Chapter 5
Process Scheduling

Revised: Feb. 7, 2011

5.1 INTRODUCTION

Because a computer system has a limited set of resources that must be shared in a controlled manner among its processes, the operating system is responsible for assigning these resources to programs to meet their needs in a fair and reasonable way. Determining the best sequence for reassigning resources, and choosing from a set of competing requests for use of a resource, are the problems addressed by **scheduling**.

We described the basic characteristics of the process model in a previous chapter. Since most scheduling is performed on behalf of processes, it is natural to refer to the scheduling problem in general as **process scheduling**.

Any time there are more processes in need of a given resource than there are instances of the resource, a scheduling decision must be made. However, the resources that are most heavily in demand and therefore most in need of scheduling are processors and memory. Every process needs a processor and some memory space in order to execute. Since memory is limited and often only one processor is available, competition is continuous and inevitable. Processes do not request these resources; they are assumed to require them at all times.

Scheduling the use of processors and memory is the subject of this chapter. We assume a computer system with a single processor. Processor and memory scheduling proceed continuously at several levels. The highest level concerns the admission of complete jobs to begin competing for resources; the lowest level concerns the selection of processes for the next immediate turn at use of the processor. Memory use may be scheduled at a level in between these two. Each scheduling level is introduced in the next section.

To achieve satisfactory scheduling, it is necessary to treat all processes fairly, but also to recognize and respond to the unique characteristics and needs of each process. The basic categories of processes were introduced previously. In the sections below we will analyze these categories to determine how their properties should influence the choice of suitable scheduling strategies.

We will then discuss the possible objectives of a scheduling strategy, and consider how we may compare strategies and determine if a given strategy is satisfactory. We also introduce several new concepts that play an important role in scheduling strategies, and discuss specific strategies, with examples, that may be used at each level for each of the three process categories.

The final two sections of the chapter discuss some combined and special-purpose strategies, and additional issues that may arise in the actual implementation of a practical scheduling mechanism.

5.2 LEVELS OF SCHEDULING

Scheduling of processor and memory use by processes in an OS proceeds at several distinct levels. Three levels are most commonly identified; each will be described in this section. The terminology we use to describe these levels varies widely. We will choose appropriate names for each level, and also list other names in common use.

We will call the highest level of process scheduling **long-term scheduling** or **job scheduling**. (Other terms used for long-term scheduling include high-level scheduling or admission scheduling.) The objective of long-term scheduling is to control the order in which new processes are admitted to the system. When a job is admitted, a process is created or assigned to perform its activities and the PCB is initialized. The new process may need to compete for its initial resource needs before actual execution can begin.

The next level of scheduling, **medium-term scheduling**, (also called intermediate-level scheduling), is concerned with selecting the set of existing processes that can compete actively for use of the processor. This set of processes is sometimes called the **active set**. This set may need to be limited due to the amount of memory space available for programs and data. To schedule this space effectively, inactive processes may be subject to **swapping**, that is, temporary copying of their programs and data to disk storage to make space available for additional processes. Since the set of active processes is the set of processes that are permitted to occupy main memory, the name **storage scheduling** is also appropriate for this level.

In some cases, processes are assigned memory when they are admitted to the system, and occupy that space continuously until complete. There is no distinct storage scheduling; its effect is performed once only by the job scheduler. In other cases, however, processes that have had a sufficient turn at execution, or need to wait a long time for some event, may be declared inactive and subject to swapping.

The lowest level of process scheduling, **short-term scheduling** (also called low-level scheduling, CPU scheduling, or **processor scheduling**), is concerned with establishing the order in which ready processes are to be assigned a turn at the processor, and selecting and preparing each process for actual execution. Often these two steps are considered distinct, and are performed by different modules of the OS. Maintaining the active set of processes and establishing the execution order is the province of a **scheduler module**, while preparing each process for execution is performed by a dispatcher. The scheduler, which can run at relatively convenient times, does as much of the decision making and preparation as possible. The **dispatcher**, which must be ready to go with high efficiency, performs only the simplest tasks.

-

The progress of a typical process through these various levels of scheduling is diagrammed in Figure 5-1. The figure is a time chart in which time flows from top to bottom. The process follows the solid line in moving through various status regions. The boundary between each pair of regions is controlled by a distinct scheduling level.

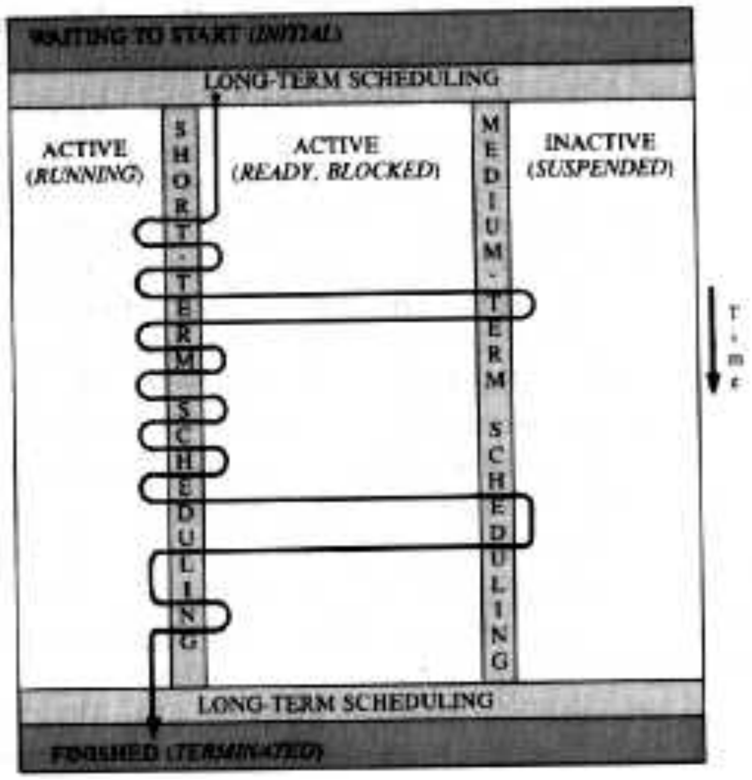

Figure 5-1: The Life of a Typical Process

Each region represents one or more process states. Thus, we should examine the progress of the example process with the help of a process state diagram, such as the one in the Process Management chapter. A slight variation of that figure appears in Figure 5-2.

The regions of Figure 5-1 correspond to process states except that, for simplicity, no distinction is shown between ready and blocked states (or their suspended counterparts). The process begins in the initial state. When admitted by the long-term scheduler, it is assigned its initial memory needs and placed in the ready state. The process remains for a period of time in the active set, alternating between the running state and the other active states (ready and blocked) under control of the short-term scheduler.

When the process needs to wait for a longer time, it may be switched to an inactive state. Even if there is no need to wait, processes may still be rotated out of the active set to provide fair sharing of the available memory. The example process periodically moves in and out of the inactive states (suspended/blocked and suspended/ready) as directed by the medium-term scheduler.

Finally, when the process chooses to exit, or is aborted due to some abnormal condition, it enters the terminal state. It may survive in this state for a while to make final status information available; it is then destroyed.

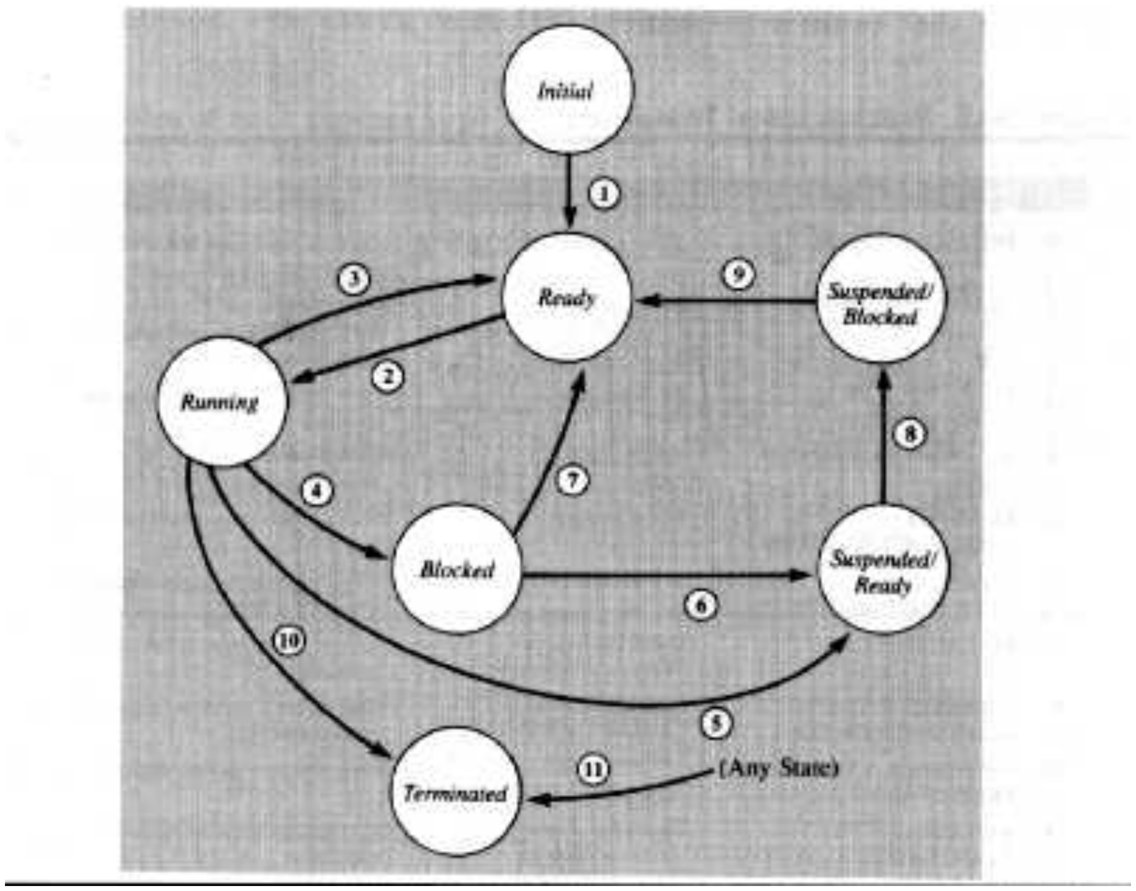

Figure 5-2: A Typical Process State Diagram

Figure 5-2 also shows each of the transitions that may reasonably occur between these states. The transitions actually permitted for a particular process may depend on the process category and the OS. The table in Figure 5-3 lists each allowable transition, the OS module that controls it, and the conditions that cause it to occur.

TRANSITION	MODULE	REASON
INITIAL --> READY	LT Scheduler	When the job's turn has arrived and initial resources are available
READY --> RUNNING	ST Dispatcher	When the process' turn arrives, the processor is available, and no higher priority process is ready

-

RUNNING --> READY	ST Scheduler	When the processor time quantum has expired
RUNNING --> BLOCKED	Process and ST Scheduler	When a resource or service has been requested
RUNNING --> SUSPENDED/BLOCKED	MT Scheduler	When storage time quantum has expired
BLOCKED --> SUSPENDED/BLOCKED	MT Scheduler	When the event being awaited is not expected soon
BLOCKED --> READY	Event Handler or Resource Scheduler	When awaited event occurs or resource is available
SUSPENDED/BLOCKED --> SUSPENDED/READY	Event Handler or Resource Scheduler	When awaited event occurs or resource is available
SUSPENDED/READY --> READY	MT Scheduler	When turn arrives and storage is available
RUNNING --> TERMINATED	Process or Timer Interrupt	On voluntary termination or time limit expiration
(non RUNNING) --> TERMINATED	Other Process or Condition Handler	When terminated by another controlling process, or due to an extraordinary system condition

Figure 5-3: Process State Transitions

5.3 PROCESS CATEGORIES

In most operating systems, each application process may be considered to be in one of three categories: batch, interactive, or real-time. The difference between these categories was introduced previously, as illustrated in Figure 5-4. Here we will examine their characteristics from a scheduling point of view.

Batch processes represent jobs that are submitted to the system as a whole, and expected to produce some output as a whole when they are complete. These jobs may access files and storage devices, but there is no interaction with users or the external world during their

-

execution. Typically the input to such a job consists of programs, data, and job control statements, some of which may be read from files. The usual output is a data file or printed listing. Users will judge the effectiveness of a batch scheduler by its turnaround time, that is, the delay between job submission and the output of the final results.

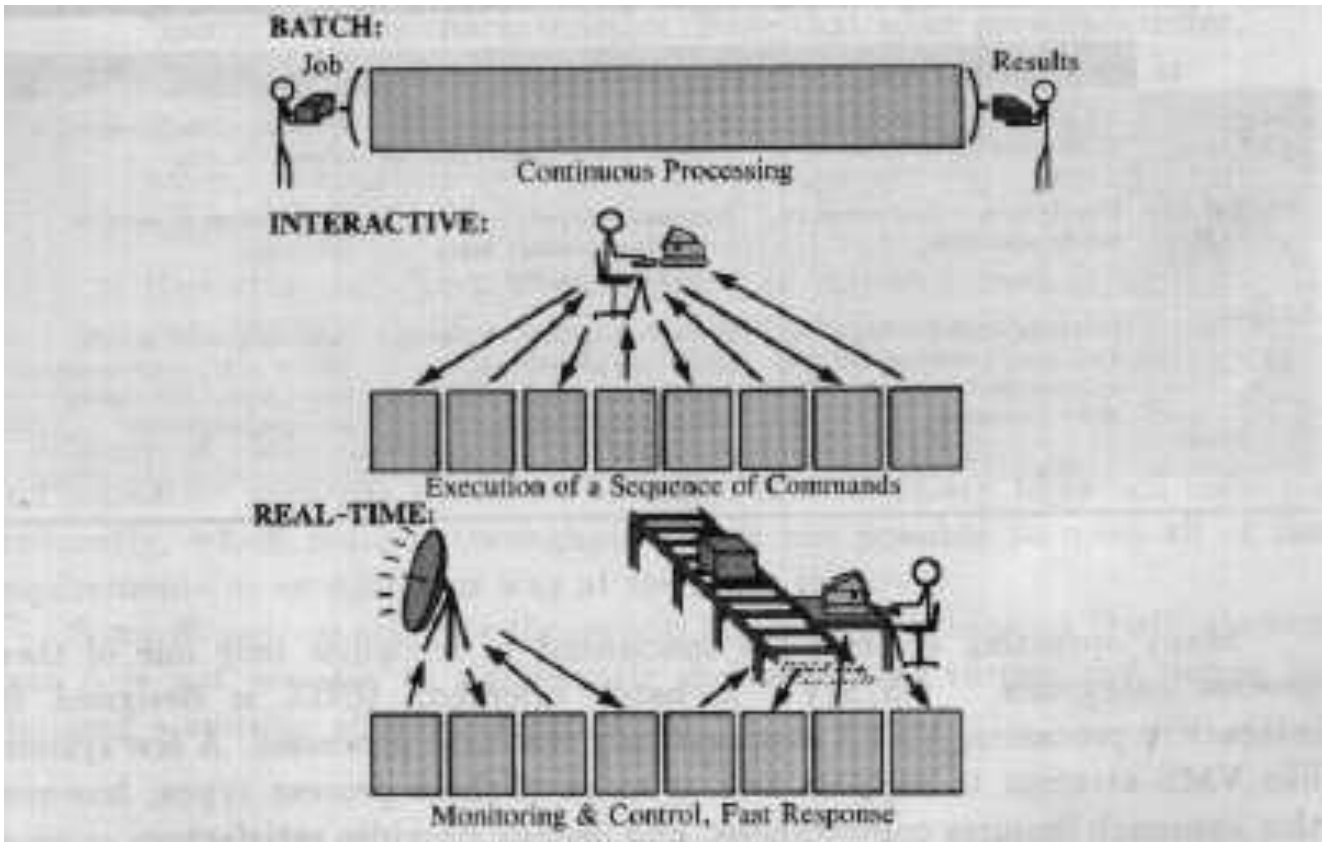
BATCH:
Job
Results
Continuous Processing
INTERACTIVE:
Execution of a Sequence of Commands
REAL-TIME:
Monitoring & Control, Fast Response

Figure 5-4: Process Categories

Batch processes are often submitted with good estimates of their expected resource usage, including memory requirements and anticipated running time. These estimates are used for long term scheduling. A job may not be admitted until the required resources are available. Resource estimates may also influence the priority assigned to a job; if resource needs are modest, a higher priority may be justified. In addition, jobs may include instructions that explicitly determine the relative importance and scheduling requirements of different processes. The user measures the effectiveness of batch scheduling by the speed with which final results are returned. Since most batch computations have substantial time and resource requirements, equally sizable processing delays may be accepted.

Interactive processes are assumed to perform work on behalf of a user who remains present at an interactive terminal. Many of these processes have very short running times and resource needs. They may display information for the user as they progress, or require terminal input at various points in their processing. Interactive processes are rarely provided with information on

-

resource use or scheduling requirements. The user initiating such a process expects rapid and consistent response throughout its execution, not only when the process is complete.

Real-time processes are designed to interact with external events and objects, such as tracking air traffic or controlling the flow of chemicals in a laboratory experiment. The resource requirements of real-time processes are generally known. They are further characterized by absolute timing requirements, which must be met regardless of other activities in the total system.

Each type of process requires a different approach to scheduling. The importance of each of the scheduling levels also varies depending on the process category. Figure 5-5 summarizes the interaction of scheduling levels and process types. Note that if the OS supports more than one category of process, the dominant category may determine the methods employed. We will consider principles and examples for scheduling each process type in later sections of this chapter.

	BATCH	**INTERACTIVE**	**REAL-TIME**
LONG-TERM	Job admission, based on characteristics and resource needs.	Sessions and processes normally accepted unless capacity reached	Processes either permanent or accepted at once
MEDIUM-TERM	Usually none. Jobs remain in storage until done	Processes swapped on rotating basis when necessary, using storage time quantum	Processes never swapped or suspended
SHORT-TERM	Processes scheduled by priority. They usually continue until they wait voluntarily, request service, or are terminated.	Processes scheduled on a rotating basis. They continue until service is requested or processor time quantum expires. Optional preemption.	Scheduling based on strict priority with immediate preemption. Optional time sharing among equal priority processes

Figure 5-5: Process Categories and Scheduling Levels

Many operating systems are specialized to recognize only one of these process categories. OS/MVT was batch oriented; UNIX is designed for interactive processes; VxWorks supports only real-time processes. Some systems like Windows attempt to support two or more of these process types; however, this approach requires compromises, and usually provides satisfactory support in, at most, one process category. A more complex scheduling strategy is necessary when several categories of processes are to be supported.

-

5.4 SCHEDULING OBJECTIVES AND MEASURES

Scheduling Objectives

The strategies used for scheduling decisions at each level vary widely. To choose a reasonable strategy, you first need to determine its objectives. The possible objectives for a good scheduling strategy are numerous. Some of the most important ones usually include:

- **Throughput:** Get the most work done in a given (long) time period.
- **Turnaround**: Complete jobs as soon as possible after they have been submitted.
- **Response**: Service individual steps within a process, especially those requiring only short processing, as quickly as possible.
- **Fairness**: Treat each process or job in a way that corresponds "fairly" with its characteristics. Note that when processes differ, there may be disagreement on what the fairest treatment is.
- **Consistency**: Treat processes with given characteristics in a manner which can be predicted and does not vary greatly from time to time.
- **Resource use**: Keep each category of resource used as fully as possible, yet avoid excessive waiting for certain resources.

Some of these objectives are inherently conflicting. To achieve short and consistent response times, for example, it is necessary to switch contexts frequently, which reduces throughput. It is not possible to meet all of the requirements in an optimum way at the same time.

Some objectives also face theoretical limitations. Kleinrock [1965] showed that it is not possible to significantly reduce overall turnaround delays by choosing a suitable scheduling algorithm.

Distinguishing Processes

Meeting the desired objectives requires that processes be distinguished based on various characteristics, which the processes may be known or observed to have. Some of the more important characteristics to be considered include:

- Process category, as discussed in the previous section
- Original priority assigned when the process was created
- Anticipated running time and resource needs
- Running time and resource use so far
- Association with an interactive terminal

-

- Frequency of I/O requests
- Time spent waiting for service

Note that some of these factors are intrinsic properties of each process, while others can change dynamically as the process executes.

Measuring Success

As observed previously, processes waiting for a resource are entered in a queue associated with that resource. In the case of processes waiting for service by the processor, the queue is the ready queue. To evaluate the success of a scheduling algorithm in meeting its goals, the queues of processes waiting for service or resources may be observed over a period of time. Alternately, the behavior of these queues may be simulated in some way or computed by analytical means. Some specific parameters of these queues are often measured. These include:

- **waiting time**, the time a process spends in a queue waiting for service. In general, this time should be as low as possible.
- **queue length**, the size of the queue of waiting requests. Queue length must be kept reasonable to conserve storage.
- **response ratio**, a measure of waiting time balanced against the service time required for each request: $R = W/(W+S)$. A consistent response ratio means that better service is being given to shorter processes, but reasonable service is being given to longer processes. This is usually considered fair.

Each of these parameters varies over time, and each is characterized by a statistical distribution. Properties of this distribution that are interesting and measurable include the **mean**, or average, value; the **variance**, which indicates how widely typical values tend to deviate from the mean; and the **kth percentile value**, which measures the maximum or minimum values encountered for k percent of all observations. Typically, we are interested in values such as k=95 or k=99. Notice that it is not possible to measure true worst case performance (k=100), since in a statistical distribution the worst case is generally unbounded.

5.5 SOME IMPORTANT CONCEPTS

Priorities

We may use many of the characteristics that distinguish one process from another to establish a relative importance among processes. This importance is usually quantified by a number, termed a **priority**, which is assigned to each process. The priority concept allows fast

-

decision making by the various scheduling levels. If two processes with different priorities are candidates for service, or use of a resource, the higher priority is always selected.

The choice of permissible values for the priority is quite variable. MVT allowed priorities to range from 0 to 15, with 0 being the highest. Windows provides priorities 0 through 31; in this case higher numbers indicate higher priorities. A priority value in UNIX may be chosen from a wide range of integers, positive or negative. Lower values indicate higher priorities. Negative priorities as a class are higher than positive ones; these are reserved for system processes.

Processes may be assigned a priority when initially submitted to the system. This initial priority may be used unchanged throughout the lifetime of the process. A scheduling strategy in which the priority of a process never changes is said to be based on **static priorities**. More often, the history of a process is also considered in determining its current priority. In this case, the priority can change with time; for example, a process that has waited a long time for service may be given a higher priority. This approach is said to use **dynamic priorities**.

The priority of a process may be stored explicitly as a number in the PCB. Alternately, it may be represented implicitly. Often this is done by maintaining a separate queue for each priority value, and linking each PCB on the appropriate priority queue (see Figure 5-6).

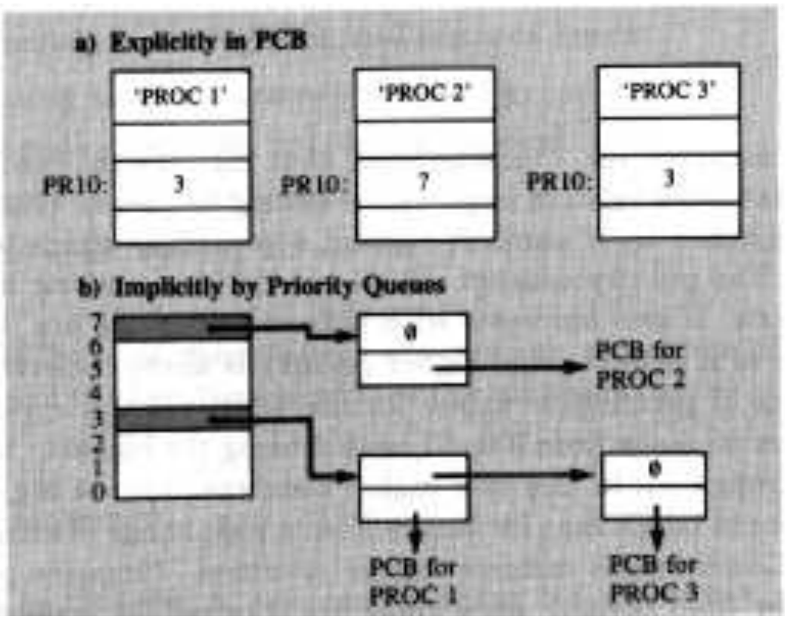

Figure 5-6: Representation of Process Priorities

Preemption

An important property of some dispatching and storage scheduling strategies is the possibility of **preemption**. In a non-preemptive strategy, a process is assigned the processor until

-

it voluntarily gives it up; thus, a single process may continue for a long time (e.g., hours) until it terminates or requests I/O or another type of OS service. Events or requirements associated with other processes cannot cause the processor to be taken away. In a preemptive strategy, when a high- priority process becomes ready, it may be given the processor immediately or within a limited period of time, although a lower- or equal-priority process is currently running. Preemption may occur at both the short-term and medium-term scheduling levels.

Note that even though a process enters the ready queue with a higher priority than the currently running process, there is no guarantee that the current process will be interrupted immediately. The dispatcher is run only when desired by the OS, and no change will take place until these procedures are executed. The OS must decide which events (such as system calls or interrupts) will cause the scheduler to run, leading to a possible change of processes. Because this change must be triggered by selected events, we say that preemptive process switching is **event-driven**.

The presence of preemption has a profound effect on the behavior of a scheduling strategy. For one thing, it tends to favor higher-priority processes more strongly than a non-preemptive method. Some batch-only systems use non-preemptive scheduling. However, preemption is necessary in an interactive environment to maintain reasonable response time for each user. It is even more necessary in a real-time system to meet the absolute timing constraints that are necessary.

Overhead

Often, analyses of dispatching and storage scheduling are simplified by assuming no effort is required to switch between processes. In reality, this context switching will take some time, introducing a performance penalty, called **overhead**, which can have a major effect on total performance.

In short-term scheduling, overhead comes from two sources: the decision making process and the work required to switch contexts. The first of these can be controlled by organizing scheduling queues in such a way that the time required to dispatch the next process is minimal. This work may be done by the scheduler at a reasonably convenient time. The second type of overhead can be reduced by hardware mechanisms that allow a context switch to be completed in a small number of machine instructions.

In medium-term scheduling, although decision making is still involved, the dominant source of overhead (in non-virtual memory systems) comes from swapping memory. The time required to copy the information for one process from memory to disk and load information for another process is a very severe penalty.

In effect, overhead makes it necessary to limit the frequency of context switching, especially when swapping may be required.

-

5.6 SCHEDULING BATCH PROCESSES

A batch job is submitted by a user complete with all information and instructions required for its execution. This initial information includes all input data, identification of files, or input devices from which the data should be taken. The job usually performs a lengthy sequence of computations or file manipulations. There is no interaction with the user while the job is in progress; the user sees only final results.

Batch jobs normally are provided by the user with estimates of their expected run time and resource needs. The user expects the OS to enforce these estimates as upper limits, but give better service to jobs whose resource estimates are low. Because of this, the resource estimates will tend to be generous but fairly accurate. In many cases, users are permitted to specify a base priority for each job, within allowable limits.

From the user's point of view, the most important objective of a batch scheduling strategy is short and reasonably predictable turnaround time. Usually, however, the delays and variance that may be tolerated are much greater than in an interactive environment. Historically, maximum total throughput of the computing system has also been an important goal.

Long-term Scheduling

The initial priorities and resource usage estimates supplied with a batch job lead to significant choices to be made at the job scheduling level. Batch jobs tend to be large consumers of resources, and relatively few of them can be admitted at one time.

The example of batch job admission and initiation shown in Figure 5-7 is based on the operation of MVS. The programs and data required for each job are submitted in a batch, accompanied by job control statements that specify a job name, a class, and a priority. The job is composed of a series of job steps, each of which will run a distinct program. Each job step includes job control statements specifying the resources it will need, including memory, files, and total processor time.

Jobs are read in from various input sources and placed into one of a set of input job queues. One such queue exists for each job class. Classes are assigned according to the nature of the job: one class for short student jobs, another for jobs using a certain compiler, and so on. Jobs in each input queue are ordered by priority and by time of arrival within each priority group.

A set of system processes called initiators examine the input queues when the system is ready to accept more work. Each initiator controls one or more input queues. The initiator selects the next job in a queue and determines if all resources needed for the first job step are available, including enough memory space. If not, the job must wait while jobs from other queues are serviced. When the needed resources are available, they are all assigned to the job step, and it begins execution.

-

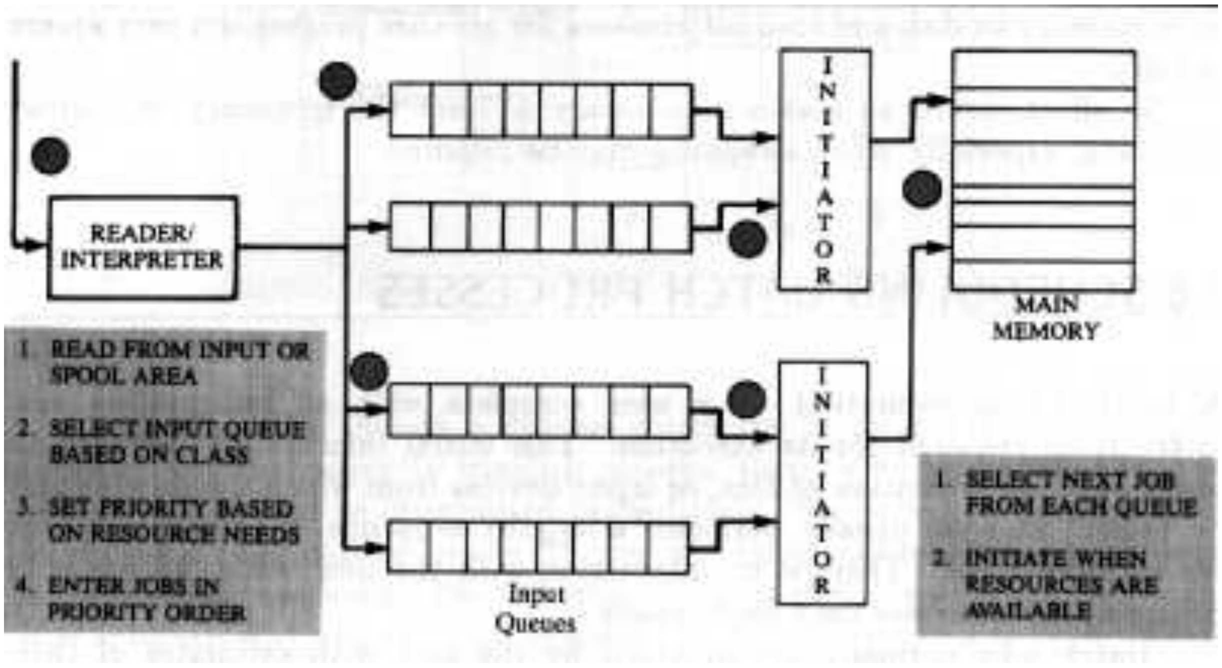
READER/ INTERPRETER
1. READ FROM INPUT OR SPOOL AREA
2. SELECT INPUT QUEUE BASED ON CLASS
3. SET PRIORITY BASED ON RESOURCE NEEDS
4. ENTER JOBS IN PRIORITY ORDER
Input Queues
INITIATOR
INITIATOR
MAIN MEMORY
1. SELECT NEXT JOB FROM EACH QUEUE
2. INITIATE WHEN RESOURCES ARE AVAILABLE

Figure 5-7: Long Term Scheduling of Batch Jobs

Each new step for the job is also admitted only when all necessary resources are available. Once a step is begun, it normally holds all its resources until completion. A timer is set while the process is running to ensure that it does not exceed its planned processor time.

Medium-term Scheduling

In a batch environment, the medium-term scheduling level is usually not distinguished. Jobs are admitted as processes only when sufficient storage is available to meet their announced needs. Thereafter they retain that storage and remain in the active set until they are terminated.

Short-term Scheduling

Short-term scheduling of batch jobs has the following characteristics:

1. When the processor is available, a ready process is selected according to some type of priority.

2. When a process is running, it continues until one of the following events occurs:
    - The process terminates
    - The process voluntarily suspends itself
    - The process requests an I/O transfer or other service for which it is required to wait
    - The process is stopped by the system because it has exhausted its running time
    - A higher-priority process becomes ready and preempts the running process

Preemption is less important in a batch environment because there is no need for a consistent pattern of response time. Because preemption is more complex to manage, increases overhead, and magnifies the difference between processes having distinct priorities, it is not usually used.

Note that with a non-preemptive strategy, it is possible for a process that performs no I/O to complete all of its work without interruption. In many pure batch environments, a single job could run continuously for hours.

The choice of a preemptive or non-preemptive strategy and the assignment of priority values lead to many possible variations. Some common ones are discussed below.

First-in, First-out Scheduling

The simplest dispatching method, based on a first-in, first-out (FIFO) queue with no preemption, is illustrated in Figure 5-8. All ready jobs are considered to have equal priority, and are serviced in the order in which they arrive in the queue. This would be the fairest method if all jobs deserved equal treatment. In practice, however, we usually want to favor some jobs, especially those with limited resource needs.

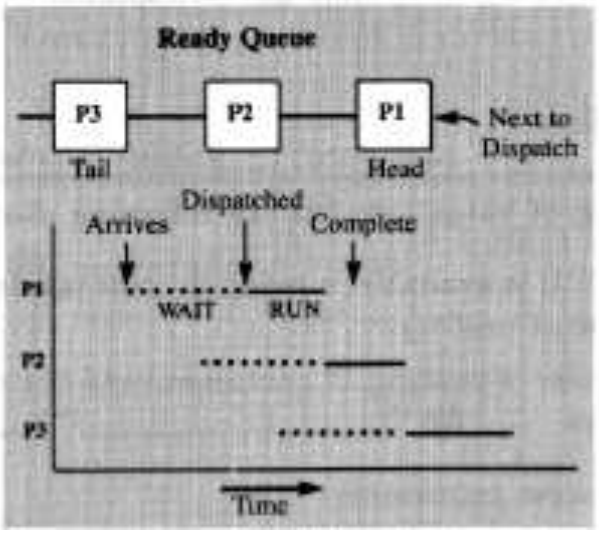

Figure 5-8: Short-Term Batch Scheduling: FIFO

Static Priority Scheduling

Priority dispatching methods select processes according to some priority value. If there is more than one ready process with the highest priority, then the first one in the queue is selected; usually this is the process that has been waiting the longest. This approach is illustrated in Figure 5-9.

-

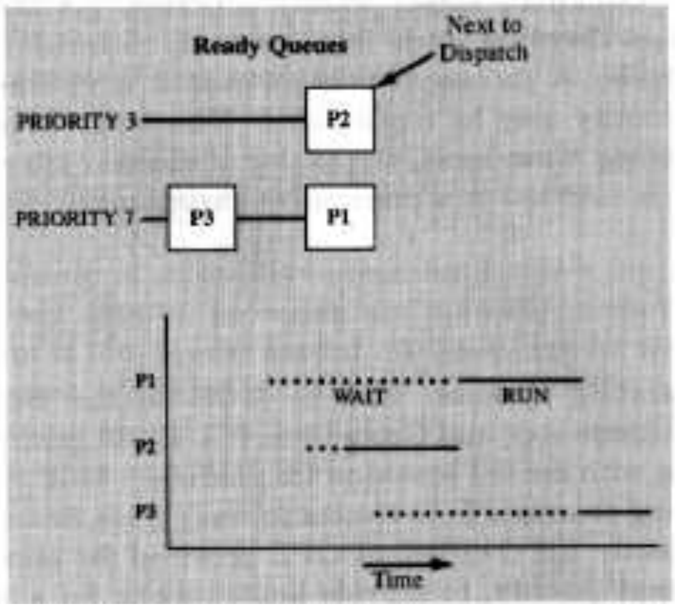

Figure 5-9: Short-Term Batch Scheduling with Priorities

In a static priority method, the priority of a process does not change. The best known example of this class is the **shortest job next (SJN)** algorithm. Here priority is based on the total expected running time of the process, which is known from initial estimates. This expected running time may be adjusted if the process suspends or waits after using part of its time, but it will not change while the process is in the ready queue.

For a given set of processes, the long-term throughput and total waiting time for all non-preemptive scheduling methods is the same. However, SJN can be shown to have the minimum possible average waiting time because most of the waiting is concentrated in a few long jobs, and that experienced by the majority of jobs is much shorter.

This method gives very favorable treatment to short jobs, which is usually desirable. However, it can be excessively unfair to large jobs. Any static priority method is subject to the possibility of **starvation**; if there are always high-priority processes waiting in the ready queue, it is possible that low-priority processes will never get service.

Dynamic Priority Scheduling

Because static methods may be too unfair to low-priority jobs, most actual schedulers use some form of dynamic priority, in which the priority of a process can be increased while the process is waiting for service. With this approach, a low-priority process that is constantly passed over will eventually become a high-priority process, and better treatment is assured.

From a theoretical viewpoint, a good compromise may be to select the process with the highest response ratio. Recall that this is a measure of both the expected service time (that is, running time) and the waiting time of a process. It serves as a priority that gradually increases for all processes, but increases more slowly for long ones.

The response ratio is a good measure in theory, but cannot be used in practice because it varies continuously and must be recalculated for each scheduling decision. A compromise technique called **aging** is usually used instead. This method periodically adds a fixed increment to the priority of each waiting process, so processes that have waited a long time will develop higher and higher priorities, eventually exceeding those of processes that arrived more recently. All waiting processes will be served in a reasonable time, even if their initial priority was very low.

Preemptive Scheduling

If preemption is employed, higher-priority processes get immediate service when ready; thus they are even more strongly favored than with non-preemptive methods. A process that is preempted is returned to the ready queue, and its priority may be recalculated. This leads to methods such as **shortest remaining time next**, the analog of shortest job next. Some type of aging method is essential in a preemptive environment.

Examples

Early batch operating systems, such as MCP on the Burroughs B5500, SCOPE on CDC computers, and George 3 on ICL 1900s, generally made use of FIFO dispatching with limited provision for priorities. MCP considered initial priorities in moving processes from blocked to ready state, but dispatched ready processes in a strictly FIFO order. SCOPE provided for priority sublevels within a single base priority, to provide limited aging for a waiting process. George 3 computed priorities dynamically in a higher-level scheduler, and passed them as fixed values to the dispatcher.

OS/MFT assigned each process a static priority based on running time and other considerations. Preemption was allowed, and there was no provision for aging. OS/MVT used a similar approach, except that the priorities were computed in a different way. MVT systems were often modified by inclusion of the HASP [IBM 1971] scheduling system. HASP substituted a scheduling method very similar to those discussed below for interactive processes.

5.7 SCHEDULING INTERACTIVE PROCESSES

An interactive process---one associated with a user at an interactive terminal---performs computations in response to requests made by the user. Although some of these computations may be quite long, the great majority are for short bursts of service, such as listing directories of editing lines of text.

-

Some OSs represent an interactive session by one continuous process, which executes a succession of programs on request. VMS is a system with this viewpoint. Other systems, such as MULTICS or UNIX, create a new process for each distinct service request.

Nothing is normally known about the resource needs of an interactive process when it is first created. All such processes are initially treated equally. Once a process has run for a while, its pattern of past behavior may be used to infer information about its probable behavior in the future.

The most important objective for most interactive process schedulers is consistent and rapid response. Interactive users expect fast response to each request, especially very short requests; even more important, they expect the response to be consistent and predictable.

Long-term Scheduling

Long-term scheduling of interactive processes involves few decisions. New sessions and newly created processes are normally accepted immediately, unless the system is too heavily loaded. A limit may be established on the number of processes permitted to exist at one time. If a request to create a new process would exceed this limit, the request is rejected. The creating process must try again at a later time.

Time Quanta

To provide adequate response time for a set of interactive processes, the dispatcher must ensure that the processor is shared at a high frequency so that no process waits too long for a turn at service. If storage scheduling is necessary, the available storage must be shared in a similar way. At both of these scheduling levels, the required sharing is enforced using the concept of a **time quantum**, a time limit assigned to a process each time it is selected for service. If the process uses up its time quantum, it is removed from service to give other processes a turn. Distinct time quanta may be used for each process for short-term and medium-term scheduling. The size of the time quanta assigned to each process can vary based on the known characteristics of that process.

Short-term scheduling makes use of a **processor time quantum**. A timer is set to this time limit when the process is selected to run. If the time is exhausted before the process voluntarily gives up the processor, the timer will cause an interrupt, and the processor will be preempted in favor of another waiting process. Typical values for this quantum may range from about 100 milliseconds to several seconds. Use of time quanta to share the processor in this manner is also called time slicing.

Medium-term scheduling makes use of a **storage time quantum**, a value that may be considerably larger than the processor time quantum. Each time a process is placed in the active set by the medium-term scheduler, a counter in its PCB is set to the value of its storage time quantum. When the process completes a turn at the processor, this counter is reduced by the running time the process consumed. A reduction may also be made when the process makes I/O requests. When the storage time quantum expires, the process leaves the active set in favor of other processes. Typical values for the storage time quantum range from a few seconds to several minutes.

-

Medium-term Scheduling

In some of the earliest timesharing systems, medium-term scheduling was identical to short-term scheduling. There was room in main memory for only one process at a time.

In a typical, current interactive environment, our description of the storage time quantum forms the heart of the medium-term scheduling technique for interactive processes. A process that has used up its storage time quantum is placed in an inactive state; inactive processes are not considered candidates for dispatching. The storage used by such a process is marked as available for swapping; it will actually be copied to the swap device only when the space is needed for another purpose.

Inactive processes are selected for return to the active set based on several factors. The assigned priority of a process, its time in the inactive group, and its storage requirements are some of the factors to be considered in making this decision.

Short-term Scheduling

The short-term scheduler in an interactive environment may select jobs based on priorities, or may consider only their order of entry in the ready queue. Preemption of a low-priority process by a higher-priority one may be allowed, but frequently is not. The most striking difference between interactive dispatching and most batch methods is use of the processor time quantum in interactive systems to limit the continuous time any one single process is permitted to run.

A running interactive process may be stopped for all of the reasons listed for batch processes, except expiration of its total running time, which is generally not known. To this list we add one more reason: A process is interrupted by the OS if its processor time quantum has expired.

Round-robin Scheduling

The simplest method for scheduling a set of ready processes in an interactive environment is called **round-robin scheduling**. In this method, illustrated in Figure 5-10, the PCBs of all processes in the ready state are linked on a single queue. Each process is assigned a fixed processor time quantum. When the processor is available, the process at the head of the queue is selected to run. If that process uses up its allotted time, it is interrupted and placed on the back of the queue. Thus, all ready processes continue to be serviced in a circular order. As new processes become ready, they are added to the end of the queue.

-

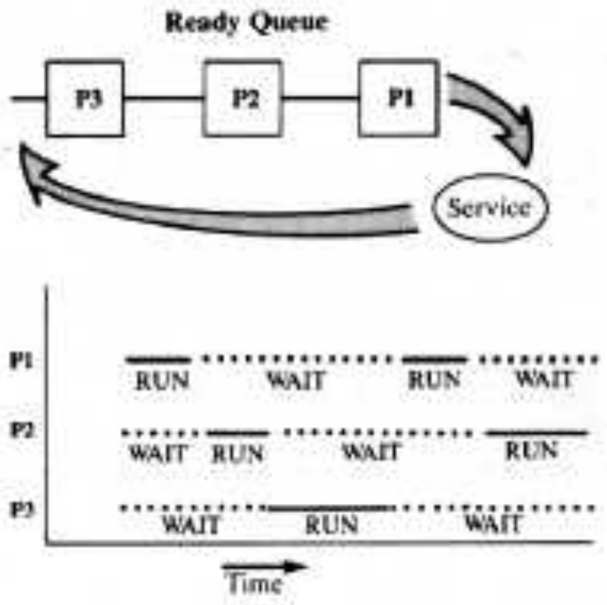

Figure 5-10: Round-Robin Scheduling

Priority Methods

The round-robin approach can be applied to a set of processes with assigned priorities. In this case, processes are maintained in order by priority; within each priority group round-robin dispatching is used. If there are a limited number of possible priority values, a separate queue may be maintained for each priority. Otherwise, a single queue is used, and processes are inserted into the queue by the scheduler so as to keep the queue ordered by priority.

In any case, the process selected to run is dispatched in round-robin fashion from the highest priority group that has a ready process. This strategy is depicted in Figure 5-11.

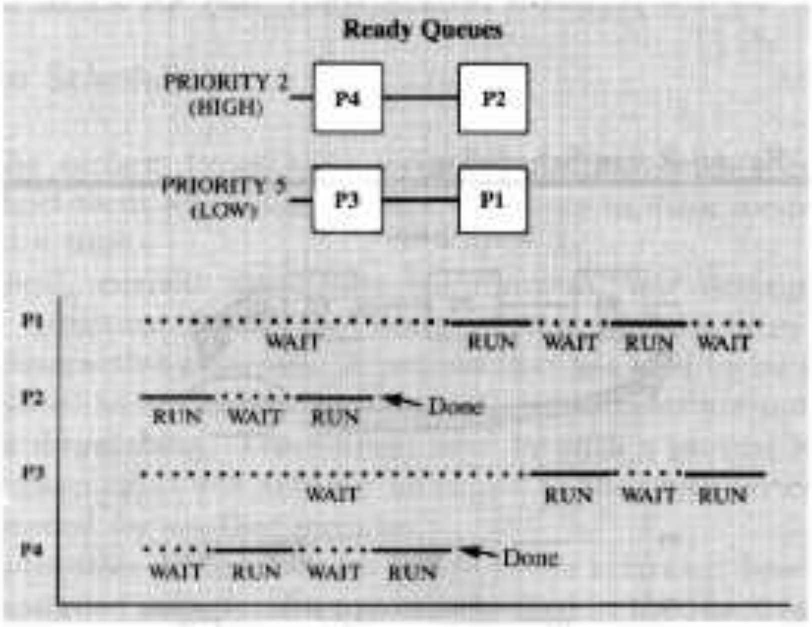

Figure 5-11: Round-Robin Scheduling with Priorities

-

When priority methods are used, the priority of each process may be computed from various factors, and usually will change with time. Often a process that has completed its time quantum will be assigned a lower priority when it is returned to the queue. Often, too, a distinct time quantum size is associated with each priority; as a general rule, lower-priority processes, which are expected to run less frequently but require more time, get longer time quanta.

These considerations lead to many variations in specific priority methods. Some common ones are described below.

Aging Methods

An aging technique similar to that described in the previous section may be used to gradually raise the priority of a process that is waiting a long time for service. Periodically, the process queues are examined; all processes that have received no service during the last time interval are raised to the next-higher-priority level. There is usually an upper bound to the priority permitted for any process.

Feedback Queues

A process performing a long computation will use up many time quanta before it terminates or performs I/O. Such a process is said to be **processor-bound** or **compute-bound**. Conversely, a process performing only one or more short computations will not use up many successive time quanta. A process in this category is called **I/O-bound**.

It is reasonable to give more favorable service to I/O-bound processes by gradually reducing the priority of processes that use up a succession of time quanta without requesting I/O. The method of feedback queues, first used in CTSS, accomplishes this effect. With this method, each time a process is stopped because its quantum has expired, it is moved to a lower-priority level. Since such a process is expected to require longer processor service, the next time it will be given a somewhat longer time quantum. The lowest-priority queue continues to be serviced with round-robin scheduling. **Feedback queue scheduling** is shown in Figure 5-12.

.

-

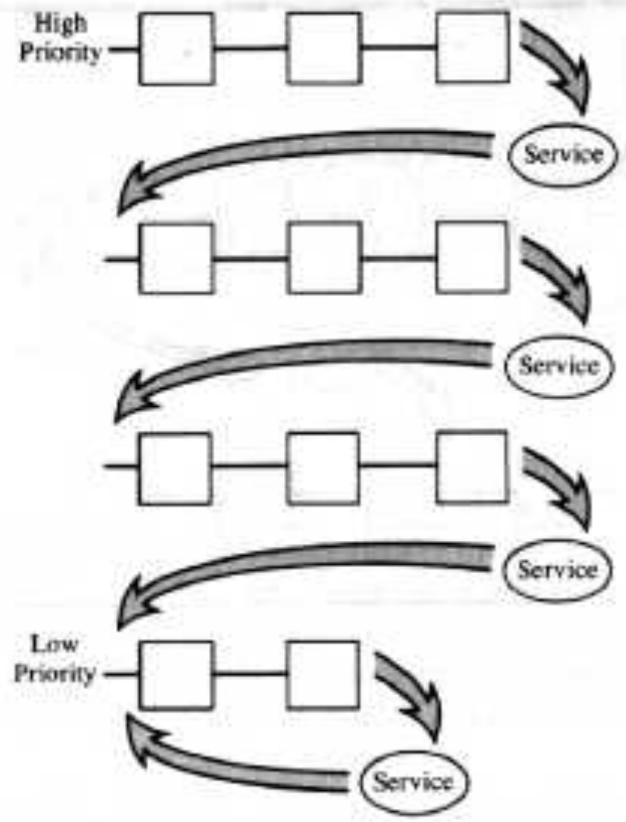

Figure 5-12: Scheduling with Feedback Queues

Selfish Scheduling

An interesting variation on basic interactive short-term scheduling methods is **selfish scheduling**, proposed by Kleinrock [1970]. This strategy favors processes that have been "accepted" for processing. New arrivals must wait to be accepted for service. Once processes are accepted, any of the usual scheduling methods may be used.

In selfish scheduling, as shown in Figure 5-13, processes newly arriving in the ready state are placed in a separate queue. All processes are assigned a special priority value, unrelated to the priority used for normal dispatching. The priority of each new process starts at zero. The priority of both new and accepted processes is increased periodically by an aging technique; however, new processes gain priority at a faster rate. When a new process catches up to the priority value held by the accepted processes (which is identical for all), it is accepted into the main ready queue.

-

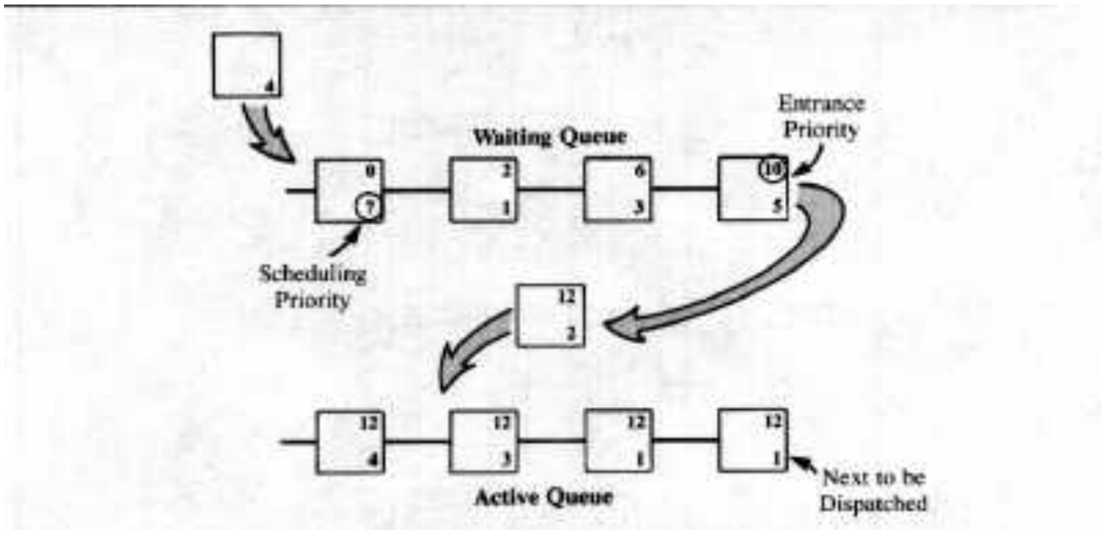

Figure 5-13: Selfish Scheduling

Fair-share Scheduling

Although the scheduling strategies discussed so far generally attempt to be fair, they cannot guarantee a specific level of service for any specific process or group of processes. For various reasons, such as service purchased by customers or real-time response requirements, a guarantee may be necessary. This problem is addressed by the concept of **fair-share scheduling**, depicted in Figure 5-14. A fair-share strategy divides all processes into distinct groups. Although the number of processes in each group may vary, each group is guaranteed an equal share of the available processor time. This objective is not difficult to meet. It requires keeping track of the total processor time used by each group, and assigning higher priorities to processes when their group processor time is low. Fair-share scheduling has been used by VM/370 [Agrawal et al. 1984], and by some variants of UNIX [Henry 1984, Kay & Lauder 1988].

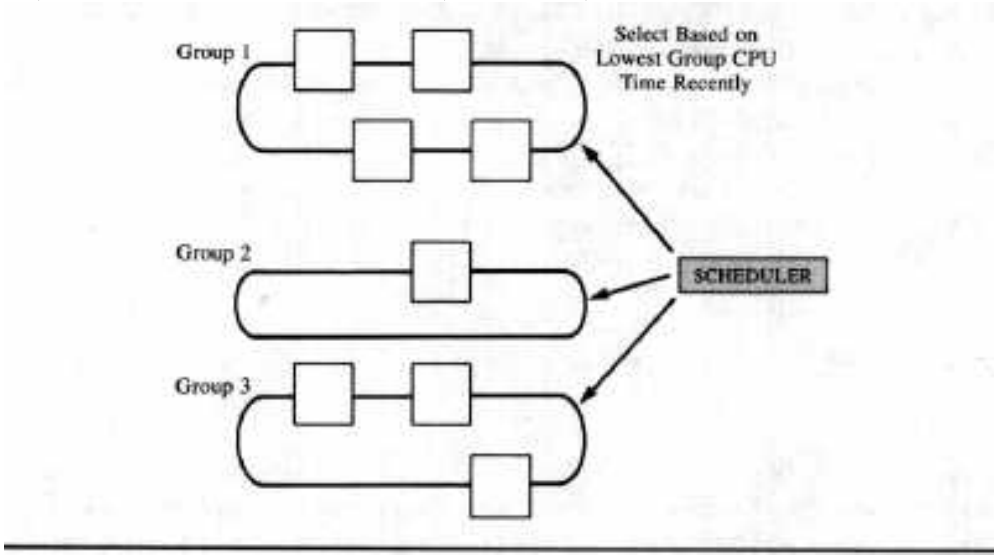

Figure 5-14: Fair-Share Scheduling

Examples

The Unix OS makes use of a single dispatching queue due to its many possible priority values. The priority of user processes is adjusted periodically within permissible ranges, based on the amount of processor time the process used during the most recent real-time interval. This formula favors processes that are not processor-bound, while providing automatic aging when processes have not received service.

Processes are not formally partitioned into active or inactive sets. However, processes may be swapped out when higher-priority processes that have been swapped are ready to run. Processes are also swapped if they suspend voluntarily, or request resources not currently available. The process that has been in memory the longest may be selected for swapping, provided it has had a certain minimum residence time. This time is thus equivalent to a storage time quantum.

CTSS provided nine priority levels, the highest being level 0. Initially, processes running small programs were assigned to level 2, while larger processes began at level 3. The priority of each process was recomputed if its size changed.

Scheduling proceeded in CTSS with multilevel feedback queues. Each lower level received a time quantum twice as large as the previous one. A process could be preempted when a higher-priority one was ready, in which case the preempted process returned to the head of its previous priority queue.

MULTICS used an explicit storage quantum as described above to partition processes into two groups, called "eligible" and "ineligible." Only eligible processes could be selected for running. Ineligible processes were subject to gradual swapping through the virtual memory system.

A multilevel feedback queue method was used in MULTICS for short-term scheduling. As in CTSS, each lower-priority queue had an associated time quantum that was double the previous one. The priority of each process could vary within a fixed range. The range allotted to known interactive processes (those connected to terminals) was higher than that for other processes.

Recent versions of Linux, however, include a feedback mechanism that actually assigns longer time quanta to higher-priority processes. The design of the Linux scheduler is partly based on providing more support for the use of multiple processors. Linux also includes a separate real-time scheduling algorithm.

Windows uses threads as the basic unit of scheduling, but inherits much of its overall scheduling strategy from VMS. This includes the use of 32 levels of priority, with the first 16 reserved for real-time threads. It also makes use of a fairly conventional feedback queuing mechanism. Windows scheduling is discussed further below.

-

5.8 SCHEDULING REAL-TIME PROCESSES

Real-time processes must be scheduled in such a way that absolute timing requirements may be met. This can require very careful scheduling that gives some processes extremely high priority, even higher than parts of the OS itself.

Real-time processes will not be examined in detail in this course. Here we will list some of the characteristics often associated with a real-time scheduling system:

- There is no long-term scheduling. Most processes are permanent. Dynamically created processes are admitted immediately.
- There is no storage scheduling. All processes remain active and resident in main memory at all times.
- Real-time processes are trusted. They may control each other, and they may influence the overall scheduling algorithm.
- The maximum resource needs of real-time processes are usually known. Other information about their expected behavior may be known and utilized as well.
- Strict priority-based scheduling is used. Usually high-priority processes can preempt those of lower priority. However, processes can lock themselves to block preemption.
- Optional timesharing is supported within priority groups. This may be enabled or disabled by any process.
- Processes may be scheduled to run periodically at specific intervals, or to be started in response to certain events.
- Processes may be required to complete an activity by a specific deadline.
- Processes may create threads and schedule these privately within their allotted time.

5.9 COMBINED METHODS

Some operating systems employ scheduling strategies that attempt to service more than one category of process. Windows is an example which recognizes all three categories: batch, interactive, and real-time. As noted above, recent versions of Windows (starting with Windows NT) inherit most of their scheduling strategy from VAX/VMS. As shown in Figure 5-15, Windows makes use of 32 priority levels, ranging from 0 (lowest) to 31 (highest). The choice of 32 levels is not arbitrary; it allows extremely efficient priority dispatching in advanced PC architectures, since the state of each queue (occupied or empty) can be summarized in a single 32-bit word.

The lower half of the priority range, values 0-15, is used for batch or interactive processes. The upper range, 16-31, is reserved for real-time processes. The scheduling strategy differs for the two priority ranges.

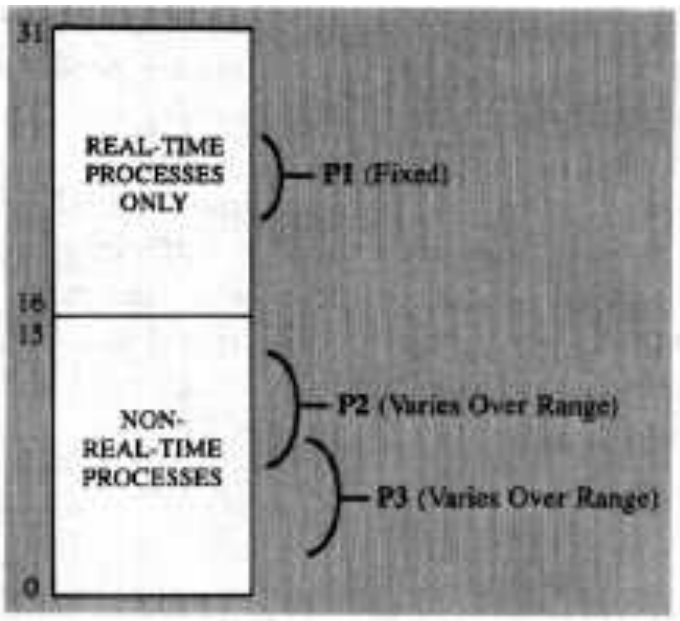

Figure 5-15: Priority Levels in Windows XP

Scheduling in Windows is actually done by threads, with each process assumed to consist of one or more threads. Interactive and batch processes are assigned a base priority and a range above that priority; that range, however, can never exceed level 15. After a process has received service, its priority is set to the lowest value in its range. It is then raised as the process waits. Batch processes in Windows are explicitly identified because they are submitted to special batch queues. However, these processes are scheduled much like interactive ones except that they may receive a lower base priority.

Real-time processes are assigned priorities in the upper range, and their priority never changes. To accommodate these processes, strict preemption is employed. A higher-priority process can always immediately preempt a lower-priority process when it becomes ready. In particular, any real-time process will preempt any non-real-time one.

Linux incorporates a real-time scheduler in addition to the regular process scheduler, which gives priority to real-time processes. Real-time processes have static priorities and are scheduled either on a first-come, first-served or a round-robin basis. Combined approaches like this can support only soft real-time scheduling.

-

5.10 SCHEDULING THREADS

Support for threads as a basic unit of activity has become increasingly common in recent operating systems. In Windows, for example, as noted above, the thread is the basic activity unit. A thread is a subunit of a process or task. Threads share the address space and resources of the process, but maintain independent points of execution.

In a thread-based OS, both processes and threads must be scheduled. System designers currently disagree on the most appropriate type of support to provide for thread scheduling. There are several views that the OS may take:

- The OS schedules processes normally, and schedules threads within each process. If a thread is blocked, then the process is blocked. Usually threads are scheduled in a simple round-robin fashion. All scheduling is handled by the OS, and process characteristics determine their share of processor time. This is an optional approach in UNIX systems that support the POSIX threads model.
- The OS schedules processes normally, and each process privately schedules its own threads. The OS is simpler, but there is a greater burden on the programs that wish to employ threads. This is also an option for POSIX threads.
- The OS schedules only threads. This is the approach taken by Mach and by Windows. In this case every process consists of one or more threads. Priority is a property of threads rather than processes. If a thread is blocked, another thread from the same process may be dispatched. Moreover, threads may be divided into subunits called fibers that receive round-robin scheduling. A drawback of this approach is that a process with many threads may receive a larger share than appropriate of total processor time.

A program making use of multiple threads must be written with explicit tasking and interprocess communication.

5.11 HARDWARE ASSISTANCE

Although the scheduling techniques discussed in this chapter can be implemented on computers having no special hardware support, schedulers can take advantage of selected hardware assistance that is sometimes provided. The most important hardware support features include timers and interrupts. Without these, little effective scheduling is possible.

A substantial impact on scheduling performance can be made by architectures that provide effective assistance for context switching. Techniques like multiple register sets and instructions

-

designed for fast swapping of appropriate registers can allow extremely fast context switches, making it feasible to perform them much more often.

Specialized processor instructions may be provided to speed up several aspects of scheduling algorithms. The VAX architecture provides direct support for insert and remove operations on queues, testing the 32-bit summary word, and other features exploited effectively by VMS.

A few computer types have offered more direct hardware support for scheduling. An early example was the Singer Ten minicomputer, which provided ten independent register sets and automatic round-robin time slicing for ten processes. This support could be quite effective if pure round-robin scheduling was desired and ten processes were sufficient, but was less helpful in supporting other requirements.

A more recent and successful type of hardware assistance was provided by the Inmos Transputer. This architecture supported hardware timesharing with two priority levels using process queues in memory. In addition there was hardware support for queues of actions to be performed at specific times. There was no inherent limit on the number of processes that can be timeshared. The Transputer provided a number of other novel mechanisms to support scheduling, such as use of memory workspaces in place of registers to minimize register swapping when a context switch occurs.

FOR FURTHER READING

A number of studies and comparisons of scheduling techniques for batch and interactive systems have been published. Coffman and Kleinrock [1968] discuss a variety of scheduling methods and ways in which users might be able to "cheat" to obtain better service. A number of timesharing strategies are examined by McKinney [1969] and Kleinrock [1970]. Bunt [1976] surveys batch and interactive techniques, and their use in MFT, MVT, MULTICS, and UNIX. An early description of interactive scheduling based on observed behavior is given by Ryder [1970].

Real-time scheduling issues are discussed, for example, by van der Linden and Wilson [1980].

More theoretical treatments of process scheduling have been given by Coffman and Denning [1973], Brinch Hansen [1973], and Kleinrock [1975]. Brinch Hansen provides a readable analysis of various scheduling algorithms based on queuing theory.

For detailed treatment of process scheduling in specific systems, see Organick [1972] (MULTICS), Kenah and Bate [1984] (VMS), Bach [1986] (UNIX), Love [2005] (Linux), Russinovich and Solomon [2005] (Windows), and McKeag et al. [1976] (MCP, SCOPE, T.H.E., and TITAN).

-

REVIEW QUESTIONS

1. Identify the three major classes of processes that must be distinguished in a process scheduling strategy. For each class, state two important facts about its properties or requirements that distinguish it from other classes.

2. Explain a possible weakness of each of the following short-term scheduling strategies for batch processes: (a) First-in, first-out; (b) Shortest job next

3. Explain why the scheduling levels listed below are not normally used for the indicated categories of processes: (a) Medium-term scheduling for batch processes; (b) Long-term scheduling for interactive processes; (c) Medium-term scheduling for real-time processes.

4. Explain how "processor-bound" processes can be identified in an interactive system, and why these processes should be distinguished.

5. Explain two possible "measures of success" for a scheduling algorithm.

6. State five criteria that can be used to select processes for low-level scheduling. Explain briefly why each is reasonable.

7. Identify the three levels of process and job scheduling, and describe briefly the purpose of each.

8. Name two significant differences between batch processes and interactive processes that affect the low-level scheduling strategy.

9. Which of the following scheduling algorithms may cause some processes to wait indefinitely: FIFO, SJF, HRN, round-robin, LRU.

10. List three types of hardware support for process scheduling provided by some processor architectures.

11. Why do Windows operating systems provide exactly 32 scheduling priority levels?

ASSIGNMENTS

1. Explain why processors and memory are the resources normally managed by the scheduling techniques discussed in this chapter. Under what conditions might scheduling techniques be useful for other resources, such as I/O devices or files? Identify some difficulties that might arise in scheduling the use of such resources.

2. Figures 5-2 and 5-3 identify 11 possible transitions between states. List the transitions that you would expect to see included in a system that primarily

-

supports (a) batch processes, (b) interactive processes, or (c) real-time processes. Justify the transitions you did not include on each list.

3. It has been suggested that a finer distinction should be made among interactive processes according to their application. For example, a process supporting a graphics terminal might require different treatment than one used for routine data entry. How might an operating system recognize different types of processes? What distinctions would be appropriate in scheduling them?

4. The discussion of feedback queues for interactive processes indicates that some processes will tend to be compute-bound, and others will be I/O-bound. In fact, many processes will move between these categories, alternating "bursts" of I/O activity with longer computations. What difficulties will processes of this type cause with feedback queues, if any? What adjustments could be made to account for their characteristics?

5 List the steps required to swap one process to disk storage and replace it by another process. Suggest some design techniques to limit the overhead caused by swapping.

6. Discuss the approach that should be taken in medium- and short-term scheduling of interactive processes if the main memory can hold only a few of the existing processes at a time. What if it can hold almost all of them?

7. Explain one possible advantage and one disadvantage of providing a large number of distinct priority values.

8. Give a possible reason for five of the characteristics of a real-time scheduling algorithm listed in this chapter.

PROJECT

This section describes a two part programming project to construct a simple, round-robin short-term scheduler and dispatcher for a simple multiprogramming operating system, and to install commands to control the loading and scheduling of processes. You should read the general description in this text in conjunction with the more specific descriptions given in your project manual. The scheduler/dispatcher is an important component of the multiprogramming executive (MPX) project.

a. Implement a simple round-robin scheduler/dispatcher using the PCB structure from Chapter 4 to represent each process. Your instructor will supply program files for five test processes to be dispatched. These processes will be linked with your dispatcher and statically loaded. Each time a test process is dispatched, it will display a message to the screen and give up control to the dispatcher, which must save the context and select the next process to be dispatched. When a process has completed, it will indicate termination as specified in the project manual. When a process terminates, the dispatcher should free its PCB in the PCB

-

queue and dispatch the next process. When all PCBs are free, the dispatcher should terminate.

b. Using the dispatcher developed in the previous step, add the commands listed below to the project COMHAN. Your instructor will provide program files for the test processes to be run. In addition, procedures will be supplied to load a file in a specified area, and to list a directory.

1. LOAD *name*: create a new process called *name*, loading the program contained in the file called name into the load area. The process should be initialized to the *suspended/ready* state.
2. RESUME *name*: resume or awaken the application process called *name*. If *name* = *, then all application processes should be resumed.
3. RUN *name*: create a new process called *name*, loading the program from the file called *name* into the load area. The process should be initialized to the *ready* state.
4. SUSPEND *name*: suspend the application process called *name*. If *name* = *, then all application processes should be suspended.
5. TERMINATE *name*: terminate the application process called *name*, and free its PCB. If *name* = *, then all application processes should be terminated.
6. SETPRIORITY *name=nnn*: set the priority of the application process called *name* to *nnn*, where 127 < *nnn* < 127.
7. DIRECTORY: display the names of all programs in your directory that are loadable as processes.
8. DISPATCH: verify that all PCBs are built correctly by dispatching each application process in the PCB queue once in priority order. Each process will display a message to the screen before relinquishing control.

Note that the PCB queue becomes a part of the command handler in this project. The above commands allow PCBs to be built, and processes to be loaded and dispatched in a test environment. This verifies that all process loading is correct before moving to a full interrupt-driven, priority dispatching environment.

-