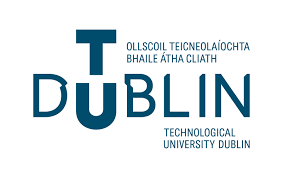


****

**Readers-Writers Problem**

By

Peter Kinsella

30/11/2019

This Report is submitted in partial fulfilment of the requirements of the Honours Degree in Electrical and Electronic Engineering (DT021) of the Technological University Dublin

Contents

[Introduction](#_30j0zll) 2

[Objective](#_1fob9te) 2

[Theory](#_3znysh7) 2

[Readers-Writers Problem](#_2et92p0) 2

[Solution](#_jvg402fem1sz) 3

[Readers Preference](#_3dy6vkm) 3

[Writers Preference](#_4d34og8) 4

[Results](#_1y810tw) 4

[Associative Memory](#_2xcytpi) 5

[Direct Mapped](#_3whwml4) 5

[2-way Set Associative](#_qsh70q) 5

[Comments and Conclusion](#_49x2ik5) 5

[Bibliography](#_2p2csry) 5

[Appendix A: Code](#_147n2zr) 6

[Readers Preference, Reader](#_v116frw7qr5j) 6

[Readers Preference, Writer](#_d1neuwyfiruf) 6

[Writers Preference, Reader](#_9b1byf7udthx) 6

[Writers Preference, Writer](#_268kj1iuf3jb) 6

List of figures

[Figure 1 Associative memory cache design [1] 2](#_tyjcwt)

[Figure 2 Direct mapped cache design [2] 3](#_1t3h5sf)

[Figure 3 2-way set associative cache design [3] 4](#_2s8eyo1)

[Figure 4 Struct to represent SRAM 5](#_26in1rg)

[Figure 5 operation of simulation 5](#_lnxbz9)

[Figure 6 Struct to represent SRAM 6](#_1ksv4uv)

[Figure 7 Operation of the Direct Mapped simulator 6](#_44sinio)

[Figure 8 Structs to represent the two SRAM's for addresses 7](#_z337ya)

[Figure 9 Operation of the 2-way Set Associative simulator 7](#_3j2qqm3)

[Figure 10 Expected hits and misses for each simulation 8](#_4i7ojhp)

[Figure 11 Associative memory results 8](#_1ci93xb)

[Figure 12 Direct mapped cache design results 9](#_2bn6wsx)

[Figure 13 2-way set associative cache design results 9](#_3as4poj)

# Introduction

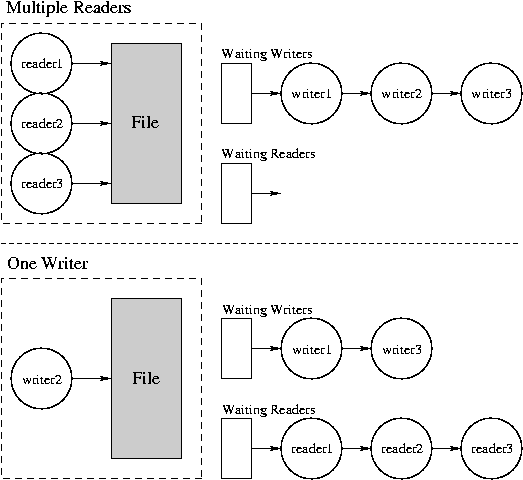
## Objective

The aim of this assignment is to simulate the function of three different cpu cache storage systems, associative memory, direct mapped and 2-way set associative.

## Theory

### Readers-Writers Problem

The readers-writers problem is an example of a common issue in computing systems where concurrent threads are trying to access a shared memory space and perform actions on it. It the case of this example, the two possible types of threads are readers and writers [1 wiki]. For this example, the aim is to be able to allow readers to access and read the shared memory space when it is free but when a writer is accessing the memory space, no other thread will be allowed to access it. This was done using semaphores. A semaphore is a shared variable amongst threads that can be used to control access to a shared memory space or resource [2 techopedia]. Binary semaphores are similar to a mutex in the way that they can be used to lock and unlock a thread, making it so no other thread can perform an action whilst the current thread is performing a process. Though the advantage to using semaphores over a mutex is that one can use counting semaphores as well. Semaphores can have a value other than 1 or 0 allowing them to be able to control multiple threads at the same time depending on the value. This is perfect for the readers and writers problem to allow multiple readers to concurrently read from the shared memory space. This is depicted in the following image.



*Figure 1 Diagram depicting problem [1]*

# Solution

## Readers Preference

The idea behind readers preference to to make a more optimal solution to the problem. If one reader were to lock out another and force the second reader to wait it would be a waste of time. It would be more beneficial to allow any amount of readers to read concurrently as a read action does not affect the shared memory. This is done by breaking the readers process up into three sections, the entry section, the critical section and the exit section. The entry section is where the writer is locked out of being able to write. In this section only one reader is allowed to perform the action at a time. This is to avoid race hazards. Therefore the reader will lock out all other processes. Then the critical section will lock the writer from writing and finally, release the semaphore that locks out other readers so that they may begin their process at the same time. This can be seen in the following code.

*Figure 2 Readers entry section, readers preference*

Once this is done the reader is finally allowed perform their reading action. In this case it involves opening up a shared text file and reading each line that has been written in by a writer line by line and printing it out to the terminal. This can be seen in the following figure.

*Figure 3 Readers reading section, readers preference*

Next, the exit section is performed similarly to the entry section. Here again the reader locks the semaphore thus restricting any other reader from from performing the same action. Once this is done the process is able to enter the second critical section where the writer semaphore is unlocked. However, if more than one reader were already reading then this is where the counting semaphore is used. Instead of one reader locking out the other one, it instead just decreases the value of the counting semaphore. Therefore, the writer won’t be unlocked until the value of the counting semaphore becomes 0 meaning that all readers have finished their read. This section can be seen in the following section of code.

*Figure 4 Readers exit section, readers preference*

Finally, the writer in this solution has a fairly simple process. It consists of the same three sections but they are much less complex. In the entry section, the writer locks the writing semaphore restricting every process from performing an action. Therefore, only one writer can work at a time. Then the critical section is where the writing is done. Here, the process will open up the shared file and append on the entered word that a user inputs onto a new line. If it was unable to open the file it prints out a suitable statement to the terminal. Finally in the exit section, the process releases the semaphore allowing other processes to perform their actions. All of this can be seen in the following figure.

*Figure 5 Writer, readers preference*

## Writers Preference

This solution was created as the readers preference is suboptimal. This is because readers have priority over writers in that solution and if enough readers keep wanting to read they will keep the writer waiting even if it joined the queue before the majority of the readers. Therefore the writers preference solution was created to make it so that a writer will only have to wait for the minimal amount of time necessary.

In this solution the writer program is very similar to the reader program in the readers preference. It has a very similar entry section where it first locks out all other writers using a writer specific semaphore, and then proceeds to lock out all readers that are trying to join the queue using a new semaphore. This can be seen in the following figure where it can be seen how similar it is to the readers preference entry section, just using different semaphores.

*Figure 6 Writers entry section, writers preference*

After the entry section, the critical section contains the entirety of the old writers program from the readers preference solution which locks out every other process from accessing the shared memory, writing to the memory and then releasing the semaphore.

Next in the exit section, it is again very similar to the readers preference in that it decreases the value of the writing semaphore, waits for all writers to have their turn writing that were in the queue, and then releases the readers locking semaphore so that they may perform their processes and finally unlocks the writer semaphore. This can be seen in the following figure.

*Figure 7 Writers writing section, writers preference*

Finally, the readers program has a small change from the readers preference version. The only difference is in the entry section. Whilst it still performs the same action of locking all processes out so that it can lock out the writers, it has a new check at the beginning. The first thing it does is perform a lock on the readers semaphore which the writer uses to lock out all readers from taking priority. If it is already locked then the reader can’t join the queue and has to wait for the writer to release the semaphore. Once it is free it is allowed to perform its process of locking out the writers and then releasing the readers semaphore so the the other readers can still read concurrently. This all can be seen in the following piece of code.

*Figure 8 Writers exit section, writers preference*

# Results

*Figure 9 Operation of the 2-way Set Associative simulator*

*Figure 10 Expected hits and misses for each simulation*

## Associative Memory

*Figure 11 Associative memory results*

## Direct Mapped

*Figure 12 Direct mapped cache design results*

## 2-way Set Associative

*Figure 13 2-way set associative cache design results*

# Comments and Conclusion

From this lab it can be seen how the readers-writers problem can be solved in two different ways using a series of semaphores to lock and unlock processes. Not only do these work for this example but can be implemented for most reading and writing programs in an operating system. There is a third solution that was not gotten to in this lab that in future would be good to test where no process is allowed to “starve” or be stuck waiting. Therefore it is almost seen as a fair queueing system where both readers and writers have the same priority and get done in the order that they enter the queue.

# Bibliography

# Appendix A: Code

## Readers Preference, Reader

## Readers Preference, Writer

## Writers Preference, Reader

## Writers Preference, Writer