Lucas Giordano 1517317
Auguste Lefèvre 1517600

Eindhoven, the 13th of December

Assignment 2 : Threading

1. Introduction

This report is a short description of our implementation of the assignment 'Threading' mainly focusing on the decision we made during the development and how the threads are synchronising and how we managed to assure mutual exclusion where needed.

2. Mutexes

We could treat the buffer as one unique resource and thus only have one mutex for the mutual exclusion of the buffer. This would work perfectly. However, as an **optimisation**, we will define as many mutexes as they are entries in the buffer (i.e.: a buffer of size 3 will result in 3 semaphores), this will allow several threads to access the buffer at the same time (if they access different entries), speeding up the processes.

3. Threads initialisation

At the beginning of the main function, after initialising the mutex, we create the thread by calling our function `'create_threads'`. To create threads we use the function `'pthread_create'` of the `pthread` library in a for loop from `0` to `NROF_THREADS.` In this function there are 3 important parameters:

1. The 1st one is a pointer of `pthread_t`, it represents the thread we create. Then, before creating threads we created an empty 'array' of `pthread_t` of size `NROF_THREADS` to store the threads.
2. The second important parameter is a function/process. In fact, we should give to the threads the process that he should perform. In our case the process is the '`flip`' function. This function takes as parameter a list of integers, and flip on the buffer all the multiple of thus numbers.
3. Finally, the 3rd important parameter is the `parameters`. As said before, the function 'flip' needs parameters to be used. So, we need to give different parameters to the different threads we are creating. To respect the good behaviour of the game, we need to split the work correctly depending on the number of threads and pieces in the game, this operation is supported by the function '`create_parameters`' (we will see how in another part).
   Now that all the threads are created we have to 'recover' what threads did. To do so we call the function '`pthread_join`', in a for loop from `0` to `NROF_THREADS`, on each thread we created. The join function waits for the specified thread to terminate. A more efficient way to use the join function would be to determine which threads terminated and call join of this thread to avoid as much waiting time as possible. However, we cannot implement this since there is no way to know where a thread will be done.

Optimisation 1: rather than create a different thread for each number we want to flip, we chose to create a thread pool of `NROF_THREADS`.
Optimisation 2: rather than sending to each tread one number at a time, we chose to pass an array of numbers directly to the thread. This will avoid unnecessary thread communication which will result in a faster program obviously. In order to do so, the `parameters`, defined in point 3.3, will thus correspond to an integer array. For fairness reason, we want to fairly distribute the numbers to each thread. In that sense, we could simply give to each thread the same amount of numbers to flip but how the numbers are actually split will be explained in the next section.

4. Splitting the work

Let be `NROF_PER_THREAD = NROF_PIECES/NROF_THREADS.`

If we simply give the first `NROF_PER_THREAD` integers to thread 0, the next `NROF_PER_THREAD` numbers to thread 1 (from `NROF_PER_THREAD` to `2*NROF_PER_THREAD`), and so on, it would create very unbalanced work between the threads. Indeed, for number 2 for example, we need to flip `NROF_PIECES/2` bits in the buffer but for number `NROF_PIECES-2`, we only need to flip one number. We can see that small numbers requires more work than large numbers.

For this reason, we decided to split the numbers across the threads in a non-trivial way:

- Number `n` is coupled with thread `t` if `n mod NROF_THREADS = t`.

Obviously, this is not the perfect binding we could think of, but it is quite simple and still achieves the goal of balancing the work between treads.

5. Deadlock analysis

We defined one mutex semaphore for each buffer entry (each `uint128t`). Formally, to have a deadlock each of the four conditions must happen simultaneously:

- Mutual exclusion: this happens due to the semaphores.
- Non-preemption: a buffer entry cannot be preempted (since each of them is protected by a mutex semaphore), the resource can only be released only voluntarily by the thread holding it, after that process has completed its task.
- Hold and wait: for this condition to hold a thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other thread. However, in this project since a thread only requires a single resource (one buffer entry) to execute his task, we cannot have a hold and wait condition.
- Circular wait: since the Hold and wait condition does not hold, the circular wait condition cannot hold either.

Since, Hold and wait and Circular wait does not hold, a deadlock cannot occur in this system. However, to be more complete, we will still give a small proof to show why a deadlock cannot occur.

Assume a deadlock has happened. This means all processes are blocked in a `mutex_lock` operation. Let `K <=M` be the smallest number of semaphores that participate in the deadlock (each `m_i for i in {0, K-1}` blocking at least one thread). The value of all of those semaphores is equal to 0 since they all block at least one process. Since the initial value of each semaphore is equal to 1, this means that there are K threads that have entered their critical section but could not exit it (otherwise the value of the corresponding semaphore will not be 0). However, by the structure of the program we see that there are not blocking operation within each critical section. This implies that all critical sections must terminate, which contradicts the deadlock state. No deadlock can thus occur.