第12章 并发编程

并发编程 **Concurrent Programming**

100076202：计算机系统导论

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

Randal E. **Bryant and** David R. O’Hallaron

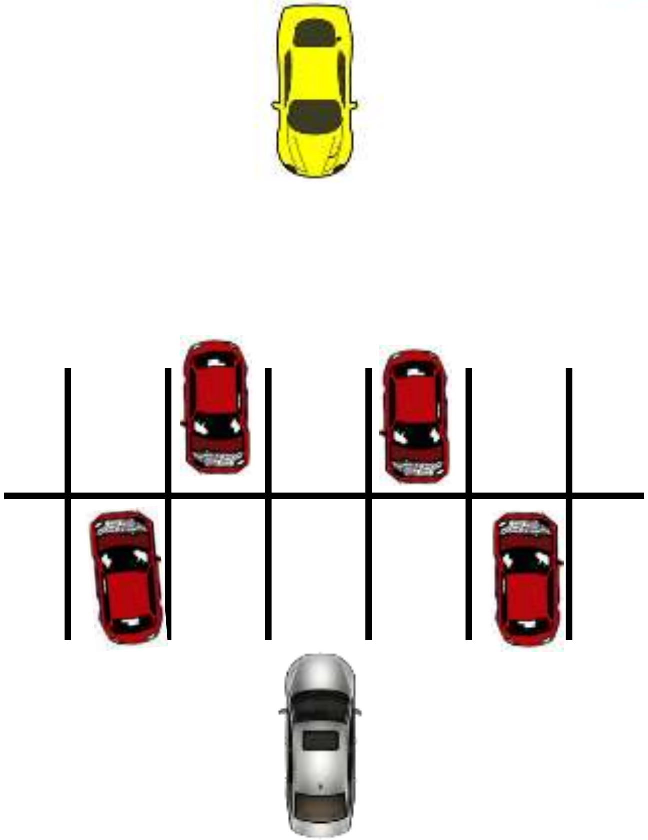
并发编程很难！
Concurrent Programming is Hard!

- 人类的思维往往是顺序的 **The human mind tends to be sequential**

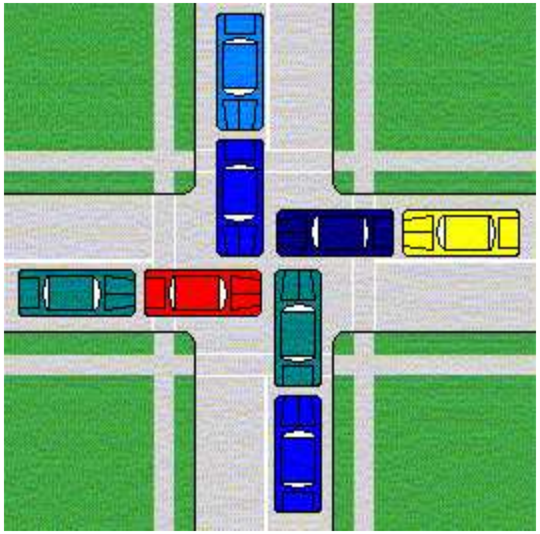
- 时间的概念常常误导人 **The notion of time is often misleading**

- 考虑计算机系统中所有可能的事件顺序非常容易出错，而且经常是不可能的 **Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible**

数据竞争 Data Race



死锁 Deadlock

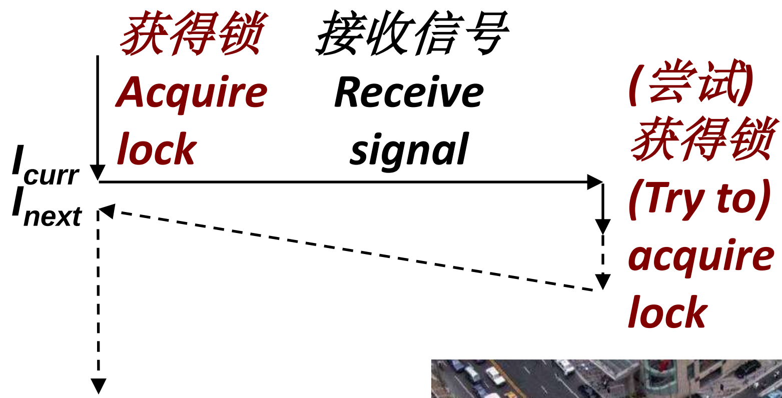


死锁 Deadlock

- 信号处理程序示例 **Example from signal handlers.**
- 为什么不在处理程序中使用**printf？ Why don't we use printf in handlers?**

```
void catch_child(int signo) {
    printf("Child exited!\n"); // this call may reenter printf/puts! BAD!  DEADLOCK!
    while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}
```

- **Printf代码： Printf code:**
  - 获得锁 Acquire lock
  - 做工作 Do something
  - 释放锁 Release lock

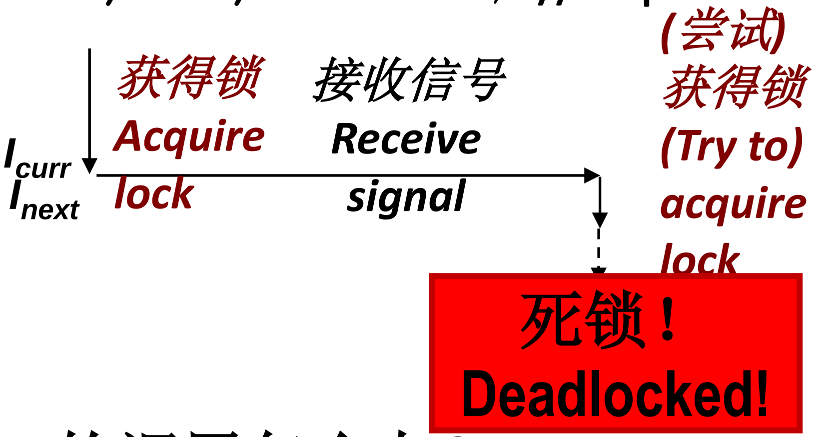


死锁 Deadlock

- 信号处理程序示例 **Example from signal handlers.**
- 为什么不在处理程序中使用printf？ **Why don't we use printf in handlers?**

```
void catch_child(int signo) {
    printf("Child exited!\n"); // this call may reenter printf/puts! BAD!  DEADLOCK!
    while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}
```

- **Printf代码： Printf code:**
  - 获得锁 Acquire lock
  - 做工作 Do something
  - 释放锁 Release lock



- 如果信号处理程序中断对printf的调用怎么办？ **What if signal handler interrupts call to printf?**

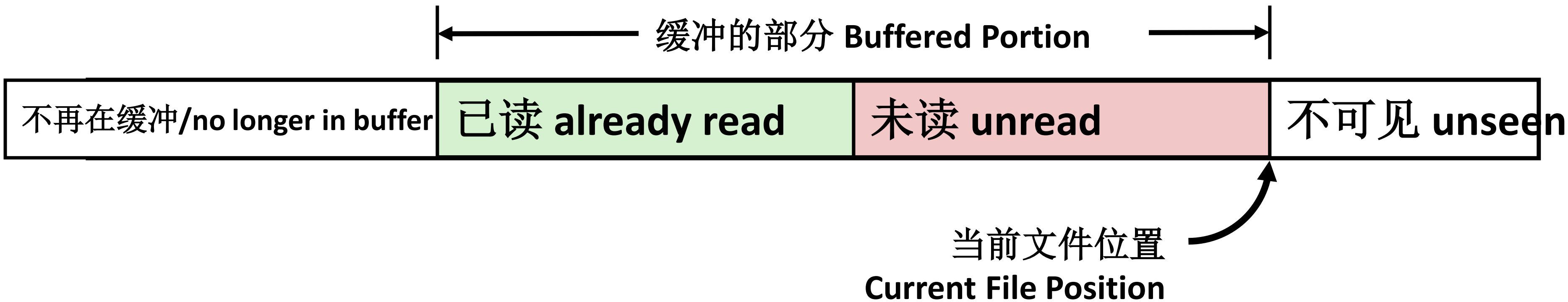
测试printf死锁 Testing Printf Deadlock

```
void catch_child(int signo) {
   printf("Child exited!\n"); // this call may reenter printf/puts! BAD!  DEADLOCK!
   while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}

int main(int argc, char** argv) {
  ...
  for (i = 0; i < 1000000; i++) {
    if (fork() == 0) {
      // in child, exit immediately
      exit(0);
    }
    // in parent
    sprintf(buf, "Child #%d started\n", i);
    printf("%s", buf);
  }
  return 0;
}
```

```
Child #0 started
Child #1 started
Child #2 started
Child #3 started
Child exited!
Child #4 started
Child exited!
Child #5 started
   .
   .
   .
Child #5888 started
Child #5889 started
```

为何printf需要锁？
Why Does Printf require Locks?

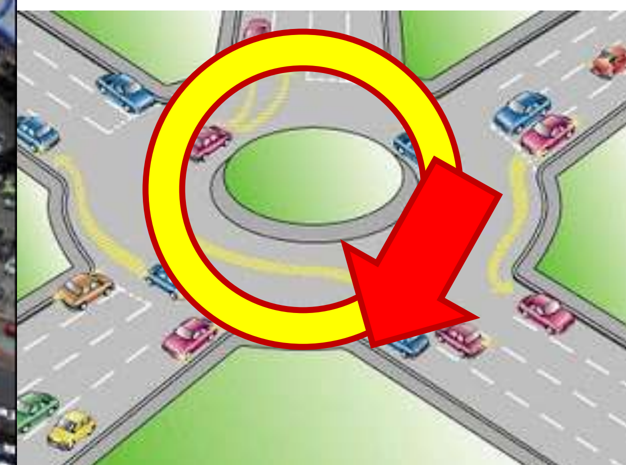
- Printf (和fprintf、sprintf)实现带缓冲的输入/输出 Printf (and fprintf, sprintf) implement *buffered* I/O



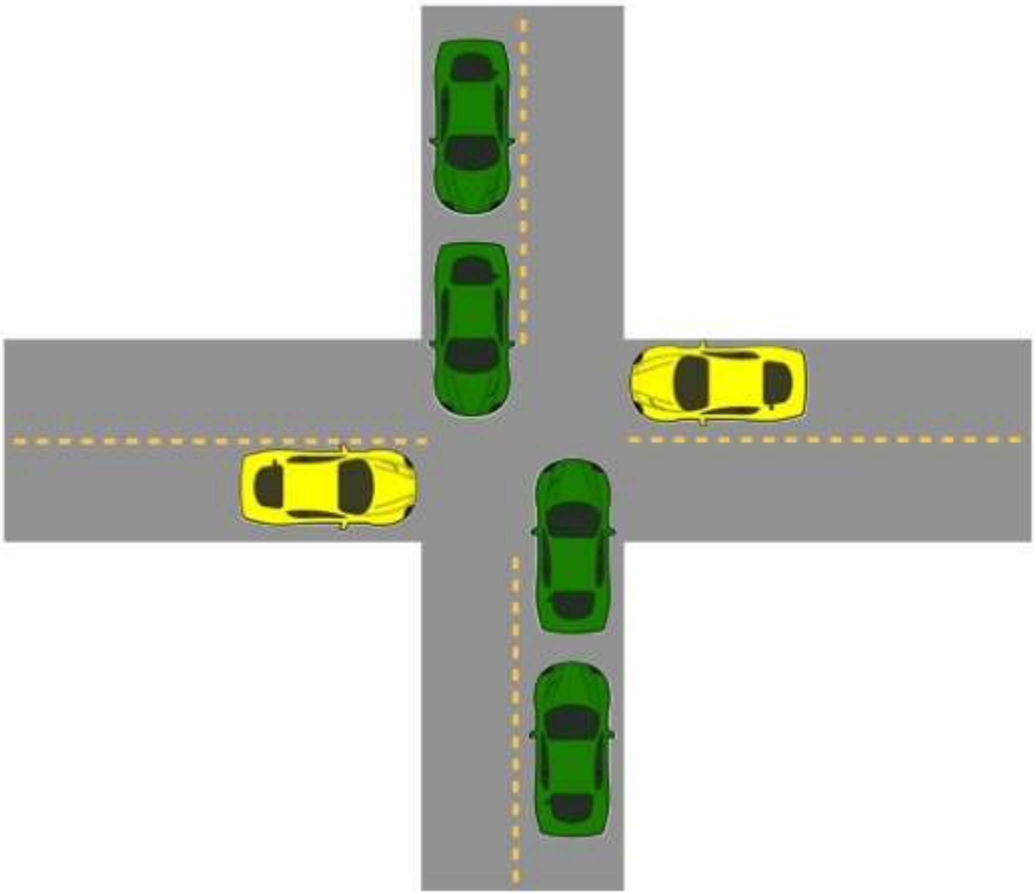
- 需要锁以访问该共享缓冲区 Require locks to access the shared buffers

活锁 Livelock

活锁 Livelock



饿死 Starvation



- 黄色车必须让位给绿色车 Yellow must yield to green
- 源源不断的绿色汽车 Continuous stream of green cars
- 整个系统取得了进展，但有些个体无限期地等待 Overall system makes progress, but some individuals wait indefinitely

并发编程很难！
Concurrent Programming is Hard!

- 并发程序的经典问题类： **Classical problem classes of concurrent programs:**
  - **竞争**：结果取决于系统其他地方的任意调度决策 ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - 示例：谁坐飞机上的最后一个座位？ Example: who gets the last seat on the airplane?
  - **死锁**：资源分配不当阻碍前进 ***Deadlock:*** improper resource allocation prevents forward progress
    - 示例：交通堵塞 Example: traffic gridlock
  - 活锁/饥饿/公平：外部事件和/或系统调度决策可能会阻止子任务进度 ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - 例如：有人总是跳到你前面排队 Example: people always jump in front of you in line

并发编程很难！
Concurrent Programming is Hard!

- 并发编程的许多方面超出了我们课程的范围。。 **Many aspects of concurrent programming are beyond the scope of our course..**
  - 但并非所有 but, not all ☺
  - 我们将在接下来的几节课中讨论这些方面 We'll cover some of these aspects in the next few lectures.

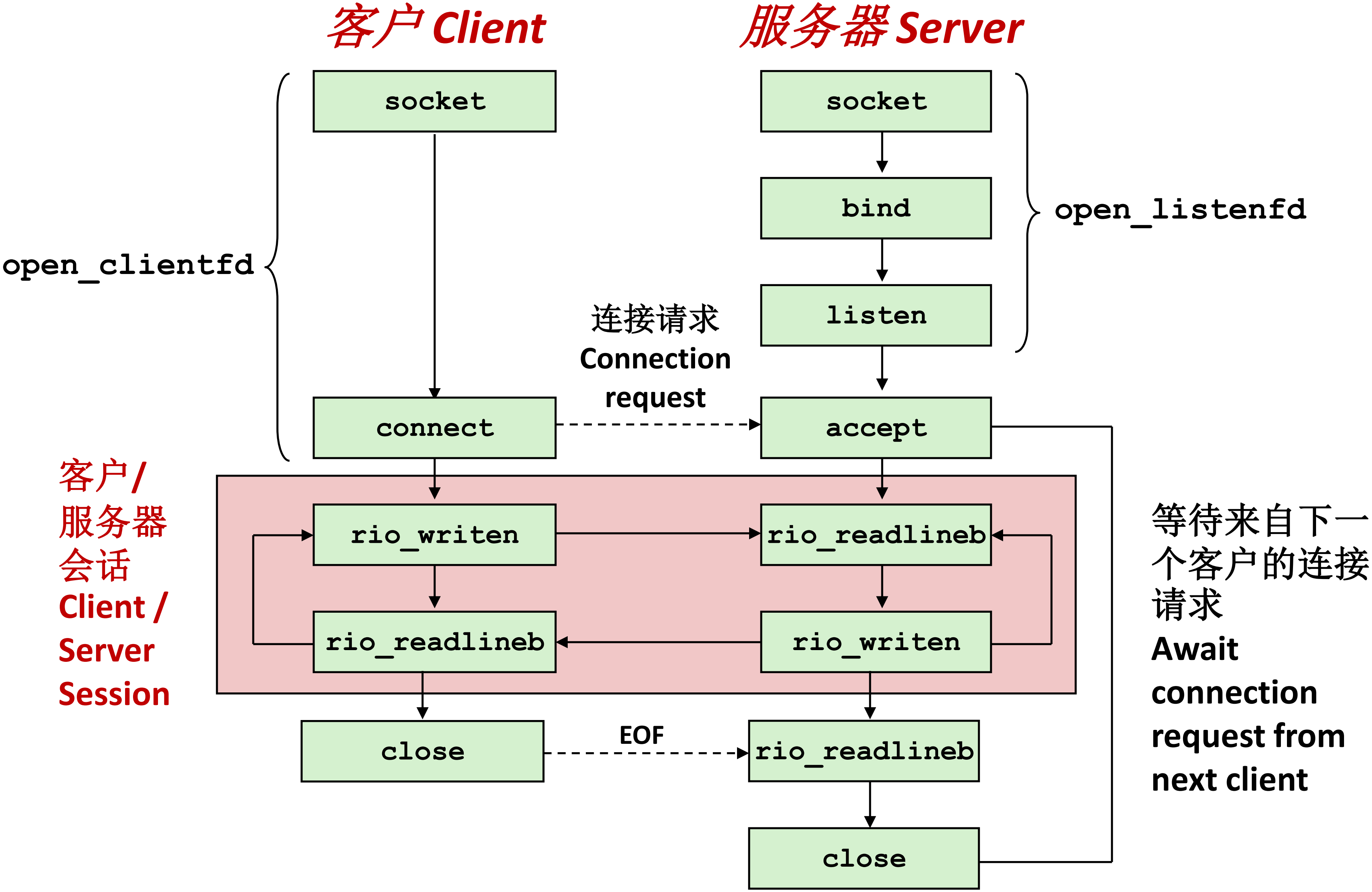
并发编程很难！
Concurrent Programming is Hard!

它可能很难，但… **It may be hard, but …**

它可能是有用的，有时也是必要的！ **it can be useful and sometimes necessary!**

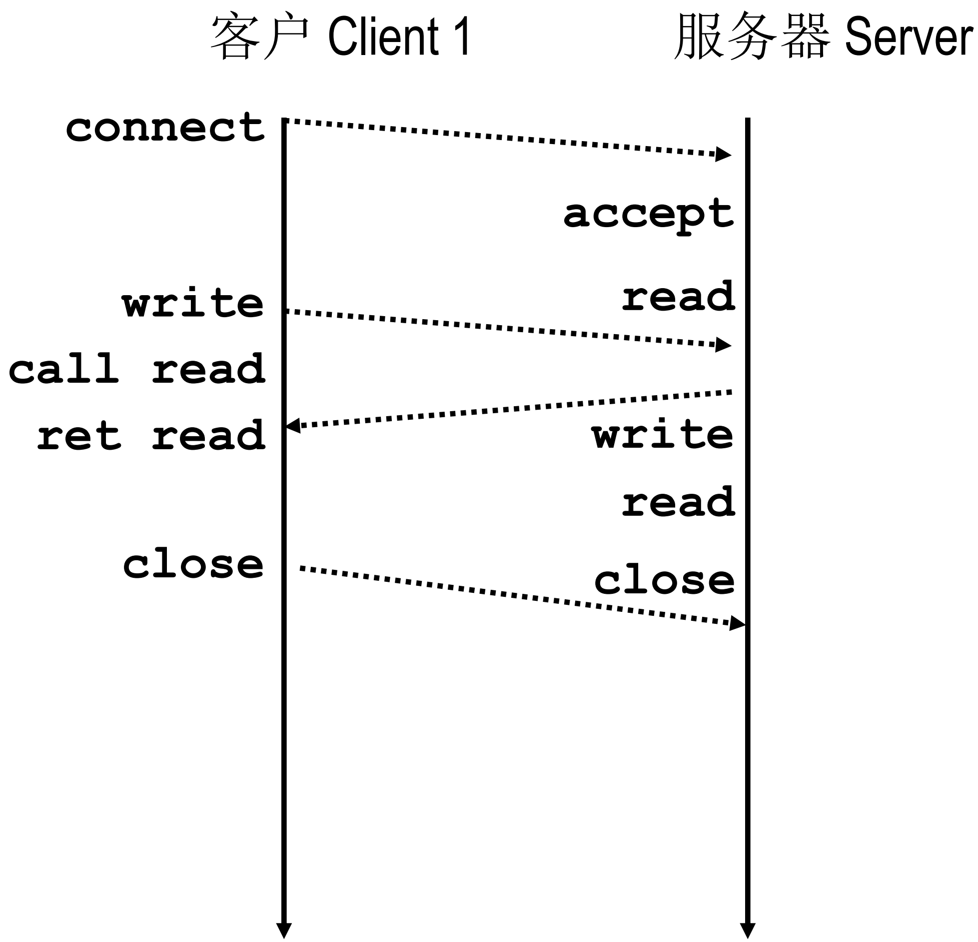
越来越有必要 **more and more necessary!**

提醒：迭代式回声服务器
Reminder: Iterative Echo Server



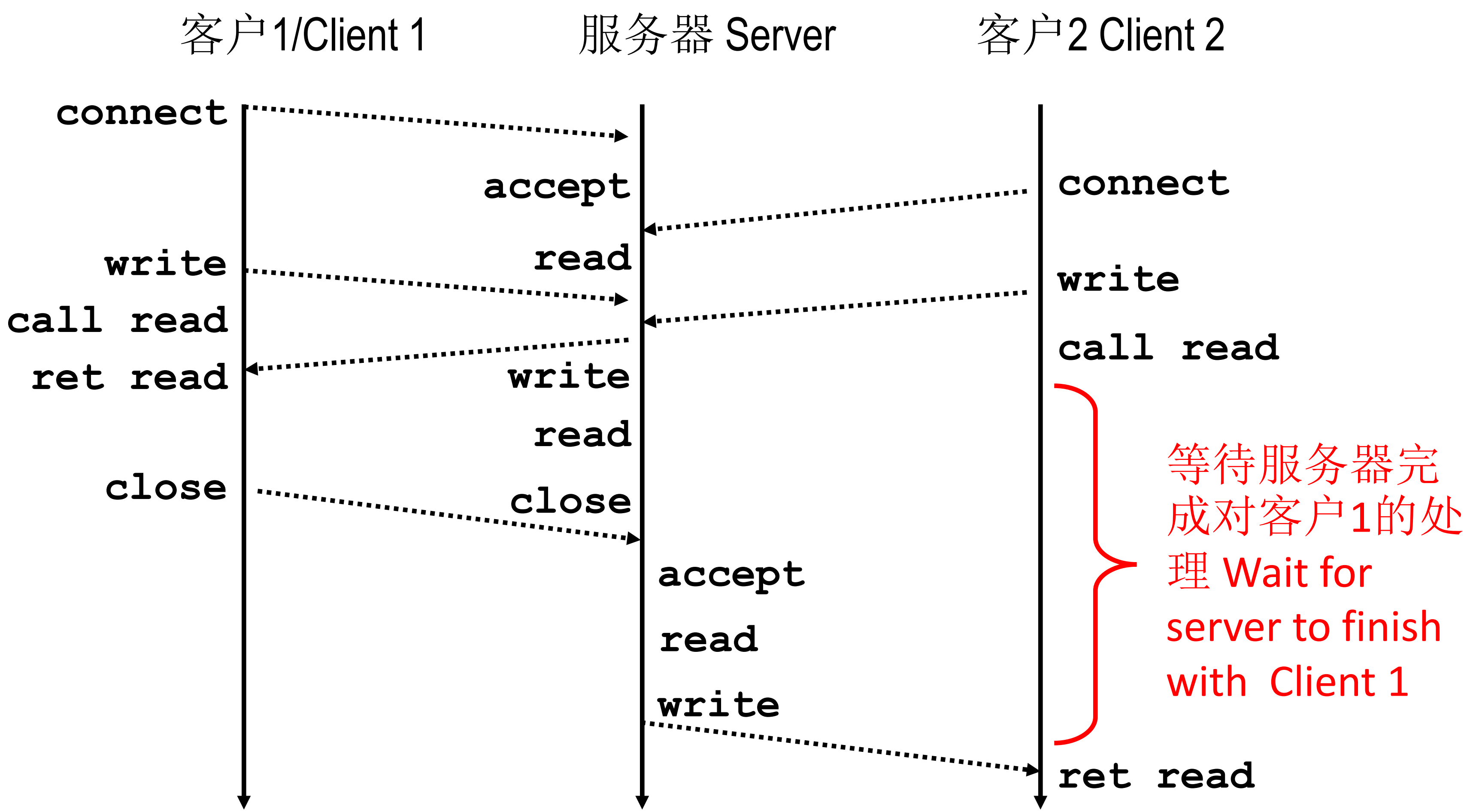
迭代服务器 Iterative Servers

- 迭代服务器一次处理一个请求 Iterative servers process one request at a time



迭代服务器 Iterative Servers

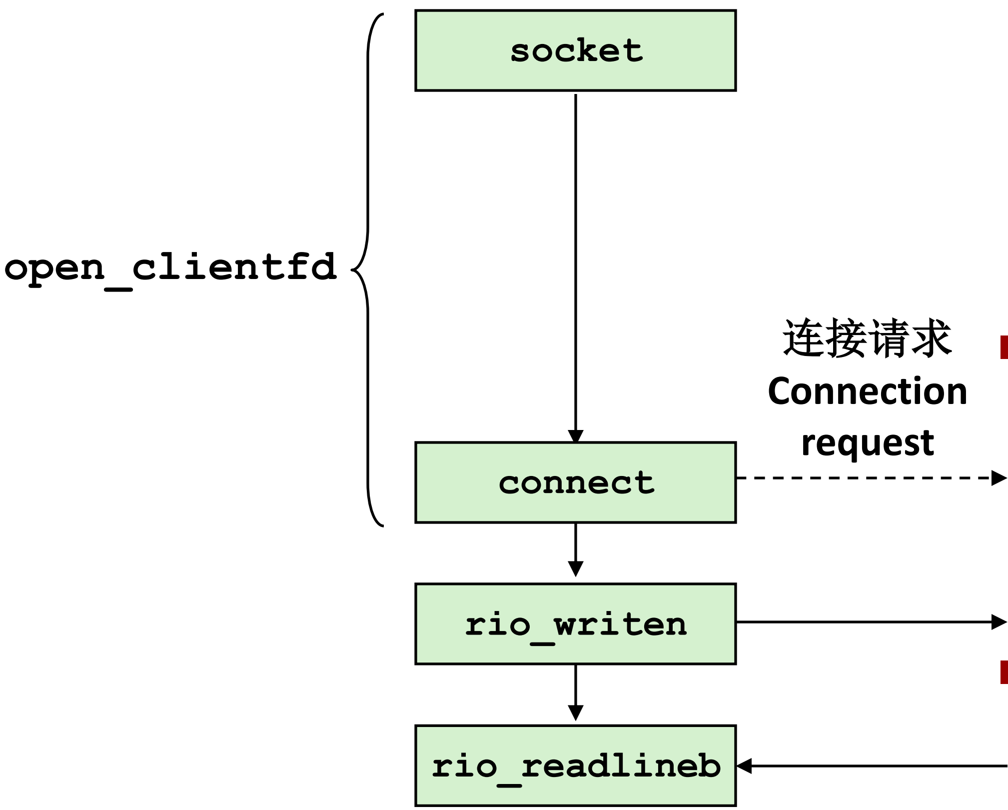
- 迭代服务器一次处理一个请求 Iterative servers process one request at a time



第二个客户阻塞在哪里？
Where Does Second Client Block?

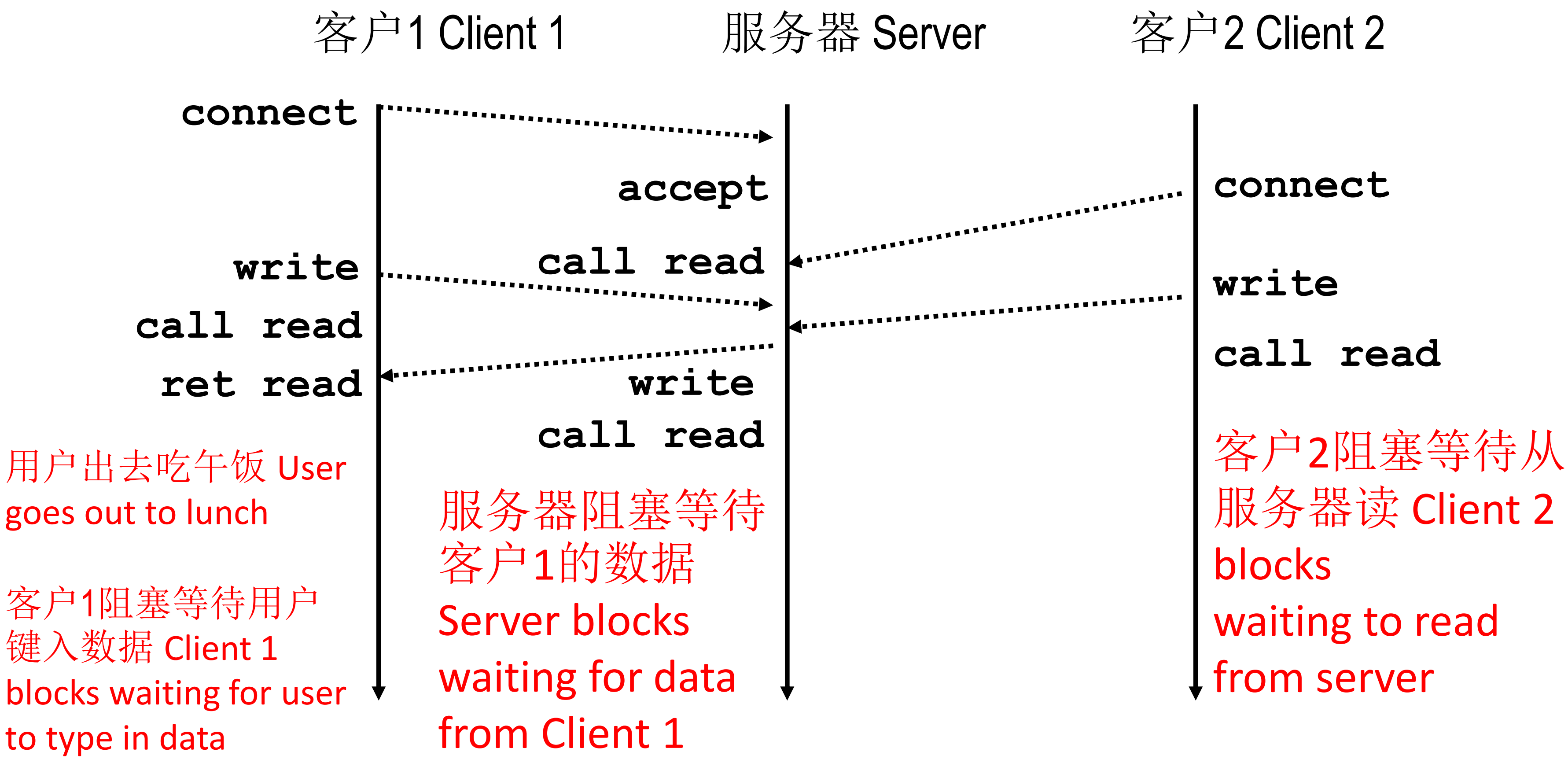
- 第二个客户尝试连接到迭代服务器 **Second client attempts to connect to iterative server**

客户 Client



- **connect调用返回 Call to connect returns**
  - 尽管连接还没有被接受 Even though connection not yet accepted
  - 服务器端TCP管理器对请求进行排队 Server side TCP manager queues request
  - 该功能称为“TCP侦听backlog” Feature known as “TCP listen backlog”
- **rio_writen调用返回 Call to rio_writen returns**
  - 服务器端TCP管理器缓冲输入数据 Server side TCP manager buffers input data
- **rio_readlineb调用阻塞 Call to rio_readlineb blocks**
  - 服务器没有写数据 Server hasn’t written anything for it to read yet.

迭代服务器的基本缺陷
Fundamental Flaw of Iterative Servers



- 解决方案：使用**并发服务器** **Solution: use *concurrent servers* instead**
  - 并发服务器使用多个并发流同时为多个客户端提供服务 Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

编写并发服务器的方法 Approaches for Writing Concurrent Servers

允许服务器并发处理多个客户 Allow server to handle multiple clients concurrently

1. 基于进程 Process-based

- 内核自动交错多个逻辑流 Kernel automatically interleaves multiple logical flows
- 每个流都有自己的私有地址空间 Each flow has its own private address space

2. 基于事件 Event-based

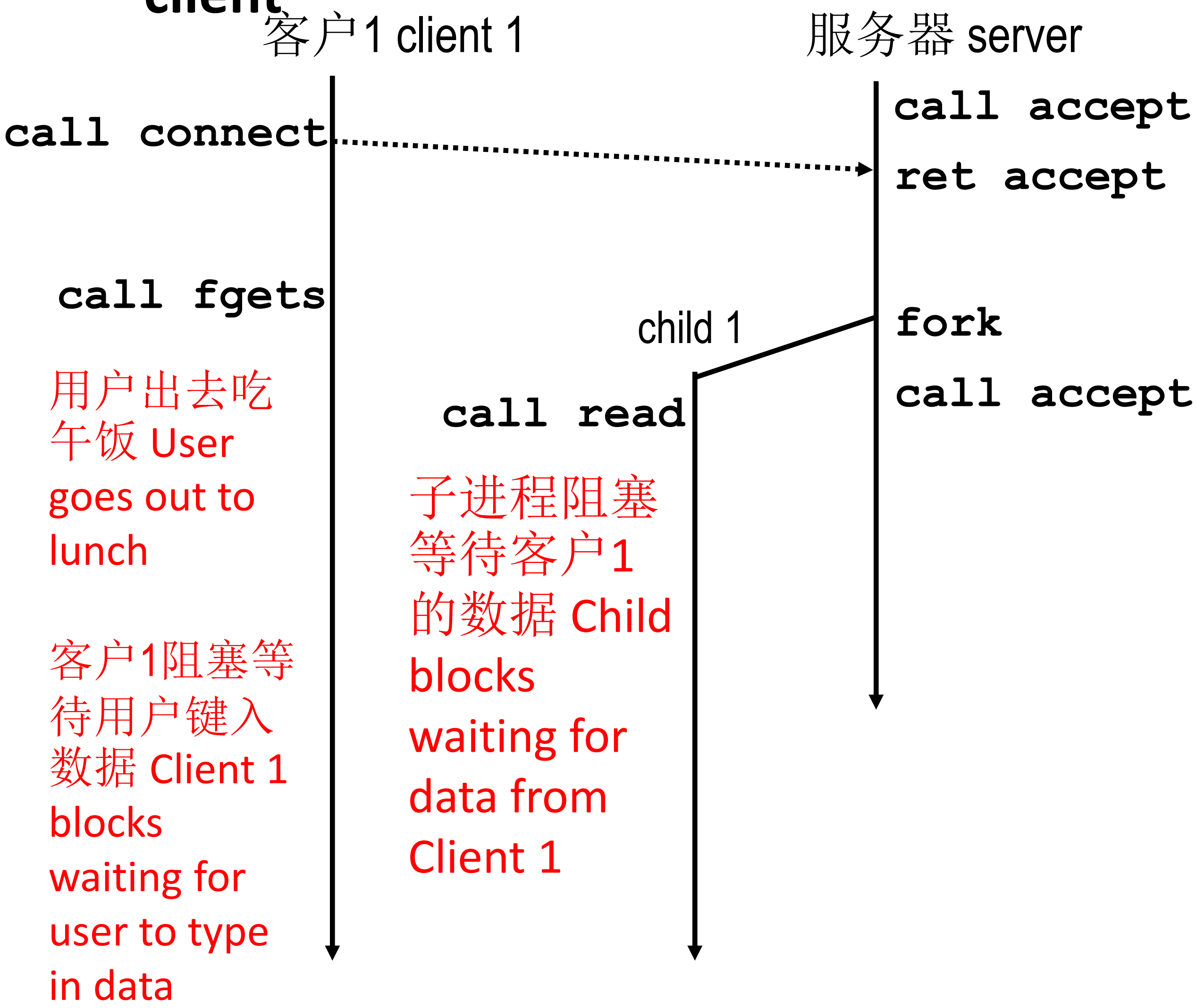
- 程序员人工交错多个逻辑流 Programmer manually interleaves multiple logical flows
- 所有流共享相同的地址空间 All flows share the same address space
- 使用称为*I/O多路复用*的技术 Uses technique called *I/O multiplexing*

3. 基于线程 Thread-based

- 内核自动交错多个逻辑流 Kernel automatically interleaves multiple logical flows
- 每个流共享相同的地址空间 Each flow shares the same address space
- 基于进程和基于事件两种方法的混合 Hybrid of of process-based and event-based

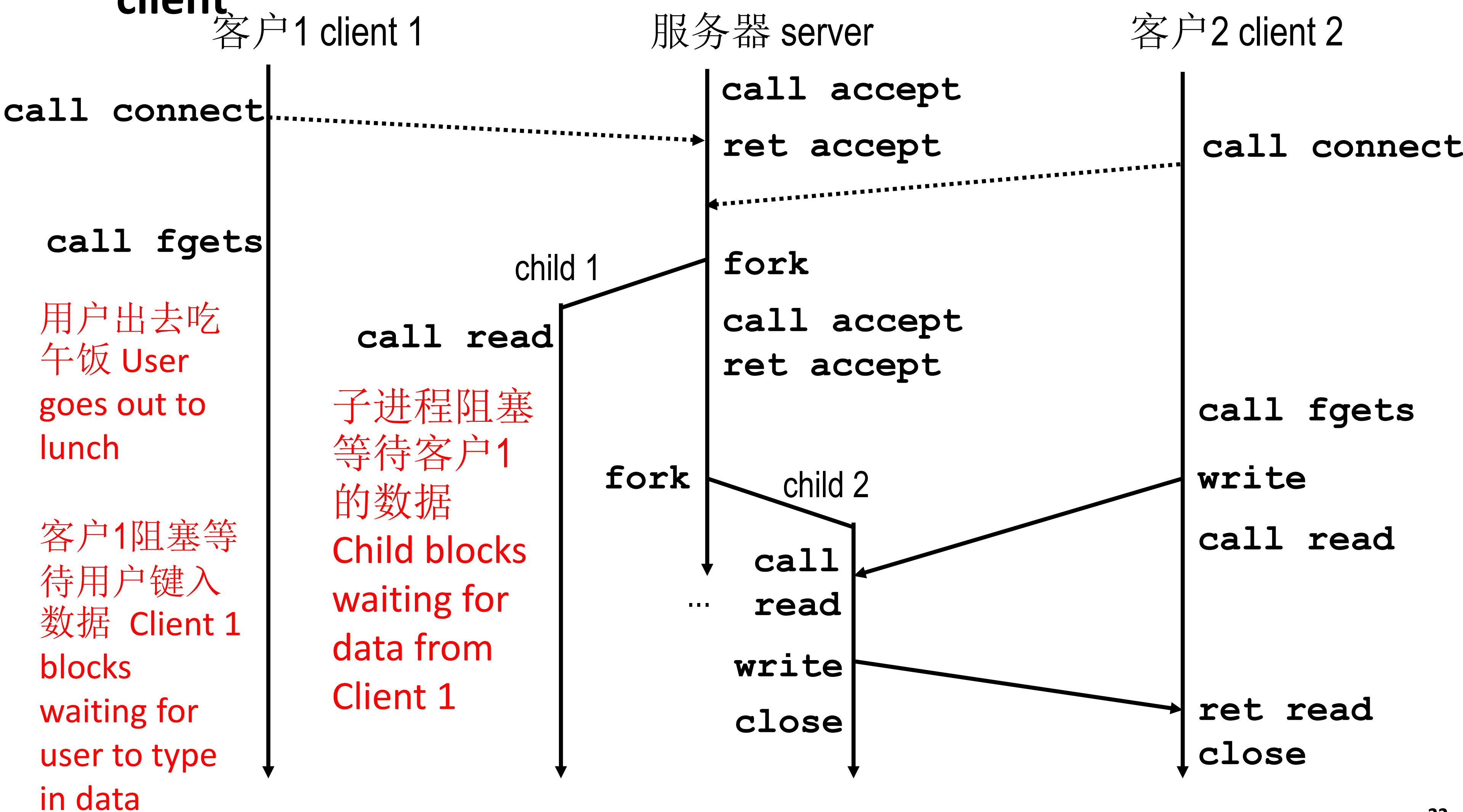
方法#1：基于进程的服务器
Approach #1: Process-based Servers

- 为每个客户生成单独的进程 Spawn separate process for each client



方法#1：基于进程的服务器
Approach #1: Process-based Servers

- 为每个客户生成单独的进程 Spawn separate process for each client



迭代式回声服务器 Iterative Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;


    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
     }
     exit(0);
}
```

- 接受一个连接请求 Accept a connection request
- 处理回声请求直到客户终止 Handle echo requests until client terminates

echoserverp.c

制作并发回声服务器
Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;


    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);


            echo(connfd);    /* Child services client */
            Close(connfd);  /* child closes connection with client */
            exit(0);



    }
}
```

echoserverp.c

制作并发回声服务器
Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;


    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {

            echo(connfd);     /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */
        }

    }
}
```

echoserverp.c

制作并发回声服务器
Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;


    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {

            echo(connfd);     /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

为何？ Why?

制作并发回声服务器
Making a Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;


    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

基于进程的并发回声服务器
Process-Based Concurrent Echo Server

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c

基于进程的并发回声服务器（续） Process-Based Concurrent Echo Server (cont)

```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

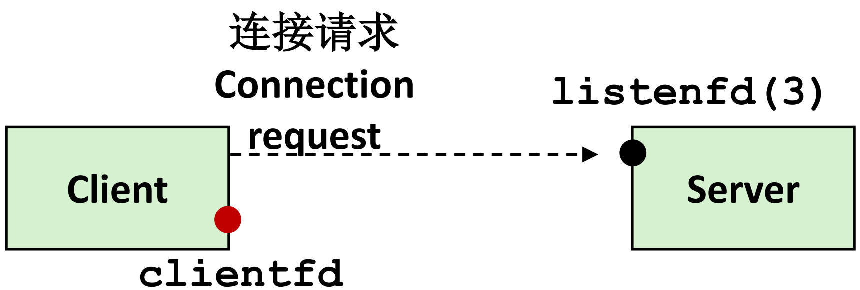
echoserverp.c

- 回收所有的僵尸子进程 Reap all zombie children

并发服务器：accept揭秘
Concurrent Server: `accept` Illustrated

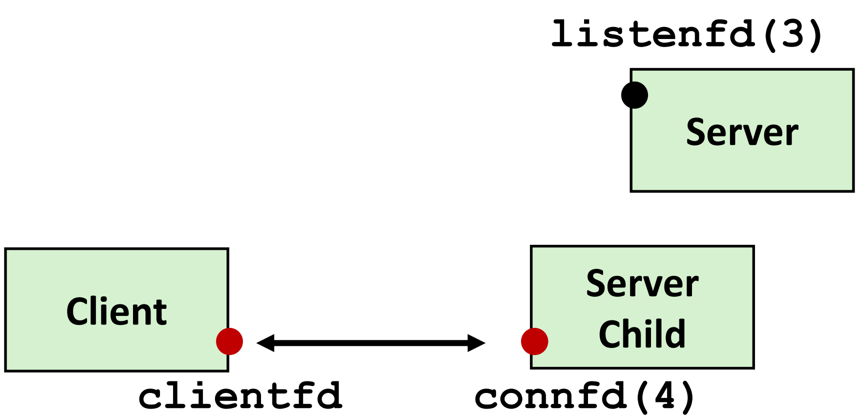
1.服务器阻塞在accept，等待侦听描述符listenfd上的连接请求

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`



2.客户端通过调用connect发出连接请求

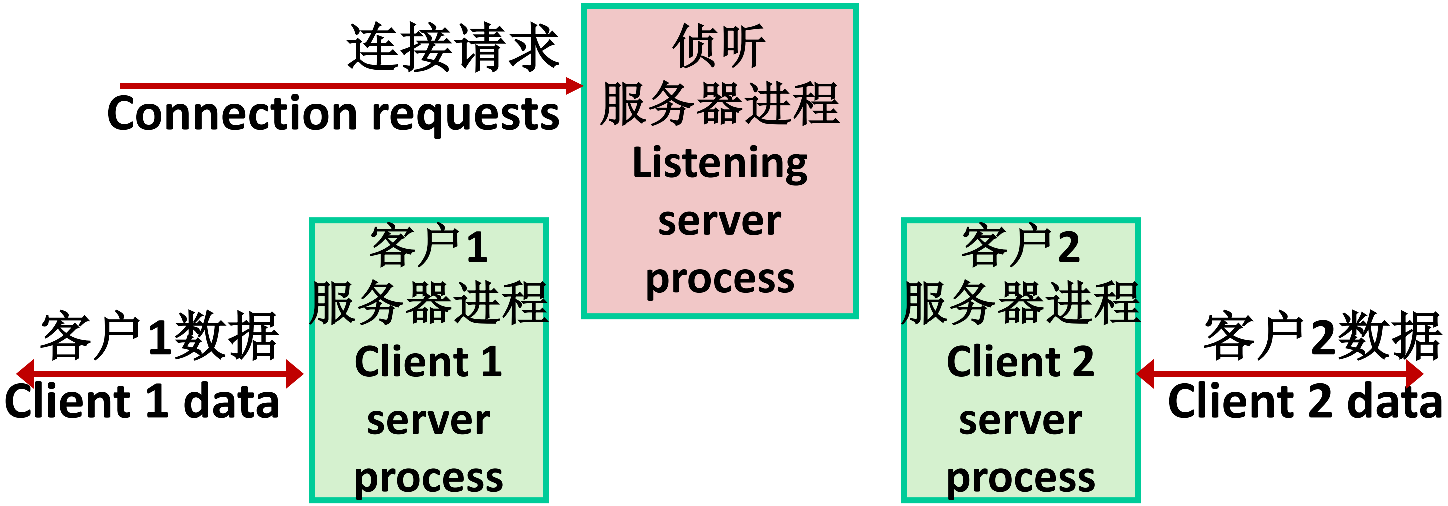
2. Client makes connection request by calling `connect`



3.服务器从accept返回connfd。创建子进程以处理客户端。现在已在clientfd和connfd之间建立连接

3. Server returns `connfd` from `accept`. Forks child to handle client. Connection is now established between `clientfd` and `connfd`

基于进程的服务器执行模型 Process-based Server Execution Model



- 每个客户端由独立的子进程处理 Each client handled by independent child process
- 它们之间没有共享状态 No shared state between them
- 父子进程都有listenfd和connfd的副本 Both parent & child have copies of listenfd and connfd
  - 父进程必须关闭connfd Parent must close **`connfd`**
  - 子进程应关闭listenfd Child should close **`listenfd`**

基于进程的服务器的问题
Issues with Process-based Servers

- 侦听服务器进程必须回收僵尸子进程 **Listening server process must reap zombie children**
  - 以避免致命的内存泄漏 to avoid fatal memory leak

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_stor          entadd

    listenfd = Open_lis       d(     1]);
    while (1) {
        clientlen = siz      struc      ade      orage);
        connfd = Accept      tenfd,         ientaddr, &clientlen);
        if (Fork() == 0)
            echo(connfd);          Chi      ces client */
            Close(connfd);               oses connection with clien
            exit(0);          /*       exits */
        }
```

基于进程的服务器的问题
Issues with Process-based Servers

- 父进程必须关闭其connfd副本 **Parent process must `close` its copy of `connfd`**
  - 内核保持每个套接字/打开文件的引用计数 Kernel keeps reference count for each socket/open file
  - 创建进程后，connfd引用计数为2 After fork, **`refcnt(connfd)=2`**
  - 在connfd引用计数为0之前，连接不会关闭 Connection will not be closed until **`refcnt(connfd) = 0`**

```
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_stor        entadd

    listenfd = Open_lis     d(     1]);
    while (1) {
        clientlen = siz    struc      ad      torage);
        connfd = Accept    tenfd,       ientaddr, &clientlen);
        if (Fork() == 0)
            echo(connfd);        Chi       ices client */
            Close(connfd);              oses connection with clien
            exit(0);             /        exits */
        }
```

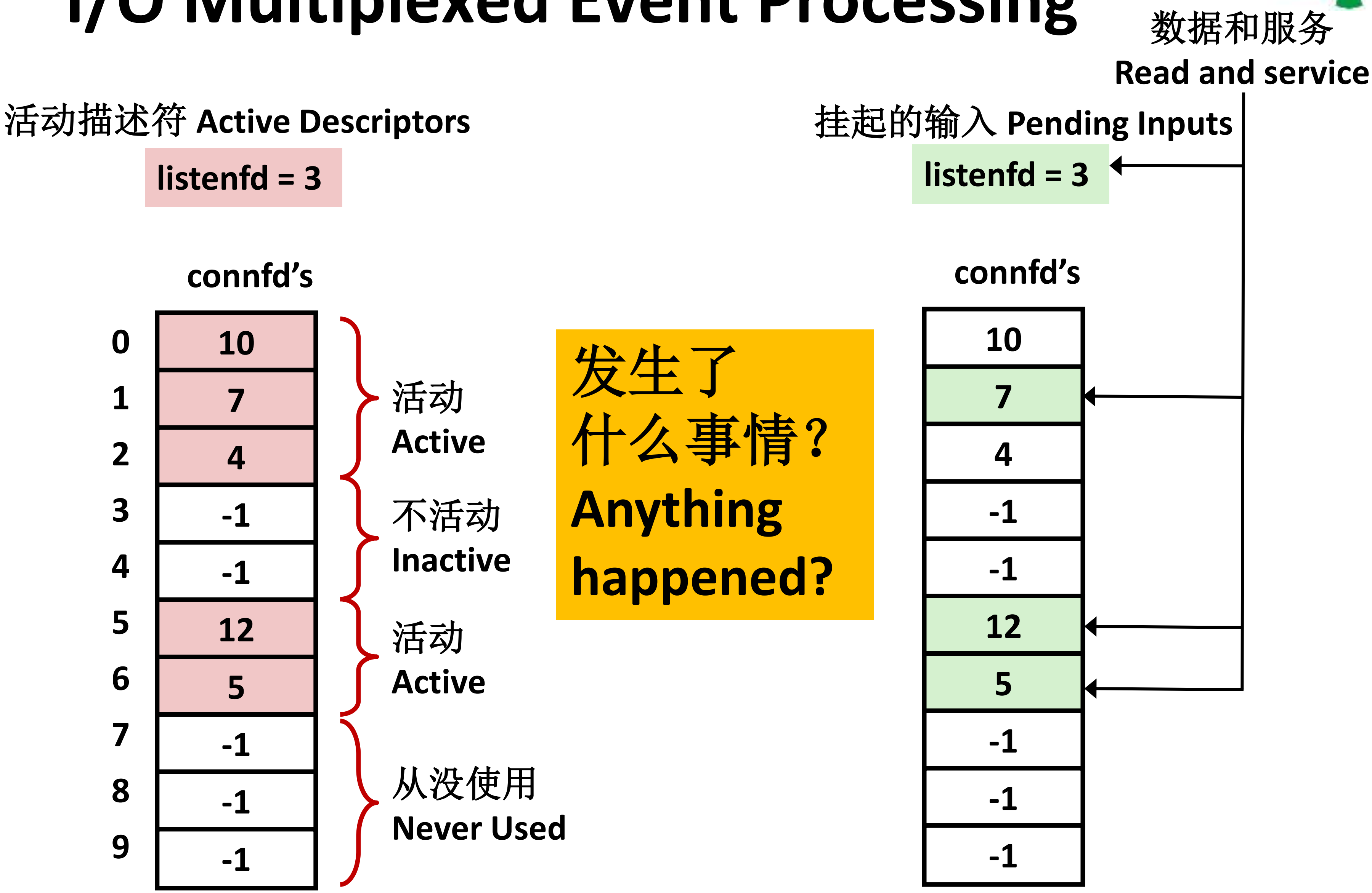
基于进程的服务器优点和缺点
Pros and Cons of Process-based Servers

- **+ 并发处理多个连接 Handle multiple connections concurrently**
- **+ 清晰的共享模型 Clean sharing model**
  - 描述符（否）descriptors (no)
  - 文件表（是）file tables (yes)
  - 全局变量（否）global variables (no)
- **+ 简单直接 Simple and straightforward**
- **– 额外的进程控制开销 Additional overhead for process control**
- **– 进程之间共享数据并不简单 Nontrivial to share data between processes**
  - (前面举的例子太过简单并不能说明问题 This example too simple to demonstrate)

方法#2：基于事件的服务器
Approach #2: Event-based Servers

- 服务器维护活动连接集合 **Server maintains set of active connections**
  - connfd数组 Array of connfd's
- 重复： **Repeat:**
  - 确定哪些描述符（connfd或listenfd）具有挂起的输入 Determine which descriptors (**`connfd`**'s or **`listenfd`**) have pending inputs
    - 例如：使用**select**函数 e.g., using **`select`** function
    - 挂起输入的到达是一个事件 arrival of pending input is an *event*
  - 如果listenfd有输入，则**接受**连接 If listenfd has input, then **`accept`** connection
    - 并将新的connfd添加到数组 and add new connfd to array
  - 使用挂起的输入服务所有连接 Service all connfd's with pending inputs
- 详细信息参见教材中基于选择的服务器 **Details for select-based server in book**

I/O多路复用事件处理
I/O Multiplexed Event Processing



基于事件的服务器优点和缺点
Pros and Cons of Event-based Servers

- **+ 一个逻辑控制流和地址空间 One logical control flow and address space.**
- **+ 可以用调试器进行单步跟踪 Can single-step with a debugger.**
- **+ 没有进程或线程控制开销 No process or thread control overhead.**
  - 成为高性能Web服务器和搜索引擎的设计选择，例如Node.js、nginx、Tornado Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado

- **– 比基于进程或线程的设计代码要明显复杂很多 Significantly more complex to code than process- or thread-based designs.**
- **– 很难提供细粒度的并发 Hard to provide fine-grained concurrency**
  - 例如如何处理部分HTTP请求首部 E.g., how to deal with partial HTTP request headers
- **– 不能利用多核的优势 Cannot take advantage of multi-core**
  - 单一的控制线程 Single thread of control

方法#3：基于线程的服务器
Approach #3: Thread-based Servers

- 与方法#1（基于进程）非常相似 **Very similar to approach #1 (process-based)**
  - …但是使用线程代替进程 …but using threads instead of processes

传统进程视图 Traditional View of a Process

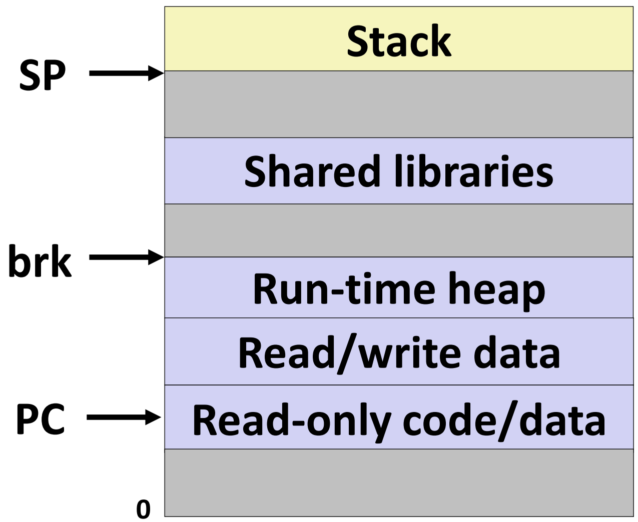
- 进程=进程上下文+代码、数据和栈 Process = process context + code, data, and stack

进程上下文
Process context

Program context:
Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

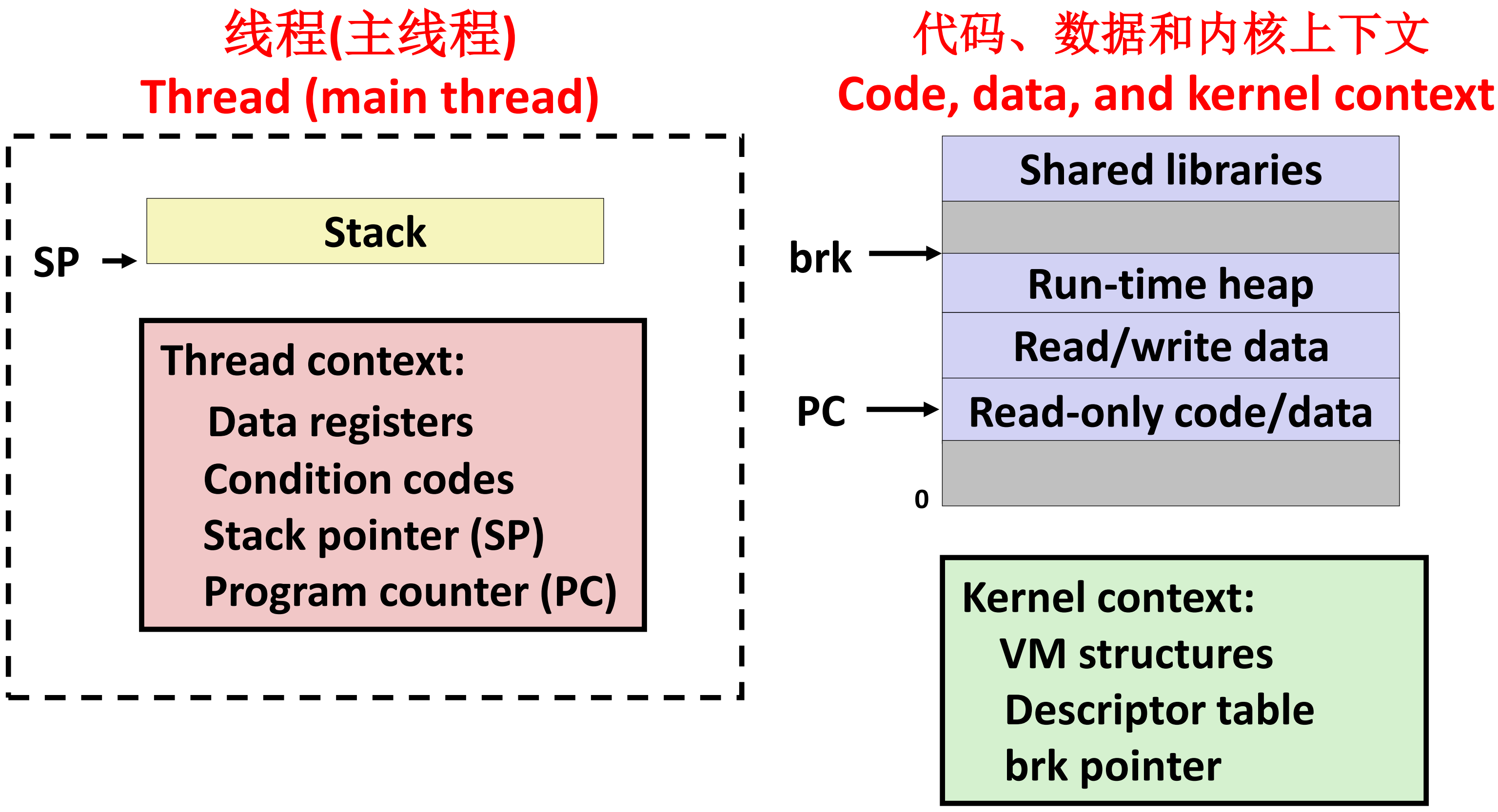
Kernel context:
VM structures
Descriptor table
brk pointer

代码、数据和栈 Code, data, and stack



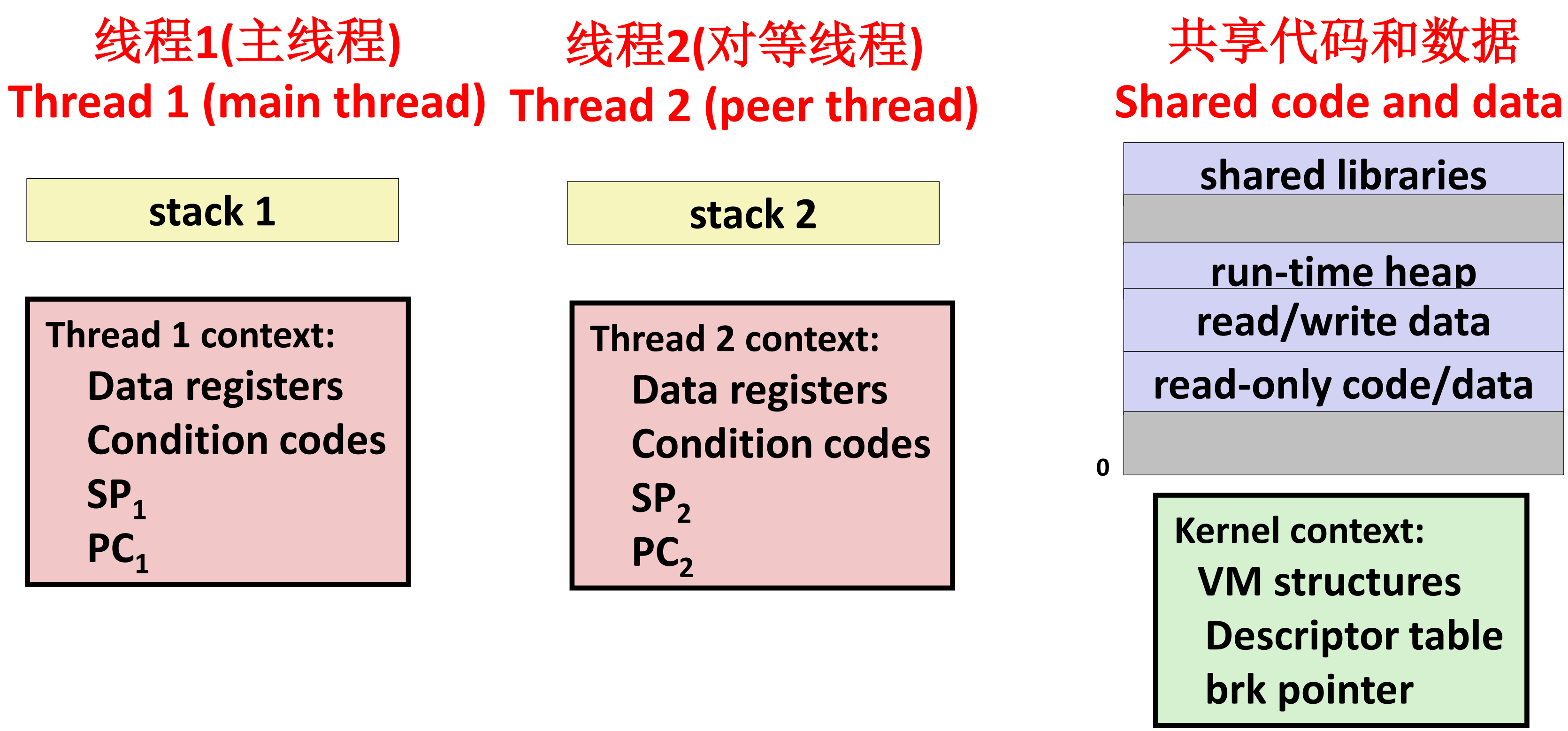
另一种进程视图 Alternate View of a Process

- 进程=线程+代码、数据和内核上下文 Process = thread + code, data, and kernel context



一个进程有多个线程-多线程进程
A Process With Multiple Threads

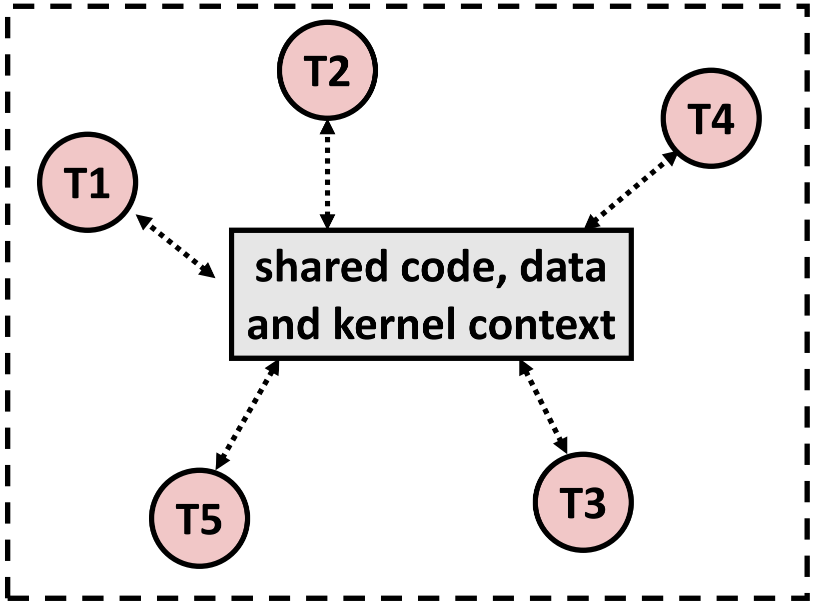
- 多个线程可以与一个进程关联 **Multiple threads can be associated with a process**
  - 每个线程都有自己的逻辑控制流 Each thread has its own logical control flow
  - 每个线程共享相同的代码、数据和内核上下文 Each thread shares the same code, data, and kernel context
  - 每个线程都有自己的局部变量栈 Each thread has its own stack for local variables
    - 但不受其他线程的保护 but not protected from other threads
  - 每个线程都有自己的线程id（TID） Each thread has its own thread id (TID)



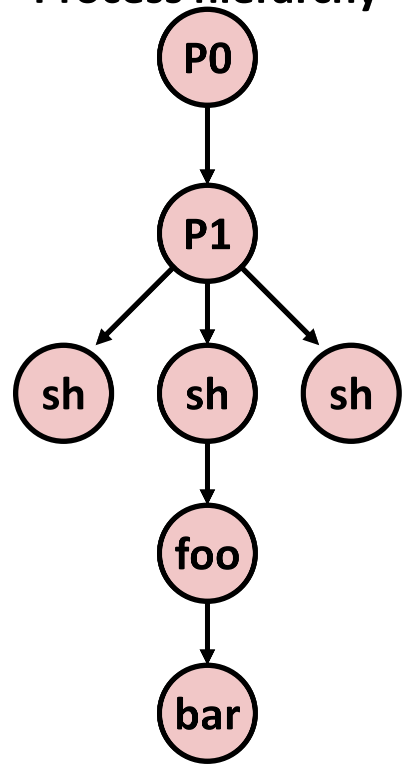
线程的逻辑视图 Logical View of Threads

- **与进程关联的线程形成对等线程池 Threads associated with process form a pool of peers**
  - 与进程形成层次树不同 Unlike processes which form a tree hierarchy

与进程foo关联的线程
Threads associated with process foo

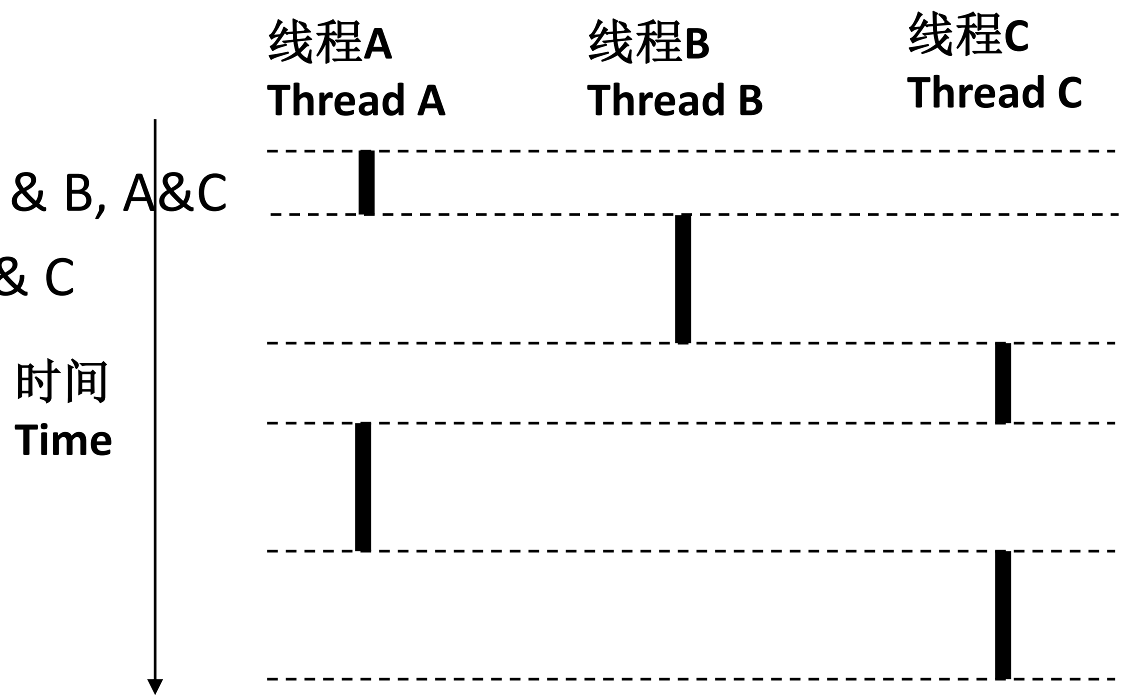


进程层次结构
Process hierarchy



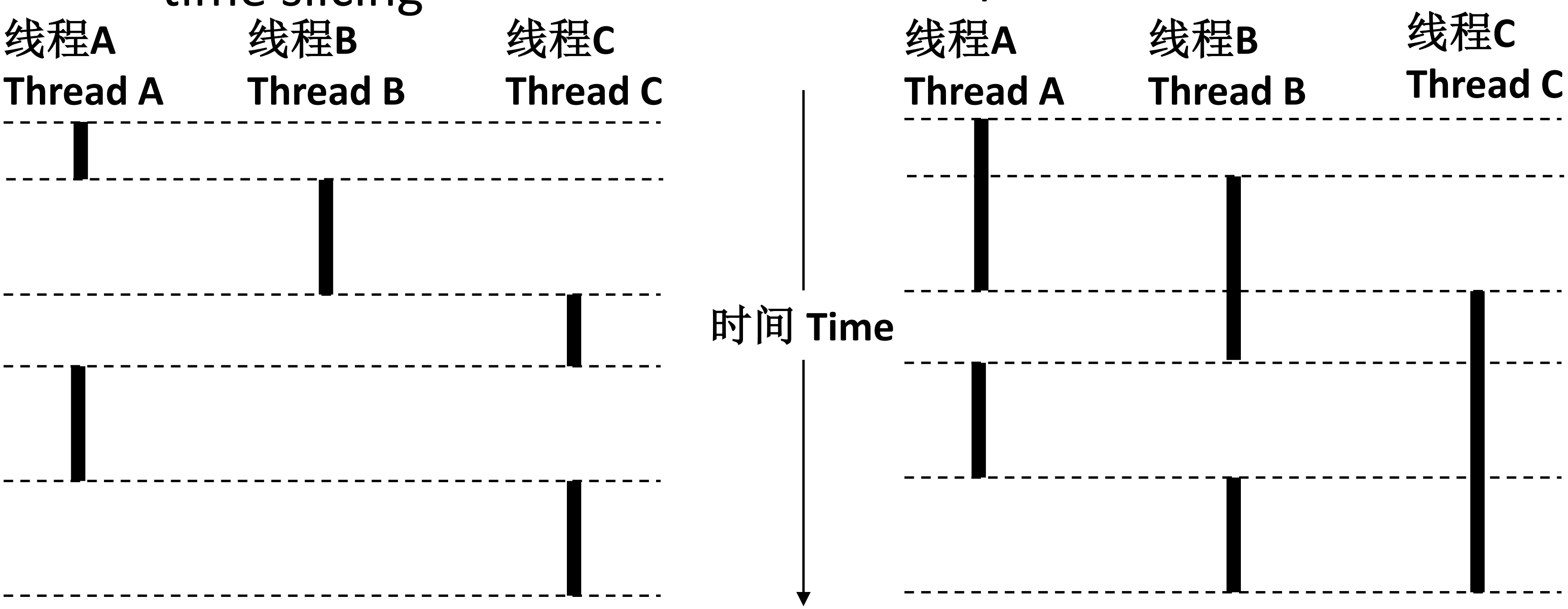
并发线程 Concurrent Threads

- 两个线程是并发的，如果它们的流程在时间上重叠 **Two threads are *concurrent* if their flows overlap in time**
- 否则，它们是顺序的 **Otherwise, they are sequential**
- 示例： **Examples:**
  - 并发 Concurrent: A & B, A&C
  - 顺序 Sequential: B & C



并发线程执行 Concurrent Thread Execution

- 单核处理器 **Single Core Processor**
  - 通过分时模拟并行
    Simulate parallelism by time slicing
- 多核处理器 **Multi-Core Processor**
  - 可以实现真正并行
    Can have true parallelism



2个核心上运行3个线程
Run 3 threads on 2 cores

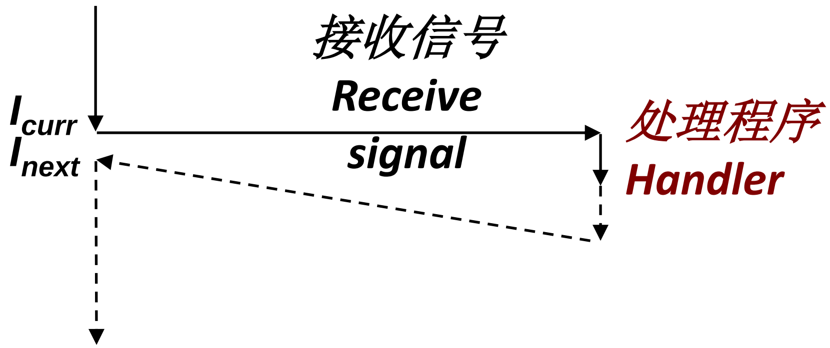
线程对比进程 Threads vs. Processes

- **线程和进程如何相似 How threads and processes are similar**
  - 每个都有自己的逻辑控制流 Each has its own logical control flow
  - 每个都可以与其他并发运行（可能在不同的核心上） Each can run concurrently with others (possibly on different cores)
  - 每个都要进行上下文切换 Each is context switched

线程对比进程 Threads vs. Processes

- **线程和进程的区别 How threads and processes are different**
  - 线程共享所有代码和数据（局部栈除外） Threads share all code and data (except local stacks)
    - 进程（通常）不会 Processes (typically) do not
  - 线程的开销略低于进程 Threads are somewhat less expensive than processes
    - 进程控制（创建和回收）的开销是线程控制的两倍 Process control (creating and reaping) twice as expensive as thread control
    - Linux上的数字： Linux numbers:
      - 约2万个时钟周期来创建和回收进程 ~20K cycles to create and reap a process
      - 约1万个时钟周期（或更少）来创建和回收线程 ~10K cycles (or less) to create and reap a thread

线程对信号 Threads vs. Signals



- 信号处理程序与普通程序共享状态 **Signal handler shares state with regular program**
  - 包括栈 Including stack
- 信号处理程序中断正常程序的执行 **Signal handler interrupts normal program execution**
  - 不预期的过程调用 Unexpected procedure call
  - 返回到正常执行流 Returns to regular execution stream
  - *不是*一个对等体 *Not* a peer
- 有限的同步形式 **Limited forms of synchronization**
  - 主程序可以阻塞/解阻塞信号 Main program can block / unblock signals
  - 主程序可以暂停信号 Main program can pause for signal

Posix线程（Pthread）接口
Posix Threads (Pthreads) Interface

- ***Pthreads*：标准接口，包含约60个函数，可以从C语言程序操作线程 *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs**
  - 创建和回收线程 Creating and reaping threads
    - **`pthread_create()`**
    - **`pthread_join()`**
  - 确定线程ID Determining your thread ID
    - **`pthread_self()`**
  - 终止线程 Terminating threads
    - **`pthread_cancel()`**
    - **`pthread_exit()`**
    - **`exit()`** [终止所有线程 terminates all threads]
    - **`return`** [终止当前线程 terminates current thread]
  - 对共享变量的访问进行同步 Synchronizing access to shared variables
    - **`pthread_mutex_init`**
    - **`pthread_mutex_[un]lock`**

Pthread的“hello, world”程序
The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    return 0;
}
```

hello.c

线程ID Thread ID

线程属性 Thread attributes (通常为空 usually NULL)

线程例程 Thread routine

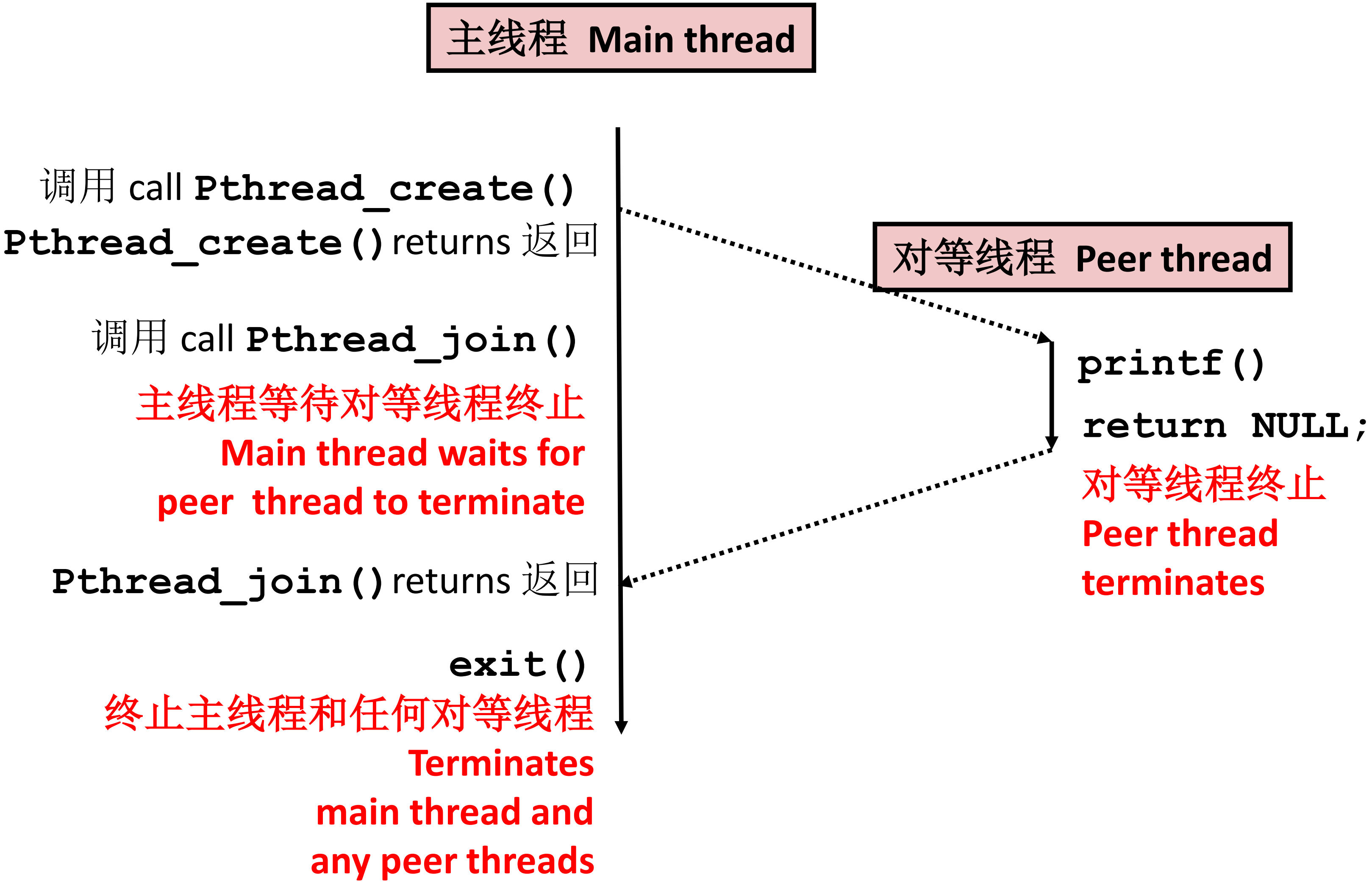
*线程参数 Thread argu (void *p)*

*返回值 Return value (void **p)*

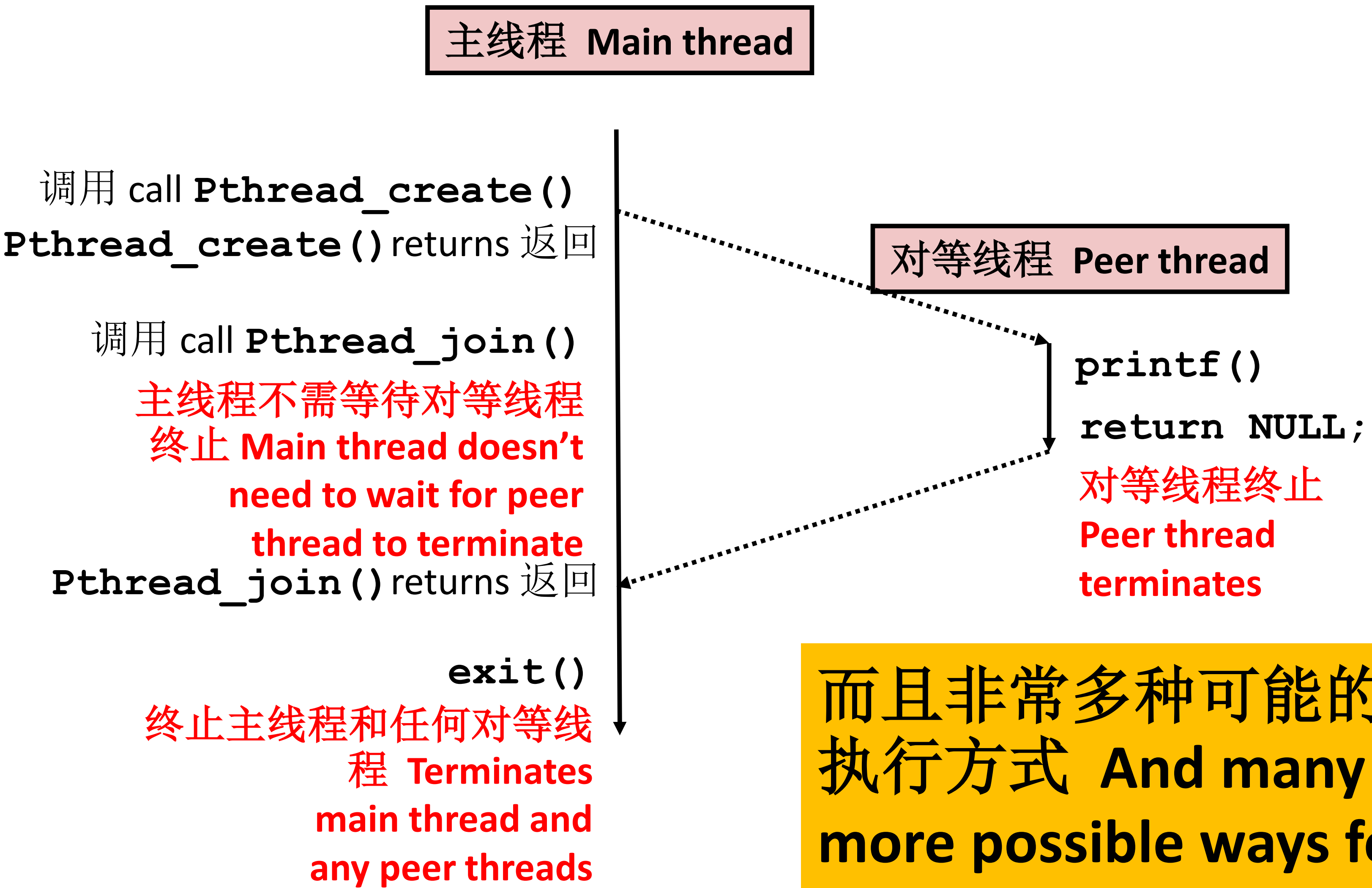
```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

线程化的“hello, world”执行
Execution of Threaded “hello, world”



或者… Or, …



而且非常多种可能的代码执行方式 And many many more possible ways for this code to execute.

基于线程的并发回声服务器
Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
    return 0;
}
```

echoservert.c

- 为每个客户生成新线程 Spawn new thread for each client
- 把连接文件描述符的拷贝传递给新线程 Pass it copy of connection file descriptor
- 注意使用Malloc()！ [但是没有释放Free()]   Note use of **Malloc()**! [but not **Free()**]

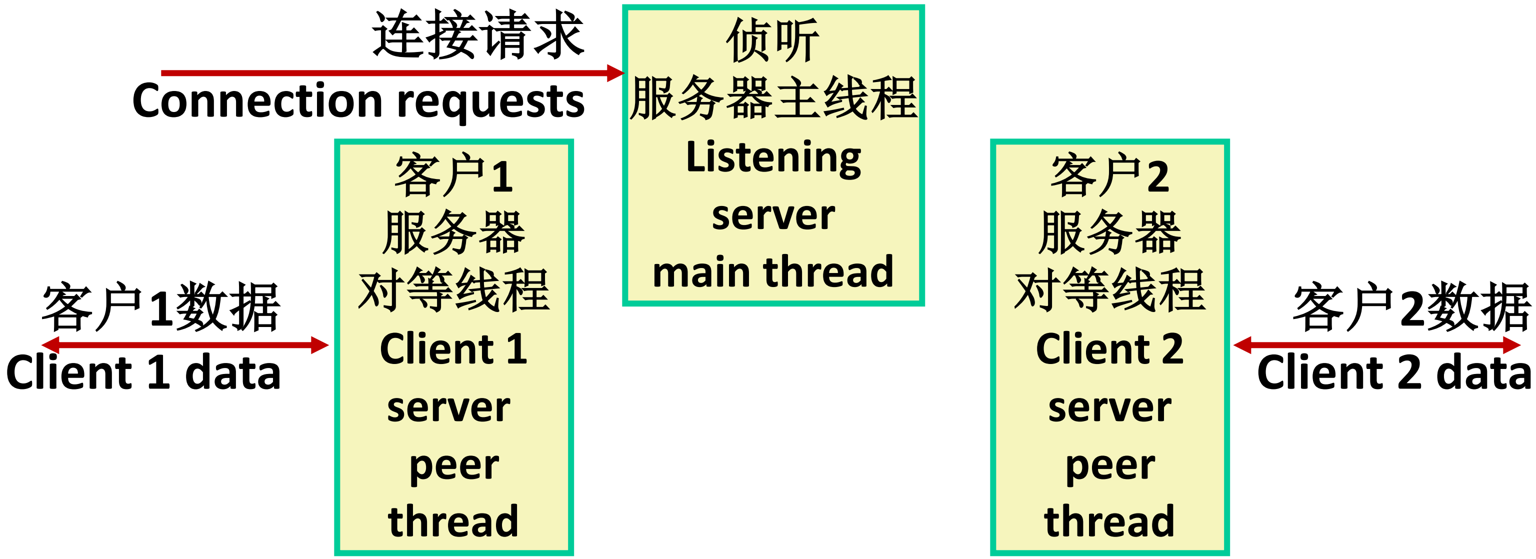
基于线程的并发服务器（续）
Thread-Based Concurrent Server (cont)

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

echoservert.c

- 运行线程在“分离的”模式 Run thread in “detached” mode.
  - 与其它线程独立运行 Runs independently of other threads
  - 当终止时自动回收（由内核） Reaped automatically (by kernel) when it terminates
- 释放分配给保存connfd的存储空间 Free storage allocated to hold **connfd**
- 关闭connfd（重要！） Close **connfd** (important!)

基于线程的服务器执行模式
Thread-based Server Execution Model



- 每个客户由单个对等线程处理 Each client handled by individual peer thread
- 线程共享除TID之外的所有进程状态 Threads share all process state except TID
- 每个线程都有一个单独的局部变量栈 Each thread has a separate stack for local variables

基于线程的服务器的问题
Issues With Thread-Based Servers

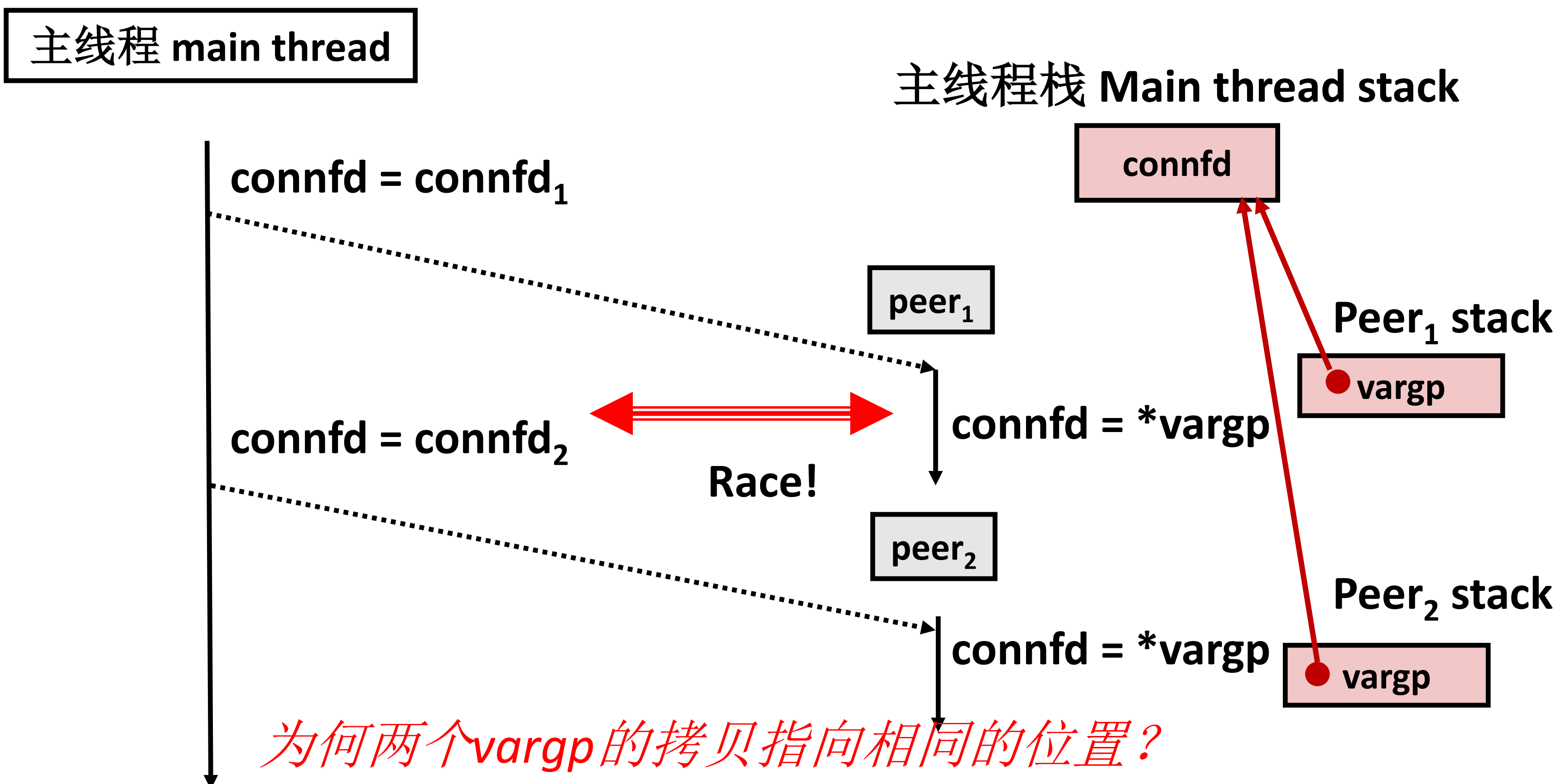
- 必须运行“分离”以避免内存泄漏 **Must run “detached” to avoid memory leak**
  - 在任何时间点，线程都是*可结合的*或*分离的* At any point in time, a thread is either *joinable* or *detached*
  - *可结合的*线程可以被其他线程回收和杀死 *Joinable* thread can be reaped and killed by other threads
    - 必须回收（使用pthread_join）以释放内存资源 must be reaped (with **`pthread_join`**) to free memory resources
  - *分离的*线程不能被其他线程回收或杀死 *Detached* thread cannot be reaped or killed by other threads
    - 终止时自动回收资源 resources are automatically reaped on termination
  - 默认状态为可结合的 Default state is joinable
    - 使用pthread_detach(pthread_self())进行分离 use **`pthread_detach(pthread_self())`** to make detached

基于线程的服务器的问题
Issues With Thread-Based Servers

- 必须小心避免意外共享 **Must be careful to avoid unintended sharing**
  - 例如，将指针传递到主线程的栈 For example, passing pointer to main thread's stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
- 线程调用的所有函数都必须是*线程安全的* **All functions called by a thread must be *thread-safe***
  - （下次课）/ (next lecture)

意外共享的潜在形式
Potential Form of Unintended Sharing

```
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, &connfd);
}
```

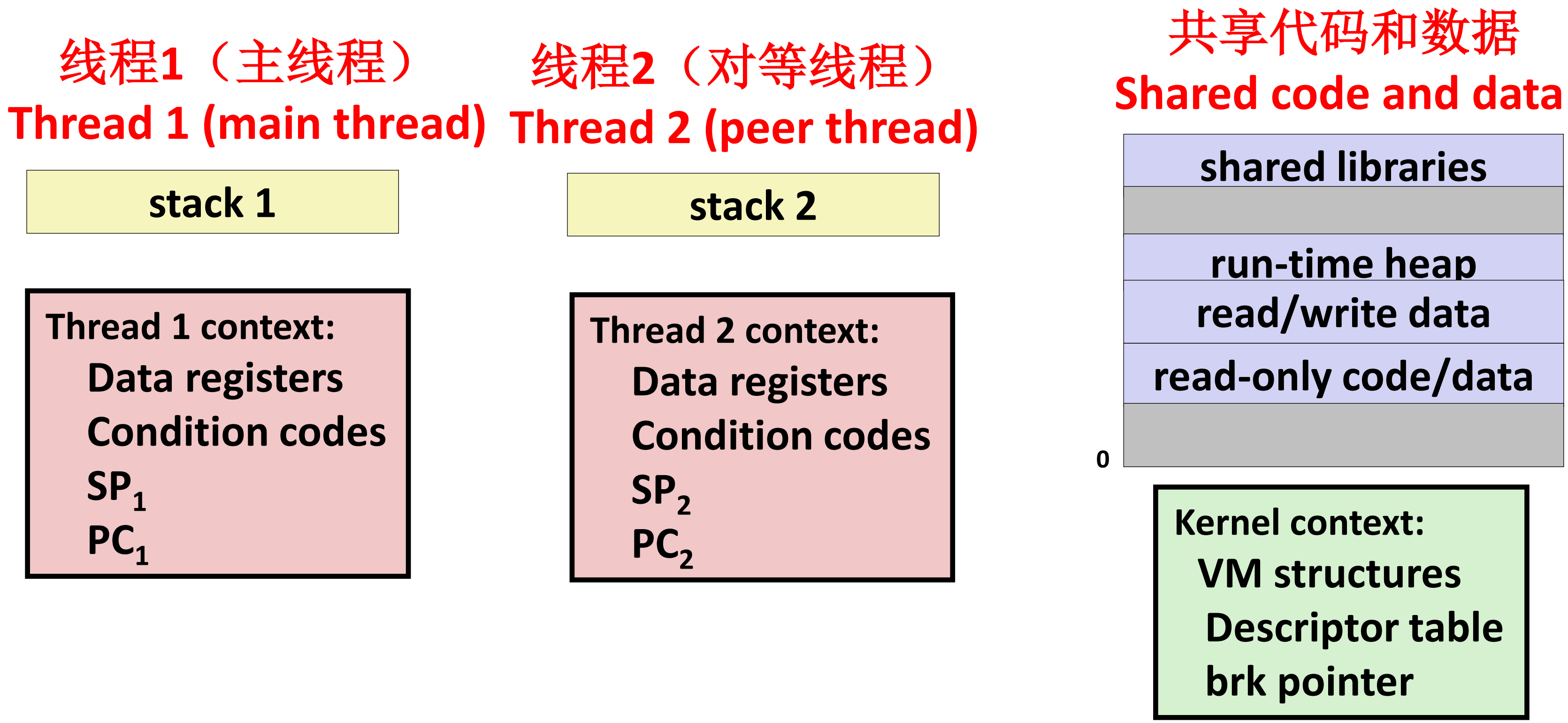


为何两个vargp的拷贝指向相同的位置？

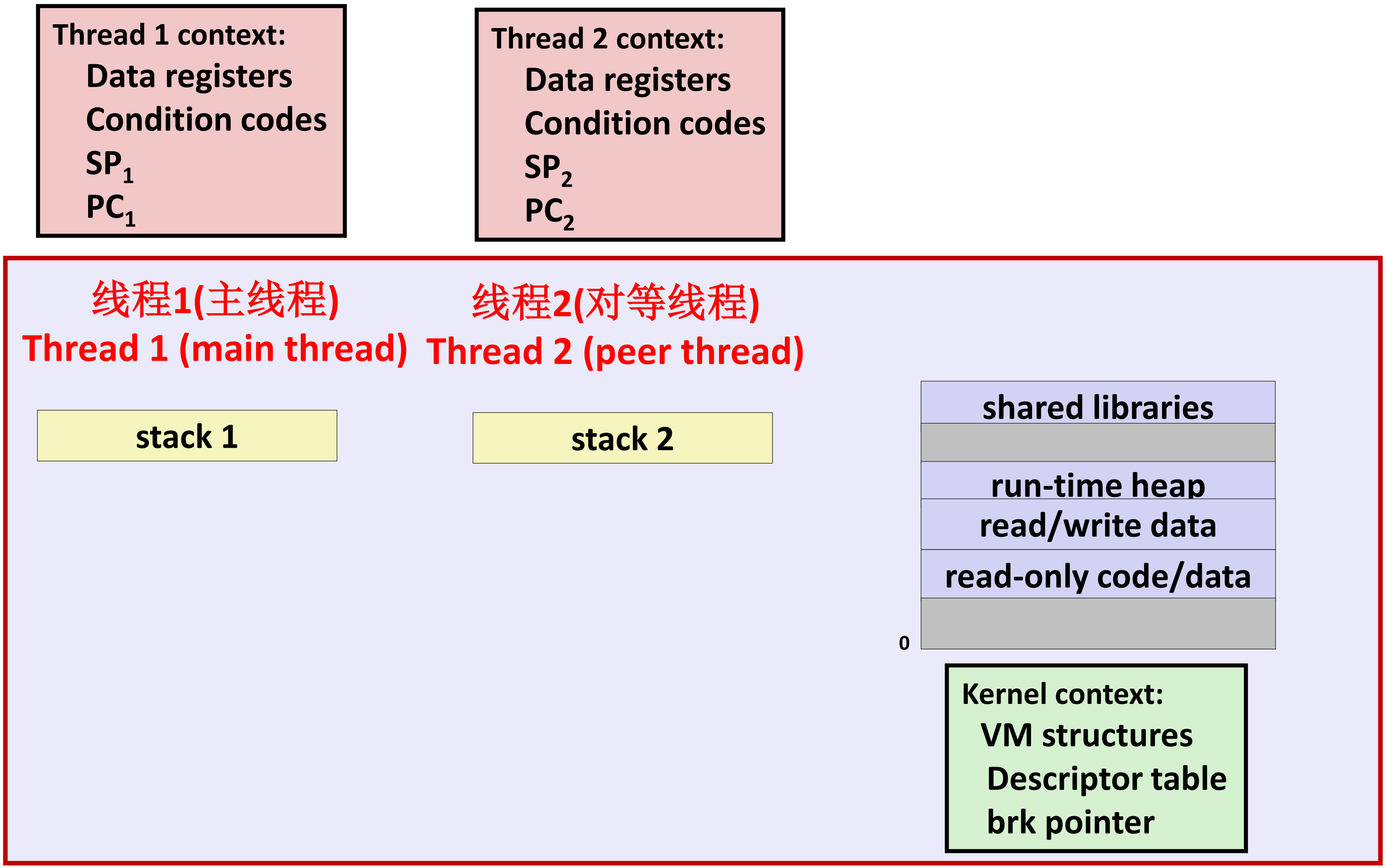
Why would both copies of vargp point to same location?

一个进程有多个线程
A Process With Multiple Threads

- 多个线程可以与一个进程关联 **Multiple threads can be associated with a process**
  - 每个线程都有自己的逻辑控制流 Each thread has its own logical control flow
  - 每个线程共享相同的代码、数据和内核上下文 Each thread shares the same code, data, and kernel context
  - 每个线程都有自己的局部变量栈 Each thread has its own stack for local variables
    - 但不受其他线程的保护 but not protected from other threads
  - 每个线程都有自己的线程id（TID） Each thread has its own thread id (TID)



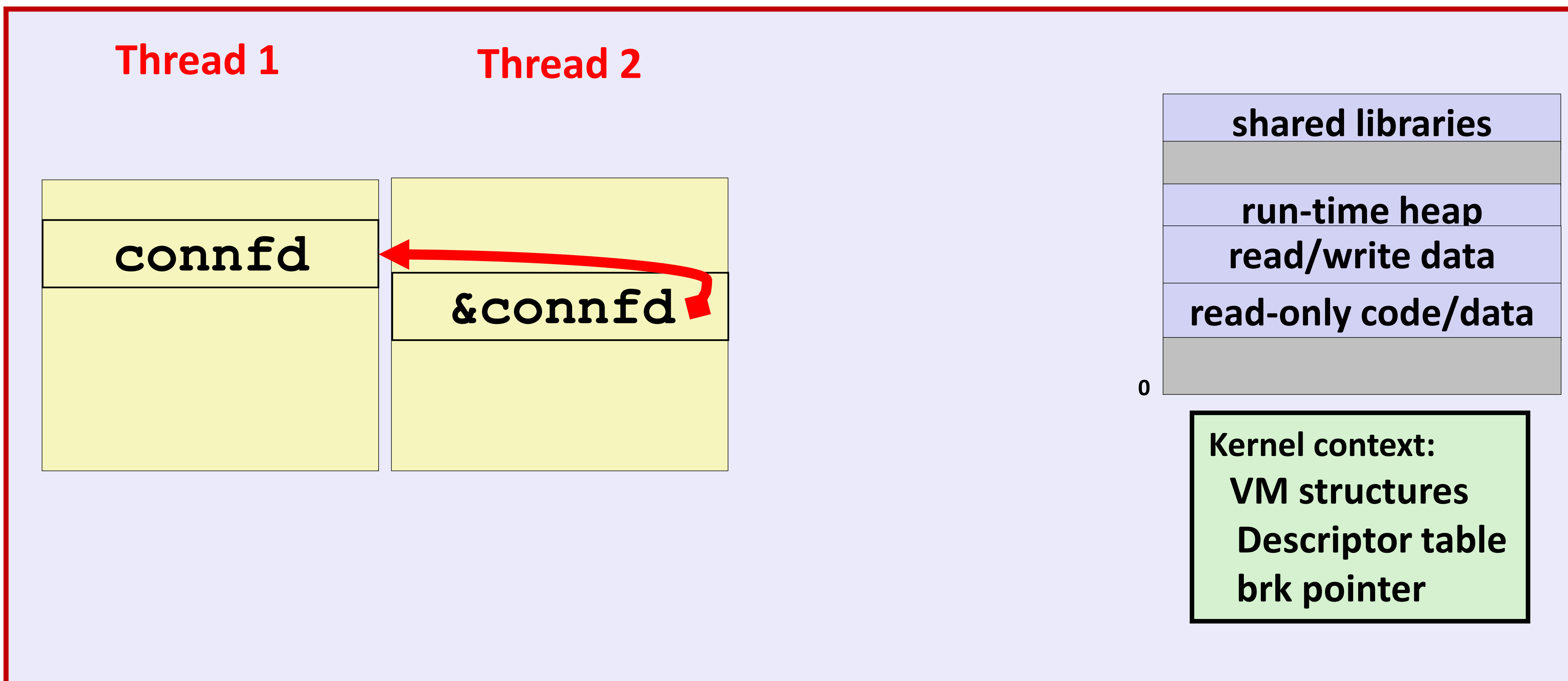
但是所有的内存都是共享的
But ALL memory is shared



```
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, &connfd);
}
```

Thread 1 context:
- Data registers
- Condition codes
- SP_1
- PC_1

Thread 2 context:
- Data registers
- Condition codes
- SP_2
- PC_2

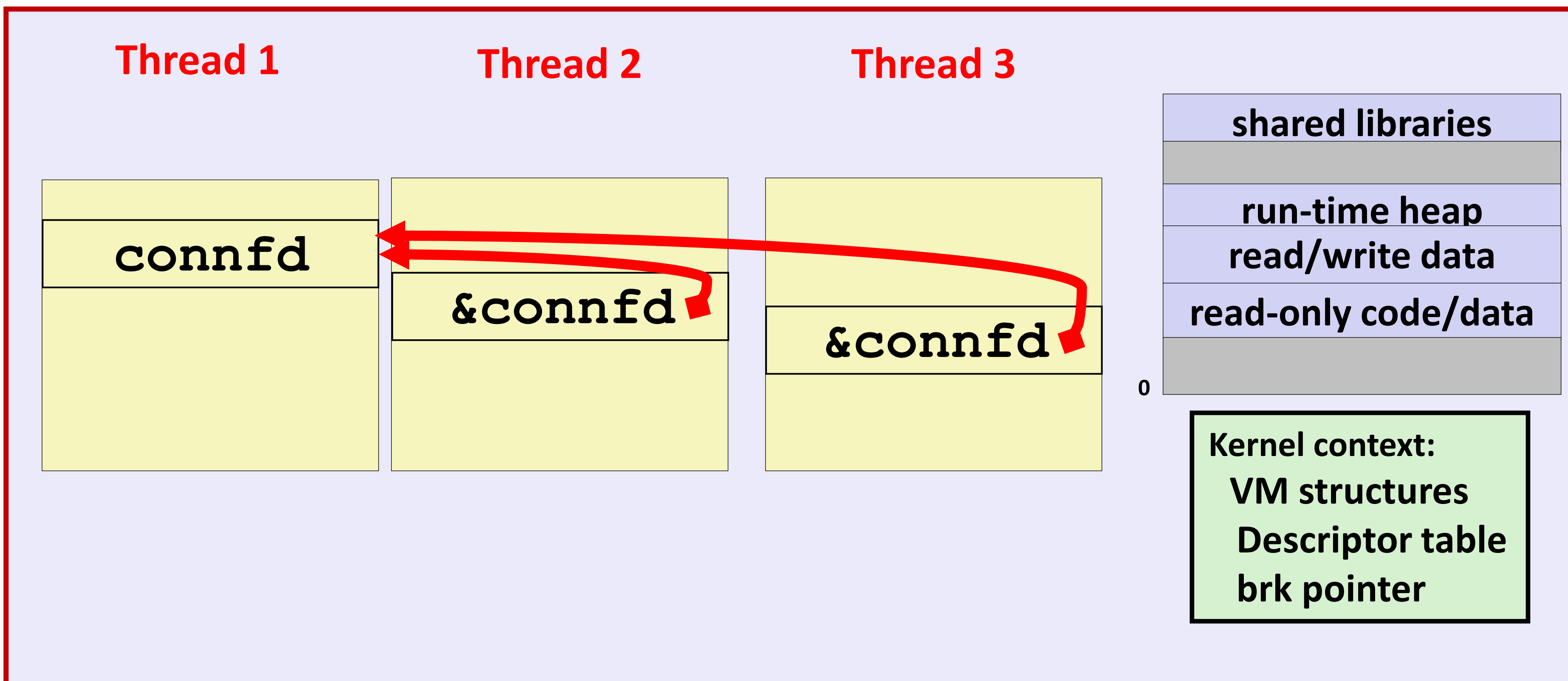


```
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, thread, &connfd);
}
```

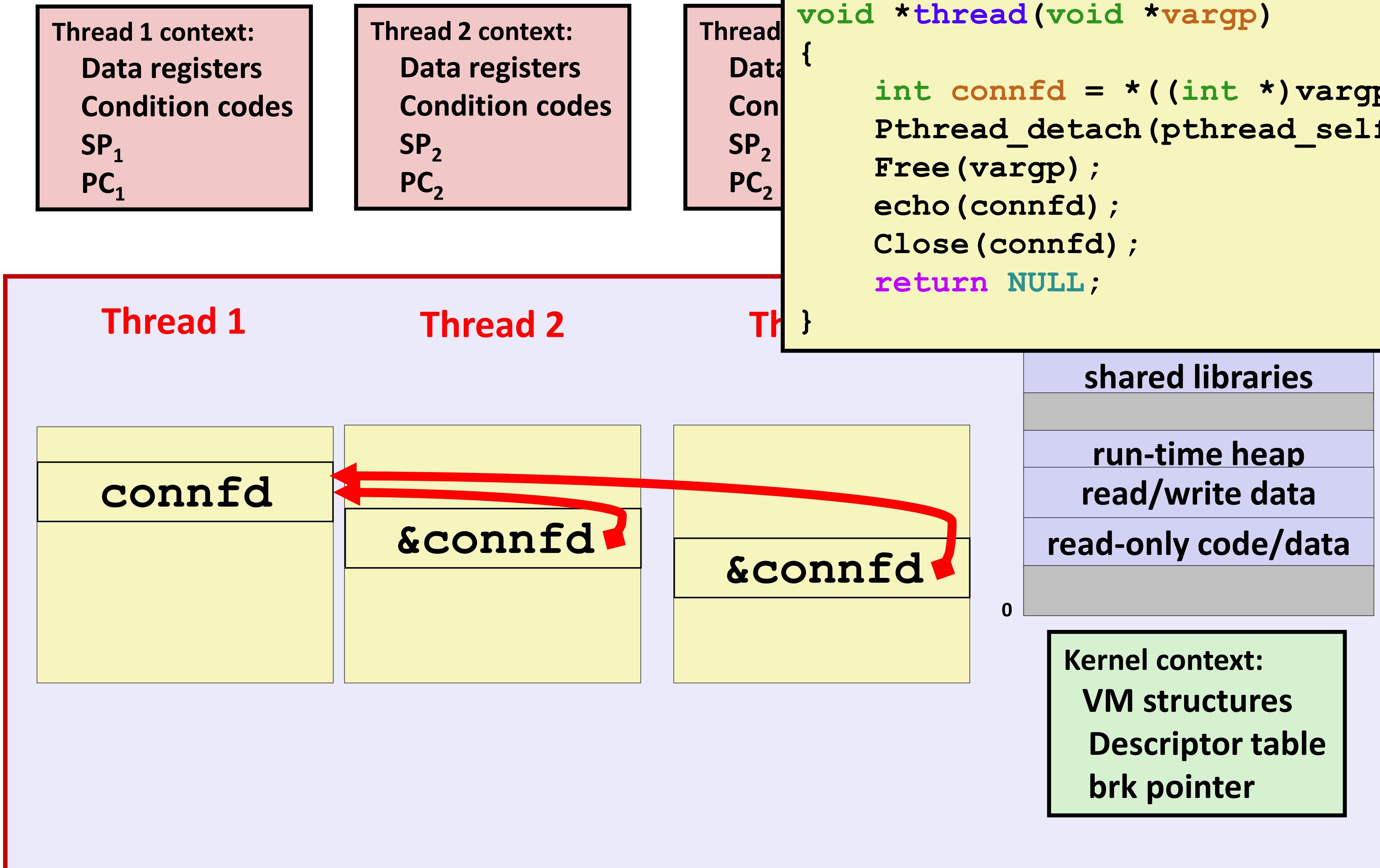
Thread 1 context:
- Data registers
- Condition codes
- SP_1
- PC_1

Thread 2 context:
- Data registers
- Condition codes
- SP_2
- PC_2

Thread 3 context:
- Data registers
- Condition codes
- SP_2
- PC_2



```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp)
    Pthread_detach(pthread_self(
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```



这样会发生竞争吗？
Could this race occur?

主线程 **Main**

```
int i;
for (i = 0; i < 100; i++) {
  Pthread_create(&tid, NULL,
                 thread, &i);
}
```

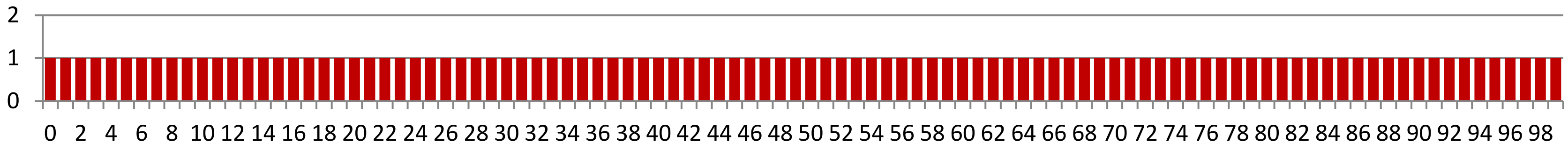
对等线程 **Thread**

```
void *thread(void *vargp)
{
  int i = *((int *)vargp);
  Pthread_detach(pthread_self());
  save_value(i);
  return NULL;
}
```

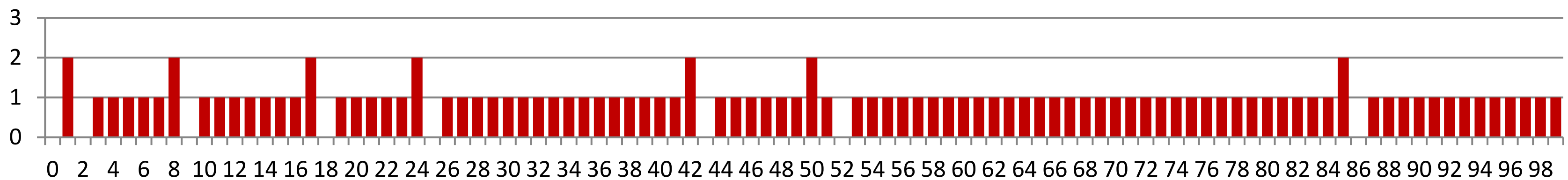
- 竞争测试 **Race Test**
  - 如果不存在竞争，那么每个线程得到不同的i值 If no race, then each thread would get different value of **i**
  - 保存值的集合将由每个0到99的拷贝组成 Set of saved values would consist of one copy each of 0 through 99

实验结果 Experimental Results

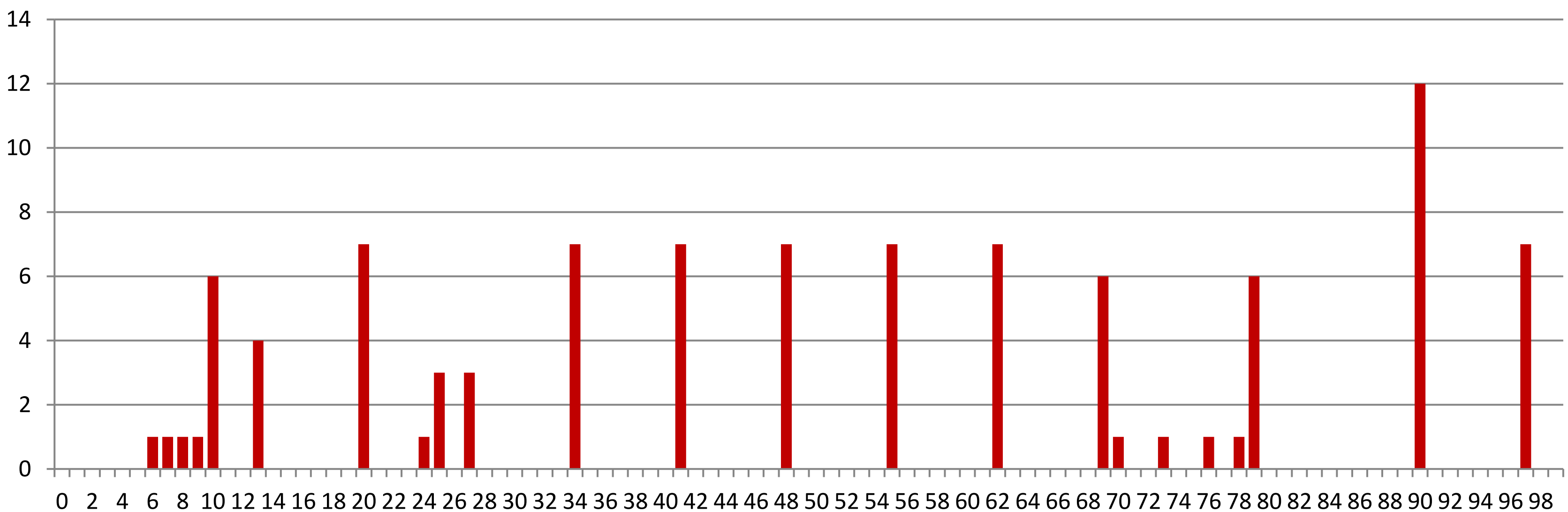
没有竞争 No Race



单核笔记本 Single core laptop



多核服务器 Multicore server



- **竞争真的会发生！ The race can really happen!**

正确传递线程参数
Correct passing of thread arguments

```
/* Main routine */
        int *connfdp;
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept( . . . );
        Pthread_create(&tid, NULL, thread, connfdp);
```

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
        . . .
    Free(vargp);
        . . .
    return NULL;
}
```

- 生产者-消费者模型 Producer-Consumer Model
  - 在main函数分配空间 Allocate in main
  - 在线程例程中释放 Free in thread routine

基于线程的设计优点和缺点 Pros and Cons of Thread-Based Designs

- **+ 易于在线程之间共享数据结构 Easy to share data structures between threads**
  - 例如日志信息、文件缓存 e.g., logging information, file cache
- **+ 线程比进程更有效率 Threads are more efficient than processes**

基于线程的设计优点和缺点
Pros and Cons of Thread-Based Designs

- **– 无意中的共享可能会导致细微且难以再现的错误！ Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - 轻松共享数据是线程的最大优势和最大弱点 The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - 很难知道哪些数据是共享的，哪些是私有的 Hard to know which data shared & which private
  - 难以靠测试检测 Hard to detect by testing
    - 竞争结果不佳的概率很低 Probability of bad race outcome very low
    - 但非零！ But nonzero!
  - 未来课次讲授 Future lectures

小结：并发的方法
Summary: Approaches to Concurrency

- 基于进程 **Process-based**
  - 难以共享资源：易于避免意外共享 Hard to share resources: Easy to avoid unintended sharing
  - 添加/删除客户的开销高 High overhead in adding/removing clients
- 基于事件 **Event-based**
  - 乏味和低级 Tedious and low level
  - 对调度的全面控制 Total control over scheduling
  - 非常低的开销 Very low overhead
  - 无法创建细粒度的并发级别 Cannot create as fine grained a level of concurrency
  - 不能使用多核 Does not make use of multi-core
- 基于线程 **Thread-based**
  - 易于共享资源：可能太容易了 Easy to share resources: Perhaps too easy
  - 中等开销 Medium overhead
  - 对调度策略没有太多控制 Not much control over scheduling policies
  - 难以调试 Difficult to debug
    - 事件顺序不可重复 Event orderings not repeatable

第12章 并发编程
同步：基础 Synchronization: Basics

100076202： 计算机系统导论

任课教师：
宿红毅 张艳 黎有琦 颜珂

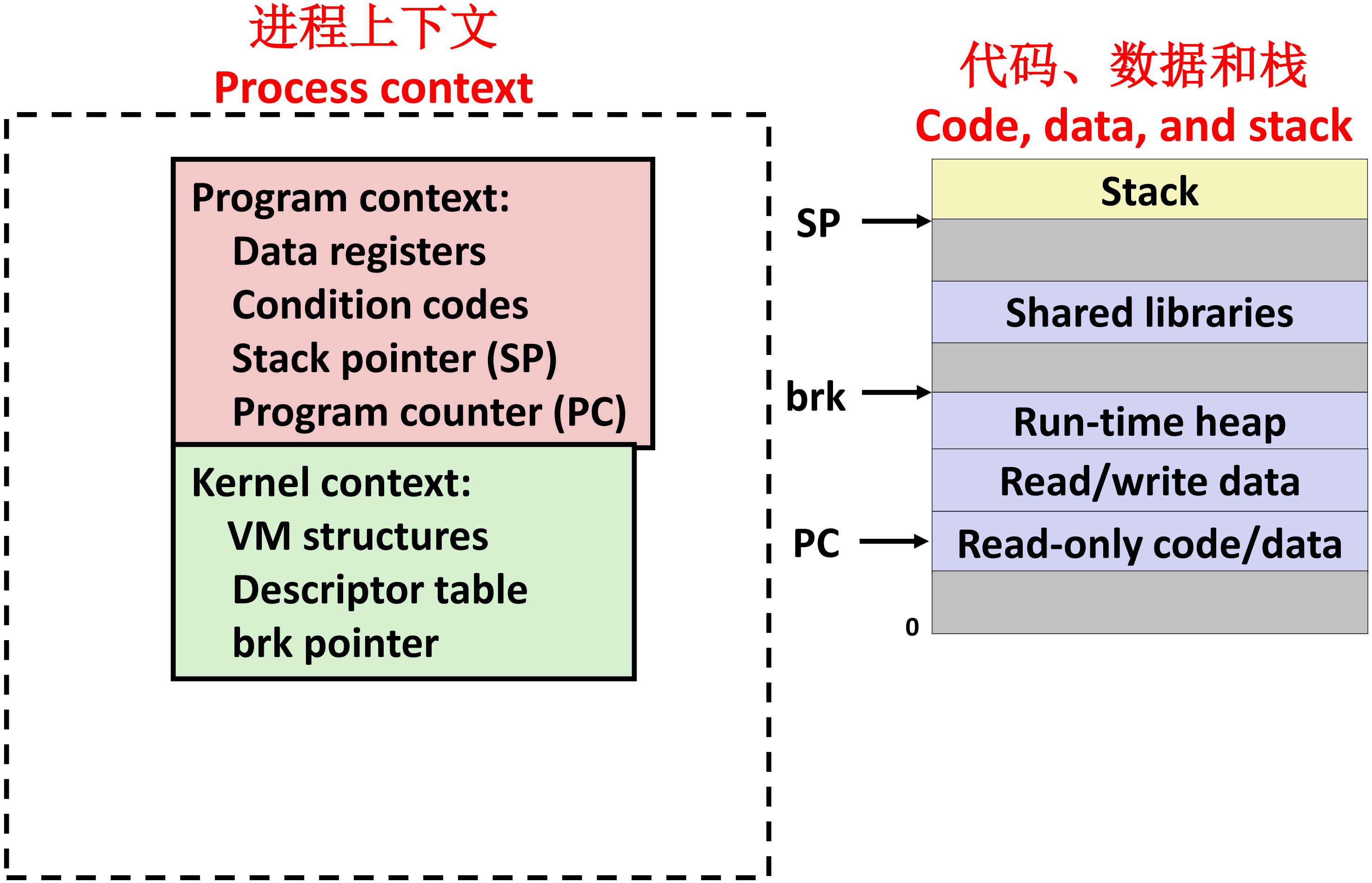
原作者：
Randal E. **Bryant and** David R. O’Hallaron

议题 Today

- **线程回顾 Threads review**
- 共享 Sharing
- 互斥 Mutual exclusion
- 信号量 Semaphores

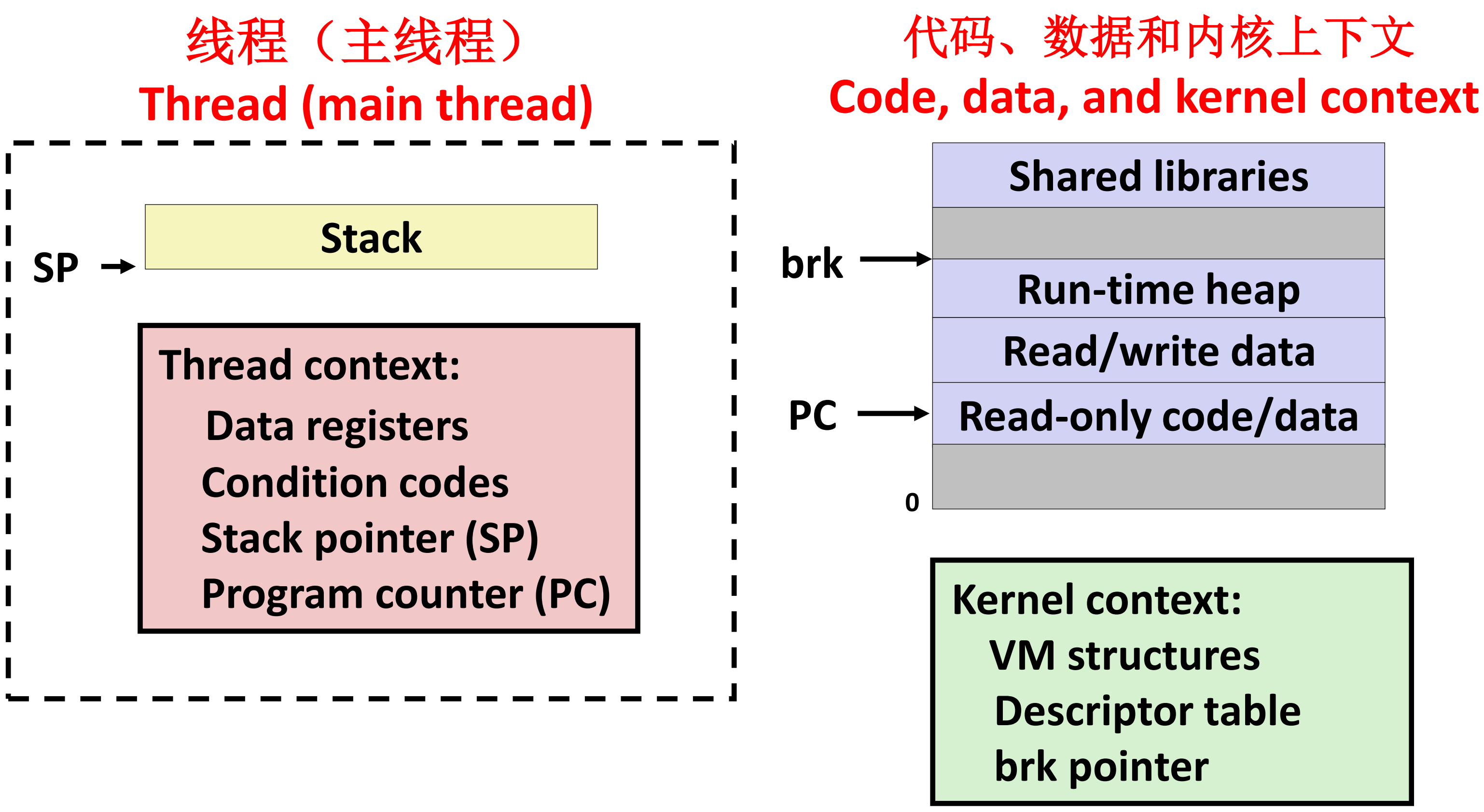
传统进程的视图 Traditional View of a Process

- 进程=进程上下文+代码、数据和栈 Process = process context + code, data, and stack



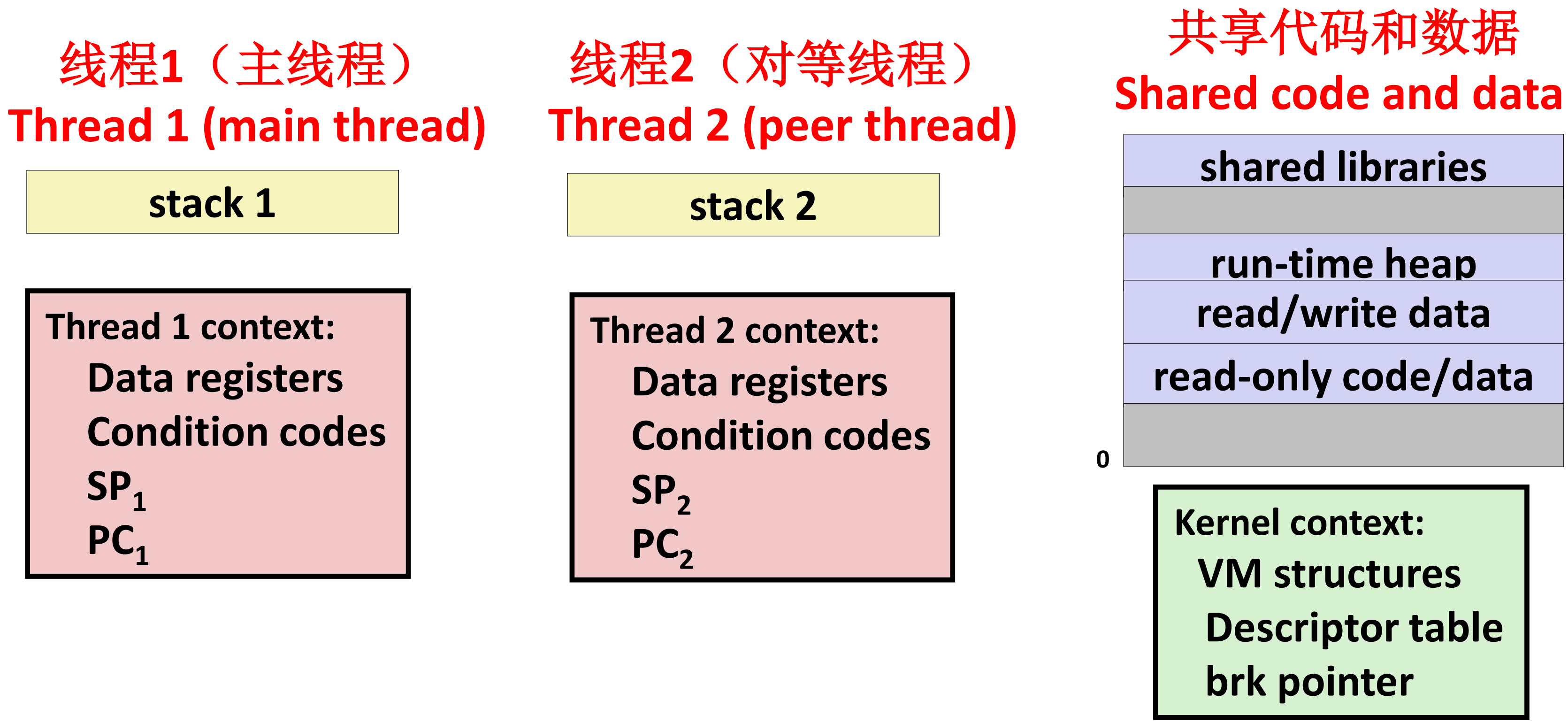
进程的替代视图 Alternate View of a Process

- 进程=线程+（代码、数据和内核上下文） **Process = thread + (code, data, and kernel context)**

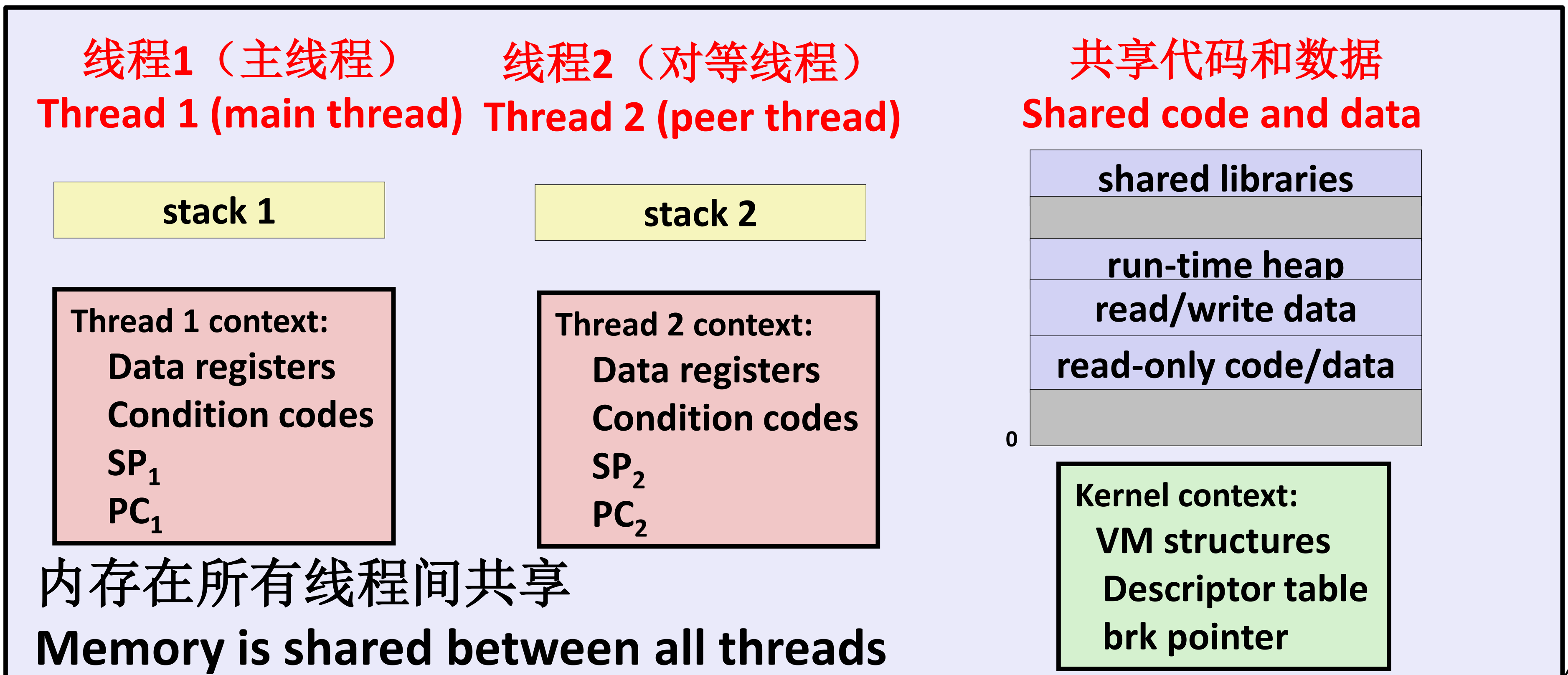


一个进程有多个线程-多线程进程
A Process With Multiple Threads

- 多个线程可以与一个进程关联 **Multiple threads can be associated with a process**
  - 每个线程都有自己的逻辑控制流 Each thread has its own logical control flow
  - 每个线程共享相同的代码、数据和内核上下文 Each thread shares the same code, data, and kernel context
  - 每个线程都有自己的局部变量栈 Each thread has its own stack for local variables
    - 但不受其他线程的保护 but not protected from other threads
  - 每个线程都有自己的线程id（TID） Each thread has its own thread id (TID)



不要让图片迷惑你！
Don’t let picture confuse you!



议题 Today

- 线程回顾 Threads review
- **共享 Sharing**
- 互斥 Mutual exclusion
- 信号量 Semaphores
- 生产者-消费者同步 Producer-Consumer Synchronization

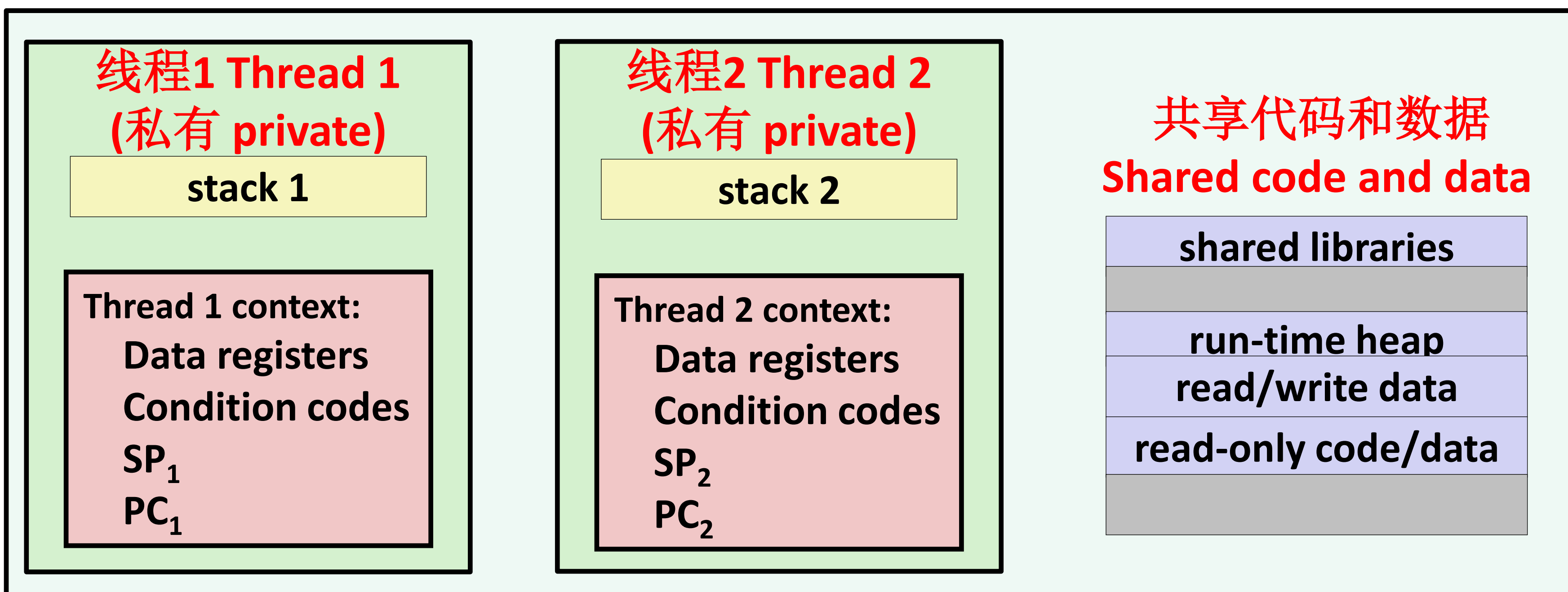
在线程化的C语言程序中共享变量
Shared Variables in Threaded C Programs

- 问题：线程化C程序中的哪些变量是共享的？ **Question: Which variables in a threaded C program are shared?**
  - 答案并不像"全局变量是共享的"和"栈变量是私有的"那么简单 The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"
- ***定义：***当且仅当多个线程引用x的某个实例时，变量x是共享的 ***Def:*** **A variable x is *shared* if and only if multiple threads reference some instance of x.**

- 需要以下问题的答案： **Requires answers to the following questions:**
  - 线程的内存模型是什么？ What is the memory model for threads?
  - 变量实例如何映射到内存？ How are instances of variables mapped to memory?
  - 有多少个线程可以引用每个实例？ How many threads might reference each of these instances?

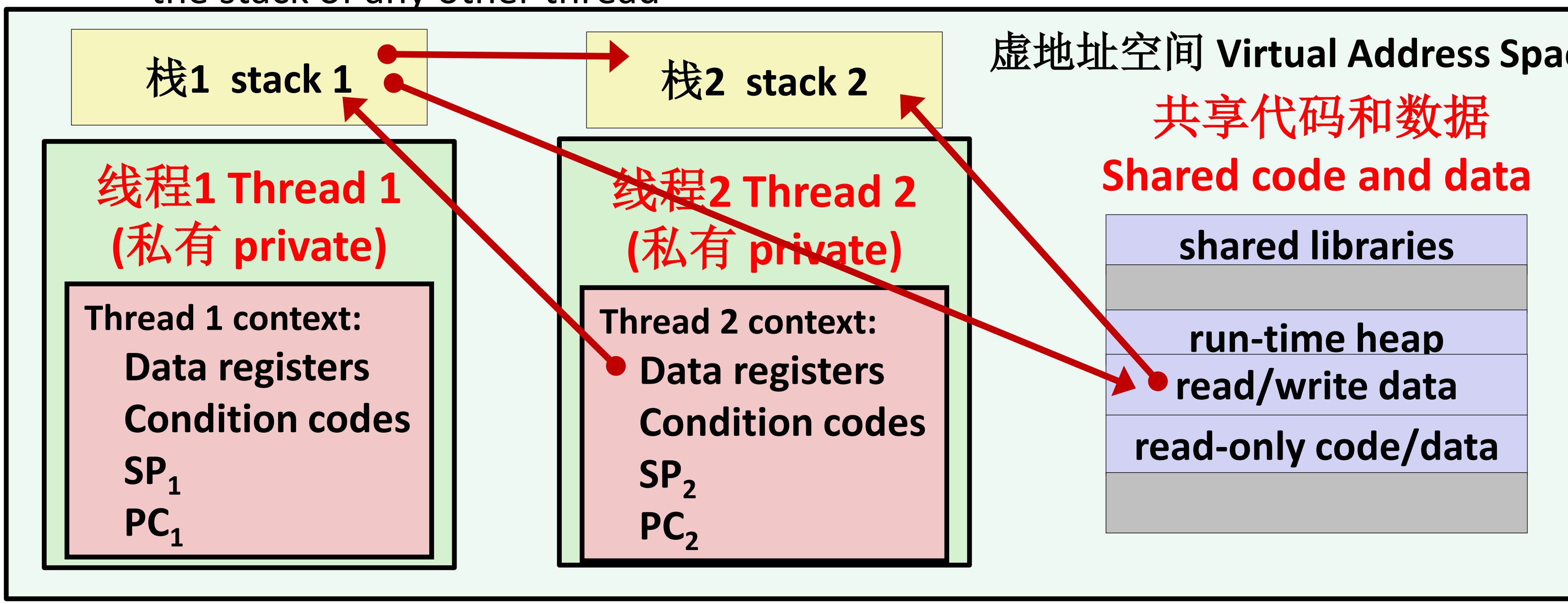
线程内存模型：概念上
Threads Memory Model: Conceptual

- 多个线程在单个进程的上下文中运行 **Multiple threads run within the context of a single process**
- 每个线程都有自己独立的线程上下文 **Each thread has its own separate thread context**
  - 线程ID、栈、栈指针、PC、条件码和GP寄存器 Thread ID, stack, stack pointer, PC, condition codes, and GP registers
- 所有线程共享剩余的进程上下文 **All threads share the remaining process context**
  - 进程虚拟地址空间的代码、数据、堆和共享库段 Code, data, heap, and shared library segments of the process virtual address space
  - 打开文件和安装的信号处理程序 Open files and installed handlers



线程内存模型：实际上
Threads Memory Model: Actual

- 未严格执行数据分离： **Separation of data is not strictly enforced:**
  - 寄存器值是真正独立和受保护的，但是… Register values are truly separate and protected, but…
  - 任何线程都可以读取和写入任何其他线程的栈 Any thread can read and write the stack of any other thread



概念模型和操作模型之间的不匹配是混淆和错误的根源

The mismatch between the conceptual and operation model is a source of confusion and errors

向线程传递参数 – 学究式方法
Passing an argument to a thread - Pedantic

```c
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++) {
      long* p = Malloc(sizeof(long));
      *p = i;
      Pthread_create(&tids[i],
                     NULL,
                     thread,
                     (void *)p);
   }
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```c
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   Free(vargp);
   return NULL;
}
```

```c
void check(void) {
   for (int i=0; i<N; i++) {
     if (hist[i] != 1) {
       printf("Failed at %d\n", i);
       exit(-1);
     }
   }
   printf("OK\n");
}
```

向线程传递参数 – 学究式方法 Passing an argument to a thread - Pedantic

```c
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++) {
      long* p = Malloc(sizeof(long));
      *p = i;
      Pthread_create(&tids[i],
                     NULL,
                     thread,
                     (void *)p);
   }
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```c
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   Free(vargp);
   return NULL;
}
```

- 使用malloc为每个线程分配堆内存存放参数 Use malloc to create a per thread heap allocated place in memory for the argument
- 记得在线程中释放内存！Remember to free in thread!
- 生产者-消费者模式 Producer-consumer pattern

向线程传递参数 – 另一种方法！
Passing an argument to a thread – Also OK!

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++)
     Pthread_create(&tids[i],
                    NULL,
                    thread,
                    (void *)i);
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[(long)vargp] += 1;
   return NULL;
}
```

- 使用强制转换也可以，因为长整数大小小于等于无类型指针的大小 **Ok to Use cast since sizeof(long) <= sizeof(void*)**
- 强制转换不会改变位模式 **Cast does NOT change bits**

向线程传递参数 – 警告！
Passing an argument to a thread – WRONG!

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
    long i;
    pthread_t tids[N];

    for (i = 0; i < N; i++)
      Pthread_create(&tids[i],
                     NULL,
                     thread,
                     (void *)&i);
    for (i = 0; i < N; i++)
      Pthread_join(tids[i], NULL);
    check();
}
```

```
void *thread(void *vargp)
{
   hist[*(long*)vargp] += 1;
   return NULL;
}
```

- 取i的地址对所有的线程来说都指向同样的位置
  &i points to same location for all threads!
- 产生数据竞争！ Creates a data race!

传递线程参数的三种方法
Three Ways to Pass Thread Arg

- 申请/释放空间 **Malloc/free**
  - 生产者申请空间，传递指针给pthread_create Producer malloc's space, passes pointer to pthread_create
  - 消费者释放指针空间 Consumer dereferences pointer
- 指向栈槽位 **Ptr to stack slot**
  - 生产者在pthread_create中传递生产者栈地址 Producer passes address to producer's stack in pthread_create
  - 消费者释放指针 Consumer dereferences pointer
- 强制转换成整数 **Cast of int**
  - 在pthread_create中生产者强制转换整数/长整数为地址 Producer casts an int/long to address in pthread_create
  - 消费者强制转换无类型指针参数回整数/长整数 Consumer casts void* argument back to int/long

示例程序说明共享
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

int main(int argc, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

对等线程间接通过全局ptr变量引用主线程的栈
Peer threads reference main thread's stack indirectly through global ptr variable

一种通用方法传递单个参数给一个线程例程 ***A common way to pass a single argument to a thread routine***

在线程化的C语言程序中共享变量
Shared Variables in Threaded C Programs

- **问题：线程化C程序中的哪些变量是共享的？ Question: Which variables in a threaded C program are shared?**
  - 答案并不像“全局变量是共享的”和“栈变量是私有的”那么简单 The answer is not as simple as “*global variables are shared*” and “*stack variables are private*”
- ***定义：*当且仅当多个线程引用x的某个实例时，变量x是共享的 *Def:* A variable x is *shared* if and only if multiple threads reference some instance of x.**

- **需要以下问题的答案： Requires answers to the following questions:**
  - 线程的内存模型是什么？ What is the memory model for threads?
  - 变量实例如何映射到内存？ How are instances of variables mapped to memory?
  - 有多少个线程可以引用每个实例？ How many threads might reference each of these instances?

映射变量实例到内存
Mapping Variable Instances to Memory

- 全局变量 **Global variables**
  - *定义：*在函数外部声明的变量 *Def:* Variable declared outside of a function
  - 虚拟内存仅包含任何全局变量的一个实例 **Virtual memory contains exactly one instance of any global variable**
- 局部变量 **Local variables**
  - *定义：*在函数内声明的没有静态属性的变量 *Def:* Variable declared inside function without `static` attribute
  - 每个线程栈包含每个局部变量的一个实例 **Each thread stack contains one instance of each local variable**
- 局部静态变量 **Local static variables**
  - *定义：*在函数内部声明的带有静态属性的变量 *Def:* Variable declared inside function with the `static` attribute
  - 虚拟内存只包含任何本地静态变量的一个实例 **Virtual memory contains exactly one instance of any local static variable.**

映射变量实例到内存
Mapping Variable Instances to Memory

```c
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

映射变量实例到内存

Mapping Variable Instances to Memory

全局变量：1个实例 ***Global var*:** 1 instance (`ptr` [data])

局部变量：1个实例 ***Local vars*:** 1 instance (`i.m`, `msgs.m`, `tid.m`)

```
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

局部变量：2个实例 ***Local var*:** 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

局部静态变量：1个实例
***Local static var*:** 1 instance (`cnt` [data])

共享变量分析 Shared Variable Analysis

- 哪些变量是共享的？ **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL, thread,(void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

共享变量分析 Shared Variable Analysis

- 哪些变量是共享的？ **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

- 答案：变量x是共享的，当且仅当多个线程引用最少一个x的实例，因此： **Answer: A variable `x` is shared iff multiple threads reference at least one instance of `x`. Thus:**
  - `ptr`、`cnt`和`msgs`是共享的 `ptr`, `cnt`, and `msgs` are shared
  - `i`和`myid`不是共享的 `i` and `myid` are *not* shared

同步线程 Synchronizing Threads

- 共享变量很方便。。。 **Shared variables are handy...**

- **……**但会引入严重同步错误的可能性 **…but introduce the possibility of nasty *synchronization* errors.**

badcnt.c:不正确的同步
badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
              *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt应该等于20,000 cnt should equal 20,000.

发生了什么错？ What went wrong?

计数循环的汇编代码
Assembly Code for Counter Loop

线程i中循环计数的C代码 C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

线程 i的汇编代码 Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

- H_i : Head 循环头
- L_i : Load cnt 装载cnt
- U_i : Update cnt 更新cnt
- S_i : Store cnt 存储cnt
- T_i : Tail 循环尾

并发执行 Concurrent Execution

- *关键思想：*一般来说，任何顺序一致的*指令交错执行都是可能的，但有些会产生意想不到的结果！ ***Key idea:*** **In general, any sequentially consistent* interleaving is possible, but some give an unexpected result!**
  - I_i表示线程i执行指令I I_i denotes that thread i executes instruction I
  - %rdxi是线程i上下文中%rdx的内容 $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

***现在。实际上，在x86上，甚至可以进行非顺序一致的指令交错执行**

****For now. In reality, on x86 even non-sequentially consistent interleavings are possible***

并发执行 Concurrent Execution

- *关键思想:* 一般来说，任何顺序一致的*指令交错执行都是可能的，但有些会产生意想不到的结果！ ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i表示线程i执行指令I I_i denotes that thread i executes instruction I
  - %rdxi是线程i上下文中%rdx的内容 $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

(green) 线程1临界区 Thread 1 critical section

(blue) 线程2临界区 Thread 2 critical section

OK

并发执行（续）
Concurrent Execution (cont)

- 不正确的顺序：两个线程递增计数器，但结果是1而不是2
  Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

哎呀！ Oops!

并发执行（续）
Concurrent Execution (cont)

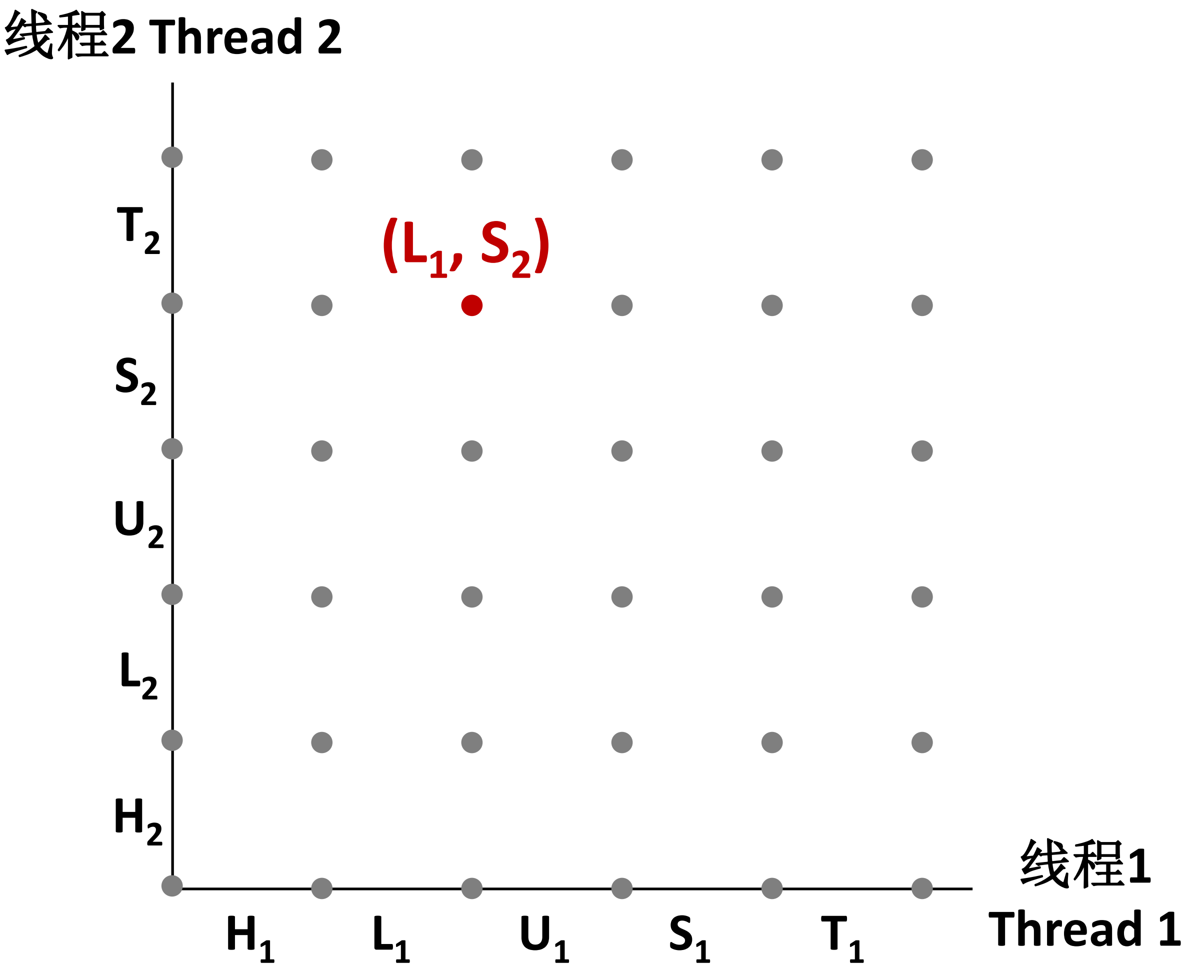
- 这个顺序会怎么样？ **How about this ordering?**

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

哎呀！ ***Oops!***

- 我们可以使用进度图分析行为 **We can analyze the behavior using a *progress graph***

进度图 Progress Graphs



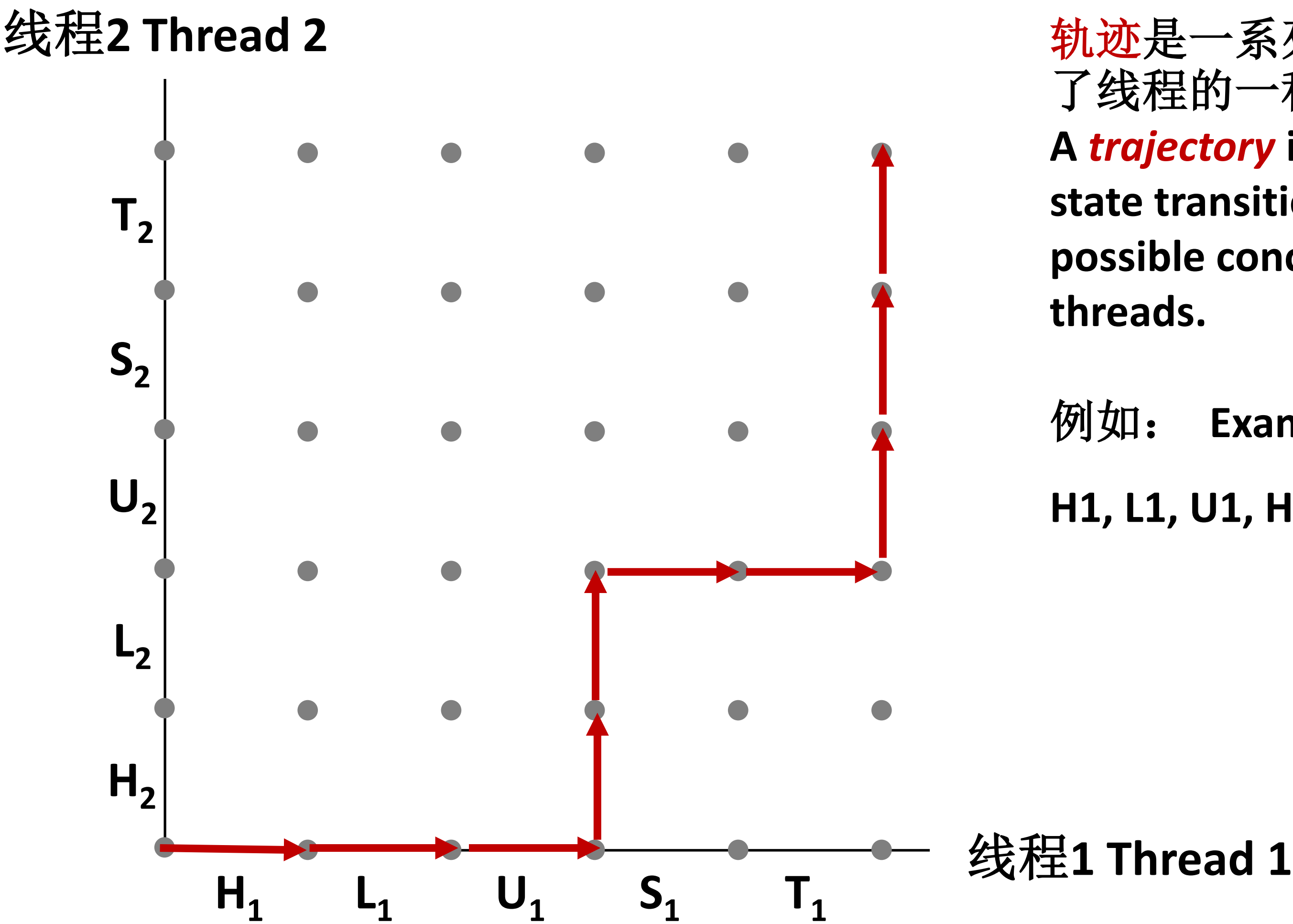
进度图描述了并发线程的离散执行状态空间 A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

每个轴对应于线程中的指令顺序 Each axis corresponds to the sequential order of instructions in a thread.

每个点对应于可能的**执行状态** Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$).

例如（L1，S2）表示状态，其中线程1已完成L1和线程2已完成S2 E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

进度图中的轨迹
Trajectories in Progress Graphs



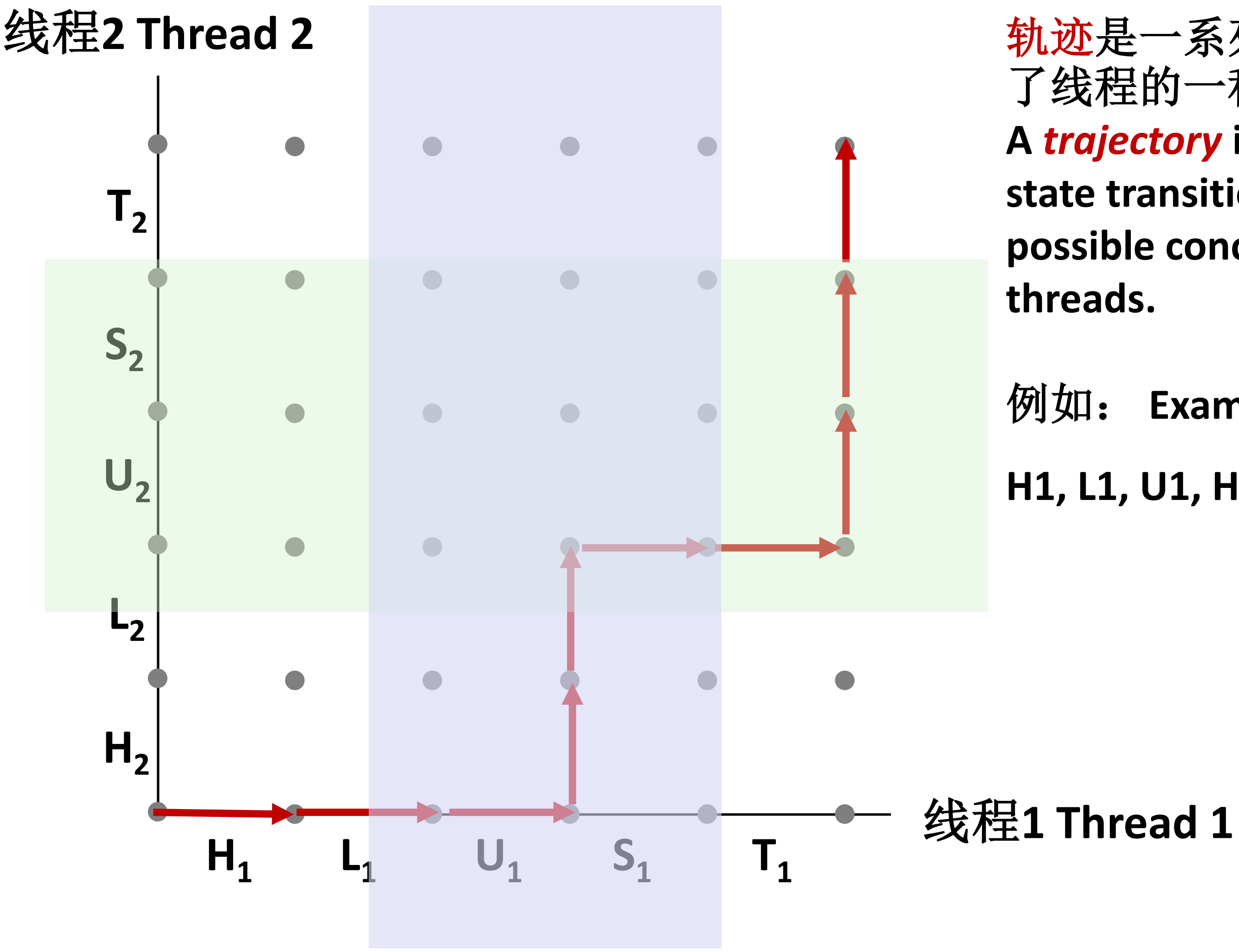
轨迹是一系列合法状态转换，描述了线程的一种可能并发执行。

A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

例如： **Example:**

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

进度图中的轨迹
Trajectories in Progress Graphs



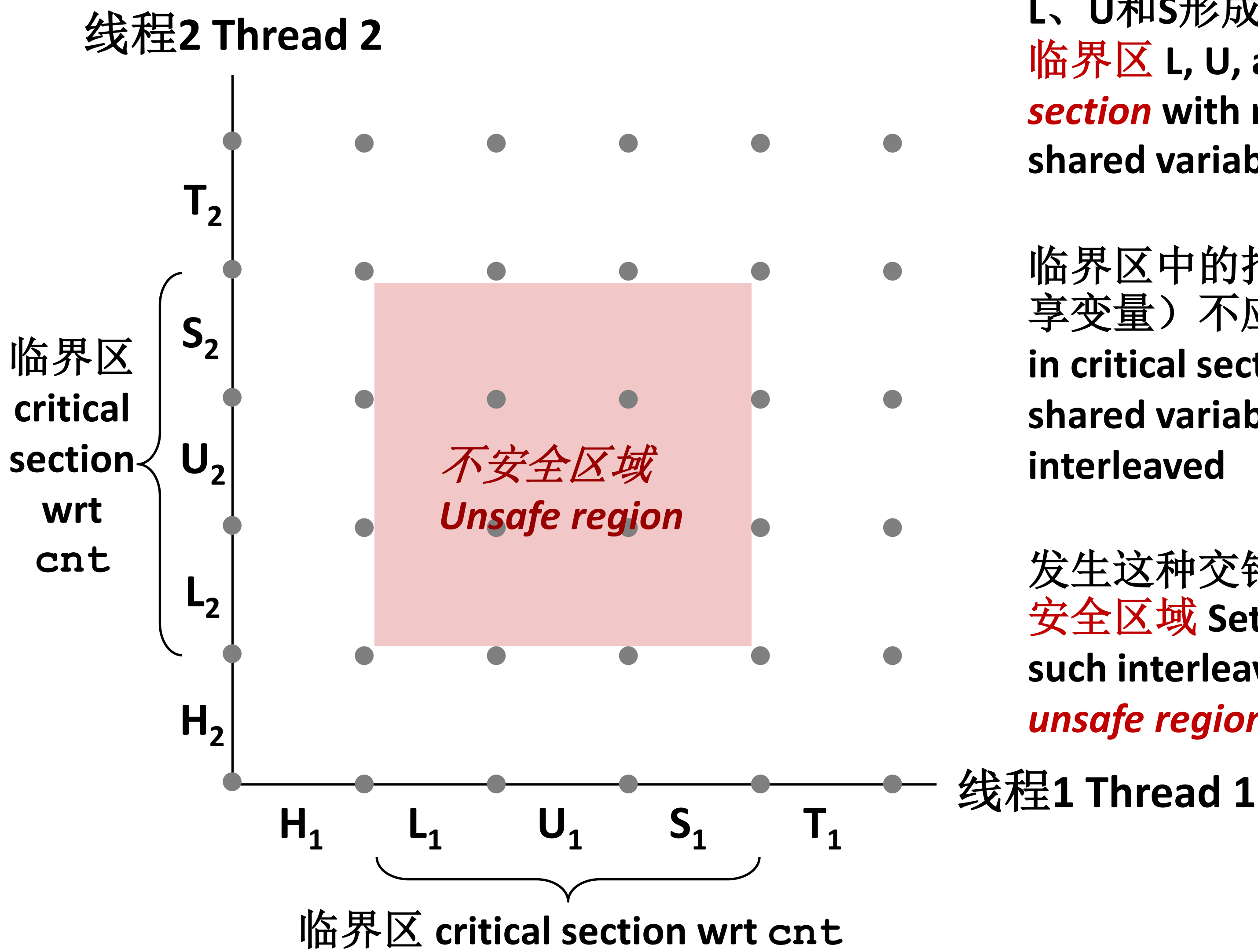
轨迹是一系列合法状态转换，描述了线程的一种可能并发执行。

A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

例如： **Example:**

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

临界区和不安全区域
Critical Sections and Unsafe Regions

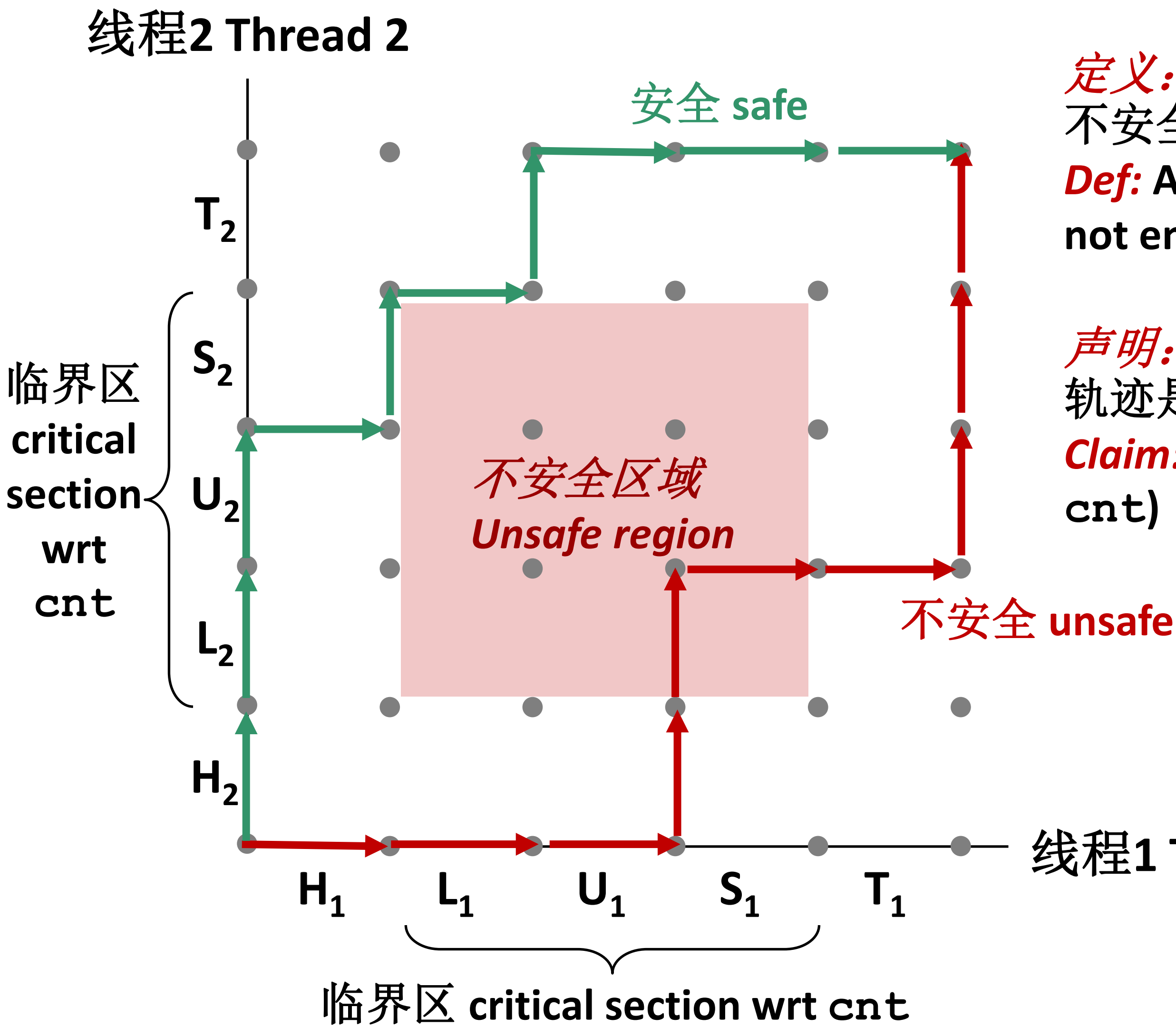


L、U和S形成关于共享变量cnt的临界区 L, U, and S form a *critical section* with respect to the shared variable `cnt`

临界区中的指令（写入一些共享变量）不应交错 Instructions in critical sections (wrt some shared variable) should not be interleaved

发生这种交错的状态集形成不安全区域 Sets of states where such interleaving occurs form *unsafe regions*

临界区和不安全区域
Critical Sections and Unsafe Regions



定义: 当且仅当轨迹不进入任何不安全区域，轨迹是安全的

Def: **A trajectory is *safe* iff it does not enter any unsafe region**

声明: 当且仅当轨迹是安全的，轨迹是正确的（写入cnt）

Claim: **A trajectory is correct (wrt `cnt`) iff it is safe**

badcnt.c: 不正确的同步

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
            *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

Variable	main	thread1	thread2
cnt			
niters.m			
tid1.m			
i.1			
i.2			
niters.1			
niters.2			

badcnt.c: 不正确的同步
badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
               *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	no	no
tid1.m	yes	no	no
i.1	no	yes	no
i.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

议题 Today

- 线程回顾 **Threads review**
- 共享 **Sharing**
- **互斥 Mutual exclusion**
- 信号量 Semaphores
- 生产者-消费者同步 **Producer-Consumer Synchronization**

执行互斥 Enforcing Mutual Exclusion

- *问题：*我们如何保证安全的轨迹？ ***Question:* How can we guarantee a safe trajectory?**

- 答：我们必须同步线程的执行，以便它们永远不会有不安全的轨迹 **Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - 即需要保证每个临界区的互斥访问 i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- 经典解决方案： **Classic solution:**
  - 互斥锁（pthreads） Mutex (pthreads)
  - 信号量（Edsger Dijkstra） Semaphores (Edsger Dijkstra)

- 其他方法（超出我们的讨论范围） **Other approaches (out of our scope)**
  - 条件变量（pthreads） Condition variables (pthreads)
  - 监视器（Java） Monitors (Java)

互斥锁（mutex）
MUTual EXclusion (mutex)

- *互斥锁*：布尔型同步变量 ***Mutex*: boolean synchronization variable**

- **enum {locked = 0, unlocked = 1}**

- **lock(m)**
  - 如果互斥锁当前未锁定，请锁定它并返回 If the mutex is currently not locked, lock it and return
  - 否则，等待（挂起、休眠等）并重试 Otherwise, wait (spinning, yielding, etc) and retry

- **unlock(m)**
  - 将互斥锁状态更新为解锁 Update the mutex state to unlocked

互斥锁（mutex）
MUTual EXclusion (mutex)

- *互斥锁*：布尔型同步变量* **Mutex: boolean synchronization variable** *

- **Swap(*a, b)**

  [t = *a; *a = b; return t;]

  // [] –通过硬件/OS的魔力实现原子操作 atomic by the magic of hardware / OS

- **Lock(m):**

  while (swap(&m->state, locked) == locked) ;

- **Unlock(m):**

  m->state = unlocked;

**现在。实际上，许多其他实现和设计选择（参见15-410、418等）。*

** For now. In reality, many other implementations and design choices (c.f., 15-410, 418, etc).*

badcnt.c: 不正确的同步
badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

如何使用同步解决此问题?

How can we fix this using synchronization?

goodmcnt.c:互斥锁同步
goodmcnt.c: Mutex Synchronization

- 为共享变量cnt定义并初始化互斥锁： **Define and initialize a mutex for the shared variable `cnt`:**

```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

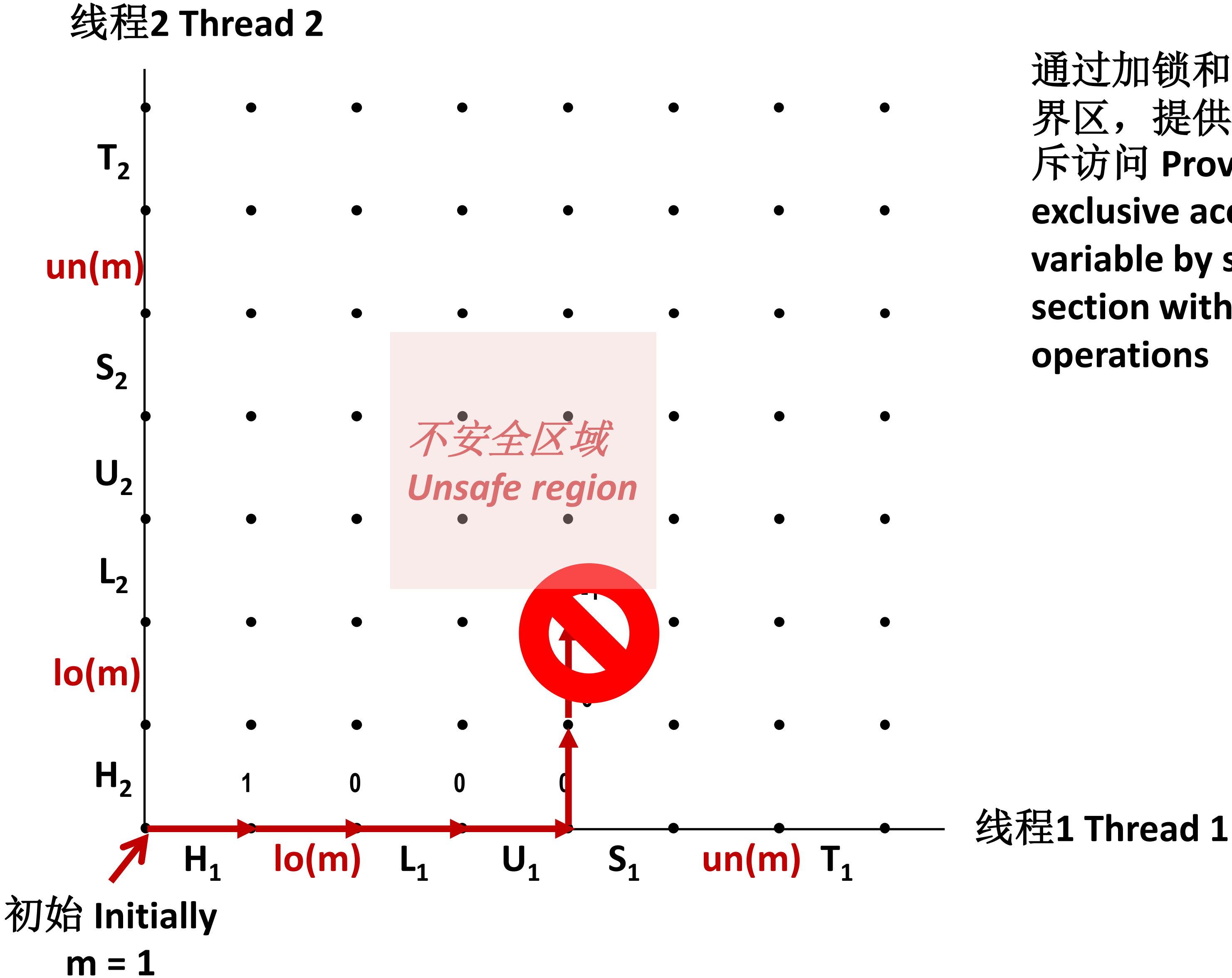
- 用加锁和解锁包围临界区： **Surround critical section with *lock* and *unlock*:**

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
```

```
linux> ./goodmcnt 10000
OK cnt=20000
linux> ./goodmcnt 10000
OK cnt=20000
linux>
```

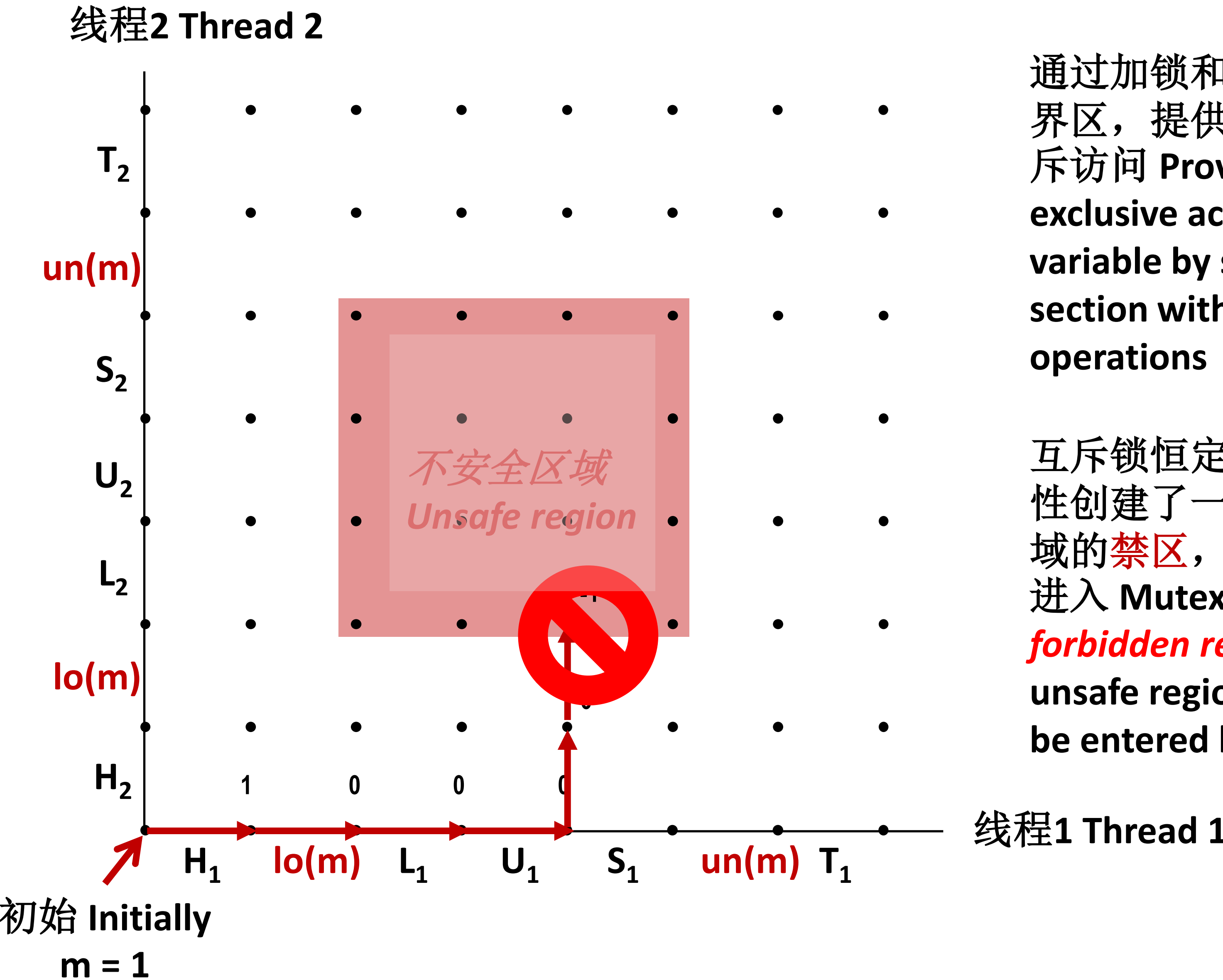
Function	badcnt	goodmcnt
Time (ms) niters = 10^6	12.0	214.0
减速 Slowdown	1.0	17.8

为什么互斥锁有效 Why Mutexes Work



通过加锁和解锁操作围绕临界区，提供对共享变量的互斥访问 Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

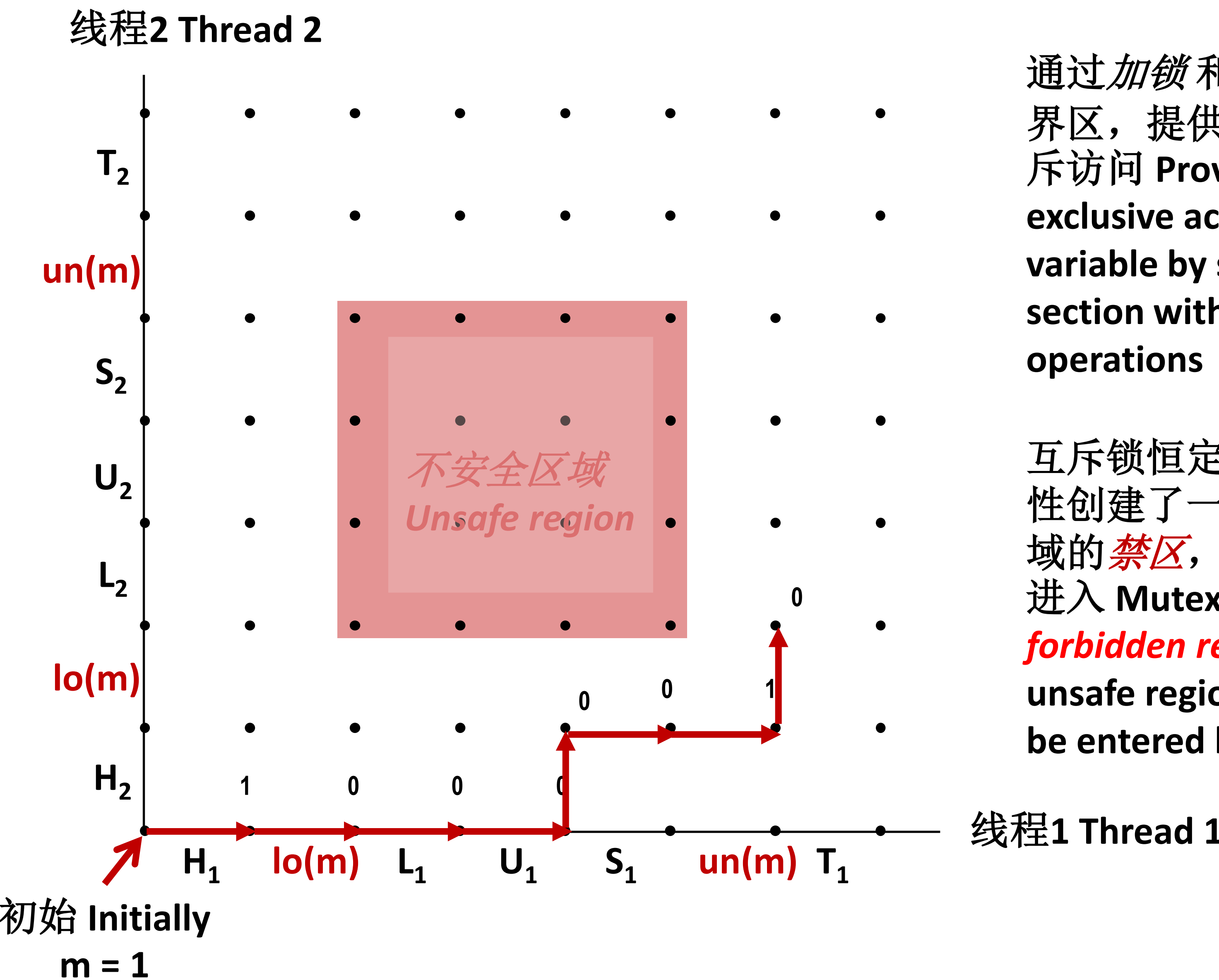
为什么互斥锁有效 Why Mutexes Work



通过加锁和解锁操作围绕临界区，提供对共享变量的互斥访问 Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的禁区，任何轨迹都无法进入 Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

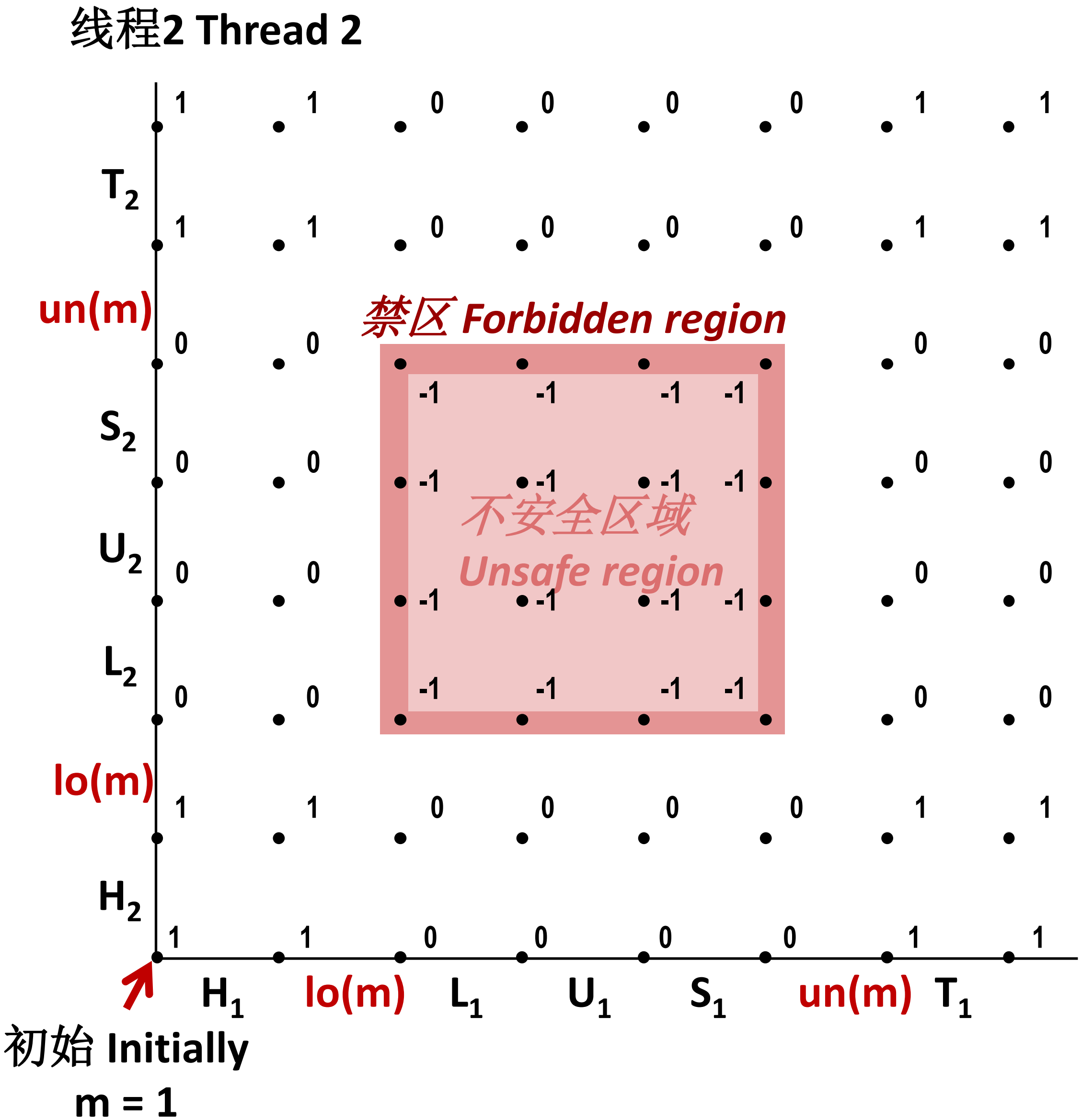
为什么互斥锁有效 Why Mutexes Work



通过*加锁*和*解锁*操作围绕临界区，提供对共享变量的互斥访问 **Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations**

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的*禁区*，任何轨迹都无法进入 **Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.**

为什么互斥锁有效 Why Mutexes Work



通过*加锁*和*解锁*操作围绕临界区，提供对共享变量的互斥访问 **Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations**

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的*禁区*，任何轨迹都无法进入 **Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.**

议题 Today

- 线程回顾/Threads review
- 共享 Sharing
- 互斥 Mutual exclusion
- **信号量 Semaphores**
- 生产者-消费者同步 Producer-Consumer Synchronization

信号量 Semaphores

- ***信号量：***非负全局整数同步变量，由P和V操作操纵 ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - 如果s为非零，则将s减1并立即返回 If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - 测试和减1操作以原子方式发生（不可分割） Test and decrement operations occur atomically (indivisibly)
  - 如果s为零，则挂起线程，直到s变为非零，并通过V操作重新启动线程 If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - 重新启动后，P操作将s减1并将控制权返回给调用者 After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - 将s递增1 Increment s by 1.
    - 增量操作以原子方式发生 Increment operation occurs atomically
  - 如果在P操作中有任何线程被阻塞，等待s变为非零，那么只重新启动其中一个线程，然后通过将s减1来完成P操作 If there are any threads blocked in a P operation waiting for s to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing s.
- **信号量恒定大于等于零： Semaphore invariant: *(s >= 0)***

信号量 Semaphores

- *信号量：*非负全局整数同步变量 ***Semaphore:*** **non-negative global integer synchronization variable**

- **由P和V操作操纵 Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0) wait(); s--;`]
    - 荷兰语单词“Proberen”（测试） Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - 荷兰语单词“Verhogen”（增加） Dutch for "Verhogen" (increment)

- **OS内核保证括号[]之间的操作不可分割地执行 OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
  - 一次只能一个P或V操作修改s Only one *P* or *V* operation at a time can modify s.
  - 当P中的while循环终止时，只有该P操作可以减少s When `while` loop in *P* terminates, only that *P* can decrement `s`

- **信号量恒定大于等于零： Semaphore invariant: *(s >= 0)***

C语言信号量操作
C Semaphore Operations

Pthread函数 Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS：APP包装器函数 CS:APP wrapper functions:

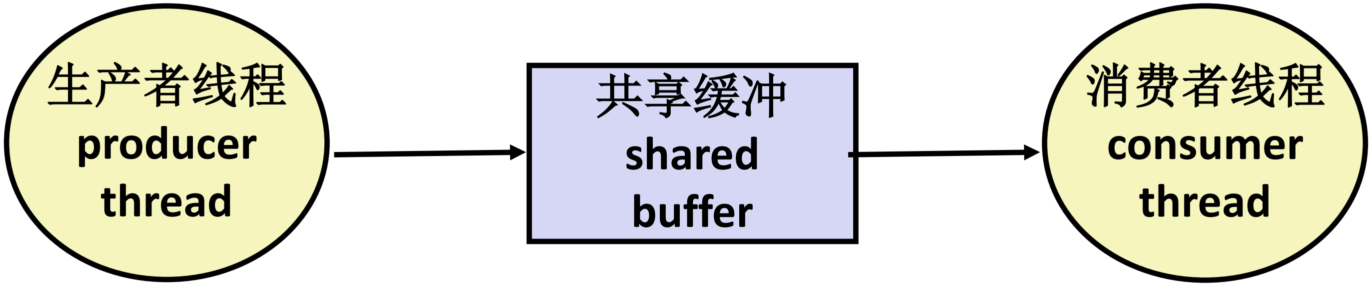
```
#include "csapp.h”

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

使用信号量协调共享资源的访问 Using Semaphores to Coordinate Access to Shared Resources

- 基本思想：线程使用信号量操作通知另一个线程某些条件已变为真 **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - 使用计数信号量来跟踪资源状态 Use counting semaphores to keep track of resource state.
  - 使用二元信号量通知其他线程 Use binary semaphores to notify other threads.

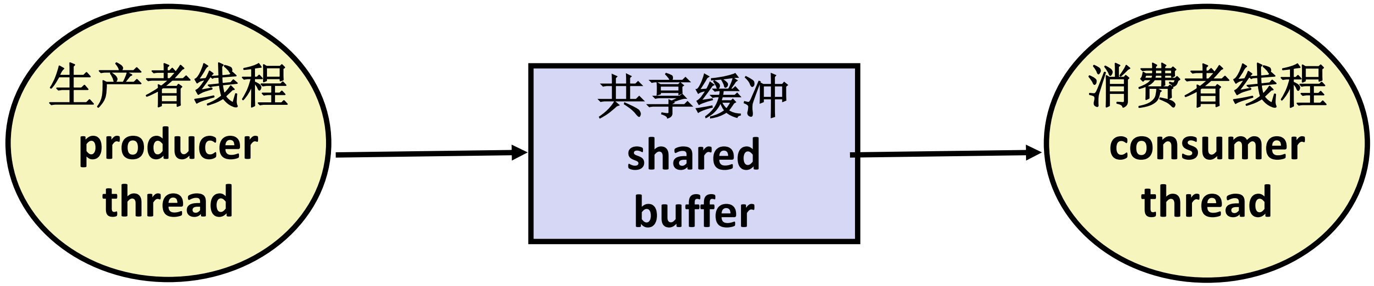
- 生产者-消费者问题 **The Producer-Consumer Problem**
  - 对进程之间的交互操作进行协调，一个进程产生信息，另一个进程使用这些消息 Mediating interactions between processes that generate information and that then make use of that information

生产者-消费者问题
Producer-Consumer Problem



- 通用同步模式： **Common synchronization pattern:**
  - 生产者等待空槽，将项目插入缓冲区，并通知消费者 Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - 消费者等待项目，将其从缓冲区中删除，并通知生产者 Consumer waits for ***item***, removes it from buffer, and notifies producer

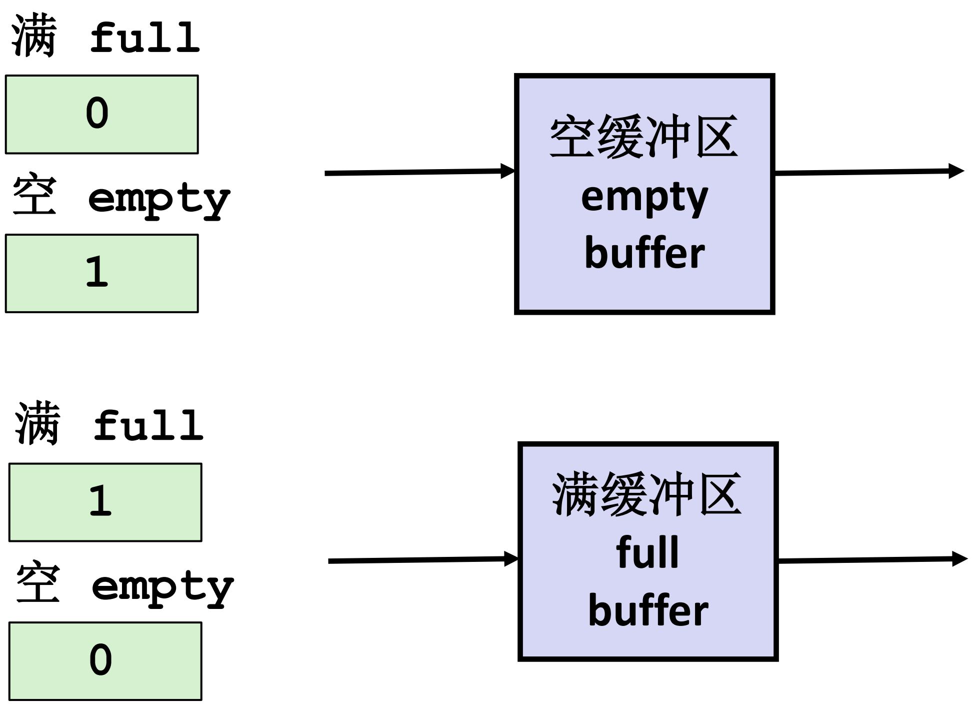
生产者-消费者问题
Producer-Consumer Problem



- **示例 Examples**
  - 多媒体处理： Multimedia processing:
    - 生产者创建视频帧，消费者对其进行渲染 Producer creates video frames, consumer renders them
  - 事件驱动的图形用户界面 Event-driven graphical user interfaces
    - 生产者检测鼠标点击、鼠标移动和键盘点击，并在缓冲区中插入相应的事件 Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - 消费者从缓冲区检索事件并绘制显示 Consumer retrieves events from buffer and paints the display

生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

- 维护两个信号量：缓冲区满full+缓冲区空 **Maintain two semaphores: `full` + `empty`**



生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

```c
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;
```

```c
int main(int argc, char** argv) {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* Initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full,  0, 0);

  /* Create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  return 0;
}
```

生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

初始： **Initially:** `empty==1, full==0`

生产者线程 Producer Thread

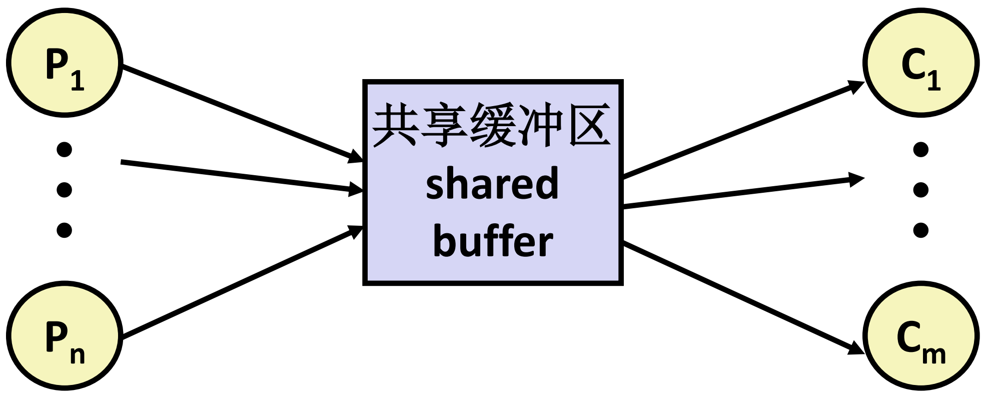
```
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* Write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

消费者线程 Consumer Thread

```
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* Consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```

为何对一个条目的缓冲区使用2个信号量？
Why 2 Semaphores for 1-Entry Buffer?

- 考虑多个生产者和多个消费者 Consider multiple producers & multiple consumers



- 生产者将与每个人竞争以获得空缓冲区 Producers will contend with each to get `empty`
- 消费者将相互竞争以获得满缓冲区 Consumers will contend with each other to get `full`

生产者 Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

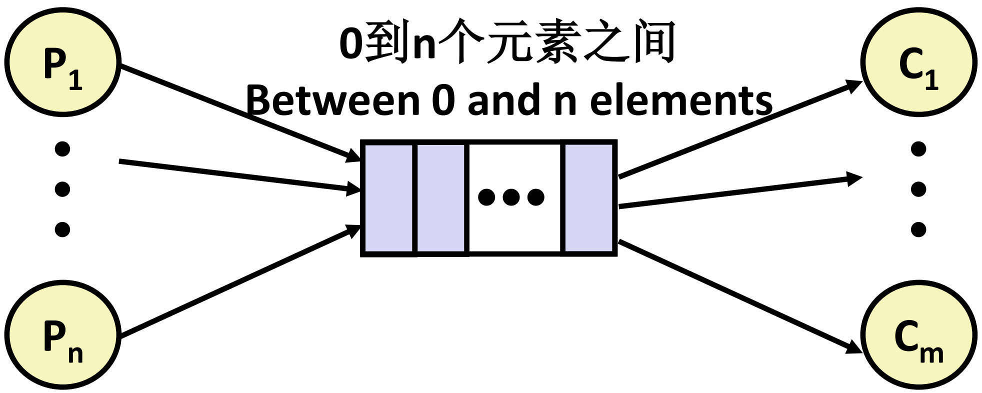
空 `empty`

满 `full`

消费者 Consumers

```
P(&shared.full);
item = shared.buf;
V(&shared.empty);
```

生产者和消费者之间有n个元素的缓冲区
Producer-Consumer on an *n*-element Buffer



- 使用名为sbuf的共享缓冲区包实现 **Implemented using a shared buffer package called `sbuf`.**

环形缓冲区（n=10）
Circular Buffer (n = 10)

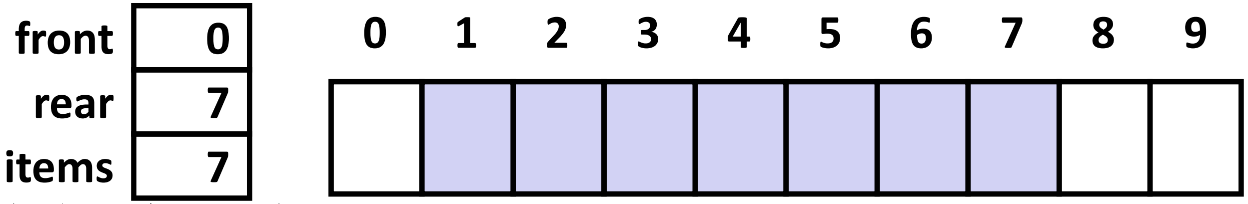
- 将元素存储在大小为n的数组中 **Store elements in array of size n**
- 项目：缓冲区中的元素数 **items: number of elements in buffer**
- 空缓冲区： **Empty buffer:**
  - front = rear
- 非空缓冲区 **Nonempty buffer**
  - rear：最近插入的元素的索引 rear: index of most recently inserted element
  - front：(要删除的下一个元素的索引–1)mod n front: (index of next element to remove – 1) mod n
- 初始 **Initially:**

front	**0**
rear	**0**
项目 **items**	**0**

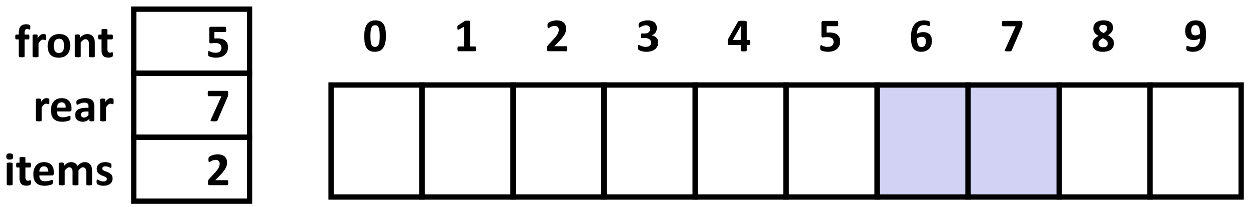
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

环形缓冲区操作（n=10）
Circular Buffer Operation (n = 10)

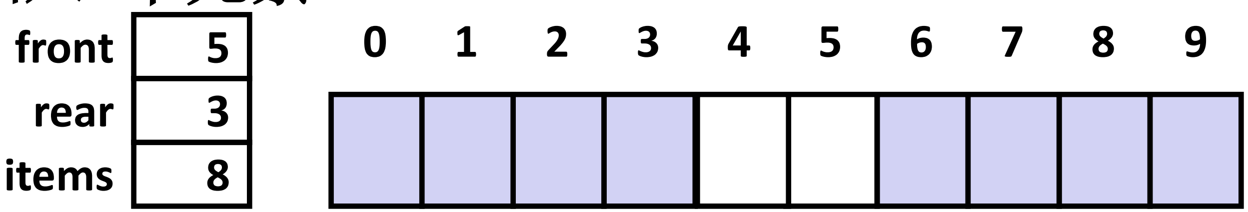
- 插入7个元素 Insert 7 elements



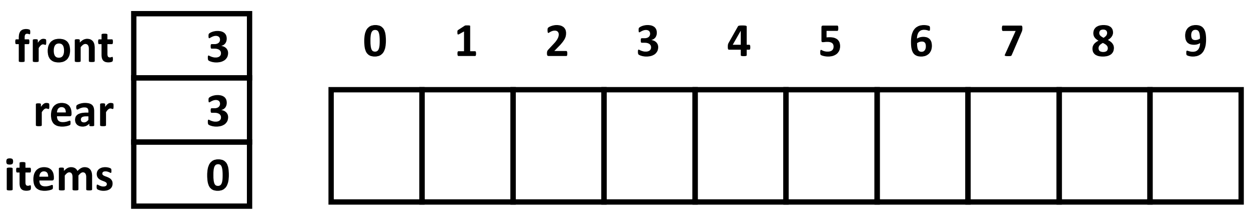
- 删除5个元素 Remove 5 elements



- 插入6个元素 Insert 6 elements



- 删除8个元素 Remove 8 elements



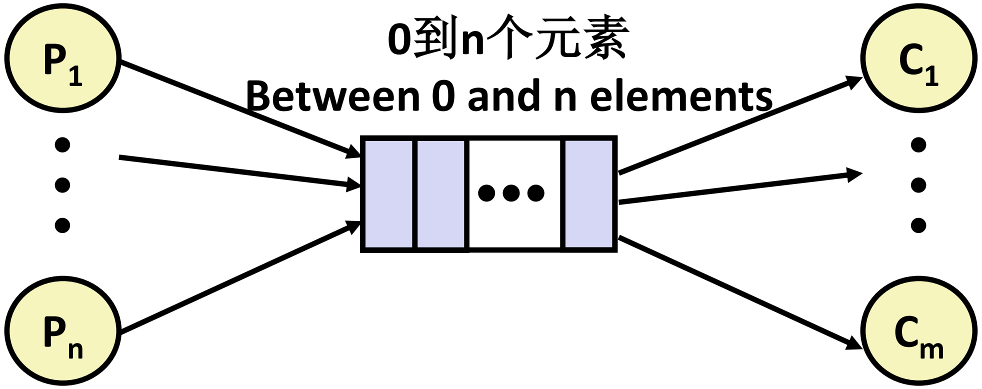
顺序环形缓冲区代码
Sequential Circular Buffer Code

```
init(int v)
{
    items = front = rear = 0;
}
```

```
insert(int v)
{
    if (items >= n)
        error();
    if (++rear >= n) rear = 0;
    buf[rear] = v;
    items++;
}
```

```
int remove()
{
    if (items == 0)
        error();
    if (++front >= n) front = 0;
    int v = buf[front];
    items--;
    return v;
}
```

生产者和消费者之间有n个元素的缓冲区
Producer-Consumer on an *n*-element Buffer



- 需要一个互斥锁和两个计数信号量： **Requires a mutex and two counting semaphores:**
  - 互斥锁：执行对缓冲区和计数器进行互斥访问 `mutex`: enforces mutually exclusive access to the buffer and counters
  - 槽位数：统计缓冲区中的可用槽位 `slots`: counts the available slots in the buffer
  - 项目：统计缓冲区中的可用项目 `items`: counts the available items in the buffer
- 使用通用信号量 **Makes use of general semaphores**
  - 值范围从0到n Will range in value from 0 to n

sbuf包-声明 sbuf Package - Declarations

```
#include "csapp.h"

typedef struct {
    int *buf;          /* Buffer array                        */
    int n;             /* Maximum number of slots             */
    int front;         /* buf[front+1 (mod n)] is first item */
    int rear;          /* buf[rear]   is last item            */
    pthread_mutex_t mutex; /* Protects accesses to buf   */
    sem_t slots;       /* Counts available slots              */
    sem_t items;       /* Counts available items              */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
```

sbuf.h

sbuf包-实现
sbuf Package - Implementation

初始化和释放共享缓冲区 **Initializing and deinitializing a shared buffer:**

```
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                       /* Buffer holds max of n items */
    sp->front = sp->rear = 0;        /* Empty buffer iff front == rear */
    pthread_mutex_init(&sp->mutex, NULL); /* lock */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

sbuf.c

sbuf包-实现
sbuf Package - Implementation

插入一个项目到共享缓冲区 Inserting an item into a shared buffer:

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    pthread_mutex_lock(&sp->mutex);  /* Lock the buffer          */
    if (++sp->rear >= sp->n)            /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item;        /* Insert the item          */
    pthread_mutex_unlock(&sp->mutex); /* Unlock the buffer   */
    V(&sp->items);                          /* Announce available item */
}
```

sbuf.c

sbuf包-实现
sbuf Package - Implementation

从共享缓冲区删除一个项目 **Removing an item from a shared buffer:**

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                      /* Wait for available item */
    pthread_mutex_lock(&sp->mutex);     /* Lock the buffer         */
    if (++sp->front >= sp->n)           /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];          /* Remove the item         */
    pthread_mutex_unlock(&sp->mutex);   /* Unlock the buffer       */
    V(&sp->slots);                      /* Announce available slot */
    return item;
}
```

sbuf.c

演示 Demonstration

- **参见code目录中的程序produce-consume.c See program produce-consume.c in code directory**
- **10个条目的共享环形缓冲区 10-entry shared circular buffer**
- **5个生产者 5 producers**
  - 代理i生成从20*i到20*i–1的数字 Agent i generates numbers from 20*i to 20*i – 1.
  - 将它们放入缓冲区 Puts them in buffer
- **5个消费者 5 consumers**
  - 每个从缓冲区中检索20个元素 Each retrieves 20 elements from buffer
- **主程序 Main program**
  - 确保0到99之间的每个值检索一次 Makes sure each value between 0 and 99 retrieved once

小结 Summary

- 程序员需要一个线程如何共享变量的清晰模型。
Programmers need a clear model of how variables are shared by threads.

- 必须保护多个线程共享的变量，以确保互斥访问
Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- 信号量是执行互斥的基本机制 **Semaphores are a fundamental mechanism for enforcing mutual exclusion.**

第12章 并发编程

同步：高级/Synchronization: Advanced

100076202： 计算机系统导论

任课教师：

宿红毅　　张艳　　黎有琦　　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

议题 Today

- **回顾：信号量、互斥和生产者-消费者 Review: Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - **读者-写者问题 Readers-writers problem**
- 其它并发问题 **Other concurrency issues**
  - 线程安全 Thread safety
  - 竞争 Races
  - 死锁 Deadlocks
  - 线程和信号处理之间交互 Interactions between threads and signal handling

提醒：信号量
Reminder: Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable**

- **Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0); s--;`]
    - Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - Dutch for "Verhogen" (increment)

- **OS kernel guarantees that operations between brackets [ ] are executed atomically**
    - Only one *P* or *V* operation at a time can modify s.
    - When `while` loop in *P* terminates, only that *P* can decrement `s`

- **Semaphore invariant: *(s >= 0)***

回顾：使用信号量通过互斥保护共享资源 Review: Using semaphores to protect shared resources via mutual exclusion

- 基本思想： **Basic idea:**
  - 将一个唯一的信号量互斥锁（mutex）（最初为1）与每个共享变量（或相关的共享变量集）相关联 Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables)
  - 用P(mutex)和V(mutext)操作包围对共享变量的每次访问 Surround each access to the shared variable(s) with *P(mutex)* and *V(mutex)* operations

```
mutex = 1

P(mutex)
cnt++
V(mutex)
```

回顾：使用锁进行互斥
Review: Using Lock for Mutual Exclusion

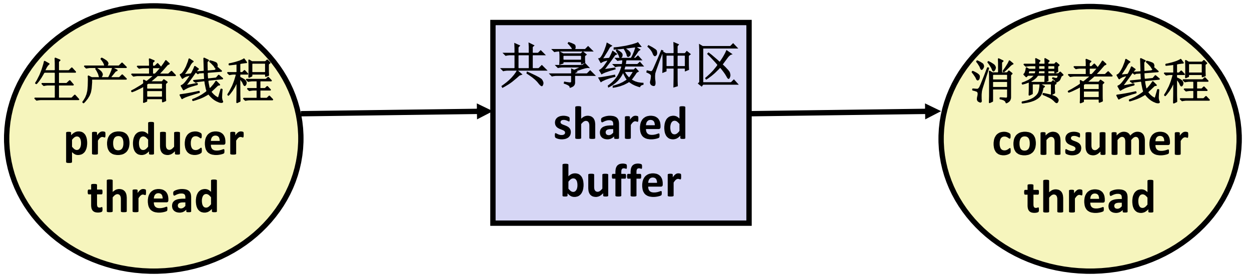
- **基本思想： Basic idea:**
  - 互斥锁Mutex是只有值0（锁定）或1（解锁）的信号量的特殊情况 Mutex is special case of semaphore that only has value 0 (locked) or 1 (unlocked)
  - *Lock(m):* [`while (m == 0); m=0;`]
  - *Unlock(m):* [`m=1`]
- **比使用信号量快约2倍 ~2x faster than using semaphore for this purpose**
  - 而且，更清楚地表明程序员的意图 And, more clearly indicates programmer's intention

```
mutex = 1

lock(mutex)
cnt++
unlock(mutex)
```

关于示例的注释 **Note about Examples**

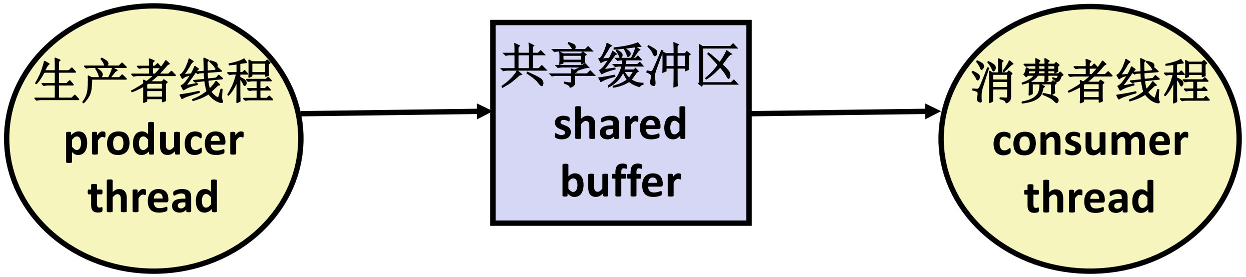
- 课程示例将使用信号量进行计数和互斥 **Lecture examples will use semaphores for both counting and mutual exclusion**
  - 代码比使用pthread_mutex短得多 Code is much shorter than using pthread_mutex

回顾：生产者-消费者问题
Review: Producer-Consumer Problem



- 通用同步模式：**Common synchronization pattern:**
  - 生产者等待空**槽位**，将项目插入缓冲区，并通知消费者 Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - 消费者等待**项目**，将其从缓冲区中删除，并通知生产者 Consumer waits for ***item***, removes it from buffer, and notifies producer

回顾：生产者-消费者问题
Review: Producer-Consumer Problem



- 示例 **Examples**
  - 多媒体处理：　Multimedia processing:
    - 生产者创建视频帧，消费者对其进行渲染　Producer creates video frames, consumer renders them
  - 事件驱动的图形用户界面　Event-driven graphical user interfaces
    - 生产者检测鼠标点击、鼠标移动和键盘点击，并在缓冲区中插入相应的事件　Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - 消费者从缓冲区检索事件并绘制显示　Consumer retrieves events from buffer and paints the display

回顾：使用信号量协调共享资源的访问 Review: Using Semaphores to Coordinate Access to Shared Resources

- 基本思想：线程使用信号量操作通知另一个线程某些条件已变为真 **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - 使用计数信号量来跟踪资源状态 Use counting semaphores to keep track of resource state.
  - 使用二元信号量通知其他线程 Use binary semaphores to notify other threads.

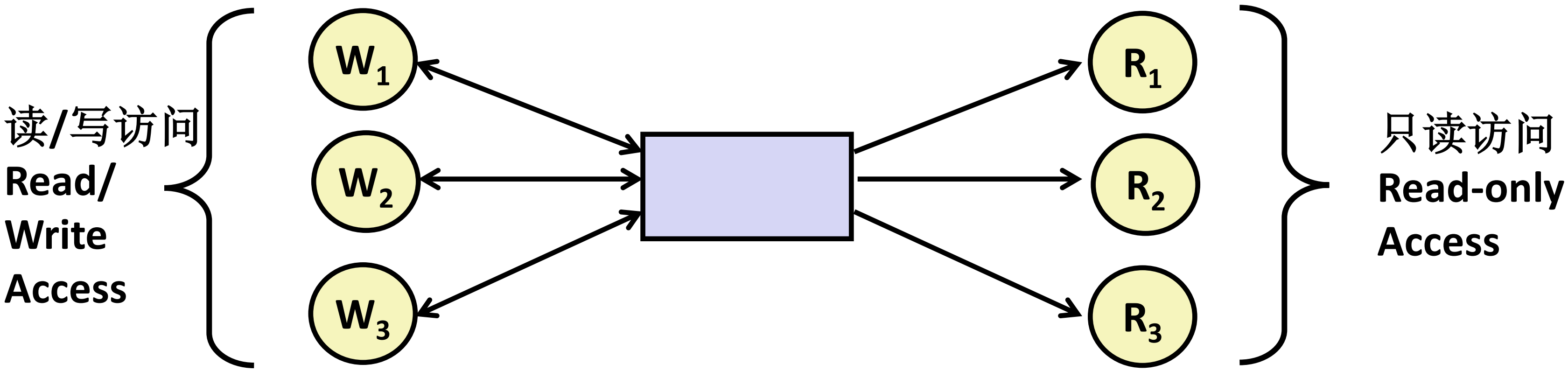
回顾：使用信号量协调共享资源的访问 Review: Using Semaphores to Coordinate Access to Shared Resources

- 生产者-消费者问题 **The Producer-Consumer Problem**
  - 对进程之间的交互活动进行协调，一个进程产生信息，另一个进使用该信息 Mediating interactions between processes that generate information and that then make use of that information
  - 用两个二元信号量实现单条目缓冲区 Single entry buffer implemented with two binary semaphores
    - 一个用于控制生产者的访问 One to control access by producer(s)
    - 一个用于控制消费者的访问 One to control access by consumer(s)
  - 使用信号量+环形缓冲区实现N个条目缓冲区 N-entry implemented with semaphores + circular buffer

议题 Today

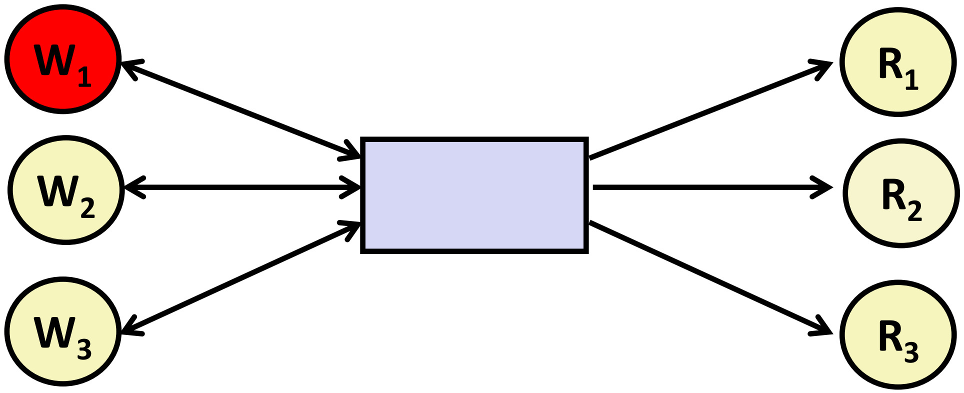
- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- **使用信号量调度共享资源 Using semaphores to schedule shared resources**
  - **读者-写者问题 Readers-writers problem**
- 其它并发问题 **Other concurrency issues**
  - 线程安全 Thread safety
  - 竞争 Races
  - 死锁 Deadlocks
  - 线程和信号处理交互 Interactions between threads and signal handling

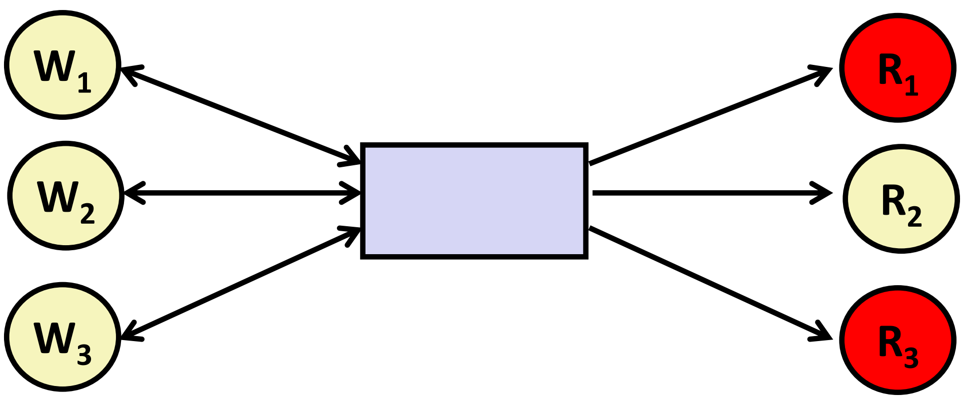
读者和写者问题
Readers-Writers Problem



- 问题陈述： **Problem statement:**
  - *读者*线程仅读取对象 *Reader* threads only read the object
  - *写者*线程修改对象（读/写访问） *Writer* threads modify the object (read/write access)
  - 写者必须具有对对象的独占访问权限 Writers must have exclusive access to the object
  - 无限数量的读者可以访问该对象 Unlimited number of readers can access the object
- 在真实系统中频繁发生，例如 **Occurs frequently in real systems, e.g.,**
  - 在线航空预订系统 Online airline reservation system
  - 多线程缓存Web代理 Multithreaded caching Web proxy

读者/写者示例
Readers/Writers Examples





读者和写者的变体 Variants of Readers-Writers

- 第一类读者-写者问题（有利于读者-读者优先）***First readers-writers problem*** **(favors readers)**
  - 除非已授予写者使用对象的权限，否则不应让任何读者等待 No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - 等待写者之后到达的读者比写者优先 A reader that arrives after a waiting writer gets priority over the writer.
- 第二类读者-写者问题（有利于写者-写者优先） ***Second readers-writers problem*** **(favors writers)**
  - 一旦写者准备好写入，它将尽快执行写入 Once a writer is ready to write, it performs its write as soon as possible
  - 在写者之后到达的读者必须等待，即使写者也要等待 A reader that arrives after a writer must wait, even if the writer is also waiting.
- 在这两种情况下都有可能出现饿死情况（线程无限期等待）***Starvation*** **(where a thread waits indefinitely) is possible in both cases.**

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

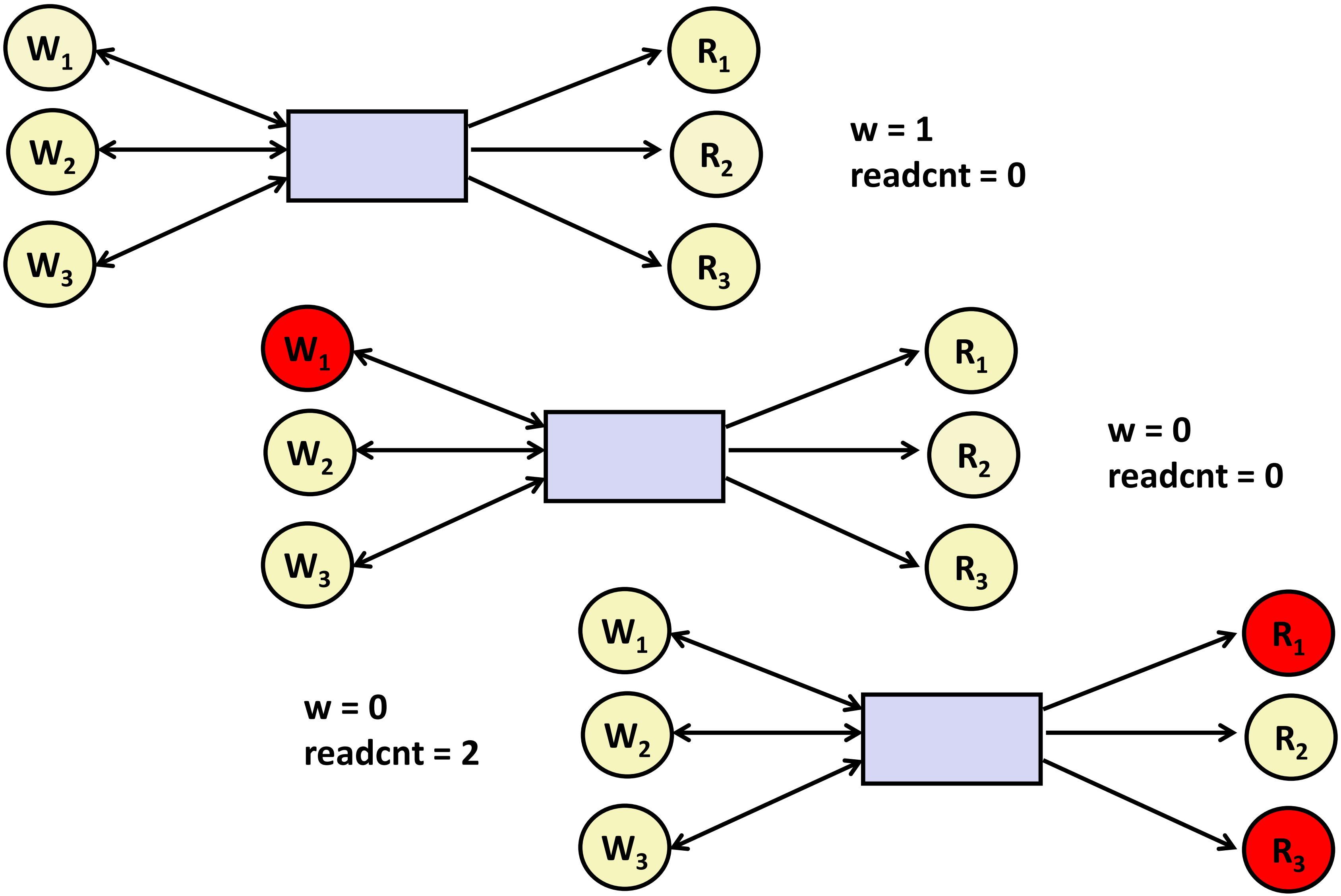
```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

读者/写者示例
Readers/Writers Examples



第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： **Arrivals: R1 R2 W1 R3**

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;     /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

R1 →  /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： **Arrivals: R1 R2 W1 R3**

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

R2 → (at `if (readcnt == 1)`)

R1 → (at `/* Reading happens here */`)

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 Readers:

R2 →
R1 →

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {   ← W1
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： **Arrivals: R1 R2 W1 R3**

Readcnt == 2
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

R2 →  /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
R1 → }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {   ← W1
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;     /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

R3 (arrow at `if (readcnt == 1)`), R2 (arrow before second `P(&mutex);`), R1 (arrow at closing `}` of while loop)

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

W1 (arrow at `P(&w);`)

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

第一类读者-写者问题的解决方案
Solution to First Readers-Writers Problem

读者 **Readers:**

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

R3 → (at /* Reading happens here */)

R2 → (at V(&mutex);)

写者 **Writers:**

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

W1 → (at P(&w);)

rw1.c

到达： **Arrivals: R1 R2 W1 R3**

Readcnt == 1
W == 0

第一类读者-写者问题的解决方案

Solution to First Readers-Writers Problem

读者 Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
R3 →V(&mutex);
  }
}
```

写者 Writers:

```
void writer(void)
{
  while (1) {
    P(&w);     ← W1

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

到达： Arrivals: R1 R2 W1 R3

Readcnt == 0
W == 1

其它读者-写者版本
Other Versions of Readers-Writers

- 第一类解决方案的不足 **Shortcoming of first solution**
  - 源源不断的读者将无限期地阻止写者 Continuous stream of readers will block writers indefinitely
- 第二个版本 **Second version**
  - 一旦写者出现，就会阻止以后的读者访问 Once writer comes along, blocks access to later readers
  - 一系列写入可能会阻止所有读取 Series of writes could block all reads
- 先进先出实现 **FIFO implementation**
  - 请参阅code目录中的rwqueue代码 See rwqueue code in code directory
  - 按顺序接收服务请求 Service requests in order received
  - 保存线程在先进先出队列中 Threads kept in FIFO
  - 每一个都有信号量，可以访问临界区 Each has semaphore that enables its access to critical section

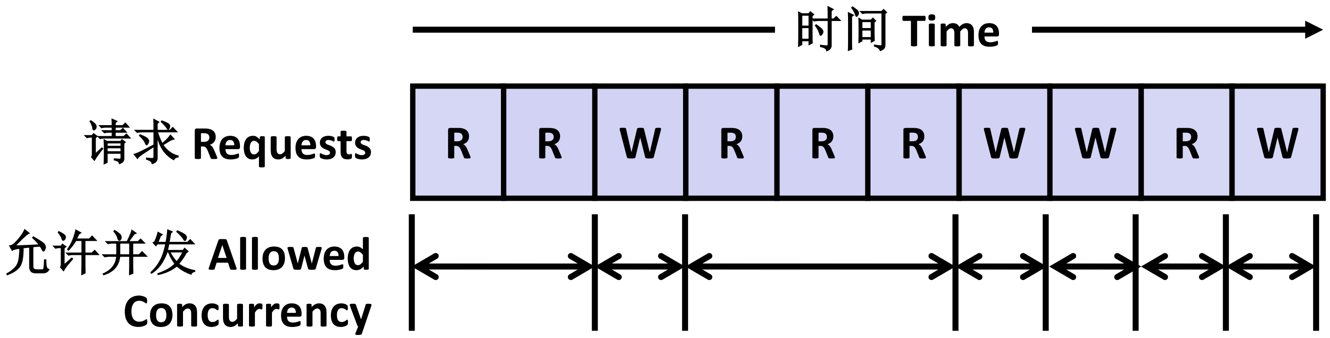
第二类读者-写者问题解决方案
Solution to Second Readers-Writers Problem

```
int readcnt, writecnt;        // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
  while (1) {
    P(&r);
    P(&rmutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&rmutex);
    V(&r)

    /* Reading happens here */

    P(&rmutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&rmutex);
  }
}
```

第二类读者-写者问题解决方案
Solution to Second Readers-Writers Problem

```
void writer(void)
{
  while (1) {
    P(&wmutex);
    writecnt++;
    if (writecnt == 1)
        P(&r);
    V(&wmutex);

    P(&w);
    /* Writing here */
    V(&w);

    P(&wmutex);
    writecnt--;
    if (writecnt == 0);
        V(&r);
    V(&wmutex);
  }
}
```

用先进先出队列管理读者/写者
Managing Readers/Writers with FIFO



- **思想 Idea**
  - 读/写请求插入先进先出队列 Read & Write requests are inserted into FIFO
  - 请求在从队列删除时进行处理 Requests handled as remove from FIFO
    - 如果当前空闲或正在处理读取，则允许继续读取 Read allowed to proceed if currently idle or processing read
    - 仅允许在空闲时继续写入请求 Write allowed to proceed only when idle
  - 请求完成后通知控制器 Requests inform controller when they have completed
- **公平 Fairness**
  - 保证最终会处理每个请求 Guarantee every request is eventually handled

读者写者先进先出实现
Readers Writers FIFO Implementation

- 完整代码见rwqueue.h和rwqueue.c Full code in rwqueue.{h,c}

```
/* Queue data structure */
typedef struct {
    sem_t mutex;  // Mutual exclusion
    int reading_count;    // Number of active readers
    int writing_count;    // Number of active writers
    // FIFO queue implemented as linked list with tail
    rw_token_t *head;
    rw_token_t *tail;
} rw_queue_t;
```

```
/* Represents individual thread's position in queue */
typedef struct TOK {
    bool is_reader;
    sem_t enable;         // Enables access
    struct TOK *next;     // Allows chaining as linked list
} rw_token_t;
```

读者写者先进先出队列使用
Readers Writers FIFO Use

- 在rwqueue-test.c程序中 In rwqueue-test.c

```
/* Get write access to data and write */
void iwriter(int *buf, int v)
{
    rw_token_t tok;
    rw_queue_request_write(&q, &tok);
    /* Critical section */
    *buf = v;
    /* End of Critical Section  */
    rw_queue_release(&q);
}
```

```
/* Get read access to data and read */
int ireader(int *buf)
{
    rw_token_t tok;
    rw_queue_request_read(&q, &tok);
    /* Critical section */
    int v = *buf;
    /* End of Critical section */
    rw_queue_release(&q);
    return v;
}
```

读者/写者锁的库函数
Library Reader/Writer Lock

- 数据类型 **Data type `pthread_rwlock_t`**
- 操作 **Operations**
  - 获得读锁 Acquire read lock

```
Pthread_rwlock_rdlock(pthread_rw_lock_t *rwlock)
```

  - 获得写锁 Acquire write lock

```
Pthread_rwlock_wrlock(pthread_rw_lock_t *rwlock)
```

  - 释放（其中一个）锁 Release (either) lock

```
Pthread_rwlock_unlock(pthread_rw_lock_t *rwlock)
```

- 观察 **Observation**
  - 必须正确使用库函数 Library must be used correctly!
    - 由程序员决定哪些需要读访问，哪些需要写访问 Up to programmer to decide what requires read access and what requires write access

议题 Today

- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 读者-写者问题 Readers-writers problem
- 其它并发问题 **Other concurrency issues**
  - **竞争 Races**
  - 死锁 Deadlocks
  - 线程安全 Thread safety
  - 线程和信号处理之间交互 Interactions between threads and signal handling

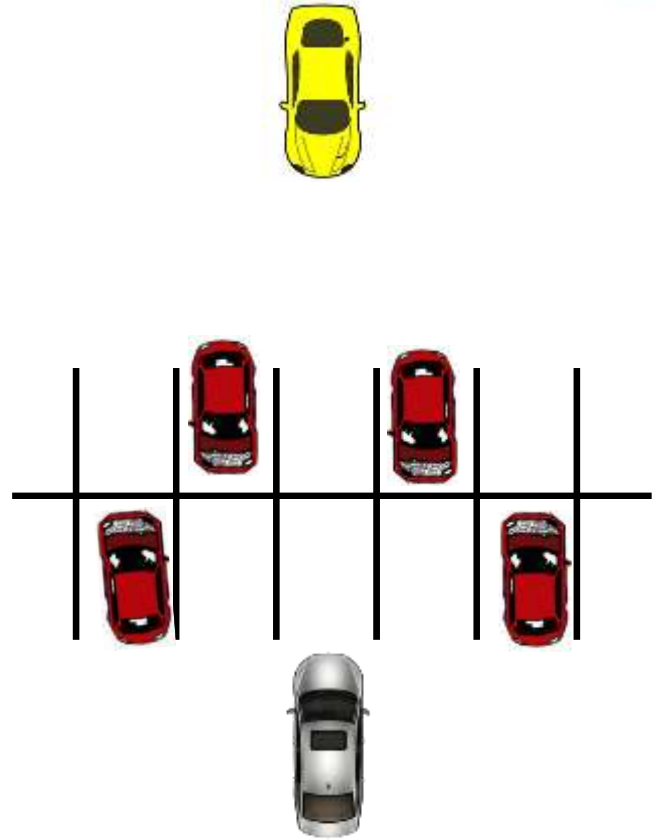
一个担忧：竞争 One Worry: Races

- 当程序的正确性取决于一个线程在另一个线程到达点y之前到达点x时，就会发生*竞争*　A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
       Pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

race.c

数据竞争 Data Race



消除竞争 Race Elimination

- 不要共享状态 **Don’t share state**
  - 例如，使用malloc为每个线程生成单独的参数拷贝 E.g., use malloc to generate separate copy of argument for each thread

- 使用同步原语控制对共享状态的访问 **Use synchronization primitives to control access to shared state**
  - 不同的共享状态可能使用不同的原语 Different shared state can use different primitives

议题 Today

- 回顾：信号量、互斥和生产者-消费者 **Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 生产者-消费者问题 Producer-consumer problem
- **其它并发问题 Other concurrency issues**
  - **竞争 Races**
  - **死锁 Deadlocks**
  - 线程安全 Thread safety
  - 线程和信号处理之间交互 Interactions between threads and signal handling

一个担忧：死锁 A Worry: Deadlock

- 定义：当且仅当一个进程正在等待一个永远不会为真的条件，那么它就会*死锁* **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true.**

- 典型场景 **Typical Scenario**
  - 进程1和进程2需要两个资源（A和B）才能继续 Processes 1 and 2 needs two resources (A and B) to proceed
  - 进程1获取A，等待B Process 1 acquires A, waits for B
  - 进程2获取B，等待A Process 2 acquires B, waits for A
  - 两个进程都将永远等待！ Both will wait forever!

一个担忧：死锁 A Worry: Deadlock

- 定义：当且仅当一个进程正在等待一个永远不会为真的条件，那么它就会*死锁*　**Def: A process is *deadlocked* iff it is waiting for a condition that will never be true.**

- 更全面的知识（超出了**213**课程的范围），死锁有四个要求 **More fully (and beyond the scope of 213), a deadlock has four requirements**
  - 互斥 Mutual exclusion
  - 循环等待 Circular waiting
  - 保持和等待 Hold and wait
  - 非抢占式 No pre-emption

信号量死锁 Deadlocking With Semaphores

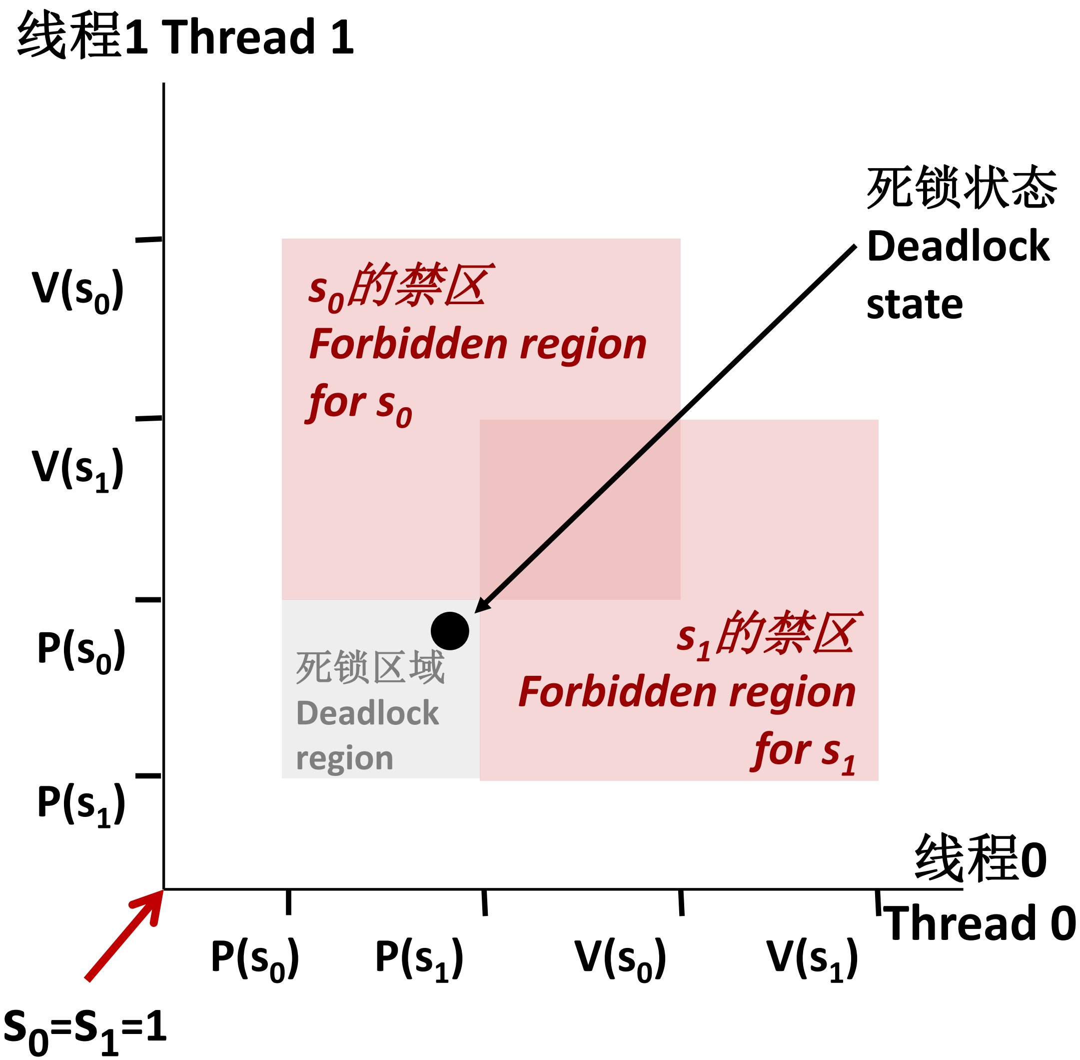
```
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    return 0;
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
$P(s_0)$;
$P(s_1)$;
cnt++;
$V(s_0)$;
$V(s_1)$;

Tid[1]:
$P(s_1)$;
$P(s_0)$;
cnt++;
$V(s_1)$;
$V(s_0)$;

进度图中显示的死锁
Deadlock Visualized in Progress Graph



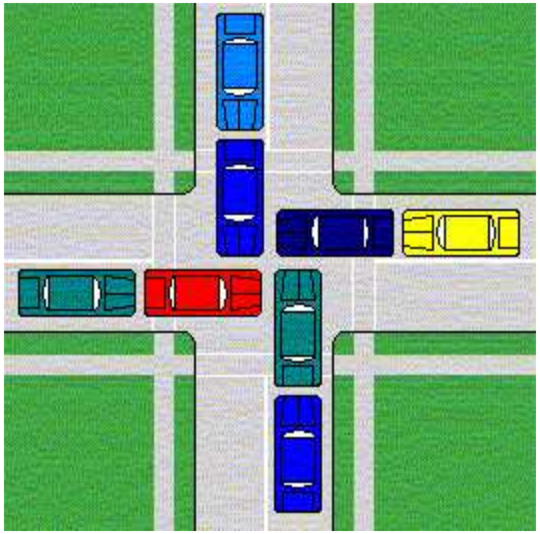
锁定引入了*死锁*的可能性：等待一个永远不会成真的条件 **Locking introduces the potential for *deadlock:* waiting for a condition that will never be true**

任何进入*死锁区域*的轨迹将最终到达*死锁状态*，等待s0或s1变为非零 **Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state,* waiting for either S_0 or S_1 to become nonzero**

其他轨迹幸运地避开了死锁区域 **Other trajectories luck out and skirt the deadlock region**

不幸的事实：死锁往往是不确定的（竞争） **Unfortunate fact: deadlock is often nondeterministic (race)**

死锁 Deadlock



避免死锁
Avoiding Deadlock

以相同的顺序获取共享资源

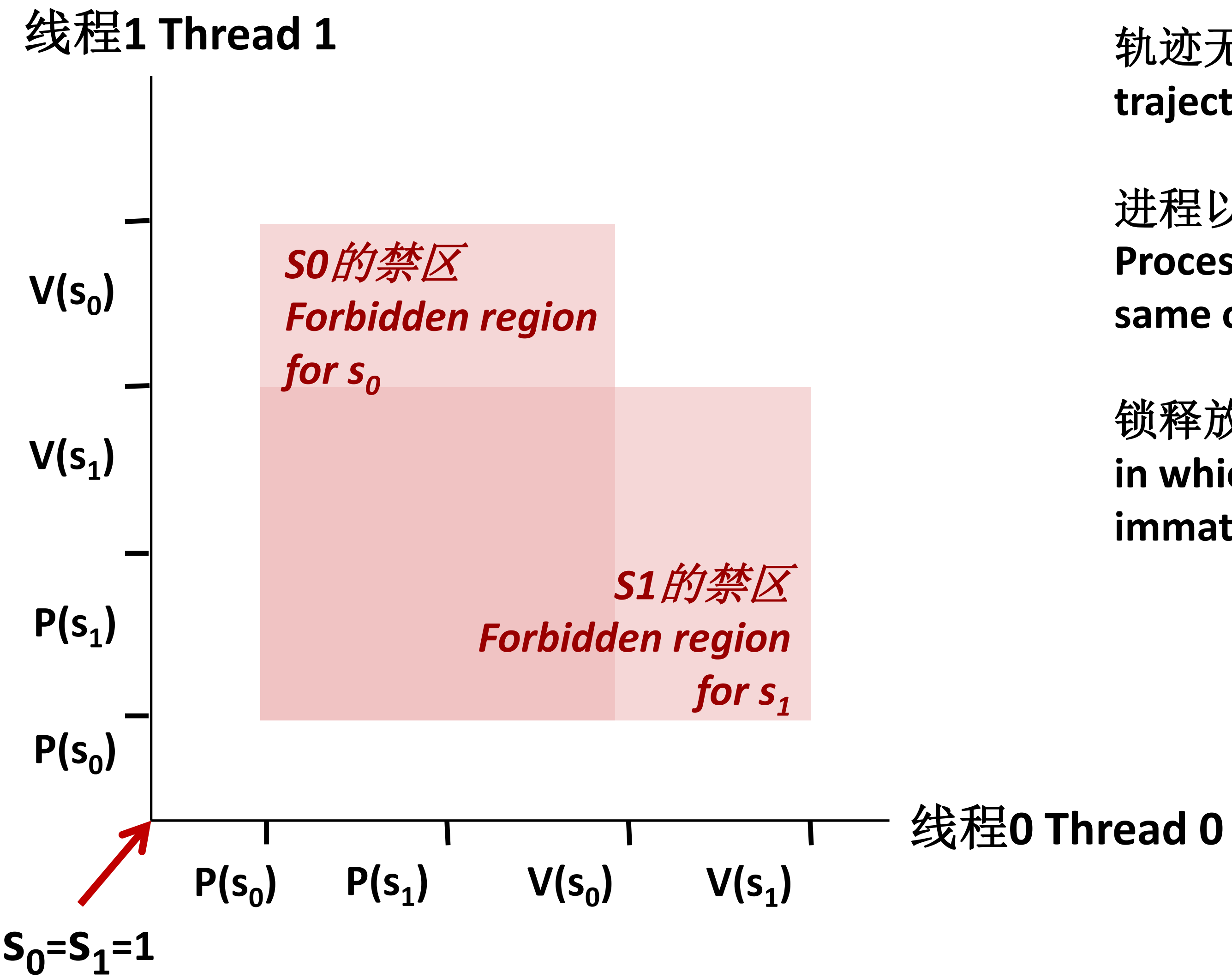
Acquire shared resources in same order

```
int main(int argc, char** argv)
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    return 0;
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:	Tid[1]:
P(s_0);	P(s_0);
P(s_1);	P(s_1);
cnt++;	cnt++;
V(s_0);	V(s_1);
V(s_1);	V(s_0);

在进度图中避免死锁
Avoided Deadlock in Progress Graph



轨迹无法卡住 No way for trajectory to get stuck

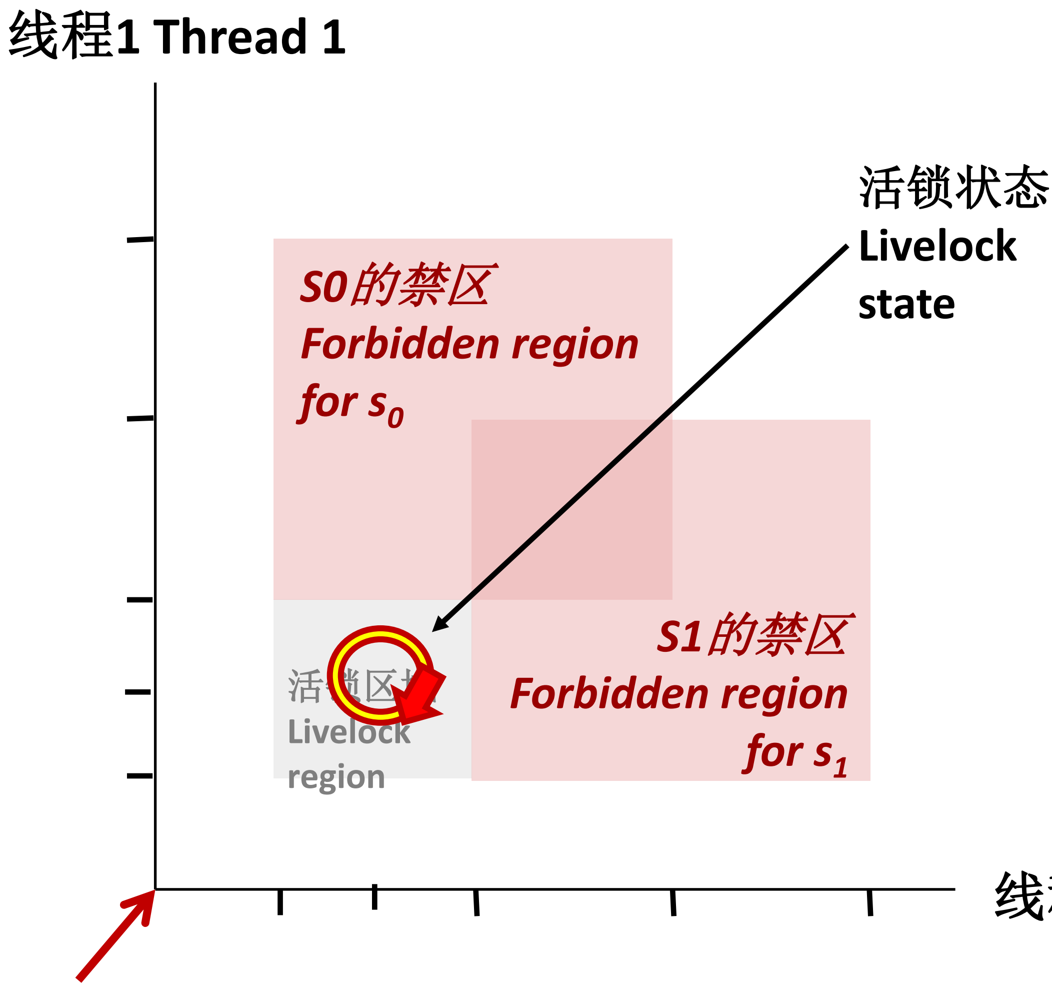
进程以相同的顺序获取锁 Processes acquire locks in same order

锁释放的顺序无关紧要 Order in which locks released immaterial

演示 Demonstration

- 参见程序deadlock.c   See program deadlock.c
- 100个线程，每个线程获得同样的两个锁   100 threads, each acquiring same two locks
- 风险模式 Risky mode
  - 偶数线程请求锁的顺序与奇数线程相反 Even numbered threads request locks in opposite order of odd-numbered ones
- 安全模式 Safe mode
  - 所有线程以相同的顺序获取锁 All threads acquire locks in same order

在进度图中显示活锁
Livelock Visualized in Progress Graph



活锁类似于死锁，只是线程改变状态，但仍处于死锁轨迹中

Livelock is similar to a deadlock, except the threads change state, but remain in a deadlock trajectory.

死锁、活锁、饿死
Deadlock, Livelock, Starvation

- **死锁 Deadlock**
  - 一个或多个线程正在等待一个永远不会为真的条件 One or more threads is waiting on a condition that will never be true

- **活锁 Livelock**
  - 一个或多个线程正在更改状态，但永远不会离开死锁/活锁轨迹 One or more threads is changing state, but will never leave a deadlock / livelock trajectory

- **饿死 Starvation**
  - 一个或多个线程暂时无法取得进展 One or more threads is temporarily unable to make progress

议题 Today

- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 读者-写者问题 Readers-writers problem
- **其它并发问题 Other concurrency issues**
  - **竞争 Races**
  - **死锁 Deadlocks**
  - **线程安全 Thread safety**
  - 线程和信号处理之间交互 Interactions between threads and signal handling

关键概念：线程安全
Crucial concept: Thread Safety

- 从线程调用的函数必须是线程安全的 **Functions called from a thread must be *thread-safe***
- *定义*：函数是线程安全的，只要它在从多个线程同时调用时总是产生正确的结果 ***Def:* A function is *thread-safe* iff it will always produce correct results when called simultaneously from multiple threads.**
- 线程不安全函数的分类： **Classes of thread-unsafe functions:**
  - 类1：不保护共享变量的函数 Class 1: Functions that do not protect shared variables
  - 类2：跨多个调用保持状态的函数 Class 2: Functions that keep state across multiple invocations
  - 类3：返回指向静态变量的指针的函数 Class 3: Functions that return a pointer to a static variable
  - 类4：调用线程不安全函数的函数 Class 4: Functions that call thread-unsafe functions

线程不安全函数（类1）
Thread-Unsafe Functions (Class 1)

- 未能保护共享变量 **Failing to protect shared variables**
  - 修复：使用P和V信号量操作（或互斥锁） Fix: Use *P* and *V* semaphore operations (or mutex)
  - 示例：goodcnt.c Example: `goodcnt.c`
  - 问题：同步操作会降低代码速度 Issue: Synchronization operations will slow down code

线程不安全函数（类2）
Thread-Unsafe Functions (Class 2)

- 跨多个函数调用依赖持久状态 **Relying on persistent state across multiple function invocations**
  - 示例：依赖于静态状态的随机数生成器 Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

线程安全随机数生成器
Thread-Safe Random Number Generator

- 传递状态作为参数的一部分 **Pass state as part of argument**
  - 从而消除静态状态 and, thereby, eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- 结论：使用rand_r的程序员必须保持种子 **Consequence: programmer using `rand_r` must maintain seed**

线程不安全函数（类3） Thread-Unsafe Functions (Class 3)

- 返回指向静态变量的指针 **Returning a pointer to a static variable**
- 修复：重写函数，以便调用方传递变量地址以存储结果 **Fix: Rewrite function so caller passes address of variable to store result**
  - 需要更改调用者和被调用者 Requires changes in caller and callee
- 修复2：用互斥锁包装函数 **Fix 2: Wrap function with mutex**
  - 调用方仍需更改 Caller still has to be changed
  - 可以保留旧函数 Can preserve old function
  - 函数可能成为瓶颈 Function may become a bottleneck

```
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    snprintf(buf, sizeof buf,
             "%d", x);
    return buf;
}
```

```
void fix_itoa(int x, char *dst,
              size_t dstsz)
{
    snprintf(dst, dstsz, "%d", x);
}
```

```
void wrap_itoa(int x, char *dst,
              size_t dstsz)
{
    static sem_t mutex;
    P(mutex);
    strncpy(dst, itoa(x), dstsz);
    V(mutex);
}
```

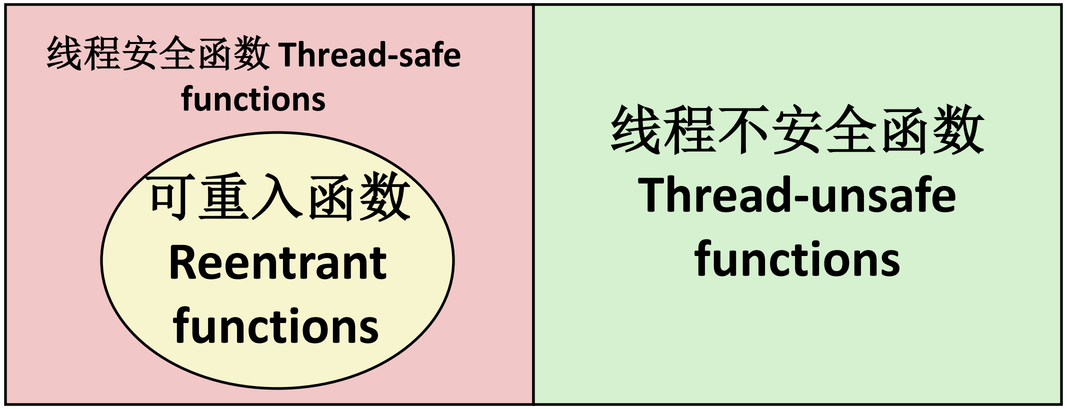
线程不安全函数（类4）Thread-Unsafe Functions (Class 4)

- 调用线程不安全函数 **Calling thread-unsafe functions**
  - 调用一个线程不安全函数会使调用它的整个函数不安全 Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - 修复：修改函数，使其仅调用线程安全函数 Fix: Modify the function so it calls only thread-safe functions ☺

可重入函数 Reentrant Functions

- 定义：当且仅当函数被多个线程调用时不访问共享变量，则该函数是*可重入*的 **Def: A function is *reentrant* iff it accesses no shared variables when called by multiple threads.**
  - 线程安全函数的重要子集 Important subset of thread-safe functions
    - 不需要同步操作 Require no synchronization operations
    - 使类2函数线程安全的唯一方法是使其可重入（例如rand_r）Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

全部函数 **All functions**



线程安全的库函数
Thread-Safe Library Functions

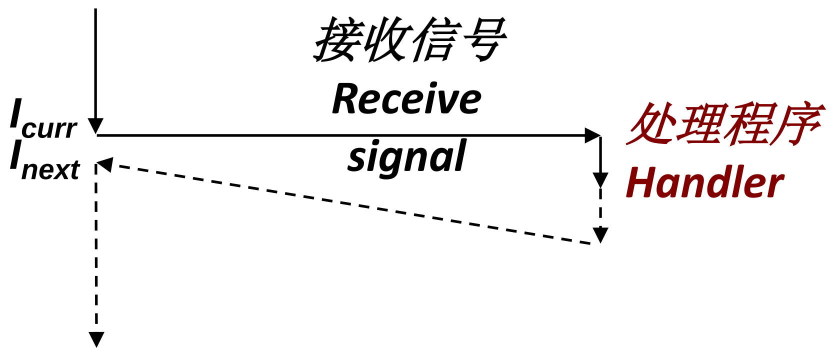
- 标准C语言库（K&R教材后面）中的大多数函数都是线程安全的 **Most functions in the Standard C Library (at the back of your K&R text) are thread-safe**
  - 示例：malloc、free、printf、scanf Examples: **`malloc`, `free`, `printf`, `scanf`**
  - 例外：strtok、rand、ctime Exceptions: **`strtok`, `rand`, `ctime`**
- **POSIX添加了更多的异常，但也添加了不安全函数的可重入版本 POSIX adds more exceptions, but also reentrant versions of unsafe functions**

线程不安全函数 Thread-unsafe function	Class	可重入版 Reentrant version
`asctime`	3	`strftime`
`ctime`	3	`strftime`
`gethostbyaddr`	3	`getnameinfo`
`gethostbyname`	3	`getaddrinfo`
`inet_ntoa`	3	`getnameinfo`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

议题 Today

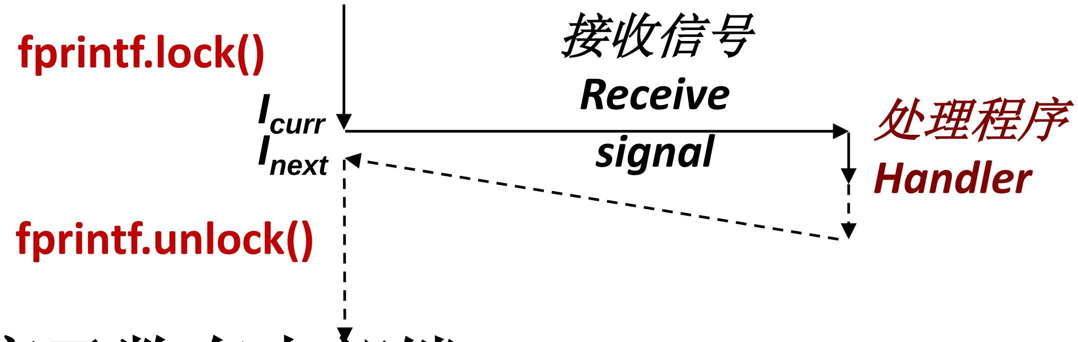
- 回顾：信号量、互斥和生产者-消费者 **Review: Semaphores, mutexes, producer-consumer**
- 使用信号量调度共享资源 **Using semaphores to schedule shared resources**
  - 读者-写者问题 Readers-writers problem
- **其它并发问题 Other concurrency issues**
  - **竞争 Races**
  - **死锁 Deadlocks**
  - **线程安全 Thread safety**
  - **线程和信号处理之间交互 Interactions between threads and signal handling**

信号处理回顾 Signal Handling Review



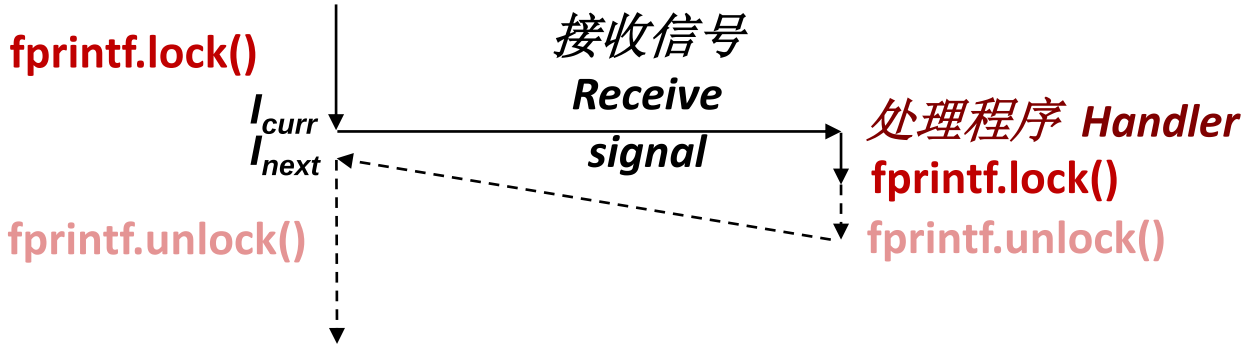
- 动作 Action
  - 信号可以发生在程序执行的任何点 Signal can occur at any point in program execution
    - 除非信号被阻塞 Unless signal is blocked
  - 信号处理程序在同一个线程内运行 Signal handler runs within same thread
  - 必须运行到完成，然后返回到正常的程序执行 Must run to completion and then return to regular program execution

线程/信号交互
Threads / Signals Interactions



- 很多库函数有内部锁 **Many library functions have internal locking**
  - 为了保护隐藏状态（避免第一类线程不安全） To protect hidden state (avoid being class 1 thread-unsafe)
  - malloc
    - 释放列表 Free lists
  - fputs, fprintf, snprintf
    - 以便从多个线程的输出不会交错 So that outputs from multiple threads don’t interleave
    - 内部使用malloc Internal use of malloc
- 不使用这些库函数的处理程序没有问题 **OK for handler that doesn’t use these library functions**

有问题的线程/信号交互
Bad Thread / Signal Interactions



- 如果以下情况会怎样：**What if:**
  - 当库函数保持加锁时接收信号 Signal received while library function holds lock
  - 处理程序调用同样（或相关）库函数 Handler calls same (or related) library function
- 死锁！ **Deadlock!**
  - 信号处理程序不能继续直到获得锁 Signal handler cannot proceed until it gets lock
  - 主程序不能继续直到处理程序完成 Main program cannot proceed until handler completes
- 关键点 **Key Point**
  - 线程采用对称并发 Threads employ symmetric concurrency
  - 信号处理是异步的 Signal handling is asymmetric

处理线程/信号交互
Handling Thread/Signal Interactions

- 临界区周围阻塞信号 **Block signals around critical sections**
  - pthread_sigmask函数类似sigprocmask，但是仅影响正调用的线程 pthread_sigmask – like sigprocmask, but only affects calling thread
- 专用于信号处理的线程 **Dedicate a thread to signal handling**
  - 循环调用sigsuspend()或sigwaitinfo() Loop calling sigsuspend() or sigwaitinfo()
  - 所有其他线程阻塞所有信号 All other threads block all signals
  - 信号处理线程可以使用异步信号不安全函数 Signal handling thread can use async-signal-unsafe functions
    - 因为我们知道信号只能在sigsuspend()期间传递 Because we know signals will only be delivered during sigsuspend()

线程小结 Threads Summary

- **线程为编写并发程序提供了另一种机制 Threads provide another mechanism for writing concurrent programs**
- **线程越来越受欢迎 Threads are growing in popularity**
  - 比进程开销小 Somewhat cheaper than processes
  - 易于在线程之间共享数据 Easy to share data between threads
- **然而，共享的便捷性有代价： However, the ease of sharing has a cost:**
  - 易于引入细微的同步错误 Easy to introduce subtle synchronization errors
  - 小心对待线程！ Tread carefully with threads!

- **有关详细信息： For more info:**
  - "Posix线程编程" D. Butenhof, "Programming with Posix Threads", Addison-Wesley, 1997

第12章 并发编程

线程级并行 Thread-Level Parallelism

100076202： 计算机系统导论

任课教师：

宿红毅 张艳 黎有琦 颜珂

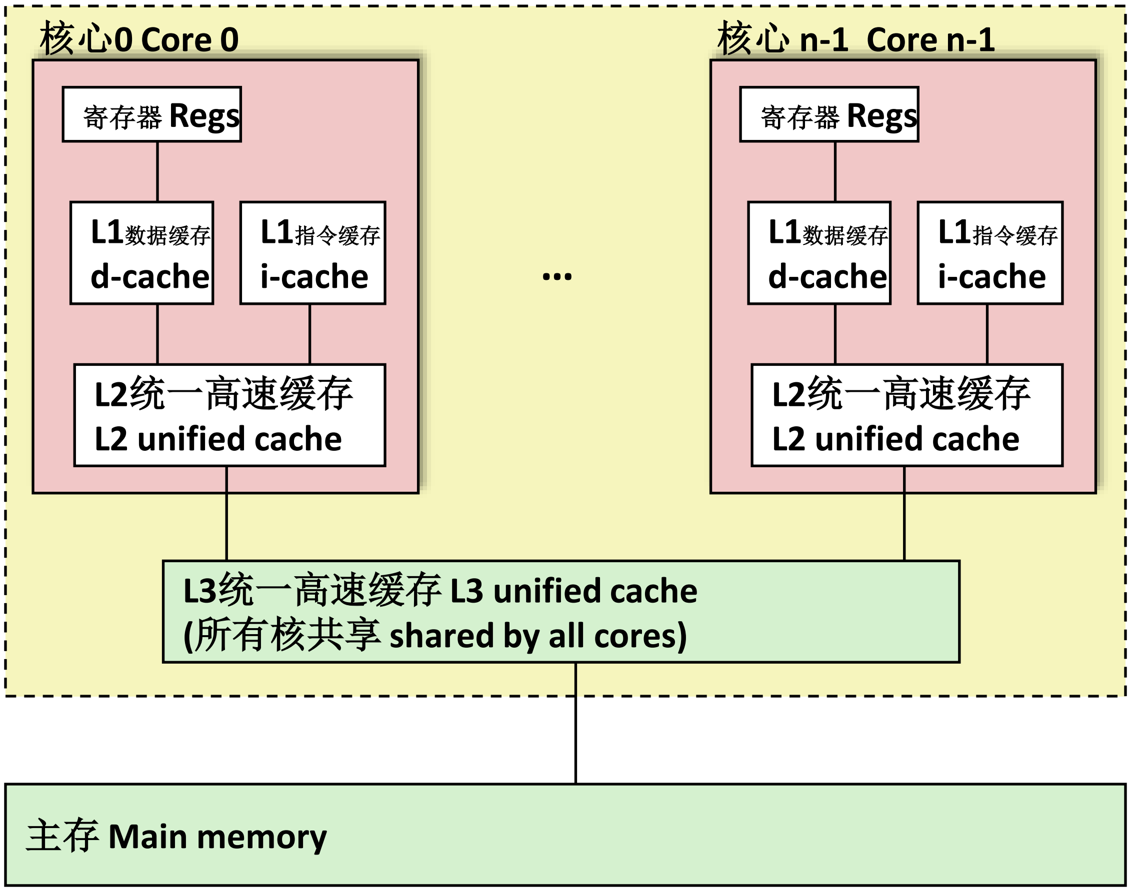
原作者：

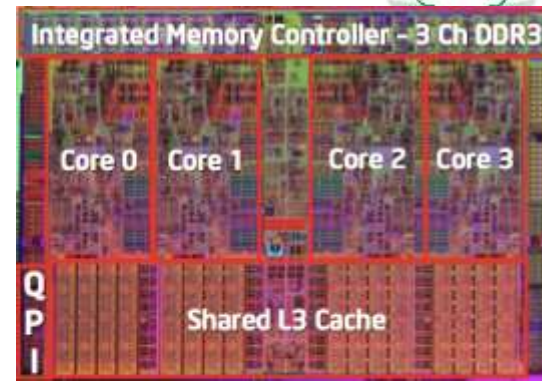
Randal E. **Bryant and** David R. O'Hallaron

议题 Today

- 并行计算硬件 **Parallel Computing Hardware**
  - 多核 Multicore
    - 单个芯片上有多个独立处理器 Multiple separate processors on single chip
  - 超线程化 Hyperthreading
    - 在单核上高效执行多个线程 Efficient execution of multiple threads on single core
- 一致性模型 **Consistency Models**
  - 当多个线程读取和写入共享状态时会发生什么 What happens when multiple threads are reading & writing shared state
- 线程级并行 **Thread-Level Parallelism**
  - 将程序拆分为独立任务 Splitting program into independent tasks
    - 示例：并行求和 Example: Parallel summation
    - 检查一些性能工件 Examine some performance artifacts
  - 分而治之 Divide-and conquer parallelism
    - 示例：并行快速排序 Example: Parallel quicksort

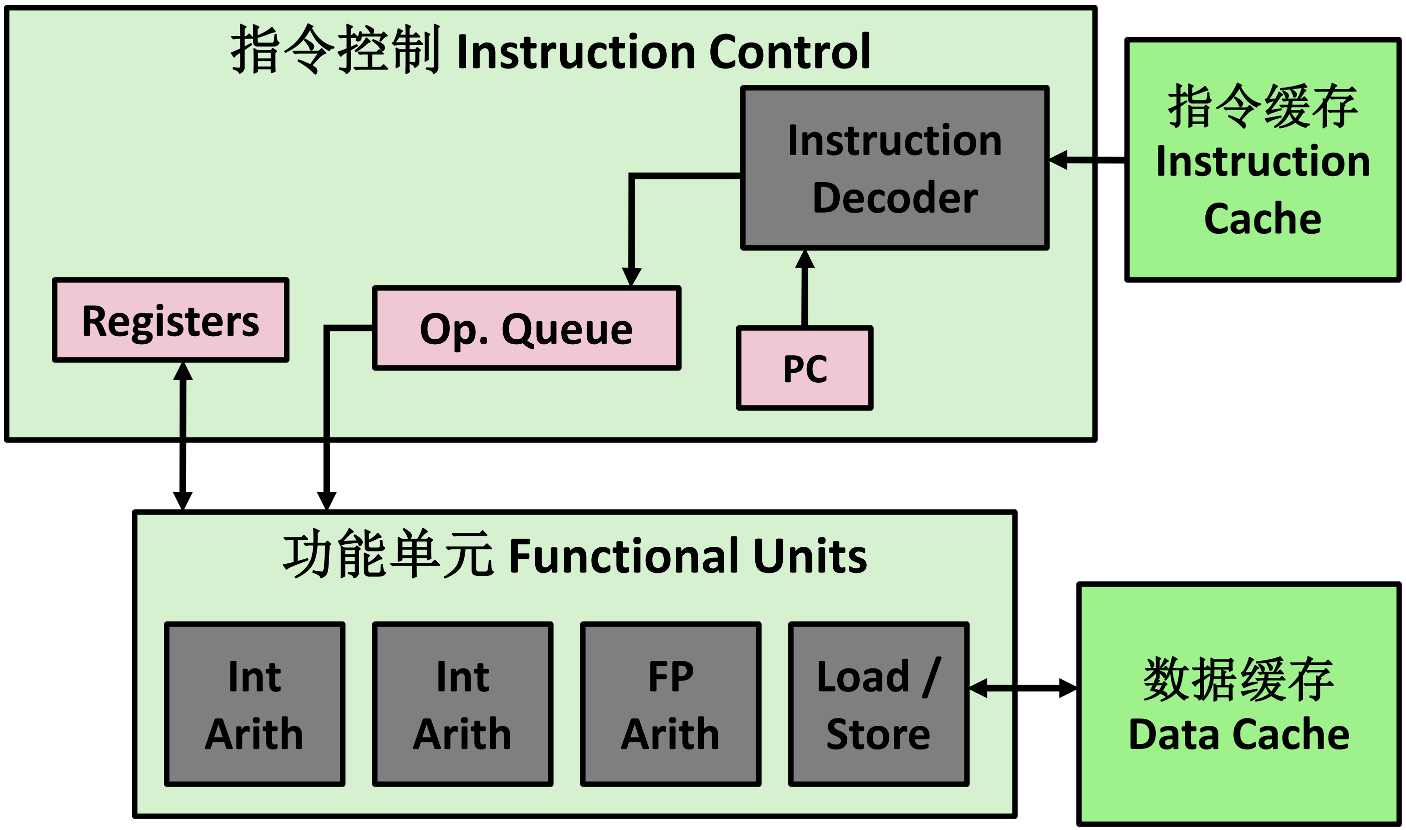
典型的多核处理器
Typical Multicore Processor





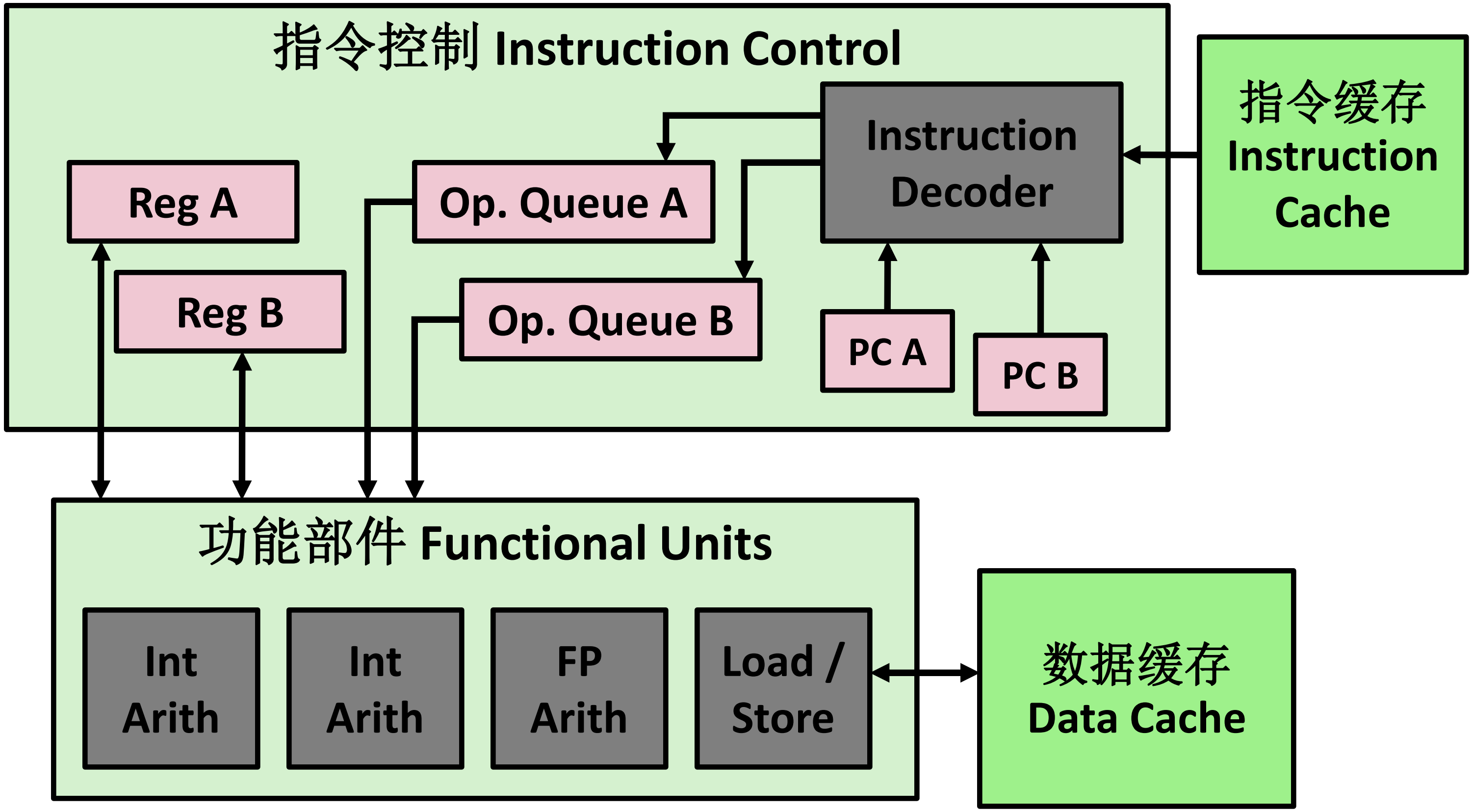
- 多个处理器以一致的内存视图运行 Multiple processors operating with coherent view of memory

乱序处理器结构
Out-of-Order Processor Structure



- 指令控制将程序动态转换为操作流 **Instruction control dynamically converts program into stream of operations**
- 操作映射到功能单元以并行方式执行 **Operations mapped onto functional units to execute in parallel**

超线程实现
Hyperthreading Implementation



- 复制指令控制以处理K个指令流 Replicate instruction control to process K instruction streams
- 所有寄存器有K份拷贝 K copies of all registers
- 共享功能单元 Share functional units

基准测试机 Benchmark Machine

- **从/proc/cpuinfo获取有关计算机的数据 Get data about machine from /proc/cpuinfo**
- **Shark机器 Shark Machines**
  - Intel Xeon E5520 @ 2.27 GHz
  - Nehalem, ca. 2010
  - 8核 8 Cores
  - 每个核心可以执行2倍超线程 Each can do 2x hyperthreading

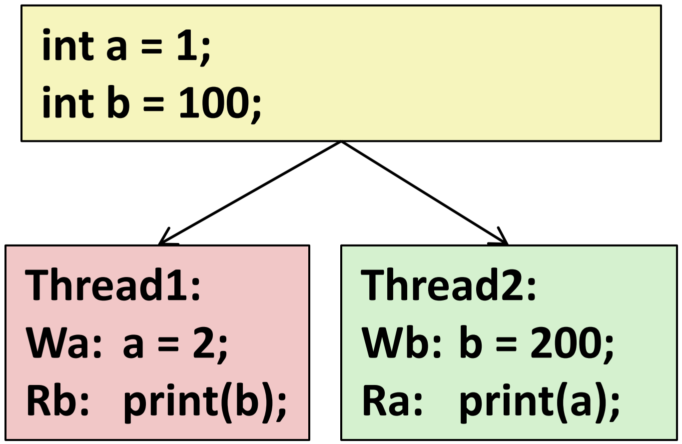
利用并行执行 Exploiting parallel execution

- **到目前为止，我们已经使用线程来处理I/O延迟 So far, we've used threads to deal with I/O delays**
  - 例如每个客户端一个线程，以防止一个线程延迟另一个线程 e.g., one thread per client to prevent one from delaying another
- **多核CPU提供了另一个机会 Multi-core CPUs offer another opportunity**
  - 在N个核心上并行执行的线程上扩展工作 Spread work over threads executing in parallel on N cores
  - 如果有许多独立任务，则自动发生 Happens automatically, if many independent tasks
    - 例如，运行许多应用程序或为许多客户端提供服务 e.g., running many applications or serving many clients
  - 还可以编写代码以加快一项大型任务的执行速度 Can also write code to make one big task go faster
    - 通过将其组织为多个并行子任务 by organizing it as multiple parallel sub-tasks

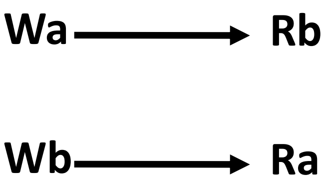
利用并行执行 Exploiting parallel execution

- **Shark机器可以同时执行16个线程 Shark machines can execute 16 threads at once**
  - 8核心，每个带2路超线程 8 cores, each with 2-way hyperthreading
  - 理论上16倍加速比 Theoretical speedup of 16X
    - 在我们的基准测试中从未达到 never achieved in our benchmarks

内存一致性 Memory Consistency



线程一致性约束
Thread consistency constraints



- 打印的可能值是什么？ **What are the possible values printed?**
  - 取决于内存一致性模型 Depends on memory consistency model
  - 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses

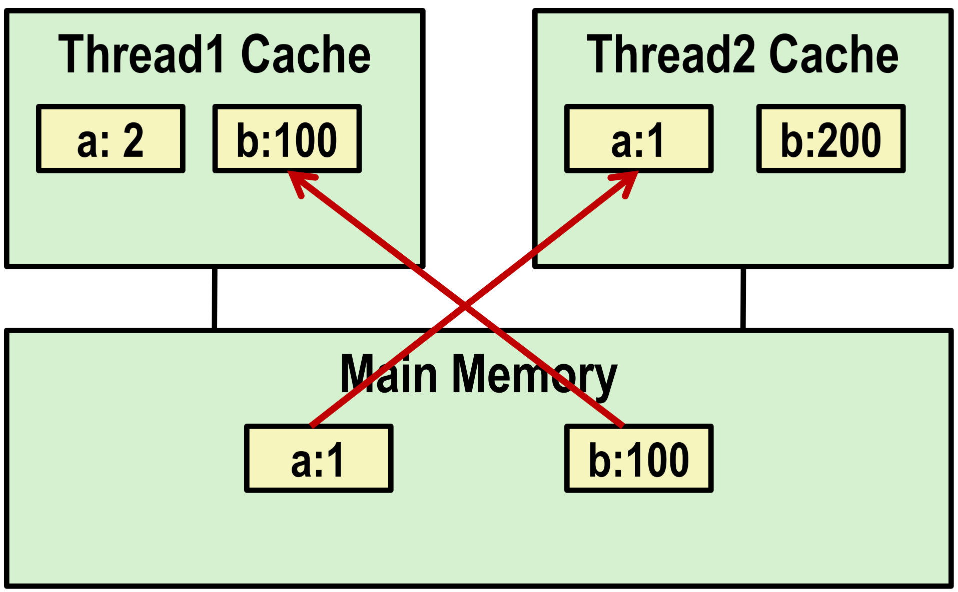
非一致性高速缓存方案
Non-Coherent Cache Scenario

- 写回高速缓存，线程间没有协作 **Write-back caches, without coordination between them**

```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:  print(a);
```



print 1

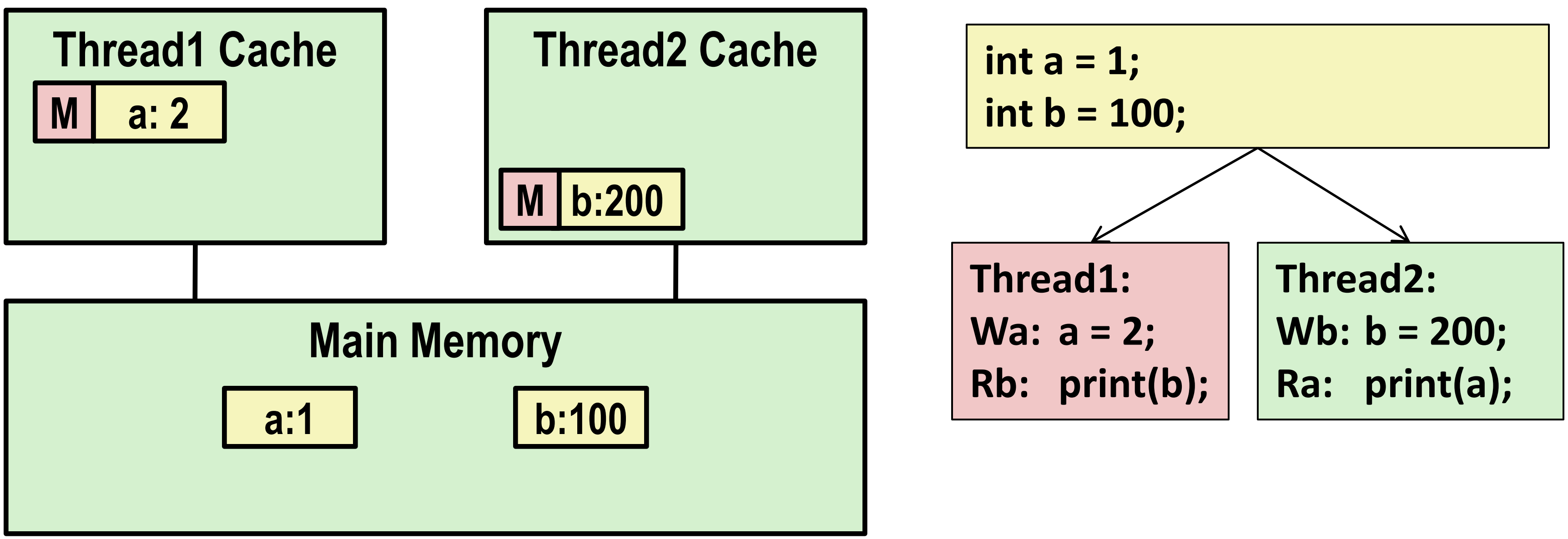
print 100

稍后，a:2和b:200被写回主存储器
At later points, a:2 and b:200 are written back to main memory

Snoopy缓存
Snoopy Caches

- 用状态标记每个缓存块 **Tag each cache block with state**

无效	Invalid	不能使用其值	Cannot use value
共享	Shared	可读拷贝	Readable copy
修改	Modified	可写拷贝	Writeable copy



Snoopy缓存
Snoopy Caches

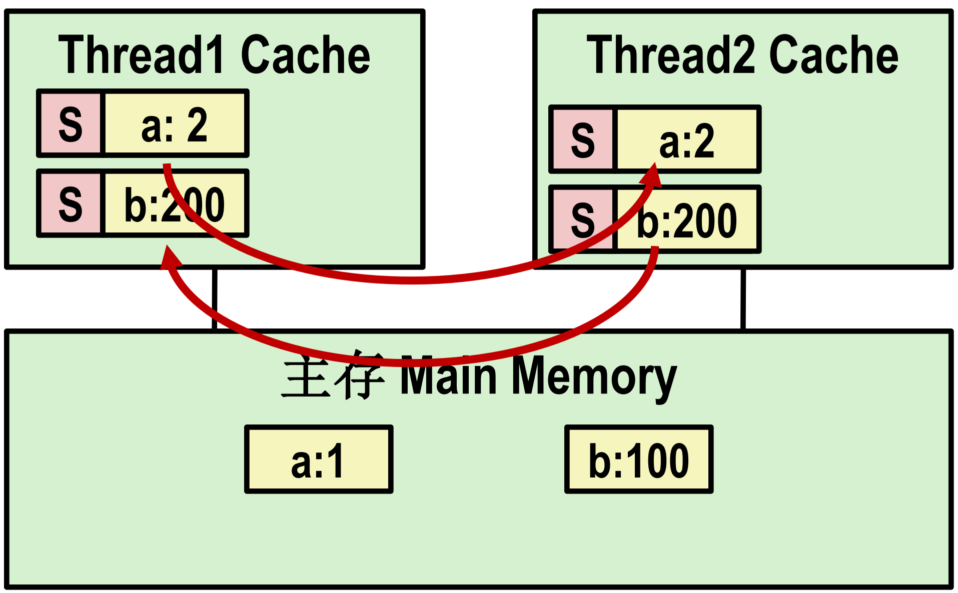
```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:  print(a);
```

- 用状态标记每个缓存块
- **Tag each cache block with state**

无效	Invalid	不能使用其值 Cannot use value
共享	Shared	可读拷贝 Readable copy
修改	Modified	可写拷贝 Writeable copy



print 2

print 200

- 当缓存看到对其M标记块之一的请求时 When cache sees request for one of its M-tagged blocks
  - 从缓存提供值（注意：内存中的值可能已过时） Supply value from cache (Note: value in memory may be stale)
  - 将标记设置为S Set tag to S

内存一致性 Memory Consistency

```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:   print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:   print(a);
```

线程一致性约束
Thread consistency constraints

Wa → Rb

Wb → Ra

- 打印的可能值是什么？ **What are the possible values printed?**
  - 取决于内存一致性模型 Depends on memory consistency model
  - 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses

内存一致性 Memory Consistency

```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:   print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:   print(a);
```

线程一致性约束
Thread consistency constraints

Wa → Rb

Wb → Ra

- **打印的可能值是什么？ What are the possible values printed?**
  - 取决于内存一致性模型 Depends on memory consistency model
  - 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses
- **顺序一致性 Sequential consistency**
  - 就好像一次只有一个操作一样，其顺序与每个线程内的操作顺序一致 As if only one operation at a time, in an order consistent with the order of operations within each thread
  - 因此，总体效果与每个单独的线程一致，但允许任意交错 Thus, overall effect consistent with each individual thread but otherwise allows an arbitrary interleaving

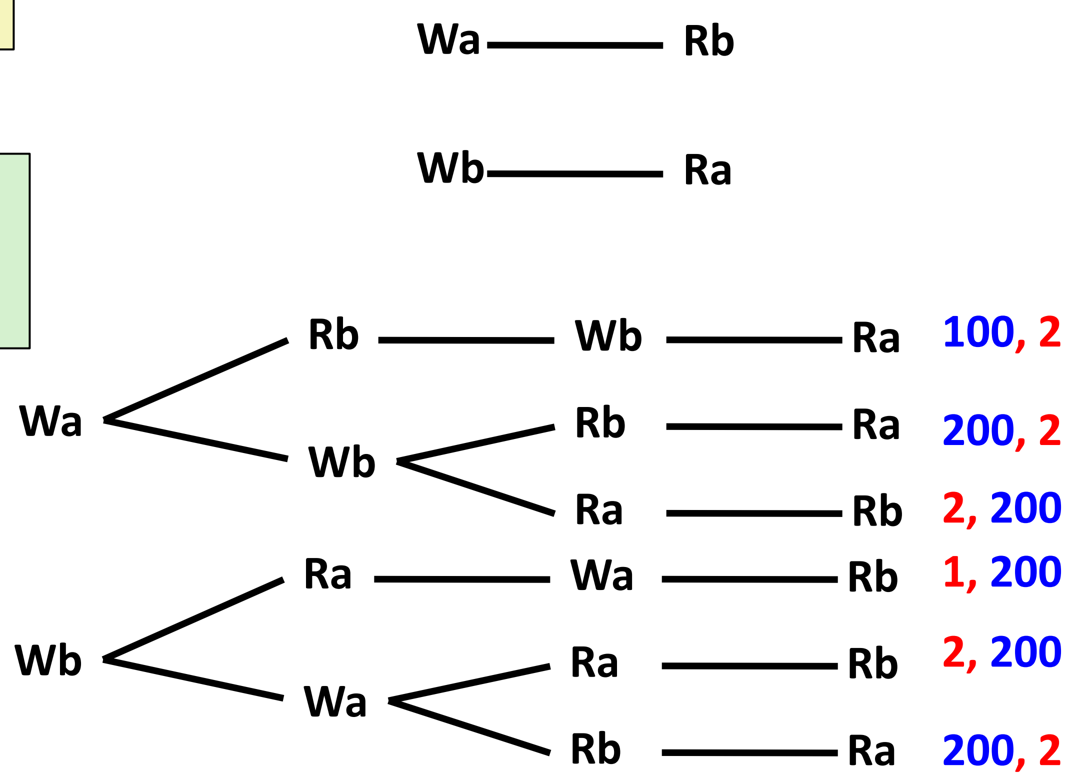
顺序一致性示例
Sequential Consistency Example

```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:  print(a);
```

线程一致性约束 Thread consistency constraints



- 不可能输出 **Impossible outputs**
  - 100, 1 and 1, 100
  - 需要在Wa或Wb之前达到Ra和Rb Would require reaching *both* Ra and Rb before *either* Wa or Wb

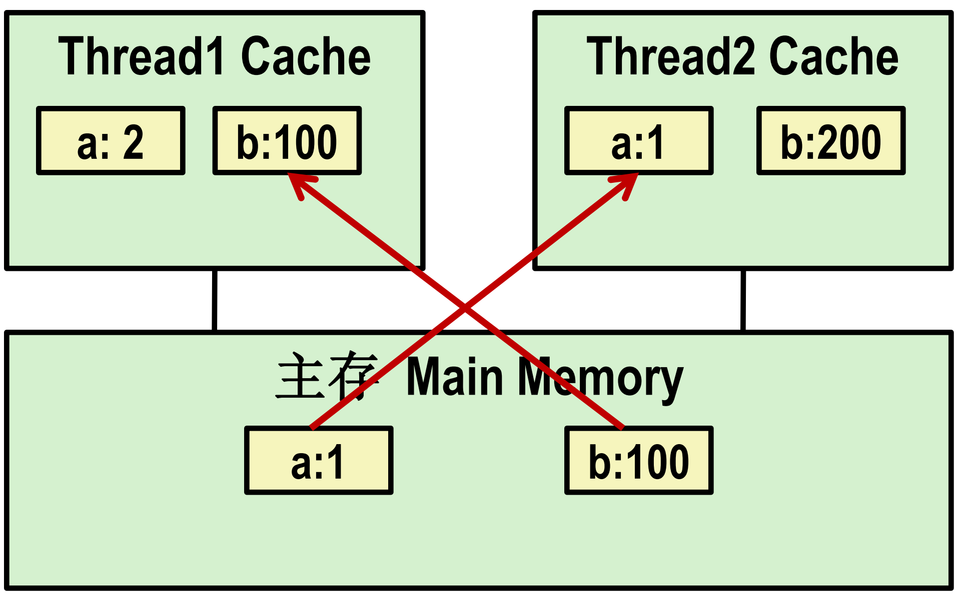
非一致性缓存方案
Non-Coherent Cache Scenario

- 写回缓存，线程间没有协作
  Write-back caches, without coordination between them

```
int a = 1;
int b = 100;
```

```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

```
Thread2:
Wb:  b = 200;
Ra:  print(a);
```



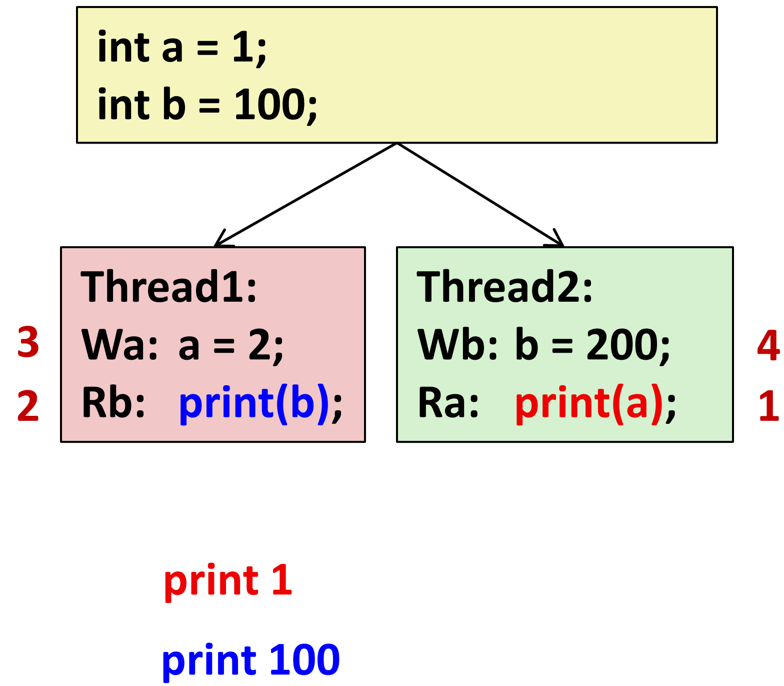
print 1

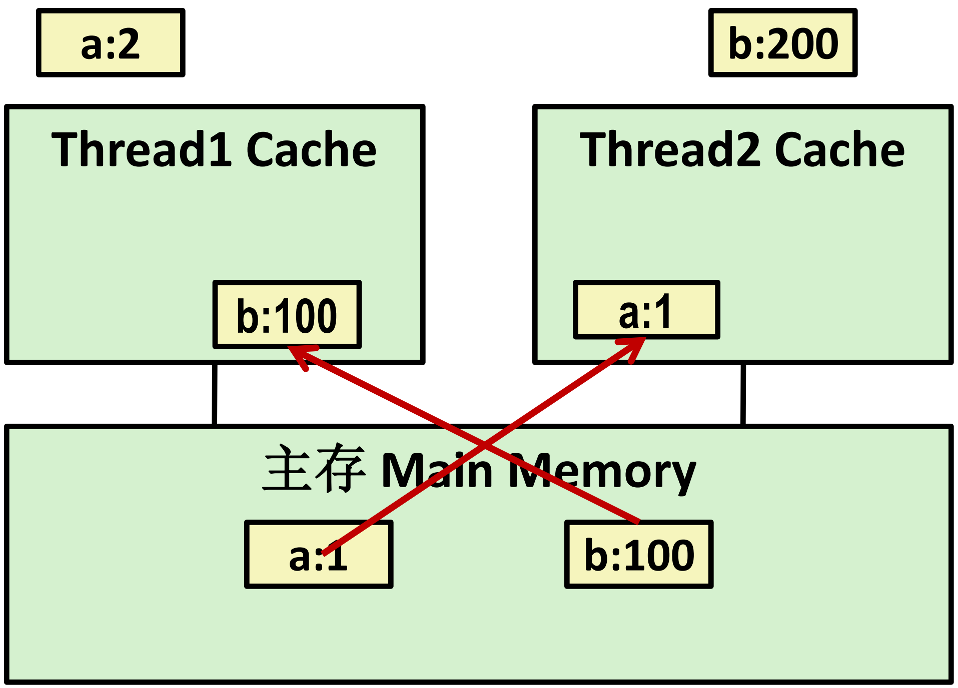
print 100

顺序一致性？ 否！
Sequentially consistent? No!

非顺序一致性方案
Non-Sequentially Consistent Scenario

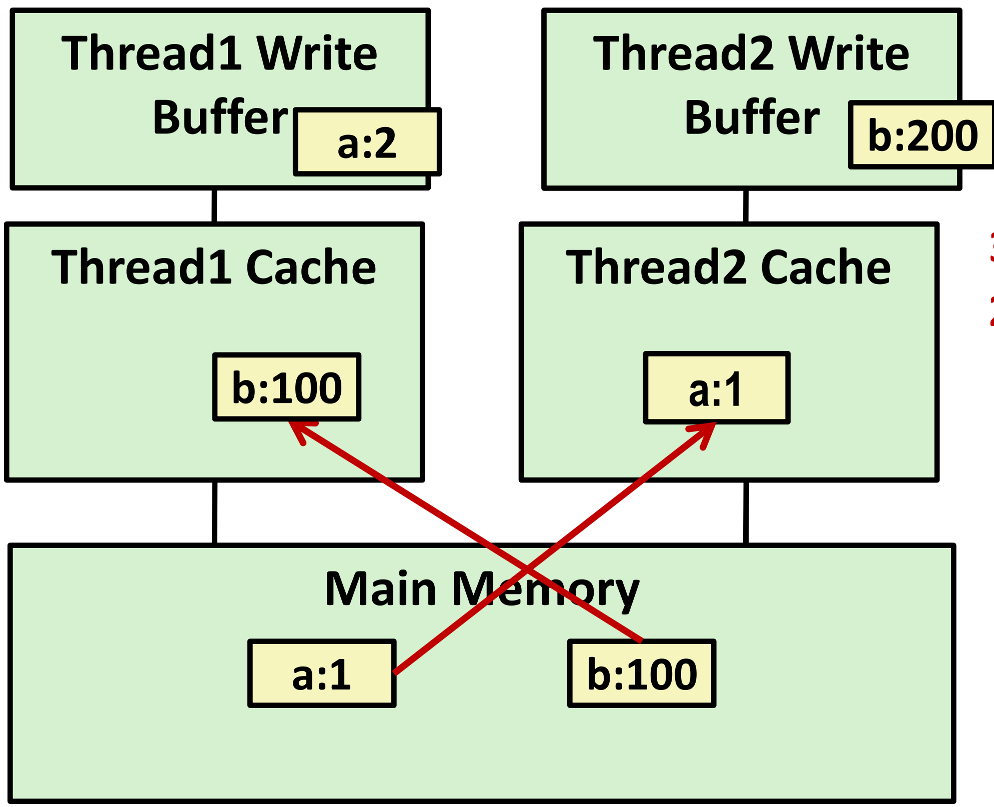
- 一致性缓存，但由于操作重新排序而违反了线程一致性约束 Coherent caches, but thread consistency constraints violated due to *operation reordering*

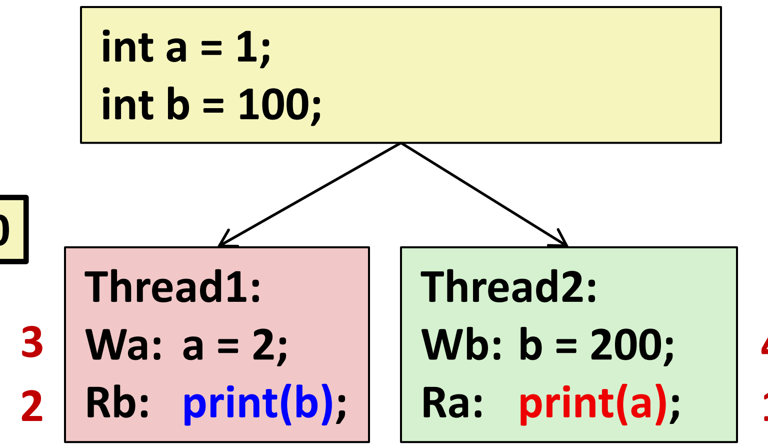




- 体系结构允许读取在写入之前完成，因为单个线程访问不同的内存位置 Architecture lets reads finish before writes because single thread accesses different memory locations

非顺序一致性方案
Non-Sequentially Consistent Scenario





- 为什么重新排序？写入需要很长时间。缓冲区写入，让读取继续。*指令级并行性* **Why Reordered? Writes take long time. Buffer write, let read go ahead. *Instruction-level parallelism***

- **修复：在Wa&Rb和Wb&Ra之间添加SFENCE指令 Fix: Add** SFENCE **instructions between Wa & Rb and Wb & Ra**

内存模型 Memory Models

- 顺序一致： **Sequentially Consistent:**
  - 每个线程以正确的顺序执行，任意交错 Each thread executes in proper order, any interleaving

- 为了确保，需要 **To ensure, requires**
  - 正确的缓存/内存行为 Proper cache/memory behavior
  - 适当的线程内排序约束 Proper intra-thread ordering constraints

- 线程排序约束 **Thread ordering constraints**
  - 使用同步确保程序没有数据竞争 Use synchronization to ensure the program is free of data races

议题 Today

- 并行计算硬件 Parallel Computing Hardware
  - 多核 Multicore
    - 单芯片上有多个独立的处理器 Multiple separate processors on single chip
  - 超线程化 Hyperthreading
    - 在单核上高效执行多个线程 Efficient execution of multiple threads on single core
- 一致性模型 Consistency Models
  - 当多个线程在读/写共享状态时会发生什么情况 What happens when multiple threads are reading & writing shared state
- 线程级并行 Thread-Level Parallelism
  - 将程序分成独立的任务 Splitting program into independent tasks
    - 例如：并行求和 Example: Parallel summation
    - 检查一些性能小工件 Examine some performance artifacts
  - 分而治之 Divide-and conquer parallelism
    - 例如：并行快速排序 Example: Parallel quicksort

求和示例 Summation Example

- **求数字0，…，N-1的和 Sum numbers 0, …, N-1**
  - 应该加起来得到(N-1)*N/2 Should add up to (N-1)*N/2
- **分区成K个区域 Partition into K ranges**
  - 每个区域有⌊N/K⌋个值 ⌊N/K⌋ values each
  - t个线程每个处理一个区域 Each of the *t* threads processes 1 range
  - 连续累加剩余值 Accumulate leftover values serially
- **方法#1：所有线程更新单个全局变量 Method #1: All threads update single global variable**
  - 1A：无同步 1A: No synchronization
  - 1B：用pthread信号量同步 1B: Synchronize with pthread semaphore
  - 1C：用pthread互斥锁同步 1C: Synchronize with pthread mutex
    - "二元"信号量，仅取值0和1 "Binary" semaphore. Only values 0 & 1

累积在单个全局变量中：声明
Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;
```

累积在单个全局变量中：声明
Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;
```

累积在单个全局变量中：声明
Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;

/* Number of elements summed by each thread */
size_t nelems_per_thread;

/* Keep track of thread IDs */
pthread_t tid[MAXTHREADS];

/* Identify each thread */
int myid[MAXTHREADS];
```

累积在单个全局变量中：操作
Accumulating in Single Global Variable: Operation

```
    nelems_per_thread = nelems / nthreads;

    /* Set global value */
    global_sum = 0;

    /* Create threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    result = global_sum;

    /* Add leftover elements */
    for (e = nthreads * nelems_per_thread; e < nelems; e++)
         result += e;
```

线程ID Thread ID

线程例程 Thread routine

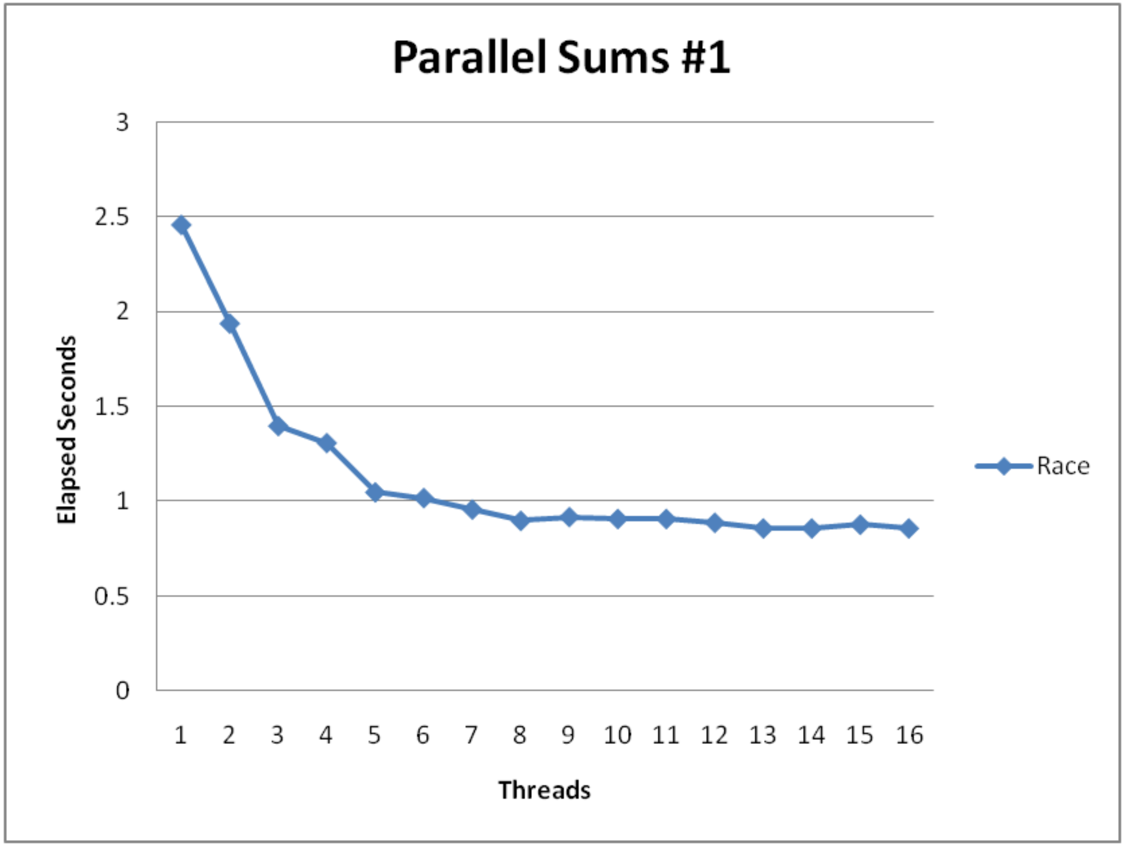
线程参数 Thread argum
*(void *p)*

线程函数：无同步

Thread Function: No Synchronization

```
void *sum_race(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        global_sum += i;
    }
    return NULL;
}
```

无同步的性能 Unsynchronized Performance



- **N = 2^{30}**
- 最佳的加速比 **Best speedup = 2.86X**
- 当大于**1**个线程时得到错误的答案 **Gets wrong answer when > 1 thread!** 为何？ **Why?**

线程函数：信号量/互斥锁

Thread Function: Semaphore / Mutex

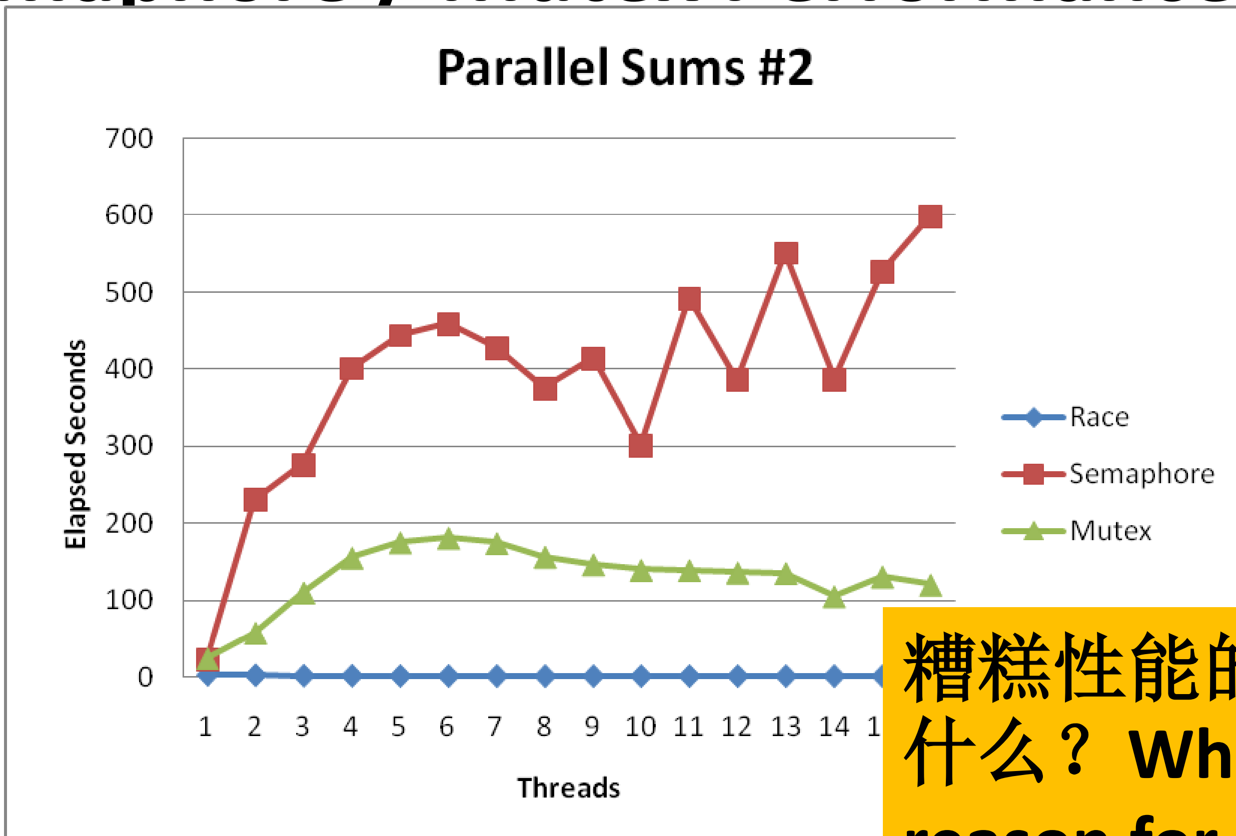
信号量 Semaphore

```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

互斥锁 Mutex

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

信号量/互斥锁性能
Semaphore / Mutex Performance



糟糕性能的主要原因是什么？ What is main reason for poor performance?

- 糟糕的性能 **Terrible Performance**
  - 2.5 seconds 秒 ➔ ~10 minutes 分钟
- 互斥锁比信号量快3倍 **Mutex 3X faster than semaphore**
- 很明显，这些方法都不成功 **Clearly, neither is successful**

单独累积 Separate Accumulation

- 方法#2：每个线程累积到单独的变量中 **Method #2: Each thread accumulates into separate variable**
  - 2A：在相邻数组元素中累加 2A: Accumulate in contiguous array elements
  - 2B：在间隔开的数组元素中累加 2B: Accumulate in spaced-apart array elements
  - 2C：在寄存器中累加 2C: Accumulate in registers

```
/* Partial sum computed by each thread */
data_t psum[MAXTHREADS*MAXSPACING];

/* Spacing between accumulators */
size_t spacing = 1;
```

单独累积：操作
Separate Accumulation: Operation

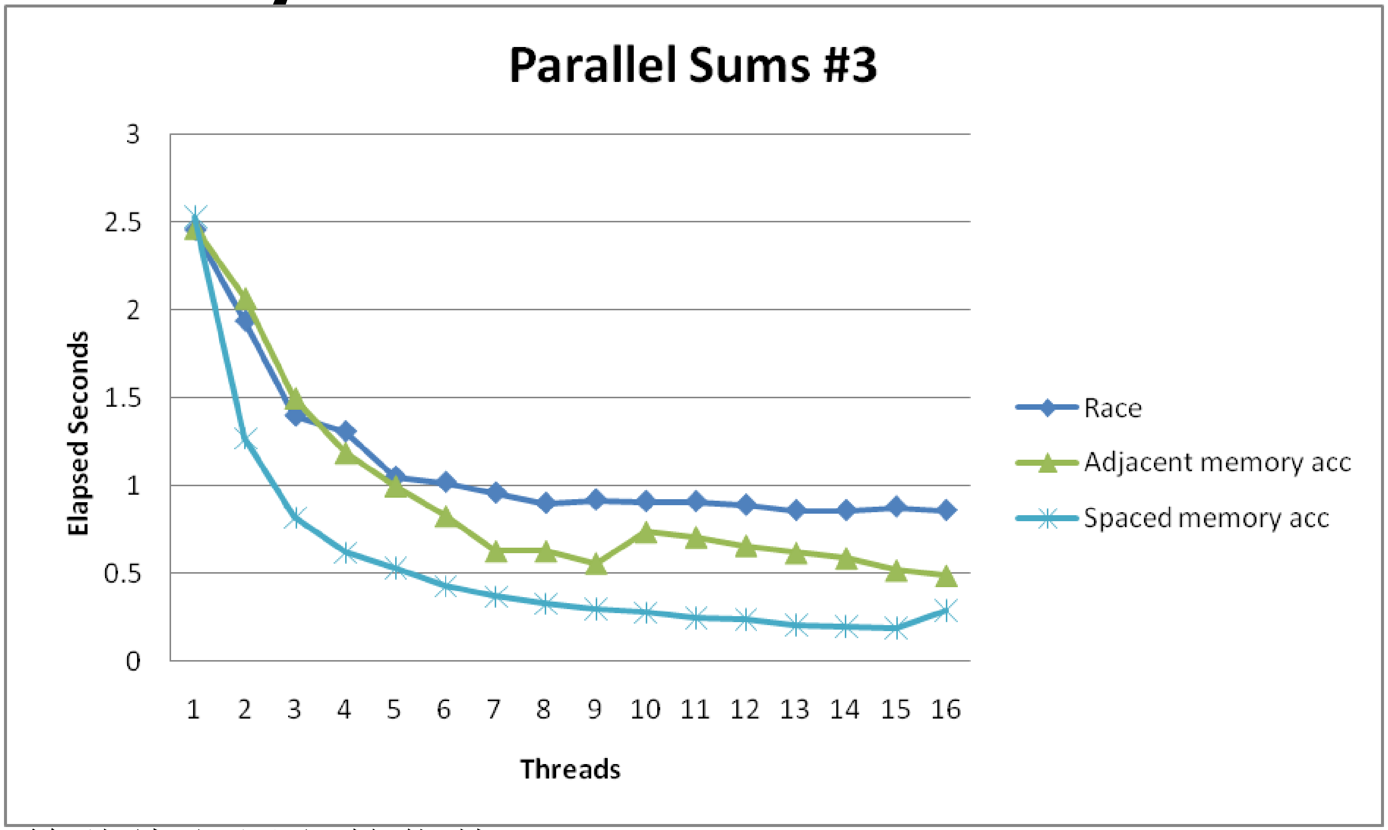
```
    nelems_per_thread = nelems / nthreads;

    /* Create threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        psum[i*spacing] = 0;
        Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    result = 0;

    /* Add up the partial sums computed by each thread */
    for (i = 0; i < nthreads; i++)
        result += psum[i*spacing];

    /* Add leftover elements */
    for (e = nthreads * nelems_per_thread; e < nelems; e++)
        result += e;
```

线程函数：内存累积
Thread Function: Memory Accumulation

互斥锁在哪？ Where is the mutex?

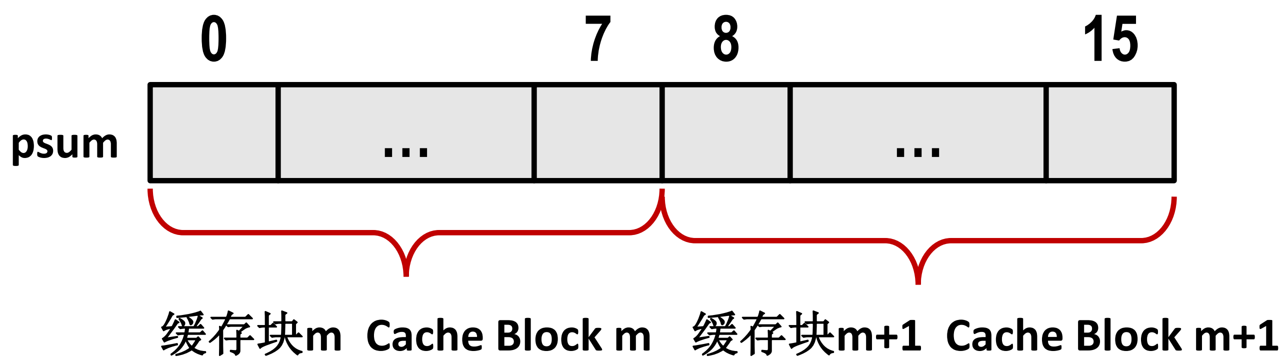
```
void *sum_global(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    size_t index = myid*spacing;
    psum[index] = 0;
    for (i = start; i < end; i++) {
        psum[index] += i;
    }
    return NULL;
}
```

内存累积性能
Memory Accumulation Performance



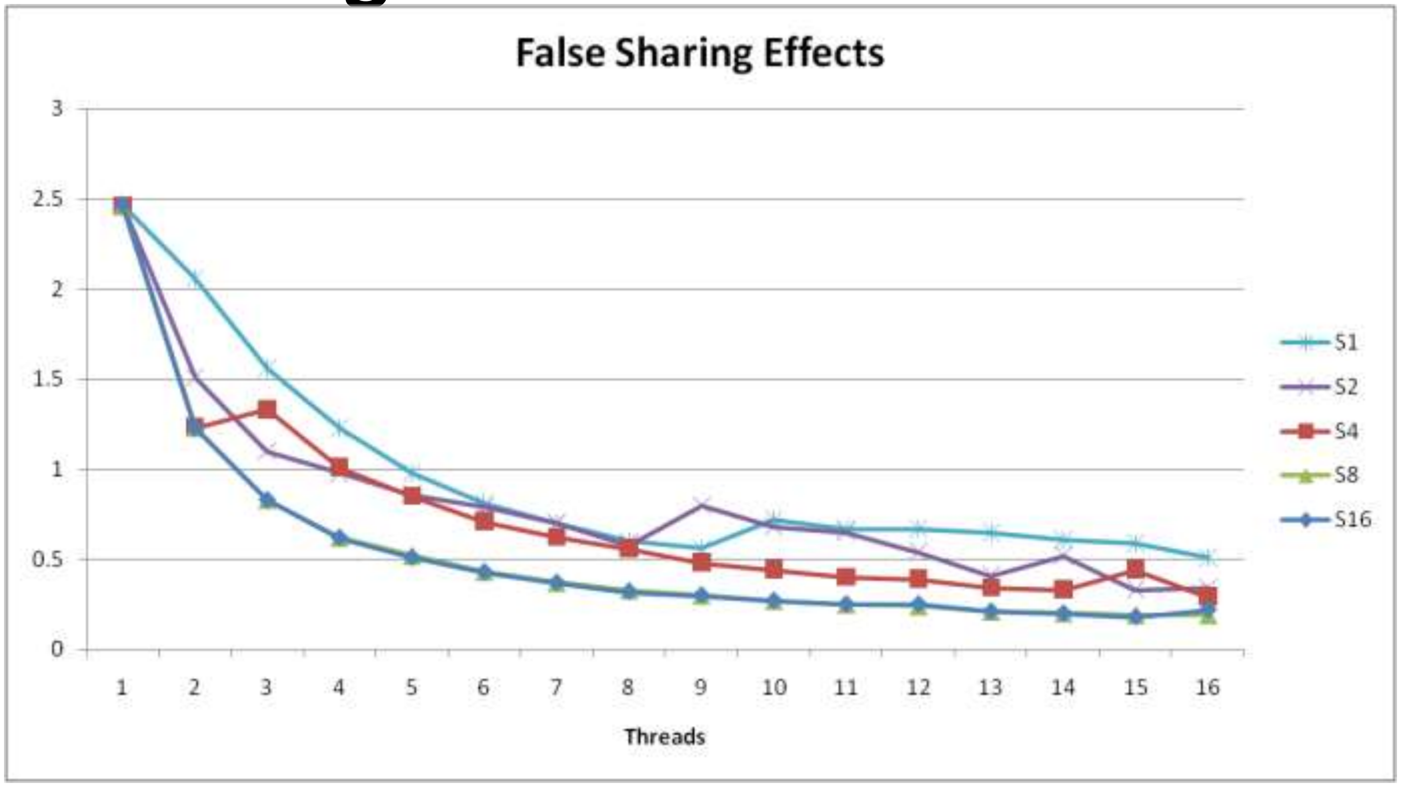
- 单独线程累积的优势 **Clear threading advantage**
  - 连续累积加速比： Adjacent speedup: 5 X
  - 间隔累积加速比： Spaced-apart speedup: 13.3 X (仅观察到加速比大于8 Only observed speedup > 8)
- 为什么进行间隔开累加性能更佳？ **Why does spacing the accumulators apart matter?**

虚假共享 False Sharing



- 缓存块上保持一致性 **Coherence maintained on cache blocks**
- 要更新psum[i]，线程i必须具有独占访问权限 **To update psum[i], thread i must have exclusive access**
  - 共享公共缓存块的线程将继续为访问块而相互争斗 Threads sharing common cache block will keep fighting each other for access to block

虚假共享的性能
False Sharing Performance



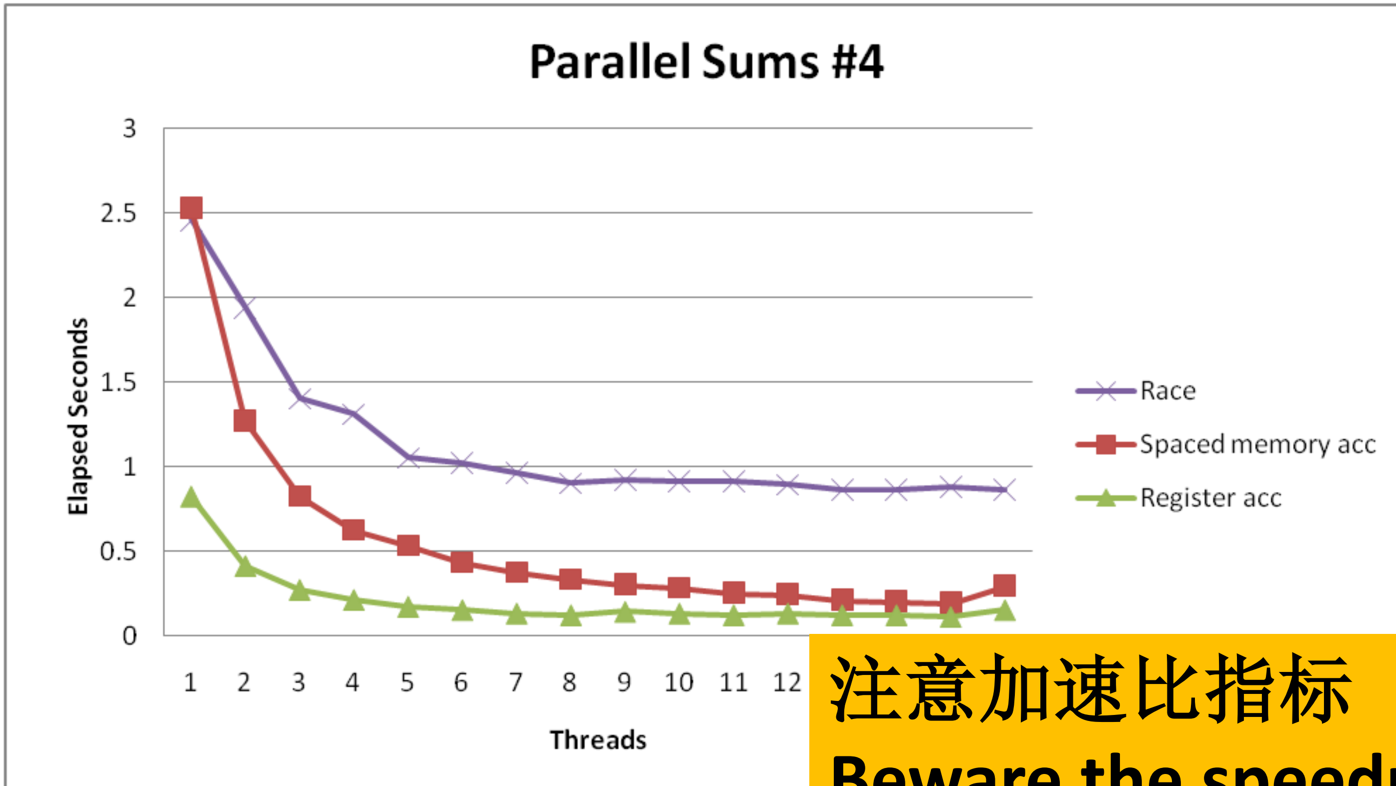
- 最佳间隔性能比最佳相邻性能高2.8倍 Best spaced-apart performance 2.8 X better than best adjacent

- **演示缓存块大小为64 Demonstrates cache block size = 64**
  - 8字节值 8-byte values
  - 将间隔增加到8以上没有性能改善 No benefit increasing spacing beyond 8

线程函数：寄存器累积
Thread Function: Register Accumulation

```
void *sum_local(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    size_t index = myid*spacing;
    data_t sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[index] = sum;
    return NULL;
}
```

寄存器累积性能
Register Accumulation Performance



注意加速比指标
Beware the speedup metric!

- **单独线程累积优势 Clear threading advantage**
  - 加速比/Speedup = 7.5 X
- **比最快的内存累积好2倍 2X better than fastest memory accumulation**

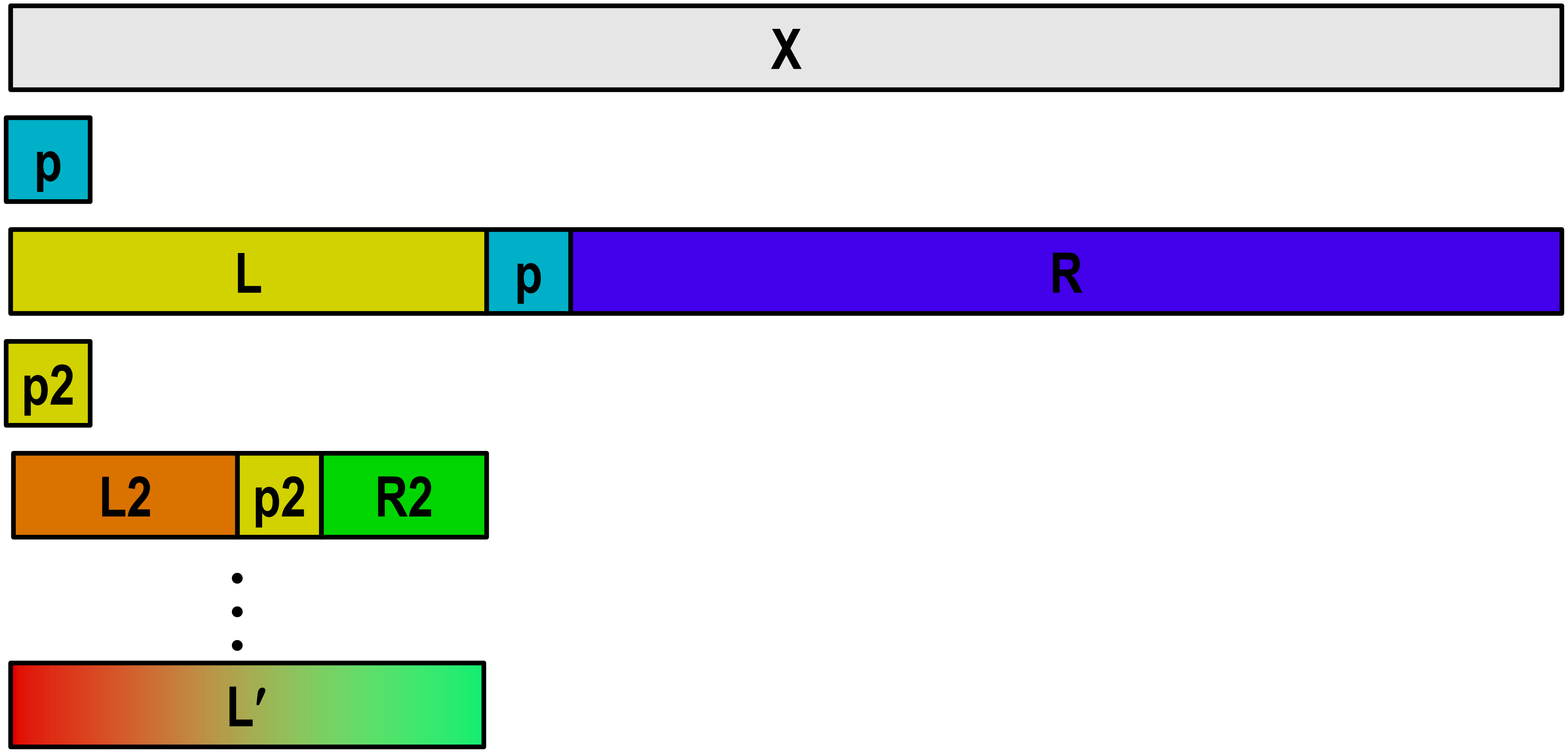
经验教训 Lessons learned

- 共享内存可能开销很高 **Sharing memory can be expensive**
  - 关注真实共享 Pay attention to true sharing
  - 注意虚假共享 Pay attention to false sharing
- 尽可能使用寄存器 **Use registers whenever possible**
  - (记住cachelab Remember cachelab)
  - 尽可能使用本地缓存 Use local cache whenever possible
- 处理剩余的数据 **Deal with leftovers**
- 在检查性能时，与最佳顺序实现进行比较 **When examining performance, compare to best possible sequential implementation**

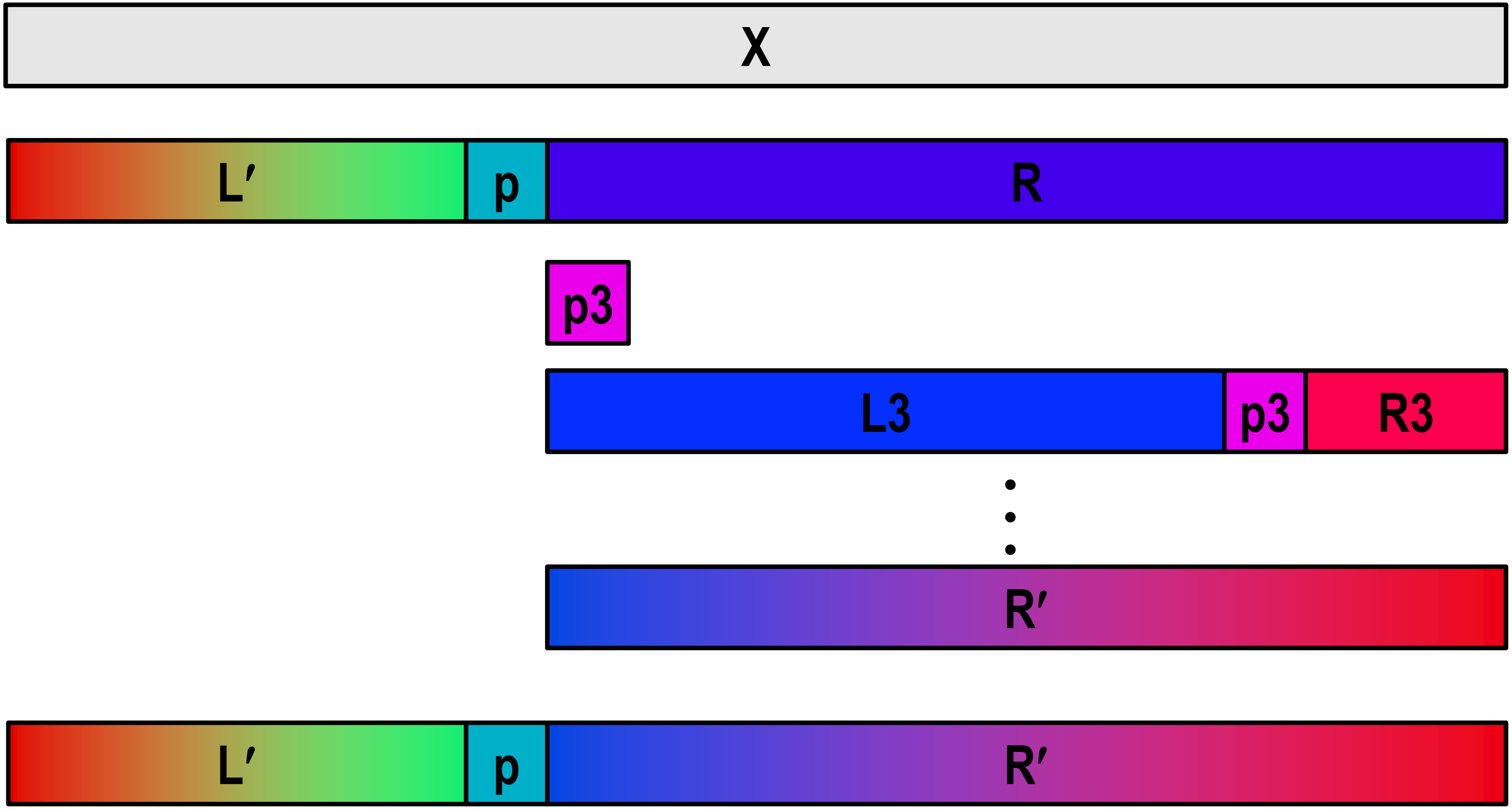
更重要的示例：排序
A More Substantial Example: Sort

- **N个随机数集合排序 Sort set of N random numbers**
- **多种可能的算法 Multiple possible algorithms**
  - 使用并行版本的快速排序 Use parallel version of quicksort
- **对X集合进行顺序快速排序 Sequential quicksort of set of values X**
  - 从X选择“中心点”p Choose “pivot” p from X
  - 重新排列X Rearrange X into
    - 左边集合：值小于等于p L: Values ≤ p
    - 右边集合：值大于等于p R: Values ≥ p
  - 对左边集合进行递归排序得到L′ Recursively sort L to get L′
  - 对右边集合进行递归排序得到R′ Recursively sort R to get R′
  - 返回 Return L′ : p : R′

顺序快速排序可视化
Sequential Quicksort Visualized



顺序快速排序可视化
Sequential Quicksort Visualized



顺序快速排序代码
Sequential Quicksort Code

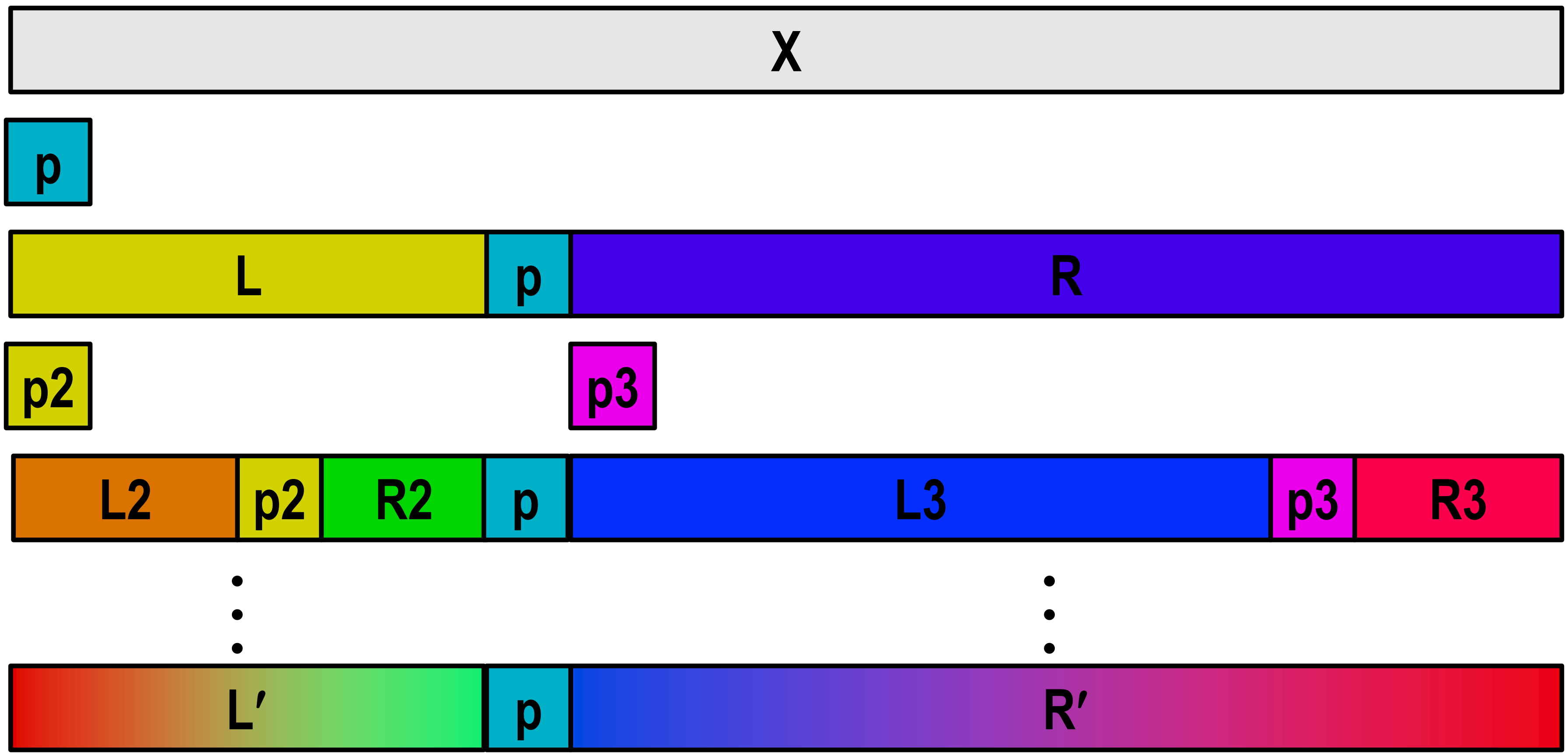
```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
  }

  /* Partition returns index of pivot */
  size_t m = partition(base, nele);
  if (m > 1)
    qsort_serial(base, m);
  if (nele-1 > m+1)
    qsort_serial(base+m+1, nele-m-1);
}
```

- 从base开始对nele个元素排序 **Sort nele elements starting at base**
  - 如果有多于一个元素，则递归排序L或R Recursively sort L or R if has more than one element

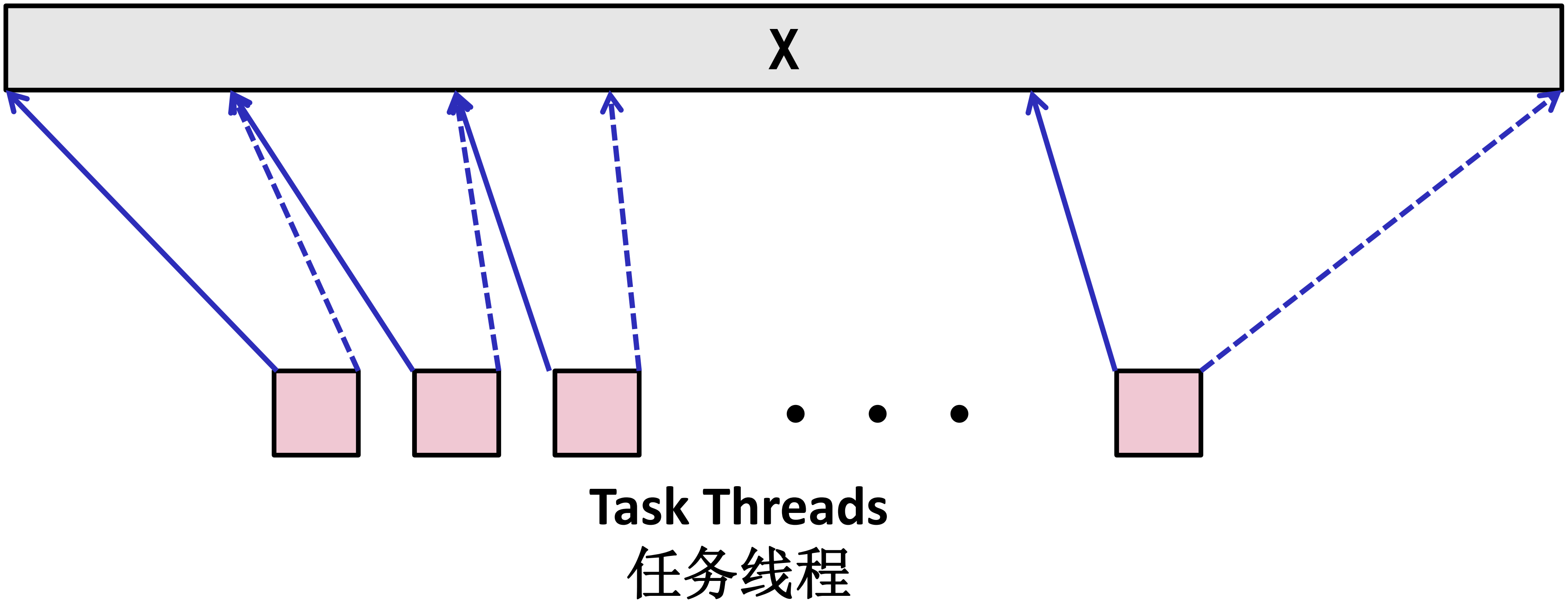
并行快速排序 Parallel Quicksort

- 集合X的并行快速排序 **Parallel quicksort of set of values X**
  - 如果N小于等于Nthresh，执行顺序快速排序 If N ≤ Nthresh, do sequential quicksort
  - 否则 Else
    - 从X选择“中心点”p Choose “pivot” p from X
    - 重新排列X Rearrange X into
      - 左集合：值小于等于p L: Values ≤ p
      - 右集合：值大于等于p R: Values ≥ p
    - 递归生成单独的线程 Recursively spawn separate threads
      - 排序L以获得L′ Sort L to get L′
      - 排序R以获得R′ Sort R to get R′
    - 返回 Return L′ : p : R′

并行快速排序可视化
Parallel Quicksort Visualized

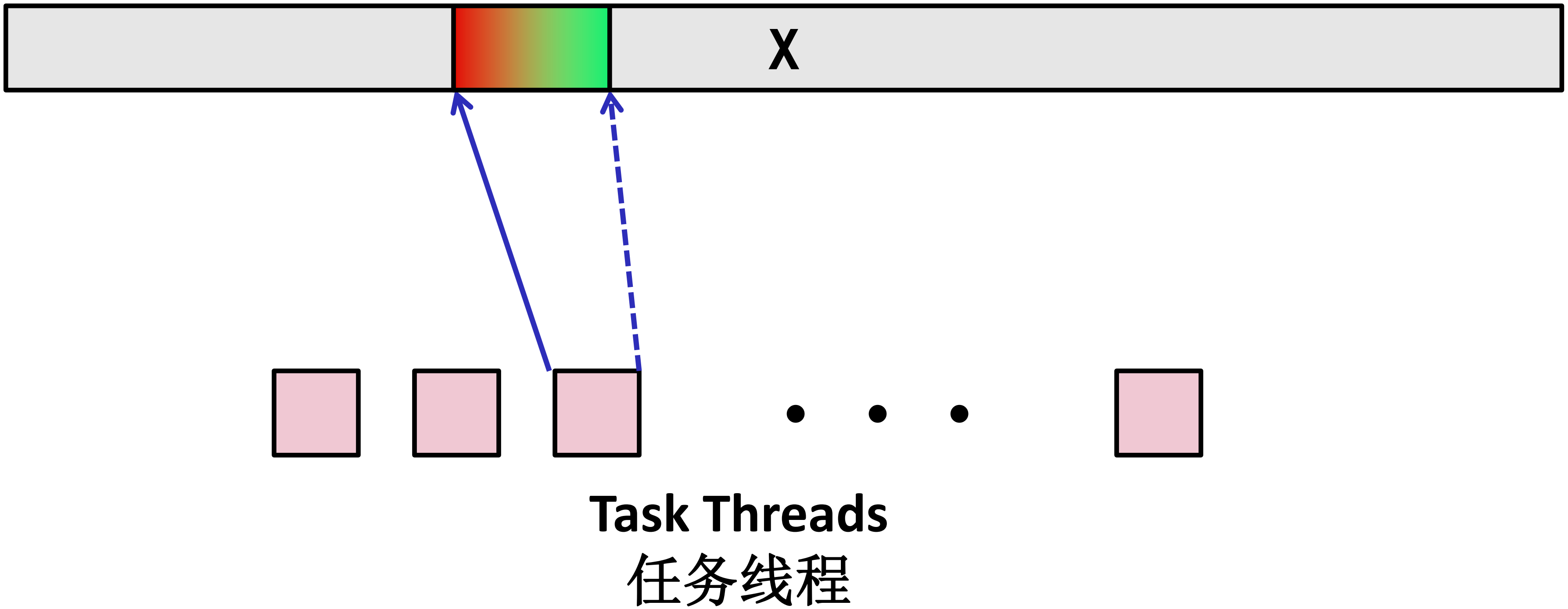


线程结构：排序任务
Thread Structure: Sorting Tasks



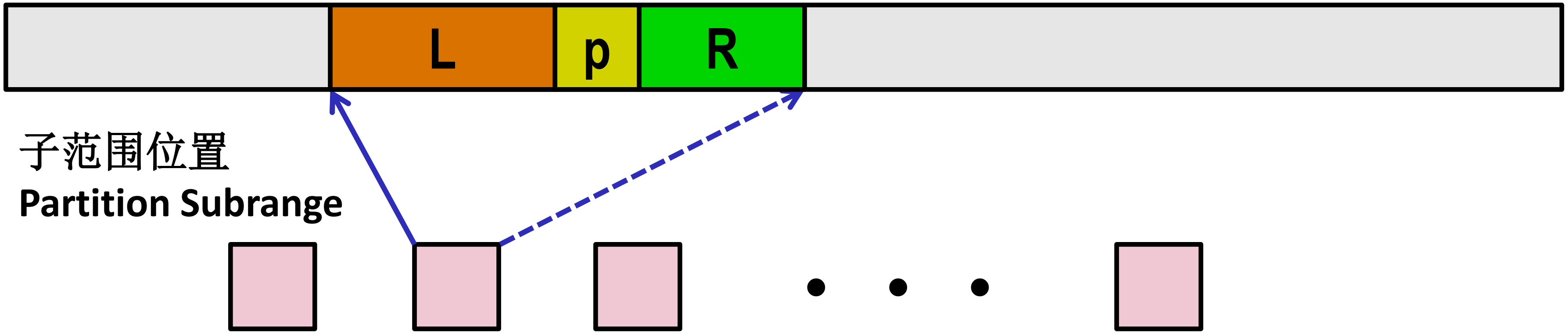
- 任务：排序子范围数据 **Task: Sort subrange of data**
  - 指定为： Specify as:
    - base：起始地址 `base`: Starting address
    - nele：子范围中的元素数 `nele`: Number of elements in subrange
- 作为单独线程运行 **Run as separate thread**

小排序任务操作 Small Sort Task Operation

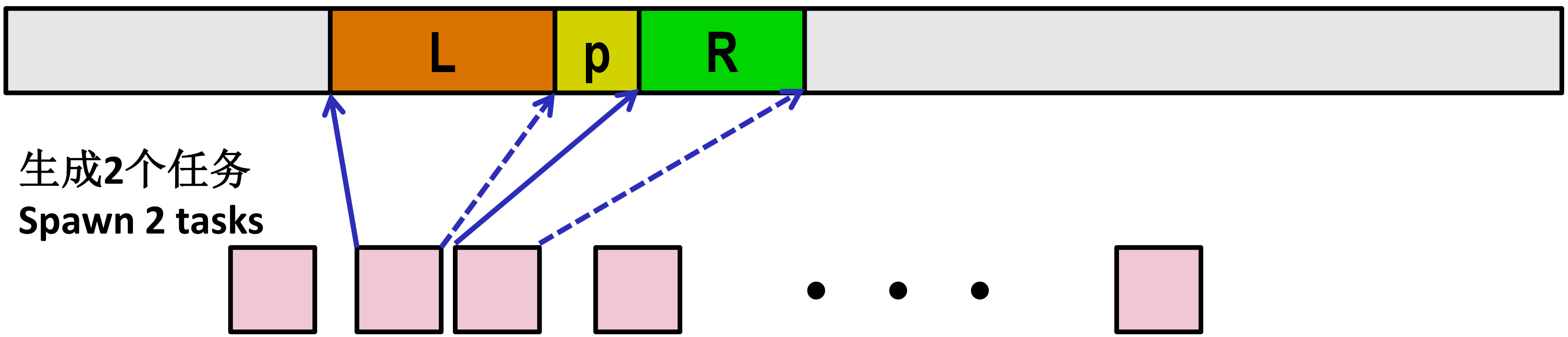


- 排序子范围数据使用串行快速排序 Sort subrange using serial quicksort

大排序任务操作
Large Sort Task Operation



子范围位置
Partition Subrange



生成2个任务
Spawn 2 tasks

顶层函数（简化）
Top-Level Function (Simplified)

```
void tqsort(data_t *base, size_t nele) {
    init_task(nele);
    global_base = base;
    global_end = global_base + nele - 1;
    task_queue_ptr tq = new_task_queue();
    tqsort_helper(base, nele, tq);
    join_tasks(tq);
    free_task_queue(tq);
}
```

- 初始化数据结构 **Sets up data structures**
- 调用递归排序例程 **Calls recursive sort routine**
- 保持加入线程，直到没有剩余 **Keeps joining threads until none left**
- 释放数据结构 **Frees data structures**

递归排序例程（简化）Recursive sort routine (Simplified)

```
/* Multi-threaded quicksort */
static void tqsort_helper(data_t *base, size_t nele,
                          task_queue_ptr tq) {
    if (nele <= nele_max_sort_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele, tq);
    spawn_task(tq, sort_thread, (void *) t);
}
```

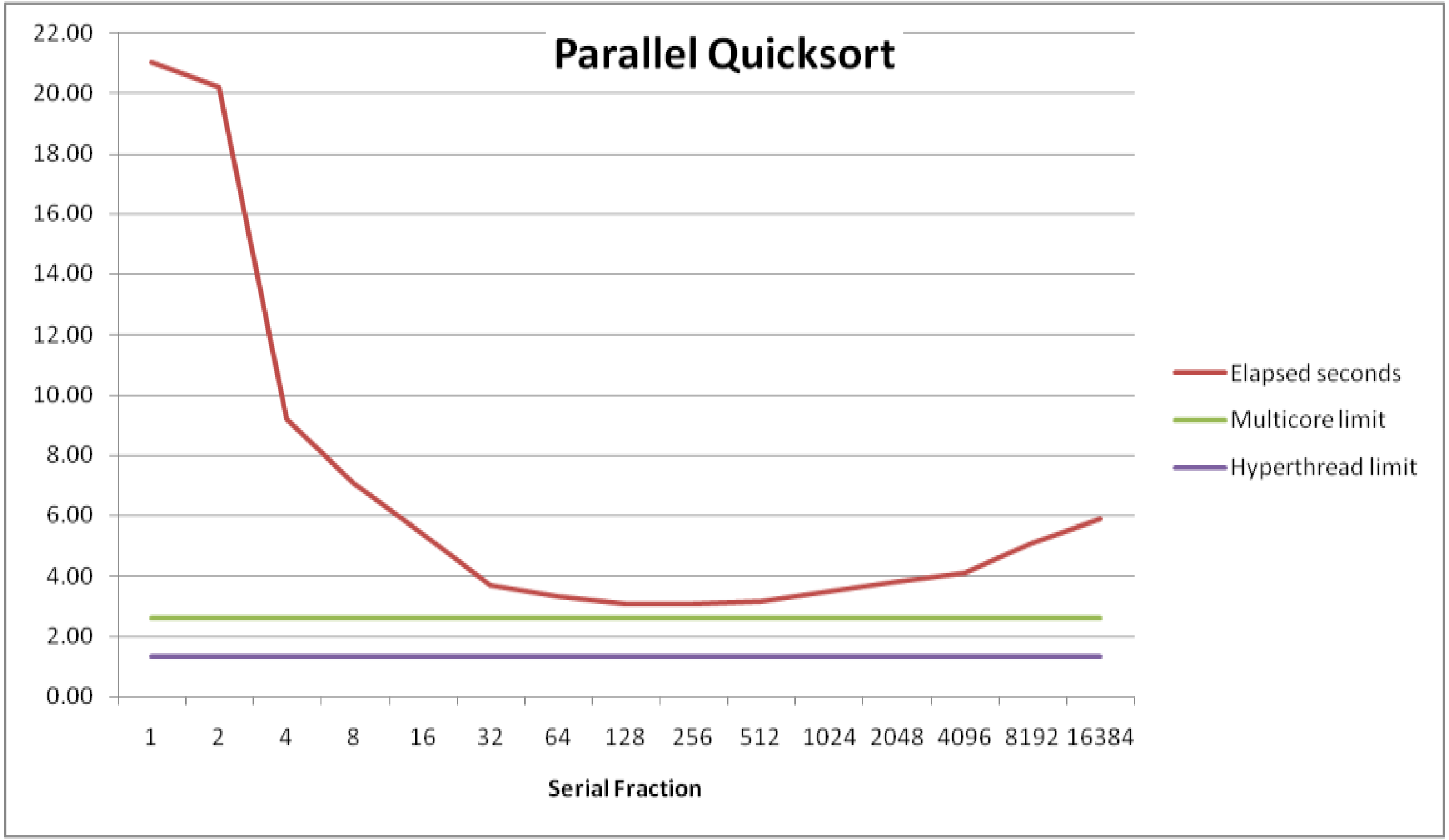
- 小分区：按顺序排序 **Small partition: Sort serially**
- 大分区：生成新的排序任务 **Large partition: Spawn new sort task**

排序任务线程（简化）
Sort task thread (Simplified)

```
/* Thread routine for many-threaded quicksort */
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    task_queue_ptr tq = t->tq;
    free(vargp);
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m, tq);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1, tq);
    return NULL;
}
```

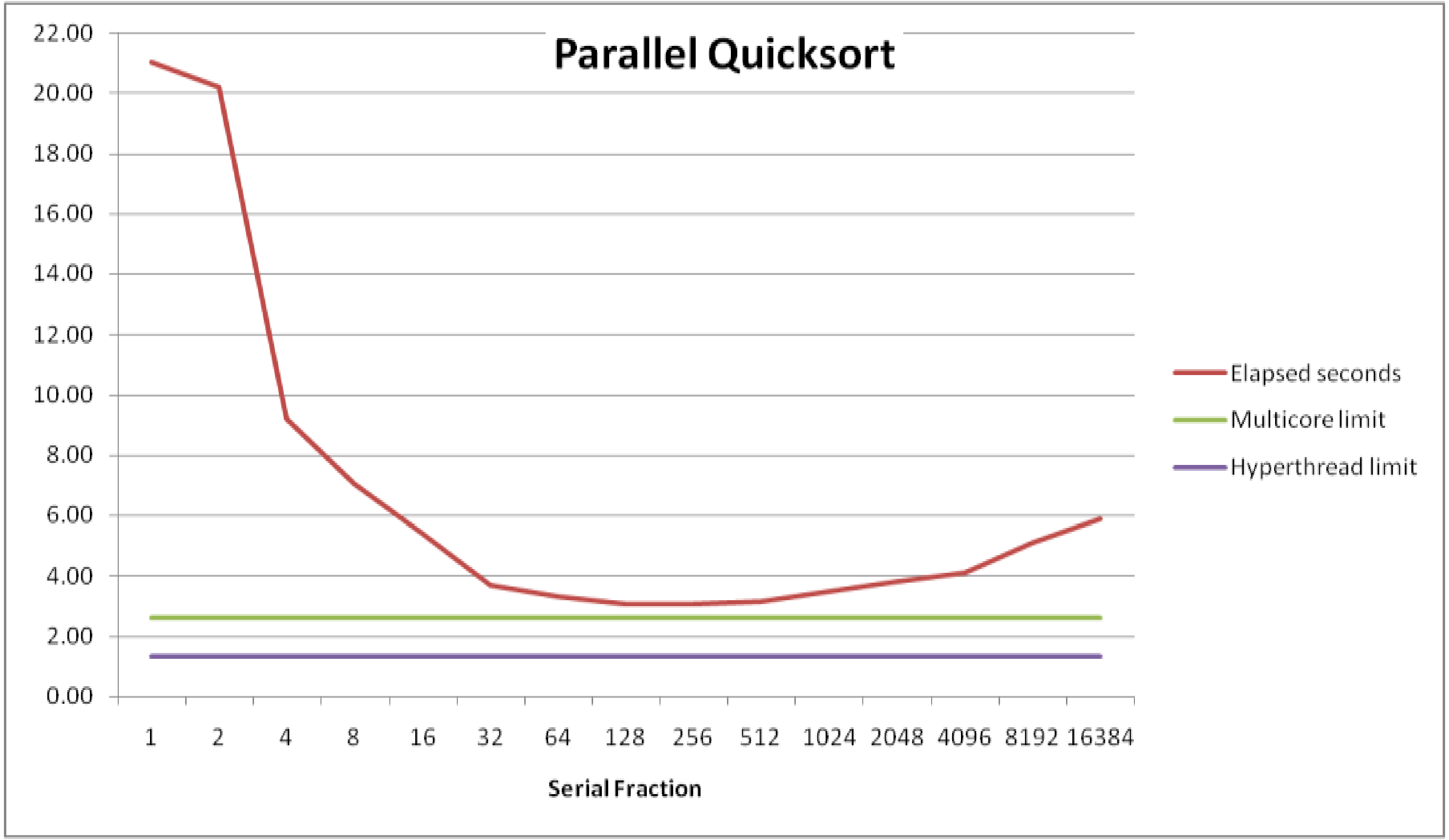
- 获取任务参数 **Get task parameters**
- 执行分区步骤 **Perform partitioning step**
- 在每个分区上调用递归排序例程（如果部分大小大于1）
  Call recursive sort routine on each partition (if size of part > 1)

并行快速排序性能
Parallel Quicksort Performance



- 串行比例：进行串行排序的输入比例 **Serial fraction: Fraction of input at which do serial sort**
- 排序128M随机值 **Sort 2^{27} (134,217,728) random values**
- 最佳加速比 **Best speedup = 6.84X**

并行快速排序性能
Parallel Quicksort Performance



- 在广泛的串行比例范围内表现良好 **Good performance over wide range of fraction values**
  - F太小：并行度不够 F too small: Not enough parallelism
  - F太大：线程开销太高 F too large: Thread overhead too high

阿姆达尔定律（旅行模拟）
Amdahl’s Law (Travel Analogy)

加速比 Speed-Up

- 从PIT直飞LHR **Flying jet non-stop from PIT -> LHR:** **7.5 Hours** **1**
- **或者，老式SST方式： Or, old fashioned SST way:**
  - Fly jet from PIT -> JFK: 1.5 Hours
  - Fly SST from JFK -> LHR: 3.5 Hours **5 Hours** **1.5x**
- **或者，使用FTL Or, Using FTL:**
  - Fly jet from PIT -> JFK: 1.5 Hours
  - Fly FTL from JFK -> LHR: .01 Hours **1.51 Hours** **~5x**

- **最好的加速比是5倍，即使是FTL，因为必须到达纽约 Best possible speed up is 5X, even with FTL because have to get to New York.**
- **PIT：匹兹堡 LHR：伦敦 JFK：纽约**
- **SST：超音速客机 FTL：超光速**

阿姆达尔定律 Amdahl's Law

- **总体问题 Overall problem**
  - T所需的总顺序执行时间 T Total sequential time required
  - p可加速的总比例 p Fraction of total that can be sped up ($0 \le p \le 1$)
  - k加速系数 k Speedup factor
- **最终性能 Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - 可以加速的部分速度快k倍 Portion which can be sped up runs k times faster
    - 无法加速的部分保持不变 Portion which cannot be sped up stays the same
  - 最大可能加速比 Maximum possible speedup
    - $k = \infty$
    - $T_\infty = (1-p)T$

阿姆达尔定律（旅行模拟）
Amdahl's Law (Travel Analogy)

		加速比 Speed-Up
■ 从PIT直飞LHR **Flying jet non-stop from PIT -> LHR:**	**7.5 Hours**	**1**
■ **或者，老式SST方式： Or, old fashioned SST way:**		
▪ Fly jet from PIT -> JFK: 1.5 Hours		
▪ Fly SST from JFK -> LHR: 3.5 Hours	**5 Hours**	**1.5x**
■ **或者，使用FTL Or, Using FTL:**		
▪ Fly jet from PIT -> JFK: 1.5 Hours		
▪ Fly FTL from JFK -> LHR: .01 Hours	**1.51 Hours**	**~5x**

- **最好的加速比是5倍，即使是FTL，因为必须到达纽约 Best possible speed up is 5X, even with FTL because have to get to New York.**
  - $T=7.5$, $p=6/7.5=.8$, $k=\infty \Rightarrow T_{\infty} = (1-p)T=1.5$ 最大加速比 max speed-up =5x

阿姆达尔定律的示例
Amdahl's Law Example

- 总体问题 **Overall problem**
  - T = 10 Total time required 所需总时间
  - p = 0.9 Fraction of total which can be sped up 可加速的总比例
  - k = 9 Speedup factor 加速系数
- 最终性能 **Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$ (5倍加速比 a 5x speedup)
- 最大可能加速比 **Maximum possible speedup**
  - $T_\infty = 0.1 * 10.0 = 1.0$ (10倍加速比 a 10x speedup)
    - 拥有**无限的**并行计算资源！ With **infinite** parallel computing resources!
  - 极限加速比显示**算法**极限 Limit speedup shows **algorithmic** limitation

阿姆达尔定律和并行快速排序
Amdahl's Law & Parallel Quicksort

- 顺序程序瓶颈 **Sequential bottleneck**
  - 顶层分区：无加速 Top-level partition: No speedup
  - 第二级：小于等于2倍加速比 Second level: ≤ 2X speedup
  - 第k级：小于等于2^{k-1}加速比 k^{th} level: ≤ 2^{k-1}X speedup
- 启示 **Implications**
  - 小规模并行的良好性能 Good performance for small-scale parallelism
  - 需要并行化分区步骤以获得大规模并行性 Would need to parallelize partitioning step to get large-scale parallelism
    - 基于规则抽样的并行排序 Parallel Sorting by Regular Sampling
      - "并行与分布式计算" H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992

经验教训 Lessons Learned

- **必须具有并行化策略 Must have parallelization strategy**
  - 划分为K个独立部分 Partition into K independent parts
  - 分而治之 Divide-and-conquer
- **内部循环必须无同步 Inner loops must be synchronization free**
  - 同步操作非常耗时 Synchronization operations very expensive
- **当心硬件瑕疵 Watch out for hardware artifacts**
  - 需要了解处理器和内存结构 Need to understand processor & memory structure
  - 共享和虚假共享全局数据 Sharing and false sharing of global data
- **当心阿姆达尔定律 Beware of Amdahl’s Law**
  - 串行代码可能成为瓶颈 Serial code can become bottleneck
- **你能行！ You can do it!**
  - 实现适度的并行性并不困难 Achieving modest levels of parallelism is not difficult
  - 建立实验框架并测试多种策略 Set up experimental framework and test multiple strategies