CS 3410 Fall 2025

Lab 11 Worksheet: Concurrent Hash Tables

1. Atomic Increment

Part 1A

Study the function below, which atomically adds 1 to an integer variable in memory and returns its original value.

```
int atomic_increment(volatile int* var) {
    int prev;
    __asm__ volatile(
        "1: lr.w.aqrl %0, (%1)\n"
        "   addi %0, %0, 1\n"
        "   sc.w.aqrl t0, %0, (%1)\n"
        "   bne  t0, zero, 1b\n"  // Jump to label 1 - retry until sc succeeds
        : "=&r"(prev)
        : "r"(var)
        : "t0", "memory");
    return prev;
}
```

Under what conditions will store-conditional (`sc`) fail? When will it succeed?

Part 1B

Using "ordinary" loads and stores cannot guarantee that memory updates will be visible to other threads in order. Study the function below.

```
int simple_increment(volatile int* var) {
    int prev;
    __asm__ volatile(
        "lw   %0, (%1)\n"
        "addi %0, %0, 1\n"
        "sw   %0, (%1)\n"
        : "=&r"(prev)
        : "r"(var)
        : "t0", "memory");
    return prev;
}
```

Two processors, A and B, are running this program concurrently.
Give an example execution order that results in an incorrect result (i.e. The result is not `prev + 2`.)

Part 1C

Study the partially implemented function below, which implements a Compare-and-Swap (CAS) operation that atomically compares the current value of an integer with an expected value. If they are equal, it updates the value of the integer to a new value. Complete the function below using LR/SC.

```
bool compare_and_swap(volatile int* var, int old, int new) {
    int prev_old;
    int SUCCESS = (a)_________;
    int FAIL = 1;
    __asm__ volatile(

        "1: (b)_______________ %0, (%1)\n"
        "            bne  %0, %2, 2f\n"

        "   (c)_______________ t0, %3, (%1)\n"
        "            bne  t0, zero, 1b\n"

        "            li   %0, (d)_____________________\n"
        "            j    3f\n"

        "2:          li   %0, (e)_____________________\n"
        "3:          \n"
        : "=&r"(prev_old)
        : "r"(var), "r"(old), "r"(new)
        : "t0", "memory");
    return prev_old == (f)_____________________;
}
```

Explain briefly why CAS operations are used in concurrent programming.

2. Locks

Part 2A

What is the purpose of a lock in a multithreaded program?

CS 3410 Fall 2025

Lab 11 Worksheet: Concurrent Hash Tables

Part 2B

Study the function below which adds 1 to the `counter`.

```
int counter = 0;
int lock = 0;

void simple() {
  if (!lock) { lock = 1; counter++; lock = 0; }
}
```

Describe a situation in which the function, which uses an if statement, leads to two threads entering the critical section at the same time. How can this be avoided?

Part 2C

When a thread is "spinning," what is it actually doing?

```
A.  Sleeping until the lock is free
B.  Checking again and again in a loop
C.  Sending a signal to the OS
```

3. Spinlocks Reasoning

Suppose we have two threads, A and B, sharing this code:

```
spin_lock(&lock);
counter = counter + 1;
spin_unlock(&lock);
```

What could go wrong if we remove the spinlock?

What does the spinlock guarantee about the variable `counter`*?*

Lab 11 Worksheet: Concurrent Hash Tables

4. Condition Variables

Study the function below, which attempts to block until `counter >= threshold`*.*
Note: you are not allowed to use `pthread.h` *in A11. Instead, you will build your own implementation of spinlocks and condition variables.*

```
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int counter = 0;
int threshold = 5;

void wait_for_condition() {
  pthread_mutex_lock(&lock);
  int before = counter;
  if (counter < threshold) { pthread_cond_wait(&cond, &lock); }
  int after = counter;
  printf("Before wait: %d, After wait: %d\n", before, after);
  pthread_mutex_unlock(&lock);
}
```

Now, another thread sets `counter` *to 5:*

```
void increment() {
  pthread_mutex_lock(&lock);
  counter = 5;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&lock);
}
```

Part 4A

Assume no other threads call `pthread_cond_signal`*.*
Is it necessarily true that `wait_for_condition` *blocks until* `counter >= threshold`*? Explain briefly.*

Part 4B

Regarding the print statement, are the values of `before` *and* `after` *necessarily different? Explain briefly.*

Part 4C

If you answered no to previous parts, suggest how `wait_for_condition` *could be modified to ensure* `before` *and* `after` *are different.*