Shubhan Singh

SE-Comps B/Batch C

2022300118

OS Experiment 2: Processes

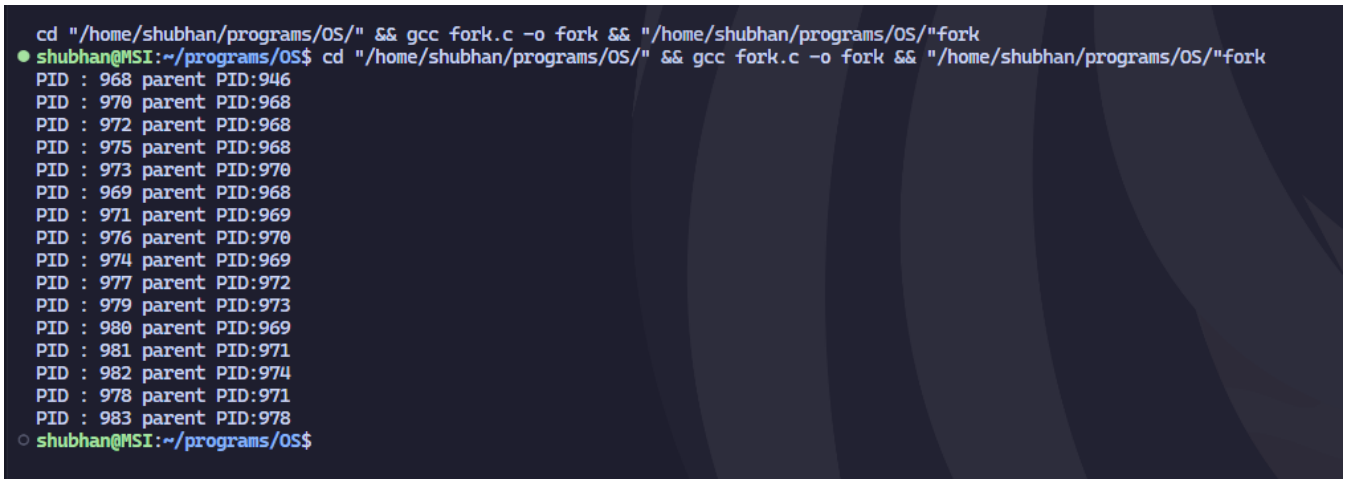
(All source code files are submitted on moodle)

Part 1:

Problem statement: Write a program which creates exactly 16 copies of itself by calling fork () only twice within a loop. The program should also print a tree of the pids.

Files used: *fork.c*

Output:



Part 2:

Aim: Demonstrate the following system calls with examples.

Fork() System call

Wait() system call

Orphan Process

Zombie process

Files used:

fork2.c

wait.c

orphan.c

zombie.c

1. **fork() system call:**

   The fork() system call in C is used 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. We can also 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.
   - If fork() returns a zero, it means we are in the child process.
   - If fork() returns a positive value, this is the process ID of the child process and this is returned to the parent process.

```c
void fork2()
{
    // child process because return value zero
    if (fork() == 0){
        printf("Hello from Child!\n");
        sleep(0.5);
    }

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}

int main()
{
    fork2();
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

cd "/home/shubhan/programs/OS/" && gcc fork2.c -o fork2 && "/home/shubhan/progra
shubhan@MSI:~/programs/OS$ cd "/home/shubhan/programs/OS/" && gcc fork2.c -o for
Hello from Child!
Hello from Parent!
shubhan@MSI:~/programs/OS$
```

Here, the child process receives 0 as the return value for fork.

2. wait() system call:

The wait() system call suspends the calling process until one of its child processes terminates. The wait() system call can return the exit status of the child, which can be extracted using macros defined in <sys/wait.h>.
Here are the possible scenarios for the wait() system call:

- If wait() returns a child's PID, then this child has terminated.
- If wait() returns -1, then no child processes are left, or an error occurred.

The wait() system call also serves to clean up the resources used by the child process.

```c
void waitfunc(){
    int pid = fork();
    if (pid > 0){
        printf("Waiting for child to terminate ... \n");
        wait(NULL);
        printf("Child has terminated.\n");
    }
    else if (pid == 0){
        sleep(0.5);
        printf("In child process.\n");
    }
}
int main(){
    waitfunc();
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

cd "/home/shubhan/programs/OS/" && gcc wait.c -o wait && "/home/shubhan/programs/OS
shubhan@MSI:~/programs/OS$ cd "/home/shubhan/programs/OS/" && gcc wait.c -o wait &&
Waiting for child to terminate...
In child process.
Child has terminated.
shubhan@MSI:~/programs/OS$ cd "/home/shubhan/programs/OS/" && gcc wait.c -o wait &&
Waiting for child to terminate...
In child process.
Child has terminated.
shubhan@MSI:~/programs/OS$
```

Here, the parent waits for the child to terminate, thus the child has terminated message always comes at the end.

3. Orphan process

When a parent process completes execution while its child processes are still running, those child processes become orphan processes. They don't become "zombies," because they're still executing, but they no longer have a parent process to control them.

The init process, automatically adopts orphaned processes. This is done to ensure that resources aren't indefinitely tied up with defunct or completed processes. Once the orphan process finishes executing, init will collect the exit status, thereby preventing the orphan from becoming a zombie process.

```c
int main()
{
    int pid = fork();

    if (pid > 0)
        printf("In parent process,PID: %d\n",getpid());
    else if (pid == 0)
    {
        sleep(2);
        printf("In child process,PID:%d, PPID:%d\n",getpid(),getppid());
    }

    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

cd "/home/shubhan/programs/OS/" && gcc orphan.c -o orphan && "/home/shubhan/programs/OS/"orphan
shubhan@MSI:~/programs/OS$ cd "/home/shubhan/programs/OS/" && gcc orphan.c -o orphan && "/home/shubhan/programs/OS/"orphan
In parent process,PID: 8022
shubhan@MSI:~/programs/OS$ In child process,PID:8023, PPID:398
```

Here, the child process has a different parent process ID

4. **Zombie process**

A zombie process is a process state when the child dies before the parent process. In this case, the kernel still maintains the information about the dead process in the process table, mainly to allow its parent to read its exit status. The term "zombie" is used because such a process is not functional anymore, but its process ID (PID) and process table entry still exist in the system.
Zombie processes can be identified in the output from the Unix "ps" command by the presence of a "Z" in the "STAT" column.
Zombies can be removed from the system in two ways:

- The parent process can read the exit status of the dead child. This is done using the wait() system call, after which the zombie process is removed.
- If the parent process dies before the zombie, the zombie process becomes a child of the init process. The init process periodically executes the wait() system call to remove any zombies that are children of init.

```c
int main(){
    pid_t child_pid = fork();
    if (child_pid > 0)
        sleep(5); // Parent process sleeps for 5 seconds
    else
        exit(1); // Child process exits immediately
    return 0;
}
```

This code creates a zombie process, as the child exits immediately, while the parent waits for 5s.