ITF22519: Introduction to Operating Systems

Fall Semester, 2021

Lab9: InterProcess Communication(IPC) 2

October 22, 2021

You have learned **Pipe, Signal** and **Sockets** in previous lab. In this lab, we will look at **Shared Memory, Message Queue and Semaphores** which different processes in the same computer can use to coordinate their activities.

When processes are in the same system, they can communicate through a material in the common memory location where they all can access. Pipe, FIFO (named pipe) and **Shared Memory** are on this category. In addition, they can also communicate by using massage passing. Sockets and **Message Queue** fall on this category. (However, when the processes are in different computers, they have to communicate by using message passing. A common approach for it is message passing interface (MPI). In this case, a network is needed to connect the computers. However, we do not look at this case in the context of this course). **Semaphores** is a kind of IPC used to synchronize activities of different processes.

1 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[
```

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, 
     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[

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

```
            return 40;
        }
        return 0;
}
```

Task 1

- 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` flag to link again `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], s
    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], s

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

Task 2

- Create the source code file *shm_read.c* in your lab9 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.

Task 3

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?
- What happens if the two applications both try to read and set a variable at the same time? (e.g. `shared_mem->count++`)?
- How can a shared memory space be deleted from the system?

2 Message Queues

Message queues are a way of passing priority messages from one process to another. The sending process sends the message in its entirety. The receiving process retrieves the message from the queue. It also receives the message in its entirety. Note that each message has its own priority. The priority is indicated by non-negative value with 0 being the lowest. When messages are put into the queue, the ones with highest priority will be at the front of the queue.

Due to the message passing approach, the sending process has to perform sending operation and the receiving process has to perform receiving operation. POSIX message queue APIs provide functions for those activities. Use man page `mq_overview(7)`, `mq_open(3)`, `mq_send(3)`, `mq_receive(3)` and `mq_notify(3)` for more information. One nice feature of message queues is the ability to subscribe to events on the queue and to handle the events asynchronously.

The following programs are simple implementations of sending and receiving a 'HELLO' message using POSIX message queue. Create two files as the following: *`mq_send.c`* contains:

```
// Simple implementation of sending "HELLO" by using POSIX message queue APIs

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

int main(){
     // Create a new message queue
     mqd_t mqd = mq_open ("/Introduction2OS", O_CREAT | O_EXCL | O_WRONLY,  0600, NULL);

     // check if message queue creation is succesful or not
    if (mqd == -1){
        perror ("mq_open");
        exit (1);
    }
    mq_send (mqd, "HELLO", 6, 15);   // send a "HELLO" message using mq_send API
// note that the size of this message is 6 including NULL at the end of the string
                                     // priority of the message is 15
    mq_close (mqd);                  // close after using
    return 0;
}
```

`mq_receive.c` contains:

```
// Simple implementation of receiving "HELLO" by using POSIX message queue APIs
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <string.h>
#include <unistd.h>
```

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

int main(){
       // open an existing message queue to read
       mqd_t mqd = mq_open ("/Introduction2OS", O_RDONLY);
       assert (mqd != -1);

       struct mq_attr attr;   // Get the attributes of the queue message
       assert (mq_getattr (mqd, &attr) != -1);

       char *buffer = calloc (attr.mq_msgsize, 1);
       assert (buffer != NULL);

        int priority = 0;  // used to get the priority of the message queue
        if ((mq_receive (mqd, buffer, attr.mq_msgsize, &priority)) == -1)
              printf ("Failed to receive message!\n");
        else
              // print the message and its priority
              printf ("Message '%s': Priority %u\n", buffer, priority);
       free (buffer);    // free buffer
       buffer = NULL;    // clean queue
       mq_close (mqd);   // close after using
       return 0;
}
```

Task 4

Compile the files `mq_send.c` and `mq_receive.c` with the `-lrt` flag. Open two terminals. In the first termial, start `mq_send` and then in the other start `mq_receive`. In your report answer the following questions:

- How do two program work?
- Run another time and report what happens.
- If you see an assertion failed message, fix the code so that receiving process can receives message from the sending process.

Task 5

Compile the files `mq_test1.c` and `mq_test2.c` with the `-lrt` flag. Open two terminals. In the first termial, start `mq_test1` and then in the other start `mq_test2`. In your report answer the following questions:

- What is the output from each program?
- What happens if you start them in the opposite order?
- Change `mq_test2.c` to send a second message which is "My name is X" where "X" is your name. Change `mq_test1.c` to wait for and print this second message before exiting.

Task 6

Write two programs `read_mq.c` and `write_mq.c` will be run on two terminals:

- `write_mq.c` while true, reads one line that user enters into one terminal and send it to a message queue.
- `read_mq.c` while true reads the message from the same message queue and print it out to the other terminal.
- Do we need to synchronize the two programs to ensure read operations will always happen after write operations?

3 Named Semaphores

POSIX semaphores come in two forms: ***unnamed*** semphores and ***named*** semphores. They differ in the way they are created and destroyed. Unnamed semaphores are the ones you learn in Lab7. Named semaphores are covered in this handout.

Named semaphores are similar to the semaphores you used in Lab7 except that they can be shared between multiple processes without being in shared memory region. The naming convention is the same as for SHM. A name starting with a / character followed by an alpha-numeric string. Again, `man page` is your best friend in Linux. Use man page `sem_overview(7)` and `sem_open(3)` for more information.

Unlike **Shared Memory** and **Message Queue** where different processes can communicate with each other via shared memory region or message passing, **Named Semaphores** do not allow data exchange among processes. Rather, they are used to synchronize proccesses especially when the processes try to access the shared resource. In this case, **Named Semaphores** are used to prevent multiple processes from manipulating the shared resource at the same time. Some applications of using Named semaphores can be banking transfer, air flight booking etc.

Task 7

The following program is a simple implementation of the synchronization between a child process and a parent process using Named Semphoare. Create a ***semaphores.c*** file contains the following code and explain how the program works:

```
// Simple implementation of synchronization
// between child process and parent process using Named Semaphores
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>
#include <assert.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/types.h>

int main() {
     //initialize and open a semaphore named Introduction2OS_Sema
     sem_t *sem = sem_open ("/Introduction2OS_Sema", O_CREAT | O_EXCL, S_IRUSR | S_IWUSR, 0);
     assert (sem != SEM_FAILED);

     pid_t pid = fork();
     assert (pid != -1);

    /* Child process: wait for semaphore, print its message and then exit */
    if (pid == 0){
        sem_wait (sem); // decrements (locks) the semaphore pointed to by sem
        printf ("I am the child. My number is 2.\n");
        sem_close (sem); // close a named semaphore
        return 0;
    }

/* Parent process: prints its message, posts to the semaphore and waits on child */
    printf ("I am the parent. My number is 1.\n");
    sem_post (sem); // increments  (unlocks)  the semaphore pointed to by sem
    wait (NULL);

    printf ("My child has printed and exited. Now, my number is 3.\n");
    sem_close (sem); // close a named semaphore
    sem_unlink ("/Introduction2OS_Sema"); // removes the named semaphore Introduction2OS_Sema
    return 0;
}
```

Task 8

Answer the following questions in your report:

- How long do semaphores last in the kernel?
- What causes them to be destroyed?
- What is the basic process for creating and using named semaphores? (list the functions that would need to be called and their order).

Task 9

Write two programs `read_shm.c` and `write_shm.c` which will be run on two terminals:

- `write_shm.c` while true: reads one line that a user enters into one terminal (maximum 1000 characters) and copy it to a string inside a shared memory area.
- `read_shm.c` while true: reads the string from the same shared memory area and prints it out to the other terminal.
- Use named semaphore to synchronize the two programs so that `read_shm.c` will always start after `write_shm.c`.

4 What To Submit

1. Upload related files and your report for this lab to your GitHub repository.
2. Write GitHub: <yourGitHubUsername> on the top of your report.