Name: ____________________ Roll No: ____________

(COL 380) Introduction to Parallel and Distributed Programming Feb 22, 2024

Mid-Term Exam

Duration: 120 minutes **(60 marks)**

Note: Be concise in your writing. You can use rough sheets for calculations. But you cannot submit any additional sheet for grading on Gradescope. So make sure you are certain when you write something (after rough work, or use a dark pencil). If you cheat, you will surely get an F in this course.

---

1. Below are two attempts to solve the well-known "Too Much Milk" problem i.e. prevent both threads from buying milk. (note: the buyMilk function will increment the milk variable):

Attempt 1:

Thread A
```
1 noteA = 1;
2 if (noteB == 0) {
3    if (milk == 0) {
4       buyMilk();
5    }
6 }
7 noteA = 0;
```

Thread B
```
1 noteB = 1;
2 if (noteA == 0) {
3    if (milk == 0) {
4       buyMilk();
5    }
6 }
7 noteB = 0;
```

Attempt2:

Thread A
```
1 noteA = 1;
2 if (noteB == 0) {
3    if (milk == 0) {
4       buyMilk();
5    }
6 }
7 noteA = 0;
```

Thread B
```
1 if (noteA == 0) {
2    noteB = 1;
3    if (milk == 0) {
4       buyMilk();
5    }
6    noteB = 0;
7 }
```

Does either approach solve the "too much milk" problem? Does either have a "too little milk" problem, i.e. neither thread buys milk? [5 + 5 = 10 marks]

Attempt 1 has too little milk problem. No milk is bought with the following interleaving of thread A and thread B.

1. noteA = 1

1. noteB = 1

2. if (noteB == 0)

2. if (noteA == 0)

no milk will be bought

Name: ______________ Roll No: ______________

Attempt 2 has too much milk problem. Milk is bought by both threads with the following interleaving.

	1. if (note A == 0) {
1. note A = 1	
2. if (note B == 0)	
	2. note B = 1
	3. if (milk == 0)
3. if (milk == 0)	
	4. buy milk ()
4. buy milk ()	

Both threads buy milk.

Attempt 1 will not have too much milk problem. This code is symmetric for both threads. So the following holds for the converse.

If we assume note A = 1 and if (note B == 0) have been done by thread A, then thread B cannot enter if (note A == 0) until thread A sets note A = 0 at the end.. But by then milk will become 1 in buy Milk() by Thread A, so thread B will not buy Milk. Thus Attempt 1 does not have too much milk problem.

Attempt 2 will not have too little milk problem. If thread A has not bought milk, then either "if (note B == 0)" failed (In this case, thread B has set note B = 1 and will go through next steps to buy milk or "if (milk == 0)" failed and that means thread B has already bought milk. Similarly for thread B not buying milk. So one thread will buy milk in this approach.

Name: ______________________ Roll No: ______________

2. Function **threadprint** accepts as arguments the current thread's unique ID and a debug message to print. If each thread calls **threadprint** exactly once on start, how many possible interleavings are there with n threads? [2 marks]

```
void threadprint(int threadid, char *message) {
    printf("Thread %d: %s\n", threadid, message);
}
```

b. Function **ordered_threadprint** attempts to print debug messages ordered by thread ID. Describe three ways in which the synchronisation in this implementation is incorrect, and provide a corrected pseudocode implementation. [8 marks]

```
int next_thread_id = 0; // Next ID to print
pthread_mtx_t ordering_mtx; // Lock protecting next ID
pthread_cond_t ordering_cv; // next_thread_id has changed

void ordered_threadprint(int thread_id, char *message) {
    pthread_mtx_lock(ordering_mtx);
    if (thread_id != next_thread_id) {
        pthread_cond_wait(ordering_cv, ordering_mtx);
    }
    next_thread_id = next_thread_id + 1;
    pthread_mtx_unlock(ordering_mtx);
    printf("Thread %d: %s\n", thread_id, message);
}
```

There can be any order of interleaving. So n! for n threads

Issues:

(i) there is no signal to wake the threads up. So they will remain in wait forever.

(ii) the condition check should be in while loop, since each thread on waking must recheck for the condition, and release the lock if it is not satisfied, since actually only 1 thread is the correct next thread.

(iii) the message should be printed before releasing the lock, else even if everything else is correct, incorrect interleavings are possible as a later scheduled thread can print earlier.

Name: ______________________ Roll No: ______________

corrected code

```
void ordered_threadprint(int thread_id, char* msg){
    pthread_mtx_lock(ordering_mtx);
(ii) → while (thread_id != next_thread_id){
        pthread_cond_wait(ordering_cv, ordering_mtx);
    }
    next_thread_id = next_thread_id+1;
(iii) → printf("thread %d : %s\n", thread_id, msg);
(i) → pthread_cond_signal(ordering_cv, ordering_mtx);
    pthread_mtx_release(ordering_mtx);

OR
pthread_cond_broadcast(ordering_cv)
```

Roll No: ____________

Name: ____________

3. Consider the semi-skip list structure pictured below. Each node maintains a pointer to the next node and the next-next node in the list. The list must be kept in sorted order.

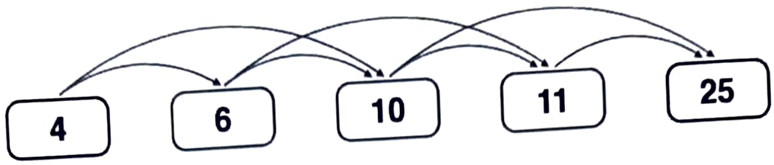


Figure 1: Example semi-skip list data structure

A node struct is given below.

```
struct Node {
    int value;
    Node* next;
    Node* skip; // note that skip == next->next
};
```

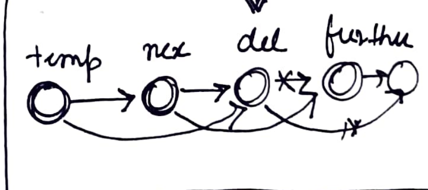
Please describe a thread-safe implementation of node deletion from this data structure. You may assume that deletion is the only operation the data structure supports. Please write C-like pseudocode. To keep things simple, you can ignore edge-cases near the front and end of the list (assume that you're not deleting the first two or last two nodes in the list, and the node to delete is in the list). If you define local variables like curNode, prevNode, etc. just state your assumptions about them. However, please clearly state what per-node locks are held at the start of your process. E.g., "I start by holding locks on the first two nodes.". Full credit will only be given for solutions that maximize concurrency. [10 marks]

```
// delete node containing value
void delete\_node(Node* head, int value) {
}
```

```
delete_node (Node* head, int value) {
    temp = head;
    while (temp -> next -> next -> value != value) {
        temp = temp -> next;
    }
    next = temp -> next;
    del = temp -> next -> next;
    further = del -> next;

    (1) temp -> next -> next = further;
    (2) next -> next = further -> next;
    (3) del -> next = NULL;
    (4) del -> next -> next = NULL;
    (5) remove del.
```

want to remove this



hold locks here

Name: ____________________

Roll No: ____________

temp is the node previous to previous of the node we want to delete.

nex is the next node of temp i.e. previous of the node we want to delete.

del is the node we want to delete.

Since we are changing outgoing pointers from temp, next del we need to grab their locks.

While searching for temp, assigning del, nex and further, I need not hold any lock.

While changing pointers or deleting nodes, I have to hold locks i.e. statements 1 - 5 on previous page.

I need to hold all 3 locks - I will start by holding lock of del, then temp, then nex - while grabbing hold of any lock if it fails, then I will preempt resources and restart.

After finishing delete - I will release the locks.

Roll No: ____________

Name: ____________________

4.
```
struct Graph_node {
    Lock lock;
    float value;
    int num_edges; // number of edges connecting to node
    int* neighbor_ids; // array of indices of adjacent nodes
};


// a graph is a list of nodes
Graph_node graph[MAX_NODES];
```

Consider the undirected graph representation shown in the code above. We need a program that atomically updates each graph node's value field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows:

```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; inum_edges; i++){
        LOCK(graph[n->neighbor_ids[i]].lock);
        // now perform computation...
        }
```

a. Consider running the update code in parallel on nodes 0 and 1 in the two graphs below. For each graph, determine if deadlock occurs. Please describe why or why not. [6 marks]

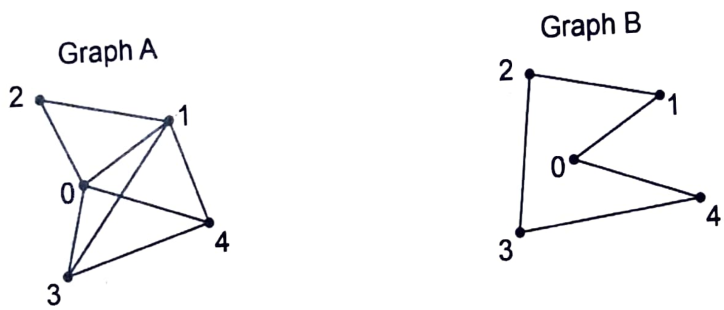


Figure 2: Graphs

b. In case deadlock occurs, explain how you might avoid it, assuming you must still only use locks. [4 marks]

In both graphs, deadlocks can occur.
If Threads T_1 and T_2 are trying to update the values in 0 and 1. T_1 acquires lock for 0 and T_2 acquires lock for 1. Now 1 is neighbor of 0 and viceversa. Therefore T_1 will try to acquire lock for 1, which is held by T_2 and T_2 will try to acquire lock for 0, which is held by T_1. Thus T_1 and T_2 will be deadlocked.

Name: ____________________

Roll No: ____________________

We can avoid deadlock, by acquiring locks in sorted order.

we first sort (neighbor ids + node id) to get sorted id. sorted-ids is an increasing array.

Next locks are acquired by each thread in increasing order.

This avoids deadlock.

```
void update (int id) {
        sorted_arr = sort (neighbor_ids, id);
        for (int i = 0; i < num_edges ; i++)
        {
                acquire lock of sorted_arr[i];
```

$\boxed{OR}$ During any lock acquisition, if a thread finds lock is already acquired, then it yields processor, releasing all locks it has acquired. When it gets to run again after other thread has released the locks, it again starts by acquiring all locks from the beginning.

Roll No: ____________

Name: ____________

5. For the vector addition kernel and the corresponding kernel launch code, answer each of the sub-questions below in a few words.

```
__global__ void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
//assume that size has been set to the actual length of arrays A, B, and C
    int size = n * sizeof(float);

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
    vecAddKernel<<<ceil(n/256), 256>>>(A_d, B_d, C_d, n);
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

}
```

(a) Assume that the size of A, B, and C is 1000 elements. How many thread blocks will be generated? [2 marks]

Number of thread blocks is ceil(n/256) ie the first term in vecAddKernel<<< ceil, 256>>> = 4 if n = 1000

$\boxed{\text{Ans: 4}}$

(b) Assume that the size of A, B, and C is 1000 elements. How many warps are there in each block? [2 marks]

Number of threads in a block is second ~~argument~~ term in kernel launch vecAddKernel <<<ceil(n/256), 256>>> ie. 256. As 1 warp has 32 threads, number of warps in block to have 256 threads is 256/32 = 8

$\boxed{\text{Ans: 8}}$

(c) Assume that the size of A, B, and C is 1000 elements. How many threads will be created in the grid? [2 marks]

Total number of threads in the grid, with which the kernel will be launched is the product of the two terms in vecAddKernel<<< >>> i.e. # blocks * # threads per block = 4 X 256 = 1024

$\boxed{\text{Ans: 1024}}$

(d) Assume that the size of A, B, and C is 1000 elements. Is there any control divergence during the execution of the kernel? If so, identify which line of the code that causes the control divergence. Explain why or why not. [2 marks]

Yes there is control divergence, which is caused by the line

if (i<n) C_d[i] = A_d[i]+ B_d[i]

Since the total number of threads in the grid is 1024, larger than the size of the arrays, the last warp will have divergence. The first 8 threads in the warp will take the true path and remaining 24 threads will take false path.

(e) Assume that the size of A, B, and C is 768 elements. Is there any control divergence during the execution of the kernel? If so, identify which line of the code that causes the control divergence. Explain why or why not. [2 marks]

In this case there will be 3 blocks and 8 warps in each block → with total threads

$3 * 8 * 32 = 24 * 32 = 768$

As the total number of threads is the same as the number of elements in the vector, all threads in the warp will take the if / true path. There will be no control divergence.