操作系统概念

2016年10月30日 14:41

1 什么是操作系统？

An OS is a program:

– Manage hardware.

– Provide app.

– Intermediary between user and hardware.

OS is a resource allocator.

OS is control program.

操作系统是一组控制和管理计算机软硬件资源、合理地对各类作业进行调度以及方便用户的程序的集合。

对于操作系统没有一个统一的适用的定义。

2 操作系统设计目标

方便性/有效性/可扩展性/开放性

3 操作系统的发展

3.1 为什么发展？

– 不断地提高计算机资源利用率

– 方便用户

– 器件的不断更新换代

– 计算机系统结构的不断发展

3.2 发展过程

1945-1955 无操作系统时期

1955-1965 批处理系统

1965-1980 分时系统

1980-现在

---实时系统

---嵌入式系统

---分布式系统
---网络系统
---.....

批处理系统Batch System：批处理是指用户将一批作业提交给操作系统后就不再干预，由操作系统控制它们自动运行。
优点：提高系统资源利用率和作业吞吐量。
缺点：批处理系统不具备交互性；作业周转时间长，用户使用不方便。

分时操作系统：采用时间片轮转的方式。具有交互性，多用户同时性和独立性。比如unix。

实时系统：保证即时响应和高可靠性。比如生产过程实时控制，航空系统实时控制，相比于上面两个系统，资源利用率较低。

嵌入式操作系统Embedded Systems：专用性强。

网络系统：通过通信网络将物理上具有自治功能的数据处理系统或者计算机系统互联起来，实现信息交换和资源共享，协同完成任务。

分布式系统：于网络系统有类似之处，但是有一些不同。第一，分布式系统虽然用网络链接，但是没有制定标准协议；第二，分布式系统要求一个统一的操作系统，实现系统操作的统一性；第三，系统具有透明性，系统为用户提供统一界面和接口，用户使用统一界面实现各种功能，但并不知道自己任务在那一台计算机上执行。

4 总结

OS is an software between user and hardware.
OS is designed for **方便性，有效性，可扩展性**，开放性。
OS has developed for several generations: NO OS, Batch, and Time-Sharing.
Computer is activated by interrupt.
Computer includes IO, Memory, CPU.

计算机系统结构

2016年10月30日 20:43

1 计算机系统组织

1.1 中断interrupt

事件的发生通常通过硬件或者软件中断来表示。硬件可以随时通过系统总线向CPU发出信号，来触发中断。软件通过执行特别操作如系统调用system call来触发调用。

硬件：随时中断。
软件：调用系统system call。

1.2 I/O structure

计算机程序必须在内存（随机访问内容random access memory，RAM）中以便于运行。内存是处理器可以直接访问的唯一的大容量存储区域。通常是使用动态随机访问内存DRAM的半导体技术来实现。

load指令将内存中字移到CPU的寄存器中，store能将寄存器的内容移到内存。

缺点：内存终究太小，无法长久存储大量数据和程序；内存是易失性存储。所以需要硬盘作为辅存。

同步I/O：程序中断后要等待I/O完成才能继续执行。
异步I/O：I/O过程中程序继续执行，I/O完成后再做其他处理。

DMA：direct memory access，直接在内存和设备间传输整块设备，无需CPU频繁干涉，有效减少中断次数。

2 操作系统几大功能模块

进程管理：执行中的程序称之为进程。进程需要一定软硬件资源完成其任务。这些资源由操作系统分配。
内存管理：记录内存那些部分正在被使用以及被谁使用；决定那些进程可以装入内存；根据需要分配和释放内存。
存储管理：包括I/O管理、文件管理、辅存管理、高速缓存管理等等。
命令解释系统Command-Interpreter Systems

安全保护：保护：一种控制进程或用户对计算机系统资源的访问限制；安全：防止系统不受外部或者内部攻击。

3 操作系统提供服务

第一类：方便用户，对用户很有用：
用户界面：user interface，UI；包括命令行界面和图形用户界面。
程序执行
I/O操作
文件系统控制
通信：进程间信息交换。
错误检测

第二类：确保系统本身可以高效执行：
资源分配
统计：统计用户资源。
保护和安全

操作系统服务大部分都是通过系统调用来实现的。

4 系统调用system call

系统调用提供了操作系统提供的有效服务界面。**是操作系统系统的服务编程接口。**操作系统的主要功能是为管理硬件资源和为应用程序开发人员提供良好的环境来使应用程序具有更好的兼容性，为了达到这个目的，内核提供一系列具备预定功能的多内核函数，通过一组称为系统调用（system call)的接口呈现给用户。系统调用把应用程序的请求传给内核，调用相应的的内核函数完成所需的处理，将处理结果返回给应用程序。

程序员大部分时候使用高级application program interface (API)来设计程序而非系统调用。**原因如下：可移植性，使用API的程序可以在支持相同API的系统上执行，而无需关注更底层的系统调用的差别；系统调用更加困难和注重细节，而API封装的很好。**

API: Win32 for windows, POSIX API, Java API.

系统调用往往需要参数，向操作系统传递参数的三种方法：

寄存器传递；参数在内存块或者表中，寄存器传递地址；参数被压入堆栈，并由操作系统弹出。

系统调用类型：进程控制；文件管理；设备管理；通信；信息维护。

5 系统程序System Programs

系统提供了一个方便的环境用户程序开发和执行。包括文件管理，状态信息，文件秀嘎，程序语言支持，程序装入和执行，通信。

6 操作系统结构

操作系统是一个庞大而复杂的软件。

简单结构：没有一个准确定义的结构；受限硬件和软件；没有清晰的系统结构。比如MS-DOS，Original UNIX。

分层方法：系统被分为多个层次，每层使用底层提供的服务。

微内核：所有没基本部分从内核中移除，并实现作为系统应用或者用户程序。优点是扩展性，便携性，安全性，可靠性提高；缺点是性能下降。

模块：内核由一系列模块组成，运行时动态链接。

7 虚拟机

虚拟机延伸于前面提及的分层系统结构。**其基本思想是将单个计算机的硬件抽象为几个不同的执行部件**，从而使得每个独立的执行环境似乎都在自己独立的计算机上执行一样。

虚拟机使得在并行不同执行环境时能共享相同硬件。

8 系统生成System Generation

9 系统启动System Boot

装入系统内核以启动计算机过程称为引导。绝大多数计算机系统都有一小块代码，称为引导程序，可以定位内核并载入执行。
---一般存储在rom或者可擦写rom中；
---开机或者重启时载入；
---诊断机器状态；
---初始化硬件；
---载入操作系统内核即操作系统。

进程管理

2016年10月30日 20:54

多个程序同时执行条件：
内存共享；CPU复用；涉及到内存管理，进程管理，文件管理，I/O管理。

程序串行执行：顺序性，封闭性，可再现性。
程序并行执行：间断性，开放性，不可再现性。

1 一些基本概念

如何称呼CPU的活动？分时系统称为用户程序或者任务（tasks），批处理系统使用作业（jobs）。

程序：被动实体，存储在硬盘上的包含一系列指令的文件。

进程：活动实体，程序的执行。

为什么要有进程？
答：多程序同时执行具有间断性，所以要求在进程暂停运行时记录程序的现场，方便下次正确继续运行；多程序同时执行有资源共享，要求能够对资源的共享情况进行记录；为保证程序的正确并发执行，必须将进程执行看成某种对象，对其进行描述并加以控制。再一次强调，进程不是程序，而是程序的执行。

一个进程包括：text section: program code，program counter and other registers，stack，heap，data section。

2 进程特性

动态性：最基本特性；并发性：多道；独立性：独立运行单元；异步性：不可知的速度向前推进；结构特征：程序段、数据段、进程控制块。

3 进程状态

三状态模型：
ready (就绪)：万事俱备，只欠CPU；
running (执行)；
waiting（等待， block-阻塞， sleeping-睡眠）。

五状态模型：
除上面三种基本状态新增：新的状态和终止状态。

4 Process Control Block (PCB)

每个进程在操作系统中使用进程管理模块Process Control Block (PCB)来表示。PCB由操作系统进程管理模块维护，常驻内存；操作系统根据PCB控制管理并发执行的进程。

PCB是进程存在的唯一标志。

PCB包含以下信息：

进程状态：就绪，等待？
进程标号：唯一标识。
程序计数器：程序下一条指令地址。
CPU寄存器：体系结构不同，略有不同。
CPU调度信息：优先级，调度队列指针等等。
内存管理信息：基址，界限寄存器值，页表，段表等。
记账信息：CPU时间，实际使用时间等等。
I/O状态信息：打开文件列表，分配给进程I/O设备表等。

5 进程调度Process Scheduling

多进程并发的目的是为了最大化CPU利用率。time sharing的目的是为了进程间频繁切换，使用户可以和各个进程交互。而调度程序挑选一个适合的进程获得CPU。

5.1 调度队列

作业队列：系统中所有进程队列。
就绪队列：驻留内存中就绪、等待运行的进程序列。
设备队列：等待特定I/O设备的进程列表。

5.2 调度程序

长期调度程序：Long term scheduler (job scheduler)，从磁盘缓冲池中选择一个进程并装入内存准备执行。执行频率较低，一般只有但进程离开系统时，才需要长期调度程序执行，有几分钟之隔。

短期调度程序：Shor term scheduler (CPU scheduler)，从准备执行的进程中选择进程并为之分配CPU。执行频率高。

进程分为I/O密集型和CPU密集型。长期调度程序有必要选择合理的I/O密集型进程和CPU密集型进程的组合，以保证系统平衡，达到最好的性能。

中期调度程序：Medium Term Scheduler，将进程从内存中移除，从而降低索道程序设计的程度。之后进程可以被重新调入内存，并从中断处继续执行。即所谓swapping交换。

6 上下文交换Context Switch

将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，这一任务称为**上下文切换**。

当中断发生，会使CPU从当前任务改变为运行内核子程序。发生中断时，系统需要保存当前运行在CPU中进程的上下文，从而在其处理完后能恢复上下文。

进程上下文使用PCB表示，包括CPU寄存器值，进程状态，和内存管理信息等。通常执行一个状态保存（state save）来保存CPU当前状态，之后执行状态恢复（state restore）重新开始运行。

上下文切换时纯粹开销，严重依赖硬件。

7 进程操作Operations on Processes

在大部分系统中，进程能够同时执行，也能被动态创建和删除。所以系统必须提供进程创建和进程终止的机制。

7.1 进程创建

父进程创建子进程，子进程又创建其他进程，形成一颗进程树。
大部分操作系统使用unique process identifier (PID)来标识进程。

图片pass

子进程和父进程：

资源共享：全共享；部分共享；完全不共享。

执行：父子同步执行；父进程等待子进程终止。

地址空间：子进程复制复制父进程地址空间；child has a program loaded into it (Linux uses exec)。

7.2 进程终止

当进程完成执行最后语句，使用系统调用exit()请求操作系统删除自身时，进程终止。进程可以返回状态值到父进程（wait()），所有资源都会被操作系统释放。kill也可以杀死一些进程。

8 进程通信Interprocess Communication

操作系统中进程基于各种理由，可能需要相互通信：**信息共享；提高运算速度；模块化；方便**。进程间通信即是：进程间共享数据和交换信息。

8.1 共享地址空间

快速，可以用户分享较大量数据和信息；系统调用仅仅用于却确定共享区域，剩余部分要由用户完成。

8.2 消息传递系统

只能用于传输少量数据；容易实现；通过系统调用send，receive来交换信息。

直接通信：通信进程必须明确命名通信的接收者和发送者；在需要通信的没对进程之间自动建立线路；一个线路只与两个进程相关；每对进程之间只有一个线路。
另有一个变形：非对称直接通信。

间接通信：通过“邮箱”来接收和发送信息，每个信箱具有唯一ID，比如send(A, msg), receive(A, msg)；只有两个进程共享一个信箱时，才能建立线路；一个线路可以与两个或更多的进程相关联；两个进程之间也可以由多个线路。

同步/异步（阻塞/非阻塞）：阻塞send，阻塞receive，非阻塞send，非阻塞receive。

缓冲：零容量；有限容量；无限容量。

线程

2016年10月31日 22:47

线程时CPU使用的基本单元，它由线程ID、程序计数器、寄存器集合和栈组成。它与属于同一进程的其他线程共享代码段、数据段和其他操作系统资源，入打开文件，信号等。

一个传统重量级进程只有单个控制线程。如果进程有多个控制线程，那么它能同时做多个任务。

线程优点：响应性更高；资源共享；经济：创建进程所需内存和资源分配更昂贵，而线程共享所属进程内存和资源，所以创建和切换更快；多处理器体系结构的利用：无论有多少CPU，单线程进程只能运行在一个CPU上，在多CPU上使用线程增强了并发功能。

两种线程：用户层用户线程和内存层内核线程；用户线程受内核支持，但是无需内核管理；内核线程由操作系统直接支持和管理。

1 三种模型

1.1 多对一模型

将多个用户级线程映射到一个内核线程。线程管理由线程库在用户空间进行，效率较高。但任一时刻只有一个线程能访问内核。

1.2 一对一模型

每个用户线程映射到一个内核线程。提供比多对一模型更好的并发性能，但是开销大。

1.3 多对多模型

多路复用了许多用户线程到同样数量或者更小数量的内核线程上。没有以上两种模型缺点。

另外还有混合模型：对于某些线程采用一对一模型，另外一些线程采用多对多模型。

2 线程库

为程序员提供创建和管理线程的API。有两种方法实现线程库。

有在用户空间提供一个没有内核支持的库，此库所有代码和数据结构都存在于用户空间中。调用库中的一个函数只是导致了用户空间中的一个本地函数调用，而不是系统调用。

执行一个由操作系统直接支持的内核级库。

三种线程库：POSIX Pthreads（用户级和内核级），Win32 threads（内核级），Java threads（与宿主操作系统有关）。

CPU调度

2016年11月1日 10:18

1 什么是调度？

调度是多道程序操作系统的基础，几乎所有资源在使用之前都需要经过调度。其目的在于在进程之间切换CPU，操作系统得以提高计算机吞吐率，最大化CPU使用效率。

2 CPU-I/O区间周期CPU-IO Burst Cycle

进程执行由CPU执行和I/O等待周期组成。进程在两个状态之间切换。进程执行从CPU区间开始，之后是I/O区间，接着是另一个CPU区间，然后又是I/O区间，如此循环，最后在一个CPU区间通过系统请求来终止执行。

3 CPU调度程序

从内存中选择一个能执行的进程，并为之分配CPU。

4 抢占式调度Preemptive Scheduling

CPU调度在如下四种可能下执行：

一个进程从运行状态切换到等待状态；
一个进程从运行状态切换到就绪状态；
一个进程从等待状态切换到就绪状态；
一个进程终止。

非抢占式调度：只有第一和第四种情况，即CPU种目前没有进程，一个新进程必须被选择。
抢占式调度：四种情况。

4 分派程序Dispatcher

分派程序是一个模块，用于将CPU的控制交给由短期调度程序选择的进程。其功能包括：

上下文切换；切换到用户模式；跳转到用户程序合适位置，以重新启动程序。

分派程序本身停止一个进程并执行另一个进程也需要时间，称为分派延迟。

5 调度准则

调度算法不同，可能对某些进程更有利。其影响很大。准则如下：

CPU使用率；
吞吐量；
周转时间：从进程提交到进程完成所用时间；
等待时间：进程等待时间之和；
响应时间：进程提交到做出第一次响应所花时间。

不同操作系统优化方向不同。

6 调度算法

FCFS Scheduling：先到先服务调度；

SJF Scheduling：最短作业优先调度；

Priority Scheduling：优先级调度，高优先级先分配CPU，相同优先级按照FCFS调度；

Round Robin Scheduling：RR轮转法调度，类似FCFS，但是增加抢占以切换进程；

Mutilevel Queue Scheduling：多级队列调度，将就绪队列划分为多个独立队列，每个队列有自己独立调度算法，队列之间采用固定优先级抢占调度，每个进程根据其属性被永久分配到某一队列；

Mutilevel Feedback Queue Scheduling：多级反馈队列调度，与多级队列调度有相似，只是进程不是固定在某一个队列，而是根据具体状况在多个队列间移动。

7 多处理器调度Multiple-Processor Scheduling

多个CPU情况下，**负载分配**成为可能，但调度问题也会更复杂。

非对称多处理：一个处理器处理所有调度决定I/O处理以及其它系统活动。

对此多处理：每个处理自我调度，进程可能在统一就绪队列种，也有可能每个处理器有各自私有就绪队列。

另外，还有处理器亲和性，负载均衡，对称多线程等问题和概念。

8 线程调度Thread Scheduling

对于支持线程的操作系统而言，系统调度的是内核线程，而不是进程。用户线程由线程库管理，内核并不了解它们。

在多对一以及多对多模型中：

线程库调度用户级线程到一个有效LWP（轻量级进程）上运行，称为**进程竞争范围PCS**，CPU竞争发生在同一进程不同线程之间。但调度用户线程到有效LWP时，并不意味着线程实际就在CPU上运行，需要操作系统将内核线程调度到物理CPU上，此调度采用**系统竞争范围SCS。**

在一对一模型中：

只使用SCS方法。

同步与锁

2016年11月1日 13:10

1 Background

进程之间有直接或者间接的合作，所以当进程并发并共享某一数据时可能导致数据不一致。要维护数据一致性，必须要有一种机制能够保证合作进程并发时以一定循序执行。

```
Race Condition: where several processes access and manipulate the same
data concurrently and the outcome of the execution depends on the
particular order in which the access take place.
```

竞争条件：多个进程并发访问和操作同一数据，执行结果与访问发生的特定的顺序有关。

Critical Resources(临界资源)：在一段时间内只允许一个进程访问的资源。
Critical Section (CS, 临界区)：一段代码，在该区中可能改变共享数据。

必须保证任何两个进程不能同时进入临界区。

临界区问题解答必须满足三个条件：Mutual Exclusion（互斥），Progress（空闲让进），Bounded Waiting（有限等待）。

2 Peterson算法

一个经典的基于软件的临界区问题解答，仅仅关注两个进程情况。下面介绍算法代码和思想。（不能在现代计算机体系架构中保证正确运行，但是作为一种经典思想有必要了解）

两个进程：
P_i and P_j
两个共享数据项：

```
int turn;
boolean flag[2];
```

算法代码如下：

```
do{
    flag[i]=TRUE; #表示i希望进入临界区
    turn=j; #假如此时另一进程j在等待此条件以进入临界区，则此时可以进入
    while(flag[j]&&turn==j); #如果j在临界区内，i必须循环等待，直到条件被破坏
    临界区;
    flag[i]=False; #离开临界区
    剩余区;
}while(TURE)
```

互斥：假设i进入临界区，说明while循环条件不满足，flag[j]不为TRUE或者turn不为j；由于i在尝试进入临界区前已经在临界区外将turn设置为j，若turn条件变化，说明此时j进程在临界区外改变了turn值，j未进入临界区；同理，flag[j]不为TRUE，说明j已经离开临界区，j同样不在临界区内；所以可知i和j不会同时在临界区内。
空闲让进和有限等待更容易证明。

3 硬件同步Synchronization Hardware

锁概念：任何解决临界区问题的方法都需要一个简单工具，锁。进入临界区必须得到锁，退出临界区必须释放锁。

在单处理器环境中，临界区可以很简单解决：在修改共享变量时要禁止中断出现。这样就能确保当前指令序列不会被中断。

在多处理器环境下，禁止中断相当费时，所以有另一种特殊硬件指令可以**原子地检查和修改字的内容或者交换两个字的内容。**

3.1 TestAndSet()

命令定义如下：

```
boolean TestAndSet(boolean *target ) {
     boolean rv= *target;
     *target=TRUE;
     return rv;
}
```

使用如下：

```
do{
     while(TestAndSetLock(&lock);
     临界区;
     lock=false;
}
```

使用时定义一个boolean类型变量lock，初始化为false。当要申请锁时，将lock作为参数传入，经过TestAndSet()指令处理之后，lock值被设置为true，同时指令返回false（lock初始值）。循环结束，进入临界区。此时，如果其他进程要进入临界区，由于lock值被设置为true，所以while循环不会结束，无法进入；直到在临界区内进程离开临界区并将lock设置为false；此时等待进入临界区的进程会将lock设置为true，并结束while循环，进入临界区。

3.2 Swap()

```
void Swap (boolean *a, boolean *b){
```

```
    boolean temp=*a;
    *a=*b;
    *b = temp;
}
```

使用如下：

```
do{
    key=true；
    while(key=true)
        Swap(&lock,&key);
    临界区;
    lock=false;
}while(TRUE)
```

每个进程都有自己一个key变量，初始化为true；lock初始化为true；当进程希望进入临界区时，while必然执行一次，然后交换key，lock值，key被替换为false，lock被替换为true，第二次循环条件不满足，循环结束，进入临界区。此时，如果其他进程希望进入临界区，由于lock值已经被替换为true，和key交换后仍旧是true，循环会一直执行，直到临界区内进程离开，并将lock设置为false。

以上指令虽然做到互斥，但是无法做到有限等待，所以有待进一步改良，详见书本。

4 信号量Semaphores

信号量定义：

```
wait(S){
    while(S<=0)
        ;
    S--
}

signal(S){
    S++;
}
```

```
do{
    wait(mutex);
    临界区;
    signal(mutex);
}while(TRUE)
```

分为计数信号量和二进制信号量。二进制信号量只能为0或者1，也被称为互斥锁，信号量初始化为1。计数信号量值被初始化为可用资源数目，每当进程需要使用资源时，需要对信号量执行wait()操作，减少信号量计数，当进程释放资源时，使用signal()操作增加信号量计数。

5 问题和改进

前面所说信号量的主要缺点是都要求忙等待（busy waiting）。换言之，当一个进程在临界区时，其它试图进入临界区的进程都必须原地不断循环，浪费了CPU时间，这种信号量也称为**自旋锁。**自旋锁有其优点，由于进程一直在执行，所以无需上下文切换，在多CPU系统中常用。

无论如何，忙等依旧是一个需要解决的问题。为解决忙等问题，将信号量定义为一个结构体：

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

每个信号量都有一个整型值和一个进程链表，当一个进程必须等待信号量时，就加入到进程链表上。操作signal()会从等待进程链表中取一个进程以唤醒。信号量操作重定义如下：

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

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

信号量的关键在于能够原子的执行，其实现方法是关中断（单CPU中）。

信号量分类：按照资源数量：**计数信号量，二进制信号量**；**按是否忙等：自旋锁，记录型信号量。**

另外还有死锁问题，后面章节讨论。

6 管程Monitors

一种用于实现进程同步的 高级抽象数据结构。封装了私有数据以及操作数据的公有方法。管程结构确保只有一个进程可以在管程中活动。
但是管程要实现某种特定同步方案，必须一些额外同步机制，由condition结构来实现。程序

员可以定义一个或多个condition变量：

```
condition x,y;
```

condition变量只有wait()和signal()两种操作，当使用wait()方法时：

```
x.wait();
```

调用进程会被挂起，直到其他某个进程调用signal()释放资源：

```
x.signal();
```

调用signal()方法之后，会有一个问题，调用signal()之后，换那个进程执行下去：一，唤醒并等待：进程P执行signal()后离开管程，由另一进程Q进入管程执行，直至Q离开，或者另一个条件满足，P再继续；二，唤醒并继续：进程P继续执行，Q等待P离开。

管程实现还有很多疑问，教材没有看懂。

7 信号量应用实例

有限缓冲问题；读写者问题；哲学家进餐问题；三个问题代码实现。

死锁

2016年11月3日 10:02

一组进程在互相等待对方的资源的情况下同时持有对方所需资源，就会死锁。

1 系统模型

一个系统拥有有限数量资源，分布在若干竞争进程之间。资源分成很多类型，每种类型有一定数量的实例。类型有CPU周期，内存空间，文件，I/O设备等等。将之抽象：

系统有m种资源：$R_1,R_2,R_3,R_4,...,R_m$；

每种资源R_i都有W_i种实例；

资源使用流程：申请，使用，释放。

2 死锁特征

当系统中以下四个条件同时满足，就会引起死锁：

互斥：至少有一个资源必须处于非共享状态；
占有并等待：一个进程必须占有至少一个资源，并等待另一资源，该资源为其他进程占有；
非抢占：资源不能被抢占，只能由持有者释放；
循环等待：相互等待对方资源，都不主动释放。

3 资源分配图

如果分配图没有环，一定没有死锁；如果分配图有环，可能存在死锁。如果环涉及一组资源类型，且每个类型只有一个实例，则发生死锁。

应对死锁有四种解决方案：死锁预防；死锁避免；死锁检测和恢复；死锁忽略。

4 死锁预防Deadlock Prevention

死锁预防提供一组方法，以保证**至少一个**必要条件**不成立**。通过限制资源申请来实现。

4.1 互斥

非共享资源必然涉及互斥条件，而有些资源本身就是非共享的，所以一般不能通过否定互斥条件实现预防死锁。

4.2 占有并等待

保证一个进程在申请其他资源时，不能占有其他资源。有两种协议来实现：一者进程一开始就申请全部资源；一者进程申请资源前先将自身持有所有资源释放。

缺点：资源利用率低；可能产生饥饿。

4.3 非抢占

如果一个进程占有资源并申请一个不能立即分配的资源，那么其现在所持有的资源都可能被抢占。

缺点：只适用与可以保存和恢复的资源，比如CPU、内存等资源，不适用于打印机等资源。

4.4 循环等待

对所有资源进行排序，要求每个进程按递增顺序申请资源。

死锁预防缺点：资源利用率很低很低。

5 死锁避免Deadlock Avoidance

死锁避免算法动态地检测资源分配状态以确保循环等待条件不可能成立。资源分配状态由可用资源，已分配资源及进程最大需求所决定。

5.1 安全状态

如果存在一个安全序列，那么系统处于安全状态。
对于进程序列<P_1,P_2,P_3,...,P_n>，对于每个进程P_i，P_i仍旧需要申请的资源数小于当前可用资源加上所有进程P_j（其中j<i）所占有资源，那当前序列被称为安全序列。

安全状态一定不会死锁；不安全状态下可能死锁，也可能不死锁。死锁避免就是让系统一直保持在安全状态下，以确保死锁一定不会出现。

5.2 资源分配图算法

略。

5.3 银行家算法

5.3.1 数据结构

假设 n 为系统进程个数，m为资源类型种类：

Available：长度为m的向量，表示每种资源的现有实例数量；
Max：n×m矩阵，定义每个进程的最大需求；
Allocation：n×m矩阵，定义每个进程现在所分配的各种资源类型的实例数量；
Need：n×m矩阵，表示每个进程仍需要的资源，可以通过Max-Allocation计算出来。

5.3.2 安全算法

(1)设Work和Finish分别为长度为 m 和 n 的向量。按如下方式进行初始化，
Work=Available，并且对于 i=0,1,…,n-1，有Finish[i]=false。
(2)查找 i，使Finish[i]=false，$Need_i$<=Work；如果 i 不存在，转到第(4)步。
(3)Work=Work + $Allocation_i$，Finish[i]=true；转到第(2)步。
(4)如果对所有 i，Finish[i]=True，那么系统处于安全状态。

5.3.3 资源请求算法

假设 $Request_i$ 为进程 P_i 请求向量则：
(1)如果 $Request_i$ <= $Need_i$，转到第(2)步，否则出错。
(2)如果 $Request_i$ <=Avilable，转到第(3)步。
(3)假定系统可以分配给进程 P_i 所请求资源，并按如下方式修改状态：

$Available=Available-Request_i$；

$Allocation_i=Allocation_i+Request_i$；

$Need_i=Need_i-Request_i$；

(4)使用安全算法判定修改后状态是否是安全状态。

6 死锁检测和恢复

6.1 检测算法

单实例情况下：使用等待图(wait-for graph)，如果图中有环，表示，系统中存在死锁。
等待图：图中每个结点表示一个进程，一个进程结点指向另一个进程结点，表示一个进程正在等待另一个进程释放资源。

在多实例情况下，等待图并不适用。一个新的算法类似于银行家算法。将每个进程当前资源请求状况作为n×m矩阵 Request，然后对矩阵中每个 Request使用类似于考察安全状态的算法检测。

6.2 死锁恢复

两种方法：进程终止和资源抢占。

7 重点回顾

死锁预防四种方法；
死锁避免银行家算法；
死锁恢复两种方法。

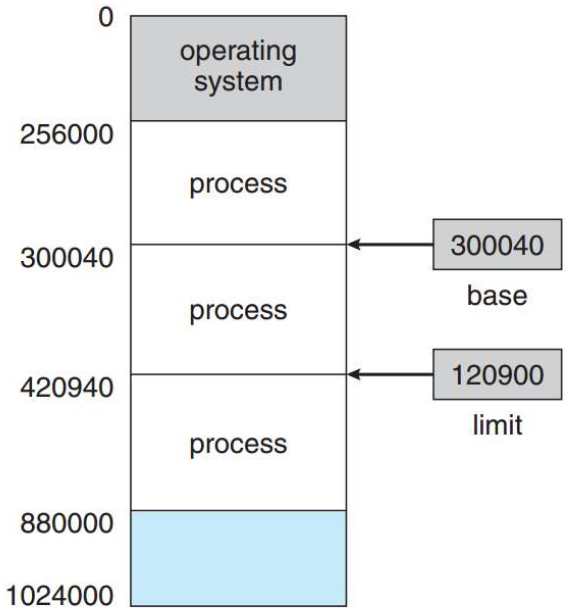
内存系统

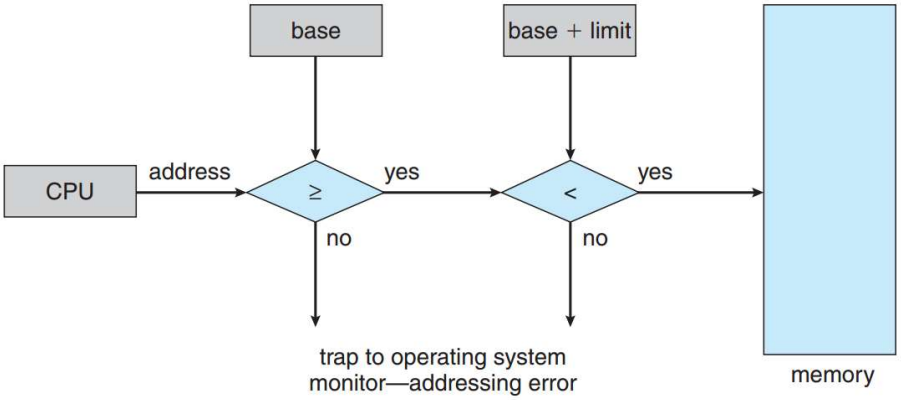
2016年11月8日　　14:35

1 背景相关

1.1 内存保护

CPU取出地址后，与base地址和limit地址对比，如果在给定地址范围之外，说明此是非法地址，则发生异常或陷阱，会直接杀死地址异常的进程。





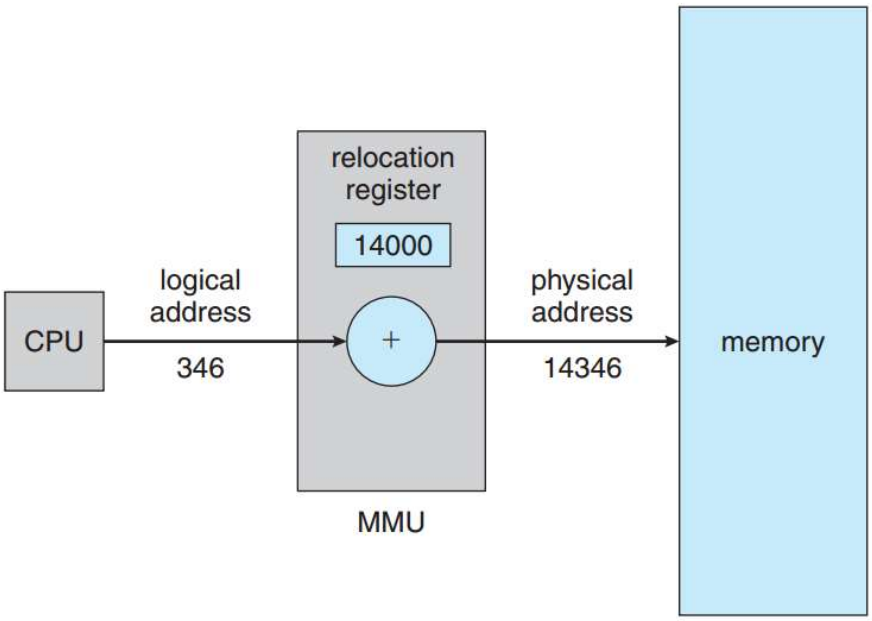
1.2 地址分类

绝对地址（absolute address）：物理地址，内存实际编址。

逻辑地址（logical address）：虚拟地址，由CPU产生和使用，每个进程都有自己独立地址空间，空间大小与计算机位数有关。地址空间大小与内存实际大小没有直接关系。CPU工作时，需要访问数据，直接使用的地址就是逻辑地址。

1.3 虚拟地址到物理地址

由于数据是存储在实际内存上，所以CPU使用逻辑地址访问数据时，必须要将虚拟地址转换为物理地址。虚拟地址绑定到物理地址根据世间不同分为三类：编译时绑定；加载时绑定；执行时绑定（通用操作系统常用）。
编译和加载时地址绑定生成相同的逻辑地址和物理地址。
执行时的地址绑定方法导致不同的逻辑地址和物理地址。由内存管理单元MMU来实现虚拟地址到物理地址转换。



1.4 程序载入和链接

动态加载：一个子程序只有在被调用时才被加载。
动态链接：只有在运行时对应的库才被链接执行。

1.5 交换

将一个进程暂时从内存交换到备份存储中，当需要再次执行时再调回内存中。
• Swap Out
– 1 select a process to be swapped out
– 2 swap out
• Swap In
– 1 Select a process to be swapped in
– 2 allocate memory space and swap

2 连续内存分配

2.1 单一连续分配(Mono-programming memory allocation)

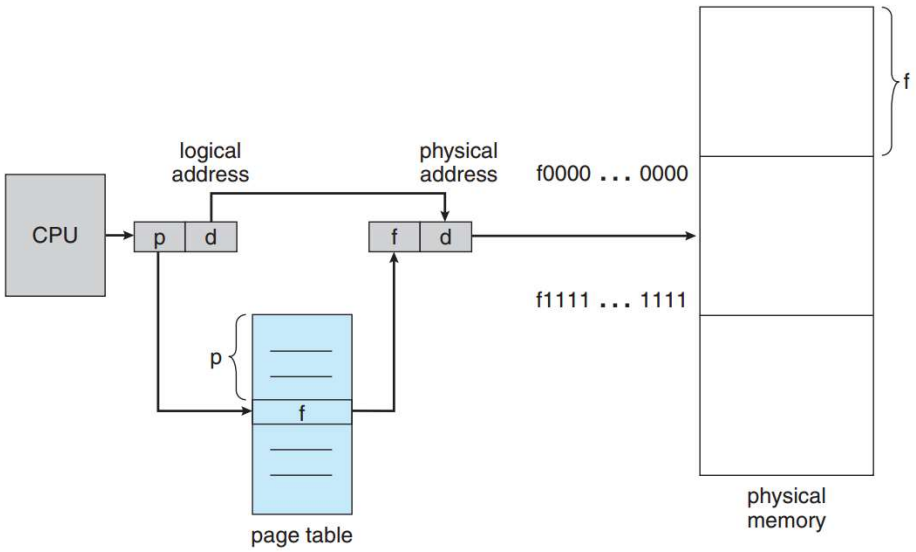
除了操作系统所占用内存空间，剩余空间都是用户空间，然后从头开始连续分配。方法非常简单，但是每次最多处理一个进程。

2.2 分区分配(Multiple-partition allocation)

将用户内存空间分割为n个分区，一个进程占用一个（可能）分区，整个用户内存空间可以被多个进程同时使用。
根据分区大小是否固定可以分为固定分区和动态分区。
固定分区内存利用率低，有内部碎片和外部碎片问题。而动态分区维护一张空闲内存分区表，根据不同的分配算法，又会有不同问题。如首次适应和最佳适应算法都存在外部碎片问题。
解决外部碎片问题的一个方法使紧缩，即将内存内容移动使空闲空间合并成一整块。

3 分页

分页内管理方案将物理内存分为固定大小的块，称之为帧(frame)，而逻辑内存同样分为同样大小的块，称为页(page)。
逻辑地址：由逻辑页号和页内偏移组成。两者位数与逻辑地址空间大小以及页大小有关。
页表：维持一个逻辑页号到物理帧号的映射表，一般而言每个进程都有自己的一张页表。
物理地址：物理页号加上页内（帧内）偏移。
分页内存管理方案没有外部碎片，每个内存帧都可以分配给需要的进程。但是具有内部碎片，即每个内存帧并不总能完全被使用。内部碎片可以通过减小帧大小来解决，但是，这样会增大页表开销。

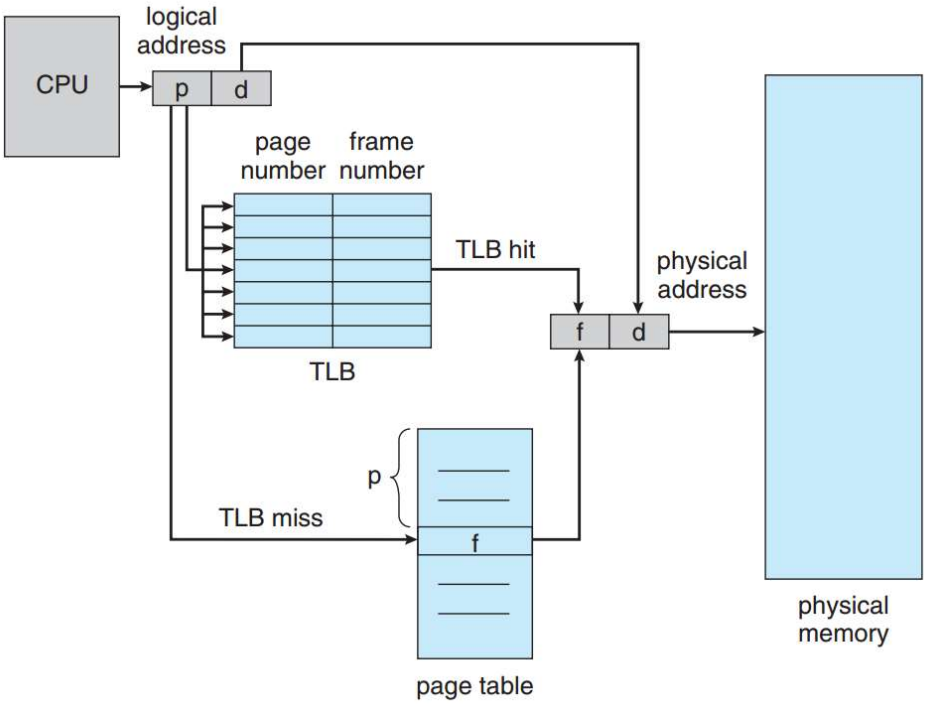


3.1 硬件支持

为了实现页表，可以有多种方式：

- 使用一组专用寄存器来实现，但是现代计算机页表数量相当大，使用专用寄存器来实现并不合理。

- 使用一个寄存器保存页表基址page table base register，一个寄存器保存页表长度，当需要访问页号 i 的内存时，使用基址加上 i 即可访问对应页表条目。
- 使用一个较小但是专用的硬件缓存（转换表缓冲区TLB）来缓存部分页表条目，当在TLB中无法查找时，再到内存中查找。

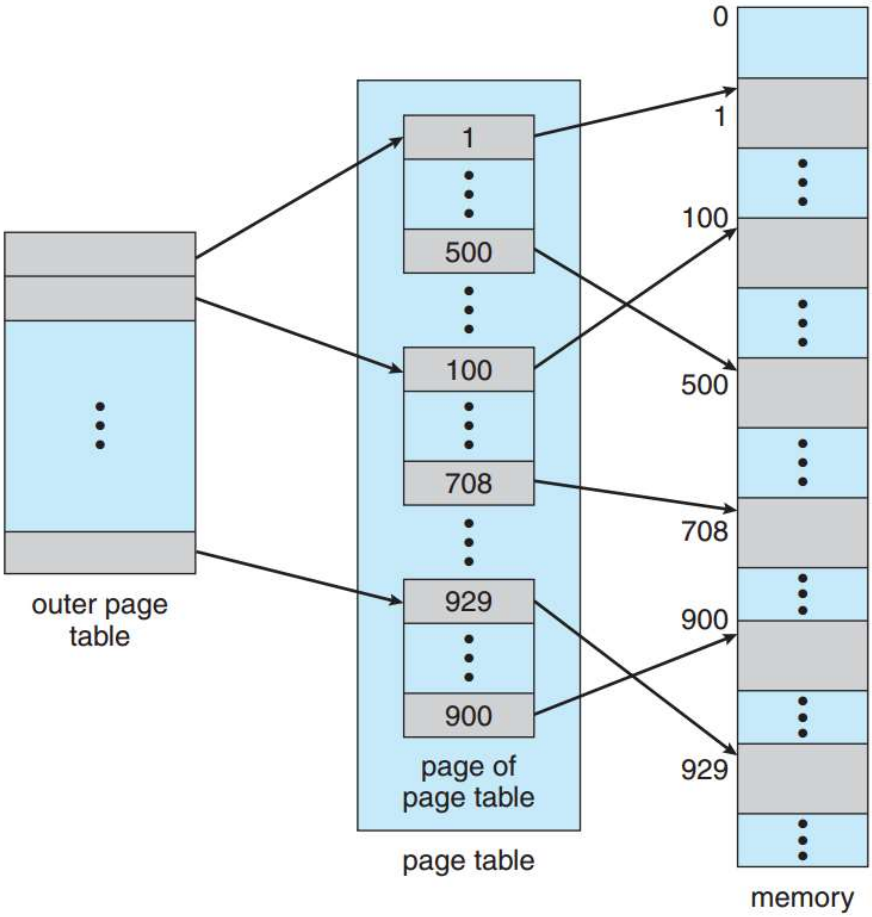


3.2 内存保护

每个物理帧都有一个对应的保护位来实现内存保护，使用这些位的信息来实现读写的权限控制。一般而言，保护位都保存在页表之中。
页表中每条条目还对应有一个有效无效位，表示相关的页是否在对应进程的逻辑地址空间中。

3.3 页表结构

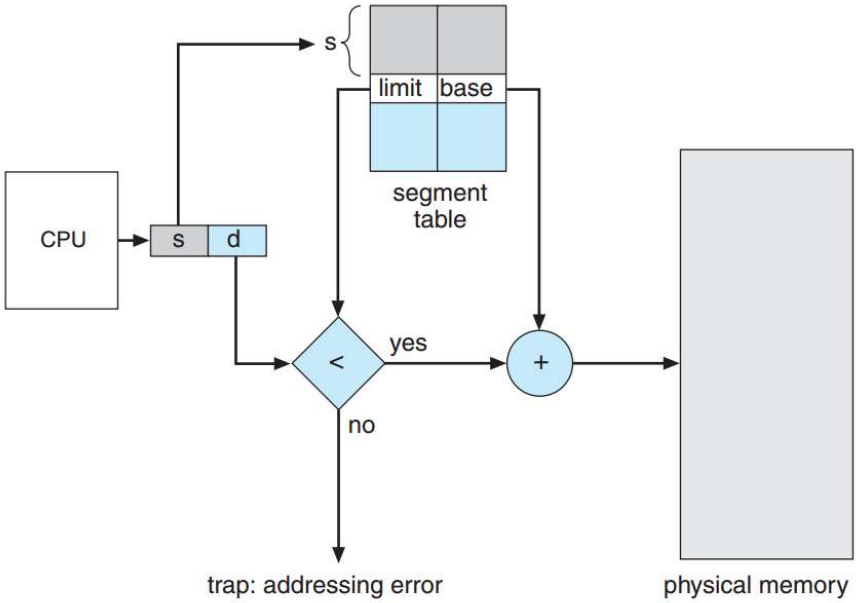
层次页表：将页表再分页，将原本的页号位分成两段，一段作为外部页表页号，一段作为页表内记录的偏移量，最后页内偏移量含义不变。



哈希页表：使用hash表来组织页表。虚拟页号作为key值。每个条目中保存一个或多个虚拟页号到物理帧号的映射对。

4 分段

逻辑地址空间由一组段组成。每个段都有名称和长度。而地址指定了段名称和段内便宜。为了将逻辑地址映射为物理地址，使用段表来实现。

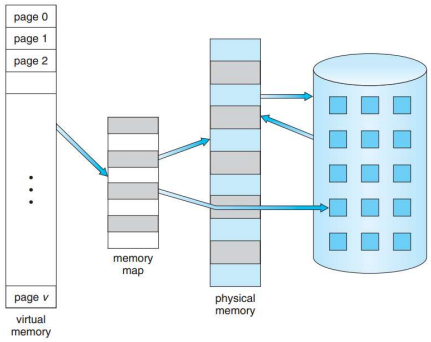


虚拟内存

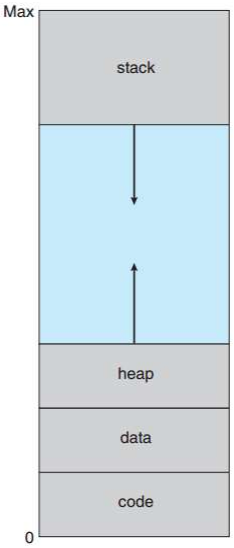
Wednesday, December 21, 2016 9:35 AM

1 虚拟内存

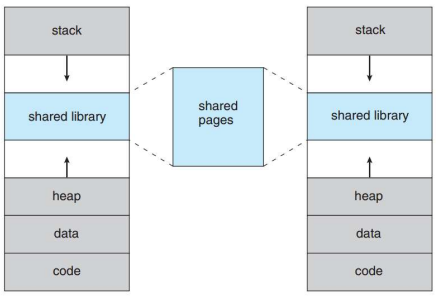
在生产实际中，在大部分情况下，并不需要将整个程序放到内存中。在一个程序中，有许多部分很少使用，即使需要完整程序，也不是同时需要所有程序。
虚拟内存将用户逻辑内存与物理内存分开，两者之间通过内存映射相互联系。虽然物理内存有限，但是却可以向用户提供巨大的虚拟内存。



虚拟内存空间如下图所示。地址从低到高分别存放代码，数据，堆，空白空间，栈。随着动态内存分配，堆可以不断向高地址增长；随着子程序不断调用，栈可以不断向低地址增长。只有在堆栈增长时，才需要实际物理页分配。



由于虚拟内存和物理内存相互分离，所以允许多个虚拟内存（进程）共享同一物理内存中数据内容。比如系统库可以为多个进程所共享，而每个进程都以为共享系统库在自己虚拟地址空间中。同理，多个进程也可以通过虚拟内存实现物理内存共享，进一步实现进程间通信。



2 按需调页

一个程序如何从磁盘载入内存往往有两种选择：在程序执行时将全部程序内容全部载入；另一种，实在需要时才调入相应的页，即按需调页，常为虚拟内存系统所采用。对于按需调页虚拟内存，只有在程序执行需要时才载入页，那些从未访问的页不会调入物理内存。

2.1 按需工作流程

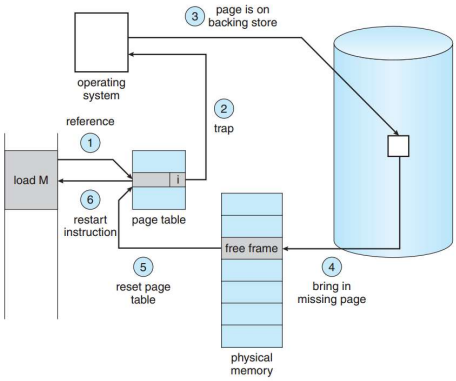
按需调页方案中，一个程序执行步骤如下所示：

- 进程执行；

交换swapper程序是针对整个进程而言，将一个进程调入内存准备执行；而掉页程序是对进程的某个页进行操作。

- 发现某一条指令或者指令数据不在内存中，引发页错误(page fault)；
- 检查进程内部页表，确定引用是否非法；
- 如果引用非法，终止进程；如果引用合法，说明对应页不在内存中；
- 选择一个空闲帧；
- 从磁盘中将需要的页调入刚刚分配的帧中；
- 修改进程内部表和页表，表示页已经在内存中；
- 重新执行引起页错误的指令。

最极端情况，进程一开始执行第一条指令都不在内存中，需要立刻中断并调取相关页到内存中，称之为存粹按需调页。



支持按需调页的硬件包括：

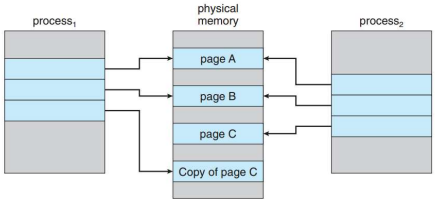
- 页表：将虚拟页号映射到物理帧号，同时通过有效无效位来表示对应的页/帧是否在内存中。
- 次级存储器：保存不在内存中的页。

2.2 缺页故障/页错误(page fault)

effective access time = (1 − p) × ma + p × page fault time #ma为内存访问时间，p为缺页故障率

3 写时复制

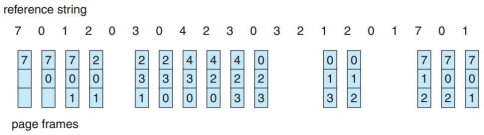
允许父进程和子进程开始时共享同一页面。页面标记为写时复制页，即如果任何一个进程需要对页进程写操作，那么就创建一个共享页副本。



4 页面置换

当发生页缺失时需要从硬盘上将页面调入内存，但是内存容量有限，所以当需要调入页面，但是内存空间不足时，需要选择一些页面作为“牺牲”帧从内存换出。如果“牺牲”帧是脏页，则将“牺牲”帧写回到硬盘，否者直接替换掉，将缺失页从磁盘中读入到内存中。此即为页置换。
当需要页置换时，必须选择要置换的帧，因此，需要有页置换算法。页置换算法，期望能够得到最小错误率。（对于一个内存引用序列，在使用某种置换算法是，缺页故障发生次数最少）

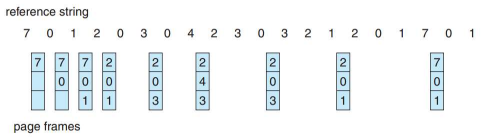
4.2 FIFO



Belady's anomaly：页错误率随着分配帧数增加而增加，违背了原本的内存增加性能改善的希望。

4.3 最优置换(OPT)

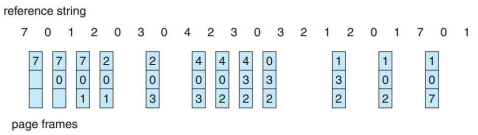
优先替换出未来最长世间内不会使用的页。页错误率最低。不会出现Belady's anomaly。



但是最优置换需要未来信息，所以往往难以实现。

4.4 LRU

最近最少使用的页被替换出去，换言之，最长时间没有使用过的页被替换出去。



4.5 类LRU

包括附加引用位算法；二次机会算法；增强型二次机会算法；等等。

4.6 基于计数的页置换

最不经常使用页置换算法；最常使用页置换算法。

4.7 页缓冲算法

4.8 应用程序与页置换

5 帧分配(Allocation of Frames)

最少帧数：必须有足够数量的帧来容纳所有单个指令所引用的页。

5.1 分配算法

平均分配：n个进程分配m个帧，每个进程分配到m/n个帧。
比例分配：根据进程大小，将可用内存分配给每个进程。
优先级分配：根据进程优先级分配内存。

5.2 全局分配和局部分配

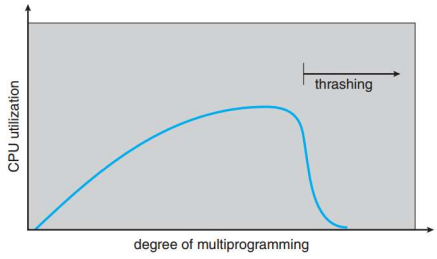
当有多个进程竞争物理帧时，可以将页置换算法分为两类：全局置换和局部置换。全局置换允许一个进程从所有帧集合中选择一个置换帧，而不管这个帧是否已经分配给其他进程；局部置换要求每个进程仅仅能够从自己的分配帧中进行选择。
局部置换策略中，分配给每个进程的帧的数量不变。而采用全部置换，进程可以从其他进程中的帧选择一个进行替换，增加了帧分配数量。全局置换的一个问题是进程无法控制自身页错误率，因为页错误率不仅取决于进程本身行为，同样取决于其他进程行为。

6 页面抖动（系统颠簸）

当进程没有它所需要的活跃使用的帧，那么它很快就会产生页错误。此时，必须置换某个页。然后，所有页都在使用，它置换某一个页后，很快又会需要此页。如此，它会一而再不断产生错误，置换一个页，而该页又立刻出错且需要立即调回。
如此频繁页调度行为即为抖动颠簸(thrashing)。如果一个进程用于换页的时间多余执行时间，那么进程就在抖动。

6.1 原因

- 当进程没有足够页，页错误率会提高，同时导致进程忙于等待页调入调出，CPU利用率下降；
- CPU利用率下降，系统会进一步调入新进程；
- 新进程进入内存，页错误率情况更严重，CPU利用率进一步下降；
- 如此恶性循环，即为系统颠簸。



6.2 解决

第一种方法：局部置换算法，即使一个进程开始抖动，但是由于只能置换自己的帧，所以不会直接影响到其他进程。但这种方法无法完全限制系统颠簸。
第二种方法：提供进程所需要的足够多的帧。问题在于如何确定进程需要多少帧。

6.3 工作集合策略

为了彻底解决系统抖动问题，需要确定一个进程究竟需要多少个帧。
局部模型：一个进程执行时，从一个局部移向另一个局部。局部是一个经常使用的页的集合。一个程序通常由多个不同局部组成，他们可能会重叠。
工作集合模型基于局部性假设（局部模型）。模型使用参数n来确定工作集合窗口。系统检查最近n个页引用，此n个页集合称之为工作集合。如果一个页正在使用，它便在工作集合内，如果不再使用，它就在上一次引用的n时间单位后从工作集合中删除。

确定参数n之后，系统会跟踪每个进程的工作集合，并为之分配大于其工作集合的帧数。如果分配所有进程所需帧后，仍旧有空闲帧，可以启动一个新进程；如果所有进程工作集合之和大于可用帧数，则需要暂停一个进程。

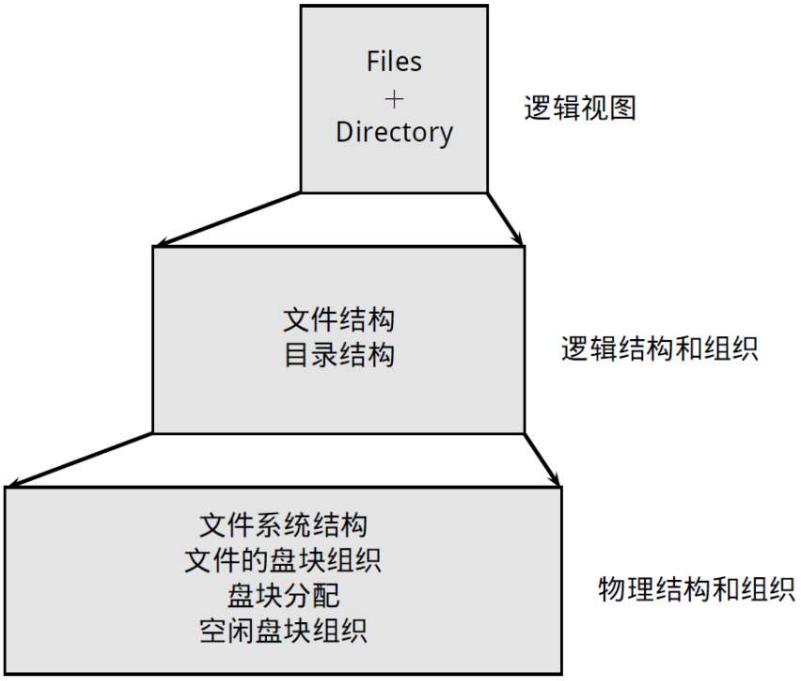
6.4 页错误频率

如果页错误频率很高，说明进程帧数不够，进行需要更多帧；如果页错误频率很低，说明进程帧数很多。所以可以通过控制页错误频率的上限和下限来实现对颠簸的控制。

文件系统

Wednesday, December 21, 2016 11:31 AM

文件系统层级如图所示：



1 文件概念

操作系统提供了信息存储的统一逻辑接口而不需要在意物理存储介质的不同。

- 文件是记录在外存上的相关信息的具有名称的集合。对于用户而言，文件是逻辑外存的最小分配单元（文件级别接口）。
- 文件主要包括程序和数据两种类型。
- 一个文件一般由数据位、字节、行或者记录组成。
- 文件具有一定结构，其结构由文件类型决定。

1.1 文件属性

文件具有/名称/标识符/类型/位置/大小/保护/时间、日期和用户标识/等属性。
一般而言，文件相关信息保存在文件目录结构之中。目录记录中一般直接只记录文件名以及文件唯一标识符。之后通过标识符来定位文件其他属性。

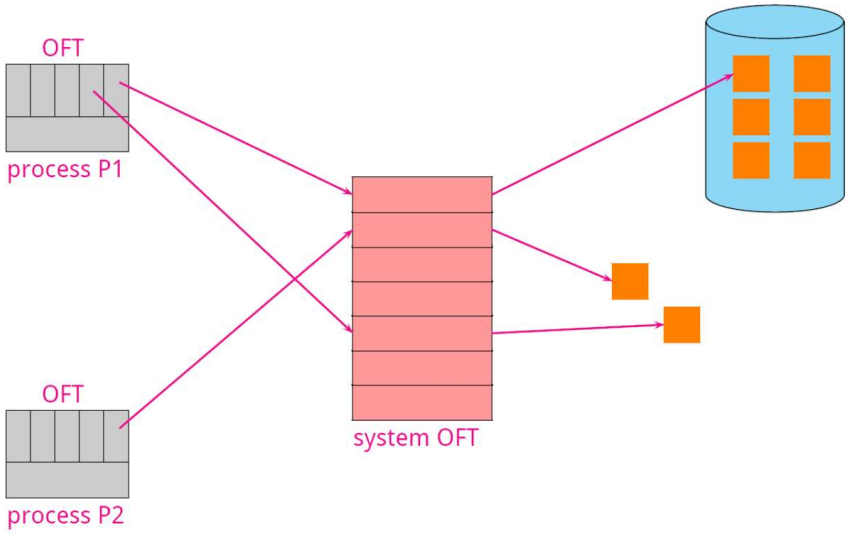
1.2 文件操作

文件具有/创建文件/写文件/读文件/在文件内重定位/删除文件/截短文件/等操作。大部分操作系统要使用或操作一个文件需要显式调用open()来打开文件。系统将所有打开文件信息记录在打开文件表中。由于一个进程可能打开多个文件，一个文件也可能为多个进程打开，所以系

统维护两类表：

- 单个进程打开文件表：包括单个进程打开的所有文件的相关信息，如文件当前文件指针，访问权限等；
- 系统打开文件表：包括系统当前打开的所有文件信息，记录文件位置，访问时间和文件大小等与进程无关的信息。

一般而言，一个打开文件具有以下相关信息：
文件指针：记录在单个进程打开文件表中，记录某个进程对一个文件的当前读写位置；
文件打开计数器：文件可能被多个进程打开，只有计数器为0，系统打开文件表条目才可以被删除；
文件磁盘位置：保存文件在磁盘上位置信息，避免每次要定位文件时都需要访问磁盘目录信息；
访问权限：每个进程对于同一文件可能有不同的访问模式。

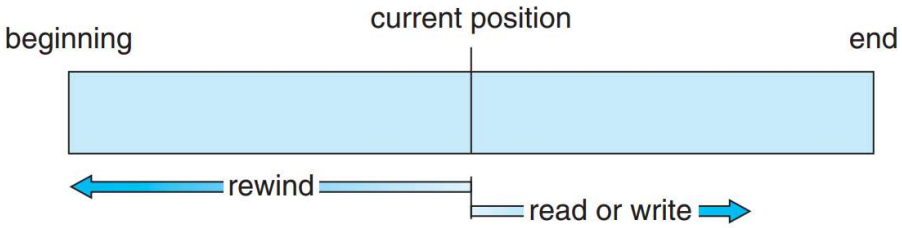


1.3 文件类型与文件结构

文件内部结构不同，自然就有不同文件类型。而操作系统只有能够识别文件类型，才能按合理方式进行操作。一般而言，常常将文件名分为两个部分，文件名称和扩展名。扩展名用于表示文件类型。
操作系统支持的文件结构（文件类型）越多，操作系统越大；另外，也有许多新类型不断出现。所以一般操作系统只支持部分或者最少量文件类型，其他的文件结构识别全部由应用来支持。操作系统只需要使用正确的应用打开文件即可，而文件扩展名可以帮助操作系统选择一个适合该文件结构（类型）的应用。
（所有操作系统至少都需要支持可执行文件结构）

2 访问方式

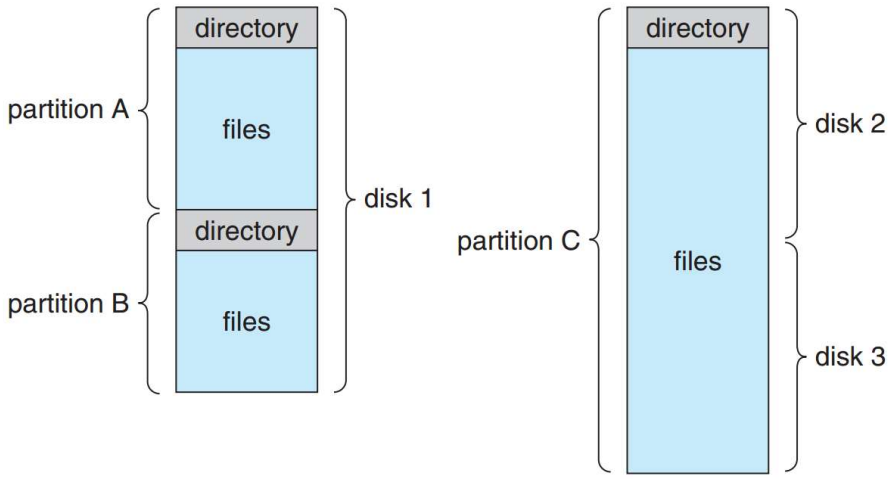
顺序访问：一个记录一个记录加以处理。基于文件的磁带模型。



直接访问：程序可以按任意顺序进行快速读和写。基于文件磁盘模型。

索引访问：基于直接访问方式。为文件建立索引文件，在访问文件前，先查找索引文件。

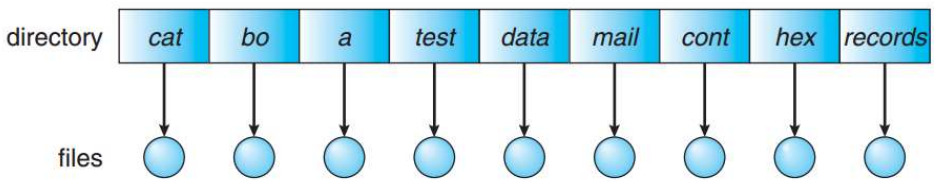
3 目录结构



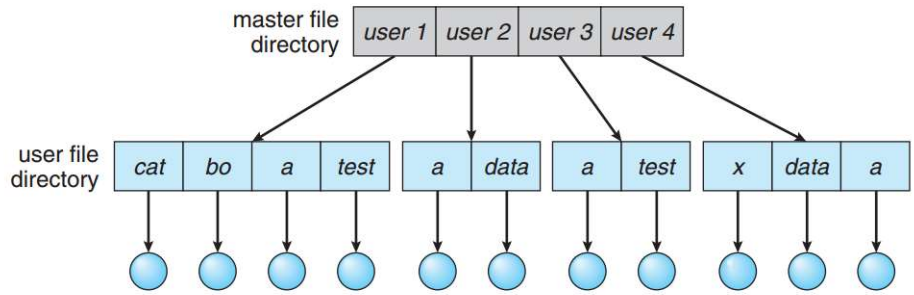
目录是包含所有文件相关信息的节点（记录）的集合。目录和文件都存储在磁盘上。目录之中包含文件名，与指向文件控制块FCB的指针（或者是其他结构，但必须能够定位FCB，比如在linux中是inumber）。FCB文件控制块中包括文件名，文件类型，文件地址，文件当前长度，文件最长长度，最近访问，最近修改，所有者ID，访问权限等等信息。目录相关操作有：

- 搜索文件；
- 创建文件；
- 删除文件；
- 遍历目录；
- 重命名文件；
- 跟踪文件系统。

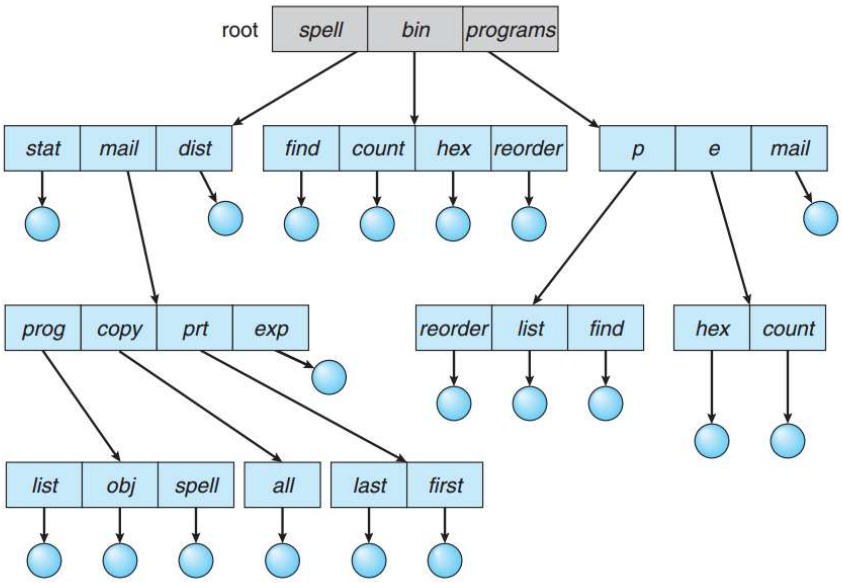
3.1 单层目录结构



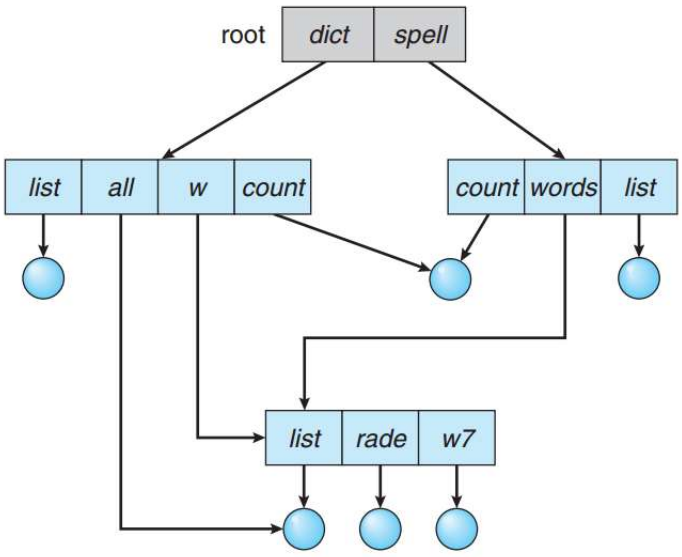
3.2 双层目录结构



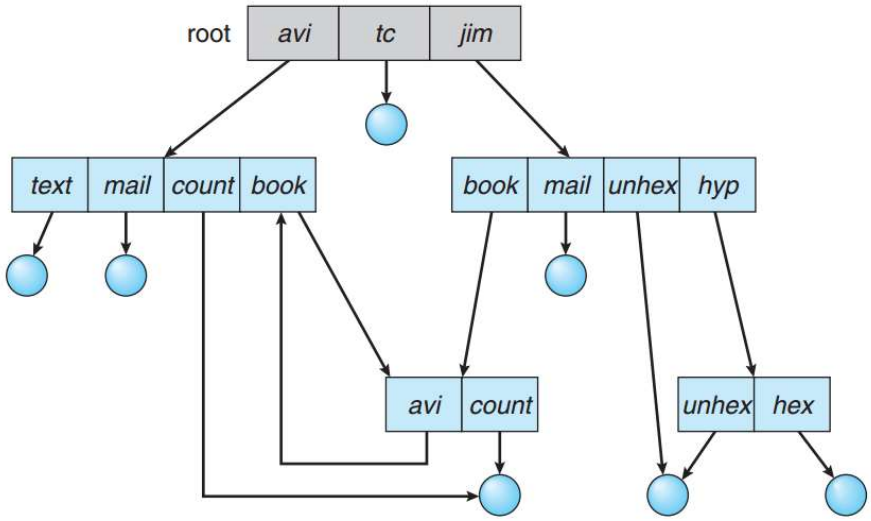
3.3 树状结构目录



3.4 无环图目录



3.5 通用图目录



4 文件系统安装

文件系统在被系统上的进程使用之前必须安装（mount）。具体的说，目录结构可以建立在多个卷上，卷必须安装以使他们在文件系统命名空间中可用。即操作系统需要知道设备名称和安装位置（安装点）。

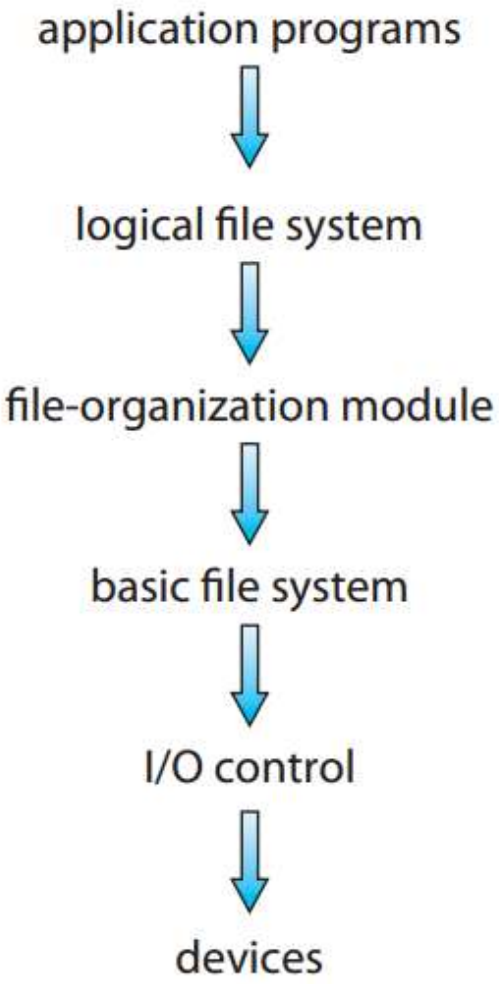
5 访问控制

操作系统为每个文件采用了三种用户类型：

- 拥有者：创建文件的用户，可以对文件进行所有操作（读，写，执行，添加，删除，列表清单）。
- 组：一组需要共享文件且具有相似访问权限的用户形成的工作组。
- 其他：其他用户。

文件系统的具体实现

Friday, December 23, 2016 9:51 AM



文件系统设计包含两个问题：第一，如何定义文件以及其属性、文件允许的操作、组织文件的目录结构等等；第二，创建数据结构以及算法来将逻辑文件系统映射到物理外存设备。
文件系统通常由多个层次组成（如上图）：

- I/O控制：实现内存与磁盘之间信息传输。
- 基本文件系统：向合适设备驱动程序发送一般命令。
- 文件组织模块：将文件逻辑块地址转换成物理地址。
- 逻辑文件系统：管理元数据。

1 文件系统

1.1 硬盘上数据结构

- boot control block引导控制块：包含操作系统自举所需要的信息，通常为卷的第一

块。

- volume control block卷控制块：包含卷的详细信息，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。在UFS（UNIX文件系统）中，卷控制块又称之为超级块superblock，而在NTFS中，它存储在主控文件表（Master File Table）中。
- 目录结构：UFS中，包括文件名和对应的索引节点号。
- 每个文件FCB：包括文件详细信息。

1.2 内存中数据结构

- 安装表：包含所有安装卷信息。
- 目录结构缓存：保存进来访问过的目录信息。
- 系统打开文件表。
- 单进程打开文件表。

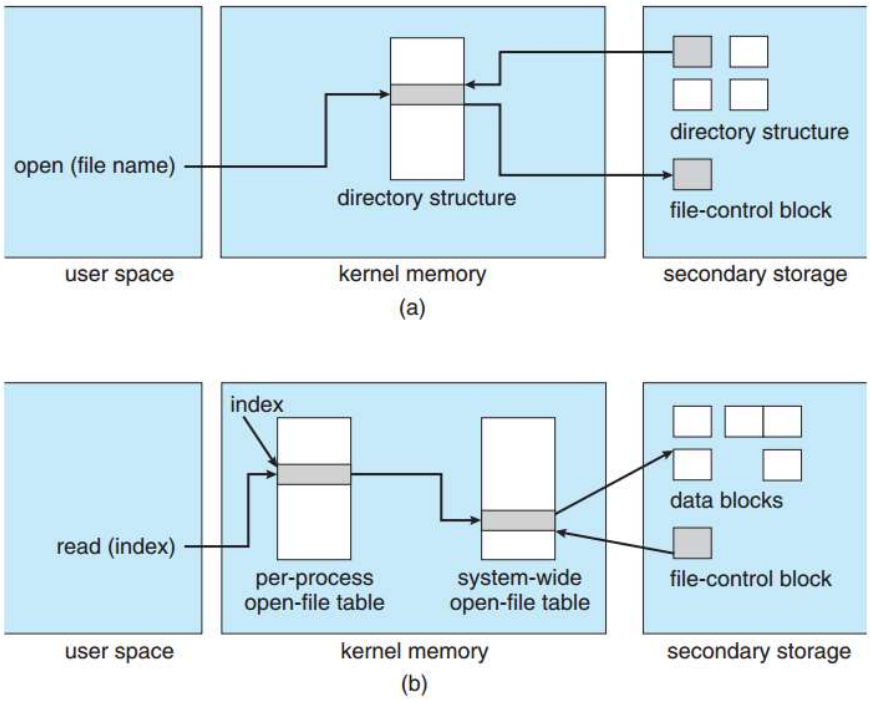
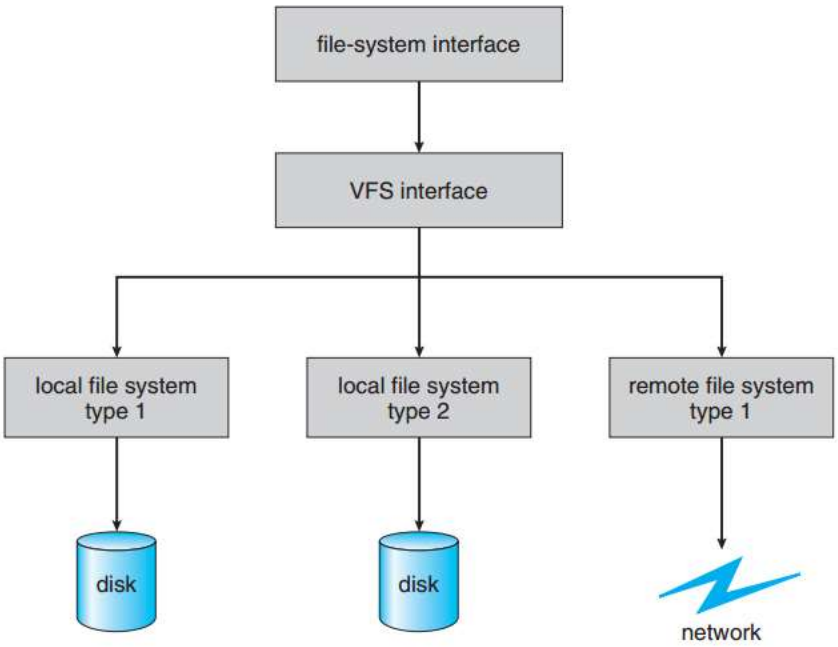


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

2 虚拟文件系统

现代操作系统往往必须同时支持多个文件系统类型。为了能将多个文件系统整合成一个目录结构，并使用户能够在多个文件系统间无缝切换，所以在内存中使用一个虚拟文件系统。



linux中VFS结构主要定义4种主要对象类型：

- 索引节点对象；
- 文件对象；
- 超级块对象；
- 目录条目对象。

3 目录实现

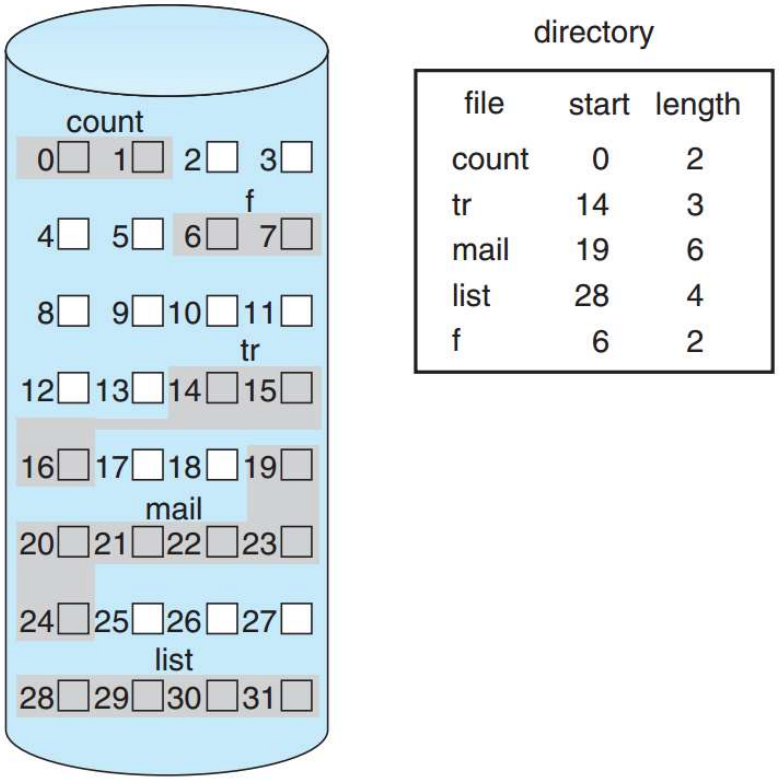
目录条目包含文件名以及对应FCB索引（linux中即为inumber）。为了实现目录在硬盘上结构，需要对目录结构进行设计。目录分配和管理算法直接影响到文件系统效率性能和可靠性。

- 线性列表：优点：实现简单；缺点：查找和创建文件都需要线性搜索文件列表。
- 哈希表：优点：搜索更快；缺点：哈希表大小有一点限制。

4 空间分配

4.1 连续分配

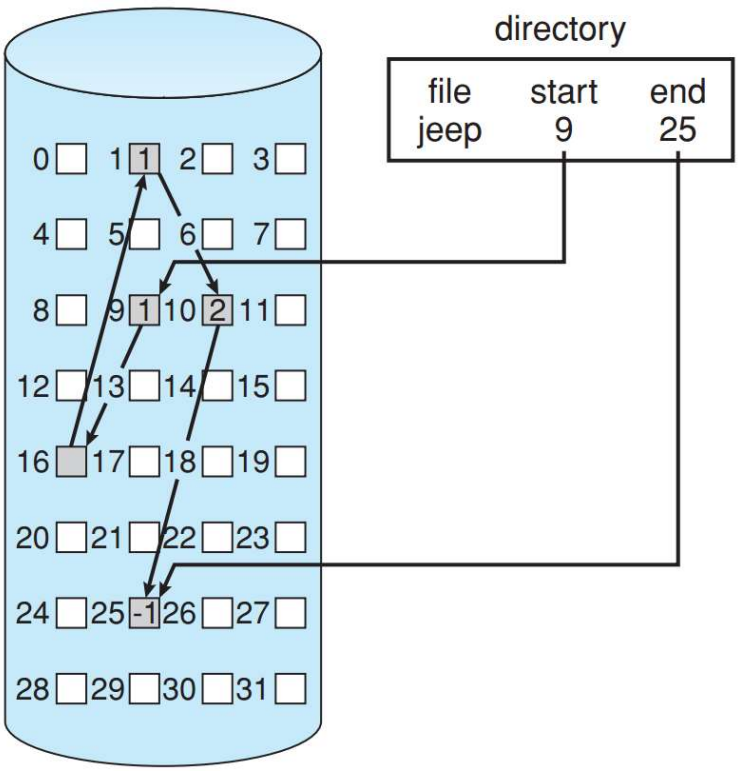
每个文件都为它分配连续的空间。
优点：文件写和读都很快；便于管理文件所在位置，只需要文件首地址和文件长度。
缺点：很难为新文件找到空间（从一组空闲空间中找到合适的连续空间分配出去，但是基本都存在碎片问题），而且一旦文件得到所需空间，后续要增长很难。



4.2 链接分配

采用链接分配，每个文件都是磁盘块的链表；磁盘块可以分布在磁盘的任何地方。

`implicit link`：目录中包含首个指针和末尾指针；每个块包含指向下一个块的指针。
优点：简单，没有空间浪费。
缺点：无法随机访问数据。

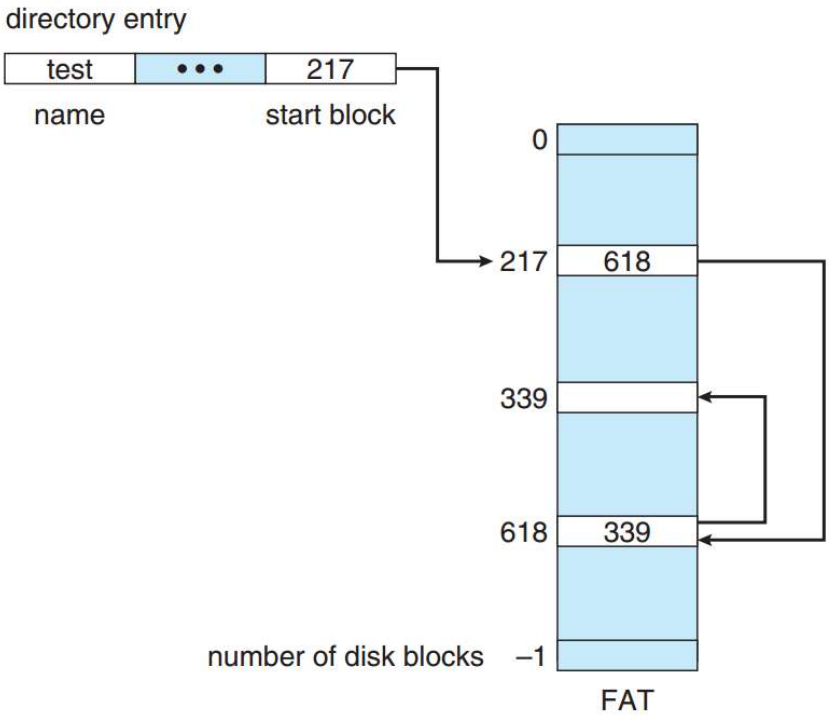


`explicit link`：使用文件分配表来索引文件位置。文件分配FAT存储在磁盘分区的开始部

分。目录条目包含首块的块号码，每个块号码对应条目中记录下一块的块号码。在访问文件之前先读取文件分配表，就可以实现随机读取，但效率较低，而且需要缓存FAT，否者就要花费大量磁头寻道时间来都FAT。

优点：实现了数据的随机访问。

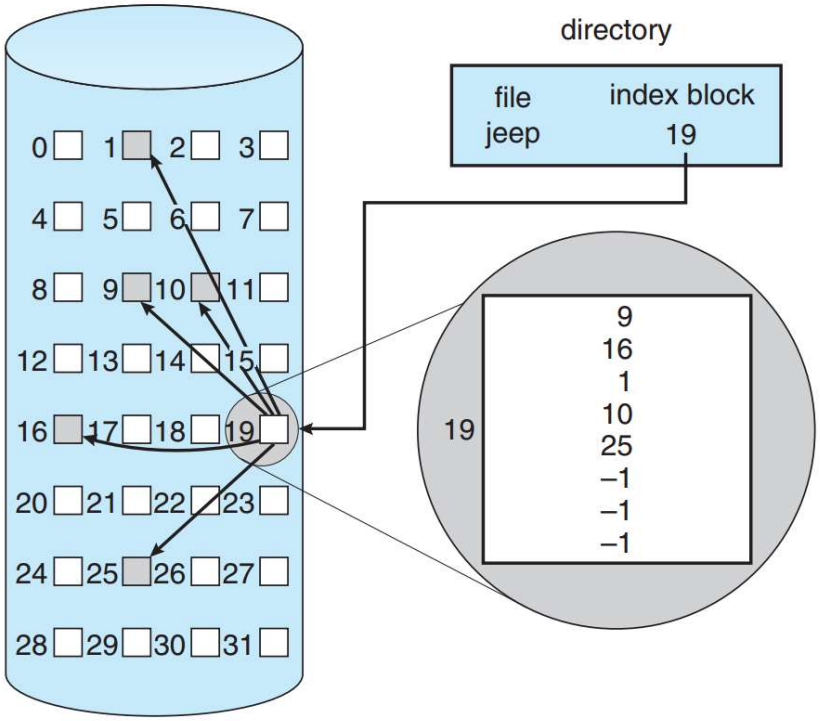
缺点：效率较低，需要缓存FAT。



4.3 索引分配

索引分配将所有块指针放在一起，通过索引块解决了连续分配的碎片和大小声明问题，也解决了链接分配的无法随机访问或者随机访问效率低的问题。

在索引分配中，目录记录了索引块的指针，而在索引块中则记录了文件块指针。

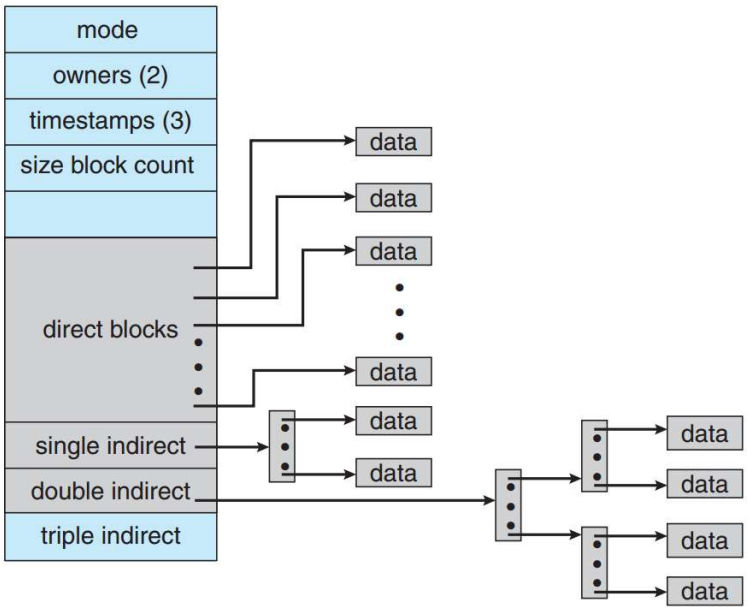


为了解决索引块大小问题提出了三种方案：

链接方案：一个索引快包含文件部分块指针地址以及指向下一个索引块的地址。

多层索引：一个索引块包含一组指向第二层索引块的地址，第二层的索引块只想文件数据块。依此原理，也可以三层索引甚至更多层索引。但是会增加访问的次数。

组合方案：部分数据直接指向，部分数据通过简介索引块指向。性能有一定保证，同时也可以适应多种文件大小。比如linux中索引节点就是通过这种方案实现的。



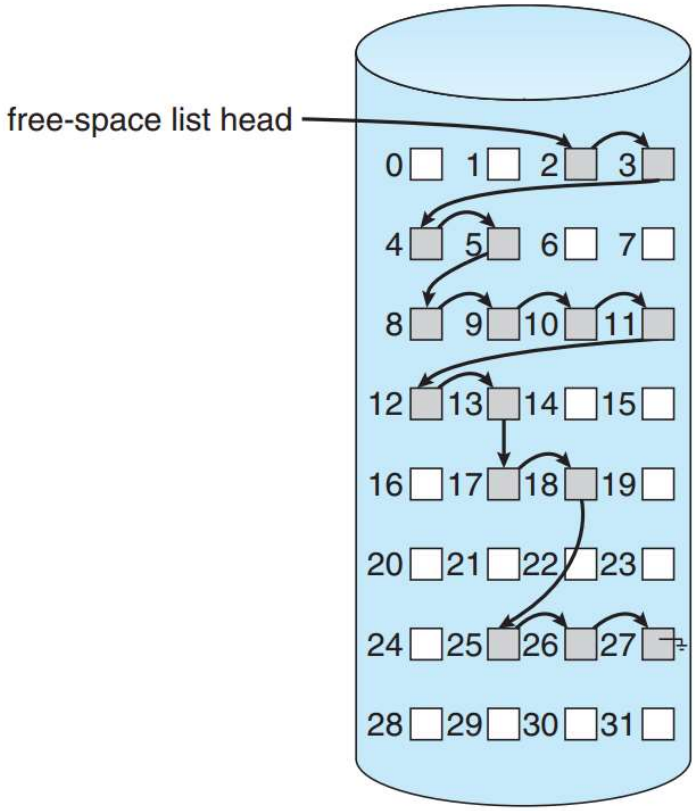
5 空闲空间管理

5.1 位图

用一个bit来表示一个块是否空闲。优点：查找第一个空闲块，或者n个连续空闲块时都相对简单和高效。缺点：当硬盘空间很大时，很难将整个bit map保存在内存中，效率会降低。

5.2 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，同时缓存在内存中。



5.3 组

将n个空闲块的地址存在第一个空闲块中。这些块的前n-1个确实为空，而最后一个块包含另外n个空闲块的地址。

5.4 计数

并非记录n个空闲块的地址，而是记录第一个空闲块的地址以及紧跟第一块的后续的连续空闲块的数量n。

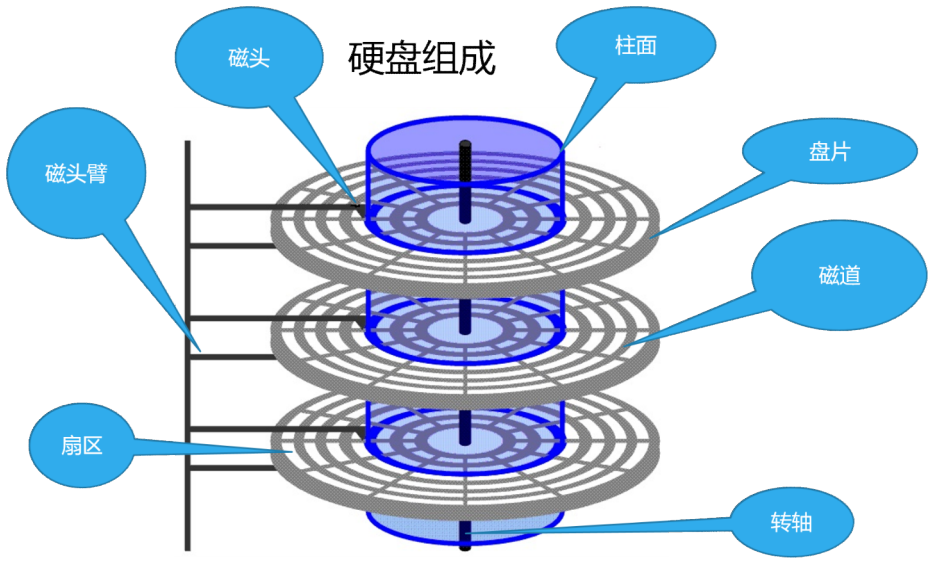
磁盘

Friday, December 23, 2016 12:40 PM

一个硬盘主要由一个或多个盘片组成，每个盘片都有两个磁盘面。盘片一般由硬质材料（如铝）制造，然后再表面涂上一层磁性材料。磁性不依赖电力维持，所以即使断电，也可以保存数据。

盘片都装在同一根轴上，轴联接电机，通过轴的带动，电机使盘片高速转动。

硬盘读写操作通过磁头完成。磁头通过电磁流改变磁性材料极性写入数据，通过相反方式读取数据。磁头通过磁头臂移动位置。



为了访问硬盘中数据，必须通过三个阶段：寻道、旋转延迟以及数据传输：

- 待访问扇区和磁头所在扇区不一定在同一磁道，所以要访问扇区，第一步就是要找到正确磁道。这个过程即为寻道，所费时间为寻道时间。寻道是成本最高的硬盘操作之一。
- 在找到正确磁道后，工作仍未结束，因为并不能保证磁头下就是待访问扇区。所以要等待待访问扇区旋转到磁头所在位置（为此有了磁道偏移）。这就是旋转延迟。
- 最后，一切都已妥当，才开始数据传输，所用时间是传输时间。

I/O操作成本相当高，如果不能减少I/O操作，可以通过对I/O操作的顺序进行合理安排以降低降低成本（主要时间成本），即磁盘调度：

- SSTF调度算法：SSTF算法根据磁头与被请求磁道距离决定处理顺序，此算法会首先处理距离磁头最近的磁道上的请求。（磁道最近，寻道时间自然最短）

- SCAN调度算法：磁头扫过整个磁盘，然后根据扫过磁道的顺序为各个请求提供服务。单躺扫过整个磁盘即为一次Sweep。假如，A I/O请求是对X磁道Y块进行操作，磁盘处理此次操作后之后，在同一Sweep范围内，B 请求又要求对Y块读写，那么磁盘不会立刻处理B请求，而是将它留待下一趟Sweep处理。