Name: ____________________

SE205 - EXAM

General Instructions

- Fill in your name at the top on each page.
- Work on your own.
- You may use the lecture slides and your TD notes.
- Cell phones and all other electronic devices switched off.
- The presentation style and clarity of your answers will be taken into account.
- Only accurate and well-justified responses will be considered.
- The exam is structured in 5 parts.
- You can get a maximum of 15 points.
- You have 90 minutes.

Question	Points	Score
Dependencies	1	
Interleavings	$1\frac{1}{2}$	
Atomics	$1\frac{1}{2}$	
Synchronization	2	
Access the clock in C	$\frac{1}{2}$	
Delay until a given instant	$1\frac{1}{2}$	
Implement a periodic thread	1	
Synchronization	3	
Algorithm	2	
Actors	1	
Total:	15	

1 Dependencies

Code Snippet

Consider the following code snippet for the questions below:

```
void inner_sum(int *a, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      a[i] += a[j];
    }
  }
}
```

Questions

1. (1 point) **Dependencies**

   Which loop-carried dependencies are present between successive iterations of the outer loop (counting on `i`)?

   ◯ Anti dependencies (Write-After-Read) between the definition of `a[i]` in the first iteration of the inner loop on the third iteration of the outer loop (`j=0` and `i=2`) and the use of `a[j]` in the third iteration of the inner loop on the first iteration of the outer loop (`j=2` and `i=0`).

   ◯ As we cannot say anything about the pointer `a`, we cannot say anything about the loop-carried dependencies.

   ◯ Output dependencies (Write-after-Write) between the definition of `a[i]` on the last iteration of the inner loop on an iteration i of the outer loop (`j=n-1` and `i=`i) and the definition of `a[i]` on the first iteration of the subsequent iteration of the outer loop (`j=0` and `i=`i`+1`).

   ◯ No loop-carried dependencies exist with regard to the outer loop, only with regard to the inner-most loop.

   ◯ True dependencies (Read-After-Write) with regard to the increment of `a[i]` between subsequent iterations of the *inner loop* for an iteration i of the outer loop.

   ◯ The program contains a race condition and thus is invalid.

Name: ______________________

2 The Shared-Memory Model

Code Snippet

Consider the following code for the subsequent questions. The code shows two threads, running in parallel, as well as some definitions of global variables shared by those threads:

```
#define ITERATIONS 1000

unsigned int readIdx = 0;
unsigned int writeIdx = 0;
unsigned int numItems = 0;

#define SIZE 100
int sharedData[SIZE];
```

Code of Thread 1

```
for(unsigned int i(0); i < ITERATIONS; i++) {
  // wait for data to become available
  while (numItems == 0);

  // read from the circular buffer and print it
  std::cout << sharedData[readIdx] << "\n";

  // increment the read index and update the item counter
  readIdx = (readIdx + 1) % SIZE;
  numItems = numItems - 1;
}
```

Code of Thread 2

```
for(unsigned int i(0); i < ITERATIONS; i++) {
  // wait until space is available in the circular buffer
  while (numItems == SIZE);

  // write random data into the circular buffer
  sharedData[writeIdx] = random();

  // update the write index and item counter
  writeIdx = (writeIdx + 1) % SIZE;
  numItems = numItems + 1;
}
```

Questions

2. (1 ½ points) **Interleavings**

   Find an interleaving explaining an execution where thread 2 adds 1000 values to the queue, but thread 1 only prints 999 numbers and then ends up waiting infinitely. It is sufficient to refer to line numbers involving the variable `numItems`.

3. (1 ½ points) **Atomics**

   How can the code from above be made thread safe? Assume that only the two threads shown here are executed. Your solution should only use functions provided by C11 and be minimal, i.e., you should propose only those modifications that are strictly necessary. Justify your solution.

4. (2 points) **Synchronization**

   Is your solution from above safe when more than two threads access the queue in parallel? Assume that all sorts of combinations of the two thread types from above might appear, even multiple times. If the program is safe, explain why. Otherwise, explain the problem.

---

3 POSIX Concurrency Problem

Bob is in charge of a POSIX application in which several functions execute periodically, their execution being interlaced. This code does not use POSIX threads yet. Bob must port this application to a multi-processor platform so that it can take advantage of real parallelism. Bob needs help! The following questions aim to address his problem progressively. All answers must rely on the lecture material or on the previous questions.

Questions

5. ($\frac{1}{2}$ point) **Access the clock in C**

   Help Bob implementing a `clock()` function for accessing the system clock (the returned value is in milliseconds). Your answer must rely on the lecture. In the following questions, Bob will use this `clock()` method to access the system clock. Complete the function below:

```
long clock(){

};
```

6. (1 $\frac{1}{2}$ points) **Delay until a given instant**

   Help Bob implementing a `delay_until(long t)` function in order to wait until time instant t (this time value is in ms). Your answer must rely on the lecture material or on the previous questions. Next, Bob will use only this `delay_until(long t)` function in order to wait until time instant t. Complete the function from below:

```
void delay_until(long t){

};
```

7. (1 point) **Implement a periodic thread**

Help Bob implementing a periodic C thread and, in particular, its main function `main_worker` that executes the `work()` function regularly, at each interval (period) provided by the `arg` parameter. Your answer must rely on the lecture material or on the previous questions.

```
void * main_worker (void * arg) {
  long period = (long)(long *)(arg);
  long t = clock() + period;
    while(1){
         work();
         /* wait until the end of the period */




         /* determine the instant of the next period */




      };
   };
```

4 Java Concurrency

In the following exercise we want to extend an existing implementation of a sorted single-linked list. In the provided code (see below), two nodes representing the minimum and maximum `int` values are always present in the list (see `FineGrainedList` constructor). We have two methods to `add` an item and `remove` an item. Each new item is inserted in the list, while keeping it sorted.

We want to protect this list against concurrent accesses through fine-grained synchronization. This means that, instead of using a single lock to synchronize every access to the list, we split the list into independently synchronized nodes, ensuring that method calls interfere only when trying to access the same node at the same time.

A `Node` object includes an `item` represented as an `int`, a reference to the `next` node and a `ReentrantLock` (or a `Lock`) `mutex`. This mutex is intended to protect the node against concurrent accesses. It protects the attributes `item` and `next` of the current node.

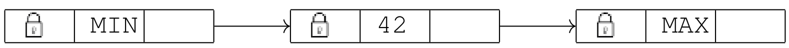
Note that to add a new node between two nodes A and B, we must lock the previous node A, then the current node B, in precisely this order to prevent deadlocks. To modify node B, we also have to lock A and then B, B because we modify node B, and A in order to prevent any `remove` operation on B.

We have two special nodes `head` (with minimum integer value `MIN`) and `tail` (with maximum integer value `MAX`). These two nodes are always present. To add a new `item` to an **empty** list, we first lock the `head` node, then the `tail` node (actually the `head.next` node, as the list is empty) to finally insert the new node between the `head` node and the `tail` node.

Questions

8. (3 points) **Synchronization**

   Extend the existing code below in order to allow concurrent fine-grained synchronization for the given list instance shown here:



The code is provided on the next page ⟶

Name: ______________________________

```java
public class Node {
  ReentrantLock mutex;
  int item;
  Node next;
  Node(int item){
    this.item = item;
    this.mutex = new ReentrantLock();
  }
}
public class FineGrainedList {
  public Node head;
  public FineGrainedList(int min, max) {
    head = new Node(min);
    head.next = new Node(max);
  }
  public boolean add(int item) {
    Node pred = head;
    try {
      Node current = pred.next;
      try {
        while (current.item < item) {
          pred = current;
          current = current.next;
         }
        if (current.item == item) {
          return false;
        }
        Node newNode = new Node(item);
        newNode.next = current;
        pred.next = newNode;
        return true;
      } finally {
      }
    } finally {
    }
  }
  public boolean remove(int item) {
    Node pred = null, current = null;
    try {
      pred = head;
      current = pred.next;
      try {
        while (current.item < item) {
           pred = current;
           current = current.next;
        }
        if (current.item == item) {
           pred.next = current.next;
           return true;
        }
        return false;
      } finally {
      }
    } finally {
    }
  }
}
```

5 Algorithms for Concurrent Systems

Our broadcast system implements Ricart-Agrawal's algorithm with Lamport's clocks (enriched with node ids). We consider a system of 3 nodes $N1$ to $N3$. Each node broadcasts one message: node $N1$ broadcasts message $M1$, and so on. Messages are received in the order given in the following table; the Lamport's clock of the receiving node is given between parentheses:

$N1$	$M3$ (15)	$M1$ (16)	$M2$ (17)
$N2$	$M2$ (16)	$M1$ (17)	$M3$ (18)
$N3$	$M1$ (17)	$M3$ (18)	$M2$ (19)

Questions

9. (2 points) **Algorithm**

According to this algorithm, in which order will these messages be delivered on each node? Does this algorithm provide a causal order, a total order or both? Only accurate and well-justified responses will be considered.

Name: ____________________

6 The Message-Passing Model

Questions

10. (1 point) **Actors**

    Explain what an actor is and which actions an actor might perform. Then explain the relation between the actors' characteristics and the main guiding principles of the actor model.
