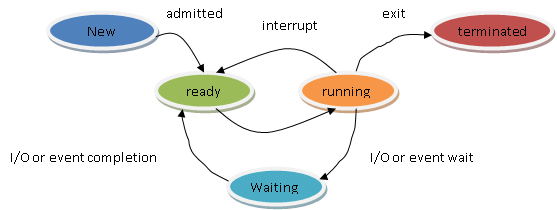
**Assignment 3**

**Read the following to understand the parent & child process concept.**

* Process state Diagram
* A process changes its state during its execution. Each process may be in one of the following states:

1. **New:**when a new process is being created.
2. **Running:** A process is said to be in running state when instructions are being executed.
3. **Waiting:** The process is waiting for some event to occur (such as an I/O operation).
4. **Ready:** The process is waiting for processor.
5. **Terminated:** The process has finished execution.
6. 

* The parent & child process concept

## Parent Process

All the processes in operating system are created when a process executes the fork() system call except the startup process. The process that used the fork() system call is the parent process. In other words, a parent process is one that creates a child process. A parent process may have multiple child processes but a child process only one parent process.

## Child Process

A child process is a process created by a parent process in operating system using a fork() system call. A child process may also be called a subprocess or a subtask.

* Understand orphan processes
* An [orphan process](https://en.wikipedia.org/wiki/Orphan_process) is a running process whose parent process has **finished** or **terminated**.
* In a Unix-like operating system any orphaned process will be immediately adopted by the special init system process. This operation is called re-parenting and occurs automatically.
* Understand zombie processes
* A [zombie process](https://en.wikipedia.org/wiki/Zombie_process) (or defunct process) is a process that has completed execution but hasn’t been reaped by its **parent process**. As result it holds a **process entry** and the **PID** in the **process table**.

A **zombie process** is similar to a [“memory leak”](https://en.wikipedia.org/wiki/Memory_leak): a loss of a system resource caused by failure to release previously reserved portion. A **zombie process** is usually a sign of invalid software behavior.

* what is **getpid()** and how to use it..??

It returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

pid\_t getpid(void);

getpid() returns the process ID of the current process. It never throws any error therefore is always successful.

// C++ Code to demonstrate getpid()

#include <iostream>

#include <unistd.h>

using namespace std;

// Driver Code

int main()

{

int pid = fork();

if (pid == 0)

cout << "\nCurrent process id of Process : "

<< getpid() << endl;

return 0;

}

* what is **getppid()** and how to use it..??
* **getppid() :** returns the process ID of the parent of the calling process. If the calling process was created by the [**fork()**](https://www.geeksforgeeks.org/fork-system-call/) function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for **init** process.  
  **Syntax:**

pid\_t getppid(void);

* **Return type:** getppid() returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

// C++ Code to demonstrate getppid()

#include <iostream>

#include <unistd.h>

using namespace std;

// Driver Code

int main()

{

int pid;

pid = fork();

if (pid == 0)

{

cout << "\nParent Process id : "

<< getpid() << endl;

cout << "\nChild Process with parent id : "

<< getppid() << endl;

}

return 0;

}

* what is **fork()**? what happens when you use fork() in your program..??

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

* If **fork()** returns a negative value, the creation of a child process was unsuccessful.
* **fork()** returns a zero to the newly created child process.
* **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid\_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.
* what is **wait()** / **waitpid()** and how to use it..??

The **wait()** system call suspends execution of the current process until one of its children terminates. The call **wait**(&status) is equivalent to: **waitpid**(-1, &status, 0); The **waitpid()** system call suspends execution of the current process until a child specified by pid argument has changed state.

* what are **zombie processes**? when they will get created.? How to kill the zombie processes..?

**Zombie processes** usually occur for child **processes**, as the parent **process** still needs to read its child's exit status. Once this is done using the wait system call, the **zombie process** is eliminated from the **process** table. This is known as reaping the **zombie process**

* what are **orphan ​processes**? when they will get created.?

An **orphan process** is a running **process** whose parent **process** has finished or terminated. In a Unix-like operating system any **orphaned process will be** immediately adopted by **the** special init system **process**. This operation is called re-parenting and occurs automatically.

* what is user space?

**User space** refers to all of the code in an operating system that lives outside of the kernel. Most Unix-like **operating systems** (including Linux) come pre-packaged with all kinds of utilities, programming languages, and graphical tools - these are **user space** applications. We often refer to this as “userland.”

* what is kernel space / system space?

**Kernel space** is where the **kernel** (i.e., the core of the operating **system**) executes (i.e., runs) and provides its services. ... User **space** is that set of memory locations in which user processes (i.e., everything other than the **kernel**) run. A process is an executing instance of a program

* What is **IPC**(inter process communication) and what is the need of it.

**Inter process communication** (**IPC**) is used for exchanging data between multiple threads in one or more **processes** or programs. ... It is a set of programming interface which allow a programmer to coordinate activities among various program **processes** which can run concurrently in an operating system.

* what is **Unnamed PIPE?** how to create it? When we need this??
* what is **named PIPE**? how to create it? When we need this??

A named pipe is a one-way or duplex pipe that provides communication between the pipe server and some pipe clients. A pipe is a section of memory that is used for interprocess communication. A named pipe can be described as first in, first out (FIFO); the inputs that enter first will be output first.  
  
A named pipe differs from an anonymous pipe in that it can exist beyond the life of its associated processes and must be explicitly deleted.

* What is the difference between
* **Unnamed PIPE** and **named PIPE**?
* **unnamed pipe**
* handles one way communication. Also called an anonymous pipe (or simply pipe), it is typically used to communicate between a parent process and a child process. Within SAS, the SAS System is the parent process that invokes (and reads data from) a child process.
* **named pipe**
* handles one-way or two-way communication between two unrelated processes. That is, one process is not started by the other. In fact, it is possible to have two applications communicating over a pipe on a network. You can use named pipes within SAS to communicate with other applications or even with another SAS session.
* what are execl,exec,execv and difference between them..??

*execl()* : NULL-terminated argument list

exec(): In computing, **exec** is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable. This act is also referred to as an overlay.

NULL-terminated array of arguments

* what is **semaphore**? Explore sem\_init(), sem\_create(), sem\_destroy(), sem\_unlink(), sem\_wait(), sem\_post().
* what is **mutex**? Difference between semaphore and mutex.

**Process Management**

Note :

1. Read Man pages of the functions/commands mentioned in the hint of each question.
2. You will get the name of header file to be included for that function to work, in the man page of that respective function.

Questions :

1. Parent and child process

Write a C/CPP program to create a child process. Child should print its pid and its parent’s pid and should exit by printing message as “Child Exiting …”. Parent should print its pid and should exit by printing message as “Parent Exiting ..”.

(Hint(functions to be used) : getpid, fork, getppid)

1. Scheduling functions

Write a program in C/CPP to check the scheduling policy used by the process and its priority.

(Hint(functions to be used) : sched\_getscheduler, getpid)

1. Scheduling functions

Write a program in C/CPP to get the current scheduling policy of the process. The program should change the scheduling policy to the other than current one. Program should report errors if it fails to set the new scheduling policy.

(Hint(functions to be used) : sched\_setscheduler, getpid)

1. Scheduling algorithm

Write a program in C/CPP to take process name, its arrival time and execution/burst time as input.

Use FCFS(non-preemptive) algorithm to calculate wait time of each process, average

wait time, turnaround time of each process and average turnaround time.

Calculation of time will start from the arrival time of first process.

execution/burst time - Time required for execution of process

Wait time of process = response time of process - arrival time process

Response time of process : time at which process is scheduled to run

Average wait time = (sum of wait time of each process) / (number of processes)

Turnaround time of process = (finish/completion time of process) - (arrival time of process)

Average turnaround time = (sum of turnaround time of each process) / (number of processes)

Sample Input

|  |  |  |
| --- | --- | --- |
| Process | arrival time | execution/burst time |
| P1 | 0 | 3 |
| P2 | 2 | 5 |
| P3 | 5 | 6 |

Sample Output

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Response time | Completion/finish time | Waiting time | Avg waiting time | Turnaround time | Avg turnaround time |
| P1 | 0 | 3 | 0-0 = 0 | (0+1+3)/3=4/3 | 3-0=3 | (3+6+9)/3=18/3=6 |
| P2 | 3 | 8 | 3-2=1 | 8-2=6 |
| P3 | 8 | 14 | 8-5=3 | 14-5=9 |