1 2010 Paper 5 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2010p5q4.pdf

In an application, processes may be identified as "readers" or "writers" of a certain data object. Multiple-reader, single-writer access to this object must be implemented, with priority for writers over readers. Readers execute procedures *startread* and *endread* before and after reading. Writers execute procedures *startwrite* and *endwrite* before and after writing one-at-a-time.

The following variables are used in an implementation of the algorithm:

ar	is the count of active readers
rr	is the count of reading readers
aw	is the count of active writers
ww	is the count of writing writers (who write one-at-a-time)

(a) For mutual exclusion:
SemCountGuard is a Semaphore under which the above contents are read and written.
SemWrite is for writers to wait on, in order to write one-at-a-time.

For condition synchronisation:
SemOKtoRead is for readers to wait until all writers have finished.
SemOKtoWrite is for writers to wait until currently reading readers have finished.

Discuss the following pseudocode for an attempted implementation of *startread*:

```
procedure startread()
wait(SemCountGuard);
ar := ar + 1;
if aw > 0 then wait(SemOKtoRead);
rr := rr + 1;
signal(SemCountGuard);
return;
```

This code will deadlock or fail to implement priority correctly. startread waits on SemOKtoRead while still holding onto SemCountGuard. This will either prevent any writing writer from decrementing ww or force it to signal SemOKtoRead first – which would allow new readers to start before the next writer.

Additionally, startread does not update SemOKtoWrite. startread changes the number of readers – if we go from 0 to 1 readers, we should acquire SemOKtoWrite so that writers cannot start. This will prevent writers writing while readers are reading. If startread does not updates SemOKtoWrite, then it is redundant and writers will have to spinlock, repeatedly polling until there are no readers left.

(b) Using the above example, comment on the ease of monitor programming and implementation, compared with Semaphore programming. Assume a monitor *ReadersWriters* defines condition variables *SemOKtoRead* and *SemOKtoWrite*.

Monitors are blocks of code with the requirement that at most one thread be executing any of them. This is a very intuitive way of thinking about concurrency and makes programming far easier.

Monitors have condition variables. These are queues of threads which are waiting for particular predicate to be true. These allow us to wake threads waiting on conditions when that condition becomes true without having those threads repeatedly retrying the condition. In this example, the condition is simple, however it can be arbitrarily complex and computationally expensive.

When a thread starts waiting on a condition variable, it will release the monitor. This allows other threads to enter the monitor. Therefore, threads which wait on condition variables must ensure all shared objects are in consistent states. This can lead to bugs.

It is important to remember wait on a condition inside a `while` loop rather than an `if` clause. Waiting on a condition variable inside an `if` clause means the condition can be changed to `true`, then changed back to `false` by another thread and the thread waitin on the condition variable may continue as if the condition is still true. This is a common source of bugs.

Monitors are a very easy way of getting good performance when code is written well. However, it is very easy to forget to signal a condition variable (leading to threads waiting unnecessarily) or to signal it when the predicate hasn't actually changed (leading to wasted work).

Furthermore, monitors can overly serialise code. Consider the example below. It's safe for Readers to become active (not reading) and writers to become active at the same time. However, under this monitor implementation that cannot occur.

Semaphores require a much deeper understanding of the system to implement properly. It's very easy to deadlock or have race conditions when using Semaphores – and avoiding either requires thorough analysis of code. Semaphores give much lower-level control of the code and can therefore be more efficient than monitors.

```
monitor ReadersWriters{
        int ar=0, rr=0, aw=0, ww=0;
        condition SemOKtoRead, SemOKtoWrite;
public:
        void startread(){
                ar++;
                while (aw){
                        wait(SemOKtoRead, ReadersWriters);
                }
                rr++;
        }

        void endread(){
                ar--;
                rr--;
                if (!rr){
                        signal(SemOKtoWrite);
                }
        }

        void startwrite(){
                aw++;
                while (rr){
                        wait(SemOKtoWrite, ReadersWriters);
                }
                ww++;
        }

        void endwrite(){
                ww--;
                aw--;
                if (aw == 1){
                        signal(SemOKtoRead)
                }
        }
};
```

(c) Describe and comment on the Java approach to supporting mutual exclusion and condition synchronisation.

The Java primitive "synchronized" implements monitors. It can be passed an object and Java will by default create a mutex around it which will prevent any other procedure to run a "synchronized" block. For instance readers could synchronize on a shared counter before changing the ar or aw counts.

It's very common to synchronize on the item itself so there is additional syntactic sugar to simplify this. The following two functions are identical – both take out a mutex on the object itself.

```
public T1 f(T2 val){
        synchronized(this){
                ...
        }
}

public synchronized T1 f(T2 val){
        ...
}
```

Java mutual exclusion is intuitive and removes a lot of low-level implementations, leaving that to the compiler. However, mutual exclusion and concurrency control in general is still difficult.

(d) Explain how active objects and guarded commands avoid some of the issues arising in the above programs.

Active Objects are objects which execute on a single thread. Other threads can send messages to Active Objects requesting the execution of methods with particular arguments. Messages are added to a queue; a scheduler will then decide which method-/argument pair to execute next. Active objects guarantee thread safety by executing serially (only on one thread). However, they often overly-serialise leading to slower execution. We can execute requests to operate on shared data in a single active object and therefore execute serially preventing race conditions while also choosing which thread to execute.

Active Objects can have schedulers to choose which threads on the queue to execute in which order and also provide safety. With an appropriate scheduler, we can give requests from different types of writer and reader different priorities therefore giving priority in the order reading readers, writing writers, active writers, active readers. This priority order would implement priority of writers over readers in the MRSW program above while also providing safety as the critical sections are being executed serially on one thread inside the Active Object.

Guarded commands are a set of boolean predicates with associated statements. Firstly all the boolean predicates are evaluated and if any evaluated to true then exactly one associated statement is executed via random choice. If none are true, then none are executed. Guarded commands execute one statement where the predicate is true randomly.

Guarded commands are used in message passing to ensure fairness between processes with the same priority. We have a guarded command with conditions as the prerequisites to each message and the conditions being carrying out the message. Therefore when multiple messages can be carried out, the choice is made randomly and hence fairly. This helps avoid processor starvation. We could use this to ensure that writers did not wait on many readers – if we had a guarded command for reader and writer messages, then when a writer arrives it will only on average have to wait for one reader to finish and as the clause is executing it will prevent further readers starting. We can therefore implement MRSW with priority trivially with guarded command.

```
guarded_command{
        (writer){
                /* while there exists a writer, wait until its safe to write
                 * and then write, blocking the controller from creating any more
                 * readers or writers until the current writer has finished */
                while (writer){
                        wait(readers);
                        write();
                }
        }
        (reader){
                /* spawn a new thread which reads so we can have multiple readers
                 * executing at the same time. We need a count of readers to know
                 * when its safe to start writing. */
                spawn(read);
        }
}
```

2 Implementation of other procedures

Write out the other three methods (endread, startwrite and endwrite) and state exactly what you are using the four variables ar, rr, aw and ww for.

Rather than implementing 3 and trying reason about the behaviour of the 4th, I decided to implement all 4 methods.

Writers take priority over readers in all situations. New readers cannot start until there are no active writers. On creation, a new writer only has to wait for reading readers or other writers. There is no situation where a reader starts reading while there is an active writer.

I used only four Semaphores. However, I did not use them for the same things as in the question. I have therefore renamed them:

- ReaderGuard

  This locks updates to ar and rr.

- WriterGuard

  This locks updates to ww

- ReaderLock

  This is used to implement priority, preventing new readers from starting until there are no active writers.

- WriterLock

  This is used to prevent multiple writers writing at once and to prevent writers starting writing while there are still any reading readers.

Uses of the four variables:

- ar

  ar is a count of the number of active readers. This is not necessary for the implementation – it is only kept up to date for consistency. ar is locked by ReaderGuard.

- rr

  rr is a count of the number of readers which are reading from the file. This is used to keep a count of how many readers the active writers are waiting for. When the number of reading readers changes from 0 to 1, WriterLock is acquired so that new writers cannot start writing while readers are still reading. When the number of reading readers decreases to zero, WriterLock is signalled allowing writers to start writing. rr is locked by ReaderGuard.

- aw

  aw is the count of the number of writers which want to write to the file. When writing writers finish writing, they check if aw is zero before releasing ReaderLock and allow new readers. New writers check aw before attempting to acquire ReaderLock. aw is locked by WriterGuard.

- ww

  ww is the number of writers who are currently writing to the file. It is not necessary for the implementation, it is just kept updated for consistency. ww is locked by WriterLock.

```
class ReadWrite(){
        Semaphore ReaderLock = Semaphore(1);
        Semaphore WriterLock = Semaphore(1);
        Semaphore ReaderGuard = Semaphore(1);
        Semaphore WriterGuard = Semaphore(1);
public:
        void startread();
        void endread();
        void startwrite();
        void endwrite();
};

void ReadWrite::startread(){
        wait(ReaderGuard);
        ar++;
        signal(ReaderGuard);
        wait(ReaderLock);
        wait(ReaderGuard);
        rr++;
        if (rr == 1){
                wait(WriterLock);
        }
        signal(ReaderGuard);
        signal(ReaderLock);
}

void ReadWrite::endread(){
        wait(ReaderGuard);
        rr--;
        if (!rr){
                signal(WriterLock)
        }
        ar--;
        signal(ReaderGuard);
}
```

```
void ReadWrite::startwrite(){
        wait(WriterGuard);
        aw++;
        if (aw == 1){
                wait(ReaderLock);
        }
        signal(WriterGuard);
        wait(WriterLock);
        ww++;
}

void ReadWrite::endwrite(){
        ww--;
        signal(WriterLock);
        wait(WriterGuard);
        aw--;
        if (!aw){
                signal(ReaderLock);
        }
        signal(WriterGuard);
}
```

3 2000 Paper 3 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2000p3q1.pdf

(a) A software module controls a car park of known capacity. Calls to the module's procedures *enter*() and *exit*() are triggered when cars enter and leave via the barriers.

Give pseudocode for the *enter* and *exit* procedures

(i) if the module is a monitor

I interpreted the question to mean "only allow cars to enter when there are spaces and only allow cars to leave when there are cars".

```
monitor CarPark{
        const int capacity;
        int cars;
        condition notfull, notempty;
public:
        CarPark(int cap, int car) : capacity(cap), cars(car) {}
        void enter(){
                while (cars == capacity) {
                        wait(notfull, CarPark);
                }
                cars++;
                signal(notempty);
        }
        void exit(){
                while (capacity == 0) {
                        wait(notempty, CarPark);
                }
                cars--;
                signal(notfull);
        }
};
```

(ii) if the programming language in which the module is written provides only Semaphores

```
class CarPark{
        Semaphore spaces, cars;
public:
        CarPark(int cap, int car) : spaces(cap), cars(car) {}
        void enter(){}
        void exit(){}
}

void CarPark::enter():
        wait(spaces)
        signal(car)

def exit():
        wait(car)
        signal(spaces)
```

(b) Outline the implementation of

(i) Semaphores

Semaphores have an integer variable, a queue of threads and two methods (wait and signal). Operations on both the variable and the queue need to be atomic.

A Semaphore is initialised with a value. The variable is set to this value. From this point the variable is not directly accessible by the user. Calling "wait" when the variable is non-zero decrements the variable atomically (using compare-and-swap or load-linked, store-conditional). Calling "wait" when the variable is zero will atomically place the thread onto the tail of the queue waiting on the Semaphore.

Threads can also call "signal", which will wake the thread on the head of the queue (if the queue is non-empty) or increments the variable held in the semaphore.

Here is an implementation of atomic increment using compare-and-swap.

```
void inc_cas(int *p){
        do{
                int i = *p;
                int j = i + 1;
        }
        while (!compare_and_swap(p, i, j))
}
```

Here are implementations of the two key operations in an atomic linked list (push and pop) using load-linked, store-conditional and compare-and-swap. For the purposes of this example, I have ignored garbage collection so as to avoid mitigations of use-after-free errors. A reasonable strategy would be to use reference counting (the using inc_cas method above) and iterate through the list of popped nodes and delete all which have reference counts of 0. An Atomic Linked List cannot be implemented with only compare-and-swap – consider if a node is deallocated and subsequently reallocated while a thread is sleeping.

```
struct AtomicNode{
        AtomicNode *next;
        thread * thr;
        int refs = 0;
        AtomicLinkedList(thread *t) : thr(t) {}
};

class AtomicLinkedList{
        AtomicNode *head;
        AtomicNode *tail;
public:
        thread *pop();
        void push(thread *thr);
};

thread *AtomicLinkedList::pop(){
        thread *thr;
        AtomicNode *next;
        AtomicNode *tmp;
        AtomicNode **thread tmp2
        while (true){
                load_linked(head, tmp);
                if (!tmp){continue;}
                next = tmp->next;
                if(!store_conditional(head, tmp, next)){
                        break;
                }
                yield();
        }
        return thr;
}

void AtomicLinkedList::push(thread *thr){
        AtomicNode node = new AtomicNode(thr);
        AtomicNode *tl1;
        AtomicNode *hd1;
        while (true){
                load_linked(tail, tl1);
                if (tl1)
                        if(compare_and_swap(tl1->next, nullptr, node) &&
                                store_conditional(tail, tl1, node)){break;}
                }
                else if (!tl1){
                        load_linked(head, hd1);
                        if (!head && compare_and_swap(head, hd1, node) &&
                                compare_and_swap(tail, hd1, node){break;}
                }
                yield();
        }
}
```

(ii) monitors

Monitors can be implemented by attaching a binary Semaphore to an object. When using any method of this object or returning from a wait on a condition variable in the object, wait on the Semaphore; and signal the Semaphore when returning from the method or waiting on a condition variable.

Condition Variables are thread queues which are implemented in the same way as the thread queues in Semaphores.