CS-453 - Project

(An overview of) Concurrent Programming in C/C++

Distributed Computing Laboratory

September 26 2023

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?

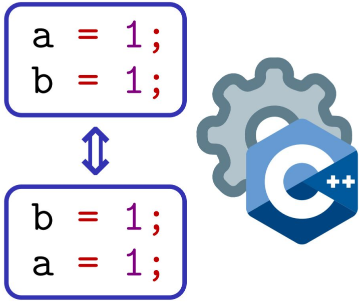
```
// Global var.

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

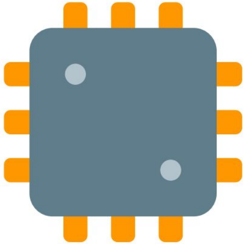
Who are the culprits?

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



2) According to the C/C++ standards, **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) Your **CPU**, depending on its consistency model, can execute **unrelated R/W** (& R/R, W/W) **out-of-order**.



But, when programming in C/C++, you shouldn't have to care about what your hardware promises, only the C/C++ standards.

The main takeaway

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

that reads/writes

are carried/observed

in program order

by **different threads**

You need to 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

- A lock (or **Mut**ual **Ex**clusion) can only be held by one thread at a time.
- Use it to **prevent data races** on shared variables.
- It will **prevent reordering** via a **fence** and ensure **sequential consistency**.

```c
#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);
}
```

```c
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);
}
```

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

- An atomic variable can **safely** be **accessed concurrently** from multiple threads (no data races)
- They offer **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,) They prevent reorderings and offer **sequential consistency**.

```c
#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
}
```

```c
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);
}
```

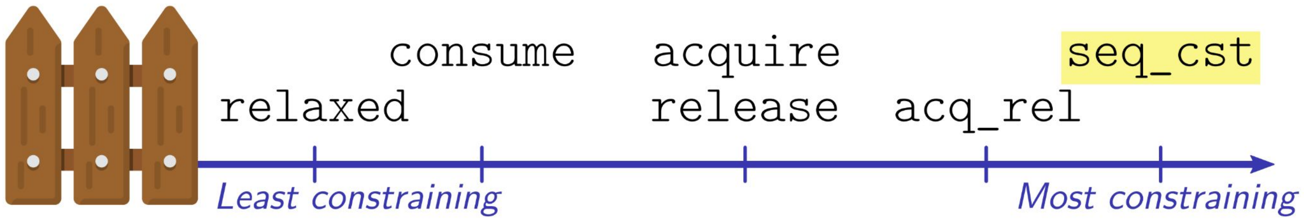
Sync primitive #2: Atomic variables (2/2)

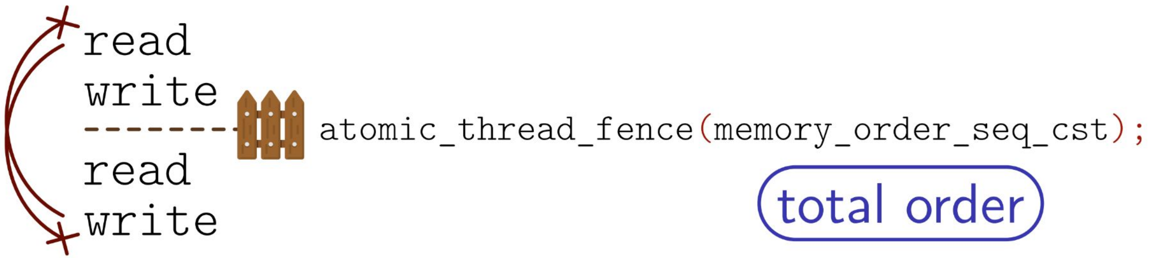
Atomic variables can be used to implement locks using the “Compare and Swap” operation

```
#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;
}
```

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.
- But it gets tricky to reason about, and you probably won't need it for this class. :)
- https://en.cppreference.com/w/c/atomic/memory_order