CPU Scheduling

by Maria Lima

Term Paper for

Professor Barrymore Warren
Mercy College
Division of Mathematics and Computer Information Science

CISC 421: Operating Systems
Summer 2006

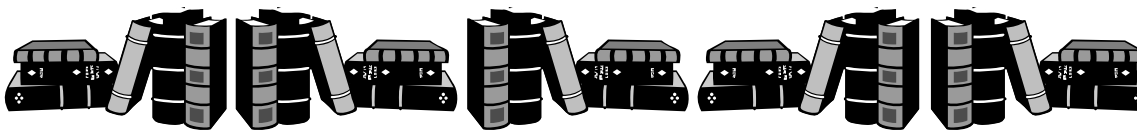

Table of Contents

Table of Figures

1. Introduction

This paper researches CPU scheduling, which is one of the areas of operating systems. Several topics related to CPU scheduling, such as processes and threads, are highlighted. A detailed discussion about CPU scheduling is provided, along with its implementation in Linux 2.6.8.1 and Windows XP.

2. CPU Scheduling Overview

By definition, CPU scheduling is a decision mechanism of the operating system to establish priorities among processes, which are ready to run, and the assignment of these processes to the CPU (central process unit) by order of priority through software, called a scheduler. As such, CPU scheduling is fundamental to the design of operating systems [1, 2].

CPU scheduling is the basis of computer multitasking and multiprocessing operating systems, as well as a key concept for real-time operating systems. Generically, one could say that multitasking is a method by which multiple processes share resources such as a CPU, and multiprocessing is a term used when a single computer system has two or more central CPUs. Also, it is understood that real-time operating systems are intended for real-time applications. Examples include programmable thermostats, household appliance controllers, mobile telephones, industrial robots, and scientific research equipment. These applications have in common that system deadlines can be met generally ("soft real-time") or deterministically ("hard real-time") to guarantee the finished product will be real-time [2].

Computer productivity and CPU utilization can be accomplished by CPU scheduling. A computer system will increase its productivity and CPU utilization if the switch of the CPU among processes and selection of the CPU scheduling algorithms are done according to the purpose or nature of the operating system. When a process is assigned to the CPU to run, it must execute until it must wait

for completion of an I/O request for instance. Since waiting time is not useful, another process should be assigned to the CPU to make the computer system productive. This is the concept of multiprogramming, which is essential for making a computer system productive. The CPU scheduling algorithms will be discussed later [1].

3. Processes and Threads

Before continuing the discussion on CPU scheduling, a brief description of programs, processes, and threads is provided.

A program is a combination of instructions and data to perform a task. Programs are like classes as defined in programming languages, such as Java [3].

A process is an instance of a program. Processes are like objects (i.e., instances of classes) as defined in programming languages, such as Java. They are instantiated to embody the state of a program during its execution. As such, processes keep track of the data that is associated with a thread or threads of execution, which includes variables, registers, program counter, contents of an address space (i.e., a set of memory addresses that a process is allowed to read and/or write to), etc. [3]. The following are distinct states of a process when it executes:

- New: the process is being created;
- Running: instructions are being executed;
- Waiting: the process is waiting for some event to occur;
- Ready: the process is waiting to be assigned to a processor;
- Terminated: the process has finished execution [1].

A thread, a lightweight process, is a basic unit of CPU utilization, which includes a thread ID, a program counter, a register set, and a stack. A traditional process, heavyweight process, has a single thread of execution. However, a process can have multiple threads of execution (threads) that work together, performing

more than one task at a time, to accomplish its goals. The kernel, which is a program of the operating system that runs at all times, must keep track of each thread's stack and hardware state (registers, program counter, etc.). Threads may or may not share address spaces. Since only one thread can be executing on a CPU at any given time, a kernel needs a CPU scheduler. Even though a process typically contained a single thread, most modern operating systems support processes that have multiple threads. A web browser is multithreaded (i.e., a process with multiple threads). As such, in a web browser, one thread handles user interface events, another thread handles network transactions, another renders web pages, etc. [1, 3].

4. Scheduling Mechanisms

An operating system that supports multiprogramming allows more than one process to be loaded in to memory and for the loaded processes to share the CPU using time-multiplexing. The latter is by definition sharing a resource by allocating the entire resource to one process for a time segment, then to another process for another time segment, and so on. A process has control of the entire resource while running. Multiprogramming is needed for two reasons: 1. since the operating system itself is implemented as one or more processes, there must be a way for the operating system and application processes to share the CPU. 2. processes need to perform I/O operations during computation, which normally take more time than CPU instructions; thus, the multiprogramming operating systems allocate the CPU to another process when a process invokes an I/O operation. As such, multiprogramming increases CPU utilization and performance of the computer system [4].

It is important to know how to design and implement the scheduling mechanism. The scheduling mechanism determines how the process manager knows it is time to multiplex the CPU, and how a process is allocated/deallocated from the CPU on the basis of a particular strategy, and it depends how it is implemented in a particular operating system [4].

Operating systems may have up to 3 distinct types of schedulers: a long-term scheduler (or job scheduler), a mid-term scheduler (or medium-term scheduler), and a short-term scheduler (or dispatcher). The long-term scheduler decides which jobs (batch systems service a collection of jobs, called a batch, by reading and executing them) or processes are to be admitted to the ready queue (set of all processes residing in main memory, ready and waiting to execute) and loads them into memory for execution. Thus, this scheduler dictates whether a high or low amount of processes are to be executed concurrently, and how the split between I/O intensive and CPU intensive processes is to be handled. The mid-term scheduler removes processes from main memory and places them on secondary memory (such as a disk drive), or vice versa. The mid-term scheduler may decide to "swap" the process "in" or "out" of main memory. For instance, a process may be removed from main memory, because it has not been active for some time, has a low priority, or is taking up a large amount of memory. That process may be placed later in main memory when more memory is available, or has been unblocked and is no longer waiting for a resource. The short-term scheduler decides which processes in the ready queue are to be executed and allocated to a CPU, following a clock interrupt, an I/O interrupt, a system call, or another form of signal. Thus, the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers. This scheduler can be preemptive, which is capable of removing processes from a CPU when the scheduler decides to allocate the CPU to another process, or non-preemptive, which is unable to remove processes from the CPU [5].

Therefore, the scheduling mechanism is the part of the process management that "handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy." Scheduling strategies are discussed next [4].

5. Scheduling Strategies

A scheduling mechanism is built around its influence in computer performance and CPU utilization. Criteria for choosing which and when a process should be executed must be defined. The scheduling policy decides which process runs at a given time. The system administrator or system designer selects the scheduling policy based on the way a computer is to be used [4].

Criteria to make CPU scheduling decisions are based on terms defined as follows:

- CPU Utilization: if it is necessary to keep the CPU as busy as possible. It may range from zero (0) to 100 percent;
- Throughput: if the number of processes that are completed per time unit, called throughput, is relevant. It may range from one process per hour to 10 processes per second, depending on long or short processes, respectively;
- Turnaround time: if it is significant to know how long it takes to execute a process; i.e., to know the interval from the time of submission to the time of completion of a process;
- Waiting time: if it is important to know the amount of time a process spends waiting in the ready queue. Waiting time is the sum of periods spent waiting in the ready queue before process is executed;
- Response time: if it is vital to know the amount of time since the submission of a request until the first response is produced [1].

Different schedulers have different goals as follows:

- Maximize throughput;
- Minimize latency;
- Prevent indefinite postponement;
- Complete process by given deadline;
- Maximize processor utilization [6].

Criteria to be used have been defined by comparing CPU scheduling algorithms, which have different properties and may favor one class of processes over another [1].

6. Scheduling Algorithms

Scheduling algorithms are needed to decide which of the processes in ready queue is to be allocated to the CPU [1]. Several CPU scheduling algorithm descriptions follow with examples to calculate the average waiting time of processes for each algorithm:

6.1 First-Come, First-Served (FCFS) Scheduling

FCFS has the simplest CPU scheduling algorithm scheme, processes are dispatched according to arrival time, and it is non-preemptible. FCFS is rarely used as a primary scheduling algorithm [6].

First-Come, First-Served (FCFS) Scheduling – Examples [8]:

Process	Burst Time (milliseconds)
P1	24
P2	3
P3	3

- If the processes arrive in the order: *P1* , *P2* , *P3*, *then*
- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27, and
- Average waiting time: (0 + 24 + 27)/3 = 17 milliseconds.

Figure 6.1 ***Gant Chart: FCFS Scheduling Algorithm – Example 1***

- If the processes arrive in the order: *P2* , *P3* , *P1*
- Waiting time for *P1* = 6*; P2* = 0*; P3* = 3, and
- Average waiting time: (6 + 0 + 3)/3 = 3 milliseconds.

Figure 6.2 ***Gant Chart: FCFS Scheduling Algorithm – Example 2***

Comparing examples above: the FCFS scheduling algorithm in Example 2 demonstrates to be much better than Example 1: the Convoy effect is present in Example 1, as all short processes wait for the one big process to be deallocated from the CPU. This effect results in lower CPU performance, which was avoided in Example 2 where the short processes were behind the long process [8].

6.2 Shortest-Job-First (SJF) Scheduling

The SJF associates each process width the length of its next CPU burst, which is characterized by a cycle of CPU execution and I/O wait. The scheduler uses the lengths of the CPU bursts to select the process with the shortest time to finish. Thus, the process that has the smallest next CPU burst is assigned when the CPU becomes available. If there are two processes with the same-length CPU burst, then FCFS scheduling is applied. There are two schemes for this algorithm:

- Non-preemptive: once CPU is allocated to the process, the process cannot be preempted until completion of its CPU burst. It results in slow response times to arriving interactive requests, and it is unsuitable for use in modern interactive systems. This scheme is optimal, because it gives minimum average waiting time, which is lower than in FCFS, for a given set of processes, reducing the number of waiting processes;

- Preemptive, also known as the Shortest-Remaining-Time-First (SRTF): if a shorter process arrives, i.e., if the CPU burst length of the process that arrives is less than the remaining time of current executing process, then the process in execution is preempted. In this scheme there is a very large variance of response times: long processes wait even longer than under SJF non-preemptive. It is not always optimal, because a short incoming process can preempt a running process that is near completion and context-switching overhead can become significant. Context switching is defined as the task of saving the state of the old process and loading the saved state for the new process when the CPU is switched from one process to another [1, 6].

Non-Preemptive SJF – Example [8]

Process	Arrival Time	Burst Time (milliseconds)
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- Average waiting time: (0 + 6 + 3 + 7)/4 = 4 milliseconds.

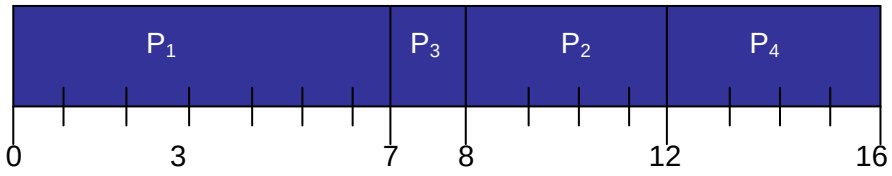

Figure 6.3 ***Gant Chart: SJF Scheduling Algorithm (non-preemptive) Ex.***

Preemptive SJF – Example [8]

Process	Arrival Time	Burst Time (milliseconds)
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- Average waiting time: (9 + 1 + 0 +2)/4 = 3 milliseconds.

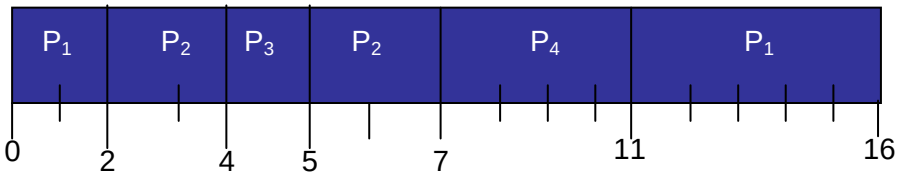

Figure 6.4 ***Gant Chart: SJF Scheduling Algorithm (preemptive) – Example***

6.3 Round Robin (RR)

The RR scheduling algorithm, which is designed for time-sharing systems, is based on FCFS, where processes run only for a limited amount of time called a time slice or quantum. A quantum is usually from 10 to 100 milliseconds. Each process gets a quantum and when that time elapses, the process is preempted and added to the end of the ready queue. If there are *n* processes in the ready queue, then each process gets 1/*n* of the CPU time in chunks of at most *q* quantum units at once. The process doesn't wait more than (*n*-1)*q* time units. The performance of RR depends on the size of the quantum, which also determines response time to interactive requests. If the quantum size is very large, then processes run for long periods and RR degenerates to FCFS. If the quantum size is very small, then the system spends more time on context switching than running processes. If the quantum size is in the middle-ground, then it is long enough for interactive processes to issue I/O requests, and the batch processes still get majority of processor time. A rule of thumb is that the CPU bursts should be 80% shorter

than the quantum size. RR requires the system to maintain several processes in memory to minimize overhead, and it is often used as part of more complex algorithms [1, 6].

RR with Time Quantum = 20 – Example [8]

Process	Burst Time (milliseconds)
P1	53
P2	17
P3	68
P4	24

- Typically, higher average turnaround than SJF, but better response;
- Average waiting time: (81 + 20 + 94 + 97)/4 = 73 milliseconds. The average waiting time is often long.

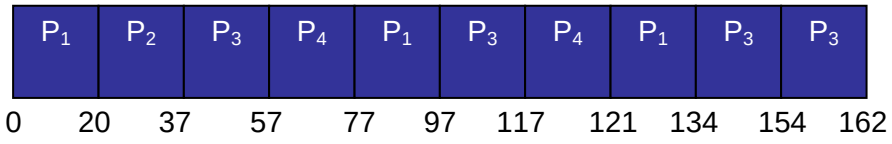

Figure 6.5 ***Gant Chart: Round Robin Scheduling Algorithm – Example***

6.4 Priority Scheduling

On modern operating systems, scheduling algorithms use priorities for processes, which determine the rank order the dispatcher uses to select a process to execute when the CPU is available [4]. The priority scheduling is described as follows:

- A priority number, an integer, is associated with each process;
- The CPU is allocated to the process with the highest priority. The allocation to the CPU could be preemptive or non-preemptive;
- SJF scheduling algorithm, which was discussed above, is a priority scheduling where priority is the predicted next CPU burst time;

- Priority scheduling could originate starvation, which is a problem when low priority processes may never execute; the solution is to increase the priority of processes as time progresses [8].

6.5 Multilevel Queue Scheduling

In multilevel queue scheduling, the ready queue is partitioned into separate queues in which processes have different scheduling needs and are permanently assigned to that queue. Processes could be grouped into two queues: foreground, which includes interactive processes, and background, which includes batch processes. Each queue has its own scheduling algorithm: the foreground supports RR, and the background supports FCFS. The scheduling must be done between the queues, each one getting a certain amount of CPU time – time slice - by serving 80% of the amount of CPU time to the foreground and the remaining 20% to the background. Another possibility could be the foreground to be served first and then the background. This can originate the problem of starvation, which is discussed above [1, 8].

6.6 Multilevel Feedback-Queue Scheduling

In multilevel feedback-queue scheduling, processes are separated by different CPU burst characteristics, and they can move between the various queues according to their priority. For instance, a process that waits too long in a lower-priority queue can be moved to a higher-priority queue - aging can be implemented this way to prevent starvation. Processes in the highest-level queue are executed with higher priority than processes in lower queues. Long processes repeatedly descend into lower levels: short processes and I/O-bound processes have higher priority and long processes will run when short and I/O-bound processes terminate. In multilevel feedback-queue scheduling, processes in each queue are serviced using round-robin and a process entering a higher-level queue preempts the running process [1, 6].

6.7 Multiple-Processor Scheduling

When multiple processors are available, CPU scheduling becomes much more complex compared with single-processor CPU scheduling and there is not a unique solution. In a multiple processor environment, it is necessary to implement load sharing, so various CPUs are allocated to processes to avoid one CPU to be busy while others are idle for instance. Also, it is necessary that two processors do not select the same process and that processes are not lost from the queue. There are two scheduling schemes in a multiple processor environment: Symmetric Multiprocessing (SMP) where each processor is self-scheduling and Asymmetric Multiprocessing where a unique processor makes the scheduling decisions while other processors execute only user code [1].

6.8 Real-Time Scheduling

Real time systems use real-time scheduling in which a process completes by a specific time. Otherwise, the results would be useless. Real-time scheduling is difficult to implement, because it must plan resource requirements in advance, incurs significant overhead, and service provided to other processes can degrade. Real-time scheduling is divided into two categories:

- Soft real-time scheduling, which does not guarantee that timing constraints will be met, but it requires that critical processes receive priority over less fortunate ones. Multimedia playback is an example of soft real-time system;
- Hard real-time scheduling in which timing constraints will always be met and failure to meet deadlines might have catastrophic results. Air traffic control is an example of hard real-time systems [1, 6].

In real time systems, processes must be scheduled so that they meet very specific deadlines, and CPU utilization should range from 40 percent to 90 percent, depending on lightly loaded systems and heavily loaded systems, respectively [1].

6.9 Thread Scheduling

A system has kernel-level threads and user-level threads. Their scheduling schemes are respectively as follows:

- Global Scheduling: the kernel decides which kernel thread to run next;
- Local Scheduling: the threads library decides which thread to put onto an available lightweight process (LWP) [1].

A scheduling algorithm is the method by which threads or processes are given access to system resources to effectively load balance a system. Usually, scheduling algorithms are only used in a kernel that supports time slice multiplexing, which is a form of preemptive multitasking wherein the system periodically suspends execution of one thread and begins execution of another thread. In order to effectively load balance a system, the kernel must be able to forcibly suspend execution of threads, so the next thread can begin execution [7].

7. CPU Scheduling: Implementation in Linux 2.6.8.1

Linux, which is a Unix-like computer operating system, is a well-known free open source development software. Originally, Linux was developed and used by individual enthusiasts on personal computers, but it has gained the support of major corporations such as IBM, Sun Microsystems, Hewlett-Packard, and Novell for use in servers, and it has become popular in the desktop market [9].

In Linux, CPU scheduling refers to scheduling processes and kernel tasks [10] - threads are called tasks in this operating system. The goals of Linux scheduling

are efficiency, interactivity, and fairness and preventing starvation. Efficiency, which is the most important goal, can be accomplished by the scheduler allowing as much work as possible to be done within restraints of other requirements, allowing tasks to run for longer periods of time, and by the scheduler's speed. As the effort to optimize Linux for desktop environments has increased, interactivity has become important. Thus, response to user within a certain time period should be imperceptible to users and giving the impression of an immediate response. Fairness and preventing starvation is an important goal for Linux scheduling; even though threads can have different priority levels, they must get a significant priority boost or one-time immediate preemption before they starve. The most welcome improvement in the Linux 2.6.8.1 was the introduction of the O(1) algorithm. This algorithm, which one might say it runs in "constant time," guarantees that every part of the scheduler executes within a certain amount of time regardless of the size of the input, i.e., how many tasks are on the system. Thus, the kernel handles a huge number of tasks without increasing overhead costs. The two data structures of the Linux 2.6.8.1, which allows the scheduler to run in O(1) time are runqueues and priority arrays. Runqueues keep track of all runnable tasks assigned to a particular CPU. One runqueue, which contains two priority arrays, is created and maintained for each CPU. All tasks on a CPU begin in an active priority array, and they are moved to the expired priority array as they run out of their time slices – a new time slice is calculated during the move. When there are no more runnable tasks in the active array, the process is swapped with the expired priority array. Priority arrays make finding the highest priority task in a system a constant-time operation and also make round-robin behavior within priority levels possible in constant-time operation. The scheduler selects the highest priority task, and if multiple tasks exist at the same priority level, they are scheduled round-robin with each other [3].

SMP Scheduling

Since the Linux kernel supports multiprocessing, the scheduler schedules tasks efficiently across multiple CPUs, which access the same memory and system

resources, making sure that one task is not executing on more than one CPU at a time, thus increasing the CPU's cache and making the best use of processor time [3].

SMT Scheduling

Since the Linux kernel supports scheduling multiple threads on a single Symmetric Multi-Threading (SMT) chip, each physical SMT chip can have more than one virtual processor with the caution that the virtual processors share certain resources, such as some types of cache. Thus, virtual processors should not be treated in the same way as regular processors [3].

NUMA Scheduling

Non-Uniform Memory Access (NUMA) is a computer architecture used in multiprocessors, where the memory access time depends on the memory location relative to a processor, which can access its own local memory faster than non-local memory (non-local memory is memory of another processor or memory shared between processors). The NUMA architecture logically follows symmetric multiprocessing (SMP) architecture. SMP systems might have 2 to 8 processors, but NUMA systems might have hundreds of processors. For instance, in February 2005, Silicon Graphics, Inc. (SGI) shipped NUMA systems containing 512 processors [3, 11].

Since the Linux kernel supports Non-Uniform Memory Access (NUMA), it treats multiple nodes as parts of a single system, which can run a single system image (i.e., one instance of the Linux kernel) across more than one node - a node is like a traditional single processor or multiprocessor that has its own CPU(s) and memory. Any CPU is capable of executing any thread, and allocates memory on any node on the system. If a CPU is executing a thread which is allocating memory from a local memory bank, it would be ineffective to move the thread across the system because it would take longer, thus resource proximity becomes

an issue. This issue makes organizing resources into groups, which is done by the scheduler domain system. The top level scheduler domain contains all CPUs in the system, and has one group for each node, each of which has one child scheduler. The scheduler might have more CPUs to schedule than most SMP systems [3].

Soft Real-Time Scheduling

Since the Linux scheduler supports soft real-time (RT) scheduling, it can schedule tasks that have strict timing requirements. The Linux kernel is capable of meeting very strict RT scheduling deadlines even though it does not guarantee that deadlines will be met. RT tasks are assigned special scheduling modes, such as FCFS and round-robin while essentially ignoring non-RT tasks on the system [3].

8. CPU Scheduling: Implementation in Windows XP

The Microsoft Windows XP is a preemptive multitasking operating system. Some of the key goals of this operating system are security, reliability, and high performance. Security: potential defects from previous version were identified and investigated to prevent security vulnerabilities. Reliability: Windows XP extends driver verification to catch more subtle bugs, adds new facilities for monitoring the health of the PC, makes the user interface easier to use. High performance: Windows XP is designed to provide high performance on desktop systems, server systems, and large multithreaded and multiprocessor environments. Windows XP has reduced the code-path length in critical functions, used better algorithms and per-processor data structures, used memory coloring for NUMA systems, etc. Threads can be preempted by higher priority threads except while they are running in the kernel dispatcher. Since Windows XP is designed for SMP, several threads can run at the same time [10].

The Windows XP kernel, which remains in memory and its execution is never preempted, has several responsibilities, one of which is thread scheduling, handled by the kernel dispatcher. The Windows XP supports a priority-based, preemptive scheduling algorithm for threads. A thread will run until is preempted by a higher-priority thread, terminates, its quantum ends, or until it calls a blocking system call. Real time threads have preferential access to the CPU when necessary. However, Windows XP is not a hard real-time system, because it does not guarantee execution of a real-time thread within any particular time constraint. Order of thread execution is set by a 32-level priority scheme. Priorities are divided into two classes: variable class, which contains threads with priorities from 1 to 15, and real-time class, which contains threads with priorities from 16 to 31. A queue is used for each scheduling priority, and the dispatcher traverses the set of queues until it finds a thread that is ready to run. The priority of each thread is identified by the priority class and its priority within each class – see figure below [10].

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 8.1 ***Windows XP Priorities***

A thread has a base priority of normal and it is interrupted when its quantum runs out. Its priority is lowered if the thread is in the variable class, but it is never lowered below the base priority. When a thread in a variable class is released from a wait operation, the dispatcher boosts the priority, but its increase depends on the nature of the wait operation. This scheduling gives good

response times to interactive threads, and I/O threads keep the I/O devices busy while computation threads use spare CPU cycles in the background. Windows XP supports foreground and background processes. When a process moves from background to foreground, the scheduling quantum increases by some factor, usually by three (3), which gives the foreground three times longer to run before a preemption occurs [10].

References

[by order of appearance in term paper]

[1] Abraham Silberschatz, Peter Galvin, Greg Gagne. *Applied Operating Systems Concepts*, first edition. Wiley, 2000, ch 6.

[2] *Scheduling*. http://en.wikipedia.org/wiki/Scheduling_%28computing%29, accessed on August 22, 2006.

[3] Josh Aas. *Understanding the Linux 2.6.8.1CPU Scheduler*. SGI, 2005. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf, accessed on August 22, 2006.

[4] Gary Nutt. *Operating Systems, A Modern Perspective*. Addison-Wesley, 1997, ch 7.

[5] *Types of Operating System Schedulers*. http://en.wikipedia.org/wiki/CPU_Scheduling, accessed on August 24, 2006.

[6] *Processor Scheduling*. Deitel & Associates, 2004. http://courses.cs.vt.edu/~cs3204/spring2004/Notes/OS3e_08.pdf#search=%22Processor%20Scheduling%2C%20deitel%20%26%20associates%22, accessed on August 24, 2006.

[7] *Scheduling Algorithm*. http://en.wikipedia.org/wiki/Scheduling_algorithm, accessed on August 28, 2006.

[8] Abraham Silberschatz, Peter Galvin, Greg Gagne. *Operating Systems Concepts with Java*. Wiley, 2003, ch 6. http://bcs.wiley.com/he-bcs/Books?action=resource&bcsId=1789&itemId=0471489050&resourceId=2670&chapterId=8665, accessed on August 29, 2006.

[9] *Linux*. http://en.wikipedia.org/wiki/Linux, accessed on August 31, 2006.

[10] Abraham Silberschatz, Peter Galvin, Greg Gagne. *Operating Systems Concepts*, with Java, sixth edition. Wiley, 2004, chs 6, 20, 21.

[11] *Non-Uniform Memory Access*. http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access, accessed on September 1, 2006.