The Dining Philosophers Problem. For this lab, we will write a multi-threaded program, in either C or Java, that implements the well-known Dining Philosophers problem. This particular problem is often used to demonstrate the use of mutual exclusion algorithms and deadlock prevention in multi-threaded applications. The scenario is that there are some number \( N \) of oriental philosophers sitting around a round table (for our implementation we will use \( N = 5 \)). Each philosopher has one chopstick at his right hand and one chopstick at his left hand. However, the right hand chopstick for philosopher \( n \) is the left hand chopstick for philosopher \( (n + 1) \mod N \). In other words, there are \( N \) chopsticks for \( N \) philosophers (one between each pair of philosophers).

As the philosophers are sitting around the table, each enters a state of deep thinking for a random amount of time. Eventually, each philosopher becomes hungry and picks up the chopstick at his right hand. If the right-hand chopstick is not available, he simply waits until it becomes available. He then picks up the left hand chopstick, again waiting if it is not available. Once he has two chopsticks, he begins eating for a random amount of time. When finished eating, he puts down the chopsticks one at the time, freeing them for use by a neighboring philosopher (ignoring for now any possible hygiene problems with this approach). If you think through this scenario, you should be able to see the potential for a deadlock, where each philosopher has a single chopstick and is waiting infinitely for the second one.

The C Implementation. For the C implementation, you are given four files as a starting point. `dining.c` is an empty C main program that you will edit to include your implementation. `miscsubs.c` is a set of subroutines provided by the instructor that will assist you in determining if your implementation is correct, and that will be used for grading. `miscsubs.h` is a C header file with function prototypes for each of the subroutines in `miscsubs.c`. Finally, the `Makefile` is used to compile and link your program. All of these files are included in the `dining-c.tar.gz` file posted on the web page.

The program should compile and execute on any Linux system (0311ece???.ece-int.gatech.edu is available), the Windows systems in the ECE labs (they have a PThreads library), or any Sparc Solaris system (such as yamsrv1.ece.gatech). The thread scheduler seems a bit better on the Solaris systems, leading to better fairness. After connecting to the Linux or Solaris system you should un–zip and un–tar the posted file as follows:

```bash
gunzip -c dining-c.tar.gz | tar -xvf -
```

This should create a directory called `DiningC`. Next change to that directory and build and execute the skeleton program as follows:

```bash
cd DiningC
make
./dining
```

If this was successful, the output will be:

```
Starting the Dining Philosophers Simulation
EatCount 0 - 0
EatCount 1 - 0
EatCount 2 - 0
EatCount 3 - 0
EatCount 4 - 0
```
Implementation Details. Your implementation should use one thread to model the behavior of each philosopher. Each thread should be assigned a unique integer philosopher identifier, in the range 0...\(N - 1\). The main program should create \(N\) threads, and should insure that each thread has begun processing before creating the next thread (use a Condition Variable for this). After all threads have been created, your main program must not just burn CPU time waiting for the philosophers (threads) to exit. Instead, you must use a condition variable which the philosophers use to inform the main program they are exiting. You should also "desk check" your code carefully to be sure that you don't have a possible deadlock situation, since just running the program for a while with no problems does not mean there are not potential problems. In my solution, I intentionally introduced a potential deadlock, but yet the program still ran to completion, just by good luck.

Each philosopher thread should loop in a thinking, hungry, eating cycle until a fixed number of eating states have been processed (we will use 500). Each chopstick should be modeled using a Mutex. The mutex is locked when it is in the hand of a philosopher, and unlocked when sitting on the table.

There are four subroutines in miscsubs.c that you should use to instrument your program. RandomDelay is called to have the philosopher do nothing for a random amount of time (both in thinking state and eating state). When a philosopher starts eating, the StartEating subroutine should be called. When a philosopher stops eating, the DoneEating subroutine should be called. Both StartEating and DoneEating have a single integer parameter, indicating which philosopher the action is for (in the range 0...\(N - 1\)). The StartEating subroutine maintains a global variable, TotalEats, that each thread can use to determine when to exit. Also defined is a global constant MAX_EATS, which is set to 500. When all threads have exited, the LogResults subroutine should be called.

Expected Results. Obviously, the program should not deadlock and should run to completion. Further, there should be some degree of fairness among the threads. In other words, each philosopher should eat approximately the same number of times. For our simulation, we have 5 philosophers and MAX_EATS of 500, so each philosopher should be able to eat about 100 times. Due to a number of thread scheduling issues, this could vary somewhat, but in no case should one or more philosophers starve.

Java Implementation. The file dining-java.tar.gz posted on the web page contains the Java skeleton dining.java and miscsubs.java. There is no Makefile in the Java implementation, so you must compile the class files individually with javac. After the un–zip and un–tar process described above, the directory DiningJava will contain the skeleton classes. Use Java threads for each philosopher, just like in the C implementation. Use java synchronize to lock the chopsticks, similar to the C Mutex approach. Use the Java wait and notify functions to implement the condition variables. The Java miscsubs.java class contains the same subroutines as miscsubs.c, and should be used in the same fashion. The expected results are the same as the C results.