



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 李秀星

原作者:

Randal E. Bryant and David R. O'Hallaron



Carnegie
Mellon
University

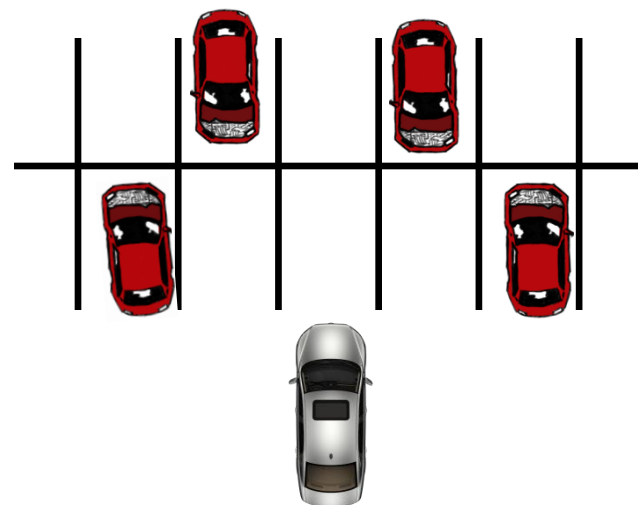
并发编程很难！

Concurrent Programming is Hard!

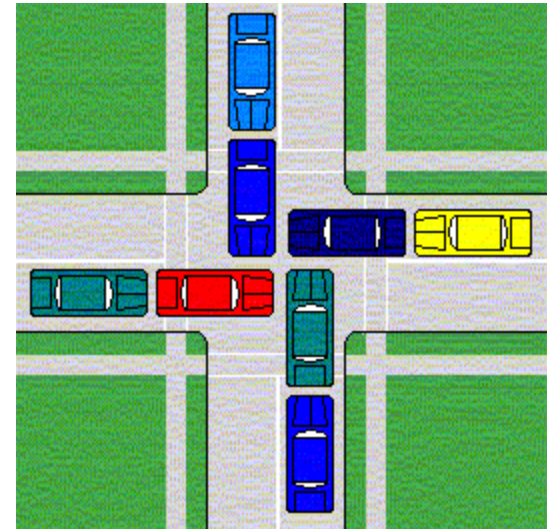


- 人类的思维往往是顺序的 The human mind tends to be sequential
- 时间的概念常常误导人 The notion of time is often misleading
- 考虑计算机系统中所有可能的事件顺序非常容易出错，而且经常是不可能的 Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

数据竞争 Data Race



死锁 Deadlock





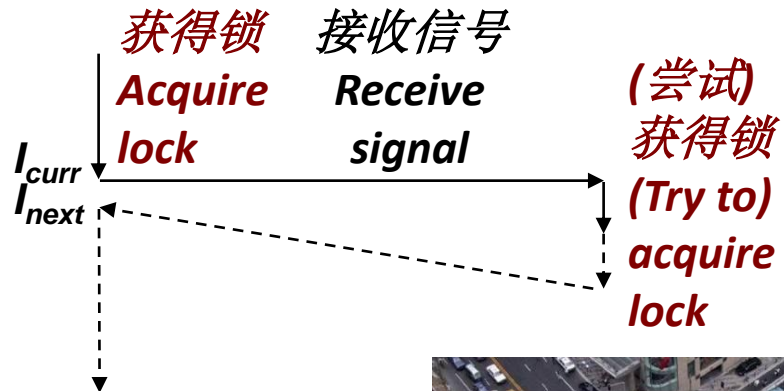
死锁 Deadlock

- 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

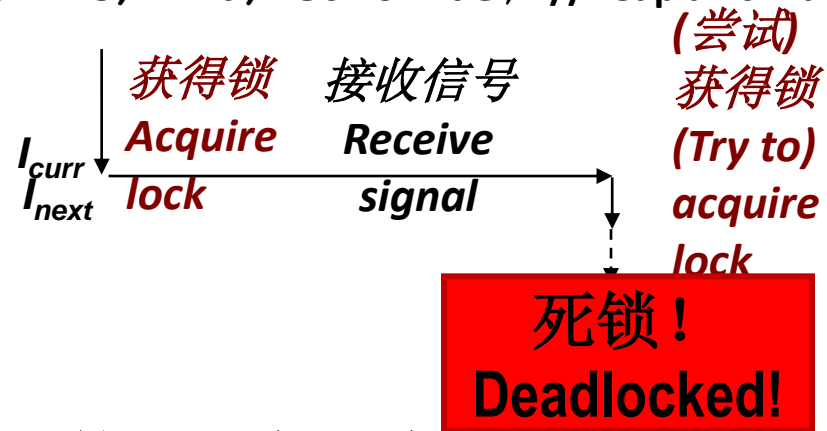


- 信号处理程序示例 Example from signal handlers
- 为什么不在处理程序中使用printf? Why don't we use printf in handlers?

```
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;
}
```

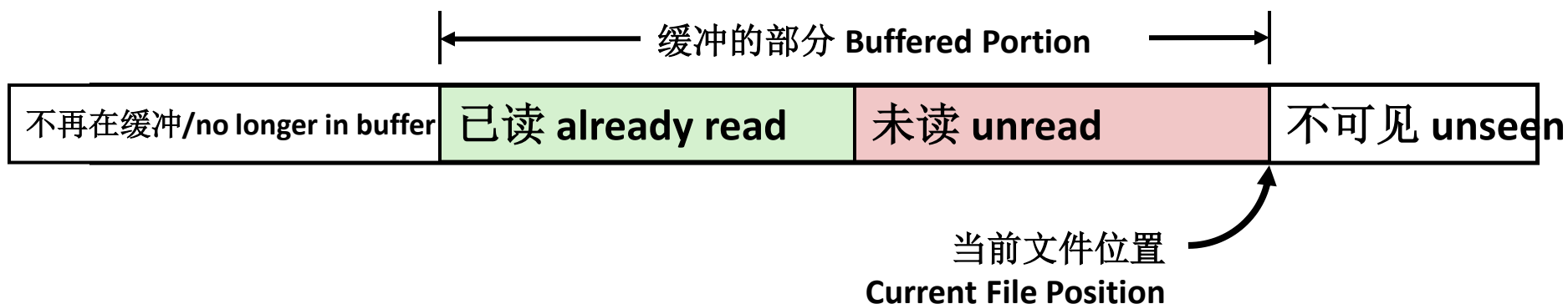
```
Child #0 started
Child #1 started
Child #2 started
Child #3 started
Child exited!
Child #4 started
Child exited!
Child #5 started
.
.
.
Child #5888 started
Child #5889 started
```

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

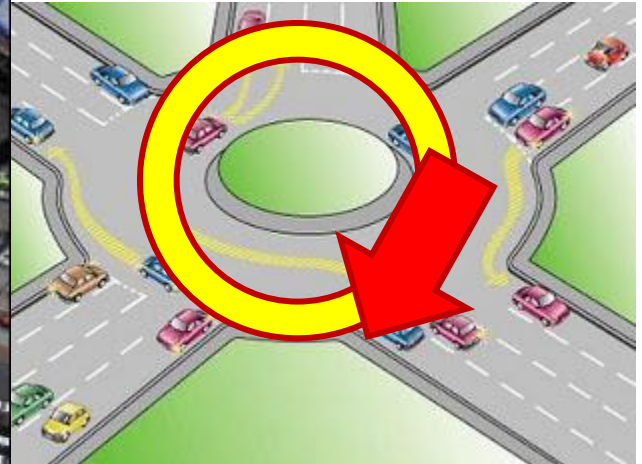


- 需要锁以访问该共享缓冲区 Require locks to access the shared buffers

活锁 Livelock

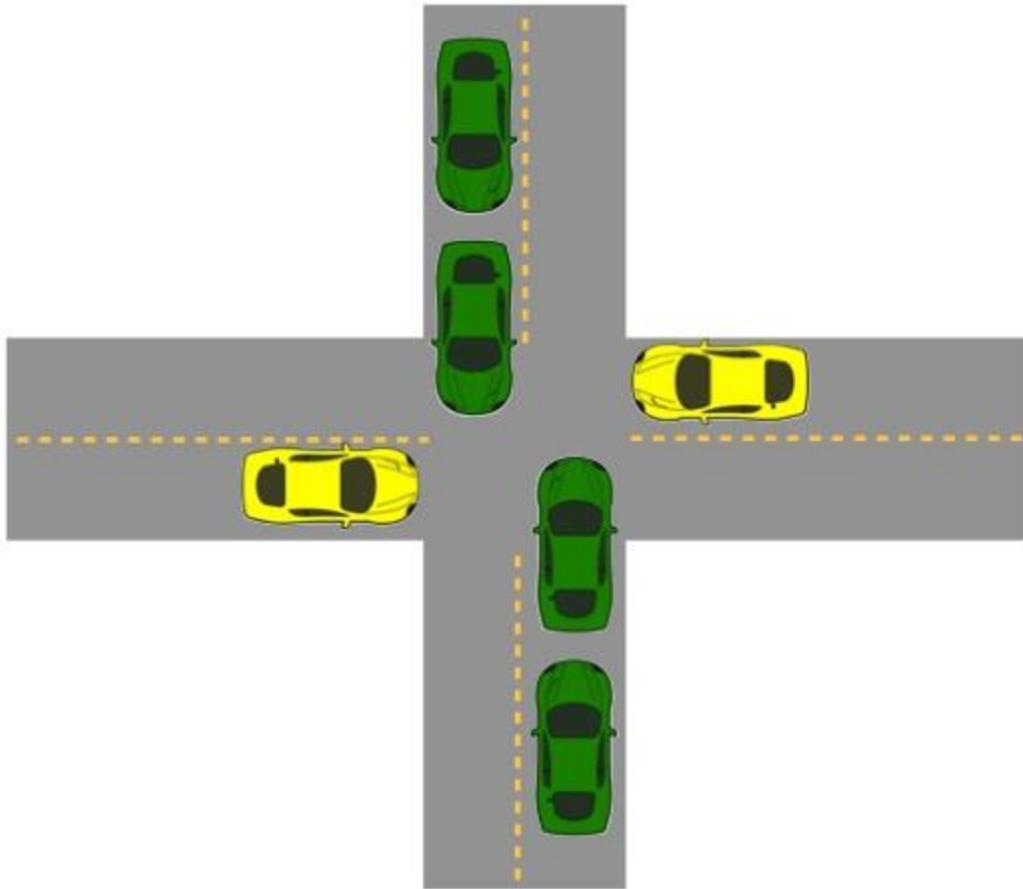


活锁 Livelock





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!

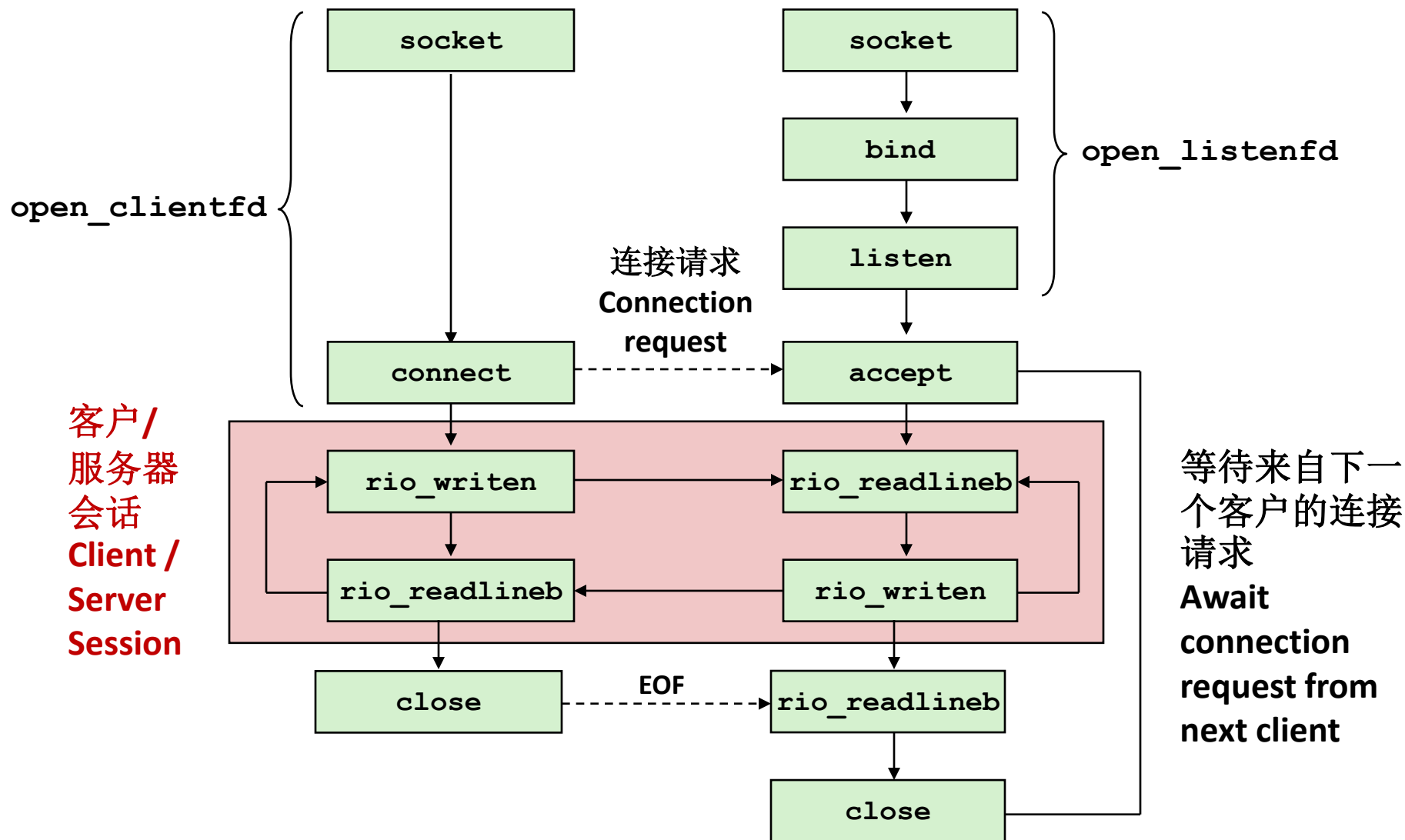
提醒：迭代式回声服务器

Reminder: Iterative Echo Server



客户 *Client*

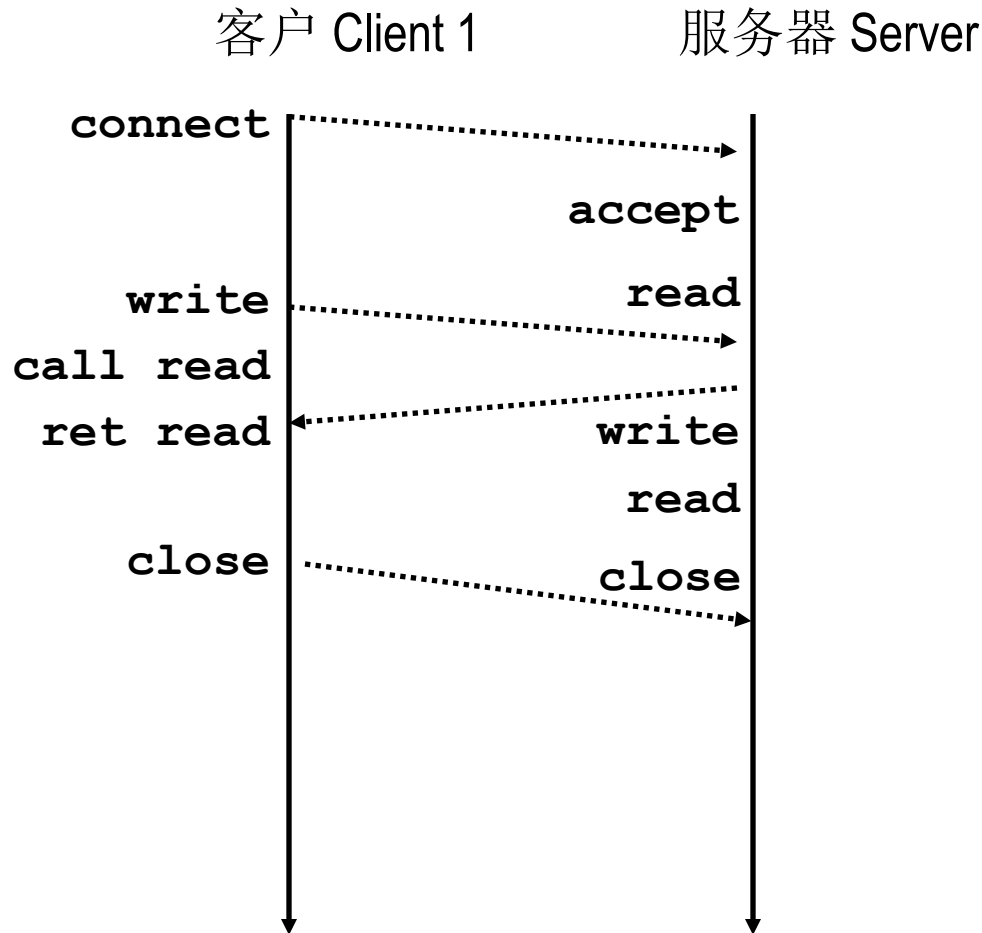
服务器 *Server*



迭代服务器 Iterative Servers



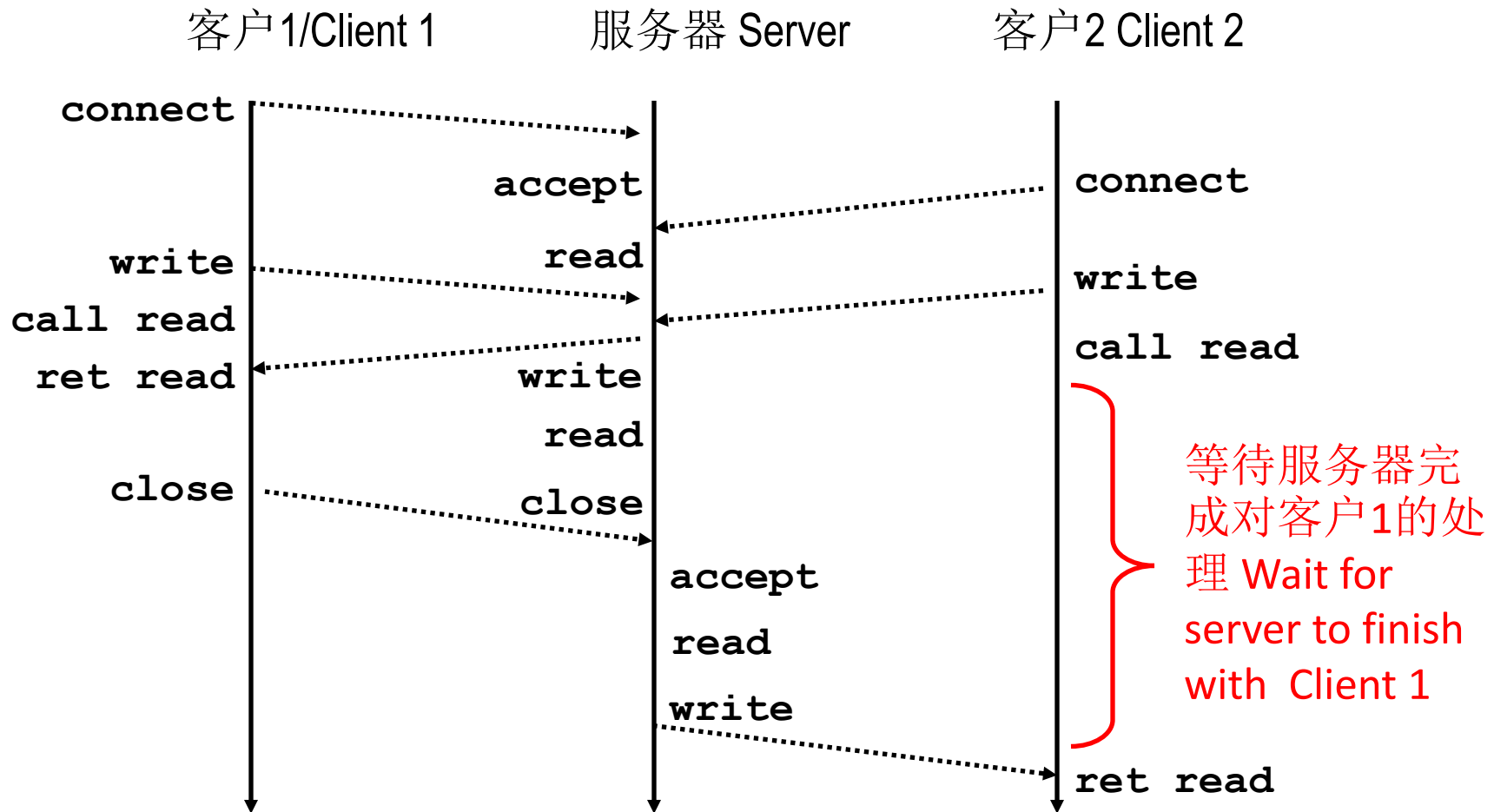
- 迭代服务器一次处理一个请求 Iterative servers process one request at a time



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



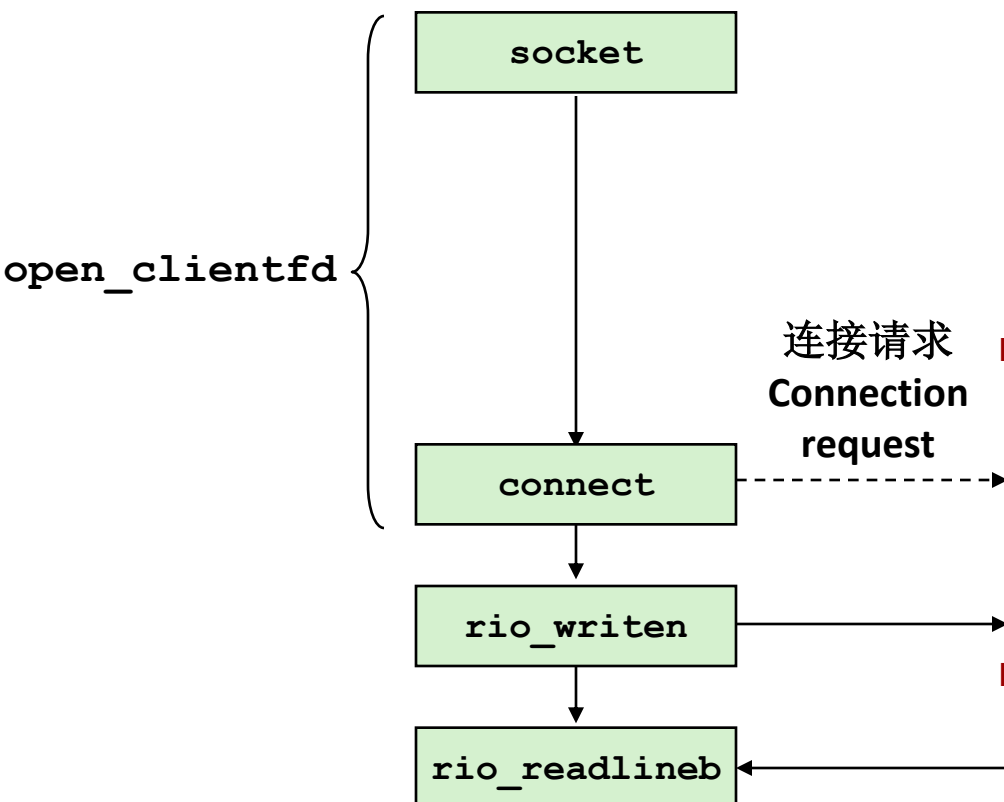
第二个客户阻塞在哪里？



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

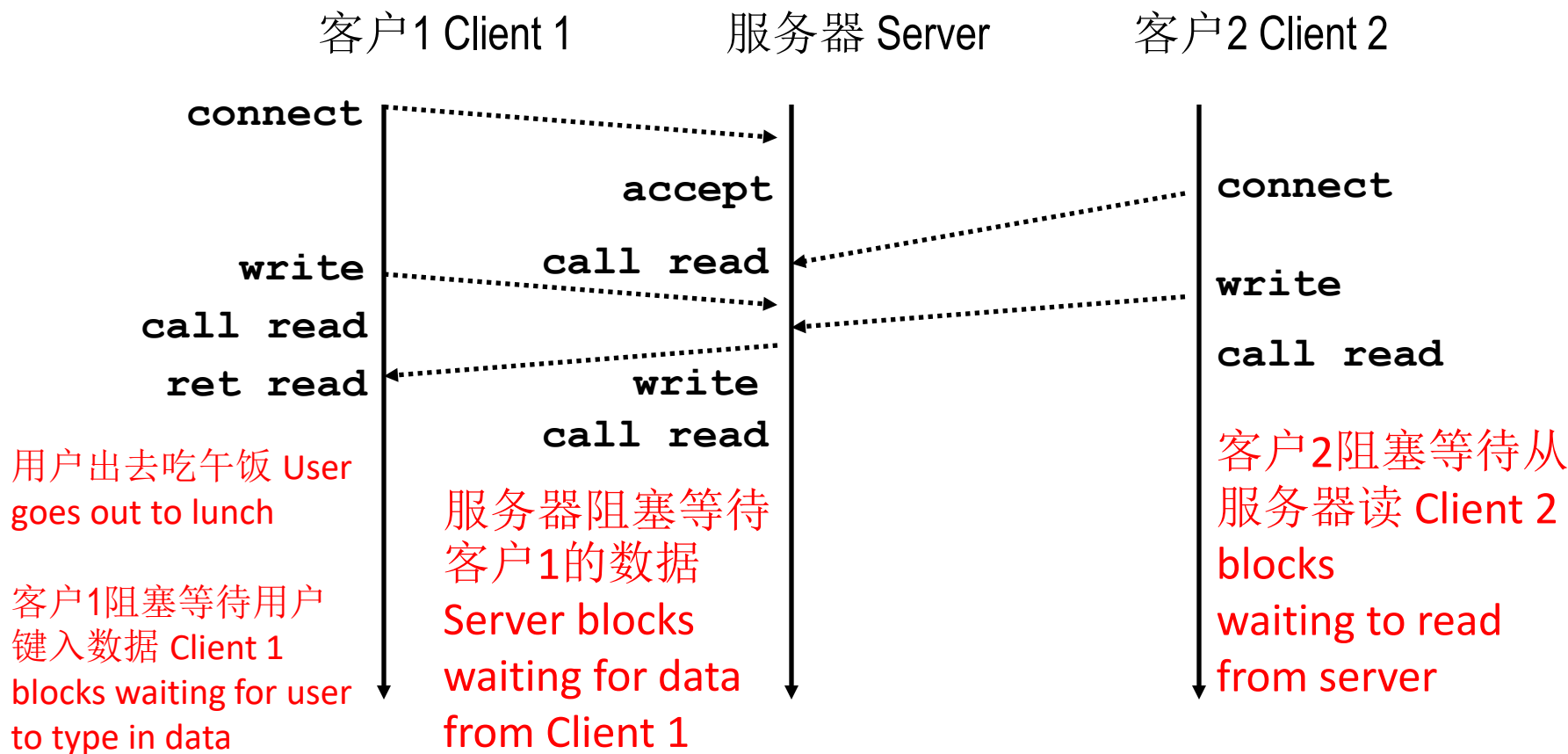
客户 Client



- `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

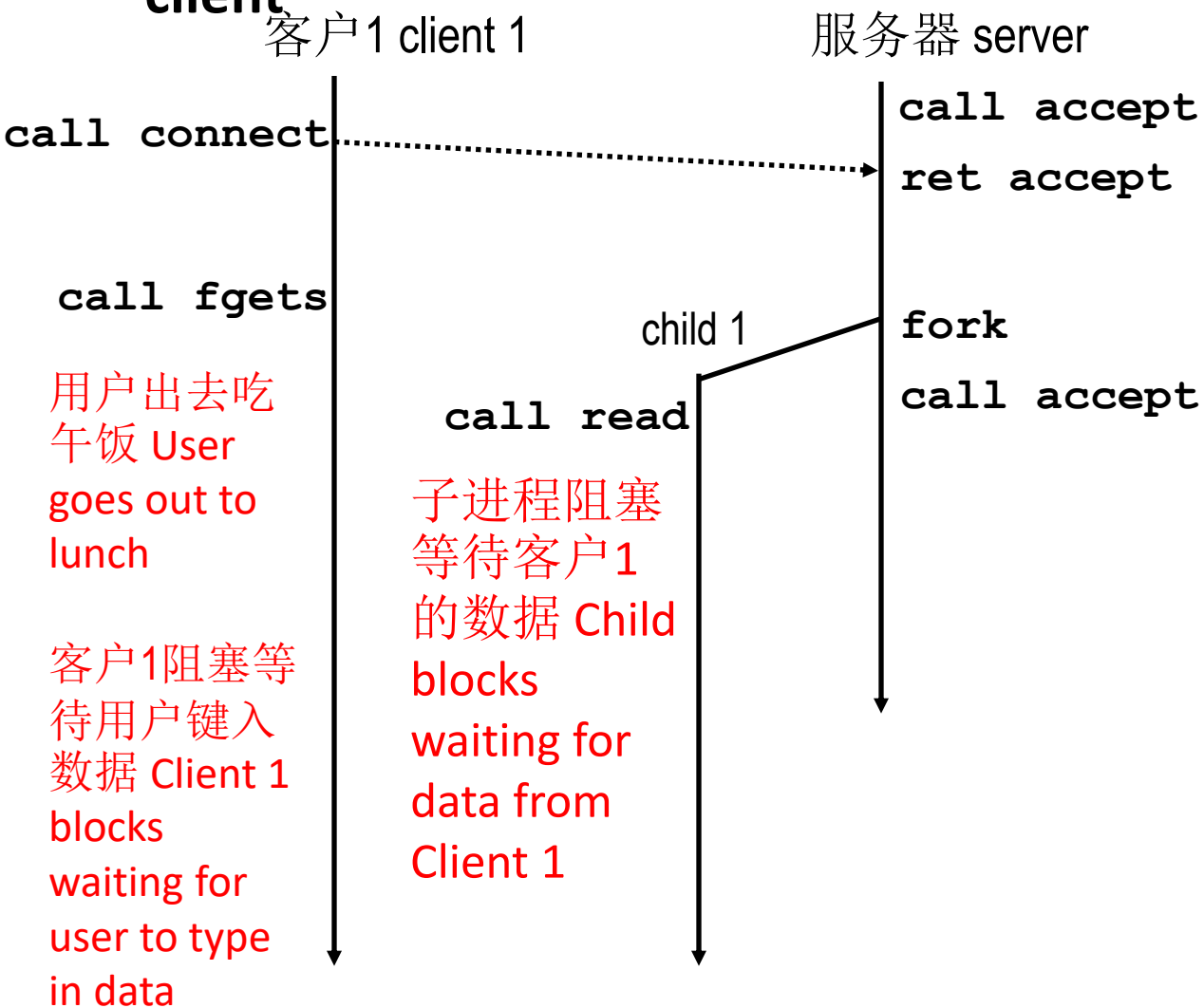
- 内核自动交错多个逻辑流 Kernel automatically interleaves multiple logical flows
- 每个流共享相同的地址空间 Each flow shares the **same** address space
- 基于进程和基于事件两种方法的混合 Hybrid of process-based and event-based

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**

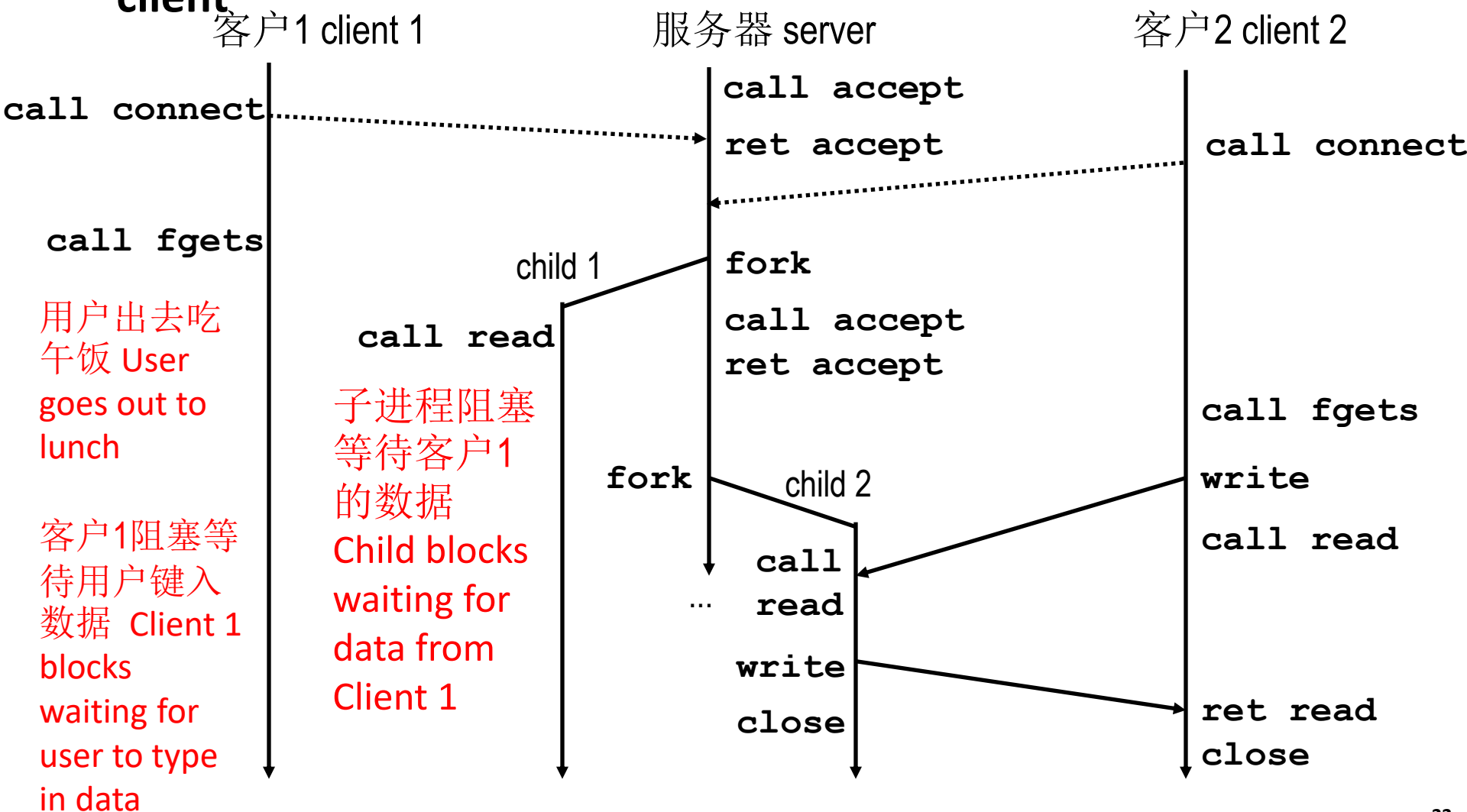


方法#1：基于进程的服务器



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

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!) */
    }
}
```

为何? Why?

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) {
            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



```
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)



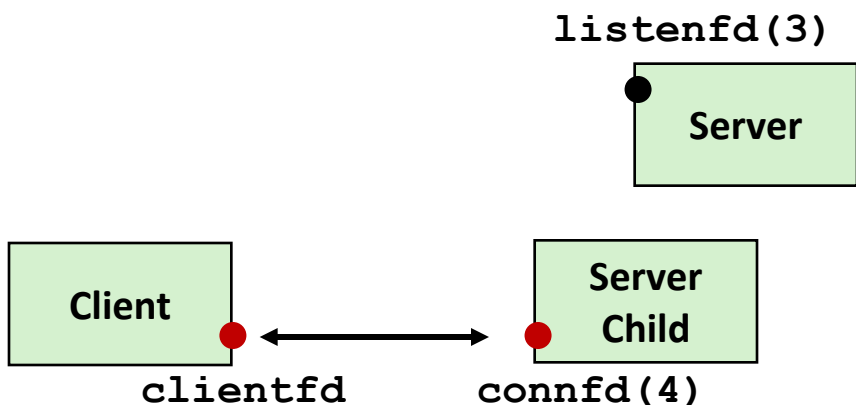
```
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

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

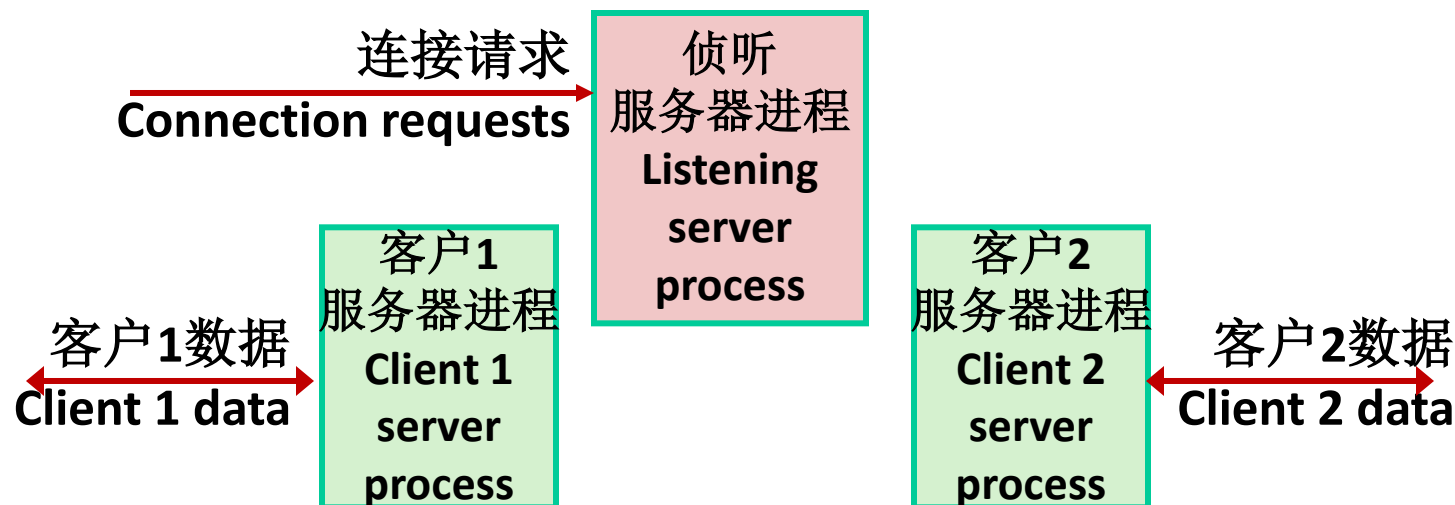
2. Client makes connection request by calling connect

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_storage clientaddr;

    listenfd = Open_listenfd(1);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, &clientaddr, &clientlen);
        if (Fork() == 0) {
            echo(connfd); /* Child receives client */
            Close(connfd); /* Parent closes connection with client */
            exit(0);
        }
    }
}
```

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_storage clientaddr;

    listenfd = Open_listenfd(1);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, &clientaddr, &clientlen);
        if (Fork() == 0) {
            echo(connfd); /* Child echoes client */
            Close(connfd); /* Parent closes connection with client */
            exit(0);
        }
    }
}
```



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



数据和服务

Read and service

活动描述符 Active Descriptors

listenfd = 3

connfd's		
0	10	活动 Active
1	7	
2	4	
3	-1	不活动 Inactive
4	-1	
5	12	活动 Active
6	5	
7	-1	从没使用 Never Used
8	-1	
9	-1	

发生了
什么事情?
**Anything
happened?**

挂起的输入 Pending Inputs

listenfd = 3

connfd's	
10	
7	←
4	
-1	
-1	
12	←
5	←
-1	
-1	
-1	

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

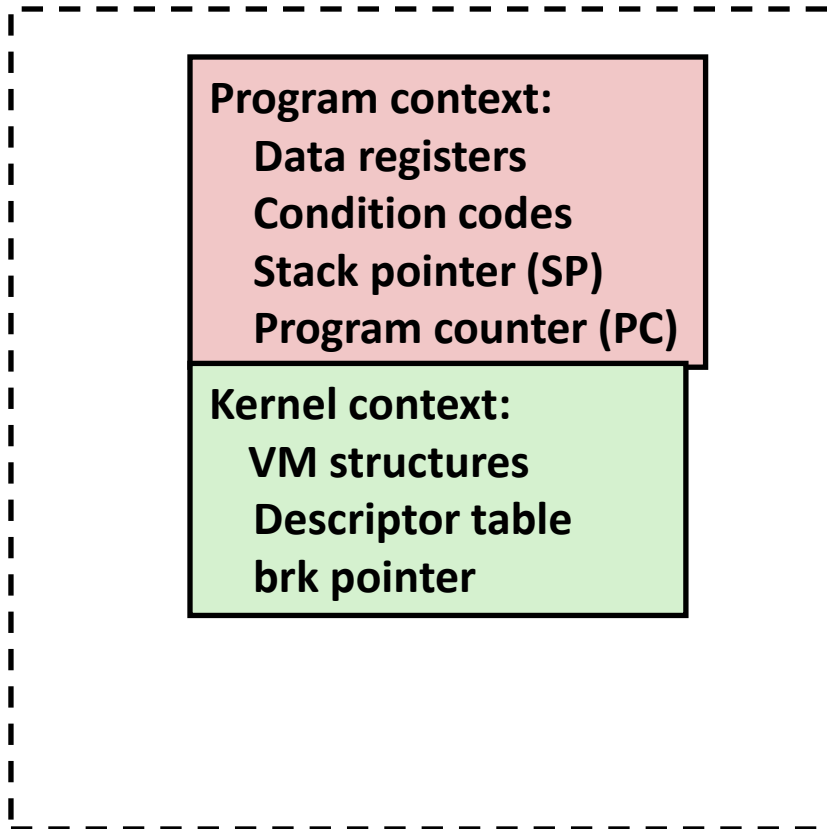
传统进程视图 Traditional View of a Process



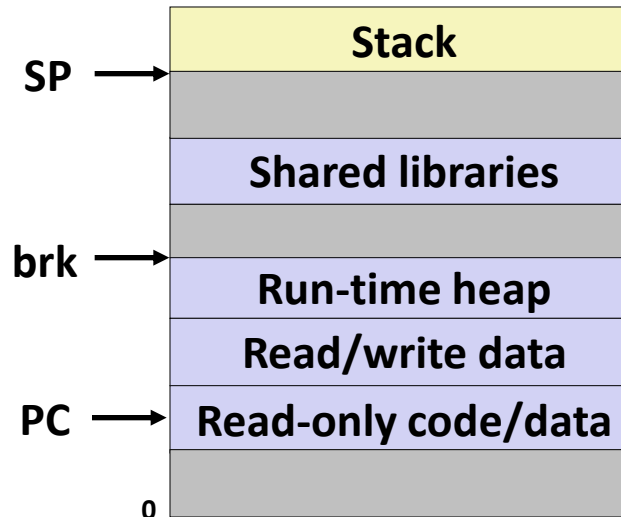
- 进程=进程上下文+代码、数据和栈 **Process = process context + code, data, and stack**

进程上下文

Process context



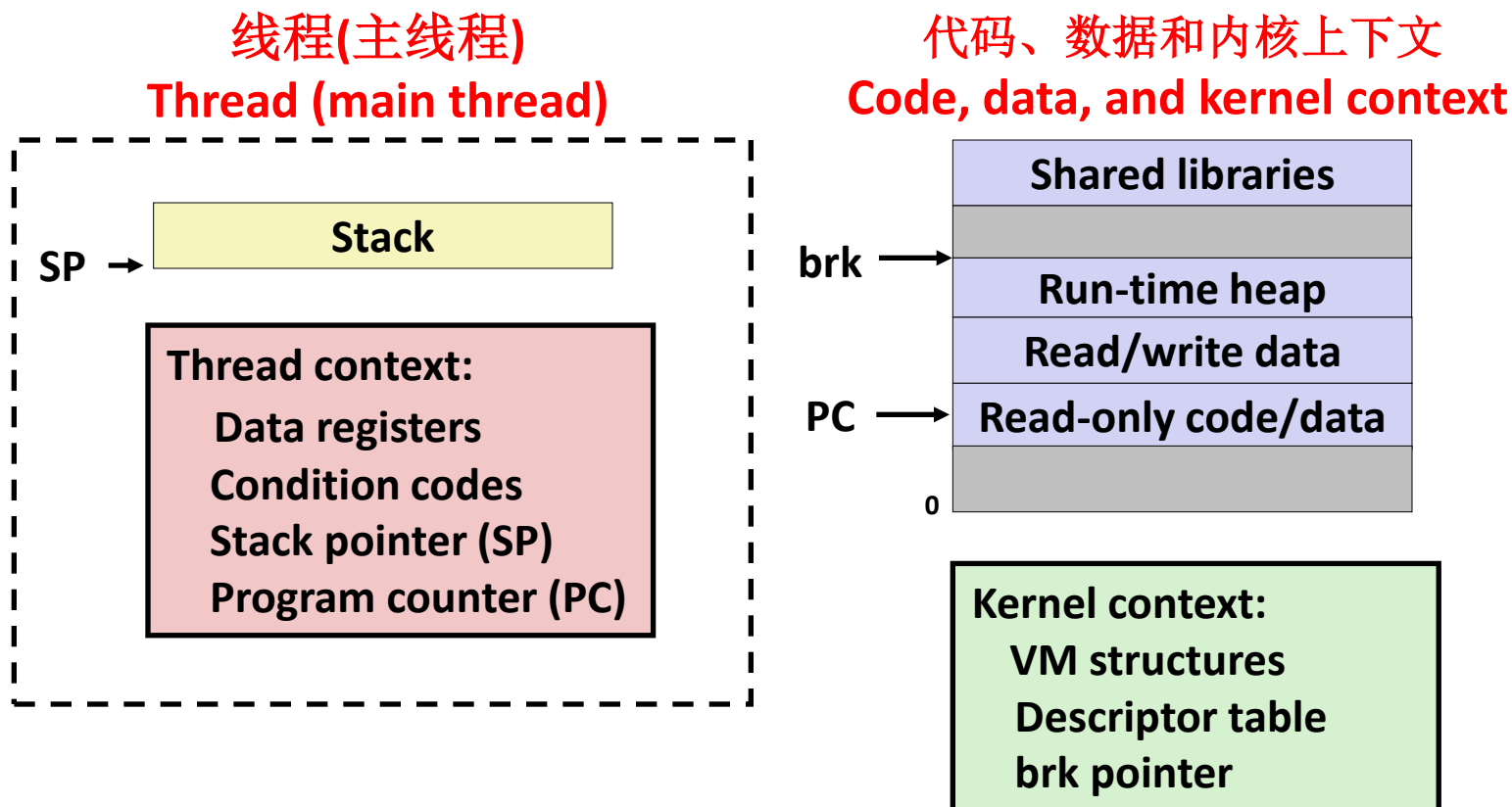
代码、数据和栈 **Code, data, and stack**



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1(主线程)

Thread 1 (main thread)

线程2(对等线程)

Thread 2 (peer thread)

共享代码和数据

Shared code and data

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
 SP_1
 PC_1

Thread 2 context:
Data registers
Condition codes
 SP_2
 PC_2

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
VM structures
Descriptor table
brk pointer

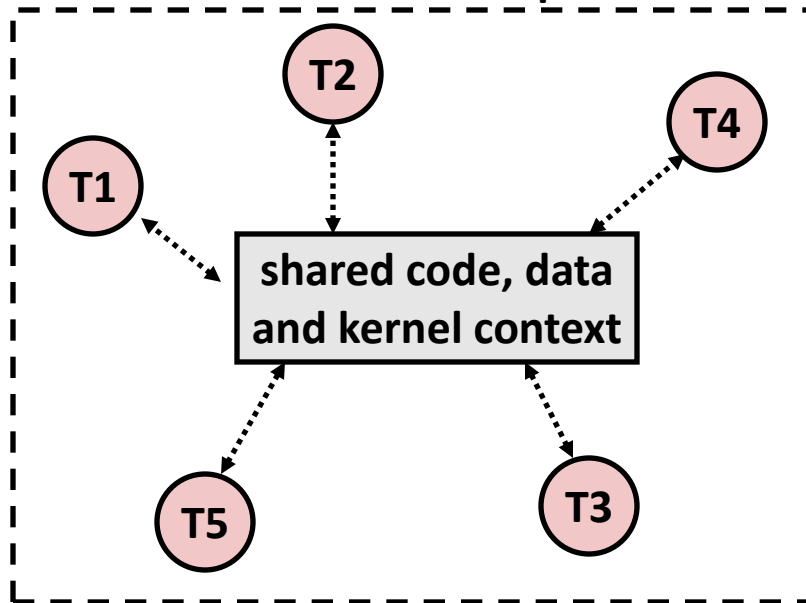
线程的逻辑视图 Logical View of Threads



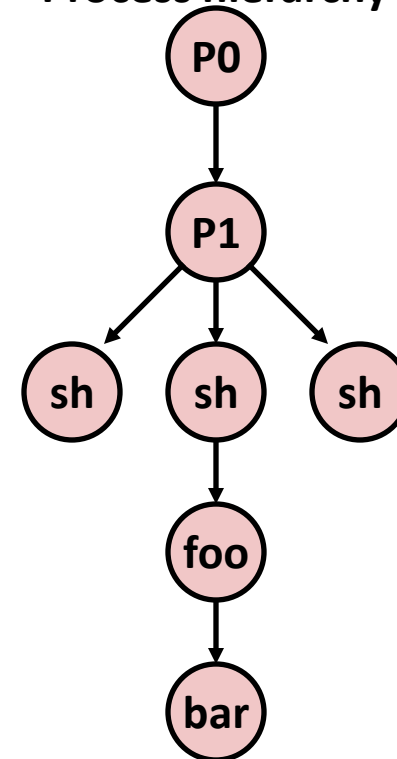
- 与进程关联的线程形成对等线程池 Threads associated with process form a pool of peers
 - 与进程形成层次树不同 Unlike processes which form a tree hierarchy

与进程foo关联的线程

Threads associated with process foo



进程层次结构
Process hierarchy





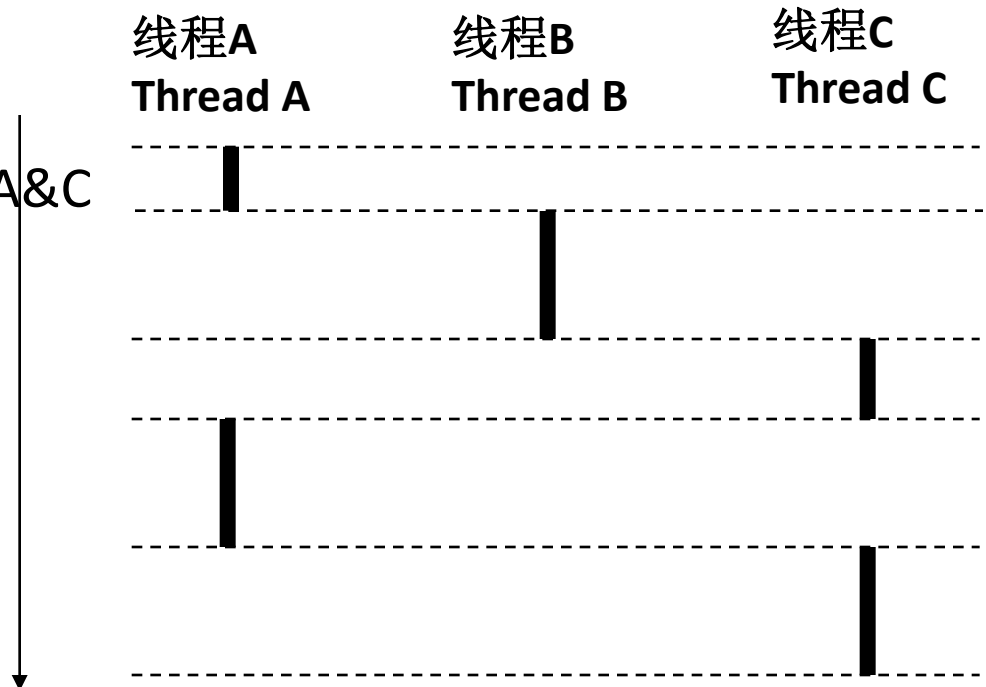
并发线程 Concurrent Threads

- 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

时间
Time

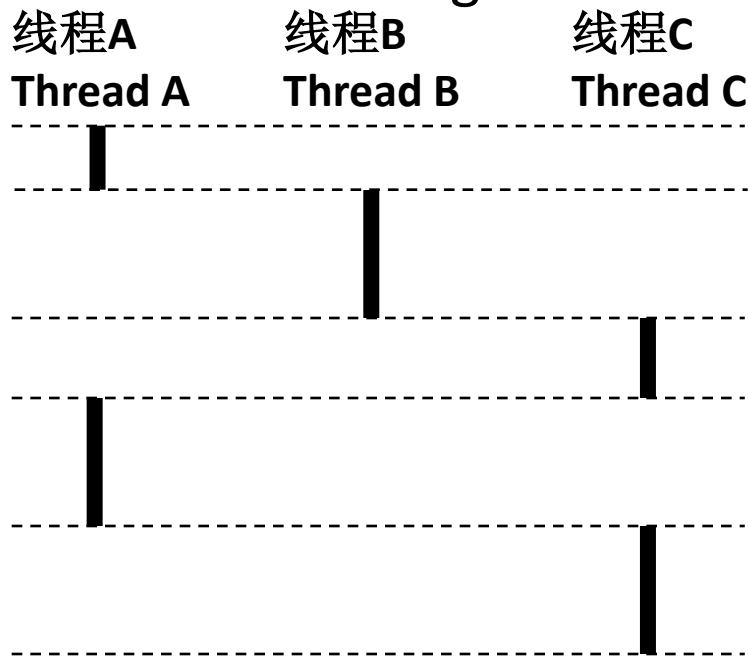




并发线程执行 Concurrent Thread Execution

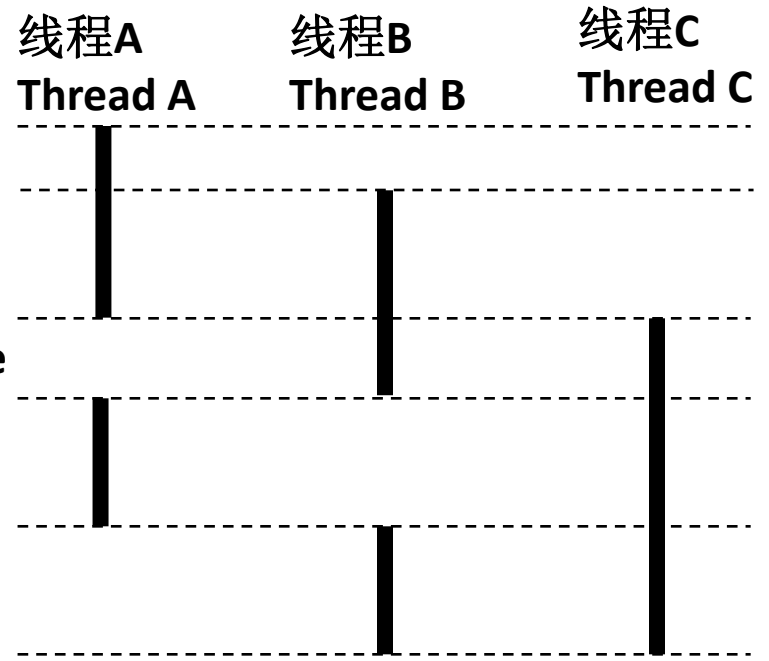
■ 单核处理器 Single Core Processor

- 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



时间 Time

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

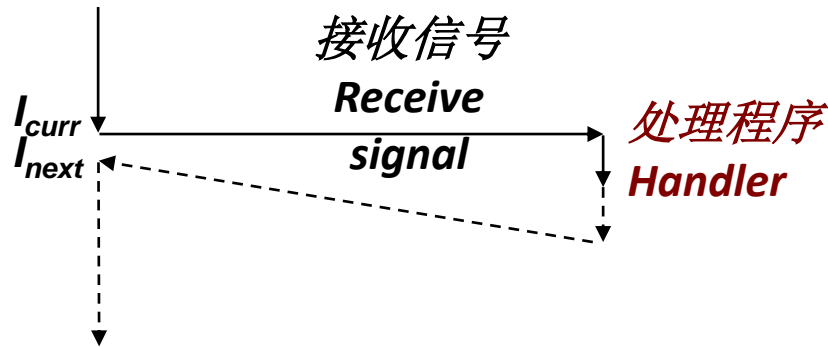


■ 线程和进程的区别 How threads and processes are different

- 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（Pthreads）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`

Pthreads的“hello, world”程序

The Pthreads "hello, world" Program



```
/*  
 * 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)

```
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”



主线程 Main thread

调用 `call Pthread_create()`

`Pthread_create()` returns 返回

调用 `call Pthread_join()`

主线程等待对等线程终止

Main thread waits for
peer thread to terminate

`Pthread_join()` returns 返回

`exit()`

终止主线程和任何对等线程

Terminates
main thread and
any peer threads

对等线程 Peer thread

`printf()`

`return NULL;`

对等线程终止
Peer thread
terminates

或者... Or, ...



主线程 Main thread

调用 `call Pthread_create()`

`Pthread_create()` returns 返回

调用 `call Pthread_join()`

主线程不需等待对等线程
终止 Main thread doesn't
need to wait for peer
thread to terminate

`Pthread_join()` returns 返回

`exit()`

终止主线程和任何对等线程
Terminates
main thread and
any peer threads

对等线程 Peer thread

`printf()`
`return NULL;`

对等线程终止
Peer thread
terminates

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;
}
```

echoserver.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)

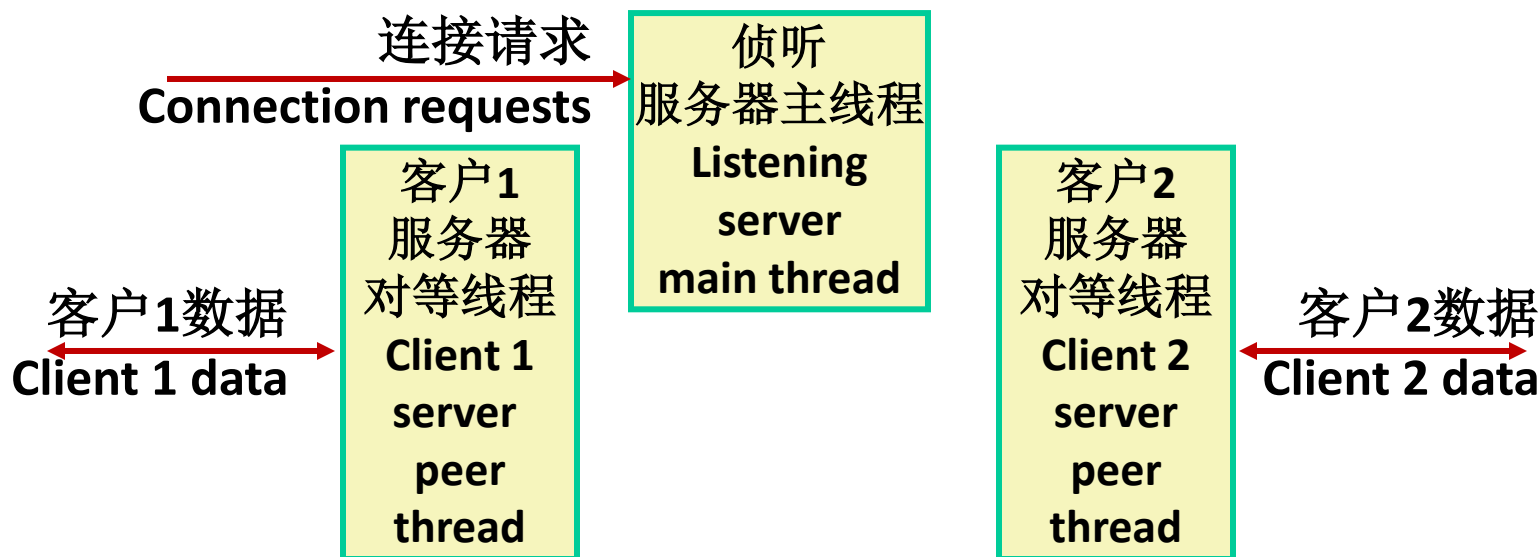


```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}                                     echoserv.c
```

- 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題



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題



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

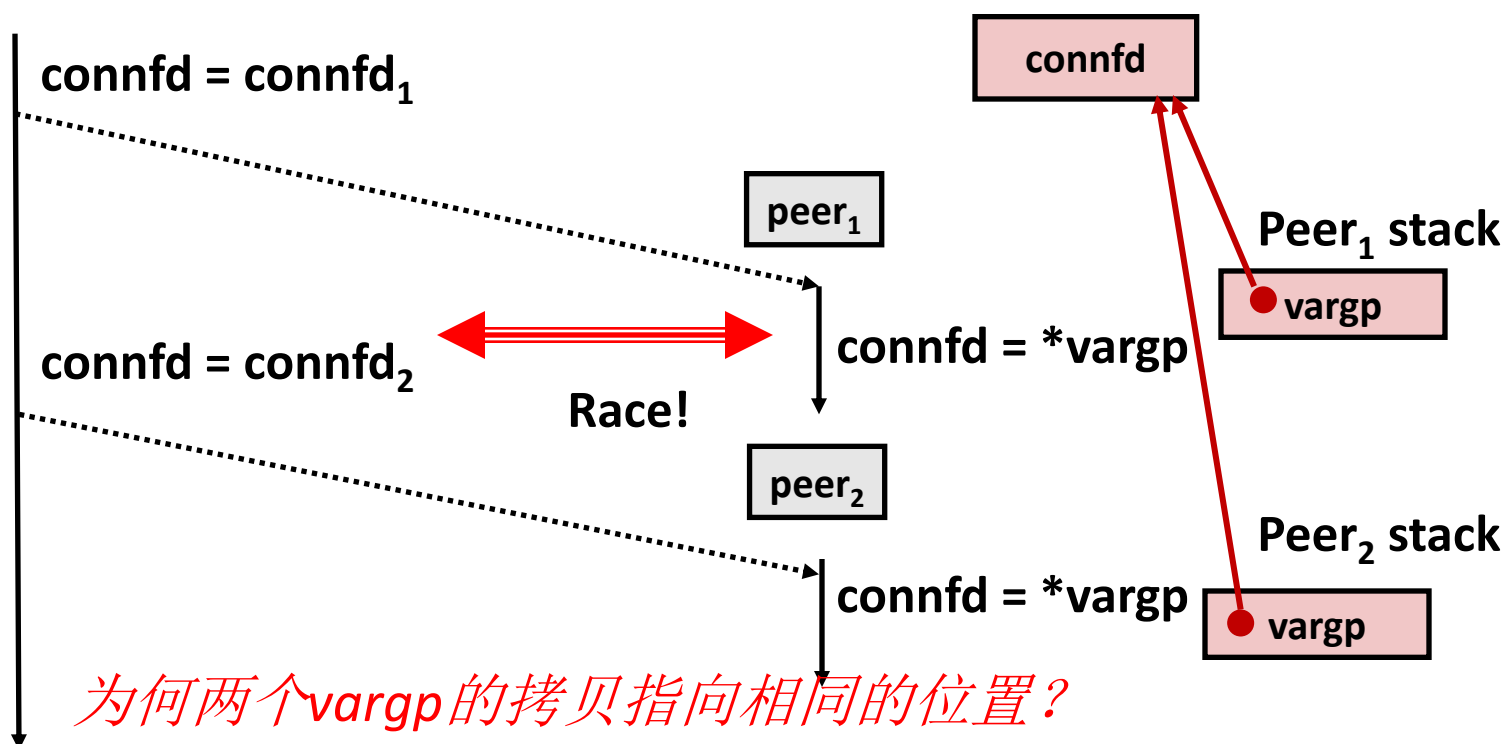
Potential Form of Unintended Sharing



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

主线程 main thread

主线程栈 Main thread stack



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)

线程1（主线程）

线程2（对等线程）

Thread 1 (main thread) Thread 2 (peer thread)

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
 SP_1
 PC_1

Thread 2 context:
Data registers
Condition codes
 SP_2
 PC_2

共享代码和数据

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
VM structures
Descriptor table
brk pointer

但是所有的内存都是共享的

But ALL memory is shared



Thread 1 context:
Data registers
Condition codes
 SP_1
 PC_1

Thread 2 context:
Data registers
Condition codes
 SP_2
 PC_2

线程1(主线程)

线程2(对等线程)

Thread 1 (main thread) Thread 2 (peer thread)

stack 1

stack 2

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
VM structures
Descriptor table
brk pointer

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

Thread 1 context:

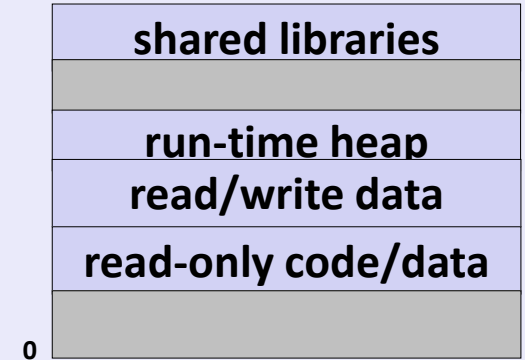
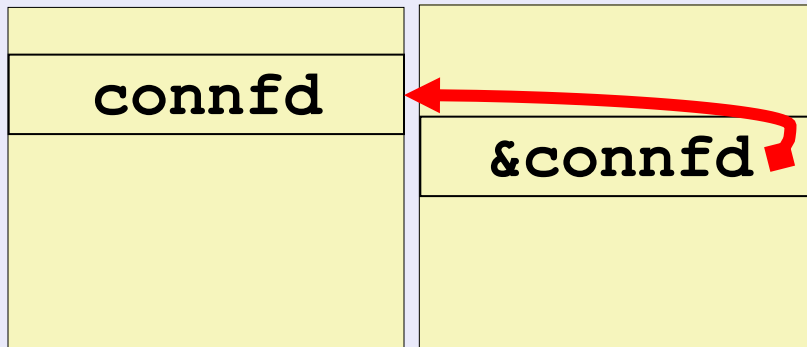
Data registers
Condition codes
 SP_1
 PC_1

Thread 2 context:

Data registers
Condition codes
 SP_2
 PC_2

Thread 1

Thread 2



Kernel context:
VM structures
Descriptor table
brk pointer


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

Thread 1 context:
Data registers
Condition codes
SP₁
PC₁

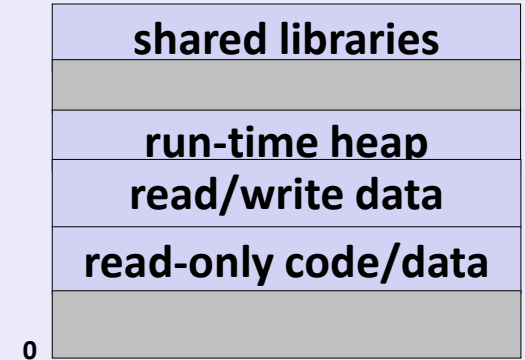
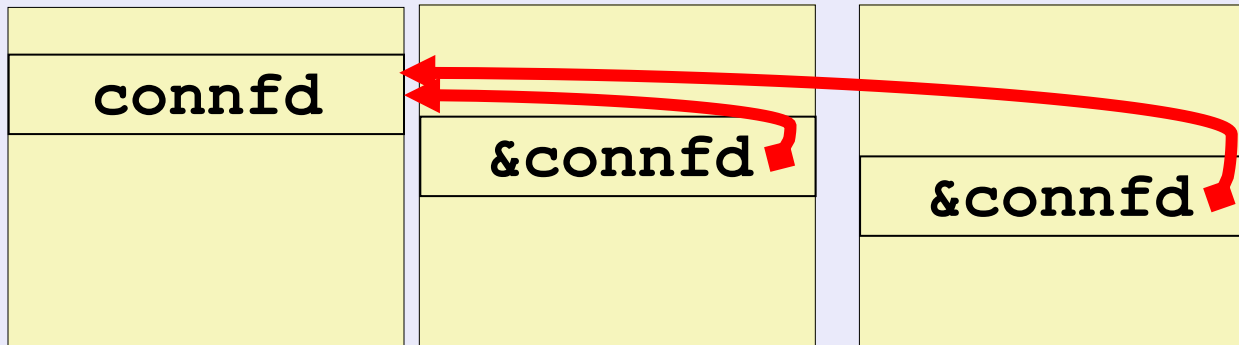
Thread 2 context:
Data registers
Condition codes
SP₂
PC₂

Thread 3 context:
Data registers
Condition codes
SP₂
PC₂

Thread 1

Thread 2

Thread 3



Kernel context:
VM structures
Descriptor table
brk pointer



Thread 1 context:
Data registers
Condition codes
 SP_1
 PC_1

Thread 2 context:
Data registers
Condition codes
 SP_2
 PC_2

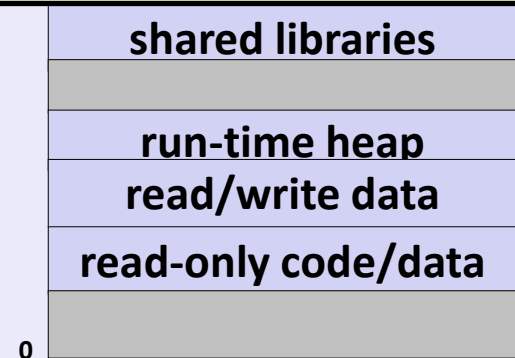
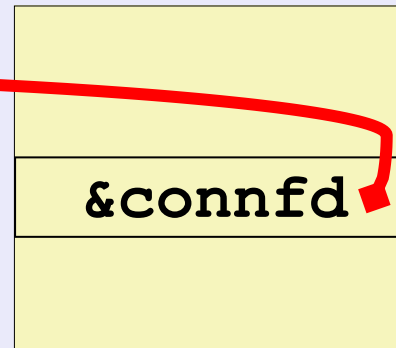
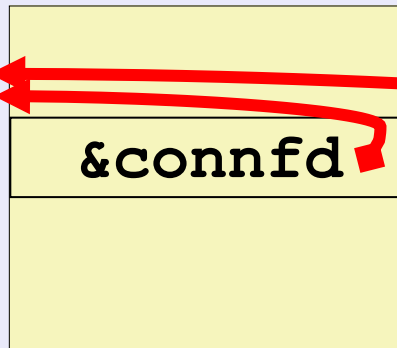
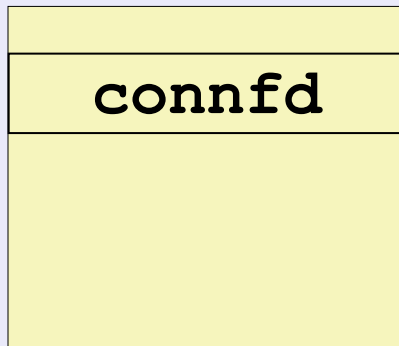
Thread 3 context:
Data registers
Condition codes
 SP_3
 PC_3

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

Thread 1

Thread 2

Thread 3



Kernel context:
VM structures
Descriptor table
brk pointer

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;
}
```

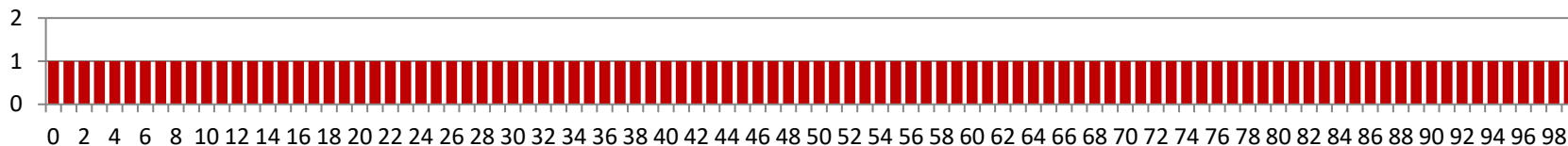
■ 竞争测试 Race Test

- 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

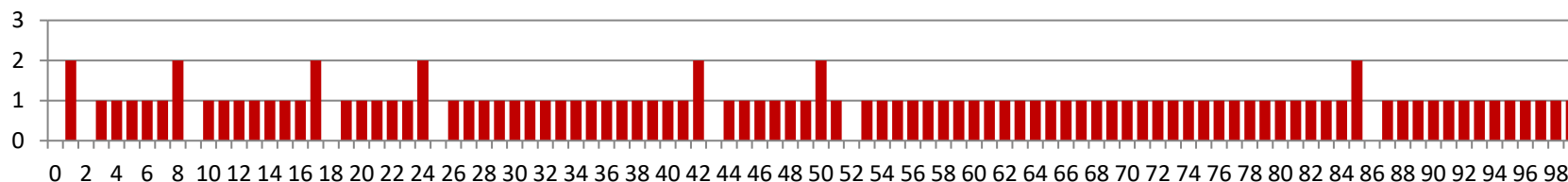
实验结果 Experimental Results



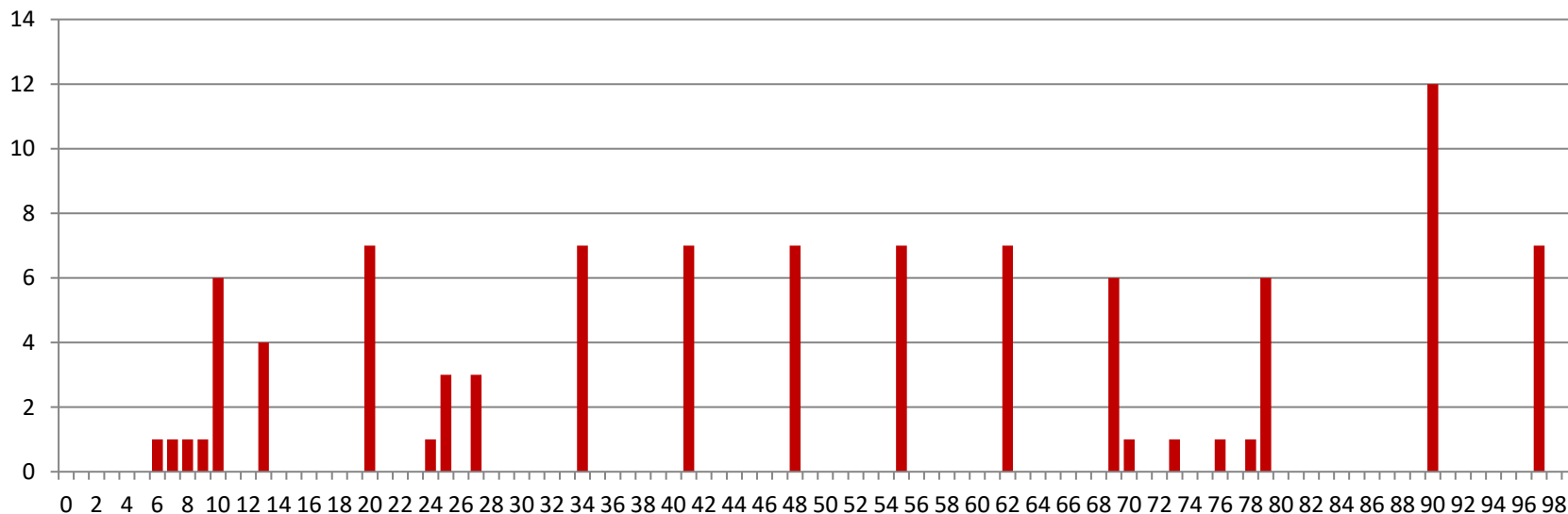
没有竞争 No Race



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