Operating Systems

Process

FBA

Process

- Book section: 3.1.1, 3.1.2, 3.1.3, 3.2 (full), 3.3.1, 3.3.2, 3.4, 3.4.1, 3.4.2

Process Concept

What to call the activities of CPU ?

Jobs	User Programs or Tasks
Batch System	Time Sharing System

These activities are called "**Processes**"

★ The terms "***job***" and "***process***" are used almost interchangeably.

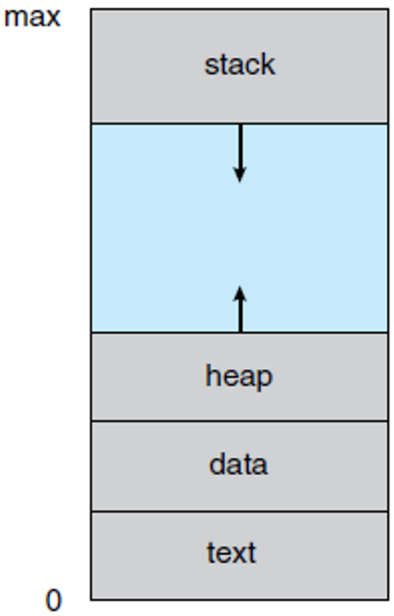
Process

A process is a program that is in execution.

But, it is more than the program codes. Program code is known as "text section" of a process.

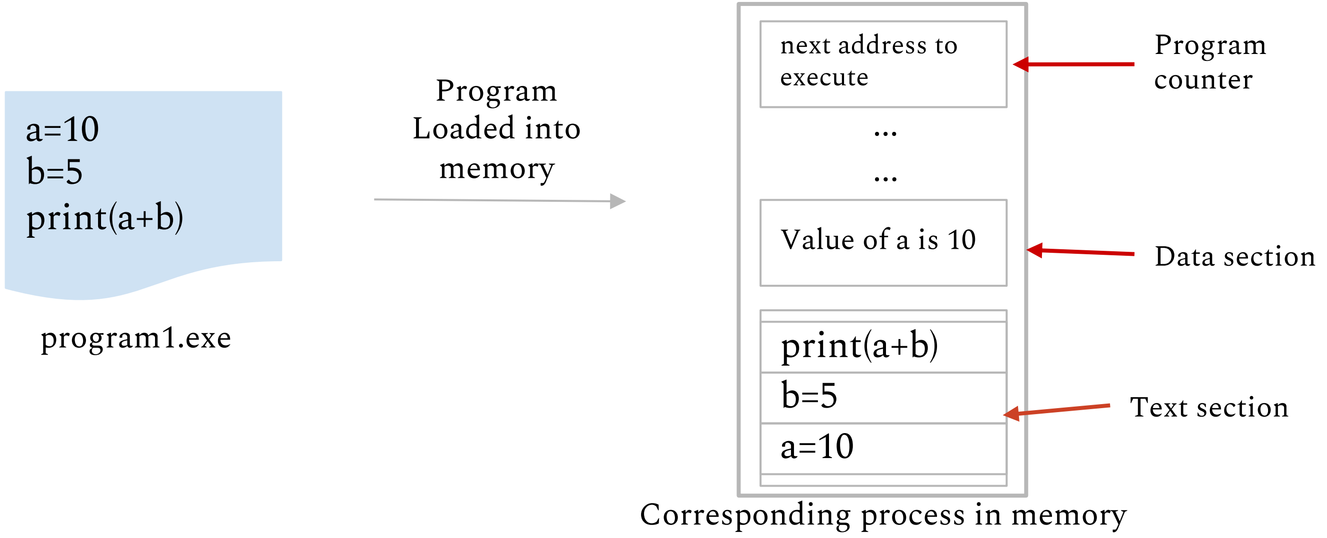
Besides code of the program, it contains -

- **Program Counter and Registers:** stores current activity of the process
- **Stack:** Temporary data (function parameter, local variables, return addresses etc.)
- **Data Section:** Global Variables
- **Heap:** dynamically allocated memory during runtime

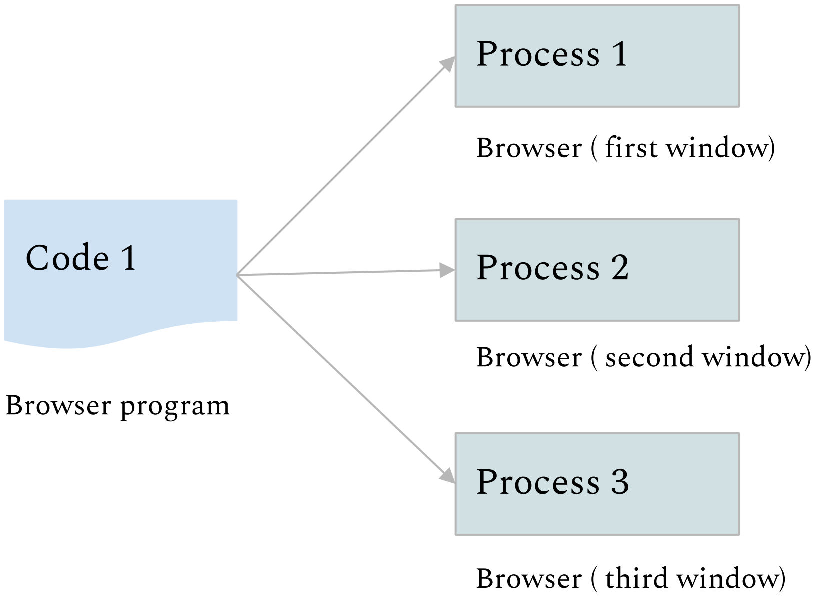


Program Vs Process

- Program is a collection of instructions that can be executed
- A program is a **passive** entity.
- A process is an **active** entity.
- A program becomes a process when it is loaded into memory for execution.



Same program, Different Process



- Program code is same
- Data, Heap, Stacks contains different information

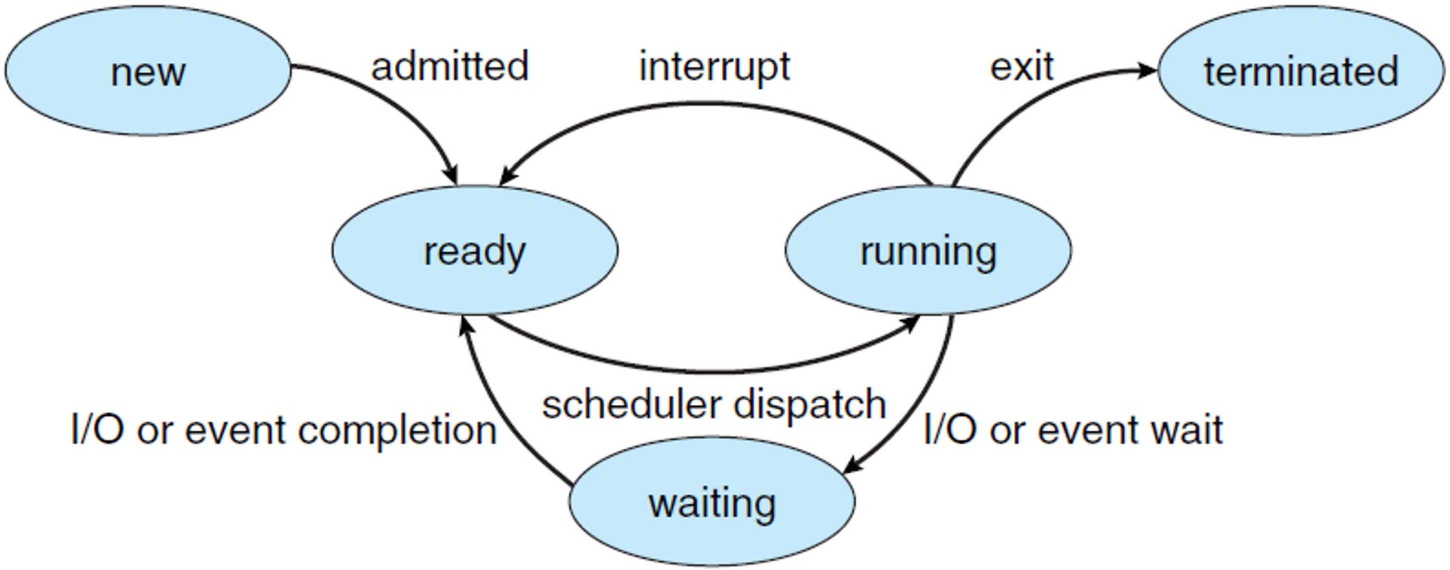
States of a Process

A process state defines the current activity of that process.

The states a process can be:

- **New**: Process is being created
- **Running**: Instructions are being executed
- **Waiting**: Process is waiting for some event to occur
- **Ready**: Waiting to be assigned to a processor
- **Terminated**: Process has finished execution

Process State Diagram



Representation of Processes in OS

Each process is represented in the operating system by a ***Process Control Block (PCB)***

PCB is a data structure to store information of Processes such as -

Process state

Program counter

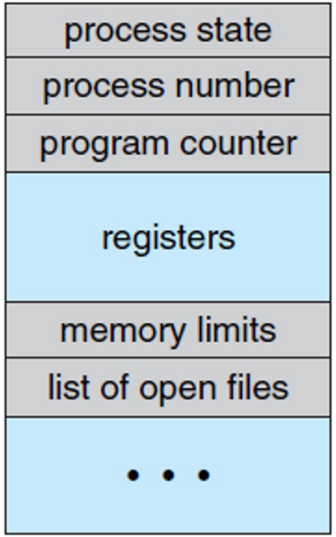
CPU registers

CPU scheduling information

Memory-management information

Accounting information

I/O status information



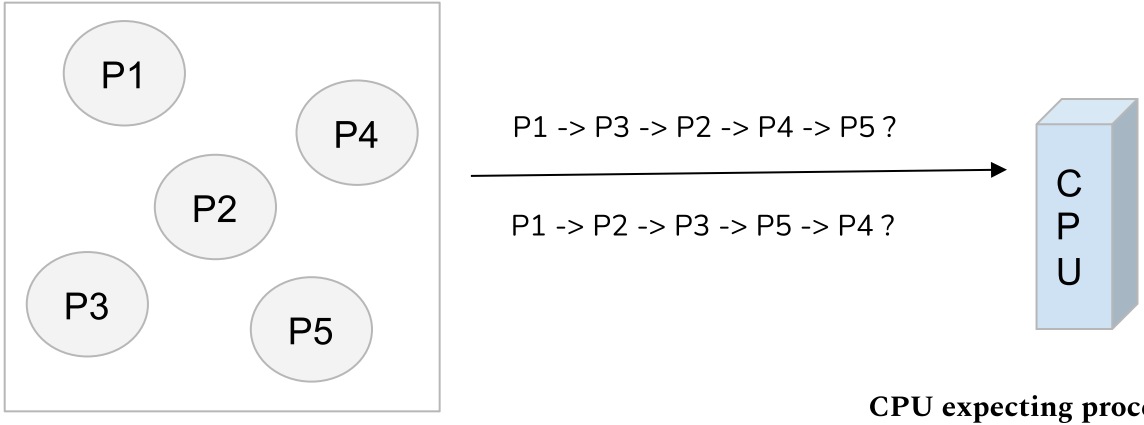
Operating Systems

Process Scheduling

FBA

Process Scheduling

Multiple process is ready to execute.
But, which Process should be executed first?



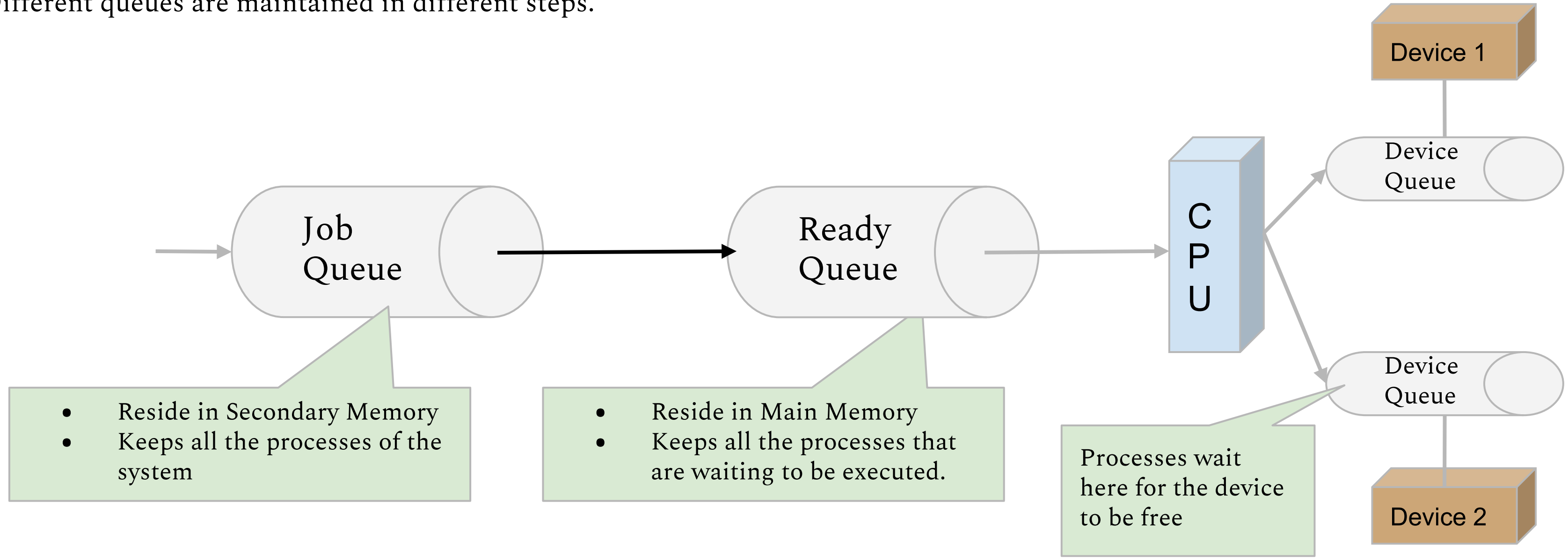
Processes needs to be executed

CPU expecting processes to execute

Scheduling Queue

Stores the processes in different steps of OS.

Different queues are maintained in different steps.



Queueing Diagram

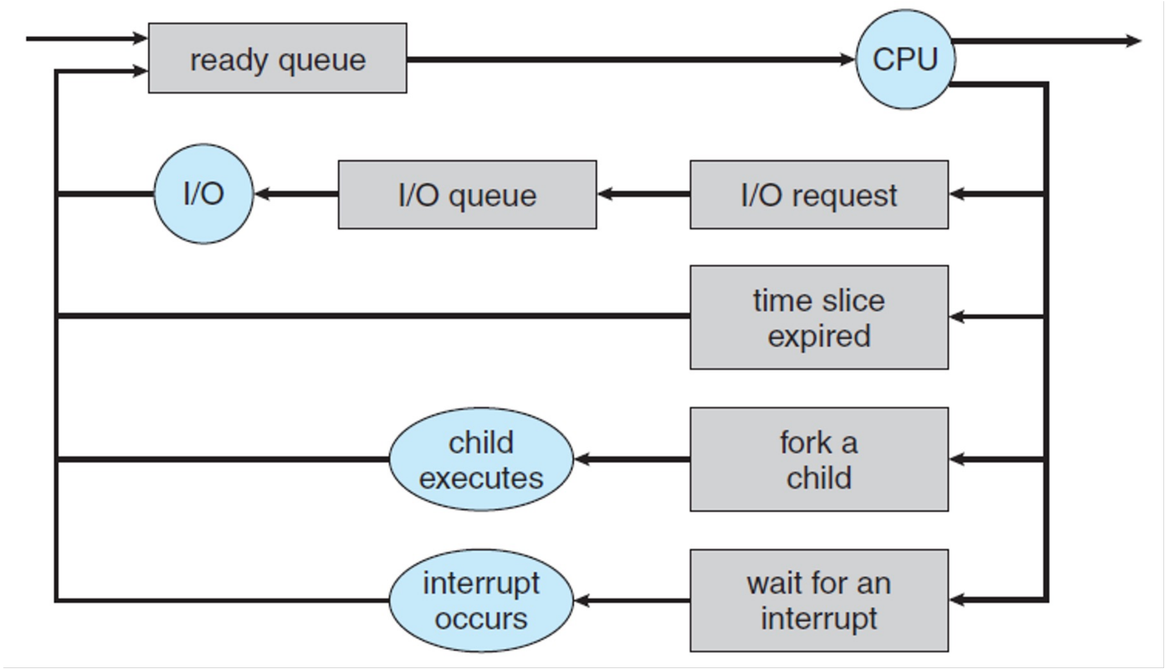
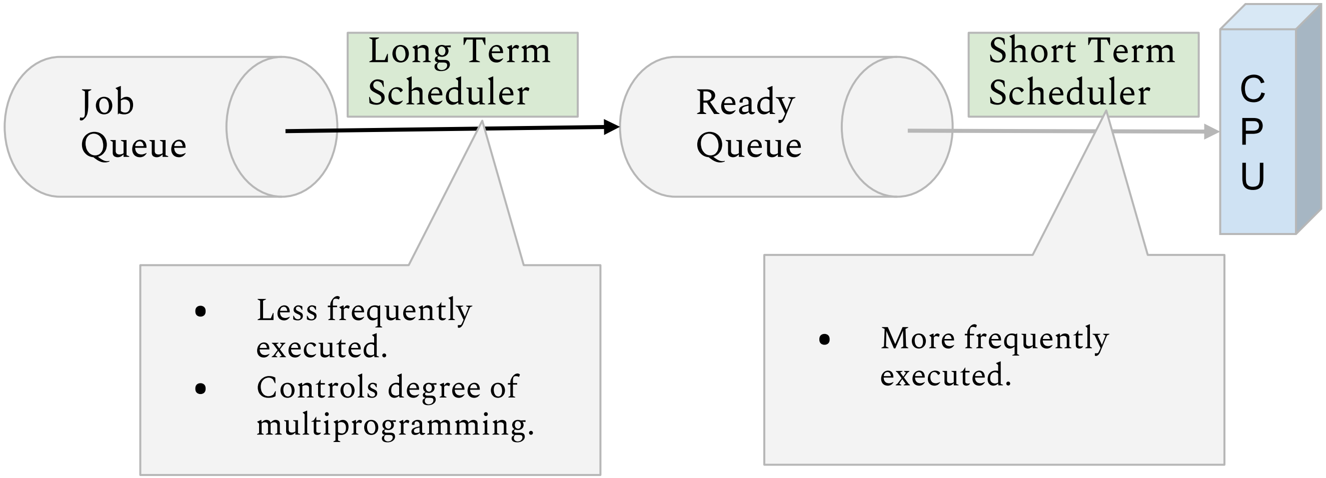


Fig: Representation of Process Scheduling using Queueing-Diagram

Schedulers

Schedulers select processes from different queues to be passed to the next phase.



Long term scheduler determines the degree of multi-programming!

CPU Bound Vs I/O Bound Process

- CPU bound processes spend more time doing computation using processors than I/O.
- I/O bound processes spend more time in I/O than CPU.

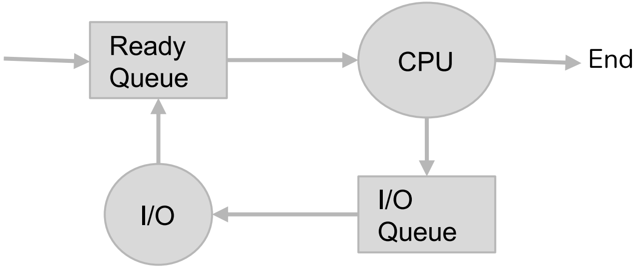
Long Term Scheduler must select wisely !

- What will happen if all processes are I/O bound ?

=> Empty ready queue

- What will happen if all processes are CPU bound ?

=> Empty waiting queue



Medium Term Scheduler

- Time-sharing system may use this scheduler.
- Swapping reduce the degree of multiprogramming.

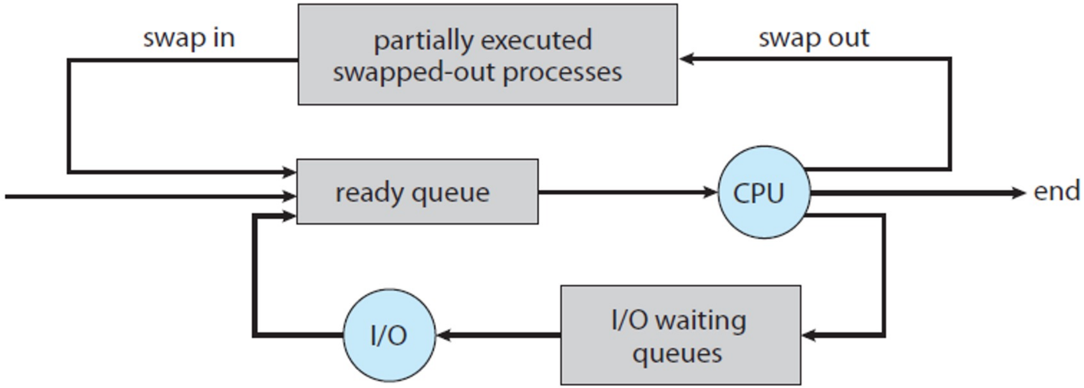
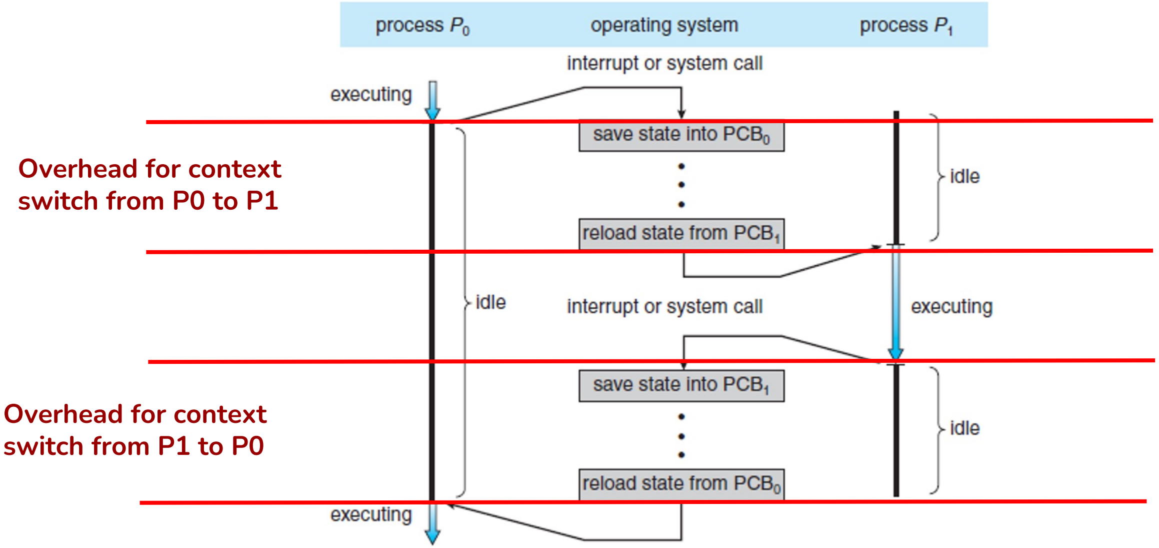


Fig: Addition of swapping in Queueing-Diagram

Context Switch

When an interrupt occurs, the system needs to save the current **context** (state) of the process running on the CPU.

Context Switch: 1. Storing currently executed process context
2. Restoring the next process context to execute



Operating Systems

Operations on Process

FBA

Process Creation

- A process is identified by a unique PID (Process Identifier) in the OS.
- A process may create new processes.

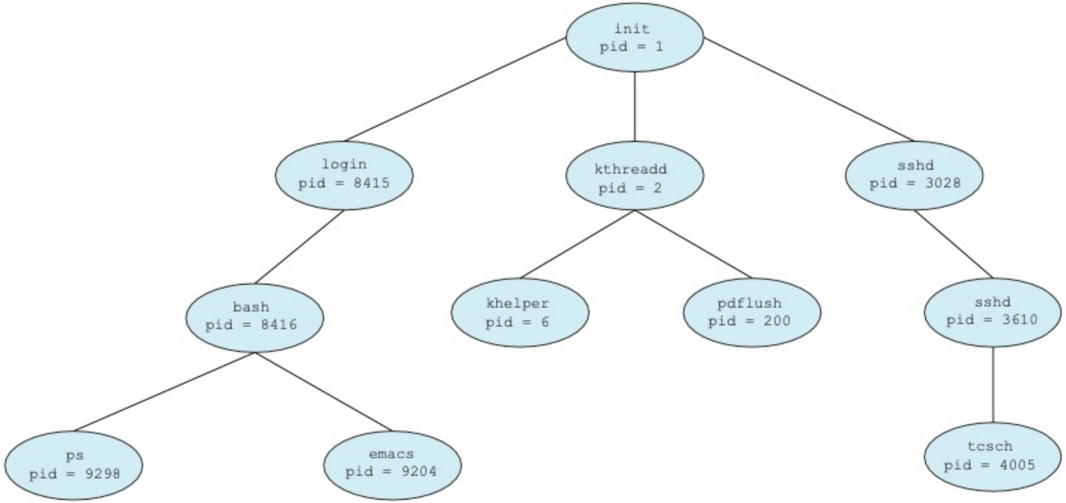
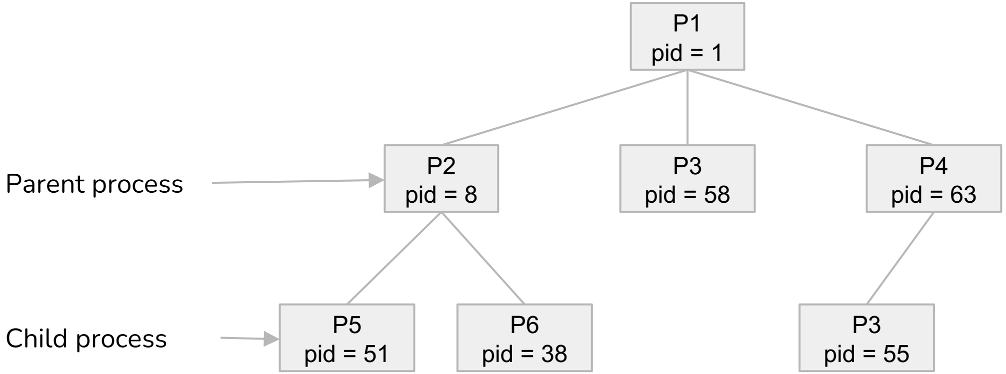


Figure 3.8 A tree of processes on a typical Linux system.

Process Creation



- Child process obtain resources from OS or are restricted to Parent's resources
- Parent process may pass initializing data to child process

Process Creation

- When a process creates new process -

The parent continues to execute concurrently with its children
Or,
The parent waits until some or all of its children have terminated

- Two address-space possibilities for the new process -

The child process is a duplicate of the parent process
Or
The child process has a new program loaded into it.

Process creation in UNIX

System Call: offers the services of the operating system to the user programs.

***fork()*:** create a new process, which becomes the child process of the caller

***exec()*:** runs an executable file , replacing the previous executable

***wait()*:** suspends execution of the current process until one of its children terminates.

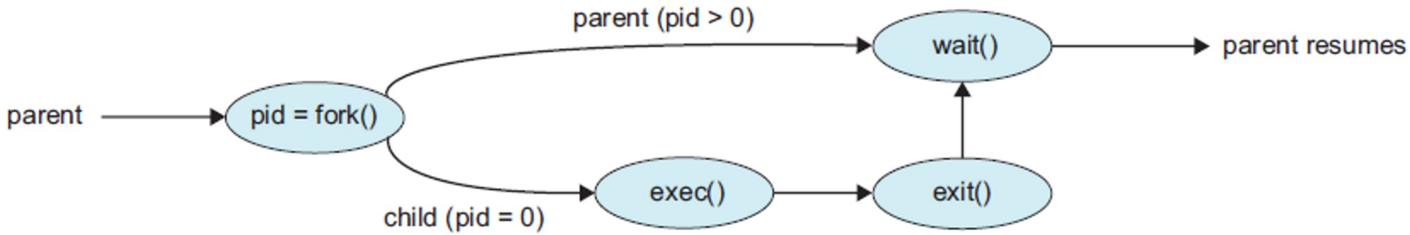


Fig: Process creation using fork() system call

```
#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;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

```c
#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;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

Parent Process

```c
    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;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

Child Process

```c
#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;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

```
/tmp/bVjXFlwKBr.o
events.txt  follow_course.csv  inpud  students.dat
Child Complete
```

```c
int main(){
	fork();
	fork();
	printf(“A”);
}
```

```c
int main(){
	fork();
	fork();
	fork();
	printf(“A”);
}
```

```
int main(){
        a = fork();
        if(a==0)
fork();
        fork();
        printf("A");
}
```

```
int main(){
        fork();
         a = fork();
         if(a==0)
fork();
        printf("A");
}
```

```
int main(){
	int x = 1;
	a = fork();
	if(a==0){
		x = x -1;
		printf(“value of x is: %d”, x);
	}
	else if (a>0){
		wait(NULL);
		x = x +1;
		printf(“value of x is: %d”, x);
	}
}
```

Process Termination

A process is terminated when -

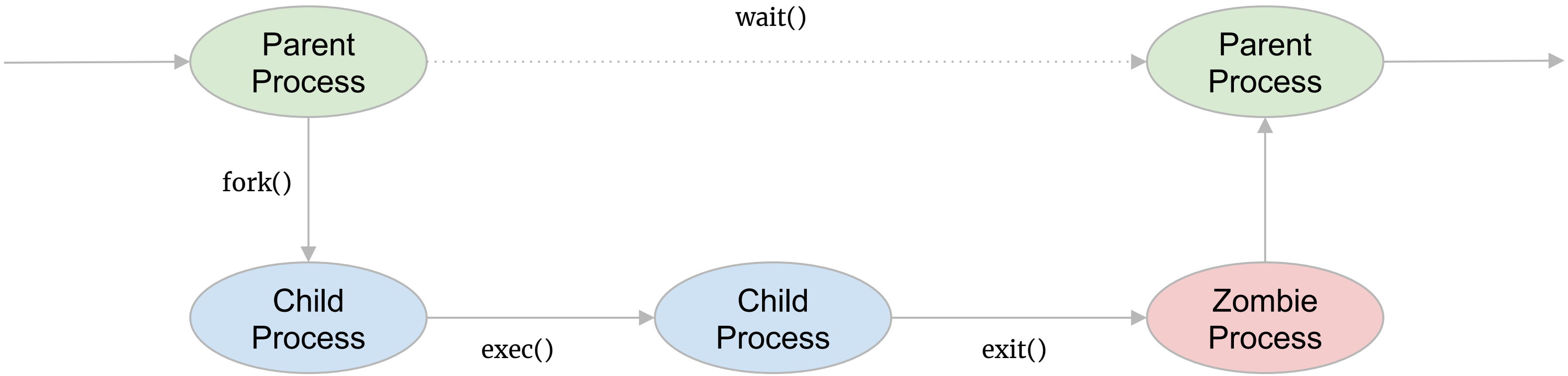
It executes its last statement
Or
Termination caused by another process

When a process is terminated, the resources are deallocated.

A parent may terminate its child if -

1. Child has exceeded the usage of resources
2. Task assigned to child is no longer needed
3. Parent is exiting (cascading termination)

Zombie Process in UNIX



Operating Systems

Interprocess Communication

FBA

Processes in the system

Processes running concurrently may be -

Independent (cannot affect or be affected by other process)

Or

Cooperating (can affect or be affected by other process)

Process cooperation is needed for -

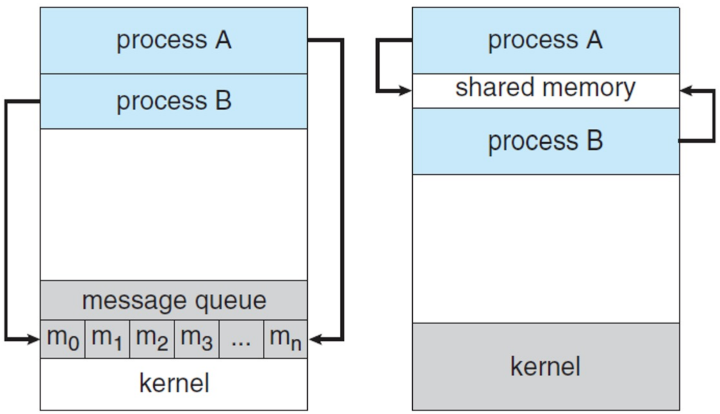
- ➔ Information sharing
- ➔ Computational speedup
- ➔ Modularity
- ➔ Convenience

Inter Process Communication

IPC is a ***mechanism*** to exchange data and information among processes.

Two fundamental model of IPC -

1. Shared Memory
2. Message Passing

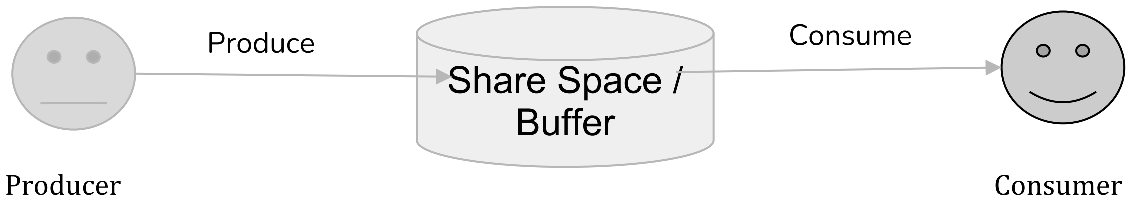


Shared Memory System

(Producer-Consumer Problem)

Producer: produces products for consumer

Consumer: consumes products provided by producer



Producer-Consumer Problem (Producer)

```
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;
}
```

in: next free position in buffer
out: first full position in buffer

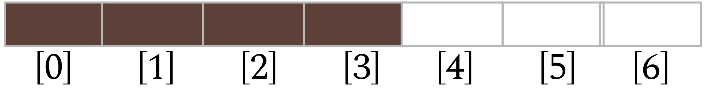
Both initialized with 0.

in = 0
out = 0

Here, BUFFER_SIZE = 7

When buffer is full,
in = 6 , out = 0

When buffer is not full,
in = 4, out = 0



Producer-Consumer Problem (Consumer)

```
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 */
}
```

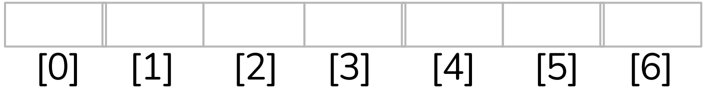
in: next free position in buffer
out: first full position in buffer

Both initialized with 0.

in = 0
out = 0

Here, BUFFER_SIZE = 7

When buffer is empty,
in = 0 , out = 0

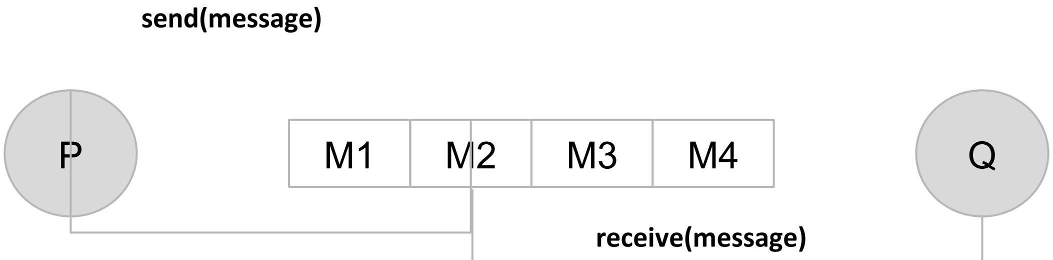


When buffer is not empty,
in = 5, out = 0

Message Passing System

If processes P and Q want to communicate, they must *send* messages to and *receive* messages from each other.

A communication link must exist between P and Q.



- Useful for exchanging small amount of data
- More suited for distributed systems than shared memory