Concurrent Programming & Synchronization: Basic & Synchronization: Advanced

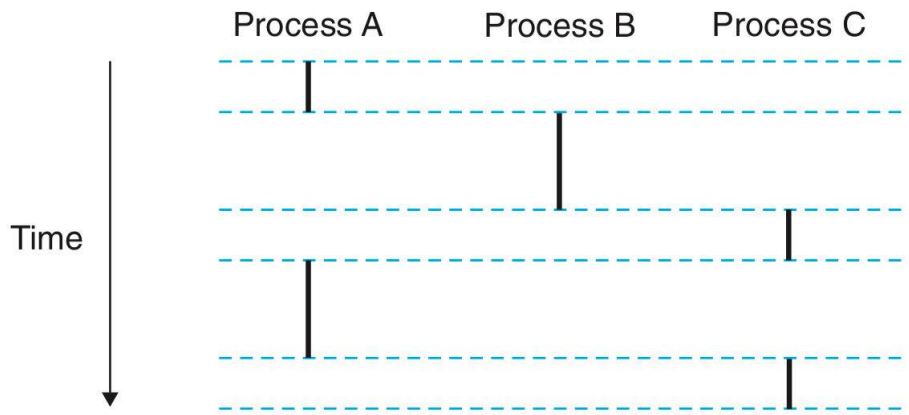
朱家启 徐梓越 许珈铭

2023.12.20

Concurrent Programming & Synchronization: Basic (CS:APP Ch. 12.1-12.4)

徐梓越

回顾“并发”



控制流在时间上重叠，那么它们就是并发的（concurrent）

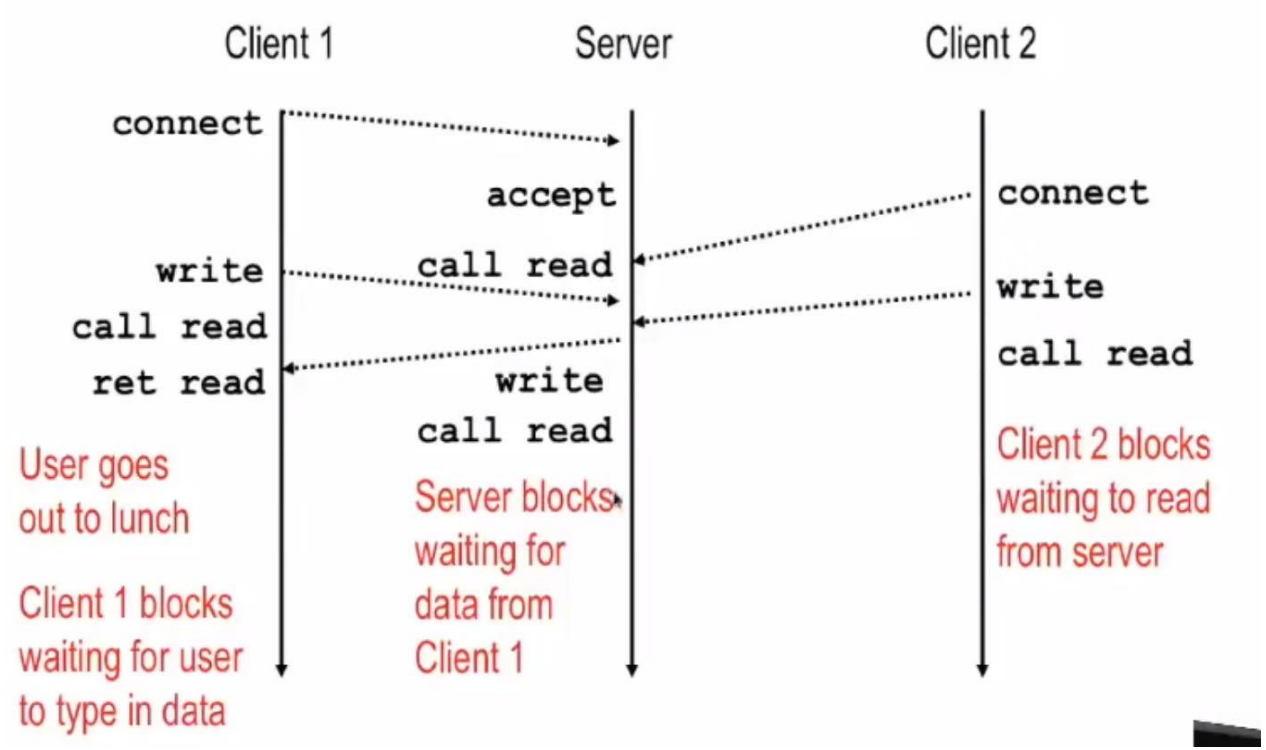
并发（concurrency）

看作是一种操作系统用来运行多个应用程序的机制。
不仅仅局限于内核，也可以在应用程序中扮演重要角色。

应用级并发的使用

访问慢速I/O设备
与人机交互
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运算

回顾迭代echo服务器 iterative server



假如：客户端的进程阻塞，那么会阻止所有其他想要访问服务器服务的客户端

Deadlock

Races, livelock/starvation/fairness

并发

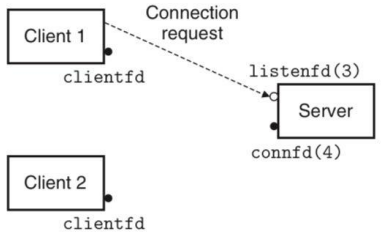
- 进程
- I/O多路复用
- 线程

构造并发程序的方法

- Process-based
- 本质：内核处理所有的调度，自动交错的执行进程
- 特点：每个流都有自己私有的地址空间
- Event-based
- 本质：程序员手动调度流程
- 特点：是一个程序，所以烘箱相同地址空间
- Thread-based
- 1+1
- 内核自动分出线程，但是共享相同的地址空间

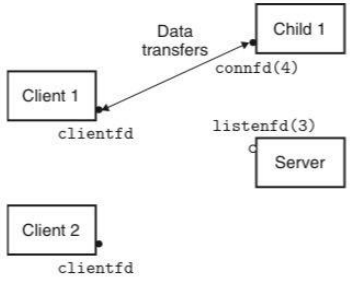
基于进程的并发编程

Figure 12.1
Step 1: Server accepts connection request from client.



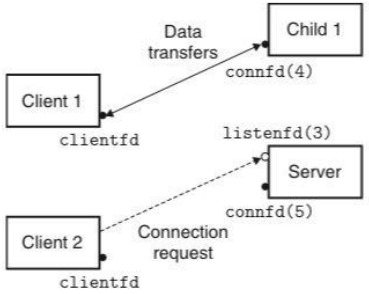
Step 1：服务器接受客户端的连接请求

Figure 12.2
Step 2: Server forks a child process to service the client.



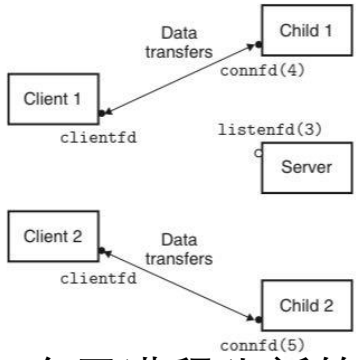
Step 2：服务器派生一个子进程为这个客户端服务

Figure 12.3
Step 3: Server accepts another connection request.



Step 3：服务器接受另一个连接请求

Figure 12.4
Step 4: Server forks another child to service the new client.



Step 4：服务器派生另一个子进程为新的客户端服务

Echo并发代码实现

code/conc/echoserverp.c

```
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.c

Figure 12.5 **Concurrent echo server based on processes.** The parent forks a child to handle each new connection request.

传入服务器监听的端口号

监听描述符和连接描述符

注意

SIGCHILD 处理程序，收回僵死子进程
避免内存泄漏

父进程必须关闭已连接描述符，否则：
①内存泄漏
②与该描述符相关联的状态会永远存在（内核不会自动关闭）

当套接字的文件表表项中的引用计数，指导父子进程的connfd都关闭了，到客户端的连接才会终止

进程并发的优劣

- 特点：进程的一个清晰的模型：共享文件表——但是不共享用户地址空间，父子进程都有私有地址空间
- pros：

一个进程不可能不小心覆盖另一个进程的虚拟内存，避免内存遗漏，就不会有令人迷惑的错误

- cons：

1.独立的地址空间使得进程共享状态信息变得更加困难

2.比较慢，进程控制和IPC的开销很高

Unix IPC Interproccess communication

- 所有允许进程和同一台主机上其他进程进行通信的技术，包括管道、FIFO、系统V共享内存、系统V信号量（semaphore）

- 在书中出现过的IPC：第八章的waitpid函数和信号，第十一章的套接字接口

新的问题

用户从标准输入键入的交互命令做出响应

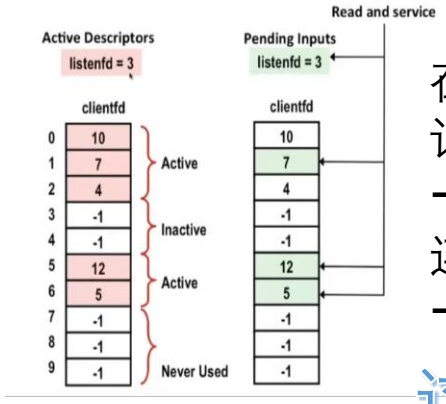
（1）那么我们需要网络客户端发起连接请求 （2） 用户在键盘上键入命令行
但是……
在accept中等待一个连接请求->我们不能相应输入的命令
Read中等待一个输入命令->不能相应任何连接请求

I/O 多路复用技术

基于I/O多路复用的并发编程

- 基本思想是“select 函数”：内核挂起进程，只有在一个或多个I/O事件发生后，才将控制返回应用程序
- Select函数： int select (int n, fd_set *fdset, NULL, NULL, NULL);

基数　　描述符集合



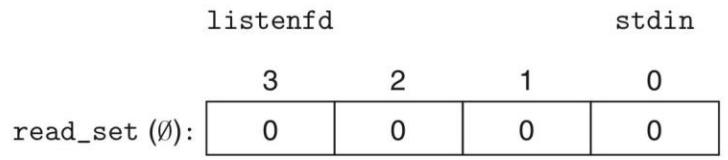
在使用一些连接描述符，其中记录每个描述连接和符号
→确定这里面的监听描述符、连接描述符……是不是可用的
→读取数据,执行工作

这些响应是并发流

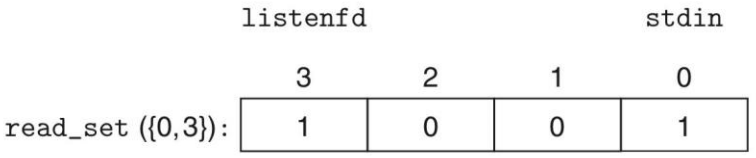
fd_set 又叫描述符集合
一个n维向量（b_{n-1}，…，b_1，b_0）
当b_k=1，描述符k才表明是描述符集合的一个元素
只能 分配它们 同类型变量赋值 用宏来修改和检查

副作用：会修改fdset指向为准备好集合（ready set）的基数
因此我们每次调用select时都更新读集合

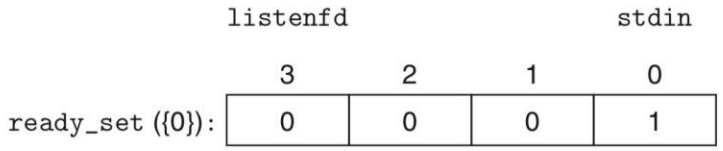
Select函数的实现例子



首先，用open_listenfd函数打开一个监听描述符，然后FD_ZERO创建一个空的读合集



监听描述符的读合集：{0,3} 0-标准输入 3-监听描述符;
调用select函数，一直保持阻塞，指导监听描述符或标准输入符准备号可以读



以右图为例，是用户按回车的情况。当描述符可读时，select函数返回ready_set的值{0}，同时FD_ISSET来指令确定那个描述符。
当0-标准输入准备好，调用command函数；如果3-监听描述符准备好，调用accept函数得到一个连接描述符（connect）然后调用echo函数

利用select实现一个迭代echo服务器

- 逻辑流模型转化为状态机——状态、输入事件、转移（制造映射）
- 循环迭代服务器用select函数来检测：

①新客户端的连接请求到达 ②一个已存在的客户端的已连接描述符准备好可以读

①情况→服务器打开连接，调用add_client函数，将客户端添加到池里

②情况→调用check_client函数，把来自每个准备好的已连接描述符的一个文本行送回去

利用select实现一个迭代echo服务器

code/conc/echoservers.c

```
1  void init_pool(int listenfd, pool *p)
2  {
3      /* Initially, there are no connected descriptors */
4      int i;
5      p->maxi = -1;
6      for (i=0; i< FD_SETSIZE; i++)
7          p->clientfd[i] = -1;
8
9      /* Initially, listenfd is only member of select read set */
10     p->maxfd = listenfd;
11     FD_ZERO(&p->read_set);
12     FD_SET(listenfd, &p->read_set);
13 }
```

code/conc/echoservers.c

Figure 12.9 `init_pool` **initializes the pool of active clients.**

Init_pool 函数 初始化客户端池

code/conc/echoservers.c

```
1  void add_client(int connfd, pool *p)
2  {
3      int i;
4      p->nready--;
5      for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
6          if (p->clientfd[i] < 0) {
7              /* Add connected descriptor to the pool */
8              p->clientfd[i] = connfd;
9              Rio_readinitb(&p->clientrio[i], connfd);
10
11             /* Add the descriptor to descriptor set */
12             FD_SET(connfd, &p->read_set);
13
14             /* Update max descriptor and pool high water mark */
15             if (connfd > p->maxfd)
16                 p->maxfd = connfd;
17             if (i > p->maxi)
18                 p->maxi = i;
19             break;
20         }
21     if (i == FD_SETSIZE) /* Couldn't find an empty slot */
22         app_error("add_client error: Too many clients");
23 }
```

code/conc/echoservers.c

Figure 12.10 `add_client` **adds a new client connection to the pool.**

Add_client 函数添加一个新的客户端到活动客户端中

code/conc/echoservers.c

```
1  void check_clients(pool *p)
2  {
3      int i, connfd, n;
4      char buf[MAXLINE];
5      rio_t rio;
6
7      for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
8          connfd = p->clientfd[i];
9          rio = p->clientrio[i];
10
11         /* If the descriptor is ready, echo a text line from it */
12         if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
13             p->nready--;
14             if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
15                 byte_cnt += n;
16                 printf("Server received %d (%d total) bytes on fd %d\n",
17                        n, byte_cnt, connfd);
18                 Rio_writen(connfd, buf, n);
19             }
20
21             /* EOF detected, remove descriptor from pool */
22             else {
23                 Close(connfd);
24                 FD_CLR(connfd, &p->read_set);
25                 p->clientfd[i] = -1;
26             }
27         }
28     }
29 }
```

code/conc/echoservers.c

Figure 12.11 `check_clients` **services ready client connections.**

Check_clients 函数回送来自每个准备好的已连接描述符的一个文本行

Event-based servers的优劣

- pros：

1.他只有一个地址空间的进程，可以使用gdb等来剖析每一行代码，方便调试——programmer friendly

2.没有进程或线程控制开销，可以访问进程的全部地址空间，因此正常开销小

- cons：

1.需要弄清回应一个事件需要的工作量，面对异常请求的解决问题→更细粒度的多路复用，需要复杂的编码

2.不能充分利用多核处理器，自动处理设计是一个顺序程序

小结

基于进程的并发编程

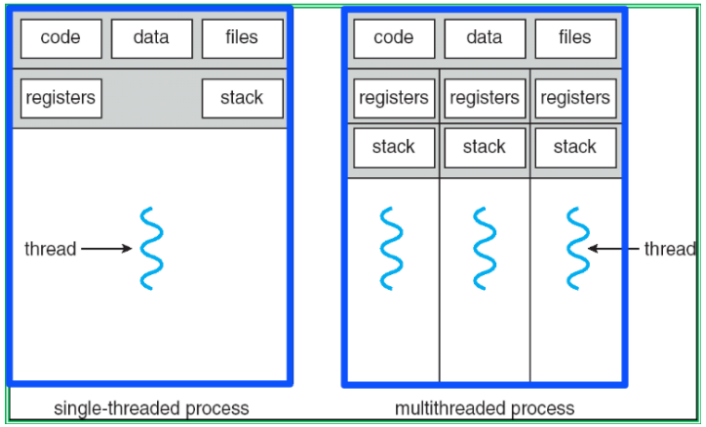
- 单独的进程
- 内核会自动调度每个进程
- 每个进程有它自己的私有地址空间，这使得流共享数据很困难

基于I/O多路复用的并发编程

- 创建自己的逻辑流，利用I/O多路复用来显示地调度流。
- 所有的流共享整个地址空间。

基于线程的并发编程

基于线程的并发编程



- 运行在进程上下文中的逻辑流
- 独立包括：整数线程ID（Thread ID）、栈、栈指针、程序计数器、通用目的寄存器和条件码
- 共享代码和数据，共享该进程的整个虚拟地址空间（相同的上下文、I/O连接……）

同进程：线程由内核自动调度，并且通过整数ID来识别线程

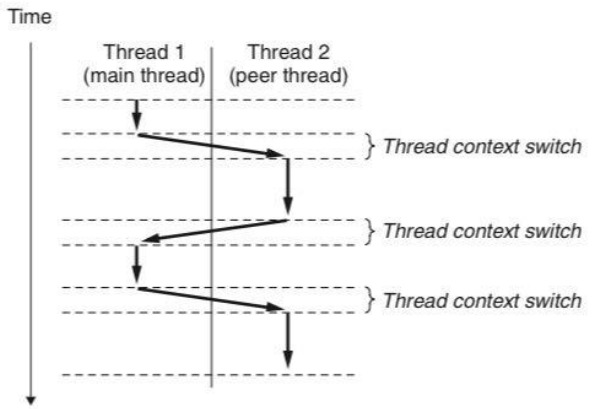
同I/O多路复用：多个线程运行在单一进程的上下文中

线程执行模型

- 主线程—创建—>对等线程

与进程的不同
①线程的上下文比进程的上下文小得多→切换快得多
②不按照严格的父子层次来组织，而是组成一个**对等池**，独立于其他线程创建的线程（主线程只是第一个运行的线程）
→一个线程可以杀死其他任何对等线程或等待它终止；每个对等线程都能读写相同的共享数据

Figure 12.12
Concurrent thread execution.



Threads vs. Processors

相似

- 都有自己的逻辑流
- 可以与其他进程同时发生
- 都由内核调度和切换

相异

- 线程共享除了本地堆栈以外的所有代码和数据
- 线程可以访问任何其他线程的堆栈
- 开销较小，与线程关联的上下文数量少于进程的上下文

Posix线程

C程序中处理器线程的一个标准接口

code/conc/hello.c

```
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.c

Figure 12.13 hello.c: **The Pthreads "Hello, world!" program.**

如果有多个输入和返回，可以将参数放到一个结构中

创建线程

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

Returns: 0 if OK, nonzero on error

线程例程f；改变默认属性attr参数

新线程可以用pthread_self函数来获得自己的线程ID

回收已终止线程的资源

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

Returns: 0 if OK, nonzero on error

函数会阻塞，直到线程tid终止，把返回的通用指针赋值到thread_return只想的位置，然后回收已终止线程占用的所有内存资源

终止线程

- 当顶层的线程历程返回时，线程会隐式地终止
- 通过调用pthread_exit函数，线程会显式地终止（主线程调用，等所有对等线程终止，返回thread_return）
- Exit函数终止进程以及所有相关线程
- 另一个对等线程通过pthread_cancel(pthread_t tid(当前线程的tid))终止当前线程

分离线程

- 可结合（joinable）或者可分离的（detached）

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, nonzero on error

- 一个可结合的线程能被其他线程杀死或者收回（不是放内存资源）
- 一个分离的线程是不能被其他线程回收或杀死的，在终止时系统自动释放其内存资源
- 用pthread_self()为参数的pthread_detach调用来分离自己
- 更多情况使用分离线程（e.g. Web服务器）

初始化线程

- pthread_once函数允许你初始化与线程例程相关的状态

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

Returns: 0 if OK, nonzero on error

当需要动态初始化多个线程共享的全局变量时，pthread_once函数是很有用的

基于线程的并发服务器

code/conc/echoservert.c

```
1   #include "csapp.h"
2
3   void echo(int connfd);
4   void *thread(void *vargp);
5
6   int main(int argc, char **argv)
7   {
8       int listenfd, *connfdp;
9       socklen_t clientlen;
10      struct sockaddr_storage clientaddr;
11      pthread_t tid;
12
13      if (argc != 2) {
14          fprintf(stderr, "usage: %s <port>\n", argv[0]);
15          exit(0);
16      }
17      listenfd = Open_listenfd(argv[1]);
18
19      while (1) {
20          clientlen=sizeof(struct sockaddr_storage);
21          connfdp = Malloc(sizeof(int));
22          *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
23          Pthread_create(&tid, NULL, thread, connfdp);
24      }
25  }
26
27  /* Thread routine */
28  void *thread(void *vargp)
29  {
30      int connfd = *((int *)vargp);
31      Pthread_detach(pthread_self());
32      Free(vargp);
33      echo(connfd);
34      Close(connfd);
35      return NULL;
36  }
```

code/conc/echoservert.c

为了避免竞争，将accept返回的每个已连接描述符分配到它自己的动态分配的内存块

如果是&connfdp 会在对等线程的赋值语句和主线程的accept语句间引入竞争。

线程例程中避免内存泄漏

释放

Close!!!

线程并发的优劣

- pros:

易于共享；相比于另外两个相当有效，

- cons:

但是困难存在unintended sharing（例如刚刚的竞争可能）

不能控制调度（unlike I/O）

12.4多线程中的共享变量

- 线程内存模型
- 将变量映射到内存
- 共享变量

内存模型

- 有独立的线程上下文（包括线程ID、栈、栈指针、程序计数器、条件码和通用目的寄存器值）每个线程和其他线程一起共享进程上下文的其他部分（包括整个用户虚拟内存空间，（只读文件、读写数据、堆以及所有的共享库代码和数据区域））相同的打开文件的集合

将变量映射到内存

- 全局变量：定义在函数之外的变量
- 本地自动变量：定义在函数内部但是没有static属性的变量
- 本地静态变量：本地静态变量是定义在函数内部并有static属性的变量
- 共享变量：一个实例被一个以上的线程引用

code/conc/sharing.c

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

code/conc/sharing.c

全局变量

本地静态变量　共享变量

变量实例	主线程引用的	对等线程0引用的	对等线程1引用的
ptr	T	T	T
cnt	F	T	T
i.m	T	F	F
msgs.m	T	T	T
myid.p0	F	T	F
myid.p1	F	F	T

注：v.t：　v是实例，t线程，m是主线程，p0对等线程0，p1对等线程1

Figure 12.15　**Example program that illustrates different aspects of sharing.**

Synchronization: Advanced (CS:APP Ch. 12.5-12.7)

朱家启

用信号量同步线程

code/conc/badcnt.c

```
1   /* WARNING: This code is buggy! */
2   #include "csapp.h"
3
4   void *thread(void *vargp);  /* Thread routine prototype */
5
6   /* Global shared variable */
7   volatile long cnt = 0; /* Counter */
8
9   int main(int argc, char **argv)
10  {
11      long niters;
12      pthread_t tid1, tid2;
13
14      /* Check input argument */
15      if (argc != 2) {
16          printf("usage: %s <niters>\n", argv[0]);
17          exit(0);
18      }
19      niters = atoi(argv[1]);
20
21      /* Create threads and wait for them to finish */
22      Pthread_create(&tid1, NULL, thread, &niters);
23      Pthread_create(&tid2, NULL, thread, &niters);
24      Pthread_join(tid1, NULL);
25      Pthread_join(tid2, NULL);
26
27      /* Check result */
28      if (cnt != (2 * niters))
29          printf("BOOM! cnt=%ld\n", cnt)
30      else
31          printf("OK cnt=%ld\n", cnt);
32      exit(0);
33  }
34
35  /* Thread routine */
36  void *thread(void *vargp)
37  {
38      long i, niters = *((long *)vargp);
39
40      for (i = 0; i < niters; i++)
41          cnt++;
42
43      return NULL;
44  }
```

```
linux>  ./badcnt 1000000
BOOM! cnt=1445085

linux>  ./badcnt 1000000
BOOM! cnt=1915220

linux>  ./badcnt 1000000
BOOM! cnt=1404746
```

- H i :循环头部的指令块
- L i :将共享的变量加载到每个线程独立的寄存器（如%rdx(i)）的指令
- U i:更新%rdx(i)的指令
- S i:将%rdx(i)存储回共享变量的指令
- T i:循环尾部的指令块

什么时候会出问题?

C code for thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx, %rcx
    jle    .L2
    movl   $0, %eax
-------------------------------
.L3:
    movq   cnt(%rip),%rdx
    addq   %eax
    movq   %eax,cnt(%rip)
-------------------------------
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

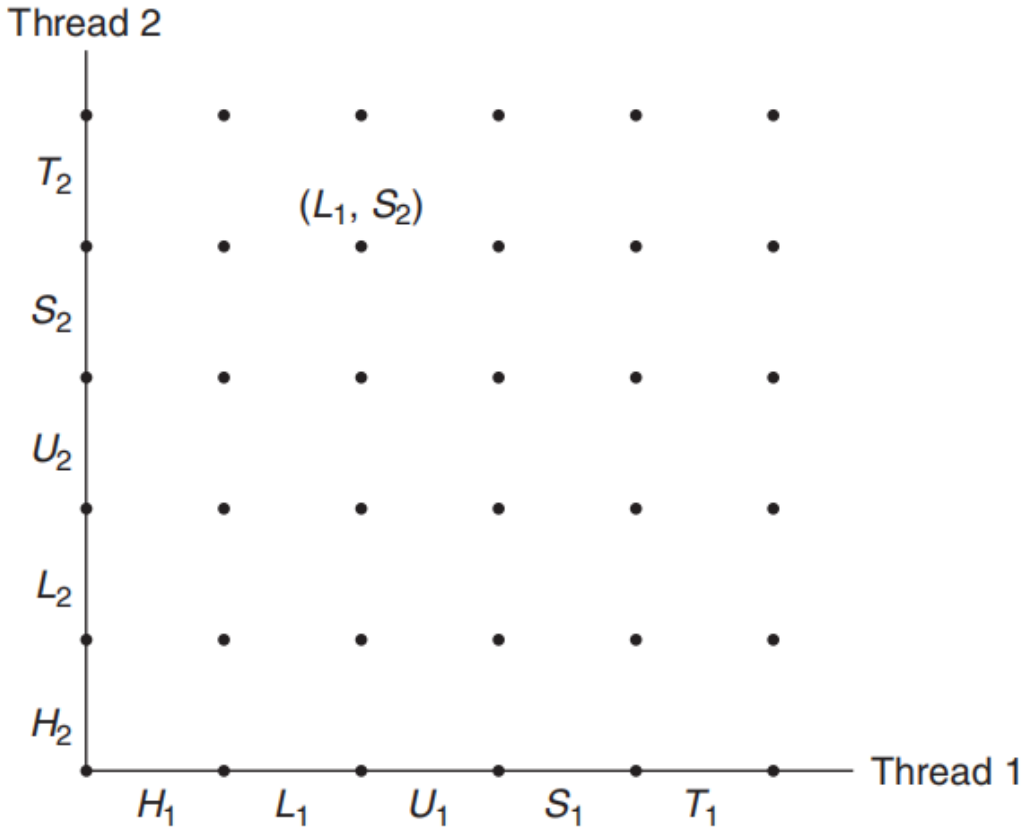
H_i: Head

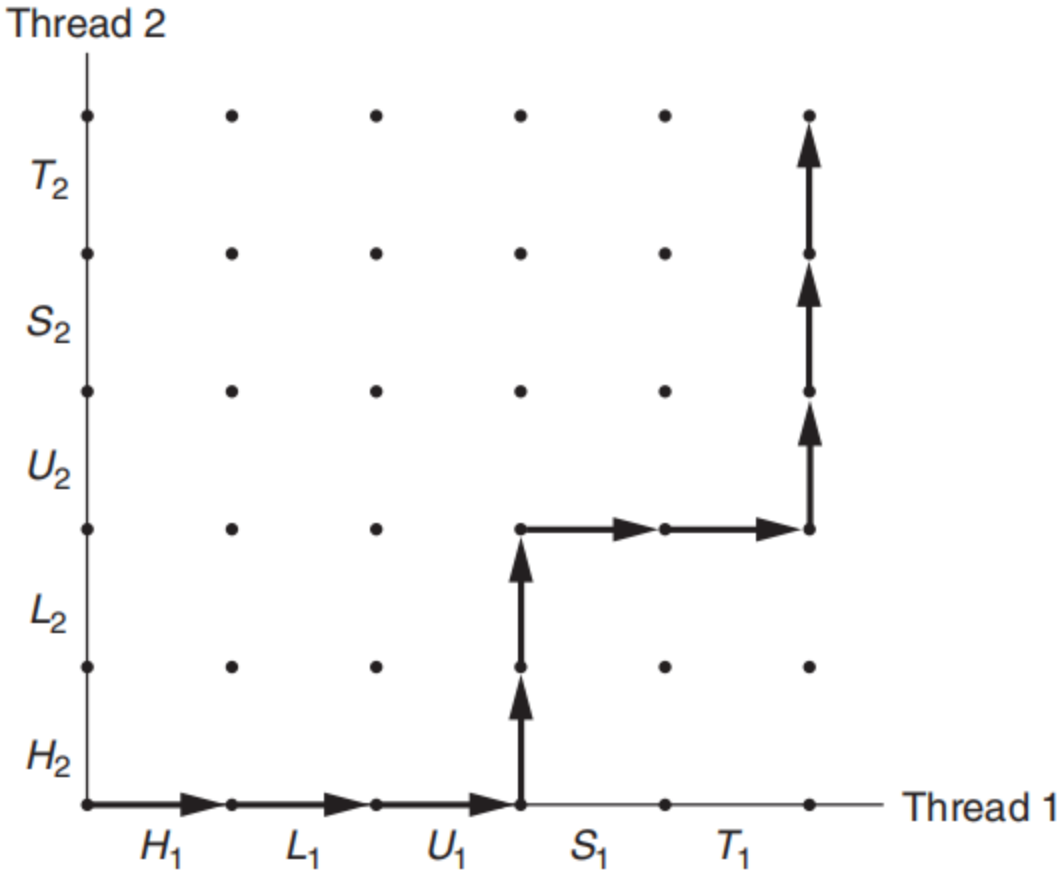
L_i: Load cnt
U_i: Update cnt
S_i: Store cnt

T_i: Tail

进度图

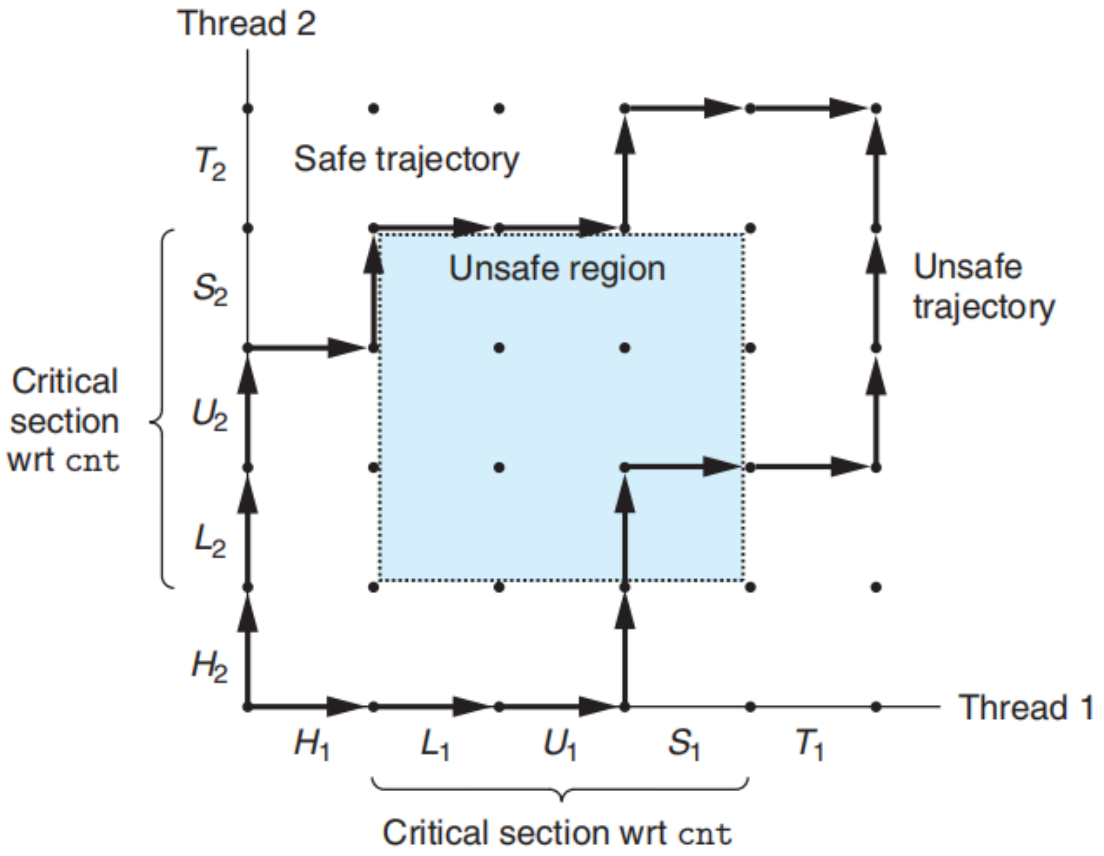
$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2$





- 临界区
- 不安全区
- 安全与不安全的轨迹线

如何刻画不安全区，使轨迹线能自动绕过不安全区？



Atomic Instruction

- Atomicity: 硬件保证在原子指令持续期间，其他指令不能进行读写操作
- 常见的原子指令：CAS(Compare and Swap) TAS(Test and Set)

```c
int cas(int* p, int old, int new){
    // atomic segment starts
    if (*p != old) return 0;
    *p = new;
    // atomic segment ends
    return 1;
}

void add(int* p, int a){ // *p+=a
    int done = 0;
    while (! done){
        int value = *p; //Doesn't need to be atomic
        done = cas(p, value, value + a);
    }
}
```

```c
#define LOCKED 1
int tas(int* lockptr){
    int oldValue;
    // atomic segment starts
    oldValue = *lockptr;
    *lockptr = LOCKED;
    // atomic segment ends
    return oldValue;
}

volatile int lock = 0;
void add(int* p, int a){
    while (tas(&lock) == 1);
    *p +=a; //only one process can be here
    lock = 0;
}
```

信号量

· 信号量 s 为具有非负整数值的全局变量
由两种原子指令 P 和 V 处理

$$P(s) = \begin{cases} s-1 \quad and \quad return, \quad if \quad s > 0 \\ hang \quad and \quad wait \quad for \quad V, \quad if \quad s == 0 \end{cases}$$

$$V(s) = s + 1, \quad \text{并随机重启一个被阻塞在P中等待s>0的线程}$$

- 信号量不变性：正确初始化了的信号量不会变成负值
- 同一时刻至多有1个原子指令(P/V)在对s进行读写!

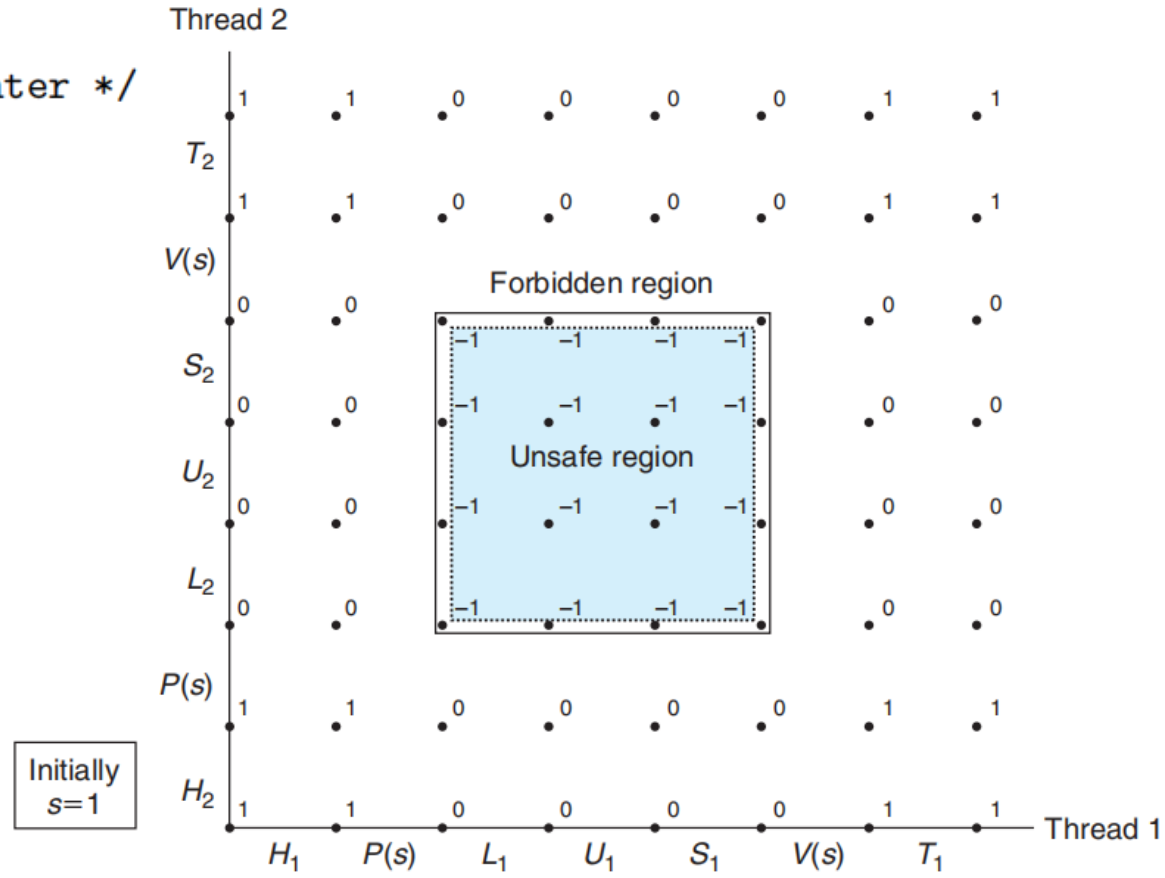
与不安全区的关系?

信号量实现互斥

```
volatile long cnt = 0; /* Counter */
sem_t mutex;           /* Semaphore that protects counter */


Sem_init(&mutex, 0, 1);  /* mutex = 1 */


for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```



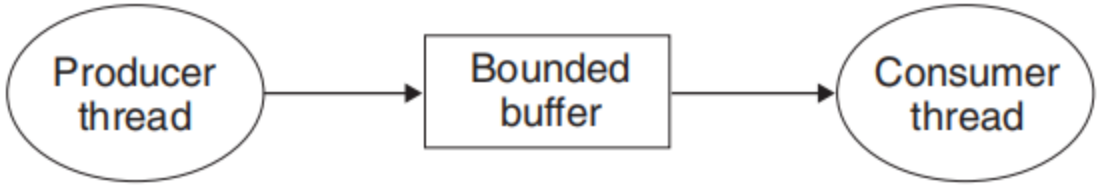
Binary Semaphore vs Counting Semaphore

- Binary Semaphore：信号量s只有两种可能状态：0或1
  - 同一时刻只有一个线程可以到达critical section（如 cnt++）

- Counting Semaphore：信号量s可能取大于等于0的任何整数
  - 同一时刻可能有多个线程到达critical section（如后面讲的生产者-消费者问题）

信号量调度资源

- 生产者-消费者问题



- 缓冲区有限，生产者产生项目放入缓冲区，消费者取出项目并消费
- 生产者之间竞争有限的空槽
- 消费者之间竞争有限的可用项目
- 生产者使用空槽，并提供可用项目
- 消费者使用可用项目，并提供空槽

生产者-消费者问题的分析

几个限制:

- （1）缓冲区在同一时刻只能被1个线程修改

  -> 信号量mutex锁缓冲区，初始化为1
- （2）缓冲区有限：大小为n，生产者不能随意生产，当缓冲区满时必须等待 -> 信号量slots锁空位数，初始化为n表示初始时有n个空位可用
- （3）可用项目要存在：消费者不能随意消费（不能在没有项目时虚空消费 -> 信号量items锁可用项目，初始化为0表示初始时没有可用项目

生产者-消费者问题的分析

- 在上述限制之外，我们要让程序各线程并行地，流畅地运行，需要我们合理地组织信号量的操作
- 生产者生产项目，要将其插入，由于生产者由（1）和（2），即mutex和slots限制，分析此时缓冲区的情况：
- 讨论优先mutex限制还是优先slots限制
  - 如果优先mutex限制，选择mutex不可用时就等待，可用的话就占用mutex（占用缓冲区），再等待slots可用
  - 如果优先slots限制，选择slots没空位时等待，有空位则占用一个空位，再等待mutex可用

生产者-消费者问题的分析

我们所要关注的是什么时候线程会停下等待，并且这种等待是不是一定会在有限时间内解除，以及会多久之后解除

若有可能进入某个状态，该状态永远无法解除等待：死锁

优先考虑mutex

slots \ mutex	可用	不可用
可用	占用缓冲区 消耗空位	等待缓冲区 不消耗空位
不可用	占用缓冲区 等待空位	等待缓冲区 不消耗空位

优先考虑slots

slots \ mutex	可用	不可用
可用	占用缓冲区 消耗空位	等待缓冲区 消耗空位
不可用	不占用缓冲区 等待空位	不占用缓冲区 等待空位

```
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;   /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}

int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];  /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

要分析各个信号顺序的选择

信号量调度资源

- 读者-写者问题

  - 一组并发的线程访问共享对象
  - 读者：只读对象
  - 写者：修改对象
  - 读者和写者之间竞争

  - 第一类读者-写者问题：读者优先，读者只在对象被某个写者占用时等待
  - 第二类读者-写者问题：写者优先，写者后到达的读者必须和写者一起等待

第一类：
R W W R R W
W W R W R

第二类：
R W W R R W
W W R W R

第一类读者-写者问题的分析

几个限制：

- （1）共享文件被写者写的时候，其他读者和写者都不能访问文件；共享文件被读者读的时候，其他写者不能写文件
  -> 文件是否能被写？由信号量w控制
- （2）文件在被读者读时，其他读者可以一起读，此时w被读者锁定（只要有人读就不能写），怎么判断什么时候解除w的锁定（都读完了）？
  -> 全局变量readcnt，所有线程共用，数有几个读者在读
- （3）全局变量readcnt需要被保护，防止多个读者同时修改产生L-U-S风险
  -> 信号量mutex保护readcnt，每次对readcnt进行调用时锁上控制权

第一类读者-写者问题的分析

- 对于读者：
  - 由（3），用mutex保护readcnt，由（2），每次根据readcnt判断有几个人在读，只要readcnt>=1，就保留对w的锁定
  - -> 当readcnt为1时，锁定w；readcnt为0时，解锁w
- 对于写者：
  - 由（1），w被锁定时等待，直到w被解除锁定，此时锁定w，进行修改共享文件，再解锁w

读者-写者问题的例子

```
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Critical section */
        /* Reading happens  */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

```
void writer(void)
{
    while (1) {
        P(&w);

        /* Critical section */
        /* Writing happens  */

        V(&w);
    }
}
```

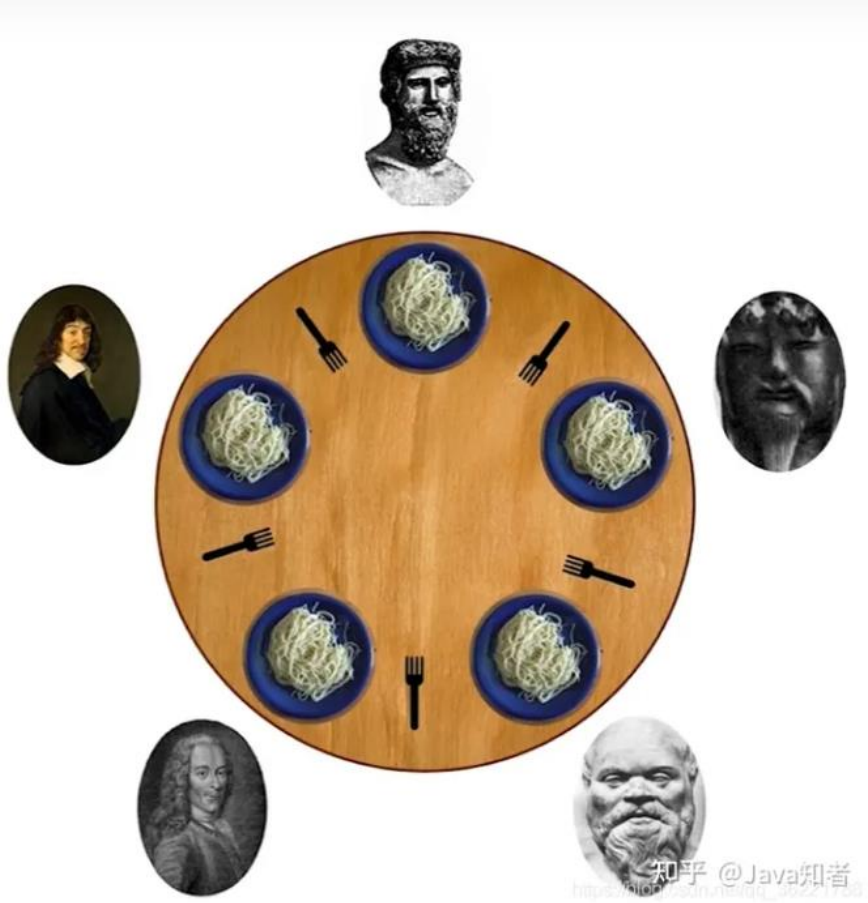
可能的问题?

哲学者用餐问题

- 死锁问题的特列

- 五个哲学家每天循环做两件事：思考，吃面
- 每人面前一个盘子，左右两边各一个叉子，面要两个叉子才能吃，每个人只能拿自己左右的叉子
- 吃的时候叉子被使用，吃完放回

- 问题：如何设计一个并发策略，使没有人挨饿；即所有人可以永远地在思考和吃面之间切换（没有人知道别人什么时候思考）。



哲学家用餐问题

- 难点：避免死锁

假设所有人都遵从以下策略:

- 1.思考，直到左手叉子可用，并在可用时拿起它
- 2.继续思考，直到右手叉子可用，并在可用时拿起它
- 3.两只手都拿叉子了，就吃饭（吃一段时间）
- 4.放下左手叉子
- 5.放下右手叉子
- 从头开始循环1-5

- 什么时候可能发生死锁?

哲学家用餐问题

- 第一种思路：
  - 如果每一个时刻都只有4个人要吃面（只有4个人去尝试拿叉子），那么5个叉子分给4个人，总有一个人手里有2个叉子，能开始吃面。
  - 用一个counting semaphore: room来控制房间里人数（能拿叉子的人数），初始化为4，之后再按照上面的逻辑来进行用餐。叉子本身用binary semaphore来控制使用权，5个不同的叉子信号量forks[5]。
    - 1.思考，直到能进入房间（room>0），否则等待
    - 2.等待左手叉子可用，并在可用时拿起它
    - 3.等待右手叉子可用，并在可用时拿起它
    - 4.两只手都拿叉子了，就吃饭（吃一段时间）
    - 5.出房间
    - 6.放下左手叉子
    - 7.放下右手叉子
    - 从头开始循环1-7

```c
sem_t room;
sem_t forks[5];

void * philosopher(void *);
void eat(int);
int main()
{
    int i,a[5];
    pthread_t tid[5];
    sem_init(&room,0,4);
    for(i=0;i<5;i++)
        sem_init(&forks[i],0,1);
    for(i=0;i<5;i++){
        a[i]=i;
        pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]
    }
    for(i=0;i<5;i++)
        pthread_join(tid[i],NULL);
}
```

```c
void * philosopher(void * num)
{
    int phil=*(int *)num;
    while (1){
        printf("\nPhilosopher %d stopped thinking",phil);
        sem_wait(&room);
        printf("\nPhilosopher %d has entered room",phil);
        sem_wait(&forks[phil]);
        sem_wait(&forks[(phil+1)%5]);

        eat(phil);
        sleep(2);

        printf("\nPhilosopher %d has finished eating",phil);
        sem_post(&room);
        sem_post(&forks[phil]);
        sem_post(&forks[(phil+1)%5]);
        printf("\nPhilosopher %d is thinking",phil);
        sleep(1);
    }
}
```

```c
void eat(int phil)
{
    printf("\nPhilosopher %d is eating",phil);
}
```

这种方法的问题?

哲学家用餐问题

- 第二种思路：
  - 问题在于拿叉子不是同时的，能不能等我能吃的时候我就拿起两个叉子吃，不能吃我就不拿叉子，这样让并行度更高（不占着叉子不吃）
  - 如何判断我能不能吃？类似于上一种思路，会有一种状态等在房间外面但是不进去拿叉子，这里也定义一种状态HUNGRY，表示该哲学家不再THINKING但是也还没有EATING的这种等待状态，存在全局变量state[5]。也就是说，当我从THINKING结束时，我要改变状态，并尝试拿叉子。调用函数 take_fork(id)处理这些事情。
  - 函数test(id)判断id能不能吃：如果我HUNGRY，而我两边的人都没EATING，那么我就可以吃，并把state变成EATING。注意这里要保护state，所以需要一个binary semaphore: mutex。
  - 每个哲学家手上要么2个叉子，要么没有叉子。哲学家怎么判断自己要不要拿起或放回叉子？
    - 上面第二条中，我从HUNGRY变成EATING开吃，拿起叉子
    - 吃完了放回叉子

哲学家用餐问题

- 第二种思路续
  - 结合上一页分析，当我HUNGRY时，实际上是在等旁边的EATING吃完，放下叉子，我再次判断自己能不能吃。所以每次EATING吃完时要向两边的人传递信息“我吃完了，叉子空出来了，你判断自己能不能吃”
  - 我如果能传递信息，说明我刚吃完，要放叉子提醒旁边人
  - 我如果不能传递信息，说明我HUNGRY，等待别人提醒我能吃，EATING完，再像别人传递信息 -> 即等待能放叉子提醒旁边人
  - 用binary semaphore S[5]控制我能不能放叉子提醒人，能放叉子就进入函数put_fork()改变自己状态为THINKING，提醒旁边人判断他自己能不能吃
  - 提醒旁边人调用函数test()，判断能不能吃

```
void* philosopher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}
```

```
void test(int phnum)
{
    //The array "state" was protected by caller
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n",
                        phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}
```

```
void take_fork(int phnum)
{
    sem_wait(&mutex);
    // state that hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}
```

```
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];
```

```
// put down chopsticks
void put_fork(int phnum)
{
    sem_wait(&mutex);
    // state that thinking
    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n",
        phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
```

Practice

朱家启

The End