5 Experiment 5: Process Management using System Calls

5.1 Objective

By the end of this experiment, students will learn about different system commands used to manage processes. They will also gain an understanding of orphan and zombie processes.

5.2 Reading Material[1][3][2]

5.2.1 Process management related system calls

System Call	Description	Programming Syntax
fork()	Creates a new process by duplicating the calling process.	$\text{pid}_t fork(void)$;
exec()	Replaces the current process with a new program.	int exec(const char *pathname, char *const argv[]);
wait()	Causes the parent process to wait until one of its child processes terminates.	$\text{pid}_t wait(int * status)$;
exit()	Terminates the calling process and returns an exit status to the operating system.	void exit(int status);
getpid()	Retrieves the process ID (PID) of the calling process.	$\text{pid}_t getpid(void)$;
getppid()	Retrieves the parent process ID (PPID) of the calling process.	$\text{pid}_t getppid(void)$;
kill()	Sends a signal to a specified process or group of processes.	int kill($\text{pid}_t pid, int sig$);
nice()	Modifies the priority of a process.	int nice(int incr);
sleep()	Causes the calling process to sleep for a specified number of seconds.	unsigned int sleep(unsigned int seconds);

Table 4: Process management related system calls

5.2.2 Orphan and Zombie Processes

Orphan Process: An orphan process is a child process whose parent process has terminated or finished before the child process completes. When the parent process exits or is terminated unexpectedly without properly waiting for the child to finish, the operating system reassigns the orphaned child process to the init process (PID 1 in Linux). The init process adopts and manages orphan processes until they complete execution.

Zombie Process: A zombie process is a terminated process that has completed its execution but still has an entry in the process table. After a process completes, it sends an exit status to its parent process and becomes a zombie waiting for the parent to retrieve the exit status using the `wait()` system call. If the parent fails to fetch the exit status of the terminated child (due to neglect or termination), the zombie process remains in the process table as an inactive process entry.

5.3 Sample Programs

1. A program to create a child process using fork system call.

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

int main() {
    pid_t child_pid;

    // Create a child process
    child_pid = fork();

    if (child_pid == 0) {
        // The child process code section
        printf("Child process: PID = %d\n", getpid());
    } else if (child_pid > 0) {
        // The parent process code section
        printf("Parent process: Child PID = %d\n", child_pid);
    } else {
        // Fork failed
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}
```

2. C program to demonstrates the creation of an orphan process.

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

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        // Child process
        printf("Child process: PID = %d\n", getpid());
        sleep(2); // Sleep to ensure the parent process terminates first
        printf("Child process: My parent's PID = %d\n", getppid());
    } else if (child_pid > 0) {
        // Parent process
        printf("Parent process: PID = %d\n", getpid());
        printf("Parent process: Terminating...\n");
```

```
    } else {
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}
```

3. C program to demonstrate the creation of a Zombie process.

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

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        // Child process
        printf("Child process: PID = %d\n", getpid());
        exit(0); // Child process exits immediately
    } else if (child_pid > 0) {
        // Parent process
        printf("Parent process: PID = %d\n", getpid());
        printf("Parent process: Child PID = %d\n", child_pid);
        sleep(10); // Sleep to allow time for the child to become a zombie
        printf("Parent process: Terminating...\n");
    } else {
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}
```

Video Reference:

https://youtube.com/playlist?list=PLWjmN065fOfGdAZrlP6316HVHh8jlve_

References

[1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts.* Wiley, 10 edition, 2018.

[2] Sumitabha Das. *Unix Concepts And Applications.* Wiley, 4 edition, 2006.

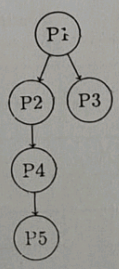
[3] Richard Stones Neil Matthew. *Beginning Linux Programming.* Wiley, 4 edition, 2007.

5.4 Lab Exercises

Exercise 1: Write a C program to illustrate that performing 'n' consecutive fork() system calls generates a total of $2^n - 1$ child processes. The program should prompt the user to input the value of 'n'."

Exercise 2: Write a C program utilizing the `fork()` system call to generate the following process hierarchy: P1 $\rightarrow$ P2 $\rightarrow$ P3. The program should display the Process ID (PID) and Parent Process IDs (PPID) for each process created.

Exercise 3: Write a C program to generate a process hierarchy as follows:



The program should create the specified process structure using the appropriate sequence of 'fork()' system calls. Print PID and PPID of each process.