COMP 737011 - Memory Safety and Programming Language Design

Lecture 5: Concurrent Access

徐 辉
xuh@fudan.edu.cn

Problem

- Risks of concurrent programs
  - data race or shared access
  - out-of-order execution
    - Compiler issue
    - CPU issue

Outline

- 1. Data race and atomicity
- 2. Instruction reorder and memory barrier

1. Data Race and Atomicity

Warm Up: Data Race

- Add operation is not atomic
  - load-add-store (multiple instructions or micro ops)

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

void *mythread(void *in) {
    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);
}
```

concurrently accessed by multiple threads

assertion could fail

Experimental hint: do not turn on optimization

Atomic Version

way1: by declaring the variable as atomic

```
#define NUM 100
atomic_int global_cnt;

void *mythread(void *from) {
    //__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);
}
```

way2: use atomic API

Mechanism of Atomicity

- 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:
   0x00401150 <+0>:     push   rbp
   0x00401151 <+1>:     mov    rbp,rsp
   0x00401154 <+4>:     mov    QWORD PTR [rbp-0x10],rdi
   0x00401158 <+8>:     mov    DWORD PTR [rbp-0x14],0x0
   0x0040115f <+15>:    cmp    DWORD PTR [rbp-0x14],0x3e8
   0x00401166 <+22>:    jge    0x401182 <mythread+50>
   0x0040116c <+28>:    lock   add DWORD PTR [rip+0x2ed0],0x1
                        # <global_cnt>
   0x00401174 <+36>:    mov    eax,DWORD PTR [rbp-0x14]
   0x00401177 <+39>:    add    eax,0x1
   0x0040117a <+42>:    mov    DWORD PTR [rbp-0x14],eax
   0x0040117d <+45>:    jmp    0x40115f <mythread+15>
   0x00401182 <+50>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00401186 <+54>:    pop    rbp
   0x00401187 <+55>:    ret
```

Mutual Exclusion or Mutex

- How to achieve atomicity for a sequence of code?
  - entering the critical region without interference
- Approaches to implement a mutex
  - based on the cmpxchg instruction of intel CPU
  - based on Peterson algorithm

CAS: Compare and Set/Swap

- How to achieve Atomic CAS?
  - 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

Peterson Algorithm for Mutex

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

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

```
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);
}
```

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

2. Instruction Reorder & Memory Barrier

Out-of-Order Execution

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

Compiler Reordering

- Suppose 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  # <a>
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);
}
```

Note: the assertion does not fail on X86 but may fail on other platforms. Please refer to 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()

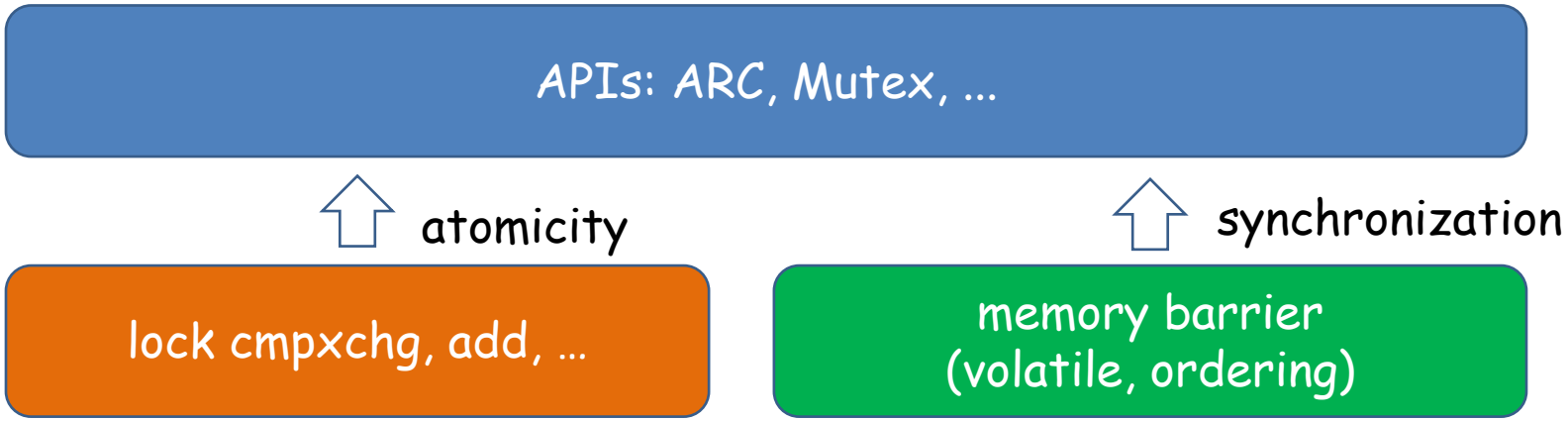
```
volatile int a = 1;
void *t0 (void* in){
    while (a) ;
}
```

```
#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

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

More Reference

- https://www.alibabacloud.com/blog/memory-model-and-synchronization-primitive---part-1-memory-barrier_597460
- https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync
- https://en.cppreference.com/w/cpp/atomic/memory_order
- https://www4.cs.fau.de/Lehre/WS17/V_CS/Uebungen/aufgaben/slides3.pdf
- https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf