Definition of Process

The notion of process is central to the understanding of operating systems. There are quite a few definitions presented in the literature, but no "perfect" definition has yet appeared.

Definition

The term "process" was first used by the designers of the MULTICS in 1960's. Since then, the term process, used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions for instance

- A program in Execution.
- An asynchronous activity.
- The 'animated sprit' of a procedure in execution.
- The entity to which processors are assigned.
- The 'dispatchable' unit.

and many more definitions have given. As we can see from above that there is no universally agreed upon definition, but the definition "*Program in Execution*" seem to be most frequently used. And this is a concept are will use in the present study of operating systems.

Now that we agreed upon the definition of process, the question is what is the relation between process and program. It is same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process. Well, to be very precise. Process is not the same as program. In the following discussion we point out some of the difference between process and program. As we have mentioned earlier.

Process is not the same as program. A process is more than a program code. A process is an 'active' entity as oppose to program which consider to be a 'passive' entity. As we all know that a program is an algorithm expressed in some suitable notation, (e.g., programming language). Being a passive, a program is only a part of process. Process, on the other hand, includes:

- Current value of Program Counter (PC)
- Contents of the processors registers
- Value of the variables
- The process stack (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
- A data section that contains global variables.

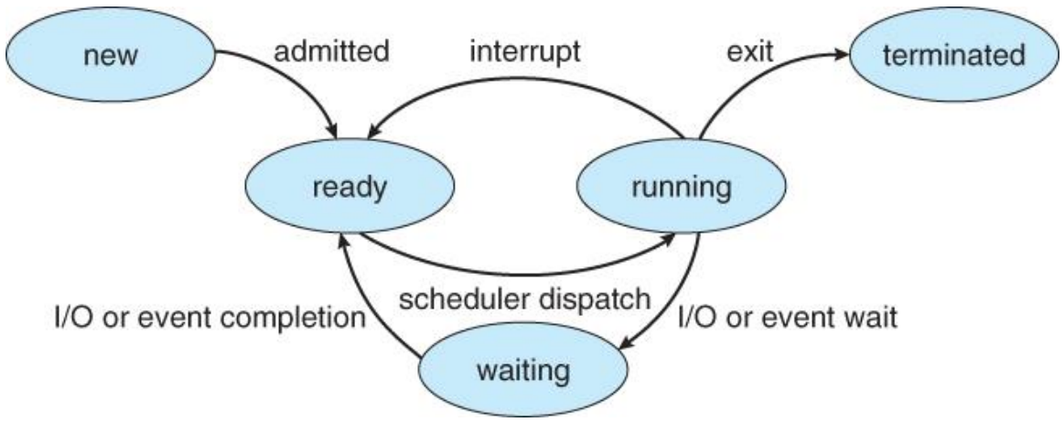
A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multiprogramming).

Process State

The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

- Code for the program.
- Program's static data.
- Program's dynamic data.
- Program's procedure call stack.
- Contents of general purpose registers.
- Contents of program counter (PC)
- Contents of program status word (PSW).
- Operating Systems resource in use.



A process goes through a series of discrete process states.

- **New State:** The process being created.
- **Running State:** A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
- **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
- **Ready State:** A process is said to be ready if it use a CPU if one were available. A ready state process is runable but temporarily stopped running to let another process run.
- **Terminated state:** The process has finished execution.

Process Operations

---

1. Process Creation

In general-purpose systems, some way is needed to create processes as needed during operation. There are four principal events led to processes creation.

- System initialization.
- Execution of a process Creation System calls by a running process.
- A user request to create a new process.
- Initialization of a batch job.

Foreground processes interact with users. Background processes that stay in background sleeping but suddenly springing to life to handle activity such as email, webpage,

printing, and so on. Background processes are called daemons. This call creates an exact clone of the calling process.

A process may create a new process by some create process such as 'fork'. It choose to does so, creating process is called parent process and the created one is called the child processes. Only one parent is needed to create a child process. Note that unlike plants and animals that use sexual representation, a process has only one parent. This creation of process (processes) yields a hierarchical structure of processes like one in the figure. Notice that each child has only one parent but each parent may have many children. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings and the same open files. After a process is created, both the parent and child have their own distinct address space. If either process changes a word in its address space, the change is not visible to the other process.

Following are some reasons for creation of a process

- User logs on.
- User starts a program.
- Operating systems creates process to provide service, e.g., to manage printer.
- Some program starts another process, e.g., Netscape calls *xv* to display a picture.

2. Process Termination

A process terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased i.e., the PCB's memory space is returned to a free memory pool. The new process terminates the existing process, usually due to following reasons:

- **Normal Exist** Most processes terminates because they have done their job. This call is exist in UNIX.
- **Error Exist** When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
- **Fatal Error** An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
- **Killed by another Process** A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

3. Process States

A process goes through a series of discrete process states.

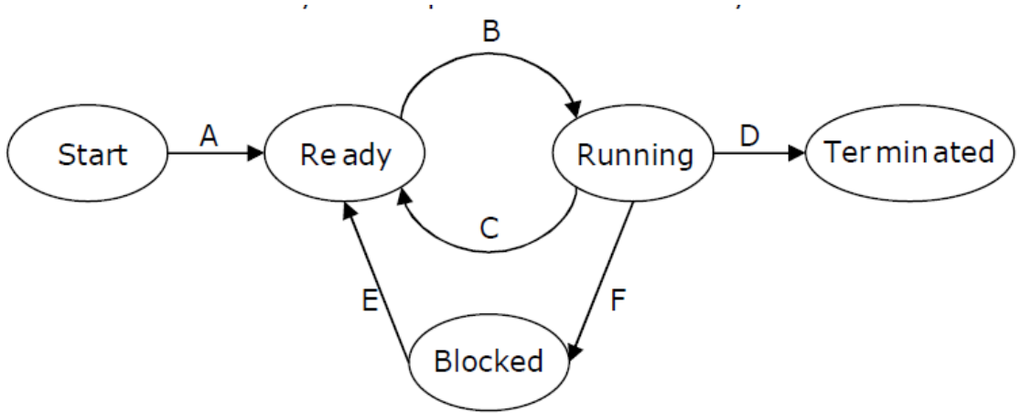
- **New State** The process being created.
- **Terminated State** The process has finished execution.
- **Blocked (waiting) State** When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.
- **Running State** A process is said t be running if it currently has the CPU, that is, actually using the CPU at that particular instant.
- **Ready State** A process is said to be ready if it use a CPU if one were available. It is runable but temporarily stopped to let another process run.

Logically, the 'Running' and 'Ready' states are similar. In both cases the process is willing to run, only in the case of 'Ready' state, there is temporarily no CPU available for it. The

'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

Process State Transitions:

Following are six(6) possible transitions among above mentioned five (5) states



FIGURE

- **Transition 1 (F)** occurs when process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.

  This state transition is:

  Block (process-name): Running → Block.

- **Transition 2 (C)** occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

  This state transition is:

  Time-Run-Out (process-name): Running → Ready.

- **Transition 3** (B) occurs when all other processes have had their share and it is time for the first process to run again

  This state transition is:

  Dispatch (process-name): Ready → Running.

- **Transition 4** (E) occurs when the external event for which a process was waiting (such as arrival of input) happens.

  This state transition is:

  Wakeup (process-name): Blocked → Ready.

- **Transition 5** (A)occurs when the process is created.

  This state transition is:

  Admitted (process-name): New → Ready.

- **Transition 6 (D)** occurs when the process has finished execution.
  This state transition is:
  Exit (process-name): Running → Terminated.

Process Control Block

A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor. The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

CPU/Process Scheduling

The assignment of physical processors to processes allows processors to accomplish work. **The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling**.

When more than one process is runnable, the operating system must decide which one first. **The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.**

Objectives of Scheduling/ Scheduling Policies:

In this section we try to answer following question: What the scheduler try to achieve?

The criteria that the schedulers generally use in attempting to optimize the system performance are:

1. To maximize CPU Utilization:

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. The CPU utilization is the average function of the clock time the CPU is busy executing user programs and the operating system functions.

2. Minimize Response Time :

A scheduler should minimize the response time for interactive user. Response time is important in time sharing and Real time operating system. It is defined as the time elapses from the moment

the last character of the command or transaction is entered until the first result or response appears on the terminal.

3. Minimizes Turnaround Time:

A scheduler should minimize the time batch users must wait for an output. Turnaround time is the time difference between job submission and job completion.

4. Maximize Throughput:

A scheduler should maximize the number of jobs processed per unit time.

5. Minimize Waiting Time:

Waiting time is essentially the time that a job spends waiting for resource allocation. It is the difference between the turnaround time and actual execution time of the job.

A little thought will show that some of these goals are contradictory. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

Preemptive Vs Nonpreemptive Scheduling:

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Non-preemptive Scheduling:

A scheduling discipline is non-preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process until it gets complete or goes in waiting state.

Following are some characteristics of non-preemptive scheduling

1. In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non-preemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.
3. In non-preemptive scheduling, a scheduler executes jobs in the following two situations.
    a. When a process switches from running state to the waiting state.
    b. When a process terminates.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can be taken away.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Scheduling Algorithms

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

Following are some scheduling algorithms we will study

1. FCFS Scheduling.
2. Round Robin Scheduling.
3. SJF Scheduling.
4. SRT Scheduling.
5. Priority Scheduling.
6. Multilevel Queue Scheduling.
7. Multilevel Feedback Queue Scheduling.

1. First-Come-First-Served (FCFS) Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

Perhaps, First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long.

The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

2. Round Robin Scheduling

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).

In the round robin scheduling, processes are dispatched in a FIFO manner but are given **a limited amount of CPU time called a time-slice or a quantum.**

If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

In any event, the average waiting time under round robin scheduling is often quite long.

3. Shortest-Job-First (SJF) Scheduling

Other name of this algorithm is Shortest-Process-Next (SPN).

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

The SJF algorithm favors short jobs (or processors) at the expense of longer ones.

The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

The best SJF algorithm can do is to rely on user estimates of run times.

> In the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their program will execute.

Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

4. Shortest-Remaining-Time (SRT) Scheduling

- The SRT is the preemtive counterpart of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- In SJF scheme, once a job begin executing, it run to completion.
- In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.
- The algorithm SRT has higher overhead than its counterpart SJF.
- The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.

- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

5. Priority Scheduling

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements,
  for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is ***aging***. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

6.Multilevel Queue Scheduling

A multilevel queue scheduling algorithm partitions the ready queue in several separate queues. In a multilevel queue scheduling processes are permanently assigned to one queues.

The processes are permanently assigned to one another, based on some property of the process, such as

- Memory size
- Process priority
- Process type

Algorithm choose the process from the occupied queue that has the highest priority, and run that process either

- Preemptive or
- Non-preemptively

Each queue has its own scheduling algorithm or policy.

Possibility I

If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes were all empty.

For example, in the above figure no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes will all empty.

Possibility II

If there is a time slice between the queues then each queue gets a certain amount of CPU times, which it can then schedule among the processes in its queue. For instance;

- 80% of the CPU time to foreground queue using RR.
- 20% of the CPU time to background queue using FCFS.

Since processes do not move between queue so, this policy has the advantage of low scheduling overhead, but it is inflexible.

7.Multilevel Feedback Queue Scheduling

Multilevel feedback queue-scheduling algorithm allows a process to move between queues. It uses many ready queues and associate a different priority with each queue.

The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively or unpreemptively. If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue. Note that this form of aging prevents starvation.

- A process entering the ready queue is placed in queue 0.
- If it does not finish within 8 milliseconds time, it is moved to the tail of queue 1.
- If it does not complete, it is preempted and placed into queue 2
- Processes in queue 2 run on a FCFS basis, only when queue 2 run on a FCFS basis, only when queue 0 and queue 1 are empty.

Multiprocessor Scheduling:

In multiprocessor system there are many processors. Here multiple CPUs are available, so scheduling problem is more complex. here we assume that the processors are identical in terms of their functionality; any available processor can then be run any process in the queue. We also assume UMA(Uniform Memory Access).

If several identical processors are available, then load sharing occurs. It could be possible to provide a separate queue for each processor. In this case if an empty queue is there then that processor will be idle while the other processors be busy. To prevent this situation a common ready queue is provided to processors where processes are scheduled to any available processor.

In such a scheme any one of the two scheduling approaches may be used. In one approach, each processor is self scheduled selecting a process from the common queue and executes it. As here, multiple processor are present there is possibility that more than one processor trying to access and update common data structure. To avoid this we must ensure that two processors do not choose the same process, and that the processes are not lost from the queue.

The another approach avoid this problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

In some systems all the scheduling activities are handled by the scheduler- the master server. The other processors only executes the user code. This asymmetric multiprocessing is simpler than symmetric multiprocessing as one processor access the system data structures, alleviating (making easy) the need for data sharing. Typically, asymmetric multiprocessing is implemented first within an operating system and is then upgraded to symmetric multiprocessing as the system evolves.