Unit II- Process Management

J.Premalatha
Professor/IT
Kongu Engineering College
Perundurai

Processes and threads

To understand processes, let us first try to understand how a program is developed. So, whenever want to make a program, **first write the program in some high level languages**. For example, write the program in C, C++, JAVA and that program is written in a high level language. The computer does not understand the high level language. But it understands only the binary codes, which are 0's and 1's. So, **the program has to be converted to binary code**. So, for that using a compiler. When program is compiled ,it convert that program into machine code which is understandable by machine. **Then it is converted into a binary executable code and it is ready for execution.** But it is **not enough to just have that binary code for a program to execute** or for **a program to tell the computer what it wants to do**. So, it has to be loaded into the memory, and for a program to execute, it needs some resources of the computer system. **The operating system will help in loading that executable program into the memory and allocate its resources and then the program will begin its execution.**

A program which is written and is ready for execution but till that time it is just a passive entity. That means it just sit there without doing anything. But **the moment it begins execution, at that instance program is called as a process.** So, a **process is a program in execution and it is active entity.**

In the earlier computers it supported only one process or one program at a time. But in today's computer it supports multiple processes or programs running at the same time. And even **one single program can have many processes associated with it.**

A thread is the unit of execution within a process. A process have one thread to many threads. **When a program is executing it is called a process**. And a **thread is the unit of execution within a process**. **So, within a process there may be one or more units of its execution and those units are known as threads.**

Thread applications- examples

•An application that creates **photo thumbnails** from a collection of images may use a **separate thread to generate a thumbnail from each separate image.** [**view the Gallery is the process**]

• A **web browser** might have one **thread display images or text** while **another thread retrieves data from the network**.[**open the web browser is the process]**

• A **word processor** may have a **thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking** in the background .[**open the word document is the process**]

In MAC, to see Processes and threads

1.Press Command + Space to open Spotlight.

2.Start typing Activity Monitor.

3.Once Activity Monitor comes up highlighted, hit Enter or click on it

In Windows, to see Processes,

Press Ctrl+Shift+Esc see windows task manager to see the processes running in a machine. Select performance tab and view process and thread details.

To see the threads in system in a detailed way,

- **then use a program known as process explorer. Download that program and it will show the threads that are running for each process**

Process Concept

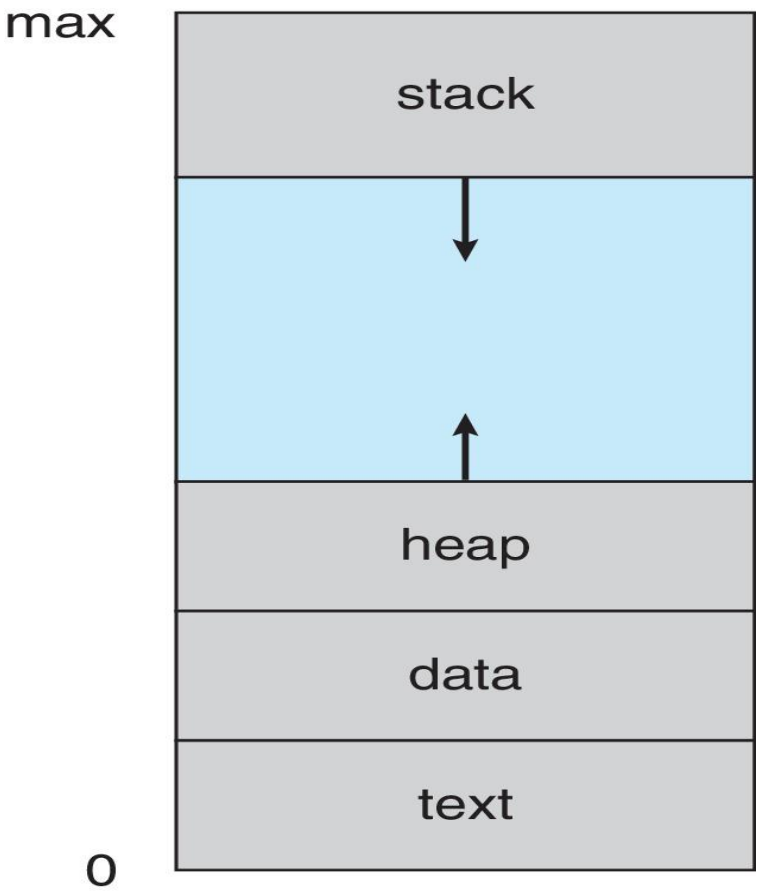
An operating system executes a variety of programs that run as a process.

Process – **a program in execution**; process execution must progress in sequential fashion.

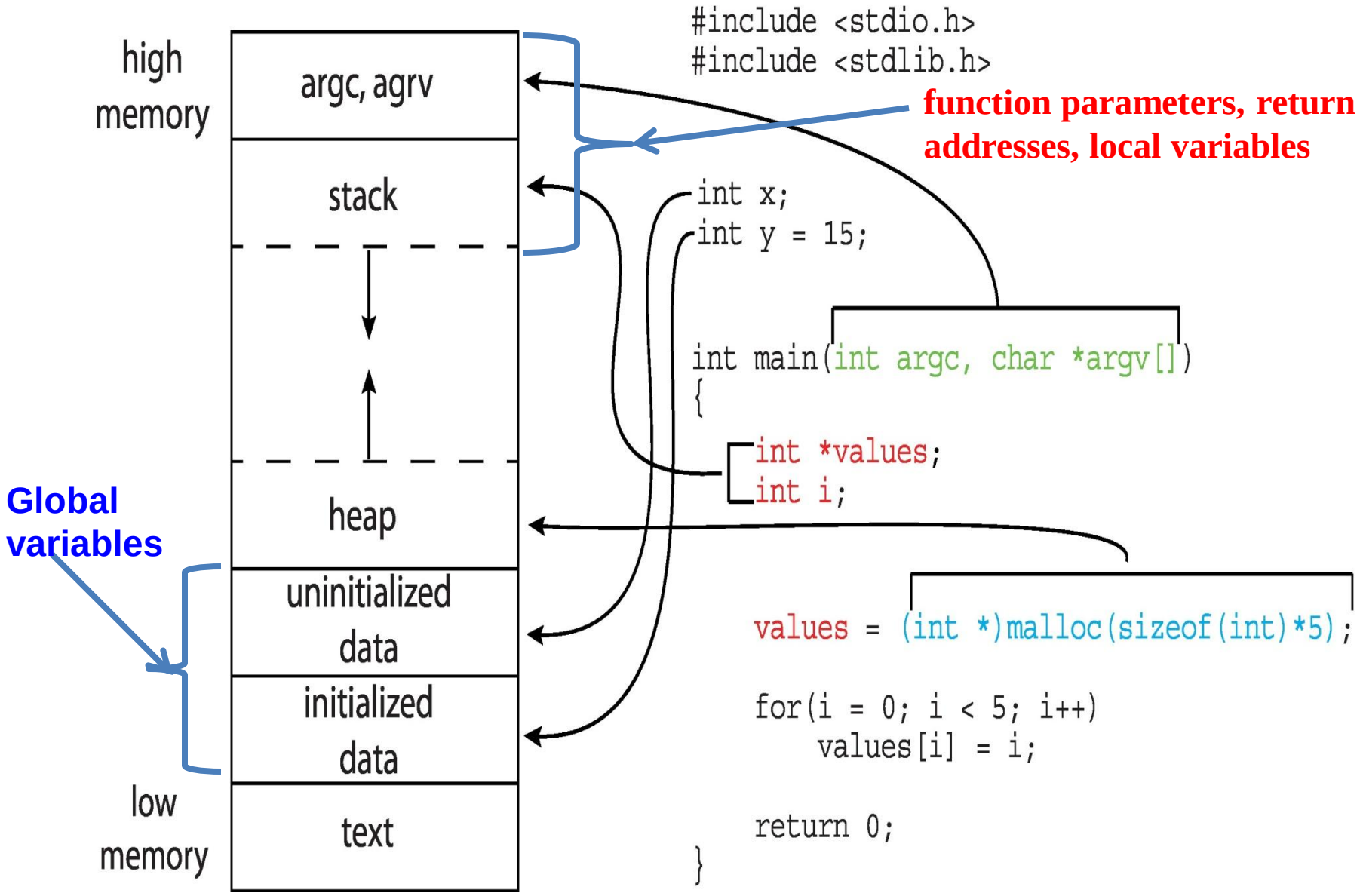
Memory layout of a process has Multiple parts :

- **Text section** —the executable code
- **Data section** —**global variables**
- **Heap section** —memory that is dynamically allocated during program run time
- **Stack section** — **temporary data storage** when invoking functions (such as **function parameters, return addresses**, and **local variables**)
- **program counter, processor registers**

Process in Memory

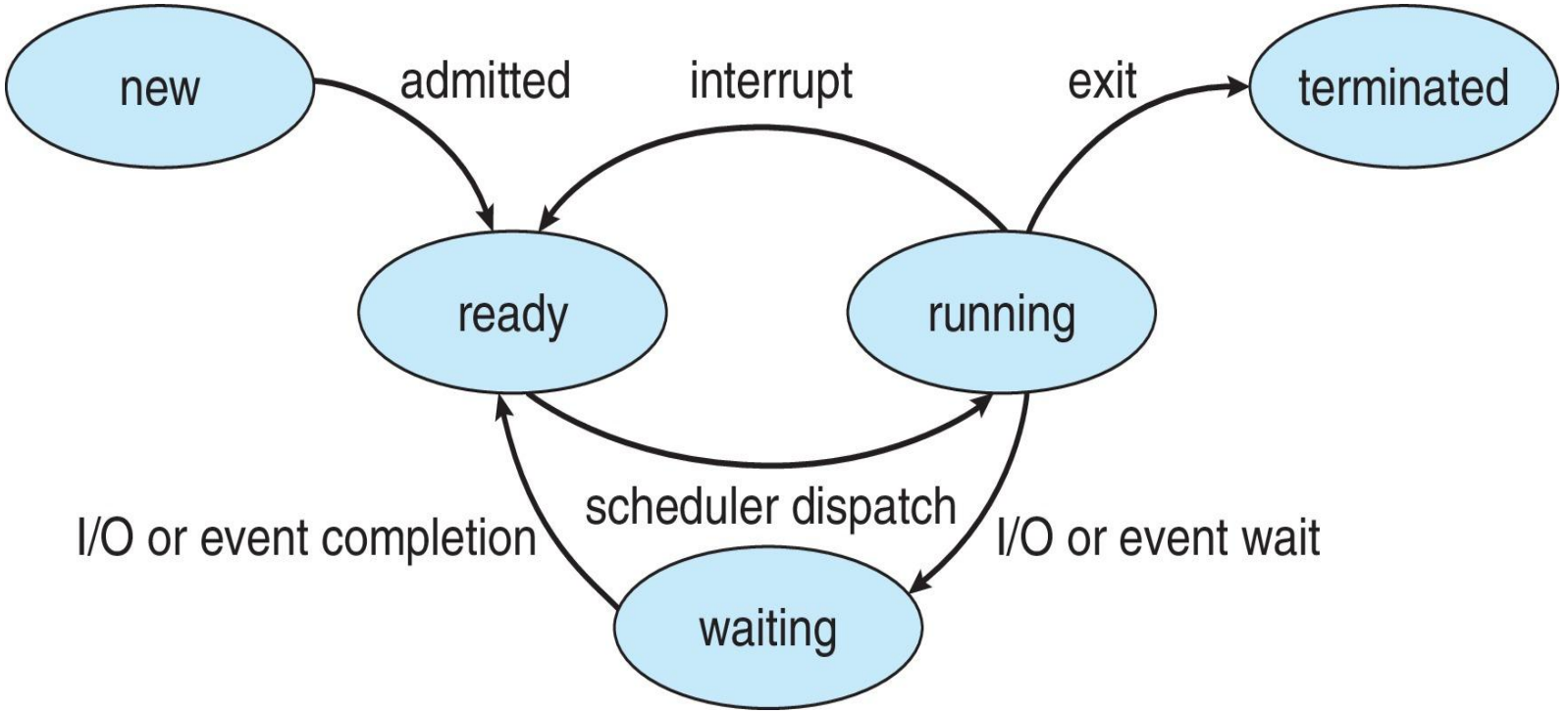


Memory Layout of a C Program



Process State

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

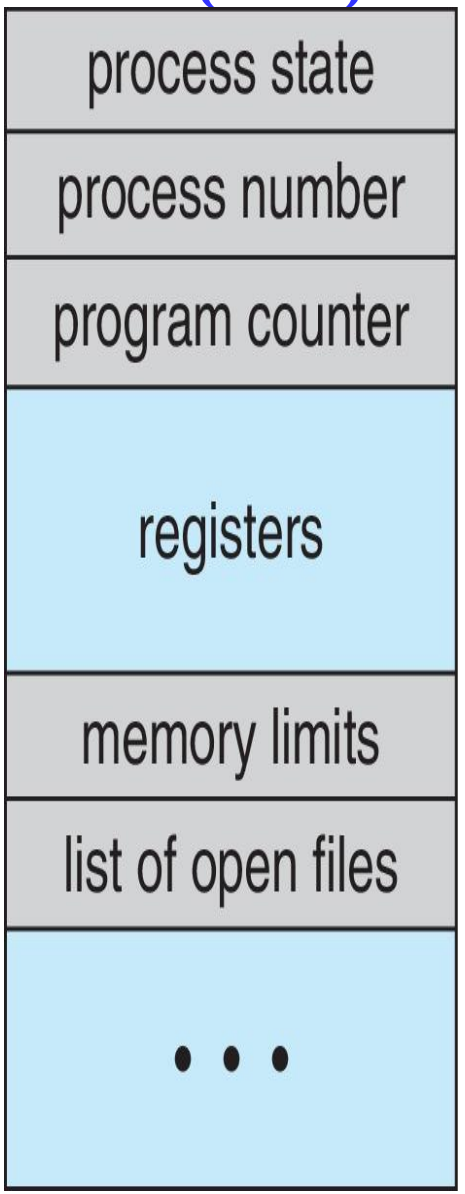


Information associated with each process is in Process Control block (also called **task control block**)

The process control block is kept **in a memory area that is protected from the normal user access**.

- **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

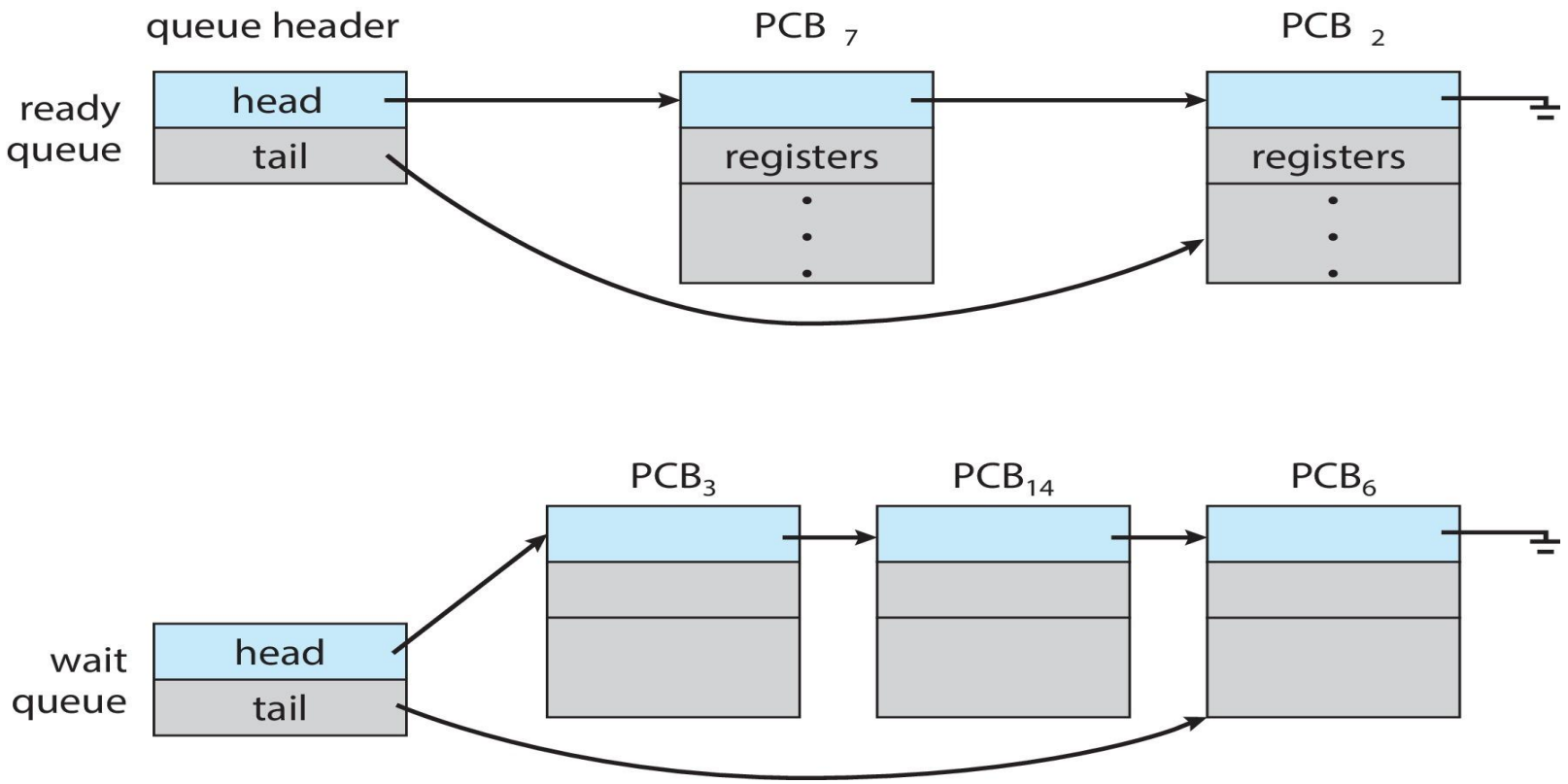
Process Control Block (PCB)



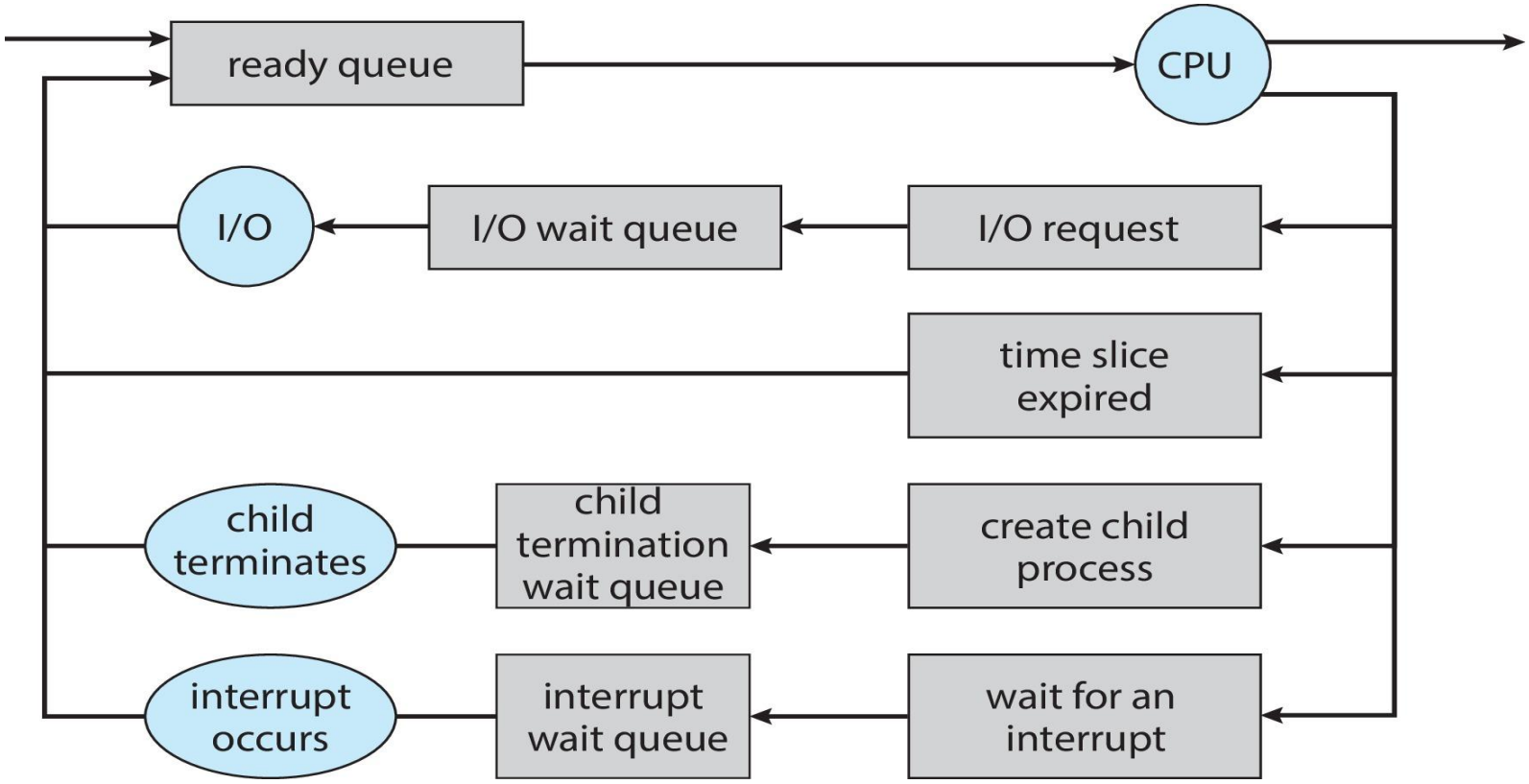
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues

Ready and Wait Queues

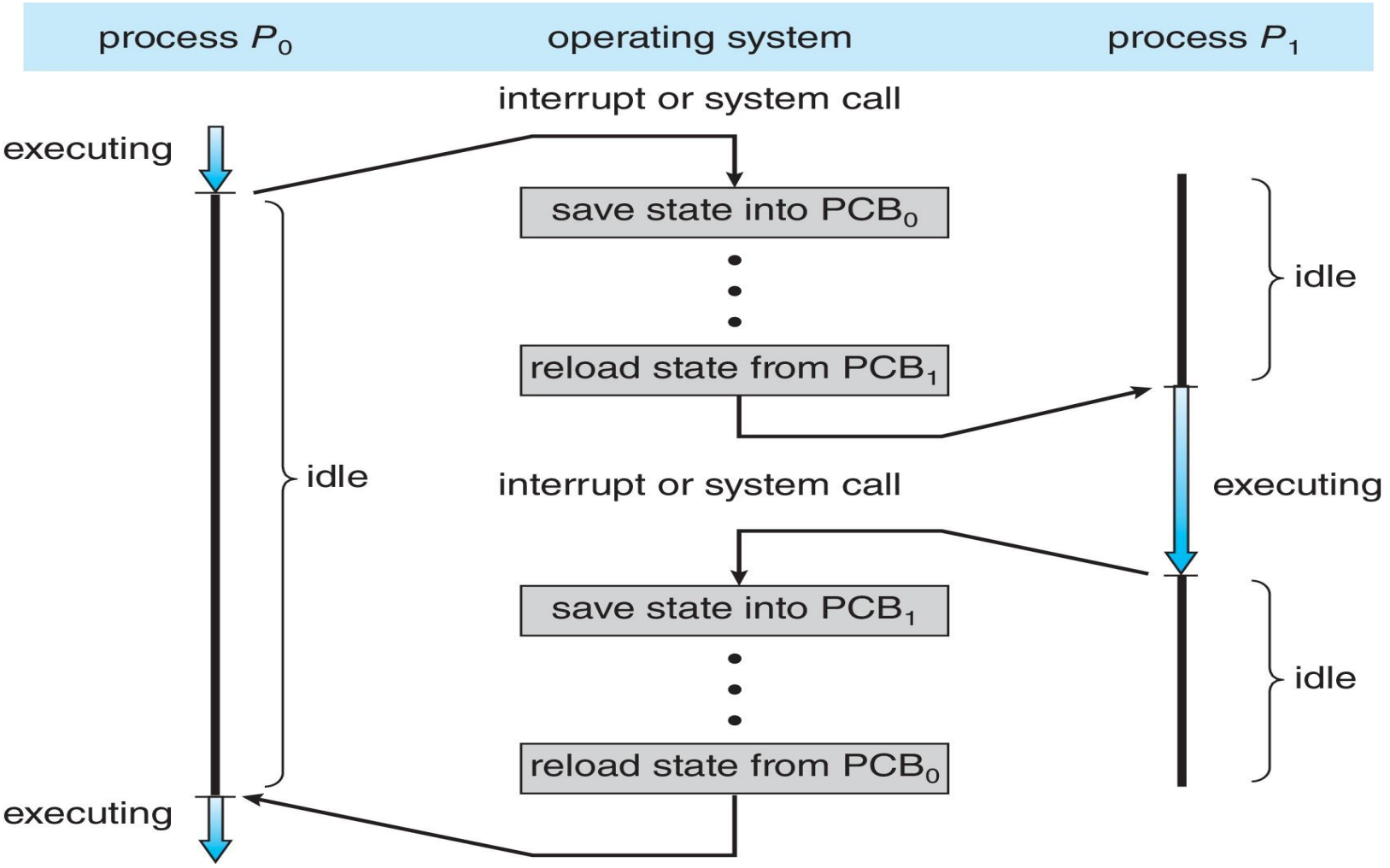


Representation of Process Scheduling



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
- **Processes that require a user to start** them or to interact with them are called **foreground processes**. **Processes that are run independently of a user** are referred to as **background processes**.
- 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 of receiving notification of events** and **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

Operations on Processes

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

Process Creation

•A **process may create several new processes**, via a **create-process system call(Windows)**, **fork system call** (Unix, Linux)during the course of execution

•**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

Parent Process: All the processes are created when a process executes the **fork() system call except the startup process.** The **process that executes** the **fork() system call** is the **parent process**. A parent process is one that creates a child process using a fork() system call. A **parent process may have multiple child processes, but a child process has only one parent process.**

On the success of a fork() system call:

- The Process ID (PID) of the child process is returned to the parent process.
- 0 is returned to the child process.

On the failure of a fork() system call :

- -1 is returned to the parent process.
- A child process is not created.

Child Process: A child process is created by a parent process in an operating system using a fork() system call. A **child process may also be known as subprocess or a subtask.**

- A child process is created as a copy of its parent process.[child pid is 0]
- If a **child process has no parent process, then the child process is created directly by the kernel.**

Why Do We Need to Create A Child Process: Sometimes there is a need for a program to perform more than one function simultaneously. **Since these jobs may be interrelated so two different programs to perform them cannot be created.** For example: **Suppose there are two jobs: copy contents of source file to target file and display an animated progress bar indicating that the file copy is in progress.** The GIF progress bar file should continue to play till file copy is taking place. Once the copying process is finished the playing of the GIF progress bar file should be stopped. Since both these jobs are interrelated they cannot be performed in two different programs. Also, they cannot be performed one after another. Both jobs should be performed simultaneously.

THE init AND systemd PROCESSES in Unix,Linux

- Traditional UNIX systems identify the process **init** as the root of all child processes. **Now modern system init is changed as systemd**
- The **systemd** process (**which always has a pid of 1) serves as the root parent process for all user processes**, and is the **first user process created when the system boots.**
- Once the system has booted, the **systemd process creates processes which provide additional services** such as a web or print server

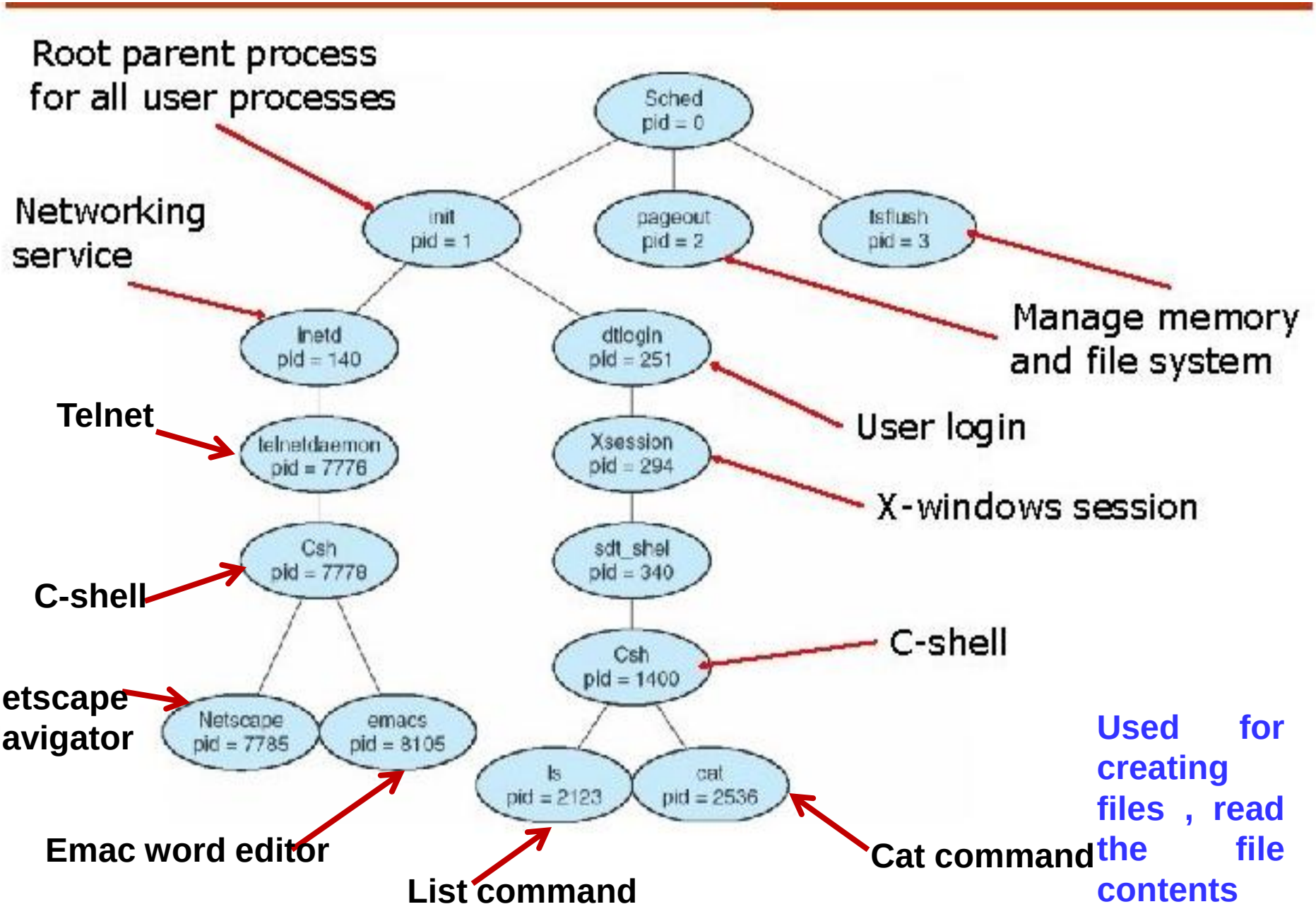
G1. A process executes the code (GATE 2012)

```
fork();
fork();
fork();
```

How many child process are created?

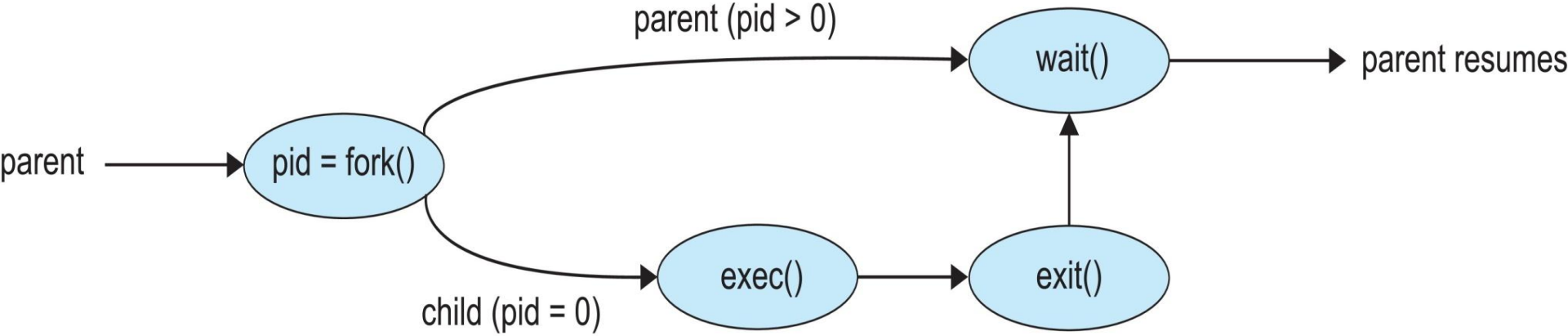
If there are n fork calls, then the **number of child processes created is 2^n-1.**

A Tree of Processes in Solaris



Process Creation (Cont.)

- **Address space**
  1. Child duplicate of parent[It has the same program and data as the parent – **Same memory address for both child and parent**]
  2. Child has a new program loaded into, it **has different address space**
- **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



Process Termination

- **Process executes last statement and then asks the operating system to delete it using the exit() system call.**
  - Returns status value[typically integer value] 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(Cont.)

- Operating systems do not allow child to exists if its parent has terminated. **If a parent 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

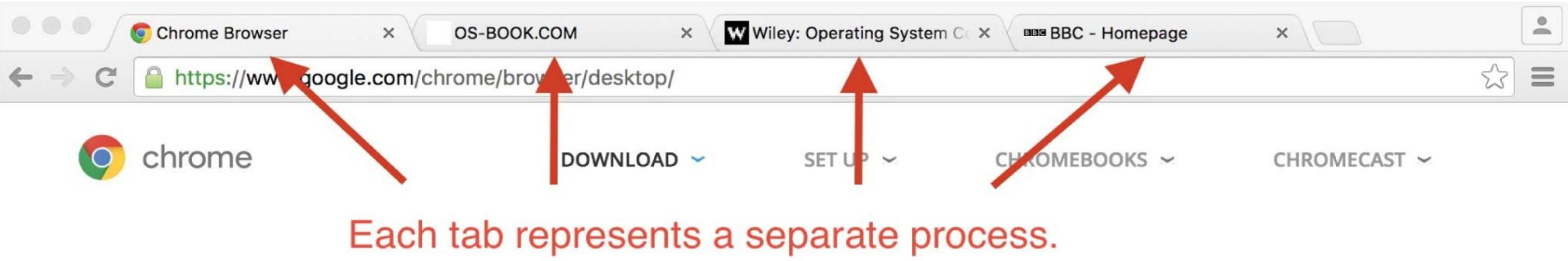
  pid = wait(&status);
- **If no parent waiting (did not invoke wait()), but the child process completes its execution , then the child process is a zombie**
- **If parent terminated without invoking wait(), then the child process is an orphan**

Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - **Foreground process**
  - **Visible process**
  - **Service process**
  - **Background process**
  - **Empty process**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in



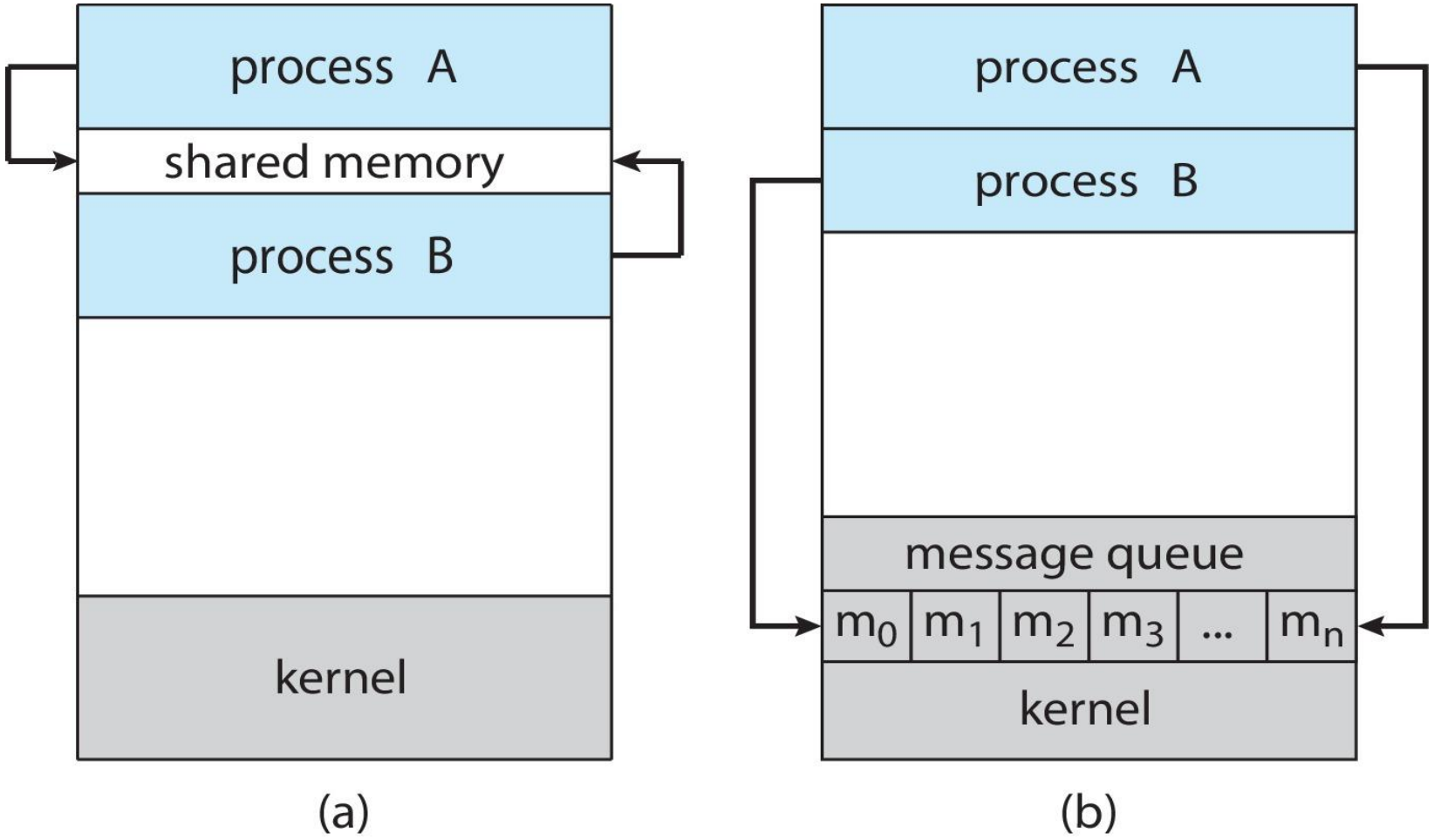
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Shared memory.

(b) Message passing.



Producer-Consumer Problem

- **Paradigm for cooperating processes:**
  - ***producer*** process produces information that is consumed by a ***consumer*** process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

IPC – Shared Memory

- An **area of memory shared among the processes that wish to communicate**
- The **communication is under the control of the users processes** not the operating system.
- Major issues is **to provide mechanism that will allow the user processes to synchronize their actions** when they access shared memory.

Bounded-Buffer – Shared-Memory Solution

Shared data : The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
Item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: **in** and **out**. The **variable in** points to the next free position in the buffer; **out** points to the first full position in the buffer. **The buffer is empty when in ==out;**
the buffer is full when
((in + 1) % BUFFER SIZE) == out.

- Solution is correct, but can only use **BUFFER_SIZE-1** elements

Producer Process – Shared Memory

```
item next_produced;

while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out);
        /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Consumer Process – Shared Memory

```
item next_consumed;

while (true) {
        while (in == out); /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in next consumed */
}
```

The **producer process has a local variable Next_produced** in which the new item to be produced is stored.

The consumer process has a local variable next_consumed in which the item to be consumed is stored.

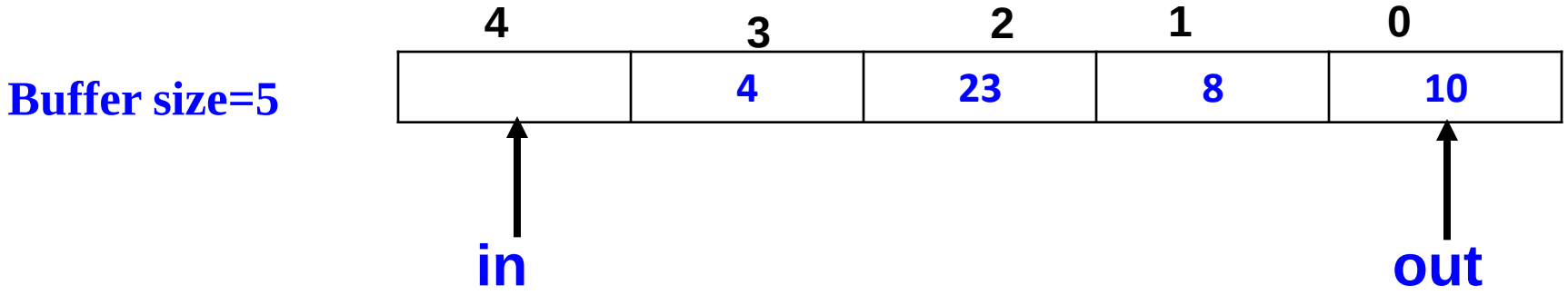
This scheme allows at most BUFFER SIZE−1 items in the buffer at the same time

Need for Synchronization: Producer –Consumer Problem

The buffer is empty when in ==out;
the buffer is full when ((in + 1) % BUFFER SIZE) == out.

```
item next_produced;
while (true)
{
/* produce an item in next produced */
while (((in + 1) % BUFFER_SIZE) == out);
                  /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
}
```

It allow at most BUFFER SIZE − 1 items in the buffer at the same time.



IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- **IPC facility provides two operations:**
  - **send(*message*)** **-receive(*message*)**
- The *message* size is either fixed or variable

➢ If processes *P* and *Q* wish to communicate, they need to:

❖Establish a ***communication link*** between them

❖Exchange messages via send/receive

Implementation issues:

✓How are links established?

✓Can a link be associated with more than two processes?

✓How many links can there be between every pair of communicating processes?

✓What is the capacity of a link?

✓Is the size of a message that the link can accommodate fixed or variable?

✓Is a link unidirectional or bi-directional?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes **(also referred to as ports)**
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- **Properties of communication link**
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Operations

Create a new mailbox (port)
Send and receive messages through mailbox
Delete a mailbox

Primitives are defined as:

send(*A*, *message*) – send a message to mailbox A
receive(*A*, *message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

Message passing may be either blocking or non-blocking

Blocking is considered **synchronous**

Blocking send -- the sender is blocked until the message is received

Blocking receive -- the receiver is blocked until a message is available

Non-blocking is considered **asynchronous**

Non-blocking send -- the sender sends the message and continue

Non-blocking receive -- the receiver receives:

A valid message, or Null message

Producer-Consumer: Message Passing

- **Producer**

```
    message next_produced;
  while (true) {
     /* produce an item in next_produced */

 send(next_produced);
}
```

- **Consumer**

```
     message next_consumed;
  while (true) {
      receive(next_consumed)

     /* consume the item in next_consumed */
 }
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways

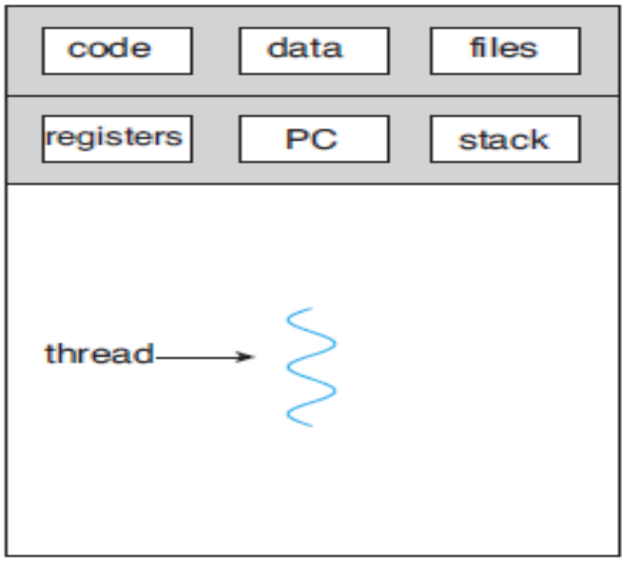
1. Zero capacity: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

2. Bounded capacity: The queue has finite length *n;* thus, atmost n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

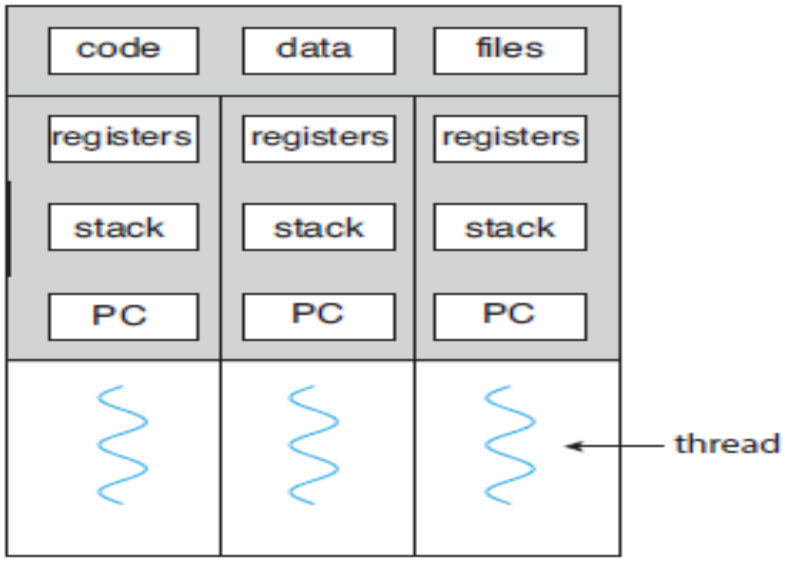
3.Unbounded capacity: The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Thread

- A **thread is a basic unit of CPU utilization;** it comprises a **thread ID, 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 **traditional process has a single thread** of control. If a **process has multiple threads** of control, it can perform more than one task at a time.



single-threaded process



multithreaded process

Multi Threaded applications- examples

•An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.

• A web browser might have one thread display images or text while another thread retrieves data from the network.

• A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Multithreaded Server Architecture

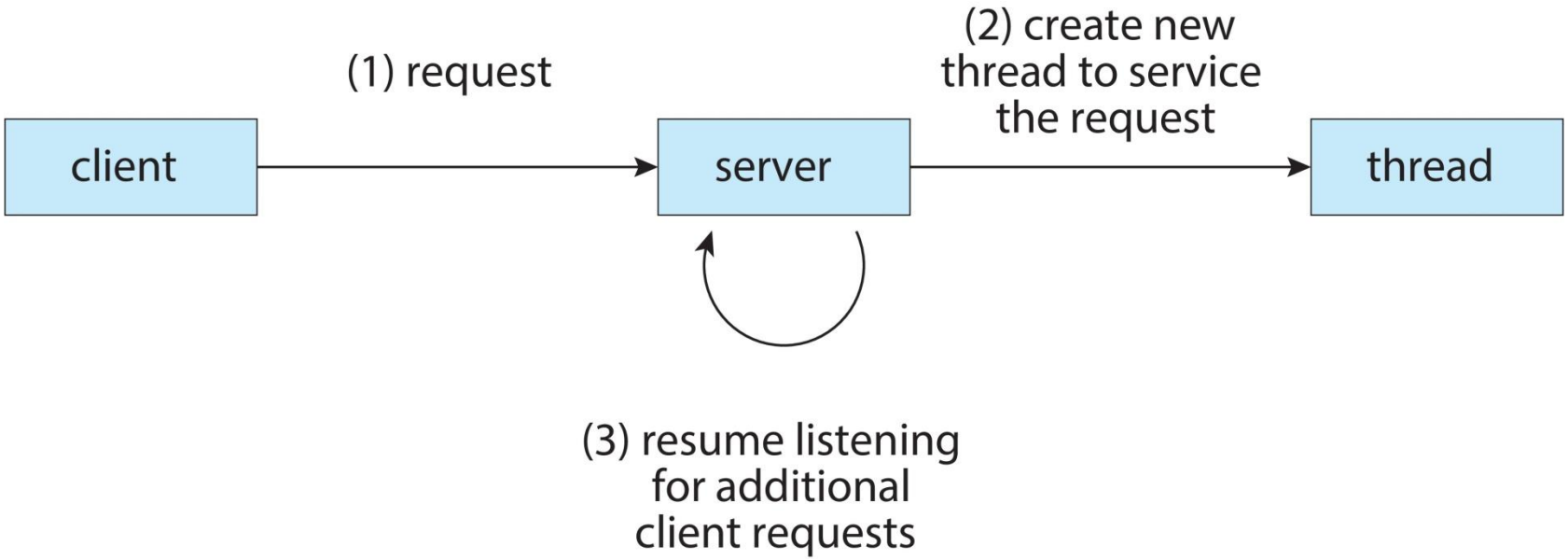
A single application may be required to perform several similar tasks.

For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it.

If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and other clients might have to wait a very long time for its request to be serviced.

One solution is, the **server run as a single process that accepts requests**. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. **Process creation is time consuming and resource intensive,** however. If the new process will perform the same tasks as the existing process, it incur all overhead. **So, process creation is heavy weight.**

Multithreaded Server Architecture



It is more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. **When a request is made, rather than creating another process, the server creates a new thread** to service the request and resumes listening for additional requests. **So, thread creation is light weight.**

Benefits

- **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

Multicore Programming

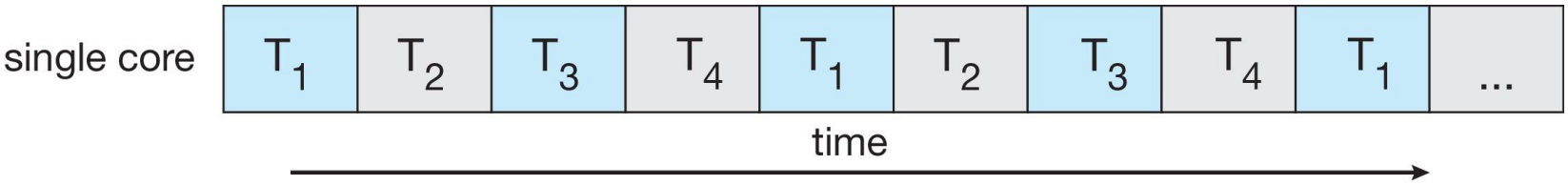
Consider an application with four threads. On a system with a single computing core, Concurrency means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core.

A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism.

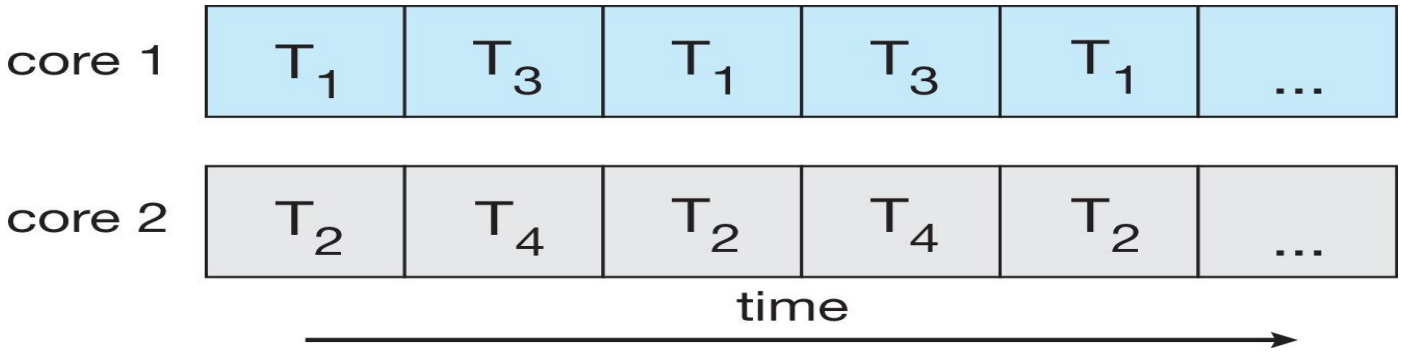
Before the advent of multiprocessor and multicore architectures, most computer systems had only a single processor, and CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

Concurrency vs. Parallelism

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



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



Five areas of challenges in programming for multicore systems

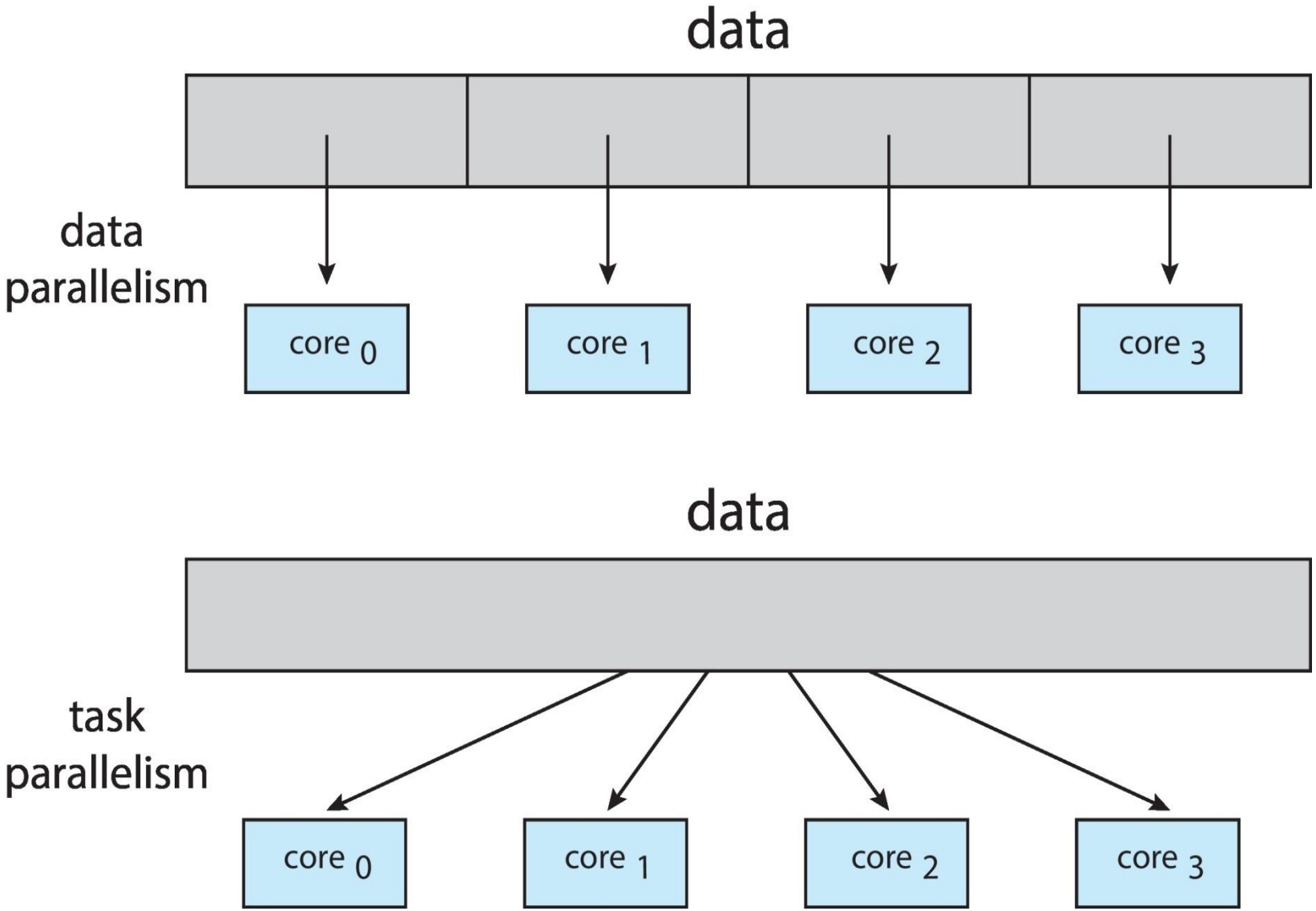
1. **Identifying tasks:** This involves examining applications to find areas that can be divided into separate, concurrent tasks.
2. **Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
3. **Data splitting:** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency:** The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
5. **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

Types of parallelism - Data and Task Parallelism

Data parallelism – **distributes subsets of the same data across multiple cores, same operation on each**. Consider, for example, summing the contents of an array of size N. **On a single-core system, one thread would simply sum the elements [0] . . . [N − 1].** On a **dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1].** The two threads would be running in parallel on separate computing cores.

Task parallelism – **distributing threads across cores**, **each thread performing unique operation.** Different threads may be **operating on the same data or operating on different data**. Consider, for example, **summing the contents of an array of size N and multiplying the contents of an array of size N .** A task parallelism might involve two threads, each performing a **unique statistical operation** on the array of elements. The **threads again are operating in parallel on separate computing cores, but each is performing a unique operation.**

Types of parallelism - Data and Task Parallelism(contd)



AMDAHL'S LAW

Amdahl's Law is a **formula that identifies potential performance gains from adding additional computing cores** to an application that has both serial (nonparallel) and parallel components.

If ***S is the portion of the application* that must be performed serially** on a system with ***N processing cores,*** *the* formula appears as follows:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Speedup $\leq 1/(0.25 + ((1-0.25)/2))$
$\leq 1/(0.25+0.375)$
≤ 1.6

As an example, assume an application that is **75 percent parallel and 25 percent serial.** Run this application on a system with two processing cores, get a speedup of **1.6** times. If add two additional cores (for a total of four), the speedup is 2.28 times.

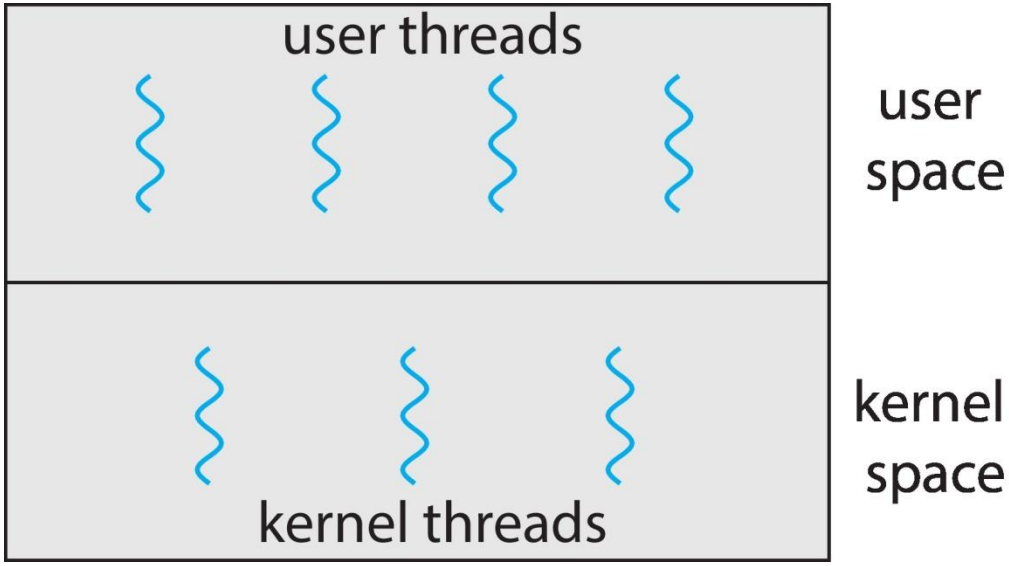
User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- **Three primary thread libraries**:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

Kernel threads - are supported by and managed directly by the OS

- Examples – virtually all general-purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads

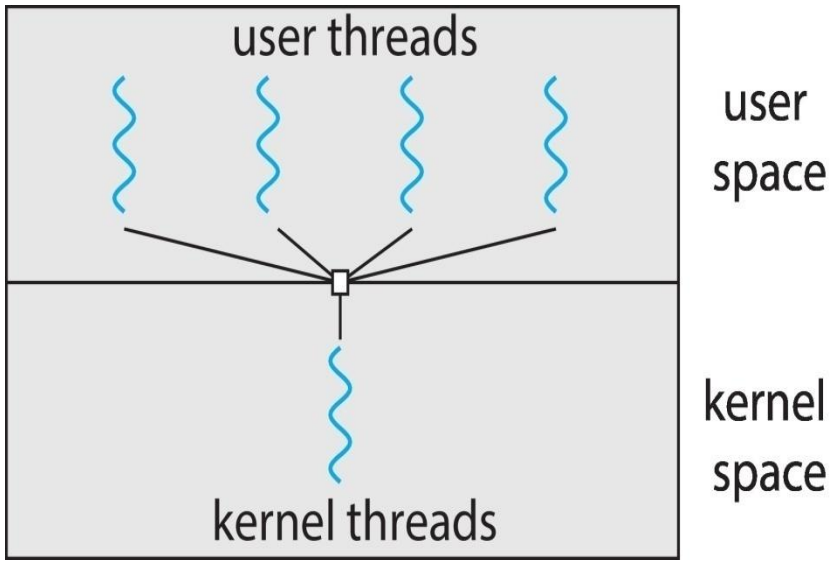


Multithreading Models

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

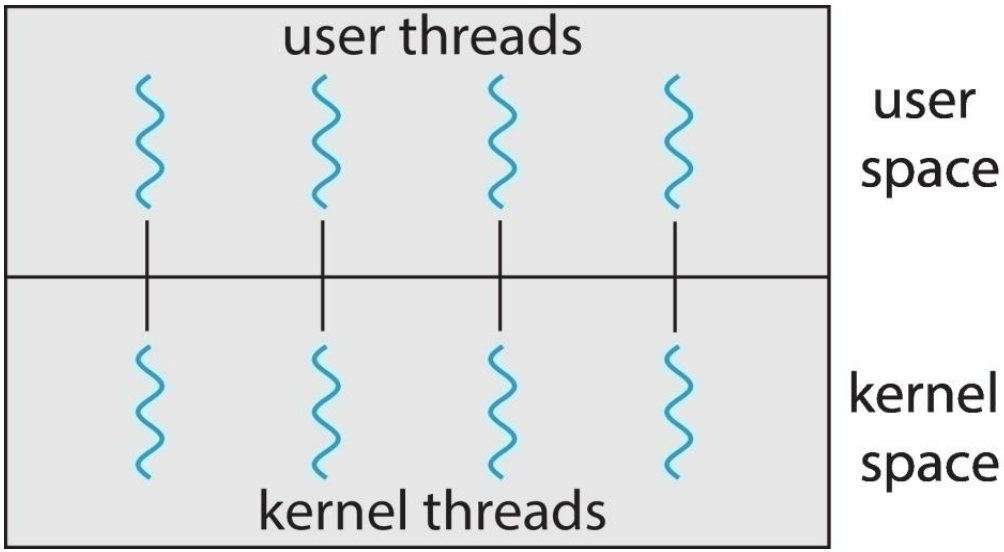
Many-to-One

- **Many user-level threads mapped to single kernel thread**
- **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**
  - **GNU Portable Threads**



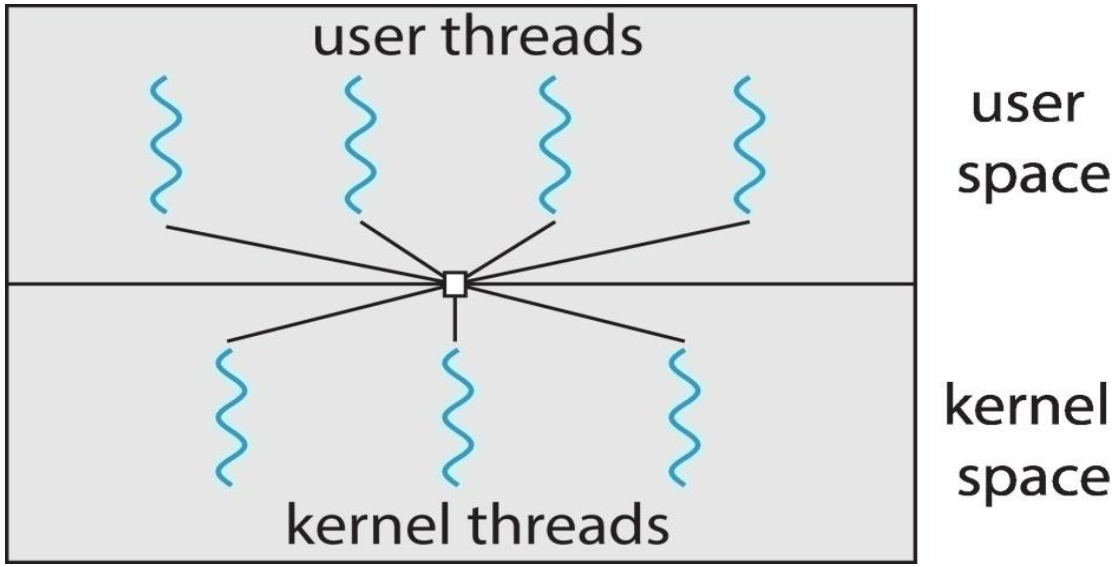
One-to-One

- **Each user-level thread maps to kernel thread**
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



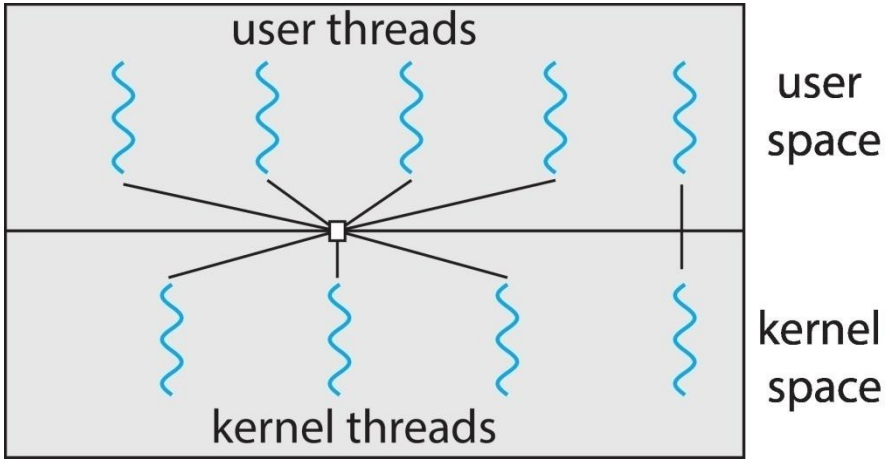
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



Two-level Model

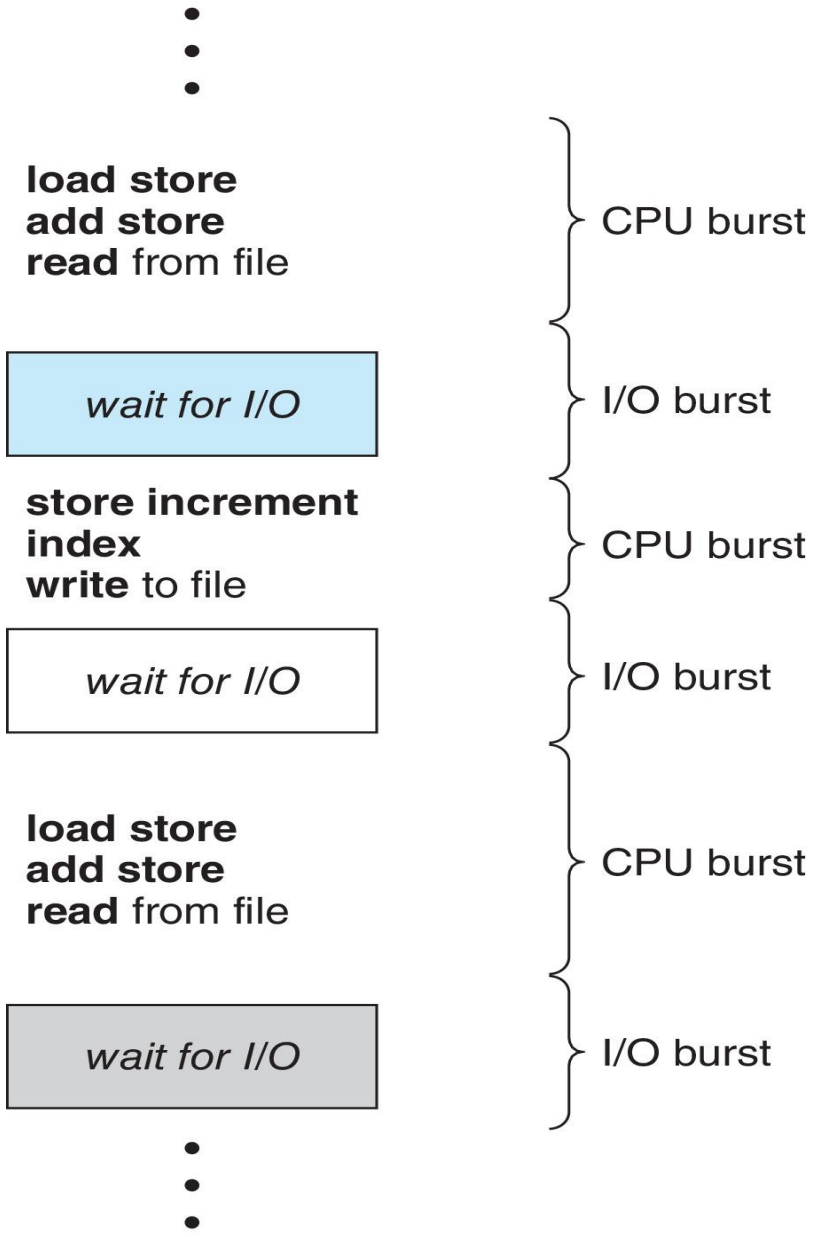
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



CPU Scheduling

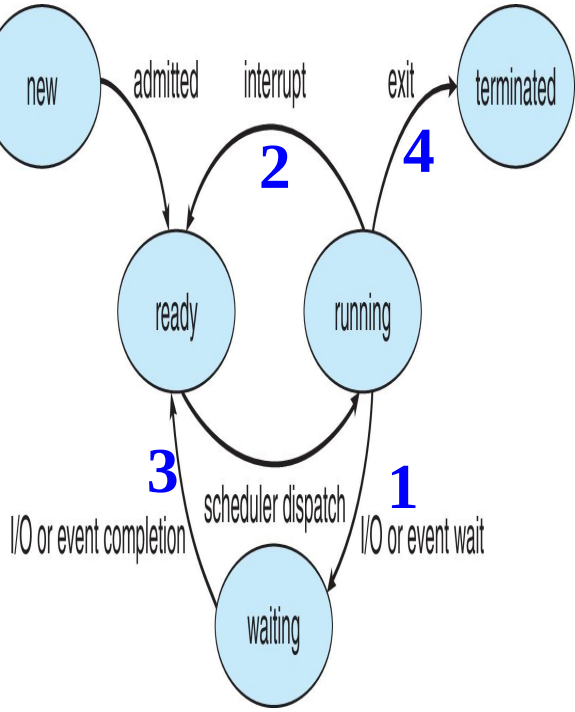
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.



Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive scheduling, **once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.**
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Preemptive Scheduling & Non Preemptive Scheduling

Preemptive Scheduling is a CPU scheduling technique that works by dividing time slots of CPU to a given process. The time slot given might be able to complete the whole process or might not be able to it. When the burst time of the process is greater than CPU cycle, it is placed back into the ready queue and will execute in the next chance. This scheduling is used when the process switch to ready state.

- **Algorithms using preemptive Scheduling :** round-robin (RR), priority, **shortest remaining time first(SRTF)**

Non-preemptive Scheduling is a CPU scheduling technique the process takes the resource (CPU time) and holds it till the process gets terminated or is pushed to the waiting state. **No process is interrupted until it is completed**, and after that processor switches to another process.

Algorithms using non-preemptive Scheduling: FCFS, non-preemptive priority, and **SJF (shortest Job first).**

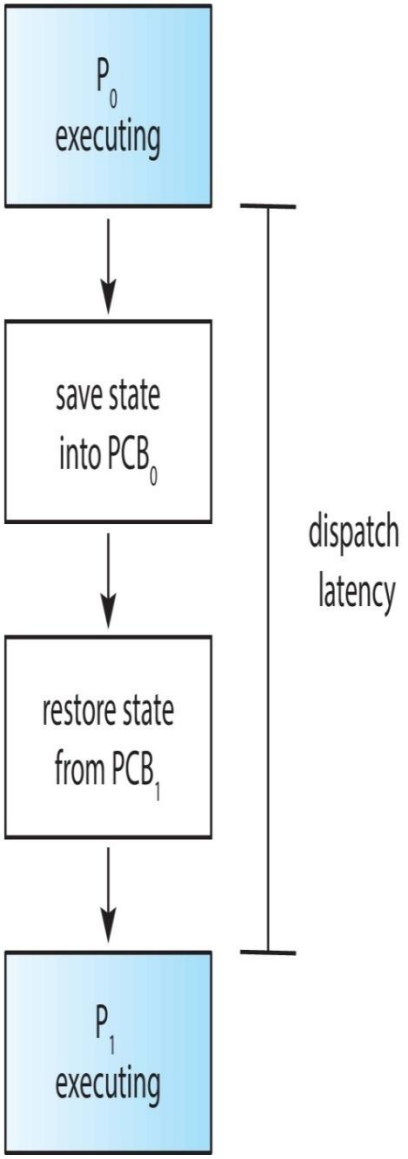
Preemptive Scheduling and Race Conditions

- **Preemptive scheduling can result in race conditions** when data are shared among several processes.

- **Race Condition:** Consider the case of two processes that share data. **While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – **time** it takes for the dispatcher **to stop one process and start another running**



Scheduling Criteria

- **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.

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Formulas for calculating Turnaround Time, Waiting Time and Response Time

Turnaround Time= Finishing time minus arrival time

Waiting Time = Turnaround Time minus Burst Time

Response Time = Time it started Executing – arrival Time

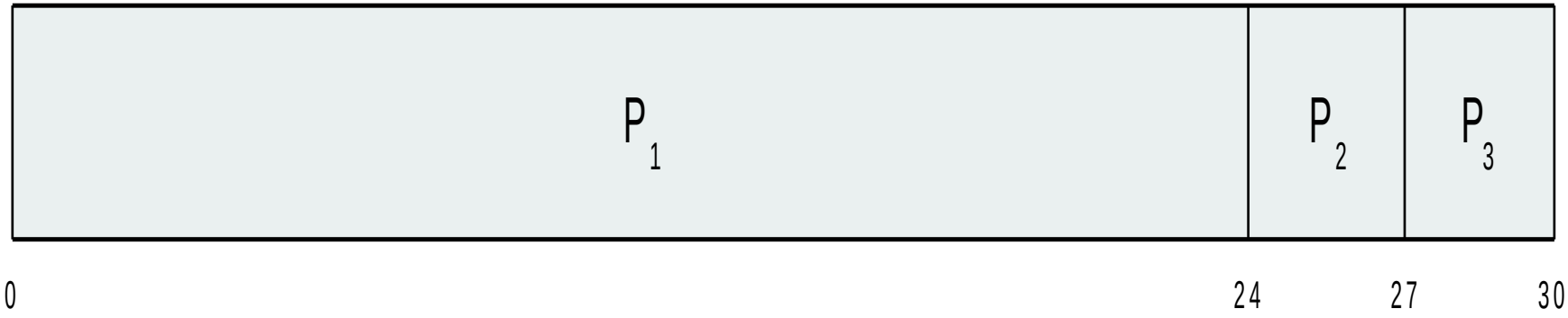
First- Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

1. First- Come, First-Served (FCFS) Scheduling

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 is:



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

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect - all the other processes wait for the one big process to get off the CPU.** This **effect results in lower CPU and device utilization** than might be possible if the shorter processes were allowed to go first.

FCFS Drawbacks :

1. Once the **CPU has been allocated to a process, that process keeps the CPU until it releases the CPU**, either by terminating or by requesting I/O.
2. The FCFS algorithm is thus **particularly troublesome for interactive systems**, where it is important **that each process get a share of the CPU at regular intervals**. It would be disastrous to allow one process to keep the CPU for an extended period.
3. **Lower CPU and device utilization**

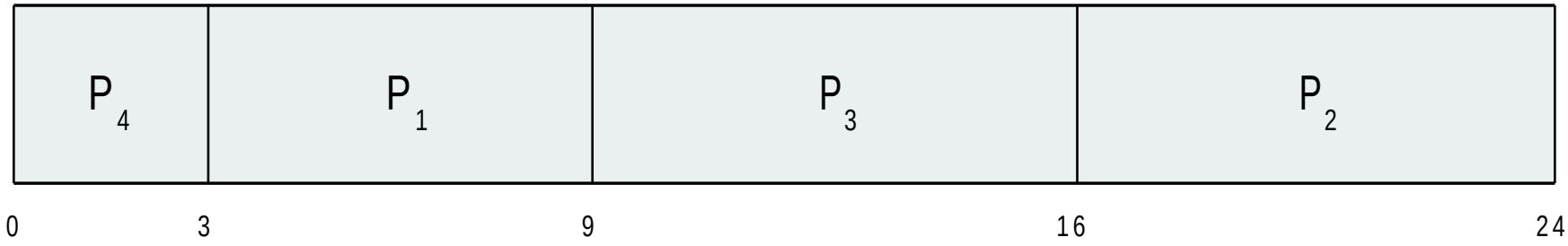
2. Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- **How do determine the length of the next CPU burst?**
  - Could ask the user
  - Estimate

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

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

- Commonly, α set to ½

3. Shortest Remaining Time First Scheduling

- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRTF more “optimal” than SJF in terms of the minimum average waiting time for a given set of processes?

Turnaround Time= Finishing time minus arrival time

Waiting Time = Turnaround Time minus Burst Time

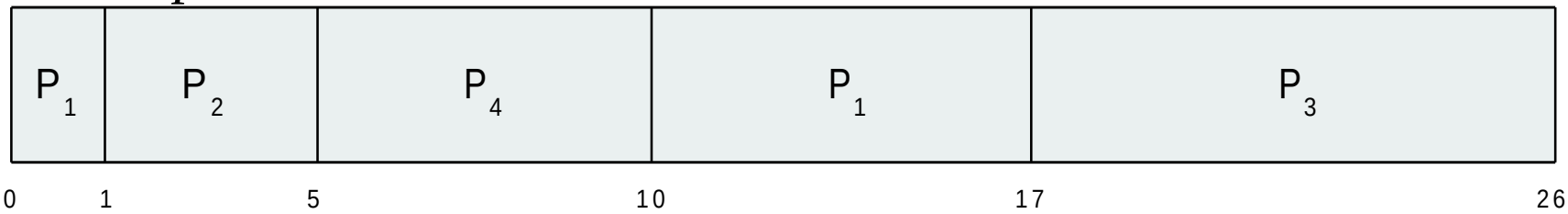
Response Time = Time it started Executing – arrival Time

Example of Shortest-remaining-time-first

- Now add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF Gantt Chart



Turnaround Time = finishing time minus arrival time

Average Turnaround Time =[(17-0) + (5-1) + (26-2) + (10-3)]/4 = 13

Waiting Time = Turnaround Time minus Burst Time

- Average waiting time = [(17-8)+(4-4)+(24-9)+(7-5)]/4 = 26 /4 = 6.5

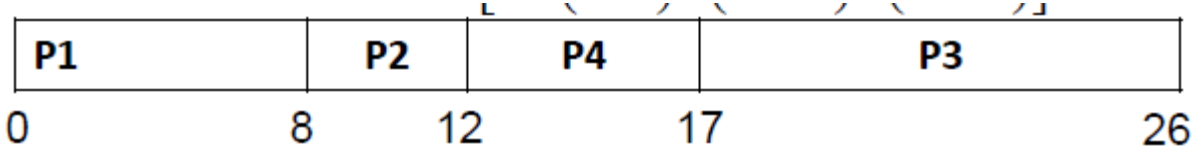
Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Turnaround Time = finishing time minus arrival time

= [(8-0) +(12-1) + (17-3) +(26-2)]/4 = 14.25

Waiting Time = Turnaround Time minus Burst Time

[(8-8) + (11-4)+(14-9)+(24-5)]/4 = 7.75



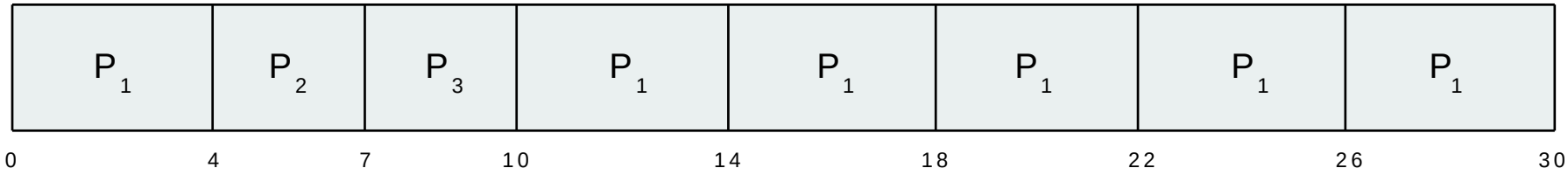
4. Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum *q***), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO (FCFS)
  - *q* small $\Rightarrow$ RR
- Note that **q must be large with respect to context switch, otherwise overhead is too high**

Example of RR with Time Quantum = 4

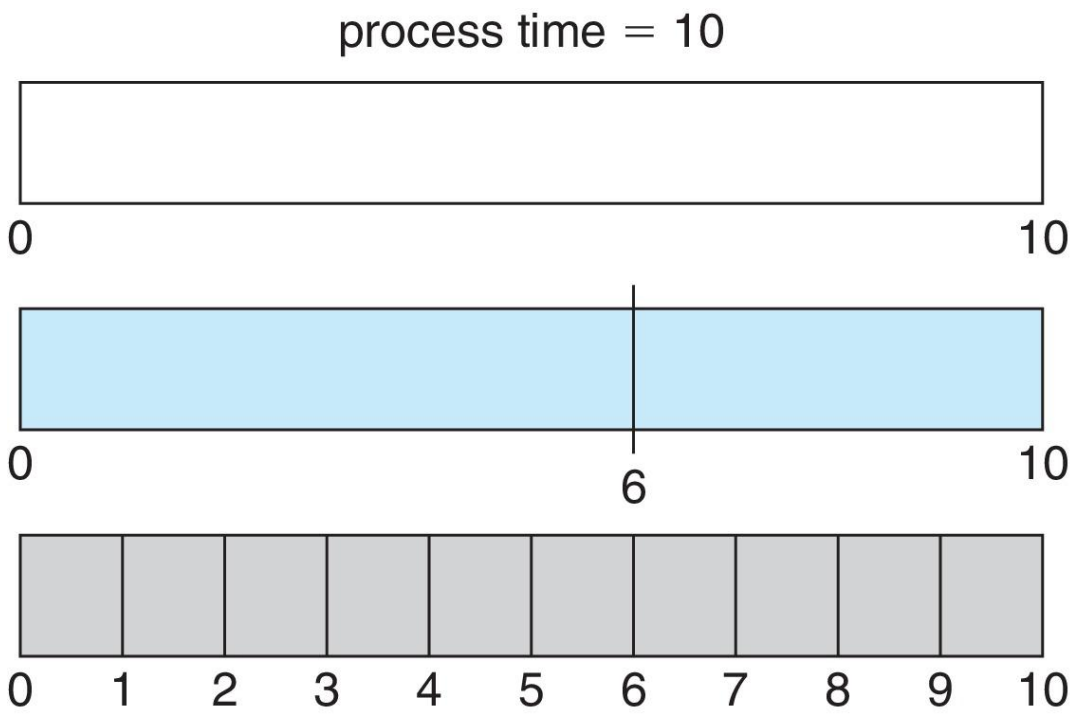
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Average waiting time=[(30-24)+(7-3)+(10-3)]/3=17/3= 5.66mseconds
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

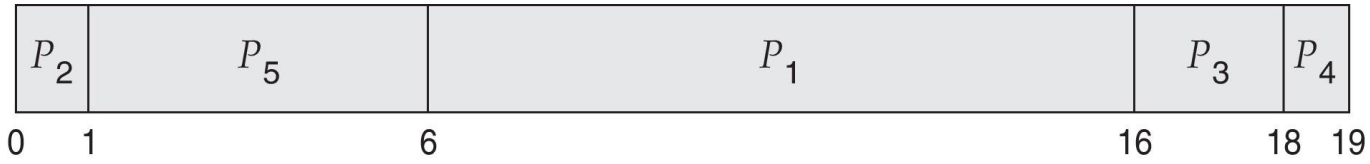
5. Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

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

- Priority scheduling Gantt Chart



- Average waiting time = [6+0+16+18+1]/5 = 41/5 = 8.2

6. Priority Scheduling with Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

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

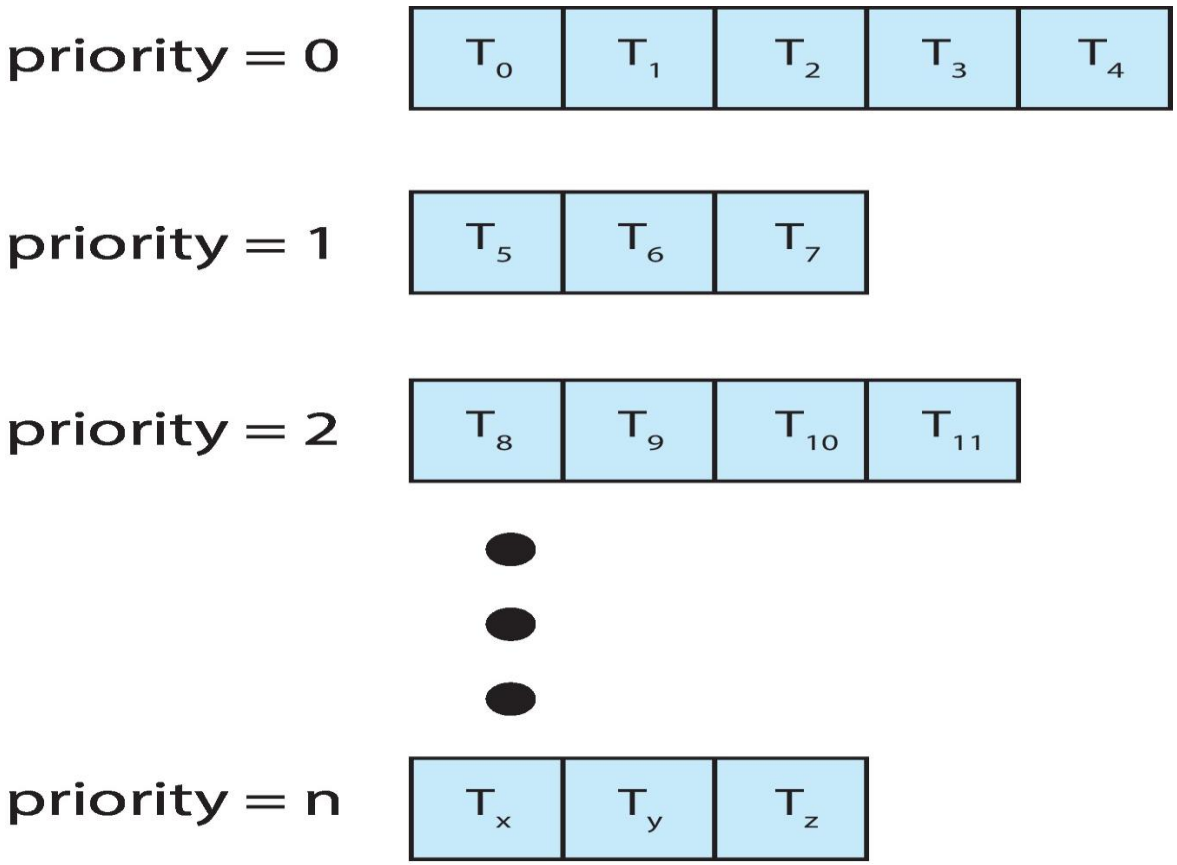
- Gantt Chart with time quantum = 2

7. Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues

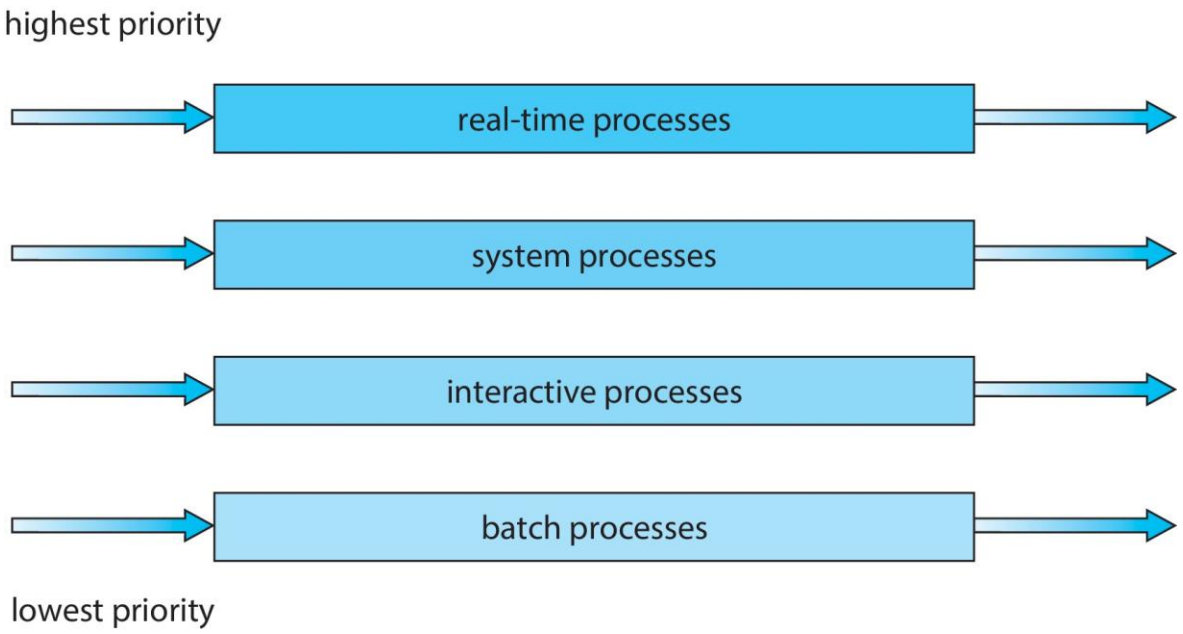
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

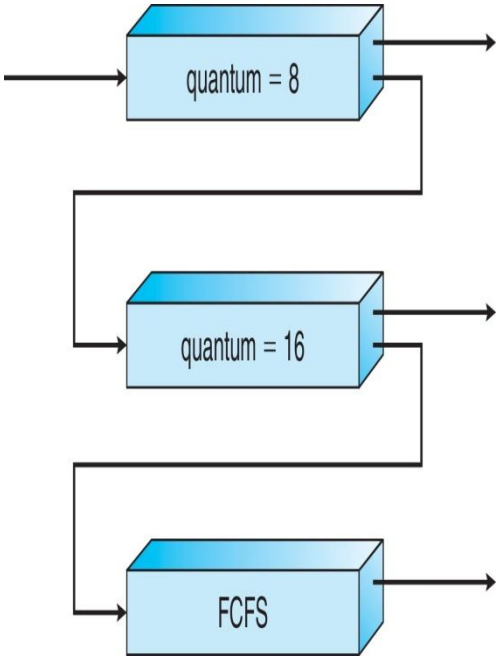


Multilevel Feedback Queue

- A process can move between the various queues.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

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



Q1. Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

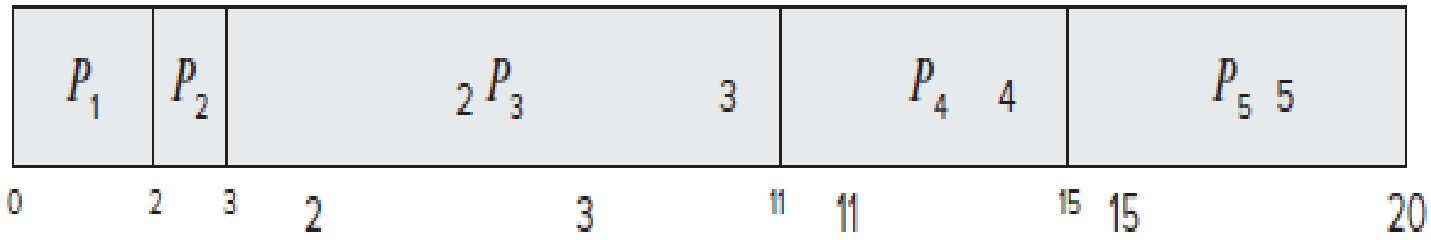
a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non- preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a? Also find average turnaround time?

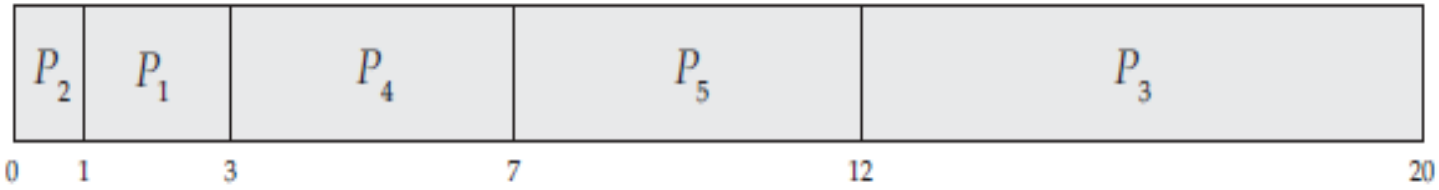
c. What is the waiting time of each process for each of these scheduling algorithms? Also find average waiting time?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

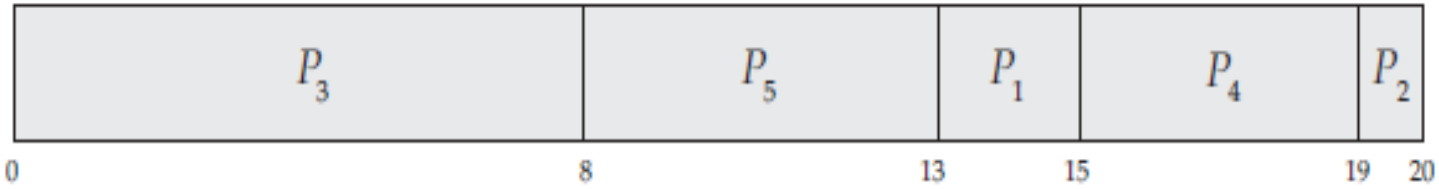
Gantt charts



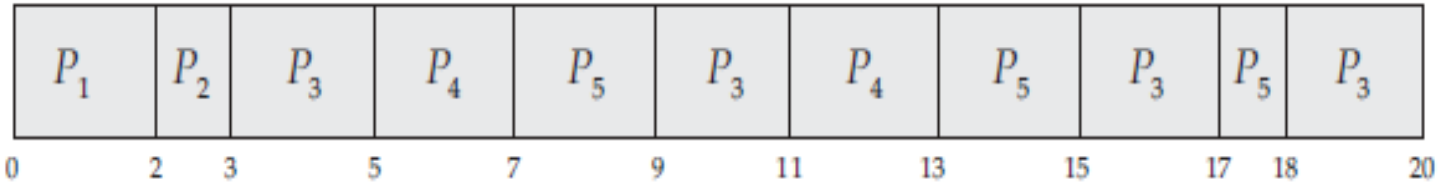
FCFS



SJF



Non Preemptive Priority



Round Robin

b. Turnaround time: It is finishing time minus arrival time

	FCFS	SJF	Priority	RR
P_1	2	3	15	2
P_2	3	1	20	3
P_3	11	20	8	20
P_4	15	7	19	13
P_5	20	12	13	18

Average turnaround time :

FCFS : [2+3+11+15+20]/5 = 10.2

SJF : [3+1+20+7+12]/5 = 8.6

Non- preemptive priority : [15+20+8+19+13]/5 = 15

Round Robin : [2+3+20+13+18]/5 = 11.2

c. Waiting time (turnaround time minus burst time):

	FCFS	SJF	Priority	RR
P_1	0	1	13	0
P_2	2	0	19	2
P_3	3	12	0	12
P_4	11	3	15	9
P_5	15	7	8	13

Average waiting time :

FCFS : [0+2+3+11+15] /5 = 6.2

SJF : [1+0+12+3+7]/5 = 4.6

Non- preemptive priority : [13+19+0+15+8]/5 = 11

Round Robin : [0+2+12+9+13]/5 = 7.2

d. SJF has the shortest wait time.

Q2.Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	**Arrival Time**	**Burst Time**
P1	***0.0***	***8***
P2	***0.4***	***4***
P3	***1.0***	***1***

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm? Ans : 10.53

b. What is the average turnaround time for these processes with the SJF scheduling algorithm? Ans : 9.53

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process *P1 at time 0 because we did not know* that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Ans : 6.86

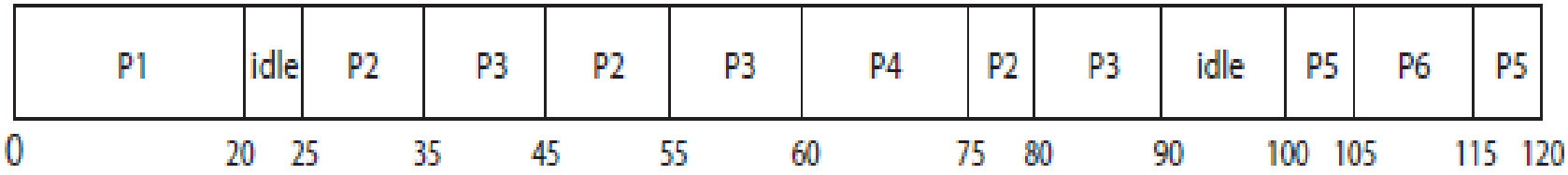
Q3. The following processes are being scheduled using a preemptive, round robin scheduling algorithm.

Process	Priority	Burst	Arrival
P1	***40***	***20***	***0***
P2	***30***	***25***	***25***
P3	***30***	***25***	***30***
P4	***35***	***15***	***60***
P5	***5***	***10***	***100***
P6	***10***	***10***	***105***

Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed above, the system also has an **idle task (which consumes no CPU resources and** is identified as *Pidle). This task has priority 0 and is scheduled whenever* the system has no other available processes to run. **The length of a time quantum is 10 units**. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.
b. What is the turnaround time for each process?
c. What is the waiting time for each process?
d. What is the CPU utilization rate?

a. The Gantt chart:



b. P1: 20-0 = 20, P2: 80-25 = 55, P3: 90 - 30 = 60, P4: 75-60 = 15,
P5: 120-100 = 20, P6: 115-105 = 10[finishing time minus arrival time]

c. P1: 20-20 =0, P2: 55-25 =30, P3: 60-25 =35, P4: 15-15=0,
P5: 20-10 =10, P6:10-10 = 0 [turnaround time minus burst time]

d. CPU utilization = CPU used time/ Total time = 105/120 = 87.5%

G2. The following C program is executed on a Unix/Linux system:

```
#include < unistd.h >
  int main ()
  {
      int i ;
      for (i=0; i<10; i++)
          if (i%2 == 0) fork ( ) ;
      return 0 ;
  }
```

The total number of child processes created is _____.

Answer : When the value of 'i' is even then fork() statement is executed.
(1) when i = 0, then fork() will executed.
(2) when i = 2, then fork() will executed.
(3) when i = 4, then fork() will executed.
(4) when i = 6, then fork() will executed.
(5) when i = 8, then fork() will executed.
So total 5 times fork() will executed. We know, **n - fork statements will create 2^n - 1 childs.** ∴ **5 fork statements will create 2^5 - 1 = 31 childs**.

G3.Consider the following statements about process state transitions for a system using preemptive scheduling.

I. A running process can move to ready state.

II. A ready process can move to ready state.

III. A blocked process can move to running state.

IV. A blocked process can move to ready state.

Which of the above statements are TRUE?

A . I, II and III only

B II and III only

C. I, II and IV only

D. I, II, III and IV

G4. Consider an arbitrary set of CPU-bound processes with unequal CPU burst lengths submitted at the same time to a computer system. Which one of the following process scheduling algorithms would minimize the average waiting time in the ready queue?

A. **Shortest remaining time first**
B. Round-robin with time quantum less than the shortest CPU burst
C. Uniform random
D. Highest priority first with priority proportional to CPU burst length

G5. Consider a uniprocessor system executing three tasks T1, T2 and T3, each of which is composed of an infinite sequence of jobs (or instances) which arrive periodically at intervals of 3, 7 and 20 milliseconds, respectively. The priority of each task is the inverse of its period and the available tasks are scheduled in order of priority, with the highest priority task scheduled first. Each instance of T1, T2 and T3 requires an execution time of 1, 2 and 4 milliseconds, respectively. Given that all tasks initially arrive at the beginning of the 1st milliseconds and task preemptions are allowed, the first instance of T3 completes its execution at the end of _____milliseconds.

(A) 5 (B) 10 **(C) 12** (D) 15

Explanation: Periods of T1, T2 and T3 are 3ms, 7ms and 20ms.
Since priority is inverse of period, T1 is the highest priority task, then T2 and finally T3. Every instance of T1 requires 1ms, that of T2 requires 2ms and that of T3 requires 4ms. Initially all T1, T2 and T3 are ready to get processor, T1 is preferred Second instances of T1, T2, and T3 shall arrive at 3, 7, and 20 respectively. Third instance of T1, T2 and T3 shall arrive at 6, 14, and 40 respectively.

Time-Interval	Tasks
0-1	T1
1-2	T2
2-3	T2
3-4	T1 [Second Instance of T1 arrives]
4-5	T3
5-6	T3
6-7	T1 [Third Instance of T1 arrives] [Therefore T3 is preempted]
7-8	T2 [Second instance of T2 arrives]
8-9	T2
9-10	T1 [Fourth Instance of T1 arrives]
10-11	T3
11-12	T3 [**First Instance of T3 completed**]

G6.Consider the following processes, with the arrival time and the length of the CPU burst given in milliseconds. The scheduling algorithm used is preemptive shortest remaining-time first.

Process	Arrival Time	Burst Time
P_1	0	10
P_2	3	6
P_3	7	1
P_4	8	3

The average turn around time of these processes is _____milliseconds.

Answer : 8.2

G7.For the processes listed in the following table, which of the following scheduling schemes will give the lowest average turnaround time?

Process	Arrival Time	Processing Time
A	0	3
B	1	6
D	4	4
E	6	2

A. First Come First Serve

B. Non-preemptive Shortest Job First

C.Shortest Remaining Time

D.Round Robin with Quantum value two

G8.Consider the following set of processes that need to be scheduled on a single CPU. All the times are given in milliseconds.

Process Name	Arrival Time	Execution Time
A	0	6
B	3	2
C	5	4
D	7	6
E	10	3

Using the shortest remaining time first scheduling algorithm, the average process turnaround time (in msec) is _____

Answer : 7.2

G9. Consider the 3 processes, P1, P2 and P3 shown in the table.

Process	Arrival Time	Time Units Required
P1	0	5
P2	1	7
P3	3	4

The completion order of the 3 processes under the policies FCFS and RR2(round robin scheduling with CPU quantum of 2 time units) are

A. FCFS:P1,P2,P3 RR2:P1,P2,P3

B. FCFS:P1,P3,P2 RR2:P1,P3,P2

C. FCFS:P1,P2,P3 RR2:P3,P1,P2

D. FCFS:P1,P3,P2 RR2:P1,P2,P3

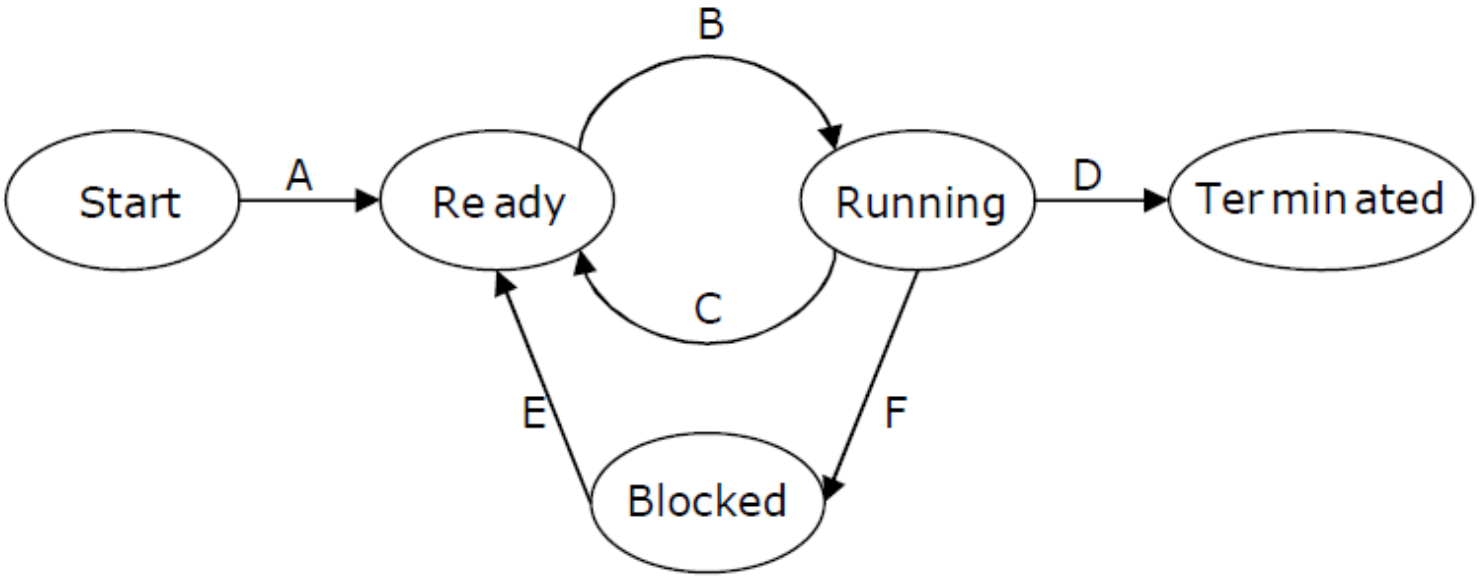
G10.Consider the following table of arrival time and burst time for three processes P0,P1 and P2

Process	Arrival Time	Burst Time
P0	**0 ms**	**9 ms**
P1	**1 ms**	**4 ms**
P2	**2 ms**	**9 ms**

The pre-emptive shortest job first scheduling algorithm is used. Scheduling is carried out only a arrival or completion of processes. What is the average waiting time for the three processes?

A.5.0 ms B. 4.33 ms C. 6.33 ms D. 7.33 ms

G11. In the following process state transition diagram for a uniprocessor system, assume that there are always some processes in the ready state: Now consider the following statements:



I. If a process makes a transition D, it would result in another process making transition A immediately.
II. A process P2 in blocked state can make transition E while another process P1 is in running state.
III. The OS uses preemptive scheduling.
IV. The OS uses non-preemptive scheduling.
Which of the above statements are TRUE?
(A) I and II (B) I and III **(C) II and III** (D) II and IV

G12.Three processes A, B and C each execute a loop of 100 iterations. In each iteration of the loop, a process performs a single computation that requires tc CPU milliseconds and then initiates a single I/O operation that lasts for tio milliseconds. It is assumed that the computer where the processes execute has sufficient number of I/O devices and the OS of the computer assigns different I/O devices to each process. Also, the scheduling overhead of the OS is negligible. The processes have the following characteristics:

Process id	**tc**	**tio**
A	100 ms	500 ms
B	350 ms	500 ms
C	200 ms	500 ms

The processes A, B, and C are started at times 0, 5 and 10 milliseconds respectively, in a pure time sharing system (round robin scheduling) that uses a time slice of 50 milliseconds. The time in milliseconds at which process C would complete its first I/O operation is_____.
(A) 500 **(B) 1000** (C) 2000 (D) 10000

Explanation:

There are three processes A, B and C that run in round robin manner with time slice of 50 ms.

Processes start at 0, 5 and 10 milliseconds.

The processes are executed in below order

A, B, C, A

50 + 50 + 50 + 50 (200 ms passed)

Now A has completed 100 ms of computations and goes for I/O now

B, C, B, C, B, C

50 + 50 + 50 + 50 + 50 + 50 (300 ms passed)

C goes for i/o at 500ms and it needs 500ms to finish the IO.

So C would complete its first IO at 1000 ms

G13. Consider four processes P, Q, R and S scheduled on a CPU as per round robin algorithm with a time quantum of 4 units. The processes arrive in the order P, Q, R, S, all at time t = 0. There is exactly one context switch from S to Q, exactly one context switch from R to Q, and exactly two context switches from Q to R. There is no context switch from S to P. Switching to a ready process after the termination of another process is also considered a context switch. Which one of the following is NOT possible as CPU burst time (in time units) of these processes?

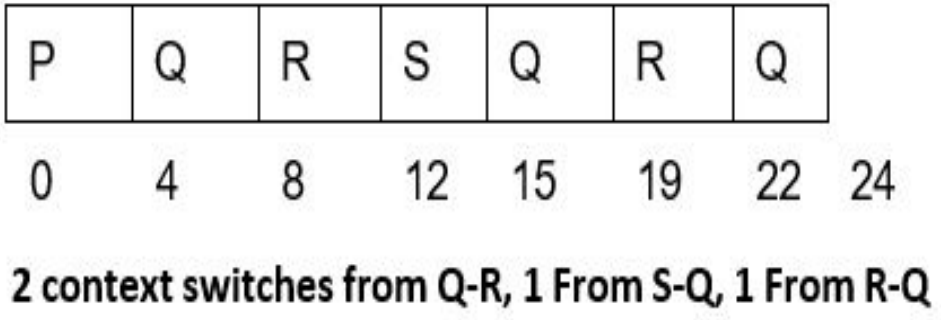
(A) P = 4, Q = 10, R = 6, S = 2

(B) P = 2, Q = 9, R = 5, S = 1

(C) P = 4, Q = 12, R = 5, S = 4

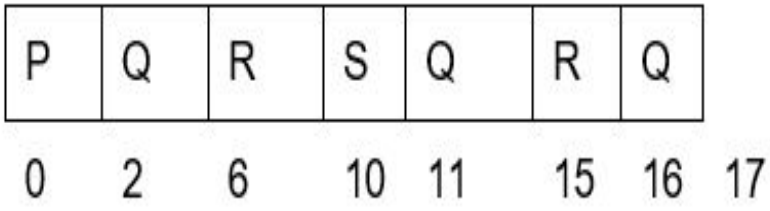
(D) P = 3, Q = 7, R = 7, S = 3

Option A



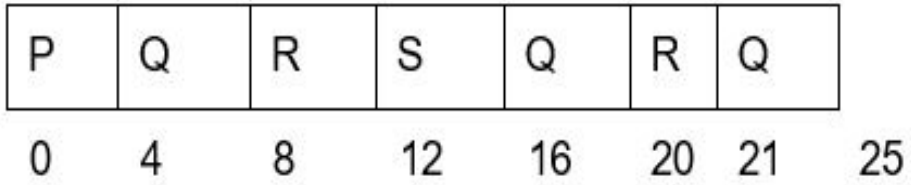
2 context switches from Q-R, 1 From S-Q, 1 From R-Q

Option B



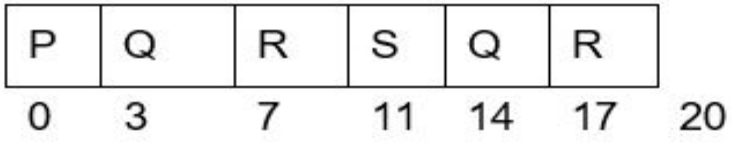
2 context switches from Q-R, 1 From S-Q, 1 From R-Q

Option C



2 context switches from Q-R, 1 FROM S-Q, 1 FROM R-Q

Option D



NOT valid here.

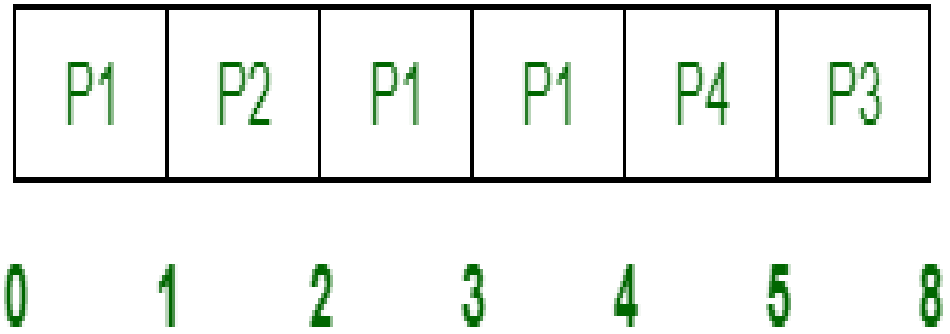
G14. Consider the following four processes with arrival times (in milliseconds) and their length of CPU burst (in milliseconds) as shown below:

Process	P1	P2	P3	P4
Arrival time	0	1	3	4
CPU burst time	3	1	3	Z

These processes are run on a single processor using preemptive Shortest Remaining Time First scheduling algorithm. If the average waiting time of the processes is 1 millisecond, then the value of Z is _____.

(A) 2 (B) 3 (C) 1 (D) 4

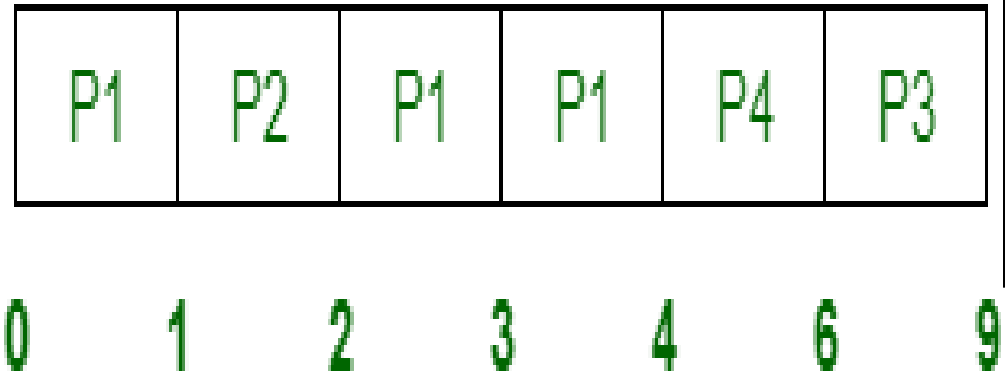
Let Z = 1, then Gantt chart will be,



Average waiting Time

={(4-0-3) + (2-1-1) + (8-3-3) + (5-4-1)} / 4 = (1 + 0 + 2 + 0) / 4 = 3 / 4 = 0.75

Let Z = 2, then Gantt chart will be



Average waiting time

= {(4-0-3) + (2-1-1) + (9-3-3) + (6-4-2)} / 4 = (1 + 0 + 3 + 0) / 4 = 4 / 4 = 1

G14. Consider the following set of processes, with the length of the CPU burst time given in milliseconds: Find the average turnaround time, waiting time and response time.

Process	Arrival Time	Burst Time
P1	0	11
P2	2	4
P3	3	2

Average turnaround time = 12.66ms
Average waiting time =7ms
Average response time = 7ms