CPU Scheduling

6.1 Basic Concepts

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. Trying to use the time productively.

6.1.1 CPU-I/O Burst Cycle

Process Execution consists of a cycle of CPU execution and I/O wait. Process execution begins with a ***CPU burst*** that is followed by an ***I/O burst*** and so on (alternating). The durations of CPU bursts have been measured extensively, they will depend of process and computer. They tend to have a frequency curve characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

6.1.2 CPU Scheduler

Whenever the CPU becomes idle, the ***short-term scheduler or CPU scheduler*** select the process from the ready queue to be executed and allocates the CPU to that process. The ready queue can be implemented as FIFO queue, a priority queue, a tree, or simply an unordered linked list. All the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is ***nonpreemptive*** or **cooperative,** otherwise it is ***preemptive***.

Nonpreemptive scheduling: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state(used by Microsoft Windows 3.x). Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Preemptive scheduling: preemptive scheduling can result in race conditions when data are shared among several processes. Preemption also affects the design of the operating-system kernel (used by Windows 95 and Mac OS X for the Macintosh)

6.1.4 Dispatcher

The dispatcher is the module that gives control of the CPU totheprocessselected bythe short-term scheduler. This function involves the following:

• Switching context

• Switching to user mode

• Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start an other running is known as the ***dispatch latency***

6.2 Scheduling Criteria

- ***CPU utilization:*** CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- ***Throughput***: One Measure of work is the number of processes that are completed per time unit. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- ***Turnaround time***: It is the interval from time of submission of a process to the time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O.
- ***Waiting time***: Time that a process spends waiting in the ready queue. It is the sum of the periods spend waiting in the ready queue
- ***Response time***: It is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average.

6.3 Scheduling Algorithms

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

The implementation of the FCFS policy is easily managed with a ***FIFO queue***. When a process enters the ready queue, its PCB (process control block) is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. On the negative side, the average waiting time under the FCFS policy is often quite long.

Example

Processes arrive at time 0, with the length of the CPU burst given in milliseconds

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds

If the processes arrive in the order P2, P3, P1, the average waiting time is (0 + 3 + 6)/3 = 3 milliseconds

Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly. The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals.

6.3.2 Shortest-Job-First-Scheduling (SJF)

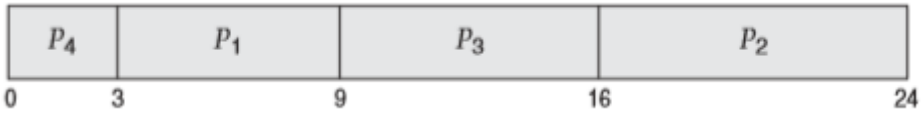
This algorithm depends on the length of the next CPU burst of a process, rather than its total length. When the CPU is available, it is assigned to ***the process that has the smallest next CPU*** burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Example

Length of the CPU burst given in milliseconds

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Utilizando el algoritmo los procesos se ejecutan en el siguiente orden (***ya que se ordena de menor a mayor***):



The average waiting time is (0 + 3 + 9 + 16)/4 = 7 milliseconds.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. SJF scheduling is used frequently in long-term scheduling. Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. Preemptive SJF scheduling is sometimes called ***shortest-remaining-time-first scheduling.***

SJF NonPreemptive tutorial: https://www.youtube.com/watch?v=LNsHRY5Nk5U

SJF Preemptive tutorial: https://www.youtube.com/watch?v=h-e7QtjfmkI

6.3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. ***We assume that low numbers represent high priority***

Example

All processes arrive at time 0 in order, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Ordering by priority we got:

The average time will be: $(\sum_1^n(initial\ time\ -arriving\ time)\ )/total\ processes$

Tutorial: https://www.youtube.com/watch?v=cNdEQKw4apM

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds

being paid for computer use, the department sponsoring the work, and other, often political, factors. Priority scheduling can be either preemptive or nonpreemptive.

A major problem with priority scheduling algorithms is ***indefinite blocking***, or ***starvation***. A solution to the problem of indefinite blockage of low-priority processes is ***aging***. Aging involves gradually increasing the priority of processes that wait in the system for a long time.

6.3.4 Round-Robin Scheduling (RR)

It is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a ***time quantum*** or ***time slice***, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue

Tutorial: https://www.youtube.com/watch?v=3N2t9_6Co3U

6.3.5 Multilevel Queue Scheduling

Created for processes that are easily classified into different groups. A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue. Each queue has its own scheduling algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

6.3.6 Multilevel Feedback Queue Scheduling

Allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

• The number of queues

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higher priority queue

• The method used to determine when to demote a process to a lower priority queue

• The method used to determine which queue a process will enter when that process needs service

6.4 Tread Scheduling

Kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a light weight process(LWP).

6.4.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as ***process***

contention scope (PCS), competition for the CPU takes place among threads belonging to the same process. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread. To decide which kernel-level thread to schedule onto a CPU, the kernel uses ***system-contention scope(SCS***) scheme, competition for the CPU takes place among all threads in the system. The SCS scheme runs on systems implementing the one-to-one model.

6.4.2 Pthread Scheduling

The POSIX Pthread API allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- ***PTHREAD_SCOPE_PROCESS*** schedules threads using PCS scheduling. This policy schedules user-level threads onto available LWPs, the number of LWPs is maintained by the thread library, perhaps using scheduler activations.
- ***PTHREAD_SCOPE_SYSTEM*** schedules threads using SCS scheduling. This policy will create and bind a LWP for each user-level thread on many-to-many systems, mapping threads using the one-to-one policy.

The Pthread IPC provides two functions for getting—and setting—the contention scope policy:

- ***Pthread_attr_setscope(pthread attr t *attr, int scope)***
  - *First parameter* > contains a pointer to the attribute set for the thread
  - *Second parameter >* receive the ***PTHREAD_SCOPE_SYSTEM*** or the ***PTHREAD_SCOPE_PROCESS*** value, indicating how the contention scope is to be set.
- ***Pthread_attr_getscope(pthread attr t *attr, int *scope)***
  - *First parameter* > contains a pointer to the attribute set for the thread
  - *Second parameter >* contains a pointer to an *int value* that is set to the current value of the contention scope

If an error occurs, each of these functions returns a nonzero value.

6.5 Multiple-Processor Scheduling

6.5.1 Approaches to Multiple-Processes Scheduling

One approach is ***asymmetric multiprocessing***, it has a master server(single processor) handled scheduling decisions, I/O processing and other system activities. This ***asymmetric multiprocessing*** is simple because only one processor accesses the system data structures, reducing the need for data sharing.

Other approach is ***symmetric multiprocessing (SMP)***, each processor is self-scheduling. All processes may be in a common ready queue or each processor may have its own private queue of ready processes. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue.

6.5.2 Processor Affinity

Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**.

Processor affinity takes several forms:

- ***Soft Affinity:*** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing that it will do so. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
- ***Hard affinity:*** When an operating system provides system calls, thereby allowing a process to specify a subset of processors on which it may run.

6.5.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. ***Load balancing*** attempts to keep the workload evenly distributed across all processors in an SMP system, it is necessary only on systems where each processor has its own private queue of eligible processes to execute.

There are two general approaches to load balancing:

- ***Push Migration:*** a specific task periodically checks the load on each processor and if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
- ***Pull Migration:*** occurs when an idle processor pulls a waiting task from a busy processor.

Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems. Load balancing often counteracts the benefits of processor affinity.

6.5.4 Multicore Processors

Multicore processor is place multiple processor cores on the same physical chip. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor. SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip. Multicore processors may complicate scheduling issues.

Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a ***memory stall***. To remedy this, many recent hardware designs have implemented multithreaded processor cores in which two (or more) hardware threads are assigned to each core. That way, if one thread stalls while waiting for memory, the core can switch to another thread.

In general, there are two ways to multithread a processing core:

- ***Coarse-grained:*** a thread executes on a processor until a long-latency event such as a memory stall occurs, with this delay caused by the long-latency event, the processor must switch to another thread to begin execution. However, the cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core.
- ***Fine-grained:*** is known also as ***interleaved***, switches between threads at a much finer level of granularity typically at the boundary of an instruction cycle. The cost of switching between threads is small.

A multithreaded multicore processor actually requires two different levels of scheduling:

- On one level are the scheduling decisions that must be made by the operating system as it chooses which software thread to run on each hardware thread (logical processor). At this level the operating system may choose any scheduling algorithm.
- A second level of scheduling specifies how each core decides which hardware thread to run, the thread-switching logic compares the urgency of the two threads and selects the thread with the highest urgency value to execute on the processor core.

6.6 Real-Time CPU Scheduling

CPU scheduling for real-time operating systems types:

- ***Soft real-time systems:*** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes
- ***Hard real-time systems:*** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

6.6.1 Minimizing Latency

When an event occurs, the system must respond to and service it as quickly as possible. We refer to ***event latency*** as the amount of time that elapses from when an event occurs to when it is serviced.

Two types of latencies affect the performance of real-time systems:

- ***Interrupt latency:*** refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When this happen, the operating system must:
    - Complete the instruction it is executing and determinate the type of interrupt occurred
    - Save the state of the current process before applying the interrupt using the specific interrupt service routine(ISR).
- ***Dispatch latency:*** The amount of time required for the scheduling dispatcher to stop one process and start another. The most effective technique for keeping dispatch latency low is to provide preemptive kernels. The *conflict phase* of dispatch latency has two components:
    - Preemption of any process running in the kernel
    - Release by low-priority processes of resources needed by a high-priority process

6.6.2 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features.

The processes are considered ***periodic***, they require the CPU at constant intervals(periods). Once a periodic process has acquired the CPU, it has a fixed processing time ***t***, a deadline ***d*** by which it must be serviced by the CPU, and a period ***p***. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The ***rate*** of a periodic task is $1/p$.

Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements. With a technique known as ***admission-control*** algorithm, the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it can not guarantee that the task will be serviced by its deadline.

6.6.3 Rate-Monotonic Scheduling

The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Example:

	Periods (p)	Processing Time (t)
P1	50	20
P2	100	35

The form know the CPU scheduling time ***t/p*** for each process. The worst-case CPU utilization for scheduling N processes is $N(2^{1/n} - 1)$

6.6.4 Earliest-Deadline-First Scheduling (EDF)

Dynamically assigns priorities according to deadline. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process.

EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable.

6.6.5 Proportional Share Scheduling

Proportional share schedulers operate by allocating ***T*** shares among all applications. An application can receive ***N*** shares of time, thus ensuring that the application will have ***N/T*** of the total processor time.

Example:

T = 100 divided in three processes

	Shares
A	50
B	15
C	20

The total shares represent the total processor time that the processes will get. Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a

client requesting a particular number of shares only if sufficient shares are available. In the example show it before we have 85 shares of the total 100, if a new process D appears requested 30 shares it will be denied.

6.6.6 POSIX Real-Time Scheduling

POSIX defines two scheduling classes for real-time threads:

- ***SCHED_FIFO:*** schedules threads according to a first-come, first-served policy using a FIFO queue. There is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks.
- ***SCHED_RR:*** uses a round-robin policy, it provides time slicing among threads of equal priority

POSIX provides an additional scheduling class ***SCHED_OTHER*** but its implementation is undefined and system specific; it may behave differently on different systems. The POSIX API specifies the following two functions for getting and setting the scheduling policy:

- Pthread_attr_getsched_policy(pthread attr t *attr, int *policy)
  - *First parameter* > contains pointer to the set of attributes for the thread
  - *Second parameter >* contains a pointer to an integer that is set to the current scheduling policy
- Pthread_attr_setsched_policy(pthread attr t *attr, int policy)
  - *First parameter* > contains pointer to the set of attributes for the thread
  - *Second parameter >* contains an integer value (SCHED_FIFO, SCHED_RR, or SCHED_OTHER)

Both functions return nonzero values if an error occurs.

6.7 Operating-System Examples

6.7.1 Example: Linux Scheduling

With Version 2.5 of the kernel, the scheduler was overhauled to include a scheduling algorithm known as O(1) that ran in constant time regardless of the number of tasks in the system. The O(1) scheduler also provided increased support for SMP systems, including processor affinity and load balancing between processors. During development of the 2.6 kernel, the scheduler was again revised; and in release 2.6.23 of the kernel, the Completely Fair Scheduler (CFS) became the default Linux scheduling algorithm.

Scheduling in the Linux system is based on scheduling classes. Each class is assigned a specific priority. By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes. Standard Linux kernels implement two scheduling classes: (1) a default scheduling class using the CFS scheduling algorithm and(2) a real-time scheduling class.

Rather than using strict rules that associate a relative priority value with the length of a time quantum, the CFS scheduler assigns a proportion of CPU processing time to each task. This proportion is calculated based on the ***nice value*** assigned to each task. Nice values range from −20 to +19, where a numerically lower nice value indicates a higher relative priority. Tasks with

lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values. The default nice value is 0.

CFS doesn't use discrete values of time slices and instead identifies a ***targeted latency***, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency. In addition to having default and minimum values, targeted latency can increase if the number of active tasks in the system grows beyond a certain threshold.

The CFS scheduler doesn't directly assign priorities. Rather, it record show long each task has run by maintaining the virtual run time of each task using the per-task variable **vruntime**. The virtual runtime is associated with a decay factor based on the priority of a task: lower-priority tasks have higher rates of decay than higher-priority tasks. To decide which task to run next, the scheduler simply selects the task that has the smallest vruntime value. In addition, a higher-priority task that becomes available to run can preempt a lower-priority task.

Linux uses two separate priority ranges, one for real-time tasks and a second for normal tasks. Real-time tasks are assigned static priorities within the range of 0 to 99, and normal tasks are assigned priorities from 100 to 139.

6.7.2 Example: Windows Scheduling

Windows schedules threads using a priority-based, preemptive scheduling algorithm. The Windows scheduler ensures that the highest-priority thread will always run. The portion of the Windows kernel that handles scheduling is called the ***dispatcher***. The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The ***variable class*** contains threads having priorities from 1 to 15, and the ***real-time class*** contains threads with priorities ranging from 16 to 31. The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the ***idle thread***.

When a user is running an interactive program, the system needs to provide especially good performance. For this reason, Windows has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows distinguishes between the ***foreground process*** that is currently selected on the screen and the ***background processes*** that are not currently selected. Windows 7 introduced ***user-mode scheduling(UMS***), which allows applications to create and manage threads independently of the kernel. For applications that create a large number of threads, scheduling threads in user mode is much more efficient than kernel-mode thread scheduling, as no kernel intervention is necessary. Earlier versions of Windows provided a similar feature known as ***fibers***, which allowed several user-mode threads (fibers) to be mapped to a single kernel thread.

6.7.3 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling. Solaris uses the system class to run kernel threads, such as the scheduler and paging daemon. Once the priority of a system thread is established, it does not change. The system class is reserved for kernel use.

The fixed-priority and fair-share classes were introduced with Solaris 9. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share scheduling class uses CPU shares instead

of priorities to make scheduling decisions. CPU ***shares*** indicate entitlement to available CPU resources and are allocated to a set of processes.

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1)blocks, (2)uses its time slice, or (3)is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue.

6.8 Algorithm Evaluation

The first problem is defining the criteria to be used in selecting an algorithm. To select an algorithm, we must first define the relative importance of CPU utilization, response time, or throughput. Our criteria may include several measures, such as these:

- Maximizing CPU utilization under the constraint that the maximum response time is 1 second
- Maximizing throughput such that turnaround time is(on average) linearly proportional to total execution time

6.8.1 Deterministic Modeling

One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number to evaluate the performance of the algorithm for that workload. ***Deterministic modeling*** is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Deterministic modeling is simple and fast. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.

6.8.2 Queueing Models

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called queueing-network analysis.

Little's formula, is particularly useful because it is valid for any scheduling algorithm and arrival distribution:

- N = average queue length
- W = average waiting time in the queue
- λ = average arrival rate for new processes in the queue

$$n = \lambda \, x \, W$$

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited.

6.8.3 Simulations

Running simulations involves programming a model of the computer system. The simulator has a variable representing a clock. As this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and soon, according to probability distributions.

We create a ***trace tape*** by monitoring the real system and recording the sequence of actual events. We then use this sequence to drive the simulation. Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs. Simulations can be expensive, often requiring hours of computer time.

6.8.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

The major difficulty with this approach is the high cost. The expense is incurred not only in coding the algorithm and modifying the operating system to support it but also in the reaction of the users to a constantly changing operating system.

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler.

The most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. Another approach is to use APIs that can modify the priority of a process or thread. The downfall of this approach is that performance-tuning a system or application most often does not result in improved performance in more general situations.