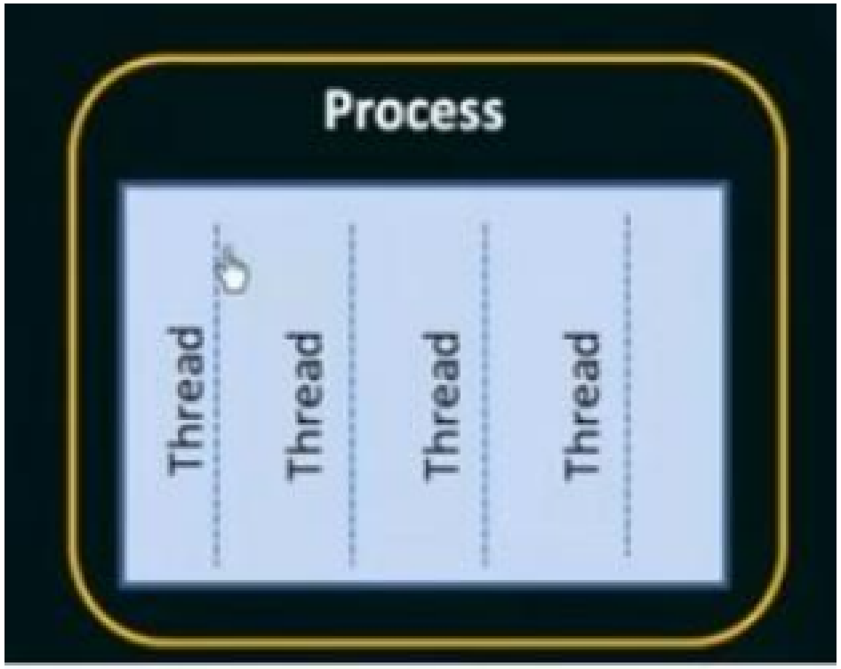
Unit II- Process Management J.Premalatha Professor/IT Kongu Engineering College Perundurai

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 rocess.** So a **rocess is a roram in execution and it is active entit.**
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

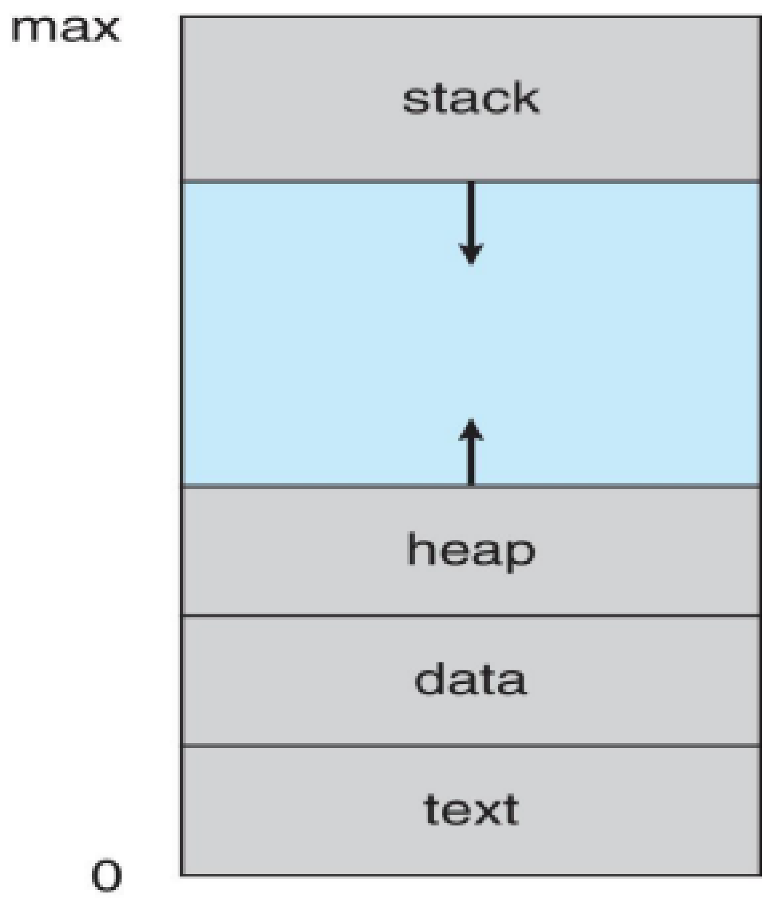
Data section —**global variables**

Heap section —memory that is dynamically allocated during program run time

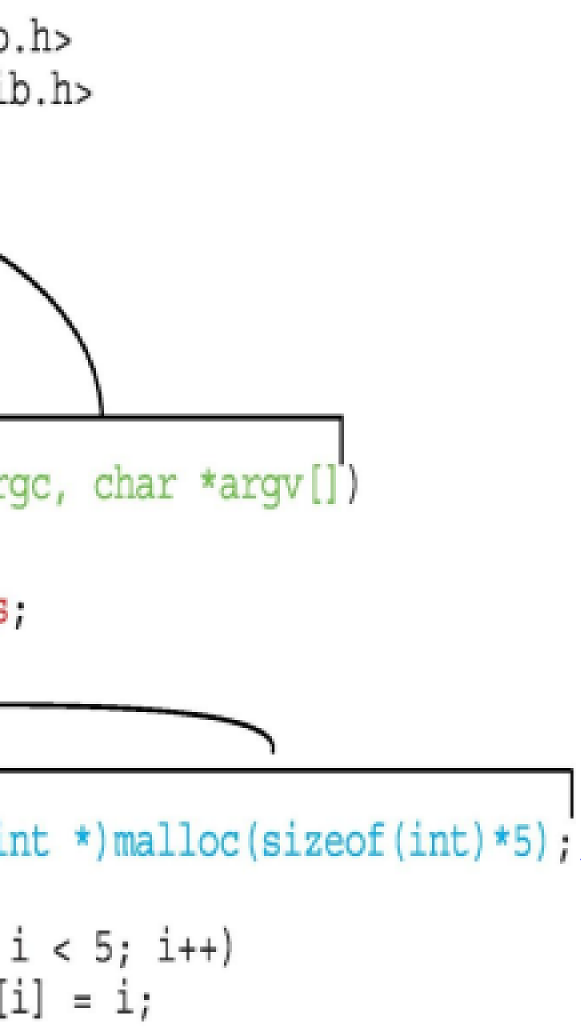
Stack section — **temporary data storage** when invoking functions (such as **function parameters, return addresses**, and **local variables**)

program counter, processor registers

Process in Memory



Memory Layout of a C Program



```
.h>
ib.h>

rgc, char *argv[])

s;

int *)malloc(sizeof(int)*5);

 i < 5; i++)
[i] = i;
```

function parameters, return addresses, local variables

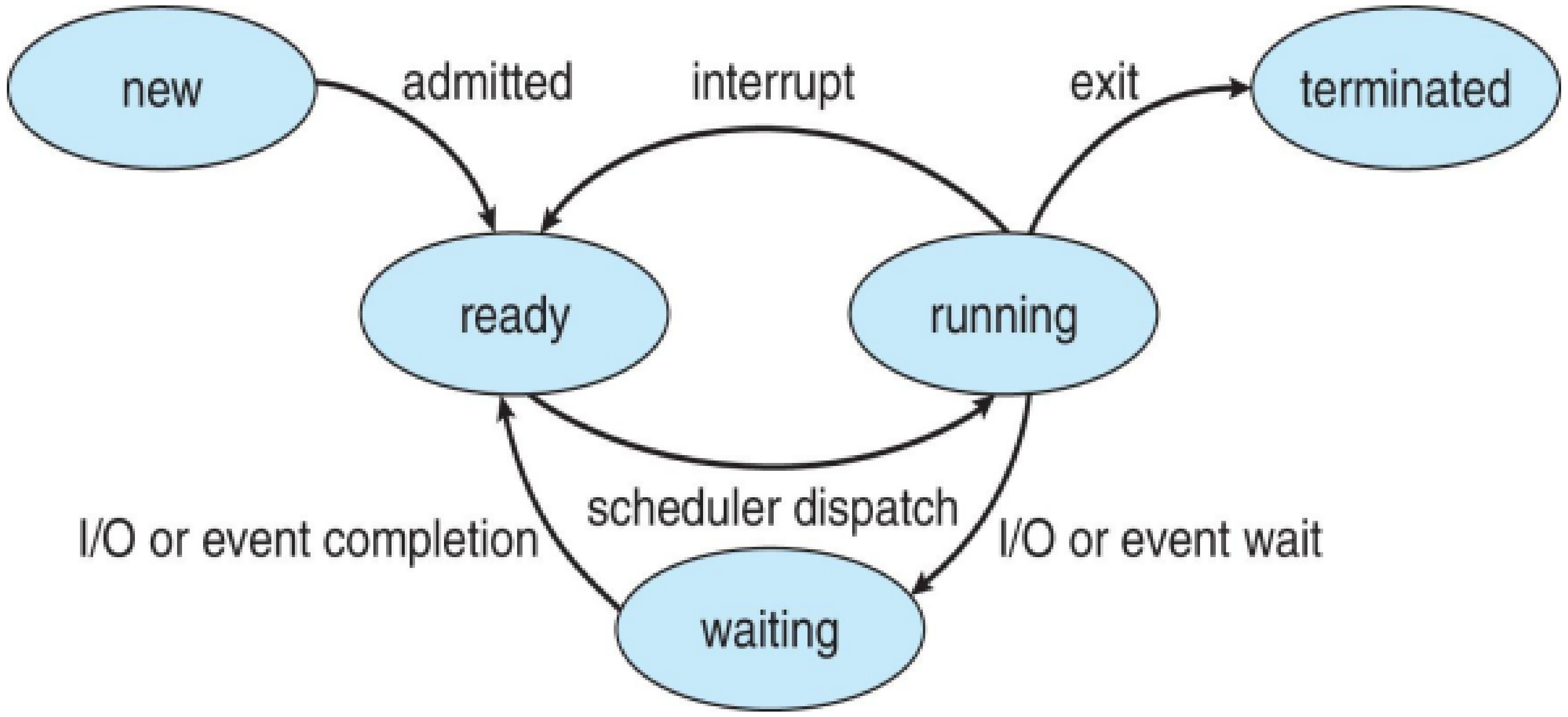
Global variables

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



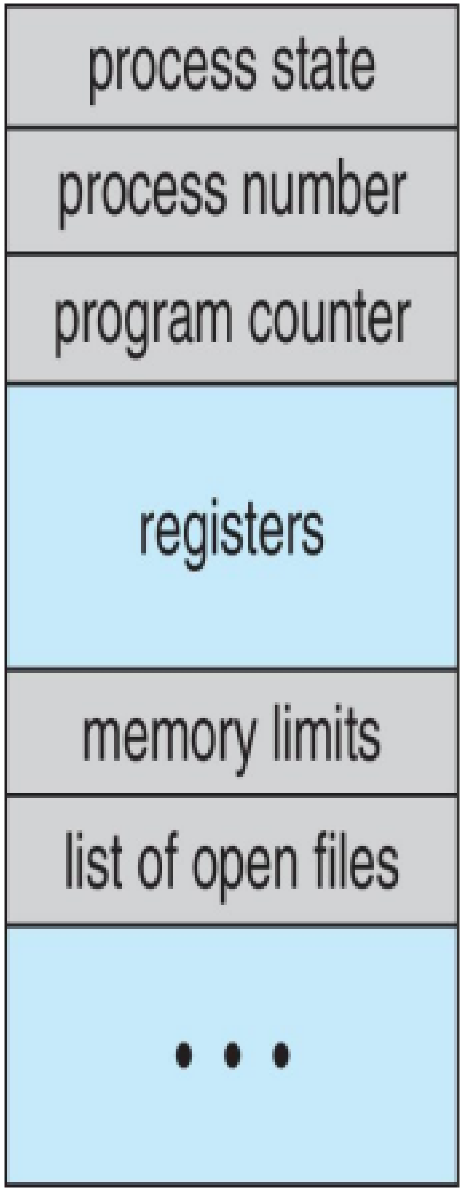
Information associated with each process is in Process Control block

Process Control Block (PCB)

(also called **task control block**)

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

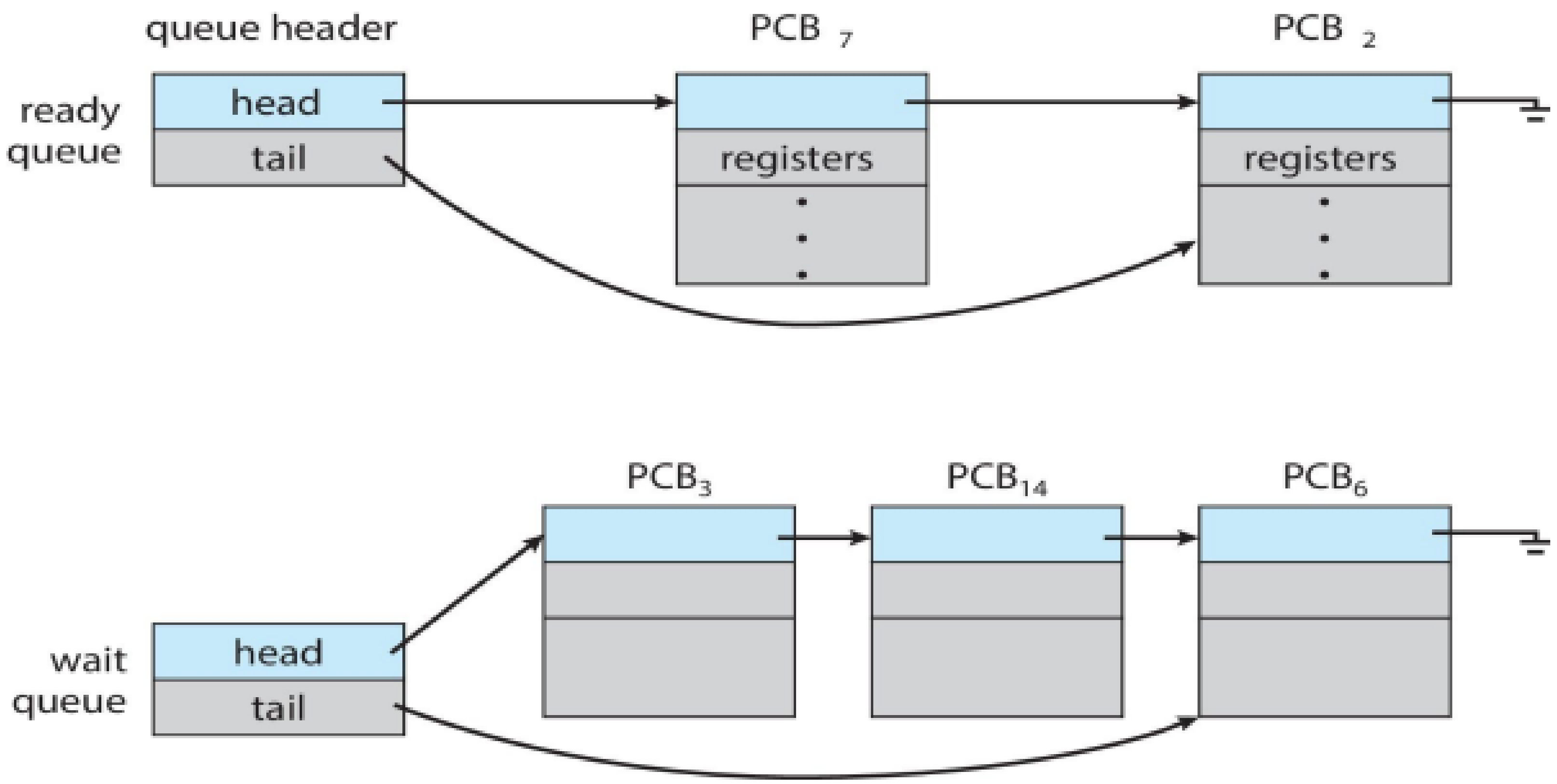
- **Process state** – running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to 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 state
process number
program counter
registers
memory limits
list of open files
• • •

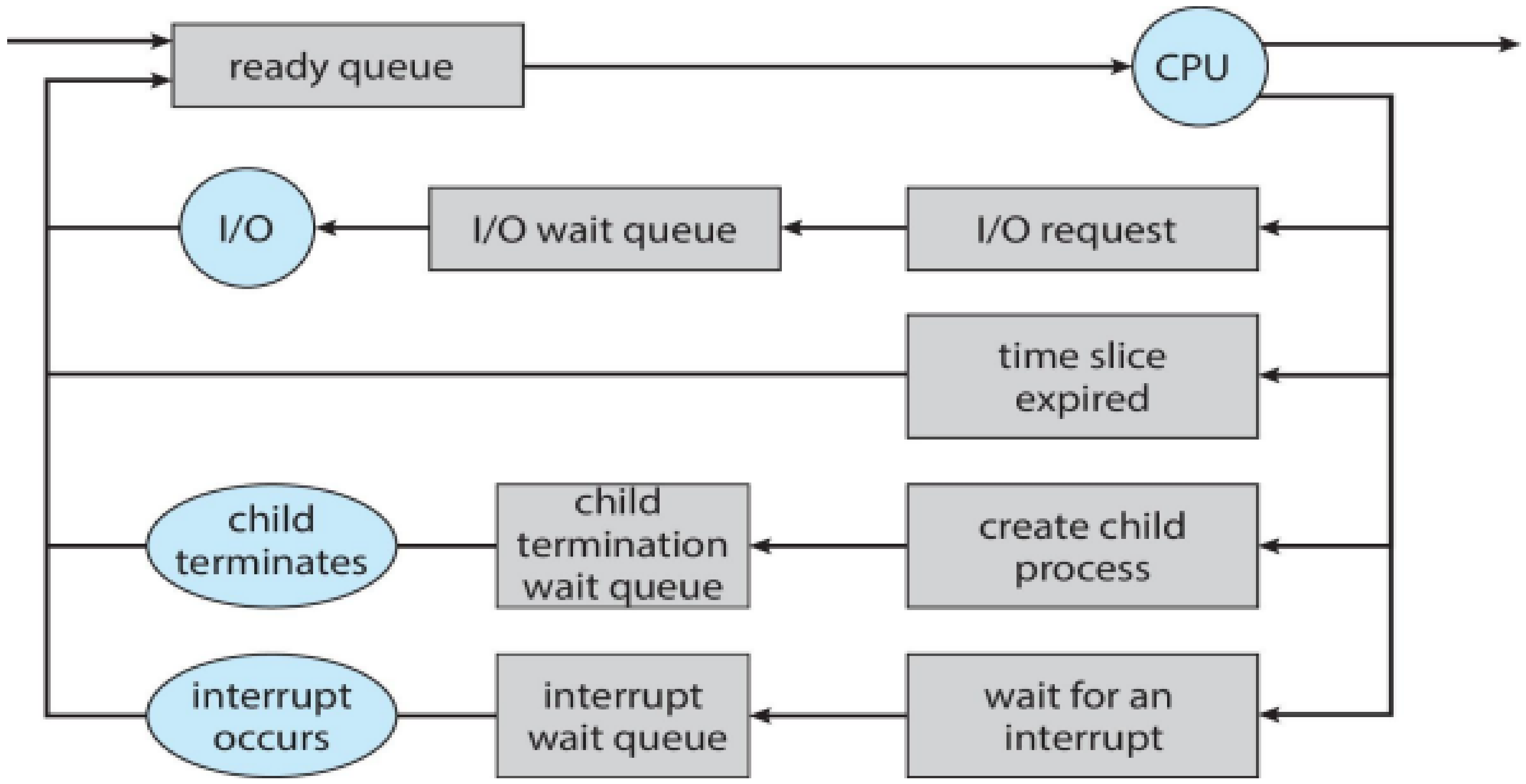
Process Scheduling

- **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

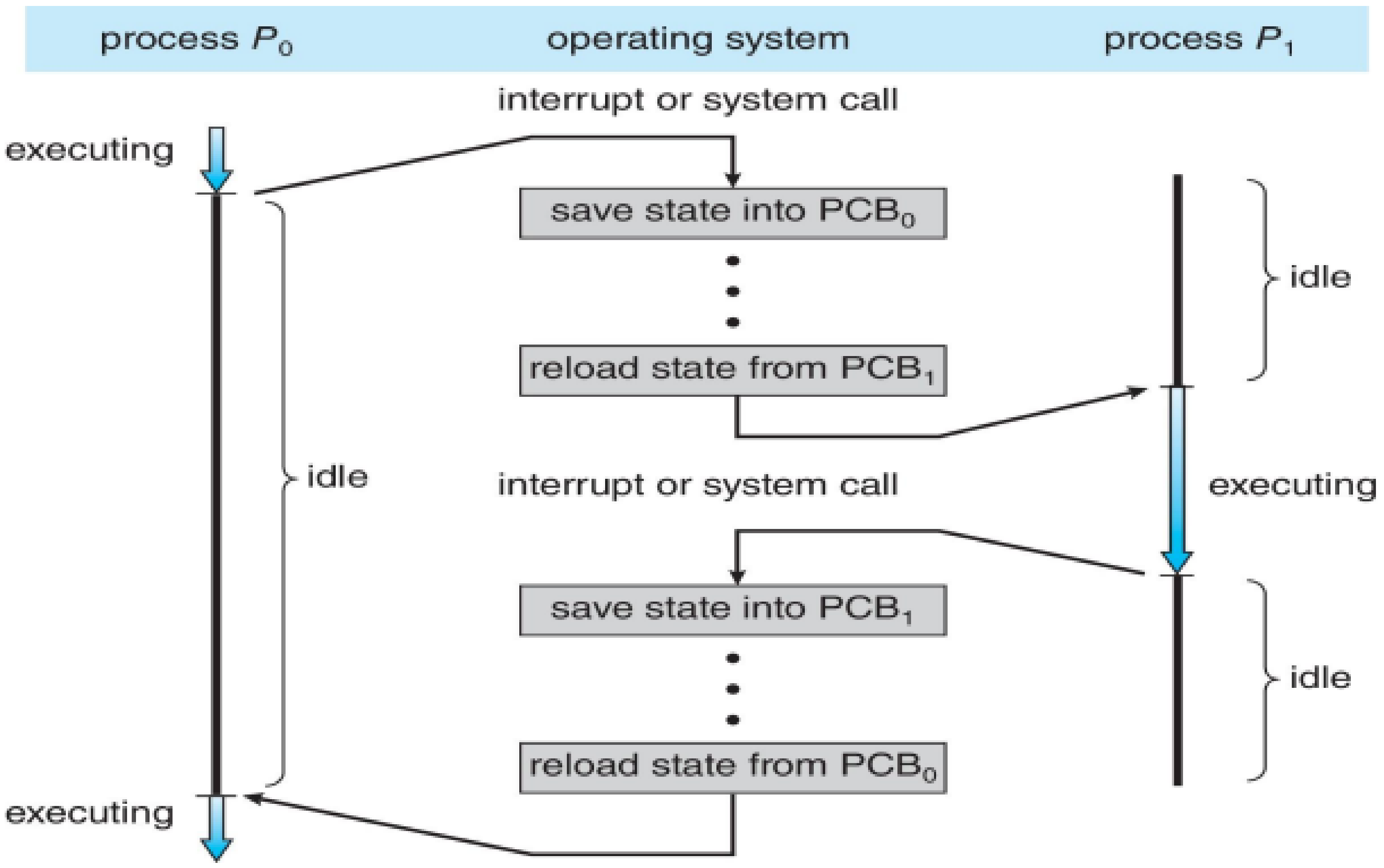


Representation of Process Scheduling



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.**

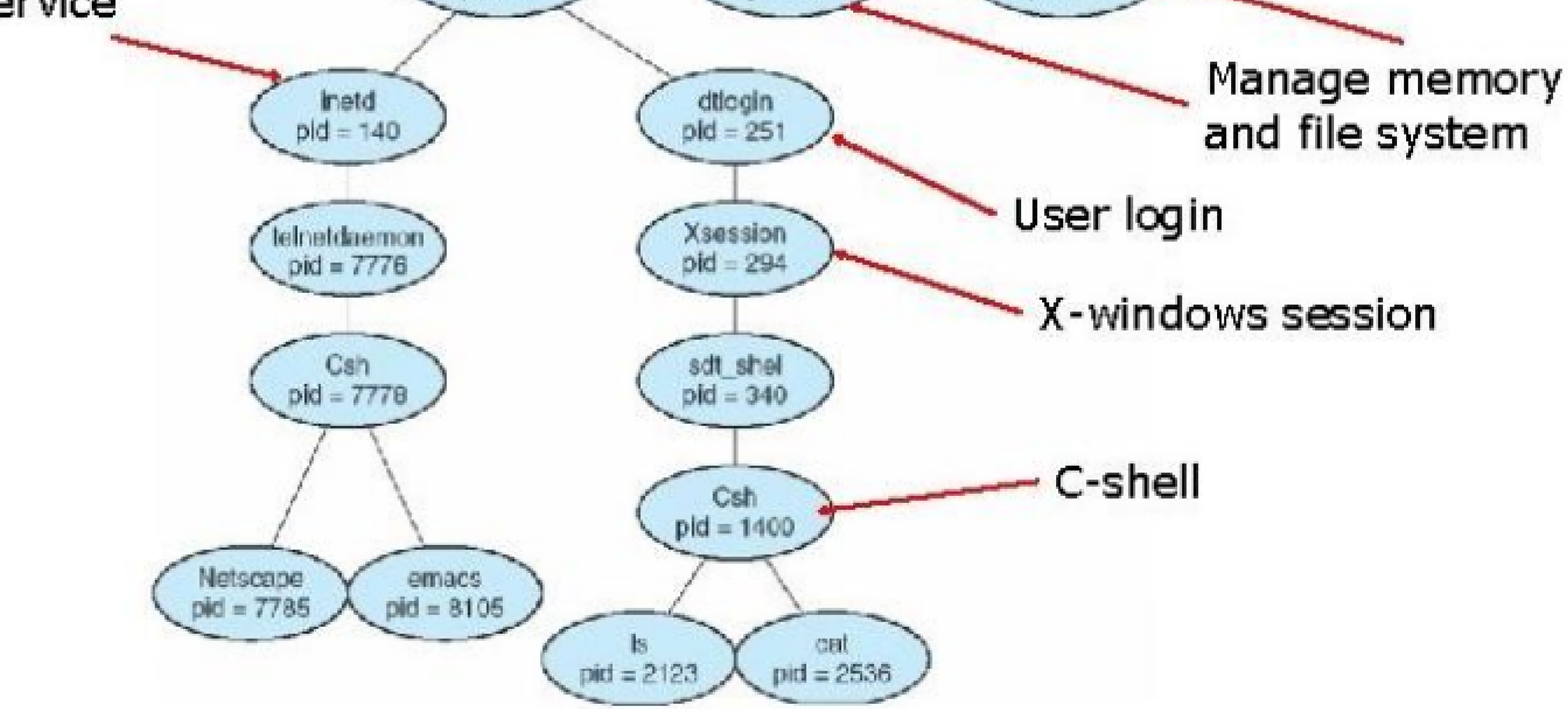
• Once the system has booted, the **systemd process creates**

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



Telnet

C-shell

Netscape Navigator

Used for creating

files , read

Emac word editor

List command

Cat command

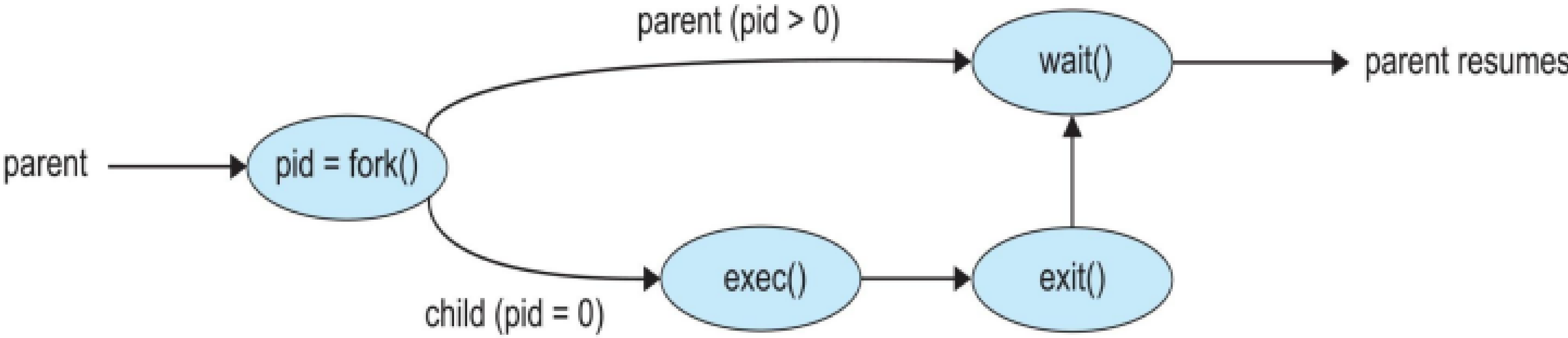
the file contents

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



– **exit() system call** delete the process

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**

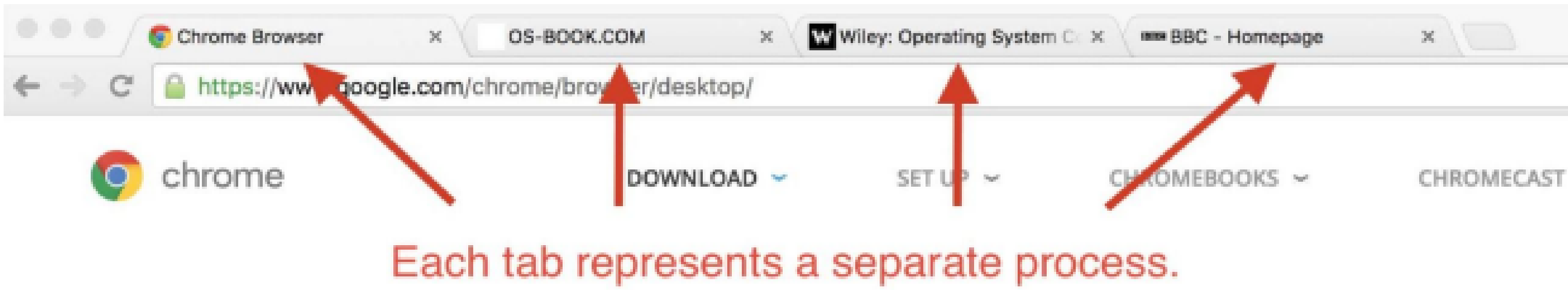
– **Empty process**

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


Chrome Browser
OS-BOOK.COM
Wiley: Operating System C
BBC - Homepage
https://www.google.com/chrome/browser/desktop/
chrome
DOWNLOAD
SET UP
CHROMEBOOKS
CHROMECAST
Each tab represents a separate process.

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

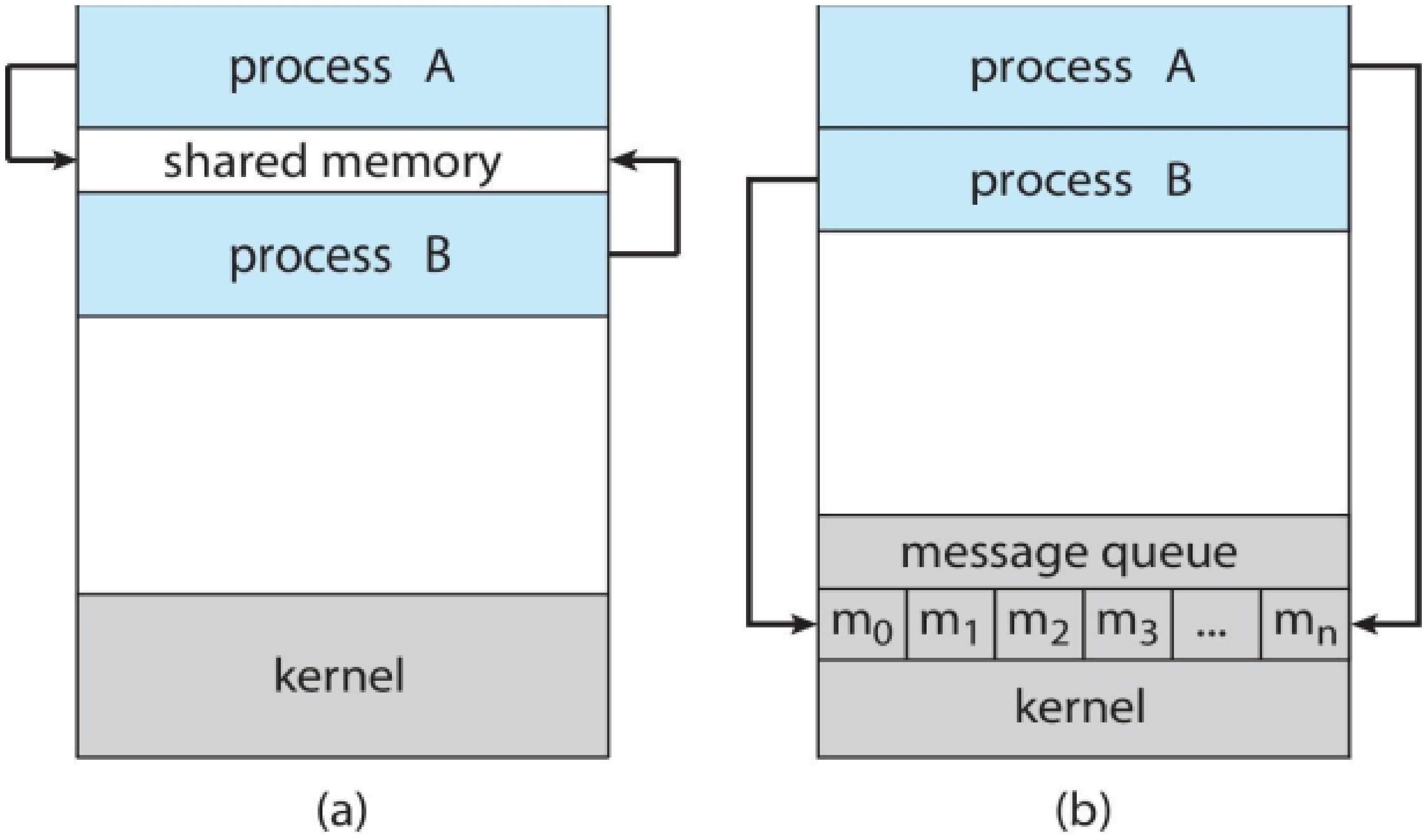
– Computation speedup

– 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

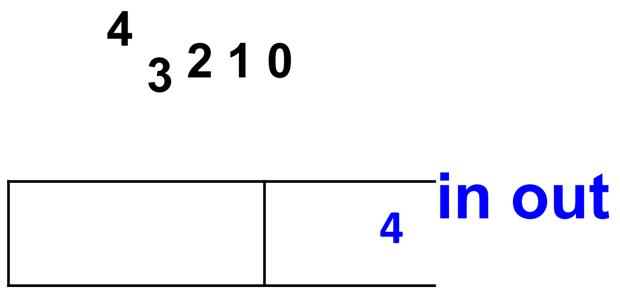
The buffer is empty when in ==out;
the buffer is full when ((in + 1) % BUFFER SIZE) == out.

```
 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.

Buffer size=5



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_3share mailbox A
  - P_1, sends; $P_2$and P_3receive
  - 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.

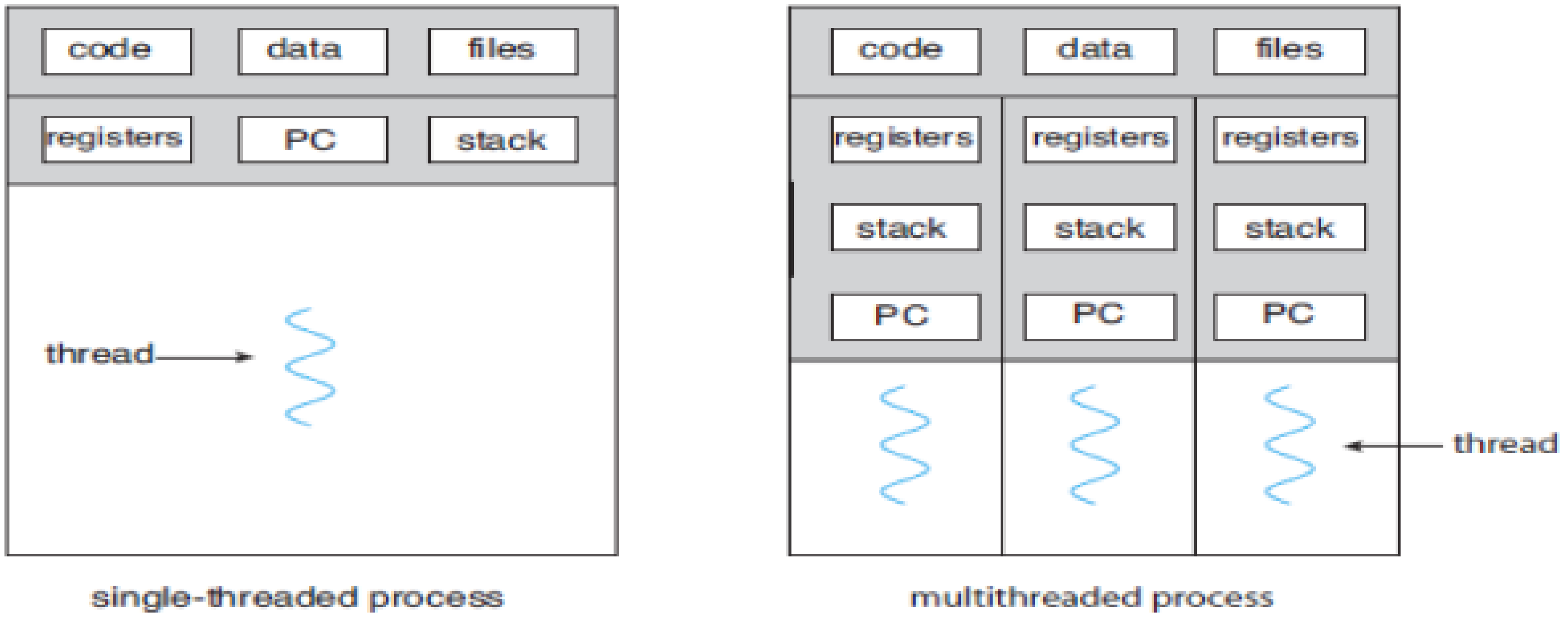
3.Unbounded capacity: The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

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.



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

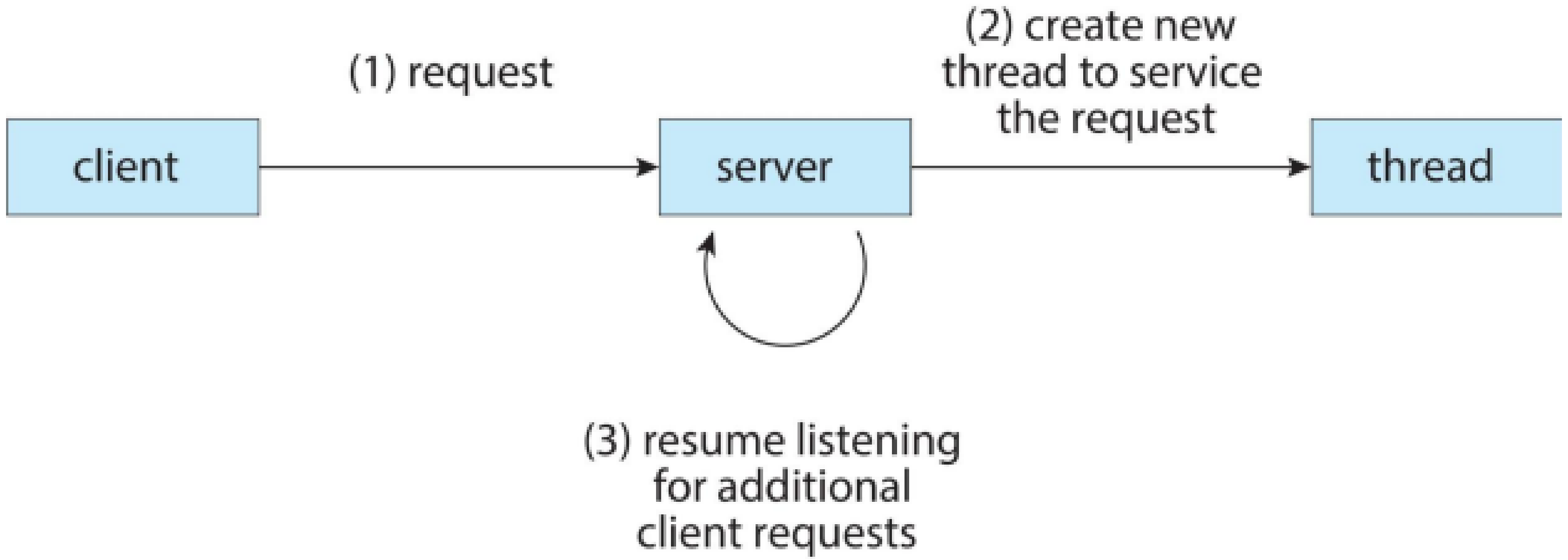
A single application may be required to perform several similar tasks.

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


(1) request
client
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

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

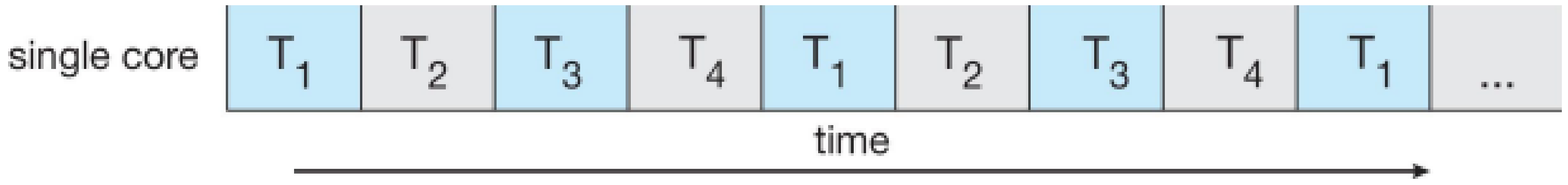
Consider an application with four threads. On a system with a single computing core, Concurrency means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency 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.**

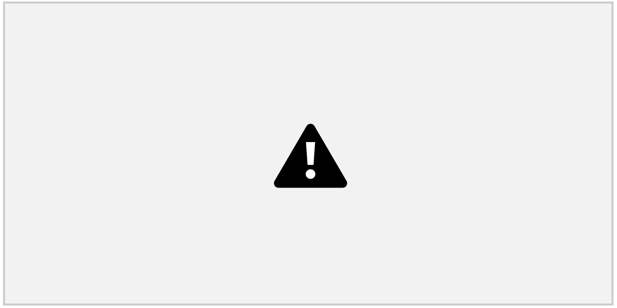
Types of parallelism - Data and Task

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:



$$\text{Speedup} \le 1/(0.25 + ((1-0.25)/2))$$
$$\le 1/(0.25+0.375)$$
$$\le 1.6$$

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

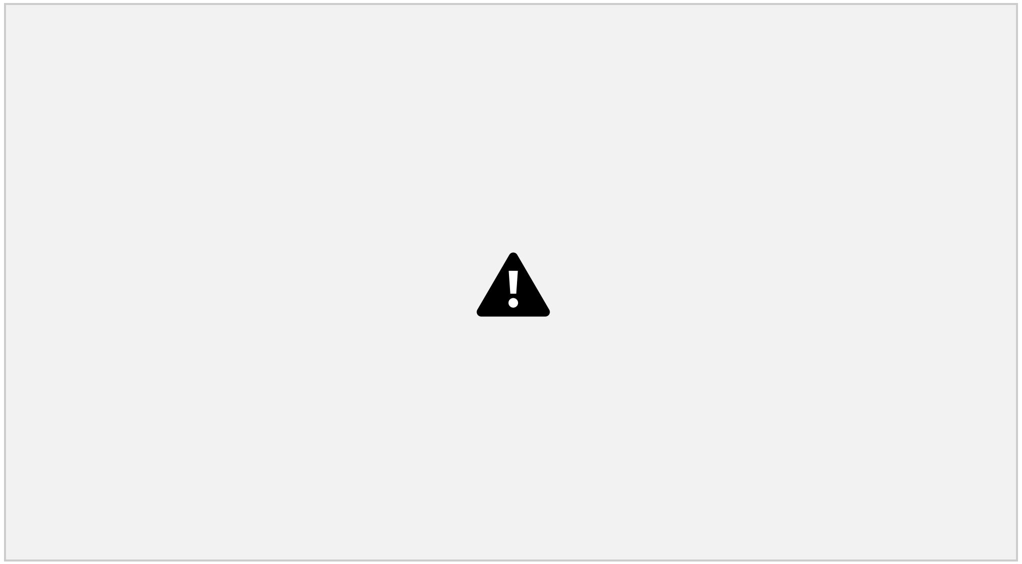
- **User threads** - management done by user-level threads library
- **Three primary thread libraries**:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

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.

ons 2 and 3, however, there is a choice.

2

3

4

1

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 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:

P P P 1 2 3

0 24 27 30

- 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:

P_1

P_2 P_3

0 3 6 30

- 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

P_3

P_2

P_4 P_1

0 3 2 4 9 1 6

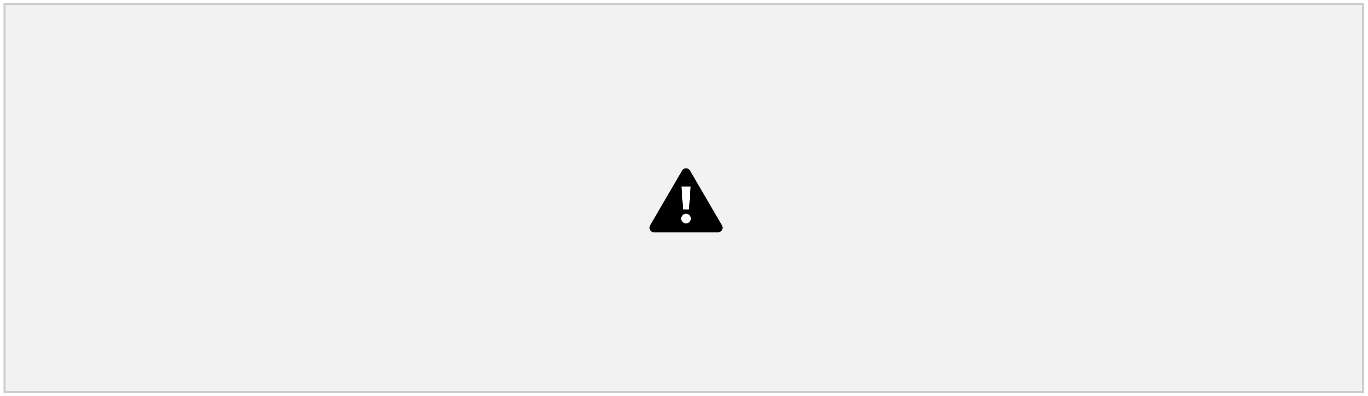
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

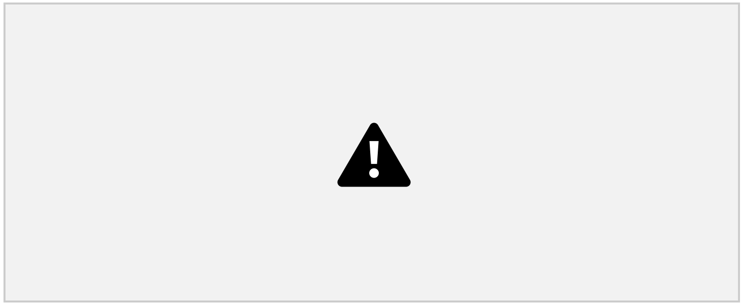
Determining Length of Next CPU Burst

- 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





• 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

P_4

P_1 P_2 P_3 P_1

0 1 2 6 5 1 7

1 0

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 $(n\text{-}1)q$ 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:

P P P $P_2P_3P_1P_1P_1$

1 1 1

0 4 7 1 0 1 4 1 8 2 2 2 6 3 0

• 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