CS 4414 Lab 5: Programming with Pthreads and Semaphores

Submitted by Bryan Walsh (bpw7xx) and Jared Culp (jjc4fb)

Problem 1 - Dining Philosophers

Our Dining Philosophers solution can be compiled regularly using “make philosophers”.

Algorithm Implementation

We used the provided naive solution as a starting point and did most of our implementation in the existing “philosopher_thread” function. Initially we thought that each chopstick would need a condition variable (to signal whether it was being used) and a mutex (to ensure mutual exclusion); however, this proved to be incorrect. By having these synchronization primitives on each chopstick, we would need to signal *two events* to allow a philosopher to eat. It turns out there is no easy way to do this, so we decided to have a **pthread_cont_t** and **pthread_mutex_t** inside the **philosopher_t** struct.

We decided that a more appropriate way to think about whether or not a philosopher can eat is whether or not both of his neighbors are in the “eating” state. In other words, when a philosopher state changes from hungry to eating, a philosopher must ensure that his neighbors do not eat. On the other hand, when a philosopher state changes from eating to thinking, a philosopher is done eating and must signal any waiting philosophers that are hungry.

In philosopher_thread, we first attempt to acquire the mutex that the particular philosopher holds, which ensure that preemption does not occur during the critical section. we then have a while loop that checks if left_phil and right_phil (neighbors) are not both in the eating state. Inside the loop we call condition_wait, which puts the thread to sleep in a nonblocking state. We use a while loop instead of an if statement so that the philosopher performs a second check when it is woken up to ensure that both of his neighbors are not eating. When the philosopher’s condition variable is woken up, it is automatically guaranteed the lock (as per pthreads). We then pick up both chopsticks, eat, and put both chopsticks back down. When we update the state to thinking, we then check if the left_phil’s left_phil is not currently eating (the other neighbor to the current philosopher’s left neighbor) and if the left_phil is currently hungry. We perform similar logic with the right_phil’s right_phil neighbor. If either of these conditions are true, we signal the appropriate condition variables for the left and/or right philosophers. We then release the mutex held by the philosopher who just ate and go back to sleep.

Deadlocks and Starvation

This solution is deadlock and starvation free. In order to prevent deadlocks, one of the following 4 conditions must be removed:

1. Mutual exclusion

2. Hold and wait
3. No preemption
4. Circular wait

We remove the hold and wait condition, because a philosopher only picks up *both* chopsticks at a time. We ensure that both are available using condition variables. A philosopher never picks up just one chopstick and waits for the other one to be available. As a result our solution is deadlock free.

Our solution is also starvation free because a chopstick is only ever contested amongst two philosophers. It is not possible for a deadlock to occur (as discussed above) due to circular wait. Thus, eventually a chopstick will become available, and the philosopher that has been waiting the longest (per its condition variable) is awoken and given priority. There is no preemption in the sense that a new philosopher can be placed at the head of the queue. The condition variable queue provided is a FIFO queue. These assumptions can be deemed to be very likely true, as many test cases were run using the philosophers_stats program, and the average times for each philosopher to be hungry, eat, and think were evenly distributed.

Problem 2 - Matchmaker

Our Matchmaker solution can be compiled regularly using “make whales”.

Algorithm Implementation

For our implementation of the male() and female() function, each makes a call to the same function, process_whale(int is_male, int id), which is responsible for checking whether or not a matchmaker is currently needed. If a matchmaker is needed then the created whale becomes a matchmaker and keeps matchmaking until there are no available matches to be made. At that point, when the whale is no longer a matchmaker, or if a matchmaker was not needed in the first place, the whale is placed its appropriate queue of same-gendered whales waiting for mates.

Deadlocks and Starvation

Our implementation is deadlock free because there is never a time when two whales of the same gender can ever hold the lock at the same time. Whales trying to acquire locks for their appropriate sex (only 1 lock for each gender is available, meaning that the whale holding the lock is the next male to be mated of that gender) request the lock, and if the lock is available then they are granted it. If the lock is currently held (meaning a whale of that gender is already waiting to mate) then the whale is forced to wait in a FIFO queue of whales of the same gender that are awaiting mates.

Our implementation is also starvation free because we attend to whales in the queue on a first in, first out basis. As soon as a mate is available, the head member of the queue of each gender is selected to mate, each of which has been waiting the longest for the lock.