ITF22519: Introduction to Operating Systems

Fall Semester, 2022

Lab11: InterProcess Communication(IPC) 1

Submission Deadline: November 8^{th}, 2022 23:59

You need to get at least 50 points to pass this lab assignment

In lab7 (Thread Synchronization), you have learned how different threads in **one process** coordinate with each other to solve **a common problem**. In some applications, different processes may have to coordinate with each other to solve the problem. These processes can be at the same computer or at different computers. In this lab, you will look at several approaches that two (or more) processes in **one machine** or in the same system communicate or coordinate their activities with each other. A quick question raised is: can two processes in the same computer communicate the same way as they are at two different computers?

IPC is nothing but a way for one proccess to exchange a piece of information with another process. They can communicate by using **Pipe, Signal, Shared Memory, Socket, Message Queue and Semaphores**. In this lab, we will go through the first three approaches.

1 Pipe

The simplest form of IPC is a **pipe**. In the textbook, page 44, "a pipe is a sort of pseudofile that can be used to connect two processes', as shown in Fig. 1. There are `unnamed pipe` and `named pipe`. The later is also called FIFO.

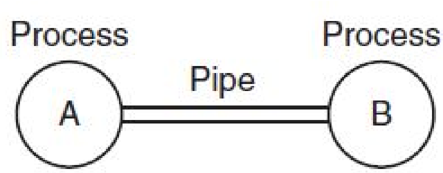

Figure 1: Two processes connected by a pipe (Figure 1-16 in the Textbook).

1.1 Unnamed Pipe

You have a bit experience with Pipe in lab1 assignment. When you pipe the output of one program into the input of another, you are creating an unnamed pipe. For example, this command in Bash would take output from `ps` and `pipe` it to `less` so that you can see it.

```
$./ps -el | less
```

Another way to create an unnamed pipe is to use a special system call in C. If processes A and B wish to talk using a pipe, they must set it up in advance. When process A wants to send data to process B, it writes on the pipe as though it were an output file. In fact, the implementation of a pipe is very much like that of a file. Process B can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in UNIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a `pipe(2)` system call [Textbook].

The function in C to be used to create a pipe is `pipe`. This function creates a **unidirectional** pipe and returns a two element array where the first element is the read and the second element is to write end of the pipe. The `pipe()` function can be called before the `fork()` system call. Then, the child and parent processes can use it to communicate. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For more information about pipes, use `pipe(2)` and `pipe(7)`.

Task 1

The program `pipe_test.c` uses unnamed pipe. Run the code example and explain the output.

- Run the code example and explain the output.
- Explain the code and the output of the program.
- How many processes are running?

1.2 Named Pipe

Another approach to create a pipe is a named pipe by using `mkfifo` command in your terminal or `mkfifo(3)` library call in C. A FIFO behaves exactly the same way as a pipe except that it has a name so that any process can reference to it.

Now, open two terminals in the same directory and create a new FIFO using `mkfifo`.

```
$ mkfifo fifo_test
```

In one terminal, run the following command:

```
$ cat fifo_test
```

This command is now watching FIFO and will show anything written to the FIFO. Now, in another terminal, type the following command:

```
$ echo "Hi FIFO" > fifo_test
```

2 Signals

You have been using signal so far but you may not be aware of it. Signal is a special command that a program receives from kernel mode in operating system. A signal is triggered by the CPU or the software (program) that runs on the CPU. When a process receives a signal, the default action happens unless the process has arranged to handle the signal. Here is one example of signal. When you press Ctrl-C in your terminal to close a program, you send SIGNIT (or kill) signal to that program. When you press Ctrl - Q to quit a program, you send SIGQUIT to the program. Another example of signal is the `kill` which you look at in lab5. In this lab, the `kill` system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call)[Textbook]. Signal is used for communication among different processes in the the system. However, unlike other kind of IPC, signal does not send any data from one process to another, it only sends the signal. For more information about signal, use `man page signal`. Signals are well defined in Linux. However, you can define your own signals and signal handlers and use them for IPC. Lets do simple practice with Ctrl-C in **Task 2**.

Task 2

The program in `sig_test.c` handles the signal `Ctrl - C` when the signal is received. Compile and run the program `sig_test.c` and answer the following questions.

- What happens when you try to quit the program by using `CTRL + C`? Explain why.
- How the process is defined to handle the signal `CTRL + C`?
- How to exit the program?

3 Shared Memory

Shared memory or share memory space (SHM) is an IPC approach that allows different processes or applications to access to the shared memory region. All data in this area is accessible to any process which opens the SHM. Linux provides you with several POSIX shared memory APIs to practice with shared memory. As usual, `man page` is your best friend in Linux. Please use the man page for `sgn_overview(7)` and `shm_open(3)` for more information.

Note: In this lab, we only use POSIX shared memory APIs. If you use other share memory functions, it is likely that you are using System V Share memory. That may cause a bug that you do not know how to fix. Now, lets do a simple practice. Suppose that you have a data structure which consists of one string and one array of integer as the following:

```
struct SHM_SHARED_MEMORY {
    char a_string[100];
    int an_array[5];
};
```

If you want to put this data in the SHM, you first need to create a memory region which can be accessed by other processes. Use `shm_open` to open a SHM and name the SHM, for example "SharedMemory", as the following:

```
fd = shm_open("SharedMemory", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP);
if (!fd) {
    perror("shm_open\n");
    return -1;
}
```

`shm_open` often goes with `ftruncate`. This function is used to set the size of the SHM. Noted that a newly created SHM has a length of zero.

```
if(ftruncate(fd, sizeof(struct SHM_SHARED_MEMORY))){
perror("ftruncate\n");
return -1;
}
```

After creating the new SHM, you need to map the SHM into the virtual address space of the calling process. POSIX API provides `mmap()` to do this.

```
shared_mem = mmap(NULL, sizeof(struct SHM_SHARED_MEMORY), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

It is better to check if `mmap()` is done or not

```
if(!shared_mem){
   perror("mmap\n");
   return -1;
}
if(close(fd)){
perror("close\n");
return -1;
}
```

Now that you have a shared memory region. Let put values for the string and for the integer array. You can do this by

```
strcpy(shared_mem->a_string, "Hello");

for (i = 0; i < 5; i++){
shared_mem->an_array[i] = i*i;
}
```

Then, check if the SHM has the correct information you just have put in:

```
printf("Printing values stored in the shared memory ...\n");
printf("String is %s \n", shared_mem->a_string);
printf("Integers are %d   %d   %d   %d   %d\n", shared_mem->an_array[0], shared_mem->an_array[1], share
```

Remember to `unmap()` the SHM after using

```
int res = munmap(NULL, sizeof(struct SHM_SHARED_MEMORY));
if (res == -1){
    perror("munmap");
    return 40;
}
```

The completed code for the example above as below

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

struct SHM_SHARED_MEMORY{
     char a_string[100];
     int an_array[5];
};
```

```
int main(){
int fd;
// open the shared memory area, create it if it doesn't exist
fd = shm_open("SharedMemory", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP);
if (!fd) {
    perror("shm_open\n");
    return -1;
}

// extend shared memory object as by default it's initialized with size 0
if(ftruncate(fd, sizeof(struct SHM_SHARED_MEMORY))){
    perror("ftruncate\n");
    return -1;
}

struct SHM_SHARED_MEMORY* shared_mem; //variable declaration
int i;

        // map shared memory to process address space
shared_mem = mmap(NULL, sizeof(struct SHM_SHARED_MEMORY), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if(!shared_mem){
perror("mmap\n");
return -1;
}
if(close(fd)){
perror("close\n");
return -1;
}
strcpy(shared_mem->a_string, "Hello");

for (i = 0; i < 5; i++){
shared_mem->an_array[i] = i*i;
}
printf("Printing values stored in the shared memory ...\n");
printf("String is %s \n", shared_mem->a_string);

printf("Integers are %d %d  %d   %d   %d\n", shared_mem->an_array[0], shared_mem->an_array[1], shared_m

        // unmap
int res = munmap(NULL, sizeof(struct SHM_SHARED_MEMORY));
if (res == -1){
perror("munmap");
return 40;
}
return 0;
}
```

- Create the source code file *shm_open.c* in your lab9 repository.
- Compile the code by using following command

```
$ gcc shm_open.c -lrt -o open
```

Note: Remember `-lrt` compilation command. In this command, `-l` flag is to link with the library whose name follows the flag, in this case is `rt`. Hence, the command `-lrt` is to tell the compiler to link with the `rt` library, or `librt`.
Now, make another process that reads the SHM created by your program in *shm_open.c* and then modifies the values in the SHM. The code below is for that purpose.

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

struct SHM_SHARED_MEMORY{
     char a_string[100];
     int an_array[5];
};

int main(){
int fd;
// open the shared memory area, create it if it doesn't exist
fd = shm_open("SharedMemory", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP);
if (!fd) {
     perror("shm_open\n");
     return -1;
}

// extend shared memory object as by default it's initialized with size 0
if(ftruncate(fd, sizeof(struct SHM_SHARED_MEMORY))){
        perror("ftruncate\n");
        return -1;
}

struct SHM_SHARED_MEMORY* shared_mem;
// map shared memory to process address space
shared_mem = mmap(NULL, sizeof(struct SHM_SHARED_MEMORY), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

if(!shared_mem){
        perror("mmap\n");
        return -1;
}
if(close(fd)){
        perror("close\n");
        return -1;
}
// Read from the shared memory
```

```
printf("Reading values from the shared memory\n");
printf("String is %s \n", shared_mem->a_string);
printf("Integers are %d   %d   %d   %d   %d\n", shared_mem->an_array[0], shared_mem->an_array[1], share
printf("Now, change the value of the first element...\n");
shared_mem->an_array[0] = 42;
printf("Integers are %d   %d   %d   %d   %d\n", shared_mem->an_array[0], shared_mem->an_array[1], share

// unmap
int res = munmap(NULL, sizeof(struct SHM_SHARED_MEMORY));
if (res == -1){
        perror("munmap");
        return 40;
}
return 0;
}
```

Task 3

- Create the source code file ***shm_read.c*** in your lab11 repository.
- Compile the code by using following command

```
$ gcc shm_read.c -lrt -o read
```

- Run `./open` in one terminal and `./read` in another terminal. Add output in your report and explain how two programs work.

4 Exercises

4.1 Exercise 1 (33 pts)

Write two programs `read_fifo.c` and `write_fifo.c` which will be run on two terminals:

- `write_fifo.c` (while true): reads one line that user enters into one terminal and writes it to a named fifo.
- `read_fifo.c` (while true): reads any input from the same named fifo and print them out to the other terminal.
- Do we need to synchronize the two programs to ensure that the read operation will always happen after the write operation?

4.2 Exercise 2 (33 pts)

Compile and run the program `sig_fork.c` and answer the following questions:

- Explain the output?
- Why the variable `count` has different values?
- Specify how variable `count` is increased in child and parent processes.

4.3 Exercise 3 (34 pts)

Compile and run the files `shm_test1.c` and `shm_test2.c`. Observe the output by running both applications at the same time. Answer the following questions:

- What is the output if you run both at the same time and calling `shm_test1.c` first?
- What is the output if you run both at the same time and calling `shm_test2.c` first?
- Why is `shm_test2.c` causing a segfault? How could this be fixed?

5 What To Submit

Upload your related files and reports to GitHub repository.