2.0 Process Management

2.1 Definition

A process has been given many definitions for instance:

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

There is no universally agreed upon definition, but the definition *Program in execution* seem to be most frequently used.

Process is not the same as program. A process is more than a program code. A process is an 'active' entity as opposed to program which is a 'passive' entity. A program is an algorithm expressed in some suitable notation, (e.g., programming language). Being 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 and a **data section** that contains global variables.

Also associated with a process is a ***thread*** which is a flow of execution through a process code, with its own program counter, systems register and stack. Therefore, OS can be classified as:

- ***Single process single threaded-*** only one process can execute at a time e.g DOS
- ***Multi-process single threaded –*** a single entity, the process is the basis of resource allocation and CPU scheduling e.g. UNIX
- ***Multi-process multithreaded –*** resources are allocated to the process but CPU scheduling can be in terms of threads e.g. windows, MAC, Solaris.

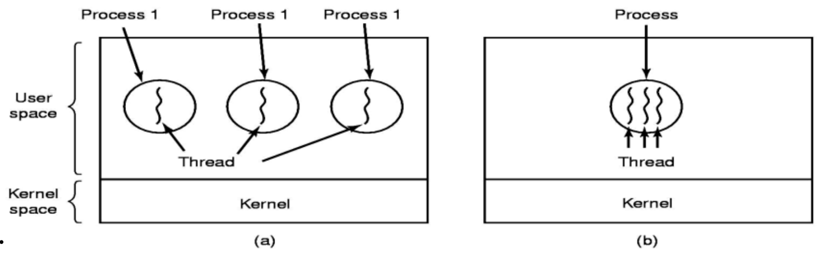


Illustration: (a) Three processes each with one thread (b) One process with three threads

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 (multiprogramming).

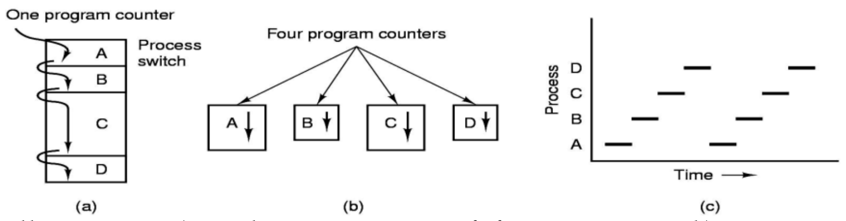


Illustration: a) Multiprogramming of four programs; b) Conceptual model of 4 independent, sequential processes; c) Only one program active at any instant.

To differentiate between single and multithreading, the term ***task*** is sometimes used to refer to a multithreaded entity while a ***heavy weight process*** refers to a task with one thread. Thread is sometimes called a ***lightweight process.*** On early batch systems, the term ***job*** was used to refer to one or more programs executed sequentially and treated by the system as a single entity.

2.2 Process Scheduling

On multiprocessing system, scheduling is used to determine which process is given control of the CPU. Scheduling may be divided into three phases:

Long term scheduling (job scheduling)

- LTS determines which ***jobs or processes*** may compete for the systems resources.
- Once the job scheduler makes a job active, it ***stays active until it terminates***.
- Main objective: to provide the medium-term scheduler with an appropriate number of jobs. Too few jobs and the CPU may sit idle because all jobs are blocked. Too many jobs and the memory management system may become overloaded, degrading system efficiency.

Medium-term scheduler (swapper)

- MTS ***Swaps processes into and out of memory***.
- It can allow more processes to share a system than can physically fit in memory.

Short-term scheduler (dispatcher)

- STS ***allocates the CPU to a process that is loaded into main memory and ready to run***.
- The dispatcher allocates the CPU for a fixed maximum amount of time.
- A process that must release the CPU after exhausting its time slot returns to the pool of processes from which the dispatcher selects the process to execute.

2.3 Process states

At its simplest, scheduling can consist of short-term scheduling. On such a system, a process exists in one of the four states.

- ***Ready:*** process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler)
- ***Blocked***: a process is waiting for some event to occur before it can continue executing e.g. waiting for I/O operation.
- ***Running***: A process that is currently being executed
- ***Terminated:*** either from the running state by completing its execution or by explicitly being killed. If a process is not removed from memory after entering this state, this state may also be called ***zombie***.

Medium-scheduling adds two process states:

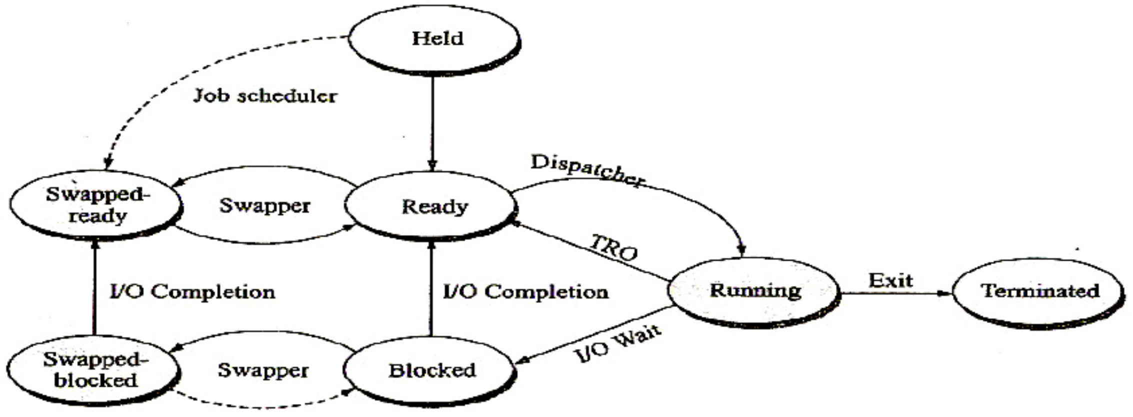
- ***Swapped-blocked***: a process that is waiting for some event to occur and has been swapped out to secondary memory.
- ***Swapped-ready:*** a process that is ready to run but is swapped out to secondary state. Such a process may not be allocated to the CPU since it is not in main memory.

Long term scheduling adds the following state:

- ***Held:*** a process that has been created will not be considered for into the memory for execution. Typically only new processes can become held processes. Once a process leaves the held state, it does not return to it.

Depending on the number of schedulers implemented, and therefore the number of states present, a number of different state models may be created to describe a system's behaviour.

The seven state model and transitions



The next table describes the various state transitions shown above.

TRANSITION		DESCRIPTION
FROM	**TO**	
Process creation	Held	Upon creation, a process is put in the held state.
Held	Swapped-Ready	Activate the process, making another process available for management by medium-term scheduler. Since there may be insufficient memory for another swapped-in process, held processes may become swapped-ready processes
Held	Ready	Same as previous
Swapped-Ready	Ready	Process swapped in when sufficient memory becomes available because of: processes releasing memory previously allocated to them, processes terminating, memory allocated to processes reduced due to changing system conditions
Swapped-Blocked	Blocked	System decides to swap in blocked processes (not all OS); blocked process doesn't execute anyway, so placing it in memory is a waste except for high priority processes or lightly loaded system with unused memory available-allows the process to execute sooner.
Ready	Swapped-Ready	MTS swaps out processes to release memory resources for other resources- due to other processes asking for additional memory or because amount of memory that must be allocated to various processes has increased due to changing system conditions
Blocked	Swapped-Blocked	Same as previous
Swapped-Blocked	Swapped-Ready	Process in one of the blocked states is waiting for an event to occur , then moves to the corresponding ready state
Blocked	Ready	Same as previous

Running	Ready	Processes allocated max CPU for max amount of time (time slice); if process does not become blocked or voluntarily release the CPU prior to the end of the its time slice, Timer Run Out (TRO) interrupt occurs, transferring control to OS and STS; process then moved to ready state. Also, transition of a high-priority process from a blocked to ready state will trigger STS to pre-empt a low-priority running process
Ready	Running	When no process is allocated to CPU, STS selects next running process from the pool of ready processes
Running	Blocked	Due to I/O operations, communicating with other processes, waiting for system conditions. CPU therefore allocated runnable processes and execution of suspended processes continues when these tasks are complete.
Running	Terminated	Due to illegal operation or normal completion of execution of process code
Any state	Terminated	Not illustrated in the diagram but a process can be terminated by actions external to it, regardless of the state it is in

2.4 Scheduling criteria

Different algorithms may favour different types of processes.

- **CPU utilization:** the CPU must be as busy as possible in performing different tasks; more important in real-time system and multi-programmed systems.
- **Throughput:** the number of processes executed in a specified time period is called throughput. It increases for short processes and vice versa.
- **Turnaround Time:** amount of time that is needed to execute a process; equal to **actual job time plus the waiting time.**
- **Waiting Time:** amount of time the process has waited. It is the turnaround time minus actual job time.
- **Response Time: a**mount of time between the time a request is submitted and the first response.
- **Predictability:** Lack of variability in other measures; users prefer consistency. For example, an interactive system that routinely responds within a second, but on occasion takes 10 seconds to respond, may be viewed more negatively than a system that consistently responds in 2s. Although average response time in the later system is greater, users may prefer the system with greater predictably
- **Fairness:** the degree to which all processes are given equal opportunity to execute. Do not allow a process to suffer from **starvation-** becomes stuck in a scheduling queue.
- **Priorities: g**ive preferential treatment to processes with higher priorities.

2.5 Scheduling algorithms

2.5.1 Non preemptive Scheduling (or run to completion) and Preemptive scheduling

Non preemptive Scheduling (or run to completion)

Once a process has been given the CPU, the CPU cannot be taken away from that process. Characteristics:

- Short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
- Response times are more predictable because incoming high priority jobs cannot displace waiting jobs.

- A scheduler executes jobs in the following two situations: when a process switches from running state to the waiting state and when a process terminates.

Preemptive Scheduling
When a process has been given the CPU, it can be taken away. It is used on today's interactive systems.

2.5.2 Common scheduling algorithms

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

- Also called: First-In-First-Out (FIFO) , Run-to-Completion, Run- Until-Done
- Processes are dispatched according to their arrival time on the ready queue.
- Being a non-pre-emptive discipline, once a process has a CPU, it runs to completion.
- 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.
- More predictable than most of other schemes but not useful in scheduling interactive users because it cannot guarantee good response time.
- Code for FCFS scheduling is simple to write and understand.
- One of the major drawbacks of this scheme is that the average time is often quite long.
- 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 algorithms.
- 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 pre-empted and given to the next process waiting in a queue. The pre-empted process is then placed at the back of the ready list.
- Preemptive (at the end of time-slice) and therefore effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.
- Major issue: ***length of the quantum***: setting the quantum too short causes too many ***context switches*** and lower the CPU efficiency while 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

- Or Shortest-Process-Next (SPN).
- ***Non-preemptive*** discipline in which ***waiting job (or process) with the smallest estimated run-time-to-completion is run next,*** i.e. when CPU is available, it is assigned to the process that has smallest next CPU burst.
- Appropriate for batch jobs for which the run times are known in advance.
- Gives the minimum average time for a given set of processes, probably optimal.
- Favours short jobs (or processors) at the expense of longer ones.
- ***Drawback***: requires precise knowledge of how long a job or process will run; this information is not usually available. The best SJF algorithm can do is to rely on user estimates of run times.
- Like FCFS, it's non preemptive therefore not useful in timesharing environment in which reasonable response time must be guaranteed.

4) *Shortest-Remaining-Time (SRT) Scheduling*

- ***Pre-emptive*** counterpart of SJF and useful in time-sharing environment.
- ***Process with the smallest estimated run-time to completion is run next, including new arrivals***.
- Once a job begins executing, it run to completion.
- A running process may be pre-empted by a new arrival process with shortest estimated run-time.
- Has higher overhead than its counterpart SJF.
- Must keep track of the elapsed time of the running process and must handle occasional Pre-emptions.
- In this scheme, small processes will run almost immediately; Longer jobs have even longer mean waiting time.

5) *Priority Scheduling*

- Each process is assigned a priority, and the ***one with higher 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 and 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 and Politics.

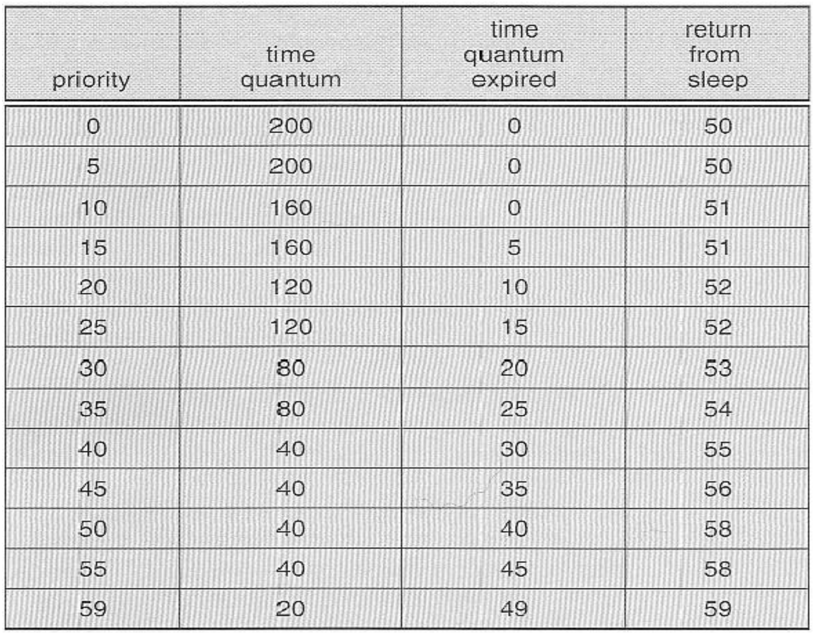
- Priority scheduling can be either pre-emptive or non-pre-emptive:
  - Pre-emptive priority algorithm will pre-emptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process
  - A non-pre-emptive priority algorithm will simply put the new process at the head of the ready queue.
- ***Drawback:*** 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.

2.5.3 Operating System examples

a) Example: Solaris Scheduling

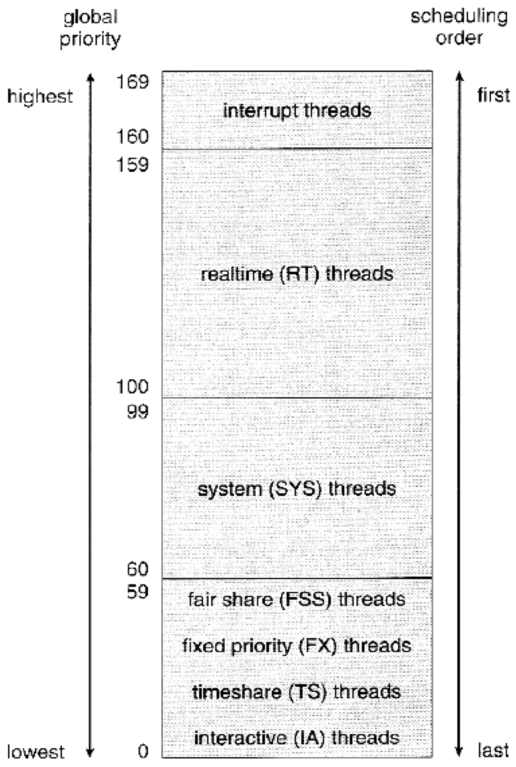
- Priority-based kernel thread scheduling.
- Four classes (real-time, system, interactive, and time-sharing), and multiple queues / algorithms within each class.
- Default is time-sharing.
  - Process priorities and time slices are adjusted dynamically in a multilevel-feedback priority queue system.
  - Time slices are inversely proportional to priority - Higher priority jobs get smaller time slices.
  - Interactive jobs have higher priority than CPU-Bound ones.

- See the table below for some of the 60 priority levels and how they shift. "Time quantum expired" and "return from sleep" indicate the new priority when those events occur. (Larger numbers are a higher, i.e. better priority.)



priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

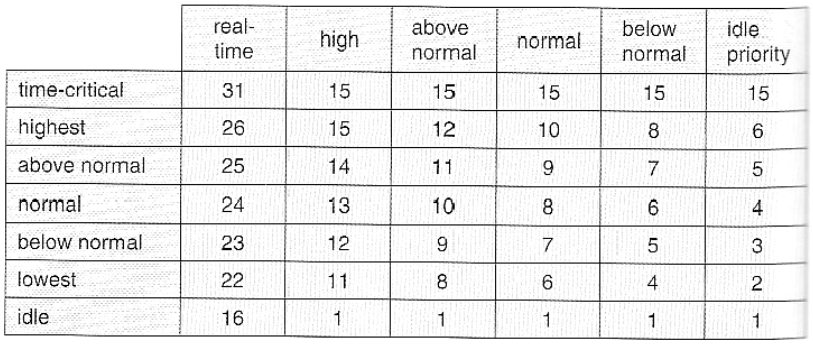
- Solaris 9 introduced two new scheduling classes: Fixed priority and fair share.
  - Fixed priority is similar to time sharing, but not adjusted dynamically.
  - Fair share uses shares of CPU time rather than priorities to schedule jobs. A certain share of the available CPU time is allocated to a project, which is a set of processes.
- System class is reserved for kernel use. (User programs running in kernel mode are NOT considered in the system scheduling class.)



Solaris scheduling

b) Example: Windows XP Scheduling

- Windows XP uses a priority-based preemptive scheduling algorithm.
- The dispatcher uses a 32-level priority scheme to determine the order of thread execution, divided into two classes -variable class from 1 to 15 and real-time class from 16 to 31, (plus a thread at priority 0 managing memory.)
- There is also a special idle thread that is scheduled when no other threads are ready.
- Win XP identifies 7 priority classes (rows on the table below), and 6 relative priorities within each class (columns.)
- Processes are also each given a base priority within their priority class. When variable class processes consume their entire time quanta, then their priority gets lowered, but not below their base priority.
- Processes in the foreground (active window) have their scheduling quanta multiplied by 3, to give better response to interactive processes in the foreground

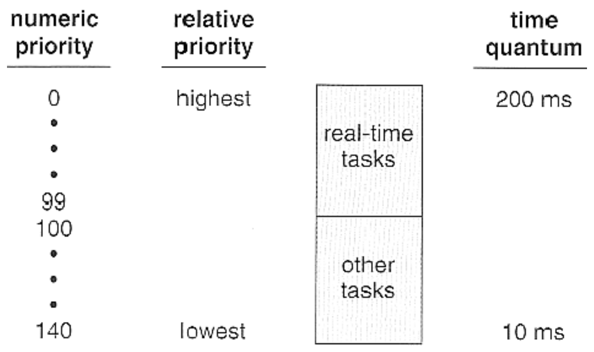


	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Windows XP priorities

c) Example: LINUX Scheduling

- Modern Linux scheduling provides improved support for SMP systems, and a scheduling algorithm that runs in O(1) time as the number of processes increases. [Symmetric Multiprocessing, SMP, is where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor]
- The Linux scheduler is a preemptive priority-based algorithm with two priority ranges - ***Real time*** from 0 to 99 and a ***nice*** range from 100 to 140.
- Unlike Solaris or XP, Linux assigns longer time quanta to higher priority tasks.

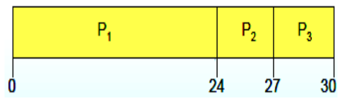


The relationship between priorities and time-slice length

- A runnable task is considered eligible for execution as long as it has not consumed all the time available in its time slice. Those tasks are stored in an ***active array,*** indexed according to priority.
- When a process consumes its time slice, it is moved to an ***expired array***. The tasks priority may be re-assigned as part of the transferal.
- When the active array becomes empty, the two arrays are swapped.
- These arrays are stored in ***runqueue*** structures. On multiprocessor machines, each processor has its own scheduler with its own run queue.

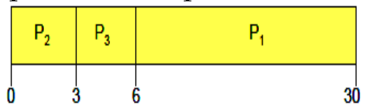
2.6 Scheduling Algorithm performance

- Performance of an algorithm for a given set of processes can be analysed if appropriate information about the processes is provided.
- Consider performance of *FCFS* algorithm for three compute-bound processes. Suppose that the processes arrive in the order: *P1, P2, P3.* The Gantt chart for the schedule is:



- Waiting time for *P1*= 0; *P2*= 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Average turnaround time: (24+27+30) = 27

- Suppose that the processes arrive in the order *P2, P3, P1.* The Gantt chart for the schedule is:



- Waiting time for *P1* =6*;P2*= 0*; P3* = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Turnaround Time = (3 + 6 + 30) = 13.
- Throughput =30 / 3 = 10.
- Much better than previous case
- *Convoy effect*- short process behind long process