

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Information and Computer Science Department
ICS 431 OPERATING SYSTEMS
Lab # 14

IPC using Shared Memory

Objective:

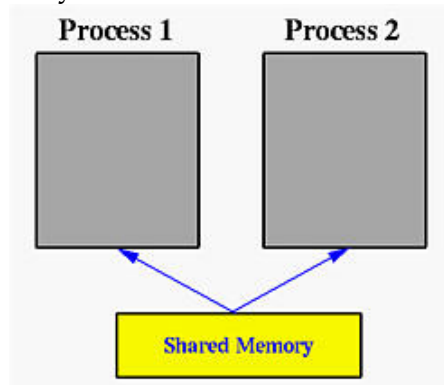
The aim of this laboratory is to show you how the processes can communicate among themselves using the Shared Memory regions. *Shared Memory* is an efficient means of passing data between programs. One program will create a memory portion, which other processes (if permitted) can access. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. In this lab the following issues related to shared memory utilization are discussed:

- ✳️ Creating a Shared Memory Segment
- ✳️ Controlling a Shared Memory Segment
- ✳️ Attaching and Detaching a Shared Memory Segment

Introduction:

What is Shared Memory?

In the discussion of the fork () system call, we mentioned that a parent and its children have separate address spaces. While this would provide a more secured way of executing parent and children processes (because they will not interfere each other), they shared nothing and have no way to communicate with each other. A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address space is "extended" by attaching this shared memory.



Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris. One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the **server**. All other processes, the **clients** that know the shared area can access it. However, there is no protection to a shared memory and any process that knows it can access it freely. To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

A shared memory segment is identified by a unique integer, the **shared memory ID**. The shared memory itself is described by a structure of type `shmid_ds` in header file `sys/shm.h`. To use this file, files `sys/types.h` and `sys/ipc.h` must be included. Therefore, your program should start with the following lines:

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

A general scheme of using shared memory is the following:

For a **server**, it should be started before any client. The **server** should perform the following tasks:

1. Ask for a shared memory with a memory key and memorize the returned shared memory ID. This is performed by system call `shmget()`.
2. Attach this shared memory to the server's address space with system call `shmat()`.
3. Initialize the shared memory, if necessary.
4. Do something and wait for all clients' completion.
5. Detach the shared memory with system call `shmdt()`.
6. Remove the shared memory with system call `shmctl()`.

For the **client** part, the procedure is almost the same:

1. Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
2. Attach this shared memory to the client's address space.
3. Use the memory.
4. Detach all shared memory segments, if necessary.
5. Exit.

Asking for a Shared Memory Segment - `shmget()` :

The system call that requests a shared memory segment is `shmget()`. It is defined as follows:

```
shm_id = shmget (
    key_t k,      /* the key for the segment */
    int  size,    /* the size of the segment */
    int  flag
); /* create/use flag */
```

In the above definition, `k` is of type `key_t` or `IPC_PRIVATE`. It is the numeric key to be assigned to the returned shared memory segment. `size` is the size of the requested shared memory. The purpose of `flag` is to specify the way that the shared memory will be used. **For our purpose, only the following two values are important:**

1. `IPC_CREAT | 0666` for a **server** (i.e., creating and granting read and write access to the server)
2. `0666` for any **client** (i.e., granting read and write access to the client)

Note that due to UNIX's tradition, `IPC_CREAT` is correct and `IPC_CREATE` is not!!!

If `shmget()` can successfully get the requested shared memory, its function value is a non-negative integer, the **shared memory ID**; otherwise, the function value is **negative**. The following is a **server** example of requesting a **private shared memory of four integers**:

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

.....
int  shm_id; /* shared memory ID */
.....
shm_id = shmget (IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0)
{
    printf("shmget error\n");
    exit(1);
}
```

/ now the shared memory ID is stored in `shm_id` */*

If a **client** wants to use a shared memory created with `IPC_PRIVATE`, it must be a **child process** of the **server**, **created after the parent has obtained the shared memory**, so that the **private key value** can be passed to the child when it is created. For a client, changing `IPC_CREAT | 0666` to `0666` works fine. **A warning to novice C programmers: don't change 0666 to 666.** The leading 0 of an integer indicates that the integer is an octal

number. Thus, 0666 is 10110110 in binary. If the leading zero is removed, the integer becomes six hundred sixty six with a binary representation 1111011010.

Server and clients can have a parent/client relationship or run as separate and unrelated processes. In the former case, if a shared memory is requested and attached prior to forking the child client process, then the server may want to use IPC_PRIVATE since the child receives an identical copy of the server's address space which includes the attached shared memory. However, if the server and clients are separate processes, using IPC_PRIVATE is unwise since the clients will not be able to request the same shared memory segment with a unique and unknown key.

Keys:

UNIX requires a key of type `key_t` defined in file `sys/types.h` for requesting resources such as shared memory segments, message queues and semaphores. A key is simply an integer of type `key_t`; however, you should not use `int` or `long`, since the length of a key is system dependent.

There are three different ways of using keys, namely:

1. a specific integer value (e.g., 123456)
2. a key generated with function `ftok()`
3. a uniquely generated key using IPC_PRIVATE (i.e., a private key).

The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource. The following example assigns 1234 to a key:

```
key_t SomeKey;  
SomeKey = 1234;
```

The `ftok()` function has the following prototype:

```
key_t ftok (  
    const char *path, /* a path string */  
    int id           /* an integer value */  
);
```

Function `ftok()` takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type `key_t` based on the first argument with the value of `id` in the most significant position. For example, if the generated integer is 35028A5D16 and the value of `id` is 'a' (ASCII value = 6116), then `ftok()` returns 61028A5D16. That is, 6116 replaces the first byte of 35028A5D16, generating 61028A5D16.

Thus, as long as processes use the same arguments to call `ftok()`, the returned key value will always be the same. The most commonly used value for the first

argument is ".", the current directory. If all related processes are stored in the same directory, the following call to `ftok()` will generate the same key value:

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

key_t  SomeKey;
SomeKey = ftok(".", 'x');
```

After obtaining a key value, it can be used in any place where a key is required. Moreover, the place where a key is required accepts a special parameter, `IPC_PRIVATE`. In this case, the system will generate a unique key and guarantee that no other process will have the same key. If a resource is requested with `IPC_PRIVATE` in a place where a key is required, that process will receive a unique key for that resource. Since that resource is identified with a unique key unknown to the outsiders, other processes will not be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.

Attaching a Shared Memory Segment to an Address Space - `shmat()` :

Suppose process 1, a server, uses `shmget()` to request a shared memory segment successfully. That shared memory segment exists somewhere in the memory, but is not yet part of the address space of process 1. Similarly, if process 2 requests the same shared memory segment with the same key value, process 2 will be granted the right to use the shared memory segment; but it is not yet part of the address space of process 2. To make a requested shared memory segment part of the address space of a process, use `shmat()`.

After a shared memory ID is returned, the next step is to attach it to the address space of a process. This is done with system call `shmat()`. The use of `shmat()` is as follows:

```
shm_ptr = shmat (
    int    shm_id,    /* shared memory ID */
    char   *ptr,      /* a character pointer */
    int    flag
);                /* access flag */
```

System call `shmat()` accepts a shared memory ID, `shm_id`, and attaches the indicated shared memory to the program's address space. The returned value is a pointer of type `(void *)` to the attached shared memory. Thus, casting is usually necessary. If this call is unsuccessful, the return value is `-1`. Normally, the second parameter is `NULL`. If the flag is `SHM_RDONLY`, this shared memory is attached as a read-only memory; otherwise, it is readable and writable.

In the following **server's** program, it asks for and attaches a shared memory of four integers.

```
/* Lab1.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int    shm_id;
key_t mem_key;
int    *shm_ptr;

mem_key = ftok(".", 'a');

shm_id = shmget(mem_key, 4*sizeof(int), IPC_CREAT | 0666);

    if (shm_id < 0)
    {
        printf("*** shmget error (server) ***\n");
        exit(1);
    }

shm_ptr = (int *) shmat(shm_id, NULL, 0); /* attach */

    if ((int) shm_ptr == -1)
    {
        printf("*** shmat error (server) ***\n");
        exit(1);
    }
```

The following is the counterpart of a **client**.

```
/* Lab2.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int    shm_id;
key_t mem_key;
int    *shm_ptr;

mem_key = ftok(".", 'a');
```

```
shm_id = shmget(mem_key, 4*sizeof(int), 0666);

if (shm_id < 0)
{
    printf("*** shmget error (client) ***\n");
    exit(1);
}

shm_ptr = (int *) shmat(shm_id, NULL, 0);

if ((int) shm_ptr == -1)
{ /* attach */
    printf("*** shmat error (client) ***\n");
    exit(1);
}
```

Note that the above code assumes the server and client programs are in the current directory. In order for the client to run correctly, the server must be started first and the client can only be started after the server has successfully obtained the shared memory.

Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the **actual address could be different** (i.e., the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

Detaching and Removing a Shared Memory Segment - `shmdt()` and `shmctl()`:

System call `shmdt()` is used to **detach a shared memory**. After a shared memory is detached, it cannot be used. However, it is still there and **can be re-attached back to a process's address space, perhaps at a different address**. To **remove a shared memory**, use `shmctl()`.

The only argument to `shmdt()` is the shared memory address returned by `shmat()`. Thus, the following code detaches the shared memory from a program:

```
shmdt (shm_ptr);
```

where `shm_ptr` is the **pointer to the shared memory**. This pointer is returned by `shmat()` when the shared memory is attached. If the detach operation fails, the returned function value is **non-zero**.

To **remove a shared memory segment**, use the following code:

```
shmctl (shm_id, IPC_RMID, NULL);
```

where `shm_id` is the shared memory ID. `IPC_RMID` indicates this is a remove operation. Note that after the removal of a shared memory segment, if you want to use it again, you should use `shmget()` followed by `shmat()`.

Program Examples:

Lab3:

Two different processes communicating via shared memory we develop two programs here that illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously:

```
/* shm_server.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSIZE 27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /* * We'll name our shared memory segment * "5678". */

    key = 5678;

    /* * create the segment. */

    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }

    /** Now we attach the segment to our data space. */

    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat");
        exit(1);
    }

    /** Now put some things into the memory for the other process to read. */
```



```

s = shm;

for (c = 'a'; c <= 'z'; c++)
    *s++ = c;
    *s = NULL;

/** Finally, we wait until the other process
 * Changes the first character of our memory
 * to '*', indicating that it has read what
 * we put there.
 */
while (*shm != '*')
    sleep(1);

exit(0);
}

```

-- Simply creates the string and shared memory portion.
 -- run it in background

```

/* shm_client.c */
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSIZE 27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*

```

```

    * Now we attach the segment to our data space.
    */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
    * Now read what the server put in the memory.
    */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
    * Finally, change the first character of the
    * segment to '*', indicating we have read
    * the segment.
    */
    *shm = '*';
    printf ("\nIts done from client.\n\n");

    exit(0);
}

-- Attaches itself to the created shared memory portion and prints the string.

```

Lab4:

Parent and Child processes communicating via shared memory

```

/*parent_child.c */

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

int main(void)
{
    int shmid;
    char *shmPtr;
    int n;

    if (fork() == 0)
    {
        sleep(5); /* UUPS */

        if( (shmid = shmget(2041, 32, 0)) == -1 )
        {

```

```

        exit(1);
    }

    shmPtr = shmat(shmid, 0, 0);
    if (shmPtr == (char *) -1)
        exit(2);

    printf ("\nChild Reading ....\n\n");
    for (n = 0; n < 26; n++)
        putchar(shmPtr[n]);
    putchar('\n'); }
else
{
    if( (shmid = shmget(2041, 32, 0666 | IPC_CREAT)) == -1 )
    {
        exit(1);
    }

    shmPtr = shmat(shmid, 0, 0);
    if (shmPtr == (char *) -1)
        exit(2);

    for (n = 0; n < 26; n++)
        shmPtr[n] = 'a' + n;
    printf ("Parent Writing ....\n\n") ;
    for (n = 0; n < 26; n++)
        putchar(shmPtr[n]);
    putchar('\n');      wait(NULL);
    shmdt(NULL);

    if( shmctl(shmid, IPC_RMID, NULL) == -1 )
    {
        perror("shmctl");
        exit(-1);
    }
}
exit(0);
}

```

-- One parent places characters in shared memory, and child reads it.

Lab5:

A program that creates a shared memory segment and waits until two other separate processes writes something into that shared memory segment after which it prints what is written in shared memory. For the communication between the processes to take place assume that the **process 1** writes **1** in first position of shared memory and waits; **process 2** writes **2** in first position of shared memory and goes on to write 'hello' and

then **process 3** writes **3** in first position of shared memory and goes on to write '**memory**' and finally the **process 1** prints what is in shared memory written by two other processes

```
/* process1.c */
```

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

```
#define SHMSIZE 27
```

```
main()
```

```
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
```

```
    /*
     * We'll name our shared memory segment
     * "5678".
     */
```

```
    key = 5678;
```

```
    /*
     * Create the segment.
     */
```

```
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
```

```
    /*
     * Now we attach the segment to our data space.
     */
```

```
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
```

```
    /*
     * Now put 1 in first place
     */
```

```
    s=shm;
    *s++='1';
    *s=NULL;
    printf("Process 1:- I have put the message %s\n",shm);
```

```
    /*
     * Finally, we wait until the other process
     */
```

```

    * changes the first character of our memory
    */
    while (*shm != '2' && *shm+6 != 'o')
        sleep(1);

    printf("Process1:- Process2 has put the message %s\n",shm);
    while (*shm != '3' && *shm+7 != 'y')
        sleep(1);

    printf("Process1:- Process3 has put the message %s\n",shm);
    printf("Process1:- I am quitting\n");
    exit(0);
}

```

/ process2.c */*

```

-
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSIZE 27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm;

    /*
    * We'll name our shared memory segment.
    */
    key = 5678;

    /*
    * Create the segment.
    */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
    * Now we attach the segment to our data space.
    */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*

```

```

    * Now put 1 in first place
    */
    *shm++='2';
    *shm++=' ';
    *shm++='h';
    *shm++='e';
    *shm++='l';
    *shm++='l';
    *shm++='o';
    *shm = NULL;
    exit(0);
}

```

/ process3.c */*

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSIZE 27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment.
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put 1 in first place

```

```

    */
    *shm++='3';
    *shm++=' ';
    *shm++='m';
    *shm++='e';
    *shm++='m';
    *shm++='o';
    *shm++='r';
    *shm++='y';
    *shm = NULL;
    exit(0);
}

```

Sample Output:

```

mammoth> gcc process3.c
mammoth> gcc process1.c -o process1
mammoth> gcc process2.c -o process2
mammoth> gcc process3.c -o process3
mammoth> process1 &
[2] 25256
mammoth> Process1:- I have put the message 1

mammoth> process2
[3] 25259
mammoth>
[3] Done process2
Process1:- Process2 has put the message 2 hello

mammoth> process3
[3] 25263
mammoth>
[3] Done process3
Process1:- Process3 has put the message 3 memory
Process1:- I am quitting

[2] Done process1

```

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual