傻逼题

device controller、spolling、user thread、safe state、TLB、VFS、device controller、绑定（编译、加载、执行时）

刷题

进程

1. 并发进程失去封闭性是指并发进程共享变量，其执行结果与速度有关
2. 同一程序经过多次创建，运行在不同数据集上形成了不同的进程
3. 某一个线程被不同进程所调用，它们是相同的线程。
    - 程序是由代码+数据构成的，代码经过多次创建可以对应不同进程。同一个系统的进程（或线程）可以由系统调用的方法被不同的进程（或线程）多次使用。
4. PCB所包含的内容：
    1. 进程描述信息 ：进程标识符、用户标识符
    2. 进程控制和管理信息 ：进程状态、进程优先级、代码运行入口地址、程序外存地址、进入内存时间、处理机占用时间、信号量使用
    3. 资源分配清单 ：代码段指针、数据段指针、堆栈段指针、文件描述符、键鼠
    4. 处理机相关信息 ：通用寄存器值、地址寄存器值、控制寄存器值、标志寄存器值、状态字。
5. 进程五个状态中，从运行态到阻塞态由其自身决定

处理机调度

1. 时间片轮转法（RR）是绝对可抢占的。

进程同步

1. 临界区是指 进程中用于访问共享资源的那段代码
2. 进程的 并发执行不需要信号量 就能实现
3. 一个正在 访问临界资源的进程 由于申请等待I/O而被 中断 ，则 允许 其他进程抢占 处理器，但 不得进入 该进程的 临界区 。
4. 在操作系统中， P、V操作是一种低级进程通信原语 。
5. 原语（Primitive/Atomic Action）是不可分割的指令序列。
6. 有 n 个进程的系统，阻塞队列中最多有 n 个进程[死锁]
7. 进程间的制约关系有同步和互斥。同步[使用数值信号量的]、互斥[使用01信号量的]

死锁

1. 死锁的 避免 是根据 防止系统进入不安全状态 来实现的
2. 解除死锁通常 不采用 从非死锁进程处 抢夺资源 的方法。
3. 死锁的四个条件中， 无法破坏 的是 互斥 使用资源。[有些资源根本不能同时访问，如 打印机]

内存管理

1. 只有分段式管理没有内部碎片
2. 操作系统实现分区管理的代价最小
3. 多级页表能够减少页表所占连续内存空间

简介

COMPUTER-SYSTEM STRUCTURES

计算机系统操作、I/O 结构、存储结构、存储体系、硬件保护、通用系统架构

分层法的主要优点是模块化。选择了分层，这样每层只能利用较低层的功能（或操作）和服务。

进程

- CPU在批处理系统（ Batch ）中的活动叫 执行作业（ Job ）
- CPU在分时系统中叫 使用用户程序（ system/user programs ）或任务（ task ）

这些活动在许多方面有着相似之处，统称为进程。

一个具有某一项功能的程序在某一个数据集上的一次执行过程，进程是执行中的程序。

进程是计算机系统资源分配和调度分配的独立个体。

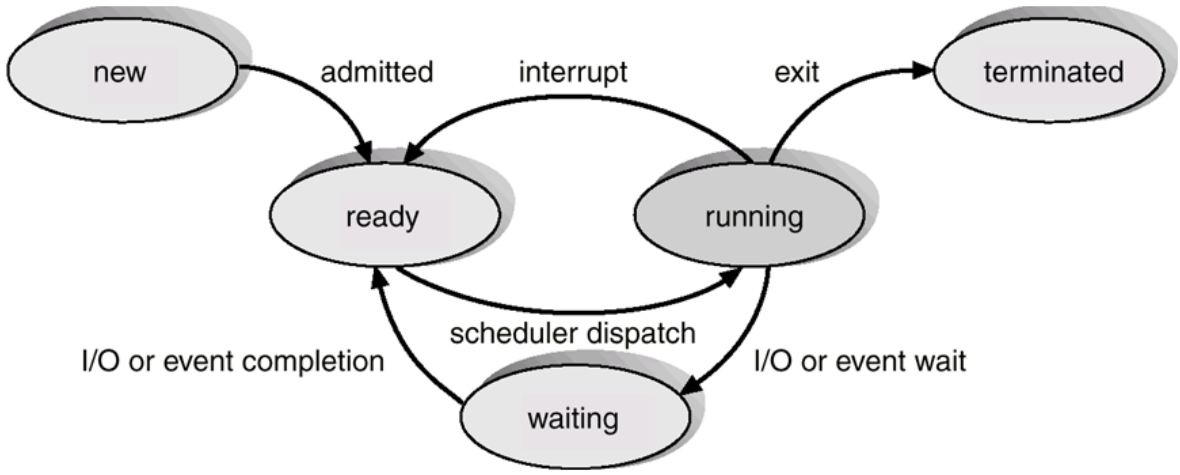
关于进程和程序

- 程序 （Program）:程序是一个 被动 的实体，是 保存在磁盘中 的程序文件，并不是一个运行的过程，是一个静态的文件
- 进程 （Process）：是一个动态的、 主动 的实体，是 正在运行 的程序，程序在运行时PCB中的数据是高速变化的。
- 二者的 关系 ： 多个进程可以对应一个程序。
- 进程和程序的根本区别： 动态和静态的特点

进程的5个状态

- new ： 进程正在被创建
- running ： 指令正在被执行
- waiting ： 进程等待某些事件的发生（如：I/O完成）
- ready ： 进程等待被分配给处理器。（得到后即可执行）
- terminated ： 进程已被执行完成

五个状态间的关系



PCB

PCB是指进程控制块，控制进程的各种状态和信息，是一种数据格式，是进程实体的一部分；当系统创建一个进程时候，同时创建PCB，进程结束时，回收PCB。**PCB是进程存在的唯一标志**

队列

一个进程在执行过程中在各种队列里迁移。

Job queue

系统中的全部进程的集合

Ready queue

主存中的所有进程的集合，准备好并等待执行。

Device queues

每个设备都有设备队列，设备队列是等待该i/o资源的一组进程

调度程序

进程的一生：Job pool → memory pool ←→ CPU pool

多道程序的度：内存中的进程数量。

Long-term scheduler

从缓冲池中选择进程加载到内存中。

- 长期调度控制进程的数量

长期调度器应选择 I/O 绑定和 CPU 绑定流程的良好流程组合。

I/O-bound process：

在执行I/O方面比执行计算要花费更多的时间。

CPU-bound process

很少产生I/O请求，将更多的时间用在执行计算上。

Medium-term scheduler

中期调度程序的核心思想是将进程从内存（或CPU竞争）中取出，从而降低多道程序程度。之后进程重新调入内存，并从中断处继续执行。

- 通过中期调度程序，进程可以换出，并在后来换入。
- 为了改善进程组合，或者因内存要求的改变引起了可用内存的过度使用而需要释放内存，就有必要使用中期调度。

Short-term scheduler

从准备执行的进程中选择进程，并为之分配CPU。

- 经常被调用，速度较快（ms级别）

Context Switch

将CPU切换到另一个进程需要保存当前进程的状态并加载新进程的状态

进程操作

- 进程创建（creation）
  - 父进程创建子进程，进而再创建其他进程，形成进程树。
  - 父子进程间的资源共享
    - 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
    - 父进程与子进程不共享任何资源
  - 父子进程间执行可能
    - 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
- 进程执行（execution）
  - 获取/设置进程属性
  - 等待时间/事件
  - 信号事件
  - 分配/释放内存

- 进程终止
  - 进程执行完成，并请求操作系统将其删除
    - 从子进程输出数据到父进程
    - 进程资源通过操作系统来取消分配
  - 父进程可以终止执行子进程
    - 子进程所用资源已超出为其分配的资源
    - 分配给子进程的任务已不再需要
    - 如果一个进程终止，那么它的所有子进程也被终止。（ 级联终止 ）

Independent process

如果一个进程不能影响其他进程或被其他进程所影响，那么该进程是独立的。显然，不与任何其他进程共享数据的进程是独立的。

Cooperating process

如果系统中一个进程能影响其他进程或被其他进程所影响，那么该进程是协作的。显然，与其他进程共享数据的进程为协作进程。

- 进程合作
  - 进程合作能够加快计算速度，使进程模块化（系统层面），为用户提供便利（用户层面）。
  - 合作机制：通信&同步
- 进程通信
  - 共享内存
  - 同一台计算机上IPC
  - 分布式计算机的C/S

Inter Process Communication

进程间的通信需要一种进程间通信机制(inter process communication, IPC)来允许进程相互交换数据信息以及同步他们的操作。IPC最好由消息传递系统提供，消息传递系统有许多定义方式。

进程间通信有两种基本模式:(1）共享内存，(2）消息传递。

- IPC与共享内存：效率不同。IPC是分布式的，共享内存是集中式的。
- IPC操作：发送和接收。
- IPC问题
  - Naming
    - 直接通信
      - 对称寻址：进程必须明确彼此的命名：send(P ,message)、receive(Q ,message)
      - 非对称寻址：只需要发送者命名接收者：send(P ,message)、receive(id ,message){id设置为与之通信的进程名}
      - 对称和非对称寻址方案的共同缺点是 限制了进程定义的模块化
    - 间接通信：消息通过邮箱（或端口）来发送和接收，每个邮箱都有唯一的ID。允许系统选择接收者，并且可以告诉发送者谁是接收者
  - 同步(Blocking&Non-blocking)
    - 阻塞send:发送进程阻塞，直到消息被接收进程或邮箱所接收。
    - 非阻塞send:发送进程发送消息并再继续操作。
    - 阻塞receive:接收者阻塞，直到有消息可用。
    - 非阻塞receive:接收者收到一个有效消息或空消息。
  - Buffering
    - 不管直接或间接通信，通信进程所交换的消息都驻留在临时队列中。
      - 0容量：发送者必须等待接收者
      - 有限容量：如果满了，发送方才需要等待
      - 无限容量：发送方永不等待

Socket

套接字被定义为通信的端口，实现客户端与服务器之间的通信。通信由一组套接字组成。e.g.:**146.86.5.2:1625**与**161.25.19.8/80**

Threads（线程）

引入线程目的：为了减少程序在并发执行时所付出的时空开销，使操作系统具有更好的并发性。

有时称线程为轻量级进程，是CPU使用的基本单元；它由线程ID、程序计数器、寄存器集合和堆栈组成。它与属于同一进程的其它线程共享其代码段、数据段和其他操作系统资源。

进程&线程

进程是一个可拥有资源的独立单元进程，同时又是一个可独立调度和分派的基本单元。

线程是程序执行流的最小单元，是CPU使用的基本单元。

多线程

在一个程序中可以定义多个线程并同时运行他们。每个线程可以执行不同的任务，彼此之间独立。

优点：

- 响应度高 ：多个线程分别处理相应事件，如多线程网页浏览器在用一个线程装入图像时，能通过另一个线程与用户交互。
- 资源共享 ：线程默认共享它们所属进程的内存和资源。
- 经济 ：进程创建所需的 内存和资源 的分配比较昂贵。由于线程能共享它们所属进程的资源，所以 线程创建和上下文切换 会更为经济。
- 多处理器体系结构的利用 ：能充分使用多处理器体系结构，以便每个线程能并行运行在不同的处理器上。

线程的实现

用户线程(User-Level Thread,ULT)or内核线程(Kernel-Level Thread,KLT)

- 用户线程在 内核之上 支持，并在用户层通过 线程库 来实现。线程库提供对线程创建、调度和管理的支持而无需内核支持。优点：能快速创建和管理。
- 内核线程由 操作系统 直接支持：内核在其空间内执行线程创建、调度和管理。内核线程的创建和管理通常要 慢于 用户线程的创建和管理。

- 用户线程：如果 内核是单线程 的，那么任何一个用户级线程若执行阻塞系统调用就会引起 整个进程阻塞 ，即使还有其他线程可以在 应用程序 内执行。
- 内核线程：由于内核管理线程，当一个线程执行阻塞系统调用时，内核能调度应用程序内的另一个线程以便执行。

多线程模型

多对一模型

多个用户级线程映射到一个内核线程。线程管理在用户空间进行，因而效率比较高，但如果一个线程执行了阻塞系统调用，那么整个进程就会阻塞。因为任一时刻只有一个线程能访问内核，多个线程不能并行运行在多处理器上。

一对一模型

每个用户线程映射到一个内核线程。该模型在一个线程执行阻塞系统调用时，能允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能；它也允许多个线程能并行地运行在多处理器系统上。

缺点：开销太大影响性能，限制了系统所支持的线程数量。

多对多模型

多路复用了许多用户级线程到同样数量或更小数量的内核线程上。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。

处理机调度

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。

调度是操作系统的基本功能。几乎所有计算机资源在使用前都要被调度。当然，CPU是最为重要的计算机资源之一。因此，CPU调度对操作系统设计来说非常重要。

CPU Scheduler

CPU调度器[本身是个程序]。每当CPU空闲时，操作系统就必须从就绪队列中选择一个进程来执行。进程选择由短期调度程序(short-term scheduler）或CPU调度程序执行。调度程序从就绪队列中选择一个能够执行的进程，并为之分配CPU。

抢占&非抢占使用时机

- nonpreemptive
  - 当一个进程从运行状态切换到等待状态（例如，I/O请求，或调用wait等待一个子）
  - ·当一个进程终止时。
- preemptive
  - ·当一个进程从运行状态切换到就绪状态（例如，当出现中断时)。
  - ·当一个进程从等待状态切换到就绪状态（例如，I/O完成)。

Dispatcher module

分派程序(dispatcher)是一个模块,用来将CPU的控制权交给由短期调度程序选择的进程。

Dispatch latency

分派程序停止一个进程而启动另一个所要花的时间称为分派延迟。

Scheduling Criteria

不同的CPU调度算法具有不同属性，且可能对某些进程更为有利。为了选择算法以适用于特定情况，必须分析各个算法的属性。为在不同算法之间进行比较，下面提出了一些评价许多准则：

CPU utilization

CPU利用率。CPU是计算机系统中最为昂贵和最重要的资源，应尽可能使CPU处于*busy*状态

CPU Throughput

一种测量工作量的方法称为吞吐量，它指一个时间单元内所完成进程的数量。

Turnaround time

从进程提交到进程完成的时间段称为周转时间。周转时间为所有时间段之和，包括等待进入内存、在就绪队列中等待、在CPU上执行和IO执行。

Waiting time

等待时间为在就绪队列中等待所花费时间之和。

Response time

从提交请求到产生第一响应的时间。这种时间称为响应时间，是开始响应所需要的时间，而不是输出响应所需要的时间。

人们需要使CPU使用率和吞吐量最大化，而使周转时间、等待时间和响应时间最小化。

调度算法

FCFS

First-Come, First-Served，先来先服务调度算法。采用这种方案，先请求CPU的进程先分配到CPU。FCFS策略可以用FIFO队列来容易地实现。当一个进程进入到就绪队列，其PCB链接到队列的尾部。当CPU空闲时，CPU分配给位于队列头的进程，接着该运行进程从队列中删除。

convoy effect

护航效果，小进程等待大进程[短作业等待长作业]。

在FCFS中，所有其他进程都等待一个大进程释放CPU，导致CPU和设备的使用率变得更低。

FCFS算法对于分时系统是尤为麻烦的。允许一个进程持有CPU的时间过长，将是个严重错误。

SJF

Shortest-Job-First，短作业优先调度算法，所用时间少的先调度。

当CPU为空闲时，它会赋给具有最短CPU区间的进程。如果两个进程具有同样长度，那么可以使用FCFS调度来处理。

SRTF

shortest-remaining-time-first scheduling，最短剩余时间优先调度算法，抢占式**SJF**调度算法。

当一个新进程到达就绪队列而以前进程正在执行时，就需要选择。与当前运行的进程相比，新进程可能有一个更短的CPU区间。最短剩余时间优先调度算法可抢占当前运行的进程。

短作业优先调度也存在着一定的缺陷：未考虑到作业的重要程度，某个较为重要的长作业可能一直不被执行。

Priority Scheduling

优先级调度算法。[在本课程中，小数字的优先级高]

每个进程都有一个优先级与其关联，具有最高优先级的进程会分配到CPU。具有相同优先级的进程按FCFS顺序调度。解决了SJF的重要程度问题，但并未解决饥饿的发生。通过**Aging（老化）**来解决饥饿问题

Aging

一个进程的优先级会随着时间的推移而增加。

RR

Round Robin，时间片轮转法。

将CPU 的处理时间划分成一个个时间片，就绪队列中的进程轮流运行一个时间片。当时间片结束时，就强迫运行进程让出CPU，该进程进入就绪队列，等待下一次调度。同时，进程调度又去选择就绪队列中的一个进程，分配给它一个时间片，以投入运行。当某一进程调度完成，时间片还有空闲时，会将时间片贡献出来。时间片的大小会影响周转时间。

Multilevel Queue

多级队列调度算法(multilevel queue scheduling algorithm)将就绪队列分成多个独立队列，根据进程的属性，如内存大小、进程优先级、进程类型，一个进程被永久地分配到一个队列，每个队列有自己的调度算法。进程并不在可队列之间移动。这种设置的优点是低调度开销，缺点是不够灵活。

Multilevel Feedback Queue

多级反馈队列调度算法(multilevel feedback queue scheduling algorithm)允许进程在队列之间移动，主要思想是根据不同CPU区间的特点以区分进程。如果进程使用过多CPU时间，那么它会被转移到更低优先级队列，此外在较低优先级队列中等待时间过长的进程会被转移到更高优先级队列，这种形式的老化阻止饥饿的发生。

Multiple-Processor Scheduling

如果有多个CPU，则负载分配（load sharing）成为可能。

同构&异构

同构（Symmetric）：处理器功能相同的系统，任何可用处理器都可用于运行队列内的任何进程。

异构（Asymmetric）：只有给定处理器指令集编译的程序才能运行在该处理器上。

real-time systems

硬实时系统需要在保证的时间内完成关键性的任务。

- 由运行在专用于关键进程的硬件上的特殊目的软件组成，因而缺乏现代计算机和操作系统的全部功能。

实现软实时系统要求仔细设计调度程序和操作系统有关方面。

1. 系统必须有优先权调度，且实时进程必须有最高的优先权。实时进程的优先权不能随时间而下降，尽管非实时进程的优先权可以。
2. 分派延迟必须小。延迟越小，实时进程在能运行时就可越快开始运行。

负载平衡(load balancing)

在SMP系统中，保持所有处理器的工作负载平衡，以完全利用多处理器的优点。否则，将会产生一个或多个处理器空闲，而其他处理器处于高工作负载状态，并有一系列进程在等待CPU。负载平衡(load balancing)设法将工作负载平均地分配到SMP系统中的所有处理器上。

进程同步

共享数据的并发访问可能导致数据的不一致，维护数据的一致性需要能够保证协作进程顺序执行的机制

Race condition

多个进程并发访问和操作同一数据且执行结果与访问发生的特定顺序有关，称为竞争条件(race condition)。

为了避免竞争条件，并发的进程必须要同步（Synchronized）

critical section

假设某个系统有n个进程，每个进程有一个代码段称为临界区(critical section)，在该区中进程可能改变共同变量、更新一个表、写一个文件等。

Solution to Critical-Section Problem

- Mutual Exclusion ： 互斥。如果进程P_i在其临界区内执行，那么其他进程都不能在其临界区内执行。
- Progress ： 有空让进。如果没有进程在其临界区内执行且有进程需进入临界区，那么只有那些不在剩余区内执行的进程可参加选择，以确定谁能下一个进入临界区，且这种选择不能无限推迟。
- Bounded Waiting： 有限等待。从一个进程做出进入临界区的请求，直到该请求允许为止，其他进程允许进入其临界区的次数有上限。

信号量的主要缺点

信号量的主要缺点是都要求忙等待(busy waiting)。当一个进程位于其临界区内时，任何其他试图进入其临界区的进程都必须在其进入代码中连续地循环。这种连续循环在实际多道程序系统中显然是个问题，因为这里只有一个处理器为多个进程所共享。忙等待浪费了CPU时钟，这本来可有效地为其他进程所使用。这种类型的信号量也称为自旋锁（spinlock)，这是因为进程在其等待锁时还在运行

deadlocked

两个或多个进程无限地等待一个事件，而该事件只能由这些等待进程之一来产生。当出现这样的状态时，这些进程就称为死锁(deadlocked)。

饥饿

无限期阻塞(indefinite blocking)或饥饿(**starvation**)，

进程在信号量内无限期等待，即一个进程永远不能从它被挂起的信号量队列中移除

死锁

一组阻塞的进程分别占有一定的资源，并等待获取一些已被其他进程占有的资源。（循环等待）这种情况称为死锁(deadlock)。

正常情况下，进程按如下顺序使用资源：

1. 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。
2. 使用：进程对资源进行操作。
3. 释放：进程释放资源。

多个线程因为竞争共享的资源很容易导致死锁的发生。

Necessary Conditions for Deadlock

以下四个条件同时满足时，会引起死锁。

Mutual exclusion

互斥。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。

Hold and wait

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，而该资源为其他进程所占有。

No preemption

非抢占。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。

Circular wait:

循环等待:有一组等待进程$\{P_0,P_1,\ldots,P_n\}$，P_1等待的资源为P_2所占有，P_2等待的资源为P_3所占有，……，P_{n-1}等待的资源为P_n所占有，P_n等待的资源为P_0所占有。

死锁问题的描述

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。这种图由一个节点的集合V和一个边的集合E组成。节点集合V分成两种类型的节点$P=\{P_1,P_2,\ldots,P_n\}$(系统活动进程的集合)和$R=\{R_1,R_2,\ldots,R_m\}$(系统所有资源类型的集合)。

请求边：有向边 $Pi \rightarrow Rj$　　分配边：有向边 $Rj \rightarrow Pi$

图中出现环，即表示可能存在死锁。

死锁的解决方案

- 可使用 协议以预防或避免死锁 ，确保系统永远不会进入死锁状态
- 可允许系统进入死锁状态，然后 检测并加以恢复
- 可 忽略 这个问题，认为死锁不可能在系统内发生。这种方法为绝大多数操作系统（如UNIX）所使用。

Deadlock Prevention

死锁预防

1. 破坏互斥：所有资源均共享。
    - 不现实
2. 破坏非抢占：当一个已保持某些不可剥夺资源的进程请求新的资源而得不到满足时，必须释放所有已保持资源。
    - 增加开销 、 降低吞吐量
3. 破坏持有并等待：静态分配，每次为进程分配所有需要的资源。
    - 资源浪费 且会导致 饥饿 现象
4. 破坏循环等待：当一个进程申请一个可用资源的时候，系统必须决定这次分配是否会使系统处在一种安全状态。存在安全序列才分配

Deadlock Avoidance

死锁避免。安全性算法和银行家算法

Deadlock Detection

死锁检测。资源分配图、死锁定理

死锁解除：

资源掠夺

挂起某些死锁进程，并抢占他们的资源。将这些资源分配给其他死锁进程。

撤销进程

强制撤销部分死锁进程并剥夺资源。

Rollback

回滚，将被抢占进程的状态恢复到某个安全状态

Memory Management

操作系统对内存的划分和动态分配，就是内存管理。

Address

Logical address

逻辑地址又叫做虚拟地址，CPU所生成的地址通常称为逻辑地址(logical address)

- 用户进程所见到的是逻辑地址

Physical address

内存单元所看到的地址(即加载到内存地址寄存器(memory-address register）中的地址）通常称为物理地址

联系

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。

- 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。

Memory-Management Unit (MMU)

运行时从虚拟地址到物理地址的映射的硬件设备被称为内存管理单元(memory-management unit，MMU）

基地址寄存器在这里称为重定位寄存器(relocation register)。

- 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。

输入队列(input queue)

根据所使用的内存管理方案，进程在执行时可以在磁盘和内存之间移动。在磁盘上等待调入内存以便执行的进程形成输入队列(input queue)。

Binding of Instructions and Data to Memory

将指令与数据绑定到内存地址通常可以发生在以下步骤中：

Compile time:

编译时。如果在编译时就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(absolute code)。如果将来开始地址发生了变化，那么就必须重新编译代码。

Load time

加载时。如果在编译时并不知道进程将驻留在内存的什么地方，那么编译器就必须生成可重定位代码(relocatable code)。对于这种情况，最后绑定会延迟到加载时才进行。如果开始地址发生变化，只需重新加载用户代码以引入改变值。

Execution time:

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，那么绑定必须延迟到执行时才进行。

Dynamic Loading

采用动态加载时，一个子程序只有在调用时才被加载。

- 更好的内存空间利用率{不用的子程序不会被装入}
- 如果大多数代码需要用来处理异常情况，如错误处理，那么这种方法特别有用。
- 动态加载不需要操作系统提供特别的支持

Dynamic Linking

链接过程推迟到执行时来进行。

Stub

如果有动态链接，二进制镜像中对每个库程序的引用都有一个存根(stub)。

存根是一小段代码，用来指出如何定位适当的内存驻留库程序，或如果该程序不在内存时应如何装入库。

存根会用子程序地址来替换自己，并开始执行子程序。

Overlay

覆盖。为了能让进程比它所分配到的内存空间大，可以使用覆盖。

只在内存中保留所需的指令和数据，当需要其他指令时，它们会装入到刚刚不再需要的指令所占用的内存空间内。

Swap

进程可以暂时从内存中交换(swap）到备份存储(backing store)[通常是快速硬盘]上，当需要再次执行时再调回到内存中。

Roll out, Roll in

滚入滚出。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。

swapping 性能

交换时间的主要部分是转移时间。总的转移时间与所交换的内存空间直接成正比。

Contiguous Memory Allocation

连续内存分配。为用户分配一段连续的内存空间。

Memory Protection

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（**relocation-register**）和界限寄存器（**limit register**），可以实现对内存的保护。可以保证操作系统和其他用户程序及数据不受该进程的运行所影响。

重定位寄存器含有最小的物理地址值;界限地址寄存器含有逻辑地址的范围值

分配方式

Multiple-partition allocation

多分区分配。将内存分为多个固定大小的分区，每个分区只能容纳一个进程。

操作系统有一个表用于记录哪些内存可用和哪些内存已用。

当有新进程需要内存时，为该进程查找足够大的Hole。如果找到，则可以为该进程分配所需的内存，未分配的可以下次再用。

Hole

可用内存块，各种不同大小的hole分布在整个内存中。

- 当进程终止时会将Hole归还， 如果 新 Hole和其他Hole相邻 则会 合并成更大的 Hole 。

Dynamic Storage-Allocation

动态分区分配。不预先分内存，当进程装入内存时，根据进程大小动态分配内存。

从一组可用Hole中选择Hole的常用方法：

First-fit

首次适应:分配第一个足够大的Hole。

查找可以从头开始，也可以从上次首次适应结束时开始。按地址递增查找，找到第一个满足要求的空闲分区，就可以停止。

Best-fit

最佳适应。 分配最小的足够大的Hole。

按容量递增查找，找到第一个满足要求的空闲分区。这种方法可以产生最小剩余孔

Worst-fit

最差适应。分配最大的Hole。

按容量递减查找，找到第一个满足要求的空闲分区。这种方法可以产生最大剩余孔，该孔可能比最佳适应方法产生的较小剩余孔更为有用。

对比

模拟结果显示首次适应和最好适应在执行时间和利用空间方面都好于最差适应。

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。

Fragmentation

Internal Fragmentation:

内存以固定大小为单元来分配。采用这种方案，进程所分配的内存可能比其所需要的内存要大，分区内部存在着空间浪费，这种现象称为内部碎片。

External Fragmentation

随着进程的装入和移出，空闲内存空间被划分为多个小片段。当所有总的可用内存之和可以满足请求，但并不连续时，这就出现了外部碎片问题。

- 可以通过 紧凑(Compaction) 技术来解决外部碎片。操作系统 不时地对进程移动和调整 ， 以便空闲空间合并。
- 另一种可能解决外部碎片问题的方法是 允许物理地址空间非连续 ， 有两种互补的机制可以实现这种算法：分页机制和分段机制。
- 分页：将主存空间 划分为大小相等且固定的块 （Block），进程也以块为单位进行划分（比固定分区的分区要 小得多 ）[物理划分]
- 分段：按照用户进程来划分。[逻辑划分]

Paging

分页(paging）内存管理方案允许进程的物理地址空间可以是非连续的。

- Frames： 帧 ， 将 物理内存 分为固定大小的块
- Pages： 页 ， 将 逻辑内存 分为固定大小的块
- Clusters：备份存储也可分为固定大小的 块 ， 其大小与内存的帧一样。
- $Frames.size = Pages.size = Clusters.size$ 、[帧、页、块的 大小相同]

Page Table

页表用于记录页面在内存中对应的物理块号，一般存放在内存中。

页表的硬件实现方法有很多，最简单的是将页表作为专用寄存器来实现。

Page-table base register (PTBR) 页表基寄存器，指向页表

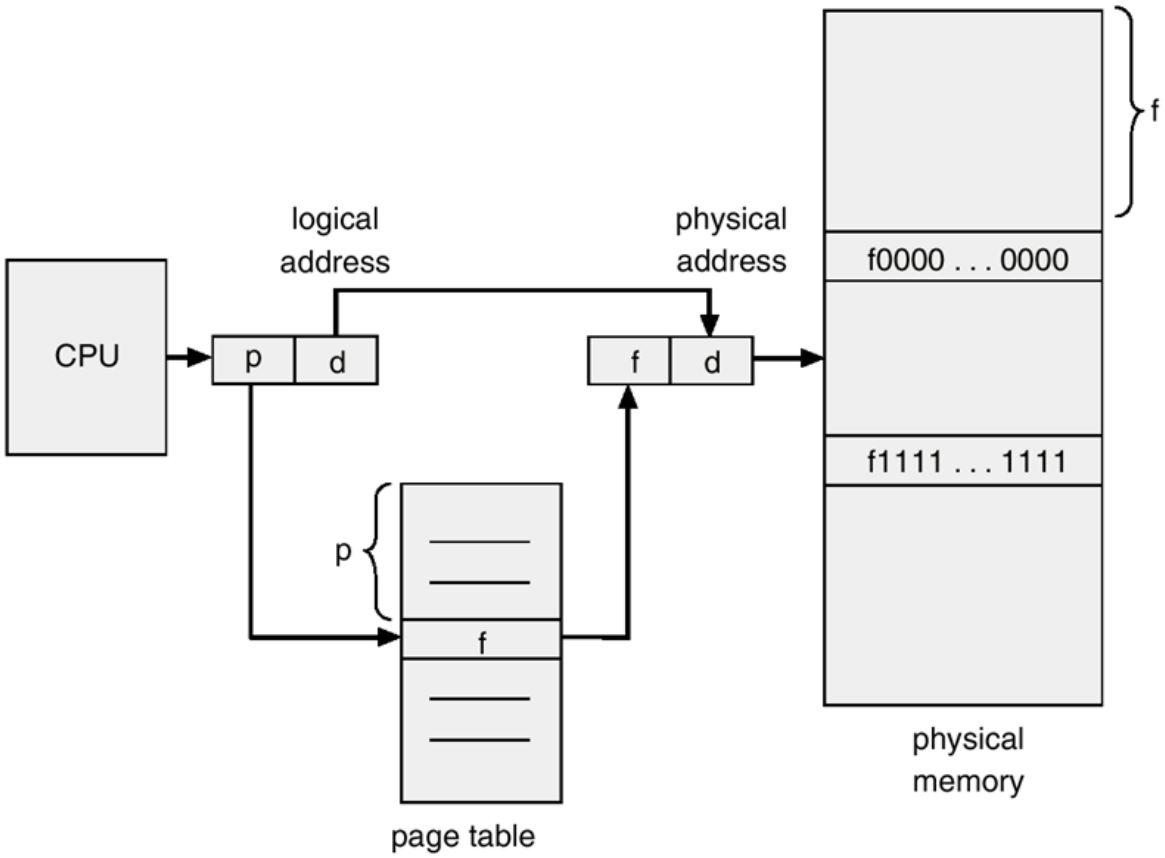
Page-table length register (PTLR) 表明页表的大小

逻辑地址结构

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。

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。

地址转换：



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。但，分页有内部碎片[页内碎片]。

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。

存取一个指令或数据至少要两次访问内存：

1. 访问页表 ， 确定 所要存取的指令或数据的 物理地址
2. 访问所确定的地址，存取指令或数据

两次内存访问问题可以用特别的快速查找硬件缓冲来解决：

TLB

关联内存、后备缓冲器、相联存储器。

用来存放当前访问的若干页表项，以加快地址变换过程。

当CPU产生逻辑地址后，其页号提交给**TLB**。如果找到页号，那么也就得到了帧号，并可用来访问内存。

如果不能在TLB 中找到页号，那就必须访问内存中的页表。当获得帧号时，我们可以用它来访问内存。另外，我们把这个页号和帧号添加到TLB 中，这样在下一次引用时可以快速的找到它们。如果TLB 已经满了，那么操作系统必须要选择一个表项置换。

hit ratio

页号在TLB中被查找到的百分比称为命中率。

protection bit

内存保护是通过与每个帧相关联的保护位来实现的，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的。

Valid-invalid bit

有效无效位与页表中的每一条目相关联

- 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。

segment

分段，分段是支持用户观点的内存管理方案。是通过逻辑单元的划分实现的。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。

逻辑地址结构

逻辑地址由两个元素组成<段号，偏移>。

Segment table

将二维的用户定义地址映射为一维物理地址。这个地址是通过段表(segment table)来实现的。

段表的每个条目都有段基地址（段的起始地址）和段界限（段的长度）。

- Segment-table base register (STBR)：段表基地址寄存器。指向内存中的段表的位置
- Segment-table length register (STLR)：段表长度寄存器。指示程序所用的段的个数

段基地址包含该段在内存中的开始物理地址，而段界限指定该段的长度。

优点

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。

分段的另一个优点是关于代码或数据的共享。每个进程都有一个段表，当该进程被允许使用CPU时，分派程序会定义一个硬件段表。当两个进程的某些条目指向同一个物理位置时，就可以共享段。

Virtual Memory

virtual memory

虚拟内存。将用户逻辑内存与物理内存分开。使计算机拥有比实际拥有的内存要大。

- 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
- 实现方式：请求页式调度、请求段式调度

Demand Paging

请求分页管理。进程驻留在次级存储器上。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。对于按需调页虚拟内存,只有程序执行需要时才载入页，那些从未访问的页不会调入到物理内存。

- 优点：需要更少的输入输出
- 更小的内存
- 更快的响应
- 更多的用户

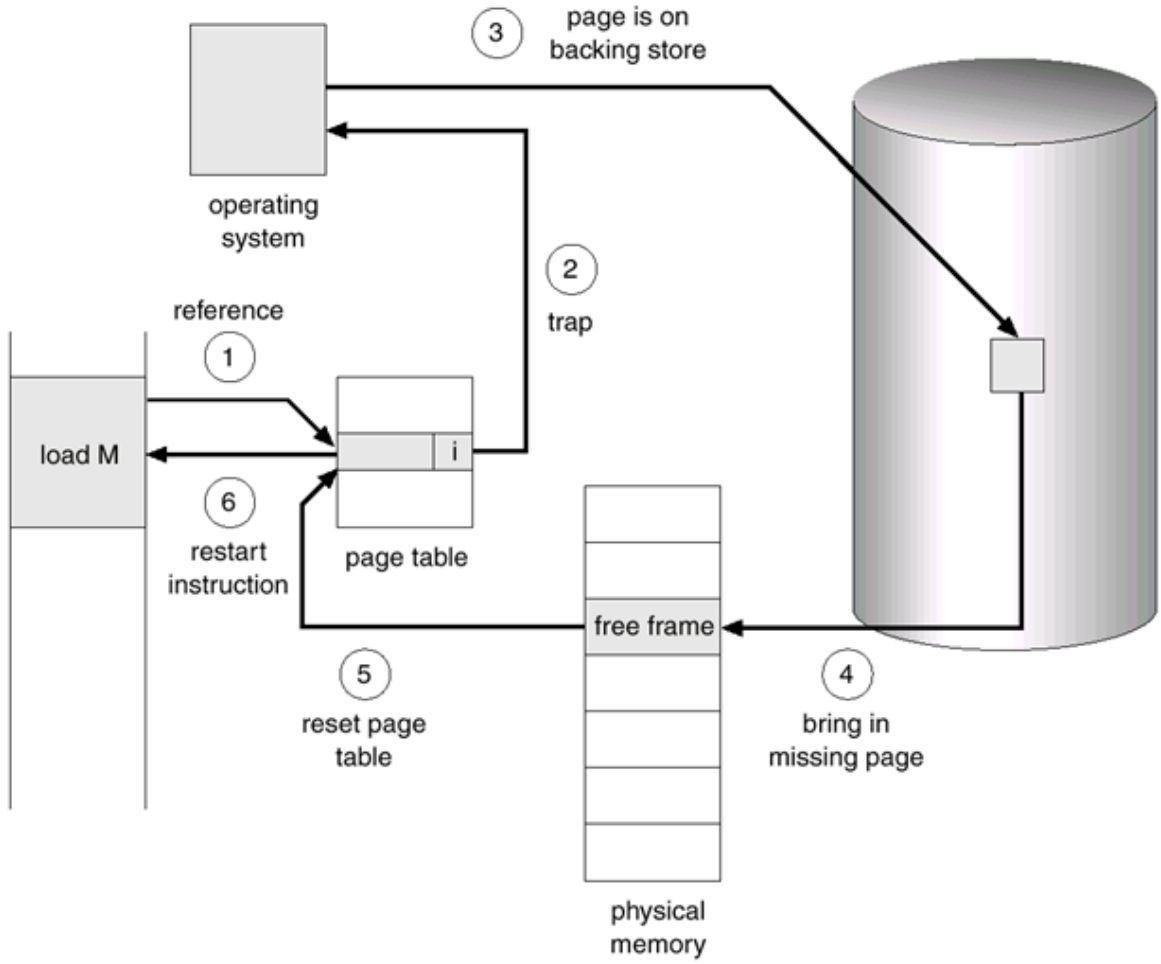
每个页表条目用一个有效-无效位来存储

- 1----有效且在内存中
- 0----无效或不在内存

在地址转换过程中，如果页表条目中的有效-无效位为0则表示页错误。

Page Faulting

页错误。当进程试图访问那些尚未调入到内存的页时，对标记为无效的访问会产生页错误，直到他所需要的所有页都在内存中。



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。

支持请求页面调度的硬件

- 页表：该表能够 通过有效-无效位 或 保护位 的特定值， 将条目设为无效 。
- 次级存储器：该次级存储器用来 保存不在内存中的页 。次级存储器通常是 快速磁盘 。

如果没有空闲帧，会发生页置换：在内存中找到一些页面，但没有真正使用，将其换出。

Page fault frequency

页错误率。理论上，每条指令可能会有页错误。但这种情况极为少见。

EAT

Effective Access Time，有效访问时间。页错误率为p，$EAT = (1-p)$(内存访问时间) $+p$（页错误时间）

- $p=0$，没有页错误
- $p=1$，每个应用都会出现页错误。
- 一般来说 $p \to 0$，EAT与 p 成正比。

虚拟内存也能在进程创建时，提供其他好处：

- 写时拷贝
- 内存映射文件

Copy-on-Write

写时拷贝。允许父进程与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副本。

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。

Memory-Mapped Files

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。

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。

PAGE REPLACEMENT

页置换。随着增加多道程序的级别，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。

reference string

内存的引用序列称为引用串

页置换算法

FIFO

先入先出。优先淘汰最早进入内存的页面。

Belady异常

对有的页置换算法[FIFO]，页错误率可能会随着所分配的帧数的增加而增加。

OPT

optimal，最优页置换算法。淘汰最长时间不被访问的页面。

- 最优页置换算法难于实现，因为 需要引用串的未来知识 。

LRU

最近最久未使用。选择过去一段时间内未访问过的页免来进行替换。

- 实现起来需要栈的支持：最近使用的放在栈顶，每次删除栈低。

基于计数器的页置换算法：

LFU：最不经常使用页置换算法，置换出计数最小的页。（使用的多表明以后使用概率也大）

MFU：最常使用页置换算法，置换出技术最大的页（技术较小的可能刚来还没来得及使用）

Frame Allocation

帧分配。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。
在这两者之间，关于帧分配还是有很多选择的。

Equal allocation

平均分配。在n个进程之间分配m个帧的最为容易的方法是给每个一个平均值，即m/n帧

Proportional allocation

比例分配。根据进程的大小按比例分配。

Global replacement

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，即一个进程可以从另一个进程中拿到帧。

Local replacement

局部置换要求每个进程仅从其自己的分配帧中进行选择。

Thrashing

频繁的页调度行为称为颠簸

解决方案：工作集合模型，页错误率。

Working-Set Model

前面提到，工作集合模型(working-set model）是基于局部性假设的。该模型使用参数△定义工作集合窗口(working-set window)。其思想是检查最近△个页的引用。这最近△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A时间单位后从工作集合中删除。因此，工作集合是程序局部的近似。

File System

file

文件是记录在外存上的相关信息的命名集合。

从用户的角度来看，文件是逻辑外存的最小分配单元。

文件包括数据和程序

File attributes

文件属性：

名称：文件符号名称是唯一的，按照人们容易读取的形式保存。有些OS区分大小写（如Linux，Unix），有些不区分（如DOS, Windows）。
标识符：标识文件系统内文件的唯一标签，通常为数字；这是文件的对人而言不可读的名称。
类型：由OS和程序定义。
位置：该信息指向设备和设备上文件位置的指针。
大小：文件当前大小，该属性也能包括可允许大小的最大值。
保护：决定谁能读、写、执行等的访问控制信息
时间、日期和用户标识：文件创建、上次修改和上次访问都可能有该信息。用于保护、安全和使用跟踪。
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

如果操作系统识别文件类型，那么它就能按合理方式对文件进行操作。

File structures

文件结构。每个应用程序必须有自己的代码对输入文件进行合适的解释。但所有的OS必须至少支持一种文件结构，即可执行文件结构，以便能装入和运行程序。

文件的结构通常由操作系统和程序来定义：

- 无：字或字节的序列
- 简单记录结构：行、固定长度、可变长度
- 复杂结构：格式化文档、可重定位装载文件

File operation

写文件、读文件、在文件内重定位

截短文件（**truncate**）：只删除文件内容而保留其属性，而不是强制用户删除文件再创建文件。

删除文件：在目录中搜索给定名称的文件，找到相关目录条目后，释放所有的文件空间以便其他文件使用，并删除相应目录条目。

Open(F_i)：在磁盘上的目录结构中查找$\mathbf{F_i}$，并将其内容复制到内存

Close(F_i)：将内存中的F_i的内容复制到位于磁盘上的目录结构中

File access

文件访问：

Sequential Access

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。顺序访问基于文件的磁带模型。

Direct Access

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。直接访问方式是基于文件的磁盘模型，这是因为磁盘允许对任意文件块进行随机读和写。对直接访问，文件可作为块或记录的编号序列。对于直接访问文件，读写顺序是没有限制的。直接访问文件可立即访问大量信息，数据库通常使用这种类型的文件。

Other Acces

其他访问方式可建立在直接访问方式之上。这些访问通常涉及创建文件索引。

Index

索引包括各块的指针。为了查找文件中的记录，首先搜索索引，再根据指针直接访问文件，以查找所需要的记录。

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。

DIRECTORY

目录结构。计算机的文件系统可以非常大，为了管理所有这些数据，需要组织它们。

1. 磁盘分为一个或多个分区 ， 或称为小型磁盘或卷 。通常，每个系统磁盘至少包括一个分区，这是用来保存文件和目录低层结构。
2. 每个分区都包括了存储在分区中的文件的信息 。这种信息保存在设备目录或卷内容表中。设备目录记录分区上所有文件的各种信息，如名称、位置、大小和类型等。

operations on directories

目录上的操作：搜索文件、创建文件、列出目录、重命名文件、遍历文件系统。

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

Directory structures

Single-Level

所有文件都包含在同一目录中，便于支持和理解。但存在命名问题与分组问题。

Two-Level

为不同的用户建立不同的目录（用户文件目录、主文件目录），不同用户的文件允许同名，但不支持分组。方便查找

Tree-Structured

最常见的目录结构。

- 具有根目录
- 每个文件都有唯一的路径名称
- 路径名称是从根到指定文件的路径，通过所有子方向。

Acyclic-graph

无环图目录。含有共享子目录和文件。同一文件或子目录可出现在两个不同目录中。无环图是树形结构目录方案的自然扩展。

- 一个文件可有多个绝对路径名。如果试图遍历整个文件系统，如查找文件，计算所有文件的统计数，复制所有文件到备份存储，可能会带来一定的问题。
- 删除文件：
  - 一种可能是每当用户删除文件时就 删除文件 ，但是这样 会留下悬挂指针指向不再存在的文件 。
  - 删除的另一种方法是保留文件直到删除其所有引用为止。

General graph

通用图目录。在无环图的基础上，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。

File System Mounting

文件系统安装。如同文件使用前必须要打开，文件系统在被系统上的进程使用之前必须安装

mount point

操作系统需要知道设备名称和文件系统的安装位置（称为安装点）

AFS semantics

AFS语义。

一个用户对打开文件的写不能被同时打开同一文件的其他用户所看见。

一旦文件关闭，对其修改只能为以后打开的会话所看见。已经打开文件的用户并不能看见这些修改。

Immutable-shared file semantics

永久共享文件语义

一旦一个文件被其创建者声明为共享，它就不能被修改。

永久共享文件有两个重要特性：文件名不能重用，且文件内容不可修改。

disk

磁盘。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）
- 为了改善I/O效率，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。

为了提供对磁盘的高效且便捷的访问，操作系统通过文件系统来轻松地存储、定位、提取数据。

文件系统

Basic file system

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。

The file-organization module

文件组成模块。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。

- 每个文件的逻辑块按从0或1到N来编码
- 每个文件的物理块地址是不同的，在分区内是唯一的。
- 空闲空间管理器
- 用来跟踪未分配的块并根据要求提供给文件组织模块。

logical file system

逻辑文件系统。逻辑文件系统管理元数据。

元数据

元数据包括文件系统的所有结构数据，而不包括实际数据（或文件内容）。

根据给定符号文件名来管理目录结构，并提供给文件组织模块所需要的信息。

逻辑文件系统通过文件控制块来维护文件结构。

在磁盘上，文件系统可能包括如下信息：如何启动所存储的操作系统、总的块数、空闲块的数目和位置、目录结构以及各个具体文件等。

磁盘结构

boot control block

引导控制块(boot control block)包括系统从该卷引导操作系统所需要的信息。如果磁盘没有操作系统，那么这块的内容为空。它通常为卷的第一块。Linux称之为引导块(boot block)，WindowsNT称之为分区引导扇区(partition boot sector)。

partition control block

(每个卷的)分区控制块（volume control block）包括卷（或分区）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。Linux称之为超级块(superblock)，而在WindowsNT中它存储在主控文件表(Master File Table）中。

directory structure

目录结构。用来组织文件

FCB

文件控制块。每个文件的FCB包括很多该文件的详细信息，如文件权限、拥有者、大小和数据块的位置。

文件系统也负责保护和安全。

内存信息（**In-memory**）用于文件系统管理和通过缓存来提高性能。

内存结构包括：

- 内存分区表：包含所有安装分区的信息
- 内存目录结构：保存近来访问过的目录信息（对安装分区的目录，可以包括一个指向分区表的指针）
- 系统范围的打开文件表：包括每个打开文件的FCB拷贝和其他信息
- 单个进程的打开文件表：包括一个指向系统范围内已打开文件表中合适条目的指针和其他信息：文件描述符（file descriptor, Linux/UNIX）、文件句柄（file handle, Windows）

采用数据结构和子程序，可以分开基本系统调用的功能和实现细节。因此，文件系统实现包括三个主要层次。

- 顶层：文件系统接口
  - 包括open、read、write和close调用及文件描述符。
- 中间层：虚拟文件系统（Virtual File Systems,VFS）

    - VFS层通过定义一个清晰的VFS接口，以将文件系统通用操作和具体实现分开
    - VFS是 基于称为vnode的文件表示结构 ， 该结构 包括一个数值指定者以表示位于整个网络范围内的唯一文件 。
    - VFS区分本地文件和远程文件，根据文件系统类型可以进一步区分不同本地文件。
- 底层：不同文件系统的实现

Directory Implementation

目录实现。

Linear list

线性表。线性表目录实现方法是使用存储文件名和数据块指针的线性表。

Hash Table

散列表。采用这种方法，除了使用线性表存储目录条目外，还使用了哈希数据结构。它可以大大地降低目录搜索时间。每个哈希条目可以是链表而不是单个值，可以采用向链表增加一项来解决冲突。

ALLOCATION METHODS

为多个文件分配磁盘空间，以便有效地使用磁盘空间和快速地访问文件。

Contiguous allocation

连续分配。方法要求每个文件在磁盘上占有一组连续的块。

- 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

Extent-Based Systems

许多新的文件系统采用一种修正的连续分配方法。

该方案开始分配一块连续空间，当空间不够时，另一块被称为扩展(extent）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。

Linked allocation：

采用链接分配，每个文件是磁盘块的链表;磁盘块分布在磁盘的任何地方。目录包括文件第一块的指针和最后一块

- 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

Indexed allocation

在链接分配的基础上，索引分配(indexed allocation)通过把所有指针放在一起，即通过索引块。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。

FREE-SPACE MANAGEMENT

为了记录空闲磁盘空间，系统需要维护一个空闲空间链表。空闲空间链表记录了所有空闲磁盘空间，即未分配给文件或目录的空间。当创建文件时，搜索空闲空间链表以得到所需要的空间，并分配给新文件。这些空间会从空闲空间链表中删除。

bit vector

比特向量。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。

Linked Lists

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

Consistency checking

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）。从备份上恢复数据以恢复丢失的文件或磁盘

I/O System

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

Polling

轮询。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)

interrupt

中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行

中断优先级

能够使CPU延迟处理低优先级中断而不屏蔽所有中断，这也可以让高优先级中断抢占低优先级中断处理。

DMA

对于需要大量传输的设备，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。

DMA控制器与设备控制器之间的握手通过一对称为DMA-request和DMA-acknowledge的线来进行。当有数据需要传输时，设备控制器就通过DMA-request线发送信号。该信号会导致DMA控制器抓住内存总线，并在内存总线上放上所需地址，并通过DMA-acknowledge线发送信号。当设备控制器收到DMA-acknowledge信号时，就可以向内存传输数据，并清除DMA-request请求信号。

内存映射文件访问是建立在块设备驱动程序之上的。

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

输出：其输出由底层的、硬件特定的命令组成，这些命令用于硬件控制器，通过硬件控制器可以使I/O设备与系统其他设备相连。

设备驱动程序通常在I/O控制器的特定位置写入特定位格式来通知控制器在什么位置采取什么动作。

I/O API

Blocking and non-blocking I/O

阻塞

- 进程悬挂直到I/O完成为止
- 容易使用与理解
- 对某些需求难以满足

非阻塞

- I/O调用立刻返回
- 用户接口，数据复制（缓冲I/O)
- 通过多线程实现
- 立刻返回读或写的字节数

同步I/O&异步I/O

同步（Synchronous）

I/O 启动后，控制仅在 I/O 完成后返回到用户程序。用户进程发出IO调用，去获取IO设备数据，双方的数据要经过内核缓冲区同步，完全准备好后，再复制返回到用户进程。而复制返回到用户进程会导致请求进程阻塞，直到I/O操作完成。

这种方法排除了多个设备的并发I/O操作，也排除了将有用计算与I/O相重叠的可能性。

异步（Asynchronous）

I/O 启动后，控制返回到用户程序，而无需等待 I/O 完成。

Scheduling

调度一组I/O请求就是确定一个好的顺序来执行这些请求。调度能改善系统整体性能，能在进程之间公平地共享设备访问，能减少I/O完成所需要的平均等待时间。

buffer

缓冲。用来保存在两设备之间或在设备和应用程序之间所传输数据的内存区域。

- 处理设备速度的差异。如调制解调器的速度与硬盘的速度的差异。
- 处理设备传输大小的差异。如计算机网络中，缓冲常常用来处理消息的分段和重组。
- 维护应用程序的“拷贝语义”

Cache

高速缓存(cache）是可以保留数据副本的高速存储器。高速缓冲区副本的访问要比原始数据访问要更为高效。

Spooling

假脱机。是用来保存设备输出的缓冲区，这些设备（如打印机）不能接收交叉的数据流。

操作系统通过截取对打印机的输出来解决这一问题。应用程序的输出先是假脱到一个独立的磁盘文件上。当应用程序完成打印时，假脱机系统将相应的待送打印机的假脱机文件进行排队。

假脱机系统一次拷贝一个已排队的假脱机文件到打印机上。

Device Driver

设备驱动程序为I/O子系统提供了统一设备访问接口，就像系统调用为应用程序与操作系统之间提供了统一的标准接口一样。

Device reservation:

设备预留。提供对设备的独占访问、谨防死锁

提高性能

- 减少上下文切换（context switch）的次数
- 减少设备和应用程序之间传递数据时在内存中的数据拷贝次数
- 使用大传输、智能控制器及轮流检测来减少中断频率。
- 通过采用DMA智能控制器和通道来为主CPU承担简单数据拷贝，以增加并发。
- 平衡CPU，内存子系统，总线和I/O的性能，因为任何一处的过载都会引起其他部分空闲。

Protection

保护是指一种控制程序、进程或用户对计算机系统资源的访问的机制。

- 操作系统中的进程必须保护加以保护，使其免受其他进程活动的干扰。
- 确保只有从操作系统中获得了恰当授权的进程才可以操作相应的文件、内存段、处理器和其他资源。

goal

目的：

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

need to know principle.

在任何时候，进程只能访问完成现阶段的任务所需要的资源

域（Domain）

访问矩阵定义了域和对象之间的权限关系。

进程必须能够在域之间切换。当需要将一个进程从一个域切换到另一个域时，其实是在一个对象上执行一个操作（切换）。

要想访问矩阵的条目内容提供受控更改需要三个额外的操作：拷贝、所有者和控制。

从访问矩阵的一个域中拷贝一个访问权限到另一个域，这种权限用附加在访问权限后面的“*”标记。

Security

安全需要考虑系统运行的外部环境，防止它们：被未经授权者访问、被恶意地更改或破坏、被意外地引入不一致问题

防止意外误用比防止恶意破坏要容易得多。

要保护系统，必须在4个层次上采取安全机制：

- 物理：必须采取物理措施保护计算机系统的站点，防止入侵者强行地或秘密地侵入。
- 人：必须谨慎筛选用户，减少授权用户授予入侵者访问权限的机会。
- 网络：现代系统中的许多计算机数据都在私人租用的线、共享的线上传播。中途截取这些数据的危害和入侵计算机的危害是等同的。
- 操作系统：操作系统必须防止自身遭受意外的或者有意的安全破坏。

Authentication

操作系统的一个主要安全问题就是验证。不同的密码可能会关联到不同的访问权限。

为了避免密嗅探或偷窥，系统可以使用配对密码集合。可以将这种算法扩展为使用一个算法作为密码。

算法是保密的。

Program Threats

Trojan Horse（木马）、Trap Door（后门）、Stack and Buffer Overflow（缓冲区溢出）

System Threats

Worms（蠕虫）、Viruses（病毒）、Denial of Service（拒绝服务攻击）

Threat Monitoring

FireWall（防火墙）、Intrusion Detection（入侵检测）