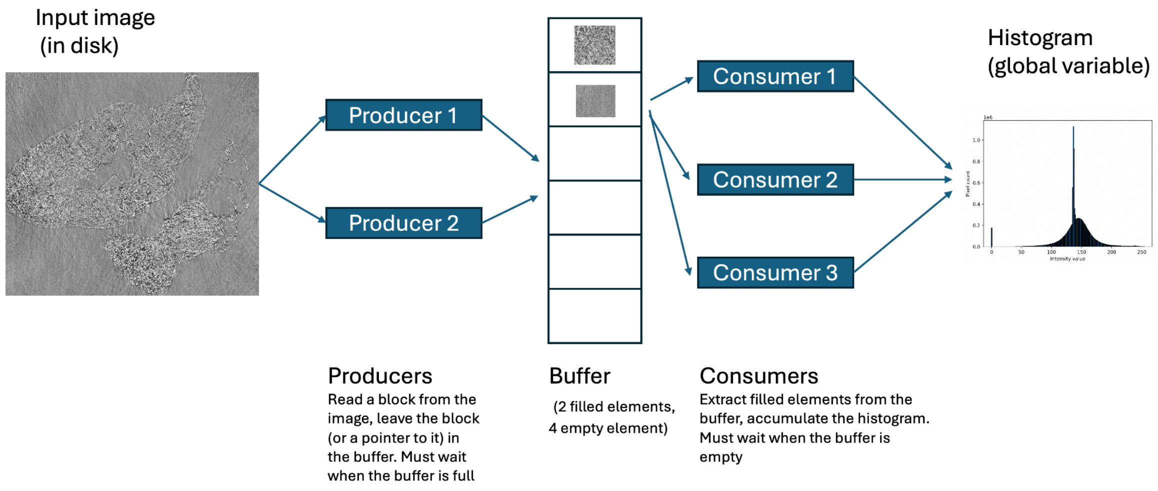
Practice 4: Producer / Consumer

The objective of this practice is to implement a producer–consumer scheme using **pthreads**, applying advanced synchronization mechanisms (such as semaphores or condition variables) to prevent typical concurrency problems.

In the same context as P3, you must implement a Producer–Consumer model (see T08, and the code examples to find a basic implementation of the producer / consumer). Producers will read blocks of 16,384 bytes (1024 × 16) from a file and place them into a shared finite length buffer. Note that Consumers will remove blocks from the buffer and accumulate their contents into a global histogram. You must add synchronization tools to temporary stop producers when the buffer is full, and to stop consumers from extracting from the buffer if it is empty. Below is a schematic of an example, with 2 producers, and 3 consumers:


Input image (in disk)
Producer 1
Producer 2
Consumer 1
Consumer 2
Consumer 3
Histogram (global variable)
Producers
Read a block from the image, leave the block (or a pointer to it) in the buffer. Must wait when the buffer is full
Buffer
(2 filled elements, 4 empty element)
Consumers
Extract filled elements from the buffer, accumulate the histogram. Must wait when the buffer is empty

Each thread will end based on different conditions: 1) producers end when they reach the EOF, and 2) consumers end when all producers have finished, and the buffer is empty. At the end of the execution, the program must ensure that consumers terminate correctly once all producers have finished and no more data will be added to the buffer. A common solution is to use a shared flag (for example, `producers_finished`) that is set to 1 when the last producer ends. Consumers should check both the buffer state and this flag: if the buffer is empty and `producers_finished` is set, they can safely exit. Special care must be taken with consumers blocked on a semaphore or waiting on a condition variable when production ends. Once the flag is set, the program must actively wake up all sleeping consumers (e.g., by broadcasting on the condition variable or posting to the semaphore) so they can recheck the termination condition and exit gracefully instead of remaining indefinitely blocked.

To obtain the highest score, your implementation must correctly handle synchronization, avoiding race conditions, starvation, busy waiting, and unnecessary lock holding time. All dynamically allocated memory must be properly released. The buffer size, as well as the number of producers and consumers, will be provided as command-line arguments.

Usage

```
> computeHistogram Data/heart.pgm Data/histogram.txt N_producers N_consumers sizeBuffer
```

Part 1 (4 pts)
Implement the consumer / producer schematic previously described, where consumers accumulate the histogram, and the producers read from the file. Assume there is a single producer and multiple consumers.

Part 2 (2 pts)
Ensure that the program terminates gracefully. You may use a global variable to indicate that production has finished. Make sure any waiting consumer is properly awakened. If you use `return` or `pthread_exit` to terminate a thread, ensure that any held locks are released before exiting.

Part 3 (2 pts)
Extend your solution to support multiple producers, possibly reading from different open files. You may maintain an index indicating the next block to read. You can use the example in `producer.c` as a starting point, adapting it to include proper synchronization.

Theory (2 pts)

1) Discuss the synchronization issues that may arise and justify the mechanisms you selected to address them.
2) Explain why your solution prevents race conditions, starvation, and busy waiting.
3) Describe how the program behaves since the first producer reaches EOF until the process ends, paying special attention to consumers that may still be blocked waiting for data. Note that when a producer reaches the EOF, there might be other producers which are still doing them.

Delivery instructions

1) Submit through gradescope before Sunday 08/03/26.
2) Add all the integrants of the group during the submission. Here you have the instructions.
3) The submission requires to **exactly** follow the structure below:
    a. *src* folder, with all the sources. There can only be one C code with a main.
    b. compile.sh, a bash script which will contain the compile command (and no other commands).
    c. integrants.txt : the name and NIA of all the group members
    d. P4_report.pdf: report answering the questions
4) Remember to remove all hidden files before submitting.
5) Non adherence to the submission instructions will result in point penalization.