并发编程

Concurrent Programming

并发编程是极具挑战性的

- 人类思维往往是顺序的
- 时间的概念经常具有误导性
- 在计算机系统中思考所有可能的事件序列通常是不可能实现的

并发编程是极具挑战性的（2）

- 并发编程经典问题：
  - **竞争**：结果取决于系统中其他地方的任意调度决策
    - 示例：谁能得到飞机上的最后一个座位？
  - **死锁**：不正确的资源分配阻止了前进
    - 示例：交通拥堵
  - **活锁 / 饥饿 / 公平性**：外部事件和/或系统调度决策可能阻止子任务的进展
    - 示例：总有人在排队时插队
- 许多并发编程的方面超出了我们课程的范围
  - 但并非全部
  - 我们将在接下来的几节课中讨论其中的一部分问题

本章内容

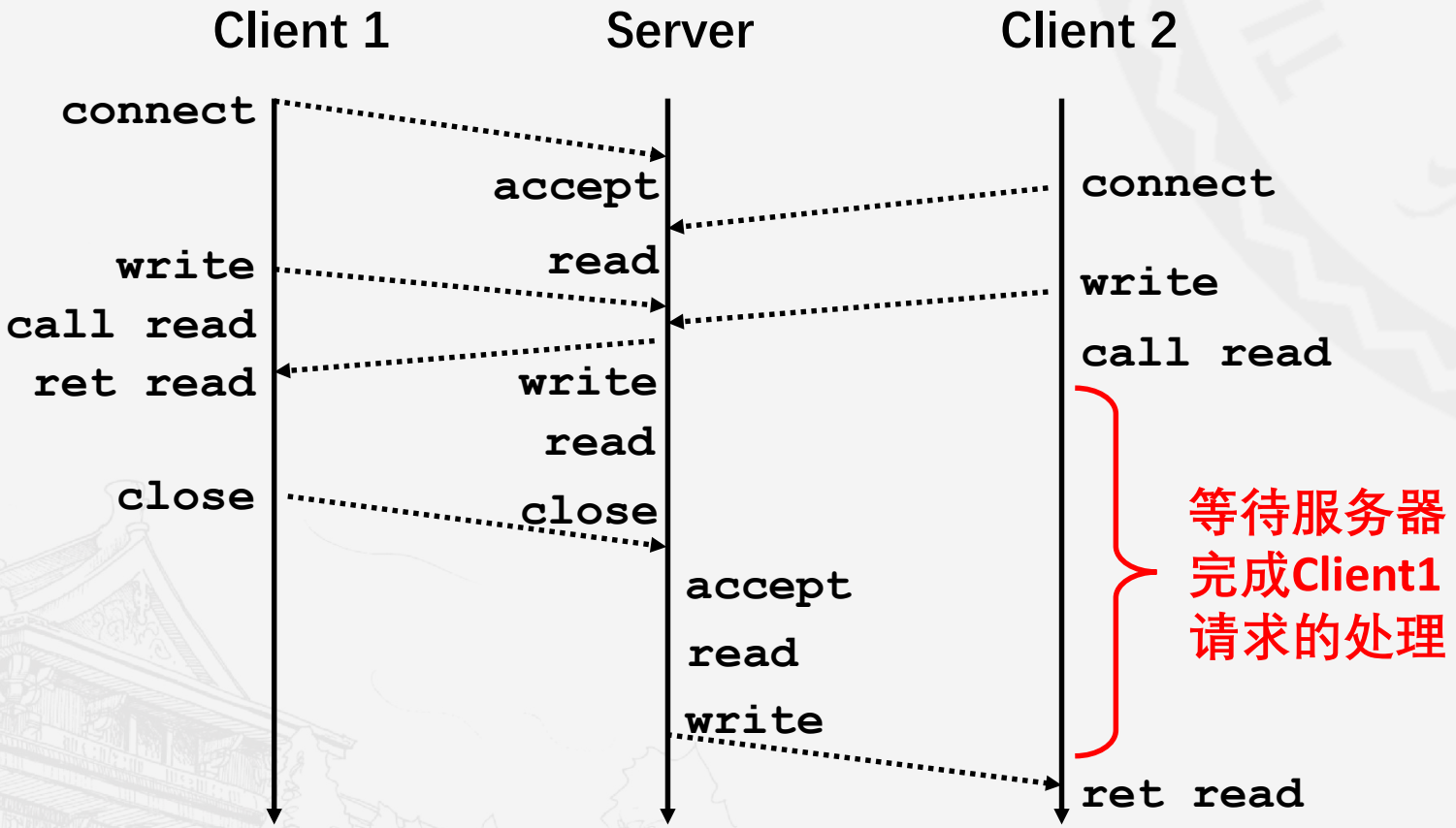
Topic

- 迭代服务
- 基于进程的服务
- 基于事件的服务
- 线程
- 基于线程的服务

迭代服务

Iterative Servers

- 特点：在同一时刻只处理一个请求

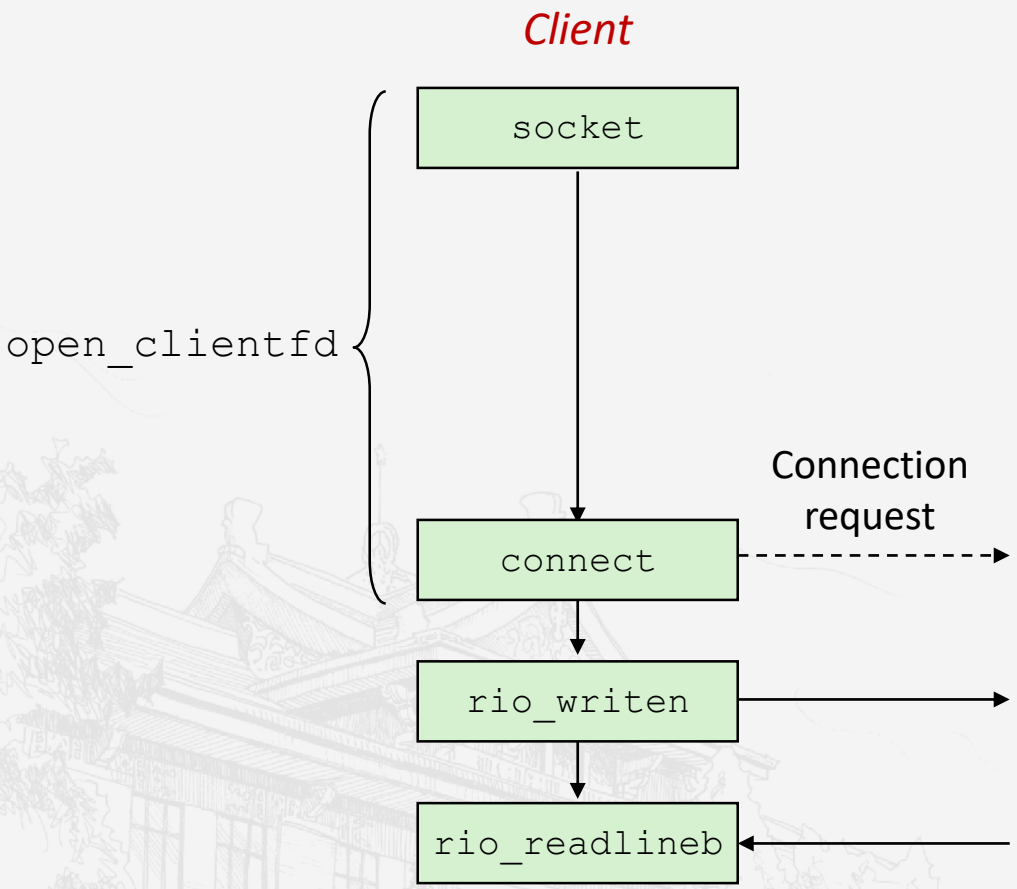


迭代服务

Iterative Servers

第二个客户端在什么地方被阻塞

第二个客户端尝试连接迭代服务器

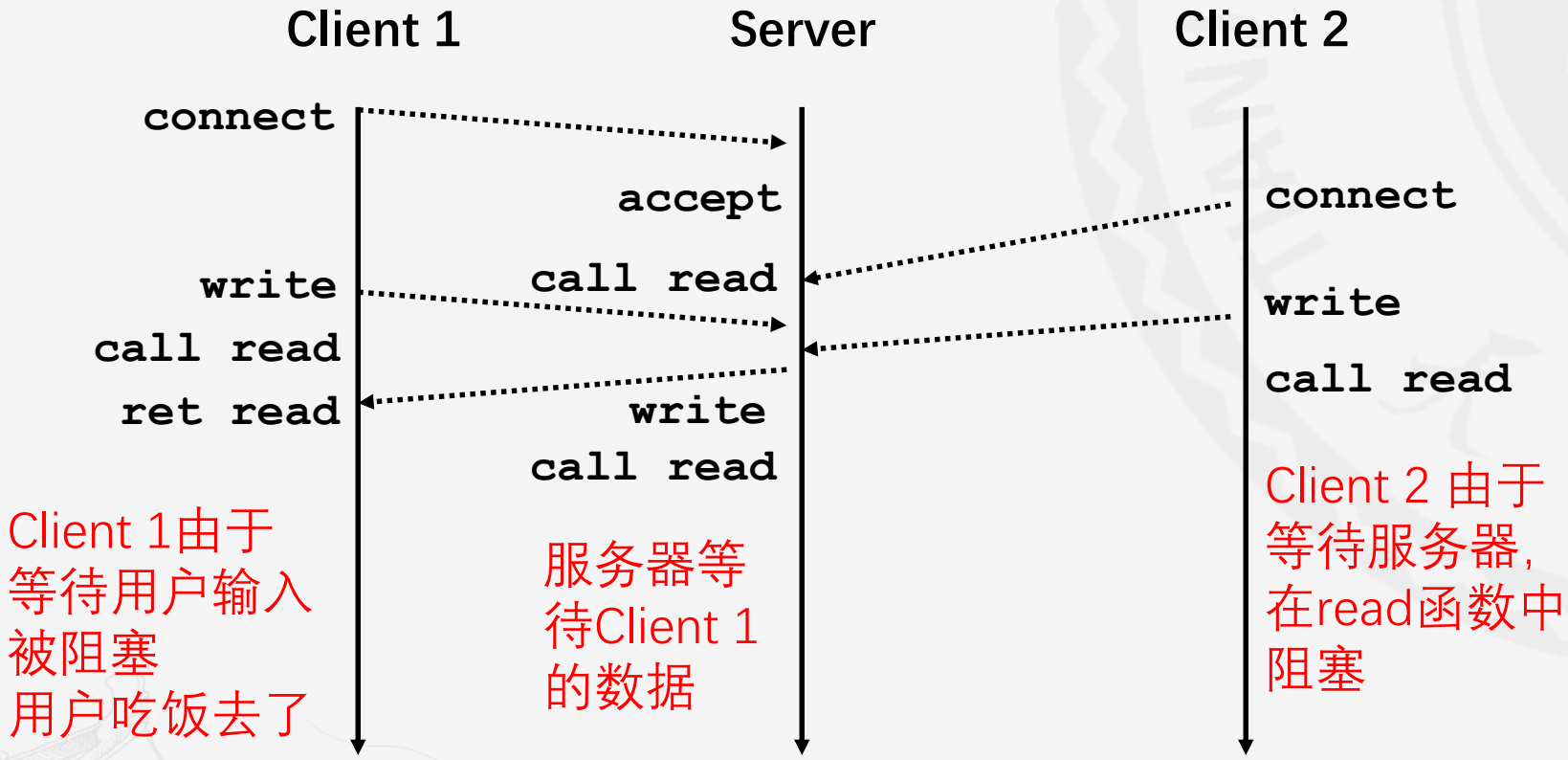


- 调用connect返回
  - 尽管连接尚未被accept
  - 服务器端TCP管理器将请求排入队列
  - 这一特性称为“TCP监听队列”
- 调用rio_writen返回
  - 服务器端TCP管理器缓冲输入数据
- 调用rio_readlineb被阻塞
  - 因为服务器尚未写入任何数据供其读取。

迭代服务

Iterative Servers

迭代式服务的缺陷



- 解决方法：使用**并发服务器**替换迭代服务器
  - 并发服务器利用多个并发流同时为多个客户端提供服务。

实现并发服务的方法

基于进程的方法

- 内核自动交错多个逻辑流
- 每个流都有自己的私有地址空间

基于事件的方法

- 程序员手动交错多个逻辑流
- 所有流共享相同的地址空间
- 使用 I/O 多路复用技术

基于线程的方法

- 内核自动交错多个逻辑流
- 每个流共享相同的地址空间
- 是基于进程和基于事件的方法的混合体

本章内容

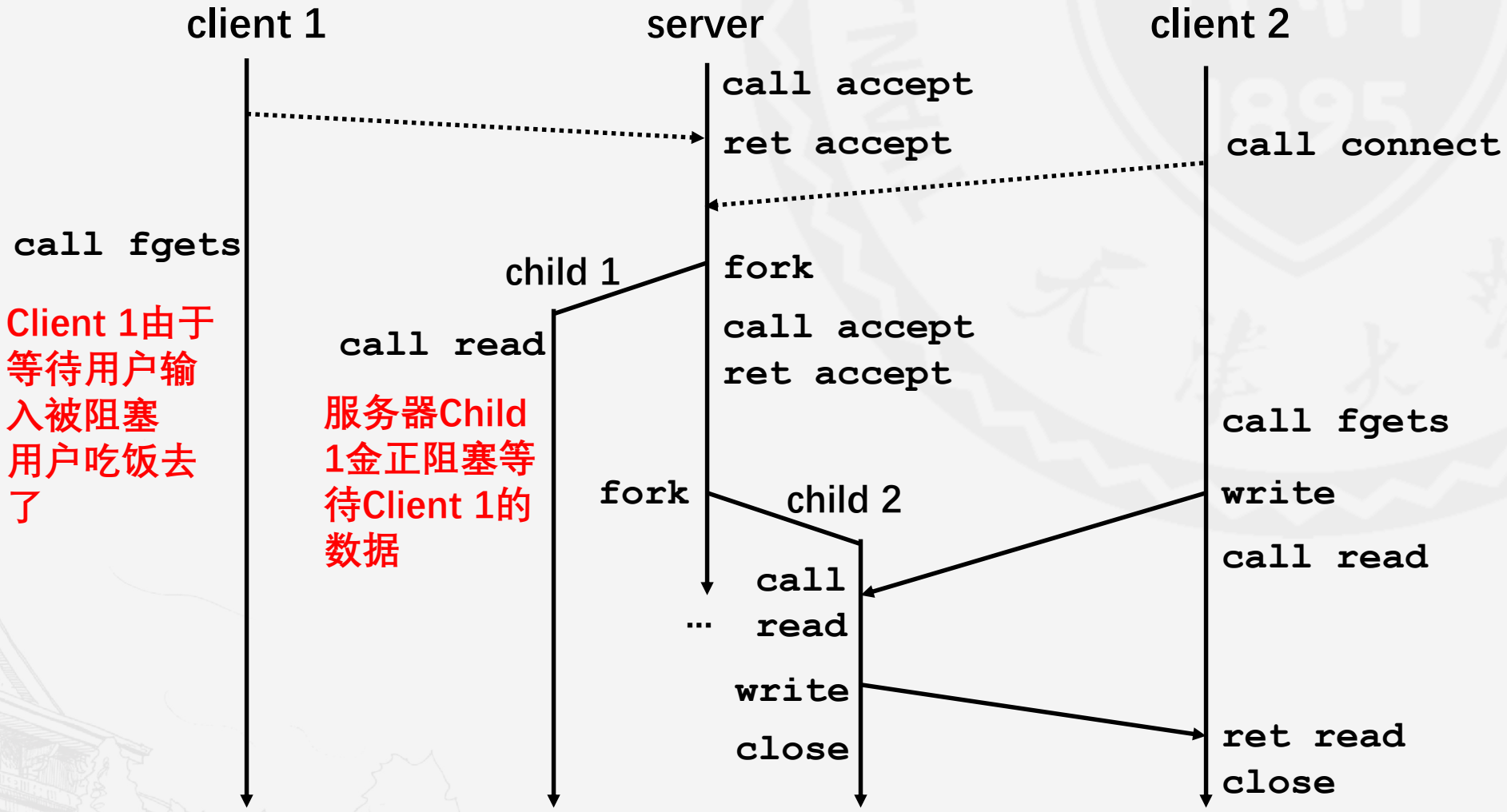
Topic

- 迭代服务
- 基于进程的服务
- 基于事件的服务
- 线程
- 基于线程的服务

基于进程的服务

Process-based Servers

- 为每个客户端创建单独的进程，处理用户请求



基于进程的服务

Process-based Servers

基于进程并发的Echo服务

```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

基于进程并发的Echo服务（2）

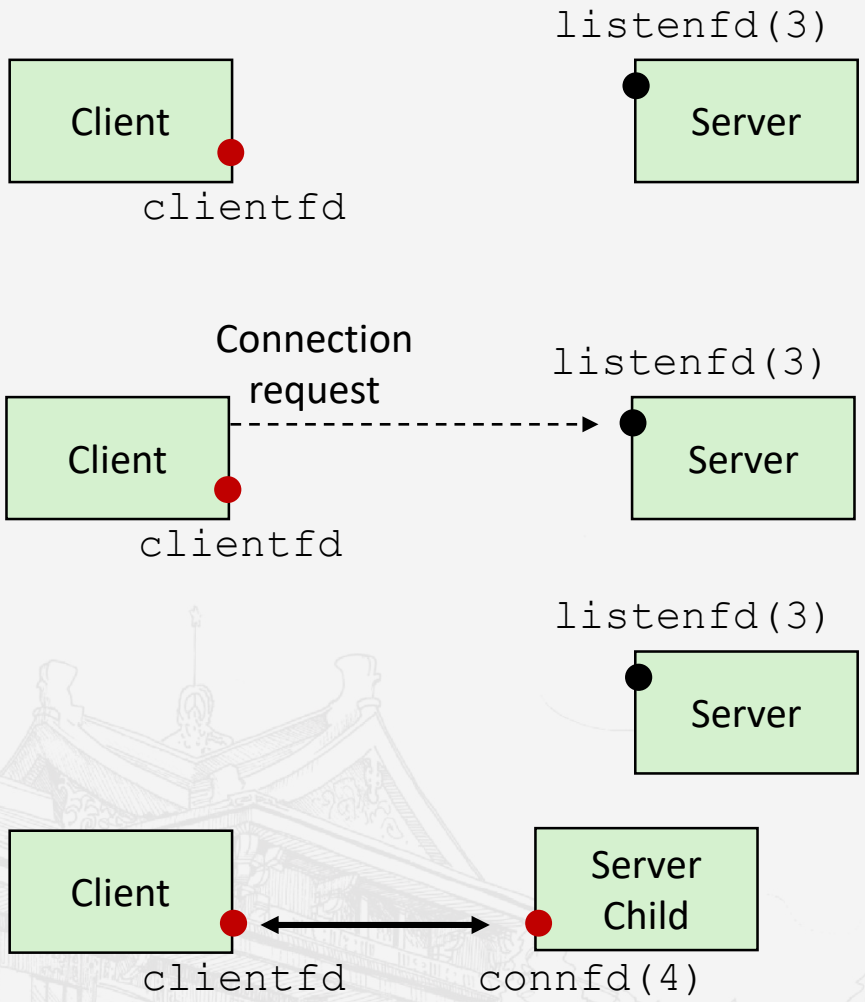
```c
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

echoserverp.c

回收僵尸进程

基于进程的服务

Process-based Servers



关于accept函数的说明

1. 服务器在accept函数中被阻塞，等待监听描述符listenfd上的连接请求。

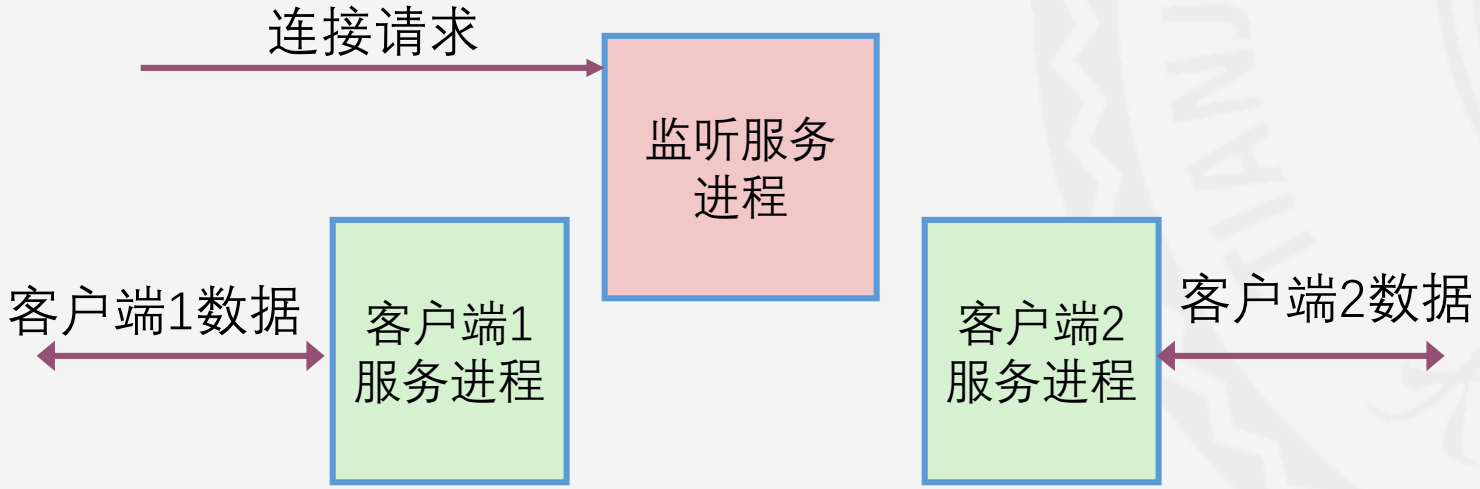
2. 客户端通过调用connect函数发起连接请求。

3. 服务器从accept函数返回connfd。然后派生子进程来处理客户端。现在clientfd和connfd之间建立了连接。

基于进程的服务

Process-based Servers

基于进程的服务执行模型



- 每个客户端由独立的子进程处理
- 它们之间没有共享状态
- 父进程和子进程都有listenfd和connfd的副本，fork执行后：
  - 父进程必须关闭connfd
  - 子进程应关闭listenfd

实现时需要注意的问题

- 监听服务进程必须清理僵尸子进程
  - 以避免**致命的**内存泄漏。
- 父进程必须关闭其connfd的副本
  - 内核为每个套接字/打开文件维护引用计数
  - 在fork之后，connfd的引用计数 refcnt(connfd) = 2
  - 直到refcnt(connfd) = 0时，连接才会真正关闭

基于进程的服务

Process-based Servers

优缺点

- 优点：
  - 能够同时处理多个连接
  - 清晰的共享模型：共享文件表（由内核维护数据结构），没有全局变量
  - 简单直接
- 缺点：
  - 进程控制引入了额外的开销
  - 在进程间共享数据实现复杂：需要使用进程间通信（IPC）机制实现
    - 如：命名管道、System V共享内存和信号量（semaphores）

本章内容

Topic

- 迭代服务
- 基于进程的服务
- 基于事件的服务
- 线程
- 基于线程的服务

基于事件的服务

Event-based Servers

- 服务器维护一组活动连接：即connfd的数组
- 重复以下步骤：
  - 确定哪些描述符（connfd或listenfd）有待处理的输入
    - 例如：使用select或epoll函数
    - 待处理输入的到达是一个事件
  - 如果listenfd有输入，则accept连接并将新的connfd添加到数组中
  - 为所有具有待处理输入的connfd提供服务
- 教材中详细介绍了基于select的服务器，而epoll可能会成为未来的趋势

基于事件的服务

Event-based Servers

```
typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* Highwater index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;
```

基于事件的服务

Event-based Servers

```
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
       p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

基于事件的服务

Event-based Servers

```
int byte_cnt = 0; /* Counts total bytes received by server */

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool); //line:conc:echoservers:initpool
```

基于事件的服务

Event-based Servers

```
    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready */
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);

        /* If listening descriptor ready, add new client to pool */
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

```
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool highwater mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    if (i == FD_SETSIZE) /* Couldn't find an empty slot */
        app_error("add_client error: Too many clients");
}
```

```
void check_clients(pool *p)
{
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;
    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];
        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                byte_cnt += n;
                printf("Server received %d (%d total) bytes on fd %d\n",
                       n, byte_cnt, connfd);
                Rio_writen(connfd, buf, n);
            }
            /* EOF detected, remove descriptor from pool */
            else {
                Close(connfd); //line:conc:echoservers:closeconnfd
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

基于事件的服务

Event-based Servers

优缺点

- 优点：
  - 单一逻辑控制流和地址空间。
  - 可以使用调试器逐步执行。
  - 没有进程或线程控制开销。
  - 高性能Web服务器和搜索引擎的首选设计。例如Node.js、nginx、Tornado
- 缺点
  - 编码比基于进程或线程的设计复杂得多。
  - 很难提供细粒度的并发性。例如：如何处理部分HTTP请求头
  - 无法利用多核：单一的控制流

本章内容

Topic

- 迭代服务
- 基于进程的服务
- 基于事件的服务
- 线程
- 基于线程的服务

传统的进程视角

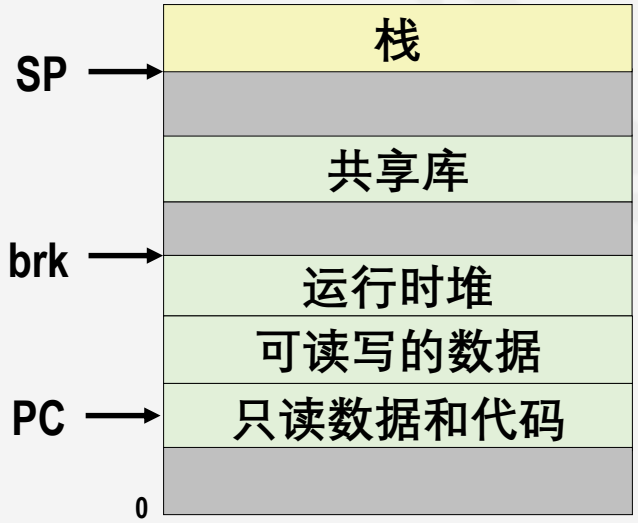
- 进程 = 进程上下文 + 代码、数据和栈

进程上下文

程序上下文：
- 通用寄存器
- 条件码
- 栈顶指针 (SP)
- 程序计数器 (PC)

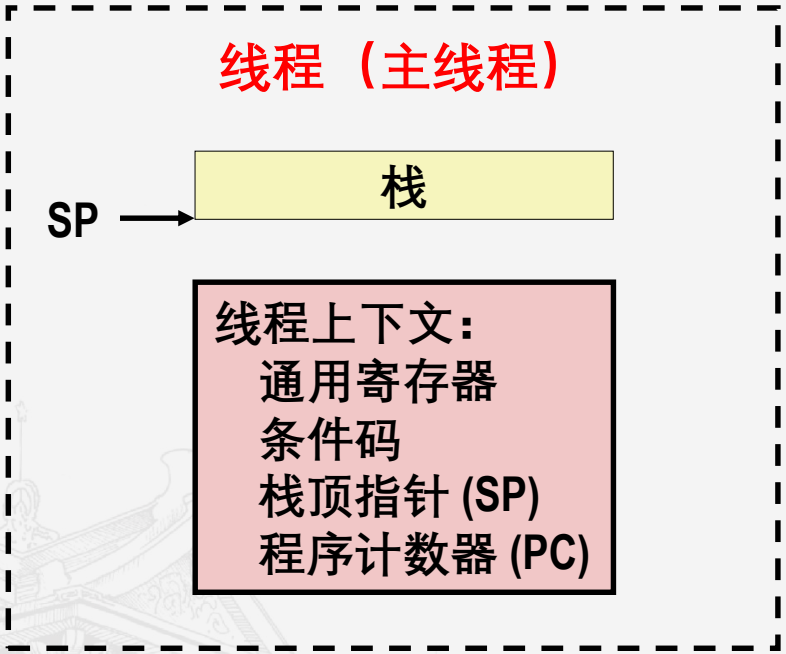
内核上下文:
- 虚拟内存结构体
- 描述符表
- brk指针 （堆）

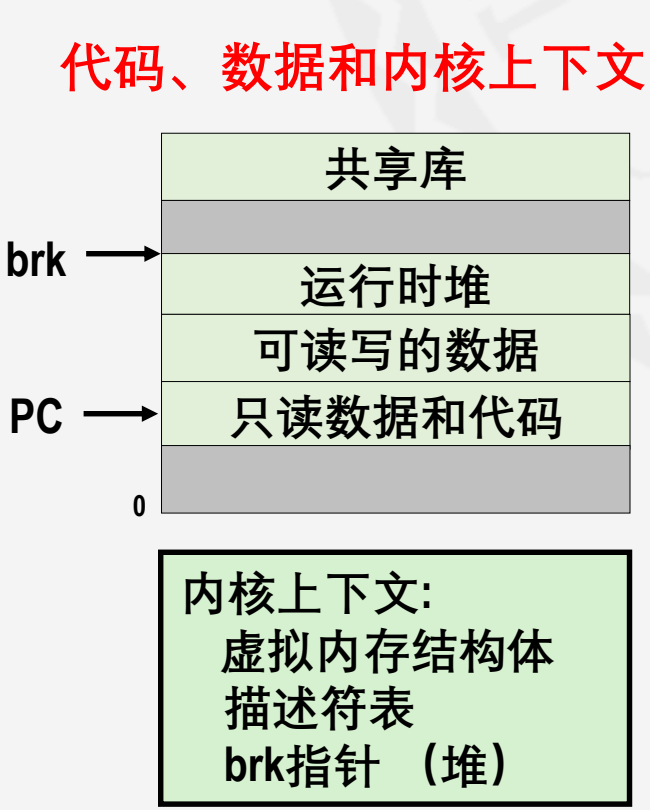
代码、数据和栈



另一个进程的视角

进程 = 线程 + 代码、数据和内核上下文



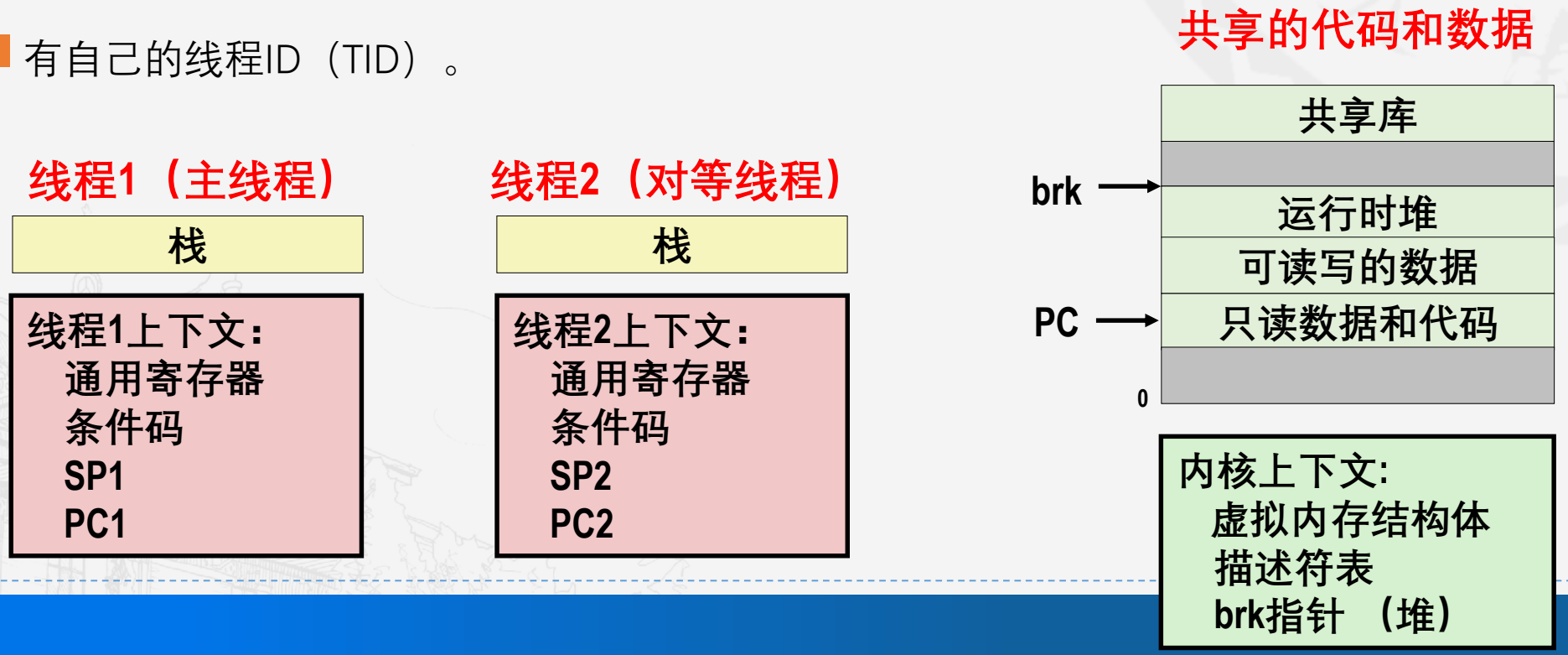


线程

Threads

包含多个线程的一个进程

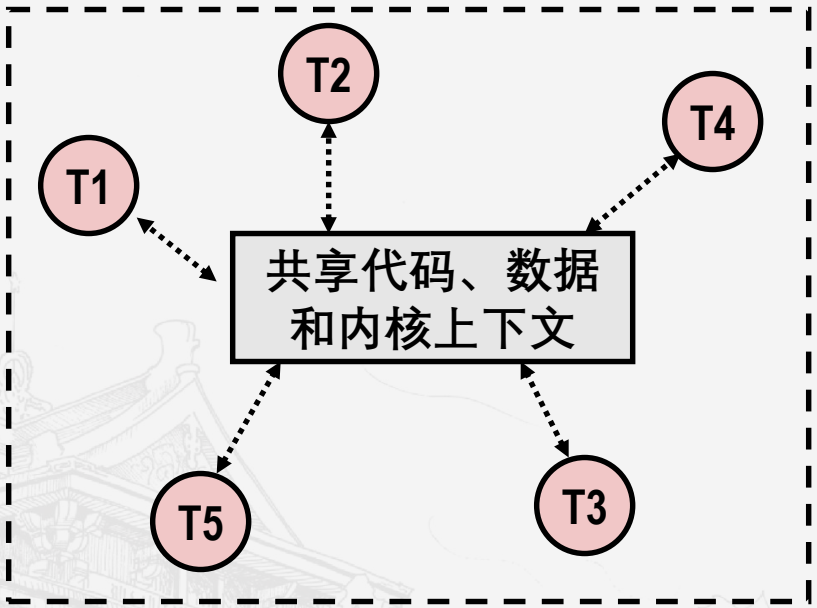
- 一个进程可以关联多个线程。每个线程：
  - 有自己的逻辑控制流。
  - 共享相同的代码、数据和内核上下文。
  - 有自己的栈用于本地变量，**但不受其他线程保护**。
  - 有自己的线程ID（TID）。



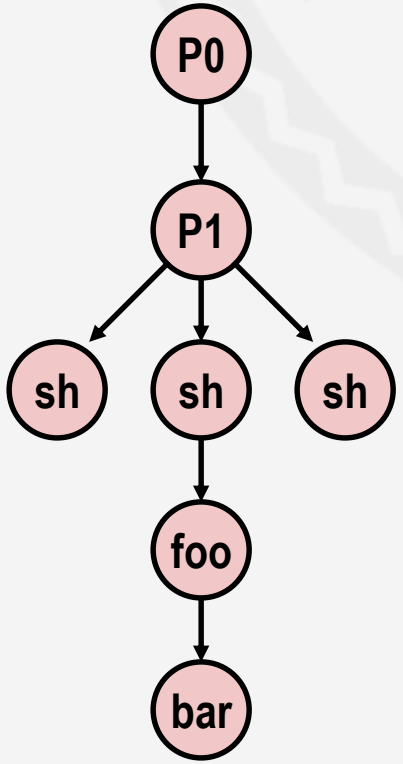
线程的逻辑视角

- 与进程关联的线程形成一个对等池
  - 与进程形成树状层次结构不同

与进程相关的线程

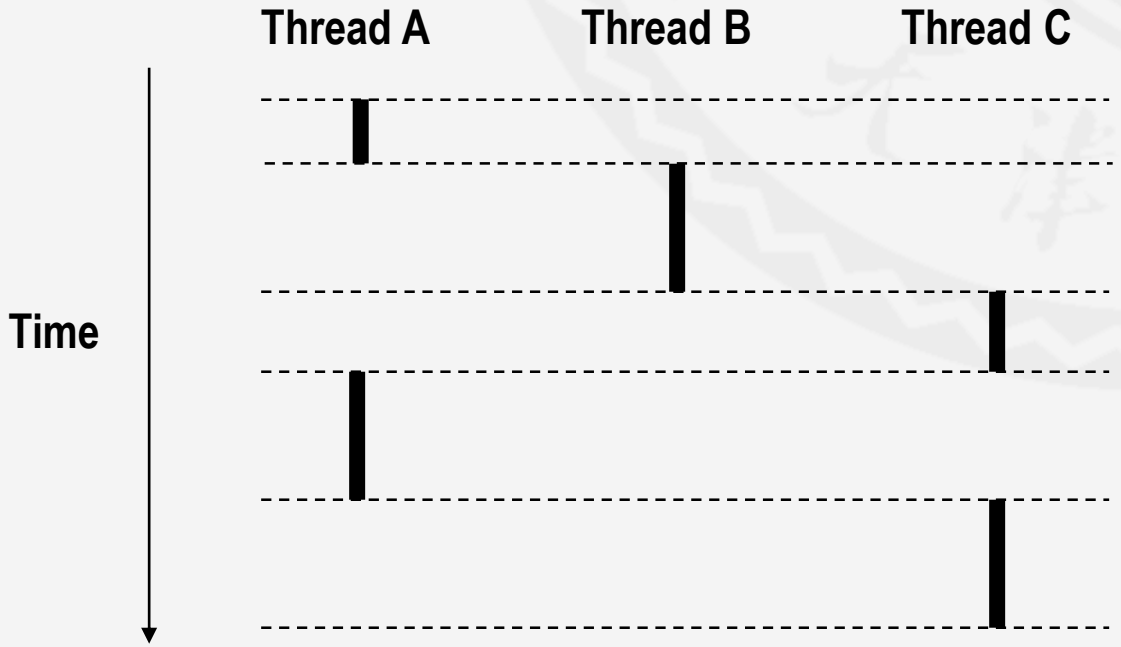


进程的层次结构



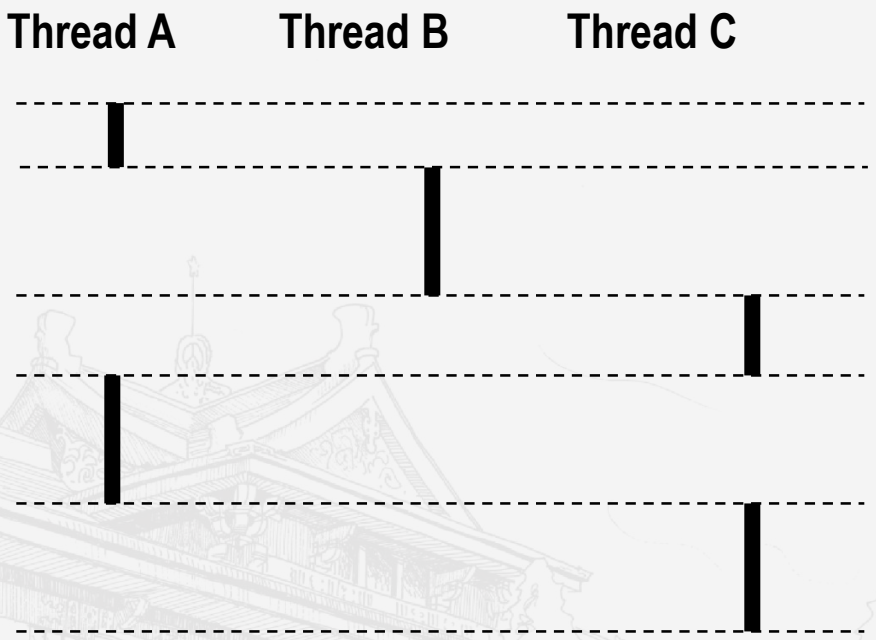
并发线程

- 如果两个线程的执行时间重叠，则它们是并发的
- 否则，它们是顺序的
- 例子：
  - 并发的：A和B，A和C
  - 顺序的：B和C

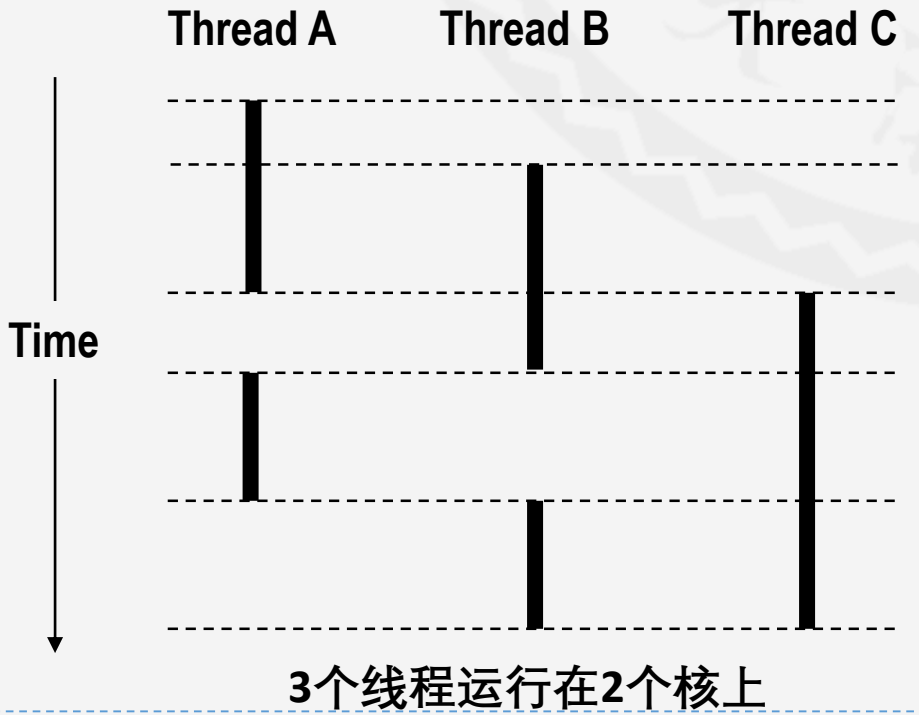


并发线程的执行

- 单核处理器
  - 通过时间片划分来模拟并行性



- 多核处理器
  - 真正的并行



3个线程运行在2个核上

线程与进程

- 线程和进程的相似之处:
  - 都有自己的逻辑控制流。
  - 都可以与其他线程/进程（可能在不同的核心上）并发运行。
  - 都会进行上下文切换。
- 线程和进程的不同之处:
  - 线程共享所有代码和数据（除了本地栈）；而进程（通常）不共享。
  - 线程的性能开销比进程略低
    - 创建和回收进程的控制开销是线程控制的两倍。
    - 在Linux中的估算数据:
      - 创建和回收一个进程大约需要20000个周期
      - 创建和回收一个线程只需要10000个周期（甚至更少）。

Posix线程（Pthread）接口

- Pthreads：基于C语言的线程控制函数集（约60个函数）
  - 创建和回收线程：pthread_create()、pthread_join()
  - 获取当前线程ID：pthread_self()
  - 终止线程：pthread_cancel()、pthread_exit()
    - exit()会终止所有线程，当前进程的主函数return，会终止当前线程
  - 线程同步：pthread_mutex_init、pthread_mutex_lock 、pthread_mutex_unlock

Pthread版本的Helloworld

```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

线程ID

线程属性
通常为NULL

线程的主函数

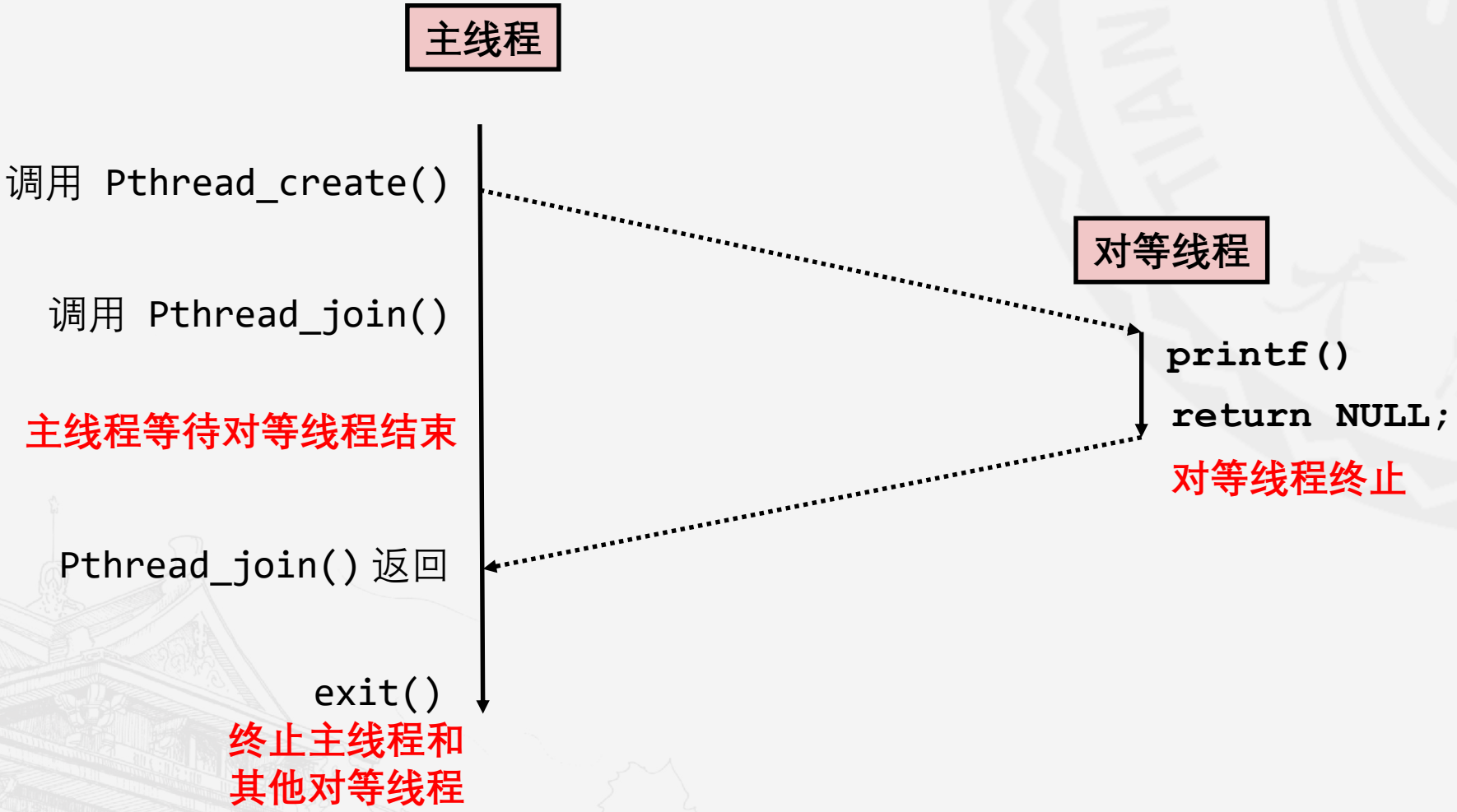
传入主函数的参数
*(void *p)*

线程主函数的返回值
*(void **p)*

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

hello.c

Pthread版本的Helloworld的执行



本章内容

Topic

- 迭代服务
- 基于进程的服务
- 基于事件的服务
- 线程
- 基于线程的服务

基于线程的服务

Thread-based Servers

基于线程的Echo服务

- 与基于进程的服务非常相似，只是使用线程替代了进程

```
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

为了避免后续发生致命竞争，必须对连接描述符使用malloc分配空间

基于线程的Echo服务（2）

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

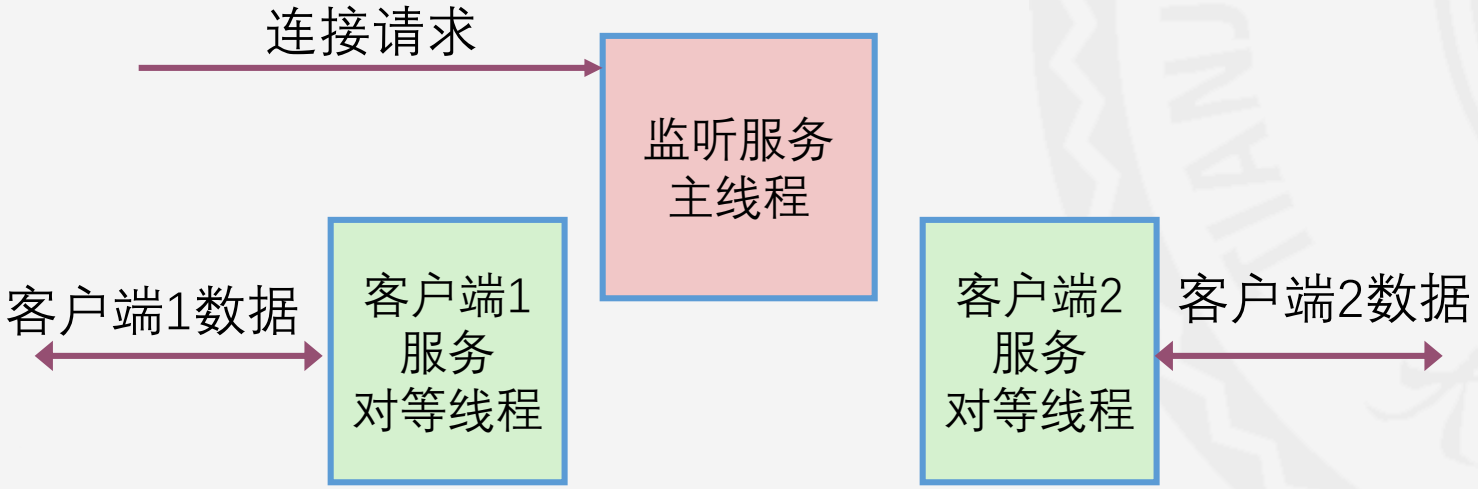
echoservert.c

- 以“分离（detached）”模式运行线程
  - 独立于其他线程运行
  - 当线程终止时，由内核自动回收
- 释放分配给保存connfd的存储空间
- 关闭connfd（重要！）

基于线程的服务

Thread-based Servers

基于线程服务的执行模型



- 每个客户端由独立的对等线程处理
- 线程共享除TID之外的所有进程状态
- 每个线程都有一个用于存储本地变量的独立的栈

基于线程的服务

Thread-based Servers

实现时需要注意的问题

- 必须以“分离”的方式运行以避免内存泄漏。
  - 在任何时刻，线程要么是可连接的（joinable），要么是分离的（detached）
  - 可连接的线程可以被其他线程回收和终止：必须被回收（使用pthread_join）以释放内存资源
  - 分离线程不能被其他线程回收或终止：资源在终止时会自动被回收
  - 默认状态是可连接的：使用pthread_detach(pthread_self())来使线程处于分离状态
- 必须小心避免意外的共享，例如：传递指向主线程栈的指针。
  - pthread_create(&tid, NULL, thread, (void *)&connfd);
- 由线程调用的所有函数必须是线程安全的（thread-safe）

基于线程的服务

Thread-based Servers

优缺点

- 优点：
  - 线程间共享数据结构十分容易，例如：日志信息、文件缓存
  - 比进程效率更高
- 缺点：
  - 意外共享可能导致难以复现的微妙错误！
    - 数据共享的便利性既是线程的最大优点，也是最大弱点
    - 很难知道哪些数据是共享的，哪些是私有的
    - 很难通过测试来检测（问题不易复现）
      - 出现竞争可能性非常低，但不为零（后续讨论）

总结

- 基于进程：
  - 资源共享困难：容易避免意外共享
  - 添加/删除客户端的开销高
- 基于事件：
  - 繁琐且底层
  - 对调度有完全控制
  - 开销非常低
  - 不能创建同等粒度的并发
  - 没有利用多核
- 基于线程：
  - 资源共享容易：也许太容易了
  - 中等开销
  - 对调度策略控制不大
  - 调试困难（事件顺序不可重复）