Description

Student 1: S.S. Iyer (866094)
Student 2: X. Teng (851499)

Overview

We created two versions of interprocess communication. One is called **normal processing** where a single farmer sends requests and receives the responses from the worker processes. We call the second version, **parallel processing** where we use the farmer to send requests to the workers but receive the responses on an extra child process of the parent (farmer) process. One of the goals of the assignment was to keep the farmer queue as full as possible while keeping the worker queue as free as possible. Normal processing does this with one hand tied behind the back. This was our motivation to parallel process the requests and the responses.

Note: By default, the program is set to run using normal processing for achieving the minimum requirements of the assignment. But there is a defined constant in the common.h file called RUN_PARALLEL which is set to 0 (by default). To test the parallel processing, set RUN_PARALLEL=1 and don't forget to remake the files.

Parallel processing

The farmer creates all the worker processes plus an extra process to receive responses from them. Farmer sends requests to workers containing a Y-pixel value. The worker s receive them, computes the colors of the pixels of the X-axis line corresponding to the Y-pixel value and sends the responses to the extra child process which outputs it to the screen. Finally, when the parent (farmer) process is done sending requests, it sends kill messages to each worker processes. Upon receiving them, the workers forward the kill message to the receiving process and shuts down. The receiving process decrements a counter every time it receives a kill message from one of the worker processes. Once the counter reaches 0, it also terminates execution. Meanwhile, the parent waits for all the worker process id's to close, closes the message queues and unlinks them before itself stopping execution.

Normal Processing

The farmer creates all the worker processes and sends requests to workers containing a Y-pixel value. The worker s receive them, computes the colors of the pixels of the X-axis line corresponding to the Y-pixel value and sends the responses back to the parent process which outputs it to the screen. Finally, when the parent (farmer) process is done sending requests, it sends kill messages to each worker processes. Upon receiving them, the workers simply shut down. The parent process continues receiving until all the worker processes shut down. Once all workers shut down, it continues to check for any unreceived messages in the response queue and outputs them till the queue is empty. After which, the parent waits for all the worker process id's to close, closes the message queues and unlinks them before itself stopping execution.

Problems faced

- **Keeping the farmer queue as full as possible while keeping the response queue as empty as possible using normal processing.**

We fixed this problem by intelligently sending messages only when the request queue is free and receiving responses as soon as there is minimum one response in the worker queue. This problem is also fixed by introducing parallel processing.

- **Figuring out how to shut down the workers.**
  Although the solution we have is quite simple and elegant, it took a lot of time figuring out. Initially, we were looking at options to force kill the worker child processes. But this was not and ideal solution and even if it was, we did not find out how to do it before we came up with our current solution. Our idea was inspired by the semaphore poison pill technique, where the producer keeps feeding the consumer process apples (messages) and sends it a poison at the end.
- **Any deadlock possibility.**
  In parallel processing, there is no possibility of deadlocking. The critical sections where this may happen only include the point of exchange between processes, i.e. where we use mq_send() and mq_receive(). For example, mq_receive() could block forever if no process is sending anything. But these issues are fixed because every process knows clearly the lifetime of the processes from which they are sending and receiving. For example, the receiving process knows it has to continue receiving until all worker processes have forwarded their kill message to it and not just when one of them forwards it to it. There is however the possibility of busy waiting, but that's only because one process is receiving too quickly or one is sending too fast. In other words, it's because of increased efficiency.

  In case of normal processing, the farmer process, only sends when the queue is free enough and receives when there is minimum one response to receive and does this as long as there are workers to send to/receive from. Once the loop exits, whatever messages are left on theresponse queue are read only until there are no more messages on the response queue to read.