CS232L Operating Systems Lab
Lab 06: Introduction to Process Management in C

CS Program
Habib University

Fall 2024

1 Introduction

In this lab you will learn how to:

1. use process management API (fork, wait, exec)

2 Process Management

The Linux kernel exposes the process management system calls which we can access using the corresponding library wrapper function from our program. We've seen some of them in the Process API section along with code examples. In this lab we'll look at them again and put them into practice.

IMPORTANT: Whenever you use a system call, ALWAYS check its return value for errors !!!

2.1 fork

Fork is the quintessential system call for process creation and, when called from a running program, will create another process which will be an exact replica of the original process. The original and the created process are said to have a parent child relation ship respectively.

Listing 1 from the book example creates a child process and displays the PIDs for both the parent as well as the child.

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
         rc, (int) getpid());
    }
    return 0;
```

```
}
```

Listing 1: fork.c

Compile the above program and run it multiple times to see the order in which these processes will be scheduled. If the order doesn't change, try putting a call to sleep() in one of them (some times you've to force things :)). Call the sleep() function in your created processes to make them exist for sufficiently long to be able to verify their existnace and then run the linux ps command to verify that they are running in your system. Note their PIDs shown by the system and the ones shown by your program.

2.2 wait

Parents can wait for the termination of their children via the wait system call.

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
         rc, wc, (int) getpid());
    }
    return 0;
}
```

Listing 2: wait.c

Read the documentation of `wait` and notice its behaviour especially its return value Create multiple children of a process and store their `PIDs`. The parent should exit only after all its children have exited.

Other variants of `wait` are `waitpid` and `waitid`. Read their documentation and use them instead of `wait`.

2.3 exec

The `exec` system call overwrites a process memory image with another program supplied as its argument. The following listing from the book shows its working.

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
```

```
13          // fork failed; exit
14          fprintf(stderr, "fork failed\n");
15          exit(1);
16      } else if (rc == 0) {
17          // child (new process)
18          printf("hello, I am child (pid:%d)\n", (int) getpid());
19          char *myargs[3];
20          myargs[0] = strdup("wc");   // program: "wc" (word count)
21          myargs[1] = strdup("p3.c"); // argument: file to count
22          myargs[2] = NULL;           // marks end of array
23          execvp(myargs[0], myargs);  // runs word count
24          printf("this shouldn't print out");
25      } else {
26          // parent goes down this path (original process)
27          int wc = wait(NULL);
28          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
29           rc, wc, (int) getpid());
30      }
31      return 0;
32  }
```

Listing 3: exec.c

2.3.1 Exercise: exec

Write a program, a sort of mini-shell, which would continuously run in a loop displaying a prompt, e.g., `prompt>`, waiting a user to input a command (with arguments optionally). It would then `fork` a child process which would execute the command passed by the user. The parent should wait for the child to terminate and then prompt the user for the next command. If the user types"exit", the program should terminate.

2.4 I/O Redirection

The following listing shows how we can redirect the default file descriptors `(stdin, stdout, stderr)` of a process.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <assert.h>
7  #include <sys/wait.h>
8
9  int
10 main(int argc, char *argv[])
11 {
12      int rc = fork();
13      if (rc < 0) {
14          // fork failed; exit
15          fprintf(stderr, "fork failed\n");
16          exit(1);
17      } else if (rc == 0) {
18    // child: redirect standard output to a file
19    close(STDOUT_FILENO);
20    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
21
22    // now exec "wc"...
23          char *myargs[3];
24          myargs[0] = strdup("wc");   // program: "wc" (word count)
25          myargs[1] = strdup("p4.c"); // argument: file to count
26          myargs[2] = NULL;           // marks end of array
27          execvp(myargs[0], myargs);  // runs word count
28      } else {
29          // parent goes down this path (original process)
30          int wc = wait(NULL);
31    assert(wc >= 0);
```

```
32      }
33      return 0;
34  }
```

Listing 4: ioredir.c

2.4.1 Exercise: I/O Redirection

Enhance the above mini-shell to add I/O redirectoin for `stdout` to a user-provided filename, i.e., `prompt> cmd1 > cmd1.out`.

2.5 fork bomb

Google it. Not necessary to try :)