Concurrent Programming

廖丁鸥

引言

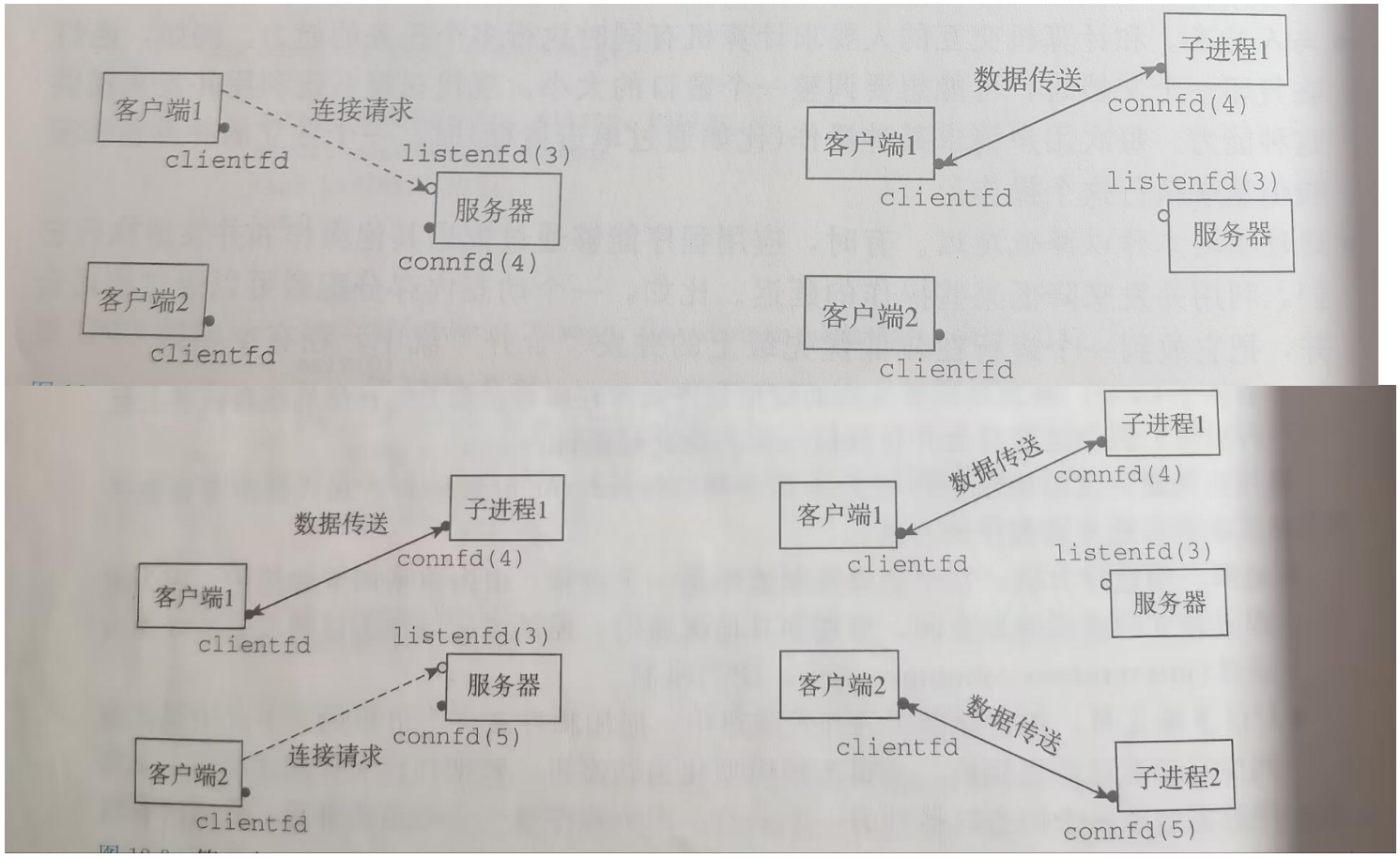
- 并发的作用：并发不仅是一种操作系统内核用来运行多个应用程序的机制，也可以在应用程序中发挥作用。应用级并发在访问慢速I/O设备、与人交互、通过推迟工作以降低延迟、服务多个网络客户端、在多核机器上进行并行计算等情况下十分有用。
- 构造并发程序的方法：
- 1.进程；
- 2.I/O多路复用；
- 3.线程。

```c
#include "csapp.h"

int main(int argc, char **argv)
{
    int clientfd;
    char *host, *port, buf[MAXLINE];
    rio_t rio;

    if (argc != 3) {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
        exit(0);
    }
    host = argv[1];
    port = argv[2];

    clientfd = Open_clientfd(host, port);
    Rio_readinitb(&rio, clientfd);

    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf));
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
    Close(clientfd);
    exit(0);
}
```

code/netp/echoclient.c

基于进程的并发编程

- 进程是构造并发程序最简单的方法，使用fork、exec和waitpid这些熟悉的函数。



code/conc/echoserverp

```
1  #include "csapp.h"
2  void echo(int connfd);
3
4  void sigchld_handler(int sig)
5  {
6      while (waitpid(-1, 0, WNOHANG) > 0)
7          ;
8      return;
9  }
10
11 int main(int argc, char **argv)
12 {
13     int listenfd, connfd;
14     socklen_t clientlen;
15     struct sockaddr_storage clientaddr;
16
17     if (argc != 2) {
18         fprintf(stderr, "usage: %s <port>\n", argv[0]);
19         exit(0);
20     }
21
22     Signal(SIGCHLD, sigchld_handler);
23     listenfd = Open_listenfd(argv[1]);
24     while (1) {
25         clientlen = sizeof(struct sockaddr_storage);
26         connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
27         if (Fork() == 0) {
28             Close(listenfd); /* Child closes its listening socket */
29             echo(connfd);    /* Child services client */
30             Close(connfd);   /* Child closes connection with client */
31             exit(0);         /* Child exits */
32         }
33         Close(connfd); /* Parent closes connected socket (important!) */
34     }
35 }
```

code/conc/echoserverp

基于进程的并发编程

- 需要注意的一些问题：
- 1.由于通常服务器会运行很长时间，所以要包括一个SIGCHLD处理程序来回收僵死子进程的资源，以避免致命的内存泄漏；
- 2.子进程和父进程必须关闭各自的connfd副本。对父进程尤为重要，否则将不会释放已连接描述符的文件表条目，引起的内存泄漏会消耗光可用的内存使系统崩溃；
- 3.因为套接字的文件表表项中的引用计数，直到父子进程的connfd都关闭了，到客户端的连接才会终止。

进程的优劣

- 优：
- 1.一个进程不可能不小心覆盖另一个进程的虚拟内存，多个进程可以在同一时间段同时执行，提高了程序的并行能力，更适合用于处理并行任务；
- 2.拥有清洁的共享模式；
- 3.简单而直接。
- 劣：
- 1.独立的地址空间使得进程共享状态信息变得更加困难，必须使用显式的IPC机制。
- 2.进程控制和IPC的开销很高导致基于进程的设计往往比较慢。

基于I/O多路复用的并发编程

- 基本思路：使用select函数，要求内核挂起进程，只有在一个或者多个I/O事件发生后，才将控制返回给应用程序
- **基于I/O多路复用的并发事件驱动服务器**

```
1   #include "csapp.h"
2
3   typedef struct { /* Represents a pool of connected descriptors */
4       int maxfd;        /* Largest descriptor in read_set */
5       fd_set read_set;  /* Set of all active descriptors */
6       fd_set ready_set; /* Subset of descriptors ready for reading  */
7       int nready;       /* Number of ready descriptors from select */
8       int maxi;         /* High water index into client array */
9       int clientfd[FD_SETSIZE];    /* Set of active descriptors */
10      rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
11  } pool;
12
13  int byte_cnt = 0; /* Counts total bytes received by server */
14
15  int main(int argc, char **argv)
16  {
17      int listenfd, connfd;
18      socklen_t clientlen;
19      struct sockaddr_storage clientaddr;
```

```
20      static pool pool;
21
22      if (argc != 2) {
23          fprintf(stderr, "usage: %s <port>\n", argv[0]);
24          exit(0);
25      }
26      listenfd = Open_listenfd(argv[1]);
27      init_pool(listenfd, &pool);
28
29      while (1) {
30          /* Wait for listening/connected descriptor(s) to become ready */
31          pool.ready_set = pool.read_set;
32          pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
33
34          /* If listening descriptor ready, add new client to pool */
35          if (FD_ISSET(listenfd, &pool.ready_set)) {
36              clientlen = sizeof(struct sockaddr_storage);
37              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
38              add_client(connfd, &pool);
39          }
40
41          /* Echo a text line from each ready connected descriptor */
42          check_clients(&pool);
43      }
44  }
```

基于I/O多路复用的并发编程

- 如果标准输入准备好了，就调用command函数，在返回主程序前，会读、解析和响应命令
- 如果监听描述符准备好了，就调用accept得到一个已连接描述符，再调用echo函数回送来自客户端的每一行，直到客户端关闭
- 假设远程客户端发送连接请求。select()返回进入客户端处理逻辑。由于这段逻辑是同步运行的。所以在select()函数返回到处理完客户端请求的这段时间内，我们没有再次调用select()，因此这段时间内我们是无法响应的。

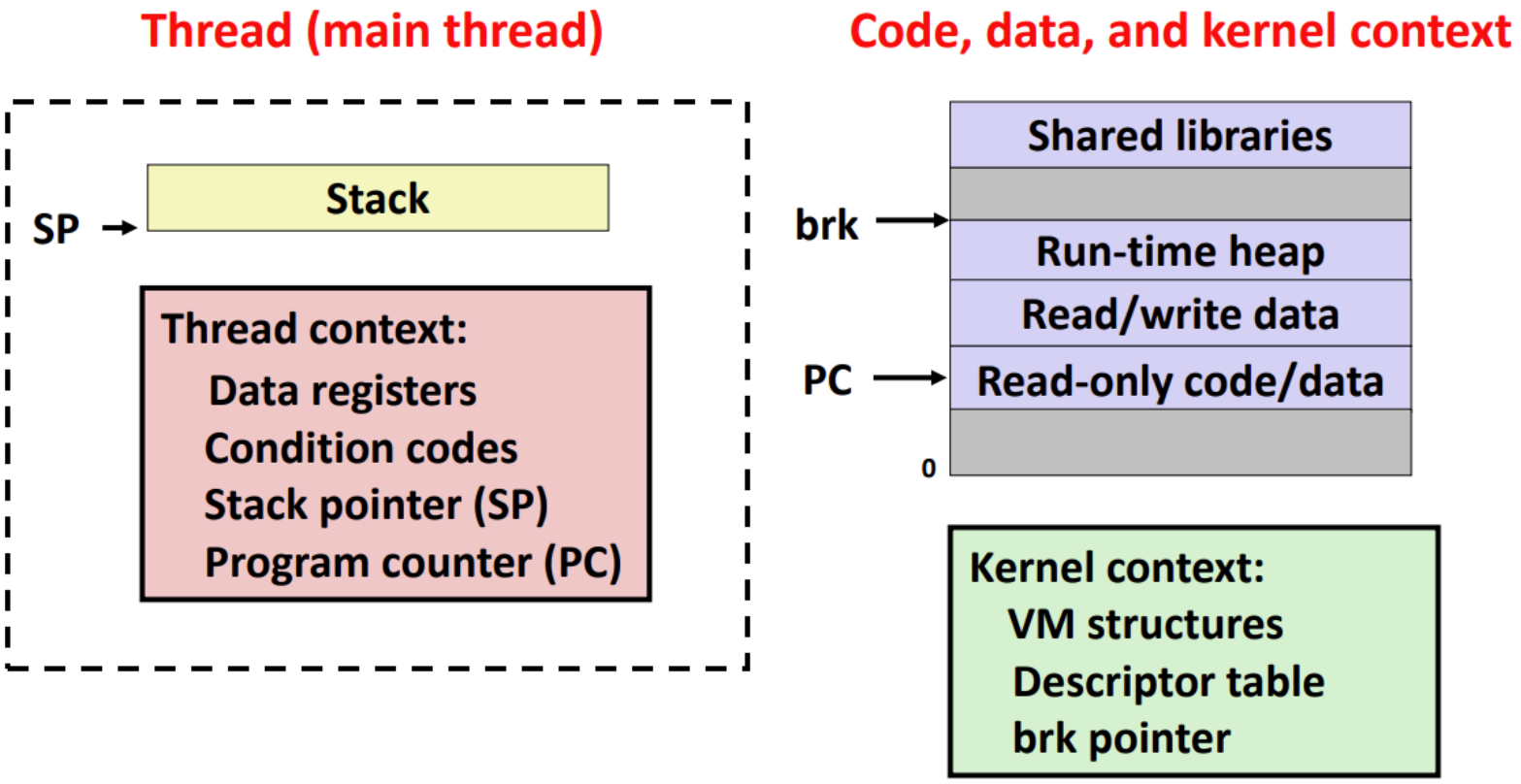
I/O多路复用技术的优劣

- 优：
- 1.比基于进程的设计给了程序员更多的对程序行为的控制；
- 2.每个逻辑流都能访问该进程的全部地址空间，使得在流之间共享数据变得很容易；
- 3.可以使用调试器单步执行；
- 4.没有进程或线程控制开销，更加高效。
- 劣：
- 1.编码复杂；
- 2.难以提供细粒度并发；
- 3.不能充分利用多核处理器。

基于线程的并发编程

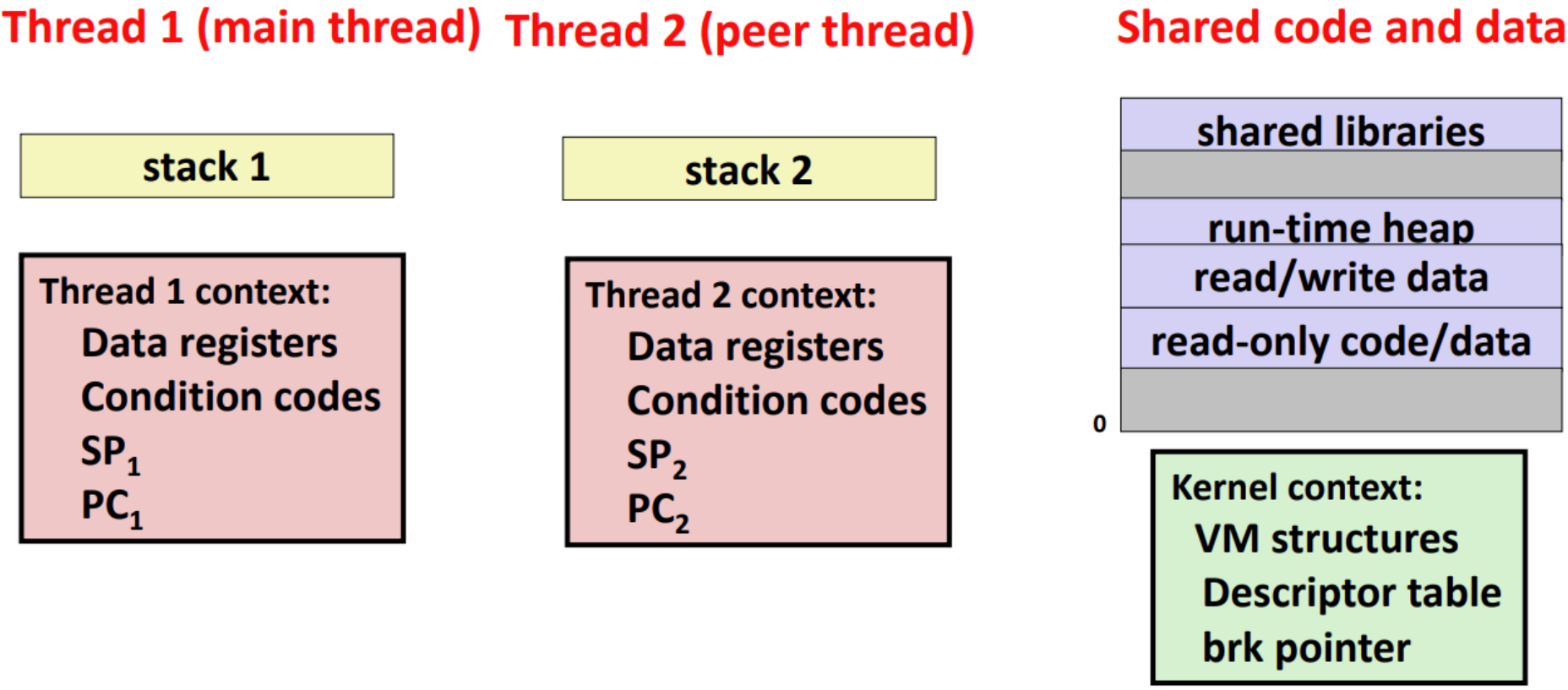
- 结合了基于进程和基于I/O多路复用的流的特性。
- 与进程相似的地方：
- 1.由内核自动调度，并且内核通过一个整数ID来识别线程，每个内核可与其他内核同时运行；
- 2.每个线程都有自己的逻辑控制流；
- 3.每个线程都是上下文切换的；
- 与I/O多路复用的流相同的地方：多个线程运行在单一进程的上下文中，共享这个进程虚拟地址空间的所有内容。

Process = thread + code, data, and kernel context



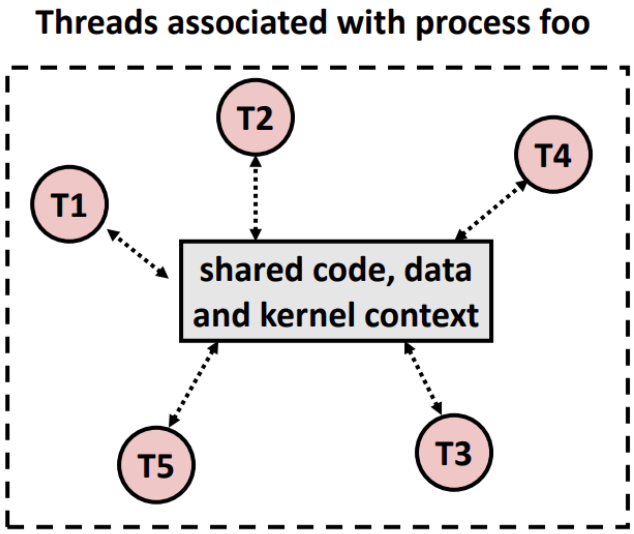
基于线程的并发编程

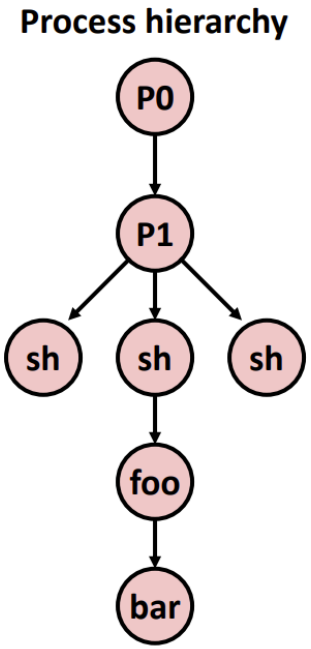
- 进程开始生命周期时为单一线程，称为主线程，某一时刻主线程创建一个对等线程，此时两个线程并发地运行。
- 多个线程可以与一个进程相关联
- 每个线程共享相同的代码、数据和内核上下文
- 每个线程有自己的局部变量堆栈，但不受其他线程保护
- 每个线程有自己的线程ID


Thread 1 (main thread)
Thread 2 (peer thread)
Shared code and data
stack 1
stack 2
shared libraries
run-time heap
read/write data
read-only code/data
0
Thread 1 context:
Data registers
Condition codes
SP1
PC1
Thread 2 context:
Data registers
Condition codes
SP2
PC2
Kernel context:
VM structures
Descriptor table
brk pointer

基于线程的并发编程

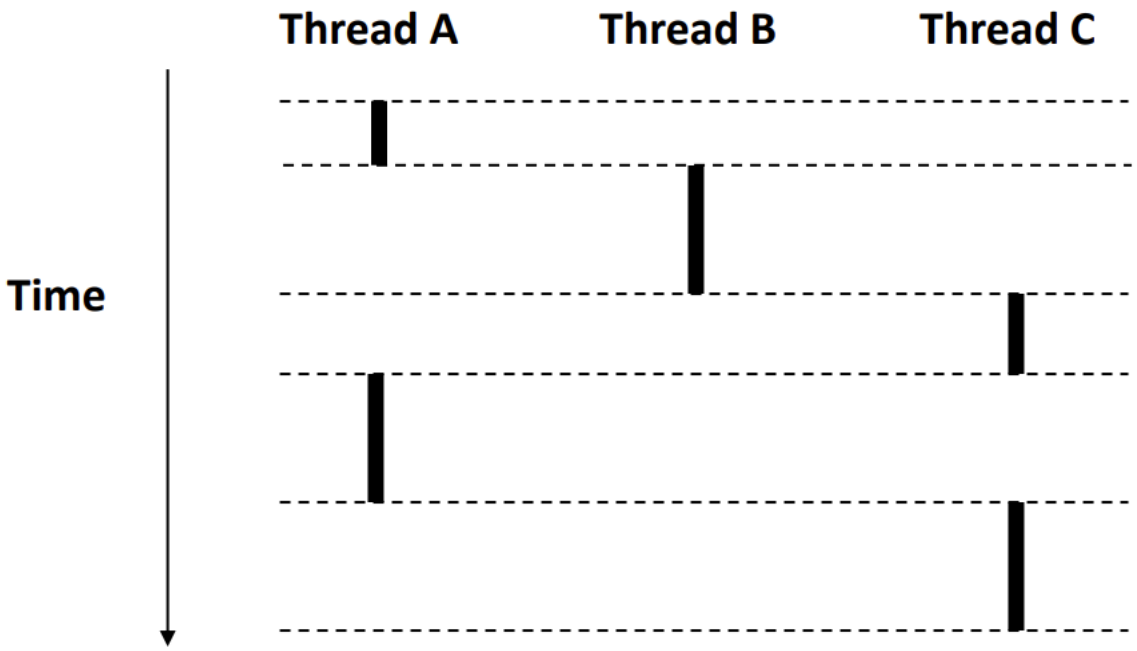
- 与进程不同的地方：不是按照严格的父子层次组织，与进程相关的线程形成对等池，独立于其他线程创建的线程。
- 主线程总是进程中第一个运行的线程。





基于线程的并发编程

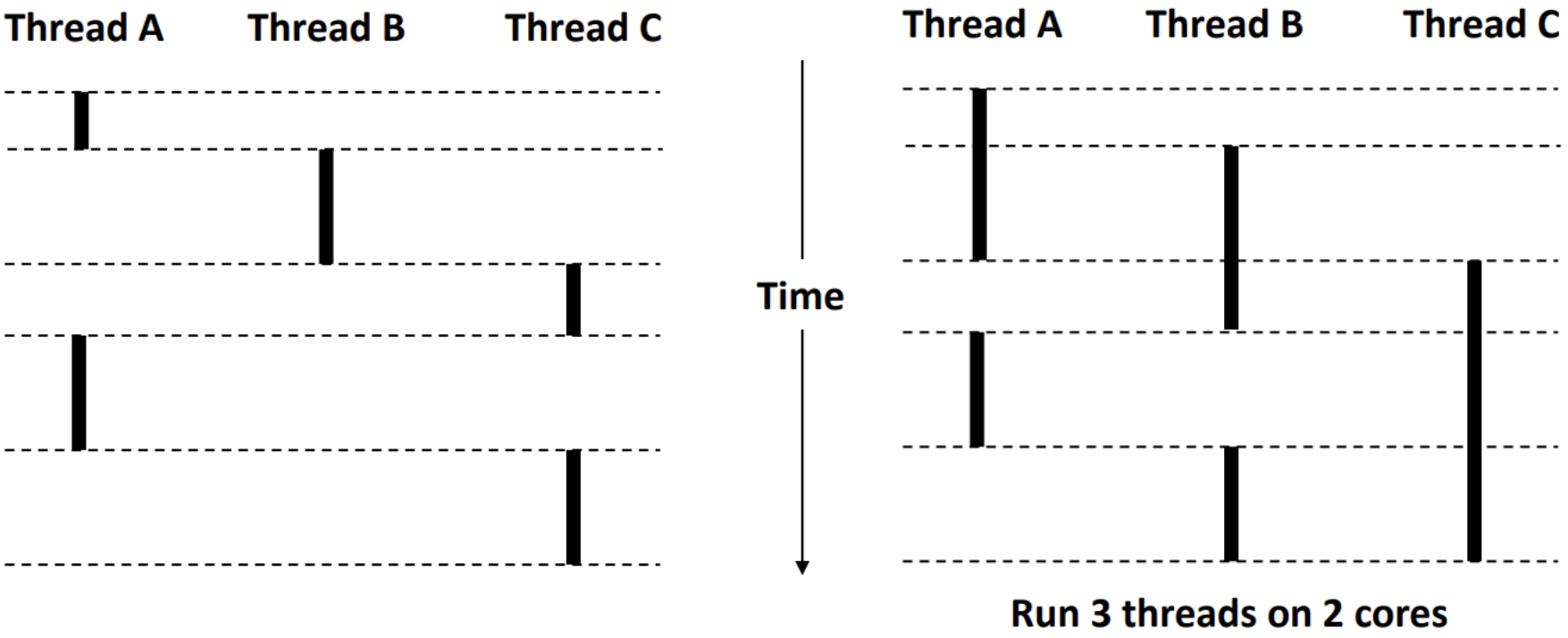
- 并发线程：若两个线程的流在时间上重叠，则这两个线程是并发的，否则就是连续的。
- 例：



并发线程的执行

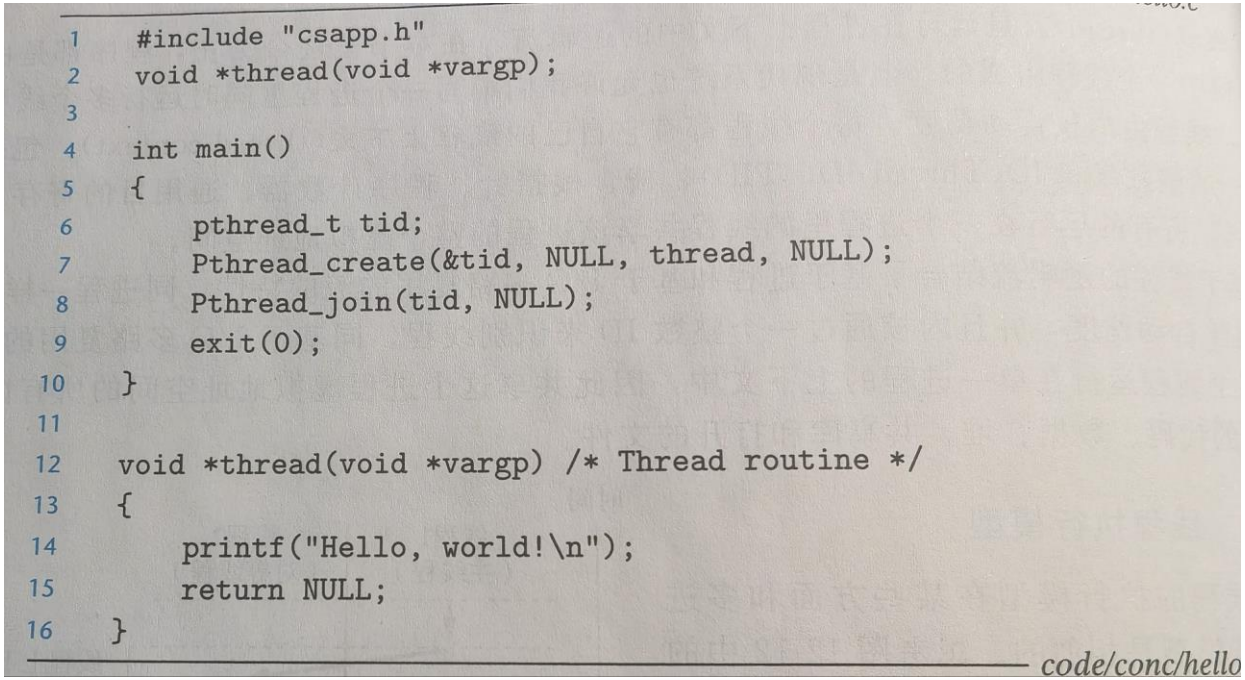
- 单核处理器
  时间分片模拟并行性

多核处理器
可具有真正的并行性



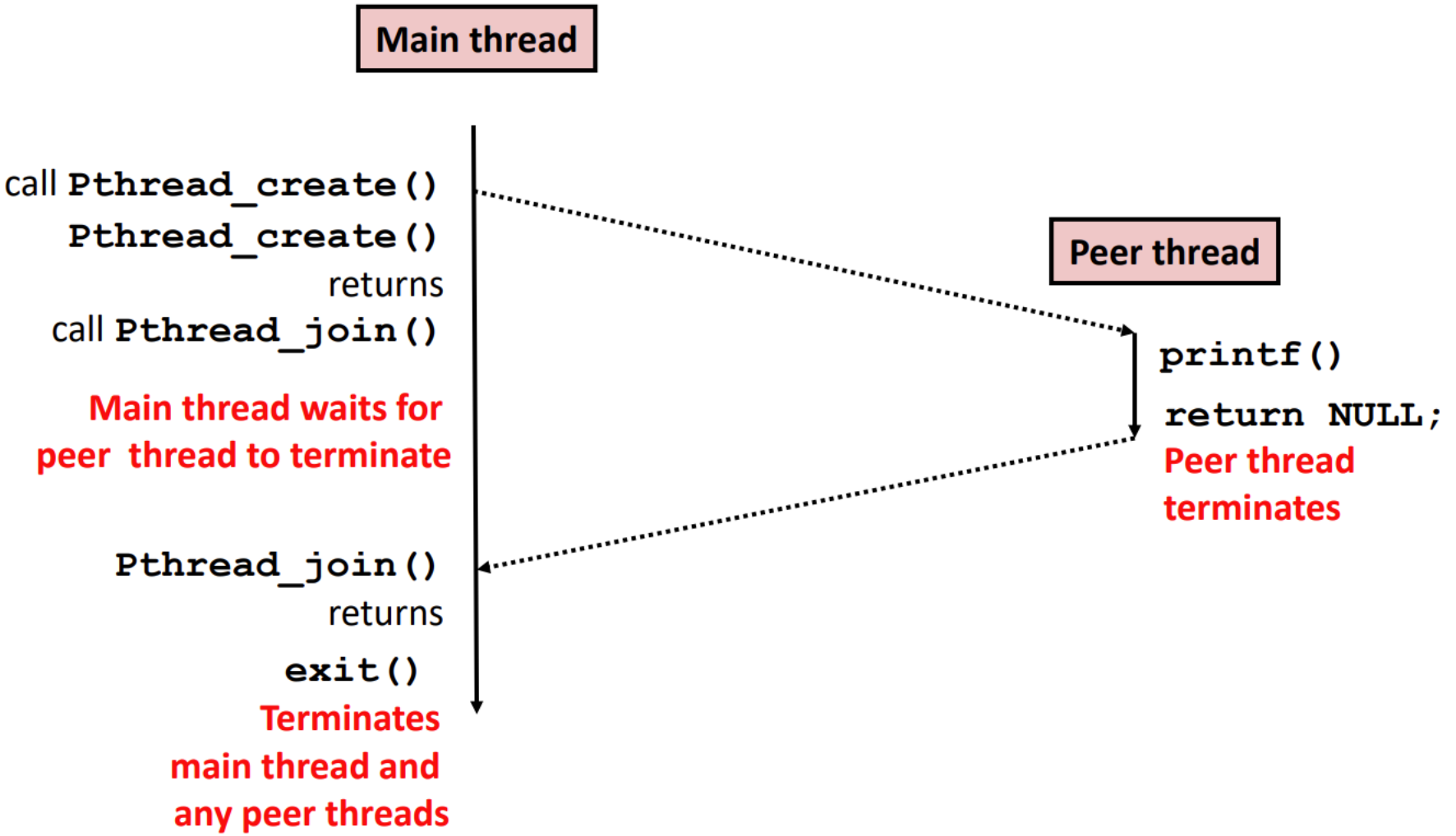
Posix线程

- Posix线程是在C程序中处理线程的一个标准切口，在所有Linux系统中可用，定义了60个左右的函数，允许程序创建（pthread_create()）、杀死和回收线程，与对等线程安全地共享数据，也可以通知对等线程系统状态的变化。



```
1   #include "csapp.h"
2   void *thread(void *vargp);
3
4   int main()
5   {
6       pthread_t tid;
7       Pthread_create(&tid, NULL, thread, NULL);
8       Pthread_join(tid, NULL);
9       exit(0);
10  }
11
12  void *thread(void *vargp) /* Thread routine */
13  {
14      printf("Hello, world!\n");
15      return NULL;
16  }
```

code/conc/hello


Main thread
call Pthread_create()
Pthread_create()
returns
call Pthread_join()
Main thread waits for
peer thread to terminate
Pthread_join()
returns
exit()
Terminates
main thread and
any peer threads
Peer thread
printf()
return NULL;
Peer thread
terminates

Posix线程

- 创建线程：线程通过调用pthread_create函数来创建其他线程。pthread_create返回时参数tid包含新创建线程的ID，新线程可通过调用pthread_self函数来获得自己的线程ID；

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   func *f, void *arg);
```

若成功则返回0，若出错则为非零

```
#include <pthread.h>

pthread_t pthread_self(void);
```

返回调用者的线程ID

Posix线程

- 终止线程：一个线程由以下方式之一来终止：1.顶层的线程例程返回时，线程会隐式地终止；2.通过调用pthread_exit函数，线程会显式地终止，返回值为thread_return；3.某个对等线程调用Linux的exit函数，该函数终止进程以及所有与该进程相关的线程；4.另一个对等线程通过以当前线程ID作为参数调用pthread_cancel函数来终止当前线程；

```
#include <pthread.h>

void pthread_exit(void *thread_return);
```

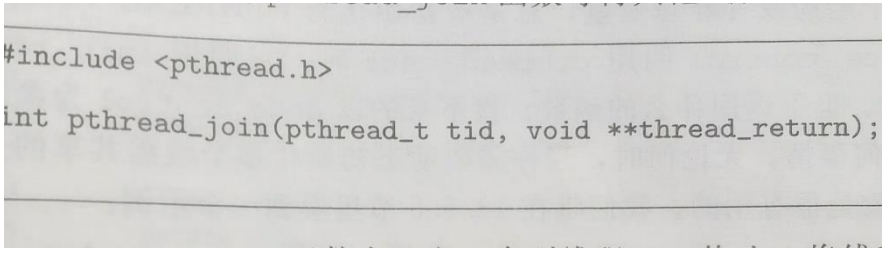
从不返回

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

若成功返回0，若出错则为非零

Posix线程

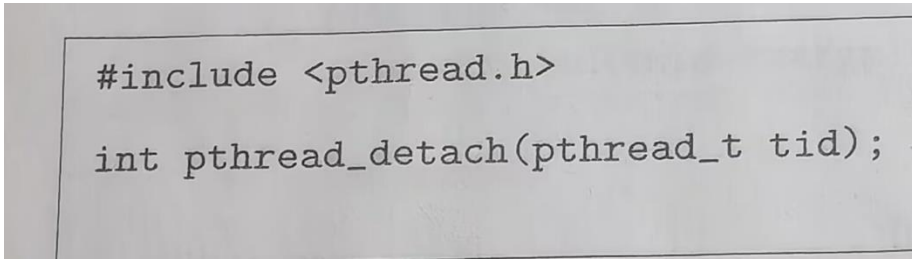
- 回收已终止线程的资源：线程通过调用pthread_join函数等待其他线程终止（pthread_join只能等待一个指定的线程终止）；



```
#include <pthread.h>
int pthread_join(pthread_t tid, void **thread_return);
```

若成功返回0，若出错则为非零

- 分离线程：默认情况线程被创建为可结合的。为避免内存泄漏，每个可结合线程都应被其他线程显式地收回，或通过调用pthread_detach函数被分离；（pthread_detach函数分离可结合线程tid）；



```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

若成功返回0，若出错则为非零

Posix线程

- 初始化线程：pthread_once函数允许初始化与线程例程相关的状态，可用于动态初始化多个线程共享的全局变量。

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
```

总是返回0

基于线程的并发服务器

```
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

- 为每个客户端生成了新线程
- 将连接文件描述符的副本传递给它

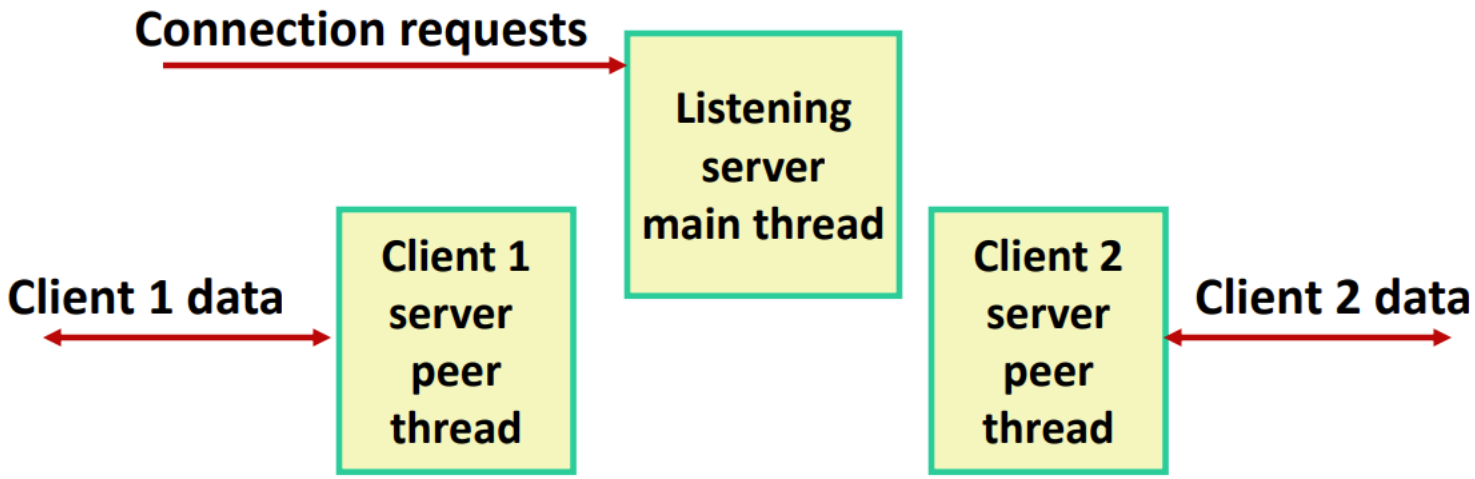
基于线程的并发服务器

```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

- 在分离的模式下运行线程可独立于其他线程运行，终止时由系统自动释放，避免内存泄漏
- 分配用于存放connfd的可用存储空间
- 关闭connfd（重点）（为什么只在一个位置关闭了已连接描述符？）

基于线程的服务器执行模型



- 每个客户端由单独的对等线程处理
- 线程共享除TID外的所有进程状态
- 每个线程都有一个单独的局部变量堆栈

基于线程的并发编程

- 基于线程的服务器需注意的问题：
- 1.必须分离运行以避免内存泄漏。线程在任何时候都是可结合或可分离的，而可结合的线程可被其他线程收回或杀死，已分离的线程不能被其他线程收回或杀死。内存资源在终止时由系统自动释放，而线程是默认可结合的。（在前面分离线程提到）
- 2.必须避免意外共享。（例如将指针传递到主线程的堆栈：pthread_create(&tid,null,thread,(void*)&connfd);
- 3.线程调用的所有函数都必须是线程安全的。

正确传递线程参数

```
/* 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
- 在main中分配，在线程例程中释放

基于线程的设计的优劣

- 优：
- 1.易于在线程之间共享数据结构；
- 2.比进程更加高效。
- 劣：
- 1.无意的共享可能引入细微且难以重视的错误；
- 2.难以知道哪些数据是共享的还是私有的；
- 3.难以通过测试检验。

1.下列关于 C 语言中进程模型和线程模型的说法中，错误的是：

A.每个线程都有它自己独立的线程上下文，包括线程 ID、程序计数器、条件码、通用目的寄存器值等

B.每个线程都有自己独立的线程栈，任何线程都不能访问其他对等线程的栈空间

C.不同进程之间的虚拟地址空间是独立的，但同一个进程的不同线程共享同一个虚拟地址空间

D.一个线程的上下文比一个进程的上下文小得多，因此线程上下文切换要比进程上下文切换快得多

解析：B

考察 12.3 12.4 线程模型，属于简单题。B 选项，不同的线程栈是不对其他线程设防的。所以，如果一个线程以某种方式得到一个指向其他线程栈的指针，那么它就可以读写这个栈的任何部分。