COMP 737011 - Memory Safety and Programming Language Design

Lecture 5: Concurrency

XU, Hui

xuh@fudan.edu.cn

Outline

1. Risks of Concurrent Programs
2. Atomicity and Lock
3. Synchronization and Memory Barrier

1. Risks of Concurrent Programs

Risks of Concurrent Programs

- Data race or shared access
- Deadlocks
- Out-of-order execution
  - Compiler issue
  - CPU issue

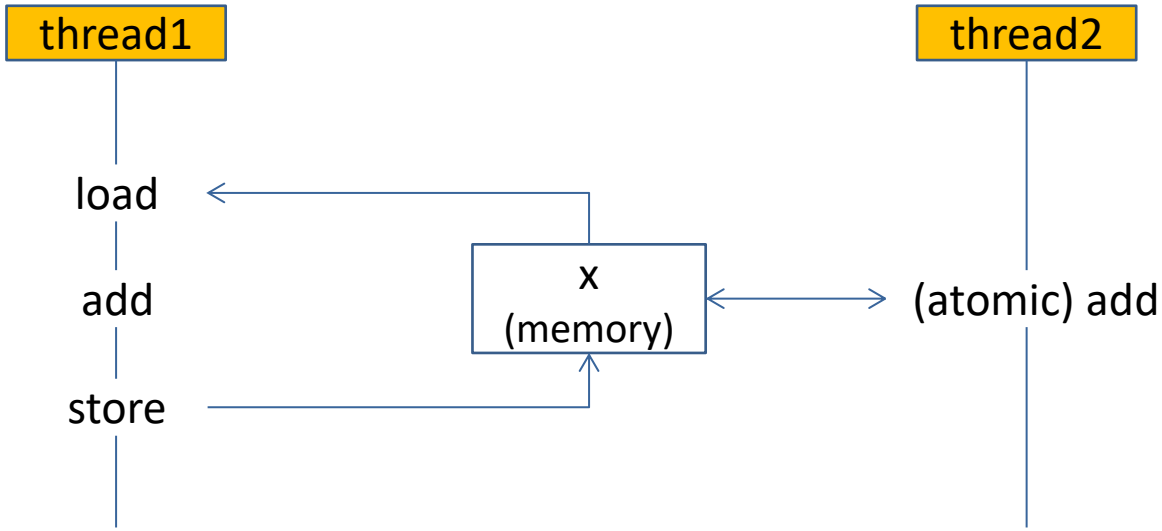
An Example of Data Race

```
#define NUM 100
int global_cnt = 0;

void *mythread(void *in) {
    for (int i=0; i<NUM; i++)
        global_cnt++; //concurrently accessed by multiple threads
}

int main(int argc, char** argv) {
    pthread_t tid[NUM];
    for (int i=0; i<NUM; i++){
        assert(pthread_create(&tid[i], NULL, mythread, NULL)==0);
    }
    for (int i=0; i<NUM; i++){
        pthread_join(tid[i], NULL);
    }
    assert(global_cnt==NUM*NUM); //assertion could fail!
}
```

Hint of experiments: do not turn on optimization

Typical Scenario of Data Race

- Multiple threads access the same memory unit concurrently
- At least one access is nonatomic (write)
- For example, add operation is not atomic in X86 (CISC)
  - load-add-store (multiple instructions or micro ops)



X86 vs ARM/RISC-V

```
void toy(int x, int y){
   int z = x+y;
}
```

X86: operands can be mem

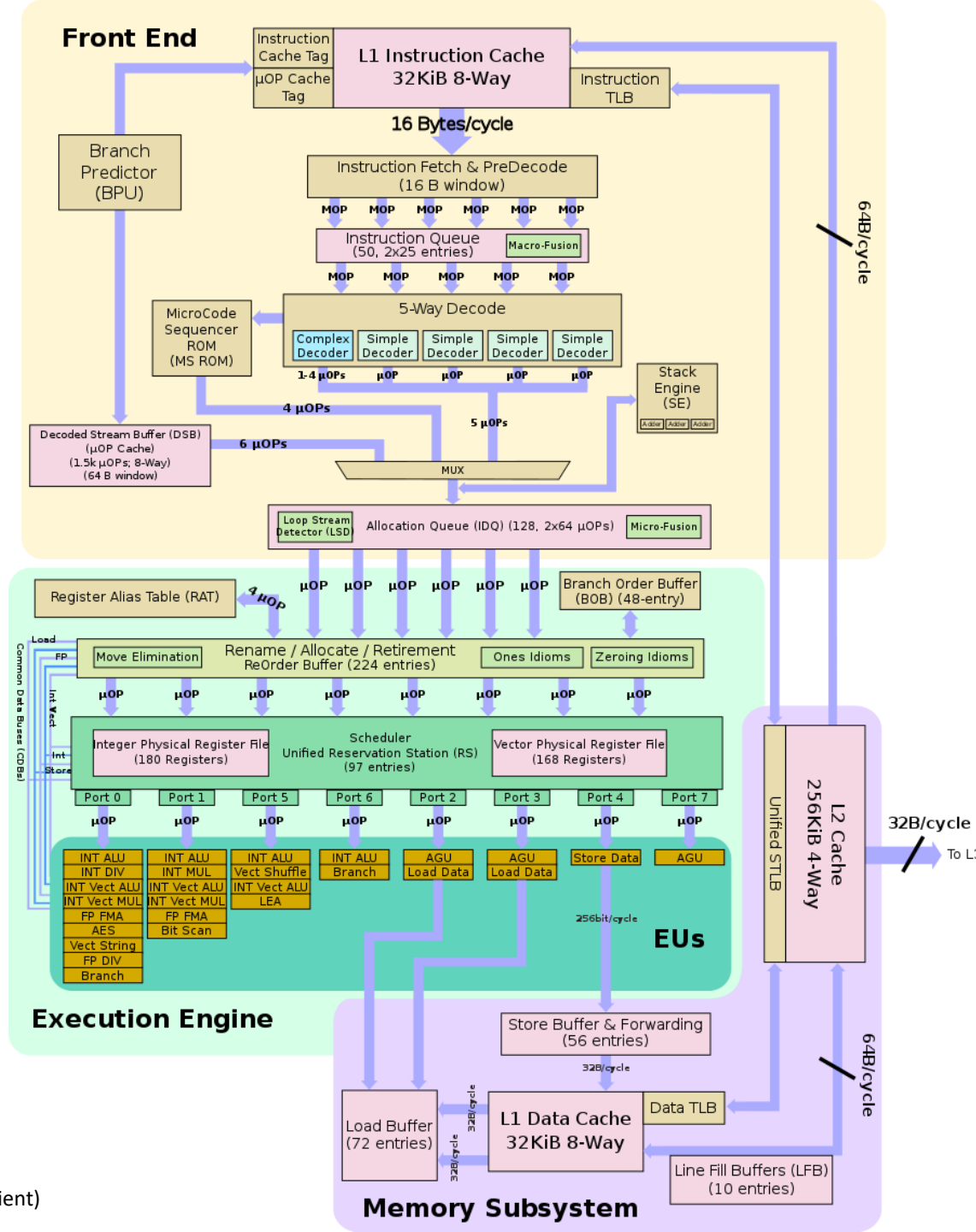
```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-0x4],edi
mov     DWORD PTR [rbp-0x8],esi
mov     eax, DWORD PTR [rbp-0x4]
add     eax, DWORD PTR [rbp-0x8]
mov     DWORD PTR [rbp-0xc], eax
pop     rbp
ret
```

RISC-V: only registers can be used as operands

```
addi    sp, sp, -32
sd      ra, 24(sp)
sd      s0, 16(sp)
addi    s0, sp, 32
sw      a0, -20(s0)
sw      a1, -24(s0)
lw      a0, -20(s0)
lw      a1, -24(s0)
addw    a0, a0, a1
sw      a0, -28(s0)
ld      ra, 24(sp)
ld      s0, 16(sp)
addi    sp, sp, 32
ret
```

Superscalar

- Micro ops
- Out-of-order execution
- Branch prediction
- …



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Question

- If the program runs on a single CPU core, will it still suffer race condition?

```
taskset -c 0 ./a.out
```

```
taskset -c 0 bash -c 'for i in {1..1000}; do ./a.out; done'
```

2. Atomicity and Lock

Atomic Instructions

- One instruction directly operates on the memory
  - Do not load the variable to the register
- Lock prefix guarantees atomicity of Micro Ops
  - X86 provides a “lock” prefix to achieve atomicity

```
mov     eax, DWORD PTR [rbp-0x14]
add     eax, 0x1
mov     DWORD PTR [rbp-0x14],eax
```

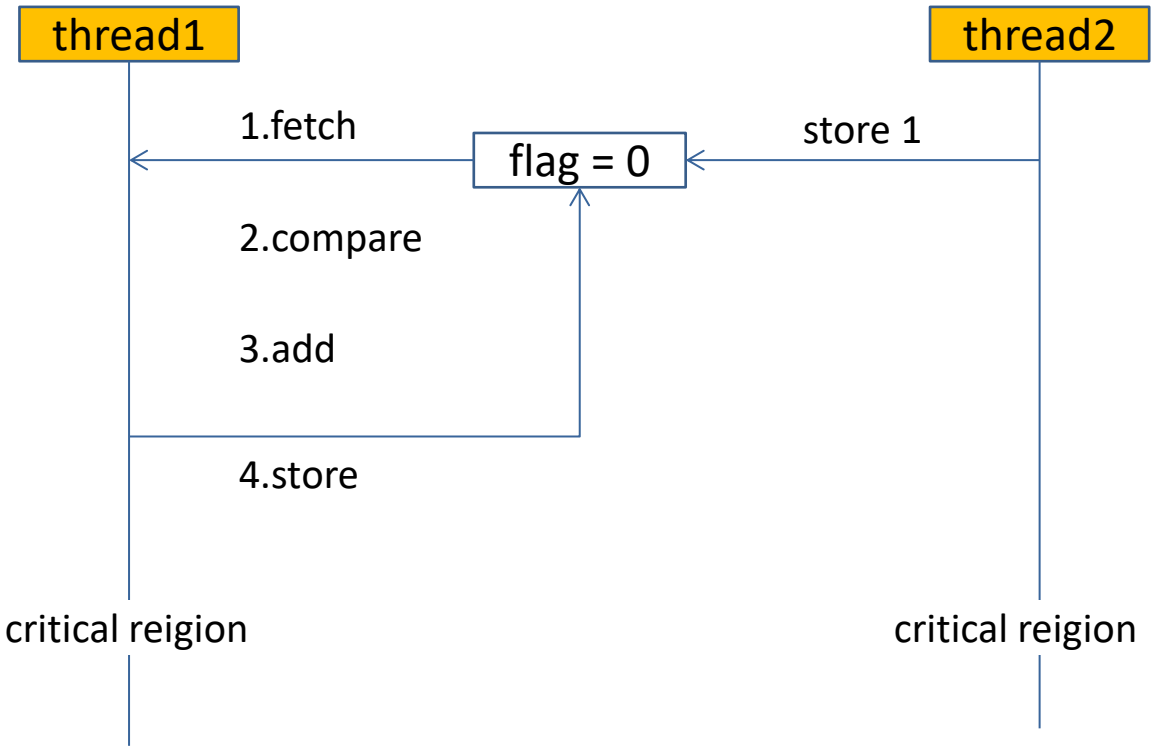
```
lock add DWORD PTR [rip+0x2ed3],0x1
```

Atomic Version

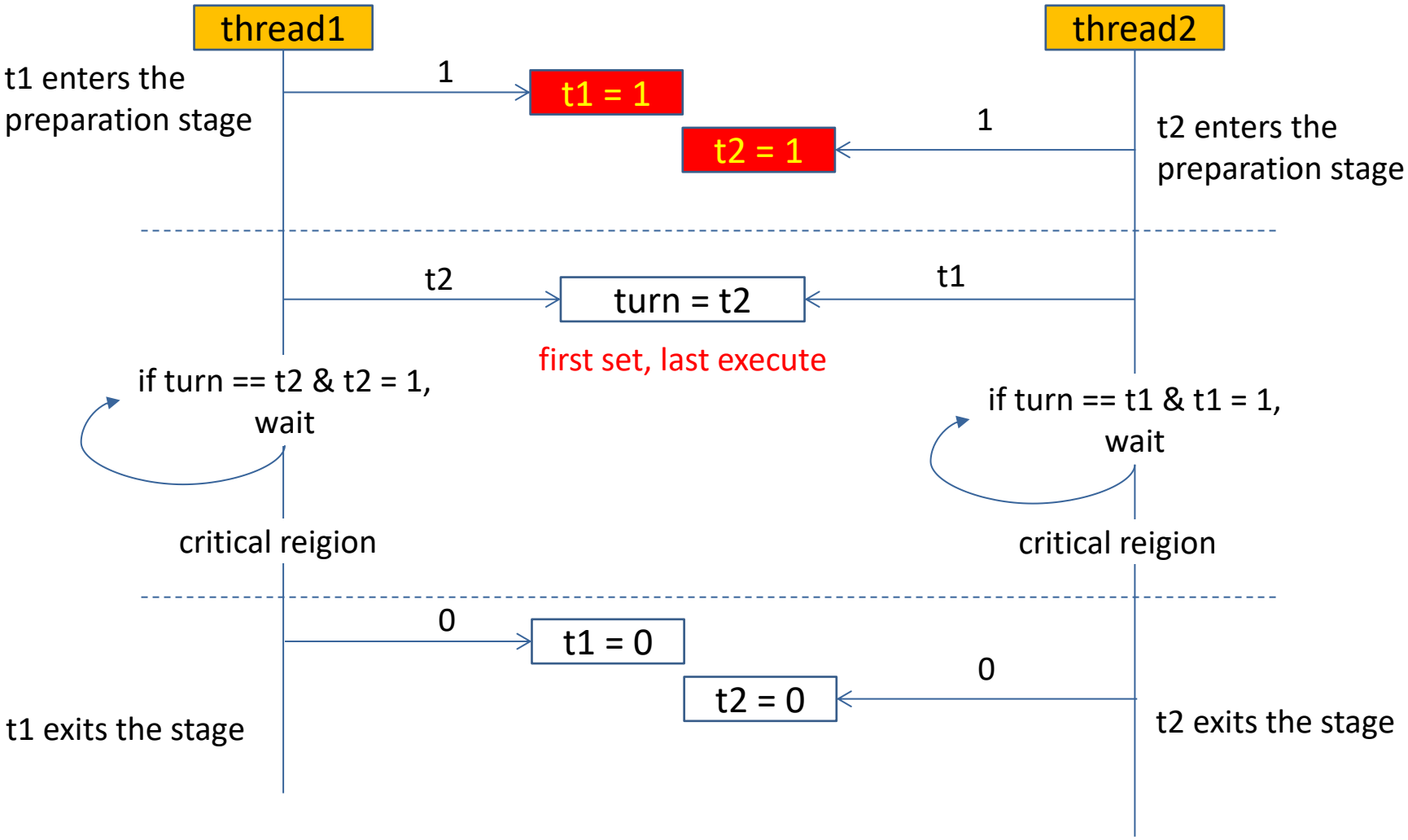
```
#define NUM 100
atomic_int global_cnt; //declaring the variable as atomic

void *mythread(void *from) {
    //use atomic API
    //__atomic_fetch_add(&global_cnt, 1, __ATOMIC_SEQ_CST);
    for (int i=0; i<NUM; i++)
        global_cnt++;
}

int main(int argc, char** argv) {
    pthread_t tid[NUM];
     for (int i=0; i<NUM; i++){
        assert(pthread_create(&tid[i], NULL, mythread, NULL)==0);
    }
    for (int i=0; i<NUM; i++){
        pthread_join(tid[i], NULL);
    }
    assert(global_cnt==NUM*NUM);
}
```

Mutual Exclusion or Mutex

- How to achieve atomicity for a sequence of code?
  - Entering the critical region without interference
- It is impossible with the "lock add" instruction.



Peterson Algorithm's for Mutex



Sample Code of Peterson's Algorithm

```
void* t0(void *from) {
    flag[0] = true;
    turn = 1;
    while(flag[1]==true && turn==1)
        sleep(1);
    do_critical();
    flag[0] = false;
}
```

Give way to t1

```
void* t1(void *from) {
    flag[1] = true;
    turn = 0;
    while(flag[0]==true && turn==0)
        sleep(1);
    do_critical();
    flag[1] = false;
}
```

Give way to t0

Mutex base on Atomic Instructions

- How to achieve atomic compare and set/swap?
  - x86 instruction: cmpxchg
  - C API: atomic_compare_exchange_strong

```
# based on rax
lock cmpxchg dst src
```

sematic

```
eax = 0, src = 1
if(dst == eax) {
    dst = src;
}
else {
    eax = dst;
}
```

- dst is the lock flag
- rax is the return value of check

```
atomic_compare_exchange_strong(&dst, &test, src)
```

exactly the same with cmpxchg

Type of Locks: Mutex Lock vs Read-Write Lock

- Mutex lock: only one thread can access a shared resource at a time.
  - Lock Acquisition (lock) – A thread requests ownership of the mutex.
  - Critical Section – The thread executes its operations on the shared resource.
  - Lock Release (unlock) – The thread releases the mutex, allowing others to proceed.
- Read-write lock: multiple threads to read concurrently; only one thread to write exclusively.
  - Unlocked – No threads hold the lock.
  - Read(n) – Multiple threads can read at the same time (n readers).
  - Write (exclusive) – Only one thread can write, and no readers are allowed.

Type of Locks: Spin Lock

- Thread tries to acquire the lock
- If the lock is free, it proceeds
- If the lock is held, the thread keeps checking (spinning) until it’s released

Type of Locks: Pessimistic vs Optimistic Lock

- Pessimistic Lock: Assumes that conflicts will occur, so it locks the resource before accessing it, e.g., mutex lock.
  - A thread acquires a lock before accessing shared data.
  - Other threads must wait until the lock is released.
- Optimistic Lock: Assumes conflicts are rare, so it accesses the resource without locking.
  - Read the data.
  - Do some work.
  - Check if the data has changed (validation).
    - If unchanged, commit the update.
    - If changed by another thread, retry or abort.

Question

- Is shared_ptr of C++ thread-safe?
  - reference counter
  - data read/write

3. Synchronization and Memory Barrier

Out-of-Order Execution

- Compiler reordering during optimization
- CPU out-of-order execution
- This lecture focuses on compile-time reordering

Compiler Reordering

- Supposing optimization (*e.g.,* -O2 or O3) is enabled, the compiler might make mistakes.

```
int a = 1;
```

```
void *t0 (void* in){
    while (a);
}
```

```
void *t1 (void* in){
    a = 0;
}
```

```
0x00401150 <+0>:    cmp    DWORD PTR [rip+0x2ee9], 0x0
0x00401157 <+7>:    je     0x401162 <t0+18>
0x00401159 <+9>:    nop    DWORD PTR [rax+0x0]
0x00401160 <+16>:   jmp    0x401160 <t0+16>
0x00401162 <+18>:   ret
```

infinite loop

Another Example

- The following assertion could fail on some platforms if the execution order cannot be guaranteed.

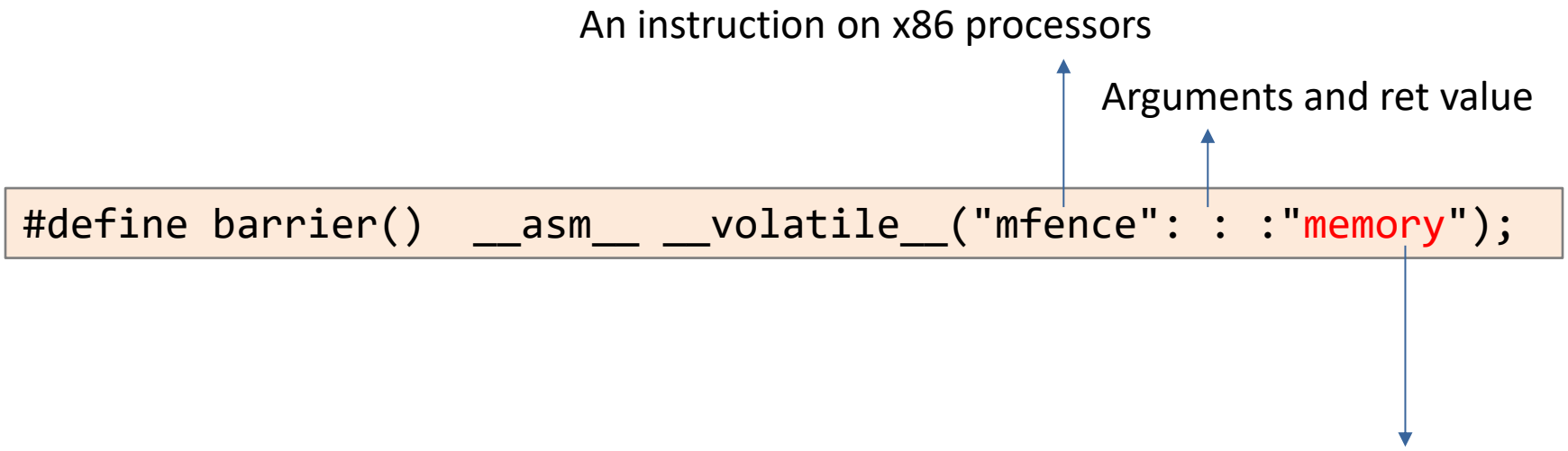
```
atomic_int a = 1;
atomic_int b = 1;
```

```
void *t0 (void* in){
    a = 0;
    b = 0;
}
```

```
void *t1 (void* in){
while(!b);
    assert(!a);
}
```

https://stackoverflow.com/questions/48139399/memory-order-relaxed-not-work-as-expected-in-code-from-c-concurrency-in-action

Use Memory Barrier (Fence)



Example

- Discard all variable values on registers
  - Reload them from memory
- Guarantee happens-before: operations prior to the barrier are always executed before operations after the barrier.

```
void *t0 (void* in){
    while (a)
        barrier();
}
```

```
void *t0 (void* in){
    a = 0;
    barrier();
    b = 0;
}
```

Relax the Synchronization Requirement

- We only want some variables to be updated.
- Use volatile when declaring a variable.
- Do not prevent reordering.

```
volatile int a = 1;
void *t0 (void* in){
    while (a) ;
}
```

Further Relax the Requirement

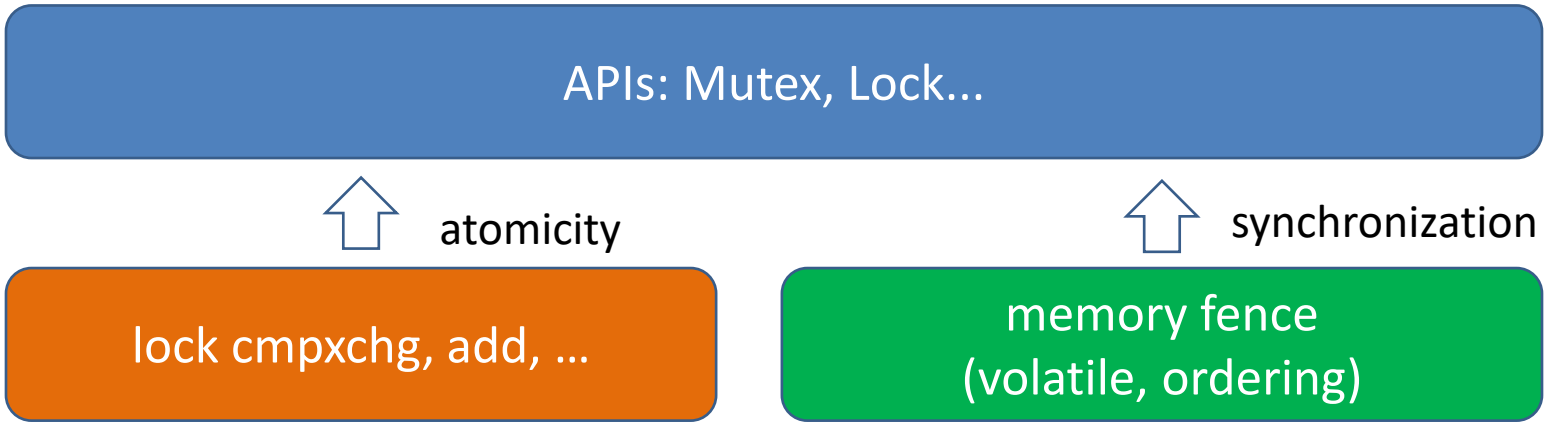
- We want the value to be updated in specific program points.
  - Use ACCESS_ONCE(), which is also based on volatile.
- Further relax the restrictions based on operations:
  - READ_ONCE and WRITE_ONCE()

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
volatile int a = 1;
void *t0 (void* in){
    while (ACCESS_ONCE(a))
        ;
}
```

Relax the Happens-Before Requirement

- Use specific memory ordering
  - Sequential consistency (default on x86)
    - The most strong one, no reordering across the barrier.
  - Acquire-release: commonly used for locks
    - Acquire: no reads or writes can be reordered before this load.
    - Release: no reads or writes can be reordered after this store.
  - Relaxed
    - No synchronization or ordering constraints, only atomicity.

Summary


APIs: Mutex, Lock...
atomicity
synchronization
lock cmpxchg, add, …
memory fence
(volatile, ordering)

In-Class Practice

1) Demonstrate that std::shared_ptr in C++ is not thread-safe.
   - Hint: Design code that triggers an error by causing a race condition in the reference counter update.
2) (Optional) Modify the shared pointer you implemented last week to make it thread-safe.