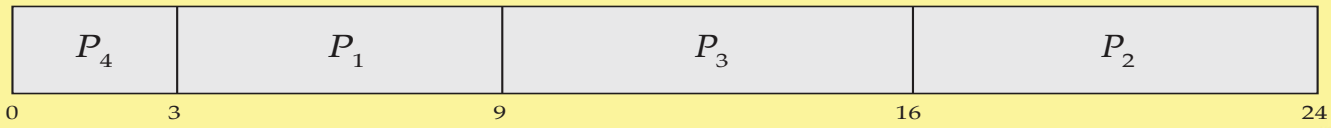
Example of SJF

- Consider the following four processes and their burst time

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate (predict) the length – in most cases should be similar to the previous CPU burst
  - Pick the process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \le \alpha \le 1$
  4. Define : $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n$.
- Commonly, α set to ½

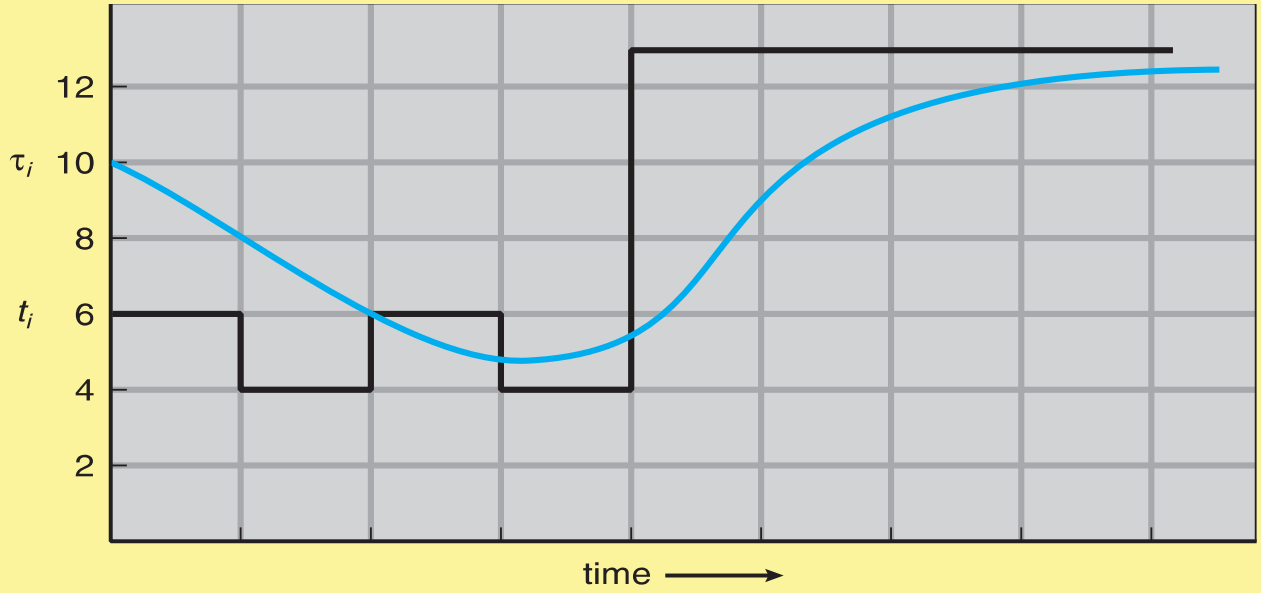

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst



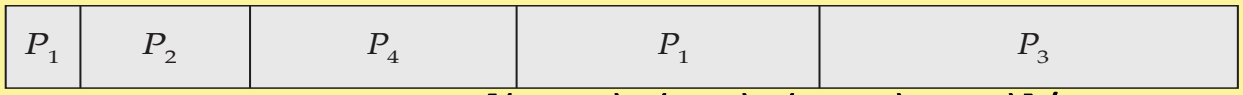
CPU burst (t_i)		6	4	6	4	13	13	13 ...
"guess" (τ_i)	10	8	6	6	5	9	11	12 ...

Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Example illustrating the concepts of varying arrival times and preemption.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4
  = 26/4 = 6.5 msec

Round Robin (RR)

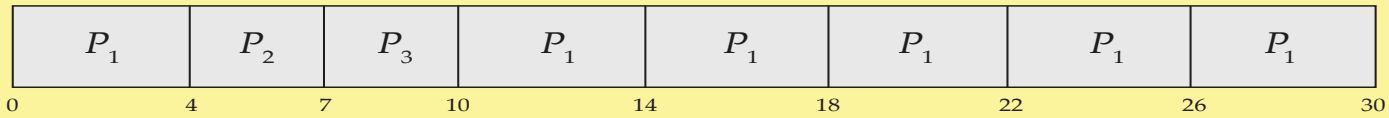
- Each process gets a small unit of CPU time (**time quantum** q). After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are N processes in the ready queue and the time quantum is q, then each process gets $1/N$ of the CPU time in chunks of at most q time units at once. No process waits more than $(N-1)*q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high


Example of RR with Time Quantum = 4

- Consider the following three processes and their burst time

Process	Burst Time
P_1	24
P_2	3
P_3	3

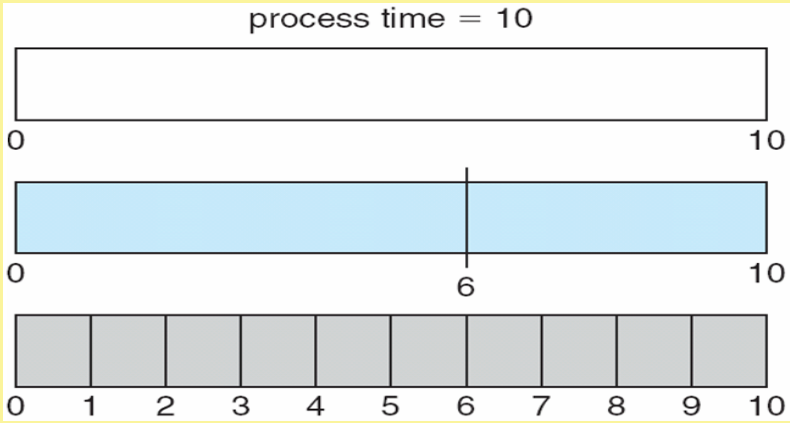
- The Gantt chart is:



- The average waiting time under the RR policy is often longer
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q is usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time

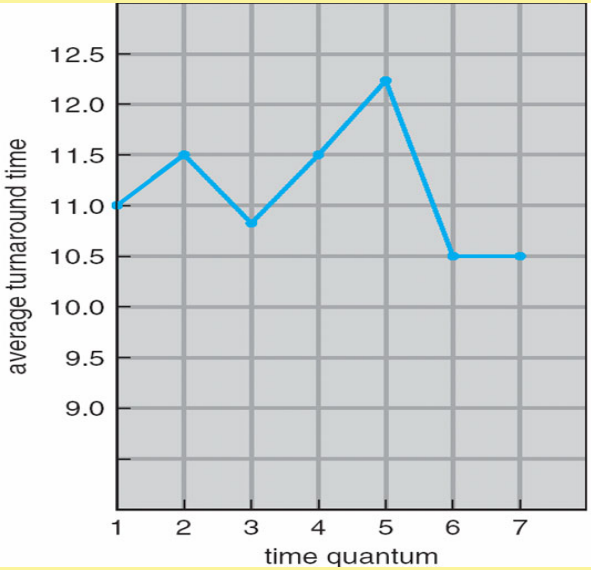
- The performance of the RR algorithm depends on the size of the time quantum. If the time quantum is extremely small (say, 1 millisecond), RR can result in a large number of context switches.



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies with the Time Quantum

- The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

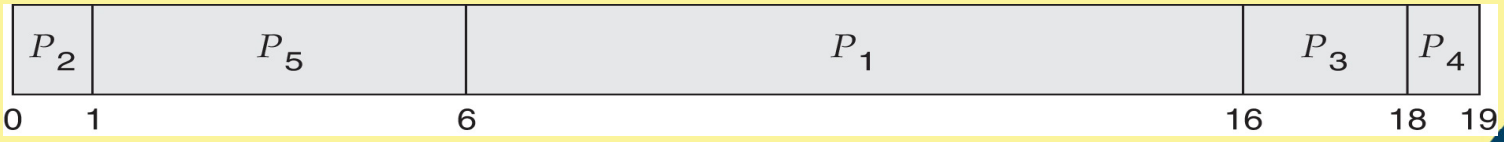
Priority Scheduling

- 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
  - Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process


Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



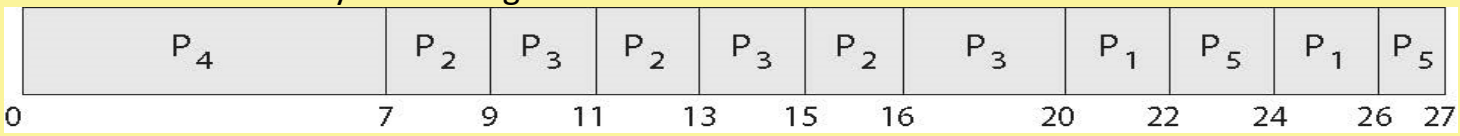
- Average waiting time = 8.2 msec

Combining Priority Scheduling and RR

- System executes the highest priority process; processes with the same priority will be run using round-robin.
- Consider the following five processes and their burst time

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Priority scheduling Gantt Chart

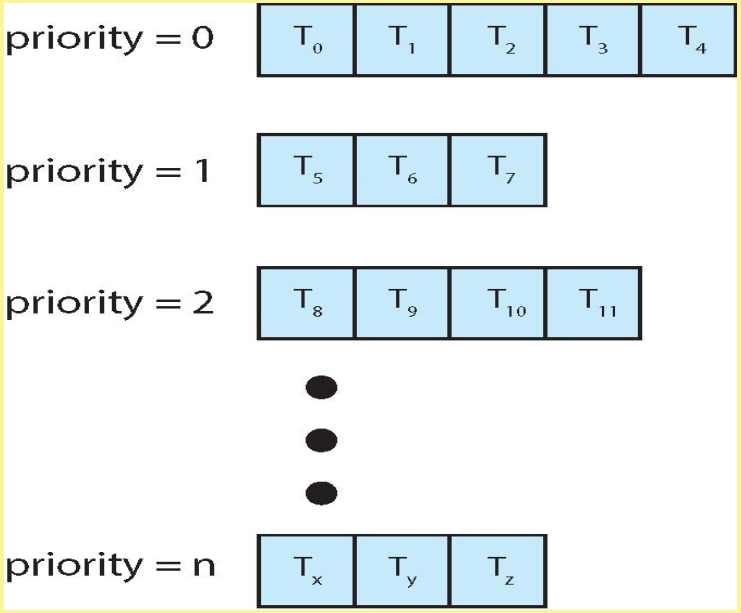


- Average waiting time = 8.2 msec

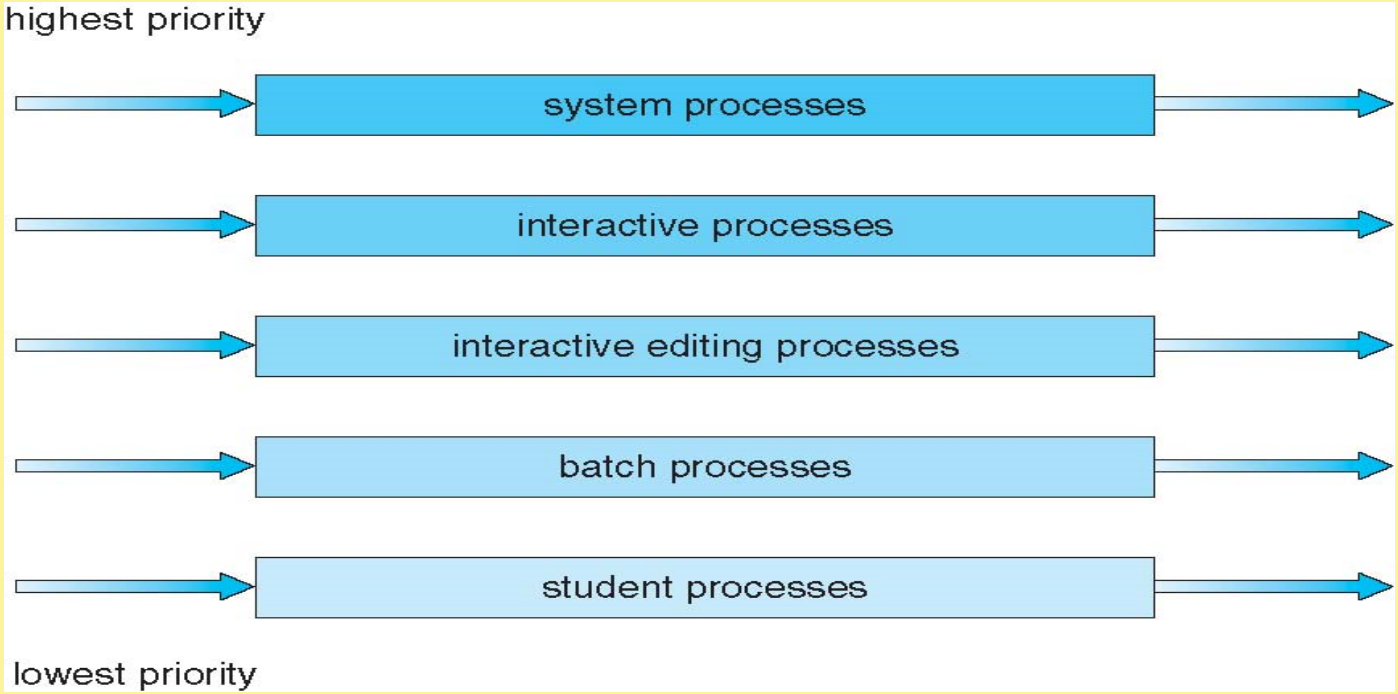
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS


Separate Queue For Each Priority



Multilevel Queue Scheduling

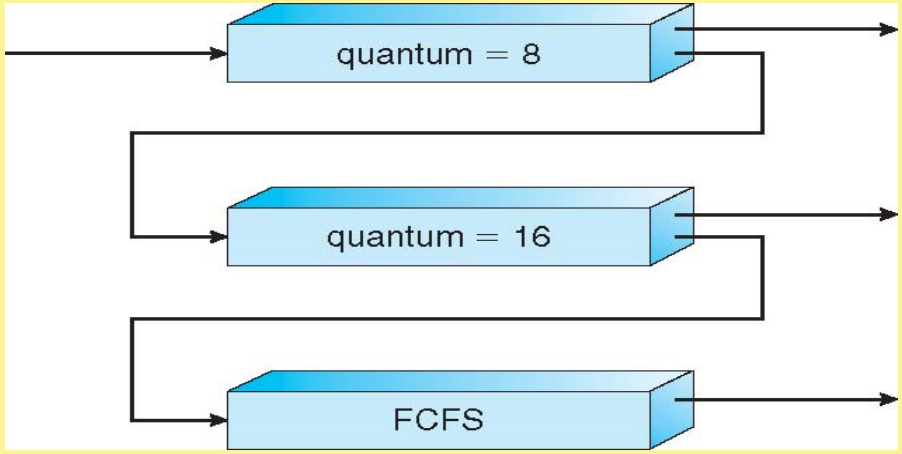


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service


Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Conclusion

Conclusion:

- CPU scheduling selects a waiting process from the ready queue
- FCFS is the simplest policy
- SJF is provably optimal, difficult to implement
- RR is appropriate for interactive systems
- Multilevel queue allows different queues for different classes of problems


NPTEL ONLINE CERTIFICATION COURSES

Thank you

Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

Process Synchronization

Concepts Covered:

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems


Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The synchronization mechanism is usually provided by both hardware and the operating system
- Illustration of the problem – The producer-Consumer problem introduced earlier
- Basic assumption – load and store instructions are atomic


Producer-Consumer Problem

- The Solution we presented in chapter 3 is correct, but can only use BUFFER_SIZE - 1 elements.
- The methodology used is to allow only a single process to increment/decrement a particular shared variable.
- There is a solution that fills ***all*** the buffers, using the same methodology:
  - The producer process increments the value on the variable "in" (but not "out") and the consumer process increments the value on the variable "'out" (but not "in")
  - The solution is more complex. Try and see if you can come up with the algorithm.


Producer-Consumer Problem (Cont.)

- Suppose that we wanted to provide a solution that fills ***all*** the buffers where we allow the producer and consumer processes to increment and decrement the same variable.
- We can do so by adding another integer variable -- **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0.
- The variable **counter** It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
- Code is shown in next two slides


Producer Process

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter = counter +1;
}
```


Consumer Process

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
       counter = counter - 1;

        /* consume the item in next consumed */

}
```


Race Condition

- **`counter = counter + 1`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter = counter -1`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}


Race Condition (Cont.)

- How do we solve the race condition?
- We need to make sure that:
  - The execution of

    `counter = counter + 1`

    is done as an "atomic" action. That is, while it is being executed, no other instruction can be executed concurrently.
    - actually no other instruction can access **`counter`**
  - Similarly for

    `counter = counter - 1`
- The ability to execute an instruction, or a number of instructions, atomically is crucial for being able to solve many of the synchronization problems.


Critical Section Problem

- Consider system of ***n*** processes $\{P_0, P_1, \ldots P_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section** code; it than executes in the critical section; once it finishes executing in the critical section it enters the **exit section** code. The process then enters the **remainder section** code.


General structure of Process Entering the Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Hardware Solution

- Entry section – first action is to “disable interrupts”.
- Exit section – last action is to “enable interrupts”.
- Must be done by the OS. Why?
- Implementation issues:
  - Uniprocessor systems
    - Currently running code would execute without preemption
  - Multiprocessor systems.
    - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Is this an acceptable solution?
  - This is impractical if the critical section code is taking a long time to execute.


Software Solution for Process P_i

- Keep a variable "turn" to indicate which process is next

```
do {
    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

- Algorithm is correct. Only one process at a time in the critical section. But:
- Results in "busy waiting".
- What if turn = j; P_i wants to enter the critical section and P_j does not want to enter the critical section?

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes