5.2. What are three contexts in which concurrency arises?
5.3. What is the basic requirement for the execution of concurrent processes?
5.5. What is the distinction between competing processes and cooperating processes?
5.7. List the requirements for mutual exclusion.

5.4. 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*?

5.6. 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 in [HYMA66]. Find a counterexample that demonstrates that this solution is incor rect. It is interesting to note that even the *Communications of the ACM* was fooled on this one.

5.12. Consider the following definition of semaphores:

```
void semWait(s)
{
     if (s.count > 0) {
       s.count--;
     }
     else {
       place this process in s.queue;
       block;
     }
}
```

```
void semSignal (s)
{
     if (there is at least one process blocked on
         semaphore s) {
         remove a process P from s.queue;
         place process P on ready list;
     }
     else
        s.count++;
}
```

Compare this set of definitions with that of Figure 5.3. Note one difference: With the preceding definition, a semaphore can never take on a negative value. Is there any difference in the effect of the two sets of definitions when used in programs? That is, could you substitute one set for the other without altering the meaning of the program?

5.13. Consider a sharable resource with the following characteristics: (1) As long as there are fewer than three processes using the resource, new processes can start using it right away. (2) Once there are three process using the resource, all three must leave before any new processes can begin using it. We realize that counters are needed to keep track of how many processes are waiting and active, and that these counters are

themselves shared resources that must be protected with mutual exclusion. So we might create the following solution:

```
1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
2   int active = 0, waiting = 0;                              /* counters, and */
3   boolean must_wait = false;                             /* state information */
4
5   semWait(mutex);                             /* Enter the mutual exclusion */
6   if(must_wait) {                          /* If there are (or were) 3, then */
7       ++waiting;                          /* we must wait, but we must leave */
8       semSignal(mutex);                       /* the mutual exclusion first */
9       semWait(block);                /* Wait for all current users to depart */
10      SemWait(mutex);                         /* Reenter the mutual exclusion */
11      --waiting;                              /* and update the waiting count */
12  }
13  ++active;                              /* Update active count, and remember */
14  must_wait = active == 3;                          /* if the count reached 3 */
15  semSignal(mutex);                           /* Leave the mutual exclusion */
16
17  /* critical section */
18
19  semWait(mutex);                                 /* Enter mutual exclusion */
20  --active;                                /* and update the active count */
21  if(active == 0) {                                   /* Last one to leave? */
22      int n;
23      if (waiting < 3) n = waiting;
24      else n = 3;                                     /* If so, unblock up to 3 */
25      while( n > 0 ) {                                 /* waiting processes */
26          semSignal(block);
27          --n;
28      }
29  must_wait = false;                       /* All active processes have left */
30  }
31  semSignal(mutex);                           /* Leave the mutual exclusion */
```

The solution appears to do everything right: All accesses to the shared variables are protected by mutual exclusion, processes do not block themselves while in the mutual exclusion, new processes are prevented from using the resource if there are (or were) three active users, and the last process to depart unblocks up to three waiting processes.

a. The program is nevertheless incorrect. Explain why.

b. Suppose we change the if in line 6 to a while. Does this solve any problem in the program? Do any difficulties remain?

5.14. Now consider this correct solution to the preceding problem:

```
1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
2   int active = 0, waiting = 0;                              /* counters, and */
3   boolean must_wait = false;                             /* state information */
4
5   semWait(mutex);                              /* Enter the mutual exclusion */
6   if(must_wait) {                           /* If there are (or were) 3, then */
7       ++waiting;                           /* we must wait, but we must leave */
8       semSignal(mutex);                        /* the mutual exclusion first */
9       semWait(block);               /* Wait for all current users to depart */
10  } else {
11      ++active;                                  /* Update active count, and */
12      must_wait = active == 3;           /* remember if the count reached 3 */
13      semSignal(mutex);                          /* Leave mutual exclusion */
14  }
15
16  /* critical section */
17
18  semWait(mutex);                                /* Enter mutual exclusion */
19  --active;                                    /* and update the active count */
20  if(active == 0) {                                    /* Last one to leave? */
21      int n;
22      if (waiting < 3) n = waiting;
23      else n = 3;               /* If so, see how many processes to unblock */
24      waiting -= n;                /* Deduct this number from waiting count */
25      active = n;                           /* and set active to this number */
26      while( n > 0 ) {                       /* Now unblock the processes */
27          semSignal(block);                                  /* one by one */
28          --n;
29      }
30      must_wait = active == 3;                /* Remember if the count is 3 */
31  }
32  semSignal(mutex);                           /* Leave the mutual exclusion */
```

a. Explain how this program works and why it is correct.
b. This solution does not completely prevent newly arriving processes from cutting in line but it does make it less likely. Give an example of cutting in line.
c. This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the **I'll Do It For You** pattern. Describe the pattern.

5.15. Now consider another correct solution to the preceding problem:

```
1   semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
2   int active = 0, waiting = 0;                              /* counters, and */
3   boolean must_wait = false;                             /* state information */
4
```

```
5   semWait(mutex);                          /* Enter the mutual exclusion */
6   if(must_wait) {                        /* If there are (or were) 3, then */
7       ++waiting;                        /* we must wait, but we must leave */
8       semSignal(mutex);                    /* the mutual exclusion first */
9       semWait(block);              /* Wait for all current users to depart */
10      --waiting;           /* We've got the mutual exclusion; update count */
11  }
12  ++active;                            /* Update active count, and remember */
13  must_wait = active == 3;                     /* if the count reached 3 */
14  if(waiting > 0 && !must_wait)            /* If there are others waiting */
15      semSignal(block);;                /* and we don't yet have 3 active, */
16                                             /* unblock a waiting process */
17  else semSignal(mutex);            /* otherwise open the mutual exclusion */
18
19  /* critical section */
20
21  semWait(mutex);                              /* Enter mutual exclusion */
22  --active;                                  /* and update the active count */
23  if(active == 0)                               /* If last one to leave? */
24      must_wait = false;               /* set up to let new processes enter */
25  if(waiting == 0 && !must_wait)           /* If there are others waiting */
26      semSignal(block);;                   /* and we don't have 3 active, */
27                                             /* unblock a waiting process */
28  else semSignal(mutex);            /* otherwise open the mutual exclusion */
```

a. Explain how this program works and why it is correct.

b. Does this solution differ from the preceding one in terms of the number of pro cesses that can be unblocked at a time? Explain.

c. This program is an example of a general design pattern that is a uniform way to implement solutions to many concurrency problems using semaphores. It has been referred to as the **Pass The Baton** pattern. Describe the pattern.