The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 1

September 24, 2009

Figure 1: Traditional dining table arrangement according to Dijkstra.

Exercise 1. The Dining Philosophers problem was invented by E. W. Dijkstra, a concurrency pioneer, to clarify the notions of *deadlock* and *starvation* freedom. Imagine five philosophers who spend their lives just thinking and feasting. They sit around a circular table with five chairs. The table has a big plate of rice. However, there are only five chopsticks (in the original formulation forks) available, as shown in Fig. 1. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he can eat for a while. After a philosopher finishes eating, he puts down the chopsticks and again starts to think.

1. Write a program to simulate the behavior of the philosophers, where each philosopher is a thread and chopsticks are shared objects. Notice that you must prevent a situation where two philosophers hold the same chopstick at the same time.

2. Amend your program so that it never reaches a state where philosophers are deadlocked, that is, it is never the case that each philosopher holds one chopstick and is stuck waiting for another to get the second chopstick.

3. Amend your program so that no philosopher ever starves.

4. Write a program to provide a starvation-free solution for any number of philosophers n.

Solution Figure shows the Fork class. Figure shows a simple, deadlock-prone solution, while Figure shows a solution that avoids deadlock by having even-numbered philosophers pick up the left chopstick first, and odd-numbered philosophers pick up the right one.

```
public class Fork {
  boolean taken;
  int id;
  public Fork(int myID) {
    id = myID;
  }
  public synchronized void get() throws InterruptedException {
    while (taken) {
      wait();
    }
    taken = true;
  }
  public synchronized void put() {
    taken = false;
    notify();
  }
}
```

Figure 2: The Fork class

Exercise 2. For each of the following, state whether it is a safety or liveness property. Identify the bad or good thing of interest.

1. Patrons are served in the order they arrive.
2. What goes up must come down.
3. If two or more processes are waiting to enter their critical sections, at least one succeeds.
4. If an interrupt occurs, then a message is printed within one second.
5. If an interrupt occurs, then a message is printed.
6. The cost of living never decreases.
7. Two things are certain: death and taxes.
8. You can always tell a Harvard man.

Solution Note that students often come up with creative alternative explanations!

1. *Safety:* serving patrons out of order is a bad thing which must never happen.
2. *Liveness:* eventually the coming-down state transition occurs.

```
public class Philosopher extends Thread {
  int id;
  Fork left;
  Fork right;
  public Philosopher(int myID, Fork myLeft, Fork myRight) {
    id = myID;
    left = myLeft;
    right = myRight;
  }
  public void run() {
    Random random = new Random();
    while (true) {
      try {
        sleep(random.nextInt(1000));
        sleep(100);
        System.out.println("Philosopher " + id + " is hungry");
        left.get();
        right.get();
        left.put();
        right.put();
      } catch (InterruptedException ex) {
        return;
      }
    }
  }
```

Figure 3: Deadlock-prone dining philosophers solution

```
public class Philosopher extends Thread {
  int id;
  Fork left;
  Fork right;
  public Philosopher(int myID, Fork myLeft, Fork myRight) {
    id = myID;
    left = myLeft;
    right = myRight;
  }
  public void run() {
    Random random = new Random();
    while (true) {
      try {
        sleep(random.nextInt(1000));
        sleep(100);
        System.out.println("Philosopher " + id + " is hungry");
        if (id % 2 == 0) {
          left.get();
          right.get();
        } else {
          right.get();
          left.get();
        }
        System.out.println("Philosopher " + id + " is eating");
        left.put();
        right.put();
      } catch (InterruptedException ex) {
        return;
      }
    }
  }
 }
}
```

Figure 4: Deadlock-prone dining philosophers solution

3. *Liveness:* eventually one thread that wants in gets in.

4. *Liveness:* eventually the message will be printed.

5. *Safety:* we will not enter the bad state where no message has occurred but an interrupt happened more than a second ago.

6. *Safety:* You never enter the (bad?) state where you pay less to stay alive.

7. *Liveness:* ironically enough. Everyone eventually undergoes these two transitions.

8. *Safety:* You never enter a state where you are in the proximity of an unrecognized Harvard man.

Exercise 3. In the producer–consumer fable, we assumed that Bob can see whether the can on Alice's windowsill is up or down. Design a producer–consumer protocol using cans and strings that works even if Bob cannot see the state of Alice's can (this is how real-world interrupt bits work).

Solution Place cans on windowsills, initially both up. Alice repeats the following:

1. She waits until her can is down.

2. She releases the pets.

3. When the pets return, Alice checks whether they finished the food. If so, she resets her can and then knocks over Bob's can.

Bob repeats the following.

1. He puts food in the yard.

2. He pulls the string and knocks Alice's can down.

3. He waits until his own can is down.

4. He resets his can.

Note that each party must reset his or her own can before knocking over the other's.

Exercise 4. You are one of P recently arrested prisoners. The warden, a deranged computer scientist, makes the following announcement:

> You may meet together today and plan a strategy, but after today you will be in isolated cells and have no communication with one another.
>
> I have set up a "switch room" which contains a light switch, which is either *on* or *off*. The switch is not connected to anything.
>
> Every now and then, I will select one prisoner at random to enter the "switch room." This prisoner may throw the switch (from *on* to *off*, or vice-versa), or may leave the switch unchanged. Nobody else will ever enter this room.
>
> Each prisoner will visit the switch room arbitrarily often. More precisely, for any N, eventually each of you will visit the switch room at least N times.
>
> At any time, any of you may declare: "we have all visited the switch room at least once." If the claim is correct, I will set you free. If the claim is incorrect, I will feed all of you to the crocodiles. Choose wisely!

- Devise a winning strategy when you know that the initial state of the switch
  is *off*.
- Devise a winning strategy when you do not know whether the initial state of the switch is *on* or *off*.

Hint: not all prisoners need to do the same thing.

Solution Designate one prisoner as the *counter*. His count is initially 1.
Here is a protocol when the initial position of the switch is off.
When the counter enters the room:

- If the switch is on, he turns it off, and increments his count. If the count reaches $P - 1$, he announces that all prisoners have been to the room.
- If the switch is off, he does nothing.

When any other prisoner enters the room

- The first time she finds the switch is off, she turns it on.
- Otherwise she does nothing.

Here is a protocol when the initial position of the switch is unknown.
When the counter enters the room:

- If the switch is on, he turns it off, and increments his count. If the count reaches $2P - 2$, he announces that all prisoners have been to the room.

- Otherwise he does nothing.

When any other prisoner enters the room

- The first two times she finds the switch is off, she turns it on.
- Otherwise she does nothing.

Note: as far as we know, this problem is folklore. Readers are invited to send us a proper attribution so we can credit it in the next edition.

Exercise 5. The same warden has a different idea. He orders the prisoners to stand in line, and places red and blue hats on each of their heads. No prisoner knows the color of his own hat, or the color of any hat behind him, but he can see the hats of the prisoners in front. The warden starts at the back of the line and asks each prisoner to guess the color of his own hat. The prisoner can answer only "red" or "blue." If he gives the wrong answer, he is fed to the crocodiles. If he answers correctly, he is freed. Each prisoner can hear the answer of the prisoners behind him, but cannot tell whether that prisoner was correct.

The prisoners are allowed to consult and agree on a strategy beforehand (while the warden listens in) but after being lined up, they cannot communicate any other way besides their answer of "red" or "blue."

Devise a strategy that allows at least $P - 1$ of P prisoners to be freed.

Solution The prisoners agree that the first prisoner will compute the parity of the red hats (even or odd) and announce "red" if the parity is even, and "blue" otherwise. If the next prisoner observes the same parity, it knows its own hat is blue and declares accordingly. If it observes a different parity, its hat is red. All prisoners after the first will be released, while the poor first prisoner has a 50/50 chance.

(This problem is also folklore.)

Exercise 6. Use Amdahl's Law to resolve the following questions:

- Suppose a computer program has a method M that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on an n-processor multiprocessor machine?
- Suppose the method M accounts for 30% of the program's computation time. What should be the speedup of M so that the overall execution time improves by a factor of 2?
- Suppose the method M can be sped up three-fold. What fraction of the overall execution time must M account for in order to double the overall speedup of the program?

Solution

- Assuming the other 60% is perfectly parallelizable, we have $p = 0.6$, yielding a speed up of
$$\frac{1}{1 - 0.6 + \frac{0.6}{n}} = \frac{n}{0.6 + 0.4n}$$
- Since M accounts for 30% of the program's computation time, and assuming everything else is parallelizable, the overall speed-up is:
$$s = \frac{1}{0.3 + \frac{1-0.3}{n}}$$
If we speed up M by a factor k, the new speedup is
$$s' = \frac{1}{\frac{0.3}{k} + \frac{1-\frac{0.3}{k}}{n}}$$
The value of k for which $s' = 2s$ is
$$\frac{6n-6}{3n-13}.$$
- Let M take fraction m of the time. Originally, speedup is
$$s = \frac{1}{m + \frac{1-m}{n}}$$
If we speed up M by 4, the new speedup is:
$$s' = \frac{1}{\frac{m}{3} + \frac{1-\frac{m}{3}}{n}}$$

The value of m for which $s' = 2s$ is $3/(n-1)$.

Exercise 8. You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's Law, explain how you would decide which to buy for a particular application.

Solution Let $F_z(k)$ be the running time of your application using k processors each of which executes z zillion instructions per second. The question asks under what circumstances is
$$F_1(10) > F_5(1).$$
We know that uniprocessor performance is given by:
$$F_1(1) = \frac{F_5(1)}{5}.$$

Let p be the *parallelism* of your application - that is, the fraction of its code that can be executed in parallel. Using Amdahl's law, the *speedup* achieved by the 10 processors machine, $\frac{F_1(10)}{F_1(1)}$, is given by: $\frac{1}{1-p+\frac{p}{10}}$ and therefore:

$$\begin{aligned}
\frac{F_1(10)}{F_1(1)} &= \frac{1}{1-p+\frac{p}{10}} \\
F_1(10) &= F_1(1)\frac{1}{1-p+\frac{p}{10}} \\
F_1(10) &= \frac{F_5(1)}{5}\frac{1}{1-p+\frac{p}{10}} \\
F_1(10) &= \frac{F_5(1)}{5-5p+\frac{p}{2}} \\
F_1(10) &= \frac{F_5(1)}{\frac{10-9p}{2}} \\
F_1(10) &= \frac{2F_5(1)}{10-9p}
\end{aligned}$$

Putting these equations together, we want to solve for p in:

$$\begin{aligned}
\frac{2F_5(1)}{10-9p} &> F_5(1) \\
\frac{2}{10-9p} &> 1 \\
p &> \frac{8}{9}
\end{aligned}$$

It follows that you should use the multiprocessor if and only if p, the parallelism of your application, is at least 8/9.