Group Member (CLID)

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

All questions have been arranged by their associated tasks.

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.

```
Philosopher joins the room wait to sit
while (not all philosophers in room)
        currentThread->Yield();
All philosiphers sit down at table
while (there are meals to be eaten) {
    if (left chopstick is there) {
        Philosopher picks up left chopstick
        if (right chopstick is there) Philosopher picks up right chopstick
        else Abort Pickup after five trials and drops the left chopstick
    } else          Abort Pickup after five tries
    if (not abort pickup) {
        if (all meals have been eaten)         //All meals have been eaten!
        else {
                Philosopher begins to eat
                Mealseaten --;
                BusyWaitingLoop();
        }
        Philosopher drops left and right chopstick
    }
    BusyWaitingLoop();//Philosopher Thinking
}
while (not all philosophers are ready to leave)          BusyWaitingLoop();
All Philosophers leave the table together.
```

Task2

Task:	Philosopher:	Meals:	Seed:	Ticks:
Task1	4	4	123	926
Task2	4	4	123	1726
Task1	6	3	987	1127
Task2	6	3	987	2688
Task1	20	20	36172	7281
Task2	20	20	36172	10889

Run Task 1 and Task 2 with the same parameter and seed. Explain the finding.

As shown by the above table, between the two tasks Task 2 takes almost double the amount of cycles to complete. Task 2 is using semaphores; therefore each thread that needs to access a protected global variable will pause, resulting in increased CPU cycles. Whereas in Task 1, there are no protected global variables and the threads are only paused in their respective busy waiting loop, and this will allow other threads to continue normally.

Make the philosopher yield between attempting to pick up the left and right chopsticks. Run Task 1 and Task 2. Explain the results.

In Task 1, yielding between picking up the chopsticks causes a deadlock issue, as other philosophers still have access to the chopstick the original philosopher would have picked up if it were not for the yield. This results in other philosophers picking up the stick and causing deadlock when the original philosopher thread is active again.

In Task 2, with yielding between picking up the chopsticks there was no deadlock like in Task 1. Using semaphores protects all global variables, so even though the yield statement affected the state of the original philosopher thread, no other threads were allowed to get a chopstick until the original owner of the chopsticks released them, alas the power of semaphores!

Data structure and algorithm used:

```
Semaphore mutex, EnterMutex, LeaveMutex, SitSemaphore, PhilGrantSemaphore [P], MealLeftMutex, LeaveSemaphore;
typedef enum {THINKING,ISEATING,ISHUNGRY} pState; //state of philosophers
DinPhilSemaphore(i) {
    EnterMutex -> P();
    NumPhilNotIn--;
    if (all philosophers are present)
        for (int j=0; j<p; j++)
            SitSemaphore -> V(); // grant P sit requests
    EnterMutex -> V();
    // All philosopher have sat at the table.
    while (there is meal left) {
        GetSticks(i);
        Eat(i);
        PutSticks(i);
        BusyWaitingLoop();
    }
    LeaveRoom(i);
}
Getsticks (i) {
    mutex->P();
    PhilState[i] = ISHUNGRY;
    if (PhilState[(i-1+P) % P] != ISEATING)        Pick up the left chopstick;
    if (PhilState[(i+1) % P] != ISEATING)          Pickup the left chopstick;
    if (PhilState[i] == ISHUNGRY && PhilState[(i-1+P) % P] != ISEATING && PhilState[(i+1) % P] != ISEATING) {
        PhilState[i] = ISEATING;
        PhilGrantSemaphore[i]->V();     // skip PhilGrantSemaphore[i] -> P() if it is executed
    }
    mutex->V();
    PhilGrantSemaphore[i]->P(); // only be exected if no fork has been acquired.
}
Eat (i) {
    MealLeftMutex -> P();
    if (MealLeft != 0)             Eat; MealLeft--;
    MealLeftMutex -> V();
    BusyWaitingLoop();
}
PutSticks (i) {
    mutex->P();
    // Put down the left stick and forcefully wake up its left neighbor to check the state to avoid deadlock
    if (PhilState[(i-1+P) % P] == ISHUNGRY) PhilGrantSemaphore[(i-1+P) % P]->V();
    // Put down the right stick and forcefully wake up its right neighbor to check the state to avoid deadlock
    if (PhilState[(i+1) % P] == ISHUNGRY) PhilGrantSemaphore[(i+1) % P]->V();
    PhilState[i] = THINKING;
    mutex->V();
}
LeaveRoom (i) {
    LeaveMutex -> P();
    NumRdyLeave++;
    if (NumRdyLeave == P)
        for (int j=0; j<P; j++)
            LeaveSemaphore -> V(); // grant P leave requests
    LeaveMutex -> V();
    LeaveSemaphore -> P();
    All Philosophers start to leave.
}
```

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. It is 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, recipient will 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.
}
```