Operating Systems

Lecture 3.4

Process Scheduling Algorithms - I

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Overview of Process Scheduling
- Process State Models
- Short and Medium Term Scheduler
- Process Scheduler & Dispatcher
- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Exponential Averaging
- Priority Scheduling
- Round Robin Scheduling (RR)

Overview of Process Scheduling

Classification of Processes

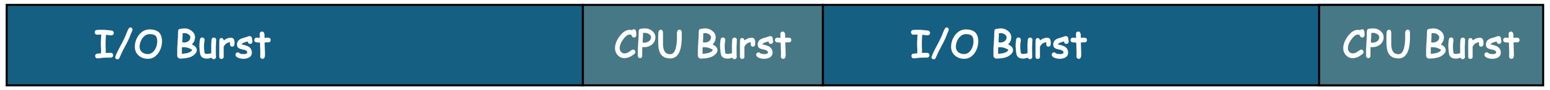
When speaking about process scheduling, processes are traditionally classified into three different classes:

- **Interactive Processes:** These interact constantly with their users. When input is received, the average delay must fall between `50-150 ms`, otherwise the user will find the system to be unresponsive. Typical interactive programs are command shells, text editors and graphical applications
- **Batch Processes:** These do not need user interaction and often execute in the background and are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines and scientific computations
- **Real-time Processes:** These processes should have a short guaranteed response time with a minimum variance. Typical real-time programs are multimedia applications, robot controllers, and programs that collect data from physical sensors

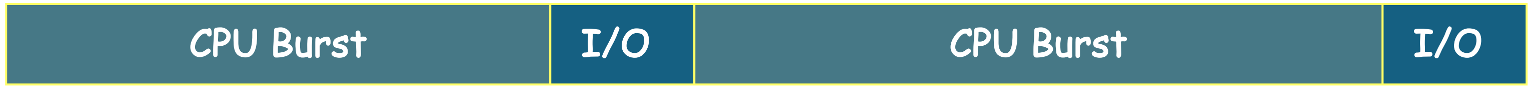
CPU Bound & I/O Bound Processes

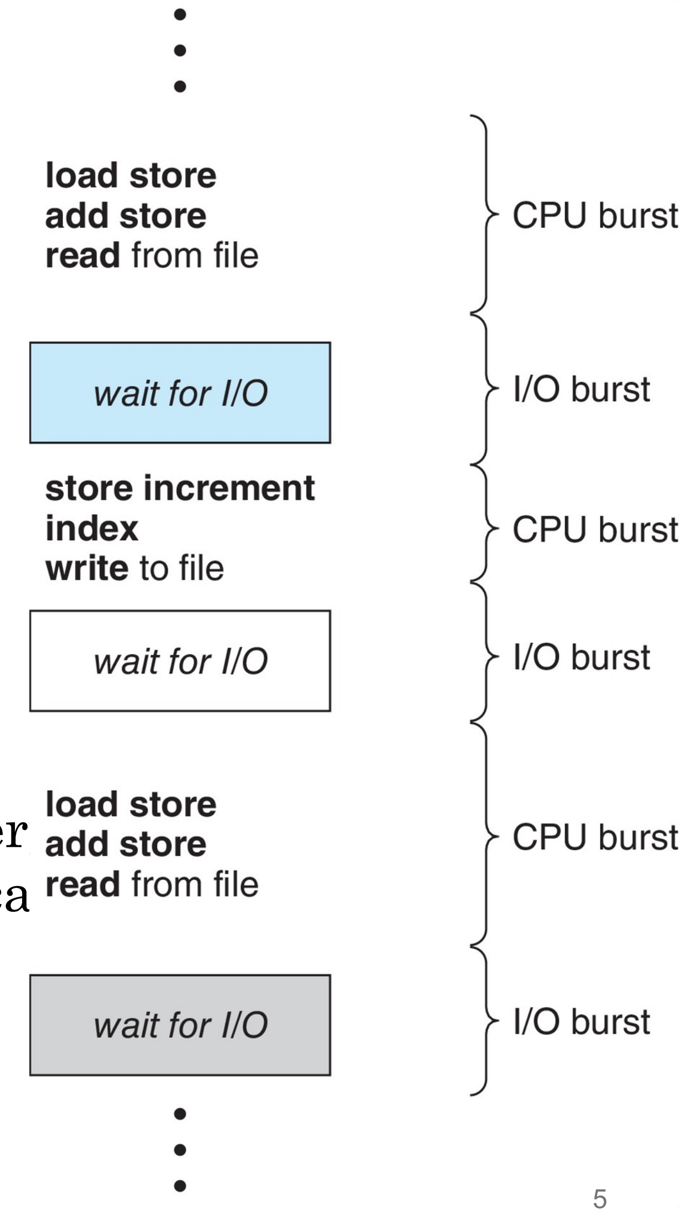
- Process execution consists of a cycle of CPU execution and I/O wait
- Processes move back & forth between these two states
- A process execution begins with a CPU burst, followed by an I/O burst and then another CPU burst and so on

- **I/O-bound process** spends more time doing I/O than computations; Many short CPU bursts
- Examples: Word processing, text editors. Billing system of Wapda which involves lot of printing



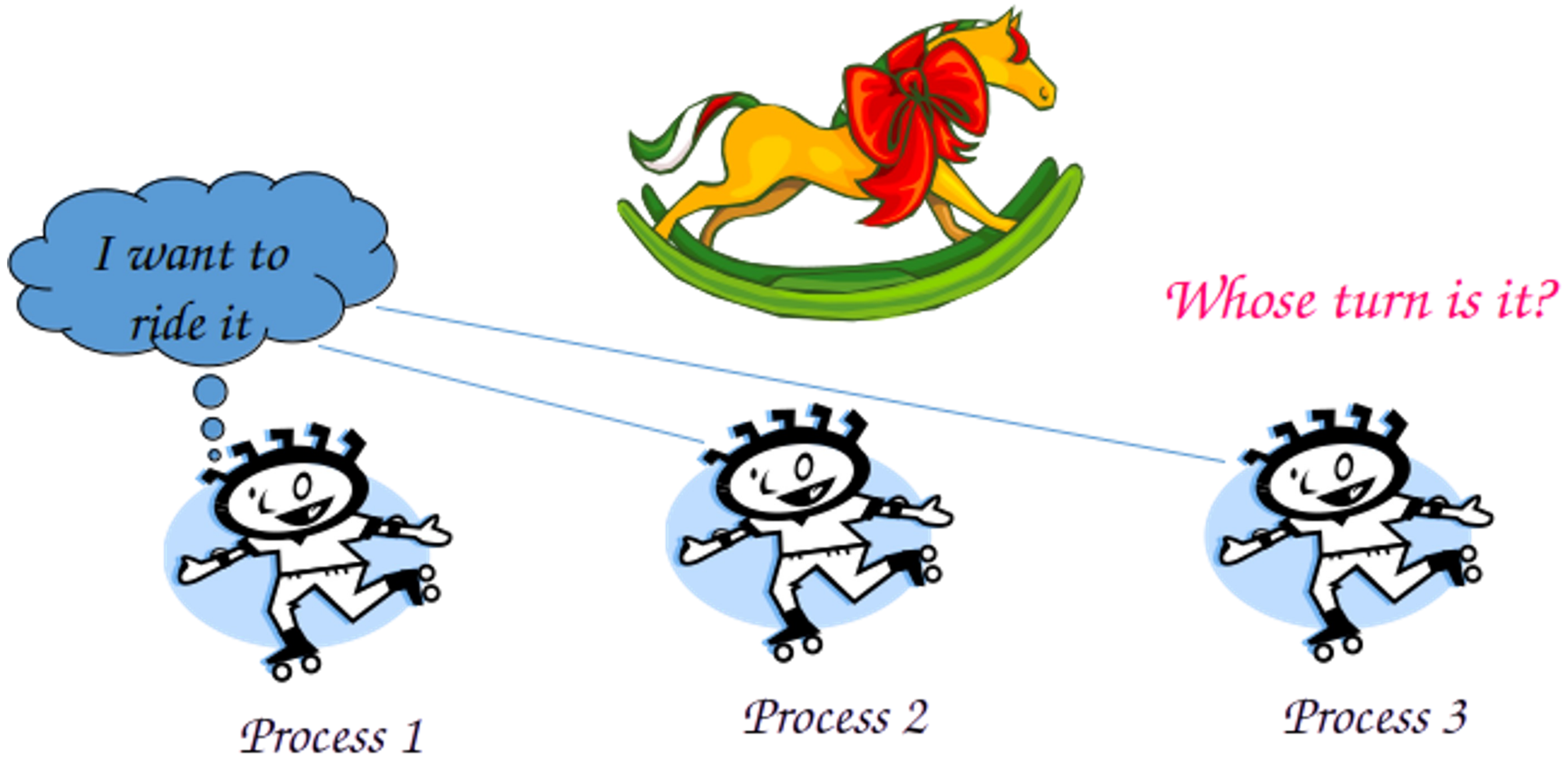
- **CPU-bound process** spends more time doing computations; Few ver
- Examples: Simulation of NW traffic involving lot of mathematica applications involving matrix multiplication, DSP applications





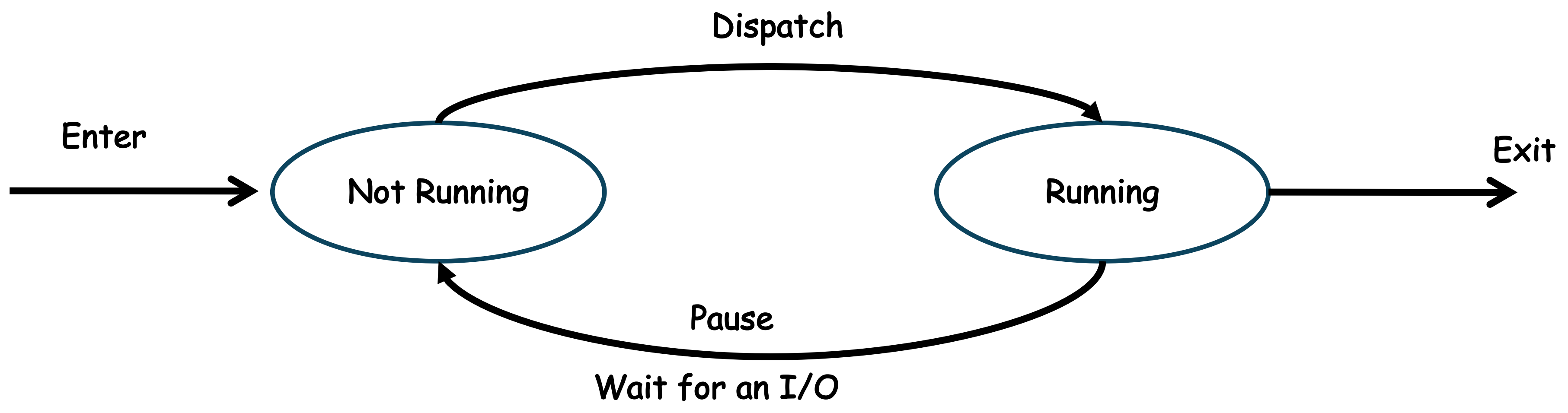
Scheduling

Deciding which process/thread should occupy a resource (CPU, disk, etc.)



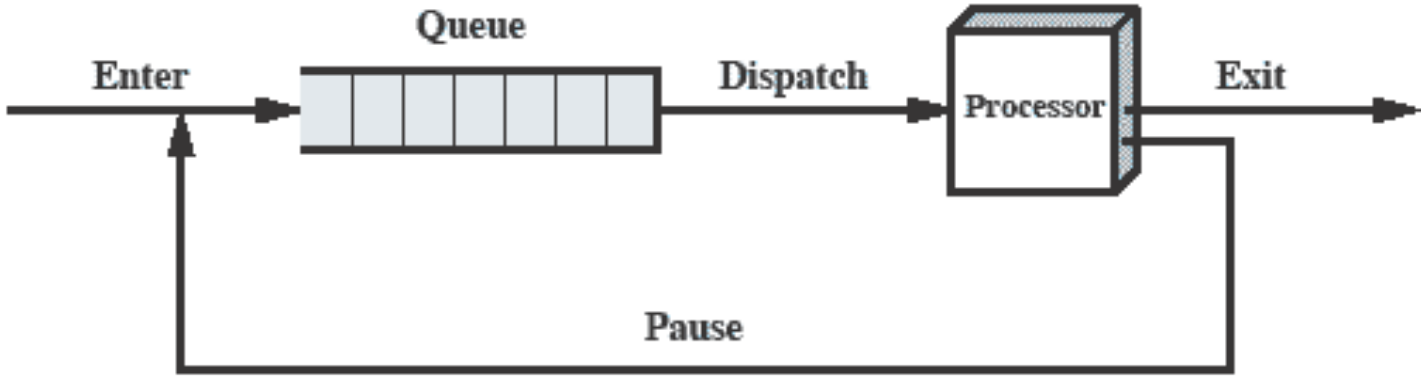
2 – State Process Model

- Broadly speaking, the life of a process consists of CPU bursts and I/O bursts. So simplest possible model can be constructed by observing that at any particular time, a process is either being executed by a processor or is not running or waiting for an I/O
- There may be a number of processes in the “not running” state but only one process will be in “running” state



Queuing Diagram (2 – State Process Model)

- **Queuing structure for a two state process model is:**



(b) Queuing diagram

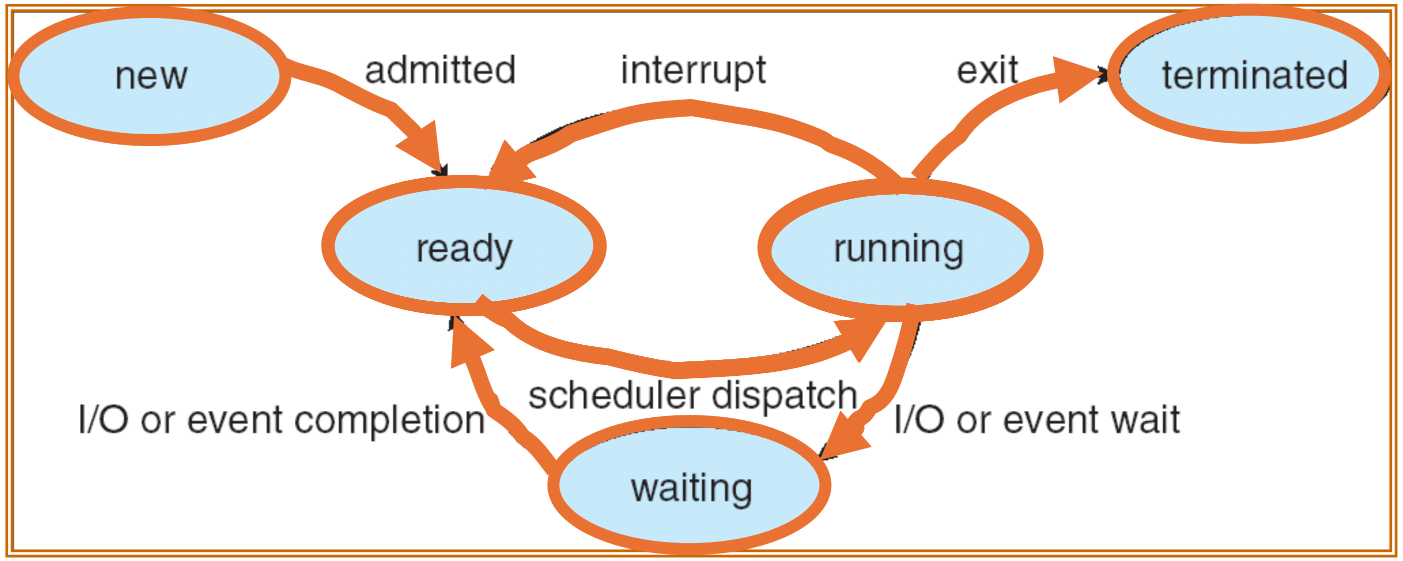
Limitations

- If all processes in the queue are always ready to execute only then the above queuing discipline will work
- But it may also be possible that some processes in the queue are ready to execute, while some are waiting for an I/O operation to complete
- So the scheduler/dispatcher has to scan the list of processes looking for the process that is not blocked and is there in the queue for the longest

5 – State Process Model

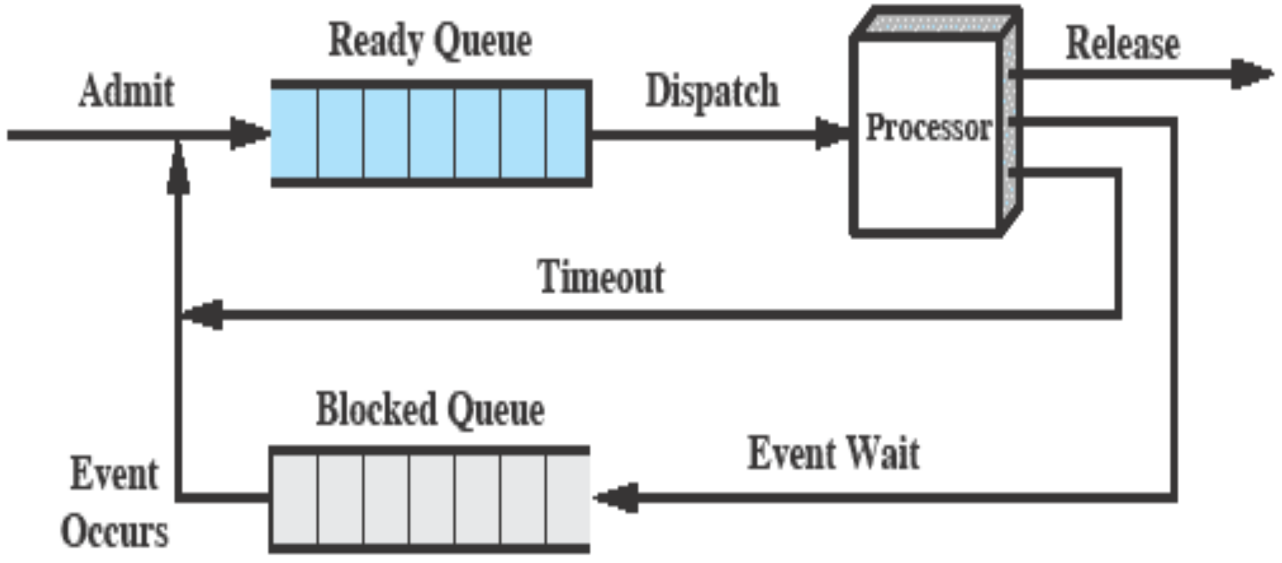
Broadly speaking the life of a process consist of CPU burst and I/O burst but in reality

- A process may be waiting for an event to occur; e.g. a process has created a child process and is waiting for it to return the result
- A process may be waiting for a resource which is not available at this time
- Process has gone to sleep for some time



- new: The process is being created (Disk to Memory)
- ready: The process is in main memory waiting to be assigned to a processor
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur (I/O completion or a signal)
- terminated: The process has finished execution

Queueing Diagram using 2 Queues



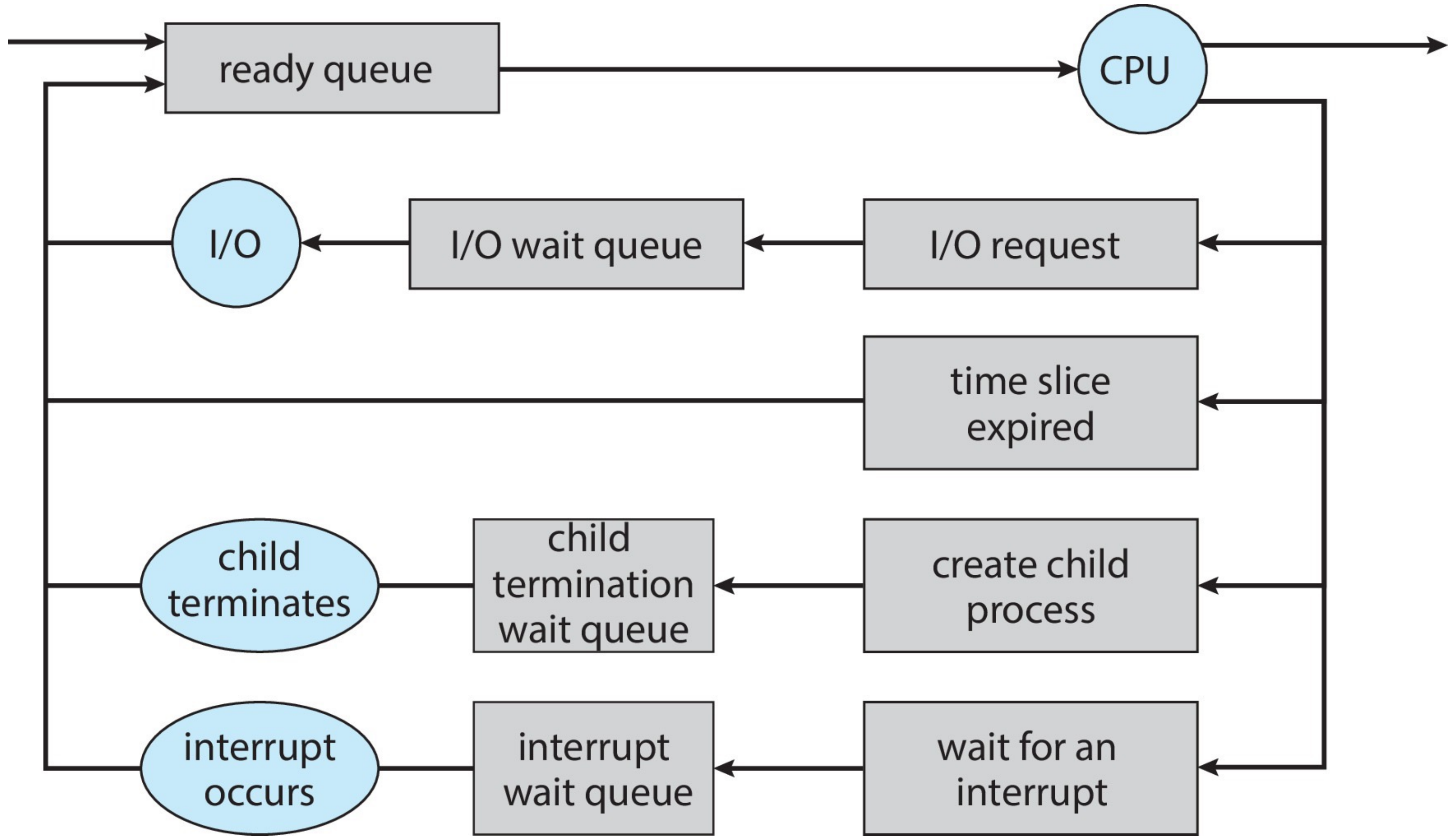
(a) Single blocked queue

Limitation:
When an event occurs, the OS must scan the entire blocked queue to find processes waiting for that event. With hundreds or thousands of processes, this becomes a major scheduling overhead.

Solution:
Instead of one large blocked queue, maintain separate waiting queues for each event or resource.

Queueing Diagram using Multiple Queues

A process migrates between various scheduling queues throughout its life cycle. OS must select processes from these queues in some predefined fashion. This selection is done by an appropriate scheduler



Scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment

Short Term / CPU / Process Scheduler

The OS code responsible for deciding which process runs next, when, and for how long is known as short-term scheduler

- Short-term scheduler is invoked very frequently (typically every few milliseconds), so it must be extremely fast and efficient.
- On a context switch, the scheduler:
  - Saves the state of the currently running process.
  - Selects the next process from the ready queue based on the scheduling policy.
  - Loads the saved state of the selected process and resumes execution.
- Invoked when following events occur
  - CPU slice of the current process finishes.
  - Current process blocks to wait for an event (e.g., I/O completion).
  - Clock interrupt occurs (timer-based scheduling).
  - I/O interrupt signals a waiting process can continue.
  - System call that may cause process blocking or preemption
  - Signal that changes a process's scheduling state.

Dispatcher

The OS code that **takes** the **CPU away** from the **current process** and hands it over to the **newly scheduled process** is known as the dispatcher

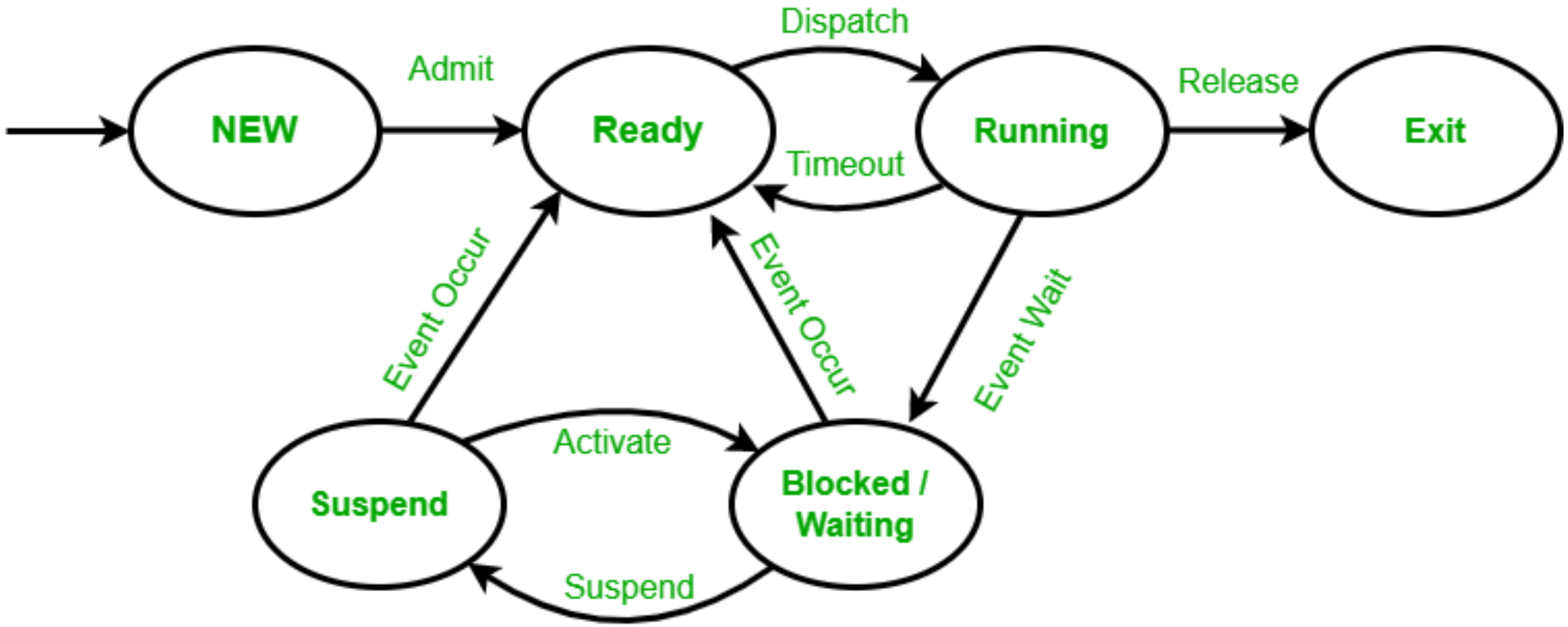
- The dispatcher is the OS component that transfers control of the CPU from the currently running process to the next process selected by the short-term scheduler.
- Works closely with the CPU scheduler to ensure seamless switching between processes.
- The primary role of dispatcher is to stop the execution of the current process, and load and start the execution of the new process chosen by the scheduler.
- Dispatcher performs following functions:
  - Context Switching: Save the state of the current process and load the state of the next process.
  - Mode Switching: Switch from kernel mode to user mode for process execution.
  - Program Restart: Jump to the correct instruction in the user program to resume execution.
- Dispatch Latency: It is the time taken by the dispatcher to stop one process and start another. Includes the overhead of context switch, mode switch, and program counter update.

Medium Term Scheduler

- The CPU is much faster than I/O, so it is common for all processes in memory to be waiting for I/O, leaving the CPU idle.
- Even with multiprogramming, CPU utilization can be low if no ready processes are in memory.
- Solution is Swapping, i.e., moving part or all of a process from main memory to disk and back.
- The medium term scheduler (swapper) temporarily removes (swaps out) a process from main memory to disk, freeing memory for other processes. A blocked process becomes suspended when swapped to disk. Later, the process can be swapped in to resume execution.
- The medium term scheduler is also known as swapper, as it selects an in-memory process and swaps it out to the disk temporarily. Blocked state becomes suspend state when swapped to disk.
- Swapping decision is based on several factors:
  - All processes in memory are blocked, and a new process needs to be brought in.
  - Arrival of a higher-priority process with insufficient memory available.
  - To improve the process mix (balance CPU-bound and I/O-bound processes).
  - To free memory when a process's memory requirement cannot be met with current allocation.

6 – State Process Model (one suspended state)

One suspended state allow processes, which are not actually running to be swapped from the Blocked/Waiting state

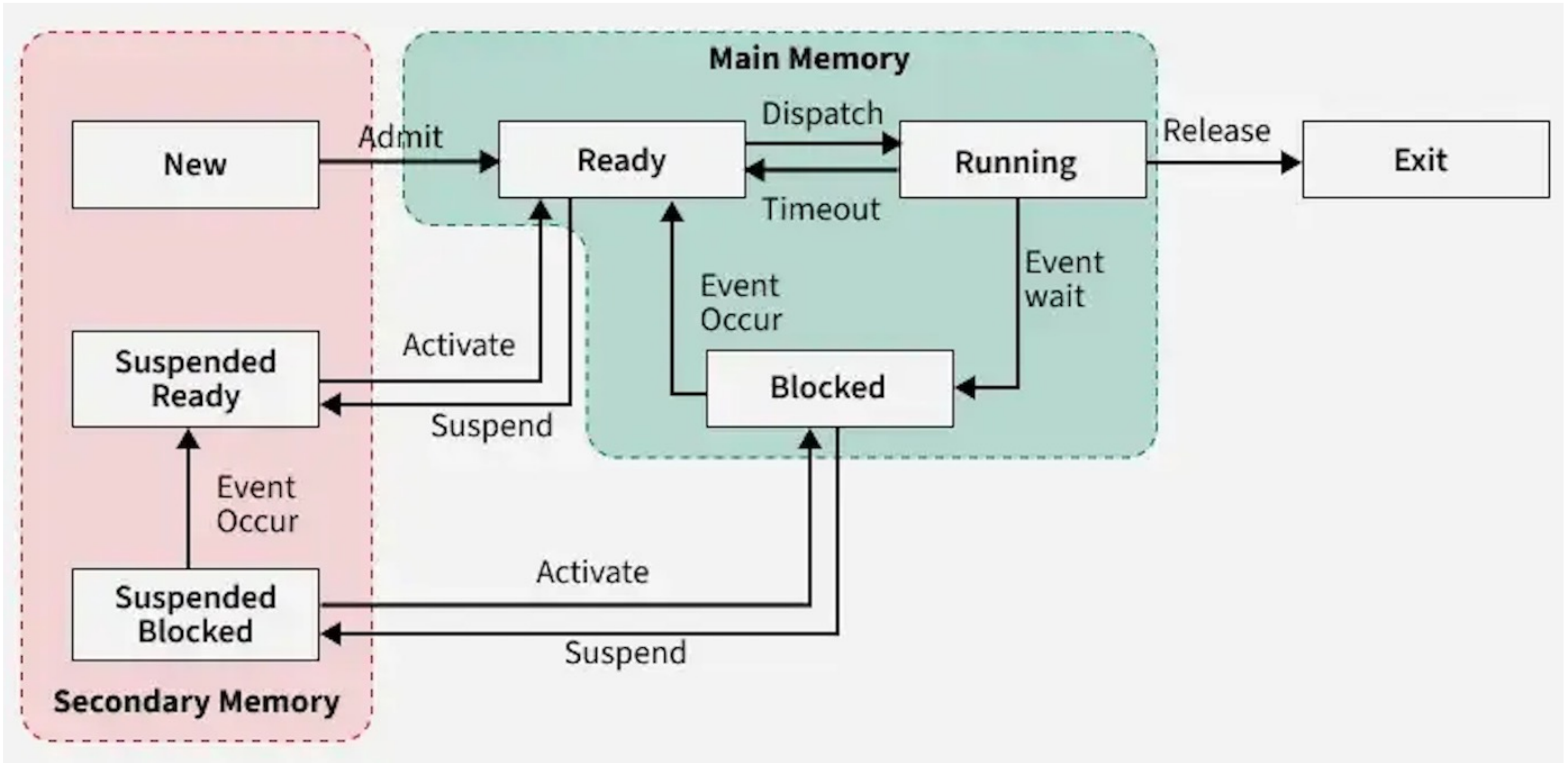


Limitation: This only allows processes which are blocked to be swapped out. What if there is no blocked process but we still need to free up memory?

Solution: Add another state Ready Suspended and swap out a process from the ready state

7 – State Process Model (two suspended states)

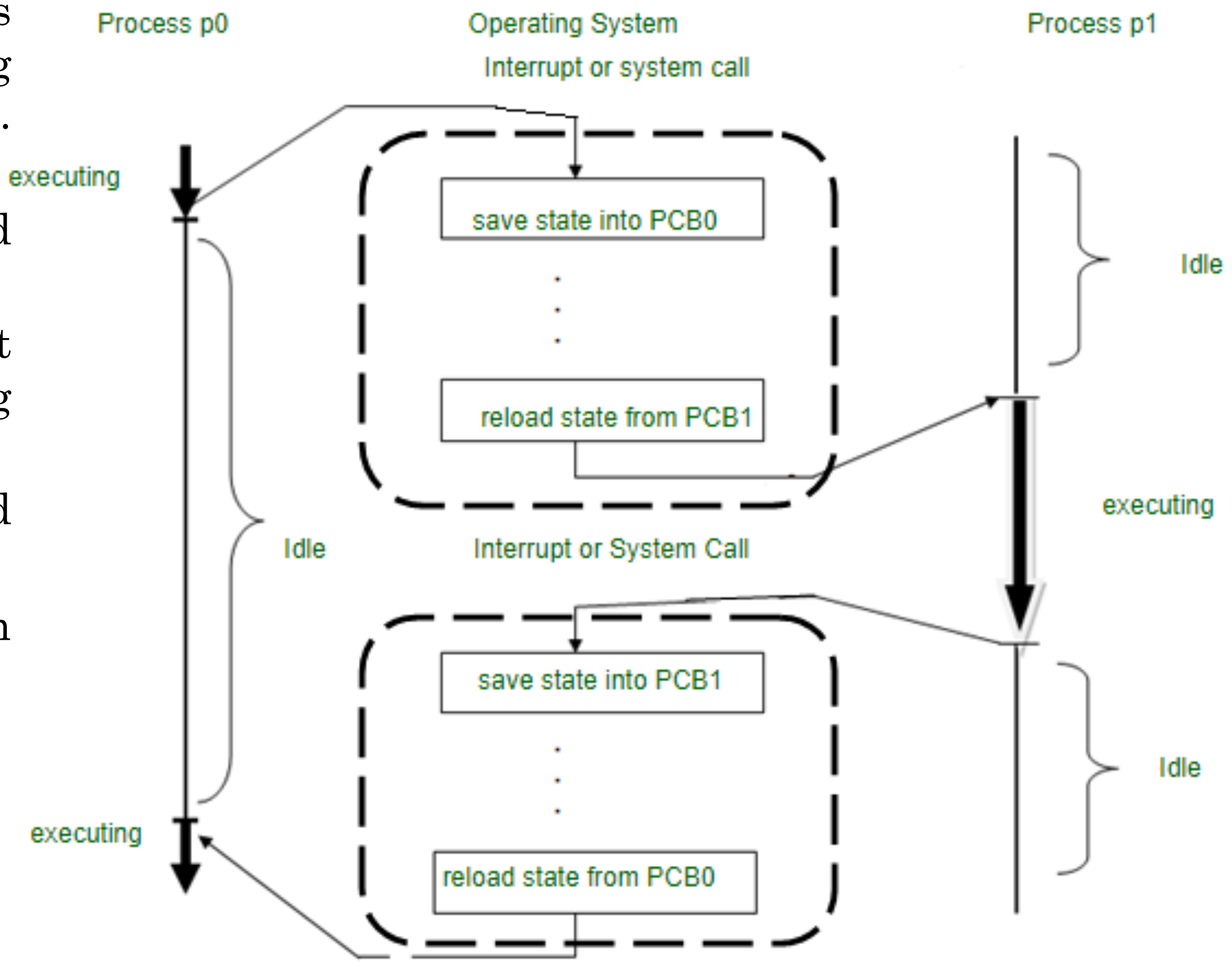
Two suspended states allow processes, which are not actually running to be swapped from the Ready as well as from the Blocked/Waiting states



Process / Context Switch

A context switch occurs when the CPU switches from one process to another due to multi-tasking requirement, an interrupt, or a user/kernel switch. Following steps occur in case of a process switch:

- The state of the current process (PCB) is saved for rescheduling.
- The OS aborts the execution of the current process and selects a process from the waiting list by tuning its PCB.
- Load the PCB of the selected process and continue its execution.
- Context-switch time is overhead; the system does no useful work while switching.



CPU Scheduler

Main Categories of Process Schedulers

Preemptive scheduling

- The scheduler can interrupt a running process and assign the CPU to another process.
- The duration a process runs before preemption is called its time slice or quantum.
- In many modern operating systems, the time slice is dynamically calculated based on process behavior and system policy.
- Improves responsiveness, particularly in interactive systems.

Non-preemptive Scheduling

- Once a process starts executing, it keeps the CPU until it voluntarily releases it by:
    - Terminating, or
    - Requesting I/O (blocking itself)
- The CPU is not taken away forcefully, rather the process runs until its CPU burst is complete.
- Simpler to implement but may lead to poor responsiveness for high-priority or interactive tasks.

Kernel Types (Based on preemptibility)

- A system executes either in *user mode* (running user-written code) or *kernel mode* (running kernel code). The **Kernel code runs in two main contexts:**
  - Process context: Code runs on behalf of a specific process (e.g., handling a system call). It can sleep or block.
  - Interrupt context: Code runs in response to hardware interrupts (interrupt service routines). It must not block and is typically short.

Kernel Types (based on pre-emptibility)

- Non-preemptive kernel: Kernel code cannot be preempted; once executing in kernel mode it runs until it blocks or returns to user mode.
- Reentrant (process-preemptible) kernel: Kernel allows preemption while running in process context but not during interrupt context.
- Preemptive (fully preemptible) kernel: Kernel allows preemption while running in both process context as well as in interrupt context.

Scheduling Objectives

1. **Fairness:** Ensure all processes get a reasonable share of CPU time without starvation.
2. **Priority Enforcement:** Honour priority levels so higher-priority tasks get faster service.
3. **Efficiency:** Keep the CPU busy as much as possible to maximize throughput.
4. **Responsiveness:** Provide low latency for interactive and time-sensitive processes.
5. **Encourage Good Behaviour:** Reward I/O-bound or cooperative processes to improve system performance.
6. **Graceful Degradation under Load:** Maintain acceptable performance even during heavy workloads.

Optimization Criteria

- **Maximize CPU Utilization:** Keep the CPU active as much as possible.
- **Maximize Throughput:** Complete the largest possible number of processes per unit time.
- **Minimize Response Time:** Lower the delay between a request submission and the first output/response.
- **Minimize Turnaround Time:** Reduce the total time from process submission to completion.
  - Turnaround Time = Finish Time – Arrival Time
- **Minimize Waiting Time:** Reduce the time processes spend in the ready queue.
  - For Non preemptive Algos = Start Time – Arrival Time
  - For Preemptive Algos = Finish Time – Arrival Time – Burst Time

First Come First Serve (FCFS)

First Come First Serve (FCFS)

Process that **requests** the CPU **FIRST** is **allocated** the CPU **FIRST**

- Simplest CPU scheduling algorithm.
- Non preemptive.
- The process that requests the CPU first is allocated the CPU first.
- Implemented with a FIFO queue. When a process enters the Ready Queue, its PCB is linked on to the tail of the Queue. When the CPU is free it is allocated to the process at the head of the Queue.

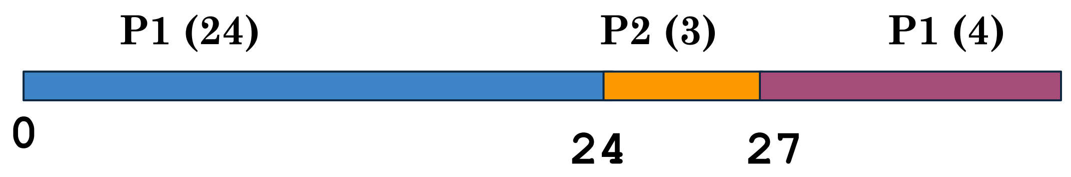
- **Limitations**
  - FCFS favor long processes as compared to short ones. (Convoy effect).
  - FCFS tends to favor processor bound processes over I/O bound processes.
  - Average waiting time is often quite long.
  - FCFS is non-preemptive, so it is troublesome for time sharing systems.

A **convoy effect** happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system

FCFS – Example 1

Processes	Duration/B.T	Order	Arrival Time
P1	24	1	0
P2	3	2	3
P3	4	3	4

Final schedule:



P1 waiting time: 0-0
P2 waiting time: 24-3
P3 waiting time: 27-4

The average waiting time:
(0+21+23)/3 = 14.667

FCFS – Example 2

Draw the **graph** (Gantt chart) and compute **average waiting time** for the following processes using FCFS Scheduling algorithm.

Processes	Arrival Time	Duration/B.T
P1	1	16
P2	5	3
P3	6	4
P4	9	2

SJF & SRTF Scheduling

SJF & SRTF Scheduling

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 bursts, FCFS scheduling is used to break the tie.

Shortest Job First (SJF)

- It's a *non preemptive* algorithm. When a new process arrives having a shorter next CPU burst than what is left of the currently executing process, it allows the currently running process to finish its CPU burst.

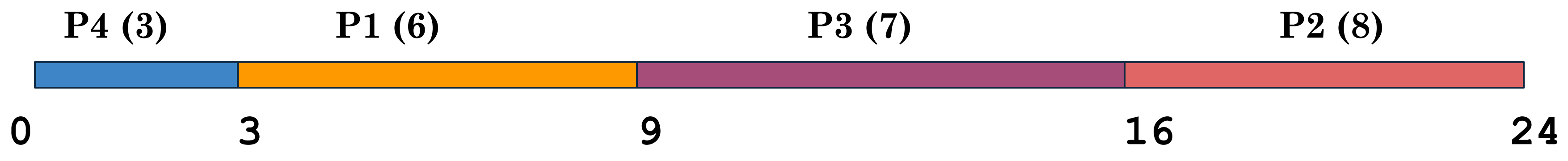
Shortest Remaining Time First (SRTF)

- It's a *preemptive* algorithm. When a new process arrives having a shorter next CPU burst than what is left of the currently executing process, it preempts the currently running process.

SJF – Example 3

Processes	Duration/B.T	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

Final schedule:



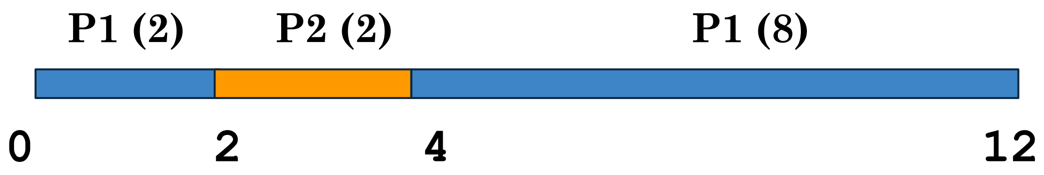
P4 waiting time: 0-0
P1 waiting time: 3-0
P3 waiting time: 9-0
P2 waiting time: 16-0

Total running time: 24
The average waiting time:
(0+3+9+16)/4 = 7 time units

SRTF – Example 4

Processes	Duration/B.T	Order	Arrival Time
P1	10	1	0
P2	2	2	2

Final schedule:



P1 waiting time: 4-2 =2
P2 waiting time: 0

The average waiting time (AWT):
(0+2)/2 = 1

Now run this using SJF!

SJF & SRTF – Example 5

Draw the graph (Gantt chart) and compute **waiting time** and **turn around time** for the following processes using **SJF** & **SRTF** Scheduling algorithm. For SJF consider no arrival time and consider all processes arrive at time 0 in sequence `P1, P2, P3, P4`.

Processes	Arrival Time	Duration/B.T
P1	0	8
P2	1	4
P3	2	9
P4	3	5

SJF & SRTF – Example 6

Draw the **graph** (Gantt chart) and **compute waiting time** and **turn around time** for the following processes using **SJF & SRTF** Scheduling algorithm.

Processes	Arrival Time	Duration/B.T
P1	0	5
P2	1	2
P3	2	3
P4	3	1

SJF & SRTF – Example 7

Draw the **graph** (Gantt chart) and **compute waiting time** and **turn around time** for the following processes using **SJF & SRTF** Scheduling algorithm.

Processes	Arrival Time	Duration/B.T
P1	0	9
P2	3	6
P3	6	2
P4	9	1

SJF & SRTF Scheduling (Cont…)

$100 QUESTION

How to compute the next CPU burst?

- One approach is to try to approximate. 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. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst

Exponential Averaging

Exponential Averaging assumes that future CPU bursts will be similar to past ones, allowing SJF and SRTF to choose processes with the shortest predicted burst.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\, \tau_n$$

Where,

t_n	=	Actual length of n^{th} (last) CPU burst
τ_n	=	Estimate for n^{th} (last) CPU burst
τ_{n+1}	=	Estimate for $n+1^{st}$ CPU burst
α	=	Weight factor ($0 \leq \alpha \leq 1$)

If $\alpha = 0$: $\tau_{n+1} = \tau_n$ (Next CPU burst estimate will be equal to previous CPU burst estimate)

If $\alpha = 1$: $\tau_{n+1} = t_n$ (Next CPU burst estimate will be equal to previous actual CPU burst)

Typical value used for α is ½. With this value, our $(n+1)^{st}$ estimate is given below, where you can note that older CPU bursts are given exponentially less weight-age :

$$\tau_{n+1} = \frac{t_n}{2} + \frac{t_{n-1}}{2^2} + \frac{t_{n-2}}{2^3} + \frac{t_{n-3}}{2^4} + \dots$$

Priority Scheduling

Priority Scheduling

A **priority number** (integer) is **associated** with each **process** and the CPU is allocated to the process with the highest priority (**smallest integer highest priority**)

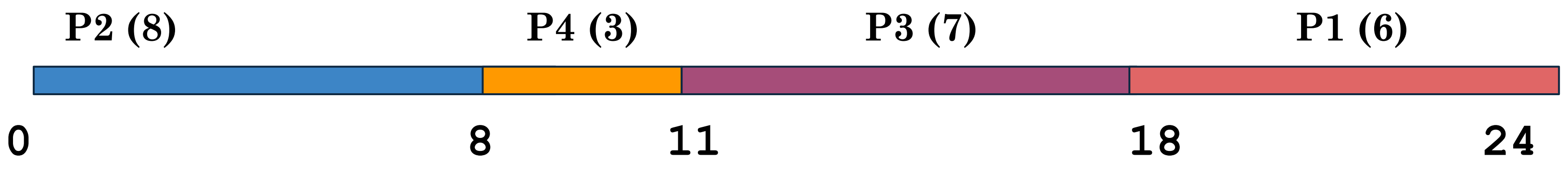
- A priority scheduling algorithm can be preemptive or non preemptive:
  - A **Preemptive priority scheduling algorithm** will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. SRTF is a priority scheduling algorithm where priority is the predicted next CPU burst time.
  - A **Non preemptive priority scheduling algorithm** will simply put the new process at the tail of the Ready Queue.

- **Problem:**
  - **Starvation / Indefinite Blocking:** i.e. Low priority processes may never execute.
- **Solution:**
  - **Aging:** A technique of gradually increasing the priority of processes that wait in the system for long time

Priority Scheduling – Example 8

Lower priority # == More important

Processes	Duration/B.T	Priority	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0



P2 waiting time: 0
P4 waiting time: 8
P3 waiting time: 11
P1 waiting time: 18

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF's)

Round Robin Scheduling Algorithm

Round Robin (RR) Scheduling

Round Robin (RR) is a **preemptive** scheduling algorithm where each process gets a **fixed time slice** (quantum). If a process doesn't finish in its turn, it's moved to the end of the queue.

- RR is a preemptive scheduling algorithm designed for time-sharing systems.
- A clock interrupt occurs at fixed intervals called the time quantum or time slice.
- The ready queue is managed as a FIFO structure.
- The scheduler assigns the CPU to the first process and sets a timer for 1 time quantum.
- If the process finishes before the time quantum, it releases the CPU voluntarily.
- If not, a timer interrupt triggers a context switch, and the process is moved to the end of the queue.
- The scheduler then picks the next process in the queue.

Round Robin (RR) Scheduling (Cont…)

The performance of the RR algorithm depends heavily on the size of the time quantum. Principal design issue in the length of the time quantum to be used are:

- If the time quantum is very large (larger than the largest CPU burst of any process), the RR policy become the FCFS policy.
- If the time quantum is very short, the RR approach is called processor sharing. Short processes will move through the system relatively quickly.

Round Robin (RR) Scheduling (Cont…)

Limitation

- Processor bound processes tend to receive an unfair portion of processor time, which result in poor performance of I/O bound processes.

Effect of context switching on the performance of RR scheduling

Consider a single process requiring 10 CPU time units:

- If the time quantum is 12 units, the process completes in a single quantum, incurring no context-switch overhead.
- If the quantum is 6 units, the process requires two quanta, leading to one context switch.
- If the quantum is 1 unit, the process will need 10 quanta, resulting in nine context switches, significantly increasing overhead.

To maintain efficiency, the time quantum should be large relative to the context-switch time. For example, if context switching takes about 5% of the quantum, then approximately 5% of total CPU time is lost to switching, keeping overhead manageable.

RR – Example 9

Draw the graph (Gantt chart) and compute turnaround time for the following processes using RR Scheduling algorithm. Consider a time slice of 4 sec.

Processes	Arrival Time	Duration/B.T
P1	0	16
P2	1	3
P3	2	3

RR – Example 10

Draw the **graph** (Gantt chart) and compute **turnaround time** for the following processes using RR Scheduling algorithm. Consider a time slice of **3 sec**

Processes	Arrival Time	Duration/B.T
P1	0	8
P2	2	4
P3	3	2
P4	6	5

Note: Priority of placing a process in Ready Queue

- New Process.
- Running Process.

RR with I/O – Example 11

Draw the graph (Gantt chart) and compute turnaround time for the following processes using RR Scheduling algorithm. Consider a time slice of 3 sec. Every even number process perform I/O after every 2 sec of its running life. I/O takes 10 seconds.

Processes	Arrival Time	Duration/B.T
P1	0	8
P2	2	3
P3	3	5
P4	6	4

Note: Priority of placing a process in Ready Queue

- New Process.
- Blocked Process or I/O process.
- Running Process.

RR with I/O – Example 12

Draw the graph (Gantt chart) and compute turnaround time for the following processes using RR Scheduling algorithm. Consider a time slice of 3 sec. Every odd number process perform I/O after every 2 sec of its running life. I/O takes 10 seconds.

Processes	Arrival Time	Duration/B.T
P1	0	8
P2	2	3
P3	3	5
P4	6	4

RR with I/O – Example 13

Schedule the following processes using RR. The processes P1, P2 and P3 have arrived at time units 0, 1 and 2 respectively. The process P1 demands CPU for 3 time quantum before going for I/O for 6 time quantum, then again demand CPU for 2 quantum and then goes for I/O for 4 quantum and finally demand CPU for 4 CPU quantum before it terminate. Assume RR time slice of 4 time units.

Processes	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst
P1	3	6	2	4	4
P2	5	4	3		
P3	6	8	5	7	2

To Do

- Watch OS video on Process Scheduling:

https://www.youtube.com/watch?v=3ap2kU4bA9E&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=12

- Watch SP video on Process Scheduling:

https://www.youtube.com/watch?v=Y86pa2nrT_k&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=21

Coming to office hours does NOT mean that you are academically weak!