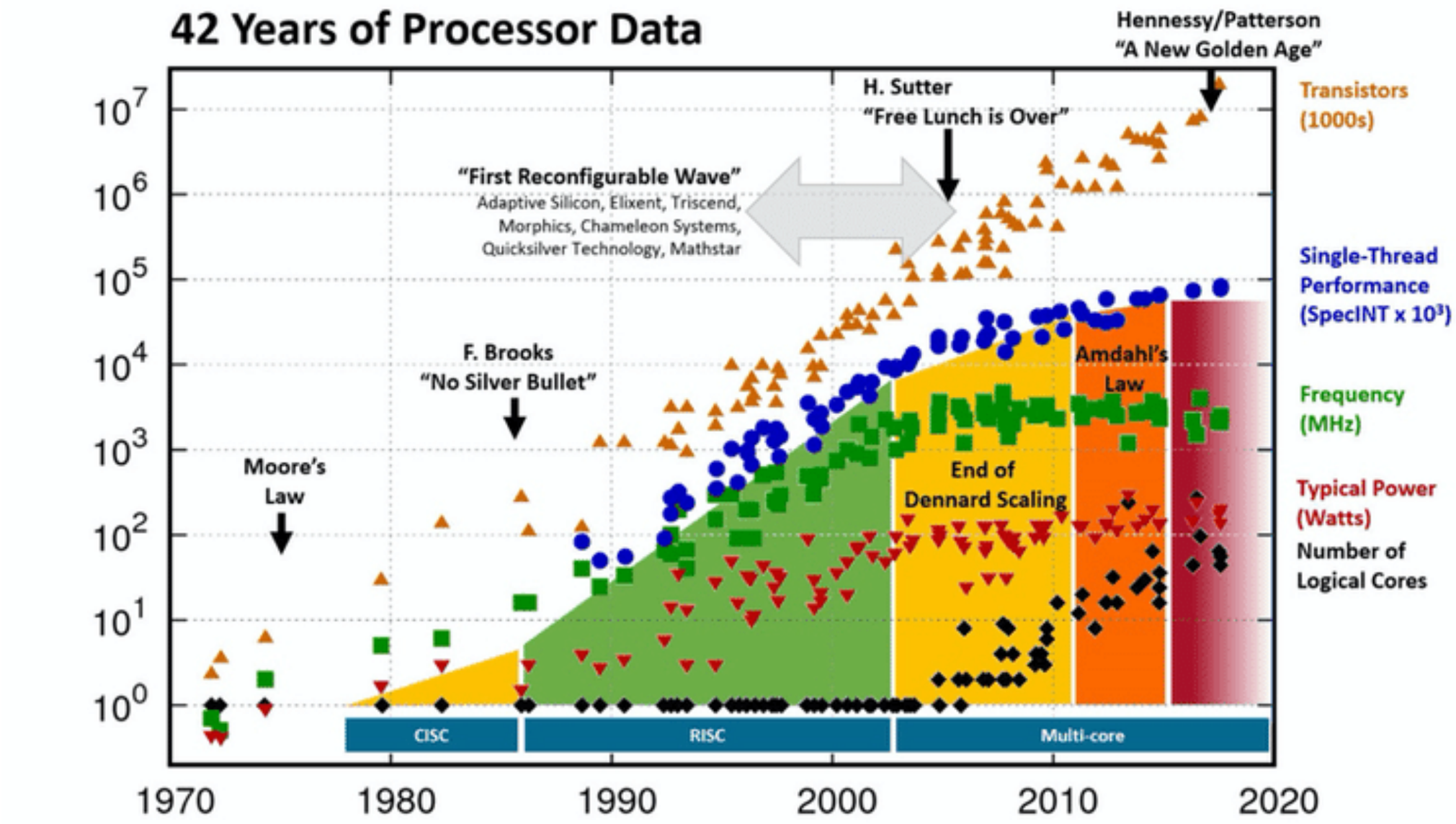
互斥和同步

Section 4: Part 1

2025

多线程编程

"美好的" 单处理器时代已经过去了



Hennessy and Patterson, Turing Lecture 2018, overlaid over "42 Years of Processors Data"
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/;"First Wave" added by Les Wilson, Frank Schirrmeister
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

多线程编程

如果单核性能无法大幅提高，只能依赖多核来并行运算

- 进程 (Process) 和线程 (Thread) 已提供了实现 Multitasking 的能力
- 例如，一个包含两个线程的进程能达到 200% 的 CPU 利用率

```
top - 06:39:23 up 2 min,  3 users,  load average: 1.18, 0.62, 0.25
Tasks: 294 total,   1 running, 293 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.1 us,  0.1 sy,  0.0 ni, 49.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3868.2 total,   2854.2 free,    798.9 used,    449.9 buff/cache
MiB Swap:   1957.0 total,   1957.0 free,      0.0 used.   3069.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1774 huayao    20   0   19072   1408   1408 S 195.7   0.0   0:56.92 a.out
     68 root      20   0       0      0      0 I   0.3   0.0   0:00.19 kworker/3:1-events
   1401 huayao    20   0   36780  16328  11392 S   0.3   0.4   0:00.58 code-ddc367ed5c
      1 root      20   0   22020  12864   9408 S   0.0   0.3   0:01.57 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_workqueue_re+
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_g
```

多线程编程

但人类并不是擅长处理 Multitasking 的生物

Human multitasking

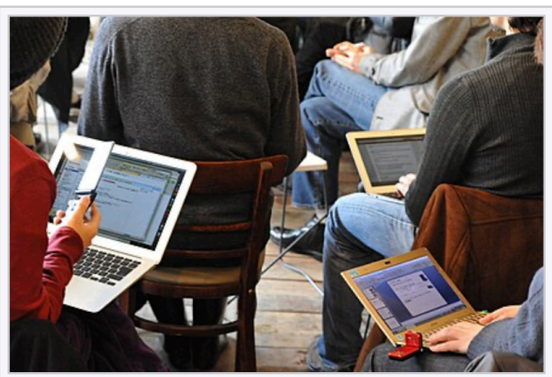
18 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

For other uses, see Multitasking (disambiguation).

Human multitasking is the concept that one can split their attention on more than one task or activity at the same time, such as speaking on the phone while driving a car.



Laptop and mobile phone

Multitasking can result in time wasted due to human context switching (e.g., determining which step is next in the task just switched to) and becoming prone to errors due to insufficient attention. Some people may be proficient at the tasks in question and also be able to rapidly shift attention between the tasks, and therefore perform the tasks well; however, self-perception of being good at multitasking or getting more done while multitasking is frequently inaccurate.[1][2]

Multitasking is mentally and physically stressful for everyone,[3] to the point that multitasking is used in laboratory experiments to study stressful environments.[4] Research suggests that people who are multitasking in a learning environment are worse at learning new information compared to those who do not have their attention divided among different tasks.[5][6][7]

多线程编程

如下程序 (创建了两个线程) 可能的输出结果是什么?

```
void* T1(void *arg) {
  printf("A\n");
}

void* T2(void *arg) {
  printf("B\n");
}

int main(int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, T1, NULL);
  pthread_create(&t2, NULL, T2, NULL);

  pthread_join(t1, NULL)
  pthread_join(t2, NULL)
  return 0;
}
```

多线程编程

我们不能假设多线程程序的执行遵循与单线程程序类似的行为 (which is typically executed in a structured and sequential way)

- 在单核 CPU 上，不同线程的指令会交叠 (interleaved) 在一起执行
  - 任何时刻都有可能产生 Context Switch
  - 调度器的行为是不确定的
    - 即使调度器实现了某种 “确定” 的算法，但影响因素太多 (e.g., unexpected interrupts, different CPU frequencies, cache hit rate, …)
- 对于多核 CPU，多个线程的指令根本就是并行执行的
- 此时，即使我们已经很小心地编程，但还是会出很多问题

多线程编程: 从入门到放弃

No Atomicity

原子性 (Atomicity): 一个操作要么根本就没有执行、要么按照预期执行完毕，不会处于任何中间状态 (all or nothing)

- 当多个线程并发对一个共享数据进行操作时，会存在竞争同一个数据的情况 (Data Race)
- 最终结果取决于操作的具体执行顺序
- 如果 context switch 发生的不巧，则会产生预料之外的结果

多线程编程: 从入门到放弃

No Atomicity

如果多个线程同时执行 `pay(100)`?

```
unsigned long balance = 100;

void pay(int amount) {
  if (balance >= amount) {
    balance -= amount;
  }
}
```

Thread 1 在执行完 `if` 但还没更新 `balance` 时产生 *context switch* 切换到 *Thread 2* 执行 ...

多线程编程: 从入门到放弃

No Atomicity

如果多个线程同时对全局变量 `count` 进行更新?

```
#define NUM 1000000
int count = 0;

void* func(void *arg) {
  for (int i = 0; i < NUM; i++) {
    count++;
  }
  return NULL;
}
```

从高级程序语言看，只有一行代码；

但实际上，是三条指令 (不同线程的指令可能会被交叠在一起执行)

```
mov $count, %eax
add 1, %eax
mov %eax, $count
```

多线程编程: 从入门到放弃

No Atomicity

如果多个线程同时对全局变量 `count` 进行更新?

Thread 1

```
mov $count, %eax
add 1, %eax
mov %eax, $count
```

Thread 2

```
mov $count, %eax
add 1, %eax
mov %eax, $count
```

`Thread 1` 和 `Thread 2` 都分别做了一次 `count++`

此时 `count += 2`

Thread 1

```
mov $count, %eax
add 1, %eax


mov %eax, $count
```

Thread 2

```
mov $count, %eax
add 1, %eax
mov %eax, $count
```

此时 `count += 1`

多线程编程: 从入门到放弃

No Atomicity

如果多个线程同时对全局变量 `count` 进行更新?

```
#define NUM 1000000
int count = 0;

void* func(void *arg) {
  for (int i = 0; i < NUM; i++) {
    asm volatile("incq %0" : "+m" (count));
  }
  return NULL;
}
```

把 `count++` 变成一条指令就正确了吗?

多线程编程: 从入门到放弃

No Sequential Order

顺序性 (In-order): 程序按照代码语句编写的顺序执行

- 只要不影响语义，指令是否按照顺序执行并不重要
  - 编译器会通过 reorder instructions 来提高程序执行速度
  - 这些优化在单线程下往往没有问题 (safe)，但在多线程下就未必

多线程编程: 从入门到放弃

No Sequential Order

如果使用 -O1 和 -O2 选项来编译执行下列代码?

```
#define NUM 1000000
int count = 0;

void* func(void *arg) {
  for (int i = 0; i < NUM; i++) {
    count++;
   }
  return NULL;
}
```

-O1 的执行结果 → 1000000 ？为什么?

count 被移出循环，最后又被加回来

-O2 的执行结果 → 2000000 ？看起来结果对了，但是真的吗?

直接优化为 count += 1000000

多线程编程: 从入门到放弃

No Sequential Order

如果我们想让线程 `T1` 等待线程 `T2` (实现一种互相等待)?

```
int flag = 0;
int x = 0;

void *T1(void* arg) {
  while (flag == 0)
    ;
  printf("%d\n", x);  // use x
  return NULL;
}

void *T2(void* arg) {
  x = 10;  // prepare x
  flag = 1;
  return NULL;
}
```

在 `-O2` 下会被优化为

```
if (flag == 0)
  while(1) ;
```

多线程编程: 从入门到放弃

No Consistency *

内存一致性模型 (Memory Consistency Model) 定义了不同处理器对于共享内存操作需要遵循的顺序 (how multiple threads see the world)

- 现代处理器往往允许指令乱序执行 (同样是为了优化性能)
  - 例如，对于高时延的访存指令 (e.g., cache miss)，处理器可以选择调度后续其他指令执行，从而缓解访存操作的时延
  - 这种乱序会导致多个处理器看到不一致的访存顺序
- 内存一致性模型就是一种硬件和软件之间的约定

多线程编程: 从入门到放弃

No Consistency *

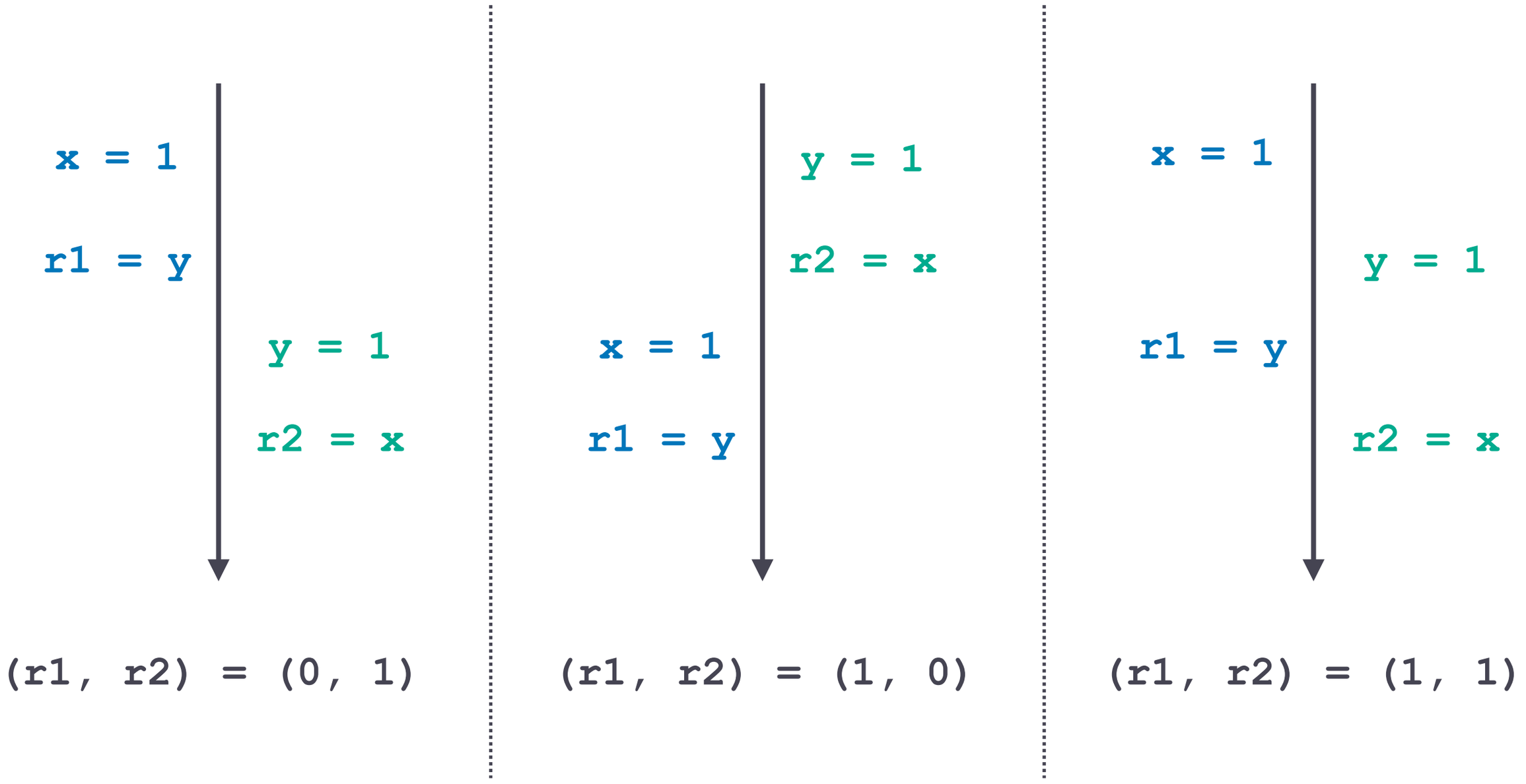
线程 `T1` 和 `T2` 运行结束后 `r1` 和 `r2` 的值分别可能是多少?

```
volatile int x = 0, y = 0;
volatile int r1, r2;

void* T1(void *arg) {
  x = 1;   // store(x)
  r1 = y;  // load(y)
}

void* T2(void *arg) {
  y = 1;   // store(y)
  r2 = x;  // load(x)
}
```

多线程编程: 从入门到放弃

No Consistency *

线程 `T1` 和 `T2` 运行结束后 `r1` 和 `r2` 的值分别可能是多少?

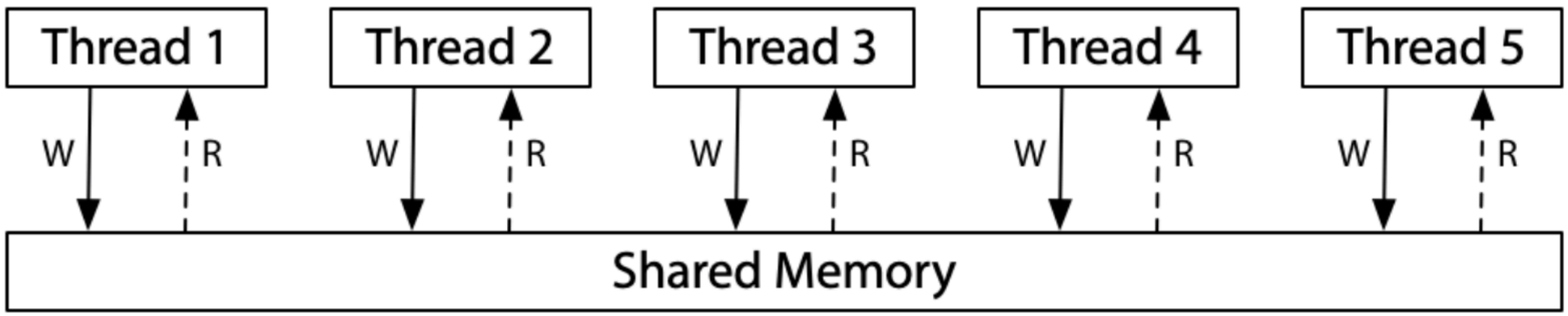


多线程编程: 从入门到放弃

No Consistency *

Sequential Consistency (an intuitive model of parallelism)

- The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program (everything must happen in order)



Every time a processor needs to read from or write to memory, that request goes to the shared memory

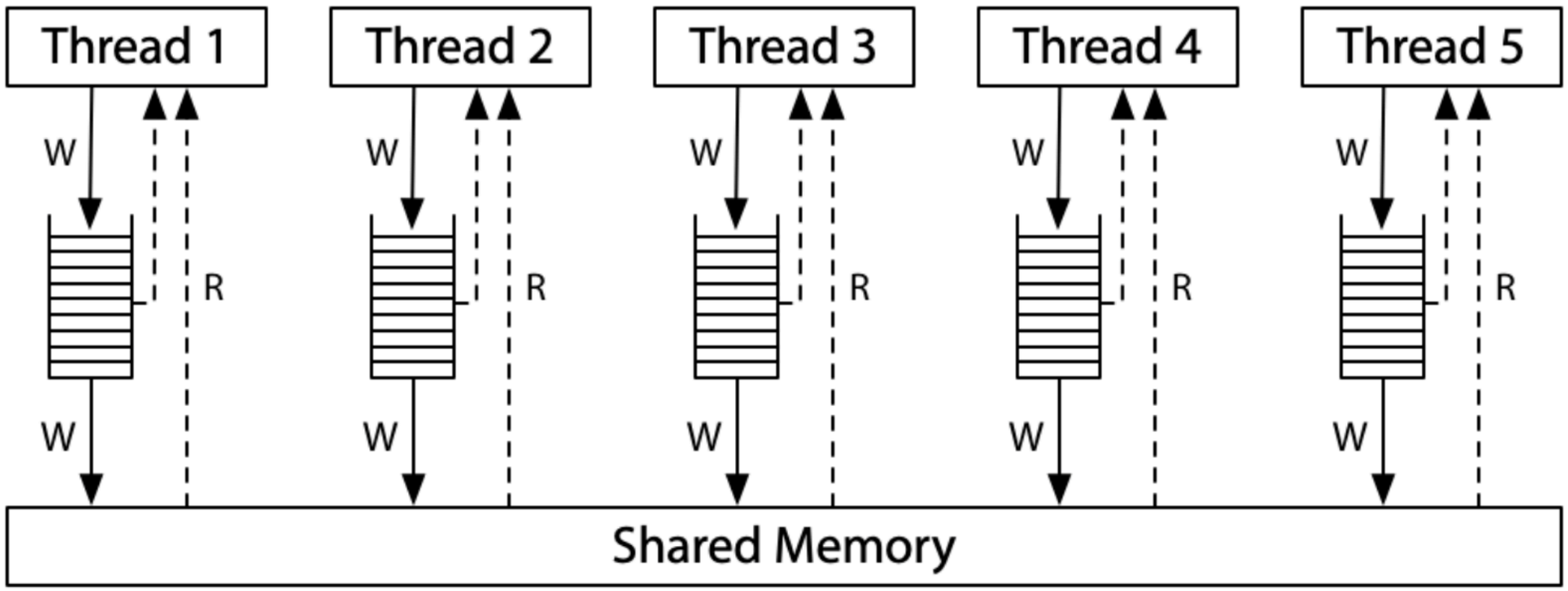
https://research.swtch.com/hwmm

多线程编程: 从入门到放弃

No Consistency *

Total Store Order (x86)

- A popular memory model that allows store buffering



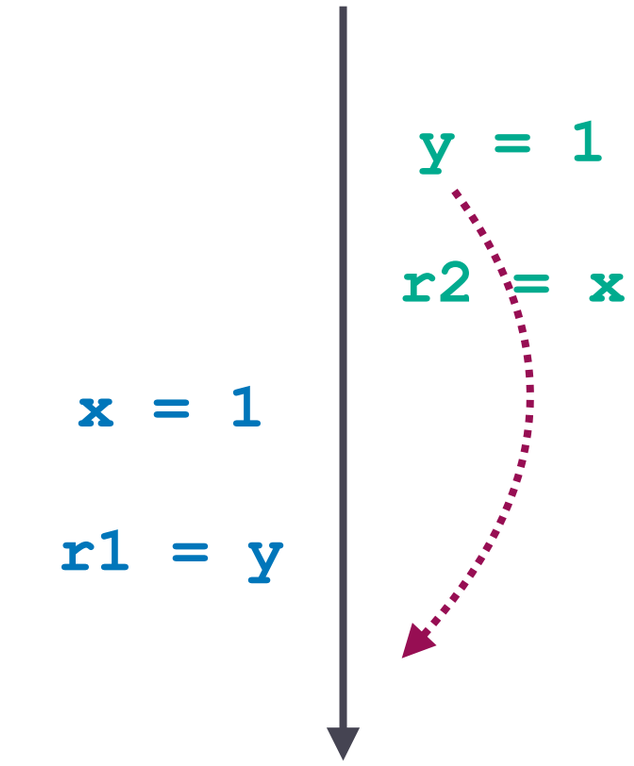
Each processor queues writes to that memory in a local write queue

https://research.swtch.com/hwmm

多线程编程: 从入门到放弃

No Consistency *

线程 `T1` 和 `T2` 运行结束后 `r1` 和 `r2` 的值分别可能是多少?



Both threads could queue their writes and then read from memory before either write makes it to memory.

`(r1, r2) = (0, 0)`

多线程编程: 从入门到放弃

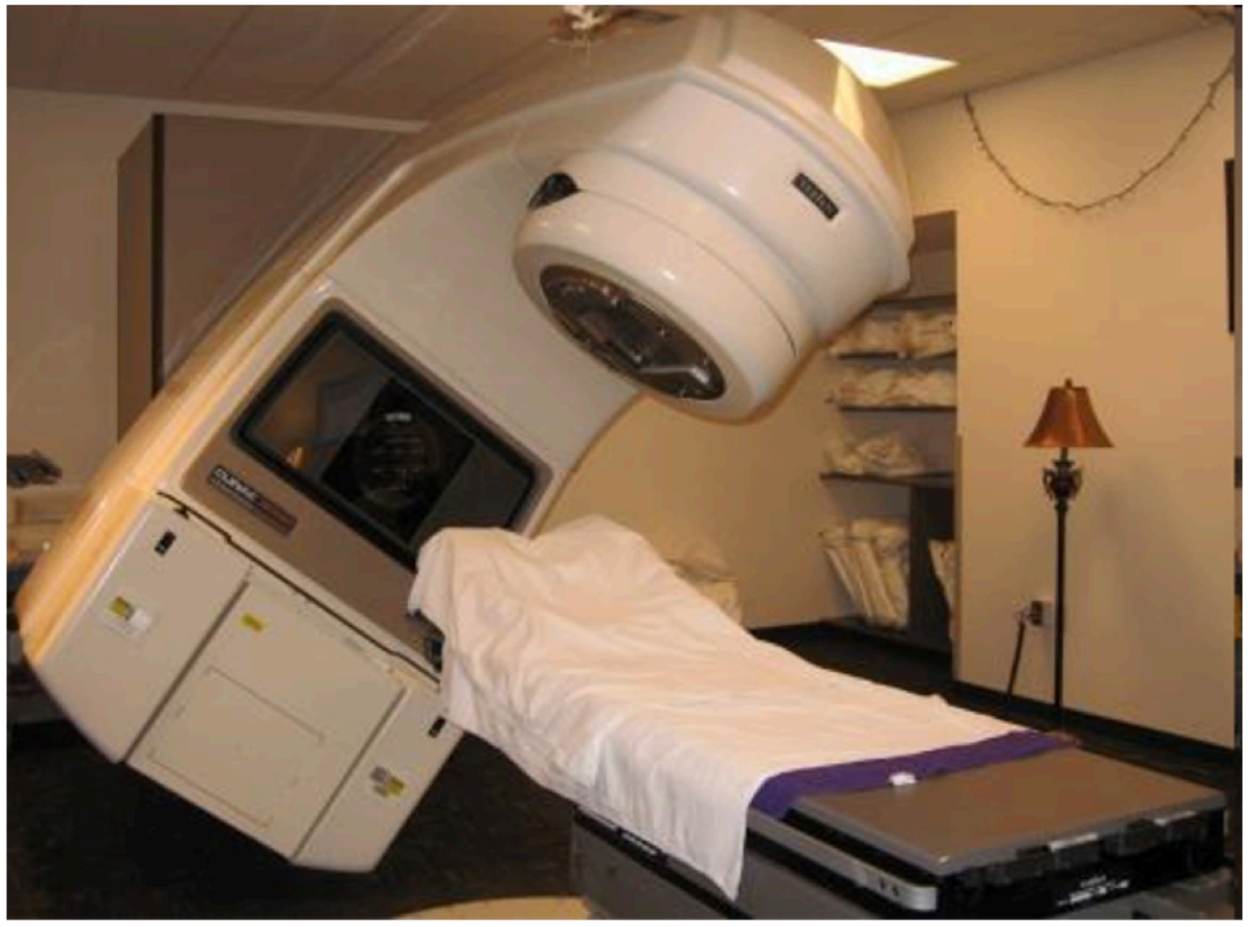
多线程并发程序很难写对，而且也很难检测其中的并发 Bugs
(the failure might be improbable to trigger, but possible)

- 多线程指令能以很多不同的方式交叠执行
- 其中只有部分交叠执行顺序会触发 Bugs (程序的非确定性行为)
  - 同样的输入在不同次执行下会产生不同的结果
  - 即使运行很久没有出问题也不代表并发程序就一定是对的
- 还需要了解编译器、以及底层硬件的很多细节，才能尝试去理解并发程序的真实表现行为

经典的并发 Bugs

Therac-25 放射线治疗仪 (1980s)

- 由于系统运行过程中的一个 race condition，导致至少六名患者因收到过量辐射而死亡或严重受伤
- Therac-25 之前的机型通过一个硬件锁来避免此情况，但在 Therac-25 中改由软件来进行检查和处理



https://en.wikipedia.org/wiki/Therac-25

经典的并发 Bugs

北美大停电 (2023)

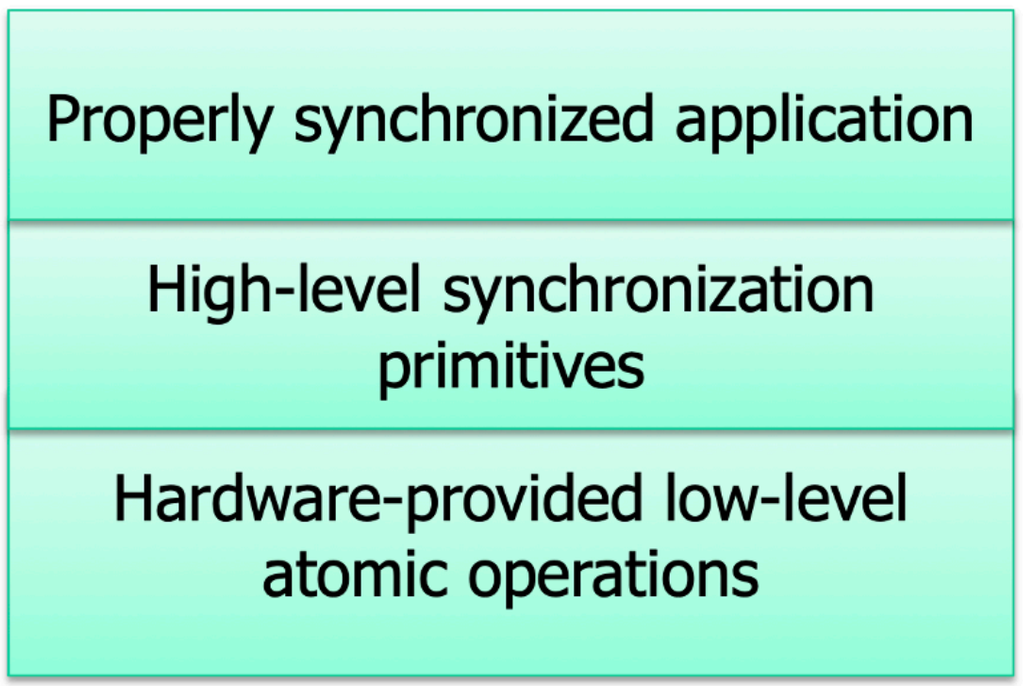
- 约有 5500 万人受到影响，经济损失估计 250～300 亿美元
- 监控系统中的一个 race conditon 部分导致警报系统失效
- 系统代码自 1990 年开始运行，在持续运行超过 300 万小时中从未出现任何 bug

https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

互斥和同步

我们需要特定的机制来重新获得多线程执行的某种确定性 (determinism)

- 协调多个线程以达成某种一致 (某种特定的指令执行序列)
- 一种分层的实现方式
  - 由硬件提供一些必要的原子性指令
  - 在此基础上，构建一些 synchronization primitives



互斥和同步

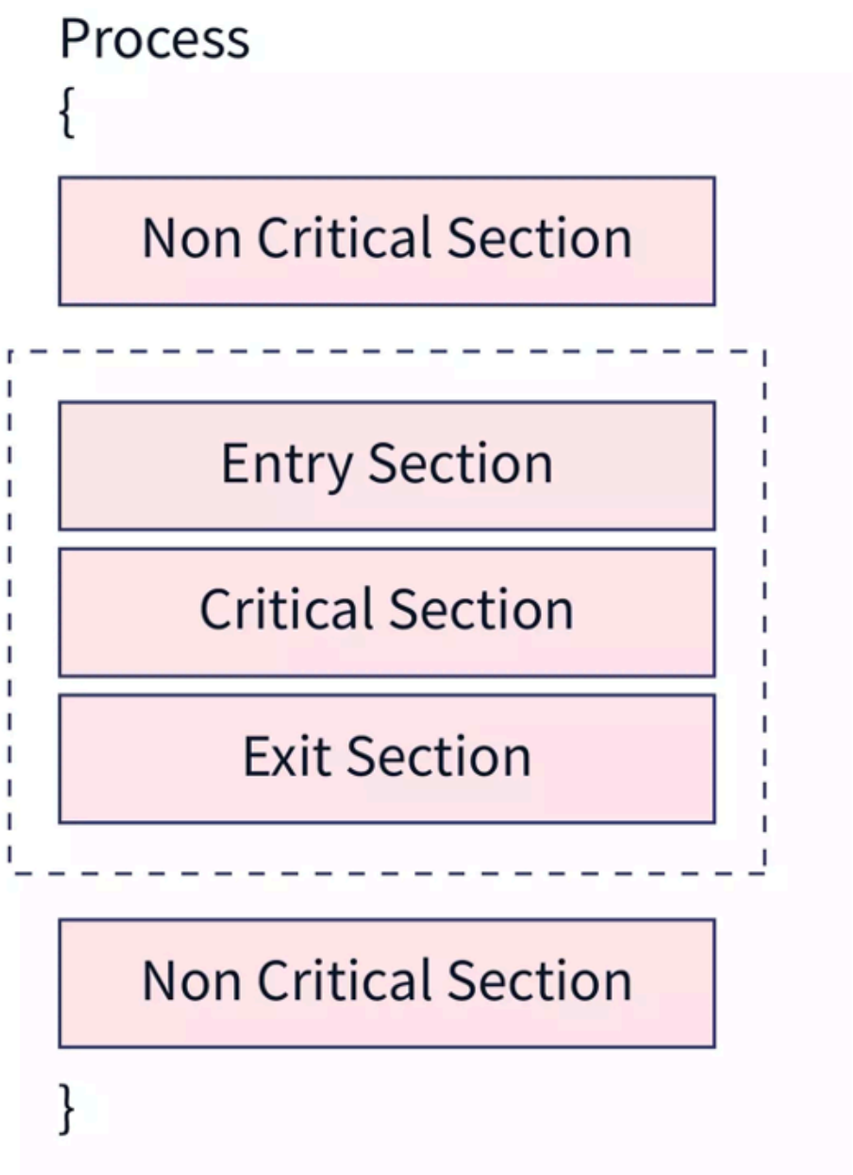
为什么我们要在操作系统课程里学习这个主题？

- 因为所有操纵系统教材里都有这一章 😵‍💫
- 操作系统就是一个重要的并发程序
  - 内核的很多内部数据结构 (例如 PID、进程列表、页表、文件系统结构等) 都存在数据竞争
- 解决并发的很多技术都源自于操作系统的设计需求和其相应的解决方案

Locks

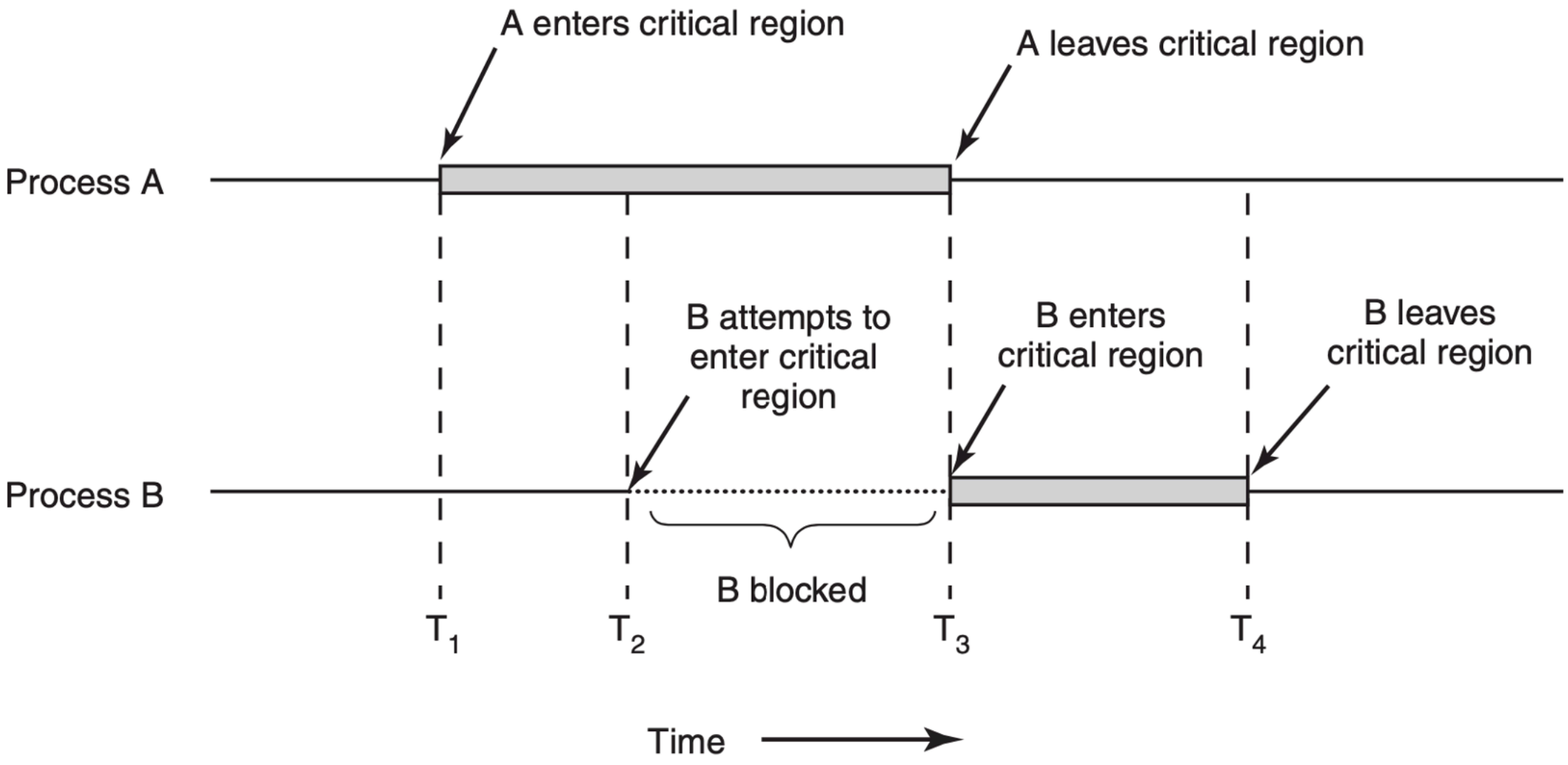
Critical Section

使用临界区 (Critical Section) 来刻画访问共享资源的一段代码



Critical Section

互斥 (Mutual Exclusion): 如果一个线程在临界区内执行，则其他线程将被阻止进入临界区 (确保临界区中的代码以某种方式顺序执行)



Critical Section

- 实现临界区需要满足的条件
  - 临界区内最多只能有一个线程执行 (Mutual Exclusion)
  - 如果一个线程在临界区外，则该线程不能阻止其它线程进入临界区 (Progress / Liveness)
  - 如果一个线程正在等待进入临界区，则该线程最终一定有机会进入 (Bounded Waiting / Fairness)
- 此外，与临界区内的计算开销相比，进入临界区和退出临界区这两个操的开销应该尽可能小 (Performance)

Locks

一个锁 (lock) 就是一个变量 (variable)，其保存了锁在某一时刻的状态 (either available/free or acquired/hold)

🔒 **`lock()`**

- 如果没有其它线程持有这个锁 (lock is available)，则执行该操作的线程获得锁，并进入临界区
- 如果其它线程持有这个锁 (lock is acquired)，则调用不会返回

🔓 **`unlock()`**

- 当持有锁的线程执行该操作时，锁变为 available 状态
- 如果此时有正在等待的线程，则其中一个会感知到此时锁状态的变化，从而获得锁并进入临界区

Locks

有了锁之后，程序员就能实现互斥了 (获得了对并发程序最基本的控制)

- 通过给某代码片段加锁，确保该段代码执行的原子性
- 但互斥实现的正确性仍取决于 "有没有正确使用锁"
  - 在哪里加锁？ 使用几个锁？ 可不可以不使用锁？

```
#define NUM 1000000
int count = 0;

void* func(void *arg) {
  for (int i = 0; i < NUM; i++) {
    count++;
   }
  return NULL;
}
```

How to build a lock

Disabling Interrupts

在进入临界区的时候关中断

- 临界区内线程的执行不会被打断 (相应地，共享资源就不会被其它线程所访问)

```
void lock() {
  DisableInterrupts();
}

void unlock() {
  EnableInterrupts();
}
```

How to build a lock

Disabling Interrupts

在进入临界区的时候关中断

- 如果允许用户程序执行关中断指令？
  - 关中断是一个特权指令 (privileged instruction)
  - 在操作系统内核中，使用关中断是一个常见的操作
- 如果临界区代码死循环 (操作系统无法重新获得控制权)？
- 中断关闭时间过长会导致外部的重要事件丢失？
- 如果本身就是多处理器系统？
  - 中断是每个处理器内部状态 (每个处理器有独立的寄存器组)

How to build a lock

Just Use Loads & Stores

使用一个 `flag` 变量来记录当前锁的状态 (assume load and store operations are atomic)

```
// 0 -> lock is available, 1 -> held
flag = 0

void lock() {
  while(flag == 1) // TEST the flag
    ;
  flag = 1;        // now SET it
}

void unlock() {
  flag = 0;
}
```

What could possibly go wrong?
(pretend you are a malicious scheduler)

How to build a lock

Just Use Loads & Stores

使用一个 `flag` 变量来记录当前锁的状态 (assume load and store operations are atomic)

```
// initially, flag = 0

Thread 1                        Thread 2

// lock()                       // lock()
while(flag == 1);
                                while(flag == 1);
                                flag = 1;

flag = 1;
// critical section             // critical section
```

How to build a lock

Just Use Loads & Stores

使用一个 `flag` 变量来记录当前锁的状态 (assume load and store operations are atomic)

- 不满足互斥需求：存在两个线程同时将 `flag` 置为 `1` 的情况 (即同时进入临界区的情况)
- 关键问题在于对 `flag` 的 *TEST & SET* 不是原子操作
  - *TEST:* 看一下当前是什么状态，但不知道后面会变成什么样
  - *SET:* 对状态做更新，但不知道后面会不会又被修改

How to build a lock

Just Use Loads & Stores

使用一个 `flag` 变量来标记当前轮到谁进入临界区

```
// assume two threads: 0 and 1
flag = 0;

void lock() {
  // wait for my turn
  while(flag == 1 - self)
    ;
}

void unlock() {
  // I am done, your turn
  flag = 1 - self;
}
```

How to build a lock

Just Use Loads & Stores

使用一个 `flag` 变量来标记当前轮到谁进入临界区

- 一种通过严格轮转 (strict alternation) 来实现互斥的方法
- 但是，一个线程能不能进入临界区取决于另一个线程是否进入过临界区 (不满足 liveness 性质)

- `T0` enters its critical section and exits; sets `turn = 1`; then executes a long time non-critical procedure
- `T1` enters its critical section and exits; sets `turn = 0`; then tries to enter its critical section again
- Now, `T0` is in its non-critical section, and `T1` is waiting for `turn` to become `0`

Peterson's Algorithm

在 1960s 人们尝试了很多纯软件方法来实现互斥，但都是错的

- 直到 Dijkstra 的一个数学家朋友 Dekker 给出了第一个正确的算法
- Peterson 在 1989 年对该算法进行了改进和简化 (结合 flag variable 和 strict alternation 两种思想)

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions. The solution presented here has an extremely simple structure and, as shown later, is easy to prove correct.

Gary L. Peterson. Myths about the mutual exclusion problem. Information Processing Letters, 1981

Peterson's Algorithm

```
// indicate the intend to hold the lock
bool flag[2] = {false, false};
// whose turn is it (thread 0 or 1)
int turn;

void lock() {
  flag[self] = true;  // I would like to enter
  turn = 1 - self;    // but make it other's turn
  while ((flag[1-self] == true) && (turn == 1 - self))
    ;
}

void unlock() {
  // undo the intent
  flag[self] = false;
}
```

Peterson's Algorithm

Thread 0

```
while(1) {
  flag[0] = true;
  turn = 1;
  while(flag[1] && turn == 1)
    ;
  // critical section
  flag[0] = false;
  // reminder section
}
```

Thread 1

```
while(1) {
  flag[1] = true;
  turn = 0;
  while(flag[0] && turn == 0)
    ;
  // critical section
  flag[1] = false;
  // reminder section
}
```

如果两个线程都想进入临界区，则 `turn` 的最终值决定了进入的顺序
(谁先置 `turn` 的值谁先进入临界区)

Peterson's Algorithm

- **满足 Mutual Exclusion**

T1 此时 while 条件是 True

```
T0:while()
T1:while()
turn=0
flag[0]=true
flag[1]=true
```

```
T0:turn = 1
T1:critical
turn=_
flag[0]=true
flag[1]=true
```

T0 会将 turn 的值设置为 1

T0 一定是从该状态进入临界区 (只有其 while 条件能被不满足)

```
T0:while()
T1:critical
turn=0
flag[0]=true
flag[1]=true
```

不失一般性，假设这个状态发生了 (T0 和 T1 都进入了临界区)

```
T0:critical
T1:critical
turn=0
flag[0]=true
flag[1]=true
```

所以不会执行到这个状态

Peterson's Algorithm

- **满足 Mutual Exclusion**
- **满足 Progress**
  - 如果 `T0` 不在临界区，则一定有 `flag[0] = false`，此时 `T1` 一定能进入临界区
- **满足 Bounded Waiting**
  - 如果 `T0` 正在等待进入临界区，则一定有 `flag[1] = true` 并且 `turn = 1`
  - 此时如果 `T1` 想要再次进入临界区，则其一定会设置 `turn = 0`，此时 `T0` 将进入临界区

Peterson's Algorithm

- 如果对 Peterson 算法稍作修改，其性质是否还能满足：
  - 交换 `flag` 和 `turn` 的赋值顺序？
  - 交换 `while` 语句两个条件的顺序？
  - …
- 通过手工证明一个程序是否满足特定性质是很繁琐且易错的事情
  - 可以借助模型检验 (Model Checking) 来实现自动化证明
    (a method for formally verifying finite-state systems)
  - 但需要有效地解决状态空间爆炸的问题

Peterson's Algorithm

- 然而，上述 Peterson 算法在目前硬件架构上并不能保证实现互斥
  - Load 和 Store 操作不一定是原子的 (e.g., write a 64-bit integer on 32-bit CPU requires two store instructions)
  - 为了提高性能，处理器会 reorder instructions (relaxed memory consistency model)
    - 可以使用 hardware memory barriers (fences) 来防止处理器 reorder instructions
- 此外，Peterson 算法的原始版本只支持两个线程
  - 可以扩展到 *N* 个线程，但需要提前知道 *N* 的值

Build Working Locks

不忘初心: 只要让 *TEST & SET* 是一个原子操作就可以了

```
// 0 -> lock is available, 1 -> held
flag = 0

void lock() {
  while(flag == 1) // TEST the flag
    ;
  flag = 1;          // now SET it
}

void unlock() {
  flag = 0;
}
```

Build Working Locks

Atomic Exchange (`xchg`)

返回 `ptr` 指向的值 (test the old value)，同时原子性地将该值更新为 `new` 对应的值 (set the memory location to a new value)

```
int xchg(int *ptr, int new) {
   int old = *ptr;  // fetch old value at ptr
   *ptr = new;      // store new into ptr
   return old;
}
```

Build Working Locks

Atomic Exchange (`xchg`)

```
typedef struct __lock_t {
   int flag;
} lock_t;

void init(lock_t *lock) {
   // 0 -> available, 1 -> held
   lock->flag = 0;
}

void lock(lock_t *lock) {
   while (xchg(&lock->flag, 1) == 1)
      ;
}

void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

Build Working Locks

Compare-and-Swap (`cmpxchg`)

测试 `ptr` 指向的值是否等于 `expected`

- 如果相等，使用 `new` 的值进行更新
- 如果不相等，不做任何操作

```
int cmpxchg(int *ptr, int expected, int new) {
   int old = *ptr;
   if (old == expected)
      *ptr = new;
   return old;
}
```

Build Working Locks

Compare-and-Swap (`cmpxchg`)

```
typedef struct __lock_t {
   int flag;
} lock_t;

void init(lock_t *lock) {
   // 0 -> available, 1 -> held
   lock->flag = 0;
}

void lock(lock_t *lock) {
   while (cmpxchg(&lock->flag, 0, 1) == 1)
      ;
}

void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

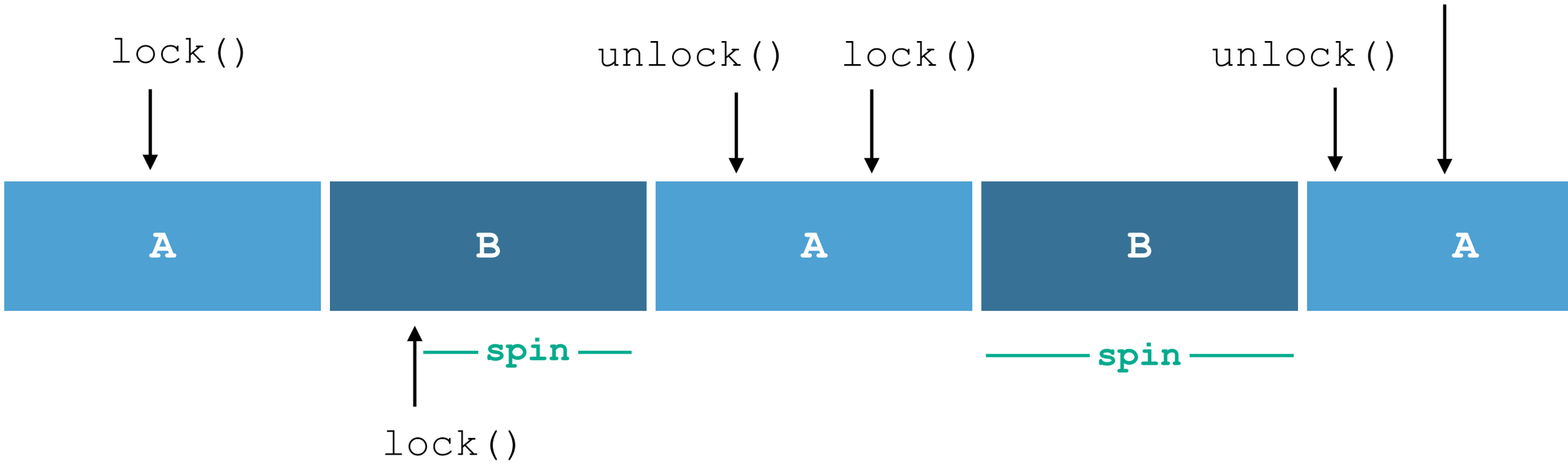
Spin Lock

上述基于 Test-And-Set 原子指令实现的锁也被称为自旋锁 (Spin Lock)

- 线程获取不到锁时不断在一个 `while` 循环中空转 (Busy Waiting)，直到锁的状态再次变为 available 为止
- 自旋锁的公平性 (Fairness) 如何？
  - 一个想进入临界区的线程一定能进入临界区吗？
  - 持有锁的线程释放锁时，哪个线程会获得锁？

Spin Lock

一个在自旋等待锁的线程可能会永远自旋下去 (starvation)



Ticket Lock

Fetch-and-Add (`xadd`)

原子性地增加 `ptr` 指向的值 (同时返回该地址的旧值)

```
int xadd(int *ptr) {
   int old = *ptr;
   *ptr = old + 1;
   return old;
}
```

Ticket Lock

Fetch-and-Add (`xadd`)

```
typedef struct __lock_t {
   int ticket;
   int turn;
} lock_t;

void init(lock_t *lock) {
   lock->ticket = 0;
   lock->turn = 0;
}

void lock(lock_t *lock) {
   int my_turn = xadd(&lock->ticket);
   while (lock->turn != my_turn)
      ;
}

void unlock(lock_t *lock) {
   lock->turn = lock->turn + 1;
}
```

线程每次想要进入临界区就拿一个“号” (ticket)，并等待“叫号”

Ticket Lock

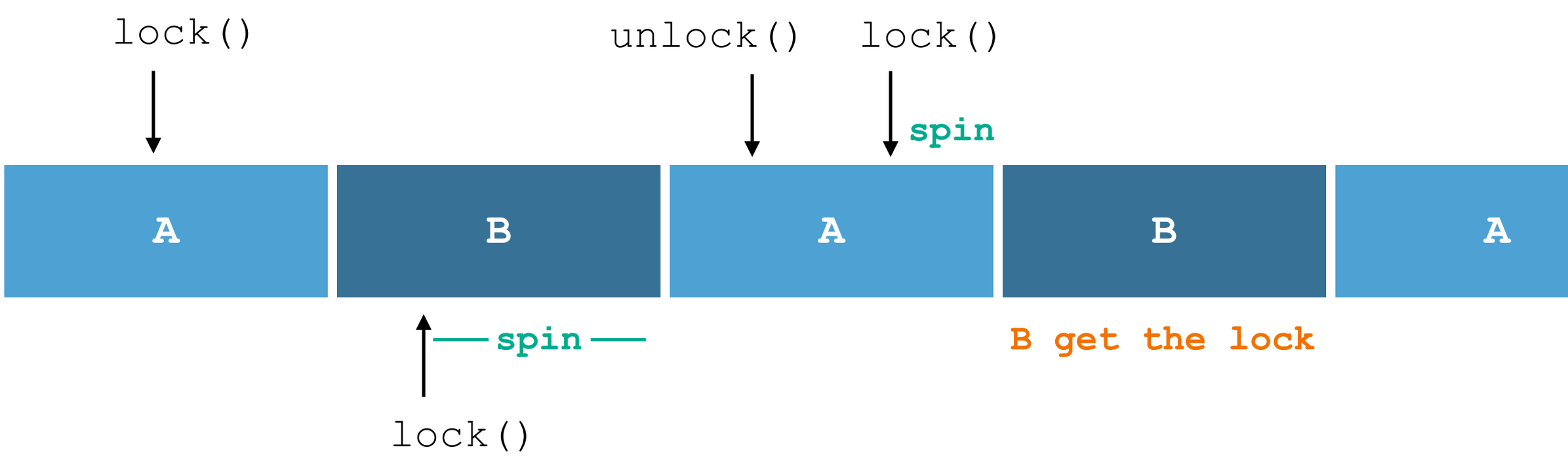
Fetch-and-Add (`xadd`)

```
init ticket = 0, turn = 0
```

```
get ticket 0,
spin until turn == 0
```

```
turn = 1
```

```
get ticket 2,
spin until turn == 2
```




```
get ticket 1,
spin until turn == 1
```

Spin Lock

- 自旋锁的性能 (Performance) 如何？
  - 除了当前持有锁的线程外，其它线程都在空转
    - 对于比较小的临界区，自旋一会儿是一种合理的选择 (但在单核下仍旧会导致性能下降)
    - 对于比较大的临界区 (锁被长时间持有)，较大的计算资源浪费
  - 更糟糕的是，如果当前持有锁的线程被切换出去了 (例如并发执行的线程个数超过处理器核个数)
    - 其它所有线程的时间片都用来自旋，不断检查一个注定不会发生改变的状态 (100% 的资源浪费)

Locks for User Apps

Just Yield

当线程无法获得锁时，利用 `yield` 系统调用主动让出 CPU (线程状态由 Running 变为 Ready)

```
void lock() {
  while (xchg(&flag, 1) == 1)
    yield();
}

void unlock() {
  flag = 0;
}
```

Locks for User Apps

Just Yield

当线程无法获得锁时，利用 `yield` 系统调用主动让出 CPU (线程状态由 Running 变为 Ready)

- 只是暂时让出 CPU，线程处于 Ready 状态可随时被再次调度
  - 在获得锁之前，反复的 “scheduled → yield” 会带来大量的 context switch 开销 (设想 100 个想获取锁的线程通过 RR 方式调度)
- 同时也存在公平性 (Fairness) 的问题

Locks for User Apps

Sleep and Wakeup

当线程无法获得锁时，通过 `sleep` 将其置于 Blocked 状态，并在锁释放时通过 `wakeup` 唤醒正在等待的一个线程 (block when waiting)

```
void lock(lock_t *lock) {
  // if cannot acquire the lock -> blocked
  while(xchg(&lock->flag, 1) == 1) {
    sleep(lock->queue);
  }
}

void unlock(lock_t *lock) {
  // if there are threads waiting -> ready
  lock->flag = 0;
  if(!is_empty(lock->queue))
    wakeup(lock->queue);
}
```

What could possibly go wrong?

Locks for User Apps

Sleep and Wakeup

Thread 0

```
// assume flag = 1
// lock()
while(xchg(&lock->flag, 1) == 1)


  sleep(lock->queue);
```

Thread 1

```


// unlock()
flag = 0;
if(!is_empty(lock->queue))
  wakeup(lock->queue);
```

Lost wakeup

Locks for User Apps

Sleep and Wakeup

Thread 0

```
// assume flag = 1
// lock()
while()
  sleep();
```

Thread 1

```
// unlock()
flag = 0;

if(!is_empty())
  wakeup();
```

Thread 2

```
// lock()
while()
// critical section
```

Wrong thread gets the lock

```
void lock(lock_t *m) {
  while (xchg(&m->guard, 1) == 1)
    ; // acquire guard lock
  if (m->flag == 0)
    m->flag = 1; // lock is acquired
    m->guard = 0;
  else
    queue_add(m->q, gettid());
    setpark();  // are going to sleep
    m->guard = 0;
    park();      // sleep
}
```

```
typedef struct {
  int flag = 0;
  int guard = 0;
  queue *q;
} lock_t
```

```
void unlock(lock_t *m) {
  while (xchg(&m->guard, 1) == 1)
    ; // acquire guard lock
  if (queue_empty(m->q))
    m->flag = 0; // no one wants the lock
  else
    unpark(queue_remove(m->q));  // wakeup
  m->guard = 0;
}
```

Example of Solaris (see textbook for more details)

Locks for User Apps

Linux 系统中提供 **`futex` (fast user space mutex)** 系统调用

`futex_wait(int *address, int expected)`

- 原子性地判断 `address` 地址上的值是否和 `expected` 相等，并在相等时阻塞调用该操作的线程
- 如果不相等，则直接返回

`futex_wake(int *address)`

- 唤醒一个正阻塞在 `address` 上的线程

Spin or Block

选择 Spin 还是 Block 实现取决于临界区的长短、以及对锁的争用情况

- Spin 实现
  - 成功获得锁时直接进入临界区 (low cost on success)
  - 但在失败时会浪费 CPU 时间自旋 (high cost on failed)
- Block 实现
  - 避免了无法获得锁时的自旋开销 (reduce cost on failed)
  - 但每次申请和释放锁时 (即使当前无人争抢锁) 都要陷入内核 (certain cost even on success)

Two-Phase Locking

Combine the best of spin and block

结合 Spin 和 Block 的优点实现一种两阶段的锁 (a hybrid approach)

- 现实中解决性能优化问题的一种常见思路
  - **Fast path:** 在获取不到锁时先自旋一会儿，期望锁能尽快被释放
  - **Slow path:** 如果还是无法获得锁，则阻塞自己

- `pthread_mutex_lock()`和 `pthread_mutex_unlock()` 提供了一个高可扩展的实现

```
#define UNLOCK    0
#define ONE_HOLD  1
#define WAITERS   2
```

```
void unlock(mutex_lock_t* lk){
  // state can only be ONE_HOLD or WAITERS
  if (atomic_dec(lk) != ONE_HOLD) {
    // has more than one waiters
    lk = UNLOCK;
    futex_wake(lk);
  }
}
```

```
void lock(mutex_lock_t* lk) {
  int c = cmpxchg(lk, UNLOCK, ONE_HOLD);
  // if the lock is previously UNLOCKED, there is nothing else to do
  // otherwise, we will probably have to wait
  if (c != UNLOCK){
    do {
      // if the lock is ONE_HOLD, now there are waiters (cmpxchg)
      if (c == WAITERS || cmpxchg(lk, ONE_HOLD, WAITERS) != UNLOCK)
        futex_wait(lk, WAITERS);
      // once futex_wait returns, or we did not make the call
      // another attempt to take the lock
     } while ((c = cmpxchg(&lk, UNLOCK, WAITERS)) != UNLOCK);
    }
  }
}
```

See Futexes are tricky by Ulrich Drepper for more details