Unit-2

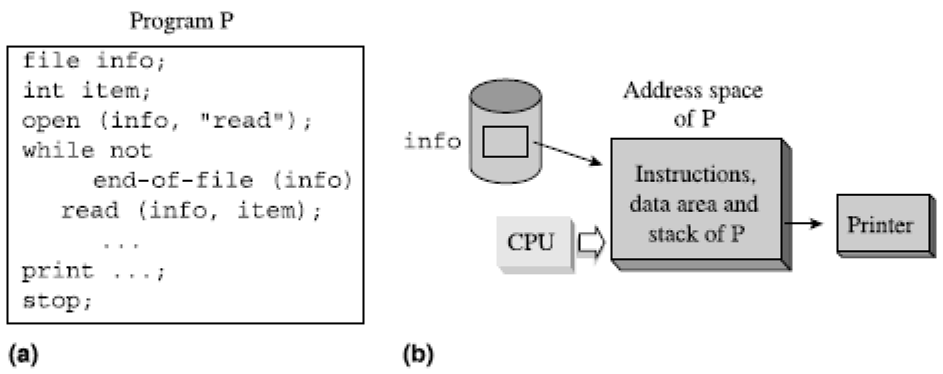
Process Management

An OS contains a large number of processes at any time. Process management involves creating processes, fulfilling their resource requirements, *scheduling* them for use of a CPU, implementing *process synchronization* to control their interactions, avoiding *deadlocks* so that they do not wait for each other indefinitely, and terminating them when they complete their operation. The manner in which an OS schedules processes for use of a CPU determines the response times of processes, resource efficiency, and system performance.

A process is *an* execution of a program using a set of resources.

What Is a Process?

Program P shown in Figure 5.1(a) contains declarations of file info and a variable item, and statements that read values from info, use them to perform some calculations, and print a result before coming to a halt.



Program P

```
file info;
int item;
open (info, "read");
while not
      end-of-file (info)
    read (info, item);
      ...
print ...;
stop;
```

Figure 5.1 A program and an abstract view of its execution.

During execution, instructions of this program use values in its data area and the stack to perform the intended calculations. Figure 5.1(b) shows an abstract view of its execution. The instructions, data, and stack of program P constitute its *address space*. To realize execution of P, the OS allocates memory to accommodate P's address space, allocates a printer to print its results, sets up an arrangement through which P can access file info, and schedules P for execution. The CPU is shown as a lightly shaded box because it is not always executing instructions of P—the OS shares the CPU between execution of P and executions of other programs.

Definition: Process-An execution of a program using resources allocated to it.

When a user initiates execution of a program, the OS creates a new process and assigns a unique id to it. It now allocates some resources to the process— sufficient memory to accommodate the address space of the program, and some devices such as a keyboard and a monitor to facilitate interaction with the user.

The process may make system calls during its operation to request additional resources such as files. We refer to the address space of the program and resources allocated to it as the address space and resources of the process, respectively.

Accordingly, a process comprises six components:

(id, code, data, stack, resources, CPU state)

Where, *id* is the unique id assigned by the OS

code is the code of the program (it is also called the *text* of a program)

data is the data used in the execution of the program, including data from files

stack contains parameters of functions and procedures called during execution of the program, and their return addresses *resources* is the set of resources allocated by the OS

CPU state is composed of contents of the PSW and the general-purpose registers (GPRs) of the CPU (we assume that the stack pointer is maintained in a GPR)

The CPU state contains information that indicates which instruction in the code would be executed next, and other information—such as contents of the *condition code* field (also called the *flags* field) of the PSW— that may influence its execution. The CPU state changes as the execution of the program progresses. We use the term *operation of a process* for execution of a program. Thus a process operates when it is scheduled.

❖ Relationships between Processes and Programs

A program consists of a set of functions and procedures. During its execution, control flows between the functions and procedures according to the logic of the program.

The OS does not know anything about the nature of a program, including functions and procedures in its code. It knows only what it is told through system calls. The rest is under control of the program. Thus functions of a program may be separate processes, or they may constitute the code part of a single process.

Table 5.1 shows two kinds of relationships that can exist between processes and programs. A one-to-one relationship exists when a single execution of a sequential program is in progress, for example, execution of program P in Figure 5.1. A many-to-one relationship exists between many processes and a program in two cases: Many executions of a program may be in progress at the same time; processes representing these executions have a many-to-one relationship with the program. During execution, a program may make a system call to request that a specific part of its code should be executed concurrently, i.e., as a separate activity occurring at the same time. The kernel sets up execution of the specified part of the code and treats it as a separate process. The new process and the process representing execution of the program have a many-to-one relationship with the program. We call such a program a *concurrent program*.

Processes that coexist in the system at some time are called *concurrent processes*. Concurrent processes may share their code, data and resources with other processes; they have opportunities to interact with one another during their execution.

Table 5.1 Relationships between Processes and Programs

Relationship	Examples
One-to-one	A single execution of a sequential program.
Many-to-one	Many simultaneous executions of a program, execution of a concurrent program.

➢ Child Processes

The kernel initiates an execution of a program by creating a process for it. For lack of a technical term for this process, we will call it the *primary process* for the program execution. The primary process may make system calls as described in the previous section to create other processes—these processes become its *child processes*, and the primary process becomes their *parent*. A child process may itself create other processes, and so on. The parent–child relationships between these processes can be represented in the form of a *process tree*, which has the primary process as its root. A child process may inherit some of the resources of its parent; it could obtain additional resources during its operation through system calls.

Typically, a process creates one or more child processes and delegates some of its work to each of them. It is called *multitasking* within an application. It has the three benefits summarized in Table 5.2.

Table 5.2 Benefits of Child Processes

Benefit	Explanation
Computation speedup	Actions that the primary process of an application would have performed sequentially if it did not create child processes, would be performed concurrently when it creates child processes. It may reduce the duration, i.e., running time, of the application.
Priority for critical functions	A child process that performs a critical function may be assigned a high priority; it may help to meet the real-time requirements of an application.
Guarding a parent process against errors	The kernel aborts a child process if an error arises during its operation. The parent process is not affected by the error; it may be able to perform a recovery action.

❖ Process States and State Transitions

An operating system uses the notion of a *process state* to keep track of what a process is doing at any moment.
Definition 5.2 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 5.3. 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.

Table 5.3 Fundamental Process States

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 *Pi* 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 *Pi* 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, *Pi*'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.

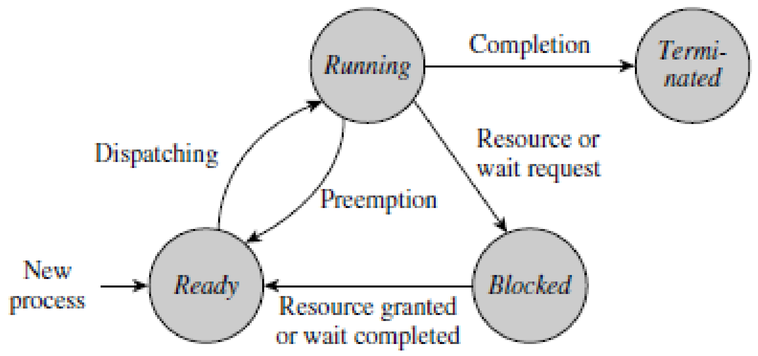


Figure 5.4 Fundamental state transitions for a process.

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.

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: • Process requests an I/O operation • Process requests a resource • Process wishes to wait for a specified interval of time • Process waits for a message from another process • Process waits for some action by another process.
running → *terminated*	Execution of the program is completed.

❖ Process Context and the Process Control Block

The kernel allocates resources to a process and schedules it for use of the CPU. Accordingly, the kernel's view of a process consists of two parts:

• Code, data, and stack of the process, and information concerning memory and other resources, such as files, allocated to it.
• Information concerning execution of a program, such as the process state, the CPU state including the stack pointer, and some other items of information described later in this section.

These two parts of the kernel's view are contained in the *process context* and the *process control block* (PCB), respectively (see Figure 5.6). This arrangement enables different OS modules to access relevant process-related information conveniently and efficiently.

➢ **Process Context:** The process context consists of the following:

1. ***Address space of the process:*** The code, data, and stack components of the process.

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.

3. ***Status of file processing activities:*** Information about files being used, such as current positions in the files.

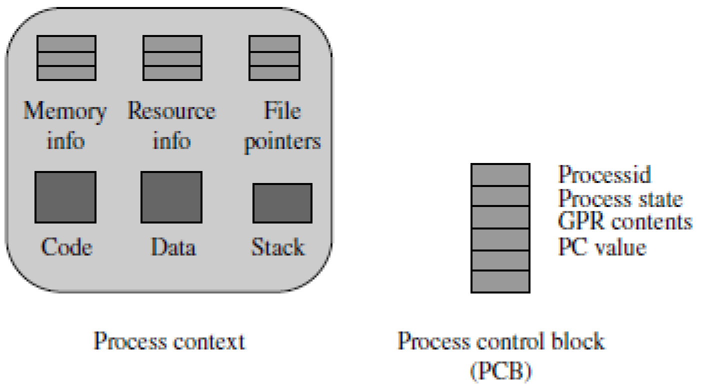


Figure 5.6 Kernel's view of a process.

4. ***Process interaction information:*** Information necessary to control interaction of the process with other processes, e.g., ids of parent and child processes, and inter process messages sent to it that have not yet been delivered to it.

5. ***Resource information:*** Information concerning resources allocated to the process.

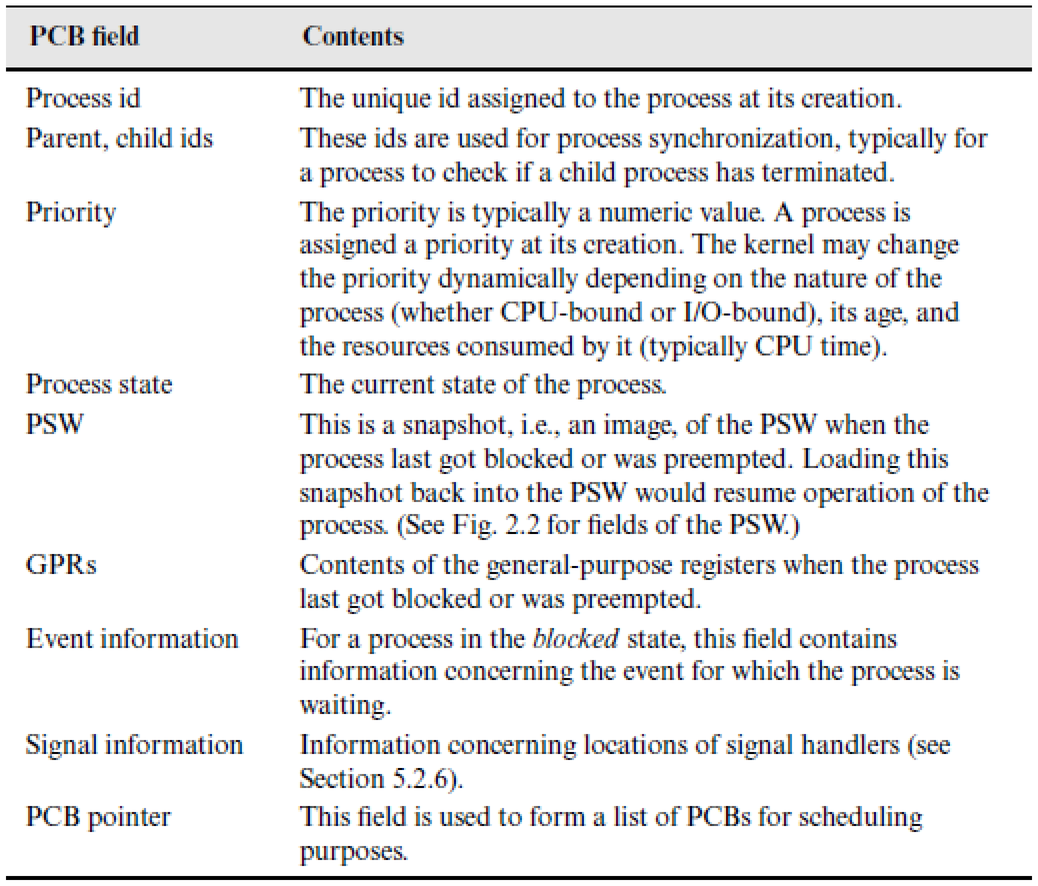
6. ***Miscellaneous information:*** Miscellaneous information needed for operation of a process.

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.

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. Table 5.6 describes the fields of the PCB data structure.

Table 5.6 Fields of the Process Control Block (PCB)



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.

❖ Sharing, Communication, and Synchronization between Processes

Processes of an application need to interact with one another because they work toward a common goal. Table 5.7 describes four kinds of process interaction. We summarize their important features in the following.

Data Sharing A shared variable may get inconsistent values if many processes update it concurrently. For example, if two processes concurrently execute the statement a:= a+1, where a is a shared variable, the result may depend on the way the kernel interleaves their execution—the value of a may be incremented by only 1! (We discuss this problem later in Section 6.2.) To avoid this problem, only one process should access shared data at any time, so a data access in one process may have to be delayed if another process is accessing the data. This is called *mutual exclusion*. Thus, data sharing by concurrent processes incurs the overhead of mutual exclusion.

Message Passing A process may send some information to another process in the form of a message. The other process can copy the information into its own data structures and use it. Both the sender and the receiver process must anticipate the information exchange, i.e., a process must know when it is expected to send or receive a message, so the information exchange becomes a part of the convention or protocol between processes.

Synchronization The logic of a program may require that an action *ai* should be performed only after some action *aj* has been performed. Synchronization between processes is required if these actions are performed in different processes—the process that wishes to perform action *ai* is made to wait until another process performs action *aj*.

Signals A signal is used to convey an exceptional situation to a process so that it may handle the situation through appropriate actions. The code that a process wishes to execute on receiving a signal is called a *signal handler*. The signal mechanism is modeled along the lines of interrupts. Thus, when a signal is sent to a process, the kernel interrupts operation of the process and executes a signal handler, if one has been specified by the process; otherwise, it may perform a default action. Operating systems differ in the way they resume a process after executing a signal handler.

Table 5.7 Four Kinds of Process Interaction

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.

❖ THREADS

Applications use concurrent processes to speed up their operation. However, switching between processes within an application incurs high process switching overhead because the size of the process state information is large, so operating system designers developed an alternative model of execution of a program, called a thread that could provide concurrency within an application with less overhead.

Definition: Thread- An execution of a program that uses the resources of a process.

A process creates a thread through a system call. The thread does not have resources of its own, so it does not have a context; it operates by using the context of the process, and accesses the resources of the process through it. We use the phrases “thread(s) of a process” and “parent process of a thread” to describe the relationship between a thread and the process whose context it uses. Note that threads are not a substitute for child processes; an application would create child processes to execute different parts of its code, and each child process can create threads to achieve concurrency.

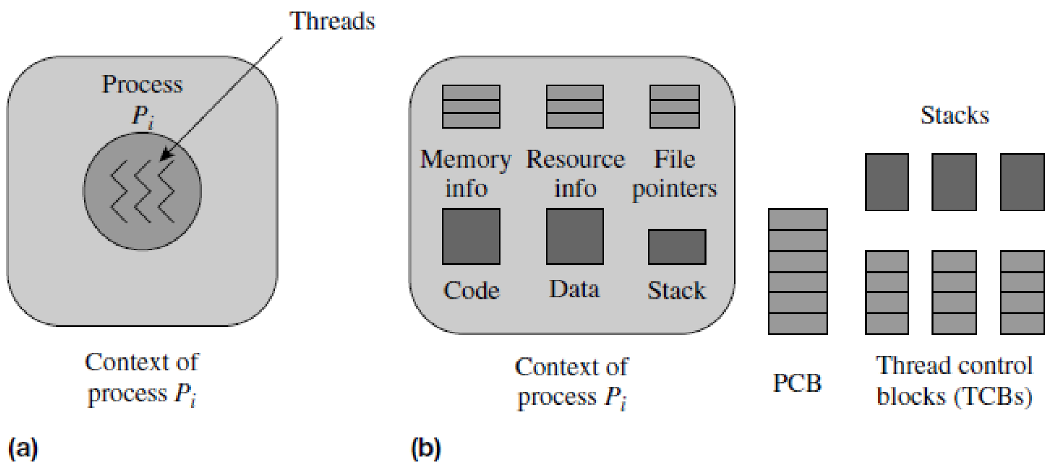


Figure 5.11 Threads in process P_i: (a) concept; (b) implementation.

Figure 5.11 illustrates the relationship between threads and processes. In the abstract view of Figure 5.11(a), process *Pi* has three threads, which are represented by wavy lines inside the circle representing process *Pi* . Figure 5.11(b) shows an implementation arrangement. Process *Pi* has a context and a PCB. Each thread of *Pi* is an execution of a program, so it has its own stack and a *thread control block* (TCB), which is analogous to the PCB and stores the following information:

1. Thread scheduling information—thread id, priority and state.
2. CPU state, i.e., contents of the PSW and GPRs.
3. Pointer to PCB of parent process.
4. TCB pointer, which is used to make lists of TCBs for scheduling.

➢ Thread States and State Transitions

Barring the difference that threads do not have resources allocated to them, threads and processes are analogous. Hence thread states and thread state transitions are analogous to process states and process state transitions. When a thread is created, it is put in the *ready* state because its parent process already has the necessary resources allocated to it. It enters the *running* state when it is dispatched. It does not enter the *blocked* state because of resource requests, because it does not make any resource requests; however, it can enter the *blocked* state because of process synchronization requirements.

➢ Advantages of Threads over Processes

Table 5.8 summarizes the advantages of threads over processes, of which we have already discussed the advantage of lower overhead of thread creation and switching. Unlike child processes, threads share the address space of the parent process, so they can communicate through shared data rather than through messages, thereby eliminating the overhead of system calls.

Table 5.8 Advantages of Threads over Processes

Advantage	Explanation
Lower overhead of creation and switching	Thread state consists only of the state of a computation. Resource allocation state and communication state are not a part of the thread state, so creation of threads and switching between them incurs a lower overhead.
More efficient communication	Threads of a process can communicate with one another through shared data, thus avoiding the overhead of system calls for communication.
Simplification of design	Use of threads can simplify design and coding of applications that service requests concurrently.

❖ Kernel-Level, User-Level, and Hybrid Threads

These three models of threads differ in the role of the process and the kernel in the creation and management of threads. This difference has a significant impact on the overhead of thread switching and the concurrency and parallelism within a process.

➢ *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 5.14 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.

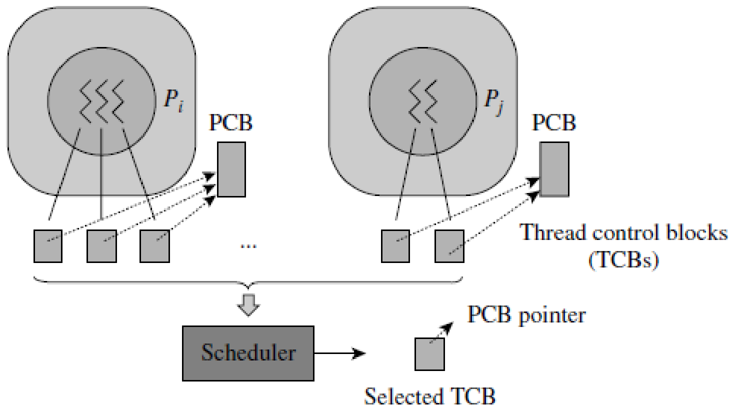


Figure 5.14 Scheduling of kernel-level threads.

When an event occurs, the kernel saves the CPU state of the interrupted thread in its TCB. After event handling, the scheduler considers TCBs of all threads and selects one ready thread; the dispatcher uses the PCB pointer in its 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.

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.

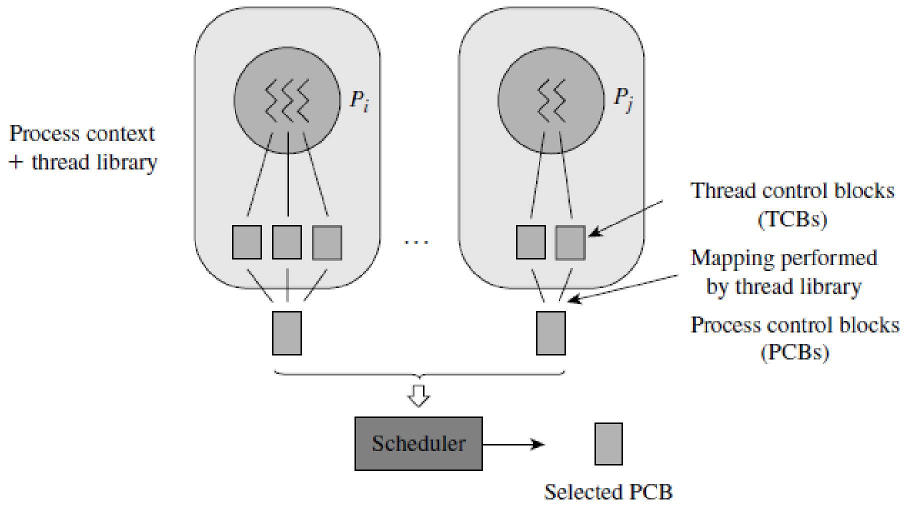


Figure 5.15 Scheduling of user-level threads.

Figure 5.15 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. The next example illustrates interesting situations during scheduling of user-level threads.

➢ *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.

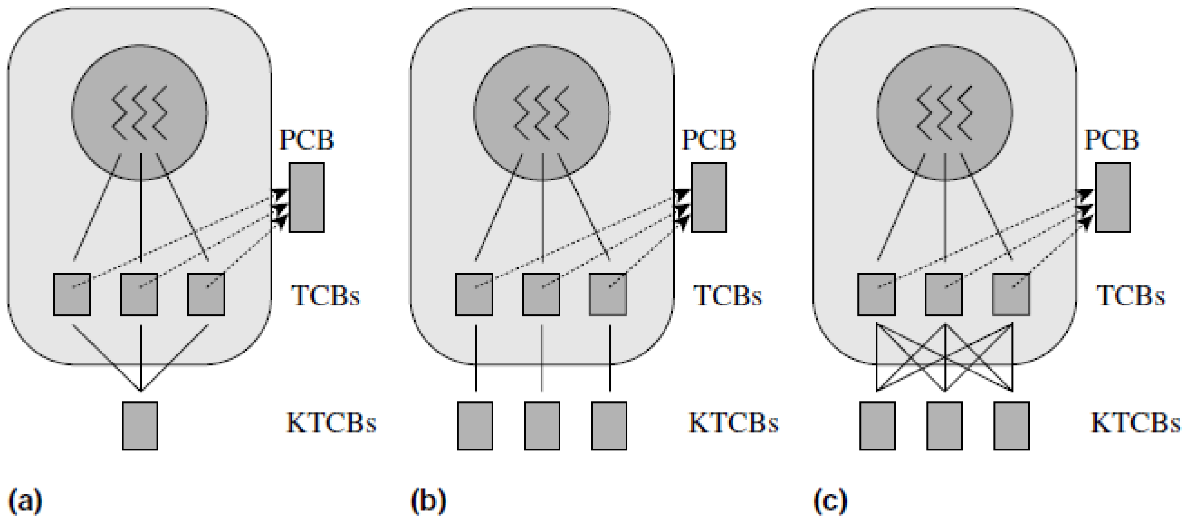


Figure 5.17 (a) Many-to-one; (b) one-to-one; (c) many-to-many associations in hybrid threads.

Figure 5.17 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 user level 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.

❖ Scheduling

A scheduling policy decides which process should be given the CPU at the present moment. This decision influences both system performance and user service.

Scheduling, very generally, is the activity of selecting the next request to be serviced by a server. Figure 7.1 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 7.1. Events related to a request are its arrival, admission, scheduling, preemption, and completion.

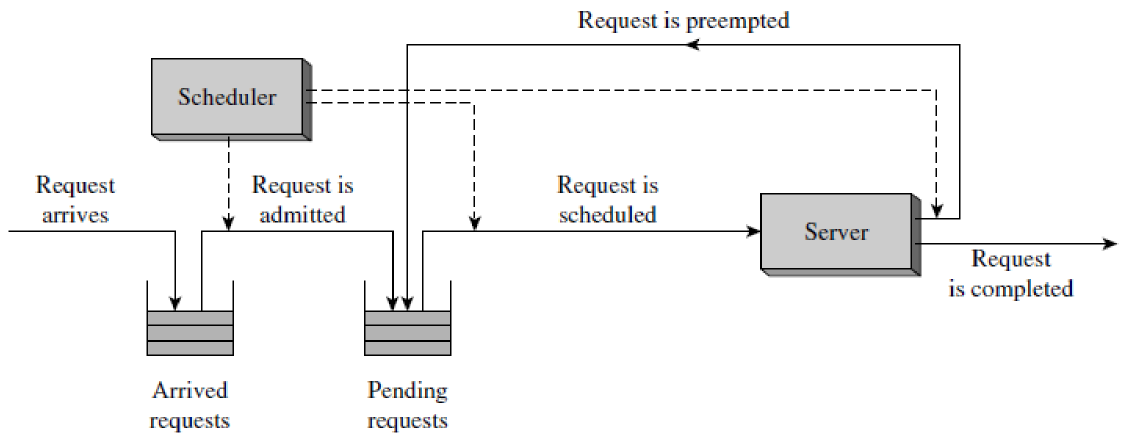


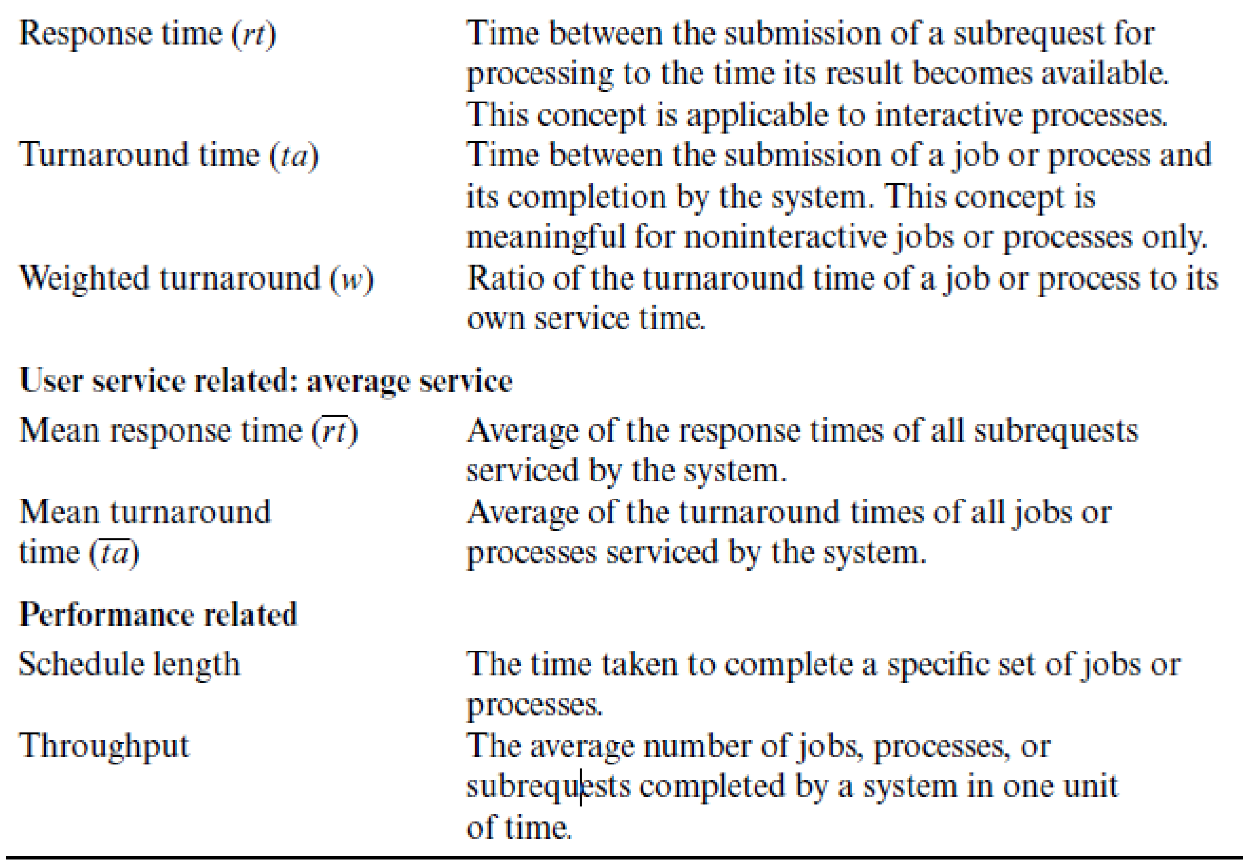
Figure 7.1 A schematic of scheduling.

In an operating system, a request is the execution of a job or a process, and the server is the CPU. A job or a process is said to arrive when it is submitted by a user, and to be admitted when the scheduler starts considering it for scheduling. An admitted job or process either waits in the list of pending requests, uses the CPU, or performs I/O operations. Eventually, it completes and leaves the system. The scheduler's action of admitting a request is important only in an operating system with limited resources.

Table 7.1 lists the key terms and concepts related to scheduling. The service time of a job or a process is the total of CPU time and I/O time required by it to complete its execution, and the deadline, which is specified only in real-time systems, is the time by which its servicing should be completed. Both service time and deadline are an inherent property of a job or a process. The completion time of a job or a process depends on its arrival and service times, and on the kind of service it receives from the OS.

Table 7.1 Scheduling Terms and Concepts

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.

➢ 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.
- ***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.

➢ Non-preemptive Scheduling Policies

In non-preemptive 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.

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

➢ Preemptive Scheduling Policies

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 discuss three preemptive scheduling policies in this section:

- Round-robin scheduling with time-slicing (RR)
- Least completed next (LCN) scheduling
- Shortest time to go (STG) scheduling

Non-preemptive Scheduling

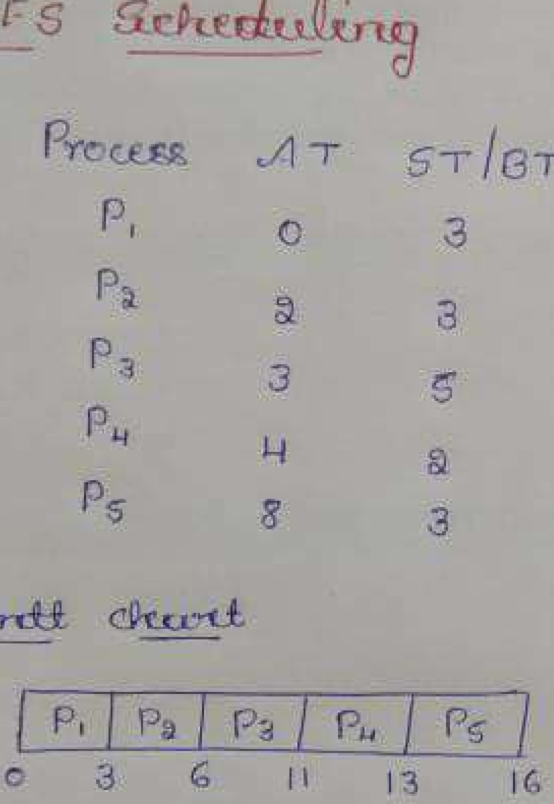
- First-come, first-served (FCFS) Scheduling
- Shortest request next (SRN) Scheduling
- Highest response ratio next (HRRN) Scheduling

* FCFS Scheduling

Process	AT	ST/BT
P_1	0	3
P_2	2	3
P_3	3	5
P_4	4	2
P_5	8	3

AT : Arrival Time
ST : Service Time
BT : Burst Time
CT : Completion Time
TAT : Turnaround Time
w : Weighted turnaround

Gantt chart



TAT

Process	AT	ST	CT	TAT	w
P_1	0	3	3	3	1
P_2	2	3	6	4	1.33
P_3	3	5	11	8	1.6
P_4	4	2	13	9	4.5
P_5	8	3	16	8	2.67

* = $TAT = CT - AT$

* $w = \frac{TAT}{ST}$

Average TAT $(\bar{t_a})$ = $(3+4+8+9+8) / 5$ = 6.40 seconds

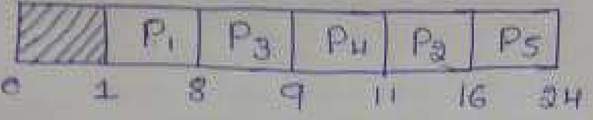
Average weighted turnaround $(\bar{w})$ = $\frac{(1+1.33+1.6+4.5+2.67)}{5}$

= $\underline{\underline{2.22}}$.

* Shortest Request Next Scheduling

Process	AT	ST
P_1	1	7
P_2	2	5
P_3	3	1
P_4	4	2
P_5	5	8

Grantt chart



Process	AT	ST	CT	TAT	W
P_1	1	7	8	7	1
P_2	2	5	16	14	2.8
P_3	3	1	9	6	6
P_4	4	2	11	7	3.5
P_5	5	8	24	19	2.3

$\overline{ta}$ = 10.6 seconds

$\overline{w}$ = 3.12

Highest Response Ratio Next Scheduling

Process	AT	ST
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

$$\text{Response Ratio} = \frac{\text{waiting time} + \text{Service time}}{\text{Service time}}$$

Gantt chart

P_1	P_2	P_3	P_5	P_4

0 3 9 13 15 20

$RR_3 = \frac{5+4}{4} = 2.25$ ✓

$RR_4 = \frac{3+5}{5} = 1.6$

$RR_5 = \frac{1+2}{2} = 1.5$

$RR_4 = \frac{7+5}{5} = 2.4$

$RR_5 = \frac{5+2}{2} = 3.5$

Process	AT	ST	CT	TAT	W
P_1	0	3	3	3	1
P_2	2	6	9	7	1.16
P_3	4	4	13	9	2.25
P_4	6	5	20	14	2.8
P_5	8	2	15	7	3.5

$\bar{t_a} = 8$ seconds

$\bar{w} = 2.142$

Preemptive Scheduling

- Round-robin Scheduling (RR)
- least completed next (LCN) scheduling
- Shortest time to go (STGO) Scheduling

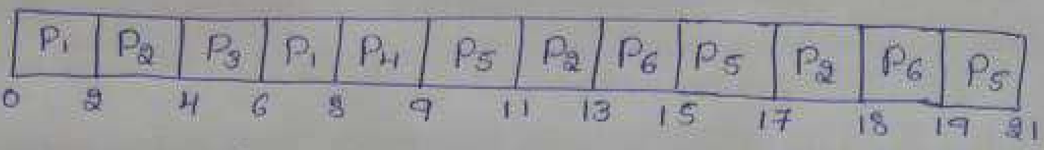
* Round-robin Scheduling

Time quantum = 2

Process	AT	BT/ST
P_1	0	$4 \to 2 \to 0$
P_2	1	$5 \to 3 \to 1 \to 0$
P_3	2	$2 \to 0$
P_4	3	$1 \to 0$
P_5	4	$6 \to 4 \to 2 \to 0$
P_6	6	$3 \to 1 \to 0$

P_1 P_2 P_3 P_1 P_4 P_5 P_2 P_6 P_5 P_2 P_6 P_5

Gantt chart



Process	AT	BT/ST	CT	TAT	W
P_1	0	4	8	8	2
P_2	1	5	18	17	3.4
P_3	2	2	6	4	2
P_4	3	1	9	6	6
P_5	4	6	21	17	2.83
P_6	6	3	19	13	4.33

$\bar{t_a} = 10.83$

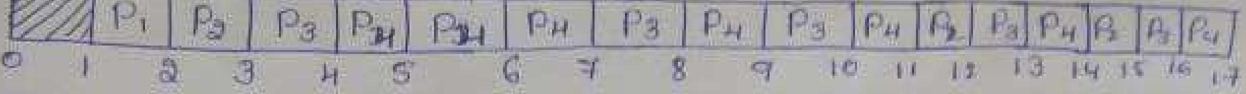
$\bar{W} = 3.42$

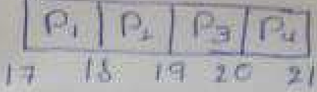
* Least Completed Next (LCN) Scheduling

TQ = 1s

Process	AT	ST
P_1	1	$2 \to 1$
P_2	2	$4 \to 3 \to 2 \to 1$
P_3	3	$6 \to 5 \to 4 \to 3 \to 2 \to$
P_4	4	$8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 1$

Gantt Chart





Process	AT	ST	CT	TAT	W
P_1	1	2	18	17	8.5
P_2	2	4	19	17	4.25
P_3	3	6	20	17	2.83
P_4	4	8	21	17	2.125

$\bar{t_a} = 17s$

$\bar{W} = 4.42$

* Shortest time to Go (STG) Scheduling

Process	AT	ST	
P_1	0	$7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$	TQ = 1x
P_2	1	$5 \to 4 \to 3 \to 2 \to 1 \to 0$	
P_3	2	$3 \to 2 \to 1 \to 0$	
P_4	3	$1 \to 0$	
P_5	4	$2 \to 1 \to 0$	
P_6	5	$1 \to 0$	

Gantt chart

P_1	P_2	P_3	P_4	P_3	P_3	P_6	P_5	P_5	P_2	P_2	P_2	P_2	P_1	P_1	P_1	P_1	P_1	P_1
0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–9	9–10	10–11	11–12	12–13	13–14	14–15	15–16	16–17	17–18	18–19

Process	AT	ST	CT	TAT	w
P_1	0	7	19	19	2.71
P_2	1	5	13	12	2.4
P_3	2	3	6	4	1.33
P_4	3	1	4	1	1
P_5	4	2	9	5	2.5
P_6	5	1	7	2	2

$\overline{Ta} = 7.168$

$\overline{w} = \underline{\underline{1.99}}$

➢ Long-, Medium-, and Short-Term Schedulers

The 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 7.9 shows an overview of scheduling and related actions. Every event that requires the kernel's attention causes an interrupt. The interrupt processing routine performs a context save function and invokes an event handler. The event handler analyzes the event and changes the state of the process, if any, affected by it. It then invokes the long-term, medium-term, or short-term scheduler as appropriate. For example, the event handler that creates a new process invokes the long-term scheduler, event handlers for suspension and resumption of processes invoke the medium-term scheduler, and the memory handler may invoke the medium-term scheduler if it runs out of memory. Most other event handlers directly invoke the short-term scheduler.

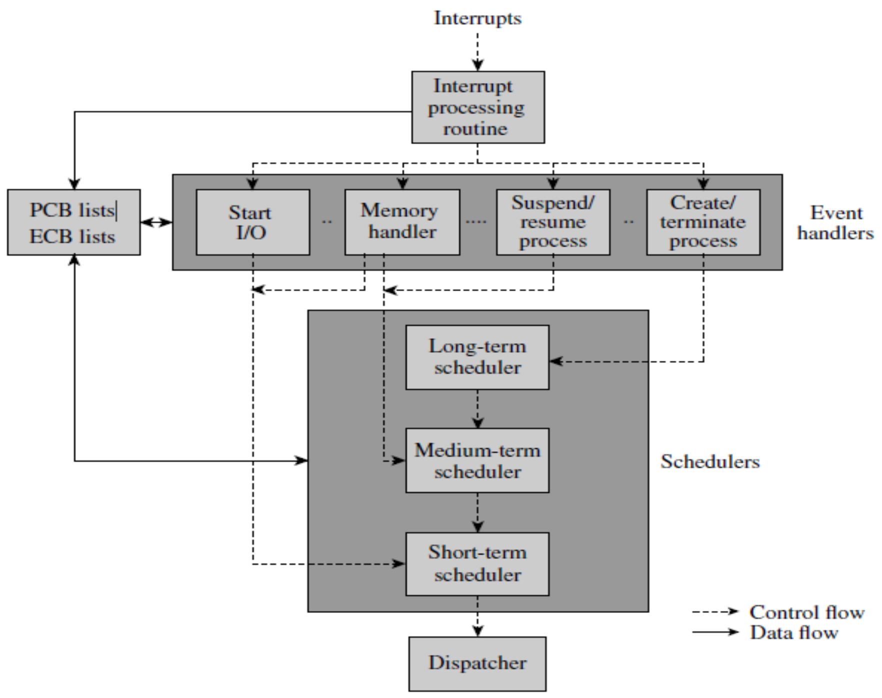


Figure 7.9 Event handling and scheduling.