Computer Operating System Experiment

Laboratory 1

Process

Objective:

Learn to work with Linux system calls related to process creation and control.

Including:

- process create and data sharing between parent and child
- the execution order of parent and child process
- Create the specified num of child processes
- Process termination
- Zombie process
- process create a child process and load a new program

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1 Linux Processes

In this lab, we start experimenting with a few Linux system calls. The first one, we will look at is fork, which is used by a process to spawn an identical copy of itself. When we learn to use a system call or a library function, it is helpful to follow the simple workflow described as follows.

Start by reading the man page of the system call or library function in which you are interested. This will help you begin to understand how it works, but it will also show you some practical details that are essential to using it successfully. In the man page, pay close attention to the SYNOPSIS; it will tell you:

- The files you must #include in your program.

- The function prototype(s) with which you will work.

For instance, if we're dealing with fork, you'll see something like:

```
FORK(2)                 Linux Programmer's Manual FORK(2)

NAME
fork - create a child process

SYNOPSIS
#include <sys/types.h>

pid_t fork(void);

DESCRIPTION
fork() creates a new process by duplicating the calling process.
The new process is referred to as the child process.
The calling process is referred to as the parent process.
...
```

From this we learn that any program calling fork will need to #include the file unistd.h. The "angle brackets" indicate that these files reside in an include directory owned by the system (most often /usr/include).

We also learn that the fork call:

- Returns a value of type pid_t (essentially, an integer), and
- Does not take any input parameters, what is indicated by the formal parameter void.

Once we have tried our best to understand that information, we should not be so bold as to throw code into a large program to see how things work out. It is often more productive to write a small program just to test that we have the right understanding about the behavior of the function. Once we have experimented a bit with this program and are convinced that the function does what we expect and that we have learned to use it effectively, we can use it in a larger context.

2 Debug Tools

to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a SIGTRAP signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to sleep in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the **ps** program to get its process ID.

If you want to follow the child process instead of the parent process, use the command set follow-fork-mode.

set follow-fork-mode mode

Set the debugger response to a program call of fork or vfork. A call to fork or vfork creates a new process. The mode argument can be:

parent

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

child

The new process is debugged after a fork. The parent process runs unimpeded.

show follow-fork-mode

Display the current debugger response to a fork or vfork call.

On Linux, if you want to debug both the parent and child processes, use the command set detach-on-fork.

set detach-on-fork mode

Tells gdb whether to detach one of the processes after a fork, or retain debugger control over them both.

on

The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

off

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of follow-fork-mode) is debugged as usual, while the other is held suspended.

3 Experiments

3.1 Experiment 1: process creation

Here is a first experiment with fork() aimed at understanding what a child process inherits from a parent.

```
#include     <sys/types.h> // need this for fork
#include     <stdio.h> // need this for printf and fflush

int i = 10;
double x =3.14159;
int pid;

int main (int argc, char* argv[]) {

   int j = 2;
   double y = 0.12345;

   if (pid = fork()) {
      // parent code
      printf("parent process -- pid= %d\n", getpid()); fflush(stdout);
      printf("parent sees: i= %d, x= %lf\n", i, x); fflush(stdout);
      printf("parent sees: j= %d, y= %lf\n", j, y); fflush(stdout);
   } else {
      // child code
      printf("child process -- pid= %d\n", getpid()); fflush(stdout);
      printf("child sees: i= %d, x= %lf\n", i, x); fflush(stdout);
      printf("child sees: j= %d, y= %lf\n", j, y); fflush(stdout);
   }

   return(0);
}
```

This code is provided to you in file fork-ex.c. Looking at this code, you may be inclined to think that you can infer the order of execution of these lines of C code. For instance: you might say that that parent executes first and the child executes next; or you might say that the order of execution is the one in which the program was written.

Don't make the mistake of thinking that you can predict the order of execution of the actions in your processes! The process scheduler in the kernel will determine what executes when and your code should not rely on any assumptions of order of execution.

Question:

1) If you change the values of variable x ,y and i in parent process, do the variable in the

child process will affected? Please give the reason.

2) Please modify the fork-ex.c, and create a Makefile that builds all the programs you created. Test your expectation.

3.2 Experiment 2: the execution order of parent and child process

Let's start slowly by investigating what a child process may be inheriting from its parent process. First, let's get this code to compile!

Take a look at the program given to you in file fork.c . Compile and execute the program. Add code to have both the child and the parent print out the value of the pid returned by the fork() system call.

```
Int num = 10;

void    ChildProcess(void);          /* child process prototype    */
void    ParentProcess(void);         /* parent process prototype */

void    main(void)
{
        pid_t    pid;
        pid = fork();
        if(pid == -1) {
            printf("something went wrong in fork");
            exit(-1);
        }else if (pid == 0)
                ChildProcess();
        else
                ParentProcess();
}
void    ChildProcess(void)
{
        int      i;
        int      mypid;
        mypid = getpid();
        for (i = 1; i <= num; i++)
                printf("      This line is from child, value = %d\n", i); fflush(stdout);
        printf("      *** Child process %d is done ***\n",mypid);
        exit(0);
}
void    ParentProcess(void)
{
        int      i,status;
        int got_pid,mypid;
        mypid = getpid();
        for (i = 1; i <= num; i++)
                printf("This line is from parent, value = %d\n", i); fflush(stdout);
        printf("*** Parent %d is done ***\n",mypid);
        got_pid = wait(&status);
        printf("[%d] bye %d (%d)\n", mypid, got_pid, status);

}
```

Question:

1. The global variable num is declared before the call to fork() as shown in this program. After the call to fork(), when a new process is spawned, does there exist only one instance of num

in the memory space of the parent process shared by the two processes or do there exist two instances: one in the memory space of the parent and one in the memory space of the child?

2. Can you infer the order of execution of these lines? Please try to decrease the num, If the value of num is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

3.3 Experiment 3: Create the specified num of child processes

you need to write a function that forks N number of child processes. For example:

void forkChildren (int nChildren)

in main function the forkChildren will be called, and each Children output their pid and parent pid.

You are expecting the following output:

I'm a child: 1 PID: xxx, my parent pid : …
I'm a child: 2 PID: xxxx, my parent pid :…
I'm a child: 3 PID: xxxx, my parent pid :…

Question:
1. please observe the pid and can you tell the policy about pid allocation? Can you determine the parent process and child process from the pid?

3.4 Experiment 4: Process termination

Now, let's experiment with forcing a specific order of termination of the processes. As given to you, the code for this problem makes no guarantee that the child will terminate before the parent does! With the concepts we have covered so far in class, we can use a very basic mechanism to establish order in process creation (with fork) and in process termination (with wait or waitpid).

Copy fork_ex.c to file fork-wait.c and modify it so that you can guarantee that the parent process will always terminate after the child process has terminated. Your solution cannot rely on the termination condition of the for loops or on the use of sleep. The right way to handle this is using a syscall such as wait or waitpid – read their man pages before jumping into this task. One more thing: Modify the child process so that it makes calls to getpid(2) and getppid(2) and prints out the values returned by these calls.

3.5 Experiment 5: Zombie process

When a process is created in Linux using fork() system call, the address space of the Parent process is replicated. If the parent process calls wait() system call, then the execution of parent is suspended until the child is terminated. If a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state". This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped". So when the child process has "died" but has not yet been "reaped", we call the process as zombie process.

Create a zombie process and use ps command to get the status of the process.

Question:

1. can you use kill command to kill the zombie process? If not, how can you reap the zombie process?

3.6 Experiment 6: create a child process and load a new program

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. This project can be completed on any Linux, LINUX, or Mac OS X system.

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt os> and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the LINUX cat command.)

os> cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.10. However, LINUX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

os> cat prog.c &

the parent and child processes will run concurrently.

```
#include <stdio.h>
#include <unistd.h>
#define MAX LINE 80 /* The maximum length command */
int main(void)
{
   char *args[MAX LINE/2 + 1]; /* command line arguments */
   int should run = 1; /* flag to determine when to exit program */
   while (should run) {
   printf("os>");
   fflush(stdout);
      /**
      * After reading user input, the steps are:
      * (1) fork a child process using fork()
      * (2) the child process will invoke execvp()
      * (3) if command included &, parent will invoke wait()
      */
}
   return 0;
}
```

1) **Creating a Child Process**

The first task is to modify the main() function so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (args in Figure 3.36). For example, if the user enters the command ps -ael at the os> prompt, the values stored in the args array are:

args[0] = "ps"
args[1] = "-ael"
args[2] = NULL

you can use read() function to read up to count bytes from standard I/O streams into the buffer starting at buf.

ssize_t read(STDIN_FILENO, void * buf, size_t count);

you can use strtok() function to delimit input character strings

char *strtok(char *str, const char *delim)

- str – The contents of this string are modified and broken into smaller strings

(tokens).

- delim – This is the C string containing the delimiters. These may vary from one call to another.

This args array will be passed to the execvp() function, which has the following prototype:

execvp(char *command, char *params[ ]);

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.