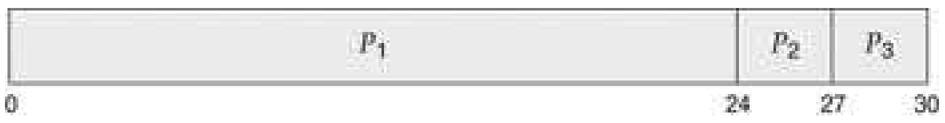
0.1 Scheduling Algorithms

0.1.1 First-Come, First-Served Scheduling

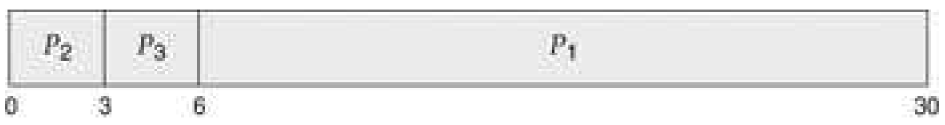
- By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this algorithm, processes are assigned the CPU in the order they request it.
- Basically, there is a single queue of ready processes. Relative importance of jobs measured only by arrival time (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:

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

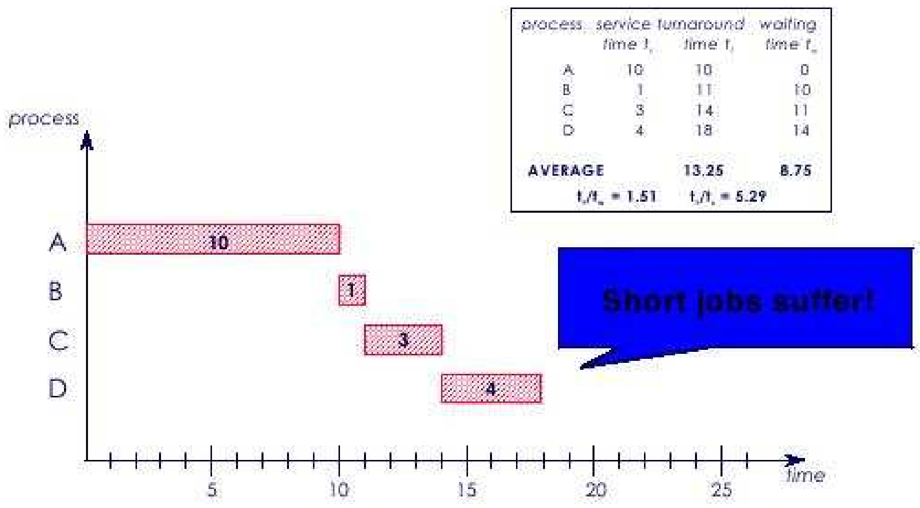
- If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following chart:



- If the processes arrive in the order P_2, P_3, P_1, the results will be as shown in the following chart:



- The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds.
- This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.
- Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system (dynamic system), the following scenario may result.
  - The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/0 and will move into the ready queue, waiting for the CPU.
  - While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/0 device.
  - All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/0 queues.
  - At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU.
  - Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done.
  - 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.
- Another Example: Suppose that there is one CPU-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete.
  - The CPU-bound process runs for 1 sec, then it reads a disk block.
  - All the I/O processes now run and start disk reads.
  - When the CPU-bound process gets its disk block, it runs for another 1 sec, followed by all the I/O-bound processes in quick succession.
  - The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish.



process	service time t_s	turnaround time t_r	waiting time t_w
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
AVERAGE		13.25	8.75
	$t_r/t_s = 1.51$	$t_r/t_s = 5.29$	

Figure 1: An example to First-Come First-Served.

  - With a scheduling algorithm that preempted the CPU-bound process every 10 msec, the I/O-bound processes would finish in 10 sec instead of 1000 sec, and without slowing down the CPU-bound process very much.

- The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0.

0.1.2 Shortest-Job-First Scheduling

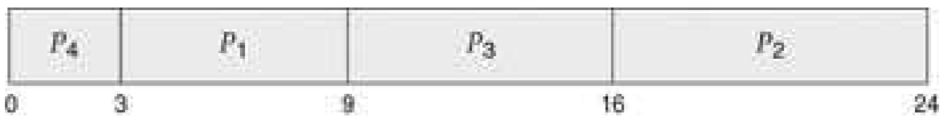
- A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst.

- 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:

Process	Burst Time	Waiting Time	Turnaround 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:



- 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 a long one decreases 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 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 can not 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.
- Also, long running jobs may starve for the CPU when there is a steady supply of short jobs.

- Example: In Fig. 2a, the average turnaround time is 14 minutes. Consider running these four jobs using SJF, as shown in Fig. 2b, the average turnaround time now becomes 11 minutes.

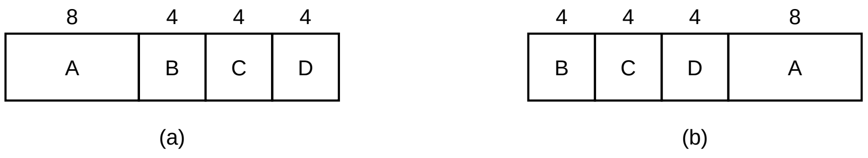
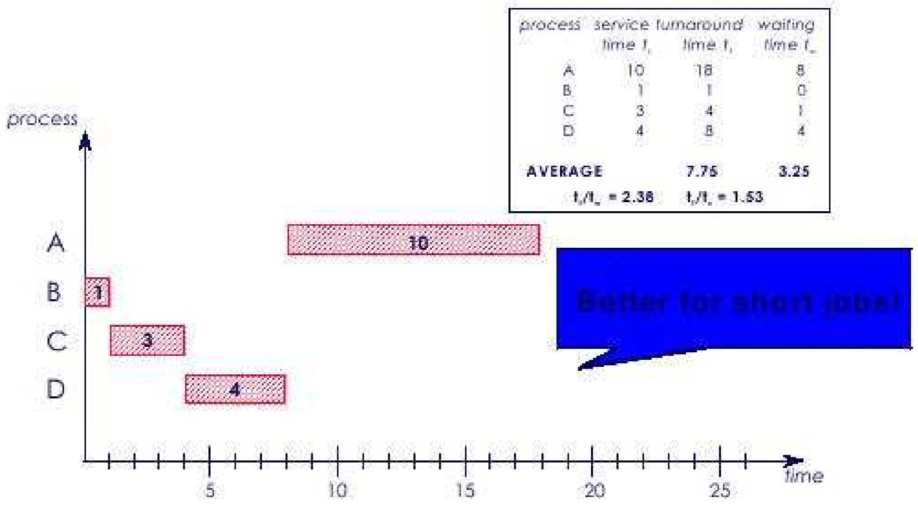


Figure 2: An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.



process	service time t_s	turnaround time t_r	waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
AVERAGE		**7.75**	**3.25**
	$t_r/t_s = 2.38$	**$t_r/t_s = 1.53$**	

Figure 3: An example to Shortest Job First.

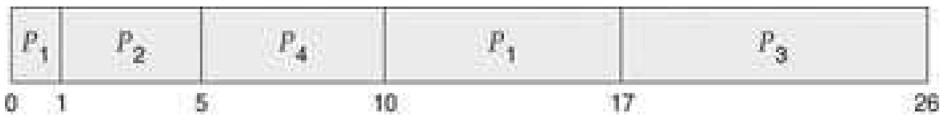
- The SJF algorithm can be either **pre-emptive** or **nonpreemptive**. The choice arises when a new process arrives at the ready queue while a previous process is still executing.

- 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.

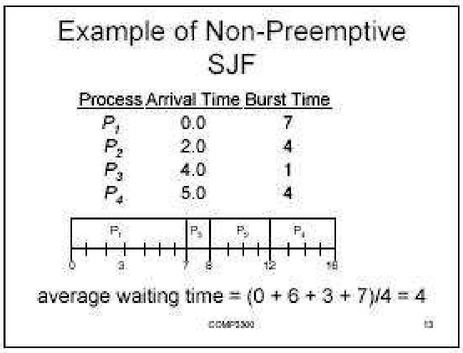
- Pre-emptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling. As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time	Waiting Time	Turnaround 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:



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



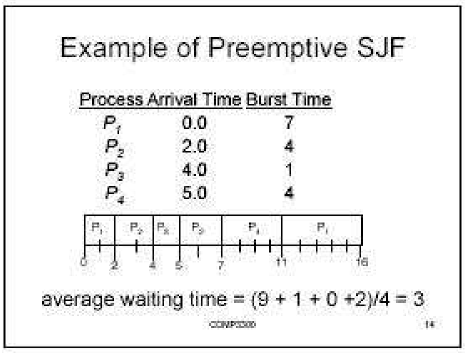
Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

P_1
P_3
P_2
P_4
0
3
7
8
12
16

average waiting time = (0 + 6 + 3 + 7)/4 = 4

COMP3300 13



Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

P_1
P_2
P_3
P_2
P_4
P_1
0
2
4
5
7
11
16

average waiting time = (9 + 1 + 0 +2)/4 = 3

COMP3300 14

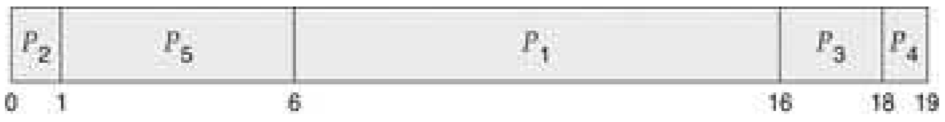
Figure 4: Example of non-pre-emptive SJF and pre-emptive SJF.

0.1.3 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 $P_1, P_2, \ldots P_5$, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority	Waiting Time	Turnaround 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 **nonpreemptive**. 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 nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

- 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.

- It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 5 shows a system with four priority classes.

- The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower priority classes, if priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower priority classes may all starve to death.

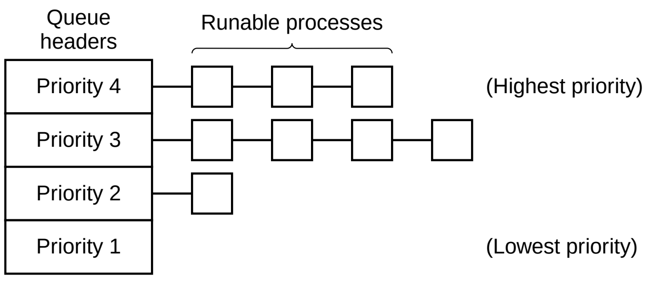
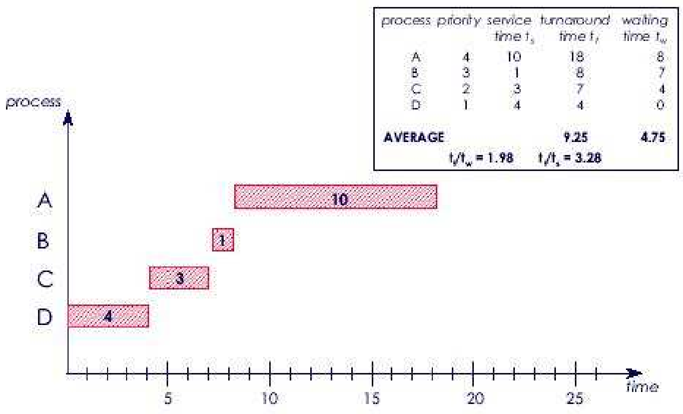


Figure 5: A scheduling algorithm with four priority classes.

- 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.



process	priority	service time t_s	turnaround time t_r	waiting time t_w
A	4	10	18	8
B	3	1	8	7
C	2	3	7	4
D	1	4	4	0
AVERAGE			**9.25**	**4.75**
		$t_r/t_w = 1.98$	**$t_r/t_s = 3.28$**	

Figure 6: An example to Priority-based Scheduling.

0.1.4 Round-Robin Scheduling

- The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption 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.

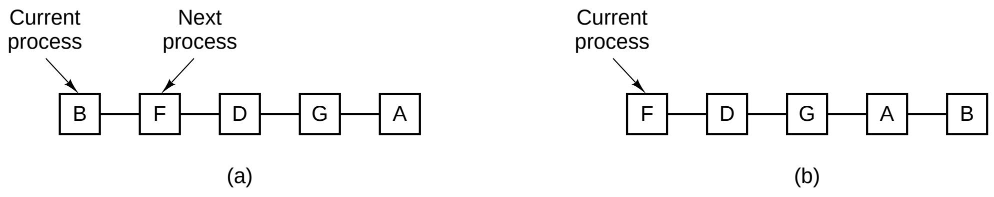
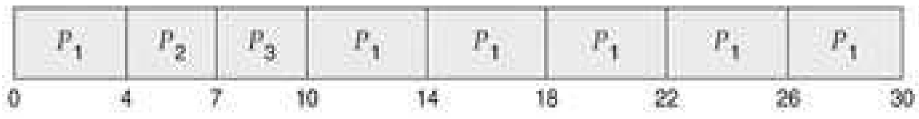


Figure 7: Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

- 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)

Process	Burst Time	Waiting Time	Turnaround 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 $\frac{1}{n}$ 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 $\frac{1}{n}$ the speed of the real processor.

- We need also to consider the effect of context switching on the performance of RR scheduling (see Fig. 8). Switching from one process to another requires a certain amount of time for doing the administration'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

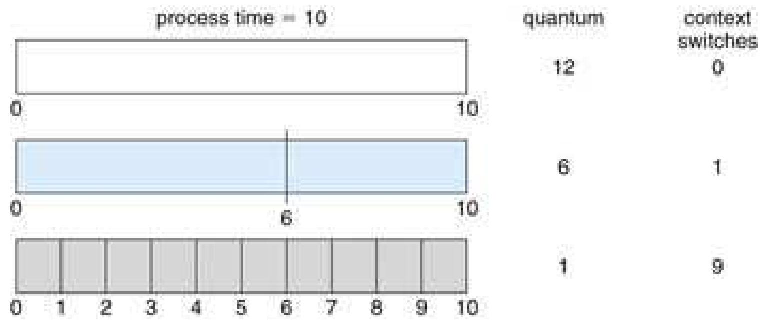
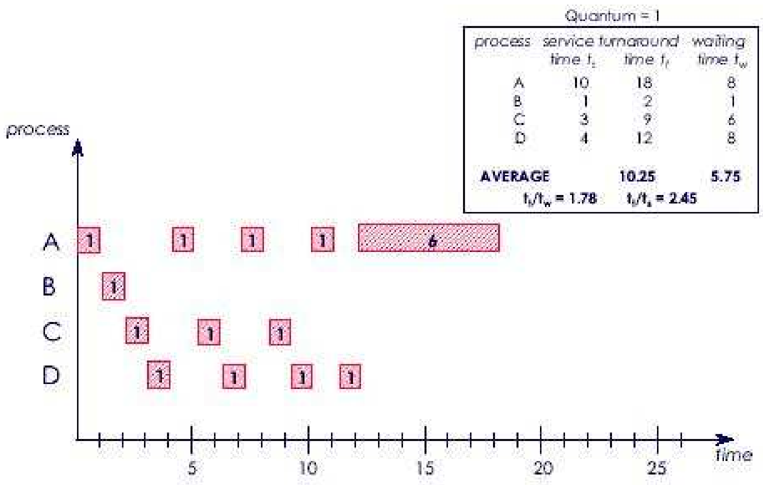


Figure 8: The way in which a smaller time quantum increases context switches.

- 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.
- Poor average waiting time when job lengths are identical; Imagine 10 jobs each requiring 10 time slices, all complete after about 100 time slices, even FCFS is better!
- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. If context-switch time is added in, the average turnaround time increases for a smaller time quantum, since more context switches are required.
- Although the time quantum should be large compared with the context-switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to FCFS policy.



Quantum = 1

process	service time t_s	turnaround time t_t	waiting time t_w
A	10	18	8
B	1	2	1
C	3	9	6
D	4	12	8
AVERAGE		**10.25**	**5.75**
	$t_t/t_w = 1.78$	$t_t/t_s = 2.45$	

Figure 9: An example to Round Robin.

0.1.5 Multilevel Queue Scheduling

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
- A common division is made between **foreground (interactive)** processes and **background (batch)** processes.

  - These two types of processes have different response-time requirements and so may have different scheduling needs.
  - In addition, foreground processes may have priority (externally defined) over background processes.

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (see Fig. 10).

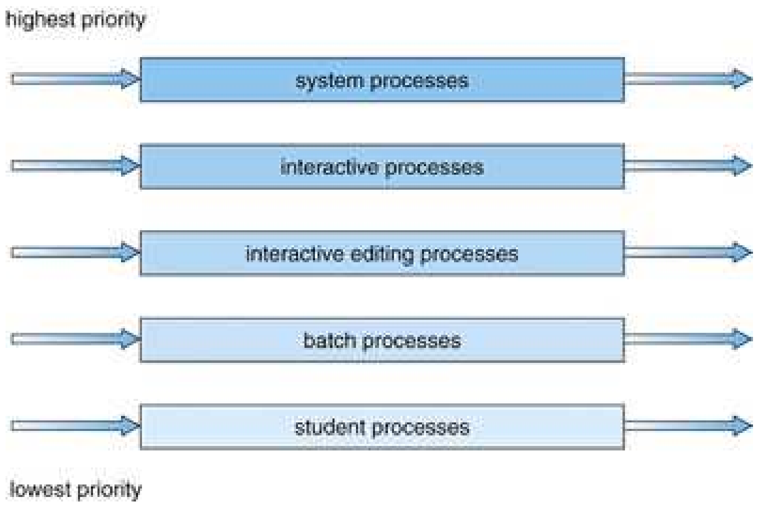


Figure 10: Multilevel queue scheduling.

- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.

- Each queue has absolute priority over lower-priority queues and also each queue has its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

0.1.6 Multilevel Feedback-Queue Scheduling

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
- This setup has the advantage of low scheduling overhead, but it is inflexible. The **multilevel feedback-queue scheduling algorithm**, in contrast, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts.
  - If a process uses too much CPU time, it will be moved to a lower-priority queue.
  - This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
  - In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

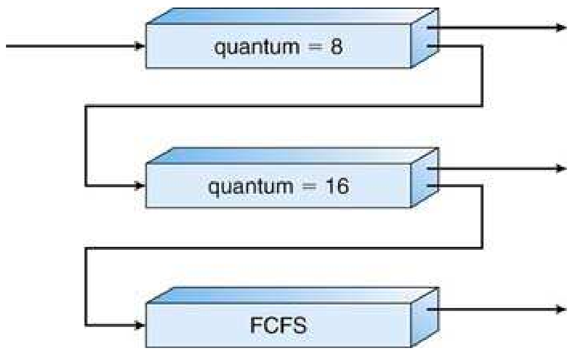


Figure 11: Multilevel feedback queues.

- This form of **aging** prevents starvation (see Fig. 11).
  - A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds.
  - If it does not finish within this time, it is moved to the tail of queue 1.

- If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
- If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

- In general, a multilevel feedback-queue scheduler is defined by the following parameters:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The method used to determine when to upgrade a process to a higher-priority queue.
  - The method used to determine when to demote a process to a lower-priority queue.
  - The method used to determine which queue a process will enter when that process needs service.
- The definition of a multilevel feedback-queue scheduler makes it the most general CPU-scheduling algorithm. Unfortunately, it is also the most complex algorithm.

0.2 Multiple-Processor Scheduling

0.2.1 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 - <u>the master server</u>.
  - The other processors execute only user code.
  - This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, <u>reducing the need for data sharing</u>.
- A second approach uses **symmetric multiprocessing (SMP)**, where <u>each processor is self-scheduling</u>.
  - 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.

0.2.2 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).

0.3 Operating System Examples

0.3.1 Example: Linux Scheduling

- The Linux scheduler is a pre-emptive, priority-based algorithm with two separate priority ranges:
  - a real-time range from 0 to 99
  - and a nice value ranging from 100 to 140.
- These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.
- Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta. The relationship between priorities and time-slice length is shown in Fig. 12.

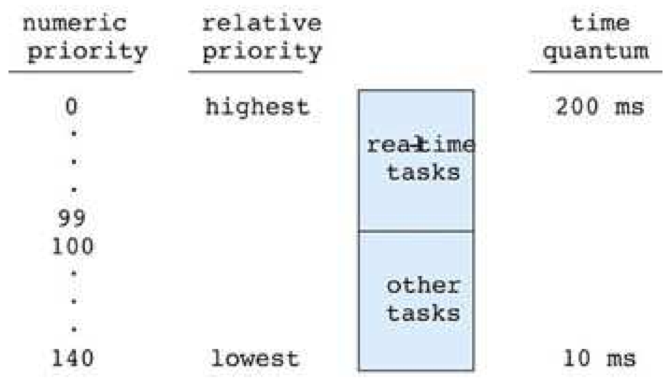


Figure 12: The relationship between priorities and time-slice length.

- A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta.

- The kernel maintains a list of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently.

- Each runqueue contains two priority arrays -**active** and **expired**.

  - The active array contains all tasks with time remaining in their time slices,
  - and the expired array contains all expired tasks.

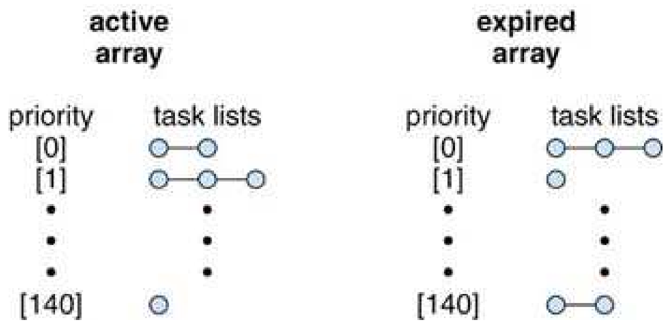


Figure 13: List of tasks indexed according to priority.

- Each of these priority arrays contains a list of tasks indexed according to priority (see Fig. 13).

  - The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure.
  - When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.

1 Process Synchronization

- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

- Concurrent access to shared data may result in data inconsistency.

- We discuss various mechanisms to ensure the orderly execution of *co-operating processes* that share a logical address space, so that data consistency is maintained.

1.1 Race Condition

- A potential problem; the order of instructions of cooperating processes (see Table 1).

Table 1: Race Condition

Process A	**Process B**	**concurrent access**
X = 1;	**Y = 2;**	*does not matter*
X = Y + 1;	**Y = Y * 2;**	*important!*

- Producer-consumer problem. It is described that how a bounded buffer could be used to enable processes to share memory
  - **Bounded buffer problem**. The solution allows at most $BUFFER\_SIZE - 1$ items in the buffer at the same time.
  - An integer variable *counter*, initialized to 0. *counter* is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- The code for the producer process:

```
while (true)
 {
   /* produce an item in next Produced */
   while (counter == BUFFER_SIZE)
     ; /* do nothing */
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
 }
```

- The code for the consumer process:

```
while (true)
 {
   while (counter == 0)
     ; /* do nothing */
   nextConsumed = buffer [out] ;
   out = (out + 1) % BUFFER_SIZE;
   counter--j;
   /* consume the item in nextConsumed */
 }
```

- Although both the producer and consumer routines are correct separately, *they may not function correctly when executed concurrently*.
- We would arrive at incorrect state because we allowed both processes to manipulate the variable *counter* concurrently.
- A ***race condition*** is a situation where two or more processes access shared data concurrently and final value of shared data depends on *timing* (*race* to access and modify data)
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter (process synchronization).

1.2 The Critical-Section Problem

- How do we avoid *race conditions*? What we need is **mutual exclusion** (see Fig. 14).

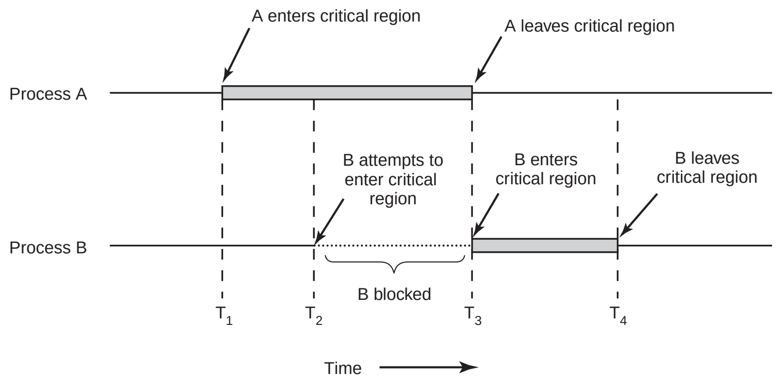


Figure 14: Mutual exclusion using critical regions.

- Consider a system consisting of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$. Each process has a segment of code, called a **critical section (CS)**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its CS, no other process is to be allowed to execute in its CS.
- That is, no two processes are executing in their CSs at the same time.
- Each process must request permission to enter its CS. The section of code implementing this request is the **entry section**.
- The CS may be followed by an **exit section**.
- The remaining code is the **remainder section** (see Fig. 15).

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Figure 15: General structure of a typical process P_i.

- A solution to the CS problem must satisfy the following requirements:
  1. **Mutual exclusion**. If process P_i is executing in its CS, then no other processes can be executing in their CSs.
  2. **Progress**. If no process is executing in its CS and some processes wish to enter their CSs, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its CS next, and this selection cannot be postponed indefinitely. (No process should have to wait forever to enter its CS.)

3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.
4. **Fault tolerance**. Process running outside its CR should not block other processes accessing the CR.
5. **No assumptions** may be made about speeds or the number of CPUs.

- **Atomic operation**. Atomic means either an operation happens in its entirely or NOT at all (it cannot be interrupted in the middle). Atomic operations are used to ensure that cooperating processes execute correctly.

- Machine instructions are atomic, high level instructions are not (count++; this is actually 3 machine level instructions, an interrupt can occur in the middle of instructions).

- Various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its CS, no other process will enter its CS and cause trouble.

  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's Solution
  - The TSL instructions (Hardware approach)

1.2.1 Disabling Interrupts: (Systems approach)

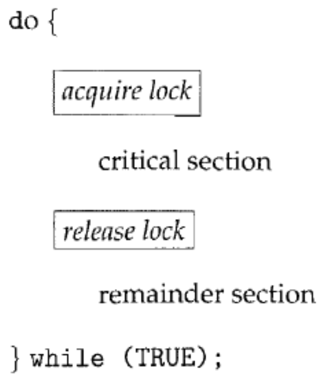
- The simplest solution is to have each process disable all interrupts just after entering its CS and re-enable them just before leaving it.

- With interrupts disabled, no clock interrupts can occur (The CPU is only switched from process to process as a result of clock or other interrupts)

- With interrupts turned off the CPU will not be switched to another process!! Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it and never turned them on again? That could be the end of the system.
- On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists.

1.2.2 Lock Variables: (Software approach)



```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Figure 16: Solution to the critical-section problem using locks.

- Consider having a single, shared (lock) variable, initially 0.
  - When a process wants to enter its CS, it first tests the lock.
  - If the lock is 0, the process sets it to 1 and enters the CS.
  - If the lock is already 1, the process just waits until it becomes 0.
  - Thus, a 0 means that no process is in its CS, and a 1 means that some process is in its CS.
- Unfortunately, this idea contains a fatal flaw;
  - Suppose that one process reads the lock and sees that it is 0.
  - Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1.
  - When the first process runs again, it will also set the lock to 1, and two processes will be in their CSs at the same time.

1.3 Mutual Exclusion with Busy Waiting

1.3.1 Strict Alternation (Software approach)

Busy waiting (notice the semicolons terminating the while statements in Fig. 17); continuously testing a variable until some value appears, a lock that uses busy waiting is called a **spin lock**. It should usually be avoided, since it wastes CPU time.

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}

                (a)
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}

                (b)
```

Figure 17: A proposed solution to the critical region problem. (a) Process 0. (b) Process 1.

- the integer variable **turn** (keeps track of whose turn it is to enter the CR),
- initially, process 0 inspects turn, finds it to be 0, and enters its CR,
- process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes,
- when process 0 leaves the CR, it sets turn to 1, to allow process 1 to enter its CR,
- suppose that process 1 finishes its CR quickly, so both processes are in their nonCR (with turn set to 0)
- process 0 finishes its nonCR and goes back to the top of its loop. Process 0 executes its whole loop quickly, exiting its CR and setting turn to 1.
- at this point turn is 1 and both processes are executing in their nonCR,
- process 0 finishes its nonCR and goes back to the top of its loop,
- unfortunately, it is not permitted to enter its CR, **turn** is 1 and process 1 is busy with its nonCR,
- it hangs in its while loop until process 1 sets turn to 0,
- this algorithm does avoid all races. But violates condition **Fault tolerance**.