Synchronization and multithreading

ABSTRACT

Computers can be found everywhere around us today, anywhere from inside a coffee machine to the airplanes that transport us around the globe. As such it is critical to design software that are efficient, fast and reliable in order to accommodate the increasing need of new and advanced computer systems. One way of designing software that fit those criteria is to take advantage of multi-core processing through multithreading. Multithreading brings forth a set of challenges that designers must be cautious and aware of when designing software.

This paper will mainly focus on synchronization problems and will present two of these problems, and they are The Producer–Consumer Problem (also called The Bounded Buffer Problem), The Readers–Writers Problem, and it will also try to present common solutions that designers use to tackle these problems.

Introduction

With the increasing digitalization that our world faces, the pressure on designers of computer software to be able to deliver software capable of handling an enormous amount of information efficiently, quickly, reliably and affordably, drastically increases. The software needs to be robust enough to handle multiple events at the same time with as close to 100% uptime as possible. So how do designers tackle these problems? They do so by taking advantage of all the power that comes from cores of a modern processor, known as a multi-core processor, through a method known as multi-threading.

Multi-threading is a powerful yet dangerous tool. That if used correctly could improve performance several times over but on the other hand if used incorrectly could cause problems such as data races, deadlocks or corrupted data. In other cases, the over use of multi-threads could make performance worse. The key to multithreaded software is proper use of synchronization.

With proper synchronization, a lot of the problems plaguing the use of multithreaded software could be avoided. Why these problems occur and how they are related to synchronization will be explained followed by how they are solved.

Problem formulation

In this section the Producer–Consumer Problem (also called The Bounded Buffer Problem) and the Readers–Writers Problem will be explained and their respective solution(s) will be presented.

The Producer–Consumer Problem

To best describe The Producer-Consumer problem would be through an example. Imagine two distinct processes, the producer and the consumer, both operating on a single shared buffer. One of them, the producer, inserts data to the buffer whilst, the consumer, retrieves data from the buffer. What would
happen if the producer tries to insert data when the buffer is full? What would happen if the consumer tries to retrieve data when there is none in the buffer? As you could imagine there would be fatal errors.

The Producer-Consumer problem is about how to manage the relationship between the producer and the consumer, by only allowing the producer to insert data when there is space in the buffer and only allowing the consumer to retrieve data when there is data in the buffer.

**Proposed solutions to the Producer-Consumer Problem**

One proposed solution to the Producer-Consumer problem is to make use of a sleep and a wakeup call in order to control the producer and the consumer. An example of this proposed solution can be found in the image to the left. But one would soon find problems with this implementation as it would create a race condition that could lead to a deadlock. Consider the following scenario:

1. The Consumer is executing. It reads the variable count as 0 and proceeds to sleep().
2. Just before executing sleep(), the consumer is interrupted and the producer starts executing.
3. The producer creates an item, puts it in the buffer and increments count.
4. Now as the count variable is 1, the producer tries to wake up the consumer but as the sleep call for the consumer was interrupted it never went to sleep and as such the wakeup call is lost. When the consumer eventually resumes it goes back to the sleep call never to have another wakeup call again due to the producer only ever calls a wakeup for the consumer when the count variable is equal to 1.
5. As the count variable now being over 1 and the consumer being asleep, the count variable would never reach 1 again and thus the producer would run until the buffer reached its maximum size and the producer will then be put to sleep and as such a deadlock have been reached.
To prevent the case of deadlocks as shown in the previous proposed solution the use of semaphores are utilized as seen in the next proposed solution.

```c
semaphore available = MAX_SLOTS;
semaphore occupied = 0;
Producer()
{   while (true){
        produceItem(item);
        wait(available);
        enterItem(item);
        signal(occupied);
    }
}
Consumer()
{   while (true){
        wait(occupied);
        removeItem(item);
        signal(available);
        consumeItem(item);
    }
}
```

In this proposed solution two semaphores instead of a count variable are used. This solution works well when there is only one producer and one consumer but runs into problems when there are multiple producers or consumers.

To demonstrate the issues with multiple producers and consumers, imagine the following scenario:

1. Two producers call the wait function.
2. The first producer finds the next empty slot in the buffer and the second producer finds the exact same empty slot as the first producer.
3. Both inserts at the slot in the buffer resulting in the loss of an item.

A real world system should be able to handle the use of multiple producers and consumers, so in the final proposed solution another semaphore is introduced in order to make sure that only one producer calls enters a item into the buffer at a time. The final solution for the use of multiple producers and consumer is as follows:

```c
semaphore available = MAX_SLOTS;
semaphore occupied = 0;
semaphore mutExcl = 1;
Producer()
{   int item;
    while (true){
        produceItem(item);
        wait(available);
        wait(mutExcl);
        enterItem(item);
        wait(mutExcl);
        signal(mutExcl);
        signal(occupied);
    }
}
Consumer()
{   int item;
    while (true){
        wait(occupied);
        wait(mutExcl);
        removeItem(item);
        signal(available);
        signal(mutExcl);
        consumeItem(item);
    }
}
```
**The Readers-Writers problem**

The Readers-Writers problem is a problem where there are multiple readers and writes accessing the same shared resources. If a writer is accessing the shared resources no other writer or reader may access the shared resources. If another writer were to access the shared resources during this time there could be corrupted data or if a reader accessed the shared resources during this time it could read a partially modified value. If on the other hand a reader is accessing the shared resources other readers may also do so.

Three different Readers-Writers problem variants will be examined and they are as followed:

1. **Readers-preference**: If one reader is accessing the shared resources, new readers may also do so (may cause writer starvation).
2. **Writers-preference**: New writers shall have to wait no more than necessary (may cause reader starvation).
3. **No-preference**: Neither writer nor reader shall be allowed to starve.

**Proposed solutions to the Reader-Writer problem**

```c
semaphore rw_mutEx = 1;
semaphore mutEx = 1;
int read_count = 0;

writer()
    {wait(rw_mutEx);
     /* Write */
     signal(rw_mutEx);
    }

read()
    {wait(mutEx); //line 01
     read_count++;
     if(read_count == 1){
         wait(rw_mutEx); //line 02
     }
     signal(mutEx);
     /* read */
     wait(mutEx); //line 03
     read_count--;
     if(read_count == 0){
         signal(rw_mutEx); //line 04
     }
     signal(mutEx);
    }
```

The first variant’s solution, **Readers-preference**, can be seen on the left and to demonstrate how it gives preference to readers imagine the following scenario:

1. A reader arrives and at line 01 it claims `mutEx` and increments `read_count`. As there are no other readers or writers, the reader will also claim `rw_mutEx` at line 02.
2. While the reading for the first reader is performed a second reader arrives. The second reader claims `mutEx` (as it was released by the first reader just after line 02), increments `read_count`, skips line 02 and resets the signal `mutEx`.
3. Now a writer arrives and waits for the `rw_mutEx` signal to be clear, but `rw_mutEx` is only ever released by the readers when all of the readers are finished reading (at line 04). Now if a third, fourth, fifth or even more readers were to arrive the time for `rw_mutEx` to clear would increase thus creating writer starvation.
The second variants solution, Writers-preference, can be seen below. To demonstrate the preference for writers imagine the following scenario:

```c
int readerCount, writerCount;
semaphore mutEx_1, mutEx_2, mutEx_3, r, w;

read(){
    wait(mutEx_3);     //line 01
    wait(r);
    wait(mutEx_1);
    readerCount++;     
    if(reader_count == 1){
        wait(w);
    }
    signal(mutEx_1);   //line 02
    signal(r);
    signal(mutEx_3);   /* read */

    wait(mutEx_1);     //line 03
    readerCount--;     
    if(readerCount == 0){
        signal(w);
    }
    signal(mutEx_1);
}

write(){
    wait(mutEx_2);     //line 04
    writerCount++;     
    if(writerCount == 1){
        wait(r);
    }
    signal(mutEx_2);   //line 05
    wait(w);
    /* write */

    signal(w);         //line 06
    wait(mutEx_2);
    writerCount--;     
    if(writerCount == 0){
        signal(r);
    }
    signal(mutEx_2);
}
```

1. A writer starts executing and at line 04 it claims mutEx_2, increments writerCount and checks if it is the first writer, if that is the case it claims r.
2. At line 05 the writer releases mutEx_2, claims w and proceeds to write.
3. If a reader now arrives it will claim mutEx_3 at line 01 but will be forced to wait when trying to claim r until the first writer has returned r.
4. Now if more and more writers arrive the reader will be forced to wait until the last writer returns r and thus a reader starvation would occur.
The third variants solution, **No-preference**, can be seen below. The focus in this solution is on allowing readers and writers to execute in the order they arrive, as to not cause any starvation:

```c
semaphore accessMutEx;
semaphore readersMutEx;
semaphore orderMutEx;
int readers = 0;

read()
{
    wait(orderMutEx);
    wait(readersMutEx);
    if(readers == 0){
        wait(accessMutEx);
    }
    readers++;
    signal(orderMutEx);
    signal(readersMutEx);

    /* read */
    wait(readersMutEx);
    readers--;  
    if(readers == 0){
        signal(accessMutEx);
    }
    signal(readersMutEx);
}

write()
{
    wait(orderMutEx);
    wait(accessMutEx);
    signal(orderMutEx);

    /* write */
    signal(accessMutEx);
}
```

**Conclusions**

As seen in the presented synchronization problems, the solution is not always so clear-cut. A solution must be tailored for the specific task at hand. But if the solution for a specific problem makes use of proper synchronization through various methods such as those seen in the proposed solutions, then the benefits far outweighs the increased execution times.
References
