PROCESS MANAGEMENT

- Process Concept
- Process Scheduling
- Operations on Processes

The operating system is responsible for the following activities in connection with Process Management

- Scheduling processes and threads on the CPUs.
- Creating and deleting both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for process synchronization.
- Providing mechanisms for process communication.

INTRODUCTION TO PROCESS

- Early computer systems allowed only one program to run at a time. In contrast, current-day systems allow multiple programs to be loaded into memory concurrently.

- Multiple programming requires firmer control which gave rise to the concept of process which is a unit of work in a modern time-sharing system.

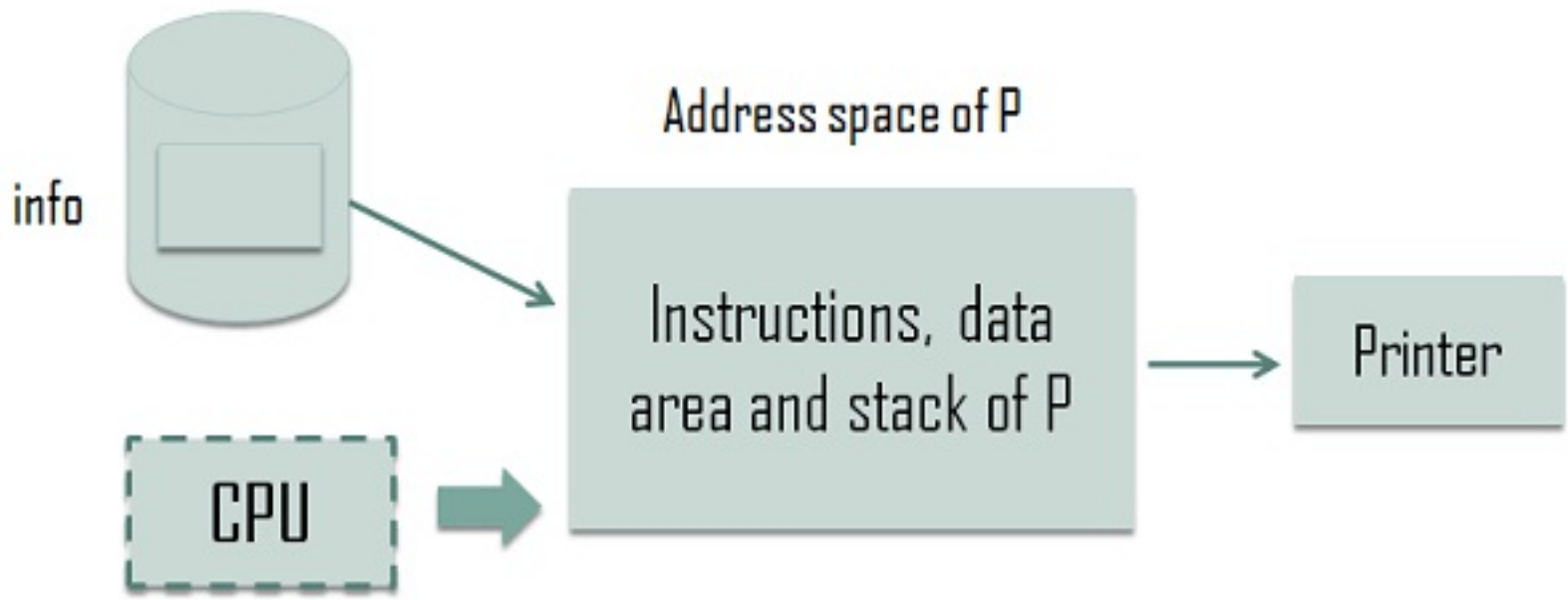
- A Program does nothing unless its instructions are executed by a CPU. A program in execution is called a process. In order to accomplish its task, process needs the computer resources.

- There may exist more than one process in the system which may require the same resource at the same time. Therefore, the operating system has to manage all the processes and the resources in a convenient and efficient way.

Program VS Process

A program is a **passive entity,** for example, a file accommodating a group of instructions to be executed (executable file). It is so called because it doesn't perform any action by itself, it has to be executed to realize the actions specified in it.

The address space of a program is composed of the instruction, data and stack. Assume P is the program we are writing, to realize execution of P, the operating system allocates memory to accommodate P's address space.



PROCESS

- **Process** is an execution of a program. It is considered as an **active entity** and realizes the actions specified in a program.

- Multiple processes can be related to the same program. It handles the operating system activities through **PCB (Process control Block)** which includes program counter, stack, state etc.

- Program counter stores the next sequence of instruction that is to be executed later.

- It needs resources like processing, memory and I/O resources to accomplish management tasks. During the execution of a program, it could engage processor or I/O operation that makes a process different from a program.

- Example; we are writing a C program. While writing and storing a program in a file, it is just a script and does not perform any action, but when it is executed it turns into process hence process is dynamic in nature.

Key Differences Between Program and Process

- A program is a definite group of **ordered operations** that are to be performed. On the other hand, an **instance** of a program being executed is a process.

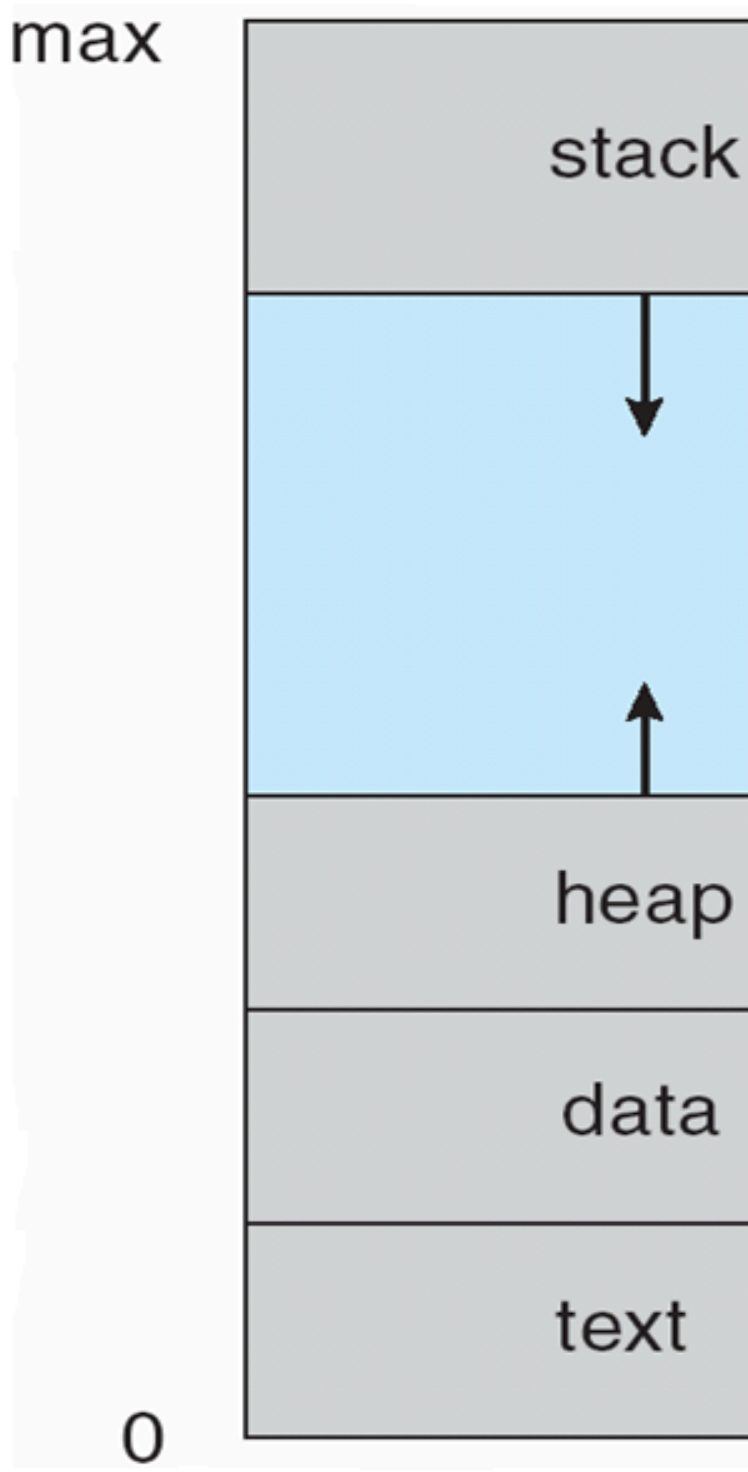
- The nature of the program is passive as it does nothing until it gets executed whereas a process is dynamic or active in nature as it is an instance of executing program and perform the specific action.

- A program has a **longer** lifespan because it is stored in the memory until it is not manually deleted while a process has a shorter and **limited** lifespan because it gets terminated after the completion of the task.

- The resource requirement is much higher in case of a process; it could need processing, memory, I/O resources for the successful execution. In contrast, a program just requires memory for storage.

PROCESS

- A stack contains temporary data such as function calls, local variables etc.
- Heap: Dynamically allocated memory during run time.
- Data section contains global variables.
- The program code, also called **text section**
- Current activity including **program counter**,

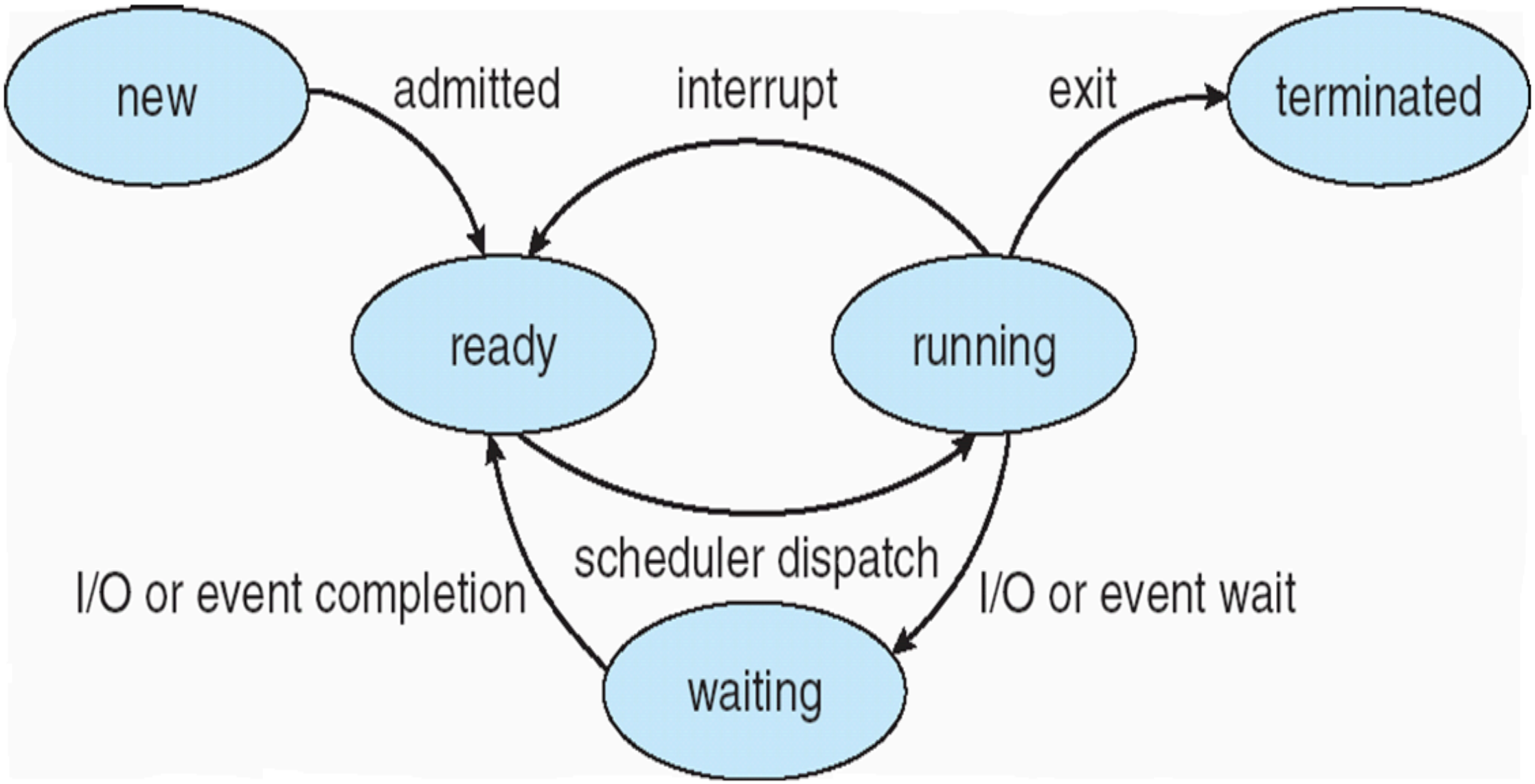


PROCESS STATE

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- New- The process is being created.
- Running- Instructions are being executed.
- Waiting- Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- Ready- The process is waiting to be assigned to a processor.
- Terminated- The process has finished execution.

PROCESS STATE DIAGRAM



PROCESS CONTROL BLOCK

- The Attributes of the process are used by the Operating System to create the process control block (PCB) for each of them. This is also called context of the process. Attributes which are stored in the PCB are described below.

- A Process Control Block is a data structure maintained by the Operating System for every process. 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

- **Process State**: The current state of the process i.e., whether it is ready, running, waiting, or whatever.

- **Program Counter-** Program Counter is a pointer to the address of the next instruction to be executed for this process.

- **CPU registers-** Various CPU registers where process need to be stored for execution for running state.

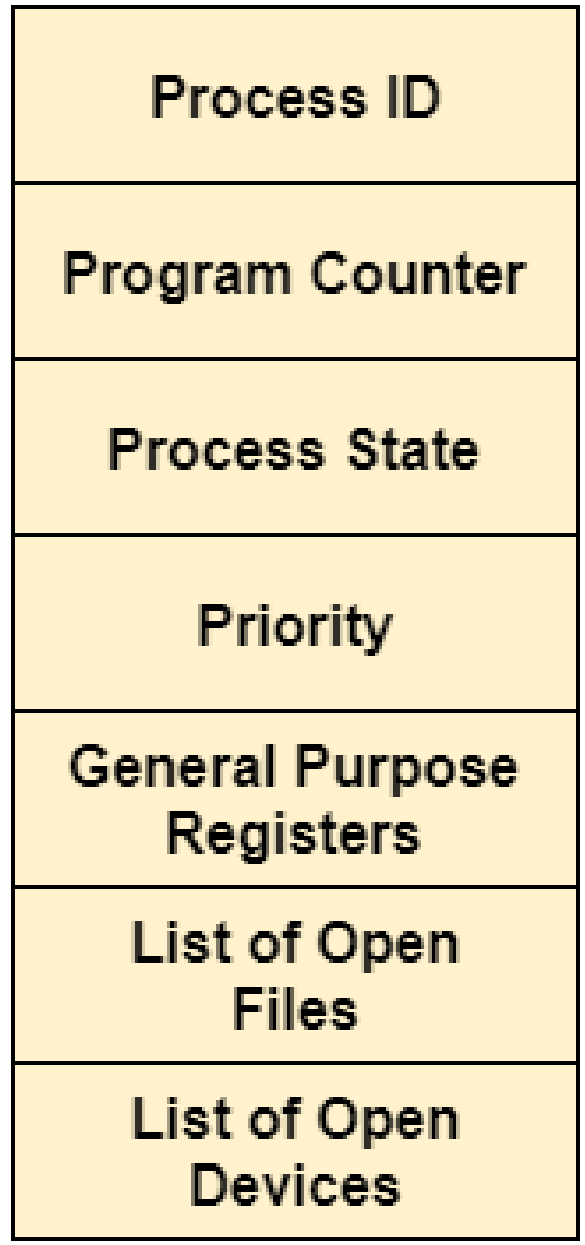
- **CPU Scheduling Information-** Process priority and other scheduling information which is required to schedule the process.

- **Memory management information-** This includes the information of page table,memory limits, Segment table depending on memory used by the operating system.

- **Accounting information-** This includes the amount of CPU used for process execution, time limits, execution ID etc.

- **IO status information**-This includes a list of I/O devices allocated to the process.

PROCESS CONTROL BLOCK



Process Attributes

Process Scheduling

- Overview
- Scheduling Queues
- Schedulers
- Context Switch

Overview:

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing
- Maximize CPU use, quickly switch processes onto CPU for time sharing

Why Scheduling?

- A process is like a job in computer system that can be executed.
- Some processes are input/output (I/O) like graphics display process, others are CPU-focused and can be transparent to users.
- If your computer freezes, sometimes the underlying issue could be that a system process is trying to acquire CPU resources, but those resources are already occupied by other processes.
- Through process scheduling, the operating system tries to avoid these kind of deadlocks and lockups.

Process Scheduling Queues

- The OS maintains all PCBs in Process Scheduling Queues.
- The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue.
- When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

Process Scheduling Queues

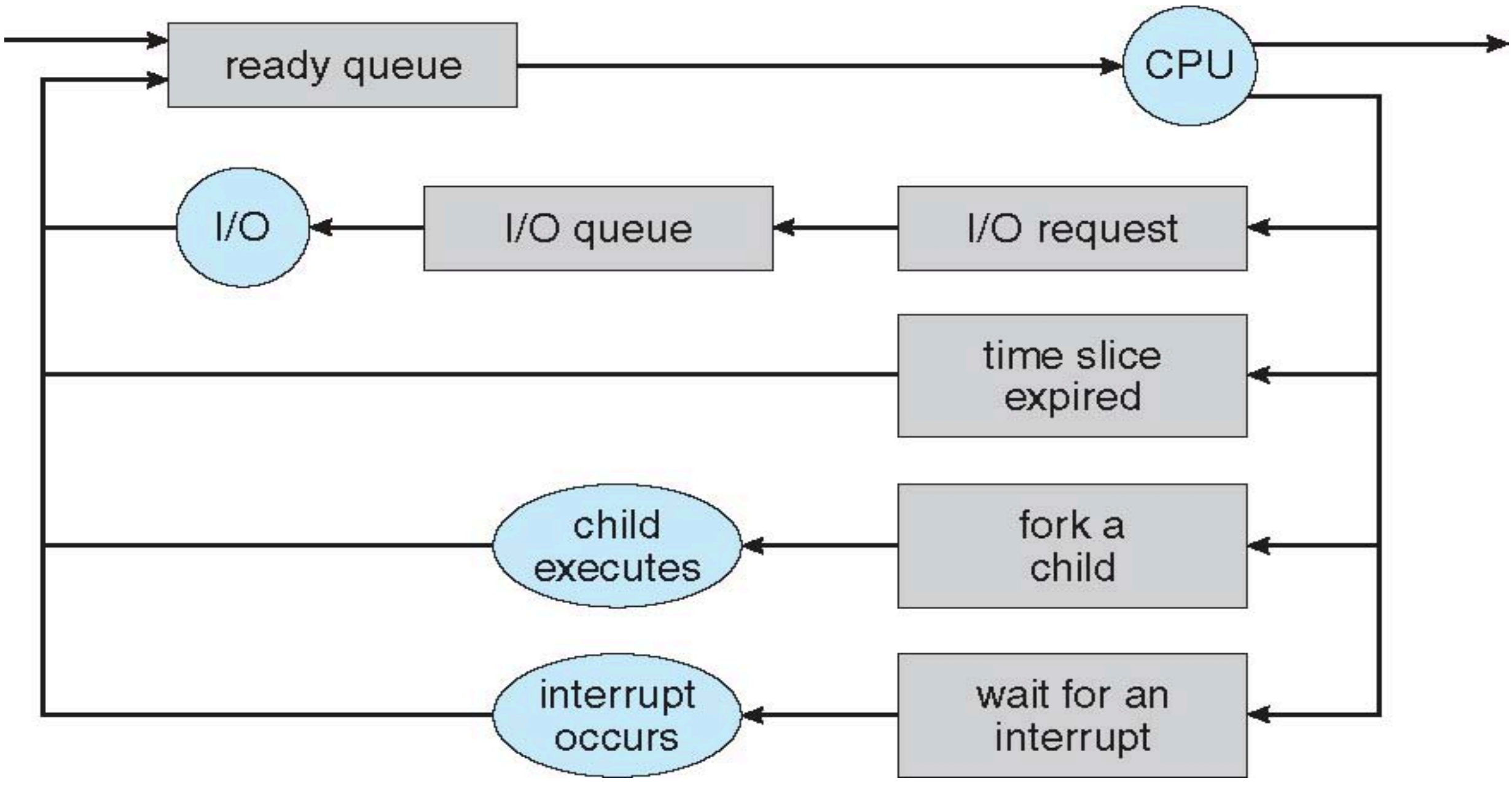
The Operating System maintains the following important process scheduling queues −

➢ Job queue − This queue keeps all the processes in the system.

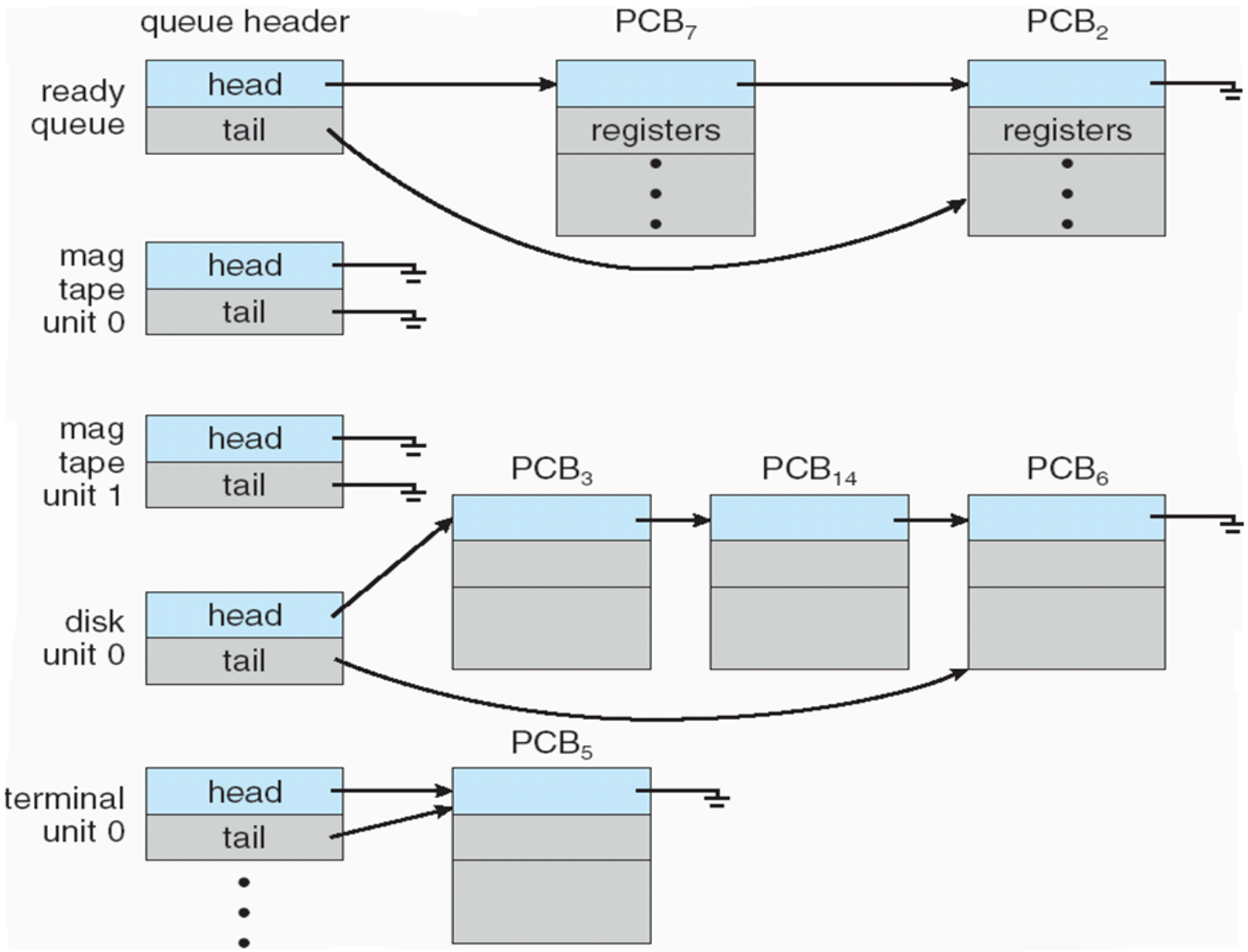
➢ Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

➢ Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue.

Process Scheduling Queues



Ready Queue And Various I/O Device Queues



SCHEDULERS

Schedulers are special system software which handle process scheduling in various ways.
Their main task is to select the jobs to be submitted into the system and to decide which process to run.

Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

- **Degree of Multiprogramming** - The number of processes in Memory.

- **I/O Bound Process** - An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.

- **CPU- Bound Process** - A CPU-bound process , generates I/O requests infrequently, using more of its time doing computations.

- **Swapping** - Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes.

Long-Term Scheduler

- It is also called a job scheduler. A long-term scheduler determines which programs are admitted to the system for processing.

- It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

- On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short-Term Scheduler

- It is also called as CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria.

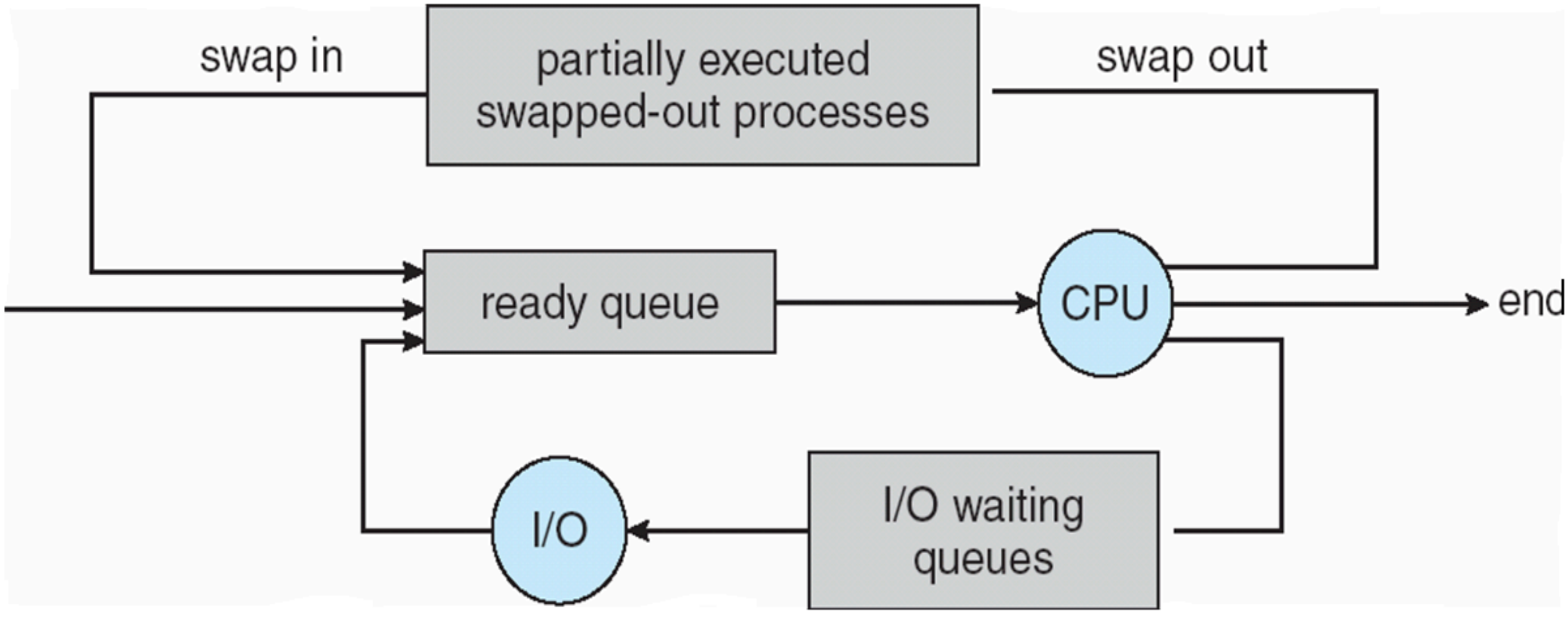
- It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium-term Scheduler

- Medium-term scheduling is a part of swapping. It removes the processes from the memory. It reduces the degree of multiprogramming.
- The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Addition of Medium Term Scheduling



Comparison between Long, short and medium level scheduler

Sr. No.	Long Term	Short Term	Medium Term
1	It is job scheduler	It is CPU Scheduler	It is swapping
2	Speed is less than short term scheduler	Speed is very fast	Speed is in between both
3	It controls degree of multiprogramming	Less control over degree of multiprogramming	Reduce the degree of multiprogramming.
4	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduler.
5	It select processes from pool and load them into memory for execution.	It select from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6	Process state is (New to Ready)	Process state is (Ready to Running)	-
7	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	-

Context Switch

- CS is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.

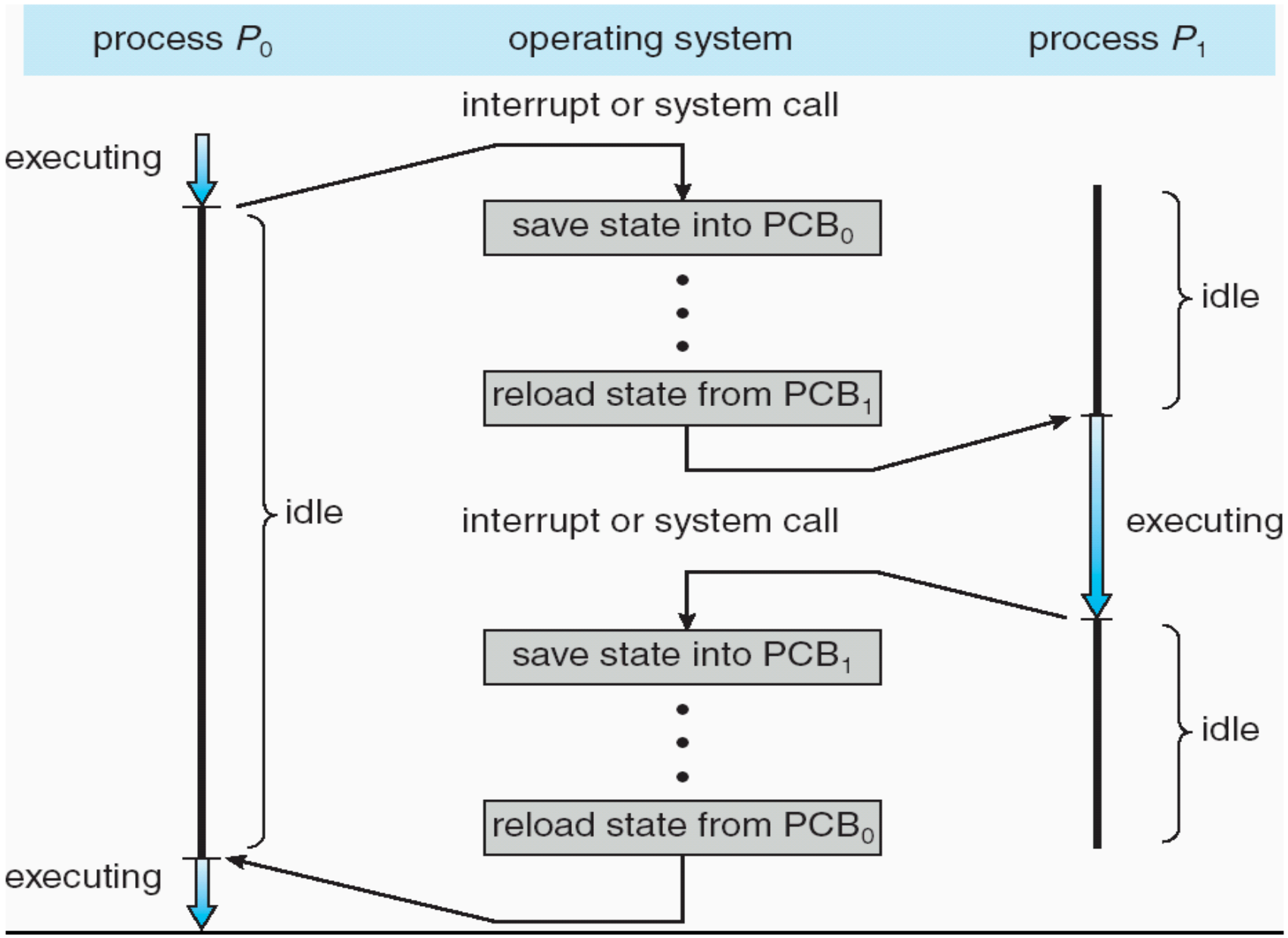
- Using this technique, a context switcher enables multiple processes to share a single CPU.

- CS is an essential part of a multitasking operating system features. When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

- Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

When the process is switched, the following information(PCB) is stored for later use.
● Program Counter ● Scheduling information ● Base and limit register value ● Currently used register ● Changed State ● I/O State information ● Accounting information

Diagram showing CPU switch from process$_0$ to process$_1$

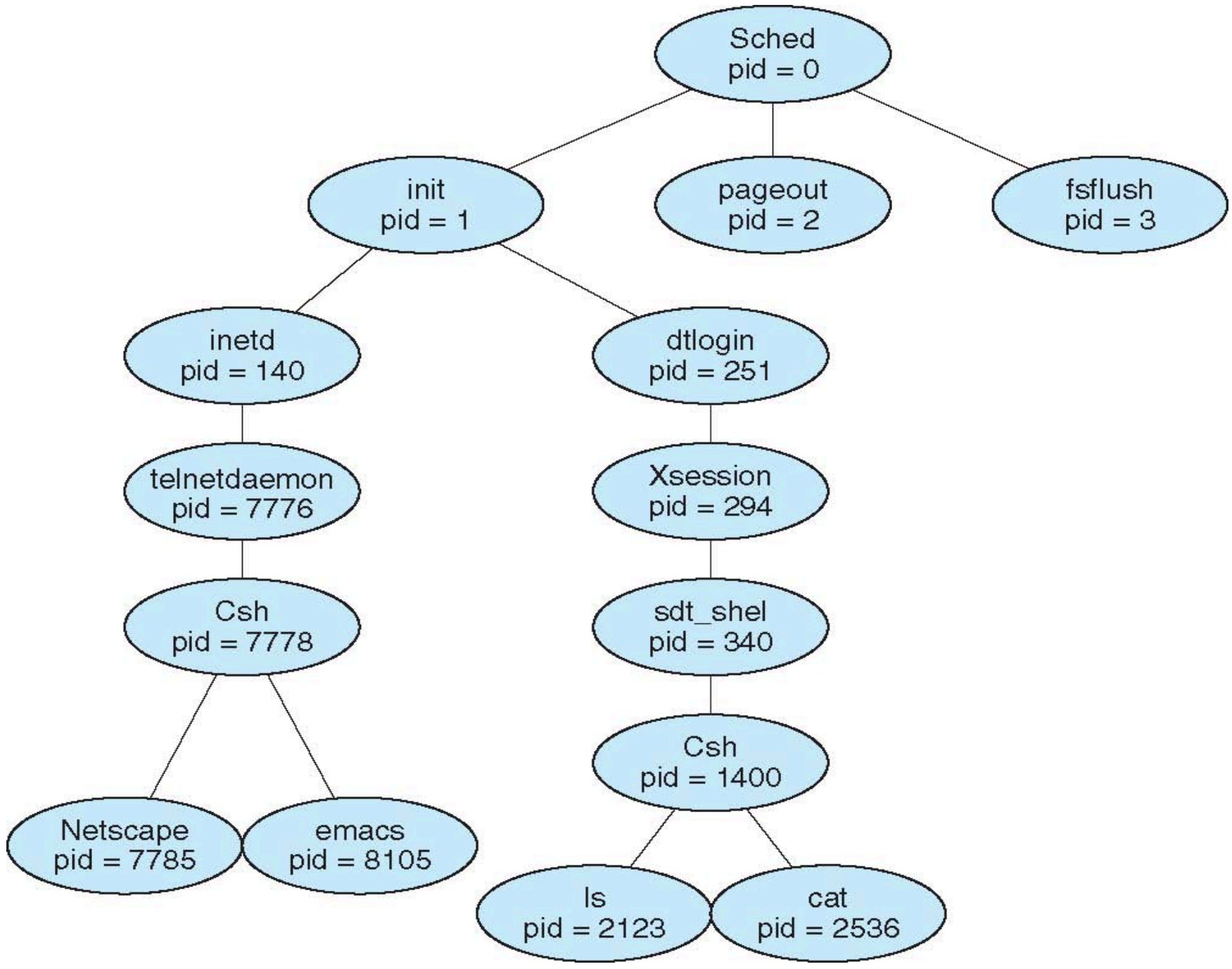


Operations on Process

There are many operations that can be performed on processes.
Some of these are:

- Process creation
- process preemption
- process blocking
- process termination.

A Tree of Processes on Solaris



A tree of processes on a typical Solaris system

- The process at the top is the parent process with pid 0 and it has three children process including- pageout and fsflush.
- The sched process also creates init process which serves as root parent for all user processes.
- Init has two children- inetd and dtlogin.
- inetd is responsible for networking services like telnet and ftp and dtlogin is process representing a user login screen.
- When a user logs in, dtlogin creates an X-windows session which in turn creates the sdt_shel process. Below it a C-shell or csh is created in which the user can invoke various processes using ls and cat commands.
- There is another process csh with pid 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser(pid of 7785) and the emacs editor(pid of 8105).

Process creation

- A process may create several new processes, via a create-process system during the course of execution.

- The creating process is called a parent process and the new process is called a child process.

- Each of these new processes may in turn create other processes thus forming a tree of processes.

- Most operating systems identify processes according to a unique process identifier (or pid) which is usually an integer.

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

- Parent and children share all resources
- Children share subset of parent’s resources
- Parent and child share no resources

Execution

- Parent and children execute concurrently
- Parent waits until children terminate

Process creation

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).

FORK()

- **It takes no parameters and returns an integer value. Below are different values returned by fork().**
- **Fork() returns process Id of the child process to the parent process and value zero to the child.**
- **After a new child process is created, both processes will execute the next instruction following the fork() system call.**

EXAMPLE OF FORK()

```
//Program when  child and parent execute same code

#include <iostream>
using namespace std;
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // make two process which run same
   // program after this instruction
   fork();

   cout<<"Hello world!\n";
   return 0;
}
```

Output:

```
Hello world!
Hello world!
```

```
#include <iostream>
#include <unistd.h>
using namespace std;
// 4 ii) same program , different code
int main()
{
cout<<"Same Program Same Code : "<<endl;
pid_t id = fork();
if (id > 0)
{
cout << "i am parent process " << getpid() << endl;
}
else if (id == 0)
{
cout << "i am child process " << getpid() << endl;
}
else
{
cout << " no child process created " << getpid() << endl;
}
return 0;
}
```

//Program when child and parent execute different code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{      int p;
       p=fork();
    // child process because return value zero
    if (p== 0)
    {
           printf("Hello from Child!\n");
           printf("I am the child process having
PID %d\n",getpid());
           printf("My Parent PID %d\n",getppid());
    }
    // parent process because return value non-zero.
    else
    {
         printf("Hello from Parent!\n");
        printf("I am the Parent process having
PID %d\n",getpid());
        printf("My child PID %d\n",p);

    }
```

```
int main()
{
    forkexample();
    return 0;
}
```

```
//Q 4 c parent and child execute before terminating, the parent waits for the
child to finish its task.
#include<iostream>
#include<unistd.h>
#include<sys/wait.h>
using namespace std;
int main()
{
	int a=fork();
	if(a<0){cout<<"cchild not created";}
	else if(a==0){
		cout<<"Child process!!!!!";
		sleep(10);
	}
	else{
		wait(NULL);
		cout<<"Parent process!!!!";
	}
	return 0;
}

~
```

//Program when parent will wait until child finishes its execution
//Program when child and parent execute different code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
void forkexample()
{      int p;
       p=fork();
    // child process because return value zero
    if (p== 0)
    {
          printf("Hello from Child!\n");
          printf("I am the child process having
PID %d\n",getpid());
          printf("My Parent PID %d\n",getppid());
    }
    // parent process because return value non-zero.
    else
    {     wait(NULL);
         printf("Hello from Parent!\n");
        printf("I am the Parent process having
PID %d\n",getpid());
```

```
int main()
{
    forkexample();
    return 0;
}
```

```
#include<iostream>
#include<unistd.h>
#include<sys/wait.h>
using namespace std;
int main()
{
pid_t a=fork();
if(a<0){
cout<<"Child not created!!"<<endl;
}
else if(a==0){
cout<<"Child process."<<endl;
sleep(30); //put the compiler to sleep
}
else{
wait(NULL); //wait for child to execute first
cout<<"Parent process."<<endl;
}
return 0;
}
```

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
    {   printf("Hello from Child!\n");
        execlp("/bin/ls"," ",NULL);
        }

    // parent process because return value non-zero.
    else
        {wait(NULL);
        printf("Hello from Parent!\n");
        }
}
int main()
{
    forkexample();
    return 0;
}
```

EXAMPLE OF FORK()

Q2. Calculate number of times hello is printed:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();

 printf("hello\n");
    return 0;
}
```

hello hello hello hello hello hello hello hello

The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, 2^3 = 8

EXAMPLE OF FORK()

Let us put some label names for the three lines:

```
fork ();     // Line 1
fork ();     // Line 2
 fork ();    // Line 3

          L1                  // There will be 1 child process
        /     \                // created by line 1.
      L2       L2             // There will be 2 child processes
     /   \    /   \           // created by line 2
   L3      L3  L3   L3         // There will be 4 child processes
                               // created by line 3
```

So there are total eight processes (new child processes and one original process).

EXAMPLE OF FORK()

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:
The main process: P0
Processes created by the 1st fork: C1
Processes created by the 2nd fork: C2, C3
Processes created by the 3rd fork: C4, C5, C6, C7

```
      C1      // There will be 1 child process created by first fork
    /    \
   C2     C3   // There will be 2 child processes created by second fork
  / \     / \
 C4  C5  C6  P7  // There will be 4 child processes created by third fork
/\  /\/\  /\
 .............  // and so on
```

EXAMPLE OF FORK()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

1.Hello from Child!
 Hello from Parent! (or)
2. Hello from Parent!
Hello from Child!

Important: Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different.
See next example:

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Two cases possible:
Parent has x = 0
Child has x = 2 (or)

Child has x = 2
Parent has x = 0

Here, global variable change in one process does not affected two other processes because data/state of two processes are different.

And also parent and child run simultaneously so two outputs are possible.

EXERCISE 1:
A process executes the following code
for (i = 0; i < n; i++)
fork();
The total number of child processes created is: (GATE-CS-2008)
(A) n
(B) 2^n – 1
(C) 2^n
(D) 2^(n+1) – 1;

```
          F0      // There will be 1 child process created by first fork
        /    \
      F1      F1    // There will be 2 child processes created by second fork
     /  \    /  \
   F2   F2  F2   F2  // There will be 4 child processes created by third
fork
/ \   / \ / \  / \
 ..............   // and so on
```

If we sum all levels of above tree for i = 0 to n-1, we get 2^n - 1. So there will be 2^n – 1 child processes. On the other hand, the total number of **process** created are (number of child processes)+1.

Exercise: The total number of child processes created ----------?
(GATE-CS-2019)

```
int main(){
int n=10;
for (i = 0; i < n; i++){
        if(i%2==0)
        fork();
        }
return 0;
}
```

fork() vs exec()

- The fork system call creates a new process. The new process created by fork() is a copy of the current process except for the returned value.

- The exec() system call replaces the current process with a new program.

- The exec family of functions replaces the current running process with a new process. It comes under the header file **unistd.h.** There are many members in the exec family which are shown below with examples.

- The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.

- It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.

Process Termination

Process executes last statement and asks the operating system to delete it (exit)

Output data from child to parent (via **wait**)

Process' resources are deallocated by operating system

Parent may terminate execution of children processes (abort)

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to a children is no longer required.
- **The parent is exiting and the operating system does not allow a child to continue if its parent terminates. In this case, all its children are also terminated. This phenomenon is known as cascading termination.**
- All the resources of the process- including physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.