操作系统

2021年8月17日 21:00

硬件和硬件之间的接口 - usb hdmi vga等等
硬件和软件之间的接口 - 指令集
软件和软件之间的接口 - api （application program interface）

应用
---------- 虚拟机接口
操作系统OS
---------- 物理机接口-调用指令集
硬件

操作系统启动
BIOS： POST(加电自检)、寻找显卡和执行BIOS
将BOOTLOADER从磁盘的引导扇区加载
跳转到BOOTLOADER
Bootloader:
将操作系统和数据加载
跳转到操作系统

面向外设---中断
面向应用程序---系统调用、异常

系统调用、异常、中断三者区别从源头、异步同步、响应区分
系统调用（syscall 来源自应用程序）
应用程序主动想操作系统发出服务请求
异步或同步事件
响应方式：等待或者持续
异常（exception 来源于不良的应用程序
非法指令或者其他坏的处理状态（如：内存出错），是应用程序意想不到的行为
同步事件
响应方式：杀死或者重新执行意想不到的应用程序指令
中断（interrupt 来源于外设）
来自不同的硬件设备的定时器和网络中断
异步事件
响应方式：是持续的，对用户应用程式是透明的

中断产生及处理：中断标记
硬件：1、将内部、外部事件设置中断标记
2、中断事件的ID
软件：1、保存当前处理状态
2、中断服务程序处理
3、清楚中断标记
4、恢复之前保存的处理状态

异常产生及处理：异常编号
- 保存现场
- 异常处理
  - 杀死产生了异常的程序
  - 重新执行异常指令
- 恢复现场

系统调用
用户态：无法直接访问IO，无法直接执行一些指令
内核态：操作系统能够执行任何指令

系统调用时会切换堆栈 从用户态到内核态

系统调用跨越操作系统边界的开销

- 在执行时间上的开销超过程序调用
- 开销：
  - 简历中断/异常/系统调用号与对应服务例程映射关系的初始化开销
  - 建立内核堆栈
  - 验证参数
  - 内核态映射到用户态的地址空间 更新页面映射权限
  - 内核态独立地址孔空间 TLB

计算机体系结构/内存分层体系

- CPU-内存-I/O设备 通过总线相连，其中cpu包含运算器、寄存器、控制器、缓存（Cache）和存储管理单元（MMU）
- 通过CPU访问存储，寄存器、缓存、主存（硬盘）、磁盘（虚拟内存），逐渐变慢
- 操作系统所要完成的目标
  - 抽象：逻辑地址空间
  - 保护：独立地址空间，隔离各个进程
  - 共享：访问相同内存，进程间能够数据传递
  - 虚拟化：更多的地址空间，硬盘上虚拟内存

地址空间&地址生成

- 地址空间定义
  - 物理地址空间：硬件支持的地址空间
  - 逻辑地址空间：一个运行的程序所拥有的的内存范围
- 地址生成

  物理地址生成

  1.运算器需要再逻辑地址的内存内容

  2.内存管理单元寻找在逻辑地址和物理地址之间的映射

  3.控制器从总线发送在物理地址的内存内容的请求

  4.内存发送物理地址内存内容给CPU

  操作系统建立逻辑地址和物理地址之间的映射。
- 地址安全检查

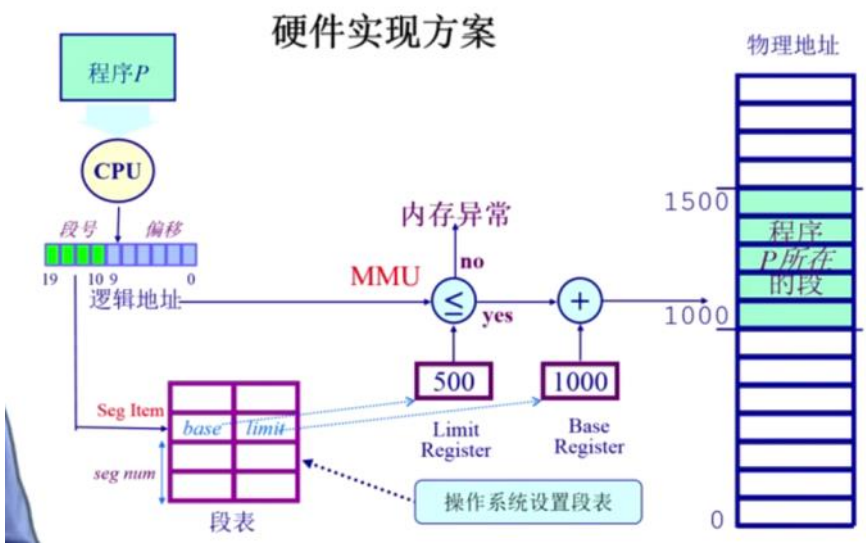
  操作系统设置了地址空间的基址和界限

连续内存分配

- 内存碎片问题：空闲内存不能被利用
  - 外部碎片：在分配单元间的未使用内存
  - 内部碎片：在分配单元中的未使用的内存
- 分区的动态分配
  - 首次适配：为了分配n字节，使用第一个可用的大于n的空闲块
    - 需求：1.按地址排序的空闲块列表 2.分配需要找一个合适的分区 3.重分配要检查能否合并相邻的空闲
    - 优势：1、简单 2、容易产生更大空闲块，
    - 劣势：1.容易产生外部碎片 2.不确定性
  - 最佳适配：找一个与n最接近的空闲块
    - 为了避免分割大的空闲块
    - 为了最小化外部碎片产生的只存
    - 需求:1.按尺寸排列的空闲块列表 2.分配需要找一个合适的分区 3重分配需要检查合并相邻空闲分区
    - 优势：1、当大部分分配是小尺寸时非常有效 2、比较简单
    - 劣势：1.外部碎片比较细 2.重分配慢 3.易产生很多微小碎片
  - 最差适配：分配n字节，找一个与n差最多的空闲块
    - 为了避免有太多微小的碎片

        - 需求：1.按尺寸排列的空闲块列表 2.分配很快，直接找最大 3.重分配需要合并
        - 优势：加入分配是中等尺寸效果最好
        - 劣势：1、重分配慢 2、外部碎片 3、容易分割大空闲块导致大的分区无法分配
- 压缩式碎片整理
    - 重置程序以合并孔洞
    - 要求所有程序是 动态可重置的
        - 何时重置?
        - 开销
- 交换式碎片整理
    - 运行程序需要更多的内存
    - 抢占等待的程序&回收他们的内存（将等待中的程序从内存放入虚拟内存中）

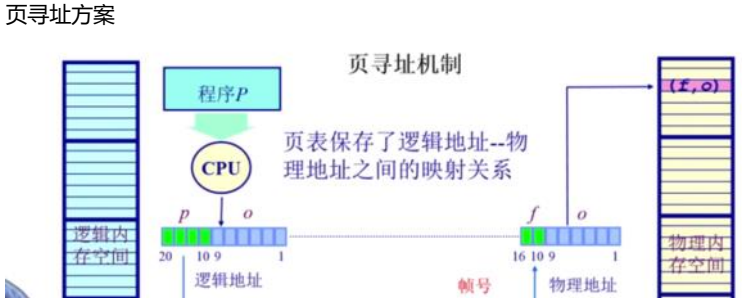
物理内存的非连续内存分配
非连续分配的优点：1、物理地址空间非连续 2、更好的内存利用和管理 3、允许共享代码和数据 4、支持动态加载和动态链接
非连续分配的缺点：如何建立虚拟地址和物理地址之间的转换：软件方案或硬件方案
两种硬件方案：分段、分页

- 分段
    - 程序的分段地址空间
        - 逻辑地址空间上连续（如堆、运行栈、程序数据、程序文本等）的，分散到多个物理地址空间（即堆、运行栈、程序数据、程序文本等按块分散）
    - 分段寻址方案
        - 段访问机制：程序的逻辑地址 去访问 内存地址需要一个二元组(s,addr)
            - s-段号
            - addr-段内偏移
    - 

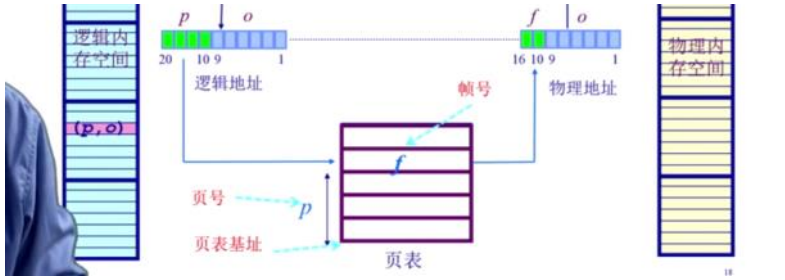
- 分页（与分段的区别：段的偏移大小是可变的，页的偏移是固定的）
    - 分页地址空间
        - 划分物理内存至乏大小的帧
            - 大小是2的幂
        - 划分逻辑地址空间至相同大小的页
        - 建立方案 转换逻辑地址为物理地址（pages to frames）
            - 页表
            - MMU(内存管理单元)/TLB(快表)

帧：物理内存被分割为大小相等的帧

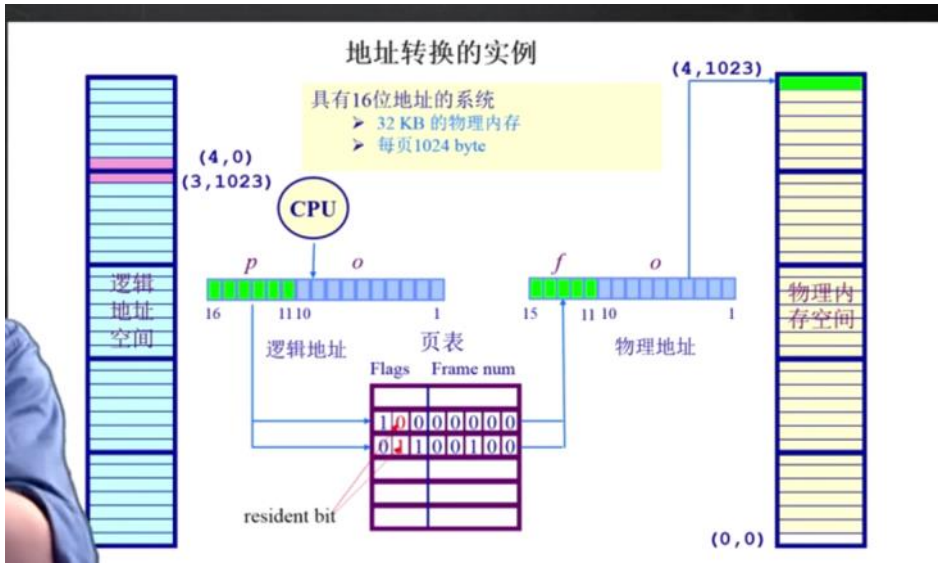
一个内存物理地址是一个二元组 (f,o)
f-帧号(F位，共有2^F个帧)
o-帧内偏移(S位，每帧有2^S字节)
物理地址= 2^S*f + o

    - 页寻址方案
    - 

页：一个程序的逻辑地址空间被划分为大小相等的页
页内偏移的大小=帧内偏移的大小
页号大小<>帧号大小
一个逻辑地址是一个二元组（p，o）
p-页号（P位，2^P个页）

- 

  一个逻辑地址是一个二元组（p，o）
  p-页号（P位，2^P个页）
  o-页内偏移（S位，每页有2^S字节）
  虚拟地址=2^Sσp + o
  - 页映射到帧
  - 页是连续的虚拟内存
  - 帧是非连续的物理内存
  - 不是所有的页都有对应的帧（一般来说逻辑地址空间大于物理地址空间）
- 页表

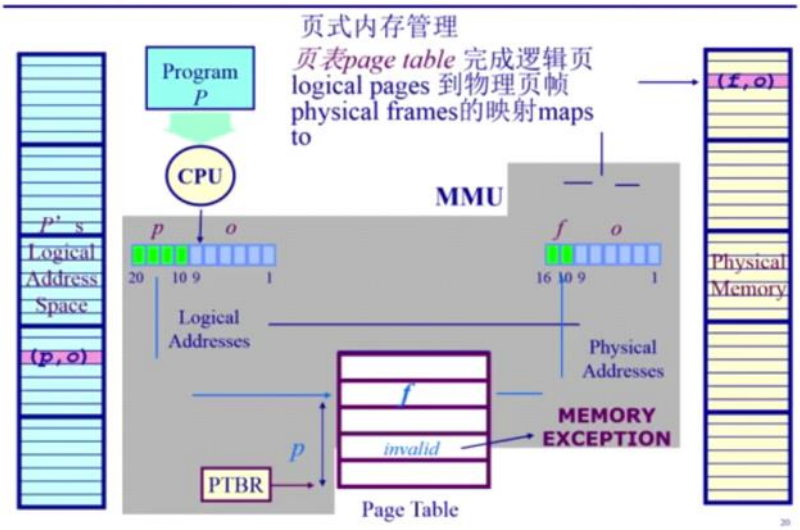


  - 页表概述：将页映射到帧
    - 每个程序都有一个页表，属于程序运行状态，会动态变化，有一个PTBR:页表基址寄存器
    - 分页机制的性能问题
      - 访问一个内存单元要两次内存访问
        - 一次用于获取页表项
        - 一次用于访问数据
      - 页表可能很大：64位机器如果每页1024字节，则需要2^64/1024=2^54字节的页表
      - 如何处理?
        - 缓存（Caching）
        - 间接（Indirection）访问
  - 转换后备缓冲区（TLB Translation Look-aside Buffer）
    - 缓存近期访问的页帧转换表项（解决访问性能问题，加快速度）
      - TLB使用associative memory（关联内存）实现，具备快速访问性能
      - 如果TLB命中，物理页号可以迅速获取
      - 如果TLB未命中，对应的表项被更新到TLB中
  - 二级/多级 页表（解决页表很大的问题）
    - 通过页号分为K个部分，来实现多级间接页表
      - 建立页表“树”
  - 反向页表
    - 基于页寄存器的方案（page registers）（物理地址映射到逻辑地址空间）
      - 利：1、转换表大小很小 2、转换表大小和逻辑地址空间大小无关
      - 弊：1、需要的信息对调了，能根据帧号查到页号 2、如何从页号查到帧号成了问题 3、要在反向页表中搜索想要的页号
    - 基于关联内存的方案
    - 基于哈希的查找方案
      - 存在哈希碰撞
      - 为了查找页需要从内存中取数，进行哈希函数计算，内存开销大

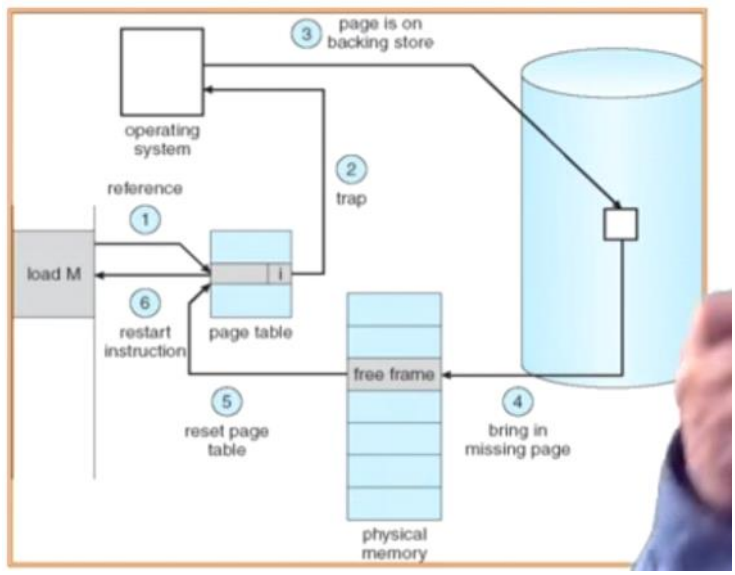
虚拟内存

- 起因
- 覆盖技术：程序太大，超过了内存容量，可以采用手动的覆盖技术，只需要把需要的指令和数据保存在内存中
  - 目标：在较小的可用内存中运行较大的程序。冲用于多道程序系统，与分区存储管理配合使用。
  - 原理：吧程序按照其自身逻辑结构，划分为若干个功能上相对独立的程序模块，那些不会同时执行的模块共享同一块内存区域，按时间先后来运行。
    - 必要部分（常用功能）的代码和数据常驻内存。
    - 可选部分（不常用功能）在其他程序模块中实现，平时存放在外存中，在需要用到时才装入内存
    - 不存在调用关系的模块不必同时装入内存，从而可以相互覆盖，即这些模块共用一个分区
  - 缺点：
    - 程序猿需要把大程序划分为小的功能块，并确定各自的覆盖关系，编程难度大
    - 覆盖模块从外存装入内存，耗费时间，是用时间换空间
- 交换技术：程序太多，超过了内存容量，可以采用自动的交换技术，把暂时不执行的程序送到外存
  - 目标：多个程序运行时，让正在运行或需要运行的程序获得更多的内存资源
  - 方法：
    - 将暂时不能运行的程序送到外存，从而获得空闲内存
    - 操作系统把一整个进程的整个地址空间的内容保存带外存（swap out），而将外存中的某个进程的地址空间读入内存（swap in）。换入换出内容的大小为整个程序的地址空间。
    - 交换技术实现中的问题：
      - 交换时机的确定：只有当内存空间不够或有不够的危险时换出
      - 交换区的大小：必须足够大的交换能存放所有内存映像的拷贝；必须能对这些内存映像进行直接存取
      - 程序换入时的重定位：换出后再换入的内存位置一定要在原来的位置上吗？最好采用动态地址映射的方法
  - 缺点：程序整个地址空间的换入换出，对处理器来说开销太大
- 覆盖技术和交换技术的比较
  - 覆盖针对一个程序，对程序分块，因此需要程序猿给出程序内的各个模块之间的逻辑覆盖结构
  - 交换针对多个程序，不需要程序猿给出各个模块的逻辑结构。
  - 交换发生在内存中的程序与操作系统之间；覆盖发生在运行程序的内部
- 虚存技术：在有限容量的内存中，以更小的页粒度为单位装入更多更大的程序，可以采用自动的虚拟存储技术
  - 目标：像覆盖技术一样，不把程序的所有内容放在内存中；像交换技术一样，能够实现进程在内存与外存之间交换
  - 程序局部性原理：程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一个区域
    - 时间局部性：一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短的时间内
    - 空间局部性：当前指令和邻近的几条指令，当前访问的数据和邻近的几个数据都集中在一个较小的区域内
  - 基本概念：可在页式或段式内存管理的基础上实现
    - 在装入程序时，只放入当前需要执行的部分页面或段到内存，就能让程序开始运行
    - 在程序运行过程中，如果需要执行的指令或访问的数据尚未在内存中（称为缺页或缺段），则有处理器通知操作系统将相应的页面或段调入内存，然后继续执行
    - 另一方面，操作系统将内存中暂时不用的页面或段调出到外存中，以腾出更多的空闲内存
  - 基本特征
    - 大的用户空间：将物理内存和外存相结合，提供的虚拟内存空间通常大于实际的物理内存，实现了两者的分离
    - 部分交换：与交换技术相比，虚拟内存的交换式对部分虚拟地址空间进行的
    - 不连续性：物理内存分配的不连续，虚拟地址空间使用的不连续
  - 虚拟页式内存管理

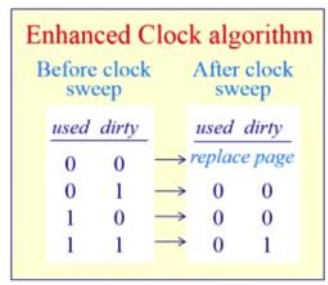
- 虚存技术--虚拟页式内存管理



  - 在页式存储管理的基础上，增加请求调页和页面置换功能
  - 基本思路
    - 当用户程序要调入内存运行时，不是将该程序的所有页面都装入内存，值装入部分页面就能运行程序
    - 运行过程中，如果发现要运行的程序或者访问数据不再内存，则想系统发出缺页中断请求，系统在处理中断时，将外存中相应的页面调入内存，使该程序能够继续运行
    - 页表表项：
      - 1.原本就有的页号、物理页帧号等
      - 2.加入了驻留位：表示该页在内存还是外存。1为内存，可以直接使用，0为外存，需要缺页中断中进行读取
      - 3.加入了保护位：表示能做什么访问，如只读、可读写、可执行等
      - 4.修改位：表示此页是够被修改过。当系统回收该物理页面时，根据此位来决定是否把他的内容写回外存。1为改过0为未改
      - 5.访问位：1为访问过0为未访问过。对于不常访问的页可以方便页面置换
    - 缺页中断处理过程



      - 1、如果在内存中有空闲的物理页面，则分配一物理页帧f，然后转4；否则转2
      - 2、采用某种页面置换算法，选择一个将被替换的物理页帧f，它所对应的逻辑页为q。如果该页在内存器件被修改过，则需要把它写回外存。
      - 3.对q所对应的页表项进行修改，驻留位置0
      - 4.将需要访问的页p装复物理页帧f中
      - 5.修改p所对应的页表项的内容，吧驻留位置1，吧物理页帧号置为f
      - 6.重新运行被中断的指令
    - 在何处保存未被映射的页?
      - 能够简单识别在二级存储器中的页
      - 交换空间：特殊格式，用于存储未被映射的页面
    - 后备存储（backing store）概念：
      - 一个虚拟地址空间的页面可以被映射到一个文件（在二级存储中）中的某个位置
      - 代码段：映射到可执行二进制文件
      - 动态加载的共享库程序段：映射到动态调用的库文件
      - 其他段：可能被映射到交换文件（swap file）
  - 虚拟内存性能：使用有效存储器访问时间effective memory access time（EAT）
    - EAT=访存时间*页表命中几率 + page fault处理时间 * page fault几率
      - 其中访存时间为访问内存的时间 page fault处理时间大概为磁盘访问时间

- 页面置换算法
  - 功能与目标
    - 功能：当缺页中断发生，需要调入新的页面而内存已满，选择内存当中哪个物理页面被置换
    - 目标：尽可能减少页面的换进换出的次数
    - 页面锁定：用于需要常驻内存的操作系统的关键部分。有一个锁定标志位。
  - 实验设置与评价方法
  - 局部页面置换算法
    - 最优页面置换算法（OPT,optimal）
      - 基本思路：当缺页中断发生时，对于内存中的每个逻辑页面，计算下一次访问需要等待多久。从中选择等待时间最长的那个，作为被置换的页面。
      - 这是一种理想情况，实际无法实现，因为操作系统无法知道每个页面下次多久被访问。
    - 先进先出算法（FIFO）
      - 基本思路：选择在内存中驻留时间最长的页面置换他。系统维护了一个链表，头部为最早进入内存的，尾部为最后进入内存的
      - 性能较差，调出的页面可能是经常要访问的页面，并且有Belady现象，FIFO算法很少单独使用
    - 最近最久未使用算法（LRU，Least Recently Used）
      - 基本思路：选择内存中最久未使用的页面，进行置换。
      - 实现方法：
        - 方案1.维护一个页面链表，最近使用的页面作为首节点，最久未使用的页面作为尾结点。每次访问内存，找到相应的页面，把他放到链表首。缺页中断发生时，淘汰链表末尾的页面。
        - 方案2.设置一个活动页面栈，当访问某页时，将此页号压入栈顶，然后考察栈内是否有与此页号相同的页号，将其抽出。当要淘汰一个页面时，总是选择栈底的页面，他就是最久未使用的。
    - 时钟页面置换算法（Clock）
      - 基本思路
        - 需要用到页表项中的访问位，当一个页面被装入内存时，把该位初始化为0.如果这个页面被访问，则置1.
        - 把各个页面组织成环形链表，把指针指向最老的页面（最先进来的）
        - 当缺页中断发生，考察指针所指向的最老的页面，如果访问位为0，则立即淘汰；如果访问位为1，则该位置0，然后移下一格。如此下去，直到找到被淘汰的页面，然后指针移到下一格
      - Dirty bit位（脏位）就是写位，进行算法优化（也叫二次机会法）
        - 这里有一个巨大的代价来替换“脏”页
        - 修改clock算法，使他允许脏页总是在一次时钟头扫描中保留下来
          - 同时使用脏位和使用位来指导置换
        - 原因：如果该页被写了，则需要更新硬盘中的这一页，开销比较大；如果该页未被写，则硬盘上存储的和内存中一致，直接释放即可
        - 

    - 最不常用算法（LFU, Least Frequently Used）
      - 基本思路：当缺页中断发生时，选择访问次数最少的那个页面，并将其淘汰
      - 实现方法：对每个页面设置一个访问计数器，每当一个页面被访问时，该页面的访问计数器加1。在发生缺页中断时，淘汰计数值最小的那个页面。
      - LFU和LRU一个是访问频率，一个是上一次访问的时间。可以对LFU优化，对于一段时间没有访问的页面，将其访问计数器变少（如右移一位）
    - Belady现象：在采用FIFO算法时，有时会出现分配的物理页面数增加，缺页率反而提高的异常现象。
    - LRU、FIFO和clock的比较
      - LRU FIFO本质上都是先进先出的思路，只不过LRU是针对页面最近的访问时间排序，FIFO是首次进入内存的时间排序。如果内存中所有页面都未曾访问过，那么LRU就会退化为FIFO。 所以重点还是算法具有局部性
      - LRU算法性能好，但是开销大;FIFO算法开销小，但是性能差。clock算法是两者的折中
  - 全局页面置换算法
    - 局部页替换算法的问题

        - 前面的各种页面置换算法，都是基于一个前提，即程序的局部性原理。
        - 如果局部性原理不成立，算法会退化
        - 如果局部性原理是成立的，那么如何证明它的存在，如何对他进行定量分析？这就是工作集模型
    - 工作集模型：一个进程当前正在使用的逻辑页面集合，可以用一个二元函数W(t,△)表示
        - t是当前的执行时刻
        - △称为工作集窗口，即一个定长的页面访问的时间窗口
        - W(t，△)=当前时刻t之前的△时间窗口当中的所有页面所组成的集合（随t的变化，集合也在变化）
        - |W(t，△)| 指工作集的大小，即页面数目
    - 常驻集：指在当前时刻，进程实际驻留在内存当中的页面集合。
    - 工作集页置换算法：固定窗口
        - 追踪之前t个内存访问的页引用是工作集，t被称为窗口大小
    - 缺页率置换算法：可变分配策略：常驻集大小可变。
        - 采用全局页面置换的方式，发生缺页中断时，被置换的页面可以是其他进程中，各个并发进程竞争的使用物理页面
        - 优缺点：性能较好，但增加了系统开销
        - 具体实现：可以使用缺页率算法（PFF, page fault frequency）来动态调整常驻集的大小。
        - 缺页率： 缺页次数/内存访问次数 或者 缺页平均时间间隔的倒数。 影响缺页率的因素：
            - 页面置换算法
            - 分配给进程的物理页面数目
            - 页面本身的大小
            - 程序的编写方法
        - 算法：保持追踪缺页发生的概率（时间窗口 最近访问页窗口）
            - 当缺页发生时，从上次页缺失起计算这个时间记录这个时间，t last是上次缺页发生的时间
            - 如果发生缺页之间的时间是“大”的，要减少工作集
            - 如果t current - t last > T，之后从内存中移除所有在[t last，t current]时间内没有被应用的页
            - 如果这个缺页发生的时间是“小”的，要增加工作集
            - 如果t current - t last<T，之后增加缺失页到工作集中
    - 抖动问题
        - 定义：如果分配给一个进程的物理页面太少，不能包含整个的工作集，即常驻集<工作集，那么进程将会造成很多缺页中断，需要频繁的页面置换，从而使进程的运行速度变得很慢，我们把这种状态称为“抖动”
        - 抖动产生的原因：随着驻留内存的进程数目增加，分配给每个进程的物理页面不断减小，缺页率不断上升。所以OS需要选择一个适当的进程数目和进程需要的帧数，以便在并发水平和缺页率之间达到一个平衡。
        - 解决方案：用更好的规则加载控制：调整MPL
            - 使平均页缺失时间（MTBF mean time between page faults）= 页缺失服务时间 （PFST page fault service time）
            - ∑WSi = 内存的大小

进程管理

- 进程（progress）描述
    - 进程定义：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程
    - 进程的组成
        - 程序的代码
        - 程序处理的数据
        - 程序计数器中的值，指示下一条将运行的指令
        - 一组通用寄存器当前的值，堆、栈
        - 一组系统资源（如打开的文件等）

        总之，进程包含了在运行的一个程序的所有状态信息。
    - 进程的特点
        - 动态性：可动态的创建、结束进程
        - 并发性：进程可以被独立调度并占用处理器运行；并发并行
            - 并发：在一段时间内有多个进程在执行
            - 并行：在一个时刻有多个时刻在运行
        - 独立性：不同进程的工作不互相影响（页表实现）
        - 制约性：因访问共享数据/资源或进程间同步而产生制约
        - 描述进程的数据结构：进程控制块（Process Control Block ,PCB）
            - 操作系统为每个进程都维护了一个PCB，用来保存与该进程有关的各种状态信息
    - 进程控制结构

        - 进程控制块：操作系统管理控制进程运行所用的信息集合
            - 操作系统用PCB来描述进程的基本情况以及运行变化的过程，PCB是进程存在的唯一标志
            - PCB包含的三大信息
                - 1、进程标识信息。如本进程的标识，本进程的父进程标识，用户标识
                - 2、处理器状态信息保存区。保存进程的运行现场信息：
                    - 用户可见寄存器，用户程序可以使用的数据，地址等寄存器
                    - 控制和状态寄存器，如程序计数器（PC），程序状态字（PSW）
                    - 栈指针，过程调用/系统调用/中断处理和返回时需要用到它
                - 3、进程控制信息
                    - 调度和状态信息，用于操作系统调度进程并占用处理器使用
                    - 进程间通信信息，为支持进程间的通信相关的各种标识、信号、信件等，这些信息存在接受方的进程控制块中
                    - 存储管理信息，包含有指向本进程映像存储空间的数据结构
                    - 进程所用资源，说明由进程打开、使用的系统资源，如打开的文件等
                    - 有关数据结构连接信息，进程可以连接到一个进程队列中，或连接到相关的其他进程的PCB
            - PCB的组织方式
                - 链表：同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
                    - 各状态的进程形成不同的链表：就绪链表、阻塞链表
                - 索引表：同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表
                    - 各状态的进程形成不同的索引表：就绪索引表、阻塞索引表
- 进程状态（state）
    - 进程的生命周期
        - 进程创建

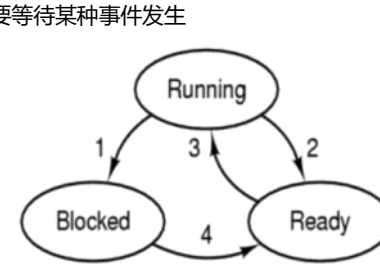
          引起进程创建的3个主要事件
            - 系统初始化时
            - 用户请求创建一个新进程
            - 正在运行的进程执行创建进程的系统调用
        - 进程运行
            - 内核选择一个就绪的进程，让它占用处理器并执行
                - 为何选择?
                - 如何选择?
        - 进程等待（阻塞）

          在以下情况，进程等待（阻塞）
            - 1.请求并等待系统服务，无法马上完成
            - 2.启动某种操作，无法马上完成
            - 3.需要的数据没有到达

          进程只能自己阻塞自己，因为只有进程自身才能知道何时需要等待某种事件发生
        - 进程唤醒

          唤醒进程的原因
            - 1.被阻塞进程需要的资源可被满足
            - 2.被阻塞进程等待的事件到达
            - 3.将该进程的PCB插入到就绪队列

          进程只能被别的进程或操作系统唤醒

        - 进程结束

          在一下四种情形下，进程结束：
            - 1.正常退出（自愿的）
            - 2.错误退出（自愿的）
            - 3.致命错误（强制性的）
            - 4.被其他进程说啥（强制性）

    - 进程状态变化模型
        - 进程的三种基本状态：
            - 运行状态（running）：进程在处理器上运行
            - 就绪状态（ready）：一个进程获得了除处理器之外的一切所需资源，一旦得到处理器即可运行
            - 等待状态（又称阻塞状态blocked）：一个进程在等待某个时间而暂停运行。如等待资源，等待输入/输出完成
        - 进程的其他的基本状态：

                - 创建状态（new）：一个进程正在被创建，还没被转到就绪状态之前的状态
                - 结束状态（exit）：一个进程正在从系统中消失时的状态，这是因为进程结束或其他原因所导致
        - 进程挂起模型
            - 进程在挂起状态时，意味着进程没有占用内存空间。处在挂起状态的进程映像在磁盘上。
            - 阻塞挂起状态：进程在外存并等待某事件出现
            - 就绪挂起状态：进程在外存，但只要进入内存，就能立即运行
            - 挂起（suspend）：把一个进程从内存转到外存；可能有一下几种情况：
                - 阻塞->阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，会进行这种转换，以提交新进程或运行就绪进程
                - 就绪->就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先就绪进程时，系统会选择挂起低优先级就绪进程
                - 运行->就绪挂起：对抢先式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态
                - 在外存时的状态转换
                    - 阻塞挂起->就绪挂起：当有阻塞挂起进程因相关事件出现时，系统会把阻塞挂起进程转换为就绪挂起进程
            - 解挂/激活（activate）：把一个进程从外存转到内存；可能有一下几种情况
                - 就绪挂起->就绪：没有就绪进程或挂起就绪进程优先级高于就绪进程时，会进程这种转换
                - 阻塞挂起->阻塞：当一个进程释放足够内存时，系统会把一个高优先级阻塞挂起（系统认为会很快出现等待时间）进程转换为阻塞进程
            - 状态队列：对于不同状态的进程有不同的状态队列，如就绪队列、阻塞队列等，队列中为进程的PCB
- 线程（thread）：进程当中的一条执行流程
    - 为什么使用线程?
        - 单线运行会因为一些步骤阻塞，导致能快速运行的步骤也被阻塞。
    - 特性：1.线程之间可以并发执行 2.线程之间共享相同的地址空间
    - 优点：
        - 一个进程中可以同时存在多个线程
        - 各个线程之间可以并发地执行
        - 各个线程之间可以通过共享地址空间和文件等资源
    - 缺点：一个线程崩溃，会导致其所属进程的所有线程崩溃。
    - 进程与线程的比较
        - 进程是资源分配单位，线程是CPU调度单位
        - 进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈
        - 线程同样具有就绪、阻塞和执行三种基本状态，同样有状态之间的转换状态
        - 线程能减少并发执行的时间和空间开销
            - 线程的创建时间比进程短
            - 线程的终止时间比进程短
            - 同一进程内的线程切换时间比进程短
            - 由于同一进程的各线程间同向内存和文件资源，可直接进行不通过内核的通信
    - 线程的实现
        - 线程分类
            - 用户线程：在用户空间实现 POSIX Pthreads,Mach C-threads, Solaris threads
                - 在用户空间实现的线程机制，它不依赖于操作系统的内核，由一组用户级的线程库函数来完成线程的管理，包括进程的创建、终止、同步和调度
                    - 由于用户线程的维护由相应进程来完成（线程库函数），不需要操作系统内核了解用户线程的存在，可用于不支持线程技术的多进程操作系统。
                    - 每个进程都需要它自己私有的线程控制块（Thread Control Block TCB）列表，用来跟踪记录它的各个线程的状态信息（PC,栈指针，寄存器），TCB由线程库函数来维护
                    - 用户线程的切换也是由线程库函数来完成，无需用户态/核心态切换，所以速度特别快
                    - 允许每个进程拥有自定义的线程调度算法
                - 用户线程缺点：
                    - 阻塞性的系统调用如何实现？如果一个线程发起系统调用而阻塞，则整个进程在等待
                    - 当一个线程开始运行后，除非它主动交出CPU的使用权，否则所在的进程中的其他线程将无法运行
                    - 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会较慢
            - 内核线程：在内核中实现 Windows, Solaris, Linux
                - 指在操作系统的内核当中实现的一种线程机制，由操作系统的内核来完成线程的创建、终止和管理

- ◇ 在支持内核线程的操作系统中，由内核来维护进程和线程的上下文信息（PCB和TCB）
- ◇ 线程的创建、终止和切换都是通过系统调用/内核函数的方式进行，由内核完成，因此系统开销大（需要用户态和内核态切换）
- ◇ 在一个进程当中，如果某个内核线程发起系统调用被阻塞，并不会影响其他内核线程的运行
- ◇ 时间片分配给线程，多线程的进程获得更多CPU时间

□ 轻量级进程：内内核中实现，支持用户线程 Solaris Linux

◆ 它是内核支持的用户线程。一个进程可有一个或多个轻量级进程，每个量级进程由一个单独的内核线程来支持

▪ 用户线程和内核线程的对应关系

□ 多对1

□ 1对1

□ 多对多

- 进程的上下文切换：停止当前运行进程（运行转变为其他状态），并且调度其他进程（其他状态转变为运行）
  - 目标：必须在切换之前存储许多部分的进程上下文；能够在之后恢复，且不能显示他被暂停过；必须快速
  - 需要存储的上下文
    - 寄存器（PC,SP,...），CPU状态
    - 一些时候可能会费时，所以我们应该尽可能避免
- 进程控制（LINUX下）
  - 进程创建
    - Fork()创建进程
      - 对子进程分配内容
      - 复制父进程的内存和CPU寄存器到子进程里
  - 进程加载和执行
    - Exec()加载一个不同的程序(99%在调用fork后调用exec)
      - 代码，stack和heap都会被重写
    - Vfork() （虚fork，virtual fork，一些时候被称为轻量级fork）
      - 一个创建进程的系统调用，不需要创建一个同样的内存映像
      - 子进程几乎立即调用exec()
      - 现在不再使用，我们现在使用copy on write（COW）技术

  - 进程等待和终止

    wait（）系统调用时被父进程用来等待子进程的结束，让父进程帮助子进程资源回收
    - 一个子进程向父进程返回一个值，所以父进程必须接受这个值并处理
    - Wait()系统调用担任这个要求
      - 它使父进程睡眠来等待子进程的结果
      - 当一个子进程调用exit（）的时候，操作系统解锁父进程，并且将通过exit传递得到的返回值返回给wait（与子进程的pid）如果没有子进程存活，则wait立即返回
      - 当然，如果这有为父进程的僵尸等待，wait（）立即返回其中一个值（并且解除僵尸状态）
        - 僵尸状态：子进程执行完exit（），而父进程还未执行wait（）时，子进程处于僵尸状态
    - 进程结束之后，它调用exit（）
      - 将执行结果作为参数返回
      - 关闭所以打开的文件、连接等
      - 释放内存
      - 释放大部分支持进程的操作系统结构
      - 检查是否父进程是存活的
        - 如果存活，要把值保留，知道父进程需要它；而此时它并未真正死亡，处于僵尸(zombie)状态
        - 如果没有存活，则释放所有数据结构，进程死亡
      - 清理所有等待的僵尸进程
    - 进程终止是最终的垃圾收集（资源回收）
- 调度
  - 背景
    - 上下文切换
      - 切换CPU当前任务，从一个进程/线程到另一个
      - 保存当前进程/线程在PCB/TCP中的执行上下文（CPU状态）
      - 读取下一个进程/线程的上下文
    - cpu调度

        - 从就绪队列中挑选一个进程/线程作为CPU将要运行的下一个进程/线程
        - 调度程序：挑选进程/线程的内核函数（通过一些调度策略）
        - 什么时候进行调度?
            - 一个进程从运行状态切换到等待状态
            - 一个进程被终结了

          不可抢占：调度程序必须等待事件结束

          可以抢占：
            - 调度程序在中断被响应后执行
            - 当前的进程从运行切换到就绪，或者一个进程从等待切换到就绪
            - 当前运行的进程可以被换出
    - cpu调度时间
- 调度准则
    - 调度策略
    - 程序执行模型
    - 比较调度算法的准则
        - CPU利用率：CPU处于忙状态所占时间的百分比
        - 吞吐量：在单位时间内完成的进程数量
        - 周转时间：从进程初始化到结束，包括所有等待时间所花费的时间
        - 等待时间：进程在就绪队列中的总时间，就绪到运行的时间
        - 响应时间：从一个请求提交到产生第一次响应所花费的总时间
    - 吞吐量vs延迟
        - 增加吞吐量
            - 减少开销（操作系统开销，上下文切换）
            - 系统资源的高效利用（CPU，I/O设备）
        - 减少等待时间
            - 减少每个进程的等待时间
    - 公平的目标
        - 保证每个进程占用相同的CPU时间
        - 保证每个进程都等待相同的时间
- 调度算法
    - FCFS 先来先服务
        - 优点：简单
        - 缺点：
            - 平均等待时间波动较大
            - 花费时间少的任务可能排在花费时间长的任务后面
            - 可能导致I/O和CPU之间的重叠处理
    - SPN(SJF) SRT 短进程优先（短作业优先） 短剩余时间优先
        - 可抢占：又叫Shortest-Remaining-Time(SRT)最短剩余时间，当前执行任务剩余长于新加入的短时间任务
        - 缺点：
            - 可能导致饥饿
                - 连续的短任务流会使长任务饥饿
                - 段任务可用时的任何长任务的CPU时间都会增加平均等待时间
            - 需要预知未来
                - 怎么预估下一个CPU突发的持续时间
                - 简单的解决办法：询问用户
                - 如果用户欺骗就杀死进程
                - 如果用户不知道怎么办
    - HRRN 最高响应比优先
        - 在SPN调度的基础上改进
        - 不可抢占
        - 关注进程等待了多长时间
        - 防止无限期推迟
    - Round Robin 轮循-使用时间切片和抢占来轮流执行任务
        - 在时间切片的离散单元中分配处理器

        - 时间片结束时，切换到下一个准备好的进程
            - 时间片未结束进程已执行完毕则立即切换到下一进程
        - 缺点:
            - 额外的上下文切换
            - 时间片太大
                - 等待时间太长
                - 极限情况退化成FCFS
            - 时间片太小
                - 反应迅速，但是开销大
                - 吞吐量由于大量的山下文切换开销收到影响
        - 目标：
            - 选择一个合适的时间片大小
            - 经验规则：位置上下文切换开销处于1%以内
    - Multilevel Feedback Queues MLFP多级反馈队列-优先级队列中的轮循
        - 就绪队列被划分成独立的队列
            - Eg. 前台（交互），后台（批处理）
        - 每个队列拥有自己的地独立策略
            - Eg. 前台- RR轮循 后台-FCFS 先来先处理
        - 调度必须在队列间进行
            - 固定优先级
                - 先处理前台，然后勐后台
                - 可能导致饥饿
            - 时间切片
                - 每个队列都得到一个确定的能够调度其进程的CPU总时间
                - Eg. 80%给使用RR的前台，20%给使用FCFS的后台
        - 一个进程可以在不同的队列中移动
        - Eg. n级优先级----优先级调度在所有级别中，RR在每个级别中
            - 时间片大小随优先级级别增加而增加
            - 如果任务在当前时间量子中没有完成，则降到下一个优先级
        - 优点：
            - CPU密集型任务的优先级下降很快
            - I/O秘籍型任务停留在高优先级
    - Fair Share Scheduling 公平共享调度
        - 一些用户组比其他用户组更重要
        - 保证不重要的组无法垄断资源
        - 未使用的资源按照每个组所分配的资源的比例来分配
        - 没有达到资源使用率目标的组获得更高的优先级
- 实时调度
    - 实时系统：正确性依赖于其时间和功能两方面的一种操作系统
        - 性能指标
            - 时间约束的及时性（deadlines）
            - 速度和平均性能相对不重要
        - 主要特性：时间约束的可预测性
        - 分类
            - 强实时系统：需要再保证的时间内完成重要的任务，必须完成。如果不完成有灾难性或严重后果
            - 弱实时系统：需求重要的进程的优先级更高，尽量完成，并非必须
        - 表示一个实时系统能否满足deadline要求
            - 决定实时任务的顺序
                - 静态优先级调度
                - 动态优先级调度
    - 可调度性
    - 单调速率（RM）
        - 最佳静态优先级调度
        - 通过周期安排优先级

        - 周期越短优先级越高
        - 执行周期最短的任务
    - 截止日期最早优先（EDF）
        - 最佳的动态优先级调度
        - deadline越早优先级越高
        - 执行deadline最早的任务
  - 多处理器调度
    - 多个相同的单处理器组成一个多处理器
    - 优点：负载共享
    - 对称多处理器（SMP）
        - 每个处理器运行自己的调度程序
        - 需要在调度程序中同步
  - 优先级反转
    - 可以发生在任何基于优先级的可抢占的调度机制中
    - 当系统内的环境强制使高优先级任务等待低优先级任务时发生
- 进程互斥与同步
  - 临界区(critical section)：临界区是指进程中的一段需要访问共享资源，并且当另一个进程处于相应代码区域时便不会被执行的代码区域
    实现方法
    - 1.禁用硬件中断
        - 没有中断，没有上下文切换，因此没有并发
        - 进入临界区：禁用中断
        - 离开临界区：开启中断
        - 缺点：
            - 一旦中断被禁用，线程就无法被停止
                - 整个系统都会为你停下来
                - 可能使其他线程处于饥饿状态
            - 要是临界区可以任意长
                - 无法限制响应中断所需的时间（可能存在硬件影响）
    - 2.基于软件的解决办法
        - 对于两个线程，T0和T1，Ti的通常结构

          Do{

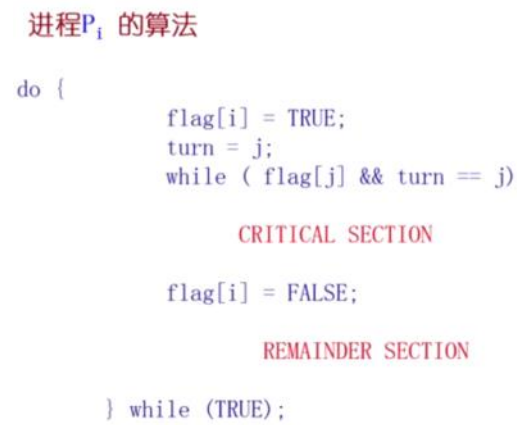
Enter section	进入区域

          临界区

Exit section	离开区域

          提醒区域

          } while (1)



        - 使用两个共享数据
            - Int turn
            - Bool flag[]
        - 进入临界区代码

          Flag[i]=true

          Turn=j

          While (flag[j] && turn==j)
        - 退出临界区代码

          Flag[i]=false
        - Dekker算法
        - Bakery算法
    - 3.更高级的抽象-原子操作
        - 硬件提供了一些原语
            - 像中断禁用，原子操作指令等
            - 大多数现代体系结构都这样
        - 操作系统提供更高级的编程抽象来简化并行编程
            - 例如：锁、信号量
            - 从硬件原语中构建
        - 锁是一个抽象的数据结构
            - 一个二进制状态（锁定/解锁），两种方法

忙等待

```
Lock::Acquire() {
  while (test-and-set(value))
    ; // spin
}

Lock::Release() {
  value = 0;
}
```

无忙等待

```
class Lock {
  int value = 0;
  WaitQueue q;
}

Lock::Acquire() {
  while (test-and-set(value)) {
    add this TCB to wait queue q;
    schedule();
  }
```

```
Lock::Release() {
  value = 0;
}
```

```
Lock::Acquire() {
  while (test-and-set(value)) {
    add this TCB to wait queue q;
    schedule();
  }
}

Lock::Release() {
  value = 0;
  remove one thread t from q;
  wakeup(t);
}
```

- 锁是一个抽象的数据结构
  - 一个二进制状态（锁定/解锁），两种方法
  - Lock::Acquire() - 锁被释放前一直等待，直到得到锁
  - Lock::Release() - 释放锁，唤醒任何等待的进程
  - 使用锁来编写临界区

```
Lock_next_pid->Acquire();
New_pid=next_pid++;
Lock-_next_pid->Release();
```

- 优点：
  - 适用于但处理器或者共享主存的多处理器中任意数量的进程
  - 简单并且容易证明
  - 可以用于支持多临界区
- 缺点：
  - 忙等待消耗处理器时间
  - 当进程离开临界区并且多个进程在等待的时候可能导致饥饿
  - 死锁
    - 如果以个低优先级的进程拥有临界区并且一个高优先级进程也需求，那么高优先级进程会获得处理器并等待临界区

- 互斥（mutual exclusion）：当一个进程处于临界区并访问共享资源时，没有其他进程会处于临界区并且访问任何相同的共享资源
  - 只有一个进程/线程在临界区中，其他进程要进入临界区需要等待。
- 死锁（dead lock）：两个或两个以上的进程，在相互等待完成特定任务，而最终没法将自身任务进行下去
- 饥饿（starvtion）：一个可执行的程序，被调度器持续忽略，以至于虽然处于可执行状态却不被执行。

- 进程间通信
  - 信号量
    - 信号量-抽象数据类型
      - 一个整型（sem）,两个原子操作
        - P()：sem减1，如果sem<0，等待，否则继续
        - V()：sem加1，如果sem<=0，唤醒一个等到的P
      - 信号量是整数
      - 信号量是被保护的变量：1.操作必须是原子操作 2.只能通过P()和V()改变
      - P()能阻塞，V()不会阻塞
      - 我们假定信号量是“公平的”
        - 没有线程被阻塞在P()仍然阻塞，如果V()被无线频繁调用（在同一个信号量）
        - 在实践中,FIFO经常被使用
      - 二进制信号量：可以是0和1
      - 一般/计数信号量：可以取任何非负值
      - 两者相互表现
    - 信号量使用
      - 可以用在两方面：
        - 互斥
        - 条件同步（调度约束----一个线程等待另一个线程的事件发生）
      - 用二进制信号量实现互斥

```
Mutex = new Semaphore(1);
Mutex->P();
临界区
Mutex->V();
```

      - 用二进制信号量实现的调度约束

```
Condition = new Semaphore(0);
Thread A                ThreadB
....                    ....
Condition->P()          condition->V()
.....                   ....
```

      - 一个线程等待另一个线程处理事情
    - 信号量实现
      - 使用硬件原语：1.禁用中断 2.原子指令
      -

使用硬件原语

禁用中断

原子指令(test-and-set)

类似锁

例如：使用‘禁用中断’

```
classSemaphore {
int sem;
WaitQueue q;
}
```

□

```
Semaphore::P() {
  sem--;
  if (sem < 0) {
     Add this thread t to q;
     block(p);
  }
}
```

```
Semaphore::V() {
  sem++;
  if (sem<=0) {
     Remove a thread t from q;
     wakeup(t);
  }
}
```

- □ 信号量的双用途
  - ◆ 互斥和条件同步
  - ◆ 但等待条件是独立的互斥
- □ 读/开发代码比较困难
  - ◆ 程序猿必须非常精通信号量
- □ 容易出错
  - ◆ 使用的信号量已经被另一个线程占用
  - ◆ 忘记释放信号量
- □ 不能够处理死锁

○ 管程

- ▪ 目的：分离互斥和条件同步的关注
- ▪ 什么是管程：包含了一系列的共享变量以及针对这些
  - □ 一个锁：指定临界区
  - □ 0或者多个条件变量：等待/通知信号量用于管理并发访问共享数据
- ▪ 一般方法
  - □ 收集在对象/模块中的相关共享数据
  - □ 定义方法来访问共享数据
- ▪ Lock()
  - □ Lock::Acquire() - 等待直到锁可用，然后抢占锁
  - □ Lock::Release() - 释放锁，唤醒等待者如果有
- ▪ Condition Variable
  - □ 允许等待状态进入临界区
    - ◆ 允许处于等待（睡眠）的线程进入临界区
    - ◆ 某个时刻原子释放锁进入睡眠
  - □ wait()
    - ◆ 释放锁，睡眠，重新获得锁返回后
  - □ Signal()
    - ◆ 唤醒等待者一个（或者所有等待者使用broadcast()），如果有

- 实现
  - 需要维持每个条件队列
  - 线程等待的条件等待signal()

□

```
Class Condition {
   int numWaiting = 0;
   WaitQueue q;
}
```

```
Condition::Wait(lock){
   numWaiting++;
   Add this thread t  to q;
   release(lock);
   schedule(); //need mutex
   require(lock);
}
```

```
Condition::Signal(){
   if (numWaiting > 0) {
      Remove a thread t from q;
      wakeup(t); //need mutex
      numWaiting--;
   }
}
```

○ 信号量和管程的经典案例

- ▪ 生产者和消费者
- ▪ 读者写者问题（reader writer）

- 读者优先：只要有一个读者处于活动，后来的读者都会被接纳，写者就会始终阻塞。(信号量实现)

Writer

```
sem_wait(WriteMutex);

  write;

sem_post(WriteMutex);
```

Reader

```
sem_wait(CountMutex);
 if (Rcount == 0)
   sem_wait (WriteMutex);
 ++Rcount;
sem_post(CountMutex);

read;

sem_wait(CountMutex);
 --Rcount;
 if (Rcount == 0)
   sem_post (WriteMutex);
sem_post(CountMutex)
```

- 写者优先（管程实现）

```
AR = 0;          // # of active readers
AW = 0;          // # of active writers
WR = 0;          // # of waiting readers
WW = 0;          // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

```
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```
Private Database::DoneRead() {
lock.Acquire();
    AR--;
    if (AR ==0 && WW > 0) {
        okToWrite.signal();
    }
lock.Release();
}
```

```
Public Database::Read(){
        //wait until no writers;
        StartRead();
        Read database;
        //check out - wake up waiting writers;
        Done Read();
}
```

```
Private Database::StartWrite() {
lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
lock.Release();
}
```

```
Private Database::DoneWrite() {
lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    }
    else if (WR > 0) {
        okToRead.broadcast();
    }
lock.Release();
}
```

```
Public Database::Write(){
//wait until no reader/writers'
StartWrite();
Write database;
//check out - wake up waiting readers/writers;
Done Write();
}
```

- 哲学家问题
  - 一个环形餐桌，5位哲学家，5把叉子，每个哲学家吃饭需要左右两把叉子一起吃。
  - 解决方案

1. 必须有一个数据结构，来描述每个哲学家的当前状态；

```
#define   N          5            // 哲学家个数
#define   LEFT       i            // 第i个哲学家的左邻居
#define   RIGHT      (i+1)%N      // 第i个哲学家的右邻居
#define   THINKING   0            // 思考状态
#define   HUNGRY     1            // 饥饿状态
#define   EATING     2            // 进餐状态
int   state[N];                   // 记录每个人的状态
```

2. 该状态是一个临界资源，对它的访问应该互斥地进行

```
semaphore   mutex;                // 互斥信号量，初值1
```

3. 一个哲学家吃饱后，可能要唤醒邻居，存在着同步关系

```
semaphore   s[N];                 // 同步信号量，初值0
```

函数philosopher的定义

```
void   philosopher(int   i)       // i的取值：0到N－1
{
```

```
//功能：要么拿到两把叉子，要么被阻塞起来
Void take_forks(int i)
{
    P(mutex); //进入临界区，保护叉子state
    State[i]=HUNGRY//我饿了！
    Test_take_left_right_forks(i) //试图拿两把叉子
    V(mutex);//退出临界区
    P(s[i]);
}

Void test_take_left_right_forks(int i)
{
    If (state[i]==HUNGRY && state[LEFT]!=EATING &&
    State[RIGHT]!=EATING)
    {
        State[i]=EATING; //两把叉子到手
        V(s[i])//通知自己吃饭
    }
}

//功能：把两把叉子放回原处，并在需要的时候叫左右
```

```
函数philosopher的定义
void   philosopher(int  i)        // i的取值：0到N－1
{
    while(TRUE)                   // 封闭式循环
    {
S1 ——→think( );                  // 思考中…
S2 -S4 ——→take_forks(i);          // 拿到两把叉子或被阻塞
S5 ——→eat( );                     // 吃面条中…
S6 -S7 ——→put_forks(i);           // 把两把叉子放回原处
    }
}
```

```
    }
}

//功能：把两把叉子放回原处，并在需要的时候叫左右
Void put_forks(int i)
{
    P(mutex);
    State[i]=THINKING;
    Test_take_left_right_forks(LEFT);
    Test_take_left_right_forks(RIGHT);
    V(mutex);
}
```

- 死锁（deadlock）
  - 死锁问题
  - 系统模型
    - 资源分配图
    - 如果图中不包含循环肯定没有死锁，如果包含循环，有可能存在死锁
  - 死锁特征：死锁出现有以下特征
    - 互斥：在一个时间只能有一个进程使用资源
    - 持有并等待：进程保持至少一个资源正在等待获取其他进程持有的额外资源
    - 无抢占：一个资源只能被进程资源释放，进程已经完成了它的任务之后
    - 循环等待：进程之间互相等待
  - 死锁处理方法
    - Deadlock prevention(死锁预防)
      - 限制申请方式-打破四个死锁特征
        - 互斥：共享资源不是必须的，必须占用非共享资源
        - 占用并等待：必须保证当一个进程请求资源时，它不持有任何其他资源
          - 可能导致资源利用率低；可能发生饥饿
        - 无抢占
        - 循环等待 - 对所有资源类型进行排序，并要求每个进程按照资源的顺序进行申请（多用于嵌入式操作系统）
    - Deadlock avoidance(死锁避免)
      - 最简单和最有效的模式是要求每个进程声明它可能需要的每个类型资源的最大数目
      - 资源的分配状态时通过限定提供与分配的资源数量，和进程的最大需求
      - 死锁避免算法动态检查资源分配状态，以确保永远不会有一个环形等待状态
      - 安全状态
        - 当一个进程请求可用资源，系统必须判断立即分配是否能使系统处于安全状态
        - 系统处于安全状态：针对所有进程，存在安全序列
        - 序列<P1,P2,...PN>是安全的：针对每个Pi,Pi要求的资源鞓能够由当前可用的资源+所有Pj持有的资源来满足，j<I
          - 如果Pi资源的需求不是立刻可用，那要等到Pj完成再执行Pi

        银行家算法避免死锁---安全状态估计算法

        n=进程数量，m=资源类型数量

        Max（总需求量）：n*m矩阵。如果Max[I,j]=k，表示进程Pi最多请求资源类型Rj的k个实例。

        Available（剩余空闲量）：长度为m的响亮。如果Avialable[j]=k。有k个Rj可用

        Allocation（已分配量）：n*m矩阵。如果Allocation[I,j]=k，表示Pi占用了k个Rj

        Need（未来需要量）：n*m。如果Need[i,j]=k,则Pi可能需要至少k个Rj实例完成任务

        Need[I,j]=Max[I,j]-Allocation[I,j]

        1.work和Finish分别是长度为m和n的向量

        初始化：work=Available

        Finish[i]=false for i=1,2...n.

        2.找这样的I

        （a）Finish[i]=false;

        （b）need[i]<=work

        没有找到这样的i，转4

        3.Work=Work+Allocation[i]

        Finish[i]=true;

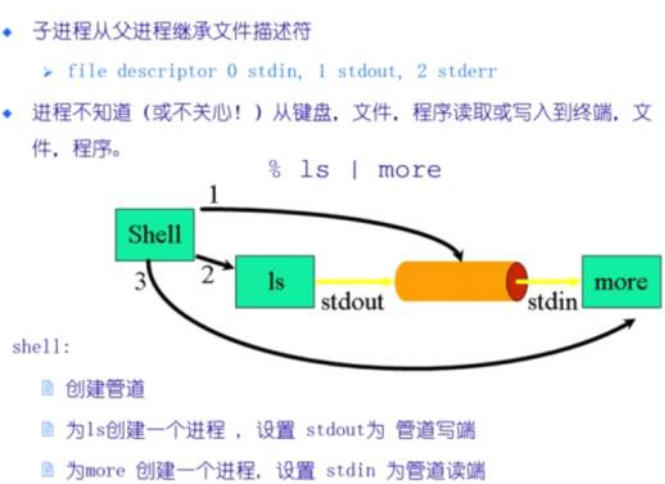
        转2

        4.如果Finish[i]=true for i=1,2,...n；那么系统处于安全状态，否则处于不安全状态

假设把request给他，判断是否安全

- Deadlock detection(死锁检测)
  - 允许系统进入死锁状态
  - 死锁检测算法
    - 定期执行银行家算法检测死锁
  - 恢复机制
- Recovery from deadlock(死锁恢复)
  - 终止所有的死锁进程
  - 在一个时间内终止一个进程知道死锁消除
  - 终止进程的顺序
    - 进程的优先级
    - 进程运行时间
    - 进程占用资源
    - 进程完成需要的资源
    - 多少进程需要被终止
    - 进程是交互还是批处理

- IPC（进程间通信）
  - 概述
    - 通信模型
    - 直接及间接通信
    - 阻塞与非阻塞
    - 通信链路缓冲
  - 信号 Signal
    - 目标：软件中断通知事件处理
    - 接收到信号时会发生什么
      - Catch：指定信号处理函数被调用
      - Ignore：依靠操作系统的默认操作
        - 例如：Abort, memory dump, suspend or resume process
      - Mask：闭塞信号因此不会传送
        - 可能是暂时的（当吹同样类型的信号）
    - 不足
      - 不能传输要交换的任何数据
    - 实现方式：
      - 注册一个handle
      - 触发软件中断时，修改内核态PC指针指向中断处理函数，之后指向PC原来的指向
  - 管道：重定向
    - 子进程从父进程继承文件描述符，子进程共享一些父进程的资源



  - 消息队列
    - 消息队列按FIFO来管理消息
    - message：作为一个字节序列存储
    - Message queue：消息数组

        - 与管道的区别：
            - 管道是父进程为子进程建立通道，如果没有父子关系，子进程无法共享；管道是字节流，没有结构
            - 消息队列没有父子关系限制，可以传递结构化数据
    - 共享内存
        - 进程
            - 每个进程有私有地址空间
            - 在每个地址空间内，明确的设置了共享内存段
        - 优点
            - 快速、方便的共享数据
        - 缺点
            - 必须同步数据访问（需要同步互斥保证）

文件系统

- 基本概念
    - 文件系统和文件
        - 文件系统：一种用于持久性存储的系统抽象
        - 文件：文件系统中一个单元的相关数据在操作系统中的抽象
    - 文件描述符
    - 目录
    - 文件别名
    - 文件系统种类
    - 文件系统功能
        - 分配文件磁盘空间
            - 管理文件块
            - 管理空闲空间
            - 分配算法
        - 管理文件集合
            - 定位文件及其内容
            - 命名：通过名字找到文件的接口
            - 最常见：分层文件系统
            - 文件系统类型（组织文件的不同方式）
        - 提供的便利及特征
            - 保护：分层来保护数据安全
            - 可靠性/持久性：保持文件的持久即使发生崩溃、媒体错误、攻击等
- 虚拟文件系统
    - 分层结构
        - 上层：虚拟（逻辑）文件系统
        - 底层：特定文件系统模块
    - 目的：对所有不同文件系统的抽象
    - 功能：
        - 提供相同的文件和文件系统接口
        - 管理所有文件和文件系统关联的数据结构
        - 高效查询例程，遍历文件系统
        - 与特定文件系统模块的交互
    - 包含
        - 卷控制块（UNIX:"superblock"）
            - 每个文件系统一个
            - 文件系统详细信息
            - 块，块大小，空余块计数/指针等
        - 文件控制块（UNIX:"vnode"or"inode"）
            - 每个文件一个
            - 文件详细信息
            - 许可，拥有者，大小，数据库位置等
        - 目录节点（Linux：“dentry”）

        - 每个目录项一个（目录和文件）
        - 将目录项数据结构及树形布局编码成树形数据结构
        - 指向文件控制块、父节点、项目列表等
- 数据块缓存
    - 数据块按需读入内存
    - 数据块使用后被缓存
    - 两种数据块缓存方式
        - 普通缓冲区
        - 页缓存：统一缓存数据块和内存页
    - 分页要求：当需要一个页时才将其载入内存
    - 支持存储：一个页（在虚拟地址空间中）可以被映射到一个本地文件中（在二级存储中）
- 打开文件的数据结构
    - 打开文件描述
        - 每个被打开的文件一个
        - 文件状态信息
        - 目录项、当前文件指针、文件操作设置等
    - 打开文件表
        - 一个进程一个
        - 一个系统级的
        - 每个卷控制块也会保存一个列表
        - 所以如果有文件被打开将不能被卸载
    - 强制和劝告
        - 强制 - 根据锁保持情况和需求拒绝访问
        - 劝告 - 进程可以查找锁的状态来决定这么做
- 文件分配
    - 分配方式
        - 连续分配
            - 文件头指定起始块和长度
            - 位置，分配策略：最先匹配，最佳匹配等
            - 优势：1.文件读取表现好 2.高效的顺序和随机访问
            - 劣势：1.碎片 2.文件增长问题（预分配？按需分配？）
        - 链式分配
            - 文件以数据块链表方式存储
            - 文件头包含了到第一块和最后一块的指针
            - 优势：1.创建、增大、缩小很容易 2.没有碎片
            - 缺点：1.不能进行真正的随机访问 2.可靠性（破坏一个链，后续就无法找到）
        - 索引分配
            - 为每个文件创建一个名为索引数据块的非数据数据块（到文件数据块的指针列表）
            - 文件头包含了索引数据块
            - 优势：1.创建、增大、缩小很容易 2.没有碎片 3.支持直接访问
            - 劣势：1.当文件很小时，存储索引的开销 2.如何处理大文件?
    - 指标
        - 高效：如存储利用（外部碎片）
        - 表现：如访问速度
- 空闲空间列表
- 多磁盘管理-RAID

    通常磁盘通过分区来最大限度减小寻道时间:

    一个分区是一个柱面的集合

    每个分区都是逻辑上独立的磁盘

    分区: 硬件磁盘的一种适合操作系统指定格式的划分

    卷: 一个拥有一个文件系统实例的可访问的存储空间(通常常驻在磁盘的单个分区上)

    使用多个并行磁盘来增加: 吞吐量(通过并行),可靠性和可用性(通过冗余)

    RAID - 冗余磁盘阵列: 各种磁盘管理技术;RAID levels: 不同RAID分类,如RAID-0,RAID-1,RAID-5

    实现: 在操作系统内核: 存储,卷管理; RAID硬件控制器(IO)

RAID-0

- □ 数据块分成多个子块, 存储在独立的磁盘中: 和内存交叉相似
- □ 通过更大的有效块大小来提供更大的磁盘带宽

RAID-1

- □ 可靠性成倍增长
- □ 读取性能线性增加(向两个磁盘写入,从任何一个读取)

RAID-4

- □ 数据块级磁带配有专用奇偶校验磁盘: 允许从任意一个故障磁盘中恢复
- □ 条带化和奇偶校验按byte-by-byte或者bit-by-bit: RAID-0,4,5: block-wise ;RAID-3: bit-wise

RAID-5

- □ 每个条带快有一个奇偶校验块,允许有一个磁盘错误

RAID-6

- □ 两个冗余块,有一种特殊的编码方式,允许两个磁盘错误

- 磁盘调度

读取或写入时,磁头必须被定位在期望的磁道,并从所期望的扇区开始

寻道时间: 定位到期望的磁道所花费的时间

旋转延迟: 从扇区的开始处到到达目的处花费的时间

平均旋转延迟时间 = 磁盘旋转一周时间的一半

寻道时间是性能上区别的原因

对单个磁盘,会有一个IO请求数目

如果请求是随机的,那么会表现很差

FIFO:

按顺序处理请求

公平对待所有进程

在有很多进程的情况下,接近随机调度的性能

最短服务优先:

选择从磁臂当前位置需要移动最少的IO请求

总是选择最短寻道时间

skan:

磁臂在一个方向上移动,满足所有为完成的请求,直到磁臂到达该方向上最后的磁道

调换方向

c-skan:

限制了仅在一个方向上扫描

当最后一个磁道也被访问过了后,磁臂返回到磁盘的另外一端再次进行扫描

c-loop(c-skan改进):

磁臂先到达该方向上最后一个请求处,然后立即反转