Interprocess Communication
Interprocess Communication

- IPC provides a mechanism to allow processes to communicate and to synchronize their action without sharing the same address space.
- IPC is particularly useful in distributed environment where the communication processes may reside on different computers connected with a network.
- An example is Chat program used on the world wide web (WWW).
• **Multiprocessing**: - a mode of operation that provides for parallel processing by two or more processors of a multiprocessor.

• **Multiprocessor**: - a computer that two or more processors that have common access to a main storage.

• **Multiprogramming**: - a mode of operation that provides for the interleaved execution of two or more computer programs by a single processor. The same as multitasking, using different terminology.
Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items
  - They do so in a way where the final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.
**Explanation**

- A race condition occurs when multiple processes or threads read and write data items so that final result depend on the order of execution of instruction in the multiple process.

- Let us see example.
Example

- Two processes, $P_1$ and $P_2$, share global variable $a$.
- At some point in its execution, $P_1$ updates $a$ to the value 1 and later on process $P_2$ updates $a$ to the value 2.
- So, the two tasks are in a race to write variable $a$.
- In this example, the “loser” of the race (the process that updates last) determines the final value of $a$. 
Another example

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Concurrent access</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1</td>
<td>B = 2</td>
<td>Does not matter</td>
</tr>
<tr>
<td>A = B + 1</td>
<td>B = B * 2</td>
<td>Important</td>
</tr>
<tr>
<td>B = B + 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Suppose our intention (purpose) is to get A as 3 and B as 6 after the execution of both the processes. The interleaving of these instruction should be done in order to avoid race condition.
- If the order of execution is like
  
  \[
  \begin{align*}
  A &= 1 \\
  B &= 2 \\
  A &= B + 1 \\
  B &= B + 1 \\
  B &= B * 2
  \end{align*}
  \]

  A will contain 3 and B will contain 6 as desired.
• Whereas if the order of execution is like:
  
  A = 1
  B = 2
  B = B * 2
  A = B + 1
  B = B + 1

• A will contain **5** and B will contain **5** which is not desired.

• The O/P of the interleaved execution depends on the particular order in which the access take place and this is called **race condition**.

• To solve this problem, shared variables **A** and **B** should not be updated simultaneously by process 1 and 2 only.

• Only one process should manipulate the shared variable.
What design and management issues are raised by the existence of concurrency? We can list following concerns

1. The OS must able to keep track of the various processes. This is done with the use of process control block.
2. The OS must allocate and deallocate various resources for each active process. At times, multiple processes want access to the same resource. These resources include process time, memory, files, I/O devices.
3. The OS must protect the data and physical resources of each process against unintended interference by other processes. This involves techniques that relate to memory, files, and I/O devices.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.
Critical Region

- To avoid race condition, in many situations involving shared memory, shared files, and shared everything else to find some way to prohibit more than one process from reading and writing the shared data at the same time.

- Here we need, **mutual exclusion**(will see later into next topic), that is, make sure that if process is using a shared variable or file, the other processes will be excluded from doing the same thing.

- A process is busy doing internal computation and other things that do not lead to race conditions.
• However, sometimes a process has to access shared memory of files, or do other critical things that can lead to races.
• The part of program where the shared memory is accessed is called the **critical region** or **critical section**.
Conditions required to avoid race condition

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.
Requirement for Mutual Exclusion

- Mutual exclusion should meet the following criteria.

1. Mutual exclusion must be enforced only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.

2. A process that halts in its noncritical section must do so without interfering with other processes.

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or starvation.

4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

5. No assumption are made about relative process speeds or number or processors.

6. A process remains inside its critical section for a finite time only.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

5. No assumption are made about relative process speeds or number or processors.

6. A process remains inside its critical section for a finite time only.
Mutual exclusion with busy waiting

- We will see examine various proposals for achieving mutual exclusion. So that while one process is busy updating shared memory in its critical region. No other process will enter its critical region and cause trouble.

1. Disabling interrupts
2. Lock Variables
3. Strict Alternation
4. Peterson’s algorithm
Disabling interrupts

- On a single processor system, the simple solution is that each process disable all interrupts after enter into critical region and re-enable after leaving critical region.

- With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process, with interrupts turned off the CPU will not be switched to another process.

- **Interrupts**: - a suspension of a process, such as the execution of a computer program, caused by an event external to that process and performed in such a way that the process can be resumed.
• Once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will interfere.

• If the system is multiprocessor, disabling interrupts affects only the CPU that executed the disable instruction. The other one will continue running and can access the shared memory.
• It is frequently continent for the kernel itself to disable interrupts for a few instruction while it is updating variables.
• Disabling interrupts is often useful technique within the operating system but is not appropriate as a general mutual exclusion mechanism for user processes.
• To achieve mutual exclusion by disabling interrupts (within kernel) is becoming less every day due to increasing number of multi core chips in PCs.
• In a multi core disabling the interrupts of one CPU does not prevent other CPUs from interfering with operation the first CPU is performing.
Loc k Variable

- Consider having a single, shared (lock) variable, initially 0.
- When a process wants to enter in critical region, it checks first lock.
- If the lock is 0, the process sets it to 1 and enters the critical region.
- If the lock is already 1, the process has to wait until it becomes 0.
- A 0 means that no process is in its critical region, and 1 means that some process in critical region.
• This idea has a problems, suppose that one process read the lock and sees that 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 critical region at the same time.
Strict Alternation

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

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

(a) Process 0.  (b) Process 1.
• Integer value turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

• Initially, process 0 inspects turn, find it to be 0 and enter into critical region.

• Process 1 also finds the turn variable and it to be 0 so it has to wait when turn becomes 0.

• Continuously testing a variable until some value appears in called busy waiting.
Mutual Exclusion with Busy Waiting:
(3) Peterson’s Solution

• The previous solution solves the problem of one process blocking another process while its outside its critical section (not a good mutual exclusion)

• Peterson’s Solution is a neat solution with busy waiting, that defines the procedures for entering and leaving the critical region.
Mutual Exclusion with Busy Waiting (2)

#define FALSE 0
#define TRUE 1
#define N 2     /* number of processes */

int turn;  /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;  /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;  /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE;  /* indicate departure from critical region */
}
• Before entering its critical region, each process calls `enter_region` with its own process number, 0 or 1, as parameter.

• This will cause to wait, if need, until it is safe to enter.

• After it has finished with the shared variables, the process calls `leave_region` to indicate that it is done and to allow the other process to enter, it is required.
• Initially neither process is in its critical region. Now process 0 calls \textit{enter\_region}.

• It showing that its interested by setting is array element and sets \textit{turn} 0.

• Since process 1 is not interested, \textit{enter\_region} returns immediately.

• If process 1 now makes a call to \textit{enter\_region}, it will hang there until \texttt{interested[0]} goes to \texttt{FALSE}, an event that only happens process 0 calls \textit{leave\_region} to exit the critical region.
Now consider the case that both processes call \textit{enter\_region} almost simultaneously.

Both will store their process number in \textit{turn}. Whichever store is done last is the one that counts; the first one is overwritten and lost.

Suppose that process 1 stores last, so \textit{turn} 1.

When both processes come to while statement, process 0 executes it zero times and enter its critical region.

Process 1 loops and does not enter its critical region until process 0 exists its critical region.
Semaphore

- The fundamental principle is this: two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at specified place until it has received a specific signal.
- Any complex coordination requirements can be satisfied by the appropriate structure of signals. For signaling, special variable called semaphores are used.
- To transmit a signal via semaphore $s$, a process execute the primitive $\text{semSignal}(s)$.
- To receive a signal via semaphore $s$, a process executes the primitive $\text{semWait}(s)$.
- If the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.
• To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

1. A semaphore may be initialized to a nonnegative integer value.

2. The \texttt{semWait} operation decrements the semaphore value. If the becomes negative, then the process executing the \texttt{semWait} is blocked. Otherwise, the process continues execution.

3. The \texttt{semSignal} operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a \texttt{semWait} operation, if any, is unblocked.
• **Pseudo-code for wait**

```plaintext
wait (s)
{
    while ( s <= 0)
        s = s - 1
}
```

• **Pseudo-code for signal:**

```plaintext
signal (s)
{
    s = s + 1
}
```
• To begin, the semaphore has a zero or positive value. If the value is positive, that value equals the number of processes that can issue a wait and immediately continue to execute.

• If the value is zero, either by initialization or because a number of processes equal to initial semaphore value have issued a wait, the next process to issue a wait is blocked, and the semaphore value goes negative.

• Each subsequent wait drives the semaphore value further into minus territory.

• The negative value equals the number of processes waiting to be unblocked.

• Each signal unblocks one of the waiting processes when the semaphore values is negative.
Binary Semaphore

- Binary semaphore is a semaphore with an integer value that can range only between 0 and 1.
- In principle, it should be easier to implement the binary semaphore.
- In this, queue is used to hold processes waiting on the semaphore.
- The process that has been blocked the longest is released from the queue.
Properties

1. A binary semaphore may be initialized to 0 or 1.
2. The \texttt{semWaitB} operation checks the semaphore value. If the value is zero then the process executing the \texttt{semWaitB} blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The \texttt{semSignalB} operation check to see if any processes are blocked on this semaphore. If so, then a process blocked by a \texttt{semWaitB} operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.
Problem

• Semaphore offer a good solution to resolve the concurrency issues when multiple processes want to access the same resource.

• However, the problem with semaphores is the actual implementation of semaphores requires programming at the level of system calls.

• That is, the application developer needs to explicitly invoke semaphores-related system calls to ensure concurrency.

• This can not be tedious, but can actually lead to an erroneous code.
Mutex

- Similar to binary semaphore. A key difference between the two is that the process that locks the mutex (set the value to zero) must be the one to unlock it (set the value to 1).
- In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.
Monitor

• A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type.

• The monitor’s variable may only be access via its access procedure and only one process may be actively accessing the monitor at any one time.

• The access procedures are critical section. A monitor have a queue of processes that are waiting to access it.
A monitor is a group or collection of data items, a set of procedures to operate on the data items and a set of special variable known as condition variable.

The condition variables are similar to semaphores. The only operations possible on a condition variable are signal and wait just like a semaphore.

An important difference between a condition variable and a semaphore is the way in which the signal operation is implemented.
• In case of a semaphore, the signal operation always increments the value the semaphore. Essentially, it changes the state of the semaphore even if there are no processes blocked on the semaphore.

• However, in case of condition variable, if there are no processes blocked on the condition variable, then the signal operation does not do anything.
difference between condition variables and semaphores

- Semaphores are sticky: they have memory, `semSignal()` will increment the semaphore, even if no one has called `semWait()`.

- Condition variables are not: if no one is waiting for a `signal()`, this `signal()` is not saved.
Advantages of semaphores

- Processes do not busy wait while waiting for resources. While waiting, they are in a “suspended” state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- It can be used to implement other synchronization tools
  Monitors, protected data type, bounded buffers, mailbox etc
Disadvantages of semaphores

- can only be invoked by processes--not interrupt service routines because interrupt routines cannot block
- Loss of mutual exclusion
Disadvantages of Monitor

- Major weakness of monitor is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.
- Monitor can not easily be added if they are not natively supported by the language.
CPU Scheduling
Basic

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

• In a uniprocessor system, only one process may run at a time, any other process must wait until the CPU is free and can be rescheduled.

• Scheduling is a fundamental operating system function. All computer resources are scheduled before use.
CPU-I/O BURST CYCLE

• The success of CPU scheduling depends on the following observed property of process

• Process execution consists of a cycle of CPU execution and I/O wait.

• Process alternate between two states. Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on.

• The last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.
CPU Scheduler

- What is short term scheduler?
- What is long term scheduler?
- What is medium term scheduler?
• The **Short-term schedulers** must select a new process for the CPU frequently.

• **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue. Long term scheduler determines which program are admitted to the system for processing. Job scheduler selects process from the queue and loads them into memory for execution.

• The **medium-term scheduler**, removes processes from memory, and thus reduce the degree of multiprogramming.
CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by short term scheduler (CPU scheduler).
- The scheduler selects from among the processes in memory that ready to execute, and allocates the CPU one of them.
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

- Scheduling under 1 and 4 is nonpreemptive(?)
- All other scheduling is preemptive(?)

51
• **Nonpreemptive**: 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.

• **Preemptive Scheduling** : - An interrupt causes currently running process to give up the CPU and be replaced by another process.
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- **Dispatch latency** — time it takes for the dispatcher to stop one process and start another running.

- The dispatcher should be as fast as possible, given that it is invoked during every process switch.
Scheduling Criteria

- **CPU utilization**: keep the CPU as busy as possible, CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent to 90 percent.

- **Throughput**: # of processes that complete their execution per time unit
  
  one measure of work is the number of processes completed per time unit, called **throughput**.

  For long processes, this rate may be 1 process per hour; for short transaction, throughput might be 10 processes per second.
- **Turnaround Time**: amount of time to execute a particular process.
- 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**.
- Turnaround time is the sum of periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
• **Waiting time**: amount of time a process has been waiting in the ready queue.

• 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.

• Waiting time is the sum of the periods spent waiting in the ready queue.
• **Response time**: amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

• 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 result are being output to the user.

• Another measure is the time from the submission of a request until the first response is produced.

• This measure, called **response time** (refer 1st sentence).
First-Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first. 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.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: $P_1, P_2, P_3$

  The Gantt Chart for the schedule is:

```
+-------+-------+-------+
| P_1   | P_2   | P_3   |
| 0     | 24    | 27    |
| 24    | 27    | 30    |
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the processes arrive in the order $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
### Another Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Grantt Chart
- Waiting Time
- Average Waiting Time
- Turnaround Time
- Average turnaround Time
- **Grantt Chart**

    | P1 | P2 | P3 | P4 |
    |----|----|----|----|
    | 0  | 5  | 12 | 15 | 19 |

- **Waiting Time**

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>15</td>
</tr>
</tbody>
</table>
• **Average waiting time:**

\[
\text{Average waiting time} = \text{waiting times of all processes} \\
\frac{0+5+12+15}{4} = 8 \text{ milliseconds}
\]
• Turnaround time:

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround time (Burst time + waiting time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>5+0=5</td>
</tr>
<tr>
<td>P2</td>
<td>7+5=12</td>
</tr>
<tr>
<td>P3</td>
<td>3+12=15</td>
</tr>
<tr>
<td>P4</td>
<td>4+15=19</td>
</tr>
</tbody>
</table>

• Average turnaround time:
  \[(5+12+15+19)/4 = 12.75\]
Shortest-Job-First (SJF) Scheduling

- This algorithm associates with each process the length of the latter’s next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If two processes have the same length next CPU burst, FCFS scheduling is used to break tie.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- 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.
## Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
</tr>
</tbody>
</table>

- Grantt Chart
  - Waiting time
  - Average Waiting time
  - Turnaround time
• Grant chart:

<table>
<thead>
<tr>
<th></th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

• Waiting Time:

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>16</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
</tr>
</tbody>
</table>

• Average waiting time

\[
\text{Average waiting time} = \frac{3 + 16 + 9 + 0}{4} = \frac{28}{4} = 7 \text{ milliseconds}
\]
**Turnaround time:**

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3 + 6 = 9</td>
</tr>
<tr>
<td>P2</td>
<td>16 + 8 = 24</td>
</tr>
<tr>
<td>P3</td>
<td>7 + 9 = 16</td>
</tr>
<tr>
<td>P4</td>
<td>3 + 0 = 3</td>
</tr>
</tbody>
</table>

**Average turn around time = 9 + 24 + 16 + 3 / 4 = 13 milliseconds**
Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).

**Preemptive**
- When a process arrives at the ready queue, its priority compared with the priority of the currently running process.
- A preemptive 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.

**Nonpreemptive**
- A nonpreemptive priority scheduling algorithm will supply put the new process at the head of the ready queue.
• A major problem with priority-scheduling algorithms is indefinite blocking.
• A process that is ready to run but lacking the CPU can be considered blocked—waiting for the CPU.
• A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU.
• In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
• Either process will be run, or the computer system will crash and lose all unfinished low-priority processes.
• 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 decrement the priority of a waiting process by 1 every 15 minutes.

• Eventually, even a process with an initial priority of 172 would have the highest priority in the system and would be executed.
### Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Gantt Chart

<table>
<thead>
<tr>
<th></th>
<th>P2</th>
<th>P5</th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>6</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waiting Time

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>16</td>
</tr>
<tr>
<td>P4</td>
<td>18</td>
</tr>
<tr>
<td>P5</td>
<td>1</td>
</tr>
</tbody>
</table>

Avg. waiting time = 8.2 milliseconds
**Turnaround Time**

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$10 + 6 = 16$</td>
</tr>
<tr>
<td>P2</td>
<td>$1 + 0 = 1$</td>
</tr>
<tr>
<td>P3</td>
<td>$2 + 16 = 18$</td>
</tr>
<tr>
<td>P4</td>
<td>$1 + 18 = 19$</td>
</tr>
<tr>
<td>P5</td>
<td>$5 + 1 = 6$</td>
</tr>
</tbody>
</table>

**Avg. turnaround time = 12 milliseconds**
Round Robin Scheduling

- It is specially designed for time-sharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A smaller unit of time, called a time quantum, is defined.
- A time quantum is generally from 10 to 100 milliseconds.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
• Same work as FIFO.
• 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 operating system.
• 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.
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.

If a process CPU burst exceeds 1 time quantum, the process is preempted and is put back in ready queue.

The performance of RR algorithm depends heavily on the size of time quantum.

At one extreme, if the time quantum is very large, to RR policy is the same as FCFS policy.

If the time quantum is very small, the RR approach is called Process sharing.
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Time quantum = 4 milliseconds
Gantt Chart, Waiting time, Average waiting time, turnaround time & average turnaround time
Waiting time:

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$0 + (10 - 4) = 6$</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
</tr>
</tbody>
</table>

Average Waiting time

$17 / 3 = 5.66$ milliseconds
Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>2</td>
</tr>
</tbody>
</table>

- Time quantum = 2 milliseconds
- Calculate
  - Gantt chart
  - Waiting time, average waiting time,
  - Turnaround time, average turnaround time
### Waiting time

<table>
<thead>
<tr>
<th>Process</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$0 + (8 - 2) = 6$</td>
</tr>
<tr>
<td>P2</td>
<td>$2 + (9 - 4) + (13 - 11) = 9$</td>
</tr>
<tr>
<td>P3</td>
<td>$4 + (11 - 6) = 9$</td>
</tr>
<tr>
<td>P4</td>
<td>$6$</td>
</tr>
</tbody>
</table>

### Average waiting time = 7.5 milliseconds
- **Turnaround time**

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$3 + 6 = 9$</td>
</tr>
<tr>
<td>P2</td>
<td>$6 + 9 = 15$</td>
</tr>
<tr>
<td>P3</td>
<td>$4 + 9 = 13$</td>
</tr>
<tr>
<td>P4</td>
<td>$2 + 6 = 8$</td>
</tr>
</tbody>
</table>

- Average turnaround time $= \frac{45}{4}$
  $= 11.25$ milliseconds