**Problems with the Dining Philosopher synchronization:**

The threads (philosophers) do not have unlimited resources. They only have two chopsticks (one on the left side, and one on the right side) which are being shared by the philosophers sitting on the left side and on the right side of them. An issue with this is if philosopher C has a chopstick in his right hand and the philosopher B has a chopstick in his right hand and philosopher has a chopstick in his right hand, so on and so forth all around the table and there are no conditions to deal with this, this creates deadlock where no philosophers (processes) go anywhere.

Another problem that is stated helps prevent deadlock, but may cause resource starvation. This occurs if there is a rule where a certain time rule is put into place where if a chopstick is picked up and after this time is exceeded the philosopher puts down the chopstick. If all five philosophers arrive at the table at the same time and each picks up the left chopstick at exactly the same time they will wait until the set amount of time, then set the chopstick down and then wait another set amount of time until the pick the chopstick up again, etc….

**Summary of one solution to the Dining Philosopher:**

The first Philosopher to pick up the chopsticks goes through a test routine that checks to make sure that the state of the current Philosopher is hungry. If so it checks also to make sure that the left and right chopsticks are available. If this is also true, the Philosopher’s state is changed to eating. Once the Philosopher is done eating it goes through the flow to put down the chopsticks. Here the Philosopher checks the Philosopher to the left to make sure that he can pick up the chopsticks and eat. This works the same for the Philosopher to the right as well. This section allows the other Philosophers to not undergo Resource Starvation and also prevents Deadlock since the Philosophers handoff the chopsticks.

Mutex semaphores are used to avoid race conditions when the Philosopher threads go into the hungry state and when they pick up chopsticks. It is also used when the threads put the chopsticks down (which is also when the Philosophers state is changed to Thinking). Again, they are used to avoid race conditions to avoid inconsistencies in the data.

One more semaphore is used to regulate the timing and order when the philosophers put down chopsticks and.

**Problems with the Bounded Buffer synchronization:**

There is a buffer of a set size and only a set amount of data can be fit into it. So this has to be dealt with to not allow any producer threads to add data into the buffer when the buffer is full, and also to not allow any data to be removed from the buffer if it is empty.

Another problem to overcome is no two threads can be in the “critical section” of the code which is the section of code that adds to or removes from the buffer as well as the section of code that increments the variable that keeps track of how many items are in the buffer.

**Summary of one solution to the Bounded Buffer:**

Semaphores can be used to provide a mutex (mutual exclusion) lock for the critical section of code where data is either inserted or removed from the bounded buffer. You just say essentially wait(mutex) before the function call and signal(mutex) after the function call to safe guard this section of code.

The way the semaphores are used in the producer and consumer thread are complete opposites essentially.

Let me explain how the semaphores are initialized first off. There is a semaphore that is initialized to the size of the buffer which ill call emptyCount, and there is another semaphore that is initialized to 1 which ill call fullCount.

In the consumer thread the fillCount semaphore is sent a wait message (which decreases the semaphore integer and possibly puts the thread in a wait queue) and after the item is removed from the buffer a signal message is sent to the emptyCount semaphore. These semaphores keep track of the amount of items in the buffer.

In the producer thread the emptyCount semaphore is sent a wait (which decreases the semaphore number and possibly puts the thread in a wait queue if necessary).

The order of these function calls are important since the emptyCount semaphore keeps tracking of the number of empty “slots” in the buffer and the fillCount semaphore keeps track of the “slots” that are full in the buffer. For instance, we would want to decrease the integer in the emptyCount semaphore if there is an item produced since this semaphore keeps track of how many items in the buffer are empty and in the section of code if an item is produced there would technically be less empty slots. Then the fillCount semaphore would need to be increased.

For the same reasons, these roles are flipped in the consumer section of code.