HW #2. Process Concept

Step 1. Get the example code

You can clone the example code tree from github

```
git clone https://github.com/RuKuWu/2013_NCTU_OS_hw.git
```

Or, you can download the zip file from github/e3 and unzip it

P.S. - Git is pre-installed on CS workstations.

Step 2. Build the example code

Go to the directory of the example code

```
cd 2013_NCTU_OS_hw-master
```

Initiate the build process by

```
make
```

Or, you can build each individual example by

```
make [ fork | fifo | shm ]
```

If you are familiar with multi-process programming, you can fast-forward to the Assignment section.

Instruction

A. Process Information

In Linux, most of things are implemented as special files, so as runtime process. `/proc` is a virtual directory. You can get the information of running processes by looking up files and directories in this folder.

Now, we will show how to get the information from `/proc`.

Step 1.

> Open any application you want to observe. We use Firefox as example, which is a default browser in most of linux-based operating systems.

Step 2.

> Type `ps aux` (it is distinct of `ps -aux`) or `ps fax` in terminal to list all running processes.
> Then, type `ps aux | grep firefox` to show process information of firefox.(Remember the pid of firefox)
>
> `ps` and `grep` are build-in program in linux. First of one can list all processes in operating system, the other one can filter the input and just print out what you want.
>
> If you want to know more details of these two program, type `man ps` and `man grep` in terminal.

Step 3.

> Type `pstree [PID]` and you can see the processes tree of firefox.

> Type `cd /proc/[PID]; ls` and you can see all things about the process of firefox.

B. Linux Multi-Process Programming

In this part, we will tall you how to write a multi-process program in linux environment. If you are already familiar with this, you can skip this part.

Before coding: some of our suggestion

1. Programming language

    > Basic C/C++ knowledge is required.

2. Editor

    > Vim is recommonded. However, you can use any editor you like.
    > If you are finding a suitable editor, you can try gedit, bluefish, code::blocks and sublime.

3. Makefile

    > Makefile can help you build the src code and save time.

Fork

`fork()` duplicates the calling process. The origin process is called as 'parent', duplicate is called as 'child'.
`fork()` returns pid of child process in parent and return 0 in child. If `fork()` is failed, -1 is returned.

If you want to know more details of fork, type `man fork` in terminal.

* Example 1-1 fork

```
#include <unistd.h>

pid_t fork(void);
```

> This example show the basic use of fork. In the example, fork is called and create the child process.
> Usually, we use if statement to decide actions of each process.

```
cpid = fork();
if (cpid < 0) { /* failed */
} else if (cpid == 0) { /* child process */
} else { /* parent process */ }
```

> You can see two of them use same variable "str" as print string buffer; however, they print out different content.
> That is because when fork is called, child process is assigned its own memory space. Thus, they would not change any variable which belongs to each other.

* Example 1-2 fork with waitpid

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

pid_t waitpid(pid_t pid, int *status, int options);
```

> We extend the example 1-1 to show how waitpid works. `waitpid()` is a function to let parent process wait for any child process to end.
>
> In our example, child uses a file to leave a message to parent. If parent reads before child writes, no content will be read. Thus, Parent should be paused, waiting for end of file writing of child and then parent could read the file. In the program, parent calls the `waitpid()` to wait for child to end.
>
> `waitpid(-1, 0, 0)` means waiting for all children done, same as `wait(0)`.
> If you want more details of `waitpid()`, type `man waitpid` in terminal.

Named pipe (a.k.a. FIFO)

Pipeline is a useful mechanism in Unix/Linux based system. It creates a buffer in kernal to let a program push some messages and any program knowing the pipe descriptor can pop out the messages.

In Part A - Step 2, we use a command `ps aux | grep firefox`. It's a use of pipeline. `command A | command B` means to pipe the standard output of A to B as input via a anonymous pipeline. Thus, in command `ps aux | grep firefox`, the output of `ps aux` is redirected to `grep firefox` as input.

First, we will introducte the anonymous pipeline in example 2-1 to let you know how pipeline works.
Then, in exmample 2-2 and 2-3, we will talk about named pipe(FIFO) and show you what's a difference between anonymous and named pipe.

* Example 2-1 pipe

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

> In this example, we call `pipe()` to create a pipeline for interprocess communication. When `pipe()` is called, it will give out two file descriptors which are not point to a file but a kernal buffer. The first descriptor fd[0] is for popping things from memory, the second descriptor fd[1] is for pushing.

```
pipe(fd);
cpid = fork();
if ( cpid < 0 ){ /*failed*/
} else if ( cpid == 0 ) { /* child process, send a message to parent */
  close(fd[0]); /* close a unused read pipe fd */
  write(fd[1], buf, sizeof(char) * (strlen(buf)+1) );
  close(fd[1]);
} else { /* parent process, recv message from child */
  close(fd[1]); /* close a unused write pipe fd */
  read(fd[0], buf, sizeof(char) * BUFFER_SIZE);
  close(fd[0]);
}
```

> **Please note that** `pipe()` should be called before `fork()`. That is because all processes should know the file descriptors of pipeline. If `pipe()` is put after `fork()`, only the process calling `pipe()` has correct value of the descriptors. In contrast, if putting `pipe()` before `fork()`, value of the file descriptors is copied to child process and child process can also use the descriptors.
>
> If you want to know more details of `pipe()`, type `man pipe` in terminal.

* Example 2-2 FIFO

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

int mkfifo(const char *pathname, mode_t mode);
```

> Named pipeline (a.k.a. FIFO) is same as anonymous pipeline, but creating a special file. This file let communication between two unrelated processes, which have no parent-child relation, becomes possible. But in this example, we show the basic use at first.
>
> To use named pipeline, `mkfifo()` is called to create a named pipeline file. And then you need open the file to use the pipeline **(use `open()`, not `fopen()`)**. After those, you can use it just as a regular file.

```
mkfifo(path, 0666);
if ( cpid < 0 ) { /* failed */
} else if ( cpid == 0 ) { /* child process, send a message */
  fd = open(path, O_WRONLY);
  write(fd, str, sizeof(char)*(strlen(str)+1));
  close(fd);
} else {
  fd = open(path, O_RDONLY);
  read(fd, str, sizeof(char)*BUFFER_SIZE);
  close(fd);
}
```

* Example 2-3 communication of two unrelated processes

> You can create a named pipeline file by your hand. Type `mkfifo test` in terminal and you can see the file named "test".

```
prw-rw-r--  1 root root 0M 10月  3 00:35 test
```

> As above, the first flag of the file is 'p', which means a pipeline file. A pipeline file links to a kernal buffer, instead of disk. You can use it to do interprocess communication.
>
> Open a new terminal and type `ls > test` in it. you can see nothing printed. Don't worry, things are quite normal.
> Open another terminal and type `cat < test`. Surprisingly, what should be printed on first terminal appear on the second terminal.
> That is because the output of `ls` is buffered, and then `cat` take them from "test" as input to display.
>
> The code "2-3" does similar things as above.
> If you want to know more details of `mkfifo()`, type `man mkfifo` and `man 3 mkfifo` in terminal.

Shared memory

A shared memory let two or more processes share a chunk of memory. All of them can access to this memory directly. You can image all these processes draw on the same paper, any of them can add and remove graphs anytime, anywhere. Thus, there is a problem if two or more processes access to same location at the same time. (multi-process sync problem)

* Example 3-1 Shared memory

> This is a example of basic use of shared memory. There are three main function - `shmget()`, `shmat(), `shmdt()`.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

> `shmget()` creates a chunk of memory. `shmat()` and `shadt()` do attaching and detaching to the memory.

```
  int shmid = shmget(IPC_PRIVATE, 1024, 0666);
  char* buf = shmat(shmid, 0, 0);
  shmdt(buf);
```

> After calling `shmat()`, you can use the memory what you want. In the example, we just use it as a string buffer.

* Example 3-2 Shared memory with key

> This example show how to use a shared memory with a key. In example 3-2, we create a shared memory with the key "IPC_PRIVATE". In fact, IPC_PRIVATE is equal to "0", it means to create a shared memory with any key. If creating it with a given key, any processes can use the same key to access the memory.

* Example 3-3 Remove the shared memory

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

> shmctl is a control function of the shared memory.

```
shmctl(id, IPC_RMID, 0);
```

> We just use it to remove a shared memory. There is a user commond with a same function.
> Type `ipcs -m` in terminal to show all shared memory in and type `ipcrm [shmid]` to remove the shared memory with "shmid".

Assignment

Part A

In this part, you are required to implement a simple game in two different way (FIFO and Shared memory).

Introduction

This game is played by two players on a 50x50 board. Each player has a unique ID. During the game, the players move by filling their IDs into the cells on the board in turn. Each cell can only be filled with at most one ID. A filled cell cannot be overwritten. The player who makes the first move can choose any cell in the board to fill his/her ID. For the follow-up moves, a player can only choose an empty cell adjacent to the cell where the last move was made. If the last move was made at coordinate (x,y), the adjacent cells are (x-1,y), (x,y-1), (x+1,y), and (x,y+1).

> Example:
> +++++++++#++++++++++
> +++++++++$++++++++++
> +++++++++#++++++++++
> ++++++++_$_+++++++++
> +++++++++_++++++++++
> $,# are the cells filled by two players. Now, it's #'s turn. _ shows the possible cells for # to fill.

When no player can fill any cell in the board, the game ends and the player who did the last filling is the winner.

Requirement

You should implement this game with multi-process programming. Each player is modeled by a process, and the two players (processes) will communicate via IPC. All the movements done by each process should be recorded in a respective file named [PID]_[FIFO | SHM].txt.

In each file, the first line contains an integer F. If the corresponding process(player) moves first, you should set F=1; Otherwise, you should set F=0.

The following lines describe the locations of the cells filled by this process(player). Each line corresponds to one location represented by a pair of integer i, j separated by single space. The location of the upper-left cell is (0, 0).

The file ends with the line containing an integer W. If the player wins, W=1; Otherwise W=0.

> Example:
> ----2131_FIFO.txt----
> 1
> 5 6
> 6 7
> .
> .
> .
> .
> .
> 0
>
> ----2132_FIFO.txt----
> 0
> 5 7
> 6 8
> .
> .
> .
> .
> .
> 1

In your report, you should explain your design and how you solve the synchronization problem in the shared memory version.

- (**10pts**) A.1: FIFO version Code

- (**10pts**) A.2: SHM version Code
- (**20pts**) A.3: Report

Part B

Take a look at shared_mem.cpp and try to compile it on a multi-core Linux machine (e.g. linux1.cs.nctu.edu.tw) with

```
g++ -O3 -DSINGLE ./shared_mem.cpp -lrt
```

And, run it on the machine.

Now, try to compile it with

```
g++ -O3 -DCOPY ./shared_mem.cpp -lrt
```

And, run it.

Now, try to compile it with

```
g++ -O3 -DSHM ./shared_mem.cpp -lrt
```

And, run it.

Now, try to compile it again with

```
g++ -O3 -DMMAP ./shared_mem.cpp -lrt
```

And, run it.

Based on the experiment results and your study of the code, please answer the following questions

- (**10pts**) B.1: What is the purpose of the program? What are the meanings of the output messages?
- (**20pts**) B.2: Tweak BUF_SIZE (in shared_mem.cpp) to 1024*1024*60 and redo the experiments. Describe your findings and explain the cause.
- (**10pts**) B.3: What are the technical differences between using -DSINGLE and -DCOPY?
- (**10pts**) B.4: What are the technical differences between using -DSINGLE and -DSHM?
- (**10pts**) B.5: What are the technical differences between using -DSHM and -DMMAP?

Submission

Please submit your homework in a zip or rar file named [Student ID].zip/rar. Including
1. FIFO version code
2. Shared memory version code
3. Report (100~300 words in part A, 200~500 words in part B) in word, pdf or markdown format.

If you have any question, e-mail me or knock the door of EC618
lukewu.cs02g@nctu.edu.tw