Operating Systems

Process

FBA

Process Concept

What to call the activities of CPU ?

Batch System	Time Sharing System
Jobs	User Programs or Tasks

These activities are called "**Processes**"

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

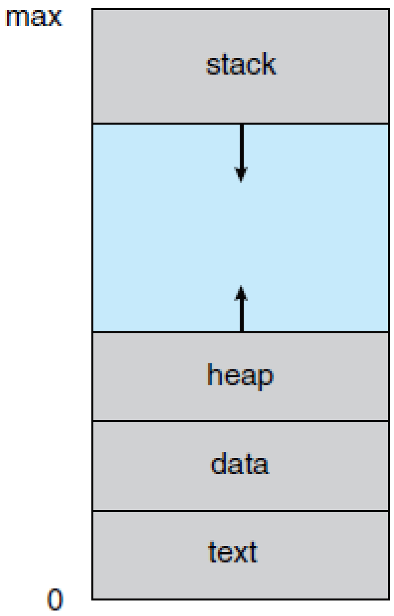
Process

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.

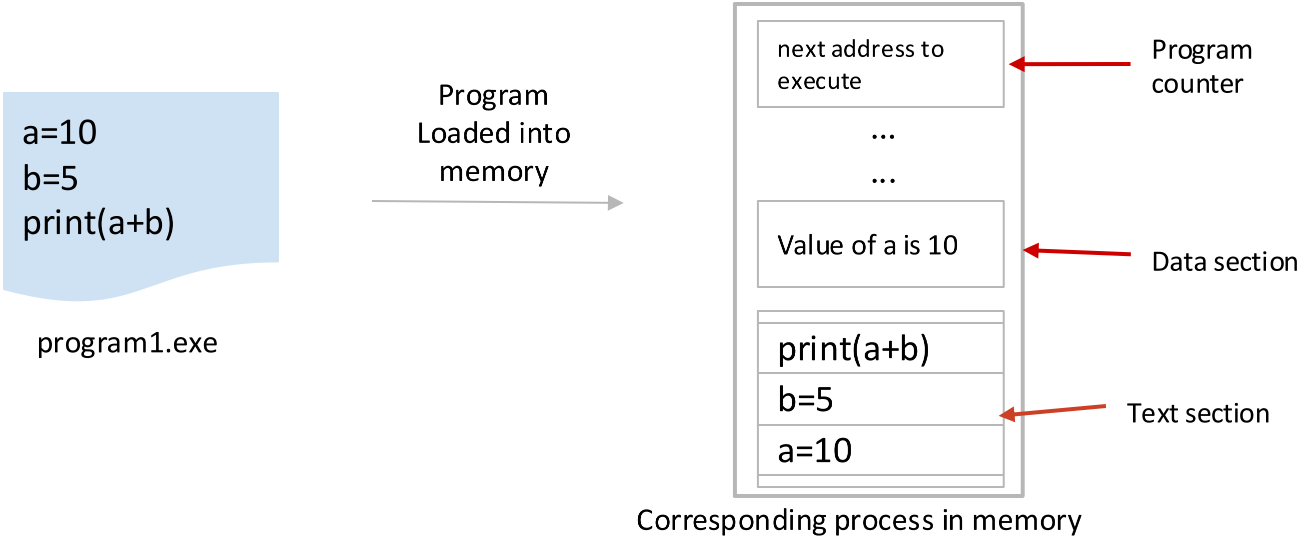
Besides code of the program, it contains -

- **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

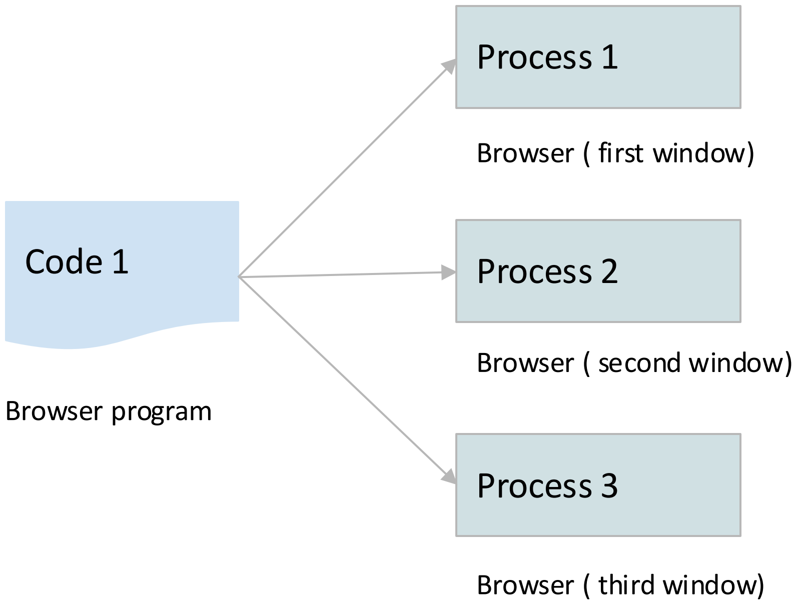


Program Vs Process

- Program is passive entity stored on disk (executable file); process is active
    - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
    - Consider multiple users executing the same program



Same program, Different Process



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

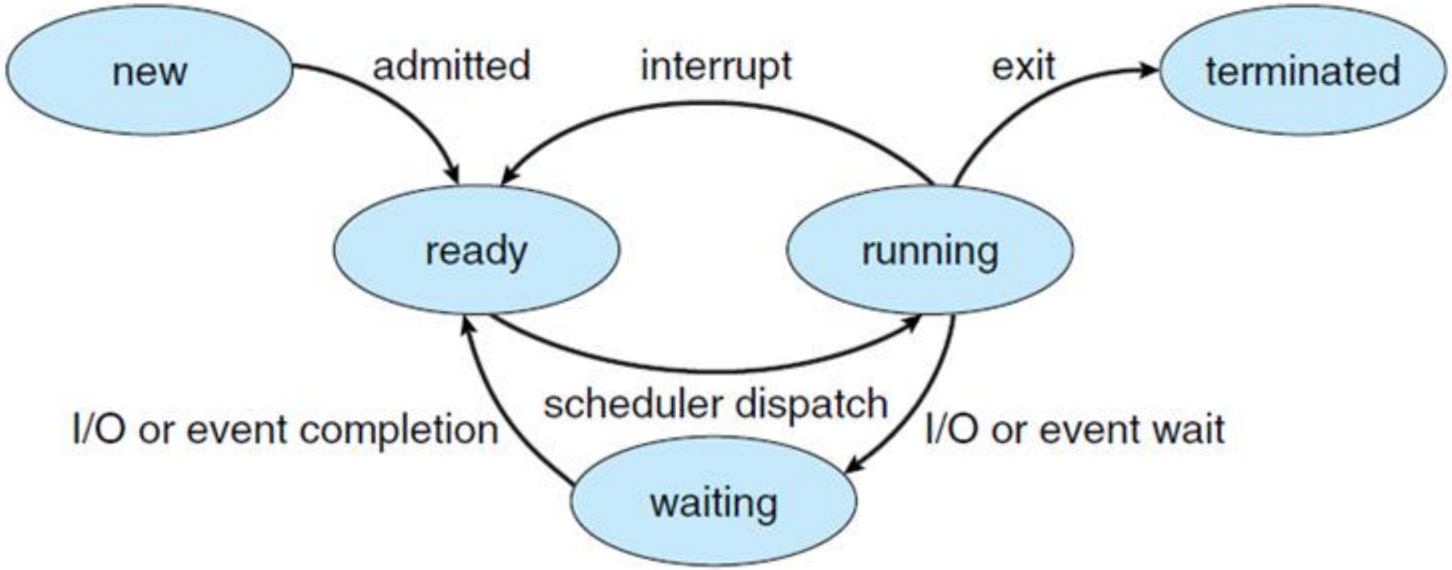
States of a Process

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

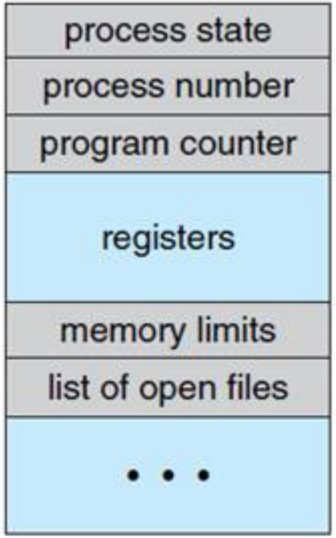


Representation of Processes in OS

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

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

- **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



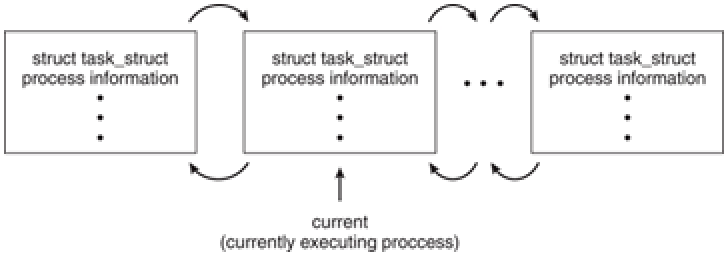
Threads

- ➔ So far, process has a single thread of execution
- ➔ Consider having multiple program counters per process
    - ◆ Multiple locations can execute at once
        - ● Multiple threads of control -> threads
- ➔ Must then have storage for thread details, multiple program counters in PCB
- ➔ Explore the details later

Process Representation in Linux

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 */
```

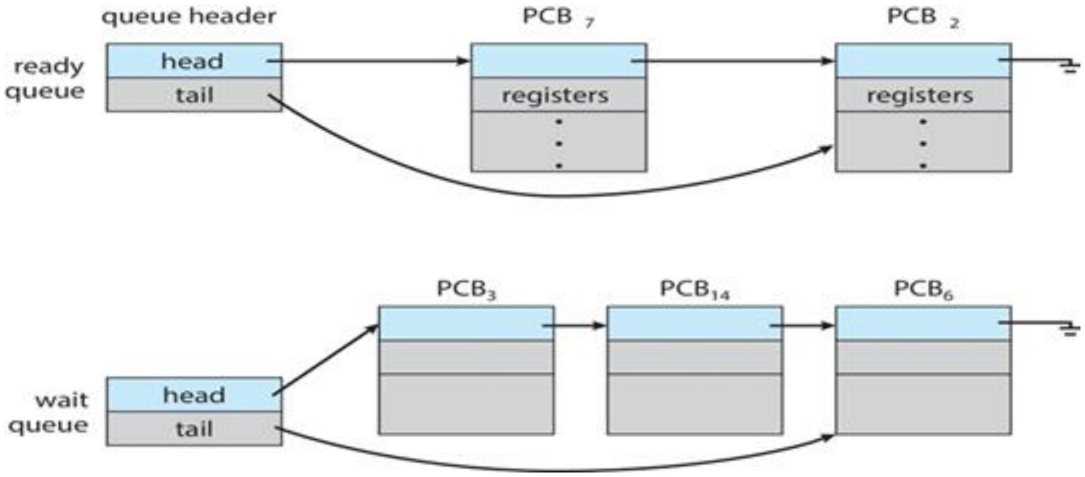


Operating Systems

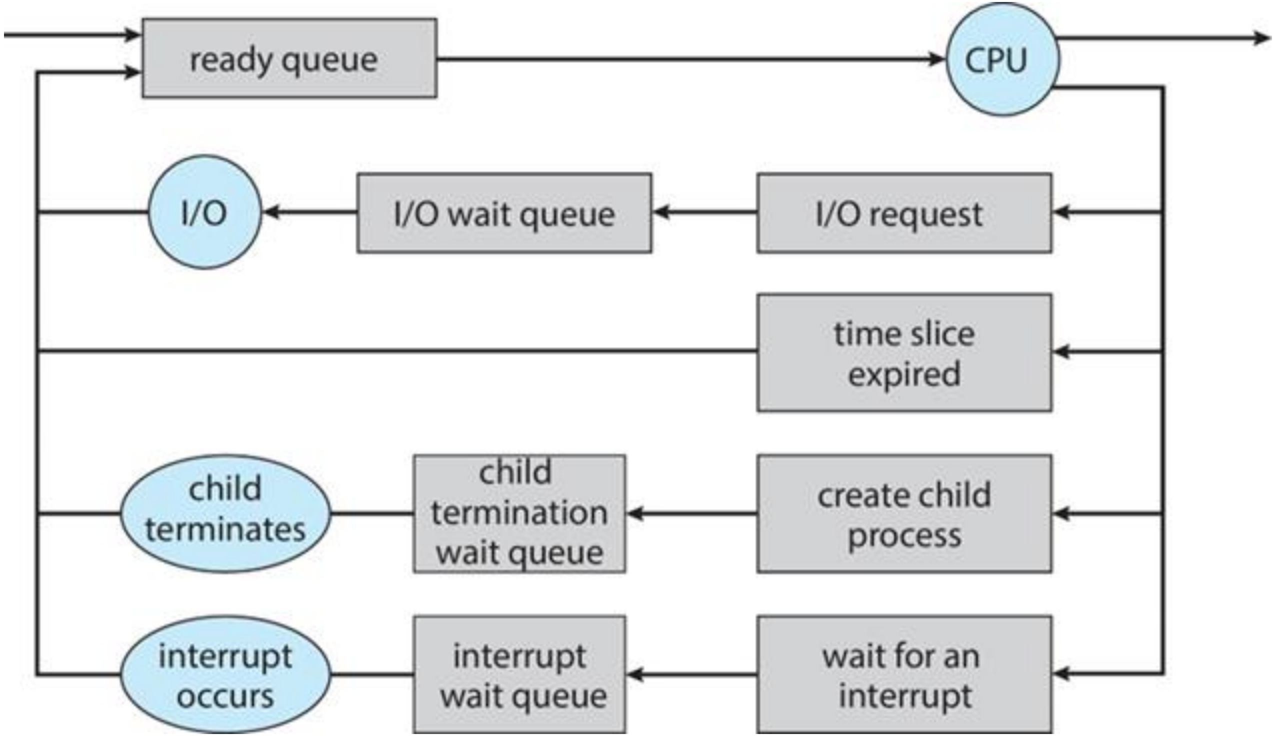
Process Scheduling

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



Representation of Process Scheduling

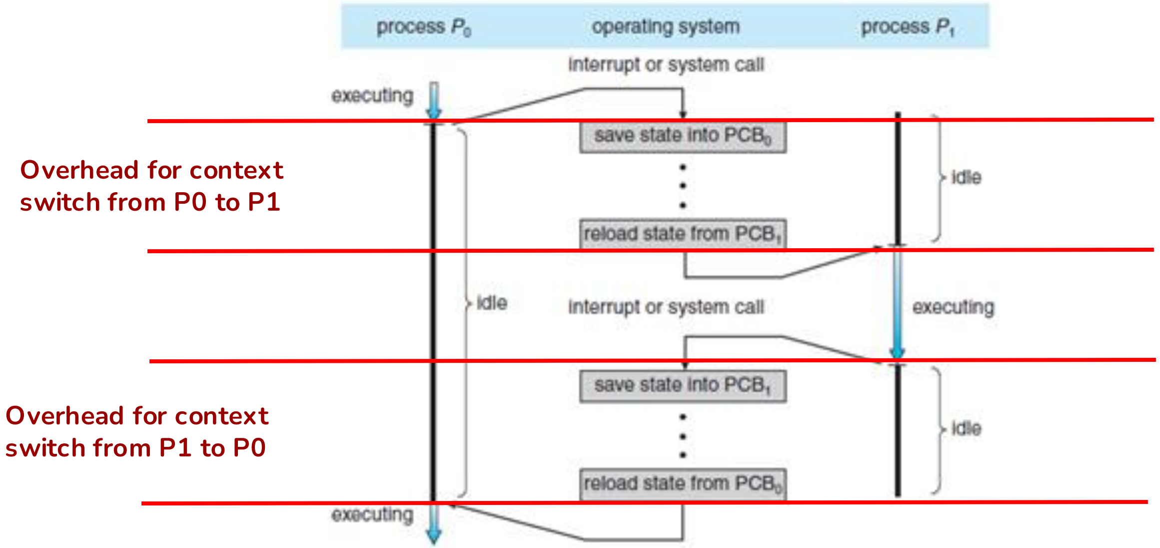


Context Switch

A **context switch** occurs when the CPU switches from one process to another.

Context Switch:

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

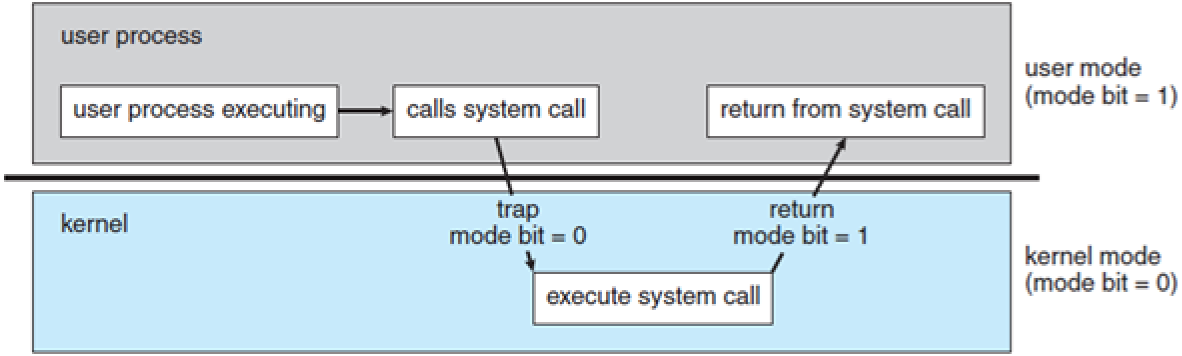


Context Switch

- 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

Dual Mode Operation

- Need to distinguish between the execution of operating-system code and user defined code
- A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1)
- Dual mode of operation provides protection of the operating system from errant users
- This protection is provided by designating some of the machine instructions that may cause harm as privileged instructions that are executed only in kernel mode



Transition from user to kernel mode.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Note: the system-call names used throughout this text are generic

System Call Implementation

- ➔ Typically, a number is associated with each system call
  - ◆ **System-call** interface maintains a table indexed according to these numbers
- ➔ The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- ➔ The caller need know nothing about how the system call is implemented
  - ◆ Just needs to obey API and understand what OS will do as a result call
  - ◆ Most details of OS interface hidden from programmer by API
    - ● Managed by run-time support library (set of functions built into libraries included with compiler)

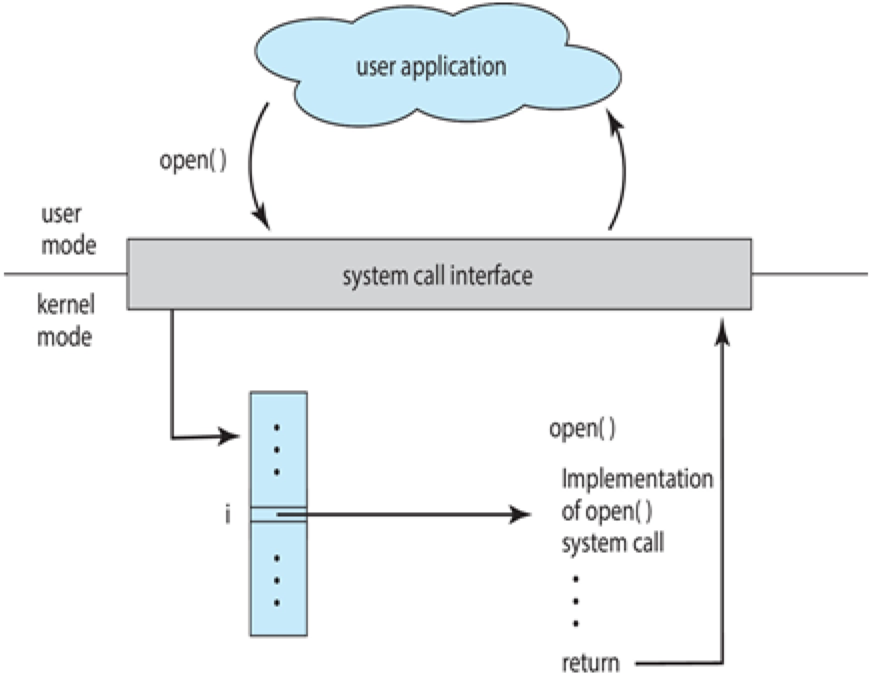


Figure: API – System Call – OS Relationship

Types of System Call

Type	Windows OS	Linux OS
Process Control	**CreateProcess()** **ExitProcess()** **WaitForSingleObject()**	**fork()** **exit()** **wait()**
File Manipulation	**CreateFile()** **ReadFile()** **WriteFile()** **CloseHandle()**	**open()** **read()** **write()** **close()**
Device Manipulation	**SetConsoleMode()** **ReadConsole()** **WriteConsole()**	**ioctl()** **read()** **write()**

Types of System Call

Type	Windows OS	Linux OS
Information Maintenance	**GetCurrentProcessID()** **SetTimer()** **Sleep()**	**getpid()** **alarm()** **sleep()**
Communication	**CreatePipe()** **CreateFileMapping()** **MapViewOfFile()**	**pipe()** **shm_open()** **mmap()**
Protection	**SetFileSecurity()** **InitlializeSecurityDescriptor()** **SetSecurityDescriptorGroup()**	**chmod()** **umask()** **chown()**

Operating Systems

Operations on Process

FBA

Operations on Processes

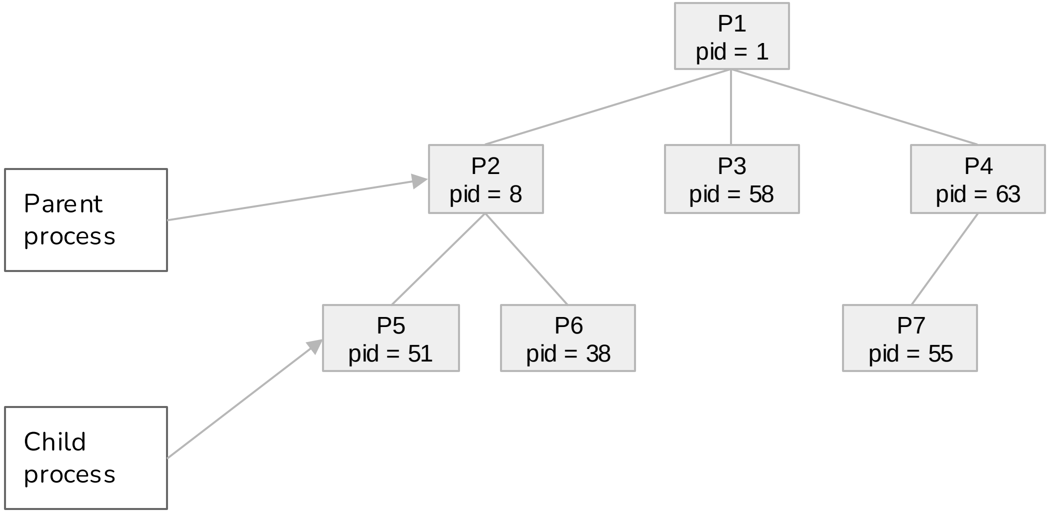
System must provide mechanisms for:

- Process creation
- Process termination

Process Creation

- ➔ **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



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.

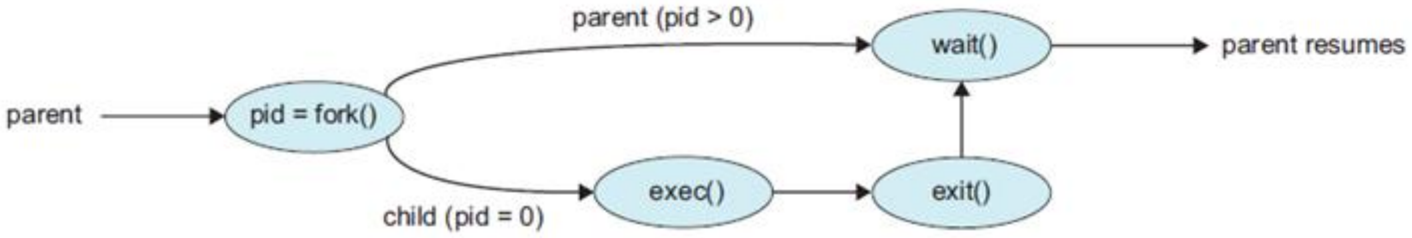


Fig: Process creation using fork() system call

```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.

```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;
}
```

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

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

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

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

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

Output

AAAA

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

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

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

Output

```
AAAAAAAA
```

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

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

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

Output

AAAAAA

```
int main(){
        fork();
         a = fork();
         if(a==0)
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”);
}
```

Output

AAAAAA

```c
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);
        }
}
```

```
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);
        }
}
```

Output

```
value of x is: 0value of x is: 2
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call
  - Returns status data 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

- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the id of the terminated process

pid = wait(&status);

- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait()**, process is an **orphan**

Process Termination

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

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 -

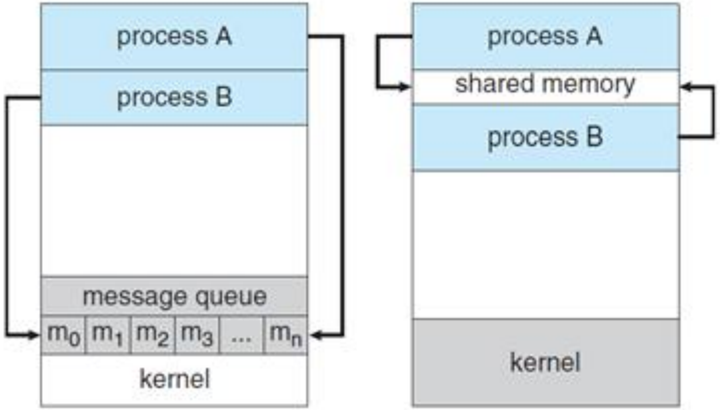
- ➔ Information sharing
- ➔ Computational speedup
- ➔ Modularity
- ➔ Convenience

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

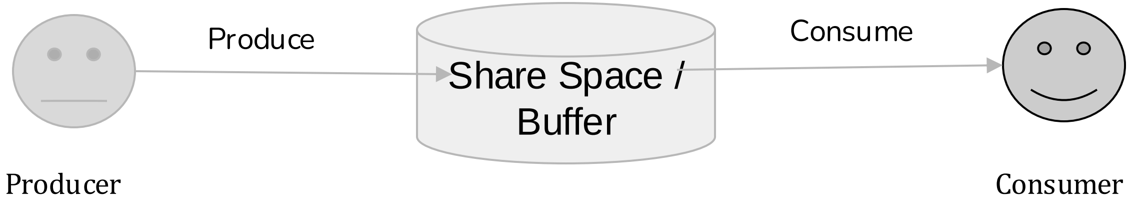


Shared Memory System

(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 */
}
```

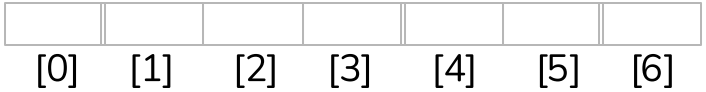
in: next free position in buffer
out: first full position in buffer

Both initialized with 0.

in = 0
out = 0

Here, BUFFER_SIZE = 7

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

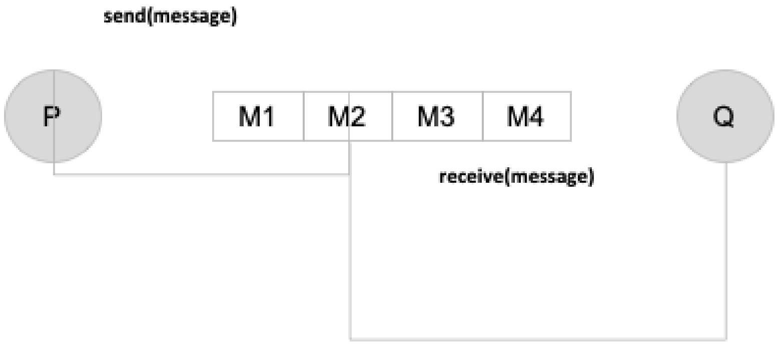


When buffer is not empty,
In = 5, out = 0

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.

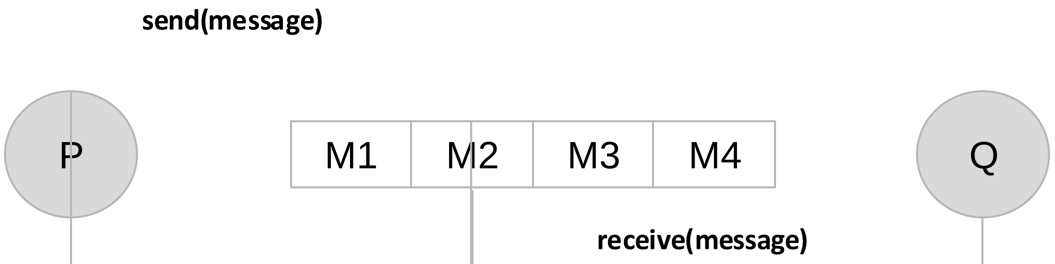


- Useful for exchanging small amount of data
- More suited for distributed systems than shared memory

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