Harry Langford **, SV**
hjel2@cam.ac.uk ,

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.

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.

```
monitor ReadersWriters:
        ar = rr = aw = ww = 0
        condition SemOKtoRead, SemOKtoWrite
        def startread():
                ar += 1
                while aw > 0:
                        wait(SemOKtoRead, ReadersWriters)
                rr += 1

        def endread():
                ar -= 1
                rr -= 1
                if rr == 0:
                        signal(SemOKtoWrite)

        def startwrite():
                aw += 1
                while rr > 0:
                        wait(SemOKtoWrite, ReadersWriters)
                ww += 1

        def endwrite():
                ww -= 1
                aw -= 1
                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.

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 4^{th}, 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.

```
ReaderLock = Semaphore(1)
WriterLock = Semaphore(1)
ReaderGuard = Semaphore(1)
WriterGuard = Semaphore(1)

def startread():
        wait(ReaderGuard)
        ar += 1
        signal(ReaderGuard)
        wait(ReaderLock)
        wait(ReaderGuard)
        rr += 1
        if rr == 1:
                wait(WriterLock)
        signal(ReaderGuard)
        signal(ReaderLock)


def endread():
        wait(ReaderGuard)
        rr -= 1
        if rr == 0:
                signal(WriterLosck)
        ar -= 1
        signal(ReaderGuard)


def startwrite():
        wait(WriterGuard)
        aw += 1
        if aw == 1:
                wait(ReaderLock)
        signal(WriterGuard)
        wait(WriterLock)
        ww += 1


def endwrite():
        ww -= 1
        signal(WriterLock)
        wait(WriterGuard)
        aw -= 1
        if aw == 0:
                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

This felt very light for 8 marks – however I could not see what else the question could be asking for.

```
public class CarPark{
        private int cars = 0;

        public synchronized void enter(){
                cars++;
        }

        public synchronized void exit(){
                cars--;
        }
}
```

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

```
WriteLock = Semaphore(1)
cars = 0;

def enter():
        wait(WriteLock)
        cars += 1
        signal(WriteLock)

def exit():
        wait(WriteLock)
        cars -= 1
        signal(WriteLock)
```

(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 Semaphores is initialised with a value. The variable is set to this value. 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 wakes the thread on the head of the queue (if the queue is non-empty) or increments the variable.

Here is an implementation of atomic increment using compare-and-swap; and load-linked, store-conditional.

```
void inc_cas(int *p){
        do{
                int i = *p;
                i++;
        }
        while (cmpxchg p, i)
}
```

(ii) monitors

Monitors can be implemented by attaching a binary Semaphore to an object. When entering any method of this object, wait on the Semaphore; and signal the Semaphore when leaving from it. This includes entering and leaving while waiting on Condition Variables. Condition Variables are threads queues which are implemented in the same way as the threads queues in Semaphores.