1. Solve Problem 7.8, the Barbershop Problem, in the book.

2. Solve Problem 7.9, the Cigarette-Smokers Problem, in the book.

3. You have been hired by the Large-Concurrent-Systems-R-Us Company to review their code. Below is their atomic_swap procedure. It should take two stack data structures as arguments, pop an item from each, and push the items onto the opposite stack. If either stack is empty, the swap should fail and the stacks should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread performing a concurrent swap should not be able to observe that an item has been removed from one stack but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated stacks to happen in parallel. You may assume that stack1 and stack2 never refer to the same stack. Here is their implementation:

```
extern Item *pop(Stack *);     // pops an item from a stack
extern void push(Stack *, Item *); // pushes an item onto a stack
void P(Semaphore *s);  // semaphore wait
void V(Semaphore *s);  // semaphore signal


void atomic_swap(Stack *stack1, Stack *stack2) {
      Item *item1;
      Item *item2; // items being transferred

      P(stack1->lock);
      item1 = pop(stack1);
      if(item1 !=  NULL) {
            P(stack2->lock);
            item2 = pop(stack2);
            if(item2 != NULL) {
                  push(stack2, item1);
                  push(stack1, item2);
                  V(stack2->lock);
                  V(stack1->lock);
            }
      }
}
```

a. Cite three problems with this code.
b. Provide a new implementation that fixes these problems.

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.

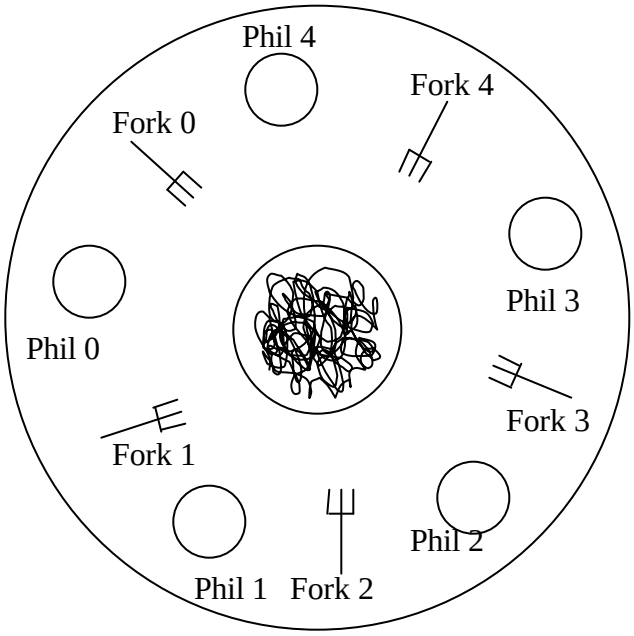
Concurrent-Systems-R-Us has tried to build a software-controlled radar using this system. The radar is controlled by two threads, a transmitter and a receiver thread, which are supposed to alternate (i.e. first the transmitter executes, then the receiver, then the transmitter, then the receiver, and 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." Identify the cause of the problem, describe what behavior the users would see as a result, and fix it using semaphores.

5. Five philosophers are seated around a dining table that has five forks. Each philosopher needs two forks to eat, and alternates between eating and thinking really hard. [Presumably, during the intense thinking periods, they are all wondering why they need *two forks* (no one knows) and whether they can get more utensils delivered (out of the question)]. Each philosopher has a unique id i. Any given philosopher can only reach the forks to her left (Fork i) and right (Fork (i+1) mod 5).



Each philosopher is modeled as a separate thread that performs the following actions:

```
void philosopher(int i) {
      while(true) {
            Think();
            // grab two forks



            Eat();
            // release the forks



      }
}
```

The dining philosophers problem requires that the following correctness properties are upheld:

- A philosopher only eats if he has two forks.
- No two philosophers may hold the same fork simultaneously.
- No deadlock.
- Efficient behavior.

Luckily, the `Think()` and `Eat()` routines are bounded; that is, they return to the caller after a short period of time, so no philosopher will ever take some forks and refuse to give them up. They eventually give up thinking and eat, or become full and stop eating.

Using semaphores, write a deadlock and starvation-free (no pun intended) solution to the dining philosophers problem. You may assume that semaphores have the same properties as the semaphores in your minithreads implementation. You may modify the template shown above for a philosopher to incorporate different behavior.

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 semaphores.

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.