Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2018 Prof. R. Wattenhofer

Computer Engineering II

Solution to Exercise Sheet Chapter 9

Quiz

1 Quiz

a) False. Internally `counter++` is translated into at least 3 operations: a load operation from memory into a register, an addition on the register and a store from the register into memory. The scheduler may preempt the process at any point during the execution meaning that the 3 operations may be interleaved with operations from other processes, which may cause conflicts.

b) True. They appear seemingly at random because they depend on scheduling decisions. Two seemingly identical runs may not produce the same result.

c) True – even if you write the code correctly, the CPU could reorder memory accesses. In Exercise 2 you have to construct a schedule that shows how the algorithm can fail if two steps that look exchangable to such CPU optimization algorithms are swapped.

Advanced

2 Bad Peterson

Take two processes P (with $P.i = 0$) and Q (with $Q.i = 1$) that will both execute Algorithm 1 Line 1 next. We could get this schedule:

(1)	P :	$\Pi := 1 - 0$	// $\Pi = 1$
(2)	Q :	$\Pi := 1 - 1$	// $\Pi = 0$
(3)	Q :	$W_1 := 1$	
(4)	Q :	**repeat until** $\Pi = 1$ or $W_0 = 0$	// succeeds because $W_0 = 0$
(5)	Q :	enters critical section in Line 4	
(6)	P :	$W_0 := 1$	
(7)	P :	**repeat until** $\Pi = 0$ or $W_1 = 0$	// succeeds because $\Pi = 0$
(8)	P :	enters critical section in Line 4	
(9)	P :	$W_0 := 0$	// exits critical section
(10)	Q :	$W_1 := 0$	// exits critical section

In Line 8 of the schedule, both processes are in their critical section simultaneously.

3 Semaphores – Blocking or Spinning?

a) A spinning semaphore is more efficient than a blocking one if a critical section takes less time than the extra context switches required in the case of a blocking semaphore. Consider the following:

Take two processes P and Q, both calling wait() on a binary semaphore, and assume P wins the race.

With a spinning semaphore, Q will remain scheduled and do busy waiting until P is done.

With a blocking semaphore, Q will be blocked, descheduled (1st context switch from Q to some ready process), then P will unblock Q at some point, and Q will eventually be scheduled to resume execution where it was blocked (2nd context switch to restore Q to execution).

b) Algorithm 6 uses a blocking semaphore. The busy waiting done in a blocking semaphore is with regard to the critical section of the semaphore. If you consider the implementations of wait() (Algorithm 4 of the exercise sheet) and signal() (Algorithm 5), you will see each critical section of the semaphore is very short, which means the busy waiting will waste few CPU cycles. The processes using the semaphore will only waste those few CPU cycles, and otherwise will only occupy a core if they are in their critical section.

Algorithm 7 uses busy waiting for the critical section of the processes that have to be synchronized. This means that if we have two processes P and Q, then Q could be doing busy waiting for the whole duration of the critical section of P, which will often be orders of magnitude longer than the critical sections of our blocking semaphore.

We get the same busy wait overhead with a spinning semaphore as with Algorithm 7.

4 Lost Wakeup Problem

a) Assume we switch Algorithm 4 Lines 4 and 5. Once a process P that wait()s on the semaphore is blocked, the scheduler will suspend its execution. This means P will not unlock R before it is preempted, and it will not be scheduled again until it is unblocked. Other processes would have to lock R when they signal() the semaphore (see Algorithm 5 Line 1), which they cannot while P holds the lock on R, instead they will be busy waiting until R is unlocked. Therefore no other process will unblock P in Algorithm 5 Line 5. Thus, P will never unlock R, and all processes using the semaphore will deadlock.

b) The scheduling problem that can happen is that a process could be unblocked before it is blocked, which means the unblock would be lost. Consider this schedule with two processes P and Q, where P runs Algorithm 4 from Line 3 and Q runs Algorithm 5 from Line 1:

```
(1) P : L.addLast(P);                          // Algorithm 4 Line 3
(2) P : unlock(R);
(3) Q : lock(R);                               // Algorithm 5 Line 1
(4) Q : S++;
(5) Q : if L is not empty then                 // succeeds because P is in L
(6) Q : P = L.removeFirst()
(7) Q : P.unblock()
(8) P : P.block()                              // Algorithm 4 Line 5
```

The problem is that unlock(R) and P.block() in Algorithm 4 do not happen atomically, thus P can be unblocked by another process before it blocks itself. There are a few ways to solve this problem:

- We can add a flag to the state information of each process that remembers whether the process received an unblock(); in parts of the literature, this possibility is referred to as adding a “wakeup waiting bit”. When a ready or running process is unblock()ed, the flag is set to 1. Whenever a process blocks, it first checks whether the flag is 1 or 0. If it is 1, then instead of setting its state to blocked, the process sets the flag back to 0 and goes to the ready state. Only if the flag is 0 will a process go to the blocked state on block(). UNIX does something very similar to this idea.

- Since the lost wakeup problem is a race condition and depends on scheduling decisions, another common approach is to tackle it in the scheduler. One solution here is to offer a low-level operation that atomically can unlock a register and set a process state to blocked, informing the scheduler that a process cannot be preempted while executing that operation.
- In general, how the lost wakeup problem is solved depends on the specific scheduler and therefore the OS, and different systems solve it differently.

5 readers-Writers Problem (or Second Readers-Writers Problem)

We solve the problem in a similar fashion to the Readers-writers Problem in the script.

The idea is this: we divide the timeline into intervals where no writer either wants to write or is currently writing (this is when readers can be active), and the rest. We call the intervals where readers can be active the *reading intervals*, and the rest of the timeline are *writing intervals*. Every new reader that wants to start reading has to check whether we are currently in a reading interval by locking a mutex **accessIntervalMutex**. The first writer after a reading interval does the same. If the mutex is held by a reader, we are in a reading interval, and if it is held by a writer, then we are in a writing interval and no reader can be allowed to start reading. Since only the first writer in a writing interval has to lock the mutex, the writing interval continues until no more writers are active.

To make sure that all readers start waiting as soon as a single writer has requested access, only one reader at a time can be allowed to try to start reading, which we control with another mutex **readRequestMutex**.

Altogether, we need the following mutexes:

- **readRequestMutex** Guarantees that at most one reader at a time can try to start reading to make sure that a writer can get access as soon as possible after he requests it.
- **accessIntervalMutex** If a writer locked the mutex last, then no reader will be able to start reading before all writers are finished. Only the first writer in a non-reader-interval locks the mutex, and the last one unlocks it.
- **accessMutex** As in the solution to the Readers-writers Problem, this controls whether at most one writer or arbitrarily many readers are currently accessing the data.
- **readCountMutex** Synchronizes access to a variable **readCount** that counts how many processes are currently reading.
- **writeCountMutex** Synchronizes access to a variable **writeCount** that counts how many processes have currently requested write access.

Algorithms 8 and 9 solve the problem.

Algorithm 8 readers-Writers problem: Reader

Input: process ID i

Shared data structures: semaphores `readCountMutex`, `writeCountMutex`, `readRequestMutex`, `accessIntervalMutex`, `accessMutex`, all initially 1; integer `readCount` initially 0

```
while true do
    readRequestMutex.wait($i$);
        accessIntervalMutex.wait($i$);
            readCountMutex.wait($i$);
                readCount++;
                if readCount == 1 then    // first reader waits until writer finishes
                    accessMutex.wait($i$);
                end if
            readCountMutex.signal();
        accessIntervalMutex.signal();
    readRequestMutex.signal();
    read();
    readCountMutex.wait($i$);
        readCount−−;
        if readCount == 0 then    // last reader lets writers start writing
            accessMutex.signal();
        end if
    readCountMutex.signal();
end while
```

Algorithm 9 readers-Writers problem: Writer

Input: process ID i

Shared data structures: semaphores `readCountMutex`, `writeCountMutex`, `readRequestMutex`, `accessIntervalMutex`, `accessMutex`, all initially 1; integer `writeCount` initially 0

```
while true do
    writeCountMutex.wait($i$);
        writeCount++;
        if writeCount == 1 then    // first writer waits until last reader finishes
            accessIntervalMutex.wait($i$);
        end if
    writeCountMutex.signal();
    accessMutex.wait($i$);
        write();
    accessMutex.signal();
    writeCountMutex.wait($i$);
        writeCount−−;
        if writeCount == 0 then    // last writer opens up next potential reading interval
            accessIntervalMutex.signal();
        end if
    writeCountMutex.signal();
end while
```

6 Dentists' Office Problem

This problem is very similar to the well-known Multiple Sleeping Barbers Problem, where a few other details are different from our formulation. There are several things we have to model to solve the Dentists' Office Problem:

- Each dentist and each patient is a separate process.
- There is a limited number of chairs for patients.
- Each dentist serves at most one patient and each patient is served by at most one dentist at any given time, i.e. we have to match up dentists and patients.

Given this, the monitor will proceed as follows: it will have two methods **getPatient** and **getDentist** that correspond to what dentists and patients want to achieve, respectively. **getPatient** will have to give a sleeping dentist process a patient as soon as one wants to be treated, and **getDentist** signals that a patient wants to be treated which we do with a condition variable. We need a waiting queue for the waiting room and a variable to check the number of patients in the waiting room. Finally, we need a counter of available dentists so patients know whether they have to go sit in the waiting room first.

Putting it all together, our solution consists of Algorithms 10, 11, 12. The solution works for arbitrarily many dentists and patients.

Algorithm 10 Dentists' Office Monitor

```
Internals: semaphore monitorMutex initially 1;
    condition variable patientsAvailable, initially empty;
    list patientsQueue, initially empty;
    integers waitingRoomSize:=n, waitingPatients:=0, availableDentists:=0;
procedure getPatient(Process dentist){
 1: monitorMutex.wait(dentist);
 2: while patientsQueue is empty do
 3:         availableDentists++;
 4:     patientsAvailable.conditionWait(monitorMutex, dentist);
 5:     availableDentists--;
 6: end while
 7: if waitingPatients > 0 then
 8:     waitingPatients--;
 9: end if
10: Process assignedPatient = patientsQueue.removeFirst();
11: monitorMutex.signal();
12: return assignedPatient;
 }
procedure getTreated(Process patient){
13: monitorMutex.wait(dentist);
14: if availableDentists > 0 OR waitingPatients < waitingRoomSize) then
15:     patientsQueue.addLast(patient);
16:     if availableDentists == 0 then  // if no dentist is immediately
                                        available, patient has to wait until
                                        dentist becomes available
17:         waitingPatients++;
18:     end if
19:     patientsAvailable.conditionSignal();
20: end if
21: monitorMutex.signal();
 }
```

Algorithm 11 Dentist for Monitor from Algorithm 10

```
Internals: Dentists' Office Monitor $M$
Input: Process dentist
1: while true do
2:     Process patient = $M$.getPatient(dentist);
3:     dentist.treatPatient(patient);
4: end while
```

Algorithm 12 Patient for Monitor from Algorithm 10

```
Internals: Dentists' Office Monitor $M$
Input: Process patient
1: $M$.getTreated(patient);
```