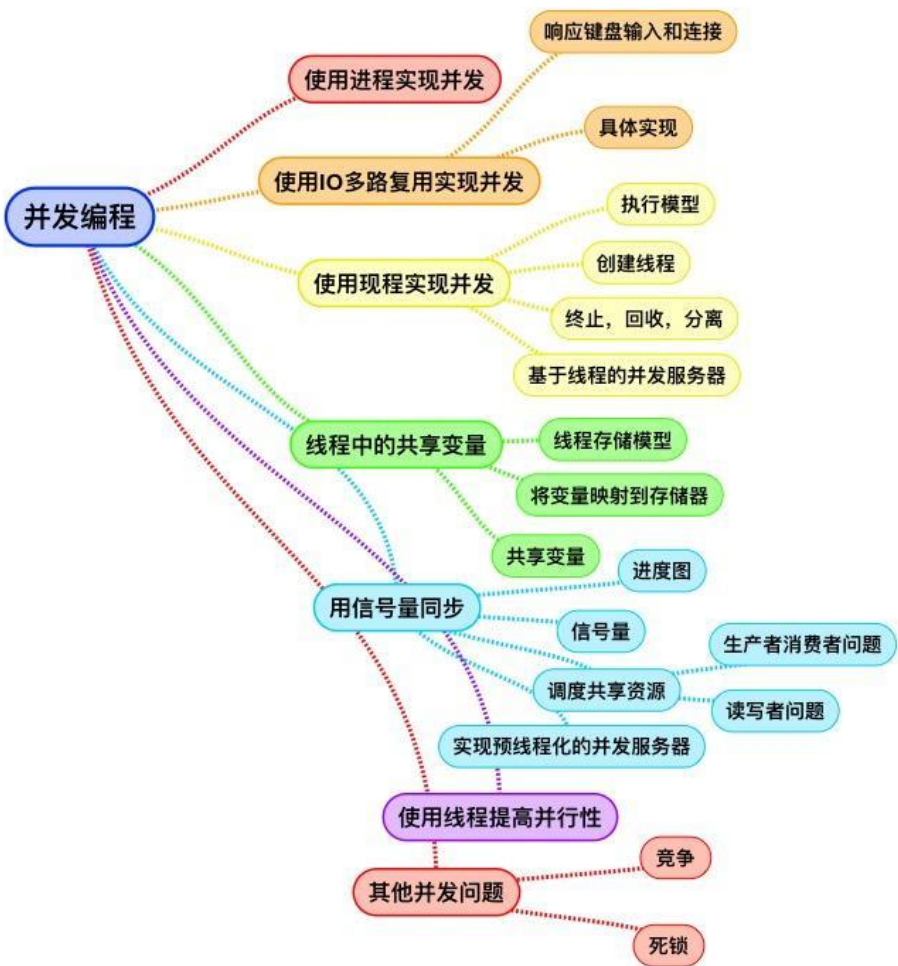


《深入理解计算机系统》并发编程



唐鱼的学习探索 关注

0.072 2018.05.30 21:18:14 字数 4,704 阅读 740



目 录

我们在上一章节中讲到的Tiny Web服务器只能为单个客户端提供访问，这一章里，我们将通过进程、多路复用和线程技术研究并发的服务器。

1.1 使用进程实现并发

我们实现过一个echo服务器，但是遗憾的是只能为一个客户端服务，这不是我们的初衷，现在我们来更新上一个版本，使得服务器在接收到连接请求的时候，创建子进程为该客户端提供服务，主进程会关闭已连接的描述符，继续监听下一个客户端，这一个过程我画了一个简图：

小鹅通

小鹅通商家数突破100万

在线直播课堂系统,最高直减5888元

立即注册

广告

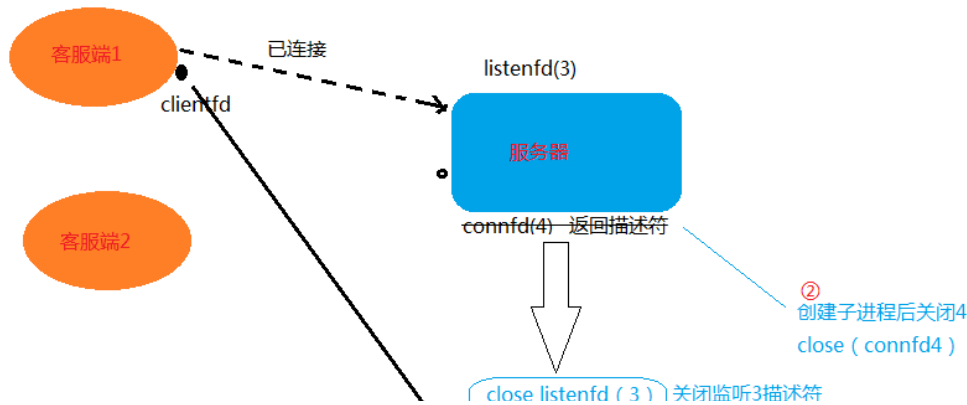


唐鱼的学习探索 关注

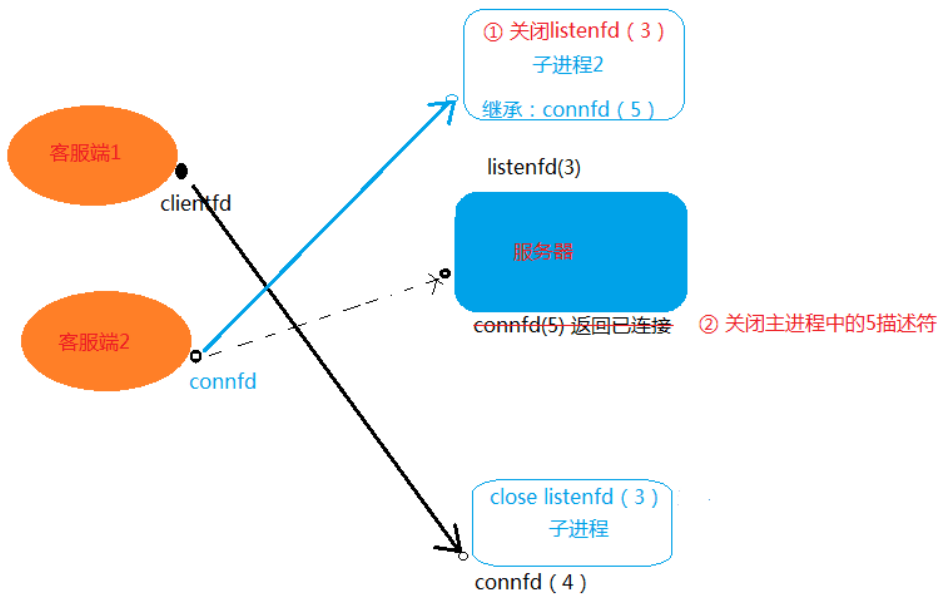
总资产29 (约2.81元)

如何高效的准备一次考试
阅读 1,737

都9102年了，你还不知道anki是什么
阅读 102



在这个过程中，客户端1连接上了服务器，并创建了一个已连接的描述符4，服务器立即派生子进程，子进程将继承原有的已连接描述符4，通过这个子进程的描述符为客户端1提供服务。这时候，主进程必须要关闭已连接描述符4，使得不至于发生内存泄漏。



客户端2的连接过程和客户端1的过程是一样的，还是由服务器创建子进程2提供服务，并关闭服务器中的已连接描述符5。我们来看看改进代码：只是加入了回收子进程，在子进程中关闭监听描述符和主进程中关闭已连接描述符。运行的效果如下：

```
pi@raspberrypi:~/code
cpfile echoserver kill setjmp signal.c waitpid.c
cpfile.c echoserver.c kill.c setjmp.c signal2
csapp.c echoserverp.c procmask shellx statcheck
csapp.h echoserverp.c.save procmask1.c shellx.c statcheck.c
pi@raspberrypi:~/code $ gcc csapp.c echoserverp.c -o echoserverp
/tmp/ccnm378x.o: In function 'main':
echoserverp.c:(.text+0xf4): undefined reference to 'echo'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~/code $ sudo nano echoserverp.c
Use "fg" to return to nano.

[1]+ Stopped sudo nano echoserverp.c
pi@raspberrypi:~/code $ gcc csapp.c echoserverp.c echo.c -o echoserverp
pi@raspberrypi:~/code $ ./echoserverp
usage: ./echoserverp <port>
pi@raspberrypi:~/code $ ./echoserverp 2017
server received 4 bytes
```

多进程 服务端 开启2017端口

可以同时为多个客户端提供服务，实现进程并发，进程级并发的一个明显的缺点是，各个进程都有独立的地址空间，使得共享信息相当困难而且慢速需要IPC，原理已经讲解了，代码就不难理解了：

```
pi@raspberrypi: ~/code
GNU nano 2.2.6 File: echoserverp.c

#include "csapp.h"

void echo(int connfd);

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

int main(int argc, char **argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;

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

    port = atoi(argv[1]);

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);

    while(1)
    {
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        if(Fork() == 0)
        {
            Close(listenfd);
            echo(connfd);
            Close(connfd);
            exit(0);
        }
        Close(connfd);
    }
}
```

回收僵尸进程

① 关闭子进程中的监听字符

② 关闭主进程中的已连接描述符

1.2 使用IO多路复用实现并发

应用程序在一个进程的上下文中显示的调度它们的逻辑流，逻辑流被模型化为状态机，数据到达文件描述后，主程序显示的从一个状态切换到另一个状态。

① 响应键盘输入和客户端连接

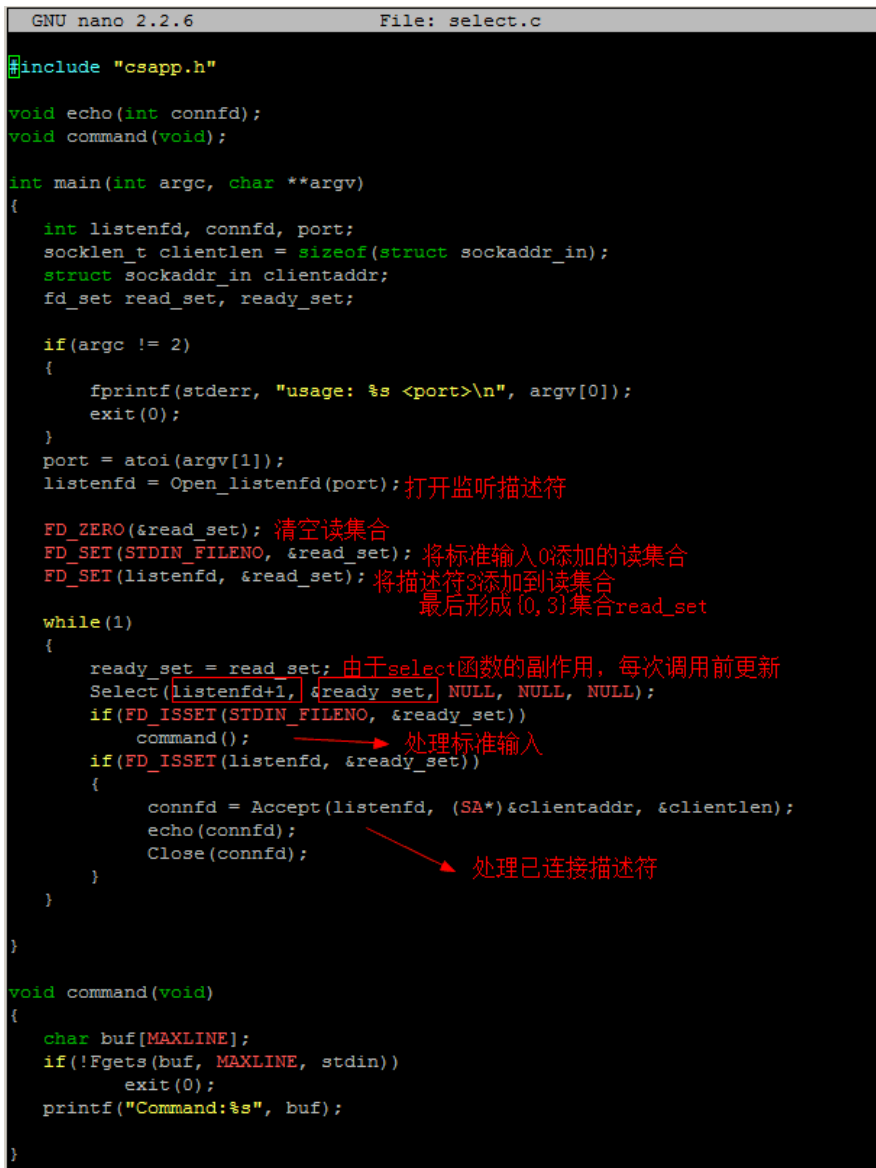
我们使用select函数创建一个描述符集合，当其中之一的描述符做好准备的时候，将控制权返回

给程序，select函数原型如下：

```
int select(int n, fd_set *fdset, NULL, NULL, NULL);
```

fdset被称为一个描述符集合，我们将需要处理的描述符添加到fdset集合中去；第一个参数n是描述符集合中最大的数。select函数会一直阻塞，直到相应的集合中的描述符准备好可以读；

我们来演示一个例子：



```
GNU nano 2.2.6 File: select.c

#include "csapp.h"

void echo(int connfd);
void command(void);

int main(int argc, char **argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;

    if(argc != 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port); 打开监听描述符

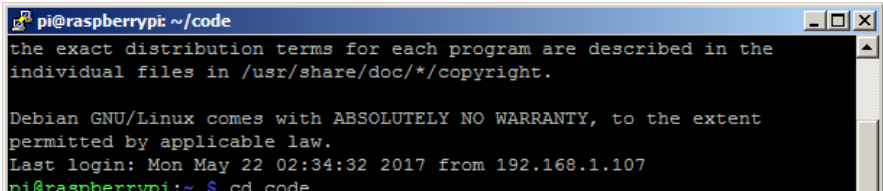
    FD_ZERO(&read_set); 清空读集合
    FD_SET(STDIN_FILENO, &read_set); 将标准输入0添加的读集合
    FD_SET(listenfd, &read_set); 将描述符3添加到读集合
                                最后形成 {0, 3} 集合 read_set

    while(1)
    {
        ready_set = read_set; 由于select函数的副作用，每次调用前更新
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if(FD_ISSET(STDIN_FILENO, &ready_set))
            command(); 处理标准输入
        if(FD_ISSET(listenfd, &ready_set))
        {
            connfd = Accept(listenfd, (SA*)&clientaddr, &clientlen);
            echo(connfd);
            Close(connfd); 处理已连接描述符
        }
    }
}

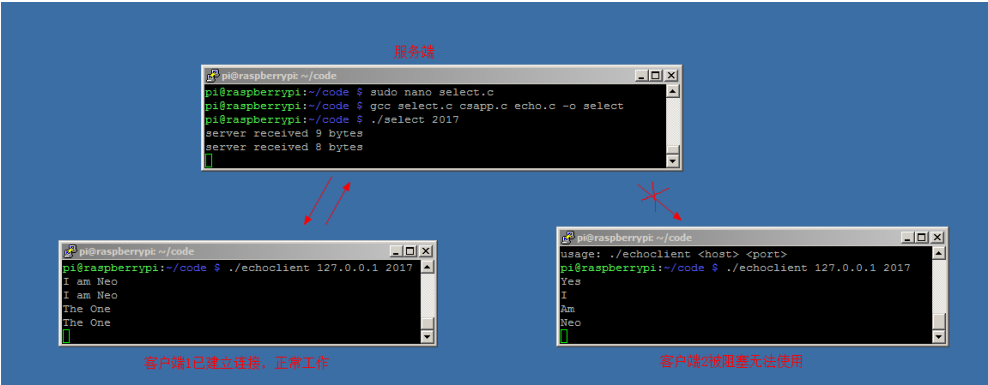
void command(void)
{
    char buf[MAXLINE];
    if(!Fgets(buf, MAXLINE, stdin))
        exit(0);
    printf("Command:%s", buf);
}
```

当我们打开了监听描述符以后，我们将一个read_set集合清空，并添加上标准输入和监听描述符3形成集合{0,3}，随后，我们进入一个无限循环，每次调用Select函数会阻塞，直到描述符0或者3到达时。

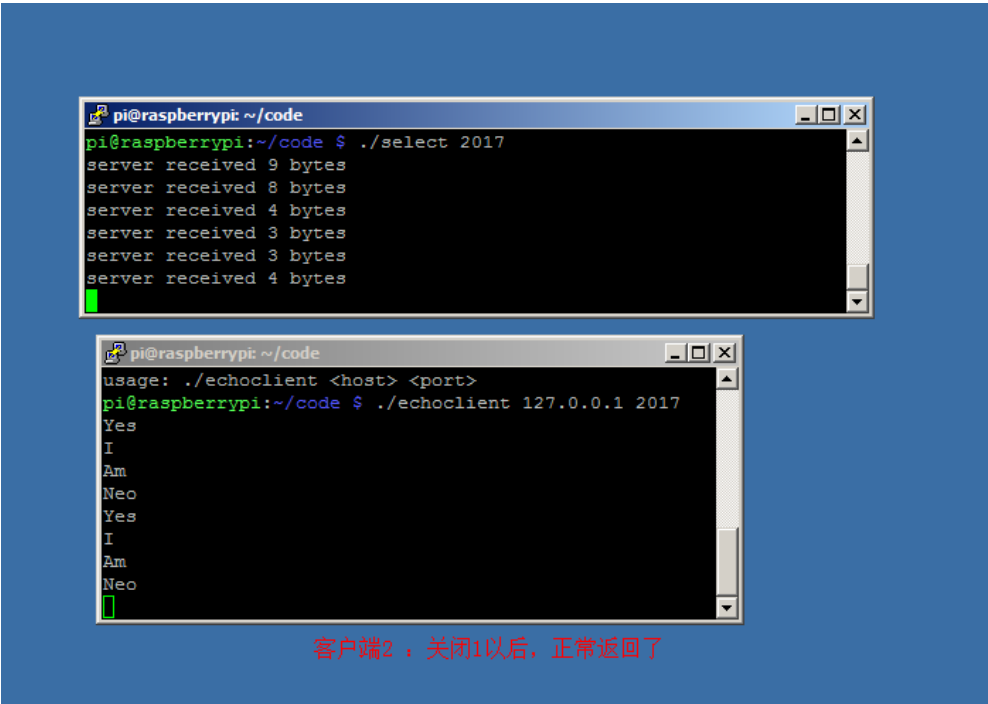
我们启动以后，随意输入内容，就会看到服务器首先响应了标准输入：



我们接下来启动 已连接描述符，就会发现一个问题：

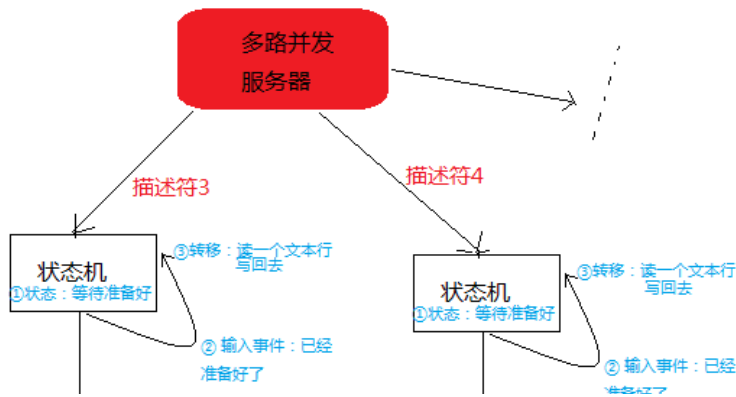


不论是服务端的标准输入，还是新启动的客户端2都被阻塞了。只有当已连接描述符客户端1关闭的时候才能使用。



一个解决之道是服务器每次循环最多回送一个文本行，就不会让已连接的描述符连续回送了。

② 多路复用实现并发



服务器为每一个客户端创建一个状态机，每个状态机三个阶段：

【准备】——【输入事件】——【写回】

我们来看看main函数主要部分：

```
GNU nano 2.2.6      File: echoservers.c

#include "csapp.h"

typedef struct
{
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready;
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
}pool;

void init_pool(int listenfd, pool *p);
void add_client(int connfd, pool *p);
void check_clients(pool *p);

int byte_cnt = 0;

int main(int argc, char **argv)
{
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    static pool pool;

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

    listenfd = Open_listenfd(port);
    init_pool(listenfd, &pool); // ①初始化状态机池塘
    while(1)
    {
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.read_set, NULL, NULL, NULL); // 阻塞检测有无新的连接到达
        if(FD_ISSET(listenfd, &pool.ready_set))
        {
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool); // ②新的连接到达，加入到pool池
        }
        check_clients(&pool); // ③转移，执行文本回送
    }
}
```

说明：活动的客户端是在pool池塘中，通过调用init_pool完成初始化后进入一个while循环，select函数检测两种不同的输入（新的连接、已经连接的描述符准备好可以读），当新的连接到达时，accept并add_client。最后使用check_clients函数将文本行回送。

分析：init_pool函数

```
void init_pool(int listenfd, pool *p)
{
    int i;
    p->maxi = -1;
    for(i=0; i < FD_SETSIZE; i++)
        p->clientfd[i] = -1;
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
```

已连接描述符为空
清空read_set集合
监听描述符是唯一的描述符

分析：add_client函数

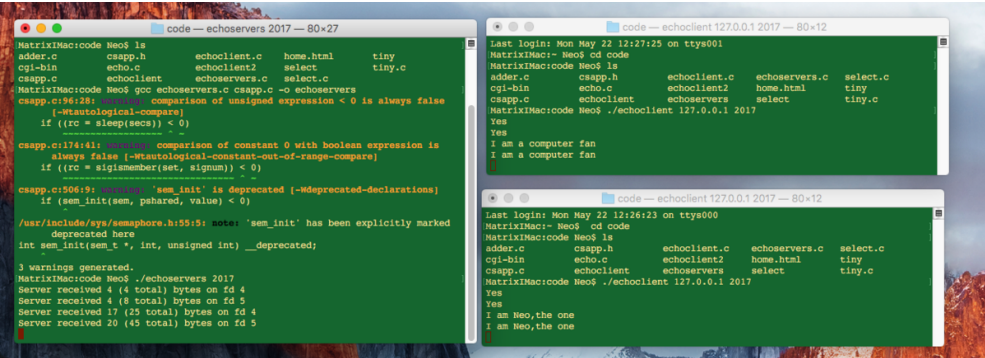
```
void add_client(int connfd, pool *p)
{
    int i;
    p->nready--;
    for(i=0; i < FD_SETSIZE; i++)
    {
        if(p->clientfd[i] < 0)
        {
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);
            FD_SET(connfd, &p->read_set);
            if(connfd > p->maxfd)
                p->maxfd = connfd;
            if(i > p->maxi)
                p->maxi = i;
            break;
        }
    }
    if(i == FD_SETSIZE)
        app_error("add_client error: Too many clients");
}
```

初始化为-1，小于0为空闲
添加到已连接数组中
初始化缓存区
将该描述符添加到读集合
最大描述符
clientfd数组最大索引

分析：check_clients函数

```
void check_clients(pool *p)
```

运行的效果如图：

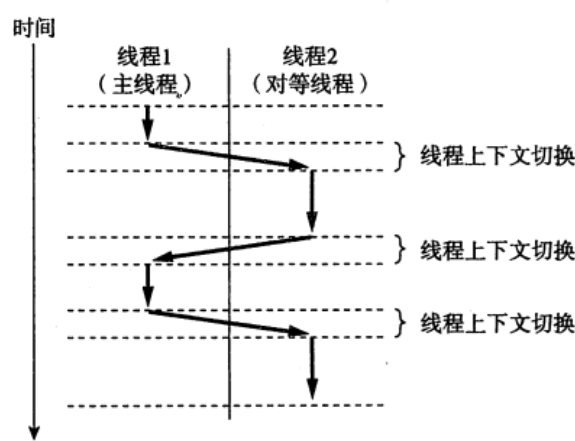


总结：我们这个版本的并发服务器，使用的是事件驱动的形式，它的优点就是共享数据的效果好很多，因为都是同一个进程上下文。开销也没有多进程的版本大，缺点就是复杂度要高些。总之，是优秀很多的。

1.3 基于线程的并发

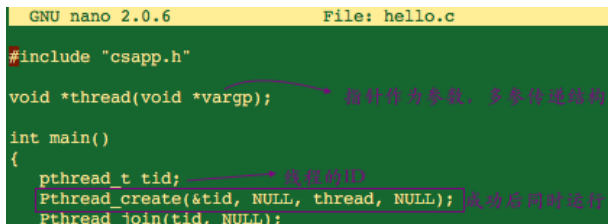
线程是一个运行在进程上下文中的逻辑流，由内核自动调度，集成了多进程与多路复用的优点，每个线程就像在舞台上跳舞的演员一样，各自分工和角色不一样，共享舞台的地址空间，当然也有自己私有的服装和台词。

① 执行模型



每个线程在开始的时候都是单一的主线程，这个主线程可以创建对等线程，然后两个线程并发执行，不断的切换上下文，分别执行一段时间。与进程之间不同的是线程的上下文切换要小的多，还有就是线程之间是完全对等的关系，也就是一个线程可以杀死它的对等线程。

我们来看一个简单的例子：



```
GNU nano 2.0.6 File: hello.c
#include "csapp.h"

void *thread(void *vargp);  // 指针作为参数，多条线程结构

int main()
{
    pthread_t tid;           // 线程的ID
    Pthread_create(&tid, NULL, thread, NULL); // 成功返回同时运行
    Pthread_join(tid, NULL);
}
```

主线程main中通过使用Pthread_create创建了一个新的tid线程，成功以后两个线程同时运行，主线程还使用了Pthread_join函数等待对等线程终止。对等线程只是简单的打印了一下Hello world。

② 创建线程

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

其中调用成功后tid是运行中的线程ID，attr设置线程默认属性，f是线程函数，arg是传递参数

可以使用：pthread_t pthread_self (void) 函数获取当前线程的ID；

③ 终止线程

原型：int pthread_cancel(pthread_t tid); 终止当前线程

原型：void pthread_exit(void *thread_return);等待所有对等线程终止

④ 回收已经终止的线程

原型：int pthread_join(pthread_t tid, void **thread_return);

函数会阻塞，直到线程tid终止并回收所有存储器资源。与wait不同的是该函数只能回收一个特定的线程；

⑤ 分离线程：分离后的线程终止以后由系统自动释放

原型：int pthread_detach(pthread_t tid);

⑥ 初始化线程

原型：int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));

⑦ 一个基于线程的并发服务器

```
GNU nano 2.0.6 File: echoserv.c
#include "csapp.h"

void echo(int connfd);
void *thread(void *vargp);

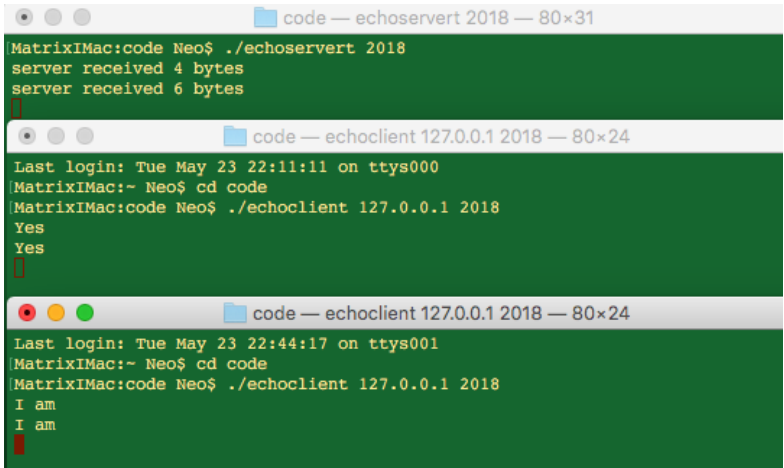
int main(int argc, char **argv)
{
    int listenfd, *connfdp, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if(argc != 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port);
    while(1)
    {
        connfdp = Malloc(sizeof(int)); // 动态分配，避免竞争
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    echo(connfd);
    Free(connfdp);
}
```

这个版本多线程的版本没有多大的变化，有两个地方需要注意，我们使用了一个connfdp指针指向一个动态分配的空间来传递已连接的描述符，避免出现竞争。同时在每个线程的函数中使用deattach进行分离，每个线程终止后由系统释放。

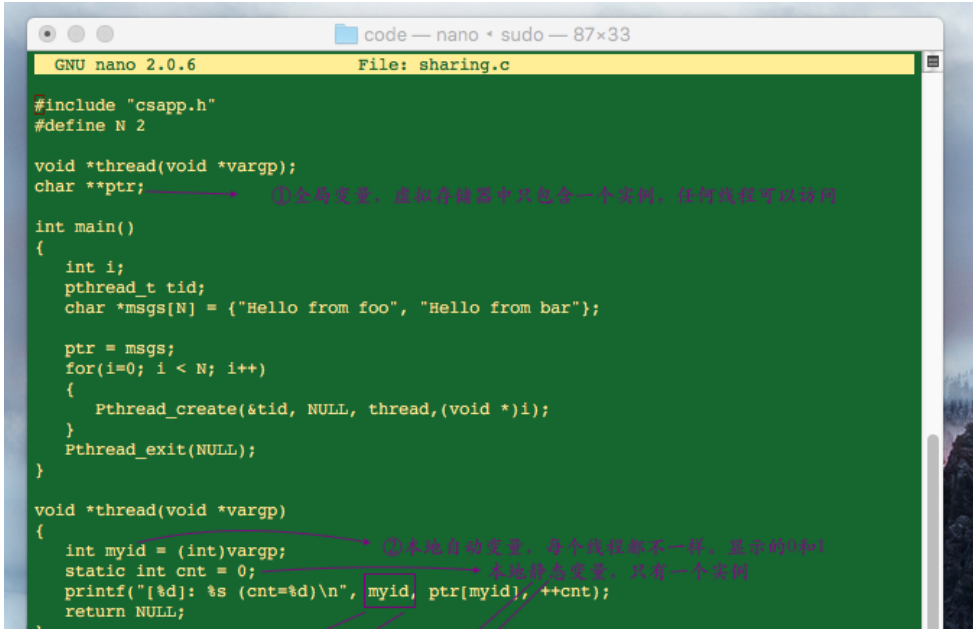
运行效果：



1.4 多线程中的共享变量

我们前面说过线程集中了多路复用中的共享的优点，也举例说了就像同一个舞台表演的不同演员一样，整个舞台空间是共享的。那么多线程中的共享是如何实现的，工作原理是什么？

我们看一个简单的例子，加入一些说明：



① 线程存储器模型

寄存器是不共享的，虚拟存储器总是共享的。就像同一个家庭的两个孩子一样，可以在一个饭厅吃饭，在客厅看电视，甚至共享同一个厕所，但是各自的房间通常是不一样的，各自的个人物品也不同。

② 将变量映射到存储器

- 全局变量：如ptr，可以使得本地变量msgs变成了共享（有时候两个孩子要共享一个厕所）；
- 本地自动变量：如myid是不能共享的，每个线程的myid都不一样；
- 本地静态变量：加入static如cnt，只有一个实例，两个对等线程访问的是同一个地方
- ③ 共享变量：被1个以上的线程访问过的变量，如cnt。需要注意的是msgs也变成了共享的。

1.5 用信号量同步线程

智人在进化意义上最成功的由于其合作的规模，单个的智人个体虽然远远不及同时代的尼安德特人，但是合作的规模更大，力量也就更大。我们今天探讨的就是线程的同步，如果每个线程都各顾各的，势必会影响到程序的正常运行。我们来看一个未经同步的线程的运行情况：

```
GNU nano 2.0.6      File: badcnt.c

#include "csapp.h"

void *thread(void *vargp);

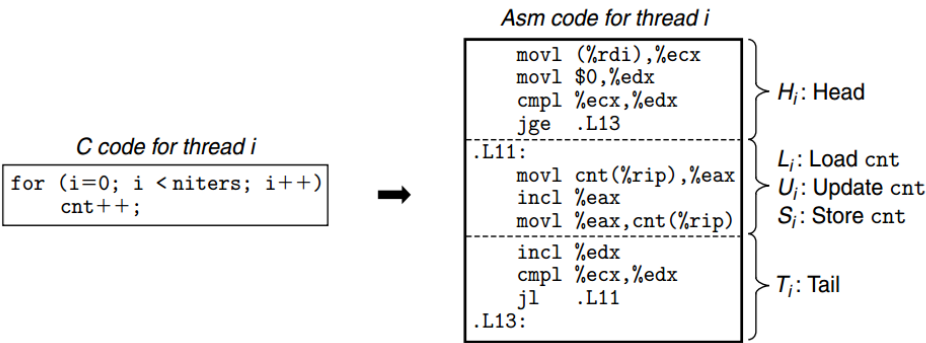
volatile int cnt = 0;

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

    if(argc != 2)
    {
        printf("usage: %s <niters>\n", argv[0]);
        exit(0);
    }
    niters = atoi(argv[1]);

    pthread_create(&tid1, NULL, thread, &niters);
    pthread_create(&tid2, NULL, thread, &niters);
}
```

这个程序的运行结果就不OK了，原因在于每个单独的进程对共享变量cnt的访问不是独占式的，这种不同步导致了错误的结果。我们来研究一下最核心的代码的运行过程：



这里我们将线程函数中的for循环翻译成汇编代码，其中：Li是循环头，Ti是循环尾，Li对应于加载cnt，Ui对应于更新cnt，Si对应于存储cnt。线程的执行顺序并不一定总是我们所期望的，如果遇到下面这种运行顺序，就可能会出错。

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H ₁	—	—	0
2	1	L ₁	0	—	0
3	1	U ₁	1	—	0
4	1	S ₁	1	—	1
5	2	H ₂	—	—	1
6	2	L ₂	—	1	1
7	2	U ₂	—	2	1
8	2	S ₂	—	2	2
9	2	T ₂	—	2	2
10	1	T ₁	1	—	2

(a) Correct ordering

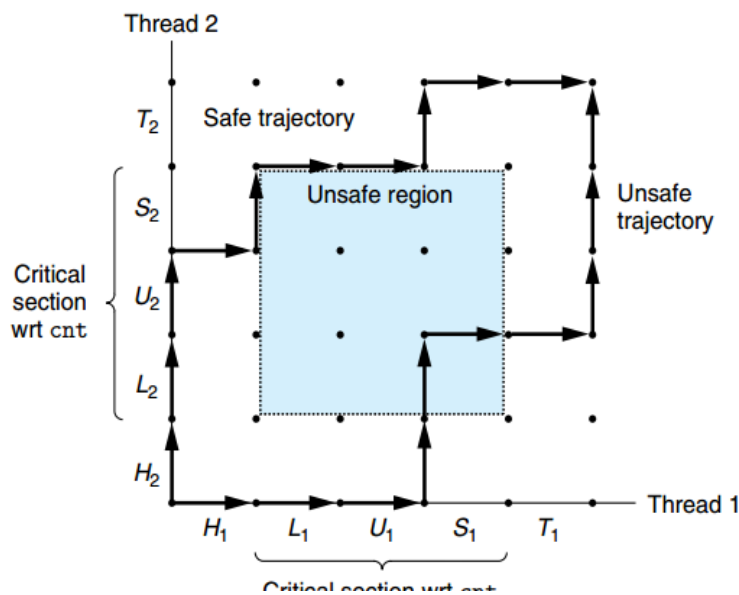
Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H ₁	—	—	0
2	1	L ₁	0	—	0
3	1	U ₁	1	—	0
4	2	H ₂	—	—	0
5	2	L ₂	—	0	0
6	1	S ₁	1	—	1
7	1	T ₁	1	—	1
8	2	U ₂	—	1	1
9	2	S ₂	—	1	1
10	2	T ₂	—	1	1

(b) Incorrect ordering

上图中左边是正确的运行顺序，（b）就会得到错误的结果，关键点在于线程1更新了eax的值以后并没有立即写入到cnt中，就开始运行了线程2，线程2由于cnt没有更新所有eax加载还是为0，当线程2完成写入命令以后cnt就仍然是1，不会得到累加。

为了帮助大家正确理解各个线程的执行顺序，我们来画图

① 进度图



上图展现了两个线程，1和2，分别用x轴和y轴表示，其中Hi、Li、Ui、Si、Ti分别代表对共享变成操作的for循环的关键步骤，其中Li、Ui、Si涉及对cnt临界区的操作，所有经过这一区域的执行顺序都是不安全的。为了使得线程之间的同步变得科学，不跨越临界区。我们发明了信号量这种特殊的变量。

② 信号量：非负整数全局变量

信号量s其实就是一个非负整数的全局变量，对这一变量有两个操作：P（s）使得s减1，而V（s）使得s加1。我们操作信号量s的时候，通常的情况是将其初始化为1，执行P操作的时候为加锁，执行V操作的时候为解锁。为了限定线程不经由不安全区域，我们将不安全区域的设置为-1，如下图：

我们的信号量s被初始化为1，只能在0和1之间变化：

1>加锁：执行P（s），有两种情况，如果原有的值为1，那么减至0；如果为0则挂起线程；

2>解锁：执行V（s），也有两种情况，如果s=0就加1；如果s=1就等待；

③ 更新我们的badcnt程序

这样以来我们的全局共享变量cnt在运行的各个线程中就会经由加锁执行++和解锁，得到正确的结果了。

④ 信号量调度共享资源

生产者——消费者问题

以小区的自动售货机为例，消费者如果直接以下订单的方式与生产者沟通，这样的效率就太低下了。我不可能想要喝一瓶可能才让可口可乐公司给我生产。这时候缓冲区就是一个很好的发明，我们发现在小区建立几个自动售货机，假设每个自动售货机可以装100瓶饮料。这样一来

只要自动售货机不为空生产者就可以将饮料放入到自动售货机中去，当然只要售货机有饮料消费者也直接从自动售货机购买饮料。这样一来就方便的多了。

我们前面讲过信号量，P操作遇到为0的情况就会等待。但是现实的生活中，这样的情况就不很科学。回到我们上面的自动售货机的例子。如果我们的消费者发现了自动售货机是空的，我们就开始在原地等待，直到生产者将生产好的饮料送到自动售货机上的时候，再购买。这样以来对个人来说是精力的极大浪费。我们有什么好的方法没有，就像我们滴滴打车一样，我们下单以后就可以去做其他事情了，一有车子接单以后就会电话联系我们。

我们使用一种新的数据结构来解决这种问题：

```

1  typedef struct {
2      int *buf;
3      int n;
4      int front;
5      int rear;
6      sem_t mutex; 互斥锁
7      sem_t slots; 生产者空位
8      sem_t items; 消费者可
9  } sbuf_t;

```

操作函数

```

void sbuf_init(sbuf_t *sp, int n) 初始化
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;
    sp->front = sp->rear = 0;
    Sem_init(&sp->mutex, 0, 1);
    Sem_init(&sp->slots, 0, n);
    Sem_init(&sp->items, 0, 0);
}

void sbuf_deinit(sbuf_t *sp) 析构
{
    Free(sp->buf);
}

void sbuf_insert(sbuf_t *sp, int item) 插入
{
    P(&sp->slots);
    P(&sp->mutex);
    sp->buf[(++sp->rear)%(sp->n)] = item;
    V(&sp->mutex);
    V(&sp->items);
}

int sbuf_remove(sbuf_t *sp) 删除
{
    int item;
    P(&sp->items);
    P(&sp->mutex);
    item = sp->buf[(++sp->front)%(sp->n)];
    V(&sp->mutex);
    V(&sp->slots);
    return item;
}

```

读者——写者问题

这个问题类似于上一个，有点儿像我们的购票系统，票数就是我们的共享变量，同一时刻我们允许多个客户从不同的端口登录查看票数在售情况（读者优先），但是当有一个购买者（写者）的买票的时候，写会独占票数。有一个解答如下：

```

1  int readcnt;           // 当前读者数量
2  sem_t mutex, w;       // w代表临界区、mutex提供互斥功能
3
4  void reader(void)
5  {
6      while(1)
7      {
8          P(&mutex);
9          readcnt++;
10         if(readcnt == 1 // 第一个读者进入加锁
11             P(&w);
12         V(&mutex);
13
14         P(&mutex);
15         readcnt--;
16         if(readcnt == 0 // 最后一个读者解锁
17             V(&w);
18         V(&mutex);
19     }
20 }
21
22 void writer(void)
23 {
24     while(1)
25     {
26         P(&w);
27
28         // 一系列写入操作
29
30         V(&w);
31     }
32 }
33

```

⑤ 实现一个预线程化的并发服务器

我们通常所用到的线程并发服务器，要求服务器为每个客户端单独生成一个线程来提供服务，就相当于一种下订单再生产的落后经济模型，我们学习了生产者消费者模型以后，尝试加入新的内容：服务器 由一个主线程和一组工作线程构成，主线程接收客户端的连接请求，并将连接的描述符放入到一个缓冲池中，每个工作线程反复的从缓冲池中取出描述符，提供服务，然后等待下一个描述符。

我们来看看实现代码：

```
#include "csapp.h"
#include "sbuf.h"

#define NTHREADS 4
#define SBUF_SIZE 16

void echo_cnt(int connfd);
void *thread(void *vargp);

sbuf_t sbuf;

int main(int argc, char **argv)
{
    int i, listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if(argc != 2)
    {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    sbuf_init(&sbuf, SBUF_SIZE); 初始化缓冲区sbuf
    listenfd = Open_listenfd(port);

    for(i=0; i < NTHREADS; i++) 创建4个工作线程
        Pthread_create(&tid, NULL, thread, NULL);
    while(1)
    {
        接受连接, 插入到缓冲区sbuf
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd);
    }

    void *thread(void *vargp)
    {
        Pthread_detach(pthread_self()); 分离线程, 自行回收
        while(1)
        {
            int connfd = sbuf_remove(&sbuf); 从sbuf中取出一个已连接的描述符
            echo_cnt(connfd); 执行后关闭
            Close(connfd);
        }
    }
}
```

1.6 使用线程提高并行性

现代的CPU往往是多核的，如何利用这个特性变得相当重要。我们这里所的并行是并发的一个子集，代表的是在多核处理器上运行的并发程序。

如果我们要计算1,2,3..... 100各个数字相加的和，我们知道经典的答案是：(1+100)

*50=5050，我们使用多线程求一个集合数字的和的方法，就是将100个数字分成5个区域，这样每个区域有20个数字，每个线程求出5个区域20个数字的和，然后由主线程将不同的和相加，就会得到这100个数字的和。我们来看一段代码：

```

1  #include "csapp.h"
2  #define MAXTHREADS 32
3
4  void *sum(void *vargp);
5
6  long psum[MAXTHREADS]; //对等线程和
7  long nelems_per_thread;
8
9  int main(int argc, char **argv)
10 {
11     long i, nelems, log_nelems, nthreads, result = 0;
12     pthread_t tid[MAXTHREADS];
13     int myid[MAXTHREADS];
14
15     if(argc != 3)
16     {
17         printf("Usage: %s <nthreads> <log_nelems>\n", argv[0]);
18         exit(0);
19     }
20     nthreads = atoi(argv[1]);
21     log_nelems = atoi(argv[2]);
22     nelems = (1L << log_nelems);
23     nelems_per_thread = nelems / nthreads; // 每个线程元素个数
24     // 创建nthreads个对等线程, 将myid传递给sum函数
25     for(i=0; i < nthreads; i++)
26     {
27         myid[i] = i;
28         Pthread_create(&tid[i], NULL, sum, &myid[i]);
29     }
30     // 等待所有线程终止
31     for(i=0; i < nthreads; i++)
32     {
33         Pthread_join(tid[i], NULL);
34     }
35     // 将每个对等线程的求和psum加入到result中
36     for(i=0; i < nthreads; i++)
37         result += psum[i];
38     // 判断结果是否正确

```

再来看看求和线程函数sum:

```

1  void *sum(void vargp)
2  {
3      int myid = *((int *)vargp); //解析传递参数
4      long start = myid * nelems_per_thread; // 计算开始位置
5      long end = start + nelems_per_thread; // 计算结束位置
6      long i sum = 0;
7
8      for(i=start; i < end; i++)
9          sum += i;
10     psum[myid] = sum; // 将结果保持到全局共享变量psum中
11
12     return NULL;
13 }

```

运行结果如下:

1.7 其他并发问题

我们在实现程序的并发操作中, 要注意很多问题。包括对共享变量的互斥访问, 使得程序无论何时何系统, 都能得到正确的返回值。不安全的操作有以下四类:

1> 不保护共享变量的函数;

- 2> 保持跨越多个调用状态的函数 (rand、srand) ；
- 3> 返回指向静态变量指针的函数 (ctime) ；
- 4> 调用线程不安全函数的函数；

说明：对于第3类函数，我们通常使用的是加锁——拷贝模式：

```
1 char *ctime_ts(const time_t *timep, char *privatep)
2 {
3     char *sharedp;
4
5     P(&mutex); // 加锁
6     sharedp = ctime(timep); // 执行3类函数
7     strcpy(privatep, sharedp); // 将返回的结果拷贝靠私有空间
8     V(&mutex); // 解锁
9     return privatep;
10 }
```

① 在库函数中使用_r版本

线程不安全函数	线程不安全类	Unix 线程安全版本
rand	2	rand_r
strtok	2	strtok_r
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(无)
localtime	3	localtime_r

以上我们列出的是线程不安全函数的_r版本，这些版本不会引用共享的数据，因而在线程中使用是安全的，我们推荐使用_r版本的这类函数。

② 竞争

要理解竞争我们最好先来看一个例子：

```
1 #include "csapp.h"
2 #define N 4
```

这是一个很简单的程序，在主线程中11-12行创建了4个对等线程，分别给每个对等线程传递了一个本地变量*i*，期望在线程函数中将每个对等线程的id号输出显示。

当竞争发生的时候：

如果：先创建了一个线程(1)，传递了本地变量1到线程函数thread中，并显示，这是合理的

如果：创建线程后，thread函数还未输出结果，就切换到主线程又创建新线程就会发生竞争

在不同的系统上得到了不同的结果，我们的改进方法如下：

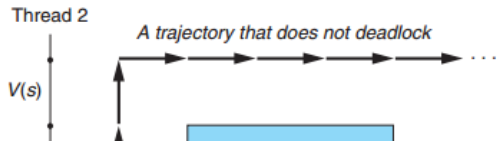
```
1 #include "csapp.h"
2 #define N 4
3
4 void *thread(void *vargp);
5
6 int main()
7 {
8     pthread_t tid[N];
9     int i, *ptr;
10
11     for(i=0; i < N; i++)
12     {
13         ptr = Malloc(sizeof(int));
14         *ptr = i;
15         Pthread_create(&tid[i], NULL, thread, ptr);
16     }
17     for(i=0; i < N; i++)
18         Pthread_join(tid[i], NULL);
19     exit(0);
20 }
21
22 void *thread(void *vargp)
23 {
24     int myid = *((int *)vargp);
25     Free(vargp);
26     printf("Hello from thread %d\n", myid);
27     return NULL;
28 }
```

分配独立的块

传递指向该出的指针

使用后释放

③ 死锁



死锁是由于我们交替对一对互斥变量（s、t）加锁，如上图所示，线程1先对s加锁，线程2先对t加锁，然后线程1要求对t加锁的时候就必须等待，线程2要求对s加锁的时候也陷入了等待，两个线程都在等待就死锁了。解决之道很简单：

线程按照相同的顺序对s、t加锁，也就是说线程1先加锁s再加锁t，线程2先加锁s再加锁t。

简书

首页

下载APP

搜索

Q

Aa

beta

登录

注册

7人点赞 >

《深入理解计算机系统》

...

7赞

赞赏支持

还没有人赞赏，支持一下

唐鱼的学习探索

如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。

总资产29 (约2.81元) 共写了10.4W字 获得530个赞 共463个粉丝

关注

写下你的评论...

推荐阅读

虐恋东风 13 九儿你终于是我的女人了
阅读 7,986

撒贝宁“洋媳妇”3年判若两人，网友感叹中国老婆好
阅读 54,395

跟钢铁直男相处是什么体验？
阅读 21,282

撑过去，你会遇见未知的自己
阅读 3,511

枕上书续(十九)冬瓜汤与白凤丸
阅读 7,516

小鹅通

小鹅通商家数突破100万

— 在线直播课堂系统,最高直减5888元 —

立即注册

广告

全部评论 1

只看作者

按时间倒序 按时间正序

WadeLiu威

2楼 2017.12.11 15:58

博主和我的看书路线基本相同，一起加油

赞 回复

被以下专题收入，发现更多相似内容

 《深入理解计算...


 程序员

 网络

推荐阅读

iOS 程序员的自我修养 — 读《程序员的自我修养-链接、装载...

2016年国庆假期终于把此书过完，整理笔记和体会于此。关于书名 书名源于俄罗斯的演员斯坦尼斯拉夫斯基创作的《演员...

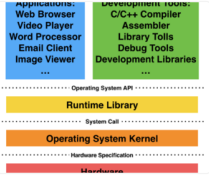
 李剑飞的简书

阅读 4,249

评论 1

赞 53

更多精彩内容>



写下你的评论...

 评论1

 赞7



转日 · Youternr<https://www.cnblogs.com/youternome/archive/2011...>

 njukay


阅读 882

评论 0

赞 52

Linux系统编程（三） ----- 多线程编程

一、线程的创建和调度 1.线程是程序执行的某一条指令流的映像。为了进一步减少处理机制的空转时间，支持多处理器及减...

 穹蓝奥义

阅读 712

评论 2

赞 4

7.线程

线程 在linux内核那一部分我们知道，线程其实就是一种特殊的进程，只是他们共享进程的文件和内存等资源，无论如何对...

 大雄good

阅读 183

评论 0

赞 2

```
1:main
2:
3:complete
4:ad 1 second handler
5:complete
6:ad 2 second handler
7:ad 2 first handler
8:operation fault ./cleanup
9:cleanup
10:
11:complete
12:complete
13:ad 1 second handler
14:ad 2 second handler
15:ad 2 first handler
16:operation fault ./cleanup
```

遇见你之前，我是谁?遇见你之后，我该如何抉择？

1.前一阵子在网看到一部很棒的电影《遇见你之前》。说实话会去看这部电影只是因为男女主角较高的颜值以及作为单身汪而...

 珊der

阅读 165

评论 0

赞 0