Process Concept & Scheduling

Swati Mali
Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Outline

- Basic Terminology
- Process

Basic Terminologies & Definitions

Task

- A task is a unit of assigned work. It can also be defined as the unit of programming controlled by an operating system (OS). Depending on the OS design, the task may involve one or more processes.
- E.g.
  - Download a file from the internet
  - Bake a cake

Program

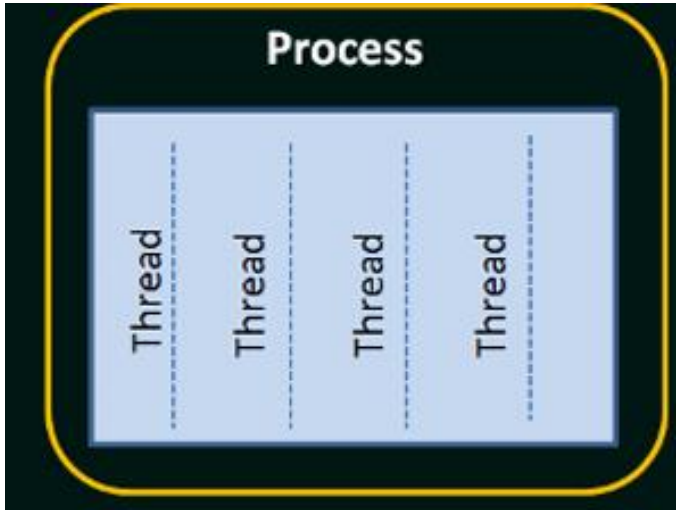
- A program is a **sequence of instructions** written to accomplish a task.
- A program may comprise **one or more processes** depending on the statement being executed.
- It is generally referred to as a **passive entity** that does not perform any action until it is executed.
- E.g.
  - A particular recipe given in a book
  - A web application developed

Job

- A job is a unit of work submitted by a user to the system.
- A job may be interactive or a batch job, which may in turn consist of one or more processes.
- E.g.
  - A user submitting a print job to a printer
  - Executing a scheduled task to back up files

Process and Thread

- A process is an instance of a program in execution.
- It is a basic unit of work that can be scheduled and executed by the operating system.
- A thread is the unit of execution within a process.
- A process can have anywhere from just one thread to many threads.



Process Mode

A process can execute in either user mode or kernel mode, with the processor switching between these modes depending on the code being executed.

- User Mode: The process runs with limited privileges, typically to prevent it from performing harmful operations.
- Kernel Mode: The process runs with full access to all system resources and hardware, necessary for performing critical tasks.
- E.g.: Typing / Editing document

Process Context

- When a running process is taken away from the processor, certain information about its state needs to be saved to allow it to resume execution later.
- The process context includes:
  - address space,
  - stack space,
  - virtual address space,
  - register set image i.e. Program Counter, Stack Pointer, Program Status Word, Instruction Register and other general processor registers,
  - accounting information,
  - associated kernel data structures and
  - current state of the process (waiting, ready, etc).

Event

- An activity that is happens or is expected to happen.
- Generally this a software message exchanged when the activity occurs.
- In operating systems, the events may cause the processes to change their state.
  - e.g. mouse click, file lock reset, etc.

- Check- event viewer in windows OS

Process Concept

- Process priority:
  - In a multiprogramming system, the processes are assigned numerical privileged values indicating their relative importance and/or urgency and/or value.
- Preemptive:
  - Preemption is the ability of the system to take over a currently executing process by another one (possibly with high privileged one) with an intention to resume the preempted process later on.
- Non-preemptive:
  - Non-preemptive entities are the one those cannot be taken over by other entities

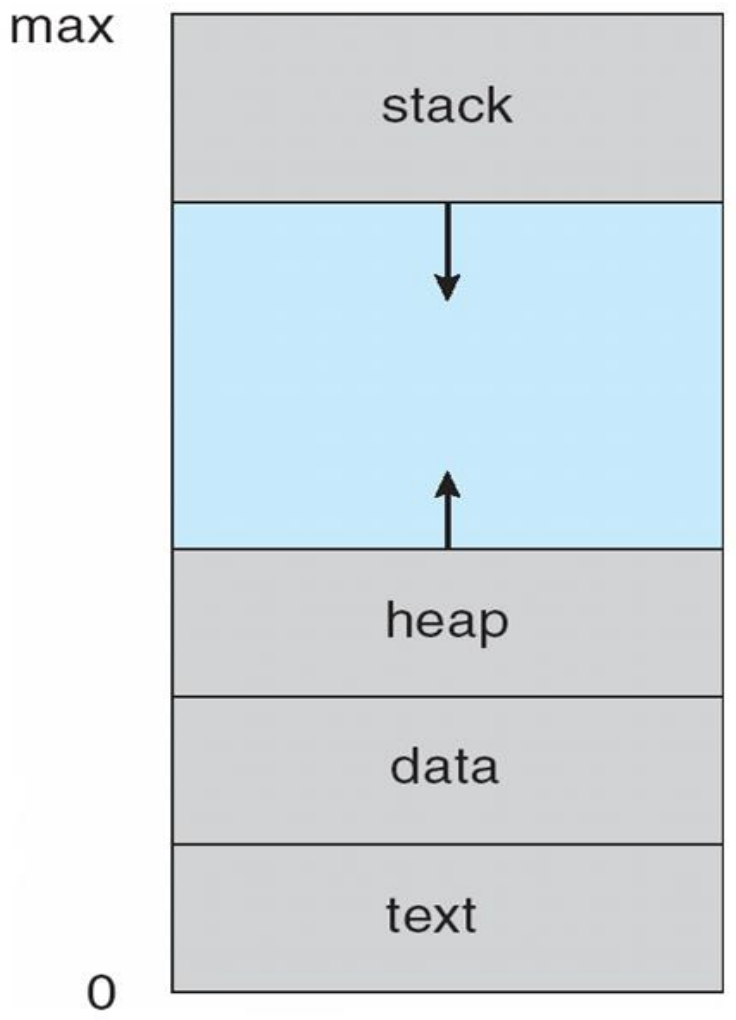
Process

Process Component

- The program code, also called **text section**
- Current activity including **program counter**, registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time

Process in Memory

- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory

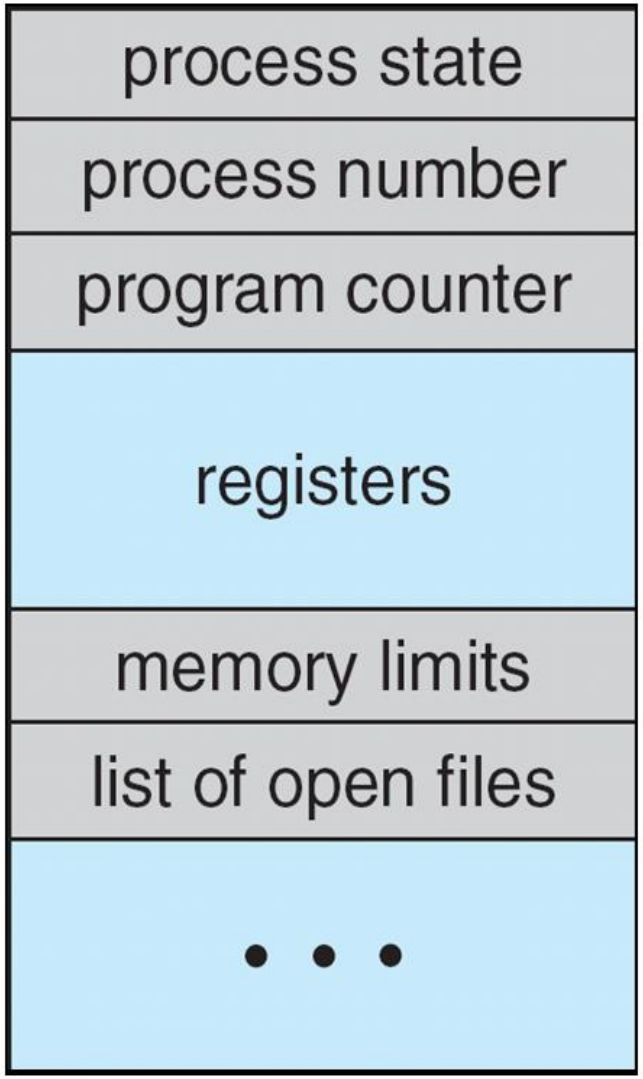


Process Control Block (PCB)

- **Information associated with each process (also called task control block)**
- Repository of any information related to Process.
- PCB components:
  - Identifiers
  - Processor State Information
  - Process Control Information

Process Control Block (PCB)

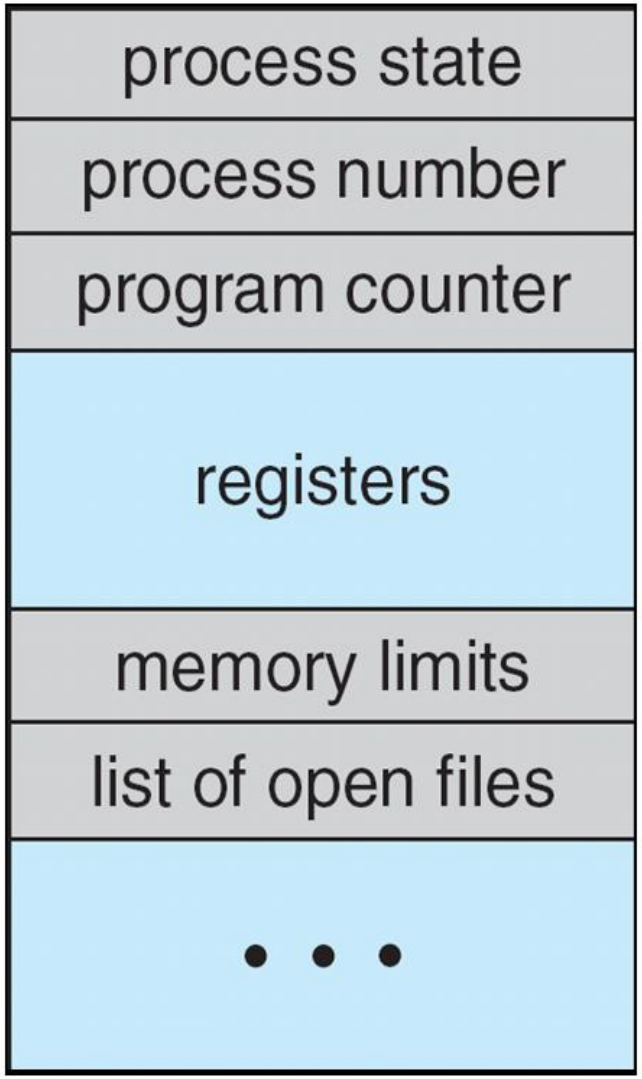
- **Process Identifier/Number -** An identifier that helps us in identifying and locating a process.
- **Process state –**It identifies the state that the process is currently in. It could be running, waiting, etc
- **Program counter –** address of the next instruction to be executed for this process.



Process Control Block (PCB)

CPU registers –

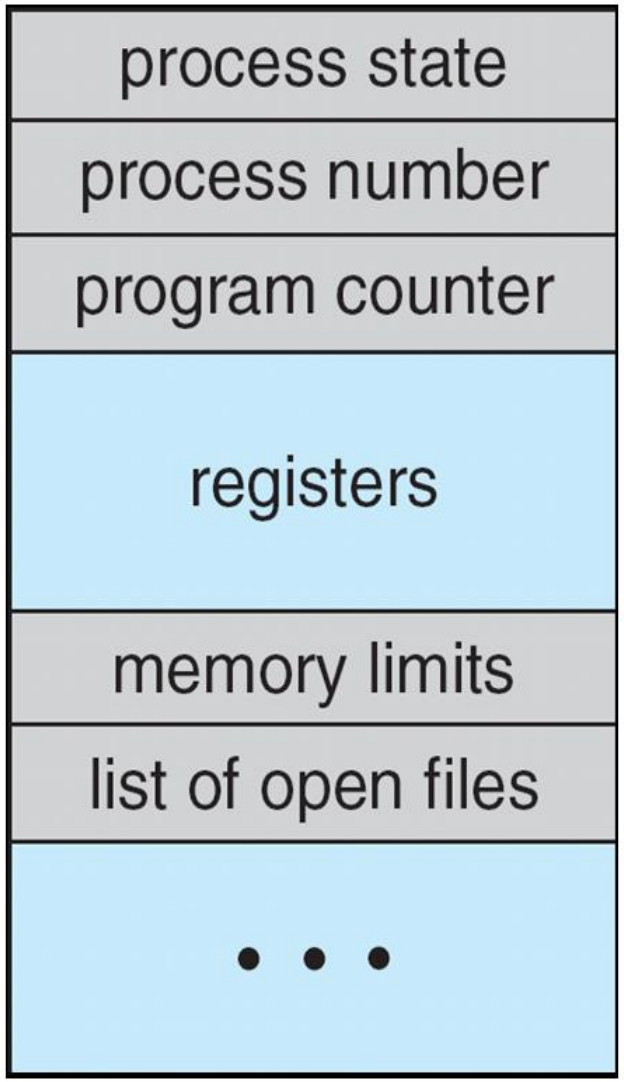
- contents of all process-centric registers,
- **Registers may vary in number and type depending on system architecture.**



Process Control Block (PCB)

CPU scheduling information-

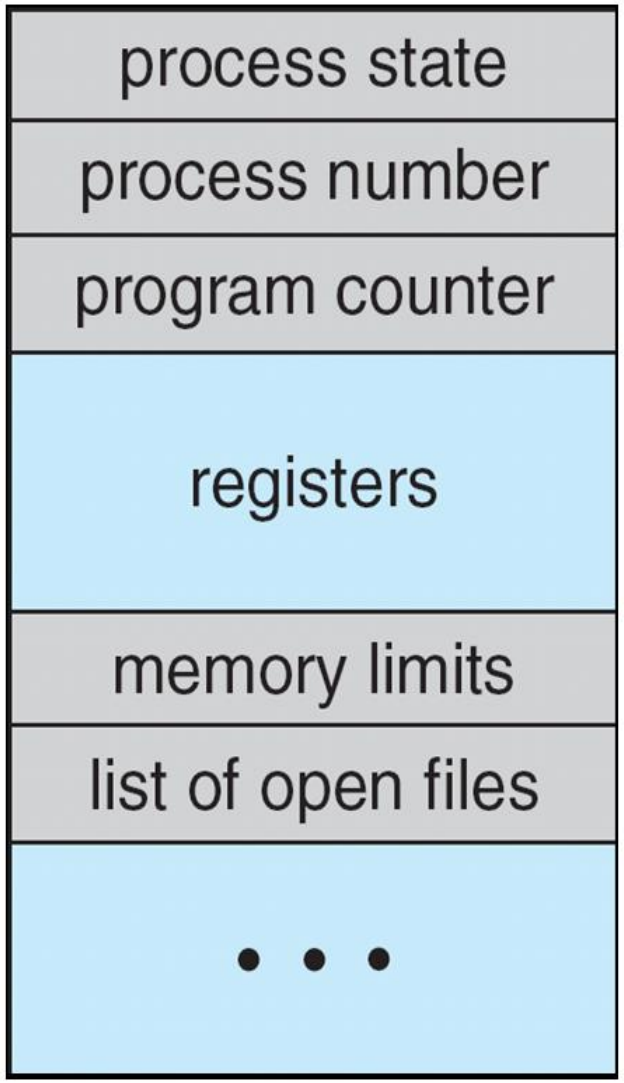
- Process priorities,
- scheduling queue pointers,
- any other scheduling parameters.



Process Control Block (PCB)

Memory-management information –

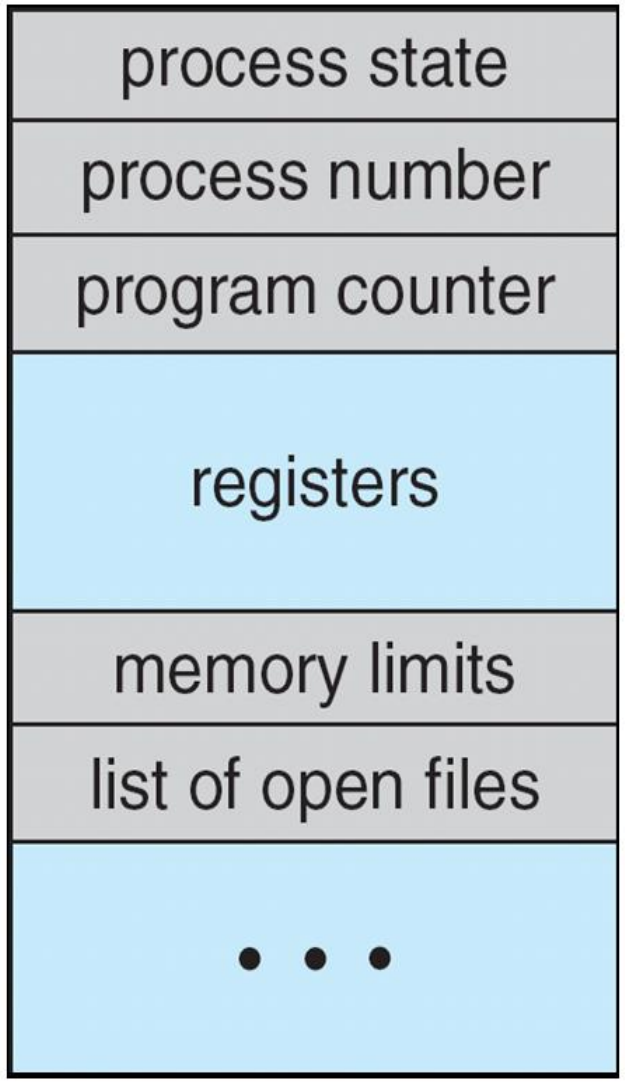
- memory allocated to the process,
- value of the Base and limit registers,
- the page tables or segment tables.



Process Control Block (PCB)

Accounting information –

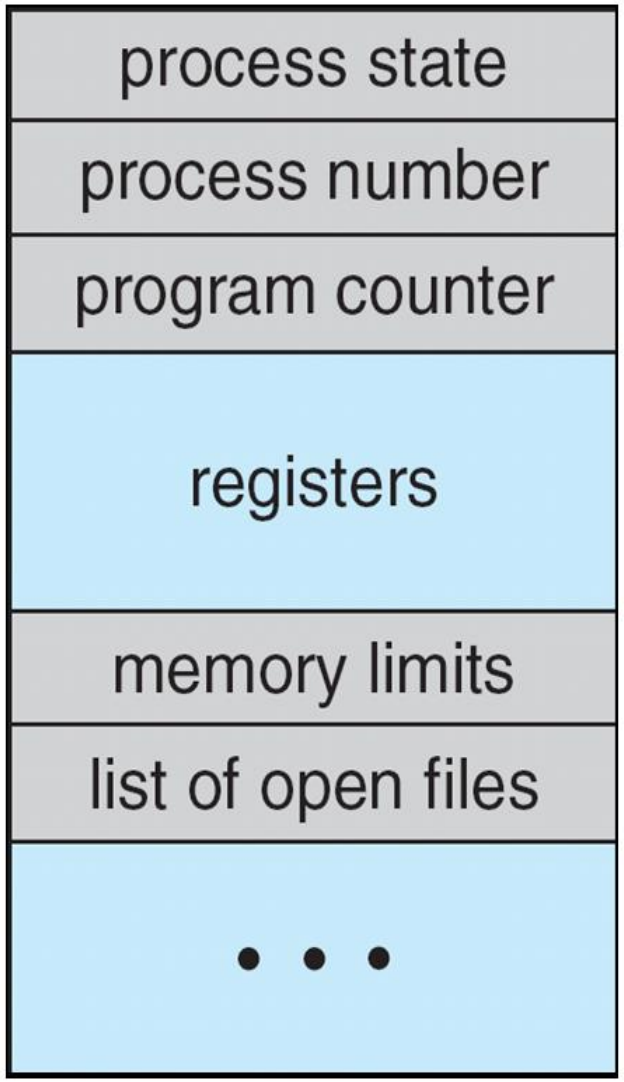
- CPU used,
- clock time elapsed since start, time limits,
- job or process numbers



Process Control Block (PCB)

I/O status information –

- list of I/O devices allocated to process,
- list of open files



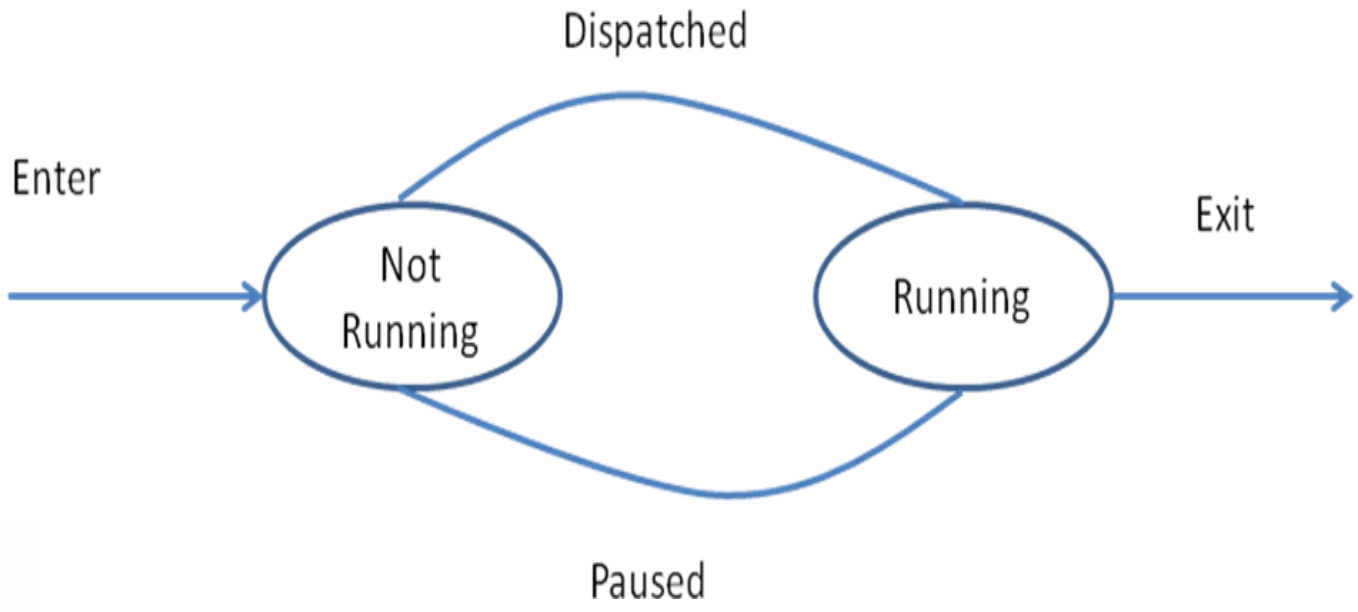
Process State

As a process executes, it changes *state*

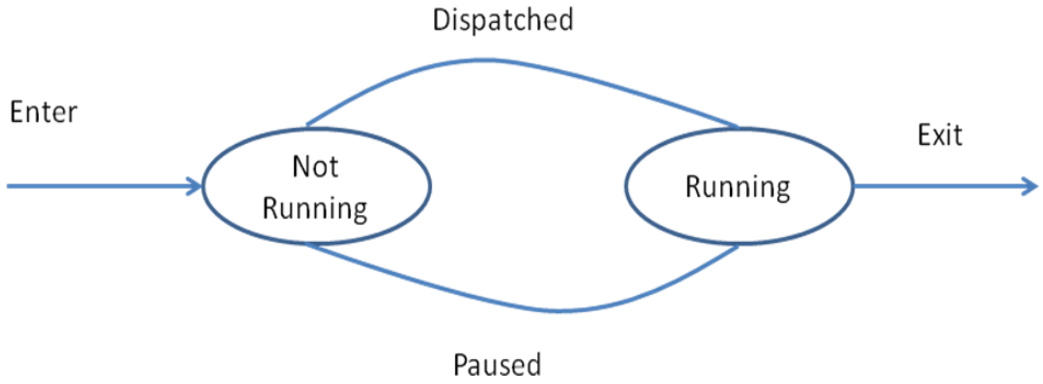
- New
  - When process creation is taking place, the process is in a new state.
- Ready
  - During this state, the process is loaded into the main memory and will be placed in the queue of processes **which are waiting for the CPU allocation**.
- Running
  - Instruction is being executed.
- waiting
  - The process is waiting for some event to occur (such as an I/O completion)
- Terminated
  - The process has finished execution.

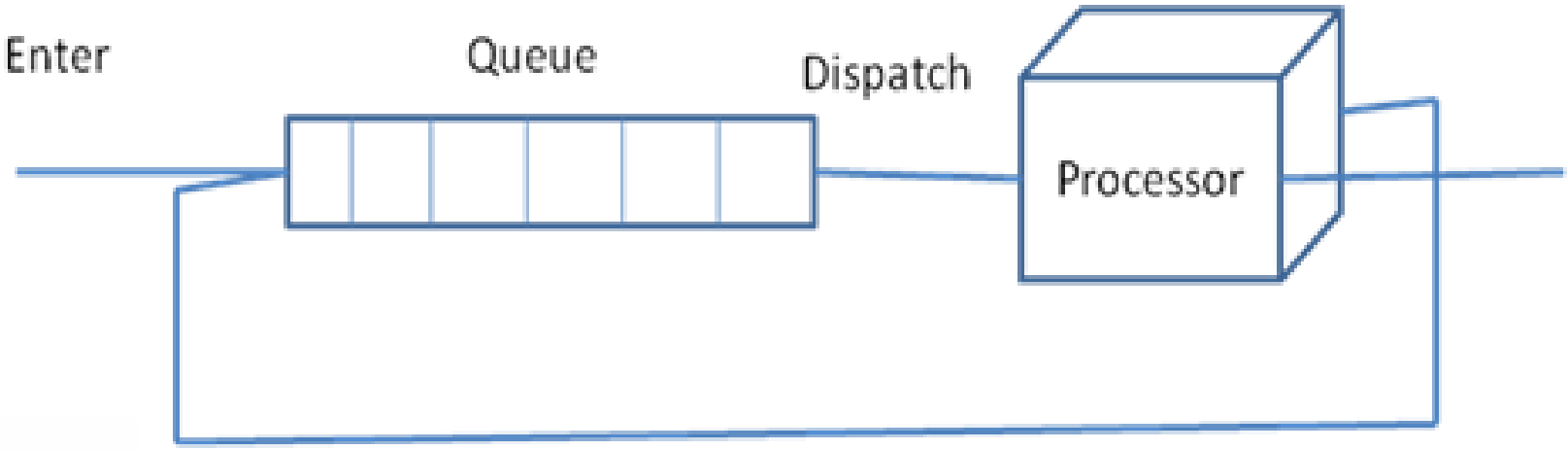
Process state transition

- A state transition for process is defined as a change in its state.
- The state transition occurs in response to some event in the computing system.
- E.g. Two State Model - State transition diagram

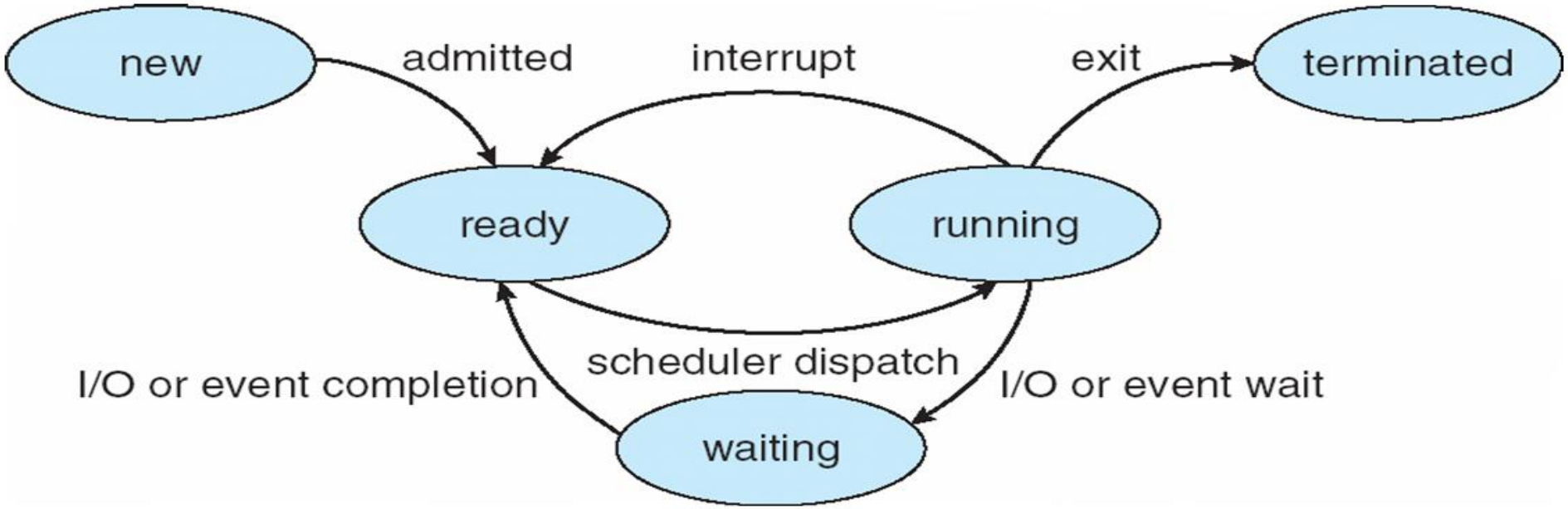


Two State Model – Queuing Diagram

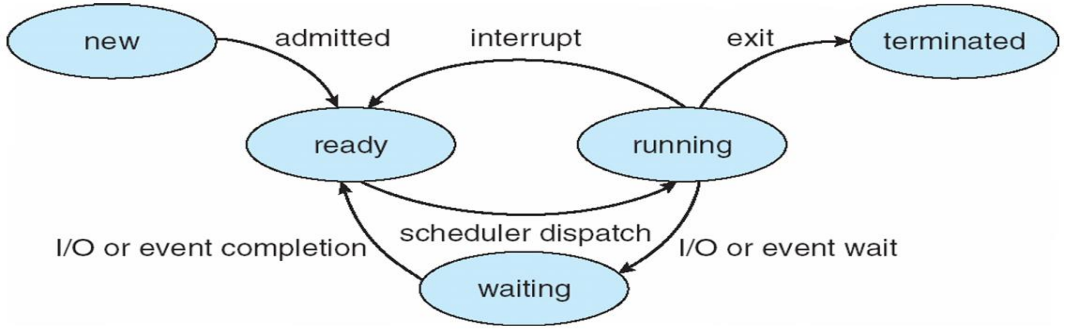




Five State Model

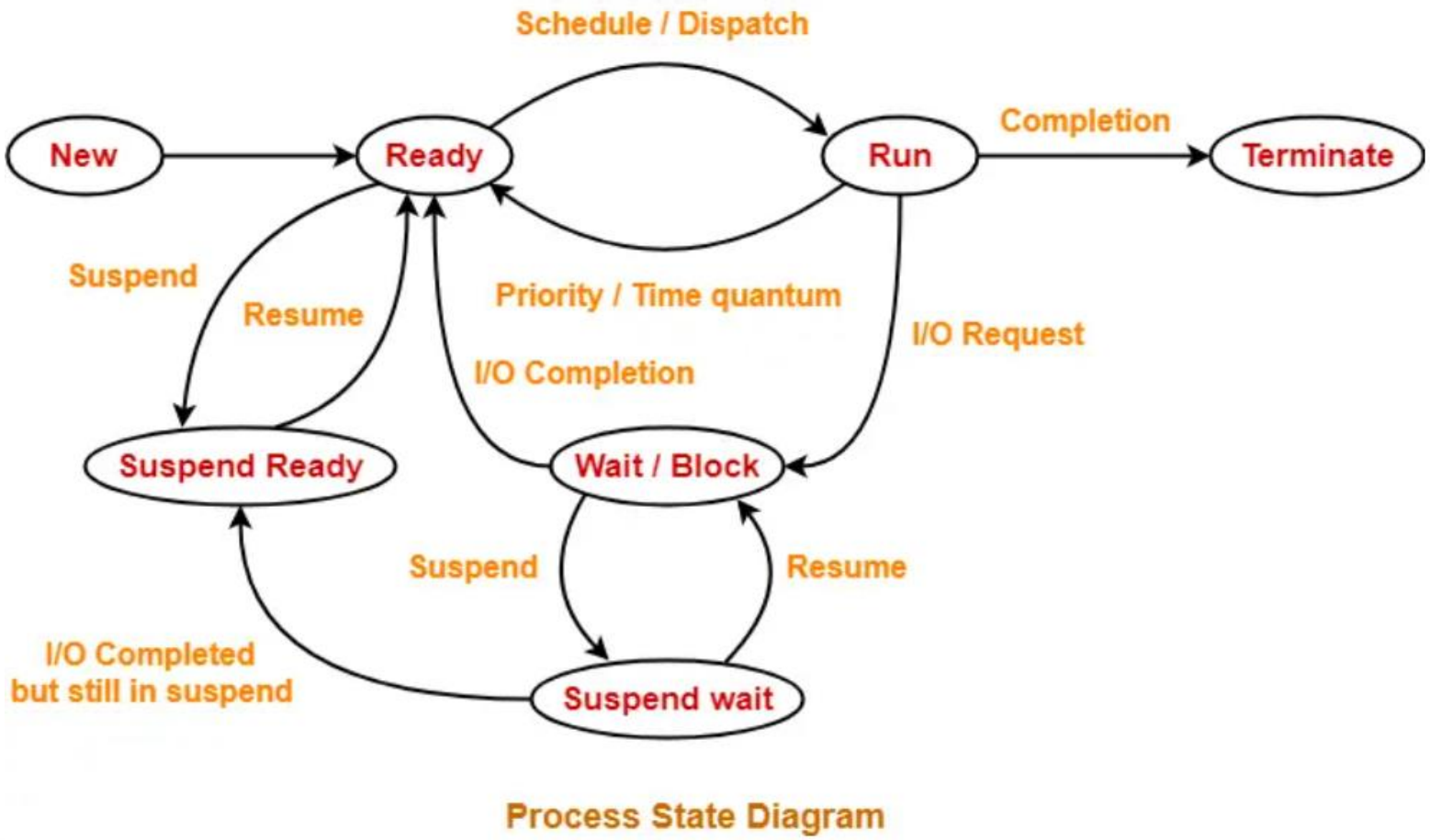


Five State Model



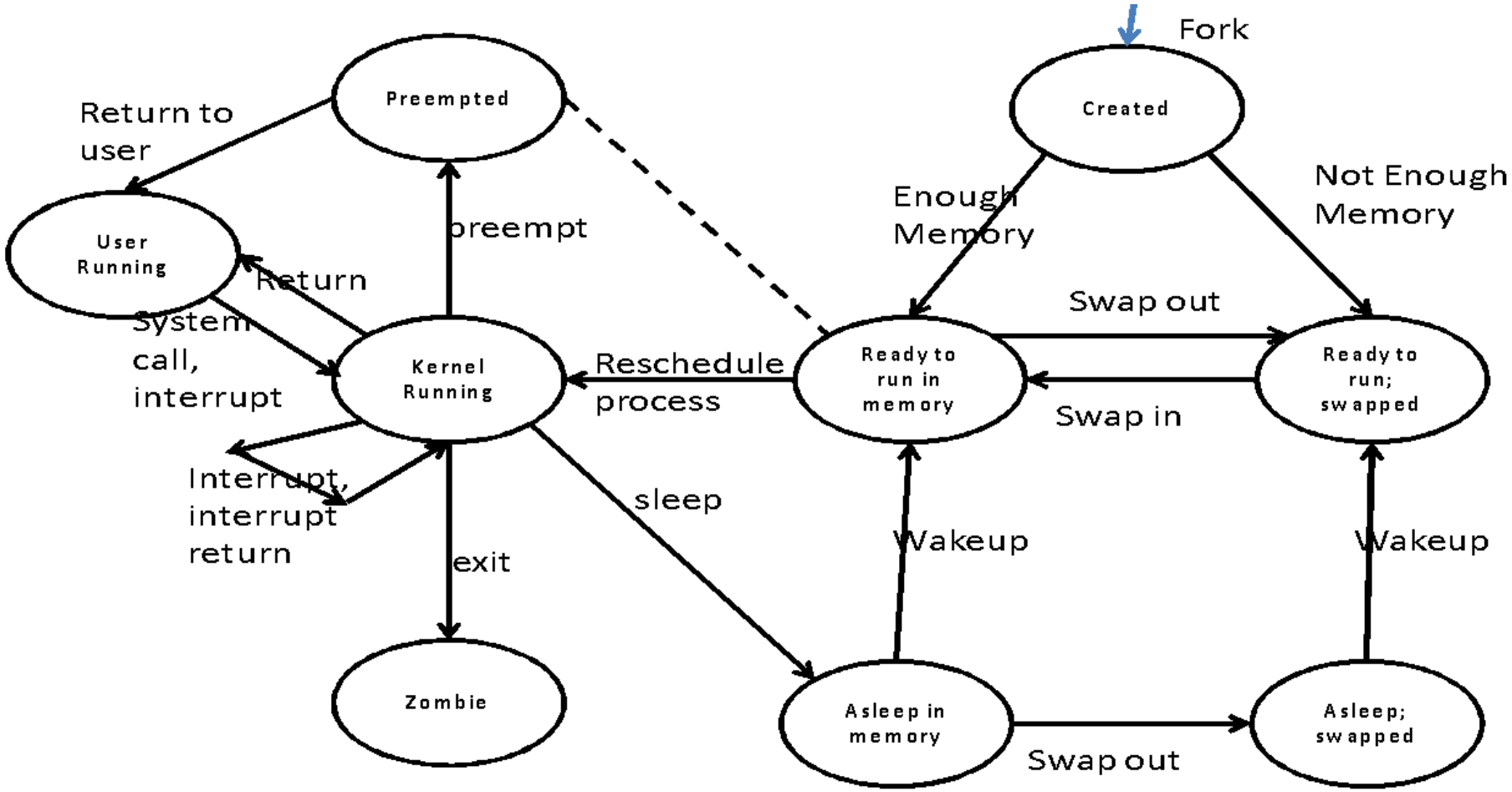
- Need of swapping (Suspend State)
  - because blocked processes hog up the main memory.
  - Swap them out from main memory to disk.
- There are two independent concepts
  - whether a process is waiting on an event (blocked or not),
  - whether a process has been swapped out of main memory (suspended or not)

7-state process model



Process State Diagram

9-state process model



State Transition

State transition	Description
Null New	A new process is created to execute a program.
New ready	The OS may move a process from New to Ready state depending on the predefined maximum number of processes allowed (Degree of multiprogramming).
Ready Running	The process is scheduled by dispatcher. The CPU starts or resumes execution of the instruction codes
Blocked Ready	The request initiated by process is satisfied or the event on which it is waiting occurs.

State Transition (cont...)

State transition	Description
Running → Ready	The process is preempted by the OS decides to execute some other process. This transition takes place may be because of expiration of time quantum or arrival of high priority process.
Running → Blocked	The running process makes request for resource(s) or needs some event to occur to proceed further. The process then calls for a system call to indicate its wish to wait till the resource or the event becomes available.
Running → Termination	The program execution is completed or terminated.

State Transition (cont...)

State transition	Description
Running Blocked	The running process makes request for resource(s) or needs some event to occur to proceed further. The process then calls for a system call to indicate its wish to wait till the resource or the event becomes available.
Running Termination	The program execution is completed or terminated.

Causes of process initiation

- **Interactive logon:** when a user logs into the system, a new process is created.
- **Created by OS to provide some service:** The OS initiates a process to perform the service requested by user directly or indirectly, without making the user to wait.
- **Spawned by an existing process**: To support Modularity and/or parallelism, a user program can create some number of new processes.
- **Program Execution**: Whenever you open an application, the OS creates a process to run it

How the OS creates a process?

1. Create a process
2. Assign a unique process ID to newly created process
3. Allocate the memory and create its process image
4. Initialize process control block
5. Set the appropriate linkages to the different data structures such as ready queue etc.
6. Create or expand the other data structures if required

Causes of process blocking

- Process requests an I/O operation
- Process requests memory or some other resource
- Process wishes to wait for a specific interval of time
- Process waits for message from some other process
- Process wishes to wait for some action to be performed by another process.

Causes of process termination

- **Normal Completion:** The process executes an OS system call to intimate that it has completed its execution.
- **Self termination** (e.g. incorrect file access privileges, inconsistent data)
- **Termination by the parent process:** a parent process calls a system call to kill/terminate its child process when the execution of child process is no longer necessary.
- **Exceeding resource utilization:** An OS may terminate a process if it is holding resources more than it is allowed to. This step can also be taken as part of deadlock recovery procedure.

Causes of process termination (cont...)

- **Abnormal conditions during execution:** the OS may terminate a process if an abnormal condition occurs during the program execution. (e.g. memory protection violation, arithmetic overflow etc)
- **Deadlock detection and recovery**

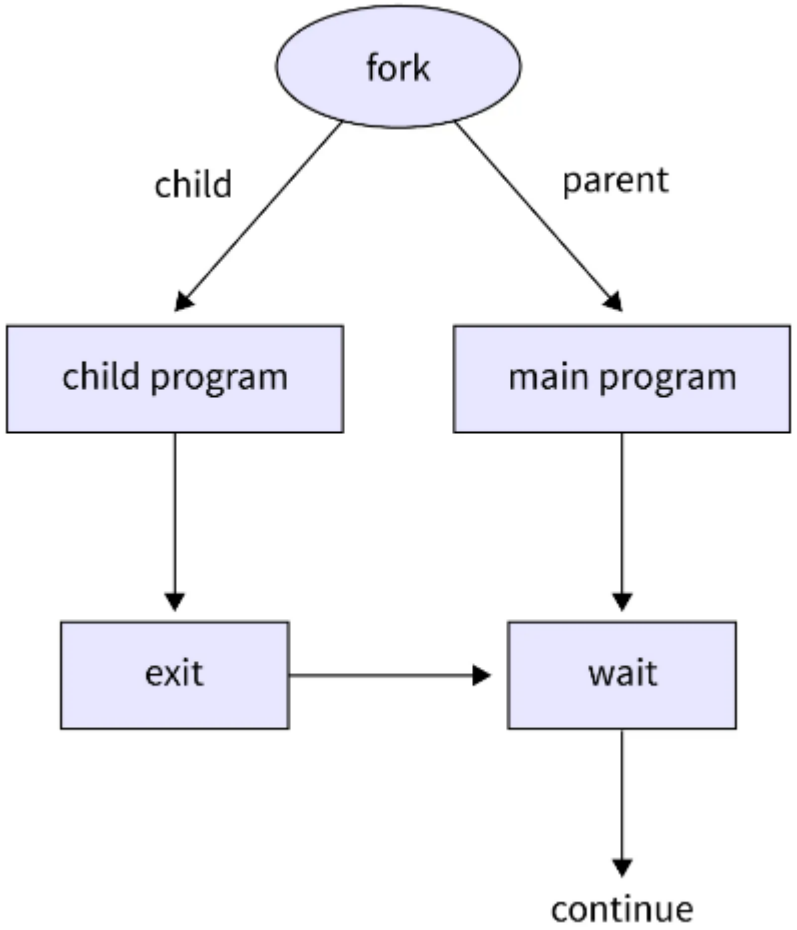
Process Creation in UNIX

- One process can create another process, perhaps to do some work for it.
  - The original process is called the ***parent***
  - The new process is called the ***child***
  - The child is an (almost) identical **copy** of parent (same code, same data, etc.)
  - The parent can either wait for the child to complete, or continue executing in parallel (concurrently) with the child

Process Creation in UNIX (cont...)

- In UNIX, a process creates a child process using the system call fork( )
  - **Negative:** A child process could not be successfully created if the fork() returns a **negative** value.
  - **Zero:** A new child process is successfully created if the fork() returns a **zero**.
  - **Positive:** The positive value is the process ID of a child's process to the parent. The process ID is the type of pid_t that is defined in OS or sys/types.h.
- Child often uses exec( ) to start another completely different program

Fork()


fork
child
parent
child program
main program
exit
wait
continue

Example - Process Creation

```c
#include <sys/types.h>
#include <stdio.h>
int a = 6; /* global (external) variable */
int main(void)
{
  int b; /* local variable */
  pid_t pid; /* process id */
  b = 88;
  printf("..before fork\n");
  pid = fork();
  if (pid == 0) { /* child */
     a++;  b++;
  } else /* parent */
     wait(pid);
  printf("..after fork, a = %d, b = %d\n", a, b);
  exit(0);
}
```

```
..before fork
..after fork, a = 7, b = 89
..after fork, a = 6, b = 88
```

System Calls for Process Management

Sr No	System Call	Description
1	fork()	This system calls creates a new process.
2	exec()	This call is used to execute a new program on a process.
3	wait()	This call makes a process wait until some event occurs.
4	exit()	This call makes a process to terminate
5	getpid()	This system call helps to get the identifier associated with the process.
6	getppid()	This system call helps to get the identifier associated with the parent process.
7	nice()	The current process priority can be changed with execution of this system call.
8	brk()	This call helps to increase or decrease the data segment size of the process.
9	Kill()	The forced termination of any process can be executed with this system call.
10	Signal()	This system call is invoked for sending and receiving software interrupts

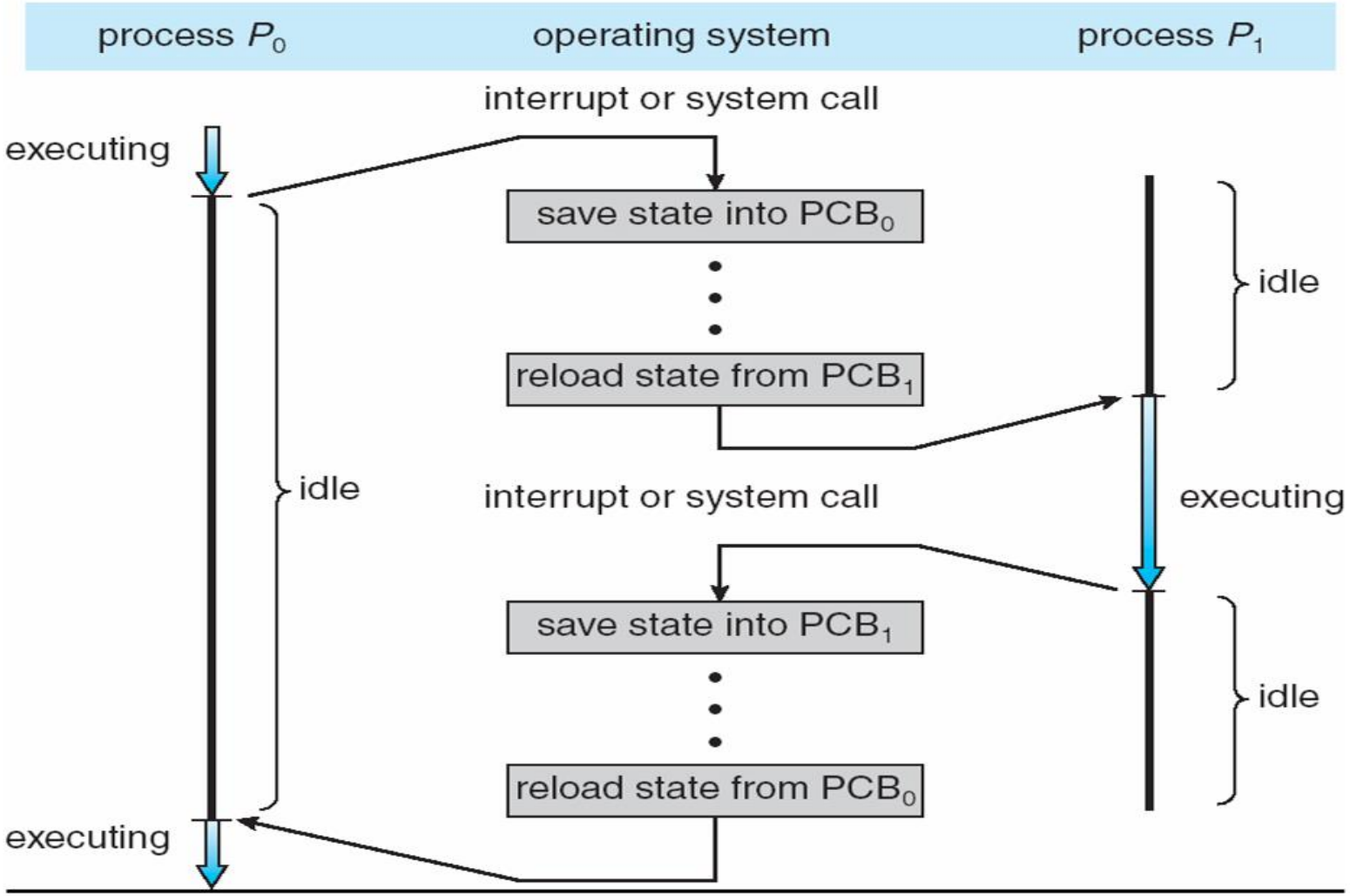
Context Switch

- Stopping one process and starting another is called a context switch
- When the OS stops a process, it stores the hardware registers (PC, SP, etc.) and any other state information in that process' PCB
- When OS is ready to execute a waiting process, it loads the hardware registers (PC, SP, etc.) with the values stored in the new process' PCB, and restores any other state information

Process context switch Vs mode switch

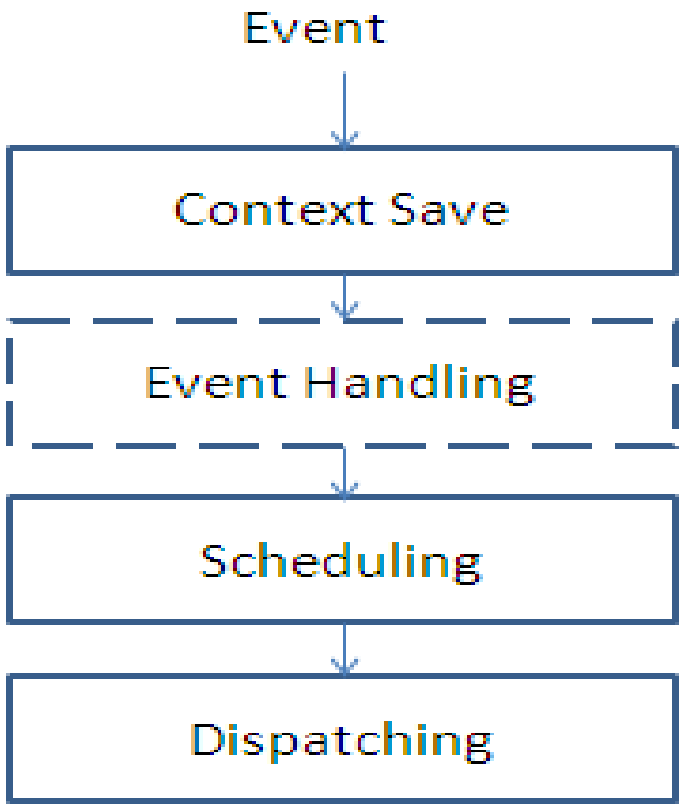
- Context Switch:
  - Execution of a process is stopped to respond an interrupt.
  - Needs to save Process Image of one process and load process image of the new process loaded.
  - The processes are switched and processes keep on changing their status as Running and Not running.
- Mode switch:
  - Every process may switch in between a low privileged user mode and high privileged kernel mode in its lifetime.
  - Process continues to execute even after mode switches.

CPU Switch From Process to Process



Fundamental kernel functions of process control

- Context save: Save information concerning an executing process when its execution gets suspended
- Scheduling: Choose the process as per the scheduling policy to be executed next on the CPU.
- Dispatching: Set up execution of the chosen process on the CPU.



Fundamental kernel functions of process control

- Occurrence of the event calls the context save functionality and an appropriate event handling procedure.
- Event handling may initiate some processes, hence the scheduling function gets invoked to choose the process and in turn,
- The dispatching function transfers control to the new process.

Control/Data structures maintained by OS to manage processes

- Memory Tables
- I/O Tables
- File Tables
- Process table

Control structures maintained by OS to manage processes

- **Memory Tables:**
  - Memory tables keep track of both main and secondary memory.
  - Active processes are stored in main memory and when required, they are moved to secondary memory through the mechanism called ‘swapping’.
  - The memory tables maintain the following information:
    - The main memory allocation to all processes in system
    - The secondary memory allocation to all processes in system
    - Shared memory regions in main and virtual memory and their attributes
    - Miscellaneous information required to manage virtual memory.

Control structures maintained by OS to manage processes

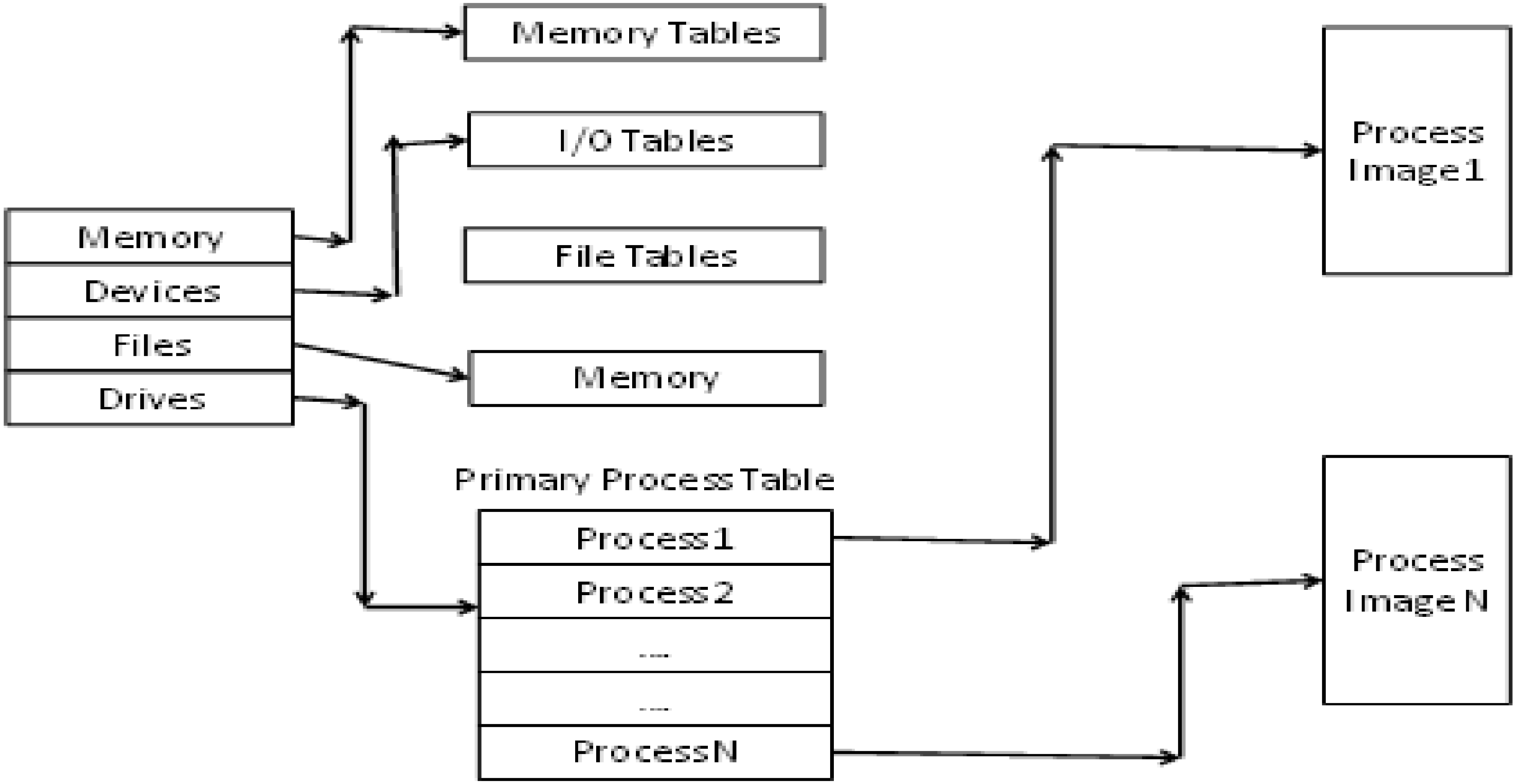
- **I/O Tables:**
  - I/O tables keep track of I/O devices and channels in the computing system.
  - The I/O devices are also resources required by processes.
  - So at any given instance, I/O devices may be available or allocated to a particular process.

Control structures maintained by OS to manage processes

- **File Tables:**
  - File tables keeps track of;
    - all files,
    - their locations on secondary memory,
    - their current statuses and
    - other attributes such security, sharing, etc.
  - Most of the operating systems, this information is maintained by a module called File Management System.

Control structures maintained by OS to manage processes

- **Process table:**
  - Process tables manage processes.
  - They maintain information of:
    - processes,
    - their child process references,
    - statuses,
    - allocated resources,
    - process contexts,
    - information required for process synchronization and so on.
  - These pieces of information are stored in process images.


Memory Tables
I/O Tables
File Tables
Memory
Memory
Devices
Files
Drives
Primary Process Table
Process 1
Process 2
...
...
Process N
Process Image 1
Process Image N

Different interaction mechanisms used by processes

Interaction Mechanism	Description
Data Sharing	The processes interact with each other by altering data values. If more than one processes update the data the same time, they may leave the shared in inconsistent state. So, shared data items are protected against simultaneous access to avoid such situation.
Message Passing	In this mechanism, the processes exchange information by sending messages to each other.
Synchronization	In certain computing environments, the processes are required to execute their actions in some particular order. To help this happen, the processes synchronize with each other to maintain their relative timings and execute in the desired sequence.
Signals	The processes may wait for events to occur. It can be intimated to processes through the signaling mechanism.

Question ?

Process Concept & Scheduling

Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Basic Concept
- Scheduling Criteria
- Scheduling Algorithm

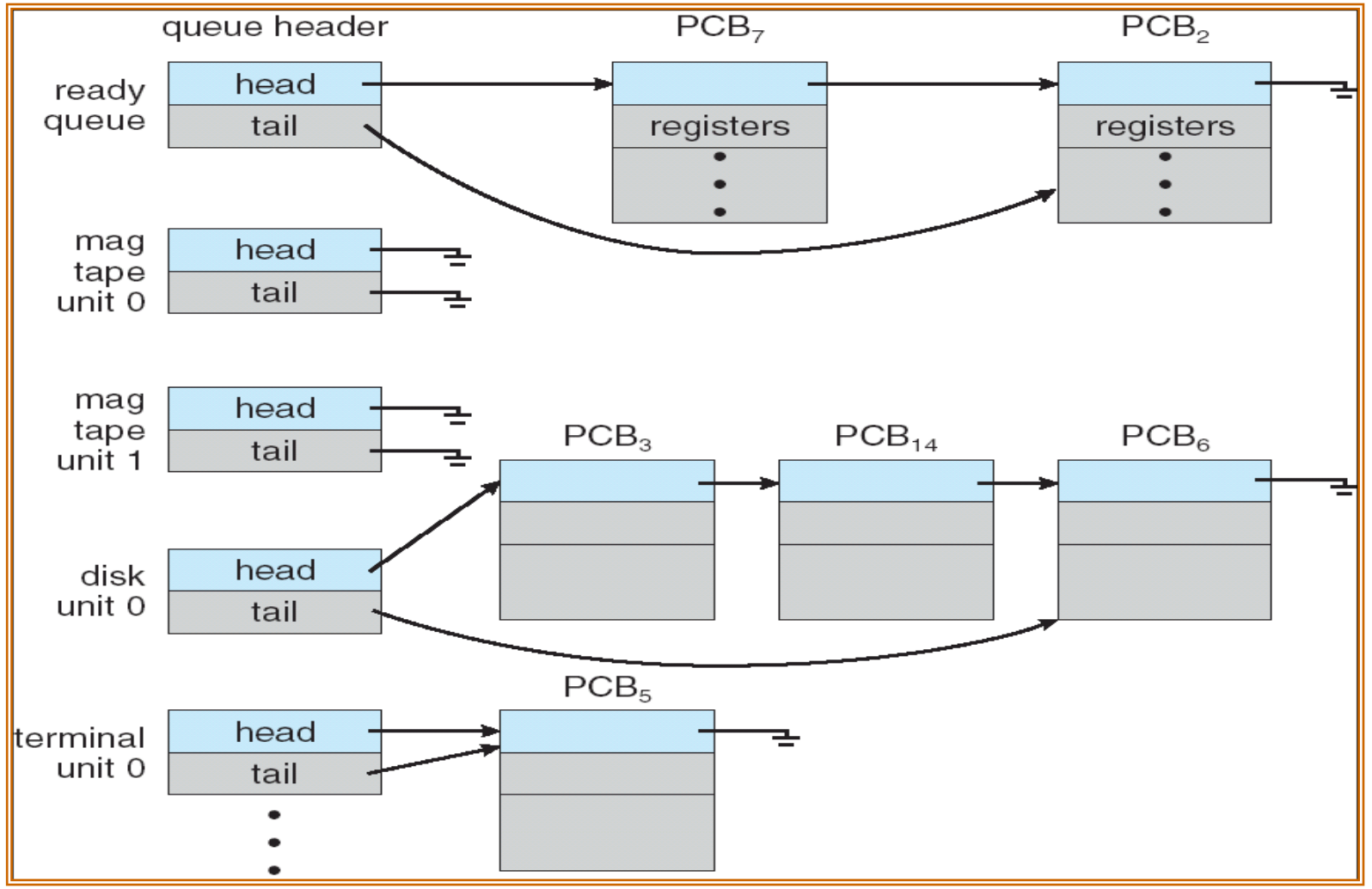
Process Scheduling

- The operating system is responsible for managing the *scheduling* activities.
  - A uniprocessor system can have only one running process at a time
  - The main memory cannot always accommodate all processes at run-time
  - The operating system will need to decide on which process to execute next (CPU scheduling), and which processes will be brought to the main memory (job scheduling)

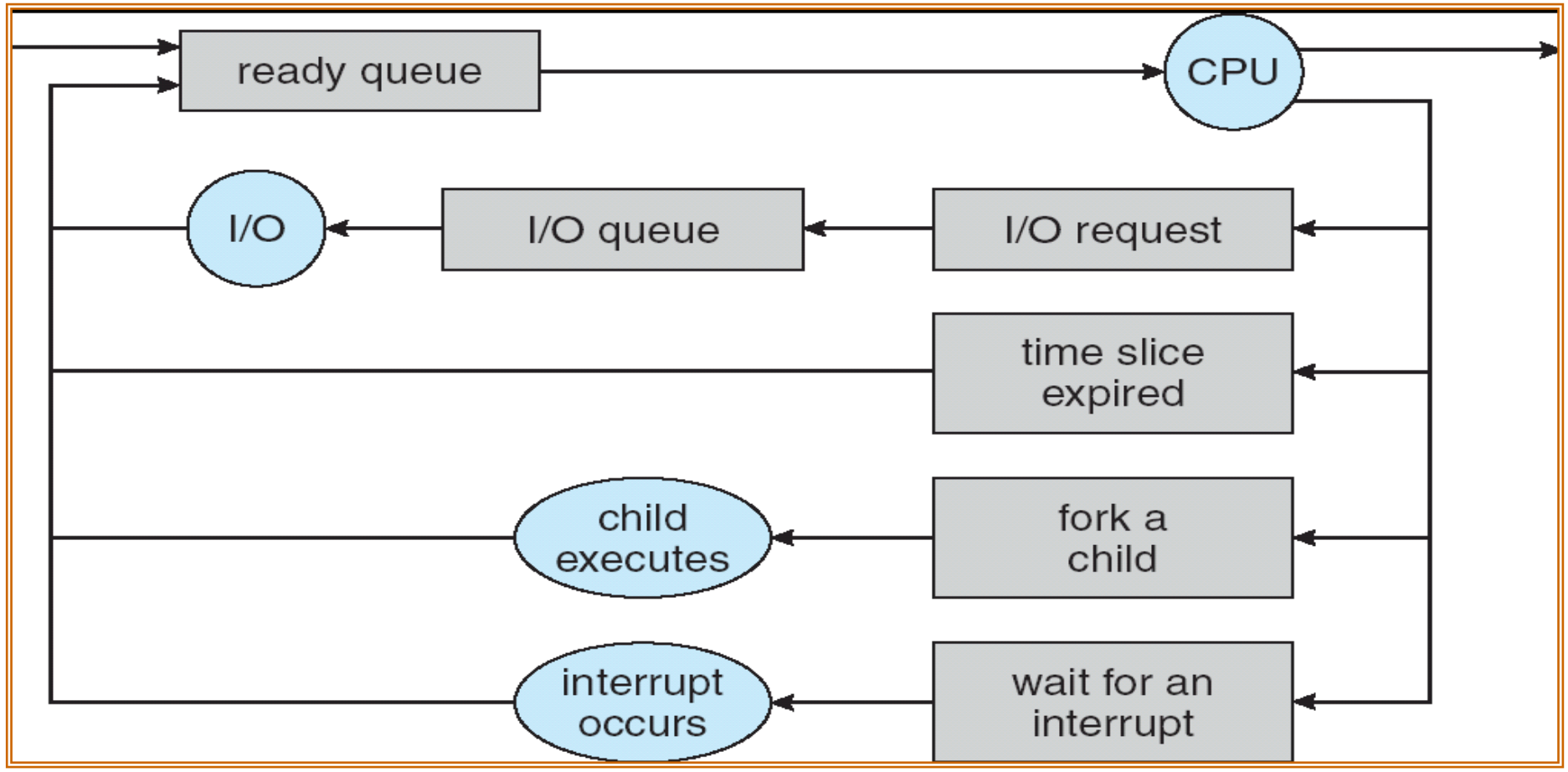
Process Scheduling Queues

- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting for CPU.
- Device queues – set of processes waiting for an I/O device.
- Process migration is possible between these queues.

Ready Queue and I/O Device Queues

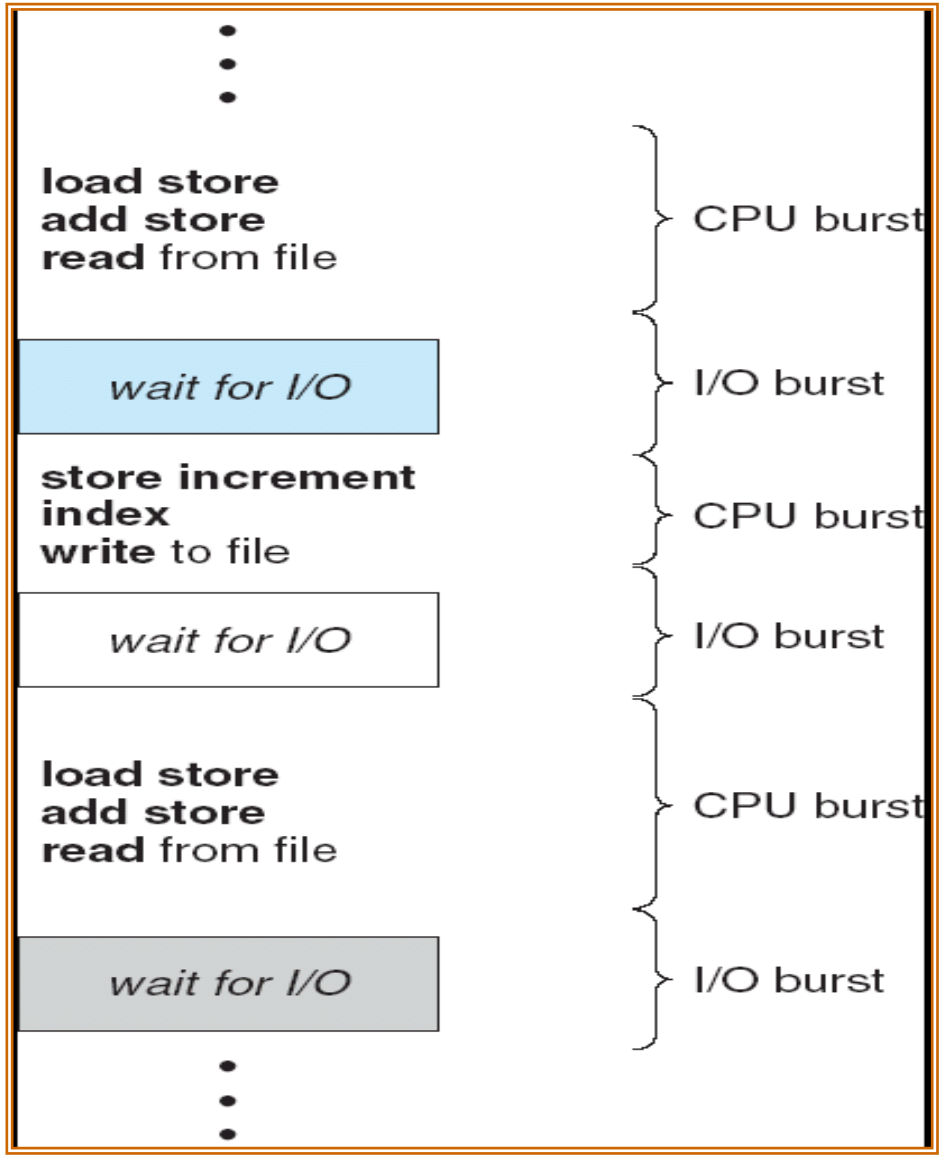


Process Lifecycle

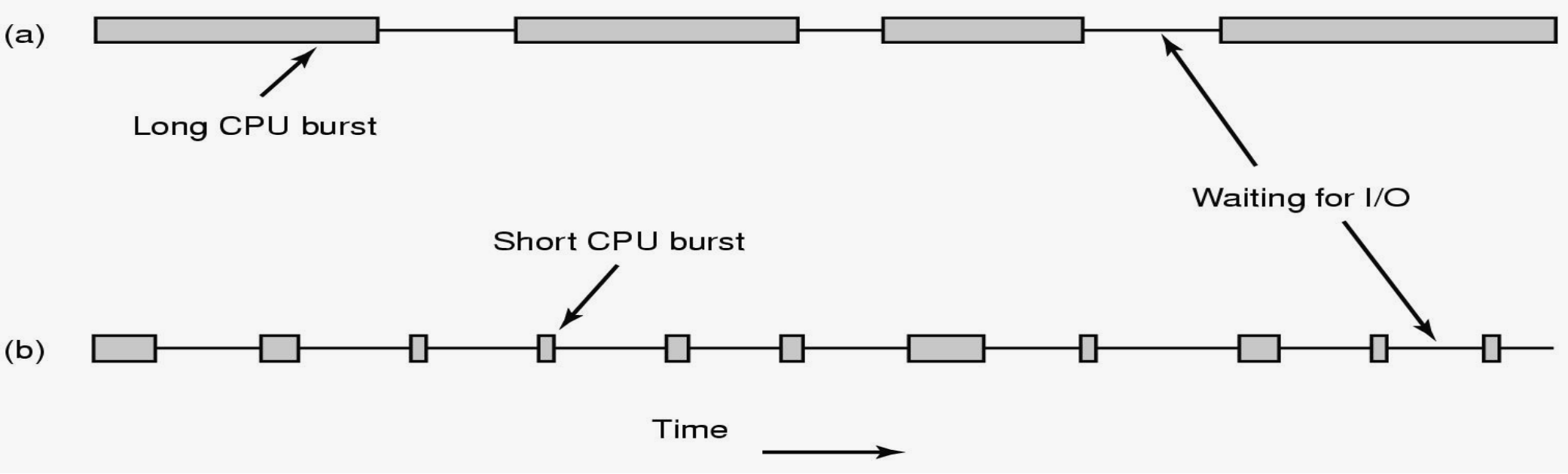


CPU and I/O Bursts

- CPU–I/O Burst Cycle –
  - Process execution consists of a *cycle* of CPU execution and I/O wait.
- I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts.
- *CPU-bound process* – spends more time doing computations; few very long CPU bursts.



CPU-bound and I/O-bound Processes



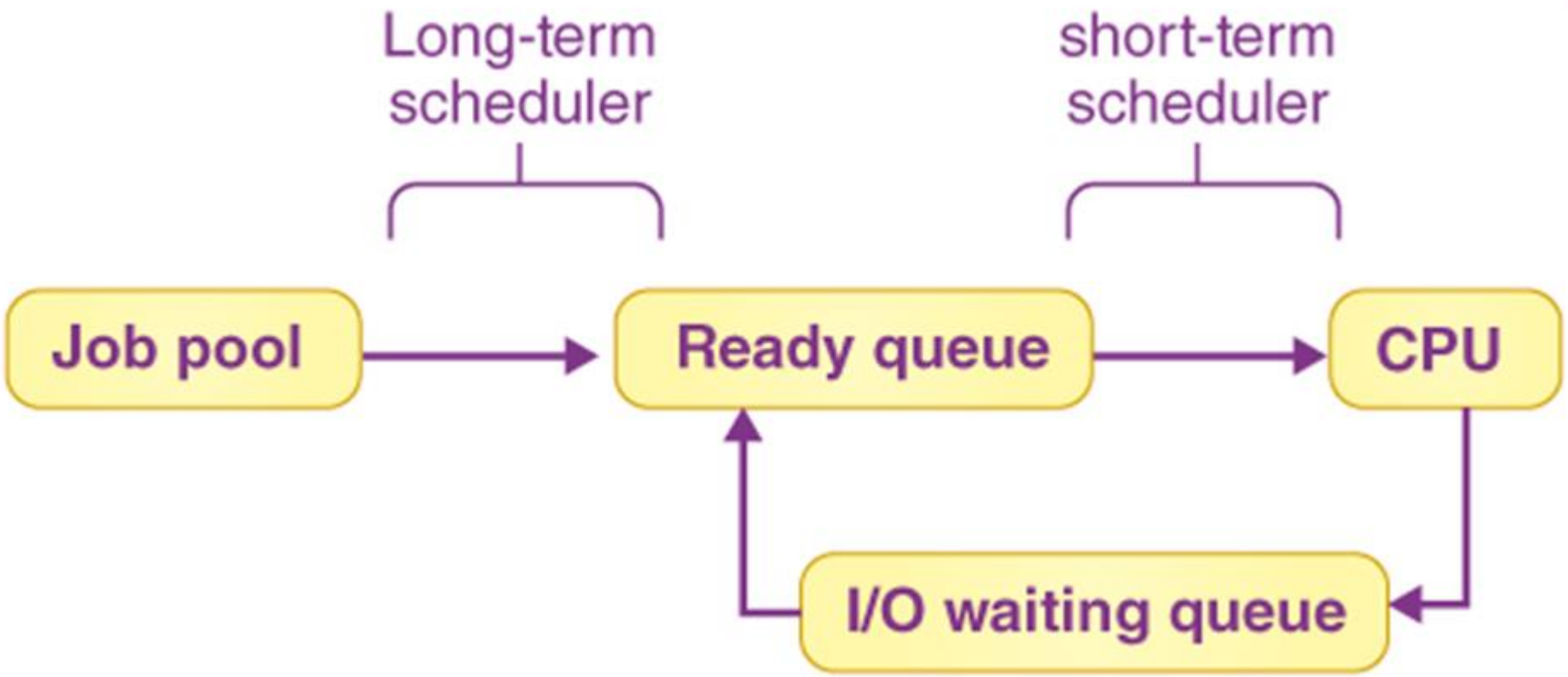
(a) A CPU-bound process

(b) An I/O-bound process

Schedulers

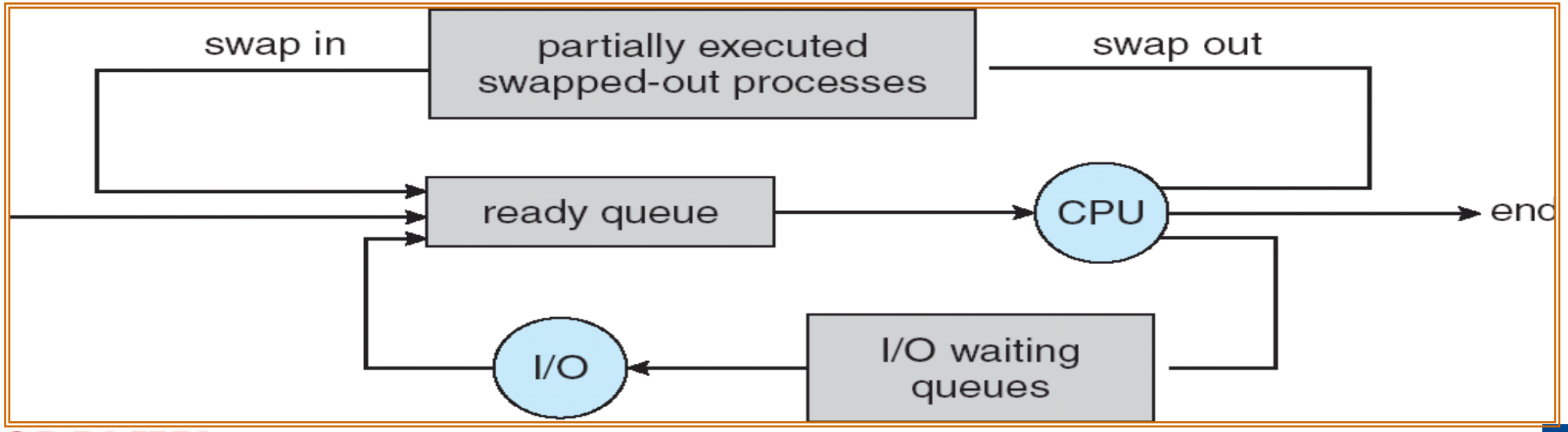
- The processes may be first spooled to a mass-storage system, where they are kept for later execution.
- The *long-term scheduler (or job scheduler)* – selects processes from this pool and loads them into memory for execution.
  - The long term scheduler, if it exists, will control the *degree of multiprogramming*
- The *short-term scheduler (or CPU scheduler)* – selects from among the *ready* processes, and allocates the CPU to one of them.
  - Unlike the long-term scheduler, the short-term scheduler is invoked very frequently.

Short Term Scheduler



Addition of Medium-Term Scheduler

- The medium-term scheduler can reduce the degree of multiprogramming by removing processes from memory.
- At some later time, the process can be re-introduced into memory (*swapping*).



Comparison of Schedulers

Parameters	Long-Term	Short-Term	Medium-Term
Type of Scheduler	It is a type of job scheduler.	It is a type of CPU scheduler.	It is a type of process swapping scheduler.
Speed	Its speed is comparatively less than that of the Short-Term scheduler.	It is the fastest among the other two.	Its speed is in between both Long and Short-Term schedulers.
Minimal time-sharing system	Almost absent	Minimal	Present

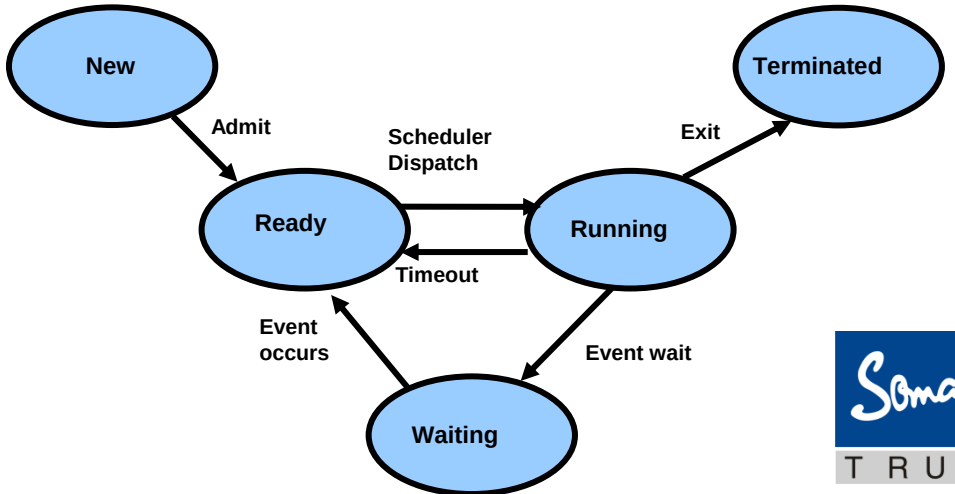
Comparison of Schedulers

Parameters	Long-Term	Short-Term	Medium-Term
Purpose	A Long-Term Scheduler helps in controlling the overall degree of multiprogramming.	The Short-Term Scheduler provides much less control over the degree of multiprogramming.	Medium-Term reduces the overall degree of multiprogramming.
Function	Selects processes from the pool and then loads them into the memory for execution.	Selects all those processes that are ready to be executed.	Can re-introduce the given process into memory. The execution can then be continued.

When to Schedule?

Process state transition model, CPU scheduler could be invoked at five different points:

1. When a process switches from the new state to the ready state.
2. When a process switches from the running state to the waiting state.
3. When a process switches from the running state to the ready state.
4. When a process switches from the waiting state to the ready state.
5. When a process terminates.



Non-preemptive vs. Preemptive Scheduling

- Under non-preemptive scheduling, each running process keeps the CPU until it completes or it switches to the waiting (blocked) state (points 2 and 5 from previous slides).

- Under preemptive scheduling, a running process may be also forced to release the CPU even though it is neither completed nor blocked.
  - In time-sharing systems, when the running process reaches the end of its time quantum (slice)
  - In general, whenever there is a change in the ready queue.

Scheduling Criteria

Several criteria can be used to compare the performance of scheduling algorithms

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been 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 the complete output.
- Fairness - Ensuring that all processes receive an equitable share of CPU time, preventing any single process from monopolizing resources or starving.
- Meeting the deadlines (real-time systems)

Optimization Criteria

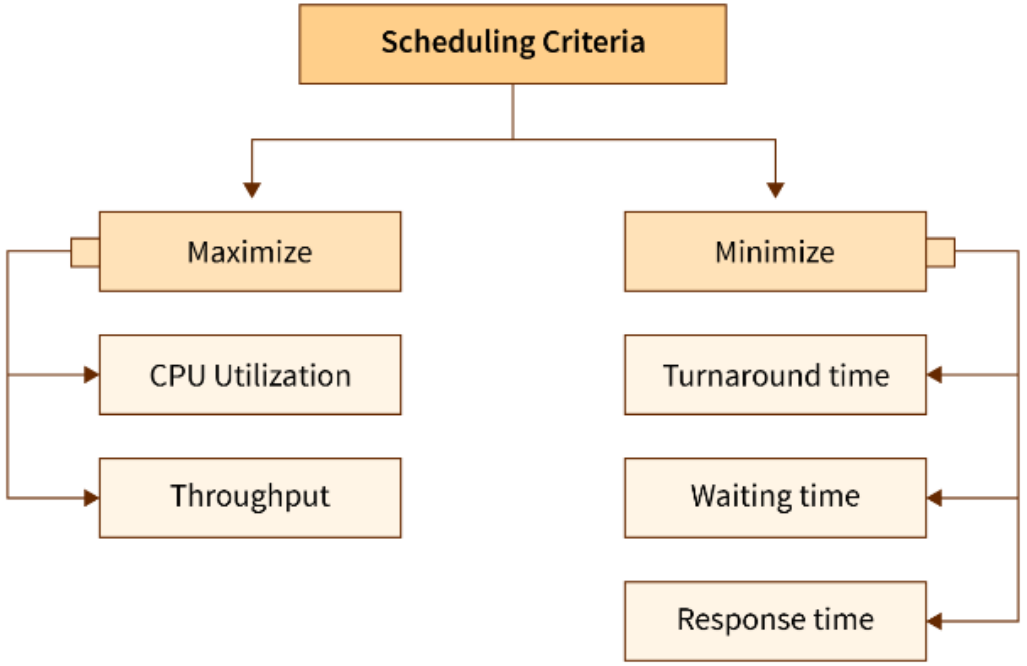


Image source : Scaler

- In the examples, we will assume
  - average waiting time is the performance measure
  - only one CPU burst (in milliseconds) per process

First-Come, First-Served (FCFS) Scheduling

- Single FIFO ready queue
- No-preemptive
  - Not suitable for timesharing systems
- Simple to implement and understand
- Average waiting time dependent on the order processes enter the system

First-Come, First-Served (FCFS) Scheduling

- Consider processes arrive at time 0

Turnaround Time = Completion Time – Arrival Time

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- The *Gantt Chart* for the schedule:

- Turnaround Time P_1 = 24; P_2 = 27; P_3 = 30
- Average turnaround time: (24+27+30)/3 = 27ms

First-Come, First-Served (FCFS) Scheduling

- Consider processes arrive at time 0

Waiting Time = Turnaround Time – Burst Time

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 ,
- The *Gantt Chart* for the schedule:

- Turnaround Time P_1 = 24; P_2 = 27; P_3 = 30
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0+24+27)/3 = 17ms

FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

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

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart for the schedule:

- Turnaround Time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Average Turnaround time: (30+3+6)/3 = 13ms
- *Problems:*
  - *Convoy effect* (short processes behind long processes)
  - Non-preemptive -- not suitable for time-sharing systems

FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order P_2 , P_3 ,

Waiting Time = Turnaround Time – Burst Time

- The Gantt chart for the schedule:

- Turnaround Time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (30+3+6)/3 = 13ms

FCFS Scheduling (Cont.)

- *Problems:*
  - *Convoy effect* (short processes behind long processes)
  - Non-preemptive -- not suitable for time-sharing systems

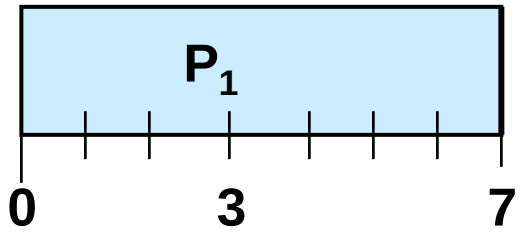
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. The CPU is assigned to the process with the smallest CPU burst (FCFS can be used to break ties).
- Two schemes:
  - nonpreemptive
  - preemptive – Also known as the Shortest-Remaining-Time-First (SRTF).
- Non-preemptive SJF is *optimal* if all the processes are ready simultaneously– gives minimum average waiting time for a given set of processes.

Example for 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

- At time 0, P_1 is the only process, so it gets the CPU and runs to completion

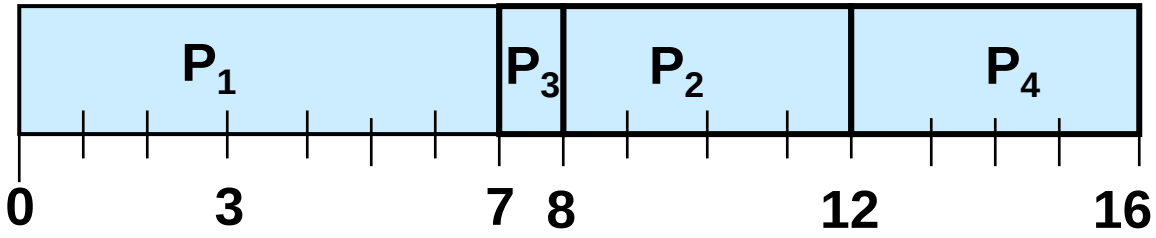


Example for Non-Preemptive SJF

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

Turnaround Time = Completion Time – Arrival Time

- Once P_1 has completed the queue now holds P_2, P_3 and P_4



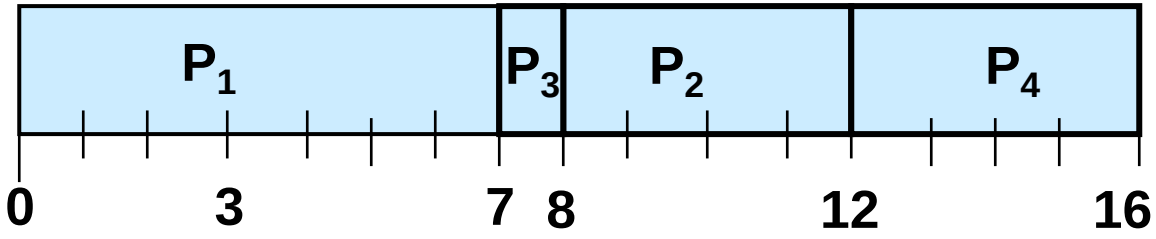
- P_3 gets the CPU first since it is the shortest. P_2 then P_4 get the CPU in turn (based on arrival time)
- Turnaround Time for process p1= 7, p2= 10. p3=4, p4=11
- Average Turnaround time : (7+10+4+11)/4 = 8ms

Example for 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

Waiting Time = Turnaround Time – Burst Time

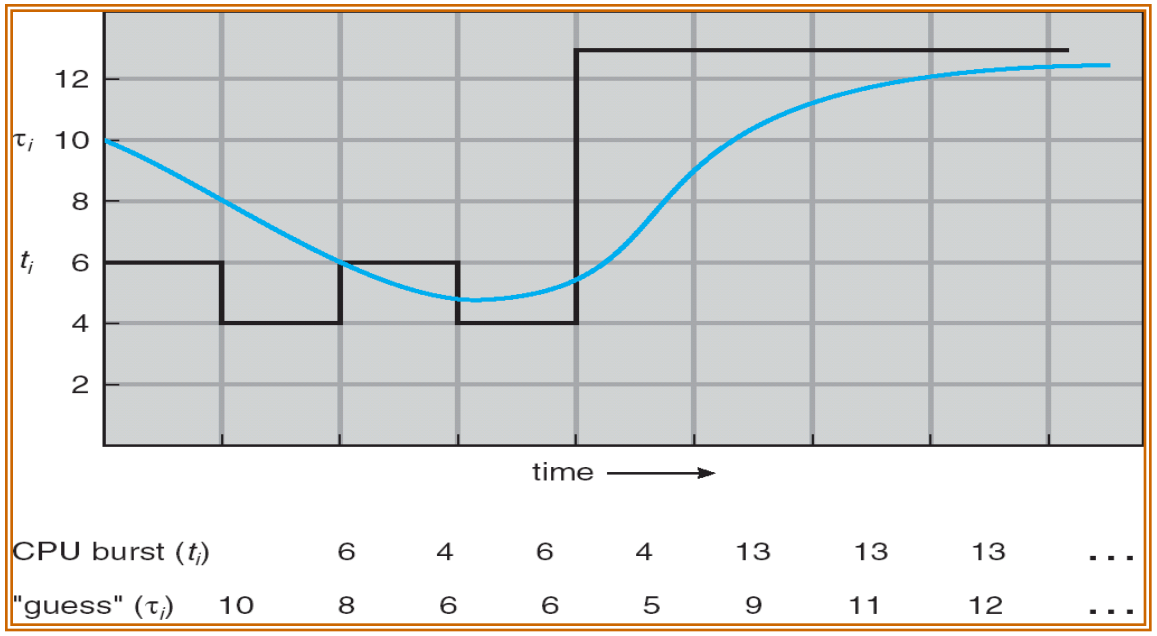
- Once P_1 has completed the queue now holds P_2, P_3 and P_4



- Turnaround Time for process p1= 7, p2= 10. p3=4, p4=11
- Waiting Time for process p1 = 0, p2=6,p3=3,p4=7

Estimating the Length of Next CPU Burst

- Problem with SJF: It is very difficult to know exactly the length of the next CPU burst.
- Idea: Based on the observations in the recent past, we can try to *predict*.
- *Exponential averaging: n*th CPU burst = t_n; the average of all past bursts τ_n, using a weighting factor $0 <= \alpha <= 1$, the next CPU burst is: $\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \tau_n$.



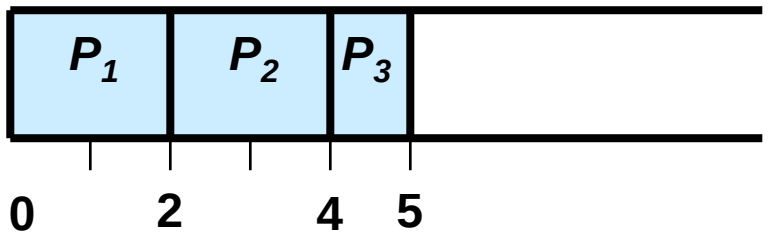
Shortest Remaining Time First (SRTF)

- Shortest Remaining Time First (SRTF) scheduling algorithm is basically a preemptive mode of the Shortest Job First (SJF) algorithm in which jobs are scheduled according to the shortest remaining time.
- In this scheduling technique, the process with the shortest burst time is executed first by the CPU, but the arrival time of all processes need not be the same.
- If another process with the shortest burst time arrives, then the current process will be preempted, and a newer ready job will be executed first.
- Also called as Shortest Remaining Time Next (SRTN)

Example for Preemptive SJF (SRTF)

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

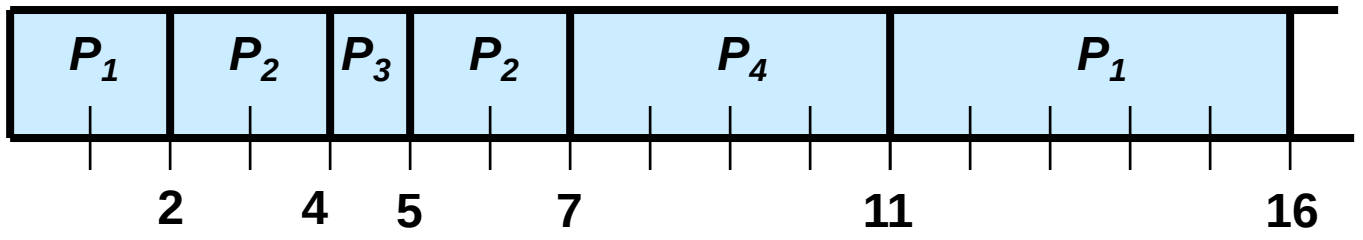
- Time 0 – P_1 gets the CPU Ready = [(P_1,7)]
- Time 2 – P_2 arrives – CPU has P_1 with time=5, Ready = [(P_2,4)] – P_2 gets the CPU
- Time 4 – P_3 arrives – CPU has P_2 with time = 2, Ready = [(P_1,5),(P_3,1)] – P_3 gets the CPU



Example for Preemptive SJF (SRTF)

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

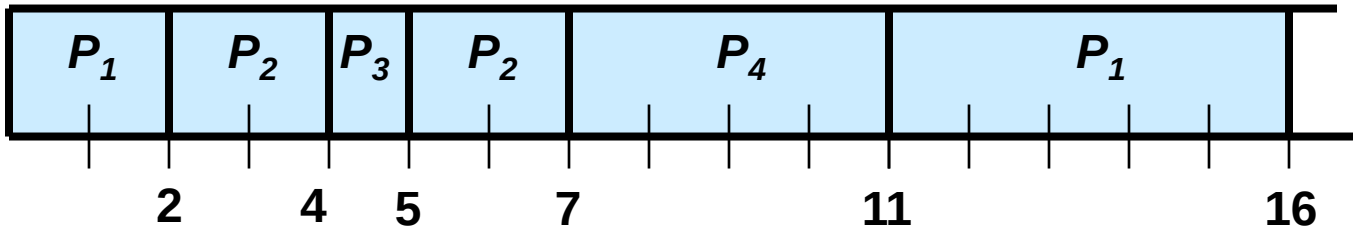
- Time 5 – P_3 completes and P_4 arrives - Ready = [(P_1,5),(P_2,2),(P_4,4)] – P_2 gets the CPU
- Time 7 – P_2 completes – Ready = [(P_1,5),(P_4,4)] – P_4 gets the CPU
- Time 11 – P_4 completes, P_1 gets the CPU



Example for Preemptive SJF (SRTF)

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

Turnaround Time = Completion Time – Arrival Time

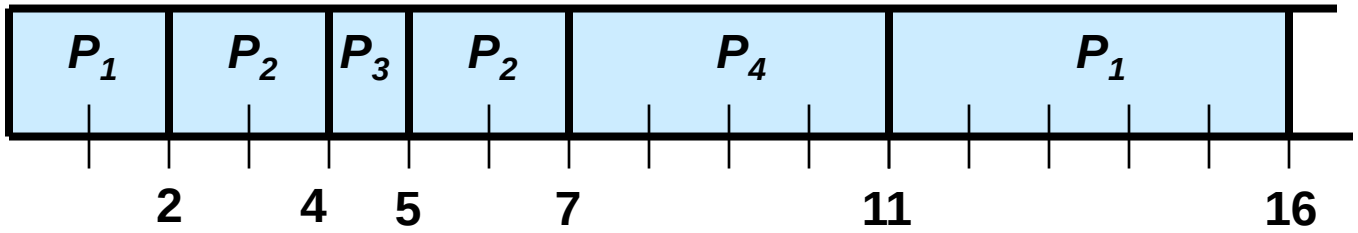


- Turnaround Time p1=16 ,p2=5,p3=1,p4=6
- Average Turnaround time = (16 + 5 + 1 +6)/4 = 7ms

Example for Preemptive SJF (SRTF)

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

Waiting Time = Turnaround Time – Burst Time



- Turnaround Time p1=16 ,p2=5,p3=1,p4=6
- Waiting Time p1=9 ,p2=1,p3=0,p4=2
- Average waiting time time = (9 + 1 + 0 +2)/4 = 3ms

Priority-Based 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).
- When a process arrives at the ready queue,
  - **the priority is compared with priority of the current running process.**
- It can be
  - **pre-emptive**
  - **non pre-emptive**

Priority-Based Scheduling (cont…)

Scenario:
If a newly arrived process has a higher priority than the currently running process.

Characteristics:

- **Preemptive Priority Scheduling Algorithm:**
  - The CPU is preempted, and the currently running process is moved to the ready queue.
  - The newly arrived process is then scheduled for execution.
- **Non-Preemptive Priority Scheduling Algorithm:**
  - The newly arrived process is placed at the tail of the ready queue.
  - The currently running process continues execution until it finishes, after which the scheduler picks the next process.

Priority-Based Scheduling (cont…)

- SJF is a special case of priority scheduling:
  - process priority = the *inverse of remaining CPU time*
  - **The larger the CPU burst, the lower the priority and vice versa**

- **Equal priority processes are scheduled in FCFS order**
  - FCFS can be used to break ties.

Example for Priority-based Scheduling

- Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds.

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

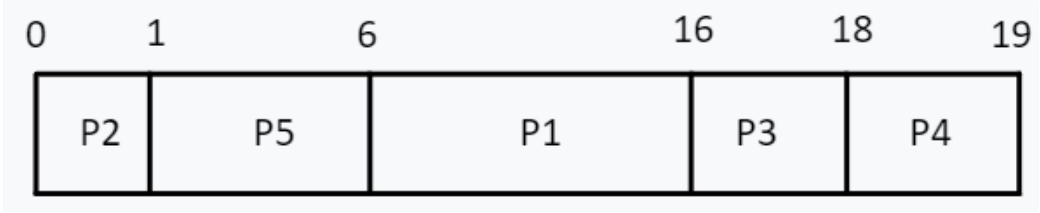
- Low number represents the high priority.

Example for Priority-based Scheduling

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Turnaround Time = Completion Time – Arrival Time

- Gantt Chart



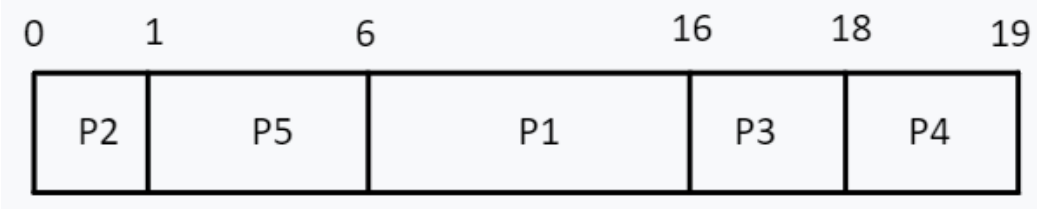
- Turnaround Time p1=16,p2=1,p3=18,p4=19,p5=6
- Average Turnaround Time = (16+1+18+19+6)/5=12ms

Example for Priority-based Scheduling

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Waiting Time = Turnaround Time – Burst Time

- Gantt Chart



- Turnaround Time p1=16,p2=1,p3=18,p4=19,p5=6
- Waiting Time p1=6,p2=0,p3=16,p4=18,p5=1
- Average Turnaround Time = (6+0+16+18+1)/5=8.2ms

Priority-Based Scheduling (Cont.)

- Problem: Indefinite Blocking (or Starvation) –
  - low priority processes may never execute.
- One solution: *Aging* – as time progresses, increase the priority of the processes that wait in the system for a long time.
- Priority Assignment
  - Internal factors: timing constraints, memory requirements, the ratio of average I/O burst to average CPU burst….
  - External factors: Importance of the process, financial considerations, hierarchy among users…

Round Robin Scheduling

- It is similar to FCFS scheduling, but preemption is added to enable the system 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 in length.
- Every process is assigned a time quantum for its execution, allowing it to execute only for that time ***quantum.***
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1-time quantum.

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (*time quantum*). After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Newly-arriving processes (and processes that complete their I/O bursts) are added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q, then no process waits more than $(n-1)q$ time units.

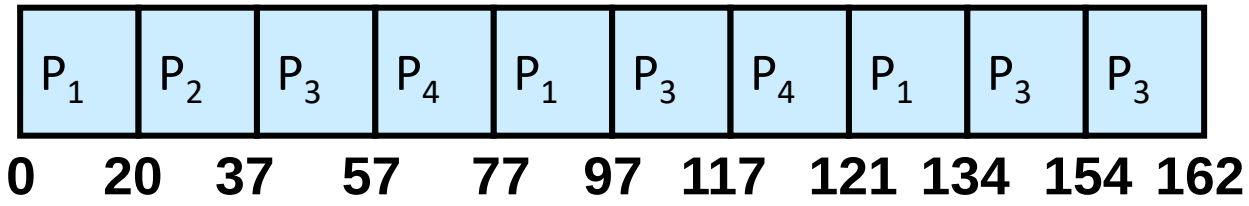
Example for Round-Robin

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Time Quantum given as 20

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart:



- Turn around time = 134+37+162+121=454/4=113.5

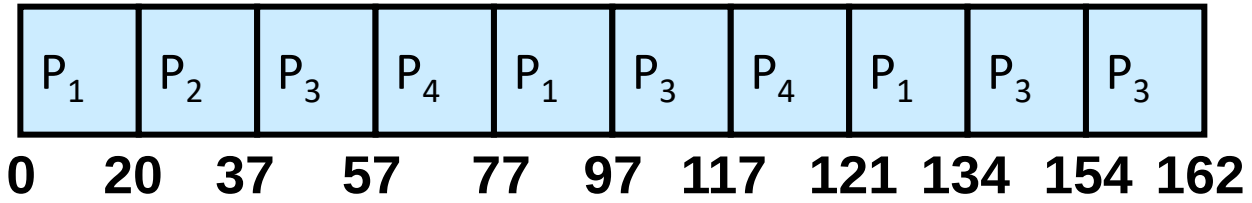
Example for Round-Robin

Process	Burst Time	Turnaround Time
P_1	53	134
P_2	17	37
P_3	68	162
P_4	24	121

- Time Quantum given as 20

Waiting Time = Turnaround Time – Burst Time

- The Gantt chart:



- Average wait time = (81+20+94+97)/4 = 73
- Typically, higher average turnaround time (amount of time to execute a particular process) than SJF, but better *response time* (amount of time it takes from when a request was submitted until the first response is produced).

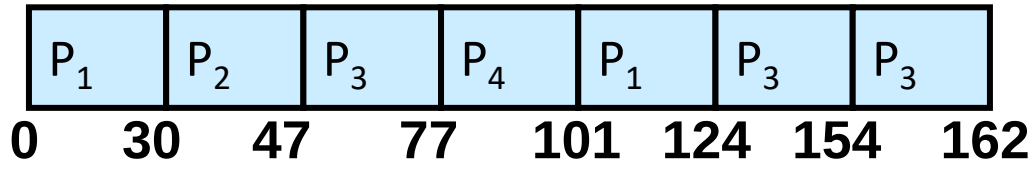
Example for Round-Robin

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Time Quantum = 30

Turnaround Time = Completion Time – Arrival Time

- The Gantt chart

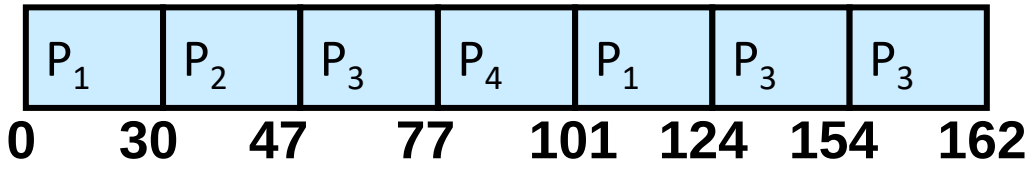


- Turn around Time = (124+47+162+101)=434/4 = 108.5

Example for Round-Robin

Process	Burst Time	Turnaround Time
P_1	53	124
P_2	17	47
P_3	68	162
P_4	24	101

The Gantt chart: (Time Quantum = 30)

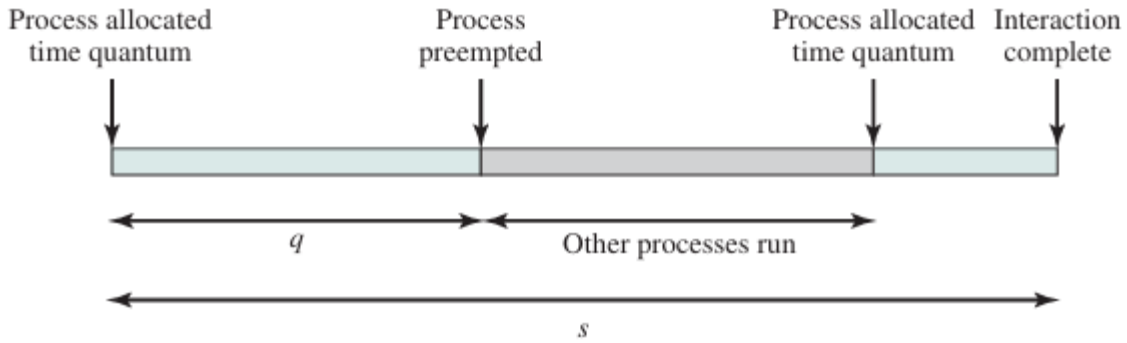


Waiting Time = Turnaround Time – Burst Time

- Average wait time = (71+30+94+77)/4 = 68

Effect of time quanta

- With round robin, the principal design issue is the length of the time quantum, or slice, to be used.

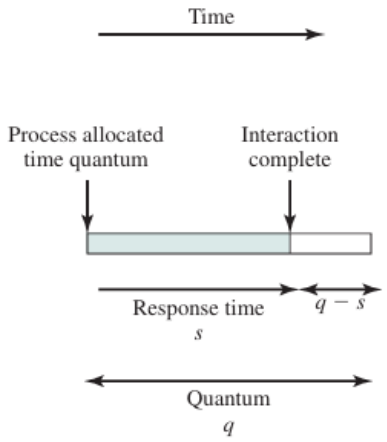


(b) Time quantum less than typical interaction

- If the quantum is very short, then short processes will move through the system relatively quickly.
- On the other hand, there is processing **over head** involved in handling the clock interrupt and performing the scheduling and dispatching function.
- Thus, very **short time quanta should be avoided**.

Effect of time quanta

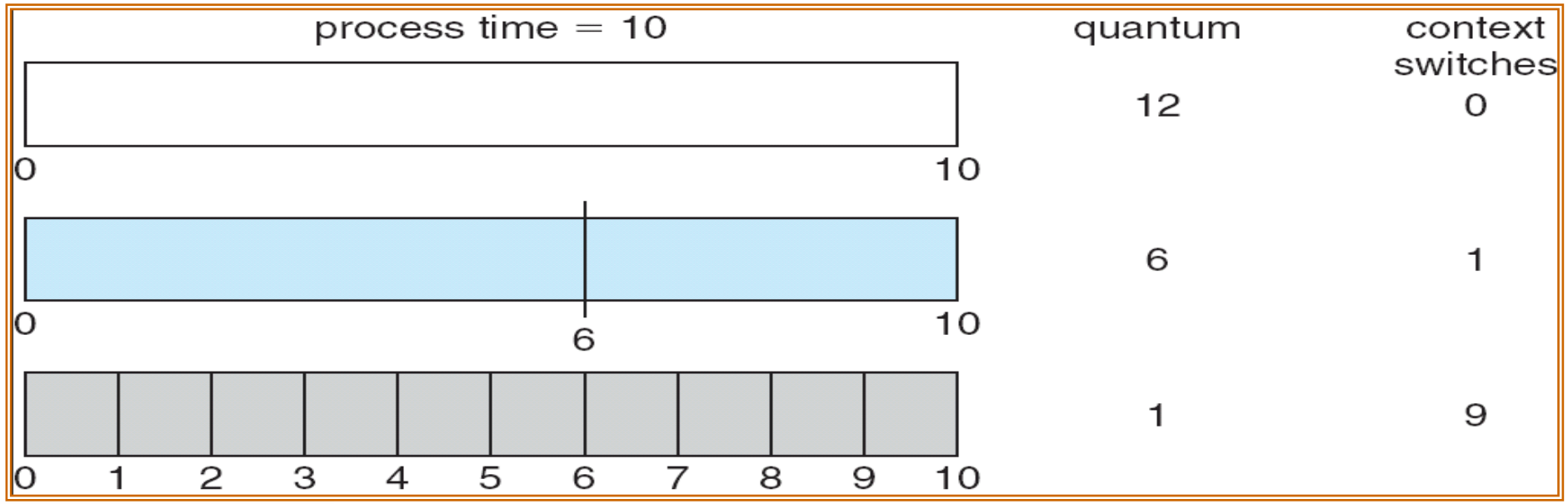
- When the time quantum is greater than the typical interaction time, it means that each process gets more time to execute before being switched out.
- This can lead to fewer context switches, which might improve efficiency for CPU-bound processes but could also increase response time for interactive processes.



(a) Time quantum greater than typical interaction

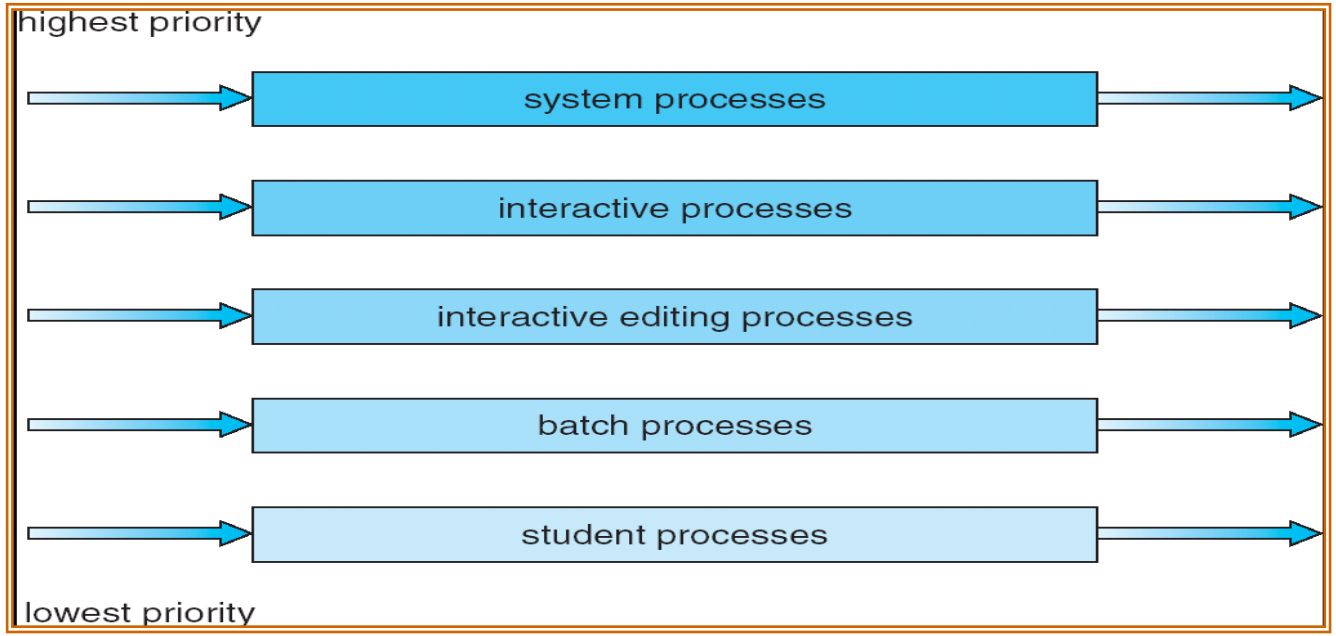
Choosing a Time Quantum

- The effect of quantum size on context-switching time must be carefully considered.
- The time quantum must be large with respect to the context-switch time
- Modern systems use quanta from 10 to 100 msec with context switch taking < 10 msec

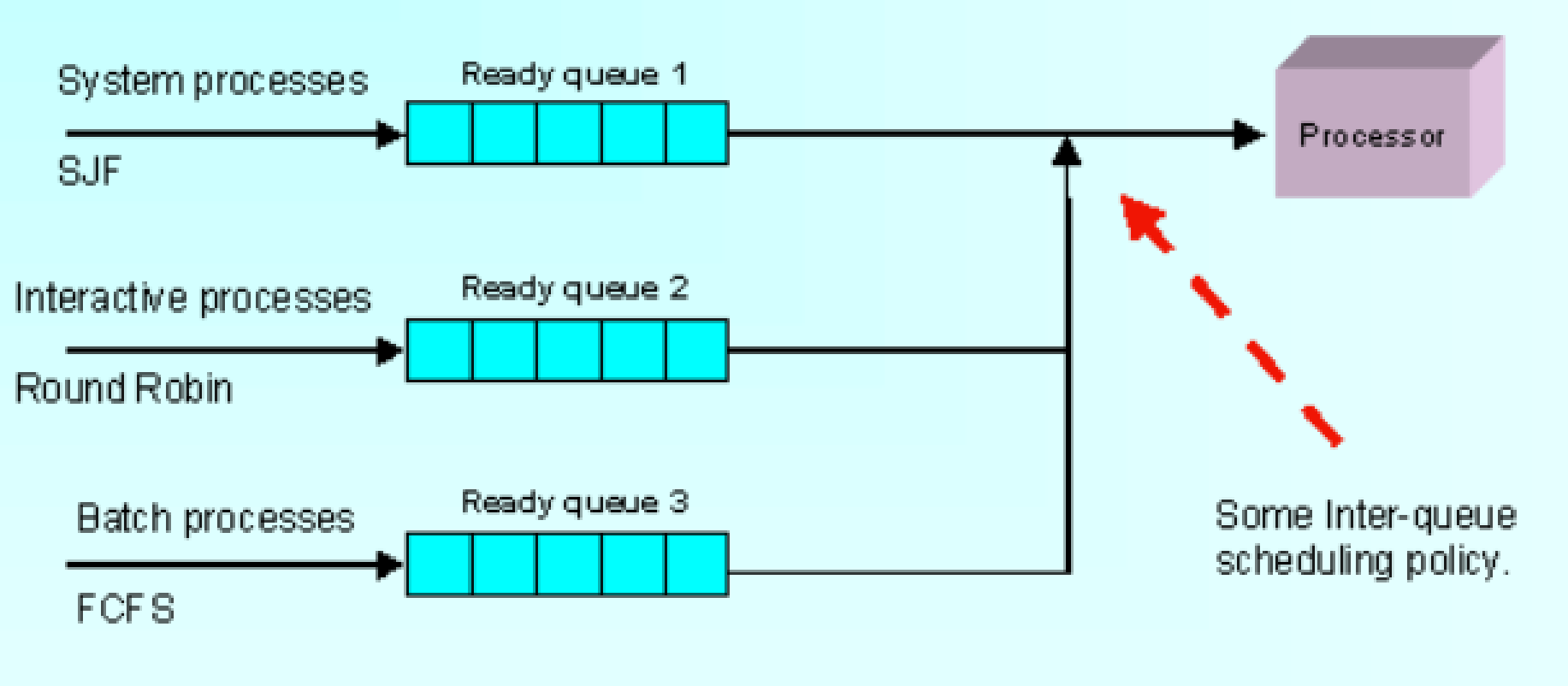


Multilevel Queue

- Ready queue is partitioned into separate queues:
  - Foreground (interactive)
  - Background (batch) processes;
- Each queue has its own scheduling policy



Multilevel Queue Scheduling

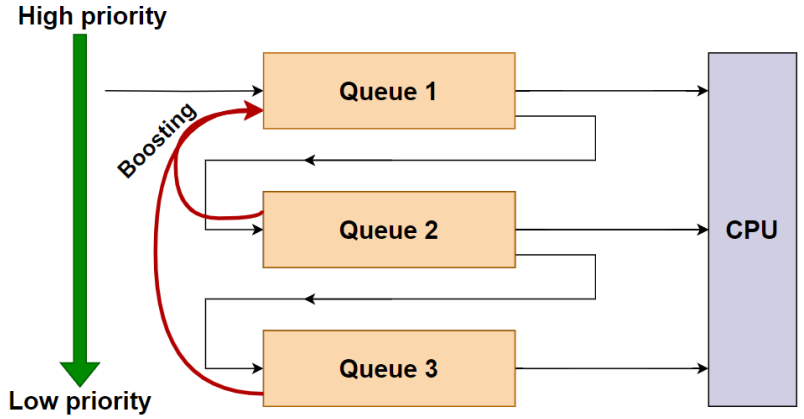


Multilevel Queue Scheduling

- Each queue may have has its own scheduling algorithm: Round Robin, FCFS, SJF…
- In addition, (meta-)scheduling must be done between the queues.
  - Fixed priority scheduling (i.e. serve first the queue with highest priority). Problems?
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; for example, 50% of CPU time is used by the highest priority queue, 20% of CPU time to the second queue, and so on..
  - Also, need to specify which queue a process will be put to when it arrives to the system and/or when it starts a new CPU burst.

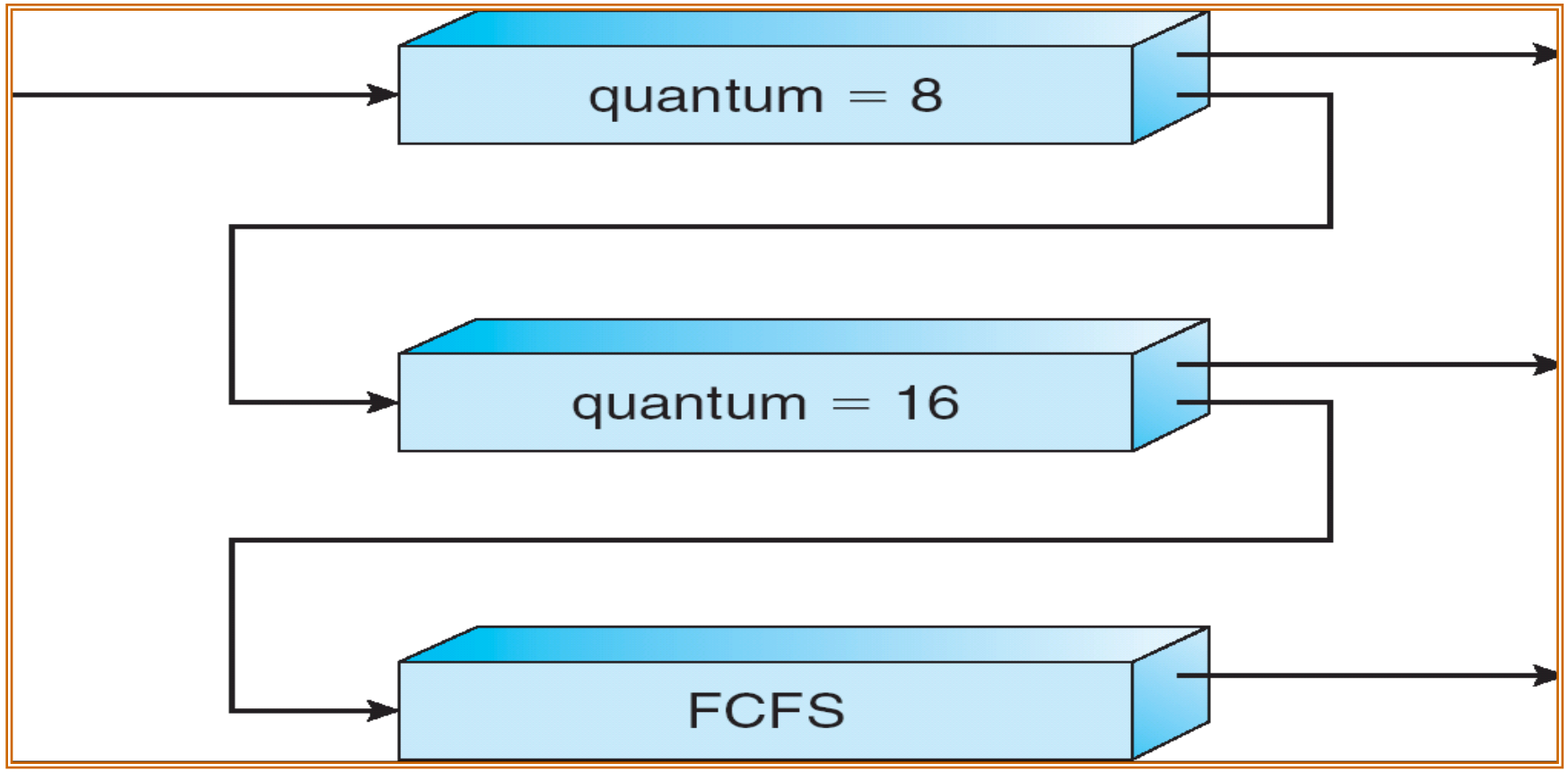
Multilevel Feedback Queue (MLFQ)

- In a multi-level queue-scheduling algorithm, processes are permanently assigned to a queue.
- Idea: Allow processes to move among various queues.



- Uses multiple queues with varying priorities to manage process execution.
- It dynamically adjusts priorities based on process behavior, promoting or demoting processes between queues.

Multilevel Feedback Queue



Multilevel Feedback Queue

- Multilevel feedback queue scheduler is 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
- The scheduler can be configured to match the requirements of a specific system.

Characteristics of Various Scheduling Policies

Scheduling Policy	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
First-Come, First-Served (FCFS)	Arrival time	Non-preemptive	Low for long processes	High for long processes	Minimal	Simple, long wait times	No, but convoy effect
Shortest Job First (SJF)	Shortest next CPU burst	Non-preemptive	High	Low for short processes	Requires knowledge of process length	Efficient for short processes	Yes, long processes may starve
Shortest Remaining Time First (SRTF)	Shortest remaining CPU burst	Preemptive	High	Low for short processes	High, frequent context switching	Efficient for short processes	Yes, long processes may starve

Characteristics of Various Scheduling Policies

Scheduling Policy	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
Priority Scheduling	Highest priority	Preemptive or Non-preemptive	Varies based on priority	Low for high-priority processes	Requires priority assignment	High-priority processes favored	Yes, low-priority processes may starve
Round Robin (RR)	Time quantum (fixed time slice)	Preemptive	Moderate	Moderate	High, frequent context switching	Fair time-sharing	No, each process gets CPU time

Characteristics of Various Scheduling Policies

Scheduling Policy	Selection Function	Decision Mode	Throughput	Response Time	Overhead	Effect on Processes	Starvation
Multilevel Queue	Based on queue priority	Preemptive or Non-preemptive	Varies based on queue configuration	Varies	High, managing multiple queues	Processes categorized into different queues	Yes, lower-priority queues may starve
Multilevel Feedback Queue	Based on aging and feedback	Preemptive	High	Low for interactive processes	Very high, complex management	Dynamic adjustment of process priority	Reduced, processes can move between queues

Question ?

Process Concept and scheduling

Ms. Swati Mali

swatimali@gmail.com
K. J. Somaiya College of Engineering
Somaiya Vidyavihar University

Process Concept and scheduling	
2..1	Process: Concept of a Process, Process States, Process Description, Process Control Block, Operations on Processes. Threads: Definition and Types, Concept of Multithreading
2.2	Multicore processors and threads. Scheduling: Uniprocessor Scheduling - Types of Scheduling: Preemptive and, Non-preemptive, Scheduling Algorithms: FCFS, SJF, SRTN, Priority based, Round Robin, Multilevel Queue scheduling.
2.3	Introduction to Thread Scheduling
2.4	Linux Scheduling.

Basic Terminologies & Definitions

Task and Program

- **Task:**
  - Task is a unit of assigned work.
  - Can also be defined as the unit of programming controlled by OS.
  - Depending on the OS design the task may involve one or more processes.
  - Example: Bake a cake
- **Program**:
  - A program is defined as sequence of instruction written to accomplish a task.
  - A program may comprise of one or more processes depending on the statement being executed.
  - Generally referred as a passive entity that does not perform any action.
  - Example: A particular recipe given in book.

Process

- This is an instance of program in execution.
- The static statements in the program when executed, take the process form.
- In contrast to the program, a process is an active entity which needs a set of resources to perform its function.
- The Linux kernel internally represents processes as tasks.
- A process elements are: a program, data and process state.
- Example: actually following the steps given the book.

Thread

- A thread is a lightweight process.
- Also defined as the smallest processing unit that is scheduled by an operating system.
- A thread must be part of a process as it shares the process environment viz, code, data and resources with other threads.
- Threading has increased the computing efficiency to significant extent.

Job

- **Job**:
  - A job is unit of work submitted by user to the system.
  - A job may be interactive or a batch job which may in turn consist of one or more processes.

Process States

- **Process State:**
  - A process state is a process' internal data maintained by OS for the purpose of supervision and control of the process.
  - Also called as executional context of the process.
- Process States: (more states, trasitions and reasons for transitions will be discussed later..)
  - New
  - Ready
  - Running
  - Swapped
  - Waiting/blocked
  - Suspended, etc

Process Context

- **Process Context:**
  - Whenever a running process is taken away from processor, some of the process state's information needs to be retained.
  - This information called as process context helps the process to resume from the point where it was stopped last time.
- The context of a process includes:
  - its address space,
  - stack space,
  - virtual address space,
  - register set image i.e. Program Counter, Stack Pointer, Program Status Word, Instruction Register and other general processor registers,
  - accounting information,
  - associated kernel data structures and
  - current state of the process (waiting, ready, etc).

Process Modes

- The operational or the privilege mode of process execution is called process mode.
- A process executes in user mode or a kernel mode.
- Processor switches the process in between these modes depending on the code the process is running.

Event

- An activity that is happens or is expected to happen.
- Generally this a software message exchanged when the activity occurs.
- In operating systems, the events may cause the processes to change their state.
  - e.g. mouse click, file lock reset, etc.

- Check- event viewer in windows OS

Process concepts

- **Process priority:**
  - In a multiprogramming system, the processes are assigned numerical privileged values indicating their relative importance and/or urgency and/or value.
- **Preemption:**
  - Preemption is the ability of the system to take over a currently executing process by another one (possibly with high privileged one) with an intention to resume the preempted process later on.
- **Preemptive:**
  - Preemptive entities are the one those can be taken over by another similar type of entities.
- **Non-preemptive:**
  - Non-preemptive entities are the one those cannot be taken over by other entities

Degree Of Multiprogramming

- With multiprogramming, the CPU can run multiple programs simultaneously or concurrently.
- The degree of multiprogramming is the maximum number of allowed processes at a time in a system that does not let the CPU performance degrade than a certain threshold.

Process

Basic elements of a process

- Process ID: This is a unique identifier assigned to the process
- Code: This is program code.
- Data : These are the data and files required for execution
- Resources : These are different types of resources allocated by OS
- Stack: This contains parameters for the functions/procedures called and their return addresses.
- Process state: This is one of the eleven states process is in.

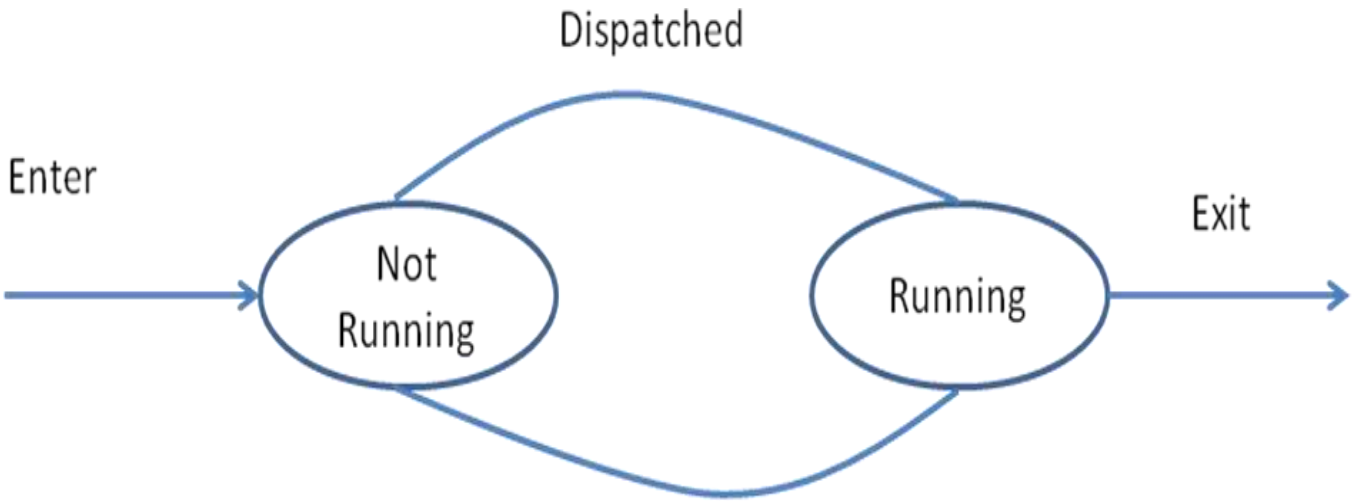
Process state transition

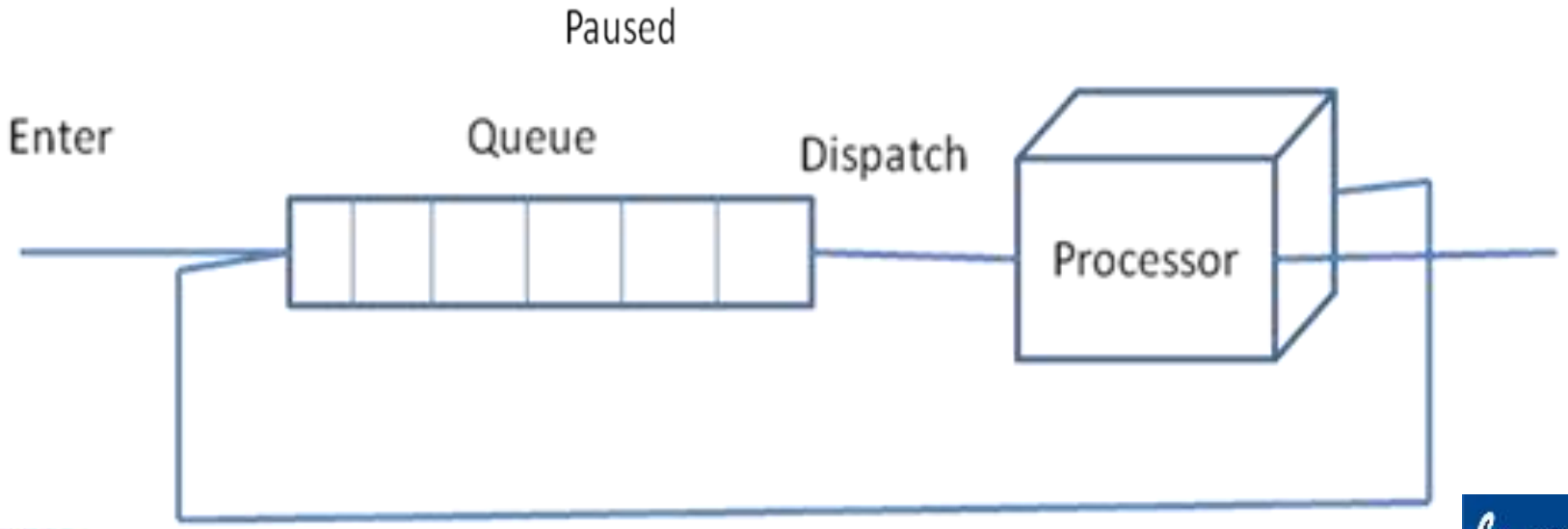
- A state transition for process is defined as a change in its state.
- The state transition occurs in response to some event in the computing system

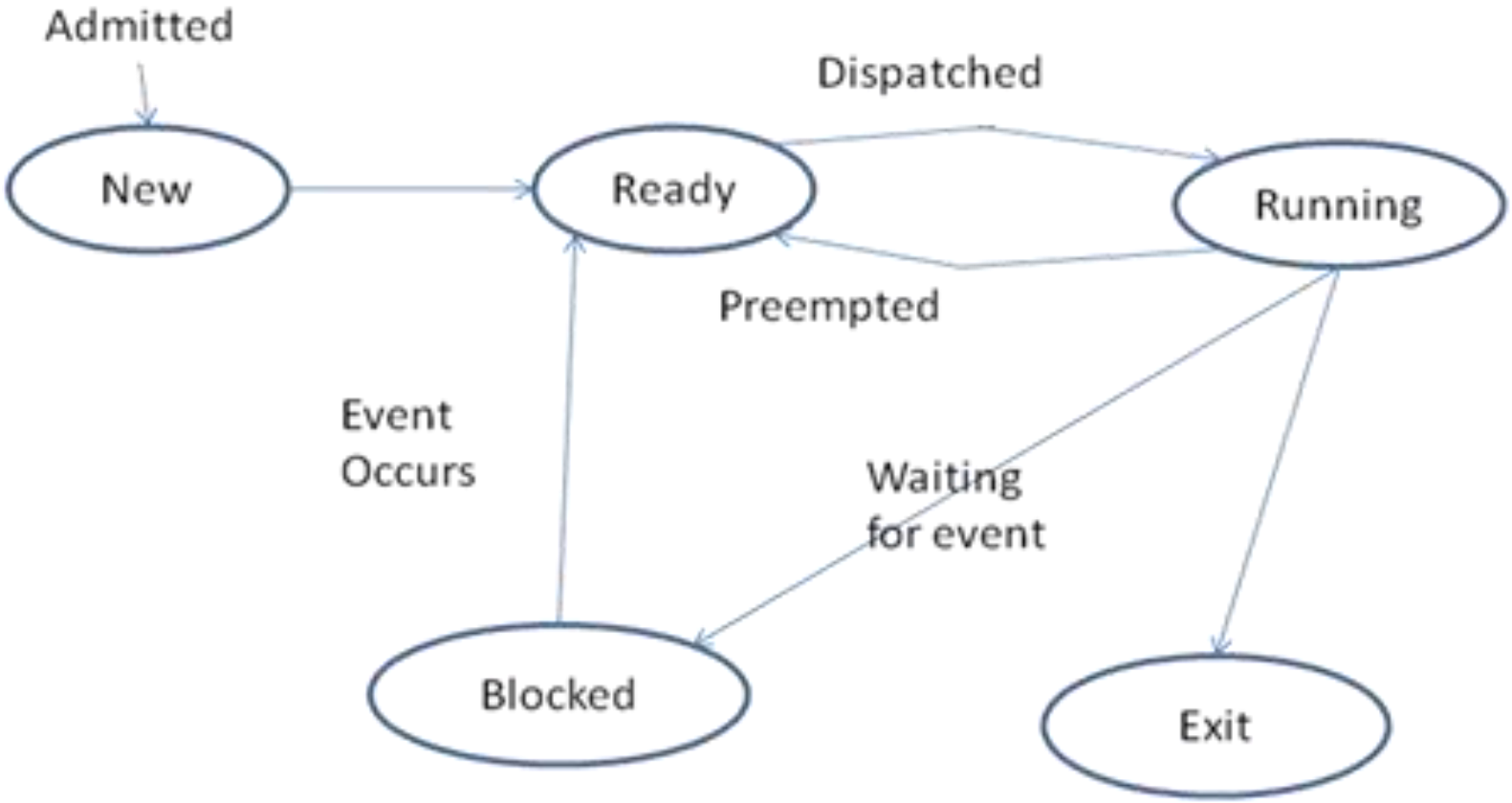
States in process state-transition diagram

- New: The process is being created
- Ready: the process is ready with all required resources and waiting to be assigned to a processor
- Waiting/Blocked: the process is waiting for some event to occur
- Suspended: the process was waiting for some event to occur, but then is swapped to secondary memory to make room for new processes getting ‘blocked’.
- Terminated: the process has finished its execution

Two State Model






Admitted
Dispatched
New
Ready
Running
Preempted
Event
Occurs
Waiting
for event
Blocked
Exit

State transition	Description
Null New	A new process is created to execute a program.
New ready	The OS may move a process from New to Ready state depending on the predefined maximum number of processes allowed (Degree of multiprogramming).
Ready Running	The process is scheduled by dispatcher. The CPU starts or resumes execution of the instruction codes
Blocked Ready	The request initiated by process is satisfied or the event on which it is waiting occurs.
Running Ready	The process is preempted by the OS decides to execute some other process. This transition takes place may be because of expiration of time quantum or arrival of high priority process.
Running Blocked	The running process makes request for resource(s) or needs some event to occur to proceed further. The process then calls for a system call to indicate its wish to wait till the resource or the event becomes available.
Running Termination	The program execution is completed or terminated.

Causes of process initiation

- **New batch job:** while processing the new batch of jobs submitted, the OS creates a process to execute the same.
- **Interactive logon:** when a user logs into the system, a new process is created.
- **Created by OS to provide some service:** The OS initiates a process to perform the service requested by user directly or indirectly, without making the user to wait.
- **Spawned by an existing process**: To support Modularity and/or parallelism, a user program can create some number of new processes.

How the OS creates a process?

1. Create a process
2. Assign a unique process ID to newly created process
3. Allocate the memory and create its process image
4. Initialize process control block
5. Set the appropriate linkages to the different data structures such as ready queue etc.
6. Create or expand the other data structures if required

```
algorithm fork
input:   none
output: to parent process, child PID number
        to child process, 0
{
      check for available kernel resources;
      get free proc table slot, unique PID number;
      check that user not running too many processes;
      mark child state "being created;"
      copy data from parent proc table slot to new child slot,
      increment counts on current directory inode and changed root (if applicable);
      increment open file counts in file table;
      make copy of parent context (u area, text, data, stack) in memory;
      push dummy system level context layer onto child system level context;
                  dummy context contains data allowing child process
                  to recognize itself, and start running from here
                  when scheduled;
      if (executing process is parent process)
      {
            change child state to "ready to run;"
            return(child ID);        /* from system to user */
      }
      else       /* executing process is the child process */
      {
            initialize u area timing fields;
            return(0);        /* to user */
      }
}
```

Causes of process blocking

- Process requests an I/O operation
- Process requests memory or some other resource
- Process wishes to wait for a specific interval of time
- Process waits for message from some other process
- Process wishes to wait for some action to be performed by another process.

Causes of process termination

- **Normal Completion:** The process executes an OS system call to intimate that it has completed its execution.
- **Self termination** (e.g. incorrect file access privileges, inconsistent data)
- **Termination by the parent process:** a parent process calls a system call to kill/terminate its child process when the execution of child process is no longer necessary.
- **Exceeding resource utilization:** An OS may terminate a process if it is holding resources more than it is allowed to. This step can also be taken as part of deadlock recovery procedure.
- **Abnormal conditions during execution:** the OS may terminate a process if an abnormal condition occurs during the program execution. (e.g. memory protection violation, arithmetic overflow etc)
- **Deadlock detection and recovery**

Example Events

- Process creation event
- Process termination event
- Timer event: occurrence of timer interrupt
- Resource request event: a resource request is made by a process
- Resource release event: process releases a resource and notifies.
- I/O initiation request event: a process wishes to initiate I/O operation
- I/O completion event: a process finishes I/O operation
- Message send event: A message is sent by one process to an another one.
- Message receive event: a process receives a massage by another one.
- Signal send event: a signal is sent by a process to an another one
- Signal receive event: a process receives a signal
- A program interrupt: An instruction in currently executing process executes some illegal operation and malfunctions.

A process image

- The collection of program, stack, data and process attributes are called process image.
- The process image is generally maintained as a continuous or contiguous block of memory which resides in secondary memory.
- To execute a process, its process image is loaded in main memory or virtual memory.

Typical Elements of Process Image

Process image element	Description
User Stack	Part of user space that contains program data, a user stack area, and modifiable/editable programs.
User Program	The program under execution.
Stack	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls.
Process Control Block	Process control block(PCB) contains the data which is used by OS to control and manage the process.

Process Control Block

- A PCB contains all information pertaining to a process that is used in controlling the process operation, resource information and information needed for inter-process communication.
- PCB components:
  - Identifiers
  - Processor State Information
  - Process Control Information

PCB components: Identifiers

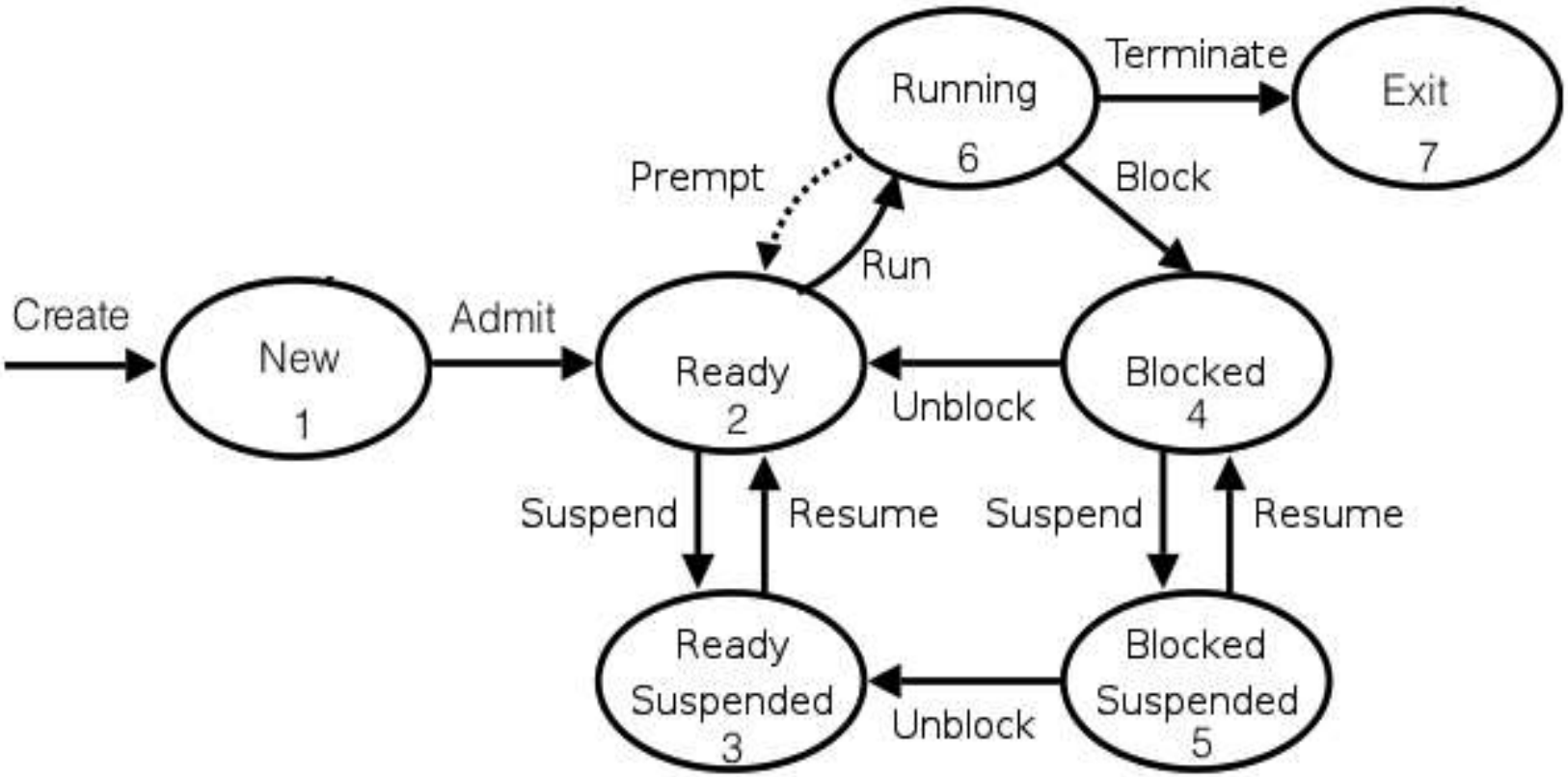
PCB Field	Contents
Identifiers	
Process ID	This is unique process ID assigned to it at the time of creation
Child and Parent IDs	The child and parent process IDs are required for process synchronization
User Identifier	The unique identifier associated with the user in multiuser systems.

Process Control Information	
Scheduling and state information	This is information about process state, priority, scheduling related information and event information in case the concerned process is waiting on that!
Data Structuring	A process may linked to other processes in queue, ring or some other structure. This information is stored in data structuring field.
Interprocess communication	The processes need various flags, messages and signals be exchanged to interact with each other. Some or all this information in stored in PCB.
Process Priority	The priority is a numeric value, which may be assigned to a process at the time of its creation. Some priorities can be altered by user and some change with process age, too.
Memory Management	This field holds the pointer to segments or to the page tables which describe portions of memory assigned to the process.
Resource ownership and Utilization	All the resources or the number of instances of resources assigned to process are described in this field.

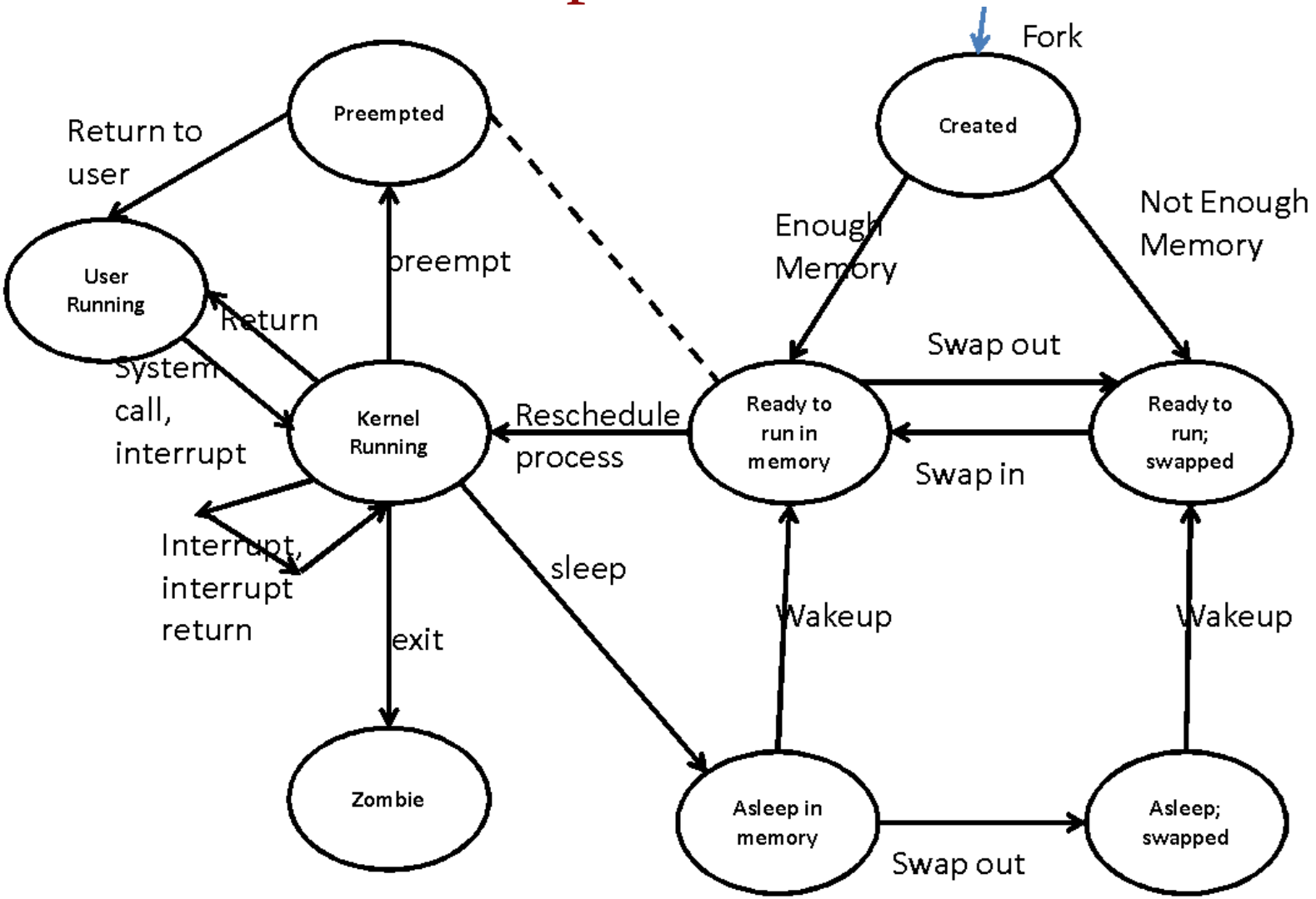
PCB components: Process State Information

Processor State Information	
User Accessible Registers	Every processor offers some general purpose registers those are accessible to user. The number of available registers differs with every processor type.
Control and Status registers	This field is also called PSW(Program Status Word) which typically contains Program counter (Contains the address of the next instruction to be fetched), Condition codes (Result of the most recent arithmetic or logical operation), Status information(Includes interrupt enabled/disabled flags, execution mode)
Stack Pointers	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls. The stack pointer points to stack top.

7- state process model



9-state process model



Process State	Description
User Running	The process is running in user mode
Kernel Running	The process is running in Kernel mode
Ready to Run, in Memory	The process is ready to run as soon as the kernel dispatches it.
Asleep in Memory	The process is blocked on some event, process is in main memory.
Ready to Run, Swapped	The process is ready to run, but the swapping module must swap it in the main memory, so that kernel can schedule it.
Sleeping, swapped	The process is blocked on some event and it is in secondary memory.
Preempted	The process was returning from kernel mode to user mode, but the kernel preempts it and performs a process switch to dispatch another process.
Created	The process is just created and is not yet ready to run. (may not have all the resources, including memory, which are required to run)
Zombie	The process no longer exists, but it leaves some information (probably accounting information) to its parent process to collect.

Reasons of process suspension

- **Swapping:** The main memory may not be enough to accommodate some ready process. So a currently not running process is shifted from main memory to secondary memory.
- **Interactive User request:** a user may wish to suspend a currently running process for debugging or to manage the use of resources
- **Timing:** A process may be executed on periodic basis and may be suspended while waiting for its next turn of execution.
- **Parent process request:** A parent process may wish to suspend its child process to examine or modify the suspended process or to coordinate the child processes.
- **Other OS reasons of suspension:** The OS may suspend a background or utility process or a process that is causing some problem in normal activities.

The process switching operation

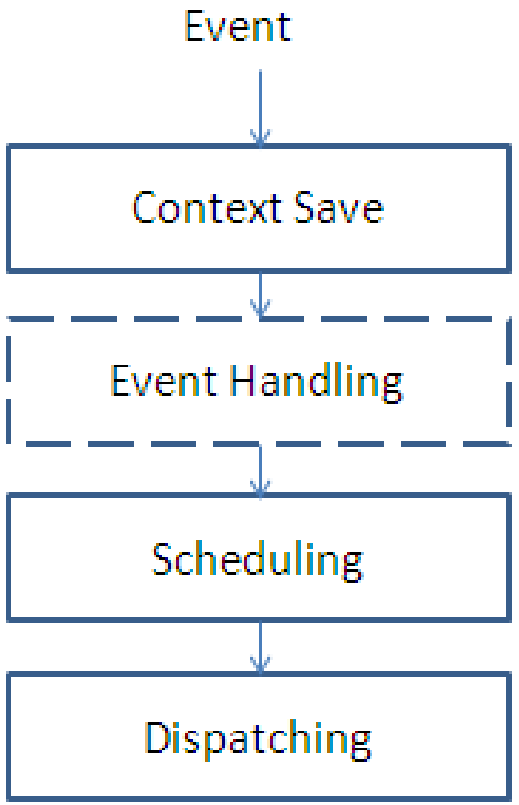
- Process context, including program counter and other registers is saved.
- Update the PCB of the process that was currently running state. The new state assigned may be one of the other states (Ready, Blocked, Ready/Suspend or Exit).
- Move this updated PCB to appropriate queue (Ready; Blocked on Event i; Ready/Suspend).
- Select another deserving process for execution.
- Update the PCB of chosen process and change the process state as Running.
- Update the memory management data structures.
- Restore the context of the chosen process so that it can resume from the point if it was interrupted last time, or can start its execution if it was loaded for the first time.

Process context switch Vs mode switch

- Context Switch:
  - Execution of a process is stopped to respond an interrupt.
  - Needs to save Process Image be saved of one process and load process image of the new process loaded.
  - The processes are switched and processes keep on changing their status as Running and Not running.
- Mode switch:
  - Every process may switch in between a low privileged user mode and high privileged kernel mode in its lifetime.
  - Process continues to execute even after mode switches

Fundamental kernel functions of process control

- Scheduling: Choose the process as per the scheduling policy to be executed next on the CPU.
- Dispatching: Set up execution of the chosen process on the CPU.
- Context save: Save information concerning an executing process when its execution gets suspended



Fundamental kernel functions of process control

- Occurrence of the event calls the context save functionality and an appropriate event handling procedure.
- Event handling may initiate some processes, hence the scheduling function gets invoked to choose the process and in turn,
- The dispatching function transfers control to the new process.

Control/Data structures maintained by OS to manage processes

- **Memory Tables**
- **I/O Tables**
- **File Tables**
- **Process table**

Control structures maintained by OS to manage processes

- **Memory Tables:**
  - Memory tables keep track of both main and secondary memory.
  - Active processes are stored in main memory and when required, they are moved to secondary memory through the mechanism called 'swapping'.
  - The memory tables maintain the following information:
    - –The main memory allocation to all processes in system
    - –The secondary memory allocation to all processes in system
    - –Shared memory regions in main and virtual memory and their attributes
    - –Miscellaneous information required to manage virtual memory.

Control structures maintained by OS to manage processes

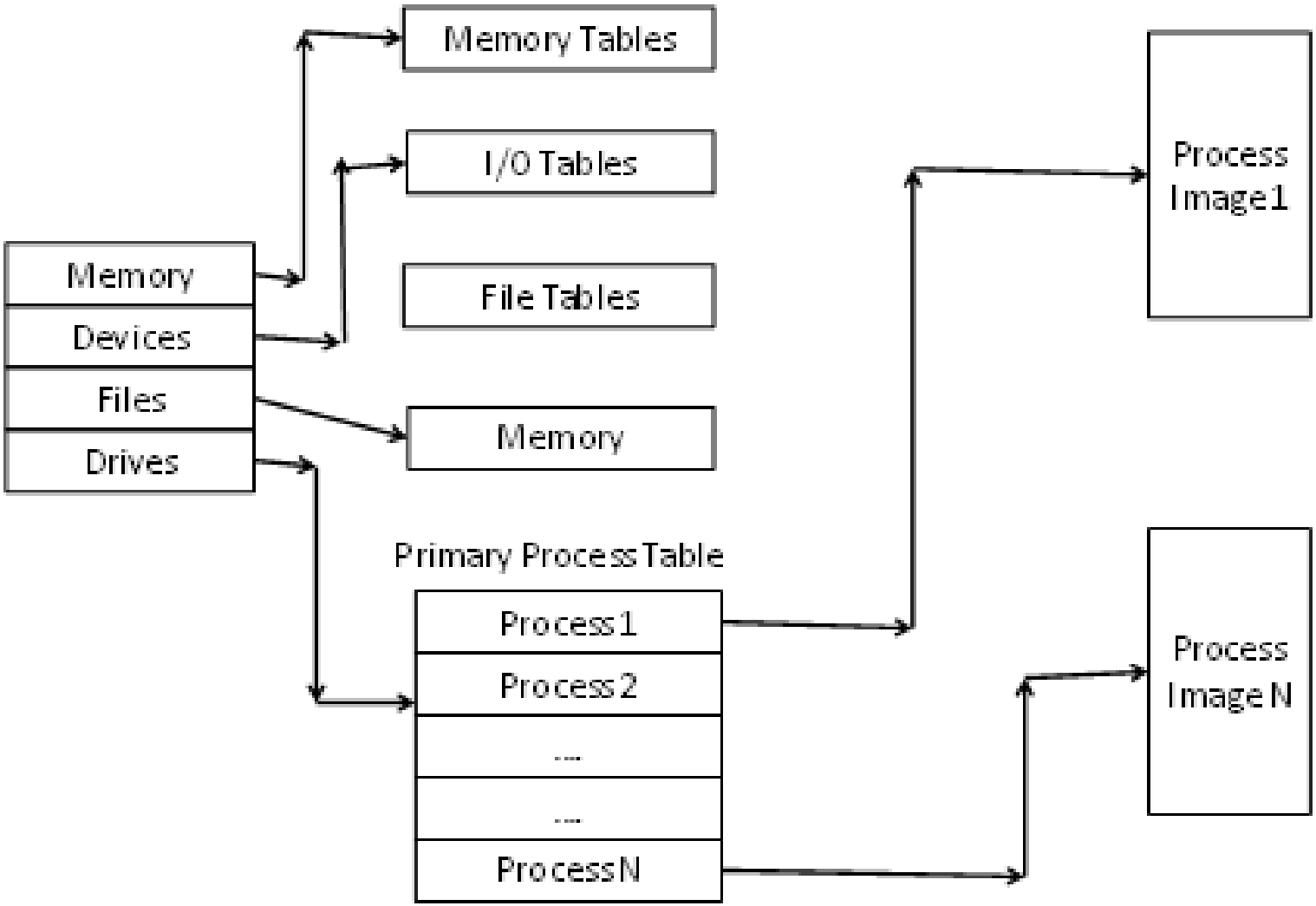
- **I/O Tables:**
  - I/O tables keep track of I/O devices and channels in the computing system.
  - The I/O devices are also resources required by processes.
  - So at any given instance, I/O devices may be available or allocated to a particular process.

Control structures maintained by OS to manage processes

- **File Tables:**
  - File tables keeps track of;
    - – all files,
    - –their locations on secondary memory,
    - –their current statuses and
    - –other attributes such security, sharing, etc.
  - Most of the operating systems, this information is maintained by a module called File Management System.

Control structures maintained by OS to manage processes

- **Process table:**
  - Process tables manage processes.
  - They maintain information of:
    - processes,
    - their child process references,
    - statuses,
    - allocated resources,
    - process contexts,
    - information required for process synchronization and so on.
  - These pieces of information are stored in process images.


Memory Tables
I/O Tables
File Tables
Memory
Memory
Devices
Files
Drives
Primary Process Table
Process 1
Process 2
...
...
Process N
Process Image 1
Process Image N

Different interaction mechanisms used by processes

Interaction Mechanism	Description
Data Sharing	The processes interact with each other by altering data values. If more than one processes update the data the same time, they may leave the shared in inconsistent state. So, shared data items are protected against simultaneous access to avoid such situation.
Message Passing	In this mechanism, the processes exchange information by sending messages to each other.
Synchronization	In certain computing environments, the processes are required to execute their actions in some particular order. To help this happen, the processes synchronize with each other to maintain their relative timings and execute in the desired sequence.
Signals	The processes may wait for events to occur. It can be intimated to processes through the signaling mechanism.

Process and thread

- Process: an instance of program in execution
- Thread : a dispatchable unit of work

Traditional process

- Single thread of execution
- Needs entire process context to execute so considered as heavyweight
- Doesn’t support multiple parallel executions

E.g. you wouldn’t get notifications in background if you are using the app

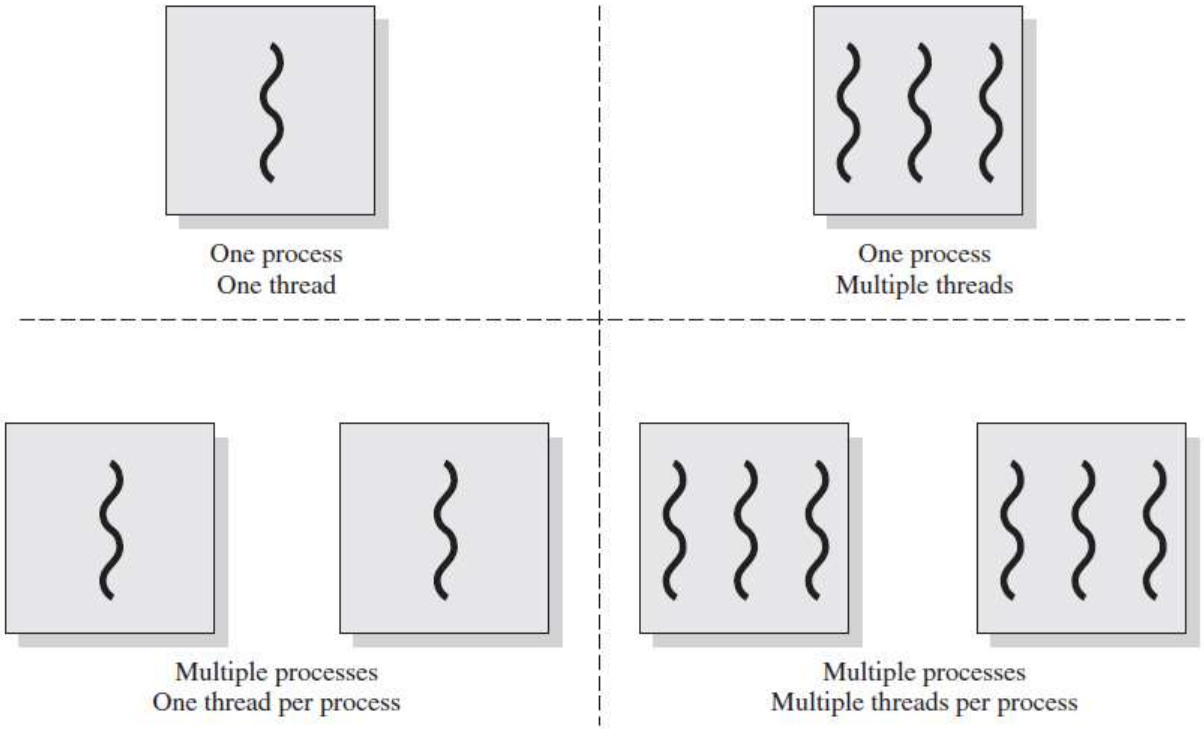
Thread

- Supports Parnellism with multiple threads of execution at a time
- A thread executes sequentially and is interruptable so that the processor can turn to another thread
- Does not need entire process context to execute so considered as lightweight
- Includes the program counter and stack pointer) and its own data area for a stack
- Supports multiple parallel executions

e.g. Notifications in background while you are using the app

- The idea is to **achieve parallelism by dividing a process into multiple threads.**

- *Multithreading* refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.



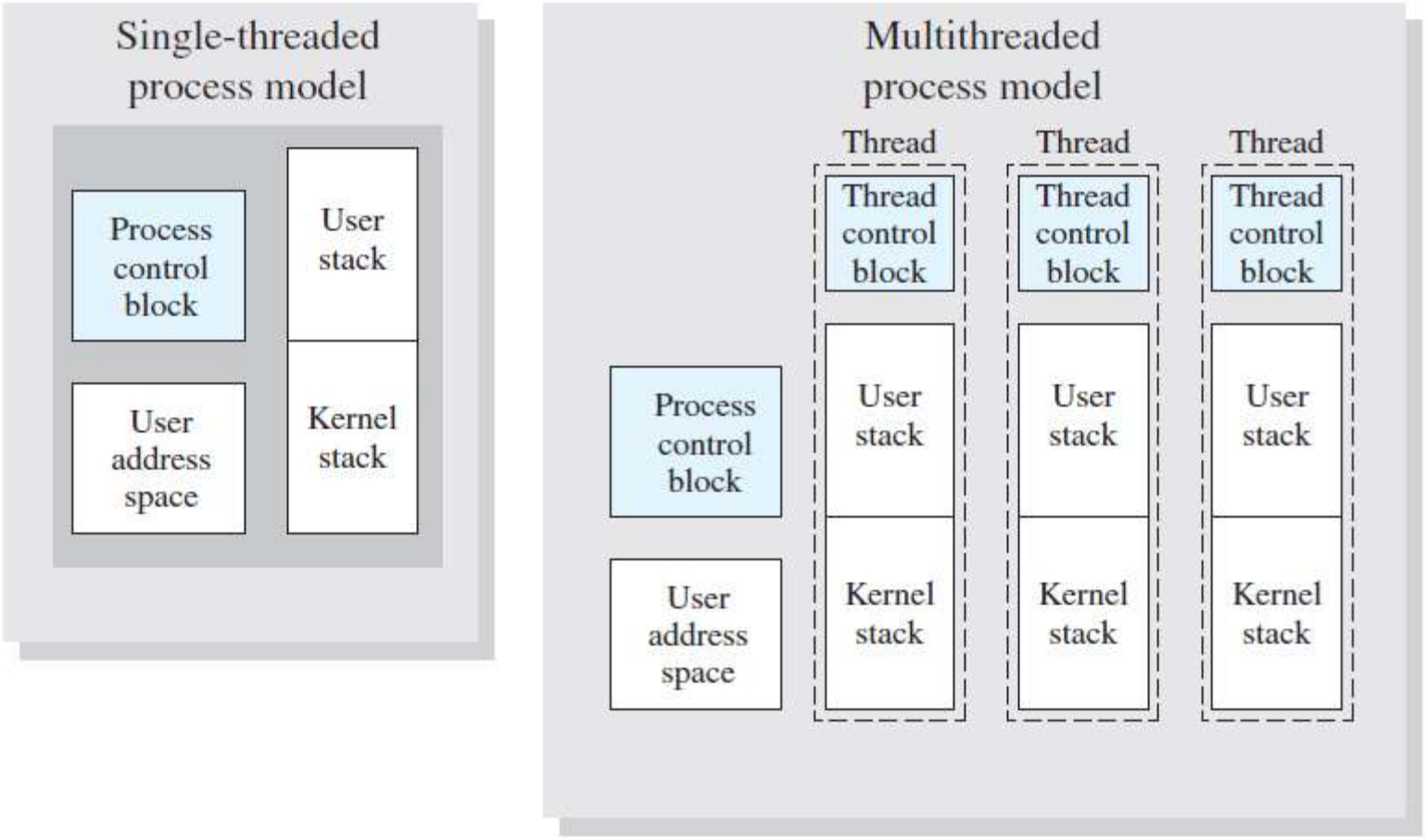
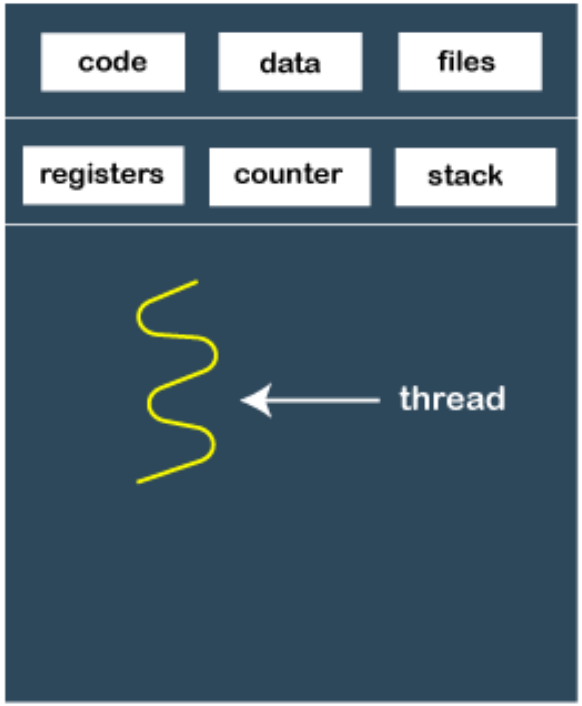
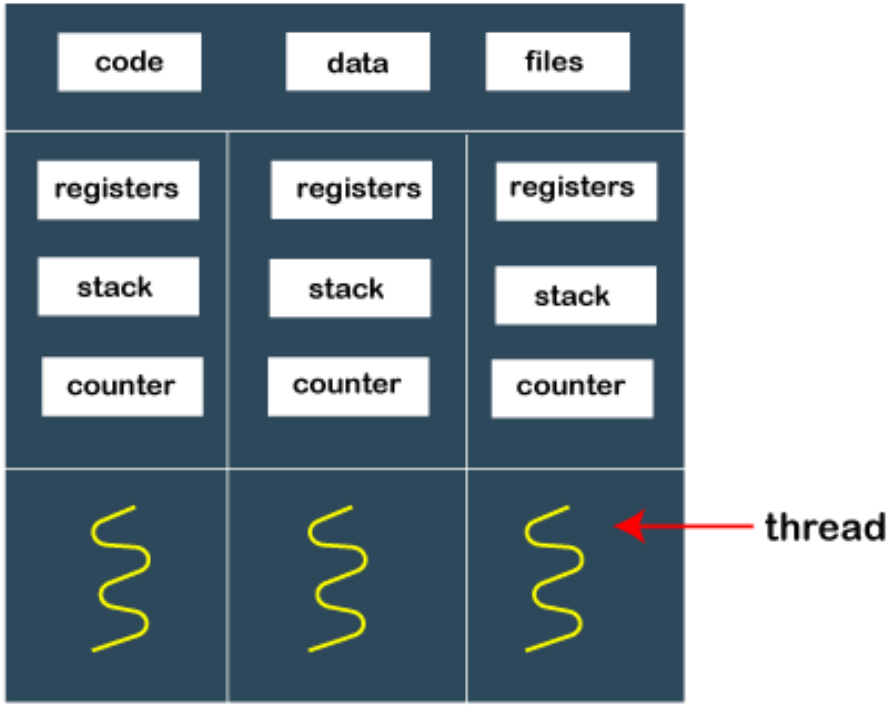


Figure 4.2 **Single Threaded and Multithreaded Process Models**



Single-threaded process



Multi-threaded process

Image courtesy : https://www.javatpoint.com/process-vs-thread

The key benefits of threads

- It takes far less time to create a new thread in an existing process than to create a brand-new process.
- It takes less time to terminate a thread than a process.

Types of threads

- Kernel level threads : managed by kernel
- User level threads : managed by user with support from programming languages and libraries
- Hybrid threads

Thread

- Create
- Join
- Terminate

Threads

```
import threading
import time

# Define a function for the thread to execute
def count_numbers(thread_name, count_to):
    for i in range(1, count_to + 1):
        print(f"{thread_name} counting: {i}")
        time.sleep(0.5)  # Sleep for half a second

# Create threads
thread1 = threading.Thread(target=count_numbers, args=("Thread 1", 5))
thread2 = threading.Thread(target=count_numbers, args=("Thread 2", 5))

# Start threads
print("Starting threads")
thread1.start()
thread2.start()

# Join threads (wait for them to complete)
thread1.join()
thread2.join()

print("Threads have completed their tasks")
```

```
PS C:\Users\Swati> python -u "c:\Users\Swati\Downloads\import
threading.py"
Starting threads
Thread 1 counting: 1
Thread 2 counting: 1
Thread 2 counting: 2
Thread 1 counting: 2
Thread 1 counting: 3
Thread 2 counting: 3
Thread 2 counting: 4
Thread 1 counting: 4
Thread 2 counting: 5
Thread 1 counting: 5
Threads have completed their tasks
```

ULT Vs KLT

Sr No	System Call	Description
1	fork()	This system calls creates a new process.
2	exec()	This call is used to execute a new program on a process.
3	wait()	This call makes a process wait until some event occurs.
4	exit()	This call makes a process to terminate
5	getpid()	This system call helps to get the identifier associated with the process.
6	getppid()	This system call helps to get the identifier associated with the parent process.
7	nice()	The current process priority can be changed with execution of this system call.
8	brk()	This call helps to increase or decrease the data segment size of the process.
9	Kill()	The forced termination of any process can be executed with this system call.
10	Signal()	This system call is invoked for sending and receiving software interrupts

fork system call

- The **fork()** system call is used to create a new process. When the system executes this system call in response to process creation request, following steps are carried out.
- The operating system allocates a slot in the process table for the newly created process.
- This new process, i.e. child process is then assigned a unique ID.
- System then creates a logical copy of the parent process context.
- The file and inode table counters associated with parent process are incremented as this area may be shared between parent process and the child process.
- The child process ID is returned to the process and ‘0’ value is assigned to the child process.
- All the above steps are carried out in kernel of the parent process.
- The control of execution may remain with the parent process, may be transferred to child process or handed over to a third process leaving the parent and child processes in ‘Ready-to-Run’ state.

Processor Scheduling

I/O scheduling Vs Processor Scheduling

- **I/O Scheduling:** I/O scheduling is a process that is involved in making the decision as to which process's pending I/O request should be handled by an available I/O device.

- **Processor Scheduling:** Processor scheduling is a process that makes a decision of which process should get hold of processor next. This process needs the 'dispatcher' – a software module of short term scheduler to make this decision.

- **I/O bound process:** I/O bound processes are the ones those spend more time doing I/O operations than computation, though it may have many short CPU bursts.
-
- **CPU bound processes:** CPU bound processes spend more time in computations and so they have long CPU bursts. Although they may also involve in very short durations of I/O operations.

Parallelism and Concurrency

- **Parallelism:**
  - Parallelism is the quality of the processes to execute at the same time.
  - Two processes or events are said to be parallel if they occur at the same time. In parallelism, multiple processes can be active at a time.
- **Concurrency:**
  - Concurrency does not mean parallel.
  - With concurrency, multiple processes are executed one after another by interleaving their execution in such a way that it creates an illusion of parallelism.
  - In concurrency only one process can be active at a time.

Parallelism and Concurrency

- Concurrency is achieved by interleaving process execution on (may be single) CPU which creates an illusion that processes run in parallel,
- While parallelism is obtained by multiple processors operating in parallel at a time.
- Both the techniques achieve computation speedup, though the inherent mechanism used by both concepts is different.

Advantages of process concurrency.

- The processes in a multiprogramming environment are a blend of I/O bound and CPU bound processes.
- If these processes are executed sequentially, they underutilize the CPU and I/O devices both.
- If executed in interleaved fashion, the system gives better efficiency, relatively lower response, turn around and waiting times.

Process Scheduler

- Process scheduler or the Dispatcher is a software module that works for CPU(processor) scheduling and chooses the next process to run.
- Its main activities involve switching the process context, switching the execution mode to user.
- The dispatcher is required to work very fast to improve the efficiency of execution.

Processor schedulers

- Long term scheduler
- Mid term scheduler
- Short term scheduler

System Calls for Process Management

Questions
?

Module 2.2 Thread

Swati Mali
Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Outline

- Thread
- Process Vs Thread
- Example

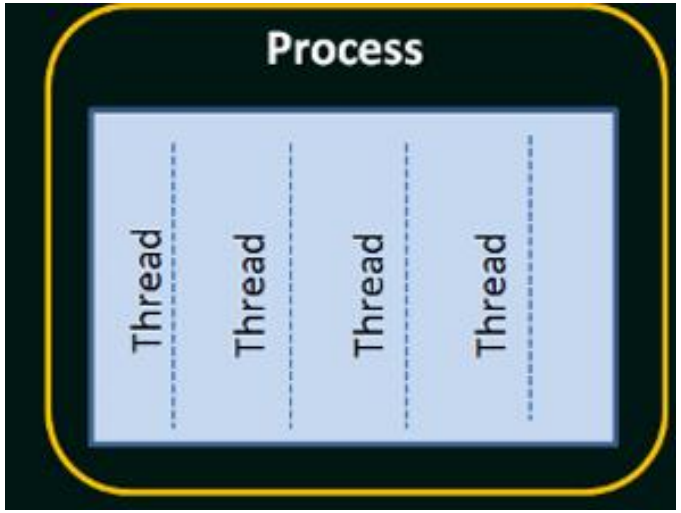
Thread

- Supports Parallelism with multiple threads of execution at a time
- A thread executes sequentially and is interrupt-able so that the processor can turn to another thread
- Does not need entire process context to execute so considered as lightweight.
- Includes the program counter and stack pointer) and its own data area for a stack
- Supports multiple parallel executions

  e.g. Notifications in background while you are using the app
- The idea is to **achieve parallelism by dividing a process into multiple threads.**

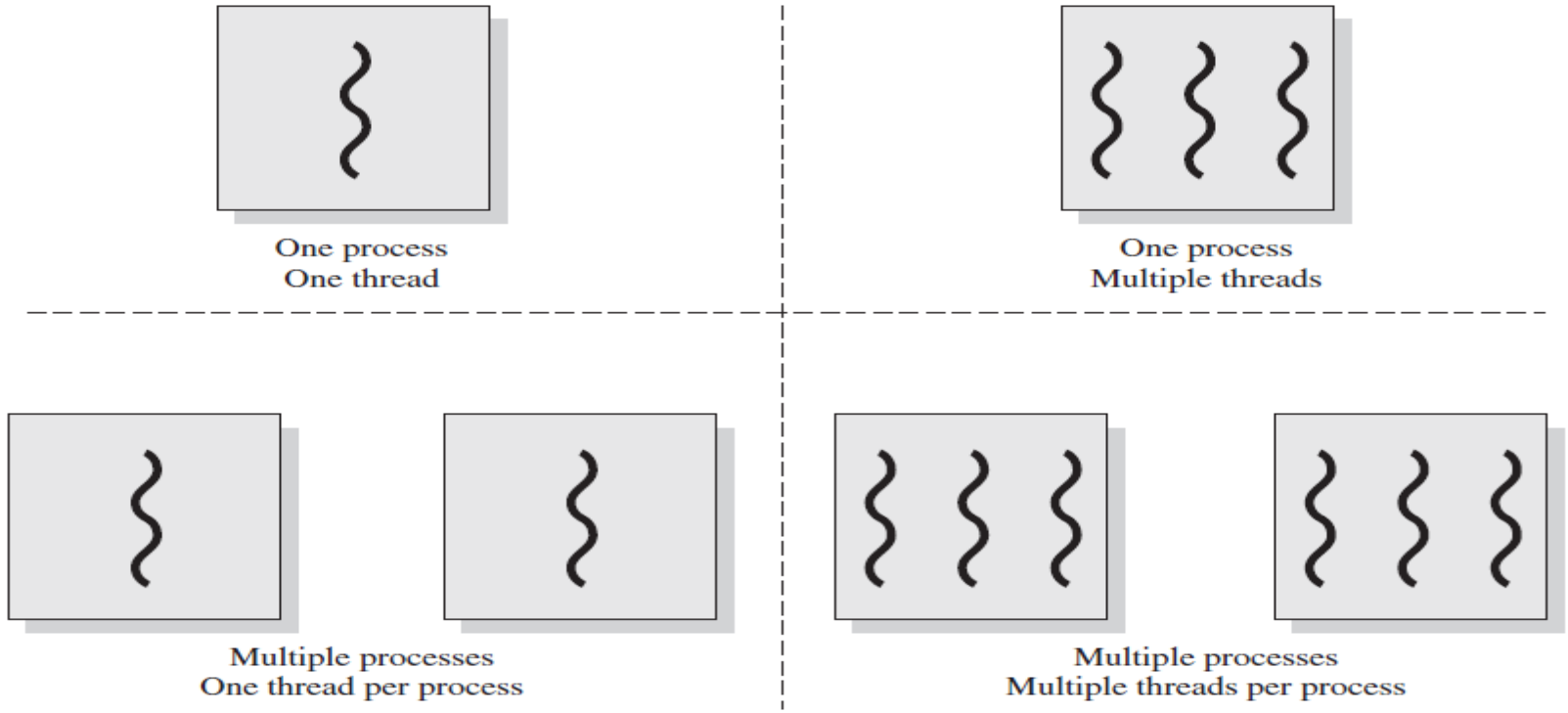
Process and Thread

- A process is an instance of a program in execution.
- It is a basic unit of work that can be scheduled and executed by the operating system.
- A thread is the unit of execution within a process.
- A process can have anywhere from just one thread to many threads.



Thread

- *Multithreading* refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.



Thread

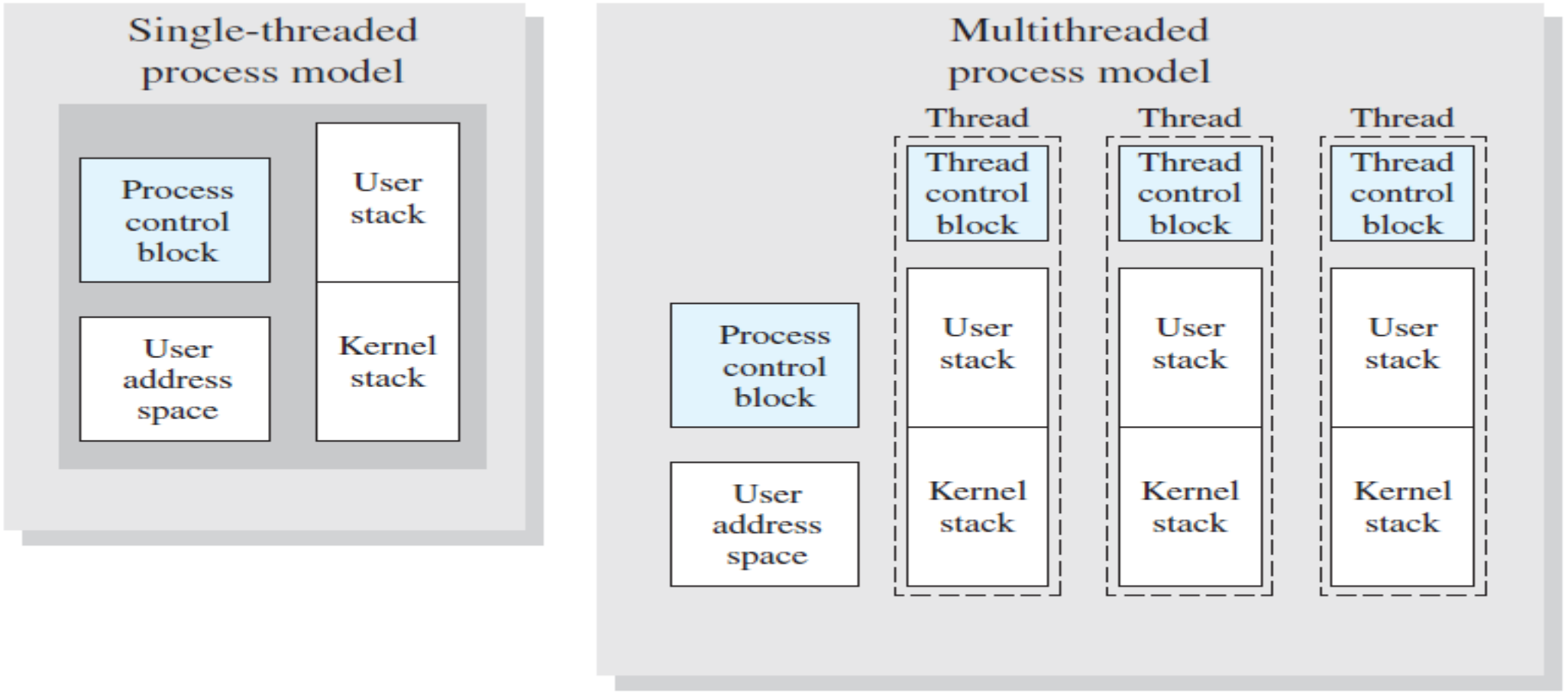


Figure 4.2 **Single Threaded and Multithreaded Process Models**

The key benefits of threads

- It takes far less time to create a new thread in an existing process than to create a brand-new process.
- It takes less time to terminate a thread than a process.

Types of threads

- Kernel level threads : managed by kernel
- User level threads : managed by user with support from programming languages and libraries
- Hybrid threads

Thread

- Thread libraries provide programmers with API for the creation and management of threads.
- Three types of Thread
  - **POSIX Pitheads** may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - **Win32 threads** are provided as a kernel-level library on Windows systems.
  - **Java threads**: Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system.

Thread

- Create
- Join
- Terminate

Create, Start and Join

- A thread can be created using the **Thread class** provided by the threading module. Using this class, you can create an instance of the Thread and then start it using the **.start()** method.

Create & Start

```python
import threading

# Creating Target Function
def num_gen(num):
    for n in range(num):
        print("Thread: ", n)

# Main Code of the Program
if __name__ == "__main__":
    print("Statement: Creating and Starting a Thread.")
    thread = threading.Thread(target=num_gen, args=(3,))
    thread.start()
    print("Statement: Thread Execution Finished.")
```

- 1st execution

```
Statement: Creating and Starting a Thread.
Thread:  0
Statement: Thread Execution Finished.
Thread:  1
Thread:  2
```

- 2nd execution

```
Statement: Creating and Starting a Thread.
Thread:  0
Thread:  1
Statement: Thread Execution Finished.
Thread:  2
```

Join() Method

- The join() method is used in that situation, it doesn't let e**xecute the code further until the current thread terminates**.

```
import threading

# Creating Target Function
def num_gen(num):
    for n in range(num):
        print("Thread: ", n)

# Main Code of the Program
if __name__ == "__main__":
    print("Statement: Creating and Starting a Thread.")
    thread = threading.Thread(target=num_gen, args=(3,))
    thread.start()
    thread.join()
    print("Statement: Thread Execution Finished.")
```

```
Statement: Creating and Starting a Thread.
Thread:  0
Thread:  1
Thread:  2
Statement: Thread Execution Finished.
```

Assignment

- ULT vs KLT
- Pthread
- POSIX

Question ?

Linux Scheduling

Nirmala Shinde Baloorkar
Assistant Professor
Department of Computer Engineering

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Outline

- Fair Share Scheduling
- Traditional UNIX Scheduling

Fair-Share Scheduling

- Scheduling decisions based on the process sets
- Each user is assigned a share of the processor
- Objective is to **monitor usage to give fewer resources to users** who have had more than their fair share and more to those who have had less than their fair share

Fair-Share Scheduler

Time	Process A Priority	Process A Process CPU count	Process A Group CPU count	Process B Priority	Process B Process CPU count	Process B Group CPU count	Process C Priority	Process C Process CPU count	Process C Group CPU count
0	60	0 1 2 • • • 60	0 1 2 • • • 60	60	0	0	60	0	0
1	90	30	30	60	0 1 2 • • • 60	0 1 2 • • • 60	60	0	0 1 2 • • • 60
2	74	15 16 17 • • • 75	15 16 17 • • • 75	90	30	30	75	0	30
3	96	37	37	74	15	15 16 17 • • • 75	67	0 1 2 • • • 60	15 16 17 • • • 75
4	78	18 19 20 • • • 78	18 19 20 • • • 78	81	7	37	93	30	37
5	98	39	39	70	3	18	76	15	18

Group 1 Group 2

Colored rectangle represents executing process

Figure 9.16 Example of Fair Share Scheduler — Three Processes, Two Groups

Traditional UNIX Scheduling

- Used in both SVR3 and 4.3 BSD UNIX
  - these systems are primarily targeted at the time-sharing interactive environment
- Designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve
- Employs multilevel feedback using round robin within each of the priority queues
- Makes use of one-second preemption
- Priority is based on process type and execution history

Scheduling Formula

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor

Bands

- Used to optimize access to block devices and to allow the operating system to respond quickly to system calls
- In decreasing order of priority, the bands are:

Example of Traditional UNIX Process Scheduling

Time	Process A Priority	Process A CPU count	Process B Priority	Process B CPU count	Process C Priority	Process C CPU count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Colored rectangle represents executing process

Figure 9.17 Example of Traditional UNIX Process Scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

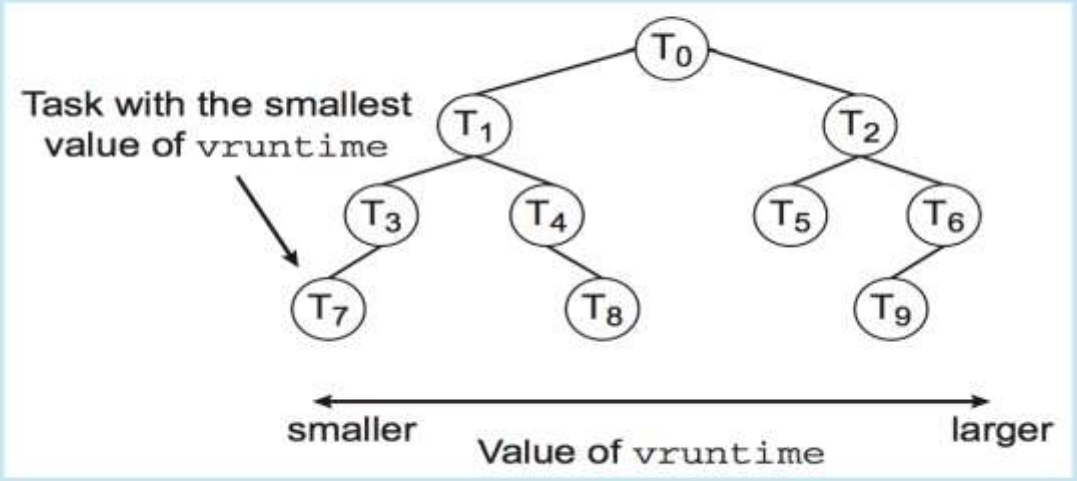
- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (cont...)

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

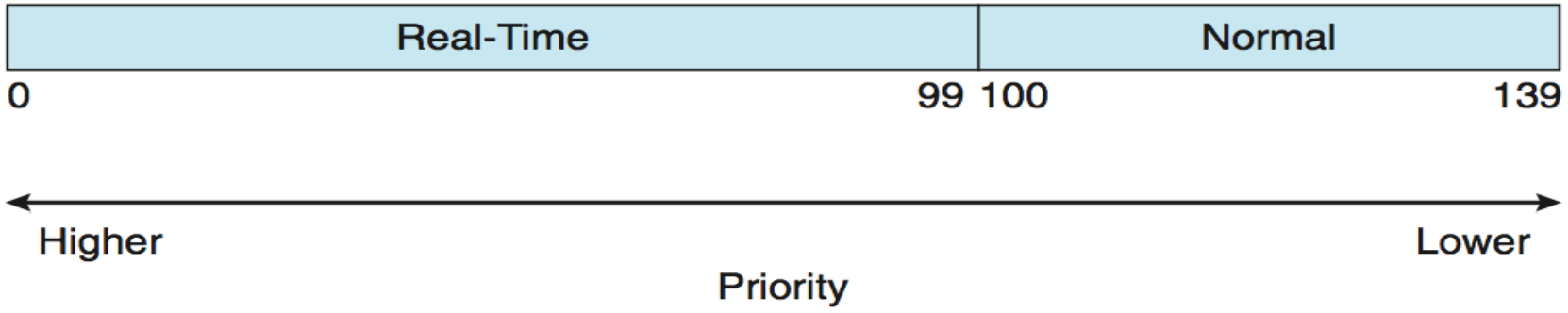
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg\,N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Question ?

Process Concept and scheduling

Ms. Swati Mali

swatimali@gmail.com
K. J. Somaiya College of Engineering
Somaiya Vidyavihar University

Process Concept and scheduling	
2..1	Process: Concept of a Process, Process States, Process Description, Process Control Block, Operations on Processes. Threads: Definition and Types, Concept of Multithreading
2.2	Multicore processors and threads. Scheduling: Uniprocessor Scheduling - Types of Scheduling: Preemptive and, Non-preemptive, Scheduling Algorithms: FCFS, SJF, SRTN, Priority based, Round Robin, Multilevel Queue scheduling.
2.3	Introduction to Thread Scheduling
2.4	Linux Scheduling.

Basic Terminologies & Definitions

Task and Program

- **Task:**
  - Task is a unit of assigned work.
  - Can also be defined as the unit of programming controlled by OS.
  - Depending on the OS design the task may involve one or more processes.
  - Example: Bake a cake
- **Program**:
  - A program is defined as sequence of instruction written to accomplish a task.
  - A program may comprise of one or more processes depending on the statement being executed.
  - Generally referred as a passive entity that does not perform any action.
  - Example: A particular recipe given in book.

Process

- This is an instance of program in execution.
- The static statements in the program when executed, take the process form.
- In contrast to the program, a process is an active entity which needs a set of resources to perform its function.
- The Linux kernel internally represents processes as tasks.
- A process elements are: a program, data and process state.
- Example: actually following the steps given the book.

Thread

- A thread is a lightweight process.
- Also defined as the smallest processing unit that is scheduled by an operating system.
- A thread must be part of a process as it shares the process environment viz, code, data and resources with other threads.
- Threading has increased the computing efficiency to significant extent.

Job

- **Job**:
  - A job is unit of work submitted by user to the system.
  - A job may be interactive or a batch job which may in turn consist of one or more processes.

Process States

- **Process State:**
  - A process state is a process' internal data maintained by OS for the purpose of supervision and control of the process.
  - Also called as executional context of the process.
- Process States: (more states, trasitions and reasons for transitions will be discussed later..)
  - New
  - Ready
  - Running
  - Swapped
  - Waiting/blocked
  - Suspended, etc

Process Context

- **Process Context:**
  - Whenever a running process is taken away from processor, some of the process state's information needs to be retained.
  - This information called as process context helps the process to resume from the point where it was stopped last time.
- The context of a process includes:
  - its address space,
  - stack space,
  - virtual address space,
  - register set image i.e. Program Counter, Stack Pointer, Program Status Word, Instruction Register and other general processor registers,
  - accounting information,
  - associated kernel data structures and
  - current state of the process (waiting, ready, etc).

Process Modes

- The operational or the privilege mode of process execution is called process mode.
- A process executes in user mode or a kernel mode.
- Processor switches the process in between these modes depending on the code the process is running.

Event

- An activity that is happens or is expected to happen.
- Generally this a software message exchanged when the activity occurs.
- In operating systems, the events may cause the processes to change their state.
  - e.g. mouse click, file lock reset, etc.

- Check- event viewer in windows OS

Process concepts

- **Process priority:**
  - In a multiprogramming system, the processes are assigned numerical privileged values indicating their relative importance and/or urgency and/or value.
- **Preemption:**
  - Preemption is the ability of the system to take over a currently executing process by another one (possibly with high privileged one) with an intention to resume the preempted process later on.
- **Preemptive:**
  - Preemptive entities are the one those can be taken over by another similar type of entities.
- **Non-preemptive:**
  - Non-preemptive entities are the one those cannot be taken over by other entities

Degree Of Multiprogramming

- With multiprogramming, the CPU can run multiple programs simultaneously or concurrently.
- The degree of multiprogramming is the maximum number of allowed processes at a time in a system that does not let the CPU performance degrade than a certain threshold.

Process

Basic elements of a process

- Process ID: This is a unique identifier assigned to the process
- Code: This is program code.
- Data : These are the data and files required for execution
- Resources : These are different types of resources allocated by OS
- Stack: This contains parameters for the functions/procedures called and their return addresses.
- Process state: This is one of the eleven states process is in.

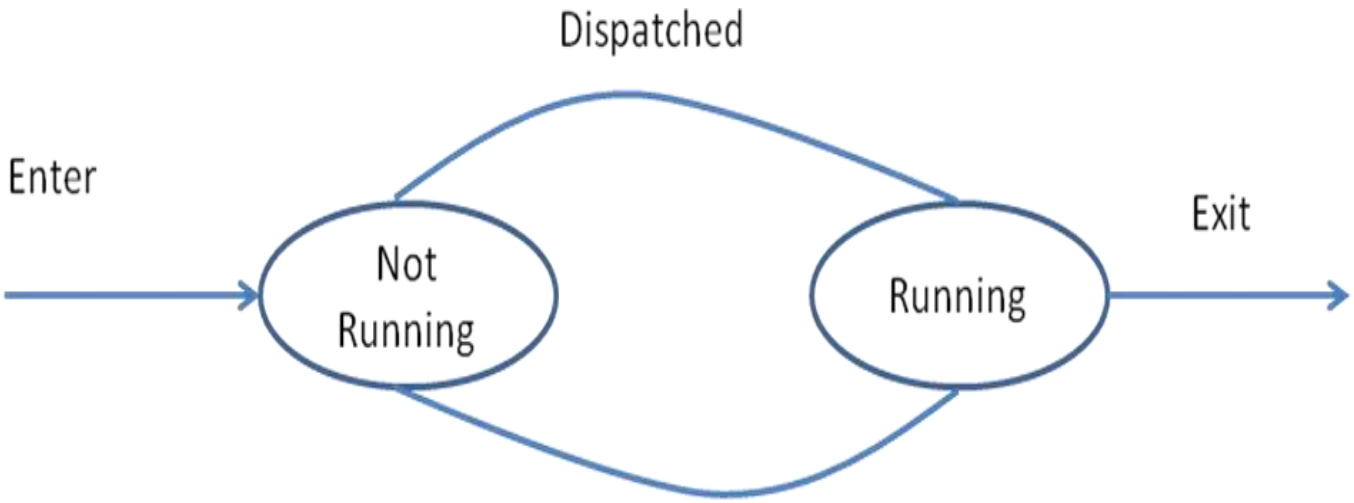
Process state transition

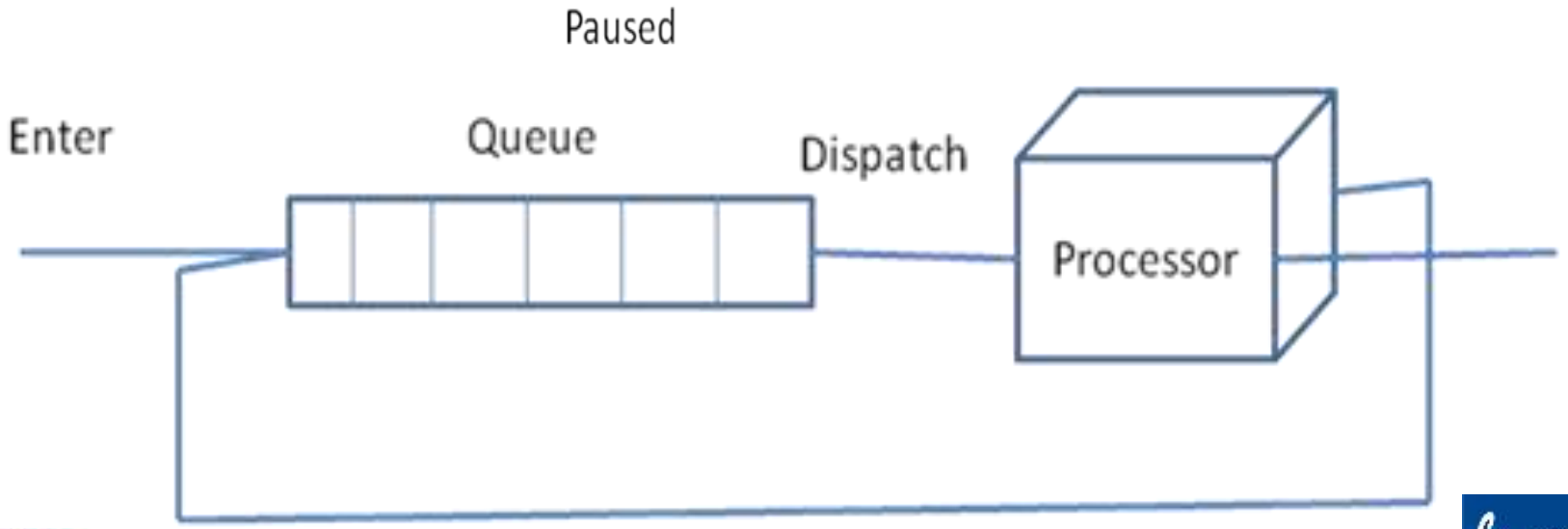
- A state transition for process is defined as a change in its state.
- The state transition occurs in response to some event in the computing system

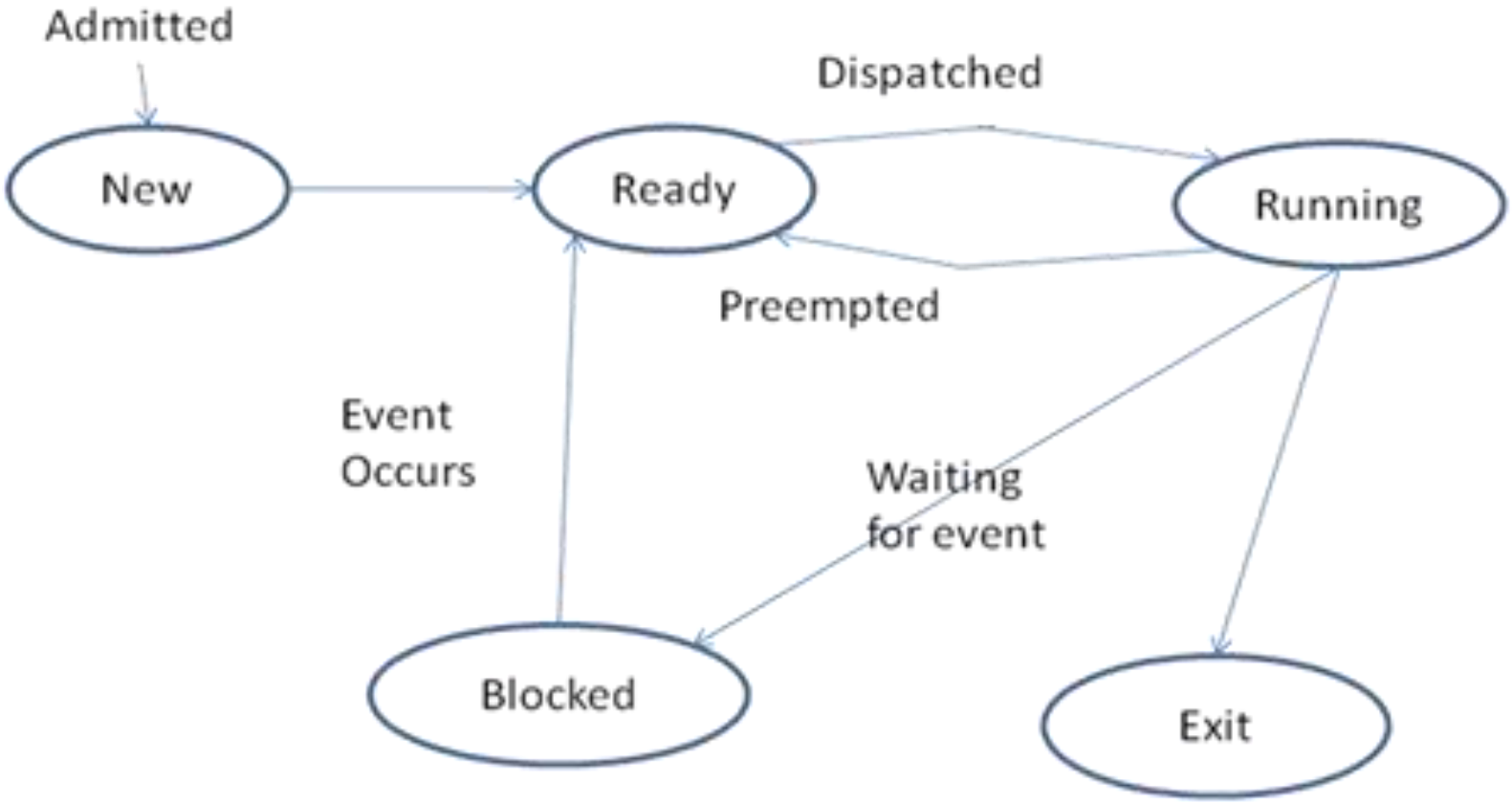
States in process state-transition diagram

- New: The process is being created
- Ready: the process is ready with all required resources and waiting to be assigned to a processor
- Waiting/Blocked: the process is waiting for some event to occur
- Suspended: the process was waiting for some event to occur, but then is swapped to secondary memory to make room for new processes getting ‘blocked’.
- Terminated: the process has finished its execution

Two State Model






Admitted
Dispatched
New
Ready
Running
Preempted
Event
Occurs
Waiting
for event
Blocked
Exit

State transition	Description
Null New	A new process is created to execute a program.
New ready	The OS may move a process from New to Ready state depending on the predefined maximum number of processes allowed (Degree of multiprogramming).
Ready Running	The process is scheduled by dispatcher. The CPU starts or resumes execution of the instruction codes
Blocked Ready	The request initiated by process is satisfied or the event on which it is waiting occurs.
Running Ready	The process is preempted by the OS decides to execute some other process. This transition takes place may be because of expiration of time quantum or arrival of high priority process.
Running Blocked	The running process makes request for resource(s) or needs some event to occur to proceed further. The process then calls for a system call to indicate its wish to wait till the resource or the event becomes available.
Running Termination	The program execution is completed or terminated.

Causes of process initiation

- **New batch job:** while processing the new batch of jobs submitted, the OS creates a process to execute the same.
- **Interactive logon:** when a user logs into the system, a new process is created.
- **Created by OS to provide some service:** The OS initiates a process to perform the service requested by user directly or indirectly, without making the user to wait.
- **Spawned by an existing process**: To support Modularity and/or parallelism, a user program can create some number of new processes.

How the OS creates a process?

1. Create a process
2. Assign a unique process ID to newly created process
3. Allocate the memory and create its process image
4. Initialize process control block
5. Set the appropriate linkages to the different data structures such as ready queue etc.
6. Create or expand the other data structures if required

```
algorithm fork
input:   none
output: to parent process, child PID number
        to child process, 0
{
      check for available kernel resources;
      get free proc table slot, unique PID number;
      check that user not running too many processes;
      mark child state "being created;"
      copy data from parent proc table slot to new child slot,
      increment counts on current directory inode and changed root (if applicable);
      increment open file counts in file table;
      make copy of parent context (u area, text, data, stack) in memory;
      push dummy system level context layer onto child system level context;
                  dummy context contains data allowing child process
                  to recognize itself, and start running from here
                  when scheduled;
      if (executing process is parent process)
      {
            change child state to "ready to run;"
            return(child ID);        /* from system to user */
      }
      else       /* executing process is the child process */
      {
            initialize u area timing fields;
            return(0);        /* to user */
      }
}
```

Causes of process blocking

- Process requests an I/O operation
- Process requests memory or some other resource
- Process wishes to wait for a specific interval of time
- Process waits for message from some other process
- Process wishes to wait for some action to be performed by another process.

Causes of process termination

- **Normal Completion:** The process executes an OS system call to intimate that it has completed its execution.
- **Self termination** (e.g. incorrect file access privileges, inconsistent data)
- **Termination by the parent process:** a parent process calls a system call to kill/terminate its child process when the execution of child process is no longer necessary.
- **Exceeding resource utilization:** An OS may terminate a process if it is holding resources more than it is allowed to. This step can also be taken as part of deadlock recovery procedure.
- **Abnormal conditions during execution:** the OS may terminate a process if an abnormal condition occurs during the program execution. (e.g. memory protection violation, arithmetic overflow etc)
- **Deadlock detection and recovery**

Example Events

- Process creation event
- Process termination event
- Timer event: occurrence of timer interrupt
- Resource request event: a resource request is made by a process
- Resource release event: process releases a resource and notifies.
- I/O initiation request event: a process wishes to initiate I/O operation
- I/O completion event: a process finishes I/O operation
- Message send event: A message is sent by one process to an another one.
- Message receive event: a process receives a massage by another one.
- Signal send event: a signal is sent by a process to an another one
- Signal receive event: a process receives a signal
- A program interrupt: An instruction in currently executing process executes some illegal operation and malfunctions.

A process image

- The collection of program, stack, data and process attributes are called process image.
- The process image is generally maintained as a continuous or contiguous block of memory which resides in secondary memory.
- To execute a process, its process image is loaded in main memory or virtual memory.

Typical Elements of Process Image

Process image element	Description
User Stack	Part of user space that contains program data, a user stack area, and modifiable/editable programs.
User Program	The program under execution.
Stack	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls.
Process Control Block	Process control block(PCB) contains the data which is used by OS to control and manage the process.

Process Control Block

- A PCB contains all information pertaining to a process that is used in controlling the process operation, resource information and information needed for inter-process communication.
- PCB components:
  - Identifiers
  - Processor State Information
  - Process Control Information

PCB components: Identifiers

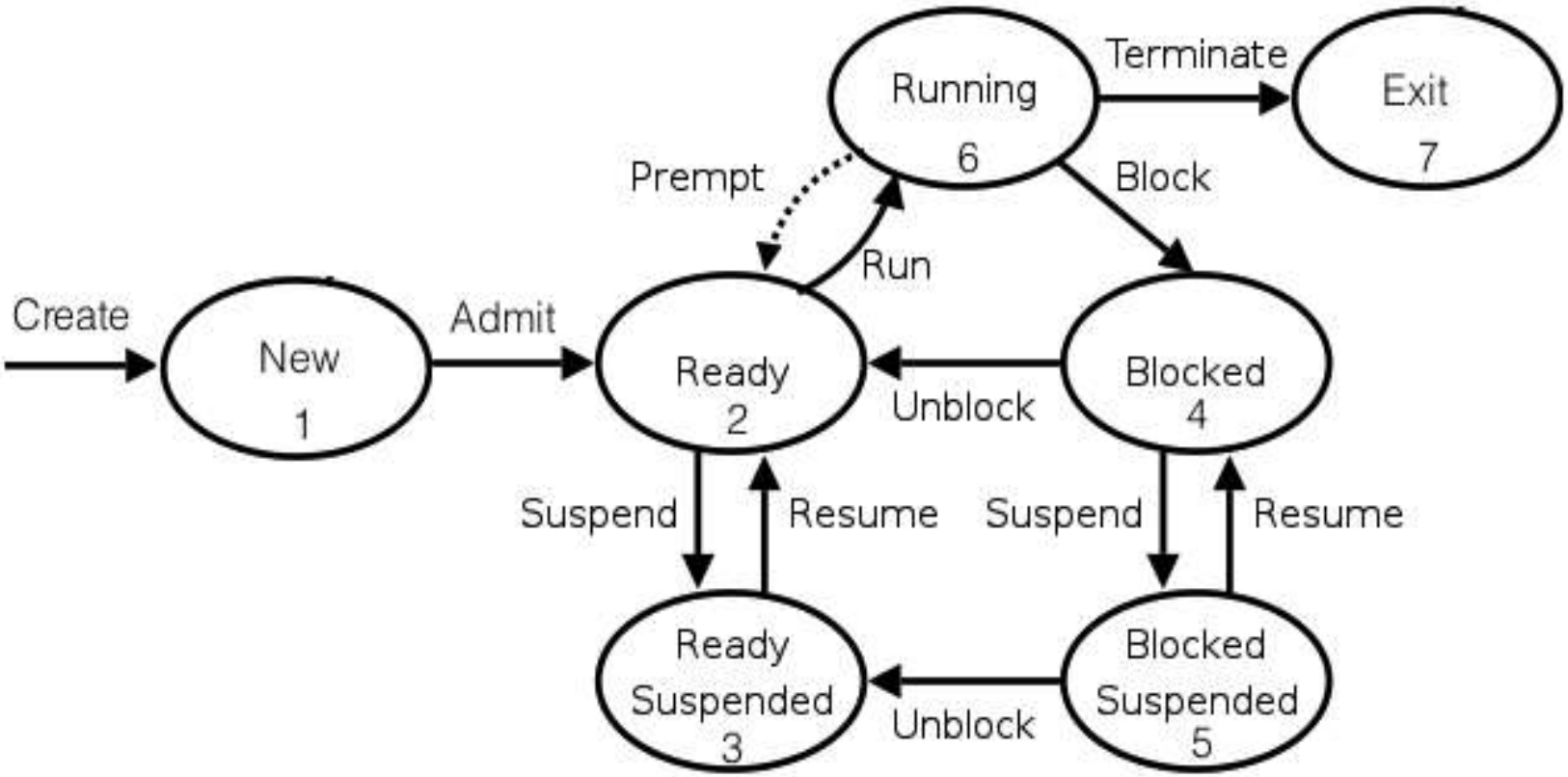
PCB Field	Contents
Identifiers	
Process ID	This is unique process ID assigned to it at the time of creation
Child and Parent IDs	The child and parent process IDs are required for process synchronization
User Identifier	The unique identifier associated with the user in multiuser systems.

Process Control Information	
Scheduling and state information	This is information about process state, priority, scheduling related information and event information in case the concerned process is waiting on that!
Data Structuring	A process may linked to other processes in queue, ring or some other structure. This information is stored in data structuring field.
Interprocess communication	The processes need various flags, messages and signals be exchanged to interact with each other. Some or all this information in stored in PCB.
Process Priority	The priority is a numeric value, which may be assigned to a process at the time of its creation. Some priorities can be altered by user and some change with process age, too.
Memory Management	This field holds the pointer to segments or to the page tables which describe portions of memory assigned to the process.
Resource ownership and Utilization	All the resources or the number of instances of resources assigned to process are described in this field.

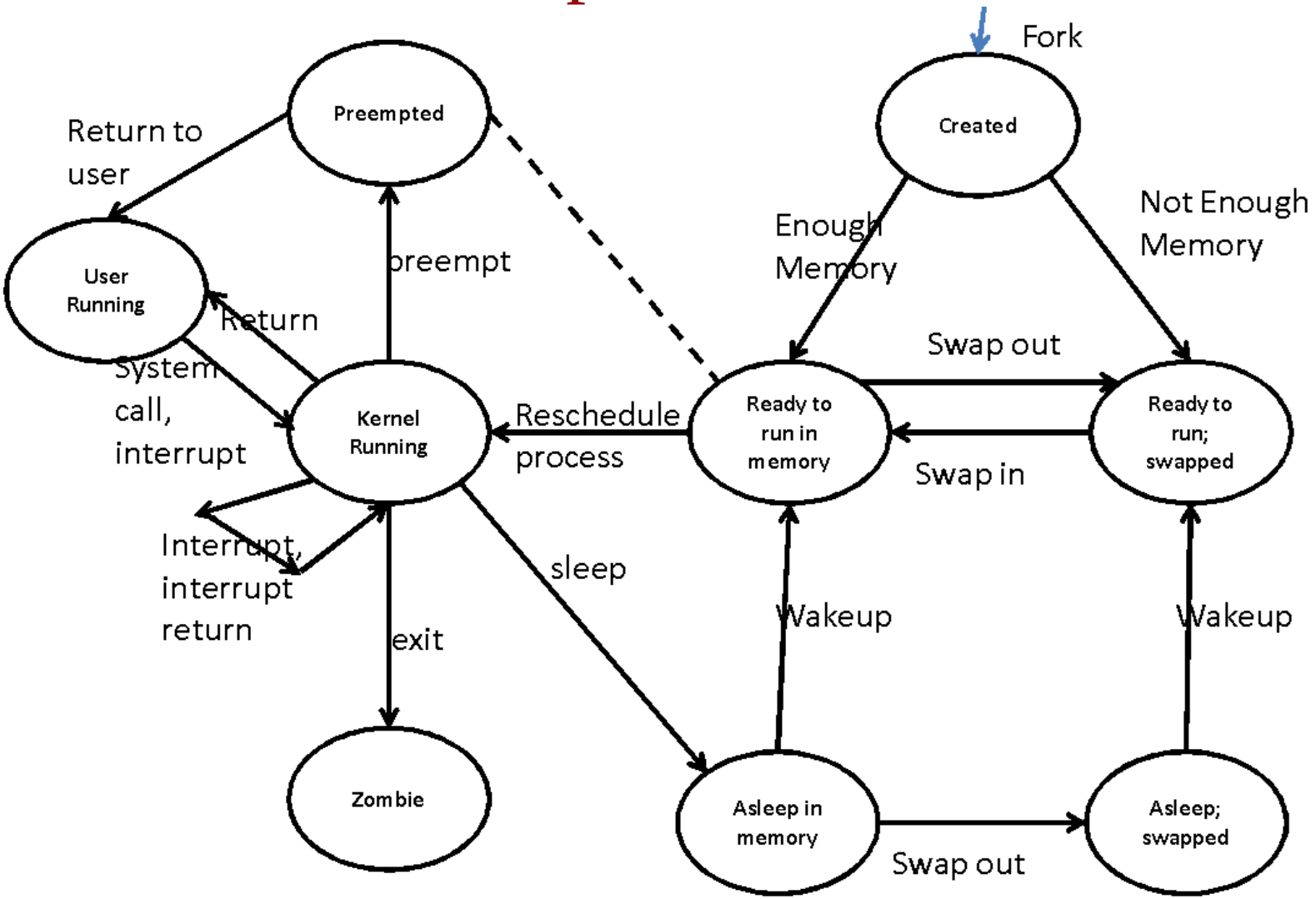
PCB components: Process State Information

Processor State Information	
User Accessible Registers	Every processor offers some general purpose registers those are accessible to user. The number of available registers differs with every processor type.
Control and Status registers	This field is also called PSW(Program Status Word) which typically contains Program counter (Contains the address of the next instruction to be fetched), Condition codes (Result of the most recent arithmetic or logical operation), Status information(Includes interrupt enabled/disabled flags, execution mode)
Stack Pointers	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls. The stack pointer points to stack top.

7- state process model



9-state process model



Process State	Description
User Running	The process is running in user mode
Kernel Running	The process is running in Kernel mode
Ready to Run, in Memory	The process is ready to run as soon as the kernel dispatches it.
Asleep in Memory	The process is blocked on some event, process is in main memory.
Ready to Run, Swapped	The process is ready to run, but the swapping module must swap it in the main memory, so that kernel can schedule it.
Sleeping, swapped	The process is blocked on some event and it is in secondary memory.
Preempted	The process was returning from kernel mode to user mode, but the kernel preempts it and performs a process switch to dispatch another process.
Created	The process is just created and is not yet ready to run. (may not have all the resources, including memory, which are required to run)
Zombie	The process no longer exists, but it leaves some information (probably accounting information) to its parent process to collect.

Reasons of process suspension

- **Swapping:** The main memory may not be enough to accommodate some ready process. So a currently not running process is shifted from main memory to secondary memory.
- **Interactive User request:** a user may wish to suspend a currently running process for debugging or to manage the use of resources
- **Timing:** A process may be executed on periodic basis and may be suspended while waiting for its next turn of execution.
- **Parent process request:** A parent process may wish to suspend its child process to examine or modify the suspended process or to coordinate the child processes.
- **Other OS reasons of suspension:** The OS may suspend a background or utility process or a process that is causing some problem in normal activities.

The process switching operation

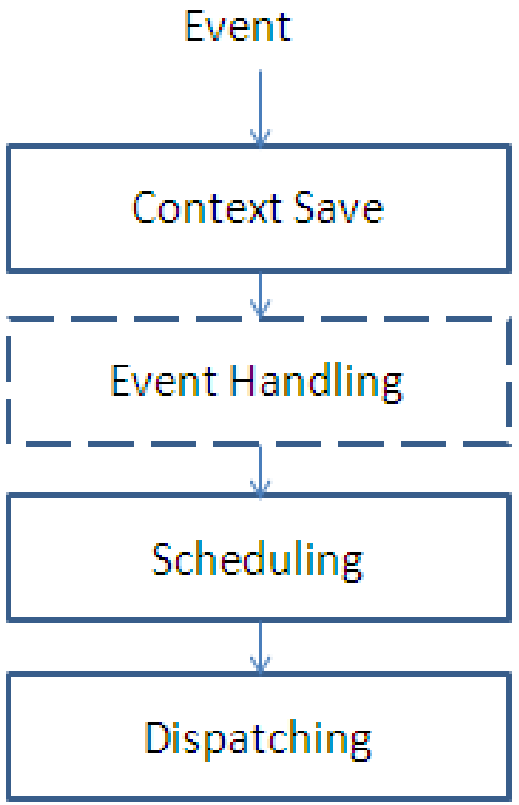
- Process context, including program counter and other registers is saved.
- Update the PCB of the process that was currently running state. The new state assigned may be one of the other states (Ready, Blocked, Ready/Suspend or Exit).
- Move this updated PCB to appropriate queue (Ready; Blocked on Event i; Ready/Suspend).
- Select another deserving process for execution.
- Update the PCB of chosen process and change the process state as Running.
- Update the memory management data structures.
- Restore the context of the chosen process so that it can resume from the point if it was interrupted last time, or can start its execution if it was loaded for the first time.

Process context switch Vs mode switch

- Context Switch:
  - Execution of a process is stopped to respond an interrupt.
  - Needs to save Process Image be saved of one process and load process image of the new process loaded.
  - The processes are switched and processes keep on changing their status as Running and Not running.
- Mode switch:
  - Every process may switch in between a low privileged user mode and high privileged kernel mode in its lifetime.
  - Process continues to execute even after mode switches

Fundamental kernel functions of process control

- Scheduling: Choose the process as per the scheduling policy to be executed next on the CPU.
- Dispatching: Set up execution of the chosen process on the CPU.
- Context save: Save information concerning an executing process when its execution gets suspended



Fundamental kernel functions of process control

- Occurrence of the event calls the context save functionality and an appropriate event handling procedure.
- Event handling may initiate some processes, hence the scheduling function gets invoked to choose the process and in turn,
- The dispatching function transfers control to the new process.

Control/Data structures maintained by OS to manage processes

- **Memory Tables**
- **I/O Tables**
- **File Tables**
- **Process table**

Control structures maintained by OS to manage processes

- **Memory Tables:**
  - Memory tables keep track of both main and secondary memory.
  - Active processes are stored in main memory and when required, they are moved to secondary memory through the mechanism called 'swapping'.
  - The memory tables maintain the following information:
    - –The main memory allocation to all processes in system
    - –The secondary memory allocation to all processes in system
    - –Shared memory regions in main and virtual memory and their attributes
    - –Miscellaneous information required to manage virtual memory.

Control structures maintained by OS to manage processes

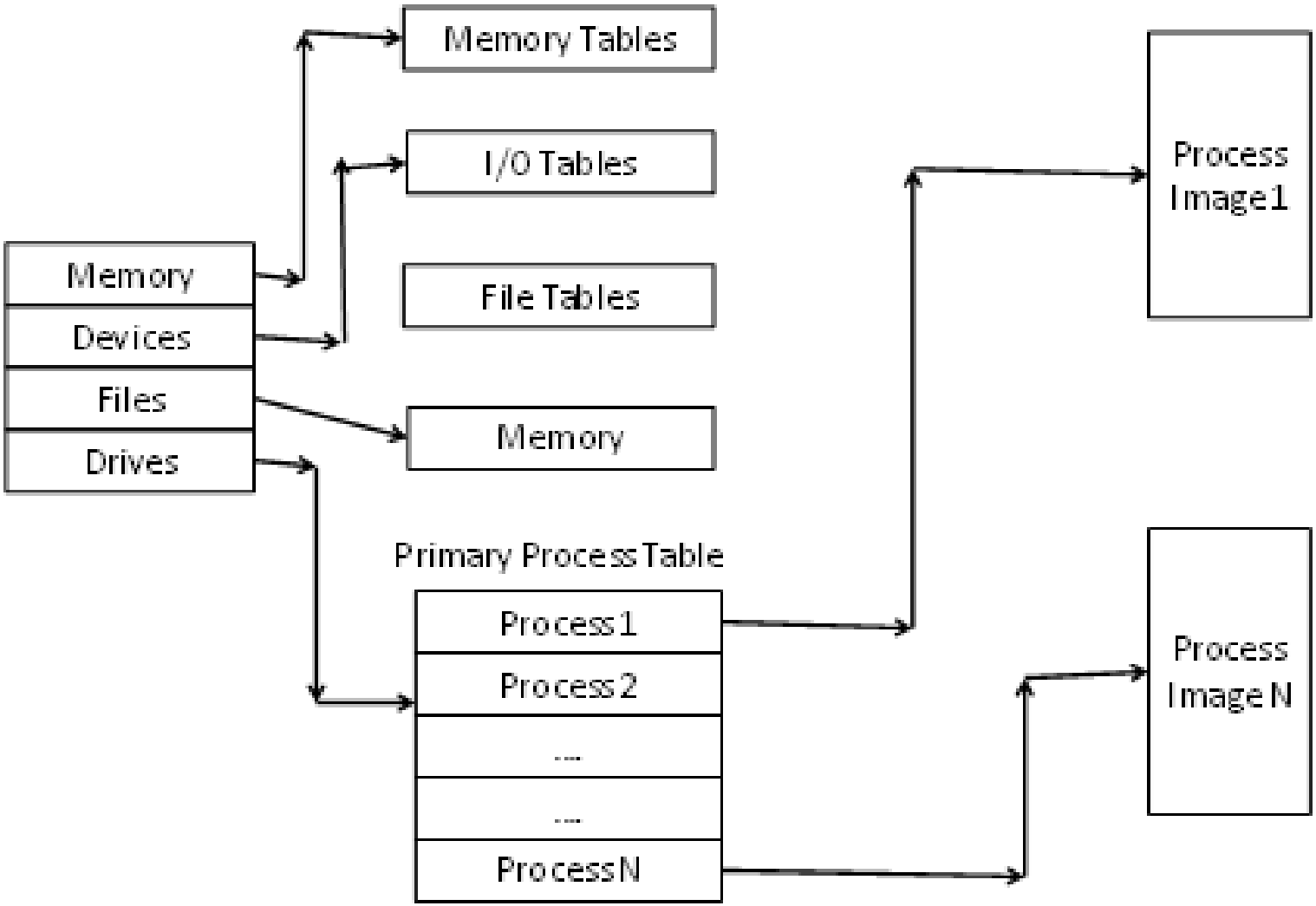
- **I/O Tables:**
  - I/O tables keep track of I/O devices and channels in the computing system.
  - The I/O devices are also resources required by processes.
  - So at any given instance, I/O devices may be available or allocated to a particular process.

Control structures maintained by OS to manage processes

- **File Tables:**
  - File tables keeps track of;
    - – all files,
    - –their locations on secondary memory,
    - –their current statuses and
    - –other attributes such security, sharing, etc.
  - Most of the operating systems, this information is maintained by a module called File Management System.

Control structures maintained by OS to manage processes

- **Process table:**
  - Process tables manage processes.
  - They maintain information of:
    - processes,
    - their child process references,
    - statuses,
    - allocated resources,
    - process contexts,
    - information required for process synchronization and so on.
  - These pieces of information are stored in process images.


Memory Tables
I/O Tables
File Tables
Memory
Memory
Devices
Files
Drives
Primary Process Table
Process 1
Process 2
...
...
Process N
Process Image 1
Process Image N

Different interaction mechanisms used by processes

Interaction Mechanism	Description
Data Sharing	The processes interact with each other by altering data values. If more than one processes update the data the same time, they may leave the shared in inconsistent state. So, shared data items are protected against simultaneous access to avoid such situation.
Message Passing	In this mechanism, the processes exchange information by sending messages to each other.
Synchronization	In certain computing environments, the processes are required to execute their actions in some particular order. To help this happen, the processes synchronize with each other to maintain their relative timings and execute in the desired sequence.
Signals	The processes may wait for events to occur. It can be intimated to processes through the signaling mechanism.

Process and thread

- Process: an instance of program in execution
- Thread : a dispatchable unit of work

Traditional process

- Single thread of execution
- Needs entire process context to execute so considered as heavyweight
- Doesn’t support multiple parallel executions

E.g. you wouldn’t get notifications in background if you are using the app

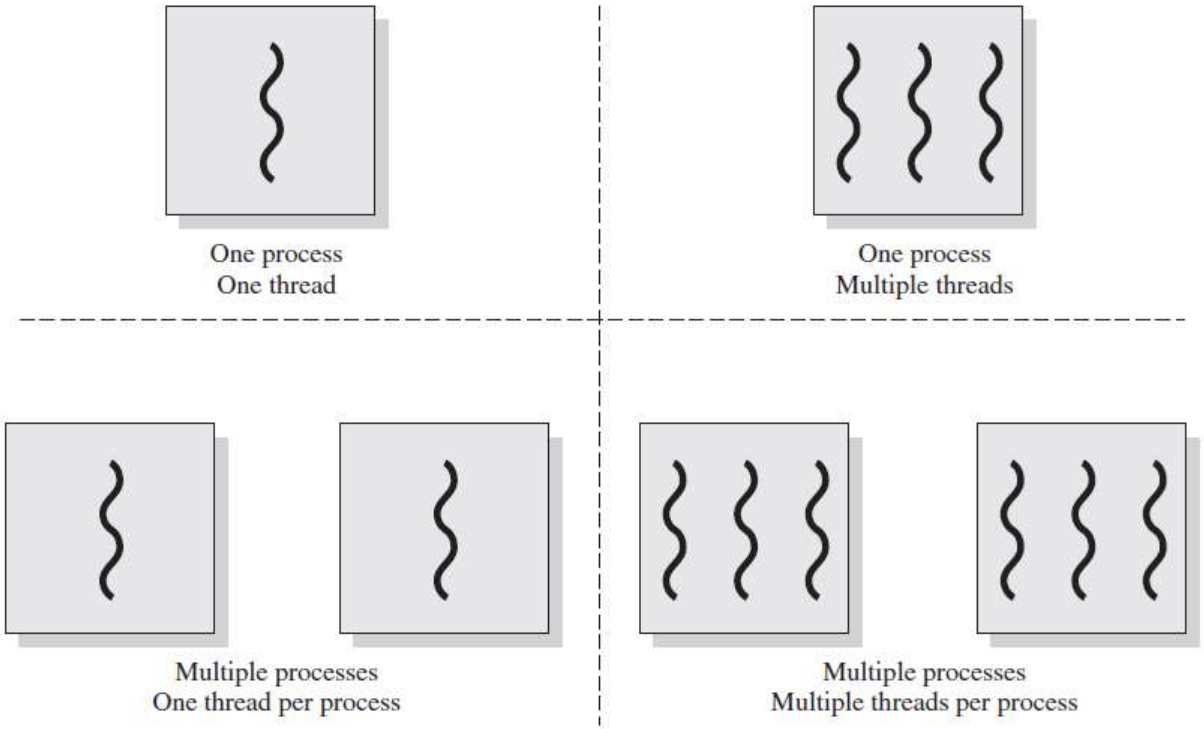
Thread

- Supports Parnellism with multiple threads of execution at a time
- A thread executes sequentially and is interruptable so that the processor can turn to another thread
- Does not need entire process context to execute so considered as lightweight
- Includes the program counter and stack pointer) and its own data area for a stack
- Supports multiple parallel executions

e.g. Notifications in background while you are using the app

- The idea is to **achieve parallelism by dividing a process into multiple threads.**

- *Multithreading* refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.



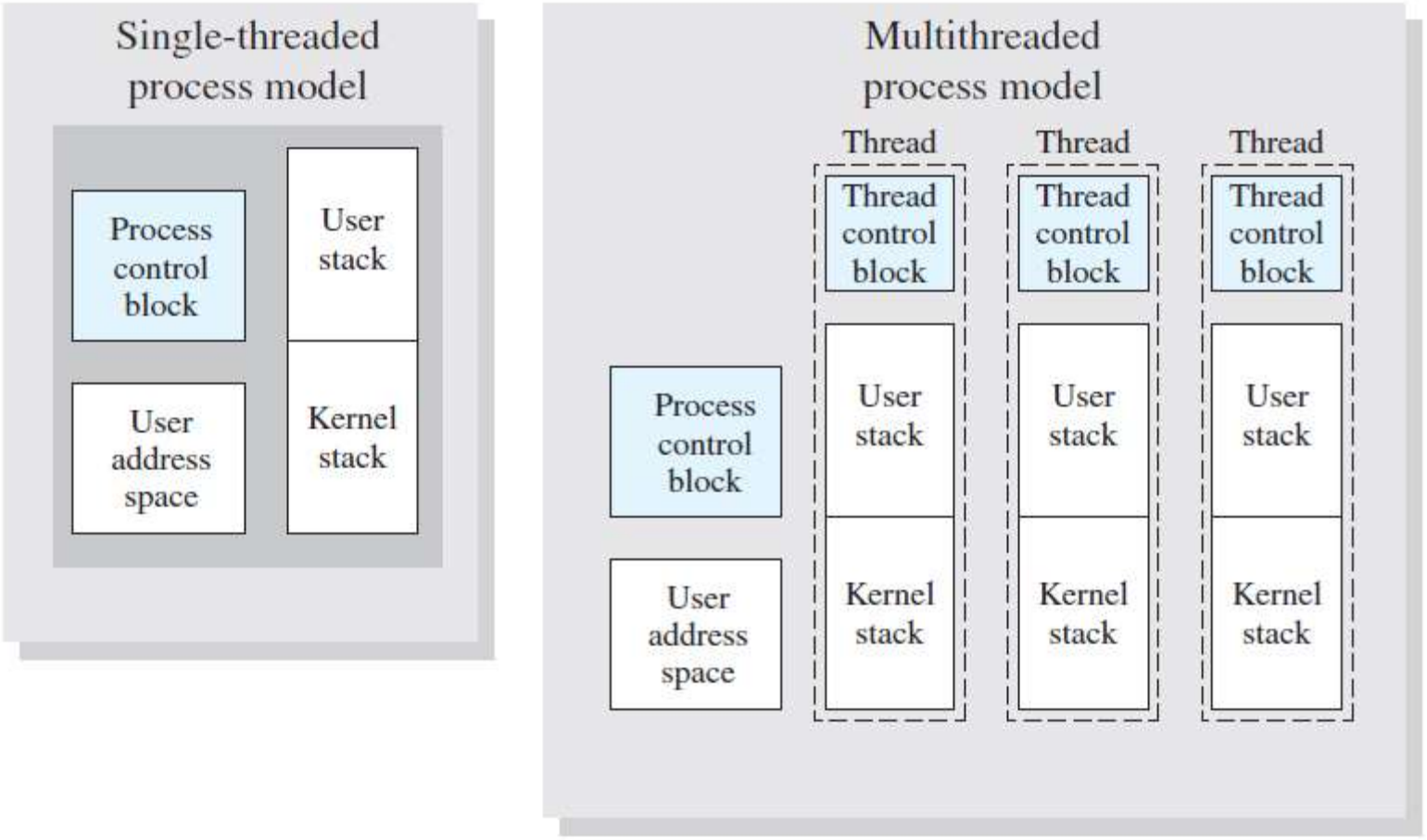
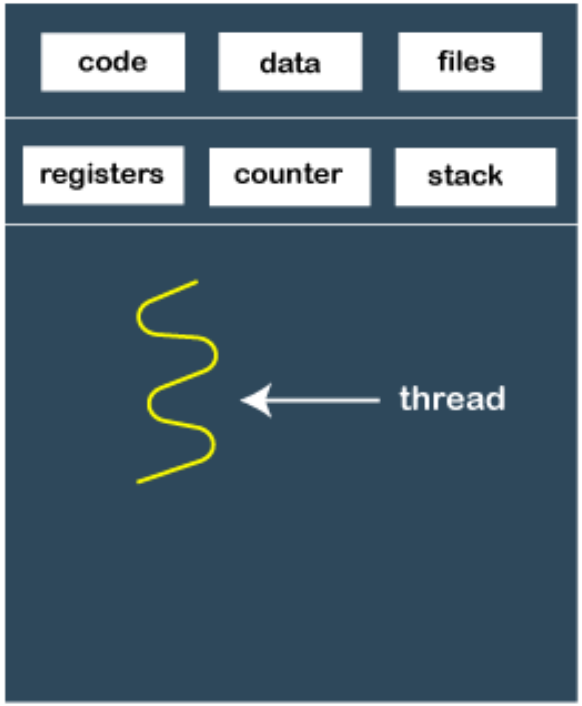
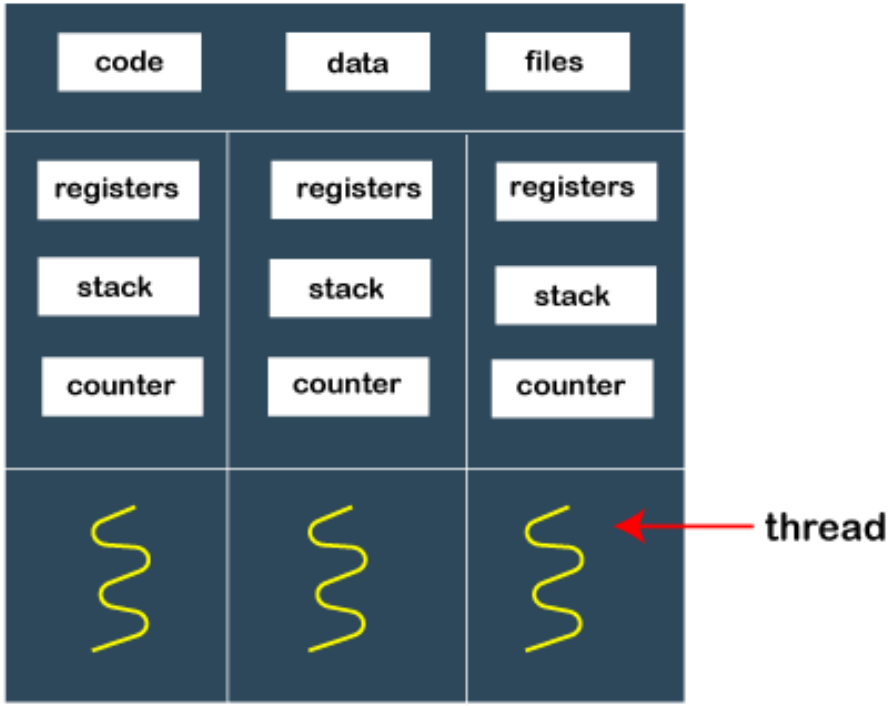


Figure 4.2 **Single Threaded and Multithreaded Process Models**



Single-threaded process



Multi-threaded process

Image courtesy : https://www.javatpoint.com/process-vs-thread

The key benefits of threads

- It takes far less time to create a new thread in an existing process than to create a brand-new process.
- It takes less time to terminate a thread than a process.

Types of threads

- Kernel level threads : managed by kernel
- User level threads : managed by user with support from programming languages and libraries
- Hybrid threads

Thread

- Create
- Join
- Terminate

Threads

```
import threading
import time

# Define a function for the thread to execute
def count_numbers(thread_name, count_to):
    for i in range(1, count_to + 1):
        print(f"{thread_name} counting: {i}")
        time.sleep(0.5)  # Sleep for half a second

# Create threads
thread1 = threading.Thread(target=count_numbers, args=("Thread 1", 5))
thread2 = threading.Thread(target=count_numbers, args=("Thread 2", 5))

# Start threads
print("Starting threads")
thread1.start()
thread2.start()

# Join threads (wait for them to complete)
thread1.join()
thread2.join()

print("Threads have completed their tasks")
```

```
PS C:\Users\Swati> python -u "c:\Users\Swati\Downloads\import
threading.py"
Starting threads
Thread 1 counting: 1
Thread 2 counting: 1
Thread 2 counting: 2
Thread 1 counting: 2
Thread 1 counting: 3
Thread 2 counting: 3
Thread 2 counting: 4
Thread 1 counting: 4
Thread 2 counting: 5
Thread 1 counting: 5
Threads have completed their tasks
```

ULT Vs KLT

Sr No	System Call	Description
1	fork()	This system calls creates a new process.
2	exec()	This call is used to execute a new program on a process.
3	wait()	This call makes a process wait until some event occurs.
4	exit()	This call makes a process to terminate
5	getpid()	This system call helps to get the identifier associated with the process.
6	getppid()	This system call helps to get the identifier associated with the parent process.
7	nice()	The current process priority can be changed with execution of this system call.
8	brk()	This call helps to increase or decrease the data segment size of the process.
9	Kill()	The forced termination of any process can be executed with this system call.
10	Signal()	This system call is invoked for sending and receiving software interrupts

fork system call

- The **fork()** system call is used to create a new process. When the system executes this system call in response to process creation request, following steps are carried out.
- The operating system allocates a slot in the process table for the newly created process.
- This new process, i.e. child process is then assigned a unique ID.
- System then creates a logical copy of the parent process context.
- The file and inode table counters associated with parent process are incremented as this area may be shared between parent process and the child process.
- The child process ID is returned to the process and ‘0’ value is assigned to the child process.
- All the above steps are carried out in kernel of the parent process.
- The control of execution may remain with the parent process, may be transferred to child process or handed over to a third process leaving the parent and child processes in ‘Ready-to-Run’ state.

Processor Scheduling

I/O scheduling Vs Processor Scheduling

- **I/O Scheduling:** I/O scheduling is a process that is involved in making the decision as to which process's pending I/O request should be handled by an available I/O device.

- **Processor Scheduling:** Processor scheduling is a process that makes a decision of which process should get hold of processor next. This process needs the 'dispatcher' – a software module of short term scheduler to make this decision.

- **I/O bound process:** I/O bound processes are the ones those spend more time doing I/O operations than computation, though it may have many short CPU bursts.
-
- **CPU bound processes:** CPU bound processes spend more time in computations and so they have long CPU bursts. Although they may also involve in very short durations of I/O operations.

Parallelism and Concurrency

- **Parallelism:**
  - Parallelism is the quality of the processes to execute at the same time.
  - Two processes or events are said to be parallel if they occur at the same time. In parallelism, multiple processes can be active at a time.
- **Concurrency:**
  - Concurrency does not mean parallel.
  - With concurrency, multiple processes are executed one after another by interleaving their execution in such a way that it creates an illusion of parallelism.
  - In concurrency only one process can be active at a time.

Parallelism and Concurrency

- Concurrency is achieved by interleaving process execution on (may be single) CPU which creates an illusion that processes run in parallel,
- While parallelism is obtained by multiple processors operating in parallel at a time.
- Both the techniques achieve computation speedup, though the inherent mechanism used by both concepts is different.

Advantages of process concurrency.

- The processes in a multiprogramming environment are a blend of I/O bound and CPU bound processes.
- If these processes are executed sequentially, they underutilize the CPU and I/O devices both.
- If executed in interleaved fashion, the system gives better efficiency, relatively lower response, turn around and waiting times.

Process Scheduler

- Process scheduler or the Dispatcher is a software module that works for CPU(processor) scheduling and chooses the next process to run.
- Its main activities involve switching the process context, switching the execution mode to user.
- The dispatcher is required to work very fast to improve the efficiency of execution.

Processor schedulers

- Long term scheduler
- Mid term scheduler
- Short term scheduler

System Calls for Process Management

Questions
?