

# **Unit - 1**

## **Process and Thread Management**

# Topics to be covered

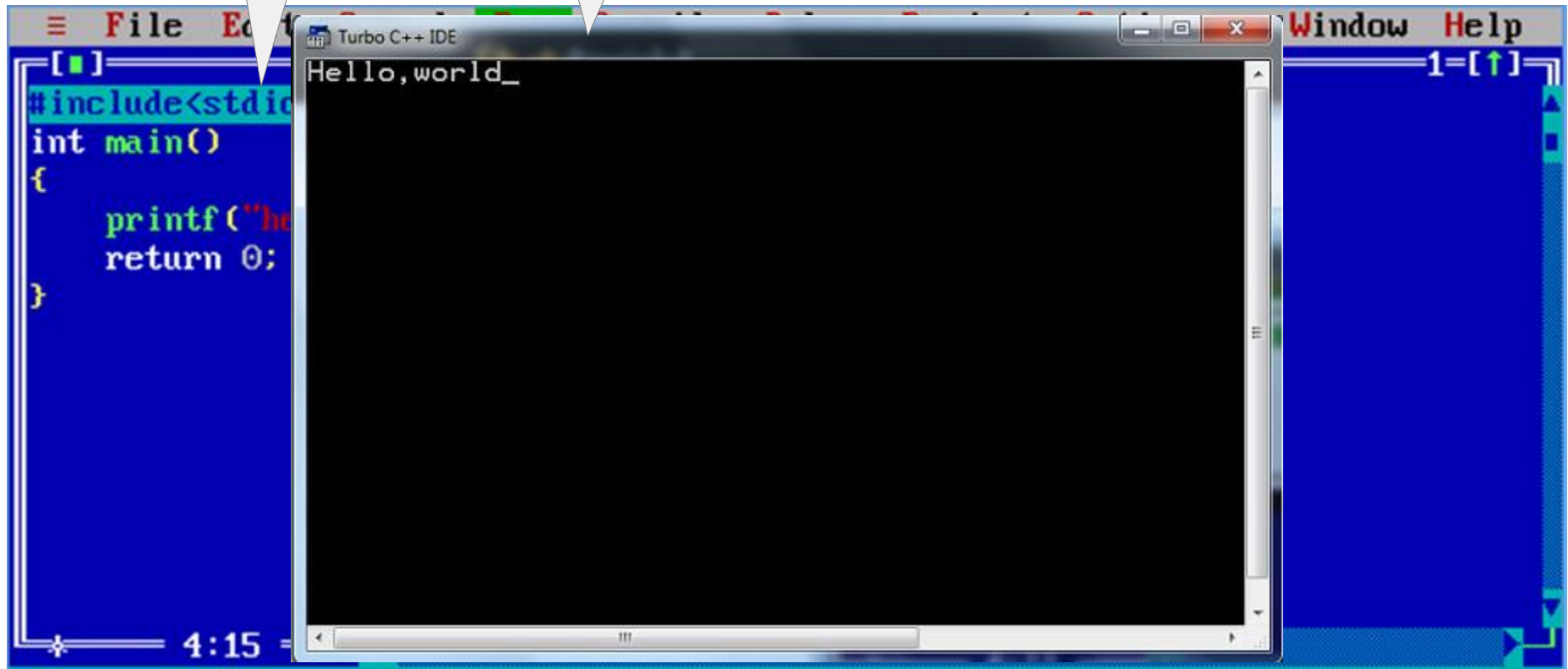
---

- Definition of process
- Process relationship
- Process states
- Process state transitions
- Process control
- Process control block
- Context switching
- Threads
- Concept of multithreads
- Benefits of threads
- Types of threads

# What is Process?

Program

Process



The image shows a screenshot of the Turbo C++ IDE. The main window displays a C program with the following code:

```
#include <stdio.h>
int main()
{
    printf("Hello, world_");
    return 0;
}
```

The output window shows the result of the program's execution: "Hello, world\_". The IDE interface includes a menu bar with "File", "Edit", "Window", and "Help", and a status bar at the bottom left showing the time "4:15".

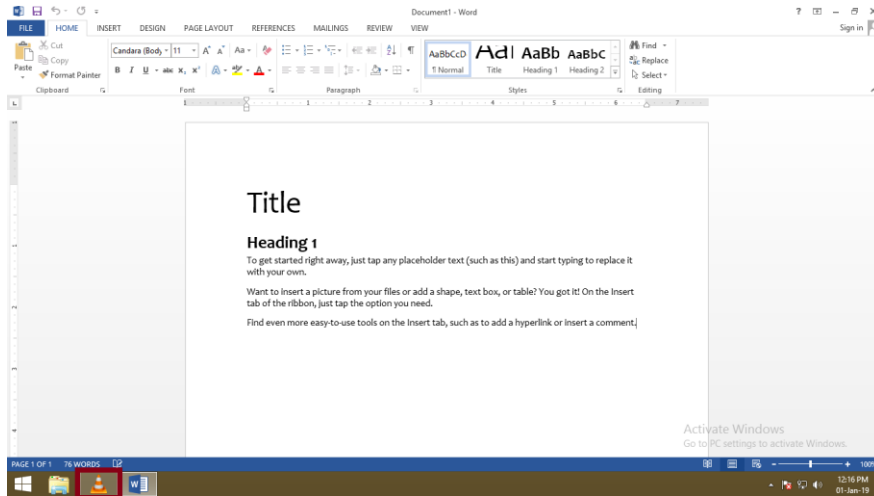
# What is Process?

---

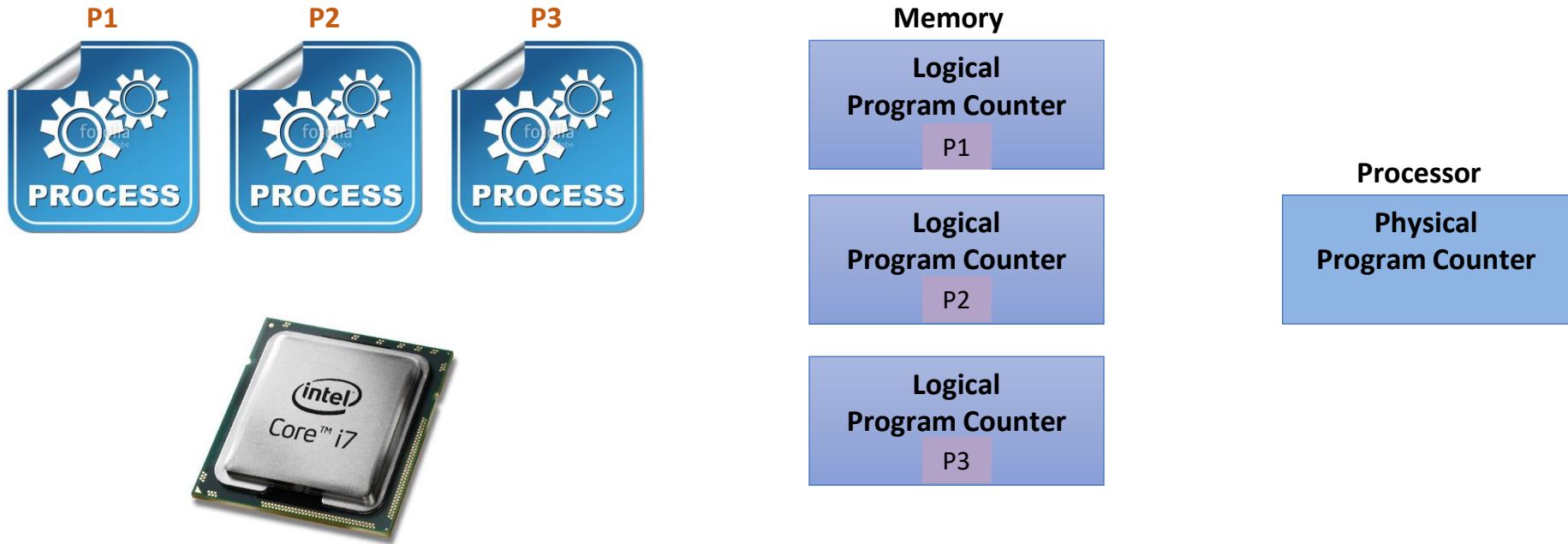
- Process is a **program under execution**.
- Process is an **abstraction of a running program**.
- Process is an **instance of an executing program**, including the current values of the program counter, registers & variables.
- Each process has its own virtual CPU.

# Multiprogramming

- The real CPU **switches back and forth** from process to process.
- This **rapid switching back and forth** is called **multiprogramming**.
- The **number of processes loaded simultaneously in memory** is called **degree of multiprogramming**.

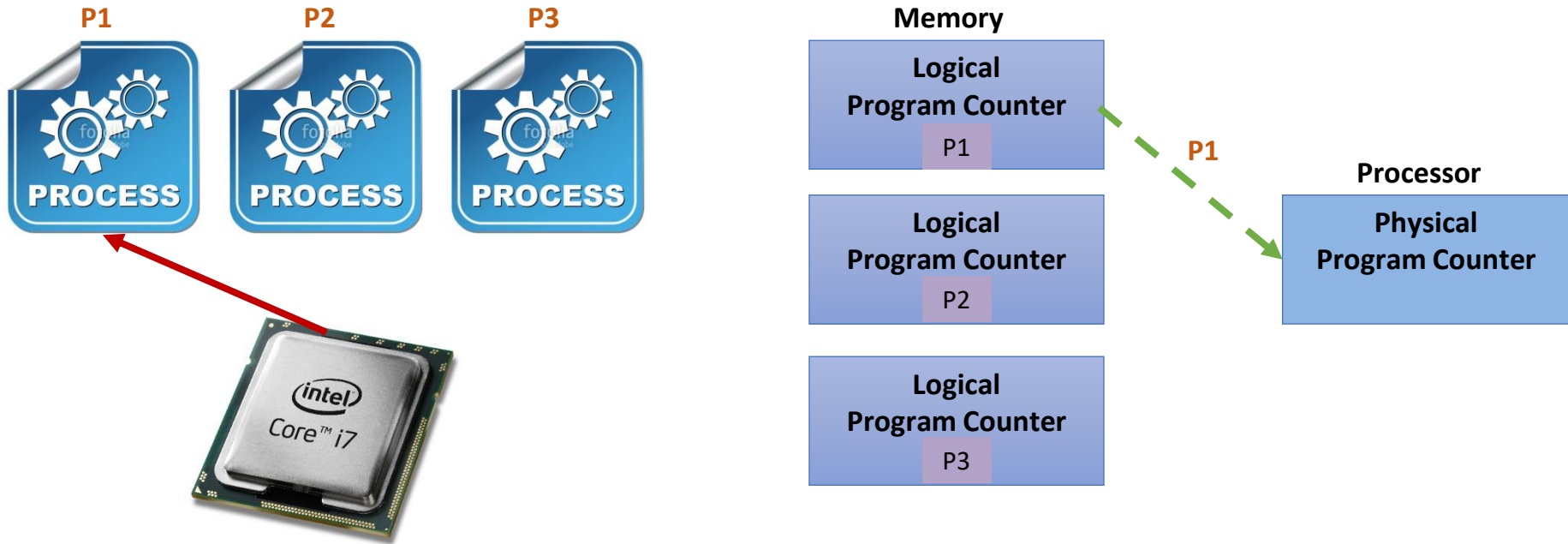


# Multiprogramming execution



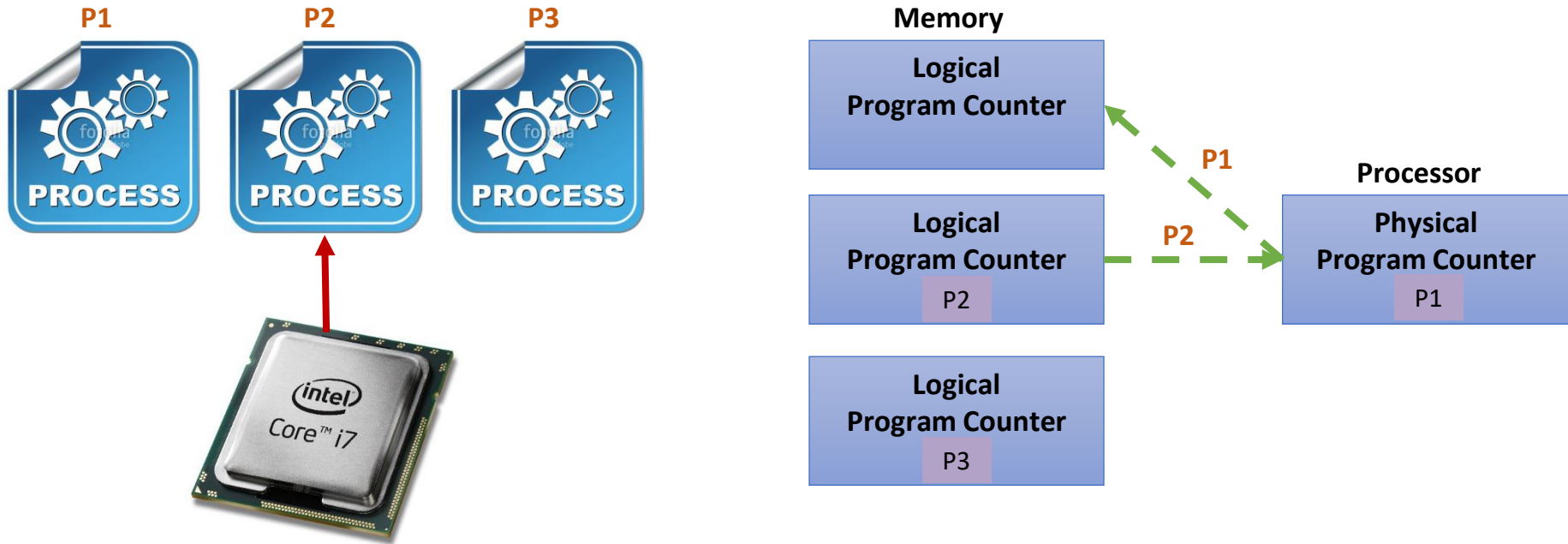
- There are **three processes**, **one processor** (CPU), **three logical program counter** (one for each processes) in memory and one physical program counter in processor.
- Here **CPU is free** (no process is running).
- No data in physical program counter.

# Multiprogramming execution



- CPU is **allocated to process P1** (process P1 is running).
- **Data of process P1 is copied** from its logical program counter to the physical program counter.

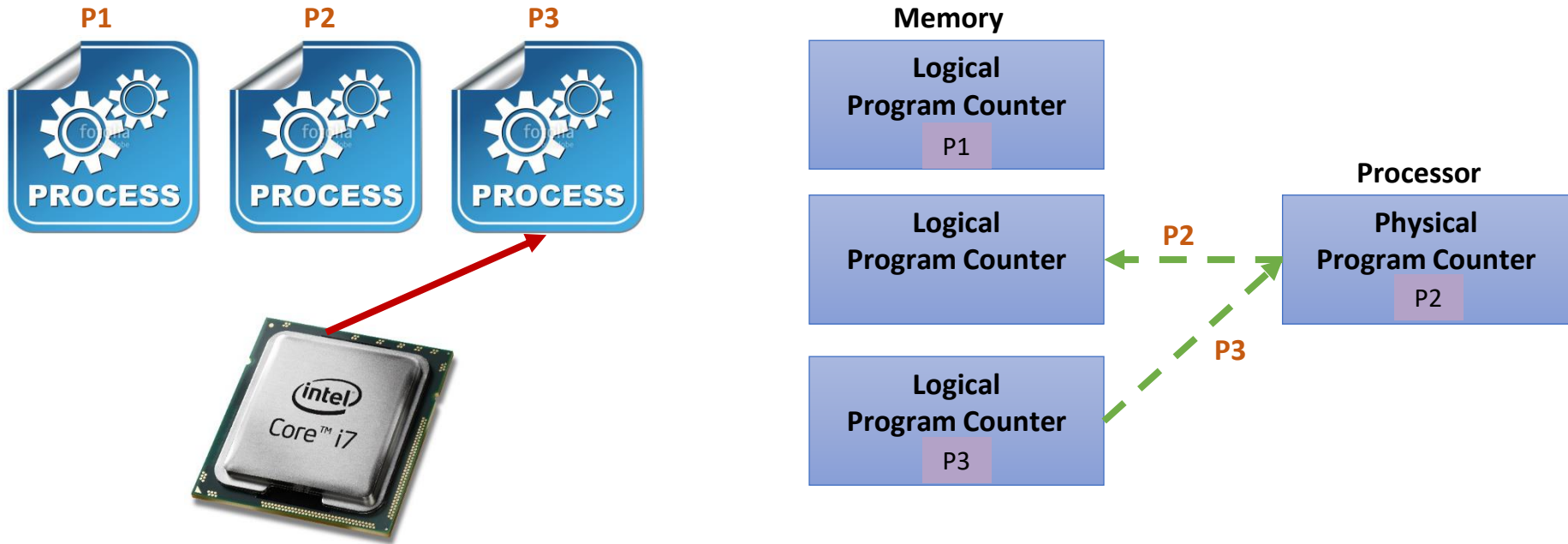
# Multiprogramming execution



- CPU **switches** from process **P1** to process **P2**.
- CPU is **allocated to process P2** (process P2 is running).
- **Data of process P1 is copied back** to its logical program counter.
- **Data of process P2 is copied** from its logical program counter to the physical program counter.



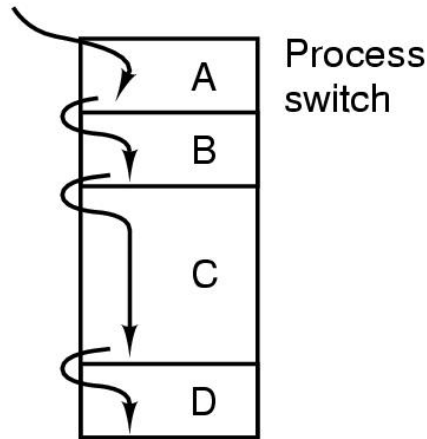
# Multiprogramming execution



- CPU **switches** from process **P2** to process **P3**.
- CPU is **allocated to process P3** (process P3 is running).
- **Data of process P2 is copied back** its logical program counter.
- **Data of process P3 is copied** from its logical program counter to the physical program counter.

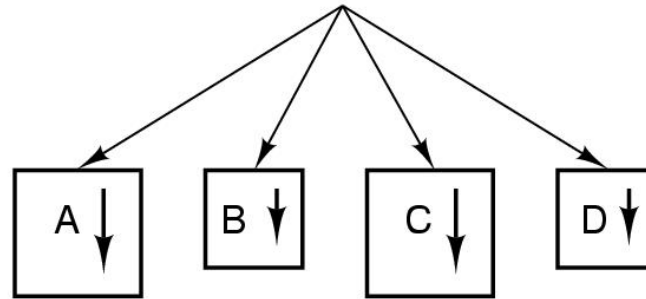
# Process Model

One program counter

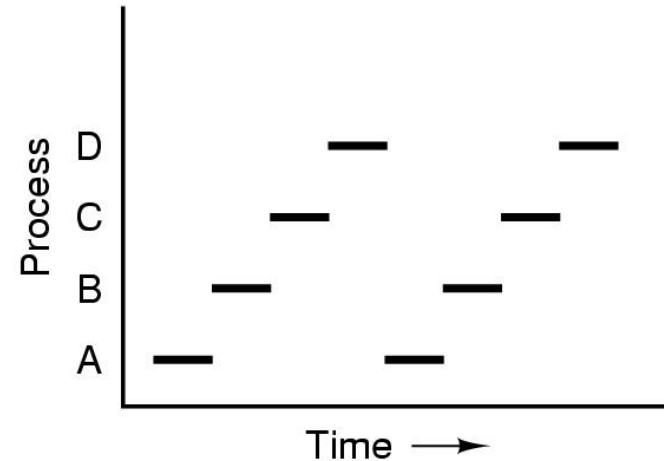


(a)

Four program counters



(b)



(c)

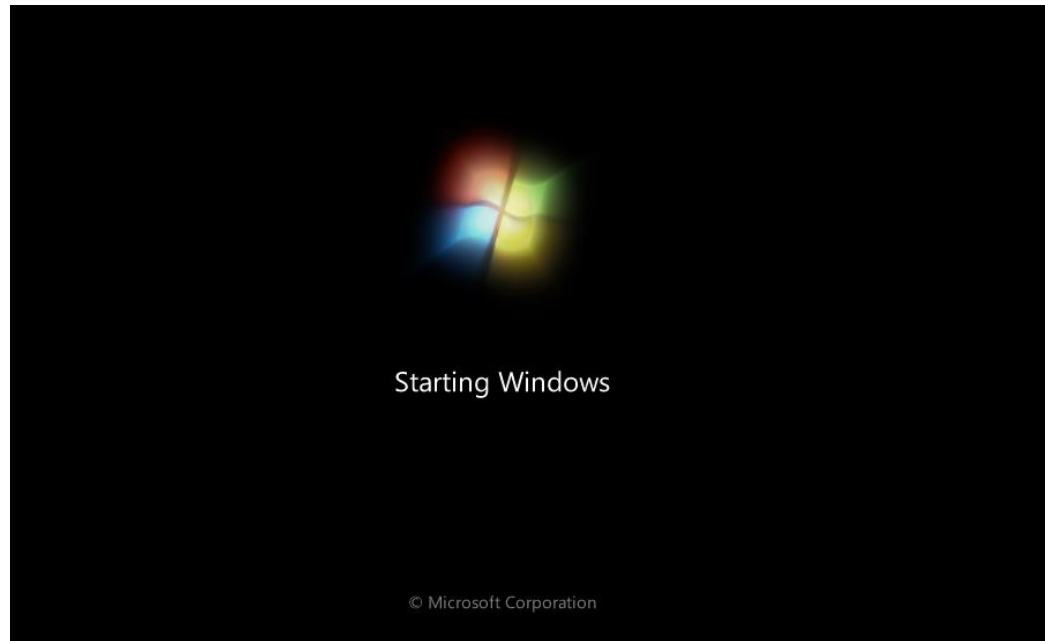
- Fig. (a) Multiprogramming of **four programs in memory**
- Fig. (b) **Conceptual model of 4 independent, sequential processes**, each with its own flow of control (i.e., its own logical program counter) and each one running independently of the other ones.
- Fig. (c) over a long period of time interval, all the processes have made progress, but **at any given instant only one process is actually running**.

# Process Creation

---

## 1. System initialization

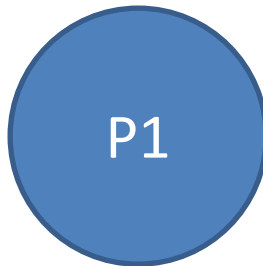
- At the time of **system (OS) booting** various processes are created
- Foreground and background processes are created
- **Background process** – that **do not interact with user** e.g. process to accept mail
- **Foreground Process** – that **interact with user**



# Process Creation

---

2. Execution of a process creation system call (**fork**) by running process
  - **Running process** will **issue system call (fork)** to create one or more new process to help it.
  - A process fetching large amount of data and execute it will create two different processes one for fetching data and another to execute it.



# Process Creation (Cont...)

3. A user request to create a new process
  - Start process by **clicking an icon** (opening word file by double click) or by **typing command**.



# Process Creation (Cont...)

---

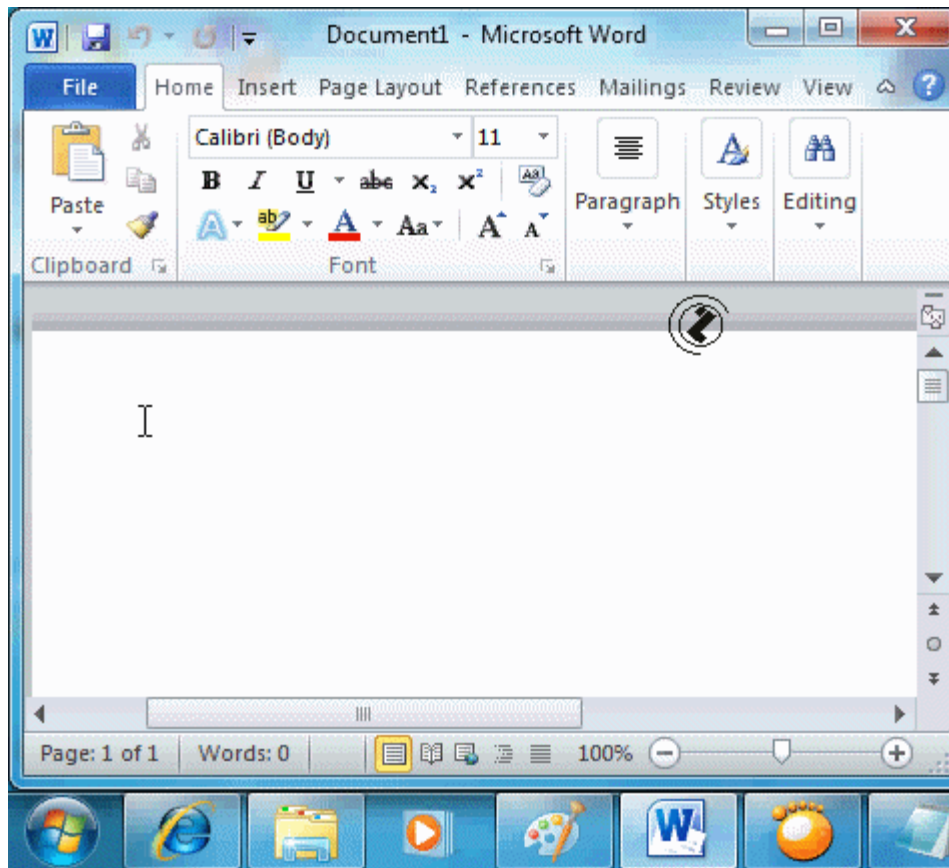
## 4. Initialization of batch process

- Applicable to only **batch system found on large mainframe**

# Process Termination

## 1. Normal exit (voluntary)

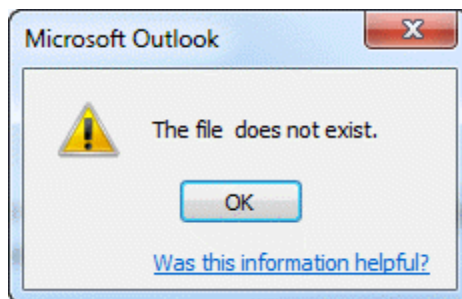
- Terminated because **process** has **done its work**.



# Process Termination

## 2. Error exit (voluntary)

- The process **discovers a fatal error** e.g. user types the command `cc foo.c` to compile the program `foo.c` and **no such file exists**, the compiler simply exit.



```
C:\Windows\system32\cmd.exe

C:\Users\gakwaya>d:

D:\>cd helloWorld

D:\helloWorld>dir
Volume in drive D has no label.
Volume Serial Number is 3493-965D

Directory of D:\helloWorld

02/13/2014  01:10 PM    <DIR>          .
02/13/2014  01:10 PM    <DIR>          ..
02/12/2014  07:42 AM                72 main.c
02/06/2014  12:50 PM                77 printHello.c
02/06/2014  12:51 PM                40 printHello.h
                3 File(s)      189 bytes
                2 Dir(s)   9,629,696,000 bytes free

D:\helloWorld> cc foo.c

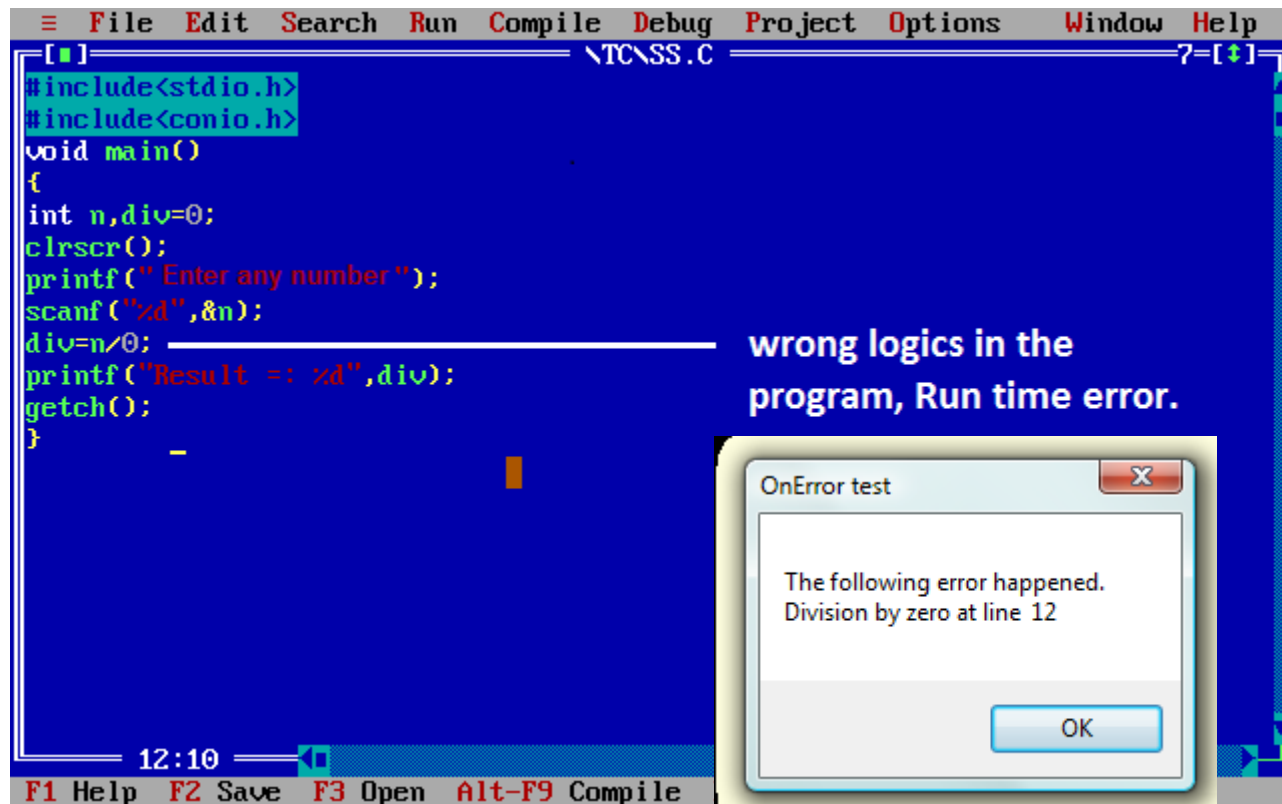
D:\helloWorld>
```



# Process Termination

## 3. Fatal error (involuntary)

- An error caused by a process often **due to a program bug** e.g. **executing an illegal instruction, referencing nonexistent memory or divided by zero.**



The screenshot shows a Turbo C++ IDE window titled "NTCSS.C". The code in the editor is as follows:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n,div=0;
clrscr();
printf(" Enter any number");
scanf("%d",&n);
div=n/0;
printf("Result =: %d",div);
getch();
}
```

A white line points from the text "wrong logics in the program, Run time error." to the line `div=n/0;` in the code. An error dialog box titled "OnError test" is open in the foreground, displaying the message: "The following error happened. Division by zero at line 12". The dialog box has an "OK" button at the bottom.

# Process Termination

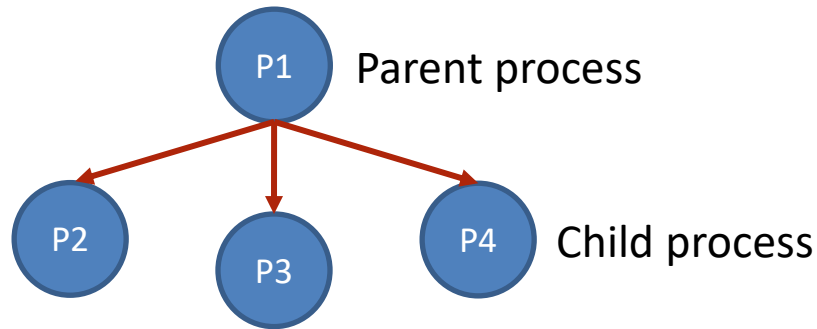
---

4. Killed by another process (involuntary)
  - A process **executes a system call** telling the OS **to kill some other process** using **kill system call**.

# Process Hierarchies

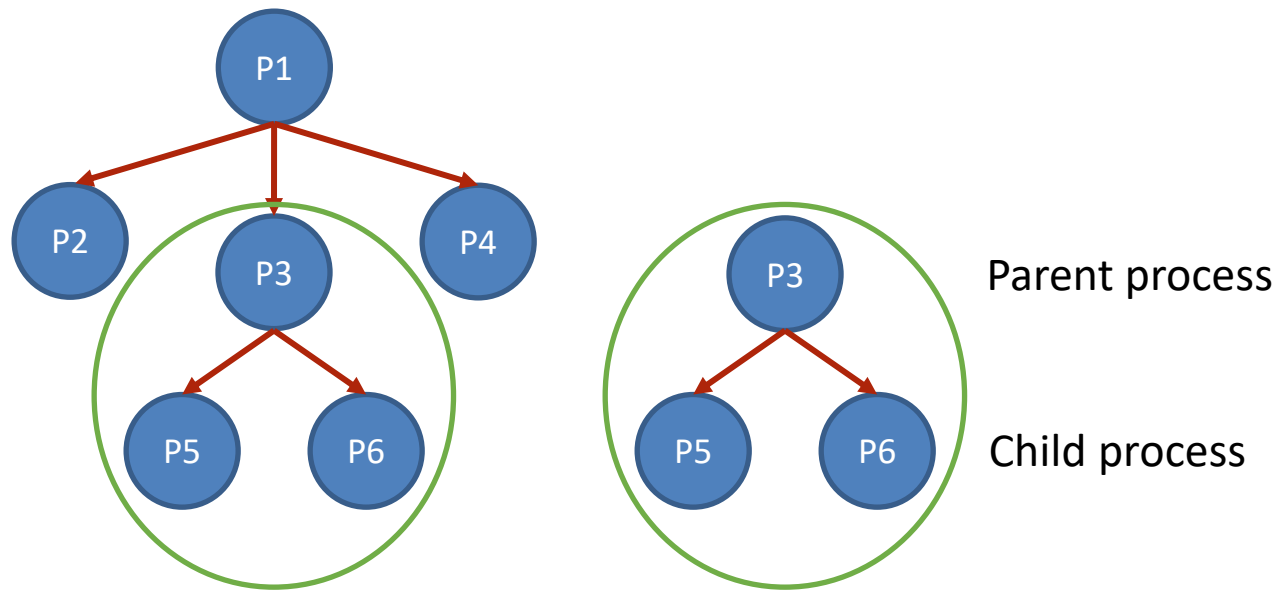
---

- Parent process can create child process, child process can create its own child process.



# Process Hierarchies

- Parent process can create child process, child process can create its own child process.

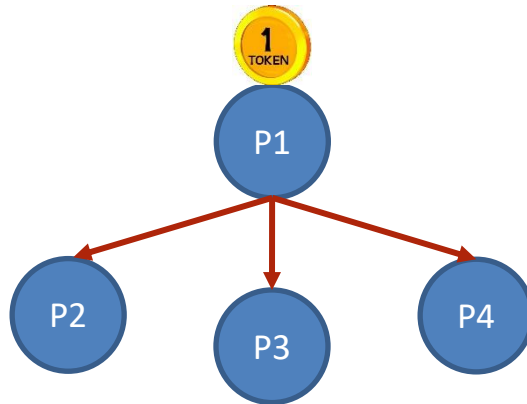


- **UNIX has hierarchy concept** which is known as process group
- **Windows has no concept of hierarchy**
  - All the process as treated equal (**use handle** concept)

# Handle

---

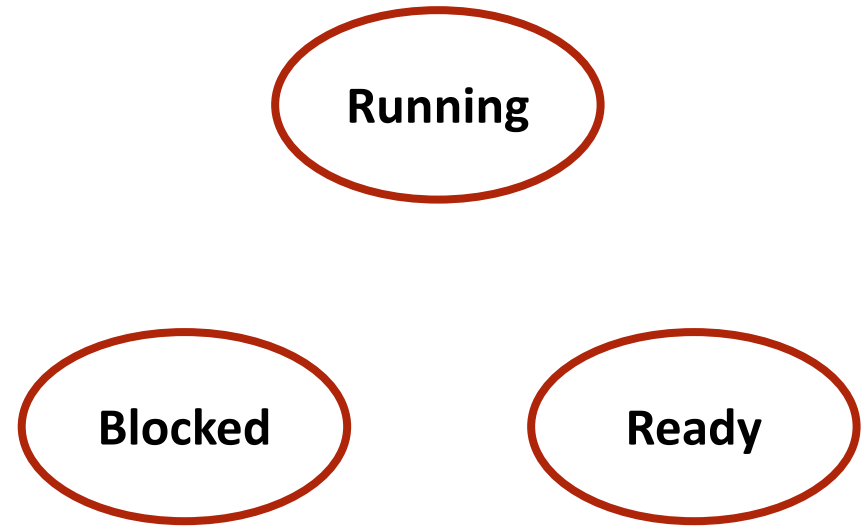
- When a **process is created**, the **parent process is given a special token called handle**.
- This handle is **used to control the child process**.
- A process is **free to pass this token** to some other process.



# Process State

---

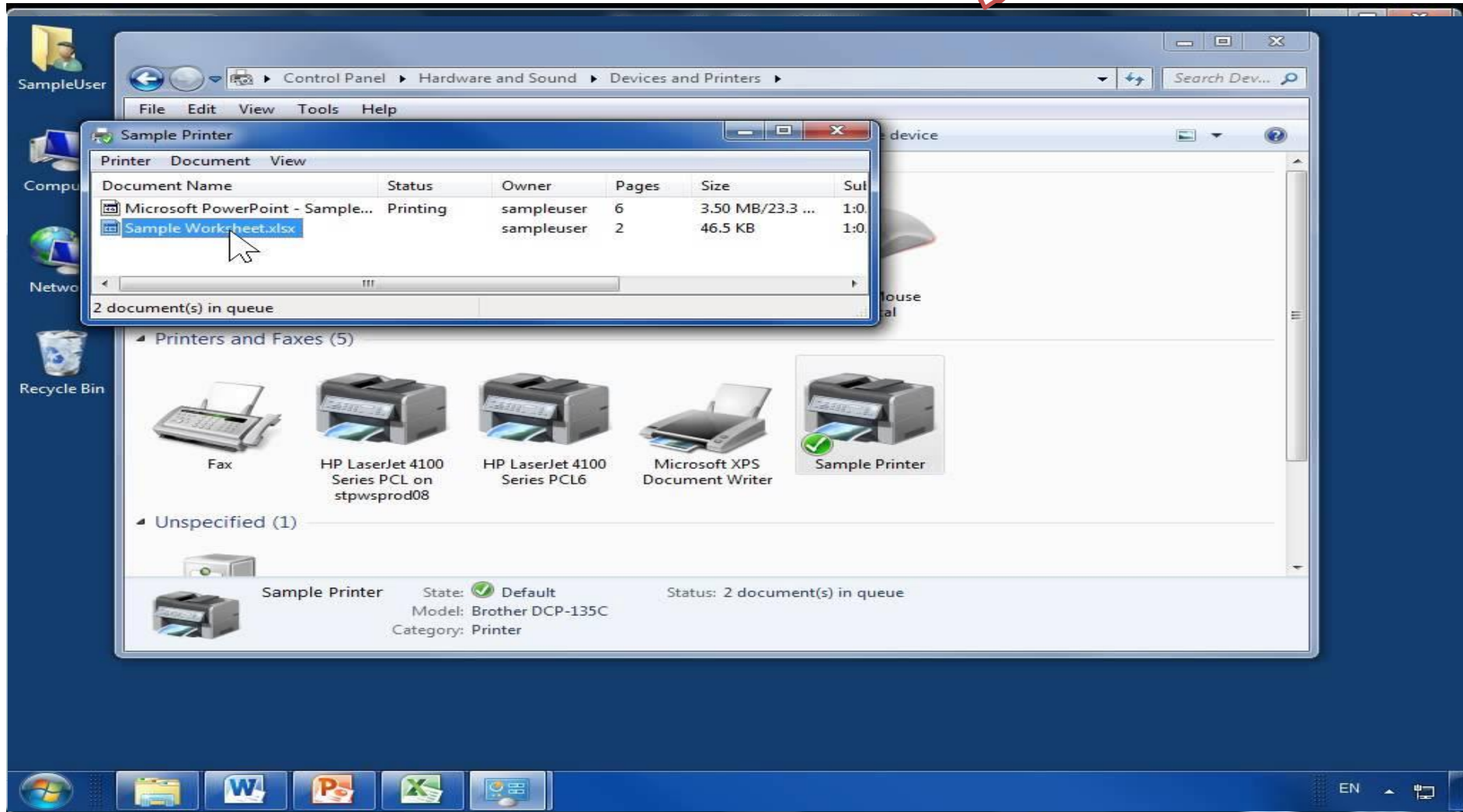
1. **Running** – Process is actually **using the CPU**
2. **Ready** – Process is runnable, **temporarily stopped to let another process to run**
3. **Blocked** – process is **unable to run until some external event happens**



Processes are always **either executing (running) or waiting to execute (ready) or waiting for an event (blocked)** to occur.

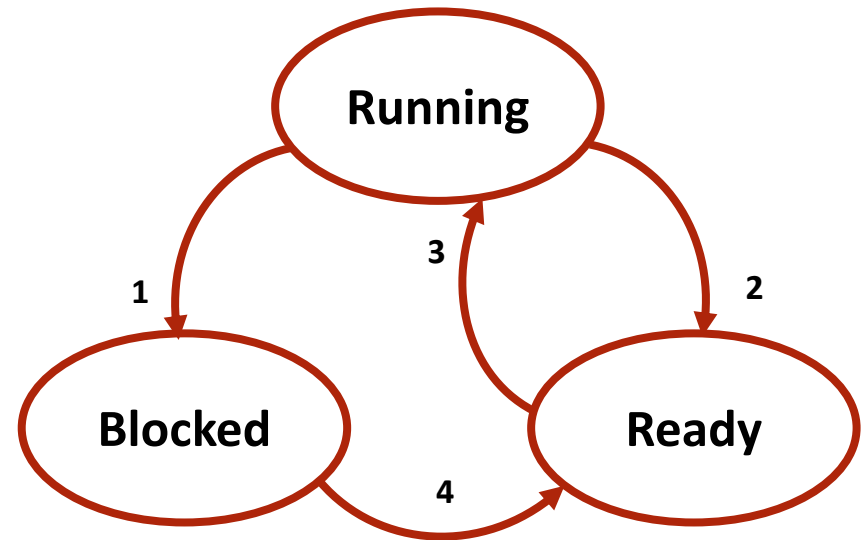
# Process State

Blocked



# Process State Transitions

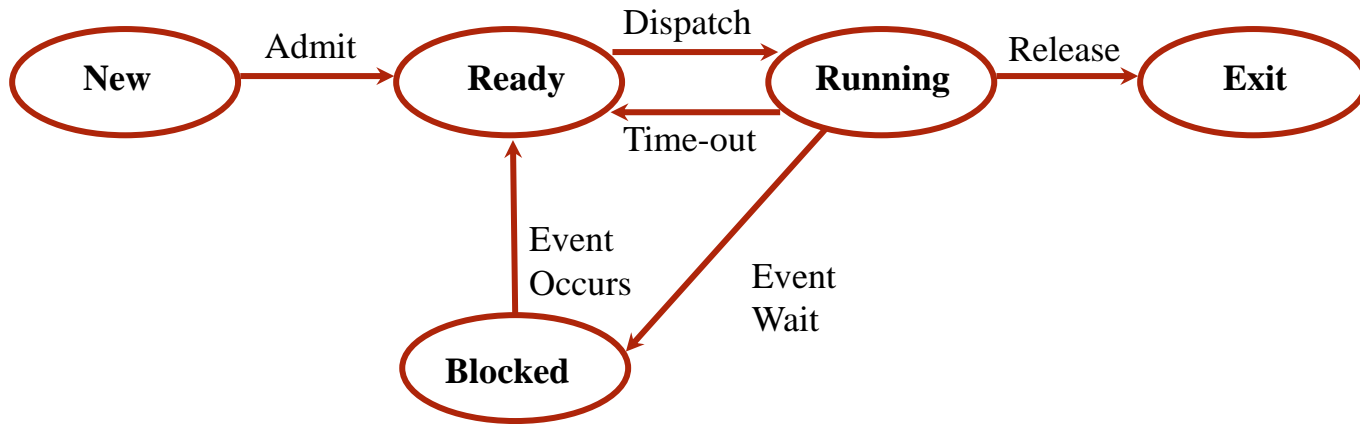
- When and how these transitions occur (process moves from one state to another)?
  1. Process blocks for input or waits for an event (i.e. printer is not available)
  2. Scheduler picks another process
    - End of time-slice or pre-emption.
  3. Scheduler picks this process
  4. Input becomes available, event arrives (i.e. printer become available)





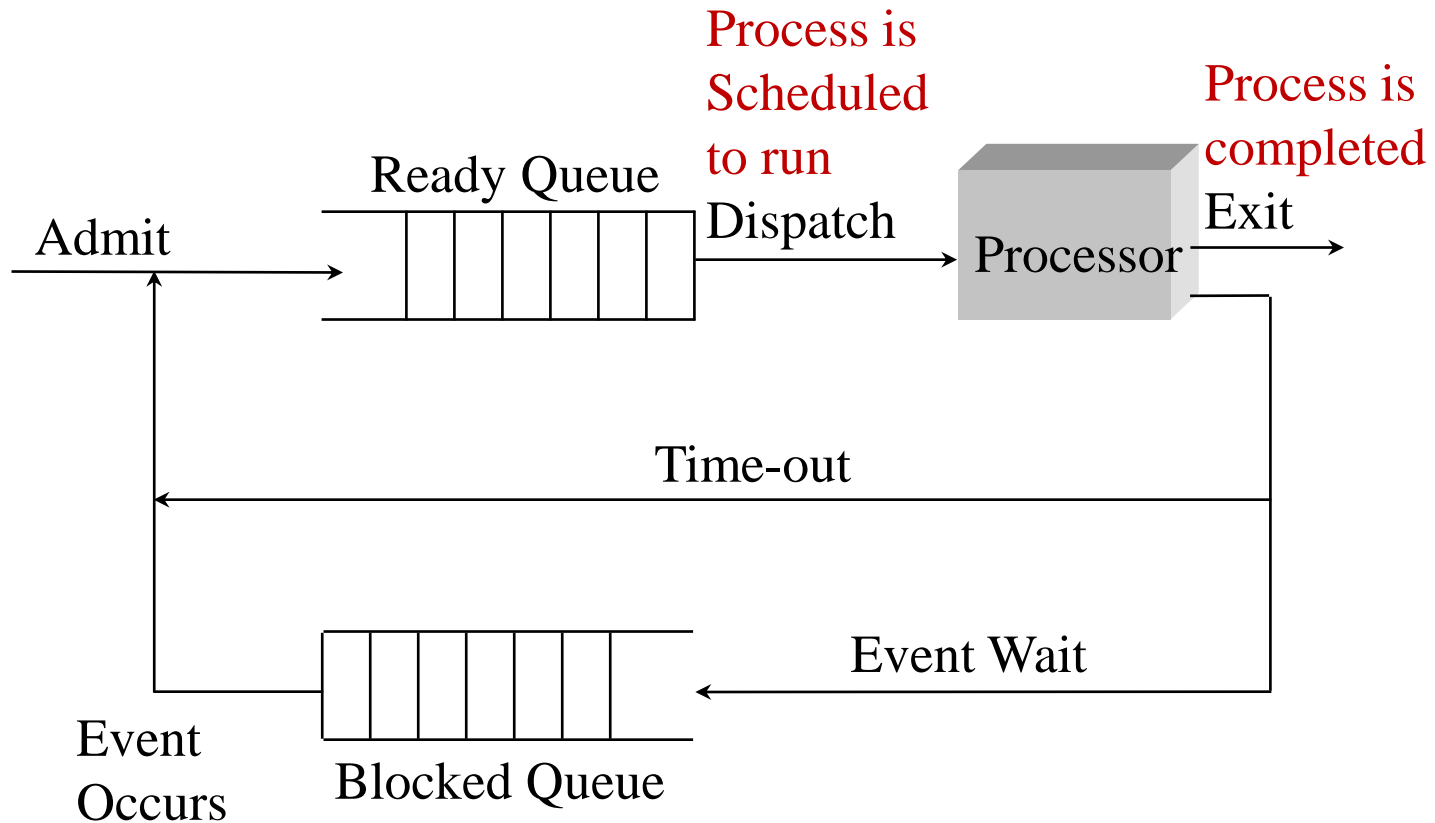
# Five State Process Model and Transitions

---



- **New** – process is being **created**
- **Ready** – process is **waiting to run** (runnable), **temporarily stopped** to let another process run
- **Running** – process is **actually using the CPU**
- **Blocked** – **unable to run** until some external event happens
- **Exit (Terminated)** – process has **finished the execution**

# Queue Diagram



# Process Control Block (PCB)

---

- A Process Control Block (PCB) is a **data structure maintained by the operating system for every process.**
- PCB is **used for storing the collection of information** about the processes.
- The PCB is **identified by** an **integer process ID (PID)**.
- A PCB keeps all the information needed to keep track of a process.

# Process Control Block (PCB)

---

- The PCB is maintained for a process **throughout its lifetime** and is **deleted once** the **process terminates**.
- The **architecture** of a PCB is completely **dependent on operating system** and may contain different information in different operating systems.
- PCB **lies in kernel** memory space.

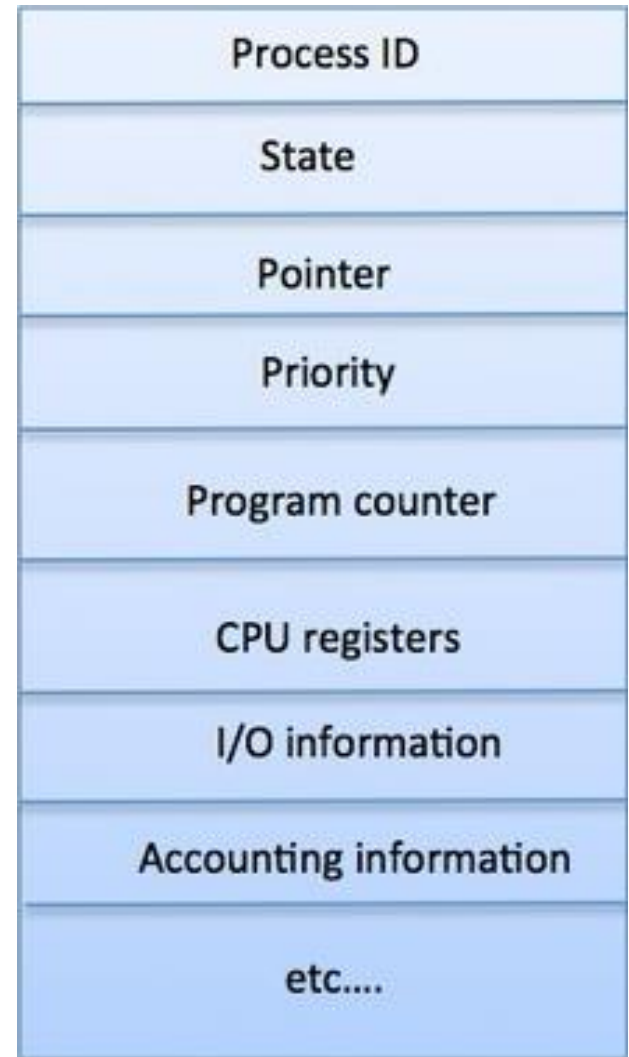
# Process Control Block (PCB) contains

- **Process ID** - **Unique identification** for each of the process in the operating system.
- **Process State** - The **current state** of the process i.e., whether it is ready, running, waiting.
- **Pointer** - A **pointer to parent process**.
- **Priority** - **Priority** of a process.
- **Program Counter** - Program Counter is a **pointer** to the **address** of the **next instruction** to be **executed** for this process.



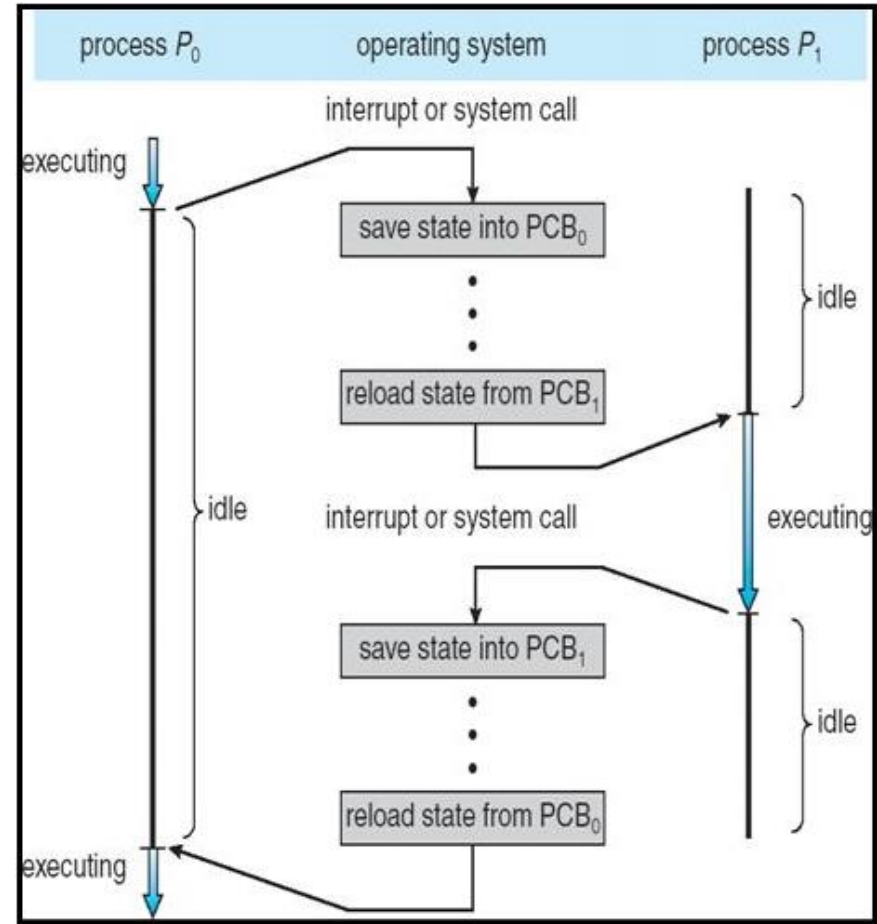
# Process Control Block (PCB) contains

- **CPU registers** - Various CPU **registers** where process **need to** be **stored** for **execution** for **running state**.
- **IO status information** - This includes a list of **I/O devices allocated** to the process.
- **Accounting information** - This includes the **amount of CPU used** for process execution, time limits etc.



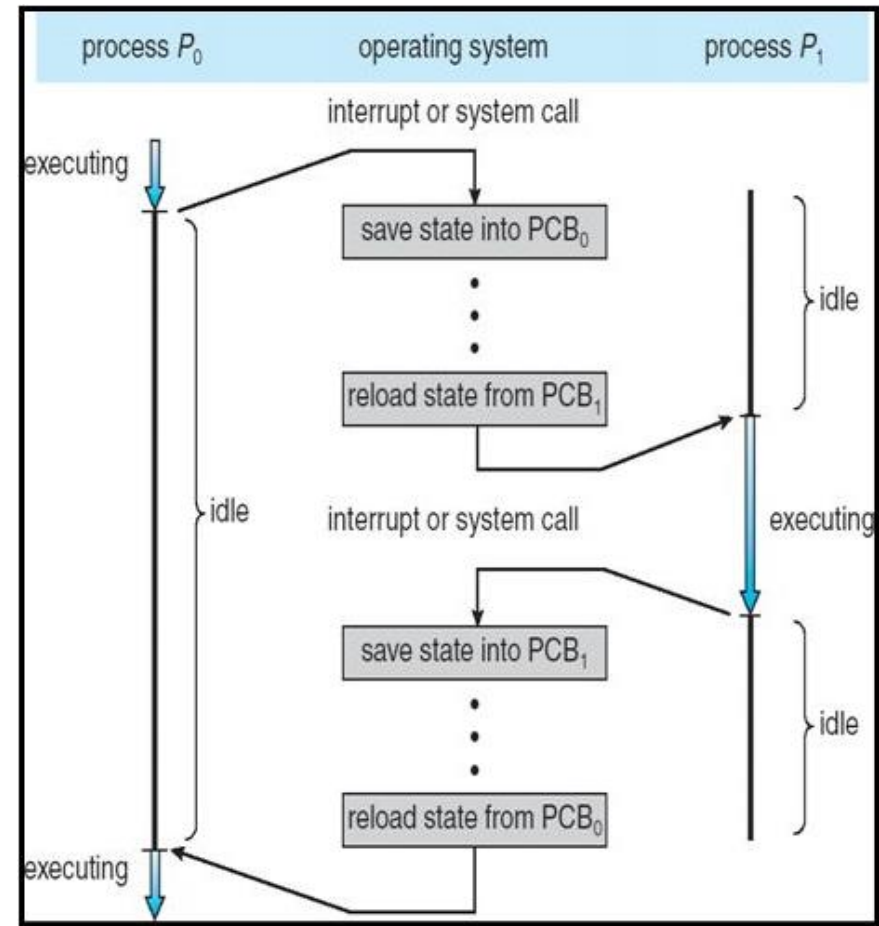
# Context switching

- Context switch means **stopping one process** and **restarting another process**.
- When an event occur, the OS **saves the state of an active process** and **restore the state of new process**.
- Context switching is **purely overhead** because system does not perform any useful work while context switch.



# Steps performed by OS during Context switching

- Sequence of action:
  1. OS **takes control** (through interrupt)
  2. **Saves context** of **running process** in the process **PCB**
  3. **Reload context** of **new process** from the **new process PCB**
  4. **Return control** to **new process**

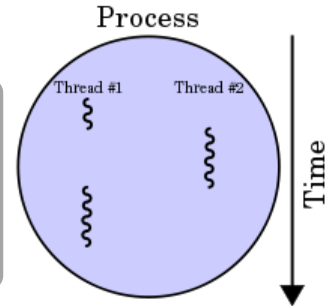




# Thread

- Thread is **light weight process** created by a process.

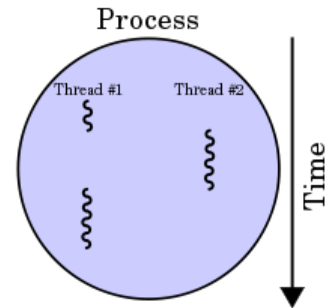
**Processes** are used to **execute large**, 'heavyweight' jobs such as working in word, while **threads** are used to **carry out smaller** or 'lightweight' jobs such as auto saving a word document.



A screenshot of the Microsoft Word application window. The title bar reads 'I am studying in Darshan Engineering College.docx - Microsoft Word'. The ribbon shows the 'Home' tab with various font and paragraph options. The main text area contains two lines of text: 'I am studying in Darshan Engineering College.' and 'This college is located on Morbi Raod.'. The word 'College.' in the first line and 'Raod.' in the second line are circled in red. A blue oval labeled 'Threads' is positioned below the text, with two red arrows pointing from it to the circled words. At the bottom of the window, the status bar shows 'Words is saving C:\Users\GNWEBSOFT\Desktop\2140702 - OS Unit - 2.pptx.' which is also circled in red. The Windows taskbar is visible at the bottom of the screen.

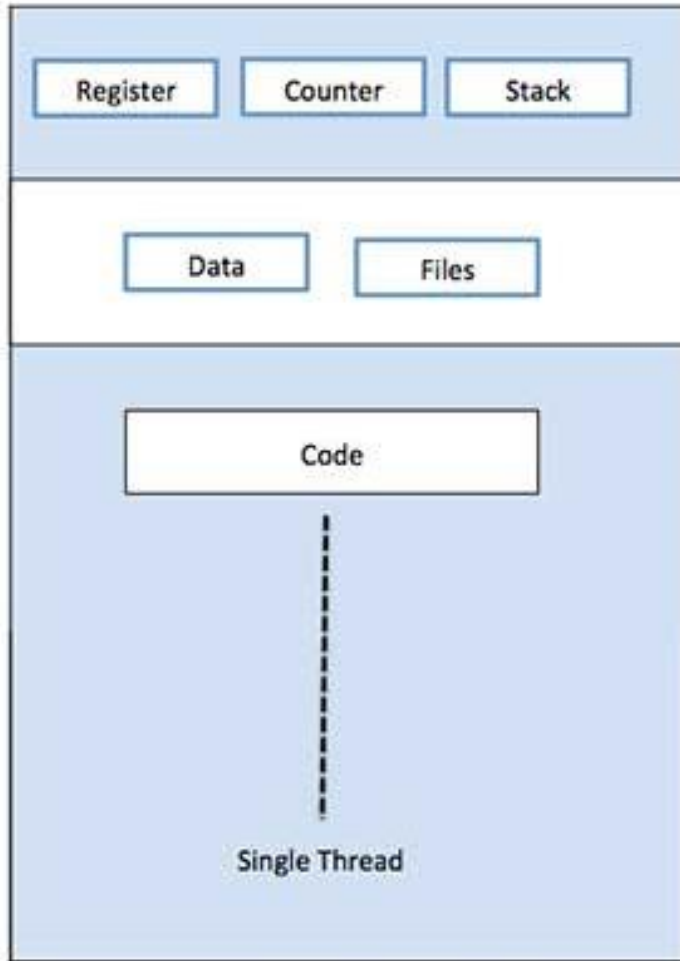
# Thread

- Thread is **light weight process** created by a process.
- Thread is a **single sequence stream** within a process.
- Thread has its own

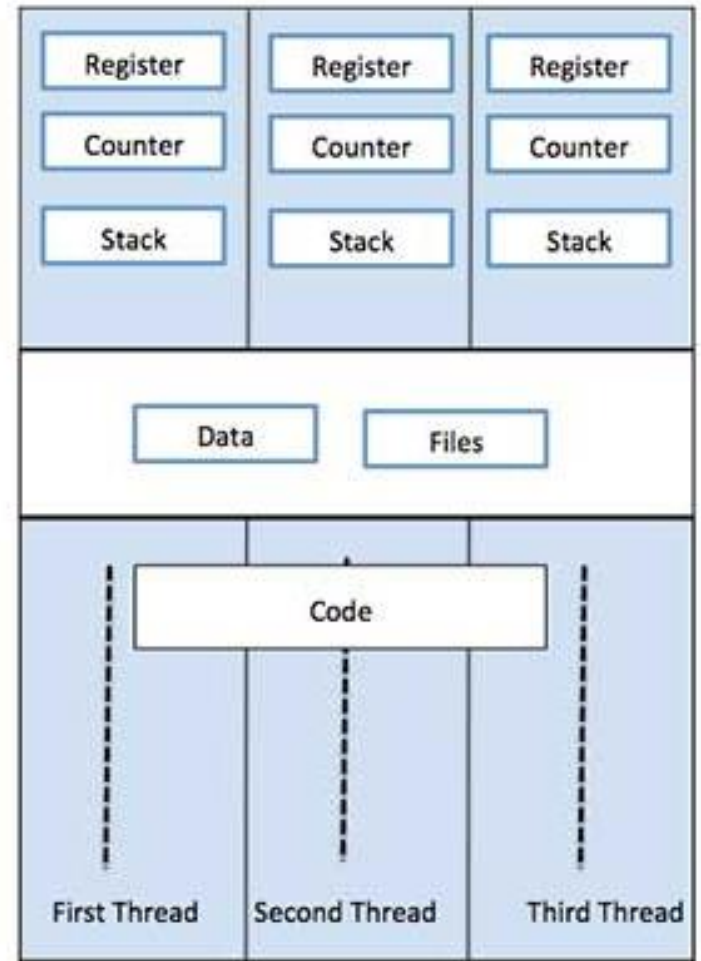


1. **program counter** that **keeps track** of **which instruction to execute next**.
2. **system registers** which **hold its current working variables**.
3. **stack** which **contains the execution history**.

# Single Thread VS Multiple Thread



Single Process P with single thread



Single Process P with three threads

# Similarities between Process & Thread

---

- Like processes threads **share CPU** and **only one thread is running at a time**.
- Like processes threads **within a process execute sequentially**.
- Like processes thread **can create childrens**.
- Like a traditional process, a thread **can be in any one of several states**: running, blocked, ready or terminated.
- Like process threads **have Program Counter, Stack, Registers and State**.

# Dissimilarities between Process & Thread

---

- Unlike processes threads are **not independent** of one another.
- Threads within the same process **share an address space**.
- Unlike processes all threads can **access every address** in the task.
- Unlike processes threads are **design to assist one other**. Note that processes might or might not assist one another because processes may be originated from different users.

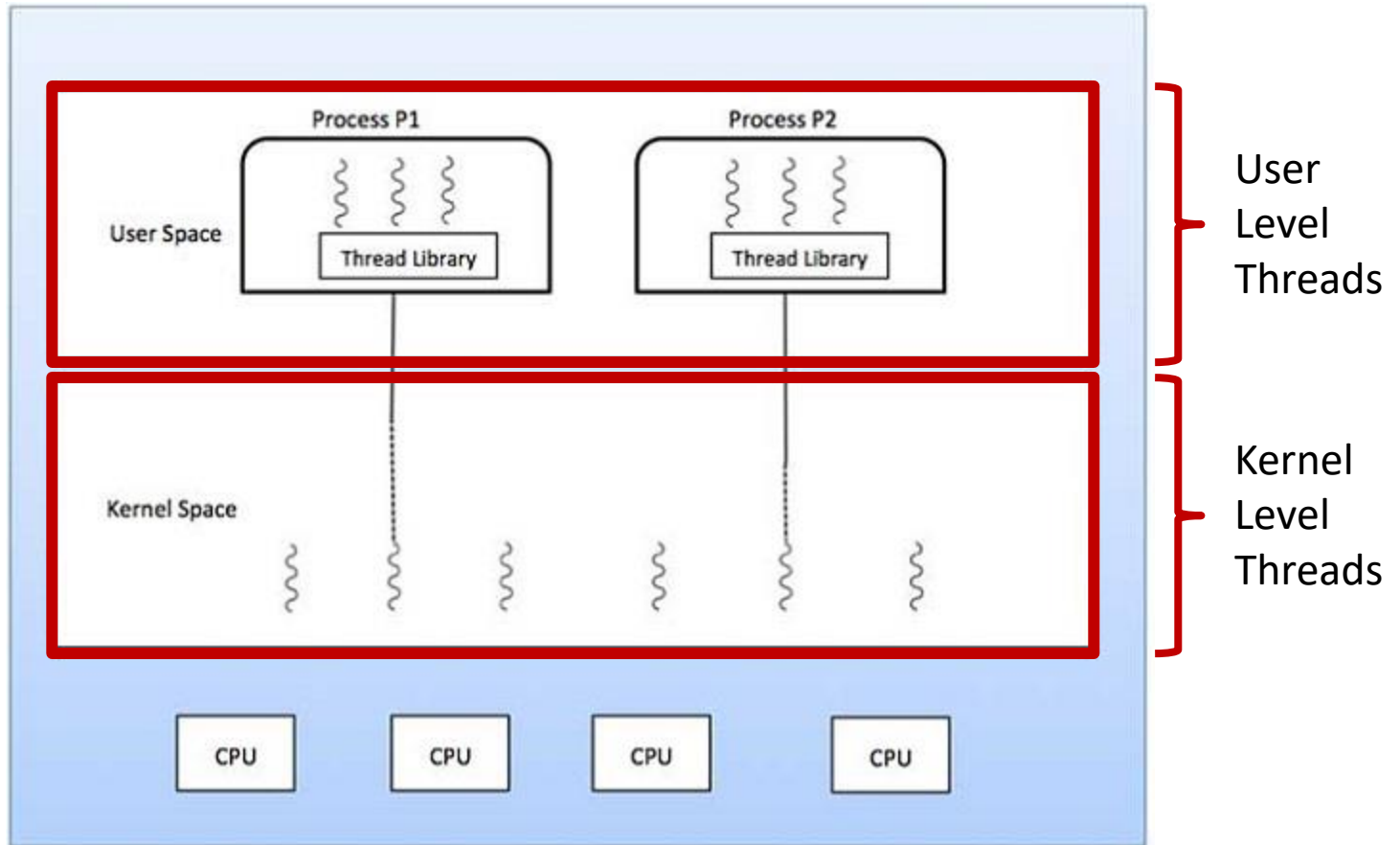
# Advantages of Threads

---

- Threads **minimize** the **context switching time**.
- Use of threads **provides concurrency** within a process.
- **Efficient communication**.
- It is more **easy to create** and **context switch** threads.
- Threads can **execute** in **parallel** on multiprocessors.
- With threads, an application can **avoid per-process overheads**
  - Thread creation, deletion, switching easier than processes.
- Threads have **full access** to **address space** (easy sharing).

# Types of Threads

1. Kernel Level Thread
2. User Level Thread



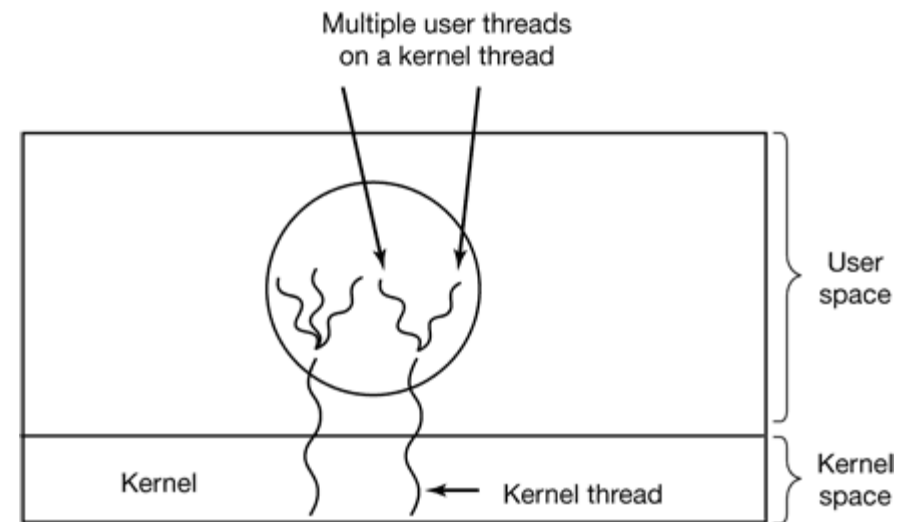
# Types of Threads (Cont...)

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are <b>implemented by users</b> .	Kernel threads are <b>implemented by OS</b> .
<b>OS doesn't recognize</b> user level threads.	Kernel threads are <b>recognized by OS</b> .
<b>Implementation</b> of User threads is <b>easy</b> .	<b>Implementation</b> of Kernel thread is <b>complex</b> .
<b>Context switch</b> time is <b>less</b> .	<b>Context switch</b> time is <b>more</b> .
Context switch <b>requires no hardware support</b> .	Context switch <b>requires hardware support</b> .
If one user level thread perform blocking operation then <b>entire process</b> will be <b>blocked</b> .	If one kernel thread perform blocking operation then <b>another thread</b> with in same process <b>can continue execution</b> .
Example : Java thread, POSIX threads.	Example : Window Solaris



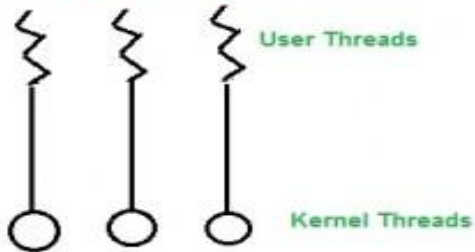
# Hybrid Thread

- Combines the advantages of user level and kernel level thread.
- It **uses kernel level thread** and then **multiplex user level thread on to** some or all of **kernel threads**.
- **Gives flexibility** to programmer that how many kernel level threads to use and how many user level thread to multiplex on each one.
- Kernel is aware of only kernel level threads and schedule it.



# Multi threading models

One to One Model

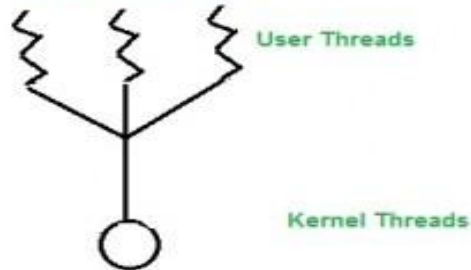


## One to One Model

Each user threads mapped to one kernel thread.

Problem with this model is that creating a user thread requires the corresponding kernel thread.

Many to One Model

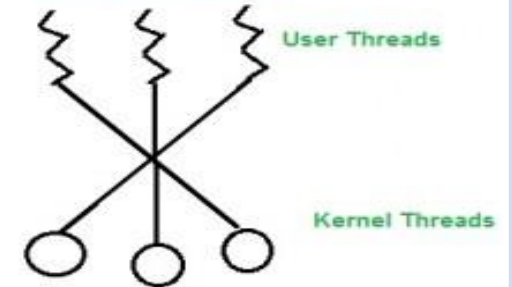


## Many to One Model

Multiple user threads mapped to one kernel thread.

Problem with this model is that a user thread can block entire process because we have only one kernel thread.

Many to Many Model



## Many to Many Model

Multiple user threads multiplex to more than one kernel threads.

Advantage with this model is that a user thread can not block entire process because we have multiple kernel thread.

# Pthread function calls

---

1. `Pthread_create`:- Create a new thread
2. `Pthread_exit`:- Terminate the calling thread
3. `Pthread_join`:- Wait for a specific thread to exit
4. `Pthread_yield`:- Release the CPU to let another thread run
5. `Pthread_attr_init`:- Create and initialize a thread's attribute structure
6. `Pthread_destroy`:- Remove a thread's attribute structure

# System calls

---

- A system call is the **programmatic way in which a computer program requests a service from the kernel** of the operating system it is executed on.
- A system call is a **way for programs to interact with the operating system**.
- A computer **program makes a system call when it makes a request to the operating system's kernel**.
- System call **provides the services of the operating system to the user programs** via Application Program Interface(API).
- It provides an **interface between a process and operating system** to allow user-level processes to request services of the operating system.
- System calls are the **only entry points into the kernel system**.
- All **programs needing resources must use system calls**.

# System calls

---

1. ps (process status):- The ps (process status) command is used to **provide information about the currently running processes**, including their process identification numbers (PIDs).
2. fork:- Fork system call is used for **creating a new process, which is called child process**, which runs concurrently with the process that makes the fork() call (parent process).
3. wait:- Wait system call **blocks the calling process until one of its child processes exits** or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.
4. exit:- Exit system call **terminates the running process normally**.
5. exec family:- The exec family of functions **replaces the current running process with a new process**.