***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.

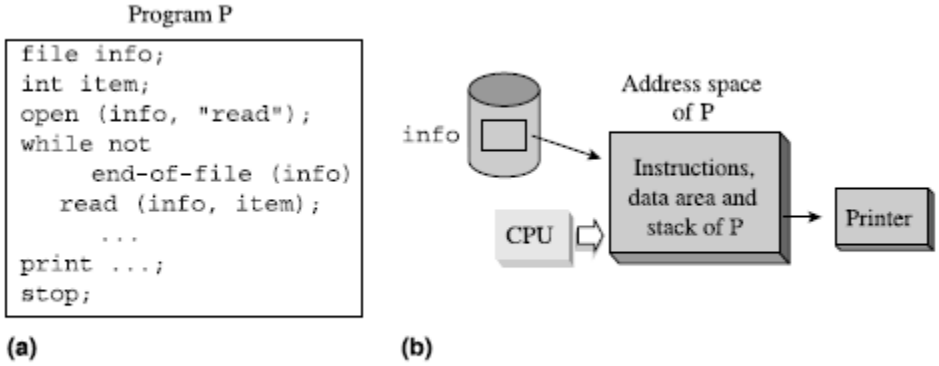


**Figure 5.1** A program and an abstract view of its execution.

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.

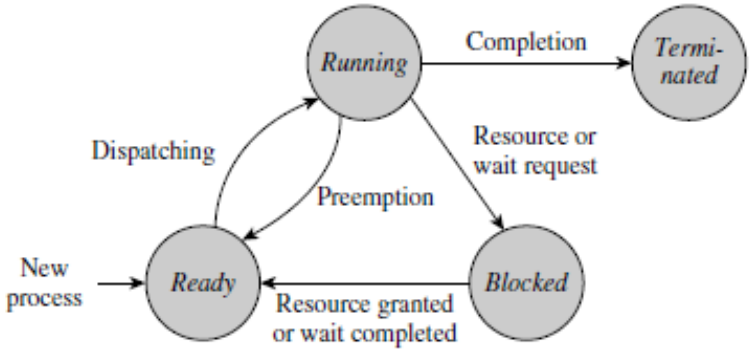


**Figure 5.4** Fundamental state transitions for a process.

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:<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. |

## ❖ 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.

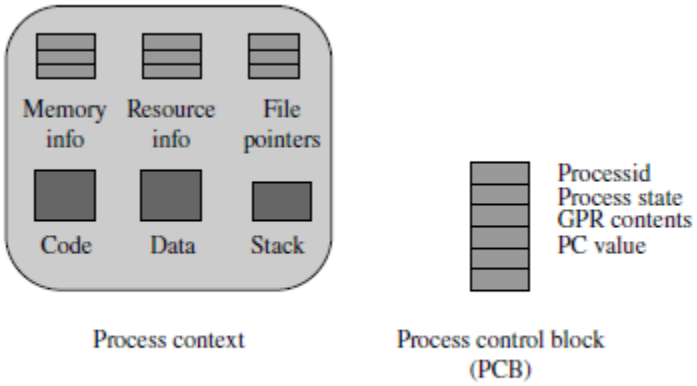


**Figure 5.6** Kernel's view of a process.

**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. |