CISC 324, Winter 2018, Assignment 3

This assignment is due Tuesday, February 13, in lecture

The first in-class exam is Thursday February 15 in Dunning 14 and Jeffery 126

Lab 4 is due Tuesday February 27 1, in lecture.

Readings for Assignment 3

In the course reader:

Pages 27-35 Readers and writers. Pages 27-33 were already given with assignment 2, here add pages 34-35.

Pages 35-42 Classic problems of synchronization. Hardware and software methods of implementing semaphores.

Pages 45-47 Description of a real-life example of a serious concurrency problem. Skim/read this but I will not ask exam questions about this particular incident. (For now, skip sections 6, 7, 8 on pages 42-44: we cover monitors and message passing later.)

Pages 48-49 Deadlock: Banker's algorithm

In the textbook:

Section 5.7 (6.6 in 8th edition) Classic Problems of Synchronization

Section 5.2-5.4 (6.2-6.4 in 8th edition) Software and hardware solutions to the critical section problem. The OS can use these solutions to implement semaphores: the OS must implement Acquire(sem) and Release(sem) as critical sections so that these semaphore operations work properly when several processes try to acquire or release the same semaphore at the same time.

Chapter 7 Deadlock. Resource allocation graphs. Banker's algorithm.

Synchronization Questions

(1) Consider the readers and writers problem where readers have priority. This code was discussed in lecture and a Java version is supplied to you in lab 3. I have added line numbering.

```
   shared variables                    reader process
1. semaphore mutex = 1;              7. acquire(mutex);
2. semaphore wrt = 1;                8. readcount = readcount + 1;
3. int readcount = 0;                9. if (readcount == 1) then acquire(wrt);
                                    10. release(mutex);
writer process                      11. --- reading is performed ---
4. acquire(wrt);                    12. acquire(mutex);
5. --- writing is performed ---     13. readcount = readcount - 1;
6. release(wrt);                    14. if (readcount == 0) then release(wrt);
                                    15. release(mutex);
```

Parts (a) (b) and (c) each propose a change to make to this code. In each case, indicate whether the change can result in any of the following three problems: (i) deadlock, (ii) illegal access, so a writer is active at the same time as another writer or reader, or (iii) a loss of parallelism, so readers are prevented from being active simultaneously. If there is a problem, **give one specific example of how the problem could occur.** Use a level of detail comparable to the following: "reader1 is active (at line 11) and writer1 arrives (waits at line 4) and then reader1 finishes (lines 12 to 15), making readcount equal to zero". You may find it easiest to create a table in which each row represents some point in time, and time advances as you go down the rows of the table. You can use one column for each process to show the line number of the statement that this process is currently executing. Use other columns to show the current value of readcount (integer) as well as the values of mutex and wrt (each semaphore value consists of an integer and a queue of PCBs).

(a) Swap lines 13 and 14, testing "readcount == 1":

```
new13. if (readcount == 1) then release(wrt);
new14. readcount = readcount - 1;
```

(b) Swap lines 14 and 15, so finishing reader executes:

```
12.    acquire(mutex);
13.    readcount = readcount - 1;
new14. release(mutex);
new15. if (readcount == 0) then release(wrt);
```

(c) Change line 9. so that a reader releases mutex while waiting for the wrt semaphore:

```
new9. if (readcount == 1) then begin release(mutex);  acquire(wrt);  acquire(mutex) end
```

(2) Page 34 of the course reader shows semaphore code (for starvation free readers&writers) that can fail because there is delayed counter update. Page 35 illustrates with an example a synchronization error that can arise when a reader executes EndRead. In this assignment question you analyze a different error-producing scenario: a synchronization error that arises when a writer executes EndWrite. The first 5 events happen in the order shownin the following list. Your assignment is to write out the list from item 6 onwards.

1. Writer W1 is active (has finished executing StartWrite). Now readersActive=0, writersActive=1, readersWaiting=0, writersWaiting=0.
2. Readers R1, R2, R3 show up: each of them gets partway through executing StartRead and waits at Acquire(readerSem). Now readersActive=0, writersActive=1, readersWaiting=3, writersWaiting=0.
3. W1 starts executing EndWrite, and is holding the mutex semaphore while doing so.
4. W2 shows up, and has to wait for mutex in the first line of code in StartWrite.
5. When W1 is done with EndWrite, it has executed "Release(readerSem)" three times and left the counter values as follows: readersActive=0, writersActive=0, readersWaiting=3, writersWaiting=0. The last thing W1 does in EndWrite is to release mutex.

[**Continue this list**: describe what events happen next and demonstrate how this leads to an error.]

(3) Algorithm 3 for the dining philosopher's problem, on page 38 of the course reader, allows at most four philosophers to enter the room and compete for chopsticks. Explain why deadlock cannot occur when this algorithm is used.

(4) Peterson's algorithm (textbook Figure 5.2) is a correct solution to the critical section problem for two processes. For each change (a) and (b) below, state whether the altered code still satisfies the three requirements of the critical section problem. (Refer to textbook section 5.2 for a statement of the three requirements). If the code can fail, give a specific example of how the failure can arise.

> Reminder: The kernel of the operating system needs to solve the critical section problem in order to implement the semaphore operations *acquire* and *release*. All other processes can leverage off of that, using acquire(mutex) and release(mutex) around their critical sections.

(a) Swap the first two lines of code in Peterson's algorithm. Here is the new code:

```
          Code for process P0                          Code for process P1
repeat                                        repeat
   ...                                           ...
   turn := 1;                                    turn := 0;
   flag[0] := true;                              flag[1] := true;
   while (flag[1] and turn = 1) do { nothing }   while (flag[0] and turn = 0) do { nothing }
   ...Critical Section...                        ...Critical Section...
   flag[0] := false                              flag[1] := false
   ...                                           ...
forever                                       forever
```

(b) Change "and" to "or" in the while-loop condition. Here is the new code:

```
          Code for process P0                          Code for process P1
repeat                                        repeat
   ...                                           ...
   flag[0] := true;                              flag[1] := true;
   turn := 1;                                    turn := 0;
    while (flag[1] or turn = 1) do { nothing }   while (flag[0] or turn = 0) do { nothing };
   ...Critical Section...                        ...Critical Section...
   flag[0] := false                              flag[1] := false
   ...                                           ...
forever                                       forever
```

(5) Hardware solutions to the critical section problem use read-modify-write instructions: these instructions are indivisible because the CPU gets to hold onto the memory bus for two cycles, to both read and then update the shared memory location. As an example, here is code for implementing mutual exclusion using a Swap () machine-code instruction; that instruction has the opcode EXCH in the Pentium instruction set http://x86.renejeschke.de/html/file_module_x86_id_328.html. [The following code is from Figure 6.7 in the 8th edition of the textbook. The closest thing in the 9th edition is compare_and_swap() in Figure 5.6,. I'm sticking with Swap() because that matches the EXCH instruction provided by Pentium.]

```
do {
      // Here is the entry code: wait for permission to enter the critical section.
      // "Lock" is an integer shared by all processes; "key" is a local variable.
      key = TRUE;
      while (key == TRUE)
            Swap(&lock, &key);

      // Critical section code goes here: access the shared data structure. ...

      lock = FALSE   // this is the exit code: we have left the critical section

      // other code (not critical section) goes here...
} while(TRUE);
```

Describe what happens if three processes try to enter their critical section at about the same time. How does the given code ensure that only one process gets entry? Hint: Pay attention to the fact that **the *lock* variable is shared by all processes, whereas *key* is a local variable**. The OS implements this in assembly language, getting the global/local effect by storing *lock* in a shared location in main memory and storing *key* in a general purpose register such as AX or BX.

Deadlock Questions

(6) Describe a method for avoiding deadlock in a system that deals with electronic transfer of funds. In this system, there are hundreds of identical processes that do the following.

1. Read an input line M, x, y, where M is an amount of money to be transferred, x is the account to transfer from, and y is the account to transfer to.
2. Lock accounts x and y. Transfer the money. Release the locks.

Deadlock can occur in this system. For example, suppose that process A wants to lock accounts u and v, and process B wants to lock accounts v and u. If it happens that process A locks u and process B locks v then there is deadlock, with process A waiting forever trying to lock v (which is held by B) and process B waiting forever trying to lock u (which is held by A).
Your job is to come up with a method for avoiding deadlocks. Your method should NOT force a process to release and relock an account record: once a process locks a record, that process keeps the lock until the money transfer has been carried out. (Hint: look at the deadlock prevention schemes described in Section 7.4 of the textbook.)

(7) The OS is applying the Banker's algorithm to a system with four resource types and five processes. (Textbook Section 7.5.3.3 provides an example.) The current state of the system is shown in the following table. As a first step, fill in the values for "Need".

Process	Max	Allocation	Need
P_1	1 3 0 1	0 2 0 1	
P_2	0 0 1 1	0 0 0 1	
P_3	2 0 1 0	1 0 0 0	
P_4	2 0 1 0	0 0 1 0	
P_5	0 2 0 3	0 1 0 1	

Available
2 1 0 0

Total Resources
3 4 1 3

In this system state, process P5 now issues a request for one unit of resource type 2; this is represented as request vector 0 1 0 0. Can the OS grant this request without the possibility of future deadlock? Justify your answer: if there is risk of deadlock, state which processes are at risk of being involved in deadlock. If there is no risk of deadlock, say why.

(8) This problem illustrates what happens when processes overestimate their need for resources on a computer where the operating system uses the Banker's algorithm to prevent deadlock. To keep this example simple, we use only one type of resource (tape drive). As you will see, the Banker's algorithm sometimes makes processes wait for tape drives even though many unused tape drives are available. This underutilization of resources is justified in cases where it is very important to prevent deadlock.

Consider a system that has 10 tape drives. The system is executing a set of 20 processes, each of which overestimates its need for tape drives:

(a) Each of the 20 processes declares Max = 2 tape drives, but each process ends up using only one tape drive during execution. What is the maximum number of tape drives that can be in use at any one time, given this job set (and given that the Banker's algorithm is being used)? Justify your answer. Hint: "5" is not the right answer.

(b) Each of the 20 processes declares Max = 10 tape drives, but each process ends up using only one tape drive. What is the maximum number of tape drives that can be in use at any one time, given this job set? Justify your answer.

(c) Generalize the answers from (a) and (b): Each of the 20 processes declares Max = N tape drives, where $1 \leq N \leq 10$; all 20 processes use the same value of N. Each process ends up using only one tape drive during execution. What is the maximum number of tape drives that can be in use at any one time, given this job set? Give an expression involving N.