Operating System Lab

Lab#5

Process and Shell Command

Goal:

This lab reviews process command and process creation. Then you learn how to create a shell in Linux.

Preparation:

Prepare the working environment

- Check out the repo from Github classroom
  - Use `git clone {repo URL}`
- Rebuild Docker image and run the container
  - In a Windows Powershell or MacOS/Liunx terminal, change the current folder to the "lab" folder (the one you check out).
    `cd {lab_folder}`
  - For **Mac** or **Linux**
    - You can execute the shell script "clean_start.sh"
      `./clean_start.sh`
  - For **Windows**
    - You need the following docker commands to remove existing image, build new image, and run a container. (You can also check the script for commands.)

```
docker container stop ubuntu-os-con
docker image rm -f ubuntu_os
docker build -t ubuntu_os .
```

```
docker run -itd -p8888:8888 --rm --name ubuntu-os-con -v"$PWD/share":/home/app ubuntu_os
```

  - To enter the "Bash Shell" of the Ubuntu container for compiling and testing codes

```
docker exec -it ubuntu-os-con /bin/bash
```

- Use VSCode or any text editors for development
  - In the host OS (Windows or MacOS), use terminal to change to the host's lab folder "{check out repo folder}/share" or VSCode to open that host's lab folder. Files you created or modified in the "{check out repo folder}/share" will be shown in docker container's "/home/app" folder.
  - Copy required files from "{check out repo folder}/share" to "{check out repo folder}/answer" and check in for submission.

Part I: Review

1. Learn Command for Listing Processes

Processes live in user space where is the portion of computer memory that holds running programs and their data. We can explore the contents of user space and obtain a listing of processes with the ps command.

```
$ ps
```

Each process has a unique identifying number called process ID or PID. -a option lists more processes including ones being run by other uses and at other terminals.

```
$ ps -a
```

-l option prints more informative lines. Figure 1 shows the output of the command.

```
$ ps -la
```

```
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1132    1130  0  80   0 - 130960 ep_pol tty2     00:00:00 Xorg
0 S  1000    1317    1130  0  80   0 - 47021 poll_s tty2      00:00:00 gnome-session-b
0 S  1000   15142    8873  0  80   0 - 313594 futex_ pts/0    00:00:00 docker
0 R  1000   24699    9154  0  80   0 -  2852 -      pts/1     00:00:00 ps
```

Figure 1: A typical output for ps -la command

Here is the summary of some important fields.

- S indicates status of each process. The process is running indicated by R and others sleeping show by S.
- Each process belongs to a user. A user ID (UID) is listed for each process. Each process has parent process ID (PPID).
- PRI and NI contains priority and niceness level. The kernel uses these values to decide when to run processes.
- SZ shows the size indicates amount of memory that is using.
- WCHAN shows why a process is sleeping. They are waiting for input and read c and do sel refers to addresses in the kernel.

2. Review on Process Creation

In general, when a process creates a child process, that child process will need certain resources (e.g., CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

Figure 2 shows a typical way of creating a process and have two different processes running copies of the same program. After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program, load a binary file into memory and starts its execution.

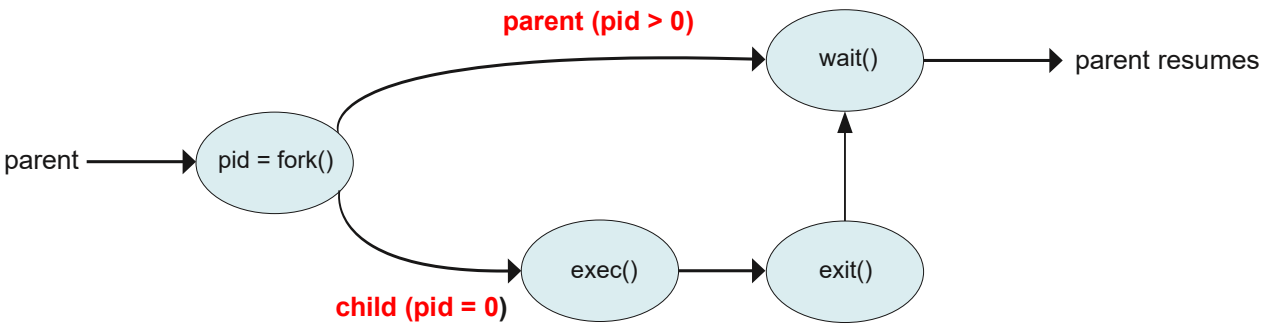

Figure 2: Creating a process by parent and wait for child

Figure 2 shows that parent and child can go their sperate ways and these two processes are able to communicate as well. The parent can issue a wait() system call if it has nothing else to do while the child runs.

When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the wait().

A code skeleton (see below) often occurs during the process creation by using return value of the fork().

```
/*fork a child process*/
fork_return = fork();
if (fork_return < 0){
    /*error occured*/
} else if (fork_return == 0){
    /*child process*/
} else {
    /*parent processs*/
}
```

To-dos (Part 1):

1. Write a shell script called "*myps.sh*" that can take a command line argument for process PID. Then, it will only print that particular PID with all fields (i.e., F, S, UID, PID, PPID, C…CMD) shown in the Figure 1. If PID is not matched (i.e., the process cannot be found), show "no process found for {PID}".

   Example Outputs:

   ```
   $ ./myps.sh 1265
   F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
   0 S 1000 1265 1263 0 80 0 - 130179 ep_pol tty2 00:00:00 Xorg

   $ ./myps.sh 12000
   no process found for 12000
   ```

2. Write a program called "*fork.c*" that follows the Figure 2's idea, the child process will call "ls -lt" command, the parent process will wait until child process completes. (Hint: review what you learn from the last lab.)
   - It also detects if the process creation fails and prints "Fork Failed" error message.
   - The parent process will print out "Child process is done" after the waiting. Which means your "Child process is done" will be at the last line of your standard output.

Part 2: Shell

1. Implementing a shell

Figure 3 shows the implementation of our first version of the shell. The program supposed to receive the command from the user and execute it though the execute() function. We use the first version of the shell to develop the final version by replacing the execute() function.

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

#define MAXARGS     20              /* cmdline args */
#define ARGLEN      100             /* token length */

int execute( char *arglist[] );

int main() {
    char    *arglist[MAXARGS+1];        /* an array of ptrs */
    int numargs;                        /* index into array */
    char    argbuf[ARGLEN];             /* read stuff here  */
    char    *makestring();              /* malloc etc       */

    numargs = 0;
    while ( numargs < MAXARGS ) {
        printf("Arg[%d]? ", numargs);
        if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' ) {
            arglist[numargs++] = makestring(argbuf);
        } else {
            if ( numargs > 0 ) {        /* any args?    */
                arglist[numargs]=NULL;  /* close list   */
                execute( arglist );     /* do the task  */
                numargs = 0;            /* and reset    */
            }
        }
    }
    return 0;
}

// NOTE: use execvp to do it
int execute( char *arglist[] ) {
    //TODO: write your code here        /* do it */

    exit(1);
}

// trim off newline and create storage for the string
char * makestring( char *buf ) {
    char    *cp;
    buf[strlen(buf)-1] = '\0';          /* trim newline */
    cp = malloc( sizeof(char)*(strlen(buf)+1) );
    if ( cp == NULL ) {                 /* or die   */
        fprintf(stderr,"no memory\n");
        exit(1);
    }
    strcpy(cp, buf);                    /* copy chars   */
    return cp;                          /* return ptr   */
}
```

Figure 3: The first version of shell implementation

```c
void execute( char *arglist[] )
/*
 *  use fork and execvp and wait to do it
 */
{
    int pid,exitstatus;                     /* of child */

    //TODO: put codes here                  /* make new process */
    switch( pid ){
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            //TODO: put codes here      /* do it */
            perror("execvp failed");
            exit(1);
        default:
            while(//TODO: Parent waits here )
                ;
            printf("child exited with status %d,%d\n",
                    exitstatus>>8, exitstatus&0377);
    }
}
```

Figure 4: The revised execute method for last version of shell implementation

To-dos (Part2):

1. Consider the code shown in Figure 3 and finish the program called "*psh1.c*". That is, complete the //TODO section of the Figure 3 in order to execute commands entered to arglist[] array. You can find the template files from the demo folder. You should understand how the program works in order to properly execute it.

2. Suppose the code in shown in Figure 4 is written to improve our first version implementation of the shell. Finish the program called "*psh2.c*". Think about why we consider wait here.
   (a) Complete the //TODO section of the Figure 4 in order to finalize our implementation of the shell. Keep in mind that, in Figure 4, only the `execute()` function is updated comparing with "*psh1.c*".
   (b) With revised version, use "`script`" command to capture screen the outputs of your improved version shell with inputs `$ls -l` and `$df -H` showing your final implementation of the shell and save the input and output in the file called "*psh2_results*".
   - To start script recording,

     `script psh2_results`
   - Use Ctrl+C to exit script recording
3. Which version do you prefer? Why do you prefer one version over the other? (psh1.c and psh2.c) in "*answer.md*".