Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
Embedded Systems

Nikolaos Kylintireas
May 2025

Source code: https://github.com/NicKylis/product_consumer

Producer Consumer Multithreaded Example

1 Abstract

This project implements and evaluates a multithreaded solution to the classic producer-consumer problem on a 16-threaded machine. It was developed as part of the Embedded Systems course under Prof. Nikolaos Pitsianis. The baseline code was originally written by Andrae Muys on September 18, 1997, and has been adapted for modern experimentation and performance analysis.

2 Introduction / Algorithm design

The producer-consumer problem is a fundamental synchronization challenge in computer science, where multiple producer threads generate data and place it into a shared buffer, while multiple consumer threads retrieve and process that data. This solution employs a multithreaded approach to efficiently handle concurrency and resource sharing in a bounded queue. To prevent race conditions, the queue operations are protected by mutex locks.

Producers generate tasks and assign each one a unique ID, which is tracked for performance analysis. Consumers execute them and calculate the waiting time for each task in the queue, enabling the system to measure and report on the efficiency of the producer-consumer interaction. The task workload given in this specific example is to do 10 sin calculations, to simulate real-world processing requirements in multi-threaded environments.

This approach ensures synchronization between threads, allowing for scalable task processing while managing contention for the shared queue. The key steps of the algorithm are outlined in Algorithm 1, which illustrates the high-level process of task production, queuing, and consumption.

Algorithm 1 Producer-Consumer (Condensed)

```
1: Initialize queue Q, mutex, cond. vars, and global counters
2: for each producer do
3:     for each task do
4:         Generate data, timestamp, unique ID
5:         Lock mutex; wait if full
6:         Enqueue task; unlock; signal notEmpty
7:     end for
8: end for
9: for each consumer do
10:    while true do
11:        Lock mutex; wait if empty
12:        Dequeue task; unlock; signal notFull
13:        if termination task then
14:            Exit
15:        end if
16:        Compute and record wait time
17:        Execute task
18:    end while
19: end for
20: Print average waiting time
```

3 Results / Evaluation

To test the aforementioned algorithm, a series of experiments were conducted on a 16-threaded machine. The workload consisted of producers generating computational tasks—each involving

10 sine calculations on randomly generated angles—and consumers retrieving and processing those tasks from a bounded queue.

Tests were performed using varying numbers of producer and consumer threads. For each configuration shown below (1, 4 and 8 producers), the average waiting time that a task spent in the queue was measured before being consumed. Waiting time was calculated as the difference between the time the task was enqueued and the time it was dequeued by a consumer, using the gettimeofday() function.

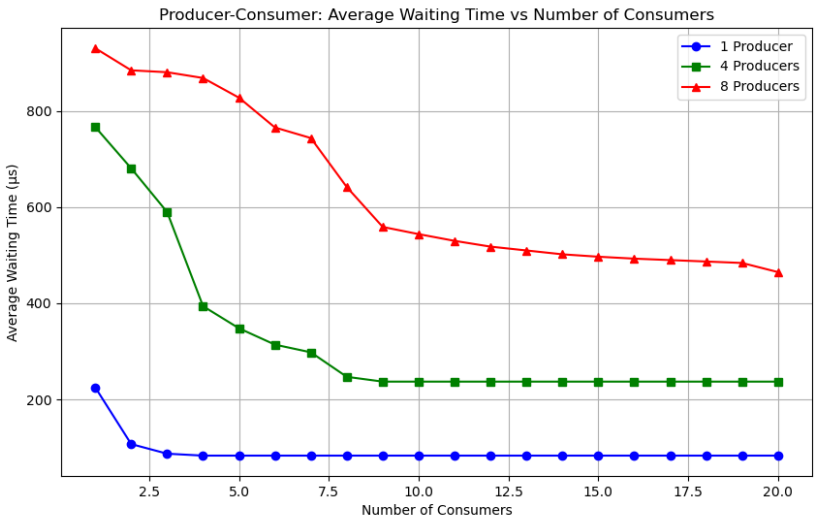

Figure 1: Average waiting time experiment

As expected, the results show that increasing the number of consumers leads to a decrease in average waiting time, up to a certain point. However, as the number of producers increases, the system begins to show signs of congestion, with the waiting time per task rising due to the queue becoming saturated. For optimal performance, a balanced number of producers (around 4 here) and consumers (around 10-15 here) should be chosen. This avoids both underutilization of resources and system congestion, ensuring that tasks are processed efficiently.

4 Tools

The following list contains all the tools used for this project:

- C Programming Language
- Visual Studio Code
- Python and Matplotlib for the plots
- Github for version control
- Windows Subsystem for Linux (WSL)
- Latex (Overleaf)