In this lab you will be running various synchronization tasks under simulation. Read the note on simulating concurrency that was handed out.

Part 1. We start with binary semaphores and critical sections. The routines to be done in parallel are simple user programs including a critical section

```
loop
        P
        CS
        V
        NCS
```

For P and V use Decker's algorithm with 2 tasks.
For CS use

```
atomic ← atomic+1
loop 200 times {empty loop body}
"print" atomic
```

Let atomic be a shared variable (i.e. the same atomic in each task) initialized to 0. Since the critical section is indivisible, each time atomic is set to atomic+1 and the values printed are 0, 1, 2, ... Do NOT actually print these values (save the trees). Instead write them out to a file and have an auxilary program check the file to see that the numbers are consecutive.

For NCS use

```
loop 0 or 200 times {empty loop body - see below}
nonatomic ← nonatomic+1
loop 200 times {empty loop body}
"print" nonatomic
```

Let nonatomic be a shared variable initialized to 0. Since the non-critical section is not indivisible, funny things happen (duplicate values and skipped values occur). The loop 0 or 200 times requires a coin flip. It is used to prevent the CS from synchronizing the tasks so that only one at a time enters the NCS. Again, do not print the values; your auxilary program should report gaps and duplicates (do NOT generate hundreds or thousands of lines of output—the program should SUMARIZE).

To determine if the program works for two tasks, in addition to the auxilary program just mentioned, declare a shared array int count[4,4] and augment the main loop to the following:

```
Task1
        loop
                state[1]:=0; count[state[1],state[2]]++; P;
                state[1]:=1; count[state[1],state[2]]++; CS;
                state[1]:=2; count[state[1],state[2]]++; V;
                state[1]:=3; count[state[1],state[2]]++; NCS;

Task2
        loop
                state[2]:=0; count[state[1],state[2]]++; P;
                state[2]:=1; count[state[1],state[2]]++; CS;
                state[2]:=2; count[state[1],state[2]]++; V;
                state[2]:=3; count[state[1],state[2]]++; NCS;
```

Do not return to your main program after a statement involving state. Print the matrix count at the end of the run. Note that count[1,1] must be 0.

Now make another run using P and V built from test-and-set.

Part 2. The second part is to use a counting semaphore and semicritical sections. Use 3 tasks and initialize the semaphore to 2 (so that up to 2 tasks may be in the semicritical section). The user programs are

```
loop
  P
  SCS
  V
  NCS
```

For P and V use a counting semaphore built from three binary semaphores, which are themselves built from test-and-set (these are the binary semaphores you built at the end of part 1).

For SCS use (NumInSCS is initially 0 and is shared).

```
NumInSCS ← NumInSCS+1
if NumInSCS > 2 then panic
loop 0 or 200 times
NumInSCS ← NumInSCS-1
```

Panic is a shorthand for print message, stop run, and debug program. Since we made the updates to NumInSCS atomic by putting them on a single line, the code should work and the panic should never happen (but you still must put in code for it).

For NCS use the same as SCS but with NumInSCS replaced by NumInNCS. For the NCS, the panic is expected to happen. That is, if NumInNCS>2 in the NCS, instead of panicing, you should record this event and, at the end of the run, print out the number of times the event occured. This number should be non zero. As in part 1, maintain a count matrix and print it at the end of the run. Since there are three tasks each with four states, count is three dimensional (int count[4,4,4]).

Now make another run using counting semaphores built from directly from fetch-and-add.

Part 3. Next do a producer consumer synchronization of three tasks. One task is

```
loop
        if flip_coin = heads
                producer
        else
                consumer
```

The second task is simply

```
loop
        consumer
```

The third task is simply

```
loop
        producer
```

Implement producer consumer using test-and-set binary semaphores and and fetch-and-add counting semaphores.

The producers execute the following code where: x, y, and atomic are private, nonatomic is shared, and each are initially 0; and t# is the task number, which is a separate constant for each task.

```
x ← atomic+1
y ← nonatomic+1
loop 0 or 200 times
atomic ← x
nonatomic ← y
Put (atomic,nonatomic,t#) into buffer
```

The consumers just “print” the tuples they consume (again this means that an auxilary program summarizes the anomalies that occur). Use a buffer of size 50.

Part 4. Do a readers writers synchronization with 3 reader tasks and 2 writer tasks. The read body is (r shared initially 0)

```
r ← r+1
if r>1 note reader concurrency {i.e. print summary at end of run}
loop randomly from 50 to 200 times
r ← r-1
```

The write body is (w shared initially 0)

```
w ← w+1
if w>1 panic
loop randomly from 50 to 200 times
w ← w-1
```

Use writer-priority readers writers implemented with fetch-and-add (i.e. the "activity counter" method).

Part 5. Implement a barrier synchronization with fetch-and-increment and fetch-and-decrement. Run the code below on 5 processes. A is shared integer array with 5 elements initialized to all zeros. Once again t# is the task number, which is a separate constant for each task.

```
loop
        loop 0 or 100 times
        A[t#] ← A[t#]+1
        barrier
        loop for i = 1 to 5
                if A[i] ≠ A[t#] panic
        barrier
```

Now omit the barrier from the code and record times when the array is not always all equal. That is the code becomes

```
loop
        loop 0 or 100 times
        A[t#] ← A[t#]+1
        loop for i = 1 to 5
                if A[i] ≠ A[t#] note array inconsistency
```

THE FOLLOWING PARAGRAPHS APPLY TO ALL PARTS

Have the scheduler (your main program that calls the others in turn) run for a fixed number of cycles (50000-100000 looks appropriate), then print out any summary data desired, and then terminate the run. For each part you should be printing enough to show that the program did what was intended. For example, in part 2, it should be clear from the output that no panic happened in the SCS (this could be just the absence of a printout) and that panics did occur in the NCS (there could be a bunch, so bumping a counter and printing its value at the end of the run seems appropriate).

When you are finished, try some runs with a less simplistic scheduler. That is instead of always running the tasks in order, have the scheduler look like

```
loop
        flip coin to choose i
        call task number i
```

If you use an unfair coin (i.e. an adversarial scheduler), you can show that the final barrier used above is necessary. I would think that the round robbin schedule would give correct results without the second barrier, which is yet more evidence of the trickiness of non-determanistic programs.