Process Management

1. Process Concept

The Process

A process is a program in execution. A processis more than the program code, which is sometimes known as the **text section.**It also includes the current activity, as represented by the value of the **programcounter** and the contents of the processor's registers. A process generally alsoincludes the process **stack,** which contains temporary data (such as functionparameters, return addresses, and local variables), and a **data section,** whichcontains global variables. A process may also include a **heap,** which is memorythat is dynamically allocated during process run time.

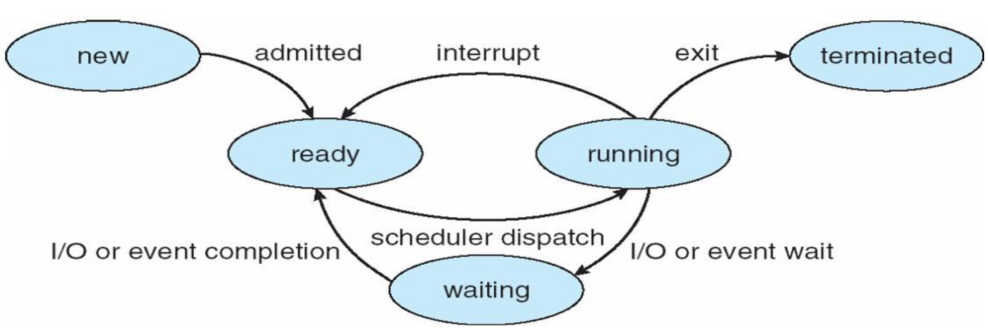
We emphasize that a program by itself is not a process; a program is a *passive*entity, such as a file containing a list of instructions stored on disk (often calledan **executable file),** whereas a process is an *active* entity, with a program counterspecifying the next instruction to execute and a set of associated resources. Aprogram becomes a process when an executable file is loaded into memory.

Process State

As a process executes, it changes **state.** The state of a process is defined inpart by the current activity of that process. Each process may be in one of thefollowing states:

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/Ocompletion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

It is important to realizethat only one process can be *running* on any processor at any instant. Many processes may be *ready* and *limiting,* however. The state diagram correspondingto these states is presented in Figure.



Process Control Block

Each process is represented in the operating system by a **process control block(PCB)**—also called a *task control block.* A PCB is shown in Figure . It containsmany pieces of information associated with a specific process, including these:

• **Process state.** The state may be new, ready, running, waiting, halted, andso on.

• **Program counter.** The counter indicates the address of the next instructionto be executed for this process.

• CPU **registers.** The registers vary in number and type, depending onthe computer architecture. They include accumulators, index registers,stack pointers, and general-purpose registers, plus any condition-codeinformation. Along with the program counter, this state information mustbe saved when an interrupt occurs, to allow the process to be continuedcorrectly afterward.

• **CPU-scheduling information.** This information includes a process priority,pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information.** This information may include suchinformation as the value of the base and limit registers, the page tables,or the segment tables, depending on the memory system used by the operating system.

• **Accounting information.** This information includes the amount of CPUand real time used, time limits, account mimbers, job or process numbers,and so on.

• **I/O status information.**This information includes the list of I/O devicesallocated to the process, a list of open files, and so on.In brief,the PCB simply serves as the repository for any information that mayvary from process to Process.

CPU Switch from Process to Process

2. Process Scheduling

The objective of multiprogramming is to have some process running at alltimes, to maximize CPU utilization. The objective of time sharing is to switch theCPU among processes so frequently that users can interact with each programwhile it is running. To meet these objectives, the **process scheduler** selectsan available process (possibly from a set of several available processes) forprogram execution on the CPU. For a single-processor system, there will neverbe more than one running process. If there are more processes, the rest willhave to wait until the CPU is free and can be rescheduled.

Scheduling Queues

As processes enter the system, they are put into a **job queue,** which consistsof all processes in the system. The processes that are residing in main memoryand are ready and waiting to execute are kept on a list called the **ready queue.**This queue is generally stored as a linked list. A ready-queue header containspointers to the first and final PCBs in the list. Each PCB includes a pointer fieldthat points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated theCPU, it executes for a while and eventually quits, is interrupted, or waits forthe occurrence of a particular event, such as the completion of an I/O request.Suppose the process makes an I/O request to a shared device, such as a disk.Since there are many processes in the system, the disk may be busy with theI/O request of some other process. The process therefore may have to wait forthe disk. The list of processes waiting for a particular I/O device is called adevice **queue.** Each device has its own device queue

The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a**queueing diagram,** such as that in Figure . Each rectangular box representsa queue. Two types of queues are present: the ready queue and a set of devicequeues. The circles represent the resources that serve the queues, and thearrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there tmtil it isselected for execution, or is **dispatched.** Once the process is allocated the CPUand is executing, one of several events could occur:

• The process could issue an I/O request and then be placed in an I/O queue.

• The process could create a new subprocess and wait for the subprocess'stermination.

• The process could be removed forcibly from the CPU, as a result of aninterrupt, and be put back in the ready queue.

Schedulers

A process migrates among the various scheduling queues throughout itslifetime. The operating system must select, for scheduling purposes, processesfrom these queues in some fashion. The selection process is carried out by theappropriate **scheduler.**

The **long-term scheduler,** or **jobscheduler,** selects processes from this pool and loads them into memory forexecution. The **short-term scheduler, or** CPU **scheduler,** selects from amongthe processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequencyof execution. The short-term scheduler must select a new process for the CPUfrequently. A process may execute for only a few milliseconds before waitingfor an I/O request. Often, the short-term scheduler executes at least once every100 milliseconds. Because of the short time between executions, the short-termscheduler must be fast. The long-term scheduler executes much less frequently; minutes may separatethe creation of one new process and the next. The long-term schedulercontrols the **degree of multiprogramming**

It is important that the long-term scheduler make a careful selection. Ingeneral, most processes can be described as either L/O bound or CPU bound. An**I/O-bound process** is one that spends more of its time doing I/O than it spendsdoing computations. A **CPU-bound process,** in contrast, generates I/O requestsinfrequently, using more of its time doing computations. It is important that thelong-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes.

If all processes are I/O bound, the ready queue will almost alwaysbe empty, and the short-term scheduler will have little to do. If all processesare CPU bound, the I/O waiting queue will almost always be empty, devices

will go unused, and again the system will be unbalanced. The system with thebest performance will thus have a combination of CPU-bound and I/O-boundprocesses.

Some operating systems, such as time-sharing systems, may introduce anadditional, intermediate level of scheduling. This **medium-term scheduler** isdiagrammed in Figure . The key idea behind a medium-term scheduler is

that sometimes it can be advantageous to remove processes from memory(and from active contention for the CPU) and thus reduce the degree ofmultiprogramming. Later, the process can be reintroduced into memory, and itsexecution can be continued where it left off. This scheme is called swapping.The process is swapped out, and is later swapped in, by the medium-termscheduler. Swapping may be necessary to improve the process mix or because

a change in memory requirements has overcommitted available memory,requiring memory to be freed up.

Context Switch

An interrupts cause the operating system to change a CPUfrom its current task and to run a kernel routine. Such operations happenfrequently on general-purpose systems. When an interrupt occurs, the systemneeds to save the current **context** of the process currently running on theCPU so that it can restore that context when its processing is done, essentiallysuspending the process and then resuming it.

The context is represented inthe PCB of the process; it includes the value of the CPU registers, the process state (see Figure), and memory-management information. Generically, weperform a **state save** of the current state of the CPU, be it in kernel or user mode,and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a stat^ save of the current process and a state restore of a different process. This task is known as a **context switch.** When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching.

3. Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these

new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems identify processes according to a unique **process identifier** (or **pid),** which is typically an integer number. Figure illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid. In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes—including pageout and f sf lush. These processes are responsible for managing memory and file systems. The sched process also creates the i n i t process, which serves as the root parent process for all user processes.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operatiiigsystem, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

When a process creates a new process, two possibilities exist in terms ofexecution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the sameprogram and data as the parent).
2. The child process has a new program loaded into it.

Process Termination

A process terminates when it finishes executing its final statement and asks theoperating system to delete it by using the exit () system call. At that point, theprocess may return a status value (typically an integer) to its parent process (viathe wait() system call). All the resources of the process—including physical andvirtual memory, open files, and I/O buffers—are deallocated by the operatingsystem.

A parent may terminate the execution of one of its children for a variety ofreasons, such as these:

• The child has exceeded its usage of some of the resources that it has beenallocated. (To determine whether this has occurred, the parent must havea mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child tocontinue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parenthas terminated. In such systems, if a process terminates (either normally orabnormally), then all its children must also be terminated. This phenomenon,referred to as cascading termination, is normally initiated by the operatingsystem.

To illustrate process execution and termination, consider that, in UNIX, wecan terminate a process by using the
e x i t() system call; its parent processmay wait for the termination of a child process by using the waitO systemcall. The wait () system call returns the process identifier of a terminated childso that the parent can tell which of its possibly many children has terminated.

4. Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system.Any process that does not share data with any other process is independent.

A process is **cooperating** if **it** can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

• **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

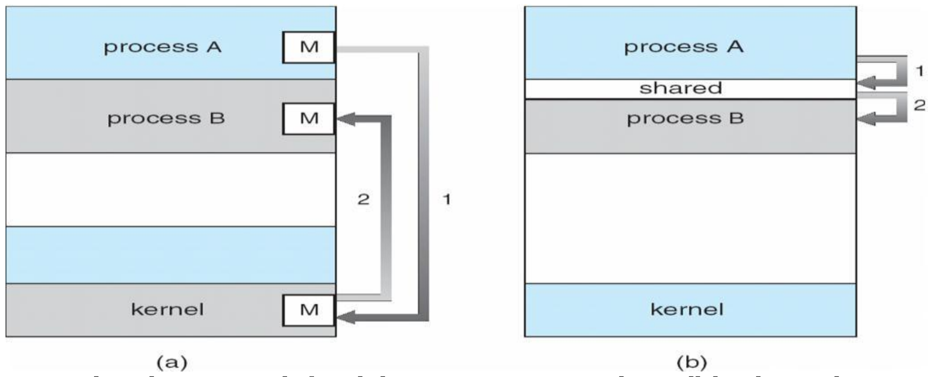
• **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **(1) shared memory** and (2) **message passing.**

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the messagepassingmodel, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure.



Communications models, (a) Message passing, (b) Shared memory.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.

Shared memory is faster than message passing, as message-passing systemsare typically implemented using system calls and thus require the more timeconsumingtask of kernel intervention. In contrast, in shared-memory systems,

system calls are required only to establish shared-memory regions. Once sharedmemory is established, all accesses are treated as routine memory accesses, andno assistance from the kernel is required.

Shared-Memory Systems

Interprocess communication using shared memory requires communicatingprocesses to establish a region of shared memory. Typically, a shared-memoryregion resides in the address space of the process creating the shared-memorysegment. Other processes that wish to communicate using this shared-memorysegment must attach it to their address space. They can then exchange information by reading and writingdata in the shared areas. The form of the data and the location are determined bythese processes and are not under the operating system's control. The processes

are also responsible for ensuring that they are not writing to the same locationsimultaneously.

To illustrate the concept of cooperating processes, let's consider theproducer-consumer problem, which is a common paradigm for cooperatingprocesses. A **producer** process produces information that is consumed by a**consumer** process.

Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating

processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways.
Here are several methods for logically implementing a link and the send()/receive () operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

We look at issues related to each of these features next.

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:

• send(P, message)—Send a message to process P.

• receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive () primitives are defined as follows:

• send(P, message)—Send a message to process P.

• receive(id, message)—-Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mailbox has a unique identification.

Two processes can communicate only if the processes have a shared mailbox, however. The sendC) and receive () primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

In contrast, a mailbox that is owned by the operating system has anexistence of its own. It is independent and is not attached to any particularprocess. The operating system then must provide a mechanism that allows aprocess to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default.Initially, the owner is the only process that can receive messages through thismailbox. However, the ownership and receiving privilege may be passed toother processes through appropriate system calls. Of course, this provisioncould result in multiple receivers for each mailbox.

Synchronization

Communication between processes takes place through calls to sendO andreceive () primitives. There are different design options for implementingeach primitive. Message passing may be either **blocking** or **nonblocking**also known as **synchronous** and **asynchronous.**

• **Blocking send.** The sending process is blocked until the message isreceived by the receiving process or by the mailbox.

• **Nonblocking send.** The sending process sends the message and resumesoperation.

• **Blocking receive.** The receiver blocks until a message is available.

• **Nonblocking receive.** The receiver retrieves either a valid message or anull.

Buffering

Whether communication is direct or indirect, messages exchanged by communicatingprocesses reside in a temporary queue. Basically, such queues can beimplemented in three ways:

• **Zero capacity.** The queue has a maximum length of zero; thus, the linkcannot have any messages waiting in it. In this case, the sender must blockuntil the recipient receives the message.

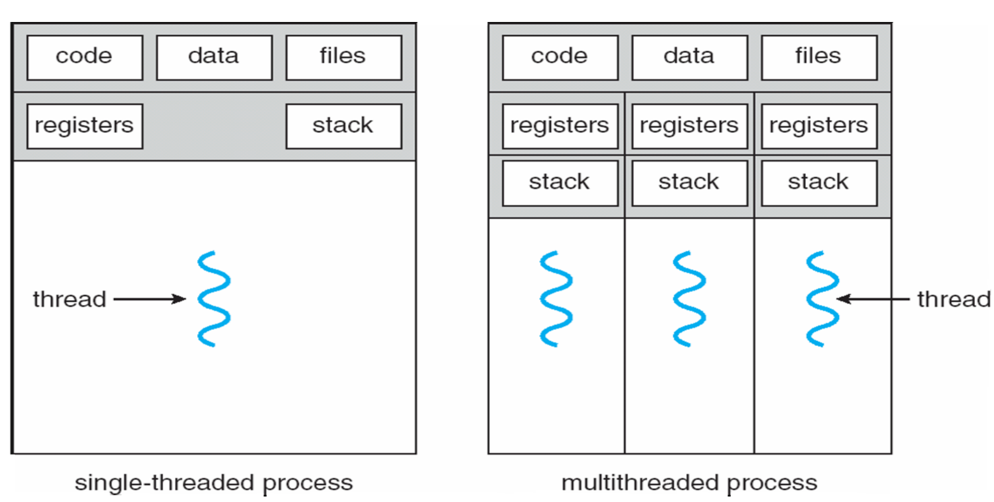
• **Bounded capacity.** The queue has finite length *n;* thus, at most *n* messagescan reside in it. If the queue is not full when a new message is sent, themessage is placed in the queue (either the message is copied or a pointerto the message is kept), and the sender can continue execution withoutwaiting. The links capacity is finite, however. If the link is full, the sendermust block until space is available in the queue.

• **Unbounded capacity.** The queues length is potentially infinite; thus, anynumber of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with nobuffering; the other cases are referred to as systems with automatic buffering.

5. Overview of Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a programcounter, a register set, and a stack. It shares with other threads belongingto the same process its code section, data section, and other operating-systemresources, such as open files and signals. A traditional (or **heavyweight)** processhas a single thread of control. process has multiple threads of control, itcan perform more than one task at a time. Figure illustrates the differencebetween a traditional **single-threaded** process and a **multithreaded** process.



Single-threaded and multithreaded processes.

The benefits of multithreaded programming can be broken down into fourmajor categories:

1. Responsiveness. Multithreading an interactive application may allow aprogram to continue running even if part of it is blocked or is performinga lengthy operation, thereby increasing responsiveness to the user. Forinstance, a multithreaded web browser could still allow user interactionin one thread while an image was being loaded in another thread.

2. Resource sharing. By default, threads share the memory and theresources of the process to which they belong. The benefit of sharingcode and data is that it allows an application to have several differentthreads of activity within the same address space.

3. Economy. Allocating memory and resources for process creation is costly.Because threads share resources of the process to which they belong, itis more economical to create and context-switch threads. Empiricallygauging the difference in overhead can be difficult, but in general it ismuch more time consuming to create and manage processes than threads.

4. Utilization of multiprocessor architectures. The benefits of multithreadingcan be greatly increased in a multiprocessor architecture, wherethreads may be running in parallel on different processors. A singlethreaded process can only run on one CPU, no matter how many areavailable. Multithreading on a multi-CPU machine increases concurrency.

Multithreading Models

Supportfor threads may be provided either at the user level, for **user threads,** or by thekernel, for **kernel threads.** User threads are supported above the kernel andare managed without kernel support, whereas kernel threads are supportedand managed directly by the operating system.

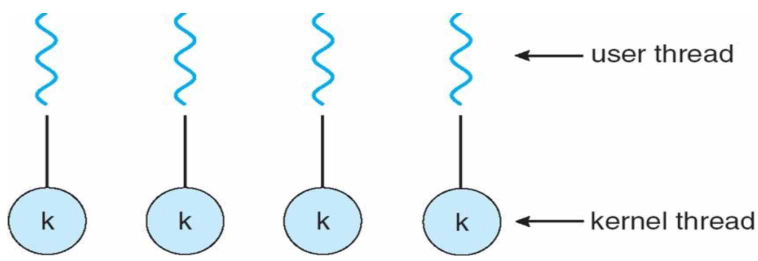
Ultimately, there must exist a relationship between user threads and kernelthreads. In this section, we look at three common ways of establishing thisrelationship.

Many-to-One Model

The many-to-one model maps many user-level threads to onekernel thread. Thread management is done by the thread library in userspace, so it is efficient; but the entire process will block if a thread makes ablocking system call. Also, because only one thread can access the kernel at atime, multiple threads are unable to run in parallel on multiprocessors. **Greenthreads**—a thread library available for Solaris—uses this model, as does GNU**Portable Threads.**

One-to-One Model

The one-to-one model maps each user thread to a kernel thread. Itprovides more concurrency than the many-to-one model by allowing anotherthread to run when a thread makes a blocking system call; it also allowsmultiple threads to run in parallel on multiprocessors. The only drawback tothis model is that creating a user thread requires creating the correspondingkernel thread. Because the overhead of creating kernel threads can burden theperformance of an application, most implementations of this model restrict thenumber of threads supported by the system. Linux, along with the family ofWindows operating systems—including Windows 95, 98, NT, 2000, and XPimplement the one-to-one model.



Many-to-Many Model

The many-to-many model multiplexes many user-level threads toa smaller or equal number of kernel threads. The number of kernel threadsmay be specific to either a particular application or a particular machine (aon a uniprocessor). Whereas the many-to-one model allows the developer tocreate as many user threads as she wishes, true concurrency is not gainedbecause the kernel can schedule only one thread at a time. The one-to-onemodel allows for greater concurrency, but the developer has to be careful notto create too many threads within an application (and in some instances maybe limited in the number of threads she can create). The many-to-many modelsuffers from neither of these shortcomings: Developers can create as many userthreads as necessary, and the corresponding kernel threads can run in parallelon a multiprocessor. Also, when a thread performs a blocking system call, thekernel can schedule another thread for execution.

One popular variation on the many-to-many model still multiplexes manyuser-level threads to a smaller or equal number of kernel threads but also allowsa user-level thread to be bound to a kernel thread. This variation, sometimesreferred to as the *tivo-level model* (Figure), is supported by operating systemssuch as IRIX, HP-UX, and Tru64 UNIX. The Solaris operating system supportedthe two-level model

6. CPU 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
2. When a process switches from the running state to the ready state
3. When a process switches from the waiting state to the ready state
4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is achoice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we saythat the scheduling scheme is **nonpreemptive** or **cooperative;** otherwise, itis **preemptive.** Under nonpreemptive scheduling, once the CPU has beenallocated to a process, the process keeps the CPU until it releases the CPU eitherby terminating or by switching to the waiting state. This scheduling methodwas used by Microsoft Windows 3.x; Windows 95 introduced preemptivescheduling, and all subsequent versions of Windows operating systems haveused preemptive scheduling.

Unfortunately, preemptive scheduling incurs a cost associated with accessto shared data. Consider the case of two processes that share data. While oneis updating the data, it is preempted so that the second process can run. Thesecond process then tries to read the data, which are in an inconsistent state.

Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher.**The dispatcher is the module that gives control of the CPU to the process selectedby the 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 dispatcher should be as fast as possible, since it is invoked during everyprocess switch. The time it takes for the dispatcher to stop one process andstart another running is known as the **dispatch latency.**

Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choiceof a particular algorithm based onMany criteria have been suggested for comparing CPU scheduling algorithms.The criteria include thefollowing:

• **CPU utilization.** We want to keep the CPU as busy as possible. Conceptually,CPU utilization can range from 0 to 100 percent. In a real system, itshould range from 40 percent (for a lightly loaded system) to 90 percent.

• **Throughput.** One measure of CPU work is the number of processes that are completedper time unit, called *throughput.* For long processes, this rate may be oneprocess per hour; for short transactions, it may be 10 processes per second.

• **Turnaround time. It is** how long it takes to execute that process. The intervalfrom the time of submission of a process to the time of completion is the*turnaround time.* Turnaround time is the sum of the periods spent waitingto get into memory, waiting in the ready queue, executing on the CPU, anddoing I/O.

• **Waiting time.** Theamount of time that a process spends waiting in the ready queue. *Waitingtime* is the sum of the periods spent waiting in the ready queue.

• **Response time. T**he time from the submissionof a request until the first response is produced. This measure, called*response time,* is the time it takes to start responding, not the time it takesto output the response. The turnaround time is generally limited by thespeed of the output device.

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

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processesin the ready queue is to be allocated the CPU. There are many different CPUscheduling algorithms. In this section, we describe several of them.

1. First-Come, First-Served Scheduling

The simplest CPU-scheduling algorithm is the **first-come, first-served(FCFS) scheduling algorithm.** With this scheme, the process that requests theCPU first is allocated the CPU first. The implementation of the

FCFS policy iseasily managed with a FIFO queue. When a process enters the ready queue, itsPCB is linked onto the tail of the queue. When the CPU is free, it is allocated tothe process at the head of the queue. The running process is then removed fromthe queue. The code for FCFS scheduling is simple to write and understand.The average waiting time under the FCFS policy, however, is often quitelong. Consider the following set of processes that arrive at time 0, with thelength of the CPU burst given in milliseconds:

Process	**Burst Time**
P1	24
P2	**3**
P3	3

If the processes arrive in the order P1, *P2, P3,* and are served in FCFS order,we get the result shown in the following **Gantt chart:**

P_1 P_2 P_3

0 24 2 30

The waiting time is 0 milliseconds for process Pi, 24 milliseconds for process *Pn,* and 27 milliseconds for process *Pj.* Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order *Pi,* P3, Pi, however, the results will be as showrn in the following Gantt chart:

P_2 P_3 P_1

0 3 6 30

The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reductionis substantial. Thus, the average waiting time under an FCFS policy is generallynot minimal and may vary substantially if the process's CPU burst times varygreatly.

In addition, consider the performance of FCFS scheduling in a dynamicsituation. Assume we have one CPU-bound process and many I/O-boundprocesses. As the processes flow around the system, the following scenariomay result. The CPU-bound process will get and hold the CPU. During thistime, all the other processes will finish their I/O and will move into the readyqueue, waiting for the CPU. While the processes wait in the ready queue, theI/O devices are idle. Eventually, the CPU-bound process finishes its CPU burstand moves to an I/O device. All the I/O-bound processes, which have shortCPU bursts, execute quickly and move back to the I/O queues. At this point,the CPU sits idle. The CPU-bound process will then move back to the readyqueue and be allocated the CPU. Again, all the I/O processes end up waiting inthe ready queue until the CPU-bound process is done.

There is a **convoy effect**as all the other processes wait for the one big process to get off the CPU. Thiseffect results in lower CPU and device utilization than might be possible if theshorter processes were allowed to go first.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has beenallocated to a process, that process keeps the CPU until it releases the CPU, eitherby terminating or by requesting I/O. The FCFS algorithm is thus particularlytroublesome for time-sharing systems, where it is important that each user geta share of the CPU at regular intervals.

2. Shortest-Job-First Scheduling

The **shortest-job-first (SJF) schedulingalgorithm** associates with each process the length of theprocess's next CPU burst. When the CPU is available, it is assigned to the processthat has the smallest next CPU burst. If the next CPU bursts of two processes arethe same, FCFS scheduling is used to break the tie. Note that a more appropriateterm for this scheduling method would be the *shortest-next-CPU-burst algorithm,*because scheduling depends on the length of the next CPU burst of a process,rather than its total length.As an example of SJF scheduling, consider the following set of processes,with the length of the CPU burst given in milliseconds:

Process	**Burst Time**
P1	6
P2	8
P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to thefollowing Gantt chart:

P_4 P_1 P_3 P_2

0 3 9 16 24

The waiting time is 3 milliseconds for process *P1,* 16 milliseconds for process*P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4. Thus, theaverage waiting time is (3 + 16 + 9 + 0)/4 - *7* milliseconds. By comparison, ifwe were using the FCFS scheduling scheme, the average waiting time wouldbe 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal,* in that it gives theminimum average waiting time for a given set of processes. Moving a shortprocess before a long one decreases the waiting time of the short process morethan it increases the waiting time of the long process. Consequently, the *average*waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the nextCPU request. There is no way to know the length of the nextCPU burst. One approach is to try to approximate SJF scheduling. We may not*know* the length of the next CPU burst, but we may be able to *predict* its value.We expect that the next CPU burst will be similar in length to the previous ones.Thus, by computing an approximation of the length of the next CPU burst, wecan pick the process with the shortest predicted CPU burst.The next CPU burst is generally predicted as an exponential average of themeasured lengths of previous CPU bursts. Let *tn* be the length of the »th CPUburst, and let T,,+I be our predicted value for the next CPU burst. Then, for a, 0< a < 1, define

$$T_{n+1} = a t_n + (l - a)T_n.$$

This formula defines an **exponential average.** The value of *tn* contains ourmost recent information; *in* stores the past history. The parameter *a* controlsthe relative weight of recent and past history in our prediction. If a = 0, thenTn,+I =Tn,, and recent history has no effect (current conditions are assumedto be transient); if a = 1, then Tn+1= *tn,* and only the most recent CPU burstmatters (history is assumed to be old and irrelevant). More commonly, a =1/2, so recent history and past history are equally weighted. The initial T0 canbe defined as a constant or as an overall system average.

The SJF algorithm can be either preemptive or nonpreemptive. The choicearises when a new process arrives at the ready queue while a previous process isstill executing. The next CPU burst of the newly arrived process may be shorterthan what is left of the currently executing process. A preemptive SJF algorithmwill preempt the currently executing process, whereas a nonpreemptiTe SJFalgorithm will allow the currently running process to finish its CPU burst.Preemptive SJF scheduling is sometimes called **shortest-remaining-time-firstscheduling.**

As an example, consider the following four processes, with the length ofthe CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If the processes arrive at the ready queue at the times shown and need theindicated burst times, then the resulting preemptive SJF schedule is as depictedin the following Gantt chart:

Process Pi is started at time 0, since it is the only process in the queue. Process*P2* arrives at time 1. The remaining time for process Pi (7 milliseconds) islarger than the time required by process P2 (4 milliseconds), so process Pi ispreempted, and process P2 is scheduled. The average waiting time for thisexample is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds.Nonpreemptive SJF scheduling would result in an average waiting time of 7.75milliseconds.

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 processwith the highest priority. Equal-priority processes are scheduled in FCFS order.An SJF algorithm is simply a priority algorithm where the priority (p) is theinverse of the (predicted) next CPU burst. The larger the CPU burst, the lowerthe priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority.Priorities are generally indicated by some fixed range of numbers, such as 0to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is thehighest or lowest priority. Some systems use low numbers to represent lowpriority; others use low numbers for high priority. This difference can lead toconfusion. In this text, we assume that low numbers represent high priority.

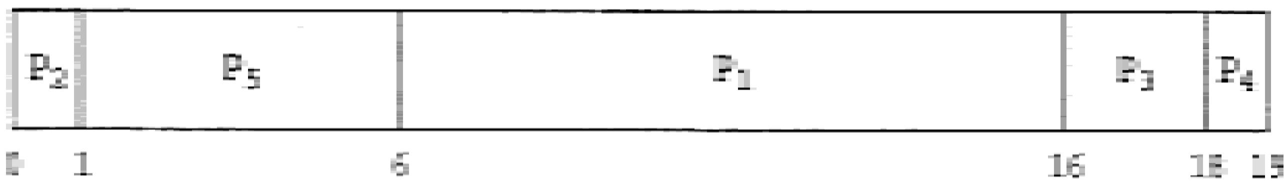
As an example, consider the following set of processes, assumed to havearrived at time 0, in the order P1, P2, • • -, *P5*, with the length of the CPU burstgiven in milliseconds

Process	*Ps*	2
Pi		1
Pi	**Burst Time**	5
P3	10	
PA	1	**Priority**

3 4 2

1 5

Using priority scheduling, we would schedule these processes according to thefollowing Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally definedpriorities use some measurable quantity or quantities to compute the priorityof a process

Priority scheduling can be either preemptive or nonpreemptive. When aprocess arrives at the ready queue, its priority is compared with the priorityof the currently running process. A preemptive priority scheduling algorithmwill preempt the CPU if the priority of the newly arrived process is higherthan the priority of the currently running process. A nonpreemptive priorityscheduling algorithm will simply put the new process at the head of the readyqueue.

A major problem with priority scheduling algorithms is **indefinite blocking,**or **starvation.** A process that is ready to run but waiting for the CPU canbe considered blocked. A priority scheduling algorithm can leave some lowpriorityprocesses waiting indefinitely. In a heavily loaded computer system, asteady stream of higher-priority processes can prevent a low-priority processfrom ever getting the CPU. Generally, one of two things will happen. Either theprocess will eventually be run or the computer system will eventually crash and lose allunfinished low-priority processes.

A solution to the problem of indefinite blockage of low-priority processesis **aging.** Aging is a technique of gradually increasing the priority of processesthat wait in the system for a long time. For example, if priorities range from127 (low) to 0 (high), we could increase the priority of a waiting process by1 every 15 minutes. Eventually, even a process with an initial priority of 127would have the highest priority in the system and would be executed. In fact,it would take no more than 32 hours for a priority-127 process to age to apriority-0 process.

4 Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for timesharingsystems. It is similar to FCFS scheduling, but preemption is added toswitch between processes. A small unit of time, called a **time quantum** or timeslice, is defined. A time quantum is generally from 10 to 100 milliseconds. Theready queue is treated as a circular queue. The CPU scheduler goes around theready queue, allocating the CPU to each process for a time interval of up to 1time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue ofprocesses. New processes are added to the tail of the ready queue. The CPUscheduler picks the first process from the ready queue, sets a timer to interruptafter 1 time quantum, and dispatches the process.One of two things will then happen.

The process may have a CPU burst ofless than 1 time quantum. In this case, the process itself will release the CPUvoluntarily. The scheduler will then proceed to the next process in the readyqueue. Otherwise, if the CPU burst of the currently running process is longerthan 1 time quantum, the timer will go off and will cause an interrupt to theoperating system. A context switch will be executed, and the process will beput at the **tail** of the ready queue. The CPU scheduler will then select the nextprocess in the ready queue.

The average waiting time under the RR policy is often long. Consider thefollowing set of processes that arrive at time 0, with the length of the CPU burstgiven in milliseconds:

Process	**Burst Time**
Pi	24
Pi	3
P3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first4 milliseconds. Since it requires another 20 milliseconds, it is preempted afterthe first time quantum, and the CPU is given to the next process in the queue,process *P2.* Since process P2 does not need 4 milliseconds, it quits before itstime quantum expires. The CPU is then given to the next process, process P3.Once each process has received 1 time quantum, the CPU is returned to processP1 for an additional time quantum. The resulting RR schedule is

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

The average waiting time is 17/3 = 5.66 milliseconds.

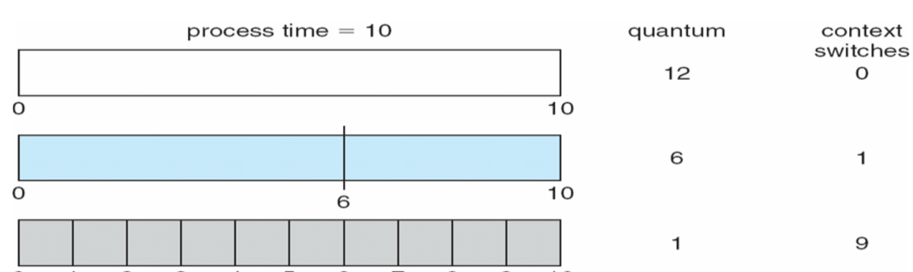
In the RR scheduling algorithm, no process is allocated the CPU for morethan 1 time quantum in a row (unless it is the only runnable process). If aprocess's CPU burst exceeds 1 time quantum, that process is *preempted* and isput back in the ready queue. The RR scheduling algorithm is thus preemptive.If there are *n* processes in the ready queue and the time quantum is *q*,then each process gets *1/n* of the CPU time in chunks of at most *q* time units.

Each process must wait no longer than *(n* — 1) x *q* time units until itsnext time quantum. For example, with five processes and a time quantum of 20milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of thetime quantum. At one extreme, if the time quantum is extremely large, the RRpolicy is the same as the FCFS policy If the time quantum is extremely small(say, 1 millisecond), the RR approach is called **processor sharing** and (in theory)creates the appearance that each of *n* processes has its own processor runningat *1/n* the speed of the real processor.

In software, we need also to consider the effect of context switching on theperformance of RR scheduling. Let us assume that we have only one process of10 time units. If the quantum is 12 time units, the process finishes in less than 1time quantum, with no overhead. If the quantum is 6 time units, however, theprocess requires 2 quanta, resulting in a context switch. If the time quantum is1 time unit, then nine context switches will occur, slowing the execution of theprocess accordingly.

Thus, we want the time quantum to be large with respect to the contextswitchtime. If the context-switch time is approximately 10 percent of thetime quantum, then about 10 percent of the CPU time will be spent in contextswitching. In practice, most modern systems have time quanta ranging from10 to 100 milliseconds. The time required for a context switch is typically lessthan 10 microseconds; thus, the context-switch time is a small fraction of thetime quantum.



Time Quantum and Context Switch Time

Turnaround time also depends on the size of the time quantum. As we cansee from Figure, the average turnaround time of a set of processes doesnot necessarily improve as the time-quantum size increases. In general, theaverage turnaround time can be improved if most processes finish their nextCPU burst in a single time quantum. For example, given three processes of 10time units each and a quantum of 1 time unit, the average turnaround time is29. If the time quantum is 10, however, the average turnaround time drops to20. If context-switch time is added in, the average turnaround time increasesfor a smaller time quantum, since more context switches are required.Although the time quantum should be large compared with the contextswitchtime, it should not be too large. If the time quantum is too large, RRscheduling degenerates to FCFS policy. A rule of thumb is that 80 percent of theCPU bursts should be shorter than the time quantum.

5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations inwhich processes are easily classified into different groups. For example, acommon division is made between **foreground** (interactive) processes and**background** (batch) processes. These two types of processes have differentresponse-time requirements and so may have different scheduling needs. Inaddition, foreground processes may have priority (externally defined) overbackground processes.

A **multilevel queue scheduling algorithm** partitions the ready queue intoseveral separate queues (Figure 5.6). The processes are permanently assigned toone queue, generally based on some property of the process, such as memorysize, process priority, or process type. Each queue has its own scheduling algorithm.

to separate queues might be used for foreground andbackground processes. The foreground quetie might be scheduled by an RRalgorithm, while the background queue is scheduled by an FCFS algorithm.In addition, there must be scheduling among the queues, which is commonlyimplemented as fixed- priority preemptive scheduling. For example, theforeground queue may have absolute priority over the background queue.Let's look at an example of a multilevel queue scheduling algorithm withfive queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues. No process in thebatch queue, for example, could run unless the queues for system processes,interactive processes, and interactive editing processes were all empty. If aninteractive editing process entered the ready queue while a batch process wasrunning, the batch process would be preempted.

6 Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processesare permanently assigned to a queue when they enter the system. If thereare separate queues for foreground and background processes, for example,processes do not move from one queue to the other, since processes do notchange their foreground or background nature. This setup has the advantageof low scheduling overhead, but it is inflexible.

The **multilevel feedback-queue scheduling algorithm,** in contrast, allowsa process to move between queues. The idea is to separate processes accordingto the characteristics of their CPU bursts. If a process uses too much CPU time,it will be moved to a lower-priority queue. This scheme leaves I/O-bound andinteractive processes in the higher-priority queues. In addition, a process thatwaits too long in a lower-priority queue may be moved to a higher-priorityqueue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with threequeues, numbered from 0 to 2. The scheduler first executes allprocesses in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0and 1 are empty. A process that arrives for queue 1 will preempt a process inqueue 2. A process in queue 1 will in turn be preempted by a process arrivingfor queue 0.A process entering the ready queue is put in queue 0. A process in queue 0is given a time quantum of 8 milliseconds. If it does not finish within this time,it is moved to the tail of queue 1. If queue 0 is empty, the process at the headof queue 1 is given a quantum of 16 milliseconds. If it does not complete, it ispreempted and is put into queue 2. Processes in queue 2 are run on an FCFSbasis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPUburst of 8 milliseconds or less. Such a process will quickly get the CPU, finishits CPU burst, and go off to its next I/O burst. Processes that need more than8 but less than 24 milliseconds are also served quickly, although with lowerpriority than shorter processes. Long processes automatically sink to queue2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

n general, a multilevel feedback-queue scheduler is defined by thefollowing parameters:

• The number of queues

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higherpriorityqueue

• The method used to determine when to demote a process to a lowerpriorityqueue

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

The definition of a multilevel feedback-queue scheduler makes it the mostgeneral CPU-scheduling algorithm.

7. Algorithm Evaluation

The first problem is defining the criteria to be used in selecting an algorithm.As we saw , criteria are often defined in terms of CPU utilization,response time, or throughput. To select an algorithm, we must first definethe relative importance of these measures. Our criteria may include severalmeasures, such as:

- Maximizing CPU utilization under the constraint that the maximumresponse time is 1 second
- Maximizing throughput such that turnaround time is (on average) linearlyproportional to total execution time

Once the selection criteria have been defined, we want to evaluate thealgorithms under consideration. We next describe the various evaluationmethods we can use.

1 Deterministic Modeling

One major class of evaluation methods is **analytic evaluation.** Analyticevaluation uses the given algorithm and the system workload to produce aformula or number that evaluates the performance of the algorithm for thatworkload.

One type of analytic evaluation is **deterministic modeling.** This methodtakes a particular predetermined workload and defines the performance of eachalgorithm for that workload.

Deterministic modeling is simple and fast. It gives us exact numbers,allowing us to compare the algorithms. However, it requires exact numbers forinput, and its answers apply only to those cases.

2 Queuing Models

On many systems, the processes that are run vary from day to day, so thereis no static set of processes (or times) to use for deterministic modeling. Whatcan be determined, however, is the distribution of CPU and I/O bursts. Thesedistributions can be measured and then approximated or simply estimated. Theresult is a mathematical formula describing the probability of a particular CPUburst. Commonly, this distribution is exponential and is described by its mean.

Similarly, we can describe the distribution of times when processes arrive inthe system (the arrival-time distribution). From these two distributions, it ispossible to compute the average throughput, utilization, waiting time, and soon for most algorithms.

Queueing analysis can be useful in comparing scheduling algorithms,but it also has limitations. At the moment, the classes of algorithms anddistributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with.

3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use**simulations.** Running simulations involves programming a model of thecomputer system. Software data structures represent the major componentsof the system. Simulations can be expensive, often requiring hours of computer time. Amore detailed simulation provides more accurate results, but it also requiresmore computer time. In addition, trace tapes can require large amounts ofstorage space. Finally, the design, coding, and debugging of the simulator canbe a major task.

4 Implementation

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

The major difficulty with this approach is the high cost. Another difficulty is that the environment in which the algorithm is usedwill change. The environment will change not only in the usual way, as newprograms are written and the types of problems change, but also as a resultof the performance of the scheduler.