Practical 06: UNIX / LINUX Processes Control

Name: __________________________________ Roll no: ________________
Date: _________________________ Section: ______________________

Practical Objectives

1. Process creation theory
2. The fork() System Call
3. Execution of parent process without waiting for child to exit
4. Execution of parent process waiting for child to exit
5. Zombie processes
6. Orphan processes

Attach this cover sheet to the front of the packet of materials you submit following the laboratory tasks.

<table>
<thead>
<tr>
<th>Activities</th>
<th>Remarks</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-lab Exercises</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Theory

#### Process Creation
A “parent process” is a process that has created one or more child processes. In UNIX, every process except process 0 (the swapper) is created when another process executes the fork system call. The process that invoked fork is the parent process and the newly created process is the “child process”. Every process (except process 0) has one parent process, but can have many child processes.

The kernel identifies each process by its process identifier (PID). Process 0 is a special process that is created when the system boots; after forking a child process (process 1), process 0 becomes the swapper process. Process 1, known as init, is the ancestor of every other process in the system.

When a child process terminates execution, either by calling the exit system call, causing a fatal execution error, or receiving a terminating signal, an exit status is returned to the operating system. The parent process is informed of its child's termination through a SIGCHLD signal. A parent will typically retrieve its child's exit status by calling the wait system call. However, if a parent does not do so, the child process becomes a “zombie process.”

Following table illustrates various process system calls.

<table>
<thead>
<tr>
<th>General Class</th>
<th>Specific Class</th>
<th>System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Related Calls</td>
<td>Process Creation and Termination</td>
<td>exec(), fork(), wait(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>Process Owner and Group</td>
<td>getuid(), geteuid(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getgid(), getegid()</td>
</tr>
<tr>
<td></td>
<td>Process Identity</td>
<td>getpid(), getppid()</td>
</tr>
<tr>
<td></td>
<td>Process Control</td>
<td>signal(), kill(), alarm()</td>
</tr>
<tr>
<td></td>
<td>Change Working Directory</td>
<td>chdir()</td>
</tr>
</tbody>
</table>

#### The fork() System Call
fork - create a child process

**Synopsis:**
#include <unistd.h>

**Return Value:**
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.

**Description:**
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.
- The termination signal of the child is always SIGCHLD.
- The child does not inherit timers from its parent.
- The child does not inherit semaphore adjustments from its parent.
- The child’s set of pending signals is initially empty.

**Example 1: Parent executes without waiting for child to exit**
In this example, parent does not wait for child to exit.

```c
#include <stdlib.h> // for exit
#include <stdio.h>
#include <string.h>
int main(void) {
    int retPID;
    retPID = fork();
    if (retPID != -1) { /*Process is created*/
        if (retPID == 0) {
            printf("n I am in Child Process\n");
        } else {
            printf("n I am in Parent Process\n");
        }
    } else { // Fork error (fork()==-1)
        printf("n Can not create processes");
        exit(0);
    }
    return(0);
}
```

**Example 2: Parent waits for child**
In this example, parent waits properly for child to exit.

```c
#include <unistd.h>
#include <sys/types.h>
#include <errno.h> // perror
#include <stdio.h>
#include <sys/wait.h> // WEXITSTATUS()
#include <stdlib.h> // for exit
int global = 0;
int main() {
    pid_t child_pid;
    child_pid = fork();
    if (child_pid >= 0) { /* fork succeeded */
        if (child_pid == 0) { /* fork() returns 0 for the child process */
            local++; global++;
            printf("child process!\n");
            printf("child PID = %d, parent pid = %d\n", getpid(), getppid());
            printf("\n child's local = %d, child's global = %d\n", local, global);
        } else {
            printf("parent process!\n");
            printf("parent PID = %d, child's pid = %d\n", getpid(), child_pid);
            exit(0);
        }
    }
}```
Orphan and Zombie Process State

Zombie Processes
On Unix and Unix-like computer operating systems, a "zombie process" or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie an undead person. When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other 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, whereupon the zombie is removed.

Example 3: Demonstration of Zombie State
/* Program zombie.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int main() {
  int childpid;

  exit(child_exit_code); // child exit code
}
else { /* parent process */
  printf("parent process!\n");
  printf("parent PID = %d, child pid = %d\n", getpid(), child.pid);
  wait(&child_exit_code); /* wait for child's exit, & store child's exit code */
  printf("Child exit code: %d\n", WEXITSTATUS(child_exit_code));
  //Change in child’s local/global should not reflect in parent process.
  printf("\nParent'z local = %d, parent's global = %d\n",local,global);
  printf("Parent says bye!\n");
  exit(0); /* parent exits */
}
else { /* failure of fork */
  perror("fork");
  exit(0); //parent exit
}

Orphan and Zombie Process State

Zombie Processes
On Unix and Unix-like computer operating systems, a 'zombie process' or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie an undead person. When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other 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, whereupon the zombie is removed.

Example 3: Demonstration of Zombie State
/* Program zombie.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
int main() {
  int childpid;

  exit(child_exit_code); // child exit code
}
else { /* parent process */
  printf("parent process!\n");
  printf("parent PID = %d, child pid = %d\n", getpid(), child.pid);
  wait(&child_exit_code); /* wait for child's exit, & store child's exit code */
  printf("Child exit code: %d\n", WEXITSTATUS(child_exit_code));
  //Change in child's local/global should not reflect in parent process.
  printf("\nParent'z local = %d, parent's global = %d\n",local,global);
  printf("Parent says bye!\n");
  exit(0); /* parent exits */
}
else { /* failure of fork */
  perror("fork");
  exit(0); //parent exit
}
/* Create a child process */
childpid = fork();
if(childpid > 0 ) {
    /* Parent Process */
    /* Sleep Process for 60 Seconds */
    /* Parent is not waiting for child to terminate */
sleep(60);
} else {
    /*childpid==0*/
    /* Child Process */
    exit(0);
}
return 0;

Compile and Run C program
$ gcc zombie.c –o zombie.o
$ ./zombie.out &
List the processes
$ ps -l
OR
$ ps -o stat,pid,ppid,cmd

Note: The Process with status as Z or marked as <defunct> i.e., de-functioning is ZOMBIE Process.

Orphan Processes
An "orphan process" is a computer process whose parent process has finished or terminated, though it remains running itself. 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. Even though technically the process has the 'init' process as its parent, it is still called an orphan process since the process that originally created it no longer exists. A process can be orphaned unintentionally, such as when the parent process terminates or crashes.

Example 4: DEMONSTRATION OF ORPHAN PROCESSES
/* Program orphan.c */
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
int main()
{
    int child_pid;
    child_pid = fork();
    if(child_pid>0){
        /* Parent Process waiting for child to terminate */
        wait(NULL);
        while(1);
    } else {
        /* Child Process, executing forever */
        while(1);
    }
    return 0;
}

Above example demonstrates how to `wait()` if you don’t care what the return value of the child is: you just call it with NULL as the argument.

Compile and Run C program
$ gcc orphan.c –o orphan.o
$ ./orphan.o &
List the processes
$ ps -o stat,pid,ppid,cmd

Find the Parent and Child Process (For this look at the PPID Column)
Kill the Parent Process, but the child is still executing, as parent had been killed the child process becomes the ORPHAN Process.
Note: Look at the PPID of child. It is 1 means; it is the Process ID of the Kernel.
The child is get adopted by the kernel.
Exercises

1. Modify Example 2 to use switch case.
2. Compile Example 3 using GNU C compiler, and note the observations.
3. Compile Example 4 using Eclipse IDE, and note the observations.
   \colong\faculty\CS\arahman\OS-CS311-FA16\eclipse-cpp-helios-SR2-linux-gtk-x86_64.tar.gz
   (or)
4. Declare an integer array of roll numbers starting from 3 numbers below your roll number and ending after 3 numbers after your roll number. (e.g., if your roll number is 14b-025-bs, you need to declare an array consisting of elements: [22, 23, 24, 25, 26, 27, 28]). Arrange elements of your array in random order [28, 22, 25, 24, 27, 26, 23]. Now modify Example 2, to sort elements of this array in the child process in descending order. Similarly, in parent process, you need to sort it in ascending order.

   **Sample declaration:**
   Array: [28, 22, 25, 24, 27, 26, 23]

   **Sample output:**
   CHILD process: sorted array elements: [28, 27, 26, 25, 24, 23, 22]
   PARENT process: sorted array elements: [22, 23, 24, 25, 26, 27, 28]

All students need to create RollNoLabNo folder in home directory and create exercise / task script files using following convention:
rollno-task01.c (source code), rollno-task01.o (binary file)
14b001bs-ex01.c, 14b001bs-ex02.c, … 14b001bs-ex11.c
14b001bs-task01.c, 14b001bs-task02.c, … 14b001bs-task11.c

**Note:**
1. Submit the screenshot of your commands, source code, and results with your lab observations in your own words.
2. Email all lab results / observations document and source code in zip format to the following email address:
   alphasecure@gmail.com
   a. Subject of Email: LabNoXX_Name_RollNo_Section
   Attachment name: LabNoXX_Name_RollNo_Section.zip
3. Email should be send during the lab timings on same day. Delayed Emails will not be considered for assessment.
4. Hard copies of Lab manual & observation should be provided during the lab timings on same day. Delayed print copies will not be considered for assessment.