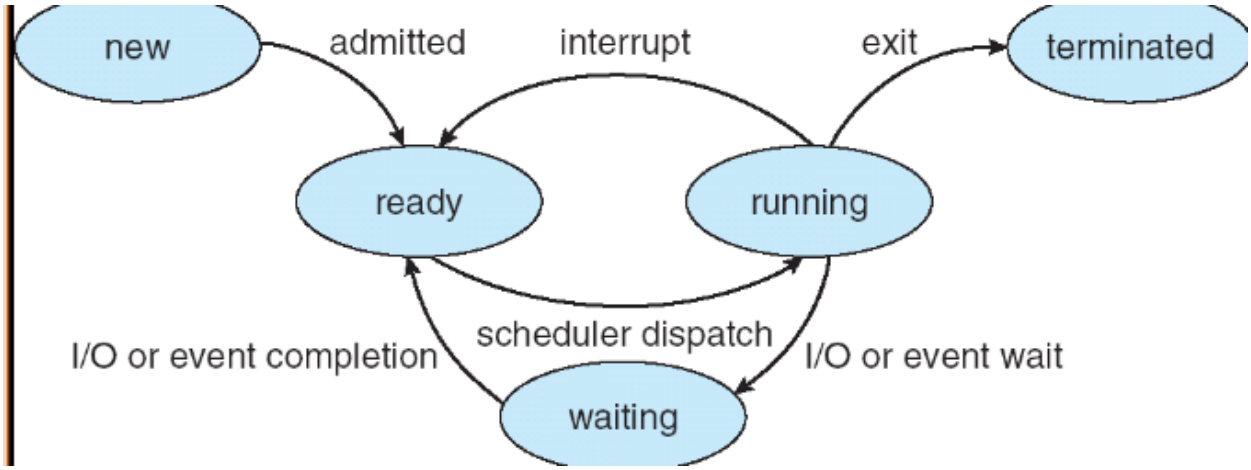
3. 进程：理解进程的并发执行

- 用户进程与内核进程的区别？
- 概念：执行中的程序，一个程序在一个数据集上的一次运行
- 进程包括：代码、程序计数器pc、堆栈、数据(全局变量、处理的文件)、堆(动态分配内存)
- 一个进程可以对应一个或多个程序，一个程序可以对应一个或多个进程，程序是进程的代码部分
- 进程的状态--进程状态变化图
  - 新建
  - 就绪
  - 运行
  - 等待
  - 终止
  - 

- 进程控制块pcb包括：
  - **程序状态**
  - 进程编号
  - 程序计数器
  - **cpu寄存器(中断以及恢复时需要用到)**
  - **cpu调度信息(进程优先级、调度队列的指针以及其它调度参数)**
  - **内存管理信息(基址与界限寄存器的值、页表或段表)**
  - **I/O状态信息(I/O设备列表、打开文件列表)**
- 在linux内核中，所有的活动进程是通过一个名为task_struct的双向链表来表示的，内核为当前正在运行的进程保存了一个指针 p75
- 多道程序设计目的是无论何时都有进程在运行、从而使得cpu利用率达到最大化; 分时系统的目的是在进程间快速切换cpu使得用户在程序运行时能与其交互
- **进程调度**：选择一个可用的进程(可能从多个可用进程集合中)到cpu上执行，**其实就是第5章cpu调度的内容**
- 作业队列、就绪队列、设备队列
- p77
- 调度程序：从各种队列中选择进程
  - 长期调度程序/作业调度程序
  - 中期调度程序
  - 短期调度程序/cpu调度程序

- 上下文切换：cpu切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态
- 进程创建：创建进程系统调用(create-process system call), solaris系统中的进程树
- fork：https://www.jianshu.com/p/586300fdb1ce
- 进程终止：子进程exit，父进程wait
- 级联终止
- 进程间通信(独立进程、协作进程)：
  - 共享内存：需要程序员自己编写代码，速度快，除了建立共享内存区域需要使用系统调用，其它无需内核介入，**在保护和同步方面，共享内存存在着一些问题**，会在第6章处理。如何达到buffer-size大小(增加一个count)
    - 实现手段：文件映射、管道、剪切板
  - 消息传递：分布式环境通信方便，需要内核(使用了系统调用)
    - 直接通信：send(P, message) / receive(Q, message) P/Q为进程
    - 间接通信：send(A, message) / receive(A, message) A为邮箱
    - 阻塞与非阻塞(同步与异步)
    - 缓冲大小
- 总结：
  - 本章主要解决进程的并发执行问题
  - 进程概念包括进程状态、pcb
  - **进程调度中 调度程序解决了如何从调度队列中选择一个进程在cpu上执行的问题**，在第5章
  - 上下文切换确保了进程并发执行时数据的保存与恢复问题(通过pcb完成)
  - 进程的操作包括创建与终止，着重讲解了父进程与子进程之间的关系
  - 进程间通信有两种手段：共享内存、消息传递确保了协作进程间共享数据
  - 针对开始提出的问题：
    - 用户进程工作在用户态、内核进程工作在内核态 ， 感觉上它们通过链表链接到一起了,由内核管理，所以到底如何区分？ p75页看一下

4. 线程：一个进程可能包含多个线程

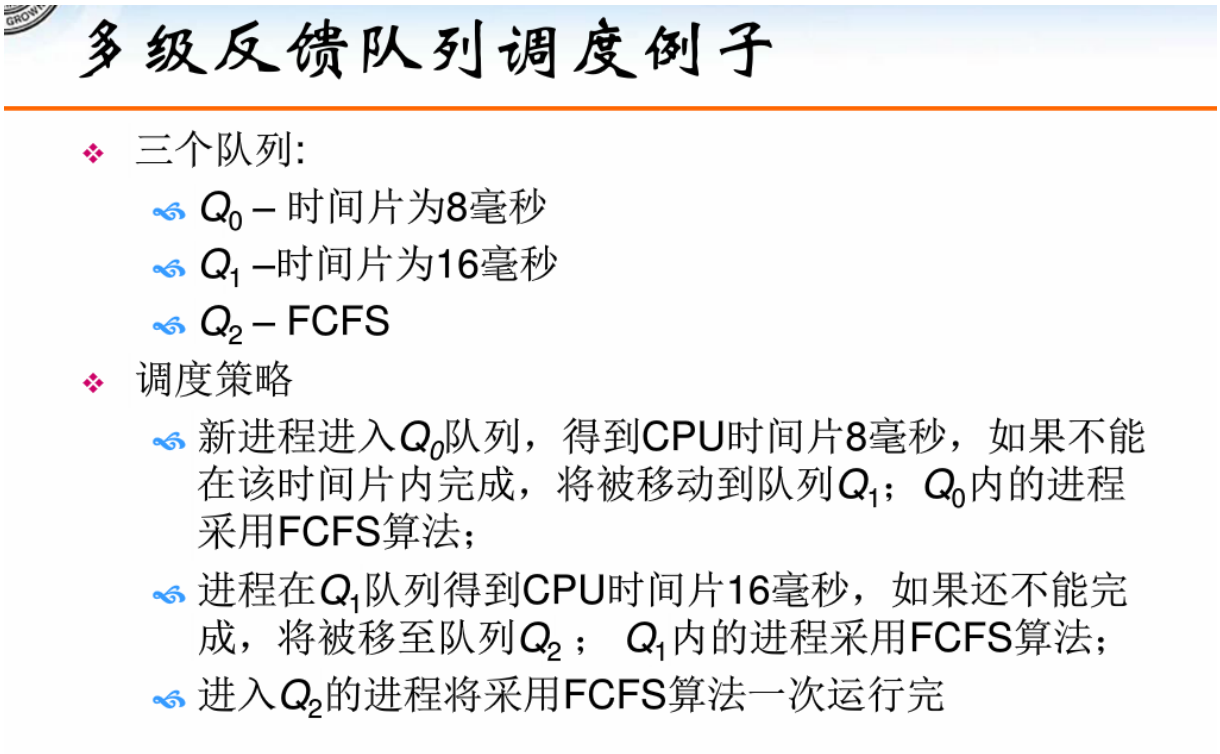
- 问题：fork为什么是轻量级线程， 进程与线程区别是什么？
- 为什么需要线程：创建进程开销大
- 线程：
  -

- **线程是在cpu上运行的基本执行单位，是基本调度单位**
- 进程内的一个代码片段可以被创建为一个线程
- **进程依旧是资源分配的基本单位**，线程自己不拥有资源，共享进程的资源，也可通过进程申请资源
- 线程操作：
- 线程控制块tcb
  - 线程id
  - 程序计数器pc
  - 寄存器集
  - 栈空间
  - 其它共享进程
- 多线程编程的优点：
  - 响应度高：web浏览器一个线程负责载入图像，另一个线程负责与用户交互
  - 资源共享：**线程默认共享所属进程的内存与资源**
  - 经济：进程创建所需内存与资源分配比较昂贵(因为子进程复制了父进程的地址空间)，线程由于共享进程的资源更为经济
  - **多处理器体系结构的利用：一个单线程的进程只能运行在一个cpu上，而一个多线程的进程可以在多个cpu上执行，这极大的加强了并发的功能。**
- 线程有两种(进程也有用户进程与内核进程)：
  - 用户线程：受内核支持而无需内核管理，由用户线程库管理，**内核看不到用户线程**，应用于只支持进程的操作系统 posix pthread / **win32 thread，与下文矛盾** / java thread
  - 内核线程：由操作系统内核管理 windows xp / solaris / linux
- **用户线程与内核线程必然存在一种关系**，是因为操作系统看不到用户线程，无法管理，所以通过把用户线程映射到内核线程上由内核管理？
  - 多对一模型：**用于不支持线程的操作系统**，将许多用户线程映射到一个**内核线程(是否为进程？)**，多个线程不能并行运行在多个处理器上，如果一个线程执行阻塞系统调用，那么整个进程会被阻塞，因为**任一时刻只有一个线程能够访问内核。**
  - 一对一模型：**用于支持线程的操作系统，**多个线程可以并行运行在多个cpu上，缺点是创建内核线程有开销，因而有上限，典型系统是linux、windows

    - 多对多模型：多路复用了许多用户线程到同样数量或更小数量的内核线程上，并发与效率兼顾，但增加了复杂度
- 线程库：为程序员提供创建和管理线程的API
    - posix线程 pthread：用户级线程库
    - java线程库：用户级别线程
    - win32线程库：内核库
- 总结：
    - 本章介绍了使用线程的原因、优点
    - 介绍了三种多线程模型：多对一、一对一、多对多
    - 介绍了三种线程库pthread、java、win32
    - 存在的问题有：为什么必须要有用户线程到内核线程的映射？不支持线程的操作系统是否只能使用一对一模型？linux到底支不支持线程？

5. cpu调度:

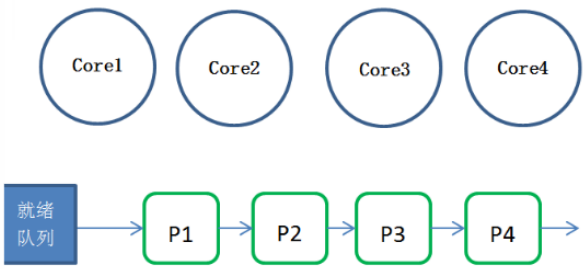
- **对于支持线程的操作系统是，是内核级线程被操作系统调度而不是进程**
- cpu为何能够调度：进程执行由cpu执行和I/O等待周期组成
- ×××状态到等待状态或者终止状态则称调度方案是非抢占的，否则为抢占式调度
    - 非抢占式调度：进程自愿放弃cpu，适合批处理，不适合交互
    - 抢占式调度：调度程序可以暂停某个正在执行的进程
    - 区别：**是否自愿放弃cpu**
- 与cpu调度相关的**分派程序**：是一个模块，负责将cpu的控制权交给短期调度程序选择的进程，功能包括
    - 上下文切换
    - 切换到用户模式
    - 跳转到用户程序的合适位置，以重新启动程序
- 长程调度(必须)：又称作业调度，“新建”状态转换到“就绪”状态，控制多道程序的道度
- 短程调度(可选)：又称cpu调度，调度程序选择下一个执行进程
- 中程调度：又称交换，将进程在内存与外存间换进换出，节省内存空间
- 调度准则：为了比较cpu调度算法
    - cpu利用率
    - 吞吐量
    - 周转时间：从进程提交到进程完成的时间段
    - 等待时间：进程等待调度(不运行)的时间片总和
    - 响应时间：从提交请求到产生第一响应的时间，也就是第一段等待时间
    - 周转时间 = 等待时间 + 运行时间，响应时间 <= 等待时间
- **调度算法**：
    - FCFS，先来先服务调度算法
        - 实现简单
        - 非抢占，公平
        - 适用于长程调度，后台批处理系统的短程调度
    - **SJF，最短作业优先调度算法**
        - 每次选择具有最短cpu区间的进程来执行，一旦区间执行不会被打断
        - 具有最短的平均等待时间
        - 难以知道下一个cpu区间的长度
        - **存在饥饿问题**
    - **SRTF，最短剩余时间优先调度，抢占式的SJF，这个可能会考**

    - 每次选择具有最短cpu区间的进程来执行，执行过程中可能被打断
- PR，优先级调度
    - 赋予进程优先级，cpu分配给具有最高优先级的进程
    - 优先级可静态不变，也可以动态调整
    - 动态优先级举例：高响应比优先调度算法
        - 优先数=响应比=等待时间/运行时间，越大优先级越高
    - 优点：实现简单，考虑了进程的紧迫程度，灵活，**可模拟其它算法**
    - 缺点：存在饥饿问题，可以通过**老化(根据进程等待时间提高优先数)**解决
- RR，时间片轮转算法
    - 转为分时系统设计，类似与FCFS，但增加了抢占
    - 为每一个进程分配不超过一个时间片的cpu，时间片用完后，该进程被抢占并插入就绪队列末尾，循环执行
- MLQ，多级队列调度
    - 不同类型的进程需要不同的策略
        - 交互进程需要短的响应时间
        - 批处理进程需要短的等待时间
    - 系统中存在多个就绪队列，每个队列有自己的调度算法
    - **需要决定将新进程加入哪个队列**
    - 先调度哪个队列？给定时间片，每个队列分配一定比例的时间
- MLFQ，MultiLevel Qeedback Queue scheduling多级反馈队列调度
    - 多级队列的延伸，增加了决定队列升级的方法，决定队列降级的方法
    - 

- 多处理器调度()
    - 适用于多核处理器的cpu调度
    - 对称多处理器(SMP)，每个处理器决定自己的调度方法，主流方案
    - 非对称多处理器(ASMP)，
    - 处理器亲和性：进程在某个cpu上尽量长时间地运行而不被迁移到其它处理器的倾向性

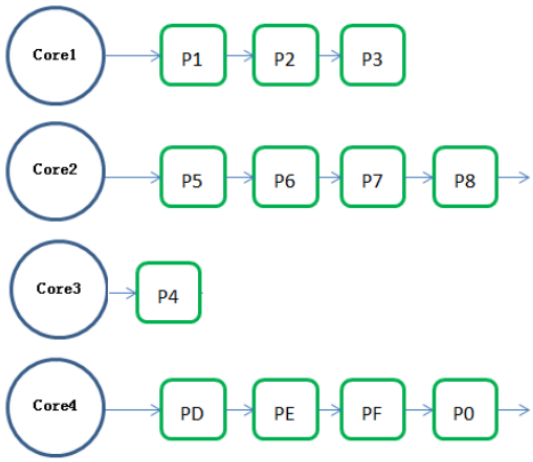
单队列调度方法（SQMP）

- 单队列多核调度方法(Single-Queue MultiProcessor Scheduling)
- 系统有一个就绪队列。当任意一个CPU空闲时，就从就绪队列中选择一个进程到该CPU上运行
- 优点：
  - 容易从单核调度算法推广到多核/多处理器、
  - 实现简单，负载均衡
- 缺点：
  - 不具有亲和性
  - 加锁问题



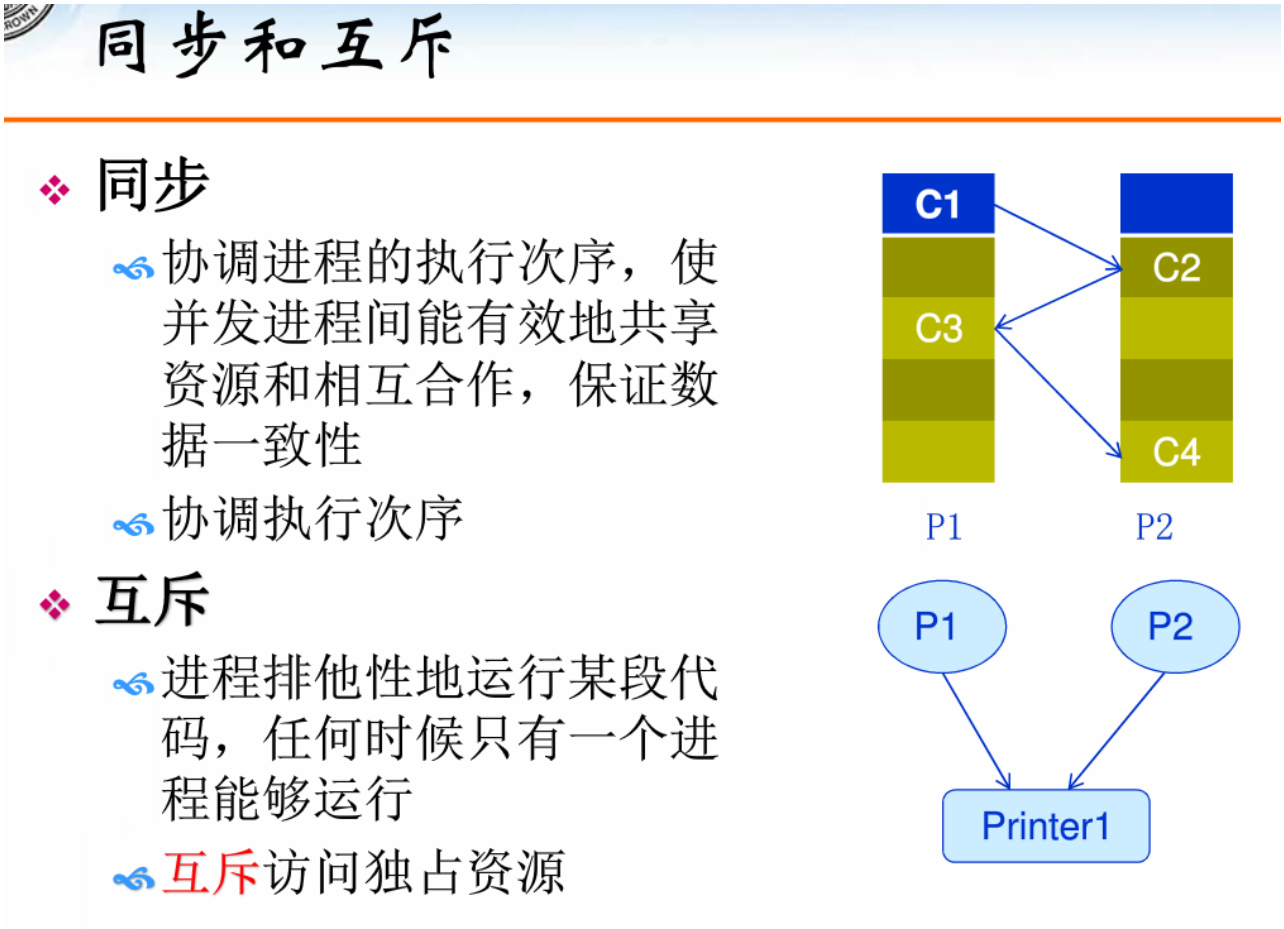
多队列调度方法（MQMP）

- 多队列调度方法(Multi-Queue MultiProcessor Scheduling)
- 系统有多个就绪队列，一般每个CPU一个。每个就绪队列有自己的调度算法，并且每个就绪队列的调度相对独立
- 优点：
  - 亲和性好
  - 不需要加锁
- 缺点：
  - 负载不均衡
- 策略：“偷”进程



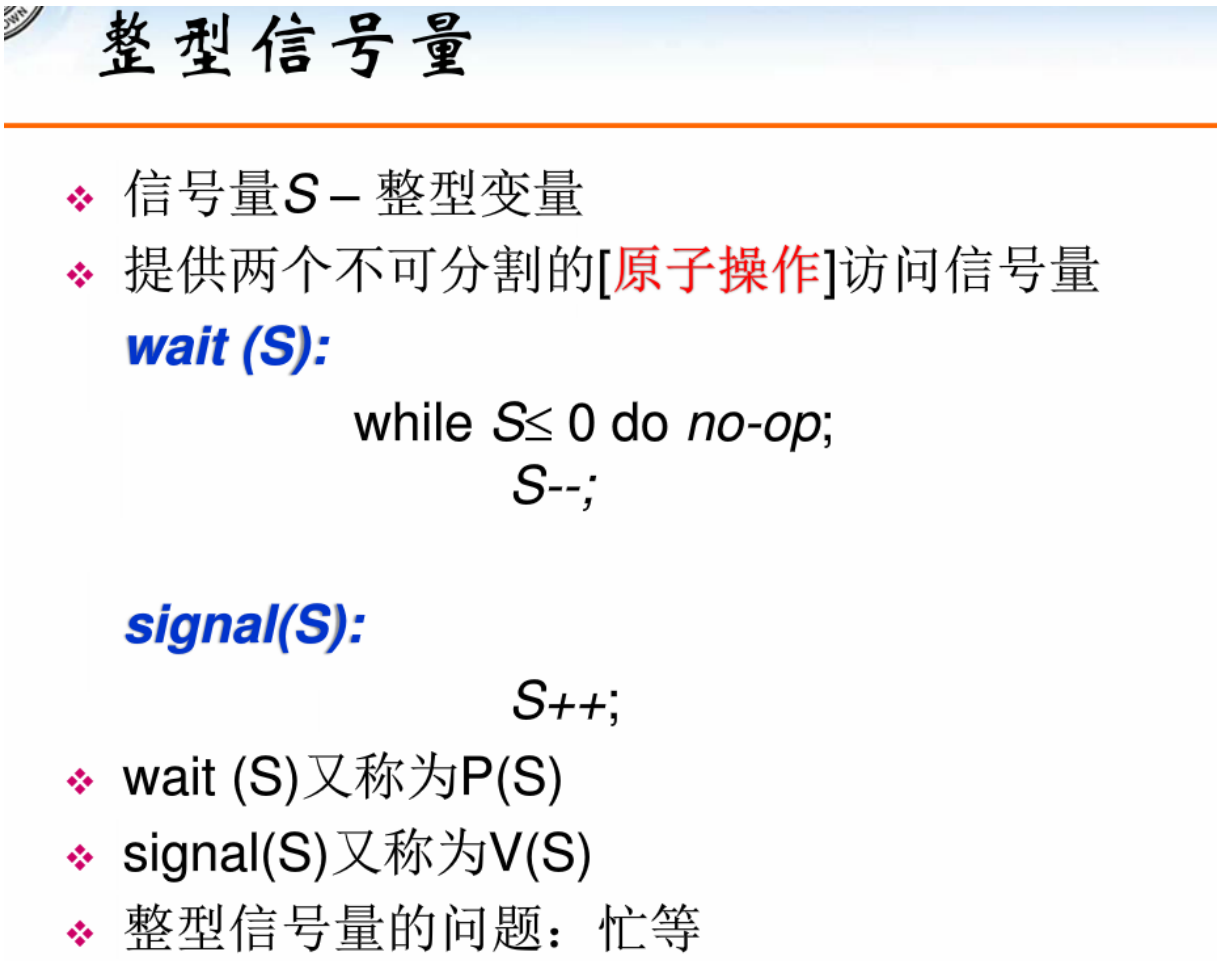
- 总结
  - 本章主要讲解了cpu调度(即进程调度)的各种算法，包括FCFS、SJF、SRTF、PR、RR、MLQ、MLFQ、、、
  - 到此为止关于进程正确并发执行还差两个问题没有解决---1.进程的同步 2.死锁
  - 有个关于调度算法的选择题需要看
  - 问题：进程调度与线程调度区别

6. 进程同步：关于count的一道选择题

- 背景：共享数据的并发访问可能产生数据的不一致性，
- 竞争条件：多个进程并发访问同一个共享数据的情况
- 防止竞争条件的方法：并发进程同步或互斥
- 

- 临界资源(critical resource)：一次只允许一个进程使用的资源，又称为互斥资源、独占资源或共享变量
- 共享资源：一次允许多个进程使用的资源
- 临界区：涉及临界资源的代码段，每个进程有一个或多个临界区，设置方法由程序员决定
  - **使用准则**：
    1. 互斥
       - **如果进程在临界区内执行，那么其它进程都不能在其临界区内执行**
       - 有相同临界资源的临界区都需互斥
       - 无相同临界资源的临界区不需要互斥
    2. 有空让进
       - **临界去内无进程执行，不能无限期地延长下一个要进临界区进程的等待时间**
    3. 有限等待
       - **每个进程进入临界区前的等待时间必须有限**
       - 不能无限等待
  - **访问临界区的过程**
    1. **在进入区实现互斥准则**
    2. **在退出区实现有空让进准则**
    3. **每个临界区不能过大，从而实现有限等待准则**

- 书上有peterson算法的相关介绍，ppt上没有
- 硬件同步
  - 对于单处理器环境：在修改共享变量时**禁止中断出现**，这种方法通常为非抢占式内核使用
  - 现代计算机有特殊的硬件中断指令，可以用这些指令来原子性地检查和修改字的内容或交换两个字的内容
- 由于硬件实现对程序员而言太复杂，因而出现信号量
  - 整型信号量，S>0可以获得信号量，S<=0不能获得信号量， 缺点是忙等待(也称为自旋锁)



  - 记录型信号量，**不是忙等而是阻塞自己，阻塞操作将一个进程放入到与信号量相关的等待队列中，并将该进程的状态切换为等待状态，接着控制转到cpu调度程序选择另一个进程来执行。**如果信号量为负，则其绝对值就是等待该信号量的进程的个数。

记录型信号量：去除忙等的信号量

```
Wait(semaphore *S)
{
    S->value--;
    if (S->value < 0) {
        add this process to list S->list;
        block();
    }
}

Signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from list S->list;
        wakeup(P);
    }
}
```

记录型信号量定义：

```
typedef struct {
        int value;
        struct process *list;
} semaphore
```

- 一个被阻塞在等待信号量S的进程，可以在其它进程执行signal()操作之后被重新执行。该进程的重新执行是通过wakeup()操作来进行的。

- 信号量的类型
  - 计数信号量==同步信号量，变化范围为没有限制的整型值
  - 二值信号量==互斥信号量，变化范围限于0与1
- 信号量S的使用
  - S必须置一次且只能置一次初值
  - S初值不能为负数
  - 除了初始化，只能通过PV操作来访问S
-

互斥信号量使用

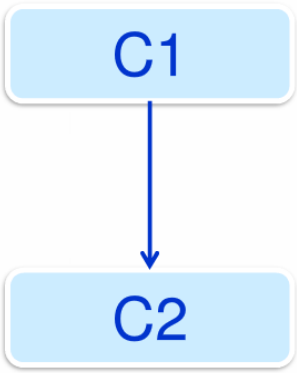
```
Semaphore *S;          // 初始化为 1

wait(S);
CriticalSection()      //临界区
signal(S);
```

-

同步信号量使用

- 实现各种同步问题
- 例子：P_1 和 P_2 需要 C_1 比C_2先运行

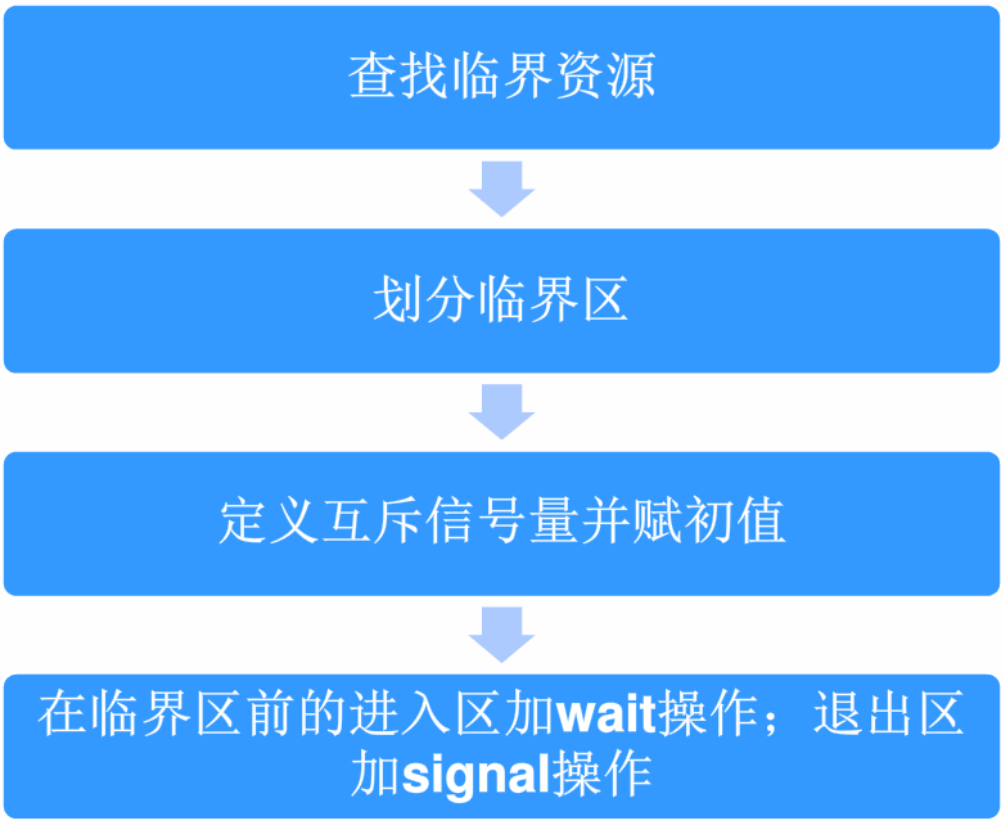
```
semaphore s=0
P1:
    C1;
    signal(s);

P2:
    wait(s);
    C2;
```



- 信号量的关键之处在于原子的执行，也就是说wait()与signal()必须原子的执行，可通过两种方法加以解决
  - 单处理器：执行wait与signal时禁用中断
  - 多处理器：禁止每个处理器的中断(困难，影响性能)
- **信号量的使用产生的问题：死锁与饥饿，其它类型的事件也可能导致死锁**，第7章会解决
- 经典同步问题
  1. **生产者消费者问题：共享有限缓冲区**
  2. **读者写者问题：数据读写问题**
  3. **哲学家就餐问题：资源竞争**
- **生产者消费者问题**

- 生产者M个，消费者N个，缓冲区大小有限，如何实现生产者与消费者之间的同步

互斥分析基本方法



6

生产者消费者的互斥分析

- 临界资源
  - 生产者
    - 把产品放入指定缓冲区
    - **in**:所有的生产者对in指针需要互斥
    - **counter**：所有生产者消费者进程对counter互斥
  - 消费者
    - 从指定缓冲区取出产品
    - **out**:所有的消费者对out指针需要互斥
    - **counter**：所有生产者消费者进程对counter互斥

```
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
counter++;
```

```
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
```

7

增加互斥机制

semaphore *m; m->vaule = 1;

生产者：

```
{
   …
   生产一个产品
   …

   wait(m);
   …
   把产品放入指定缓冲区
   …
   signal(m);

}
```

临界区

消费者：

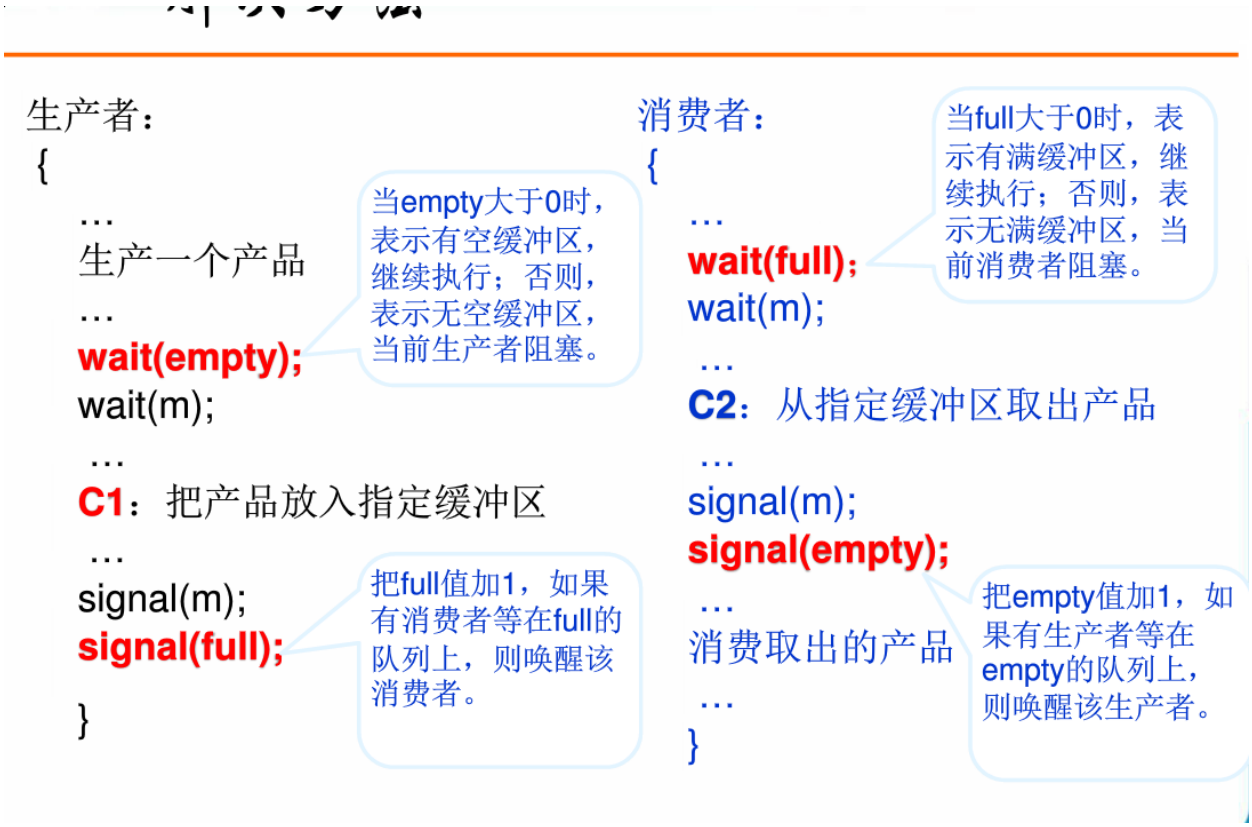
```
{
   …

   wait(m);
   …
   从指定缓冲区取出产品
   …
   signal(m);

   …
   消费取出的产品
   …
}
```

临界区

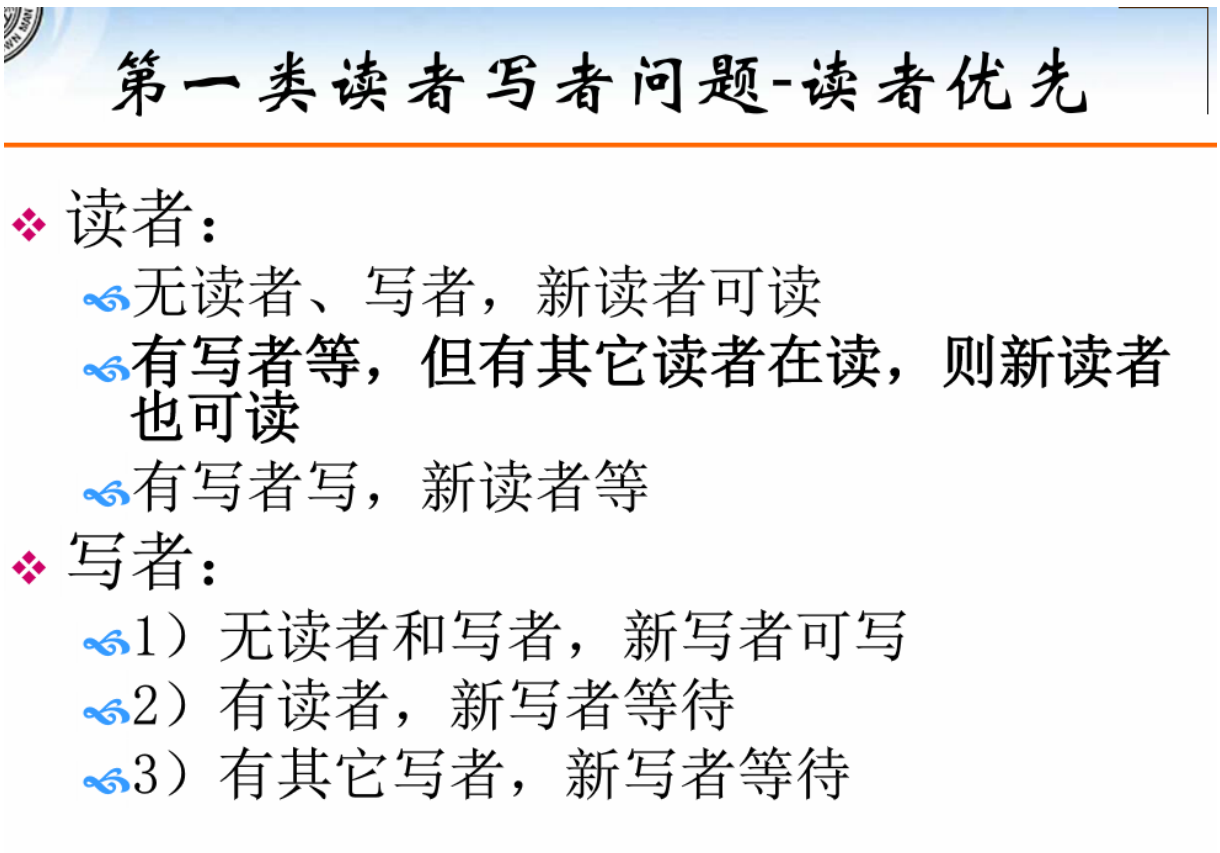
同步分析

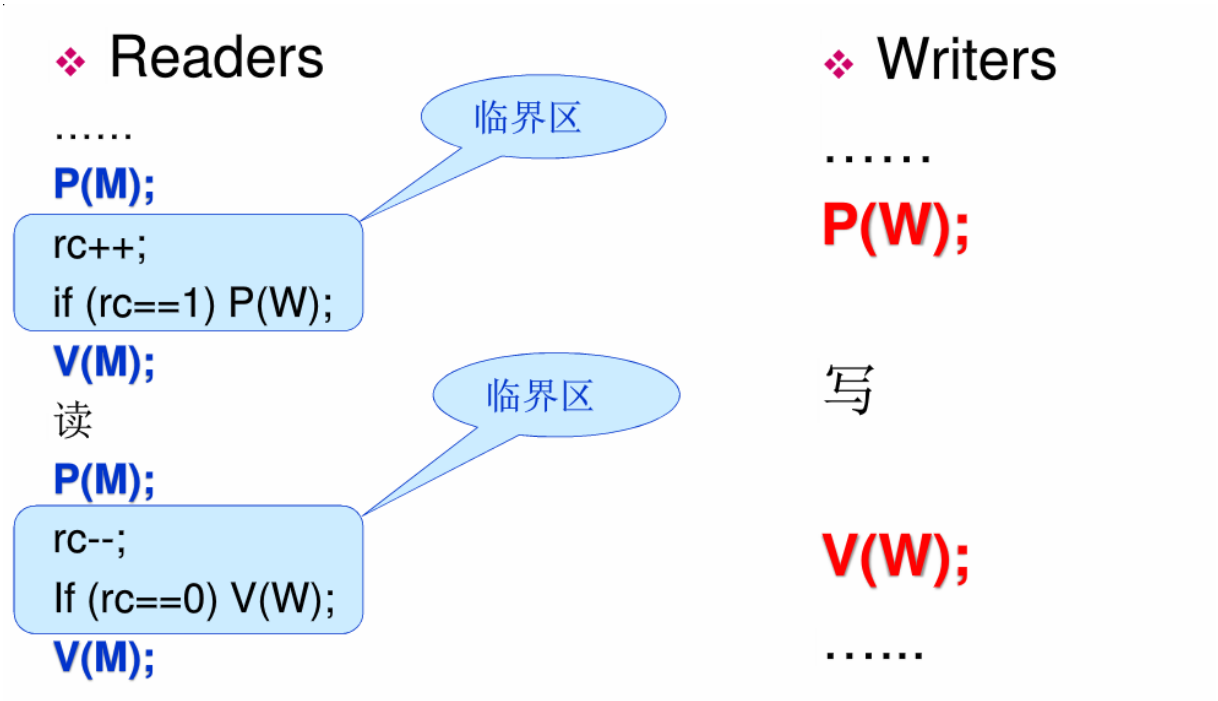
- 找出需要同步的代码片段（关键代码）
- 分析这些代码片段的执行次序
- 增加同步信号量并赋初始值
- 在关键代码前后加**wait**和**signal**操作

同步分析较为困难

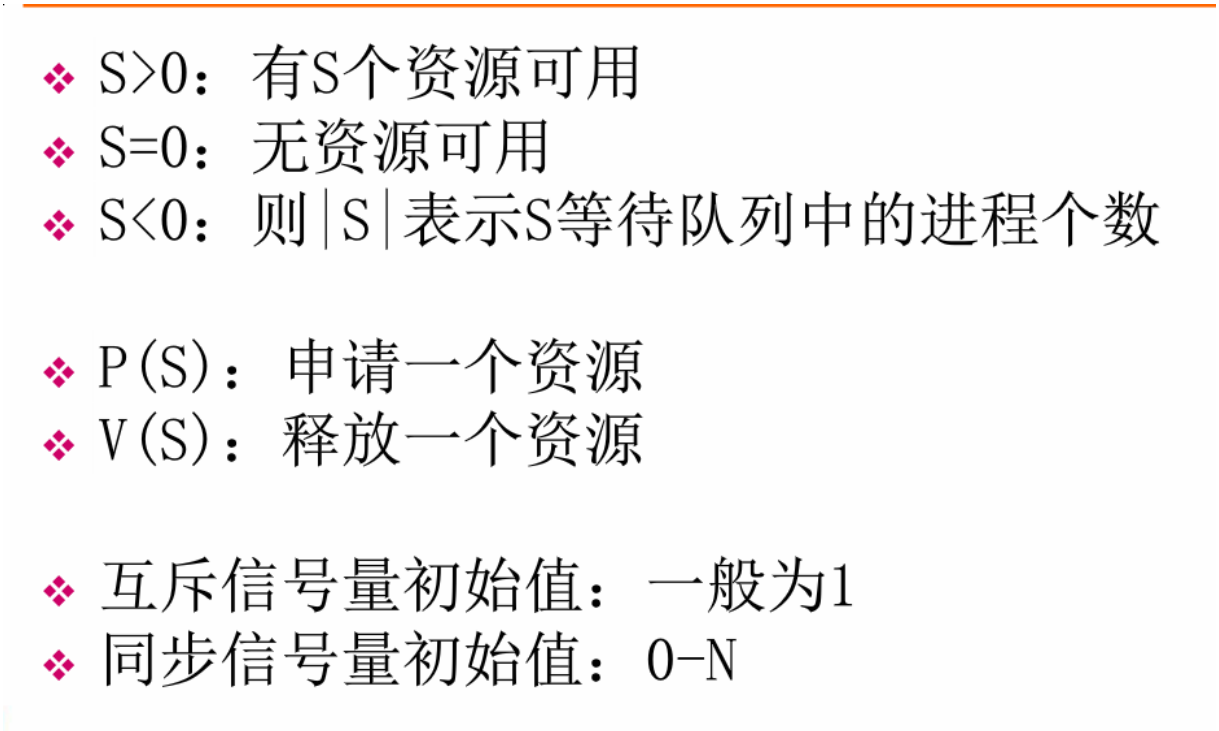


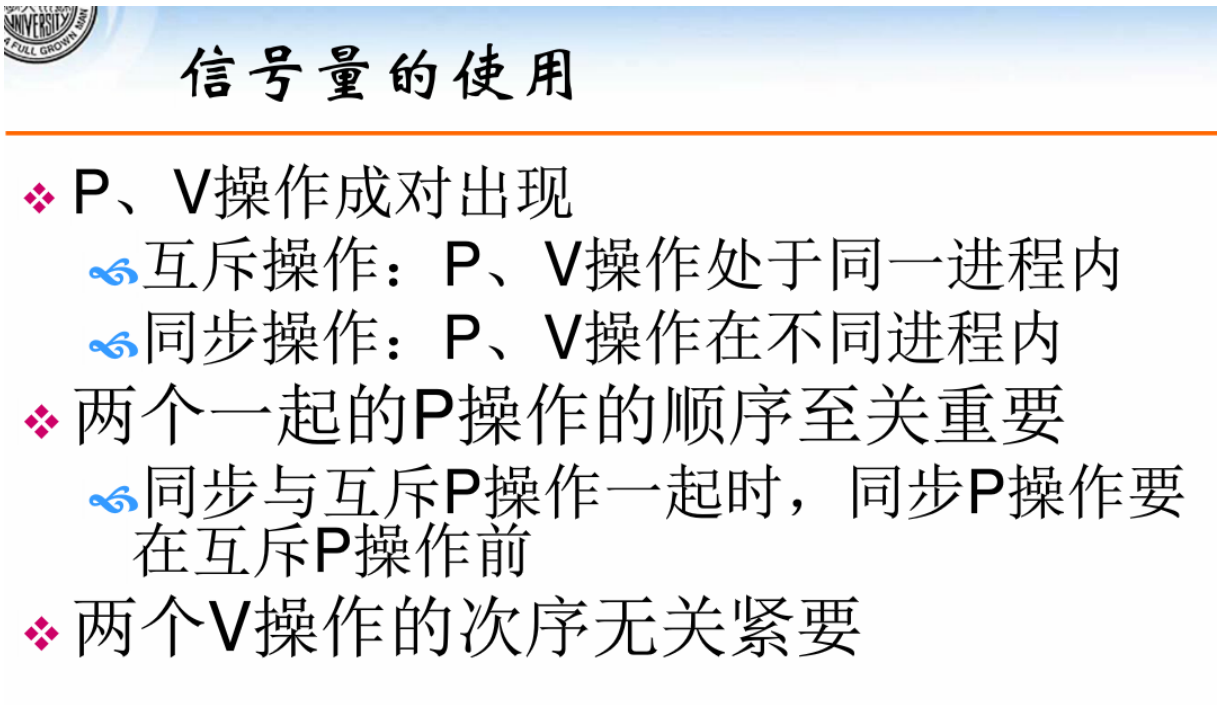
- 读者写者问题
  - 两组并发进程为读者与写者进程，共享一组数据区进行读写，要求允许多个读者同时读，不允许读者写者同时读写，不允许多个写者同时写
  - 读者优先





- 哲学家就餐问题
  - 看ppt
- 信号量S的物理含义





- 管程
  - 信号量的问题：编程容易出错
  - 解决方法：管程monitor=========高级语言构造
  - 什么是管程：一个管程定义了**一组数据结构**和**能为并发进程所执行(在该数据结构上)的一组操作**，这组操作能同步进程和改变管程中的数据
  - 管程的功能：
    - 互斥
      1. 管程中的变量只能被管程中的操作访问
      2. 任何时候只有一个进程在管程中操作
      3. 类似临界区
      4. 由编译器完成
    - 同步
      1. **条件变量**------->为了编写同步代码而设计的，是一个条件队列，执行wait()操作时把调用进程挂起到该条件队列上，执行signal()操作时从条件队列中查看是否有挂起进程，没有则什么都不做，有则唤醒一个进程。

条件变量问题

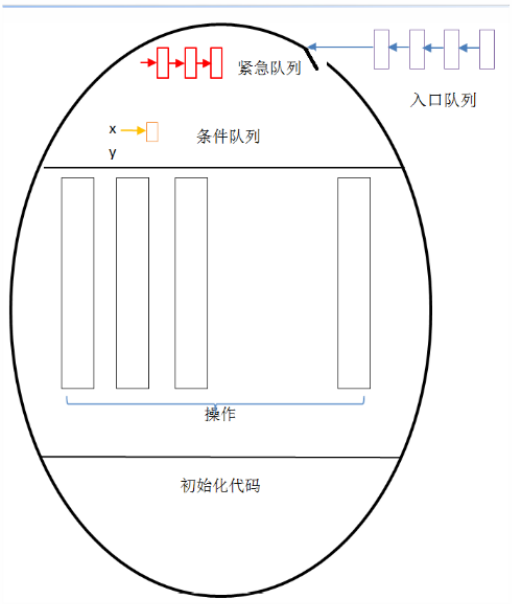
- 管程内可能存在不止1个进程
  - 如：进程P调用signal操作唤醒进程Q后
- 存在的可能
  - P等待直到Q离开管程 （Hoare）
  - Q等待直到P离开管程（Lampson & Redll，MESA语言）
  - P的singal操作是P在管程内的最后一个语句(Hansen，并行Pascal)

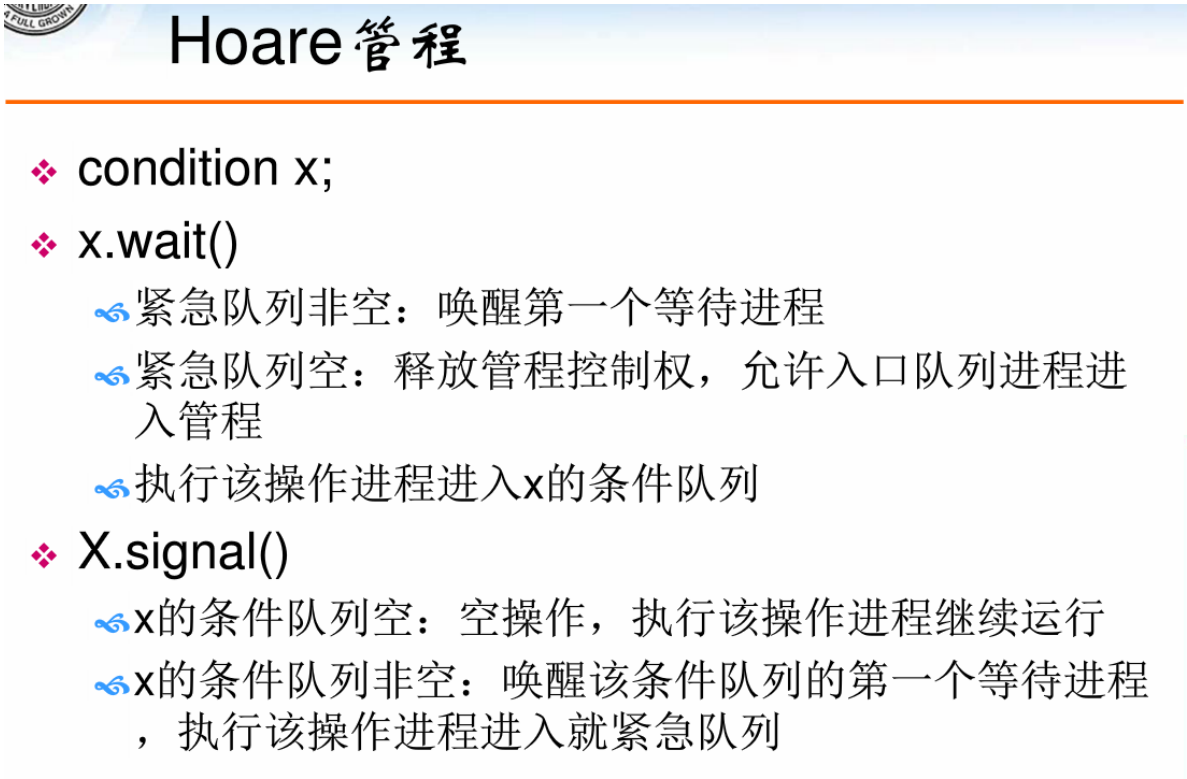
2. 唤醒和阻塞操作

- HOARE管程

Hoare管程

- 进程互斥进入管程
  - 如果有进程在管程内运行，管程外的进程等待
  - 入口队列：等待进入管程的进程队列
- 管程内进程P唤醒Q后
  - P等待，Q运行
  - P加入紧急队列
  - 紧急队列的优先级高于入口队列





- 哲学家就餐问题的Hoare管程解决方案，看ppt，看视频
- windows与linux的同步机制，看ppt，看视频

- 总结
  - 本章主要讲解了进程的同步与互斥
  - 介绍了临界区的概念
  - 介绍了两种实现进程同步与互斥的方法，包括信号量、管程
  - 具体讲解了三类经典同步问题：1.有限缓冲问题； 2. 读者-写者问题(读者优先、写者优先未涉及); 3.哲学家就餐问题(包括用信号量与管程机制解决)
  - 最后是windows与linux的进程同步实例

7. 死锁

- 概念：一组等待的进程，其中每一个进程都持有资源，并且等待着由这个组中其他进程所持有的资源
- **死锁的特征(4个必要条件同时出现，死锁将发生)**：
  - 互斥：一次只有一个进程可以使用一个资源
  - 占有并等待：一个至少持有一个资源的进程在等待另一个资源，而该资源为其它进程所占有
  - 非抢占：资源只能在进程完成任务后才能被自动释放
  - 循环等待：等待资源的进程之间存在环{P0，P1，...，P0}，P0等待P1的资源，P1等待P2的资源，Pn等待P0的资源
- 资源分配图

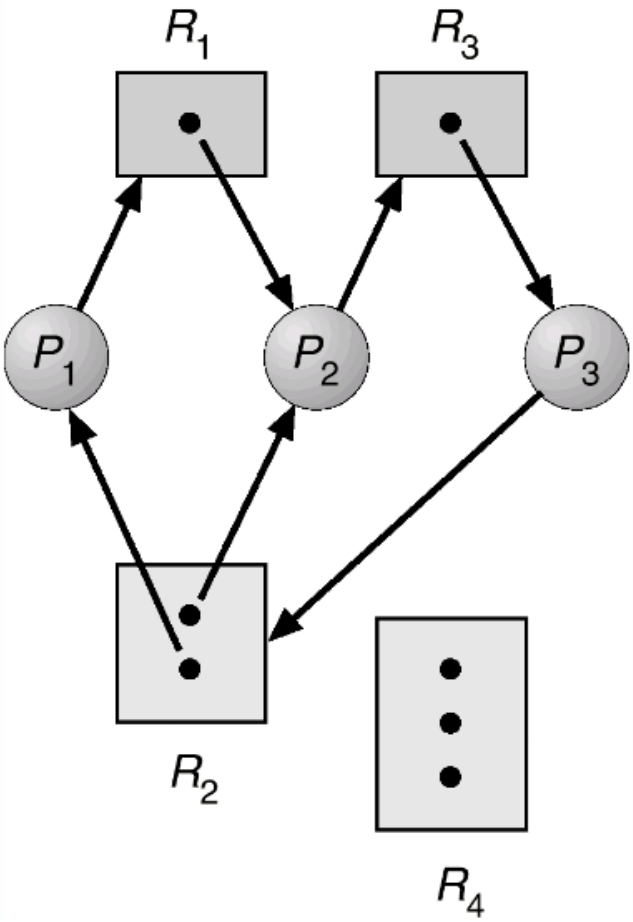
资源分配图

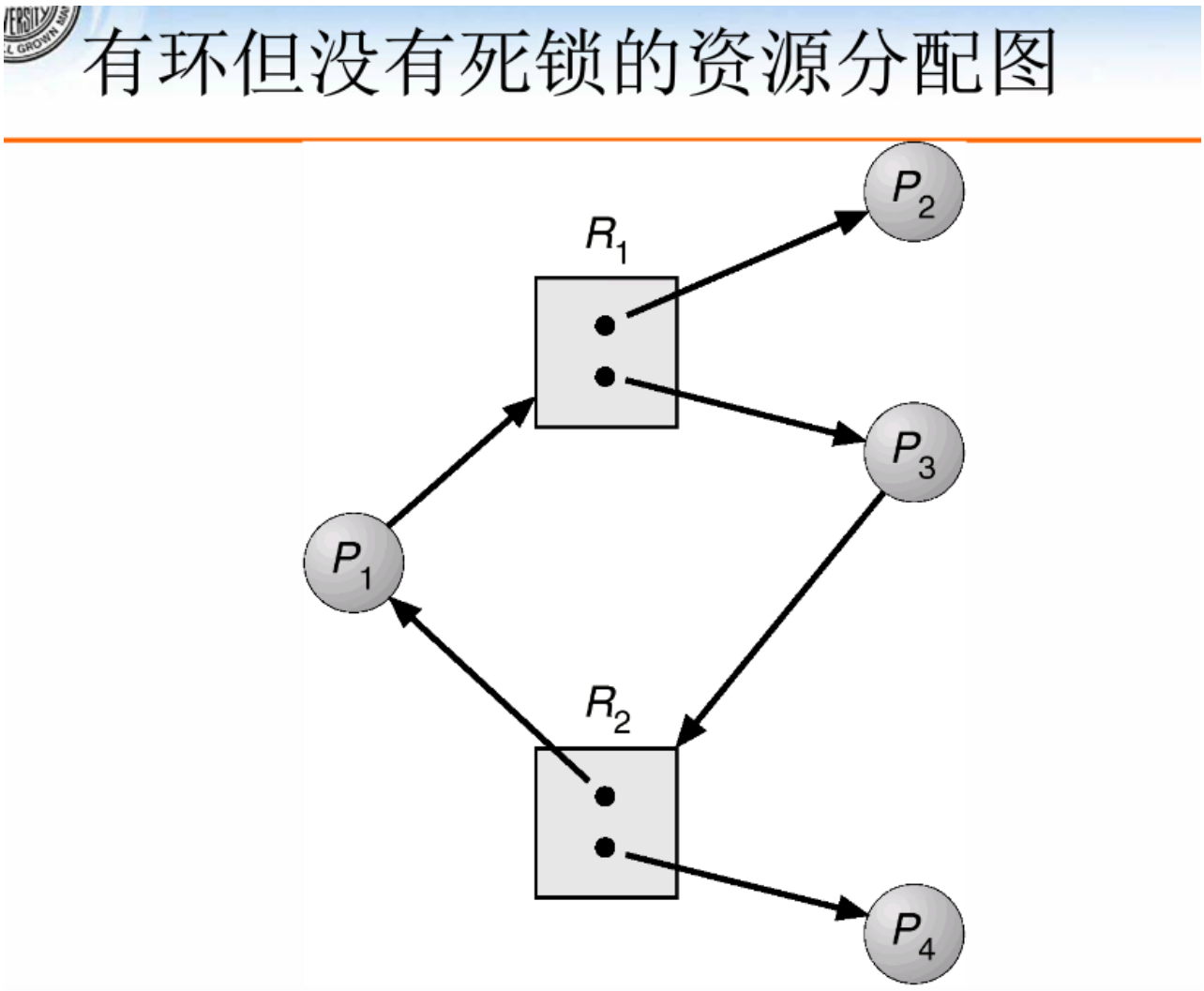
一个顶点的集合**V**和边的集合**E**

- V被分为两个部分
  - $P = \{P_1, P_2, \ldots, P_n\}$, 含有系统中全部的进程
  - $R = \{R_1, R_2, \ldots, R_m\}$, 含有系统中全部的资源
- 请求边：有向边$P_i \rightarrow R_j$
- 分配边：有向边$R_i \rightarrow P_j$

- 进程用圆形，资源用矩形
- 资源中的实心点表示资源实例

有环有死锁的资源分配图



- 

- 如果图没有环，那么不会产生死锁
- 如果图有环
  - **如果每一个资源只有一种实例，那么死锁发生**
  - **如果每一个资源有多个实例，那么可能死锁**
- 处理死锁的方法
  - 确保系统永远不会进入死锁状态 死锁预防 死锁避免
  - 允许系统进入死锁状态，然后检测它并恢复 死锁检测 死锁恢复
  - 忽视该问题，假装系统从未出现死锁 最常用，由开发人员自己处理死锁
- **死锁预防**
  - **确保4个必要条件至少一个不成立**
  - 1. 互斥：现代操作系统中的虚拟化技术可以将互斥资源改造为共享资源
    2. 占有并等待：静态分配策略(进程在执行前一次性申请所有的资源，利用率低可能出现饥饿)
    3. 非抢占：资源不能被抢占，即资源只能在进程完成任务后自动释放，可用于寄存器和内存，不适合其它资源
    4. 循环等待：对所有资源排序，每个进程按照递增顺序来申请资源
- 死锁避免(需要系统有一些额外的信息，确保循环等待条件不可能成立)
  - 安全状态

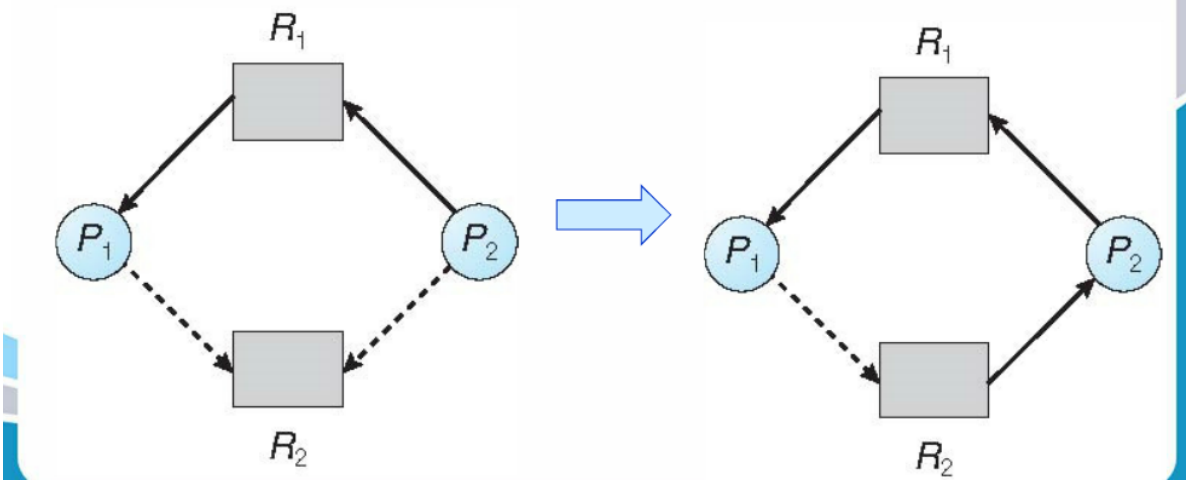
安全状态

- 当进程申请一个有效的资源的时候，系统必须确定分配后是安全的
- 如果存在一个安全序列，系统处于安全态
- 进程序列<P_1, P_2, …, P_n>是安全的，如果每一个进程Pi所申请的可以被满足的资源数加上其他进程所持有的该资源数小于系统总数
  - 如果 P_i 需要的资源不能马上获得，那么P_i 等待直到所有的P_{i-1} *进程*结束。
  - 当P_{i-1} 结束后， P_i获得所需的资源，执行、返回资源、结束。
  - 当P_i结束后， P_{i+1}获得所需的资源执行，依此类推。

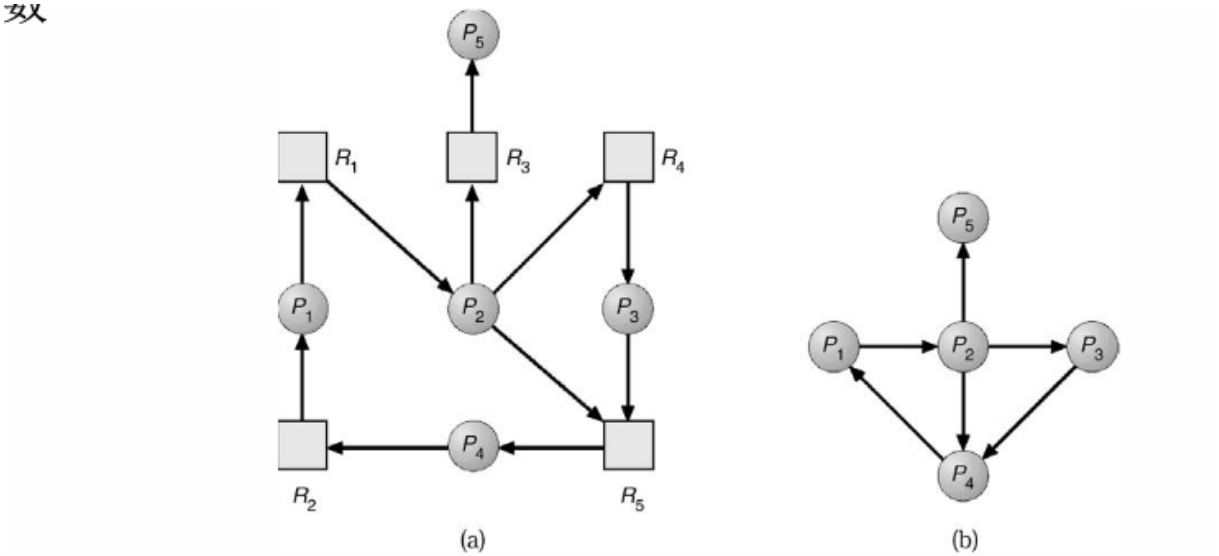
- 如果一个系统在安全状态，就没有死锁
- 如果一个系统不是处于安全状态，就有可能死锁
- 死锁避免就是确保系统永远不会进入不安全状态
- 避免算法
  - 单实例资源：资源分配图法
  - 多实例资源：银行家算法
- 资源分配图法(单实例)：
  - 在资源分配图上加上需求边：p-->R表示P可能以后需要申请的资源R，用虚线表示

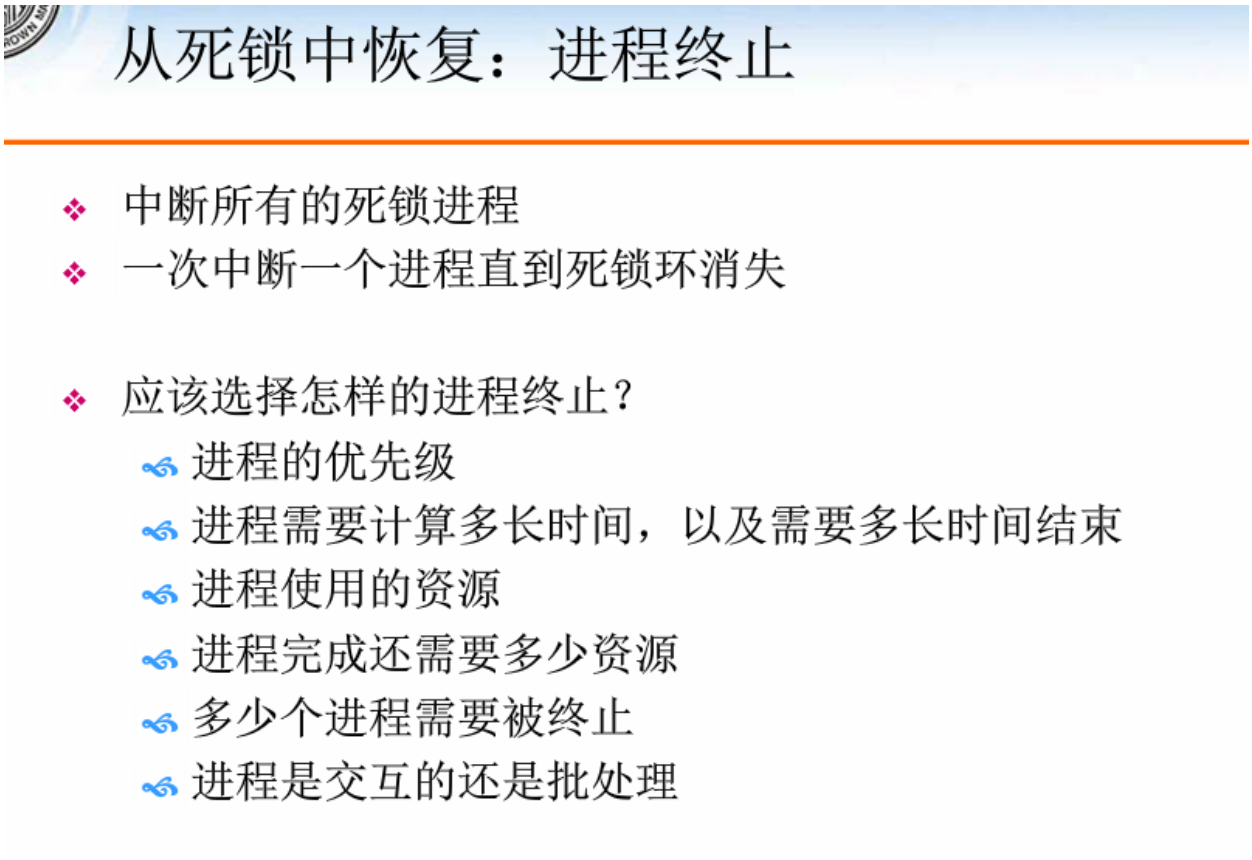
资源分配图算法

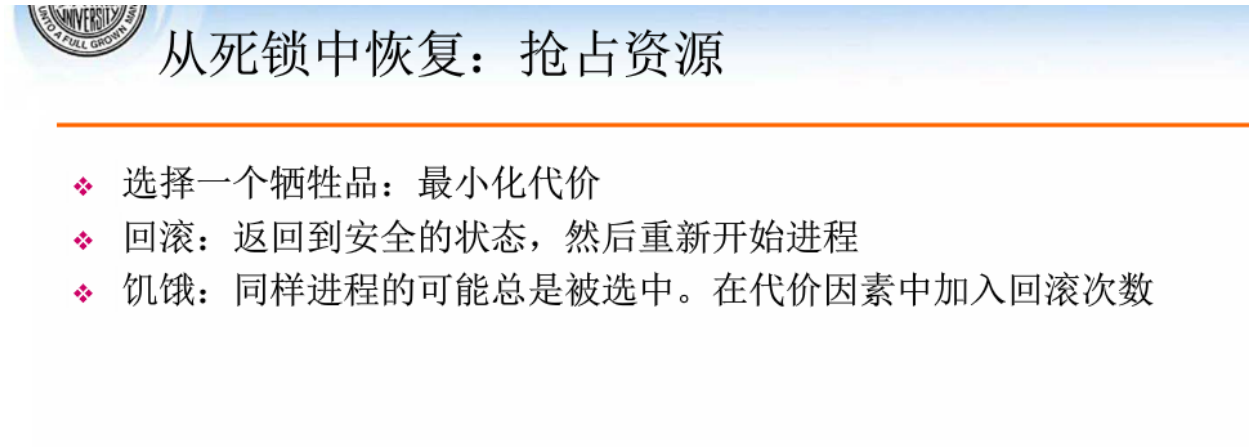
- 假设 P_i 申请资源 R_j
- 请求能满足的前提是：把请求边转换为分配边后不会导致环存在



- 银行家算法(多实例)
  - 有一个安全性算法用于判断当前系统是否处于安全状态
  - Max表示进程P请求的所有资源的最大数量
  - Allocation表示给进程P已经分配的资源数量
  - Available表示整个系统空闲的资源数量
  - Need表示当前进程P还需要的资源数量 Need=Max-Allocation
  - 如何判断进程对资源的请求 Request 能否满足：假定满足，Available=Available-Request Allocation=Allocation+Request Need=Need-Request，执行安全性算法判断当前系统是否处于安全状态，若处于则可以满足请求，若不处于则不能满足，需回复原来状态
- 死锁检测
  - 如果一个系统不采用死锁预防也不采用死锁检测算法，则可能出现死锁
  - 单实例
    - 等待图====资源分配图的一个变种，去掉所有资源，改为进程到进程的请求边



    - 等待图中出现环则存在死锁O(n2), n为节点个数
  - 多实例
    - 与银行家算法相比：**没有设置每个进程所能申请的资源最大数量。**为什么？因为是检测而不是预防
  - 何时调用检测算法：1. 死锁发生的频率；2. 有多少进程受影响
- 死锁恢复
  - 人工恢复
  - 进程中止
  - 

  - 资源抢占
  - 

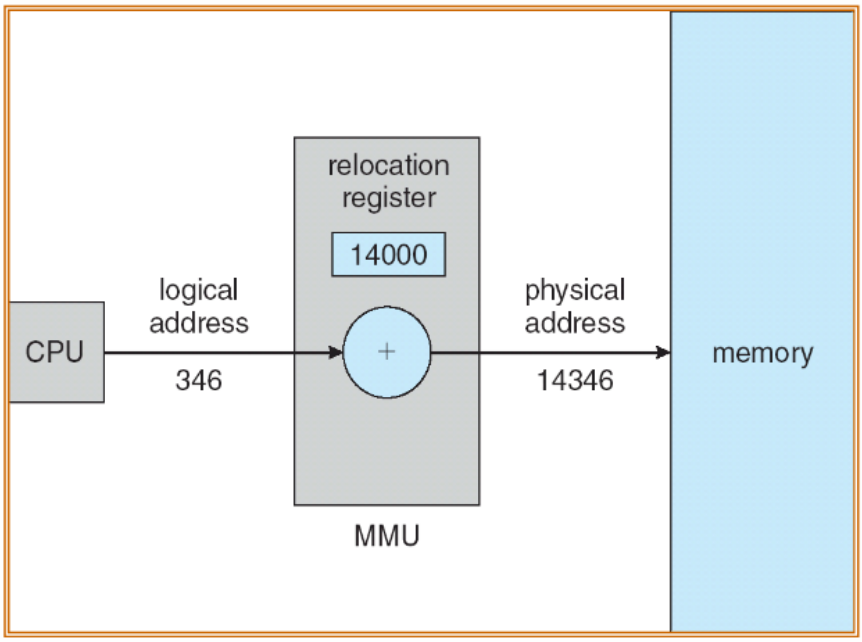
- 总结
  - 本章讲解了死锁的4个必要条件以及如何通过破坏其中的条件来预防或者解除死锁
  - 4个必要条件包括：互斥、占有并等待、非抢占、循环等待
  - 死锁的处理方法包括：死锁预防、死锁避免(资源分配图法与银行家算法)、死锁检测(单实例与多实例检测算法)、死锁恢复(进程终止、资源抢占)
  - 问题：

    - 死锁预防与避免有什么区别？ 死锁避免(确保循环等待条件不成立)是死锁预防的子集
    - 死锁避免与死锁检测区别？ 死锁避免与死锁预防都是为了系统中不会出现死锁，而死锁避免是出现死锁后检测出来并应用死锁恢复算法恢复
  - 进程管理部分包括：进程、线程、cpu调度、进程同步、死锁

8.内存管理

- 背景：为了提高cpu的使用率以及计算机的交互能力有了之前几个章节的进程管理，但是进程是存放在内存中的，因而需要多种内存管理方法
- 为什么需要cache：cup通常可以在一个cpu时钟周期内解析并执行指令，完成内存访问需要多个cpu周期，cpu通常需要暂停(读取数据时，cpu空闲)。cache协调了速度差异
- 如何保证进程之间不会相互干扰(内存访问越界)：基址寄存器(base，**进程最小的合法物理内存地址**)+界限寄存器(limit：**进程地址的长度**)，cpu在执行指令时候需要进行合法性检验
- 地址绑定：在程序装入内存时，把程序中的相对地址转化为绝对地址的过程，可以在三个不同阶段
  1. 编译时绑定：MS-DOS的.COM格式程序
  2. 加载时绑定
  3. **执行时绑定：进程在执行时可以从一个内存段移动到另一个内存段，那么绑定必须延迟到执行时进行，绝大多数操作系统采用这种方法**
  4. 编译与加载时绑定生成相同的逻辑地址与物理地址，执行时绑定不同
- 逻辑地址空间与物理地址空间
  1. 逻辑地址(虚拟地址)：cpu生成的地址
  2. 物理地址(绝对地址)：内存单元所看到的地址
  3. 逻辑地址空间：程序所生成的所有逻辑地址集合
  4. 虚拟地址空间：逻辑地址对应的物理地址集合
- 内存管理单元：MMU，书上使用了一种简单的MMU方案即重定位寄存器来实现虚拟地址到物理地址的映射

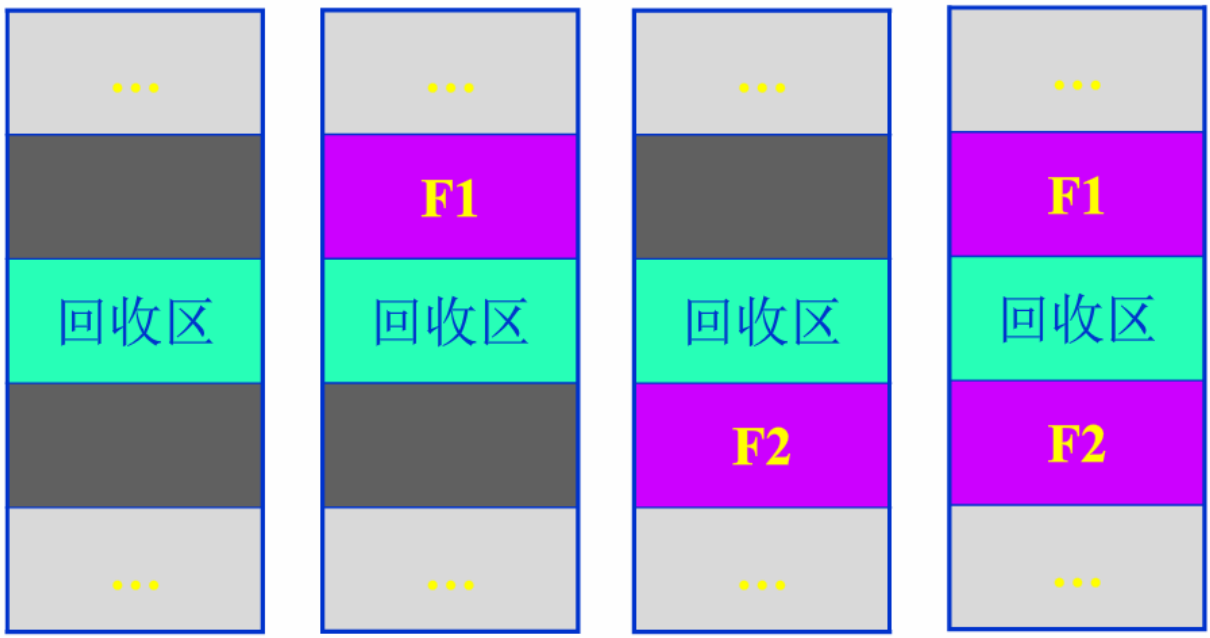
使用重定位寄存器的动态重定位



- 用户处理逻辑地址，内存映射硬件将逻辑地址转化为物理地址
- 动态加载：之前讨论的是进程的程序与代码都必须位于内存中，因此进程的大小会受物理内存大小的限制，动态记载可以获得更好的空间利用率，没有被用到子程序不会被调入内存，如果大量代码是用来处理错误的，这将非常有用，而且无需操作系统支持
- 动态链接：各种库文件的链接被推迟到执行时期，需要操作系统支持
- 连续内存分配：为每一个用户程序分配 一个连续的内存空间，早期的内存分配模式，运用于内存较少的系统
  1. 单一连续分配：整个用户的内存空间由一个程序独占
  2. 固定分区分配：预先把内存空间分为多个分区(大小可以不同)，每个分区装一个且只能装一个程序
  3. 可变分区分配：分区称为孔，操作系统包含已分配的分区表和空闲分区表，当有新进程需要内存时，为该进程查找足够大的孔，如果找到则分配内存，进程执行完毕后回收占用的孔，如果新孔与旧孔相邻可以合并成大孔。现在的问题是如何从一组可用孔中选择一个空闲孔。
- 存储分配算法
  - 首次适应
  - 最佳适应
  - 最差适应：寻找最大的分区进行分配
- 内存回收

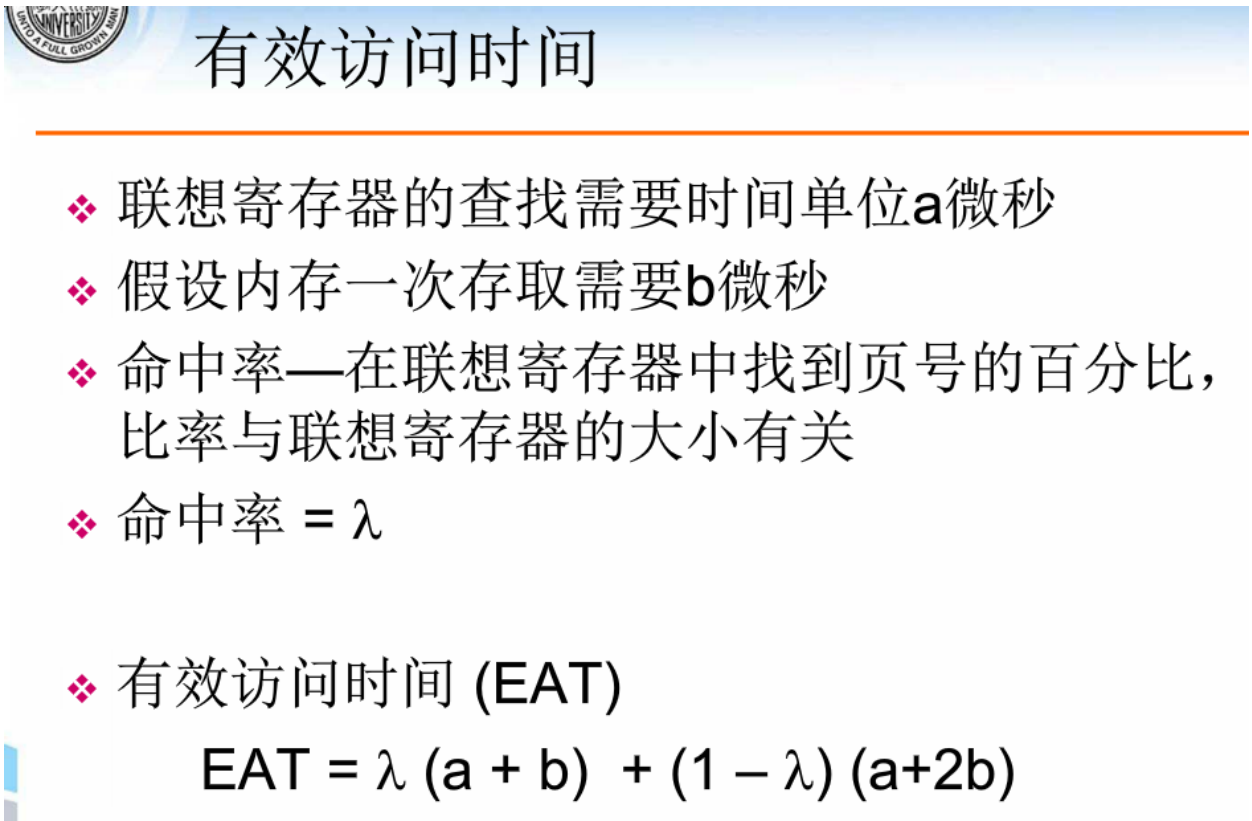
内存回收

- 存在4种情况
  - a. 回收内存块前后无空闲块
  - b. 回收内存块前有后无空闲块
  - c. 回收内存块前无后有空闲块
  - d. 回收内存块前后均有空闲块



- 碎片：
  - 外碎片：整个可用内存空间可以用来满足一个请求，但它不是连续的
  - 内碎片：分配的内存比申请的内存大，两者之差称为内部碎片
  - 解决外碎片的办法：
    - 紧缩：把一些小空闲块结合成大的空闲块，只适用于重定位是动态的，开销可能较大
    - 允许物理地址空间非连续，即分页/分段
- 分页

- 首先把物理内存分为大小固定的块，称为帧，大小为2的幂，现在4k-64k，早期512字节-8k
- **其次把逻辑内存也分为同样大小的块，称之为页**
- 系统保留所有空闲帧的记录，运行N页大小的程序，需要N个空闲帧来装入程序
- **建立一个页表，把逻辑地址转换为物理地址**
- **不会产生外碎片，但缺点是存在内碎片**
- 逻辑地址：页号+页偏移，假设页号为m位，页偏移为n位，则物理地址这样获得：首先查看页表获得物理页为p，物理地址=p×页大小+页偏移;
- P248最后一行是如何计算的？
-
  - 页表被保存在主存中
  - 页表基址寄存器(PTBR)指向页表
  - 页表限长寄存器(PRLR)表明页表的长度
  - 在这个机制中，每一次的数据/指令存取需要两次内存存取，一次是存取页表，一次是存取数据/指令
  - 解决两次存取的问题，是采用小但专用且快速的硬件缓冲，这种缓冲称为转换表缓冲器(TLB)或联想寄存器
-



- 帧表：操作系统需要知道物理内存的分配情况：哪些帧已经被占用，哪些帧没有被占用，这些信息被保存在帧表中
- pcb中保存页表的副本，当一个进程分配给cpu时，cpu需要根据该副本来定义硬件页表，因此分页增加了切换时间。
- 分页下的内存保护
  - 通过与每个帧相关联的保护位来实现，这些位通常保存在页表中。
- 页共享

- 页表结构

页表结构

- 例子：
  - 32位逻辑地址、页大小4KB
  - 一个页表最多可包含1百万个表项（$2^{32}/2^{12}$）
  - 每个页表项4个字节，需4MB空间放页表，1024个连续页面
  - 需要这么多个连续页面来存放页表不一定能实现
- 解决方法：
  - 层次页表
  - 哈希页表
  - 反向页表

- 分段
  - 分页的问题：用户视角的内存与实际物理内存相分离
  - 一个程序是一些段的集合
  - 分段逻辑地址：段号+段偏移
  - **有外碎片**
- 段页式管理：
  - 分段和分页原理的结合
  - 先将用户程序分成若干个段，再把每个段分成若干个页，并为每个段赋予一个段号
  - 逻辑地址：<段号，页号，页内偏移>

  - 存在内碎片
  - 无外碎片
  - 存在内碎片、无外碎片

- 内存扩充技术
  1. 进缩(可变分区)
  2. 覆盖技术
     - 解决问题：程序大小超过物理内存大小
     - 程序的不同部分在内存中相互替换
     - 由程序员声明覆盖结构，不需要操作系统的特别支持，较为复杂，早期系统使用
  3. 交换技术
     - 一个进程可以暂时被交换到内存外的一个备份区，随后可以被换回内存继续执行
  4. 虚拟内存
- 总结
  - 本章主要讲解了内存管理
  - 涉及逻辑地址空间与物理地址空间，连续内存分配的三种方式以及三种分配算法、分页、分段、段页式。。。