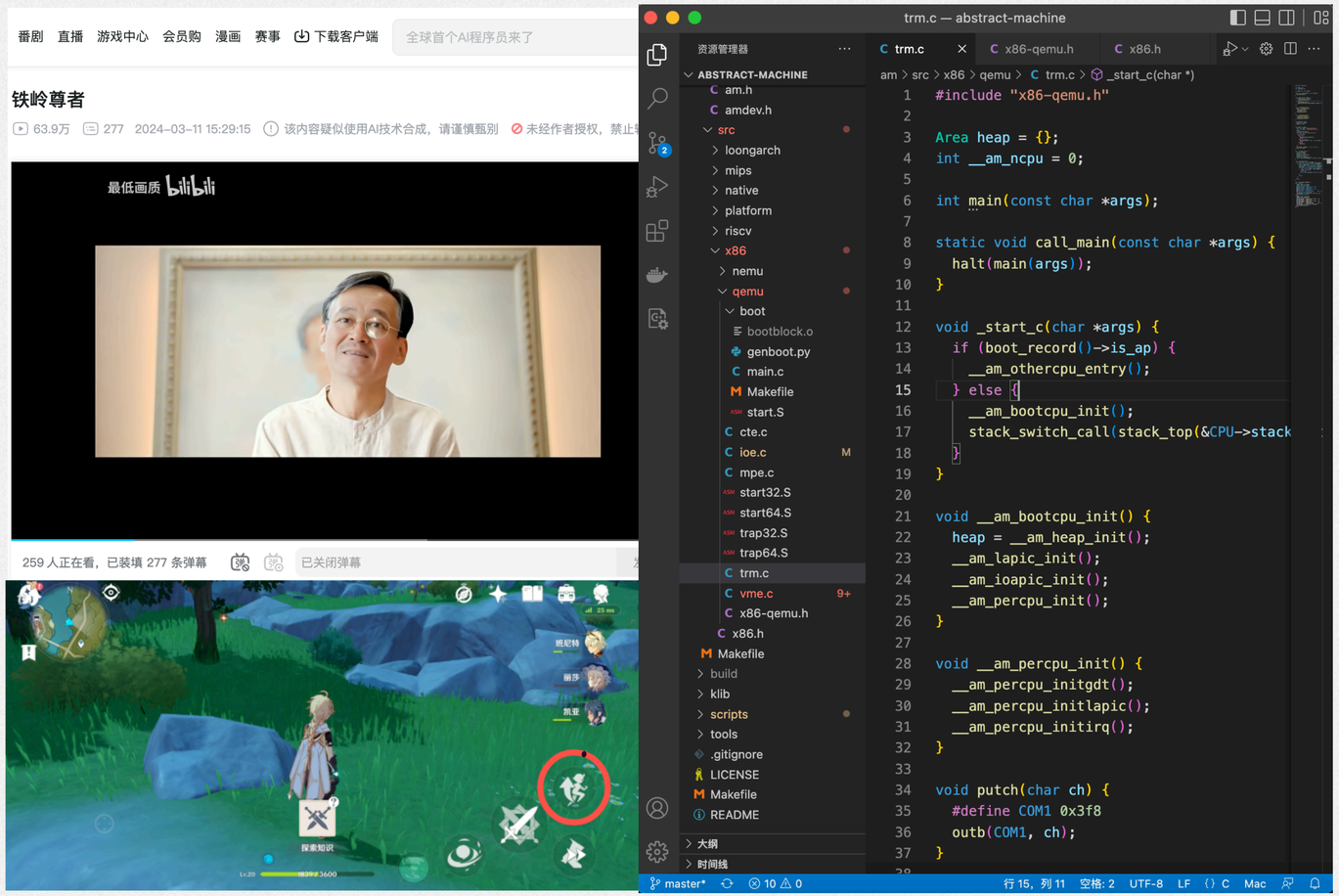
多处理器编程
Multiprocessor Programming

钮鑫涛
南京大学
2024春

本门课由蒋炎岩和钮鑫涛共同构建

人类多任务(Human multitasking)

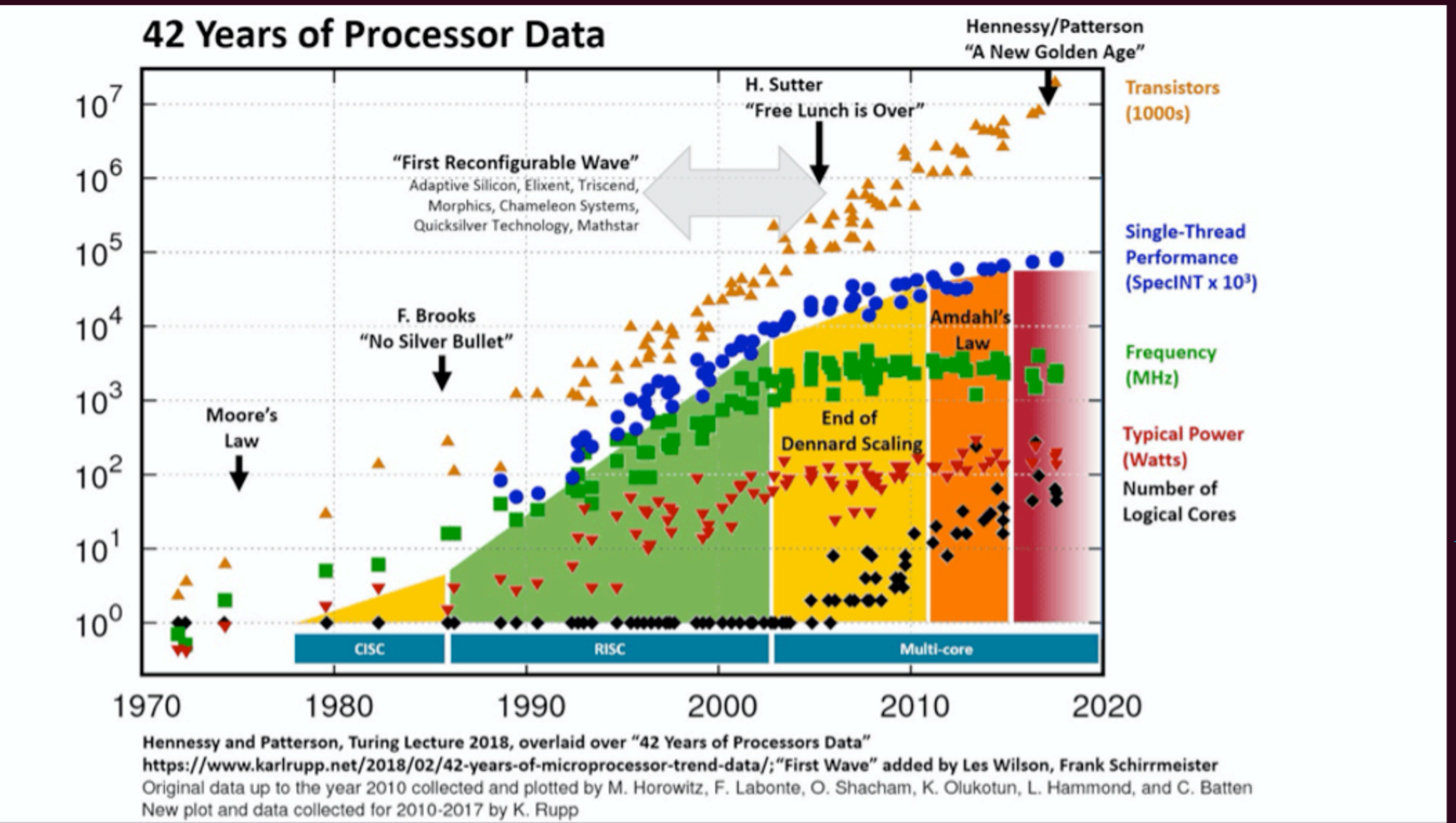


难以做到，至少对于我而言😅

- 人类天生是一个“线性”的动物
- 多任务往往是违反直觉的
- Fact about human multitasking:
  - In 2008, it was estimated that $650 billion a year is wasted in US businesses due to multitasking.

为什么我们要在一个操作系统课学这个?

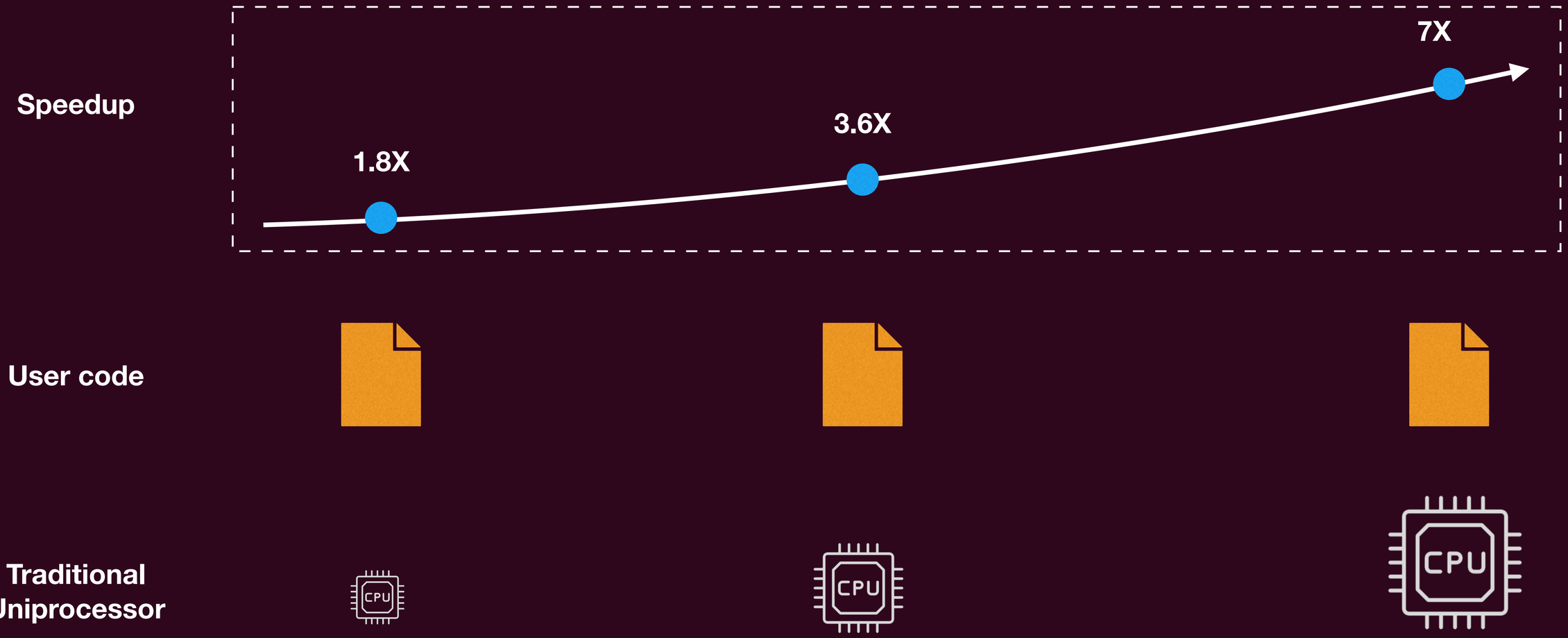
- “美好的”单处理器时代已经过去了，现在是“万恶的”多处理器时代!



一块集成电路板上可容纳的晶体管的数目，每2年翻一番。但是自21世纪初开始，时钟速度基本上保持不变，每个处理器核心的性能只得到了很少的提升，想要继续提升性能，只能...堆更多核心，指望并行计算!

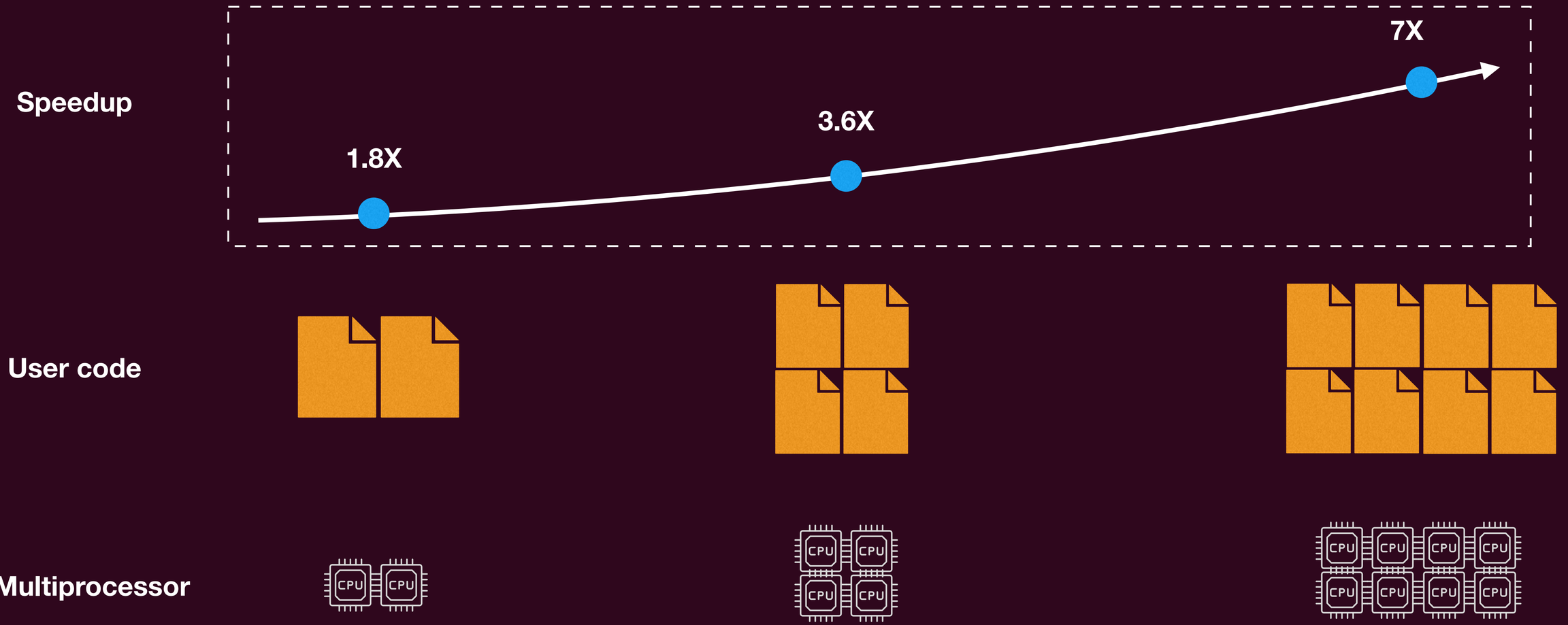
为什么我们要在一个操作系统课学这个?

- 在单处理（单核心）上，程序员是轻松愉快的，只要写一次程序，其他的交给“摩尔定律”就行了，随着处理器越来越快，他们的程序也会越来越快！



为什么我们要在一个操作系统课学这个?

- 如果核心的速度上不去，堆核心数，然后将代码并行化，是不是也能达到相同的效果? Unfortunately, not so simple…



为什么我们要在一个操作系统课学这个?

- Amdahl's Law:
  - 我们定义并行加速比：$\text{speedup} = \dfrac{\text{1-thread execution time}}{\text{n-thread execution time}}$
  - 让程序中可以被并行化的指令部分的比例为p, 而程序不可并行化的部分为$1-p$（注：不是所有逻辑都是可以并行化，如果存在前后依赖的话就难以并行化）。
  - 那么在有n个并行执行流的情况下，程序的加速比（相比于单个执行流）为

$$\text{speedp} = \frac{1}{1-p+\frac{p}{n}}$$

为什么我们要在一个操作系统课学这个?

- Amdahl's Law： $\text{speedp} = \frac{1}{1-p+\frac{p}{n}}$

  $1-p$: Sequential fraction; p: Parallel fraction; n: number of threads

  - 即使有无限多的CPU, 受限于可以并行化的部分比例，也不可能无限加速

- *实际上，有时甚至不是n越大越好，比如Parallel Max问题：

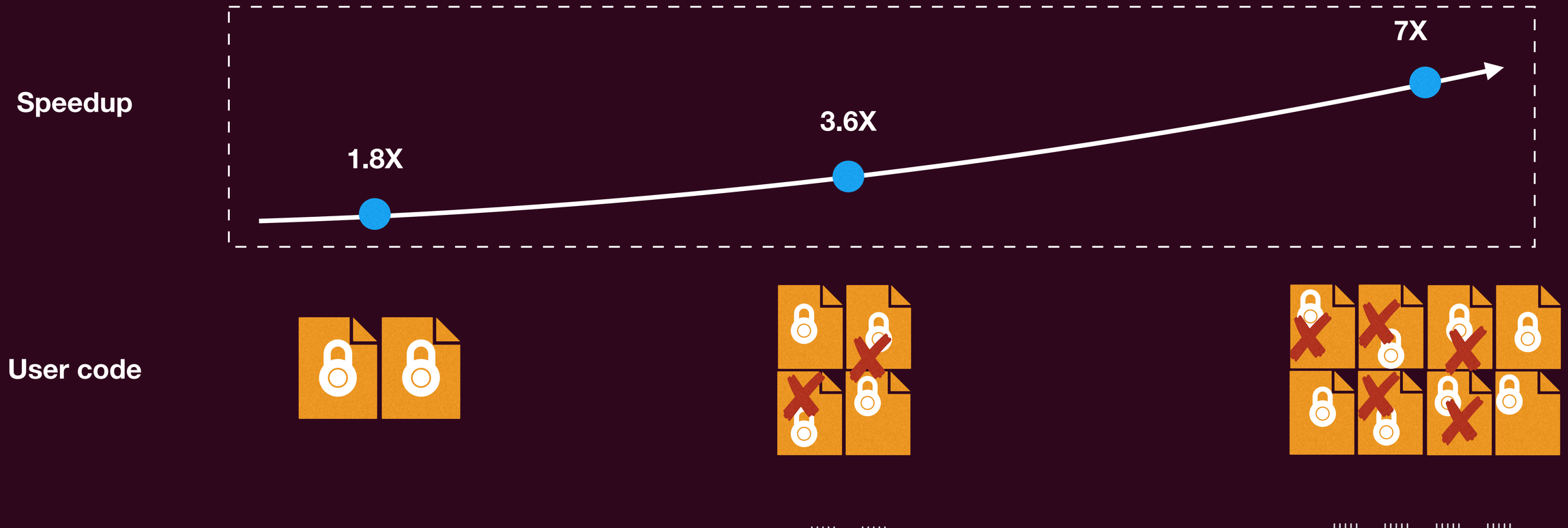
  - 计算n个不同数字中最大的数字，计算模型：可以并行地计算，基于比较的计算

  - “两轮”的并行比较才最优？（注意：如果想要“一轮”做完的话，最起码需要$\binom{n}{2}$个执行流才行！）

*Turán Theorem and Alternation technique in Combinatorics

为什么我们要在一个操作系统课学这个?

- 当然更加严重的问题(也是这门课关心的问题)是：代码的并行化和同步并不简单，非常容易出错！ 人类是天生的线性动物



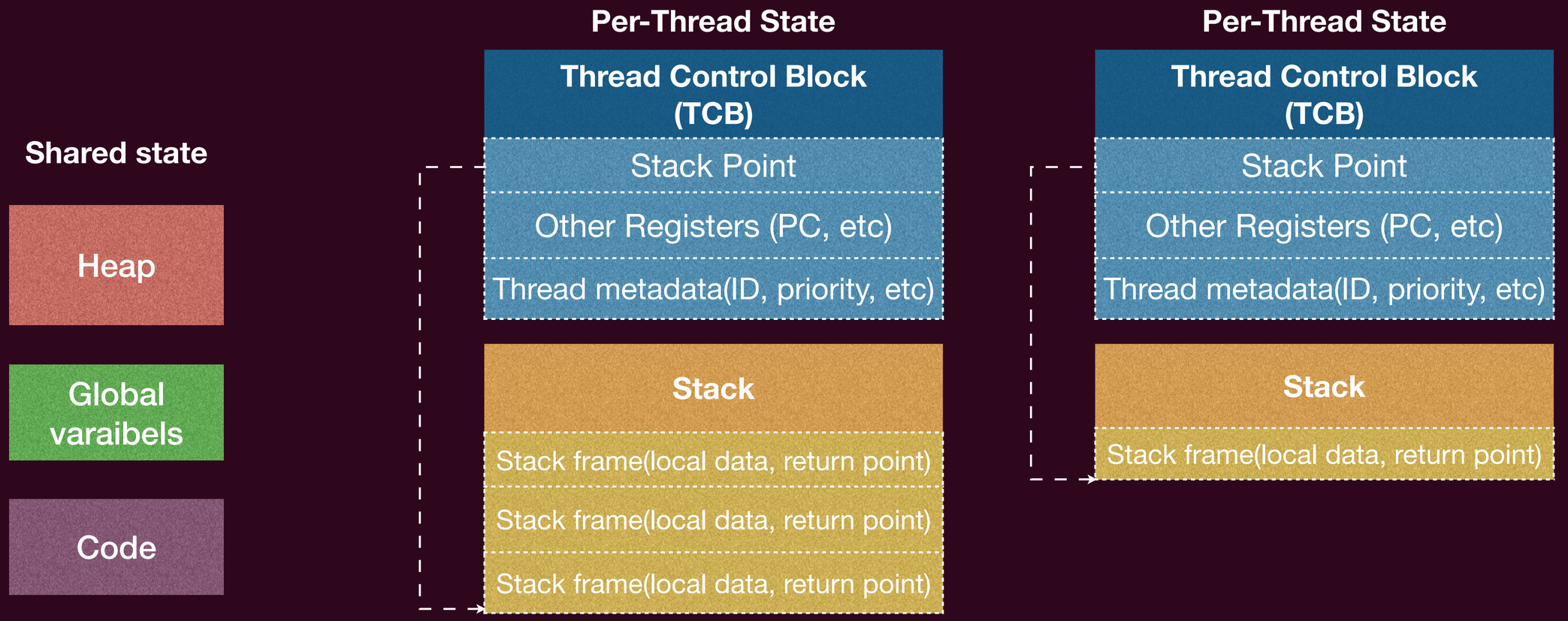
为什么我们要在一个操作系统课学这个?

- 操作系统本身就是世界上第一个并发程序!
  - 并发就是操作系统的核心之一
    - 操作系统的很多内部数据结构（如进程列表、页表、文件系统结构）都得考虑数据竞争的可能。
  - 并发的很多技术都是源自于操作系统的设计需求和其相应的解决方案。
- 学习和训练这种反直觉的程序设计模式才能渐渐让其“直觉化”

多线程编程入门

并发的基本单位是线程（Thread）

- 什么是线程：共享内存的执行流
  - 拥有独立的“上下文”和栈帧列表，共享全局变量、堆空间

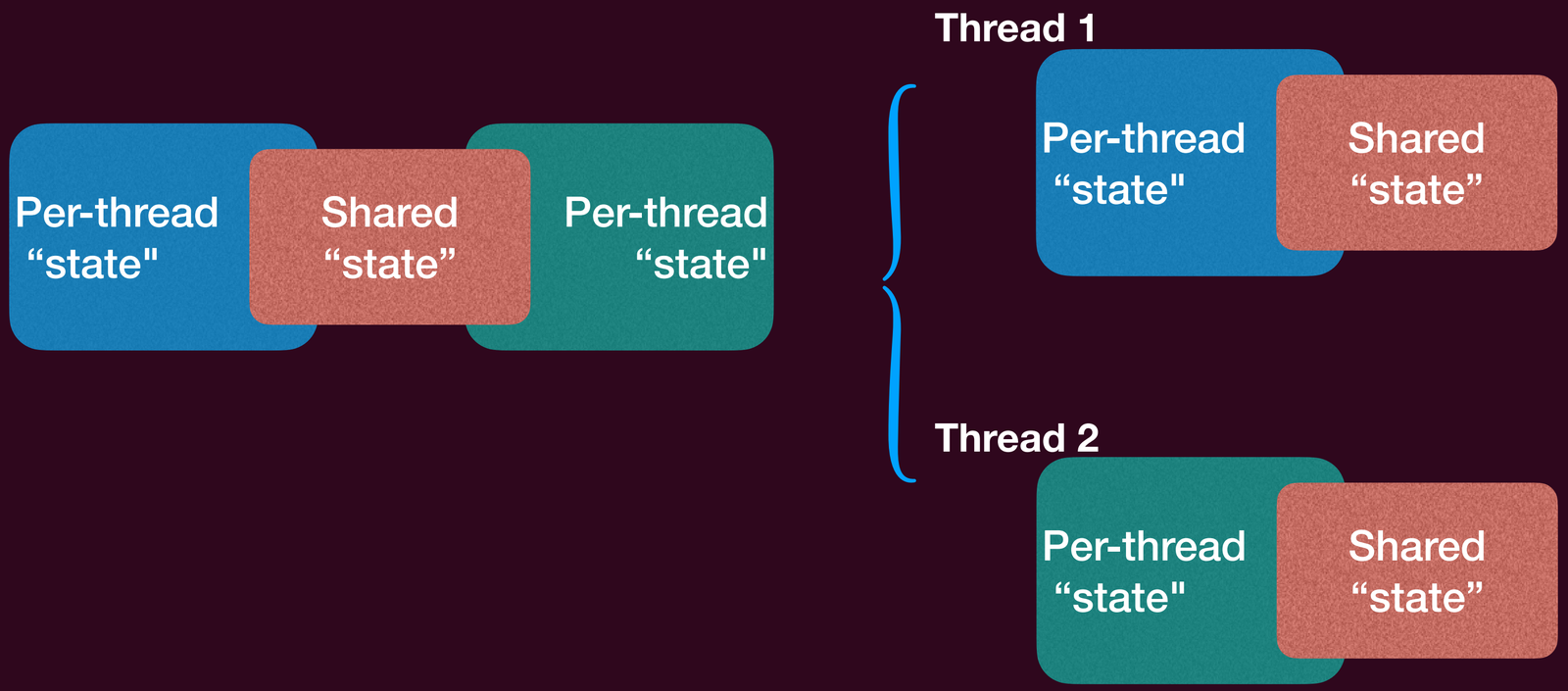


线程就是代表着程序的“执行”单位，操作系统可以随时运行、暂停、和恢复执行它。

有了线程，我们可以“线性的”写多个执行流，然后他们可以“并发”的执行，这个抽象使得我们更容易掌控并发！

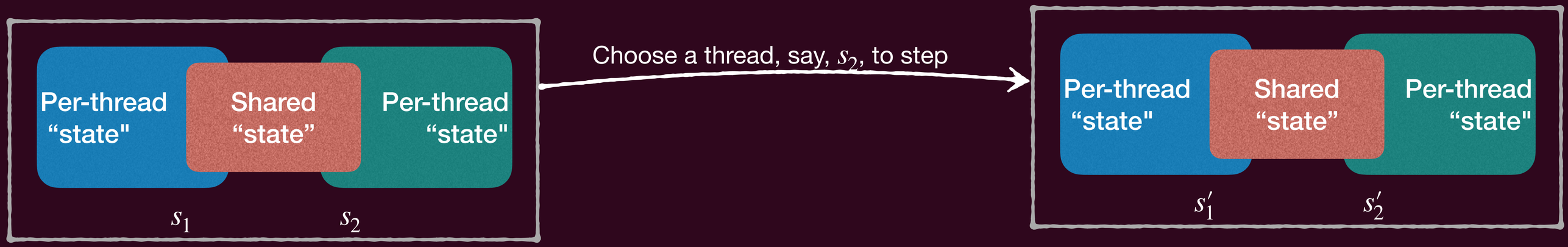
多线程编程模型

- 从状态机的角度来看，多个线程就是多个共享内存的状态机
  - 初始状态为线程创建时刻的状态， 状态迁移为调度器任意选择一个线程执行一步



多线程编程模型

- 从状态机的角度来看，多个线程就是多个共享内存的状态机
  - 初始状态为线程创建时刻的状态， 状态迁移为调度器任意选择一个线程执行一步



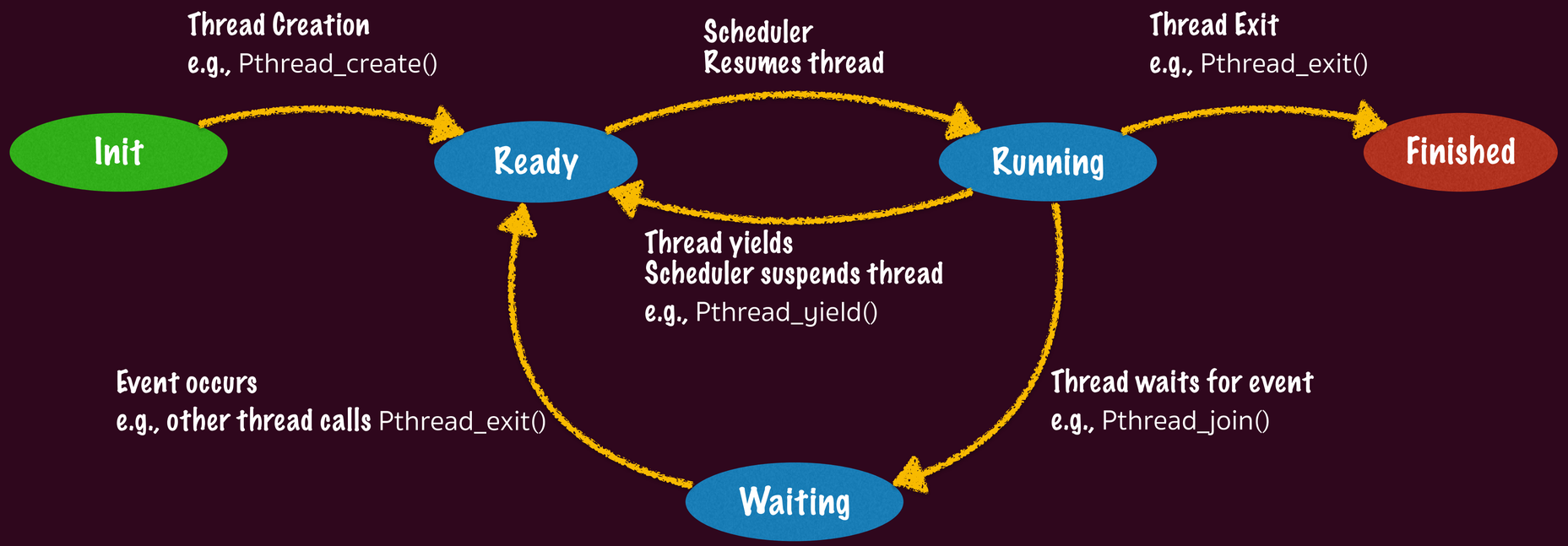
值得注意的是：虽然这里是thread$_2$执行一步$s_2 \to s_2'$，然而这个状态变化可能包括共享状态，因此可能顺带着改变了thread$_1$的状态$s_1 \to s_1'$。 这里和以往地方不同的地方在于，thread$_1$并没有主动“求变”，而是其不知道的情况下发生了改变！

入门：Posix基本线程API

Thread call	描述
`Pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void*), void * arg)`	创建一个线程thread，其将以实参（arg）运行函数(start_routine）。 attr是这个线程的属性，默认为NULL，可以通过**pthread_attr_init**函数来初始化属性如栈大小、优先级等。
`Pthread_exit(void *retval)`	在线程调用这个函数时，结束自己这个线程，并向Pthread_join自己的线程返回值retval。(注：即使不调用该函数，start_routine结束时改线程也结束，并并向Pthread_join自己的线程返回return语句的值)
`Pthread_join(pthread_t thread, void **retval)`	(阻塞自己)等待线程thread结束，thread返回的值将被放到retval所指定的内存
`Pthread_yield`	放弃当前CPU的使用 （现在已经Deprecated了，推荐使用sched_yield）
`Pthread_detach(pthread_t thread)`	使线程thread不被别的进程join，即使主进程结束也不被杀死，例子：pthread_detach(pthread_self()); —> 是自己“脱缰”

线程的生存周期

- 线程的一生经历初始化、就绪、运行、等待和结束的周期



- 只有在运行阶段，其Context才会在CPU上，其余都在内核栈上，当线程处于就绪阶段时其TCB在OS维护的ready列表上等待调度，当线程处于等待阶段时，其TCB在OS维护的同步等待列表上等待同步事件发生

例子

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10
void *print_hello_world(void *tid){
    printf("Hello World. Greetings from thread %d\n",tid);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;
    for(i=0; i < NUMBER_OF_THREADS; i++){
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    for (i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i], NULL);}
}
```

- 编译时需要增加 `-lpthread`

例子

- 上述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%
- 线程的先后执行顺序是不确定的

GDB调试多线程

- 基本命令：
  - `info threads`：显示所有线程
  - `thread` *`thread-id`*：跳到ID为*`thread-id`*的线程，使其为“当前线程”，可以进行查看其信息
  - `break` *`line`* `thread` *`thread-id`*: 线程ID为*`thread-id`*的线程的第*`line`*行设断点

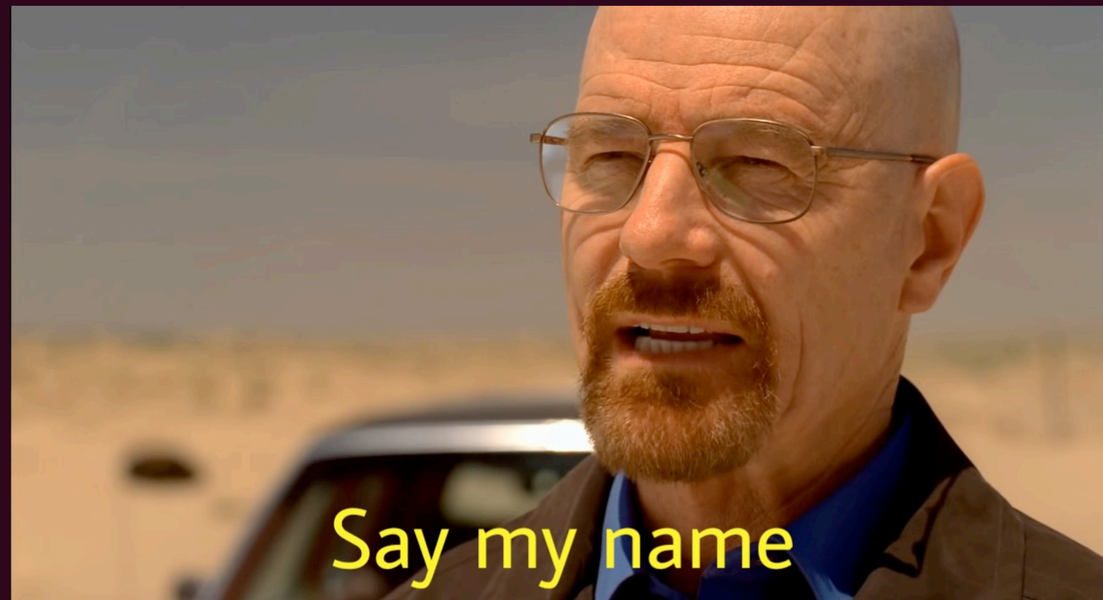
更多命令可见文档

Heisenbug

- 不确定的线程调度（到底选哪一个线程进行下一步），以及可能“被迫”改变的线程状态（由于共享内存），会使得程序非常容易出现BUG，并且这些BUG往往很微妙、难以捉摸（不是每一次执行都能确定捕捉）

- 即使我们小心处理共享数据，Heisenbugs也不受控制的产生，根因是以下三个挑战，这些挑战使得我们以往建立的编程逻辑发生改变：

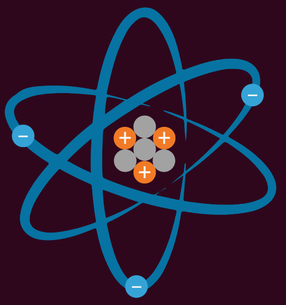
  - 原子化丧失
  - 顺序化丧失
  - 全局一致化丧失



原子性丧失

原子性(atomicity)

- 原子性: 一个原子性的操作即是一个在其“更高”的层面上无法感知到它的实现是由多个部分组成的，一般来说，其具有两个属性：
  - **[All or nothing]**: 一个原子性操作要么会按照预想那样一次全部执行完毕，要么一点也不做，不会给外界暴露中间态！
  - **[Isolation]** 一个原子性的操作共享变量时中途不会被其他操作干扰！其他所有关于这个共享变量的操作要么在这个原子性操作之前，要么在其之后！



原子性(atomicity)

- 然而就像物理世界的“原子”也可以继续分,现实的语句执行也不是原子的!

- 下面的代码结果是多少?

```c
#define N 100000000
#define NUMBER_OF_THREADS 2

long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        sum++;
    }
}

int main() {
    pthread_t threads[NUMBER_OF_THREADS];
    pthread_create(&threads[0], NULL, T_sum,  NULL);
    pthread_create(&threads[1], NULL, T_sum,  NULL);
    for (int i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i], NULL);}

    printf("sum = %ld\n", sum);
    printf("2*n = %ld\n", 2L * N);
}
```

```
mov  $sum, %rax
add 1, %rax
mov %rax, $sum
```

原子性丧失的后果

[All or nothing]违背

时间

Sequence 1

Thread 1	Thread 2
`mov $sum, %rax`	
`add 1, %rax`	
`mov %rax, $sum`	
	`mov $sum, %rax`
	`add 1, %rax`
	`mov %rax, $sum`

Sum 总共增加2

Sequence 2

Thread 1	Thread 2
`mov $sum, %rax`	
	`mov $sum, %rax`
`add 1, %rax`	
	`add 1, %rax`
`mov %rax, $sum`	
	`mov %rax, $sum`

Sum 总共增加1

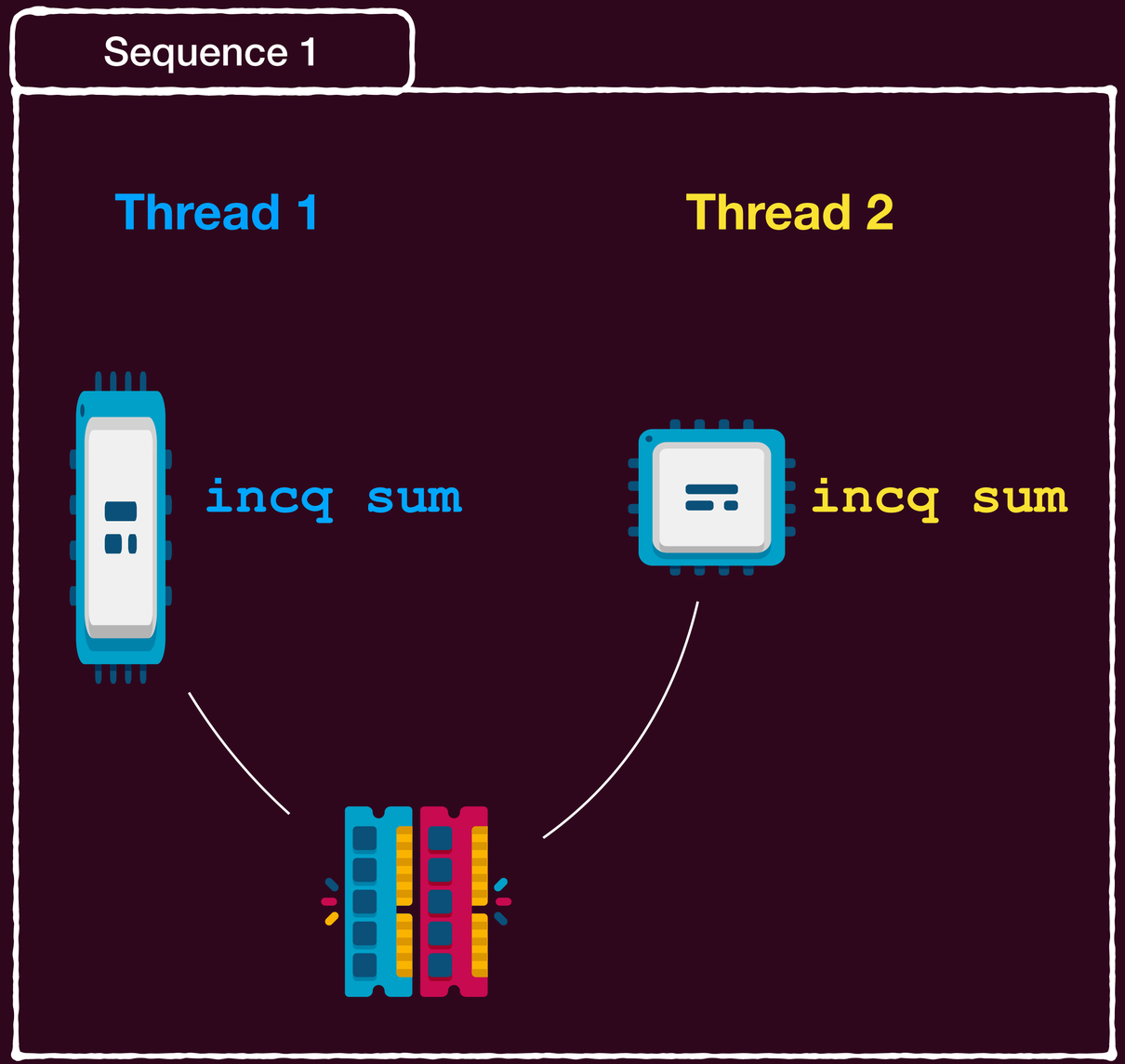
原子性(atomicity)

- 即使我们强制让sum++变成一个“单个指令”结果仍然不对！

```
#define N 100000000
#define NUMBER_OF_THREADS 2

long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        asm volatile(“incq %0": “+m"(sum));
    }
}

int main() {
    pthread_t threads[NUMBER_OF_THREADS];
    pthread_create(&threads[0], NULL, T_sum,  NULL);
    pthread_create(&threads[1], NULL, T_sum,  NULL);
    for (int i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i], NULL);}

    printf("sum = %ld\n", sum);
    printf("2*n = %ld\n", 2L * N);
}
```

原子性丧失的后果

[Isolation]违背



竞争同一个data （data race）
可能发生一个线程先做这个指令，但中途（比如图里thread1还没做完）另一个线程也做这个指令，最终导致sum的增加不对！

原子性的丧失

- 原子性的丧失
  - 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(**[All or nothing]**无法保证)
  - 多处理器多线程
    - 线程根本就是并行执行的（**[All or nothing]**和**[Isolate]**都无法保证）
- 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

实现原子性

- 互斥和原子性是本学期的重要主题
  - lock(&lk)
  - unlock(&lk)
    - 实现临界区 (critical section) 之间的绝对串行化
    - 程序的其他部分依然可以并行执行
  - 此外，操作系统需要维护一个“并发”队列，worker thread 选择队列中的进程进入临界区！

顺序性丧失

顺序性（In-order）

- 顺序性：
  - 程序语句按照既定的顺序执行！
- 然而，只要不影响语义，其实指令是否按照顺序执行并不重要！
  - 编译器就会通过reorder instructions来优化程序！
  - 这些优化在单线程下往往没有问题，但一旦到了多线程，很多逻辑就错了！

顺序性（In-order）丧失

- 如果打开 O1 选项?

```c
#define N 100000000
#define NUMBER_OF_THREADS 2

long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        sum++;
    }
}

int main() {
    pthread_t threads[NUMBER_OF_THREADS];
    pthread_create(&threads[0], NULL, T_sum,  NULL);
    pthread_create(&threads[1], NULL, T_sum,  NULL);
    for (int i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i], NULL);}

    printf("sum = %ld\n", sum);
    printf("2*n = %ld\n", 2L * N);
}
```

sum 被移出循环，最后再加回来了

顺序性（In-order）丧失

- 如果打开 O2 选项？结果好像是对的？

Unfortunately, not so simple…
[Isolation]还是违背的，只是这个错误只是很难触发而已！

```c
#define N 100000000
#define NUMBER_OF_THREADS 2

long sum = 0;

void *T_sum(void* nothing) {
    for (int i = 0; i < N; i++) {
        sum++;
    }
}

int main() {
    pthread_t threads[NUMBER_OF_THREADS];
    pthread_create(&threads[0], NULL, T_sum,  NULL);
    pthread_create(&threads[1], NULL, T_sum,  NULL);
    for (int i=0; i < NUMBER_OF_THREADS; i++){ pthread_join(threads[i], NULL);}

    printf("sum = %ld\n", sum);
    printf("2*n = %ld\n", 2L * N);
}
```

直接优化为sum += 100000000

顺序性（In-order）丧失

- 再来看一个简单的例子， -O2优化

```
bool done = false;
void *T_loop(void* nothing) {
    while (!done);
}
void *T_setbool(void* nothing) {
    done = true;
}
int main() {
    pthread_t threads[2];
    pthread_create(&threads[0], NULL, T_loop,  NULL);
    pthread_create(&threads[1], NULL, T_setbool,  NULL);
    for (int i=0; i < 2; i++){ pthread_join(threads[i], NULL);}
    printf("main stop");
}
```

被优化为了 `if (!done) while (1);`

控制执行顺序

- 方法1: 在代码中插入 “优化不能穿越” 的 barrier
  - asm volatile ("" ::: “memory");
    - Barrier 的含义是告诉编译器这里 “可以读写任何内存”
- 方法2: 使用volatile变量，标记其每次load/store为不可优化
  - bool volatile flag;

当然，这些都不是操作系统课的推荐解决方案，这门课的解决方案是：锁!

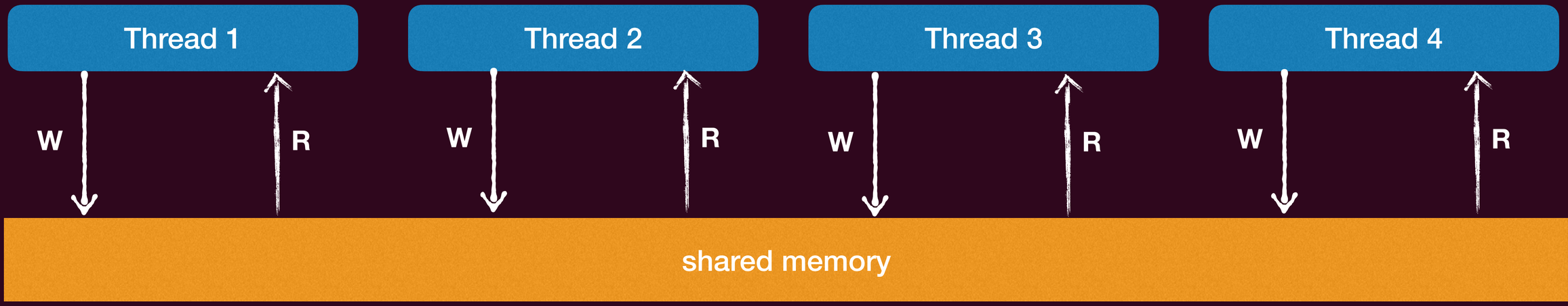
全局一致性丧失

*内存一致性模型

- 现代处理器往往允许指令乱序执行（编译器是将原始执行语句打乱，处理器本身面对指令序列进行乱序执行）。
  - 比如对于高时延的访存指令（如cache miss），处理器可以选择调度后续其他指令执行，从而隐藏访存操作的时延！
- 然而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（Memory Consistency Model，简称内存模型）明确定义了不同核心对于共享内存操作需要遵循的顺序。

顺序一致性模型(Sequential Consistency)

- 顺序一致性模型提供了以下保证:
  - 首先，不同核心看到的访存操作顺序完全一致，这个顺序称为全局顺序;
  - 其次，在这个全局顺序中，每个核心自己的读写操作可见顺序必须与其程序顺序保持一致。



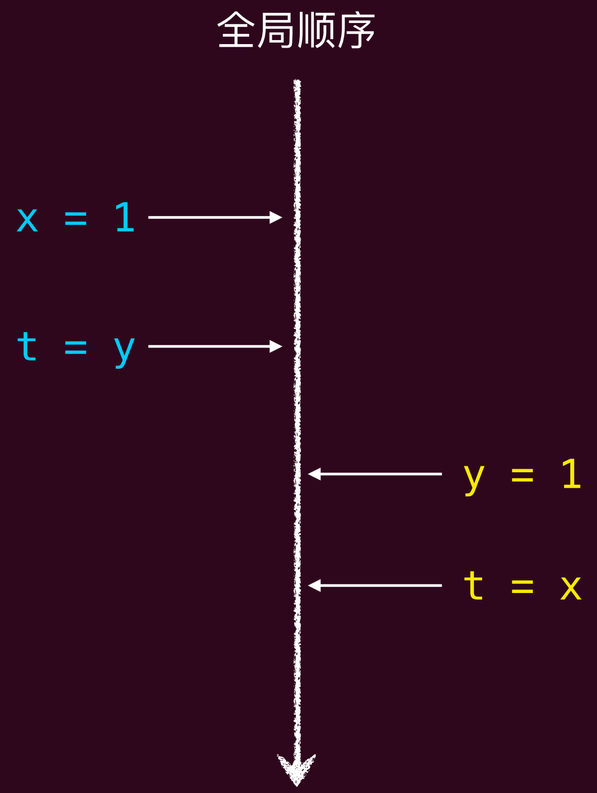
可以看成一个“时间单位”上只能选择一个线程读或写共享内存一次，最终形成一个统一的全局的访问内存的顺序

顺序一致性模型(Sequential Consistency)

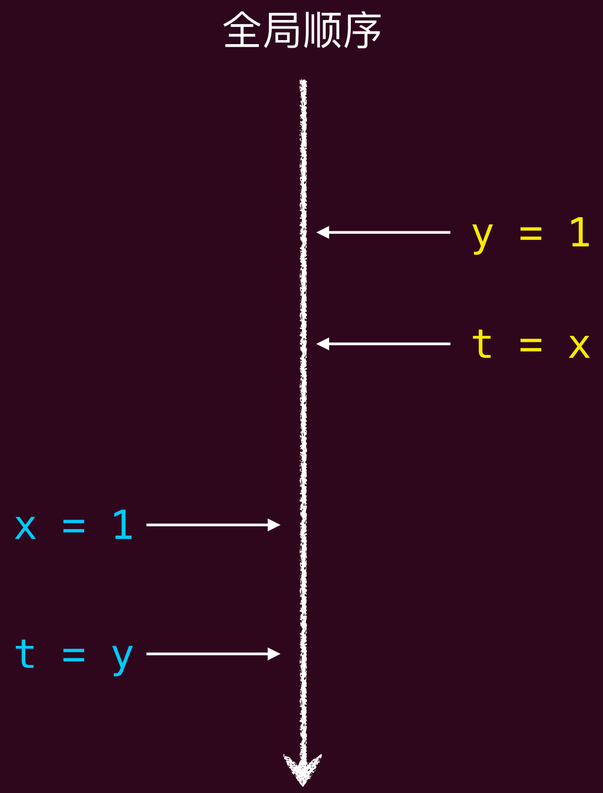
- 在此模型下下面的程序输出应该是什么?
  - 建立两个线程，线程1跑`T1()`，线程2跑`T2()`

```
int x = 0, y = 0;

void T1() {
  x = 1; int t = y; // Store(x); Load(y)
  __sync_synchronize(); // a barrier
  printf("%d", t);
}

void T2() {
  y = 1; int t = x; // Store(y); Load(x)
  __sync_synchronize();
  printf("%d", t);
}
```

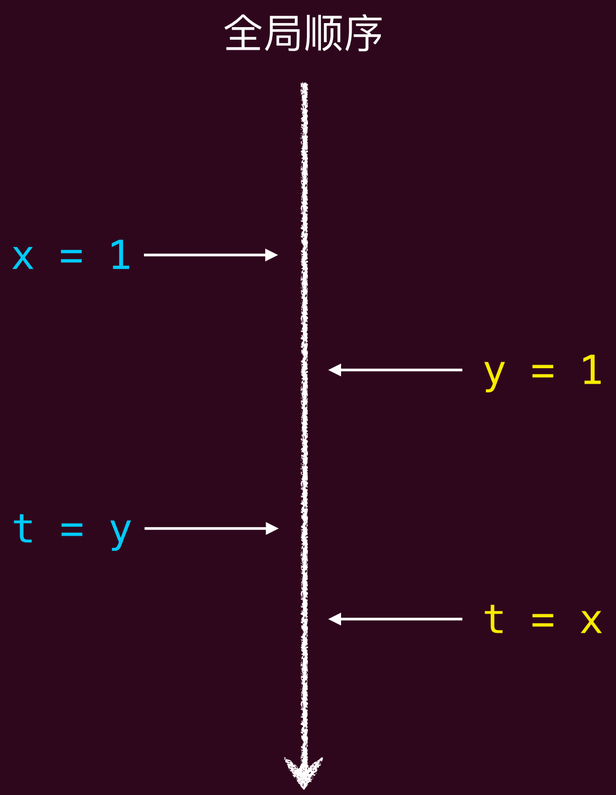
顺序一致性模型(Sequential Consistency)



打印出来的可能是
0， 1



打印出来的可能是
1， 0



打印出来的可能是
1， 1

可能打印出来 0, 0 吗? 🤔

顺序一致性模型(Sequential Consistency)

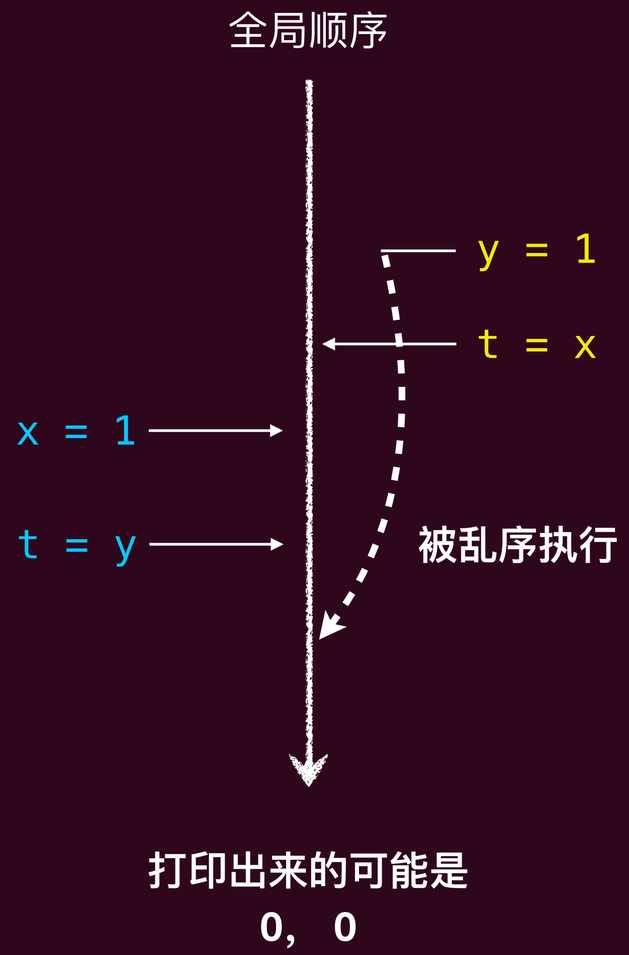
- 然而事实是：

```
ubuntu@primary:~/Home/OSCodeShow/third/automatic/mem-model$ ./mem-model | head -n 100000 | sort | uniq -c
  16798 0 0
  65322 0 1
  17878 1 0
      2 1 1
```

- 显然，x86系列的CPU不支持Sequential Consistency！
- 事实上，目前市面上已经没有支持Sequential Consistency的CPU了（Dual 386， MIPS R10000已经被淘汰了）
- 要支持Sequential Consistency在架构和性能上要付出很多！

TSO(Total Store Ordering)内存模型

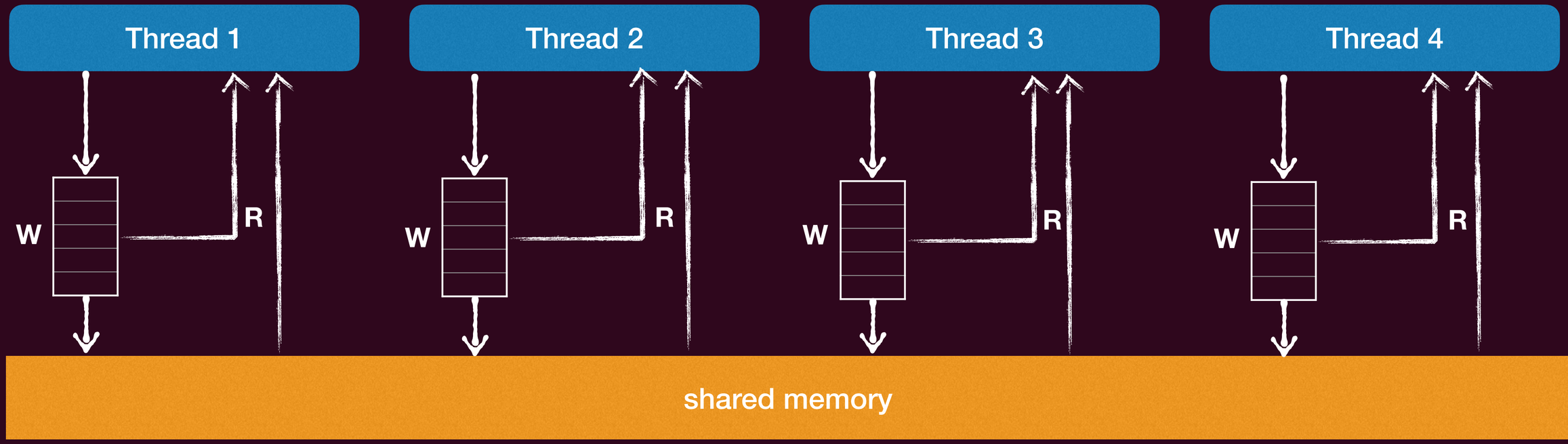
- 0，0是怎么得到的?



线程1看到线程2是先读 x，然后写y，但对于线程2而言，其是先执行写y，然后再读x（虽然这个顺序被处理器打乱了，因为处理器觉得两个顺序不重要，x和y不存在依赖），因此不再一致！

*TSO内存模型

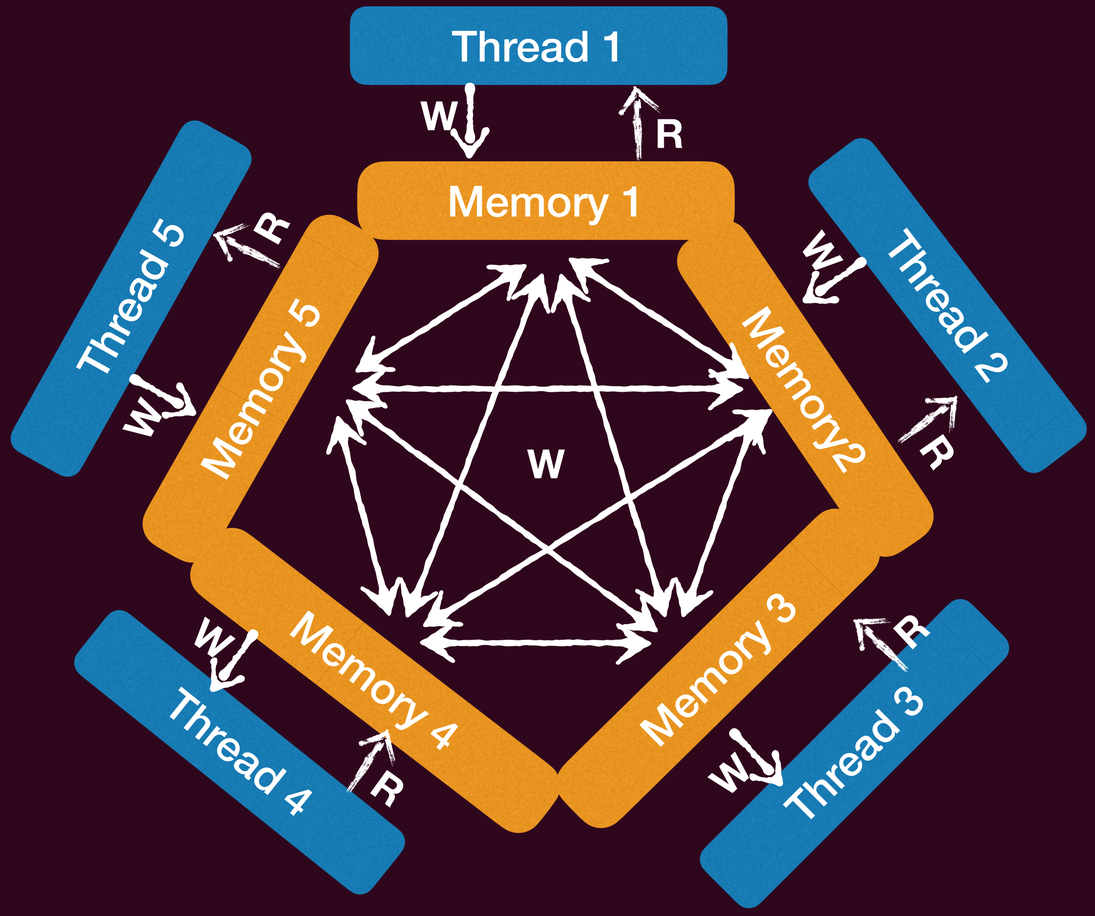
- TSO 一致性模型中，其保证对不同地址且无依赖的“读读”、“读写”、“写写”操作之间的全局可见顺序，但不保证“写读”的全局可见顺序 （X86支持的就是这个模型）



一个处理器会比其他处理器更早的看到自己的读！因此，“写读”全局不一致！但所有的处理器都对所有的写操作到共享内存的顺序是一致的

宽松内存模型(Relaxed Memory Model)

- 不保证任何不同地址且无依赖的访存操作之间的顺序，也即读读，读写，写读与写写操作之间都可以乱序全局可见(Arm和RISC-V的内存模型)。



宽松内存模型(Relaxed Memory Model)

- 下面展示了一种基于共享内存的消息传递机制。两个需要通讯的线程分别运thread_A与thread_B代码段。发送者先填充数据，再通过标记flag来通知接收者数据准备就绪。然而，在宽松内存模型下，可能会出错（写写全局顺序不一致）， TSO模型下是正确的。

```
#define NOT_READY 0;
#define READY 1;
int data = 0;
int flag = NOT_READY;

void thread_A(void){
  data = 123;
  flag = READY;
}
void thread_B(void)
{
while ( flag != READY) ; /* 循环忙等 */
  handle(data);
}
```

宽松内存模型下，想要正确，需要手动插入硬件内存屏障(__sync_synchronize()

总结

- 并发的基本单位是线程
  - 即共享部分内存的状态机（有自己的私有状态）
  - 其状态的变化可以随着另外的进程的“步进”而被动改变
- Posix提供的标准多线程编程库Pthread
- 多处理器编程充满挑战，数据竞争下难以保障正确
  - 原子性、顺序性和全局一致性都会丧失

阅读材料

- [OSTEP] 第25、26、27章