Chapter 6 Processor Scheduling
(Revision number 15)

6.1 Introduction

Processor design and implementation is no longer a mystery to us. From the earlier chapters, we know how to design instruction-sets, how to implement the instruction-set using an FSM, and how to improve upon the performance of a processor using techniques such as pipelining.

We turn our attention to a complementary topic, namely, how to manage the processor as a resource in a computer system. To do this, we do not need to know the internals of the processor. That is the power of abstraction. We can consider the processor as a black box, an important and scarce resource, and figure out the software abstractions that would be useful in managing this scarce resource.

Let us consider a simple analogy. You have laundry to do. You have tests to prepare for. You have to get some food ready for dinner. You have to call mom and wish her happy birthday. There is only one of you and you have to get all of this done in a timely manner. You are going to prioritize these activities but you also know that not all of them need your constant attention. For example, once you get the washing machine for the laundry started until it beeps at you at the end of the cycle, you don't have to pay attention to it. Similarly, with our zapping culture, all you need to do to get dinner ready is to stick the "TV dinner" into the microwave and wait until that beeps. So, here is a plausible schedule to get all of this done:

1. Start the wash cycle
2. Stick the food in the microwave and zap it
3. Call Mom
4. Prepare for tests

Notice, that your attention is needed only for a very short time for the first two tasks (relative to tasks 3 and 4). There is a problem, however. You have no idea how long tasks 3 and 4 are going to take. For example, Mom's call could go on and on. The washer may beep at you and/or the microwave might beep at you while mom is still on the line. Well, if the washer beeps at you, you could politely tell your mom to hold for a minute, go transfer the load from the washer to dryer and come back to mom's phone call. Likewise, you could excuse yourself momentarily to take the food out of the microwave and keep it on the dinner table ready to eat. Once you are done with mom's call, you are going to eat dinner peacefully, then start preparing for the tests. You have a total of 8 hours for studying and you have four courses to prepare for. All the tests are equally important for your final course grades. In preparing for the tests, you have a choice to make. You could either spend a little bit of time on each of the courses and cycle through all the courses to ensure that you are making progress on all of them. Alternatively, you could study for the tests in the order in which you have to take them.

At this point, you are either scratching your head asking yourself what does any of this have to do with processor scheduling, or more than likely you can already see what is going on here. You are the scarce resource. You are partitioning your time to allocate yourself to these various tasks. You have given higher priority to mom's phone call relative to preparing for the test. You will see later when we discuss processor scheduling algorithms a similar notion of *priority*. You have given starting the wash cycle and the microwave higher priority compared to the other two tasks. This is because you know that these require very little of your time. You will see a similar notion of *shortest job first* scheduling policy later on for processors. When the washer beeps while you are still on the phone, you temporarily put your mom on hold while you attend to the washer. You will see a similar concept, namely *preemption*, later on in the context of processor scheduling. In studying for the tests, the first choice you could make is akin to a processor scheduling policy we will see later on called *round robin*. The second choice is similar to the classic *first come first served* processor-scheduling policy.

6.2 Programs and Processes

Let's start the discussion of processor scheduling with a very basic understanding of what an operating system is. It is just a *program*, whose sole purpose is to supply the resources needed to execute users' programs.

To understand the resource requirements of a user program, let us review how we create programs in the first place. Figure 6.1 shows a plausible layout of the memory footprint of a program written in a high-level language such as C.

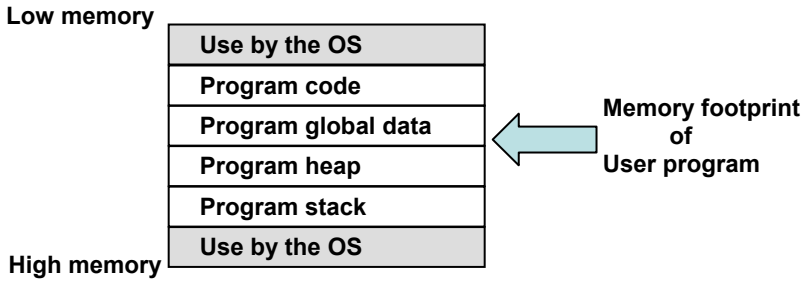
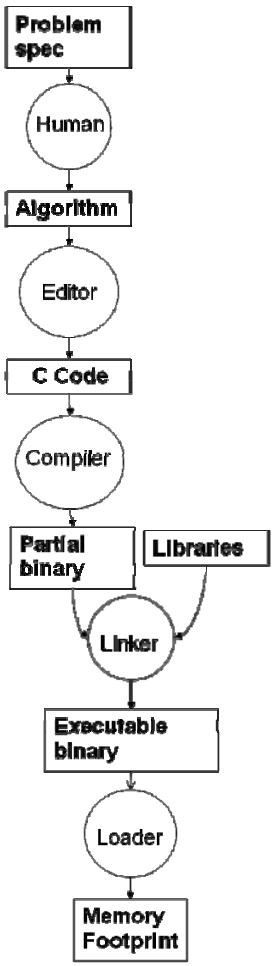


Figure 6.1: Memory Footprint

We use the term *program* in a variety of connotations. However, generally we use the term to denote a computer solution for a problem. The program may exist in several different forms.

Figure 6.2 shows the life cycle of program creation in a high-level language. First, we have a problem specification from which we develop an algorithm. We code the algorithm in some programming language (say C) using an editor. Both the algorithm and the C code are different representations of the *program*, a means of codifying our solution to a problem. A compiler compiles the C code to produce a binary representation of the program. This representation is still not "executable" by a processor. This is because the program we write uses a number of facilities that we take



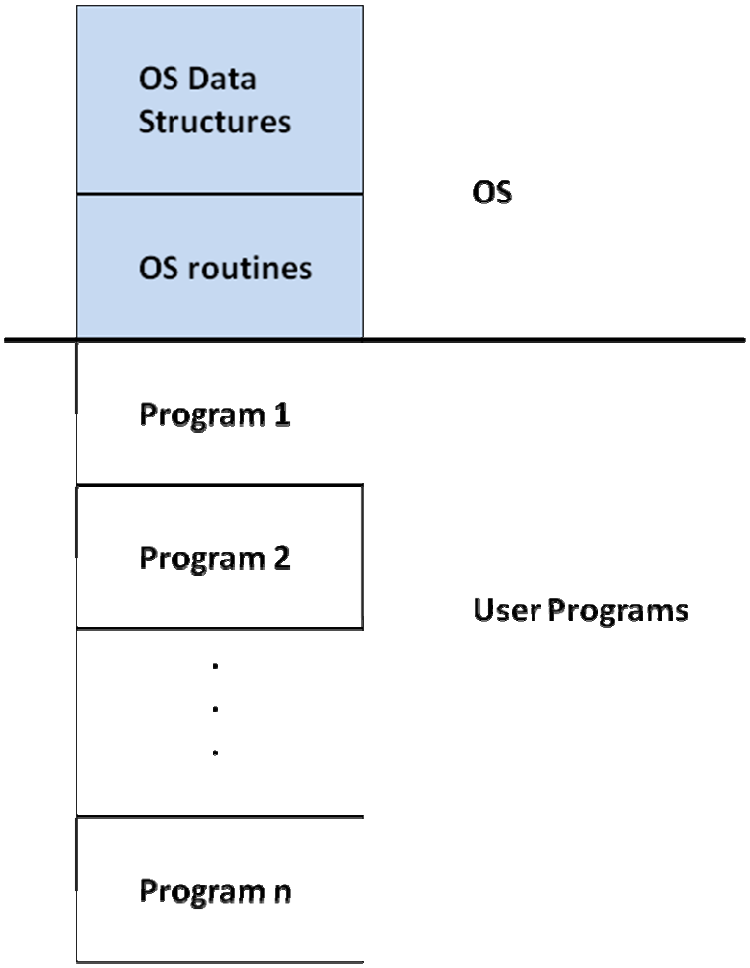


Figure 6.2: Life cycle of program creation

Figure 6.3: OS and user program in memory

for granted and are supplied to us by "someone else". For example, we make calls to do terminal I/O (such as *scanf* and *printf*), and calls to do mathematical operations (such as *sine* and *cosine*). Therefore, the next step is to *link* together our code with that of the libraries provided by someone else that fill in the facilities we have taken for granted. This is the work of the linker. The output of the linker is once again a binary representation of the program, but now it is in a form that is ready for execution on the processor. These different representations of your program (English text, unlinked binary, and executable binary) ultimately end up on your hard drive usually. *Loader* is another program that takes the disk representation and creates the memory footprint shown in Figure 6.1.

Each of Editor, Compiler, Linker, and Loader are themselves programs. Any program needs resources to execute. The resources are processor, memory, and any input/output devices. Let us say you write a simple "Hello World" program. Let us enumerate the

resources needed to run this simple program. You need the processor and memory of course; in addition, the display to send your output. The operating system gives the resources needed by the programs.

As should be evident from the discussion so far, just like any other program the operating system is also memory resident and has a footprint similar to any other program. Figure 6.3 shows the memory contents of user programs and the operating system.

In this chapter, we will focus on the functionality of the operating system that deals with allocating the processor resource to the programs, namely, the *scheduler*.

A scheduler is a set of routines that is part of the operating system; just like any other program, the scheduler also needs the processor to do its work, namely, of selecting a program to run on the processor. The scheduler embodies an algorithm to determine a winner among the set of programs that need cycles on the processor.

You have heard the term *process*. Let us understand what a process is and how it differs from a program. *A process is a program in execution*. With reference to Figure 6.1, we define the *address space* of a process as the space occupied in memory by the program. Once a program starts executing on the processor, the contents of the memory occupied by the program may change due to manipulation of the data structures of the program. In addition, the program may use the processor registers as part of its execution. The current contents of the address space and the register values together constitute the *state* of a program in execution (i.e., the state of a process). We will shortly see how we can concisely represent the state of a process.

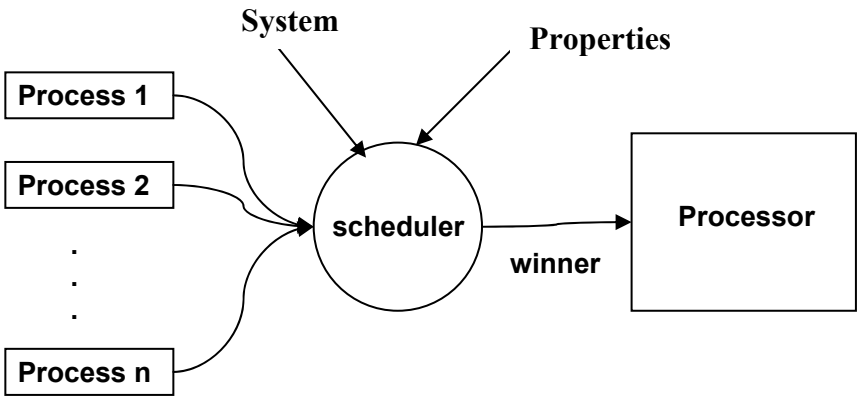


Figure 6.4: Scheduler

Quite often, a process may also be the unit of scheduling on the processor. The inputs to the scheduler are the set of processes that are ready to use the processor, and additional properties that help the scheduler to pick a winner from among the set of ready processes as shown in Figure 6.4. The properties allow the scheduler to establish a sense of *priority* among the set of processes. For example, *expected running time, expected memory usage,* and *expected I/O requirements* are *static* properties associated with a program. Similarly, *available system memory, arrival time of a program,* and *instantaneous memory requirements of the program* are dynamic properties that are

available to the scheduler as well. *Urgency* (either expressed as *deadlines*, and/or *importance*) may be other extraneous properties available to the scheduler. Some of these properties (such as urgency) are explicitly specified to the scheduler; while the scheduler may infer others (such as arrival time).
Quite often, terminologies such as *tasks* and *threads* denote units of work and/or units of scheduling. We should warn the reader that while the definition of process is uniform in the literature, the same could not be said for either a task or a thread. In most literature, a task has the same connotation as a process. In this chapter, we will use the term task only to mean a unit of work. We will give a basic definition of threads that is useful from the point of view of understanding the scheduling algorithms.

An analogy will be useful here. You get the morning newspaper. It is lying on the breakfast table (Figure 6.5-(a)). Nobody is reading it yet. That is like a program dormant in memory. You pick it up and start reading (Figure 6.5-(b)).

(a) **(b)** **(c)**

Figure 6.5: You and your sibling reading the newspaper

Now there is one active entity reading the paper, namely, you. Notice that depending on your interest, you will read different sections of the paper. A process is similar. Depending on the input and the logic of the program, a process may traverse a particular path through the program. Thus, this path defines a *thread of control* for the process. Let us motivate why it makes sense to have multiple threads of control within the process by returning to our paper reading analogy. Imagine now, as you are reading the paper, your sibling joins you at the breakfast table and starts reading the paper as well (Figure 6.5-(c)). Depending on his interests, perhaps he is going to start reading a different section of the paper. Now there are two activities (you and your sibling) or two threads of control navigating the morning paper. Similarly, there could be multiple threads of control within a single process. We will elaborate on why having multiple threads in a process may be a good idea, and the precise differences between a thread and a process later in the context of multiprocessors and multithreaded programs. At this point, it is sufficient to understand a thread as a unit of execution (and perhaps scheduling as well) contained within a process. All the threads within a process execute in the same address space, sharing the code and data structures shown in the program's memory footprint (Figure 6.1). *In other words, a process is a program plus all the threads that are executing in that program.* This is analogous to the newspaper, your sibling, and you, all put together.

Name	Usual Connotation	Use in this chapter
Job	Unit of scheduling	Synonymous with process
Process	Program in execution; unit of scheduling	Synonymous with job
Thread	Unit of scheduling and/or execution; contained within a process	Not used in the scheduling algorithms described in this chapter
Task	Unit of work; unit of scheduling	Not used in the scheduling algorithms described in this chapter, except in describing the scheduling algorithm of Linux

Table 6.1: Jobs, Processes, Threads, and Tasks

There could be multiple threads within a single process but from the point of view of the scheduling algorithms discussed in this chapter, a process has a single thread of control. Scheduling literature uses the term *job* as a unit of scheduling and to be consistent we have chosen to use this term synonymously with process in this chapter. Just as a point of clarification, we summarize these terminologies and the connotations associated with them in Table 6.1.

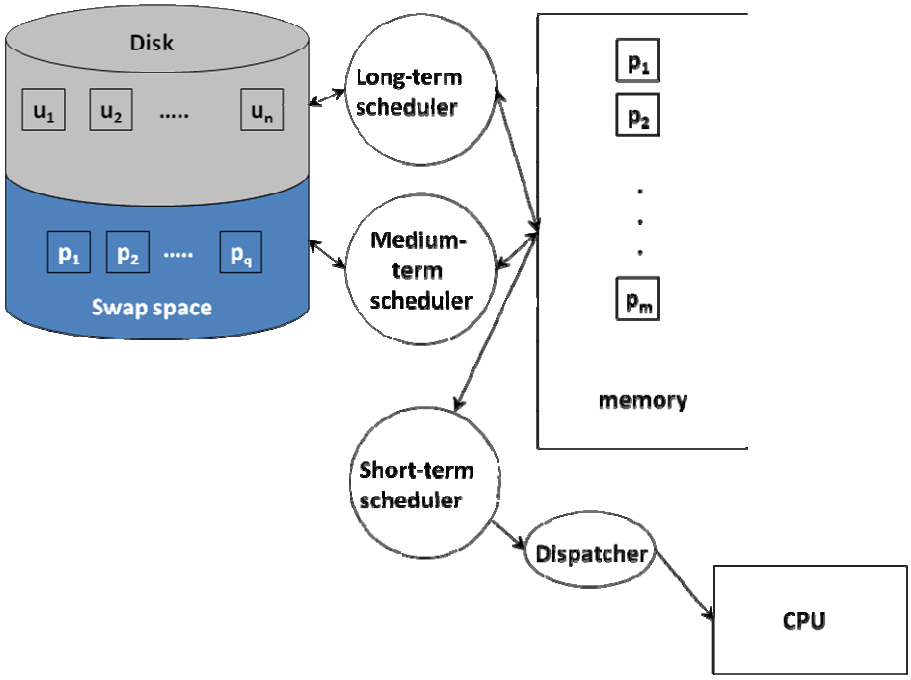


Figure 6.6: Types of schedulers
(u_i represents user programs on the disk;
p_i represents user processes in memory)

6.3 Scheduling Environments

In general, a processor may be dedicated for a specific set of tasks. Take for example a processor inside an embedded device such as a cell phone. In this case, there may be a task to handle the ringing, one to initiate a call, etc. A scheduler for such a dedicated environment may simply cycle through the tasks to see if any one of them is ready to run.

In the age of large batch-oriented processing (60's, 70's, and early 80's), the scheduling environment was *multiprogrammed*; i.e., there were multiple programs loaded into memory from the disk and the operating system cycled through them based on their relative priorities. These were the days when you handed to a human operator, punched cards containing a description of your program (on a disk) and its execution needs, in a language called *job control language (JCL)*. Typically, you would come back several hours later to collect your output. With the advent of data terminals and mini-computers, *interactive* or *time-sharing* environments became feasible. The processor is time-shared among the set of interactive users sitting at terminals and accessing the computer. *It is important to note that a time-shared environment is necessarily multiprogrammed, whereas a multiprogrammed environment need not necessarily be time-shared.* These different environments gave rise to different kinds of schedulers as well (see Figure 6.6).

A *long-term scheduler*, typically used in batch-oriented multiprogrammed environments, balances the job mix in memory to optimize the use of resources in the system (processor, memory, disk, etc.). With the advent of personal computing and time-shared environments, long-term schedulers are for all practical purposes non-existent in most modern operating systems. Instead, a component of the operating system called *loader* creates a memory footprint when the user (for e.g., by clicking an icon on the desktop or typing a program name from a command line) starts a program that is resident on the disk. The long-term scheduler (or the loader) is responsible for creating the memory resident processes (p_i) out of the disk resident user programs (u_i).

A *medium term scheduler,* used in many environments including modern operating systems, closely monitors the dynamic memory usage of the processes currently executing on the CPU and makes decisions on whether or not to increase or decrease the *degree of multiprogramming*, defined as the number of processes coexisting in memory and competing for the CPU. This scheduler is primarily responsible for controlling a phenomenon called *thrashing*, *wherein the current memory requirements of the processes exceed the system capacity, resulting in the processes not making much progress in their respective executions*. The medium-term scheduler moves programs back and forth between the disk (shown as *swap* space in Figure 6.6) and memory when the throughput of the system reduces. We will re-visit the thrashing concept in much more detail in Chapter 8.

A *short-term scheduler*, found in most modern operating systems, made its first appearance in time-sharing systems. This scheduler is responsible for selecting a process to run from among the current set of memory resident processes. The focus of this chapter as well as the algorithms presented mostly concern the short-term scheduler. Lastly, a *dispatcher* is an entity that takes the process selected by the short-term scheduler and sets up the processor registers in readiness for executing that process. Long-term scheduler, medium-term scheduler, short-term scheduler, and dispatcher are all components of the operating system and coordinate their activities with one another. Table 6.2 summarizes the different types of schedulers found in different environments and their respective roles.

Name	Environment	Role
Long term scheduler	Batch oriented OS	Control the job mix in memory to balance use of system resources (CPU, memory, I/O)
Loader	In every OS	Load user program from disk into memory
Medium term scheduler	Every modern OS (time-shared, interactive)	Balance the mix of processes in memory to avoid thrashing
Short term scheduler	Every modern OS (time-shared, interactive)	Schedule the memory resident processes on the CPU
Dispatcher	In every OS	Populate the CPU registers with the state of the process selected for running by the short-term scheduler

Table 6.2: Types of Schedulers and their roles

6.4 Scheduling Basics

It is useful to understand program behavior before delving deep into the scheduler. Imagine a program that plays music on your computer from a CD player. The program repeatedly reads the tracks of the CD (I/O activity) and then renders the tracks (processor activity) read from the CD player to the speakers. It turns out that this is typical program behavior, cycling between bursts of activity on the processor and I/O devices (see Figure 6.7).

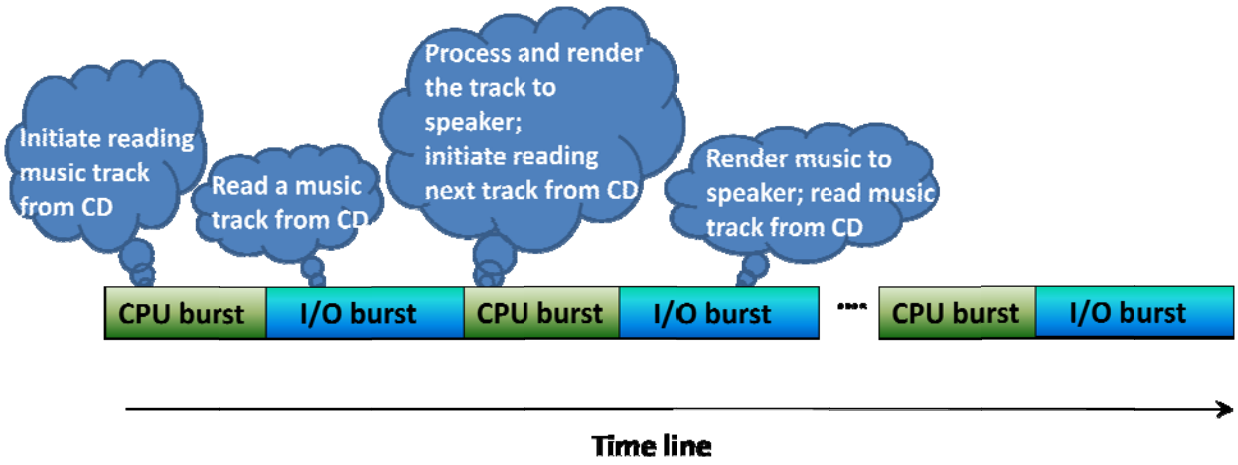


Figure 6.7: Music playing on your CD player

We will use the term *CPU burst* to denote the stretch of time a process would run without making an I/O call. Informally, we define CPU burst of a process as the time interval of continuous CPU activity by the process before making an I/O call. Using the analogy from the beginning of the chapter, CPU burst is similar to stretch of time you would do continuous reading while preparing for a test before going to the fridge for a soda. Similarly, we will use the term *I/O burst* to denote the stretch of time a process would need to complete an I/O operation (such as reading a music file from the CD). It is important to note that during an I/O burst the process does not need to use the processor. In Chapter 4, we introduced the concept of interrupts and its use to grab the attention of the processor for an external event such as I/O completion. Later in Chapter 10, we will discuss the actual mechanics of data transfer to/from the I/O device from/to the processor and memory. For now, to keep the discussion of processor scheduling simple, assume that upon an I/O request by a process, it is no longer in contention for the CPU resource until its I/O is complete.

Processor schedulers are divided into two broad categories: *non-preemptive* and *preemptive*. A process either executes to completion or gives up the processor on its own accord, i.e., voluntarily, (to perform I/O) in a non-preemptive scheduler. On the other hand, the scheduler yanks the processor away from the current process to give it to another process in a preemptive scheduler. In either case, the steps involved in scheduling are the following:

1. Grab the attention of the processor.
2. Save the state of the currently running process.
3. Select a new process to run.
4. Dispatch the newly selected process to run on the processor.

The last step *dispatch* refers to loading the processor registers with the saved state of the selected process.

Let us understand the state of the running program or process. It includes where we are currently executing in the program (PC value); what the contents of the processor registers are (assuming that one of these registers is also the stack pointer); and where in

memory is the program's footprint. Besides, the process itself may be newly loaded into memory, ready to run on the processor, or waiting for I/O, currently running, halted for some reason, etc.

```
enum  state_type {new, ready, running, waiting, halted};

typedef struct control_block_type {
   enum state_type state;          /* current state */
   address PC;                     /* where to resume */
   int reg_file[NUMREGS];          /* contents of GPRs */
   struct control_block *next_pcb; /* list ptr */
   int priority;                   /* extrinsic property */
   address address_space;          /* where in memory */
          ….
          ….
} control_block;
```

Figure 6.8: Process Control Block (PCB)

Additionally, properties (intrinsic or extrinsic) that the scheduler may know about a process such as process priority, arrival time of the program, and expected execution time. All this state information is aggregated into a data structure, *process control block (PCB)*, shown in Figure 6.8. There is one PCB for each process and the scheduler maintains all the PCBs in a linked list, the *ready queue,* as shown in Figure 6.9.

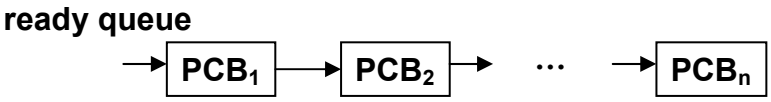


Figure 6.9: Ready queue of PCBs

The PCB has all the information necessary to describe the process. It is a key data structure in the operating system. As we will see in later chapters that discuss memory systems and networking, the PCB is the aggregation of all the state information associated with a process (such as memory space occupied, open files, and network connections). Ready queue is the most important data structure in the scheduler. Efficient representation and manipulation of this data structure is a key to the performance of the scheduler. The role of the scheduler is to quickly make its scheduling decision and get out of the way, thus facilitating the use of the CPU for running user programs. Therefore, a key question is identifying the right metrics for evaluating the *efficiency* of a scheduling algorithm. Intuitively, we would like the time spent in the scheduler to be a small percentage of the total CPU time. We will see in a case study (Section 6.12), how Linux scheduler organizes its data structures to ensure high efficiency.

Note that we do not have the internal registers of the CPU datapath (Chapters 3 and 5) as part of the process state. The reason for this will become clear towards the end of this subsection.

Similar to the ready queue of process control blocks, the operating system maintains queues of PCBs of processes that are waiting on I/O (see Figure 6.10).

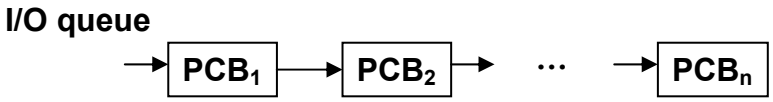


Figure 6.10: I/O queue of PCBs

For the purposes of this discussion, the PCBs go back and forth between the ready queue and the I/O queue depending on whether a process needs the processor or I/O service.

The CPU scheduler uses the ready queue for scheduling processes on the processor. Every one of the scheduling algorithms we discuss in this chapter assumes such a ready queue. The organization of the PCBs in the ready queue depends on the specifics of the scheduling algorithm. The PCB data structure simplifies the steps involved in scheduling which we identified earlier in this section. The scheduler knows exactly which PCB corresponds to the currently running process. The state saving step simply becomes a matter of copying the relevant information (identified in Figure 6.8) into the PCB of the currently running process. Similarly, once the scheduler chooses a process as the next candidate to run on the processor, dispatching it on the processor is simply a matter of populating the processor registers with the information contained in the PCB of the chosen process.

From Chapter 4, we know that system calls (such as an I/O operation) and interrupts are all different flavors of program discontinuities. The hardware treats all of these program discontinuities similarly, waiting for a clean state of the processor before dealing with the discontinuity. Completion of an instruction execution is such a clean state of the processor. Once the processor has reached such a clean state, the registers that are internal to the processor (i.e., not visible to the programmer) do not contain any information of relevance to the currently running program. Since, the scheduler switches from one process to another triggered by such well-defined program discontinuities, there is no need to save the internal registers of the processor (discussed in Chapter 3 and 5) that are not visible to the programmer through the instruction-set architecture.

Table 6.3 summarizes the terminologies that are central to scheduling algorithms.

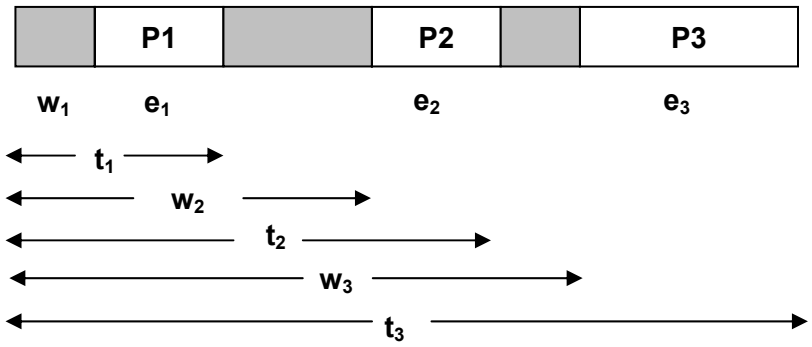
Name	Description
CPU burst	Continuous CPU activity by a process before requiring an I/O operation
I/O burst	Activity initiated by the CPU on an I/O device
PCB	Process context block that holds the state of a process (i.e., program in execution)
Ready queue	Queue of PCBs that represent the set of memory resident processes that are ready to run on the CPU
I/O queue	Queue of PCBs that represent the set of memory resident processes that are waiting for some I/O operation either to be initiated or completed
Non-Preemptive algorithm	Algorithm that allows the currently scheduled process on the CPU to voluntarily relinquish the processor (either by terminating or making an I/O system call)
Preemptive algorithm	Algorithm that forcibly takes the processor away from the currently scheduled process in response to an external event (e.g. I/O completion interrupt, timer interrupt)
Thrashing	A phenomenon wherein the dynamic memory usage of the processes currently in the ready queue exceeds the total memory capacity of the system

Table 6.3: Scheduling Terminologies

6.5 Performance Metrics

In discussing scheduling algorithms, we use the term *jobs* and *processes* synonymously. Scheduler, a part of the operating system, is also a program and needs to run on the processor. Ultimately, the goal of the scheduler is to run user programs. Therefore, a useful way to think about putting the processor to good use is by using it for running user programs and not the operating systems itself. This brings us to the question of the right metrics to use for evaluating the efficiency of a scheduling algorithm. *CPU Utilization* refers to the percentage of time the processor is busy. While this is a useful metric, it does not tell us what the processor is doing. So let us try some other metrics. The metrics could be *user centric* or *system centric*. *Throughput* is a system centric metric that measures the number of jobs executed per unit time. *Average turnaround time* is another system centric metric that measures the average elapsed time for jobs entering and leaving the system. *Average waiting time* of jobs is another system centric metric. *Response time* of a job is a user centric metric that measures the elapsed time for a given job.

Figure 6.11 shows a time line of scheduling three processes P1, P2, and P3 on the processor. Assume that all the processes start at time '0'. For the sake of this discussion, assume that in the shaded regions the processor is busy doing something else unrelated to running these processes.



w_i, e_i, and t_i, are respectively the wait time, execution time, and the elapsed time for a job j_i

Figure 6.11: Time line of scheduling three processes P1, P2, P3

With respect to Figure 6.11,

Throughput = $3/t_3$ jobs/sec

Average turnaround time = $(t_1 + t_2 + t_3)/3$ secs

Average wait time = $((t_1-e_1) + (t_2-e_2) + (t_3-e_3))/3$ secs

Generalizing for n jobs,

Throughput = n/T jobs/sec, where T is the total elapsed time for all n jobs to complete,

Average turnaround time = $(t_1+t_2+\ldots t_n)/n$ secs

Average wait time = $(w_1+w_2+ \ldots +w_n)/n$ secs

Response time, is the same as per-process turnaround time,

$R_{P1} = t1$

$R_{p2} = t2$

$R_{p3} = t3$

….

$R_{pn} = tn$

Variance[1] in response time is a useful metric as well. In addition to these quantitative metrics, we should mention two *qualitative* metrics of importance to scheduling algorithms:

- **Starvation**: In any job mix, the scheduling policy should make sure that all the jobs make forward progress towards completion. We refer to the situation as *starvation*, if for some reason a job does not make any forward progress. The quantitative manifestation of this situation is an unbounded response time for a particular job.
- **Convoy effect**: In any job mix, the scheduling policy should strive to prevent long-running jobs from dominating the CPU usage. We refer to the situation as *convoy effect*, if for some reason the scheduling of jobs follows a fixed pattern (similar to a military convoy). The quantitative manifestation of this phenomenon is a high variance in the response times of jobs.

[1] Variance in response time is the average of the squared distances of the possible values for response times from the expected value.

Name	Notation	Units	Description
CPU Utilization	-	%	Percentage of time the CPU is busy
Throughput	n/T	Jobs/sec	System-centric metric quantifying the number of jobs *n* executed in time interval *T*
Avg. Turnaround time (t_{avg})	$(t_1+t_2+\ldots+t_n)/n$	Seconds	System-centric metric quantifying the average time it takes for a job to complete
Avg. Waiting time (w_{avg})	$((t_1-e_1) + (t_2-e_2)+ \ldots + (t_n-e_n))/n$ or $(w_1+w_2+ \ldots +w_n)/n$	Seconds	System-centric metric quantifying the average waiting time that a job experiences
Response time/ turnaround time	t_i	Seconds	User-centric metric quantifying the turnaround time for a specific job *i*
Variance in Response time	$E[(t_i - e_i)^2]$	$Seconds^2$	User-centric metric that quantifies the statistical variance of the actual response time (t_i) experienced by a process (P_i) from the expected value (t_{avg})
Starvation	-	-	User-centric qualitative metric that signifies denial of service to a particular process or a set of processes due to some intrinsic property of the scheduler
Convoy effect	-	-	User-centric qualitative metric that results in a detrimental effect to some set of processes due to some intrinsic property of the scheduler

Table 6.4: Summary of Performance Metrics

We will discuss several scheduling algorithms in the next few sections. Before we do that, a few caveats:

- In all the scheduling algorithms, we assume the time to switch from one process to another is negligible to make the timing diagrams for the schedules simple.
- We mentioned that a process might go back and forth between CPU and I/O requests during its lifetime. Naturally, the I/O requests may be to different devices at different times (output to the screen, read from the disk, input from the mouse, etc.). However, since the focus in this chapter is on CPU scheduling, for simplicity we just show one I/O queue.
- Once again, to keep the focus on CPU scheduling, we assume a simple model (first-come-first-served) for scheduling I/O requests. In other words, intrinsic or extrinsic properties of a process that the CPU scheduler uses, do not apply to I/O scheduling. The I/O requests are serviced in the order in which they are made by the processes.

Table 6.4 summarizes the performance metrics of interest from the point of view of scheduling.

6.6 Non-preemptive Scheduling Algorithms

As we mentioned earlier, a non-preemptive algorithm is one in which the scheduler has no control over the currently executing process once it has been scheduled on the CPU. The only way the scheduler can get back control is if the currently running process voluntarily relinquishes the CPU either by terminating or by making a blocking system call (such as a file I/O request). In this section, we will consider three different algorithms that belong to this class: FCFS, SJF, and priority.

6.6.1 First-Come First-Served (FCFS)

The intrinsic property used in this algorithm is the *arrival time* of a process. Arrival time is the time when you launch an application program. For example, if you launched winamp at time t_0 and realplayer at a later time t_1, then winamp has an earlier arrival time as far as the scheduler is concerned. Thus, during the entire lifetime of the two programs, whenever both programs are ready to run, winamp will always get picked by the scheduler since it has an earlier arrival time. Remember that this "priority" enjoyed by winamp will continue even when it comes back into the ready queue after an I/O completion. Example 1 show this advantage enjoyed by the early bird process compared to other ready processes in the ready queue.

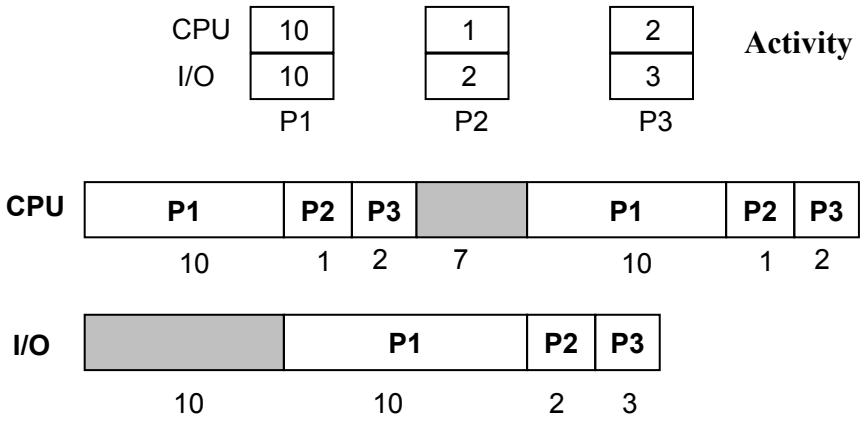


Figure 6.12: FCFS convoy effect

Figure 6.12 shows a set of processes and their activities at the top of the figure. Each process's activity alternates between CPU burst and I/O burst. For example, P2 does 1 unit of processing and 2 units of I/O, and repeats this behavior for its lifetime. The bottom of Figure 6.12 presents a timeline showing the FCFS scheduling of these processes on the CPU and I/O. There can be exactly one process executing on the CPU or carrying out an I/O activity at a time. Assume that each process has to do 2 bursts of CPU activity interspersed with one burst of I/O. All three processes are in the ready queue of the scheduler at the beginning of the time line; however, P1 is the first to arrive, followed by P2 and P3. Thus, given a choice the scheduler will always pick P1 over P2 or P3 for scheduling due to the FCFS policy. The waiting times for the three jobs P1, P2, and P3 are 0, 27, and 26, respectively.

The algorithm has the nice property that there will be no *starvation* for any process, i.e., the algorithm has no inherent bias that results in denial of service for any process. We will see shortly that not all algorithms have this property. However, due to its very nature there can be a huge variation in the response time. For instance if a short job arrives just after a long job then the response time for the short job will be very bad. The algorithm also results in poor processor utilization due to the *convoy effect* depicted in Figure 6.12. The term comes from the primarily military use of the word "convoy", which signifies a group of vehicles traveling together. Of course, in the military sense a convoy is a good thing since the vehicles act as support for one another in times of emergency. You may inadvertently become part of a convoy in a highway when you are stuck behind a slow moving vehicle in single lane traffic. The short jobs (P2 and P3) are stuck behind the long job (P1) for CPU. The convoy effect is unique to FCFS scheduling and is inherent in the nature of this scheduling discipline. Many of us have experienced being stuck behind a customer with a cartload of stuff at the checkout counter, when we ourselves have just a couple of items to check out. Unfortunately, the convoy effect is intrinsic to the FCFS scheduling discipline since by its very nature it does not give any preferential treatment for short jobs.

Example 1:

Consider a non-preemptive First Come First Served (FCFS) process scheduler. There are three processes in the scheduling queue and the arrival order is P1, P2, and P3. The arrival order is always respected when picking the next process to run on the processor. Scheduling starts at time t = 0, with the following CPU and I/O burst times:

	CPU burst time	I/O burst time
P1	8	2
P2	5	5
P3	1	5

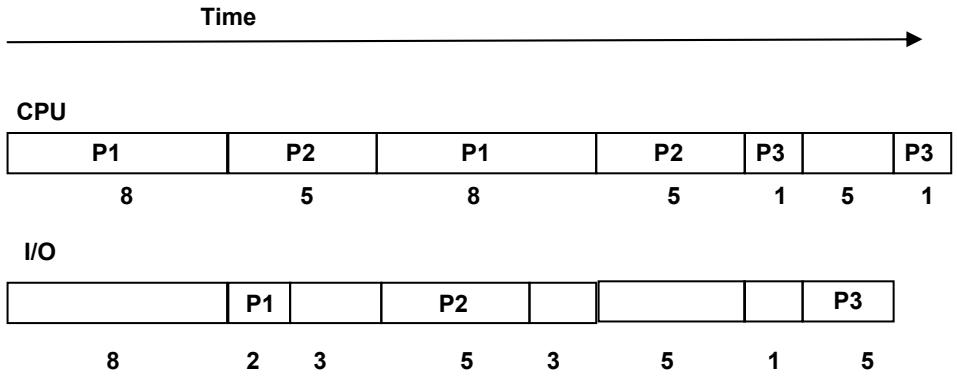
Each process terminates after completing the following sequence of three actions:

1. CPU burst
2. I/O burst
3. CPU burst

a) Show the CPU and I/O timelines that result with FCFS scheduling from t = 0 until all three processes complete.
b) What is the response time for each process?
c) What is the waiting time for each process?

Answer:

a)



Notice that at time t=8, P2 and P3 are in the ready queue. P1 makes an I/O request and relinquishes the processor. The scheduler picks P2 to run on the processor due to its earlier arrival time compared to P3. At time t=10, P1 completes its I/O and rejoins the ready queue. P3 is already in the ready queue at this time. However, by virtue of its earlier arrival time, P1 gets ahead of P3 in the ready queue, which is why P1 gets picked by the scheduler at time t=13.

b)

We compute the response time for each process as the total time spent by the process in the system from the time it enters to the time it departs.

Response time (P1) = 21
Response time (P2) = 26
Response time (P3) = 33

c)

Each process does useful work when it is either executing on the CPU or performing its I/O operation. Thus, we compute the wait time for each process by subtracting useful work done by a process from its total turnaround time (or response time).

For example, the useful work done by P1
= First CPU burst + I/O burst + Second CPU burst
= 8 + 2 + 8
= 18

Thus, the wait time for P1
Wait-time (P1) = (21 – 18) = 3

Similarly,
Wait-time (P2) = (26 – 15) = 11
Wait-time (P3) = (33 – 7) = 26

As we mentioned earlier, the scheduler always respects the arrival time of a process for its lifetime, when picking a winner to schedule on the CPU. Further, this intrinsic property is meaningful only for CPU scheduling and not I/O. These two points are illustrated in the following example.

Example 2:

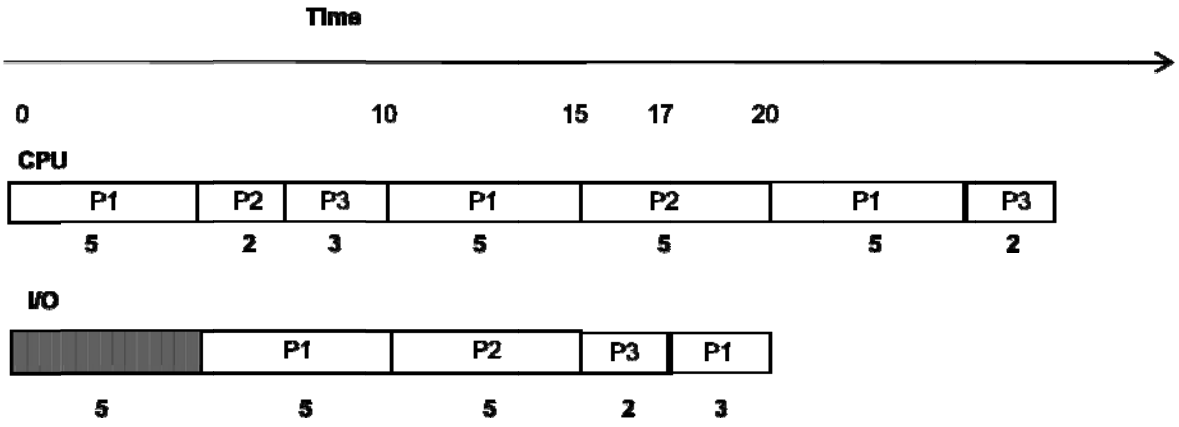
Consider a FCFS process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, the scheduler always respects the arrival time in selecting a winner. Assume that P1, P2, and P3 arrive in that order into the system. Scheduling starts at time t = 0.

The CPU and I/O burst patterns of the three processes are as shown below:

	CPU	I/O	CPU	I/O	CPU	
P1	5	5	5	3	5	**P1 is done**
P2	2	5	5			**P2 is done**
P3	3	2	2			**P3 is done**

Show the CPU and I/O timelines that result with FCFS scheduling from t = 0 until all three processes complete.

Answer:



Notes:

1) At time t=15, both P3 and P1 need to perform I/O. However, P3 made the I/O request first (at time t=10), while P1 made its request second (at time t=15). As we mentioned earlier, the I/O requests are serviced in the order received. Thus P3 gets serviced first and then P1.
2) At time t=20, both P3 and P1 are needing to get on the CPU. In fact, P3 finished its I/O at time t=17, while P1 just finished its I/O at time t=20. Yet, P1 wins this race to get on the CPU since the CPU scheduler always respects the arrival time of processes in making its scheduling decision.

6.6.2 Shortest Job First (SJF)

The term "shortest job" comes from scheduling decisions taken in shop floors, for example in a car mechanic shop. Unfortunately, this can sometimes lead to the wrong connotation when literally applied to CPU scheduling. Since the CPU scheduler does not know the exact behavior of a program, it works only with partial knowledge. In the case of SJF scheduling algorithm, this knowledge is the *CPU burst time* needed, an intrinsic property of each process. As we know, each process goes through bursts of CPU and I/O activities. The SJF scheduler looks at the CPU burst times needed by the current set of processes that are ready to run and picks the one that needs the shortest CPU burst. Recalling the analogy at the beginning of the chapter, you started the washer and the microwave before you made your phone call to your mom. SJF scheduler uses the same principle of favoring short jobs. In reality, a scheduler does not know the CPU burst time for a program at program start up. However, it infers the expected burst time of a process from past CPU bursts.

This algorithm results in a better response time for short jobs. Further, it does not suffer from the convoy effect of FCFS since it gives preferential treatment to short jobs. It turns

out that this algorithm is also provably optimal for yielding the best average waiting time. Figure 6.13 shows the time line for executing the same process activities as in Figure 6.12 for FCFS. The waiting times for the three jobs P1, P2, and P3 are 4, 0, and 9, respectively.

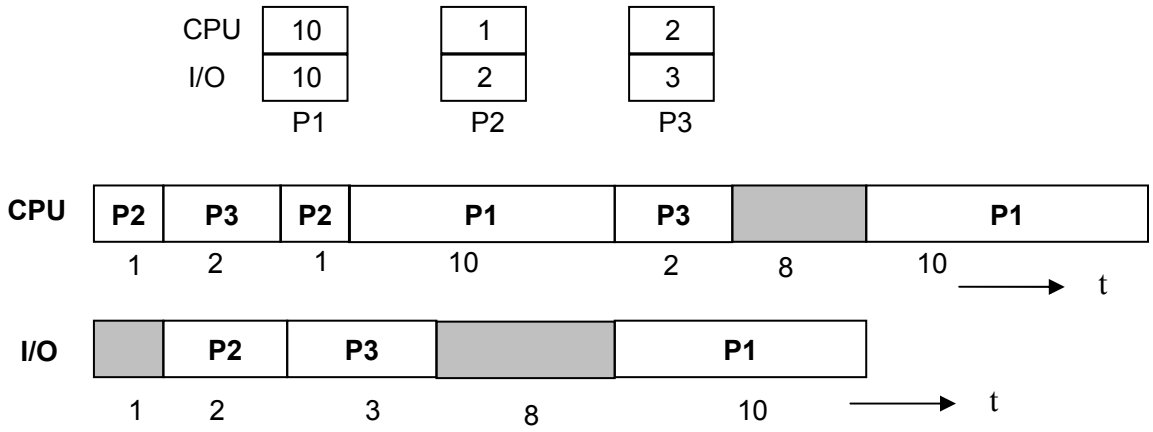


Figure 6.13: SJF schedule

Notice that the shortest job, P2, gets the best service with this schedule. At time t = 3, P2 has just completed its I/O burst; fortuitously, P3 has also just finished its CPU burst. At this time, though P1 and P2 are ready to be scheduled on the CPU, the scheduler gives preference to P2 over P1 due to P2's shorter CPU burst requirement. At time t = 4, P2 finishes its CPU burst. Since there is no other shorter job to schedule, P1 gets a chance to run on the CPU. At time t = 6, P3 has just completed its I/O burst and is ready to be scheduled on the CPU. However, P1 is currently executing on the processor. Since the scheduler is non-preemptive, P3 has to wait for P1 to give up the processor on its own accord (which it does at t = 14).

There is a potential for starvation for long jobs in the SJF schedule. In the above example, before P1 gets its turn to run, new shorter jobs may enter the system. Thus, P1 could end up waiting a long time, perhaps even forever. To overcome this problem, a technique, *aging,* gives preference to a job that has been waiting in the ready queue for a long time over other short jobs. Basically, the idea is for the scheduler to add another predicate to each job, namely, the time it entered the scheduling mix. When the "age" of a job exceeds a threshold, the scheduler will override the SJF principle and give such a job preference for scheduling.

Example 3:

Consider a non-preemptive Shortest Job First (SJF) process scheduler. There are three processes in the scheduling queue and assume that all three of them are ready to run. As the scheduling discipline suggests, always the shortest job that is ready to run is given priority. Scheduling starts at time t = 0. The CPU and I/O burst patterns of the three processes are as shown below:

	CPU	I/O	CPU	I/O	CPU	
P1	4	2	4	2	4	**P1 is done**
P2	5	2	5			**P2 is done**
P3	2	2	2	2	2	**P3 is done**

Each process exits the system once its CPU and I/O bursts as shown above are complete.

a) Show the CPU and I/O timelines that result with SJF scheduling from t = 0 until all three processes exit the system.

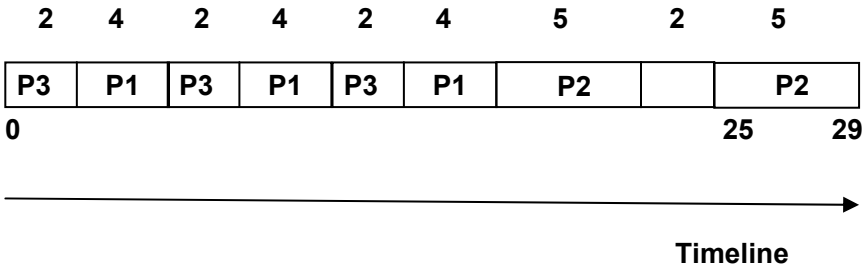
b) What is the waiting time for each process?

c) What is the average throughput of the system?

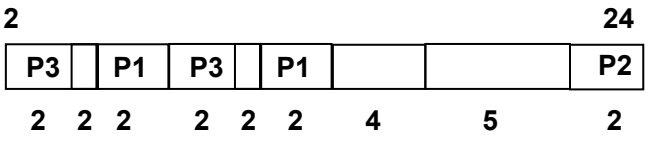
Answer:

(a)

CPU Schedule (SJF)



I/O Schedule



b)

We compute the wait times for each process as we did in Example 1.

Wait-time (P1) = (18 – 16) = 2
Wait-time (P2) = (30 – 12) = 18
Wait-time (P3) = (14 – 10) = 4

c)

Total time = 30

Throughput = number of processes completed / total time
= 3/30
= 1/10 processes per unit time

6.6.3 Priority

Most operating systems associate an extrinsic property, *priority*, a small integer value, with each process to denote its relative importance compared to other processes for scheduling purposes. For example, in the Unix operating system every user level process starts with a certain default priority. The ready queue consists of several sub-queues, one for each priority level as shown in Figure 6.14.

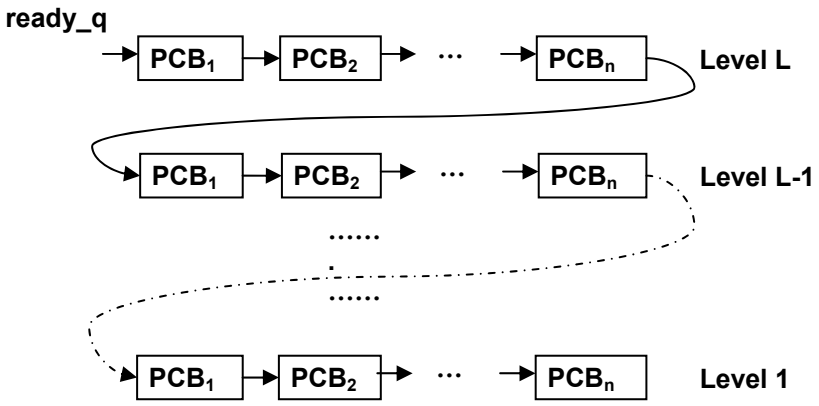


Figure 6.14: Multi-level Ready Queue for Priority Scheduler

The scheduling within each level is FCFS. New processes are placed in the queue commensurate with their priority levels. Since priority is an extrinsic property, the scheduler has complete control on the assignment of priorities to different processes. In this sense, this scheduling discipline is highly flexible compared to SJF or FCFS. For example, the scheduler may assign priorities based on the class of users. This would be particularly attractive from the point of view of running data centers[2] wherein different users may be willing to pay different rates for their respective jobs.

[2] Data centers are becoming popular as a way of providing access to high-performance computing resources to users without the need for individual users owning such resources. Companies such as

Priority is a very natural way of providing differential service guarantees for different users, not just in processor scheduling, but also in almost every walk of life. As an example, call centers operated by businesses that deal with customer service requests place different callers in different queues based on their profiles. "High maintenance" callers go into a slower queue with longer service times as opposed to preferred customers. Similarly, airlines use first class, business class, and economy class as a way of prioritizing service to its customers.

With a little bit of reflection, one could figure out that SJF is really a special case of priority scheduling wherein the priority level,

$$L = 1/CPU\_burst\_time$$

Thus, similar to SJF, priority scheduler has the problem of starvation of processes with low priority. Countering this problem with explicit priority levels assigned to each process is straightforward. Similar to the solution we suggested for SJF, when the age of a process exceeds a preset threshold, the scheduler will artificially bump up the priority of the process.

If you think about it, you will realize that FCFS is also a priority-based algorithm. It is just that the scheduler uses arrival time of the process as its priority. Due to this reason, no newly arriving process can have a priority higher than the current set of processes in FCFS, which is also the reason FCFS does not have the problem of starvation.

Due to the similarity with FCFS, a priority-based algorithm could also exhibit the convoy effect. Let us see how this could happen. If a process with a higher priority also turns out to be a long running process then we could end up with a situation similar to what we observed in the FCFS schedule. However, in FCFS, once assigned the process priority never changes. This need not be the case in a priority-based scheduler; the same mechanism (bumping up the priority based on age) used to overcome the starvation problem would also help to break the convoy effect.

6.7 Preemptive Scheduling Algorithms

This class of scheduling algorithms implies two things simultaneously. First, the scheduler is able to assume control of the processor anytime unbeknown to the currently running process. Second, the scheduler is able to save the state of the currently running process for proper resumption from the point of preemption. Returning to our analogy from the beginning of the chapter, when the washer beeps while you are on the phone with mom, you excuse yourself; and make a mental note of where to resume the conversation when you get back on the phone again.

In principle, any of the algorithms discussed in the previous section could be made preemptive. To accomplish that, in the case of FCFS, whenever a process rejoins the ready queue after I/O completion, the scheduler can decide to preempt the currently running process (if its arrival time is later than the former). Similarly, for SJF and

---

Amazon, Microsoft, HP, and IBM are at the forefront of providing such services. Cloud computing is the industry buzzword for providing such services.

priority, the scheduler re-evaluates and takes a decision to preempt the currently running process whenever a new process joins the ready queue or an existing one rejoins the queue upon I/O completion.

Shortest Remaining Time First (SRTF) is a special case of the SJF scheduler with preemption added in. The scheduler has an estimate of the running time of each process. When a process rejoins the ready queue, the scheduler computes the remaining processing time of the job. Based on this calculation, the scheduler places this process at the right spot in the ready queue. If the remaining time of this process is lower than that of the currently running process, then the scheduler preempts the latter in favor of the former.

Example 4:

Consider the following four processes vying for the CPU. The scheduler uses **SRTF**. The table shows the arrival time of each process.

Process	Arrival Time	Execution Time
P1	T_0	4ms
P2	T_0+1ms	2ms
P3	T_0+2ms	2ms
P4	T_0+3ms	3ms

a) Show the schedule starting from time T_0.

Answer:

1) At T_0, P1 starts running since there is no other process.
2) At T_0+1, P2 arrives. The following table shows the remaining/required running time for P1 and P2:

Process	Remaining time
P1	3ms
P2	2ms

Scheduler switches to P2.

3) At T_0+2, P3 arrives. The following table shows the remaining/required running time for P1, P2, and P3:

Process	Remaining time
P1	3ms
P2	1ms
P3	2ms

Scheduler continues with P2

4) At T_0+3, P4 arrives. P2 completes and leaves. The following table shows the remaining/required running for P1, P3, and P4:

Process	Remaining time
P1	3ms
P3	2ms
P4	3ms

Scheduler picks P3 as the winner and runs it to completion for the next 2 ms.

5) At T0+5, P1 and P4 are the only two remaining processes with remaining/required times as follows:

Process	Remaining time
P1	3ms
P4	3ms

It is a tie, and the scheduler breaks the tie by scheduling P1 and P4 in the arrival order (P1 first and then P4), which would result in a lower average wait time.

The following table shows the schedule with SRTF.

Interval T_0+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Running	**P1**	**P2**	**P2**	**P3**	**P3**	**P1**	**P1**	**P1**	**P4**	**P4**	**P4**		

b) What is the waiting time for each process?

Answer:

Response time = completion time – arrival time
R_{p1} = 8-0 = 8 ms
R_{p2} = 3-1 = 2 ms
R_{p3} = 5-2 = 3 ms
R_{p4} = 11-3 = 8ms

Wait time = response time – execution time
$W_{p1} = R_{p1} – E_{p1}$ = 8-4 = **4 ms**
$W_{p2} = R_{p2} – E_{p2}$ = 2-2 = **0 ms**
$W_{p3} = R_{p3} – E_{p3}$ = 3-2 = **1 ms**
$W_{p4} = R_{p4} – E_{p4}$ = 8-3 = **5 ms**

c) What is the average wait time with SRTF?

Answer:

Total waiting time = $W_{p1}+W_{p2}+W_{p3}+W_{p4}$ = **10 ms**
Average wait time = 10/4 = 2.5 ms

d) What is the schedule for the same set of processes with FCFS scheduling policy?

Answer:

Interval T_0+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Running	**P1**	**P1**	**P1**	**P1**	**P2**	**P2**	**P3**	**P3**	**P4**	**P4**	**P4**		

e) What is the average wait time with FCFS?

Answer:

Response time = completion time – arrival time
R_{p1} = 4-0 = 4 ms
R_{p2} = 6-1 = 5 ms
R_{p3} = 8-2 = 6 ms
R_{p4} = 11-3 = 8ms

Wait time = response time – execution time
$W_{p1} = R_{p1} – E_{p1}$ = 4-4 = **0 ms**
$W_{p2} = R_{p2} – E_{p2}$ = 5-2 = **3 ms**
$W_{p3} = R_{p3} – E_{p3}$ = 6-2 = **4 ms**
$W_{p4} = R_{p4} – E_{p4}$ = 8-3 = **5 ms**

Total waiting time = $W_{p1}+W_{p2}+W_{p3}+W_{p4}$ = **12 ms**
Average wait time = 12/4 = 3 ms

Note: You can see that SRTF gives a lower average wait time compared to FCFS.

6.7.1 Round Robin Scheduler

Let us discern the characteristics of a scheduler that is appropriate for time-shared environments. As the name implies, in this environment, every process should get its share of the processor. Therefore, a non-preemptive scheduler is inappropriate for such an environment.

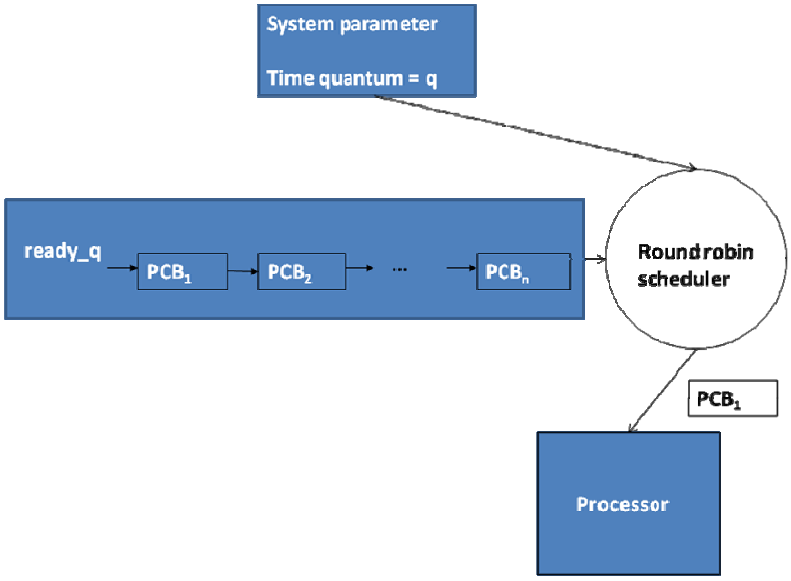


Figure 6.15: Round Robin Scheduler

Time-shared environments are particularly well served by a round robin scheduler. Assume that there are *n* ready processes. The scheduler assigns the processor in time quantum units, *q*, usually referred to as *timeslice,* to each process (Figure 6.15). You can

see the connection between this type of scheduler and the first strategy we suggested in the analogy at the beginning of the chapter for test preparation. Switching among these ready processes in a round robin scheduler does not come for free. The *Context switching time* has to be taken into account in determining a feasible value for the time quantum q.

Each process in the ready queue get its share of the processor for timeslice q. Once the timeslice is up, the currently scheduled process is placed at the end of the ready queue, and the next process in the ready queue is scheduled on the processor.

We can draw a connection between the round robin scheduler and the FCFS scheduler. FCFS is a special case of the round robin scheduler wherein the time quantum is *infinity*. *Processor sharing* is another special case of round robin wherein each process gets 1 unit of time on the processor. Thus, each process gets the illusion of running on a dedicated processor with a speed of *1/n*. The processor-sharing concept is useful for proving theoretical results in scheduling.

6.7.1.1 An Example

Let us consider an example of round robin scheduling. Given in Figure 6.16 are three processes with CPU and I/O bursts as noted. Let us assume that the order of arrival of the processes is P1, P2, and P3; and that the scheduler uses a timeslice of 2. Each process exits the system once its CPU and I/O bursts as shown in the figure are complete.

	CPU	I/O	CPU	I/O	
P1	4	2	2		**P1 is done**
P2	3	2	2		**P2 is done**
P3	2	2	4	2	**P3 is done**

Figure 6.16: CPU and I/O bursts for Round robin example

At time t =0, the scheduling queue looks as shown in Figure 6.17. The key point to note is that this order at time t = 0 may not be preserved as the processes go back and forth between the CPU and I/O queues. When a process rejoins the CPU or the I/O queue it is always at the end of the queue since there is no inherent priority associated with any of the processes.

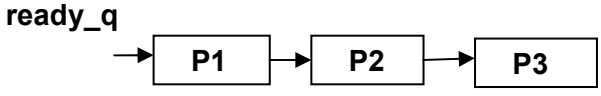


Figure 6.17: Ready Queue for example in Figure 6.16 at time t = 0

Figure 6.18 show a snapshot of the schedule for the first 17 time units (t = 0 to t = 16). Note that P2 uses only one 1 unit of its timeslice the second time it is scheduled (at t = 6), since it makes an I/O burst after 3 units of CPU burst. In other words, in a round robin schedule, the timeslice is an upper bound for continuous CPU usage before a new scheduling decision. Also, at time t = 12 P2 has just completed its I/O burst, and is ready to rejoin the CPU queue. At this time, P1 is in the middle of its CPU timeslice and P3 is on the ready queue as well. Therefore, P2 joins the ready queue behind P3 as shown in Figure 6.19.

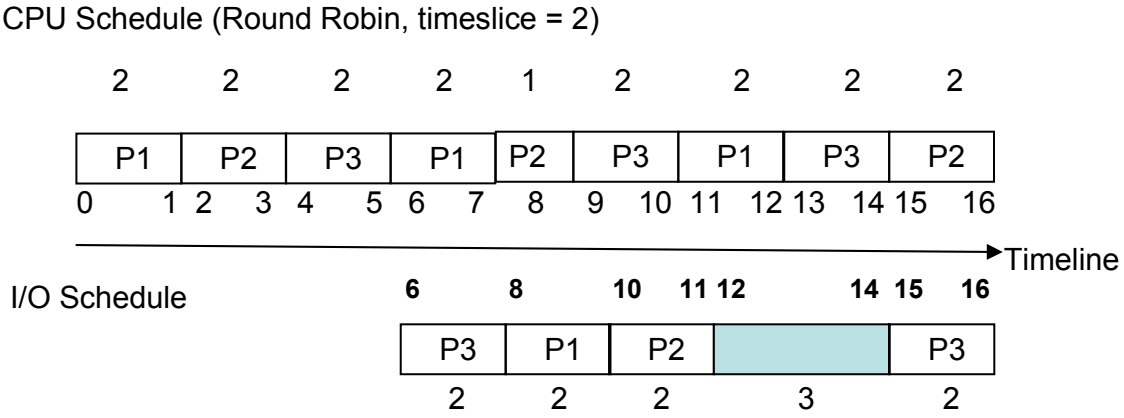


Figure 6.18: Round robin schedule for example in Figure 6.16

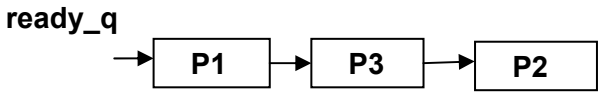


Figure 6.19: Ready Queue for example in Figure 6.16 at time t = 12

Example 5:

What is the wait time for each of the three processes in Figure 6.16 with round robin schedule?

Answer:

As in examples 1 and 3,
Wait time =
(response time – useful work done either on the CPU or I/O)
Wait time for P1 = (13 – 8) = 5
Wait time for P2 = (17 – 7) = 10
Wait time for P3 = (17 – 10) = 7

6.7.1.2 Details of round robin algorithm

Let us turn our attention to getting control of the processor for the scheduler. A hardware device, *timer*, interrupts the processor when the time quantum *q* expires. The interrupt handler for the timer interrupt is an integral part of the round robin scheduler. Figure 6.20 shows the organization of the different layers of the system. At any point of time, the processor is hosting either one of the user programs or the scheduler. Consider that a user program is currently running on the processor. Upon an interrupt, the timer interrupt handler takes control of the processor (refer to the sequence of steps taken by an interrupt in Chapter 4). The handler saves the context of the currently running user program in the associated PCB and hands over control to the scheduler. This handoff, referred to as an *upcall*, allows the scheduler to run its algorithm to decide the next process to be dispatched on the processor. In general, upcall refers to invoking a system function (i.e. a procedure call) from a lower level of the system software to the upper levels.

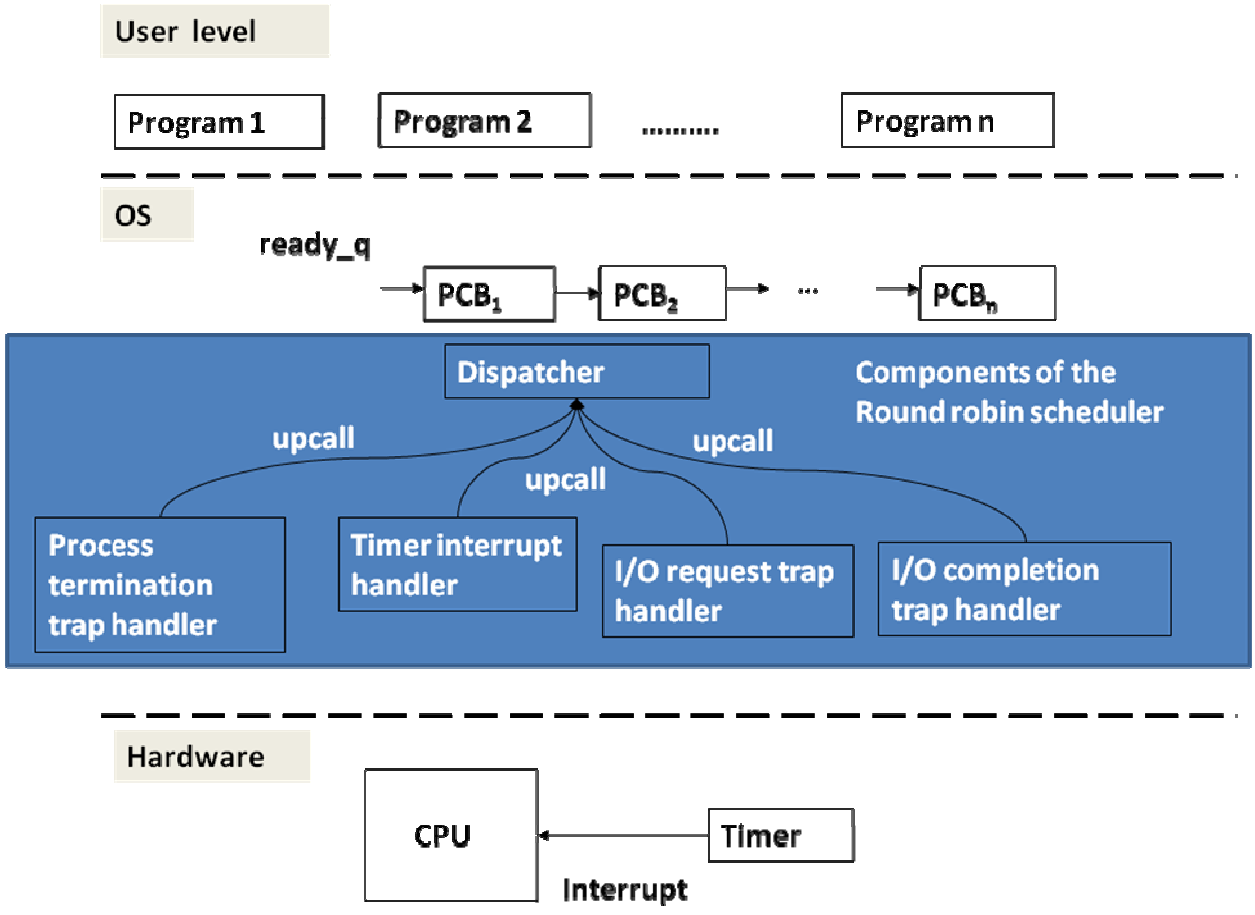


Figure 6.20: Different layers of the system incorporating a round-robin scheduler

Figure 6.21 summarizes the round robin scheduling algorithm. The algorithm comprises of five procedures: *dispatcher, timer interrupt handler, I/O request trap, I/O completion interrupt handler,* and *process termination trap handler*.

```
Dispatcher:
        get head of ready queue;
        set timer;
        dispatch;

Timer interrupt handler:
        save context in PCB;
        move PCB to the end of the ready queue;
        upcall to dispatcher;

I/O request trap:
        save context in PCB;
        move PCB to I/O queue;
        upcall to dispatcher;

I/O completion interrupt handler:
        save context in PCB;
        move PCB of I/O completed process to ready queue;
        upcall to dispatcher;

Process termination trap handler:
        Free PCB;
        upcall to dispatcher;
```

Figure 6.21: Round Robin Scheduling Algorithm

The dispatcher simply dispatches the process at the head of the ready queue on the processor setting the timer to the time quantum q. The currently scheduled process may give up the processor in one of three-ways: it makes an I/O call, or it terminates, or it completes its assigned time quantum on the processor. I/O request trap is the manifestation of the first option, for example, if the currently scheduled process makes a file read request from the disk. In this case, the I/O request trap procedure saves the state of the currently scheduled process in the PCB, moves it to the I/O queue, and upcalls the dispatcher. Timer interrupt handler is the manifestation of the time quantum for the current process expiring. The hardware timer interrupts the processor resulting in the invocation of this handler. As can be seen in Figure 6.21, the timer interrupt handler saves the state in the PCB of the current process, moves the PCB to the end of the ready queue, and makes an upcall to the dispatcher. Upon completion of an I/O request, the processor will get an I/O completion interrupt. The interrupt will result in the invocation of the I/O completion handler. The role of this handler is tricky. Currently some process is executing on the processor. This process's time quantum is not up since it is still executing on the processor. The handler simply saves the state of the currently running process in the corresponding PCB. The I/O completion is on behalf of whichever process requested it in the first place. The handler moves the associated PCB of the process that requested the I/O to the end of the ready queue and upcalls the dispatcher. The dispatcher will do the needful to dispatch the original process that was interrupted by the I/O completion interrupt (since it still has time on the processor). The process

termination handler simply frees up the PCB of the process, removing it from the ready queue and upcalls the dispatcher.

6.8 Combining Priority and Preemption

General-purpose operating systems such as Unix and Microsoft Windows XP combine notions of priority and preemption in the CPU scheduling algorithms. Processes have priorities associated with them determined at the time of creation by the operating system. They are organized into a multi-level ready queue similar to Figure 6.14. Additionally, the operating system employs a round robin scheduling with a fixed time quantum for each process at a particular priority level. Lower priority processes get service when there are no higher priority processes to be serviced. To avoid starvation, the operating system may periodically boost the priority of processes that have not received service for a long time (i.e., aging). Further, an operating system may also give larger time quanta for higher priority processes. The software system for such a scheduler has a structure depicted in Figure 6.20.

6.9 Meta Schedulers

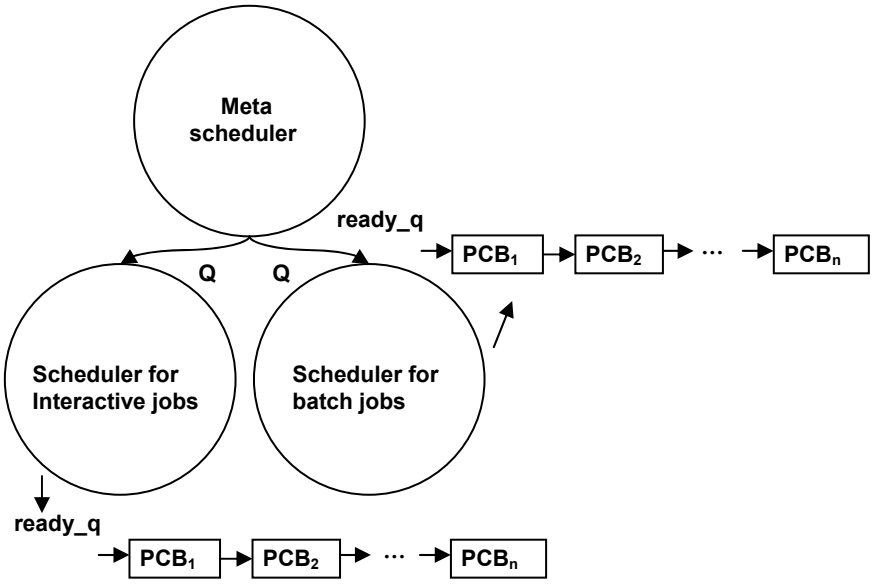


Figure 6.22: Meta Scheduler

In some operating systems, catering to multiple demands, the scheduling environment may consist of several ready queues, each having its own scheduling algorithm. For example, there may be a queue for foreground interactive jobs (with an associated scheduler) and a queue for background batch-oriented jobs (with an associated scheduler). The interactive jobs are scheduled in a round-robin fashion, while the background jobs are scheduled using a priority based FCFS scheduler. A meta-scheduler sits above these schedulers timeslicing among the queues (Figure 6.22). The parameter Q is the time quantum allocated by the meta-scheduler to each of the two schedulers at next

level. Often the meta-scheduler provides for movement of jobs between the ready queues of the two lower level schedulers due to the dynamic changes in the application (represented by the different processes) requirements.

A generalization of this meta-scheduler idea is seen in *Grid Computing*, a somewhat nomadic Internet-based computing infrastructure that is gaining traction due to the ability it offers for using *high-performance* computational resources without actually owning them. Grid computing harnesses computational resources that may be geographically distributed and crisscrossing administrative boundaries. The term Grid arises from the fact that the arrangement of these computational resources is analogous to the "Power Grid" distributing electricity. Users of the Grid submit jobs to this environment that are executed using the distributed computational resources. The term high-performance refers to the fact that there may be hundreds or even thousands of computational resources working in concert to cater to the computational demands of the application. For example, an application for such an infrastructure may be global climate modeling. Computing environments designed for grid computing employ such meta-schedulers. The interested reader is referred to advanced books on this topic[3].

6.10 Evaluation

Let us understand how we evaluate the efficiency of a scheduler. The key property of any OS entity is to quickly provide the requested resources to the user and get out of the way. Scheduling algorithms are evaluated with respect to the performance metrics we mentioned earlier in the context of the specific environments in which they are to be deployed. We characterized different environments by attributes such as timeshared, batch-oriented, and multiprogrammed. These environments cater to different market forces or application domains for computing. For example, today we can identify the following domains in which computing plays a pivotal role:

- **Desktop**: We are all quite familiar with this personal computing domain.
- **Servers**: This is the domain of mail servers, files servers, and web servers.
- **Business**: This domain encompasses e-commerce and financial applications, and often used synonymously with **enterprise** computing.
- **High-Performance Computing (HPC)**: This is the domain requiring high-performance computational resources for solving scientific and engineering problems.
- **Grid**: This domain has all the elements of HPC with the added wrinkle that the computers may be geographically distributed (for example, some machines may be in Tokyo and some in Bangalore) and may overlap administrative boundaries (for example, some machines may belong to Georgia Tech and some to MIT).
- **Embedded**: This is emerging as the most dominant computing domain with the ubiquity of *personal digital assistants* (PDAs) including cellphones, iPODs, iPhones, and the like. This domain also encompasses dedicated computing systems found in automobiles, aircrafts, and space exploration.

[3] Foster and Kesselman, "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufmann publishers.

- **Pervasive**: This emerging domain combines elements of HPC and embedded computing. Video-based surveillance in airports employing camera networks is an example of this domain.

We use the term *workload* to signify the nature of the applications that typify a particular domain. We can broadly classify workload as *I/O bound* or *Compute bound*. However, the reader should note that this broad classification might not always be useful, especially in the case of I/O. This is because the nature of I/O can be vastly different depending on the application domain. For example, one could say that both business applications and desktop computing are I/O intensive. Business applications manipulate large databases and hence involve a significant amount of I/O; desktop computing is also I/O bound but differs from the business applications in that it is interactive I/O as opposed to I/O involving mass storage devices (such as disks). Applications in the server, HPC, and grid domains tend to be computationally intensive. Embedded and pervasive computing domains are quite different from the above domains. Such domains employ sensors and actuators (such as cameras, microphones, temperature sensors, and alarms) that need rapid response similar to the interactive workloads; at the same time, analysis of sensor data (for example, camera images) tends to be computationally intensive.

Table 6.5 summarizes the characteristics of these different domains.

Domains	Environment	Workload characteristics	Types of schedulers
Desktop	Timeshared, interactive, multiprogrammed	I/O bound	Medium-term, short-term, dispatcher
Servers	Timeshared, multiprogrammed	Computation bound	Medium-term, short-term, dispatcher
Business	Timeshared, multiprogrammed	I/O bound	Medium-term, short-term, dispatcher
HPC	Timeshared, multiprogrammed	Computation bound	Medium-term, short-term, dispatcher
Grid	Batch-oriented, timeshared, multiprogrammed	Computation bound	Long-term, Medium-term, short-term, dispatcher
Embedded	Timeshared, interactive, multiprogrammed	I/O bounds	Medium-term, short-term, dispatcher
Pervasive	Timeshared, interactive, multiprogrammed	Combination of I/O bound and computation bound	Medium-term, short-term, dispatcher

Table 6.5: Different Application Domains

One can take one of three different approaches to evaluate a scheduler: *modeling, simulation,* and *implementation*. Each of these approaches has its *pros* and *cons*. Modeling refers to deriving a mathematical model (using techniques such as queuing theory) of the system. Simulation refers to developing a computer program that simulates

the behavior of the system. Lastly, implementation is the actual deployment of the algorithm in the operating system. Modeling, simulation, and implementation represent increasing amounts of effort in that order. Correspondingly, there are payoffs from these approaches (in terms of understanding the performance potential of the system) commensurate with the effort. At the same time, modeling and simulation offer the ability to ask "what if" questions, while the implementation freezes a particular algorithm making it difficult to experiment with different design choices. In general, early estimates of system performance are obtained using modeling and simulation before committing the design to an actual implementation.

Name	Property	Scheduling criterion	Pros	Cons
FCFS	Intrinsically non-preemptive; could accommodate preemption at time of I/O completion events	Arrival time (intrinsic property)	Fair; no starvation;	high variance in response time; convoy effect
SJF	Intrinsically non-preemptive; could accommodate preemption at time of new job arrival and/or I/O completion events	Expected execution time of jobs (intrinsic property)	Preference for short jobs; provably optimal for response time; low variance in response times	Potential for starvation; bias against long running computations
Priority	Could be either non-preemptive or preemptive	Priority assigned to jobs (extrinsic property)	Highly flexible since priority is not an intrinsic property, its assignment to jobs could be chosen commensurate with the needs of the scheduling environment	Potential for starvation
SRTF	Similar to SJF but uses preemption	Expected remaining execution time of jobs	Similar to SJF	Similar to SJF
Round robin	Preemptive allowing equal share of the processor for all jobs	Time quantum	Equal opportunity for all jobs;	Overhead for context switching among jobs

Table 6.6: Comparison of Scheduling Algorithms

6.11 Summary and a Look ahead

Table 6.6 summarizes the characteristics of the different scheduling algorithms discussed in this chapter. There are several advanced topics in scheduling for the interested reader. The strategies discussed in this chapter do not guarantee any specific quality of service. Several environments may demand such guarantees. Consider, for example a system that controls a nuclear reactor; or a rocket launcher; or the control system on board an aircraft. Such systems, dubbed *real-time* systems, need deterministic guarantees; advanced topics in scheduling deal with providing such real-time guarantees. For example, *deadline scheduling*, provides hints to the scheduler on the absolute time by which a task has to be completed. The scheduler will use these deadlines as a way of deciding process priorities for scheduling purposes.

As it turns out, many modern applications also demand such real-time guarantees. You play music on your iPOD, play a game on your XBOX, or watch a movie on your laptop. These applications need real-time guarantees as well. However, the impact of missing deadlines may not be as serious as in a nuclear reactor. Such applications, often called *soft real-time* applications, have become part of the applications that run on general-purpose computers. Scheduling for such applications is an interesting topic for the reader to explore as well.

Embedded computing is emerging as a dominant environment that is making quite a bit of inroads, exemplified by cellphones, iPODs, and iPhones. The interested reader should explore scheduling issues in such environments.

Moving beyond uniprocessor environments, scheduling in a multiprocessors environment brings its own set of challenges. We defer discussion of this topic to a later chapter in the book. Lastly, scheduling in distributed systems is another exciting topic, which is outside the scope of this textbook.

6.12 Linux Scheduler – A case study

As we mentioned already, modern general-purpose operating systems use a combination of the techniques presented in this chapter. As a concrete example, let us review how scheduling works in Linux.

Linux is an "open source" operating system project, which means that a community of developers has voluntarily contributed to the overall development of the operating system. New releases of the operating system appear with certain regularity. For example, circa December 2007, the release of the kernel has the number 2.6.13.12. The discussion in this section applies to the scheduling framework found in Linux release 2.6.x.

Linux offers an interesting case study since it is at once trying to please two different environments: (a) desktop computing and (b) servers. Desktop computing suggests an interactive environment wherein response time is important. This suggests that the

scheduler may have to make frequent context switches to meet the interactive needs (mouse clicks, keyboard input, etc.). Servers, on the other hand, handle computationally intensive workload; thus, less the context switching, more the useful work done in a server. Thus, Linux sets out to meet the following goals in its scheduling algorithm:

- High efficiency, meaning spending as little time as possible in the scheduler itself, an important goal for the server environment
- Support for interactivity, which is important for the interactive workload of the desktop environment
- Avoiding starvation, to ensure that computational workload do not suffer as a result of interactive workloads
- Support for soft real-time scheduling, once again to meet the demands of interactive applications with real-time constraints

Linux's connotation for the terms process and threads are a bit non-standard from the traditional meanings associated with these terms. Therefore, we will simply use the term task as the unit of scheduling in describing the Linux scheduling algorithm[4].

Linux scheduler recognizes three classes of tasks:

- Real-time FCFS
- Real-time round robin
- Timeshared

The scheduler has 140 priority levels. It reserves levels 0-99 for real-time tasks, and the remaining levels for the timeshared tasks. Lower number implies higher priority, thus priority level 0 is the highest priority level in the system. The scheduler uses real-time FCFS and real-time round-robin classes for interactive workloads, and the timeshared class for computational workloads. Real-time FCFS tasks enjoy the highest priority. The scheduler will not preempt a real-time FCFS task currently executing on the processor unless a new real-time FCFS task with a higher priority joins the ready queue. Real-time round robin tasks have lower priority compared to the real-time FCFS tasks. As the name suggests, the scheduler associates a time quantum with each round robin task. All the round robin tasks at a given priority level have the same time quantum associated with them; the higher the priority level of a round robin task the higher the time quantum associated with it. The timeshared tasks are similar to the real-time tasks except that they are at lower priority levels.

The main data structure (Figure 6.23) in the scheduler is a *runqueue*. The runqueue contains two *priority* arrays. One priority array is the active one and the second is the expired one. Each priority array has 140 entries corresponding to the 140 priority levels. Each entry, points to the first task at that level. The tasks at the same level are linked together through a doubly-linked list.

[4] Please see: http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf if you are interested in reading more about the details of Linux CPU scheduler.

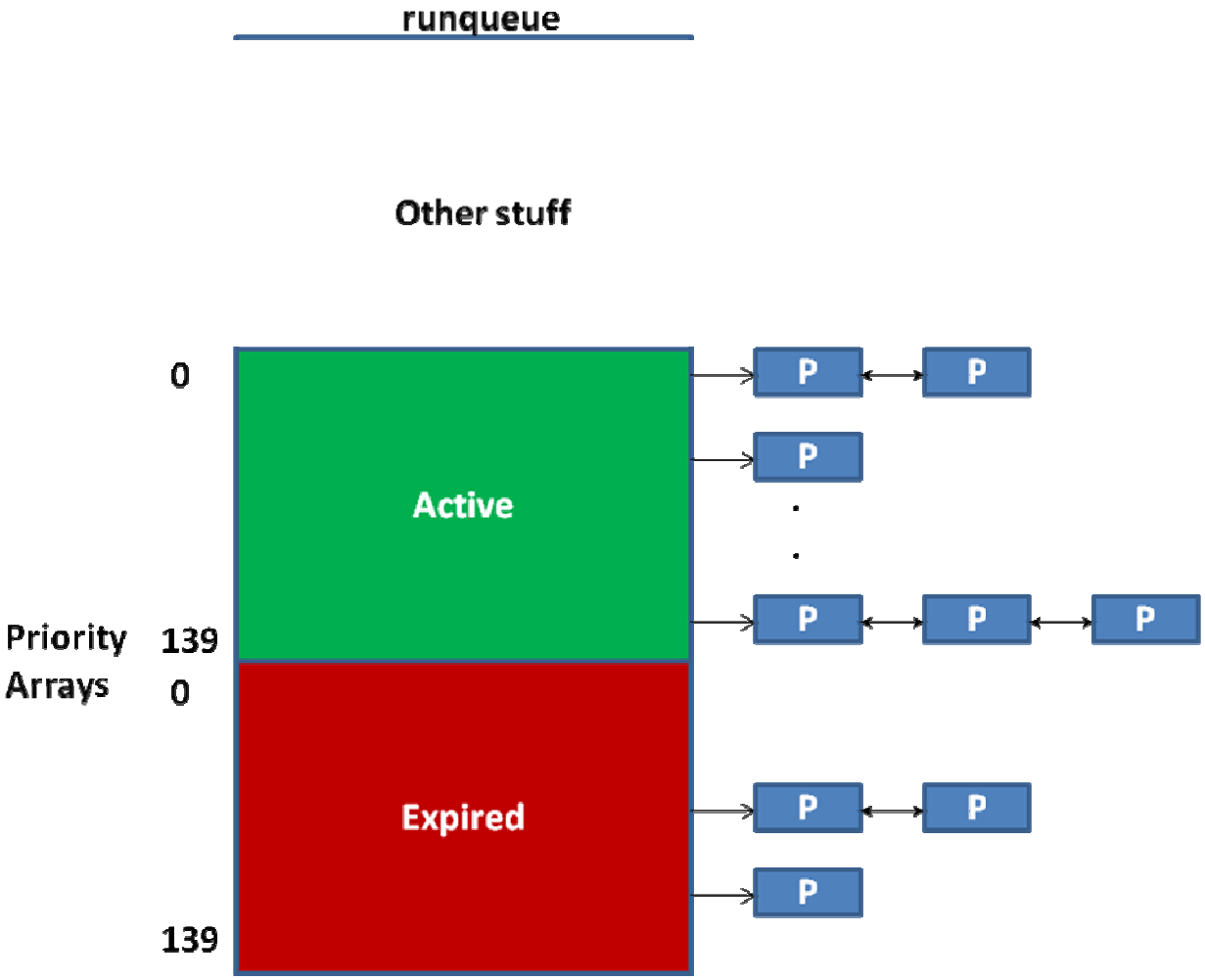


Figure 6.23: Linux Scheduling Data Structures

The scheduling algorithm is straightforward:

- Pick the first task with the highest priority from the active array and run it.
- If the task blocks (due to I/O) put it aside and pick the next highest one to run.
- If the time quantum runs out (does not apply to FCFS tasks) for the currently scheduled task then place it in the expired array.
- If a task completes its I/O then place it in the active array at the right priority level adjusting its remaining time quantum.
- If there are no more tasks to schedule in the active array, simply flip the active and expired array pointers and continue with the scheduling algorithm (i.e., the expired array becomes the active array and vice versa).

The first thing to note with the above algorithm is that the priority arrays enable the scheduler to take a scheduling decision in a *constant* amount of time independent of the number of tasks in the system. This meets the efficiency goal we mentioned earlier. For

this reason, the scheduler is also called O(1) scheduler implying that the scheduling decision is independent of the number of tasks in the system.

The second thing to note is that the scheduler has preferential treatment for meeting soft real-time constraints of interactive tasks through the real-time FCFS and real-time round robin scheduling classes.

As we said earlier, but for the relative priority levels, there is not much difference in terms of scheduling between the real-time round robin and timeshared classes of tasks. In reality, the scheduler does not know which of these tasks are truly interactive. Therefore, it uses a heuristic to determine the nature of the tasks from execution history. The scheduler monitors the pattern of CPU usage of each task. If a task makes frequent blocking I/O calls, it is an interactive task (I/O bound); on the other hand, if a task does not do much I/O, it is a CPU intensive task (CPU bound).

Many us may be familiar with the phrase "carrot and stick." It is an idiom used to signify rewarding good behavior and punishing bad behavior. The scheduler does the same, rewarding tasks that are interactive by boosting their scheduling priority dynamically; and punishing tasks that are CPU intensive by lowering their scheduling priority dynamically[5]. The scheduler boosts the priority level of an interactive task so that it will get a higher share of the CPU time thus leading to good response times. Similarly, the scheduler lowers the priority of a compute bound task so that it will get a lesser share of the CPU time compared to the interactive tasks.

Lastly, for meeting the starvation goal, the scheduler has a starvation threshold. If a task has been deprived of CPU usage (due to interactive tasks getting higher priority) beyond this threshold then the starved task gets CPU usage ahead of the interactive tasks.

6.13 Historical Perspective

Operating systems have a colorful history just as CPUs. Microsoft Windows, MacOS, and Linux dominate the market place today. Let us take a journey through the history books to see how we got here.

Evolution of the operating system is inexorably tied to the evolution of the processor. Charles Babbage (1792-1871) built the first calculating machine (he called it the "analytical engine") out of purely mechanical parts: gears, pulleys, and wheels. Not much happened in the evolution of computing until World War II. It is sad but true that wars spur technological innovations. William Mauchley and Presper Eckert at University of Pennsylvania built ENIAC (Electronic Numerical Integrator and Calculator) in 1944 using vacuum tubes, funded by the U.S. Army to carry out the calculations needed to break the German secret codes. ENIAC and other early machines of this era (1944 to 1955) were primarily used in standalone mode and did not need any operating system. In

[5] A task has a static priority at the time of task creation. Dynamic priority is a temporary deviation from this static level as either a reward or a punishment for good or bad behavior, respectively.

fact, “programs” in these early days of computing were just wired up circuits to perform specific computations, repetitively.

The appearance of mainframes in the late 50’s using solid-state electronics gave birth to the first images of computing machinery, as we know it today. IBM was the main player in the mainframe arena, and introduced the FORTRAN programming language and possibly the first operating system to support it called FMS (Fortran Monitoring System) and later IBSYS, IBM’s operating system for IBM 7094 computer. This is the age of the *batch-oriented* operating system; users submitted their jobs as a deck of punched cards. The punched cards contained the resource needs of the program in a language called the *job control language (JCL)* and the program itself in FORTRAN. The jobs were run one at a time on the computer requiring quite a bit of manual work by the human operator to load the tapes and disks relevant to the resource requirements of each program. Users collected the results of the run of their program on the computer at a later time.

Two distinct user communities were discernible in these early days of computing: scientific and business. In the mid 60’s, IBM introduced the 360 family of computers aimed at integrating the needs of both these communities under one umbrella. It also introduced OS/360 as the operating system for this family of machines. The most important innovation in this operating system is the introduction of *multiprogramming* that ensured that the CPU was being used for some other process when the system was performing I/O for some users. Despite multiprogramming, the external appearance of the system was still a batch-oriented one, since the users submitted their jobs at one time and collected the results later.

The desire to get interactive response to the submitted jobs from the point of view of analyzing the results for further refinement of the programs and/or for simply debugging led to the next evolution in operating systems, namely, *timesharing*, which had its roots in CTSS (Compatible Time Sharing System) developed at MIT in 1962 to run on IBM 7094. A follow on project at MIT to CTSS was MULTICS (MULTiplexed Information and Computing Service), which was perhaps way ahead of its time in terms of concepts for information service and exchange, given the primitive nature of the computing base available at that time. It introduced several seminal operating system ideas related to information organization, sharing, protection, security, and privacy that are relevant to this day.

MULTICS was the seed for the development of the UNIX operating system by Dennis Ritchie and Ken Thompson in Bell Labs in 1974. UNIX[6] (the name comes from the operating system’s creators to develop a stripped down one-user version of MULTICS) became instantly popular with educational institutions, government labs, and many companies such as DEC and HP. The emergence of Linux obliquely out of UNIX is an interesting story. With the popularity of UNIX and its adoption by widely varying entities, soon there were several incompatible versions of the operating system. In 1987,

[6] It was originally named UNICS (UNIplexed Information and Computing Service) and the name later changed to UNIX.

Andrew Tannenbaum[7] of Vrije University developed MINIX as a small clone of UNIX for educational purposes. Linus Torvalds wrote Linux (starting from MINIX) as a free production version of UNIX and soon it took a life of its own due the open software model promoted by the GNU foundation. Despite the interesting history behind the evolution of these different UNIX-based systems, one thing remained the same, namely, the operating systems concepts embodied in these different flavors of UNICES.

All of the above operating systems evolved to support mainframes and minicomputers. In parallel with this evolution, microcomputers were hitting the market and slowly transforming both the computing base as well as the usage model. Computers were no longer necessarily a shared resource, the model suggested by the expensive mainframes and minicomputers. Rather it is a *Personal Computer (PC)* intended for the exclusive use of a single user. There were a number of initial offerings of microcomputers built using Intel 8080/8085 single-chip processors in the 70's. What should the operating system look like for such a personal computer? A simple operating system called CP/M (an acronym for *Control Program Monitor*) was an industry standard for such microcomputers.

IBM developed the IBM PC in the early 1980s. A small company called Microsoft offered to IBM an operating system called MS-DOS for the IBM PC. Early versions of MS-DOS had striking similarities to CP/M, but had a simpler file system and delivered higher performance. With the blessings of an industry giant such as IBM, soon MS-DOS took over the PC market, and sidelined CP/M. Initially, MS-DOS was quite primitive in its functionality given the intended use of the PC, but slowly it started incorporating the ideas of multitasking and timesharing from the UNIX operating system. It is interesting to note that while MS-DOS evolved from a small lean operating system and started incorporating ideas from UNIX operating system, Apple's Mac OS X, also intended for PCs (Apple's Macintosh line), was a direct descendent of UNIX operating system.

When Apple adopted the GUI (Graphical User Interface) in Macintosh personal computer, Microsoft followed suit, offering Windows on top of MS-DOS as a way for users to interact with the PC. Initial offerings of Windows were wrappers on top of MS-DOS until 1995, when Microsoft introduced Windows 95, which integrated the Windows GUI with the operating system. We have gone through several iterations of Windows operating systems since Windows 95 including Windows NT (NT stands for New Technology), Windows NT 4.0, Windows 2000, Windows XP, and Windows Vista (as of January 2007). However, the basic concepts at the level of the core abstractions in the operating system have not changed much in these iterations. If anything, these concepts have matured to the point where they are indistinguishable from those found in the UNIX operating system. Similarly, UNIX-based systems have also absorbed and incorporated the GUI ideas that had their roots in the PC world.

[7] Author of many famous textbooks in operating systems and architecture.

6.14 Review Questions

1. Compare and contrast process and program.
2. What items are considered to comprise the state of a process?
3. Which metric is the most user centric in a timesharing environment?
4. Consider a pre-emptive priority processor scheduler. There are three processes P1, P2, and P3 in the job mix that have the following characteristics:

Process	Arrival Time	Priority	Activity
P1	0 sec	1	8 sec CPU burst followed by 4 sec I/O burst followed by 6 sec CPU burst and quit
P2	2 sec	3	64 sec CPU burst and quit
P3	4 sec	2	2 sec CPU burst followed by 2 sec I/O burst followed by 2 sec CPU burst followed by 2 sec I/O burst followed by 2 sec CPU burst followed by 2 sec I/O burst followed by 2 sec CPU burst and quit

   What is the turnaround time for each of P1, P2, and P3?
   What is the average waiting time for this job mix?

5. What are the important deficiencies of FCFS CPU scheduling?

6. Name a scheduling algorithm where starvation is impossible?

7. Which scheduling algorithm was noted as having a high variance in turnaround time?

8. Which scheduling algorithm is provably optimal (minimum average waiting time)?

9. Which scheduling algorithm must be preemptive?

10. Given the following processes that arrived in the order shown

	CPU Burst Time	IO Burst Time
P1	3	2
P2	4	3
P3	8	4

    Show the activity in the processor and the I/O area using the FCFS, SJF and Round Robin algorithms.

11. Redo Example 1 in Section 6.6 using SJF and round robin (timeslice = 2)

12. Redo Example 3 in Section 6.6 using FCFS and round robin (timeslice = 2)