant le même bon de commande ou la même facture doivent se trouver regroupées, et non entremêlées. Il faut donc que l'imprimante soit en exclusion mutuelle entre les deux processus.

4.9.2. Question B

4.9.2.1. Question B.1

La solution proposée garantit bien l'accès en exclusion mutuelle à l'imprimante par les deux processus. Le processus de facturation réserve l'imprimante pendant le traitement des factures d'une période. Ces factures seront donc bien éditées de manière consécutive. Par ailleurs, comme le processus réserve également le fichier des commandes, aucune commande ne peut être en cours de saisie pendant l'édition des factures.

4.9.2.2. Question B.2

L'interblocage est une situation où un ensemble de processus sont bloqués en attente d'une ressource possédée par un autre processus de l'ensemble. Chacun attend qu'un autre veuille bien libérer la ressource qu'il attend. Ceci ne peut se faire sans une intervention extérieure, puisqu'ils sont tous bloqués. Or on ne peut débloquer un processus qu'en lui donnant toutes les ressources nécessaires, et donc en réquisitionnant celle qu'il attend et qui est possédée par un autre processus de l'ensemble.

Dans la solution proposée, on peut imaginer que le processus employé réserve le fichier COM et commence à saisir la commande. À ce moment le processus de facturation est activé, et réserve l'imprimante, puis se bloque en attente du fichier COM. Lorsque l'employé a terminé la saisie, il réserve l'imprimante, mais comme celle-ci est déjà réservée par le processus de facturation, il se bloque en attente de la libération. Nous avons alors deux processus qui attendent mutuellement la libération d'une ressource possédée par un autre processus de l'ensemble : ces deux processus sont en interblocage.

4.9.2.3. Question B.3

Pour ne plus avoir d'interblocage, une des solutions est de réserver les ressources dans le même ordre, puisque, dans ce cas, il ne peut plus y avoir de circularité dans les attentes de ressources. Dans le processus de facturation, il faut donc réserver le fichier COM en premier.

```
processus facturation;
début répéter indéfiniment
          réserver (COM);
          réserver (IMP);
          tant qu'il y a des commandes à facturer dans COM faire
               lire la prochaine commande dans COM;
               éditer la facture correspondante sur IMP;
          fait;
          libérer(IMP);
          libérer(COM);
          attendre la prochaine période de facturation;
       fait;
fin;
```

4.10. Gestion de comptes bancaires

On considère un système de gestion de comptes bancaires permettant d'effectuer les deux opérations suivantes :

`CredDeb_Compte (Numéro_Compte, opération, somme)` crédite ou débite le compte `Numéro_Compte` de la valeur `somme` en fonction de l'opération demandée (`opération` = `débiter` ou `créditer`).

`Donner_Solde (Numéro_Compte, var Solde)` renvoie le `Solde` du compte.

Un client peut accéder à chacune de ces opérations indépendamment les unes des autres et plusieurs clients accèdent en parallèle au système de gestion des comptes bancaires. On supposera que chaque compte bancaire est stocké sur disque dans un fichier appelé `Fichier_Compte`.

On donne ci-dessous le pseudo-code de ces deux fonctions :

```
procédure CredDeb_Compte (Numéro_Compte, opération, somme)
début     --lecture dans le fichier Fichier_Compte
          -- du solde du compte Numéro_Compte
     Lire (Fichier_Compte, Numéro_Compte, solde_compte);
     si (opération = débiter)
     alors solde_compte := solde_compte - somme;
     sinonsolde_compte := solde_compte + somme;
     fsi
          -- écriture dans le fichier Fichier_Compte
          -- du nouveau solde du compte Numero_Compte
     Ecrire (Fichier_Compte, Numéro_Compte, solde_compte);
fin
```

```
procédure Donner_Solde (Numéro_Compte, var Solde)
début     -- lecture dans le fichier Fichier_Compte
          -- du solde du compte Numéro_Compte
     Lire (Fichier_Compte, Numéro_Compte, solde_compte);
     Solde := solde_compte;
fin
```

A- En prenant un exemple, montrez que des incohérences peuvent survenir sur le solde d'un compte si plusieurs utilisateurs accèdent à un compte en parallèle.

B- On dispose des deux opérations suivantes :

`Verrouiller (Nom_Fich)` qui bloque le processus effectuant cette opération si un autre processus a verrouillé le fichier `Nom_Fich`

`Déverrouiller (Nom_Fich)` qui libère l'accès au fichier `Nom_Fich` et réveille un processus bloqué par l'opération `Verrouiller (Nom_Fich)`, si il en existe un.

Comment utilisez ces opérations pour résoudre le problème de la question A. Justifiez votre raisonnement.

C- On désire autoriser les accès en écriture simultanément avec les accès en lecture pour un même compte. Y a-t-il une difficulté ? Justifiez votre raisonnement.

D- On désire rajouter une opération qui permet d'obtenir les soldes d'un ensemble de comptes, par exemple tous les comptes d'un même client. Son pseudo code est le suivant.

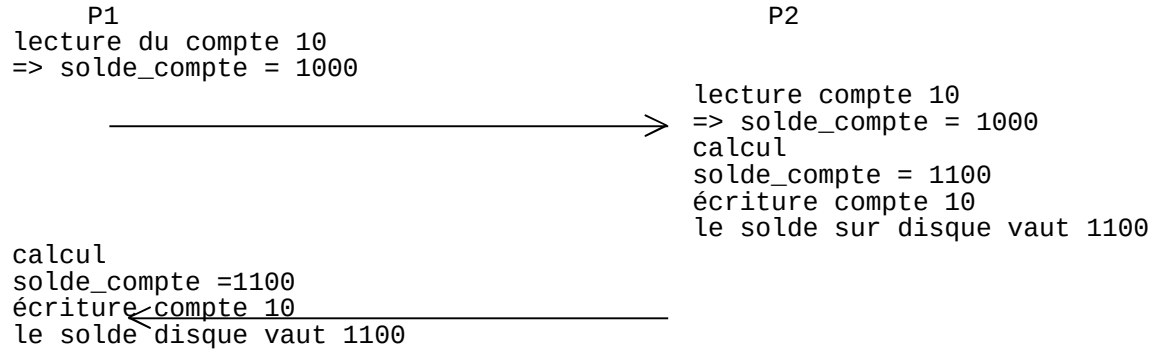
```
procédure Donner_Les_Soldes (Nb, Numéro_Compte[], var Solde[])
début     -- lecture dans le fichier Fichier_Compte
          -- des soldes des comptes du tableau Numéro_Compte
     pour I de 1 à Nb faire
          Lire(Fichier_Compte, Numéro_Compte[I], solde_compte);
          Solde[I] := solde_compte;
     fait ;
fin
```

Quelle difficulté y a-t-il à permettre les accès en écriture simultanément à l'exécution de cette opération ? Comment peut-on résoudre cette difficulté ?

Solution de l'exercice 4.10

4.10.1. Question A

Supposons que deux processus P1 et P2 exécutent, chacun de leur côté, la même opération `CredDeb_Compte (10, créditer, 100)`. Le déroulement de ces opérations peut être les suivant :



Le solde final sur disque vaut 1100 au lieu de 1200.

4.10.2. Question B

Pour palier cette difficulté, il faut insérer `Verrouiller (Fichier_Compte)` au début de la procédure `CredDeb_Compte`, et `Deverrouiller (Fichier_Compte)` à la fin de `CredDeb_Compte`.

Dans l'exemple ci-dessus, le verrouillage par P1 avant la lecture du compte interdira à P2 de lire le compte tant que P1 ne l'a pas libéré, donc après la réécriture du compte (valeur 1100). P2 sera alors libéré, et verra le compte modifié par P1, donc la valeur 1100, et il réécrira la nouvelle valeur 1200 ensuite.

4.10.3. Question C

La lecture du solde simultané aux écritures ne posent pas de difficulté particulière : le processus qui exécute `Donner_Solde` verra la valeur du solde soit avant soit après la modification par l'écrivain, comme si l'exécution complète du processus avait eu lieu avant le début ou après la fin d'exécution de l'écrivain.

4.10.4. Question D

Cette fois, l'écrivain peut venir modifier un compte déjà lu (le premier du tableau par exemple) et un autre qui sera lu plus tard (le dernier du tableau par exemple). Si l'action de l'écrivain consiste à débiter 1000 F du premier compte et créditer 1000 F au deuxième compte, le débit ne sera pas vu par Donner_Les_Soldes mais le crédit le sera. Le client apparaîtra plus riche de 1000 F qu'il ne l'est en réalité.

Pour résoudre ce problème, il suffit de verrouiller le fichier au début de Donner_Les_Soldes et de le déverrouiller à la fin. Ce verrouillage, s'il résout le problème est plus fort que nécessaire, car il interdit plusieurs exécutions de Donner_Les_Soldes en parallèles alors qu'elles peuvent avoir lieu sans problème puisque l'opération ne modifie pas les comptes. Pour le permettre, il faudrait mettre en œuvre le schéma lecteur-rédacteur.

4.11. Variante du petit système temps réel

L'exercice est une variante du problème 1.3 «Petit système temps-réel», où nous supposons certaines fonctionnalités du système d'exploitation pour résoudre le problème.

On dispose d'un ordinateur monoprocesseur, équipé d'un système capable de gérer plusieurs processus en pseudo-parallélisme. On désire construire une application qui permette l'exécution périodique, avec une période de T millisecondes, d'un cycle de mesure:

- prélèvement de mesures sur des capteurs, (durée: *tmes*)
- traitement des mesures (durée: *tcalc*)
- écriture dans un fichier des résultats (durée: *tvid*)

Le système fournit les opérations suivantes:

`procedure lecture_capteur (var Z : tampon);` Le système lance la commande de prélèvement de mesures dans le tampon `Z`, et bloque le processus jusqu'à ce que le prélèvement soit terminé.

`procedure ecriture_fichier (f : flot; Z : tampon);` Le système lance la commande d'écriture du tampon `Z` dans le fichier relié au flot `f`, et bloque le processus jusqu'à ce que cette écriture soit terminée.

`procedure attendre_pendant (D : durée);` Le système bloque le processus pendant `D` millisecondes. Lorsque cette durée est écoulée, le processus est remis dans la file des processus prêts. Notons que ces opérations ont un temps d'exécution processeur négligeable.

A- La période T est supposée assez longue vis à vis des durées *tmes*, *tcalc* et *tvid*.

A.1- Proposer un programme qui implante l'application au moyen d'un processus unique.

A.2- Quelle difficulté résulte de la procédure `attente_pendant`? (Pensez à évaluer la véritable période de votre solution).

B- On suppose maintenant que *tmes* est constant et vaut 0.1*T, que *tvid* est constant et vaut 0.7*T et que *tcalc* est variable: en général il vaut 0.3*T, sauf une fois tous les 5 prélèvements où il vaut 3*T. Notons que *tmes* et *tvid* représentent des temps d'attente de fin d'entrées-sorties, alors que *tcalc* représente un temps d'exécution processeur.

Le système fournit un mécanisme de communication entre processus par boîtes aux lettres (§ 13.2.3). Les messages envoyés ou extraits par les processus sont gérés dans un tampon interne au système, propre à chaque boîte aux lettres, et de taille limitée à 10 messages. Les boîtes aux lettres ont un nom qui est une chaîne de caractères, et le système gère une table des noms qui sont associés aux boîtes aux lettres existantes. Les opérations de base sont les suivantes, (en plus des précédentes):

`fonction ouvrir_boîte_aux_lettres (nom: chaîne):id_bal;` Le système recherche dans la table l'`id_bal` de la boîte aux lettres associée à `nom`, et la crée si elle n'existe pas. Il retourne ensuite l'identité de la boîte aux lettres déjà existante ou qui vient d'être créée.

`procedure envoyer (bal : id_bal; Z : tampon);` Le système met dans la boîte aux lettres `bal` le message contenu dans le tampon `Z`. S'il n'y a pas de place, le processus demandeur est bloqué jusqu'à ce qu'il y en ait. Par ailleurs, si la boîte aux lettres était vide et qu'un processus était en attente de message, celui-ci est délivré.

`procedure recevoir (bal : id_bal; var Z : tampon);` Le système extrait un message de la boîte aux lettres et le met dans le tampon `Z`. S'il n'y en avait pas, le processus demandeur est bloqué jusqu'à ce qu'il y en ait. Par ailleurs, si la boîte aux lettres était pleine et qu'un processus était en attente d'envoi de message, ce processus est débloqué.

B.1- Proposer une solution avec trois processus communiquant par boîtes aux lettres: un processus prélèvement, un processus traitement et un processus écriture.

B.2- En considérant que le premier prélèvement est celui qui nécessite un traitement de 3*T, donner un chronogramme (ou suite des événements datés) des activités des trois processus pour 7 périodes successives. On suppose qu'il n'y a que ces trois processus en mémoire, et que, si plusieurs d'entre eux sont prêts, l'allocateur du processeur choisit de rendre actif de préférence prélèvement puis écriture et enfin traitement.

B.3- Qu'apportent les boîtes aux lettres dans cette solution, et qui lui permet de satisfaire aux valeurs données de *tmes*, *tvid* et *tcalc*.

B.4- Que devient la difficulté évoquée en A.2?

C- Le système effectue lui-même les prélèvements à la période T, et les range dans un tampon L. Il fournit la procédure suivante:

`procedure lecture_prélèvement (var Z: tampon; var perte : boolean);` Si les données du tampon L n'ont pas encore été lues par un processus, le système les recopie dans le tampon `Z`, en indiquant dans `perte` si les données actuelles de L ont écrasé des prélèvements qui n'ont pas été lus. Si les données du tampon L ont déjà été lues, le processus est bloqué jusqu'au prochain prélèvement.

C.1- Modifier le processus prélèvement pour utiliser cette procédure sans vous préoccuper de la valeur de `perte` au retour.

C.2- Pourquoi ne pas permettre au processus prélèvement d'accéder directement au tampon L?

C.3- La difficulté A.2 est elle résolue?

C.4- Avait-on quelle que chose d'analogue à la valeur de `perte` dans les solutions précédentes? Que peut-on en faire?

Solution de l'exercice 4.11

4.11.1. Question A

4.11.1.1. Question A.1

Le programme pourrait être le suivant:

```
répéter
     lecture_capteur (Tloc); /* prélèvement en tampon local */
     traitement_mesures; /* traitement des mesures dans Tloc */
     ecriture_fichier (f, Tloc) /* écriture résultats depuis le tampon Tloc */
     attendre_pendant (T - tmes - tcalc - tvid);
fin répéter         /* attendre jusqu'à la fin de la période */
```

4.11.1.2. Question A.2

La procédure `attendre_pendant` ne permet pas d'avoir une période précise des cycles de mesures. Au moment de l'appel à cette procédure, on a tenté d'évaluer le temps déjà écoulé depuis le début du cycle en cours, de façon à déterminer le temps restant jusqu'au début du cycle suivant, et demander un blocage pendant ce délai. Cependant on n'a aucun moyen de connaître ce temps écoulé, or plusieurs éléments peuvent modifier ce temps d'un cycle au suivant:

- La durée d'exécution effective des différentes procédures utilisées n'est sans doute pas fixe. On sait bien, par exemple, qu'une opération d'écriture sur disque varie de façon aléatoire dans certaines limites. De même, le temps de traitement n'est sans doute pas fixe, mais dépend sans doute des données.
- Dans un contexte multiprocessus, il est difficile d'évaluer le temps qu'un processus passe dans la file d'attente des processus prêts, parce que d'autres processus utilisent le processeur. Or la procédure `attendre_pendant` garantit simplement que le processus est bloqué pendant `T – tmes – tcalc - tvid` millisecondes. En primant la durée effective des opérations, et en notant `tpret` le temps passé par le processus dans la file des processus prêts, on peut donc en déduire que la période effective sera

  ```
  T' = T+(tmes'-tmes)+(tcalc'-tcalc)+(tvid'-tvid)+tpret
  ```

Si la période T est assez longue vis à vis des durées *tmes*, *tcalc* et *tvid* et si le système n'est pas trop chargé, on peut penser néanmoins que l'erreur relative commise sur la période ne sera pas trop importante.

4.11.2. Question B

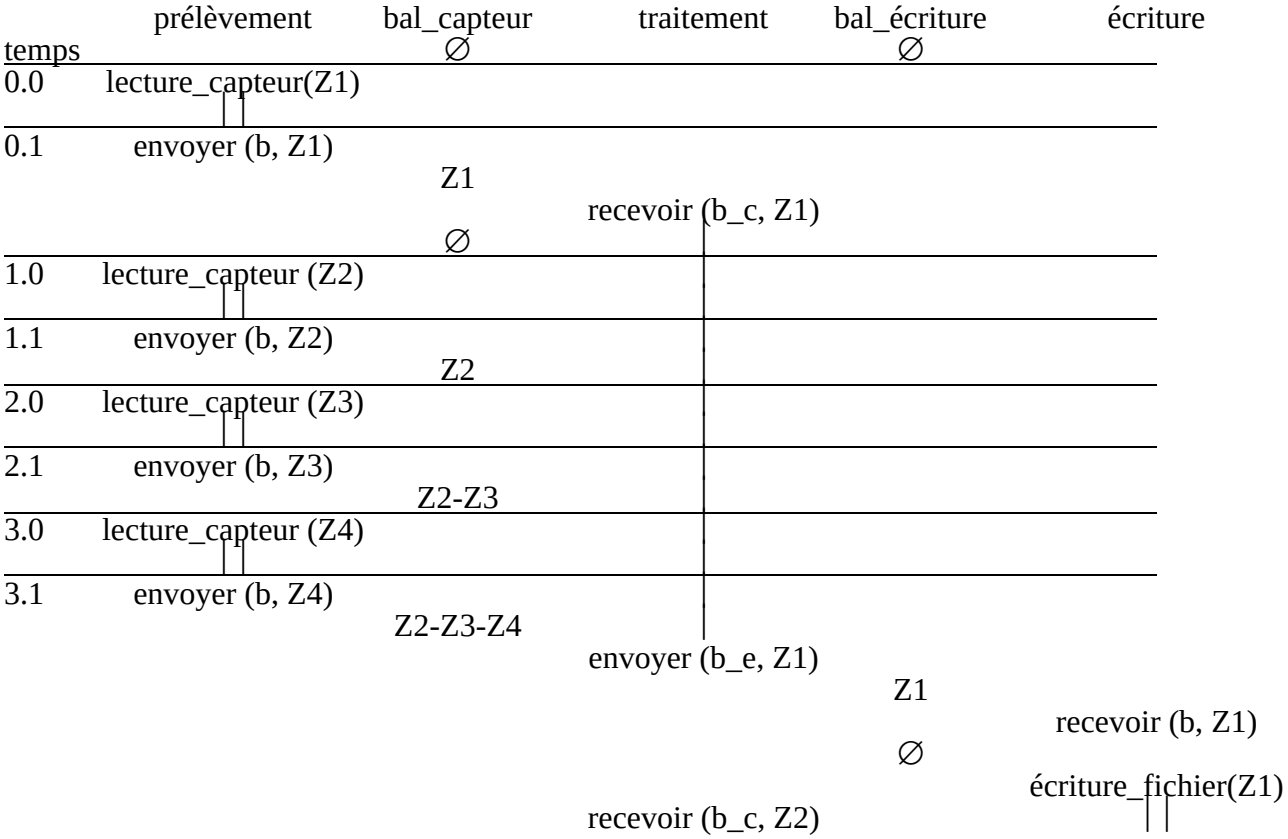
4.11.2.1. Question B.1

Il est nécessaire de disposer de deux boîtes aux lettres permettant aux processus de communiquer entre eux, l'une appelée «`bal_capteur`» entre le processus prélèvement et le processus traitement et l'autre appelée «`bal_écriture`» entre le processus traitement et le processus écriture. Les programmes pour ces trois processus pourraient être:

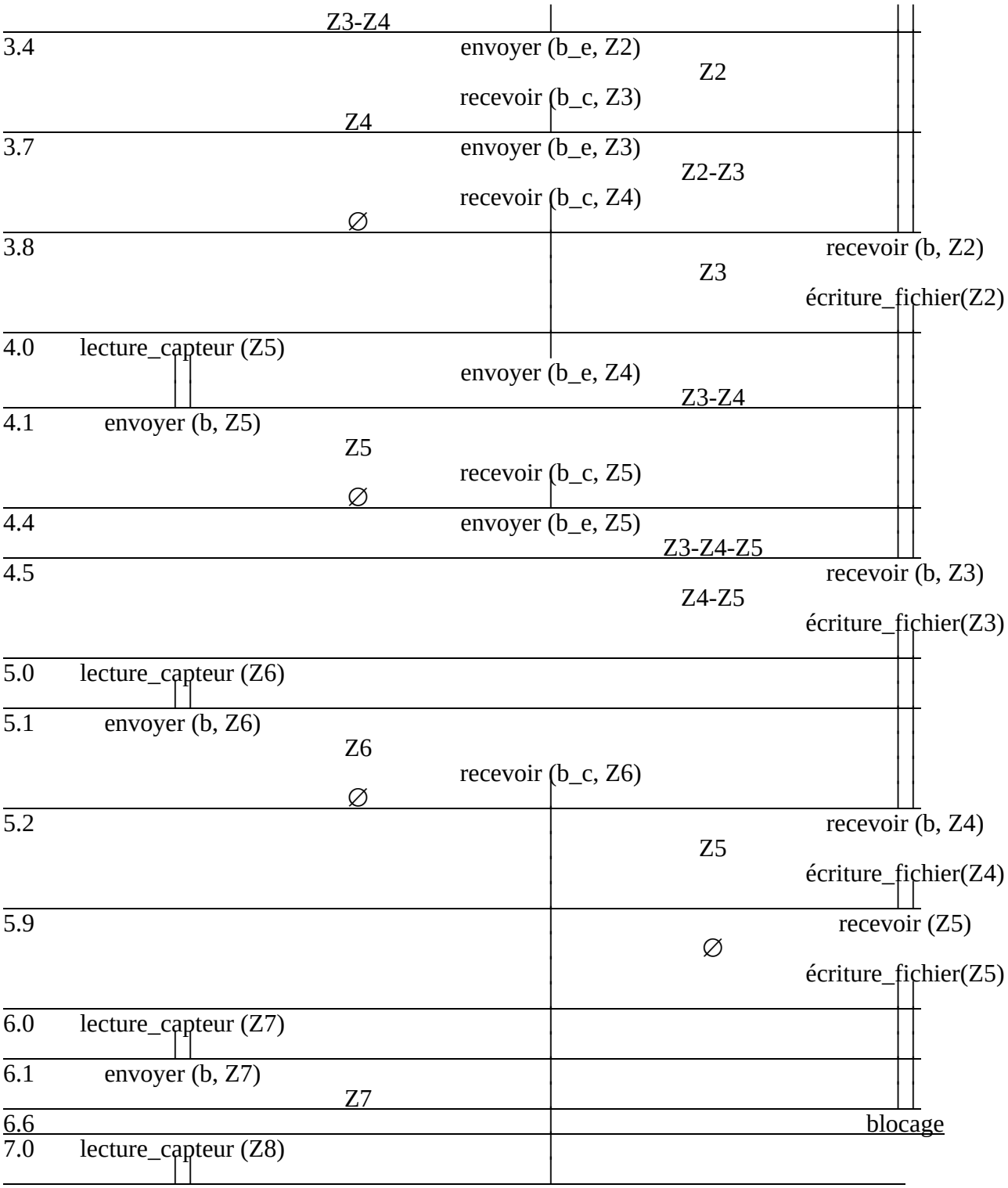
```
processus prélèvement;
     b := ouvrir_boite_aux_lettres ( "bal_capteur");
     répéter
          lecture_capteur (Tloc); /* prélèvement dans un tampon local */
          envoyer (b, Tloc); /* envoi au processus traitement */
          attendre_pendant (T - tmes);
     fin répéter       /* attendre jusqu'à la fin de la période */
fin processus prélèvement;

processus traitement;
     b_c := ouvrir_boite_aux_lettres ("bal_capteur");
     b_e := ouvrir_boite_aux_lettres ("bal_écriture");
     répéter
          recevoir (b_c, Tloc);   /* prélèvement suivant dans un tampon local */
          traitement_mesures;          /* traitement des mesures dans Tloc */
          envoyer (b_e, Tloc);    /* envoi au processus écriture */
     fin répéter
fin processus traitement;

processus écriture;
     b := ouvrir_boite_aux_lettres ( "bal_écriture");
     répéter
          recevoir (b, Tloc); /* résultats suivants dans un tampon local */
          ecriture_fichier (f, Tloc) /* écriture des résultats depuis Tloc */
     fin répéter
fin processus écriture;
```

4.11.2.2. Question B.2

Le chronogramme des activités des trois processus est donné page suivante. Le premier prélèvement nécessite 3*T. On peut constater qu'à la fin, en 7.0, on se trouve dans une situation comparable à celle de l'instant 2.0, où le processus de prélèvement lance une `lecture_capteur`, le processus de traitement a encore un traitement de 1.1*T à effectuer sur le prélèvement en cours, et la boîte aux lettres `bal_capteur` contient un message, alors que le processus écriture est en attente sur la boîte aux lettres `bal_écriture` qui est vide[10].



[10] La barre verticale simple représente une activité processeur, et la barre verticale double une attente d'entrées sorties.



4.11.2.3. Question B.3

Les boîtes aux lettres permettent à chacun des processus de travailler à son propre rythme. Evidemment, il faut que la valeur moyenne de *tcalc* soit inférieure à T, sinon la boîte aux lettres `bal_capteur` contiendra de plus en plus de message, et lorsqu'elle sera pleine entraînera un blocage du processus prélèvement, et donc un retard de ce prélèvement. On peut constater que dans la première partie, la boîte aux lettres `bal_capteur` permet au prélèvement de ne pas être ralenti par le traitement de la première mesure, et que dans la seconde partie, la boîte aux lettres `bal_écriture` permet au processus traitement de rattraper son retard, sans être freiné par le processus écriture.

En particulier, on peut constater que si cette deuxième boîte aux lettres était remplacée par un tampon d'un seul message, traitement ne pourrait délivrer ses résultats que tous les 0.7*T, et non

tous les 0.3*T. Cela impliquerait qu'il ne pourrait extraire les 4 derniers messages de `bal_capteur` que tous les 0.7*T, et donc une durée entre l'extraction du message Z1 et le message Z6 de (3 + 4 * 0.7) * T, soit 5.8*T > 5*T.

4.11.2.4. Question B.4

La difficulté évoquée en A.2 reste, mais elle est atténuée, puisque elle ne repose plus que sur *tmes* qui est constant, d'après l'énoncé. Évidemment l'allongement de la période peut néanmoins être causé par la charge processeur qui entraîne que le processus prélèvement peut rester un certain temps dans la file des processus prêts, et par un blocage de ce processus si la boîte aux lettres `bal_capteur` est pleine. Si nous avons pu constater que cela n'arrivait pas lorsqu'il n'y a que ces trois processus, nous ne pouvons rien dire en présence d'autres processus qui peuvent retarder les actions de chacun des trois.

4.11.3. Question C

4.11.3.1. Question C.1

Le processus prélèvement devient:

```
processus prélèvement;
     b := ouvrir_boite_aux_lettres ( "bal_capteur");
     répéter
          lecture_prélèvement (Tloc, perte);/* transfert dans le tampon local */
          envoyer (b, Tloc); /* envoi au processus traitement */
     fin répéter
fin processus prélèvement;
```

La procédure `attendre_pendant` n'est plus nécessaire, puisque l'opération `lecture_prélèvement` interdit de lire deux fois les données prélevées à la période.T.

4.11.3.2. Question C.2

On ne permet pas au processus prélèvement d'accéder directement au tampon L, pour deux raisons:

- Ce tampon serait une ressource critique entre le processus système qui assure les prélèvements à la période T et le processus prélèvement. En particulier, il faudrait empêcher le processus système de mémoriser un nouveau prélèvement dans ce tampon si le processus prélèvement est en train d'y accéder. Pour être certain que la durée de cet accès soit bref, il est préférable de l'enfermer dans une opération système qui assurera l'exclusion mutuelle nécessaire de façon la plus brève possible.
- Il faudrait que le processus prélèvement connaisse l'adresse de ce tampon L, ce qui pose des problèmes de définition du lien correspondant. Si le lien est établi à l'édition de lien, un changement de version de système impliquera de ne pas oublier de refaire l'édition de lien de prélèvement. Si le lien est établi dynamiquement, il faut disposer d'une fonction système qui permette d'établir ce lien. Enfin un tel lien signifie qu'un processus peut avoir des accès à certaines parties du système, ce qui peut nuire à la sécurité du système et à sa fiabilité.

4.11.3.3. Question C.3

La difficulté A.2 est résolue, sous la responsabilité du système, c'est-à-dire, si le système effectue bien les prélèvements à la période T, comme indiqué dans l'énoncé.

Remarquons que l'exercice 1.3 montre comment ceci peut être réalisé: un compteur externe au processeur est décrémenté régulièrement par une horloge externe à quartz, par exemple, et provoque une interruption au passage de ce compteur à 0, en même temps que sa réinitialisation. Cette interruption est donc régulière, à la période T. Il suffit qu'elle soit suffisamment «prioritaire» pour que le système la prenne en compte rapidement pour exécuter les prélèvements.

4.11.3.4. Question C.4

Si la lecture des prélèvements n'a pas encore été effectuée lors de l'arrivée de l'interruption d'horloge ci-dessus, cela implique que les données prélevées seront écrasées par les nouvelles. Cela

veut dire que le processus prélèvement a été trop lent à réagir. Dans les solutions précédentes, le ralentissement du processus prélèvement avait simplement pour conséquence l'allongement de la période de prélèvement. Nous avons vu qu'il était difficile de s'en rendre compte avec les moyens dont on disposait. On voit donc que l'indicateur `perte` permet de savoir maintenant que la durée moyenne du cycle dépasse la période.T. La première utilité est en général au niveau des traitements pour lesquels il peut être important de savoir que des données ont été perdues, et éviter de donner des résultats faux. La deuxième utilité est au niveau de l'utilisateur de savoir que la charge du processeur est trop grande pour satisfaire les contraintes. Il peut alors arrêter d'autres processus moins importants, ou revoir la configuration matérielle et logicielle de son installation.

4.12. Communication par boîte aux lettres

On dispose d'un fichier constitué de 100000 blocs (un bloc = un secteur), chaque bloc étant constitué de 10 articles. On désire construire une application qui effectue un traitement sur chacun des articles du fichier. Le schéma général est donné ci-dessous.

```
module de lecture
     numéro_dans_bloc : entier;
     bloc : tableau 1..10 de type_article;
procédure initialiser;
début ouvrir (...); numéro_dans_bloc := 11
fin;
procédure obtenir_suivant (a : type_article);
début si numéro_dans_bloc = 11 alors
          lire bloc;
          numéro_dans_bloc := 1;
       finsi;
       a := bloc [numéro_dans_bloc];
       numéro_dans_bloc := numéro_dans_bloc + 1;
fin;
programme
     a : type_article;
début pour i dans 1 .. 1000000 faire
          obtenir_suivant (a);
          -- traitement de l'article a
       fait;
fin;
```

La lecture d'un bloc dure 20 ms, correspondant à l'attente du secteur, que l'on supposera fixe; elle consomme aussi 0,1 ms d'unité centrale, mais ceci peut être négligé. Le traitement d'un article consomme 2 ms d'unité centrale.

A- Calculer la durée totale d'exécution du programme ci-dessus, avec un seul processus.

B- Le système utilisé est multiprocessus, et propose un mécanisme de communication entre processus par l'intermédiaire d'un tampon de taille fixe selon le schéma producteur consommateur. Deux opérations sur le tampon sont fournies (Il ne s'agit pas de les redéfinir):

`dépôt(a)` met l'article `a` dans une case libre du tampon. S'il n'y a pas de place, le processus est bloqué jusqu'à ce qu'il y en ait. Par ailleurs, lorsque le dépôt a eu lieu, si un processus est en attente de retrait, l'article lui est délivré.

`retrait(a)` extrait un article du tampon et le met dans la variable argument `a`. S'il n'y en a pas, le processus est bloqué jusqu'à ce qu'il y en ait. Par ailleurs, lorsque le retrait a eu lieu, si un processus est en attente de dépôt, il est débloqué.

Remarque 1: Les articles sont extraits dans l'ordre où ils ont été déposés.

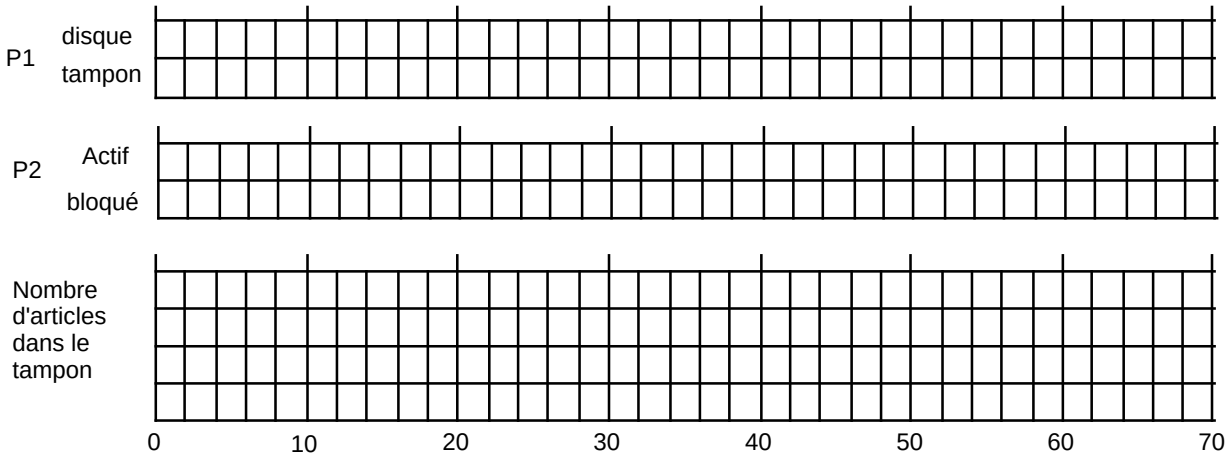
Remarque 2: Le temps d'exécution de ces opérations est considéré comme négligeable.

Pour gagner les temps d'attente des secteurs de l'application ci-dessus, et diminuer ainsi la durée totale d'exécution, on découpe le programme sous forme de deux processus P1 et P2. Le processus P1 est prioritaire sur P2, ce qui veut dire que si les deux processus sont prêts, P1 est activé.

```
P1   a : type_article;
     début pour i dans 1 .. 1000000 faire
               obtenir_suivant (a);
               dépôt (a);
           fait;
     fin;
P2   a : type_article;
     début pour i dans 1 .. 1000000 faire
               retrait (a);
               -- traitement de l'article a
           fait;
     fin;
```

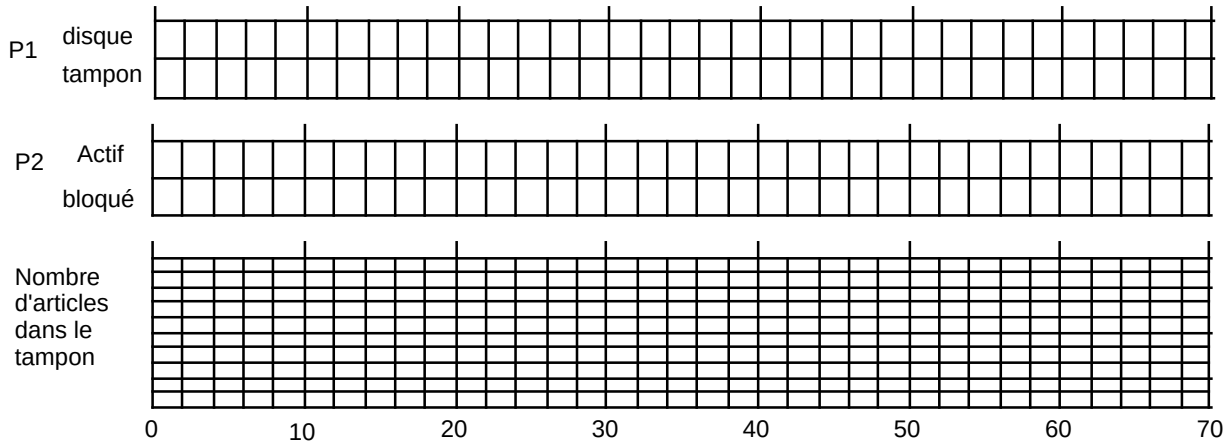
B.1- Le tampon peut contenir 4 articles. Les deux processus sont prêts au même instant initial (temps 0). Décrire ce qui se passe au temps 20 ms, lors de la fin de la première lecture disque, jusqu'au moment où le processeur commence à traiter le premier article.

B.2- Donner sur le graphique ci-après l'évolution temporelle des processus et du nombre de cases occupées dans le tampon durant les 70 premières millisecondes. Etant donné la priorité de P1 et son temps négligeable dans l'état actif, il y a lieu de préciser essentiellement, pour P1, la cause de blocage (attente disque ou attente tampon), et pour P2, les états actif ou bloqué en attente du tampon. Justifier votre réponse.



B.3- Que se passe-t-il au delà? En déduire le temps total.

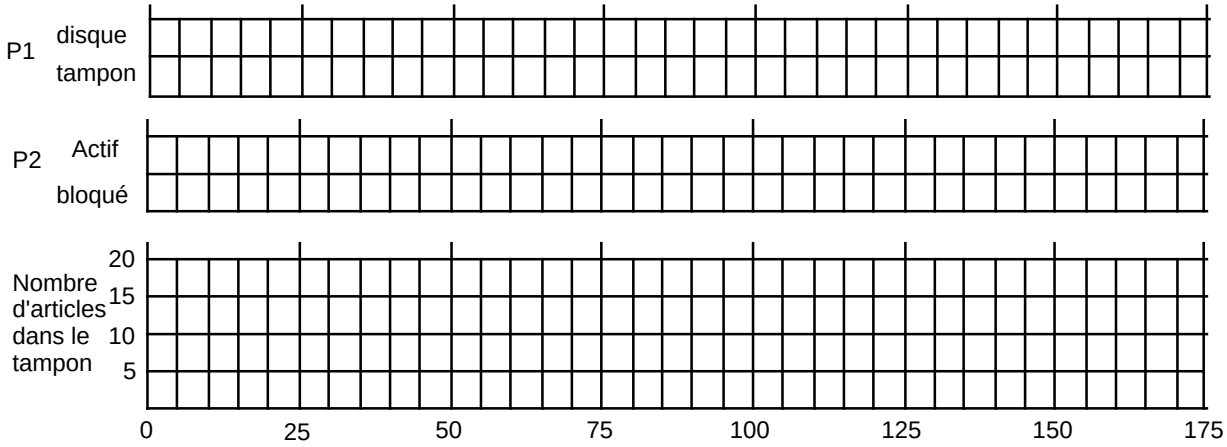
C- Le tampon peut contenir 10 articles. Les deux processus sont prêts au même instant initial (temps 0). Donner sur le graphique ci-après l'évolution temporelle des processus et du nombre de cases occupées dans le tampon. En déduire le temps total.



D- Cela apporterait-il quelque chose d'augmenter la taille du tampon au delà de 10 articles?

E- Le temps de traitement des articles n'est plus fixe de 2 ms, mais de 1 ms par article, auquel il faut ajouter 25 ms tous les 25 articles (notons que la moyenne reste donc à 2 ms par article). Les deux processus sont toujours lancés en même temps. P2 demande une activité processeur de 45 ms

avant de commencer les cycles traitement de 25 articles - traitement de 25 ms. Le tampon peut contenir 20 articles. Donner sur le graphique ci-après l'évolution temporelle des processus et du nombre de cases occupées dans le tampon.



Solution de l'exercice 4.12

4.12.1. Question A

Il y a 100000 blocs à lire, chaque lecture prenant 20 ms, soit au total 2000 secondes. Par ailleurs, il y a 1000000 traitements, chacun d'eux prenant 2 ms, soit au total 2000 secondes. Si les opérations se font avec un seul processus, cela veut dire que ce processus soit effectue les traitements soit attend la fin de la lecture du bloc suivant. L'ensemble durera donc 4000 secondes.

4.12.2. Question B

4.12.2.1. Question B.1

Dès la fin de la lecture, se déroule la séquence suivante :

4 Le processus P1 devient prêt, dépose 4 articles dans le tampon et se bloque le tampon étant plein.

5 P2 devient prêt dès le dépôt du premier article, mais la priorité de P1 l'empêche de passer actif.

6 P2 devient actif lors du blocage de P1. Il extrait un article, ce qui entraîne le déblocage de P1.

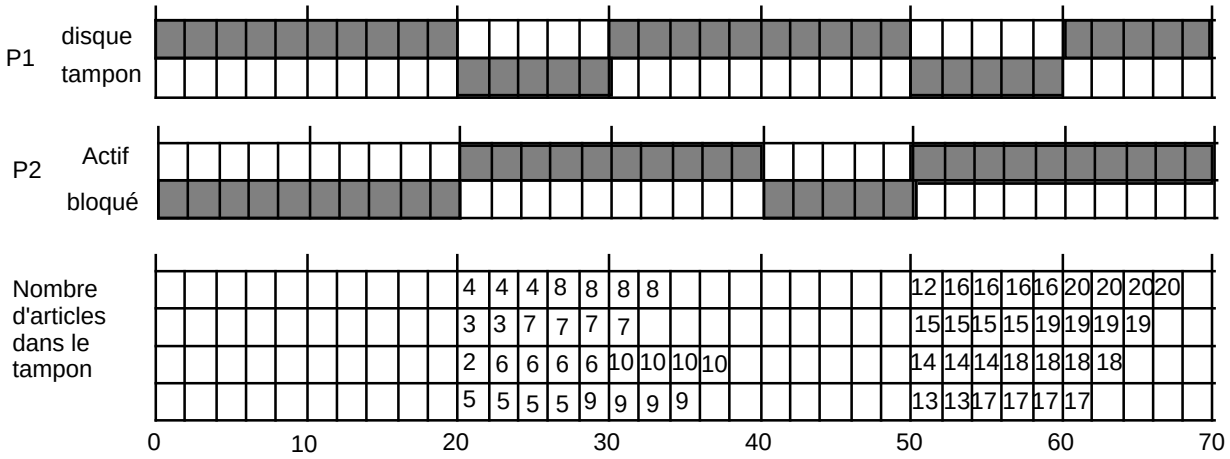
7 P1 dépose le cinquième article et se bloque.

8 P2 repasse actif et commence à traiter le premier article.

4.12.2.2. Question B.2

Au temps 30 ms, l'extraction de l'article 6 par P2 permet le dépôt de l'article par P1 qui lance alors la lecture du bloc suivant et se bloque. Au temps 40 ms, P2 finit de traiter le dixième article et se bloque. Au temps 50 ms, on se retrouve dans la même situation qu'au temps 20 ms : P2 est en attente d'article, et P1 passe prêt avec un bloc complet à traiter.

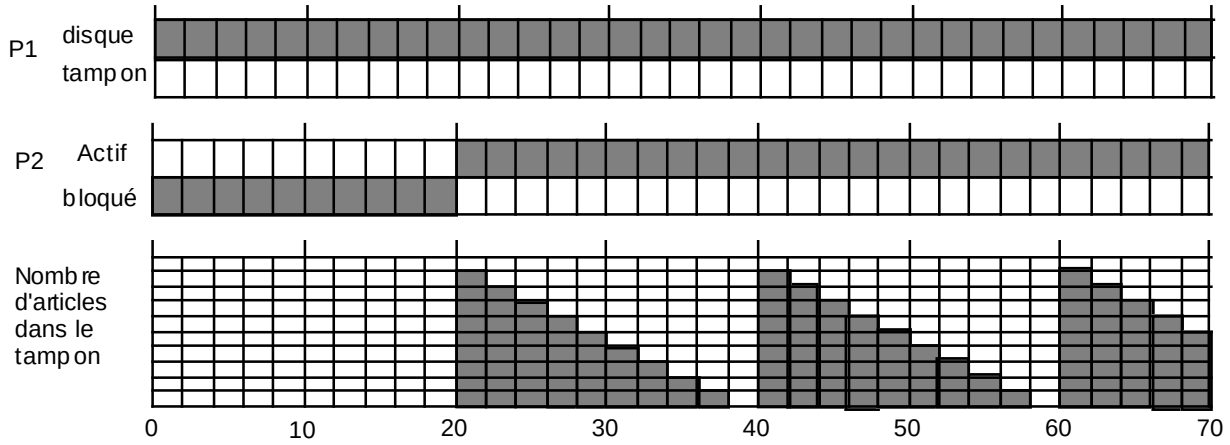
Pour faciliter la compréhension, les numéros des articles sont mis dans les cases du tampon correspondant. Les cases blanches signifient qu'elles sont vides. Notons que l'article 1 et 11 n'apparaissent pas, car ils sortent pratiquement tout de suite du tampon, puisque pour ceux-là, P2 est en attente.



4.12.2.3. Question B.3

Comme nous l'avons dit précédemment, au temps 50 ms, on se retrouve dans la même situation qu'au temps 20 ms : P2 est en attente d'article, et P1 passe prêt avec un bloc complet à traiter. Durant ces 30 ms, 10 articles ont été traités complètement. On peut donc en conclure que la durée totale d'exécution est maintenant de 3000 secondes. Il y a un parallélisme partiel : P1 est en attente de disque et P2 est actif.

4.12.3. Question C



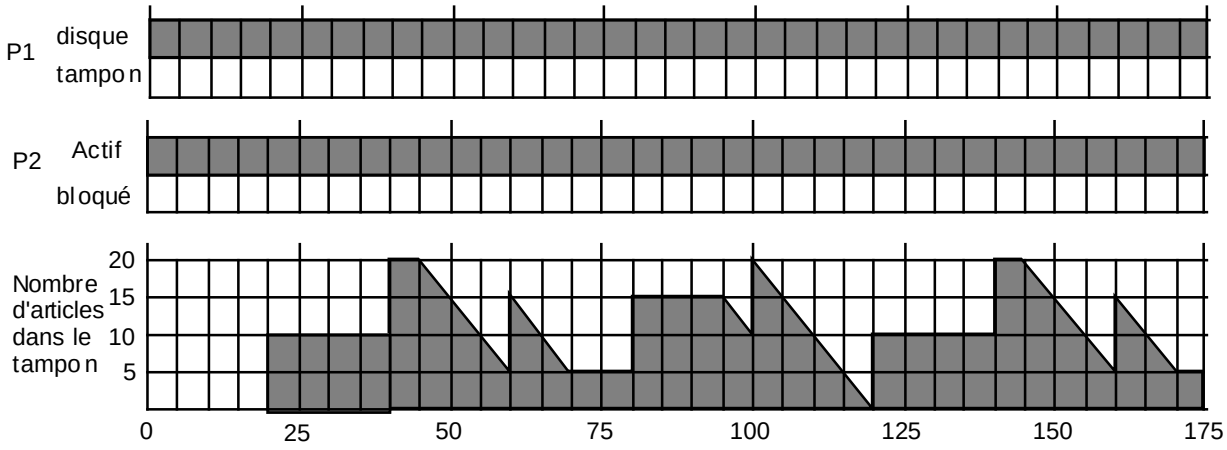
Si on porte la taille du tampon à 10 articles, P1 va pouvoir déposer tous les articles du bloc, pratiquement immédiatement, et relancer la lecture du bloc suivant. Pendant ce temps P2 va traiter les 10 premiers articles. Lorsqu'il aura fini de les traiter, ce sera la fin de la lecture du deuxième bloc, entraînant la réactivation de P1 pour lui permettre de déposer les 10 articles suivants. Nous aurons donc un parallélisme complet, et une durée totale de 2000 secondes. Notons que sur le graphique, le tampon semble ne jamais être plein. En fait il l'est pendant un très court instant, entre le moment où P1 dépose le dernier article du bloc et celui où P2 extrait le premier de ce bloc. Notons que 9 cases seraient sufffisantes : lorsque la neuvième est pleine, P1 se bloquerait, permettant à P2 d'extraire le premier et de le débloquer pour lui permettre de faire son dernier dépôt. En fait, ceci entraînerait une succession de changements de contexte du processeur, et donc une certaine déperdition inutile.

4.12.4. Question D

Il est inutile d'augmenter la taille du tampon, puisque nous avons vu qu'il n'était plein que pendant un très court instant. Les cases supplémentaires ne sont pas nécessaires.

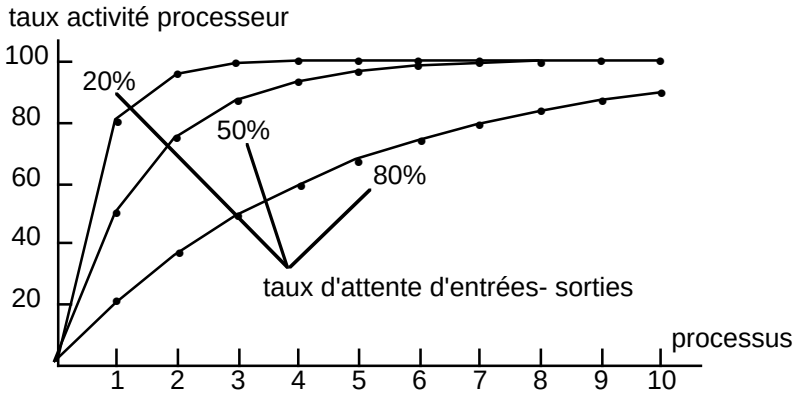
4.12.5. Question E

Au temps 20 ms, le tampon se remplit de 10. Au temps 40 ms, il se remplit de nouveau de 10 et il devient plein. Au temps 45 ms P2 commence à extraire les articles à raison de 1 par ms. Au temps 60 ms, P2 en a traité 15 et il en reste encore 5 dans le tampon lorsque P1 en rajoute 10. Le tampon en contient donc 15. Au temps 70 ms, P2 arrête l'extraction alors que le tampon en contient encore 5. Au temps 80 ms, P1 en remet 10 et le tampon en contient 15. Au temps 95, P2 reprend l'extraction. Au temps 100, le tampon contient 10 articles et P1 en rajoute 10 ; le tampon est plein. Au temps 120, P2 arrête l'extraction (25ième), le tampon est vide et P1 en met 10. Au temps 140, P1 en rajoute 10 et le tampon passe à 20. Nous sommes ramenés à la même situation qu'en 40 : le tampon est plein, P1 lance une lecture et il reste 5 ms d'activité continue pour P2 avant de reprendre l'extraction. On constate que le temps total reste de 2000 secondes, avec un parallélisme complet, la taille du tampon permettant d'amortir les variations du temps de traitement de P2.



4.13. Multiprogrammation et pagination

La figure suivante a été présentée dans le chapitre 14:



A- Expliquez ce dessin (1/2 page).

Pour la suite, on pourra approcher ces courbes par la formule suivante:

$$taux\_cpu = 1 - p^n ,$$

où p est le taux d'attente pour entrées-sorties d'un processus, et n le nombre de processus. On ne vous demande pas de justifier cette formule.

B- On dispose d'une machine équipée d'un système de page à la demande. L'espace mémoire centrale disponible pour les processus est de 800 Ko. Un défaut de page entraîne le blocage du processus en attente de page pendant en moyenne 50 millisecondes.

Les caractéristiques moyennes des programmes de l'installation sont les suivantes:

- Lorsqu'un programme est exécuté seul, dans 800 Ko de mémoire, il n'y a pas de défaut de page, il utilise 40 secondes d'unité centrale, et attend ses entrées-sorties pendant 60 secondes.

- Lorsqu'il est exécuté dans un espace mémoire réduit, les nombres de défauts de page sont les suivants:

espace mémoire	100 Ko	200 Ko	400 Ko
défauts de page	10000	2500	1000

B.1- On répartit équitablement l'espace mémoire centrale disponible entre 2, puis 4, et ensuite 8 processus. Donner dans chaque cas le taux d'attente pour entrées-sorties, et en déduire le taux d'activité du processeur.

B.2- Même question que B.1. si on remplace le disque utilisé pour la pagination (et seulement lui) par un disque plus performant permettant de réduire de 50 à 20 millisecondes le temps d'attente de page.

B.3- Quels commentaires cela vous suggère-t-il, selon que vous preniez le point de vue de l'utilisateur, ou le point de vue du chef d'exploitation?

Solution de l’exercice 4.13

4.13.1. Question A

La multiprogrammation est un concept qui a été introduit pour améliorer l'efficacité de l'utilisation du processeur. En effet, l'exécution d'un programme est une suite de périodes de traitement et d'attente d'entrées-sorties. Lorsque le programme est seul en mémoire, le processeur n'a rien à faire pendant ces attentes d'entrées-sorties. Si plusieurs programmes sont présents en mémoire au même moment, ces attentes pour l'un des programmes peuvent être récupérées pour les autres. Ainsi avec 5 programmes en mémoire, tels que chacun passe 20 % du temps en traitement et 80 % en attente d'entrées-sorties, on peut espérer théoriquement une activité du processeur proche de 100 % répartie entre les 5 programmes. En fait les périodes de traitement et d'attente de ces programmes ne sont pas constantes dans le temps, et la valeur de 80 % d'attente est une valeur moyenne. Lorsqu'on combine ces 5 programmes il y a amélioration, mais pas situation idéale. La figure montre quel est le taux d'activité du processeur obtenu en fonction du nombre de processus mis en mémoire simultanément, pour différentes valeurs du taux d'attente d'entrées-sorties de ces programmes. Par exemple, en prenant 5 programmes qui attendent en moyenne 80 % du temps après des entrées-sorties, la figure indique que, dans ce cas, le taux d'activité du processeur n'est que d'environ 70 %.

Information complémentaire: La formule taux_cpu = 1 - p^n se justifie de la façon suivante: le taux_cpu est en fait la probabilité qu'au moins un programme ne soit pas en attente; la formule découle alors du fait que p^n est la probabilité que tous les n programmes soient en attente.

4.13.2. Question B

4.13.2.1. Question B.1

Lorsqu'on répartit équitablement l'espace mémoire centrale disponible entre n processus, cela signifie que chaque processus s'exécute avec 800/n Ko de mémoire. Il s'ensuit donc des défauts de page pour chacun de ces processus, et donc une augmentation du taux d'attente de ces processus pour entrées-sorties. Les résultats des calculs sont résumés dans le tableau suivant:

Nombre	espace	défauts de page		temps		taux	
processus	mémoire	nombre	attente	E/S	total	attente	taux_cpu
1	800 Ko	0	0	60 sec	100 sec	60 %	40 %
2	400 Ko	1000	50 sec	110 sec	150 sec	73 %	47 %
4	200 Ko	2500	125 sec	185 sec	225 sec	82 %	55 %
8	100 Ko	10000	500 sec	560 sec	600 sec	93 %	44 %

4.13.2.2. Question B.2

La conséquence du remplacement du disque de pagination par un disque à 20 ms., est de diminuer les temps d'attente pour défauts de page des programmes. Il s'ensuit une diminution du temps d'entrées-sorties, et donc une amélioration des taux d'attente de chaque processus, et enfin du taux_cpu. Les résultats des calculs sont résumés dans le tableau suivant:

Nombre	espace	défauts de page		temps		taux	
processus	mémoire	nombre	attente	E/S	total	attente	taux_cpu
1	800 Ko	0	0	60 sec	100 sec	60 %	40 %
2	400 Ko	1000	20 sec	80 sec	120 sec	67 %	55 %
4	200 Ko	2500	50 sec	110 sec	150 sec	73 %	72 %
8	100 Ko	10000	200 sec	260 sec	300 sec	87 %	66 %

4.13.2.3. Question B.3

Prenons d'abord le point de vue du chef d'exploitation. Dans chacun des cas, il constate que le meilleur taux d'activité du processeur est obtenu avec 4 programmes présents simultanément en mémoire. Par rapport à la monoprogrammation qui donne un taux d'activité du processeur de 40 %, la rentabilité de son système passe à 55 % dans le cas de disques à 50 ms., et à 72 % s'il dispose de disques de pagination à 20 ms. Il ne devrait pas s'étonner que la rentabilité diminue avec 8 programmes simultanés, car cela indique simplement que les programmes n'ont plus assez d'espace mémoire pour s'exécuter sans trop de défauts de page. Il y a début d'écroulement.

Le point de vue de l'utilisateur ne sera pas aussi favorable dans une première approche. En effet, il constate que son programme devait rester 100 secondes en mémoire dans le cas de la monoprogrammation, alors qu'il doit rester au moins 225 secondes en B.1, et 150 en B.2. Expérimentalement il constatera que ces deux derniers temps seront en fait plus grands. En effet, les 4 programmes ont besoin de 160 secondes de processeur au total; le taux_cpu étant alors de 55 % dans le cas B.1, la durée totale d'exécution de ces 4 programmes est de 290 secondes (160/0.55). Dans le cas B.2, le taux_cpu passant à 72 %, la durée totale d'exécution sera alors de 222 secondes (160/0.72). Dans chaque cas, le temps de réponse se dégrade donc. Dans une deuxième approche, il constatera que si les 4 programmes sont pour lui, l'amélioration est assez sensible, puisque les 4 programmes demandent 400 secondes en monoprogrammation! Il ne pourra alors que féliciter le chef d'exploitation d'avoir amélioré ainsi les temps de réponse globaux.

4.14. Page à la demande et allocation processeur

On dispose d'un système monoprocesseur en multiprogrammation, doté d'une mémoire virtuelle paginée. Les processus se partagent, outre le processeur, un disque comme ressource. Les transferts disque-mémoire centrale sont assurés par un processeur spécialisé.

A- Donner une définition de la multiprogrammation. Dire pourquoi elle peut être réalisée dans notre système et expliquez-en les avantages et les inconvénients en au plus 20 lignes (on pourra prendre le point de vue de l'ingénieur système et celui de l'utilisateur)

B.- La mémoire physique est une suite d'emplacements, découpée en blocs de taille fixe, appelés cases et repérés chacun par un numéro. La mémoire linéaire (mémoire virtuelle) de chaque processus est découpée en blocs de taille fixe (celle d'une case), appelés pages et repérés par un numéro de page virtuelle. Une image de la mémoire virtuelle d'un processus est maintenue sur disque par le système (espace de pagination).

Une adresse a la structure suivante :

bit 31 bit 12	bit 11 bit 0
numéro de page virtuelle	déplacement dans la page

A chaque processus est associée une table des pages, dont la ième entrée décrit la ième page virtuelle du processus :

bit 31	bit 30	bit 29	bit 28 bit 20	bit 19 bit 0
présence	modifiée	accédée	protection	n° de case

- *présence* vaut 1 si la page est présente en mémoire centrale, 0 sinon

- *modifiée* vaut 1 si la page a été modifiée depuis son chargement, 0 sinon
- *accédée* indique si la page a été référencée depuis un certain temps
- *protection* indique les droits d'accès du processus à la page
- *n° de case* indique le numéro de case de mémoire physique où se trouve la page virtuelle si elle est présente en mémoire centrale.

Dans le contexte de chaque processus, un registre repère le début de la table des pages.

B.1- Décrire sous forme algorithmique, les opérations réalisées lors du décodage d'une adresse. La table des pages d'un processus est résidente en mémoire centrale. Applications :

- un processus référence l'adresse 10/3000
- l'entrée 10 indique *présence* =1 *n° case* =20. Donner l'adresse physique correspondante.
- puis il référence l'adresse 15/2500
- l'entrée 15 indique *présence* =0 *n° case* =12. Peut-on déduire immédiatement l'adresse physique correspondante? Sinon que doit faire le système? (expliquez très succinctement).

B.2- Comment peut-on utiliser le bit *page modifiée*? On envisagera les cas où la page virtuelle est remplacée et celui où le processus se termine.

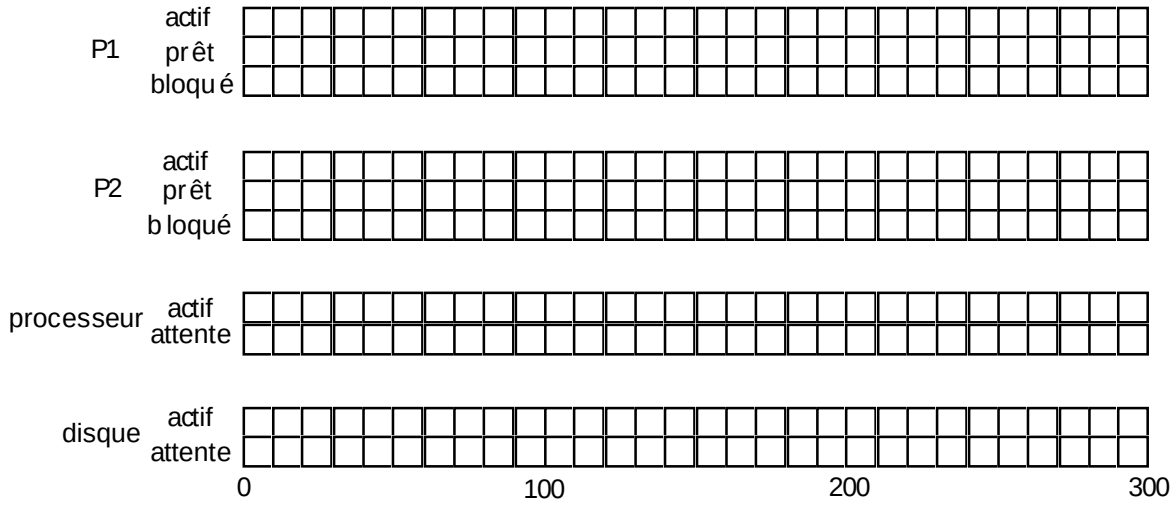
B.3- Le bit *page accédée* est positionné à 1 à chaque référence de la page par le processus, et remis périodiquement à 0 par le système. Quel algorithme de remplacement de page utilise cette information?

C- Le système comporte deux processus ayant chacun trois pages de mémoire virtuelle. A la création d'un processus, le système lance une phase d'initialisation de ce processus sans accéder à la mémoire virtuelle. Le système de pagination est à la demande, c'est-à-dire qu'initialement, aucune page n'est en mémoire centrale. Lors d'un défaut de page, le système initialise un transfert disque de la page vers une case, réalisé par le processeur spécialisé, qui dure 20 ms. On supposera, pour simplifier, que la mémoire centrale contient suffisamment de cases pour les six pages des processus. Lorsqu'un processus se termine, le système réécrit chacune de ses pages sur disque, ce transfert requiert 20ms par page.

L'allocation du processeur est faite selon l'ancienneté dans la file des processus dans l'état "prêt" et les requêtes disque sont gérées à l'ancienneté; en cas d'égalité P1 est prioritaire sur P2. Une seule requête disque peut être traitée à la fois. Le comportement des processus est le suivant :

```
P1   Création du processus pendant 10ms
     Calcul utilisant la page 1 pendant 40ms
     Calcul utilisant les pages 1 et 2 pendant 10ms
     Calcul utilisant la page 2 pendant 20ms
     Calcul utilisant la page 3 pendant 30ms
     Fin P1
P2   Création du processus pendant 10ms
     Calcul utilisant la page 3 pendant 30ms
     Calcul utilisant la page 2 pendant 10ms
     Calcul utilisant les pages 2 et 1 pendant 20ms
     Fin P2
```

On crée le processus P1, puis le processus P2 après 10ms. Etablir le chronogramme des deux processus, du disque et du processeur sur le diagramme ci-dessous, et justifier votre réponse.



Solution de l'exercice 4.14

4.14.1. Question A

La multiprogrammation est le fait d'avoir plusieurs processus (correspondant souvent à plusieurs programmes) simultanément présents en mémoire centrale. Pour pourvoir la mettre en œuvre, il faut pouvoir garantir que ces processus puissent vivre en bonne intelligence, c'est-à-dire, évoluer chacun de leur côté sans se perturber les uns les autres. Dans notre système, les entrées-sorties sont gérées par un processeur spécialisé, ce qui permet libère le processeur central pendant la durée des transferts et lui permet de travailler pour le compte d'un autre processus.

Du point de vue de l'ingénieur système, l'avantage principal est évidemment une meilleure utilisation du processeur et donc une amélioration du temps moyen de réponse de l'ensemble des travaux. Du point de vue de l'utilisateur, deux aspects interviennent. Soit la machine est libre au moment où il soumet sa demande et le temps de réponse sera meilleur s'il est seul présent en mémoire. Par contre si la machine n'est pas libre, en monoprogrammation il devra d'abord attendre que tous les programmes arrivés avant lui soient terminés, alors qu'en multiprogrammation il pourra utiliser le processeur pendant les périodes d'inactivités des programmes arrivés avant lui.

Le système d'exploitation devient cependant plus complexe et occupe une part du temps processeur pour sa gestion (déperdition) : gestion de l'allocation du processeur, gestion du partage de ressources, gestion des défauts de page, gestion de la mémoire centrale, etc.

4.14.2. Question B

4.14.2.1. Question B.1

L'opération de décodage d'adresse peut se représenter comme suit :

```
Procédure décodage (var adreel : adresse ; advirt :adresse) est
var A : adresse ;
début A := table_de_page (advirt.numéro_page_virtuelle) ;
       si non A.présence alors traitement défaut de page ; finsi ;
       adreel.déplacement := advirt.déplacement ;
       adreel.numéro_de_page := A.n°_de_case ;
fin ;
```

En appliquant cet algorithme à l'adresse 10/3000, avec une entrée 10 de la table des pages telle que `présence`=1 et `n°_de_case`=20, il n'y a pas de défaut de page, et l'adresse réelle résultante est 20/3000.

En appliquant cet algorithme à l'adresse 15/2500, avec une entrée 15 de la table des pages telle que `présence`=0 et `n°_de_case`=15, il y a défaut de page, et l'on ne peut savoir actuellement ce que sera l'adresse réelle. Le n° de case associé n'a actuellement aucune signification : c'est sans doute la dernière case associée à cette page, mais elle ne l'est plus maintenant. Le système, pour traiter ce

défaut de page, doit allouer une nouvelle case à cette page, charger son contenu depuis le disque et réinitialiser l'entrée correspondante de la table des pages avant de relancer l'opération de décodage.

4.14.2.2. Question B.2

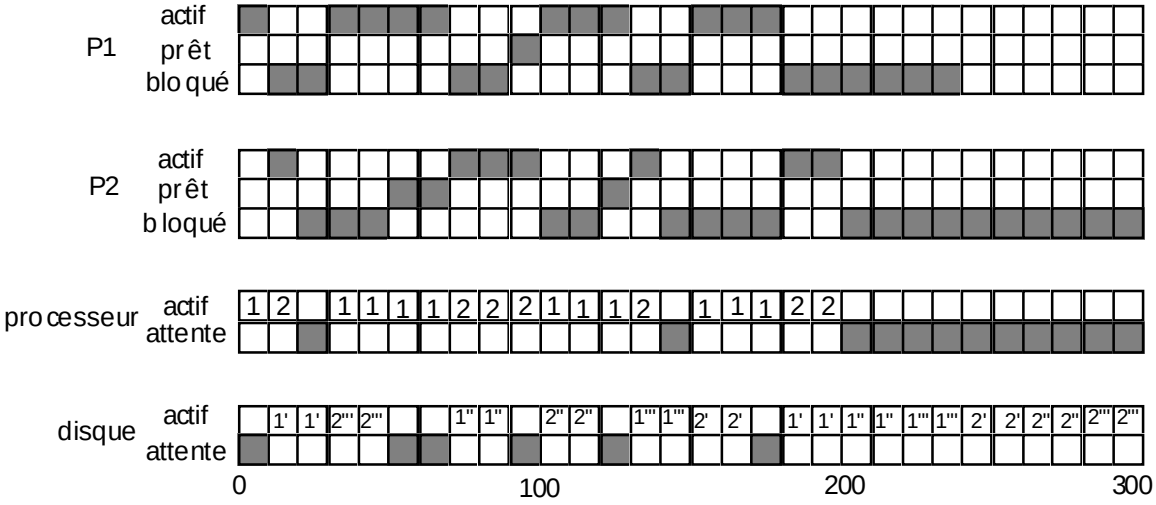
Le bit *modifiée* indique que la page a été modifiée depuis qu'elle a été chargée en mémoire. En d'autres termes, ce bit est mis à 1 chaque fois que le processus référence la page en écriture. Lorsque la page est remplacée, ce bit permet de savoir si elle doit être réécrite ou non sur disque avant de charger celle qui la remplace dans la case. Lorsque le processus se termine, les cases qu'il occupait en mémoire centrale sont libérées. Ce bit permet de savoir s'il est nécessaire de réécrire les pages correspondantes sur disque avant leur libération, pour ne pas perdre les dernières modifications, du moins lorsqu'elles doivent survivre à la fin d'exécution du processus (elles correspondent à des objets externes).

4.14.2.3. Question B.3

Le bit *accédée* indique que la page a été référencée récemment. Ce bit est mis à 1 lors de chaque accès à la page, et remis à 0 de temps en temps par l'algorithme de remplacement de page. Deux algorithmes utilisant ce bit ont été présentés en §14.3.4 : l'algorithme de la seconde chance et NRU. L'algorithme de la seconde chance est une modification de l'algorithme FIFO : au lieu de remplacer systématiquement la page la plus ancienne en mémoire, on ne la remplace que si elle n'a pas été référencée dans un passé récent. L'algorithme NRU tente d'approcher l'algorithme LRU, en dressant régulièrement les listes des pages ayant été référencées dans la période récente.

4.14.3. Question B

Voici le graphe, suivi de la chronologie des événements.



À 10, P1 accède à sa page 1, ce qui provoque un défaut de page, et P1 se bloque en attendant la fin de la lecture (fin en 30). P2 devient actif.

À 20, P2 accède à sa page 3, ce qui provoque un défaut de page, et P2 se bloque en attendant la fin de la lecture, mais le disque est occupé par la lecture de la page 1 de P1.

À 30, la page 1 de P1 étant lue, P1 redevient actif, jusqu'à son prochain défaut (en 70), et la lecture de la page 3 de P2 commence (fin en 50).

À 50, P2 devient prêt.

À 70, P1 accède à sa page 2, ce qui provoque un défaut de page et bloque P1 en attendant la fin de la lecture (fin en 90) ; P2 devient actif, jusqu'à son prochain défaut (en 100).

À 90, P1 devient prêt.

À 100, P2 accède à sa page 2, ce qui provoque un défaut de page et bloque P2 en attendant la fin de la lecture (fin en 120) ; P1 devient actif, jusqu'à son prochain défaut (en 130).

À 120, P2 devient prêt.

À 130, P1 accède à sa page 3, ce qui provoque un défaut de page et bloque P1 en attendant la fin de la lecture (fin en 150) ; P2 devient actif, jusqu'à son prochain défaut (en 140).

À 140, P2 accède à sa page 1, ce qui provoque un défaut de page et bloque P2 en attendant la fin de la lecture, mais le disque est occupé par la lecture de la page 3 de P1.

À 150, la page 3 de P1 étant lue, P1 redevient actif, jusqu'à sa terminaison (en 180), et la lecture de la page 1 de P2 commence (fin en 170).

À 170, P2 devient prêt.

À 180, P1 se termine, et les 3 pages de P1 doivent être réécrites sur le disque (fin en 240) ; P2 devient actif jusqu'à sa terminaison (en 200)

À 200, P2 se termine, et les 3 pages de P2 doivent être réécrites sur le disque, qui est occupé avec la réécriture de celles de P1.

À 240, les pages de P1 sont toutes réécrites, et on commence la réécriture des 3 pages de P2 (fin en 300).

À 300, toutes les pages de P2 ont été réécrites sur le disque.

4.15. Algorithmes de pagination

On dispose d'un système doté d'une pagination à la demande, suivant deux algorithmes A_1 et A_2. Au cours de son exécution, un programme accède successivement aux pages 1, 5, 2, 5, 1, 4, 1, 5, 3. Le système alloue à ce programme un espace de trois pages.

Avec l'algorithme A_1, on constate que l'on a successivement en mémoire les pages suivantes:

```
1    1    1    1    1     2    1    1    1
     5    2    2    2     4    2    4    3
               5    5    5    5    4    5    5
```

Avec l'algorithme A_2, on constate que l'on a successivement en mémoire les pages suivantes:

```
1    1    1    1    1    1    1    1    1
     5    2    2    2    4    4    4    3
               5    5    5    5    5    5    5
```

A- A votre avis, lequel des deux algorithmes correspondrait à l'algorithme FIFO, et lequel correspondrait à LRU? Justifiez votre raisonnement.

B- Déterminer dans chacun des cas le nombre de défauts de pages.

Solution de l'exercice 4.15

4.15.1. Question A

Le principe d'une pagination à la demande est le suivant: le système attribue au processus un certain nombre de pages (ici 3) lorsque le processus tente d'accéder à une page qui n'est pas présente en mémoire, il se produit un déroutement, et le système choisit suivant un algorithme de remplacement déterminé, la page qui doit être supprimée de la mémoire pour y être remplacée par celle qui est demandée. L'algorithme FIFO remplace la page qui est la plus ancienne en mémoire, alors que l'algorithme LRU remplace la page qui est la plus ancienne référencée.

Dans le cas de l'algorithme A_1, on constate que 4 remplace 1 qui est la plus ancienne présente en mémoire et la plus récemment référencée, il ne peut donc s'agir de LRU. En poursuivant l'analyse, 1 remplace 5 qui est maintenant la plus ancienne, puis 5 remplace 2, troisième entrée, et enfin 3 remplace 4, quatrième entrée. L'algorithme A_1 peut donc être FIFO.

Dans le cas de l'algorithme A_2, on constate que 4 remplace 2 dernière amenée en mémoire, ce ne peut donc être FIFO. Par contre 2 est alors la plus ancienne référencée, puisque les dernières références sont 1 puis 5. De même par la suite, 3 remplace 4 qui est aussi à ce moment la plus ancienne référencée. L'algorithme A_2 peut donc être LRU. Notons que cet algorithme pourrait aussi être LFU.

4.15.2. Question B

Les défauts de pages sont les déroutements provoqués par le processus qui référence une page qui n'est pas présente en mémoire. Le défaut entraîne un remplacement, donc un changement de l'ensemble des pages présentes en mémoire.

Avec l'algorithme A_1, les changements sont les suivants, indiqués par *:

```
*     *     *                 *     *     *     *
1     1     1     1     1     2     1     1     1
      5     2     2     2     4     2     4     3
            5     5     5     5     4     5     5
```

Il y a donc 7 défauts de pages.

Avec l'algorithme A_2, les changements sont les suivants, indiqués par *:

```
*     *     *                 *                 *
1     1     1     1     1     1     1     1     1
      5     2     2     2     4     4     4     3
            5     5     5     5     5     5     5
```

Il y a donc 5 défauts de pages.

4.16. Algorithmes de pagination

On s'intéresse aux systèmes de pagination à la demande.

A- Expliquer brièvement le principe de la pagination à la demande. Détailler les algorithmes FIFO et LRU. (au plus 20 lignes)

B- Au cours de son exécution, un programme accède successivement aux pages 0, 1, 4, 2, 0, 1, 3, 0, 1, 4, 2, 3.

B.1- On utilise l'algorithme FIFO et le système alloue à ce programme un espace de 3 pages (initialement vides). Donner la suite des pages présentes en mémoire et en déduire le nombre de défauts de page.

B.2- On utilise l'algorithme FIFO et le système alloue à ce programme un espace de 4 pages (initialement vides). Donner la suite des pages présentes en mémoire et en déduire le nombre de défauts de page.

B.3- On utilise l'algorithme LRU et le système alloue à ce programme un espace de 3 pages (initialement vides). Donner la suite des pages présentes en mémoire et en déduire le nombre de défauts de page.

B.4- On utilise l'algorithme LRU et le système alloue à ce programme un espace de 4 pages (initialement vides). Donner la suite des pages présentes en mémoire et en déduire le nombre de défauts de page.

Solution de l’exercice 4.16

4.16.1. Question A

Le principe de la pagination à la demande est le suivant. Le système attribue au processus un certain nombre de cases (ici 3 ou 4) qui peuvent recevoir des pages. Initialement toutes les cases sont vides. Lorsque le processus tente d'accéder à une page qui n'est pas présente en mémoire (elle n'est pas dans une case), il se produit un déroutement. S'il reste des cases libres, le système en choisit une quelconque, et y met la page demandée. S'il n'en reste pas, il choisit une case occupée par une page suivant un algorithme de remplacement déterminé, retire la page de la case et la remplace par celle qui est demandée.

L'algorithme de remplacement peut être FIFO. Dans ce cas, le système choisit la case qui contient la page la plus ancienne présente en mémoire.

L'algorithme de remplacement peut être LRU. Dans ce cas, le système choisit la case qui contient la page qui est la plus ancienne référencée parmi celles présentes en mémoire.

4.16.2. Question B

4.16.2.1. Question B.1

Partant d'un espace de 3 pages initialement vide, la suite des pages présentes en mémoire est la suivante, où les colonnes représentent les configurations successives. Une '*' dans une colonne indique un défaut de page. Cela se présente lorsque la page référencée n'est pas dans la colonne précédente.

référence	0	1	4	2	0	1	3	0	1	4	2	3
défauts	*	*	*	*	*	*	*			*	*	
	0	0	0	2	2	2	3	3	3	3	3	3
		1	1	1	0	0	0	0	0	4	4	4
			4	4	4	1	1	1	1	1	2	2

L'algorithme FIFO implique un remplacement dans l'ordre où les pages sont amenées en mémoire, c'est-à-dire dans l'ordre 0, 1, 4, 2, 0, 1, 3, 4, 2. En comptant les '*', on constate qu'il y a 9 défauts de page.

4.16.2.2. Question B.2

Partant d'un espace de 4 pages initialement vide, la suite des pages présentes en mémoire est la suivante, où les colonnes représentent les configurations successives. Une '*' dans une colonne indique un défaut de page. Cela se présente lorsque la page référencée n'est pas dans la colonne précédente.

référence	0	1	4	2	0	1	3	0	1	4	2	3
défauts	*	*	*	*			*	*	*	*	*	*
	0	0	0	0	0	0	3	3	3	3	2	2
		1	1	1	1	1	1	0	0	0	0	3
			4	4	4	4	4	4	1	1	1	1
				2	2	2	2	2	2	4	4	4

L'algorithme FIFO implique un remplacement dans l'ordre où les pages sont amenées en mémoire, c'est-à-dire dans l'ordre 0, 1, 4, 2, 3, 0, 1, 4, 2, 3. En comptant les '*', on constate qu'il y a 10 défauts de page.

Remarque (hors sujet): on constate que le processus se comporte moins bien que précédemment, alors qu'il a plus d'espace. Cette anomalie est connue sous le nom d'anomalie de Belady.

4.16.2.3. Question B.3

Partant d'un espace de 3 pages initialement vide, la suite des pages présentes en mémoire est la suivante, où les colonnes représentent les configurations successives. Une '*' dans une colonne indique un défaut de page. Cela se présente lorsque la page référencée n'est pas dans la colonne précédente.

référence	0	1	4	2	0	1	3	0	1	4	2	3
défauts	*	*	*	*	*	*	*			*	*	*
	0	0	0	2	2	2	3	3	3	4	4	4
		1	1	1	0	0	0	0	0	0	2	2
			4	4	4	1	1	1	1	1	1	3

L'algorithme LRU implique lors de chaque remplacement de rechercher la page la plus anciennement référencée, parmi celles de la colonne précédente. Par exemple, la deuxième référence à la page 4 entraîne le remplacement de la page 3. En comptant les '*', on constate qu'il y a 10 défauts de page.

4.16.2.4. Question B.4

Partant d'un espace de 4 pages initialement vide, la suite des pages présentes en mémoire est la suivante, où les colonnes représentent les configurations successives. Une '*' dans une colonne

indique un défaut de page. Cela se présente lorsque la page référencée n'est pas dans la colonne précédente.

```
référence     0    1    4    2    0    1    3    0    1    4    2    3
défauts       *    *    *    *              *                   *    *    *
              0    0    0    0    0    0    0    0    0    0    0    3
                   1    1    1    1    1    1    1    1    1    1    1
                        4    4    4    4    3    3    3    3    2    2
                             2    2    2    2    2    2    4    4    4
```

Dans ce cas, la première référence à la page 3 implique le remplacement de la page 4, et la deuxième référence à la page 4 implique le remplacement de la page 2. En comptant les '*', on constate qu'il y a 8 défauts de page.

4.17. Choix de page remplacée

On s'intéresse à la pagination à la demande. Le système dispose de 4 cases qui sont toutes occupées, le tableau donnant ci-dessous, pour chacune d'elles, la date en microsecondes du chargement de la page qu'elle contient, la date en microsecondes du dernier accès à cette page et l'état des indicateurs de la case, `accédée`, `modifiée` et `présence`.

Case	Chargement	Accès	`accédée`	`modifiée`	`présence`
0	126	279	0	0	1
1	230	260	1	0	1
2	120	272	1	1	1
3	160	280	1	1	1

En justifiant votre réponse, donner quelle sera la page remplacée pour chacun des 4 algorithmes de remplacement suivants : LRU, FIFO, seconde chance et NRU.

Solution de l'exercice 4.17

Dans l'algorithme LRU, on prend la page la moins récemment référencée. Il s'agit donc de prendre la dernière selon le critère de la colonne *Accès*, c'est-à-dire, celle de la case 1 qui a été accédée en 260.

Dans l'algorithme FIFO, on prend la page qui est en mémoire depuis le plus longtemps. Il s'agit donc de suivre le critère de la colonne *Chargement*, c'est-à-dire, celle de la case 2 qui est en mémoire depuis le temps 120.

Dans l'algorithme de la seconde chance, on prend la page qui est en mémoire depuis le plus longtemps, donc selon le critère de la colonne *Chargement*, sauf si son bit `accédée` est à 1, auquel cas on le remet à 0 et on poursuit le recherche dans l'ordre. Dans l'exemple, la case 2 est la plus ancienne, mais son bit `accédée` est à 1. La suivante dans l'ordre est la case 0 dont le bit `accédée` est à 0. C'est donc celle qui est choisie.

Dans NRU, les pages sont classées dans 6 catégories selon l'état des bits `accédée`, `modifiée` et `présence`. Les deux catégories correspondantes au bit de `présence` à 0 sont vides, dans ce cas. La catégorie suivante à considérée est celle où les bits `accédée` et `modifiée` sont tous deux à 0. Cette catégorie ne contenant qu'une seule case, la case 0, c'est celle qui est choisie.