Daniel Perry
CS5460 / Fall 2005
Synchronization and Deadlock Homework
19 October 2005

1) Proving Mutual Exclusion

To show that the baker's algorithm is a correct implementation of the N-thread critical section problem, we will consider the three requirements: Mutual Exclusion, Progress, and Bounded Waiting.

1. Mutual Exclusion

   a. *One process is in critical section, another process tries to enter: show that the second process will block in entry code*

   If Pk is in the critical section, it means it will have compared its number[k] (or pid) with every other number[j], so when Pi enters the entry code, and chooses a number[i], it will be greater than number[i].

   If Pi already chose a number[i], number[k] will still be greater than, because otherwise Pk would not have entered the critical section (it would sill be "spinning" on the while ((number[i] < number[k]) ...)).

   b. *Two (or more) processes are in entry code: show that at most one will enter critical section*

   If, two (same argument for more than 2) processes Pk and Pi are in the entry code at the same time.

   The worst case scenario would be that they both choose the same number for choose[i] and choose[k]. This would be resolved in the final while() statement where the process with the smaller unique process id would enter the critical section, and the process with the greater process id would stay in the while loop, until the first set it's number[] = 0.

2. Progress

   a. *No process in critical section, P1 arrives: show that P1 enters*

   If there are no processes in the critical section and process P1 arrives, it will choose the next number[1]. It will then check this number[1] against all other number[n]'s. At some point in time it will become the lowest number (as no processes are in the critical section, it is assumed this process will probably have the lowest), and will pass the for and while loops, and enter the critical section.

   b. *Two (or more) processes are in the entry code: show that at least one will enter critical section.*

   Let processes P1...Pn enter the entry code, let us consider the first of those processes, Pk, that enters the entry code before all other processes, and there is no process in the critical section, it will receive the smallest number[k] assignment, and will then be able to progress through the for and while loops to enter the critical section – because it will have the smallest number[k].

   In the special case that another process Pi enters the entry code at the same time as Pk and has the same number[] assignment, at least one (and actually only one) process will still be able to enter the critical section.

3. Bounded Waiting

   a. *One process in critical section, another process is waiting to enter: show that if first process exits the critical section and attempts to re-enter, the waiting process will get in first.*

   Assuming some process Pk is in the critical section, and another process Pi enters the entry code, assuming fair scheduling of processes, the process Pi will receive a number[i], the max of all assigned number[n]'s. At anytime after this, if process Pk exits the critical section, and

enters the entry code, it will receive a number[k] higher than number[i]. This will guarantee that Pi will have an opportunity to enter the critical section.
Even if Pi and Pk were assigned the same number[] values, and Pk had a lower pid, Pk may beat Pi again, but eventually Pi will have an opportunity to enter the critical section, assuming the scheduler gives process Pi time to progress through the while and for loops.

2) Semaphores and Condition Variables

There is a problem with mutual exclusion, because there is a race condition. Consider the following scenario:

There are 2 utes waiting (utes_waiting = 2). Another ute arrives and calls UteArrives(), he gets the mutex, and enters the first if() statement, sends the 2 ute signals, calls RideTrax, and releases the mutex, and is context switched.
Now, two different processes are swapped in one after another that call UteArrives() and beat the previously waiting ute processes to the mutex. These new processes increment utes_waiting to 4 (each add one).
Now, before the waiting utes can context switch in and decrement, a third ute arrives, calls UteArrives() and beat the first two again, incrementing utes_waiting to 5, and waits.
Now the count is all messed up. It will just keep going up and up as utes arrive, until the game is over, no one else made it to the game, and they all go home crying.
Because even if at this point the original two utes were context switched in and got the mutex, and decremented their count, the utes_waiting would still be 3, which will never be caught in an if statement. A similar condition could occur with different permutations of utes_waiting and cougars_waiting, but the same basic problem.

One solution to this problem would be to do the decrementing in the successful if statement before letting the mutex go. For example, after each "signal(utes);" there was a "utes_waiting--;".
In the above scenario, if the first successful if() statement, (where it tested true for utes_waiting == 2) would have decremented utes_waiting by two, before calling V(mutex), it would have solved the problem (you would of course then NOT decrement it in the third else clause as it is currently implemented). This would solve that problem.

3) Deadlock (Book problem 7.2)

Deadlock occurs in the dinning philosopher's problem, because it fulfills each of the four necessary conditions of deadlock:

1. Mutual Exclusion. The resource (chopstick) cannot be shared, one chopstick can only be used by one philosopher at a time – in other words, two or more philosophers cannot share a chopstick – it would be pretty awkward, especially at the dinner table.
2. Hold and Wait. In the case where each philosopher grabs one chopstick to begin (let's say for this exercise they first grab the chopstick to the right), then to complete their task of eating they need another chopstick. Since all other chopsticks are being held by the other philosophers,

they must wait. In other words, each is holding a resource and waiting for another resource to be freed.

3. No Preemption. Once a philosopher gets his chopstick he will not give it up – they are far too stubborn. In other words, once a resource (chopstick) is acquired, it will not be freed until the philosopher is done eating.
4. Circular wait. Because they are seated around a table, and each grabs one chopstick (to the right), they must each wait for the chopstick to their left. This creates a circular wait, because, say they are numbers P1, P2, ..., PN, and the chopsticks, C1, C2, ..., CN. When each grabs a chopstick, say P1 grabs C1, P2 C2, ... , and PN CN, then each must wait for the chopstick to the left – P1 waits for CN, P2 for C1, ..., PN for C(N-1). This is a circular wait – one must give up a fork for the wait to be broken.

Deadlocking in this situation could be avoided by eliminating any one of the conditions stated above, they are considered here:

1. Mutual Exclusion. If the resources (chopsticks) were not mutually exclusive – or if they chopsticks could be shared among the philosophers, the deadlock would not occur. In our example, each PK would grab their corresponding CK, and then instead of waiting for the chopstick to the left, simply shared that chopstick with the philosopher to the left, there would be no deadlock – theoretically they could all eat – although realistically it would be pretty funny to watch N philosophers try sharing chopsticks to eat.
2. Hold and Wait. Somewhat related to circular wait, if not all of the philosophers held a resource and was waiting for another to free up, the deadlock would not occur. As a solution, if atomically each philosopher could request both required chopsticks at once, they would not be holding a resource and waiting for another – they would have both needed chopsticks. So, in our example, if you tied two chopsticks together, and you had to grab a pack of two chopsticks to eat, it would solve the problem.
3. No Preemption. Although it may take a while to convince a philosopher to give up his chopstick before he was done eating, if it could be done, it would solve the problem. In other words, if the philosophers could be preempted to give up their resource (chopstick) so that the philosopher next to them could eat, it would end/prevent the deadlock. But good luck convincing them.
4. Circular wait. If somehow you could eliminate the circular wait, by allowing one philosopher to eat, the deadlock would end. You could do this by using the same method mentioned under "Hold and Wait" - making the allocation of two chopsticks atomic, so that when requesting resources they got all the resources required to eat. Allowing preemption or eliminating mutual exclusion would also kill the circular wait.

4) Deadlock-free synchronization (Book problem 7.14).

In this situation, the critical section of the farmers trip is the bridge. The bridge is the resource that should be mutually exclusive, so that only one farmer is on the bridge at any one time. One method to ensure this would be to establish a variable to indicate which direction the bridge was functioning at that time, lets call it direction_var, a counter for how many cars are on the bridge, lets call it

Daniel Perry
CS5460 / Fall 2005
Synchronization and Deadlock Homework
19 October 2005

counter_var, and a mutex for the both those variables, called bridge_mutex.
Whenever a farmer wanted to use the bridge he encounters three situations:

1. direction_var is his direction. He would simply increase the counter, cross the bridge, and decrease the counter when he was done.
2. direction_var is not in his direction and counter_var = 0. He would change the direction_var, increase the counter_var, cross the bridge, and decrease the counter_var.
3. direction_var is not in his direction and counter_var > 0. He would wait until he arrives at situation 2 or 1.

The pseudo-code would like this:

```
bool EnterBridge(){
        P( bridge_mutex );
        if( direction_var == my_direction ){
                counter_var++;
                V( bridge_mutex );
                return false; // go cross bridge
        }else{
                if( counter == 0 ){
                        direction_var = my_direction;
                        counter_var++;
                        V( bridge_mutex );
                        return false; // go cross bridge
                }else{
                        V( bridge_mutex );
                        return true; // don't cross bridge yet.
                }
        }
}
void ExitBridge(){
        P( bridge_mutex );
        counter_var--;
        V( bridge_mutex );
}

main(){
        while( EnterBridge );
        CrossBridge();
        ExitBridge();
}
```

This solution does have the characteristic of starving one side or the other, because one a side gets the direction_var, it will be difficult to get it back.

Daniel Perry
CS5460 / Fall 2005
Synchronization and Deadlock Homework
19 October 2005

5) Starvation-free synchronization (Book Problem 7.15)

To solve this problem we use the above solution with one modification, a preferred_direction_var on which we will test when we want to enter the bridge, and which will be switched when another counter, counter_var_total, which counts the total number of crossings, reaches a limit

Here is the pseudo-code (only EnterBridge changes):

```
bool EnterBridge(){
        P( bridge_mutex );
        if( direction_var == my_direction && preferred_direction_var == my_direction ){
                counter_var++;
                counter_var_total++;
                if( counter_var_total > limit ){
                        preffered_direction_var = other_direction;
                        counter_var_total = 0;
                }
                V( bridge_mutex );
                return false; // go cross bridge
        }else{
                if( counter == 0 ){
                        direction_var = my_direction;
                        counter_var++;
                        V( bridge_mutex );
                        return false; // go cross bridge
                }else{
                        V( bridge_mutex );
                        return true; // don't cross bridge yet.
                }
        }
}
```

This modification has the effect of stopping one side from entering once the total crossings has reached a specified limit. Then the other side will gain control and cross. Of course if either lets their counter_var go to zero (no more trucks crossing from that side) it can switch as well.