PART FOUR

Scheduling

An operating system must allocate computer resources among the potentially competing requirements of multiple processes. In the case of the processor, the resource to be allocated is execution time on the processor and the means of allocation is scheduling. The scheduling function must be designed to satisfy a number of objectives, including fairness, lack of starvation of any particular process, efficient use of processor time, and low overhead. In addition, the scheduling function may need to take into account different levels of priority or real-time deadlines for the start or completion of certain processes.

Over the years, scheduling has been the focus of intensive research, and many different algorithms have been implemented. Today, the emphasis in scheduling research is on exploiting multiprocessor systems, particularly for multithreaded applications, and real-time scheduling.

ROAD MAP FOR PART FOUR

Chapter 9 Uniprocessor Scheduling

Chapter 9 concerns scheduling on a system with a single processor. In this limited context, it is possible to define and clarify many design issues related to scheduling. Chapter 9 begins with an examination of the three types of processor scheduling: long term, medium term, and short term. The bulk of the chapter focuses on short-term scheduling issues. The various algorithms that have been tried are examined and compared.

Chapter 10 Multiprocessor and Real-Time Scheduling

Chapter 10 looks at two areas that are the focus of contemporary scheduling research. The presence of multiple processors complicates the scheduling decision and opens up new opportunities. In particular, with multiple processors it is possible simultaneously to schedule for execution multiple threads within the same process. The first part of Chapter 10 provides a survey of multiprocessor and multithreaded scheduling. The remainder of the chapter deals with real-time scheduling. Real-time requirements are the most demanding for a scheduler to meet, because requirements go beyond fairness or priority by specifying time limits for the start or finish of given tasks or processes.

CHAPTER 9

UNIPROCESSOR SCHEDULING

In a multiprogramming system, multiple processes exist concurrently in main memory. Each process alternates between using a processor and waiting for some event to occur, such as the completion of an I/O operation. The processor or processors are kept busy by executing one process while the others wait.

The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved (Table 9.1). One of these, I/O scheduling, is more conveniently addressed in Chapter 11, where I/O is discussed. The remaining three types of scheduling, which are types of processor scheduling, are addressed in this chapter and the next.

This chapter begins with an examination of the three types of processor scheduling, showing how they are related. We see that long-term scheduling and medium-term scheduling are driven primarily by performance concerns related to the degree of multiprogramming. These issues are dealt with to some extent in Chapter 3 and in more detail in Chapters 7 and 8. Thus, the remainder of this chapter concentrates on short-term scheduling and is limited to a consideration of scheduling on a uniprocessor system. Because the use of multiple processors adds additional complexity, it is best to focus on the uniprocessor case first, so that the differences among scheduling algorithms can be clearly seen.

Section 9.2 looks at the various algorithms that may be used to make short-term scheduling decisions.

9.1 TYPES OF PROCESSOR SCHEDULING

The aim of processor scheduling is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency. In many systems, this scheduling activity is broken down into three separate functions: long-, medium-, and short-term scheduling. The names suggest the relative time scales with which these functions are performed.

Figure 9.1 relates the scheduling functions to the process state transition diagram (first shown in Figure 3.9b). Long-term scheduling is performed when a new process is created. This is a decision whether to add a new process to the set of processes that are currently active. Medium-term scheduling is a part of the swapping function. This is a decision whether to add a process to those that are at least partially in main memory and therefore available for execution. Short-term scheduling is the actual decision of which ready process to execute next. Figure 9.2 reorganizes the state transition diagram of Figure 3.9b to suggest the nesting of scheduling functions.

Table 9.1 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

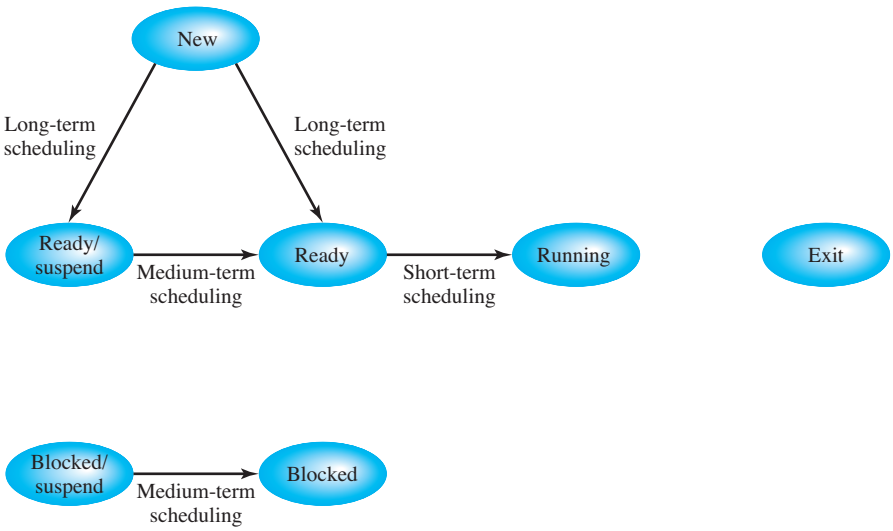


Figure 9.1 **Scheduling and Process State Transitions**

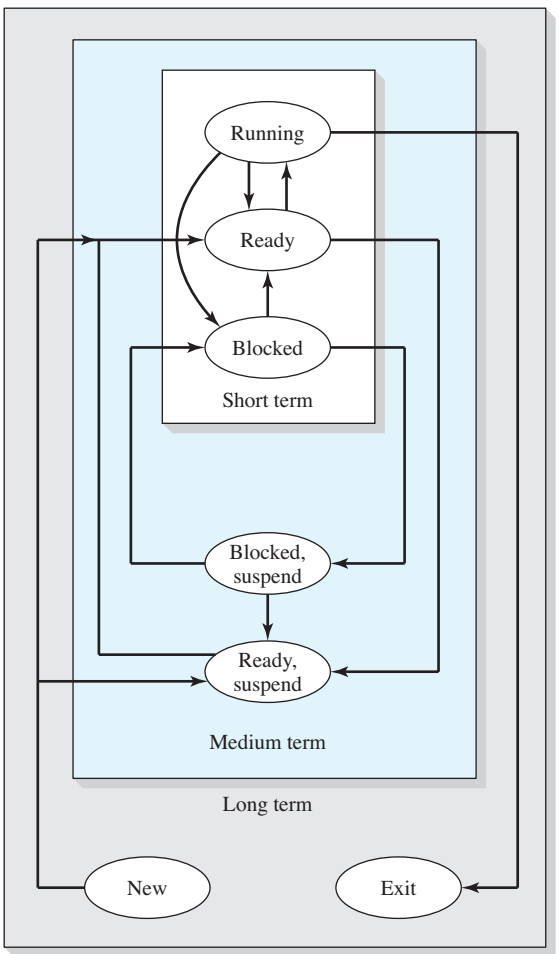


Figure 9.2 **Levels of Scheduling**

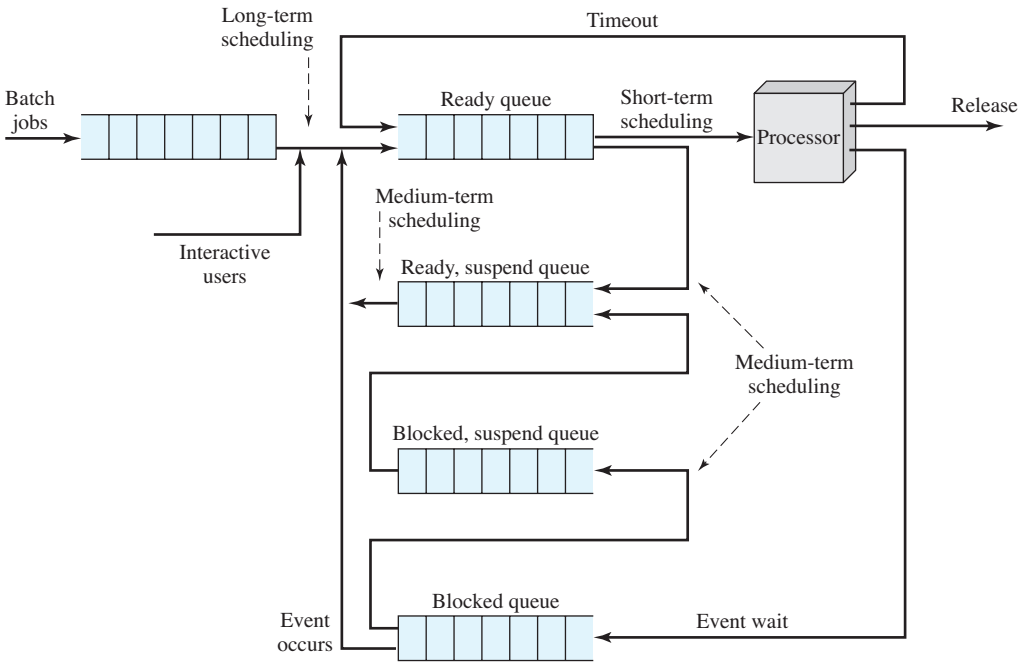


Figure 9.3 **Queuing Diagram for Scheduling**

Scheduling affects the performance of the system because it determines which processes will wait and which will progress. This point of view is presented in Figure 9.3, which shows the queues involved in the state transitions of a process.[1] Fundamentally, scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment.

Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming. Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

In a batch system, or for the batch portion of a general-purpose operating system, newly submitted jobs are routed to disk and held in a batch queue. The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide when the operating system can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. Let us briefly consider these two decisions.

The decision as to when to create a new process is generally driven by the desired degree of multiprogramming. The more processes that are created, the smaller

[1]For simplicity, Figure 9.3 shows new processes going directly to the Ready state, whereas Figures 9.1 and 9.2 show the option of either the Ready state or the Ready/Suspend state.

is the percentage of time that each process can be executed (i.e., more processes are competing for the same amount of processor time). Thus, the long-term scheduler may limit the degree of multiprogramming to provide satisfactory service to the current set of processes. Each time a job terminates, the scheduler may decide to add one or more new jobs. Additionally, if the fraction of time that the processor is idle exceeds a certain threshold, the long-term scheduler may be invoked.

The decision as to which job to admit next can be on a simple first-come-first-served basis, or it can be a tool to manage system performance. The criteria used may include priority, expected execution time, and I/O requirements. For example, if the information is available, the scheduler may attempt to keep a mix of processor-bound and I/O-bound processes.[2] Also, the decision may be made depending on which I/O resources are to be requested, in an attempt to balance I/O usage.

For interactive programs in a time-sharing system, a process creation request can be generated by the act of a user attempting to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the operating system will accept all authorized comers until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function. The issues involved are discussed in Chapters 3, 7, and 8. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

In terms of frequency of execution, the long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process and which one to take. The medium-term scheduler is executed somewhat more frequently to make a swapping decision. The short-term scheduler, also known as the dispatcher, executes most frequently and makes the fine-grained decision of which process to execute next.

The short-term scheduler is invoked whenever an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another. Examples of such events include

- Clock interrupts
- I/O interrupts
- Operating system calls
- Signals (e.g., semaphores)

[2]A process is regarded as *processor bound* if it mainly performs computational work and occasionally uses I/O devices. A process is regarded as *I/O bound* if the time it takes to execute the process depends primarily on the time spent waiting for I/O operations.

9.2 SCHEDULING ALGORITHMS

Short-Term Scheduling Criteria

The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior. Generally, a set of criteria is established against which various scheduling policies may be evaluated.

The commonly used criteria can be categorized along two dimensions. First, we can make a distinction between user-oriented and system-oriented criteria. User-oriented criteria relate to the behavior of the system as perceived by the individual user or process. An example is response time in an interactive system. Response time is the elapsed time between the submission of a request until the response begins to appear as output. This quantity is visible to the user and is naturally of interest to the user. We would like a scheduling policy that provides "good" service to various users. In the case of response time, a threshold may be defined, say 2 seconds. Then a goal of the scheduling mechanism should be to maximize the number of users who experience an average response time of 2 seconds or less.

Other criteria are system oriented. That is, the focus is on effective and efficient utilization of the processor. An example is throughput, which is the rate at which processes are completed. This is certainly a worthwhile measure of system performance and one that we would like to maximize. However, it focuses on system performance rather than service provided to the user. Thus, throughput is of concern to a system administrator but not to the user population.

Whereas user-oriented criteria are important on virtually all systems, system-oriented criteria are generally of minor importance on single-user systems. On a single-user system, it probably is not important to achieve high processor utilization or high throughput as long as the responsiveness of the system to user applications is acceptable.

Another dimension along which criteria can be classified is those that are performance related and those that are not directly performance related. Performance-related criteria are quantitative and generally can be readily measured. Examples include response time and throughput. Criteria that are not performance related are either qualitative in nature or do not lend themselves readily to measurement and analysis. An example of such a criterion is predictability. We would like for the service provided to users to exhibit the same characteristics over time, independent of other work being performed by the system. To some extent, this criterion can be measured, by calculating variances as a function of workload. However, this is not nearly as straightforward as measuring throughput or response time as a function of workload.

Table 9.2 summarizes key scheduling criteria. These are interdependent, and it is impossible to optimize all of them simultaneously. For example, providing good response time may require a scheduling algorithm that switches between processes frequently. This increases the overhead of the system, reducing throughput. Thus, the design of a scheduling policy involves compromising among competing requirements; the relative weights given the various requirements will depend on the nature and intended use of the system.

Table 9.2 Scheduling Criteria

User Oriented, Performance Related
Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.
Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.
Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.
User Oriented, Other
Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.
System Oriented, Performance Related
Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.
Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.
System Oriented, Other
Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.
Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.
Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

In most interactive operating systems, whether single user or time shared, adequate response time is the critical requirement. Because of the importance of this requirement, and because the definition of adequacy will vary from one application to another, the topic is explored further in Appendix 9A.

The Use of Priorities

In many systems, each process is assigned a priority and the scheduler will always choose a process of higher priority over one of lower priority. Figure 9.4 illustrates the use of priorities. For clarity, the queuing diagram is simplified, ignoring the existence of multiple blocked queues and of suspended states (compare Figure 3.8a). Instead of a single ready queue, we provide a set of queues, in descending order of priority: RQ0,

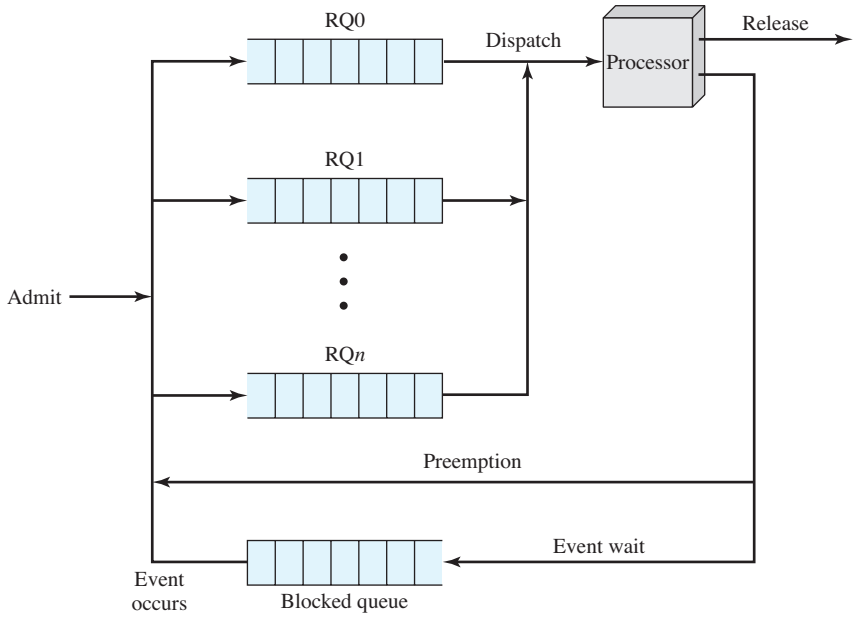


Figure 9.4 **Priority Queuing**

RQ1, . . . RQn, with priority[RQi] > priority[RQj] for $i < j$.[3] When a scheduling selection is to be made, the scheduler will start at the highest-priority ready queue (RQ0). If there are one or more processes in the queue, a process is selected using some scheduling policy. If RQ0 is empty, then RQ1 is examined, and so on.

One problem with a pure priority scheduling scheme is that lower-priority processes may suffer starvation. This will happen if there is always a steady supply of higher-priority ready processes. If this behavior is not desirable, the priority of a process can change with its age or execution history. We will give one example of this subsequently.

Alternative Scheduling Policies

Animation: Process Scheduling Algorithms

Table 9.3 presents some summary information about the various scheduling policies that are examined in this subsection. The **selection function** determines which process, among ready processes, is selected next for execution. The function may be based on priority, resource requirements, or the execution characteristics of the process. In the latter case, three quantities are significant:

w = time spent in system so far, waiting

e = time spent in execution so far

s = total service time required by the process, including e; generally, this quantity must be estimated or supplied by the user

[3] In UNIX and many other systems, larger priority values represent lower priority processes; unless otherwise stated we follow that convention. Some systems, such as Windows, use the opposite convention: a higher number means a higher priority.

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s − e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

For example, the selection function max[w] indicates a first-come-first-served (FCFS) discipline.

The **decision mode** specifies the instants in time at which the selection function is exercised. There are two general categories:

- **Nonpreemptive:** In this case, once a process is in the Running state, it continues to execute until (a) it terminates or (b) it blocks itself to wait for I/O or to request some operating system service.
- **Preemptive:** The currently running process may be interrupted and moved to the Ready state by the operating system. The decision to preempt may be performed when a new process arrives; when an interrupt occurs that places a blocked process in the Ready state; or periodically, based on a clock interrupt.

Preemptive policies incur greater overhead than nonpreemptive ones but may provide better service to the total population of processes, because they prevent any one process from monopolizing the processor for very long. In addition, the cost of preemption may be kept relatively low by using efficient process-switching mechanisms (as much help from hardware as possible) and by providing a large main memory to keep a high percentage of programs in main memory.

Table 9.4 Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

As we describe the various scheduling policies, we will use the set of processes in Table 9.4 as a running example. We can think of these as batch jobs, with the service time being the total execution time required. Alternatively, we can consider these to be ongoing processes that require alternate use of the processor and I/O in a repetitive fashion. In this latter case, the service times represent the processor time required in one cycle. In either case, in terms of a queuing model, this quantity corresponds to the service time.[4]

For the example of Table 9.4, Figure 9.5 shows the execution pattern for each policy for one cycle, and Table 9.5 summarizes some key results. First, the finish time of each process is determined. From this, we can determine the turnaround time. In terms of the queuing model, **turnaround time** (TAT) is the residence time T_r, or total time that the item spends in the system (waiting time plus service time). A more useful figure is the normalized turnaround time, which is the ratio of turnaround time to service time. This value indicates the relative delay experienced by a process. Typically, the longer the process execution time, the greater the absolute amount of delay that can be tolerated. The minimum possible value for this ratio is 1.0; increasing values correspond to a decreasing level of service.

First-Come-First-Served The simplest scheduling policy is first-come-first-served (FCFS), also known as first-in-first-out (FIFO) or a strict queuing scheme. As each process becomes ready, it joins the ready queue. When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running.

FCFS performs much better for long processes than short ones. Consider the following example, based on one in [FINK88]:

Process	Arrival Time	Service Time (T_s)	Start Time	Finish Time	Turnaround Time (T_r)	T_r/T_s
W	0	1	0	1	1	1
X	1	100	1	101	100	1
Y	2	1	101	102	100	100
Z	3	100	102	202	199	1.99
Mean					100	26

[4]See Appendix 9B for a summary of queuing model terminology.

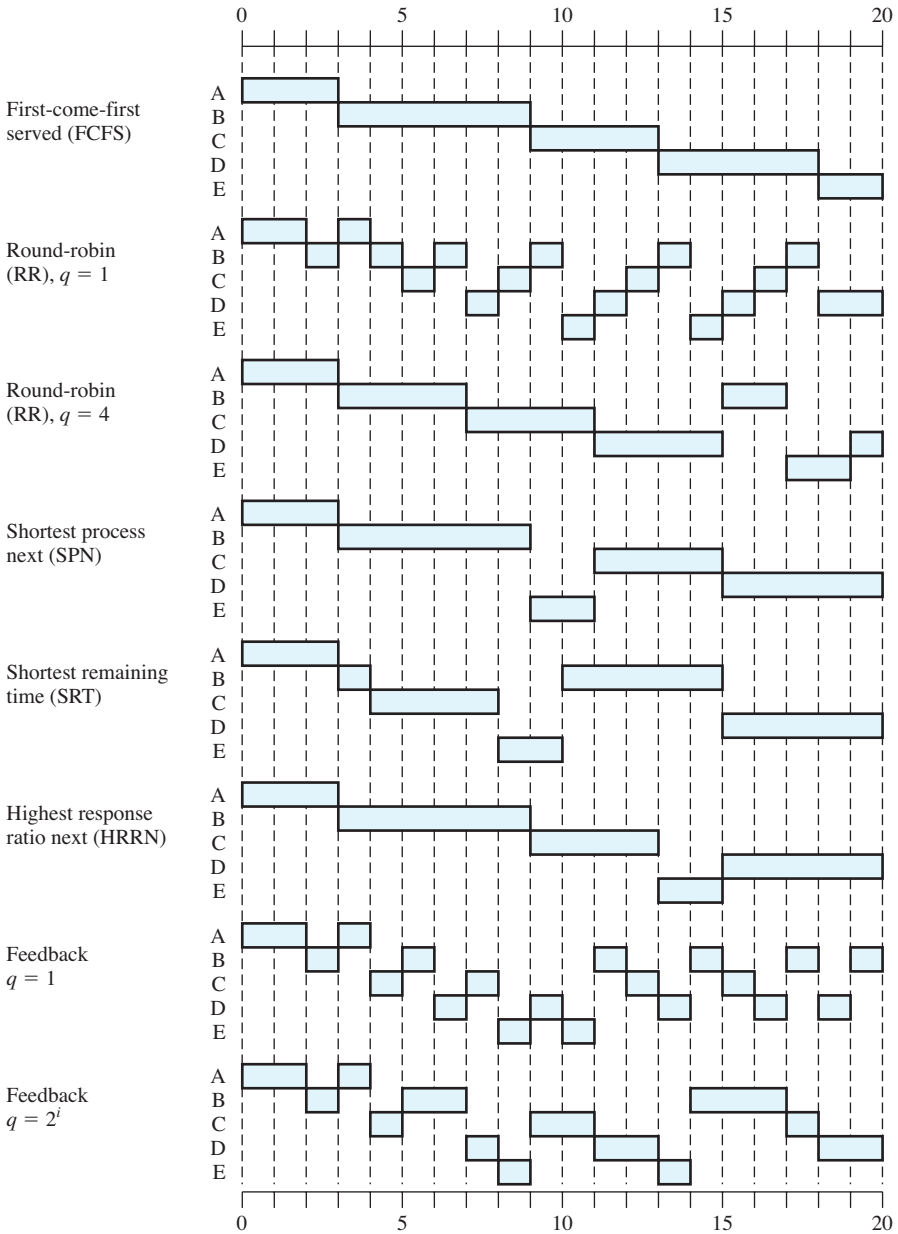


Figure 9.5 **A Comparison of Scheduling Policies**

The normalized turnaround time for process Y is way out of line compared to the other processes: the total time that it is in the system is 100 times the required processing time. This will happen whenever a short process arrives just after a long process. On the other hand, even in this extreme example, long processes do not fare poorly. Process Z has a turnaround time that is almost double that of Y, but its normalized residence time is under 2.0.

Table 9.5 A Comparison of Scheduling Policies

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_s)	3	6	4	5	2	Mean
FCFS						
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56
RR $q = 1$						
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_s	1.33	2.67	3.25	2.80	3.50	2.71
RR $q = 4$						
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71
SPN						
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84
SRT						
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_s	1.00	2.17	1.00	2.80	1.00	1.59
HRRN						
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_s	1.00	1.17	2.25	2.80	3.5	2.14
FB $q = 1$						
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_s	1.33	3.00	3.00	2.60	1.5	2.29
FB $q = 2^i$						
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_s	1.33	2.50	3.50	2.80	3.00	2.63

Another difficulty with FCFS is that it tends to favor processor-bound processes over I/O-bound processes. Consider that there is a collection of processes, one of which mostly uses the processor (processor bound) and a number of which favor I/O (I/O bound). When a processor-bound process is running, all of the I/O bound processes must wait. Some of these may be in I/O queues (blocked state) but

may move back to the ready queue while the processor-bound process is executing. At this point, most or all of the I/O devices may be idle, even though there is potentially work for them to do. When the currently running process leaves the Running state, the ready I/O-bound processes quickly move through the Running state and become blocked on I/O events. If the processor-bound process is also blocked, the processor becomes idle. Thus, FCFS may result in inefficient use of both the processor and the I/O devices.

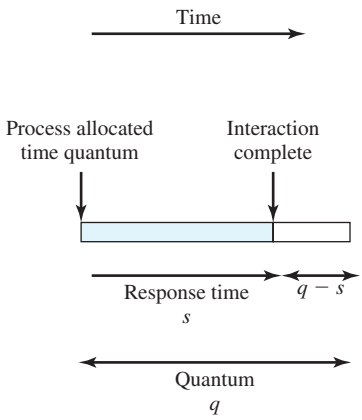
FCFS is not an attractive alternative on its own for a uniprocessor system. However, it is often combined with a priority scheme to provide an effective scheduler. Thus, the scheduler may maintain a number of queues, one for each priority level, and dispatch within each queue on a first-come-first-served basis. We see one example of such a system later, in our discussion of feedback scheduling.

Round Robin A straightforward way to reduce the penalty that short jobs suffer with FCFS is to use preemption based on a clock. The simplest such policy is round robin. A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis. This technique is also known as **time slicing**, because each process is given a slice of time before being preempted.

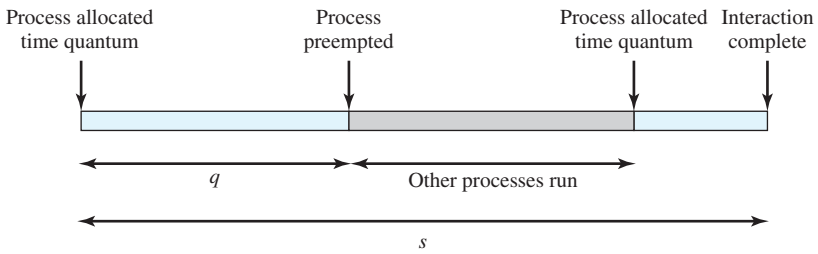
With round robin, the principal design issue is the length of the time quantum, or slice, to be used. If the quantum is very short, then short processes will move through the system relatively quickly. On the other hand, there is processing overhead involved in handling the clock interrupt and performing the scheduling and dispatching function. Thus, very short time quanta should be avoided. One useful guide is that the time quantum should be slightly greater than the time required for a typical interaction or process function. If it is less, then most processes will require at least two time quanta. Figure 9.6 illustrates the effect this has on response time. Note that in the limiting case of a time quantum that is longer than the longest-running process, round robin degenerates to FCFS.

Figure 9.5 and Table 9.5 show the results for our example using time quanta q of 1 and 4 time units. Note that process E, which is the shortest job, enjoys significant improvement for a time quantum of 1.

Round robin is particularly effective in a general-purpose time-sharing system or transaction processing system. One drawback to round robin is its relative treatment of processor-bound and I/O-bound processes. Generally, an I/O-bound process has a shorter processor burst (amount of time spent executing between I/O operations) than a processor-bound process. If there is a mix of processor-bound and I/O-bound processes, then the following will happen: An I/O-bound process uses a processor for a short period and then is blocked for I/O; it waits for the I/O operation to complete and then joins the ready queue. On the other hand, a processor-bound process generally uses a complete time quantum while executing and immediately returns to the ready queue. Thus, processor-bound processes tend to receive an unfair portion of processor time, which results in poor performance for I/O-bound processes, inefficient use of I/O devices, and an increase in the variance of response time.



(a) Time quantum greater than typical interaction



(b) Time quantum less than typical interaction

Figure 9.6 Effect of Size of Preemption Time Quantum

[HALD91] suggests a refinement to round robin that he refers to as a virtual round robin (VRR) and that avoids this unfairness. Figure 9.7 illustrates the scheme. New processes arrive and join the ready queue, which is managed on an FCFS basis. When a running process times out, it is returned to the ready queue. When a process is blocked for I/O, it joins an I/O queue. So far, this is as usual. The new feature is an FCFS auxiliary queue to which processes are moved after being released from an I/O block. When a dispatching decision is to be made, processes in the auxiliary queue get preference over those in the main ready queue. When a process is dispatched from the auxiliary queue, it runs no longer than a time equal to the basic time quantum minus the total time spent running since it was last selected from the main ready queue. Performance studies by the authors indicate that this approach is indeed superior to round robin in terms of fairness.

Shortest Process Next Another approach to reducing the bias in favor of long processes inherent in FCFS is the Shortest Process Next (SPN) policy. This is a non-

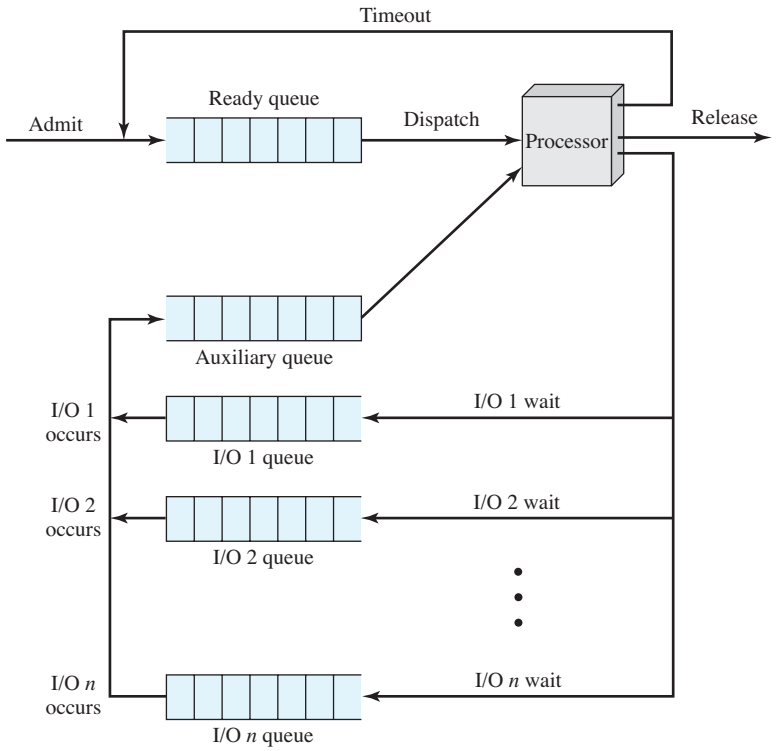


Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

preemptive policy in which the process with the shortest expected processing time is selected next. Thus a short process will jump to the head of the queue past longer jobs.

Figure 9.5 and Table 9.5 show the results for our example. Note that process E receives service much earlier than under FCFS. Overall performance is also significantly improved in terms of response time. However, the variability of response times is increased, especially for longer processes, and thus predictability is reduced.

One difficulty with the SPN policy is the need to know or at least estimate the required processing time of each process. For batch jobs, the system may require the programmer to estimate the value and supply it to the operating system. If the programmer's estimate is substantially under the actual running time, the system may abort the job. In a production environment, the same jobs run frequently, and statistics may be gathered. For interactive processes, the operating system may keep a running average of each "burst" for each process. The simplest calculation would be the following:

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i \qquad \textbf{(9.1)}$$

where

T_i = processor execution time for the ith instance of this process (total execution time for batch job; processor burst time for interactive job)

S_i = predicted value for the ith instance

S_1 = predicted value for first instance; not calculated

To avoid recalculating the entire summation each time, we can rewrite Equation (9.1) as

$$S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n \quad \textbf{(9.2)}$$

Note that this formulation gives equal weight to each instance. Typically, we would like to give greater weight to more recent instances, because these are more likely to reflect future behavior. A common technique for predicting a future value on the basis of a time series of past values is **exponential averaging**:

$$S_{n+1} = \alpha T_n + (1-\alpha)S_n \quad \textbf{(9.3)}$$

where α is a constant weighting factor ($0 < \alpha < 1$) that determines the relative weight given to more recent observations relative to older observations. Compare with Equation (9.2). By using a constant value of α, independent of the number of past observations, we have a circumstance in which all past values are considered, but the more distant ones have less weight. To see this more clearly, consider the following expansion of Equation (9.3):

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1 \quad \textbf{(9.4)}$$

Because both α and $(1-\alpha)$ are less than 1, each successive term in the preceding equation is smaller. For example, for $\alpha = 0.8$, Equation (9.4) becomes

$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032\ T_{n-2} + 0.0064\ T_{n-3} + \ldots$$

The older the observation, the less it is counted in to the average.

The size of the coefficient as a function of its position in the expansion is shown in Figure 9.8. The larger the value of α, the greater the weight given to the more recent observations. For $\alpha = 0.8$, virtually all of the weight is given to the four most recent observations, whereas for $\alpha = 0.2$, the averaging is effectively spread out over the eight or so most recent observations. The advantage of using a value of α close to 1 is that the average will quickly reflect a rapid change in the observed quantity. The disadvantage is that if there is a brief surge in the value of the observed quantity and

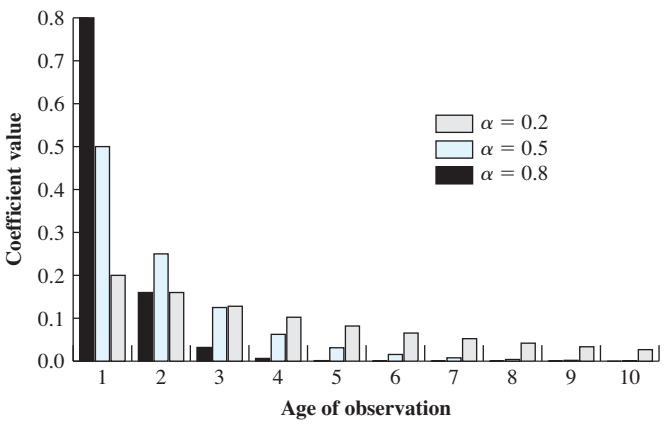
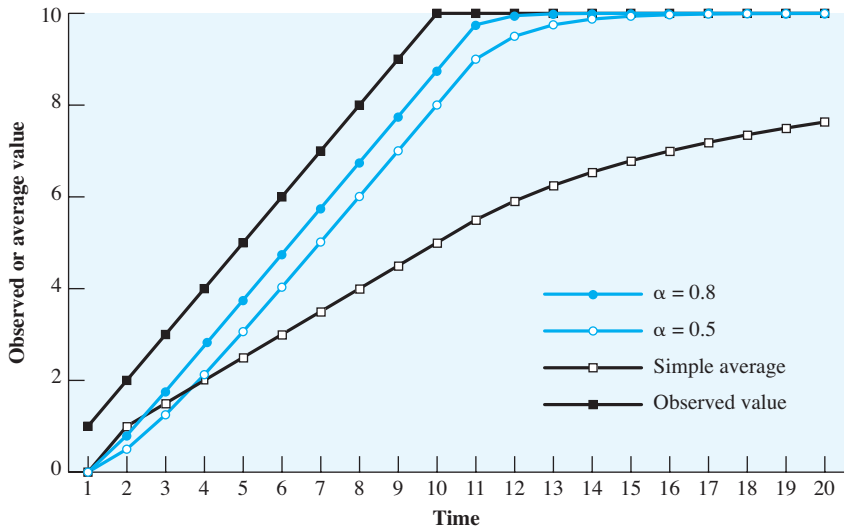


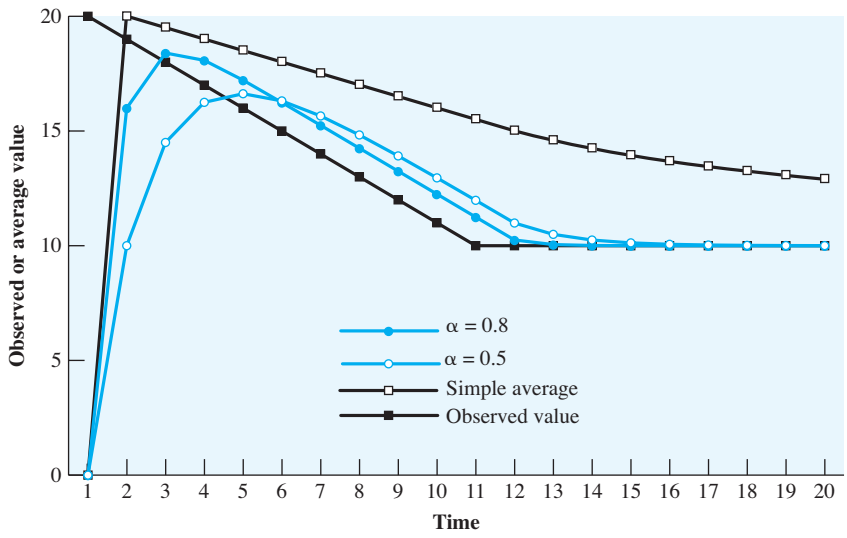
Figure 9.8 **Exponential Smoothing Coefficients**

it then settles back to some average value, the use of a large value of α will result in jerky changes in the average.

Figure 9.9 compares simple averaging with exponential averaging (for two different values of α). In Figure 9.9a, the observed value begins at 1, grows gradually to a value of 10, and then stays there. In Figure 9.9b, the observed value begins at 20, declines gradually to 10, and then stays there. In both cases, we start out with an estimate of $S_1 = 0$. This gives greater priority to new processes. Note that exponential averaging tracks changes in process behavior faster than does simple averaging



(a) Increasing function



(b) Decreasing function

Figure 9.9 **Use of Exponential Averaging**

and that the larger value of α results in a more rapid reaction to the change in the observed value.

A risk with SPN is the possibility of starvation for longer processes, as long as there is a steady supply of shorter processes. On the other hand, although SPN reduces the bias in favor of longer jobs, it still is not desirable for a time-sharing or transaction processing environment because of the lack of preemption. Looking back at our worst-case analysis described under FCFS, processes W, X, Y, and Z will still execute in the same order, heavily penalizing the short process Y.

Shortest Remaining Time The shortest remaining time (SRT) policy is a preemptive version of SPN. In this case, the scheduler always chooses the process that has the shortest expected remaining processing time. When a new process joins the ready queue, it may in fact have a shorter remaining time than the currently running process. Accordingly, the scheduler may preempt the current process when a new process becomes ready. As with SPN, the scheduler must have an estimate of processing time to perform the selection function, and there is a risk of starvation of longer processes.

SRT does not have the bias in favor of long processes found in FCFS. Unlike round robin, no additional interrupts are generated, reducing overhead. On the other hand, elapsed service times must be recorded, contributing to overhead. SRT should also give superior turnaround time performance to SPN, because a short job is given immediate preference to a running longer job.

Note that in our example (Table 9.5), the three shortest processes all receive immediate service, yielding a normalized turnaround time for each of 1.0.

Highest Response Ratio Next In Table 9.5, we have used the normalized turnaround time, which is the ratio of turnaround time to actual service time, as a figure of merit. For each individual process, we would like to minimize this ratio, and we would like to minimize the average value over all processes. In general, we cannot know ahead of time what the service time is going to be, but we can approximate it, either based on past history or some input from the user or a configuration manager. Consider the following ratio:

$$R = \frac{w + s}{s}$$

where

R = response ratio
w = time spent waiting for the processor
s = expected service time

If the process with this value is dispatched immediately, R is equal to the normalized turnaround time. Note that the minimum value of R is 1.0, which occurs when a process first enters the system.

Thus, our scheduling rule becomes the following: When the current process completes or is blocked, choose the ready process with the greatest value of R. This approach is attractive because it accounts for the age of the process. While shorter jobs are favored (a smaller denominator yields a larger ratio), aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs.

As with SRT and SPN, the expected service time must be estimated to use highest response ratio next (HRRN).

Feedback If we have no indication of the relative length of various processes, then none of SPN, SRT, and HRRN can be used. Another way of establishing a preference for shorter jobs is to penalize jobs that have been running longer. In other words, if we cannot focus on the time remaining to execute, let us focus on the time spent in execution so far.

The way to do this is as follows. Scheduling is done on a preemptive (at time quantum) basis, and a dynamic priority mechanism is used. When a process first enters the system, it is placed in RQ0 (see Figure 9.4). After its first preemption, when it returns to the Ready state, it is placed in RQ1. Each subsequent time that it is preempted, it is demoted to the next lower-priority queue. A short process will complete quickly, without migrating very far down the hierarchy of ready queues. A longer process will gradually drift downward. Thus, newer, shorter processes are favored over older, longer processes. Within each queue, except the lowest-priority queue, a simple FCFS mechanism is used. Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution. Thus, this queue is treated in round-robin fashion.

Figure 9.10 illustrates the feedback scheduling mechanism by showing the path that a process will follow through the various queues.[5] This approach is known

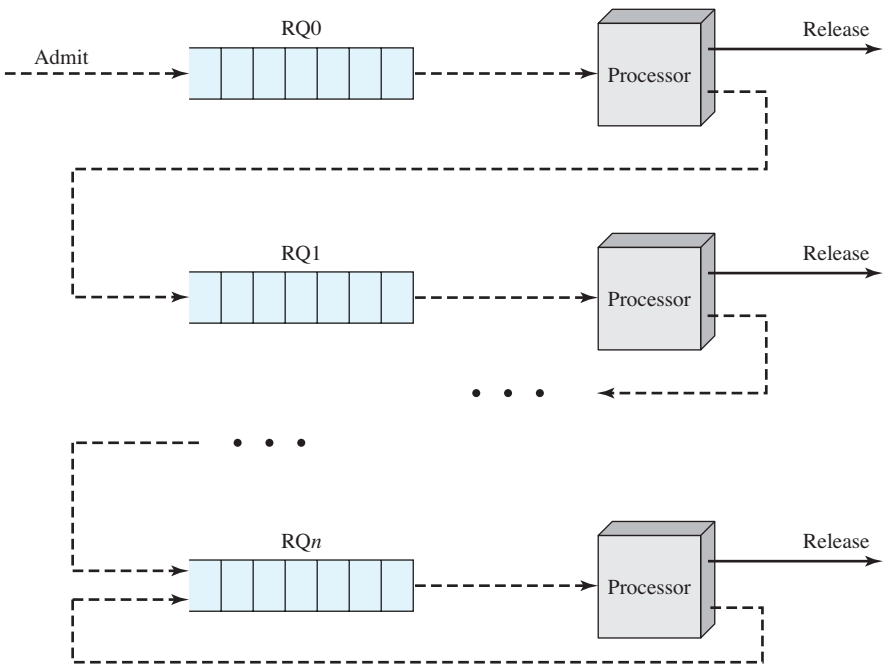


Figure 9.10 Feedback Scheduling

[5]Dotted lines are used to emphasize that this is a time sequence diagram rather than a static depiction of possible transitions, such as Figure 9.4.

as **multilevel feedback**, meaning that the operating system allocates the processor to a process and, when the process blocks or is preempted, feeds it back into one of several priority queues.

There are a number of variations on this scheme. A simple version is to perform preemption in the same fashion as for round robin: at periodic intervals. Our example shows this (Figure 9.5 and Table 9.5) for a quantum of one time unit. Note that in this case, the behavior is similar to round robin with a time quantum of 1.

One problem with the simple scheme just outlined is that the turnaround time of longer processes can stretch out alarmingly. Indeed, it is possible for starvation to occur if new jobs are entering the system frequently. To compensate for this, we can vary the preemption times according to the queue: A process scheduled from RQ0 is allowed to execute for one time unit and then is preempted; a process scheduled from RQ1 is allowed to execute two time units, and so on. In general, a process scheduled from RQi is allowed to execute $2i$ time units before preemption. This scheme is illustrated for our example in Figure 9.5 and Table 9.5.

Even with the allowance for greater time allocation at lower priority, a longer process may still suffer starvation. A possible remedy is to promote a process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue.

Performance Comparison

Clearly, the performance of various scheduling policies is a critical factor in the choice of a scheduling policy. However, it is impossible to make definitive comparisons because relative performance will depend on a variety of factors, including the probability distribution of service times of the various processes, the efficiency of the scheduling and context switching mechanisms, and the nature of the I/O demand and the performance of the I/O subsystem. Nevertheless, we attempt in what follows to draw some general conclusions.

Queuing Analysis In this section, we make use of basic queuing formulas, with the common assumptions of Poisson arrivals and exponential service times.[6]

First, we make the observation that any such scheduling discipline that chooses the next item to be served independent of service time obeys the following relationship:

$$\frac{T_r}{T_s} = \frac{1}{1 - \rho}$$

where

T_r = turnaround time or residence time; total time in system, waiting plus execution

T_s = average service time; average time spent in Running state

ρ = processor utilization

[6]The queuing terminology used in this chapter is summarized in Appendix 9B. Poisson arrivals essentially means random arrivals, as explained in Appendix 9B.

In particular, a priority-based scheduler, in which the priority of each process is assigned independent of expected service time, provides the same average turnaround time and average normalized turnaround time as a simple FCFS discipline. Furthermore, the presence or absence of preemption makes no differences in these averages.

With the exception of round robin and FCFS, the various scheduling disciplines considered so far do make selections on the basis of expected service time. Unfortunately, it turns out to be quite difficult to develop closed analytic models of these disciplines. However, we can get an idea of the relative performance of such scheduling algorithms, compared to FCFS, by considering priority scheduling in which priority is based on service time.

If scheduling is done on the basis of priority and if processes are assigned to a priority class on the basis of service time, then differences do emerge. Table 9.6 shows the formulas that result when we assume two priority classes, with different service times for each class. In the table, λ refers to the arrival rate. These results can be generalized to any number of priority classes. Note that the formulas differ for nonpreemptive versus preemptive scheduling. In the latter case, it is assumed that a lower-priority process is immediately interrupted when a higher-priority process becomes ready.

As an example, let us consider the case of two priority classes, with an equal number of process arrivals in each class and with the average service time for the lower-priority class being 5 times that of the upper priority class. Thus, we wish to

Table 9.6 Formulas for Single-Server Queues with Two Priority Categories

Assumptions:
1. Poisson arrival rate.
2. Priority 1 items are serviced before priority 2 items.
3. First-come-first-served dispatching for items of equal priority.
4. No item is interrupted while being served.
5. No items leave the queue (lost calls delayed).

(a) General formulas

$$\lambda = \lambda_1 + \lambda_2$$

$$\rho_1 = \lambda_1 T_{s1}; \rho_2 = \lambda_2 T_{s2}$$

$$\rho = \rho_1 + \rho_2$$

$$T_s = \frac{\lambda_1}{\lambda} T_{s1} + \frac{\lambda_2}{\lambda} T_{s2}$$

$$T_r = \frac{\lambda_1}{\lambda} T_{r1} + \frac{\lambda_2}{\lambda} T_{r2}$$

(b) No interrupts; exponential service times

$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1} + \rho_2 T_{s2}}{1 - \rho_1}$$

$$T_{r2} = T_{s2} + \frac{T_{r1} - T_{s1}}{1 - \rho}$$

(c) Preemptive-resume queuing discipline; exponential service times

$$T_{r1} = T_{s1} + \frac{\rho_1 T_{s1}}{1 - \rho_1}$$

$$T_{r2} = T_{s2} + \frac{1}{1 - \rho_1}\left(\rho_1 T_{s2} + \frac{\rho T_s}{1 - \rho}\right)$$

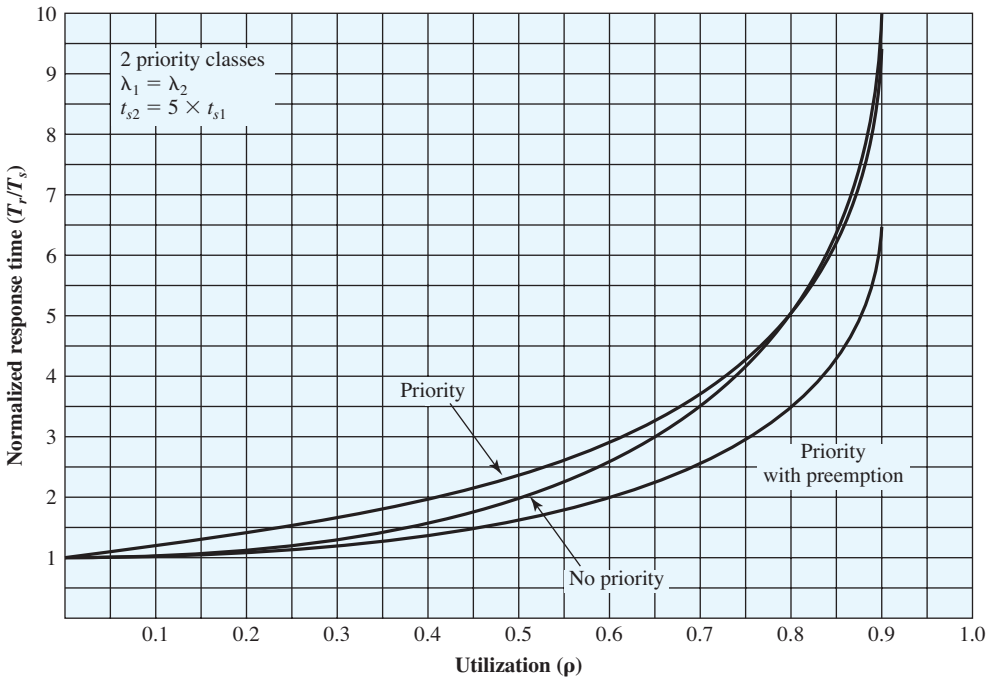


Figure 9.11 **Overall Normalized Response Time**

give preference to shorter processes. Figure 9.11 shows the overall result. By giving preference to shorter jobs, the average normalized turnaround time is improved at higher levels of utilization. As might be expected, the improvement is greatest with the use of preemption. Notice, however, that overall performance is not much affected.

However, significant differences emerge when we consider the two priority classes separately. Figure 9.12 shows the results for the higher-priority, shorter processes. For comparison, the upper line on the graph assumes that priorities are not used but that we are simply looking at the relative performance of that half of all processes that have the shorter processing time. The other two lines assume that these processes are assigned a higher priority. When the system is run using priority scheduling without preemption, the improvements are significant. They are even more significant when preemption is used.

Figure 9.13 shows the same analysis for the lower-priority, longer processes. As expected, such processes suffer a performance degradation under priority scheduling.

Simulation Modeling Some of the difficulties of analytic modeling are overcome by using discrete-event simulation, which allows a wide range of policies to be modeled. The disadvantage of simulation is that the results for a given "run" only apply to that particular collection of processes under that particular set of assumptions. Nevertheless, useful insights can be gained.

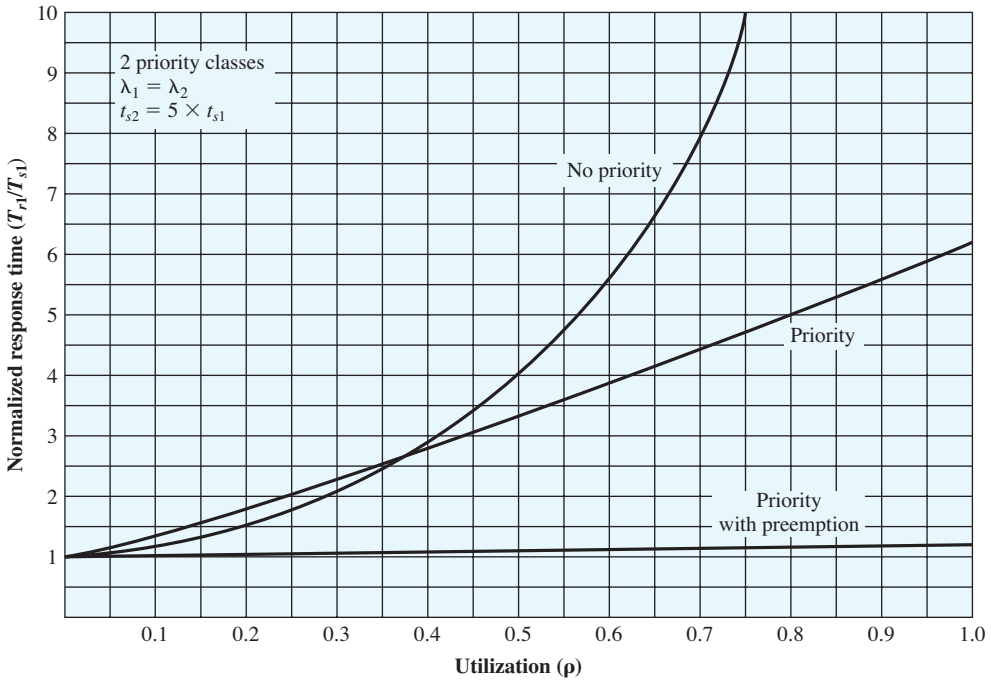


Figure 9.12 **Normalized Response Time for Shorter Processes**

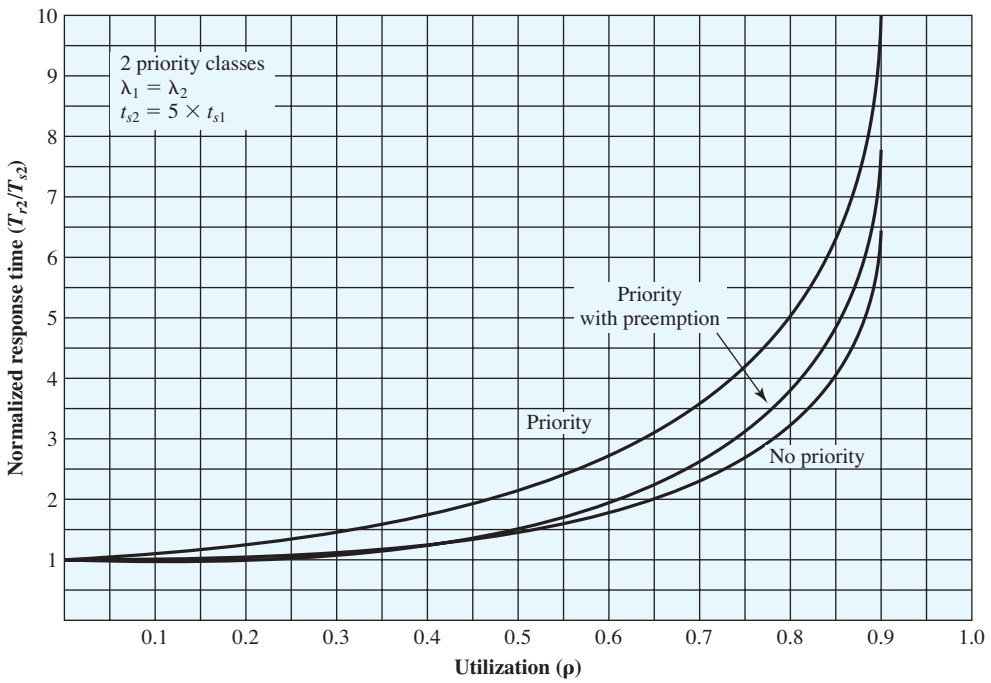


Figure 9.13 **Normalized Response Time for Longer Processes**

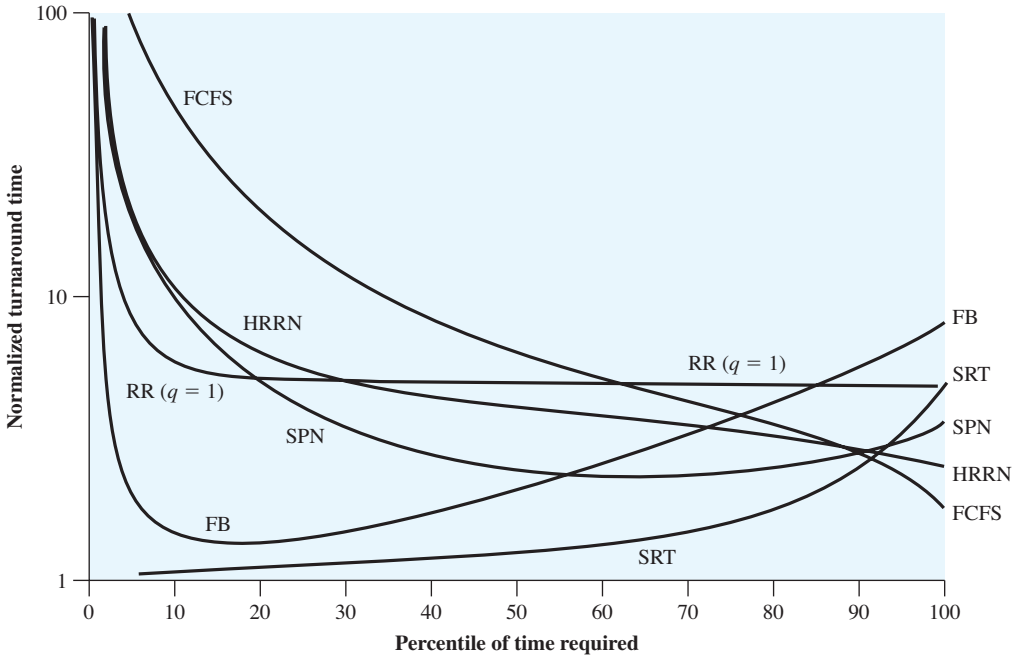


Figure 9.14 **Simulation Result for Normalized Turnaround Time**

The results of one such study are reported in [FINK88]. The simulation involved 50,000 processes with an arrival rate of $\lambda = 0.8$ and an average service time of $T_s = 1$. Thus, the assumption is that the processor utilization is $\rho = \lambda Ts = 0.8$. Note, therefore, that we are only measuring one utilization point.

To present the results, processes are grouped into service-time percentiles, each of which has 500 processes. Thus, the 500 processes with the shortest service time are in the first percentile; with these eliminated, the 500 remaining processes with the shortest service time are in the second percentile; and so on. This allows us to view the effect of various policies on processes as a function of the length of the process.

Figure 9.14 shows the normalized turnaround time, and Figure 9.15 shows the average waiting time. Looking at the turnaround time, we can see that the performance of FCFS is very unfavorable, with one-third of the processes having a normalized turnaround time greater than 10 times the service time; furthermore, these are the shortest processes. On the other hand, the absolute waiting time is uniform, as is to be expected because scheduling is independent of service time. The figures show round robin using a quantum of one time unit. Except for the shortest processes, which execute in less than one quantum, round robin yields a normalized turnaround time of about 5 for all processes, treating all fairly. Shortest process next performs better than round robin, except for the shortest processes. Shortest remaining time, the preemptive version of SPN, performs better than SPN except for the longest 7% of all processes. We have seen that, among nonpreemptive policies, FCFS favors long processes and SPN favors short ones. Highest response ratio next is intended to be a

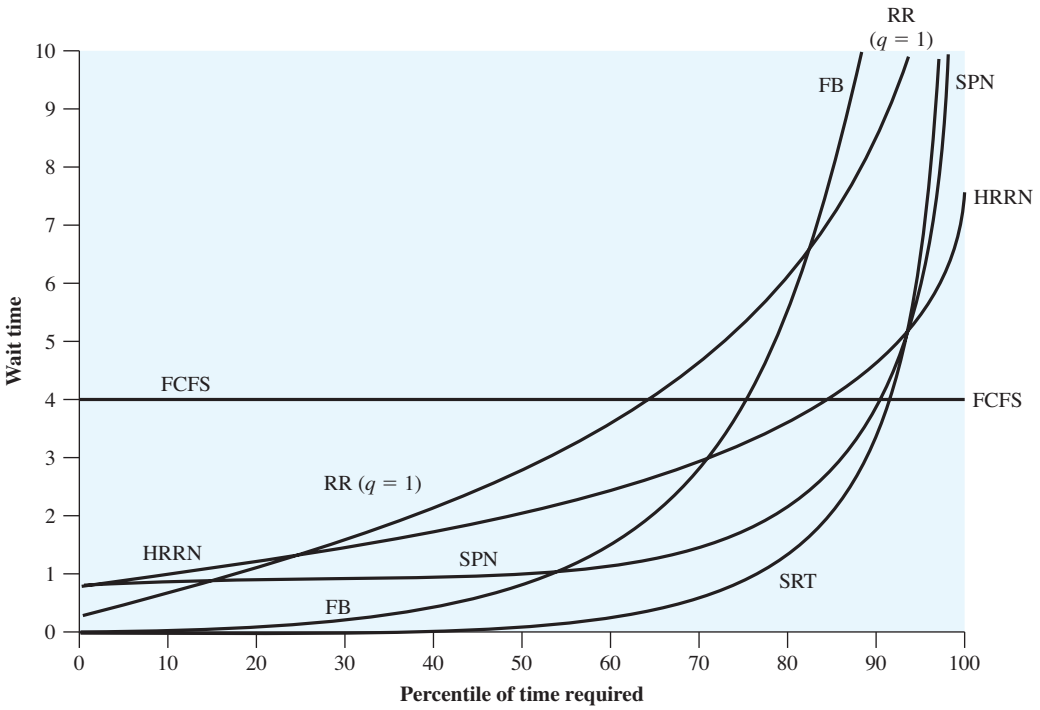


Figure 9.15 **Simulation Result for Waiting Time**

compromise between these two effects, and this is indeed confirmed in the figures. Finally, the figure shows feedback scheduling with fixed, uniform quanta in each priority queue. As expected, FB performs quite well for short processes.

Fair-Share Scheduling

All of the scheduling algorithms discussed so far treat the collection of ready processes as a single pool of processes from which to select the next running process. This pool may be broken down by priority but is otherwise homogeneous.

However, in a multiuser system, if individual user applications or jobs may be organized as multiple processes (or threads), then there is a structure to the collection of processes that is not recognized by a traditional scheduler. From the user's point of view, the concern is not how a particular process performs but rather how his or her set of processes, which constitute a single application, performs. Thus, it would be attractive to make scheduling decisions on the basis of these process sets. This approach is generally known as fair-share scheduling. Further, the concept can be extended to groups of users, even if each user is represented by a single process. For example, in a time-sharing system, we might wish to consider all of the users from a given department to be members of the same group. Scheduling decisions could then be made that attempt to give each group similar service. Thus, if a large number of people from one department log onto the system, we would like to see response time degradation primarily affect members of that department rather than users from other departments.

The term *fair share* indicates the philosophy behind such a scheduler. Each user is assigned a weighting of some sort that defines that user's share of system resources as a fraction of the total usage of those resources. In particular, each user is assigned a share of the processor. Such a scheme should operate in a more or less linear fashion, so that if user A has twice the weighting of user B, then in the long run, user A should be able to do twice as much work as user B. The objective of a fair-share scheduler is to monitor usage to give fewer resources to users who have had more than their fair share and more to those who have had less than their fair share.

A number of proposals have been made for fair-share schedulers [HENR84, KAY88, WOOD86]. In this section, we describe the scheme proposed in [HENR84] and implemented on a number of UNIX systems. The scheme is simply referred to as the fair-share scheduler (FSS). FSS considers the execution history of a related group of processes, along with the individual execution history of each process in making scheduling decisions. The system divides the user community into a set of fair-share groups and allocates a fraction of the processor resource to each group. Thus, there might be four groups, each with 25% of the processor usage. In effect, each fair-share group is provided with a virtual system that runs proportionally slower than a full system.

Scheduling is done on the basis of priority, which takes into account the underlying priority of the process, its recent processor usage, and the recent processor usage of the group to which the process belongs. The higher the numerical value of the priority, the lower the priority. The following formulas apply for process j in group k:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$GCPU_k(i)$ = measure of processor utilization of group k through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

W_k = weighting assigned to group k, with the constraint that $0 < W_k \leq 1$ and $\sum_k W_k = 1$

Each process is assigned a base priority. The priority of a process drops as the process uses the processor and as the group to which the process belongs uses the processor. In the case of the group utilization, the average is normalized by dividing by the weight of that group. The greater the weight assigned to the group, the less its utilization will affect its priority.

Figure 9.16 Example of Fair-Share Scheduler—Three Processes, Two Groups

Figure 9.16 is an example in which process A is in one group and process B and process C are in a second group, with each group having a weighting of 0.5. Assume that all processes are processor bound and are usually ready to run. All processes have a base priority of 60. Processor utilization is measured as follows: The processor is interrupted 60 times per second; during each interrupt, the processor usage field of the currently running process is incremented, as is the corresponding group processor field. Once per second, priorities are recalculated.

In the figure, process A is scheduled first. At the end of one second, it is preempted. Processes B and C now have the higher priority, and process B is scheduled.

At the end of the second time unit, process A has the highest priority. Note that the pattern repeats: the kernel schedules the processes in order: A, B, A, C, A, B, and so on. Thus, 50% of the processor is allocated to process A, which constitutes one group, and 50% to processes B and C, which constitute another group.

9.3 TRADITIONAL UNIX SCHEDULING

In this section we examine traditional UNIX scheduling, which is used in both SVR3 and 4.3 BSD UNIX. These systems are primarily targeted at the time-sharing interactive environment. The scheduling algorithm is designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve. Although this algorithm has been replaced in modern UNIX systems, it is worthwhile to examine the approach because it is representative of practical time-sharing scheduling algorithms. The scheduling scheme for SVR4 includes an accommodation for real-time requirements, and so its discussion is deferred to Chapter 10.

The traditional UNIX scheduler employs multilevel feedback using round robin within each of the priority queues. The system makes use of 1-second preemption. That is, if a running process does not block or complete within 1 second, it is preempted. Priority is based on process type and execution history. The following formulas apply:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor

The priority of each process is recomputed once per second, at which time a new scheduling decision is made. The purpose of the base priority is to divide all processes into fixed bands of priority levels. The *CPU* and *nice* components are restricted to prevent a process from migrating out of its assigned band (assigned by the base priority level). These bands are used to optimize access to block devices (e.g., disk) and to allow the operating system to respond quickly to system calls. In decreasing order of priority, the bands are

- Swapper
- Block I/O device control
- File manipulation
- Character I/O device control
- User processes

This hierarchy should provide the most efficient use of the I/O devices. Within the user process band, the use of execution history tends to penalize processor-bound processes at the expense of I/O-bound processes. Again, this should improve efficiency. Coupled with the round-robin preemption scheme, the scheduling strategy is well equipped to satisfy the requirements for general-purpose time sharing.

An example of process scheduling is shown in Figure 9.17. Processes A, B, and C are created at the same time with base priorities of 60 (we will ignore the *nice* value). The clock interrupts the system 60 times per second and increments a counter for the running process. The example assumes that none of the processes

Time	Process A		Process B		Process C	
	Priority	CPU count	Priority	CPU count	Priority	CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Colored rectangle represents executing process

Figure 9.17 Example of a Traditional UNIX Process Scheduling

block themselves and that no other processes are ready to run. Compare this with Figure 9.16.

9.4 SUMMARY

The operating system must make three types of scheduling decisions with respect to the execution of processes. Long-term scheduling determines when new processes are admitted to the system. Medium-term scheduling is part of the swapping function and determines when a program is brought partially or fully into main memory so that it may be executed. Short-term scheduling determines which ready process will be executed next by the processor. This chapter focuses on the issues relating to short-term scheduling.

A variety of criteria are used in designing the short-term scheduler. Some of these criteria relate to the behavior of the system as perceived by the individual user (user oriented), while others view the total effectiveness of the system in meeting the needs of all users (system oriented). Some of the criteria relate specifically to quantitative measures of performance, while others are more qualitative in nature. From a user's point of view, response time is generally the most important characteristic of a system, while from a system point of view, throughput or processor utilization is important.

A variety of algorithms have been developed for making the short-term scheduling decision among all ready processes:

- **First-come-first-served:** Select the process that has been waiting the longest for service.
- **Round robin:** Use time slicing to limit any running process to a short burst of processor time, and rotate among all ready processes.
- **Shortest process next:** Select the process with the shortest expected processing time, and do not preempt the process.
- **Shortest remaining time:** Select the process with the shortest expected remaining process time. A process may be preempted when another process becomes ready.
- **Highest response ratio next:** Base the scheduling decision on an estimate of normalized turnaround time.
- **Feedback:** Establish a set of scheduling queues and allocate processes to queues based on execution history and other criteria.

The choice of scheduling algorithm will depend on expected performance and on implementation complexity.

9.5 RECOMMENDED READING

Virtually every textbook on operating systems covers scheduling. Rigorous queuing analyses of various scheduling policies are presented in [KLEI04] and [CONW67]. [DOWD93] provides an instructive performance analysis of various scheduling algorithms.

CONW67 Conway, R.; Maxwell, W.; and Miller, L. *Theory of Scheduling.* Reading, MA: Addison-Wesley, 1967. Reprinted by Dover Publications, 2003.

DOWD93 Dowdy, L., and Lowery, C. *P.S. to Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1993.

KLEI04 Kleinrock, L. *Queuing Systems, Volume Three: Computer Applications.* New York: Wiley, 2004.

9.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

arrival rate	long-term scheduler	service time
dispatcher	medium-term scheduler	short-term scheduler
exponential averaging	multilevel feedback	throughput
fair share scheduling	predictability	time slicing
fairness	residence time	turnaround time (TAT)
first-come-first-served (FCFS)	response time	utilization
first-in-first-out (FIFO)	round robin	waiting time
	scheduling priority	

Review Questions

9.1 Briefly describe the three types of processor scheduling.
9.2 What is usually the critical performance requirement in an interactive operating system?
9.3 What is the difference between turnaround time and response time?
9.4 For process scheduling, does a low-priority value represent a low priority or a high priority?
9.5 What is the difference between preemptive and nonpreemptive scheduling?
9.6 Briefly define FCFS scheduling.
9.7 Briefly define round-robin scheduling.
9.8 Briefly define shortest-process-next scheduling.
9.9 Briefly define shortest-remaining-time scheduling.
9.10 Briefly define highest-response-ratio-next scheduling.
9.11 Briefly define feedback scheduling.

Problems

9.1 Consider the following set of processes:

Process Name	Arrival Time	Processing Time
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Perform the same analysis as depicted in Table 9.5 and Figure 9.5 for this set.

9.2 Repeat Problem 9.1 for the following set:

Process Name	Arrival Time	Processing Time
A	0	1
B	1	9
C	2	1
D	3	9

9.3 Prove that, among nonpreemptive scheduling algorithms, SPN provides the minimum average waiting time for a batch of jobs that arrive at the same time. Assume that the scheduler must always execute a task if one is available.

9.4 Assume the following burst-time pattern for a process: 6, 4, 6, 4, 13, 13, 13, and assume that the initial guess is 10. Produce a plot similar to those of Figure 9.9.

9.5 Consider the following pair of equations as an alternative to Equation (9.3):

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n$$
$$X_{n+1} = \min[Ubound, \max[Lbound, (\beta S_{n+1})]]$$

where *Ubound* and *Lbound* are prechosen upper and lower bounds on the estimated value of T. The value of X_{n+1} is used in the shortest-process-next algorithm, instead of the value of S_{n+1}. What functions do α and β perform, and what is the effect of higher and lower values on each?

9.6 In the bottom example in Figure 9.5, process A runs for 2 time units before control is passed to process B. Another plausible scenario would be that A runs for 3 time units before control is passed to process B. What policy differences in the feedback scheduling algorithm would account for the two different scenarios?

9.7 In a nonpreemptive uniprocessor system, the ready queue contains three jobs at time t immediately after the completion of a job. These jobs arrived at times t_1, t_2, and t_3 with estimated execution times of r_1, r_2, and r_3, respectively. Figure 9.18 shows the linear increase of their response ratios over time. Use this example to find a variant of response ratio scheduling, known as minimax response ratio scheduling, that minimizes the maximum response ratio for a given batch of jobs ignoring further arrivals. (*Hint:* Decide first which job to schedule as the last one.)

9.8 Prove that the minimax response ratio algorithm of the preceding problem minimizes the maximum response ratio for a given batch of jobs. (*Hint:* Focus attention on the job that will achieve the highest response ratio and all jobs executed before it. Consider the same subset of jobs scheduled in any other order and observe the response ratio of the job that is executed as the last one among them. Notice that this subset may now be mixed with other jobs from the total set.)

9.9 Define residence time T_r as the average total time a process spends waiting and being served. Show that for FIFO, with mean service time T_s, we have $T_r = T_s/(1 - \rho)$, where ρ is utilization.

9.10 A processor is multiplexed at infinite speed among all processes present in a ready queue with no overhead. (This is an idealized model of round robin scheduling among

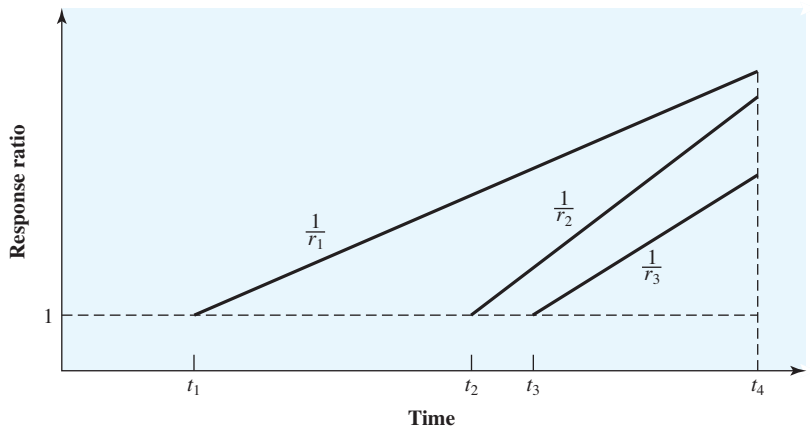


Figure 9.18 **Response Ratio as a Function of Time**

ready processes using time slices that are very small compared to the mean service time.) Show that for Poisson input from an infinite source with exponential service times, the mean response time R_x of a process with service time x is given by $R_x = x/(1 - \rho)$. (*Hint:* Review the basic queuing equations in the Queuing Analysis document at WilliamStallings.com/StudentSupport.html. Then consider the number of items waiting, w, in the system upon arrival of the given process.)

9.11 Most round-robin schedulers use a fixed size quantum. Give an argument in favor of a small quantum. Now give an argument in favor of a large quantum. Compare and contrast the types of systems and jobs to which the arguments apply. Are there any for which both are reasonable?

9.12 In a queuing system, new jobs must wait for a while before being served. While a job waits, its priority increases linearly with time from zero at a rate α. A job waits until its priority reaches the priority of the jobs in service; then it begins to share the processor equally with other jobs in service using round robin while its priority continues to increase at a slower rate β. The algorithm is referred to as selfish round robin, because the jobs in service try (in vain) to monopolize the processor by increasing their priority continuously. Use Figure 9.19 to show that the mean response time R_x for a job of service time x is given by

$$R_x = \frac{s}{1 - \rho} + \frac{x - s}{1 - \rho'}$$

where

$$\rho = \lambda s \quad \rho' = \rho\left(1 - \frac{\beta}{\alpha}\right) \quad 0 \leq \beta < \alpha$$

assuming that arrival and service times are exponentially distributed with means $1/\lambda$ and s, respectively. (*Hint:* Consider the total system and the two subsystems separately.)

9.13 An interactive system using round-robin scheduling and swapping tries to give guaranteed response to trivial requests as follows: After completing a round robin cycle among all ready processes, the system determines the time slice to allocate to each ready process for the next cycle by dividing a maximum response time by the number of processes requiring service. Is this a reasonable policy?

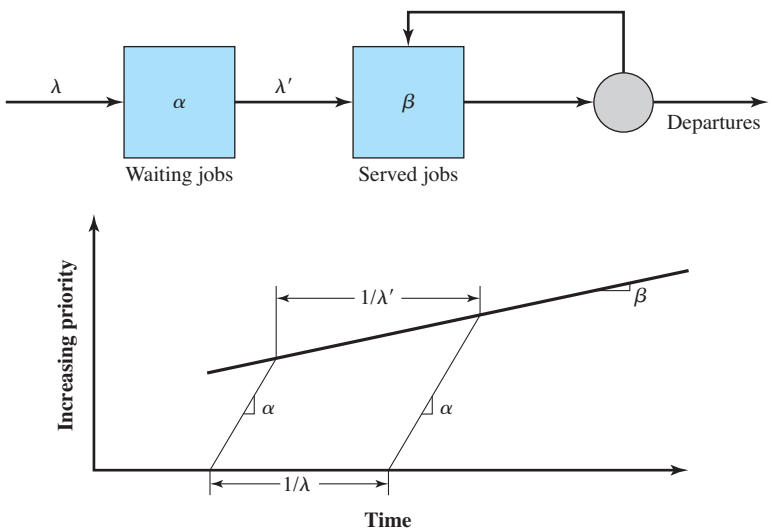


Figure 9.19 **Shelfish Round Robin**

9.14 Which type of process is generally favored by a multilevel feedback queuing scheduler—a processor-bound process or an I/O-bound process? Briefly explain why.

9.15 In priority-based process scheduling, the scheduler only gives control to a particular process if no other process of higher priority is currently in the ready state. Assume that no other information is used in making the process scheduling decision. Also assume that process priorities are established at process creation time and do not change. In a system operating with such assumptions, why would using Dekker's solution (see Section A.1) to the mutual exclusion problem be "dangerous"? Explain this by telling what undesired event could occur and how it could occur.

9.16 Five batch jobs, A through E, arrive at a computer center at essentially the same time. They have an estimated running time of 15, 9, 3, 6, and 12 minutes, respectively. Their (externally defined) priorities are 6, 3, 7, 9, and 4 respectively, with a lower value corresponding to a higher priority. For each of the following scheduling algorithms, determine the turnaround time for each process and the average turnaround for all jobs. Ignore process switching overhead. Explain how you arrived at your answers. In the last three cases, assume that only one job at a time runs until it finishes and that all jobs are completely processor bound.

a. round robin with a time quantum of 1 minute
b. priority scheduling
c. FCFS (run in order 15, 9, 3, 6, and 12)
d. shortest job first

APPENDIX 9A RESPONSE TIME

Response time is the time it takes a system to react to a given input. In an interactive transaction, it may be defined as the time between the last keystroke by the user and the beginning of the display of a result by the computer. For different types of applications, a slightly different definition is needed. In general, it is the time it takes for the system to respond to a request to perform a particular task.

Ideally, one would like the response time for any application to be short. However, it is almost invariably the case that shorter response time imposes greater cost. This cost comes from two sources:

- **Computer processing power:** The faster the processor, the shorter the response time. Of course, increased processing power means increased cost.
- **Competing requirements:** Providing rapid response time to some processes may penalize other processes.

Thus the value of a given level of response time must be assessed versus the cost of achieving that response time.

Table 9.7, based on [MART88], lists six general ranges of response times. Design difficulties are faced when a response time of less than 1 second is required. A requirement for a subsecond response time is generated by a system that controls or in some other way interacts with an ongoing external activity, such as an assembly line. Here the requirement is straightforward. When we consider human-computer interaction, such as in a data entry application, then we are in the realm of conversational response time. In this case, there is still a requirement for a short response time, but the acceptable length of time may be difficult to assess.

Table 9.7 Response Time Ranges

Greater than 15 seconds

This rules out conversational interaction. For certain types of applications, certain types of users may be content to sit at a terminal for more than 15 seconds waiting for the answer to a single simple inquiry. However, for a busy person, captivity for more than 15 seconds seems intolerable. If such delays will occur, the system should be designed so that the user can turn to other activities and request the response at some later time.

Greater than 4 seconds

These are generally too long for a conversation requiring the operator to retain information in short-term memory (the operator's memory, not the computer's!). Such delays would be very inhibiting in problem-solving activity and frustrating in data entry activity. However, after a major closure, such as the end of a transaction, delays from 4 to 15 seconds can be tolerated.

2 to 4 seconds

A delay longer than 2 seconds can be inhibiting to terminal operations demanding a high level of concentration. A wait of 2 to 4 seconds at a terminal can seem surprisingly long when the user is absorbed and emotionally committed to complete what he or she is doing. Again, a delay in this range may be acceptable after a minor closure has occurred.

Less than 2 seconds

When the terminal user has to remember information throughout several responses, the response time must be short. The more detailed the information remembered, the greater the need for responses of less than 2 seconds. For elaborate terminal activities, 2 seconds represents an important response-time limit.

Subsecond response time

Certain types of thought-intensive work, especially with graphics applications, require very short response times to maintain the user's interest and attention for long periods of time.

Decisecond response time

A response to pressing a key and seeing the character displayed on the screen or clicking a screen object with a mouse needs to be almost instantaneous—less than 0.1 second after the action. Interaction with a mouse requires extremely fast interaction if the designer is to avoid the use of alien syntax (one with commands, mnemonics, punctuation, etc.).

That rapid response time is the key to productivity in interactive applications has been confirmed in a number of studies [SHNE84; THAD81; GUYN88]. These studies show that when a computer and a user interact at a pace that ensures that neither has to wait on the other, productivity increases significantly, the cost of the work done on the computer therefore drops, and quality tends to improve. It used to be widely accepted that a relatively slow response, up to 2 seconds, was acceptable for most interactive applications because the person was thinking about the next task. However, it now appears that productivity increases as rapid response times are achieved.

The results reported on response time are based on an analysis of online transactions. A transaction consists of a user command from a terminal and the system's reply. It is the fundamental unit of work for online system users. It can be divided into two time sequences:

- **User response time:** The time span between the moment a user receives a complete reply to one command and enters the next command. People often refer to this as think time.
- **System response time:** The time span between the moment the user enters a command and the moment a complete response is displayed on the terminal.

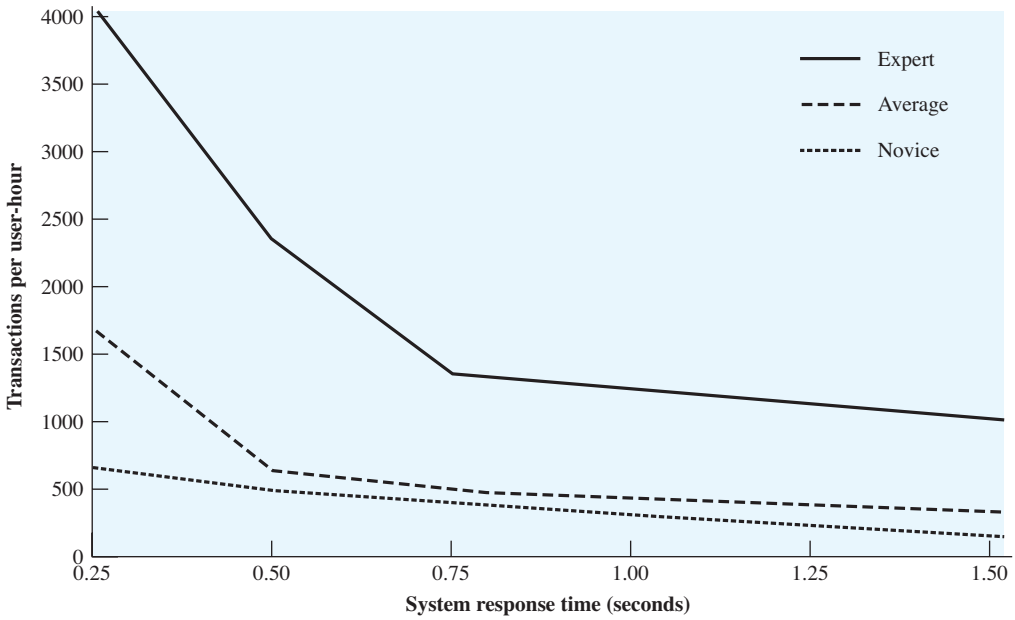


Figure 9.20 **Response Time Results for High-Function Graphics**

As an example of the effect of reduced system response time, Figure 9.20 shows the results of a study carried out on engineers using a computer-aided design graphics program for the design of integrated circuit chips and boards [SMIT83]. Each transaction consists of a command by the engineer that alters in some way the graphic image being displayed on the screen. The results show that the rate of transactions increases as system response time falls and rises dramatically once system response time falls below 1 second. What is happening is that as the system response time falls, so does the user response time. This has to do with the effects of short-term memory and human attention span.

Another area where response time has become critical is the use of the World Wide Web, either over the Internet or over a corporate intranet. The time it takes for a typical Web page to come up on the user's screen varies greatly. Response times can be gauged based on the level of user involvement in the session; in particular, systems with vary fast response times tend to command more user attention. As Figure 9.21 indicates [SEVC96], Web systems with a 3-second or better response time maintain a high level of user attention. With a response time of between 3 and 10 seconds, some user concentration is lost, and response times above 10 seconds discourage the user, who may simply abort the session.

APPENDIX 9B QUEUING SYSTEMS

In this chapter, and several subsequent chapters, results from queuing theory are used. In this appendix we present a brief definition of queuing systems and define key terms. For the reader not familiar with queuing analysis, a basic refresher can

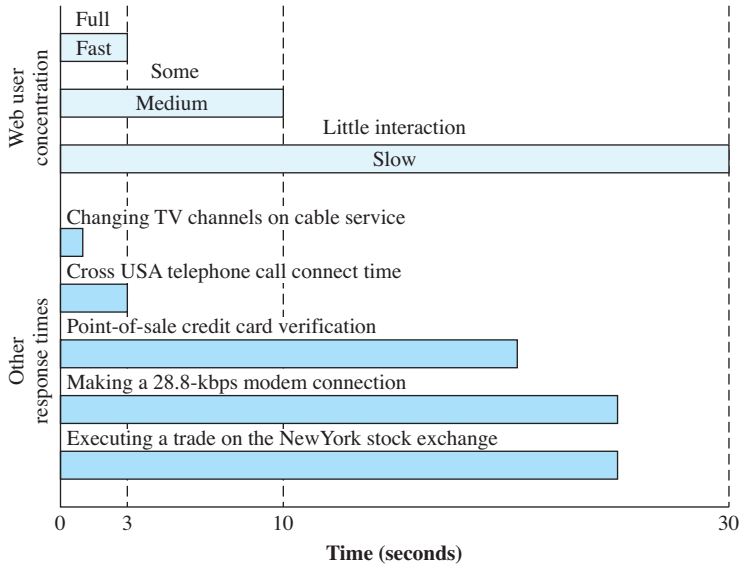


Figure 9.21 **Response Time Requirements**

be found at the Computer Science Student Resource Site at WilliamStallings.com/StudentSupport.html.

Why Queuing Analysis?

It is often necessary to make projections of performance on the basis of existing load information or on the basis of estimated load for a new environment. A number of approaches are possible:

1. Do an after-the-fact analysis based on actual values.
2. Make a simple projection by scaling up from existing experience to the expected future environment.
3. Develop an analytic model based on queuing theory.
4. Program and run a simulation model.

Option 1 is no option at all: we will wait and see what happens. This leads to unhappy users and to unwise purchases. Option 2 sounds more promising. The analyst may take the position that it is impossible to project future demand with any degree of certainty. Therefore, it is pointless to attempt some exact modeling procedure. Rather, a rough-and-ready projection will provide ballpark estimates. The problem with this approach is that the behavior of most systems under a changing load is not what one would intuitively expect. If there is an environment in which there is a shared facility (e.g., a network, a transmission line, a time-sharing system), then the performance of that system typically responds in an exponential way to increases in demand.

Figure 9.22 is a representative example. The upper line shows what typically happens to user response time on a shared facility as the load on that facility increases. The load is expressed as a fraction of capacity. Thus, if we are dealing with a

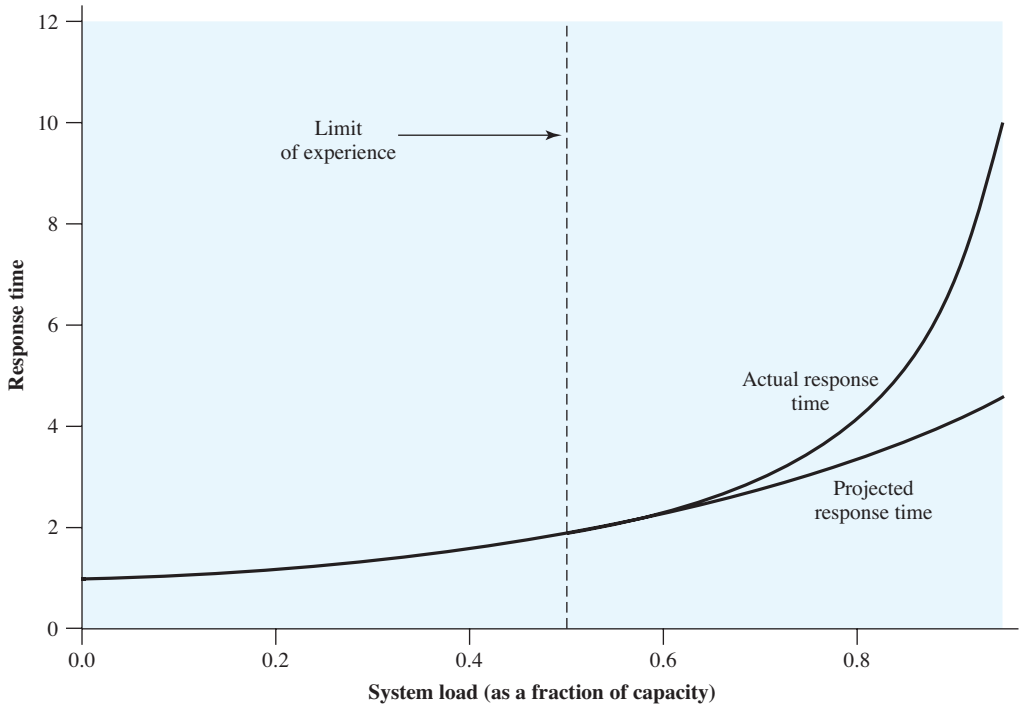


Figure 9.22 **Projected Versus Actual Response Time**

router that is capable of processing and forwarding 1000 packets per second, then a load of 0.5 represents an arrival rate of 500 packets per second, and the response time is the amount of time it takes to retransmit any incoming packet. The lower line is a simple projection[7] based on a knowledge of the behavior of the system up to a load of 0.5. Note that while things appear rosy when the simple projection is made, performance on the system will in fact collapse beyond a load of about 0.8 to 0.9.

Thus, a more exact prediction tool is needed. Option 3 is to make use of an analytic model, which is one that can be expressed as a set of equations that can be solved to yield the desired parameters (response time, throughput, etc.). For computer, operating system, and networking problems, and indeed for many practical real-world problems, analytic models based on queuing theory provide a reasonably good fit to reality. The disadvantage of queuing theory is that a number of simplifying assumptions must be made to derive equations for the parameters of interest.

The final approach is a simulation model. Here, given a sufficiently powerful and flexible simulation programming language, the analyst can model reality in great detail and avoid making many of the assumptions required of queuing theory. However, in most cases, a simulation model is not needed or at least is not advisable as a first step in the analysis. For one thing, both existing measurements and projections of future load carry with them a certain margin of error. Thus, no matter how good the simulation model, the value of the results is limited by the quality of the input.

[7]The lower line is based on fitting a third-order polynomial to the data available up to a load of 0.5.

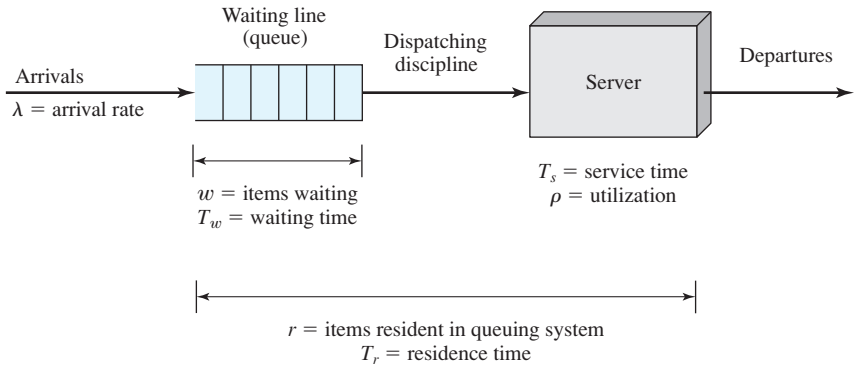


Figure 9.23 **Queuing System Structure and Parameters for Single-Server Queue**

For another, despite the many assumptions required of queuing theory, the results that are produced often come quite close to those that would be produced by a more careful simulation analysis. Furthermore, a queuing analysis can literally be accomplished in a matter of minutes for a well-defined problem, whereas simulation exercises can take days, weeks, or longer to program and run.

Accordingly, it behooves the analyst to master the basics of queuing theory.

The Single-Server Queue

The simplest queuing system is depicted in Figure 9.23. The central element of the system is a server, which provides some service to items. Items from some population of items arrive at the system to be served. If the server is idle, an item is served immediately. Otherwise, an arriving item joins a waiting line.[8] When the server has completed serving an item, the item departs. If there are items waiting in the queue, one is immediately dispatched to the server. The server in this model can represent anything that performs some function or service for a collection of items. Examples: a processor provides service to processes; a transmission line provides a transmission service to packets or frames of data; an I/O device provides a read or write service for I/O requests.

Table 9.8 summarizes some important parameters associated with a queuing model. Items arrive at the facility at some average rate (items arriving per second) λ.

Table 9.8 Notation for Queuing Systems

λ	= arrival rate; mean number of arrivals per second
T_s	= mean service time for each arrival; amount of time being served, not counting time waiting in the queue
ρ	= utilization; fraction of time facility (server or servers) is busy
w	= mean number of items waiting to be served
T_w	= mean waiting time (including items that have to wait and items with waiting time = 0)
r	= mean number of items resident in system (waiting and being served)
T_r	= mean residence time; time an item spends in system (waiting and being served)

[8]The waiting line is referred to as a queue in some treatments in the literature; it is also common to refer to the entire system as a queue. Unless otherwise noted, we use the term *queue* to mean waiting line.

At any given time, a certain number of items will be waiting in the queue (zero or more); the average number waiting is w, and the mean time that an item must wait is T_w. T_w is averaged over all incoming items, including those that do not wait at all. The server handles incoming items with an average service time T_s; this is the time interval between the dispatching of an item to the server and the departure of that item from the server. Utilization, ρ, is the fraction of time that the server is busy, measured over some interval of time. Finally, two parameters apply to the system as a whole. The average number of items resident in the system, including the item being served (if any) and the items waiting (if any), is r; and the average time that an item spends in the system, waiting and being served, is T_r; we refer to this as the mean residence time.[9]

If we assume that the capacity of the queue is infinite, then no items are ever lost from the system; they are just delayed until they can be served. Under these circumstances, the departure rate equals the arrival rate. As the arrival rate increases, the utilization increases and with it, congestion. The queue becomes longer, increasing waiting time. At $\rho = 1$, the server becomes saturated, working 100% of the time. Thus, the theoretical maximum input rate that can be handled by the system is

$$\lambda_{\max} = \frac{1}{T_s}$$

However, queues become very large near system saturation, growing without bound when $\rho = 1$. Practical considerations, such as response time requirements or buffer sizes, usually limit the input rate for a single server to between 70 and 90% of the theoretical maximum.

The following assumptions are typically made:

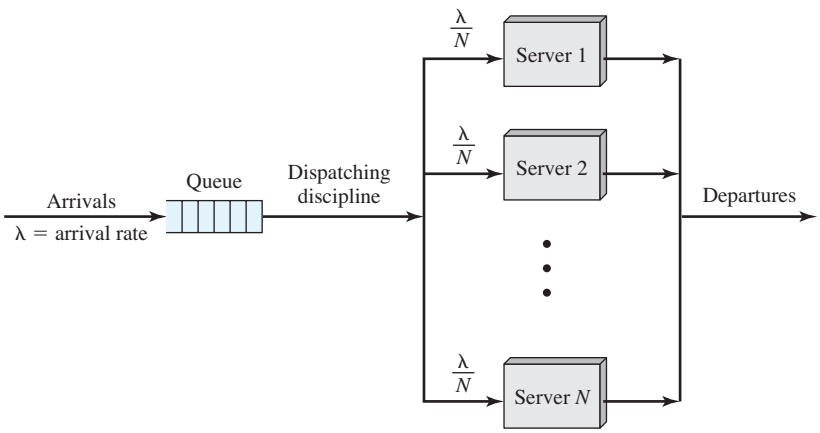
- **Item population:** Typically, we assume an infinite population. This means that the arrival rate is not altered by the loss of population. If the population is finite, then the population available for arrival is reduced by the number of items currently in the system; this would typically reduce the arrival rate proportionally.
- **Queue size:** Typically, we assume an infinite queue size. Thus, the waiting line can grow without bound. With a finite queue, it is possible for items to be lost from the system. In practice, any queue is finite. In many cases, this will make no substantive difference to the analysis.
- **Dispatching discipline:** When the server becomes free, and if there is more than one item waiting, a decision must be made as to which item to dispatch next. The simplest approach is first-in-first-out; this discipline is what is normally implied when the term *queue* is used. Another possibility is last-in-first-out. One that you might encounter in practice is a dispatching discipline based on service time. For example, a packet-switching node may choose to dispatch packets on the basis of shortest first (to generate the most outgoing packets) or longest first (to minimize processing time relative to transmission time). Unfortunately, a discipline based on service time is very difficult to model analytically.

[9]Again, in some of the literature, this is referred to as the mean queuing time, while other treatments use mean queuing time to mean the average time spent waiting in the queue (before being served).

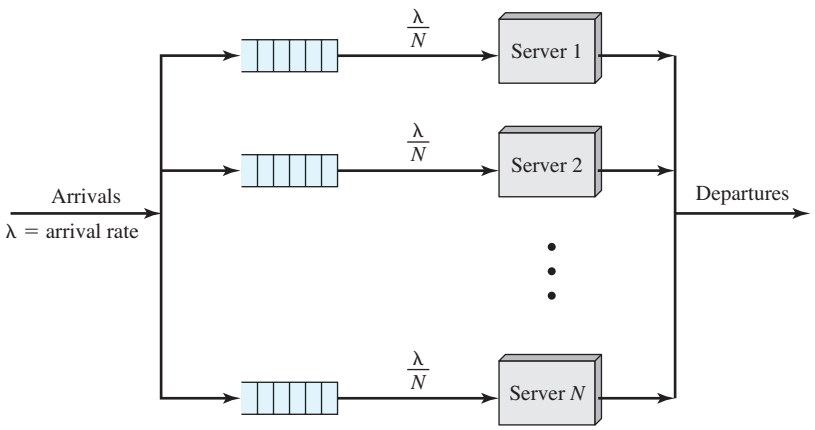
The Multiserver Queue

Figure 9.24 shows a generalization of the simple model we have been discussing for multiple servers, all sharing a common queue. If an item arrives and at least one server is available, then the item is immediately dispatched to that server. It is assumed that all servers are identical; thus, if more than one server is available, it makes no difference which server is chosen for the item. If all servers are busy, a queue begins to form. As soon as one server becomes free, an item is dispatched from the queue using the dispatching discipline in force.

With the exception of utilization, all of the parameters illustrated in Figure 9.23 carry over to the multiserver case with the same interpretation. If we have N identical servers, then ρ is the utilization of each server, and we can consider $N\rho$ to



(a) Multiserver queue



(b) Multiple single-server queues

Figure 9.24 Multiserver versus Multiple Single-Server Queues

be the utilization of the entire system; this latter term is often referred to as the traffic intensity, u. Thus, the theoretical maximum utilization is $N \times 100\%$, and the theoretical maximum input rate is

$$\lambda_{\max} = \frac{N}{T_s}$$

The key characteristics typically chosen for the multiserver queue correspond to those for the single-server queue. That is, we assume an infinite population and an infinite queue size, with a single infinite queue shared among all servers. Unless otherwise stated, the dispatching discipline is FIFO. For the multiserver case, if all servers are assumed identical, the selection of a particular server for a waiting item has no effect on service time.

By way of contrast, Figure 9.24b shows the structure of multiple single-server queues.

Poisson Arrival Rate

Typically, analytic queuing models assume that the arrival rate obeys a Poisson distribution. This is what is assumed in the results of Table 9.6. We define this distribution as follows. If items arrive at a queue according to a Poisson distribution, this may be expressed as

$$\Pr[k \text{ items arrive in time interval } T] = \frac{(\lambda T)^k}{k!} e^{-\lambda T}$$
$$\mathrm{E}[\text{number of items to arrive in time interval } T] = \lambda T$$
$$\text{Mean arrival rate, in items per second} = \lambda$$

Arrivals occurring according to a Poisson process are often referred to as **random arrivals**. This is because the probability of arrival of an item in a small interval is proportional to the length of the interval and is independent of the amount of elapsed time since the arrival of the last item. That is, when items are arriving according to a Poisson process, an item is as likely to arrive at one instant as any other, regardless of the instants at which the other customers arrive.

Another interesting property of the Poisson process is its relationship to the exponential distribution. If we look at the times between arrivals of items T_a (called the interarrival times), then we find that this quantity obeys the exponential distribution:

$$\Pr[T_a < t] = 1 - e^{-\lambda t}$$
$$\mathrm{E}[T_a] = \frac{1}{\lambda}$$

Thus, the mean interarrival time is the reciprocal of the arrival rate, as we would expect.

PROGRAMMING PROJECT 2

THE HOST DISPATCHER SHELL

The Hypothetical Operating System Testbed (HOST) is a multiprogramming system with a four level priority process dispatcher operating within the constraints of finite available resources.

Four-Level Priority Dispatcher

The dispatcher operates at four priority levels:

1. Real-time processes must be run immediately on a first-come-first-served (FCFS) basis, preempting any other processes running with lower priority. These processes are run until completion.
2. Normal user processes are run on a three level feedback dispatcher (Figure P2.1). The basic timing quantum of the dispatcher is 1 second. This is also the value for the time quantum of the feedback scheduler.

The dispatcher needs to maintain two submission queues—Real-Time and User priority—fed from the job dispatch list. The dispatch list is examined at every dispatcher tick and jobs that "have arrived" are transferred to the appropriate submission queue. The submission queues are then examined; any Real-Time jobs are run to completion, preempting any other jobs currently running.

The Real-Time priority job queue must be empty before the lower priority feedback dispatcher is reactivated. Any User priority jobs in the User job queue that can run within available

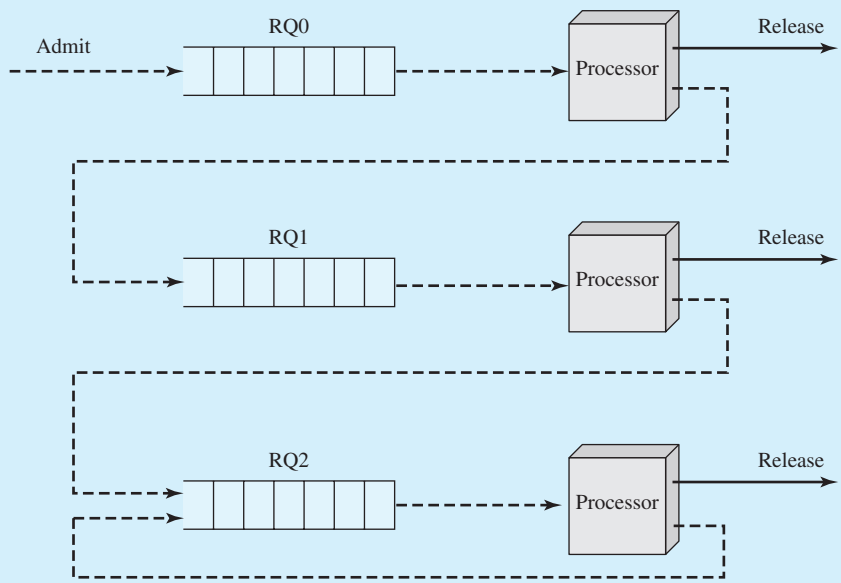


Figure P2.1 **Three-Level Feedback Scheduling**

Figure P2.2 Round-Robin Dispatcher

resources (memory and I/O devices) are transferred to the appropriate priority queue. Normal operation of a feedback queue will accept all jobs at the highest priority level and degrade the priority after each completed time quantum. However, this dispatcher has the ability to accept jobs at a lower priority, inserting them in the appropriate queue. This enables the dispatcher to emulate a simple round robin dispatcher (Figure P2.2) if all jobs are accepted at the lowest priority.

When all "ready" higher priority jobs have been completed, the feedback dispatcher resumes by starting or resuming the process at the head of the highest priority nonempty queue. At the next tick the current job is suspended (or terminated and its resources released) if there are any other jobs "ready" of an equal or higher priority.

The logic flow should be as shown in Figure P2.3 (and as discussed subsequently in this project assignment).

Resource Constraints

The HOST has the following resources:

- 2 Printers
- 1 Scanner
- 1 Modem
- 2 CD drives
- 1024 Mbyte memory available for processes

Low-priority processes can use any or all of these resources, but the HOST dispatcher is notified of which resources the process will use when the process is submitted. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the "ready-to-run" dispatch queues: from the initial transfer from the job queue to the Priority 1–3 queues through to process completion, including intervening idle time quanta.

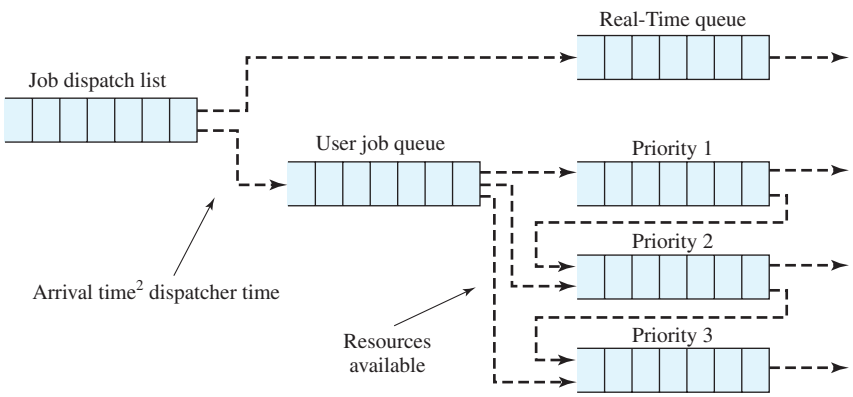


Figure P2.3 Dispatcher Logic Flow

Real-Time processes will not need any I/O resources (Printer, Scanner, Modem, CD) but will obviously require memory allocation—this memory requirement will always be 64 Mbytes or less for Real-Time jobs.

Memory Allocation

For each process, a contiguous block of memory must be assigned. The memory block must remain assigned to the process for the lifetime of the process.

Enough contiguous spare memory must be left so that the Real-Time processes are not blocked from execution—64 Mbytes for a running Real-Time job, leaving 960 Mbytes to be shared among "active" User jobs.

The HOST hardware MMU cannot support virtual memory so no swapping of memory to disk is possible. Neither is it a paged system.

Within these constraints, any suitable variable partition memory allocation scheme (First Fit, Next Fit, Best Fit, Worst Fit, Buddy, etc.) may be used.

Processes

Processes on HOST are simulated by the dispatcher creating a new process for each dispatched process. This process is a generic process (supplied as `process - source: sigtrap.c`) that can be used for any priority process. It actually runs itself at very low priority, sleeping for 1-second periods and displaying the following:

1. A message displaying the process ID when the process starts;
2. A regular message every second the process is executed; and
3. A message when the process is Suspended, Continued, or Terminated.

The process will terminate of its own accord after 20 seconds if it is not terminated by your dispatcher. The process prints out using a randomly generated color scheme for each unique process, so that individual "slices" of processes can be easily distinguishable. Use this process rather than your own.

The life cycle of a process is as follows:

1. The process is submitted to the dispatcher input queues via an initial process list that designates the arrival time, priority, processor time required (in seconds), memory block size, and other resources requested.
2. A process is "ready-to-run" when it has "arrived" and all required resources are available.
3. Any pending Real-Time jobs are submitted for execution on a first-come-first-served basis.
4. If enough resources and memory are available for a lower priority User process, the process is transferred to the appropriate priority queue within the feedback dispatcher unit, and the remaining resource indicators (memory list and I/O devices) updated.
5. When a job is started (`fork and exec("process",...)`), the dispatcher will display the job parameters (Process ID, priority, processor time remaining (in seconds), memory location and block size, and resources requested) before performing the `exec`.
6. A Real-Time process is allowed to run until its time has expired when the dispatcher kills it by sending a `SIGINT` signal to it.
7. A low priority User job is allowed to run for one dispatcher tick (one second) before it is suspended (`SIGTSTP`) or terminated (`SIGINT`) if its time has expired. If suspended, its priority level is lowered (if possible) and it is requeued on the appropriate priority queue as shown in Figures P2.1 and P2.3. To retain synchronization of output between your dispatcher and the child process, your dispatcher should wait for the process to

respond to a SIGTSTP or SIGINT signal before continuing (waitpid(p->pid, &status, WUNTRACED)). To match the performance sequence indicated in the comparison of scheduling policies (see Figure 9.5), the User job should not be suspended and moved to a lower priority level unless another process is waiting to be (re)started.

8. Provided no higher-priority Real-Time jobs are pending in the submission queue, the highest priority pending process in the feedback queues is started or restarted (SIGCONT).
9. When a process is terminated, the resources it used are returned to the dispatcher for reallocation to further processes.
10. When there are no more processes in the dispatch list, the input queues and the feedback queues, the dispatcher exits.

Dispatch List

The Dispatch List is the list of processes to be processed by the dispatcher. The list is contained in a text file that is specified on the command line. That is,

```
>hostd dispatchlist
```

Each line of the list describes one process with the following data as a "*comma-space*" delimited list:

```
<arrival time>, <priority>, <processor time>, <Mbytes>, <#printers>,
<#scanners>, <#modems>, <#CDs>
```

Thus,

```
12, 0, 1, 64, 0, 0, 0, 0
12, 1, 2, 128, 1, 0, 0, 1
13, 3, 6, 128, 1, 0, 1, 2
```

would indicate the following:

1st Job: Arrival at time 12, priority 0 (Real-Time), requiring 1 second of processor time and 64 Mbytes memory—no I/O resources required.

2nd Job: Arrival at time 12, priority 1 (high priority User job), requiring 2 seconds of processor time, 128 Mbytes of memory, 1 printer, and 1 CD drive.

3rd Job: Arrival at time 13, priority 3 (lowest priority User job), requiring 6 seconds of processor time, 128 Mbytes of memory, 1 printer, 1 modem, and 2 CD drives.

The submission text file can be of any length, containing up to 1000 jobs. It will be terminated with an end-of-line followed by an end-of-file marker.

Dispatcher input lists to test the operation of the individual features of the dispatcher are described subsequently in this project assignment. It should be noted that these lists will almost certainly form the basis of tests that will be applied to your dispatcher during marking. Operation as described in the exercises will be expected.

Obviously, your submitted dispatcher will be tested with more complex combinations as well!

A fully functional working example of the dispatcher will be presented during the course. If in any doubt as to the manner of operation or format of output, you should refer to this program to observe how your dispatcher is expected to operate.

Project Requirements

1. Design a dispatcher that satisfies the above criteria. In a formal design document,
   a. Describe and discuss what memory allocation algorithms you could have used and justify your final design choice.

b. Describe and discuss the structures used by the dispatcher for queuing, dispatching, and allocating memory and other resources.

c. Describe and justify the overall structure of your program, describing the various modules and major functions (descriptions of the function "interfaces" are expected).

d. Discuss why such a multilevel dispatching scheme would be used, comparing it with schemes used by "real" operating systems. Outline shortcomings in such a scheme, suggesting possible improvements. Include the memory and resource allocation schemes in your discussions.

The formal design document is expected to have in-depth discussions, descriptions and arguments. The design document is to be submitted separately as a physical paper document. The design document should NOT include any source code.

2. Implement the dispatcher using the C language.

3. The source code MUST be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret and it is in your interests to ensure that the person marking your project is able to understand your coding without having to perform mental gymnastics.

4. Details of submission procedures will be supplied well before the deadline.

5. The submission should contain only source code file(s), include file(s), and a `makefile`. No executable program should be included. The marker will be automatically rebuilding your program from the source code provided. If the submitted code does not compile, it cannot be marked.

6. The `makefile` should generate the binary executable file `hostd` (all lowercase, please). A sample `makefile` would be as follows:

```
# Joe Citizen, s1234567 - Operating Systems Project 2
# CompLab1/01 tutor: Fred Bloggs
hostd: hostd.c utility.c hostd.h
gcc hostd.c utility.c -o hostd
```

The program `hostd` is then generated by typing `make` at the command line prompt. Note: The fourth line in the above `makefile` **MUST** begin with a `tab`.

Deliverables

1. Source code file(s), include file(s), and a makefile.
2. The design document as outlined in Project Requirements section 1 above.

Submission of Code

A `makefile` is required. All files will be copied to the same directory; therefore, *do not include any paths in your makefile*. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not submit any binary or object code files. All that is required is your source code and a `makefile`. Test your project by copying the source code only into an *empty* directory and then compile it with your `makefile`.

The marker will be using a shell script that copies your files to a test directory, performs a `make`, and then exercises your dispatcher with a standard set of test files. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, etc., then the marking sequence will also stop. In this instance, the only further marks that can be awarded will be for the source code and design document.