LABORATORY REPORT

Lab 2 - Programming Lab - Sharing memory between processes

Autors :

Maxime Heurtevent
Maxence Lecoq

Lab instructor :

Khoury Christian

- Prerequisites –

- PART I. Shared Memory –

A. Understanding shmget( ), shmat( ), KEY
B. Code and Functions Explanation

- PART II. Parallel Computing -

A. Code & Output
B. Code explanation

- PART III. Implementing Copy / Paste Between Processes -

A. Using **shmget** and **shmat**
B. Using **msg.h** and **ipc.h**
C. How to run codes

Prerequisites

How to compile a program from the command line:

```
$ gcc program.c -o program
```

How to execute a program from the command line:

```
$ ./program
```

Shared memory:

Understanding shmget( ), shmat( ), KEY

Each process has its own distinct context and does not share it with other processes. Memory is where the context is and therefore, if two processes need to share information in memory, they need to create this bit of space explicitly. In this lab, you'll be experimenting with the different system functions used to create such a shared space.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <unistd.h>

#define KEY 4567
#define PERMS 0660

int main(int argc, char **argv) {
    int id;
    int i;
    int *ptr;
    system("ipcs -m");
    id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);
    system("ipcs -m");
    ptr = (int *) shmat(id, NULL, 0);
    *ptr = 54;
    i = 54;
    if (fork() == 0) {
        (*ptr)++;
        i++;
        printf("Value of *ptr = %d\nValue of i = %d\n", *ptr, i);
        exit(0);
    } else {
        wait(NULL);
        printf("Value of *ptr = %d\nValue of i = %d\n", *ptr, i);
        shmctl(id, IPC_RMID, NULL);
    }
}
```

figure 1.0 : Proposed code

What could you infer from the output regarding the state of i and ptr?

```
Value of *ptr = 55
Value of i = 55
Value of *ptr = 55
Value of i = 54
```

figure 1.1 : result of proposed code

The value of **i** is not shared between the two processes, but the value of ***ptr** is shared between the two processes.

Explain what the functions shmget(), shmat() do. What 's the need for a « KEY » here ?

- shmget( ) **creates** a **shared memory segment**. The key is used to identify the shared memory segment.
- shmat( ) **attaches** the **shared memory segment** identified by the key to the **address space** of the calling process.
- The KEY is used to **identify** the **shared memory segment**.

Code and Functions Explanation:

After **variables declaration**, we execute the command below with the system function system( ) explained in the **previous lab**:

```
$ ipcs -m
```

figure 1.2 : command to display shared memory segments

The command **ipcs -m** displays information about the shared memory segments. The output is:

```
IPC status from <running system> as of Tue Sep 19 22:15:46 CEST 2023
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 131072 0xca0f6571 --rw------- maxime_hrt    staff
m 131073 0x510fe80b --rw------- maxime_hrt    staff
```

figure 1.3 : shared memory segments

The first column is the **type** of the **IPC object**. The second column is the **ID** of the **IPC object**. The third column is the **key** of the **IPC object**. The fourth column is the **mode** of the **IPC object**. The fifth column is the **owner** of the **IPC object**. The sixth column is the **group** of the **IPC object**.

Then we **create** a **shared memory segment** with the function shmget( ). The first argument is the **key** of the shared memory segment. The second argument is the **size** of the shared memory segment. The third argument is the **flags**.

The flags are used to specify the permissions of the shared memory segment. The function **returns the ID** of the shared memory segment.

```
id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);
```

figure 1.4 : create a shared memory segment

Execute the previous command again as a proof that the shared memory segment has been created:

```
IPC status from <running system> as of Tue Sep 19 22:15:46 CEST 2023
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 131072 0xca0f6571 --rw------- maxime_hrt    staff
m 131073 0x510fe80b --rw------- maxime_hrt    staff
m 131074 0x000011d7 --rw-rw---- maxime_hrt    staff
```

figure 1.5 : new segment show

The third parameter of the function shmget() is the flags. The flags are:

- IPC_CREAT: **create** the shared memory segment **if it does not exist**.
- IPC_EXCL: **fail** if the shared memory segment **already exists**.
- IPC_NOWAIT: **fail** if the shared memory segment **is in use**.
- IPC_RW: read and write permission.

Then we **attach the shared memory segment to the address space of the calling process** with the function shmat( ). The first argument is the **ID** of the shared memory segment. The function returns the address of the shared memory segment.

```
ptr = (int *) shmat(id, NULL, 0);
```

figure 1.6 : address of shared memory

Then we write the value 54 to the shared memory segment and to the variable i.

Then we create a child process with the function fork( ). The child process increments the value of the shared memory segment and the value of the variable **i**. The parent process **waits** for the child process to terminate. Then the parent process **displays** the value of the shared memory segment and the value of the variable i. Then the parent process **removes** the shared memory segment with the function shmctl( ). The first argument is the **ID** of the shared memory segment. The second argument is the command. The third argument is the structure shmid_ds. The command is **IPC_RMID** to **remove** the shared memory segment.

Parallel Computing:

Code & Output

*Write a program that computes the following expression "(a + b) * (c + d) * (e + f)" using 3 different processes.*

```c
int affectValue(const char *message);
void handleShmatError(int *pointer, const char *message);
void handleShmctlError(int result, const char *message);

int main() {
    int id_child, id_grandChild, status;
    int a, b, c, d, e, f, result;
    int *p1, *p2;

    system("ipcs -m");
    id_child = shmget(KEY_CHILD, sizeof(int), IPC_CREAT | PERMS);
    id_grandChild = shmget(KEY_GRANDCHILD, sizeof(int), IPC_CREAT | PERMS);
    system("ipcs -m");

    p1 = (int*) shmat(id_child, NULL, 0);
    handleShmatError(p1, "SHMAT P1 ERROR");
    p2 = (int*) shmat(id_grandChild, NULL, 0);
    handleShmatError(p2, "SHMAT P2 ERROR");

    pid_t child = fork();

    // Error handling
    if (child < 0) {
        perror("child' Fork failed");
        printf("Error code: %d", errno);
    }

    if (child == 0) {
        pid_t grandChild = fork();
        if (grandChild < 0) {
            perror("'grandChild' Fork failed");
            printf("Error code: %d", errno);
        }

        if (grandChild == 0) {
            a = affectValue("Enter the \" A \" value");
            b = affectValue("Enter the \" B \" value");
            *p1 = a + b;
            printf("A + B = %d\n", *p1);
            exit(0);
        } else {
            waitpid(grandChild, &status, 0);
            c = affectValue("Enter the \" C \" value");
            d = affectValue("Enter the \" D \" value");
            *p2 = c + d;
            printf("C + D = %d\n", *p2);

            handleShmctlError(shmctl(id_grandChild, IPC_RMID, NULL), "SHMCTL ERROR
GRANDCHILD");
        }
    } else {
        waitpid(child, &status, 0);
        e = affectValue("Enter the \" E \" value");
        f = affectValue("Enter the \" F \" value");
        printf("E + F = %d\n", (e + f));
        printf("======================\n");
        printf("The result is %d\n", ((*p1) * (*p2) * (e + f)));

        handleShmctlError(shmctl(id_child, IPC_RMID, NULL), "SHMCTL ERROR CHILD");
    }
    return 0;
}
```

figure 1.7 : main code for parallel computing

```c
int affectValue(const char *message) {
    int var = 0;
    int scanResult;
    printf("%s: ",message);

    scanResult = scanf("%d", &var);
    if (scanResult != 1) {
        printf("ERROR NOT VALID INPUT");
        exit(EXIT_FAILURE);
    }
    return var;
}

void handleShmatError(int *pointer, const char *message) {
    if (pointer == (int*)-1) {
        perror(message);
        exit(EXIT_FAILURE);
    }
}

void handleShmctlError(int result, const char *message) {
    if (result == -1) {
        perror(message);
    }
}
```

figure 1.8 : annexe function for the main

Output:

```
IPC status from <running system> as of Wed Sep 20 01:30:53 CEST 2023
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 131072 0xca0f6571 --rw------- maxime_hrt    staff
m 131073 0x510fe80b --rw------- maxime_hrt    staff

IPC status from <running system> as of Wed Sep 20 01:30:53 CEST 2023
T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m 131072 0xca0f6571 --rw------- maxime_hrt    staff
m 131073 0x510fe80b --rw------- maxime_hrt    staff
m 393218 0x0000270f --rw-rw---- maxime_hrt    staff
m 262147 0x000022b8 --rw-rw---- maxime_hrt    staff

Enter the " A " value: 2
Enter the " B " value: 8
A + B = 10
Enter the " C " value: 12
Enter the " D " value: 13
C + D = 25
Enter the " E " value: 2
Enter the " F " value: 2
E + F = 4
======================
The result is 1000
```

figure 1.9 : output of the parallel computing

Code explanation

First, we **create two shared memory segments** with the function shmget( ). Then we **attach the shared memory segments** to the address space of the calling process with the function shmat( ). Then we **create a child process** with the function fork( ). The child process creates a grandchild process with the function fork( ). The grandchild process reads the values of A and B, then it **writes** the sum of A and B **to the shared memory segment**. The **child process waits** for the **grandchild process to terminate**. The child process **reads** the **values** of C and D, then it writes the sum of C and D **to the shared memory segment**. The parent process **waits** for the **child process to terminate**. The parent process **reads** the values of E and F, then it **displays** the result of the expression "(a + b) * (c + d) * (e + f)". Then the **parent process removes** the **shared memory segments** with the function shmctl( ).

Implementing Copy/Paste between processes:

*Using **shmget** and **shmat***

Write a program that implements a copy/paste mechanism between two processes.

The first process will **read a string** from the keyboard and will **write** it in a **shared memory segment**. The second process will **read** the string from the shared memory segment and will **write** it **to the screen**.

```c
// writer.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/ipc.h>

#define KEY 4568
#define MAX_SIZE 256

int main() {
    char input[MAX_SIZE];

    // Create shared memory segment
    int shmid = shmget(KEY, MAX_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    char *shared_mem = (char *) shmat(shmid, NULL, 0);

    printf("Enter your text: ");
    fgets(input, MAX_SIZE, stdin);

    strcpy(shared_mem, input);  // Copy the input to shared memory

    printf("Text saved to shared memory. You can now read it from the other process.\n");

    shmdt(shared_mem);

    return 0;
}
```

figure 1.10 : write program

First we **create** the shared memory segment with the function shmget( ). Then we **attach** the shared memory segment to the **address** space of the calling process with the function shmat( ).

Then we **read** the **input** from the keyboard with the function fgets( ). The first argument is the string where the input will be stored. The second argument is the maximum number of characters to read. The third argument is the stream to read from.

Then we **copy** the input to **the shared memory segment** with the function strcpy( ). The first argument is the **destination string**. The second argument is the **source string** (the function returns the destination string).

Then we **detach** from the shared memory segment with the function shmdt( ). The argument is the address of the shared memory segment.

```c
// reader.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <string.h>

#define KEY 4568
#define MAX_SIZE 256

int main() {
    char command[MAX_SIZE];

    printf("Type 'read' to fetch the text from shared memory: ");
    fgets(command, MAX_SIZE, stdin);
    command[strcspn(command, "\n")] = 0;  // Remove the newline character

    if (strcmp(command, "read") == 0) {
        int shmid = shmget(KEY, MAX_SIZE, 0666);
        if (shmid == -1) {
            perror("shmget");
            exit(1);
        }

        char *shared_mem = (char *) shmat(shmid, NULL, 0);

        printf("Fetched text: %s\n", shared_mem);

        shmdt(shared_mem);
    } else {
        printf("Invalid command. Exiting.\n");
    }

    return 0;
}
```

figure 1.11 : read program

First we read the command from the keyboard with the function fgets( ). If the input is "read", it will read the input from the **write** program.
Then we remove the newline character from the command with the function strcspn( ). The first argument is the **string** where the newline character will be **removed**. The second argument is the string **containing** the **characters** to **remove**. The function returns the position of the character wanted to be removed.

Then we **compare** the command with the string "read" with the function strcmp( ). The first argument is the **first string** to compare. The second argument is the **second string** to compare. The function **returns 0** if the strings are **equal**.

*Using **msg.h** and **ipc.h***

```
#include "stdio.h"
#include "sys/ipc.h"
#include "sys/msg.h"

#define MESSAGE_SIZE 256

struct message {
    long mtype;
    char mtext[MESSAGE_SIZE];
};

int main() {
    key_t key;
    int msgid;
    struct message msg;

    //Generate a unique key
    key = ftok("sender.c", 'A');

    //Get the message queue id
    msgid = msgget(key, 0666 | IPC_CREAT);


    msg.mtype = 1;
    printf("Enter a message to send to receiver: ");
    fgets(msg.mtext, MESSAGE_SIZE, stdin);

    //Send the message
    msgsnd(msgid, &msg, sizeof(msg), 0);

    printf("Message sent to receiver: %s\n", msg.mtext);

    return 0;
}
```

figure 1.12 : sender.c program

```c
#include "stdio.h"
#include "sys/ipc.h"
#include "sys/msg.h"
#include "string.h"

#define MESSAGE_SIZE 256

struct message {
    long mtype;
    char mtext[MESSAGE_SIZE];
};

int main() {
    key_t key;
    int msgid;
    struct message msg;

    //Get the key from the sender.c file
    key = ftok("sender.c", 'A');

    //Get the message queue id
    msgid = msgget(key, 0666 | IPC_CREAT);

    //Receive the message
    msgrcv(msgid, &msg, sizeof(msg), 1, 0);

    //Print the message
    printf("Message received from sender: %s\n", msg.mtext);

    //Remove the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

figure 1.13 : receiver.c program

Explanation of the msg.h and ipc.h library functions used in the code:

- ftok( ): generates a unique **key**. (from the **ipc.h** library)
- msgget( ): gets the message queue id. (from the **msg.h** library)
- msgsnd( ): sends a message. (from the **msg.h** library)
- msgrcv( ): receives a message. (from the **msg.h** library)
- msgctl( ): removes the message queue. (from the **msg.h** library)

message structure is a convention used to send messages between processes. It contains two fields:

- mtype: the type of the message.
- mtext: the content of the message.

How to run codes

To execute the code you need to **first compile the two files**, then **execute** the **receiver first** and **then the sender in a different terminal**, write a message in the sender terminal and press enter, the message will be displayed in the receiver terminal.

```
$ gcc sender.c -o sender
$ gcc receiver.c -o receiver
$ ./receiver
```

```
$ ./sender
```

figure 1.14 : execute command

```
Enter a message to send to receiver: SOSO
Message sent to receiver: SOSO
```

figure 1.15 : output sender program

```
Message received from sender: SOSO
```

figure 1.16 : output receiver program

To achieve this, we used **the message queue system**. The message queue system is a system that **allows processes to communicate** with each other by sending messages.

If we execute the **ipcs -q** between the two **executions**, we can see that the message queue is created by the sender.

```
$ ipcs -q
```

```
Message Queues:
q 262144 0x410f036c --rw-rw-rw- maxime_hrt    staff
```