CIS 452 Lab: Week #2
Process Management Concepts

Overview

The purpose of this lab is to become familiar with the mechanisms used by operating systems for process creation, execution management, blocking and termination. Specifically (for UNIX / Linux systems), it introduces and experiments with the `fork()`, `exec()`, `wait()`, and `exit()` system calls.

Hand-in:

- A word document containing the answer to the numbered questions. (The questions should also be included either in bold or italicized).
- Program source code for the lab programming assignment (no zip files and in a separate file for each source file)
- Any requested screenshots embedded in the word document.

Process Creation

All processes in UNIX are created, or spawned, from existing processes via the **`fork()`** system call. Both processes (parent and child) continue execution after the call. Review the textbook, the man pages, or a reference to understand what the **`fork()`** call does and how it operates. Familiarize yourself with the **`ps`** (Process Status) utility and its various options - it provides a great deal of useful information about a process. Review the command-line mechanism (**`&`**) for inducing background execution of a process and the **`sleep()`** function for temporarily suspending execution of a process.

The two following sample programs illustrate the operation of the `fork()` system call.

sampleProgramOne

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

```
int main()
{
    printf("Before fork\n");
    fork();
    printf("After fork\n");
    return 0;
}
```

Start up a Terminal session, which provides you with the UNIX command line interface.
Perform the following operations and answer the questions:

1. Compile and run Sample Program 1, how many lines are printed by the program? (2pts)
2. Describe what is happening to produce the output observed (4 pts)
3. Insert a 10-second call to the function **`sleep()`** after the fork in Sample Program 1 and recompile. Run Program 1 in the background (use **`&`**). Consult the man pages for the `ps` (process status) utility; they will help you determine how to display and interpret the various types of information that is reported. Look especially for "verbose mode" or "long format". Then, using the appropriate options, observe and report the PIDs and the status (i.e., execution state info) of your executing program. Provide a brief explanation of your observations. (4 pts)

sampleProgramTwo

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

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

    if (argc < 2) {
        fprintf(stderr, "Usage: must supply a limit value\n");
        exit(1);
    }
    limit = atoi(argv[1]);

    fork();
    fork();
   printf ("PID#: %d\n", getpid());
     for (i=0; i<limit; i++)
        printf("%d\n", i);
    return 0;
}
```

Perform the following operations and answer the questions:

Study the code for sampleProgramTwo until you understand it.

4. Create a diagram illustrating how sampleProgramTwo executes (i.e., give a process hierarchy diagram (see figure 3.8 from the textbook)). Run the program several times with small input values (e.g., 2...5) to help you understand exactly what is happening (5pts).
5. In the context of process state, process operations, and especially *process scheduling*, describe what you observed and try to explain what is happening to produce the observed results (5 pts).

Process Suspension and Termination

This section introduces the **wait()** system call and the **exit()** function, which are usually related as in parent and child. Note that there are several different versions of wait() (e.g., some specify who to wait for). The exit() function causes program termination; resources are recovered, files are closed, resource usage statistics are recorded, and the parent is notified via a signal (provided it has executed a wait). Refer to the man pages to learn the syntax for using these functions.

sampleProgramThree

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>


int main()
{

    // use these variables

    pid_t pid, child;
    int status;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Fork failure: %s", strerror(errno));
```

```
        exit(1);
    }
    else if (pid == 0) {
        printf("I am child PID %ld\n", (long) getpid());
        /* insert an appropriate form of the exit() function here */

    }
    else {
        /* insert an appropriate form of the wait() system call here */

        printf("Child PID %ld terminated with return status %d\n", (long)
child, status);
    }
    return 0;
}
```

Perform the following operations and answer the questions:

Add function calls to sample program 3 so that it correctly uses wait and exit (i.e., replace the comments with code).

6. Provide the exact line of code that you inserted for the `wait()` system call (2pts)
7. Which prints first, the child or the parent? Why? Describe the interaction between the `exit()` function and the `wait()` system call. You may want to experiment by changing the value to better understand the interaction (6 pts).

Process Execution

The **`exec()`** family of system calls provides a means of specifying that a process (typically the just-spawned child) should be overlayed with a new executable. The child will then execute different code from its parent. There are several different forms of the `exec()` system call. Those with a 'v' (e.g. `execve()`) require a vector of pointers, whereas those with an 'l' (e.g. `execle()`) expect a list of pointers. Those with an 'e' allow you to specify an environmental variable, those with a 'p' allow you to specify a pathname to the executable. The following sample program shows one form of the `exec()` call. It is used in this program to execute any command (e.g., "`ls`", "`ps`") that is issued by the user.

sampelProgramFour

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

```
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc < 3) {
        fputs("Usage: must supply a command\n", stderr);
        exit(1);
    }

    puts("Before the exec");
    if (execvp(argv[1], &argv[1]) < 0) {
        perror("exec failed");
        exit(1);
    }
    puts("After the exec");

    return 0;
}
```

Perform the following operations and answer the questions:

Compile, run and test sampleProgramFour using various commands (e.g., "`date`", "`ls`")

8. When is the second print line ("`After the exec`") printed? Explain your answer. (3pts)
9. Explain how the second argument passed to **`execvp()`** is used (3pts)?

Lab Programming Assignment (Simple Shell) (25 pts)

At this point you should have a good understanding of the relationship between the fundamental system calls and library functions involved in process management on a UNIX system. You should now be able to write your own simple command interpreter, or shell. To do so, you will need to combine all of the ideas (creation, suspension, execution, termination) covered in class and in the lab. A good starting point is to modify the code in sampleProgramFour.

Your program should:

- Display a prompt to the user
- Receive and parse the user input
    - Read a line into a character array (preferably using **`fgets()`**)
    - Depending on your implementation:

You can tokenize the line using your own custom function
or by using **`strtok()`**

```
word_1 = strtok (line, " ");
word_2 = strtok (NULL, " ");
```

or by using **`strsep()`**

```
word_1 = strsep (&lineptr, " ");
word_2 = strsep (&lineptr, " ");
```

- Spawn a child process to execute the command
  - Preferably use **`execvp()/execve()`** as in Sample Program 4 of this lab. Note: you must create and pass a *vector* of pointers to arguments.
  - *or* use **`execlp()/execle()`** as in the example call below. Note: in this case you must pass a fixed *list* of arguments.

```
execlp ("prog_name", "prog_name", ARG, (char *) 0);
```

- Find and use the appropriate system call to collect *resource usage* statistics about each executed process
  - Output the "user CPU time used" for *each* individual child process spawned by the shell
  - Output the number of "involuntary context switches" experienced by *each* individual child process spawned by the shell
- Repeat until the user enters "`quit`"

Submit your code and a screenshot of your sample output.