

UNIT-II

PROCESS MANAGEMENT

PROCESS CONCEPT

The process concept includes the following:

1. Process
2. Process state
3. Process Control Block
4. Threads

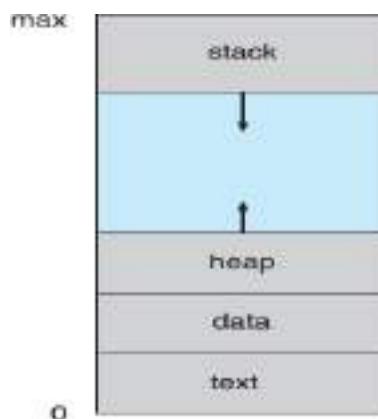
Process

A process can be thought of as a program in execution (or) A process is the unit of work in a modern time-sharing system.

A process will need certain resources such as CPU time, memory, files and I/O devices to accomplish its task.

These resources are allocated to the process either when it is created or while it is executing.

The below figure shows the structure of process in memory:



The process contains several sections: Text, Data, Heap and Stack.

- **The Text Section** contains the program code. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- **Process stack** contains temporary data such as function parameters, return addresses and local variables.
- **Data section** contains global variables.
- **Heap** is memory that is dynamically allocated during process run time.

Difference between Program and Process:

- A program is a **passive** entity, such as a file containing a list of instructions stored on disk often called an **executable file**.
- A process is an **active** entity with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line as in prog.exe or a.out.

Although two processes may be associated with the same program, they are considered as two separate execution sequences. For instance, several users may be running different copies of the mail program or the same user may invoke many copies of the web browser program. Each

of these is considered as a separate process.

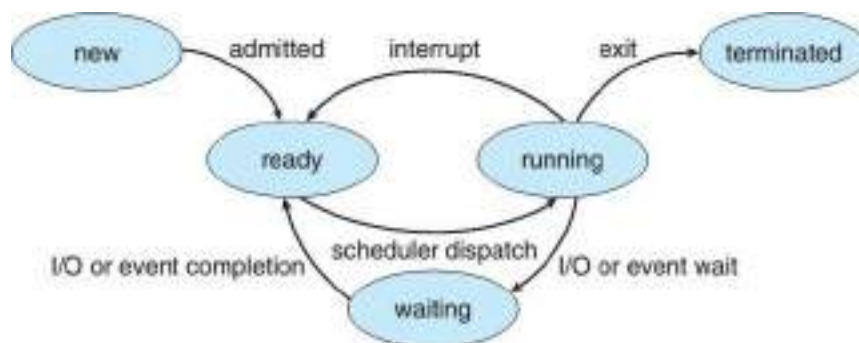
Process State

As a process executes, it changes **state**. The process state defines the current activity of that process.

A process may be in one of the following states:

- **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 such as an I/O completion or reception of a signal.
- **Terminated:** The process has finished execution.

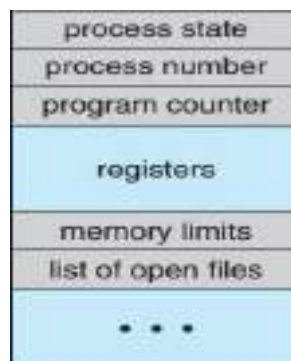
Note: Only one process can be *running* on any processor at any instant of time.



Process Control Block

Each process is represented in the operating system by a **Process Control Block (PCB)**. It is also called a **Task Control Block**.

PCB serves as the repository for any information that may vary from process to process.



The PCB contains information related to process such as:

- **Process state:** The state may be new, ready, running, waiting and terminated.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers etc. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues and any other scheduling parameters.
- **Memory-management information:** This information includes the base and limit registers

values, the page tables or the segment tables depending on the memory system used by the operating system.

- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files and so on.

Threads

In a single processor system a process is a program that performs a single **thread** of execution.

- **Example:** When a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at a time.
- The user cannot simultaneously type in characters and run the spell checker within the same process.

A multicore system or multi-processor system allows a process to run multiple threads of execution in parallel.

On a system that supports threads, the PCB is expanded to include information for each thread.

Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the **Process Scheduler** selects an available process for program execution on the CPU.

Process scheduling involves three things:

1. Scheduling Queues
2. Schedulers
3. Context Switch

Scheduling Queues

There are several queues implemented in the operating system such as Job Queue, Ready Queue, Device Queue.

- **Job Queue:** It consists of all processes in the system. As processes enter the system, they are put into a **job queue**.
- **Ready Queue:** The processes that are residing in main memory and they are ready and waiting to execute are kept on a list called the **Ready Queue**. Ready queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue. waiting for a particular I/O device.

- **Device Queue:** Each device has its own device queue. It contains the list of processes. Consider the above Queuing Diagram:

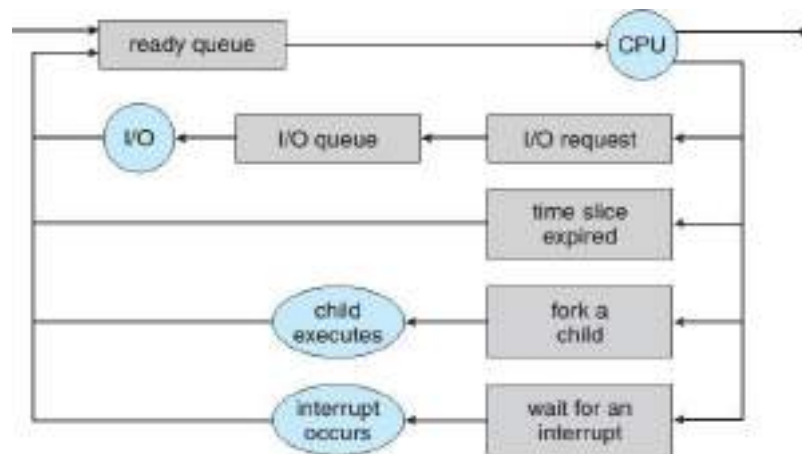


Figure 3.6 Queuing-diagram representation of process scheduling.

- Two types of queues are present: the **Ready Queue** and a set of **Device Queues**. CPU and I/O are the resources that serve the queues.
- A new process is initially put in the ready queue. It waits there until it is selected for execution or **dispatched**.

Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt and be put back in the ready queue.

Schedulers

A process migrates among the various scheduling queues throughout its lifetime. For scheduling purposes, the operating system must select processes from these queues. The selection process is carried out by the **Scheduler**.

There are three types of Schedulers are used:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Long Term Scheduler (New to ready state)

- Initially processes are spooled to a mass-storage device (i.e Hard disk), where they are kept for later execution.
- Long-term scheduler or job scheduler selects processes from this pool and loads them into main memory for execution. (i.e. from Hard disk to Main memory).
- The long-term scheduler executes much less frequently, there may be minutes of time between creation of one new process to another process.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory).

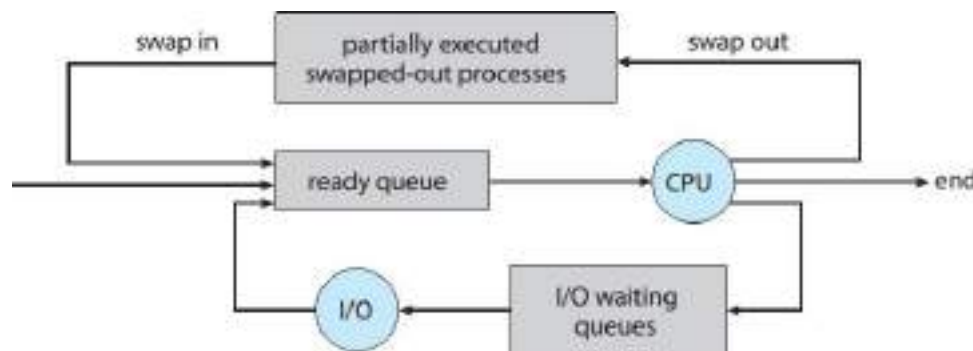
Short Term Scheduler (Ready to Running)

- Short-term scheduler or CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them. (i.e. a process that resides in main memory will be taken by CPU for execution).
- The short-term scheduler must select a new process for the CPU frequently.
- The short term scheduler must be very fast because of the short time between executions of processes.

Medium Term Scheduler

Medium Term Scheduler does two tasks:

1. **Swapping:** Medium-term scheduler removes a process from main memory and stores it into the secondary storage. After some time, the process can be reintroduced into main memory and its execution can be continued where it left off. This procedure is called Swapping.
2. Medium Term Scheduler moves a process from CPU to I/O waiting queue and I/O queue to the ready queue.



The processes can be described as two types:

1. I/O bound process is one that spends more of its time doing I/O than it spends doing computations.
2. The CPU Bound process uses more of its time doing computations and generates I/O requests infrequently.

The long-term scheduler selects a good **process mix** of I/O-bound and CPU-bound processes.

- If all processes are I/O bound, the ready queue will almost always be empty and the CPU will remain idle for a long time because I/O device processing takes a lot of time.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty. I/O devices will be idle and the CPU is busy for most of the time.
- Thus if the system maintains the combination of CPU bound and I/O bound processes then the system performance will be increased.

Note: Time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

Context Switching

- Switching the CPU from one process to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **Context Switch**.
- The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information.

- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching. Context switch time may be in a few milliseconds.

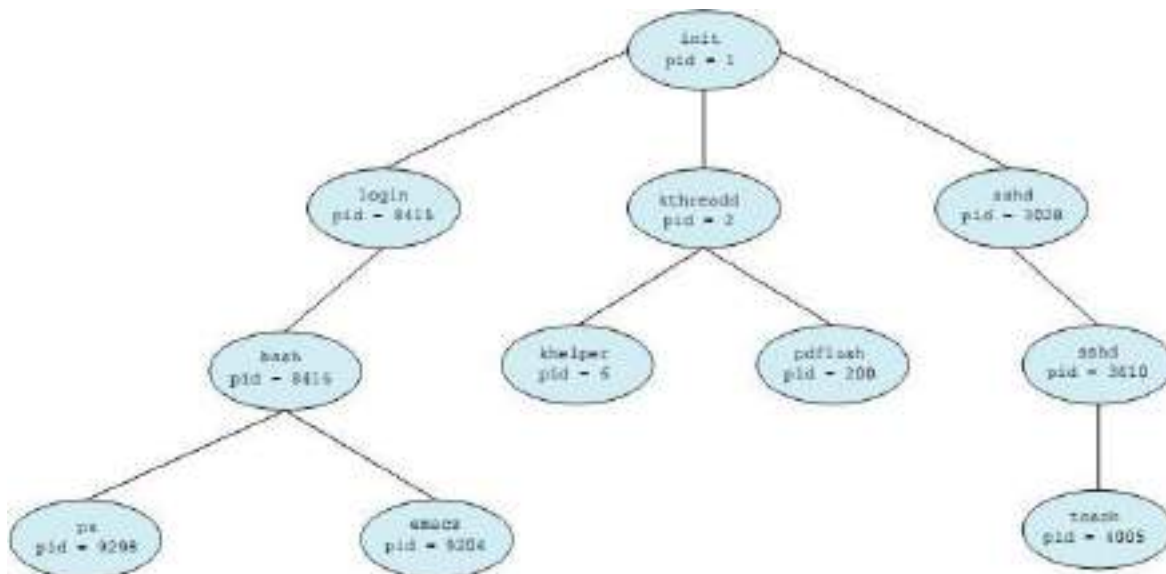
Operations on Processes

1. Process Creation
2. Process Termination

Process Creation

- During the execution of a process in its lifetime, a process may create several new processes.
- The creating process is called a parent process and the new processes are called children process.
- Each of these new processes may create other processes forming a **tree** of processes.
- Operating system identifies processes according to the process **identifier (pid)**.
- Pid provides an unique integer number for each process in the system.
- Pid can be used as an index to access various attributes of a process within the kernel.

The below figure shows the process tree for the Linux OS that shows the name of each process and its pid. In Linux the process is called a task.



- The init process always has a pid of 1. The init process serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server etc.
- kthreadd and sshd are child processes of init.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel.
- The sshd process is responsible for managing clients that connect to the system by using a secure shell (ssh).

ps command is used to obtain a list of processes:

ps -el

The command will list complete information for all processes currently active in the system.

- When a process creates a child process, that child process will need certain resources such as CPU time, memory, files, I/O devices to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children or it may be able to share some resources such as memory or files among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

When a process creates a new process there exist two possibilities for execution:

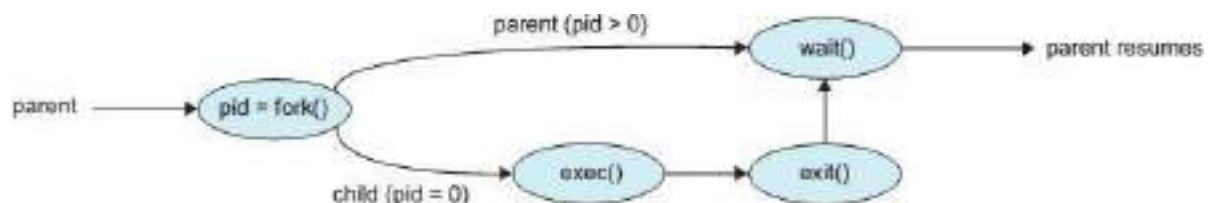
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (i.e) it has the same program and data as the parent.
2. The child process has a new program loaded into it.

Process System calls in Unix/ Linux: **fork()**, **exec()**, **wait()**, **exit()**

- **fork()**: In UNIX OS a new process is created by the **fork()** system call.
- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both the parent and the child processes continue execution at the instruction after the **fork()**.
- For the new child process (i.e. Child Process) the return code for the **fork()** is zero.
- The nonzero process identifier of the child is returned to the parent.
- **exec()**: After a **fork()** system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.
- The **exec()** system call loads a binary file into memory and starts its execution.
- In this way, the two processes are able to communicate and then go their separate ways.
- **wait()**: The parent can create more children or if the parent has nothing else to do while the child process is running then the parent process can issue a **wait()** system call to move itself out of the Ready Queue until the child process terminates.
- The call to **exec()** overlays the process's address space with a new program or the call to **exec()** does not return control unless an error occurs.



Program for Creating a separate process using the UNIX **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;
}

```

The above C program shows the UNIX system calls fork, exec, wait. Two different processes are running copies of the same program.

- The only difference is that the value of pid for the child process is zero, while the value of pid for the parent is an integer value greater than zero (i.e. the actual pid of the child process).
- The child process inherits privileges and scheduling attributes from the parent as well as certain resources such as open files.
- The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes by either implicitly or explicitly invoking exit(), the parent process resumes from the call to wait(), where it completes using the exit() system call.

Process Termination: exit()

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit**() system call.
- The process may return a status value to its parent process via the wait() system call.
- All the resources of the process including physical and virtual memory, open files and I/O buffers are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons such as:

1. The child has exceeded its usage of some of the resources that it has been allocated.
2. The task assigned to the child is no longer required.
3. The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

Cascading Termination

If a parent process terminates either normally or abnormally then all its children must also be terminated is referred as Cascading Termination. It is normally initiated by operating system.

In Linux and UNIX systems, a process can be terminate by using the `exit()` system call providing an exit status as a parameter:

```
/* exit with status 1 */
```

```
exit(1);
```

Under normal termination, `exit()` may be called either directly (i.e. `exit(1)`) or indirectly (i.e. by a return statement in `main()`).

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid;
```

```
int status;
```

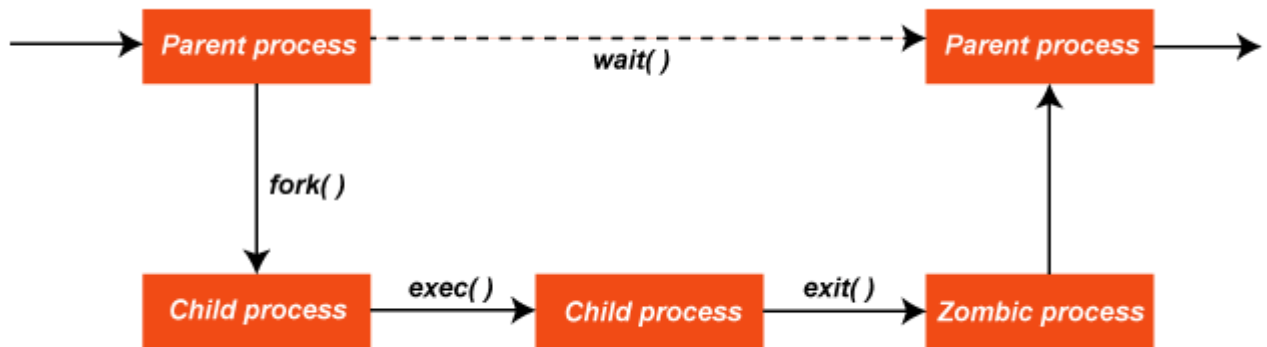
```
pid = wait(&status);
```

Zombie process

A ***zombie process*** or ***defunct process*** is a process that has completed execution (via the `exit` system call) but still has an entry in the process table. This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status. Once the exit status is read via the `wait` system call, the zombie's entry is removed from the process table and said to be "***reaped***". A child process always first becomes a zombie before being removed from the resource table.

In most cases, zombies are immediately waited on by their parents and then reaped by the system under normal system operation. Processes that stay zombies for a long time are generally an error and cause a resource leak, but they only occupy the process table entry.

In the term's metaphor, the child process has died but has not yet been reaped. Also, unlike normal processes, the `kill` command does not affect a zombie process.

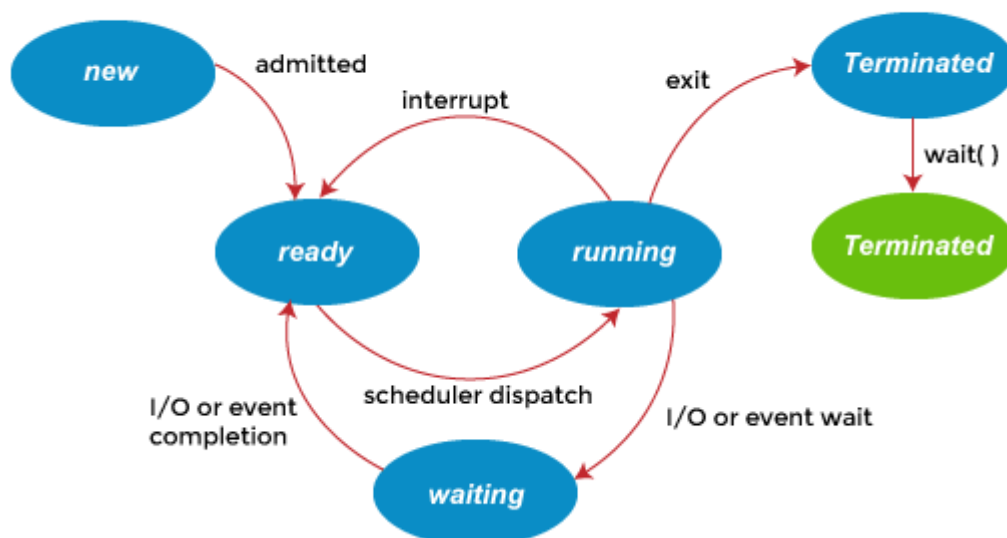


Zombie processes should not be confused with orphan processes. An orphan process is a process that is still executing, but whose parent has died. When the parent dies, the orphaned child process is adopted by init (process ID 1). When orphan processes die, they do not remain as zombie processes; instead, they are waited on by init. The result is that a process that is both a zombie and an orphan will be reaped automatically.

How Does the Zombie Process Work?

In an operating system, a zombie process works in the following way: When a process ends via exit, all of the memory and resources associated with it are deallocated so other processes can use them.

Zombie Processes



- However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, where the zombie is removed. The wait call may be executed in sequential code, but it is commonly executed in a handler for the **SIGCHLD** signal, which the parent receives whenever a child has died.
- After the zombie is removed, its process identifier(PID) and entry in the process table can then be reused. However, if a parent fails to call wait, the zombie will be left in the process table, causing a resource leak. In some situations, this may be desirable, and the parent process wishes to continue holding this resource. For example, if the parent creates another child process, it will not be allocated the same PID.

- The following special case applies to modern UNIX-like systems. If the parent explicitly ignore **SIGCHLD** by setting its handler to **SIG_IGN** (rather than simply ignoring the signal by default) or has the **SA_NOCLDWAIT** flag set, all child exit status information will be discarded, and no zombie processes will be left.
- Zombies can be identified in the output from the UNIX "ps" command by the presence of a "Z" in the "STAT" column. Zombies that exist for more than a short time typically indicate a bug in the parent program or just an uncommon decision to not reap children.
- If the parent program is no longer running, zombie processes typically indicate a bug in the operating system. As with other resource leaks, the presence of a few zombies is not worrying in it but may indicate a problem that would grow serious under heavier loads. Since there is no memory allocated to zombie processes, the only system memory usage is for the process table entry itself. The primary concern with many zombies is not running out of memory but rather out of process table entries and concretely process ID numbers.
- USING THE KILL COMMAND, the **SIGCHLD** signal can be sent to the parent manually to remove zombies from a system. If the parent process still refuses to reap the zombie, and if it would be fine to terminate the parent process, the next step can be to remove the parent process. When a process loses its parent, init becomes its new parent. Init periodically executes the wait system call to reap any zombies with init as a parent.

```
// A C program to demonstrate the Zombie Process.
// Child becomes Zombie as parent is sleeping
// when the child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

Orphan Processes

If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans** are called Orphan processes.

- Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.
- The init process periodically invokes wait(), thereby allowing the exit status of any

orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

// A C program to demonstrate Orphan Process.

// Parent process finishes execution while the

// child process is running. The child process

// becomes an orphan.

```
#include<stdio.h>
```

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

```
#include <unistd.h>
```

```
int main()
```

```
{
```

```
    // Create a child process
```

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

```
    if (pid > 0)
```

```
        printf("in parent process");
```

```
    // Note that pid is 0 in child process
```

```
    // and negative if fork() fails
```

```
    else if (pid == 0)
```

```
    {
```

```
        sleep(30);
```

```
        printf("in child process");
```

```
    }
```

```
    return 0;
```

}

Process Data structures

Process Table

§It is maintained by the OS and is stored in RAM, every process currently executing is maintained in the process table.

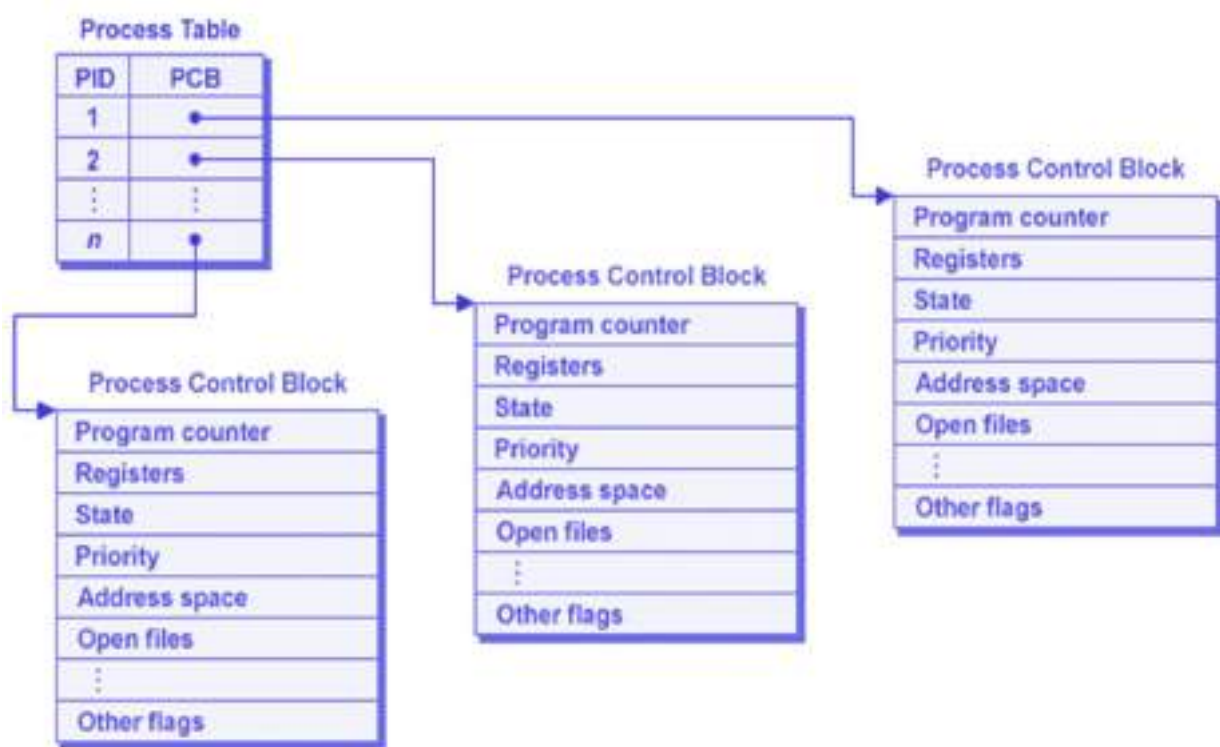
It contains:

§Identifier: Process ID, Parent Process ID and Child Process ID, User ID.

§State: Process state, priority

§Resource: CPU and Memory

Process Control Block (PCB)

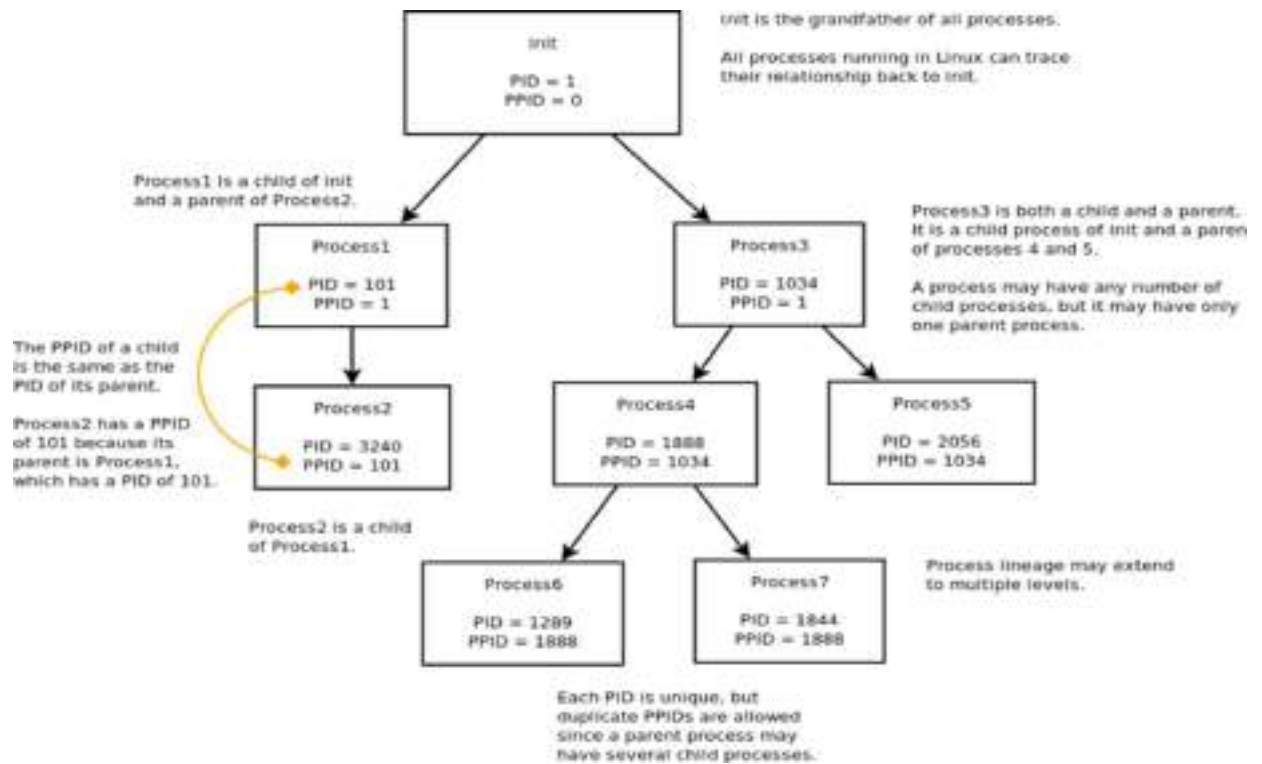


Difference between PID and PPID:

PID: When a process is started, it is given a unique number called process ID (PID).

Init: There is one important process called init, which is the grandfather of all other processes. It has a PID of 1 and the Kernel itself has a PID of 0.

PPID: In addition to a unique process ID, each process is assigned a parent process ID, that tells which process started it.



Threads in OS

A thread is a path of execution within a process. A process can contain multiple threads.

Why Multithreading?

A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

Process vs Thread

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

Advantages of Thread over Process

1. *Responsiveness*: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. *Faster context switch*: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
3. *Effective utilization of a multiprocessor system*: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
4. *Resource sharing*: Resources like code, data, and files can be shared among all threads within

a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. *Communication*: Communication between multiple threads is easier, as the threads share common address space. While in process we have to follow some specific communication techniques for communication between two processes.

6. *Enhanced throughput of the system*: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system

Types of Threads

There are two types of threads.

User Level Thread

Kernel Level Thread

Thread is a single sequence stream within a process. Threads have the same properties as of the process so they are called light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter
2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources etc.

Similarity between Threads and Processes –

- Only one thread or process is active at a time
- Within process both execute sequential
- Both can create children

Differences between Threads and Processes –

- Threads are not independent, processes are.
- Threads are designed to assist each other, processes may or may not do it

Types of Threads: OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

- Advantages of ULT –
 - Can be implemented on an OS that doesn't support multithreading.
 - Simple representation since thread has only program counter, register set, stack space.
 - Simple to create since no intervention of the kernel.
 - Thread switching is fast since no OS calls need to be made.
 - Limitations of ULT –
 - No or less co-ordination among the threads and Kernel.
 - If one thread causes a page fault, the entire process blocks.
2. Kernel Level Thread (KLT) – Kernel knows and manages the threads. Instead of a thread table in each process, the kernel itself has a thread table (a master one) that keeps track of all the threads in the system. In addition, the kernel also maintains the traditional process

table to keep track of the processes. The OS kernel provides system calls to create and manage threads.

- Advantages of KLT –
 - Since the kernel has full knowledge about the threads in the system, the scheduler may decide to give more time to processes having a large number of threads.
 - Good for applications that frequently block.
- Limitations of KLT –
 - Slow and inefficient.
 - It requires a thread control block so it is an overhead.

Threading issues in OS



1. The fork() and exec() System Calls

The fork() and exec() are the system calls. The fork() call creates a duplicate process of the process that invokes fork(). The new duplicate process is called child process and the process invoking the fork() is called the parent process. Both the parent process and the child process continue their execution from the instruction that is just after the fork().

Let us now discuss the issue with the fork() system call. Consider that a thread of the multithreaded program has invoked the fork(). So, the fork() would create a new duplicate process. Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or the duplicate process will be single-threaded.

2. Thread cancellation

Termination of the thread in the middle of its execution is termed as ‘thread cancellation’. Let us understand this with the help of an example. Consider that there is a multithreaded program which has let its multiple threads search through a database for some information. However, if one of the threads returns with the desired result the remaining threads will be canceled.

Now a thread which we want to cancel is termed as target thread.

Thread cancellation

Thread cancellation can be performed in two ways:

Asynchronous Cancellation: In asynchronous cancellation, a thread is employed to terminate the target thread instantly.

Deferred Cancellation: In deferred cancellation, the target thread is scheduled to check itself at regular intervals whether it can terminate itself or not.

The issue related to the target threads are listed below:

What if the resources had been allotted to the cancel target thread?

What if the target thread is terminated when it was updating the data, it was sharing with some other thread.

Here the asynchronous cancellation of the thread where a thread immediately cancels the target thread without checking whether it is holding any resources or not is troublesome.

However, in deferred cancellation, the thread that indicates the target thread about the cancellation, the target thread cross checks its flag in order to confirm that it should be canceled immediately or not. The thread cancellation takes place where they can be canceled safely such points are termed as cancellation points by Pthreads.

3. Signal Handling

Signal handling is more convenient in the single-threaded program as the signal would be directly forwarded to the process. But when it comes to a multithreaded program, the issue arrives at which thread of the program the signal should be delivered.

Let's say the signal would be delivered to:

- §All the threads of the process.

- §To some specific threads in a process.

- §To the thread to which it applies

- §Or you can assign a thread to receive all the signals.

Signal Handling

Well, how the signal would be delivered to the thread would be decided, depending upon the type of generated signal. The generated signal can be classified into two type's synchronous signal and asynchronous signal.

Synchronous signals are forwarded to the same process that leads to the generation of the signal.

Asynchronous signals are generated by the event external to the running process thus the running process receives the signals asynchronously.

Signal Handling

So, if the signal is synchronous, it would be delivered to the specific thread causing the generation of the signal. If the signal is asynchronous, it cannot be specified to which thread of the multithreaded program it would be delivered. If the asynchronous signal is notifying to terminate the process the signal would be delivered to all the threads of the process.

The issue of an asynchronous signal is resolved up to some extent in most of the multithreaded UNIX systems. Here the thread is allowed to specify which signal it can accept and which it cannot. However, the Windows operating system does not support the concept of the signal; instead it uses an asynchronous procedure call (ACP) which is like the asynchronous signal of the UNIX system.

UNIX allows the thread to specify which signal it can accept and which it will not whereas the ACP is forwarded to the specific thread.

4. Thread Pool

When a user requests for a web page to the server, the server creates a separate thread to service the request. Although the server also has some potential issues. Consider if we do not have a bound on the number of active threads in a system and would create a new thread for every new request then it would finally result in exhaustion of system resources.

We are also concerned about the time it will take to create a new thread. It must not be that case that the time require to create a new thread is more than the time required by the thread to service the request and then getting discarded as it would result in wastage of CPU time

Thread Pool

The solution to this issue is the thread pool. The idea is to create a finite number of threads when the process starts. This collection of threads is referred to as the thread pool. The threads stay in the thread pool and wait till they are assigned any request to be serviced.

Whenever the request arrives at the server, it invokes a thread from the pool and assigns it the request to be serviced. The thread completes its service and returns to the pool and waits for the next request.

If the server receives a request and it does not find any thread in the thread pool it waits for some or the other thread to become free and return to the pool. This is much better than creating a new thread each time a request arrives and convenient for the system that cannot handle many concurrent threads.

5. Thread Specific data

We all know the threads belonging to the same process share the data of that process. Here the issue is what if each thread of the process needs its own copy of data. So, the specific data associated with the specific thread is referred to as thread-specific data.

Consider a transaction processing system, here we can process each transaction in a different thread. To determine each transaction uniquely we will associate a unique identifier with it. Which will help the system to identify each transaction uniquely.

Thread Specific data

As we are servicing each transaction in a separate thread. So, we can use thread-specific data to associate each thread to a specific transaction and its unique id. Thread libraries such as Win32, Pthreads and Java support thread-specific data.

So, these are threading issues that occur in the multithreaded programming environment. We have also seen how these issues can be resolved

Process vs Thread

Process:

Processes are basically the programs that are dispatched from the ready state and are scheduled in the CPU for execution. PCB(Process Control Block) holds the concept of process. A process can create other processes which are known as Child Processes. The process takes more time to terminate and it is isolated, meaning it does not share the memory with any other process.

The process can have the following [states](#) new, ready, running, waiting, terminated, and suspended.

Thread:

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread has three states: Running, Ready, and Blocked.

The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.

Difference between Process and Thread:

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.

- | | | |
|-----|--|--|
| 10. | If one process is blocked then it will not affect the execution of other processes | If a user-level thread is blocked, then all other user-level threads are blocked. |
| 11. | The process has its own Process Control Block, Stack, and Address Space. | Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space. |
| 12. | Changes to the parent process do not affect child processes. | Since all threads of the same process share address space and other resources, any changes to the main thread may affect the behavior of the other threads of the process. |

Multi-Threading Models

It is a process of multiple threads executed at same time.

Many operating systems support kernel thread and user thread in a combined way. Example of such a system is Solaris. Multi threading models are of three types.

- Many to many models.
- Many to one model.
- one to one model.

Many to Many Model

In this model, we have multiple user threads multiplexed to the same or lesser number of kernel level threads. Number of kernel level threads are specific to the machine, the advantage of this model is if a user thread is blocked we can schedule other user threads to another kernel thread. Thus, System doesn't block if a particular thread is blocked.

Many to One Model

In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access the kernel at a time, so multiple threads are not able access the multiprocessor at the same time.

The thread management is done on the user level so it is more efficient.

One to One Model

In this model, one to one relationship between kernel and user thread. In this model multiple threads can run on multiple processors. Problem with this model is that creating a user thread requires the corresponding kernel thread.

As each user thread is connected to a different kernel, if any user thread makes a blocking system call, the other user threads won't be blocked.

CPU SCHEDULING

CPU scheduling is the basis of Multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

- In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled.
- The CPU will sit idle and wait for a process that needs an I/O operation to complete. If the I/O operation completes then only the CPU will start executing the process. A lot of CPU time has been wasted with this procedure.
- The objective of multiprogramming is to have some process running at all times to maximize CPU utilization.
- When several processes are in main memory, if one process is waiting for I/O then the operating system takes the CPU away from that process and gives the CPU to another process. Hence there will be no wastage of CPU time.

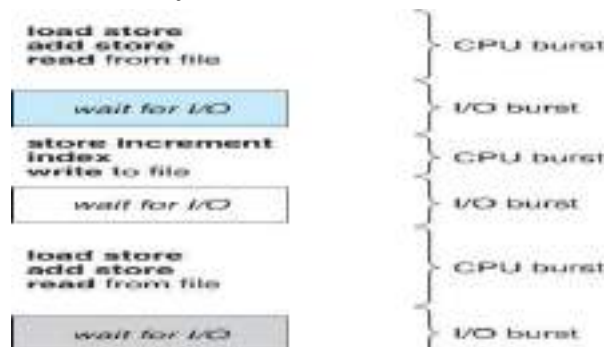
Concepts of CPU Scheduling

1. CPU-I/O Burst Cycle
2. CPU Scheduler
3. Preemptive Scheduling
4. Dispatcher

CPU-I/O Burst Cycle

Process execution consists of a **cycle** of CPU execution and I/O wait.

- Process execution begins with a **CPU burst**. That is followed by an **I/O burst**. Processes alternate between these two states.
- The final CPU burst ends with a system request to terminate execution.
- Hence the **First cycle** and **Last cycle** of execution must be CPU burst.



CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **Short-Term Scheduler** or **CPU scheduler**.

Preemptive Scheduling

CPU-scheduling decisions may take place under the following four cases:

1. When a process switches from the running state to the waiting state.
Example: as the result of an I/O request or an invocation of wait() for the termination of a child process.
2. When a process switches from the running state to the ready state.
Example: when an interrupt occurs
3. When a process switches from the waiting state to the ready state.
Example: at completion of I/O.
4. When a process terminates.

For situations 2 and 4 are considered as **Preemptive scheduling** situations. Mach OS X, WINDOWS 95 and all subsequent versions of WINDOWS are using Preemptive scheduling.

Dispatcher

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. Dispatcher function involves:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another process running is known as the **Dispatch Latency**.

SCHEDULING CRITERIA

Different CPU-scheduling algorithms have different properties and the choice of a particular algorithm may favor one class of processes over another.

Many criteria have been suggested for comparing CPU-scheduling algorithms:

- **CPU utilization:** CPU must be kept as busy as possible. CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent.
- **Throughput:** The number of processes that are completed per time unit.
- **Turn-Around Time:** It is the interval from the time of submission of a process to the time of completion. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O.
- **Waiting time:** It is the amount of time that a process spends waiting in the ready queue.
- **Response time:** It is the time from the submission of a request until the first response is produced. Interactive systems use response time as its measure.

Note: It is desirable to maximize CPU utilization and Throughput and to minimize Turn-Around Time, Waiting time and Response time.

CPU SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU. Different CPU-scheduling algorithms are:

1. First-Come, First-Served Scheduling (FCFS)
2. Shortest-Job-First Scheduling (SJF)
3. Priority Scheduling
4. Round Robin Scheduling

5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

Gantt Chart is a bar chart that is used to illustrate a particular schedule including the start and finish times of each of the participating processes.

First-Come, First-Served Scheduling (FCFS)

In FCFS, the process that requests the CPU first is allocated the CPU first.

- FCFS scheduling algorithm is Non-preemptive.
- Once the CPU has been allocated to a process, it keeps the CPU until it releases the CPU.
- FCFS can be implemented by using FIFO queues.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

Example:1 Consider the following set of processes that arrive at time 0. The processes are arrived at in the order P1, P2, P3, with the length of the CPU burst given in milliseconds.

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for FCFS is:



The average waiting time under the FCFS policy is often quite long.

- The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 and 27 milliseconds for process P3.
- Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds.

Convoy Effect in FCFS

Convoy effect means, when a big process is executing in CPU, all the smaller processes must have to wait until the big process execution completes. This will affect the performance of the system.

Example:2 Let us consider the same example above but with the processes arriving in the order P2, P3, P1.



The processes coming at P2, P3, P1 the average waiting time $(6 + 0 + 3)/3 = 3$ milliseconds whereas the processes come in the order P1, P2, P3 the average waiting time is 17 milliseconds.

Disadvantage of FCFS:

FCFS scheduling algorithm is Non-preemptive, it allows one process to keep CPU for a long time. Hence it is not suitable for time sharing systems.

Shortest-Job-First Scheduling (SJF)

The SJF algorithm is defined as “when the CPU is available, it is assigned to the process that has the smallest next CPU burst”. If the next CPU bursts of two processes are the same, FCFS

scheduling is used between two processes.

SJF is also called the Shortest-Next **CPU-Burst** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

Example: Consider the following processes and CPU burst in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart of SJF algorithm:



Waiting Time for Processes:

Process	Burst Time (ms)	Waiting Time
P1	6	3
P2	8	16
P3	7	9
P4	3	0
Average Waiting Time		7 ms

- By looking at the above table the average waiting time by using SJF algorithm is 7ms.
- SJF gives the minimum average waiting time for a given set of processes. SJF is optimal.
- The average waiting time decreases because moving a short process before a long process decreases the waiting time of the short process more than it increases the waiting time of the long process.

Difficulty with SJF

The difficulty with the SJF algorithm is “knowing the length of the next CPU request”. With Short-Term Scheduling, there is no way to know the length of the next CPU burst. It is not implemented practically.

Solution for the difficulty

One approach to this problem is to try to approximate SJF scheduling.

- We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones.
- By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
- The next CPU burst is generally predicted as an **Exponential Average** of the measured lengths of previous CPU bursts.

The following formula defines the Exponential average:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

t_n be the length of the nth CPU burst (i.e. contains the most recent information).

τ_n stores the past history.

τ_{n+1} be our predicted value for the next CPU burst.

α controls the relative weight of recent and past history in our prediction ($0 \leq \alpha \leq 1$)

- If $\alpha=0$, then $\tau_{n+1} = \tau_n$, recent history has no effect
- If $\alpha=1$ then $\tau_{n+1} = f_n$, only the most recent CPU burst matters.
- If $\alpha = 1/2$, so recent history and past history are equally weighted.

Shortest Remaining Time First Scheduling (SRTF)

SRTF is the preemptive SJF algorithm.

- A new process arrives at the ready queue, while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than the currently executing process.
- SRTF will preempt the currently executing process and execute the shortest job.

Consider the four processes with arrival times and burst times in milliseconds:

Process	Arrival time	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt Chart for SRTF



- Process P1 is started at time 0, since it is the only process in the queue.
- Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted and process P2 is scheduled.
- The average waiting time = $26/4 = 6.5$ milliseconds.

Priority Scheduling

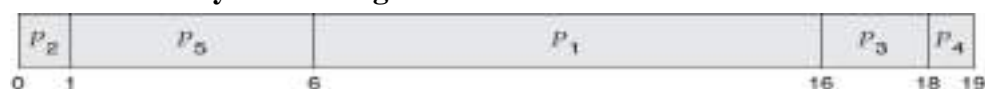
A priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

- An SJF algorithm is a special kind of priority scheduling algorithm where small CPU bursts will have higher priority.
- Priorities can be defined based on time limits, memory requirements, the number of open files etc.

Example: Consider the following processes with CPU burst and Priorities. All the processes arrive at time $t=0$ in the same order. Low numbers are having higher priority.

Process	Burst time (ms)	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt chart for Priority Scheduling:



Process	Burst time (ms)	Waiting Time
P1	10	6
P2	1	0
P3	2	16
P4	1	18
P5	5	1
Average Waiting Time		8.2 ms

Priority scheduling can be either **Preemptive or Non-preemptive**.

A **Preemptive Priority** Scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

Problem: Starvation or Indefinite Blocking

- In priority Scheduling when there is a continuous flow of higher priority processes to the ready queue then all the lower priority processes must wait for the CPU until all the higher priority processes execution completes.
- This leads to lower priority processes blocked from getting CPU for a long period of time. This situation is called Starvation or Indefinite blocking.
- In the worst case indefinite blocking may take years to execute the process.

Solution: Aging

Aging involves gradually increasing the priority of processes that wait in the system for a long time.

Round-Robin Scheduling (RR)

Round-Robin (RR) scheduling algorithm is designed especially for Time Sharing systems.

- RR is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time called a **Time Quantum** or **Time Slice** is defined. A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a **Circular queue**. New processes are added to the tail of the ready queue.
- The CPU scheduler goes around the ready queue by allocating the CPU to each process for a time interval of up to 1 time quantum and dispatches the process.
- If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.

In RR scheduling one of two things will then happen.

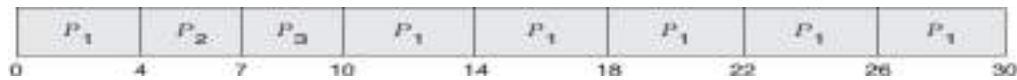
1. The process may have a CPU burst of less than 1 time quantum. The process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
2. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Consider the following set of processes that arrive at time 0 and the processes are arrived at in the order P1, P2, P3 and Time Quanta=4.

Process	Burst Time
---------	------------

P1	24
P2	3
P3	3

Gantt chart of Round Robin Scheduling



- If we use a time quantum of 4 milliseconds, then process $P1$ gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first quantum and the CPU is given to the next process in the queue, process $P2$.
- CPU burst of Process $P2$ is 3, so it does not need 4 milliseconds then it quits before its time quantum expires. The CPU is then given to the next process $P3$.
- Once each process has received 1 time quantum, the CPU is returned to process $P1$ for an additional time quantum.

The average waiting time under the RR policy is often long.

- $P1$ waits for 6 milliseconds ($10 - 4$), $P2$ waits for 4 milliseconds and $P3$ waits for 7 milliseconds. Thus, the average waiting time is $17/3 = 5.66$ milliseconds.

The performance of the RR algorithm depends on the size of the Time Quantum.

- If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- If the time quantum is extremely small (i.e. 1 millisecond) the RR approach can result in a large number of context switches.
- The time taken for context switch value should be a small fraction of Time quanta then the performance of the RR will be increased.

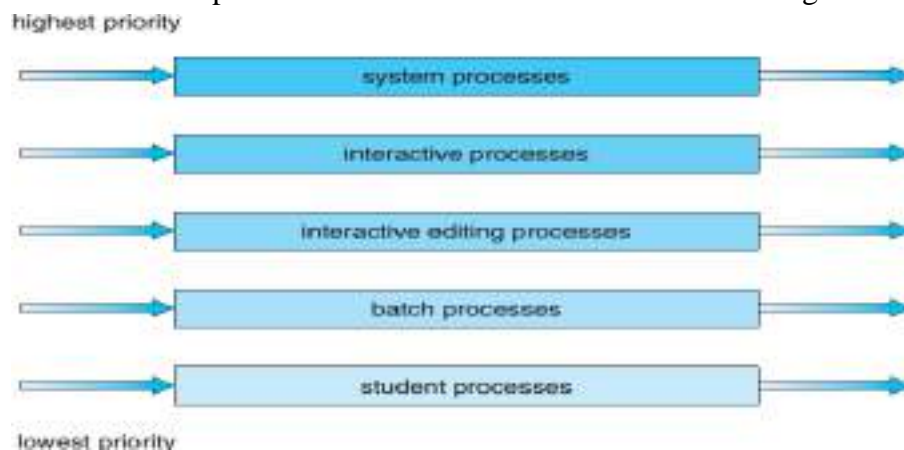
Note: A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Multi-Level Queue Scheduling (MLQ)

In the Multilevel Queue Scheduling algorithm the processes are classified into different groups.

- A Multilevel queue scheduling partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue based on memory size, process priority or process type. Each queue has its own scheduling algorithm.

Example: Foreground processes have highest priority over background processes and these processes have different response times hence it needs different scheduling.



The above figure shows Multi-level queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower-priority queues.

- No lower level queue processes will start executing unless all the processes in the higher level queue are empty.

Example: The interactive processes start executing only when all the processes in the system queue are empty.

- If a lower priority process is executing and a higher priority process enters into the queue then a lower priority process will be preempted and starts executing a higher priority process.

Example: If a system process entered the ready queue while an interactive process was running, the interactive process would be preempted.

Disadvantage: Starvation of Lower level queue

The multilevel queue scheduling algorithm is inflexible.

- The processes are permanently assigned to a queue when they enter the system. Processes are not allowed to move from one queue to another queue.
- There is a chance that lower level queues will be in starvation because unless the higher level queues are empty no lower level queues will be executing.
- If at any instant of time if there is a process in higher priority queue then there is no chance that lower level process can be executed eternally.

Multilevel Feedback Queue Scheduling is used to overcome the problem of Multi-level queue scheduling.

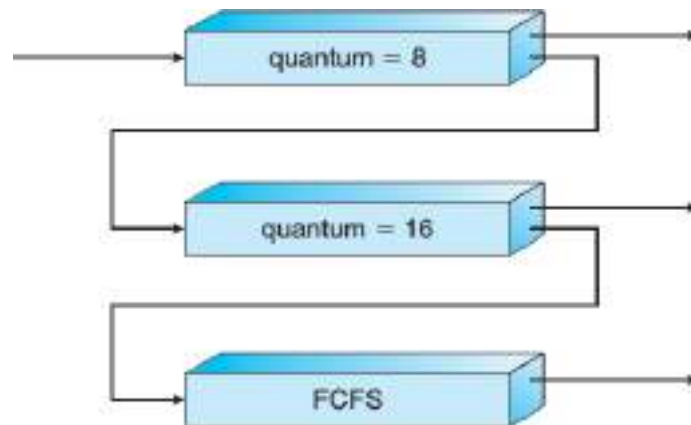
Multilevel Feedback Queue Scheduling (MLFQ)

Multilevel feedback queue scheduling algorithm allows a process to move between queues.

- Processes are separated according to the characteristics of their CPU bursts.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- A process that waits too long in a lower-priority queue moved to a higher-priority queue.
- This form of aging prevents starvation.

Consider a multilevel feedback queue scheduler with three queues: queue0, queue1, queue2.

- The scheduler first executes all processes in queue0 then queue1 and then queue2.
- Only when queue 0 and queue1 is empty, the scheduler will execute processes in queue2.
- A process that arrives for queue1 will preempt a process in queue2. A process in queue1 will in turn be preempted by a process arriving for queue0.



- A process entering the ready queue is put in queue0. A process in queue 0 is given a time quantum of 8ms. If it does not finish within this time, it is moved to the tail of queue 1.
- If queue 0 is empty, the process at the head of queue1 is given a quantum of 16ms. If it does not complete, it is preempted and is put into queue2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8ms or less. Such a process will quickly get the CPU and finish its CPU burst and go off to its next I/O burst.
- Processes that need more than 8ms but less than 24ms are also served quickly, although with lower priority than shorter processes.
- Long processes automatically sync to queue2 and are served in FCFS order with any CPU cycles left over from queue 0 and queue1.

A Multi-Level Feedback queue scheduler is defined by the following parameters:

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher priority queue.
- The method used to determine when to demote a process to a lower priority queue.
- The method used to determine which queue a process will enter when that process needs service.

MULTIPLE-PROCESSOR SCHEDULING

Scheduling process will become complex with multiple CPU structures but Load Sharing is possible with multiple CPU structures.

In multiple processor systems, we can use any available processor to run any process in the queue.

Multiprocessor Scheduling Approaches

There are two approaches to multiprocessing: **Asymmetric** and **Symmetric** Multiprocessing.

- In **Asymmetric Multiprocessing**, Master and Worker relationships exist.

The master server is a single processor that handles the activities related to all scheduling

decisions, I/O processing and other system activities. The other processors execute only user code.

- Asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.
- In **Symmetric Multiprocessing (SMP)** each processor is self-scheduling. All processes may be in a common ready queue or each processor may have its own private ready queue processes.
- Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
- When multiple processors try to access and update a common data structure, the scheduler must be programmed carefully.
- Two separate processors do not choose to schedule the same process and that processes are not lost from the queue.

Processor Affinity

The Process Affinity is “to make a process run on the same CPU it ran on last time”.

- The processor cache contains the data that is most recently accessed by the process.
- If the process migrates from one processor to another processor, the contents of cache memory must be invalidated for the first processor and the cache for the second processor must be entered again.
- Because of the high cost of invalidating and re-entering caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.

Processor affinity can be implemented in two ways: Soft affinity and Hard affinity.

- **Soft affinity:** The operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
- **Hard affinity:** Some systems provide system calls that support **Hard Affinity**, thereby allowing a process to specify a subset of processors on which it may run.

Load Balancing

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

Load balancing is necessary only on systems where each processor has its own private queue of processes to execute.

There are two approaches to load balancing: **Push Migration** and **Pull Migration**.

- **Push migration:** A specific process periodically checks the load on each processor. If the task finds an imbalance then it evenly distributes the load by moving (pushing) processes from overloaded processors to idle or less-busy processors.
- **Pull migration:** It occurs when an idle processor pulls a waiting process from a busy processor.

Multicore Processors

In a multicore processor each core acts as a separate processor. Multicore processors may complicate scheduling issues.

- When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This waiting time is called a **Memory Stall**.
- Memory Stall may occur for several reasons for example Cache miss.

- In order to avoid memory stall, many recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. That way, if one thread stalls while waiting for memory, the core can switch to another thread.



Figure 6.11 Multithreaded multicore system.

- The execution of thread0 and the execution of thread 1 are interleaved on a dual-threaded processor core.
- From an operating-system perspective, each **Hardware Thread** appears as a **Logical Processor** that is available to run a software thread.
- Thus, on a **Dual-threaded, Dual-core** system, **Four** logical processors are presented to the operating system.

There are two ways to multithread a processing core: **Coarse-grained** and **Fine-grained**

Coarse-Grained Multithreading:

- A thread executes on a processor until a long-latency event such as a memory stall occurs.
- The processor must switch to another thread to begin execution, because of the delay caused by the long-latency event.
- The cost of switching between threads is high, since the instruction pipeline must be flushed before the other thread can begin execution on the processor core.
- Once this new thread begins execution, it begins filling the pipeline with its instructions.

Fine-grained (or) Interleaved Multithreading:

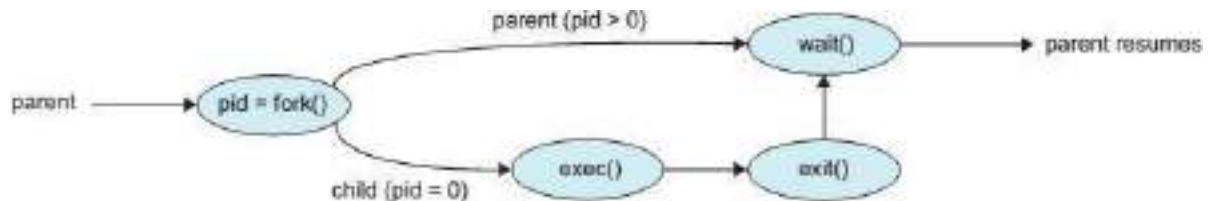
- Thread switching done at the boundary of an instruction cycle.
- The architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.

INTRODUCTION TO SYSTEM CALLS

Process System calls in Unix/ Linux: fork(), exec(), wait(), exit(), vfork(), waitpid()

- **fork()**: In UNIX OS a new process is created by the **fork()** system call.
- The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both the parent and the child processes continue execution at the instruction after the fork().
- For the new child process (i.e. Child Process) the return code for the fork() is zero.
- The nonzero process identifier of the child is returned to the parent.
- **exec()**: After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.

- The `exec()` system call loads a binary file into memory and starts its execution.
- In this way, the two processes are able to communicate and then go their separate ways.
- **wait() :** The parent can create more children or if the parent has nothing else to do while the child process is running then the parent process can issue a **wait()** system call to move itself out of the Ready Queue until the child process terminates.
- The call to `exec()` overlays the process's address space with a new program or the call to `exec()` does not return control unless an error occurs.



Program for Creating a separate process using the UNIX 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);
        /* child to complete */
        wait(NULL); printf("Child
        Complete");}

    return 0;
}
}
else { /* parent process */
    /* parent will wait for the

```

The above C program shows the UNIX system calls fork, exec, wait. Two different processes are running copies of the same program.

- The only difference is that the value of pid for the child process is zero, while the value of pid for the parent is an integer value greater than zero (i.e. the actual pid of the child process).
- The child process inherits privileges and scheduling attributes from the parent as well as certain resources such as open files.
- The child process then overlays its address space with the UNIX command /bin/l_s (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes by either implicitly or explicitly invoking exit(), the parent process resumes from the call to wait(), where it completes using the exit() system call.

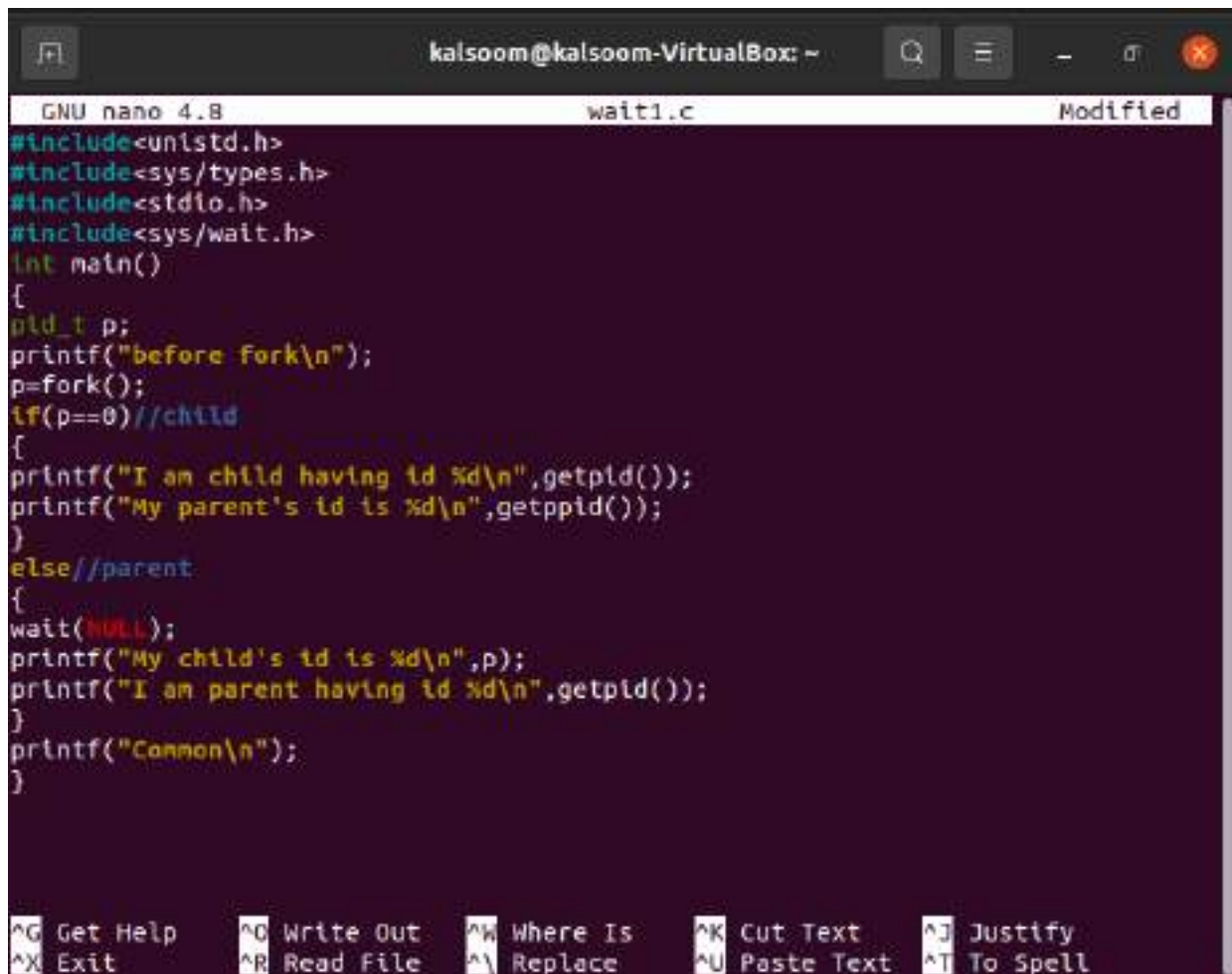
wait() and waitpid()

When a process develops a child process, it's occasionally important for the parent process to wait till the child has completed it before continuing. This is exactly what the wait () system function accomplishes.

Waiting causes the parent to wait for the child to alter its state. The status change could be due to the child process being terminated, stopped by a signal, or resumed by a signal. In some circumstances, when a child process quits or switches state, the parent process should be notified of the child's alteration in state or termination state. In that instance, the parent process uses functions like wait () to inquire about the update in the state of the child process.

Wait () suspends the caller process till the system receives information on the ending child's status. Wait () returns instantly if the system already has status information on a finished child process when invoked. If the caller process receives the signal with action to run a signal handler or terminate the process, wait () is also terminated.

The waitpid () system function pauses the current process until the pid argument specifies a child with an altered state. Waitpid() waits solely for terminated children by default; however, this behavior can be changed. The wait () system call accepts just one parameter, which holds the process's information and updates. If you don't care about the child process's exit status and only care about making the parent wait for the child, use NULL as the value. In this guide, we will elaborate on an example for the understanding of the Wait () system call in C programming.



```
GNU nano 4.8 wait1.c Modified
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)//child
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else//parent
{
wait(NULL);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell

amming.



```
kalsoom@kalsoom-VirtualBox:~$ ./a.out
before fork
I am child having id 8039
My parent's id is 8038
Common
My child's id is 8039
I am parent having id 8038
Common
kalsoom@kalsoom-VirtualBox:~$
```

Waitpid()

The **waitpid()** system call monitors a child of the caller process for state changes and retrieves information about the child whose behavior has changed. The child was halted by a signal or resumed by a signal regarded as a state shift. Waiting for a terminated child enables the system to free the resources associated with the child; if no wait is conducted, the terminated child will remain in a “**zombie**” condition

The **waitpid()** system function pauses the current process until the PID argument specifies a child who has changed. The calling process is paused until a child’s process completes or is terminated. **Waitpid()** halts the calling process till the system receives information about the child’s status. **Waitpid()** returns quickly if the system already has status information on a suitable

child when it is called. If the caller process gets a signal with the action of either executing a signal handler or terminating the process, **waitpid()** is terminated. The **waitpid()** function will pause the caller thread's execution until it receives information and updates for one of its terminated child processes or a signal that will either run a signal-catching procedure or terminate the process.

Syntax of waitpid():

```
pid_t waitpid(pid_t pid, int *status, int options);
```

The value of pid can be:

- **< -1:** Wait for any child process whose process group ID is equal to the absolute value of pid.
- **-1:** Wait for any child process.
- **0:** Wait for any child process whose process group ID is equal to that of the calling process.
- **> 0:** Wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

- **WNOHANG:** Return immediately if no child has exited.
- **WUNTRACED:** Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.
- **WCONTINUED:** Also return if a stopped child has been resumed by delivery of SIGCONT.

Example:



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

int main()
{
    pid_t p, q;
    printf("Before fork\n");
    p=fork();
    if(p==0)
    {
        printf("I am the first child having pid is %d\n",getpid());
        printf("My parent id is %d\n", getppid());
    }
    else
    {
        q=fork();
        if(q==0)
        {
            sleep(2);
            printf("I am a second child having pid %d\n",getpid());
            printf("My parent pid is %d\n",getppid());
        }
        else
        {
            waitpid(p, &WEXITSTATUS, 0);
            printf("I am parent having id %d\n",getpid());
            printf("My first child pid is %d\n",p);
            printf("My second child pid is %d\n",q);
        }
    }
    printf("Program\n");
}
```

Output:

```
priyanka@ECSTCP-ALH082:~$ gcc waitpid.c -o waitpid
priyanka@ECSTCP-ALH082:~$ ./waitpid
before fork
I am the first child having pid is 47
My parent id is 46
I am parent having id 46
My first child pid is 47
My second child pid is 48
common
priyanka@ECSTCP-ALH082:~$ I am a second child having pid 48
My parent pid is 1
common
```

Exiting a process: Process Termination: `exit()`

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **`exit()`** system call.
- The process may return a status value to its parent process via the `wait()` system call.
- All the resources of the process including physical and virtual memory, open files and I/O buffers are deallocated by the operating system.

A parent may terminate the execution of one of its children for a variety of reasons such as:

1. The child has exceeded its usage of some of the resources that it has been allocated.
2. The task assigned to the child is no longer required.
3. The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

The `exit()` is such a function or one of the system calls that is used to terminate the process. This system call defines that the thread execution is completed especially in the case of a multi-threaded environment. For future reference, the status of the process is captured.

After the use of `exit()` system call, all the resources used in the process are retrieved by the operating system and then terminate the process. The system called `Exit()` is equivalent to `exit()`.

Synopsis

```
#include <unistd.h>

void _exit(int status);

#include <stdlib.h>

void _Exit(int status);
```


Cascading Termination

If a parent process terminates either normally or abnormally then all its children must also be terminated is referred as Cascading Termination. It is normally initiated by the operating system.

In Linux and UNIX systems, a process can be terminate by using the `exit()` system call providing an exit status as a parameter:

```
/* exit with status 1 */
```

```
exit(1);
```

Under normal termination, `exit()` may be called either directly (i.e. `exit(1)`) or indirectly (i.e. by a return statement in `main()`).

A parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid_t pid; int status;
```

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

vfork()

`vfork()` is a special case of `clone(2)`. It is used to create new processes without copying the page tables of the parent process. It may be useful in performance sensitive applications where a child will be created which then immediately issues an `execve()`.

`vfork()` differs from `fork()` in that the parent is suspended until the child makes a call to `execve(2)` or `_exit(2)`. The child shares all memory with its parent, including the stack, until `execve()` is issued by the child. The child must not return from the current function or call `exit()`, but may call `_exit()`.

`vfork` → create a child process and block parent process.

Note:- In `vfork`, signal handlers are inherited but not shared.

```
admin@linux ~}$ man vfork
```

This will yield output mentioning what is `vfork` used for, syntax and along with all the required details.

```
pid_t vfork(void);
```

`vfork` is as same as `fork` except that behavior is undefined if process created by `vfork` either modifies any data other than a variable of type `pid_t` used to store the return value `p` of `vfork` or calls any other function between calling `_exit()` or one of the `exec()` family.

Note: `vfork` is sometimes referred to as special case of `clone`.

Following is the C programming example for `vfork()` how it works.

```

admin@linux ~}$ vim 1.vfork.c

#include<stdio.h>

#include<unistd.h>

Int main(void)

{

printf("Before fork\n");

vfork();

printf("after fork\n");

}

admin@linux ~}$ vim 1.vfork.c

admin@linux ~}$ cc 1.vfork.c

admin@linux ~}$ ./a.out

Before vfork

after vfork

after vfork

a.out: cxa_atexit.c:100: __new_exitfn: Assertion `l!= NULL'
failed.

Aborted

```

Note:– As explained earlier, many a times the behaviour of the vfork system call is not predictable. As in the above case it had printed before once and after twice but aborted the call with _exit() function. It is better to use fork system call unless otherwise and avoid using vfork as much as possible.

3.2 Deadlocks

A set of processes is in a Deadlock state when every process in the set is waiting for an event that

can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and resource release.

3.2.1 System Model: A system consists of a finite number of resources to be distributed among a number of competing processes.

Resources are categorized into two types: Physical resources and Logical resources

- **Physical resources:** Printers, Tape drives, DVD drives, memory space and CPU cycles
- **Logical resources:** Semaphores, Mutex locks and files.

Each resource type consists of some number of identical instances. (i.e.) If a system has two CPU's then the resource type CPU has two instances.

A process may utilize a resource in the following sequence under normal mode of operation:

- **Request:** The process requests the resource. If the resource is being used by another process then the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
- **Use:** The process can operate on the resource.

Example: If the resource is a printer, the process can print on the printer.

- **Release:** The process releases the resource.

System calls for requesting and releasing resources:

- Device System calls: request() and release()
- Semaphore System calls: wait(), signal()
- Mutex locks: acquire(), release().
- Memory System Calls: allocate() and free()
- File System calls: open(), close().

A **System Table** maintains the status of each resource whether the resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

3.2.2 Deadlock Characterization

describes the distinctive features that are the cause of deadlock occurrence. Deadlock is a condition in the multiprogramming environment where the executing processes get stuck in the middle of execution waiting for the resources that have been held by the other waiting processes thereby preventing the execution of the processes.

Deadlock Characterization

- i. Deadlock Prerequisites
- ii. Systems Resource Allocation Graph

i. Deadlock Prerequisites

Four Necessary Conditions of Deadlock

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **Mutual exclusion.** Only one process at a time can use the resource. If other process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- **No preemption.** If a process holding a resource and the resource cannot be preempted until the process has completed its task.
- **Circular wait.** A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_0 .

ii. Systems Resource Allocation Graph

The resource allocation graph is used for identification of deadlocks in the system.

A System Resource-Allocation Graph $G=\{V,E\}$ is a directed graph that consists of a set of vertices V and a set of edges E .

The set of vertices V is partitioned into two types of nodes: Processes and Resources.

1. Process set $P= \{P_1, P_2, ..., P_n\}$ consisting of all the active processes in the system.
2. Resource set $R= \{R_1, R_2, ..., R_m\}$ consisting of all resource types in the system.

The set of Edges E is divided into types: Request Edge and Assignment Edge.

1. Request Edge ($P_i \rightarrow R_j$): It signifies that process P_i has requested an instance of resource type R_j and P_i is currently waiting for the resource R_j .
2. Assignment edge ($R_j \rightarrow P_i$): It signifies that an instance of resource type R_j has been allocated to process P_i .

Processes can be represented in Circles and Resources can be represented in Rectangles.

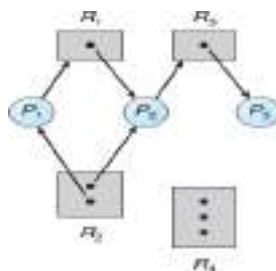
Instance of resource can be represented by a Dot.

Resource allocation graph shows three situations:

1. Graph with No deadlock
2. Graph with a cycle and deadlock
3. Graph with a cycle and no deadlock

Resource Allocation Graph without Deadlock

The below graph consists of three sets: Process **P**, Resources **R** and Edges **E**



- Process set $P= \{P_1, P_2, P_3\}$.
- Resources set $R= \{R_1, R_2, R_3, R_4\}$.
- Edge set $E= E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$.

Resource type R_1 and R_3 has only one instance and R_2 has two instances and R_4 has three instances.

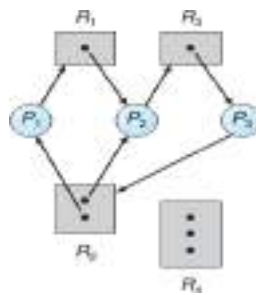
The Resource Allocation Graph depicts that:

- $R_2 \rightarrow P_1, P_1 \rightarrow R_1$: P_1 is holding an instance of resource type R_2 and is waiting for an instance of R_1 .
- $R_1 \rightarrow P_2, R_2 \rightarrow P_2, P_2 \rightarrow R_3$: Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3 .
- $R_3 \rightarrow P_3$: Process P_3 is holding an instance of R_3 .

The above Resource allocation graph does not contain any cycle then there is no

process in the system is deadlocked.

Resource Allocation Graph with a Cycle and Deadlock

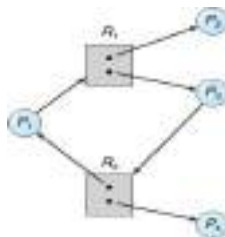


Consider the above graph, with processes and Resources and have some

edges: $P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

- Process P2 is waiting for the resource R3, which is held by process P3.
- Process P3 is waiting for either process P1 or process P2 to release resource R2.
- Process P1 is waiting for process P2 to release resource R1. Hence the Processes $P1$, $P2$ and $P3$ are deadlocked.



Resource Allocation Graph with a Cycle and No Deadlock

The graph has a cycle: $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$.

- This cycle does not lead to deadlock, because the process P4 and P2 is not waiting for any resource.
- Process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

3.3 Methods For Handling Deadlocks

The Deadlock can be handled by 3 methods:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

3.3.1 Deadlock Prevention: Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. (i.e.) Deadlock can be prevented if any of mutual exclusion, Hold and wait, No preemption and Circular wait condition cannot hold.

Mutual Exclusion The mutual exclusion condition must hold when at least one resource must be non-sharable.

- We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources by default are non-sharable.

Example 1: A mutex lock cannot be simultaneously shared by several processes.

Example 2: Printer is a resource where only one process can use it.

- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Example: Read-only files.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource.

Hold and Wait To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

- **Protocol 1:** Each process can request the resources and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

Example: Consider a process that copies data from a DVD drive to a file on Hard disk, sorts the file and then prints the results to a Printer.

If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file and Printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

- **Protocol 2:** A process can be allowed to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Example: Consider a process that copies data from a DVD drive to a file on Hard disk, sorts the file and then prints the results to a Printer.

The process to request initially only the DVD drive and Hard disk file. It copies from the DVD drive to the Hard disk and then releases both the DVD drive and the disk file. The process must then request the Hard disk file and the Printer. After copying the disk file to the printer, it releases these two resources and terminates.

Problem: Starvation and Low Resource utilization

Resource utilization is low, since resources may be allocated but unused for a long period.

A process that needs several resources may have to wait indefinitely leads to starvation.

No Preemption To ensure that No preemption condition does not hold, we can use the following protocol:

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted (i.e.) resources are implicitly released.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources as well as the new resources that it is requesting.

Circular Wait

One way to ensure that circular wait condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Consider the set of resource types $R = \{R_1, R_2, \dots, R_m\}$ and N be the set of natural numbers. we define a one-to-one function $F: R \rightarrow N$.

- The function assigns each resource type to a unique integer number, which allows us to compare two resources and to determine whether one resource precedes another resource in our ordering.

Example: If the set of resource types R includes tape drives, disk drives and printers, then the function $F: R \rightarrow N$ might be defined as follows:

$F(\text{Tape drive}) = 1$ ($F: \text{Tape drive}$
 $\rightarrow 1$) $F(\text{Disk drive}) = 5$ ($F: \text{Disk}$
 $\text{drive} \rightarrow 1$) $F(\text{Printer}) = 12$ ($F:$

Printer $\rightarrow 1$)

We can now consider the following protocol to prevent deadlocks:

- Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type R_i .
- After that, the process can request instances of resource type R_j iff $F(R_j) > F(R_i)$

Example: A process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

- Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.

Disadvantage of Deadlock Prevention

Deadlock-prevention algorithms lead to low resource utilization and the system throughput will be reduced.

3.3.2 Deadlock Avoidance In the process first informs the operating system about their maximum allocation of resources to be requested and used during its life time.

- With this information, the operating system can decide for each request whether the resource will be granted immediately or the process should wait for resources.
- To take this decision about the resource allocation, the operating system must consider the resources currently available, the resources currently allocated to each process and the future requests and releases of each process.

Algorithms for Deadlock Avoidance

A deadlock-avoidance algorithm dynamically examines the Resource-Allocation State to ensure that a circular-wait condition can never exist.

The Resource Allocation State is defined by the number of available resources and allocated resources and the maximum demands of the processes.

There are three algorithms designed for deadlock avoidance:

1. Safe State
2. Resource Allocation Graph Algorithm
3. Bankers Algorithm

1. Safe State Algorithm If the system can allocate resources to each process up to its maximum in some order and still avoid a deadlock then the state is called Safe state.

- A system is in a safe state only if there exists a Safe sequence.
- A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state, if for each process P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$.
- In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished.
- When P_j have finished its task, P_i can obtain all of its needed resources and after completing its designated task P_i can return its allocated resources and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources and soon.
- If no such sequence exists, then the system state is said to be unsafe.

Example: Consider a system with 12 magnetic tape drives and 3 processes: P_0 , P_1 and P_2 .

Process	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

The above table describes as follows:

- Process **P0** requires 10 tape drives, **P1** needs 4 tape drives and **P2** need 9 tapedrives.
- At time t_0 , process P0 is holding 5 tape drives, P1 and P2 is holding 2 tape drives each.
- Now there are 3 free tapedrives.

At time t_0 , the system is in a safe state. $\langle P1, P0, P2 \rangle$ sequence satisfies the safety condition.

- Process **P1** can immediately be allocated all its tape drives and then return all 4 resources. (i.e.) P1 currently holding 2 tape drives and out of 3 free tape drives 2 tape drives will be given to P1. Now P1 is having all 4 resources. Hence P1 will use all of its resources and after completing its task P1 releases all 4 resources and then returns to the system. Now the system is having **5 available** tapedrives.
- Now process **P0** needs 5 tape drives and the system has 5 available tape drives. Hence **P0** can get all its tape drives and it reaches its maximum 10 tape drives. After completing its task P0 returns the resources to the system. Now system has 10 available tapedrives.
- Now the process **P2** needs 7 additional resources and system have 10 resources available. Hence process **P2** can get all its tape drives and return them. Now the system will have all 12 tape drives available.

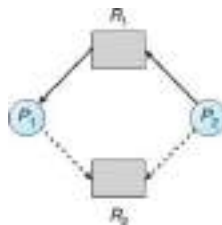
Problem: Low Resource utilization

If a process requests a resource that is currently available, it may still have to wait. Hence there exist a low resource utilization is possible.

2. Resource-Allocation-Graph Algorithm

In this algorithm we use three edges: request edge, assignment edge and a claim edge.

- Claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future.
- Claim edge resembles a request edge in direction but is represented by dashed line.
- When process P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge.
- When a resource R_j is released by P_i , the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.
- The resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-



allocation graph.

Now suppose that process P_i requests resource R_j .

- The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm.
- If no cycle exists, then the allocation of the resource will leave the system in a

safestate.

- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied.

Example: consider the above resource-allocation graph. Suppose that P_2 requests R_2 .

- R_2 is currently free still we cannot allocate it to P_2 , since this will create a cycle in graph.
- A cycle indicates that the system is in an unsafe state.
- If P_1 requests R_2 and P_2 requests R_1 , then a deadlock will occur.

Problem: The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

3. Banker's Algorithm is used in a system with multiple instance of each resource type.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Banker's algorithm uses two algorithms:

1. Safety algorithm
2. Resource-Request algorithm

Banker's algorithm:

- When a new process enters the system, the process must declare the **Maximum** number of instances of each resource type that it may need.
- The **Maximum** number may not exceed the **Total** number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If the system is in safe state then the resources are allocated.
- If the system is in unsafe state then the process must wait until some other process releases enough resources.

Data structures used to implement the Banker's algorithm

Consider the system with n number of processes and m number of resource types:

- **Available_m:** A vector of length m indicates the number of available resources of each type.
- **Max_{n×m}:** An $n \times m$ matrix defines the maximum demand of each process.
- **Allocation_{n×m}:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Need_{n×m}:** An $n \times m$ matrix indicates the remaining resource need of each process.

$$\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j].$$

- $\text{Available}[j] = k$ means then k instances of resource type R_j are available.
- $\text{Max}[i][j] = k$ means process P_i may request at most k instances of resource type R_j .
- $\text{Allocation}[i][j] = k$ means process P_i is currently allocated k instances of resource type R_j .
- $\text{Need}[i][j] = k$ means process P_i may need k more instances of resource type R_j to complete its task.

Each row in the matrices $\text{Allocation}_{n \times m}$ and $\text{Need}_{n \times m}$ are considered as vectors and refer to them as Allocation_i and Need_i .

- The vector Allocation_i specifies the resources currently allocated to process P_i .

- The vector $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Safety algorithm

Safety algorithm finds out whether the system is in safe state or not. The algorithm can be described as follows:

1. Let **Work** and **Finish** be vectors of length m and n , respectively. We initialize
 $Work = Available$
 $Finish[i] = false$ for $i = 0, 1, \dots, n - 1$.
2. Find an index i such that both
 $Finish[i] ==$
 $false$ $Need_i$
 $\leq Work$
 If no such i exists, go to step 4.
3. $Work = Work +$
 $Allocation_i$ $Finish[i]$
 $= true$
 Go to step 2.
4. If $Finish[i] == true$ for all i , then the system is in a safe state.

Note: To determine a safe state, this algorithm requires an order of $m \times n^2$ operations.

Resource-Request Algorithm

This algorithm determines whether requests can be safely granted.

Let $Request_i$ be the request vector for process P_i . If $Request_i[j] == k$, then process P_i wants k instances of resource type R_j .

When a request for resources is made by process P_i , the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2.
 Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3.
 Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
 $Available = Available - Request_i$
 $;$
 $Allocation_i = Allocation_i +$
 $Request_i;$
 $Need_i = Need_i - Request_i ;$
4. If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources.
 If the new state is unsafe, then P_i must wait for $Request_i$ and the old resource-allocation state is restored.

Example for Banker's Algorithm

Consider a system with 5 processes: **P0, P1, P2, P3, P4** and 3 resource types **A, B** and **C** with **10, 5, 7** instances respectively. (i.e.) . Resource type **A=10, B= 5** and **C=7** instances. Suppose that, at time T_0 , the following snapshot of the system has been taken:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C

P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

The Available vector can be calculated by subtracting total no of resources from the sum of resources allocated to each process.

Available resources of A= Total resources of A – Sum of resources allocated to Process P1 to P4

The Need matrix can be obtained by using $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$

	Max			Allocation			Need		
	A	B	C	A	B	C	A	B	C
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

By using the banker's algorithm we can decide whether the state is safe or not.

After solving the above problem by using bankers algorithm we will get to a safe state with safe sequence $\langle P1, P3, P4, P0, P2 \rangle$.

3.3.3 Deadlock Detection: If a system does not employ either a Deadlock-Prevention or a Deadlock-Avoidance algorithm then a deadlock situation may occur. In this environment, the system may provide:

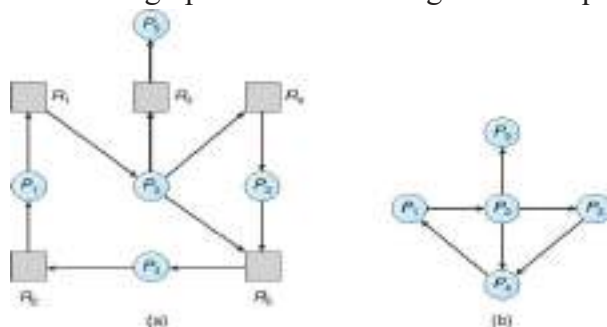
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock.

Deadlock Detection in Single Instance of Each Resource Type

If all resources have only a single instance then we can define a Deadlock-Detection algorithm that uses a variant of the resource-allocation graph called a wait-for graph.

We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

- An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q .



- In above figure we present a resource-allocation graph and the corresponding wait-for graph. A deadlock exists in the system if and only if the wait-for graph contains acycle.

- To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph.
 - An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

We will implement a Deadlock Detection algorithm that is similar to the Banker's algorithm. The data structures used in Deadlock Detection algorithm is:

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process.

If $\text{Request}[i][j] = k$, then process P_i is requesting k more instances of resource type R_j .

The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively. We initialize $Work = Available$. For $i = 0, 1, \dots, n-1$. if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
2. Find an index i such that both
 - a. $Finish[i] == false$
 - b. $Request_i \leq Work$
 If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
 $Finish[i] = true$
 Go to step 2.
4. If $Finish[i] == false$ for some i , $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.
5. This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Example: Consider a system with 5 processes: P_0, P_1, P_2, P_3, P_4 and 3 resource types A, B and C with 10, 5, 7 instances respectively. (i.e.) . Resource type A=7, B= 2 and C=6 instances. Suppose that, at time T_0 , we have the following resource-allocation state:

Allocation			Request			Available		
A	B	C	A	B	C	A	B	C

P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Initially the system is not in Deadlock State. If we apply the Deadlock Detection algorithm we will find the sequence < P0, P2, P3, P1, P4 > results in Finish[i] == true for all i. The system is in safe state hence there is no deadlock.

3.3.4 Deadlock Recovery

There are two options for breaking a deadlock.

1. Process termination
2. Resource Preemption

1. **Process Termination** To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all Deadlocked processes:** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the Deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Many factors may affect which process is chosen for preempting includes:

1. Priority of the process.
2. How long the process has computed and how much longer the process will compute before completing its designated task.
3. How many and what types of resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated?
6. Whether the process is Interactive or Batch.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. There are 3 issues related to Resource Preemption:

1. **Selecting a victim.** As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. **Rollback.** If we preempt a resource from a process then the process cannot continue with its normal execution. It is missing some needed resource. We must do total roll back of the process and restart it from that state: abort the process and then restart it.
3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process.