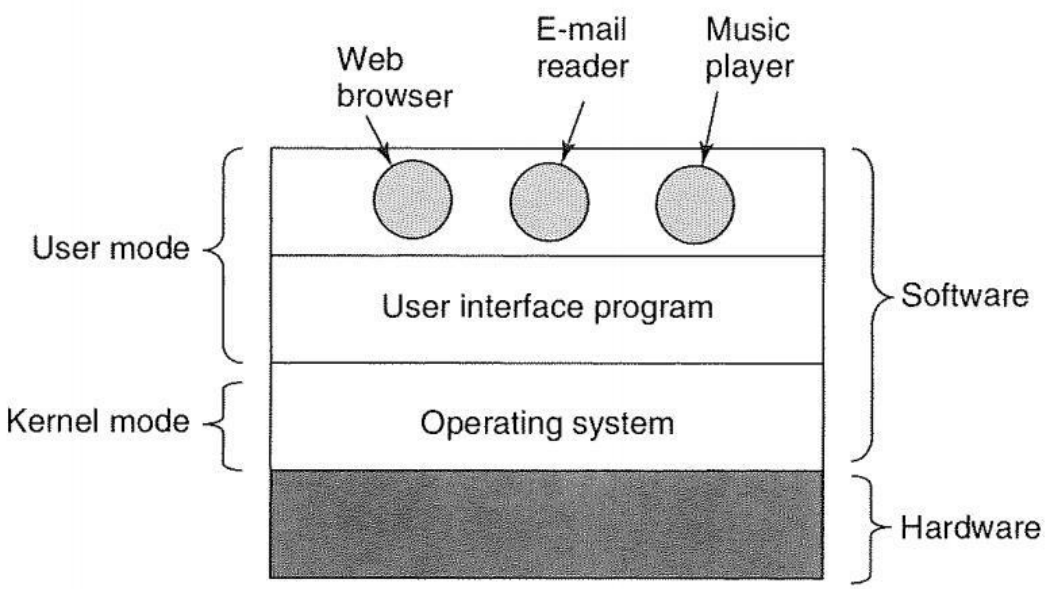
现代操作系统[1]

1、导论

与用户交互的程序：

- 基于文本的 shell
- 基于图标的图形化用户界面（GUI）

操作系统所处的位置：



多数计算机有两种运行模式：

- 内核态（管态），操作系统运行在此模式，能够执行任何指令。
- 用户态，用户软件运行在此模式，使用机器指令中的子集。

操作系统的功能：

- 为用户程序提供抽象
- 管理计算机资源

抽象是管理复杂性的一个关键。好的抽象可以把一个几乎不可能管理的任务划分为两个可管理的部分：

[1] 引自：http://blog.csdn.net/houjian914/article/details/50762056

- 有关抽象的定义和实现
- 随时用这些抽象解决问题

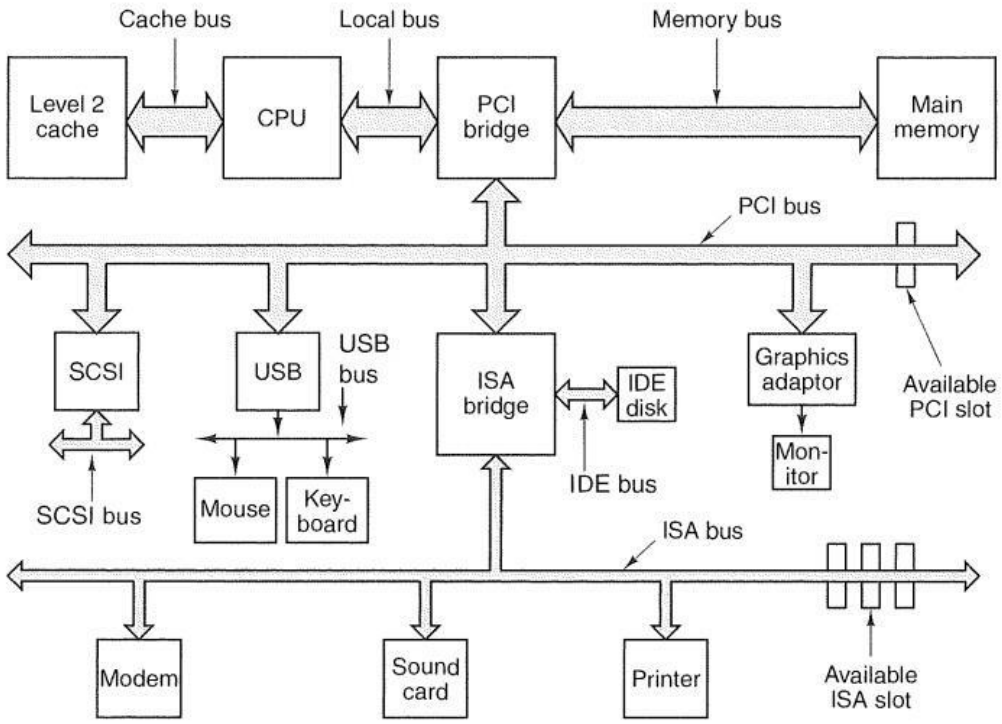
操作系统发展历史

1. 1945~1955：真空管和穿孔卡片
2. 1955~1965：晶体管和批处理系统
3. 1965~1980：集成电路芯片和多道程序设计
4. 1980~至今：个人计算机

计算机硬件

- CPU
- 存储器
- 磁盘
- 磁带
- I/O 设备
- 总线

大型 Pentium 系统结构：



Pentium 系统的启动过程：

- 主板上有一个基本输入输出系统（BIOS），其中有底层 I/O 软件。
- 计算机启动时，BIOS 开始运行。
- 首先检查安装 RAM 数量，键盘和其他基本设备是否已安装并正常相应。
- 开始扫描 ISA 和 PCI 总线并找出连接在上面的所有设备，记录下来。
- 如果现有设备和系统上一次启动时的设备不同，则配置新的设备。
- BISO 通过存储在 CMOS 存储器中的设备清单决定启动设备。
- 启动设备上的第一个扇区被读入内存并执行。
- 启动扇面末尾的分区表检查的程序，确定哪个分区是活动的。
- 从活动分区读入第二个启动装载模块，装在模块被读入操作系统。
- 操作系统询问 BIOS，以获得配置信息。
- 系统检查每种设备驱动程序是否存在，有就将设备驱动程序调入内核。
- 系统创建背景进程，在终端上启动登录程序或 GUI。

操作系统概念

进程：

进程本质是正在执行的一个程序。与一个进程有关的所有信息，除了该进程自身地址空间的内容以外，均存放在操作系统的一张表中，称为进程表（数组或链表结构）。

地址空间：

现代操作系统通常使用虚拟内存技术。操作系统可以把部分地址空间装入主存，部分留在磁盘上，在需要时再交换它们。

文件：

大多数系统都有目录结构，目录项可以是文件或者目录，构成了一种层次结构（文件系统）。进程和文件层次都可以组织成树状结构，一般进程的树状结构层次不深，而且是暂时的；文件树的层次常常多达四层、五层或者更多层，存在时间可能达数年。

输入/输出：

所有计算机都有用来获取输入和产生输出的物理设备。包括键盘、显示器、打印机等。

保护：

例如 UNIX 系统中对文件实现保护，三个 3 位保护字段（rwxrwxrwx），分别表示所有者、所有者同组用户、其他用户的读、写、执行权限。

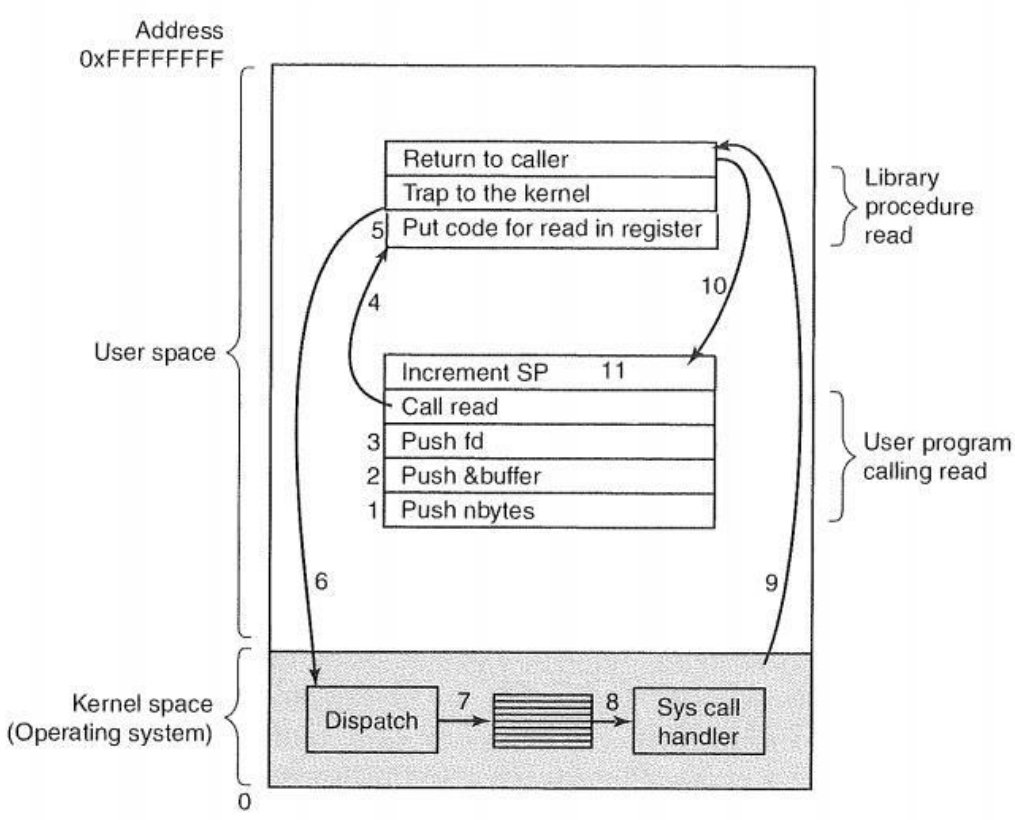
shell：

UNIX 的命令解释器称为 shell，不是操作系统的一部分。shell 是终端用户与操作系统之间的界面，除非用户使用的是 GUI 界面。

系统调用

系统调用 `read(fd, &buffer, nbytes)`函数的过程：

1. 参数 nbytes 压栈
2. 参数&buffer 压栈
3. 参数 fd 压栈
4. 对库过程 read 进行实际调用
5. 把系统调用的编号放在寄存器中
6. 执行 TRAP 指令，切换到内核态，在内核中一个固定地址开始执行
7. 内核代码检查系统调用编号，发出系统调用处理指令
8. 系统调用句柄执行
9. 控制返回给用户空间库过程
10. 以通常的过程调用返回的方式，返回到用户程序
11. 用户程序清除堆栈空间



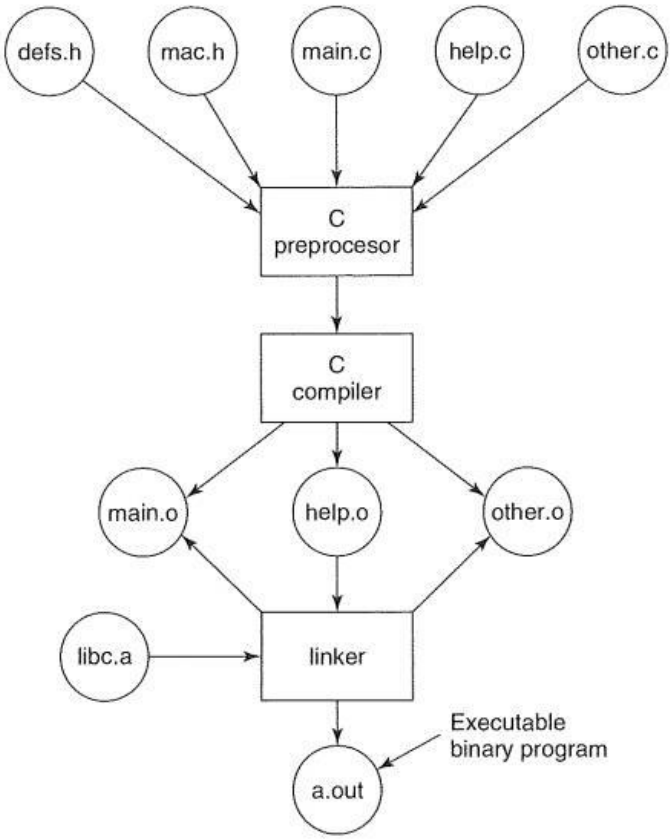
操作系统结构

- 单体系统
- 层次式系统
- 微内核
- 客户机-服务器模式

- 虚拟机
- 外核

C 语言

编译 C 和头文件，构件可执行程序的过程：



2、进程与线程

进程

进程模型

操作系统中最核心的概念是进程：这是对正在运行程序的一个抽象。
一个进程就是一个正在执行程序的实例、包括程序计数器、寄存器和变量的当前值。

在多道程序设计中，一个 CPU 能在多个进程之间来回快速切换，达到（伪）并行效果。

一个进程是某种类型的一个活动，它有程序、输入、输出以及状态。
单个处理器可以被若干进程共享，它使用某种调度算法巨顶何时停止一个进程的工作，并转而为另一个进程提供服务。

创建进程

4 种主要事件导致进程的创建：

1. 系统初始化
2. 执行了正在运行的进程所调用的进程创建系统调用
3. 用户请求创建一个新进程
4. 一个批处理作业的初始化

在 UNIX 系统中，只有一个系统调用可以用来创建新进程：`fork`。
这个系统调用会创建一个与调用进程相同的副本。
在调用 `fork` 后，这两个进程（父进程和子进程）拥有相同的存储映像、同样的环境字符串和同样打开的文件。
通常子进程接着执行 `execve` 系统调用，以修改其存储映像并运行一个新的程序。

在 Windows 系统中，一个 Win32 函数调用 `CreateProcess` 即处理进程的创建，也负责把正确的程序装入新的进程。

在 UNIX 中，子进程的初始地址空间是父进程的一个副本，不可写的内存区是共享的，新进程有可能共享其创建者的其他资源。
在 Windows 中，从一开始父进程的地址空间和子进程的地址空间解释不同的。

进程的终止

进程的终止通常由以下条件引起：

1. 正常退出（自愿的）
2. 出错退出（自愿的）
3. 严重错误（非自愿）
4. 被其他进程杀死（非自愿）

进程的层次结构

在 UNIX 系统中，进程只有一个父进程，但可以有多个子进程。
进程和它的所有子女以及后裔共同组成一个进程组。

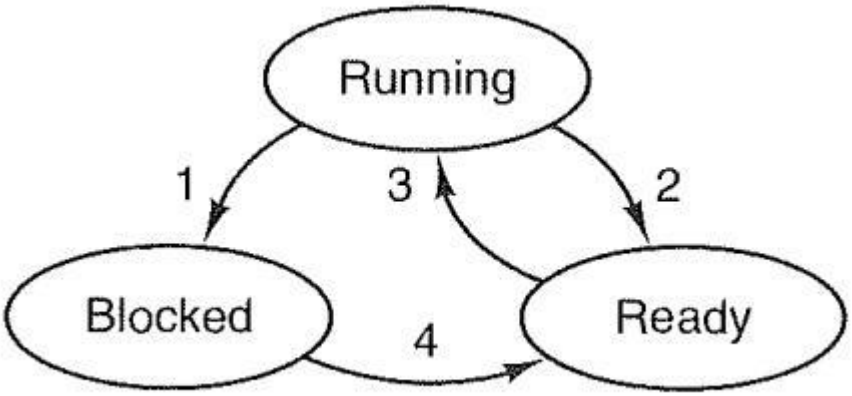
Windows 系统中没有进程层次的概念，所有的进程都是地位相同的。

进程的状态

进程的三种状态：

1. 运行态（该时刻进程实际占用 CPU）

2. 就绪态（可运行，但应为其他进程正在运行而暂时停止）
3. 阻塞态（除非某种外部事件发生，否则进程不能运行）



1. 进程为等待输入而阻塞
2. 调度程序选择另一个进程
3. 调度程序选择这个程序
4. 出现有效输入

进程的实现

操作系统维护着一张表（一个结构数组），即进程表（process table）。
每个进程占用一个进程表项（又称进程控制块）。

终端发生后操作系统最底层的工作步骤：

1. 硬件压入堆栈程序计数器等。
2. 硬件从中断向量装入新的程序计数器。
3. 汇编语言过程保存寄存器值。
4. 汇编语言过程设置新的堆栈。
5. C 终端服务例程运行（典型地读和缓冲输入）。
6. 调度程序决定下一个将运行的进程。
7. C 过程返回至汇编代码。
8. 汇编语言过程开始运行新的当前进程。

2. 线程

线程的使用

线程是一种轻量级的进程。

- 多个线程拥有共享同一个地址空间和所有可用数据的能力。
- 线程比进程更容易创建和销毁。

- 在大量计算和大量 I/O 处理过程中，多个线程能够加快程序执行速度。

经典的线程模型

进程模型基于两种对立的概念：资源分组处理与执行。
理解进程的一个角度是，用某种方法把相关的资源集中在一起。
另一个概念是，进程拥有一个执行的线程。
线程拥有自己的程序计数器、寄存器、堆栈。
进程用于把资源集中到一起，而线程则是在 CPU 上被调度执行的实体。

线程可以处于若干状态中的任何一个：运行、阻塞、就绪或终止。

POSIX 线程

为实现可移植的线程程序，IEEE 指定了线程的标准。
它定义的线程包叫做 Pthread，大部分 UNIX 系统都支持该标准。

所有 Pthread 线程都有某些特性。
每一个都含有一个标识符、一组寄存器（包括程序计数器）和一组存储在结构中的属性。
这些属性包括堆栈大小、调度参数以及使用线程需要的其他项目。

线程调用	描述
`pthread_create`	创建一个新线程
`pthread_exit`	结束调用的线程
`pthread_join`	等待一个特定的线程退出
`pthread_yield`	释放 CPU 来运行另外一个线程
`pthread_attr_init`	创建并初始化一个线程的属性结构
`pthread_attr_destory`	删除一个线程的属性结构

线程包的实现方式

在用户空间中实现

把整个线程包放在用户空间中，内核对线程包一无所知。从内核角度考虑，就是单线程进程。
线程在一个运行时系统的顶部运行，这个运行时系统是一个管理线程的过程的集合。
每个进程有其专用线程表（thread table）。

优点：用户线程包可以在不支持线程的操作系统上实现。
不需要陷进，不需要上下文切换，不需要对内存高速缓存进行刷新，使得线程调度非常快。
允许每个进程有自己定制的调度算法。

问题：如何实现阻塞系统调用，使用阻塞调用会阻塞其他的线程。
页面故障问题，如果某个调用跳转到了一条不再内存的指令上，就会发生页面故障，内核由于不知道线程的存在，通常会把整个进程阻塞到 I/O 完成。
如果一个线程开始运行，那么该进程中的其他线程就不能运行，除非第一个线程自动放弃 CPU。

在内核中实现线程

此时不需要运行时系统，内核中有用来记录系统中所有线程的线程表。
当一个线程阻塞时，内核根据其选择，可以运行同一个进程中的另一个线程或者运行另一个进程中的线程。

优点：内核很容在线程阻塞时切换到另一个线程执行。
内核线程不需要任何新的、非阻塞系统调用。

问题：在内核中创建或销毁线程的代价比较大。
进程创建问题，一个多线程进程创建新线程出现的问题。
当信号到达时，应该有哪一个线程处理。

混合实现

使用内核线程，然后将用户级线程与某些或者全部内核线程多路复用起来。
内核只识别内核线程，并对其进行调度。一些内核线程会被多个用户级线程多路复用。

这一模型能够带来最大的灵活度。

调度程序激活机制

当内核了解到一个线程被阻塞之后，内核通知该进程的运行时系统，并且在堆栈中以参数的形式传递有问题的线程的编号和所发生事件的一个描述。
内核通过在一个已知的起始地址启动运行时系统，从而发出通知，这是对 UNIX 中信号的一种粗略模拟。
这个机制称为上行调用（upcall）。

调度程序激活机制的一个目标是作为上行调用的信赖基础，这是一种违反分层系统内在结构的概念。

进程间通信

进程间通信（Inter Process Communication，IPC）简要的说有三个问题：

1. 一个进程如何把信息传递给另一个。
2. 确保两个或更多的进程在关键活动中不会出现交叉。
3. 保证进程以正确的顺序执行。

竞争条件

在一些操作系统中，协作的进程可能共享一些彼此都能读写的公用存储区。
两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序，称为竞争条件（race condition）。

临界区

阻止多个进程同时读写共享的数据可以通过互斥（mutual exclusion）。
确保党一个进程在使用一个共享数据时，其他进程不能做同样的操作。
我们把对共享内存进行访问的程序片段称作临界区（critical section）。

一个好的解决方案，需要满足一下 4 个条件：

1. 任何两个进程不能同时处于临界区。
2. 不应对 CPU 的速度和数量做任何假设。
3. 临界区外运行的程序不得阻塞其他进程。
4. 不得使进程无限期等待进入临界区。

忙等待的互斥

屏蔽中断

每个进程在刚刚进入临界区后立即屏蔽所用中断，并在就要离开之前再打开中断。

适用于单核操作系统，对操作系统本身而言很有用，单对于用户进程则不是一种合适的互斥机制。

锁变量

共享锁变量，初始为 0。当进程想进入临界区时，首先测试这把锁。
如果锁为 0，则进程将所设置为 1 并进入临界区。
如果锁为 1，则进程将等待其值变为 0。

这种方式同样存在竞争条件。

严格轮换法

连续测试一个变量直到某个值出现为止，称为忙等待。
这种方式浪费 CPU 时间，通常应该避免。
只有在有理由认为等待时间是非常短的情形下，才使用忙等待。
用于忙等待的锁，称为自旋锁（spin lock）。

Peterson 解法

```
#define FALSE   0

#define TRUE    1
```

```
#define N       2                       /* 进程数量 */


int turn;                               /* 现在轮到谁？ */

int interested[N];                      /* 所有值初始化为 0(FALSE) */


void enter_region(int process)          /* 进程是 0 或 1 */
{
    int other;                          /* 其他进程号 */


    other = 1 - process;                /* 另一方进程 */
    interested[process] = TRUE;         /* 表明所感兴趣的 */
    turn = process;                     /* 设置标志 */
    while (turn == process && interested[other] == TRUE);
/* 空语句 */
}


void leave_region(int process)          /* 进程：谁离开？ */
{
```

```
    interested[process] = FALSE;          /* 表示离开临界区 */

}
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

```
19
20
21
```

TSL 指令

需要硬件支持的一种方案。
某些计算机中，特别是那些设计为多处理器的计算机。
都有指令 `TSL RX, LOCK`。
称为测试并加锁（Test and Set Lock），他将一个内存字 lock 读到寄存器 RX 中，然后再该内存地址上存一个非零值。
读字和写字操作保证是不可分割的。

一个可替代 TSL 的指令时 XCHG，它原子性的交换两个位置的内容。

睡眠与唤醒

Peterson 解法和 TSL 或 XCHG 解法都有忙等待的缺点。
这种方法不仅浪费了 CPU 时间，而且还可能引起预想不到的结果。

生产者-消费者问题。两个进程共享一个公共的固定大小的缓冲区。
其中一个是生产者，将信息放入缓冲区；另一个是消费者，从缓冲区中取出信息。

信号量

信号量（semaphore）是 Dijkstra 在 1965 年提出的一种方法，使用一个整型变量来累计唤醒次数。

两种操作：down 和 up。
对一信号量执行 down 操作，则是检查其值是否大于 0。
若大于 0，则将其值减 1 并继续；若为 0，则进程将睡眠，此时 down 操作并未结束。
up 操作对信号量的值增 1。
如果一个或多个进程在该信号量上睡眠，无法完成一个先前的 down 操作，则有系统选择其中一个允许进程完成它的 down 操作。

信号量可以用来实现同步（synchronization）。

互斥量

如果不需要信号量的技术能力，有时可以使用信号量的一个简化版本，称为互斥量（mutex）。
互斥量是一个可以处于两态之一的变量：解锁和加锁。

一些与互斥量相关的 pthread 调用：

线程调用	描述

线程调用	描述
pthread_mutex_init	创建一个互斥量
pthread_mutex_destroy	撤销一个已存在的互斥量
pthread_mutex_lock	获得一个锁或阻塞
pthread_mutex_trylock	获得一个锁或失败
pthread_mutex_unlock	释放一个锁

一些与条件变量相关的 pthread 调用：

线程调用	描述
pthread_cond_init	创建一个条件变量
pthread_cond_destroy	撤销一个条件变量
pthread_cond_wait	阻塞以等待一个信号
pthread_cond_signal	向另一个线程发信号来唤醒它
pthread_cond_broadcast	向多个线程发信号来让他们全部唤醒

管程

一个管程（monitor）是一个由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包。
进程可在任何需要的时候调用管程中的过程，但他们不能在管程之外声明的过程中直接访问管程内部的数据结构。
管程是一种语言概念，C 语言不支持。

消息传递

消息传递（message passing）面临许多问题和设计难点，特别是位于网络中不同机器上的通信进程的情况。
消息系统还需要解决进程命名问题。身份认证（authentication）也是一个问题。
消息从一个进程复制到另一个进程通常比信号量操作和进入管程要慢。

屏障

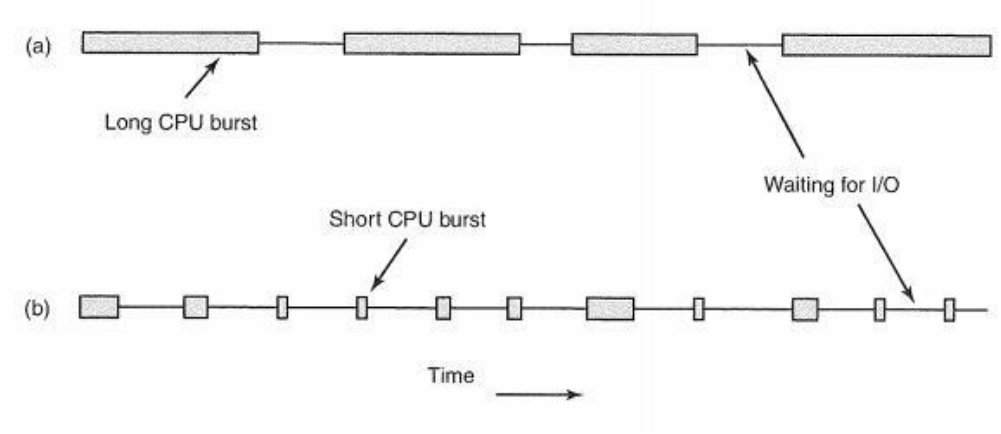
屏障是用于进程组而不是用于双进程的生产者-消费者情形的。
某些应用中划分了若干阶段，除非所有的进程都准备着手下一个阶段，否则任何进程都不能进入下一个阶段。
可以通过在每个阶段的结尾安置屏障（barrier）来实现。

调度

多道程序设计系统中，多个线程或进程同时竞争 CPU，当只有一个 CPU 可用时，那么久必须选择下一个要运行的进程。
完成选择工作的这一部分称为调度程序（scheduler），该程序使用的算法称为调度算法（scheduling algoritgm）。

进程行为

几乎所有进程的（磁盘）I/O 请求或计算都是交替突发的。
CPU 不停顿地运行一段时间，然后发出一个系统调用以便读写文件。
在完成系统调用之后，CPU 又开始计算，直到它需要读取更多的数据或写更多的数据为止。



某些进程花费了绝大多数时间在计算上，而其他进程则在等待 I/O 上花费了绝大多数时间。
前者称为计算密集型（computer-bound），后者称为 I/O 密集型（I/O-bound）。

何时调度

需要处理各种情形：

1. 在创建一个新进程后，需要决定是运行父进程还是运行子进程。
2. 在一个进程退出时必须做出调度决策。
3. 当一个进程阻塞在 I/O 和信号量上或者由于其他原因阻塞时，必须选择另一个进程运行。
4. 在一个 I/O 中断发生时，必须做出调度决策。

非抢占式调度算法：挑选一个进程，然后让该进程运行直至被阻塞，或者直到进程自动释放 CPU。

抢占式调度算法：选择一个进程，让该进程运行某个固定时段的最大值。
如果时段结束进程任在运行则被挂起，调度程序选择另外一个进程运行。

调度算法分类

三种环境

1. 批处理。非抢占式，或长时间周期的抢占式算法
2. 交互式。抢占式算法
3. 实时。抢占有时是不需要的。

调度算法的目标

- 所有系统
    - 公平——给每个进程公平的 CPU 份额
    - 策略强制执行——看到锁宣布的策略执行
    - 平衡——保持系统的所有部分都忙碌
- 批处理系统
    - 吞吐量——每小时最大作业数
    - 周转时间——从提交到终止间的最小时间
    - CPU 利用率——保持 CPU 时钟忙碌
- 交互式系统
    - 响应时间——快速响应请求
    - 均衡性——满足用户的期望
- 实时系统
    - 满足截止时间——避免丢失数据
    - 可预测性——在多媒体系统中避免品质降低

批处理系统中的调度

- 先来先服务
- 最短作业优先
- 最短剩余时间优先

交互式系统中的调度

- 轮转调度
- 优先级调度
- 多级队列
- 最短进程优先
- 保证调度
- 彩票调度
- 公平分享调度

实时系统中的调度

实时系统的调度算法可以是静态的或动态的。
前者在系统开始运行之前做出调度决策；后者在运行过程中进程调度决策。

3、存储管理

现代操作系统使用分层存储器体系（memory hierarchy）。操作系统中管理分层存储器体系的部分称为存储管理器（memory manager），即记录那些内存是正在使用的，那些内存是空闲的；在进程需要时为其分配内存，在进程是用完后释放内存。

无存储抽象

最简单的存储器抽象就是根本没有抽象。早期计算机没有存储器抽象，每个程序都直接访问物理内存。想要在内存中同时运行两个程序是很困难的。

一种存储器抽象：地址空间

暴露物理内存给进程的问题：

1. 如果用户程序可以寻址内存中的每个字节，它能很容易破坏操作系统。
2. 使用这种模型想同时运行多个城市是很困难的。

地址空间的概念

保证多个应用程序同时处于内存中并且不互相影响，需要解决两个问题：保护和重定位。

地址空间为程序创造了一种抽象的内存。每个进程都有自己的地址空间，并且这个地址空间独立于其他进程的地址空间。

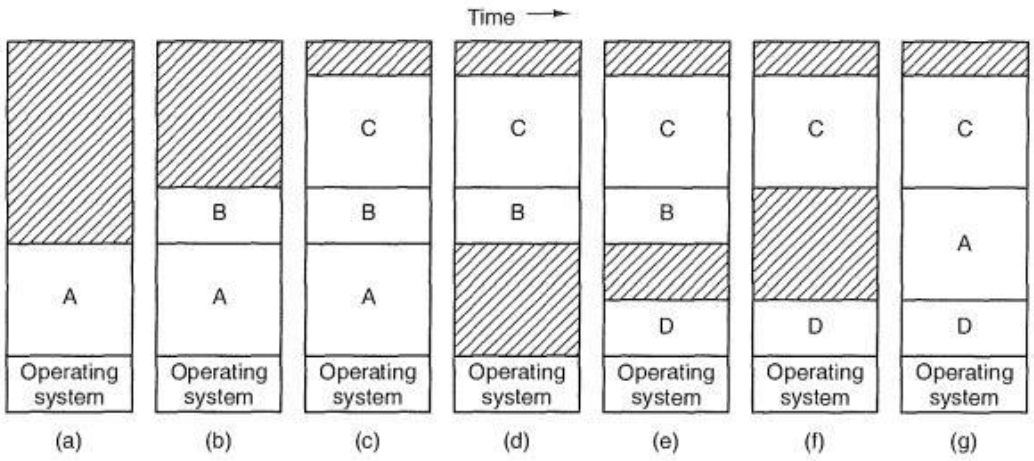
基址寄存器与界限寄存器

动态重定位，简单地把每个进程的地址空间映射到物理内存的不同部分。程序装在到内存期间无须重定位，程序的起始物理地址装在到基址寄存器中，程序的长度装在到界限寄存器中。

缺点：每次访问内存都需要进行加法和比较运行。

交换技术

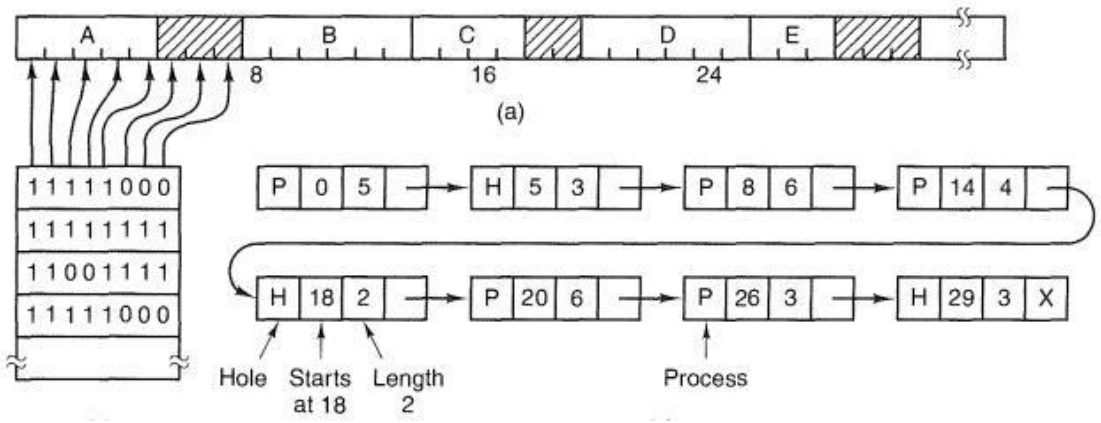
两种处理内存超载的方法：交换（swapping）技术，虚拟内存（virtual memory）。



如果大部分进程在运行时都要增长，可以在换入或移动进程时为它多分配一些额外的内存，例如数据段与堆栈段。

空闲内存管理

两种跟踪内存使用情况的方法：位图，空闲链表。



查找位图中指定长度的连续 0 串是耗时操作，这是位图的缺点。

链表管理法适配方式：

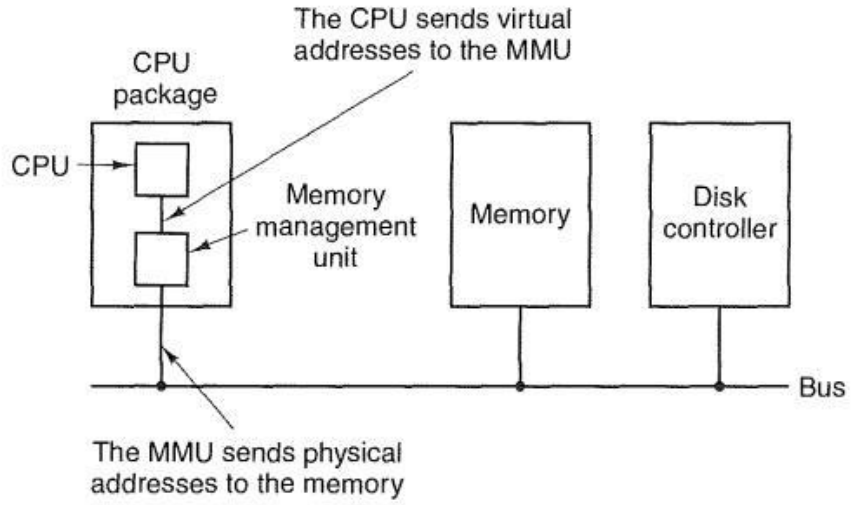
- 首次适配（first fit）：存储管理器沿着链表进行搜索，直到找到一个足够大的空闲区。
- 下次适配（next fit）：每次找到合适的空闲区都记录，以便下次从此开始搜索。
- 最佳适配（best fit）：搜索整个表，找出能够容纳进程的最小空闲区。
- 最差适配（worst fit）：总是分配最大的可用空闲区，使新的空闲区比较大。
- 快速适配（quick fit）：为那些常用大小的空闲区维护单独的链表。

虚拟内存

基本思想：每个程序拥有自己的地址空间，这个空间被分割成多个块，每一块称作一页或页面（page）。每一页有连续的地址范围。这些页被映射到物理内存，但并不是多有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间使，由硬件立刻执行必要的映射。放程序引用到一部分不再物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令。

分页

由程序产生的地址称为虚拟地址（virtual address），它们构成了一个虚拟地址空间（virtual address space）。在没有虚拟内存的计算机上，系统直接将虚拟地址送到内存总线上，读写操作使用具有同样地址的物理内存字；而在使用虚拟内存的情况下，虚拟地址不是被直接送到内存总线上，而是被送到内存管理单元（Memory Management Unit，MMU），MMU 把虚拟地址映射为物理内存地址。



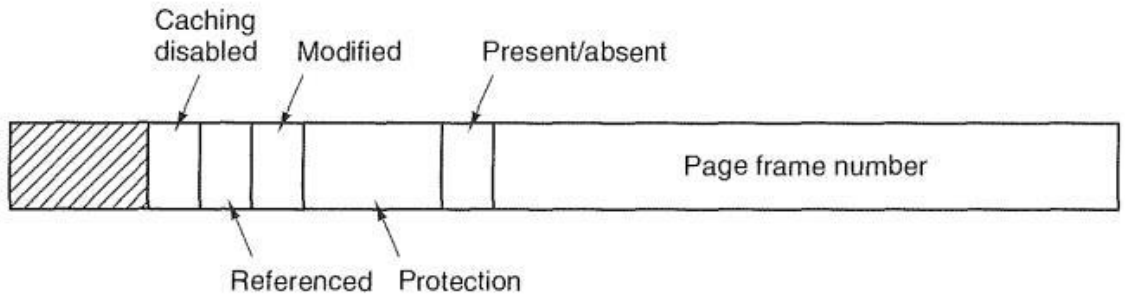
虚拟地址空间按照固定大小划分成页面（page）的若干单元。在物理内存中对应的单元称为页框（page frame）。页面和页框的大小通常是一样的。

当 MMU 注意到某页面没有被映射，于是使 CPU 陷入到操作系统，这个陷进称为缺页中断（page fault）。操作系统找到一个很少使用的页框且把它的内容写入磁盘。随后把需要访问的页面读到刚才回收的页框中，修改映射关系，然后重新启动引起陷进的指令。

页表

简单的，虚拟地址被分成虚拟页号（高位部分）和偏移量（低位部分）。虚拟页号可用作页表的索引，以找到该虚拟页面对应的页表项。由页表项可以找到页框号。然后把页框号拼接到偏移量的高位端，以替换掉虚拟页号，形成送往内存的物理地址。

页表项的结构



- 页框号
- “在/不在”位：对应的虚拟页面在不在内存中
- 保护位：三位，表示读写执行
- 修改位
- 访问位
- 高速缓存禁止位

加速分页过程

在任何分页式系统中，都需要考虑两个主要问题：

1. 虚拟地址到物理地址的映射必须非常快。
2. 如果虚拟地址空间很大，页表也会很大。

解决方式：

- 转换检测缓冲区（Translation Lookaside Buffer，TLB）。大多数程序总是对少量的页面进行多次访问。
- 软件 TLB 管理

针对大内存的页表

- 多级页表：避免把全部页表一直保存在内存中。
- 倒排页表：在实际内存中每一个页框有一个表项，而不是每一个虚拟页面有一个表项。

页面置换算法

当发生缺页中断时，操作系统必须在内存中选择一个页面将其换出内存。如果要换出的页面在内存驻留期间已被修改，就必须写回磁盘；如果没有修改直接被覆盖就可以了。

- 最优页面置换算法：置换最久将要被访问的页面。此算法无法实现。
- 最近未使用页面置换算法（Not Recently Used，NRU）：随机地从类编号最小的非空类中挑选一个页面淘汰之。

    1. 没有被访问，没有被修改

2. 没有被访问，已被修改
3. 已被访问，没有被修改
4. 已被访问，已被修改

- 先进先出页面置换算法（First-In，First-Out，FIFO）：每次置换链表的表头。
- 第二次机会页面置换算法：检查最老页面的访问位。如果是 0 则立刻置换，如果是 1 则修改为 0 并把该页面放到链表尾端。
- 时钟页面置换算法：环形链表，有一个表针指向最老页面。首先检查表针，如果 R 位是 0 就淘汰页面，插入页面后表针前移一个位置。如果 R 位是 1 就清除 R 并把表针前移一个位置，直到找到 R 为 0 的页面。
- 最近最少使用页面置换算法（Least Recently Used，LRU）：置换未使用时间最长的页面。
- 工作集页面置换算法：进程当前正在使用的页面的集合称为它的工作集（working set）。缺页中断时淘汰一个不再工作集中的页面。
- 工作集时钟页面置换算法

页面置换算法小结

算法	注释
最优算法	不可实现，但可用作基准
NRU（最近未使用）算法	LRU 的很粗糙的近似
FIFO（先进先出）算法	可能抛弃重要的页面
第二次机会算法	比 FIFO 有大的改善
时钟算法	现实的
LRU（最近最少使用）算法	很优秀，但很难实现
NFU（最不常用）算法	LRU 的相对粗略的近似
老化算法	非常近似 LRU 的有效算法
工作集算法	实现起来开销很大
工作集时钟算法	好的有效算法

分页系统中的设计问题

- 局部分配策略与全局分配策略：当发生缺页中断时，页面置换算法在寻找最近最少使用的页面时，只考虑分配给该进程的页面，称为局部页面置换算法；考虑所有在内存中的页面，称为全局页面置换算法。
- 负载控制

- 页面大小：确定最佳页面大小。
- 分离的指令空间和数据空间：为指令和数据设置分离的地址空间
- 共享页面：UNIX 系统中进行 `fork` 系统调用后，父子进程共享程序文本和数据。这些进程拥有自己的页表，但都指向同一个页面集合。只要有一个进程要求更新数据，就会触发只读保护，引发操作系统陷进，生成该页的一个副本。这种方式称为写时复制。
- 共享库：如果其他程序已经装载了某个共享库，另一个程序就没必要再次装载了。编译共享库时，使用-fPIC 产生位置无关代码。
- 内存映射文件：进程可以通过发起一个系统调用，将一个文件映射到虚拟地址空间的一部分。在访问页面时每次一页的读入。多个进程可以通过映射同一个文件来通信。
- 清除策略：如果发生缺页中断时系统中有大量的空闲页框，此时分页系统工作在最佳状态。一种实现清除策略的方法就是使用一个双指针时钟。
- 虚拟内存接口

有关实现的问题

- 与分页有关的工作：操纵系统要在四段时间里做与分页相关的工作。进程创建时，进程执行时，缺页中断时，进程终止时。
- 缺页中断处理

    1. 硬件陷入内核，在堆栈中保存程序计数器。
    2. 启动一个汇编代码例程保存通用寄存器和其他易失的信息，以免被操作系统破坏。
    3. 当操作系统发现一个缺页中断时，尝试发现需要哪个虚拟页面。
    4. 一旦知道发生缺页中断的虚拟地址，操作系统检查这个地址是否有效，并检查存取与保护是否一致。
    5. 如果选择的页框“脏”了，安排该页写回磁盘，并发生一次上下文切换，挂起产生缺页中断的进程，让其他进程运行直至磁盘传输结束。
    6. 一旦页框“干净”后，操作系统查找所需页面在磁盘上的地址，通过磁盘操作将其装入。
    7. 当磁盘中断发生时，表明该页已经被装入，页表已经更新可以反应它的位置，页框也被标记为正常状态。
    8. 恢复发生缺页中断指令以前的状态，程序计数器重新指向这条指令。
    9. 调度引发缺页中断的进程，操作系统返回调用它的汇编语言例程。
    10. 该例程恢复寄存器和其他状态信息，返回到用户空间继续执行，就好像缺页中断没有发生过一样。
- 指令备份：使用一个隐藏的内部寄存器。在每条指令执行之前，把程序计数器的内容复制到该寄存器中。
- 锁定内存中的页面：保证正在做 I/O 操作的页面不会被移出内存。
- 后备存储
- 策略和机制的分离

分段

分页和分段的比较

考察点	分页	分段
需要程序员了解正在使用这种技术吗？	否	是
存在多少线性地址空间？	1	许多
整个地址空间可以超出物理存储器的大小吗？	是	是
过程和数据可以内区分并分别被保护吗？	否	是
其大小浮动的表可以很容易提供吗？	否	是
用户间过程的共享方便吗？	否	是
为什么发明这种技术？	为了得到大的线性地址空间而不必购买更大的物理存储器	为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护

4、文件系统

文件

- 文件命名：在具体系统中规则不一，UNIX 文件系统区分大小写，MS-DOS 文件系统不区分大小写。
- 文件结构：字节序列、记录序列、树。所有 UNIX、MS-DO、Windows 系统都把文件看成无结构字节序列。
- 文件类型：ASCII 文件和二进制文件。
- 文件存取：顺序存取和随机存取。
- 文件属性：文件的附加信息，称为属性（attribute）或元数据（matedata）。

- 文件操作：常用系统调用

    - `create`：创建不包含任何数据的文件。
    - `detele`：删除文件释放磁盘空间。
    - `open`：打开文件，把文件属性和磁盘地址表装入内存，便于后续调用。
    - `close`：关闭文件释放内部表空间。
    - `read`：在文件中读取数据。
    - `write`：向文件中写数据。
    - `append`：在文件末尾添加数据。
    - `seek`：对于随机存取文件，指定从何处开始取数据。
    - `get attribute`：读取文件属性。
    - `set attribute`：某些属性可由用户设置。
    - `rename`：改变已有文件的名字。

目录

- 一级目录系统：一个目录中包含所有文件，称为根目录。能快速定位文件。
- 层次目录系统：层次化的目录树结构，几乎所有现代文件系统都采用。
- 路径名：绝对路径与相对路径。
- 目录操作：常用基本系统调用。

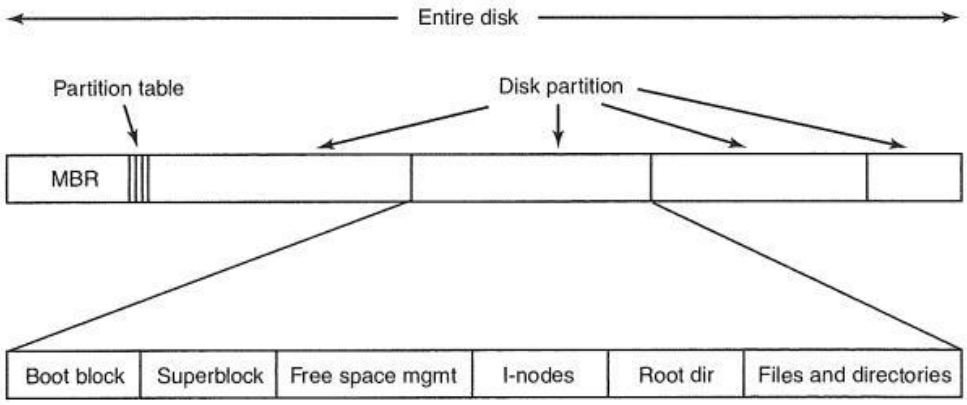
    - `create`：创建目录，处理目录项.和..外，目录内容为空。
    - `delete`：删除目录。
    - `opendir`：打开目录以备读取。
    - `closedir`：关闭目录释放内部表空间。
    - `readdir`：读取目录返回目录下一个目录项。
    - `rename`：更改目录名称。
    - `link`：连接技术允许在多个目录中出现同一个文件。
    - `unlink`：删除目录项。

文件系统的实现

文件系统布局

文件系统存放在磁盘上。多数磁盘划分为一个或多个分区，每个分区中有一个独立的文件系统。磁盘的 0 号扇区称为主引导记录（Master Boot Record，MBR），用来引导计算机。在 MBR 的结尾是分区表。该表给出了每个分区的起始地址和结束地址。表中的一个分区被标记为活动分区。在计算几被引导时，BIOS 读入并执行 MBR。MBR 做的第一件事是确定活动分区，读入它的第一个块，称为引导块（boot block），并执行之。引导块中的程序将装在该分区中的操作系统。

一种可能的文件系统布局：



- 超级块（superblock）：包含文件系统的所有关键参数。
- 空闲空间管理：空闲块的信息，可以用位图或指针表形式给出。
- i 节点：数据结构数组，每个文件一个，说明文件的方方面面。
- 根目录：目录树的根部。
- 目录和文件：存放其他所有目录和文件

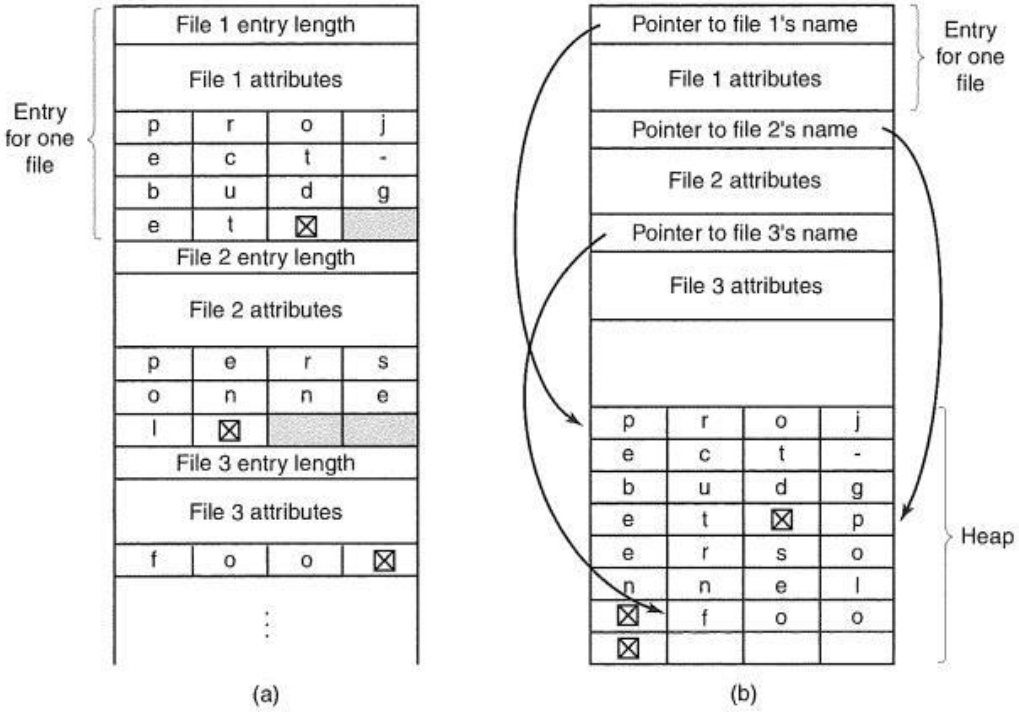
文件的实现

- 连续分配：把每个文件作为一连串连续数据块存储在磁盘上。易产生零碎空间。
- 链表分配：为每个文件构造磁盘块链表。随机存取缓慢。指针占去了一些空间。
- 在内存中采用表的链表分配：取出每个磁盘块的指针字，把它放在内存的一个表中。称为文件分配表（File Allocation Table，FAT）。对于大磁盘而言占用内存太多。
- i 节点：给每个文件赋予一个称为 i 节点（index-node）的数据结构，其中列出了文件属性和文件块的磁盘地址。

目录的实现

1. 目录中有一个固定大小的目录项列表，每个文件对应一项，其中包含一个文件名、一个文件属性结构以及泳衣说明磁盘块位置的一个或多个磁盘地址。
2. 采用 i 节点的系统，把文件属性存放在 i 节点中，目录项更短：只有文件名和 i 节点号。

两种处理目录项中长文件名的方式：



共享文件

1. 磁盘块不列入目录中，而是列入一个与问价本身关联的小型数据结构（i 节点中）中。目录将指向这个小型数据结构。
2. 建立一个类型为 link 的新文件，使其与另一个文件存在连接。新文件中指包含了它所连接的文件的路径名。称为符号连接（symbolic linking）。

日志文件系统

基本思想是将整个磁盘结构化为一个日志。每隔一段时间，或是有特殊需要时，被缓冲在内存中的所有未决的血操作都被放到一个单独的段中，作为在日志末尾的一个邻接段写入磁盘。

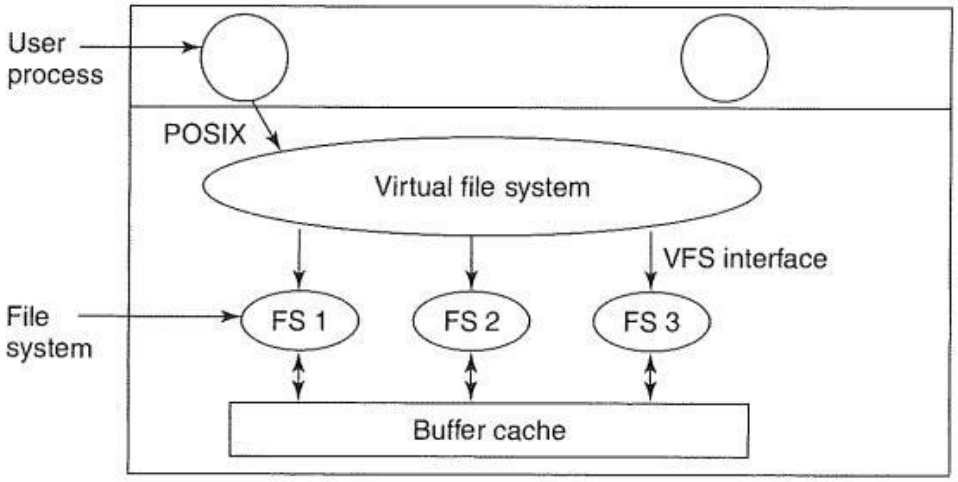
保存一个用于记录系统下一步将要做什么的日志。这样当系统在完成它们即将完成的任务钱崩溃时，重新启动之后，可以通过查看日志，获取崩溃前计划完成的任务，并完成它们。这样的文件系统称为日志文件系统。

为了增加可信性，一个文件系统可以引入数据库中原子事务（atomic transaction）的概念。

虚拟文件系统

大多数 UNIX 操作系统都使用虚拟文件系统概念尝试将多种文件系统统一成一个有序的框架。
关键思想解释抽象出所有文件系统的共有部分，并且将这部分代码放在单独的一层，该层调用底层的实际文件系统来具体管理数据。

虚拟文件系统的位置：



文件系统管理和优化

- 磁盘空间管理
    - 块大小
    - 记录空闲块
    - 磁盘配额
- 文件系统备份
- 文件系统的一致性
- 文件系统的性能
    - 高速缓存
    - 块提前读
    - 减少磁盘臂运动
- 磁盘碎片整理

5、输入输出

I/O 硬件原理

I/O 设备大致分为两类：

- 块设备（blocl device）：所有传输以一个或多个完整的（连续的）块为单位。
- 字符设备（character device）：以字符为单位发送或接受一个字符流。

I/O 设备一般由机械部件和电子部件两部分组成，电子部件称作设备控制器（device controller）或适配器（adapter）。

内存映射 I/O 的优点：

1. 对于内存映射 I/O，设备控制寄存器只是内存中的变量，在 C 语言中可以和任何其他变量一样寻址。
2. 对于内存映射 I/O，不需要特殊的保护机制来阻止用户进程执行 I/O 操作。
3. 对于内存映射 I/O，可以引用内存的每一条指令也可以引用控制寄存器。

I/O 软件原理

在设计 I/O 软件时一个关键的概念是设备独立性（device independence）。

I/O 软件的几个问题：

- 统一命名：不应该依赖于设备。
- 错误处理：尽可能在接近硬件的层面得到处理。
- 同步和异步传输。
- 缓冲：数据离开设备之后不能直接存放到最终目的地。
- 共享设备的独占问题。

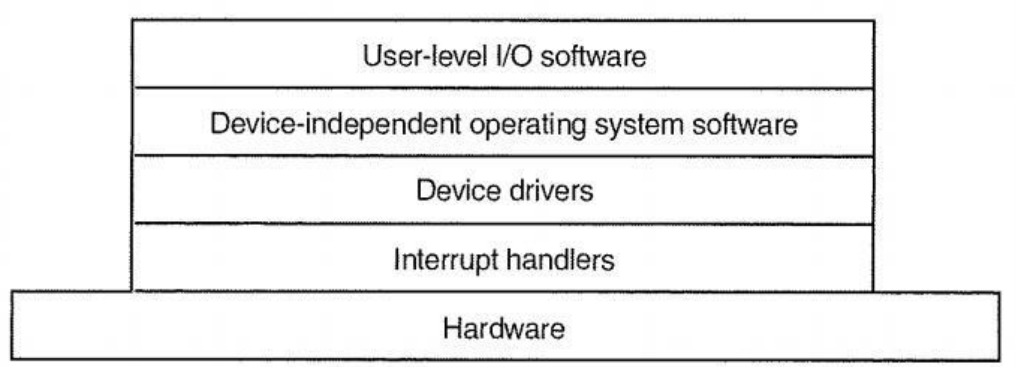
I/O 的最简单形式是让 CPU 做全部工作，这一方法称为程序控制 I/O。

使 CPU 在等待打印机变为就绪的同时做某些其他事情的方式就是使用中断驱动 I/O。缺点是中断发生在每个字符上，浪费时间。

由 DMA 控制器处理全部的 I/O 工作，使 CPU 可以在 I/O 期间做其他工作。

I/O 软件层次

I/O 软件通常组织成四个层次：



- 中断处理程序

- 设备驱动程序
- 与设备无关的 I/O 软件
- 用户空间的 I/O 软件

盘

- 磁盘
- RAID
- CD-ROM
- 可刻录 CD
- 可重写 CD
- DVD

影响磁盘块读写速度的因素

1. 寻道时间（将磁盘臂移动到适当的柱面上所需的时间）
2. 旋转时间（等待适当扇区旋转到磁头下所需的时间）
3. 实际数据传输时间。

时钟

- 时钟硬件：晶体振荡器、计数器和存储寄存器。
- 时钟软件
- 软定时器

用户界面：键盘、鼠标和显示器

- 输入软件
    - 键盘软件
    - 鼠标软件
- 输出软件
    - 文本窗口
    - X 窗口系统
    - 图形用户界面
    - 位图
    - 字体

6、死锁

资源

所有需要排他性使用的对象都可以称作资源（resource）。资源可以分为两类：

- 可抢占资源。可以从拥有它的进程中抢占而不会产生任何副作用，例如存储器。
- 不可抢占资源。在不引起相关的计算失败的情况下，无法把它从占有它的进程处抢占过来。

死锁概述

死锁的规范定义：如果一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么，该进程集合就是死锁的。

资源死锁的 4 个条件：

1. 互斥条件。
2. 占有和等待条件。
3. 不可抢占条件。
4. 环路等待条件。

四种处理死锁的策略：

1. 忽略该问题。
2. 检测死锁并恢复。
3. 仔细对资源进行分配，动态的避免死锁。
4. 通过破坏引起死锁的四个必要条件之一，防止死锁的产生。

鸵鸟算法

鸵鸟算法：把头埋到沙子里，假装根本没有问题发生。

死锁检测和死锁恢复

- 每种类型一个资源的死锁检测。检测是否存在有向图环路。
- 每种类型多个资源的死锁检测。基于矩阵的算法。
- 从死锁中恢复
    - 利用抢占恢复
    - 利用回滚恢复
    - 通过杀死进程恢复

死锁避免

- 资源轨迹图
- 安全状态和不安全状态
- 单个资源的银行家算法
- 多个资源的银行家算法

死锁预防

- 破环互斥条件
- 破坏占有和等待条件
- 破坏不可抢占条件
- 破坏环路等待条件

其他问题

- 两阶段加锁
- 通信死锁
- 活锁
- 饥饿

7、多媒体操作系统

多媒体简介

多媒体的两个关键特征：

1. 多媒体使用极高的数据率。
2. 多媒体要求实时回放。

多媒体文件

视频编码

基于人眼的特性：当一幅图像闪现在视网膜上时，在它衰退之前将保持几毫秒时间。如果一个图像序列以每秒 50 或更多张图像闪现，眼界就不会注意到它看到的是不连续的图像。

为了将二维图像表示作为时间函数的一维电压，摄像机用一个电子束对图像进行横向扫描并缓慢地向下移动，记录下电子束经过处光的强度。在扫描的终点处，电子束折回。称为一帧（frame）。

彩色视频采用与单色视频相同的扫描模式，只不过使用了三个同时运动的电子束来显示图像。红、绿、蓝（RGB）三原色。

数字视频最简单的表示方法是帧的序列，每一帧由呈矩形栅格的图像要素即像素（pixel）组成。每个像素 RGB 三色中的每种颜色用 8 个二进制位来表示。

音频编码

音频波可以通过模数转换器（Analog Digital Converter，ADC）转换成数字形式。ADC 以电压作为输入，并且生成二进制数作为输出。

由于每一样本的位数有限而引入的误差称为量化噪声（quantizatuin）。

电话系统使用的实时脉冲编码调制（pulse code modulation），脉冲编码调制以每秒 7 位或 8 位对声音采样 8000 次，故这一系统的数据率为 56000bps 或 64000bps。由于每秒只有 8000 个样本，所以 4kHz 以上的频率就丢失了。

音频 CD 是以每秒 44100 个样本的采样率进行数字化的，足以捕获高达 22050Hz 的频率。

视频压缩

所有的压缩系统都需要两个算法：一个用于在源端进行数据压缩，另一个用于在目的端对数据进行解压缩。分别为编码算法和解码算法。

编码与解码具有某些不对称性。视频信号经过编码和解码之后与原始信号存在轻微差异时，系统被称为是有损的（lossy）。

JPEG 标准

用于压缩连续色调静止图像的 JPEG（Joint Photographic Experts Group，联合摄影专家组）。用于压缩运动图像的 MPEG 不过是分别对每一帧进行 JPEG 编码，再加上某些帧间压缩和运动补偿等额外的特征。

MPEG 标准

MPEG（Motion Picture Experts Group，运行图像专家组）标准是用于压缩视频的主要算法。

MPEG 的两个版本均利用了在电影中存在的两类冗余：空间冗余和时间冗余。MPEG-2 输出有三种不同的帧组成：

1. I 帧：自包含的 JPEG 编码静止图像
2. P 帧：与上一帧逐块的差。
3. B 帧：与上一帧和下一帧的差。

音频压缩

最流行的是 MP3（MPEG 音频层 3），属于 MPEG 视频压缩标准里的音频部分。

音频压缩可由两种方法完成。波形编码技术和感知编码。

多媒体进程调度

- 调度同质进程
- 一般实时调度
- 速率单调调度
- 最早最终时优先调度

多媒体文件系统范型

- VCR 控制功能
- 近似视频点播
- 具有 VCR 功能的近似视频点播

文件存放

多媒体文件非常庞大、通常只写一次而读许多次，并且倾向于被顺序访问。

- 在单个磁盘上存放文件
- 两个替代的文件组织策略
- 近似视频点播的文件存放
- 在单个磁盘上存放多个文件
- 在多个磁盘上存放文件

高速缓存

- 块高速缓存
- 文件高速缓存

多媒体磁盘调度

- 静态磁盘调度
- 动态磁盘调度

8、多处理机系统

多处理机

共享存储器多处理机（multiprocessor）是这样一种计算机系统，其两个或更多的 CPU 全部共享访问一个公用的 RAM。

- UMA（Uniform Memory Access，统一存储器访问）
- NUMA（Nonuniform Memory Access，非统一存储器访问）

多处理机硬件：

- 基于总线的 UMA 多处理机体系结构
- 使用交叉开关的 UMA 多处理机
- 使用多级交换的 UMA 多处理机
- NUMA 多处理机
- 多核芯片

多处理机操作系统类型：

- 每个 CPU 有自己的操作系统
- 主从多处理机
- 对称多处理机

多处理机调度

- 分时
- 空间共享
- 群调度

多计算机

多处理机硬件

- 互连技术
- 网络接口

底层通信软件

用户层通信软件

- 发送和接收
- 阻塞调用和非阻塞调用

分布式共享存储器

- 复制
- 伪共享
- 实现顺序一致性

负载平衡

- 图论确定算法
- 发送者发起的分布式启发算法
- 接收者发起的分布式启发算法

虚拟化

分布式系统

网络硬件

- 以太网
- 因特网

网络服务协议

- 网络服务
- 网络协议

基于文档的中间件

基于文件系统的中间件

- 传输模式
- 目录层次
- 命名透明性
- 文件共享的语义

基于对象的中间件

基于协作的中间件

- Linda
- 发布/订阅
- Jini

网络

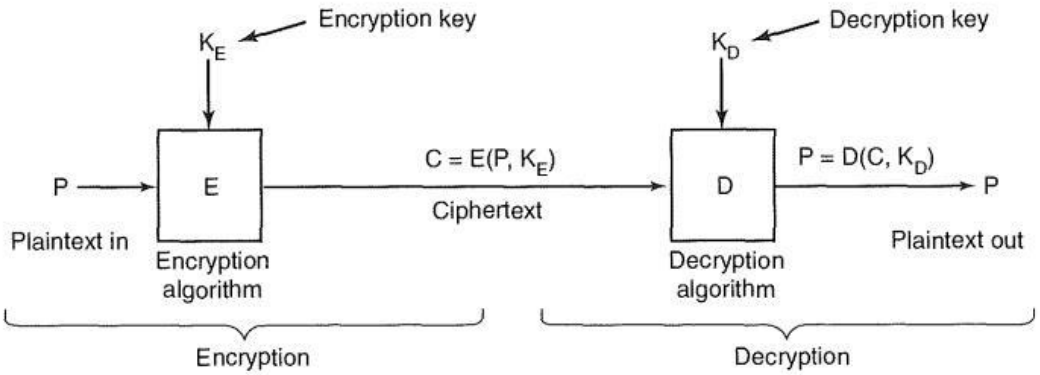
9、安全

环境安全

- 威胁
    - 计算机系统主要安全目标
        1. 数据机密性
        2. 数据完整性
        3. 数据可用性
        4. 排外性
    - 计算机系统主要安全威胁
        1. 数据暴露
        2. 数据篡改
        3. 拒绝服务
        4. 系统被病毒控制
- 入侵者
    - 入侵者表现形式
        - 被动入侵
        - 主动入侵
    - 入侵者种类
        0. 非专业用户的随意浏览
        1. 内部人员的窥视
        2. 为获取利益而尝试
        3. 商业或军事间谍
- 数据意外遗失
    - 天灾
    - 软硬件错误
    - 认为过失

密码学原理

明文与密文之间的关系：



- 私钥加密技术。对称加密，安全性取决于对密钥的管理
- 公钥加密技术。每个人都有公钥和私钥，公开其中的公钥，用公钥加密，只能用对应私钥解密。
- 单向函数。散列函数
- 数字签名
- 可信平台模块（Trusted Platform Modules，TPM）。是一种加密处理器，使用内部的非易失性存储介质来保存密钥。

保护机制

- 保护域
- 访问控制表
- 权能
- 可信系统
- 可信计算基
- 安全系统的形式化模型
- 多级安全
- 隐蔽信道

认证

- 使用口令认证
- 使用实际物体的认证方式
- 使用生物识别的验证方式

内部攻击

- 逻辑炸弹
- 后门陷阱
- 登录欺骗

利用代码漏洞

- 缓冲区溢出攻击
- 格式化字符串攻击
- 返回 libc 攻击
- 整数溢出攻击
- 代码注入攻击
- 权限提升攻击

恶意软件

- 特洛伊木马
- 病毒
- 蠕虫
- 间谍软件
- rootkit

防御

- 防火墙
- 反病毒和抑制病毒技术
- 代码签名
- 囚禁
- 基于模型的入侵检测
- 封装移动代码
    - Java 安全性