操作系统复习

二、计算机系统结构

1、同步I/O：在I/O开始后，控制权由用户->内核，直到I/O完成再返回给用户。

2、异步I/O：在I/O开始后，控制权就返回到用户层，无需等待I/O的完成。

3、DMA帮助控制组线：用于帮助高速I/O设备用接近内存的速度传输信息。

4、cache寄存器：用于存放经常访问的部分数据。

5、保护Protection:

- 硬件保护：
  - （1）二重模式：用户模式、内核模式（代表操作系统）
  - （2）内存保护：用两个寄存器帮助进程的保护（基址寄存器base、界限寄存器limit）来判断地址是否合规
  - （3）CPU保护：timer用于中断，确保操作系统的控制。防止进程运行过长时间。
  - （4）I/O保护：所有的I/O操作在内核模式检验，可行后执行请求（确保用户程序永远无法获得内核模式权限）。

6、Buffer和Cache的区别：

- Buffer（缓冲区）是系统两端处理速度平衡（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到 流量整形 的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。
- Cache（缓存）则是系统两端 处理速度不匹配 时的一种折衷策略。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memory hierarchy）的策略来减小这种差异带来的影响。

四、进程

是一个拥有资源和可独立运行的基本单位

1、进程和程序的区别：

- 程序是静态的，进程是动态的。
- 程序是永久的，进程是暂时的。
- 进程具有并发性
- 一个程序可以对应多个进程，一个进程可以执行一个或多个程序。

2、进程的调度：

- 长程调度：从外存中选择一个任务（作业）送到内存中，为之创建线程，并将这个线程加入准备队列
- 中程调度：从将外存中挂起的线程中选择线程送到内存，也可以是将进程从内存或从CPU竞争中移出，从而降低多道程序设计的程度
- 短程调度：从准备队列中选择线程送到CPU执行（分为抢占式和非抢占式）

3、I/O为主的进程花更长的时间进行I/O操作，CPU为主的进程花更多时间进行计算。

4、CPU进行进程切换的时候，系统将旧进程的内容保存，然后加载新的进程所保存的内容。其中的额外开销（overhead）一定要尽可能小，它取决于硬件。

5、三种资源共享方式：

```
父进程与子进程共享所有资源
子进程共享父进程中的部分资源:读取的时候共享，写入时创建新的空间
父进程与子进程不共享任何资源
```

当进程创建新进程时，有两种执行可能

```
父进程与子进程并发执行
父进程等待，直到某个或全部子进程执行完毕
```

6、级联终止：如果一个进程终止（正常或不正常），那么它的所有子进程也被终止。

7、进程分为：

- 独立进程：不能影响其他进程和被其他进程影响
- 合作进程：能影响其他进程和被其他进程影响

8、生产者消费者模型：

- 生产者进程生产信息、消费者进程消费信息
- 存在一个缓冲池（buffer pool）：生产者填充、消费者消费；分为有限缓冲和无限缓冲；

9、IPC(Inter-Process Communication)内部进程通信：

- 如何通知对方：
  - 直接通信：明确接受者，直接去找他，发送进程和接收进程必须命名对方，以方便通信。
  - 间接通信：通过邮箱或端口，每个邮箱有固定的ID
- 如何相互同步：通信进程所交换的消息都驻留在临时队列中队列实现有三种方法：零容量；有限容量；无限容量。
  - 阻塞式：队列满时，等待
  - 非阻塞：队列满时，进行其他操作
- 若三个进程通信如何解决：允许系统选择接收者，并且可以告诉发送者谁是接收者

五、线程

1、为了减少程序在并发执行时所付出的时空开销，使得OS有更好的并发性：故引入线程

2、线程（又称轻量级进程Lightweight process）,是cpu使用的基本单元

3、一个传统进程有多个线程，若控制多个线程则能同时执行多条任务

4、多线程编程的优点：

- 响应度高：由于是多个线程分别处理事件，故当一个线程处理一项任务时，能通过另一线程和用户交互。
- 资源共享：线程默认共享所属的进程和资源。
- 经济：进程创建所需的内存和资源的分配比较昂贵；而由于线程资源共享，故线程创建和上下文的切换更经济。

- 多处理器体系结构的利用：可以让每个进程并行运行在不同的处理器上。

5、线程分类（2类）

- 用户线程（user thread）：在内核上支持，由线程库实现。线程库提供对线程的创建、调度和管理。
- 内核线程（kernel thread）：由操作系统支持，内核线程的创建和管理通常要慢于用户线程

6、多线程模型（3个）

- 多对一模型（many to one）：多个用户线程对应一个内核线程。
  - 多个用户线程映射到一个内核线程上，线程管理在用户空间所以效率高。
  - 当一个线程阻断系统调用，整个线程会阻塞，因为同一时刻只有一个线程能访问内核。
  - 多个线程不能并行的运行在多处理器上。
- 一对一模型（one to one）：每个用户线程映射到一个内核线程上。
  - 当一个线程阻塞系统调用时，另一个线程能继续执行，故其提供了更好的并发功能。
  - 多进程可以并行的运行在多处理器上。
  - 缺点是：每创建一个用户进程就要创建一个内核进程，而内核线程的开销影响程序性能，故该模型限制了系统支持的最大线程数量。
- 多对多模型（many to many）：多个用户线程对应多个内核线程。
  - 一个线程执行阻塞系统调用时，内核能调度另一个线程来执行。
  - 开发人员可以创建任意多的必要用户线程，并且相应内核线程能在多处理器系统上并行执行。

六、CPU调度（进程调度）

计算*1
几乎所有计算机资源在使用前都要被调度，调度是操作系统最基本的功能。

调度基本概念

1、多道程序设计的目标是：在任何时候都有一个进程在运行，以使CPU使用率最大化。

2、I/O为主程序通常具有大量短CPU区间。CPU为主程序可能有少量的长CPU区间。

3、CPU Scheduler(CPU 调度器)：用于从就绪队列中选择进程并分配其CPU资源。进程选择由短期调度程序执行，短期调度程序必须要快。

4、进程派遣器（Process dispatcher）

- 派遣程序（dispatcher module）：
- 派遣延迟（dispatcher latency）：停止一个进程，唤醒另一个进程的时间

调度的评价标准

1、评价标准

- CPU使用率：需要使CPU尽可能忙，对真实系统，它应从40%到90%
- 吞吐量：一个时间单元内所完成进程的数量
- 周转时间：从 进程提交到完成的时间间隔 ，是所有时间段之和，包括等待进入内存、在就绪队列中等待、在CPU上执行和I/O执行
- 等待时间：进程在就绪队列中等待时间之和， 即从提交到开始执行所需时间
- 响应时间：从提交请求到产生第一响应时间。

2、需要使CPU使用率和吞吐量最大化，而使周转时间、等待时间和相应时间最小化。

调度算法

四种（FCFS, SJF, Priority Scheduling, RR）

1、FCFS先来先服务：性能不稳定，取决于进程到来的顺序。

- 护航效果（Convoy effect）:短进程排在长进程之后，导致效果差（平均等待时间和周期变长）。

为了解决护航问题

2、SJF最短作业优先：当cpu可用时调用最短cpu区间的进程,若相同则使用FCFS。

- 抢占式：其中有SRTF最短剩余时间优先，如出现更短剩余CPU区间时，可抢占。
- 非抢占式
- SJF是最佳的，但是无法实现：因为无法预测未来进程CPU区间长度，故效率取决于预测的准确程度。

3、Priority Scheduling优先权调度：每个进程都有一个与之相关的优先数

- 抢占式、非抢占式

- 饥饿：低优先级进程永远不会执行
- 老化：随着时间增加，进程优先级提高

4、RR轮转法：进程平等、需要设定一个时间片q

- 当进程完成且剩余时间，贡献剩余时间
- 进程未完成则置于队尾
- 重要的是q的设置：太大会变成FCFS，太小则浪费过多时间在切换上。如果绝大多数进程能在一个时间片内结束其下一个CPU区间，那么平均周转时间会有所改善。

5、Multilevel queue algorithm多级队列调度：在进程容易分成不同组的情况下，可以有另一类调度算法。例如，进程通常分为前台进程（或交互进程）和后台进程（或批处理进程）。这两种类型的进程具有不同的响应时间要求，进而也有不同调度需要。另外，与后台进程相比，前台进程可能要有更高的优先级（外部定义）。

6、Multilevel feedback queue algorithm多级反馈队列调度：

- 1）进程在进入待调度的队列等待时，首先进入优先级最高的Q1等待。
- 2）首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3三个队列，只有在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。
- 3）对于同一个队列中的各个进程，按照时间片轮转法调度。比如Q1队列的时间片为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。
- 4）在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配给新到达的作业（抢占式）。

算法评估

1、Deterministic modeling确定性模型：采用特定预先确定的负荷，定义在给定负荷下每个算法的性能，测定的是精确的数据，用处有限。

2、Queuing models排队模型：计算机系统可描述为服务器网络。每个服务器都有一个等待进程队列。CPU是具有就绪队列的服务器，而I/O系统具有设备队列。知道了到达率和服务率，就可以计算使用率、平均队列长度、平均等待时间等。这种研究领域称为排队网络分析。

3、Simulations模拟：为了获得对调度算法更为精确的评估，可使用模拟。

4、Implementation实现：即使模拟其精确度也是有限的。针对评估调度算法，惟一完全精确的方法是对它进行程序编码，将其放在操作系统内，并观测它如何工作。

七、进程同步

计算*1
设计利用信号量实现进程同步

1、原子操作：一个操作整体完成而没有中断。

2、关键区问题：关键区能够共享数据的部分代码，当一个进程访问关键区时，其他进程不允许在关键区执行。

满足：

- 互斥（Mutual Exclusion）：同时刻只有一个进程在关键区
- 有空让进（progress）：不存在关键区无进程时，关键区还无法进入的情况。
- 有限等待（Bounded Waiting）：

3、PV操作：实现进程互斥与同步的有效方法。PV操作与信号量的处理相关，P表示通过的意思（wait），V表示释放的意思（signal）。

4、Semaphores信号量：三个原子操作

- 信号量被初始化为非负的值
- wait：-1操作，若信号量为负数则挂起。
- signal：+1操作，若信号量非正则被唤醒。

5、信号量的死锁和饥饿：

- 死锁：两个或更多的进程无限的等待一个事件，而该中止等待的事件只能由这些等待进程之一来产生。
- 饥饿：无限阻塞，一个被悬挂的进程可能永远无法从信号量队列中移出。

6、信号量类型：

- 二值信号量：0，1
- 计数信号量：0~ n,可以用两个二值信号量来表示一个计数信号量。

7、生产者-消费者模型：

```
mutex =1;  二值信号量，用于保护缓冲池
full = 0;  计数有进程的单元
empty = 0; 计数空闲的单元
生产者:                                                    消费者:
while(1)                                                   while(1)
{                                                          {
   ...                                                        ...
   produce an item in nextp                                    wait(full);       占用单元-1，当无占用单元时挂起
   ...                                                        wait(mutex);
   wait(empty);      空闲单元-1，当没有空闲单元存放时，挂起                  ...
remove one buffer to nextc
                                                              signal(mutex);
   wait(mutex);
   ... add nextp to buffer                                    signal(empty);    唤醒生产者
   signal(mutex);                                             ...
                                                              consume the item in nextc
   signal(full);     占用单元+1                                  ...
}                                                          }
```

8、读者-写者问题：

```
读者优先:
   readcount = 0:  用于记录多少进程正在进行读操作。
   mutex = 1:      当更新变量readcount时，确保不会同时记录两个reader，即同时刻只能有一个reader计入读者队列中
   wrt = 1         读写互斥变量

读者:                                                      写者:
   wait(mutex);                                               wait(wrt);
   readcount++;  当有一个读者时，就加锁防止写者进入                      进行写操作
   if(readcount == 1) wait(wrt);                              signal(wrt);
   signal(mutex);
      进行reading操作
   wait(mutex);
   readcount--;  当不剩下读者时，解锁此时写者可以进入
   if(readcount == 0) signal(wrt);
   signal(mutex);
```

9、哲学家问题：有五名哲学家和五根筷子，哲学家只会吃饭和思考。

为了防止死锁，最多每次只能有4个哲学家吃饭（因为五个哲学家都取左侧筷子时会产生死锁）

```
chopstick[5] = {1,1,1,1,1} 有五根筷子，即五个二值信号量
coord = 4;          每次只能有四个哲学家吃饭

哲学家i:
do{
    wait(coord);
    wait(chopstick[i]);
    wait(chopstick[i+1] % 5);
        ...eat...
    signal(chopstick[i]);
    signal(chopstick[i+1] % 5);
    signal(coord);
        ...think...
}while(i)
```

八、死锁

计算*1

1、死锁：一组阻塞进程分别占有一定的资源并等待获取另外一些已经被同组其他进程所占有的资源。

2、正常操作模式下，进程按如下顺序使用资源：

- 申请：如果申请不能立即被允许，那么申请进程必须等待直到它获得该资源为止。
- 使用：进程对资源进行操作。
- 释放：进程释放资源。

3、产生死锁条件（4'）：

- 互斥Mutual exclusion：至少有一个资源必须处于非共享模式；即一次只有一个进程使用。如果另一资源申请该资源，那么申请进程必须延迟直到该资源释放为止。

- 占有并等待Hold and wait：一个进程必须占有至少一个资源，并等待另一资源，而该资源为其他进程所占有。
- 非抢占No preemption：资源不能被抢占；即，只有进程完成其任务之后，才会释放其资源。
- 循环等待Circular wait：有一组进程{P0, P1, …, Pn}，P0等待的资源为P1所占有，P1等待的资源为P2所占有，Pn-1等待的资源为Pn所占有，Pn等待的资源为P0所占有。

4、死锁问题可用称为系统资源分配图的有向图进行更为精确地描述：

- 无环：如果图没有环，那么系统就一定没有进程死锁。
- 有环：如果图有环，那么可能存在死锁。
  - 资源只有一个实例时一定死锁。
  - 资源有多个实例时可能死锁。

5、解决死锁的操作：

- 使用协议以 预防或避免 死锁，确保系统永远不会进入死锁状态
- 允许系统进入死锁状态，然后 检测 它，并加以 恢复
- 忽略 这个问题，认为死锁不可能在系统内发生。为绝大多数操作系统如UNIX使用。

6、死锁预防：只要打破死锁出现的四个必要条件之一就可以。

- 互斥：对于非共享资源，必须要有互斥条件。共享资源不要求互斥访问。
- 占有并等待：为了确保占有并等待条件不会在系统内出现，必须保证——当一个进程申请一个资源时，它不能占有任何其他资源。
  - 一种方法是每个进程在执行前申请并 获得所有资源 。
  - 另一种方法是允许进程 在没有资源时才可申请资源 。
  - 两种方法都有资源利用率低和可能发生饥饿的缺点。
- 非抢占：如果一个进程占有资源并申请另一个不能立即分配的资源，那么其现已分配的资源都被抢占。通常应用于其状态可以保存和恢复的资源。
- 循环等待：对所有资源进行完全排序，且要求每个进程按递增顺序来申请资源

对于死锁的互斥条件和不可抢占条件，由于它们是由资源本身的独占性所决定的，不允许进程同时共享和抢占，所以这两个条件不可能被破坏，因此只能从另两个必要条件入手，出现了静态资源分配法（避免占有并等待）和有序资源分配法（避免循环等待）。

7、死锁的避免：

- 方法一：假定有了关于每个进程的申请与释放的完全顺序。可决定进程是否因申请而等待。每次申请要求系统考虑现有可用资源、现已分配给每个进程的资源和每个进程将来申请与释放的资源，以决定当前申请是否满足或必须等待从而避免死锁将来发生的可能性。
  - 通过限制资源申请的方法来预防死锁。
  - 副作用是设备使用率低和系统吞吐率低。
- 方法二：要求每个进程事先声明它所需要的每种资源的最大数量；死锁避免算法动态检查资源分配状态，以保证不存在循环等待的条件。（银行家算法）

8、安全状态Safe state：（要求会算安全序列）

- 如果存在包含所有进程的一种安全序列，则系统是安全的。
- 若不存在，则是不安全状态，但不一定死锁。
- 死锁避免：让程序保持在安全状态下。

9、如果一个系统既不采用死锁预防算法也不采用死锁避免算法，那么可能会出现死锁。

系统需提供：

- 死锁检测
- 思索恢复

10、死锁恢复：（两种方式：进程终止，资源抢占）

- 进程终止：
  - 终止所有进程
  - 每次kill掉一个进程直到解决死锁
- 资源抢占
  - 选择一个牺牲品：选择代价最小的
  - 回滚：必须将被抢占进程的状态恢复到某个安全状态
  - 可能会造成 饥饿 （低优先级的进程的资源始终被抢占）；解决方案：在代价因素中加上回滚次数。

九、内存管理

计算*1 ppt p27

背景

1、地址：

- 逻辑地址：由CPU生成；也称为虚拟地址
- 物理地址：内存单元所看到的地址
- 在编译时和加载时的地址绑定方案中，逻辑地址与物理地址是相同的。但是，执行时的地址绑定方案导致不同的逻辑地址和物理地址。
- 关键点：编译/加载之前已经形成绝对地址，逻辑地址的值已经等于绝对地址。

2、MMU内存映射单元：帮助从虚拟地址映射到物理地址。

- 用户进程所生成的地址在送交内存之前，都将加上 重定位寄存器 的值。 用户程序处理的是逻辑地址 ，它永远不会看到真实的物理地址。

3、Binding of Instructions and Data to Memory指令和数据到内存的地址绑定：（3'）

- 编译时：如果在 编译时知道进程将在内存中的驻留地址 ，那么就可以生成绝对代码。如果 将来开始地址发生了变化，那么就必须重新编译代码 。
- 加载时：如果在 编译时并不知道进程将驻留在何处 ，那么编译器就必须生成可重定位代码。(例如：从本模块开始的第14字节)，对于这种情况，最后 捆绑会延迟到加载时才进行 。
- 运行时：如果进程在 执行时可以从一个内存段移到另一个内存段 ，那么 捆绑必须延迟到执行时 才进行。采用这种方案需要特定硬件支持才行。

4、动态加载：执行程序时，并非一次性将子程序加载。一个子程序只有在调用时才被加载。

5、动态链接：链接过程推迟到执行时来进行，即用到才会链接，不用便不链接，只存地址。

6、overlays覆盖：

- 当程序大于内存所分配空间时，需要overlays
- 在任何给定时间内，内存只需要保存需要的（needed）指令和数据。

7、swapping交换：进程可以暂时从内存中交换到备份存储上，当需要再执行时再调回到内存中。（换入换出的基本单位为进程）

滚进、滚出 — 是交换策略的一个变种，被用于基于优先权的调度算法中。如果一个更高优先级进程来了且需要服务，内存管理可以交换出低优先级的进程，以便可以装入和执行更高优先级的进程。当更高优先级进程执行完后，低优先级进程可以交换回内存以继续执行。

- 交换swapping通常不执行，但当有许多进程运行且内存空间吃紧时，交换开始启动。如果系统负荷降低，那么交换就暂停。
- 交换时间的主要部分是转移时间(transfer time)。总的转移时间与所交换的内存空间直接成正比。
    - 如 1000KB / 5000KB/s = 0.2s = 200ms

连续内存分配

1、内存通常分为两个区域：

- 一个用于驻留操作系统，常与中断向量一起放在低内存
- 另一个用于用户进程，常放在高内存。

2、内存保护

- 保护操作系统不受用户进程所影响
- 保护用户进程不受其他用户进程所影响
- 通过采用重定位寄存器和界限寄存器，可以实现对内存的保护
    - 重定位寄存器含有最小的物理地址值
    - 界限寄存器含有逻辑地址值的范围
    - 每个逻辑地址必须小于界限寄存器
    - MMU（内存映射单元）动态地将逻辑地址加上重定位寄存器的值后映射成物理地址送交给内存单元

3、Multiple-partition allocation（多重分区分配）：

- （Hole）孔：可用的内存块；不同大小的孔分散在整个内存中。
    - 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，可以为该进程分配所需的内存，未分配的可以下次再用。
    - 如果孔太大，那么就分为两块：一块分配给新进程；另一块还回到孔集合。
    - 如果新孔与其它孔相邻，那么将这些孔合并成大孔。
    - 当进程终止时，它将释放其内存，该内存将还给孔集合。
- 从一组可用孔中选择一个空闲孔的最为常用方法：首次适应和最好适应在利用空间方面难分伯仲，但是首次适应要快些。都优于最差适应

- 首次适应(First fit)：分配第一个足够大的孔。
- 最佳适应(Best fit)：分配最小且足够的孔。
- 最差适应(Worst fit)：分配最大的孔。

4、Fragmentation碎片问题：

- 外部碎片问题：（未分配出去）总的内存空间之和满足请求，但不连续。
  - 解决：1）将已用空间上移，但要求重定位是动态的。2）允许物理地址空间非连续。3）允许物理地址空间为非连续，分页、分段或两者综合的机制。
- 内部碎片问题：（已分配，属于某些进程）进程所分配的内存可能比所需要的要大。这两个数字之差称为内部碎片，这部分内存在分区内，而又不能用。
  - 解决：使用合适的基本单位。

Paging分页

非连续的内存分配，可用于解决外部碎片

- Paging页式管理
- Segmentation段式管理
- 段页式管理

1、Paging页式管理：

- 物理内存分为固定大小的块，称为 帧 。
- 逻辑内存也分为同样大小的块，称为 页 。
- 备份存储也可分为固定大小的块，其大小与内存的 帧一样 。
- 页大小（与帧大小一样）是由 硬件决定 的。

2、CPU生成的地址（逻辑地址）包含两部分：页号、页偏移量

- 页号(p)：页号作为页表中的索引。页表中包含每页所在物理内存的基地址且存储在物理内存中
- 页偏移(d)：与页的基地址组合就形成了物理地址，就可送交物理单元。

3、页的大小：页大小（与帧大小一样）是由硬件来决定的。页的大小通常为2的幂，根据计算机结构的不同，其大小从512B到16MB字节不等。

- 选择2 的幂作为页面大小可以很容易的把逻辑地址转换成页号和页偏移。
- 如果逻辑地址空间是2^m，页面大小是2^n（byte 或word），那么逻辑地址的前m – n位是页号，后n 位是页偏移。

4、采用分页技术不会产生外部碎片：每个帧都可以分配给需要它的进程。不过，分页有内部碎片。随着时间的推移，页的大小也随着进程、数据和内存的不断增大而增大。

5、当系统需要执行一个进程时，它将检查该进程所需要的页数。如果进程需要n页，那么内存中至少应有n个帧。如果有，那么就可分配给新进程。进程的第一页装入一个已分配的帧，帧号放入进程的页表中。下一页分配给另一帧，其帧号也放入进程的页表中。ppt p32

6、页表的实现

- 若作为一组快速寄存器：（最简单）但当页表非常大时不可行。
- 若放入内存：慢
- 页表放在内存中，并将页表基寄存器（PTBR）指向页表：大大降低了切换时间
- PTLR页表长度寄存器：记录页表大小
- 两次内存访问问题可以用特别的快速查找硬件缓冲TLBs（称为关联内存或翻译后备缓冲器）来解决。

7、TLB关联内存进行并行搜索：当关联内存根据给定值查找时，它会同时与所有键进行比较。如果找到条目，那么就得到相应的值域
地址转换（A’, A’’）

- 如果A’在关联寄存器中，则直接取出其对应的frame #
- 否则从内存中的页表当中得到frame #
- 一个TLB 中往往包含很少的表项，查找的速度非常快；但是硬件昂贵。数量通常在64 到1,024 之间。

8、在TLB中找到特定的页号的时间概率为命中率（hit ratio） α

9、有效访问时间EAT：EAT = (1+ε)α+(2+ε)(1-α) = 2+ε+α；其中并行搜索时间为：ε ms；存储周期时间：1ms

分页环境下的 内存保护

1、通过与每个帧相关联的保护位来实现的。Valid (v) 、Invalid (i)

- 通常这些位保存在页表中，任何一位都能定义一个页是可读、可写或只可读。
- 有效无效位与页表中的每一条目相关联
  - 当该位有效时，该值表示 相关的页在进程的逻辑地址空间内， 因此是合法的页。
  - 该位无效时，该值表示 相关的页不在进程的逻辑地址空间内 。

页表结构

Hierarchical Paging (层次化分页)、
Hashed Page Tables (Hash页表)

方法：
1、Hierarchical Paging(层次化分页)：绝大多数现代计算机系统支持大逻辑地址空间，页表本身可以非常大。

- 1）是使用分页算法，就是将页表再分页。ppt p45。
  - 地址由外到内，外页表记录索引，页表的页记录页偏移。
- 2）当地址空间>32bits时，分页不现实。

2、Hashed Page Tables (Hash页表)：虚拟地址中的虚拟页号被放入hash页表中。hash页表的每一条目都包括一个链接组的元素。

- 虚拟页号与链表中的每 一个元素的第一个域相比较。
  - 如果匹配，那么对应的帧码就用来形成位置地址。
  - 如果不匹配，那么就对链表中的下一个域进行页码比较。
- 每个元素有三个域：虚拟页码、所映射的帧号和指向链表中下一个元素的指针。

共享页表

每个进程有自己的数据页。只需要在物理内存中保存一个编辑器拷贝。每个用户的页表映射到编辑器的同一物理拷贝，而数据页映射到不同的帧。

可共享的数据包括可重入的代码，而进程执行所需数据不可共享。

分段

分页内存管理中用户视角的内存和实际内存分离。用户视角的内存需要映射到实际内存。然而，用户并不会希望把内存看作一个线性字节数组。用户通常会愿意将内存看做为一组不同长度的段的集合，这些段之间并没有一定的顺序。分段是支持用户观点的内存管理方案。

（程序包含许多段。每个段有名字。每个段有不同的长度，段的长度由段在程序中的目的所决定。段中的每一元素由它们相对段的开始的偏移量所标识。各个段之间的顺序没有任何意义。）

1、段的逻辑地址由两个部分组成：<段号、段偏移量>

段号s，段偏移量为d；段号用做段表的索引。逻辑地址的偏移d应位于0和段界限之间。

2、段表：将二维的用户定义地址映射为一维物理地址。段表的每个条目都有段基地址和段界限（段表是一组基址和界限寄存器对）。

- 段基地址：包含段的起始地址
- 段界限：指定段的长度
- 段表基地址寄存器（STBR）指向内存中的段表的位置
- 段表长度寄存器（STLR）指示程序所用的段的个数，即段号s＜STLR才是有效的
- 如果偏移d合法，那么就与基地址相加而得到所需字节在物理内存的地址。

  ppt p63

3、段的保护：分段的一个显著优点是可以将段与对其的保护相关联。内存映射硬件会检查与段条目相关联的保护位以防止对内存的非法访问。

4、共享段：代码段通常会包括对自己的引用；如果共享这个段，那么所有共享进程必须使得共享代码段有同样的段号。

- 例如，如果共享sqrt子程序，一个进程需要将它作为段4，而另一个进程需要将它作为17，那么sqrt子程序如何来引用自己呢？因为只有一个sqrt子程序的物理拷贝，对于两个用户来说该子程序必须用同样方法来引用自己：它必须有一个唯一的段号。随着共享段用户的增加，寻找一个可以接受的段号的难度会增加。

5、段页式：

- 分页：无逻辑，无外部碎片
- 分段：有逻辑，无外部碎片
- 段表的条目包含的不是段的基地址，而是该段的页表的基地址

十、虚拟内存

计算*1 页置换

背景

在内存管理方法中要求在进程执行之前必须将这个进程放入内存之中，然而在许多情况下并不需要将整个程序放到内存中。即使在需要完整程序时，并不是同时需要所有的程序（例如，与overlay相似的情况）。

1、程序部分执行的好处：

- 物理内存空间不再受限，简化编程任务

- 使用更少的物理内存，故允许更多程序同时进行
- 装入或交换用户程序从内存到I/O会更少，故用户程序运行更快

2、虚拟内存：使计算机拥有比实际拥有的内存要大，虚拟内存将内存抽象成一个巨大的、统一的存储数组，进而将用户看到的逻辑内存与物理内存分开。（只在页面需要时，才把它们载入内存）

- 只要部分需要的程序放在内存中就能使程序执行
- 逻辑地址空间可以比物理地址空间大。
- 允许地址空间被多个进程共享
- 允许更多进程被创建
- 总结：需要更少的输入输出，更小的内存，更快的响应，更多的用户

3、虚拟内存的实现：

- 页式调度
- 段式调度

页式调度

1、页式调度：类似于分页+交换(Swapping)(实际不称为交换)

- 进程驻留在后备存储器上
- 当需要执行进程时，将它调入内存。不过，不是将整个进程换入内存，而是使用lazy swapper。lazy swapper只有在需要页时，才将它调入内存。
- 由于将进程看作一系列的页，而不是一个大的连续空间，因此使用“交换”从技术上来讲并不正确。交换程序对整个进程进行操作，而调页程序（pager）只是对进程的单个页进行操作。因此，在讨论有关请求页面调度时，需要使用调页程序而不是交换程序。
- 对计算机性能有重要影响
  - 若无页错误时，相当于访问内存
  - 存在页错误时，相当于访问磁盘

2、判断进程页在磁盘/内存上：添加有效-无效位

- 当该位设置为“有效”时，该值表示相关的页合法且也在内存中。
- 当该位设置为“无效”时，该值表示相关得页为无效（也就是，不在进程的逻辑地址空间中）或者有效但在磁盘上。（不在内存中）

3、页式调度步骤：

- 1）检查进程的页表，以确定该引用是合法还是非法的地址访问。

- 2）如果引用非法，那么终止进程。如果引用有效但是尚未调入页面，那么现在应调入。
- 3）找到一个空闲帧（从空闲帧链表中取一个）
- 4）调度一个磁盘操作，以便将所需要的页调入刚分配的帧
- 5）当磁盘读操作完成后，修改进程的内部表和页表，以表示该页已在内存中。
- 6）重新开始因非法地址陷阱而中断的指令。进程现在能访问所需的页，就好像它似乎总在内存中。

4、支持页式调度的硬件：

- 页表：可以记录有效-无效位
- 次级存储器：用于保护不在内存中的页

5、当在内存中找不到空闲帧时：用页置换算法

- 找到不在使用的页，并将其换出
- 理想状态下：产生最少的页错误

6、计算页式调度内存的有效访问时间：EAT = (1-p)(memory access) + p(page fault time)

进程创建

1、虚拟内存也能在进程创建时，提供其他好处：

- 写时拷贝：
  - 写时拷贝允许父进程和子进程开始时共享同一页面。这些页面标记为写时复制，即如果任何一个进程需要对页进行写操作，那么就创建一个共享页的拷贝。
  - 采用写时拷贝技术，很显然只有被进程所修改的页才会复制，因此创建进程更有效率。
  - 写时拷贝时所需的空闲页来自一个空闲缓冲池。该缓冲池中的页在分配之前先填零，以清除以前的页内容。
- 内存映射文件：
  - 内存映射文件I/O将文件I/O作为普通内存访问，它允许一部分虚拟内存与文件逻辑相关联。
  - 文件的内存映射可将一磁盘块映射成内存的一页。
  - 开始的文件访问按普通请求页式调度来进行，会产生页面错误。这样，一页大小的部分文件从文件系统读入物理页，以后文件的读写就按通常的内存访问来处理。

- 通过内存的文件操作而不是使用系统调用read和write，简化了文件访问和使用。
- 有的操作系统只能通过特定的系统调用提供内存映射，而通过标准的系统调用处理所有其他文件I/O。
- 有的操作系统将所有文件I/O作为内存映射，以允许文件访问在内存中进行。
- 多个进程可以允许将同一文件映射到各自的虚拟内存中，以允许数据共享。

页置换

随着多道程序的度的增加，平均内存使用接近可用的物理内存时，这种情况就可能发生。

1、出现页错误（未找到页）时，操作系统会检查其内部表以确定该页错误是合法还是非法的内存访问。进而操作系统会确定所需页在磁盘上的位置，但是却发现空闲帧列表上并没有空闲帧：

解决：

- 1）终止当前出现错误的用户进程
- 2）操作系统交换出一个进程，以释放其占用的所有帧，并降低多道程序的级别。
- 3）执行页置换。

2、页置换算法思想：

- 1）查找所需页在磁盘上的位置
- 2）查找一空闲帧
  - 如果有空闲帧，那么就使用它
  - 如果没有空闲帧，那么就使用页置换算法以选择一个“牺牲”帧（victim frame）。
  - 将“牺牲”帧的内容写到磁盘上；改变页表(valid-invalid bit)和帧表。
- 3）将所需页读入（新）空闲帧；改变页表和帧表
- 4）重启用户进程。

使用页置换，页错误处理时间加倍了。

3、modify bit (or dirty bit)修改位/脏位：用于降低额外开销

- 每页或帧可以有一个修改位通过硬件与之相关联。每当页内的任何字或字节被写入时，硬件就会设置该页的修改位以表示该页以修改。

4、如何选择一个置换算法：通常采用最小页错误率的算法。

注：内存的引用序列称为引用串。

- 针对特定内存引用串运行某个置换算法，并计算出页错误的数量。
- 仿真
- 实际应用（costly）

5、如何减少页错误率：

- 增加物理帧（内存）的数量：随着帧数量增加，那么页错误数量会降低至最小值。
- 选择合适的页置换算法

6、页置换算法的策略：（FIFO，OPT，LRU，LRU Approximation近似，LFU，MFU）

- 1）FIFO：替换最旧的页 ppt p37
  - Belady异常：对有的页置换算法，页错误率可能会随着所分配的帧数的增加而增加。
- 2）OPT最优置换算法：替换将来最晚被使用的页
  - 置换算法是所有算法中产生页错误率最低的，且决没有Belady异常的问题。
- 3）LRU：置换那些在最长时间中不会被使用的页（实际操作系统采用的）
  - 如何使用LRU：

    （1）计数器Counters：对每次内存引用，计数器都会增加。置换时置换具有最小时间的页。

    （2）页码堆栈Stack：每次引用一个页，该页就从堆栈中删除并放在顶部。故堆栈底部总是被置换的LRU页。
  - 近似LRU（LRU Approximation）：页表内的每项都关联着一个引用位。

  附加引用位算法

（1）可以为位于内存中的每个表中的页保留一个8bit的字节。

（2）在规定的时间间隔（如，每100ms）内，时钟定时器产生中断并将控制转交给操作系统。操作系统把每个页的引用位转移到其8bit字节的高位，而将其他位向右移，并抛弃最低位。这些8bit移位寄存器包含着该页在最近8个周期内的使用情况。

（3）如果将这8bit字节作为无符号整数，那么具有最小值的页为LRU页，且可以被置换。例如，具有值为11000100的移位寄存器的页要比值为01110111的页更为最近使用。

二次机会算法

（1）当要选择一个页时，检查其引用位。
（2）如果其值为0那么就直接置换该页。
（3）如果该引用位为1,那么就给该页第二次机会，并选择下一个FIFO页。当一个页获得第二次机会时，其引用位清零，且其到达时间设为当前时间。
（4）一种实现二次机会算法的方法是采用循环队列。

- 4）LFU(Least Frequently Used) 最不经常使用页置换算法：要求置换计数最小的页。
- 5）MFU (Most Frequently Used) 最常使用页置换算法: 要求置换计数最大的页。（基于具有最小次数的页可能刚刚调进来，且还没有使用。）

帧的分配

1、每个进程帧的最小数量是由体系结构来定义的，而最大数量是由可用物理内存的数量来定义的。

2、帧的分配算法：（3' 平均分配、比例分配、根据优先级分配）

- 1）平均分配：如100帧，5个进程，则给每个进程20帧
- 2）比例分配：根据进程的大小按比例分配。每个进程所分配的数量会随着多道程序的级别而有所变化。如果多道程序程度增加，那么每个进程会失去一些帧以提供给新来进程使用。反之，原来分配给离开进程的帧可以分配给剩余进程
- 3）根据进程优先级分配：如果进程 Pi 产生了一个页错误，那么从自身的帧中选择用于替换，从比自身优先级低的进程中选取帧用于替换

3、帧的全局分配和局部分配

- 1）全局置换：允许一个进程从所有帧集合中选择一个置换帧，而不管该帧是否已分配给其他进程；一个进程可以从另一个进程中取帧。
  - 可能产生很高的页错误率
  - 全局置换通常会有更好的系统吞吐量，且更为常用。
- 2）局部置换：要求每个进程仅从其自己的分配帧中进行选择
  - 可以很好地控制

系统颠簸

1、颠簸Thrashing：如果一个进程出现了页错误，必须置换某个页。然而所有页都在使用，它置换一个页，但又立刻再次需要这个页。因此，它会不断地产生页错误。进程继续页错误，置换一个页而该页又立即出错且需要立即调进来。这种频繁的页调度的行为称做颠簸。如果一个进程在换页上用的时间要多于执行时间，那么这个进程就在颠簸。

2、产生原因：CPU调度程序发现CPU利用率降低，因此会增加多道程序的程度。新进程试图从其他运行进程中拿帧，从而引起更多页错误，更长的调页设备的队列。进而CPU利用率进一步降低，CPU调度程序试图再增加程序的程度。这样系统颠簸就出现了，系统吞吐量突降，页错误显著增加。因此，有效访问时间增加。最终因为进程主要忙于调页，系统不能完成一件工作。

- 进程多到一定程度时，页错误陡然增加，cpu资源全去处理页错误了故而利用率低。

3、局部Locality：局部是一个经常使用页的集合。

- 局部模型说明，当进程执行时，它从一个局部移向另一个局部。
- 一个程序通常由多个不同局部组成，它们可能重叠。
- 如果进程有足够的帧包含它的所有局部，它会运行的很顺利。

4、颠簸的解决（2'）

为每个进程分配可以满足其当前局部的帧。该进程在其局部内会出现页错误，直到所有页在内存中；接着它不再会出现页错误直到它改变局部为止。如果分配的帧数少于现有局部的大小，那么进程会颠簸，这是因为它不能将所有将经常使用的页放在内存中。

- 工作集窗口（Working set window）：最近Δ个页面引用
  - 最近Δ个引用的页集合称为工作集合（Working set）
  - 如果一个页正在使用中，那么它就在工作集合内。如果它不再使用，那么它会在其上次引用的Δ时间单位后从工作集合中删除。工作集合是程序局部的近

似。

- 工作集合精度与Δ的选择有关。如果Δ太小，那么它不能包含整个局部；如果Δ太大，那么它可能包含多个局部。如果Δ为无穷大，那么工作集合是进程执行所碰到的所有页的集合。

- 页错误率策略：一种更为直接的方法
  - 建立可接受的页错误率
  - 如果错误率太低，则进程可能有太多的帧，因此应丢弃一些帧
  - 如果错误率太高，则为进程分配更多帧
  - 与工作集合策略一样，也可能必须暂停一个进程。 如果页错误增加且没有可用帧，那么必须选择一个进程暂停。接着，可将释放的帧分配给那些具有高页错误率的进程。

十一、文件系统挂载

基本概念

1、文件基本概念：逻辑存储单元（文件）

- 1）文件的信息被保存在目录结构中，而目录结构也保存在外存上。
- 2）打开文件的信息:
  - 指向文件的指针
  - 打开文件数，当为0时即表示可以删除
  - 文件在磁盘中的位置
  - 文件访问权限

文件访问

1、顺序访问：文件信息按顺序，一个记录接着一个记录地加以处理。（基于文件的磁带模型）

2、直接访问：文件由固定长度的逻辑记录组成，以允许程序按任意顺序进行快速读和写。（基于文件的磁盘模型，磁盘允许对任意文件块进行随机读和写）

-可以用直接访问来模拟顺序访问，而用顺序访问模拟直接访问则是极为低效和笨重的。

3、其他访问：可建立在直接访问方式之上，通常涉及创建文件索引。

- 对于大文件，索引本身可能太大以至于不能保存在内存中。解决方法之一是为索引文件再创建索引。初级索引文件包括二级索引文件的指针，而二级索引再包括真正指向数据项的指针。

文件目录结构

1、目录操作如下：搜索文件、创建文件、列出目录、重命名文件、遍历文件系统。

2、目录结构标准：

- 有效：迅速定位文件
- 命名：方便用户
- 两个不同的用户的文件名称可以相同
- 同一文件可以有不同的名称
- 分组：按文件的属性逻辑分组（如所有java程序，所有游戏等）

3、目录结构：

- Single-level directory
- Two-level directory
- Tree-structured directory
- Acyclic graph directory无环图：无环图是树形结构目录方案的自然扩展
- General graph directory通用图目录：采用无环图结构的一个特别重要的问题是要确保没有环，然而，当对以存在的树结构目录增加链接时，树结构就破坏了，产生了简单的图结构。

文件共享

期望多用户系统上文件的共享，共享可以通过保护机制来实现

文件保护

```
1                       RWX
2   a) owner access     7    -> 1 1 1
3                       RWX
4   b) group access     6    -> 1 1 0
5                       RWX
6   c) public access    1    -> 0 0 1
```

十二、文件系统实现

文件系统结构

1、磁盘提供大量的外存空间来维持文件系统，磁盘的两个特点使其成为存储多个文件的方便媒介

- 可以原地重写；可以从磁盘上读一块，修改该块，并将它写回到原来的位置
- 可以直接访问磁盘上的任意一块信息。（随机或顺序方式）
- 内存与磁盘之间的I/O转移是以块为单位而不是以字节为单位来进行的

2、分层的文件系统：上层需要用到下层的应用

- I/O控制：由设备驱动程序和中断处理程序组成，实现内存与磁盘之间的信息转移
- 基本文件系统：向合适的设备驱动程序发送一般命令就可对磁盘上的物理块进行读写。每个块由其磁盘地址来标识。
- 文件组织模块：知道文件及其逻辑块和物理块，可以将逻辑块地址转换成基本文件系统所用的物理块地址。

3、磁盘结构包括：

- 引导控制块（boot control block）：包括系统从该分区引导操作系统所需的信息。通常为分区的第一块。如果该分区没有OS，则为空。（其他名称：引导块（Linux）、分区引导扇区（WindowsNT））
- 分区控制块（partition control block）：包括分区详细信息，如分区的块数、块的大小、空闲块的数量和指针、空闲FCB的数量和指针等（亦称为超级块（Linux）、主控文件表（WindowsNT））
- 目录结构：用来组织文件
- 文件控制块（FCB）：包括很多文件信息，如文件许可、拥有者、大小和数据块的位置等。UFS称之为索引节点。NTFS将这些信息存在主控制文件表中，主控文件采用关系

为文件分配磁盘空间

常用的磁盘空间分配方法有三个：连续、链接和索引。

1、Contiguous allocation(连续分配)：每个文件占据磁盘上的一组连续的块

- 特点：
  - 简单：只需要记录文件的起始位置（块号）及长度。
  - 访问文件很容易，所需的寻道时间也最少
- 存在的问题

    - 为新文件找空间比较困难（类似于内存分配中的连续内存分配方式）
    - 文件很难增长
    - 可能会出现外部碎片
- 采用一种修正的连续分配方法：该方法开始分配一块连续空间，当空间不够时，另一块被称为扩展的连续空间会添加到原来的分配中。文件块的位置就成为开始地址、块数、加上一个指向下一扩展的指针。每个文件是磁盘块的链表；磁盘块分布在磁盘的任何地方

2、Linked allocation(链接分配)：

- 优点：
    - 简单：只需起始位置
    - 文件创建与增长容易
- 缺点：
    - 不能随机访问
    - 块与块之间的链接指针需要占用空间
    - 簇：将多个连续块组成簇，磁盘以簇为单位进行分配
    - 存在可靠性问题：单链安全性差，当指针链中断时，整体文件损坏

3、Indexed allocation(索引分配)：将所有的数据块指针集中到索引块中（存在计算*1）

- 索引块中的第i个条目指向文件的第i块
- 目录条目包括索引块的地址
- 需要索引块
- 优点：
    - 可以顺序和直接访问
- 缺点：
    - 浪费空间

例题：

```
Mapping from logical to physical in a file of maximum size of 256K words (or 1024KB) and block size of 512 words.
We need only 1 block for index table. (512x2KB=1024KB)

1 word = 4
```

空闲空间管理

1、通常，空闲空间表实现为位图或位向量。每块用一位表示。

- 如果一块为空闲，那么其位为1；
- 如果一块已分配，那么其位为0。
  -当要查找第一空闲块，Macintosh操作系统会按顺序检查位图的每个字以检查其是否为0，再对第一个值为非0的字进行搜索值为1的偏移，该偏移对应着第一个空闲块。该块号码的计算如下：
  （一个字的位数）x（值为0的字数）+第一个值为1的位的偏移

2、链表(空闲链表)：将所有空闲磁盘块用链表连接起来，并将指向第一空闲块的指针保存在磁盘的特殊位置，同时也缓存在内存中。

- 不易得到连续空间
- 没有空间浪费

恢复

1、一致性检查：比较目录中的数据与磁盘中的数据块，以消除不一致性

- 使用系统程序将数据从磁盘备份到其他存储设备（如磁盘，磁带）

2、增量备份：从备份上恢复数据以恢复丢失的文件或磁盘

十三、IO系统##

1、对与计算机相连设备的控制是操作系统设计者的主要任务之一。因为I/O设备在其功能与速度方面存在很大差异，所以需要采用多种方法来控制设备。这些方法形成了I/O子系统的核心，该子系统使内核其他部分不必涉及复杂的I/O设备的管理。

2、I/O控制器和CPU之间的互动模型

- Polling(轮询)
  - 1.主机不断地读取忙位，直到该位被清除 (这个过程称为轮询，亦称忙等待-busy waiting)
  - 2.主机设置命令寄存器中的写位并向数据输出寄存器中写入一个字节。
  - 3.主机设置命令就绪位
  - 4.当控制器注意到命令就绪位已被设置，则设置忙位。

- 5.控制器读取命令寄存器，并看到写入命令。它从数据输出寄存器中读取一个字节，并向设备执行I/O操作。
- 6.控制器清除命令就绪位，清除状态寄存器的故障位以表示设备I/O成功，清除忙位以表示完成。

- Interrupt(中断)
  - CPU硬件有一条中断请求线（interrupt-request line, IRL），由I/O设备触发
  - 设备控制器通过中断请求线发送信号而引起中断，CPU捕获中断并派遣到中断处理程序，中断处理程序通过处理设备来清除中断。

  两种中断请求
  - 非屏蔽中断：主要用来处理如不可恢复内存错误等事件
  - 可屏蔽中断：由CPU在执行关键的不可中断的指令序列前加以屏蔽
- DMA(直接内存访问)
  - 对于需要大量传输的设备，例如磁盘驱动器，如果使用昂贵的通用处理器来观察状态位并按字节来向控制器送入数据（Programming I/O，PIO），那么就浪费了。
  - 许多计算机为了避免用PIO而增加CPU的负担，将一部分任务下放给一个专用处理器，这称为DMA（direct-memory access）控制器。

十四、保护

1、什么是保护：防止用户有意地、恶意地违反访问约束；确保系统中活动的程序组件只以同规定的策略相一致的方式使用系统资源。

2、域的常见操作/一个域可以通过以下几种不同的途径来实现：

- 每个用户是一个域。可以访问的对象集取决于用户的身份。
- 每个进程是一个域。对象集的访问取决于进程的身份。
- 每个过程是一个域。可以访问的对象集对应这个过程中所定义的局部变量。

十五、安全

1、4个层次上采取安全机制：物理、人、网络、操作系统