Assignment 3 is due at the last lecture before reading week. If you are away then, hand your assignment in at an earlier lecture, or bring it to the School of Computing Office (Goodwin 557) before the due date.

The midterm exam is in Dunning 14 on Thursday, Feb 26, 5:30-7:30PM. See the course web page for more information.

Lab 4 is important in preparing for the midterm exam. Plan carefully before you begin coding. This lab will probably take you two to five hours.

**Readings for Assignment 3**

In the course reader:

Pages 20-45 Overview of Process Synchronization. Part of this reading was already given with Assignment 2. For now, skip sections 5 and 6 (pages 40-42) on monitors and message passing; we cover these later in the course. Pay special attention to section 3 Classic Problems of Synchronization (page 28-37) and Section 4 Implementation of Semaphores (pages 37-40). Section 7 (page 43) describes a widespread power outage caused by incorrect access to shared data: read about this real-life example of serious disruption caused by a concurrency problem, but be assured that I will not ask exam questions about this particular incident.

Pages 46-48 Deadlock; Banker’s algorithm

In the textbook:

Section 5.7 (6.6 in earlier editions) Classic Problems of Synchronization

Section 5.2-5.4 (6.2-6.4 in earlier editions) 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 the semaphore counter and queue are tested and updated properly, even when several processes want to acquire or release the same semaphore at the same time.

Chapter 7 Deadlock

**Assignment 3 Questions**

1. Processes P and Q are executing concurrently, using a shared semaphore named *sem* that has the initial value 3.

    ```plaintext
    var sem: semaphore := 3;
    Process P
    loop
        acquire(sem);
        printf("P speaking");
    endloop
    Process Q
    loop
        printf("Q speaking");
        release(sem)
    endloop
    ```

    Pnum and Qnum are the number of times processes P and Q have printed their messages. Choose the correct answer, and write a brief justification.

    (a) Pnum = Qnum-3  (b) Pnum = Qnum+3  (c) Pnum ≤ Qnum-3  (d) Pnum ≤ Qnum+3
    (e) Pnum ≥ Qnum-3  (f) Pnum ≥ Qnum+3  (g) Pnum does not depend on Qnum  (h) none of the above
The following processes are executing concurrently. This code does not occur in a loop: each process $Pi$ executes its $work_i$ code exactly once.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;pre-code&gt;</td>
<td>&lt;pre-code&gt;</td>
<td>&lt;pre-code&gt;</td>
<td>&lt;pre-code&gt;</td>
<td>&lt;pre-code&gt;</td>
</tr>
<tr>
<td>$work_1$</td>
<td>$work_2$</td>
<td>$work_3$</td>
<td>$work_4$</td>
<td>$work_5$</td>
</tr>
<tr>
<td>&lt;post-code&gt;</td>
<td>&lt;post-code&gt;</td>
<td>&lt;post-code&gt;</td>
<td>&lt;post-code&gt;</td>
<td>&lt;post-code&gt;</td>
</tr>
</tbody>
</table>

(a) Use semaphores to enforce the following constraint: Process P1 can start executing $work_1$ code only when two or more of the other four processes have reached their <post-code>. State the initial value of any semaphore(s) you use. Add the necessary acquire and release calls to the <pre-code> and <post-code> sections of the processes.

(b) Use semaphores to enforce the following constraint: Process P1 can start executing $work_1$ code only when at least one of the following is true:
- P2 has finished $work_2$ and P3 has finished $work_3$
- P4 has finished $work_4$ and P5 has finished $work_5$

In your solution, the <post-code> for P2, P3, P4, P5 should not force these processes to wait for other processes to finish. You may use other shared variables (e.g. counters) in addition to semaphores, or you may create additional processes to help with the synchronization.

(3) 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
1. semaphore mutex = 1;
2. semaphore wrt = 1;
3. int readcount = 0;

writer process
4. acquire(wrt);
5. --- writing is performed ---
6. release(wrt);

reader process
7. acquire(mutex);
8. readcount = readcount + 1;
9. if (readcount == 1) then acquire(wrt);
10. release(mutex);
11. --- reading is performed ---
12. acquire(mutex);
13. readcount = readcount - 1;
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 one of the following three problems: (i) deadlock, (ii) illegal access (a writer is active at the same time as another writer or reader), or (iii) a loss of parallelism (readers are not allowed to be 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 draw a table in which each row represents some point in time, and time advances as you go down the rows of the table. Use some table columns to show the line numbers of the statements that processes are executing, and other columns to show the current values of mutex and wrt (integer value and a queue of PCBs), and the value of readcount.

(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 does:
```
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
```
(4) Refer to algorithm 3 for the dining philosopher's problem, on page 32 of the course reader. This algorithm allows at most four philosophers to be competing for the chopsticks. Explain why deadlock cannot occur when this algorithm is used.

(5) Page 36 of the course reader shows not-quite-correct semaphore code for the “starvation free readers&writers”. Page 37 illustrates a synchronization error that can occur when a reader executes EndRead. Here is a different error-producing scenario (a writer executing EndWrite): your job is to complete this list by writing out list items from item 6 onwards.

1. Writer W1 is active (has finished executing StartWrite). Now numberReading=0, numberWriting=1, wantToRead=0, wantToWrite=0.
2. Readers R1, R2, R3 show up: each of them gets partway through executing StartRead, and waits at Acquire(readerSem). Now numberReading=0, numberWriting=1, wantToRead=3, wantToWrite=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: numberReading=0, numberWriting=0, wantToRead=3, wantToWrite=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.]

(6) 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) for critical sections.

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

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

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

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

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

repeat Code for process P1
... flag[1] := true;
turn := 0;
while (flag[0] or turn = 0) do { nothing }
...Critical Section...
flag[1] := false
...forever
```
(7) Here is code for implementing mutual exclusion using the Swap() machine-code instruction. [This code is from Figure 6.7 in the 8th edition of the textbook. The closest thing in the 9th edition is Figure 5.6, using compare_and_swap(). We stick with Swap() because it matches the EXCH machine-code instruction in the Pentium instruction set http://x86.renejeschke.de/html/file_module_x86_id_328.html.]

```c
    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. In assembly language, this effect is achieved by storing lock in a shared location in main memory, and storing key in a general purpose register such as AX or BX.

(8) 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.)

(9) 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".

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1 3 0 1</td>
<td>0 2 0 1</td>
<td>2 1 0 0</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>0 0 1 1</td>
<td>0 0 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 0 1 0</td>
<td>1 0 0 0</td>
<td></td>
<td>Total Resources</td>
</tr>
<tr>
<td>P4</td>
<td>2 0 1 0</td>
<td>0 0 1 0</td>
<td>3 4 1 3</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>0 2 0 3</td>
<td>0 1 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.