Université de Liège

Computation Structures

Project 2 «Bounce» : report

3rd bachelor of engineering

Authors :
Antoine Louis
Tom Crasset

Professor :
P. Wolper

Academic year 2017-2018

0.a Shared memory

. We decided to format the shared memory so that it could contain a *point* structure, which we defined to pass in the coordinates of each square. We also have two boolean values, *finish* and *allUpdated*. The first one is used to signal to the program that the user wants to quit using the 'Enter' key and the second one is used to let the workers know that they all have updated their position. They both were encoded in the *.x* field of the *point* structure and in shared memory they occupy the first and the last position respectively. The rest of the shared memory is dedicated to store the position of each square and its state, i.e whether the position of one square is updated or not. Quick illustration : [finish, square1, square2, square3, ...,isUpdated1,isUpdated2,...,allUpdated].

0.b Semaphores

For semaphores, we have a total of 2 sets of mutex and 3 mutex by themselves. The 2 sets are :

1. *workers_semid*[SQUARE_COUNT]
2. *collision_semid*[SQUARE_COUNT]

The first set is used to signal to each *worker* that it can resume its execution mutex . Each *worker* will receive a signal from *master*. The other set is used by one *worker* to signal a collision with another.

The 2 stand-alone mutex are :

1. *posUpdated_semid*[1]
2. *access_semid*[1]
3. *control_semid*[1]

The first one is placed in a loop in the *master process* and acts as a counter that counts the number of squares that have updated their position. When that number hits SQUARE_COUNT it means that all workers have updated their position and thus the *master process* can update the grid. The second mutex is used to restrict the access to the shared memory to only one *worker* at a time. The last one is a special, one time use mutex where the control process will wait on for the *master* to finish it's last iteration. After that, the control process cleanly exits the program.

```
/*****************SHARED MEMORY***********/
point finish;
point square_position[SQUARE_COUNT]; //To store position of the
    squares
point isUpdated[SQUARE_COUNT];
point allUpdated;
/*****************SEMAPHORE SETS***********/
workers_semid[SQUARE_COUNT] : {0,0,...,0}
posUpdated_semid[1] = 0;
access_semid[1] = 0;
collision_semid[SQUARE_COUNT] : {0,0,...,0}
control_semid[1] : 0
/*****************MESSAGE QUEUES***********/
msgq_id[1];
struct mymsgbuf {
        long type;
        int sender;
struct speed_s speed;
};

struct speed_s{
  int speed_x;
  int speed_y;
};
```

0.c Master process

We subdivided the *master process* into one *control process* that will delete all elements allocated and cleanly quit the program and one *master process* that will manage the *workers*. Upon entering the *master process*, the mutex giving the access to the shared memory is signaled once, thus giving access to one *worker* at random. This is only executed once and from here on the *workers* lock and unlock this mutex by themselves. Then, we enter a while loop that will loop until the user wishes to quit the program.
When the user wishes to stop the program, the *master* will cycle one last time, exit the while loop and then will signal a semaphore that will tell the *control process* to delete all the elements.»

Cycle

Next, line 7, we have a mutex inside a while loop that will act as a counter to count the number of *workers* having updated their position. When that is the case, we change the bool value *allUpdated*, which is in shared memory, to true.
This is useful when we next release all the workers that were stuck on the *collision_semid* semaphore because then they won't enter the while loop(line 38 of *workers*) and they will be waiting to be re-executed on the *workers_semid* semaphore.
Once the grid has been updated, we set the *isUpdated* tags of all the *workers* and also *allUpdated* to false. Finally, we unlock every *worker*, thus starting another cycle.

```
master_process(point* segptr,int SQUARE_COUNT,semaphore_Ids) {
        bool allUpdated,isUpdated;
        //Give access to the shared memory
        signal(access_semid);
        while(!<ENTER>){
                //Wait before all workers have updated their position
                for(int cntr = 0, cntr < SQUARE_COUNT ; cntr++)
                        wait(posUpdated_semid);

                //Set allUpdated to true in shared memory
                allUpdated = true;

                //Unlock all semaphores waiting for collision
                for(id = 1; id <= SQUARE_COUNT; id++)
                        signal(collision_semid[id-1]);

                updatingPixels();

                //Set isUpdated tag in shared memory for every worker
                    to false
                for(id = 1; id <= SQUARE_COUNT; id++)
                        isUpdated[id] = false;

                //Set allUpdated to false in shared memory
                allUpdated = false;

                //Wait a bit
                usleep(15000);

                        //Unlock all the workers
                for(id = 1; id <= SQUARE_COUNT; id++)
                        signal(workers_semid[id-1]);
        }
        wait(control_semid)
}
```

```
control_process(point* segptr, int SQUARE_COUNT,semaphoreIDs,msgq_id,
    shmid){

    point finish = {.x = 1};
    bool stop = false;
    //Loop until the users wishes to exit
    // by pressing <ENTER>
    while(!stop){
        if(kbhit())
            //Change the loop condition
            stop = true;
            //Set finished to true in shared memory
            finish = true;

            //One last master process interation
            for(int id = 1; id <= SQUARE_COUNT; id++)
                unlocksem(posUpdated_semid,0);

            // Wait for the master process to finish its last
               iteration
            locksem(control_semid,0);
```

```
20              // Close messages queues
                remove_queue(msgq_id);
22              // Close shared memory
                remove_shm(shmid);
24              //Close semaphores
                remove_sem(workers_semid);
26              remove_sem(access_semid);
                remove_sem(posUpdated_semid);
28              remove_sem(collision_semid);
                remove_sem(control_semid);
30      }
}
```

0.d Worker process

The *worker process* has the same while loop encompassing the whole code as the *master process*. In this loop, most of the *workers* are stuck waiting on the access to the table. Once one *worker* has gained access to it, he retrieves his position, computes the next one and checks if he doesn't hit any walls.

Next, that *worker* is going to check if he collides with any other *worker* (line 15), but only *workers* that have already updated their position. That is because there can be a case when both are travelling in the same direction, hugging each other. If the one updating it's position is behind the other, he might think that he will collide at the next cycle but that isn't the case because the front one still needs to update his position.
If he collides, he is going to send his speed to the one he is colliding with and will also receive the other one's speed. We also had the case that two *workers* became stuck together when they had the same speed, so we added a condition that if the *worker* issuing the collision has the same speed as the other one, he will just take the opposite speed.
We still restart the loop in case there are multiple collisions per *worker*. Finally, that *worker* is going to use that speed and compute his new position and store it in the table and set his *isUpdated* tag in shared memory to true.

Then he gives away his access to shared memory and also signals to the *master process* that he has updated his position. The worker is now stuck on the *collision_semid*, and is waiting for either a collision from another *worker* or from a signal from the*master process* if all squares have updated their position.
If there is a collision with a *worker*, the other *worker* will signal him that, like we said above, and the current *worker* will enter the while loop, wait for a message, send his speed, update his speed and then wait again on the *collision_semid*.

```
worker(int id,point* segptr,int SQUARE_COUNT,semaphore_Ids,int
    msgq_id,int speed) {
        point next_pos;
        point current_pos;

        while(!<ENTER>){
                //Mutex for the shared memory
                wait(accessPositionTable)

                getCurrentPosition();
                ComputeNextPosition();
                CheckOutOfBounds();

                //checkCollisionSender
                for(int other_id = 1; other_id <= SQUARE_COUNT;
                    other_id++){
                        if(hasIntersection(id,other_id))
                                if position_updated[other_id] == true
                                        signal(collision_semid[
                                            other_id-1])
                                        // Send my speed
                                        msgq_id!(other_id,mySpeed)
                                        // Receive other speed
                                        msgq_id?(id,otherSpeed)
                                        if(mySpeed == otherSpeed);
                                                mySpeed *= -1;
                                        else
                                                mySpeed = otherSpeed;
                                        updatePosition();
                                        other_id = 0;
                                        continue;
                        }

                                store_position()
                isUpdated[id] = true;  //in shared memory
                //Give access to the shared memory to someone else
                signal(accessPositionTable);
                //Signal to the master that you are updated
                signal(posUpdated_semid);

                //checkCollisionReceiver

                // Wait for the first collision
                wait(collision_semid[id-1])
                while(!allUpdated)
                        //Receive other speed
                        msgq_id?(id,otherSpeed)
                        // Send my speed
                        msgq_id!(other_id,mySpeed)
                        mySpeed = otherSpeed;
                        // wait for subsequent collisions
                        wait(collision_semid[id-1]);

                // Wait for the master process
                wait(workers_semid[id-1]);
        }
}
```