Practical-3

Aim: Process Creation and Termination (Use of fork, wait, getpid, and getppid system calls).

Explanation:

Fork System Call:

#include <unistd.h>

pid_t fork(void);

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group.
- The child's parent process ID is the same as the parent's process ID.

On success, the PID of the child process is returned in the parent, and 0 is returned in the child.

On failure, -1 is returned in the parent, no child process is created, and *errno* is set appropriately.

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.

A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.

Task 1:

Call fork once, twice, thrice and print “Hello”. Observe and interpret the outcomes.

Getpid and Getppid System Calls:

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

pid_t getpid(void);
pid_t getppid(void);

getpid() returns the process ID of the calling process.

getppid() returns the process ID of the parent of the calling process.

Task 2:

Print PID and PPID for parent and child processes. Observe and interpret the outcomes.

Wait System Call:

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

pid_t wait(int *_status_);

wait system call is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If a child has already changed state, then this call returns immediately. Otherwise it blocks until either a child changes state or a signal handler interrupts the call.

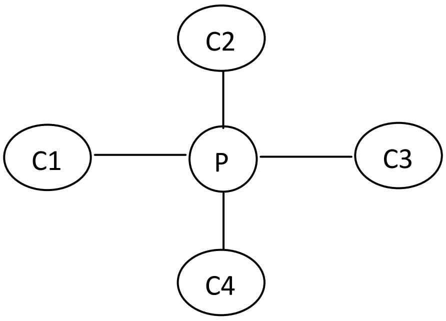
On success, returns the process ID of the terminated child; on error, -1 is returned.

Task 3:

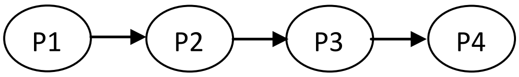
Add wait to the code of task 2. Observe and interpret the outcomes.

Programs:

1. Write a program to implement fan of n processes.



2. Write a program to implement chain of n processes.



Solution Logic :

Fanning :

- The original program should be the main parent process.
- Main parent should call the fork() to create a child.
- Child process should simply print some message with its process ID and exits.
- Parent should continue to create more children, until it reaches the upper value given by user.

Chaining :

- The original program should be the main parent process.
- Main parent should call the fork() to create a child.
- Parent process should simply print some message with its process ID and exits.
- Child should continue to create its child and becomes a parent.
- Again parent should print some message with its process ID and exit, and its child should create another child and continue, until it reaches the upper value given by user.