Operating System Lab

Lab#4

Introduction to Process

Goal:

This lab will let you know important system calls that will control processes. It is very important to understand how processes work.

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:

1. Learn about Process

From page 1 of the book, "Interprocess Communications in Unix"[1]:

Fundamental to all operating systems is the concept of a process. While somewhat abstract, a process consists of an executing (running) program, its current values, state information, and the resources used by the operating system to manage the execution of the process. A process is a dynamic entity. In a UNIX-based operating system, at any given point in time, multiple processes appear to be executing concurrently. From the viewpoint of each of the processes involved it appears they have access to, and control of, all system resources as if they were in their own stand-alone setting. Both viewpoints are an illusion. The majority of UNIX operating systems run on platforms that have a single processing unit capable of supporting many active processes. However, at any point in time only one process is actually being worked upon. By rapidly changing the process it is currently executing, the UNIX operating system gives the appearance of concurrent process execution. The ability of the operating system to multiplex its resources among multiple processes in various stages of execution is called multiprogramming (or multitasking). Systems with multiple processing units, which by definition can support true concurrent processing are called multiprocessing.

With the exception of some initial processes, all processes in UNIX are created by the fork system call. The initiating process is termed the parent and the newly generated process the child. In fact, our shell is constantly forking processes.

In this lab, we ask you to do exercises on the system calls for process controls to be familiar with the control mechanism.

The main system calls that will be needed for this lab are:

- fork()
- exec*()
- wait()
- getpid(), getppid()

2. System Calls for Process Controls

- **fork()**

  - When a `fork()` system call is made, the operating system generates a copy of the parent process which becomes the child process.
  - If the fork system call is successful a child process is produced that continues execution at the point where it was called by the parent process.
  - After the fork system call, both the parent and child processes are running and continue their execution at the next statement in the parent process.
    ```
    fork_return = fork();
    ```
  - A summary of fork() return values follows:
    - fork_return > 0: this is the parent
    - fork_return == 0: this is the child
    - fork_return == -1: fork() failed and there is no child.
  - The operating system will pass to the child process most of the parent's process information.
  - However, some information is unique to the child process:
    - The child has its own process ID (PID)
    - The child will have a different PPID than its parent

- System imposed process limits are reset to zero
- All recorded locks on files are reset
- The action to be taken when receiving signals is different

To use the fork() in the C, you need to

```
#include <sys/types.h>
#include <unistd.h>
```

An example of using fork()

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
   printf("Hello \n");
   fork();
   printf("Bye\n");
   return 0;
}
```

The above code will output as follows:

```
$ ./fork_ex1
Hello
Bye
Bye
```

The above result output "Bye" twice because at this point, a child process has been forked. So, two processes both execute `printf("Bye\n");`

- **wait()**

A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions. The `wait()` system call allows the parent process to suspend its activities until one of these actions has occurred.

- The `wait()` system call accepts a single argument, which is a pointer to an integer and returns a value defined as type `pid_t`.
- If the calling process does not have any child associated with it, wait will return immediately with a value of -1.
- If any child processes are still active, the calling process will suspend its activity until a child process terminates.

The following shows the essential part of using `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

Two useful ones are:

- `WIFEXITED` evaluates as true, or 0, if the process ended normally with an exit or return call.
- `WEXITSTATUS` if a process ended normally, you can get the value that was returned with this macro.

```c
int status;
pid_t fork_return;

fork_return = fork();

wait(&status);
if (WIFEXITED(status)) {
    printf("Child returned: %d\n", WEXITSTATUS(status));
    printf("Now Parent can work\n");
}

for (int i = 0; i< 3; i++) {
    printf("Working!!\n");
    sleep(1);
}
```

Figure 1 wait()

The codes in Figure 1 shows that the `wait(&status)` holds the parents work until the child is done with its work. The line `fork_return = fork()` creates a new process (child) by duplicating the calling process (parent). The `if` statement checks to see that the child is returned; that is, `fork_return` is 0 for a child process and > 0 for a parent process. The output of the program in Figure 1 is shown in the Figure 2. Prints a message indicating the parent can now work. In the parent process, the program enters a loop and prints "Working !!" three times, with a one-second delay between each print.

```
Child returned: 42
Now parent can work.
Working !!
Working !!
Working !!
```

Figure 2 The output of the Figure1's program

- **getpid() and getppid()**

- `getpid()` returns the process id of the current process. The process ID is a unique positive integer identification number given to the process when it begins executing.
- `getppid()` returns the process id of the parent of the current process. The parent process forked the current child process.

To use both functions, we write:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

The following is an example:

```
int status;
pid_t fork_return;

fork_return = fork();

int pid = getpid();
int ppid = getppid();
printf("my id: %i\n", pid);
printf("my parent id: %i\n", ppid);
```

Figure 3 getpid() and getppid()

To-do (Part 1):

1. Write a program called "myfork.c" that modifies the following codes (see Figure 4). In the program, please replace "I am here" with "Hello, {your name} is here". For example, "Hello, Alex is here". Please put your explanations about what you see in the output of the myfork.c execution in the file "answer.md".

   Note: It is important for you to understand what happened in this program.

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4
5   int main(void)
6   {
7       printf("Let's Start!!\n");
8       fork();
9       fork();
10      fork();
11      printf("Yes I am here\n");
12      return 0;
13  }
```

Figure 4 fork()

2. Now, you know how to use `getpid()` and `getppid()`, you need to implement a program, called "pc_pid.c" which can identify parent and child processes, print their pids. Both

processes need to do the task – printing "Working" 10 times. For the child, using `wait()` and `sleep()` methods to give 2 seconds to the interval between two print out "Child Process {the_child_id} Working!" and let child finish the work first. The following is an output.

```
Process 4648 I am working!
Create child process!!
Child  Proces  4649  of  Parent
Process 4648
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Child Process 4649: Working!
Parent Process 4648
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
Parent Process 4648: Working!
```

Part II:

1. System Calls – exec() Family

Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing. The exec family of functions replaces the current running process with a new process.

- It can be used to run a C program by using another C program.
- The text, data and stack segment of the process are replaced and only the u (user) area of the process remains the same.
- The family includes `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp`.
- If successful, the exec system calls do not return to the invoking program as the calling image is lost.

- It is possible for a user at the command line to issue an exec system call, but it takes over the current shell and terminates the shell.

  ```
  $exec command [arguments]
  ```

We will only cover "`execvp`".

- **execvp()**

  `execvp` : This system call is used when the numbers of arguments for the program to be executed is dynamic.

  The following Figure 5 is a demo of how to use `execvp`. The Figure 6 is a result of the program. Keep in mind, your result will be different. The "Ending------" will not be printed because the process has been transferred.

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main() {

    // NULL terminated array of char* strings
    char* args[] = {"tail", "-2", "/var/log/syslog", NULL};
    execvp(args[0], args);
    // All statements are ignored after execvp() call
    // as this whole is replaced by another process
    printf("Ending-----");

    return 0;
}
```

Figure 5 execvp()

```
Sep 17 03:59:31 ubex systemd[1215]: jupyter.service: Main process exited, code=exited, status=216/GROUP
Sep 17 03:59:31 ubex systemd[1215]: jupyter.service: Failed with result 'exit-code'.
```

Figure 6 the result of last execvp() demo

To-do (Part 2):

1. Please learn from the Figure 5. Write a "hello.c" program that takes a command line parameter and then only print out the "Hello World, {the_command_line_parameter }". When you compile hello.c into hello, in the command line you type `./hello alex` the output will be "`Hello, alex`".

   Write another program called "exec_to-do.c" and use "`execvp`" to run the "hello_world" program. So, the command line parameter for exec_to-do.c will be passed to hello_world program. You also need to avoid errors when unexpected command line user input.