

# Programming Project 3: Barbers and Gamblers

**Due Date:** Monday, October, 31, 2005, start of class

**Duration:** Two weeks

## **Overview and Goal**

In this project you will gain additional familiarity writing programs involving concurrency control. You will implement a couple of classical IPC problems to become more comfortable with Semaphores, Mutexes, Condition Variables, and the Monitor approach.

## **Download New Files**

Set up a new directory for this project called **cs333/p3** and download all the files from

```
~walpole/public_html/class/cs333/fall2005/spank/p3
```

Even though some of the files have the same names, be sure to get new copies for each project. In general some files may be modified.

Please keep your old files from previous projects separate and don't modify them once you submit them. This is a good rule for all programming projects in all classes. If there is ever any question about whether code was completed on time, we can always go back and look at the Unix "file modification date" information.

For this project, you should get the following files:

```
makefile
DISK
System.h
System.c
Runtime.s
Switch.s
List.h
List.c
Thread.h
Thread.c
Synch.h
Synch.c
Main.h
Main.c
```

The file **Synch.c** contains my solution for project 2. You may use either my version of this file, or the version of **Synch.c** you created.

I have also included the file

**Proj2Sol.pdf**

which contains my solution code for Producer-Consumer and Dining Philosophers. If you had difficulty with it, you can take a look at the code I wrote.

You will modify and submit the following files:

**Main.h**

**Main.c**

### **Task 1: Implement the Sleeping Barber Problem**

The textbook gives a solution to the “Sleeping Barber” problem in C on page 131; translate this into a working Spankish program.

You’ll also need to create some code to print out what happens when you run the program.

What will you do for the “cut\_hair” and the “get\_haircut” methods? You’ll need at least one print statement in each routine, but be careful: they both should run at more-or-less the same time.

One barber is okay, but how many customers will you model? Will you just throw a bunch of customers at the barbershop all at once? This might not test your code very well.

How long will the haircut last? You can implement waiting with a busy loop, such as

```
for i = 1 to 100
  .. want to yield here?...
endFor
```

The goals are:

- (1) Practice creating a Spankish class from scratch
- (2) Practice testing your code
- (3) Gain experience with semaphores, mutex locks, and thread synchronization

### **Task 2: The Gaming Parlor Problem**

Here is the problem scenario: groups of customers come in to a “gaming parlor” to play games. They go to the front desk to obtain one or more dice, which are used by the group while they are playing their

game, and then returned to the front desk. The front desk is in charge of lending out the dice and collecting them after each game is finished.

The gaming parlor owns only 8 dice, which are available at the front desk before the first group comes in.

The customers can play the following games. After each game, the number of dice required to play that game is listed in parentheses.

Backgammon (4)  
 Risk (5)  
 Monopoly (2)  
 Pictionary (1)

Model the front desk as a monitor with the following entry methods:

**Request** (numberOfDice: int)  
**Return** (numberOfDice: int)

Model each group of customers as a thread. When a group is ready to play, they must obtain the necessary number of dice. If the required number of dice is not available, then the group (i.e., the thread) must wait. You might use a condition variable that “more dice have become available.”

Model the following eight different groups. Each group plays one game, as shown below, but they play the game 5 times. They must return their dice after each game and then re-acquire the dice before playing again.

A – Backgammon  
 B – Backgammon  
 C – Risk  
 D – Risk  
 E – Monopoly  
 F – Monopoly  
 G – Pictionary  
 H – Pictionary

To simulate playing the game, simply call the **Yield** method.

This problem is a generalization of the problem of resource allocation where (1) there are a number of resources (dice) but each is identical; (2) every requesting process needs a one or more units of the resource, (3) each requesting thread knows how many units it will need before requesting any units and that info is included in the request, (4) all units are returned before any further requests are made.

In a monitor, each method is either an “entry” method or a local method. There is one Mutex lock associated with the monitor and zero or more condition variables. An entry method must always begin by locking the monitor’s mutex. Before it returns, it must always unlock the mutex. Within the entry method, the code may Signal (or Broadcast) to the condition variables and may Wait on condition

variables. However, within an entry method, the code should not touch the monitor's mutex lock, or look inside the condition variables.

Hint: Model the front desk as a monitor. Use condition variables instead of semaphores.

If deadlock occurs, the program will freeze up and some requests for dice will go unsatisfied forever. Regardless of the order in which the groups make their requests, your solution should be structured such that deadlock can never occur.

Your solution must not be subject to any race conditions. In other words, regardless of the order in which the groups make their requests and return their dice, each die must never be allocated to more than one group at a time. It should never be the case that groups are allowed to proceed when there are too few dice. Likewise, if a group has returned its dice, other groups which are waiting must be allowed to proceed once enough dice have become available.

Starvation can occur if it is possible that one thread can be delayed infinitely by other threads' requests. If the game parlor problem is extended to assume that each group will continue to play game after game forever, then starvation might be possible, if you are not careful in your solution. If some group X comes in requesting a lot of dice, it will be made to wait until enough dice are available. If it is possible that other groups can come in, request a small number of dice, and have their requests granted, then group X might get delayed forever, since there are never enough dice at the front desk at once to satisfy group X's needs. In other words, it might be possible for starvation of X to occur. In your solution starvation should not be possible. (Of course, with each group limited to playing only 5 games, all the outstanding dice will always get returned eventually and starvation can never occur, but your solution should also avoid starvation in the presence of a never-ending sequence of **Request** and **Return** operations.)

To verify that your code is working, please insert print statements to produce output like this...

```

Initializing Thread Scheduler...
Initializing Idle Process...
A requests 4
-----Number of dice now avail = 8
A proceeds with 4
-----Number of dice now avail = 4
B requests 4
-----Number of dice now avail = 4
B proceeds with 4
-----Number of dice now avail = 0
D requests 5
-----Number of dice now avail = 0
E requests 2
-----Number of dice now avail = 0
A releases and adds back 4
-----Number of dice now avail = 4
B releases and adds back 4
-----Number of dice now avail = 8
C requests 5

```

```

-----Number of dice now avail = 8
H requests 1
-----Number of dice now avail = 8
B requests 4
-----Number of dice now avail = 8
D proceeds with 5
-----Number of dice now avail = 3
...etc...

```

This output makes it fairly easy to see what the program is doing and verify that it is correct.

Below is the method I used to produce this output. Feel free to include this method in your program.

```

behavior GameParlor

  ... Other methods...

  method Print (str: String, count: int)
    --
    -- This method prints the current thread's name and the arguments.
    -- It also prints the current number of dice available.
    --
    print (currentThread.name)
    print (" ")
    print (str)
    print (" ")
    printInt (count)
    nl ()
    print ("-----Number of dice now avail = ")
    printInt (numberDiceAvail)
    nl ()
  endMethod

endBehavior

```

This method is called in several places...

At the beginning of my **Request** method:

```
self.Print ("requests", numNeeded)
```

At the end of my **Request** method:

```
self.Print ("proceeds with", numNeeded)
```

In my **Return** method:

```
self.Print ("releases and adds back", numReturned)
```

## What to Hand In

Please submit hardcopy of the following files:

**Main.h**  
**Main.c**

Also hand in hardcopy showing your output for both tasks.

For the Sleeping Barber problem, I am not providing any testing material; you'll have to devise some tests on your own. Please hand in something showing how you have tested your code.

In LARGE BLOCK LETTERS, write your full name, as it appears in the PSU registration system. In parentheses, list your nickname, if any.

PLEASE STAPLE ALL PAGES THAT YOU HAND IN. This applies to homeworks, too.

### **Regarding The Due Date and Time**

Please bring your project submission to class. It will be due at the beginning of class. I will collect project submissions before lecture and *I will not accept project submissions after I begin class.* I do not want students to miss lectures because they are trying to finish up a project. Plan ahead. The due time is "hard." I will make extensions to deadlines in the case of medical emergencies and business travel if you contact me ahead of time.

### **Working Together**

I encourage students to discuss the material in this class. Feel free to discuss the programming projects amongst yourselves.

However, **this is not a group project.** *You must create the code on your own.*

It is okay to look at someone else's code for the purposes of helping that person or for comparing your different approaches, but each student must do the coding alone. You must type in every line of code you submit. You must **not** copy another student's code. Do not look at some one else's code, then turn around and type that code in. You must create your code on your own.

It will be considered cheating to copy code.

### **Questions/Comments via Email**

Questions/comments may be directed to the mailing list, TA, or to me:

cs333-001@cecs.pdx.edu ← *Email to the class mailing list*

If you email questions to me or the TA, please include “cs333” in the subject line so we don’t accidentally identify your email as junk! (This is not necessary with the mailing list, since MailMan automatically adds [cs333].)