CMR
INSTITUTE OF
TECHNOLOGY

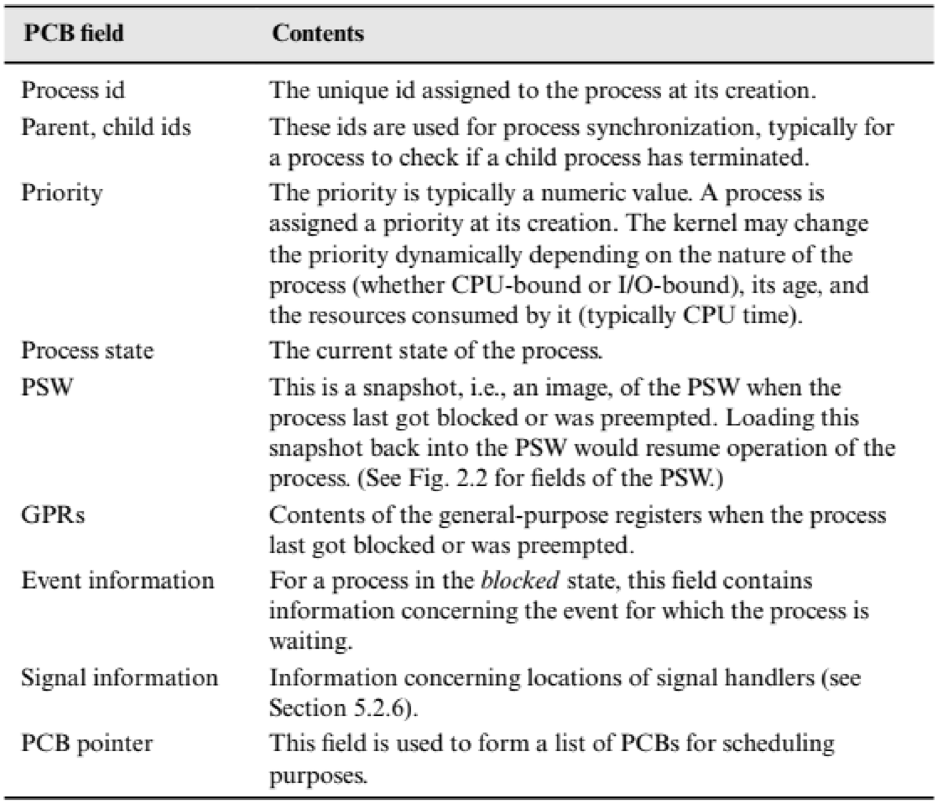
Internal Assesment Test – II JUNE 2022

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Sub: | Operating Systems | | | | | | | Code: | 18EC641 |
| Date: | 09/06/2022 | Duration: | 90 mins | Max Marks: | 50 | Sem: | 6 | Branch: | ECE |

**Answer any five questions out of seven questions**

## Scheme of solutions

| Q. no. | Questions | Marks |
|---|---|---|
| 1. | Discuss Process Context and PCB with respect to Process.<br>The kernel allocates resources to a process and schedules it for use of the CPU.<br>Accordingly, the kernel's view of a process consists of two parts:<br>• Code, data, and stack of the process, and information concerning memory and other resources, such as files, allocated to it.<br>• Information concerning execution of a program, such as the process state, the CPU state including the stack pointer, and some other items of information.<br>These two parts of the kernel's view are contained in the *process context* and the *process control block* (PCB), respectively This arrangement enables different OS modules to access relevant process-related information conveniently and efficiently.<br>Process Context The process context consists of the following:<br>**1.** *Address space of the process:* The code, data, and stack components of the process<br>**2.** *Memory allocation information:* Information concerning memory areas allocated to a process. This information is used by the memory management unit (MMU) during operation of the process<br>**3.** *Status of file processing activities:* Information about files being used, such as current positions in the files.<br>**4.** *Process interaction information:* Information necessary to control interaction ofthe processwith other processes, e.g., ids of parent and child processes, and interprocess messages sent to it that have not yet been delivered to it.<br>**5.** *Resource information:* Information concerning resources allocated to the process.<br>**6.** *Miscellaneous information:* Miscellaneous information needed for operation of a process. The OS creates a process context by allocating memory to the process, loading the process code in the allocated memory and setting up its data space. Information concerning resources allocated to the process and its interaction with other processes is maintained in the process context throughout the life of the process. This information changes as a result of actions like file open and close and creation and destruction of data by the process during its operation. | 5M |
| | **Process Control Block (PCB)** The *process control block* (PCB) of a process contains three kinds of information concerning the process—identification information such as the process id, id of its parent process, and id of the user who created it; process state information such as its state, and the contents of the PSW and the general-purpose registers (GPRs); and information that is useful in controlling its operation, such as its priority, and its interaction with other processes. It also contains a pointer field that is used by the kernel to form PCB lists for scheduling, e.g., a list of *ready* processes. | 5M |



| PCB field | Contents |
|---|---|
| Process id | The unique id assigned to the process at its creation. |
| Parent, child ids | These ids are used for process synchronization, typically for a process to check if a child process has terminated. |
| Priority | The priority is typically a numeric value. A process is assigned a priority at its creation. The kernel may change the priority dynamically depending on the nature of the process (whether CPU-bound or I/O-bound), its age, and the resources consumed by it (typically CPU time). |
| Process state | The current state of the process. |
| PSW | This is a snapshot, i.e., an image, of the PSW when the process last got blocked or was preempted. Loading this snapshot back into the PSW would resume operation of the process. (See Fig. 2.2 for fields of the PSW.) |
| GPRs | Contents of the general-purpose registers when the process last got blocked or was preempted. |
| Event information | For a process in the *blocked* state, this field contains information concerning the event for which the process is waiting. |
| Signal information | Information concerning locations of signal handlers (see Section 5.2.6). |
| PCB pointer | This field is used to form a list of PCBs for scheduling purposes. |

The priority and state information is used by the scheduler. It passes the id of the selected process to the dispatcher. For a process that is not in the *running* state, the *PSW* and *GPRs* fields together contain the *CPU state* of the process when it last got blocked or was. Operation of the process can be resumed by simply loading this information from its PCB into the CPU. This action would be performed when this process is to be dispatched. When a process becomes *blocked*, it is important to remember the reason. It is done by noting the cause of blocking, such as a resource request or an I/O operation, in the *event information* field of the PCB. Consider a process $P_i$ that is blocked on an I/O operation on device *d*. The *event information* field in $P_i$'s PCB indicates that it awaits end of an I/O operation on device *d*. When the I/O operation on device *d* completes, the kernel uses this information to make the transition *blocked* → *ready* for process $P_i$.

2. Define threads? Discuss User-level threads with neat sketch.
Thread is defined as an execution of a program that uses the resources of a process.

2M

**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 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.

8M

An overview of creation and operation of threads is as follows: A process invokes the library function *create_thread* to create a new thread. The library function creates a TCB for the new thread and starts considering the new thread for "scheduling." When the thread in the *running* state invokes a library function to perform synchronization, say, wait until a specific event occurs, the library function performs "scheduling" and switches to another thread of the process. Thus, the kernel is oblivious to switching between threads; it believes that the *process* is continuously in operation. If the thread library cannot find a ready thread in the process, it makes a "block me" system call. The kernel now blocks the process. It will be unblocked when some event activates one of its threads and will resume execution of the thread library function, which will perform "scheduling" and switch to execution of the newly activated thread.

Scheduling of User-Level Threads Figure 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.

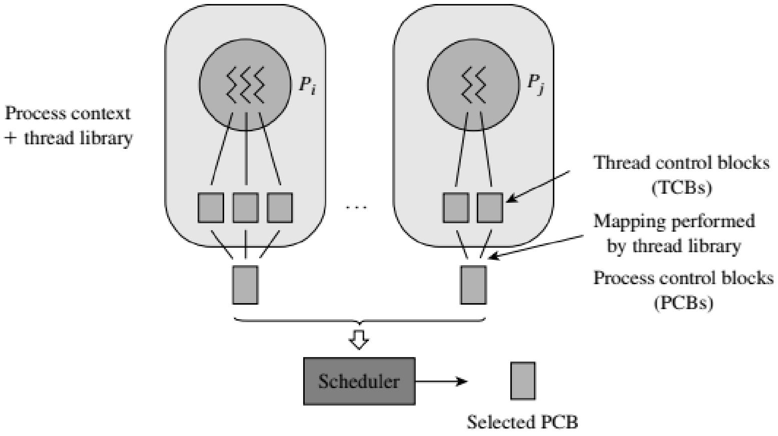


**Figure.** Scheduling of User Level Threads

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 non privileged 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 *programcounter* (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.

3.

5M

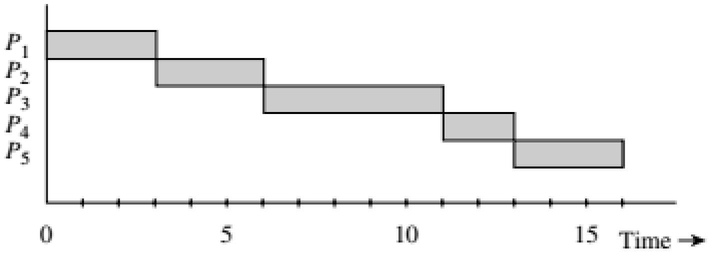
**Table 7.2** Processes for Scheduling

| **Process** | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| **Admission time** | 0 | 2 | 3 | 4 | 8 |
| **Service time** | 3 | 3 | 5 | 2 | 3 |

| **Time** | **Completed process** | | | **Processes in system (in FCFS order)** | **Scheduled process** |
|---|---|---|---|---|---|
| | ***id*** | ***ta*** | ***w*** | | |
| 0 | – | – | – | $P_1$ | $P_1$ |
| 3 | $P_1$ | 3 | 1.00 | $P_2, P_3$ | $P_2$ |
| 6 | $P_2$ | 4 | 1.33 | $P_3, P_4$ | $P_3$ |
| 11 | $P_3$ | 8 | 1.60 | $P_4, P_5$ | $P_4$ |
| 13 | $P_4$ | 9 | 4.50 | $P_5$ | $P_5$ |
| 16 | $P_5$ | 8 | 2.67 | – | – |

$\overline{ta}$ = 6.40 seconds

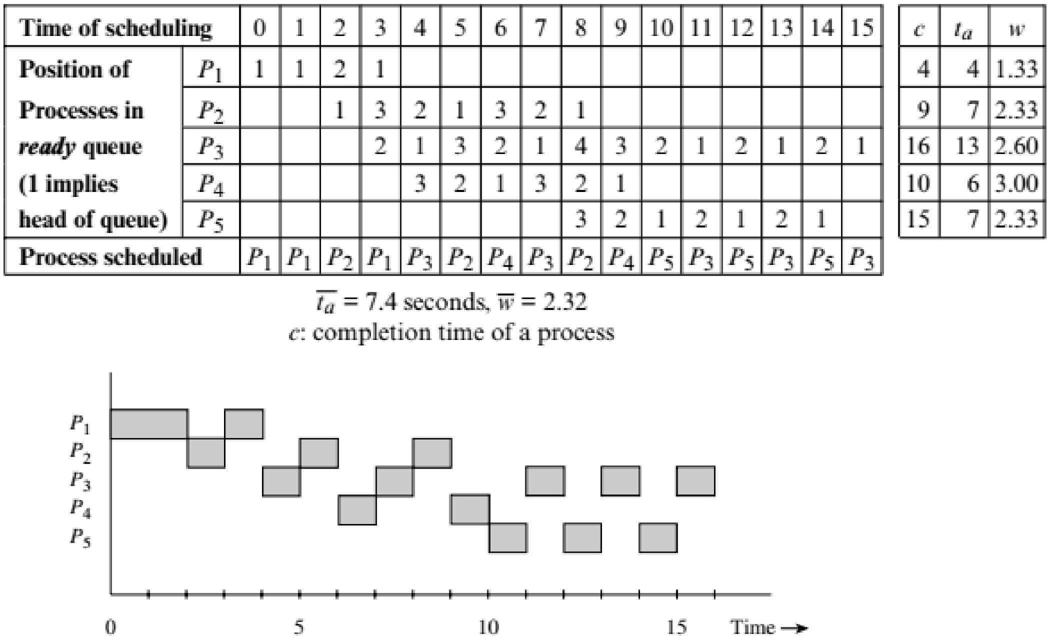
$\overline{w}$ = 2.22



Scheduling using the FCFS policy.

Figure 7.5 summarizes operation of the RR scheduler with $\delta = 1$ second for the five processes shown in Table 7.2. The scheduler makes scheduling decisions every second. The time when a decision is made is shown in the first row of the table in the top half of Figure 7.5. The next five rows show positions of the five processes in the ready queue. A blank entry indicates that the process is not in the system at the designated time. The last row shows the process selected by the scheduler; it is the process occupying the first position in the ready queue. Consider the situation at 2 seconds. The scheduling queue contains $P_2$ followed by $P_1$. Hence $P_2$ is scheduled. Process $P_3$ arrives at 3 seconds, and is entered in the queue. $P_2$ is also preempted at 3 seconds and it is entered in the queue. Hence the queue has process $P_1$ followed by $P_3$ and $P_2$, so $P_1$ is scheduled.

5M



| **Time of scheduling** | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | $c$ | $t_a$ | $w$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Position of** | $P_1$ | 1 | 1 | 2 | 1 | | | | | | | | | | | | | 4 | 4 | 1.33 |
| **Processes in** | $P_2$ | | | 1 | 3 | 2 | 1 | 3 | 2 | 1 | | | | | | | | 9 | 7 | 2.33 |
| ***ready* queue** | $P_3$ | | | | 2 | 1 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 2 | 1 | 2 | 1 | 16 | 13 | 2.60 |
| **(1 implies** | $P_4$ | | | | | 3 | 2 | 1 | 3 | 2 | 1 | | | | | | | 10 | 6 | 3.00 |
| **head of queue)** | $P_5$ | | | | | | | | | 3 | 2 | 1 | 2 | 1 | 2 | 1 | | 15 | 7 | 2.33 |
| **Process scheduled** | | $P_1$ | $P_1$ | $P_2$ | $P_1$ | $P_3$ | $P_2$ | $P_4$ | $P_3$ | $P_2$ | $P_4$ | $P_5$ | $P_3$ | $P_5$ | $P_3$ | $P_5$ | $P_3$ | | | |

$\overline{t_a}$ = 7.4 seconds, $\overline{w}$ = 2.32

$c$: completion time of a process

**Figure 7.5** Scheduling using the round-robin policy with time-slicing (RR).

The turnaround times and weighted turnarounds of the processes are as shown in the right part of the table. The *c* column shows completion times. The turnaround times and weighted turnarounds are inferior to those given by the non preemptive policies discussed because the CPU time is shared among many processes because of time-slicing. It can be seen that processes $P_2$, $P_3$, and $P_4$, which arrive at around the same time, receive approximately equal weighted turnarounds. $P_4$ receives the worst weighted turnaround because through most of its life it is one of three processes present in the system. $P_1$ receives the best weighted turnaround because no other process exists in the system during the early part of its execution. Thus weighted turnarounds depend on the load in the system.

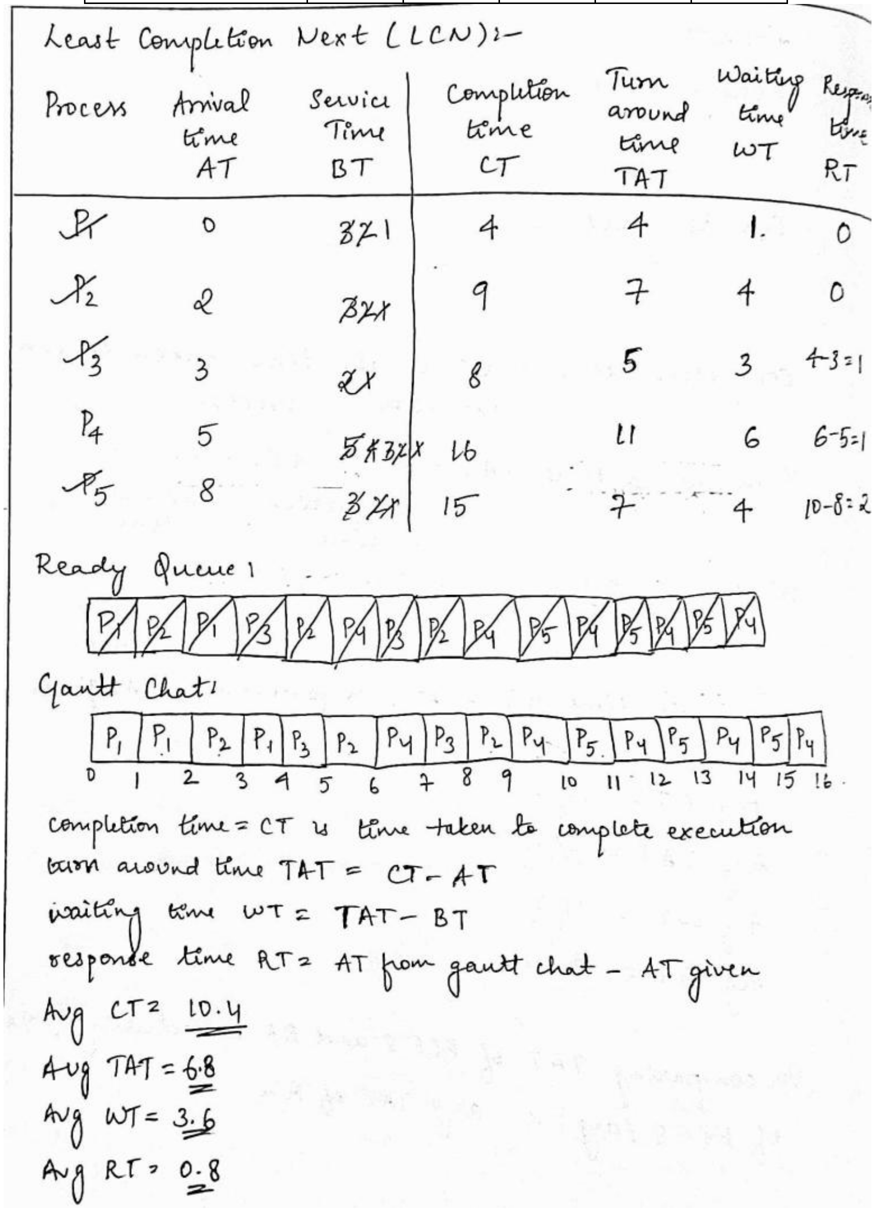
4.

Apply LCN scheduling method for the given below table and find the mean turnaround time and weighted turnaround time. Assume Time Slice of 1 Sec.

5M

Table.2 Processes for Scheduling

| **Process** | **P1** | **P2** | **P3** | **P4** | **P5** |
|---|---|---|---|---|---|
| **Admission Time** | 0 | 2 | 3 | 5 | 8 |
| **Service Time** | 3 | 3 | 2 | 5 | 3 |



Least Completion Next (LCN):-

| Process | Arrival time AT | Service Time BT | Completion time CT | Turn around time TAT | Waiting time WT | Response time RT |
|---|---|---|---|---|---|---|
| ~~P1~~ | 0 | ~~3~~ ~~2~~ 1 | 4 | 4 | 1. | 0 |
| ~~P2~~ | 2 | ~~3~~ ~~2~~ ~~1~~ | 9 | 7 | 4 | 0 |
| ~~P3~~ | 3 | ~~2~~ ~~1~~ | 8 | 5 | 3 | 4-3=1 |
| P4 | 5 | ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ | 16 | 11 | 6 | 6-5=1 |
| ~~P5~~ | 8 | ~~3~~ ~~2~~ ~~1~~ | 15 | 7 | 4 | 10-8=2 |

Ready Queue:

| ~~$P_1$~~ | ~~$P_2$~~ | ~~$P_1$~~ | ~~$P_3$~~ | ~~$P_2$~~ | ~~$P_4$~~ | ~~$P_3$~~ | ~~$P_2$~~ | ~~$P_4$~~ | ~~$P_5$~~ | ~~$P_4$~~ | ~~$P_5$~~ | ~~$P_4$~~ | ~~$P_5$~~ | ~~$P_4$~~ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Gantt Chat:

| $P_1$ | $P_1$ | $P_2$ | $P_1$ | $P_3$ | $P_2$ | $P_4$ | $P_3$ | $P_2$ | $P_4$ | $P_5$ | $P_4$ | $P_5$ | $P_4$ | $P_5$ | $P_4$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

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

completion time = CT is time taken to complete execution

turn around time TAT = CT - AT

waiting time WT = TAT - BT

response time RT = AT from gantt chat - AT given

Avg CT = 10.4

Avg TAT = 6.8

Avg WT = 3.6

Avg RT = 0.8

5M

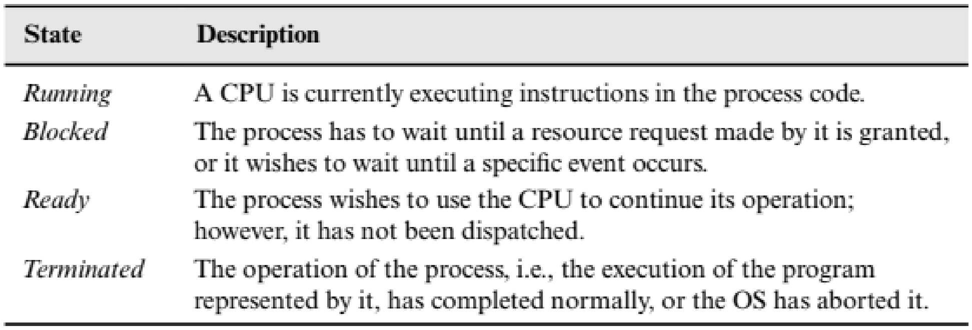
5.

## With neat diagram discuss Process states and their transition along with reasons of transitions.

5M

**Process state:** The indicator that describes the nature of the current activity of a process.

The kernel uses process states to simplify its own functioning, so the number of process states and their names may vary across OSs. However, most Oss use the four fundamental states described in Table. The kernel considers a process to be in the *blocked* state if it has made a resource request and the request is yet to be granted, or if it is waiting for some event to occur. A CPU should not be allocated to such a process until its wait is complete. The kernel would change the state of the process to *ready* when the request is granted or the event for which it is waiting occurs. Such a process can be considered for scheduling. The kernel would change the state of the process to *running* when it is dispatched. The state would be changed to *terminated* when execution of the process completes or when it is aborted by the kernel for some reason. A conventional computer system contains only one CPU, and so at most one process can be in the *running* state. There can be any number of processes in the *blocked*, *ready*, and *terminated* states. An OS may define more process states to simplify its own functioning or to support additional functionalities like swapping.



| State | Description |
|---|---|
| *Running* | A CPU is currently executing instructions in the process code. |
| *Blocked* | The process has to wait until a resource request made by it is granted, or it wishes to wait until a specific event occurs. |
| *Ready* | The process wishes to use the CPU to continue its operation; however, it has not been dispatched. |
| *Terminated* | The operation of the process, i.e., the execution of the program represented by it, has completed normally, or the OS has aborted it. |

**Process State Transitions** A *state transition* for a process $P_i$ is a change in its state. A state transition is caused by the occurrence of some event such as the start or end of an I/O operation. When the event occurs, the kernel determines its influence on activities in processes, and accordingly changes the state of an affected process. When a process $P_i$ in the *running* state makes an I/O request, its state has to be changed to *blocked* until its I/O operation completes. At the end of the I/O operation, $P_i$'s state is changed from *blocked* to *ready* because it now wishes to use the CPU. Similar state changes are made when a process makes some request that cannot immediately be satisfied by the OS. The process state is changed to *blocked* when the request is made, i.e., when the request event occurs, and it is changed to *ready* when the request is satisfied. The state of a *ready* process is changed to *running* when it is dispatched, and the state of a *running* process is changed to *ready* when it is preempted either because a higher-priority process became ready or because its time slice elapsed. Table 5.4 summarizes causes of state transitions. Figure 5.4 diagrams the fundamental state transitions for a process. A new process is put in the *ready* state after resources required by it have been allocated. It may enter the *running, blocked*, and *ready* states a number of times as a result of events described in Table 5.4. Eventually it enters the *terminated* state.

5M

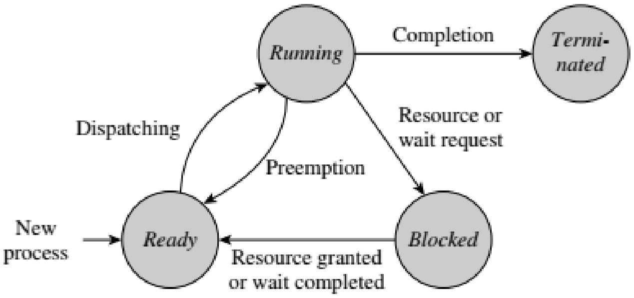


**Figure 5.4** Fundamental state transitions for a process.

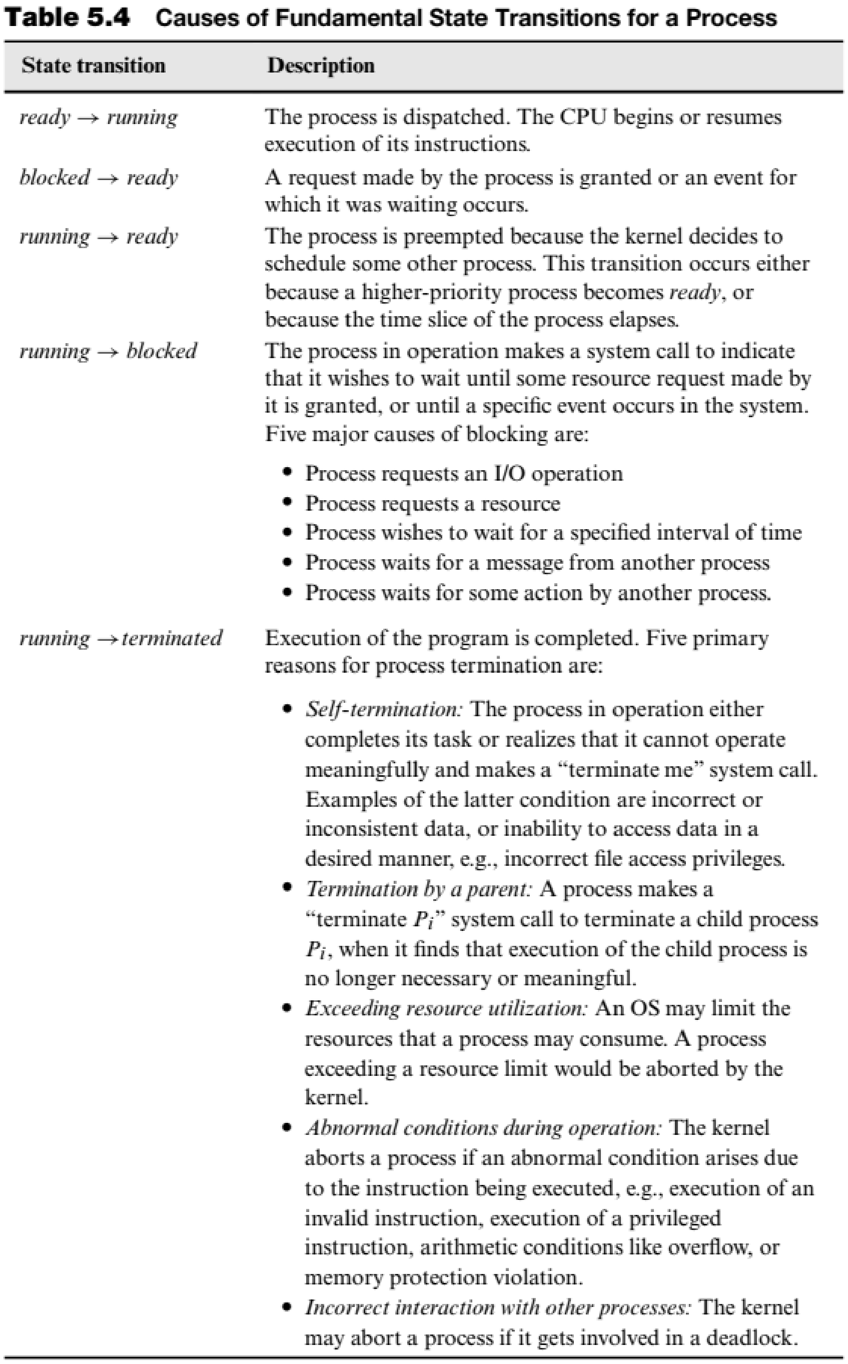


**Table 5.4** Causes of Fundamental State Transitions for a Process

| State transition | Description |
|---|---|
| *ready → running* | The process is dispatched. The CPU begins or resumes execution of its instructions. |
| *blocked → ready* | A request made by the process is granted or an event for which it was waiting occurs. |
| *running → ready* | The process is preempted because the kernel decides to schedule some other process. This transition occurs either because a higher-priority process becomes *ready*, or because the time slice of the process elapses. |
| *running → blocked* | The process in operation makes a system call to indicate that it wishes to wait until some resource request made by it is granted, or until a specific event occurs in the system. Five major causes of blocking are:<br>• Process requests an I/O operation<br>• Process requests a resource<br>• Process wishes to wait for a specified interval of time<br>• Process waits for a message from another process<br>• Process waits for some action by another process. |
| *running → terminated* | Execution of the program is completed. Five primary reasons for process termination are:<br>• *Self-termination:* The process in operation either completes its task or realizes that it cannot operate meaningfully and makes a "terminate me" system call. Examples of the latter condition are incorrect or inconsistent data, or inability to access data in a desired manner, e.g., incorrect file access privileges.<br>• *Termination by a parent:* A process makes a "terminate $P_i$" system call to terminate a child process $P_i$, when it finds that execution of the child process is no longer necessary or meaningful.<br>• *Exceeding resource utilization:* An OS may limit the resources that a process may consume. A process exceeding a resource limit would be aborted by the kernel.<br>• *Abnormal conditions during operation:* The kernel aborts a process if an abnormal condition arises due to the instruction being executed, e.g., execution of an invalid instruction, execution of a privileged instruction, arithmetic conditions like overflow, or memory protection violation.<br>• *Incorrect interaction with other processes:* The kernel may abort a process if it gets involved in a deadlock. |

6. Write a short notes on Scheduling Policies

4M

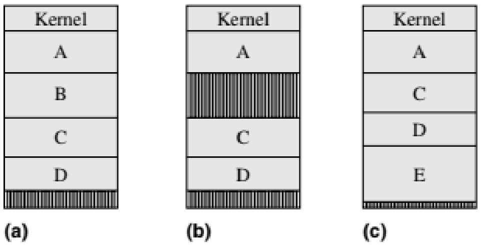
**Nonpreemptive scheduling**

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 never occurs. Non preemptive 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. We discuss three non preemptive scheduling policies in this section:

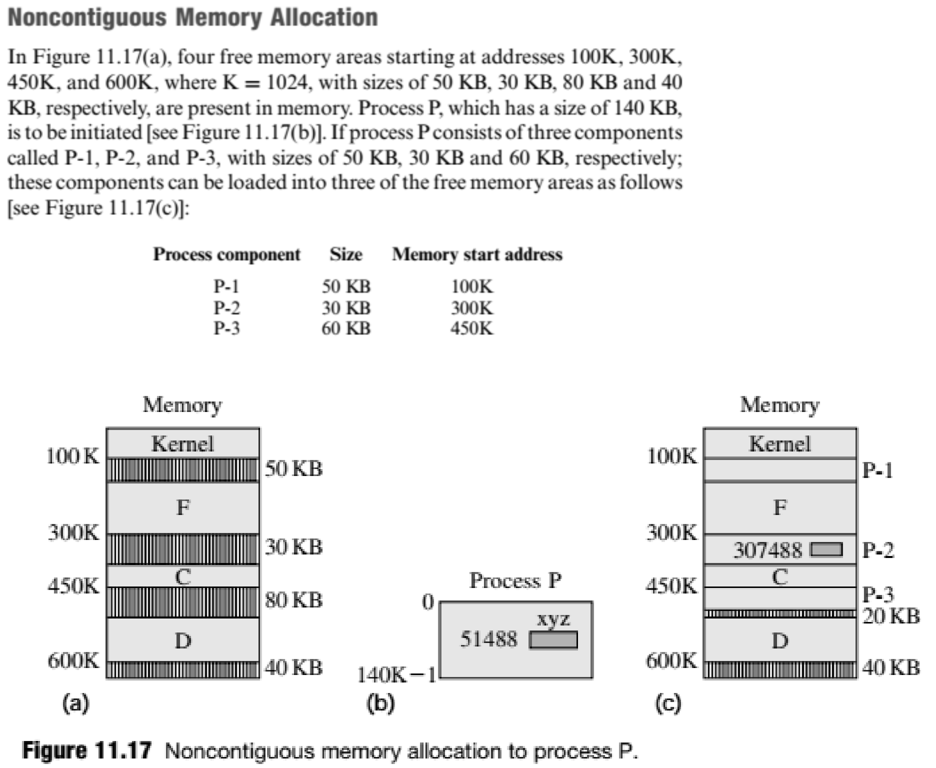
- First-come, first-served (FCFS) scheduling
- Shortest request next (SRN) scheduling
- Highest response ratio next (HRN) scheduling

**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.<br>**Shortest Request Next (SRN) Scheduling**<br>The SRN scheduler always schedules the request with the smallest service time.<br>Thus, a request remains pending until all shorter requests have been serviced. | |
| | **Preemptive scheduling**<br>In *preemptive scheduling*, the server can be switched to the processing of a new request before completing the current request. The preempted request is put back into the list of pending requests. 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 non preemptive scheduling is used. We discussed preemptive scheduling in multiprogramming and time-sharing operating systems.<br>Three preemptive scheduling policies are:<br>• Round-robin scheduling with time-slicing (RR)<br>• Least completed next (LCN) scheduling<br>• Shortest time to go (STG) scheduling<br>The RR scheduling policy shares the CPU among admitted requests by servicing them in turn. The other two policies take into account the CPU time required by a request or the CPU time consumed by it while making their scheduling decisions<br>The RR policy aims at providing good response times to all requests. The time slice, which is designated as $\delta$, is the largest amount of CPU time a request may use when scheduled. A request is preempted at the end of a time slice. To facilitate this, the kernel arranges to raise a timer interrupt when the time slice elapses.<br>**Least Completed Next (LCN) Scheduling**<br>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. | 5M |
| 7. | Discuss contiguous memory allocation and compare with Non-contiguous memory allocation<br>**Contiguous memory allocation:** Contiguous memory allocation is the classical memory allocation model in which each process is allocated a single contiguous area in memory. Thus the kernel allocates a large enough memory area to accommodate the code, data, stack, and PCD data of a process as shown in Figure 11.9. Contiguous memory allocation faces the problem of memory fragmentation. In this section we focus on techniques to address this problem. Relocation of a program in contiguous memory allocation and memory protection.<br>**Handling Memory Fragmentation** Internal fragmentation has no cure in contiguous memory allocation because the kernel has no means of estimating the memory requirement of a process accurately. The techniques of memory<br> <br>**Figure 11.16** Memory compaction. | 5M |
| | **Noncontiguous memory allocation:** Modern computer architectures provide the *noncontiguous memory allocation*model, in which a process can operate correctly even when portions of its address space are distributed among many areas of memory. This model of memory allocation permits the kernel to reuse free memory areas that are smaller than the size of a process, so it can reduce external fragmentation. | 5M |

Noncontiguous memory allocation using paging can even eliminate external fragmentation completely. We use the term *component* for that portion of the process address space that is loaded in a single memory area.



**Noncontiguous Memory Allocation**

In Figure 11.17(a), four free memory areas starting at addresses 100K, 300K, 450K, and 600K, where K = 1024, with sizes of 50 KB, 30 KB, 80 KB and 40 KB, respectively, are present in memory. Process P, which has a size of 140 KB, is to be initiated [see Figure 11.17(b)]. If process P consists of three components called P-1, P-2, and P-3, with sizes of 50 KB, 30 KB and 60 KB, respectively; these components can be loaded into three of the free memory areas as follows [see Figure 11.17(c)]:

| Process component | Size | Memory start address |
|---|---|---|
| P-1 | 50 KB | 100K |
| P-2 | 30 KB | 300K |
| P-3 | 60 KB | 450K |

**Figure 11.17** Noncontiguous memory allocation to process P.