

COMP2310 Assignment 2

Felix Andrews 3285754

October 11, 2002

Contents

1	A Locking Problem	2
1.1	Simple Mutual Exclusion	2
1.2	Fair Mutual Exclusion	2
1.2.1	Linear Waiting	2
1.2.2	First In, First Out	4
1.3	A Real Instruction	5
2	The Sightseeing Problem	6
2.1	Single Coach	6
2.2	Multiple Coaches	6
3	A Monitor Problem	7
4	A Message-Passing Problem	9
5	Reflection	11
5.1	Time	11
5.2	Easiness	11
5.3	Difficulties	11
5.4	Learning	11

1 A Locking Problem

1.1 Simple Mutual Exclusion

```
shared INTEGER a = 1, c = 0
```

```
pre-protocol:  
  while (FROBNITZ(a,c,c)==0) {}
```

```
post-protocol:  
  a = c + 1
```

The FROBNITZ instruction called with the argument list (a, c, c) can be characterised as the following function:

$$\text{FROBNITZ}(a, a, c) \rightarrow \begin{cases} 0 & \text{if } (a = c) \\ 1 & \text{if } (a \neq c) \end{cases} \quad \text{and set } a = c$$

This simplified behaviour is similar to the test-and-set instruction. It is easy to implement mutual exclusion with it.

Initially, $a \neq c$. A safety property is that $a = c$ iff a process is in the critical region. If a process tries to enter the critical region while it is occupied, it is prevented (held up in the busy-waiting pre-protocol). When the critical region is eventually left, the post-protocol ensures that $a \neq c$. The next process to call FROBNITZ(a, a, c) (one of the waiting processes) then sets $a = c$, thereby gaining the lock, and enters.

1.2 Fair Mutual Exclusion

I will present two attempts at a *fair* solution to the mutual exclusion problem above. There are different types of fairness.

In the following code and discussion, I will refer to a process in the critical region as a *critic* and to a process attempting to enter as a *contender*. There are n processes; it is assumed they have access to their unique index id in the range $[0, n - 1]$. If it is not possible to have local variables, index $[id]$ of a shared array could be used instead.

1.2.1 Linear Waiting

```
shared BOOLEAN[0..n-1] contender = {False, ..., False}  
shared INTEGER critic = -1  
local INTEGER critic_test  
local INTEGER i
```

```

pre-protocol:
    contender[id] = True
    while (critic != id) {
        critic_test = -1
        FROBNITZ(critic_test, id, critic)
    }

```

```

post-protocol:
    contender[id] = False
    for (i = 1..n-1) {
        if contender[(id+i) modulo n] {
            critic = (id+i) modulo n
            break
        }
    }
    if (critic==id) { critic = -1 }

```

The variable `critic` stores the current process `id` in the critical region, or `-1` if it is free. When a process wants to enter, essentially it ‘puts up its flag’ (`contender[id]`) and waits to be given access (waits until `critic==id`). This will eventually be granted in the post-protocol of an exiting critic. What happens is, the critic scans forward from its own `id` index in the `contender` array, and the first contender it finds is granted access. If no contenders are found, the lock is set free.

While waiting in the pre-protocol, contenders call `FROBNITZ` to gain the lock if it is free. The call is equivalent to the following atomic chunk:

```

if (critic == -1) { critic = id; return 0 }
else { critic_test = critic; return 1 }

```

We are not interested in the return values or the else case. If `critic` was free (`-1`) it will now be `id`, and the loop will terminate. Why is this in the busy-waiting loop? Intuitively processes should take the lock in absence of contention and bypass the loop entirely (the loop body would then be empty). However, the following interleaved execution sequence would cause problems:

contender	critic
	check for contention—none
raise contention flag	
FROBNITZ—lock not free	
wait to be given access	
	free lock

In this case, the contender would remain in the pre-protocol until another process entered and exited the critical region—but this might never happen, so the

fairness property does not hold. To avoid this we have the free-lock-acquisition (FROBNITZ instruction) in the loop, although it should only ever be successful on the first iteration or in a race condition as above; the lock is never freed between queued contenders. A contender may be preempted in acquiring the free lock by another process, but this cannot happen repeatedly: when that process exits it will grant access (perhaps through a sequence of others) to the original contender.

This solution is fair in the sense of linear waiting: a contender will be allowed into the critical region after one access by at most n other processes.

1.2.2 First In, First Out

This is a variation of the previous algorithm, attempting a higher degree of fairness.

```
shared INTEGER[0..n-1] fifo = {-1, ..., -1}
shared INTEGER fifo_head
shared INTEGER critic = -1
local INTEGER critic_test
local INTEGER my_index

pre-protocol:
    critic_test = -1
    my_index = fifo_head
    while (FROBNITZ(critic_test, id, fifo[my_index])==1) {
        critic_test = -1
        my_index = (my_index + 1) modulo n
    }
    while (critic != id) {
        critic_test = -1
        FROBNITZ(critic_test, id, critic)
    }
    fifo_head = (fifo_head + 1) modulo n

post-protocol:
    fifo[my_index] = -1
    critic = fifo[fifo_head]
```

The first loop in the pre-protocol attempts to secure a position in the FIFO queue. Entries are tested-and-set (conditionally on that slot being empty (-1)) starting at the `fifo_head` until the process id is stored at the end of the queue. The next loop is the same as in the previous version, it waits to be granted access, or gives itself the lock if available. Once the contender has access, it increments the FIFO head. In the post-protocol, the only things to do are clearing the

place we occupied in the queue, and granting the lock to the next queued process (which is either a process id or -1 if the queue is empty).

There is a concurrency problem with this algorithm, as illustrated in the following example.

```

    The lock is free; i.e. critic == -1.
process A           process B
queues self at head
                                queues self at (head + 1)
                                secures lock
                                increments head

```

The FIFO state around the head is now as follows:

-1	A	B	-1
----	---	---	----

 When B executes the post-protocol, if the state has not changed, the lock is freed, so that A can claim it. It is possible that processes might keep entering the spin loop and pre-empting A for the lock, but each one must first queue itself and move the head along—it will reach A after worst-case n other processes. Alternatively, more processes might queue themselves before B exits, so the lock is not freed but passed on. In this case, too, the head must move on, so eventually A will get the lock.

Another potential problem arises if `my_index` is set to `fifo_head`, but `fifo_head` has been incremented by the time the contender tries to queue itself at `my_index`. The previous `fifo_head` isn't cleared until the post-protocol, so normally the incorrect index would be tested and found occupied, then incremented harmlessly to the correct `fifo_head`. A problem only occurs if the entire critical region and post-protocol is executed before a single statement of the contender. Then the contender would find the index empty and queue itself at the wrong end. This is the same situation as above.

This algorithm provides FIFO fairness in normal operation, but reverts to worst-case linear waiting in the race condition outlined above.

1.3 A Real Instruction

The closest instruction I could find to FROBNITZ is `cmpxchg` on Intel 486 and Pentium. However, one of the variables is actually the accumulator, and the return values are swapped.

Sources:

- Li, Kai. (1999) "Preemptive Scheduling and Mutual Exclusion with Hardware Support". Princeton University. http://www.cs.princeton.edu/courses/archive/fall99/cs318/Notes/preemptive_files/v3_slide0016.htm

- Orlarey, Yann. (2000) “Lock-free LIFO used in MidiShare”. [linux-audio-dev]. <http://eca.cx/lad/2000/Jul/0319.html> (3 screens down)

2 The Sightseeing Problem

2.1 Single Coach

There are n tourist processes, and one coach process to hold $C < n$ tourists.

```
SEMAPHORE empty_seats => initialize(C)
SEMAPHORE boarding   => initialize(0)
```

tourists:

```
VOID coach_tour() {
    empty_seats.wait()
    boarding.signal()
    get_on_coach()
    empty_seats.signal()
}
```

coach:

```
while (True) {
    repeat C times {
        boarding.wait()
    }
    tour()
}
```

Only when there are any empty seats is a potential tourist allowed on the coach. It then signals this on the `boarding` semaphore. The coach is suspended on each iteration of its loop until the next tourist boards (of course, several tourists may board before the coach calls `wait`—in which case the coach skips suspension that many times). The loop terminates when exactly C tourists have boarded; the coach is then full and does its `tour()`. The boarded tourists have called `get_on_coach()` which only returns when the tour is over. They then disembar and signal `empty_seats`, allowing more tourists on board. The entire process can be repeated indefinitely.

2.2 Multiple Coaches

There are m coaches each capable of holding $C < n$ tourists. One coach is loaded at a time, and they cannot overtake each other. It is assumed that each coach knows its unique number in the range $[0, m - 1]$.

```

SEMAPHORE      empty_seats => initialize(C)
SEMAPHORE[0..m-1] boarding  => initialize(0)
INTEGER current = 0

```

tourists:

```

VOID coach_tour() {
    empty_seats.wait()
    boarding[current].signal()
    get_on_coach(current)
}

```

coaches:

```

while (True) {
    repeat C times {
        boarding[id].wait()
    }
    current = (current + 1) modulo m
    repeat C times {
        empty_seats.signal()
    }
    tour()
}

```

The index `current` identifies the coach that is currently boarding. Each coach has its own semaphore for boarding (it is signalled whenever a tourist enters that coach). However, only one semaphore is needed to handle the availability of seats on the current coach. Unless a tourist is blocked due to lack of seats, it signals the boarding semaphore for the current coach and calls `get_on_coach(id)`—assumed to handle the coach identifier—which blocks until the tour is over. Unlike the previous algorithm, the tourists don't free their own seats. Each coach is suspended until it has received C boarding signals. It then must be full. The current index is incremented before signalling C empty seats. All tourists thus activated will board the next coach.

3 A Monitor Problem

n consumers and one producer share a bounded buffer having b slots. Every consumer must read each item. This monitor is implemented using the *Signal and Continue* signalling discipline. Each consumer passes its unique `id` in the range $[0, n - 1]$ to the read function.

```

MONITOR broadcast_buffer

```

```

CONDITION notfull, moredata
INTEGER[0..b-1] buffer
INTEGER[0..b-1] reads = {0, ..., 0}
INTEGER[0..n-1] read_head = {0, ..., 0}
INTEGER head = 0, count = 0

VOID write(INTEGER x) {
    if (count == b) {
        notfull.wait()
        notfull.signal()
    }
    buffer[head] = x
    head = (head + 1) modulo b
    ++count
    moredata.signal()
}

INTEGER read(INTEGER id) {
    if (read_head[id] == head AND count != b) {
        moredata.wait()
        moredata.signal()
    }
    Result = buffer[read_head[id]]
    if (++reads[read_head[id]] == n) {
        /* everyone's read this item */
        --count
        reads[read_head[id]] = 0
        notfull.signal()
    }
    read_head[id] = (read_head[id] + 1) modulo b
    if (read_head[id] == head AND count == b) {
        notfull.wait()
        notfull.signal()
    }
}
}

```

END

The head variable is the position of the next write to the buffer. Each process stores the position of its own 'read head' where the next read will occur. The reads array keeps track of how many processes have read each entry in the buffer. Count is obviously the number of elements in the buffer.

The Signal and Continue discipline allows a sort of signal propagation to all interested parties. For example, if a consumer tries to read beyond the limit of currently available data it waits on the `moredata` condition. Several processes may be in this state. When one is woken up by a signal from the producer, it resends the signal, so that all waiting processes are activated (sequentially) on the one signal.

Usually, if the read head has the same index as the write head, then a consumer has hit the limit of available data and should wait. However, if the buffer is full the write head wraps around to the index of the oldest item—it may be that the consumer hasn't yet read the oldest item; in that case if it were to wait the system would deadlock. To avoid this we catch consumers at the end of the read function if they are at the end of a full buffer, and they are released when the buffer is no longer full. Now we can be sure that consumers trying to read at the write head in a full buffer are slow (not fast) and should be allowed. The same signal propagation technique is used as above.

4 A Message-Passing Problem

n processes are interconnected. Each has a unique id in the range $[0, n - 1]$. Each has an array `linked` such that if it is connected to process j , `linked[j] = 1`, and 0 otherwise. The variable `pair` should contain the index of each process's pair—or itself if it's single—at the end. No two adjacent processes should be single at the end. Each process executes the following algorithm:

```

INTEGER[0..n-1] singles
INTEGER pair
INTEGER j
MESSAGE msg
unique constant INTEGER { REQ, ACK, PAIRED }

singles = linked.clone()

for (j = 0..id-1) {
    while (singles[j] == 1) {
        msg = receive()
        if (msg.value == REQ) {
            pair = msg.source
            send(pair, ACK)
            send_broadcast(PAIRED)
            exit
        } else if (msg.value == PAIRED) {
            singles[msg.source] = 0
        }
    }
}

```

```

    }
  }
}

/* no more lower links */

for (j = id+1..n-1) {
  if (singles[j] == 0) { next }
  send(j, REQ)
  while (singles[j] == 1) {
    msg = receive()
    if (msg.value == ACK) {
      pair = msg.source
      send_broadcast(PAIRED)
      exit
    } else if (msg.value == PAIRED) {
      singles[msg.source] = 0
    }
  }
}
}
pair = id

```

There are three types of messages in this system: REQ is for requests to form a pair; ACK is for the acceptance of a pairing request; PAIRED is to signal a formed pair to others.

Processes only send requests to those with a higher id, but first they must account for all process with a lower id. Any one of the lower processes might send a pair request—the first one we receive, we accept, set `pair` accordingly and broadcast the fact to all neighbours. In fact, we only need to send the PAIRED message to the remaining single neighbours, but it's harmless for the others. A paired process is done, and exits. If we receive a PAIRED message, we update the singles array so that process is treated as if it isn't linked to us. The first loop only terminates when all neighbours with lower ids have paired up.

If all of our neighbours with lower ids have paired up (to other processes, not us)—or there were no neighbours with lower ids—the next loop comes into play. For each process with a higher id, we send it a pairing request. If we receive an acknowledgement, we take that process as our pair, broadcast the fact, and exit. Otherwise it is the same as the previous loop.

If all our neighbours have paired up, but not with us, both loops terminate and we set `pair` to our own id to indicate our single status.

5 Reflection

5.1 Time

My time taken was very roughly as follows:

1(a)	1 hour
1(b)	4 hour
1(c)	3.5 hours
2(a)	1 hour
2(b)	1 hour
3-paper	1 hour
3-design	2 hours
4	2 hours
writeup	7 hours
total	22 hours

The only research I did was reading Hoare's paper on Monitors (1 hour), looking up the definitions of fairness in the brick, and the web research question 1(c). Everything else was design and experiment (and writeup), although obviously making use of material learned from the course.

5.2 Easiness

I found questions 1(a) and 2(a) easy because I understand the concepts of locking and semaphores, and those problems were typical examples.

5.3 Difficulties

I found 1(b) difficult because of the mind-bending complexity of concurrent programming. I found the wrapping-head problem of question 3 surprisingly challenging. In question 4, I had to do some experimentation after a false start before coming up with a good algorithm. This was because I hadn't designed a distributed algorithm before. Question 1(c) was just @\$* frustrating.

5.4 Learning

I better appreciate concurrent programming in general. I'm aware of a larger range of atomic CPU instructions. I better understand the design patterns and function of monitors (and semaphores). I have had a taste of the coolness of distributed algorithms.