COEN 177: Operating Systems
Lab 2: Programming in C and use of Systems Calls

Objectives

1. To develop sample C programs
2. To develop programs with two or more processes using fork( ), exit( ), wait( ), and exec( ) system calls
3. To demonstrate the use of lightweight processes - threads

Guidelines

For COEN 177L, you must develop a good knowledge of C in a Linux development environment. You are highly encouraged to use command line tools in developing, compiling, and running your programs.

Please pay attention to your coding style and good programming practices; if your program is not worth documenting, it will not be worth running.

Skills in developing multi-processing and multi-threading applications are required to create parallelism.
A process is defined as a program in execution. Multiple processes can execute in the same program, with each process owning its copy of the program within its own address space and executing it independently of the other processes.

A fork() system call is used in Linux to create a child process. The fork() takes no arguments, returns 0 for the child process, and returns the process ID (PID) for the parent process. After a new child process is created, the parent and the child processes execute the next instruction following the fork() system call. Therefore, the returned value of the fork() is used to distinguish between the parent and child processes.

- If fork() returns a negative value, the system call has failed to execute in the kernel
- If fork() returns a zero, a newly created child process
- If fork() returns a positive value, a process ID of the child process to the parent.

Include the following libraries for defining the process pid_t type and using fork().

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

Note that the child process inherits an exact copy of the parent's address space, and both have separate address spaces. There are two types of processes:

Zombie Process: A process is Zombie, i.e., inactive, as long as it maintains its entry in the parent's process table. Therefore, it is recommended that a child process uses exit() system call at the end of its execution. The exit status of the child process is then passed to the parent process to remove its entry.

Orphan Process: A process is Orphan if its parent process no more exists. This is either because the parent process is either finished or terminated without waiting for its child process to terminate. Therefore, it is recommended that a parent process uses wait() system call to wait until the child process exits, i.e., terminates.

In contrast, a thread is a single sequence stream within a process and is often referred to as a lightweight process. Threads operate faster than processes during creation and termination, context switching, and communication. Threads are not independent like processes; they share their code and data and open file descriptors with other threads. Threads, however, still maintain their own program counters, registers, and stack.
Include the pthread.h library and use the function pthread_create () instead of fork().

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                    void *(*start_routine) (void *arg), void *arg);
```

C Program with two or more processes

Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment

Step 1. [10 points] Write the following C program in a Linux environment using vi.

```
/* C program to demonstrate the use of fork()*/
#include <stdio.h>      /* printf, stderr */
#include <sys/types.h>  /* pid_t */
#include <unistd.h>     /* fork */
#include <stdlib.h>     /* atoi */
#include <errno.h>      /* errno */
/* main function */
int main() {
   pid_t  pid;
   int i, n = 3000; // n is a delay in microseconds inserted in parent and child iterations
   printf("\n Before forking.\n");
   pid = fork();
   if (pid < 0) {
      fprintf(stderr, "can't fork, error %d\n", errno);
          exit(0);
   }
   if (pid){
      // Parent process: pid is > 0
      for (i=0;i<10;i++) {
         printf("\t \t \t I am the parent Process displaying iteration %d \n",i);
         usleep(n);
      }
   }
   else{
      // Child process: pid = 0
      for (i=0;i<10;i++) {
         printf("I am the child process displaying iteration %d\n",i);
         usleep(n);
      }
   }
   return 0;
}
```

Step 2. [10 points] Compile the program using gcc compiler by typing gcc YourProgram.c – o ExecutableName. When it compiles without errors or warnings, run the program by typing ./ExecutableName and take note of your observations.

Step 3. [10 points] Rewrite the program in Step 1 so that the delay is determined by the user entered as a command line argument. In this case, the main function should be replaced by

```
/* main function with command-line arguments to pass */
int main(int argc, char *argv[]) {
```

The value of n then is taken as

```
n = atoi(argv[1]); // n microseconds is taken as command-line argument
```

Compile and run the program by typing ./ExecutableName 3000.

You may consider making sure that the user enters a delay in the command line using:

```
if (argc != 2){
        printf ("Usage: %s <delay> \n",argv[0]);
        exit(0);
 }
```

Step 4. [10 points] Note that the delay in the loop depends on the command line argument you give; here, the delay is 3000 microseconds. What happens when you change the delay to 500 and 5000?

Step 5. [10 points] Re-write the program in Step 2 to create 4 processes so that each process iterates with a different delay taken as a command-line argument. In this case, you will have 4 command-line arguments. When your program compiles without errors or warnings, upload the source file to Camino.

Step 6. [15 points] Write a C program to create 5 processes (including the initial program itself – parent) so that each process iterates with a different delay taken as a command-line argument. In this case, you will have 5 command-line arguments. When your program compiles without errors or warnings, upload the source file to Camino.

Changing the context of the process

Processes are often created to run separate programs. In this case, a process context is changed by causing it to replace its execution image with an execution image of a new program, exec( ) system call is used. Although the process loses its code space, data space, stack, and heap, it retains its process ID, parent, child processes, and open file descriptors. Six versions of exec( ) exist. The simplest and widely used:

- execlp(char *filename, char *arg0, char arg1,..... , char *argn, (char *) 0);
- e.g. execlp(“sort”, “sort”, “-n”, “foo”, 0);

Step 7. [20 points] Rewrite the program in Step 1, so that the child process runs the ls command and the parent process waits until the child process terminates before it exits. You may use the following code snippet.

```
else if(pid == 0)
   {
      execlp("/bin/ls", "ls", 0);
   }
else
   {
      wait(0);
      printf("Child has completed the task! \n");
      exit(0);
   }
```

C Program with two threads

Step 8. [15 points] Rewrite the program in Step 3 with two threads instead of two processes. When your program compiles without errors or warnings, upload the source file to Camino.

Requirements to complete the lab

1. Demo to the TA the correct execution of your C programs. Consider creating a Makefile and a shell script to automate the compilation and run of all your programs as an easy way to demo to the TA.
2. Upload your observations to Camino
3. Upload your source code for all your programs as .c file(s) to Camino.

Please start each program/ text with a descriptive block that includes the following information minimally:

```
# Name: <your name>
# Date: <date> (the day you have lab)
# Title: Lab2 – step.. task
# Description: This program computes … <you should
# complete an appropriate description here.>
```