

【ICS】 并发编程

特别感谢@[Musicminion - Overview](#)，本文档在他的笔记之上改造而成。

1. What 并发

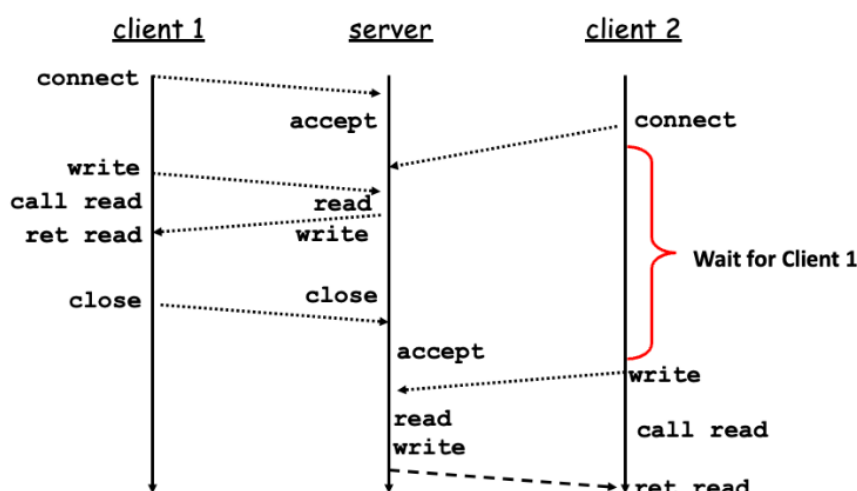
我们其实已经见过了一些并发的场景了：

- 硬件中断的handler
- shell

2. Why 并发编程

顺序执行的缺点：

- 以我们的Webserver为例子。我们之前实现的那个服务器要是遇到多个客户端在尝试链接的时候，第二个客户端就会在那里等第一个结束，会一直阻塞在那里。



- 所以我们需要并发，可以分化出很多执行流，每一个客户端来了之后都分化出执行流，然后就可以了。

应用程序级别的并发很有用：

- 响应异步事件：我明明在做一些事情，这时候一个 signal 来了，有人要打扰我，并发可以实现一些异步的事件的处理，在你做主线的时候，同时兼顾其他事情。等下面的 child 进程退出了之后我就去回收一下。
- 在多处理器上并行计算（充分利用计算资源）

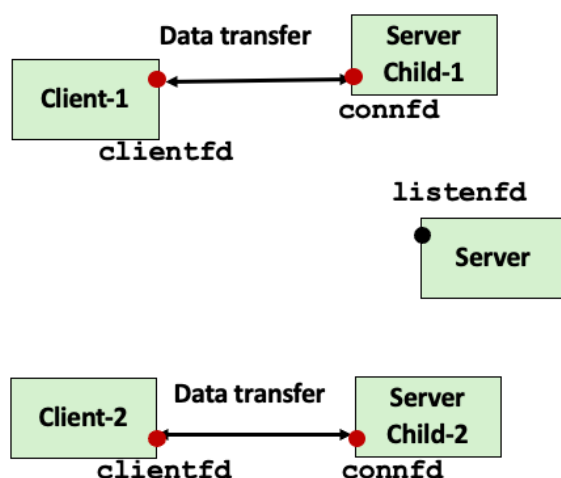
- 访问慢速I/O设备（如果只有一个执行流，就卡在那了，那我要是并发就能更快）
- 与人类交互（等用户反馈的时候，又可以做别的事情，用户输入可能要很久）
- 通过推迟工作减少延迟
- 为多个网络客户端提供服务

3. How 并发编程

- 并发编程：在应用程序层面的并发
- 基本的方法有：
 - Processes、Threads、I/O multiplexing

3.1 基于进程的并发编程

我们可以用Process来实现并发，当有一个客户端来了之后，我就创建一个child-fd，然后父进程马上可以等待下一个链接，这样就很快。

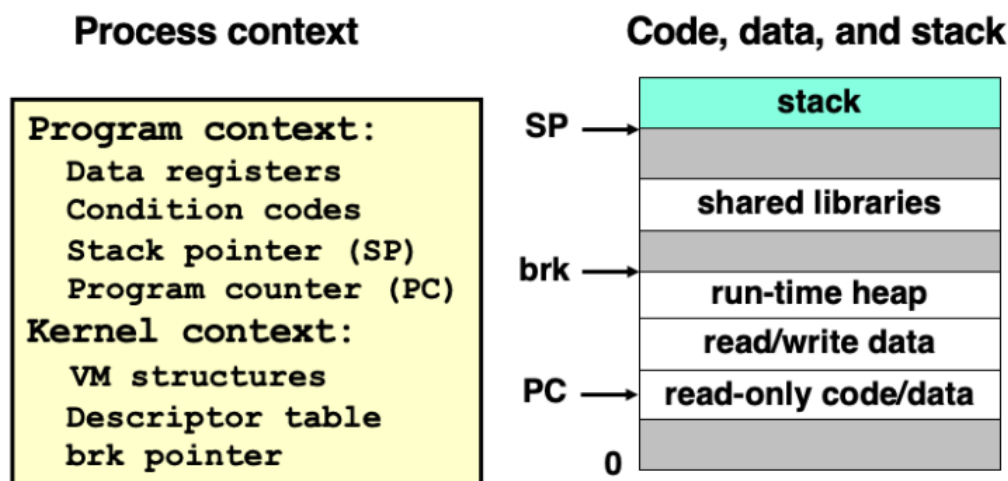


- 优点：逻辑清晰，可以提供多进程的服务
- 缺点：
 - 每个进程都有自己的空间，父子进程交互不了，只能通过signal通信，非常的麻烦。
 - 模型本身是比较清楚的，难点就在于你要去实现这些辅助的程序：handler、资源回收、解决共享困难。比如想要统计客户端传递了多少数据，很麻烦。
 - 最大的问题：开销大，每次都拷贝了内存，开销太大。

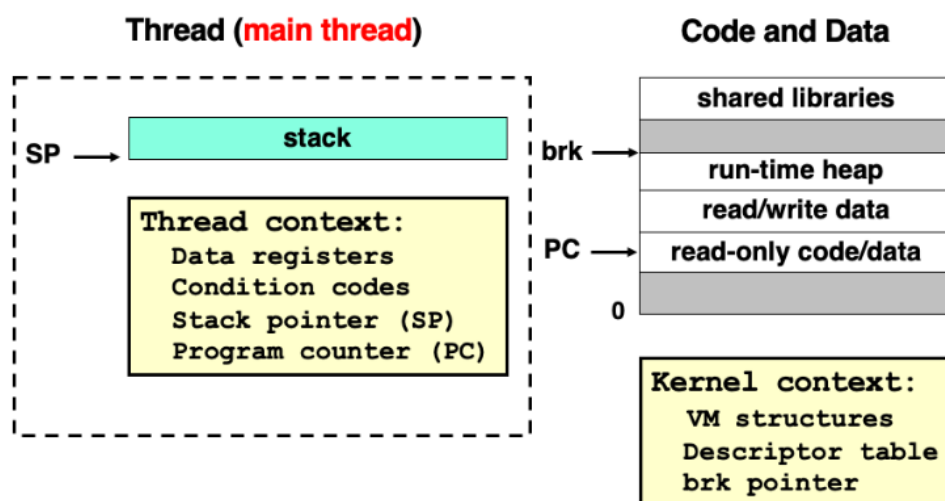
3.2 基于线程的并发编程

3.2.1 Ideas

- 进程的模型如下所示：



- 我们希望把进程做一个更细粒度的划分，让进程里面可以有多个控制流。怎么办？

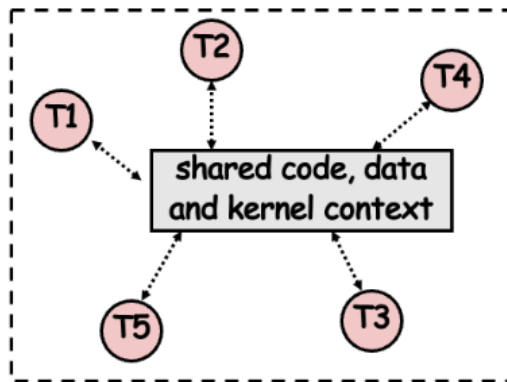


- 如上图所示，为什么需要stack？因为不同的执行流可能会调用不同的函数，栈帧都不一样，所以每一个子的控制流需要有自己的stack，此外子执行流还需要有自己的SP、PC、条件码、寄存器数据之类的。但是进程里面的Kernel Context可以分为VM Structure、描述符表等等的在创建多个执行流的时候不需要拷贝，不需要单独存一份自己的数据。
- 所以线程是进程一个轻量的抽象，包含自己的上下文和Stack。他们可以共享内存空间、程序的代码段。我们要实现并发就可以创建多个线程，他们共用一份代码段。
- 所以现在，每一个进程被创建的时候，都有一个main Thread，我们可以创建多个thread。他们都会共享一份全局变量，这样就方便了多个控制流的共享通讯。

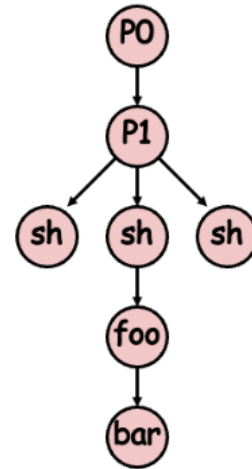
Thread和Process优点区别：两方面

- 一个方面是它的创建，只需要创建一个进程的一部分，所以它的 overhead 比较小。
- 第二个就是在这些 thread 之间共享数据，或者互相通信交流，就通过我写变量，因为他们全局变量都是共享的。

Threads associated with process foo



Process hierarchy



- 但是线程的缺点就是：隔离力度变小了，多个线程都能改全局变量，出现Bug的时候就更麻烦了。

3.2.2 Show Me the F**king Code

- 线程有两种，一种是detached线程，一种是joinable线程
 - joinable需要被其他的线程回收或者杀死，必须要依靠其他的线程
 - detached thread 会自动的释放空间
- 默认情况下，线程都是joinable类型的；为了防止内存泄漏，每个可结合线程要么被其他线程显式地收回，要么通过pthread_detach函数被分离。

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, nonzero on error

- 其实在现实世界中，有很多场景使用的都是分离的线程。比如高性能的web服务器，我们需要在每次接收到网络请求的时候都创建对等线程；对于服务器而言，每次都显式地等待每个对等线程终止就是一件非常抽象的事情，因此干脆都分离得了，自生自灭去。

```

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 }

```

1 // 特别注意这一段代码：必须要重新分配空间
2 // 之前我们的connfdp是个全局变量，现在必须要给每个线程开一个对应的connectFD
3 // 不然在并发的情况下，后面一个connectFD就把前面的覆盖了
4 connfdp = Malloc(sizeof(int));

```

1 /* thread routine */
2 void *thread(void *vargp)
3 {
4     int connfd = *((int *)vargp);
5     // detach 自己执行完自己退出，不需要mainThread
6     // 哪怕mainThread执行完成了，这个线程也可以继续执行！！
7     Pthread_detach(pthread_self());
8     // 记得释放空间
9     Free(vargp);
10    echo(connfd);
11    Close(connfd);
12    return NULL;

```

- 我们有 thread 概念了之后，操作系统是以线程为粒度进行调度的管理的，这样也就使得你的这个程序才能够真正的并行。每一个进程如果是一个顺序执行流，实际上是一个main线程。
- thread会被操作系统可能调度到多核心的CPU执行，所以不能对于多线程的情况做任何的假设。

3.3 基于I/O多路复用的并发编程

3.3.1 Idea

- 如果你是开饭店的，那么有很多客人，你想扩大经营怎么办？
 - **进程**：你在旁边又开了一家店，里面桌椅板凳、都需要复制一份，开销很大
 - **线程**：在原来的饭店里面放了更多的桌子
 - **IO多路复用**：你雇了一个非常nb的服务员。一个客人来了之后，他把他迎进来；然后隔壁桌的客人要点菜了，你就冲过去给人家点菜；然后第三个客人来了，他这边点菜完了，服务员就可以跑过去服务第三个人。

说实话这玩意我第一次学的时候没学懂。后面看了蒋炎岩的os的前几讲状态机的东西，稍微能够理解了一些：

- 每个客人就是一个状态机；它要点菜、埋单，就要各种状态转移；
- 我们还是只有一个服务员，服务员做的事情就是通过 `select` 检测每个人的状态，一旦说某个人准备好转移了，服务员就冲过去帮你做状态的转移。
- 所以，整个的逻辑就是 “事件驱动” 的 (event-driven)

 You

io多路复用和多线程有什么区别？

 ChatGPT

IO多路复用和多线程都是用于处理并发IO操作的技术，但它们有一些关键的区别：

1. 并发模型：

- **IO多路复用**：使用单个线程来监听多个IO事件，一旦有IO事件就绪，就会通知线程进行处理。因此，IO多路复用是一种单线程并发模型。
- **多线程**：使用多个线程来处理并发的IO操作，每个线程负责处理一个IO事件。因此，多线程是一种多线程并发模型。

2. 资源消耗：

- **IO多路复用**：由于使用单个线程来处理多个IO事件，因此线程的创建和销毁开销较小，相比于多线程模型，资源消耗更低。
- **多线程**：每个线程都需要占用一定的内存资源，并且线程的创建和销毁开销较大，尤其是在大规模并发的情况下，资源消耗相对较高。

3. 编程模型：

- **IO多路复用**：通常需要使用特定的API或库函数来实现，例如在Linux下可以使用select、poll、epoll等，或者在高级语言中使用相关的库函数来实现。
- **多线程**：通常使用线程库或语言提供的多线程机制来创建和管理线程，如Java的Thread类、C++的std::thread等。

4. 适用场景：

- **IO多路复用**：适用于IO密集型的场景，即大部分时间都是在等待IO操作完成，如网络服务器等。
- **多线程**：适用于CPU密集型的场景，即大部分时间都是在执行计算密集型的任务，如图像处理、数据计算等。

总的来说，IO多路复用适用于IO密集型的场景，可以降低系统的资源消耗；而多线程适用于CPU密集型的场景，可以充分利用多核CPU的性能优势。在实际应用中，可以根据具体的场景和需求选择合适的并发处理技术。

3.3.2 🌶️ Show Me the F**king Code

```
#include <sys/select.h>

int select (int maxfd, fd_set *readset,
            NULL, NULL, NULL);

Return nonzero count of ready descriptors, -1 on error
```

- **select()**

- Sleeps until one or more file descriptors in the set readset are ready for reading
- Returns the number of ready descriptors and sets each bit of **readset** to indicate the ready status of its corresponding descriptor

- `readset`
 - bit vector (max `FD_SETSIZE` bits) that indicates membership in a **descriptor set**
 - if bit `k` is 1, then descriptor `k` is a member of the descriptor set
- `maxfd`
 - Cardinality of the `readset`
 - tests descriptors `0, 1, 2, ..., maxfd-1` for set membership

```
/* clear all bits in fdset. */
void FD_ZERO(fd_set *fdset);
/* clear bit fd in fdset */
void FD_CLR(int fd, fd_set *fdset);
/* turn on bit fd in fdset */
void FD_SET(int fd, fd_set *fdset);
/* Is bit fd in fdset on? */
int FD_ISSET(int fd, *fdset);
```



You

io多路复用中的readset是什么



ChatGPT

在IO多路复用中，`readset`通常是一个用于表示可读文件描述符集合的数据结构。它是一种用于向操作系统指示应该监听哪些文件描述符的机制。当文件描述符中有数据可读时，操作系统会通知程序，从而使程序可以执行相应的读取操作。

在使用`select`、`poll`或`epoll`等IO多路复用的机制时，通常会创建一个`readset`，用于存储需要监听的文件描述符。程序将需要监听的文件描述符添加到`readset`中，然后将`readset`传递给IO多路复用的相关函数，如`select()`、`poll()`或`epoll_wait()`。这样，当其中任何一个文件描述符上有数据可读时，操作系统会通知程序，程序就可以进行相应的读取操作。

总的来说，`readset`是用于表示可读文件描述符集合的数据结构，用于在IO多路复用中指示应该监听哪些文件描述符。

The best way to understand `select` is to study a concrete example. Figure 12.6 shows how we might use `select` to implement an iterative echo server that also accepts user commands on the standard input. We begin by using the `open_listenfd` function from Figure 11.19 to open a listening descriptor (line 16), and then using `FD_ZERO` to create an empty read set (line 18):

	listenfd			stdin
	3	2	1	0
read_set (\emptyset):	0	0	0	0

Next, in lines 19 and 20, we define the read set to consist of descriptor 0 (standard input) and descriptor 3 (the listening descriptor), respectively:

	listenfd			stdin
	3	2	1	0
read_set ({0,3}):	1	0	0	1

At this point, we begin the typical server loop. **But instead of waiting for a connection request by calling the `accept` function, we call the `select` function, which blocks until either the listening descriptor or standard input is ready for reading (line 24).** For example, here is the value of `ready_set` that `select` would return if the user hit the enter key, thus causing the standard input descriptor to

become ready for reading:

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

Once `select` returns, we use the `FD_ISSET` macro to determine which descriptors are ready for reading. If standard input is ready (line 25), we call the `command` function, which reads, parses, and responds to the command before returning to the main routine. If the listening descriptor is ready (line 27), we call `accept` to get a connected descriptor and then call the `echo` function from Figure 11.22, which echoes each line from the client until the client closes its end of the connection.

While this program is a good example of using `select`, it still leaves something to be desired. The problem is that once it connects to a client, it continues echoing input lines until the client closes its end of the connection. Thus, if you type a command to standard input, you will not get a response until the server is finished with the client. A better approach would be to multiplex at a finer granularity, echoing (at most) one text line each time through the server loop.

```

1  #include "csapp.h"
2  void echo(int connfd);
3  void command(void);
4
5  int main(int argc, char **argv)
6  {
7      int listenfd, connfd;
8      socklen_t clientlen;
9      struct sockaddr_storage clientaddr;
10     fd_set read_set, ready_set;
11
12     if (argc != 2) {
13         fprintf(stderr, "usage: %s <port>\n", argv[0]);
14         exit(0);
15     }
16     listenfd = Open_listenfd(argv[1]);
17
18     FD_ZERO(&read_set);          /* Clear read set */
19     FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
20     FD_SET(listenfd, &read_set);    /* Add listenfd to read set */
21
22     while (1) {
23         ready_set = read_set;
24         Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25         if (FD_ISSET(STDIN_FILENO, &ready_set))
26             command(); /* Read command line from stdin */
27         if (FD_ISSET(listenfd, &ready_set)) {
28             clientlen = sizeof(struct sockaddr_storage);
29             connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
30             echo(connfd); /* Echo client input until EOF */
31             Close(connfd);
32         }
33     }
34 }
35
36 void command(void) {
37     char buf[MAXLINE];
38     if (!Fgets(buf, MAXLINE, stdin))
39         exit(0); /* EOF */
40     printf("%s", buf); /* Process the input command */
41 }

```

4. Synchronization 同步

4.1 并发bug杀人事件

共享变量：

- 取决于有没有多个线程都访问，无论是全局变量、Local变量、static变量，只要被多个线程访问（比如通过指针访问）就是共享变量

```

1  #include "csapp.h"
2  #define NITERS 1000000000
3  void *count(void *arg){
4      int i;
5      for (i=0; i<NITERS; i++)

```

```

6     cnt++;
7     return NULL;
8 }
9
10  /* shared variable */
11  unsigned int cnt = 0;
12
13
14  int main()
15  {
16      pthread_t tid1, tid2;
17
18      Pthread_create(&tid1, NULL, count, NULL);
19      Pthread_create(&tid2, NULL, count, NULL);
20      Pthread_join(tid1, NULL);
21      Pthread_join(tid2, NULL);
22      17
23      if (cnt != (unsigned)NITERS*2)
24          printf("BOOM! cnt=%d\n", cnt);
25      else
26          printf("OK cnt=%d\n", cnt);
27      exit(0);
28  }

```

- 这个例子是开了两个线程，每个线程执行一个for循环，把一个全局变量cnt++
- 最终的结果不是200000000，因为哪怕最简单的cnt++在汇编里面也是多个指令

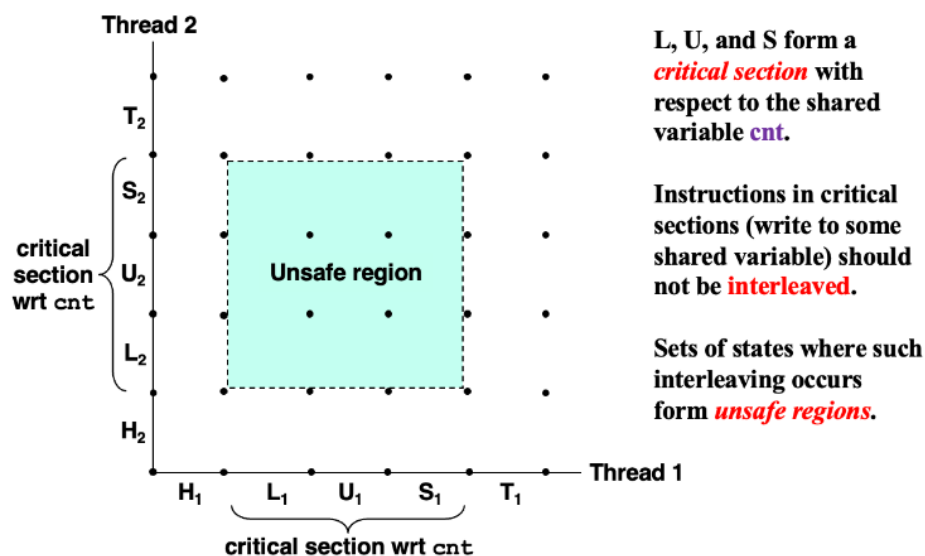
```

1  movq cnt(%rip),%rdx # Load
2  addq %rdx           # Update
3  movq %rdx,cnt(%rip) # Store

```

Asm code for thread i

Head (H_i)	{	<code>movq (%rdi), %rcx #i:-4(%ebp)</code>
		<code>testq %rcx, %rcx</code>
		<code>jle .L2</code>
		<code>movl \$0, %eax</code>
Load cnt (L_i)	{	<code>.L3:</code>
Update cnt (U_i)		<code>movq cnt(%rip), %rdx # Load</code>
Store cnt (S_i)		<code>addq %rdx, cnt(%rip) # Update</code>
		<code>movq %rdx, cnt(%rip) # Store</code>
	{	<code>addq \$1, %rax</code>
Tail (T_i)		<code>cmpq %rcx, %rax</code>
		<code>jne .L3</code>
		<code>.L2:</code>



4.2 信号量

- 所以我们希望，不要让两个人同时修改某个全局变量。我们可以用信号量来实现。

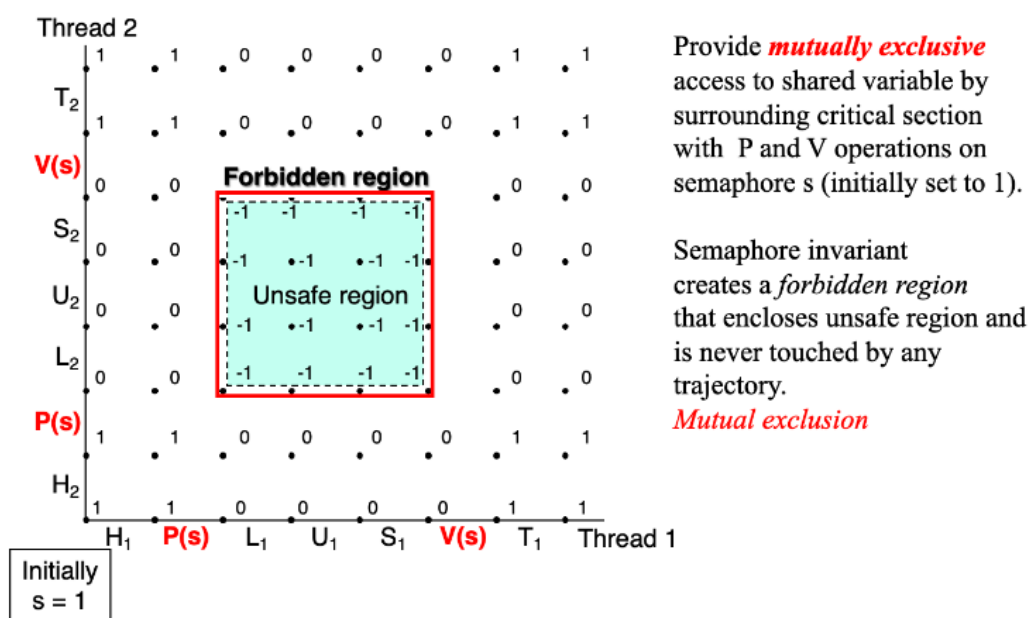
◦ P(s):

```
1 while (s == 0)
2     wait();
3 s--;
```

◦ V(s):

```
1 s++;
```

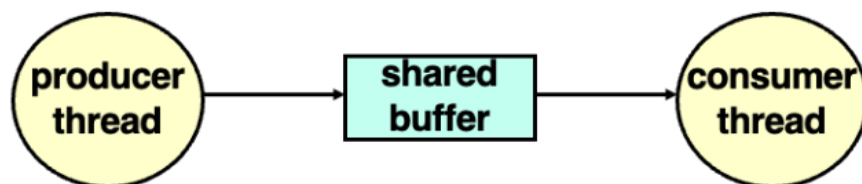
- s 就是信号量的大小，如果我们保证只能有一个线程访问，设置 s 值为1。一旦一个线程执行了P函数，后面的其他线程想要拿到信号量，就不可能！必须等待。只有当那个线程释放了锁，也就是说执行了V函数，其他的线程等待才会结束。



- 信号量有下面几种
 - **counting semaphores**: 比如系统有10个资源，有100个线程并发执行，每个线程只能拿一个资源，我们就可以使用counting semaphores，把刚刚说的 s 设置为10，当 s 被减小到0的时候，其他的线程就会等待；
 - **binary semaphores**: 就是类比上面的例子， $s=1$ ，只能有一个线程访问。
- 经典案例：生产者消费者问题、读者写者问题。

4.3 生产者消费者问题

- 问题概述：生产者消费者问题就是生产者往一个buffer生产东西，消费者从里面读取东西。Buffer里面大小是有限的。



- 假设Buffer里面可以放很多东西，producer往里面放东西，每个东西是一个slot单位。Producer只有当Buffer有空位的，才能往里面放东西，他的操作就是把 item 往里面放到这个 buffer 的位置里面，然后他需要去通知consumer。（并不一定要直接通知，可能是通过信号量的值改变，然后Consumer自己就停止等待了）
- 速度的协调：东西没有生产出来，消费者就不能拿到。
- 读写的协调：写的时候你可能写了一半，写的不完整，那可能被读到就会有问题。

TODO 看书吧

4.4 读写者问题

TODO 看书吧

5. Take away messages

- **Processes**
 - Hard to share resources: Easy to avoid unintended sharing
 - High overhead in adding/removing clients
- **Threads**
 - Easy to share resources: Perhaps too easy
 - Medium overhead
 - Not much control over scheduling policies
 - Difficult to debug: event orderings not repeatable
- **I/O Multiplexing**
 - Tedious and low level
 - Total control over scheduling
 - Very low overhead
 - Cannot create as fine grained a level of concurrency
 - Does not make use of multi-core