COEN 177: Operating Systems
Lab assignment 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 light weight processes - threads

Guidelines

For COEN 177L you need to 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 is not worth running.

Multiprocessing and multithreading applications is a way to create parallelism. A process is defined as a program in execution. Multiple processes can be executing the same program, with each process owning its own copy of the program within its own address space and executes it independently of the other processes.

A fork() system call is used in Linux to create a child process. The fork() call takes no arguments, returns 0 to the child process, and returns the child's process ID (PID) to the parent process.
After the child process is created, the parent and child processes *both* resume execution at the next instruction following the fork() system call. Therefore, the returned value of the fork() is critical to distinguish the parent from the child.

- If fork() returns a negative value, the system call has failed and you are still the only process
- If fork() returns a zero, you are a newly created child process
- If fork() returns a positive value, you are the parent process with a child having that process ID.

Include the following libraries for definitions of the pid_t type and the fork() system call.

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

Note that the child process inherits an exact copy of the parent's address space, but both address spaces are separate from each other.

Additionally, there are two special types of processes:

Zombie Process: A process is a zombie process if it retains its entry in the parent's process table despite having completed its execution. To avoid this, it is recommended that a child process use the exit( ) system call at the end of its execution, which passes its exit status to its parent, who can then remove the process table entry.
Orphan Process: A process is an orphan process if it is still executing when its parent process no longer exists. This is typically because the parent process has finished or been terminated without waiting for its child process to terminate. To avoid this, it is recommended that parent processes use the wait() system call as many times as it has children. This call waits until a child exits if any exist, or returns immediately if there are no children remaining.

A thread is a single sequence stream within a process. These are often used as a lightweight child process. Using threads is faster than using child processes for creation and termination, context switching, and communication. Threads are not independent like processes, however, as they share with other threads their code, data and open file descriptors. Threads, however, still maintain their own program counters, registers, and stack. To use threads, you must 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);
```

Note that pthread_create() takes a function pointer argument (for the new thread to run), and that function must take a single void pointer as an argument and return a single void pointer. If you must pass other types of arguments, e.g. integers, typecast them with (void*)(size_t) or malloc a struct to hold them and typecast the struct pointer with (void*). Additionally, you will need to keep track of the thread ID (written to the address passed as the first argument) so that you can call pthread_join() on it later, lest the process end before the child thread

completes.

C Program with two processes

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

Step 1. Please write the following C program in a Linux environment using vi, emacs, or an editor of your choice

```
/*Sample C program for Lab assignment 1*/
#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 with command-line arguments to pass */
int main(int argc, char *argv[]) {
   pid_t  pid;
   int i, n = atoi(argv[1]); // n microseconds to input from keyboard for delay
   printf("\n Before forking.\n");
   pid = fork();
   if (pid == -1) {
      fprintf(stderr, "can't fork, error %d\n", errno);
   }
   if (pid){
      // Parent process
      for (i=0;i<100;i++) {
         printf("\t \t \t Parent Process %d \n",i);
         usleep(n);
      }
   }
   else{
      // Child process
      for (i=0;i<100;i++) {
         printf("Child process %d\n",i);
         usleep(n);
      }
   }
   return 0;
}
```

Step 2. Compile the program using the gcc compiler by typing gcc YourProgram.c -o ExecutableName. When it compiles without errors or warnings, proceed to step 3.

Step 3. Attempt to run the program by typing ./ExecutableName and note down what happens.

Step 4. Rerun the program by typing ./ExecutableName 3000. Note that the delay in the loop depends on the command line argument you give, interpreted in microseconds.

a. Enter delays of 500 and 5000, what happens?

Step 5. Take-Home Programming Task (attempt this on your own and be prepared to demo your solution): Write a program that will result in the creation of exactly seven processes, one of which is the original parent process, while following the rule that every process with children must have exactly two children. All 7 processes must each print its own PID, the PID of its parent, and the PIDs of all its children, as returned from fork() or wait(0). Use getpid() and getppid() to get process IDs as appropriate.

C Program with two threads

Step 6. Rewrite the program in Step 1 with two threads instead of two processes. When it compiles without errors or warnings, proceed to Step 7.

Changing the context of the process

Processes are often created to run separate programs. In this case, to change the context of the process by replacing its execution image with an execution image of a new program, the exec( ) system call is used. Although

the process will lose its code space, data space, stack, and heap, it retains its process ID, parent and child processes, and open file descriptors. Six versions of exec( ) exist, but the simplest and most widely used follows:

- execlp(char *filename, char *arg0, char *arg1,..... , char *argn, (char *) NULL);

- e.g. execlp(“sort”, “sort”, “-n”, “foo”, 0); ◊ $ sort -n, foo

Step 7. Write a program in which the parent process spawns a child which runs the ls command. Make sure that the parent process waits until the child process terminates before it exits. When everything compiles and runs without errors or warnings, demonstrate to the TA. Use the following code snippet.

```
if(pid == 0)
{
        execlp("/bin/ls", "ls", NULL);
}
else
{
        wait(NULL);
        printf("Child Complete");
        exit(0);
}
```

Requirements to complete the lab

1. Show the TA correct execution of steps 3 through 7.
2. Submit your answers to questions, observations, and notes as .txt file and upload to Camino
3. Submit the source code for all your programs as .c file(s) and upload to Camino.

Be sure to retain copies of your .c and .txt files. You will want these for study purposes and to resolve any grading questions (should they arise)

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

```
//Name:
//Date:
//Title: Lab2 -
//Description:
```