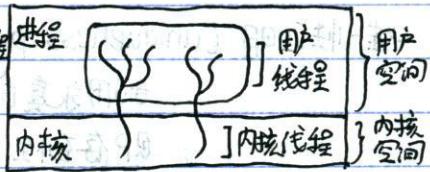


# Operating System - P18

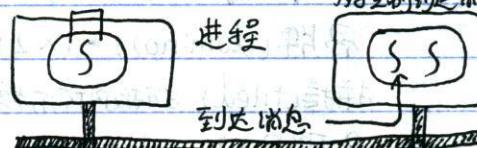
**混合实现** 将用户线程与某些或全部内核线程多路复用，多用户线程对应一个内核线程  
 内核只识别内核级线程，并对其进行调度  
 每个内核级线程有一个可轮流使用的用户级线程集合



**调度程序激活 (Scheduler activation)**: 目的是模拟内核线程的功能，但为线程包提供用户空间的性能和灵活性  
 如果用户线程从事某种系统调用时是安全的，就不应该使用非阻塞调用或是前挂起  
 如果一个线程阻塞在系统调用或页面故障，若同一进程有就绪进程，就应该运行其他线程  
**虚拟处理器**: 调度程序激活机制中，内核给每个进程安排一定数量的虚拟处理器，运行时系统分配线程  
 在多处理系统中，也可以是真实的CPU

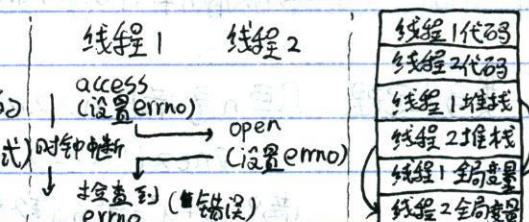
当内核了解到一个线程被阻塞，通知进程运行时系统，在堆栈中以参数形式传递线程  
 编号和事件描述，然后在一个已知地址启动运行的系统（算是对UNIX信号量的模拟）  
 运行时系统将线程标记为阻塞，并从就绪表中取出另一个线程并设置寄存器  
**上行调用 (upcall)**: 调用程序激活机制的一个目标是作为上行调用的信赖基础  
 这是一种违反分层次系统内在结构的概念

**弹出式线程** 传统方法处理到来消息：将进程或线程阻塞在 receive 调用上  
 现存线程  
 消息到达时 | 调用并检查  
 弹消息到达导致系统创建一个处理线程（需要提前计划）  
 可以快速创建这类线程，因为没有历史（寄存器或堆栈）



**多线程化的问题**：对线程而言是全局变量，并不是对整个程序也是全局的

**全局变量**：方案1. 全面禁止全局变量，可能与现有程序冲突，(Haskell采用这种形式)  
 方案2. 为每个线程赋予私有的全局变量



为全局变量分配一块内存，将它转送给线程中的每個进程作为额外参数

**不可重入的**：对于任何给定的过程，当前面的过程尚未结束之前，不可以进行二次调用

如发送消息时线程1编辑到缓冲区后遇到时钟中断而使得线程2重写缓冲区，

或UNIX中的malloc维护内存使用向关键表指针，在短暂的不一致状态时发生线程切换，从而指针无效

方案1. 重写整个库，但可能导致更多错误 方案2. 为每个过程提供一个包装器，但会降低系统潜在并行性

**信号**：有些信号在逻辑上不是线程专用，如果在用户空间实现线程，则内核由于不知道线程而无法正确地发送信号

如硬盘中断，由于不是线程专用，则应考虑由指定线程，所有线程或弹出式线程捕捉中断

如某个线程修改信号处理程序而未通知，或不同线程对于特定信号的不同处理

**堆栈**：由于内核不了解线程堆栈而不能使堆栈如进程般自动增长，造成堆栈出错（溢出）

**结论**：给已有系统引入线程而不进行实质性的重新设计系统是不可行的，至少重写库以重新定义系统调用的意义

# Operating System - P19

进程间通信 (Inter Process Communication), 目的是使用结构良好的方式而不使用中断

IPC

一. 进程如何把信息传递给另一个

线程容易解决, 因为同一进程的线程共享地址空间

二. 确保进程在关键活动中不会出现交叉

需要梳理清楚并保持恰当的顺序同样适用于线程

三. 如果顺序有关联, 确保顺序正确

且同样问题可以用同样方法

在一些OS中, 协作的进程可能共享一些彼此可读写的公用存储区, 考虑一个假脱机打印程序

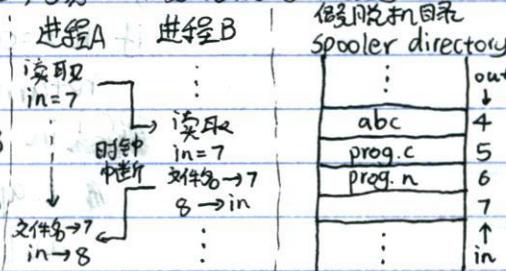
进程需要打印文件时, 文件名放入假脱机目录 (spooler directory)

打印机守护进程周期性地检查并打印, 打印后 out 指针 +1

A 读取 in (指向下一个空闲槽位) 为 7 后遇到时钟中断, 切换到进程 B

B 读取 in 值为 7, 将 B 文件名存入 7 后, 将 in 置为 8, 随即切换到 A

A 将文件名存入 7 后, 将 in 置为 8.



由于假脱机目录内部是一致的, 打印机守护进程不会发现错误, 但 B 文件名丢失了

Murphy 法则: 任何可能出错的地方终将出错

竞争条件

(race condition), 多个进程读写一些共享数据后, 而最后结果取决于进程运行的精确顺序

实际上凡是涉及共享内存, 共享文件以及共享任何资源的情况都可能产生类似错误 (CPU 流水线设计)

关键是需要某种途径阻止多个进程同时读写数据

互斥

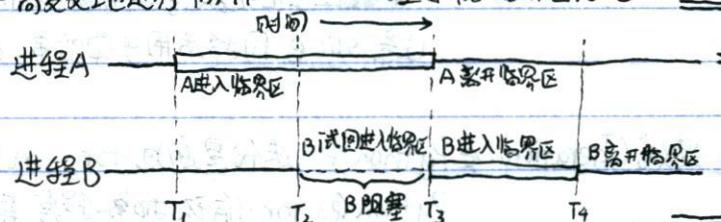
(mutual exclusion), 即确保当一个进程在使用一个共享变量或文件时, 其他进程不能进行同样操作

临界区域 (critical region) 或称临界区 (critical section), 即对共享内存进行访问的程序片段

可以适当安排使得两个进程无法同时处于临界区中, 能够避免竞争条件

但仍不能保证共享数据的并发进程能正确和高效地进行协作

1. 任何两个进程不能同时处于其临界区



2. 不应对 CPU 的速度和数量做任何假设

3. 临界区外运行的进程不得阻塞其他进程

4. 不得使进程无限等待进入临界区

互斥实现

在单处理器系统中, 最简单的办法是进程进入临界区后立即屏蔽所有中断 (disable 指令)

屏蔽中断

由于时钟中断也被屏蔽, 所以期间 CPU 不会切换到其他进程, 即可以检查和修改内存而不用担心其他进程

一方面把屏蔽中断的权力交给用户进程并不明智, 且 disable 指令仅对执行 CPU 生效

另一方面对内核来说, 在其更新变量或列表的几条指令期间将中断屏蔽是很方便的

所以屏蔽中断对 OS 本身是很有用的技术, 但对于用户进程并不是合适的通用互斥机制

# Operating

## System - P20

### 锁变量

一种软件解决方案，假定存在一个共享（锁）变量，初始值为0，用1表示存在进程进入临界区

当进程试图进入临界区时，首先测试锁变量，为0，则进入临界区并将锁变量值置为1

若锁变量值为1，则进程等待直到锁变量值为0

这种方法存在与假脱机目录相同的问题，即在读取共享变量值命令与根据共享变量值进行其他运算之间，永远存在一个可中断的位置，从而造成竞争条件（即在中断处为另一进程的读取与运算）

### 严格轮换

定义一个共享变量用于记录轮到哪个进程进入临界区，并检查或更新共享内存。考虑两个进程情形：

进程会在一个无限循环中不断测试turn值，

while (TRUE) { 进程0 } while (TRUE) {

直到某个特定值（如PID）出现，然后进入临界区

while (turn != 0); ← while (turn != 1);

退出临界区时将turn值置为下一个特定值（轮换中PID） critical\_region();

critical\_region();

忙等待（busy waiting）连续测试变量直到特定值出现（浪费CPU时间） noncritical\_region();

noncritical\_region();

自旋锁（spin lock），用于忙等待的锁

turn = 1; } turn = 0; }

问题：如果两个进程运行速度差异较大，则较快的进程可能存在必须等待较慢的进程  
进入一次临界区才能进入的情形，即此进程被一个临界区外的进程阻塞了（违反规则3）

严格地轮流进入临界区虽然可以避免竞争条件，但不是一个好的备选方案（因为条件3）

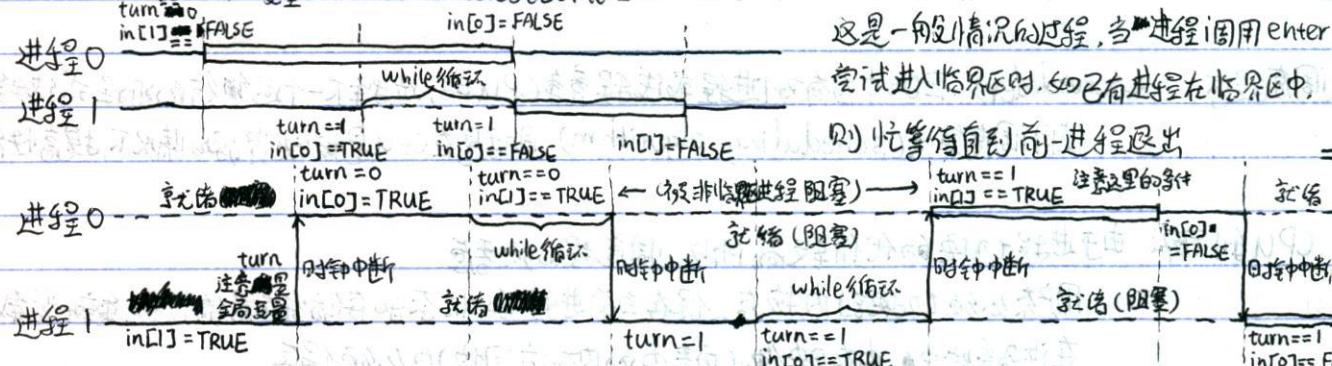
Peterson算法 Dekker算法（1965）结合锁变量和警告变量的思想，是一个不需要严格轮换的软件互斥算法

Peterson算法（1981）简单得多，由ANSI C编写（注意ANSI C要求为定义和使用的所有函数提供原型）

```
#define FALSE 0 void enter (int process) {  
#define TRUE 1 int other = 1 - process; (因为只有两个进程) interest [process] = FALSE;  
#define N 2 interest [process] = TRUE; (表示进入) } (表示离开)
```

int turn; (轮序) 间用两个全局变量

int interest [N]; (是否进入) while (interest [other] == TRUE); } (循环)



注意在这种两个进程几乎同时调用enter的情况下，全局变量turn的值取决于后执行赋值turn的进程（竞争条件）。而进程0成功进入临界区的原因并非 turn == 0, in[1] == FALSE, 而是 turn == 1, in[1] == TRUE。因此进程0实际上是被非临界区进程阻塞的。

另外，如果进程有优先级而进程1优先级过低，可能导致进程0被长时间阻塞，从而违反条件3。

# Operating

## System - P21

### TSL 指令

设计为多处理器的计算机有指令 TSL RX,LOCK, 称为测试并加锁 (test and set Lock)

用于将一个内存字读入寄存器 RX, 然后在该内存字存入一个非零值, 读写操作保证是不可分割的

执行 TSL 的 CPU 将锁住内存总线, 以禁止其他 CPU 在指令结束前访问内存, 需要特殊硬件设施

注意: 锁住内存总线不同于屏蔽中断, 因为屏蔽中断对其他处理器没有影响

enter: TSL REG, LOCK  
 $\begin{array}{l} \text{REG} \\ \text{LOCK} \end{array} \leftarrow \begin{array}{l} \text{REG} \leftarrow \text{LOCK} \\ \text{LOCK} \leftarrow \#1 \end{array}$

当 LOCK 为 0 时, TSL 指令取出 0 并置 1, 即完成上锁

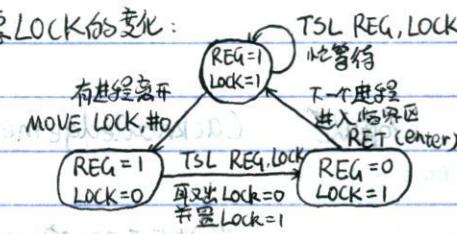
相当于一个忙等待, 不断测试 LOCK 的值直到 LOCK 为 0

CMP REG, #0      REG == 0?      JNE enter      REG != 0: enter

RET

leave: MOVE LOCK, #0      Lock  $\leftarrow \#0$       RET

置 LOCK 为 0



因为 TSL 保证读写操作不同分割, 从而避免出现竞争条件

### XCHG 可以替代 TSL 的指令, 用于原子性地交换两个位置的内容, 如一个寄存器 REG 和一个内存字 LOCK

如 MOVE REG, #1      REG  $\leftarrow \#1$       可见与 TSL REG, LOCK 是等价的

XCHG REG, LOCK      TEMP  $\leftarrow \text{LOCK}$       LOCK  $\leftarrow \text{REG}$       REG  $\leftarrow \text{TEMP}$       在 Intel X86 CPU 底层同步中使用 XCHG 指令

注意: TSL 和 XCHG 也存在忙等待的问题, 即优先级反转问题 (priority inversion problem)

即存在两个就绪即运行的高优先级进程 H 准备进入临界区, 而临界区是由低优先级进程 L

则因为 L 不被调用而无法离开临界区, 而 H 则阻塞在忙等待中, 违反了条件 3

### sleep/wakeup

sleep 是一个引起调用进程阻塞的系统调用 (类似于线程的 pthread\_yield), wakeup 调用有一个参数即唤醒进程

或者让 sleep 和 wakeup 各有一个参数用于匹配 sleep 和 wakeup 的内存地址

如此进程将不再忙等待, 而是使用 sleep 阻塞, 直到被 wakeup

### producer/consumer (生产者/消费者问题), 也称有限缓冲区问题 (bounded-buffer), 产生于多个进程共享一个固定大小公共缓冲区

即存在将信息放入缓冲区的生产者 (producer) 和从缓冲区取出信息的消费者 (consumer)

当生产者向已满的缓冲区放入信息 / 消费者从已空的缓冲区取出信息时, 则睡眠 (sleep)

直到消费者取出信息 / 生产者放入信息后, 则唤醒 (wake up)

void producer { int item;

while (TRUE) { item = item\_produce(); 生产项目; 如果缓冲为空, if (count == 0) sleep(); }

\* if (count == N) sleep(); 缓冲区已满

item\_insert(item); 向缓冲区放入信息

Count += 1; 如果 Count 为 1 (即放入后为空)

if (count == 1) ↓

wakeup(consumer); 唤醒 consumer

void consumer {

从缓冲区读取 item = remove\_item();

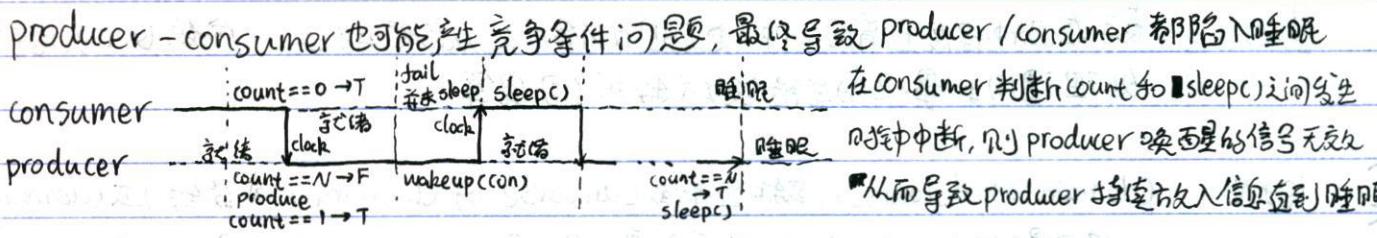
Count -= 1; Count -= 1;

如果 N-1 if (count == N-1) ↓

wakeup(producer);

取出信息 consume\_item(item);

# Operating System - P21



唤醒等待位：解决 producer/consumer 竞争条件的方案是修改规则加入唤醒等待位，用于取消 sleep() 作用是保存唤醒信号直到信号得到响应，即当信号发放给未睡眠的进程时，保存直到调用睡眠。当有多个进程时，虽然可以设置多个唤醒等待位，但并未从根本上解决问题。

信号量 Semaphore E.W. Dijkstra 1965 年提出用一个整型变量累计唤醒次数，0 表示没有保存下来的唤醒操作，正值表示有唤醒操作 down(一般化为 sleep) 检查值是否大于 0，是则减并继续(消耗 1 个信号)，为 0 则睡眠(注意此时 down 尚未结束) up(一般化为 wake up) 检查是否有进程在该信号量上睡眠，有则由 CPU 选择唤醒其一，否则加 1(累积 1 个信号) 注意这些进程都是因睡眠而 down 操作无法完成，唤醒则 down 完成，相当于累积与消耗相抵消。原子操作 一组相关联的操作要么都不间断地执行，要么都不执行(类似于 FOK, Fail Or Kill) 检查数值，修改变量值以及可能发生的睡眠操作均作为单一的，不可分割的原子操作 对于解决同步问题和避免竞争条件的绝对必要的(如用 TSL 和 XCHG 解决竞争条件)

解决互斥 用信号量解决 producer-consumer 问题，通常是将 up/down 作为系统调用实现的。且 OS 在执行测试信号量，更新信号量以及需要时使某进程睡眠操作时暂时屏蔽全局中断。由于操作仅需几条指令，所以操作其间屏蔽中断不会带来副作用。如果有多个 CPU，则需要一个锁变量保护信号量，通过 TSL 和 XCHG 确保同一时刻只有一个 CPU 操作信号量。注意与生产者/消费者忙等待对方操作是不同的，因为信号量操作仅数据指令，耗时 ms。解决方案设置 3 个信号量：full 用于记录充满的缓冲槽数目，empty 用于记录空的槽数目，mutex 用于防止同时访问。full 初值为 0，empty 初值为缓冲区中槽数，mutex 初始值为 1。

二元信号量 (binary semaphore) 供多个进程使用的信号量，初值为 1，保证同时只有一个进程进入临界区。如果每个进程进入时 down，离开时 up，则可以实现互斥，表现为只有 0 和 1，所以称二元 (binary)。这是因为每个 up 前必有一个 down，所以信号量不会大于 1，而不会因为 down 降至 0 以下，所以只有 0 和 1。

信号 使用信号量的进程有例如 I/O 设备，每个 I/O 设备设置一个初值为 0 的信号量。当设备启动时，管理进程立即对该 I/O 设备信号量执行 down，然后被阻塞。当 I/O 操作完成发送中断，中断处理程序对该信号量执行 up，则 I/O 设备管理进程就绪。

同步 synchronization 信号量用于保证某种事件的顺序发生或不发生 full 和 empty 保证缓冲区满时生产者停止，缓冲区空时消费者停止。

# Operating

## System - P22

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
#define semaphor type synonym int

void producer() { int item;
    while (TRUE) { item = produce_item();
        down(&empty); // 保证满时 producer 等待
        down(&mutex); // 信号量实现互斥
        insert_item(item);
        up(&mutex);
        up(&full); } } // 保证空时 consumer 等待

void consumer() { int item;
    while (TRUE) { down(&full); // 保证满时 consumer 等待
        down(&mutex); // 信号量实现互斥
        item = remove_item();
        up(&mutex);
        up(&empty); } } // 信号量实现同步
        consume_item(item); }
```

### 互斥量

#### mutex

不需要信号量的计数能力时的一个简化版本，仅适用管理共享资源或一段代码。

互斥量是一个处于两态（解锁/加锁）之一的变量，实现容易且有效，可用于用户空间线程包

mutex\_lock:

TSL REG, MUTEX

CMP REG, #0 ←

JZ ok

CALL thread\_yield (自动放弃CPU, 切换另一线程)

JMP mutex\_lock

ok: RET

mutex\_unlock:

MOVE MUTEX, #0

RET

注意与 enter\_region 的区别：

enter\_region 采用忙等待，在时间过长时

会被阻塞

而线程由于没有时间中断，所以采用忙等待

会进入死循环，所以调用 thread\_yield

由于没有用到系统调用，可以实现完全在用户空间  
同步，仅需少量指令

### 共享内存

Peterson 算法，信号量都隐藏了一个前提：进程间存在可访问的公共内存

方案一：存放在内核中的共享数据结构（如信号量），只能通过系统调用访问

方案二：进程间共享地址空间（部分），可以实现（缓冲区和其他数据结构共享）

方案三：可以使用共享文件，如果没有其他途径

即使进程间共享地址空间，其他单个进程的特性如打开文件，定时器均是独立的。

而在同一进程中的线程，共享进程的全部特性

共享公共地址空间无法使进程达到用户级线程的效率，因为涉及内核调用

### futex

（快速用户空间互斥），Linux 的一个特性，实现了基本的锁（类似互斥锁），但避免陷入内核。

包含一个用户库和一个内核服务，内核提供等待队列，允许多个进程在一个锁上等待，直到内核明确解除阻塞

进程共享通用的锁变量（对齐的 32 位整数锁），初始为 1，表示锁的释放状态

进程通过“减少并抢夺”（类似 down）来夺取锁，如果已锁，则执行系统调用进入等待队列

原子操作

进程通过“增加并抢夺”（类似 up）来释放锁，如果等待队列非空，则通知内核解除一个阻塞

注意没有竞争时 futex 完全在用户空间，内核不参与，而进程放入等待队列所需切换至内核的开销是合理的

# Operating System - P23

**条件变量** pthread 的另一种同步机制，条件变量允许线程由于一些未达到的条件而阻塞，通常与互斥量一起使用

**conditional** (线程调用)

- [`pthread_mutex_init`] 创建
- [`pthread_mutex_destroy`] 撤消
- 互斥量** `pthread_mutex_lock` 获得锁/阻塞
- `pthread_mutex_trylock` 获得锁/返回失败码 (相比 `lock` 更灵活)
- `pthread_mutex_unlock` 释放锁
- [`pthread_cond_init`] 创建
- [`pthread_cond_destroy`] 撤消
- `pthread_cond_wait` 阻塞并等待信号
- `pthread_cond_signal` 发信号唤醒
- `pthread_cond_broadcast` 广播信号唤醒

```
#define MAX ... int buffer = 0; //缓冲区
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; //互斥量 / 条件变量

int main(int argc, char **argv) {
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0); //创建互斥量
    pthread_cond_init(&condc, 0); //创建条件变量
    pthread_cond_init(&condp, 0); //创建条件变量
    pthread_create(&con, 0, consumer, 0); //创建消费者
    pthread_create(&pro, 0, producer, 0); //创建生产者
    pthread_join(pro, 0); //等待 pro, con 结束
    pthread_cond_destroy(&condc); //撤消条件变量
    pthread_cond_destroy(&condp); //撤消条件变量
    pthread_mutex_destroy(&the_mutex); //撤消互斥量
}

void *producer(void *ptr) {
    int i;
    for (i=1; i<=MAX; i++) {
        pthread_mutex_lock(&the_mutex);
        while (buffer != 0) //缓冲区非空时阻塞并等待
            pthread_cond_wait(&condp, &the_mutex);
        buffer = i; //向缓冲区放入数据 (简化)
        pthread_cond_signal(&condc);
        pthread_mutex_unlock(&the_mutex);
    }
    pthread_exit(0);
}

void *consumer(void *ptr) {
    int i;
    for (i=1; i<=MAX; i++) {
        pthread_mutex_lock(&the_mutex);
        while (buffer == 0) //缓冲区空时阻塞并等待
            pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; //从缓冲区取出数据 (简化)
        pthread_cond_signal(&condp);
        pthread_mutex_unlock(&the_mutex);
    }
    pthread_exit(0);
}
```

条件变量允许阻塞与等待原语本地进行，阻塞与等待的原因不是等待与发信号协议的一部分 (与机制分离)

阻塞的过程通常是等待其他线程完成工作，才可以继续运行，即需要保证顺序发生

注意 `pthread_cond_wait` (&cond, &mutex) 会原子性地调用并释放 mutex，使得其他线程可以进入

特别注意：条件变量 比信号量不保存在内存中，如果传递给一个没有线程等待的条件变量，则信号丢失

注意在这个例子中，只有一个 producer，一个 consumer 和一个缓冲池，所以设成简单地交替着运行

**死锁** (`dead lock`) 指两个或多个进程永远地阻塞下去，且解除阻塞的条件是另一个解除、阻塞并完成工作

如调换生产者-消费者 `producer { ... down(&mutex); ... }` `consumer { ... down(&mutex); ... }`

`down(&empty) go` `down(&empty);` `down(&full);`

`down(&mutex) if full` `down(&mutex);` `insert_item(); ... }` `remove_item(); ... }`

缓冲池已满时： `producer` `down(&mutex);` `down(&empty);` `--阻塞` `consumer` `就绪` `mutex未释放` `down(&mutex);` `--阻塞`

可见 `producer` 解除阻塞需要 `consumer` `down(&empty)`  
但 `consumer` 解除阻塞需要 `producer` `up(&mutex)`

# Operating System - P24

管程 (monitor), 由过程、变量及数据结构等组成的一个集合, 一个特殊的模块或软件包 (private)

进程可以调用管程中的过程, 但不能在管程之外声明的过程直接访问管程内的数据结构 (public)

注意: 管程的重要特性是任一时刻管程中只能有一个活跃进程(过程), 使管程有效完成互斥

典型方案为, 管程过程的前几条指令将检查是否有其他活跃进程, 有则挂起调用进程直到其他进程离开

注意: 管程是编译语言的组成部分, 进入管程的互斥由编译器负责, 相比由程序员安排出错可能更小,

通常的做法是用互斥量或二元信号量, 而程序员只需将临界区转换成管程过程即可

另外: 引入条件变量 (condition variable) 以及 wait/ signal, 实现进程逻辑上无法继续运行时被阻塞

在执行 signal 唤醒其他进程时, 需要设定规则以避免管程中有两个活跃进程

Hoare 建议让唤醒的进程运行, 而挂起调用 signal 的进程

Brinch Hansen 建议执行 signal 的进程必须立即退出, 即 signal 只能是管程过程的最后一条语句

相对而言 Brinch Hansen 的建议更简单也更容易实现

另一种方法是, 执行 signal 的进程继续运行, 在其退出后才允许其唤醒的进程运行

monitor ProducerConsumer

condition full, empty; integer count := 0;

procedure insert(item: integer);

begin

if count = N then wait(full); item := produce\_item;

insert(item);

count := count + 1;

唤醒消费者

if count = 1 then signal(empty);

end; end;

function remove: integer;

begin

if count = 0 then wait(empty);

remove := remove\_item;

item := ProducerConsumer.remove; item := ProducerConsumer.item;

count := count - 1; consumer\_item(item);

唤醒生产者

if count = N-1 then signal(full)

procedure producer; begin

begin

while true do

begin

producer(item);

end; end;

procedure consumer; begin

while true do

begin

consumer(item);

end; end;

end monitor;

注意: 尽管 wait/signal 与 sleep/wakeup 很像, 但管程的自动互斥避免竞争条件

可以将 count 看作一个在上下界间振荡的变量

一个在上下界间振荡的变量

一个在上下界间振荡的变量