ING4 SI03

Operating Systems

Lab 2 : Processes & Shared memory

Author(s) :
Romain BRISSE
Hyunjae LEE

Teacher(s) : Mr. KHOURY

I attest that this work completely comes from the author(s) mind(s). If it is otherwise you will be able to find the source in the concerned section.

Paris, 26/10/2018

LaTeX

Table des matières

0.1 Shared Memory

Question 1 : What could you infer from the output regarding the state of i and *ptr ?

```
(gdb) r
Starting program: /home/hyunjae/lab2_lee

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status


------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x000011d7 0          hyunjae    660        4          0

Child
Value of *ptr = 55
Value of i = 55
Parent
Value of *ptr = 55
Value of i = 54
[Inferior 1 (process 19) exited normally]
```

FIGURE 1 – Result from given codes

**As you can see on the result above, the values of *ptr when Child or Parent are asked are the same. However it is not the case when it comes ti the value of i. Indeed, the value of i in the parent process didn't increase. The reason why it didn't increase is simple.
In the code, we incremented the value of i while it not being attached to shared memory. So, we can easily figure out the fact that the value of "i" is considered as different variable in each process, whereas the value of *ptr is stored in the shared memory space and so can be normally accessed by both processes.
What we should remember here is that using shared memory spaces can overcome the problem caused by the duplication of data in parent and child processes. But can we not find a better solution ? As we saw in class we will be able to do the same thing using threads which should be a lot simpler to use.**

Question 2 : Read the code carefully and add your comments to all the lines

```
//definition of the access key for the future shared memory space
#define KEY 4567
//definition of access permissions for the shared memory space (in octal)
#define PERMS 0660

int main(int argc, char **argv)
{
        //definition of necessary variables
        int id;
        int i;
        int *ptr;

        //this command is used to display the state of the segments of shared memory
        system("ipcs -m");
        /*
         * creation of the shared memory space:
         *      - Key is it's identifier
         *      - sizeof(int) represents the size of the memory space
         *      - IPC_CREAT is the keyword indicating we are creating the shared memory space
         *      - PERMS represents the permissions defined earlier
         */
        id = shmget(KEY, sizeof(int), IPC_CREAT | PERMS);

        system("ipcs -m");

        /*
         * then, we relocate the pointer into the share memory space:
         *      - id is the name of the shared memory space in which we relocate
         *      - NULL  means the system will choose himself the address in the shared space
         *      - 0 means there are no flags attached to this declaration
         */
        ptr = (int *) shmat(id, NULL, 0);
        //then, intialize the values of ptr and i
        *ptr = 54;
        i = 54;

        //fork, creating a child process
        if (fork() == 0)
        {
                //in the child process: increment the values of i and ptr
                (*ptr)++;
                i++;
                printf("Value of *ptr = %d\nValue of i = %d\n", *ptr, i);
                exit(0);

        }
        else
        {
                //in the parent process, wait for the child and print tha values of i and ptr.
                wait(NULL);
                printf("Value of *ptr = %d\nValue of i = %d\n", *ptr, i);
                shmctl(id, IPC_RMID, NULL);
        }
}
```

0.2 Parallel Computing

0.2.1 Using - wait(NULL) ; -

```c
#define KEY 4567
#define PERMS 0660

int main(int argc, char **argv)
{
	int id; int *a;

	//creation of the shared memory space
	id = shmget(KEY,6*sizeof(int),IPC_CREAT | PERMS);

	//reloacating the array a in the shared memory space
	a = (int*) shmat(id,NULL,0);
	//initialize the values in the array
	a[0]=1;
	a[1]=2;
	a[2]=3;
	a[3]=4;
	a[4]=5;
	a[5]=6;

	//fork a first time to create process p2
	if(fork()==0)
	{
		//fork a second time to create process p3
		if(fork()==0)
		{
			//execute first step of calculus
			a[0]=a[0]+a[1];
			printf("a+b = %d\n",a[0]);
		}
		else
		{
			//execute second part of calculus
			a[2]=a[2]-a[3];
			printf("c-d = %d\n",a[2]);
			//wait for the first part of calculus to have ended
			wait(NULL);
			//then do the fourth part of calculus
			a[0]=a[0]*a[2];
			printf("a*c = %d\n",a[0]);
		}
	}
	//this is process p1
	else
	{
		//execute third par of calculus
		a[4]=a[4]+a[5];
		printf("e+f = %d\n",a[4]);
		//wait for the fourth part of calculus to have ended
		wait(NULL);
		//then do the fifth of calculus
		a[0]=a[0]+a[4];
		printf("result: %d\n",a[0]);
	}
```

FIGURE 2 – parallel computing using wait instructions

0.2.2 Using flags

```c
/* Program resolving the question 2.2 of lab 2
 * Authors are: Romain Brisse & Hyunjae Lee
 * All rights of diffusion are reserved to the authors.
 */

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

#define KEY 4567
#define PERMS 0660
#define KEYBIS 1234

int main(int argc, char **argv)
{
        int id;
        int *a;

        //creation of the shared memory space regarding the calculus
        id = shmget(KEY,6*sizeof(int),IPC_CREAT | PERMS);

        //relocating the a array in the shared memory space
        a = (int*) shmat(id,NULL,0);
        //initialize the values in my array
        a[0]=1;
        a[1]=2;
        a[2]=3;
        a[3]=4;
        a[4]=5;
        a[5]=6;

        int flag;
        int *b;

        //creation of the shared memory sapce regarding the flags
        flag = shmget(KEYBIS,2*sizeof(int),IPC_CREAT | PERMS);
        //reloacting the b array in the shared memory space
        b = (int*) shmat(flag,NULL,0);

        //initialize the values in the array
        b[0]=0;
        b[1]=0;
```

FIGURE 3 – parallel computing using flags

```c
	//fork a first time to create process p2
	if(fork()==0)
	{
		//fork a second time to create process p3
		if(fork()==0)
		{
			//execute the first step of calculus
			a[0]=a[0]+a[1];
			//change the value of the first boolean to true
			b[0]=1;
		}
		else
		{
			//execute the second part of the calculus
			a[2]=a[2]-a[3];
			//wait for the first boolean to change its value indicating that the first step of the calculus has ended
			while(!b[0]){}
			//then execute the fourth part of the calculus
			a[0]=a[0]*a[2];
			//and change the value of the second boolean
			b[1]=1;
		}
	}
	//this is process p1
	else
	{
		//execute third part of process
		a[4]=a[4]+a[5];
		//wait for the second boolean to change its value indicating that the fourth part of the calculus has ended
		while(!b[1]){}
		//then execute the fifth part of the calculus
		a[0]=a[0]+a[4];
		//print the result
		printf("result: %d\n",a[0]);
	}
}
```

Figure 4 – parallel computing using flags

0.2.3 conclusion

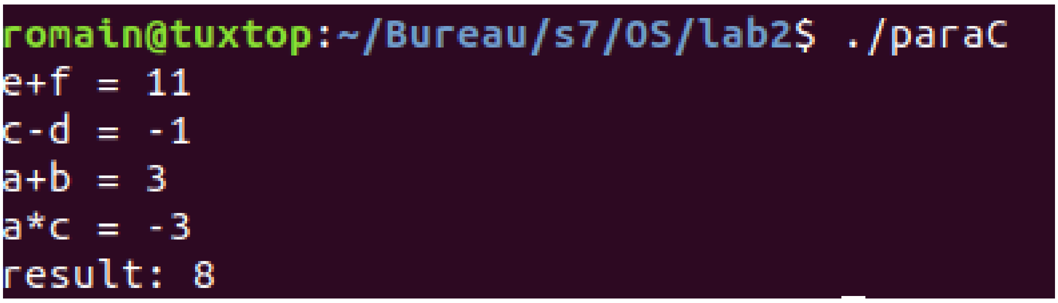

FIGURE 5 – Result

In both the previous programs, there are 3 processes : a first generation parent, a first generation child and a second generation child. First of all, the parent process (p1) creates a child (p2) which then creates its own child (p3). That way we are able to obtain three processes running at the same time and that will be able to wait for one another. In order to do this, we use 'Wait(Null)', or flags that makes a parent process wait for its child to finish all he is doing before going on. Now, as the results showed, we are able to execute the steps of the calculus concurrently and in an order that will not render it false.
To conclude, parallel computing can be achieved without many difficulties using the fork() function, but there is the problem of the shared memory to deal with since parent and child processes do not share data. Also, it is possible that programs using fork many times in the same file get a lot more difficult to understand and to manage. Finally, at our scale of testing, using flags or 'wait(NULL)' does not have much of an impact but using empty while loops, and more shared memory for said flags is not a really clean solution. However the wait(NULL) instruction also has its flaws since it only allows a process to wait for its child !