

Unit II

Process and Thread Management

Presented By

Dr. Khursheed Ahmad Bhat

Assistant Professor

School of Computer Science

UPES



Lecture Outline

- Process Concept
- Process States
- PCB



Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution;
- Program is **passive** entity stored on disk; process is **active**
 - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- Processes can be CPU-bound or I/O bound in nature.
- Execution vs I/O

Process Management

Process

- ① Process is created by operating system in order to execute a program.

a program in execution

an instance of a running program

the entity that can be assigned to, and executed on, a processor

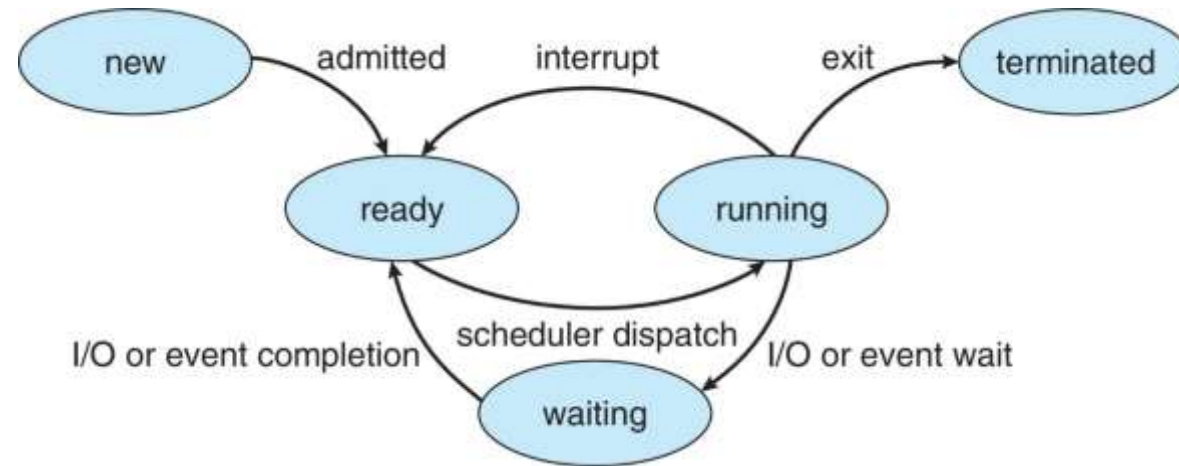
a unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources



Process State

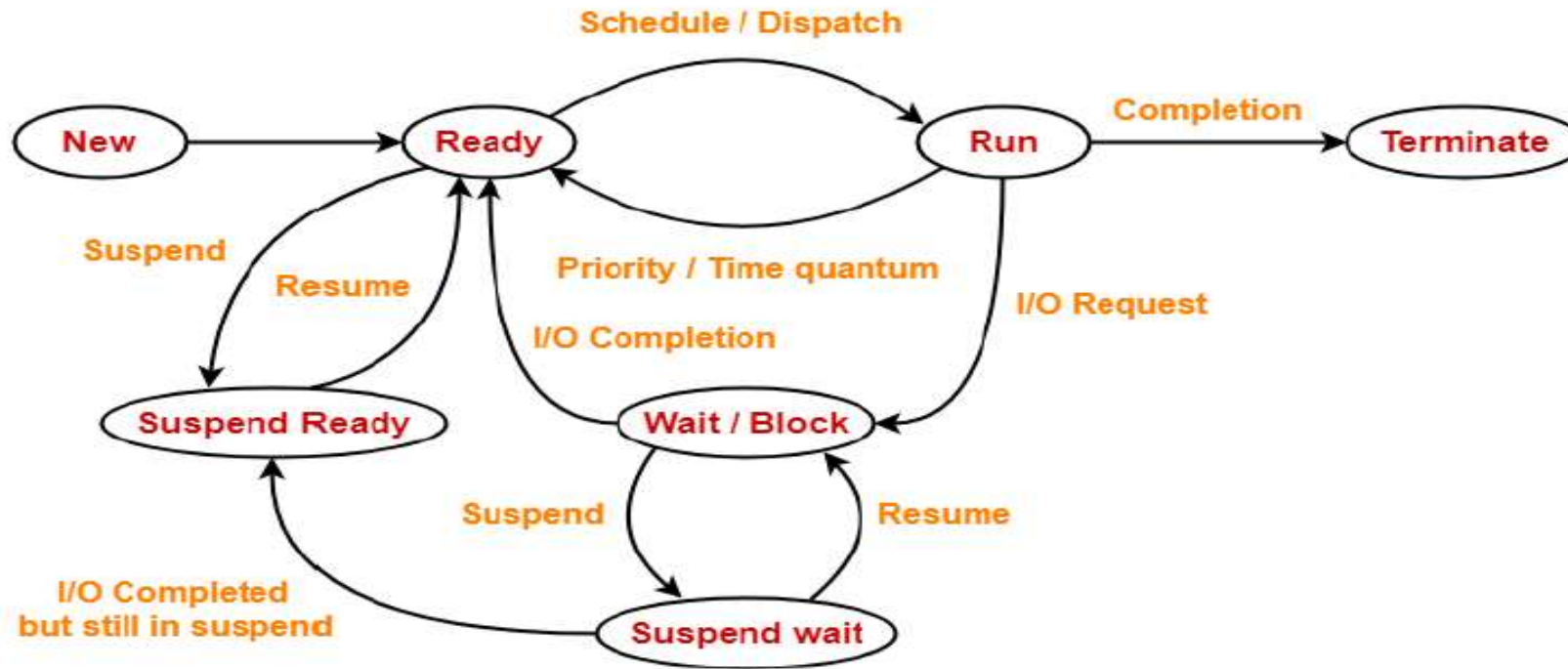
- As a process executes, it changes **state**
 - **New**: The process is being created
 - **Running**: Instructions are being executed
 - **Waiting**: The process is waiting for some event to occur
 - **Ready**: The process is waiting to be assigned to a processor
 - **Terminated**: The process has finished execution

Diagram of Process State



Case Study: Process Management in Linux

Process States



Process State Diagram

- **Long-Term Scheduler:** New → Ready (controls admission from job pool).
- **Short-Term Scheduler:** Ready → Running (decides CPU execution).
- **Medium-Term Scheduler:** Moves processes to/from Suspended states.



1. New State-

- A process is said to be in new state when a program present in the secondary memory is initiated for execution.

2. Ready State-

- A process moves from new state to ready state after it is loaded into the main memory and is ready for execution.
- In ready state, the process waits for its execution by the processor.
- In multiprogramming environment, many processes may be present in the ready state.

3. Run State-

- A process moves from ready state to run state after it is assigned the CPU for execution.

4. Terminate State-

- A process moves from run state to terminate state after its execution is completed.
- After entering the terminate state, context (PCB) of the process is deleted by the operating system.

5. Block Or Wait State-

- A process moves from run state to block or wait state if it requires an I/O operation or some blocked resource during its execution.
- After the I/O operation gets completed or resource becomes available, the process moves to the ready state.

6. Suspend Ready State-

- A process moves from ready state to suspend ready state if a process with higher priority has to be executed but the main memory is full.
- Moving a process with lower priority from ready state to suspend ready state creates a room for higher priority process in the ready state.
- The process remains in the suspend ready state until the main memory becomes available.
- When main memory becomes available, the process is brought back to the ready state.

6. Suspend Wait State-

- A process moves from wait state to suspend wait state if a process with higher priority has to be executed but the main memory is full.
- Moving a process with lower priority from wait state to suspend wait state creates a room for higher priority process in the ready state.
- After the resource becomes available, the process is moved to the suspend ready state.
- After main memory becomes available, the process is moved to the ready state.

Note-01:

A process necessarily goes through minimum 4 states.

- The minimum number of states through which a process compulsorily goes through is 4.
- These states are new state, ready state, run state and terminate state.
- However, if a process also requires the I/O operation, then minimum number of states is 5.

Note-02:

A single processor can execute only one process at a time.

- A single processor can not more than one processes simultaneously.
- If n processors are present in the system, then only n processes can be executed simultaneously.

Operations on Process

1. Creation

- System initialization
- Execution of a process creation system call by a running process
- A user request to create a new process
- Initiation of a batch job

2. Scheduling

- Long term scheduling (Degree of Multiprogramming)
 - Short term scheduling (Dispatcher)
 - Medium term scheduling (Moves processes to/from Suspended states).
- Note: Number of processes that are present in the ready queue at maximum is called Degree of Multiprogramming.

Operations on Process

3. Execution

- Once a process is given to CPU, it will start executing
- Arrival time, Burst time, Waiting time, Response time

4. Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (Involuntary)
- Killed by another process (Involuntary)

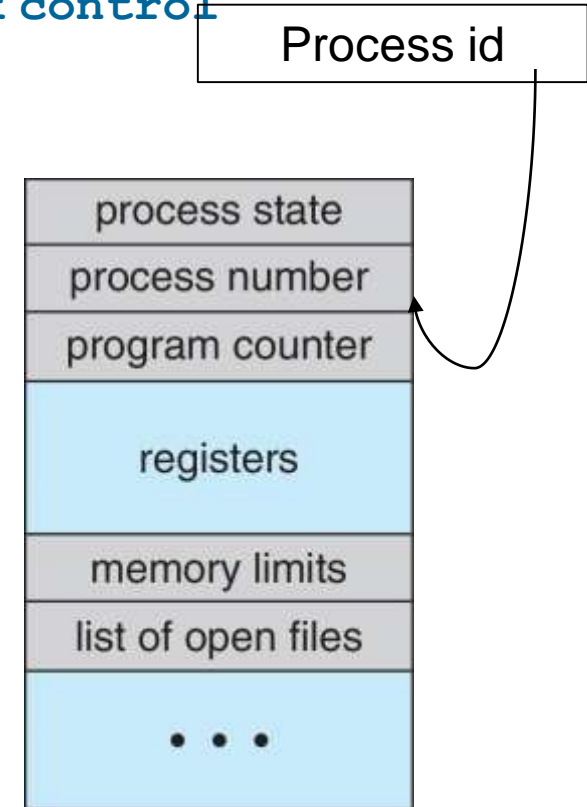
OS provides process abstraction

- When you run an exe file, the OS creates a process = a running program
- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU
 - Policy: which process to run
 - Mechanism: how to “context switch” between processes

Process Control Block (PCB)

Information associated with each process(also called **task control block**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



OS data structures

- OS maintains a data structure (e.g., list) of all active processes
- Information about each process is stored in a process control block (PCB)
 - Process identifier
 - Process state
 - Pointers to other related processes (parent)
 - CPU context of the process (saved when the process is suspended)
 - Pointers to memory locations
 - Pointers to open files

Data Structure

- A data structure is a way of organizing and storing data in a computer so that it can be used efficiently.
- A **data structure** is a specialized format for **organizing, storing, and managing data** in a computer so that it can be **accessed, modified, and processed efficiently**.
- Integers, floats, characters, pointers, etc
- Arrays, Linked Lists, Stacks, Queues
- Trees, Graphs

Process Control Block

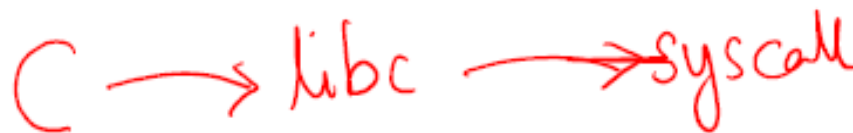
- Process Control Block (PCB) is a data structure that stores information about a particular process.
- This information is required by the CPU while executing the process.
- Each process is identified by its own process control block (PCB).
- It is also called as **context** of the process.

Operating System API

- API = Application Programming Interface
= functions available to write user programs
- API provided by OS is a set of “system calls”
 - System call is a function call into OS code that runs at a higher privilege level of the CPU
 - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
 - Some “blocking” system calls cause the process to be blocked and descheduled (e.g., read from disk)

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
 - Programs written to the POSIX API can run on any POSIX compliant OS
 - Most modern OSes are POSIX compliant
 - Ensures program portability
- Program language libraries hide the details of invoking system calls
 - The printf function in the C library calls the write system call to write to screen
 - User programs usually do not need to worry about invoking system calls



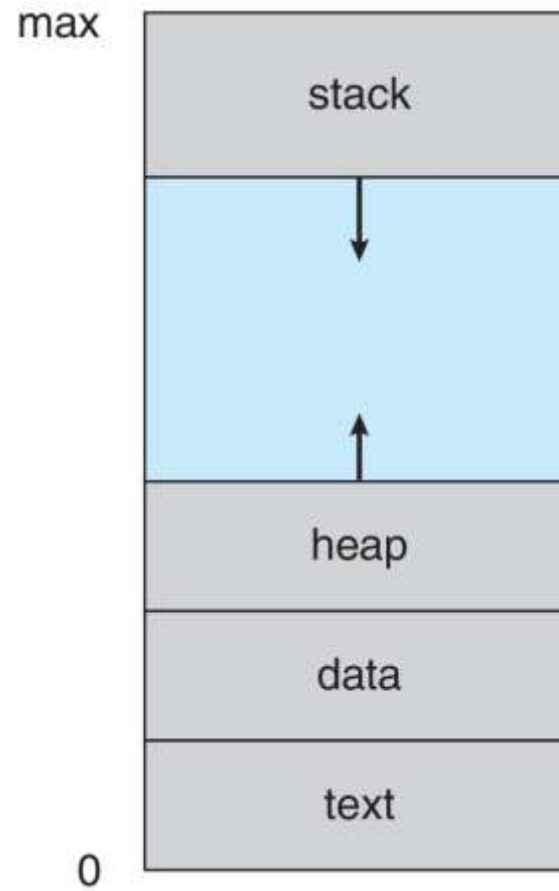
Lecture Outline

- Process related system calls in Unix
- CPU-bound vs I/O-bound Processes
- Memory layout of a C program
- Context Switch
- CPU Scheduling

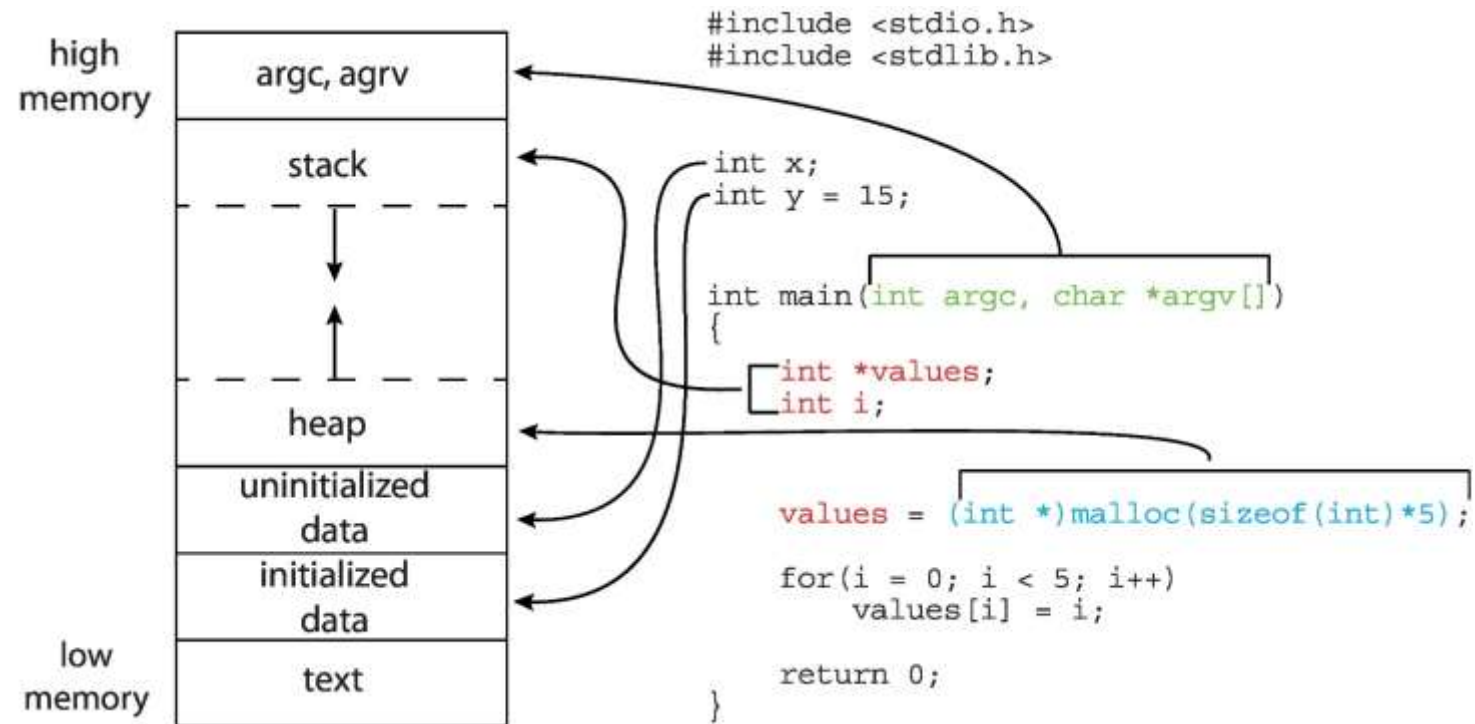
Process related system calls (in Unix)

- fork() creates a new child process
 - All processes are created by forking from a parent
 - The init process is ancestor of all processes
- exec() makes a process execute a given executable
- exit() terminates a process
- wait() causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

Process in Memory

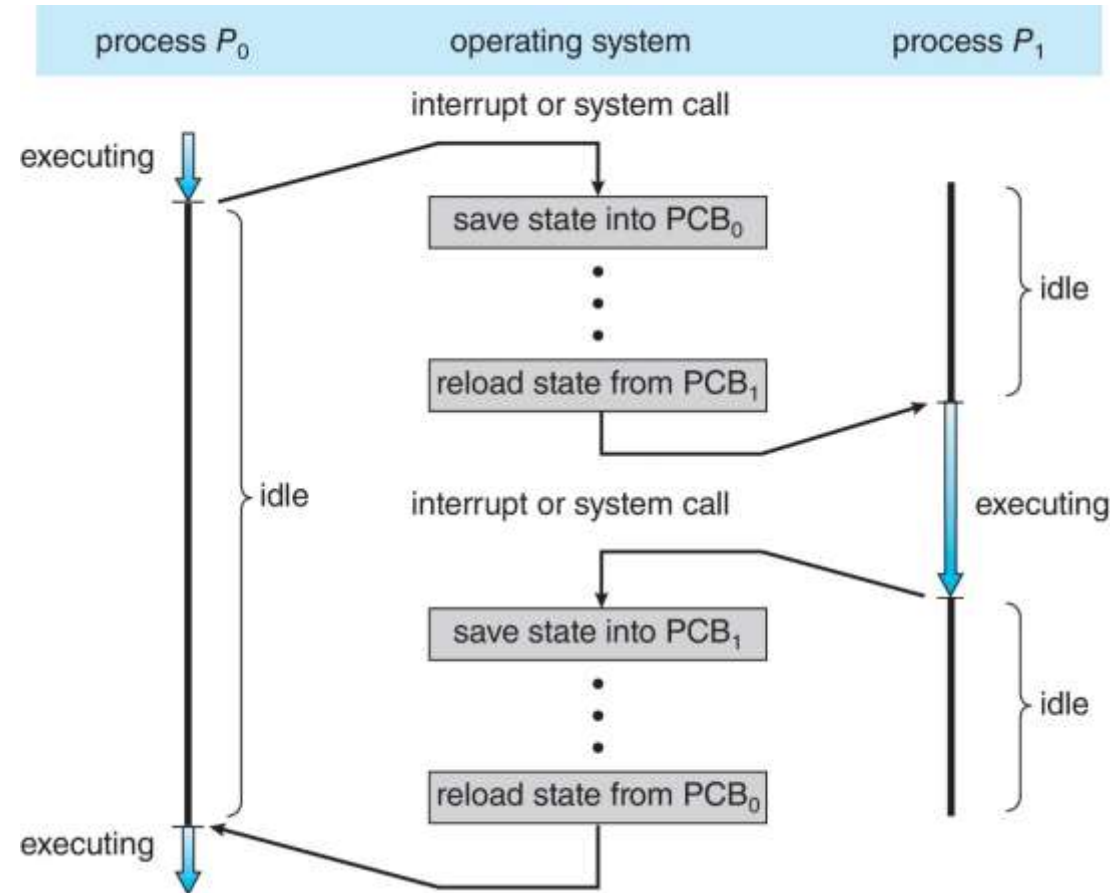


Memory Layout of a C Program



CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
 - Single **foreground** process- controlled via user interface
 - Multiple **background** processes— in memory, running, but not on the display, and with limits
 - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
 - Background process uses a **service** to perform tasks
 - Service can keep running even if background process is suspended
 - Service has no user interface, small memory use

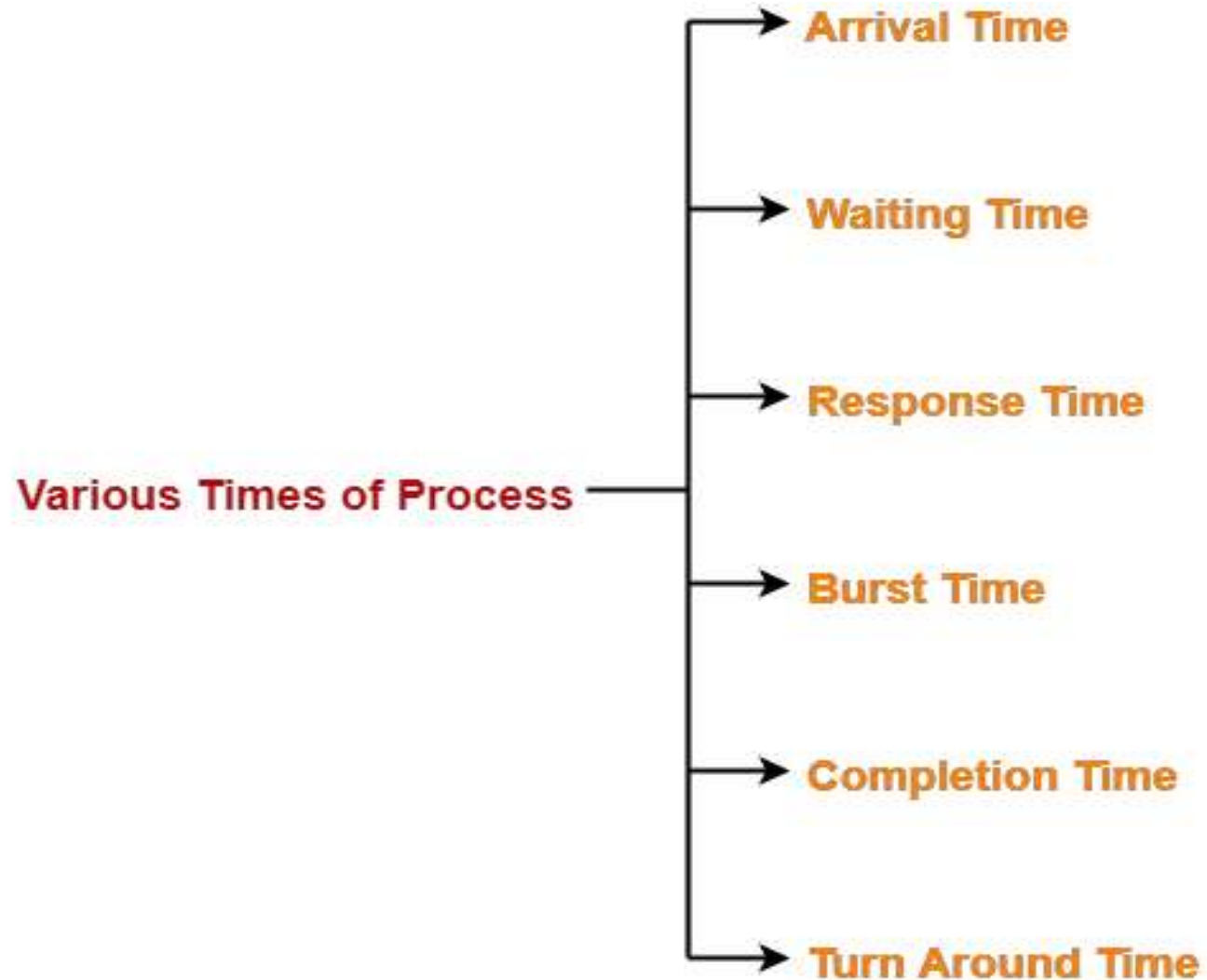
What is a scheduling policy?

- On context switch, which process to run next, from set of ready processes?
- OS scheduler schedules the CPU requests (bursts) of processes
 - CPU burst = the CPU time used by a process in a continuous stretch
 - If a process comes back after I/O wait, it counts as a fresh CPU burst

What are we trying to optimize?

- Maximize (utilization = fraction of time CPU is used)
- Minimize average (turnaround time = time from process arrival to completion)
- Minimize average (response time = time from process arrival to first scheduling)
- Fairness: all processes must be treated equally
- Minimize overhead: run process long enough to amortize cost of context switch (~1 microsecond)

Various **Times** related to Processes



Various **Times** related to Processes

❖ Turn-around Time:

- i. The difference between “Completion time” and “Arrival time”.
- ii. In between arrival time and completion time, process will be either “waiting or executing”, assuming there is no I/O request.
- iii. Turn around time = $CT - AT = \text{Burst Time} + \text{Waiting Time}$

❖ Waiting Time:

$$\text{Waiting Time} = \text{Turn around Time} - \text{Burst Time}$$

❖ Response Time:

The time at which the process hits the CPU first time, it arrives.

$$\text{Response Time} = \text{First Time} - \text{Arrival Time}$$

CPU Scheduling

- **First Come First Serve (FCFS):**

- The process which arrives first in the ready queue is firstly assigned the CPU.
- In case of a tie, process with smaller process id is executed first.
- It is always non-preemptive in nature

- **Advantages-**

- It is simple and easy to understand.
- It can be easily implemented using queue data structure.

- **Disadvantages-**

- It does not consider the priority or burst time of the processes.
- It suffers from **convoy effect**

1

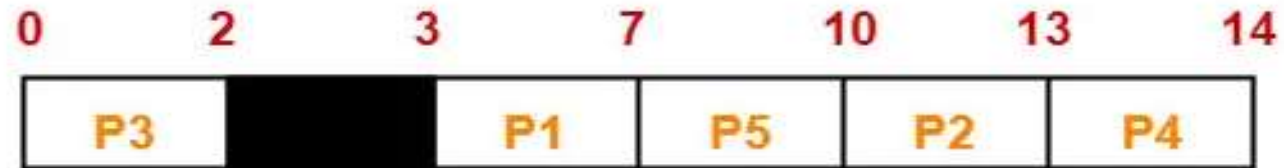
Process No	Arrival Time	Burst Time
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

2

Process No	Arrival Time	Burst Time
P1	0	4
P2	1	3
P3	2	1
P4	3	2
P5	4	5

Solution to Problem 1

Gantt Chart-



Gantt Chart

CT			
Process Id	Exit time	Turn Around time	Waiting time
P1	7	$7 - 3 = 4$	$4 - 4 = 0$
P2	13	$13 - 5 = 8$	$8 - 3 = 5$
P3	2	$2 - 0 = 2$	$2 - 2 = 0$
P4	14	$14 - 5 = 9$	$9 - 1 = 8$
P5	10	$10 - 4 = 6$	$6 - 3 = 3$

Longest Job First Algorithm-

In LJF Scheduling,

- Out of all the available processes, CPU is assigned to the process having largest burst time.
- In case of a tie, it is broken by **FCFS Scheduling**.



- LJF Scheduling can be used in both preemptive and non-preemptive mode.
- Preemptive mode of Longest Job First is called as **Longest Remaining Time First (LRTF)**.

Problem-01:

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	3
P2	1	2
P3	2	4
P4	3	5
P5	4	6

If the CPU scheduling policy is LJF non-preemptive, calculate the average waiting time and average turn around time.



Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	3	$3 - 0 = 3$	$3 - 3 = 0$
P2	20	$20 - 1 = 19$	$19 - 2 = 17$
P3	18	$18 - 2 = 16$	$16 - 4 = 12$
P4	8	$8 - 3 = 5$	$5 - 5 = 0$
P5	14	$14 - 4 = 10$	$10 - 6 = 4$

Now,

- Average Turn Around time = $(3 + 19 + 16 + 5 + 10) / 5 = 53 / 5 = 10.6$ unit
- Average waiting time = $(0 + 17 + 12 + 0 + 4) / 5 = 33 / 5 = 6.6$ unit

Problem-02:

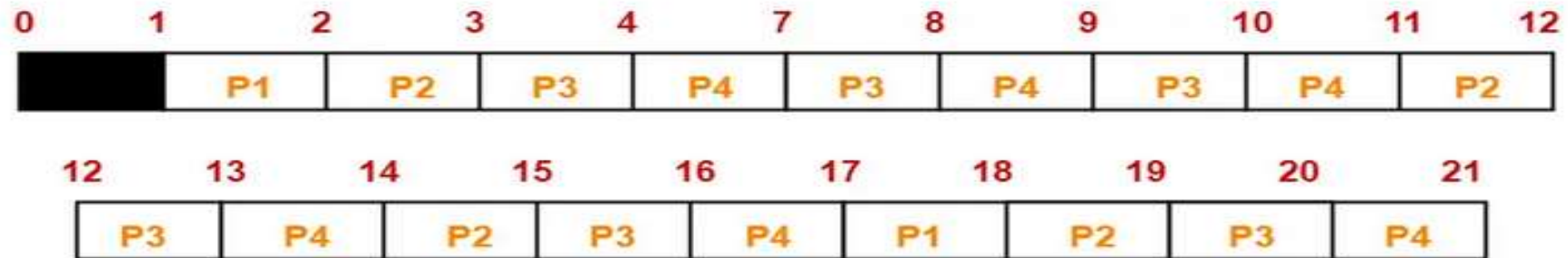
Consider the set of 4 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	1	2
P2	2	4
P3	3	6
P4	4	8

If the CPU scheduling policy is LJF preemptive, calculate the average waiting time and average turn around time.

Solution-

Gantt Chart-



Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	18	$18 - 1 = 17$	$17 - 2 = 15$
P2	19	$19 - 2 = 17$	$17 - 4 = 13$
P3	20	$20 - 3 = 17$	$17 - 6 = 11$
P4	21	$21 - 4 = 17$	$17 - 8 = 9$

- Average Turn Around time = $(17 + 17 + 17 + 17) / 4 = 68 / 4 = 17$ unit
- Average waiting time = $(15 + 13 + 11 + 9) / 4 = 48 / 4 = 12$ unit

Problem-03:

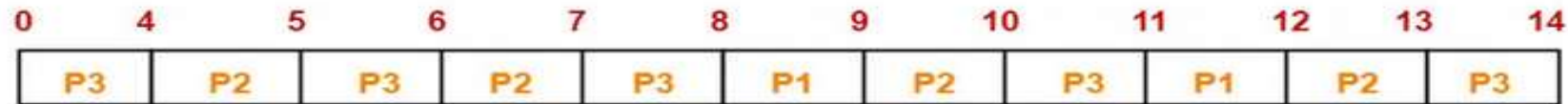
Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (LRTF) scheduling algorithm. In LRTF, ties are broken by giving priority to the process with the lowest process id. The average turn around time is-

1. 13 unit
2. 14 unit
3. 15 unit
4. 16 unit

We have the set of 3 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	2
P2	0	4
P3	0	8

Gantt Chart-



Gantt Chart

Process Id	Exit time	Turn Around time
P1	12	$12 - 0 = 12$
P2	13	$13 - 0 = 13$
P3	24	$14 - 0 = 14$

Now,

$$\text{Average Turn Around time} = (12 + 13 + 14) / 3 = 39 / 3 = 13 \text{ unit}$$

Priority Scheduling:-

In Priority Scheduling,

- Out of all the available processes, CPU is assigned to the process having the highest priority.
- In case of a tie, it is broken by FCFS Scheduling.



- Priority Scheduling can be used in both preemptive and non-preemptive mode.

Advantages-

- It considers the priority of the processes and allows the important processes to run first.
- Priority scheduling in preemptive mode is best suited for real time operating system.

Disadvantages-

- Processes with lesser priority may starve for CPU.
- There is no idea of response time and waiting time.

Note-01:

- The waiting time for the process having the highest priority will always be zero in preemptive mode.
- The waiting time for the process having the highest priority may not be zero in non-preemptive mode.

Note-02:

Priority scheduling in preemptive and non-preemptive mode behaves exactly same under following conditions-

- The arrival time of all the processes is same
- All the processes become available

Problem-01:

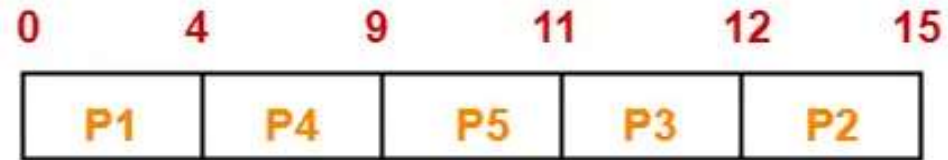
Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

Solution-

Gantt Chart-



Gantt Chart

Process Id	Exit time	Turn Around time	Waiting time
P1	4	$4 - 0 = 4$	$4 - 4 = 0$
P2	15	$15 - 1 = 14$	$14 - 3 = 11$
P3	12	$12 - 2 = 10$	$10 - 1 = 9$
P4	9	$9 - 3 = 6$	$6 - 5 = 1$
P5	11	$11 - 4 = 7$	$7 - 2 = 5$

Now,

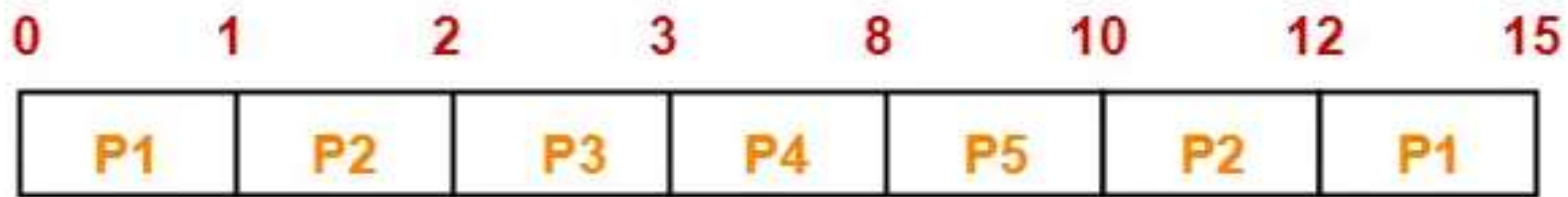
- Average Turn Around time = $(4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2$ unit
- Average waiting time = $(0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2$ unit

Problem-02:

Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time	Priority
P1	0	4	2
P2	1	3	3
P3	2	1	4
P4	3	5	5
P5	4	2	5

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)



Gantt Chart

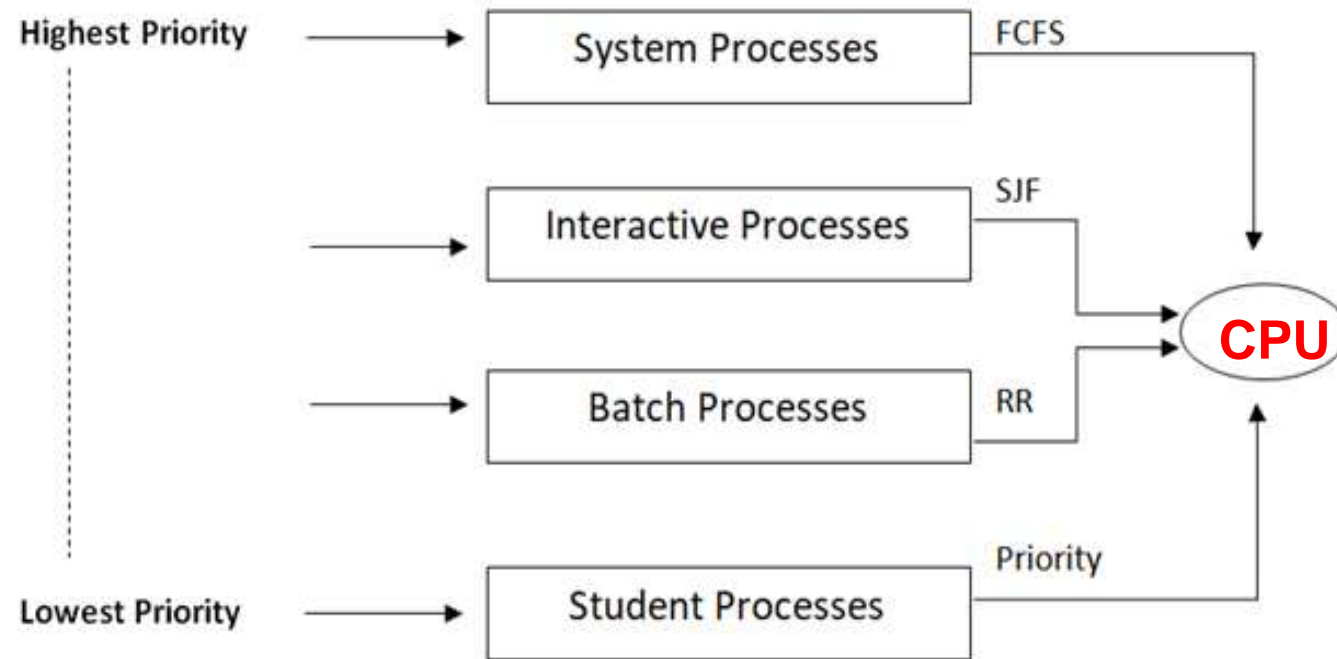
Process Id	Exit time	Turn Around time	Waiting time
P1	15	$15 - 0 = 15$	$15 - 4 = 11$
P2	12	$12 - 1 = 11$	$11 - 3 = 8$
P3	3	$3 - 2 = 1$	$1 - 1 = 0$
P4	8	$8 - 3 = 5$	$5 - 5 = 0$
P5	10	$10 - 4 = 6$	$6 - 2 = 4$

Now,

- Average Turn Around time = $(15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6$ unit
- Average waiting time = $(11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6$ unit

Multilevel Queue

- Having one Queue and scheduling all the processes is difficult, in the sense that there are various processes in a computer.
- Processes are permanently assigned to one queue, based on some property of the process, such as memory size, priority, type etc.
 - i. System processes
 - ii. Interactive processes (Online Games) (Foreground processes)
 - iii. Batch processes (generally Background processes)
 - iv. Student processes (Normal programs)



Advantages: Different type of process has different scheduling algorithm, as per requirement.

Disadvantages: Lowest priority process gets starvation for the higher priority process because here priority is static.

Process	Arrival Time	CPU Burst Time	Queue Number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	4

Queue 1 has a higher priority than queue **2**. Round Robin is used in queue **1 (Time Quantum = 2)**, while FCFS is used in queue **2**.



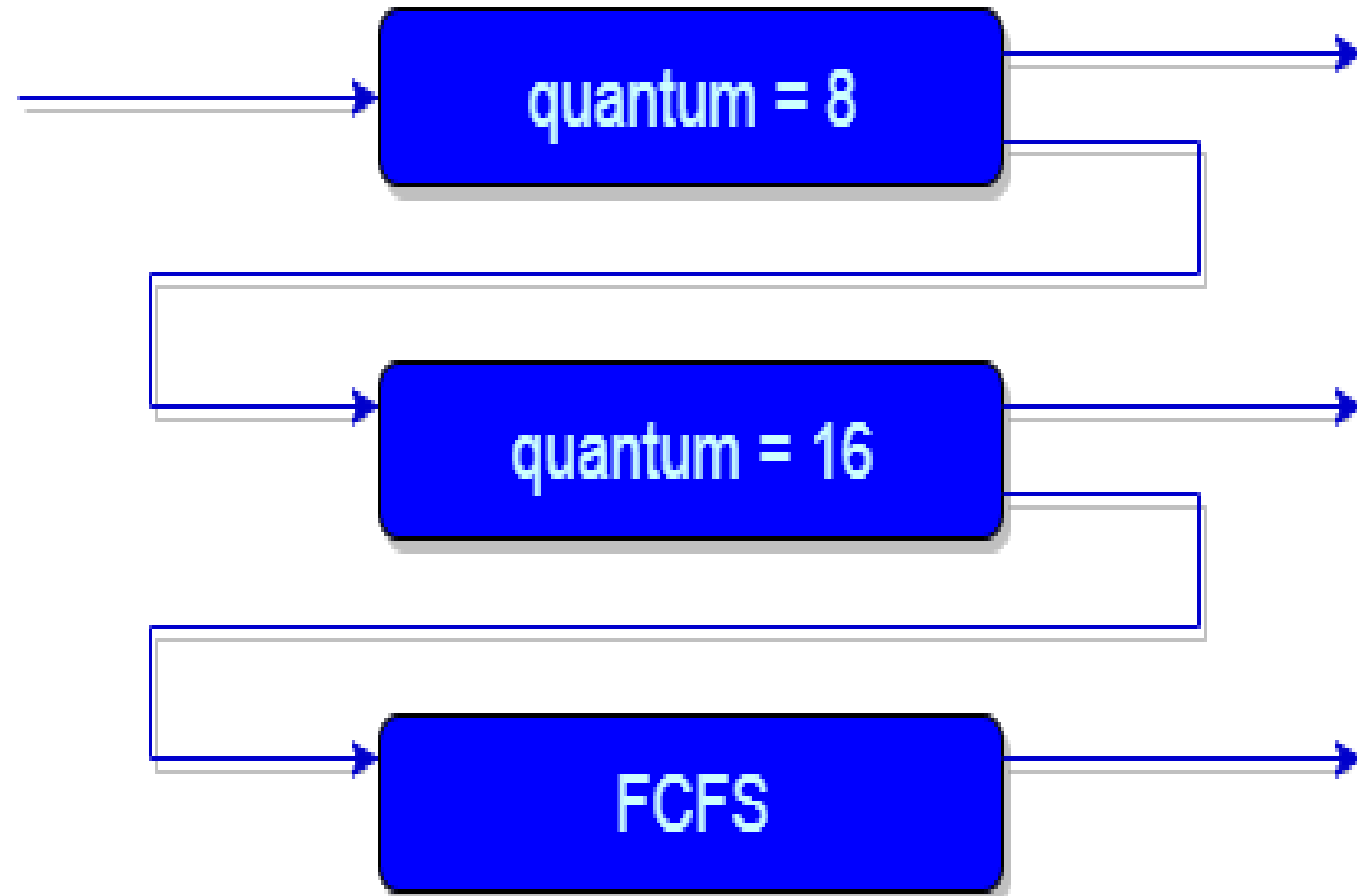
Working:

1. Both queues have been processed at the start. Therefore, **queue 1 (P1, P2)** runs first (due to greater priority) in a round-robin way and finishes after 7 units.
2. The process in **queue 2 (Process P3)** starts running (since there is no process in queue 1), but while it is executing, P4 enters queue 1 and interrupts P3, and then P3 takes the CPU and finishes its execution.

Multilevel Feedback Scheduling

- This strategy prioritizes operations that require I/O and are interactive.
- It enables a process to switch between queues.
- If a process consumes too much processor time, it will be switched to the lowest priority queue.
- A process waiting in a lower priority queue for too long may be shifted to a higher priority queue. This type of aging prevents starvation.

Multilevel Feedback Scheduling



Parameters for multilevel feedback scheduling

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher priority queue.
- The method used to determine when to demote a process to a lower priority queue.
- The method used to determine which queue a process will enter when that process needs service.

Guaranteed Scheduling

If n users are logged in while you are working, you will receive about $1/n$ of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get $1/n$ of the CPU cycles. That seems fair enough.

Guaranteed Scheduling

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by n . Since the amount of CPU time each process has actually had is also known, it is fairly straightforward to compute the ratio of actual CPU time consumed to CPU time entitled.

A ratio of 0.5 means a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above that of its closest competitor. Then that one is chosen to run next.

Guaranteed Scheduling

For example, suppose we have processes A, B, and C that so far have run for 2s, 3s, and 1s. (Assume we haven't yet put our algorithm into practice.) The total CPU time thus far is 6s. Split among three processes, this means each process should've ideally used 2s of CPU time. Clearly, that's not the case—B has been given preferential treatment. We'll allow C to run first until it hits 3s and then run A for an additional 1s to balance things out.

Fair Share Scheduling

Generally in multitasking operating system CPU scheduling policies is to provide equitable service to processes, rather than to user or their applications.

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up nine processes and user 2 starts up one process, with round-robin or equal priorities, user 1 will get 90% of the CPU and user 2 only 10% of it.

To prevent this situation, some systems take into account which user owns a process (or group of processes) before scheduling it. In this model, each user (or group of processes owned by this user) is allocated some fraction of CPU, this is called a **fair share**.

Thus, it would be attractive to make scheduling decisions on the basis of these groups of process sets (User's processes). This approach is generally known as fair share scheduling.

Consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, A, B, C, and D, and user 2 has only one process, E. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Numerous other possibilities exist, of course, and can be exploited, depending on what the notion of fairness is.

Scheduling is done on the basis of priority, which takes into account the underlying priority of the process, its recent processor usage, and the recent processor usage of the group to which the process belongs. The higher the numerical value of priority, the lower priority.

For example, if four users (A,B,C,D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole ($100\% / 4 = 25\%$). If user B starts a second process, each user will still receive 25% of the total cycles, but each of user B's processes will now be attributed 12.5% of the total CPU cycles each, totaling user B's fair share of 25%. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole ($100\% / 5 = 20\%$).

Another layer of abstraction allows us to partition users into groups, and apply the fair share algorithm to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles will be distributed as follows:

$100\% / 3 \text{ groups} = 33.3\% \text{ per group}$
Group 1: $(33.3\% / 3 \text{ users}) = 11.1\% \text{ per user}$
Group 2: $(33.3\% / 2 \text{ users}) = 16.7\% \text{ per user}$
Group 3: $(33.3\% / 4 \text{ users}) = 8.3\% \text{ per user}$.

Lottery Scheduling

- Lottery scheduling is a process scheduling algorithm used in operating systems that assign processes a fixed number of "lottery tickets" based on their priority, determining their likelihood of execution.
- The higher the priority of a process, the more tickets the lottery process scheduling algorithm receives.
- The scheduler chooses a ticket at random from the pool of available tickets. For execution, this algorithm chooses the process that owns the winning ticket.
- The lottery scheduling algorithm is probabilistic.

Lottery Scheduling

In **lottery scheduling**, each process is allocated a certain number of virtual “tickets.” At set intervals, the scheduler will draw a ticket at random and allow the winning process to run on the CPU. This solves the problem of starvation: As long as we give at least one ticket to each process, then we’re able to guarantee a non-zero probability of selection. The idea here is that a process with a fraction $\frac{1}{N}$ of the total tickets will get to use the CPU for a fraction $\frac{1}{N}$ of the time. Notice that if we assign each process the same number of tickets, then we’re using guaranteed scheduling!

1. We have two processes A and B. A has 60 tickets (ticket number 1 to 60) and B has 40 tickets (ticket no. 61 to 100).
2. Scheduler picks a random number from 1 to 100. If the picked no. is from 1 to 60 then A is executed otherwise B is executed.
3. An example of 10 tickets picked by the Scheduler may look like this follows:

```
Ticket number - 73 82 23 45 32 87 49 39 12 09.  
Resulting Schedule - B B A A A B A A A A.
```

1. A is executed 7 times and B is executed 3 times. As you can see that A takes 70% of the CPU and B takes 30% which is not the same as what we need as we need A to have 60% of the CPU and B should have 40% of the CPU. This happens because shares are calculated probabilistically but in a long run(i.e when no. of tickets picked is more than 100 or 1000) we can achieve a share percentage of approx. 60 and 40 for A and B respectively.

Example

- Three threads
 - A has 5 tickets
 - B has 3 tickets
 - C has 2 tickets
- If all compete for the resource
 - B has 30% chance of being selected
- If only B and C compete
 - B has 60% chance of being selected

Real-Time Scheduling Algorithms

Real-time scheduling algorithms are grouped into two primary categories:



Static Scheduling

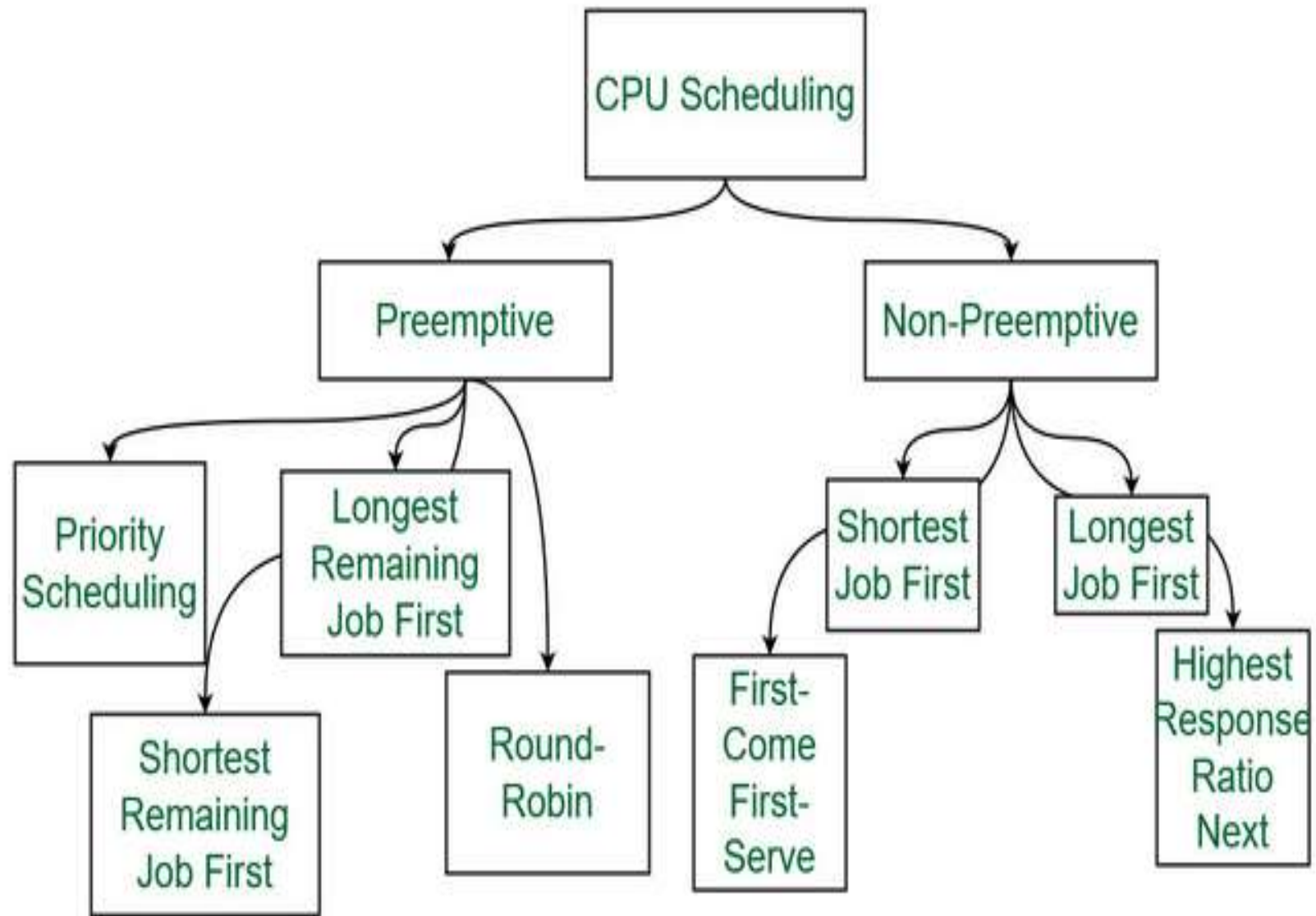
With static scheduling, decisions about what to run next are not made in real time. We can do this by:

1. Feeding the system a pre-made list of processes and the order in which they should run. This can help us save time that would otherwise be lost due to an inefficient scheduling algorithm or real-time decision-making (if there are lots of processes to run).
2. Building scheduling decisions into our code by means of control mechanisms like locks and semaphores to allow threads to take turns and to share resources like the CPU, RAM, buffers, files, and so on. When a thread attempts to claim a lock on a resource that's currently in use, it will simply be blocked until that resource is freed.

Dynamic Scheduling

Dynamic scheduling is a broad category of scheduling that employs any of the algorithms we've looked at so far. However, in the context of real-time systems, it prioritizes scheduling according to the system's deadlines:

-  **Hard real-time deadlines** are ones that we simply can't afford to miss; doing so may result in a disaster, such as a jet's flight controls failing to respond to a pilot's input.
-  **Soft real-time deadlines** are ones that will produce a minor inconvenience if they aren't met. An example of this is a video whose audio isn't properly synced, causing a noticeable delay between what is shown and what's actually heard.



- Categories of Scheduling Algorithm Systems
- 1. Batch Scheduling Algorithms
 - First Come First Serve (NP)
 - Shortest Job First (NP), aka Shortest Job Next
 - Shortest Remaining Time First §
- 2. Interactive Scheduling Algorithms
 - Round Robin Scheduling §
 - Preemptive Priority §
 - Proportionate Scheduling §
 - Guaranteed Scheduling
 - Lottery Scheduling
 - Fair-Share Scheduling
- 3. Real-Time Scheduling Algorithms
 - Static Scheduling
 - Dynamic Scheduling

Lecture Outline

Operations on Processes

Creation

Termination

Zombie Process

Orphan Process

Threads

Operations on Processes

- System must provide mechanisms for:
 - Process creation
 - Process termination

Process related system calls (in Unix)

- fork() creates a new child process
 - All processes are created by forking from a parent
 - The init process is ancestor of all processes
- exec() makes a process execute a given executable
- exit() terminates a process
- wait() causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- **Resource sharing options**
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- **Execution options**
 - Parent and children execute concurrently
 - Parent waits until children terminate

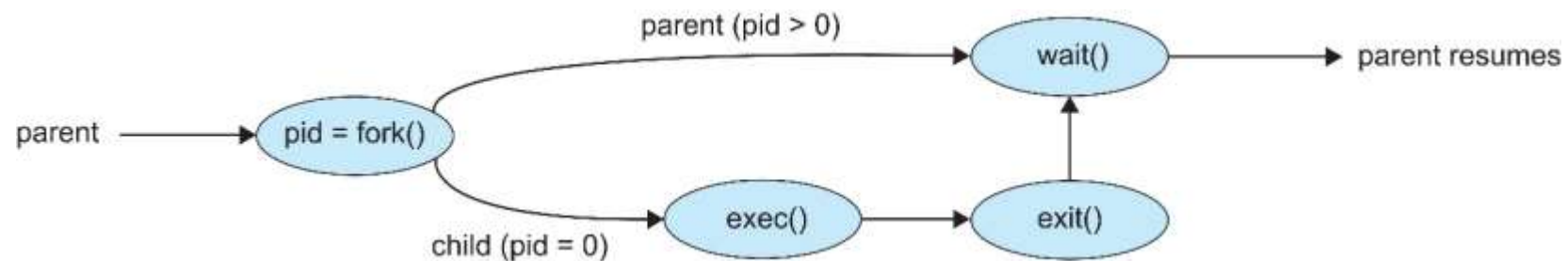
Process Creation (Cont.)

- **Address space**

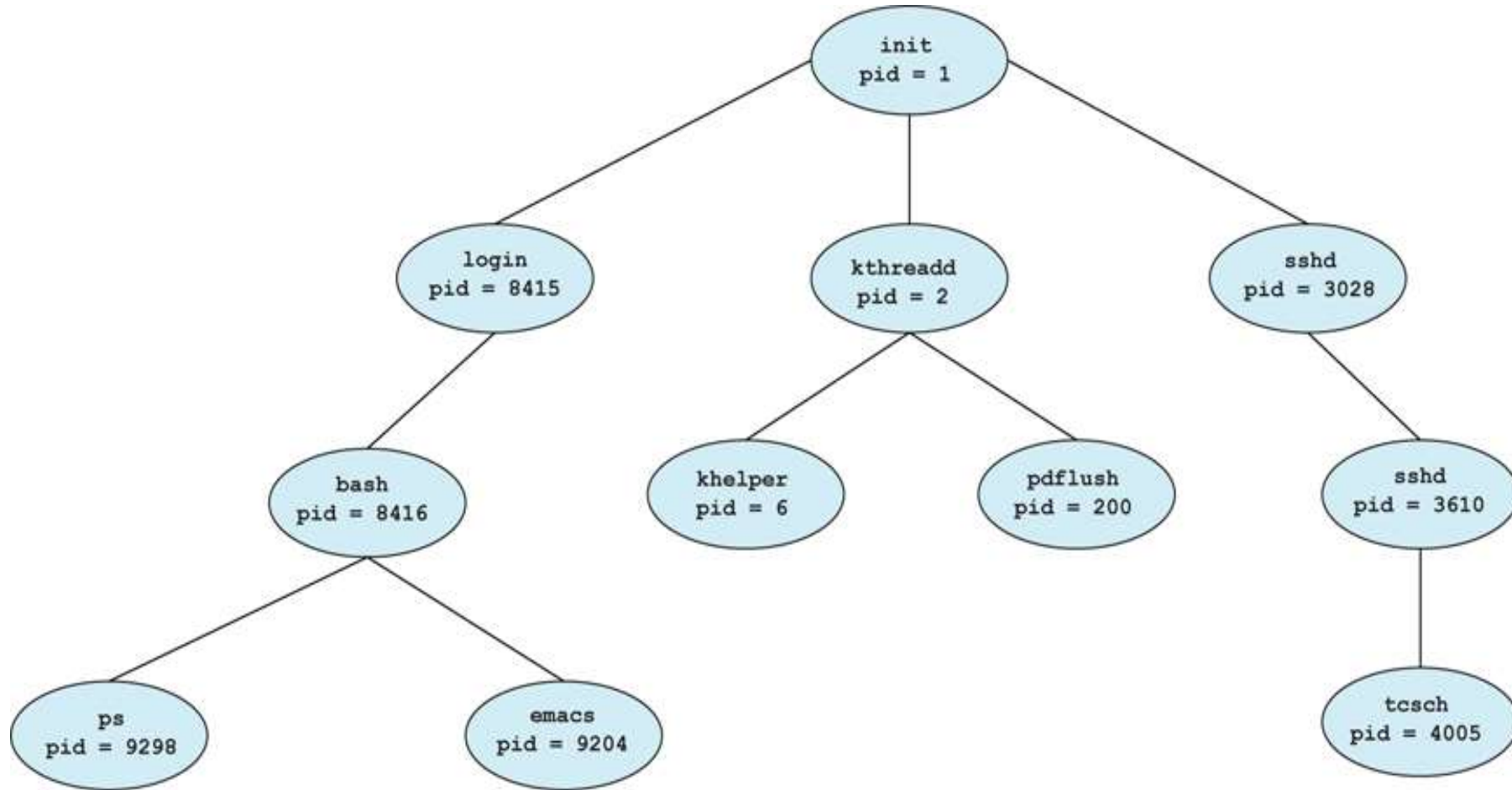
- Child duplicate of parent
- Child has a program loaded into it

- **UNIX examples**

- `fork()` system call creates new process
- `exec()` system call used after a `fork()` to replace the process' memory space with a new program
- Parent process calls `wait()` waiting for the child to terminate

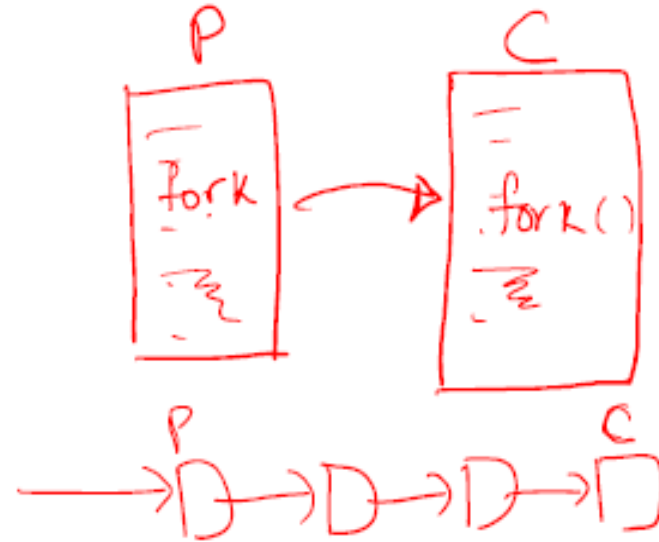


A Tree of Processes in Linux



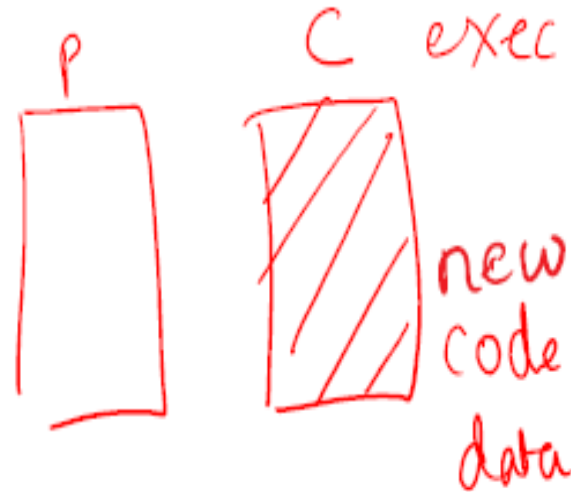
What happens during a fork?

- A new process is created by making a copy of parent's memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify the memory data independently



What happens during exec?

- After fork, parent and child are running same code
 - Not too useful!
- A process can run `exec()` to load another executable to its memory image
 - So, a child can run a different program from parent
- Variants of `exec()`, e.g., to pass commandline arguments to new executable



C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

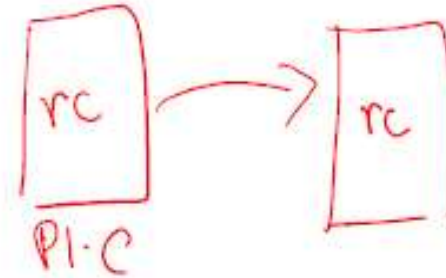
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {          // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {              // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18     }
19     return 0;
20 }

```



// child
} parent

Calling `fork()` (p1.c)

When you run this program (called `p1.c`), you'll see the following:

↕

```

prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>

```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
 - Returns status data from child to parent (via `wait()`)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates

Process Termination

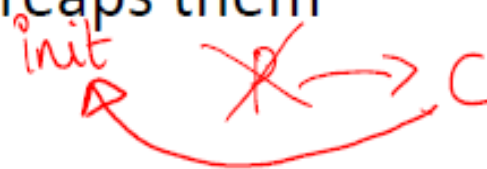
contd..

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc., are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait()`, process is an **orphan**

Waiting for children to die...

- Process termination scenarios
 - By calling `exit()` (exit is called automatically when end of main is reached)
 - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent calls `wait()`, zombie child is cleaned up or “reaped”
- `wait()` blocks in parent until child terminates (non-blocking ways to invoke wait exist)
- What if parent terminates before child? `init` process adopts orphans and reaps them



Introduction to Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution.

For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker.

Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. **This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.**

A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread.

Threads

- ❑ A thread is a basic unit of CPU utilization.
- ❑ It comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- ❑ A thread is the smallest unit of execution within a process.
- ❑ Threads are light weight processes.

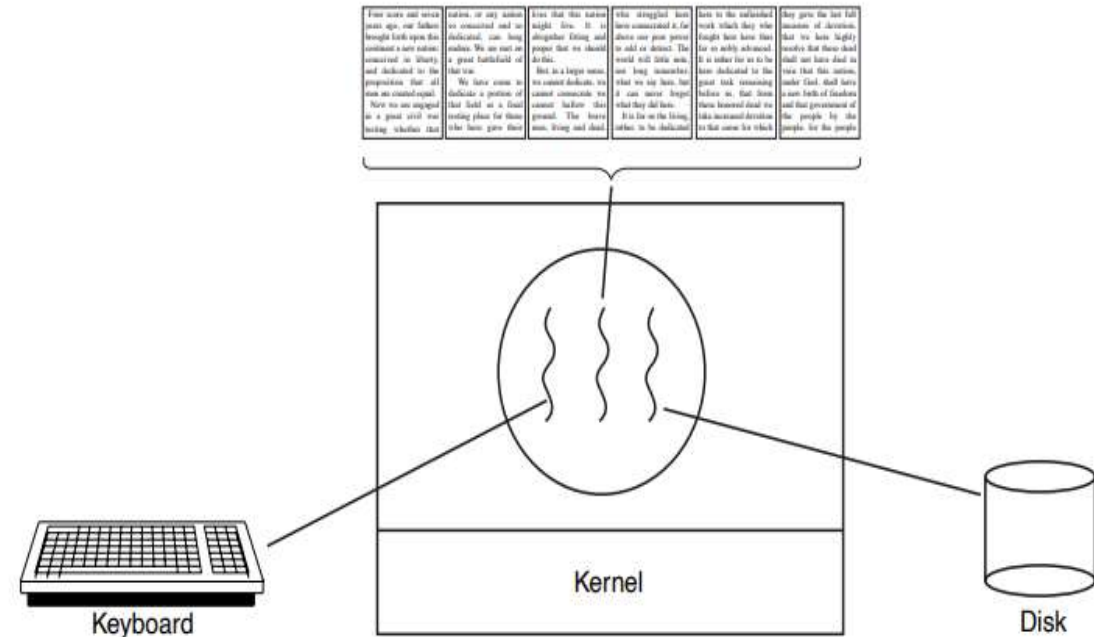
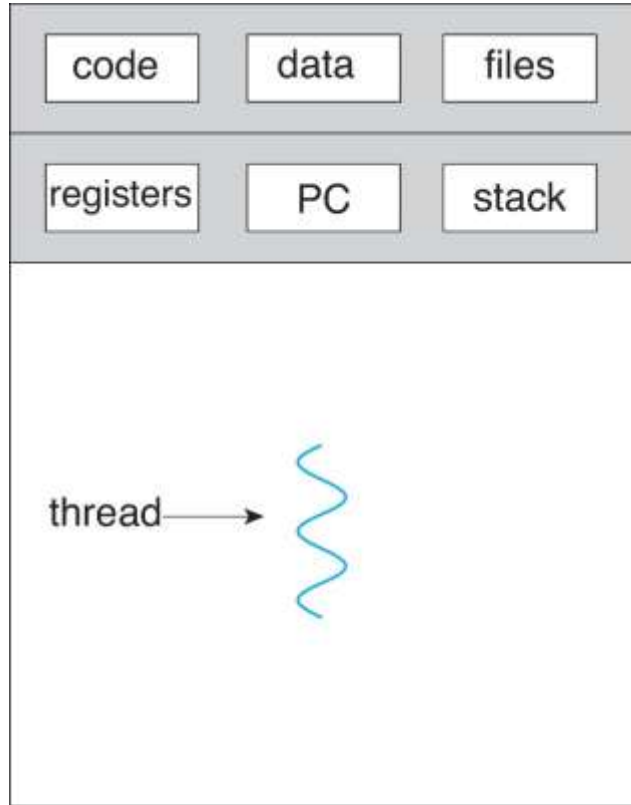
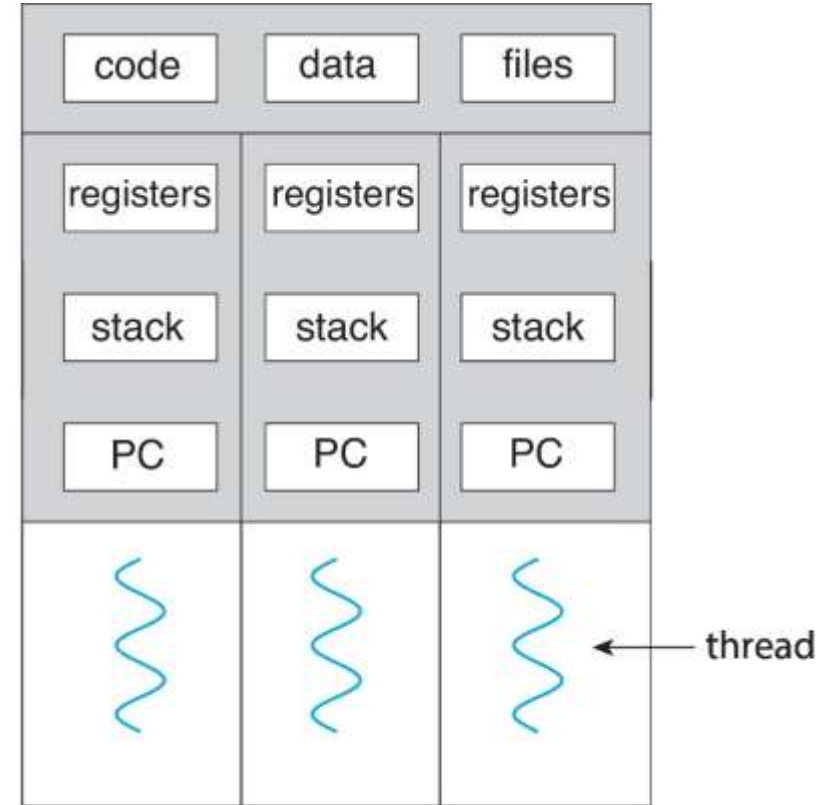


Figure 2-7. A word processor with three threads.

Single and Multithreaded Processes



single-threaded process



multithreaded process

Benefits of Multithreading

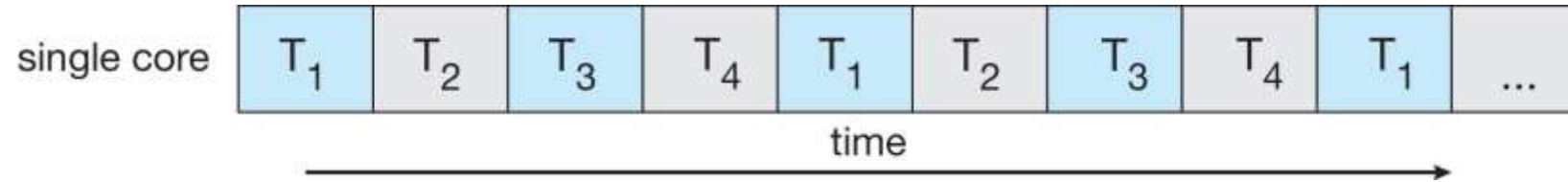
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing.
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching.
- **Scalability** – process can take advantage of multicore architectures.

Challenges in programming for multicore systems:

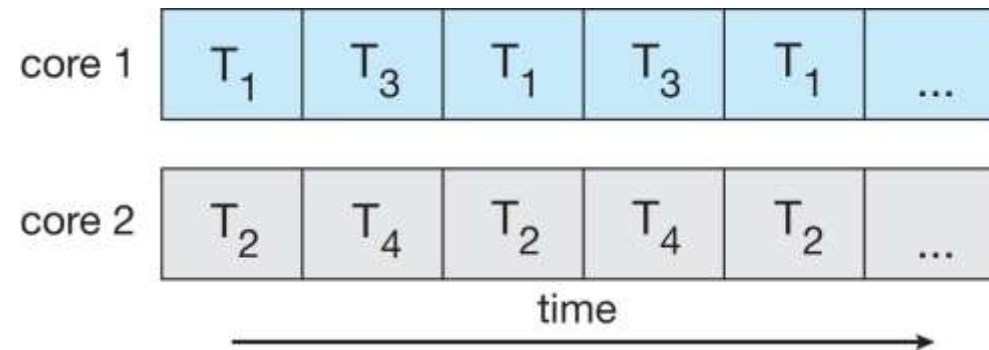
- **Multicore** or **multiprocessor** challenges include:
 - **Identifying tasks**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Multicore Programming

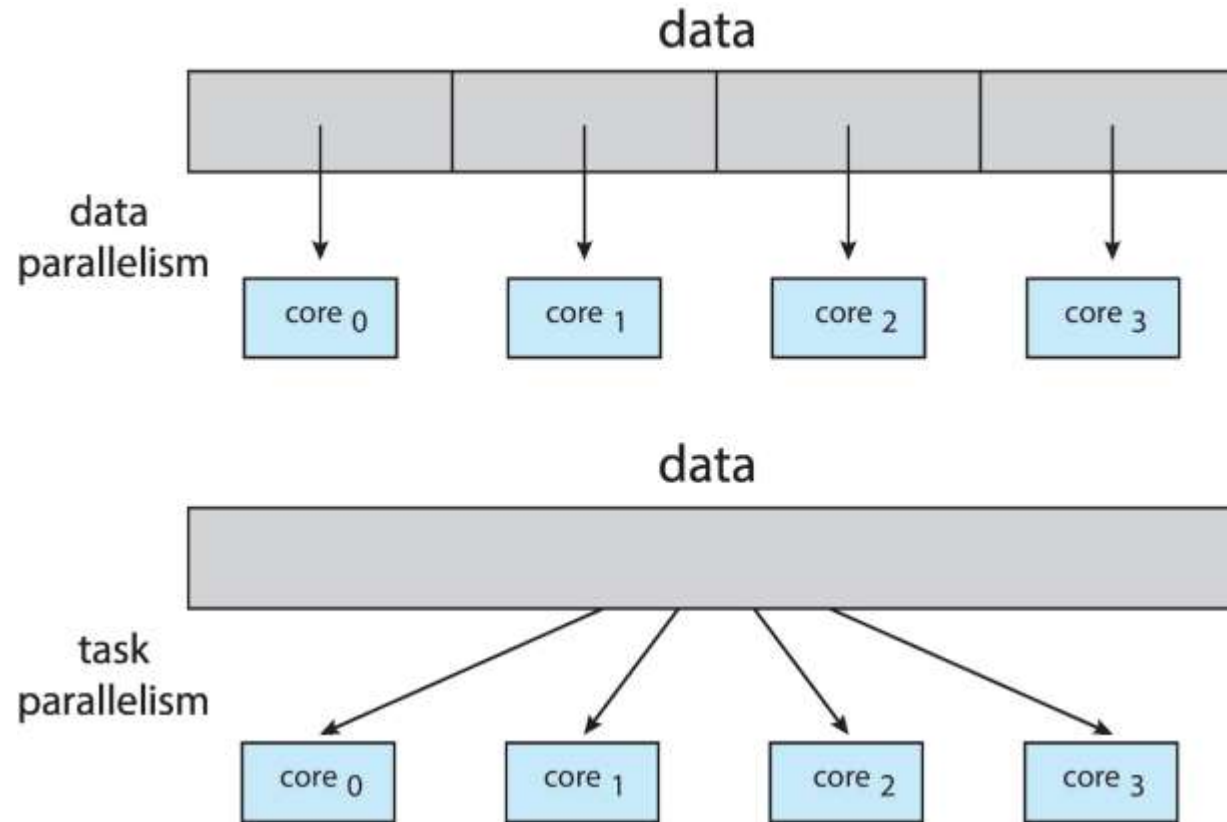
Parallelism implies a system can perform more than one task simultaneously.

Concurrency supports more than one task making progress.

Single processor / core, scheduler providing concurrency

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism

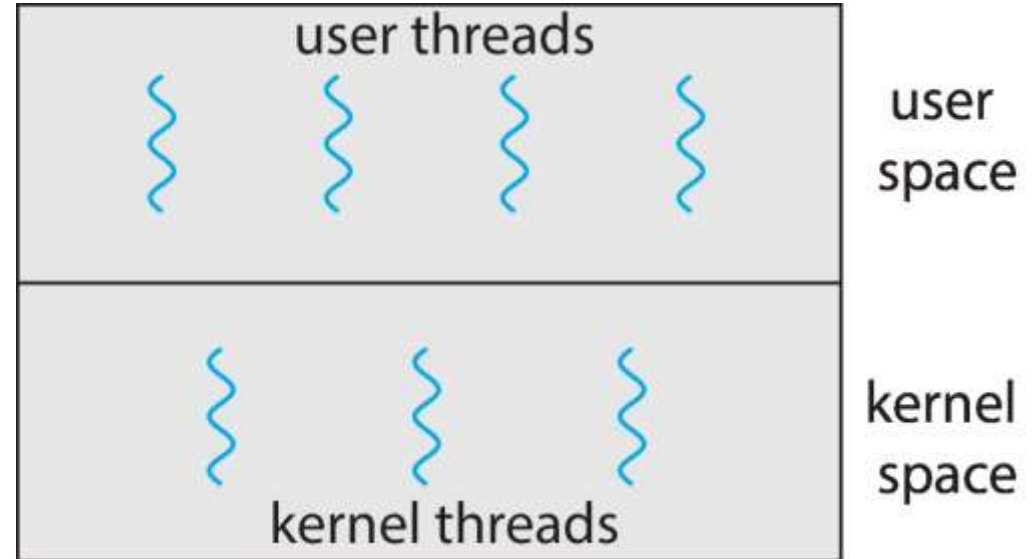


Types of Threads

User and Kernel Threads

UT: Supported above the kernel and are managed without kernel support.

KT: Supported and managed directly by the OS.



Ultimately, there must exist a relationship between user threads and kernel threads.

Multithreading models

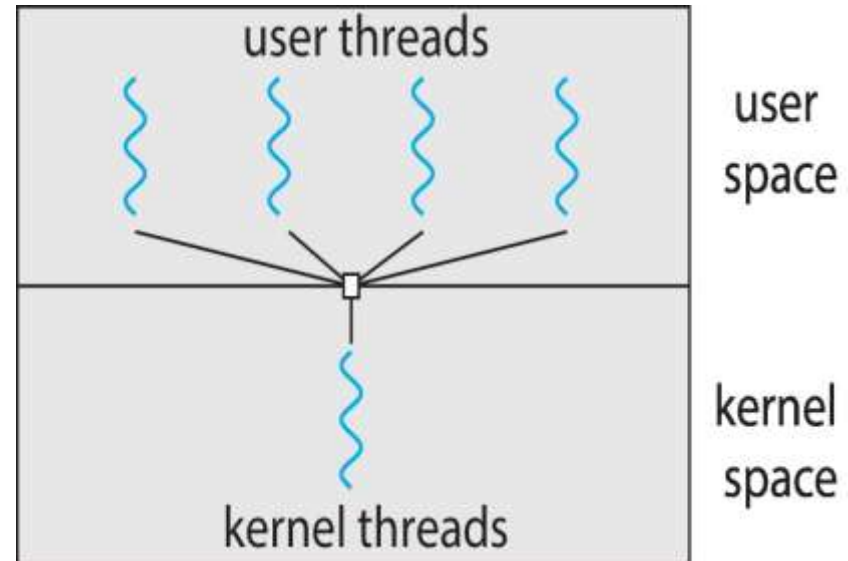
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

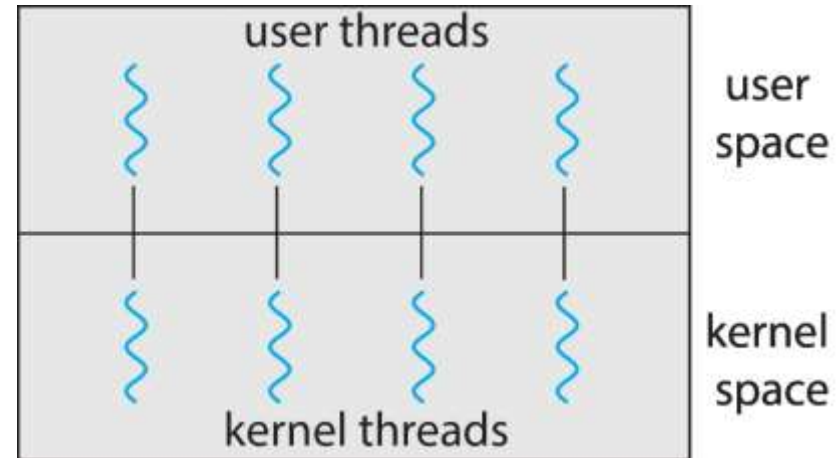
- Many user-level threads mapped to single kernel thread
- Thread mgt. is done by the thread library in the user space, so it is efficient.
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:

- Solaris Green Threads



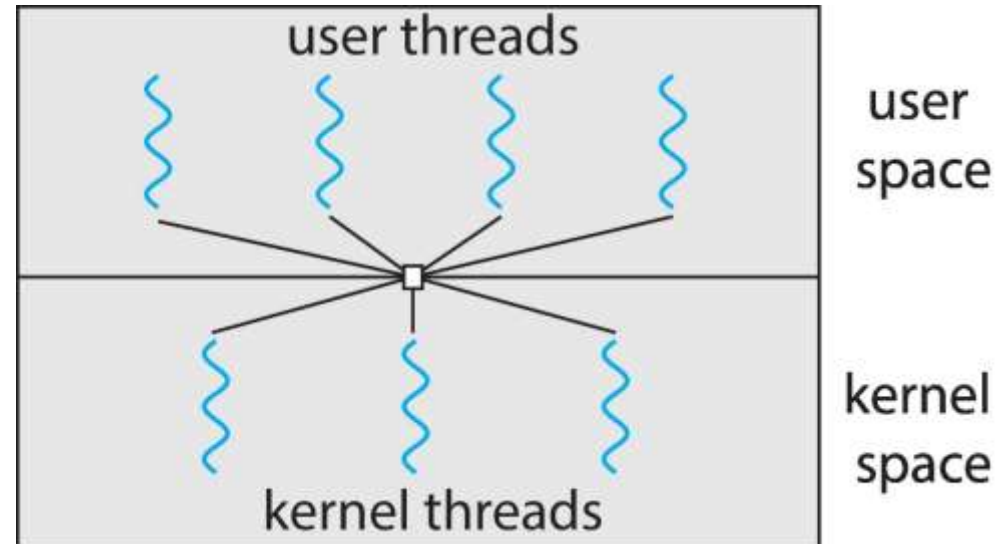
One-to-One

- maps each user-level thread to kernel thread
 - Creating a user-level thread creates a kernel thread
 - More concurrency than many-to-one
 - Also allows multiple threads to run in parallel on multiprocessors.
-
- Creating a user thread requires creating a kernel thread.
 - Number of threads per process sometimes restricted due to overhead of kernel threads.



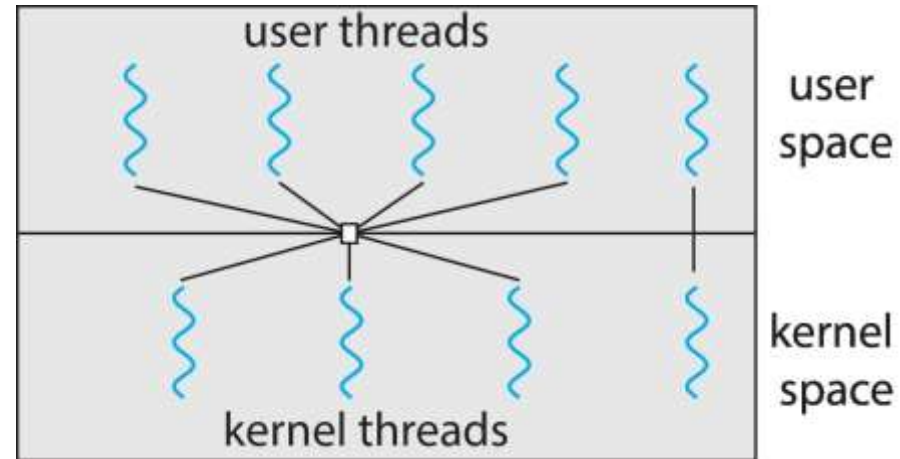
Many-to-Many Model (Hybrid)

- Allows many user level threads to be mapped to many kernel threads.
- Multiplexes many user threads to smaller or equal number of kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Hyperthreading or SMT

- **Simultaneous Multithreading (SMT).**
- Each physical CPU core appears as **two (or more) logical cores** to the OS.
- The hardware can execute instructions from multiple threads in parallel if resources (like execution units) are free.
- Example: A quad-core CPU with hyper-threading shows up as **8 logical processors**.

NOTE:

- **Multithreading models = OS/software concept** (how threads map to kernel scheduling units).
- **Hyper-Threading = CPU/hardware concept** (how a physical core can execute multiple instruction streams at once).

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred

Semantics of `fork()` and `exec()`

- Does **`fork()`** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of `fork`
- **`exec()`** usually works as normal – replace the running process including all threads

Thread Cancellation

- Thread cancellation is the task of terminating a thread before it has completed.
- For example,

Ex.1. If multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.

Ex. 2. When a user presses a button on a web browser that stops a web page from loading any further, all threads loading the page are canceled.

➤ Thread to be canceled is referred as **target thread**.

Thread Cancellation

- Cancellation of a target thread may occur in two different scenarios:
- **Asynchronous cancellation:** One thread immediately terminates the target thread.
- **Deferred cancellation:** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Where the difficulty lies in thread cancellation?

In situations where:

- Resources have been allocated to a canceled thread.
- A thread is canceled while in the midst of updating data it is sharing with other threads.

Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources.

Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation:

One thread indicates that a target thread is to be canceled.

But cancellation occurs only after the target thread has checked a **flag** to determine if it should be canceled or not.

This allows a thread to check whether it should be canceled at a point when it can be canceled safely.



Thank You





Thank You

