概述

1s = 1000ms 1ms= 1000 μs 1 μs = 1000ns

操作系统定义：一组有效控制和管理计算机各种硬件和软件资源，合理的组织计算机的工作流程，以及方便用户的程序的集合。

计算机系统分为四大部分：硬件、OS、应用程序和用户。

操作系统特征：并发（并发与并行有区别）、共享（互斥共享/同时访问）、虚拟、异步（前两个最基本）

操作系统是用户与计算机硬件之间的接口。

命令级接口：联机命令（交互式命令）/脚机命令（批处理命令）。

程序级接口：系统调用，又称广义指令。

操作系统是计算机系统资源的管理者。

大型机系统 Mainframe System

OS 发展：批处理系统 batch system（单道（每次一个作业）/多道（阻塞才切换，交互性弱，交互性弱是主要缺点））、分时系统。

Time Sharing system 分时系统

分时系统是多个用户分时共享，把系统资源分割，每个用户轮流使用一个时间片。

集群系统 Clustered：一组互联主机构成的统一的计算机资源。有对称集群，多个节点跑程序，互相监视。非对称：一台机器处于热备用模式。集群用于高性能计算。

实时系统 Real Time 软实时/硬实时（顺序严格）。

网络操作系统：各种资源共享/通信

分布式计算机系统：同一任务的分布与处理

操作系统市场格局，

三大体系：Unix 服务器/Win 桌面/Android 手机

特权指令：用户程序不能直接使用，如 IO/时钟/寄存器/中断，**系统调用不是特权指令**

用户态（目态）：执行应用程序时。

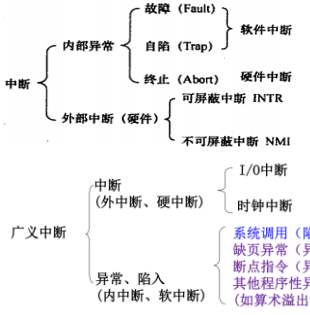
内核态（管态/内核态）：执行操作系统程序时。

内核态和用户态最本质的区别是能不能直接 access 到 hardware 和 memory。

仅内核态可做：时钟管理, Clear memory, 中断机制, Modify entries in device-status table, Access I/O device，原语

Interrupt(外中断)：Caused by external events. 可以在指令执行的任意时刻产生。

Exceptions(内中断): Caused by executing instructions. 一条指令终止执行后 CPU 才会发出中断。



INTR:用于实现多重中断

NMI:紧急硬件故障

故障：执行指令异常：除零、溢出、缺页

自陷：主动调用

终止：导致 CPU 无法继续运行的硬件故障

储存结构：

寄存器>高速缓存>主存>Secondary>tertiary

一级存储：register，cache，main memory

二级存储：nonvolatile memory，hard-disk

三级存储：optical disk（光盘），magnetic tape

分时系统响应时间：根据用户所能接受的等待时间确定。分时：工作时间分成时间片。

响应时间 S=进程数 n×每个进程运行时间片 q

操作系统结构

操作系统服务：

系统调用：User Interface/程序执行/IO 操作/错误检测/资源分配/账户管理/设备管理/文件管理/进程控制/进程通信和保护/内存管理/状态信息维护

程序运行从用户态切换到核心态时，需要使用访管指令

操作系统的用户界面

系统调用是进程和 OS 内核间的程序接口。

一般用 C/C++写，大多数由程序提供的叫做 API 应用程序接口，而不是直接的系统调用。

三个常见的 API 是 win32 API，POSIX API 和 JAVA API。

Time()是一个系统调用

系统调用有三种传参方式：寄存器传参/参数存在内存的块和表中，将块和表的地址通过寄存器传递，linux 和 solaris 用这种/参数通过堆栈传递。

操作系统结构：

MSDOS，小、简单功能有限的系统，没有划分为模块，接口和功能层次没有很好的划分

原始 UNIX：受到硬件功能限制，结构受限，分为两部分：系统程序和内核。

分层结构 layered kernel:最底层(层 0)为硬件，最高层为用户接口；效率较低，层际界限难定 与之相对的是 monolithic kernel 是一个整体

微内核结构 microkernel system

微内核+服务器，最基本的功能直接由微内核实现，其他功能都委托给独立进程。基本功能：进程（通信）/低级存储器/中断与陷入处理

好处：利于拓展、容易移植，更加可靠（内核态运行的代码更少了），更安全。

缺点：用户空间和内核空间的通信开销很大。Windows NT windows 8 10 mac OS L4 not efficient

单/宏内核 monolithic kernel

内核的全部代码，包括子系统都在一个文件中。更加简单更加普遍的 OS 体系。优点：组件之间直接通讯，开销小；缺点：很难避免源代码错误 很难修改和维持；内核越来越大。如 OS/360，VMS Linux

模块化结构 modules

大多数现代操作系统都实现了内核模块。面向对象，内核部件分离，通过已知接口进行沟通，都是可以载入到内核中的。如 Linux solaris。

混合系统 Hybrid 大多数非单一模型。

Linux 和 solaris 是 monolithic + module。

Windows 大部分是 monolithic 加上 microkernel。

Mac 是层次 Hybrid

操作系统的引导

Bootstrap program is loaded at power-up or reboot. 激活 CPU，在 ROM 读取 boot 程序，指令寄存器置为 BIOS；硬件自检，BIOS 程序在内存最开始构建中断向量表，通电自检；加载主引导记录 MBR，区分引导/非引导硬盘，主引导记录告诉我们去哪里找 OS；扫描磁盘分区表，MBR 包含硬件分区表；加载分区引导记录 PBR（活动分区第一扇区）；加载管理器与操作系统

计算机系统启动时，首先执行的是 BIOS 引导程序，完成自检，并加载主引导代码和分区表，然后执行主引导程序，由它加载激活分区（安装操作系统的分区）引导记录，再执行分区引导记录程序，加载操作系统，最后执行操作系统，配置系统。

虚拟机（可以用硬件或软件实现）

第一类虚拟机（裸金属架构）：裸机 + 虚拟机管理程序

第二类虚拟机（寄居架构）：有宿主操作系统（一般使用）

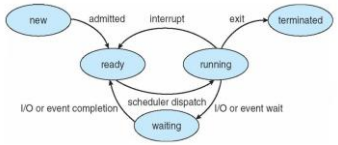
Process 进程

进程包括：PCB，程序段，相关数据段。

进程状态:

new, running, ready, waiting, terminated.

Wait->run 和 ready->wait 一般不可能发生。单处理器下，最多一个 run，ready 进程构成就绪队列，wait 进程构成多种等待队列。运行最多 1 最少 0，等待最多 n 最少 0（死锁），就绪最多 n-1 最少 0。



通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

a process is waken up --> 变为 ready 态而不是 execute 态

OS 用 **PCB** 来表示进程，PCB 包括

进程状态/优先级，进程/用户标识符，代码运行入口，程序外存地址，处理机占用时间，进入内存时间，信号量使用（但不包含信号量），代码段/数据段/堆栈段/指针，文件描述符，通用/地址/标志/控制寄存器值，状态字，PC，CPU 调度信息，内存管理/账户/文件/IO 状态信息

Linux 的 PCB 保存在 struct task_struct 里

Fork: fork() -> exec*() [replace memory space] -> parent calls wait() for children to terminate

wait(): block until any child terminates. waitpid(): block until a **specific** child completes

父进程 fork 之后的返回值是子进程的 pid，子进程自己的返回值是 0

Signal: signal() system call allows a process to specify what action to do on a signal

Zombies&Orphans: 僵尸进程：子进程运行终止，父进程尚未调用 wait()。孤立进程:父进程没有调用 wait()就终止了孤儿进程不会变为僵尸进程。Zombie die until its parent has called wait() or its parent dies. Orphan is “adopted” by the process with pid 1.

进程的创建：linux 中最初的进程为 systemd，pid 为 1，ppid 为 0.子进程的 fork 返回 0

进程的终止：有资源释放和返回状态值两个部分，其状态值被 parent 进程接受，若不接受其会一直有所占用，UNIX 的解决方法是，让 init/systemd 进程来 wait 它们。

进程调度

并发 concurrency 和并行 parallelism

长程/作业调度：选择应该被带入就绪队列的进程，调度频率低，控制多道程序设计的程度(内存中进程数)。

短程/CPU 调度：从接下来要执行的在就绪态的进程中选择并分配 CPU，频率很高。

中程调度：能将进程从内存或 CPU 竞争中溢出，降低多道程序设计的程度。进程可以被换出，之后再换入。

进程可以分为 IO 型和 CPU 型(CPU-bound)。

进程操作

父子进程资源共享模式：共享全部资源/部分/不共享；可并发执行/也可父进程等待子进程结束再执行；地址空间：子拷贝父/子进程装入另一个新程序。

进程终止：父进程终止时，不同 OS 对子进程不同：不允许子进程继续运行/级联终止/继承到其他父进程上。

合作进程，独立进程：运行期间不会受到其他进程影响。进程合作优点：信息共享,运算速度提高,模块化,方便。

进程通信 inter-process communication IPC:

类型有直接通信、间接通信

PV 是低级通信方式，较高效率的传输大量数据是高级传输方式

Shared memory: 需要使用同步互斥

低级的共享 -- 基于数据结构

高级的共享 -- 基于存储区

Message passing: 原语实现（send，receive）

直接通信，直接把消息发送给接受进程

间接通信，消息通过中间实体 mailbox 传递

Signals, Pipes: 允许多写多读，可介导父子进程通信，可能阻塞/半双工通信 生产者消费者固定大小缓冲区（大小受限），单向的

读进程可能快于写进程，需提供互斥同步和确定对方存在，管道是特殊文件 4KB

RPC remote procedure calls

semaphores

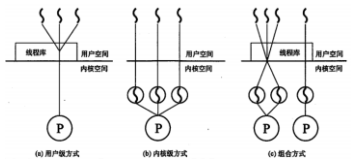
Thread 线程

调度单元被称为线程或轻量进程 LWP lightweight process.资源拥有单元被称为进程任务。**线程就是进程内一个执行单元或可调度的实体。**重量型/传统线程=单线程的进程

但是进程仍然是资源分配的基本单位

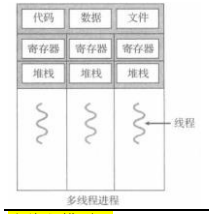
如果子进程 create 了一个线程，修改了进程的全局变量，那么这一修改对整个子进程可见（但是对父进程不可见）

线程能力：有状态转移（执行/阻塞/就绪三状态）,不运行时保存上下文,有**独立执行栈**,局部变量的静态存储,可以存取所在线程的资源,可以创建撤销其他线程



用户级线程（user-level thread）：不依赖于 OS 核心(内核不了解用户线程的存在)，应用进程利用线程库提供创建 一个线程发起系统调用而阻塞，则整个进程在等待（一对多模型中）。线程切换不需要内核，可以灵活使用不同的调度算法，没有多处理机优势

内核级线程（KLT）：依赖于 OS 核心，由内核的内部需求进行创建和撤销 ，用来执行一个指定的函数。内核维护进程和线程的上下文信息。线程切换由内核完成。一个线程发起系统调用而阻塞，不会影响其他线程。时间片分配给线程，所以多线程的进程获得更多 CPU 时间。



多线程模型：

多对一：将多个用户级线程映射到一个内核线程。由线程库在用户空间进行。优点：无需 OS 支持，可以调整调度策略满足应用需求，无系统调用线程操作开销很低。缺点：无法利用多处理器 不是真并行，一个线程阻塞时整个进程也阻塞

一对一：一个用户到一个内核。每个内核线程独立调度，线程操作由 OS 完成，win NT/XP/2000 linux Solaris 9 later。优点：每个内核线程可以并行跑在多处理器上，一个线程阻塞，进程的其他线程可以被调度。缺点：线程操作开销大,OS 对线程数的增多处理必须很好

多对多：允许 OS 创建足够多的内核线程，要求用户线程大于等于内核线程，Solaris prior v9 win NT/2000 with ThreadFiber。

两极模型：多对多的变种，一部分多对多，但是有一个线程是绑定到一个内核上。IRIX HP-UX Tru64 Solaris 8 earlier

隐式多线程 Implicit Threading：将线程的创建与管理交给编译器和运行时库来完成。如线程池、Fork Join、OpenMP

线程调用 fork：两种情况：仅复制线程、复制整个进程的所有线程（Linux 为第一种）

两种线程取消：

Asynchronous：立即终止目标线程;

Deferred：目标线程不断检查自己是否该终止。

信号处理：信号由特定事件产生，信号必须要发送给进程，信号被发送后需要被处理。

选择：发送信号到信号所应用的线程；到进程内的每个线程；进程内的某些线程；规定特定线程接收信号。

线程池：优点：用现有线程处理请求比等待创建新线程快；限制了可用线程的数量。

线程特有数据：允许线程自己保存数据拷贝，在无法控制创建线程时很有用。

CPU Scheduling

调度只能发生在时间片用尽时叫非抢占(non-preemptive)，否则叫做抢占(preemptive)。

分派程序(dispatcher)将 CPU 的控制交给由短程调度选择的进程。功能有：上下文切换 切换到用户模式/跳转到用户程序的合适位置来重启程序。分派程序停止一个进程而启动另一个所需要花费的时间叫分派延迟(dispatcher latency)。

调度算法的选择准则和评价：

turnaround time(周转时间):进程从 commit 到完成所用时间，包括就绪和阻塞中的等待时间。周转时间 T=完成时间-提交时间

不能只看甘特图，甘特图的开始并非最开始

带权周转时间 W=周转时间/CPU 执行时间

平均周转时间 = Σ周转时间/进程数

平均带权周转时间=ΣW/进程数

Response time 响应时间：从提交到首次被系统响应 **Waiting time 等待时间：**等待队列中等待时间=周转时间-运行时间

另外还有截止时间/公平性/优先级评价机制。

吞吐量 throughput：单位时间内 CPU 完成作业数量

CPU utilization：CPU 有效工作时间/（CPU 有效工作时间 + 空闲等待时间）

调度算法

用户级线程调度（对进程，实际线程透明）/内核级线程调度（对线程，直接调度）

First-Come, First-Served (FCFS) Scheduling:

利于长进程，不利于短进程，利于 CPU 型，不利于 IO 型。

Shortest-Job-First (SJF/SPF) Scheduling

抢占式：如果新来的进程时间比当前的剩余时间还短则抢占，Shortest-Remaining-Time-First (SRTF)。

它能给出最小平均等待/平均周转时间。而 **SJF 是无法实现的，因为不知道下一个 CPU 脉冲 burst 时长**。不利于长作业，存在饥饿。

比较的是当前运行剩余时间。预测下一次 CPU Burst 时长：$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$，其中$\tau$为预测时间，t 为实际时间。

优先级调度（会饥饿）静态优先级：创建时就确定好 动态优先权：适时进行优先级调整 Linux 中**一般数字越小优先级越高**。

优先级：系统进程 > 用户进程；交互性进程 > 非交互型进程；I/O 型进程 > CPU 型进程

优先级调度也可以是抢占（到则抢）/非抢占（等待当前时间片结束）。可用老化**(aging)**逐渐增加在系统中等待时间长的进程的优先级，来解决饥饿的问题。

高响应比优先调度算法（不会饥饿）

响应比 R =（等待时间 + 要求服务时间）/要求服务时间，取响应比最大的进行调度；利于短作业，利于等待时间较长的作业。

时间片轮转调度 Round Robin(RR)

一般来说 RR 比 SJF 有更长的平均周转时间，无饥饿，响应时间更段。时间片固定时，用户越多响应时间越长。

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units. 响应时间=n*q。q 不能比上下文切换时间短，越小切换开销占比越大。

多级队列调度

将就绪队列根据性质或类型的不同分为多个独立队列区别对待，综合调度。每个作业固定归入一个队列。不同队列有不同的调度算法。

一般，分成前台 foreground(交互式 interactive)和后台(批处理)，前台一般用 RR，后台则用 FCFS。多级队列在队列间的调度算法有：固定优先级，即先前台后后台，可能有饥饿；给定时间片，如 80%执行前台的 RR，20%执行后台的 FCFS。

Multilevel Feedback Queue Scheduling 多级反馈队列

是 RR、FCFS 和优先级算法的综合。与多级队列的区别:**允许进程在不同的就绪队列切换**，等待时间长的进程会进入到高优先级队列中。

若新进程进入任何优先级较高的队列时，立即把正在运行的进程放回当前队列的末尾，

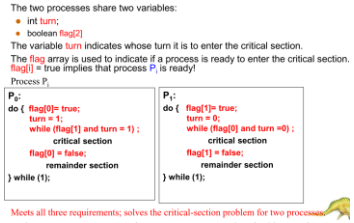
并把处理机分配给新高优先级进程。
短作业优先，周转时间较短，基本没有饥饿

Process synchronization

临界资源：一次仅允许一个进程使用资源
Entry section：检查是否可进入临界区；
Critical：访问临界资源的代码
Exit：访问临界区标志的清除；
Remainder：代码中剩余部分；
同步：直接制约关系
互斥：异步制约关系
临界区问题的解决必须满足三个要求：**忙则等待（mutual exclusion）；空闲让进（progress）；有限等待（bounded waiting）**。让权等待非必须（不要忙等）。

Peterson 算法（不饥饿）

只用于两个进程的情况，并且假设 load 和 store 是原子操作，是一种软件解决方法。



现代 OS 不适用(编译器优化可能交换指令)

Bakery Algorithm (面包房算法)

choosing[i]：进程 i 是否正在获取排队号
number[i]：进程 i 的排队号，0 表示不需要

```
do {
   choosing[i] = true;        // 保证计算number[i]值的原子操作
   number[i] = max(number[0], number[1], …, number [n – 1])+1;
   choosing[i] = false;
   for (j = 0; j < n; j++)
    { while (choosing[j]) ;
     while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;
    }
    critical section
   number[i] = 0;
    remainder section
} while (1);
```

硬件同步方法

1. 中断屏蔽：在临界区前后关闭开启中断。
多处理器：**Memory barriers**(an instruction forcing any change in memory to be propagated (made visible) to all other processors)

2. 硬件指令方法

Test_and_set 存在 busy waiting，lock 变量共享（swap），存在忙等问题

```
boolean TestAndSet(boolean &target) {
   boolean rv = target;
   target = true;
   return rv;
}
while(1) {
     key = TRUE;
     while (key == TRUE)
        Swap(&lock , &key);
     critical section
     lock = false;
     remainder section
}
```

```
while(1) {
     while (TestAndSet(lock)) ;
          critical section
     lock = false;
          remainder section
};
```

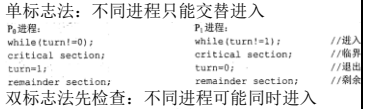
硬件方法优点：进程数任意，支持多个临界区，简单易验证正确性；缺点：无法让权等待，可能饥饿，可能死锁。
解决饥饿：

```
Boolean waiting[n], lock ;  // to be initialize to false, waiting[]：排队队列
while(1) {
     waiting[i]=true;                    解决“饥饿”问题
     key= true;
     while ( waiting[i] && key ) key=TestAndSet(lock);
     waiting[i]=false;
     critical section;
     j= (i+1) % n ;                      选择下一个进入临界区的进程，选择下一个waiting[j]=false
     while ( (j != i) && !waiting[j] ) j= (j+1) % n ;
     if (j == i)  lock = false ;
     else waiting[j] = false ;
      remainder section ;
}
```

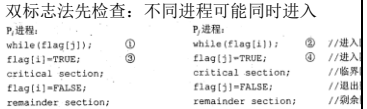
自旋锁 spinlock：适用于多处理器，当一个进程欲访问已被其他进程锁定的资源时，进程循环检测该锁是否被释放

软件（bad）

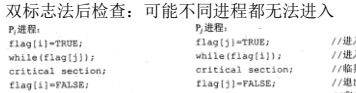
单标志法：不同进程只能交替进入



双标志法先检查：不同进程可能同时进入



双标志法后检查：可能不同进程都无法进入



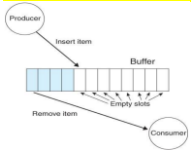
semaphores

两个操作 wait/P/down 申请和 signal/V/up 释放。信号量分为计数信号量，整型信号量(有忙等的问题)，二值信号量（互斥锁(mutex locks)），记录型信号量：

```
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);
        }
}
```

具有忙等的信号量值非负，**解决忙等如上**信号量可以为负，负数的绝对值代表等待该信号量的进程数，0 代表无资源可用。Block()用于自我阻塞，wake_up()用于进程唤醒，
Wait 和 signal 成对出现，互斥要求在同一进程出现，同步要求在不同进程。**连续的 wait 需要注意顺序**。同步 wait 和互斥 wait 相邻时，要先同步 wait，不然可能会死锁
前驱关系：利用同步的信号量思想，控制语句间的执行顺序
优先级倒置(priority inversion)：当优先级较低的进程持有较高优先级进程所需的锁时的调度问题。**优先级继承：**将等待进程中的最高优先级临时分配给持有锁的进程。

Bounded-Buffer Problem




```
do {
   ...
  produce an item in nextp
   ...
  wait(empty);
  wait(mutex);
   ...
  add nextp to buffer
   ...
  signal(mutex);
  signal(full);
} while (1);
```

```
do {
   wait(full);
   wait(mutex);
   ...
  remove an item from buffer to nextc
   ...
  signal(mutex);
  signal(empty);
   ...
  consume the item in nextc
   ...
} while (1);
```

设置 N 个缓冲项：信号量 full 初始化为 0，表示满缓冲项的个数；信号量 empty 初始化为 N 表示空缓冲项的个数。
信号量 mutex 初始化为 1，用来保证对缓冲池访问的互斥要求。

Readers-Writers Problem（读者写者问题）

允许多个读者同时读；只允许一个写者进行写，此时写者可能饿死（读者不断到达）。
共享数据有访问的数据、mutex 初始 1,保证更新 readcount（记录读者数量）时互斥；wrt 初始 1,为读写公用，供写者作为互斥信号量，同时供第一个进入临界区和最后一个离开临界区的读者使用。

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
     wait(wrt);
   signal(mutex);
     ...
    reading is performed
     ...
   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);
   signal(mutex);
} while (TRUE);
```

```
do {
   wait (wrt) ;
     //   writing is performed
   signal (wrt) ;
} while (TRUE);
```

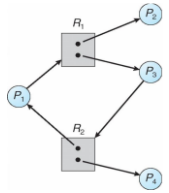
Dining-Philosophers Problem 哲学家进餐

典型的同步问题，问题描述：N 个哲学家坐在圆桌，每个哲学家和邻居共享一根筷子；哲学家吃饭要用身边的两只筷子一起吃；邻居不允许同时吃饭；哲学家只会思考或者吃饭。可能解决（死锁/饥饿）方案：最多只允许 4 个哲学家坐在桌上/临界区内必须同时拿起两根筷子/使用非对称的解决方法：奇数先拿左手，偶数先拿右手；加入吃饭时碗的限制。

Deadlock 死锁

四个必要条件：
Mutual exclusion（互斥条件）
hold and wait（请求并保持条件）
No preemption（不剥夺条件）
circular wait（循环等待）

资源分配图，由点 V 和边 E 组成，V 被分为两部分：系统活动进程的集合/系统所有资源类型的集合。进程 Pi 到资源 Rj 的有向边记为 Pi->Rj，表示进程 Pi 正在申请资源类型 Rj 的一个实例，表示请求边；资源 Rj 到进程 Pi 的有向边表示资源类型 Rj 的一个实例已经分配给进程 Pi，表示分配边。进程用圆表示，资源类型用方表示，资源实体是方内部的点。
如果分配图无环表示没有产生死锁，如果有环，那么**可能**死锁（注意只是可能）。
如果每个资源恰好只有一个实例，有环则必定死锁。
P4 可能释放 R2 的实例，这个资源分配给 P3，这样就打破了死锁。



死锁处理

保证系统不进入死锁：prevention/avoidance
允许进入死锁但可恢复：detection/recovery。
Unix/Linux/Windows 三个系统都不考虑死锁，产生死锁时再进行处理。

死锁预防 Prevention

通过破坏死锁产生的必要条件来预防死锁。
Mutual exclusion：非共享资源必须互斥，而共享资源不需要互斥，也不导致死锁。
Hold and wait：**采用预先静态分配**，进程在执行前就要申请并获得所有需要的资源。缺点：低资源利用率、可能饥饿。不破坏循环等待）
No preemption：如果一个进程占有资源并且申请了另一个不能立即分配的资源，那么它现已分配的资源都可以被抢占，即被隐式释放了。
Circular wait：**采用顺序资源分配法**，给资源设置显式序号，请求必须按照资源序号递增的方式进行。

死锁避免

要求每个进程声明它**可能需要的每种类型的资源的最大数量**。死锁避免算法动态检查资源分配状态，确保不会出现循环等待。资源分配状态由可用和已分配资源的数量以及进程的最大需求定义。
安全状态：对于所有进程，如果存在一个安全序列，那么系统就处于安全状态。对于进程序列 P1,P2,…,Pn，如果对于每一个 Pi,Pi 仍然可以申请的资源数小于当前可用的资源加上所有进程 Pj(i>j)所占有的资源，那么这一序列是安全序列。
安全状态->没有死锁；不安全状态->可能有死锁；避免->保证系统永远不进入非安全状态。
资源分配图，single instance 死锁避免算法：
Single instance：每种资源只有一个
引入一种新边 claim edge 需求边，Pi->Rj 表示进程 Pi 在未来可能请求资源 Rj，用虚线表示。当进程真正请求资源时，用请求边覆盖掉需求边。当资源被分配给进程后，用分配边来覆盖掉请求边，当资源被释放后，分配边恢复为需求边。
假设进程 Pi 申请资源 Rj。只有在需求边 Pi->Rj 变成分配边 Rj->Pi 而**不会导致资源分配图形成环**时，才允许申请。。
Banker，银行家算法：
每个进程实现说明最大需求；进程请求资源时可能会等待；进程拿到资源后必须在有限时间内释放它们。
N 进程数，m 资源类型的种类数；
Available 现有的 Max：最大需求 Allocation：当前分配 Need：当前需求
没有操作系统使用 banker 死锁避免。
安全状态检测算法：
1. 设 work 和 finish 分别是长度为 m 和 n 的向量，初始化：work=available，finish[i]=false；
算法需要 m*n*n 的操作数量级确定系统状态
资源请求算法：*
死锁检测
允许系统进入死锁状态的话，那么系统就需要提供检测算法和恢复算法。
死锁定理：S 为死锁状态的充分条件是，尚且仅当 S 状态的资源分配图是不可完全简化的
单实体资源检测算法还是使用**资源分配图**有无环来判断是否死锁
多实体资源类型检测算法（类似银行家算法）：
数据结构：Available，alloction 是一样的，request：n*m 的矩阵，表示当前各进程的资源请求状况，request[i][j]=k 表示 Pi 正在请求 k 个资源 Rj。
1.设 work 和 finish 分别是长度为 m 和 n 的向量，初始化：work=available，如果 allocation[i]非 0，finish[i]=false 否则初始化为 true；
2.寻找 i 满足 finish[i]=false 且 request[i]<=work，如果 i 不存在跳到第四步；
3.work=work+allocation[i],finish[i]=true，返回第二步；
4.如果某个 finish 是 false，那么系统处于死锁状态，且对应下标的进程 Pi 死锁。
算法需要 m*n*n 的操作数量级确定系统状态
死锁检测算法的应用
检测算法的调用时刻及频率取决于：死锁发生频率以及思索发生时受影响的进程数。如果经常发生死锁，那么就要经常调用检测。如果在不确定的时间调用检测算法，资源图可能有很多环，通常不能确定哪些造成了死锁

死锁恢复（解除）
进程终止（撤销进程）
两种方法来恢复死锁：终止所有死锁进程/一次终止一个进程直到不死锁。许多因素影响终止进程的选择：优先级/进程已经计算了多久/还要多久完成/进程使用了哪些类型的资源等等
抢占资源（资源剥夺法）
选一个进程挂起：代价尽量最小化；
回滚：回退到安全状态；
饥饿：确保被挂起的进程不会长时间得不到资源而产生饥饿
进程回退法
让一个或多个进程回退到足以回避死锁的状态，回退时进程自愿释放资源

Main Memory 主存

编译：将用户源代码编译成若干目标模块
链接：将编译后的一组目标模块以及需要的库函数链接在一起（**逻辑地址的产生**）
装入：程序装入内存运行
静态链接：编译后所有目标模块都是从 0 开始编址，需要修改相对地址使多个模块的地址具有一致把外部调用符号转变为相对地址
装入时动态链接：边装入边链接，便于实现目标模块的共享、修改与更新
运行时动态链接：未被用到的目标模块，都不会被调入内存和链接到模块上
绝对装入：只适合单道程序环境，编译程序将产生绝对地址的代码
可重定位装入（静态重定位）：地址信息可以产生一定的偏移；作业装入内存时，要求分配全部完整的内存空间，程序在内存中不能移动
动态运行装入（动态重定位）：支持程序在内存中移动，将地址转换推迟到程序真正需要运行时进行；需要借助重定位寄存器实现功能
层次存储中主存 cache 寄存器为 volatie 易失的。
逻辑地址/虚地址/相对地址：由 CPU 生成，首地址为 0，逻辑地址无法在内存中读取信息。
物理地址/实地址/绝对地址：内存中储存单元的地址，可以直接寻址。
物理地址中的逻辑地址空间是通过一对**基址寄存器（base register）**和**界限地址寄存器（limit register）**控制。
Memory-Management Unit (MMU)
就是将虚拟地址映射到物理地址的硬件设备。在 MMU 中，base 寄存器叫做重定位寄存器（relocation register），用户进程送到内存前，都要加上重定位寄存器的值。PA = relocation register + LA。用户程序只能处理 LA，永远看不到真的 PA。
Dynamic Loading（动态加载）
进程大小会收到物理内存大小的限制，为了有更好的空间使用率，采用动态加载。一个子程序只有在调用时才被加载，所有子程序都可以重定位的形式存在磁盘上，需要的时候装入内存中。
Swapping（交换技术）
进程可以暂时从内存中交换到备份存储 backing store 上，当需要再次执行时再调回。
需要动态重定位 dynamic relocate
备份存储：是快速硬盘，可以容纳所有用户的所有内存映像，并为这些内存映像提供直接访问，如 Linux 交换区 windows 的交换文件 pagefile.sys
Roll out roll in：如果有一个更高优先级的进程需要服务，内存交换出低优先级的进程以便装入和执行高优先级进程；介导程序在辅存与内存之间的交换。
交换时间的主要部分是转移时间 transfer time。总转移时间与所交换的内存大小成正比。系统维护一个就绪的可立即运行的进程队列，并在磁盘上有内存映像。
正常情况下，禁止交换；当空闲内存低于某一个阈值时，启用交换换出；当空闲内存增加至一定数量时，停止换出

Contiguous Allocation（连续分配）
单一连续分配
内存通常分为两个区域：一个驻留 resident 操作系统，一个用于用户进程（仅有一道用户程序），由于中断向量一般位于低内存，所以 **OS 也放在低内存**。
重定位寄存器用于保护各个用户进程以及 OS 的代码和数据不被修改。Base 是 PA 的最小值；limit 包含了 LA 的范围，每个 LA 不能超过 Limit。MMU 地址映射是**动态**的。
Multiple-partition allocation(可能外部碎片也可能内部)：分区式管理将内存划分为多个连续区域叫做分区，每个分区放一个进程。
固定分区：又分为分区大小相等、分区大小不等，产生内部碎片
动态分区（可变分区分配）：
动态划分内存，在程序装入内存时切出一个连续的区域 hole 分配给进程，分区大小恰好符合需要。往往产生外部碎片。
操作系统需要维护一个表，记录哪些内存可用哪些已用。从一组可用的 hole 选择一个空闲 hole 的常用算法 first/best/worst/next-fit 四种：First：分配到第一个足够大的；best（最容易产生碎片）：分配到最小的足够大的；worst：分配到最大的；next：从上次分配的地址开始查找的 first-fit。Best 和 worst 都要搜索整个 list，除非按 size 排序好了。First 和 best 在时间和空间利用率都比 worst 好。
紧凑技术：对系统中的磁盘碎片进行整理。
碎片 fragmentation
first 和 best 都存在外部碎片的问题。外碎片指所有的总可用内存可以满足请求，但是并不连续。内碎片是进程内部无法使用的内存，这是由于零头和块大小造成的。

分页存储管理
分页允许进程的 PA 空间非连续；将物理内存分为固定大小的块，叫做**帧 frame/物理块/页框**，将逻辑内存也分为同样大小的块叫做**页**

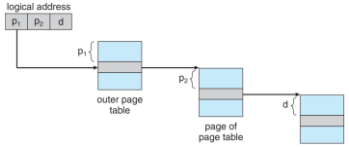
page，Linux Win(x86)是 4KB。
页表：记录页面在内存中对映的物理帧号。
逻辑地址和页表大小及虚存空间有关：虚存大小 2^m B 那么逻辑地址长度为 m，页表大小 2^n B，则页偏移位数 n，页号位数 m-n。

页表的实现
页表放在内存中。PTBR page-table base reg 指向页表，切换页表只需要改变这个寄存器就可以 PRLR page-table length register 说明页表长度，这样的模式下每次数据/指令访问都需要两次内存访问，一次查页表一次查数据/指令。
为了加速这个过程，引入了特殊的转换表缓冲区 TLB，是一种硬件 cache。部分 TLB 维护了 ASID addressspace identifier，用来唯一地标识进程，为进程提供空间保护。否则每次切换根页表需要 flush TLB。
*Associative memory:一种支持并行搜索的内存，如果虚页号与其中记录匹配上，则直接返回物理帧。

Effective Access Time 有效访问时间 EAT
Associative lookup = t1 查 TLB 表的时间
Memory access time = t2 内存访问时间
αTLB 命中率
那么 EAT = (t1+t2) *α+ (t1+t2+t2) *(1-α)
也就是查 TLBmiss 后，需要进内存查一次页表，再去取一次数据，命中就直接取数据。

保护 protection
内存保护通过与每个帧关联的保护位实现。
Valid bit 存在页表中的每一个条目上。
Shared code 共享代码：如果代码是可重入代码，即不可修改代码 reentrant code 或者是纯代码 pure code，可以共享，共享代码在各个进程中的逻辑地址空间相同。

分级页表 Hierarchical page table（最外面是第一级）



P1 是用来访问外页表的索引，p2 是内页表的页偏移，然后 d 是页内偏移。对于一个 32 位的 LA，一般 10 位外 10 位内 12 位偏移
三级索引：2nd outer -> outer -> inner -> offset

哈希页表
超过 32 位 LA 地址空间时，一般采用哈希页表，将虚页号的哈希值存到哈希表里，哈希表的每一项都是链表，链着哈希值相同的页号。然后在查表时用虚页号与链表中的每个元素进行比较从而查物理表号

反向页表 Inverted page table
对于每个 physical frame 有一个条目。每个条目包含映射到该 frame 的虚拟页的**虚地址及拥有该页的进程 PID**。因此整个系统只有一个页表，对每个物理内存的帧也只有一条相应的条目。拿时间换空间。

分段 Segmentation（二维）
分段管理支持用户视角的内存管理方案，LA 空间是由一组段组成的，每个段都有其名称和长度，地址指定段名称和段内偏移。
LA 通过有序对<segment-number,offset>构成。（段内要求连续，段间不必要）。段是例如函数、堆栈、main program 等的逻辑 unit。

段表将用户定义的二维地址映射成一维，每一个条目包含段号、base 起始和 limi 长度 t。STBR segment table base reg 指向内存中段表的位置，STLR 一个程序使用的段长度，用户使用的有序对中的 **segment-number 必须小于 STLR，offset 需要小于段长 C。**
同样有 valid 位，还有读写执行的权限设置，也可以进行 code share。内存分配是动态存储分配问题。

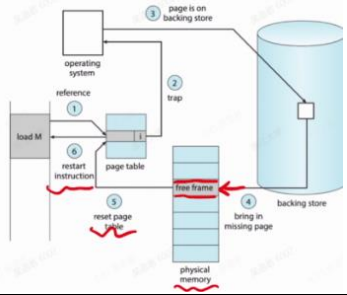
段页式管理
分页存储管理能有效地提高内存利用率，而分段存储管理能反映程序的逻辑结构并有利于段的共享和保护。
LA 由“段号 + 页号 + 页内偏移量”构成
系统建立一张段表，每一分段建立一张页表；先通过段表查到页表起始地址，再通过页表找到页帧号。

Virtual Memory 虚存

虚存将用户的路基存储和物理存储分开；LA 空间可以大于 PA 空间；允许 PA 空间被多个进程共享。
时间局部性原理+空间局部性原理
按需调页 Demand Paging
指在需要时才调入相应的页的技术。采用 lazy swapper 的方式，除非需要页面，否则不进行任何页面置换。
Page fault
非法地址访问和不在主存或无效的页都会 page fault。**Page fault rate 等于 1 不代表 every page is a page fault。**
更完整的页表项请求分页中
虚拟页号 物理帧号 状态位 P(存在位 页是否已调入内存) 访问字段 A(记录页面访问次数) 修改位 R/W(调入内存后是否被修改过) 外存地址(用来调页)
Effective memory-access time 有效访问时间 EAT = (1 - p) * memory access time + p * page fault time（注意 1 - p）
Page fault time 包括 page fault overhead, swap page out, swap page in, restart overhead 等 **page fault 会把页调入 TLB**
为了计算 EAT，必须知道需要花多少时间处理 page fault，page fault 会引起以下动作
6.CPU 调度
7.IO 中断
其中的三个主要 **page fault 时间是缺页中断服务时间/缺页读入时间和重启进程时间**

写时复制 copy-on-write
COW copy on write 允许父子进程开始时共享同一页面（只读），在父/子进程要修改共享页时，它才会拷贝一份该页面进行写（触发缺页异常，原来的页仍然只读）。
COW 加快了进程创建速度。当确定一个页采用 COW 时，那些空闲页在进程栈或堆必须拓展时可用于分配充当 COW 页。Linux 有 vfork，其不使用 COW，因为针对那种马上用 exec 的进程

页面置换
寻找一些内存中没有使用的页换出去。内存的过度分配 over-allocation 会导致 page fault 调页后发现所有页都在使用。
使用 dirty/modify 位来减少页传输的开销，只有脏页才需要写回硬盘。
基本页面置换过程：



页面置换算法
针对特定的内存引用序列，运行算法，计算出页错误数。引用序列叫做引用串 reference string。
注意两个事实：给定页大小，只需要关心页码，不用管完整地址；紧跟页 p 后面对页 p 的引用不会引起页错误。

First-In-First-Out Algorithm (FIFO，先进先出算法)
FIFO 会出现可用物理帧越多，page fault 数越大的问题，这种结果叫 **Belady’s Anomaly**
Optimal Page Replacement OPT 最佳页置换
Least Recently Used LRU 最近最久使用
LRU 算法如何获取多长时间没引用？
计数器 counter：每一个页表条目都有一个 counter，每次被引用，就把时钟信息复制到 counter。当置换时，置换时间最小的页，最近越使用，clock 越大。
栈：维护一个页码栈，栈由双向链表实现。引用页面时将该页面移动到顶部。替换时直接替换栈底部页就是 LRU 页。
位移寄存器：页使用则最高位置 1，定时右移，高位补 0，数值最小的也被替换
LRU Approximation (NRU) LRU 近似
很少有计算机有足够的硬件支持真正的 LRU，因此许多系统为页表中的每项关联一个引用位 reference bit，初始化为 0，当引用一个页时，读写都可以，对应页面的引用位设为 1。替换时替换掉引用位为 0 的(存在的话)。
Additional reference bits 附加引用位算法
在规定时间间隔内记录引用位。在规定的时间间隔内，时钟产生一个中断并且交控制权给 OS，OS 把每个页的引用位转移到其 8 位字节的高位，其他位向右移 1 位，抛弃最低位。这些 8 位寄存器包含着该页在最近 8 个周期内的使用情况，全 0 说明没用过，全 1 说明每个周期至少都用过 1 次，值越大越最近使用。有最小值的页是 LRU 页，被置换。被访问时左边最高位置 1，定期右移并且最高位补 0.
Second chance 二次机会/clock 算法 NRU
基本算法是 FIFO，选择页时，检查**当前指向**引用位，如果为 0 直接置换。如果为 1…
也有 **Belady’s Anomaly 现象，**可用物理帧越多，page fault 数越多的情况

一种实现二次机会算法的方法是采用循环队列，用一个指针表示下一次要置换哪一页。当需要一个帧时，指针向前移动直到找到一个引用位 0 的页，在其向前移动的过程中，它会清楚引用位。最坏情况下所有帧都会被给二次机会，他就会清除所有引用位之后再选择页进行置换。
Enhanced Second chance 改进 clock 增强二次机会
通过将引用位和脏位作为有序对来考虑，可以改进二次机会算法。
淘汰次序(0,0)-> (0,1)-> (1,0)-> (1,1)
当页面需要被置换时，使用时钟算法，置换(0,0)的页，在进行置换前可能要多次搜索循环队列。改进的点子在于给未引用但是修改了的页更高优先级，降低了 I/O 次数。
第一轮扫描寻找(0,0)，不改变引用位；第二轮扫描寻找(0,1)，遍历过的引用位均置为 0；第三轮扫描一定可以找到优质替换页。
Counting 基于计数的置换算法
为每个页保存一个用于记录引用次数的计数器，具体方案有两种：
Least frequently used LFU：置换计数最小的。但是有问题：一个页可能一开始狂用，但是后来不用了，他的计数可能很大，但是不会被替换。解决方法是定期右移次数寄存器。
Most frequently used MFU：置换计数最大的，因为最小次数的页可能刚调进来，还没来得及用。
这两种很没用，实现开销很大，而且还很难近似 OPT。
Page Buffering 页面缓冲
通过被置换页面的缓冲，有机会找回刚被置换的页。
被置换页面的选择和处理：用 FIFO 选择置换页，把被置换的页面放到两个链表之一。即：如果页面无修改，将其归入空闲页链表，否则归入已修改页面链表。
需要调入新页面时，将新页面内容读入空闲页面链表的第一项所指的页面，然后将其删除。

帧分配 allocation of frames
每个进程都需要最小数目的页。两种分配模式：
平均分配算法 Equal allocation
所有可供分配的空闲物理块分配给各个进程
按比例分配 Proportional allocation
根据进程的大小按比例分配。
优先级分配 Priority allocation
按比例分配，但是用优先级进行比例分配。
全局置换 global allocation
允许一个进程从所有帧中选择一个帧进行替换，不管该帧是否已分配给其他进程（物理帧使用数可增加）
局部置换 local allocation
每个进程只能从自己的分配帧中进行置换（物理帧使用数不变）
固定分配局部置换：物理块分配数目一定，发生缺页只能进行页面替换
可变分配全局置换：先分配一定数量的物理帧，若发生缺页，系统直接从空闲物理块中取出一块分配给该进程
可变分配局部置换：为每个进程分配一定数量的物理块；只有缺页发生过于频繁时，系统才为该进程分配额外的若干物理块；若进程缺页率较低，则可适当减少分配给该进程的物理块
Major/Minor Page Fault Major:访问的页不在内存中；Minor:访问的页在内存中（shared library；某页被 reclaimed 了但还没实际换出）

颠簸 抖动 Thrashing
频繁的页调度行为叫做颠簸（抖动），会导致 CPU 利用率低
给一个进程分配的物理页框集合为这个进程的**驻留集**
颠簸就等价于一个进程不断换入换出页
通过局部置换算法能限制系统颠簸。
L=S 准则：L 产生缺页的平均时间=S 系统处理进程缺页的平均时间
按需调页能成的原因是局部性原理，进程从一个局部性移动到另一个，局部性可能重叠。为什么颠簸会发生，因为局部大小大于总内存大小，不能将全部经常用的页放到内存中。

工作集合模型 Working set model
WS 工作集：最近Δ个页的引用集合（是一个 set，不记录重复页面，该时间点前）
Δ = 工作集窗口 = 固定数目的页引用
WSSi 进程 Pi 的工作集大小 = 在最近的Δ内总的页面引用次数（不重复页面出现的个数），如果Δ太小，不能包含整个局部；Δ太大，可能包含过多局部；Δ无穷工作集为进程执行所接触到的所有页的集合。
D = Σ WSSi = 帧的总需求量。m = 帧的总可用量。如果 D > m 就会发生颠簸。这时可以挂起某些进程，以消除颠簸现象。

页错误频率 Page fault frequency schema
WS 模型能用于预先调页，但是控制颠簸不是很灵活，更直接的方法是 PFF。
可以为所期望的页错误设置一个上限和下限，如果页错误率超过上限，那么分配更多的帧，如果低于下限，那么可以从进程中移走帧。
Memory-Mapped Files 内存映射文件
使用虚存技术将文件 I/O 转变为普通文件访问的技术。
开始的文件访问按照普通按需请求调度，会出现页错误。这样，一页大小的部分文件从文件系统中读入物理页，以后的文件访问就可以按照通常的内存访问来处理，这样就可以用内存操作文件，而非 read write 等系统调用，简化了文件访问和使用。多个进程可以将同一文件映射到各自的虚存中，达到数据共享的目的。

Allocating Kernel Memory 内核内存分配
与对待用户内存不同：内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续。
预调页 prepaging
为了减少冷启动时大量的 page fault。
将所有需要的页一起调入内存，但是如果预调页没有被用到，那么 I/O 就被浪费了。
假设 s 页被预调到内存，其中 a 部分被用到了。问题在于节省的 s*a 个 page fault 的成本是大于还是小于其他 s*(1-a)不必要的预调页开销。
如果 a 接近于 0，调页失败；a 接近 1，调页成功。
页大小
页大小的考虑因素：碎片⬇、页表大小⬆、IO 开销⬆、page fault 数量⬆、TLB 大小及效率⬆（箭头表示需要越大/小的页）
TLB 范围 TLB reach
TLB 范围指通过 TLB 可以访问到的内存量。
TLB Reach = TLB size（表示个数） * Page Size。理想情况下，每个进程的 WS 应该位于 TLB 中，否则就有不通过 TLB 调页导致的大量 I/O
增大页大小来缓解 TLB 压力；但可能会导致不需要大页表的进程带来的内碎片
提供多种页大小的支持：那么 TLB 无法硬件化，性能降低。
程序结构 Program Structure
调整循环次序/分块等减少 page fault
I/O 互锁 I/O interlock
I/O 互锁指一些页面有时必须被锁在内存中。必须锁住用于从设备复制文件的页，以便通过页面置换驱逐。

File System Interface 文件系统接口

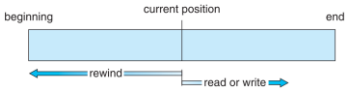
文件是存储某种介质上的（如磁盘、光盘、SSD 等）并具有文件名的一组相关信息的集合
文件属性（通过 FCB 记录）
名称/标识符(唯一标识该文件的数字)/类型/位置/大小/保护（访问控制信息 RWE 等）/时间日期/用户标识（所有者、创建者）
所有的文件信息都保存在目录结构中，而目录结构保存在外存上。
文件操作
文件是 ADT 抽象数据类型，其操作有：create/read/write/delete/open/close/Truncate（截断文件：文件所有属性不变，但删除文件内容，长度置 0 并释放空间（目录项依旧保持）/ Reposition within file（重定位，目录中文件的重定位，不涉及读写文件）
Open(Fi) 在硬盘上寻找/创建目录结构并且移动到内存中
Close(Fi)将内存中的目录结构移动到磁盘中
读文件前先用 open 打开文件，open 的参数包括文件名/路径等。**Read 只需要用 open 获得的文件描述符，不需要文件名。**
打开文件
每个打开文件都有以下信息：
文件指针：跟踪上次读写位置作为当前文件位置指针
文件打开计数器 file-open count：跟踪文件打开和关闭的数量，在最后关闭时，计数器为 0，系统可以从 open file table 移除该条目。
文件磁盘位置 disk location of file：用于定位磁盘上文件位置的信息
访问权限：访问模式信息
锁机制：mandatory lock:根据目前锁请求与占有情况拒绝 access；advisory lock:进程查看锁情况来决定访问策略
将文件属性从外存复制到内存打开文件表的一个表目中，并将索引编号返回给用户
整个系统有系统表，每个进程有进程表。对于同一个文件 F，在系统打开文件表中仅有一个表项包含 F 的属性，但各进程的用户打开文件表中关于 F 的表项内容不一定相同。当所有进程都关闭该文件时，才会从系统打开表中删除。
对于访问打开文件表的索引，Unix 称为文件描述符，windows 称为文件句柄
文件内部结构 File Structure
流文件结构：字节/bytes 流
Simple record structure 记录文件结构：lines, fixed length, variable length（相似记录的集合）
Complex Structures：formatted document, relocatable load file
可以通过插入适当的控制字符，用第一种结构模拟最后两个结构
这些模式由 OS 和程序所决定。
文件控制块 FCB(文件标识符是不跨 process)
用来存放控制文件需要的各种信息的数据结构，以实现“按名存取”
FCB 的有序集合是文件目录
包含基本信息（文件名/物理位置/逻辑、物理结构），存取控制信息（存取权限），使用

信息（建立时间/上次修改时间）
Unix 采用文件名与文件描述信息分开的方法，使用**索引节点**（文件描述信息单独形成的数据结构，称为 i 结点），文件目录中每个目录项仅含文件名与该文件对映 i 结点的指针
磁盘索引节点（文件唯一）：含有文件主标识符、类型、存取权限、物理地址、长度、存取时间、链接计数（指向该文件的文件名数量）
内存索引节点：文件被打开时，从磁盘复制至内存中，含有索引节点编号、状态、访问计数、逻辑设备号（所属文件系统）、链接指针（指向空闲链表与散列列表的指针）

访问方法（类似于文件的逻辑结构描述）

Sequential access 顺序访问

文件信息按顺序，一个记录接着一个记录处理。读操作读取文件下一文件部分，并自动前移文件指针，跟踪 I/O 位置。写操作向文件尾部增加内容，相应文件指针到新文件结尾。顺序访问基于文件的磁带模型，也适用于随机访问设备。可以重新设置指针到开始位置或者向前向后跳过记录；不允许超范围读。



Direct access 直接访问

文件由固定长度的逻辑记录组成，允许程序按任意顺序进行快速读写，直接访问是基于文件的磁盘模型。文件可作为块或记录的编号序列。读写顺序没有限制。可以立即访问大量信息。往往用指向 blocks 的 index 实现。
文件操作必须经过修改从而能将块号作为参数，有读 n 操作，而不是读下一个；写 n 操作；定位到 n；要实现读 n 只需要定位 n 再读下一个即可。

Indexed block access 索引顺序访问访问

目录：一个 FCB 就是一个文件目录项，FCB 的有序集合称为文件目录。

目录结构

目录是包含所有文件信息（属性）节点的集合。目录结构和文件在磁盘上。

磁盘结构

Disk 可以被 subdivide into 分区（又被称为 mini-disk，slice）;Disk/partition 可以用 RAID 维护；disk/partition 可以不需要 file system，直接 raw 使用；含有文件系统的 disk 称为 volume（卷）。含有文件系统的卷跟踪 device directory 或 volumn table of contents 中的文件系统信息。

目录操作

搜索文件/创建文件/删除文件/遍历 list 目录/重命名文件/遍历 traverse 文件系统

Organize the Directory (Logically) to Obtain:
Efficiency：快速定位一个文件；
Naming 重名：方便用户，多个用户不同文件同名，或相同文件不同名；
Grouping 分组：通过文件属性来逻辑上分组。

单级目录

整个文件系统只有一张目录表，所有文件包含在同一目录中，一个文件系统提供给所有用户。所有文件在同一级，从而不能有重名；创建/删除文件需要遍历，效率较低。也有分组问题

二级目录

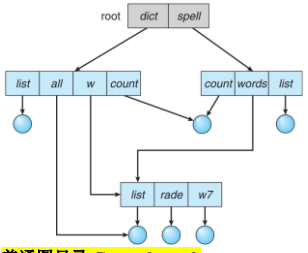
为每个用户创建独立目录。采用两级方案，有主文件目录（master file directory MFD）和用户文件目录（UFD）。
主文件目录记录用户名以及用户文件目录所在的存储位置；**用户文件目录**记录该用户文件的 FCB 信息。不同用户可以有同名文件，搜索效率高，但是没有分组分类能力。

树形目录

将目录二级目录拓展即可。搜索高效且有分组能力。引入绝对路径与相对路径的概念；文件访问都是相对于**当前目录（工作目录）**而言的；按路径名逐级访问中间节点 ---> 树形目录在文件查找上增加了磁盘访问次数。

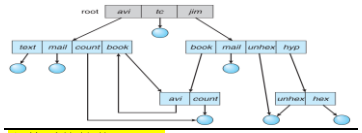
无环图目录 Acyclic-graph

树形结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件。
实现文件和目录共享，UNIX 采用创建一个叫做链接的新目录条目。链接实际上是另一个文件的指针。链接通过使用路径名定位真正文件。
注意，无环图目录对于一个文件可以有**多个绝对路径名**。不同文件名可能指向同一文件，因而存在 dangling pointer 问题：删除一个文件后指向该文件的其他链接会成为 dangling。
删除的另一方法是保留文件直到删除其全部引用，从而引入了计数，删除一次链接条目就计数 -1，到 0 时才完全删除文件，UNIX 的**硬链接采用这种方法**，在 inode 中保留一个引用计数。



普通图目录 General graph

采用这种目录必须确保没有环：仅允许向文件链接，不允许向目录链接。每次加入链接都要执行环检测算法。垃圾回收机制。



文件系统挂载 mount

文件系统在访问前必须挂载。**挂载**是将文件系统连接到文件系统层次结构中的特定位置（挂载点）的过程。**挂载点（安装点）的挂载会覆盖原有内容（基于子树的全部覆盖）**
挂载也称为文件系统的安装。

文件共享 file sharing

多用户系统的文件共享很有用。文件共享需要通过一定的保护机制实现；在分布式系统，文件可以通过网络访问；网络文件系统 NFS 是常见的分布式文件共享方法。NFS 是 UNIX 文件共享协议 CIFS 是 WIN 的协议。
基于索引结点的共享方式（硬链接）
不同用户的用户文件目录中都有目录项的指针指向同一索引节点；索引节点使用 count 计数，只有计数为 0 时才允许删除文件。
利用符号链实现的文件共享（软连接）
共享时创建 LINK 类型的新文件，在访问时若发现文件为 LINK 类型，则根据该文件中的路径名去寻找文件。只有文件主才拥有指向该文件索引节点的指针；在文件主删除共享文件，且其他用户又试图使用符号链去访问它时，访问才会失败。**符号链访问共享文件时，系统需要逐级查找目录，直到找到相应的索引节点。**
软/硬链接均为静态共享方法，动态共享方法允许两个进程同时对一个文件进行操作。

文件保护 Protection

访问类型：读（从文件读）、写（向文件写）、执行（文件装入内存并执行）、追加 append（新信息添加到文件尾部）、删除（删除文件）、列表清单 list（列出文件名与文件属性）

访问控制列表 access-control list ACL

三种用户类型：
拥有者 owner access：创建文件的用户
组 group access：一组需要共享文件且具有类似访问的用户
其他 public access：所有其他用户
在 UNIX 里，一个类型有 rwx 三个权限，所以一个文件需要 3*3=9 位说明文件访问权限。
口令和密码是另外两种访问控制方法
注意，其只是防止用户文件被他人存取或窃取，并没有控制用户对文件的访问类型

File System Implementation

文件系统：文件系统储存在二级存储中。文件系统按层来组织。

分层设计的文件结构

应用程序：发出文件请求者。
逻辑文件系统：管理元数据（文件系统的所有结构数据，不包括文件的实际数据）；逻辑文件系统管理目录结构；逻辑文件系统通过 FCB 来维护文件结构；文件保护。
文件组织模块：知道文件逻辑块到物理块的映射，包括空闲空间管理器。
基本文件系统：向合适的设备驱动程序发送指令以对磁盘上的物理块进行读写；在操作前需要分配合适缓冲区并对缓冲区进行管理。
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。

一般的文件系统布局

多数磁盘会划分为一个或多个分区，每个分区有一个独立的文件系统。
一般文件系统包括：
主引导记录（MBR，master root record）：位于磁盘 0 号扇区；后接分区表，给出了每个分区的起始和结束地址；MBR 需要确定活动分区，并读入第一块（引导块）。
引导块（boot block）：负责启动该分区的操作系统。
超级块（super block）：包含文件系统的所有关键信息；常包含分区块的数量，块的大小，空闲块的数量及指针，空闲块的 FCB 数量等。
文件系统中空闲块信息：一般用位图或指针链接形式给出。

文件系统实现

On-Disk File System structure

文件系统可能包括以下信息：如何启动所储存的操作系统；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。
磁盘结构包括：每个卷的引导控制块 boot control block 包括从该卷引导操作系统所需的信息；每个卷的卷控制块 volume control block 包括卷的详细信息：目录结构来组织文件；每个文件的 FCB。文件系统 open 的时候没有把 fcb 从 disk 整个加载并存储到 table，而是存储了 inode number，目录实现 filename 到 FCB 的映射（目录项：filename 和指向 FCB/Inode 的指针）

In-Memory File System structure

安装表（mount table，每个已安装文件系统分区的有关信息）；
An in-memory partition table 分区表
An in-memory directory structure 目录结构的缓存（最近访问目录的信息，可以包括一个指向分区表的指针）；
The system-wide open-file table 系统打开文件表；
The per-process open-file table 进程打开文件表。
对于同一个文件 F，在系统打开文件表中仅有一个表项包含 F 的属性，但各进程的用户打开文件表中关于 F 的表项内容不一定相同。当所有进程都关闭该文件时，才会从系统打开表中删除。

虚拟文件系统 VFS

VFS 提供面向对象的方法实现文件系统。允许将相同的系统调用接口（API）用于不同类型的文件系统，屏蔽了不同文件系统的差异与操作细节。
Linux 主要抽象了四种对象类型
超级块对象：对应于磁盘特定扇区的文件系统超级块，用于存储已安装文件系统的元信息。元信息中包含文件系统的基本属性信息，也有操作方法指针指向该超级块的操作方法表，包含分配 inode、销毁 inode、读 inode、写 inode、文件同步等函数。
索引节点对象：文件系统处理文件所需要的所有信息，都存放在索引结点的数据结构中，索引结点对文件是唯一的。
目录项对象：由于 VFS 经常执行切换目录操作，为提高效率，引入了目录项。目录项对象是一个路径的组成部分，可以是目录名，也可以是文件名。（硬链接对映同一个目录项对象）
文件对象：同一文件在内存中可能存在多个对应的文件对象（每个进程都可以打开一个相同的文件），但对应的索引结点和目录项唯一。
打开文件与文件对象一一对映，相同文件与索引节点对象一一对映。

目录实现

线性列表（linear list）：
使用储存文件名和数据块指针的线性表。实现简单，但采用线性搜索来查找，查找费时；需要考虑目录项的重用。许多操作系统采用软件缓存来储存最近访问过的目录信息，如 Linux 的目录项对象。
哈希表：采用 hash 数据结构；哈希表根据文件名得到一个值返回一个指向线性表中元素的指针。查找迅速/插入删除简单，减少了目录搜索时间，但需要避免冲突：两个文件名哈希到相同的位置。哈希表的最大困难是其通常固定的大小和哈希函数对大小的依赖性。

文件的逻辑结构（文件的内部）

无结构文件（流式文件）：Byte 为单位的流
有结构文件（记录式文件）
顺序文件：串结构（按时间顺序排列）；顺序结构）按关键字排列。N 条记录平均查找 N/2
索引文件：定长记录文件，直接计算 offset；变长记录文件（基于索引表）可以进行随机检索，本身只能顺序查找
索引顺序文件：将顺序文件中的所有记录分为若干组，为部分顺序文件建立一张索引表，每组仅第一项被索引表记录。分为 √N 组，平均查找 √N/2

文件的物理结构（存储结构）

分配方法 Allocation Method

常见的主要磁盘空间分配方法：连续、链接和索引。

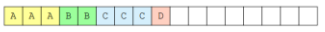
连续分配 Contiguous Allocation

每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需要起始块位置和块长度就可以读取。浪费空间，存在动态存储分配问题。存在外碎片问题，此外文件大小不可增长。
逻辑到物理的映射：
一种动态扩展的方法
Four files allocated contiguously to disk:

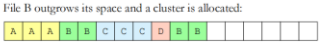
File B outgrows its space and is reallocated:

基于长度系统（Extent-Based Systems）的扩展。利于 Veritas FS 采用。解决了文件大小无法增长的问题，增加了另一 extent 连续空间给空间不够的文件，与原文件块之间有个指针。一个文件可以有多个 extent。文件块的位置(文件属性之一)为开始地址、块数、加上一个指向下一扩展的指针。

- Four files allocated contiguously to disk:



- File B outgrows its space and a cluster is allocated:

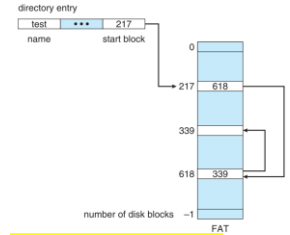


Linked Allocation 链接分配（显式链接）

解决了连续分配的所有问题。每个文件都是磁盘块的链表。访问起来只需要一个起始地址。没有空间管理问题，不会浪费空间，文件创建与增长容易。缺点：不能随机访问，因为每个索引块的末尾节点是用来链接下一个索引块的，不链数据块，必须要遍历；块与块之间的链接指针需要占用空间；存在可靠性问题。
簇：将多个连续块组成簇，磁盘以簇为单位进行分配。引入“簇”Cluster 的使用，在增大一定内部碎片的同时使得指针记录开销相对下降，分配簇比分配块更节省指针占用的空间，减少访问时间。

FAT File allocation table 文件系统（隐式链接）

磁盘空间分配用于 MS-DOS 和 OS/2。FAT32 引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。原本的链接分配指针在每个块中都会占空间，且任何指针丢失都会导致文件其余部分丢失。FAT 采用单独的磁盘区保存数据链接信息。
一般而言，可以使用-1 表示文件的最后一块，用-2 表示当前磁盘块为空闲；**FAT（文件分配表）**在操作系统启动时被读入内存。



Indexed Allocation 索引分配

索引分配把所有指针放在一起，通过索引块解决这个问题。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第 i 块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第 i 个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引、然后互相连接。访问需要索引表，支持 random access，动态访问没有外碎片，但有索引开销。

I 节点结构(ext2 文件系统，又称混合索引分配)

UNIX 采用的机制，inode 下 15 个指针，前 12 个直接指向 direct block，第 13 个指向 single indirect block, 第 14 个指向 double indirect block, 15 个指向 triple indirect block

索引计算

连续、链接组织的最大文件大小可以大到整个磁盘文件分区。
考虑每块大小 4KB，块地址 4B。
直接存储：文件系统中一条记录位置对映一个 block size 的存储空间
二级索引：一个索引块可以再继续连接到索引块，因此有 block size/entry size * block size 的最大文件。
三级索引同理，存储空间成倍上升，但 page access 访存次数也需相应提升。

空闲空间管理

位向量表和第一个成组链块往往存放在磁盘的卷头位置，需要在操作文件前将其读入内存。
位图（bit vector）：空闲块 0，分配块 1；
默认盘块号从 1 开始编号时，块分配计算为 b=n(i-1)+j，i 表示行，j 表示列
回收盘块时，可以进行如下转换：
i=(b-1)/n+1，j=(b-1)%n+1.
空闲链表管理：将磁盘上所有空闲块连接成链。当用户因创建文件而请求分配存储空间时，系统从链首开始，依次摘下适当数目的空闲盘块分配给用户；当用户因删除文件而释放存储空间时，系统将回收的盘块依次插入空闲盘块链的末尾。因为以盘块为单位，所以空闲盘块链可能会很长。
可以将空闲盘块链改进为空闲盘区链（每个盘区含有若干盘块）；每个盘区不仅有指向下一盘区的指针，还有指明当前盘区大小的信息；回收时需要进行空闲盘区的合并。
空闲表法：连续分配方式，空闲表记录“第一空闲盘块号 + 空闲盘块数”的信息；回收时需要考虑空闲区域的合并记录。
成组链接法：用来存放一组空闲盘块号的盘块称为**成组链块**，把顺序的 n 个空闲盘块号保存在第一个成组链块中，最后一个空闲盘块用于保存另一组空闲盘块号（充当成组链

块），不断套娃。
空闲盘块分配时，从第一个成组链块开始分配，指针逐一下移直至第一个成组链块的空闲盘块全部分配完毕，再分配第二个成组链块。
盘块的回收，记录回收盘号，不断上移指针，若当前成组链块已满，则回收盘块充当新一成组链块。

页面缓冲 page buffer

将文件数据作为页而不是磁盘块缓冲起来到虚存。

Unified Buffer Cache

A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

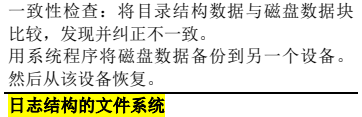
恢复 Recovery

一致性检查：将目录结构数据与磁盘数据块比较，发现并纠正不一致。
用系统程序将磁盘数据备份到另一个设备。然后从该设备恢复。

日志结构的文件系统

日志文件系统记录文件系统的更新为事务。事务会被写到日志里。事务一旦写入日志就是已经 commit 了，否则文件系统则还未更新。

FAT32 磁盘结构



FAT32 引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。
FAT 的每个目录项为 32 个字节
FAT2 是 FAT1 的备份，最大支持 32GB，但单文件支持最大为 4GB
exFAT（windows 也有使用）
克服单文件 4GB 的限制
DBR 及保留扇区 + FAT + 簇位图（多用于 U 盘，SD 闪存设备等）

NTFS file system（windows 使用）

引导扇区	主文件表(MFT)（前16个为系统文件）	文件区	MFT前16个元数据文件备份	文件区

MFT 由一个个 MFT 项（也称文件记录）组成，每个 MFT 项占用 1024 字节的空间。
MFT 前 16 个记录用来存放元数据文件的信息，它们占有固定的位置；后侧的备份本质上不属于 NTFS。

ReFS 文件系统

旨在最大限度地提高数据可用性、跨不同的工作负荷高效地扩展到大型数据集，并提供数据完整性，使其能够恢复损坏。旨在解决存储方案的扩展集问题。

功能	ReFS	NTFS
最大文件名称长度	255 个 Unicode 字符	255 个 Unicode 字符
最大路径名称长度	32K Unicode 字符	32K Unicode 字符
文件大小上限	35 PB (pb)	256 TB
最大卷大小	35 PB	256 TB

文件系统集合

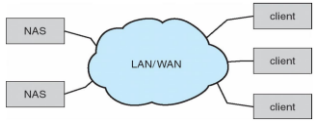
FAT（MS-DOS文件系统）
FAT32（VFAT），exFAT（64位）
NTFS（NT文件系统）
ReFS(Resilient File System)复原文件系统（Windows 8后、server）
S51K/S52K（AT&T UNIX sysv）
ext (minix文件系统)
ext2、ext3、ext4（linux文件系统、Android）
proc、sysfs（linux虚拟文件系统）
yaffs (Yet Another Flash File System)
ReiserFS (Linux一种日志文件系统)
HPFS（OS/2高性能文件系统）
UFS(BSD UNIX的一种文件系统)
HFS+ (Mac OS，iOS文件系统)
iso9660（通用的光盘文件系统）
NFS (网络文件系统)
VFS (Linux虚拟文件系统)
✓ VFS是物理文件系统与服务之间的一个接口，它屏蔽各类文件系统的差异，给用户和程序提供一个统一的接口
ZFS （Open Solaris文件系统）
LTFS(Liner Tape File System，线性磁带文件系统) 为磁带提供了一种通用、开放的文件系统。
APFS（Apple File System）苹果新一代的文件系统，MAC OS 10.13以后、iOS 10.3以后

Mass storage system 大容量存储

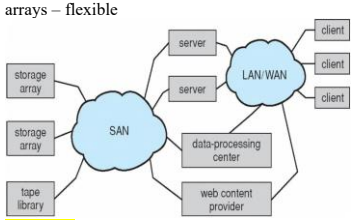
Host controller in computer uses bus to talk to disk controller built into drive or storage array
磁盘的 0 扇区是最外面的磁道的第一个扇区。
逻辑块时最小传出单位 512B
Platter 表面被分成了很多环形 track（磁道），再细分为 sector（扇区）。在一个 arm position 下的 track 组成一个 cylinder（柱面）。

Disk Attachment

三种方式：DAS(Direct/Host Attached Storage) /NAS(Network Attached Storage 网络附加存储)/SAN(Storage-Area Network 存储区域网)
Host-attached storage accessed through I/O ports talking to I/O busses
Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)



SAN: Multiple hosts attached to multiple storage arrays – flexible



磁盘调度

Position time =accesstime= seek time + rotational latency
IO time = position time + transfer time+control delay
Seek time 寻道时间，磁头移动到包含目标扇区的柱面的时间。
旋转延迟 rotational latency：旋转到目标扇区的时间（average latency=1/2*latency=1/2*1/rpm *60s，即平均情况下，旋转半圈，磁盘的转速一般以 round per minute(rpm) 表示）。
传输时间 transfer time：数据传输时间。
FCFS 先来先服务：算法公平，不会存在饥饿。
SSTF（shortest seek time first）最短寻道时间优先：处理靠近当前磁头位置的请求，本质上和 SJF 一样，有可能一些请求会永远无服务，时间非最优。
SCAN（电梯调度算法）：从磁盘一端到另一端服务请求，对所有路上经过的柱面进行服务。到达另一端时改变移动方向，继续处理。
C-SCAN 磁头从一端移动到另一端，到了另一端就马上返回到磁盘开始，返回路径中不服务。
LOOK：磁头从一端到另一端，到达另一端最远的服务就不继续向前，开始折返服务。
C-LOOK：磁头从一端到另一端，到达另一端最远服务就立即返回到磁盘开始的第一个服务，返回路径不服务。
调度算法选择：
SSTF 一般来说比较好，通用而自然。
SCAN C-SCAN 对于高负荷 IO 磁盘表现更好。
SSTF 和 LOOK 都是默认算法的合理选择

减少延迟时间：

交替编号，错位命名

磁盘管理

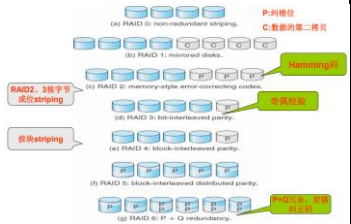
低级格式化（物理格式化、磁盘初始化）：将磁盘划分为扇区（使用特殊的物理结构），头部尾部添加一些磁盘控制器使用信息。会保留一些块作为备用（扇区备用）。
分区：将磁盘的若干柱面分区，每个分区的起始扇区和大小记录于主引导记录的分区表中。
逻辑（高级）格式化：创建文件系统。引导扇区产生在这里。
启动块 Boot Block：启动块初始化系统，引导(Bootstrap 自举)程序存储在 ROM 中，引导程序装载程序。
坏块 Bad Block：处理方法:format, chkdsk 指令

交换空间管理

Windows：存在 pagefile.sys 文件中，简单但低效（外部碎片）
Linux：独立的磁盘分区 SWAP 分区(内部碎片)

RAID

0：无冗余 1：镜像 2：纠错码 3：按 bit 对每个盘进行奇偶校验，结果放在 1 个盘 4：与 3



类似，按块条带化 Striping 5：校验值分散到各个盘 6：P+Q 冗余，差错纠正码

三级存储 Tertiary storage device

Low cost is the defining characteristic of tertiary storage. Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available

Swap space 三级存储 Tertiary storage device

虚存使用硬盘空间作为主存。两种形式：普通文件系统：win 都是 pagefile.sys 独立硬盘分区 linux solaris 都是 swap 分区。还有一种方法：创建在 raw 的磁盘分区上，这种速度最快。

性能：

Sustained bandwidth 大传输的平均速率 字节/时间。Effective bandwidth IO 时间下的平均速率。前者是数据真正流动时的速率，后者是驱动器能够提供的能力，一般驱动器带宽指前者。

NVM Devices¶

包括固态硬盘 (Solid-state disks, SSD), USB drives (thumb drive, flash drive), DRAM disk replacements 等。
NVM devices 比 HDD 更可靠，也更贵，可能寿命更短，容量更小，但是快很多。标准总线可能会太慢，所以有的 SSD 设计成直接连接到系统总线，如 PCI。
没有需要「移动」的部分，因此没有 seek time or rotational latency。

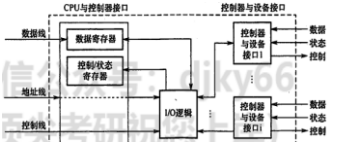
IO system

IO 接口：串行、并行、USB、键盘、硬盘
IO 端口：设备与计算机通信的连接点。端口地址。I/O 端口通常有四种寄存器，即状态、控制、数据输入与数据输出寄存器。
IO 方式：程序 IO，中断 IO（包括同步/异步），DMA 方式，通道方式。

I/O 设备

设备分类

块设备：信息交换以数据块为单位（结构设备，磁盘（使用 DMA I/O）等）；可寻址，可随机读写。
字符设备：信息交换以字节为单位（无结构设备，鼠标键盘打印机等），不可寻址常使用中断 I/O。（多为独占设备，互斥共享）
I/O 接口（设备控制器）
位于设备与 CPU 之间
设备控制器与 CPU 接口：数据线（数据寄存器/状态寄存器）、地址线、状态线。
设备控制器与设备的接口：一个设备控制器可以连接多个设备，也有数据、地址、状态三种类型信号。
I/O 逻辑：用于实现对设备的控制，对 CPU 的 I/O 命令进行译码并选择相应设备进行控制。



I/O 端口：设备控制器中可被 CPU 直接访问的寄存器。
数据寄存器：CPU 与设备间的数据缓冲。
状态寄存器：执行结果/设备状态寄存器。
控制寄存器：由 CPU 写入控制命令。

编址方式

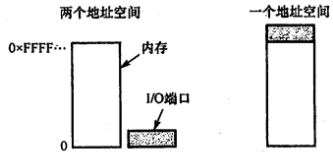


图 5.2 独立编址 I/O 和内存映射 I/O

独立编址：为每个端口分配 I/O 端口号，所有端口形成 I/O 端口空间。
统一编址（内存映射 I/O）：每个端口分配唯一的内存地址，通常靠近地址空间顶端。

IO 控制方式

Polling 程序直接控制方式（字为单位）：CPU 对外设状态进行轮询 polling，直到确定该字已经在 I/O 控制器的数据寄存器中。CPU 利用率相当低。
中断驱动方式（字为单位）：用户程序→系统调用处理程序→设备驱动程序→中断处理程序，中断处理结束后存放内容在内核缓冲区
DMA(direct memory access，块为单位)
用于避免程序 I/O for large data movement，需要 DMA controller，绕过 CPU 来直接在 IO 设备和内存之间传输数据
传送的数据，直接从设备进入内存，只需在传送一个/多个数据块的开始/结束时，CPU 进行干预处理即可。
任务分配给 DMA 控制器，在 DMA 开始传输时，主机向内存中写入 DMA 命令块，然后 DMA 自己操作内存总线，直接可向内存进行传输。传输结束后，DMA 发送中断信号给处理器。
其中 DMA 的 4 类寄存器
命令状态寄存器（CR）：接受 CPU 的 I/O 指令以及控制信息与设备状态。
内存地址寄存器（MAR）：输入时存放内存起始目标地址；输出时存放设备的内存源地址。
数据寄存器（DR）：暂存内存与设备间的数据。
数据计数器（DC）：存放本次传送的字节数。
IO 分类：
block I/O(read, write, seek);
character I/O (stream, keyboard, clock);
memory-mapped file access;
network sockets

I/O 软件层级结构

分为四层：中断处理程序，设备驱动程序，与设备无关的操作系统软件，以及用户软件(指用户空间的 IO 软件)
用户层 I/O 软件：与用户交互的接口（I/O 库函数）
设备独立性（设备无关性）软件：用于实现用户程序与设备驱动器的统一接口、设备命令、设备的保护及设备的分配与释放等，提供必要的存储空间
可以实现逻辑设备名到物理设备名的映射，便于实现重定向，增加设备分配的灵活性
设备驱动程序：每类设备配置一个设备驱动程序，通常以**进程**的形式存在；设备驱动程序向上层用户程序提供一组标准接口，设备具体的差别被设备驱动程序所封装（上层同一操作可操纵不同的硬件设备）
中断处理程序：用于保存被中断进程的 CPU 环境，转入相应的中断处理程序进行处理，处理完毕再恢复被中断进程的现场后，返回到被中断进程。
最后是硬件设备进行相应的操作。、

时钟和定时器 Clocks and Timers

提供获取当前时间/获取已经逝去的时间/设置定时器以在 T 时触发操作 X 三种函数。
测量逝去时间和触发器操作的硬件称为可编程间隔定时器（programmable interval timer）

内核 IO 子系统

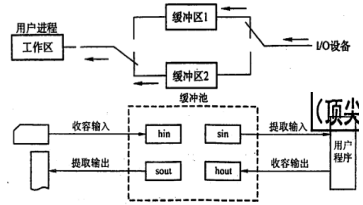
有关服务：IO 调度、Buffering、caching、spooling 虚拟化、device reservation、error handling
负责：文件和设备命名空间的管理，文件和设备访问控制，操作控制（for example,a moderm cannot seek()），文件系统空间的分配，设备分配，缓冲、高速缓存、假脱机，I/O 调度，设备状态监控、错误处理、失败恢复，设备驱动程序的配置和初始化

设备独立性软件

用户程序的设备独立性是：用户程序不直接使用物理设备名（或设备的物理地址），而只能使用逻辑设备名；而系统在实际执行时，将逻辑设备名转换为某个具体的物理设备名，实施 I/O 操作。
I/O 软件的设备独立性是：除了直接与设备打交道的低层软件之外，其他部分的软件并不依赖于硬件。I/O 软件独立于设备，就可以提高设备管理软件的设计效率。
逻辑设备是实际物理设备属性的抽象，它并不限于某个具体设备。
Disk cache（磁盘高速缓存）：利用内存中的存储空间来暂存从磁盘中读到的一系列盘块中的信息。逻辑上属于磁盘，物理上是驻留在内存中的盘块。
两种形式：在内存中开辟一个单独的空间作为磁盘高速缓存；把未利用的内存空间作为一个缓冲池。

缓冲区（buffer）

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
缓和 CPU 与 I/O 设备间速度不匹配的矛盾
减少对 CPU 的中断频率，放宽对 CPU 中断响应时间的限制
维持拷贝语义 copy semantics 的要求
解决基本数据单元大小不匹配的问题
提高 CPU 与 I/O 设备的并行性
假定 C 为数据处理时间，M 为数据从缓冲区传送到用户去的时间，T 为数据从磁盘传送到缓冲区的时间
单缓冲 最后加一个 C
单缓冲区处理每块数据用时为 max(C,T) +M
双缓冲 最后加一个 C+M
双缓冲区处理一块数据的用时为 max(C+M,T)



高速缓存与缓冲区的相同点与对比

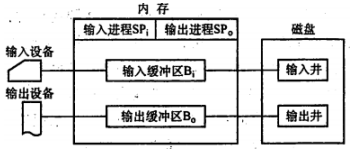
相同点	高速缓存	
区别	存放数据	存放的是低速设备上的某些数据的复制数据，即高速缓存上有的，低速设备上面必然有
	目的	高速缓存存放的是高速设备经常要访问的数据，若高速设备要访问的数据不在高速缓存中，则高速设备就需要访问低速设备

缓冲区	
速设备和低速设备之间	
存放的是低速设备传递给高速设备的数据（或相反），而这些数据在低速设备（或高速设备）上却不一定有备份，这些数据再从缓冲区传送到高速设备（或低速设备）	
高速设备和低速设备的通信都要经过缓冲区，高速设备永远不会直接去访问低速设备	

设备的分配与回收

每当进程向系统提出 I/O 请求时，设备分配程序按照一定的策略，把其所需的设备及其有关资源（如缓冲区、控制器和通道）分配给该进程。

根据设备的固有属性而采取的分配策略有：
独占设备：进程分配则独占设备
共享设备：可以同时分配给多个进程，**分时共享**使用
虚拟设备（SPOOLing 方式）：设备可以同时分配给多个进程。
设备分配算法：先来先服务/优先级高者优先
相关数据结构
设备控制表（DCT）：一个设备控制表表征一个设备，记录设备各项属性；需要维持申请该设备的 PCB 指针队列，有指向控制器表的指针。
控制器控制表（COCT）：有表项存放指向 CHCT 的指针；同时记录控制器队列。
通道控制表（CHCT）：记录服务的设备控制器表项。（一个通道可为多个设备控制器服务）
系统设备表（SDT）：整个系统只有一张 SDT，记录连接到系统的所有物理设备，每个物理设备占一个表目。
逻辑设备表（logical unit table LUT）：完成逻辑设备到物理设备的映射工作。（可以系统仅维护一张或每个用户自己维护一张）
SPOOLING 技术（假脱机技术）
Simultaneous Peripheral Operation On Line
通过这种技术处理过的设备称为**虚拟设备**。
用来保存设备输出的缓冲，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机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
现代操作系统基本都会采用假脱机
缓和 CPU 高速性与 I/O 设备低速性之间的矛盾。组成：预输入、井管理、缓输出程序。
输入井/输出井：磁盘上的两个存储区域，收容 I/O 设备的输入与进程的输出。
输入/输出缓冲区：为输入/输出井缓冲。
优点：可以提高 I/O 速度，将独占设备转化为共享设备，实现虚拟设备的功能。



IO 应用接口
实现统一的 IO 接口，设备驱动提供了 API 来操控 IO 设备(Linux：最底层为 **ioctl**)
设备分成很多种：
Sequential or random-access 顺序或随机访问；
Sharable or dedicated 共享或独占设备；
Speed of operation Operating System jjm 操作速度（快速、中速、慢速）
read-write, read only, write only 读写/只读/只写；
Synchronous I/O：包括阻塞 I/O 和非阻塞 I/O
阻塞 I/O:进程挂起直到 I/O 完成
非阻塞 I/O：I/O 调用立刻返回，返回尽可能多的数据。用户接口就是，接收鼠标键盘输入，还要在屏幕上输出，放视频也是，从磁盘读帧然后显示。
Asynchronous I/O：IO 与进程同时运行
非阻塞和异步的区别：非阻塞的 read 会马上返回，虽然可能读取的数据没有达到要求的，或者就没读到；异步会择时（可能延迟）运行，一定要完整执行完后才返回。
网络设备接口：网络套接字接口。
磁盘
磁盘是表面涂有磁性物质的物理盘片，通过**磁头**从磁盘存取数据。磁盘盘面数据存储在一组同心圆中，称为**磁道**。磁道又划分为若干**扇区**，一个扇区称为一个盘块，磁盘的存储能力受限于最内道的最大记录密度（每个扇区的存储大小一样）；所有盘片上相对位置相同的磁道构成**盘面**。磁道/扇区间均有间隙（横向/纵向）。
固定头磁盘：每个磁道一个磁头
活动头磁盘：磁头可移动
固态硬盘（SSD）：基于闪存技术的存储器。闪存翻译层充当磁盘控制器角色，没有寻道时间，没有旋转延迟。
磨损均衡
闪存的擦写寿命有限
动态磨损均衡：写入数据时，选择较新闪存块
静态磨损均衡：让老闪存块无需承担写数据的任务；为新的闪存块腾出读写操作的空间。

Windows 的优先级设置（优先级越高越优先）

线程相对优先级	进程优先级类					
	Idle	Below Normal	Normal	Above Normal	High	Real-Time
Time-critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

设备预定 Device Reservation
提供对设备的独占访问，分配和再分配的系统调用，有可能产生死锁
错误处理 Error Handling
操作系统可以恢复磁盘读，设备无效，暂时的失败。当 I/O 失败时，大多数返回一个错误码。系统日志记录了出错报告。
I/O Requests to Hardware Operations
Consider reading a file from disk for a process:
确定保存文件的设备，转换名字到设备的表示法，把数据从磁盘读到缓冲区中，通知请求进程数据现在是有效的，把控制权返回给进程

1. Linux unix WIndows 都是 monolithic
2. 对于其他的存储介质，由于没有磁盘的种种限制，因此 FCFS 即可
3. Win XP: 当一个线程 wakeup 的时候提升优先级
4. ELF 是可执行文件和链接库的格式
5. 首先源代码编译成 relocatable object file, 然后 linker 将多个目标文件和 library 链接成单个二进制可执行文件，此时程序 reside on secondary storage device 等待 loader 将其放入内存
6. 现代系统使用 Dynamic link library，dll 在运行的时候才加入内存
7. ELF 既可以被 linker 解析，作为可执行文件的一部分与其他 linker 链接，又可以被 loader 加载到内存里运行

ELF 文件包含以下部分
1. ELF header 放着文件类型 program header table 和 section header table， 程序的入口地址和 magic number
2. Program header table，描述了 program segment 在 ELF 文件中的位置（loader 使用）
3. Section header table 用于描述文件中各个 section 的信息，例如.text .data .bss(linker 使用)
4. .text 代码段
5. .rodata 只读数据段 const 常量
6. .data 数据段 非 0 非 const 的全局变量
7. .bss 未初始化全局变量，由 section 段说明，但是并未占据空间

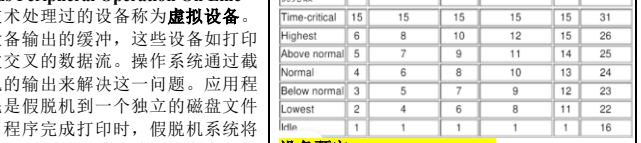
Links
1. 静态链接 将 object file 和 library 链接为一个可执行文件，静态链接的 ELF 中无.interp 段
2. 动态链接 在 ELF 文件的.interp section 记录动态链接库信息（.so 结尾），运行的时候动态加载
3. .interp 是 ELF 中第一个被加载的 section，其实放在 section header 中

运行可执行文件
1. Loader 面对 ELF 文件，需要将其信息映射到内存中，并把 dll 加载到内存

```
1. Who sets ELF file mapping? - kenel (exec syscall)
2. Who set stack and heap? - kenel (exec syscall)
3. Who setups libraries? - loader(ld-xxx)
```

2.

Main 从哪里开始
1. 静态链接 main 就是 _start
2. 动态链接会先进行动态加载


其中相对文件偏移 p_offset 指出相应 segment 的内容从 ELF 文件的第 p_offset 字节开始，在文件中的大小为 p_filesz，它需要被分配到以 p_vaddr 为首地址的虚拟内存位置，在内存中它占用大小为 p_memsz，也就是说，这个 segment 使用的内存就是 [p_vaddr, p_vaddr + p_memsz) 这一连续区间，然后将 segment 的内容从 ELF 文件中读入到这一内存区间，并将 [p_vaddr + p_filesz, p_vaddr + p_memsz) 对应的物理区间清零。(本段内容引用自南京大学 PA)

你也可以参考这篇 blog 中关于静态链接程序的载入过程来进行你的载入。

这里有不少例子可以学，为了避免同学们在实验中花太多时间，我们告诉大家可以怎么找到实验中这些相关变量：(注意以下的 _sramdisk 类型使用的是 char*，如果你在使用其他类型，需要根据你使用的类型去调整针对指针的算数运算)

- ELF64_Ehdr *ehdr = (ELF64_Ehdr *)(_sramdisk)，从地址 _sramdisk 开始，便是我们要找的 Ehdr
- ELF64_Phdr *phdrs = (ELF64_Phdr *)(_sramdisk + ehdr->phoff)，是一个 Phdr 数组，其中的每个元素都是一个 ELF64_Phdr
- phdrs + 1 是指向第二个 Phdr 的指针
- phdrs->p_type == PT_LOAD，说明这个 Segment 的类型是 LOAD，需要在初始化时被加载进内存

Syscall
1. Syscall 本身不是 privilege
2. Syscalld 的 state 存在 kernel stack 的 pr_regs 上
3. x86_64 用 syscall，32 用 int 0x80, arm 用 svc, riscv 用 ecall
4. 系统调用中的传参：可以用寄存器传参，或者将参数放在内存中，传对应的地址，或者直接存储在栈上

同步题目
1. 银行提供 1 个服务窗口和 10 个的等待位置，顾客到达的时候，若有空位，则到取号机上领取一个号，等待叫号，取号机仅有一个，当营业员空闲的时候，通过叫号选取一位顾客

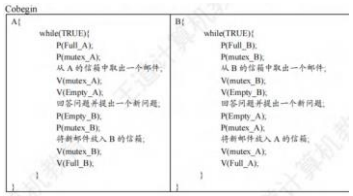
```
semaphore empty = 10;  // 空座位数量，初值为 10
semaphore mutex = 1;   // 互斥使用取号机
semaphore full = 0;    // 已占座位数量
semaphore service = 0; // 等待叫号
cobegin
Process 顾客 i {
    P(empty);       // 等待空位
    P(mutex);       // 申请使用取号机
    从取号机上取号;   // 取号
    V(mutex);       // 取号完毕
    V(full);        // 通知营业员有新顾客
    P(service);     // 等待营业员叫号
    接受服务;       // 接受服务
}

Process 营业员 {
    while (True) {
        P(full);    // 没有顾客则休息
        V(empty);   // 离开座位
        V(service); // 叫号
        为顾客服务;  // 为顾客服务
    }
}
```

2. 两人通过各自的信箱交流 A 信箱上限 M，初始为 x，B 信箱上限 N，初始 y

```
semaphore Full_A = x;      // Full_A 表示 A 的信箱中的邮件数量
semaphore Empty_A = M-x;   // Empty_A 表示 A 的信箱中还可存放的邮件数量
semaphore Full_B = y;      // Full_B 表示 B 的信箱中的邮件数量
semaphore Empty_B = N-y;   // Empty_B 表示 B 的信箱中还可存放的邮件数量
semaphore mutex_A = 1;     // mutex_A 用于 A 的信箱互斥
semaphore mutex_B = 1;     // mutex_B 用于 B 的信箱互斥
```

上下文切换过程
1. 上下文切换也是特权指令，只能放在 kernel 中
2. User process 调用 kernel entry 进入
3. Context switch 只保存 kernel 寄存器

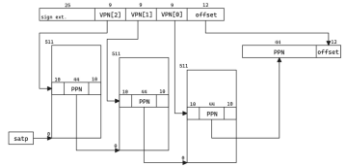


3. n 个哲学家，中心有 m 个碗，每两个哲学家中间一根筷子，使尽可能多哲学家就餐，但是不出死锁

```
semaphore bowl;// 用于协调哲学家对碗的使用
semaphore chopsticks[n]; // 用于筷子的使用

for (int i = 0; i < n; i++) chopsticks[i] = 1;
bowl = min(n - 1, m);   // bowl ≤ n-1，
Cobegin
while (TRUE) {          // 哲学家 i 的程序
    思考;               // 哲学家思考
    P(bowl);            // 取碗
    P(chopsticks[i]);   // 取左边的筷子
    P(chopsticks[(i + 1) % n]); // 取右边的筷子
    就餐;               // 哲学家就餐
    V(chopsticks[i]);   // 放下左边的筷子
    V(chopsticks[(i + 1) % n]); // 放下右的筷子
    V(bowl);            // 放下碗
}
CoEnd
```

1. 进程是 Resource allocation 和 protection 的单元，而线程是 execution 的单元
4. 捕获异常
5. 寻找当前 task 中导致产生了异常的地址对应的 VMA
6. 如果当前访问的虚拟地址在 VMA 中没有记录，即是不合法的地址，则运行出错（本实验不涉及）
7. 如果当前访问的虚拟地址在 VMA 中存在记录，则需要判断产生异常的原因：
8. 如果是匿名区域，那么开辟一页内存，然后把这一页映射到产生异常的 task 的页表中
9. 如果不是，则访问的页是存在数据的（如代码），需要从相应位置读取出内容，然后映射到页表中
10. 返回到产生了该缺页异常的那条指令，并继续执行程序



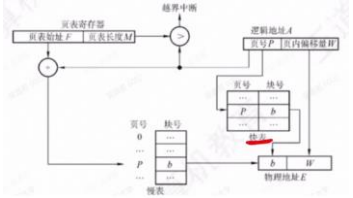
Buddy 系统
从物理上连续的大小固定的段上进行分配。
内存分配按 2 得幂的大小来分配：
请求大小必须是 2 的幂；如果不是，那么调整到下一个更大的 2 得幂；当需要比可用的更小的分配时，当前块分成两个下一个较低幂的段。继续这一过程直到适当大小的块可用。
Buddy 系统的优点是可以通过合并快速形成更大的段。明显缺点是由于调整到下一个 2 的幂容易产生内碎片。

静态代码 → 动态进程过程中的内存转换
- compile time - 将符号地址（**symbol address**）转换为可重定位地址（**relocatable address**，相对量）地对地址（**absolute address**，但是一旦起始地址发生改变，就需要重新编译）
- load time - 将可重定位地址转换为绝对地址，此时如果起始地址改变只需重新装载
  - 动态装载（不需要操作系统支持）：未被调用的进程的可重定位地址存在磁盘；被调用时装载进内存
  - 动态链接库/共享库（需要操作系统支持）：能被动态链接的库
- execute time - 如果进程允许被移动，将可重定位地址到绝对地址的转换放在这一步

在 context switch 的时候，我们需要清空 TLB，进行 flush 操作，否则会产生错误。
段页式管理：一个进程一个段表，一个段一个页表
Compiler-linker-loader
不管进程是否执行，其页表都驻留在内存中
计算机启动过程：CPU 加电，CS:IP 指向 FFFF0H → 执行 JMP 指令跳转到 BIOS → 登记 BIOS 中断程序入口地址 → 硬件自检 → 进行操作系统引导
处理外部中断时：
由硬件保存（中断隐指令自动保存）：PC（断点）、PSW（程序状态字）内容，TLB、cache 内容；将 CPU 模式改为内核态；
由操作系统完成：保存通用寄存器内容，执行系统调用的服务例程，保存中断屏蔽字
virtual devices are implemented by SPOOLing
RAID5 - provides high reliability inexpensively
NVM（flash memory）没有磁盘的种种性质，使用 FCFS 就可以了
linux 把 I/O 设备当作特殊文件
1. 磁头=/w heads→柱面cylinders→磁道tracks - 寻道时间seek time
2. 磁头→扇区sectors - 旋转时延rotational latency - 平均旋转时延 $\frac{1}{2}\cdot\frac{1}{rpm}\cdot 60s$
   - 和寻道时间相加是average access time（开销大头，距离越远开销越大）
3. 数据传输 - 传输时间transfer time = $\frac{\text{data to transfer}}{\text{transfer rate}}$

此外计算average I/O time的时候还要加上控制器开销controller overhead

衡评估I/O性能的指标为disk bandwidth = $\frac{\text{传输数据量}}{\text{请求开始到传输完成的时间间隔}}$
段式存储的分段由用户编程决定，不方便分区代价最小，在动态装入的时候建立
页式存储不可静态重定位
形成逻辑地址是 link



- 虚拟存储器的最大容量由计算机的地址结构决定
- 虚拟内存都是执行期间内存绑定
- 对换区大小和进程优先级与抖动无关
- 虚拟内存的实现只能建立在离散分配的内存管理基础上
- 虚拟内存的实际容量受外存和内存容量之和的限制，虚拟内存的最大容量是由计算机的地址位数决定的
- 产生缺页中断说明已经正常进行到查询页表的阶段
- 对各进程进行固定分配时页面数不变，不可能出现全局置换
- 页缓冲队列是将被淘汰的页面缓存下来，暂时不写回磁盘，队列长度会影响页面置换的速度，但是不会影响缺页率
- Unix 使用 Unix FS (UFS)，它基于 BFFS (Berkeley Fast FS)：
- Windows 支持 File Allocation Table(FAT), FAT32 和 Windows NT File System (NTFS)：
- Linux 的标准文件系统是 extended file system，最常见的是 ext3 和 ext4。不过 Linux 支持 40+ 种 FS。
- VFS - 支持 multiple file systems

Linux 中的目录是文件，只是它的 data block 存储的不是数据而是目录项，Windows 中的目录结构并不是文件
hard Link：硬链接 - 目录项，表现为一个指向文件的目录 entry，不能跨文件系统
soft Link：软链接 - path（绝对路径从 root 开始），表现为一个包含另外一个文件路径的文件，可以跨文件系统
使用 hard link 时，link()syscall 接收两个参数：old pathname 和 new pathname,会新创建一个目录 entry，没有新的 inode 创建，一个空的目录也有 2 个 hard link，一个指向当前目录，一个指向父目录。删除文件的时候，使用 unlink()，会删除从文件名到 inode 的映射，此时检查是否有其他的文件正在使用这个文件(reference count)，如果没有使用那么删除 inodeDynamic relocation relies on a relocation register
- 在程序运行时，CPU 会通过重定位寄存器调整指令中的地址，使得程序可以正确地访问物理内存。

UNIX 的 FCB 使用了一种混合的实现，ext2_inode 中前 12 个 pointer 直接指向数据块，后面 3 个指向 index block。
一个包含了文件系统的 Partition 被称为一个 Volumn(卷)，在磁盘上。
(3)采用(1)中的分页存储管理方式，一个代码段起始逻辑地址为 0000 8000H，其长度为 8KB，被装载到从物理地址 0090 0000H 开始的连续主存空间中。页表从主存 0020 0000H 开始的物理地址处连续存放。如下图所示(地址大小自下向上递增)，请计算出该代码段对应的两个页表项物理地址。这两个页表项中的帧号以及代码页面 2 的起始物理地址。

物理地址 1 = 0020 0020H
物理地址 2 = 0020 0024H
物理地址 3 = 0090 1000H
磁盘里的：boot control block, volumn control block, directory, per-file FCB
内存里的：mount table, directory cache, global open-file table, per-process open-file table, buffer cache