Wayne State University

CSC 4421 - Fall 2016
Computer Operating Systems Labs
Lab 4 - Process Control I

Instructor 001: David Warnke
Instructor 003: Rui Chen

Points Possible: 100
Due: 11:59 pm, February 20, 2017

Tasks

Task 1

Here is a C program that prints out the process id, the parent process id, the process user id, and group id. Please type it in your computer and save as task1.c. Now compile it and run it to see how it works.

```
#include <stdio.h>
#include <unistd.h>
int main (void)
{
  printf("I am process %ld\n", (long) getpid());
  printf("My parent is %ld\n", (long)getppid());
  printf("Process user id is %ld\n", (long)getuid());
  printf("Group id is %ld\n", (long)getgid()); return 0;
}
```

Task1.c

Task 2* (50 points)

To create a new process, you can use the system call 'fork(void)'. Incomplete code is given below. The fork(void)system call is used to create a child process. Replace the _____ with proper code. It should work as follows.

- For the parent process, the program will print out "Parent Process: Global variable: 4 Function variable: 22";
- For the child process, it will print out "Child Process: Global variable: 3 Function variable: 21".

```c
#include <sys/types.h>
#include"stdio.h"
#include <unistd.h>
int globalVariable = 2;
int main()
{
  char parentStr[] = "Parent Process";
  char childStr[] = "Child Process";
  char *string = NULL;
  int functionVariable = 20;

  // Create a child process.
  pid_t pid = ______;

  if (______) // Failed to fork
  {
    perror("Unable to create child process");
    return 1;
  }

  else if (______) // child
  {
    // Code only executed by child process
    string = &childStr[0];
    globalVariable++;
    functionVariable++;
  }

  ______    // parent
  {
    // Code only executed by parent process
    string = &parentStr[0];
    globalVariable += 2;
    functionVariable += 2;
  }

  // Code executed by both parent and child.
  printf("%s\n", string);
  printf(" Global Variable: %d\n",globalVariable);
  printf(" Function Variable: %d\n", functionVariable);
}
```

Task2.c

Task 3*(50 points)

There are many variations of the exec system call that can be used to execute an executable file (program). Calling exec does not create a new process, it replaces the current process with the new process. If the exec command fails, then it returns -1. But if it succeeds it does not return because execution proceeds at the beginning of the program that was executed. We will consider some useful forms of exec commands.

1. execl: takes the path name of an executable as its first argument, the rest of the arguments are the command line arguments ending with a NULL.

```
execl("/bin/ls", "ls", "-a", "-l", NULL)
```

2. execv: takes the path name of a binary executable as its first argument, and a NULL terminated array of arguments as its second argument. The first element can be the command or " ".

```
static char* args[ ] = {"ls", "-a", "-l", NULL};
execv("/bin/ls", args);
```

3. execlp: same as execl except that we donâĂŹt have to give the full path name of the command.

```
execlp("ls", "ls", "-a", "-l", NULL);
```

4. execvp: Same as exexv except that we don't have to give the full path name of the command.

```
static char* args[ ] = {"ls", "-a", "-l", NULL};
execvp("ls", args);
```

shell.c below is a stripped down shell program. It will run a Unix command without any arguments. Modify the program so that it can execute any shell command. It should be able to take at least 2 arguments. Usually, you need to parse the command line you enter. Using execvp instead of execlp will probably help. It should work on any UNIX command consisting of 0-2 arguments after the command entered by the user, and be error free. The following are a few test cases that the user could enter.

$ cp file1 file2
$ ls -l
$ who

`char *strtok(char *str, const char *delim)` is a useful function for parsing the input string. You can see below how it used "\n" as a delimiter to get everything before that character. You can specify " " as a delimiter to split the string into arguments. The first time you use it, give it a string as the first parameter to tokenize. This will return the first token. You can then use it subsequent times with NULL as the first parameter and it will return subsequent tokens in order. After the last token was returned, it returns NULL. For example if you wrote:

```
char input[] = "Hello World!";
strtok(input, " ");
strtok(NULL, " ");
strtok(NULL, " ");
```

...the first call would return "Hello". The second call would return "World!". And the third call would return NULL.

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

int main()
{
  int PID;
  char lineGot[256];
  char *cmd;
  while (1){
    printf("cmd: ");
    fgets(lineGot, 256, stdin); // Get a string from user (includes \n)
    cmd = strtok(lineGot, "\n"); // Get the string without the \n
    if( strcmp(cmd, "e") == 0 ) // loop terminates when "e" is typed
      exit (0);
    // creates a new process. Parent gets the childâĂŹs process ID. Child gets 0.
    if ( (PID=fork()) > 0)
    {
      wait(NULL);
    }
    else if (PID == 0) /* child process */
    {
      execlp (cmd, cmd, NULL);
      /* exec cannot return. If so do the following */
      fprintf (stderr, "Cannot execute %s\n", cmd);
      exit(1); /* exec failed */
    }
    else if ( PID == -1)
    {
      fprintf (stderr, "Cannot create a new process\n");
      exit (2);
    }
  }
}
```

shell.c