

多处理器编程

目录

- 入门：多线程编程库
- 放弃：原子性、可见性、顺序

多线程编程库

多线程共享内存并发

线程：共享内存的执行流

- 执行流拥有独立的堆栈/寄存器

简化的线程 API (thread.h)

- `spawn(fn)`
 - 创建一个入口函数是 'fn' 的线程，并立即开始执行
 - `void fn(int tid) { ... }`
 - 参数 `tid` 从 1 开始编号
 - 行为: `sys_spawn(fn, tid)`
- `join()`
 - 等待所有运行线程的返回（也可以不调用）
 - 行为: `while (done != T) sys_sched()`

多线程共享内存并发：入门

多处理器编程：一个 API 搞定

```
1 #include "thread.h"
2
3 void Ta() { while (1) { printf("a"); } }
4 void Tb() { while (1) { printf("b"); } }
5
6 int main() {
7     create(Ta);
8     create(Tb);
9 }
```

- 这个程序可以利用系统中的多处理器
 - 操作系统会自动把线程放置在不同的处理器上
 - CPU 使用率超过了 100%

`thread.h` 背后：POSIX Threads

想进一步配置线程？

- 设置更大的线程栈
 - 设置 `detach` 运行（不在进程结束后被杀死，也不能 `join`）
 -
-

POSIX 为我们提供了线程库 (pthreads)

- `man 7 pthreads`
- 练习: 改写 `thread.h`, 使得线程拥有更大的栈
 - 可以用 `stack probe` 的程序验证

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdatomic.h>
5  #include <assert.h>
6  #include <unistd.h>
7  #include <pthread.h>
8  #define NTHREAD 64
9  enum { T_FREE = 0, T_LIVE, T_DEAD, };
10 struct thread {
11     int id, status;
12     pthread_t thread;
13     void (*entry)(int);
14 };
15
16 struct thread tpool[NTHREAD], *tptr = tpool;
17
18 void *wrapper(void *arg) {
19     struct thread *thread = (struct thread *)arg;
20     thread->entry(thread->id);
21     return NULL;
22 }
23
24 void create(void *fn) {
25     assert(tptr - tpool < NTHREAD);
26     *tptr = (struct thread) {
27         .id = tptr - tpool + 1,
28         .status = T_LIVE,
29         .entry = fn,
30     };
31     pthread_create(&(tptr->thread), NULL, wrapper, tptr);
32     ++tptr;
33 }
34
35 void join() {
36     for (int i = 0; i < NTHREAD; i++) {
37         struct thread *t = &tpool[i];
38         if (t->status == T_LIVE) {
39             pthread_join(t->thread, NULL);
40             t->status = T_DEAD;
41         }
42     }
43 }
```

```
1 __attribute__((destructor)) void cleanup() {  
2     join();  
3 }
```

放弃原子性

例子：求和

分两个线程，计算 $1+1+1+\dots+1\dots1+1+1+\dots+1$ （共计 $2n$ 个1）

```
1  #define N 100000000
2  long sum = 0;
3
4  void Tsum() { for (int i = 0; i < N; i++) sum++; }
5
6  int main() {
7      create(Tsum);
8      create(Tsum);
9      join();
10     printf("sum = %ld\n", sum);
11 }
```

可能的结果

- 119790390, 99872322（结果可以比 `N` 还要小），...
- 直接使用汇编指令也不行

放弃：指令/代码执行原子性假设

“处理器一次执行一条指令”的基本假设在今天的计算机系统上不再成立（我们的模型作出了简化的假设）。

单处理器多线程

- 线程在运行时可能被中断，切换到另一个线程执行

多处理器多线程

- 线程根本就是并行执行的

（历史）1960s，大家争先在共享内存上实现原子性（互斥）


- 但几乎所有的实现都是错的，直到 [Dekker's Algorithm](#)，还只能保证两个线程的互斥

放弃原子性假设的后果

`printf` 还能在多线程程序里调用吗？

```
1 void thread1() { while (1) { printf("a"); } }
2 void thread2() { while (1) { printf("b"); } }
```

我们都知道 `printf` 是有缓冲区的（为什么？）

- 如果执行 `buf[pos++] = ch`（`pos` 共享）不就  了吗？

放弃：执行顺序

```
1  #define N 100000000
2  long sum = 0;
3
4  void Tsum() { for (int i = 0; i < N; i++) sum++; }
5
6  int main() {
7      create(Tsum);
8      create(Tsum);
9      join();
10     printf("sum = %ld\n", sum);
11 }
```

如果添加编译优化？

- `-O1`: 100000000 🤖🤖
- `-O2`: 200000000 🤖🤖🤖

编译器对内存访问 “eventually consistent” 的处理导致共享内存作为线程同步工具的失效。

刚才的例子

- `-O1`: `R[ecx] = sum; R[ecx] += N; sum = R[ecx]`
- `-O2`: `sum += N;`
- (你的编译器也许是不同的结果)

另一个例子

```
1  while (!done);
2  // would be optimized to
3  if (!done) while (1);
```

保证执行顺序

回忆 “编译正确性”

- C 状态和汇编状态机的 “可观测行为等价”
- 方法 1: 插入 “不可优化” 代码: “Clobbers memory”

- ```
1 asm volatile ("" ::: "memory")
```

- 方法 2: 标记变量 load/store 为不可优化
  - 使用 `volatile` 变量

```
1 extern int volatile done;
2 while (!done) ;
```

# 放弃：多处理器间内存访问的即时可见性

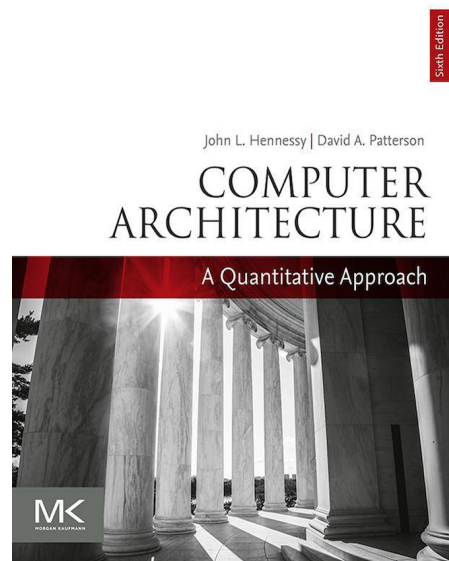
## 例子

```
1 int x = 0, y = 0;
2
3 void T1() {
4 x = 1; // Store(x)
5 __sync_synchronize();
6 printf("%d", y); // Load(y)
7 }
8
9 void T2() {
10 y = 1; // Store(y)
11 __sync_synchronize();
12 printf("%d", x); // Load(x)
13 }
```

遍历模型告诉我们：01, 10, 11

- 机器永远是对的
- Model checker 的结果和实际的结果不同 → 假设错了

 **现代处理器也是（动态）编译器！**



错误（简化）的假设

- 一个 CPU 执行一条指令到达下一状态。

## 实际的实现

- 电路将连续的指令“编译”成更小的  $\mu\text{ops}$

```
1 RF[9] = load(RF[7] + 400)
2 store(RF[12], RF[13])
3 RF[3] = RF[4] + RF[5]
```

在任何时刻，处理器都维护一个  $\mu\text{op}$  的“池子”

- 与编译器一样，做“顺序执行”假设：没有其他处理器“干扰”
- 每一周期执行尽可能多的  $\mu\text{op}$  - 多路发射、乱序执行、按序提交

满足单处理器 eventual memory consistency 的执行，在多处理器系统上可能无法序列化！

当  $x \neq y$  时，对  $x, y$  的内存读写可以交换顺序

- 它们甚至可以在同一个周期里完成（只要 load/store unit 支持）
- 如果写  $x$  发生 cache miss，可以让读  $y$  先执行
  - 满足“尽可能执行  $\mu\text{op}$ ”的原则，最大化处理器性能

```
1 # <-----+
2 movl $1, (x) # |
3 movl (y), %eax # --+
```

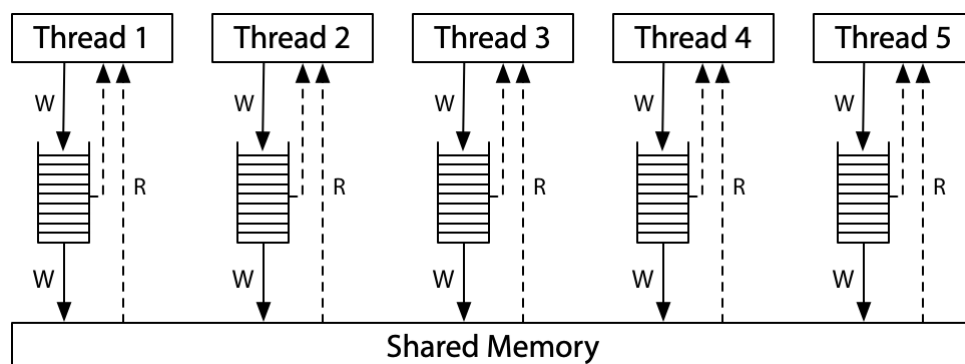
- 在多处理器上的表现
  - 两个处理器分别看到  $y = 0$  和  $x = 0$

## 宽松内存模型（Relaxed/Weak Memory Model）

宽松内存模型的目的是使单处理器的执行更高效。

x86 已经是市面上能买到的“最强”的内存模型了 😊

- 这也是 Intel 自己给自己加的包袱
- 看看 [ARM/RISC-V](#) 吧，根本就是个分布式系统





## Take-away Messages

---

在一个简化的模型中，多线程/多进程程序就是“状态机的集合”，每一步选一个状态机执行一步。然而，真实的系统可能带来一些复杂性：

- 指令/代码执行原子性假设不再成立
- 程序的顺序执行假设不再成立
- 多处理器间内存访问无法即时可见

然而，人类本质上是物理世界（宏观时间）中的“sequential creature”，因此我们在编程时，也“只能”习惯单线程的顺序/选择/循环结构，真实多处理器上的并发编程是非常具有挑战性的“底层技术”，例如 [Ad hoc synchronization](#) 引发了很多系统软件中的 [bugs](#)。因此，我们需要并发控制技术（之后的课程涉及），使得程序能在不共享内存的时候并行执行，并且在需要共享内存时的行为能够“容易理解”。

# 课后习题/编程作业

---

## 1. 阅读材料

教科书 [Operating Systems: Three Easy Pieces](#):

- 第 25 章 - Dialogue on Concurrency
- 第 26 章 - Concurrency and Threads
- 第 27 章 - Thread API

注意：我们的课程和教科书有较大的重叠，但教科书提供了许多授课时间比较难以花时间讲清楚的细节，因此仔细阅读教科书同样重要。

## 2. 编程实践

在你的 Linux 中运行课堂上的代码示例。同学们也可以打开 `thread.h`：它使用起来很简单（直接创建线程即可），但实现也很有趣。例如，`pthread` 线程接受一个 `void *` 类型的参数，且必须返回一个 `void *`。我们用 wrapper function 的方法解决这个问题：所有的线程的实际入口都是名为 `wrapper` 的函数，它会在内部调用线程的 `entry` 函数。

```
1 void *wrapper(void *arg) {
2 struct thread *thread = (struct thread *)arg;
3 thread->entry(thread->id);
4 return NULL;
5 }
```

此外，代码中还有一些可以学习的编程技巧，例如 `ctor` 和 `dtor` 等。