The FreeBSD Process Scheduler

By Jon P. Giza
Milwaukee School of Engineering
For Dr. Mark Sebern
January 29, 1998

Table of Contents

I Introduction

II FreeBSD Startup

III FreeBSD Process Scheduling

1 Scheduling Algorithm

2 Priority Calculation

3 FreeBSD Interrupts

4 Context Switching

IV Error Handling

V Conclusion

VI Appendices

1 Appendix A: Process Priority Values and Descriptions

2 Appendix B: Diagram of a Process in Memory or Storage

VII Bibliography

Introduction

A process is any program or task in execution. In a multi-user operating system like FreeBSD, there can be many different programs trying to execute at the same time. Programs called by users start from a parent process, and could possibly spawn many other child processes. It is the job of the operating system, to rotate through the waiting processes so that all processes complete. Obviously, it is not an option for the operating system to take the processes in chronological order, and wait for each to complete before executing the next process. So in order to let all users and processes have equal time on the processor the operating system must schedule a rotation between waiting processes. The process scheduler can operate on numerous different algorithms, and here the design of the 4.4BSD operating system will be discussed.

FreeBSD Startup

When a FreeBSD system boots, there are three processes that are vital to the operating system:

1. Swapper (PID = 0)
2. PageDaemon (PID = 2)
3. Init (PID = 1)

The swapper and the pagedaemon are kernel processes, which means they operate with the kernel's execution mode. The *init* process runs just like any other user process, but

has a much higher priority. *Init* is the workhorse of the UNIX operating system, it is responsible for spawning all processes for remote terminals and changing run-levels from multi-user to single-user. The *init* process can also take on the role of parent process in case any child process becomes orphaned. The swapper kernel process is for switching processes from main memory to long-term storage. The pagedaemon is the process that does the conversion for addresses to support the virtual memory subsystem. After the system completes the startup script, future processes are created using the *exec* system call. Each process that is created has a priority associated with it that is used in scheduling the order in which processes are executed.

FreeBSD Process Scheduling

The 4.4BSD kernel uses a priority-based scheduling algorithm that is biased to promote interactive processes which typically come from users. FreeBSD does not do deadline scheduling, also known as real-time scheduling. When a process is first created, it is initially given a high priority by the scheduler. This gives the process an opportunity to complete quickly if it is a short-burst process. The time given to each process by the scheduler is called the quantum. For a 4.4BSD system, the quantum is set to a time specific to each machine. The calculation of the quantum is shown later on, as is dependent on the system clock speed and a fixed value of 100 milliseconds. This is the same quantum time that has been used for over fifteen years by the designers of this operating system. If a process with a high priority runs through its specified quantum without completing, it's priority is down-graded to allow other processes to run.

Processes that require a large amount of CPU time or a large block of memory can also be degraded in priority. This scheduling policy allows for long-running processes to be run when there is time, and user processes like editors to run when necessary.

The Scheduling Algorithm

The scheduling of processes is a very complex task that is done by a number of scheduling techniques. Processes in a 4.4BSD system are held in a number of queues like shown below:

Queue 1	**Queue 2**	**Queue 3**	**…Queue 32**
Header Info	Header Info	Header Info	Header Info
Process A Process C Process I	Process B Process F Process G	Process D Process E Process H	Process n-2 Process n-1 Process n

In this example, there are three queues each holding three processes. Currently, processes A, C, and I are all at the same priority level. This means that they are all of equal importance, and therefore, they are give equal time slices in the kernel. The processes in queue's 3 and 4 are lower priority because they are either long-running processes, they are in sleep mode, or they have been specified as a low-priority process. The processes in queue 1 are rotated in and out of the kernel mode by a round-robin

scheduling function that rotates processes every once every time quantum as seen here in the source code:

```
#define ROUNDROBIN_INTERVAL (hz / quantum)
int roundrobin_interval(void)
{
	return ROUNDROBIN_INTERVAL;
}
```

This function returns an integer value that is hz divided quantum. The hz is an integer value that represents the system clock rate, and the quantum is a variable usually set to 100 milliseconds.

```
static void roundrobin(arg)
	void *arg;
{
	struct proc *p = curproc;
```

The object p is a process that used to represent the current process

```
#ifdef SMP
	need_resched();
	forward_roundrobin();
#else
	if (p == 0 || RTP_PRIO_NEED_RR(p->p_rtprio.type))
		need_resched();
```

need_resched() is called if the process has completed, or if the quantum has expired and the process did not complete.

```
#endif
	timeout(roundrobin, NULL, ROUNDROBIN_INTERVAL);
}
```

Timeout changes the current process at a time interval based on the ROUNDROBIN_INTERVAL

This function very simply rotates between the processes in the current queue and then runs the function need_resched() based on the output. If the output is zero, meaning the process terminated, or if the process exceeded the quantum, the operating system must reset the priority on that process and then continue onto the next process. The round-

robin scheduler stays in the queue for the processes with the highest priority, and it is the processes that leave the queue either by completing, or by being down-graded.

The following function runs through a case statement to determine the current state of the process “p” in the scheduling queue.

```
void
setrunnable(p)
        register struct proc *p;
{
        register int s;
        s = splhigh();              // Set the Interrupt mask to block all
        switch (p->p_stat) {        // Case statement based on p’s status
        case 0:
        case SRUN:
        case SZOMB:
        default:
                panic("setrunnable");
        case SSTOP:
        case SSLEEP:                // Awaken the process to it’s status can be updated
                unsleep(p);
                break;
        }
        p->p_stat = SRUN;           // **The important part of the function** \\
        if (p->p_flag & P_INMEM)    /* If the process can be run and it is already in
        setrunqueue(p);                     memory, then move it to the runqueue */
        splx(s);                    // Re-enable the interrupts
        if (p->p_slptime > 1)       /* If the process has been sleeping for 1 second
                updatepri(p);               Set the priority, and return to wait status */
        p->p_slptime = 0;
}
```

Priority Calculation

The actual priority value for a user process is calculated using the following formula:

$$p_usrpri = PUSER + \left[\frac{p_estcpu}{4}\right] + 2(p_nice)$$

The variables in this equation are defined below:

PUSER: The value of the process when not in kernel mode which ranges from 50 to 127 with lower numbers being higher priority (See Table 1 in Appendix A)

p_estcpu: An estimation of the previous usage of the CPU by the process during it's last time slice.

p_nice: A user-specifiable value for boosting a processes priority.

The value for p_estcpu is incremented every clock tick for which it is active in the kernel. It is also decreased every second using this formula:

$$p_estcpu = \frac{2 * load}{2 * load + 1} p_estcpu + p_nice$$

By this equation, the value for p_estcpu is decreased by 90% in approximately five seconds. By decreasing the value, it allows the process to be upgraded into the running queue with a higher priority. Processes can also be upgraded in priority by coming out of a sleep routine, or by an interrupt.

FreeBSD Interrupts

The most important interrupts for an operating system are the ones generated by the system clock. These interrupts are created on nearly exact intervals and are used in

the synchronization of the entire operating system. The frequency of the clock ticks vary between different systems, so it is necessary for the operating system to know the type of processor and it's clock rate. In FreeBSD there is an Interrupt Service Routine called hardclock() which is given a very high priority. Given the extremely fast frequency of clock ticks, it is imperative that the hardclock() routine finish very quickly so that no ticks are missed. Hardclock() is responsible primarily responsible for updating the time of day, and signaling the operating system to perform a context switch and allow another process to begin execution. The hardclock() routine also starts another function called softclock(). This function does not have the same priority as hardclock(), so it does not block as many processes as hardclock(). Softclock() is responsible for the following tasks:

1. Retransmission of dropped network packets
2. Watching peripherals for other signals
3. System process rescheduling events

Other interrupts that are generated are handled according to the following table. Processes and the operating system both use interrupts for certain tasks. Processes need to block certain interrupts so that a sleep() call can function properly, but they also need to accept interrupts while sleeping to be notified of a child process terminating. When the operating system wants to block interrupts of certain priority, one of the following "set-priority-level" calls are made:

Interrupts from lowest to highest priority	Blocks
spl0{}	nothing
splsoftclock{}	low-priority clock processing
splnet{}	network protocol processing
spltty{}	terminal multiplexers
splbio{}	storage controllers
splimp{}	network device controllers
splclock{}	high-priority clock processing
splhigh{}	all interrupts

*Table from Kirk McKusick's "The Design and Implementation of the 4.4BSD Operating System"

Context Switching

When a operating system uses a round-robin scheduler, the kernel must switch between the currently executable processes. This is done by a process called context switching. Context switching is an important part of scheduling because the frequency of switching can cause delays seen by the user if it is set too high or low. If the frequency is set high, the kernel will spend a lot of time in I/O loading the processes, but if it is set too low, the processes will be utilizing the CPU time efficiently. To perform a context switch, the scheduler must find an executable process to load into kernel mode. In a FreeBSD system, there are 32 queues that can each hold a number of processes. The queues are doubly-linked lists of pointers to processes, with header information that can specify if the queue is empty or not. The processes are put in to the appropriate queue by

simply dividing the priority of the process by 4. In FreeBSD there are two different types of context switching: voluntary, and involuntary. A voluntary switch occurs when a process no longer requires the use of the CPU because it is either waiting on a resource to become freed, or it has completed. An involuntary switch occurs when the time slice for the process expires, or a process awakens with a higher priority than that of the currently running process. A voluntary switch is performed by a process initiating a sleep process, and happens quite frequently in normal operation of the operating system. Another example of a voluntary switch is when a parent process goes into wait mode while waiting for child processes to finish. This mode removes the process from the kernel, and does not affect the scheduling process. When a child completes, it signals the parent process to symbolize completion, and the parent's state is altered. When the process comes out of sleep mode by receiving a wakeup() signal, the scheduler is responsible for correctly prioritizing the process by:

1. Removing the process from the sleep queue, and flagging it as being executable
2. Re-compute the priority of the process
3. Call the swapper process to move the process back into main memory if necessary, and put the process into the proper runqueue.

The processes that are stored on disk or in memory all take the same format for ease of switching. The order in which the user data for the process is stored is shown in Appendix B.

Error handling

A problem that all operating systems must deal with is that of thrashing. Thrashing occurs when there is not enough memory available to let a process complete.

This state occurs when there are many processes all waiting for CPU time which have a small amount of memory already delegated to them. 4.4BSD actively checks for this condition by watching the amount of free memory available, and the number of requests made to the memory. If the operating system believes that thrashing is occurring, it will mark the process that hasn't run in the greatest period of time blocked. Then, the pagedaemon will be called to move that process to storage to free the resources it was using. The operating system will continue in this manner until enough resources have been freed to resume normal operation. Once the amount of free memory begins to rise, the pagedaemon will pull the process data back out of storage, and the operating system will remove the blocking mark on the process.

Another problem that needs to be dealt with is deadlock. Deadlock can occur when two or more processes are all trying to request the same resources. Without proper handling of this situation, two processes could wait indefinitely for the other resource to become freed. A 4.4BSD system handles this by its policy of processes executing in the kernel mode. When a process is currently in the kernel mode, it may lock all resources that needs for the duration of it's time slice because it is impossible for it to be preempted. If the process is unable to complete it's job with the resources that are available, it must relinquish control of the processor and it's resources before it performs a sleep().

Conclusion

The FreeBSD process scheduler is a complex algorithm that handles the process rotation on a multi-user platform. It is the job of the scheduler to prioritize all processes

so that equal time is given to each process, and the CPU performs with maximum efficiency. This is accomplished by rotating through the highest priority processes at the correct intervals. The time slice quantum and the system clock rate are used to calculate the rotation interval, and it is usually around 100 milliseconds. Between these rotations, there is a context switch that needs to be executed quickly so that there is not an excess of wasted CPU time. Priorities of processes are based on factors such as recent CPU usage, a user-defined value, and the current load on the system. On a FreeBSD machine, long running processes like the Name Service Daemon or Sendmail are given a low priority until a request comes to them. Processes like the vi editor or a terminal shell have a higher priority.

Appendix A

Process Priority Values and Descriptions

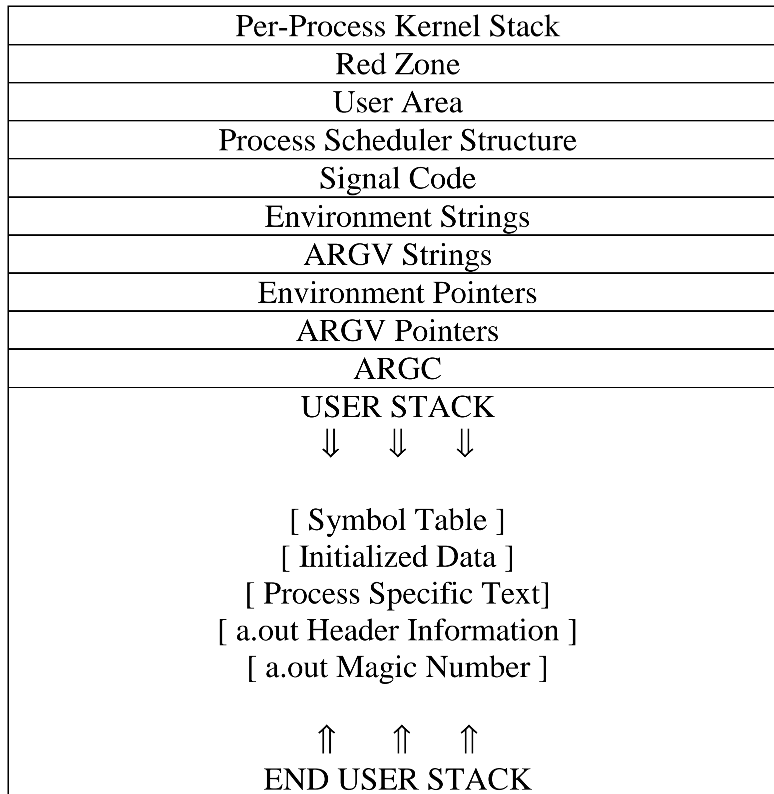
Priority	Value	Description
PSWP	0	Priority while swapping process
PVM	4	Priority while waiting for memory
PINOD	8	Priority while waiting for file control info
PRIBIO	16	Priority while waiting on disk I/O
PVFS	20	Priority while waiting for a kernel-level lock
PZERO	22	Baseline Priority
PSOCK	24	Priority while waiting on a socket
PWAIT	32	Priority while waiting for a child to exit
PLOCK	36	Priority while waiting for user-level lock
PPAUSE	40	Priority while waiting for a signal to arrive
PUSER	50	Base Priority for user-mode execution

Appendix B

Diagram of a Process in Memory or Storage

*Reproduced from McKusick's "The Design and Implementation of the 4.4BSD Operating System

Starting Address: Hex FFF00000



Ending Address: Hex 00000000

Bibliography

The Home Page of FreeBSD includes links to all source code, developers, and tutorials.
http://www.freebsd.org

An HTML version of the source code written by Warren Toomey, a developer.
http://minnie.cs.adfa.oz.au/FreeBSD-srctree/FreeBSD.html

Salim Douba, UNIX Unleashed – The System Administrator's Edition, SAMS Publishing 9/1998

Kirk McKusick, Keith Bostic, Michael Karels & John Quartermann. The Design and Implementation of the 4.4BSD Operating System ©1996, Addison-Wesley Publishing

Uresh Vahalia. UNIX Internals ©1996, Prentice-Hall Publishing