Refersher Lab for Process Management

Shared memory and Threads

In this lab you will learn the following

1. Creating shared memory
2. Using the shared memory for interprocess communication
3. Creating POSIX threads
4. Problems with synchronization

Shared Memory

Shared memory is one of the three interprocess communication (IPC) mechanisms available under Linux and other Unix-like systems. In case of shared memory, a shared memory segment is created by the kernel and mapped to the data segment of the address space of a requesting process. A process can use the shared memory just like any other global variable in its address space.

Shared Memory in the interprocess communication mechanisms like the pipes, fifos and message queues, the work involved in sending data from one process to another is as follows. Process P1 makes a system call to send data to Process P2. The message is copied from the address space of the first process to the kernel space during the system call for sending the message. Then, the second process makes a system call to receive the message. The message is copied from the kernel space to the address space of the second process. The shared memory mechanism does away with this copying overhead. The first process simply writes data into the shared memory segment. As soon as it is written, the data becomes available to the second process. Shared memory is the fastest mechanism for interprocess communication.

System V IPC key

To use a System V IPC mechanism, we need a System V IPC key. To create a key we use the *ftok* function, which is called with following parameters.

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

key_t ftok (const char *pathname, int proj_id);
```

The pathname is an existing file in the filesystem. The last eight bits of proj_id are used; these must not be zero. A System V IPC key is returned by the function.

System V shared memory creation, attachment to a local memory

Once a key is generated a shared memory creation within a user program has three discreet steps

1. Creating a shared memory in the kernel
2. Attaching that shared memory to a local variable
3. Detaching the shared memory from the local variable once the task is completed

Step 1 - shmget

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

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

As the name suggests, *shmget* gets you a shared memory segment associated with the given key. The key is obtained earlier using the *ftok* function. If there is no existing shared memory segment corresponding to the given key and IPC_CREAT flag is specified in *shmflg*, a new shared memory segment is created. Also, the key value could be

IPC_PRIVATE, in which case a new shared memory segment is created. *size* specifies the size of the shared memory segment to be created; it is rounded up to a multiple of PAGE_SIZE. If *shmflg* has IPC_CREAT | IPC_EXCL specified and a shared memory segment for the given key exists, shmget fails and returns -1, with errno set to EEXIST. The last nine bits of shmflg specify the permissions granted to owner, group and others. The execute permissions are not used. If shmget succeeds, a shared memory identifier is returned. On error, -1 is returned and errno is set to the relevant error.

Step 2 - shmat

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

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

With *shmat*, the calling process can attach the shared memory segment identified by shmid. The process can specify the address at which the shared memory segment should be attached with shmaddr. However, in most cases, we do not care at what address system attaches the shared memory segment and shmaddr can conveniently be specified as NULL. shmflg specifies the flags for attaching the shared memory segment. If shmaddr is not null and SHM_RND is specified in shmflg, the shared memory segment is attached at address rounded down to the nearest multiple of SHMLBA, where SHMLBA stands for Segment low boundary address. The idea is to attach at an address which is a multiple of SHMLBA. On most Linux systems, SHMLBA is the same as PAGE_SIZE. Another flag is SHM_RDONLY, which means the shared memory segment should be attached with read only access.On success, shmat returns pointer to the attached shared memory segment. On error, (void *) -1 is returned, with errno set to the cause of the error.

Step 3 - shmdt

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

int shmdt (const void *shmaddr);
```

shmdt detaches a shared memory segment from the address space of the calling process. shmaddr is the address at which the shared memory segment was attached, being the value returned by an earlier shmat call. On success, shmdt returns 0. On error, shmdt returns -1 and errno is set to indicate the reason of error.

Producer- Consumer problem

Consider the following programs

1. There is a shared array that can store 10 strings
2. Producer program – It takes an input string from the user and create and store that string in an array as long as the array have space
3. Consumer program – When the array is not empty the consumer takes the top most string from the array and prints it

Producer program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

```
#define MAX_BUFFERS 10

struct shared_memory {
    char buf [MAX_BUFFERS] [256];
    int buffer_index;
};



int main()
{
        key_t s_key;

        int shm_id,i=0;
        struct shared_memory *mem;

        char sentence[100];

        s_key = ftok("hello", 1);

        if (s_key == -1){
                perror("ftok");
                exit(1);
        }

        if ((shm_id = shmget (s_key, sizeof (struct shared_memory),  0660 | IPC_CREAT)) == -1)
                perror ("shmget");


        if ((mem = (struct shared_memory *) shmat (shm_id, NULL, 0)) == (struct shared_memory *) -1)
                perror ("shmat");

        mem->buffer_index = 0;

        printf("Enter a string\n");
        scanf("%s",sentence);

        sleep(1);

        for(i=0;i<10;i++){

                strcpy(mem->buf[mem->buffer_index], sentence);
                mem->buffer_index++;
        }

}
```

Consumer program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

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


#define MAX_BUFFERS 10

struct shared_memory {
    char buf [MAX_BUFFERS] [256];
    int buffer_index;
};



int main()
{
        key_t s_key;

        int shm_id;
        struct shared_memory *mem;

        char sentence[100];

        s_key = ftok("hello", 1);

        if (s_key == -1){
                perror("ftok");
                exit(1);
        }

        if ((shm_id = shmget (s_key, sizeof (struct shared_memory),  0)) == -1){
        perror ("shmget");
        }

        if ((mem = (struct shared_memory *) shmat (shm_id, NULL, 0)) == (struct shared_memory *) -1)
                perror ("shmat");



        while(1){

                if(mem->buffer_index > 0){

                        printf("Buffer content is %s\n", mem->buf[mem->buffer_index-1]);
                        mem->buffer_index--;
                }
        }

}
```

Running the program:

Compile producer.c and consumer.c with gcc as follows

```
gcc producer.c –o producer
gcc consumer.c –o consumer
```

Open 3 terminals. Run producer program on 2 terminals and consumer program on the third terminal. Input different texts for each of the producer program (example: Hello and World). Observe the results.

Make a slight modification to the producer program. Insert sleep(1) in the last for loop in producer.c as follows.

```
strcpy(mem->buf[mem->buffer_index], sentence);
sleep(1);
mem->buffer_index++;
```

Recompile, the programs and rerun producer and consumer in three terminals. Do you notice any difference?

Q1. Use a shared memory to store one integer which is initialized to 0. Write an ADDER.c program which accepts a number from the user and adds that number to the shared memory. Write another program, RESULT.c that reads the value from the shared memory and prints it when the value is not zero. Execute 5 adder program and one result program in different terminals and show how the sum is accurately printed by the result program. Show the result to one of the lab supervisors.

POSIX Threads

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

A simple example of threads – 2 threads in the process running parallel to print 2 different messages

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

   /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
```

```
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}



void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Compile the program by using gcc –lpthread program.c

Q2: What is the function of pthread_create and pthread_join?

Sharing global variables by threads

Threads can share global variables in the process, thereby can communicate with each other through global variables, instead of shared memory. The following program creates 10 threads and each of the thread increments the value of a particular global variable. The main process will print the value of the global variable after all the threads executions are completed.

```
#define THREADS 10

void *incr();

int a = 0;
int main()
{

        pthread_t thread[THREADS];
        int  iret,i;

    /* Create independent threads each of which will execute function  incr*/

        for(i=0;i<THREADS;i++){
                iret = pthread_create( &thread[i], NULL, incr, (void*) NULL);

        }


        for(i=0;i<THREADS;i++){
                pthread_join( thread[i], NULL);
        }

        printf("All thread returns and a = : %d\n",a);
        exit(0);
}

void * incr()
{
        a = a + 1;
}
```

Compile the program and find the result of the program.

Change the number of threads to 20,000. Compile and run the program a few times. DO you get the same answer all the time?

Q3. Implement producer-consumer program with the help of threads. Create 3 threads for 3 producers and 3 threads for 3 consumers running infinitely. The producers keep producing numbers and store in an array of 5 elements and consumers keep consuming the numbers from the array.