

CISC 324, Winter 2012

Assignment 3, due Thursday, February 16, in lecture

Lab 4 due Monday, February 27, in lecture

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 Wednesday, Feb 29, 7:00-9:00PM in Dunning 14. The midterm will be based on assignments 1-3, and on labs 1-4. There will not be exam questions specific to the x86 (Pentium) assembly-language and machine-code.

Lab 4 is an important part of the preparation for taking the midterm exam. This lab will probably take you two to five hours; plan carefully before you begin coding Lab 4.

Readings for Assignment 3

In the course reader:

Pages 15-32 Overview of Process Synchronization. Part of this reading was already given with Assignment 2. For now, skip Section 3 (monitors) and Section 6 (message passing); we cover these later.

Pay attention to Section 4 (classic problems of synchronization) and Section 5 (implementation of semaphores).

Section 7 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. (Don't try to memorize this: I will not ask exam questions about this particular incident.)

Page 33-34 Deadlock; Banker's algorithm

In the textbook, read

Section 6.5 Semaphores

Section 6.6 Classic Problems of Synchronization

Section 6.2-6.4 Software and hardware solutions to the critical section problem. (This describes methods that can be used to implement semaphores: the sections of operating system code that implement "Acquire(sem)" and "Release(sem)" are critical sections.)

Chapter 7 Deadlock

Assignment 3 Questions

(1) Processes P and Q are executing concurrently, using a shared semaphore named *sem*.

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

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

- (a) $Pnum = Qnum - 3$ (b) $Pnum = Qnum + 3$ (c) $Pnum \leq Qnum - 3$ (d) $Pnum \leq Qnum + 3$
(e) $Pnum \geq Qnum - 3$ (f) $Pnum \geq Qnum + 3$ (g) Pnum does not depend on Qnum (h) none of the above

(2) The following processes are executing concurrently. This code does not occur in a loop: each process P_i executes its `worki_code` exactly once.

| <u>Process P1</u> | <u>Process P2</u> | <u>Process P3</u> | <u>Process P4</u> | <u>Process P5</u> |
|-------------------|-------------------|-------------------|-------------------|-------------------|
| <pre-code> | <pre-code> | <pre-code> | <pre-code> | <pre-code> |
| work1_code | work2_code | work3_code | work4_code | work5_code |
| <post-code> | <post-code> | <post-code> | <post-code> | <post-code> |

(a) Use semaphores to enforce the following constraint: Process P1 can start executing the `work1_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 the work1_code only when **at least one** of the following is true:

- P2 has finished its work2_code and P3 has finished its work3_code
- or - P4 has finished its work4_code and P5 has finished its work5_code

In your solution, the <post-code> for P2, P3, P4, P5 should not involve waiting for other processes to finish. You may use other shared variables (in addition to semaphores), or create additional processes to help with 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.

| <u>shared variables</u> | <u>reader process</u> |
|---------------------------------|--|
| 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); |
| <u>writer process</u> | 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 code can result in one of the following three problems: (i) deadlock, (ii) illegal access (a writer is active at the same time that another writer or reader is active), 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 to the next row 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, wrt, and readcount.

(a) Swap statements 13 and 14, testing “readcount == 1”:

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

(b) Swap statements 14 and 15, so a finishing reader says:

```
acquire(mutex);
readcount = readcount - 1;
release(mutex);
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 21 of the course notes. This algorithm allows at most four philosophers to be competing for the chopsticks. Explain why deadlock cannot occur when this algorithm is used.

(5) Peterson's algorithm (textbook Figure 6.2) is a correct solution to the critical section problem for two processes. For the changes indicated in (a) and (b), state whether the altered code still satisfies the three requirements of the critical section problem (stated in textbook section 6.2). If the code can fail, give a specific example of how the failure can arise.

Reminder: The operating system needs to solve the critical section problem in order to implement the semaphore operations *acquire* and *release*. Once that is done, everyone else can use 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:

| <u>Code for process P0</u> | <u>Code for process P1</u> |
|---|---|
| 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 |
|--|---|
| <pre>repeat ... flag[0] := true; turn := 1; while (flag[1] or turn = 1) do { nothing } ...Critical Section... flag[0] := false ... forever</pre> | <pre>repeat ... flag[1] := true; turn := 0; while (flag[0] or turn = 0) do { nothing }; ...Critical Section... flag[1] := false ... forever</pre> |

(6) Figure 6.7 of the textbook shows a way to implement mutual exclusion using the Swap machine-code instruction. (Note that the *lock* variable is shared by all processes, whereas *key* is a local variable. This is implemented by having each process keep its own *key* value in a CPU register, whereas *lock* is stored in a shared location in main memory.) 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?

(7) 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, whereas 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 to lock v (which is held by B) and process B waiting to lock u (which is held by A).

Your job is to come up with a method for avoiding deadlocks. Note that a process is not allowed 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.)

(8) We are 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 as shown in the following table. You can start solving this problem by filling in the values for "Need".

| Process | Max | Allocation | Need | Available |
|----------------|---------|------------|------|------------------------|
| P ₁ | 1 3 0 1 | 0 2 0 1 | | 2 1 0 0 |
| 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 | | |
| | | | | <u>Total Resources</u> |
| | | | | 3 4 1 3 |

In this system state, process P₅ now issues a request for one unit of resource type 2 (represented as 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.