Assignment 3 : condition synchronization

Lucas Giordano | 1517317
Auguste Lefèvre | 1517600

January 2020

1 Introduction

This report is a short description of our implementation of the assignment 'Condition Variable' and particularly the implementation of the bounded buffer problem. We'll mainly focusing on the decision we made during the development : how the threads are used to represents producer and consumer and how we used condition variables to synchronized them.

2 Bounded buffer

2.1 General

To do this assignment, we used a circular buffer (FIFO) data structure . Our buffer is in reality a array of $ITEM$ of size $BUFFER\_SIZE$ controlled by a FSM. Indeed, they are 3 states for our buffer : $\{EMPTY, MID, FULL\}$. We also created two auxiliary methods to $buffer\_push$ and $buffer\_get$ element from the array. Those obviously depends on the current state of the buffer.

2.2 Mutual exclusion

The buffer implementation does not directly deal with mutual exclusion, instead it is up to the threads to make sure that they have exclusive access before accessing it. This will be done via a global mutex, used by all producers and consumer. Each time a producer thread get a new item to push in the buffer he locks the mutex using' $pthread\_mutex\_unlock$. Then the thread do his job, push the item in the buffer and finally unlock the mutex when it's done. We repeat the same process for the consumer thread, except that in this case the thread pop a item and not push one.

3 Condition variables

We chose to implement this assignment using 3 condition variables such that we are able to only signal thread that are waiting and that could be woken up to reduce signaling as explained in the following sections:

1. $full\_condition$: this condition is used to ensure that no producer can add an element into a buffer that is already full. They have to wait until the consumer has taken out at least one number. Because of the order of the wait code lines of this condition and the third one, we can infer that only one producer can be blocked at each point of time.

2. $empty\_condition$: this condition is used to ensure that the consumer cannot retrieve an element from an empty buffer. It must wait until a producer has put a new element into the buffer.

3. $turn\_condition$: this condition is used to maintain the requirement that items should be placed in the buffer in ascending order by the producers.

4 Signaling

4.1 General

Since a thread shall not signal 'uninterested' threads, the consumer will only signal producers if the previous state of the buffer was *FULL*. A similar argument applies for the producers : they will only signal a waiting consumer if the previous state of the buffer was *EMPTY*. For the third condition variable, producers with signal other producers, we need to *pthread_cond_broadcast* (Sigall) since there maybe at most 1 specific waiting thread that could be woken up.

4.2 Optimisation

4.2.1 Description

It is possible to reduce signaling due to the third condition. Observe that if within the threads that are waiting on the third condition, if none of the numbers generated correspond to the next item that should be put in the buffer (according to the ascending order policy) we do not need the call to *pthread_cond_broadcast* since the number we want has not been generated yet. If we do broadcast, every thread woken up will simply go back in a waiting state, which should be avoided.
To do this, we maintain a array *waiting_items* where the element a index i correspond to the number generated by the waiting thread i (if it is not waiting then it corresponds to a number already put in the buffer so we do not care). To reduce signaling we thus simply perform a linear search (function named *need_signal* in our code)is this array before each broadcast by a producer, if the number that we want is found, we do broadcast, otherwise we don't.

4.2.2 Measures

To show that this optimization indeed reduce signaling, we counted the number of calls to *pthread_cond_broadcast* and compute the percentage of avoided calls due to this optimisation. We can find that with the basis set up ($NROF\_PRODUCER = 10$ and $NROF\_ITEMS = 2000$), we usually reduce signaling by 25%.
If we fix the number of items to 2000 and let the number of producer grows, we have figure 1. This is indeed the shape that we could expect, more producers means that the probability of having already generated the next number is higher and we need more signaling.

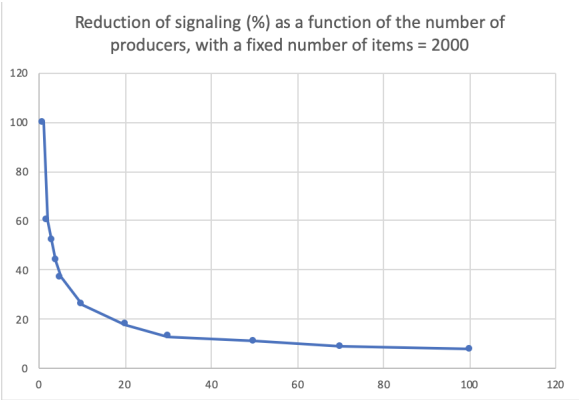

Figure 1: measures

We repeat the experience but this time we fix the number of producers to 5 and let the number of items grow from 0 to 1,000,000 and we obtain figure 2. We observe that we usually reduce signaling of 28%, but it looks like there is an increasing of the percentage when the number of items tends to infinity. Unfortunately, we can not increase more the number of items due to the limited capacity of computing of our laptops.

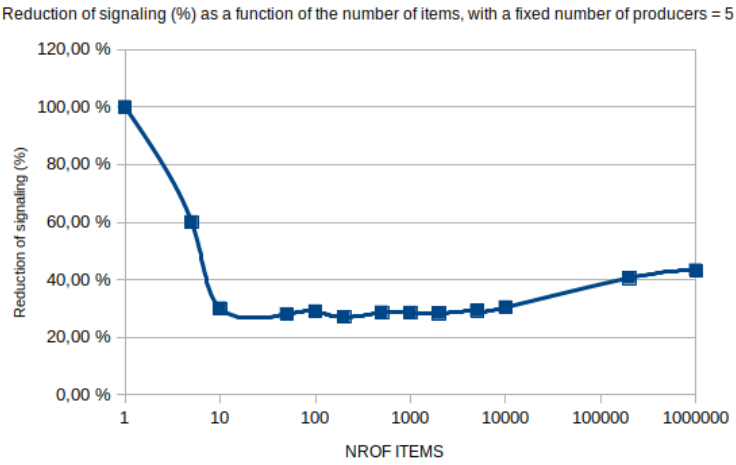

Figure 2: measures 2

5 Deadlock analysis

Formally, to have a deadlock each of the four conditions must happen simultaneously:

- Mutual exclusion: this happens due to the mutex semaphore.
- 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 and release it via *pthread_cond_wait*, 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 occured. This means that each thread is waiting on one the 3 condition to become true or that it was not signaled when needed, since the mutex semaphore cannot obviously generate any deadlock here. The signaling was discussed in the previous section and we can easily convince ourselves that, our solution models the right behaviour. Due to the structure of the program (condition 1 and 3 for producers and condition 2 for consumer), this means one of these combinations of conditions leads to both conditions being false at the same time :

- buffer is full and buffer is empty: obviously that can never happen since the buffer can only be in one state at given time, not both at the same time (property of FSM).
- this is the turn of some thread and buffer is empty : the first one being false means that each producer thread has generated a number that is not the next number that we need to add in the buffer (according to the ascending order policy). However, the implementation of *get_next_item* implies that this would never happen. Indeed, if $NROF\_PRODUCER - 1$ thread have generated another number, we are certain that the last thread will generate the number we expect. And if this thread is blocked on a full buffer this means that the buffer cannot be empty at the same time and that as discussed in the previous case, that we will be able to reach at some point a state where the buffer is not full anymore. Hence not deadlock could occur in this case.

Since none of these cases can lead to a deadlock, no deadlock can occur is this system.