

Homework 6

*Reading: Tannenbaum, Chapter 3**Due: Wed, Apr 4, 2007***Problem 1** (0 Points)

Consider the algorithm shown below and assume that we have created N processes numbered from 0 to $N - 1$ running on N processors. Furthermore, N is an odd integer.

```

Shared Variables:
    Semaphore      X[N] = 0; // array of N semaphores initialized to 0
    Semaphore      Y = 1;
    int            a = 0;
Process i:        // 0 <= i <= N-1
    int           b;
    if ((i Mod 2) == 0) { // beginning of body
        ... Compute b ...
        Wait (Y);
        a = a + b;
        Signal (Y);
    } else {
        Wait(X[i-1]);
        Wait(X[Min(i+2,N-1)]);
        ... Compute b ...
        Wait (Y);
        a = a + b;
        Signal (Y);
    }
    Signal (X[i]);          // end of body

```

- Consider the case of $N = 7$ and where each process is running uninterrupted (except for synchronization) on its own processor. In what order will each process compute their local values of b ? That is, give the sequence of process numbers. Explain.
- What is the purpose of each of the semaphores $X[i]$, and Y ?
- Suppose that we put the body of the above process code in a `do forever ...` loop. Show how you can insure that each process does not start loop iteration $i + 1$ until all processes have completed iteration i .

Problem 2 (6 Points)

Barrier synchronization between N processes works as follows:

- A counter is initialized to N , the number of processes participating in the barrier.
- The first $N - 1$ processes arriving to a barrier should wait.
- The N th process arriving to a barrier should unblock the $N - 1$ waiting processes so that all N processes can continue with the rest of the program.

The last page of this assignment defines the assembler instruction set of a machine which includes the *atomic swap* instruction. The atomic swap instruction denoted by $R1 \leftrightarrow X$ exchanges the contents of register $R1$ with the word at the memory location named X without allowing any intervening interrupts or deferred traps.

- a) Suppose that your machine has enough processors so that each process can execute on its own CPU. Give a barrier synchronization algorithm that uses the **atomic swap** hardware instruction and busy waiting. Assume that there are no synchronization functions except the **atomic swap** instruction. Note that the algorithm needs to perform a barrier synchronization between N processes only once. Also, assume that there are no other synchronization primitives available.
- b) Explain how your algorithm works.

Problem 3 (4 Points)

Consider the following software approach to mutual exclusion between N processes numbered 0, 1, ... , and $N - 1$. `num[4]` and `testing[4]` are shared global arrays (integer and boolean respectively).

```
Process i:
1  testing[i] = TRUE;
2  num[i] = 1 + Max(num);    // Max() returns maximum of all num[]
3  testing[i] = FALSE;
4  for j=0 to N-1 {
5      while (testing[j]) { ... spin ... }
6      while ((num[j] != 0) and ((num[j], j) < (num[i], i))) {...spin...}
7  } // NOTE: (a,b) < (x,y) if (a<x) OR ((a==x) AND (b<y))
8  ... critical section ...
9  num[i] = 0;
```

- a) All software mutual exclusion algorithms have two rules: 1) An initial ordering rule and 2) a tie-breaking rule. What are these two rules for this algorithm?
- b) Does this algorithm guarantee fairness? Explain.

Problem 4 (6 Points)

The bank teller example with 1 teller given in class has the following features;

- There are $M = 1$ tellers and N customers.
- The bank lobby has a capacity of $K = 20$ customers.
- One teller services customers one at a time from a common FIFO queue.
- The teller tells customers when he/she is ready before a customer can come to the teller window.
- Customers arriving to a full lobby return after a random delay.

Here is the algorithm given in class:

Global Variables:

```
Semaphore      tRdy = 0, cRdy = 0, tDone = 0;
int            n;           // # in lobby
Semaphore      nLock = 1;   // protect n
```

```
Process customer (int i) {
  do forever {
    ... Random Delay ...
  } until [[ n < 20 &&
           n = n+1; ]]
  Wait(tRdy);
  Signal(cRdy);
  ... Get service ...
  Wait(tDone);
  [[ n = n-1; ]] // [[ ]] means Wait(nLock); ... Signal(nLock);
  ... Leave bank ...
}

Process teller (int i) {
  do forever {
    Signal(tRdy);
    Wait(cRdy);
    ... Serve customer ...
    Signal(tDone);
  }
}
```

The bank has decided to add more tellers so that there will be a total of M tellers to serve the single line of customers. But the bank needs help in deciding how to implement the service so that there is no confusion between the customers and the tellers. The most difficult problem seems to be the case when more than one teller is free to service customers. Industry reports indicate that there will be confusion when more than one teller waives their hand indicating that they are free. So, the bank would like to make sure that only one free teller signals the next customer and when there is more than one free teller that they get the customers in a first-come-first-served order.

- a) Describe the key conceptual changes that you made to the original algorithm and the rationale for these changes.
- b) Give the new algorithm that will meet the above requirements.
- b) Mark with a '+' in the left margin those lines that are different from the original algorithm.

Problem 5 (4 Points) [From Tanenbaum]

A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	1 0 2 1 1	1 1 3 1 2	0 0 x 1 1
Process B	2 0 1 1 0	2 2 3 1 0	
Process C	1 1 0 1 0	2 1 4 1 0	
Process D	1 1 1 1 0	1 1 3 2 1	

What is the smallest value of x for which this is a safe state? Explain.

Problem 6 (6 Points) [From Bic and Shaw]

Consider the following asynchronous execution of the three processes P1, P2, and P3:

P1	P2	P3
.	.	.
Wait(X)	Wait(Y)	Wait(Z)
.	.	.
.	. <==	.
.	.	.
Wait(Z) <==	Wait(Y)	Wait(X) <==
.	.	.
Signal(X)	.	Signal(Z)
.	.	.
Signal(Z)	.	Signal(X)

X, Y and Z are semaphores whose counters have been initialized to 1. The arrows indicate which instruction the corresponding process is currently executing.

- Draw the *resource graph* for the above situation where each semaphore is interpreted as a resource, and the **Wait** and **Signal** operations represent *requests* and *releases* of the resource.
- The *state* of a resource graph is the set of request edges ($P_i \rightarrow R_j$) and allocation edges ($R_j \rightarrow P_i$). The state representing the situation in Part a is a *deadlock state* because it contains deadlocked processes. Which processes are deadlocked?
- If you could increase the number of units (i.e., initial counter value) of any of the three resources, which increase (if any) would resolve the deadlock? Explain.

Extra Credit (Choose One Problem)

Problem 7 (2 Points)

A student claims that the algorithm in Problem 3 would still guarantee mutual exclusion if we replaced the **Max** function in Line 2 with the function **PositiveRandom**() which returned a positive random integer. Discuss whether this claim is true or not. If not true, give a counterexample. If true, explain why the claim is true and whether there are any positive or negative consequences of replacing the **Max** function with **PositiveRandom**.

Problem 8 (2 Points) [From Tanenbaum]

A system has a single resource type with r units and p processes. If each process needs a maximum of m resources, derive a lower bound on the number of resources required to make the system deadlock free.

Problem 9 (2 Points) [From Bic and Shaw]

Prove by counter example that a cycle in a resource graph is not a sufficient condition for deadlock (it is a necessary condition).

A Simple Assembler Language (ASAL)

The following specifies the syntax and informal semantics of the assembler language for a RISC-like machine. BNF (Backus-Naur Form) is used to describe the syntax where the metasymbols are ::= (is defined as), <...> (class), and | (or). All other symbols are terminal symbols. Note that everything to the right of the # symbol should be treated as a comment.

```
<program> ::= <statement> ; | # a single statement
           <label>: <statement> ; | # labeled statement
           <statement> ; <program> | # sequence of statements
<statement> ::= <register> <--> <variable> | # load
               <register> <--> <integer> | # load immediate
               <register> <--> <variable> | # store
               <variable> <--> <register> | # atomic swap
               goto <label> | # branch
               <conditional> | # conditional
               <operation> | # arithmetic operation
<register> ::= R0 | R1 | R2 | R3
<variable> ::= C-variable name # name of memory location
<label> ::= L0 | L1 | ... | L9
<integer> ::= an integer
<conditional> ::= ifeq0 <register> goto <label> | # if reg equal 0
                ifne0 <register> goto <label> | # if reg not equal 0
                ifgt0 <register> goto <label> | # if reg > 0
                iflt0 <register> goto <label> | # if reg < 0
```

In this language, a variable refers to the contents of a memory location, and a register name refers to the contents of a register. In a multiprocessor, each CPU has its own set of registers. For example, if there are four processors, each processor has R0, R1, R2 and R3 registers. The following is a program that continuously examines a memory location until it contains a 0 and then loads the value 1 into x:

```
...
L0:  R1 <-- 1;           # load the constant 1
     R0 <-- x;         # load R0 with contents of x
     ifne0 R0 goto L0; # branch to L0 if R0 not equal to 0
     R1 <-- x;         # put a 1 into x
```