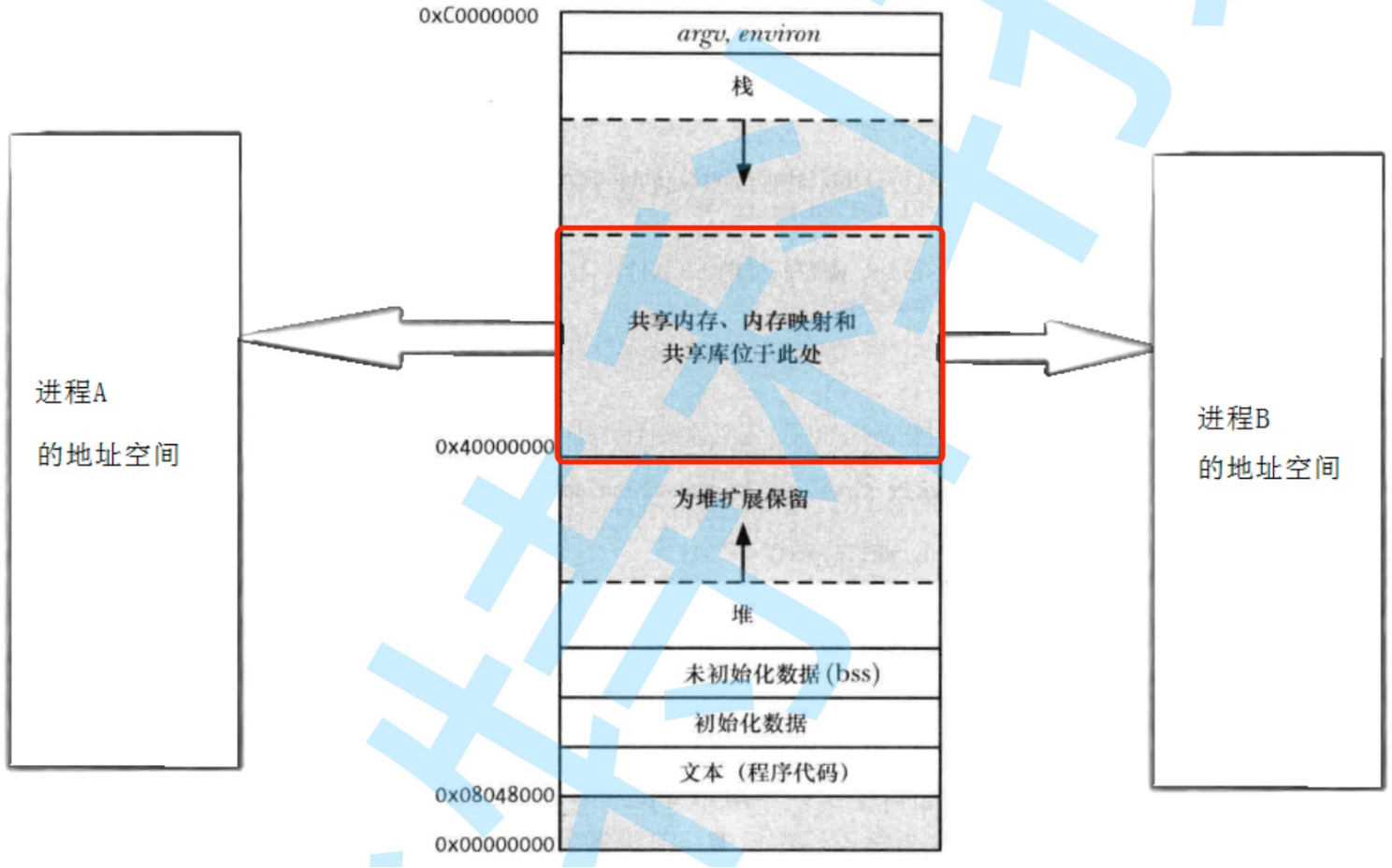
1201进程间通信

共享内存示意图



现在的一个问题：
这部分共享区域是属于用户空间呢还是属于内核空间呢？

答案是用户空间！

用户空间：
不用经过系统调用，可以直接访问！

双方如果要通信，直接进行内存级别的读和写就行了！

之前无论pipe，fifo都是要通过read，write通信
这是为什么呢？

因为文件是系统的东西

shmClient.cc | shmServer.cc ● | comm.hpp

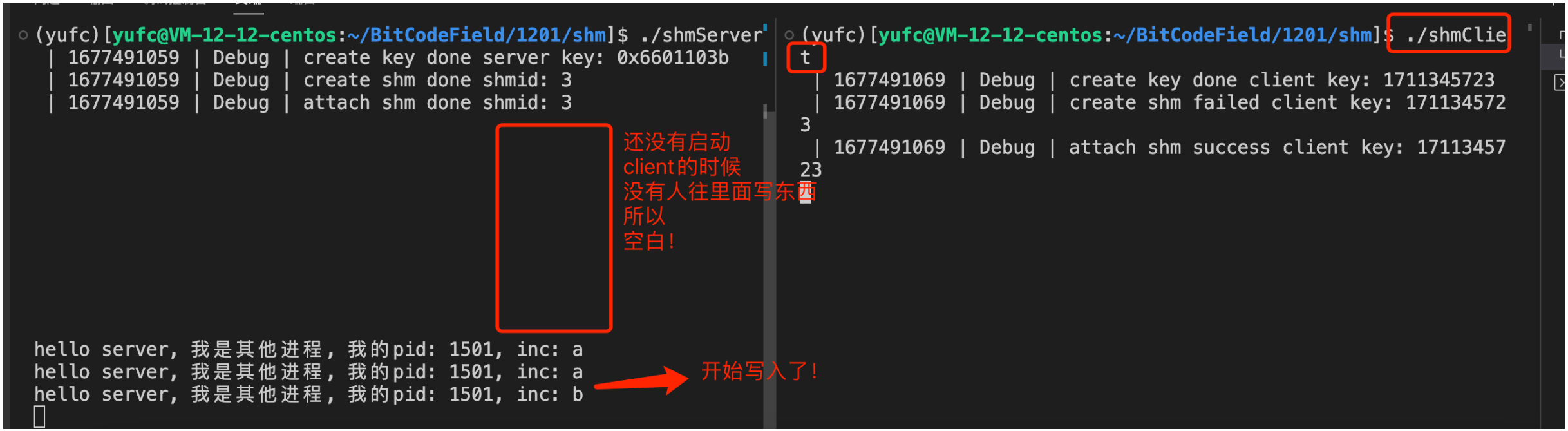
BitCodeField > 1201 > shm > shmServer.cc > main()

```
22      if (shmid == -1)
23      {
24          // 创建失败
25          perror("shmget");
26          exit(1);
27      }
28      Log("create shm done", Debug) << " shmid: " << shmid << std::endl;
29
30      //3.将制定的共享内存，挂接到自己的地址空间
31      // sleep(3);
32      char* shmaddr = (char*)shmat(shmid,nullptr,0);
33      Log("attach shm done", Debug) << " shmid: " << shmid << std::endl;
34      // sleep(3);
35      //我们会看到 挂接数会从0变成1!
36
37      // ================= 这里面就是通信的逻辑了! ================
38      //...
39      //将共享内存当做一个大字符串
40      for(;;)
41      {
42          printf("%s\n",shmaddr);
43          sleep(1);
44      }
45
46      // ================= 这里面就是通信的逻辑了! ================
47
```

shmClient.cc × | shmServer.cc ● | comm.hpp

BitCodeField > 1201 > shm > shmClient.cc > main()

```
17          Log("create shm success", Error) << " client key: " << k << std::endl;
18          exit(2);
19      }
20      Log("create shm failed", Debug) << " client key: " << k << std::endl;
21      // sleep(5);
22      //
23      char *shmaddr = (char *)shmat(shmid, nullptr, 0);
24      if (shmaddr == nullptr)
25      {
26          Log("attach shm failed", Error) << " client key: " << k << std::endl;
27          exit(3);
28      }
29      Log("attach shm success", Debug) << " client key: " << k << std::endl;
30      // sleep(5);
31
32      // 使用
33      // 将共享内存看作一个char类型的buffer
34      char a = 'a';
35      for (; a <= 'z'; a++)
36      {
37          //每一次都向shmaddr[共享内存的起始地址]写入
38          snprintf(shmaddr, SHM_SIZE - 1, "hello server, 我是其他进程, 我的pid: %d, inc: %c", getpid(), a);
39          sleep(2);
40      }
41
```



结论：

1. 只要通信双方使用shm，一方直接向共享内存中写入数据，另一方立刻就可以看到 --- 共享内存是所有进程间通信（ipc）中，速度最快的！不需要过多的拷贝！（不需要将数据交给操作系统）

```
// 使用
while (1)
{
    ssize_t s = read(0, shmaddr, SHM_SIZE - 1); // 直接从键盘读取，读取完直接写到共享内存里面
    if (s > 0)
    {
        shmaddr[s] = 0;
        if (strcmp(shmaddr, "quit") == 0)
            break;
    }
}
```

现在我们直接从键盘里面读

```
abcd
abcd
abcd
abcd
hello world
a
s
afsdfad
```

```
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$ ./shmClien
  | 1677637520 | Debug | child pid is: 8251
  | 1677637520 | Debug | create key done client key: 1711345723
  | 1677637520 | Debug | create shm failed client key: 1711345723
  | 1677637520 | Debug | attach shm success client key: 171134572
3
abcd
hello world
a
a
a
adsfa
s
afsdfad
```

但是，我们是输入quit为什么不退出？

因为有个回车

s-1 就行了，直接把\n设置成\0

```
// 使用
while (1)
{
    ssize_t s = read(0, shmaddr, SHM_SIZE - 1); // 直接从键盘读取,
    if (s > 0)
    {
        shmaddr[s - 1] = 0;
        if (strcmp(shmaddr, "quit") == 0)
            break;
    }
}
```

```
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$ make
g++ -o shmClient shmClient.cc -std=c++11
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$ ./shmServe
  | 1677637672 | Debug | create key done server key: 0x6601103b
  | 1677637672 | Debug | create shm done shmid: 6
  | 1677637672 | Debug | attach shm done shmid: 6
a
b
b
quit
  | 1677637681 | Debug | detach shm done shmid: 6
  | 1677637681 | Debug | delete shm done shmid: 6
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$
```

```
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$ ./shmClien
  | 1677637675 | Debug | child pid is: 9105
  | 1677637675 | Debug | create key done client key: 1711345723
  | 1677637675 | Debug | create shm failed client key: 1711345723
  | 1677637675 | Debug | attach shm success client key: 171134572
3
a
a
a
b
b
b
quit
  | 1677637680 | Debug | detach shm success client key: 171134572
3
(yufc)[yufc@VM-12-12-centos:~/BitCodeField/1201/shm]$
```

结论：

1. **只要通信双方使用shm，一方直接向共享内存中写入数据，另一方立刻就可以看到 --- 共享内存是所有进程间通信（ipc）中，速度最快的！不需要过多的拷贝！（不需要将数据交给操作系统）**

2. **共享内存缺乏访问控制！会带来并发问题。【如果我想进行一定程度的访问控制呢？能】**

怎么做呢？
我让你读你再读

```cpp
class Init
{
public:
    Init()
    {
        umask(0);
        int n = mkfifo(FIFO_NAME, 0666);
        assert(n == 0);
        (void)n;
        Log("create fifo success", Notice) << std::endl;
    }
    ~Init()
    {
        unlink(FIFO_NAME);
        Log("rm fifo success", Notice) << std::endl;
    }
};

Init init; // 对应的程序在加载的时候，会自动构建全局变量，就要调用构造函数
// 此时管道文件就会被自动创建
// 程序退出的时候，析构被调用，管道文件被删除
```

我这边唤醒你
你才出来读
否则别出来读取

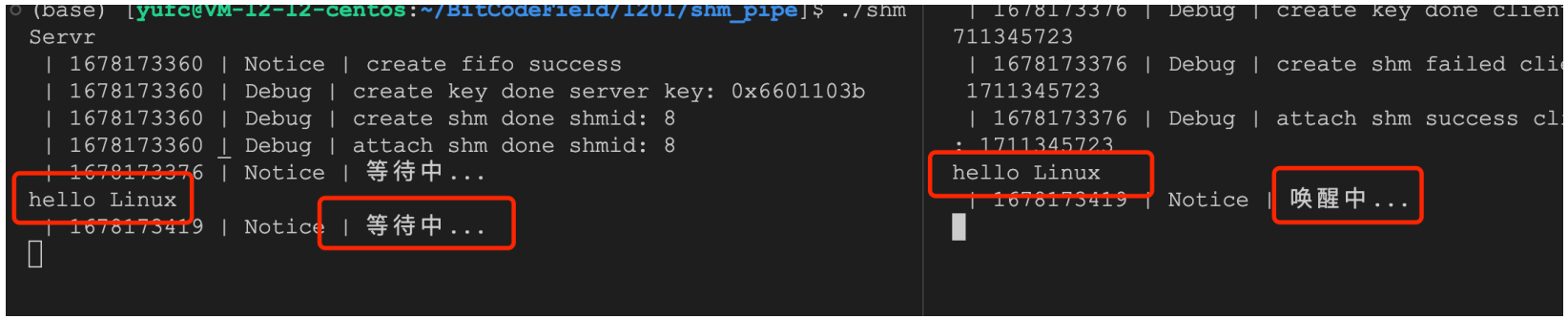
这样写会有一个小问题
如果放在全局
会被调用两次构造
第二次mkfifo就会失败
所以把init放在server里面就行了

```cpp
#define READ O_RDONLY
#define WRITE O_WRONLY
int OpenFIFO(std::string pathname, int flags)
{
    int fd = open(pathname.c_str(), flags);
    assert(fd > 0);
    return fd;
}

void Wait(int fd)
{
    Log("等待中...",Notice) << std::endl;
    uint32_t temp = 0;
    ssize_t s = read(fd, &temp, sizeof(uint32_t));
    assert(s == sizeof(uint32_t));
    (void)s;
}

void Signal(int fd)
{
    Log("唤醒中...",Notice) << std::endl;
    uint32_t temp = 1;
    ssize_t s = write(fd, &temp, sizeof(uint32_t));
    assert(s == sizeof(uint32_t));
    (void)s;
}

void CloseFIFO(int fd)
{
    close(fd);
}
```



这个就是，通过管道
完成两个进程之间通信的协同工作

system V消息队列 - 选学了解即可

- 消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法
- 每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型值
- 特性方面
    - IPC资源必须删除，否则不会自动清除，除非重启，所以system V IPC资源的生命周期随内核

system V信号量 - 选学了解即可

信号量主要用于同步和互斥的，下面先来看看什么是同步和互斥。

进程互斥

- 由于各进程要求共享资源，而且有些资源需要互斥使用，因此各进程间竞争使用这些资源，进程的这种关系为进程的互斥
- 系统中某些资源一次只允许一个进程使用，称这样的资源为临界资源或互斥资源。
- 在进程中涉及到互斥资源的程序段叫临界区
- 特性方面
    - IPC资源必须删除，否则不会自动清除，除非重启，所以system V IPC资源的生命周期随内核

基于对于共享内存的理解

为了让进程间通信 --- 让不同进程之间，看到同一份资源 --- 我们之前讲的所有通信方式，本质都是优先解决一个问题，让不同进程看到同一份资源

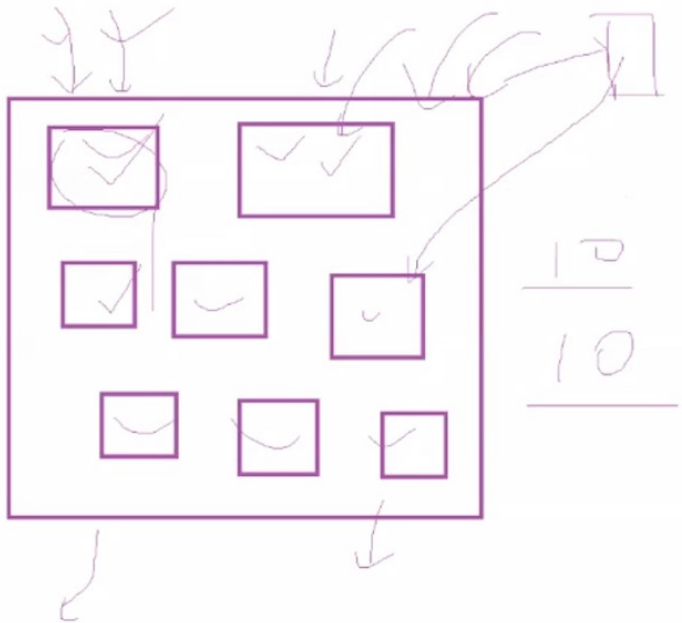
也带来了一些时序问题，造成数据不一致问题。

1. 我们把多个进程（执行流）看到的公共的一份资源 -- 临界资源
2. 我们把自己的进程，访问临界资源的代码 -- 临界区

   所以，多个执行流，互相运行的时候互相干扰，主要是我们不加保护的访问了同样的资源，在非临界区，多个执行流是互相不影响的！
3. 为了更好地进行临界区的保护，可以让多执行流在任何时刻，都只能有一个进程进入临界区，这种动作叫做**互斥**！！！

4、

看电影一定要有座位(放映厅里面的一个资源)， 这个座位真正属于你，是不是你自己坐在这个位置上，这个座位才属于你？不是！！
先买票，只要我买了票，我就拥有了这个座位。 买票的本质：对座位的 |预定| 机制



每一个进程想进入临界资源，访问临界资源中的一部分，不能让进程直接去使用临界资源(不能让用户直接去电影院内部占座位)， 你的先申请 信号量 (你的先买票！！)

本质是一个计数器，类似 int count = n; (不准确)

先申请信号量
1. 申请信号量的本质：让信号量计数器--
2. 主要申请信号量成功，临界资源内部，一定给你预留了你想要的资源 -- 申请信号量本质其实是对临界资源的一种 预定 机制

信号量是一个计数器， int n = 10；用一个整数，能不能标识信号量呢？

int n = 10;　　假设让多个进程(整数n在共享内存里)，看到同一个全局变量，大家都进行申请信号量 n --　　-- 也是不可以的！！

父进程

子进程

client:

n--

写入数据到共享内存

n++

server:

n--

读取共享内存

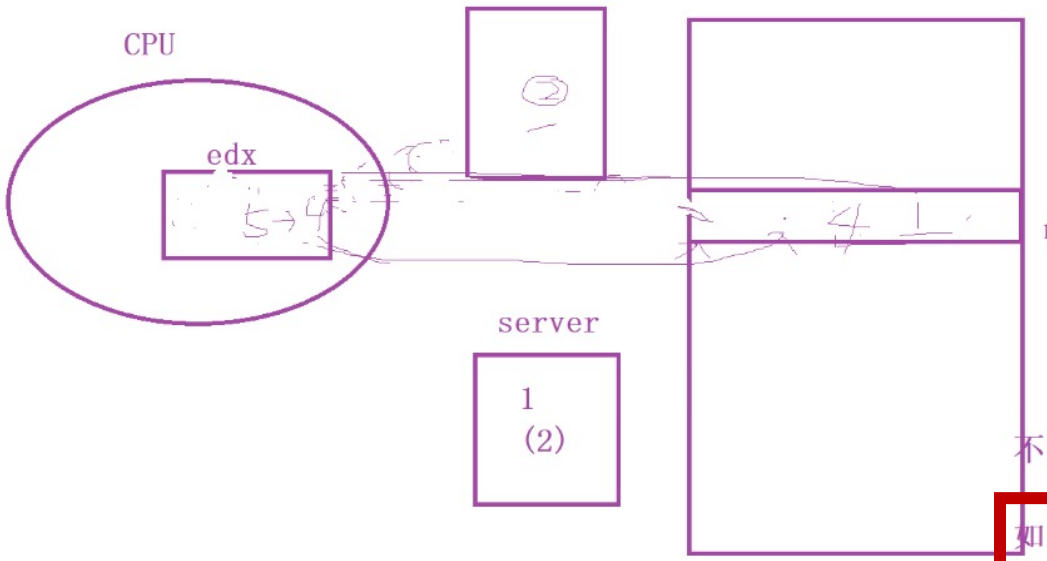
n++;

计算要在CPU内

数据在内存的n变量里面

CPU执行指令的时候

1. 将内存中的数据加载到cpu内的寄存器中(读指令)
2. n--(分析&&执行指令)
3. 将CPU修改完毕的n写回内存(写回结果)



n--：因为时序问题，而导致n有中间状态，可能导致数据不一致！！

不安全的

执行流在执行的时候，在任何时刻都可能被切换！！

寄存器只有一套，被所有的执行流共享，但是寄存器里面的数据，属于每一个执行流，属于该执行流的上下文数据！！

上下文保护 && 上下文恢复的！

如果一个n--操作只有一行汇编，该操作是原子的！！

信号量计数器　　是对临界资源的预定机制！！

申请信号量 -》 计数器-- -》 P操作 -> 必须是原子的

释放信号量 -》 计数器++ -》 V操作 -> 必须是原子

原子性：要么不做，要么做完，没有中间状态，就称之为原子性！！