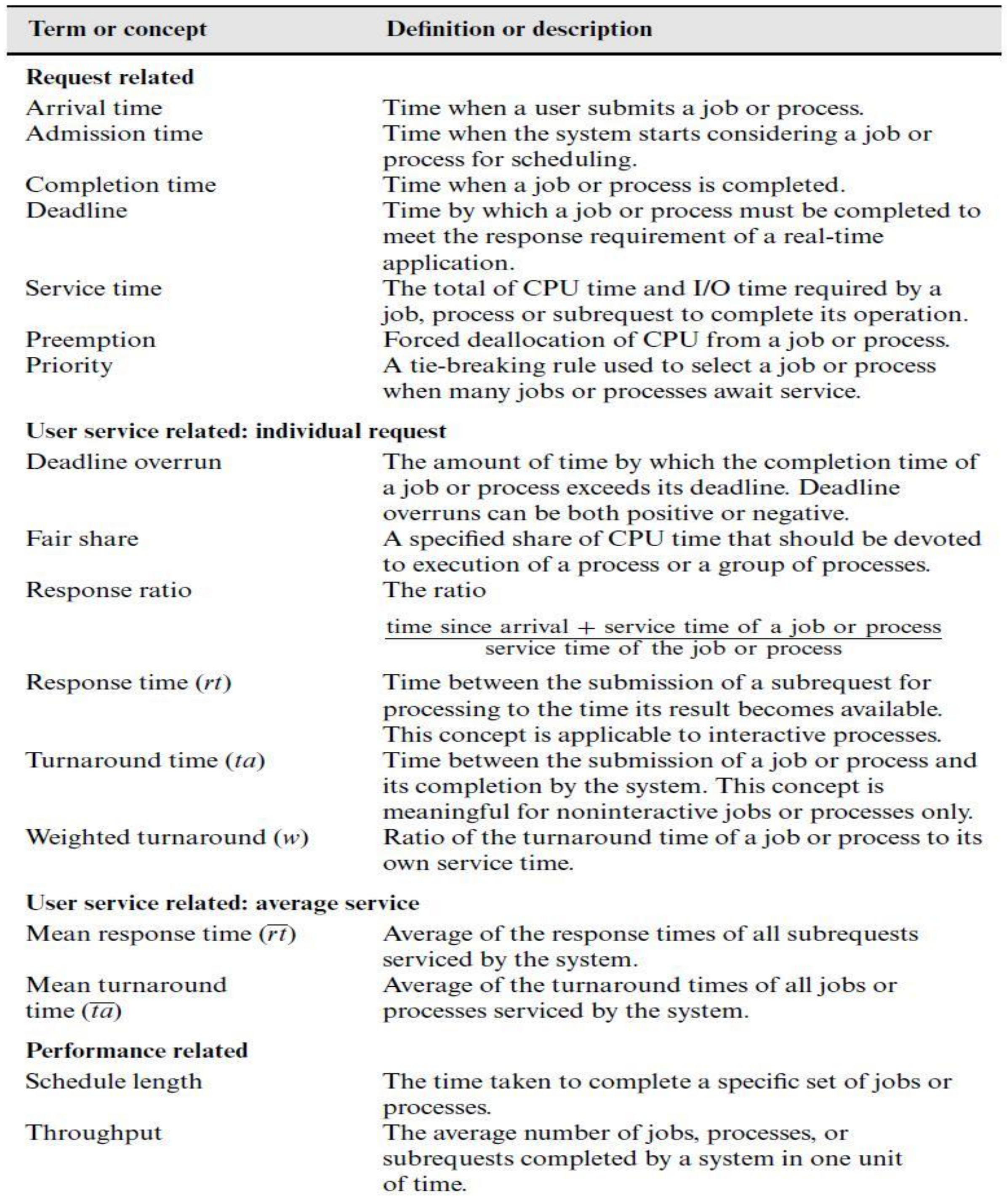
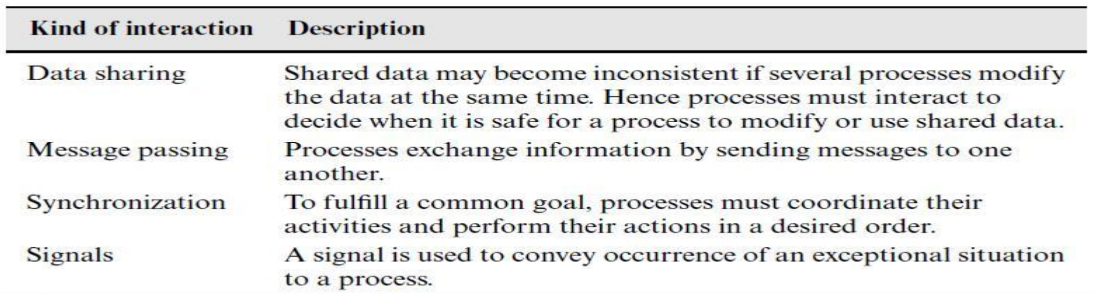
2nd internals Solution

1 (a) List and explain the scheduling concepts which are used to analyze the performance of any scheduling algorithm.



Term or concept	Definition or description
Request related	
Arrival time	Time when a user submits a job or process.
Admission time	Time when the system starts considering a job or process for scheduling.
Completion time	Time when a job or process is completed.
Deadline	Time by which a job or process must be completed to meet the response requirement of a real-time application.
Service time	The total of CPU time and I/O time required by a job, process or subrequest to complete its operation.
Preemption	Forced deallocation of CPU from a job or process.
Priority	A tie-breaking rule used to select a job or process when many jobs or processes await service.
User service related: individual request	
Deadline overrun	The amount of time by which the completion time of a job or process exceeds its deadline. Deadline overruns can be both positive or negative.
Fair share	A specified share of CPU time that should be devoted to execution of a process or a group of processes.
Response ratio	The ratio $\frac{\text{time since arrival} + \text{service time of a job or process}}{\text{service time of the job or process}}$
Response time (*rt*)	Time between the submission of a subrequest for processing to the time its result becomes available. This concept is applicable to interactive processes.
Turnaround time (*ta*)	Time between the submission of a job or process and its completion by the system. This concept is meaningful for noninteractive jobs or processes only.
Weighted turnaround (*w*)	Ratio of the turnaround time of a job or process to its own service time.
User service related: average service	
Mean response time ($\overline{rt}$)	Average of the response times of all subrequests serviced by the system.
Mean turnaround time ($\overline{ta}$)	Average of the turnaround times of all jobs or processes serviced by the system.
Performance related	
Schedule length	The time taken to complete a specific set of jobs or processes.
Throughput	The average number of jobs, processes, or subrequests completed by a system in one unit of time.

b) List the different types of process interaction and explain them in brief.



Kind of interaction	Description
Data sharing	Shared data may become inconsistent if several processes modify the data at the same time. Hence processes must interact to decide when it is safe for a process to modify or use shared data.
Message passing	Processes exchange information by sending messages to one another.
Synchronization	To fulfill a common goal, processes must coordinate their activities and perform their actions in a desired order.
Signals	A signal is used to convey occurrence of an exceptional situation to a process.

2 (a) Explain process scheduling methods for real time applications.

Real-Time Scheduling

Real-time scheduling must handle two special scheduling constraints while trying to meet the deadlines of applications. First, the processes within a real-time application are interacting processes, so the deadline of an application should be translated into appropriate deadlines for the processes. Second, processes may be periodic, so different instances of a process may arrive at fixed intervals and all of them have to meet their deadlines. Example 7.10 illustrates these constraints; in this section, we discuss techniques used to handle them.

Process Precedences and Feasible Schedules

Processes of a real-time application interact among themselves to ensure that they perform their actions in a desired order (see Section 6.1).We make the simplifying assumption that such interaction takes place only at the start or end of a process. It causes dependences between processes, whichmust be taken into account while determining deadlines and while scheduling. We use a *process precedence graph* (PPG) to depict such dependences between processes.

Process Pi is said to *precede* process Pj if execution of Pi must be completed beforePj can begin its execution. The notation$Pi \rightarrow Pj$ shall indicate that process Pi directly precedes process Pj . The precedence relation is transitive; i.e., $Pi \rightarrow Pj$ and$Pj \rightarrow Pk$ implies thatPi precedesPk. The notation$Pi \rightarrow^* Pk$ is used to indicate that process Pi directly or indirectly precedes Pk. A *process precedence graph* is a directed graph $G \equiv (N, E)$ such that $Pi \in N$ represents a process, and an edge $(Pi , Pj) \in E$ implies $Pi \rightarrow Pj$. Thus, a path $Pi , \ldots , Pk$ in PPG implies $Pi \rightarrow^* Pk$. A process Pk is a descendant of Pi if $Pi \rightarrow^* Pk$.

A *hard real-time system* as one that meets the response requirement of a real-time application in a guaranteed manner, even when fault tolerance actions are required. This condition implies that the time required by the OS to complete operation of all processes in the application does not exceed the response requirement of the application. On the other hand, a *soft real-time system* meets the response requirement of an application only in a probabilistic manner, and not necessarily at all times. The notion of a *feasible schedule* helps to differentiate between these situations.

Feasible Schedule A sequence of scheduling decisions that enables the processes of an application to operate in accordance with their precedences and meet the response requirement of the application.

Deadline Scheduling

Two kinds of deadlines can be specified for a process: a *starting deadline*, i.e., the latest instant of time by which operation of the process must begin, and a *completion deadline*, i.e., the time by which operation of the process must complete.

Deadline Estimation A system analyst performs an in-depth analysis of a realtime application and its response requirements. Deadlines for individual processes are determined by considering process precedences and working backward from the response requirement of the application. Accordingly, *Di* , the completion deadline of a process *Pi*, is

$$D_i = D_{\text{application}} - \sum_{k \in descendant(i)} x_k \qquad (7.2)$$

Where $D_{\text{application}}$ is the deadline of the application, x_k is the service time of process *Pk*, and *descendant(i)* is the set of descendants of *Pi* in the PPG, i.e., the set of all processes that lie on some path between *Pi* and the exit node of the PPG. Thus, the deadline for a process *Pi* is such that if it is met, all processes that directly or indirectly depend on *Pi* can also finish by the overall deadline of the application. This method is illustrated in Example 7.11.

Determining Process Deadlines Example 7.11

Each circle is a node of the graph and represents a process. The number in a circle indicates the service time of a process. An edge in the PPG shows a precedence constraint. Thus, process *P*2 can be initiated only after process *P*1 completes, process *P*4 can be initiated only after processes *P*2 and *P*3 complete, etc. We assume that processes do not perform I/O operations and are serviced in a nonpreemptive manner. The total of the service times of the processes is 25 seconds. If the application has to produce a response in 25 seconds, the deadlines of the processes would be as follows:

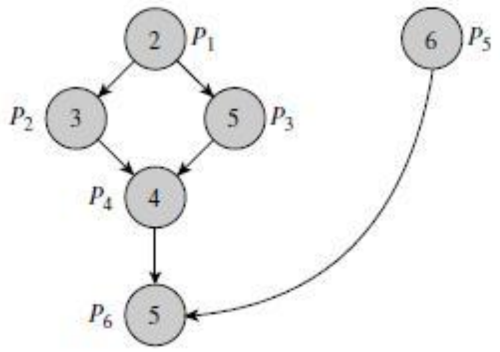


Figure shows the PPG of a real-time application containing 6 processes.

Process	*P*1	*P*2	*P*3	*P*4	*P*5	*P*6
Deadline	8	16	16	20	20	25

A practical method of estimating deadlines will have to incorporate several other constraints as well. For example, processes may perform I/O. If an I/O operation of one process can be overlapped with execution of some independent process, the deadline of its predecessors (and ancestors) in the PPG can be relaxed by the amount of I/O overlap. For example, processes *P*2 and *P*3 in Figure 7.13 are independent of one another. If the service time of *P*2 includes 1 second of I/O time, the deadline of *P*1 can be made 9 seconds instead of 8 seconds if the I/O operation of *P*2 can overlap with *P*3's processing. However, overlapped execution of processes must consider resource availability as well. Hence determination of deadlines is far more complex than described here.

Earliest Deadline First (EDF) Scheduling As its name suggests, this policy always selects the process with the earliest deadline. Consider a set of real-time processes that do not perform I/O operations. If *seq* is the sequence in which processes are serviced by a deadline scheduling policy and *pos*(*Pi*) is the position of process *Pi* in *seq*, a deadline overrun does not occur for process *Pi* only if the sum of its own service time and service times of all processes that precede it in *seq* does not exceed its own deadline, i.e.,

$$\sum_{k:pos(P_k)\leq pos(P_i)} x_k \leq D_i \qquad (7.3)$$

where *xk* is the service time of process *Pk*, and *Di* is the deadline of process *Pi*. If this condition is not satisfied, a deadline overrun will occur for process *Pi*. When a feasible schedule exists, it can be shown that Condition 7.3 holds for all processes; i.e., a deadline overrun will not occur for any process. Table 7.4 illustrates operation of the EDF policy for the deadlines of Example 7.11. The notation *P*4: 20 in the column *processes in system* indicates that process *P*4 has the deadline 20. Processes *P*2, *P*3 and *P*5, *P*6 have identical deadlines, so three schedules other than the one shown in Table 7.4 are possible with EDF scheduling. None of them would incur deadline overruns.

The primary advantages of EDF scheduling are its simplicity and nonpreemptive nature, which reduces the scheduling overhead. EDF scheduling is a good policy for static scheduling because existence of a feasible schedule, which can be checked *a priori*, ensures that deadline overruns do not occur. It is also a good dynamic scheduling policy for use in soft real-time system; however, the number of processes that miss their deadlines is unpredictable. The next example illustrates this aspect of EDF scheduling.

Operation of Earliest Deadline First (EDF) Scheduling

Time	Process completed	Deadline overrun	Processes in system	Process scheduled
0	–	0	$P_1 : 8, P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_1
2	P_1	0	$P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_2
5	P_2	0	$P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$	P_3
10	P_3	0	$P_4 : 20, P_5 : 20, P_6 : 25$	P_4
14	P_4	0	$P_5 : 20, P_6 : 25$	P_5
20	P_5	0	$P_6 : 25$	P_6
25	P_2	0	–	–

Rate Monotonic Scheduling

When processes in an application are periodic, the existence of a feasible schedule can be determined in an interesting way. Consider three independent processes that do not perform I/O operations:

Process	P_1	P_2	P_3
Time period (ms)	10	15	30
Service time (ms)	3	5	9

Process *P*1 repeats every 10 ms and needs 3 ms of CPU time. So the fraction of the CPU‘s time that it uses is 3⁄10, i.e., 0.30. The fractions of CPU time used by *P*2 and *P*3 are analogously 5⁄15 and 9⁄30, i.e., 0.33 and 0.30. They add up to 0.93, so if the CPU overhead of OS operation is negligible, it is feasible to service these three processes. In general, a set of periodic processes *P*1, . . . , *Pn* that do not perform I/O operations can be serviced by a hard real-time system that has a negligible overhead if

$$\Sigma_{i=1...n} \frac{x_i}{T_i} \leq 1 \qquad (7.4)$$

where *Ti* is the period of *Pi* and *xi* is its service time.

To schedule these processes so that they can all operate without missing their deadlines. The *rate monotonic* (RM) scheduling policy does it as follows: It determines the *rate* at which a process has to repeat, i.e., the number of repetitions per second, and assigns the rate itself as the priority of the process. It now employs a priority-based scheduling technique to perform scheduling. This way, a process with a smaller period has a higher priority, which would enable it to complete its operation early.

In the above example, priorities of processes *P*1, *P*2, and*P*3 would be 1⁄0.010, 1⁄0.015, and 1⁄0.025, i.e., 100, 67, and 45, respectively. Figure 7.14 shows how these processes would operate. Process *P*1 would be scheduled first. It would execute once and become dormant after 3 ms, because *x*1 = 3 ms. Now *P*2 would be scheduled and would complete after 5 ms. *P*3 would be scheduled now, but it would be preempted after 2 ms because *P*1 becomes *ready* for the second time, and so on. As shown in Figure 7.14, process *P*3 would complete at 28 ms. By this time, *P*1 has executed three times and *P*2 has executed two times.

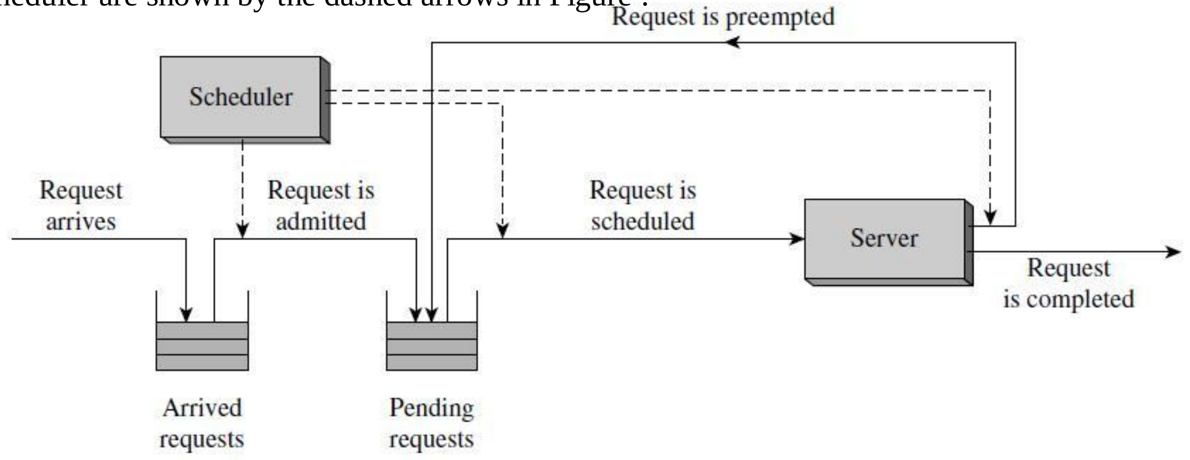
Rate monotonic scheduling is not guaranteed to find a feasible schedule in all situations. For example, if process *P*3 had a time period of 27 ms, its priority would be different; however, relative priorities of the processes would be unchanged, so *P*3 would complete at 28 ms as before, thereby suffering a deadline overrun of 1ms. A feasible schedule would have been obtained if *P*3 had been scheduled at 20 ms and *P*1 at 25 ms; however, it is not possible under RM scheduling because processes are scheduled in a priority-based manner. Liu and Layland (1973) have shown that RM scheduling may not be able to avoid deadline overruns if the total fraction of CPU time used by the processes according to Eq. (7.4) exceeds $m(21/m - 1)$, where *m* is the number of processes. This expression has a lower bound of 0.69, which implies that if an application has a large number of processes, RM scheduling may not be able to achieve more than 69 percent CPU utilization if it is to meet deadlines of processes.

**3 (a)	Write a program to illustrate the use of pthreads in the real time data logging application.**

```
#include <pthread.h>
 #include <stdio.h>
int size, buffer[100], no_of_samples_in_buffer; int main()
{
pthread_t id1, id2, id3;
pthread_mutex_t buf_mutex, condition_mutex; pthread_cond_t buf_full,
buf_empty; pthread_create(&id1, NULL, move_to_buffer, NULL);
pthread_create(&id2, NULL, write_into_file, NULL);
pthread_create(&id3, NULL, analysis, NULL); pthread_join(id1, NULL);
pthread_join(id2,    NULL);    pthread_join(id3,
NULL); pthread_exit(0);
}
void *move_to_buffer()
{
/* Repeat until all samples are received */ /* If no space in
buffer, wait on buf_full */
/* Use buf_mutex to access the buffer, increment no. of samples */ /* Signal
buf_empty */
pthread_exit(0);
}
void *write_into_file()
{
/* Repeat until all samples are written into the file */ /* If no data in
buffer, wait on buf_empty */
/* Use buf_mutex to access the buffer, decrement no. of samples */ /* Signal buf_full
*/
pthread_exit(0);
}
void *analysis()
{
/* Repeat until all samples are analyzed */
/* Read a sample from the buffer and analyze it */ pthread_exit(0);
}
```

(b) Explain process schedule with neat diagram.

Scheduling, very generally, is the activity of selecting the next request to be serviced by a *server*. Figure below is a schematic diagram of scheduling. The scheduler actively considers a list of pending requests for servicing and selects one of them. The server services the request selected by the scheduler. This request leaves the server either when it completes or when the scheduler preempts it and puts it back into the list of pending requests. In either situation, the scheduler selects the request that should be serviced next. From time to time, the scheduler admits one of the arrived requests for active consideration and enters it into the list of pending requests. Actions of the scheduler are shown by the dashed arrows in Figure .



Events related to a request are its *arrival, admission, scheduling, preemption*, and *completion*

4 (a) With a neat diagram explain the working and functions of long, medium and short term scheduling in a time sharing system.

Long-, Medium-, and Short-Term Schedulers

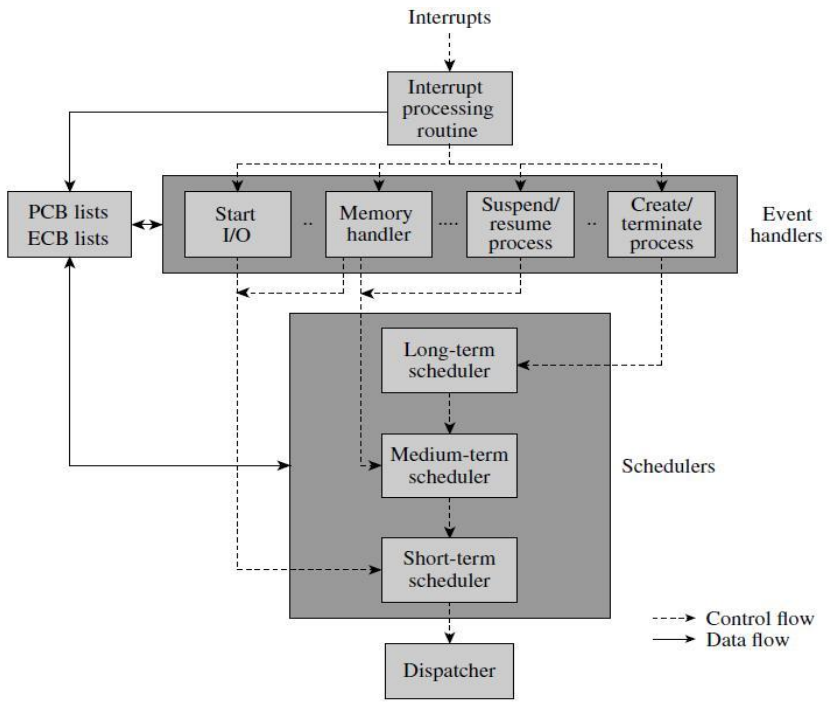
These schedulers perform the following functions:

• *Long-term scheduler:* Decides when to admit an arrived process for scheduling, depending on its nature (whether CPU-bound or I/O-bound) and on availability of resources like kernel data structures and disk space for swapping.

• *Medium-term scheduler:* Decides when to swap-out a process from memory and when to load it back, so that a sufficient number of *ready* processes would exist in memory.

• *Short-term scheduler:* Decides which *ready* process to service next on the CPU and for how long. Thus, the *short-term scheduler* is the one that actually selects a process for operation. Hence it is also called the *process scheduler*, or simply the *scheduler*.

Figure below shows an overview of scheduling and related actions. The operation of the kernel is interrupt-driven. Every event that requires the kernel‘s attention causes an interrupt.



Long-Term Scheduling The long-term scheduler may defer admission of a request for two reasons: it may not be able to allocate sufficient resources like kernel data structures or I/O devices to a request when it arrives, or it may find that admission of a request would affect system performance in some way; e.g., if the system currently contained a large number of CPU-bound requests, the scheduler might defer admission of a new CPU-bound request, but it might admit a new I/O-bound request right away.

Long-term scheduling was used in the 1960s and 1970s for job scheduling because computer systems had limited resources, so a long-term scheduler was required to decide *whether* a process could be initiated at the present time. It continues to be important in operating systems where resources are limited. It is also used in systems where requests have deadlines, or a set of requests are repeated with a known periodicity, to decide *when* a process should be initiated to meet response requirements of applications. Long-term scheduling is not relevant in other operating systems.

Medium-Term Scheduling Medium-term scheduling maps the large number of requests that have been admitted to the system into the smaller number of requests that can fit into the memory of the system at any time. Thus its focus is on making a sufficient number of *ready* processes available to the short-term scheduler by suspending or reactivating processes. The medium term scheduler decides when to swap out a process from memory and when to swap it back into memory, changes the state of the process appropriately, and enters its process control block (PCB) in the appropriate list of PCBs. The actual swapping-in and swapping-out operations are performed by the memory manager.

The kernel can suspend a process when a user requests suspension, when the kernel runs out of free memory, or when it finds that the CPU is not likely to be allocated to the process in the near future. In time-sharing systems, processes in *blocked* or *ready* states are candidates for suspension.

Short-Term Scheduling Short-term scheduling is concerned with effective use of the CPU. It selects one process from a list of *ready* processes and hands it to the dispatching mechanism. It may also decide how long the process should be allowed to use the CPU and instruct the kernel to produce a timer interrupt accordingly.

(b) Explain briefly the fundamental techniques of scheduling.

Fundamental Techniques of Scheduling

Schedulers use three fundamental techniques in their design to provide good user service or high performance of the system:

• *Priority-based scheduling:* The process in operation should be the highest priority process requiring use of the CPU. It is ensured by scheduling the highest-priority *ready* process at any time and preempting it when a process with a higher priority becomes *ready*. multiprogramming OS assigns a high priority to I/O-bound processes; this assignment of priorities provides high throughput of the system.

• *Reordering of requests:* Reordering implies servicing of requests in some order other than their arrival order. Reordering may be used by itself to improve user service, e.g., servicing short requests before long ones reduces the average turnaround time of requests. Reordering of requests is implicit in preemption, which may be used to enhance user service, as in a time-sharing system, or to enhance the system throughput, as in a multiprogramming
system.

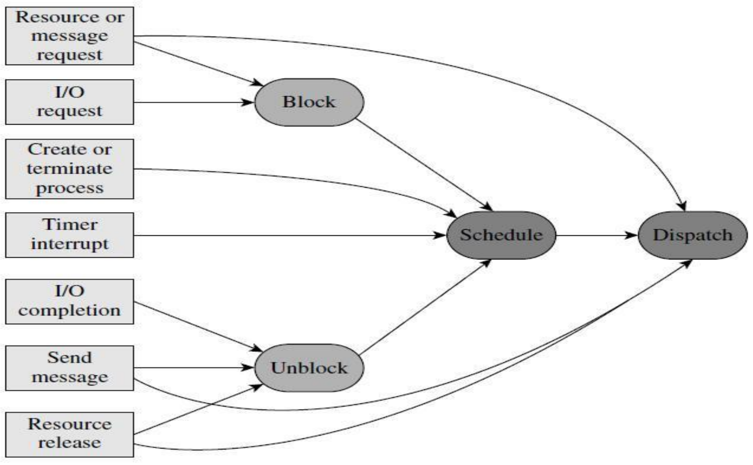
• *Variation of time slice:* When time-slicing is used, $\eta = \delta/(\delta + \sigma)$ where η is the CPU efficiency, δ is the time slice and σ is the OS overhead per scheduling decision. Better response times are obtained when smaller values of the time slice are used; however, it lowers the CPU efficiency because considerable process switching overhead is incurred. To balance CPU efficiency and response times, an OS could use different values of δ for different requests—a small value for I/O-bound requests and a large value for CPU-bound requests—or it could vary the value of δ for a process when its behavior changes from CPU-bound to I/O-bound, or from I/O bound to CPU-bound.

5 (a) List the events occur during the operation of OS. With a neat diagram discuss the event handling actions of kernel.

The following events occur during the operation of an OS:

1. *Process creation event:* A new process is created.
2. *Process termination event:* A process completes its operation.
3. *Timer event:* The timer interrupt occurs.
4. *Resource request event:* Process makes a resource request.
5. *Resource release event:* A process releases a resource.
6. *I/O initiation request event:* Process wishes to initiate an I/O operation.
7. *I/O completion event:* An I/O operation completes.
8. *Message send event:* A message is sent by one process to another.
9. *Message receive event:* A message is received by a process.
10. *Signal send event:* A signal is sent by one process to another.
11. *Signal receive event:* A signal is received by a process.
12. *A program interrupt:* The current instruction in the *running* process malfunctions.
13. *A hardware malfunction event:* A unit in the computer‘s hardware malfunctions.

The timer, I/O completion, and hardware malfunction events are caused by situations that are external to the running process. All other events are caused by actions in the *running* process. The kernel performs a standard action like aborting the *running* process when events 12 or 13 occur.



Event handling actions of the kernel.

When a process releases a resource, an *unblock* action is performed if some other process is waiting for the released resource, followed by scheduling and dispatching because the unblocked process may have a higher priority than the process that released the resource. Again, scheduling is skipped if no process is unblocked because of the event.

(b) Explain with a neat diagram, the different states and transitions of process in UNIX Operating system.

A process in the *running* state is put in the *ready* state the moment its execution is interrupted. A system process then handles the event that caused the interrupt. If the running process had itself caused a software interrupt by executing an <*SI_instrn*>, its state may further change to *blocked* if its request cannot be granted immediately. In this model a user process executes only user code; it does not need any special privileges. A system process may have to use privileged instructions like I/O initiation and setting of memory protection information, so the system process executes with the CPU in the kernel mode. Processes behave differently in the Unix model. When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call. To ensure that it has the necessary privileges, it needs to execute with the CPU in the kernel mode. A mode change is thus necessary every time a system call is made. The opposite mode change is necessary after processing a system call. Similar mode changes are needed when a process starts executing the interrupt servicing code in the kernel because of an interrupt, and when it returns after servicing an interrupt.

The Unix kernel code is made reentrant so that many processes can execute it concurrently. This feature takes care of the situation where a process gets blocked while executing kernel code, e.g., when it makes a system call to initiate an I/O operation, or makes a request that cannot be granted immediately. To ensure reentrancy of code, every process executing the kernel code must use its own kernel stack. This stack contains the history of function invocations since the time the process entered the kernel code. If another process also enters the kernel code, the history of its function invocations will be maintained on its own kernel stack. Thus, their operation would not interfere. In principle, the kernel stack of a process need not be distinct from its user stack; however, distinct stacks are used in practice because most computer architectures use different stacks when the CPU is in the kernel and user modes. Unix uses two distinct running states. These states are called user running and kernel running states. A user process executes user code while in the user running state, and kernel code while in the kernel running state. It makes the transition from user running to kernel *running* when it makes a system call, or when an interrupt occurs. It may get blocked while in the *kernel running* state because of an I/O operation or non availability of a resource.

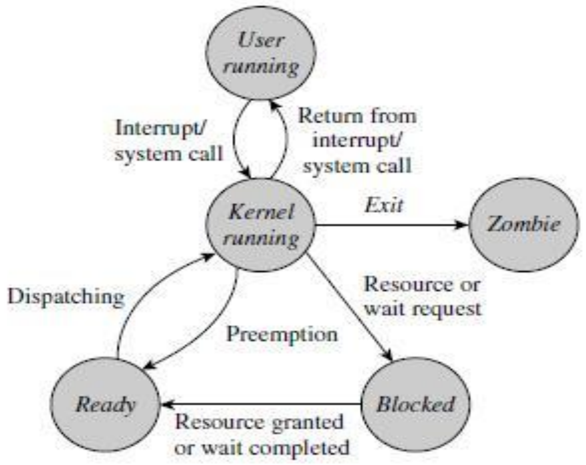
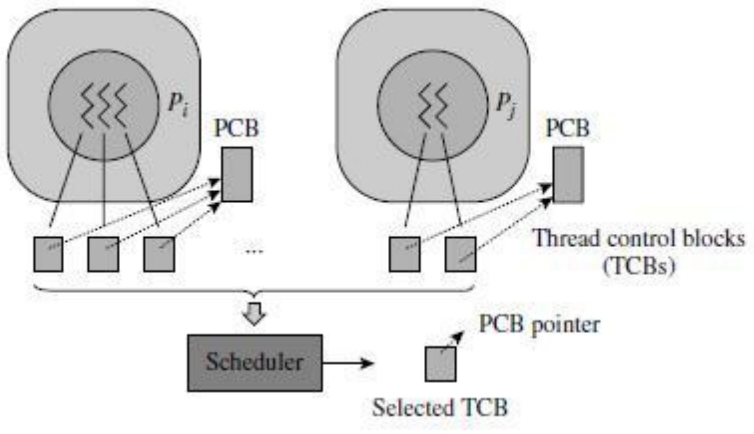


Figure Process state transitions in Unix.

6 (a) Explain briefly Kernel level, user level and hybrid threads, specifying advantages and disadvantages.

Kernel-Level Threads

A kernel-level thread is implemented by the kernel. Hence creation and termination of kernel-level threads, and checking of their status, is performed through system calls. Figure 3.14 shows a schematic of how the kernel handles kernel-level threads. When a process makes a *create_thread* system call, the kernel creates a thread, assigns an id to it, and allocates a thread control block (TCB). The TCB contains a pointer to the PCB of the parent process of the thread.



TCB to check whether the selected thread belongs to a different process than the interrupted thread. If so, it saves the context of the process to which the interrupted thread belongs, and loads the context of the process to which the selected thread belongs. It then dispatches the selected thread. However, actions to save and load the process context are skipped if both threads belong to the same process. This feature reduces the switching overhead, hence switching between kernel-level threads of a process could be as much as an order of magnitude faster, i.e., 10 times faster, than switching between processes.

Advantages and Disadvantages of Kernel-Level Threads

A kernel-level thread is like a process except that it has a smaller amount of state information. This similarity is convenient for programmers—programming for threads is no different from programming for processes. In a multiprocessor system, kernel-level threads provide parallelism, as many threads belonging to a process can be scheduled simultaneously, which is not possible with the user-level threads described in the next section, so it provides better computation speedup than user-level threads.

However, handling threads like processes has its disadvantages too. Switching between threads is performed by the kernel as a result of event handling. Hence it incurs the overhead of event handling even if the interrupted thread and the selected thread belong to the same process. This feature limits the savings in the thread switching overhead.

User-Level Threads

User-level threads are implemented by a *thread library*, which is linked to the code of

a process. The library sets up the thread implementation arrangement shown in Figure 5.11(b) without involving the kernel, and itself interleaves operation of threads in the process. Thus, the kernel is not aware of presence of user-level threads in a process; it sees only the process. Most OSs implement the pthreads application program interface provided in the IEEE POSIX standard in this manner.

Scheduling of User-Level Threads

Figure below is a schematic diagram of scheduling of user-level threads. The thread library code is a part of each process. It performs ―scheduling‖ to select a thread, and organizes its execution. We view this operation as ―mapping‖ of the TCB of the selected thread into the
PCB of the process.

The thread library uses information in the TCBs to decide which thread should operate at any time. To ―dispatch‖ the thread, the CPU state of the thread should become the CPU state of the process, and the process stack pointer should point to the thread‗s stack. Since the thread library is a part of a process, the CPU is in the user mode. Hence a thread cannot be dispatched by loading new information into the PSW; the thread library has to use nonprivileged instructions to change PSW contents. Accordingly, it loads the address of the thread‗s stack into the stack address register, obtains the address contained in the *program counter* (PC) field of the thread‗s CPU state found in its TCB, and executes a branch instruction to transfer control to the instruction which has this address.

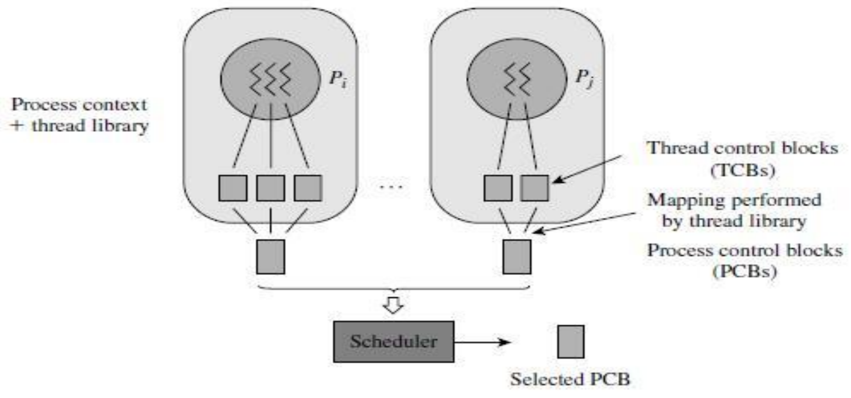

Process context
+ thread library
P_i
P_j
Thread control blocks (TCBs)
Mapping performed by thread library
Process control blocks (PCBs)
Scheduler
Selected PCB

Figure Scheduling of user-level threads.

Advantages and Disadvantages of User-Level Threads

Thread synchronization and scheduling is implemented by the thread library. This arrangement avoids the overhead of a system call for synchronization between threads, so the thread switching overhead could be as much as an order of magnitude smaller than in kernel-level threads.

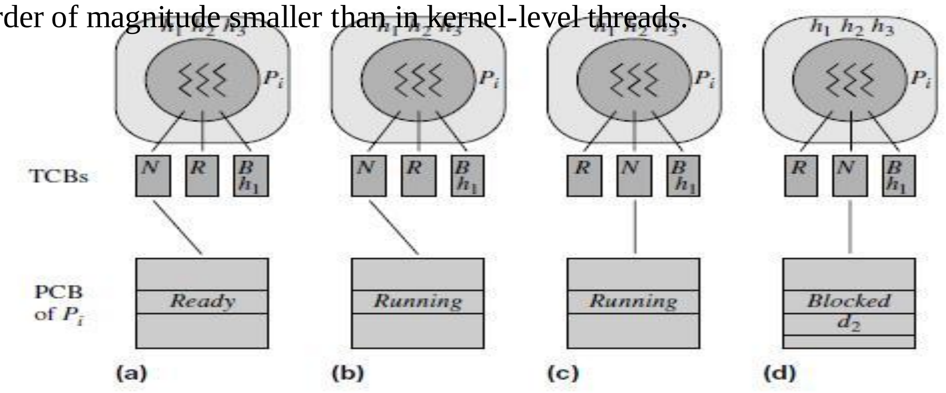


Figure Actions of the thread library (*N*,*R*,*B* indicate *running*, *ready*, and *blocked*).

This arrangement also enables each process to use a scheduling policy that best suits its nature. A process implementing a real-time application may use priority-based scheduling of its threads to meet its response requirements, whereas a process implementing a multithreaded server may perform round-robin scheduling of its threads. However, performance of an application would depend on whether scheduling of user-level threads performed by the thread library is compatible with scheduling of processes performed by the kernel.

For example, round-robin scheduling in the thread library would be compatible with either round-robin scheduling or priority-based scheduling in the kernel, whereas priority-based scheduling would be compatible only with priority-based scheduling in the kernel.

Hybrid Thread Models

A hybrid thread model has *both* user-level threads and kernel-level threads and a method of associating user-level threads with kernel-level threads. Different methods of associating user- and kernel-level threads provide different combinations of the low switching overhead of user-level threads and the high concurrency and parallelism of kernel-level threads.

Figure illustrates three methods of associating user-level threads with kernel-level threads. The thread library creates user-level threads in a process and associates a *thread control block* (TCB) with each user-level thread. The kernel creates kernel-level threads in a process and associates a *kernel thread control block* (KTCB) with each kernel-level thread. In
the many-to-one association method, a single kernel-level thread is created in a process by the kernel and all userlevel threads created in a process by the thread library are associated with this kernel-level thread. This method of association provides an effect similar to mere user-level threads: User-level threads can be concurrent without being parallel, thread switching incurs low overhead, and blocking of a user-level thread leads to blocking of all threads in the process.

In the one-to-one method of association, each user-level thread is permanently mapped into a kernel-level thread. This association provides an effect similar to mere kernel-level threads: Threads can operate in parallel on different CPUs of a multiprocessor system; however, switching between threads is performed at the kernel level and incurs high overhead. Blocking of a user-level thread does not block other user-level threads of the process because they are mapped into different kernel-level threads.
The many-to-many association method permits a user-level thread to be mapped into different kernel-level threads at different times.

It provides parallelism between user-level threads that are mapped into different kernel-level threads at the same time, and provides low overhead of switching between user-level threads that are scheduled on the same kernel-level thread by the thread library. However, the many-to-many association method requires a complex implementation.

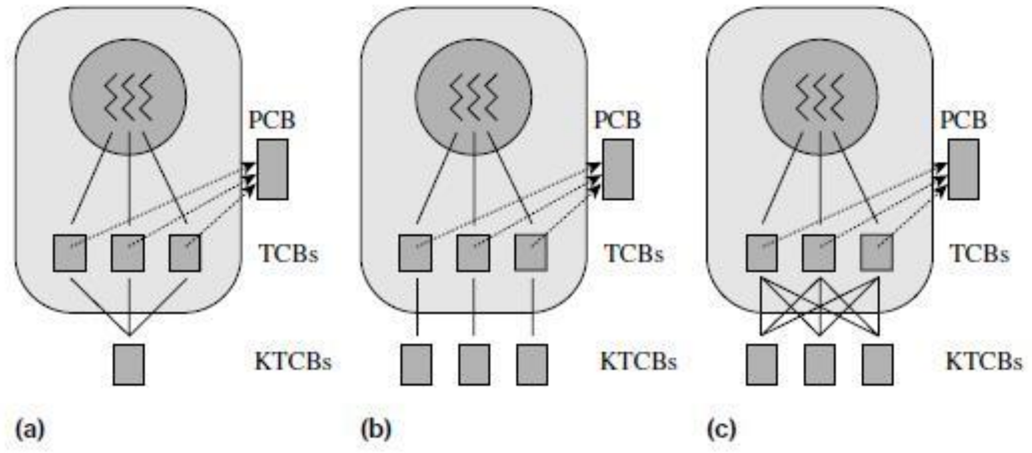


Figure (a) Many-to-one; (b) one-to-one; (c) many-to-many associations in hybrid threads.

(b) Explain race condition with an example.

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, and the outcome depends on the order of execution of processes. But because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Suppose for a moment that two processes need to perform a bit flip at a specific memory location. Under normal circumstances the operation should work like this:

Process 1	**Process 2**	**Memory Value**
Read value		0
Flip value		1
	Read value	1
	Flip value	0

In this example, Process 1 performs a bit flip, changing the memory value from 0 to 1. Process 2 then performs a bit flip and changes the memory value from 1 to 0.
If a race condition occurred causing these two processes to overlap, the sequence could potentially look more like this:

Process 1	**Process 2**	**Memory Value**
Read value		0
	Read value	0
Flip value		1
	Flip value	1

In this example, the bit has an ending value of 1 when its value should be 0. This occurs because Process 2 is unaware that Process 1 is performing a simultaneous bit flip.

7 (a) List out the Non-Preemptive and Preemptive scheduling algorithms along with the advantages and disadvantages for each.

Nonpreemptive Scheduling Policies

In *nonpreemptive scheduling*, a server always services a scheduled request to completion. Thus, scheduling is performed only when servicing of a previously scheduled request is completed and so preemption of a request as shown in Figure 7.1 never occurs. Nonpreemptive scheduling is attractive because of its simplicity—the scheduler does not have to distinguish between an unserviced request and a partially serviced one. Since a request is never preempted, the scheduler‘s only function in improving user service or system performance is reordering of requests. The three nonpreemptive scheduling policies are:

- First-come, first-served (FCFS) scheduling

Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organized as a queue. The scheduler always schedules the first request in the list. An example of FCFS scheduling is a batch processing system in which jobs are ordered according to their arrival times (or arbitrarily, if they arrive at exactly the same time) and results of a job are released to the user immediately on completion of the job.

- Shortest request next (SRN) scheduling

Use of the SRN policy faces several difficulties in practice. Service times of processes are not known to the operating system *a priori*, hence the OS may expect users to provide estimates of service times of processes. However, scheduling performance would be erratic if users do not possess sufficient experience in estimating service times, or they manipulate the system to obtain better service by giving low service time estimates for their processes. The SRN policy offers poor service to long processes, because a steady stream of short processes arriving in the system can starve a long process.

- Highest response ratio next (HRN) scheduling

The response ratio of a newly arrived process is 1. It keeps increasing at the rate *(*1*/*service time*)* as it waits to be serviced. The response ratio of a short process increases more rapidly than that of a long process, so shorter processes are favored for scheduling. However, the response ratio of a long process eventually becomes large enough for the process to get scheduled. This feature provides an effect similar to the technique of *aging*, so long processes do not starve.

Preemptive Scheduling Policies

In *preemptive scheduling*, the server can be switched to the processing of a newrequest before completing the current request. The preempted request is put back into the list of pending requests (see Figure 7.1). Its servicing is resumed when it is scheduled again. Thus, a request might have to be scheduled many times before it completed. This feature causes a larger scheduling overhead than when nonpreemptive scheduling is used.

The three preemptive scheduling policies are:

- Round-robin scheduling with time-slicing (RR)

TheRRpolicy provides comparable service to all CPU-bound processes. This feature is reflected in approximately equal values of their weighted turnarounds. The actual value of the weighted turnaround of a process depends on the number of processes in the system.Weighted turnarounds provided to processes that perform I/O operations would depend on the durations of their I/O operations. The RR policy does not fare well on measures of system performance like throughput because it does not give a favored treatment to short processes.

- Least completed next (LCN) scheduling

The LCN policy schedules the process that has so far consumed the least amount of CPU time. Thus, the nature of a process, whether CPU-bound or I/O-bound, and its CPU time requirement do not influence its progress in the system. Under the LCN policy, all processes will make approximately equal progress in terms of the CPU time consumed by them, so this policy guarantees that short processes will finish ahead of long processes. Ultimately, however, this policy has the familiar drawback of starving long processes of CPU attention. It also neglects existing processes if new processes keep arriving in the system. So even not-so-long processes tend to suffer starvation or large turnaround times.

- Shortest time to go (STG) scheduling

The shortest time to go policy schedules a process whose remaining CPU time requirements are the smallest in the system. It is a preemptive version of the shortest request next (SRN) policy. So it favours short processes over long ones and provides good throughput. Additionally, the STG policy also favours a process that is nearing completion over short processes entering the system. This feature helps to improve the turnaround times and weighted turnarounds of processes. Since it is analogous to the SRN policy, long processes might face starvation.

(b) Explain briefly, scheduling in UNIX.

UNIX is a pure time-sharing operating system. It uses a multilevel adaptive scheduling policy in which process priorities are varied to ensure good system performance and also to provide good user service. Processes are allocated numerical priorities, where a larger numerical value implies a lower effective priority.

In Unix 4.3 BSD, the priorities are in the range 0 to 127. Processes in the user mode have priorities between 50 and 127, while those in the kernel mode have priorities between 0 and 49.When a process is blocked in a system call, its priority is changed to a value in the range 0–49, depending on the cause of blocking.
When it becomes active again, it executes the remainder of the system call with this priority. This arrangement ensures that the process would be scheduled as soon as possible, complete the task it was performing in the kernel mode and release kernel resources. When it exits the kernel mode, its priority reverts to its previous value, which was in the range 50–127.

Unix uses the following formula to vary the priority of a process:

$$\text{Process priority} = \text{base priority for user processes} + f(\text{CPU time used recently}) + \text{nice value} \qquad (7.5)$$

It is implemented as follows: The scheduler maintains the CPU time used by a process in its process table entry. This field is initialized to 0. The real-time clock raises an interrupt 60 times a second, and the clock handler increments the count in the CPU usage field of the running process. The scheduler recomputes process priorities every second in a loop. For each process, it divides the value in the CPU usage field by 2, stores it back, and also uses it as the value of *f*. Recall that a large numerical value implies a lower effective priority, so the second factor in Eq. (7.5) lowers the priority of a process. The division by 2 ensures that the effect of CPU time used by a process *decays*; i.e., it wears off over a period of time, to avoid the problem of starvation faced in the least completed next (LCN) policy.

A process can vary its own priority through the last factor in Eq. (7.5). The system call —*nice*(<*priority value*>);‖ sets the *nice value* of a user process. It takes a zero or positive value as its argument. Thus, a process can only decrease its effective priority to be nice to other processes. It would typically do this when it enters a CPU-bound phase.

Table : **Operation of a Unix-like Scheduling Policy When Processes Perform I/O**

	P_1		P_2		P_3		P_4		P_5		
Time	P	T	P	T	P	T	P	T	P	T	Scheduled process
0.0	60	0									P_1
1.0		60									
	90	30									P_1
2.0		90		0							
	105	45	60	0							P_2
3.0		45		60		0					
	82	22	90	30	60	0					P_3
3.1	82	22	90	30	60	6					P_1
4.0		76		30		6					
	98	38	75	15	63	3					P_3
4.1	98	38	75	15	63	9					P_2
5.0		38		69		9		0			
	79	19	94	34	64	4	60	0			P_4
6.0		19		34		4		60			
	69	9	77	17	62	2	90	30			P_3

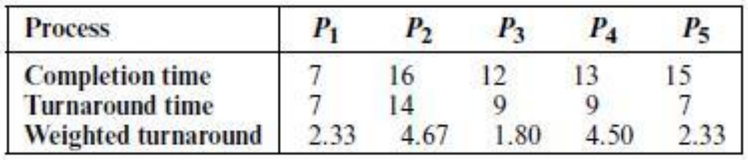
Fair Share Scheduling

To ensure a fair share of CPU time to groups of processes, Unix schedulers add the term *f* (CPU time used by processes in the group) to Eq. (7.5). Thus, priorities of all processes in a group reduce when any of them consumes CPU time. This feature ensures that processes of a group would receive favored treatment if none of them has consumed much CPU time recently. The effect of the new factor also decays over time.

Example 7.14

Table 7.6 depicts fair share scheduling of the processes of Table 7.2. Fields *P*, *T*, and *G* contain process priority, CPU time consumed by a process, and CPU time consumed by a group of processes, respectively. Two process groups exist.

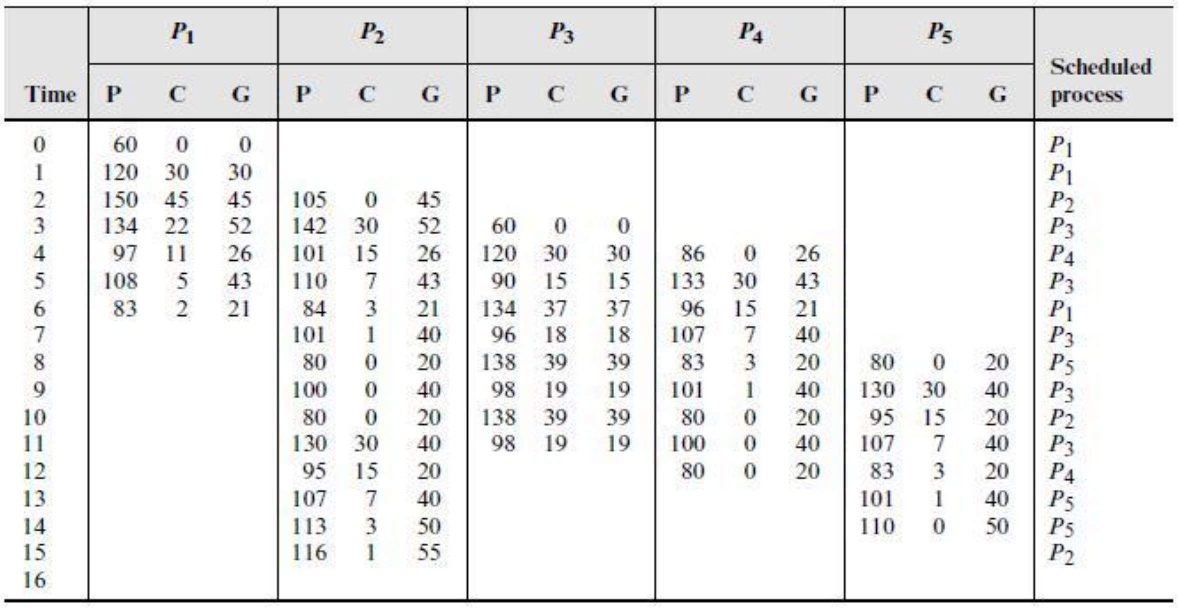
The first group contains processes *P*1, *P*2, *P*4, and *P*5, while the second group contains process *P*3 all by itself. At 2 seconds, process *P*2 has just arrived. Its effective priority is low because process *P*1, which is in the same group, has executed for 2 seconds. However, *P*3 does not have a low priority when it arrives because the CPU time already consumed by its group is 0. As expected, process *P*3 receives a favored treatment compared to other processes. In fact, it receives every alternate time slice. Processes *P*2, *P*4, and *P*5 suffer because they belong to the same process group. These facts are reflected in the turnaround times and weighted turnarounds of the processes, which are as follows:



Process	P_1	P_2	P_3	P_4	P_5
Completion time	7	16	12	13	15
Turnaround time	7	14	9	9	7
Weighted turnaround	2.33	4.67	1.80	4.50	2.33

Mean turnaround time *(ta)* = 9.2 seconds

Mean weighted turnaround *( )* = 3.15



	P_1			P_2			P_3			P_4			P_5			Scheduled process
Time	P	C	G	P	C	G	P	C	G	P	C	G	P	C	G	
0	60	0	0													P_1
1	120	30	30													P_1
2	150	45	45	105	0	45										P_2
3	134	22	52	142	30	52	60	0	0							P_3
4	97	11	26	101	15	26	120	30	30	86	0	26				P_4
5	108	5	43	110	7	43	90	15	15	133	30	43				P_3
6	83	2	21	84	3	21	134	37	37	96	15	21				P_1
7				101	1	40	96	18	18	107	7	40				P_3
8				80	0	20	138	39	39	83	3	20	80	0	20	P_5
9				100	0	40	98	19	19	101	1	40	130	30	40	P_3
10				80	0	20	138	39	39	80	0	20	95	15	20	P_2
11				130	30	40	98	19	19	100	0	40	107	7	40	P_3
12				95	15	20				80	0	20	83	3	20	P_4
13				107	7	40							101	1	40	P_5
14				113	3	50							110	0	50	P_5
15				116	1	55										P_2
16																

2) SHORTEST REQUEST NEXT ALGORITHM : (SRN)

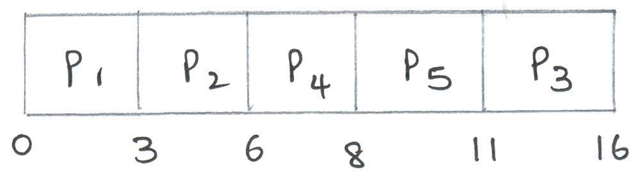
PROCESS	A.T	S.T
P_1	0	3
P_2	2	3
P_3	3	5
P_4	4	2
P_5	8	3

TA = COMPLETION TIME − ARRIVAL TIME

$$W = \frac{\text{TURN-AROUND TIME}}{\text{SERVICE TIME}}$$

TIME	PROCESS COMPLETED			PROCESS IN SYSTEM	PROCESS SCHEDULED
	ID	TA	W		
0	–	–	–	P_1	P_1
3	P_1	3	1	P_2, P_3	P_2
6	P_2	4	1·3	P_3, P_4	P_4
8	P_4	4	2	P_3, P_5	P_5
11	P_5	3	1	P_3	P_3
16	P_3	13	2·6	–	–

GANT CHART:



$$\overline{TA} = \frac{3+4+4+3+13}{5} = \underline{5 \cdot 4 \text{ seconds}}$$

$$\overline{W} = \frac{1+1 \cdot 3+2+1+2 \cdot 6}{5} = \underline{1 \cdot 58 \text{ seconds}}.$$

ROUND-ROBIN ALGORITHM : (RR)

PROCESS	AT	ST
P_1	0	3
P_2	2	3
P_3	3	5
P_4	4	2
P_5	8	3

TIME SLICE $\delta = 1$ Sec

TIME OF SCHEDULING		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
POSITION OF PROCESS IN READY Q 1 ⇒ FRONT OF THE READY Q	P_1	1	1	2	1												
	P_2			1	3	2	1	3	2	1							
	P_3				2	1	3	2	1	4	3	2	1	2	1	2	1
	P_4					3	2	1	3	2	1						
	P_5									3	2	1	2	1	2	1	

PROCESS	COMPLETED PROCESS		
	C	TA	W
P_1	4	4	1.3
P_2	9	7	2.3
P_3	16	13	2.6
P_4	10	6	3
P_5	15	7	2.3

$\overline{TA}$ = 7.4 seconds

$\overline{W}$ = 2.3 seconds.

P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5	P_3

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

5. Reference String : 5 4 3 2 1 4 3 5 4 3 2 1 5

FIFO

5	5	5	2	2	2	3	3	3	1
	4	4	4	1	1	1	5	5	5
		3	3	3	4	4	4	2	2

No of Page faults = 10

5 4 3 2 1 4 3 5 4 3 2 1 5

LRU

5	5	5	2	2	2	3	3	3	3	5
	4	4	4	1	1	1	5	2	2	2
		3	3	3	4	4	4	4	1	1

No of page Faults = 11

OPT 5 4 3 2 1 4 3 5 4 3 2 1 5

5	5	5	2	1	5	5	5
	4	4	4	4	4	2	2
		3	3	3	3	3	1

No of Page Faults = 8