Concurrent Programming
并发编程

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. Bryant and David R. O’Hallaron

并发编程相对比较困难/Concurrent Programming is Hard!

- 人类思维更倾向于串行/The human mind tends to be sequential

- 关于时间的相关标识通常也有诸多误解/The notion of time is often misleading

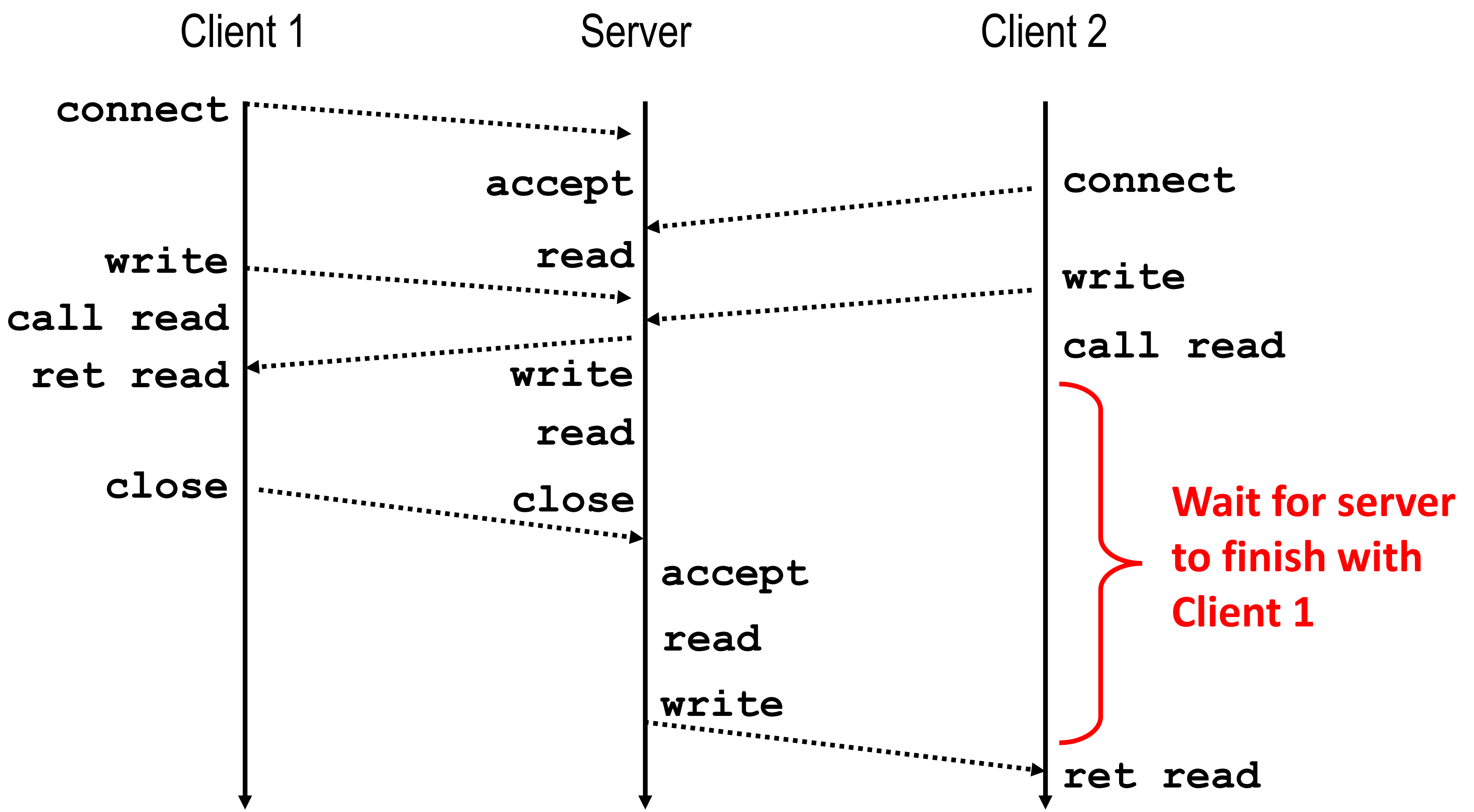
- 考虑系统中所有可能的事件序列是易于出错，更多时候也是不可能的/Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

并发编程相对比较困难/Concurrent Programming is Hard!

- 并发程序的常见问题/**Classical problem classes of concurrent programs:**
  - ***竞争/Races:*** 结果依赖于系统中调度策略/outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***死锁/Deadlock:*** 不恰当的资源分配会阻碍进程执行/improper resource allocation prevents forward progress
    - 例如交通阻塞/Example: traffic gridlock
  - ***活锁/饿死/公平/Livelock / Starvation / Fairness***: 外部事件或者系统调度决策会阻止子任务进展/external events and/or system scheduling decisions can prevent sub-task progress
    - 例如，总是有人在你前面插队/Example: people always jump in front of you in line
- 关于并发的许多内容不在本课程的讨论范围内/**Many aspects of concurrent programming are beyond the scope of our course..**
  - 但是，不是所有/but, not all ☺
  - 我们会覆盖部分内容/We'll cover some of these aspects in the next few lectures.

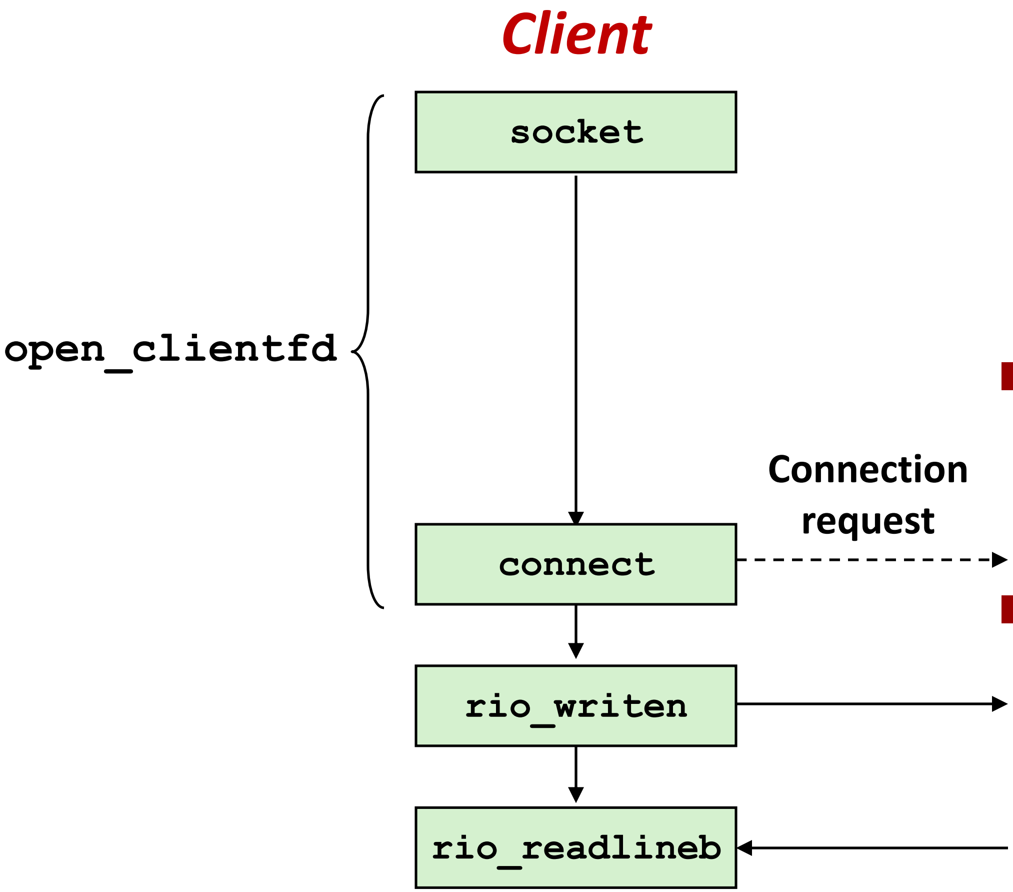
迭代服务器/Iterative Servers

- 迭代服务器每次处理一个请求/Iterative servers process one request at a time



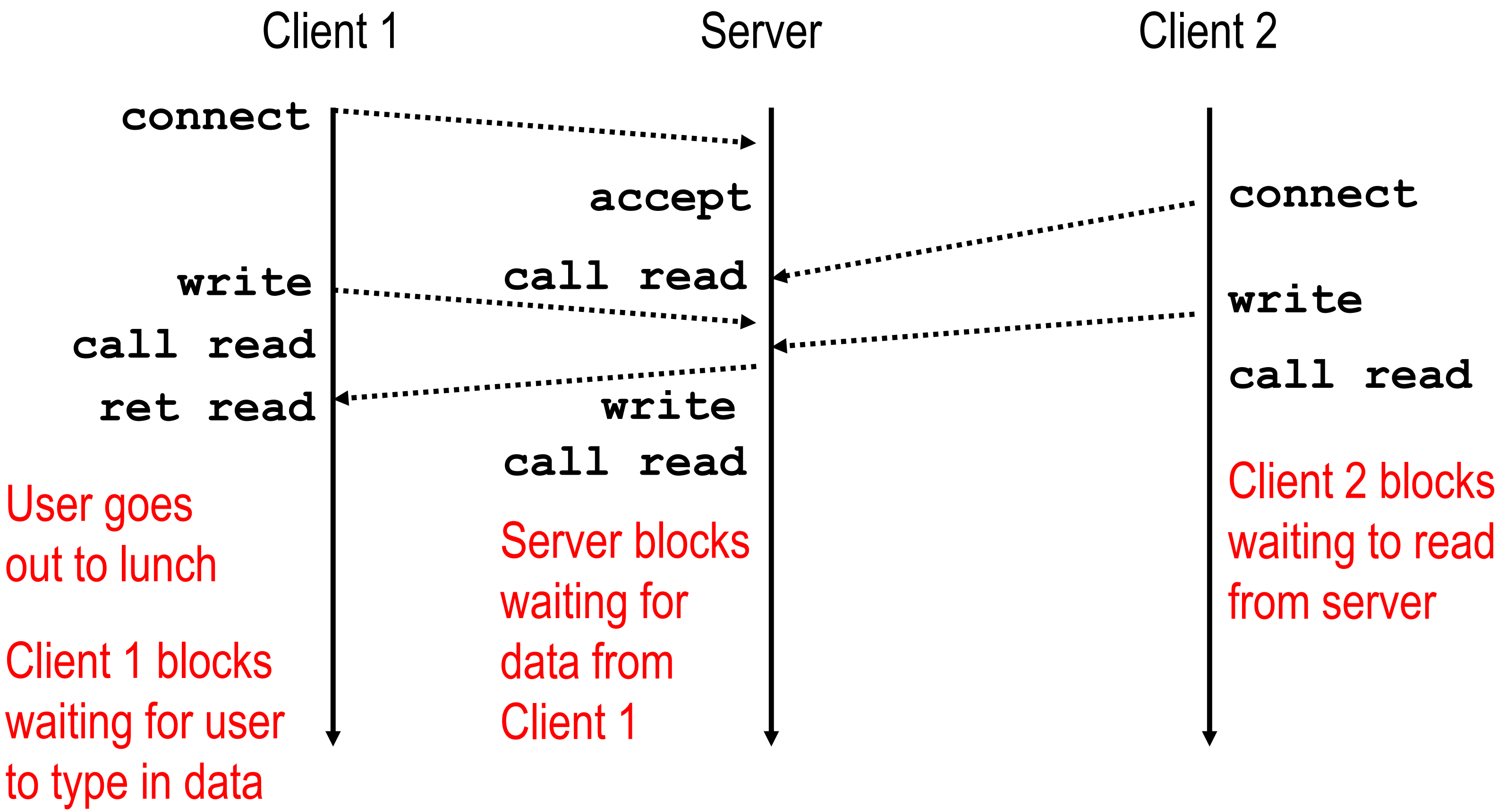
第二个客户端阻塞在哪里/Where Does Second Client Block?

- **第二个客户端尝试去链接服务器/Second client attempts to connect to iterative server**



- **Call to connect returns**
  - Even though connection not yet accepted
  - Server side TCP manager queues request
  - Feature known as “TCP listen backlog”
- **Call to rio_writen returns**
  - Server side TCP manager buffers input data
- **Call to rio_readlineb blocks**
  - Server hasn’t written anything for it to read yet.

迭代服务器的主要缺陷/Fundamental Flaw of Iterative Servers



- 方案：使用并发服务器替代/**Solution: use *concurrent servers* instead**
  - 并发服务器使用多个并发流同时处理多个客户端/Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

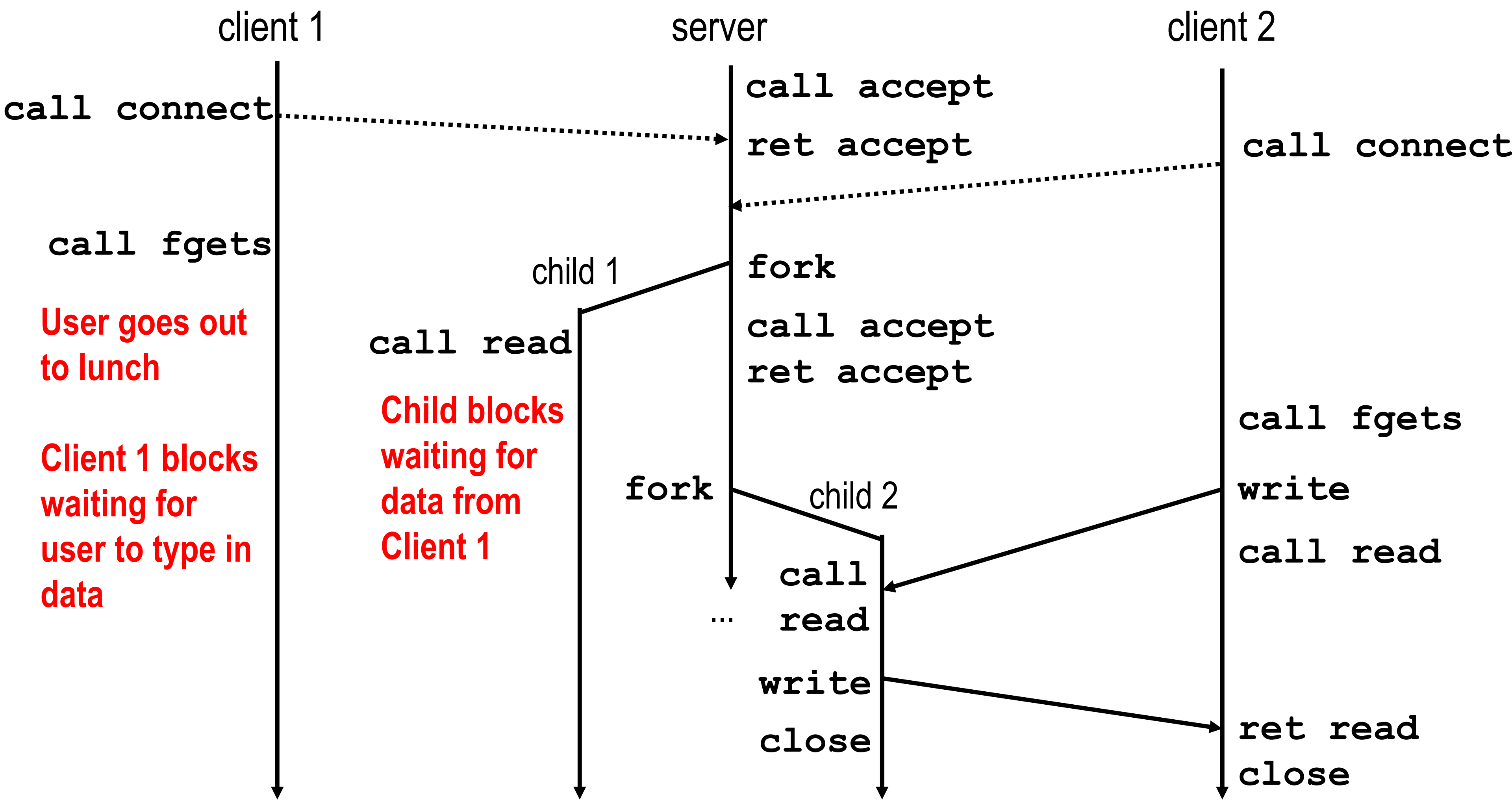
实现并发服务器的方案/Approaches for Writing Concurrent Servers

服务器可以并发处理多个客户端/Allow server to handle multiple clients concurrently

1. **基于进程的/Process-based**
   - 内核自动调度多个逻辑流/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. **基于事件的/Event-based**
   - 程序员手动管理控制流/Programmer manually interleaves multiple logical flows
   - 所有流共享同一个地址空间/All flows share the same address space
   - 通过I/O多路复用技术/Uses technique called *I/O multiplexing*.
3. **基于线程的/Thread-based**
   - 内核自动调度逻辑流/Kernel automatically interleaves multiple logical flows
   - 每个流共享同样的地址空间/Each flow shares the same address space
   - 基于线程和基于事件的混合/Hybrid of of process-based and event-based.

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

- 为每个客户端生成独立的进程/Spawn separate process for each client



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!) */
    }
}
```

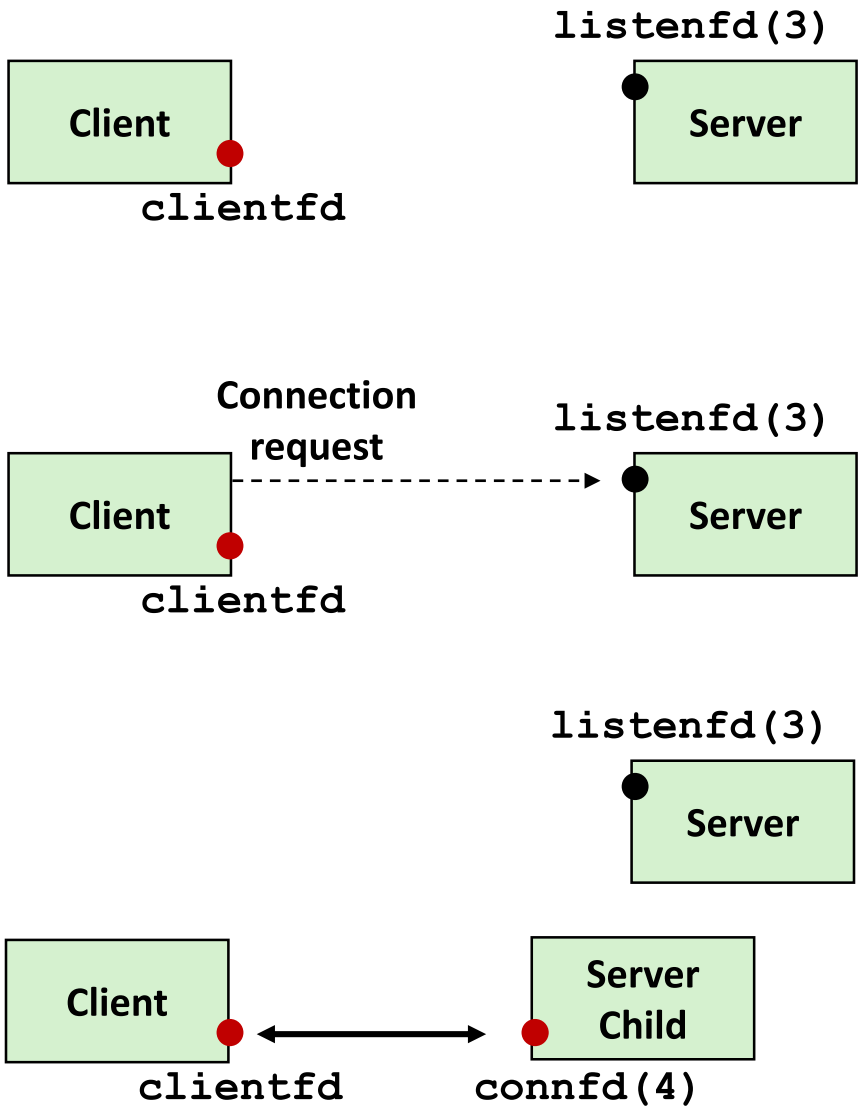
echoserverp.c

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

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

- 回收所有僵尸子进程/Reap all zombie children

并发服务器:accept过程/Concurrent Server: `accept` Illustrated

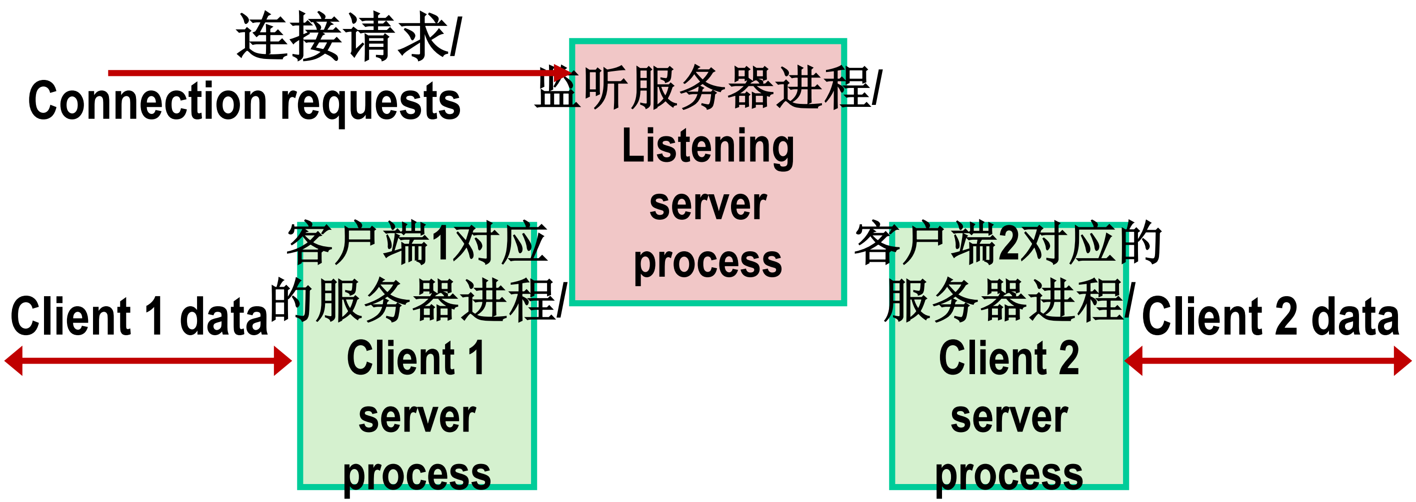


1. 服务器在accept中阻塞，并在描述符listenfd上等待监听/Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. 客户端通过connect发起连接/Client makes connection request by calling `connect`

3. 服务器端从accept返回connfd，并fork子进程处理客户端。连接在clientfd和connfd之间建立/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负责/Each client handled by independent child process
- 他们之间无共享状态/No shared state between them
- 父进程和子进程都有listenfd和connfd/Both parent & child have copies of listenfd and connfd
  - 父进程必须关闭connfd/Parent must close `connfd`
  - 子进程必须关闭listenfd/Child should close `listenfd`

基于进程的服务器问题/Issues with Process-based Servers

- **监听服务器进程必须回收僵尸子进程/Listening server process must reap zombie children**
  - 为了避免造成严重的内存泄露/to avoid fatal memory leak
- **父进程必须关闭connfd副本/Parent process must `close` its copy of `connfd`**
  - 内核对打开的文件和socket进行引用计数/Kernel keeps reference count for each socket/open file
  - fork后引用计数加1/After fork, `refcnt(connfd) = 2`
  - 引用计数为0时连接才会关闭/Connection will not be closed until `refcnt(connfd) = 0`

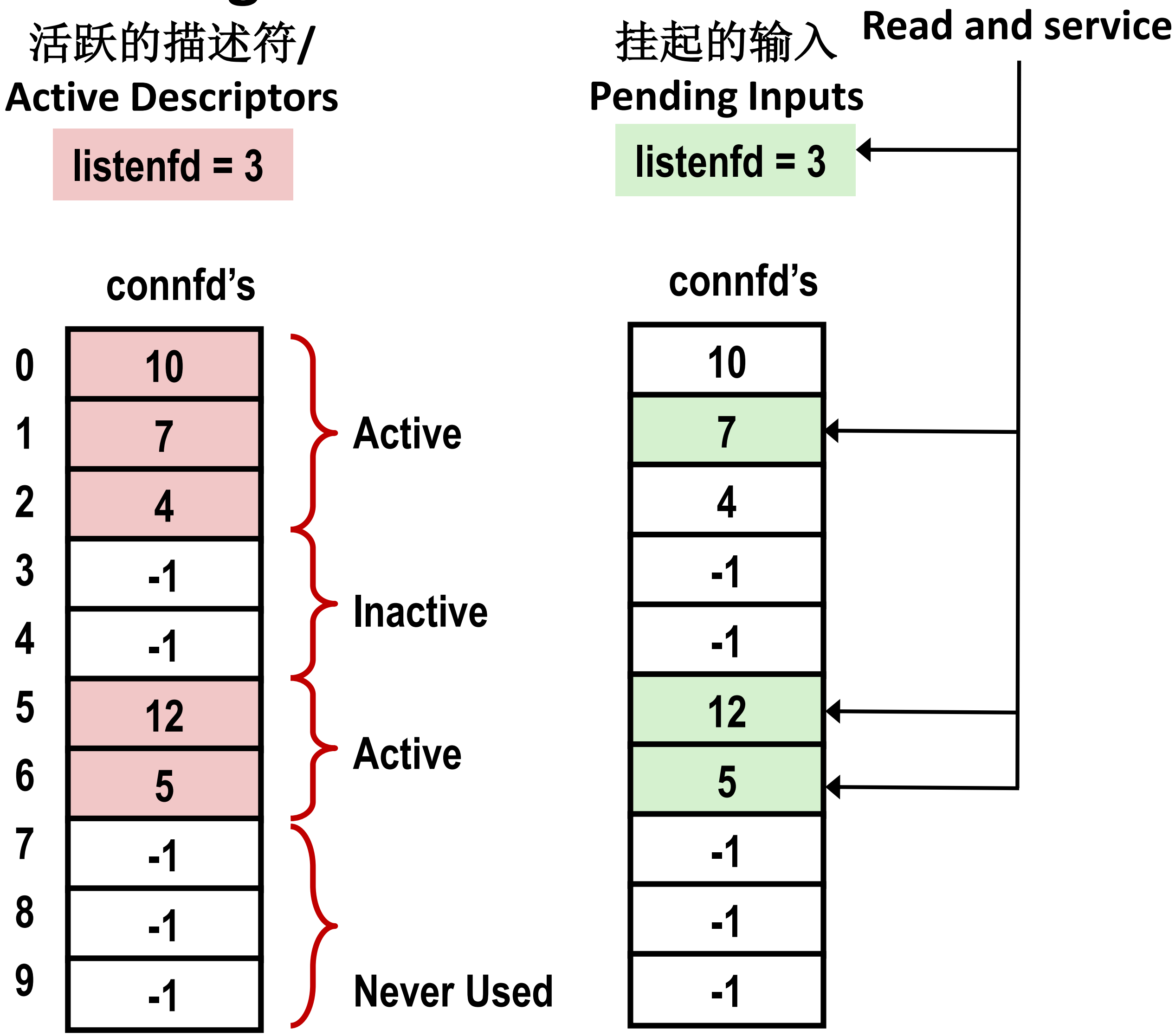
基于进程的服务器优缺点/Pros and Cons of Process-based Servers

- **+ 可以并发处理多个连接/Handle multiple connections concurrently**
- **+ 共享模型比较清晰/Clean sharing model**
  - 描述符/descriptors (no)
  - 文件表/file tables (yes)
  - 全局变量/global variables (no)
- **+简单并直观/ Simple and straightforward**
- **– 进程控制引入额外开销/Additional overhead for process control**
- **– 进程之间共享数据比较麻烦/Nontrivial to share data between processes**
  - 需要进程间交互机制/Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes)，System V shared memory and semaphores

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

- **服务器维护了一个活跃连接集合/Server maintains set of active connections**
  - connfd数组/Array of connfd's
- **重复以下步骤/Repeat:**
  - 检查哪个描述符有等待的输入/Determine which descriptors (connfd's or listenfd) have pending inputs
    - 例如使用select或者epoll函数/e.g., using `select` or `epoll` functions
    - 输入挂起是一个事件/arrival of pending input is an *event*
  - 如果listenfd 有输入，则accept连接/If listenfd has input, then `accept` connection
    - 将新的connfd添加到数组中/and add new connfd to array
  - 对所有有挂起输入的connfd进行处理/Service all connfd's with pending inputs
- **详见教材/Details for select-based server in book**

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



基于事件的服务器的优缺点/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.**
- **+ 可以进行单步调试/Can single-step with a debugger.**
- **+ 没有进程或者线程控制开销/No process or thread control overhead.**
  - 是许多高性能Web服务器和搜索引擎采用的方案/Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado

- **– 与基于进程或线程的设计相比代码更加复杂/Significantly more complex to code than process- or thread-based designs.**
- **– 难以实现细粒度并行/Hard to provide fine-grained concurrency**
  - 例如，如何处理部分HTTP请求头/E.g., how to deal with partial HTTP request headers
- **– 无法利用多核的优势/Cannot take advantage of multi-core**
  - 单线程控制/Single thread of control

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

- 与方案1非常类似（基于进程的）**Very similar to approach #1 (process-based)**
  - 但是使用线程替代了进程/…but using threads instead of processes

传统进程视图/Traditional View of a Process

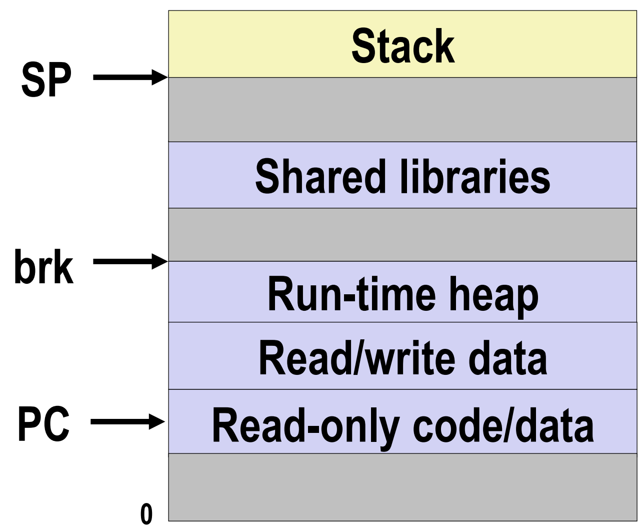
- **Process = process context + code, data, and stack**

进程上下文/
Process context

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

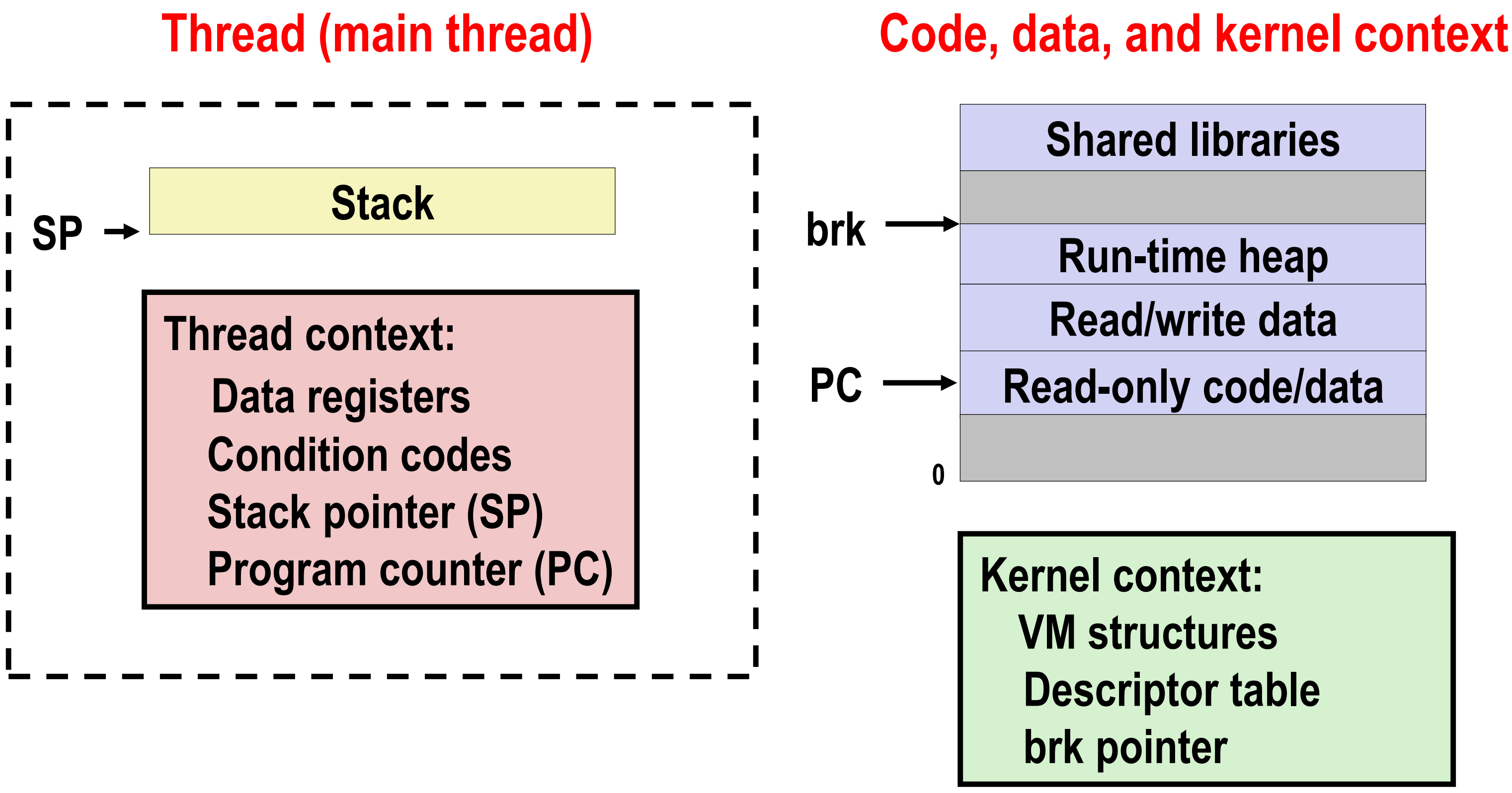
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



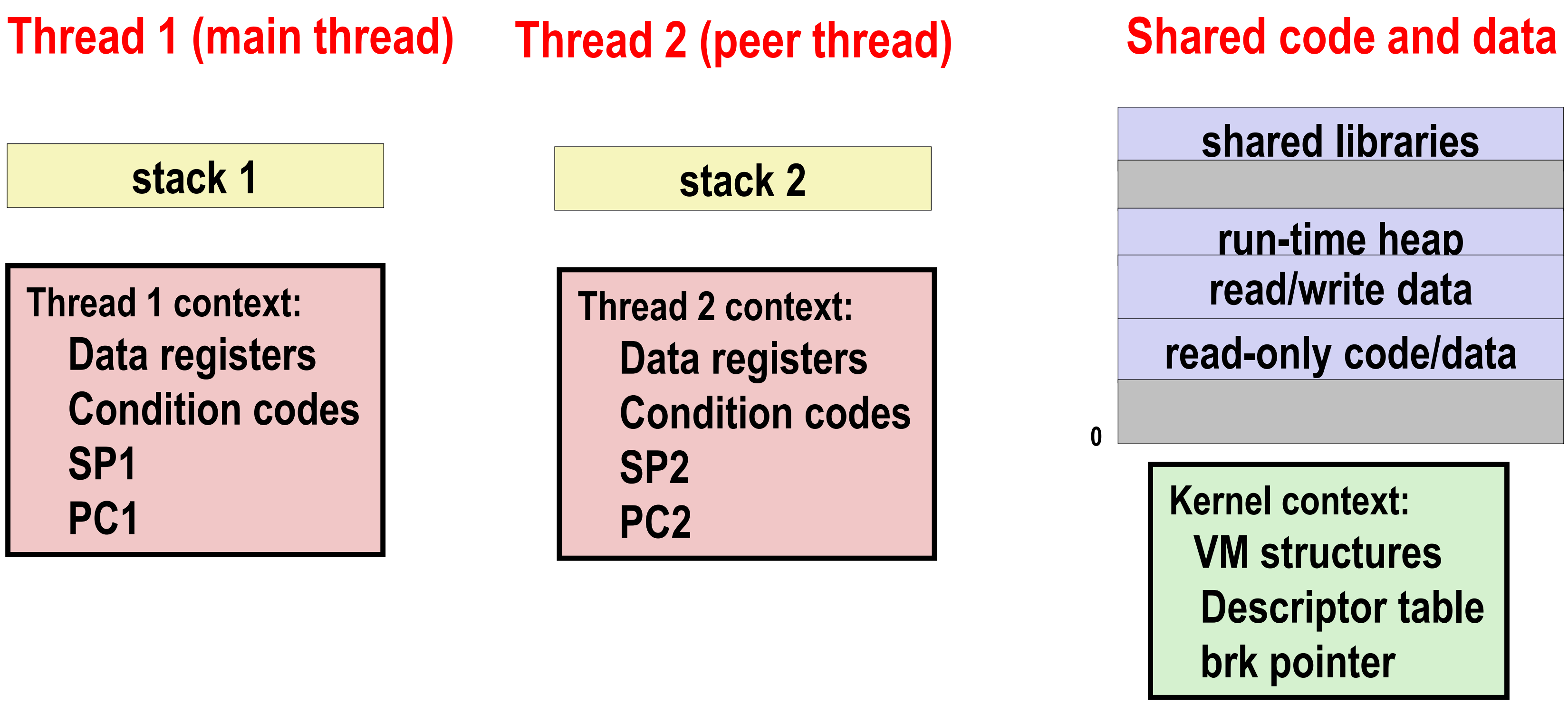
进程的另一个视图/Alternate View of a Process

- Process = thread + code, data, and kernel context



多线程进程/A Process With Multiple Threads

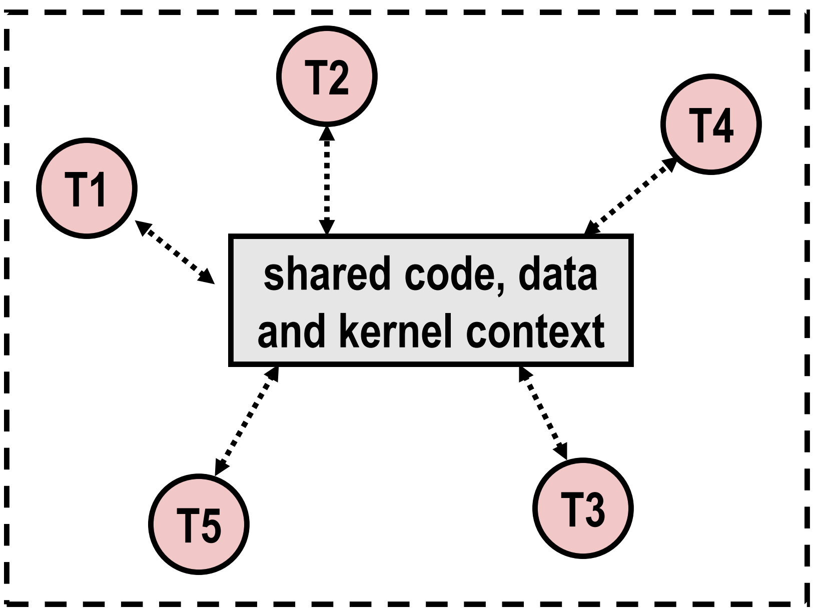
- 一个进程可以有多个线程/**Multiple threads can be associated with a process**
  - 每个线程有自己的逻辑控制结构/Each thread has its own logical control flow
  - 每个线程共享同样的代码、数据和内核上下文/Each thread shares the same code, data, and kernel context
  - 每个线程有局部变量的本地栈/Each thread has its own stack for local variables
    - 但是不受保护/but not protected from other threads
  - 每个线程有自己的ID/Each thread has its own thread id (TID)



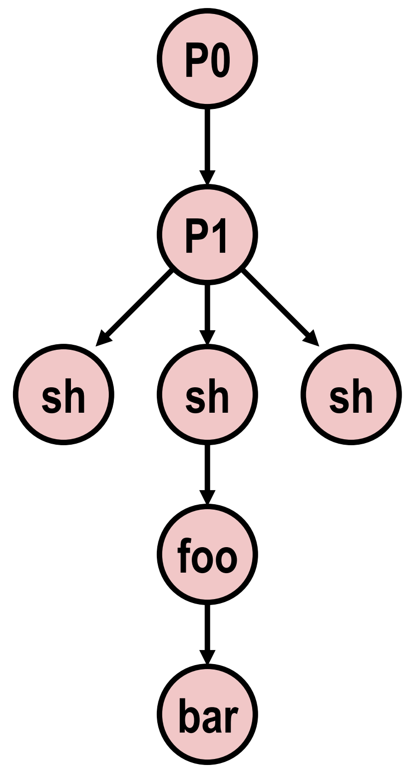
线程逻辑视图/Logical View of Threads

- **与一个进程关联的所有线程构成一个对等线程池 /Threads associated with process form a pool of peers**
  - 而不像进程形成一个树状层次结构/Unlike processes which form a tree hierarchy

与进程foo相关的线程/Threads associated with process foo



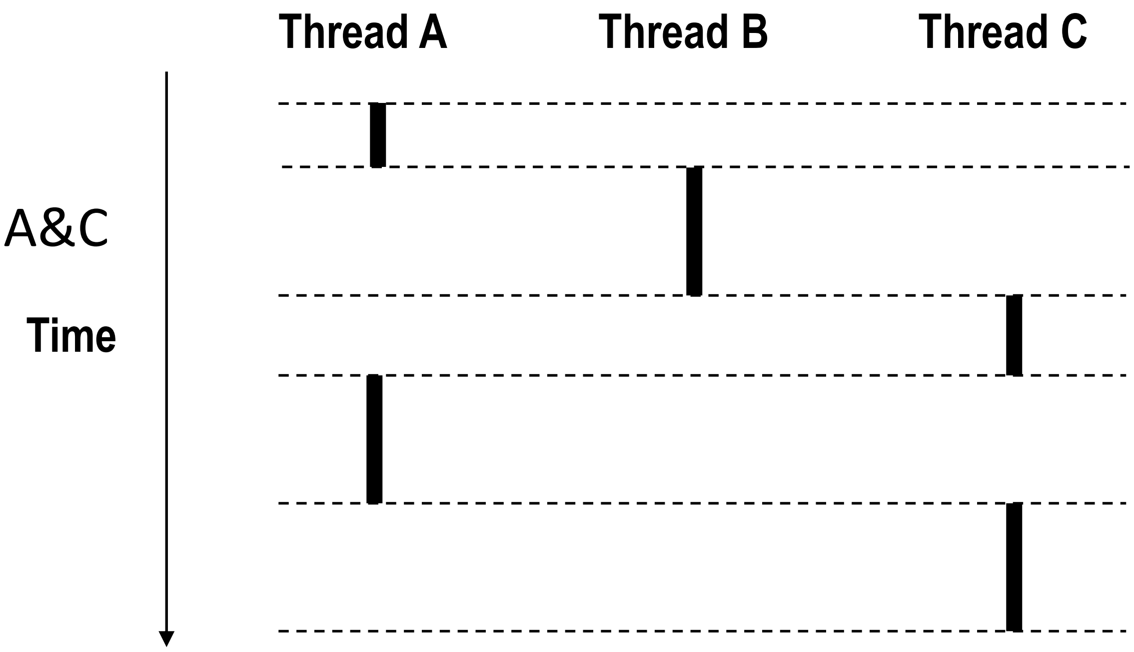
进程层次结构/Process hierarchy



并发线程/Concurrent Threads

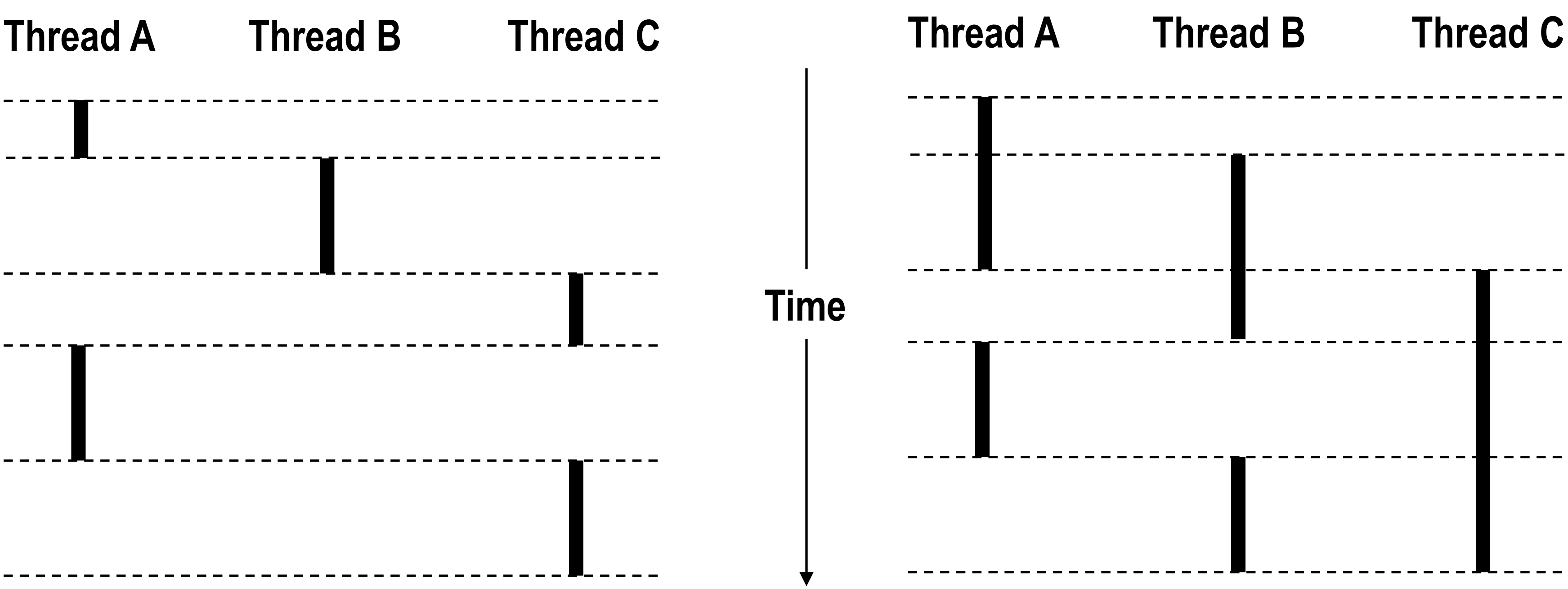
- 如果两个线程在执行时间上有重叠就是并发的/Two threads are *concurrent* if their flows overlap in time
- 否则，他们就是串行的/Otherwise, they are sequential

- 例如/Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



并发线程执行/Concurrent Thread Execution

- **单核处理器/Single Core Processor**
  - 通过时分模拟并行/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



Run 3 threads on 2 cores

线程 vs. 进程 /Threads vs. Processes

- **线程和进程有哪些相似点/How threads and processes are similar**
  - 都有自己的逻辑控制流/Each has its own logical control flow
  - 都可以相互并发运行（有可能在不同的核上）/ Each can run concurrently with others (possibly on different cores)
  - 都需要上下文切换/Each is context switched
- **线程和进程的不同点/How threads and processes are different**
  - 线程之间共享代码和数据（除了本地栈）Threads share all code and data (except local stacks)
    - 进程不这样做/Processes (typically) do not
  - 线程比进程开销更小一些/Threads are somewhat less expensive than processes
    - 进程控制的开销是线程控制的两倍左右/Process control (creating and reaping) twice as expensive as thread control
    - Linux的数值/Linux numbers:
      - 创建和回收进程大概需要~20K 周期/~20K cycles to create and reap a process
      - 创建和回收线程大概需要~10K 周期/~10K cycles (or less) to create and reap a thread

Posix Threads 接口/Posix Threads (Pthreads) Interface

- ***Pthreads:*** **在C程序中对线程进行操作和控制的60多个标准接口/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()` [terminates all threads] , `RET` [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`

基于Pthreads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()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

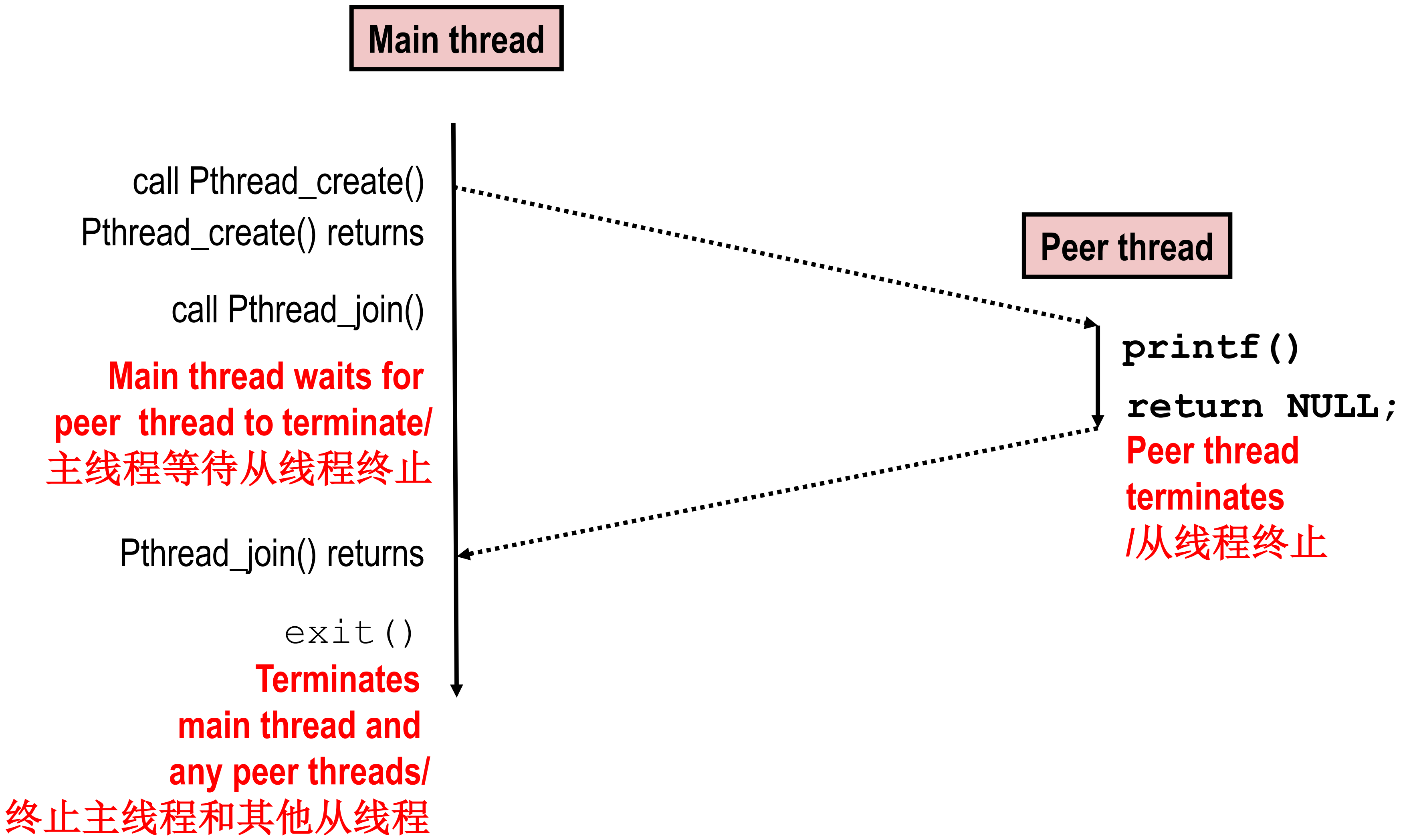
Thread arguments (void *p)

Return value (void **p)

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

hello.c

运行线程化的“hello, world” /Execution of Threaded “hello, world”



基于线程的回声服务器/Thread-Based Concurrent Echo Server

```c
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);
    }
}
```

echoservert.c

- 需要用**malloc**分配连接描述符以避免竞争/**malloc** of connected descriptor necessary to avoid deadly race (later)

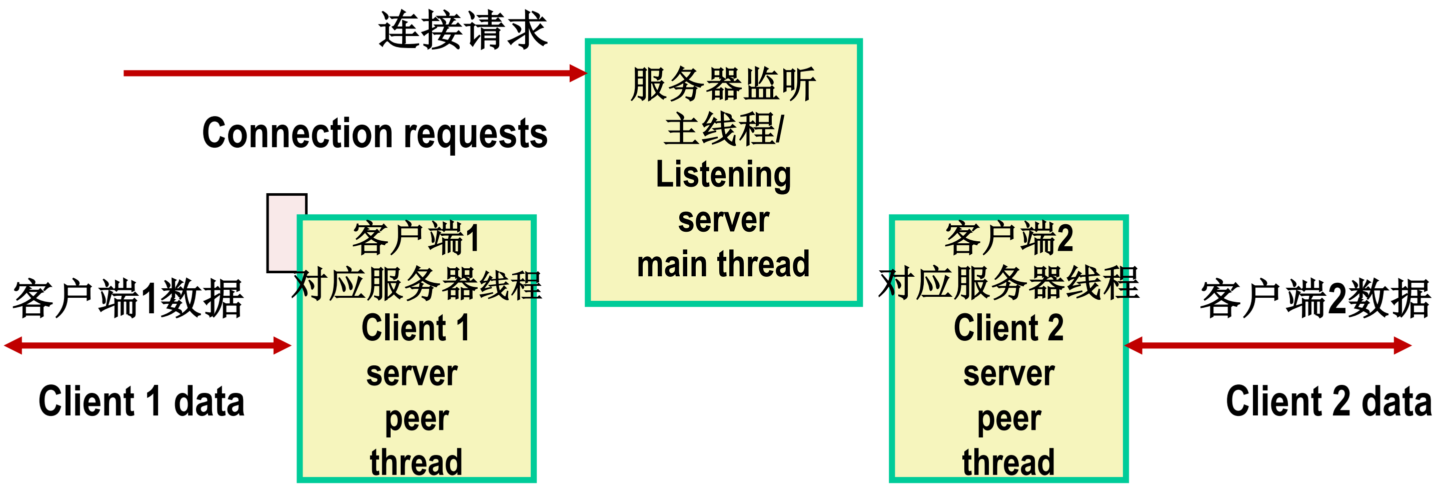
基于线程的并发服务器/Thread-Based Concurrent Server (cont)

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

echoservert.c

- 以分离式运行线程/Run thread in “detached” mode.
  - 独立于其他线程运行/Runs independently of other threads
  - 终止后由内核自动回收/Reaped automatically (by kernel) when it terminates
- 释放持有connfd的存储/Free storage allocated to hold `connfd`.
- 关闭connfd(非常重要)/Close `connfd` (important!)

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



- 每个客户端都有一个独立的线程对应/Each client handled by individual peer thread
- 线程共享除了TID之外的进程状态/Threads share all process state except TID
- 每个线程都有存储局部变量的独立堆栈/Each thread has a separate stack for local variables

基于线程的服务器的一些问题/Issues With Thread-Based Servers

- **必须以“分离式”运行以避免内存泄露/Must run “detached” to avoid memory leak**
  - 任何一个时间点，一个线程是可加入或者分离的/At any point in time, a thread is either *joinable* or *detached*
  - *可加入的线程是可以回收或者被其他线程杀死的/Joinable* thread can be reaped and killed by other threads
    - 必须使用`pthread_join`回收以释放内存资源/must be reaped (with `pthread_join`) to free memory resources
  - *分离的线程不能回收或者被其他线程杀死/Detached* thread cannot be reaped or killed by other threads
    - 终止后自动回收资源/resources are automatically reaped on termination
  - 默认是可加入的/Default state is joinable
    - 用特定函数变为分离式/use `pthread_detach(pthread_self())` to make detached
- **必须格外小心以避免意外的共享/Must be careful to avoid unintended sharing**
  - 例如，将指针传递给主线程的栈/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必须是线程安全的/All functions called by a thread must be *thread-safe***
  - (next lecture)

基于线程设计的优缺点/Pros and Cons of Thread-Based Designs

- **+方便在线程间共享数据结构/+ Easy to share data structures between threads**
  - 例如登录信息和文件缓存/e.g., logging information, file cache
- **+线程比进程更加高效/+ Threads are more efficient than processes**

- **-无意的共享可能会导致难以觉察和复现的错误/– 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
  - 难以获知哪些数据是共享的哪些是私有的/Hard to know which data shared & which private
  - 难以通过测试发现问题/Hard to detect by testing
    - 竞争复现的概率非常低/Probability of bad race outcome very low
    - 但是不是0概率/But nonzero!
  - 后续将进行介绍/Future lectures

总结：并发方法/Summary: Approaches to Concurrency

- **进程级并行/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
  - 创建和销毁的开销大/High overhead in adding/removing clients
- **基于事件的/Event-based**
  - 低级繁琐/Tedious and low level
  - 可以超越调度进行控制/Total control over scheduling
  - 开销极低/Very low overhead
  - 无法实现细粒度并发/Cannot create as fine grained a level of concurrency
  - 无法利用多核/Does not make use of multi-core
- **线程级并行/Thread-based**
  - 容易共享资源：也许太容易了/Easy to share resources: Perhaps too easy
  - 开销中等/Medium overhead
  - 不能超越调度策略进行控制/Not much control over scheduling policies
  - 难以调试/Difficult to debug
    - 事件排序无法重复/Event orderings not repeatable