

# Operating Systems - Processes

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

**Acknowledgements:** Material based on the textbook Operating Systems Concepts (Chapter 3)

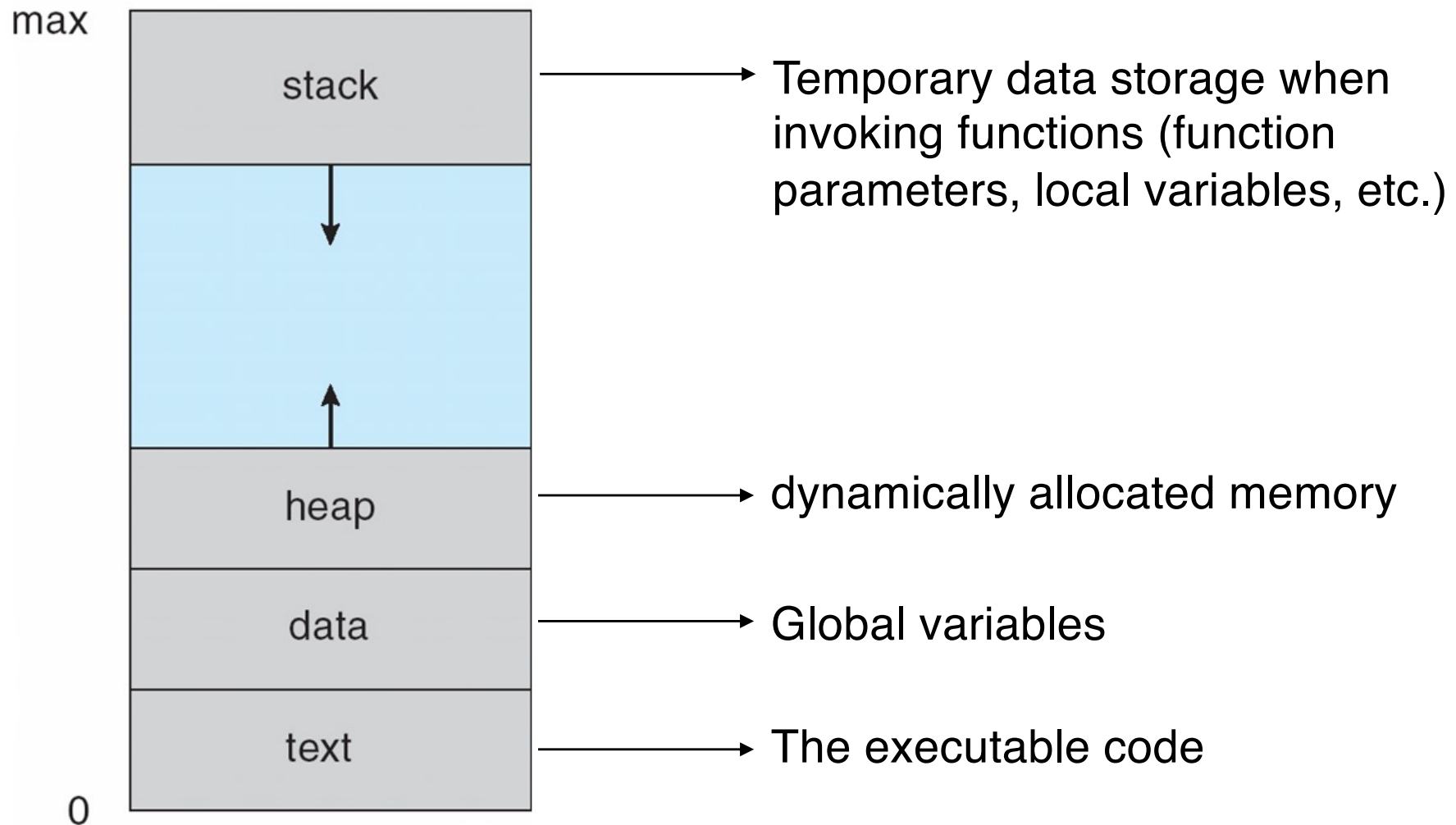
# Process

- Process is a program in execution
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory

# Questions

**Can you have one program and many processes?**

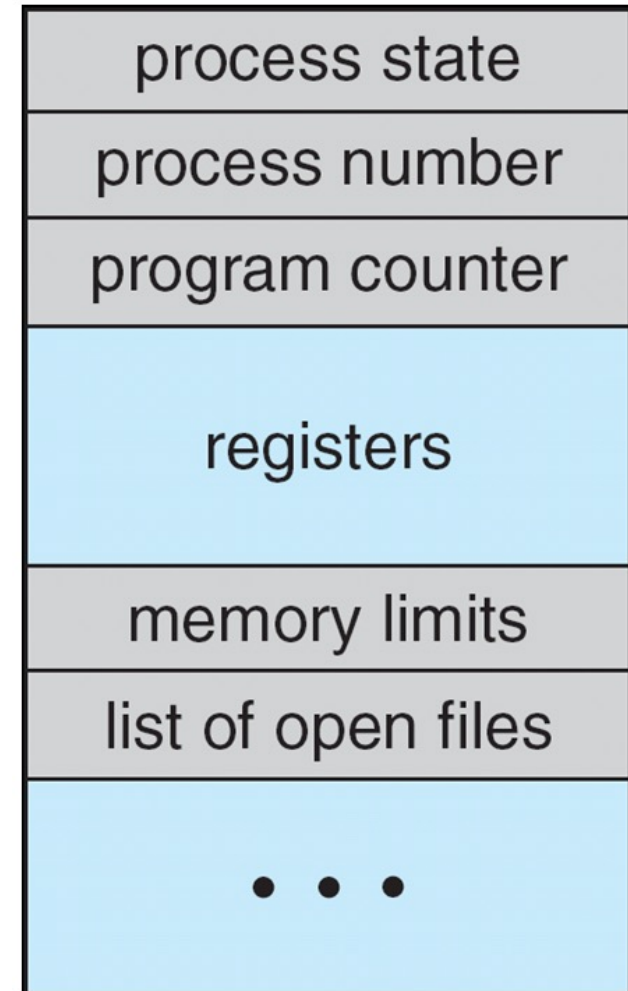
# Process in Memory



# Process Control Block (PCB)

PCB – Stores all the information associated with each process (also called **task control block**)

- **Process state** – running, waiting, etc.
- **Process number** – Process ID
- **CPU registers and program counter** – contents of all process-centric registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- And much more ....



# Process Representation in Linux

Processes in Linux are referred to as tasks.

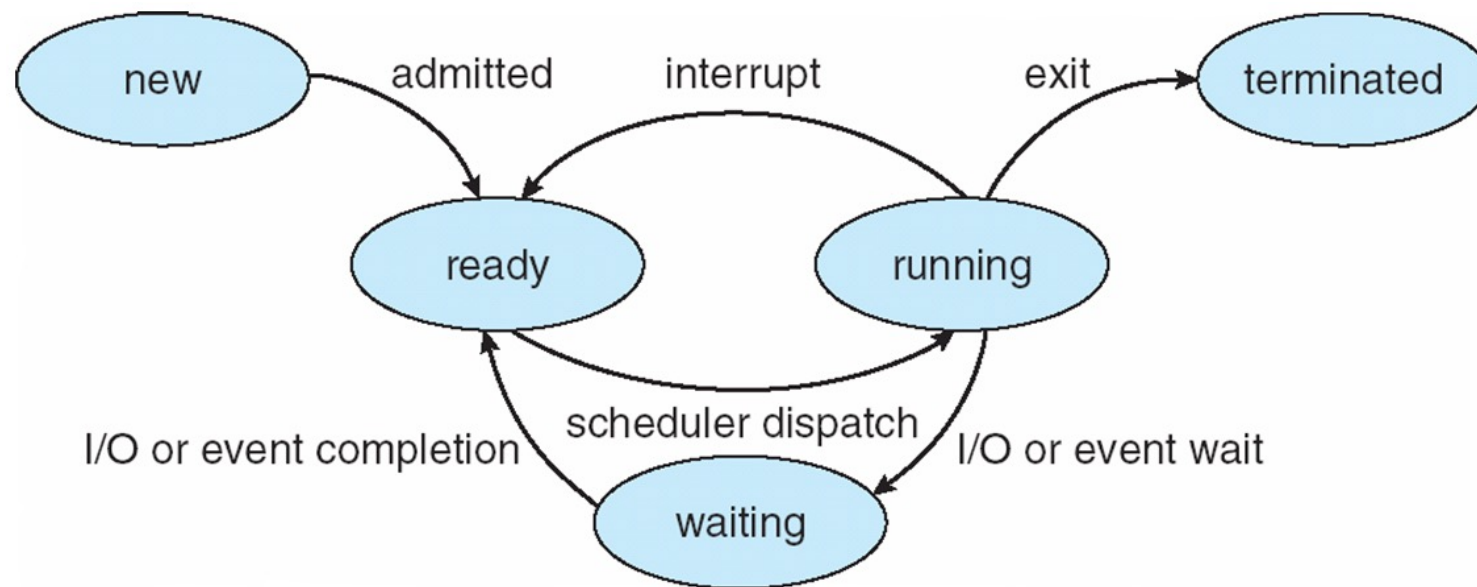
Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

# 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

# Diagram of Process State





# Types of Processes

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

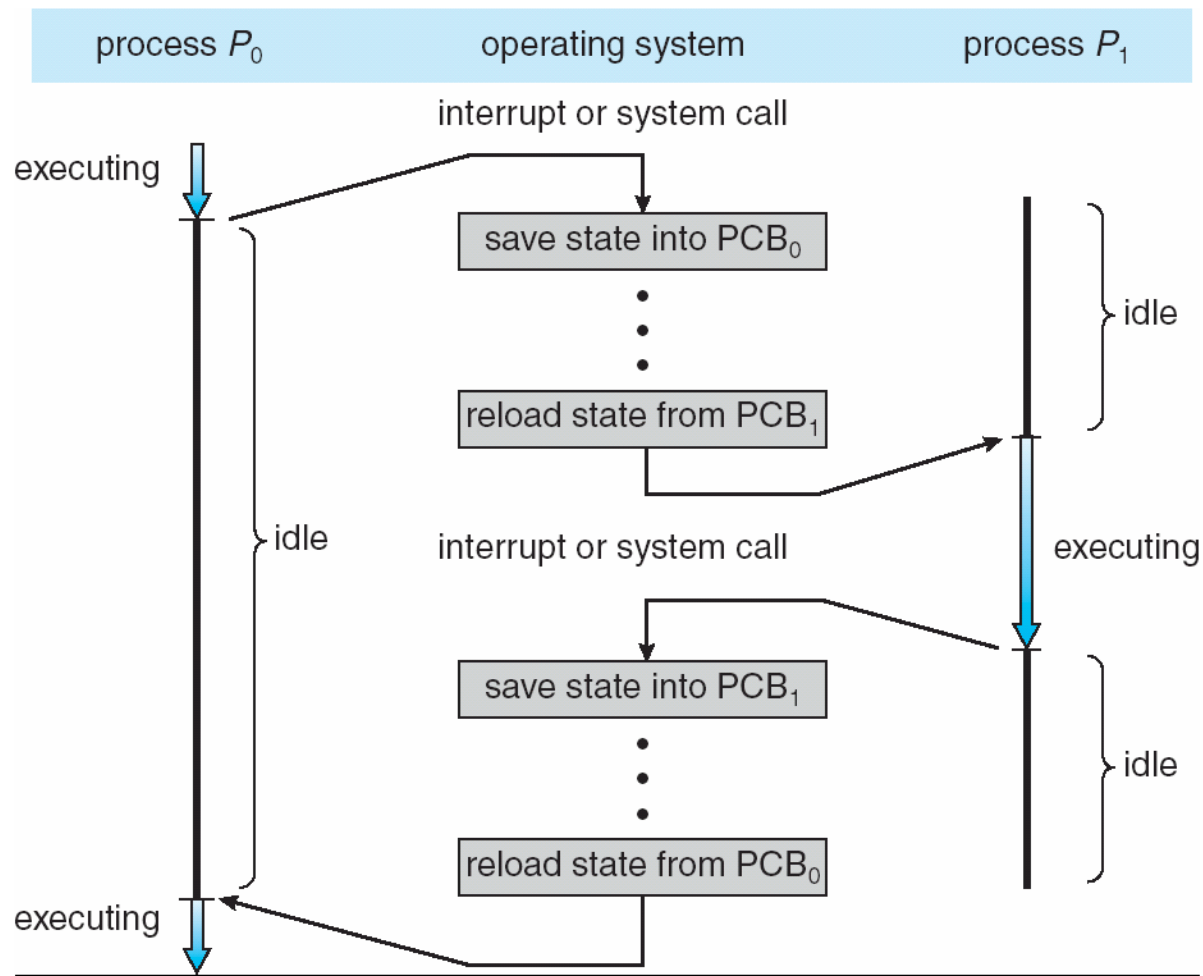
# Threads

- Modern systems allow a process to have multiple threads associated with it.
- These threads can execute concurrently.
- More on Threads in the next chapter.

# Context Switch

- **Context** of a process is represented in the PCB (see slide# 5)
- When CPU switches to another process, the system must **save the context** of the old process and load the **saved context** for the new process. This process is called **context switching**
- Context-switch time is an 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

# CPU Switch From Process to Process



# Operations on Processes

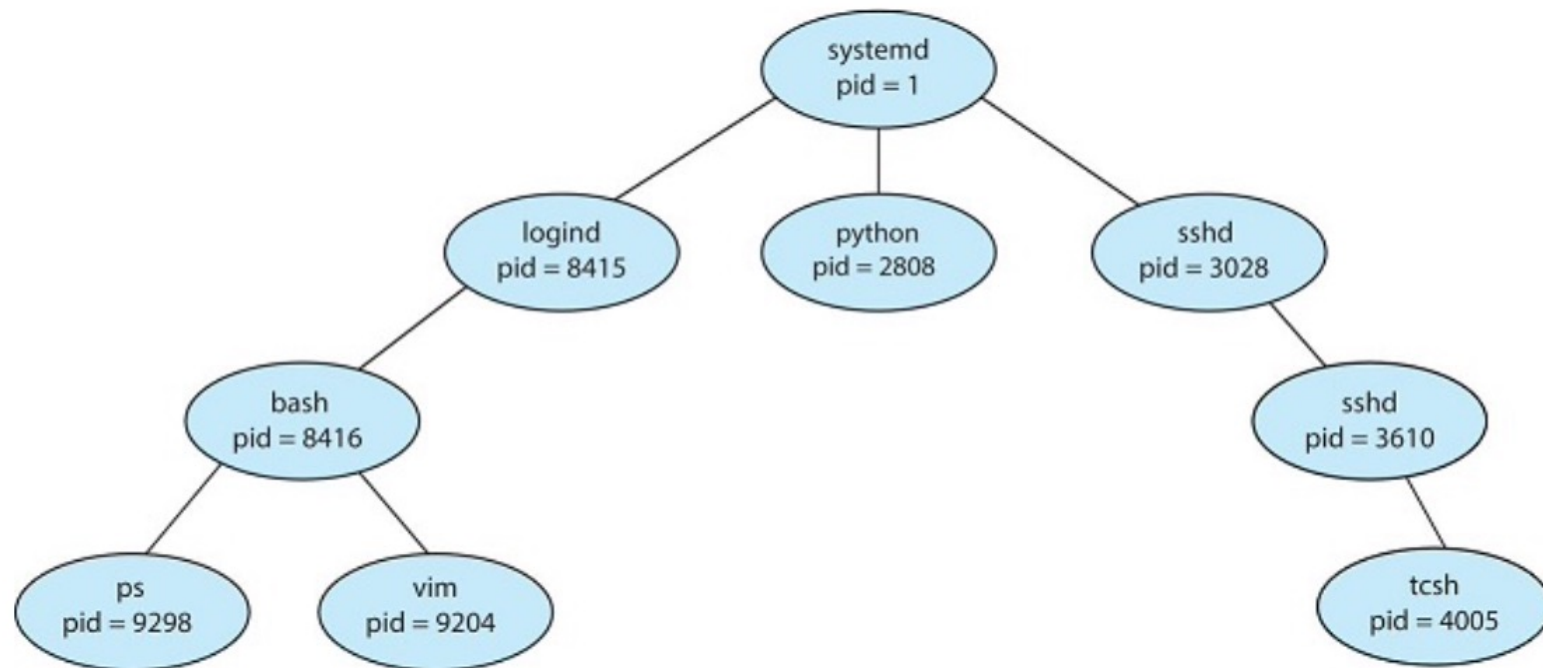
- System must provide mechanisms for:
  - process creation,
  - process termination

# Process Creation

- Every process is given an integer identifier called the **process identifier (pid)**:
- A process (**parent process**) can create another process (**child process**).
- **Parent** process creates **child** processes (using system calls), which, in turn creates other processes, forming a **tree of processes** or **process tree**.
- In addition to PID of a process, its parent PID (termed as PPID) is stored as well.

# A Tree of Processes in Linux

- At system startup, `init` or `systemd` (in newer Linux distributions) process is executed with process identifier is 1.
  - `init` then launches all system daemons and user logins and becomes the parent of all other processes.



# Parent – Child Sharing

- **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

- **Address space sharing options:**

- Child is a duplicate of parent (has the same program and data as the parent)
- Child has a new program loaded into it



# Creating Processes in Linux/Unix

- `fork()` system call is used to create a **new** process
  - `fork()` takes no arguments
- The new process created by `fork` becomes the **child** process of the calling process **parent**.
- Only if the creation of the child process was unsuccessful, `fork()` returns a negative value.
- This child process has the same environment as its parent; that is, it is an exact copy of the parent with only a different process ID.
  - The child and parent processes have different address space.
- After a new child process is created, both the parent and child will execute the next instruction following the `fork()` system call.

# Question 1

How many processes (including the parent) are created when the below code is executed?

Draw the process tree for the below code.

```
int main()
```

```
{
```

```
    fork() ;
```

```
    fork() ;
```

```
    fork() ;
```

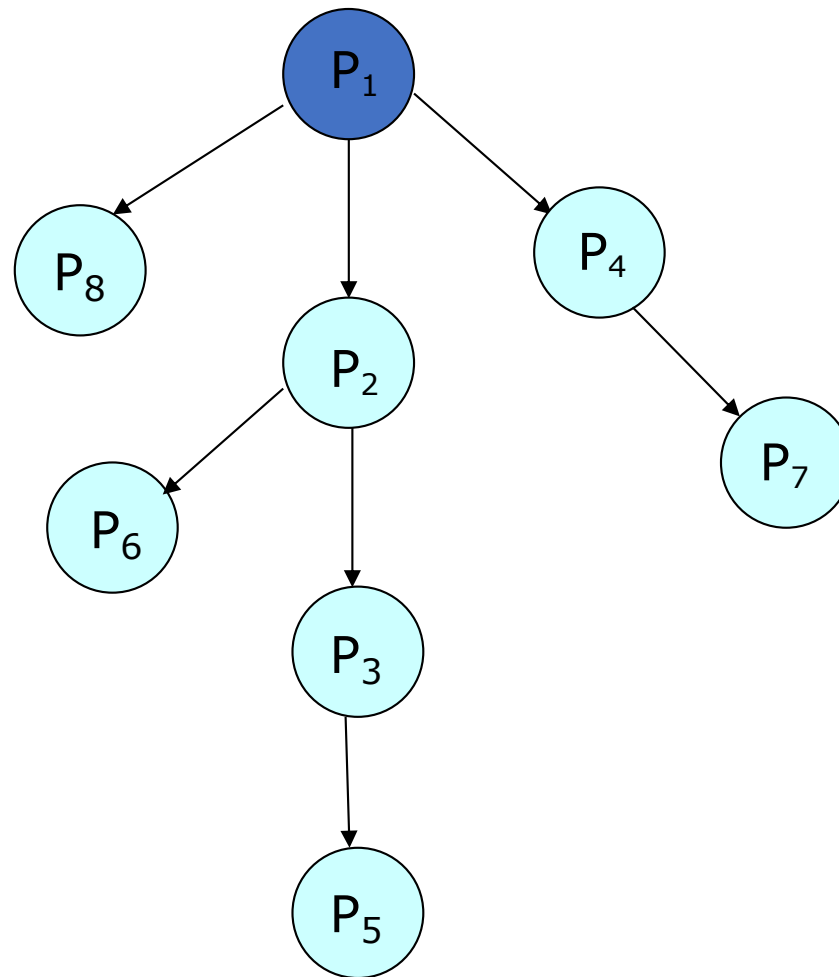
```
    return 0 ;
```

```
}
```

After executing this `fork()`, the child process created from it, executes instructions starting from the next statement; that is, `fork()`.

`return 0;` returns the exit status of main, where 0 indicates the program executed normally.

## Question 1 - Process Tree



# Distinguishing Child and Parent Process

- **How do we distinguish between a child process and parent process?**
  - `fork()` returns a **zero in the child process**
  - `fork()` **returns the PID of the child process in parent**
- **How to get the PID of process and parent?**
  - Every process can query its own PID using the `getpid()`.
  - Every process can query its parent PID using `getppid()`.

## Question 2

**How many processes (including the parent) are created when the below code is executed?**  
**Draw the process tree for the below code.**

```
int main()
```

```
{
```

```
    pid_t pid;
```

```
    pid = fork();
```

```
    if(pid == 0) { /*Child process*/
```

```
        printf("Child Process with PID: %d\n",getpid());
```

```
    }
```

```
    else { /*Parent process*/
```

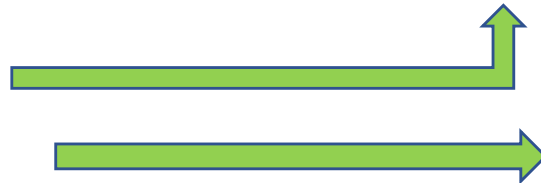
```
        printf("Parent Process with PID: %d\n",getpid());
```

```
    }
```

```
    return 0;
```

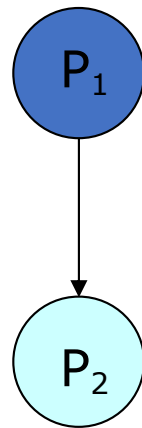
```
}
```

pid\_t is the type to store Process ID



fork() returns the PID of the child process in the parent process and returns 0 in the child process.

## Process tree for Question 2



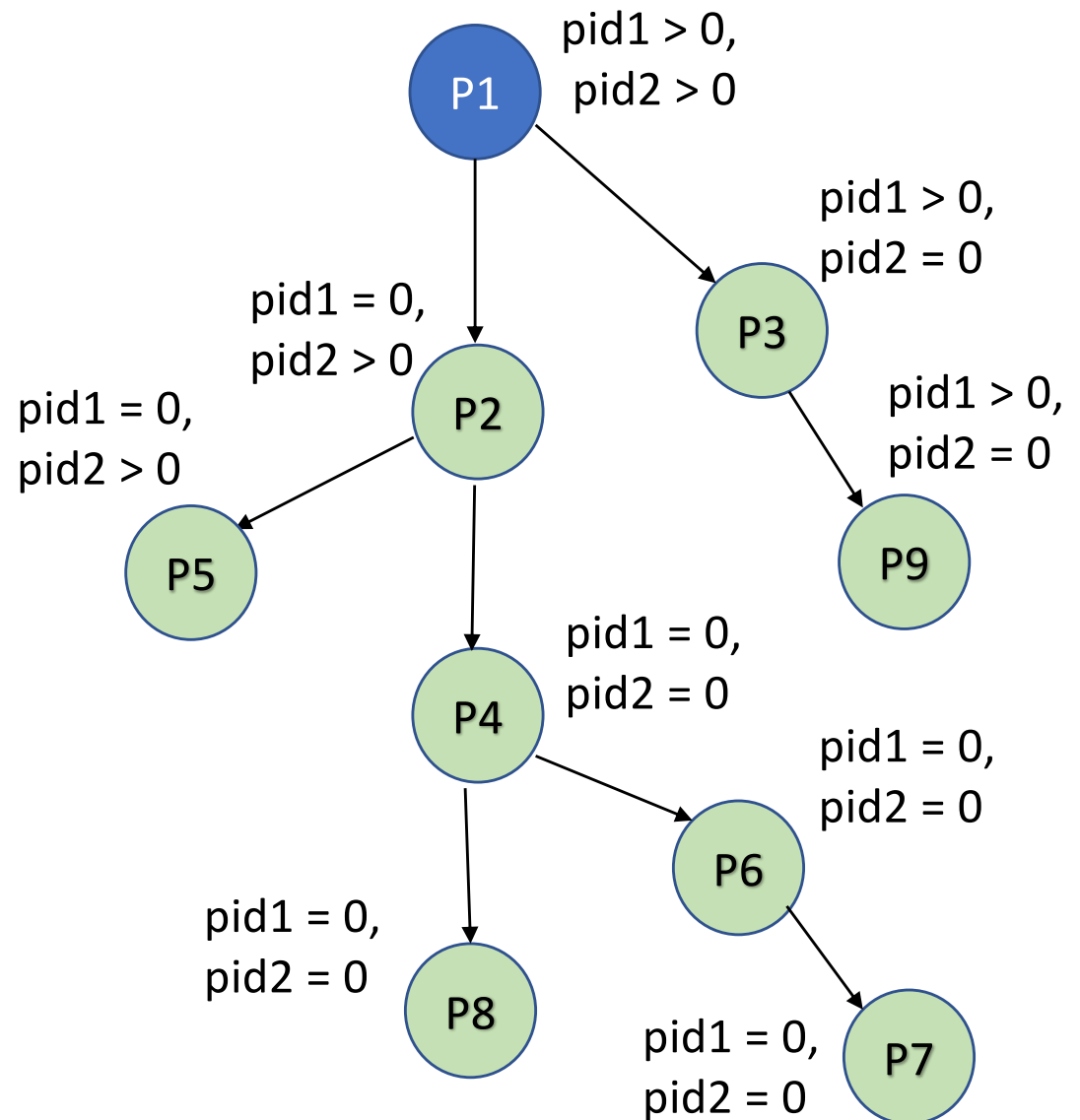
## Question 3

```
int main() {  
    pid_t pid1, pid2;  
    pid1 = fork();  
    pid2 = fork();  
    if (pid1 == 0) {  
        fork();  
    }  
    if (pid2 == 0) {  
        fork();  
    }  
    return 0;  
}
```

Draw the process for the code in question on the left and indicate pid1 and pid2 for each process created.

# Process Tree for Question 3

```
int main() {  
    pid_t pid1, pid2;  
    pid1 = fork();  
    pid2 = fork();  
    if (pid1 == 0) {  
        fork();  
    }  
    if (pid2 == 0) {  
        fork();  
    }  
    return 0;  
}
```





# Question 4

What is the output at Line A and why?

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

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        return 0;
    }
}
```

# Question 5

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

int main()
{
    pid_t pid, pid1;

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

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d",pid); /* C */
        printf("parent: pid1 = %d",pid1); /* D */
        wait(NULL);
    }
    return 0;
}
```

What are the values of pid at lines A, B, C, and D? Assume that the actual pids of the parent and child are 2600 and 2603, respectively

## exec() system call

- The **exec()** system call used right after **fork()** enables the child process to execute a different program than the one it inherits from its parent process.
- This act is also referred to as an **overlay**.
- There is a family of exec() functions, all of which have slightly different characteristics. See the Linux manual for more details.

# Process creation using the fork() and Exec() system calls

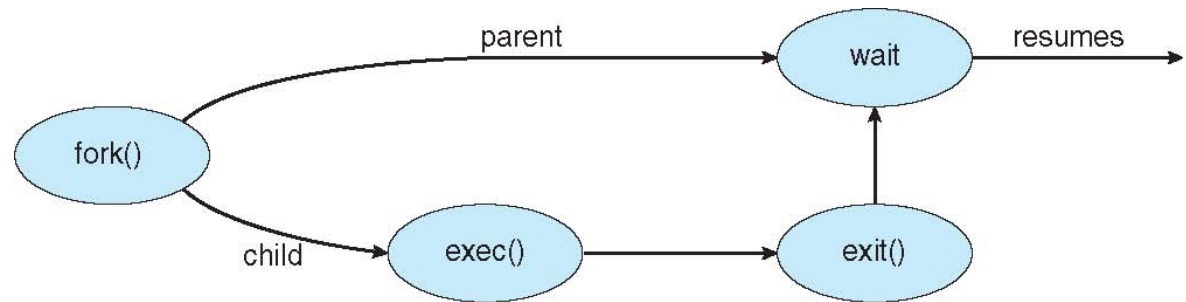
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

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

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

    return 0;
}
```



# Process Termination

- After executing the last statement, a process is terminated
  - Implicitly – using the **return** statement
  - Explicitly – using the **exit()** system call
- Process' resources are deallocated by operating system

# Child Process Termination

- 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
- Sometimes the operating systems does not allow a child to continue if its parent terminates. Therefore,
  - The OS initiates **cascading termination** - All children, grandchildren, etc. are terminated.

# Child Process – Termination status

- When a child process terminates, the parent can know the child's exit status using the `wait()` system call.
- The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- Unix/Linux maintains a **table of processes**. This table contains the list of all processes running and includes the process status.
- If a parent process terminates, its entry is removed from the table.
- If a child process terminates, its entry is removed from the table **only after the parent process invokes a `wait()`**.
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait()`, process is an **orphan**

# Process Scheduling

We will discuss this in detail in Chapter 5

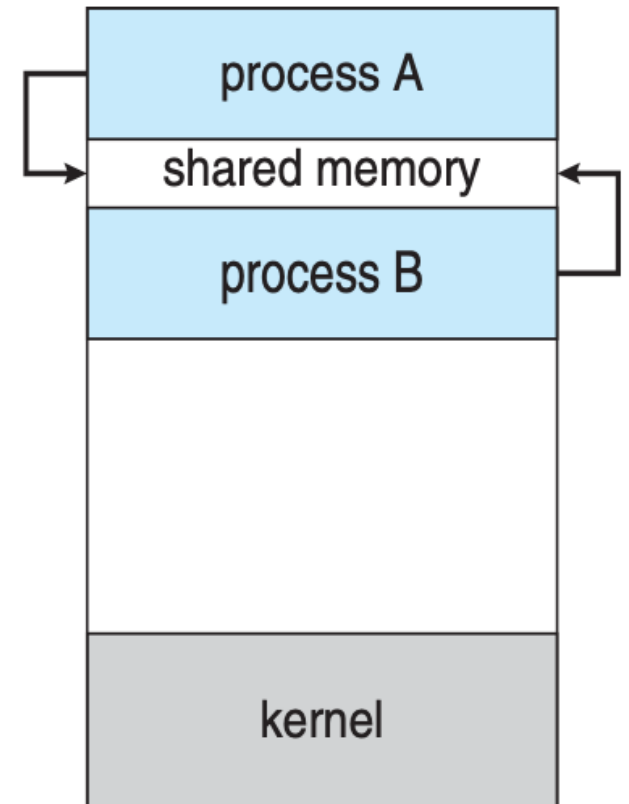


# Interprocess Communication (IPC)

- Processes within a system may be
  - **Independent** or
  - **Cooperating** (*it can affect or be affected by other processes, including sharing data*)
- Reasons for cooperating processes:
  - Information sharing, computation speedup, modularity and convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

# IPC – Shared Memory

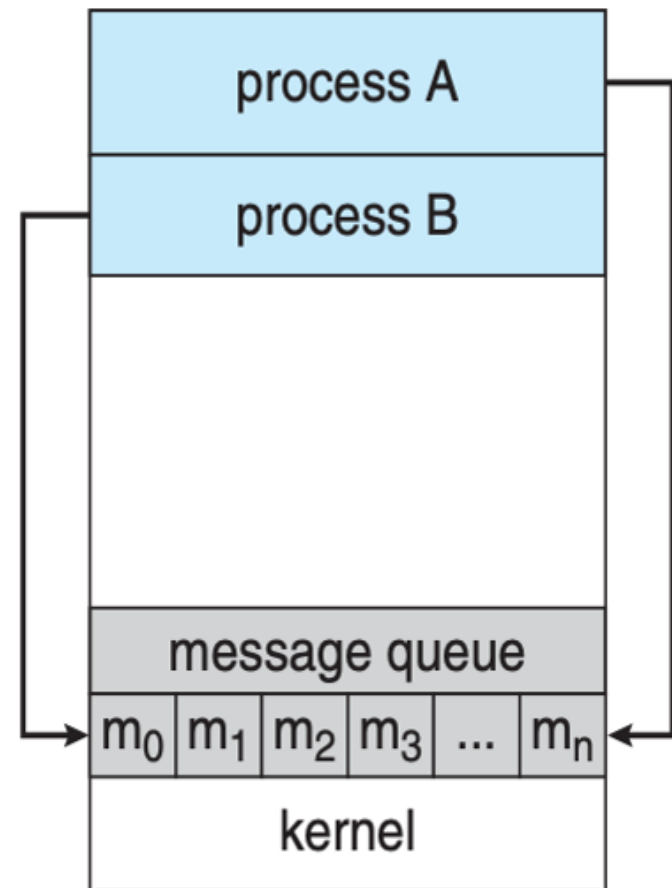
- An area of memory shared among the processes that wish to communicate
- The communication is under the **control of the user** processes not the operating system.
- It is more complicated to set up and doesn't work as well across multiple computers.
- Used for sharing large amount of data.
- Major issues – synchronize process actions when accessing shared memory.



**Shared Memory Model**

# IPC – Message Passing

- Operating system provides message passing capability.
- As a result, Message Passing requires system calls for every message transfer, and is therefore slower.
- However, it is simpler to set up and works well across multiple computers.



**Message Passing model**