TARBE Théophile

QUIDET Victor

ING4 - GR01

LAB 5 :

Interprocess Synchronization

Operating Systems

Mr KHOURY

I - Concurrent Access To Shared Memory : Race Problems

1. Code with race problems

Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part1.c > ...

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

int i = 65;

void *child(void *arg){
    printf("Child before : %d\n", i);
    int reg = i;
    sleep(2);
    ++reg;
    i = reg;
    printf("Child after (++i) : %d\n", i);
    return NULL;
}

void *parent(void *arg){
    printf("Parent before : %d\n", i);
    int reg = i;
    sleep(2);
    --reg;
    i = reg;
    printf("Parent after (--i) : %d\n", i);
    return NULL;
}

int main (int argc, char *argv[]) {

    int iret1, iret2;
    pthread_t thread1, thread2;

    printf("initial value i = %d\n", i);

    //printf("child: begin\n");
    iret1 = pthread_create(&thread1, NULL, child, NULL);
    if(iret1) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        exit(EXIT_FAILURE);
    }

    //printf("parent: begin\n");
    iret2 = pthread_create(&thread2, NULL, parent, NULL);
    if(iret2) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        exit(EXIT_FAILURE);
    }

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

    printf("final value : i = %d\n", i);

    return 0;
}
```

2. Results

```
[→  lab5 gcc -g -o ex1bis part1bis.c
[→  lab5 ./ex1bis
initial value i = 65
Child before : 65
Parent before : 65
Child after (++i) : 66
Parent after (--i) : 64
final value : i = 64
```

In this exemple, we can observe **critical race conditions** because the threads depend on the same shared value of "`i`". The two operations we carry out on variable "`i`" are **not mutually exclusive**. It means that there is **no restriction to access to the shared data** and **operations can be interrupted** while accessing the segment of shared memory. As a result, the **execution is invalid** and we get a wrong final result.

Here is what happens in the code with the shared data (inspired by Wikipedia, see III - Sources) :

Thread 1	Thread 2		Integer value
			65
read value		←	65
	read value	←	65
increase value			66
	decrease value		64
write back		→	66
	write back	→	64
final value		→	64

II - Solving the Problem : Synchronizing access using semaphores

1. Solving race problem

Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-1.c > ...

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

sem_t * s;
int i = 65;

void *mythread1(void *arg){
    printf("thread1 before : %d\n", i);
    ++i;
    printf("thread1 after (++i) : %d\n", i);
    sem_post(s);
    return NULL;
}

void *mythread2(void *arg){
    sem_wait(s);
    printf("thread2 before : %d\n", i);
    --i;
    printf("thread2 after (--i) : %d\n", i);
    return NULL;
}

int main (int argc, char *argv[]) {

    int iret1, iret2;
    pthread_t thread1, thread2;

    s = sem_open("sem", O_CREAT, S_IRUSR | S_IWUSR, 1);

    printf("initial value i = %d\n", i);

    iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
    if(iret1) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        exit(EXIT_FAILURE);
    }

    iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
    if(iret2) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        exit(EXIT_FAILURE);
    }

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

    printf("final value : i = %d\n", i);

    return 0;
}
```

Results

```
[→  lab5 gcc -g -o ex21 part2-1.c
[→  lab5 ./ex21
initial value i = 65
thread1 before : 65
thread1 after (++i) : 66
thread2 before : 66
thread2 after (--i) : 65
final value : i = 65
```

The race condition problem has been solved using semaphores which allows the first `thread` to be fully executed before the second one is launched using `sem_wait()` and `sem_post()` :

- 1st : `sem_post()` unlocks the semaphore pointed by sem. The value of the semaphore becomes greater than zero (process1)
- 2nd : `sem_wait()` will be woken up and proceed to lock the semaphore (process2)

If we had more than 2 processes, we should determine an order of execution for each of them in order to be executed one after another. If not, we **could have a deadlock situation** in which a process is waiting for some resource held by another process waiting for it to release another resource.

That is why we can **use lock to enforce a mutual exclusion concurrency control policy**.

2. Force a deadlock situation using semaphores (and 3 threads)

Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-2.c > ...

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

sem_t * mutex1;
sem_t * mutex2;
sem_t * mutex3;
int i = 65;

void *mythread1(void *arg){
    sem_wait(mutex1);
    sleep(2);
    sem_wait(mutex2);
    printf("T1 before : %d\n", i);
    ++i;
    printf("T1 after (++i) : %d\n", i);
    sem_post(mutex2);
    sem_post(mutex1);
    return NULL;
}

void *mythread2(void *arg){
    sem_wait(mutex2);
    sleep(2);
    sem_wait(mutex3);
    printf("T2 before : %d\n", i);
    --i;
    printf("T2 after (--i) : %d\n", i);
    sem_post(mutex3);
    sem_post(mutex2);
    return NULL;
}

void *mythread3(void *arg){
    sem_wait(mutex3);
    sleep(2);
    sem_wait(mutex1);
    printf("T3 before : %d\n", i);
    i+=10;
    printf("T3 after (i+=10) : %d\n", i);
    sem_post(mutex1);
    sem_post(mutex3);
    return NULL;
}
```

```c
int main (int argc, char *argv[]) {

    int iret1, iret2, iret3;
    pthread_t thread1, thread2, thread3;

    mutex1 = sem_open("mutex1", O_CREAT, S_IRUSR | S_IWUSR, 1);
    mutex2 = sem_open("mutex2", O_CREAT, S_IRUSR | S_IWUSR, 1);
    mutex3 = sem_open("mutex3", O_CREAT, S_IRUSR | S_IWUSR, 1);

    printf("initial value i = %d\n", i);

    iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
    if(iret1) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        exit(EXIT_FAILURE);
    }

    iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
    if(iret2) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        exit(EXIT_FAILURE);
    }

    iret3 = pthread_create(&thread3, NULL, mythread3, NULL);
    if(iret3) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret3);
        exit(EXIT_FAILURE);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    printf("final value : i = %d\n", i);

    return 0;
}
```

Results

```
[→  lab5 gcc -g -o ex22 part2-2.c
[→  lab5 ./ex22
initial value i = 65
```

In this case **we created a deadlock situation using semaphores**, because each thread is waiting for another to be unlocked (thread 2 for thread 1 / thread 3 for thread 2 / thread 1 for thread 3).

It is forming a **loop of blocked processes**. Therefor the compilation is endless and won't stop until we manually do it:

```
[→  lab5 gcc -g -o ex22 part2-2.c
[→  lab5 ./ex22
initial value i = 65
^C
→  lab5
```

3. **Run 3 different applications**

```c
sem_t * s2, *s3;

void *firefox(){
        if (fork() == 0){
            execlp("firefox", "firefox", NULL);
        }
        sem_post(s2);
    }

void *vi(){
        sem_wait(s2);
        if (fork() == 0){
            execlp("vi", "vi", NULL);
        }
        sem_post(s3);
    }

void *emacs(){
        sem_wait(s3);
        if (fork() == 0){
            execlp("emacs", "emacs", NULL);
        }
    }

int main (int argc, char *argv[]) {

    pthread_t p1, p2, p3;

    s2 = sem_open("s2", O_CREAT, S_IRUSR | S_IWUSR, 0);
    s3 = sem_open("s3", O_CREAT, S_IRUSR | S_IWUSR, 0);

    pthread_create(&p1, NULL, firefox, NULL);
    pthread_create(&p2, NULL, vi, NULL);
    pthread_create(&p3, NULL, emacs, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);

}
```

4. Implement a parallelized calculation with semaphores

Users > theophiletarbe > Desktop > cours > ING4 > OS > lab5 > C part2-4.c > main(int, char * [])

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

//(a+b)*(c-d)*(e+f)

sem_t * mutex1;
sem_t * mutex2;
sem_t * mutex3;
int a=10, b=5, c=7, d=2, e=1, f=1;
int count = -1; // counter for the number of results in : results[]
int results[3]; // results table

void *mythread1(void *arg){

    sem_wait(mutex3);
    count++;
    results[count] = (a+b);
    printf("T1: (a+b) = %d\n", results[count]);

    if (count == 1) // get 2 results in the table
    {
        sem_post(mutex1); // unlock mutex1 in mythreads4
    }
    else if (count == 2) // get 3 results in the table
    {
        sem_post(mutex2); // unlock mutex2 in mythreads4
    }
    else // get 0 or 1 result : call 1 of the 2 remaining threads
    {
        sem_post(mutex3);
    }
    return NULL;
}
```

```c
void *mythread2(void *arg){

    sem_wait(mutex3);
    count++;
    results[count] = (c-d);
    printf("T2: (c-d) = %d\n", results[count]);

    if (count == 1)
    {
        sem_post(mutex1);
    }
    else if (count == 2)
    {
        sem_post(mutex2);
    }
    else
    {
        sem_post(mutex3);
    }
    return NULL;
}

void *mythread3(void *arg){

    sem_wait(mutex3);
    count++;
    results[count] = (e+f);
    printf("T3: (e+f) = %d\n", results[count]);

    if (count == 1)
    {
        sem_post(mutex1);
    }
    else if (count == 2)
    {
        sem_post(mutex2);
    }
    else
    {
        sem_post(mutex3);
    }
    return NULL;
}

void *mythread4(void *arg){

    sem_wait(mutex1);
    results[0] = results[0]*results[1]; // stock the 1st result (btw the 2 first threads) in the table
    printf("T4: First Result (T1*T2) = %d\n", results[0]);

    sem_post(mutex3);

    sem_wait(mutex2);
    results[0] = results[0]*results[2]; // stock the final result
    printf("T4: Final Result (T1*T2*T3) = %d\n", results[0]);

    return NULL;
}
```

```c
int main (int argc, char *argv[]) {

    int iret1, iret2, iret3,iret4;
    pthread_t thread1, thread2, thread3, thread4;

    // Initialization to 0 : waiting for the 3 other threads to call sem_post()
    // and increment the value of semaphores from 0 to 1
    mutex1 = sem_open("mutexx1", O_CREAT, S_IRUSR | S_IWUSR, 0);
    mutex2 = sem_open("mutexx2", O_CREAT, S_IRUSR | S_IWUSR, 0);
    // Initialization to 1 : waiting for 1 of the 3 threads to call sem_wait()
    // and decrement the value of semaphores from 1 to 0
    mutex3 = sem_open("mutexx3", O_CREAT, S_IRUSR | S_IWUSR, 1);

    iret1 = pthread_create(&thread1, NULL, mythread1, NULL);
    if(iret1) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        exit(EXIT_FAILURE);
    }

    iret2 = pthread_create(&thread2, NULL, mythread2, NULL);
    if(iret2) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        exit(EXIT_FAILURE);
    }

    iret3 = pthread_create(&thread3, NULL, mythread3, NULL);
    if(iret3) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret3);
        exit(EXIT_FAILURE);
    }

    iret4 = pthread_create(&thread4, NULL, mythread4, NULL);
    if(iret4) {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret4);
        exit(EXIT_FAILURE);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    pthread_join(thread4, NULL);

    return 0;
}
```

Results

```
➜  lab5 gcc -g -o ex24 part2-4.c -lpthread
➜  lab5 ./ex24
T1: (a+b) = 15
T2: (c-d) = 5
T4: First Result (T1*T2) = 75
T3: (e+f) = 2
T4: Final Result (T1*T2*T3) = 150
➜  lab5
```

In this exemple, we used **4 differents threads with 3 semaphores** in order to synchronize them as we want and compute the operation:

- The first 3 threads separately compute a part of the operation and use 1 semaphore (`mutex3`) in order to prevent a race problem. Moreover, the conditions on the counter determine how many operations the threads have already done and unlock `mutex1` or `mutex2` in consequence.
- The last thread realises the 2 main operations from the results of T1, T2 and T3.

The `main()` initialize `mutex1` & `mutex2` at 0, and `mutex3` at 1 so that we have **the desired order of execution** of the threads (see *"31.3 Semaphores For Ordering"* in the course).

III - Sources

https://en.wikipedia.org/wiki/Race_condition

https://medium.com/swlh/race-conditions-locks-semaphores-and-deadlocks-a4f783876529

Course : Semaphores (on Campus)