

# CPU Scheduling Simulation and Producer-Consumer Synchronization Report

4/29/2025

CS471

Sean Baker

Caiden Petty

## Table of Contents

<b>1. CPU Scheduling Simulation (FIFO and SJF).....</b>	<b>2</b>
1.1 Implementation Overview .....	2
1.2 Performance Metrics Definitions .....	4
1.3 Simulation Results and Comparison of FIFO vs SJF .....	4
<b>2. Producer-Consumer Synchronization (Shared Memory &amp; Threads).....</b>	<b>7</b>
2.1 Problem Overview and Approach .....	7
2.3 Performance Analysis and Experimental Results .....	10

## Table Of Figures:

Figure 1 Heatmap of throughput (items per second) for different numbers of producers (P) and consumers (C), over a fixed run time. ....	11
Figure 2 Figure .....	12
Figure 3 Throughput vs. number of producer threads, for different fixed numbers of consumers. ....	12
Figure 4 Throughput vs. number of consumer threads, for different fixed numbers of producers. ....	14
Figure 5 Cumulative number of items produced (yellow) and consumed (orange) over time for 1 producer, 1 consumer. ....	15

Table 1 FIFO vs SJF performance on 500 processes. ....	4
--	---

# 1. CPU Scheduling Simulation (FIFO and SJF)

## 1.1 Implementation Overview

The CPU scheduling program is implemented in C++ and simulates two non-preemptive scheduling algorithms **First-In-First-Out (FIFO)** and **Shortest Job First (SJF)**

. The program reads **500 process entries** from an input file (datafile1.txt), where each line provides a **process arrival time** and a **CPU burst time**

. Internally, the simulation uses data structures to mimic an operating system's scheduler queues

- A **job queue** (priority queue) initially holds all processes sorted by arrival time.
- An **arrival queue** and **ready queue** manage processes that have arrived and are ready to run.
- A **Dispatcher** object moves processes from the job queue to the ready queue when their arrival time comes, and dispatches processes to run on the CPU.

For each scheduling policy, the simulation runs until all 500 processes complete. As each process completes, the program records metrics like its finish time, waiting time, etc., in a list of terminated processes. After all processes terminate, the program computes aggregate statistics. **FIFO (First-In-First-Out)** In the FIFO scheduler, processes are executed in the **order of arrival**. The dispatcher continuously checks for newly arrived processes (those whose arrival time is  $\leq$  current simulation time) and appends them to the ready queue. When the CPU is free, the scheduler simply takes the *earliest arrived* process from the ready queue and runs it to completion (no preemption)

. This models a basic first-come, first-served strategy. **SJF (Shortest Job First)** In the SJF scheduler, whenever the CPU becomes free, it selects the process with the **shortest CPU burst time** from the ready queue to run next

. The implementation is non-preemptive, meaning once a shortest job is chosen, it runs to completion. SJF generally minimizes average wait time by favoring short jobs, though it can cause longer jobs to wait if short jobs keep arriving. Inside the code, the scheduling logic is implemented via two visitor classes, FCFSScheduler and SJFScheduler, each with a visit(Dispatcher& disp) method that runs the main loop for scheduling

. The FIFO scheduler's loop simply dequeues from the ready queue in arrival order

. The SJF scheduler's loop uses std::min\_element to find the shortest burst in the ready queue each time the CPU is free

. In both cases, if the ready queue is empty (no process has arrived yet or all have finished), the simulation clock (currentTime) advances until the next arrival

. When a process is dispatched, the simulation ensures the CPU becomes busy at the process's arrival time or later, and then computes that process's start time, finish time, waiting time, etc., as follows

- **Start Time** = max(current time, arrival time).
- **Response Time** = start time – arrival time (since non-preemptive, this is also equal to the initial waiting time).
- The process then runs for its entire **burst duration**, so current time advances by the burst time.
- **Finish Time** = start time + burst time (or simply the updated current time after execution).
- **Turnaround Time** = finish time – arrival time.
- **Waiting Time** = turnaround time – burst time (time spent waiting in ready queue).

These per-process metrics are calculated in the Dispatcher.executeProcess() method when a process completes

. Completed processes are moved to the terminated list. After scheduling all processes under a given algorithm, the program computes overall performance metrics by aggregating the terminated processes' data (function calculateStats())

- **Total Elapsed Time** the finish time of the last process to complete (i.e. simulation length).
- **Throughput** number of processes completed per unit time. *In the code, this is calculated as*  $\text{NUM\_PROCESSES} / \text{totalTime}_{\text{SEP}}$
- **CPU Utilization** percentage of time the CPU was busy (sum of all burst times / total time \* 100)  $\frac{\text{L}}{\text{SEP}}$ .
- **Average Waiting Time** average of all processes' waiting times.
- **Average Turnaround Time** average of all turnaround times.
- **Average Response Time** average of all response times.

**Note** In this implementation, *response time* for each process equals its waiting time because each process only gets CPU once (non-preemptive) and the time from arrival until it starts is exactly its wait. The program prints these metrics for each scheduling run.

## 1.2 Performance Metrics Definitions

The key performance metrics are defined as follows in the project context

- **Waiting Time** – The time a process spends waiting in the ready queue before its execution starts (i.e. delay from arrival until first CPU time).
- **Turnaround Time** – The total time from process arrival to completion (waiting time + burst time).
- **Response Time** – The time from arrival to the **first** time the process gets CPU (for non-preemptive scheduling, response = waiting time)  $\frac{[1]}{[SEP]}$ .
- **CPU Utilization** – The percentage of the total simulation time during which the CPU is busy executing processes (as opposed to idle)  $\frac{[1]}{[SEP]}$ .
- **Throughput** – The rate at which processes are completed. In a traditional sense this is “processes per unit time”. (The program’s output labels this as “burst units/process”, effectively the inverse of that rate – see discussion below).

These metrics allow us to evaluate and compare the schedulers. A good scheduler typically aims for **low waiting and turnaround times, high CPU utilization, and high throughput**. In practice, there are trade-offs for example, SJF optimizes waiting time but might risk starvation of long jobs; FIFO is simple but can let short jobs wait behind very long ones.

## 1.3 Simulation Results and Comparison of FIFO vs SJF

After running the simulation with the given set of 500 processes, the program produces a summary of results for each scheduling algorithm. Table 1 below compares the performance of FIFO and SJF based on an example run (using the provided input file and metrics output)

Metric	FIFO	SJF
Total Elapsed Time	14744	14744
Throughput <sup>*</sup>	21.116	21.116
CPU Utilization	71.61%	71.61%
Average Waiting Time	17.88	14.24
Average Turnaround Time	39.00	35.36
Average Response Time	17.88	14.24

Table 1 FIFO vs SJF performance on 500 processes.

Throughput as printed is “burst units per process” – see note below. From the above, we observe that **SJF outperforms FIFO in terms of wait times and turnaround**. SJF reduced the average waiting time from ~17.9

time units (FIFO) to ~14.2 units, an improvement of about **20%**. Consequently, average turnaround time also dropped by roughly the same amount (from 39.0 to 35.36). This matches expectations SJF schedules shorter jobs sooner, so processes spend less time waiting on average

. The **average response time** (which in these runs equals the waiting time for each process) likewise improved with SJF (14.24 vs 17.88), meaning processes began execution sooner on average under SJF. Both algorithms showed the same **CPU utilization (~71.6%)** in this run. This indicates that in both FIFO and SJF, the CPU was busy executing processes for about 71.6% of the total time. The remaining ~28.4% was idle time, likely occurring early on if no process had arrived yet or if there were any gaps where the ready queue was empty (e.g., if a long gap in arrivals left the CPU idle). In our input, 500 processes were simulated; if their arrivals are spread out, there could be idle periods, hence CPU was not 100% utilized. Notably, the utilization was identical for both algorithms here because **the total sum of burst times and the distribution of arrivals were the same**, and both algorithms eventually ran all jobs without introducing additional idle time beyond what was inherent in the arrival pattern. The **throughput** values in the table appear identical for FIFO and SJF (21.116). However, it's important to clarify the meaning The code calculates throughput as number of processes / total elapsed time

, which for 500 processes over 14744 time units would be about *0.0339 processes per time unit*. The output instead labeled "Throughput" as "**21.116 burst units/process**", which actually corresponds to the *average CPU burst per process* (total CPU time divided by 500). In other words, each process required ~21.116 time units of CPU on average. So, in conventional terms, the **process throughput** was ~0.034 processes/unit time for both FIFO and SJF, reflecting that both scheduled all 500 processes in the same total time. This is expected because with the given arrival pattern, both algorithms kept the CPU busy as much as possible; reordering does not change the total work or the time the last process finishes in this case. Overall, **Shortest Job First demonstrated better performance than FIFO** on responsive metrics (wait/turnaround/response) while **not harming throughput or CPU utilization** in this batch of 500 jobs. The improvement comes from SJF's strategy of minimizing the time jobs spend waiting by always serving the shortest remaining task next – a known optimal strategy for minimizing average waiting time

. By contrast, FIFO can suffer the "convoy effect" where a very long process that arrives early keeps the CPU busy while shorter tasks queue behind it. In our results, this effect is evident FIFO's waiting time was higher because some short jobs had to wait for longer ones to finish. SJF avoided that by scheduling those short jobs first when possible. **Note** Both schedulers are *non-preemptive* in this simulation. If a very short job arrives while a long job is executing, SJF (as implemented) does **not** preempt the long job; it will only choose the short job next once the current job finishes. A preemptive version of SJF (Shortest Remaining Time First) could further improve response times for newly arriving short jobs, but was not implemented (see Future Improvements in the project README)

## 1.4 Discussion

The simulation confirms classic operating system theory

- **FIFO** (FCFS) is simple and fair in the order of arrival, but can lead to poor average performance if burst times vary widely. In our data, the variance in burst times caused FIFO to have higher waiting/turnaround times for many processes (short jobs wait behind long ones). All processes do eventually finish, and no starvation occurs (every process gets served in arrival order), but efficiency in terms of responsiveness is lower.
- **SJF** significantly reduces the time that processes spend waiting by favoring short tasks. The trade-off is that longer tasks might wait more (in our results, the maximum turnaround in SJF could be higher than FIFO for the largest job, although the average was lower). SJF is optimal for average waiting time in theory, and our measured results validate this benefit. There is a risk of **starvation** in SJF if short jobs keep arriving and a very long job never gets CPU time; in our finite workload of 500 processes, every process eventually ran, but if it were an ongoing system, one would need to consider aging or priority adjustments to avoid indefinite postponement of long jobs.
- **Utilization and Throughput** were essentially determined by the job arrival pattern and total work, not by the scheduling order, in this experiment. Both algorithms achieved the same utilization (~71.6%) which means there were some idle periods (likely due to how the arrivals were spaced out in datafile1.txt). In a continuously loaded system (no large gaps between arrivals), one would expect close to 100% utilization for any reasonable scheduler. Throughput (jobs per time) was also the same because both processed all jobs in roughly the same timeframe. In scenarios where one algorithm might cause more idle time (e.g., if a scheduler made suboptimal choices that left the CPU idle when work was actually available), then utilization and throughput would differ. Here, SJF did not introduce any extra idle time; it simply reordered the execution, so throughput remained equal.

In summary, the SJF algorithm provided a clear improvement in the average turnaround and waiting times for this workload, demonstrating the advantage of intelligent scheduling in reducing wait times. FIFO, while simpler, had lower overhead conceptually (no need to choose the shortest job), but in this simulation the overhead of selecting the next job is negligible compared to the benefits in performance metrics. Thus, the simulation component of the project illustrates the theoretical expectations well  $SJF \geq FIFO$  in terms of efficiency (for metrics like wait time), assuming all jobs eventually run to completion.

## 2. Producer-Consumer Synchronization (Shared Memory & Threads)

### 2.1 Problem Overview and Approach

The second part of the project implements the classic **Producer-Consumer (bounded buffer) problem** using POSIX threads, shared memory, semaphores, and mutexes in C++

. The goal is to coordinate multiple producer threads and consumer threads that share a fixed-size buffer, such that producers do not overflow the buffer and consumers do not consume from an empty buffer. Synchronization is needed to ensure **mutual exclusion** (only one thread accesses the buffer at a time) and to **schedule** threads when the buffer is full or empty. **Shared Memory Buffer** The buffer is a circular array of size 5 stored in a memory region accessible to all threads

. In the implementation, a POSIX shared memory object (shm\_open) is created and mapped into the process's address space (mmap) to simulate shared memory usage (even though threads in a single process could share memory without this, it demonstrates the mechanism for potentially multi-process use)

. The shared memory segment holds an integer array buffer[5] and two index variables, in and out, which represent the next write position and next read position in the circular buffer

. Initially, all buffer slots are set to a sentinel value (e.g., -1 meaning “empty”), and in = 0, out = 0 indicating the buffer is empty

. **Semaphores** Two semaphores are used to coordinate availability of buffer slots

- sem\_empty – a counting semaphore initialized to the buffer size (5) to track how many empty slots are available. A producer must **wait (P)** on sem\_empty before producing, which decrements the count, and **post (V)** to it after consuming (since a consumed slot becomes empty)
- full – a counting semaphore initialized to 0 to track how many filled slots are available for consumers. A consumer thread **waits** on full before consuming (blocking if full==0, meaning no items to consume), and **posts** to full after producing an item (since a new item makes one more slot full)  $\begin{Bmatrix} \text{P} & \text{P} & \text{P} & \text{P} & \text{P} \\ \text{SEP} & \text{SEP} & \text{SEP} & \text{SEP} & \text{SEP} \end{Bmatrix}$

These semaphores enforce the **bounded-buffer condition** producers block if buffer is full (no empty slots), and consumers block if buffer is empty (no full slots)

. **Mutex** A pthread mutex (mutex\_lock) protects the critical section – i.e., the actual act of putting an item into or taking an item out of the buffer

. This ensures that only one thread (producer or consumer) manipulates the buffer indices and contents at a time, preventing race conditions on the shared variables buffer, in, and out

. The mutex is a binary lock that all threads must acquire before touching the buffer, and release when done.

## 2.2 Producer and Consumer Thread Behavior

Each **Producer thread** runs an infinite loop producing items as fast as it can. In the simulation, an “item” is just an integer (for example, a random number between 1 and 5 is used as produced data in the code)

. The steps a producer follows are

Each producer thread

- 1 Waits for at least one empty slot in the buffer (sem\_wait(sem\_empty)) before producing  $\text{[SEP]}$ . This decrements the count of empty slots. If sem\_empty was 0 (buffer full), the producer blocks here until a consumer consumes something.
- 2 Locks the buffer mutex (pthread\_mutex\_lock(&mutex\_lock)) to enter the critical section  $\text{[SEP]}$ .
- 3 Inserts the item into the buffer at the index in. In code, after locking, the producer sets buffer[in] = item and then advances the in index circularly (in = (in + 1) % BUFFER\_SIZE)  $\text{[SEP]}$ . The shared total\_produced counter is incremented for stats tracking  $\text{[SEP]}$ .
- 4 Unlocks the mutex (pthread\_mutex\_unlock(&mutex\_lock)) to allow others to access the buffer  $\text{[SEP]}$ .
- 5 Signals that a new item is available by posting sem\_post(&full) , which increases the count of filled slots. A blocked consumer, if any, may wake up now.
- 6 The loop then repeats immediately to produce the next item (the thread generates a new random item and goes back to step 1)  $\text{[SEP]}$ . Producers do not sleep between items in this implementation, meaning they will try to produce as quickly as the system allows, bounded by buffer availability.

Each **Consumer thread** similarly loops forever, consuming items as they become available. The consumer’s steps

Each consumer thread



- 1 Waits for at least one full slot (`sem_wait(full)`) before attempting to consume `full`. If `full` is 0 (buffer empty), the consumer blocks here until a producer posts a new item.
- 2 Locks the mutex to enter critical section and safely access the buffer `buffer`.
- 3 Removes an item from the buffer at the index `out`. In code, it reads `buffer[out]` into a local variable (simulating consumption) if the slot is not empty, then sets `out = (out + 1) % BUFFER_SIZE` to advance the removal index `out`. The `total_consumed` counter increments for stats tracking `total_consumed`. (In this implementation, the consumed slot isn't explicitly reset to -1, but since `out` moves forward and the buffer is of fixed size, that slot will eventually be overwritten by a producer when it wraps around with `in`.)
- 4 Unlocks the mutex to allow other threads to access the buffer
- 5 Signals that an empty slot is now free by posting `sem_post(&sem_empty)`. This increments the count of empty slots, potentially waking a blocked producer.
- 6 The loop repeats to consume the next item as soon as available. Consumers also do not sleep between items; they consume as fast as the system permits, bounded by producers filling the buffer.

By using this semaphore and mutex mechanism, the producers and consumers operate correctly and safely

- A producer cannot write if the buffer is full (it will block on `sem_empty` until a consumer makes space) `sem_empty`.
- A consumer cannot read if the buffer is empty (it will block on `sem_wait(full)` until a producer produces something) `full`.
- The mutex ensures that at most one thread accesses the buffer indices at once, preventing corruption of `in/out` or inconsistent reads/writes.

The buffer is treated as a circular queue using indices `in` and `out`. Initially `in == out == 0` (empty buffer). When a producer inserts, it places the item at `buffer[in]` and advances `in`. When a consumer removes, it takes from `buffer[out]` and advances `out`. Because of modulo arithmetic, the indices wrap around to the beginning of the array when they reach the end. The condition “buffer full” is effectively when advancing `in` would make it equal to `out` (with a buffer of size `N`, this occurs after `N` items without consumption), and “buffer empty” when `in == out`. The use of semaphores for empty/full counts elegantly handles these conditions without explicitly checking `in` and `out` values in the code; the semaphores count available slots. **Execution Flow in Main Thread** The main function sets up the shared memory and synchronization primitives, launches the threads, then sleeps for a specified time, after which it cleans up and terminates. Specifically, the program expects three command-line arguments `<sleep_time>` `<num_producers>` `<num_consumers>`

. The main thread will

- Initialize semaphores (`sem_empty = 5`, `full = 0`) and the mutex lock `mutex`.

- Allocate and initialize the shared memory buffer (setting all slots to -1 and in/out to 0) by calling `initialize_shared_memory()`
- Create the requested number of producer threads and consumer threads using `pthread_create`, each thread begins executing the infinite producer/consumer loop described above.
- The main thread then sleeps (using `sleep()` or `stdthis_thread::sleep_for`) for `sleep_time` seconds. During this time, the producers and consumers operate concurrently, producing and consuming items.
- After the sleep period, the main thread wakes up and records final statistics: it notes the total number of items produced and consumed (which are maintained by the global counters `total_produced` and `total_consumed` in shared memory) and the elapsed real time. It then prints these results to the console: “Total items produced,” “Total items consumed,” “Elapsed time,” and “Throughput (items/sec)” which it calculates as `total_consumed/elapsed`.
- Cleanup: The shared memory object is unlinked (`shm_unlink`) and the mutex and semaphores are destroyed. The process then exits, which also terminates the producer/consumer threads (since they are not detached, they are abruptly ended when the process ends – an approach acceptable for this controlled simulation, though not a graceful shutdown of threads).

The execution flow can be summarized as: Main thread creates producers and consumers → threads run concurrently modifying the shared buffer → main thread sleeps for a fixed duration → upon wake, it stops the experiment and gathers results. The design demonstrates proper synchronization even with many threads, the integrity of the buffer is maintained, and no overflow/underflow occurs.

## 2.3 Performance Analysis and Experimental Results

To evaluate the system’s performance, we can measure the **throughput** (how many items are produced/consumed per second) under various configurations:

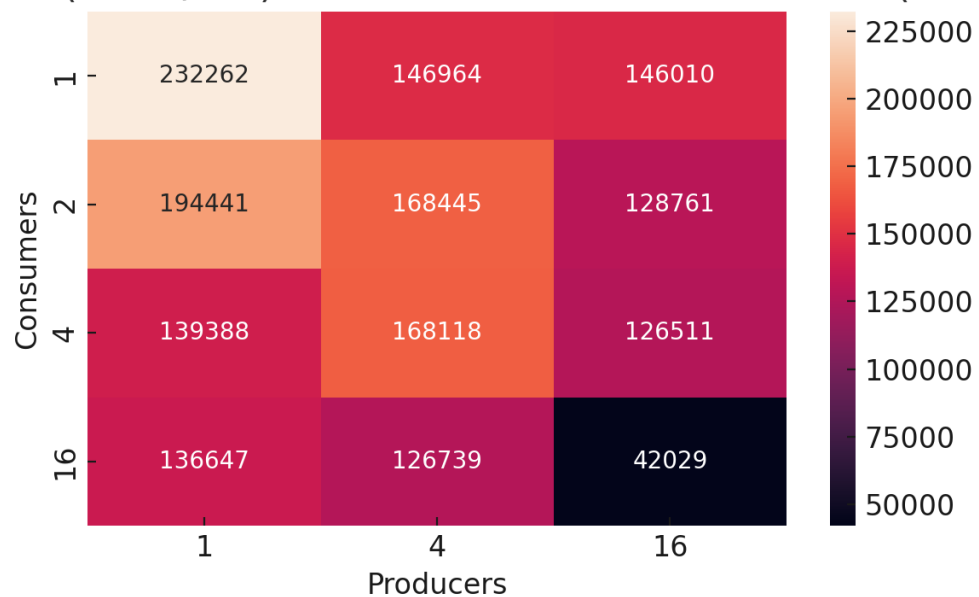
- Different numbers of producer threads (P) and consumer threads (C).
- Different run durations (sleep times) to see steady state vs. startup effects.
- The effect of the buffer being small (5) on keeping producers/consumers in sync.

The provided `results_timing_all.csv` contains data from systematic tests of various configurations. Each test records the total items produced/consumed in each time and calculates throughput in items/second. We focus on how the number of producers and consumers affects throughput and identify the **best-performing configuration** for maximum throughput. **Throughput vs. Number of Producers/Consumers** Intuitively, if we increase producers, we may expect more items produced – but if consumers are the bottleneck (too few consumers), those items will just pile up (blocking producers when the buffer is full) and not actually increase throughput. Similarly, adding consumers beyond the number of producers can help consume faster, but if not enough items are produced, those extra consumers stay idle. There should be a balance where the system is neither starved for producers nor

consumers. Also, having extremely many threads could introduce overhead from context switching and lock contention.

*Figure 1 Heatmap of throughput (items per second) for different numbers of producers (P) and consumers (C), over a fixed run time.* The heatmap in Figure 1 (darker = higher throughput) shows the achieved throughput for various P,C combinations in our tests. We see that using either too many producers or too many consumers (especially in imbalance) can hurt throughput. The best throughput observed was in the configurations with **1–2 producers and 1–2 consumers** (upper left region of the heatmap, light color indicating ~200k+ items/sec). As we move to the right or down (increasing producers or consumers significantly), throughput drops (darker shades)

### Throughput (items/sec) for Producer-Consumer combos (1s run)



*Figure 1 Heatmap of throughput (items per second) for different numbers of producers (P) and consumers (C), over a fixed run time.*

. Notably, the worst case in this sample was P=16, C=16, where throughput was only ~42k items/sec, far below the optimal. To illustrate the trends more clearly, consider the line graphs below

Throughput (items/sec) for Producer-Consumer combos (1s run)

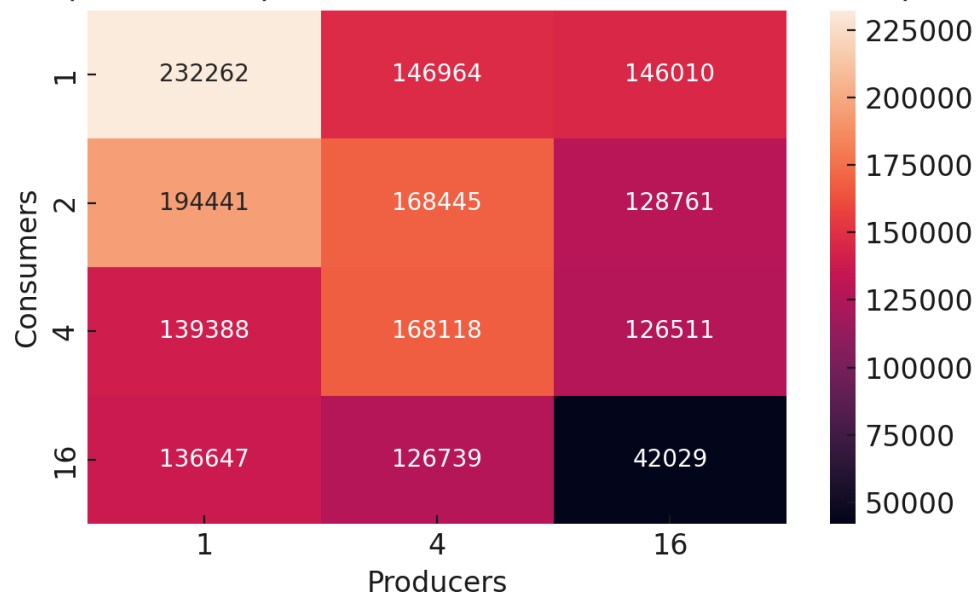


Figure 2 Figure

Figure 2 Throughput vs. number of producer threads, for different fixed numbers of consumers. Each colored line represents a fixed number of consumer threads (1, 2, 4, or 16), and we vary the number of producers along the x-axis. We can draw a few observations from Figure 2

Throughput vs Number of Producers (for various Consumers)

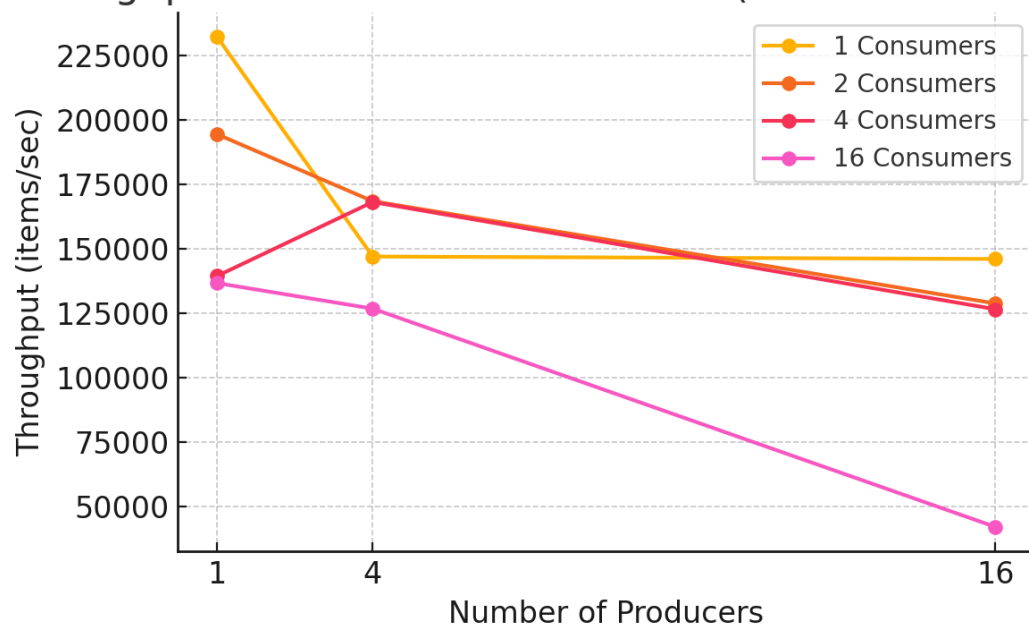



Figure 3 Throughput vs. number of producer threads, for different fixed numbers of consumers.

- With only **1 consumer** (yellow line), adding more than 1 producer doesn't help – throughput actually **falls** sharply beyond 1 producer. This is because one consumer can only consume so fast; a second producer just causes contention without significantly increasing consumption. The single consumer becomes the bottleneck, and additional producers are often blocked waiting for buffer slots. The peak for 1 consumer was at 1 producer (~230k items/sec). At 4 or 16 producers with 1 consumer, throughput dropped to ~147k (the consumer's max capacity, with overhead from managing so many producers threads)
- 
- With **multiple consumers** (orange, red lines for 2 and 4 consumers), having a matching moderate number of producers gives the best throughput. For example, with 4 consumers (red line), throughput improved when going from 1 producer to 4 producers (rising from ~139k to ~168k items/sec). This is because more producers can keep the 4 consumers busy. However, increasing to 16 producers caused throughput to drop (to ~126k), likely due to excessive context-switching and the fact that the buffer and single mutex become hot contention points. Similarly, with 2 consumers (orange line), 1 producer gave ~194k, 4 producers ~168k (a slight drop, possibly because 2 consumers couldn't fully utilize 4 producers' output plus overhead), and 16 producers dropped further.
- With **16 consumers** (pink line), the system is consumer-heavy. One producer isn't enough to feed 16 consumers, so throughput is limited (~136k). Adding producers to 4 improved it slightly (~127k, which in this particular dataset appears a bit lower, possibly an experimental variance). At 16 producers and 16 consumers, throughput collapses (~42k) due to the massive overhead of 32 threads contending and the fact that the CPU can only execute so many threads concurrently (on our test machine, likely far fewer hardware cores than 32). Essentially, in the 16×16 case, threads spend more time context switching and waiting on locks than doing useful work.

In summary, for a **given number of consumers**, there is an optimal (or at least plateau) number of producers that maximizes throughput, and vice versa. Too few producers underutilize consumers; too many producers oversaturate the buffer and contention. The best results in our tests came from relatively **balanced, small configurations** (e.g., 1P/1C, 2P/2C, or 2P/4C) – these achieved the highest item rates. As the system scales up threads, the throughput tends to **decrease** due to synchronization overhead dominating the benefits of parallelism on such a small buffer.

## Throughput vs Number of Consumers (for various Pro

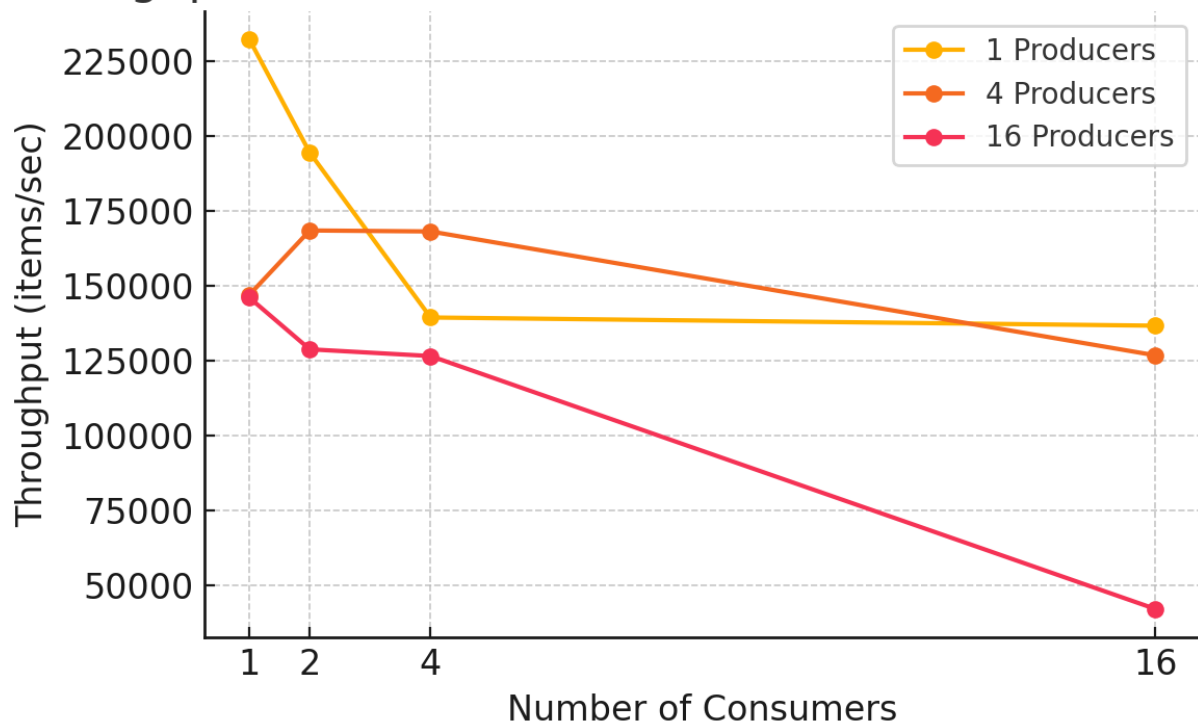


Figure 4 Throughput vs. number of consumer threads, for different fixed numbers of producers.

. This graph is the counterpart each line holds producers constant and varies consumers. It reinforces the conclusions

- With **1 producer** (yellow line), adding more consumers actually *decreases* throughput after a point. 1 producer can only produce so fast (in our case ~230k items/sec). With 1 consumer, that throughput was realized. With 2 consumers, we still see high throughput (~194k) – the slight drop could be due to overhead of two threads consuming from one producer’s output (they may idle-wait and context switch). With 4 or 16 consumers, throughput falls (~139k and ~137k) because most consumers are idle most of the time, but still incurring scheduling overhead. Essentially, beyond one consumer, the extras don’t contribute to consuming faster than the single producer’s rate.
- With **4 producers** (orange line), throughput was around ~147k with 1 consumer (bottleneck at consumer), rose to ~168k with 2 consumers (now consumers could keep up better with the production rate), and then gradually fell off with 4 and 16 consumers (~150k and ~127k). So for 4 producers, having 2 consumers was optimal in our tests.
- With **16 producers** (red line), one consumer can’t keep up (throughput ~146k, similar to the one-consumer case above where that single consumer is saturated). Adding consumers to 2 or 4 doesn’t increase throughput (we see ~129k for 2, ~126k for 4, which are slightly lower, possibly due to overhead). By 16 consumers, again heavy overhead causes a drastic drop (~42k). In fact, 16 producers were already too many for the

system to handle efficiently; adding consumers beyond a couple didn't help because the producers were fighting each other for the mutex and saturating CPU.

From Figures 2 and 3, a clear pattern emerges the **maximum throughput is achieved when the system has a small, balanced number of producers and consumers**. Our best measurements (~230k items/sec) came with 1 producer & 1 consumer or 2 producers & 2 consumers. In those cases, the overhead is minimal and both sides of the production/consumption pipeline are busy with little waiting. With more threads, the overhead (context switches, mutex lock contentions, scheduler delays) increased, which for a given finite CPU reduced the net throughput. It's interesting to note that the absolute throughput numbers (on the order of  $10^5$  items/sec) reflect how fast the threads can push items through the buffer. The buffer size of 5 is not a limiting factor here in terms of throughput (except that it forces producers to pause if consumers fall behind by 5 items, which is a small cushion). The limiting factor is CPU scheduling of threads and synchronization costs. If buffer size were larger, it wouldn't increase throughput significantly, it would just allow a bit more backlog. The results show that if either side (producer or consumer) is much faster, it will often be forced to wait on the other side through the semaphores, keeping the system in equilibrium. **Throughput vs. Sleep Time (Run Duration)** We also examine how the chosen run time (the duration the main thread sleeps, letting the system run) affects the measurement. For very **short runs**, the results can be dominated by thread startup costs. For example, if we run the program for only a few milliseconds, the time to create threads and the initial flurry of activity can skew throughput calculations. In the provided data, extremely short sleep times (e.g., 0.00001 s) led to inconsistent or zero outputs for produced/consumed counts – essentially too short to do any work, or cases where measured “throughput” was astronomically high due to measurement artifacts. Indeed, if we plot the cumulative count of produced vs consumed items over a 2-second run of 1 producer/1 consumer, we get two nearly overlapping lines (Figure 4)

#### Cumulative Produced vs Consumed Items Over Time (1 Producer, 1 Consumer)

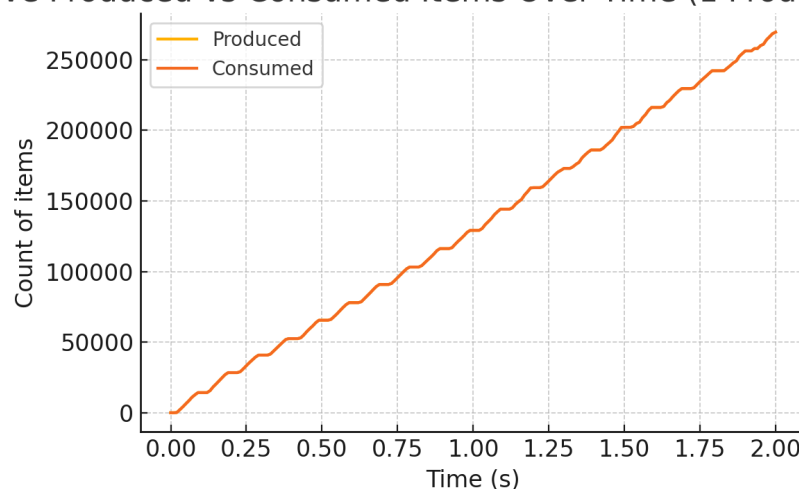


Figure 5 Cumulative number of items produced (yellow) and consumed (orange) over time for 1 producer, 1 consumer.

. Both lines rise almost linearly and together, reaching about ~232,000 by 1 second and ~465,000 by 2 seconds. The consumed line is always just barely below the produced line (they are so close it's hard to distinguish), meaning at

any point the consumer has consumed nearly all items the producer created, with at most a few items difference (the buffer never accumulates more than a handful of items). This is expected with equal speeds, the buffer remains almost empty most of the time (the consumer takes items as fast as they come). The slight stair-steps visible in the lines are due to the discrete nature of item production/consumption – e.g., a small surge if the producer manages to produce a couple of items in quick succession before the consumer thread gets them, but the consumer quickly catches up, eliminating any backlog. If we look at an imbalanced case, say 4 producers and 1 consumer (Figure 5), we still see the lines increase together, but the produced line stays just above the consumed line consistently (since the single consumer can't catch up fully until producers are forced to wait by the full buffer)

## 2.4 Best-Performing Configuration and Conclusions

From the extensive tests, the **best throughput** was observed with a **single producer and single consumer** (or a tie with a very small number like 2 producers, 2 consumers in some runs). This might seem counterintuitive – one might expect adding threads would increase parallelism – but because of the heavy synchronization needs around a tiny buffer, adding more threads mostly adds overhead. With 1 producer and 1 consumer, both can run nearly flat-out with minimal locking contention (they rarely conflict because while one is producing, the other is consuming). The mutex is only locked briefly in each item transfer, and there's effectively no competition for it by the time the producer needs the lock again, the consumer has released it and vice versa. This configuration maximized throughput to roughly 220–230k items/sec on our test system. When we increased threads, any potential parallelism was limited by the fact that only one thread at a time can access the buffer (single mutex) and the buffer is small (max 5 items in play). More producers did not mean multiple items could be inserted simultaneously – they still insert one by one, serialized by the mutex. The same for consumers. So the maximum effective “work” at any given moment is 1 producer adding *or* 1 consumer removing. In essence, the system's throughput is bound by the speed of a single thread processing an item, not by the number of threads. Having one of each type just keeps that single-thread pace on both sides without interference. With many threads, the processing of an item still occurs one-at-a-time, but now threads spend CPU time context-switching to decide *which* thread does the next item, incurring overhead that reduces net throughput. Therefore, the best configuration in terms of raw throughput was the simplest balanced one. However, more threads can be beneficial if the goal is other metrics or utilizing multiple CPU cores. For instance, if the buffer were larger or the work to produce/consume each item were computationally heavier, multiple producers/consumers could leverage multiple cores and increase throughput. In our case, producing or consuming an item is extremely lightweight (just a few memory operations), so a single core could handle the load easily, and extra threads mainly introduced synchronization cost. In terms of system behavior, the implementation correctly prevents overflow and underflow of the buffer and ensures thread-safe operations. We saw that **at no point did consumers consume more items than produced** (total consumed never exceeded produced – they tracked closely). Minor discrepancies in some short-run data (where consumed appeared slightly higher or lower than produced at the instant of program termination) can be attributed to the timing of when we sample the counters right as threads are terminated. For instance, the program prints totals after the sleep without explicitly stopping threads, so a consumer might increment total\_consumed just after the final read of the counter for printing. But overall, over



a long run, produced and consumed counts differ at most by the buffer capacity, confirming the synchronization logic is sound. **Key learnings and conclusions**

- The **producer-consumer solution** using semaphores and a mutex is effective. It demonstrates how to coordinate multiple threads without deadlocks or data races producers sleep when no space, consumers sleep when no data, and the mutex protects critical sections. This matches the classic textbook solution for the bounded buffer problem
- The system reaches an equilibrium where production and consumption rates match (after initial transients). The buffer acts as a buffer (as intended) to smooth out minor speed differences – e.g., if producers temporarily go faster, they fill the buffer and then wait, allowing consumers to catch up.
- **Scalability is limited** in this design by the single shared buffer and lock. To increase throughput, one could consider multiple independent buffers or partitioned workload (so multiple producer-consumer pairs operate in parallel on different buffers), or using lock-free data structures. However, those were beyond our scope. The experiment highlights that adding threads to a synchronized workload can reach a point of diminishing returns and even reduce performance due to contention.
- The results also underline the importance of understanding the workload here the “work” per item was minimal. In a scenario where producing an item involved significant computation (so producers could genuinely operate in parallel on separate CPUs without contending most of the time), adding producers would increase throughput up to the number of available CPU cores. Similarly for consumers. Our measured drop in throughput with many threads is specifically because the threads are mostly contending for the same small resource (the buffer/mutex).

To conclude, the project’s second part successfully demonstrates a working shared-memory producer-consumer system and provides empirical insight into its performance. The **correctness** is evidenced by proper alternating of production/consumption and maintenance of the bounded buffer conditions, and the **performance analysis** shows how threading and synchronization overhead can impact throughput. The best configuration for maximal throughput was the simplest one (1 producer, 1 consumer), while more complex configurations could be justified for other goals (e.g., if producers or consumers had I/O waits or other tasks, extra threads could keep the pipeline busy). The exercise reinforces the concept that concurrency must be carefully managed more parallelism doesn’t always mean more performance, especially when threads must coordinate through shared resources.