COMP 737011 - Memory Safety and Programming Language Design

Lecture 6: 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

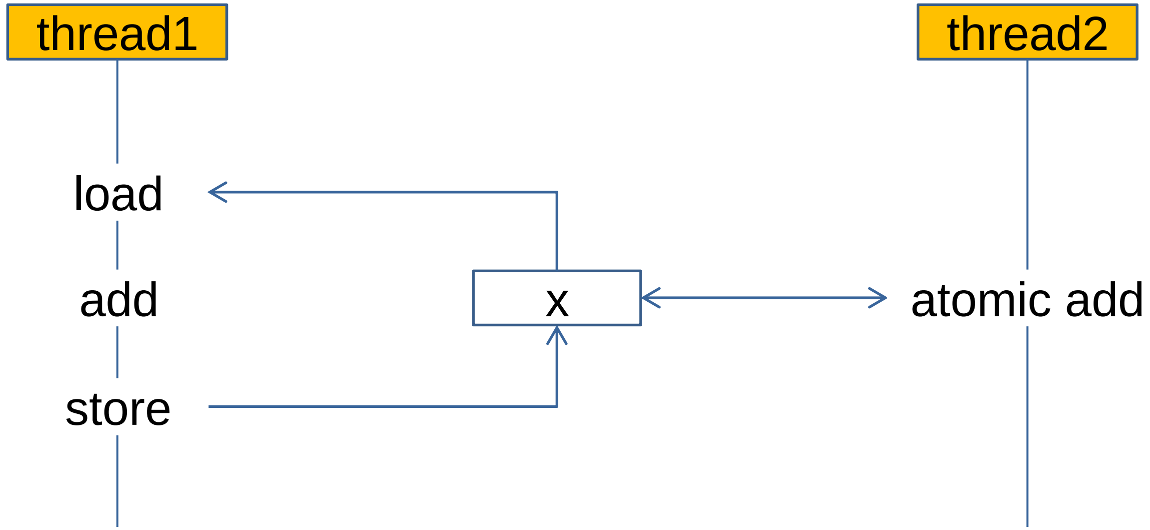
An Example of Data Race

```
#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

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

X86: oprand should 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
```

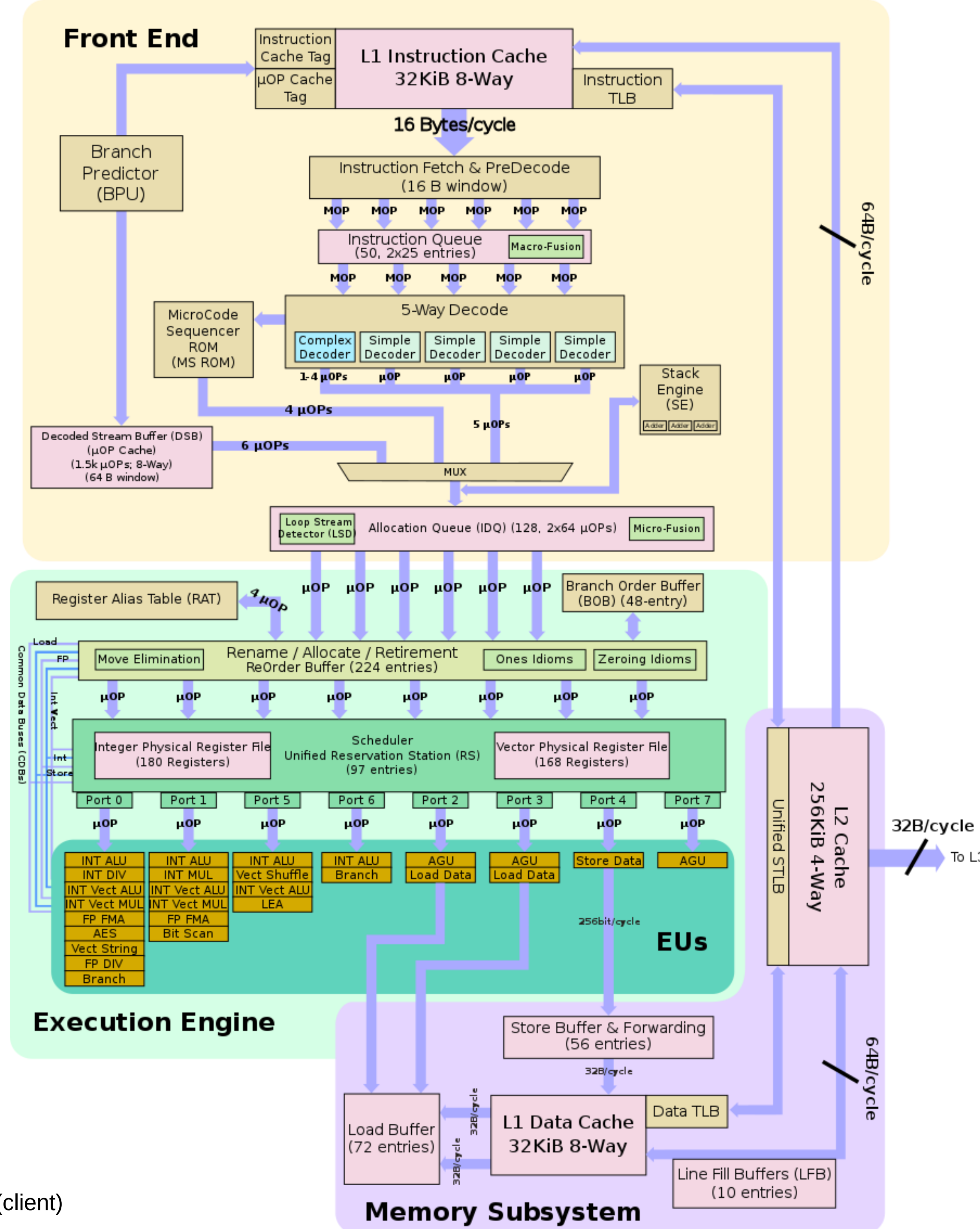
RISC-V: only registers can be used as operands

```
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
```

https://godbolt.org/

Superscalar

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



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

2. Atomicity and Lock

Use 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 for atomicity

```
gef➤  disass mythread
Dump of assembler code for function mythread:
   ...
0x0040117a <+42>:    mov    eax, DWORD PTR [rbp-0x14]
0x0040117d <+45>:    add    eax, 0x1
0x00401180 <+48>:    mov    DWORD PTR [rbp-0x14],eax
   ...
```

⇩

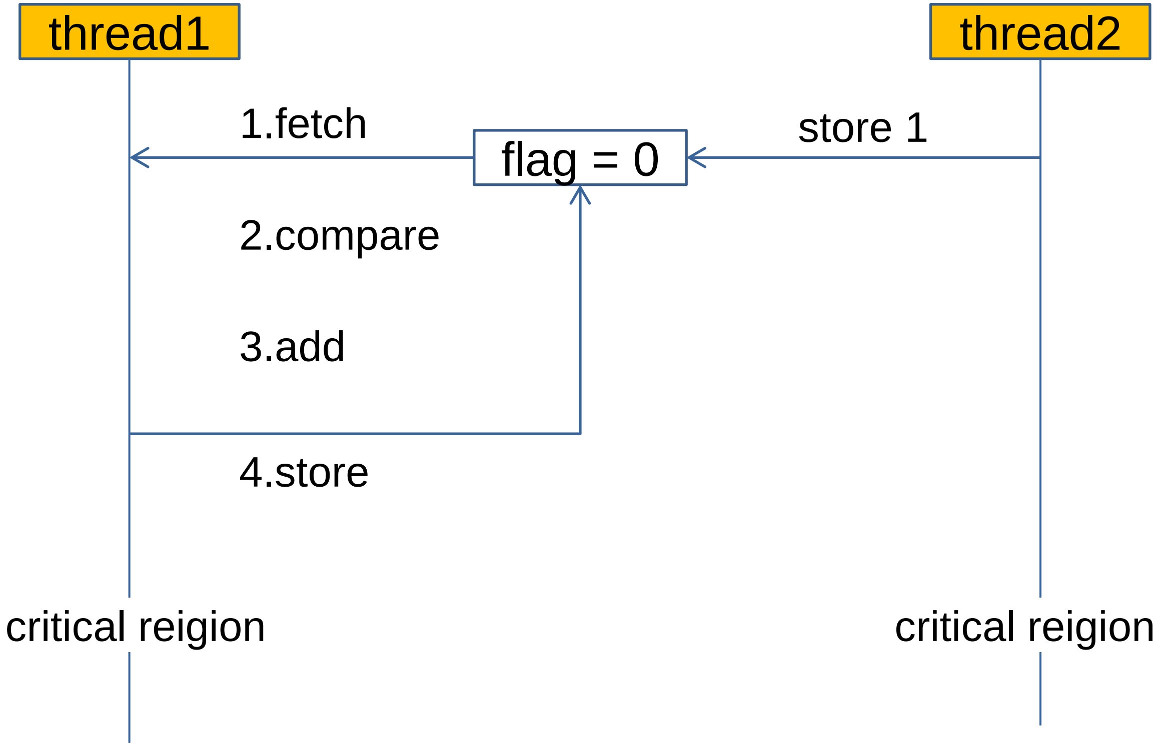
```
lock add DWORD PTR [rip+0x2ed3],0x1
```

Atomic Version

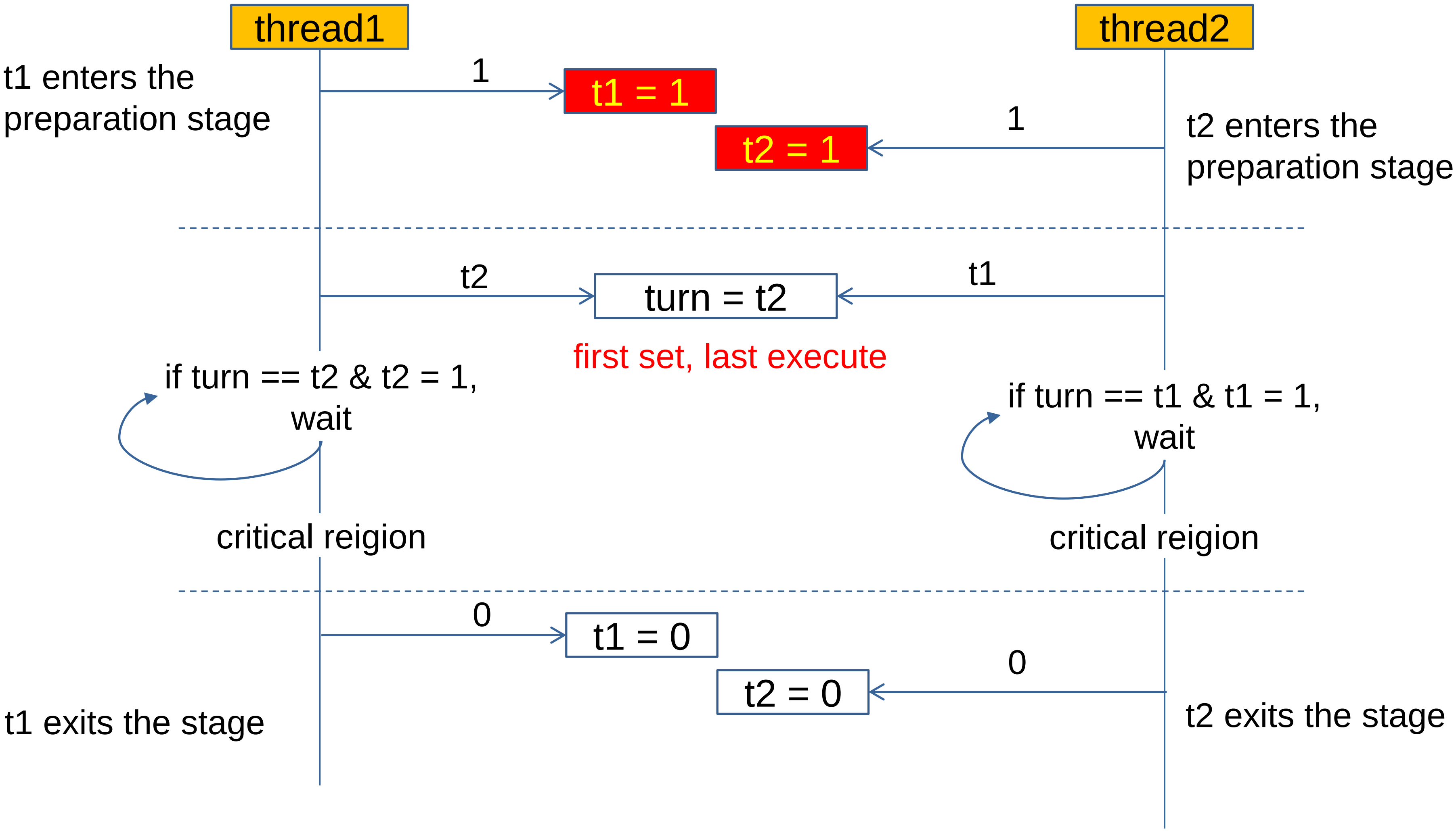
```
#define NUM 100
atomic_int global_cnt; //way 1: by declaring the variable as atomic

void *mythread(void *from) {
    //way 2: 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

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

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

```c
int flag[2], turn;
int x=0;

void do_critical(){
    x++;
}

int main(int argc, char** argv) {
    pthread_t tid[2];
    assert(pthread_create(&tid[0], NULL, t0, NULL)==0);
    assert(pthread_create(&tid[1], NULL, t1, NULL)==0);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

Use 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
```

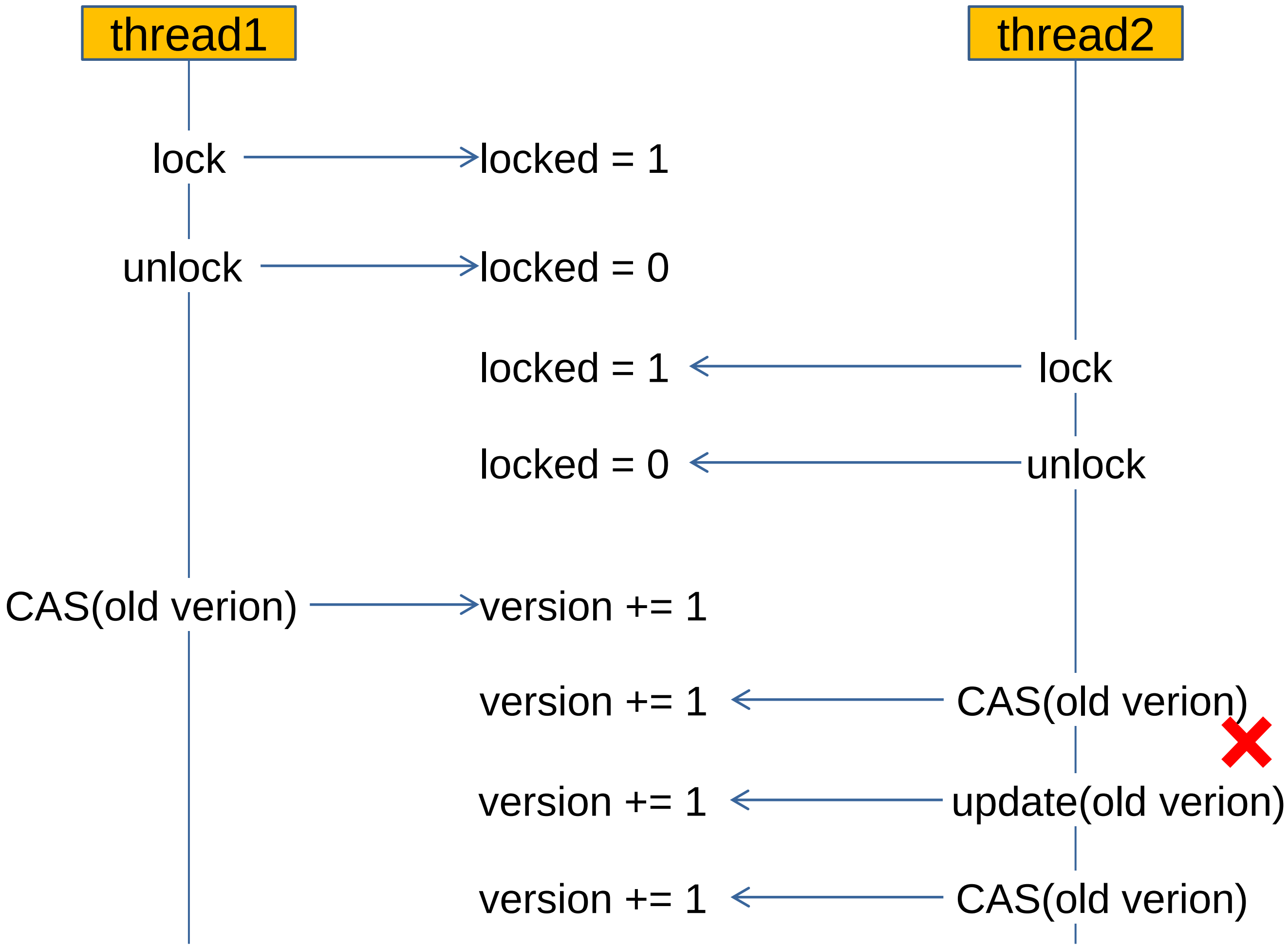
Semantic:

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

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

exactly the same with cmpxchg

Pessimistic Lock vs Optimistic Lock



More Types of Locks

- Mutex lock
- Spin lock: continuously try to acquire the lock
- Read-write lock:
  - State: write/read(n)/unlocked
  - A variable can be read by multiple thread concurrently.
  - Weakness: cannot guarantee serializability of operations

Question

- Is shared_ptr of C++ thread-safe?
  - reference counter
  - data read/write
- Can you implement a thread-safe construct?
- We will learn ARC<Mutex<T>> in Rust

3. Synchronization and Memory Barrier

Out-of-Order Execution

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

Compiler Reordering

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

```
atomic_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)

- 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.

```
#define barrier()  __asm__ __volatile__("": : :"memory");
```

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

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

Relax the Variables

- We only want some variables to be updated:
  - use volatile when declaring a variable
- We only want the variable 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()

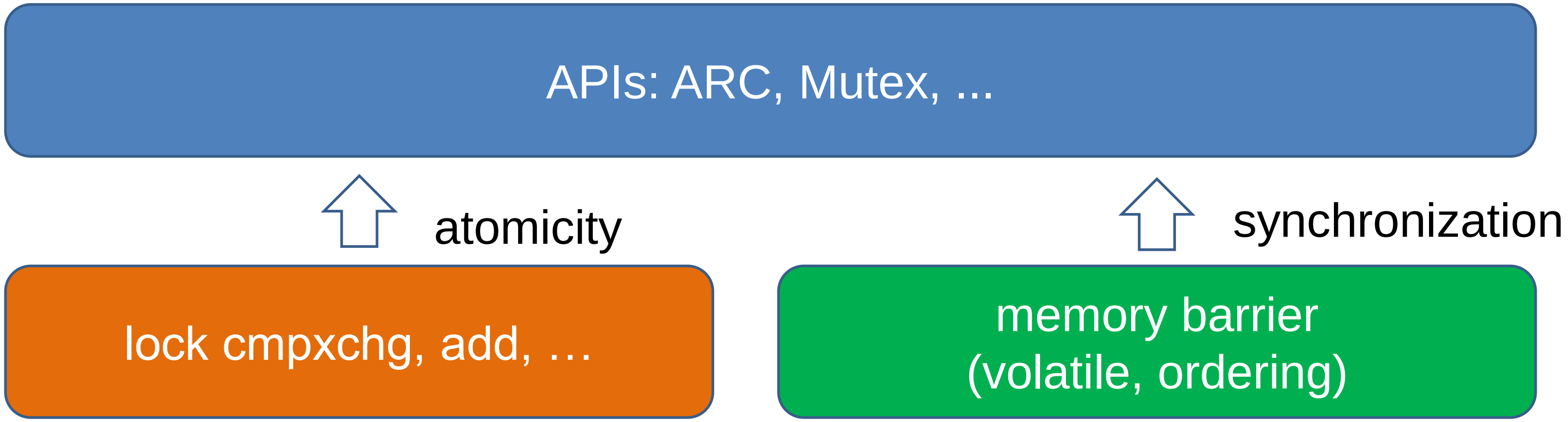
```
volatile int a = 1;
void *t0 (void* in){
    while (a) ;
}
```

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

Relax Happens-Before Requirement

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

Summary


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

In-Class Practice

1) Implement locks in C and and demonstrate the effectiveness
   - a mutex (lock)
   - an optimistic lock
   - based on the atomic_compare_exchange_weak() API, https://en.cppreference.com/w/c/atomic/atomic_compare_exchange
2) Optional: try to implement a thread-safe shared pointer for a data structure with mutex member functions.