Lucas Giordano 1517317
Auguste Lefèvre 1517600

Eindhoven, the 29th of November

Assignment 1 : interprocess communication

1. Introduction

This report is a short description of our implementation of the assignment *« interprocess communication »* mainly focusing on the decisions we made during the development and how we managed the inter process communication, as well as the process of creation by the farmer.

2. Data Structure

For this project we had to create two specific data structures, one that represents a request message and one that represents a response message. The request message struct `MQ_REQUEST_MESSAGE` is composed of 5 attributes: `work_id, kill_request, first_letter, start_ alphabet, end_ alphabet, md5s`. The struct response message `MQ_RESPONSE_MESSAGE` is composed of 3 attributes: `work_id, key, ack_kill_request`. You can find a description of thus attributes on the documentation in `common.h`.

3. Worker

When a worker is created by the farmer, it waits until there is a request message in the queue (since queues are blocking). Then it processes the request message by generating all the string possibilities using as prefix the `first_letter` and as alphabet: the one in the range `of start_` and `end_ alphabet` and testing if the hash of thus string is equal to the *md5s* value we are looking for. This process is done by recursion using the function `generateKey`. To create responses we use the function `construct_response` that return an `MQ_RESPONSE_ANSWER` and then we send this message to the queue using the function `mq_send` (from the `mqueue` package).

4. Managing the child processes

To avoid creating a new process for each job (time and resources consuming), we want to reuse the previously created workers. This means that each child process should, in a loop, check for a job to execute and should terminate when there are no more jobs. This termination condition has been implemented through “kill workers” messages. When a worker receives such a query, it simply terminates normally (call `exit(0)`). Thus `NROF_WORKERS_REQUIRED` kill messages need to be sent after all jobs.
Optimisation : we decided to create only `NROF_WORKERS_REQUIRED = min(NROF_WORKERS, JOBS_NROF)` since if they `NROF_WORKERS` is less than `JOBS_NROF`, it makes no sense to create `NROF_WORKERS`.

5. Synchronise communication

Synchronisation is implemented via blocking queues: one for sending messages to the workers and one for receiving the answers (point of view of the farmer).
Optimisation: communication through message passing is expansive. So, we want to reduce as much as possible the number of messages sent.

- The child processes should only send results when they actually found a hash. This achieves the desired goal but introduces a new problem for the farmer: it is now impossible to know when a child terminates the computation of a job that has no result. To fix this, a child will send a kill request acknowledgement message to the farmer just before calling `exit(0)`. By counting the number of acknowledgements received, the farmer can terminate as soon as it gets the last one
- As soon as the farmer receives a valid response, it can stop putting jobs related to the same md5 hash in the request queue since they will fail to find a valid hash match and wont return anything. This has been implemented by keeping state of an array of indices (in the struct `request_state_t`) for each md5 hash. Ex: `indices[2] = 3` means that jobs related to md5 number 2 until the 3rd char of the alphabet have been put in the request been. Set `indices[n] = ALPHABET_NROF_CHAR` will thus have the desired behaviour if a response for md5 number n was previously received.

Since in the `manage_responses`, we dequeue `curmsgs` time the response queue, we ensure that the queue is as empty as possible. At each iteration of the while loop we enqueue 1 job in the request queue which makes it as full as possible, as required.

6. Deadlock analysis

A deadlock can only occur in the following scenarios :

- A worker can block is when calling the `mq_receive` method but we see in our code (main method of worker) that if there is a killing request the worker exit the main procedure. Moreover, we see in the function `create_request` of the farmer that we create one kill message for each worker created after all the request messages have been created. So workers will never wait for a query since they are killed after all requests have been processed. And we are sure that those kill requests will be sent since the while loop of `manage_responses` only terminates if `(nbr_worker_left <= 0)` which happens only if all children have sent a kill request acknowledgement. But they can only send this one if they have received the kill request. Therefore, a worker can never block indefinitely.
- By a similar argument, we can show that a worker cannot block indefinitely long when sending a message in the response queue. Assume that a worker `w` wants to send a message in the response queue and that the call to `send(mf_response_queue)` blocks. This means that there are already M messages in the response queue (M is the size of the queue). However, in the farmer `manage_responses` function: the main while loop will still be running (`(nbr_worker_left > 0)` since at least worker w has not sent an acknowledgement kill request because it still wants to put at least one message in the queue). The code inside the while loop is executed. This line is thus executed :
  ```
  long int curmsgs = get_curmsgs(mq_fd_response);
  ```
  Since they are M messages in response queue: `curmsg = 10`. This means that the following for-loop will be executed M times. And since it sufficient to execute this loop once to wake up worker `w, w` can now put its message in the queue. We just showed that w cannot block indefinitely in this scenario.
- A farmer cannot block while executing `mq_receive(mq_fd_response)`since we execute the for loop `curmsgs` times and we are sure that they are at least `curmsgs` in the response queue. Since the farmer is the only entity that could dequeue messages from the queue (worker can only enqueue messages but this is not a problem: if `curmsgs` increase, `mq_receive` has even fewer chances to block), we have proved that the farmer cannot be blocked by this call to `mq_receive`.
- The last scenario where the farmer could possibly block is when a `mq_send` is executed. Assume thus that this append, we will show that the farmer can still continue its execution at some point (no deadlock). If this occurs, this means that the request contains already 10 messages. If we arrive there, it means that we still need to send at least one kill request to at least one worker (since those are the last jobs we send to the workers). This means that there is at least one worker running their main loop:
  - the worker cannot be blocked by a receive call since this queue is full (because farmer blocks). Therefore at least one worker will, at some point call the `receive` method which will unblock the farmer.
  - The worker cannot be blocked by a send call. This would mean that the answer queue is full. But we know from the previous iteration of while loop of the farmer, that the farmer has emptied the answer queue. This means that at least M workers should have sent an answer message between the execution of the `receive` and the `send` of the farmer (if M is the size of the queues). But if this occurs, this means that those M workers can now dequeue the request queue. As a result, the farmer can thus push its job into the queue and can continue its execution.

Since the queues are the only variables shared that may cause a deadlock and that every combination of receive and send (which are atomic) between workers and farmer will not block both entities. So, no deadlock will occur.