# Operating Systems Principles

# Programming Assignment 1

## Me

Student ID: S3851781

Name: Jeffrey Zezhong Tan

GitHub Link: https://github.com/JeffreyTan1/OSP-PrgAsg1

## (1) Reader-Writers Problem

### Progression

Initial (naive) solution

Use two mutexes. One mutex to edit readCount and the other mutex to lock the writers from writing if readCount is 1 and unlocks when readCount is 0 (similar to the design shown in week 6 lecture slides).

The clear limitation to this solution is the starvation of writers. Since readers are allowed to go into the ready state immediately after reading, readCount will always be greater than zero and usually 5 (max number of reader threads), or when decremented to 4 after a read loop completes, is incremented to 5 again once the loop restarts.

The writers thread depends on the writer mutex to be unlocked to proceed. And since the writers mutex is unlocked when readCount hits zero, writers never get a chance to write except for at the beginning of the program when readCount starts at zero, or at the end of the program when the reader threads are terminated.

Slight improvement

Ideally, we would like a good mix of interweaved read and write outputs showing us that readers and writers can concurrently execute at the same time.
A fix to the starvation issue was to add system sleeps.

- If we make readers sleep for 1 second after a read loop, the output shows long sequences of writers writing without reads in-between.
- If we make the writers also sleep for 1 second after a write loop, output had a decent interweaving of writers and readers in the output.

However, this 'fix' cannot easily be extended to support more readers and writers. The programmer will need to trial-and-error different sleep times so that the reads and writes are appropriately interweaved.

## Final Solution

### Global Variables

| Name | Purpose |
|---|---|
| resource; | Integer resource that the readers and writers access. |
| readCount; | Counts how many readers are reading the resource. |
| rcMutex; | Mutex to lock readCount. |
| wMutex | Mutex to lock writers from writing. |
| stop; | Stop program after 10 seconds. |
| startThreadsLock;<br>startThreadsCond; | Used to pause execution until all threads are generated (makes output more readable). |
| threadLocks[];<br>threadCondVars[];<br>threadSleep[]; | Arrays of lengths NUM_THREADS * 2 which store the mutex locks, condition variables, and Boolean loop+ variables respectively.<br><br>+(Boolean loop does not imply busy waiting, I use them to ensure that the threads don't awaken due to other reasons apart from the signal) |
| threadQueue;<br>threadIDQueue; | Used to implement first-in-first-served for readers and writers to be served in sequence. |
| qMutex; | Mutex to lock the queues. |

### Algorithm

#### Main thread

| | |
|---|---|
| Lines - | Create reader and writer threads. |
| Lines - | Start threads. Allows them to enter the queue. |
| | Signals the first in the queue to begin. |

Readers thread - Read()

| | |
|---|---|
| Lines - | Enter queue and wait until awakened. |
| Lines - | Lock readCount mutex, and increment it. |
| Lines - | If readCount is 1, then lock writers. |
| Lines - | Signal next thread to be served. |
| Lines - | Unlock readCount mutex |
| Lines - | Do reading |
| Lines - | Lock readCount mutex, and decrement it. |
| Lines - | If readCount is 0, then unlock writers. |
| Lines - | Unlock readCount mutex. |
| REPEAT | Until stop = true |

Writers thread – Write()

| | |
|---|---|
| Lines - | Enter queue and wait until awakened. |
| Lines - | Wait for all readers to finish reading so writer can lock writers mutex. |
| Lines - | Do writing. |
| Lines - | Signal next thread to be served. |
| Lines - | Unlock writers mutex. |
| REPEAT | Until stop = true |

Main thread

| | |
|---|---|
| Lines - | At 10 seconds, make stop = true. |
| Lines - | Wake up any sleeping readers and writers. |
| Lines - | Join threads. |
| Lines - | Clean up. |
| Lines - | Exit gracefully. |

Eliminating Starvation

The final solution uses the STL queue data structure to store all the reader and writer threads. The nature of the queue dictates a first-in-first-served flow.

- If a writer is at the front of the queue, we ensure all readers complete their read loop and unlock the writer's mutex, then we may write to the resource.
- If a reader is at the front of the queue, we can read the resource if there is no writer writing, otherwise we wait.

Avoiding Deadlocks

This solution simultaneously has

- Mutual exclusion: Mutex locks are used when threads access a global variable.
- Hold and Wait: The read thread, whilst locking the readCount mutex, also locks the writer mutex.
- No preemption.
- No circular wait prevention.

So, the solution does not strictly follow the deadlock prevention rules, yet it does not deadlock.

In the entry section of the read thread, the thread locks the readCount mutex, and also the writer mutex, before releasing the readCount mutex. The writer mutex may already by held by a writer thread.
The reason this does not deadlock is because the writer thread does not depend on any other lock, and so will guarantee that it will unlock the mutex.

## Issues and Limitations

Limitations

(1) This solution has equal priority given to readers and writers.
We use a queue which does first-come-first-serve flow for the next thread to serve. This solution cannot be extended to favour readers or writers over the other.

(2) We assume that the queue has readers and writers shuffled randomly.
However, if for example, 5 readers join the queue followed by 5 writers, we will lose the interweaving of reader and writer operations and instead have 5 writers and 5 reader operations in sequence. Although at least no thread is starved.

(3) Overall number of threads serviced is reduced.
The naïve solution to this problem, where read threads starve write threads, efficiently uses CPU time. At all points during the program, a thread is serviced albeit mostly read threads.

In the final solution, in order to avoid writers' starvation, we guarantee a writer to write if it is at the top of the queue. This is done by waiting for all reader threads to finish reading, thus unlocking the writer lock, then proceeding to write. During the process of waiting, no more threads can be popped from the queue.

Overall, we have less threads serviced, but a much better mix of threads is achieved.

Potential Solutions

Use a scheduling algorithm to manage the queue.
For (1), (2) and (3), the algorithm:

- Orders the threads in a way that will reduce estimated time wasted.
- Strives to order the best mix of readers and writers, based on which it favours more
- Adjusts the queue priority pre-emptively to take into account thread age, so that unfavoured threads are not neglected.

For example:
If Readers more important than Writers then,
Update at each clock cycle:
Reader priority = R * age
Writer priority = W * age
Where R > W.

## Real-World Applications

The readers-writers problem demonstrates a problem where multiple people are accessing a resource at the same time. Accessing a resource could be either reading or writing to a file. Content managers and collaborative applications have become necessary for academic and business purposes.

Shared Documents
Real-time collaboration documents (or 'shared documents') allow for groups of people to share and work on the same file at the same time. Popular implementations include Google's Docs, Slides, Sheets and Microsoft's Word, PowerPoint, and Excel using Co-authoring.

At its simplest, the shared documents problem similar to the readers-writers problem.

| | Shared Documents | Reader-Writers Problem |
|---|---|---|
| Shared Resource | A document stored on the cloud: .doc, .csv, .ppt, etc. | An integer |
| Reader Threads | Users with the document open are serviced with a reader thread. | Read() |
| Writer Threads | Users writing to the document are serviced with a writer thread. | Write() |

Every user with the document open is a reader and a writer (assuming they have required permissions). When the user makes a write, the web app sends the change to the server, which then runs a thread/process to make those changes in the back end.
Once a change has been committed, read threads/processes run to read the document, and return the data to all the document viewers, updating the documents on-screen.
Also, when a new user joins the document, a read thread/process is run to get the up-to-date version for that user.

It is clear that in the back end, shared documents need to process multiple read and write requests by creating threads that access/modify the data.

Databases
Databases are used to store and retrieve data records. A highly used application which connects to a database may have people reading records and writing to them as well. An example would be an e-commerce website where a purchase writes an order to the database, and the warehouse packagers read the list of all orders to fulfill them.

Ensuring atomicity of reads and writes to a database is usually not up to the programmer, but the implementation of the database management system (DBMS) used. When receiving multiple requests on a database, the DMBS ensures that only one request edits the database at a time.

| | Database | Reader-Writers Problem |
|---|---|---|
| Shared Resource | Database tables | An integer |
| Reader Threads | “SELECT” command from user. | Read() |
| Writer Threads | “INSERT” command from user. | Write() |

It is clear that the logic of the readers-writers problem needs to be translated by the DBMS developer to ensure no race conditions occur.

# 2 - Sleeping-Barbers Problem

## Progression

Initial Solution (+ Multiple Seats)

One customer’s thread which, at random times, creates a new customer, then wakes up the barber thread if the number waiting is equal to 1. If the number waiting is less than the size of the array, the customer places itself into the array of customers to be served. Otherwise, it prints a message saying that they will return later.

One barber thread which goes to sleep if there are no customers waiting and is woken up when by the customers thread when the number waiting increments to 1.

More Barbers

Supporting more barbers means having more condition variables to manage their sleep state. We have an array of condition variables, and each barber uses the array index equivalent to their thread ID.

When a barber goes to sleep, they add themselves to a queue to be woken up by the customers.

The customer’s thread now needs to wake up the barber at the front of the queue whenever the number waiting is 1.

We ensure against race conditions by introducing a mutex for the queues and condition variables.

Equal Work

I change the queue into a priority queue which takes a tuple of 1. Total customers served, 2. Condition Variable, 3. Thread ID.
In this way, the next barber that is woken up to take a job is the barber that has done the least work.

Service queue added so the customers are served FIFO.

## Final Solution

### Global Variables

| Name | Purpose |
|---|---|
| waiting[];<br>waitingLen = 0; | Resource to count the customers in the shop. |
| stop = false; | Stop program after 10 seconds. |
| barberLocks[];<br>barberCondVars[];<br>barberSleep[]; | Arrays of lengths NUM_BARBERS which store the mutex locks, condition variables, and Boolean loop* variables respectively.<br><br>*(Boolean loop does not imply busy waiting, I use them to ensure that the threads don't awaken due to other reasons apart from the signal) |
| barberPQueue; | The priority queue which stores the condition variables of the barbers using a min-heap for the number of jobs completed. |
| mutex | Used to lock when waitingLen and waiting[] variables are edited. |
| serviceQueue | Used to implement first-in-first-served for customers. |
| startThreadsLock<br>startThreadsCond<br>waitStart = true; | Pause threads until all threads are created. (to make the early output more readable) |

### Algorithm

Main thread

| | |
|---|---|
| Lines – | Create barbers and customer threads. |
| Lines - | Start the 10 second timer. |

Continued…

Customers thread - Enter()

| | |
|---|---|
| Lines - | Lock mutex. |
| Lines - | Check if full. |
| Lines - | If shop is not full, add the customer to the queue. Otherwise come back later. |
| Lines - | If the customer is added and the number of waiting == 1, then wake up the next barber in the barber's queue. |
| Lines - | Unlock mutex |
| Lines - | Pause for a random number of milliseconds. |
| REPEAT | Until stop = true |

Barbers thread – Serve()

| | |
|---|---|
| Lines - | Lock mutex. |
| Lines - | Check for customers. |
| Lines - | If there are customers, assign the next customer to self. |
| Lines - | Unlock mutex |
| Lines - | If assigned customer to self, cut the customer's hair for a random number of milliseconds, and increment the total served. |
| Lines - | Lock mutex. |
| Lines - | Check the waiting room for more customers. |
| Lines - | If there are customers, unlock mutex and repeat the loop. |
| Lines - | Otherwise put the thread to sleep and add the thread to the queue to wake up later. |
| Lines - | Wake up when signal sent. |
| REPEAT | Until stop = true |

Main thread

| | |
|---|---|
| Lines - | At 10 seconds, make stop = true. |
| Lines - | Wake up any sleeping barbers. |
| | Join threads. |
| | Print the number of customers served. |
| | Exit gracefully. |

Avoiding Starvation

Each barber is placed into a priority queue when they go to sleep, this is to ensure the work is spread out evenly. The priority queue works on a min-heap where the next barber to awake is the one with the fewest jobs completed. So, the algorithm does in-fact starve barber threads, but only those who have done more work than the others.
This starvation is intended behaviour and does not lead to some threads being unable to execute at all. That is, a starved thread will move further up the queue over time, so they will eventually execute if required.

Avoiding Deadlocks

For all the code in this solution, there are no sections of code which, while locking a resource, waits for another resource (hold-and-wait). Since the hold-and-wait condition does not occur in the code, no deadlocks can occur.

## Issues and Limitations

Limitations
Fair work solution is not perfect. The priority queue is used to balance the number of customers served per barber. This is done by making the least worked barber be waked up in the event that the shop becomes non-empty.
When there is a high frequency of customers, barbers have less time to sleep. So the functionality of the priority queue does not come into use often enough to balance the work perfectly. In contrast when there is a low frequency of customers, the balancing works well.

| Conditions | Difference<br>(Highest – lowest performing barber) |
|---|---|
| Run time = 10 seconds,<br>Barbers = 3<br>Random customer frequency = 0-250ms | 4 |
| Run time = 10 seconds,<br>Barbers = 3<br>Random customer frequency = 0-500ms | 1 |

Potential Solution
We can keep track of all the barber's performance globally, then when a customer enters the store, they go into the lowest performing barber's personal queue.

## Real-World Applications

If a process decides to enter a busy wait while-loop until a condition is satisfied, valuable CPU cycles are wasted to upkeep that process.
The sleeping barber problem demonstrates how we can put processes to sleep and only using CPU time when required.

Many-to-many Kernel Thread Model
An operating system needs to provide kernel threads for software to do kernel level commands such as allocating resources and managing memory.
In the many-to-many model, when a user thread requests access to the kernel, they connect to one of the kernel threads in the thread-pool.

Let's compare this to the sleeping barbers problem:

| | M-T-M Kernel Thread Model | Sleeping Barbers |
|---|---|---|
| Workers | Kernel Thread-Pool | Barbers |
| Work Makers | User Thread requests for kernel<br>OR<br>Light Weight Processes (LWP) scheduler | Customers |

Like the sleeping barber problem, we have worker threads who go to sleep if there are no user-threads requesting kernel operations.

Only when that signal is made does the kernel thread awaken to perform any necessary CPU operations. After the user-thread no longer requires the kernel thread, the kernel thread is put back to sleep awaiting the OS the wake it up again at a later date. Sleeping these threads allows for more CPU resources available to the user.

## Webserver Architecture

Webservers require the ability to service multiple users. A webserver without concurrency is useless, as only one person can do things at a time. When a user makes a request to login, the server dispatches a thread to authenticate that user. That thread is sourced from the server's thread-pool.

Let's compare this to the sleeping barbers problem:

| | Webserver Architecture | Sleeping Barbers |
|---|---|---|
| Workers | Server Thread-Pool | Barbers |
| Work | Process Web Requests | Cut Hair |
| Work Makers | Website Users | Customers |

Like the sleeping barber problem, server threads manage the actions of users, and when the number of users is less than the number of server threads, these threads go to sleep so as to reduce on cloud computing prices.

Typically, the programmer does not have to do multi-programming, the developers of webservers like Apache's Tomcat must implement concurrency at the low-level.