未命名

> *操作系统：管理软硬件资源，为程序提供服务。*

操作系统的运行机制：

“指令”：CPU能识别、执行的最基本命令。（2进制机器指令）

CPU处于内核态和用户态最大的区别：能否执行特权指令。
内核态-->用户态：执行一条特权指令，改变标志位；
用户态-->内核态：由“中断”引发，硬件自动完成变态的过程。操作系统夺回CPU使用权。

中断与异常：

中断是操作系统内核夺回CPU控制权的方式。

内中断：中断信息来自CPU内部。（执行的指令是非法的或者参数是非法的、应用程序请求内核服务，执行陷入指令）
外中断：信息来自外部。（时钟中断、IO中断）

一般中断特指外中断，内中断成为异常。

系统调用：

应用程序通过系统调用来请求获得操作系统内核的服务。

库函数与系统调用的区别：

库函数是编程语言提供的，有时会将系统调用封装成库函数。
操作系统向上提供系统调用，使得上层程序能请求内核的服务。

涉及到内存存储的分配、IO操作、文件管理等与共享资源有关的操作，都需要使用系统调用来向操作系统发出请求。

操作系统的体系结构：

操作系统内部可以分为**内核功能**和**非内核功能**。
不同的操作系统的内核大小是不一样的，因此出现了所谓的宏内核、微内核等。

进程

程序：静态的。存放在磁盘里的可执行文件，就是一系列指令集合。
进程：动态的。是程序的一次执行过程。

进程的组成：

进程控制块PCB，当一个进程被创建后，操作系统会为其创建PCB，结束后会回收。

进程状态的转换：

5种状态及转变条件。

进程通信：（IPC）

进程是分配系统资源的单位（包括内存地址空间）。
三种通信方式：

1. 共享存储：
   一块多个进程都能访问的地址空间。（进程对这片区域的访问要互斥）
2. 消息传递：
   进程间的数据交换以格式化的消息为单位。
   将要发送的消息放到接收消息的进程的消息队列中。
3. 管道通信：
   一个管道：一端是一个进程写入数据，另一端是一个进程读取数据。
   一个管道是一个单向的。

信号：

实现进程间的通信。
用于通知进程某个特定事件已经发生。进程收到一个信号后，对该信号进行处理。

信号量： 实现进程间的同步、互斥。

线程：

线程是一个基本的CPU执行单元。
进程是除CPU之外的系统资源的分配单位。
每一个线程都有一个线程控制块TCB。

同一个进程的不同线程共享进程资源。

调度：

按照某种算法从就绪队列中选择一个进程为其分配CPU。

评价调度算法：

1. CPU利用率：
   CPU忙碌的时间/总时间

2. 吞吐量：（单位时间内完成作业的数量）
   完成多少任务/总时间
3. 周转时间：
   从任务被提交开始，到任务完成的时间间隔。
   任务完成时间-任务提交时间
4. 平均周转时间：
   各作业周转时间之和/作业数
5. 带权周转时间：
   作业周转时间/作业实际运行时间
6. 等待时间：
   一个进程被建立之后，等待CPU处理的时间之和。
7. 响应时间：
   从提出请求到首次产生响应的时间。

互斥与同步：

并发必然导致异步性，但我们有时需要进程发生的先后顺序，所以要解决异步问题，这就是进程同步。

临界资源：
一个时间段内只允许一个进程使用。
临界资源的访问必须是互斥的。

软件实现互斥：

自私的实现：

```
int enter=0;
//指令流S0
while(enter==1){}
enter=1;//这一步是自私的
……//访问临界资源的代码
enter=0;

//指令流S1
while(enter==1){}
enter=1;//这一步是自私的
……//访问临界资源的代码
enter=0;
```

最后会无法解决互斥，因为指令流执行时会随时发生上下文切换，不受指令流自身控制。
要让这种写法工作，**检查条件与设置标志必须是原子操作。**

改进方法：用谦让的方法。

谦让的实现：

```
int want[2] = {0};
//指令流S0
want[0] = 1;
while (want[1] == 1) {}//这一步是谦让的体现
……访问临界资源;
want[0] = 0;
//--指令流S1
want[1] = 1;
while (want[0] == 1) {}
……访问临界资源;
want[1] = 0;
```

但可能会出现死锁。

改进方法：不要互相等待。如果僵持，就放弃进入。

不僵持的实现：

```
int want[2]={0};
//指令流0
while(1){
    want[0]=1;
    if(want[1]==0)//看你想不想进
        break;//不进我就进了
    else
        want[0]=0;
}
……访问邻接资源
want[0]=0;
//指令流1
while (1)
{
    want[1] = 1;
    if (want[0] == 0)
        break;
    else
        want[1] = 0;
}
……访问临界资源;
want[1] = 0;
```

可能有活锁问题。

合作的实现：

```
int turn = 0;
//指令流S0
while (turn == 1) {}
访问临界资源;
turn = 1;
//指令流S1
while (turn == 0) {}
访问临界资源;
turn = 0;
```

解决了互斥问题，绝对公平，一个指令一旦使用过了临界区，必须等另一个指令流使用过临界区才能再一次访问临界区。
绝对的公平有什么问题?
如果两个指令流对临界区的使用频率先天就不同的话→饥饿

竞争的实现（合作与竞争结合）：Peterson算法

```
int want[2]={0};
int turn=0;
//指令流0
want[0]=1;
turn=1;
while(want[1]==1&&turn!=0){}
……访问临界资源
want[0]=0;
//指令流1
want[1] = 1;
turn = 0;
while (want[0] == 1 && turn != 1) {}
……访问临界资源;
want[1] = 0;
```

解决了所有的问题。
我想要（want），但我让你先走（turn），如果你不想要并且让我先走，我就走。

不会出现死锁，因为turn保证了有一方的等待条件总是遭到破坏。
不会出现活锁，因为双方都在等待时均不放弃自己的进入意图。
不会出现饥饿，因为turn只代表发生竞争时的优先进入权。

进程互斥：锁

互斥锁：一种用以控制临界区访问的互斥访问原语，分为加锁和解锁两个原子操作。进入临界区先加锁，退出临界区后解锁。

需要连续循环忙等的互斥锁称为自旋锁。
好自旋锁的三个标准：
忙则等待、空闲让进、有限等待。

阻塞锁与自旋锁的区别只有一个，就是当指令流无法获得锁时就停止执行，并等待锁的释放。
好阻塞锁的标准：
自旋锁上加一个“让权等待”。

阻塞锁与自旋锁相比，全都是优点吗？它的**缺点**（尤其是多核并发环境 下的线程级别的阻塞锁）是什么？
如果锁的并发度（指一齐竞争的线程数）很少，但并发争用行为很频繁、 临界区很短，系统调用以及线程阻塞、切换和唤醒的开销就不可忽视了。

信号量机制：

由一对原语实现进程的互斥、同步等。
wait（S），signal（S），可以简写为P，V操作。

```
int S = 1;  //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S) {    //wait 原语，相当于“进入区”
  while (S <= 0);      //如果资源数不够，就一直循环等待
  S=S-1;               //如果资源数够，则占用一个资源
}

void signal (int S) { //signal 原语，相当于“退出区”
  S=S+1;          //使用完资源后，在退出区释放资源
}
```

```
进程P0:
...
wait(S);           //进入区，申请资源
使用打印机资源...  //临界区，访问资源
signal(S);         //退出区，释放资源
...
```

将循环改为阻塞，就可以了。

PV实现进程同步：
在“前操作”之后执行V操作，在“后操作”之前执行P操作。

生产者/消费者问题：

有一个共同的缓冲区，各进程必须互斥的访问缓冲区。
两个同步关系：缓冲区没满-->生产者生产；缓冲区没空-->消费者消费
使用PV操作的实现。
实现互斥的P操作一定在实现同步的P操作之后。
多生产者/消费者问题：

读写者问题：

当第一个读进程读取文件时，上锁，最后一个读完后，解锁。
为防止多个读进程同时上锁，要在读进程之间进行一个互斥，防止同时上锁。

哲学家就餐问题：

条件变量 Condition Variable
一种同步原语，可以使指令流阻塞并等待，直到某个条件发生。它总是与一个锁配合使用：如果条件不满足，则指令流自动释放锁并加入等待队列；而当条件满足时，等待队列头部的指令流将被唤醒并自动恢复对锁的持有。
惊群效应 Thundering Herd
当多个指令流（尤指并发情况下的线程）的阻塞被同时解除，导致其同时被唤醒、系统负载瞬时极大升高的情况，可能导致系统短暂失去响应。俗称炸窝。

条件变量的Mesa语义：
条件变量的唤醒操作仅保证被唤醒的线程进入就绪状态，且当它们被调度时有机会参与锁的竞争，并不保证该线程立即得到调度。

用条件变量与互斥锁实现生产者/消费者问题的代码：

```
mutex_t queue = 0;
int length = 0;
condition_t touch = COND_INIT;
```

生产者	消费者
lock(&queue);	lock(&queue);

length++;
enqueue(item);
cond_signal(&touch, &queue);
	while (length == 0)
	cond_wait(&touch, &queue);
	length--;

unlock(&queue);	item = dequeue();
	
	unlock(&queue);

考虑缓冲区长度：

```
mutex_t queue = 0;
int length = 0;
condition_t touch = COND_INIT;
```

生产者	消费者
lock(&queue);	lock(&queue);

if (length == N)
cond_wait(&touch, &queue);
length++;
	if (length == 0)
	cond_wait(&touch, &queue);
	length--;

enqueue(item); cond_signal(&touch, &queue);	item = dequeue(); cond_signal(&touch, &queue);
unlock(&queue);	unlock(&queue);

看上去超简单，者地方加一个判断就可以了。

这个有问题，因为无法保证signal成功唤醒正确的。
所以我们可以使用cond_signal_all，但是会有惊群效应。

正确的做法是来使用空和满两个信号量。

```
mutex_t queue = 0;
int length = 0;
condition_t full = empty = COND_INIT;
```

生产者	消费者
lock(&queue);	lock(&queue);
while (length == N) cond_wait(&full, &queue);	while (length == 0) cond_wait(&empty, &queue);
length++;	length--;
enqueue(item);	item = dequeue();
cond_signal(&empty, &queue);	cond_signal(&full, &queue);
unlock(&queue);	unlock(&queue);

信号量：

将一个条件变量与一个计数器封装起来，就可以得到（计数）Semaphore信号量。它是一种比锁和条件变量都更强大的互斥/同步通用工具，同时具备资源计数和等待两种功能。
（把条件变量与和它配对的互斥锁封装起来就是PV操作）。

问题 对比互斥锁、条件变量和信号量。它们有什么相同点和不同点？
提示 从主要作用、使用方法、复杂程度、可替代性等角度思考。

项目	互斥锁	条件变量	信号量
主要作用	临界区互斥	基于任意条件的同步	基于资源数量的同步
使用方法	同一个指令流内成对的lock()和unlock()	与锁配对使用，但对使用场景无要求	生产者负责release()，消费者负责acquire()
复杂程度	低	中等	高
唤醒丢失	-	可能丢失	基于计数，不会丢失
可替代性	?	?	?

在大多数场合，信号量是可以替代条件变量的，因为大多数场景或多或少都是生产者-消费者场景。而且，信号量带计数，不会丢失唤醒，很多时候无需额外加锁，用起来比条件变量更方便。使用到全部唤醒（cond_signal_all）的场合，或者非标准生产者-消费者场景的场合，信号量不能代替条件变量。

吸烟者问题：
一个供应商不断提供材料，三者分别只缺一种。

```
mutex_t queue = 0;
semaphore_t tobacco = paper = match = more = 0;

A                          B                          C

sem_acquire(&tobacco);     sem_acquire(&paper);       sem_acquire(&match);

smoke();                   smoke();                   smoke();

sem_release(&more);        sem_release(&more);        sem_release(&more);

Coordinator（不允许修改）

switch(rand()%3) {
case 0: sem_release(&tobacco); break;
case 1: sem_release(&paper); break;
case 2: sem_release(&match); break; }

sem_acquire(&more);
```

原始的吸烟者问题：

一个供应商一次提供两种材料，三者分别缺两种。

```
mutex_t queue = 0;
semaphore_t tobacco = paper = match = more = 0;

A                                  B                                  C

sem_acquire(&tobacco);             sem_acquire(&paper);               sem_acquire(&match);
if(sem_try_acquire(&paper)) {      if(...) {                          if(...) {
    smoke();                           smoke();                           smoke();
    sem_release(&more);                sem_release(&more);                sem_release(&more);
}                                  }                                  }
else                               else                               else
    sem_release(&tobacco);             sem_release(&paper);               sem_release(&match);

Coordinator（不允许修改）

switch(rand()%3) {
case 0: sem_release(&tobacco); sem_release(&paper); break;
case 1: sem_release(&paper); sem_release(&match); break;
case 2: sem_release(&match); sem_release(&tobacco); break; }

sem_acquire(&more);
```

添加使用try的acquire可以避免死锁，但是会有活锁。

另一种解法:

利用信号量完成吸烟者问题（增加难度）

```
mutex_t queue = 0;
semaphore_t tobacco = paper = match = A = B = C = more = 0;
```

A	B	C
sem_acquire(&A);	sem_acquire(&C);	sem_acquire(&C);
sem_acquire(&tobacco);	sem_acquire(&paper);	sem_acquire(&match);
sem_acquire(&paper);	sem_acquire(&match);	sem_acquire(&tobacco);
smoke();	smoke();	smoke();
sem_release(&more);	sem_release(&more);	sem_release(&more);

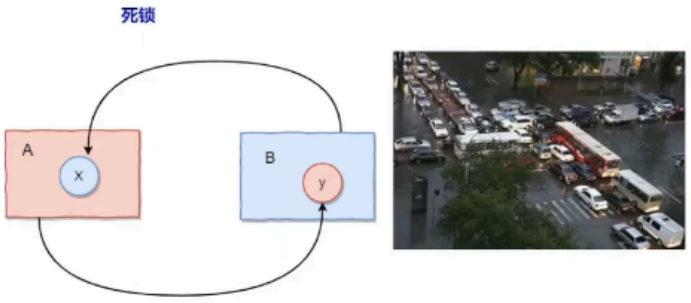
Coordinator（不允许修改）

```
switch(rand()%3) {
case 0: sem_release(&tobacco); sem_release(&paper); sem_release(&A); break;
case 1: sem_release(&paper); sem_release(&match); sem_release(&B); break;
case 2: sem_release(&match); sem_release(&tobacco); sem_release(&C); break; }

sem_acquire(&more);
```

死锁

- 定义：多个进程因竞争资源而造成的一种僵局，如果没有外力，这些进程将无法推进
- 产生的原因：非剥夺资源的竞争和进程的不恰当推进顺序
- 解决方法(一定都要记住，尤其是预防死锁)
  - (1)预防死锁：
    - 破坏互斥条件
    - 破坏不剥夺条件
    - 破坏请求和保持条件
    - 破坏循环等待条件
  - (2)避免死锁：安全状态、银行家算法
  - (3)检测死锁：利用死锁定理
  - (4)解除死锁：资源剥夺法撤销进程法 进程回退法



至少两个任务中的每一个都等待另一个任务持有的锁的情况

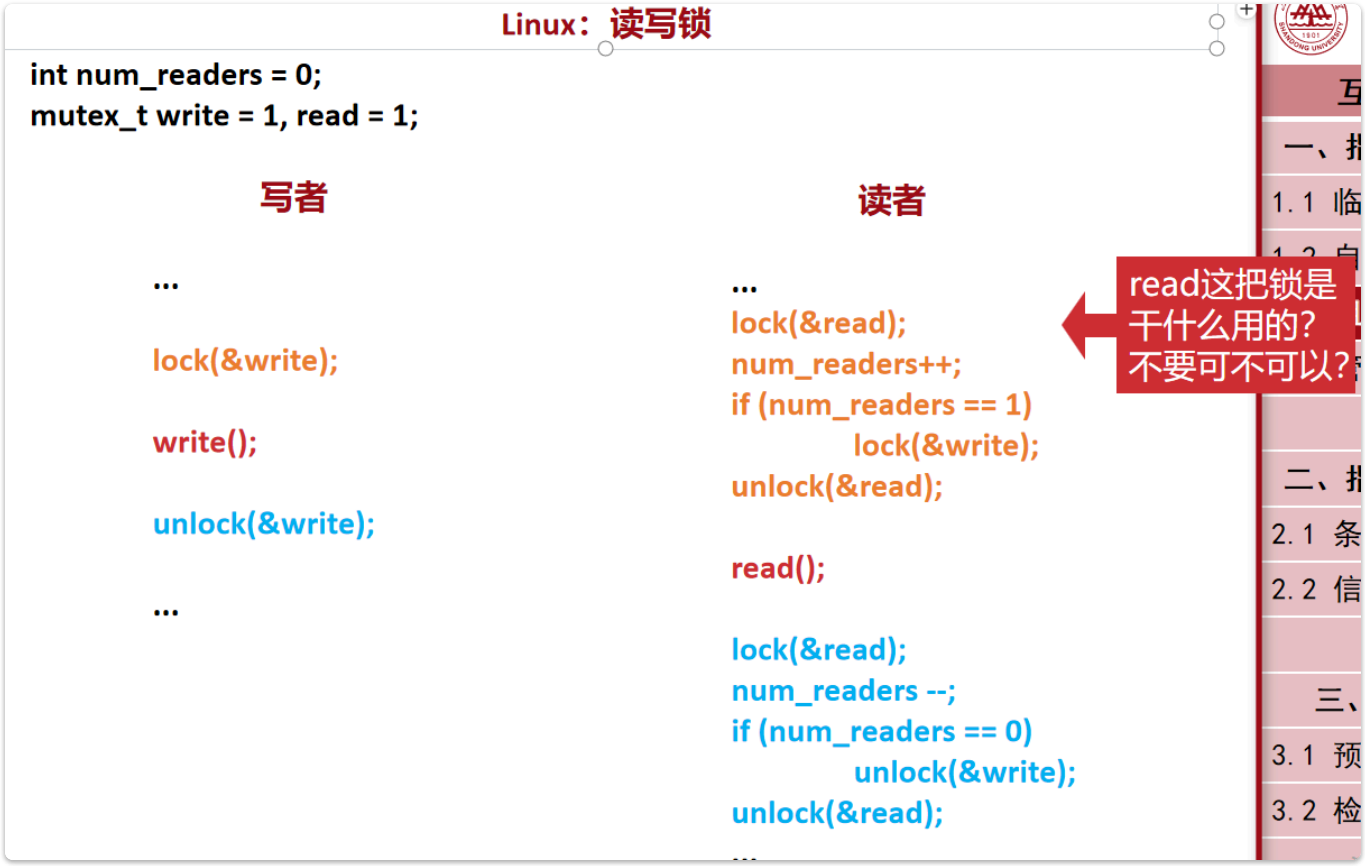
互斥条件、持有条件（保持请求、无法剥夺）、循环等待。

活锁：指令流并未死锁，但其在多次反复尝试获取资源均失败，无法进展或进展缓慢。

资源分配图可以检查死锁是否发生。

死锁的避免：
避免进入不安全状态、银行家算法。
银行家算法的开销很大，因为每次进行资源分配都需要对所有的运行该算法。
银行家算法对分配请求是在线算法，但对最大资源用量却是离线算法。

读写锁：



read是为了安全的修改num_readers的值。