

CPU Scheduling,
With an Emphasis on Algorithms

CS-384

Written by Kevin Mellott

Submitted to Dr. Chris Taylor

February 3, 2003

Table of Contents

1	Introduction.....	4
1.1	What is CPU Scheduling?	
1.1.1	Purpose of CPU Scheduling	
1.2	Processes	
1.2.1	Process Creation	
1.2.2	Process State	
1.2.3	Process Control Block	
1.3	Threads	
1.3.1	User Threads	
1.3.2	Kernel Threads	
2	CPU Scheduling Concepts.....	7
2.1	Bursts	
2.2	Preemptive Scheduling	
2.3	Dispatcher	
3	Basic Algorithms.....	9
3.1	First-Come, First-Served	
3.2	Shortest-Job-First	
3.3	Priority	
3.4	Round-Robin	
3.5	Multilevel Queue	
3.6	Multilevel Feedback Queue	
3.7	Choosing an algorithm	

4	Additional Algorithms.....	13
4.1	Fixed Priority	
4.1.1	Rate Monotonic	
4.1.1.1	Rate Monotonic with Delayed Preemption	
4.1.2	Deadline Monotonic	
4.2	Dynamic Priority	
4.2.1	Earliest Deadline First	
4.2.2	Least Slack Time First	
5	Multiprocessor Scheduling.....	17
5.1	Architecture	
5.1.1	Homogeneous	
5.1.2	Heterogeneous	
5.2	Advantages	
5.3	Disadvantages	
6	Real-Time Scheduling.....	20
6.1	Hard real-time	
6.2	Soft real-time	
7	Bibliography.....	23

Introduction

What is CPU Scheduling?

CPU scheduling strategically allocates the CPU to a process based on a specified criteria. There are many different methods of selecting which process will be given control of the CPU. Each of these methods follows a different scheduling algorithm and has advantages and disadvantages.

Purpose of CPU Scheduling

The purpose of CPU scheduling is to maximize the utilization of the CPU. In order to do this, a process should be running at all times. For example, if a process is waiting for some event to occur before it is able to continue execution, then it should not have control of the CPU. If it does, then the CPU is being wasted.

Processes

A process is the execution of a program. It is often thought of as the unit of work in a computer system, because the operating system is constantly running processes.

Process Creation

Processes can be created by the kernel of the operating system or by the user. An example of a process created by the user is Microsoft Word. This process is only created once the user decides to launch Word. Processes that are created by the kernel take care of various system tasks. An existing process can also create a process, which is then referred to as the child process. The creating process is labeled the parent process.

Process State

There are five possible states that a process may be in: new, running, waiting, ready, or terminated. These states are described below.

- New – The process is being formed.
- Running – Instructions from the process are being executed.
- Waiting – The process is waiting for an event to happen before it can resume its execution.
- Ready – The process is ready to be run, but is waiting for the processor.
- Terminated – The process has completed its execution.

Process Control Block

The process control block (PCB) stores information pertaining to a specific process. It stores the process state, program counter, CPU registers, CPU-scheduling information, memory-management information, accounting information, and I/O status information. The PCB tells the CPU what values to load into its registers when it is given the process, as well as where to begin executing the process. Overall, the PCB serves as a data banker that stores important information for each process.

Threads

A thread consists of a thread ID, program counter, set of registers, and a stack. It is the fundamental unit of CPU utilization. Single-threaded processes can only perform one task at a time, and multi-threaded processes can perform many different jobs

concurrently. Since a multi-threaded process contains a number of threads of execution, it is able to do more than one job at a time.

User Threads

User threads are implemented by a thread library at the user level, and are created without help from the kernel. All user threads are created and scheduled in user space, and the kernel is completely unaware of user threads. Since user threads are independent of the kernel, they are typically fast to create and manage.

Kernel Threads

Kernel threads are created and managed directly by the operating system. All of this is done in kernel space. Since the operating system creates the kernel threads, the threads are usually slower to create and manage than the user threads are.

CPU Scheduling Concepts

Bursts

An executing process alternates between two cycles: CPU execution and I/O wait. The amount of time it takes for each CPU execution to be completed is called the CPU burst, and the time the process spends waiting for an I/O request to be fulfilled is the I/O burst. Historical data for the burst times of a process are taken into account when selecting a CPU scheduling algorithm. The point of this is to prevent a process with a very long burst time from taking control of the CPU before a process with a short burst time is able to run.

Preemptive Scheduling

A preemptive scheduling algorithm has the power to take control of the CPU away from a process. This is done when the process currently running has either exceeded its allotted CPU usage time, or if another process has been given a higher priority than the current process. Preemptive scheduling is a method that can be incorporated into a number of different scheduling algorithms and can prevent a single process from tying up the CPU.

Dispatcher

The dispatcher is responsible for performing context switches. This means that it gives control of the CPU to the process that is to be run next. First, it saves the state of the currently running process. Next, the state of the selected process is loaded into the

registers. Finally, the process begins executing at the location specified by the program counter.

Basic Algorithms

First-Come, First-Served

First-Come, First-Served (FCFS) is the most basic scheduling algorithm. It simply allocates the CPU to the process that requested it first. All other processes that request the CPU are placed in the ready queue and are served in the order in which they arrive. This algorithm is non-preemptive; once a process is given control of the CPU it will keep it until the process releases it. A process will release the CPU when it has either finished executing, or if it needs to wait for an I/O event to occur before it can resume its execution.

Shortest-Job-First

Shortest-Job-First (SJF) examines the amount of time each process is expected to take during its next CPU burst. It uses this information to form a queue, and then gives control of the CPU to the process that is expected to have the shortest next CPU burst. In theory, this algorithm is very efficient, because it minimizes the average waiting time for each process. However, since the next CPU burst time can be very difficult to accurately determine, this algorithm does not always produce the intended results.

Priority

Priority scheduling algorithms assign each process a priority level, which is represented by a number. Some systems consider low numbers to have high priorities, and others designate the high numbers to indicate a higher priority. These priorities can be assigned internally (by the operating system) or externally (by the user). Considering

some sort of measurable quantity, such as how many resources each process requires, usually sets internal priority levels. External priority levels reflect a user-specified order of importance for each process. The process with the highest priority is initially given control of the CPU. If the algorithm is preemptive, then the currently running process will be preempted once a process in the ready queue has a higher priority. However, if the algorithm is non-preemptive, then once the process begins running all other processes will have to wait until it releases the CPU, regardless of what their priority levels may be.

The major concern with priority scheduling is the concept of starvation. This occurs when a process has such a low priority that it is never given control of the CPU. A solution to this is aging, which gradually increases the priority of a process over time.

Round-Robin

The Round-Robin scheduling algorithm (RR) defines a time quantum, which is the amount of time each process in its queue is allocated the CPU. This algorithm is preemptive and fills its queue using the FCFS method. After a process has exceeded its time limit, it is moved to the end of the queue and the next process in the queue is given control of the CPU. Setting a correct time quantum is critical to the performance of the RR algorithm. If the time quantum is too small, then few processes will complete the first time they are run and much time will be spent performing context switches.

Alternatively, if the time quantum is too large, then most processes will finish and the CPU will sit idle for the remainder of the quantum. In both of these cases, the CPU is not being used efficiently. Generally, it is desirable for around 80 percent of the processes to complete before the time quantum is reached.

Multilevel Queue

A multilevel queue scheduling algorithm separates the ready queue into a number of different queues. Each of these queues uses their own scheduling algorithm. For instance, there may be separate queues created for CPU-bound and I/O-bound processes. The multilevel queue algorithm is responsible for allocating the CPU to each of these different queues. Any scheduling algorithm can be used to achieve this goal, however, fixed-priority preemptive scheduling is commonly used. If the CPU-bound queue were assigned the higher priority, then the CPU would not be given to the I/O-bound queue until all of the processes in the CPU-bound queue had completed. A multilevel queue is useful when the processes can be easily grouped by importance. Otherwise, this method will prevent entire groups of processes from running and would not be effective.

Multilevel Feedback Queue

This algorithm works like the multilevel queue, but it is able to move a process from one queue to another. When a process uses too much CPU time, then it may be moved to a queue with a lower priority. Alternatively, a process that is not given enough CPU time may be promoted to a queue with a higher priority. This flexibility prevents starvation and also prevents a process from tying up the CPU.

Choosing an algorithm

Many different algorithms exist, and they can be mixed to create custom algorithms. Each of them have advantages and disadvantages. Therefore, algorithms have

to be chosen based on certain desired qualities. The qualities most commonly observed are listed below.

- CPU utilization – This can range from 0 to 100 percent. It is desirable to have a higher percent of CPU utilization, because it indicates that the hardware is not being wasted.
- Throughput – The number of processes completed per unit of time is the throughput. This is a measurement of the work being done by the CPU. It is not always a good indicator, however, because it depends directly on the length of the processes being executed.
- Turnaround time – This measures the amount of time it takes between the submission of a process and the time the process completes its execution. It is measured for each process and takes all the time the process spends waiting and executing into account.
- Waiting time – This is a good measure of how efficiently the scheduling algorithm is allocating the CPU to ready processes. It measures the sum of the time periods a process spends waiting in the ready queue.
- Response time – This is measured only in interactive systems, and is the amount of time it takes to start responding to a request. The speed of the output device usually limits the response time.

Additional Algorithms

Mathematical equations are needed to analyze the following algorithms. The sum of the requested utilization from all the tasks in a system,

$$U = U_1 + U_2 + \dots + U_n$$

is used to determine whether or not the deadlines of all tasks can be met (Stewart).

Utilization is the percent of the CPU that is being used, and can be found using

$$U = C / T$$

where C is the worst case execution time of a task, and T is its period (Stewart).

Fixed Priority

In a fixed priority algorithm, each process is initially assigned a priority level. These levels are permanent and can never change. Fixed priority algorithms are easy to understand and manage; however they do not discourage starvation. A process with a low priority will always have that low priority and will never run until all processes with a higher priority have completed.

Rate Monotonic

Priority is assigned to processes based on the lengths of their periods of execution. Processes with shorter periods are given a higher priority than those with longer periods. A system using the Rate Monotonic algorithm will only accept a process if its deadline and period are equal (Echague 184).

The worst-case scheduling bound of the Rate Monotonic algorithm is represented by

$$W_n = n * (2^{(1/n)} - 1)$$

where n is the number of tasks in the system. Anytime the total requested utilization is less than or equal to W_n , the system will be able to complete all tasks by their respective deadlines (Stewart 80).

The biggest problem with the Rate Monotonic algorithm is the lack of CPU utilization. Whenever more than one task is present in the system, the worst-case schedulable bound is always less than 100 percent. This means that the CPU can never be 100 percent utilized and still complete all tasks by their deadlines. Furthermore, the RM algorithm's worst-case schedulable bound can be as low as 69 percent and averages 88 percent (Stewart). Despite these disadvantages, the Rate Monotonic algorithm is optimal amongst fixed priority scheduling algorithms.

Rate Monotonic with Delayed Preemption

One major problem with the Rate Monotonic algorithm is that it is quite possible to encounter a large number of context switches, which can be very expensive (Gopalakrishman 56). The point of the Rate Monotonic with Delayed Preemption is to allow a running process to delay yielding the CPU to a process with a higher priority, thus reducing the number of context switches (Manabe 212).

Deadline Monotonic

The length of time between the start of the period and the deadline of the process determines priority in the Deadline Monotonic (DM) algorithm. Processes with the shorter lengths of time are given higher priorities. Systems using the DM algorithm

impose the constraint that each processes deadline must be less than or equal to its period (Echague 184).

Dynamic Priority

In dynamic priority algorithms, the priority of a process can be changed. This makes the algorithm much more complex, but also significantly more powerful. It can eliminate starvation by aging processes; if a process has been in the queue for a specified amount of time and has not received control of the CPU, then its priority is increased. Dynamic priority algorithms are very useful in real-time applications. If a process needs to finish and its deadline is approaching, then its priority can be increased so that it is able to complete on time. The cost incurred by using a dynamic rather than a fixed priority algorithm is an increase in the amount of run-time overhead.

A dynamic algorithm will be able to finish its task set if the total CPU utilization requested is less than 100 percent. This means that its worst-case schedulable bound is 100 percent (Stewart). Dynamic algorithms can be very difficult to predict when the system is experiencing a very heavy workload.

Earliest Deadline First

This assigns priority based on how close each process is to its deadline. The process with the shortest amount of time remaining until its deadline is scheduled to arrive is given the highest priority. Earliest Deadline First (EDF) is a preemptive algorithm and gives control of the CPU to whichever process has the highest priority

(Kuhns). Since priorities change, the currently running process is immediately preempted once a different process with a higher priority becomes ready.

This algorithm will complete all scheduled tasks on time when the tasks all have relative deadlines equal to their periods, and the total utilization is no greater than 100 percent.

Least Slack Time First

Priority is assigned based on the slack time of each process. The slack time of a process is determined using

$$\text{Slack time} = \text{absolute deadline} - \text{current time} - \text{remaining compute time}.$$

Processes with smaller slack times are given higher priorities (Kuhns). The slack time represents the amount of time the execution of a task can be delayed before it is sure to miss its deadline. This algorithm usually produces a large number of context switches and always requires knowledge of the tasks execution time.

Multiprocessor Scheduling

Architecture

A multiprocessor scheduling environment uses more than one processor to execute its processes. This can be done in a number of different ways and is generally much more difficult to implement than a single processor algorithm due to its complex nature. It is very important to pay close attention to details when creating a multiprocessor scheduling algorithm, or else processes can be lost very easily.

Multiprocessor scheduling algorithms require a number of processors to work together to complete the tasks the system is given, and its structure is either homogeneous or heterogeneous.

Homogeneous

In a homogeneous system, all of the processors are identical in terms of their functionality. This type of system assumes uniform memory access, which means that each of the processors have access to the same memory. Therefore, any available processor is used to run any process waiting in the ready queue.

One possible issue on homogeneous systems is resource sharing. For example, if an I/O device is attached locally to one of the processors, then any process wishing to use that particular device has to be scheduled to run on that processor. Otherwise, the device will never be available to them. Allocating a different ready queue to each processor, which is referred to as load sharing, can solve this problem. However, one processor may be heavily utilized while another is idle. Because of this, a single ready queue is almost always used.

Heterogeneous

In a heterogeneous multiprocessor system, different processors are capable of performing different tasks. For example, one processor might be responsible for all I/O requests while another handles the CPU requests. This method dedicates each processor to a specific task, therefore separating the functionality of the system. Sometimes one processor will act as the scheduler and will assign processes to each of the processors in the system. This avoids the situation where two processors try to access the same process and also eliminates a lot of the overhead when programming the individual processors.

Advantages

Multiprocessor systems have the obvious advantage of having more hardware available to them than a single processor system. This can lead to an increase in throughput and a decrease in waiting time, if the system is implemented properly.

Disadvantages

A major disadvantage to a multiprocessor system is the additional implementation that is needed to ensure its correct operation. In most cases, each processor needs to know what the other processors are doing. In the situation where each processor selects its own process to run, each processor must know a number of things about the process they are selecting. They must know whether or not they have access to the resources that the process requires as well as whether or not another processor is currently attempting to select the same process.

Communication between the processes and the processors can slow down a multiprocessor system. The additional time it takes for the processors to select a process, or have one selected for them, and perform a context switch can be a significant disadvantage.

Real-Time Scheduling

Hard real-time

A hard real-time system guarantees that a critical task will be completed within a specified amount of time. Each process and its allowed running time are presented to the system, and it determines whether or not it is possible to complete the process in the indicated amount of time. To do this, the system looks at the sum of the time it takes to run each of the various operating system functions that the process uses and compares it to the allowed run time of the process. If it is possible to complete the task, then the system accepts it. Otherwise, it is deemed impossible and rejected by the system.

The method described above is intended for a hard real-time system consisting of special-purpose software running on hardware that is dedicated to a critical process. These specialized systems lack the full functionality offered by modern computers and operating systems.

Hard real-time guarantees cannot be made on systems with secondary storage or virtual memory. There is an unpredictable variance in the amount of time it takes to execute a particular process on one of these systems. It may take significantly longer to run a process on a machine with virtual memory if the effective access time is high.

Whenever a page fault occurs, the access time increases. This is shown by

$$\text{Effective access time} = (1 - p) * ma + p * (\text{page fault time}),$$

where p is the probability of a page fault and ma is the memory access time (Stewart 79).

Soft real-time

A soft real-time system attempts to complete its tasks by their respective deadlines, but does not guarantee this will happen. Critical processes receive higher priority over those that are not critical. In a time-sharing system with soft real-time capabilities, those processes that are real-time must be given the highest priority and this priority must not degrade over time. However, the priority of the processes that are not real-time in the system can decrease over time. This can cause long delays for these processes; sometimes even starvation. The dispatch latency in a soft real-time system must be as small as possible. A small switching time will provide more time for each process to run.

Dispatch latency times can become very long in an operating system that requires a system call to complete or an I/O block to occur before performing a context switch. This is the case in the UNIX operating system (Kuhns). One way to solve this issue is to make the system calls preemptible, by inserting preemption points in the system calls. A preemption point is a location in the code that is not critical; interrupting the process at this point will not harm its data. Whenever a preemption point is reached, the system checks to see if a higher-priority process needs to be run. If it does, control of the CPU is given to the process. Once it is finished, the CPU goes back to the initial preemption point and continues execution.

Priority inversion occurs in a soft real-time system when one or more processes are using resources needed by a high-priority process. In this situation, the lower-priority processes are given a priority equal to the high-priority process until they are done using the desired resource or resources. Once the resource is released, each process is given its previous priority level. This technique is known as priority-inheritance protocol, and it

prevents high-priority processes from spending a lot of time waiting for resources to become available.

Bibliography

- Echague, Juan, Ripoll, Ismael, and Crespo, Alfons. "Hard Real-Time Preemptively Scheduling with High Context Switch Cost." *IEEE Computer Society*. 14 March 1995: 184 – 190.
- Gopalakrishman, R., and Parulkar, Gurudatta. "RMDP – A Real-time CPU Scheduling Algorithm to Provide QoS Guarantees for Protocol Processing." *IEEE Computer Society*. 15 May 1995: 56 – 57.
- Kuhns, Fred. "CS523 Operating Systems Lecture Notes." 16 Oct. 2001. Washington University in Saint Louis Online. 6 Jan. 2003.
<<http://www.cs.wustl.edu/~fredk/Courses/cs523/fall01/Lectures/periodic.pdf>>
- Manabe, Yoshifumi, and Aoyagi, Shigemi. "A Feasibility Decision Algorithm for Rate Monotonic Scheduling of Periodic Real-Time Tasks." *IEEE Computer Society*. 15 May 1995: 212 – 219.
- Silberschatz, Abraham, Galvin, Peter, and Gagne, Greg. *Operating Systems Concepts*, 6th ed. New York, NY: John Wiley & Sons, 2003.
- Stewart, David and Michael Barr. "Rate Monotonic Scheduling." *Embedded Systems Programming*. March 2002; 79 – 80.

Stewart, David. "Introduction to Real-Time Scheduling Theory." Spring 2000. University of Maryland Online. 30 Jan. 2003.

< <http://www.ece.umd.edu/class/enee642.S2000/handout7.shtml> >