CS-453 - Project
(A short overview of) Concurrent Programming in C/C++

Distributed Computing Laboratory

September 16, 2025

Back to CS101

```
// a single thread

int a = 0;
int b = 0;
print(a, b);       // a = 0, b = 0

a = 1;
print(a, b);       // a = 1, b = 0

b = 1;
print(a, b);       // a = 1, b = 1
```

What if we have two threads?

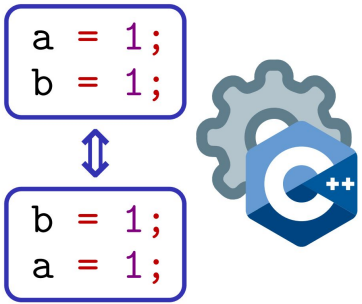
```
// Global var.

int a = 0;
int b = 0;

// Thread A

a = 1; // write
b = 1; // write
```

```
// Thread B

int v = b; // read
if (v==1) {
     print(a, v);
     // a = 1, v = 1?
     // a = 1, v = 0?
     // a = 0, v = 1?
     // a = 0, v = 0?
}
```

According to common sense / “sequential consistency”	What will happen in C/C++	What could happen according to the C/C++ standards
		Format your disk

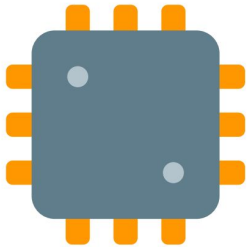
Who are the culprits?

1) **C/C++ compilers** can **reorder** instructions if it **doesn't** have any **local side effects**.



2) C/C++ standards say **accessing a variable** that is being written by another thread **without synchronization** (data race) is an **Undefined Behavior**, it can lead to absolutely **anything**.

3) **CPUs**, depending on their consistency model, can execute **unrelated operations out-of-order**.



When coding in C/C++, you should only care about the **C/C++ model, forget about hardware promises**!

The main takeaway

C/C++ do **NOT** ensure (without extra care)

that reads/writes

are carried/observed

in program order

by **different threads**

Use synchronization primitives when sharing data across threads to restore sequential consistency!

Example: let's build a concurrent counter

```
#include <pthread.h>
#include <assert.h>

static int counter = 0;

void* thread(void* null) {
  counter = counter + 1; // race condition
}
```

```
int main() {
  pthread_t handlers[2];
  for (int i = 0; i < 2; i++)
    pthread_create(&handlers[i], NULL, thread, NULL);
  for (int i = 0; i < 2; i++)
    int res = pthread_join(handlers[i], NULL);
  assert(counter == 2);
}
```

Let's try to fix this example by using synchronization primitives!

Sync primitive #1: Locks/Mutexes

- No two threads can hold a lock concurrently.
- Lock before accessing shared variable to **prevent data races**. (Don't forget to unlock.)
- **Prevent reordering** via **fences** and ensure **sequential consistency**.

```
#include <pthread.h>
#include <assert.h>

pthread_mutex_t mutex;
static int counter = 0;

void* thread(void* null) {
  pthread_mutex_lock(&mutex);
  counter = counter + 1;
  pthread_mutex_unlock(&mutex);
}
```

```
int main() {
  pthread_mutex_init(&mutex, NULL);
  pthread_t handlers[2];
  for (int i = 0; i < 2; i++)
    pthread_create(&handlers[i], NULL, thread, NULL);
  for (int i = 0; i < 2; i++)
    int res = pthread_join(handlers[i], NULL);
  assert(counter == 2);
  pthread_mutex_destroy(&mutex);
}
```

Be careful about deadlocks! (e.g., always lock in the same order)

Sync primitive #2: Atomic variables (1/2)

- **Safe concurrent access** from multiple threads (no data races)
- Provide **atomic operations** (i.e., no other thread can observe partially-completed ops):
  - read (atomic_load) / write (atomic_store)
  - increment (atomic_fetch_add) / compare and swap (atomic_compare_exchange_strong)
- (By default,) prevent reorderings and offer **sequential consistency**.

```
#include <pthread.h>
#include <assert.h>
#include <stdatomic.h>

static atomic_int counter = 0;

void* thread(void* null) {
  atomic_fetch_add(&counter, 1);
}
void* bad_thread(void* null) {
  counter = counter + 1; // 2 ATOMIC OPERATIONS (LOAD and STORE) INSTEAD OF 1
}
```

```
int main() {
  pthread_t handlers[2];
  for (int i = 0; i < 2; i++)
    pthread_create(&handlers[i], NULL, thread, NULL);
  for (int i = 0; i < 2; i++)
    int res = pthread_join(handlers[i], NULL);
  assert(counter == 2);
}
```

Sync primitive #2: Atomic variables (2/2)

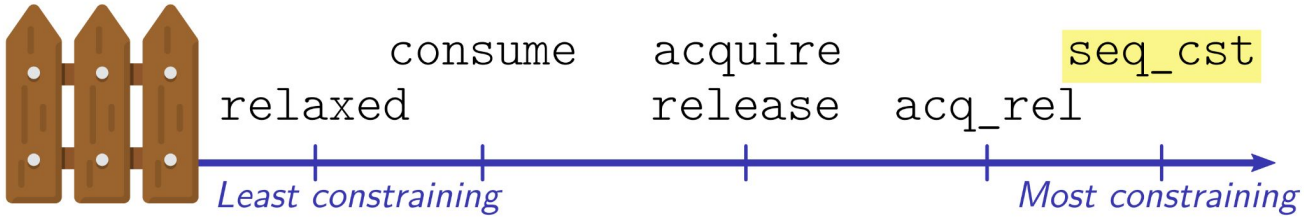
Atomic variables can be used to implement locks using the “Compare and Swap” operation

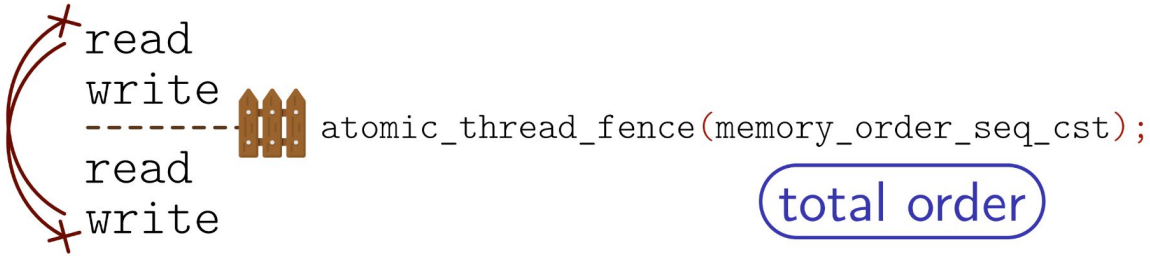
```
#include <stdatomic.h>


#define UNLOCKED 0
#define LOCKED 1

struct lock {
  atomic_bool state;
};

void init_lock(struct lock* lock) {
  lock->state = UNLOCKED;
}
```

```
void take_lock(struct lock* lock) {
  while (true) {
    bool expected = UNLOCKED;
    atomic_compare_exchange_strong(
       &lock->state, &expected, LOCKED);
    if (expected == UNLOCKED) break;
  }
}


void release_lock(struct lock* lock) {
  lock->state = UNLOCKED;
}
```

- Busy waiting can seriously harm performance. Cooperate with your scheduler.
- 99.99% of the time: use the locks provided by your platform.

Sequential Consistency is a strict ordering





- Sequential Consistency prevents all reordering and can become a bottleneck.
- You can make your program more efficient by allowing some reordering.
- Very tricky to reason about + you probably won't need it for this class. :)
- https://en.cppreference.com/w/c/atomic/memory_order

Takeaways

- **Never access data** while it is **being modified** by another thread.
- Option #1, **atomic variables**:
  - Few operations (read/write/f&a/c&s)
  - Multiple operations: not atomic! (but no data race)
- Option #2, **locks**:
  - Lock before accessing shared data, unlock after
  - Arbitrary logic
  - Be careful about deadlocks!
  - Do not use your own implementation!
- Sequential consistency is an overkill you can tolerate