Homework 2: Written Problems

Problem 1: Exercise 3.2, page 142

Suppose a computer has atomic decrement DEC and increment INC instructions that also return the value of the sign bit of the result. In particular, the decrement instruction has the following effect:

```
DEC(var, sign):
  < var = var - 1;
   if (var >= 0)
     sign = 0;
   else
     sign = 1; >
```

INC is similar, the only difference being that it adds 1 to var.

Using DEC and/or INC, develop a solution to the critical section problem for n processes. Do not worry about the eventual entry property. Describe clearly how your solution works and why it is correct.

```
mutex = 0, sign = 0
process CS [ i = 1 to n ] {
        while ( true ) {
                <await( sign == 0) DEC(mutex,sign);>
                critical section S;
                INC(mutex,sign)
                non-critical code
        }
}
```

How solution works: var/mutex starts at one, and when the first process reaches the first sign check, since the mutex is non-negative, the process will make it's way to the first atomic DEC action, taking the sign to 1. Because the dec action is atomic, all the other processes will be stuck at the await statement. Finally, after the critical section goes, var is incremented, sign is back to 0, and the next process can start.

Problem 2: Exercise 3.4, page 143

Suppose a computer has an atomic Compare-and-Swap instruction, which has the following effect:

```
CSW(a, b, c):
  < if ( a == c ) {
      c = b;
      return (0);
    }
    else {
      a = c;
      return (1);
    } >
```

Parameters **a**, **b**, and **c** are simple variables, such as integers. Using **CSW**, develop a solution to the critical section problem for **n** processes. Do not worry about the eventual entry property. Describe clearly how your solution works and why it is correct.

```
global = 0
process CS [ i = 1 to n ] {
        local = 0

        while (CSW(local, 1, global) == 1) {
                local
                continue;
        }
        critical section S;
        global = 0
        }
}
```

Global: c
Local: a
Constant: b = 1

Explanation: For the n processes that make their way to the instruction, the first processes that executes CSW ends up going through to CS but global is now = 1. During this time, any process entering while loop will have CSW = 1, changing the value of local to 1. However, we want to keep this lock spinning, so we reset local to 0, and thus will not be able to enter the critical section. This resolves with the CS exit code, which resets global to 0 and allows a process to exit while loop and enter CS again.

Problem 3: Exercise 4.11, page 194

It is possible to implement a reusable n-process barrier with two semaphores and one counter. These are declared as follows:

```
int count = 0;
sem arrive = 1, go = 0;
```

Develop a solution. (*Hint*: Use the idea of passing the baton.)

```
# n = the number of threads
P(arrive)              // wait here letting one thread n
count = count + 1      // increment count, one thread at a time
V(arrive)              // thread that incremented can move on

If (count == n) V(go)  // if all threads have incremented counter, allow the nth thread to pass
                       // through
P(go)                  // first n – 1 threads wait here. When V(go) is executed, a thread is
                       // allowed to pass through
V(go)                  // thread that passes through has to signal other threads to move on so they
                       // don't get stuck
```

Problem 4: Exercise 4.13, page 194

Consider the following proposal for implementing **await** statements using semaphores:

```
sem e = 1, d = 0;  # entry and delay semaphores
int nd = 0;        # delay counter
# implementation of <await (B)  S;>
P(e);
while (!B) {
        nd = nd + 1;
        V(e);
        P(d);
        P(e);
}
S;
while (nd > 0) {
        nd = nd - 1;
        V(d);
}
V(e);
```

For each of the following questions, either give a convincing argument why the answer is "yes", or give an execution sequence that illustrates why the answer is "no".

a.) Does this code ensure that the **await** statement is executed atomically?

Not necessarily, other elements can enter and be in the while loop, with a sample execution being that B is true, so multiple processes can execute S (due to the entry lock being lifted in the while loop) without finishing the function.

b.) Does it avoid deadlock?

Yes, because as soon as B turns, S is immediately executed. There's no requisite action before because P(e) is already executed by the time you get to the while loop. In addition, due to the first semaphore, other processes are blocked there or just delaying in the while loop.

c.) Does it guarantee that **B** is true before **S** is executed?

Yes, if B is not true, the program will spin in the while loop until B is true. The entry semaphore is already passed, but no processes will execute S until B is true.

Problem 5: Exercise 4.20, page 196
Explain the role of each semaphore. Develop assertions that indicate what is true at critical points. In particular, show that the solution ensures that writers have exclusive access to the database and a writer excludes readers. Give a convincing argument that the solution gives writers preference over readers.

sem m2: controls mutual exclusion to the number of writers counter. Next, provides mutual exclusion to decrementing the number of writers and passing baton to readers.
sem m1: first provides inner mutual exclusion to number of readers counter. Next, it provides mutual exclusion to decreasing number of readers and passing the baton to writers.
sem m3: provides mutual exclusion to read semaphore.
sem read: In reader process: controls when a process can enter in order to increment number of readers and disable write if the number of readers is one. If not, then the process holds on the second line so that nothing is executed. In writer process, if the number of writers is 1, access is closed to read, and released after there are no writers left. So, in effect**, this means that no readers can exist in the database while there is at least one writer currently running**.
sem write: In reader process, allows reader number to be incremented, and allows for number of readers to be decremented. **Importantly, it also allows for no writers to be in the database while there is at least one reader in there**. **In writer process, handles code so that only one writer can write at a time.**

Basically, if there is one or more readers or writers in the current process, access is closed to the other. Also multiple readers allowed to read at one time, but only one writer can write at a time. And if no more readers exist for the reader process, both read and write semaphores are open. Other mutex semaphores exist so these actions of swapping/incrementing/decrementing are atomic.

But what makes this code writer preference? Well, for starters, readers are delayed if there is one or more writers in line. And the delayed readers are only awakened if there are no writers available. Finally, the fact that an extra mutex surrounds the incrementing of readers means that writers can enter the queue more quickly, thus showing that this code favors writers.

Problem 6: Exercise 4.21, pages 197-8

Consider the following solution to the readers/writers problem. It employs the same counters and semaphores as in Figure 4.13 (page 176), but uses them differently.

a.) Carefully explain how this solution works. What is the role of each semaphore? Show that the solution ensures that writers have exclusive access to the database and a writer excludes readers.

Semaphore controls access to the baton switching code. R controls read access to the database (can only pass P(r) if number of readers has been incremented). W controls write access to the database (only opens if there is a writer available for the database). When a reader process comes alive, it takes the mutual exclusion semaphore, increments number of readers if number of writers is 0 (else delays reader), closes, me semaphore, and awaits to read the db. Once db has been read, it opens the mutual exclusion semaphore again, decrements number of readers, and if the number of readers is 0 and number of writers is more than 0, opens writer semaphore. When a writer process comes alive, it adds to number of writers and opens itself or delays, writers, and opens another writer or passes baton to reader processes.

b.) What kind of preference does the solution have? Readers preference? Writers preference? Alternating preference? Explain.

This code has an alternating preference.

1. It delays a new reader when writer is waiting (line 9 of reader)
2. It delays new writer when readers are waiting (line 9 of writer)
3. It awakens a writer when reader finishes (line 18 of reader)
4. It awakens all readers when writer finishes (while loop at end of writer).

According to the textbook, this is the definition of an alternating preference solution, so I believe this code has alternating preference.

c.) Compare this solution to the one in Figure 4.13 (p. 176). How many P and V operations are executed by each process in each solution in the best case? In the worst case?

Figure 4.13:

Best case for reader: 4 ops
Worst case for reader: 6 ops
Best case for writer: 4 ops
Worst case for writer: 6 ops

This figure:

Best case for reader: 5 ops
Worst case for reader: 7 ops
Best case for writer: 5 ops
Worst case for writer: >=7 ops (depending on how many readers are waiting)

Figure 4.13 is more efficient, but is reader favored.