Lecture 6.1

并发功能

徐 辉

xuh@fudan.edu.cn

大纲

一、原子访问

二、数据同步

数据竞争问题：多线程的例子

- 加法不是原子操作
  - load-add-store（多条指令或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);
}
```

多个线程并发访问

assertion fail

实验需关闭优化

原子访问

方式一：声明为原子变量类型

```
#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);
}
```

方式二：使用原子运算API

原子类型/访问的实现方式：原子指令

- X86架构：指令翻译时使用带LOCK前缀的指令
- ARM架构：LDREX/STREX指令

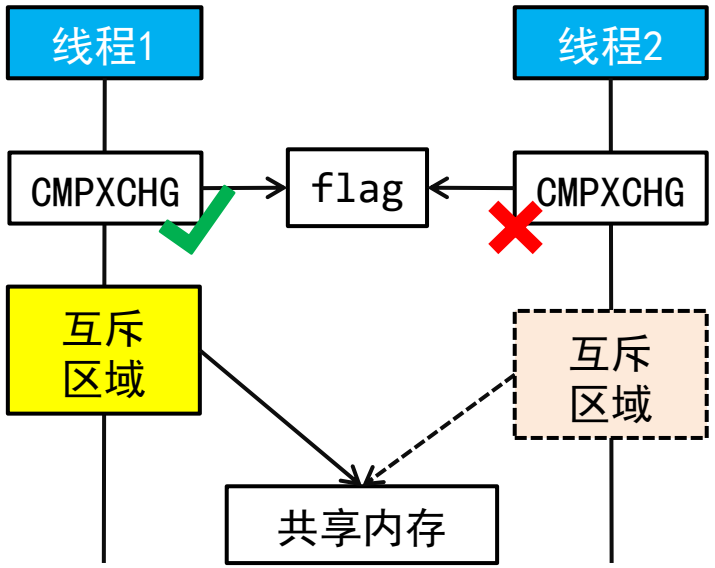
```
gef➤  disass mythread
Dump of assembler code for function mythread:
   0x00401150 <+0>:     push   rbp
   0x00401151 <+1>:     mov    rbp,rsp
   0x00401154 <+4>:     mov    QWORD PTR [rbp-0x10],rdi
   0x00401158 <+8>:     mov    DWORD PTR [rbp-0x14],0x0
   0x0040115f <+15>:    cmp    DWORD PTR [rbp-0x14],0x3e8
   0x00401166 <+22>:    jge    0x401182 <mythread+50>
   0x0040116c <+28>:    lock   add DWORD PTR [rip+0x2ed0],0x1
   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
```

如何实现一段连续指令的原子性

- 基于Compare and Set/Swap机制
  - X86_64：CMPXCHG指令
  - 语言API支持（C）：atomic_compare_exchange_strong

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

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



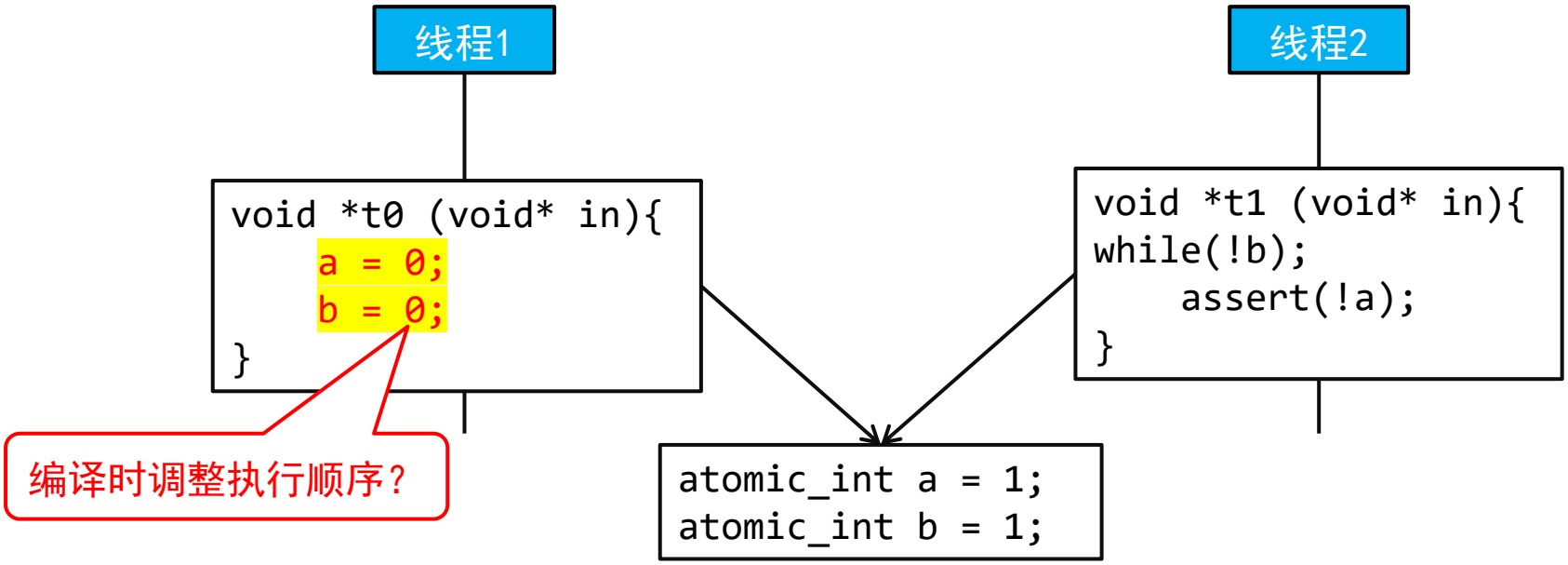
大纲

一、原子访问

二、数据同步

指令重排问题

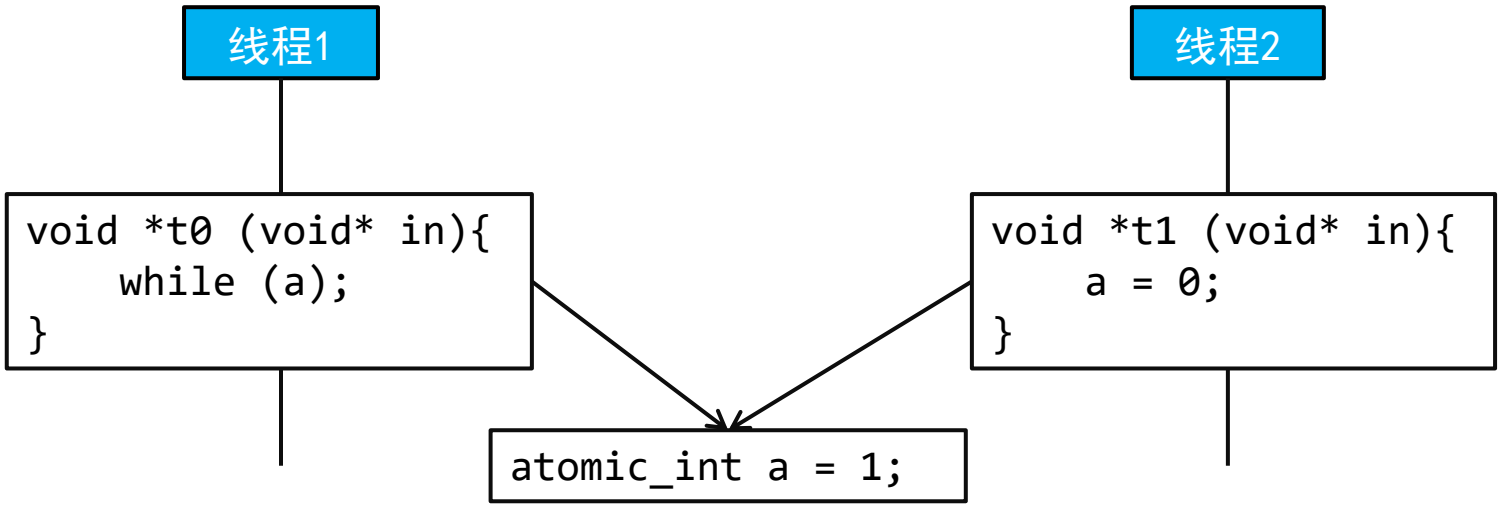
- 指令执行的先后顺序不同会对其它线程产生影响



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

指令重排

- 编译优化可能会误将指令重排

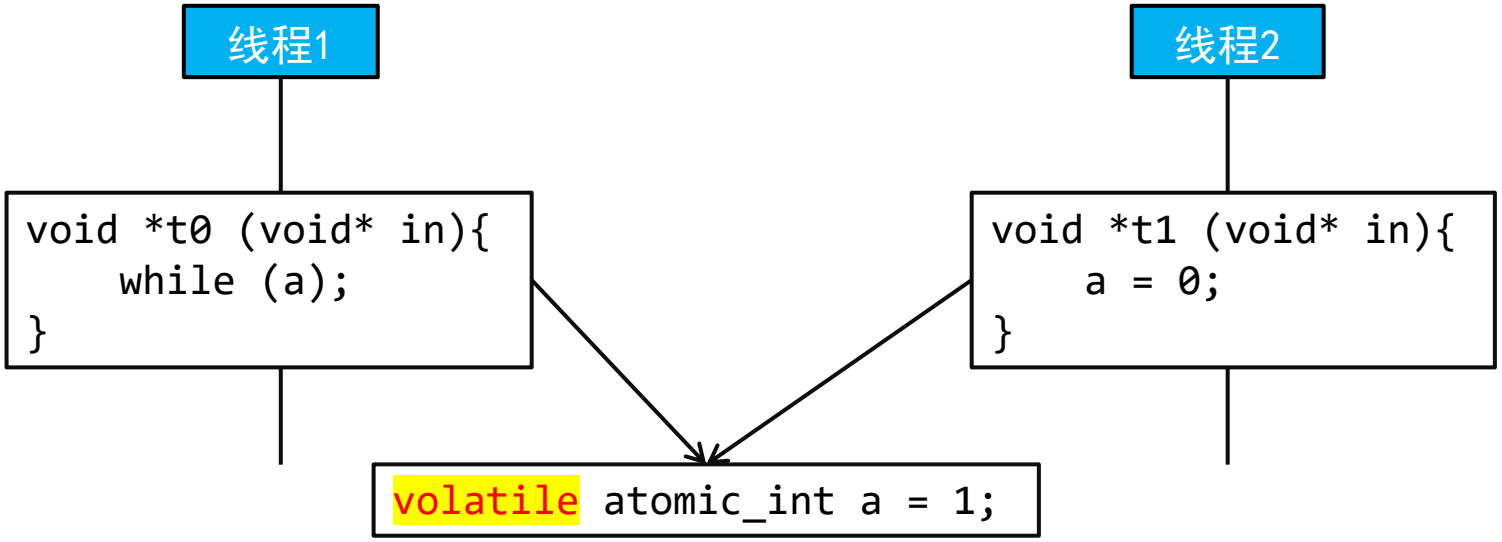



```
0x00401150 <+0>:    cmp     DWORD PTR [rip+0x2ee9],0x0  # <a>
0x00401157 <+7>:    je      0x401162 <t0+18>
0x00401159 <+9>:    nop     DWORD PTR [rax+0x0]
0x00401160 <+16>:   jmp     0x401160 <t0+16>
0x00401162 <+18>:   ret
```

infinite loop

易变内存访问：Volatile

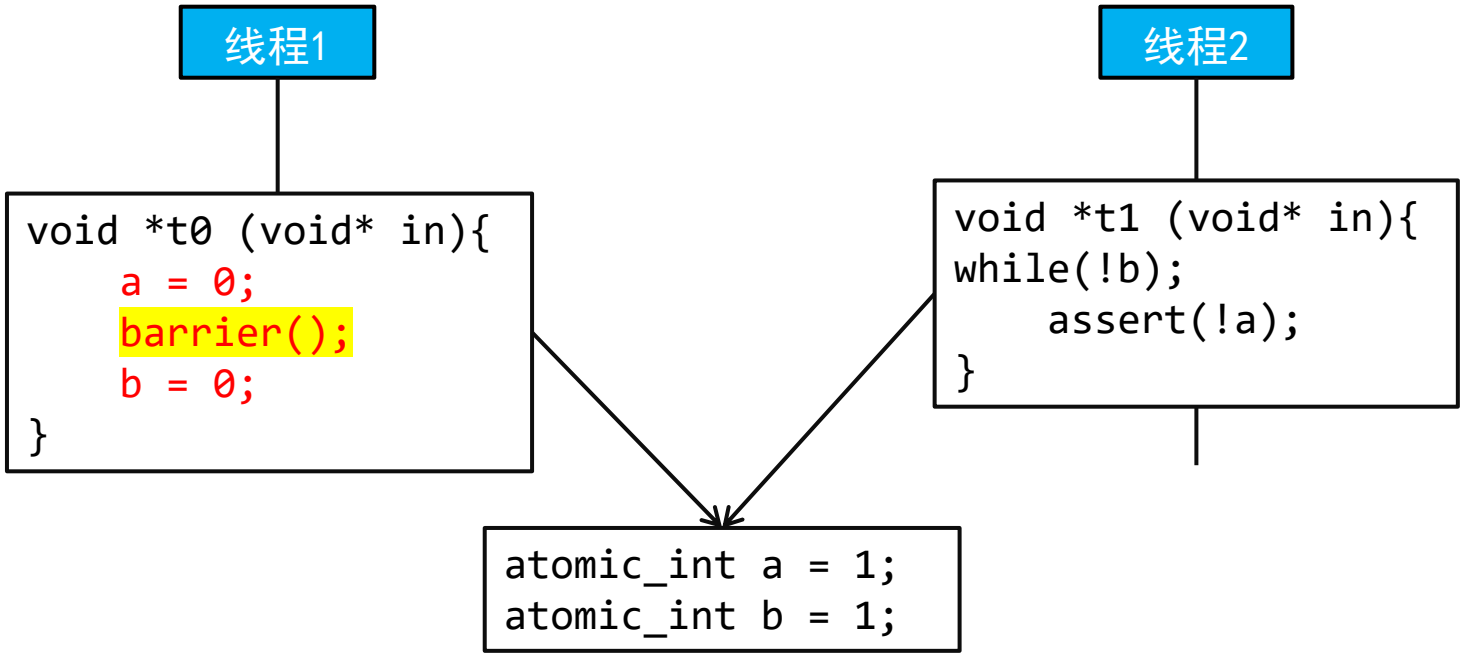
- 丢弃所有寄存器中的值，重新从内存加载



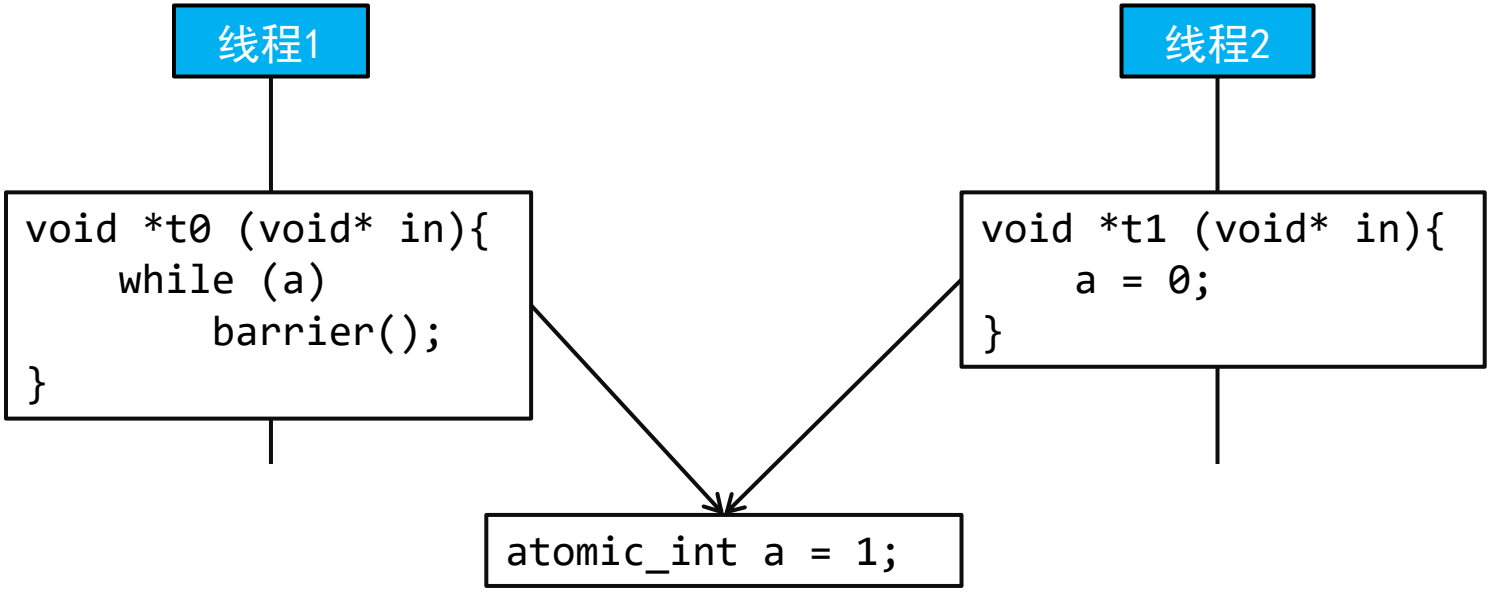
内存屏障

- Memory Barrier (Fence)：
  - 屏障之前的操作总是在其之前完成，确保happens-before顺序（编译器）
  - 丢弃所有寄存器中的值，重新从内存加载（编译器）
  - 并发控制：MFENSE/SFENSE/LFENSE（CPU实现）

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



使用内存屏障


线程1
线程2
void *t0 (void* in){
while (a)
barrier();
}
void *t1 (void* in){
a = 0;
}
atomic_int a = 1;

放松同步要求

- 使用具体的memory ordering
  - Sequential consistency
    - 默认模式，最强同步模式
  - Acquire-release
    - 一般用于锁的实现
    - release：当前线程的读和写操作保证在store fence前完成
    - acquire：当前线程的读和写操作保证在load fence后开始
  - Relaxed
    - 没有同步要求，仅保证原子性

总结