Computation Structures

Rapport du projet 2

Pierre Hockers Olivier Moitroux
3e bac ingénieur civil

18 décembre 2017

1 Explication de notre logique

Le code se doit d'être exécuté avec un paramètre en console qui n'est autre que la nombre de carrés souhaités. N'ayant reçu strictement aucune consigne à propos du nombre d'itérations, nous choisissons que la seule manière d'arrêter notre programme, sera tout simplement d'appuyer sur une touche au clavier.

Au tout début, la fonction main() initialise tout. Elle demande à l'utilisateur si il souhaite une initialisation automatique (aléatoire) des carrés ou bien une initialisation manuelle. Cette fonction gère donc les inputs, place les carrés dans la table de pixels après l'avoir initialisée à 0, met à jour l'interface graphique et initialise tout les autres variables dont nous avons besoin.

Ensuite nous créons autant de `worker` process (WP) qu'il y a de carrés. De plus, nous créons un process chargé de gérer la fin du progamme. Appelons-le `controler` (CP). Il ne fait que vérifier que l'on n'entre rien en console et sinon, après avoir ordonné que tous les autres process se terminent, supprime les espaces mémoires utilisés pour la gestion du multi-cœur.

Dans le `master` process, nous envoyons un signal (via la sémaphore *compute*) à chaque WP pour qu'il calcule sa nouvelle position basée sur son ancienne position et sur sa vitesse actuelle. Le MP attend de recevoir leur confirmation qu'ils ont bien calculé leur nouvelle position, en ayant déjà tenu compte des possibles collisions avec les murs. Cela est fait via les sémaphores *wall_computed*. Les WP ont dans la foulée mis à jour les vecteurs X et Y contenant leurs positions dans la shared memory. Comme ils ne modifient pas les mêmes emplacements du vecteur, ceci n'est pas protégé.

Une fois cela fait, le MP reprend la main et met à jour *priorityQueue*, un vecteur contenant les ID des WP ordonnés selon leur priorité. Un square a la priorité sur un deuxième quand il est plus haut que ce deuxième dans la grille. Le cas échéant, une égalité de hauteur est départagée par un critère horizontal : le plus à gauche l'emporte. Une fois ce vecteur mis à jour, le MP envoie un signal via le vecteur de sémaphores *Check_for_collisions* au WP de priorité maximale, donc celui d'indice 0 dans *priorityQueue*. Ce worker va donc se réveiller.

De là, deux possibilités :

- Soit le square du WP n'était entré en collision avec aucun autre et donc rien ne se passe et nous débloquons (*signal(check_for_collision[id]))*) le WP suivant dans la priorityQueue.
- Soit le square du WP en cours d'exécution est entré en collision avec un autre square (qui est donc forcément de priorité inférieure). Dans ce cas, le WP prioritaire envoie par *message queue* ses vitesses au WP en collision. Pour plus de clarté, nous allons nommer le WP prioritaire WP1 et l'autre WP2. WP1 va envoyer ses vitesses par message queue à WP2. Il va ensuite lui envoyer un *signal* par la sémaphore *check_for_collision* lui correspondant afin de le réveiller. WP1 va ensuite attendre la réponse par message queue du WP2. WP2, une fois réveillé, va entrer dans le *else if*, étant en collision sans être prioritaire. Il va donc stocker les vitesses obtenues par la message queue avant d'envoyer les siennes (un autre type de message queue) et de les mettre à jour.

  Ensuite il va réveiller WP1 par la sémaphore habituelle et attendre qu'on le réveille de nouveau par *check_for_collision*. On fait cela pour éviter qu'il ne réveille le WP suivant par ordre de priorité tout de suite, instruction qui vient après le else if. Nous revenons donc au WP1, qui lit ses nouvelles vitesses et vérifie qu'il n'est plus en collision. S'il l'est encore, les deux squares allaient l'un vers l'autre et se recouvraient de 2 pixels, et non de un. On sort alors du if, on n'est pas concerné par le else if et on arrive a la dernière instruction, qui va juste envoyer un signal *check_for_collisions* au WP suivant par ordre de priorité.

Le processus va alors se répéter, les workers (dont l'id est stocké *priorityQueue*) vont un à un être visités et réveillés. Il viendra un moment ou nous retomberons sur un WP qui avait été réveillé en tant que moins prioritaire, mais le travail est déjà fait : il a réglé sa collision. Il n'a plus qu'à réveiller le suivant. Comme il n'y a jamais deux WP actifs en même temps dans cette section critique, aucun risque que deux d'entre eux jouent avec les mêmes données sensibles en même temps. Quand nous arriverons au dernier WP selon l'ordre

de *priorityQueue*, on renvoie un *signal(end_check_position)* au MP.

Une fois que le MP est réveillé par un *end_check_position*, il met à jour la table de pixels et l'affiche.

2 Pseudo Code

```
struct square_t{
    shared_t int* x;
    shared_t int* y;
    int color;
    int speedx;
    int speedy;
};
#define SQUARE_COUNT 5 // exemple
typedef struct square_t square;

 shared_t int x[SQUARE_COUNT];
 shared_t int y[SQUARE_COUNT];
 shared_t bool stop = 0;
 shared_t int priorityList[SQUARE_COUNT];

 semaphore_t exited[SQUARE_COUNT] = 0;
 semaphore_t compute[SQUARE_COUNT] = 0;
 semaphore_t wall_computed[SQUARE_COUNT] = 0;
 semaphore_t check_for_collisions[SQUARE_COUNT];
 semaphore_t end_check_collisions;


/* ----------
* MAIN_PROCESS: wait for worker completion for the current tick.
*               Only when workers have computed the position for tick T
*               can the master process update the display.
* ---------- */
MAIN_PROCESS{

    Initialize_everything();     // Initialize table of pixels, squares,...

    update_table_of_pixels(x,y);
    display(table_of_pixels);

    SQUARE_COUNT * create(worker_process, id = [0:SQUARE_COUNT-1], (struct square_t) square
    // WORKERS(square); where color, speedx, speedy, x, y stored in the struct. square


    1 * create(process_to_quit_properly, id = SQUARE_COUNT);
    // EXIT_PROCESS

    // MASTER PROCESS :
    while(stop == 0 ){
                    // Launch signal for workers
        signal each(compute[id]);      // id = [0:SQUARE_COUNT-1]
        wait each(all_computed[id]);   // id = [0:SQUARE_COUNT-1]
```

```
        // vector of IDs classed by order of priority (height and width based)
        priorityList = updatePriority(x[], y[]);

        signal(CheckForPosition[ priorityList[0] ]);
        wait(end_check_collisions);

        update_table_of_pixels(x,y);
        display(table_of_pixels);
    }
   signal(exited);
}

/*----------

WORKERS_PROCESS(id) : calculate the next position of its square.

*----------*/
WORKER_PROCESS(square obj){

    while(stop == 0){

        wait(compute);
        // [moving x]
        obj.x += obj.speedx;

        if(obj.x > SIZE_X - SQUARE_WIDTH){ // if right bound encountered
            obj.x =  SIZE_X - SQUARE_WIDTH;
            obj.speedx = -1;                        // back off
        }
        if(obj.x < 0){ // if left bound encountered
            obj.x = 0;
            obj.speedx = 1;                                 // back Off
        }
        // [moving y]
        obj.y = obj.speedy;
        if(obj.y > SIZE_Y - SQUARE_WIDTH){ // lower bound encountered
            obj.y =  SIZE_Y - SQUARE_WIDTH;
            obj.speedy = -1;                          // back off
        }
        if(obj.y < 0){ // upper bound encountered
            obj.y = 0;
            obj.speedy = 1;      // bak off
        }

        update(x[], y[]);

        // we are done checking for collisions with walls with this square
        signal(wall_wall_computed[id]);

        wait(check_for_collisions[id]);

        if ("this square collided with another square of ID id2 AND has a higher priority")
```

```
            postToqueue1([obj.speedx, obj.speedy]); // q! first

            signal(check_for_collisions[id2]);  // we wake id2 up

            [speedx, speedy] = readFromQueue2(); // q? second

            if(hasIntersection(obj, x[], y[]) with the same square id2){
              // they overlapped more than one pixel
              obj.x -= obj.speedx;
              obj.y -= obj.speedy;
            }

            obj.speedx = speedx;
            obj.speedy = speedy;
            // call next worker in priority
        }
        else if (" Intersection BUT with a square of lower priority "){
            // id2(t-1) is named id in this case and id(t-1) is now id2. Not the same
            // name as in the if, because, it's no more the same process.
            [speedx, speedy] = readFromQueue1(); // q? first
            obj.x -= obj.speedx;
            obj.y -= obj.speedy;

            postToqueue2([obj.speedx, obj.speedy]); // q! second

            obj.speedx = speedx;
            obj.speedy = speedy;

            wait(check_for_collisions[id]); // we'll come back here later

        }

        if ("not the end of priority queue")
            signal(check_for_collisions[find(id, priorityList) + 1 ]); // next id in prior

        else signal(end_check_collisions);
    }
    signal(exited);
}

/*----------
EXIT_PROCESS(id) == Controler
*----------*/
EXIT_PROCESS{
    // this process is blocked on getchar()
    getchar(); // kbhit()
    stop = 1;
    for(int id = 0; id < SQUARE_COUNT; ++id){
        wait(exited[id]);
    }
    delete_semaphores();
    delete_shared_memory();
    delete_queues();
    return EXIT_SUCESS;
```

}

3 Résultat

Notre code permet de gérer les carrés comme escompté. Les collisions murales et inter-squares réagissent correctement, et nous avons même pu assister à une triple collision fructueuse.

Quelques remarques cependant s'imposent :

1. Premièrement, il arrive de temps en temps que le rebond ne paraisse pas tout à fait physique ... mais c'est tout autant le cas dans la version single core.
2. Deuxièmement, il semble que le code n'arrive pas à afficher plus de 4 carrés à l'écran.
3. Enfin, notre process `controler` nous donne encore du fil à retordre. En effet, dès que l'instruction 288 exécutant le code propre à ce process est décommentée, le programme semble ne plus répondre. Nous l'avons donc commentée, à défaut, pour pouvoir observer le bon comportement. Afin d'arrêter le programme, il convient en conséquence d'utiliser ctrl+c.