1. Basic concept of Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

In multiprogramming systems, whenever two or more processes are simultaneously in the ready state, a choice has to be made which process to run next.

- The part of the OS that makes the choice is called the **scheduler**
- and the algorithm it uses is called the **scheduling algorithm**.

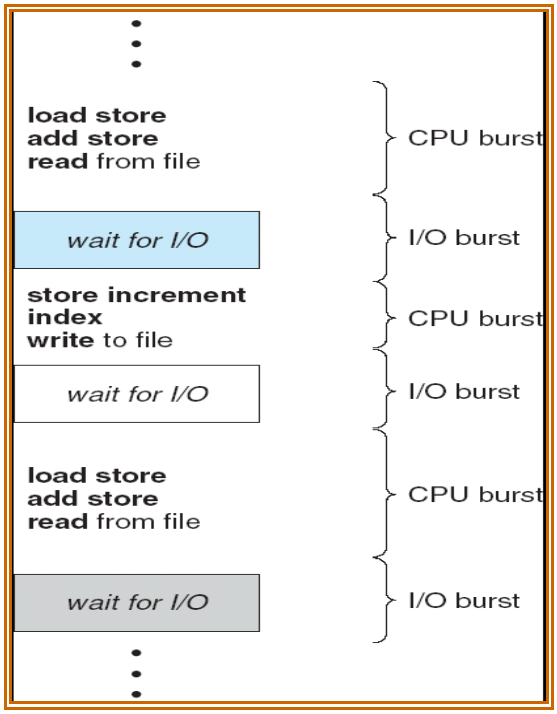
- In a single-processor system, only one process can run at a time though several processes are kept in memory; any others must wait until the CPU is free and can be rescheduled.
- Every time one process has to wait, another process can take over use of the CPU.
- Scheduling of this kind is a fundamental OS function. Almost all computer resources are scheduled before use.
- The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to OS design.
- Many of the same issues that apply to process scheduling also apply to thread scheduling, although some are different.

1.1 CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes:

- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.

Eventually, the final CPU burst ends with a system request to terminate execution



This distribution can be important in the selection of an appropriate CPU-scheduling algorithm

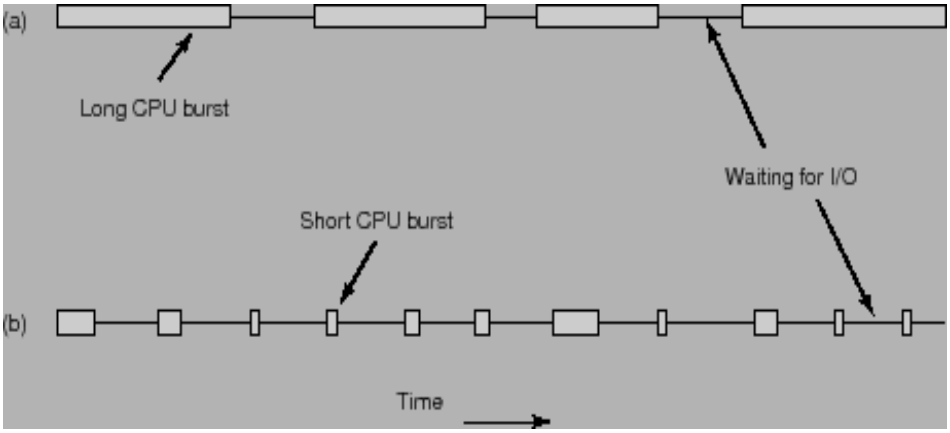


Figure: Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

- Some processes, such as the one in Fig. a, spend most of their time computing (CPU-bound), while others, such as the one in Fig. b, spend most of their time waiting for I/O (I/O-bound).
- Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then when they are finished loading and running all the I/O-bound jobs (a careful mix of processes).

1.3 CPU Scheduler

- Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Conceptually all the processes in the ready queue are lined up waiting for a chance to run on the CPU.

2. Preemptive and Non-preemptive

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O, on a semaphore, or for some other reason).
4. When a process terminates. If no process is ready, a system-supplied idle process is normally run.

- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3. When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive** or cooperative; otherwise, it is **pre-emptive**.
- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- Unfortunately, pre-emptive scheduling incurs a cost associated with access to shared data.
    - Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data.
    - Preemption also affects the design of the Operating system kernel
- A **non-preemptive scheduling algorithm** picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. First-Come-First-Served (FCFS), Shortest Job first (SJF).
- In contrast, a **pre-emptive scheduling algorithm** picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run. Round-Robin (RR), Priority Scheduling. Also shortest job first (SJF) algorithm could also be preemptive.

2.1 Example

The non-preemptive scheduling method was used by Windows3.x; Windows 95 introduce the preemptive scheduling method and all subsequent version of Windows operating system use the preemptive scheduling method.

2.2 Dispatcher

- Another component involved in the CPU-scheduling function is the **dispatcher**.
- The scheduler is concerned with deciding policy, not providing a mechanism. The dispatcher is the low-level mechanism (Responsibility: Context-switch).
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:
    - Switching context,
    - Switching to user mode,
- Jumping to the proper location in the user program to restart that program.
- The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

2.3 Scheduling criteria

- Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another.
- In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
  - **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
  - **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
  - **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.

    $T_r = T_s + T_w$
    T_s: Execution time.
    T_w: Waiting time.

  - **Waiting time**. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
  - **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced.
- A typical scheduler is designed to select one or more primary performance criteria and rank them in order of importance. One problem in selecting a set of performance criteria is that they often conflict with each other. For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time decreases.
- It is desirable to maximize **CPU utilization** and **throughput** and to minimize turnaround time, waiting time, and response time. (called as optimization criteria)
- In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.
- A scheduling algorithm that maximizes throughput may not necessarily minimize turnaround time.
  - Given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs.
  - If short jobs kept arriving at a steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

- Investigators have suggested that, for interactive systems (such as time-sharing systems), it is more important to minimize the **variance** in the response time than to minimize the **average** response time.

2.4 Goals of the scheduling algorithm under different circumstances

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Under all circumstances, fairness is important. Another general goal is keeping all parts of the system busy when possible. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second.

5. Scheduling algorithm

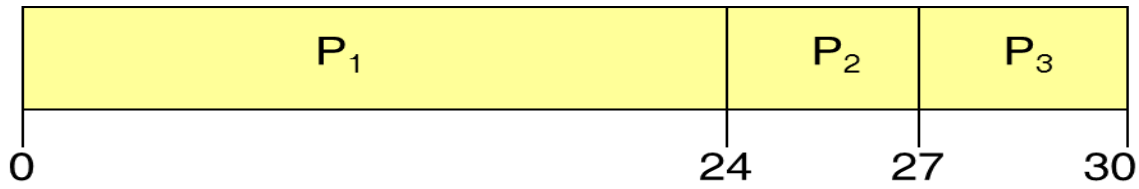
CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

5.1 First-Come, First-Serve Scheduling (FCFS)

- By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this algorithm, processes are assigned the CPU in the order they request it.
- Basically, there is a single queue of ready processes. Relative importance of jobs measured only by arrival time (poor choice).
- The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- The average waiting time under the FCFS policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

	Burst	Waiting	Turnaround
Process	Time	Time	Time
P_1	24	0	24
P_2	3	24	27
P_3	3	27	30
Average	-	17	27

- If the processes arrive in the order P_1,P_2 and P_3 are served in FCFS order, we get the result shown in the following chart:



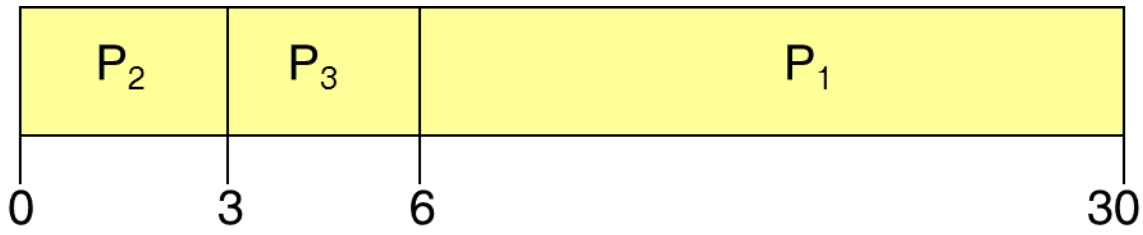
Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Disadvantage:

- There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. A long CPU-bound job may take the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods.
- This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Solution:

Suppose that the processes arrive in the order P_2, P_3, P_1
The Gantt chart for the schedule is:



Waiting time for P_1 = 6; P_2 = 0, P_3 = 3
Average waiting time: (6 + 0 + 3)/3 = 3
This approach is much better than previous case where the dispatcher select the shortest CPU burst allocate the CPU which lead to the non- preemptive Shortest Job First (SJF) scheduling.

5.2 Shortest-Job-First Scheduling

- A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used.
- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

	Burst	Waiting	Turnaround
Process	Time	Time	Time
P_1	6	3	9
P_2	8	16	24
P_3	7	9	16
P_4	3	0	3
Average	-	7	13

- Using SJF scheduling, we would schedule these processes according to the following chart: (P_4,P_1,P_3 and P_2)

- By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- The SJF scheduling algorithm gives the minimum average waiting time for a given set of processes.
  - Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process.
  - Consequently, the average waiting time decreases.

*The SJF algorithm can be either **pre-emptive** or **non-preemptive**. The choice arises when a new process arrives at the ready queue while a previous process is still executing.*

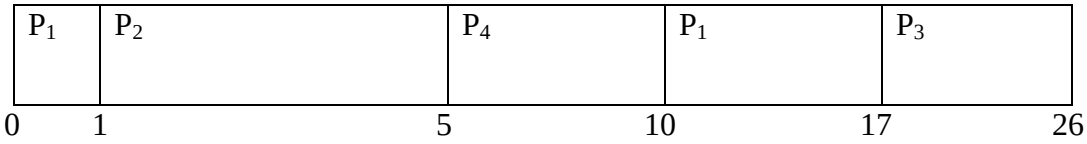
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A pre-emptive SJF algorithm will preempt the

currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

- Pre-emptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling. As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

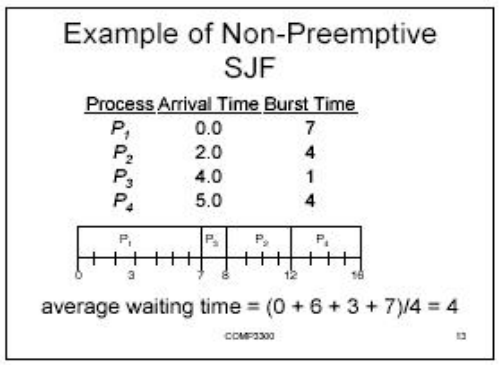
	Arrival	Burst	Waiting	Turnaround
Process	Time	Time	Time	Time
P_1	0	8	9	17
P_2	1	4	0	4
P_3	2	9	15	24
P_4	3	5	2	7
Average	-	6.5	13	

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting pre-emptive SJF schedule is as depicted in the following chart:



Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Some more examples:



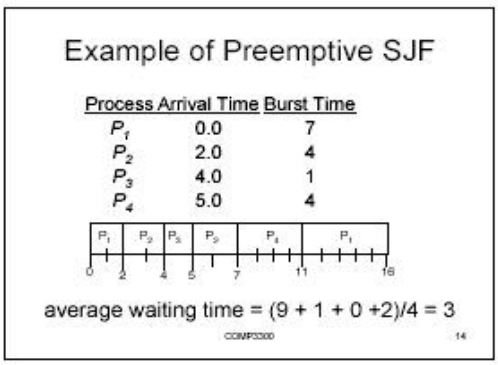


Figure: Example of non-pre-emptive SJF and pre-emptive SJF.

Advantage:

- SJF always gives minimum waiting time and turnaround time.
- Increase the throughput.

Disadvantage

- The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job.
- Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst.
- We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones.
- Every time new process will arrive, we have to check the shortest job.
- Also, long running jobs may starve for the CPU when there is a steady supply of short jobs.

5.3 Priority Scheduling

- The SJF algorithm is a special case of the general **priority scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. We use as low numbers represent high priority.
- As an example, consider the following set of processes, assumed to have arrived at time 0, in the order, P_1, P_2, P_3, P_4, P_5 with the length of the CPU burst given in milliseconds:

	Burst		Waiting	Turnaround
Process	Time	Priority	Time	Time
P_1	10	3	6	16
P_2	1	1	0	1
P_3	2	4	16	18
P_4	1	5	18	19
P_5	5	2	1	6
Average	-	-	8.2	12

- Using priority scheduling, we would schedule these processes according to the following chart:

- Priorities can be defined either **internally** or **externally**.
  - Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.
  - External priorities are set by criteria outside the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Priority scheduling can be either **pre-emptive** or **non-preemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
  - A pre-emptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  - A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

5.3.1 Starvation

- A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked.
  - A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
  - In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

5.3.2 A solution to indefinite blocking

A solution to the problem of indefinite blockage of low-priority processes is **aging**.
Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

- For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

5.3.3 Setting the priority of process

You can set the process nice value using nice and adjust it using renice. The nice command increases the nice value of a process by 10, giving it a lower priority.
You can view the nice values of active processes using the –l or –f (for long output) option to ps.
The value you are interested in is shown in the NI (nice) column.
$ **ps -l**

```
$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S   500  1259  1254  0  75   0 -   710 wait4  pts/2    00:00:00 bash
000 S   500  1262  1251  0  75   0 -   714 wait4  pts/1    00:00:00 bash
000 S   500  1313  1262  0  75   0 -  2762 schedu pts/1    00:00:00 emacs
000 S   500  1362  1262  2  80   0 -   789 schedu pts/1    00:00:00 oclock
000 R   500  1363  1262  0  81   0 -   782 -      pts/1    00:00:00 ps
```

Here you can see that the `oclock` program is running (as process 1362) with a default nice value. I had been started with the command

```
$ nice oclock &
```

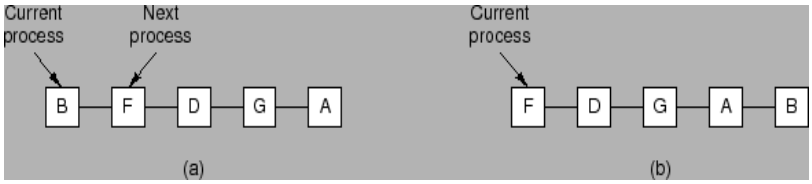
It would have been allocated a nice value of +10. If you adjust this value with the command

$ **renice 10 1362**

1362: old priority 0, new priority 10

5.4 Round-Robin Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes.
- A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue.
- To implement RR scheduling,
    - We keep the ready queue as a FIFO queue of processes.
    - New processes are added to the tail of the ready queue.
    - The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
    - The process may have a CPU burst of less than 1 time quantum.
        - In this case, the process itself will release the CPU voluntarily.
        - The scheduler will then proceed to the next process in the ready queue.
    - Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
        - the timer will go off and will cause an interrupt to the OS.
        - A context switch will be executed, and the process will be put at the tail of the ready queue.
        - The CPU scheduler will then select the next process in the ready queue.

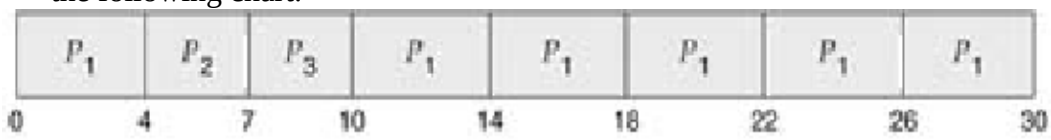


Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

- The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

	Burst	Waiting	Turnaround
Process	Time	Time	Time
P_1	24	6	30
P_2	3	4	7
P_3	3	7	10
Average	-	5.66	15.66

- Using round-robin scheduling, we would schedule these processes according to the following chart:

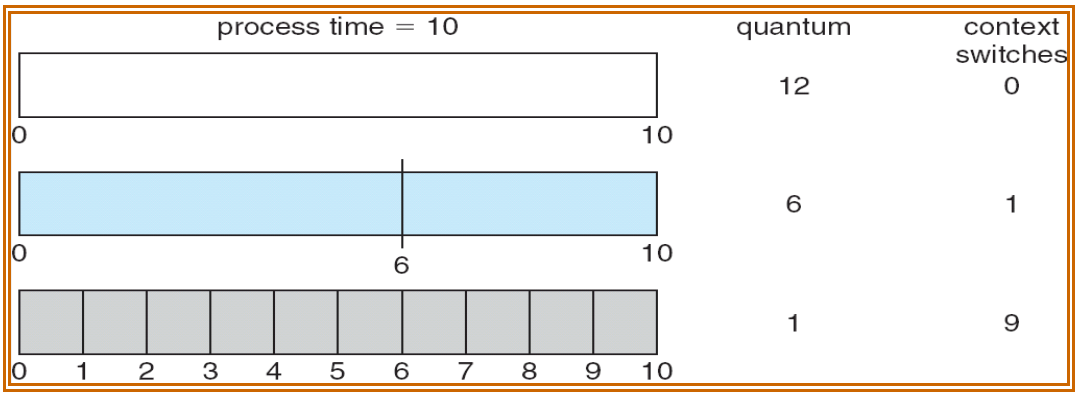


- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process).
- If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus pre-emptive.

    - If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
    - Each process must wait no longer than (n-1)*q time units until its next time quantum.
    - For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

- The performance of the RR algorithm depends heavily on the size of the time quantum.
    - If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
    - If the time quantum is extremely small (say, 1 millisecond), the RR approach is called **processor sharing** and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor.

- We need also to consider the effect of context switching on the performance of RR scheduling Switching from one process to another requires a certain amount of time for doing the saving and loading registers and memory maps, updating various tables and lists, flushing and reloading the memory cache, etc.
  - Let us assume that we have only one process of 10 time units.
  - If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
  - If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
  - If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly



- Thus, we want the time quantum to be large with respect to the context-switch time.
  - If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.
  - In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds.
  - The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.
- Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.

6. Approaches to Multiple-Processor Scheduling

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor - the master server.
  - The other processors execute only user code.
  - This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.
- A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling.
  - All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

    - Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
- if we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.
- Virtually all modern OSs support SMP.

6.1 Processor affinity

The data most recently accessed by the process populate the cache for the processor; and as a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The content of cache memory must be invalidated for the first processor and cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes on the same processor. This is known as processor affinity-that is, a process has an affinity for the processor on which it is currently running.

7. Load Balancing

- On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU.
- Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.
    - Load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute.
    - On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.
- It is important to note that in most contemporary OSs supporting SMP, each processor does have a private queue of eligible processes.
- There are two general approaches to load balancing: **push migration** and **pull migration**.
    - With push migration, a specific task periodically checks the load on each processor and -if it finds an imbalance- evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
    - Pull migration occurs when an idle processor pulls a waiting task from a busy processor.
    - Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.
- For example, the Linux scheduler and the ULE scheduler available for FreeBSD systems implement both techniques.
- Linux runs its load balancing algorithm every 200 milliseconds (push migration) or whenever the run queue for a processor is empty (pull migration).

8. Algorithm Evaluation

First, choose the criteria you want to use to judge the algorithm. E.g.

- Maximize the CPU utilization under the constraint that the maximum response time is 1 second.
- Maximize throughput such that turnaround time is on average linearly proportional to total execution time.

Next, evaluate the algorithms under consideration.
Methods of evaluation:

- Analytic evaluation (Deterministic modeling)
- Queueing models
- Simulations
- Implementation

8.1 Deterministic Modeling

- Analytic Evaluation is a method in which a given algorithm and system workload are used to produce a formula or number that evaluates the performance of the algorithm for that particular workload.
- Deterministic modeling is one type of analytic evaluation.

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- Compute the average wait time for FCFS, SJF and RR (quantum = 10 msec)
- Which algorithm gives the minimum wait time?
  - FCFS: 28 msec
  - SJF: 13 msec
  - RR: 23 msec
- SJF gives the shortest wait time (as expected, since it is optimal for shortest wait time).

Pros and Cons of Deterministic Modeling

- **Advantages of deterministic modeling**
  - Simple and fast
  - Gives an exact number
  - Makes it easy to compare algorithms
- **Disadvantages:**
  - Requires exact numbers for input.
  - Answers only apply to the specific case tested.
  - Does not give a general answer (too specific).

8.2 Queueing Models

- Network queueing analysis models the system as a set of servers with associated queues of waiting processes:
  - CPU and ready queue
  - I/O system and device queues

- etc.

- This method uses information about the distribution of CPU and I/O bursts (determined by measurement or estimation).
- The method also requires a distribution of process arrival times (either measured or estimated).
- Using the above information, queueing analysis can be used to compute CPU utilization, average queue length, average wait time, etc.

8.2.1 Little's Formula

- At steady state, the same number of processes are arriving in a queue as are leaving the queue.
- In this case, Little's formula applies:
- $n = \lambda W$

where:

n = the average queue length
W = the average wait time
λ = the average arrival rate of processes

If one knows two of the above variables, one can compute the third.

Limitations of Queueing analysis

- There are limited classes of algorithms and distributions that can be handled by queueing analysis.
- Mathematics can be difficult to work with.
  - Distributions are often defined in mathematically tractable ways that are unrealistic.
- To work with the mathematics, inaccurate assumptions may be made.
- The theory therefore only yields approximate answers.

8.3 Simulations

- Simulations are programs that models the components of the computer system. (e.g. a variable is used to represent the system clock).
- The simulator modifies the system state to reflect the activities of the devices, scheduler and processes.
- The simulator keeps track of statistics about system performance.
- Input data (arrival times and burst times) can be generated by either:
  - A random number generator
  - A trace tape (a record of arrival and burst times on an actual system).
- Drawbacks of simulation:
  - Simulation is expensive. It can take hours of computing time.
  - More detail yields more accurate results, but takes more time to compute.