1. Describe the difference between Hoare-style (also known as signal-immediate) and Mesa-style (also known as signal-delayed) condition variables.

2. Solve Problem 7.8, the Barbershop Problem, in the book, using monitors and Mesa-style condition variables.

3. Solve Problem 7.9, the Cigarette-Smokers Problem, in the book, using monitors and Mesa-style condition variables.

4. Concurrent-Systems-R-Us has built a new operating system. This simple system provides three primitives for managing threads:

```
int  thread_create(void (*f)());
void thread_start(int threadid);
void thread_stop();
```

`thread_create`() creates a new thread and returns its thread identifier. This newly created thread is in the suspended state. `thread_start`() takes a suspended thread, marks that thread as runnable, and places it on the scheduler's run queue. The behavior of `thread_start`() is undefined if the thread is not suspended. `thread_stop`() operates implicitly on the calling thread – it marks the currently running thread as suspended, and removes it from the run queue. The scheduling policy is unspecified, but is known to be preemptive.

Concurrent-Systems-R-Us has tried to build a software-controlled radar using this system. The radar is controlled by two threads, a transmitter and a receiver thread, which are supposed to alternate (i.e. first the transmitter executes, then the receiver, then the transmitter, then the receiver, and so on). To achieve this alternation, the programmers decided to use low-level thread manipulation, instead of standard synchronization primitives like monitors with condition variables – efficiency was their excuse. Here is their code:

```
int xmit, rcv;

void transmitter() {
     while(TRUE) {
          printf("In transmitter!\n");
          /* do transmission work */
          thread_start(rcv);
          thread_stop();
     }
}
void receiver() {
     while(TRUE) {
          thread_stop();
          printf("In receiver!\n");
          /* do reception work */
          thread_start(xmit);
     }
}
void startradar() {
     xmit = thread_create(transmitter);
     rcv = thread_create(receiver);
     thread_start(xmit);
     thread_start(rcv);
     /* main thread loops forever now */
```

```
            while(TRUE);
      }
```

Concurrent-Systems-R-Us reports that their software radar "does not work reliably." You already identified the problem last week. Fix it using monitors and Mesa-style condition variables.

5. It's the year 2030. People live longer, there are programmable robots everywhere, but an evil dictator has taken over the country through a coup. The dictator has passed a new draconian law governing the behavior of citizens.

The new law divides the populace into three categories: People who are young (<40), middle-aged (40-90), and old (>90). The law bans everyone from drinking alcohol when young people are present, and it also bans everyone from smoking in the presence of old people. It's acceptable for people of all ages to mingle as long as these two invariants are upheld (e.g. middleaged people who do not want to smoke may happily coexist with old folks). All establishments need to enforce these rules.

You own a restaurant and need to enforce this law on your customers. Since it's very rude to kick existing customers out, you will have to enforce the new law by keeping people out if their entrance would violate the new law. For instance, you will need to keep old people out of your restaurant as long as there are smokers inside. Similarly, young people need to be kept out as long as there are drinkers inside, and middleaged people who want to smoke or drink should not enter the restaurant if there are, respectively, old or young people inside.

Luckily, you own a programmable Bouncer-Robot that can control the entrance to your restaurant. This Bouncer-Robot internally creates a separate process for every new customer it sees. This process invokes a routine called EnterRestaurant(int age, boolean wantToDrink, boolean wantToSmoke) when that customer wants to enter your restaurant. A corresponding ExitRestaurant(int age, boolean wasADrinker, boolean wasASmoker) routine is invoked when the customer is finished dining and leaves your restaurant. Returning from the EnterRestaurant routine implicitly allows the BouncerRobot to let the customer in; consequently, this routine must block until it is safe for the caller to enter your restaurant. Since you need to earn a living, you should not turn away customers or keep them waiting unnecessarily. However, there is no expectation of fairness or bounded-delay from the Bouncer-Robot. And customers, once they declare their intentions to smoke or drink at the door, do not change their minds once they are inside.

Using monitors with condition variables, program your Bouncer-Robot by filling in the code template given below. Your code must use Mesa-style condition variables since that's the only signalling discipline supported by the Bouncer-Robot runtime.

```
Monitor BouncerRobot {
  int EnterRestaurant(int age, boolean wantToDrink, boolean wantToSmoke) {

  }

  int ExitRestaurant(int age, boolean wasADrinker, boolean wasASmoker) {

  }
}
```

6. You have been hired to coordinate the Willy Wonka Chocolate Chip Factory. Your task is to make chocolate chip cookies using equal parts of dough and chocolate chips as ingredients. The simulation is set up such that there are N threads, corresponding to little kids, each of which supply a pound of dough. In addition, there are M threads, corresponding to grownups, each of which supply five pounds of chips. When five pounds of chocolate chips are combined with five pounds of dough, the factory produces ten pounds of chocolate chips. Grownups and the little kids then get to consume the chocolate chip cookies. Combining the ingredients in unequal amounts leads to chips that contain too much chocolate or too much dough, and gets you fired by the quality control folks. Failing to make cookies when there is a sufficient amount of cookies and dough also gets you fired. However, it is fine to allow a late arriving kid or grownup to eat cookies before a grownup or kid who has been waiting for cookies for some time.

The little kids and grownups behave according to the following routines:

```
littlekid() {
    while(true) {
        // get a pound of dough
        doughReady();
        // doughReady should not return until cookies are
        // ready
        eatcookies();
    }
}
grownup() {
    while(true) {
        // get five pounds of chocolate chips
        chipsReady();
        // chipsReady should not return until cookies are
        // ready
        eatcookies();
    }
}
```

N and M, the number of little kids and grownups, respectively, are chosen at random. Your task is to coordinate the factory by writing the code for the two procedures chipsReady() and doughReady().

Synchronize the factory using monitors and Mesa-style condition variables.

Hint:A correct solution will uphold the following conditions at all times:
***Correctness conditions**: (1) there must exist a 1:5 mapping from grownups to kids among the participants eating cookies at all times, and (2) the number of grownups eating cookies must correspond to the batches of cookies made.*
***Progress condition**: if the ingredients for making cookies are available, your code must make cookies and enable the appropriate number of people to eat them.*

Note: A participant is said to be eating cookies as soon as your functions (doughReady() and chipsReady()) have returned.

Note also the lack of a fairness condition, which simplifies your implementation.