Name: Sample Solutions   Student #: ____________

Engineering 8893
Concurrent Programming
Mid-Term Test

Dr. D. K. Peters

March 2, 2004

Instructions: Answer all questions. Write your answers on this paper. This is a closed book test, no textbooks, notes, calculators or other aides are permitted.

Q1	/10
Q2	/ 5
Q3	/10
Q4	/25
Total	/ 50

Axioms & Inference Rules

Assignment Axiom: $\{P_{\texttt{x}\leftarrow e}\}$ `x := e` $\{P\}$

Composition Rule: $$\frac{\{P\}\ \texttt{S}_1\ \{Q\},\ \{Q\}\ \texttt{S}_2\ \{R\}}{\{P\}\ \texttt{S}_1;\texttt{S}_2\ \{R\}}$$

If Statement Rule: $$\frac{\{P \wedge B\}\ \texttt{S}\ \{Q\},\ (P \wedge \neg B) \rightarrow Q}{\{P\}\ \texttt{if (B) S;}\ \{Q\}}$$

While Statement Rule: $$\frac{\{I \wedge B\}\ \texttt{S}\ \{I\}}{\{I\}\ \texttt{while (B) S;}\ \{I \wedge \neg B\}}$$

Rule of Consequence: $$\frac{P' \rightarrow P,\ \{P\}\ \texttt{S}\ \{Q\},\ Q \rightarrow Q',}{\{P'\}\ \texttt{S}\ \{Q'\}}$$

Await Statement Rule: $$\frac{\{P \wedge B\}\ \texttt{S}\ \{Q\},}{\{P\}\ \texttt{< await (B) S; >}\ \{Q\}}$$

Co Statement Rule: $$\frac{\{P_i\}\texttt{S}_i\{Q_i\} \text{ are interference free}}{\{P_1 \wedge \ldots \wedge P_n\}\ \texttt{co S}_1\texttt{; //} \ldots \texttt{// S}_n\texttt{; oc}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

Semaphore Rules: $$\frac{(R \wedge g > 0) \rightarrow Q_{g\leftarrow(g-1)}}{\{R\}\texttt{P(g)}\{Q\}}$$

$$\frac{R \rightarrow Q_{g\leftarrow(g+1)}}{\{R\}\texttt{V(g)}\{Q\}}$$

1. [10 points] Consider the following program:

```
int x := 0;
## { x == 0 }
co
  ## { P1 }
  < x := x + 2 >
  ## { Q1 }
  //
  ## { P2 }
  < x := x - 3 >
  ## { Q2 }
  //
  ## { P3 }
  < x := x + 4 >
  ## { Q3 }
oc
## { x == 3 }
```

a) [6 points] Give the assertions P1, Q1, P2, Q2, P3 and Q3 so that the above program is a correct proof outline. (Hint: use the technique of weakened assertions.)

$\mathtt{P1} \overset{df}{=} x = 0 \lor x = -3 \lor x = 4 \lor x = 1$
$\mathtt{Q1} \overset{df}{=} x = 2 \lor x = -1 \lor x = 6 \lor x = 3$
$\mathtt{P2} \overset{df}{=} x = 0 \lor x = 2 \lor x = 4 \lor x = 6$
$\mathtt{Q2} \overset{df}{=} x = -3 \lor x = -1 \lor x = 1 \lor x = 3$
$\mathtt{P3} \overset{df}{=} x = 0 \lor x = 2 \lor x = -3 \lor x = -1$
$\mathtt{Q3} \overset{df}{=} x = 4 \lor x = 6 \lor x = 1 \lor x = 3$

b) [4 points] List (but do not prove) the triples that must be verified to show non-interference.

$\{\mathtt{P1} \land \mathtt{P2}\}\ \langle x := x + 2\rangle\ \{\mathtt{P2}\}$
$\{\mathtt{P1} \land \mathtt{Q2}\}\ \langle x := x + 2\rangle\ \{\mathtt{Q2}\}$
$\{\mathtt{P1} \land \mathtt{P3}\}\ \langle x := x + 2\rangle\ \{\mathtt{P3}\}$
$\{\mathtt{P1} \land \mathtt{Q3}\}\ \langle x := x + 2\rangle\ \{\mathtt{Q3}\}$
$\{\mathtt{P2} \land \mathtt{P1}\}\ \langle x := x - 3\rangle\ \{\mathtt{P1}\}$
$\{\mathtt{P2} \land \mathtt{Q1}\}\ \langle x := x - 3\rangle\ \{\mathtt{Q1}\}$
$\{\mathtt{P2} \land \mathtt{P3}\}\ \langle x := x - 3\rangle\ \{\mathtt{P3}\}$
$\{\mathtt{P2} \land \mathtt{Q3}\}\ \langle x := x - 3\rangle\ \{\mathtt{Q3}\}$
$\{\mathtt{P3} \land \mathtt{P1}\}\ \langle x := x + 4\rangle\ \{\mathtt{P1}\}$
$\{\mathtt{P3} \land \mathtt{Q1}\}\ \langle x := x + 4\rangle\ \{\mathtt{Q1}\}$
$\{\mathtt{P3} \land \mathtt{P2}\}\ \langle x := x + 4\rangle\ \{\mathtt{P2}\}$
$\{\mathtt{P3} \land \mathtt{Q2}\}\ \langle x := x + 4\rangle\ \{\mathtt{Q2}\}$

2. [5 points] Consider the following program:

```
co < await (x > 0) x := x - 1; >
  // < await (x < 0) x := x + 2; >
  // < await (x == 0) x := x - 1; >
oc
```

a) [2 points] Assuming that the scheduling is weakly fair, for what initial values of x does the program terminate? Explain.

$-1 \leq x \leq 1$ — *For all of these values an* `await` *condition is true initially and the code makes another condition true, which then makes the third true.*

b) [1 point] What are the final value(s) of x in (a)?

Initial x	*Final* x
-1	-1
0	0
1	1

c) [2 points] Does the fairness assumption in (a) matter? Explain.

If we assume that these are the only three processes operating then it doesn't. However, if any other processes are running on the system then weak fairness is required to assure that these processes will eventually get to execute.

3. [10 points] Consider the following proposed solution to the two process critical section problem.

```
int n0 := 0;
int n1 := 0;

process P0 {                              process P1 {
    non_critical_section_0;                   non_critical_section_1;
    n0 := 1;                                  n1 := 1;
    n0 := n1 + 1;                             n1 := n0 + 1;
    while (n1 != 0 && n1 <= n0) skip;         while (n0 != 0 && n0 < n1) skip;
    critical_section_0;                       critical_section_1;
    n0 := 0;                                  n1 := 0;
}                                         }
```

For each of the essential properties of a critical section solution listed in (a) through (d), state if the above solution satisfies the property or not, and justify your answer.

a) [2 points] Mutual exclusion.

Yes. In a manner similar to the proof of Peterson's algorithm (which this is a slight variation on) in the notes, let's add thought variables `r0` *and* `r1` *as follows:*

```
int n0 := 0;
int n1 := 0;
bool r0 := false;
bool r1 := false;

process P0 {                              process P1 {
    non_critical_section_0;                   non_critical_section_1;
    < n0 := 1; r0 = true; >                   < n1 := 1; r1 = true; >
    < n0 := n1 + 1; r0 = false; >             < n1 := n0 + 1; r1 = false; >
    while (n1 != 0 && n1 <= n0) skip;         while (n0 != 0 && n0 < n1) skip;
    ## n0 >= 1 /\ !r0 /\                      ## n1 >= 1 /\ !r1 /\
    ## (r1 \/ n1 == 0 \/ n0 < n1)             ## (r0 \/ n0 == 0 \/ n1 <= n0)
    critical_section_0;                       critical_section_1;
    n0 := 0;                                  n1 := 0;
}                                         }
```

There is no interferece with the assertions and they cannot both be true at once, so the configuration is excluded.

b) [2 points] Absence of deadlock.

Yes. The conditions on the spin loops cannot both be true, so one will always get to enter.

c) [2 points] Absence of unnecessary delays.

Yes. In the case where the other process is in its non-critical section then the corresponding n*x variable will be 0 and the other process will enter immediately.*

d) [2 points] Eventual entry.

Yes. The algorithm functions like Peterson's algorithm in the use of `n`x *as a 'turn' indicator. When there is constant contention (i.e., when the critical section is much longer than the non-critical section) the processes will alternate turns in the critical sections.*

e) [2 points] Why are the assignments `n0 := 1` and `n1 := 1` necessary in the above algorithm?

Without these assignments mutual exclusion is not assured. Consider the following sequence (initially $\texttt{n0} == \texttt{n1} == 0$*):*

P0	***P1***
	$r_i := \texttt{n0} + 1$
$\texttt{n0} := \texttt{n1} + 1$	
$\texttt{n1}! = 0$	
critical_section_0	
	$\texttt{n1} := r_i$
	$\texttt{n0}! = 0 \&\& \texttt{n0} < \texttt{n1}$
	critical_section_1

4. [25 points] *The One-Lane Bridge.* Cars coming from the east and west arrive at a one-lane bridge. Cars heading in the same direction can cross the bridge at the same time, but cars heading in opposite directions cannot. Cars arriving from the east call `eastEnter` before beginning to cross the bridge and `eastExit` when they have crossed the bridge. Similarly cars arriving from the west call `westEnter` to enter, and `westExit` to leave.

   a) [5 points] Give a the pseudo-code for a coarse-grained implementation (i.e., using conditional synchronization statements) for the four enter and exit functions, including declarations and initial values for all shared variables. Your solution **need not** ensure eventual entry but should not cause unnecessary delay (i.e., a car entering from one direction should never have to wait if there are no cars from the other direction either on the bridge or waiting).

```
int numEast = 0; # number of cars on bridge from east
int numWest = 0; # number of cars on bridge from west
## INV: numEast > 0 -> numWest == 0 /\
##      numWest > 0 -> numEast == 0

proc eastEnter() {
  < await(numWest == 0) numEast++; >
}

proc eastExit() {
  < numEast--; >
}

proc westEnter() {
  < await(numEast == 0) numWest++; >
}

proc westExit() {
  < numWest--; >
}
```

b) [5 points] Modify your solution in (a), above, such that it ensures eventual entry (i.e., a continuous stream of cars from one direction shouldn't indefinitely delay a car attempting to enter from the opposite direction). Give a the pseudo-code for a coarse-grained implementation for the four enter and exit functions, including declarations and initial values for all shared variables. *(If your solution in (a) does ensure eventual entry, then simply state this.)*

```
int numEast = 0; # number of cars on bridge from east
int numWest = 0; # number of cars on bridge from west
int delayedEast = 0; # number of cars waiting at east entry
int delayedWest = 0; # number of cars waiting at west entry
int turn = 0; # 0 = east takes priority, 1 = west takes priority
## INV: numEast > 0 -> numWest == 0 /\
##      numWest > 0 -> numEast == 0

proc eastEnter() {
  delayedEast++;
  < await(numWest == 0 && (delayedWest == 0 || turn == 0))
      numEast++; delayedEast--; >
}

proc eastExit() {
  < numEast--; turn := 1; >
}

proc westEnter() {
  delayedWest++;
  < await(numEast == 0 && (delayedEast == 0 || turn == 1))
      numWest++; delyedWest--; >
}

proc westExit() {
  < numWest--; turn := 0; >
}
```

c) [15 points] Using the `Semaphore` class (as used in assignment #2, with methods `P` and `V`) for synchronization, give a pseudo-Java implementation of a class that implements your solution in (b), above. The class must have public methods `eastEnter`, `westEnter`, `eastExit`, `westExit` and a constructor to give the initial values of all the variables. Do **not** use the `synchronized` Java keyword. *An implementation of a solution that does not ensure eventual entry (i.e., from (a)) will be considered for a maximum of 10 points.*

```
/*****************************************************************
 * Control access to the one lane bridge.
 *****************************************************************/
class BridgeFair {
  /** mutex semaphore */
  Semaphore entry;
  /** east entry semaphore */
  Semaphore eastEntry;
  /** west entry semaphore */
  Semaphore westEntry;
  /** number of cars on bridge from East */
  int numEast;
  /** number of cars on bridge from West */
  int numWest;
  /** number of cars waiting to enter from the East */
  int delayedEast;
  /** number of cars waiting to enter from the West */
  int delayedWest;
  /** 0 = east takes priority, 1 = west takes priority */
  int turn;

  /**
   * @param maxR Maximum number of concurrent readers
   */
  BridgeFair()
  {
    entry = new Semaphore(1);
    eastEntry = new Semaphore(0);
    westEntry = new Semaphore(0);
    numEast = 0;
    numWest = 0;
    delayedEast = 0;
    delayedWest = 0;
    turn = 0;
  }
```

```
/**************************************************************
 * Request permission to enter from the East.
 * @param id Requesting car.
 * @throws InterruptedException
 *************************************************************/
public void eastEnter(int id)
  throws InterruptedException
{
  entry.P();
  if (numWest > 0 || (delayedWest > 0 && turn == 1)) {
    delayedEast++;
    System.out.println("EAST " + id + " waiting.");
    entry.V();
    eastEntry.P();
  }
  System.out.println("EAST " + id + " enters.");
  numEast++;
  signal();
}

/**************************************************************
 * exit from East.
 * @param id car exiting.
 * @throws InterruptedException
 *************************************************************/
public void eastExit(int id)
  throws InterruptedException
{
  entry.P();
  System.out.println("EAST " + id + " exit.");
  numEast--;
  turn = 1; // Allow entry from the West if any waiting
  signal();
}
```

```
/****************************************************************
 * Request permission to enter from the West.
 * @param id Requesting car.
 * @throws InterruptedException
 ***************************************************************/
public void westEnter(int id)
  throws InterruptedException
{
  entry.P();
  if (numEast > 0 || (delayedEast > 0 && turn == 0)) {
    delayedWest++;
    System.out.println("WEST " + id + " waiting.");
    entry.V();
    westEntry.P();
  }
  System.out.println("WEST " + id + " enters.");
  numWest++;
  signal();
}

/****************************************************************
 * exit from West.
 * @param id car exiting.
 * @throws InterruptedException
 ***************************************************************/
public void westExit(int id)
  throws InterruptedException
{
  entry.P();
  System.out.println("WEST " + id + " exit.");
  numWest--;
  turn = 0; // Allow entry from the East if any waiting
  signal();
}
```

```
  /**************************************************************
   * signal some waiting process
   **************************************************************/
  private void signal()
    throws InterruptedException
  {
    checkSafety();
    if (numWest == 0 && delayedEast > 0 && (delayedWest == 0 || turn == 0)) {
      // Let in from the East
      delayedEast--;
      eastEntry.V();
    } else if (numEast == 0 && delayedWest > 0 &&
                (delayedEast == 0 || turn == 1)) {
      // Let in from the West
      delayedWest--;
      westEntry.V();
    } else { // nobody waiting, allow entry again
      entry.V();
    }
  }

  /**************************************************************
   * check safety property
   ***************************************************************/
  private void checkSafety()
  {
    if (numEast > 0 && numWest > 0) {
      System.out.println("CRASH: numEast = " + numEast +
                         ", numWest = " + numWest + " -- CRASH");
    }
  }
}
```