Name: Jonah Kubath
Class: CS5541
Assignment: A5 Dining Philosopher with threads
Summary: This is the report for my implementation of the dining philosopher problem using threads. I will also answer a few questions about how I solved the typical issues of using threads with shared memory.

Implementation:

The general code outline for this problem is this:

1. Philosophers sit at the table
2. Philosophers attempt to pick up two forks to eat
3. Philosophers will continually attempt to eat until their food is gone
4. Once all philosophers have eaten their food, they leave.

I used pthreads_t to act as the different philosophers.

Main: The main function of the program is fairly short and simple. It starts by creating all the threads. The threads are joined together so the main program does not exit before the threads finish executing.

When the threads are created, the inThread() function is called.

inThread: The inThread function takes in the philosopher id so the different people can be distinguished. This function handles the bulk of the operations. There are 3 data structures that are global (shared) for all the threads to access and modify.

1. int forks[] represents the availability of each fork
2. int phils[] represents the status of each philosopher
3. sem_t *change semaphore used to lock when changes to forks[] is being queried or changed

The philosophers will start by checking if the fork in front of them and the fork next to them is available. While this action is taking place the semaphore is locked to ensure no other thread is making changes or using the same data. If the forks are ready, then the philosopher will call the eat() function. If the forks are not ready, then the philosopher will acknowledge they have been skipped this turn and wait for the resources to open. This wait period is where starvation can creep into the program. I handle this by keeping track of how many times a philosopher "skips" eating. Once this hits a threshold, the philosopher will lock the forks and wait for the needed forks to become available before anyone else can change their status. The forks will become available when the eating time slice has finished for the philosopher next to the current and then the program will continue as normal.

eat: The eat() function handles the time slice allocated for each philosopher to use when forks are available. The status of the forks are changed to UP so no other philosopher can use them. The philosopher will then eat. When the philosopher is done eating the forks are changed to DOWN for others to use.

Extra Questions:

1. What method did you use to ensure that one version of your program was susceptible to starvation but not deadlock?

I took out of my implementation the counter for how many times the philosophers are "skipped". To make it even more obvious when a philosopher is skipped, a sleep() timer is set off. This causes the philosopher to continue to be skipped and the others to eat. The other philosophers will notice that no one is trying to use their forks. When they are done eating, they will pick back up the forks and eat again causing the skipped philosopher to starve.

2. What method did you use to ensure that one version of your program was susceptible to deadlock but not starvation?

I allowed the philosophers to pick up any fork that is available rather than forcing them to pick up both forks at the same time. I ensured the deadlock by sleeping after the first fork is picked up. This will allow the others philosophers time to pick up their first fork causing deadlock. Each philosopher is holding one fork and never puts it down.

3. What method did you use to ensure that one version of your program was not susceptible to starvation OR deadlock?

Starving: The philosophers will keep track of how many times they have skipped eating because of unavailable resources. Once this hits a certain threshold, the forks will be locked until the current philosopher can take the forks needed. After this happens, the resources are freed and the other philosophers can continue as normal

Deadlock: I force the philosophers to pick up both forks at the same time. I also lock the semaphore while checking the availability of the forks so that no other philosopher uses the shared data at the same time. Without locking the data, a philosopher may read that a fork is DOWN, when in reality a thread has picked it up it just has not changed the status at this point.