FIFO Queue Parallelized with POSIX Threads

Real Time Embedded Systems
Levi Ioannis 9392
Project Part 1/3

1. Introduction

In this project, we were given an example to base our project upon. In the example, a FIFO queue is implemented, in which with the help of a producer and a consumer, items were added and removed from the queue. Our objective was to implement this using multiple threads both adding and removing to the queue. Given a certain other struct, multiple producer threads add the items to the queue waiting to be executed, and in turn multiple consumer threads remove and execute the struct's function. The code also measures the time between an item's insertion to the queue and the time a consumer removes it, but before it executes it. Finally the program was measured for different sets of parameters and the results are discussed in the end of this report.
The code has been uploaded on github at:
https://github.com/yiannislevy/rtes

2. Code Implementation

Taking into account the fact that the given example which the project is based upon, is known to the reader, and also it is included to the repository, I will elaborate in short on the parts that were altered and or added to.
The FIFO queue, gets *struct* type items, which have three objects. The one is a double variable used to measure time, then there is a pointer to a function of my choosing and a pointer to a variable. The function calculates the cube of a random number every time it is run, and then it prints the number/ID of the process executed. The pointer to a variable of the struct is used in the measurement of this number/ ID.
In *main*, the desired number of threads are created with loops. In order for the program to run smoothly, the producer threads *'wait'* each other to finish their number of work cycles. Since we were asked to implement a **while(1)** loop on the consumers, I implemented the code in **Algorithm 1** in order to decide when and how it would be safe for the consumer to stop waiting for a signal *'fifo is not empty'*.

Algorithm 1 Stop consumers from waiting

```
total_consumers = defined by user
consumer_counter = 0 : total_consumers
total_producers = defined by user
producer_counter = 0 : total_producers
//inside *consumer
while (true)
    pthread_mutex_lock(fifo→ mut)
    while(fifo→ empty)
        pthread_cond_wait(fifo→ notEmpty)
        if (fifo→ empty &
         producer_counter = total_producers)
            pthread_mutex_unlock(fifo→ mut)
            return(NULL)

//inside main
while (true)
    pthread_cond_broadcast(fifo→ notEmpty)
    if (consumer_counter = total_consumers)
        break
```

In **Algorithm 1**, simplified parts of the code are shown in pseudocode. The variables *total_prod*ucers *a*nd *total_consumers* are predefined by the user as in how many threads they want to create for each process (inserting and removing from the queue). The purpose of *producer_counter and consumer_counter* is to keep track of how many processes have completed their job. The first **while** loop is inside the ***consumer** function. The loop runs forever for each consumer thread called. When producers have finished adding items and after consumers have removed everything from the queue, it gets signaled as empty, thus the consumer threads enter a waiting condition, until the fifo is signaled as *'not empty'*. To overcome this, after producers have **joined**, the **while** loop inside main starts signaling that the fifo is *'not empty'*, without actually flagging the fifo as '*notEmpty*', so that the consumer threads are awakened and enabling them to continue to the **if** condition. The condition checks whether the queue is actually empty and if producers have finished adding items to the queue. When consumers will have finally removed every item from the queue, and

have executed the function, the fifo will unlock. Then they will all **join** and finally delete the queue, freeing any reserved memory.

3. Results

The code's performance was tested in a 4-core processor with 8 threads. For the measurements, the producer threads were 8. The number of consumer threads varied. Also, the number of work cycles each producer was requested to perform, remained the same at 30. The queue's size was at 10 items.

In **Figure 1** we see the wait time for each work cycle, when using 8 producers, for 2 to 8 consumers. Overall we notice the same pattern for each number of consumers, with the main difference being lower wait times as the consumers increase, but that will be discussed at **Figure 2**. At the beginning of the work cycles, we can notice how the queue operates. The producers start adding items to the queue, and since it is a First-In-First-Out queue, we see that the wait time of the first 8 items is higher with every inserted item, until the first consumers are created and wait times drop. That is because the first 8 threads created are producers, which start working before the first consumer is created. The drop in wait times, starts after the 8th cycle. When the program is stabilized, we can notice abstractly a sawtooth wave.

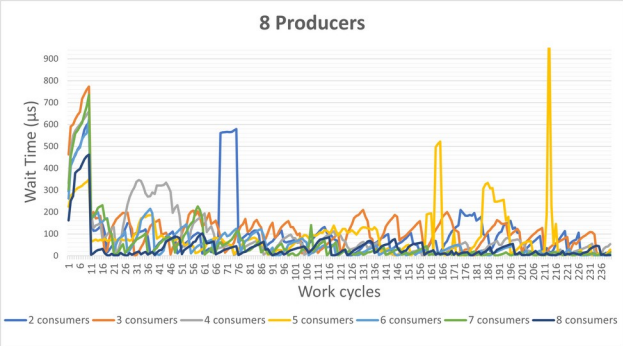

Figure 1

In **Figure 2** we see the average wait time, when using 8 producers, for a variety consumers. It is clear that the more consumers used, the lower the wait times. The average wait times follow an exponential distribution. Experimenting with various consumer threads, I found that the systems' threads do not limit the performance of more pthreads. Thus with 256 consumer threads, the average waiting time of an item inside the fifo queue dropped to approximately 1 microsecond (10^{-6}s), when with 2 consumers it was at 422 microseconds.

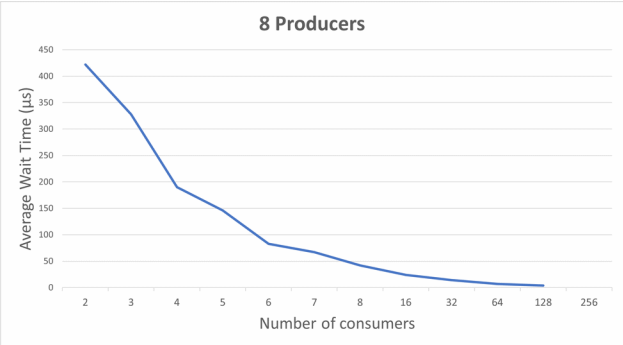

Figure 2