Name:_____ Student ID:_____

CSSAP 443: Mid Term Exam

February 21, 2002
3:30pm – 5:30pm

- This exam contains **4 problems** on **6** pages. You have 120 minutes to earn 100 points.
- This exam is open book.
- Do not spend too much time on any problem. Read them all through first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat!
- Please show all the details of your work and justify your answers. As in all cases in this class, correct answers without justifications will **not** earn you full credit.

Problem 1 (25%)	
1a (5%)	
1b(5%)	
1c (5%)	
1d(10%)	
Problem 2 (30%)	
2a(5%)	
2b(10%)	
2c(15%)	
Problem 3 (30%)	
3a(10%)	
3b(5%)	
3c(15%)	
3d(10%)	
Problem 4 (15%)	

Problem 1 (25%). Student John Doe learned the lesson from Homework Assignment #1 (hw1) and came up with a better design for solving the thread termination sequencing problem. Here is his new and improved TsyncObject:

```
#define kLastThread 5

class TsyncObject {
    private:
        int             fCommand;
        CMutex          *fCommandMutex;
    public:
        TsyncObject() : fCommand(-1) {
                        fCommandMutex = new CMutex();
                        fCommandMutex->Unlock();        }

        void SetCommand(int id)     {
                        fCommandMutex->Lock();
                        fCommand = id;
                        fCommandMutex->Unlock();        }

        bool Identified(int id) { return (id == fCommand);      }
};
```

As in hw1, this is what TthreadBase looks like:

```
class TthreadBase {
    private:
        static      TsyncObject sfSyncObj;
    protected:
        int         fID;        // id of this thread
    public:
        static TsyncObject& GetSyncObject() { return sfSyncObj; }

        TthreadBase(int id) : fID(id)  {}
        const int ThreadID() const { return fID; }
        void StartThread() {
            _beginthread(StupidThreadWorker, 0, (void *) this );
        }
};
```

Where John's ***modified*** StupidWorkerThread() looks like:

```
void StupidThreadWorker(void *parm) {
    TthreadBase *td = (TthreadBase *) parm;
    TsyncObject &syncObj = td->GetSyncObject();

    while ( !syncObj.Identified(td->ThreadID()) );
    cerr << "Thread: id[" << td->ThreadID() << "]: Done!" << endl;

    ////////////////////////////////////////////////////////////////////////////////
    // Update the id in SynObject for other threads
    // to have an opportunity to terminate.
    ////////////////////////////////////////////////////////////////////////////////
    id = td->ThreadID() - 1;
    if (id < 0)
        id = kLastThread - 1;
    syncObj.SetCommand(id);

    _endthread();
}
```

and here is his ***slightly*** modified test driver:

```
int main() {
        TsyncObject &sync = TthreadBase::GetSyncObject();
        //
        // Start all the worker threads ...
        //
        for (int i=0; i<kLastThread; i++) {
                TthreadBase *td = new TthreadBase(kLastThread-1-i);
                td->StartThread();
        }

        int command;
        cout << "Command? ";
        cin >> command;

        ////////////////////////////////////////////////////////////////////////////////////////////
        // Notice: Comparing to hw1, command vs command+1 changes here.
        ////////////////////////////////////////////////////////////////////////////////////////////
        sync.SetCommand(command);
        while (!sync.Identified(command+1));
        cerr << "Main terminates!" << endl;
        return 0;
}
```

a. **(5%)** When the user enters a 3 as input, what would you expect the output to look like? Please provide descriptions justifying why the output.

b. **(5%)** After looking at the code for 8.6 seconds, Student Mary pointed out this is still a really bad design because the threads are still performing busy wait. Is Student Mary correct? Please justify your answers. *Yes/no without reasons will not earn you any credit.*

c. **(5%)** Student Mary claimed that with this horrible design, we could remove the fCommandMutex and still achieve the same results (from that of Part **a**.). Is Mary correct? Please justify your answers. *Once again, yes/no without reasons will not earn you any credit.*

d. **(10%)** If all we care is to ensure the threads terminate in sequence under the user's control, show how you would modify John Doe's design to achieve this task. *Hint: you **do not** need any extra synchronization variables, remember, we are **not** worried about efficiency, we just want to control the termination sequence of the threads.*

Problem 2 (30%). The first time I tried to develop MergeSort, this is what I did:

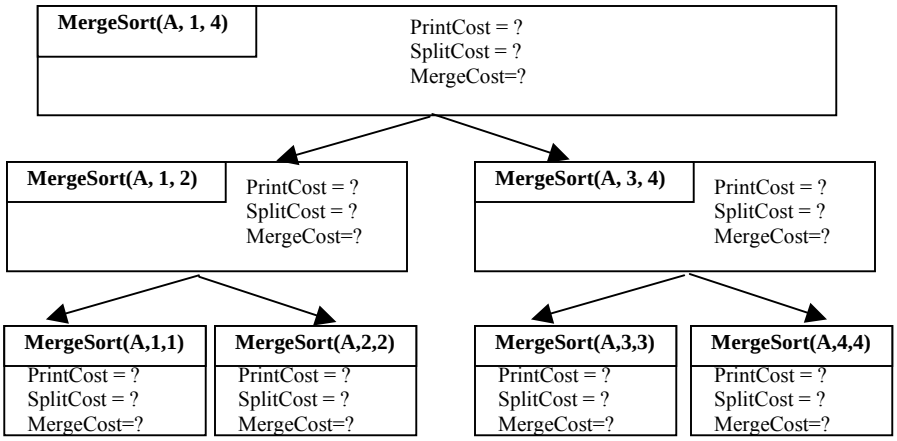
```
MergeSort(A, startIndex, endIndex)
        // A is the array containing all the input numbers
        // startIndex is the starting position we will sort
        // endIndex is the end position

        if startIndex < endIndex then

                PrintArray(A, 1, arraySize)
                        // Print out the content of the array from 1 to arraySize
                        // arraySize is the size of the input array.
                        // NOTE: arraySize is not endIndex-startIndex+1
                        //       arraySize is the size of the original input array.

                midIndex = (startIndex + endIndex) / 2;
                MergeSort(A, startIndex, midIndex);
                MergeSort(A, midIndex+1, endIndex);
                Merge(A, startIndex, midIndex, endIndex);
```

Feeling the sorting performance was sluggish; I wanted to know what kind of running time I am dealing with here. I know the tight asymptotic bound on the worst-case running time of **Merge** is θ(n), and **PrintArray** always runs in θ(arraySize). With these observations, I derived the recurrence relation describing my implementation being:

$$T(n) = 2T(\frac{n}{2}) + \Theta(arraySize) + \Theta(n)$$

a. **(5%)** Explain why this equation is ***not*** in the *correct form* for Master's Theorem and thus we **cannot** use Master's Theorem to solve this.

b. **(10%)** Feeling quite helpless, I decided to draw a recursion tree for a 4-element array and compute the actual tight asymptotic bound (θ-notation) of the worst case running time for my **MergeSort** for this specific case. Here is my recursion tree, please help me complete the tree. What is the tight asymptotic bound for my MergeSort(A, 1, 4)?



c. **(15%)** Base on what we learned from this example; derive the tight asymptotic bound (θ-notation) of the worst case running time for my **MergeSort**(A, 1, n). **Warning:** *This problem can be quite involve, do not spend time getting stuck on this problem.*

Problem 3 (30%). Refer the following array based linked list implementation:

```
#define NullAddress -1
int fSystemMemory[20];
int             KeyField(int ptr)  { return fSystemMemory[ptr];   }
int             NextField(int ptr) { return fSystemMemory[ptr+1]; }

void WhyThis(int cnt, int first, int second) {
        if (first == kNullAddress)
                fSystemMemory[cnt+1] = second;
                return;
        else
        if (second == kNullAddress)
                fSystemMemory[cnt+1] = first;
                return;

        ////////////////////////////////////////////////////////////////////////////////////
        // LOCATION A: For debugging purposes:
        //            PrintList(first);
        //            PrintList(second);
        ////////////////////////////////////////////////////////////////////////////////////

        if (KeyField(first) < KeyField(second))
                fSystemMemory[cnt+1] = first;
                cnt = first;
                first = NextField(first);
        else
                fSystemMemory[cnt+1] = second;
                cnt = second;
                second = NextField(second);

        WhyThis(cnt, first, second);
}

int main()  { // This is where we begin
        int result, cnt;
        int first = 4, second = 2;

        PrintList(first);
        PrintList(second);

        if (KeyField(first) < KeyField(second))
                result = first;
                first = NextField(first);
        else
                result = second;
                second = NextField(second);

        fSystemMemory[result+1] = kNullAddress;
        cnt = result;

        WhyThis(cnt, first, second);

        PrintList(result);
}
```

Given that initially fMemorySystem looks like:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
19	-1	0	12	1	8	17	-1	8	18	9	12	6	16	7	6	16	6	14	0

And, this is what **PrintList** looks like:

```
void PrintList(int head) {
        int p = head;
        cout << "LinkList is: ";
        while (p != kNullAddress)
                cout << KeyField(p) << " ";
                p = NextField(p);
        cout << endl;
}
```

a. **(10%)** What is the output from the **main**()?

b. **(5%)** What does **WhyThis**() do?

c. **(15%)** Please derive and solve the equation describing the tight asymptotic (θ-notation) bound on the worst-case running time of **WhyThis**. *Hint: it is more straightforward to derive an equation based on the total number of elements in **both** (first and second) of the input lists.*

d. ~~**(10%)** Please derive and solve the equation describing the tight asymptotic bound on the worst-case running time of **WhyThis** if I uncomment the two **PrintList** at **LOCATION A**.~~ **Problem removed. Please do not do this. We will discuss this in class.**

Problem 4 (15%). You are given that, with an n-element array the running time of :

KelvinSort() is exactly	$8n\log_{10} n + 2n^{1.5}$
KelvinSearchSortedArray() is exactly	$4\log_{10} n$
KelvinSearchUnsortedArray() is exactly	n.

We are given an input array size of *1,000,000*, and you are told you have to search the array exactly 2075 times. You have to use the above three routines to perform your searches. To search the array exactly 2075 times, is it faster to sort the array with KelvinSort and then use KelvinSearchSortedArray to search? Or is it faster to just search the unsorted array directly? *Answers without justification will not earn you any credit.*