CS232L Operating Systems Lab
Lab 06: Introduction to C Programming

CS Program
Habib University

Fall 2019

1 Introduction

In this lab you will learn how to:

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

2 Get user input string

On command line users are mostly entering command as strings. You should be able to get user input in string form and use it inside your program. The following code snippet uses `scanf` function for this purpose:

```c
#include <stdio.h>

int main () {

  char str[20];
  printf("please enter a string:\n");
  scanf("%s",str);
  //scanf("%[^\n]s",str);
  printf("you entered: %s\n", str);


  return 0;
}
```

Listing 1: inputstr.c

Notice how you've to allocate space for the string storage yourself.

2.1 Todo: inputstr

1. compile and run `inputstr.c`
2. see what happens when we replace the `scanf` with the commented line
3. run the program and input a string that's much longer than 20 characters
4. two other functions used for the same purpose are `fgets` and `getline`. Read their documentation and rewrite this program using these two functions.

2.2 Todo: strtok

Once we've obtained the user input, the next step is to parse it.

Read the documentation of `strtok` function and use it to parse the user input string using the semicolon a as separator. Display each component of the user string separated by a semicolon on a separate line.

3 Process Management

The Linux kernel exposes the process management system calls which we can access using the corresponding library wrapper function from our program.

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

3.1 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.

The following listing 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 2: p1.c

3.2 Todo: fork

- call the `sleep` function in your created processes 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.

3.3 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());
  sleep(1);
    } 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 3: p2.c

3.4 Todo: wait

- 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`.

3.5 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) {
        // 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());
        char *myargs[3];
        myargs[0] = strdup("wc");    // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;            // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } 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 4: p3.c

3.6 Todo: exec

- write a program, a sort of mini-shell, which would continuously run in a loop waiting for 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 ëxit; the program should terminate.
- the `exec` system call has many variants. Try using a couple in your program.

3.7 I/O Redirection

The following listing shows how we can redirect the default file descriptors (STDIN, STDOUT, STDERR) of a process.

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

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
  // child: redirect standard output to a file
  close(STDOUT_FILENO);
  open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

  // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");    // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;            // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
  assert(wc >= 0);
    }
    return 0;
}
```

Listing 5: p4.c

3.8 fork bomb

Google it. Not necessary to try :)