**Group Member (CLID)**

|  |  |
| --- | --- |
| **Bradley Milliman (bjm6183)** | **Travis Aucoin (tna9502)** |
| **Marcus Amos (mja0161)** | **Xiyue Xiang (xxx1698)** |

**All questions have been arranged by their associated task.**

**Task 1**

*When implementing and testing your solution, did you notice any deadlock? How did you solve any deadlock problems?*

Yes, I did encounter deadlock, it was resolved by dropping the left chopstick if the right one is not available. This way another philosopher would not be stuck because another philosopher is using up the resource without accomplishing anything.

*Make the philosophers yield between attempting to pick up the left and right chopsticks. Run at least 3 test with various parameters and –rs seeds. Record your findings and explain your results. Undo any changes made to accommodate this question before submitting your assignment.*

My findings where interesting, basically what looks to be happening is a game of leap frog. After one philosopher picks up the left chopstick it is being skipped over by the one behind it. This makes it so that everyone picks up there left chopstick and would deadlock in some situations if I had not put in an abort so that after five tries of picking up the right chopstick the philosopher gives up and drops his left chopstick, which allows another philosopher the tools he needs to eat and solve the problems of the world!

*In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them?*

Implementation can be seen by the algorithm provide below. Yes I did encounter bugs they were mostly resolved by dropping the left chopstick if the right one is not available. This way another philosopher would not be stuck because another philosopher is using up the resource without accomplishing anything.

*Data structure and algorithm used:*

I used an integer array to hold a 1 or a 0 for each chopstick at the table. If the chopstick had a value of 0 it was being used and if the chopstick had a value of 1 it was available for use. Pickup Order: Philosopher will try to pickup the left chopstick first. If succeed, he will then try to reach the right one. Abort: Whenever failing for 5 times, he will abort pickup. If he can't even pickup the left one, he simply abort pickup. If he has already picked up the left one, but fails to pickup the right one, he has to abort the pickup attempt and then drop the left one which has already being picked up.

**Task2**

//Philosopher joins the room

//Philosopher waits to sit

while (not all philosophers in room)

currentThread->Yield();

if (all philosophers in room )

{

//All philosiphers sit down at table

}

while (there are meals to be eaten)

{

//Philosopher try to pick up left chopstick

if (left chopstick is there)

{

//Philosopher has picked up left chopstick

//Philosopher try to pick up right chopstick

if (right chopstick is there)

//Philosopher has picked up right chopstick

else

{

//Abort Pickup after five tries

//Philosopher drops right chopstick

}

}

else

{

//Abort Pickup after five tries //Philosopher drops left chopstick }

// Proceed if both chopsticks are picked up successfully.

if (not abort pickup)

{

if (check if meals have all been eaten)

//All meals have been eaten!

else

{

// begin to eat

Mealseaten --;

//Philosopher begins to eat

BusyWaitingLoop();

//Philosopher %d finishes eating

}

//Philosopher drops left chopstick

//Philosopher drops right chopstick

}

//Philosopher Thinking

BusyWaitingLoop();

}

//Philosopher is waiting to leave

while (waiting for all philosophers to be ready)

{

BusyWaitingLoop();

}

//All Philosophers leave the table together.

**Task 3**

*Explain the method you used to resolve the deadlock problem. Why did you choose this particular method?*

A local variable *sentCount* is defined to keep tracking the number of attempt to send a mail. Itis incremented if the destined mailbox is full. To prevent potential deadlock, the sending attempt will be aborted after failed for three times, that is, when *sentCount* reaches 3. I chose this method because busy waiting loop can provide me the possibility to keep tracking the number of sending attempts made by each person. In case of task 4, this might not be feasible.

*Data structure and algorithm used:*

A *MailBox* array consists of *P* x *S* slots, where P is the number of people and S is the capacity of each person’s mailbox. Each slot contain information about content (*char\* Msg*) and sender (*int Who*). A pointer *MsgPtr* is associated with each mailbox, which points to the next available message. Each *MailBox* acts as a FILO (First In Last Out) queue. So *MsgPtr* will be incremented(or decremented) after a message has been received (or read). A person will read mailbox only if his mailbox is not empty (*MsgPtr* > 0) and sent mail when the destined mailbox is not full (*MsgPtr* < *S*). *MsgSentCnt* is defined to track the overall number of message being sent. Once it reaches the prompted value *M*, no sending request will be granted. Since *MailBox*, *MsgPtr*, *MsgSentCnt* are all global variables and may be modified by different procedure, Semaphores are applied for guaranteeing synchronization. Also, each person is associated with a Boolean variable *done*, which will be set TRUE when all messages have been sent and the corresponding mailbox is empty. The program is terminated when all elements in Boolean array *done* are set TRUE.

**Task 4**

*Did you experience any deadlock when testing this task? How was it different from Task 3?*

In Task 3, when the recipient’s mailbox is full, sender will try three times before abort. But in Task 4, since no busy waiting loop is allowed, it is necessary to keep tracking the number of available slots remained in each mailbox when granting sending request. If the recipient’s mailbox is full, program will be put into queue associated with *freeSpaceSemaphore*. So the trick here is that sender should always try to claim *freeSpaceSemaphore* first and then *mailboxSemaphore* of the recipient. In this way, recipientwill be able to claim *mailboxSemaphore* and read messages and increase the value to wake up the thread being queued. Otherwise, deadlock is inevitable.

*Data structure and algorithm used:*

Each mailbox is associated with one *freeSpaceSemaphore* and *mailboxSemaphore.* The latter is used to protect the shared resources — mailbox. The sending request is granted when all the following three conditions are satisfied. First, the recipient’s mailbox is not full — *freeSpaceSemaphore* is larger than 0. Second, the recipient is not reading mail at this moment — *mailboxSemaphore* is 1. Third, no one is currently sending mail or trying to update the number of message being sent so far — *MsgCntSemaphore* is 1. The basic algorithm for task 4 is similar to that of task 3.

**while** (Person *i* can’t leave permanently) {

Person *i* enter the post office

**while** (mailbox[*i*] is not empty){

mailboxSemaphore[*i*] -> P();

read msg in mailbox[*i*];

freeSpaceSemaphore[*i*] ->V();

mailboxSemaphore[*i*] -> V();

yield;

}

Compile message to Person *j*; // j is randomly selected other than themselves.

freeSpaceSemaphore[*j*] -> P();

mailboxSemaphore[*j*] -> P();

MsgCntSemaphore->P();

**if** (MsgSentCnt < M) {

Send message to Person *j*;

MsgSentCnt++;

} **else**

freeSpaceSemaphore[*j*]->V(); // Abort sending attempt

mailboxSemaphore[*j*]->V();

MsgCntSemaphore->V();

Leave the office;

Wait for 2-5 cycles;

Check to see if Person *i* can leave permanently.

}