Problem 2 (4 points) A *ticket lock* is a lock implemented using two shared counters, `next_ticket` and `now_serving`, both initialised to 0. A thread wanting to acquire the lock uses an atomic operation to fetch the current value of `next_ticket` as its unique sequence number and increments it by 1 to generate the next sequence number. The thread then waits until `now_serving` is equal to its sequence number. Releasing the lock consists on incrementing `now_serving` in order to pass the lock to the next waiting thread.

Given the following data structure and incomplete implementation of the primitives that support the `ticket lock` mechanism:

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

We ask:

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
// obtain my unique sequence number from next_ticket
// generate the next_ticket sequence number
// wait until my sequence number is equal to now_serving
}
```

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
// obtain my unique sequence number from next_ticket
// generate the next_ticket sequence number
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? WHY?

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
// obtain my unique sequence number from next_ticket
// generate the next_ticket sequence number
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **Yes!**
WHY?
You have to get the current next_ticket value and update it for next thread, and it **SHOULD BE UNIQUE!!!!**

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
// obtain my unique sequence number from next_ticket
// generate the next_ticket sequence number
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **Yes!**
WHY?
You have to **get** the current next_ticket value and **update** it for next thread, and it **SHOULD BE UNIQUE!!!!**

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
// obtain my unique sequence number from next_ticket
// generate the next_ticket sequence number
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **Yes!**
WHY?
You have to **FETCH** the current next_ticket value and **INC** it for next thread, and it **SHOULD BE UNIQUE!!!!**

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket;
    local_next_ticket = fetch_and_inc(&lock->next_ticket)
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **Yes!**
WHY?
You have to **FETCH** the current next_ticket value and **INC** it for next thread, and it **SHOULD BE UNIQUE!!!!**

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
      int local_next_ticket;
      local_next_ticket = fetch_and_inc(&lock->next_ticket)
// wait until my sequence number is equal to now_serving
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **No!**
WHY?
We only need to read and wait to have the now_serving equal to my UNIQUE ticket

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

(a) `fetch_and_inc` atomic operation:

```
int fetch_and_inc(int *addr);
```

Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket;
    local_next_ticket = fetch_and_inc(&lock->next_ticket)
    while (lock→now_serving!=local_next_ticket);
}
```

DO WE NEED TO DO IT IN AN ATOMIC WAY? **No!**
WHY?
We only need to read and wait to have the now_serving equal to my UNIQUE ticket

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   ```
   int fetch_and_inc(int *addr);
   ```

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket;
    local_next_ticket = fetch_and_inc(&lock->next_ticket)
    while (lock→now_serving!=local_next_ticket);
}
```

How can we do that with:
load_linked and
store_conditional ?

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

(b) `load_linked` and `store_conditional` operations:

```
int load_linked (int *addr);
int store_conditional (int *addr, int value);
```

Recall that `store_conditional` returns 0 in case it fails or 1 otherwise.

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

Load value of my possible **UNIQUE** ticket

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket; int res=0;
    do {
        local_next_ticket = load_linked(&lock→next_ticket);
    } while (store_conditional(&lock→next_ticket, local_next_ticket +1)==0);
    while (lock→now_serving!=local_next_ticket);
}
```

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

(b) `load_linked` and `store_conditional` operations:

```
int load_linked (int *addr);
int store_conditional (int *addr, int value);
```

Recall that `store_conditional` returns 0 in case it fails or 1 otherwise.

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket; int res=0;
    do {
        local_next_ticket = load_linked(&lock→next_ticket);
    } while (store_conditional(&lock→next_ticket, local_next_ticket +1)==0);
    while (lock→now_serving!=local_next_ticket);
}
```

Try to increment value next ticket ... but if fail (returns 0)... we should try again to get next ticket!!! OTHERWISE IT WOULD **NOT BE UNIQUE!**

If not fail (returns 1) we stop waiting, we have oure **UNIQUE** Ticket

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

(b) `load_linked` and `store_conditional` operations:

```
int load_linked (int *addr);
int store_conditional (int *addr, int value);
```

Recall that `store_conditional` returns 0 in case it fails or 1 otherwise.

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

```
void ticket_lock_acquire (tTicket_lock * lock) {
    int local_next_ticket; int res=0;
    do {
        local_next_ticket = load_linked(&lock→next_ticket);
    } while (store_conditional(&lock→next_ticket, local_next_ticket +1)==0);
    while (lock→now_serving!=local_next_ticket);
}
```

Once we have obtain the ticket and increment next... we only need to wait

Problem 2 (4 points) A *ticket lock* is a lock implemented using two shared counters, `next_ticket` and `now_serving`, both initialised to 0. A thread wanting to acquire the lock uses an atomic operation to fetch the current value of `next_ticket` as its unique sequence number and increments it by 1 to generate the next sequence number. The thread then waits until `now_serving` is equal to its sequence number. Releasing the lock consists on incrementing `now_serving` in order to pass the lock to the next waiting thread.

Given the following data structure and incomplete implementation of the primitives that support the `ticket lock` mechanism:

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
    lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
    // obtain my unique sequence number from next_ticket
    // generate the next_ticket sequence number
    // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
    lock->now_serving++;
}
```

DO WE NEED TO DO IT IN ATOMIC WAY?

We ask:

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

Problem 2 (4 points) A *ticket lock* is a lock implemented using two shared counters, `next_ticket` and `now_serving`, both initialised to 0. A thread wanting to acquire the lock uses an atomic operation to fetch the current value of `next_ticket` as its unique sequence number and increments it by 1 to generate the next sequence number. The thread then waits until `now_serving` is equal to its sequence number. Releasing the lock consists on incrementing `now_serving` in order to pass the lock to the next waiting thread.

Given the following data structure and incomplete implementation of the primitives that support the `ticket lock` mechanism:

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

DO WE NEED TO DO IT IN ATOMIC WAY?
No, there is only one doing that, **THE ONE THAT IS IN THE EXCLUSIVE AREA!**

We ask:

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations: