Sheet4

CONCURRENCY: MUTUAL EXCLUSION

1) List four design issues for which the concept of concurrency is relevant.
   - Communication among processes.
   - Sharing of and competing for resources.
   - Synchronization of the activities of multiple processes.
   - Allocation of processor time to processes.
2) What are three contexts in which concurrency arise?
   - Multiple applications.
   - Structured applications.
   - Operating-system structure.
3) What is the basic requirement for the execution of concurrent processes?
   The ability to enforce mutual exclusion.
4) List three degrees of awareness between processes and briefly define each.
   - Processes unaware of each other: These are independent processes that are not intended to work together.
   - Processes indirectly aware of each other: These are processes that are not necessarily aware of each other by their respective process IDs, but that share access to some object, such as an I/O buffer.
   - Processes directly aware of each other: These are processes that are able to communicate with each other by process ID and which are designed to work jointly on some activity.
5) What is the distinction between competing processes and cooperating processes?
   Competing processes need access to the same resource at the same time, such as a disk, file, or printer. Cooperating processes either share access to a common object, such as a memory buffer or are able to communicate with each other, and cooperate in the performance of some application or activity.
6) List the three control problems associated with competing processes and briefly define each.
   - Mutual exclusion: competing processes can only access a resource that both wish to access one at a time; mutual exclusion mechanisms must enforce this one-at-a-time policy.
   - Deadlock: if competing processes need exclusive access to more than one resource then deadlock can occur if each processes gained control of one resource and is waiting for the other resource.
   - Starvation: one of a set of competing processes may be indefinitely denied access to a needed resource because other members of the set are monopolizing that resource.

7) List the requirements for mutual exclusion.
   - Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
   - A process that halts in its non-critical section must do so without interfering with other processes.
   - It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
   - When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
   - No assumptions are made about relative process speeds or number of processors.
   - A process remains inside its critical section for a finite time only.
8) What operations can be performed on a semaphore?
   - A semaphore may be initialized to a nonnegative value.
   - The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
   - The signal operation increments the semaphore value. If the value is not positive, then a process blocked by a wait operation is unblocked
9) What is the difference between binary and general semaphores?
   A binary semaphore may only take on the values 0 and 1. A general semaphore may take on any integer value.
10) What is the difference between strong and weak semaphores?
   A strong semaphore requires that processes that are blocked on that semaphore are unblocked using a first-in-first-out policy. A weak semaphore does not dictate the order in which blocked processes are unblocked.
11) At the beginning of Section 5.1, it is stated that multiprogramming and multiprocessing present the same problems, with respect to concurrency. This is true as far as it goes. However, cite two differences in terms of concurrency between multiprogramming and multiprocessing.
   On uniprocessors you can avoid interruption and thus concurrency by disabling interrupts. Also on multiprocessor machines another problem arises: memory ordering (multiple processors accessing the memory unit).
12) Consider the following program:

```
   P1: {
shared int x;
x = 10;
while (1) {
    x = x - 1;
    x = x + 1;
    if (x != 10)
      printf("x is %d",x)
    }
  }
}
```

```
   P2:{
shared int x;
x = 10;
while ( 1 ) {
    x = x - 1;
    x = x + 1;
    if (x!=10)
      printf("x is %d",x)
    }
  }
}
```

Note that the scheduler in a uniprocessor system would implement pseudo-parallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

a) Show a sequence (i.e., trace the sequence of interleavings of statements) such that the statement "x is 10" is printed.

b) Show a sequence such that the statement "x is 8" is printed. You should remember that the increment/decrements at the source language level are not done atomically, that is, the assembly language code:

```
LD R0,X          /* load R0 from memory location x */
INCR R0          /* increment R0 */
STO R0,X         /* store the incremented value back in X */
```

Implements the single C increment instruction (x = x + 1).

a) For "x is 10", the interleaving producing the required behavior is easy to find since it requires only an interleaving at the source language statement level. The essential fact here is that the test for the value of x is interleaved with the increment of x by the other process. Thus, x was not equal to 10 when the test was performed, but was equal to 10 by the time the value of x was read from memory for printing.

	M(x)
P1: x = x - 1;	9
P1: x = x + 1;	10
P2: x = x - 1;	9
P1: if(x != 10)	9
P2: x = x + 1;	10
P1: printf("x is %d", x);	10

"x is 10" is printed.

b) For "x is 8" we need to be more inventive, since we need to use interleavings of the machine instructions to find a way for the value of x to be established as 9 so it can then be evaluated as 8 in a later cycle. Notice how the first two blocks of statements correspond to C source lines, but how later blocks of machine language statements interleave portions of a source language statement.

Instruction	M(x)	P1-R0	P2-R0
P1: LD R0, x	10	10	--
P1: DECR R0	10	9	--
P1: STO R0, x	9	9	--
P2: LD R0, x	9	9	9
P2: DECR R0	9	9	8
P2: STO R0, x	8	9	8
P1: LD R0, x	8	8	8
P1: INCR R0	8	9	--
P2: LD R0, x	8	9	8
P2: INCR R0	8	9	9
P2: STO R0, x	9	9	9
P2: if(x != 10) printf("x is %d", x);			
P2: "x is 9" is printed.			
P1: STO R0, x	9	9	9
P1: if(x != 10) printf("x is %d", x);			

```
P1: "x is 9" is printed.
P1: LD R0, x              9          9          9
P1: DECR R0               9          8          --
P1: STO R0, x             8          8          --
P2: LD R0, x              8          8          8
P2: DECR R0               8          8          7
P2: STO R0, x             7          8          7
P1: LD R0, x              7          7          7
P1: INCR R0               8          8          7
P1: STO R0, x             8          8          7
P1: if(x != 10) printf("x is %d", x);
P1: "x is 8" is printed.
```

13) Consider the following program:

```
const int n = 50;
int tally;
void total() {
     int count;
     for (count = 1; count <= n; count++) {
          tally++;
     }
}
void main() {
     tally = 0;
     parbegin (total (), total ());
     write (tally);
}
```

a) Determine the proper lower bound and upper bound on the final value of the shared variable tally output by this concurrent program. Assume processes can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

b) Suppose that an arbitrary number of these processes are permitted to execute in parallel under the assumptions of part (a). What effect will this modification have on the range of final values of tally?

a) On casual inspection, it appears that tally will fall in the range $50 \leq tally \leq 100$ since from 0 to 50 increments could go unrecorded due to the lack of mutual exclusion. The basic argument contends that by running these two processes concurrently we should not be able to derive a result lower than the result produced by executing just one of these processes sequentially. But consider the following interleaved sequence of the load, increment, and store operations performed by these two processes when altering the value of the shared variable:

(1) Process A loads the value of tally, increments tally, but then loses the processor (it has incremented its register to 1, but has not yet stored this value.

(2) Process B loads the value of tally (still zero) and performs forty-nine complete increment operations, losing the processor after it has stored the value 49 into the shared variable tally.

(3) Process A regains control long enough to perform its first store operation (replacing the previous tally value of 49 with 1) but is then immediately forced to relinquish the processor.

(4) Process B resumes long enough to load 1 (the current value of tally) into its register, but then it too is forced to give up the processor (note that this was B's final load).

(5) Process A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of tally to 50.

(6) Process B is reactivated with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.

Some thought will reveal that a value lower than 2 cannot occur. Thus, the proper range of final values is $2 \leq tally \leq 100$.

b) For the generalized case of N processes, the range of final values is 2 □ tally □ (N □ 50), since it is possible for all other processes to be initially scheduled and run to completion in step (5) before Process B would finally destroy their work by finishing last.

14) Is busy waiting always less efficient (in terms of using processor time) than a blocking wait? Explain.

On average, yes, because busy-waiting consumes useless instruction cycles. However, in a particular case, if a process comes to a point in the program where it must wait for a condition to be satisfied, and if that condition is already satisfied, then the busy-wait will find that out immediately, whereas, the blocking wait will consume OS resources switching out of and back into the process.

15) Consider the following program:

```
boolean blocked [2];
int turn;
void P (int id)
{
 while (true) {
    blocked[id] = true;
    while (turn != id) {
       while (blocked[1-id])
          /* do nothing */;
       turn = id;
    }
    /* critical section */
    blocked[id] = false;
    /* remainder */
 }
}
void main()
{
 blocked[0] = false;
 blocked[1] = false;
 turn = 0;
 parbegin (P(0), P(1));
}
```

This software solution to the mutual exclusion problem for two processes is proposed. Find a counterexample that demonstrates that this solution is incorrect.

Consider the case in which turn equals 0 and P(1) sets blocked[1] to true and then finds blocked[0] set to false. P(0) will then set blocked[0] to true, find turn = 0, and enter its critical section. P(1) will then assign 1 to turn and will also enter its critical section

16) A software approach to mutual exclusion is Lamport's bakery algorithm [LAMP74], so called because it is based on the practice in bakeries and other shops in which every customer receives a numbered ticket on arrival, allowing each to be served in turn. The algorithm is as follows:

```
boolean choosing[n];
int number[n];
while (true) {
   choosing[i] = true;
   number[i] = 1 + getmax(number[], n);
   choosing[i] = false;
   for (int j = 0; j < n; j++){
     while (choosing[j]) { };
     while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
   }
   /* critical section */;
   number [i] = 0;
   /* remainder */;
}
```

The arrays choosing and number are initialized to false and 0, respectively. The i th element of each array may be read and written by process i but only read by other processes. The notation (a , b) < (c , d) is defined as: (a < c) or (a = c and b < d)

a) Describe the algorithm in words.

When a process wishes to enter its critical section, it is assigned a ticket number. The ticket number assigned is calculated by adding one to the largest of the ticket numbers currently held by the processes waiting to enter their critical section and the process already in its critical section. The process with the smallest ticket number has the highest precedence for entering its critical section. In case more than one process receives the same ticket number, the process with the smallest numerical name enters its critical section. When a process exits its critical section, it resets its ticket number to zero.

b) Show that this algorithm avoids deadlock.

If each process is assigned a unique process number, then there is a unique, strict ordering of processes at all times. Therefore, deadlock cannot occur.

c) Show that it enforces mutual exclusion.

To demonstrate mutual exclusion, we first need to prove the following lemma: if Pi is in its critical section, and Pk has calculated its number[k] and is attempting to enter its critical section, then the following relationship holds:

(number[i], i) < (number[k], k)

To prove the lemma, define the following times:

- T_{w1} → Pi reads choosing[k] for the last time, for j = k, in its first wait, so we have choosing[k] = false at T_{w1}.
- T_{w2} → Pi begins its final execution, for j = k, of the second while loop. We therefore have $T_{w1} < T_{w2}$.
- T_{k1} → Pk enters the beginning of the repeat loop.
- T_{k2} → Pk finishes calculating number[k].
- T_{k3} → Pk sets choosing[k] to false.

We have $T_{k1} < T_{k2} < T_{k3}$.

Since at T_{w1}, choosing[k] = false, we have either $T_{w1}<T_{k1}$ or $T_{k3}<T_{w1}$. In the first case, we have number[i] < number[k], since Pi was assigned its number prior to Pk; this satisfies the condition of the lemma.

In the second case, we have $T_{k2}<T_{k3}<T_{w1}<T_{w2}$, and therefore $T_{k2}<T_{w2}$. This means that at T_{w2}, Pi has read the current value of number[k]. Moreover, as T_{w2} is the moment at which the final execution of the second `while` for j = k takes place, we have (number[i], i) < (number[k], k), which completes the proof of the lemma.

It is now easy to show the mutual exclusion is enforced. Assume that Pi is in its critical section and Pk is attempting to enter its critical section. Pk will be unable to enter its critical section, as it will find number[i]≠ 0 and (number[i], i) < (number[k], k).

17) Now consider a version of the bakery algorithm without the variable choosing. Then we have

```
1 int number[n];
2 while (true) {
3    number[i] = 1 + getmax(number[], n);
4    for (int j = 0; j < n; j++){
5      while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };
6    }
7    /* critical section */;
8    number [i] = 0;
9    /* remainder */;
10 }
```

Does this version violate mutual exclusion? Explain why or why not.

Suppose we have two processes just beginning; call them p0 and p1. Both reach line 3 at the same time. Now, we'll assume both read number[0] and number[1] before either addition takes place. Let p1 complete the line, assigning 1 to number[1], but p0 block before the assignment. Then p1 gets through the while loop at line 5 and enters the critical section. While in the critical section, it blocks; p0 unblocks, and assigns 1 to number[0] at line 3. It proceeds to the while loop at line 5. When it goes through that loop for j = 1, the first condition on line 5 is true. Further, the second condition on line 5 is false, so p0 enters the critical section. Now p0 and p1 are both in the critical section, violating mutual exclusion. The reason for choosing is to prevent the while loop in line 5 from being entered when process j is setting its number[j]. Note that if the loop is entered and then process j reaches line 3, one of two situations arises. Either number[j] has the value 0 when the first test is executed, in which case process i moves on to the next process, or number[j] has a non-zero value, in which case at some point number[j] will be greater than number[i] (since process i finished executing statement 3 before process j began). Either way, process i will enter the critical section before process j, and when process j reaches the while loop, it will loop at least until process i leaves the critical section.
