Laboratory Assignment
Process Management Concepts

It is highly recommended that you work in a group of two students.

Introduction

Unix provides a number of system calls related to process creation, termination, and interaction. In this laboratory exercise, you will experiment with these system calls and observe their behavior.

Objectives

Upon the completion of this lab exercise, students will be able to

- List various states in a Unix process lifecycle
- Use Unix system calls for process management: `fork()`, `exec*()`, `wait*()`, and `exit()`
- Use C compiler directives to control conditional compilation

The asterisk (*) indicates that the system call has a number of variants.

Process Management

In Unix, process creation is handled by the `fork()` system calls. The caller of `fork()` is the parent process and the new process is the child. Both processes continue to execute **independently**; *no data are shared among these two processes*.

To help you understand this lab assignment better, please do the following:

- Review your classnotes, textbook, and the man page(s) to understand what for **fork()** does and how it operates.
- Familiarize yourself with the **ps** Unix command and its various options.
- Review the Unix command-line mechanism (&) to initiate a background execution of a process
- Understand the **sleep** function for temporarily suspending execution of a process

PROCESS CREATION

The following sample program illustrates the operation of the `fork()` system call

```
/* Sample 1 */
#include <stdio.h>
#include <unistd.h>

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

1. Compile and run the above program. How many line are printed by the program?
2. Describe what is happening to produce your answer to the above question

Insert a sufficiently long call to **sleep()** *after* the fork() call (long enough for you to do step 3 below). Recompile and run the program in background (&)

3. Use the **ps** utility with *appropriate options* (yes, you may have to read the man page again), observe and report the PIDs and the status of your processes. Provide a brief explanation.

```
/* Sample 2*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    int k, limit;
    if (argc < 2) {
        fputs ("Usage: must supply a limit value\n", stderr);
        exit(1);
    }
    limit = atoi (argv[1]); /* ascii to integer */
    fork();
    fork();
    printf ("PID: %d\n", getpid());
    for (k = 0; k < limit; k++)
        printf ("%d\n", k);
    return 0;
}
```

atoi() is similar to Java's `Integer.parseInt()`.

```
int z;

z = atoi ("263"); /* z will hold two hundred sixty three */
```

Study the above code (sample2) until you understand what it is doing.

4. Create a diagram showing how you would expect this program to execute (i.e. show all processes)

Run the program *several times* with a small limit value (<10) and observe the **order** of how lines are printed in the output. Rerun with larger input values (e.g. 10, 100, 1000, 10000). Note: you may find it useful to pipe the output to the wordcount (**wc**) utility.

```
    ./samp2 | wc
```

Warning: **DO NOT** redirect your stdout to a file. The file will NOT capture the correct output

```
    ./samp2 > myoutput.txt      # output in myoutput.txt
will be corrupted
```

5. In the context of our classroom discussion on process state, process operations, process scheduling, etc., describe what you observed and try explain what is happening to produce the observed results, especially the difference noted when using a small vs. a large limit value. Note: your results will depend, to some extent, on the machine you are using. This is primarily an experiment; look for apparent anomalies and try to explain them.

PROCESS TERMINATION

A process terminates its execution when the system call `exit()` is invoked. To check the (termination) status of its child processes, a parent process has to call `wait()` or `waitpid()`. The "status" parameter of these two system calls encodes the child exit status (the number passed to the `exit()` system call in the child process).

```
/* Sample 3 */
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main () {
    pid_t pid, child;
    int status;

    pid = fork ();
    if (pid < 0) {
        perror ("fork failure");
        exit (1);
    }
    else if (pid == 0) {        /* code executed in child */
        printf ("I am child PID %ld\n", (long) getpid ());
        /* insert an appropriate form of the exit() call here */
    }
    else {
        /* insert an appropriate form of wait() call here */
        printf ("I am parent PID %d\n", getpid());
        printf ("Child [PID %d] exit code is %d\n",
                child, WEXITSTATUS (status));
    }
    return 0;
}
```

- Add the correct form of **wait()** and **exit()** in the above program. Note: use the variables provided and referenced in the `printf` statements
- Run the program several times

6. Who prints first, the child or the parent? Why?
7. What line did you add for the wait() system call?

Notice that the parent incorrectly prints the child's PID. Make necessary change to fix this problem.

8. Describe the interaction between the **exit()** function and the **wait()** system call. To understand this better, you may want to do a number of experiments by changing the value passed (by the child) to the `exit()` call. [i.e. try exit(76), exit(43),...]

Replacing Process Image

Another important system call in Unix process control is `execve()`. There are several library wrapper functions to this system call. Check the manual page of `execl()` for more detail. Using this system call, a process can execute an external program by loading the (external) program binary executable into the current process image, hence **replacing** the original process image with that of the external program. However, when the external binary is not found, the original process image will **remain intact**.

```
/* Sample 4 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main () {
    char *m_args[] = { "July 2050", "7", "2050", NULL };

    printf ("I am process %d, created by process %d\n", getpid (),
            getppid ());
#if 1
    printf ("Using execl()\n");
    execl ("/usr/bin/cal", "Lakers", "10", "1988", NULL);
#else
    printf ("Using execvp()\n");
    execvp ("cal", m_args);
#endif
    printf ("End of program\n");
    return 0;
}
```

- Compile and run the above program
- Compare the out of the program with the output of the following command typed on the command line:

- `cal 10 1988`

- Replace "#if 1" with "`#ifdef ULIMARTA`". (The missing 'D' is intentional). Compile using the following command line and run again (replace "sample4" as necessary):

  ```
  gcc -Wall sample4.c
  ```

- Now compile again by including the `-D` flag.

  ```
  gcc -DULIMARTA -Wall sample4.c
  ```

  Run the executable

9. Where is the string "End of program" printed? Explain what happens
- Replace "cal" in the execvp call with your name [or some invalid command]. Compile without the -D and run (again).
10. In the context of your response to the previous question, explain what happens
11. What is the significance of the **first** argument passed to the binary executable by **exec*()** (i.e. the **second** arg of execl: "Lakers" [or "July 2050"] in the above code)?

Replacing Child Image

Sample 4 demonstrates how **exec()** replaces the image of the current process (with an external binary executable). Combining this call with the**fork()** system call enables us to replace **only** the child image (with an external binary executable) *while keeping the parent image intact*. See Sample 5 below.

```
/* Sample 5 */
#include <errno.h>      /* for perror() and errno */
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

/* This program has a few bugs that must be fixed */
int main (int argc, char*argv[]) {
    pid_t pid;
    pid = fork();
    if (pid < 0)
    {
        perror ("Unable to fork(): ");
        exit (errno);    /* errno is a global variable declared in errno.h */
    }
    if (pid == 0) /* child */
    {
        printf ("Child PID is %d\n", getpid());
        printf ("Running \"%s\" in child\n", argv[1]);
        execvp (argv[1], &argv[1]);
    }
    else { /* parent */
        int status;
        pid_t who_stop;

        printf ("Parent PID is %d\n", getpid());

        who_stop = wait (&status);
```

```
        if (WEXITSTATUS(status) == 0)
            printf ("Successful completion of child %d\n", who_stop);
        else
            printf ("Unknown command \"%s\" (status = %d)\n", argv[1],
                    WEXITSTATUS(status));
    }
    printf ("[Printed by %d]\n", getpid());
    return 0;
}
```

Before you compile and run the above code, understand the flow of execution in **both** child and parent processes.

12. Run Sample 5 while changing its command-line arguments in order to identify the bugs. The following assumes that the name of the executable is `sample5`.

```
13../sample5
14../sample5 cal      # expected output: the current
   month
15../sample5 cal -3   # expected output: 3 consecutive
   months (prev, curr, next)
16../sample5 calen    # expected output: Unknown command
   ....
```

    Describe the bugs that you observe and modify the code to remove the bugs. *Hint: Make sure you understand the execution flow of parent process and the child process, especially when the child attempted to execvp() a non-existant (external) binary executable.*

Rudimentary Shell

Now you have learned how `fork()`, `exec*()`, `exit()`, and `wait()` work together. You have the skills needed to write your own rudimentary Unix shell, let's call it **`rush`** (since you finish it in a hurry!). Recall that a shell reads user-typed commands (from stdin) and run them in a child process.

13. Write a C program `rush.c` that implements the following specifications (not in order):
    - Prompt the user for a command
    - Terminate `rush` when the user enters "quit"
    - Run the user-typed command in a (new) child process. The child exits when the command exits.
    - Print an error message "unknown command" if necessary
    - Prompt the user (again) for a command

Several design aspects to consider in `rush.c`

- The shell needs a loop to process the user commands. **Incorrect** placement of this loop might turn your C program into an **infinite fork() machine!**
- Use `fgets()` to read the user commands
- Use `strcmp()` or `strncmp()` to compare two strings
- Use `strtok()` to parse the user command into its tokens. The following snippet shows examples of using `strtok()`:

```
char tester[50] = "9/7/2010 11:59pm";
const char *DELIM = "/";
char *tok;

tok = strtok (tester, DELIM);
printf ("%s", tok);     //  output: "9"

/* in subsequent calls to strtok(), first
argument must be NULL */
tok = strtok (NULL, DELIM);
printf ("%s", tok);     //  output: "7"

tok = strtok (NULL, DELIM);
printf ("%s", tok);     //  output: "2010
11:59pm"

tok = strtok (NULL, DELIM);  /* returns
NULL in tok */
```

```
char tester[50] = "9/7/2010
11:59pm";
const char *DELIM = ":/";  /* colon
OR slash */
char *tok;
tok = strtok (tester, DELIM);
printf ("%s", tok);     //  output:
"9"

tok = strtok (NULL, DELIM);
printf ("%s", tok);     //  output:
"7"

tok = strtok (NULL, DELIM);
printf ("%s", tok);     //  output:
"2010 11"

tok = strtok (NULL, DELIM);
printf ("%s", tok);     //  output:
"59pm"

tok = strtok (NULL, DELIM);  /*
returns NULL in tok */
```

- The common practice is to call `strtok()` in **a loop** until it returns NULL. The following snippet of code will take up to 24 tokens. The address of each token is stored into my_args[0], my_args[1], ….

```
char* my_args[25];  /* an array to hold string addresses */
char *p;
int k;
p = strtok (a_string_to_tokenize, DELIMITER);
k = 0;
while (p && k < 24) {
    my_args[k] = p;   /* I don't want to use strcpy() because I want to
                         copy only the address of the string */
    k++;
    p = strtok (NULL, DELIMITER);   /* use NULL on subsequent calls */
}
my_args[k] = NULL; /* place a NULL at the end */
```

- The while loop limits k < 24 to make sure we have a spot to place NULL at the end.
- Consider breaking up your program into smaller functions, so it is easy to spot which function(s) runs in parent and which one(s) runs in child

- Preferably use `execvp()` to run the user command in your child process (so user commands with a long list of options/flags can be handled correctly)

SAMPLE RUSH SESSION

```
Welcome to RuSH!

rush> time
Usage: time [-apvV] [-f format] [-o file] [--append] [--verbose]
       [--portability] [--format=format] [--output=file] [--version]
       [--help] command [arg...]
Unable to run "time"
rush> date
Tue Sep 14 11:28:34 EDT 2010
rush> date -w
date: invalid option -- 'w'
Try `date --help' for more information.
Unable to run "date"
rush> ls -l
total 164
-rw----r-- 1 dulimarh users   826 Jan 15  2004 child-exec.c
-rwx---r-x 1 dulimarh users  6737 Aug 31 10:56 prog1
-rw----r-- 1 dulimarh users   166 Aug 31 11:47 prog1.c
-rwx---r-x 1 dulimarh users  7333 Aug 31 11:48 prog2
-rw----r-- 1 dulimarh users   383 Aug 31 11:48 prog2.c
-rwx---r-x 1 dulimarh users  7135 Aug 31 13:57 prog3
-rw----r-- 1 dulimarh users   624 Aug 31 13:57 prog3.c
-rwx---r-x 1 dulimarh users  6976 Aug 31 15:47 prog4
-rw----r-- 1 dulimarh users   381 Aug 31 17:14 prog4.c
-rwx---r-x 1 dulimarh users  7527 Sep 14 09:43 prog5
-rw----r-- 1 dulimarh users   766 Sep 14 10:10 prog5.c
-rwx---r-x 1 dulimarh users  8675 Sep 14 11:27 rush
-rw------- 1 dulimarh users  1526 Sep 14 11:27 rush.c
rush> calen
Unable to run "calen"
rush> quit
Bye!
```

By the way, did you try to run `rush` under *its own* rush session?

EXTRA CREDITS (RUSH)

A number of suggestions for extra credits

- Implement a builtin command to change the prompt
- Distinguish between "unknown command" vs. "invalid option" errors
- Print a summary at the end of rush session that shows the number of commands entered, the number of successful execution, ...