操作系统概念

编 写
袁郭苑 鲍 伟
林子童 徐卫霞
孙吉鹏 张晓敏

特别鸣谢
韩芳溪老师

吉鹏智库-让知识回归平凡

目录-*带 *的内容为各位作者认为该章节中比较重要的内容*

第一部分 概述

作为十几年中式教育的佼佼者，你会敏锐地发现考试内容只是平时课堂内容的子集，平日里学习的大部分内容都不会出现在最后的那份试卷里。你可以选择在最后对那个子集进行突击，平日去做自己内心想做的事；也可以选择把这门知识当作一种经历，成为你以后的阅历。无可厚非，每一种选择都很有价值。

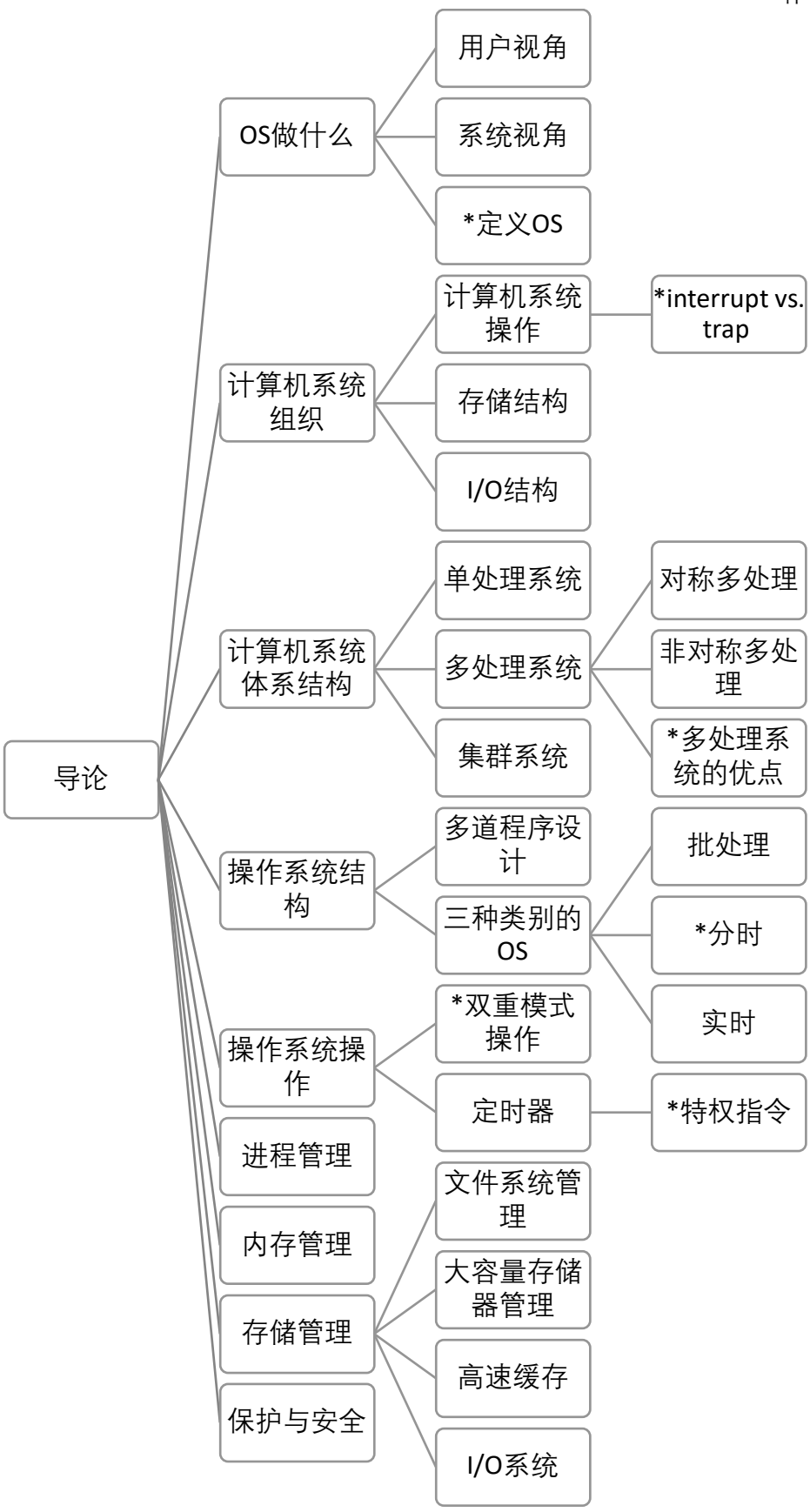
智库想去做的，是让每个人都能方便找到对自己的选择有用的内容。所以既不会像课本那样繁杂，也不会像考纲那样无趣，满足你的求知欲也不会遗漏那个子集。

所以，那就让我们开始？

这四章大都是涉及到概念原理内容，定义题居多，计算内容较少，但都是后面几章计算的基础，重点在于理解归纳。

第一章 导论

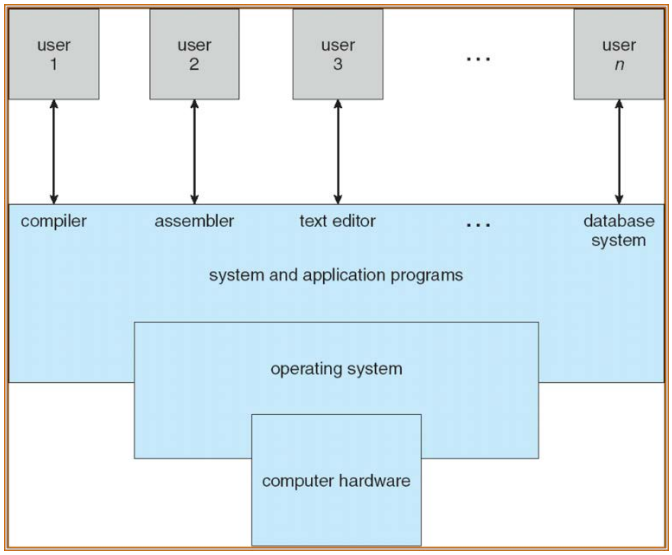
作者：袁郭苑


导论
OS做什么
用户视角
系统视角
*定义OS
计算机系统组织
计算机系统操作
*interrupt vs. trap
存储结构
I/O结构
计算机系统体系结构
单处理系统
多处理系统
对称多处理
非对称多处理
*多处理系统的优点
集群系统
操作系统结构
多道程序设计
三种类别的OS
批处理
*分时
实时
操作系统操作
*双重模式操作
定时器
*特权指令
进程管理
内存管理
存储管理
文件系统管理
大容量存储器管理
高速缓存
I/O系统
保护与安全

1.1 操作系统的功能

1.1.1 计算机系统的组成部分？

硬件，操作系统，系统程序和应用程序，用户



1.1.2 两个视角

①用户视角---OS 设计目的是为了用户使用方便，性能是相对次要的

②系统视角---OS 是资源分配器&&OS 是控制程序

1.1.3 *操作系统的定义？（对下管理，对上服务）

操作系统是一组控制和管理计算机硬件和软件资源、合理地对各类作业进行调度，以及方便用户的程序集合。操作系统是用户和计算机的接口，同时也是计算机硬件和其他软件的接口。

1.2 计算机系统的组织（机组知识的回顾）

1.2.1 *中断

生活中的小例子，上自习时家里突然来电话，你需要“中断”手头的作业去接电话，之后回来继续从当时写的地方写下去，这就是一个“中断”处理过程。

换成 CPU 的话需要如何处理中断呢？

(1) 保护断点：即保存下一将要执行的指令的地址，就是把这个地址送入堆栈。

(2) 寻找中断入口：根据不同的中断源所产生的中断向量，查找不同的入口地址，入口地址处存放着中断处理程序。

(3) 执行中断处理程序。

(4) 中断返回：执行完中断指令后，就从中断处返回到主程序，继续执行。

1.2.2 陷阱

陷阱是一种**软件**产生的中断，源于程序**出错**（如除 0 或访问内存无效）或者源于用户程序的**特别请求**（系统调用 system call），完成中断处理后将 CPU 控制权再交给提出陷阱请求的程序。

1.2.3 *中断（interrupt）与陷阱（trap）的区别

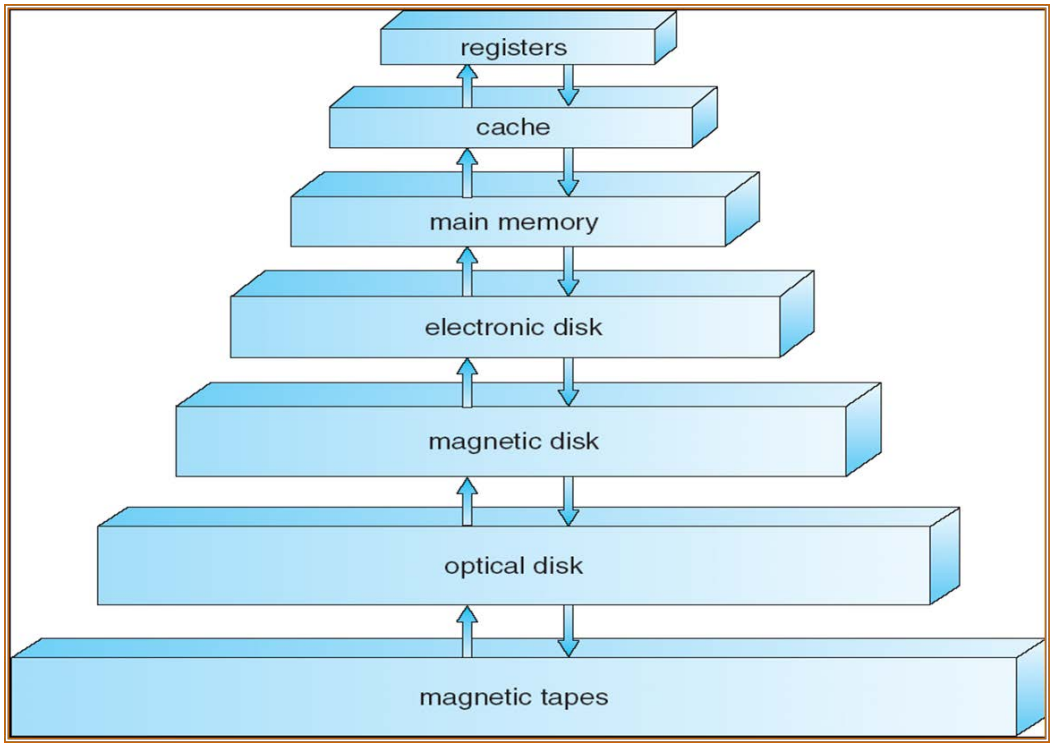
陷阱又被称作软中断，与（硬）中断相比，软中断是*软件*实现的中断，也就是程序运行时其它程序对它的中断；而硬中断是*硬件*实现的中断，是程序运行时设备对它的中断。（硬中断是由外部事件引起的，因此具有随机性和突发性；而软中断的发生不是随机的，而是由程序安排好的）

1.2.4 计算机启动和固件的概念

计算机启动过程一般是指计算机从加电到加载操作系统的过程。

计算机打开电源或重启时，要运行一个引导程序（bootstrap program），其常位于 ROM 或 EEPROM 中，称为计算机硬件中的固件。

1.2.5 存储结构（不妨回忆机组相关知识）



电子磁盘（electronic disk）之上为易失存储（volatile storage），之下为非易失存储（nonvolatile storage），电子磁盘本身可以被设计为易失或非易失的

1.2.6 I/O 结构（不妨回忆机组相关知识）

通用计算机系统由一个或多个 CPU 和若干设备控制器组成，它们通过共同的总线连接起来。

操作系统为每个设备控制器提供一个设备驱动程序。

DMA（direct memory access）方式---每一个块产生一个中断；内存与设备之间直接进行，无需 CPU 干预。

1.3 计算机系统体系结构

1.3.1 *多处理器系统中对称多处理和非对称多处理的区别

非对称多处理（AMP）处理器间是主从关系，一个主处理器控制系统并向其他从处理器分配任务，主处理器单独做 IO 任务

对称多处理（SMP）处理器间是平等的关系，IO 可以被任一处理器处理。

1.3.2 *多处理器系统的优点

(1) 增加吞吐量

(2) 规模经济

(3) 增加可靠性

1.4 操作系统结构

1.4.1 多道程序（multiprogramming）的概念

操作系统同时把多个任务保存在内存中，如果一个执行中的任务需要等待一个事件的完成则 CPU 切换到另一个任务并执行而不是空等待原任务完成。

1.4.2 三种主要类别的操作系统

(1) 批处理系统：用户将作业交给系统操作员，系统操作员将许多用户的作业组成一批作业(jobs)之后输入到计算机中，在系统中形成一个自动转接的连续的作业流，系统自动、依次执行每个作业。最后由操作员将作业结果交给用户。

优点：作业流程自动化；效率高；吞吐量高

缺点：无交互手段；调试程序困难

(2) *分时系统：操作系统将 CPU 的时间划分成若干个片段,称为**时间片**。操作系统以时间片为单位，在用户间快速切换，轮流为每个终端用户服务，每次服务一个时间片。系统的快速切换使用户感到整个系统只为自己所用。是多道程序设计的延伸，由于切换频率很高，用户可以在程序运行期间与之进行交互

(3) 实时系统：指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统作出快速响应，并控制所有实时任务协调一致运行的操作系统。

1.5 操作系统的操作

1.5.1 *双重模式操作（Dual-Mode Operation）

组成：用户模式（user mode）和核心模式（kernel mode），即用户模式下应用程序需要操作系统的服务时必须切换至核心模式由操作系统完成相应请求。

动机：将能引起机器损害的机器指令作为*特权指令（privileged instruction），如转换到用户模式，IO 控制，定时器管理和中断管理等，通过识别**模式位**保证特权指令只能由操作系统完成，保护操作系统和用户程序不受错误用户程序影响。

1.5.2 定时器

确保 OS 对 CPU 的控制，同时防止用户程序陷入死循环等情况不将控制权返回 OS，使用定时器在给定时间后中断。

1.6 例题

1.6.1

说明对 CPU，I/O，memory 的保护的目的及其保护措施。

答：CPU—确保操作系统维持对其的控制，防止用户程序陷入死循环或不调用系统服务，并且不将控制权返回到操作系统。

对此，使用定时器，将其设置为在给定时间后中断计算机。用于修改定时器操作的指令时特权指令。

I / O – 防止用户直接发出 I/O 指令，并且防止用户非法使用 I/O 设备。

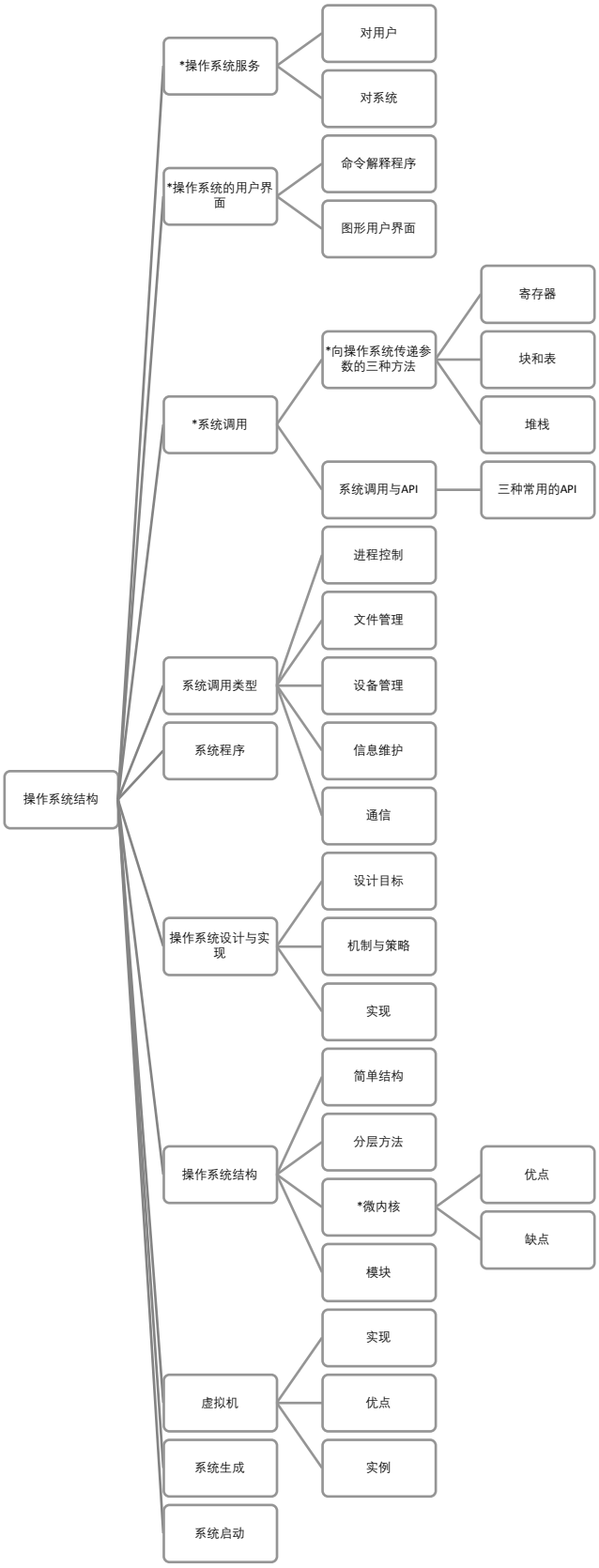
保护措施可以将 I/O 指令设置为特权指令。这样就可以防止在用户模式下执行 I/O 指令，从而实现 I/O 保护。

memory-防止用户非法地访问和修改内存其他位置的信息。

保护措施是将修改基址寄存器（base register）和长度寄存器（limit register）的指令设置为特权指令。

第二章 操作系统结构

作者：袁郭苑



2.1 *操作系统服务

2.1.1 对用户

用户界面（命令行界面 CLI&批界面&图形用户界面 GUI），程序执行，I/O 操作，文件系统操作，通信，错误检测

2.1.2 对系统

资源分配，统计，保护和安全

2.2 *操作系统的用户界面

2.2.1 命令解释程序

执行命令的两种常用方法：①命令解释程序本身包含代码以执行命令②由系统程序实现绝大多数命令（UNIX）

2.2.2 图形用户界面（占空间多一些）

2.3 *系统调用

2.3.1 *系统调用（system call）

操作系统内核提供一系列预定功能，通过一组称为系统调用的接口呈现给编程人员，系统调用把应用程序的请求传给内核，系统调用相应的内核函数完成所需的处理，将处理结果返回给应用程序。C 或 C++编写。**系统调用是内核的一部分。**

2.3.2 如何找到相应的内核函数？

实际上每一个系统调用都和一个**数**相关联，通过索引表找到相应的内核函数，类似于中断向量表，系统调用本质上也是一种软中断（trap）。

2.3.3 三种常用的 API（应用程序接口 Application Program Interface）

(1) 适用于 Windows 系统的 Win32API
(2) 适用于 POSIX 系统的 POSIXAPI（UNIX，Linux，MacOSX）
(3) 运行于 Java 虚拟机程序的 JavaAPI

2.3.4 使用 API 而不使用系统调用的原因

如果说系统调用是内核和应用程序信息交流的通道，那么 API 就是程序和开发人员之间交流的通道，它为开发人员掩盖系统调用的细节，提供访问并操作硬件的权利。

原因：API 的可移植性强；系统调用操作难度较大（细节和困难）

2.3.5 *向操作系统传递参数的三种方法

(1) 通过寄存器来传递参数
(2) 将参数存在内存的块和表中，将块的地址通过寄存器来传递
(3) 参数通过程序压入堆栈，通过操作系统弹出

2.4 系统调用类型

系统调用大致分为五大类：进程控制，文件管理，设备管理，信息维护和通信

2.5 系统程序

(1) 系统程序提供了一个方便的环境，以开发程序和执行程序

(2) 一小部分系统程序只是系统调用的简单接口，其他的可能是相当复杂的。分为如下几类：文件管理，状态信息，文件修改，程序语言支持，程序执行和装入，通信。

2.6 操作系统设计及和实现

2.6.1 策略与机制

机制决定如何做，策略决定做什么。

2.7 操作系统结构

2.7.1 四种结构

简单结构（Simple Structure）：早期的操作系统并没有很好的架构，有些甚至没有区分核心态和用户态。导致系统的不稳定性。

分层方法（Layered approach）：将操作系统分为多个层次，最底层的是硬件（层0），最高层为用户接口（层N），每一层都只是依靠于**更底层**的功能来实现的，这样形成一个层次化的结构。层次结构的好处是简化了构造和调试，弊端是层次之间耦合性高，层次之间很难界定。另一个问题就是层次结构的效率较低，一个系统调用可能要陷入多个层再返回，增加了很多花销。

微内核（Microkernels）（下面展开讲）

模块（Modules）：采用面向对象的特点，将各个功能模块化，每一个模块之间使用**接口**进行通讯，必要的时候可以将一部分的内容加载到内核中进行操作。它类似于层次结构，但是模块之间没有上下层的依赖关系，模块之间可以相互调用，更加灵活；它类似于微核，但是必要时候会将内容加载到内核中，比微核更有效率。

许多操作系统采用混合结构。

2.7.2 *微内核

将所有非基本部分从内核中移走，并将它们实现为系统程序或用户程序，从而得到更小的内核。微内核通常包括最小的*进程和内存管理*以及*通信*功能。

用户程序和系统服务（运行在用户空间）通过微内核以**消息传递**形式进行通信，并不会直接交互。

2.7.3 *微内核的优点和缺点

优点：

(1)便于扩充操作系统

(2)操作系统容易移植

(3)更好的安全性和可靠性

缺点：

由于系统功能总开销的增加而导致系统性能的下降（各个服务之间缺少通信，只能依靠微核的信息传递，导致效率下降。）

2.8 虚拟机

2.8.1 核心思想

虚拟机的核心思想就是将一套硬件设备抽象成多套。采用 CPU 调度和虚拟内存的技术，制造每一个进程都有自己单独的处理器和内存的假象。

2.8.2 虚拟机的优点

对于操作系统设计员，可以在虚拟机上进行系统开发，而不必进行中断，停止当前操作系统；对于用户来说，每个虚拟机完全独立于其他虚拟机，所以不同系统资源有完全的保护，没有安全问题。

2.8.3 虚拟机资源共享的两种实现方法

(1) 可以通过共享**小型磁盘**来共享文件（模拟共享物理磁盘，但通过软件实现）
(2) 可以通过定义一个**虚拟机的网络**（模拟物理通信网络，但通过软件实现）

2.8.4 虚拟机实例

JVM；Vmware；VirtualBox；Cygwin（基于 Windows 的 Unix 模拟环境）

2.9 系统启动

2.9.1 生成操作系统之后，它必须要为硬件所使用。

绝大多数计算机系统都有一小块代码，称为引导程序或引导装载程序，这段代码能定位操作系统内核，将它装入内存，开始执行。

2.10 例题

2.10.1

Operating System Design and Implementation

答：对于操作系统的设计和实现时所遇到的一些问题，虽然没有完整的解决方案，但是有些方法还是很成功的。

系统设计的第一个问题是定义系统的目标和规格（设计目标）。一个重要的原理是策略（policy）和机制（mechanism）的区分，机制决定如何做，策略决定做什么；策略和机制的区分对于灵活性来说很重要。传统的操作系统是用汇编语言来编写的，不过现在操作系统都是用高级语言如 **C** 或 **C++**来编写的。与其他系统一样，操作系统的重要性能改善很可能是由于更好的**数据结构**和**算法**，而不是由于优秀的汇编语言代码。

2.10.2

操作文件和设备时，采用同样的系统调用界面有什么优点和缺点？

答：

优点：采用同样的系统调用界面，可以使用户的程序代码用相同的方式被写入设备和文件，利于用户程序的开发。还利于设备驱动程序代码，可以支持规范定义的API。

缺点：系统调用为所需要的服务提供最小的系统接口来实现所需要的功能，由于设备和文件读写速度不同，若是同一接口的话可能会处理不过来。

2.10.3

命令解释器的用途是什么？为什么它经常是与内核分开的？是否可能采用操作系统提供的系统调用接口为用户开发一个新的命令解释器？

答：用途是从**用户**或**命令文件**读入命令并执行它，通常将其变成一个或多个系统调用它们。

它通常不是内核的一部分，因为命令解释是会**改变**的，不是固定的。

可以。因为所有功能都可能被用户程序通过系统调用来实现。

2.10.4

模块化内核方法和分层方法在哪些方面类似？哪些方面不同？

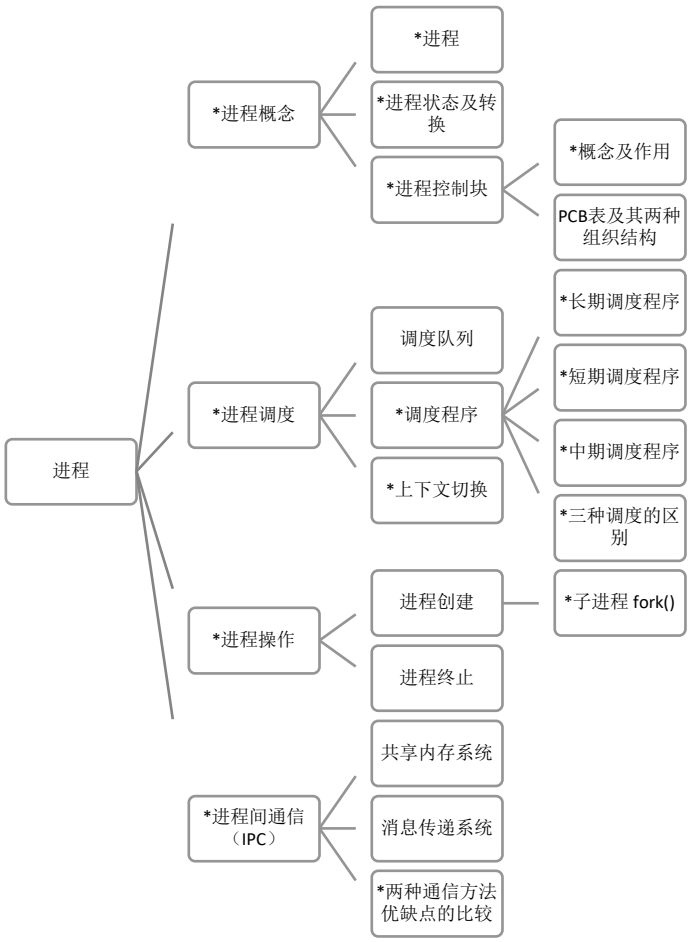
答：模块化内核方法要求子系统通过创建的一般而言狭隘的接口来相互作用。分层内核方法在细节上与分层方法相似。但是，分层内核必须要是有严格排序的子系统，这样的子系统在较低层次中不允许援引业务相应的上层子系统。在模块化内核方法中没有太多的限制，模式是随意援引彼此的，没有任何约束。

第二部分 进程管理

进程是操作系统中一个重要的概念，进程中包含若干个线程，它们相互协作完成任务。当用户运行多个程序时，就是创建了多个进程，它们等待着 CPU 的调度，不同的调度算法各有优劣。当多进程并发，并且存在抢占的时候，那些共享的资源就可能会出现数据的不一致，这个时候线程的同步问题就显得尤为重要。线程的同步问题可能导致系统死锁，如何预防线程的死锁，解决死锁带来的问题也成了操作系统的一个重要课题。

第三章 进程

作者：袁郭苑


进程
*进程概念
*进程
*进程状态及转换
*进程控制块
*概念及作用
PCB表及其两种组织结构
*进程调度
调度队列
*调度程序
*上下文切换
*长期调度程序
*短期调度程序
*中期调度程序
*三种调度的区别
*进程操作
进程创建
*子进程 fork()
进程终止
*进程间通信（IPC）
共享内存系统
消息传递系统
*两种通信方法优缺点的比较

3.1 *进程概念

3.1.1 *进程

关于进程定义的描述有很多，从不同的角度可以给出不同的定义，关键是理解这些描述。简单地讲，进程是执行中的应用程序。

进程不只是程序代码，还包括当前活动。包括*文本段*（或代码段），*数据段*（包括全局变量），*堆*（运行期间动态分配的内存），*栈*（临时数据，如函数参数，返回地址和局部变量），还有*程序计数器*和*处理器寄存器*的支持。

3.1.2 进程与程序

程序本身不是进程，程序只是被动实体，而进程是活动实体（当一个可执行文件被装入内存时，一个程序才能成为进程）

“一个程序不能两次属于同一个进程”——结合着进程的特征来理解这句话。

PS：进程的基本特征。

(1) 并发性：多个进程实体同存于内存中，能在一段时间内同时执行

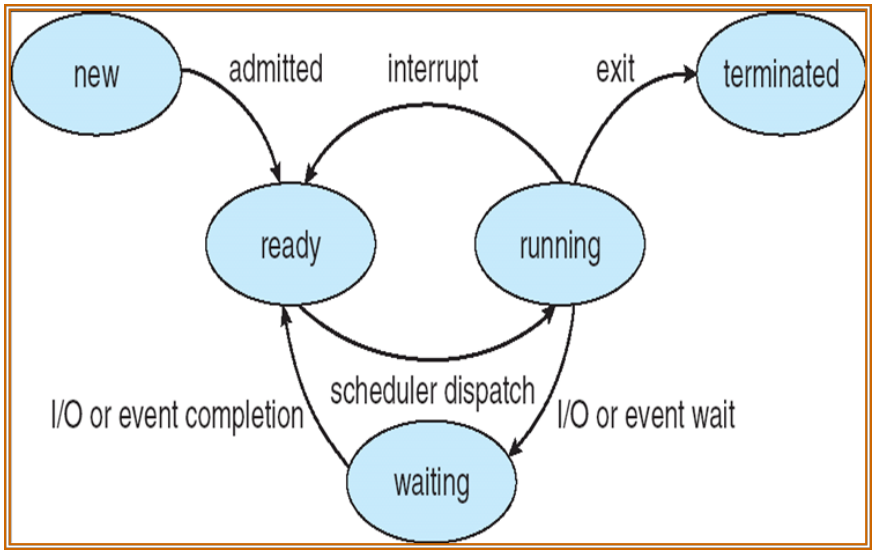
(2) 动态性：进程由创建而产生，由调度而执行，因得不到资源而暂停执行，由撤销而消亡。

(3) 独立性：进程实体是一个能独立运行的基本单位，同时也是系统中独立获得资源和独立调度的基本单位。

(4) 异步性：进程按各自独立的、不可预知的速度向前推进。

(5) 结构特征：从结构上看，进程实体由程序段、数据段以及进程控制块组成。

3.1.3 *进程状态及转换



五个状态：新的（new），就绪（ready），运行（running），等待（waiting），终止（terminated）

*状态间转换条件

一次只有一个进程可在一个处理器上运行，但是多个进程可处于就绪或等待状态

3.1.4 *进程控制块（PCB）

PCB 是系统为了管理进程设置的一个专门的数据结构，用它来记录进程的**外部特征**，描述进程的**运动变化过程**。

3.1.5 PCB 的作用

系统利用 PCB 来**控制**和**管理**进程

PCB 是进程存在的唯一标志

操作系统通过 PCB 而感知进程的存在

3.1.6 PCB 包含的信息

进程状态；程序计数器；CPU 寄存器；CPU 调度信息；内存管理信息；记账信息；I/O 状态信息等

3.1.7 PCB 表（决定了系统的并发度）

系统把所有 PCB 组织在一起，并把它们放在内存的固定区域，就构成了 PCB 表。

PCB 表的两种组织结构：链接结构（就绪链表，运行链表…），索引结构（不同状态的进程设置各自的 PCB 索引表，表明在 PCB 表中的地址）

3.1.8 并发与并行

这是以后经常遇见的两个概念，希望大家可以区分清楚。

并发：在某一时间段内任务交替执行。

并行：多个任务同时执行。

3.2 *进程调度

3.2.1 调度队列

三种较重要的队列：

①作业队列（Job queue）：系统中的所有进程

②就绪队列（Ready queue）：驻留在内存中，就绪的等待运行的进程

③设备队列（Device queue）：等待 I/O 设备的进程

3.2.2 *长期调度程序（作业调度程序）（to Ready queue）

从缓冲池中选择进程，并装入内存以准备执行

特点：调度频率较低；可以控制系统多道程序度；平衡 I/O 设备与 CPU 的利用率（选择合理的包含 *I/O 为主*和 *CPU 为主*的进程的组合）

3.2.3 *短期调度程序（CPU 调度程序）（to Running queue）

从准备执行的进程中选择进程，并为之分配 CPU

特点：调度频繁；调度快慢影响 CPU 利用率和系统吞吐量

3.2.4 *中期调度程序（swapper）

将进程从内存（或从 CPU 竞争）中移出，从而降低多道程序设计的程度；之后，进程能被重新调入内存，并从**中断处**继续执行。

3.2.5 *三种调度程序的区别

长期调度程序又称为作业调度程序，从缓冲池中选择进程，并装入内存以准备执行。长期调度程序执行得并不频繁，控制多道程序设计的程度，还能平衡 IO 与 CPU 的利用率。

短期调度程序又称为 CPU 调度程序，从准备执行的进程中选择进程，并为之分配 CPU。短期调度程序要频繁地为 CPU 选择新进程，所以必须要快，它影响着 CPU 的利用率和与系统吞吐量。

中期调度程序的核心思想是能将进程从内存（或从 CPU 竞争）中移出，从而降低多道程序设计的程度。进程可换出，并在后来可被换入，一种交换（swapping）的方案。

3.2.6 *上下文切换

将 CPU 切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态，这一任务称为上下文切换。

CPU 在进程间的切换由中断或系统调用引起，需要保存状态至原进程的 PCB，然后再从新进程的 PCB 中获取状态。

上下文切换时间与硬件支持密切相关（多组寄存器集合 vs 单组寄存器集合）

3.3 *进程操作

3.3.1 进程创建与进程树

父进程创建子进程。从 root 进程开始，操作系统中的进程像树一样延伸下去。

3.3.2 *进程创建---系统调用 fork()

大多数操作系统根据一个唯一的**进程标识符**（process identifier，pid）来识别进程，pid 通常是一个整数值。

理解进程创建的几个重要概念，这一项也是本章的重点！

Fork()的两个要点：

①内核为子进程做一个父进程的上下文拷贝。

拷贝包括：（1）复制父进程的 PCB 作为子进程的 PCB（2）在新的地址空间中复制父进程的一个拷贝

②父进程与子进程在不同的地址空间上运行。

关于①的理解：子进程与父进程共享子进程创建前的所有资源。

关于②的理解：子进程在创建后和父进程是竞争关系，两个进程依据进程调度的规则分别执行。

总结：**先继承，后分离**。

Fork()的返回值：

```
Pid=fork();
```

正确执行：父进程返回子进程号；子进程返回 0。

出现错误：返回-1。

这一部分除了理解概念之外，还需要多看例子，加深理解。在看例题时可以画出进程树，尤其是创建多进程问题，有助于分析问题。

3.3.3 进程结束时的问题

有了父进程与子进程的概念后，我们来讨论一下进程结束的问题。正常情况下，子进程结束后会有父进程回收子进程的资源。但是有些时候在子进程结束前，父进程已经结束或是没有 wait()语句等待子进程结束，这时子进程就不可能被父进程处理了，需要操作系统出面解决问题。不同的系统有不同的处理方式，有的系统**不允许**子进程在没有父进程的情况下继续执行，而 Linux 和 UNIX 会交由 1 号进程（init）作为父进程回收这类子进程。

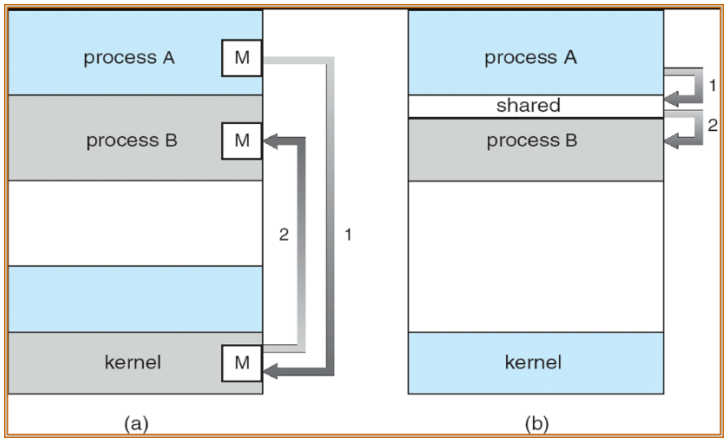
3.4 进程间通信（IPC）

3.4.1 进程协作（Cooperating Processes）的优点

(1) 信息共享
(2) 提高运算速度（多个处理单元&子任务并行执行）
(3) 模块化
(4) 方便

3.4.2 *进程间通信的两种基本模式

(1) 消息传递（a）
(2) 共享内存（b）



3.4.3 *两种通信方法优缺点的比较

消息传递适用于交换较少数量的数据；易于实现计算机间的通信；但是速度慢（由于通过系统调用实现，需要内核介入）

共享内存速度快（仅在建立共享内存区域时需要系统调用）

3.5 例题

3.5.1

描述内核在两个进程间进行上下文切换的过程

答：将 CPU 切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态，这一任务称为上下文切换（进程上下文用进程的 PCB 表示）。状态保存＆状态恢复

当发生上下文切换时，内核会将旧进程的状态保存在其 PCB 中，然后装入经调度要执行的并已保存的新进程的上下文。

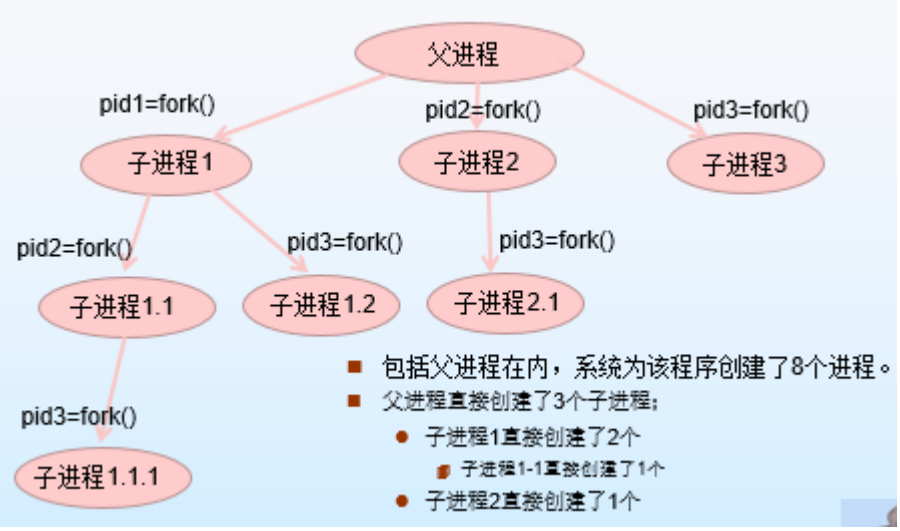
3.5.2

包括最初的父进程，这段代码一共创建了多少进程？

```
#include <stdio.h>
#include <unistd.h>

int main()
{
      fork(); /* fork a child process */
      fork(); /* fork another child process */
      fork(); /* and fork another */
      return 0;
}
```

解答：



子进程被创建出来之后，将与父进程一起接着往下执行

3.5.3

解释概念 PCB

答：系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程，包含与一个特定进程相关的信息：进程状态、程序计数器、cpu 寄存器、cpu 调度信息、内存管理信息、记账信息和 IO 状态信息。系统利用 PCB 来控制和管理进程，所以 PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的。

3.5.4

试比较进程和程序的区别

答：进程和程序是既有联系又有区别的两个概念，它们的主要区别如下：

（1）进程是程序在处理机上的一次执行过程，是一个动态概念；而程序是代码的有序集合，其本身没有任何运行的含义，是一个静态的概念。

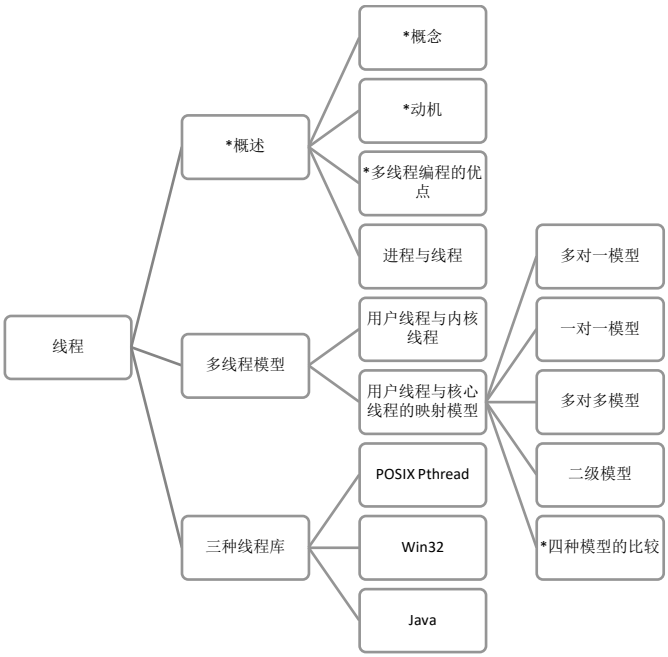
（2）进程是一个状态变化的过程，是有生命期的，表现在它因创建而产生，因调度而执行，因得不到资源而暂停，因撤销而消亡；而程序是永久的，可以长久保存。

（3）进程和程序的组成不同。进程由程序、数据和进程控制块组成，而程序仅是代码的有序集合。

（4）进程与程序之间不是一一对于的。通过多次运行，同一个程序可以对应多个进程过调用关系，一个进程可以包含多个程序。

第四章 线程

作者：袁郭苑


线程
*概述
*概念
*动机
*多线程编程的优点
进程与线程
多线程模型
用户线程与内核线程
用户线程与核心线程的映射模型
多对一模型
一对一模型
多对多模型
二级模型
*四种模型的比较
三种线程库
POSIX Pthread
Win32
Java

4.1 *概述

4.1.1 *概念

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

4.1.2 进程的两个基本特征

（1）可**拥有资源**的独立单位

（2）可**独立调度和分派**的基本单位

4.1.3 *动机

OS 中进程的引入是为了使多个程序并发执行，改善资源利用率，提高系统的吞吐量；OS 中引入线程是为了减少程序并发执行时所付出的时空开销，使 OS 具有更好的并发性。

将进程的两个基本特征分开：作为调度和分派的基本单位，不同时作为独立分配资源的单位，使之轻装上阵；作为拥有资源的基本单位，不频繁对之进行切换。这种观点导致了线程的产生---是进程的一个实体，是被系统独立调度和分派的基本单位，自己基本不拥有资源，只拥有一点在运行中必不可少的资源（程序计数器、寄存器、栈），可与同属于一个进程的其他线程共享进程所拥有的资源。

4.1.4 *多线程编程的优点

（1）响应度高：对一个交互程序采用多线程，即使其部分阻塞或执行较冗长的操作，该程序仍能继续执行，从而增加了对用户的响应程度

（2）资源共享：线程默认共享它们所属进程的内存和资源

（3）经济：创建和切换线程比较经济

（4）多处理器体系结构的利用

4.1.5 进程与线程

传统的 OS 中，调度和分派的基本单位是进程，拥有资源的基本单位也是进程；引入线程的 OS 中，**调度和分派的基本单位是线程，拥有资源的基本单位是进程**；线程能轻装上阵，可显著地提高系统的并发度；同一进程中，线程的切换不会引起进程切换；不同进程中的线程之间的切换要引起进程的切换

引入线程的系统中，同一进程中的多个线程之间可并发执行，使系统具有更好的并发性，进一步提高了资源的利用率及系统的吞吐量

同一进程中的多个线程具有相同的地址空间，致使他们之间的同步和通信的实现，也变得比较容易

4.2 多线程模型

4.2.1 用户线程（user-level threads）与内核线程（kernel-level threads）

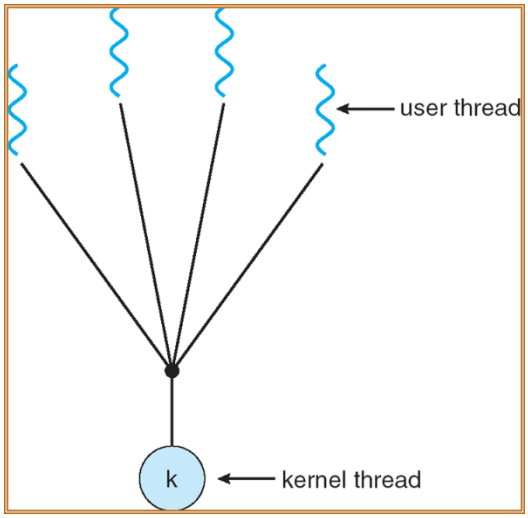
用户级线程仅存在于用户空间中。对于这种线程的创建、 撤消、切换、线程之间的同步与通信等功能，都无须利用系统调用来实现，而是通过**用户级线程库**来实现。**用户线程受内核支持，而无需内核管理**。

内核线程由操作系统直接支持和管理。

4.2.2 用户线程与核心线程的映射模型

当用户级线程执行时，要将线程映射到核心线程才能执行

(1) 多对一模型

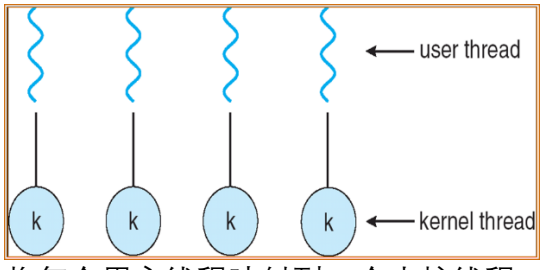


将许多用户线程映射到一个内核线程。用户线程管理由线程库在用户空间进行，故效率比较高。

缺点：但是如果一个线程执行阻塞了系统调用，那么整个进程会阻塞。而且多个线程不能并行运行在多处理器上（任一时刻只有一个线程能访问内核）

常用于不支持内核线程的系统中

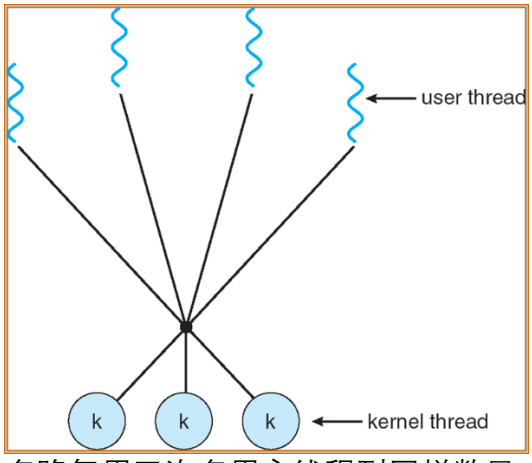
(2) 一对一模型



将每个用户线程映射到一个内核线程。提供了比多对一模型更好的并发功能（在一个线程执行阻塞了系统调用时，能允许另一个线程继续执行）。允许多个线程并行地运行在多处理器系统上。

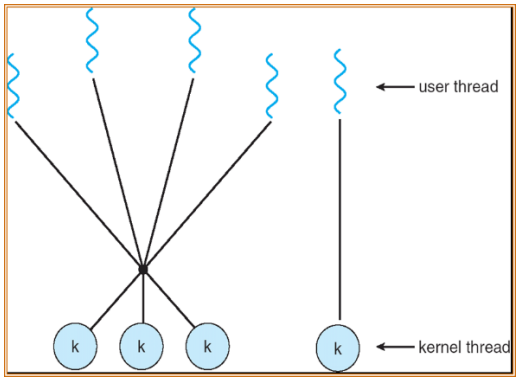
缺点：会限制系统所支持的线程数量（一对一地创建内核线程的开销会影响应用程序的性能）

(3) 多对多模型



多路复用了许多用户线程到同样数量或更小数量的内核线程上。开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。

(4) 二级模型



在多对多模型的基础上也允许将一个用户线程绑定到某个内核线程上

4.2.3 *四种模型的比较

多对一模型将许多用户级线程映射到一个内核线程。线程管理是由线程库在用户空间进行的，因而效率比较高。但是如果一个线程执行阻塞了系统调用，那么整个进程会阻塞。因为任意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。**（可创建任意多用户线程，不并发，不并行）**

一对一模型将每个用户线程映射到一个内核线程。它提供了比多对一模型更好的并发性，也允许多个线程能并行地运行在多处理器系统上。**（限制线程数量，并发，并行）**

多对多模型多路复用了许多用户线程到同样数量或更小数量的内核线程上。开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。而且，当一个线程执行阻塞系统调用时，内核能调度另一个线程来执行。**（可创建任意多用户线程，并发，并行）**

二级模型在多对多模型的基础上也允许将一个用户线程绑定到某个内核线程上。**（可创建任意多用户线程，并发，并行）**

4.3 线程库

4.3.1 实现线程库的两种方法

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

(2) 执行一个由操作系统直接支持的内核级的库（库的代码和数据结构存在于内核空间中），调用库中的一个 API 函数通常会导致对内核的系统调用。

4.3.2 三种线程库

(1) POSIX Pthread
(2) Win32 线程
(3) Java 线程

4.4 例题

4.4.1

描述线程库进行用户级线程上下文切换的过程所采取的措施

答：用户线程是通过线程库实现的。它们可以在没有内核参与下创建，释放和管理。线程库提供了同步和调度的方法。这样进程可以使用大量的线程而不消耗内核资源，而且省去大量的系统开销。用户线程的实现是可能的，因为用户线程的上下文可以在没有内核干预的情况下保存和恢复。每个用户线程都可以有自己的用户堆栈，用来保存用户级寄存器上下文以及如信号屏蔽等状态信息的内存区。库通过保存当前线程的**堆栈和寄存器内容**载入新调度线程的那些内容来实现用户线程的调度和上下文切换。

4.4.2

在多线程进程中，下列哪些程序状态组成被共享？

a. 寄存器值 b.堆内存 c.全局变量 d.栈内存

答：b,c

4.4.3

使用多用户线程的多线程解决方案，在多处理器系统中可以比在单处理器系统中获得更好的性能吗？

答：一个包括多用户线程的多线程系统无法在多处理系统上同时使用不同的处理器。操作系统只能看到一个单一的进程，不会调度在不同处理器上的不同进程的线程。因此可以得出结论，多处理器系统执行多个用户线程是没有性能优势的。

4.4.4

考虑多处理器系统和采用多对多线程模式编写的多线程程序，使程序中用户级线程数比系统中处理器数多，讨论下列情形的性能影响：

a.分配给程序的内核线程数比处理器数少
b.分配给程序的内核线程数与处理器数相等
c.分配给程序的内核线程数比处理器数多，但少于用户线程数

答：当内核线程数比处理器数量少时，会有一部分处理器处于闲置状态；当内核线程与处理器数量相等时，可能所有的处理器将同时使用；当内核线程比处理器数量多时，需要对处理器进行调度，将一部分内核线程调出，再换入另一些准备执行的内核线程，这样可以增加多处理器的利用率。

4.4.5

举两个多线程程序设计的例子，其中多线程的性能比单线程的差

答：

(1) 网络服务器处理，这时有 10 万条请求；使用单线程，只开一个线程处理这 10 万条请求；使用多线程，每个请求开一个线程，请求处理的时间极短，迅速返回，一次性提交 10 万个请求，则有 10 万个线程创建和销毁。这样的话，多线程性能比单线程要差，因为开很多个线程，每个线程只处理很少量的工作，创建线程的开销要远大于实际的工作成本

(2) 由于临界资源的存在，有的时候多线程抢占会耗费更多的时间，有的时候多线程可能会因为使用了临界资源变量造成时间相关的错误。

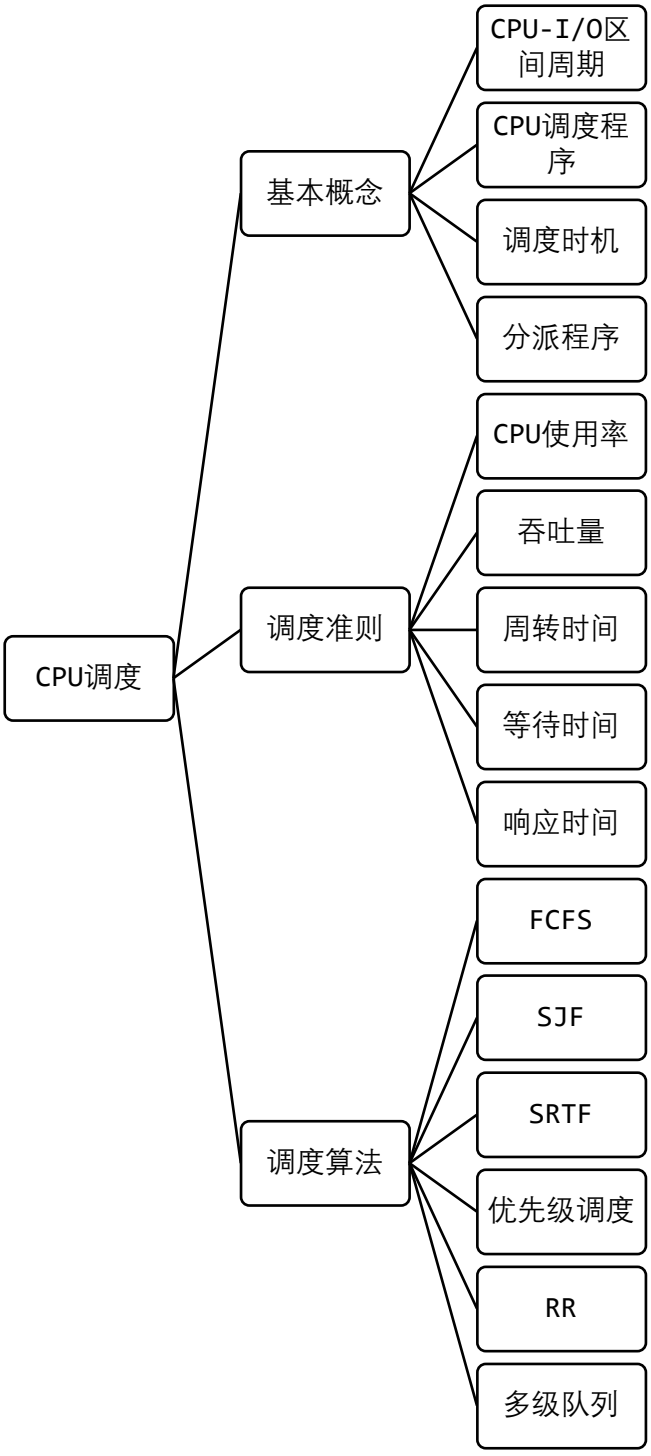
4.4.6

在什么环境中，采用多内核线程的多线程方法比单处理器系统的单线程提供更好的性能？

答：当一个内核线程发生页面错误时，系统会采用有用的方法切换到另一个进程去使用交错时间。而且当发生页面错误时，单线程进程不能有效地发挥作用。因此，在一个程序可能频繁地发生页面错误或者需要等待其他的系统事件的情况下，采用多线程的处理方法是要优于单处理器系统的。

第五章 CPU 调度

作者：鲍伟



5.1 基本概念

5.1.1 CPU-I/O 区间周期

CPU 成功调度依赖于进程执行的如下特性：进程执行由 CPU 执行和 I/O 等待周期组成，进程在这两个状态之间切换。最终，最后的 CPU 区间通过系统请求终止执行。

回顾：CPU 约束程序和 I/O 约束程序

CPU 约束程序（CPU-bound process）:I/O 请求并不频繁，花费更多的时间在计算上。

I/O 约束程序(I/O-bound process)：相比于计算，花费更多的时间在 I/O 上。

5.1.2 CPU 调度程序

CPU 调度程序（也成为短期调度程序）负责在 CPU 空闲的时候，从就绪队列中选择合适的进程分配给 CPU，就绪队列中等待的记录通常是进程控制块（PCB）。

5.1.3 *调度的时机和抢占的时机

调度时机可以发生在如下几种情况：

(1)当一个进程从运行状态切换到等待状态（例如 I/O 请求，或等待子进程结束）。

(2)当一个进程从运行状态切换到就绪状态（例如，当出现中断）

(3)当一个进程从等待状态切换到就绪状态（例如，I/O 完成）

(4)当一个进程结束

CPU 调度策略分为抢占（preemptive）和非抢占（nonpreemptive），在 CPU 调度的时机 2、3 时，可以发生抢占，不难看出就是有新进程添加进就绪队列的时候，可以发生抢占。抢占对于共享数据是有代价的，可能造成数据的不一致，这就涉及到第六章中讨论的同步问题。

5.1.4 分派程序（dispatcher）

分派程序是 CPU 调度的一个部分，它负责将 CPU 的控制交给由 CPU 调度程序选择的进程。其功能包括：

（1）切换上下文。

（2）切换到用户模式

（3）跳转到用户程序的合适位置，以重新启动程序

5.2 *调度的准则

5.2.1 准则如下：

（1）CPU 使用率：需要使 CPU 尽可能的忙碌。

（2）吞吐量（throughput）：它指一个时间单位内所完成的进程的数量。

（3）周转时间：从进程提交到进程完成的时间段称为周转时间。

（4）等待时间：等待时间为在就绪队列中等待所花费的时间的总和。

（5）响应时间：从提交请求到产生第一响应的时间。

5.2.2 关于准则的若干说明

需要使 CPU 使用率和系统的吞吐量最大化，周转时间、等待时间、响应时间最小化。

为了使吞吐量最大化，系统要平衡 CPU 约束程序和 I/O 约束程序，不能让 CPU 一直忙碌，或者一直空闲。

周转时间 = 等待时间 + 执行时间 = 结束时间 - 进入就绪队列时间。

5.3 调度算法

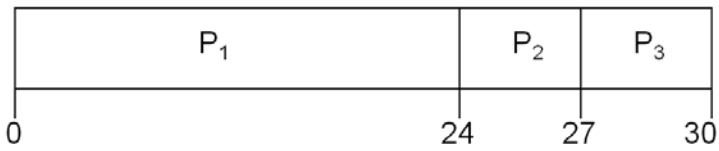
5.3.1 *先到先服务调度算法（first-come,first-served(FCFS)）

这个算法很容易理解，也很容易实现，但是平均等待时间通常会较长。下面我们结合甘特图说明一下。

先到先服务调度算法(FCFS)

进程	区间时间
P_1	24
P_2	3
P_3	3

- 假设进程到达的顺序是：P_1，P_2，P_3
  调度算法的甘特图如下：



- 等待时间：$P_1 = 0$；$P_2 = 24$；$P_3 = 27$
- 平均等待时间：$(0 + 24 + 27)/3 = 17$
- 周转时间：$P_1 = 24$；$P_2 = 27$；$P_3 = 30$

FCFS 的弊端从这个甘特图中也可以看出来，如果先调度 p2 和 p3，那么平均等待时间将大大减少，所以 FCFS 调度算法的平均等待时间和进程进入就绪队列的顺序有关，平均等待时间较长。而且，当执行一个大进程时，所有的进程都要等待大进程释放 CPU，这被称为护航效应。

5.3.2 *最短作业优先调度算法（shortest-job-first(SJF)）

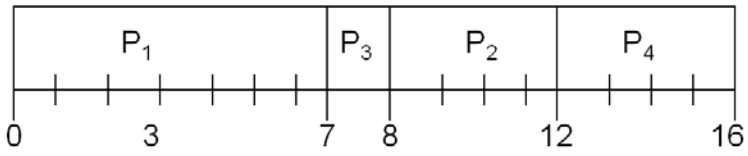
SJF 调度就是从就绪队列中调度最短 CPU 区间的的进程执行，如果有多个满足条件的进程则按照 FCFS 的策略调度。

同样使用甘特图说明一下：

最短作业优先调度算法（SJF）

进程	到达时间	区间时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- 非抢占的SJF算法甘特图如下



- 周转时间：$P_1 = 7$；$P_2 = 12-2$；$P_3 = 8-4$；$P_4 = 16-5$
- 等待时间：p1 = 0 ，p2 = 12-2-4 = 6，p3 = 8-4-1 = 3，p4 = 16 - 5 - 4 = 7。
- 平均等待时间：（0 + 6 + 3 + 7）/4 = 4

- Notes：调度时从就绪队列中选择符合条件的进程获得CPU执行权，所以虽然p3是CPU时间最少的进程，但是0.0时刻p3并不在就绪队列中，所以只能调度p1。

SJF 调度算法的平均等待时间是最小的，同时系统的吞吐量也增大了。有利于短进程，不利于长进程，长进程可能导致饥饿。但是 SJF 算法的困难在于很难知道下一个 CPU 区间的长度，所以它很难实现。SJF 调度经常用于长期调度。

5.3.3 *抢占的 SJF 调度算法—最短剩余时间优先调度（shortest-remianing-time-first(SRTF)）

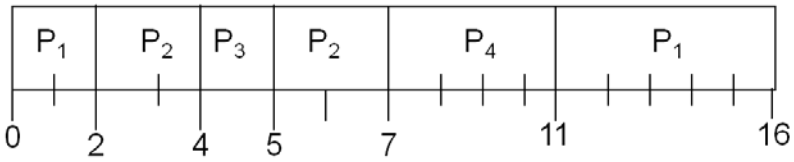
当前面介绍的 CPU 抢占的时机发生时，这个时候进程调度的时候就要考虑新添加进来的进程 pi 的 CPU 区间时间和当前正在执行的进程 pj 的剩余时间的大小，如果 pi<pj，则发生抢占。

甘特图说明如下：

最短剩余时间调度（SRTF）

进程	到达时间	区间时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- 抢占方式的SJF调度（SRTF）



- 周转时间 P1 = 16-0; P2 = 7-2; P3 = 5-4; P4 = 11-5
- 等待时间： P1 = 16-0-7; P2 = 7-2-4; P3 = 5-4-1; P4 = 11-5-4
- 平均等待时间： (9 + 1 + 0 +2)/4 = 3

实例说明：2.0 时刻，p2 进入就绪队列，满足抢占的条件，这个时候 p1 的剩余时间是 5，p2 的 CPU 区间时间是 4，所以 p2 抢占 p1。4.0 时刻 p3 进入就绪队列，满足抢占的条件，这个时候 p2 的剩余时间是 2，p3 的 CPU 区间时间是 1，所以 p3 抢占 p2。后面的分析如上。

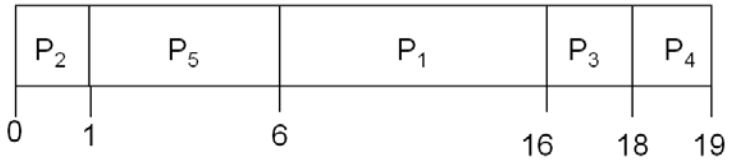
5.3.4 *优先级调度(priority)

每个进程都和一个优先级对应，高优先级的进程比低优先级的进程更早得到 CPU 的调度。具有相同优先级的进程，按照 FCFS 调度。我们可以将 SJF 调度看成一种简单的优先级调度，它的优先级同 CPU 区间时间成反比。（在教材和本复习材料中，我们都统一认为小数字表示高优先级）。

优先级调度算法

进程	区间时间	优先级
P_1	10	3
P_2	1	1
P_3	2	4
p4	1	5
p5	5	2

优先级调度算法的甘特图如下：



- 周转时间：P_1 = 16；P_2 = 1；P_3 = 18；P_4 = 19；P_5 = 6；
- 等待时间：P_1 = 16−10；P_2 = 1−1；P_3 = 18−2；P_4 = 19−1；

P_5 = 6−5；

- 平均等待时间：(6 + 0 + 16+ 18 + 1)/5 = 8.2

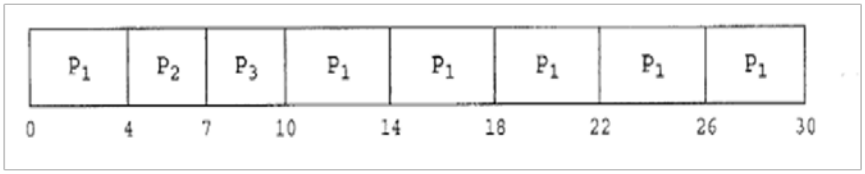
从甘特图中我们可以看到，p4 的 CPU 区间时间最短，但是优先级最低，所以最后被调度，这就造成了很长的等待时间。这也是优先级调度的一个问题—饥饿现象。所谓饥饿就是可以运行，但是缺乏 CPU，一直等待 CPU 调度。解决饥饿现象的一种机制是老化（aging），就是逐渐增加在系统中等待很长时间的进程的优先级。

5.3.5 *轮转法调度（RR）

轮转法类似于 FCFS，但是增加了抢占以切换进程。抢占的机制是定义一个较小的时间单元称为时间片，当分配给一个进程的时间片结束时，切换进程。如果进程在时间片内运行结束，那么多余的时间片回收，然后切换进程。

时间片为4的轮转法调度

Process	Burst Time
P_1	24
P_2	3
P_3	3



平均等待时间=[(30−24)+(7−3)+(10−3))]/3=17/3=5.66
周转时间：p1:30, p2=7，p3=10

实例说明：p1 执行完 4 个单位的时间片后，进程切换到 p2，p2 运行 3 个时间单位结束，剩余的时间片自动释放。p3 运行 3 个时间单位后结束，剩余时间片释放。

RR 算法的性能很大程度上取决于时间片的大小。RR 的调度算法在 FCFS 基础上改进的，改进了护航效应，但是仍然存在平均等待时间较长的缺点。

5.3.6 多级队列调度和多级反馈队列调度

多级队列调度算法将就绪队列分成多个独立队列，每个队列都有自己的调度算法，每个队列与更低队列相比都有绝对的优先级。

多级反馈队列调度允许进程在队列之间移动，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。这种形式的老化可以阻止饥饿。

5.4 多处理器调度

多处理器分为非对称多处理和对称多处理（SMP）

处理器亲和性：努力让一个进程在同一个处理器上运行，避免进程在处理器之间转移。

负载平衡：设法将工作负载平均地分配到 SMP 系统中的所有处理器上。

5.5 例题精析

5.5.1

为什么对调度程序而言，区分 CPU 约束程序和 I/O 约束程序很重要？

答：I/O 约束程序通常是在 I/O 之前进行很短时间的计算，随后长时间的 I/O；而 CPU 约束程序则是长时间的计算，很少时间的 I/O；如果我们可以给 I/O 约束程序更高的优先级，让它在 CPU 约束程序之前执行，那么整个资源的利用率会提高，不会出现 I/O 约束程序长时间等待 CPU 的情况。

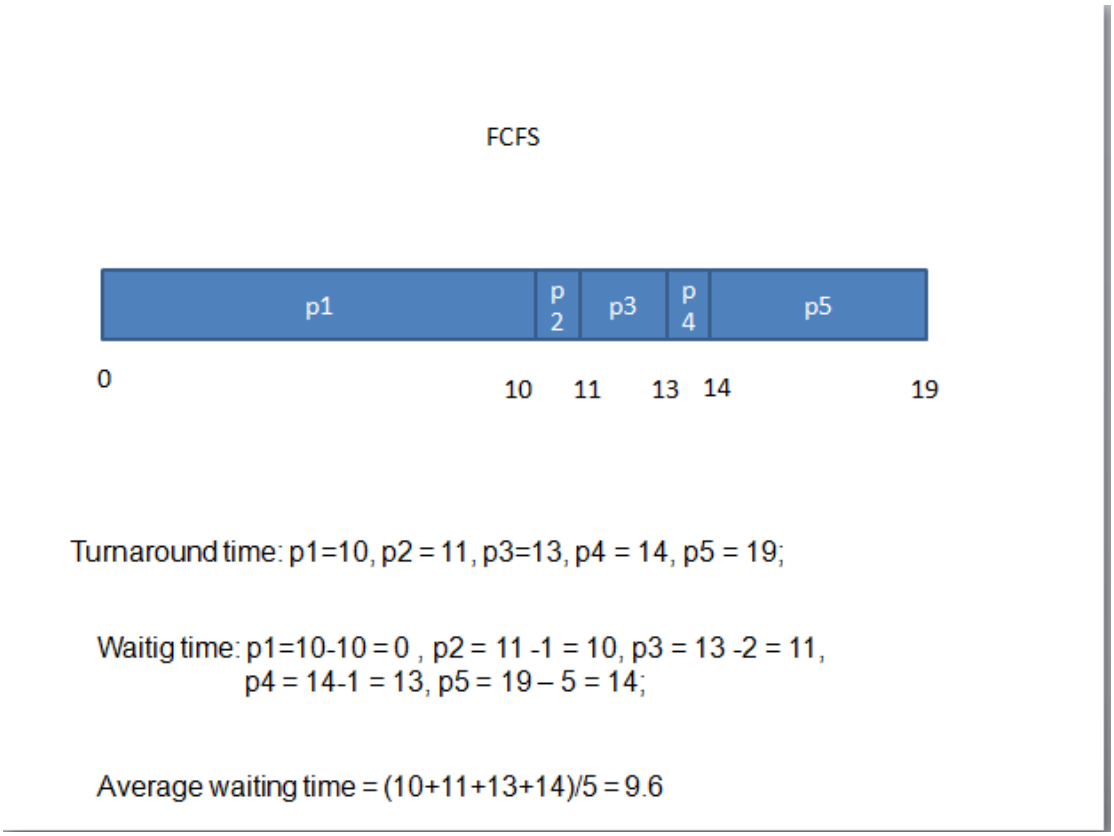
5.5.2

下面有一组程序

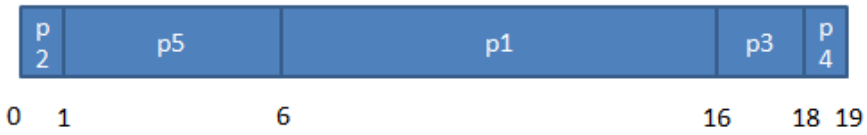
Process	Burst Time	Priority
P 1	10	3
P 2	1	1
P 3	2	3
P 4	1	4
P 5	5	2

用四个甘特图分别演示使用 FCFS，SJF，非抢占优先级，RR 算法的调度执行过程。

答案如下：



Nonpreemptive Priority

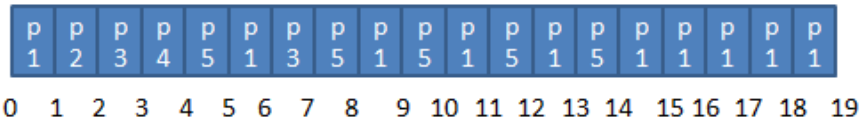


Turnaround time: p1=16, p2 = 1, p3=18, p4 = 19, p5 = 6;

Waitig time: p1=16-10 = 6 , p2 = 1 -1 = 0, p3 = 18 -2 = 16,
p4 = 19-1 = 18, p5 = 6 – 5 = 1;

Average waiting time = (6+16+18+1)/5 = 8.2

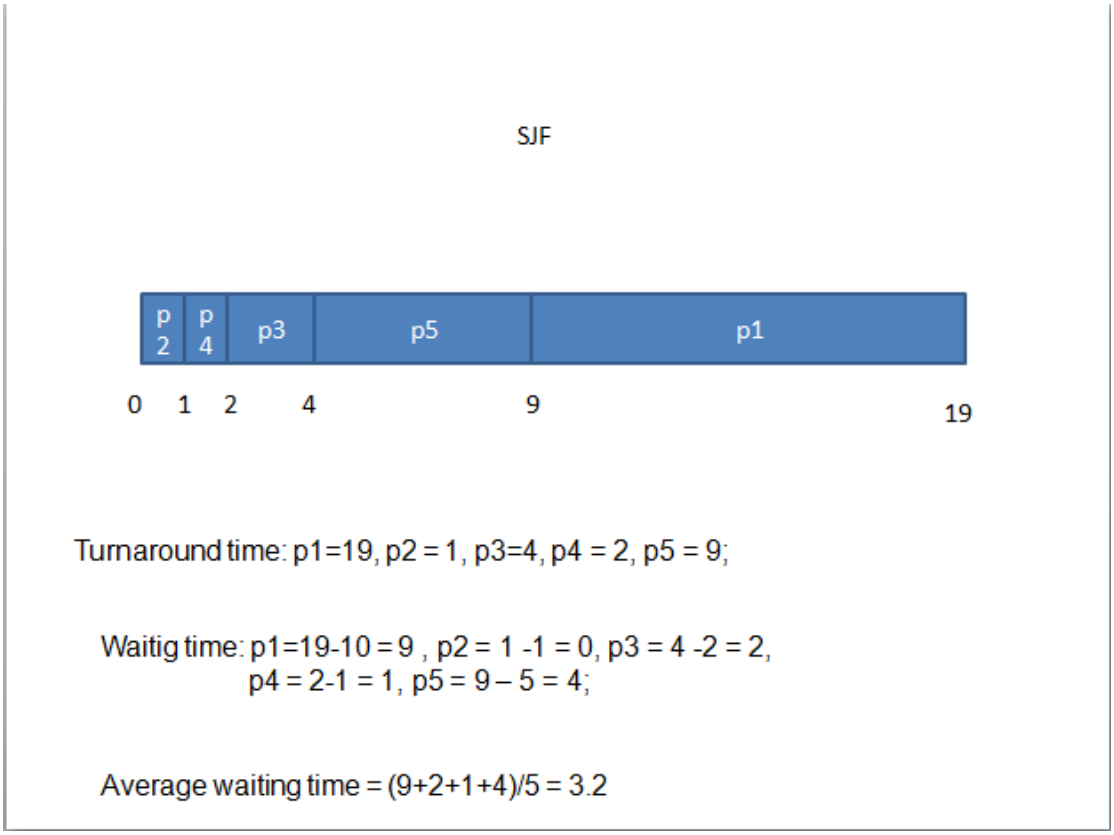
RR



Turnaround time: p1=19, p2 = 2, p3=7, p4 = 4, p5 = 14;

Waitig time: p1=19-10 = 9 , p2 = 2 -1 = 1, p3 = 7 -2 = 5,
p4 = 4-1 = 3, p5 = 14 – 5 = 9;

Average waiting time = (9+1+5+3+9)/5 = 5.4



5.5.3

下面哪种调度算法能导致饥饿？

a、FCFS

b、SJF

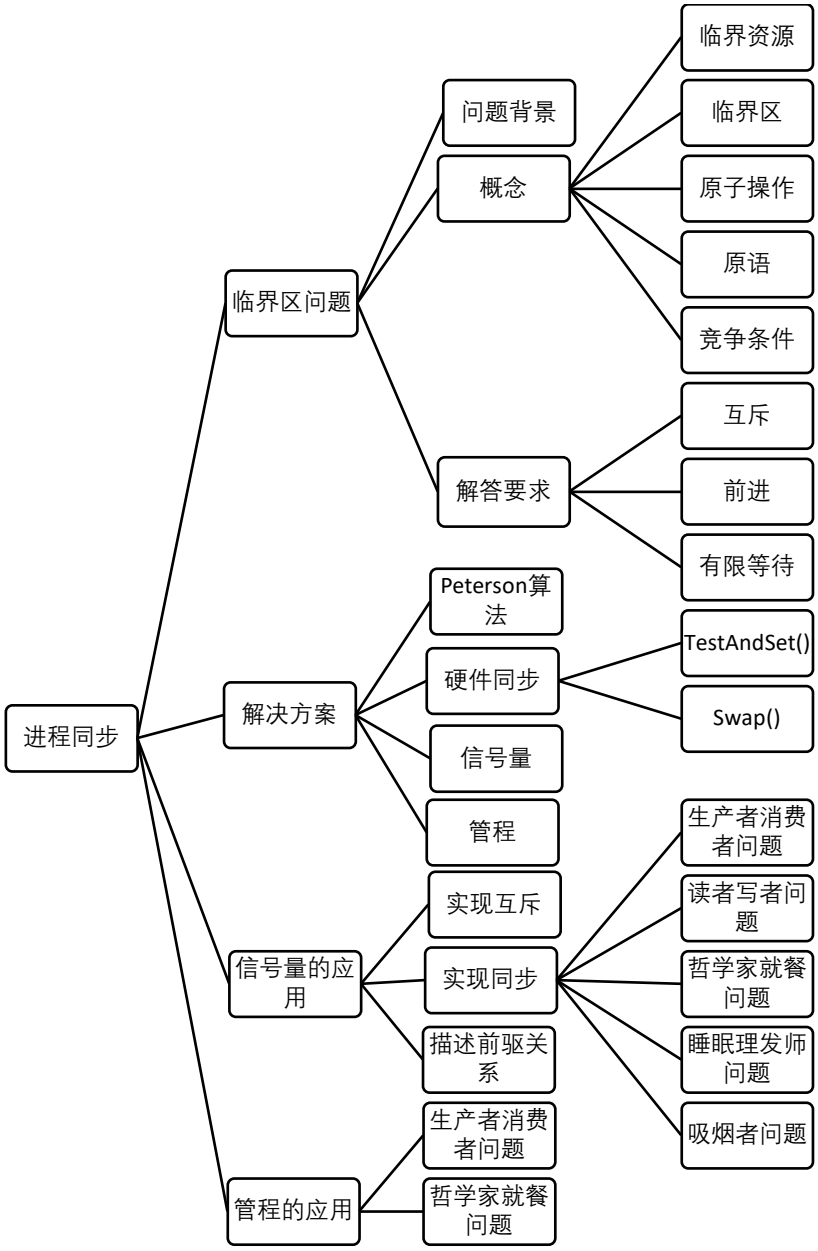
c、RR

d、优先级调度

优先级调度和 SJF 可能导致饥饿。优先级调度会让优先级低的进程饥饿，SJF 是让长进程饥饿。

第六章 进程同步

作者：林子童



进程同步是非常重要的一部分，除了基本的几个同步问题需要掌握以外，还要学会使用信号量灵活处理各种同步问题，这需要大量习题的练习积累。

6.1 临界区问题

6.1.1 背景

若干个进程或者线程同步进行的时候可能会共享数据，当它们几乎同时对一个数据进行处理的时候，很有可能会带来数据的不一致性。

例如，公园有一个出口一个入口，系统设置一个计数器记录公园中游客数量，这就是一个由一个中心数据库服务器及两个客户端组成的系统。

每进入一人，入口处系统执行如下操作：R1=count; R1++; count=R1;

每离开一人，出口处系统执行如下操作：R2=count; R2--; count=R2;

如果同时有人进入和离开，那么由于细小的时间差造成前面 6 条语句执行顺序不同，系统的 count 可能会出现+1、-1 和不变三种情况。

6.1.2 概念

(1) 原子操作 atomic operation：一个完整的没有中断的操作，要么全做要么全不做，是一种不可分操作。这些原子操作在核心态下运行，常驻内存

(2) 原语 primitive：一段完成一定功能的执行期间不能被中断的程序段

(3) 临界资源 critical resource：系统中可以供给多个进程使用但是同一段时间内却只允许一个进程访问的资源。当一个进程正在访问该资源时，其它欲访问的进程必须等待知道该进程访问完并释放该资源后。临界资源的例子包括一个变量、表格、文件、打印机等

(4) 临界区 critical section：程序中访问临界资源的那段代码，要将对临界资源的互斥访问转化为对临界区的互斥访问，即没有两个进程同时在临界区内执行

```
do {
      entry section
         critical section
      exit section
         reminder section
   } while (1);
```

临界区问题是设计一个以便进程协作的协议，每个进程必须请求进入其临界区（进入区 entry section），临界资源使用结束后还要有退出区 exit section，其他代码为剩余区 reminder section

(5) 竞争条件 race condition：多个进程并发访问和操作统一数据且执行结果和访问发生的特定顺序有关

6.1.3 临界区问题解答原则

(1) 互斥 mutual exclusion:忙则等待。进程不同时在临界区内执行

(2) 前进 prograss：有空让进。当无进程在临界区执行时，若有进程进入应允许

(3) 有限等待 bounded waiting：进程进入临界区的要求必须在有限时间内得到满足

(4) 让权等待 no busy waiting：等待的时候可以选择释放 CPU 执行权（非必须）

有两种方式用于处理操作系统内的临界区问题：抢占内核和非抢占内核。非抢占内核不允许处于内核模式的进程被抢占，根本不会导致竞争条件，因为只能有一个进程处于内核模式。但是抢占内核需要认真设计才能保证不会导致竞争条件。

6.2 解决方案

6.2.1 Peterson 算法

这是一个经典算法，可以通过这个算法理解同步的三个要求。这个算法经历了三个阶段。

(1) 使用令牌。turn 的值相当于给一个进程令牌。当获得这个令牌后，进入；在退出时，移交令牌。

```
Process Pi
do {
while (turn != i) ;
  critical section
turn = j;
  reminder section
} while (1);
```

```
Process Pj
do {
while (turn != j) ;
  critical section
turn = i;
  reminder section
} while (1);
```

这个方式存在问题：当获得令牌的那一方不执行程序的时候，没有令牌的一方即使在临界区空闲的时候也会等待，不符合 progress 规则。

(2) 登记簿。使用 flag[]做登记簿记录进程进入临界区

```
Process Pi
do {
  flag[i] := true;
  while (flag[j]) ;
     critical section
  flag [i] = false;
     remainder section
} while (1);
```

```
Process Pj
do {
  flag[j] := true;
  while (flag[i]) ;
     critical section
  flag [j] = false;
     remainder section
} while (1);
```

正常情况是可以的，但是当两者几乎同时到达的时候，可能两个进程测试另一个进程已经登记，就会互相谦让，都会陷入无限循环之中，不符合 progress，且会无限等待。

(3) 使用登记簿和令牌，欲进入临界区，则登记并将令牌移交给另一个进程

```
Process Pi
do {
  flag [i]:= true;
  turn = j;
  while (flag [j] and turn = j) ;
     critical section
  flag [i] = false;
     remainder section
} while (1);
```

```
Process Pj
do {
  flag [j]:= true;
  turn = i;
  while (flag [i] and turn = i) ;
     critical section
  flag [j] = false;
     remainder section
} while (1);
```

如果对方已经登记并且有令牌那么进入，否则自己进入。

可以看到，Peterson 算法只适合只有两个进程共享的资源。但是依然可以从中总结解题思路。分析给出的算法，找出特例说明这个机制违法了哪个准则。一般边界比较容易出问题，像同时申请易出毛病。

6.2.2 硬件同步(Synchronization Hardware)

通过上面我们可以看到，临界区问题都需要一个锁来防护，即进程进入临界区以前得到锁，退出临界区时释放锁。

硬件解决临界区问题，对于单处理器环境，在修改共享变量时禁止中断出现即可。但是多处理器需要将消息传给所有处理器，费时，所以不采用这种办法。

计算机提供了特殊硬件指令来简单的解决临界区问题。

(1) 指令 TestAndSet()主要特点是可以原子的执行，即两个指令在不同 CPU 上执行时会按顺序执行。使用这个指令需要声明一个 boolean 变量 lock 初始化为 false

这个指令的作用是返回 target 的现在的值并将 target 设置为 true。它的使用方法是，每一个进程都不停地执行这个指令直到发现 lock 为 false，说明 lock 正处于初识状态或者另一个占用临界区的进程已经解锁 lock

```
boolean TestAndSet (boolean *target)
{
   boolean rv = *target;      //取锁的状态(test)
   *target = TRUE;            //加锁(set)
   return rv:                 //返回锁原来的状态
}
```

```
while (true) {
     while ( TestAndSet (&lock )) ;
     //返回 lock的值，并置 lock为true
     //如果原来已经上锁，自己也上锁，
     //并等待
      //lock is true(自己已经加锁)
      //   critical section
      lock = FALSE;   //开锁
      //      remainder section
}
```

(2) 指令 Swap()与上面类似，不过要操纵两个数据 key 和 lock，lock 初始值为 false。swap 的功能就是交换两个 Boolean 值

其工作原理基本相似，进程不停地交换 lock 和 key 值，只有初始 lock=false 的时候或者另一个进程释放资源将 lock 设置为 false 的时候，交换获得的 key 值才能为 false，进程才能跳出循环

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b
   *b = temp:
}
```

```
while (true)  {
    key = TRUE; //自己加锁
    while ( key == TRUE)
    Swap (&lock, &key ) ;
    // key 与lock的值互换，lock now is true,
    // if lock原值为false，then key now is false;
    //如果原来已经加锁，自己也加锁，并等待
    //   critical section
    //   lock is TRUE, key is FALSE
     lock = FALSE;
    //      remainder section
}
```

6.2.3 信号量 Semaphore

(1) 概念

信号量 S 是一个整数变量，除了初始化以外，他只能通过两个标准的原子操作 wait()和 signal()来访问，这两个操作可以被简写为 P、V

下面是一个比较简单的 PV 操作的例子。可以看到，wait(S)方法相当于申请临界资源，申请不到就原地循环；signal 相当于在释放了临界资源以后发出一个信号。

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

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

(2) 用法

操作系统有计数信号量和二进制信号量（互斥锁），计数信号量值域不受限，互斥锁值只能为 0 或 1，可以提供互斥。

```
do {
        ...
        wait(mutex);
        critical section;
        signal(mutex);
        reminder section;
    } while (1);
```

二进制信号量可以处理多进程临界区问题，它们共用一个初始值为 1 的信号量 mutex，每个进程的结构如右图

计数信号量可以用来控制访问具有若干个实例的某种资源，此时 mutex 的值为剩余可用资源数目。需要该资源的时候使用 wait，释放使用 signal，计数值为 0 的时候所有请求该资源的进程都会阻塞直到有资源释放。计数信号量和二进制信号量本质上是相同的

(3) 实现

①自旋锁

上面定义的信号量主要缺点是忙等待 busy waiting，也就是所有处于等待的进程都必须进入代码中不停循环。这种类型的信号量也称为自旋锁 spinlock。

自旋锁缺点是循环等待，占用 CPU 时间；优点是循环等待的时候不需要进行上下文切换，减少系统开销，当等待锁的时间较短时，自旋锁是有效的。自旋锁一般在多处理器系统中使用。但在单处理器系统中，当一个进程等待一个事件，而该事件需要其它进程产生，而其它进程无法执行，也就无法产生该事件。后面要讲的非忙等信号量机制更适合于单处理器。

②非忙等信号量

有两个操作：

block：一个进程发现信号量不为正的时候进入等待状态，但是等待不是进入忙等，而是将自己阻塞。阻塞的方法就是将自己送入该信号量的等待队列中。

wakeup：signal 执行这个操作，将等待队列中的一个进程唤醒

为了实现这种操作，信号量被重新定义为一个结构，包含整型数和等待链表。那么 PV 操作：

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

```
wait (S)  {
   value--;
   if (value < 0) {
      add this process to waiting queue
      block(); }
}
```

```
signal (S)  {
   value++;
   if (value <= 0) {
   remove a process P from the waiting queue
   wakeup(P); }
}
```

忙等信号量值不可能为负，但是非忙等则有可能。value 为正数表示可用资源数；value 为负数表示等待进程数

可以看到，wait 就是对 value-1 同时在没有可用资源的时候将当前进程加入等待队列，而 signal 则是 value+1 同时将正在等待的一个进程唤醒。

信号量的关键在于原子性，这属于临界区问题。单处理器环境下可以在执行 PV 操作时简单的不允许中断，而多处理器环境必须提供加锁技术（如自旋锁）确保原子执行。

(4) 死锁与饥饿 deadlock，starvation

具有等待队列的信号量实现可能会导致死锁。

死锁：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。

如右图所示，S，Q 初值皆为 1，二者互等

饥饿：进程在信号量中无限等待

P_1	P_0
wait (Q);	wait (S);
wait (S);	wait (Q);
.	.
.	.
.	.
signal (Q);	signal (S);
signal (S);	signal (Q);

6.2.4 管程 monitor

信号量提供了一种方便有效的机制处理同步，但是使用不正确仍然会导致一些时序错误，如上面所说的死锁与饥饿。

管程类型提供了一组由程序员定义的、在管程内互斥的操作。管程中包括一组变量的声明和对这些变量操作的子程序和函数的实现。管程结构确保一次只能有一个进程在管程内活动。

定义：管程是一种程序结构，结构内的多个子程序形成的多个工作线程互斥的访问共享资源

管程中需要定义一些额外的同步机制，这些可由条件 condition 结构提供。condition x,y;条件变量仅有的操作是 wait()和 signal()；当某进程通过管程请求临界资源而未满足时，管程调用 wait 原语使该进程等待，并将它排在等待队列上；当另一进程访问完并释放之后，管程调用 signal 原语唤醒等待队列中的(队首)进程；

可以把管程的定义理解为一个类的定义，与一般类不同的是，管程有条件变量可用于控制进程之间的同步。

管程的实例将在后面讨论。

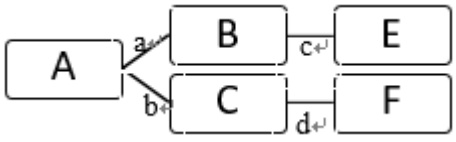
6.3 *信号量的应用

6.3.1 互斥

实现互斥很简单，初始化一个二进制信号量为 1，在临界区前后分别加上 wait 和 signal 操作即可。

6.3.2 描述前驱关系

A:x=i+j;y=p+q
B:z=x+i
C:k=y+q
E:h=z+j
F:l=k+p



描述前驱关系的题目，只需要将前趋图画出来，点为进程，线为信号量，根据前驱关系用信号量协调进程的执行。上面的例题中，ABCDE 分别是五个进程，通过上面的算式可以看抽象出：A 是 BC 的前驱，B 是 E 的前驱，C 是 F 的前驱。根据抽象出来的关系，画出前驱图如上，而每个进程如下：

```
A{A;signal(a);signal(b);}
B{wait(a);B;signal(c);}
E{wait(c);E;}
```

6.3.3 同步

两进程合作问题三要素：信号量设置、信号量初值、算法描述

信号量：同一信号量 P、V 必须成对出现，互斥信号量必须成对出现在一个程序中，资源信号量出现在不同程序中；多个 wait 操作顺序是：同步在前、互斥在后

(1) 生产者消费者问题（有限缓冲问题）Bounded-Buffer

问题描述：缓冲池中有 n 个缓冲项，每个缓冲项能存一个数据，有多个生产者向缓冲池中送数，同时有多个消费者从缓冲池中取数。所有生产者消费者因为都要修改缓冲池，所以必须要互斥

生产者等待消费者取数，消费者等待生产者放数，二者等待的信号量不是同一信号量。同时，由于互斥访问，还需要加一个互斥锁

```
Shared data
semaphore empty, full, mutex;
Initially:
empty= n, full = 0, mutex=1;
```

生产者

```
while (true) {
        // produce an item
     wait (empty);
     wait (mutex);
        // add the item to the buffer
     signal (mutex);
     signal (full);
}
```

消费者

```
while (true) {
     wait (full);
     wait (mutex);
          // remove an item from buffer
     signal (mutex);
     signal (empty);
          // consume the removed item
}
```

(2) 读者写者问题 Readers and Writers

问题描述：一个数据库被多个并发进程共享，有的进程只需要读数据库，读者不需要互斥；有的进程需要写数据库，写者之间必须互斥并且与读者互斥。写者进程比较简单，只需要实现简单的互斥。

对于读者，有两个比较重要：第一个进入的和最后一个离开的。第一个进入的应该拒绝写者但不能拒绝其他读者；最后一个离开的应唤醒写者。因此需要一个计数器记录读者个数。计数器被读者共享，需要互斥锁。

初始值：Semaphore mutex=1, wrt=1；

int readcount=0；

写者

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

读者

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

上面的这个算法可能会导致写者饥饿。

另一种算法是只要写者准备就绪，就会阻止新的读者进入，这种方法的实现需要加上一个互斥锁 read=1 将写者、读者的进入这两部分互斥。

(3) 哲学家就餐问题 Dining-Philosophers

问题描述：五个哲学家用五支筷子在一个圆桌上就餐，每个哲学家需要两只筷子吃饭。

最简单的方法是分别为五支筷子设置信号量初始化为 1,每个哲学家的进程都是申请两只筷子吃饭然后释放。

```
Semaphore chopstick[5];
While (true)  {
       wait ( chopstick[i] );
       wait ( chopStick[ (i + 1) % 5] );
             // eat
       signal ( chopstick[i] );
       signal (chopstick[ (i + 1) % 5] );
           //  think
}
```

但是这样会导致死锁，也就是会出现一人拿到一只筷子的情况

解决方案：

①最多允许 4 个人坐在桌上，即设置一个信号量 seat=4（下左图）

②第 5 个哲学家先申请右边的筷子，其他人先申请左边的筷子

③第奇数个哲学家先申请左边，第偶数个先申请右边（右图）

④使用管程

```
do {
   wait(seats)
   wait(chopstick[i])
   wait(chopstick[(i+1) % 5])
      // eat
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
   signal(seats)
      // think
   } while (1);
```

```
do {
     if (i%2==0) {
            wait(chopstick[i])
            wait(chopstick[(i+1) % 5])  }
     else {
            wait(chopstick[(i+1) % 5])
            wait(chopstick[i])    }
            //eat
     signal(chopstick[i]);
     signal(chopstick[(i+1) % 5]);
            //think
     } while (1);
```

（4）睡眠理发师问题

问题描述：某理发店有一个接待室和一个理发室组成。理发室中有一把理发椅，接待室中有 n 把椅子。若没有顾客等待理发，则理发师睡眠等待。当一个顾客到达理发店后，若发现座位已满，则选择离开；若发现理发师忙而接待室中有空座位，顾客则坐在椅子上等待；若发现理发师正在睡眠，则将理发师唤醒。

问题分析：

理发师：发现有顾客，则呼叫下一个，否则等待

顾客：到达理发店，没有空座位则离开；否则告诉理发师有人在等待然后坐下等待理发师呼叫

需要整型变量 `waiting=0` 记录等待的顾客数，互斥锁 `barber=0` 记录理发师是否可以服务，互斥锁 `mutex=1` 实现对 `waiting` 的互斥访问。记住，这个模型初识状态是理发师在睡觉，需要被唤醒

```
Barber:
 while (true) {
    wait(customers); //检查有无顾客
    //等待被来到的顾客唤醒
    wait(Mutex); // 实现对waiting的互斥访问
    waiting=waiting-1;
    signal(Mutex); //释放waiting的访问权
    signal(barber); //理发师准备好可以服务
    cut-hair;          //理发
}
```

```
Customer:
While (1) {
   wait(waitingMutex); //实现对waiting的互斥访问
   if (waiting < CHAIRS) {  //如果有座位空闲
     waiting=waiting +1;
     signal(waitingMutex);
     signal(customers);      //通知理发师
     wait(barberReady);      //等待理发师呼叫
   }
   else  { // 理发店已满，离开
     signal(waitingMutex);
     leaving;
   }
}
```

（5）吸烟者问题

问题描述：一支烟需要三种材料，系统中有三个吸烟者每人各有一种材料同时有一个进程随机供应两种材料。吸烟者等待适配的材料然后抽烟。

三个吸烟者进程，一个供应者进程，三个供应材料信号，一个吸烟结束信号。以一个吸烟者为例：

```
smokers:
// the smoker that has matches
while( true ) {
     wait( tobacco_paper );
    // roll cigarette and smoke
     signal( doneSmoking ); }

// the smoker that has paper
while( true ) {
     wait(tobacco_matches ) ;
    // roll cigarette and smoke
     signal( doneSmoking ); }

// the smoker that has tobacco
while( true ) {
     wait(paper_matches ) ;
    // roll cigarette and smoke
     signal( doneSmoking );  }
```

```
agent:
while( true ) {
     pick a random number from 1-3;
     if random number is 1  {
      // put tobacco and paper on table
      signal( tobacco_paper ) }
     else if random number is 2  {
        // put tobacco and matches on table
         signal( tobacco_matches )
     else if random number is 3 {
       // put paper and matches on table
       signal( paper_matches ) }
     wait( doneSmoking )
   }  /* while */
```

6.4 管程的应用

6.4.1 利用管程解决 P-C 问题

根据这个实例可以基本了解管程的工作原理

(1) 建立一个管程，其中包含两个过程

put(item):生产者利用该过程将自己生成的消息放到缓冲池；缓冲池满的时候进入等待。

get(item):消费者利用该过程获得一个消息，缓冲池空的时候进入等待。

显然，需要一个变量 count 表示缓冲池中已有消息数，

```
monitor pc
 {
       int in,out,count;
        item buffer[n];
        condition empty,full;
        void put(item i)
        void get(item i)
        void init() {
                in=0;
                 out=0;
                 count=0;
        }
 }
```

(2) 两个过程的使用

使用管程的作用就是简化时序，对信号量就简分析。可以看到，生产者只需要生产然后 put，消费者只需要获取然后消费，不需要考虑等待。这也就使得 put 和 get 的设计变得尤为重要

```
Producer: //生产者
do {
      …
      produce an item in nextp;
      pc.put(nextp);
      …
} while (1);
```

```
Consumer:     //消费者
do {
      …
      pc.get(nextc);
      consumer the item in nextc;
      …
} while (1);
```

(3) 实现这两个过程

```
void put(item i)
{
  //缓冲池满，等待空缓冲区
  if (count >= n) empty.wait;
  buffer[in] = i;
  in =  (in+1)%n;
  count++;
  //有等待则唤醒没有则不修改
  full.signal;
}
```

```
void get(item i)
{
  //缓冲池空，等待满缓冲区
  if (count <= 0) full.wait;
  i = buffer[out] ;
  out =  (out+1)%n;
  count- -;
  //若有生产者等待，则唤醒 之
  empty.signal;
}
```

6.4.2 哲学家就餐问题

筷子的分配由管程来控制，哲学家只管申请筷子和放回筷子

dp.pickup (i)//如果饥饿并且左右两只筷子空闲则吃饭否则等待 EAT

dp.putdown (i)//放下筷子测试左右邻居是否在等待，如果等待则唤醒

```
monitor DP
  {
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];  //为每个哲学家分别设置一个条件变量

   void pickup (int i) {
        state[i] = HUNGRY;
         //一个哲学家吃饭应具备的条件：1、自己状态为饥饿 2、左右两个哲学家都不在吃饭
         //如果一个哲学家具备吃饭的条件，则把自己的状态设为正在吃饭，
         //如果自己原来等待吃饭，则被唤醒；
        test(i);
        if (state[i] != EATING) self [i].wait;  //如果测试后自己不具备吃饭的条件，则等待
   }

     void putdown (int i) {
        state[i] = THINKING;
             // test left and right neighbors
         test((i + 4) % 5);  //自己吃完饭，测试左边的哲学家是否等待吃饭，如果是，唤醒之；
         test((i + 1) % 5);  //自己吃完饭，测试右边的哲学家是否等待吃饭，如果是，唤醒之；
      }
  void test (int i) {
       //如果左右两个哲学家都不在吃饭且自己饥饿，则把自己的状态设为吃，
       //如果自己原来等待吃饭，则被唤醒
       if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) )  {
           state[i] = EATING ; //自己具备了吃饭的条件
            self[i].signal () ; //自己被唤醒
         }//如果不满足条件，可以看到pickup方法中本方法后还有个if语句
         //意思是不满足条件那么进入等待
  }
    initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
  }
}
```

6.4.3 总结

通过前面两个例子可以看到，使用管程的时候，每个进程都不需要考虑别的进程怎么样、临界资源怎么样，而是把这些考虑留给了内部的实现。

6.5 *例题精析

6.5.1 需要掌握的概念：

race condition，critical reource，critical section、atomic operation,semaphoer,wait() and signal() operation，monitor，busy waiting

6.5.2 进程同步与互斥问题解题技巧

(1) 生产者-消费者问题

特点：对同一块东西有取有放，取放的所有个体之间都是互斥的。这种类型只要抽象出生产者和消费者以及信号量 full、empty 就可以很简单的做出来了。

例题：有一个仓库，可以存放 A 与 B 两种产品，仓库的存储空间足够大，但要求：

①每次只能存入一种产品（A 或 B）；

②-N<A 产品数量 – B 产品数量<M；

其中，N 和 M 是正整数。

试用“存放 A”和“存放 B”和 wait、signal 描述产品 A 与产品 B 的入库过程。

解析：每次存放 A 以前都要检查 A-B 是否已经超过 M-1 了，存放 A 结束以后告知 B-A 这个数需要减一，也就是如果 B 原本已经到达上限，那么现在就又可以继续放 B 了。B 的存放同理。现在将 A-B 和 B-A 抽象为两个信号量。

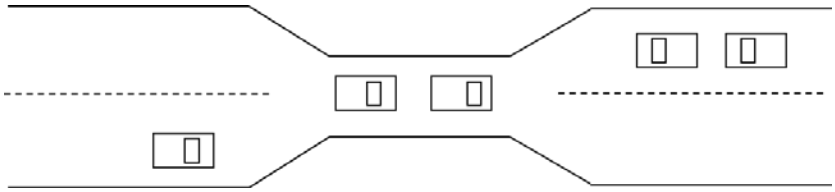
```
semorphore
mutex=1
sa=M-1
sb=N-1;
```

```
Process Input _A:
    while (true){
        Get a product A;
        wait(sa);
        wait(mutex);
        // put product A into the
        // depository;
        signal(mutex);
        signal(sb); }
    }
```

```
Process Input _B :
    while (1) {
        Get a product B;
        wait(sb);
        wait(mutex);
        // put product B into the
        // depository;
        signal(mutex);
        signal(sa);
    }
```

(2) 读者写者问题

这个问题最大的特色是第一个读者进入以后，读者就可以连绵不断地进入直到最后一个人离开才让写者进入。汽车过桥问题就是从这个模型中变形出来的



桥只允许一辆车通过，且一辆车上桥以后，它后面的车就可以跟着上桥直到这一边的车走完。这相当于两种写者

其他问题抽象比较简单，不再赘述。现在讨论无法抽象的问题

(3) 无法抽象的题目解题思路

①抽象出几个具体的进程

②分析每个进程完成的主要事件，在相应的进程中罗列出来

③对进程之间的时间分析，看哪些进程是相互制约的，在被制约的事件前加上 wait，在制约事件后加上 signal

④根据制约关系确定信号量的个数

例题：某寺庙，有小和尚、老和尚若干。庙内有一水缸，由小和尚提水入缸，供老和尚饮用。水缸可容纳 30 桶水，每次入水、取水仅为 1 桶，不可同时进行。水取

自同一井中，水井径窄，每次只能容纳一个水桶取水。现有水桶 5 个，供小和尚入水及老和尚取水使用。规定入水、取水后，把桶放下，使用时再重新取。

解析：只有老和尚和小和尚两个进程。老和尚做的就是申请桶、申请缸里的水来打水，最后放下桶；小和尚打水前则要考虑缸是否已经满了、是否有水桶、是否有人正在水井打水。这道题可以理解为 P-C 问题，不过已经没有那个必要了

初始化

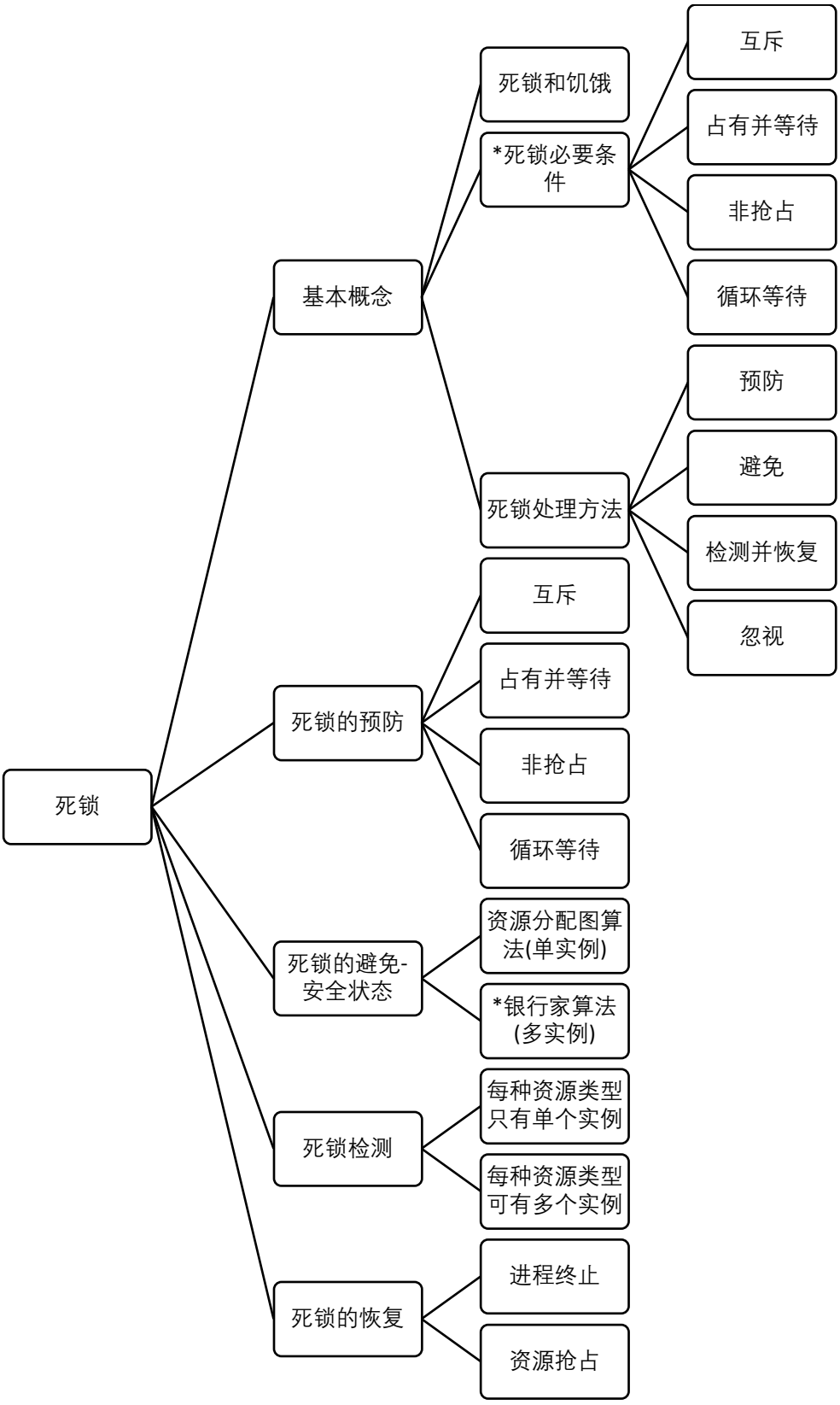
```
semaphore empty=30; // 缸中目前还能装多少桶水
semaphore full=0; // 缸中有多少桶水
semaphore buckets=5; // 有多少只空桶可用
semaphore mutex_well=1; // 用于实现对井的互斥操作
semaphore mutex_bigjar=1; // 用于实现对缸的互斥操作
```

```
young_monk() { //到井中打水，然后倒入缸中
while(1){
   P(empty);       //水缸是否已满
   P(buckets);     //申请一个空桶
   get a bucket; //可以考虑对取桶操作实现互斥
   go to the well;
   P(mutex_well); //实现对井的互斥操作
   get water;
   V(mutex_well); //释放井的使用权
   go to the temple;
   P(mutex_bigjar); //实现对缸的互斥操作
   pour the water into the big jar;
   V(mutex_bigjar); //释放缸的使用权
   V(buckets); //将桶放下
   V(full);       //老和尚可以取水
  }
}
```

```
old_monk() { //取水饮用
while(1){
   P(full);        //缸中是否有水
   P(buckets);  //申请一个空桶
   get a bucket; //考虑对取桶操作实现互斥
   P(mutex_bigjar); //申请对缸的使用权
   get water;
   Drink water;
   V(mutex_bigjar); //释放水缸的使用权
   V(buckets);  //放下桶
   V(empty);    //小和尚可以打水了
  } while(1)
}
```

第七章 死锁*(本章例题已用于知识点讲解)*

作者：林子童



7.1 基本概念

7.1.1 死锁和饥饿

(1) 死锁：一组处于等待（阻塞）状态的进程，每一个进程持有其他进程所需要的资源，而又等待使用其他进程所拥有的资源，致使这组进程互相等待，均无法向前推进。另一种定义：当一组进程中每个进程都在等待一个事件，而这一事件只能由这一组进程的另一个进程引起时，这组进程处于死锁状态。

(2) 饥饿：就绪进程长时间得不到调度是处于等待状态，而不是死锁中的互相等待。若信号量的等待队列按照 LIFO 或优先级管理，则可能导致饥饿。

7.1.2 进程使用资源的顺序

(1) 申请

(2) 使用

(3) 释放

资源的申请与释放为系统调用：设备 request()/release();文件 open()/close();内存 allocate()/free()；其他资源申请与释放可以通过信号量 wait()/signal()操作来完成。

7.1.3 死锁特征（必要条件）

如果四个条件同时满足，就会引起死锁

(1) mutual exclusion 互斥：至少一个资源要求互斥地共享

(2) hold and wait 占有并等待：一个进程至少占有一个资源并等待另一资源，该资源为其它进程所占有

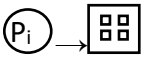
(3) no preemption 非抢占：资源不能被抢占，只能进程完成任务后自动释放

(4) circular wait 循环等待：互相等待形成一个环

7.1.4 资源分配图

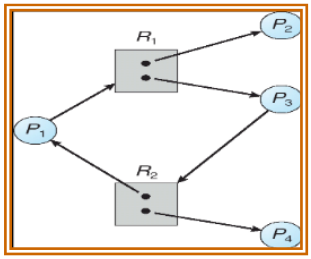
 —一个进程

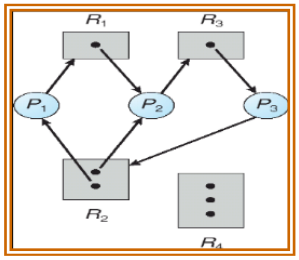
 —一个拥有 4 个实例的资源

 进程 P_i 申请一个资源 R_j 实例

 进程 P_i 占有资源 R_j 的一个实例

资源分配图如果没有环就一定没有死锁，如果有环且环中每种资源只有一个实例也一定是死锁，否则就要根据四个必要条件展开讨论。





左图中 p1、p2、p3 每个进程都占有资源并等待着其他进程释放资源，且循环等待，所以形成了死锁；虽然右图进程中也成环了，但是可以看到 p2 和 p4 都没有在等待组内的资源，任意一个进程结束释放资源后，环就会消失，所以不是死锁。

7.1.5 死锁的处理方法

从原理上讲，有三种方法可以处理死锁：

(1) 使用协议预防或避免死锁

死锁预防 prevention 是一组方法，需要确定至少一个必要条件不成立

死锁避免 avoidance 要求操作系统事先得到有关进程申请使用资源的额外信息。当进程申请资源时，若发现满足该资源的请求可能导致死锁发生，则拒绝该申请。

(2) 允许进入死锁状态，检测并加以恢复

(3) 忽视死锁问题：大多数系统使用，因为死锁发生并不频繁，预防、避免和恢复耗费太大。

7.2 死锁预防

预防的特点是一定不会死锁，但是会降低资源利用率和系统吞吐量。

7.2.1 互斥

非共享资源必须互斥，共享资源不要求互斥所以不会死锁。无法从互斥条件下手避免死锁。

7.2.2 占有并等待

(1) 拥有不等待，资源静态分配策略，要求一个进程在执行前获得所有资源

(2) 等待不拥有，进程在申请其他资源的时候必须释放已分配的资源。

缺点：资源利用率低，分配以后可能很久不被使用；产生饥饿，对于第一种协议，如果有进程需要多个常用资源，就可能会永久等待。

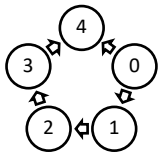
7.2.3 非抢占

如果一个进程占有一些资源并在申请一些无法立刻分配到的资源，那么它占有的这些资源就都可以被抢占。该进程将会在它重新获得原有的资源以及原本要申请的资源的时候重启。

用于状态可保存和恢复的资源如 CPU，memory 等，不适用于打印机等。

7.2.4 循环等待

对所有资源类型进行完全排序，要求每个进程按递增顺序申请资源。



如哲学家就餐问题，桌上的五根筷子分别编号，所有人只能按照从小到大的顺序申请筷子，所以永远不会出现每个人获得一只筷子的情况。

再比如，当一个进程对资源的使用顺序为 5→2→4→3→1 的时候，它的申请顺序仍然是 12345，也就是说即使 1 是最后使用，也要先申请。这种浪费在程序越大、申请资源越多的时候越是明显。

7.3 *死锁避免

死锁避免算法动态的检测资源分配状态以确保循环等待条件不可能成立。资源分配状态 resource-allocation state 是由可用资源、已分配资源和进程最大需求所决定的。所以这种算法要求每个进程说明可能需要的每种资源类型实例的最大需求。

死锁避免算法会因为追踪当前资源分配成本增加运行成本，但是相对于静态的死锁预防方法它允许更多的并发使用资源，所以系统吞吐量大于死锁预防。

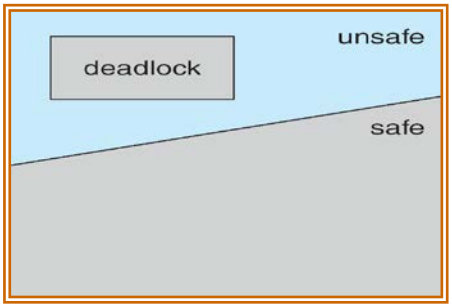
7.3.1 安全状态 safe state

如果系统能按照某个顺序为每个进程分配资源并能避免死锁，系统状态就是安全的。或者说，如果存在一个安全序列 safe sequence<P0,P1,P2,…,Pn>使得前面的进程能够得到足够的资源完成，同时它释放的资源又能满足后面的进程的话，就是安全的。下面给出一个简单的例子：

process	maximum	allocation	need	available
p0	10	5	5	3
p1	4	2	2	
p2	9	2	7	

现在给出一个安全序列<P1,P0,P2>，之所以说它是安全序列，可以这样分析：P1 分配到 2 个资源以后可以完成任务释放 4 个资源，那么总共可用资源变成 5 个，分配给 P0 以后待其完成任务释放 10 个资源可以让 P2 使用，这个分配顺序能确保不会产生死锁，所以系统状态是安全的。

但是，不安全状态并不一定会导致死锁。因为我们这里的最大资源需求量很有可能会超出真正需求量，反之，死锁状态一定是不安全状态。



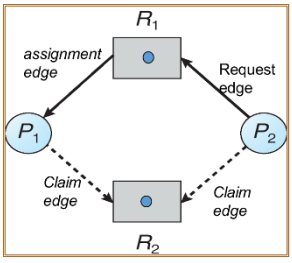
有了安全状态的概念，就可以定义必避免算法以确保系统不会死锁，其思想就是简单的确保系统始终处于安全状态。

采用这种方案，如果进程申请一个现已可用资源，也可能必须等待。因此资源利用率比没有采用死锁避免的算法低一些。

7.3.2 单实例——资源分配图(RAG)

single instance，resource-allocation graph

使用具有 claim 边的 RAG，适用于每种资源类型有单个实例的资源分配系统



虚边：将要请求或可能使用 claim edge

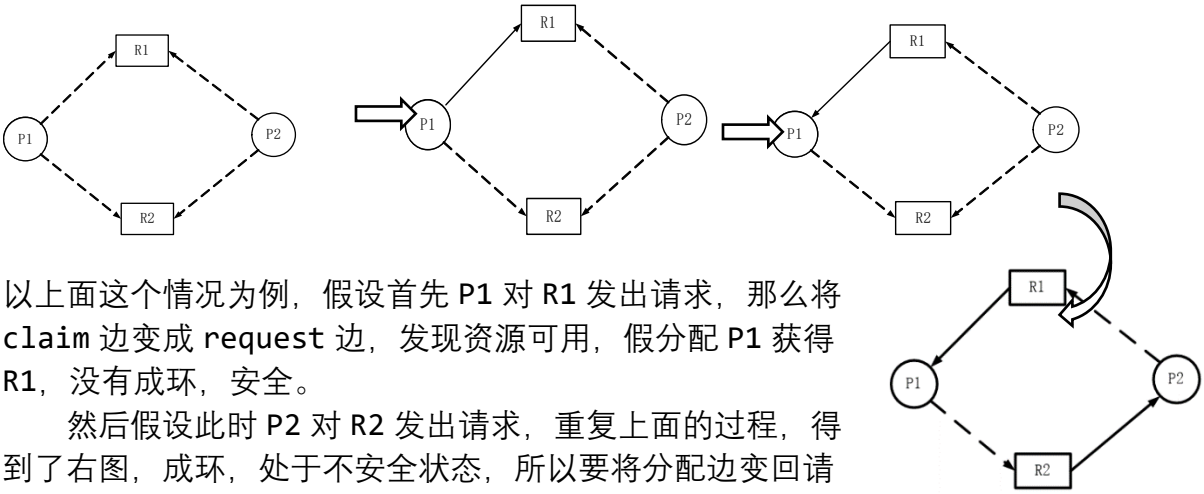
实边：请求边和分配边 request，assignment

规定：图中如果没有环，就是安全状态；如果有环，即使环中有虚边，也是不安全状态，在避免算法里面是不允许出现的；如果出现了实边环，就是死锁。

算法步骤：

①请求资源，看资源是否可用，不可用等待，可用转下一步

②假分配看是否会出现死锁，如果不会就进行分配，如果会有死锁那么请求资源的进程进入等待状态



以上面这个情况为例，假设首先 P1 对 R1 发出请求，那么将 claim 边变成 request 边，发现资源可用，假分配 P1 获得 R1，没有成环，安全。

然后假设此时 P2 对 R2 发出请求，重复上面的过程，得到了右图，成环，处于不安全状态，所以要将分配边变回请求边，进入等待状态。

7.3.3 *多实例——银行家算法 multiple instances，Banker’s Algorithm

效率低于资源分配图

当新进程进入系统时，必须说明其可能需要的每种类型资源实例的最大数量。用户申请一组资源时，系统必须确保这些资源分配后系统仍处于安全状态。

设系统中共有 n 个进程和 m 种资源类型

(1) 安全性算法：确定计算机系统是否处于安全状态

①向量 finish[n]存储进程是否已经完成，初始状态为 false；向量 work[m]存储当前每种资源的剩余可用量，初始值为资源总量。

②寻找是否存在 finish=false 且所有资源 need≤work 的进程，如果存在，让这个进程获得所需要的资源执行结束，然后释放资源，这个时候它的 finish=true，而总的 work 也要加上该进程原来占有的资源

③循环执行②直到没有符合条件的进程。这时候如果 finish 全部为 true，那么系统处于安全状态，我们就能获得一个安全序列

(2) 资源请求算法：判断是否可安全允许请求

①确定 request≤need 否则出错；request≤available，否则等待；

②按照请求假分配，修改系统当前的 available、need，然后判断假分配以后系统状态是否安全。如果安全，该分配得到允许

(3) 举例

初始状态：根据所给的条件，列出下面的表格，注意，如果给的是 max，那么通过 max-allocation 获得 need。

process	allocation			need			available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

可以给出一个安全序列<P1，P3，P4，P2，P0>，安全序列的寻找遵循上面的安全性算法，根据 available 和 need 的关系来获得。

这个时候，如果 P1 发出请求(1,0,2):

①request≤need，√；request≤available，√。

②列出假分配以后的资源分配状态表

process	allocation			need			available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3			
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0	2	3	0
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

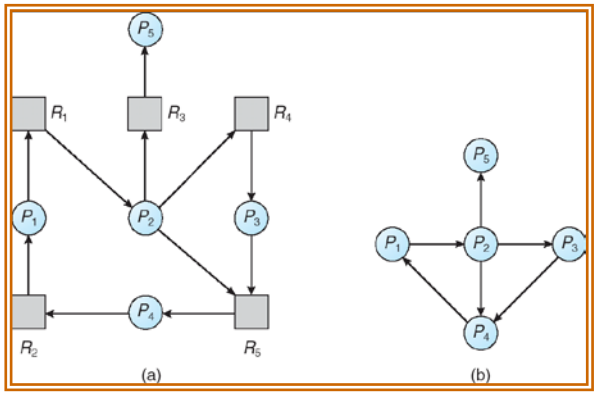
根据安全性算法求出一个安全序列<P1,P3,P4,P0,P2>,那么资源分配以后仍然在安全状态，请求被允许。

7.4 死锁检测

检测并恢复方案会有额外开销，包括维护所需信息和执行检测算法的运行开销和死锁恢复所引起的损失

7.4.1 每种资源类型只有单个实例：等待图 wait-for graph

点表示进程，$P_i \rightarrow P_j$ 表示 P_i 在等待 P_j。如果等待图中出现了环，就存在死锁。



(a)资源分配图

(b)相应的等待图

通过资源分配图可以看到 P5 占用了 R3，P3 占用了 R4，P4 占用了 R5，P1 占用了 R2，P2 占用了 R1，那么相应的等待资源也就变成了等待占有该资源的那个进程。

7.4.2 每种资源类型可有多个实例

与银行家算法相似，只不过现在我们不需要知道每个进程的最大需求资源量，而是假定在一个进程的请求被接受以后执行完这一段程序就会释放占用的资源，其实本质上和银行家算法是一样的，不过是把 need 变成了 request。银行家算法是重点内容，为了防止读者混乱，在这里再列一个多实例死锁检测的例子，就会发现它和银行家算法并没有什么不同

T0 时刻

process	allocation			request			available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0			
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0	0	0	0
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

可以很容易求出一个安全序列<P0,P2,P3,P1,P4>,检测结果：没有死锁

这时如果 P2 又有了一个 C 的请求，活用银行家算法可以发现无论如何 finish 向量中都会有 false，所以就检测到了死锁

死锁检测算法与死锁避免的算法的不同在于，银行家算法是假分配，而这里是分配以后进行检测，相对而言变得简单了一些

7.5 死锁恢复

7.5.1 进程终止

无论采用哪种方法，系统都会回收分配给终止进程的所有资源。

(1) 终止所有死进程：代价高昂，被终止的进程都要重新计算

(2) 一次只终止一个直到死锁消失：开销大，因为需要不停调用死锁检测

7.5.2 资源抢占

从进程中抢占资源给其他进程用。需要处理以下三个问题：

(1) 选择一个牺牲品：代价最小化

(2) 回滚：被抢占的进程需要回滚到安全状态

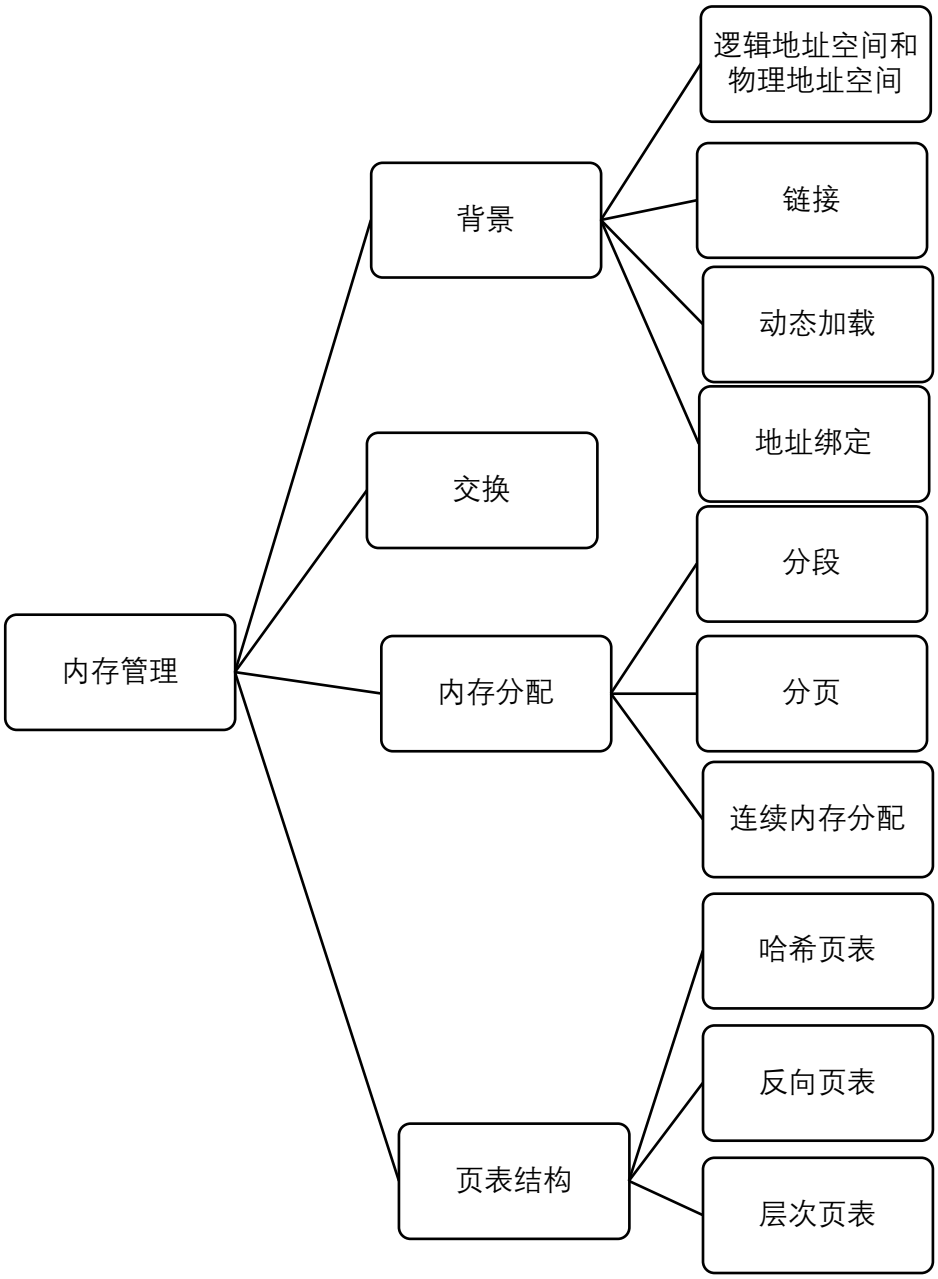
(3) 饥饿：必须保证不会发生饥饿，因为有可能被抢占的总是同一个进程。

第三部分 内存管理

内存管理主要在于追求内存资源的最大利用率，采用连续内存分配、分页、分段三种主要方法。虚拟内存是“小内存运行大程序”，给进程按照什么方式分配内存、分配多少内存、程序以哪种方式调入内存（按需调页等）、调入内存后所产生的内存管理问题（页面置换、抖动等）等，牵扯到按需调页、写时复制、页面置换的不同方式、抖动等问题。

第八章 内存管理

作者：徐卫霞



8.1 背景

8.1.1 地址绑定

指令和数据绑定到内存地址

方式：

（1）编译时：编译时就知道进程将在内存中的驻留地址，故在编译时生成绝对代码。例如继承驻留在内存地址 R 处，则生成的编译代码就从 R 处开始向后扩展，若开始地址发生改变，需要重新编译。

缺点：不利于程序在内存中的浮动，不支持虚拟内存机制。

（2）加载时：绑定在加载时完成。若开始地址改变，只需要重新加载用户代码来引入改变值。

缺点：不利于程序的浮动，不支持虚拟内存机制。

（3）执行时：进程在执行时可以从一个内存段移动到另一个内存段，则绑定必须延迟到执行时完成。要有硬件支持，支持虚拟内存机制。

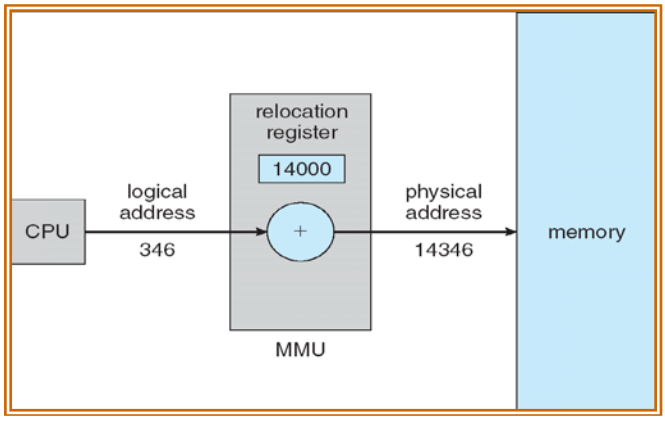
8.1.2 逻辑地址空间和物理地址空间

（1）逻辑地址空间：CPU 生成的地址称为逻辑地址（也称虚拟地址），由程序所生成的所有逻辑地址的集合叫逻辑地址空间。

物理地址空间：内存单元的地址叫做物理地址，与程序的逻辑地址空间所对应的为物理地址空间。

（2）映射：虚拟地址到物理地址的映射由内存管理单元（MMU）完成，此处先用一个简单的映射方式介绍：利用一个基地址寄存器（此处称为重定位寄存器）完成映射，如下图：

若逻辑地址范围为 0~max，基地址为14000，则对应的物理地址为 14000~14000+max



（3）动态加载：

定义：主程序装入内存执行，当一个子程序需要被调用时将其装入内存。

优点：提高了内存的利用率

缺点：管理复杂，执行速度慢。

绝对方式加载：在进程执行前将其所有模块都装入内存。内存利用率低，但是执行速度快，管理简单。

（4）链接：

静态链接：

定义：运行前将所有程序模块链接在一起，形成可执行文件，运行时直接装入内存。

优点：运行速度快。

缺点：链接及装入过程费时，可能装入用不到的模块，不利于模块的升级。

动态链接：

定义：运行时链接，且仅链接需要的模块。

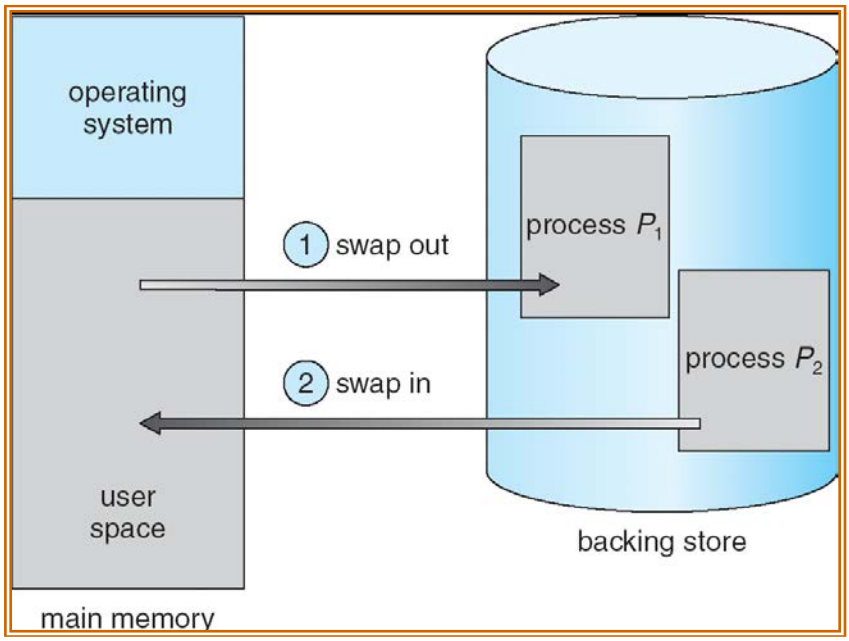
优点：便于模块的升级，减少了链接的时间，节省内存空间。

缺点：运行速度慢。

8.2 交换（swap）

8.2.1 例

当前 CPU 调度算法采用轮转法，当前时间片用完时需要将当前进程的交换到备份存储上，将另一进程放入内存执行，被换出的进程再次被执行时会再次调入内存。这种调出、调入内存也叫滚出、滚入。如下图所示：



8.2.2 概念

备份存储一般为快速磁盘。

就绪队列：在备份存储上或在内存中准备运行的进程的集合。

交换的限制因素：上下文切换时间较长；若要换出进程，需要保证该进程完全处于空闲状态（尤其是待处理的 I/O，若该进程有待处理的 I/O，可能导致换出后该 I/O 会使用已经属于新的进程的内存）

8.3 内存分配：

8.3.1 概念：

（1）内存主要分为两个区域：一部分用于驻留操作系统（通常位于低内存），另一部分用于用户进程

（2）碎片：

定义：无用的空闲内存空间

分类：内碎片和外碎片。内碎片存在于进程所在内存区域内部，是属于该进程的。外碎片不属于任何进程，在所有进程外。

降低外碎片的方法：

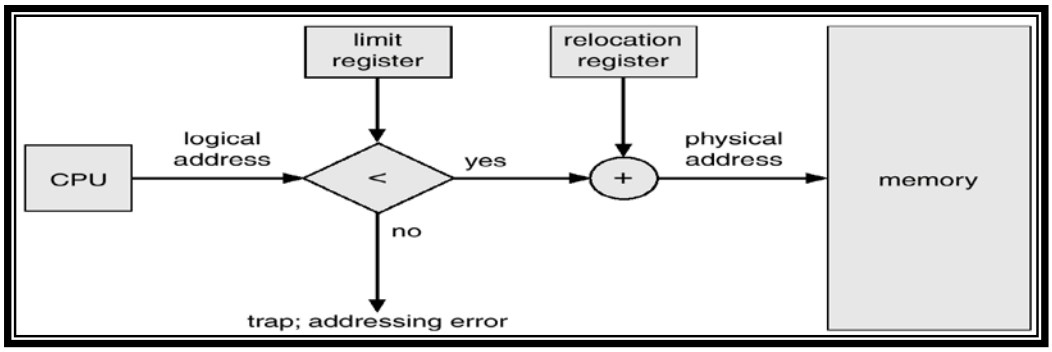
a.紧缩:移动内存内容，使空闲空间形成一整块。需要重定向是动态的并且在运行时采用。

b.允许物理地址空间不连续。

8.3.2 *连续内存分配（Contiguous Memory Allocation）

（1）特点：为每个调入内存的进程分配一段连续的内存区域。

（2）内存映射和保护方法：利用重定位寄存器和界限地址寄存器实现。



（3）内存分配：

a. 单分区：用户区中只有一个分区，同一时刻只能运行一个程序。

b. 多分区：将用户区分为多个分区，每个分区每一时刻只能运行一个程序，支持多道程序设计。

（4）分类：

a. 固定分区：

特点：用户区划分的分区个数固定、大小固定。

固定分区表：记录各分区的位置、大小、使用情况，可用于系统对分区进行管理。

碎片：会产生内碎片。

b.可变分区：

特点：初始化时只有一个分区，“按需分配”。

分区表：对已用分区进行管理。

空闲分区表：对空闲分区进行管理。

PS：孔（hole）：空闲分区

（5）分区分配算法：为进程在空闲分区表中寻找合适的分区，产生外碎片

a.首次适应：分配找到的第一个足够大的可用分区，寻找的起始位置可以是表头，也可以是上次首次适应结束时的位置。

b.最佳适应：分配足够大的里面最小的孔，可以产生最小的剩余孔。必须要遍历整个表，除非这个表按照大小排序过。

c.最差适应：分配最大的表，可以产生最大的剩余孔。也必须遍历整个表，除非这个表按大小排序过。

d.（补充）next-fit：从上次找到合适孔的位置继续往后找此次合适的孔。

（6）分区式内存管理特点：作业/进程放在一段连续的内存区域；管理简单；比较大的作业要找到足够大的内存区域比较困难。

8.3.3 *分页（paging）

引入：连续内存分配存在的碎片问题很严重，而且为较大作业分配足够大的内存区域比较困难（在对换时的交换区也有此问题），故引入分页。

（1）基本概念：

页（page）：将逻辑内存分成的固定大小的块

帧（frame）：将物理内存分成的固定大小的块。

页表：将页和帧完成映射（逻辑地址到物理地址的映射）。由于页号是从 0 开始按顺序排列的，所以页表中忽略页号，只留帧号。

逻辑地址格式：页号+页内偏移量

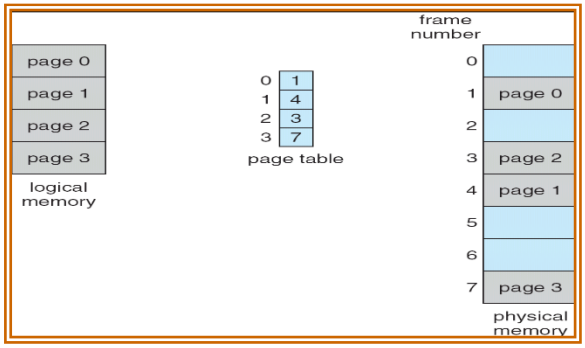
帧表（frame table）：内存分配的细节，如哪些帧已用，哪些帧可用，总帧数等。每个条目代表一个帧，表示该帧空闲或占有（被哪个进程的哪个页占用）。

（2）基本思想：一个作业在内存的各个页面可分配到不同帧中，但每个页面是连续的。用户视角的内存和实际的物理内存的分离。内存分帧，作业分页，帧的大小和页的大小相同。内存分配以页为单位。会产生内碎片。

（3）地址映射解析：

地址映射：

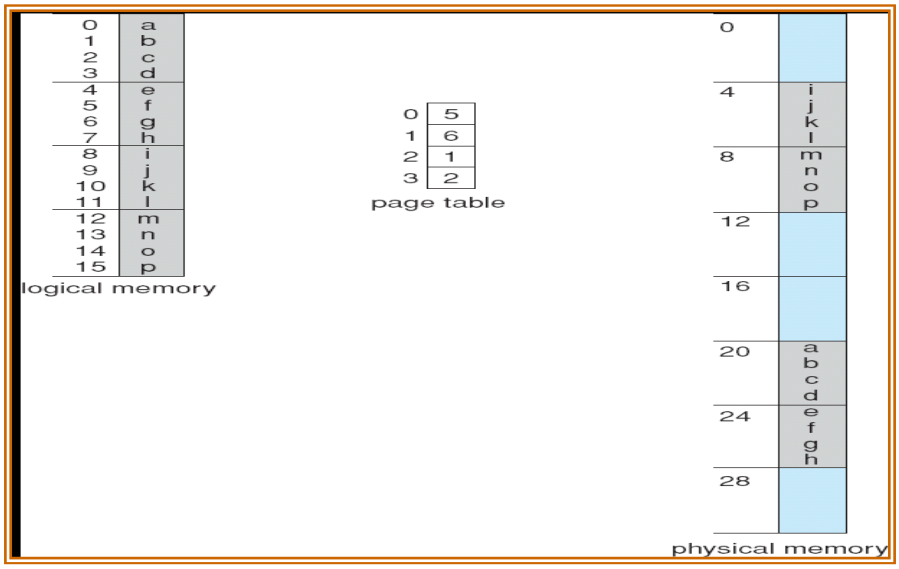
根据页表显示，逻辑地址空间
的 0 号页映射到 1 号帧，
1 号页映射到 4 号帧，
2 号页映射到 3 号帧，
3 号页映射到 7 号帧，



进一步看地址映射（页内偏移量）：

逻辑地址 2 即页号为 0，页内偏移量为 2，映射到 5 号帧内的偏移量为 2 的位置，即物理地址为（5*4+2）=22.

再如逻辑地址为 10,10/4=2……2，即页号为 2，页内偏移量为 2，映射到 1 号帧的偏移量为 2 的位置，物理地址为（1*4+2）=6

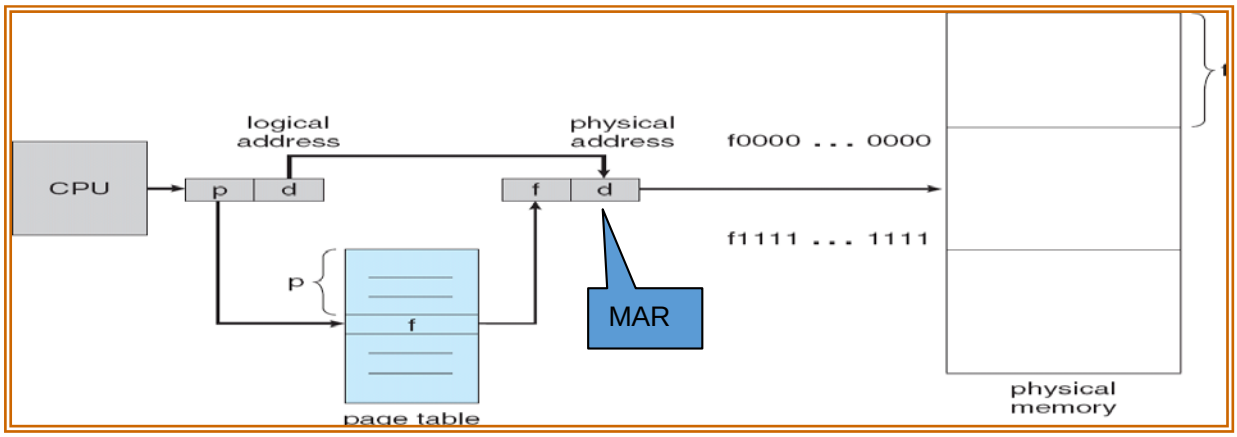


（4）地址变换过程如下图：

当进程要访问某个逻辑地址中的数据或指令时，MMU 中的分页地址变换机构会自动地将有效地址(相对地址)分为页号与页内地址两部分，再以页号为索引去检索页表。（查找操作由硬件执行）。

在执行检索之前，先将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间，产生地址越界。

若未出现越界错误，则将页表始址与页号和页表项长度的乘积相加，便得到该表项在页表中的位置，于是可从中得到该页的物理块号，将之装入 MAR 中的高位部分，再将逻辑地址中页内偏移量送入 MAR 的低位部分。这样便完成了从逻辑地址到物理地址的变换。（或者将帧号与页内偏移量拼接后形成物理地址，再送入 MAR）



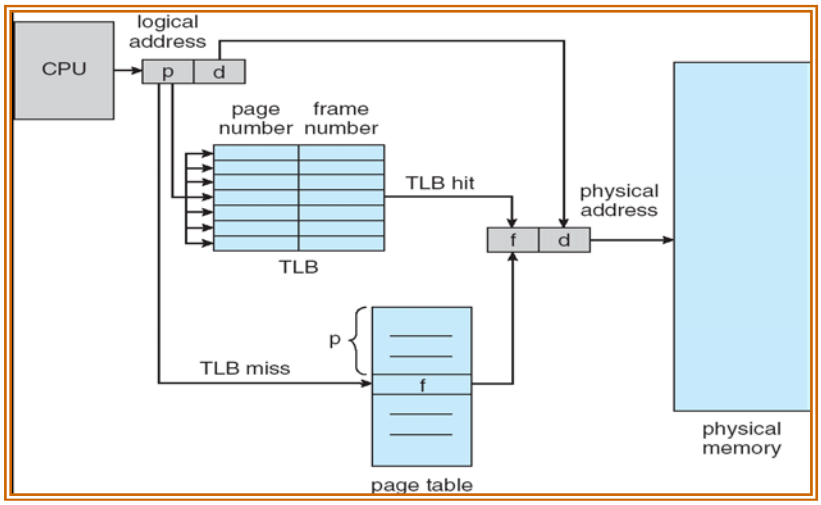
（5）硬件支持：

页表的硬件实现：

a、作为一组专用寄存器来实现：寄存器的访问速度快，所以地址变换的速度快。但是页表较大时成本太大。

b、将页表放入内存，使用页表基寄存器（**Page-table base register, PTBR**）指向页表，页表长度寄存器（**Page-table length register, PTLR**）记录页表长度 。进程未执行时页表的初始地址和长度保存在 PCB 中，执行时装入到相应寄存器中。此方法访问一个字节需要先从内存中访问页表得到物理地址，然后再次访问内存取得目的数据，一次需要两次访问内存，过于耗时。

c、使用转换表缓冲区（**translation look-aside buffers , TLB**）：类似于高速缓存，是一种快速内存，TLB 中只包含页表中的一部分条目，CPU 产生逻辑地址后，先在 TLB 中查找是否有相应页号，若找到则得到了对应帧号，再根据帧号访问内存即可。若未找到页号（称为 TLB 失效），则与 2 中查找方式相同，访问两次内存，并把对应页号和帧号放入 TLB。若 TLB 已满，则操作系统选择一个进行替换（有些 TLB 允许某些条目固定在其中，永远不被替换出去，例如一些内核代码的条目），带有 TLB 的分页如下图：



（6）内存保护：

a、设置 valid/invalid 标志位来区分有效/无效位。当系统调用一个进程时，将该进程 PCB 中的页表信息放入系统页表中，并设置相应的控制寄存器和标志位。

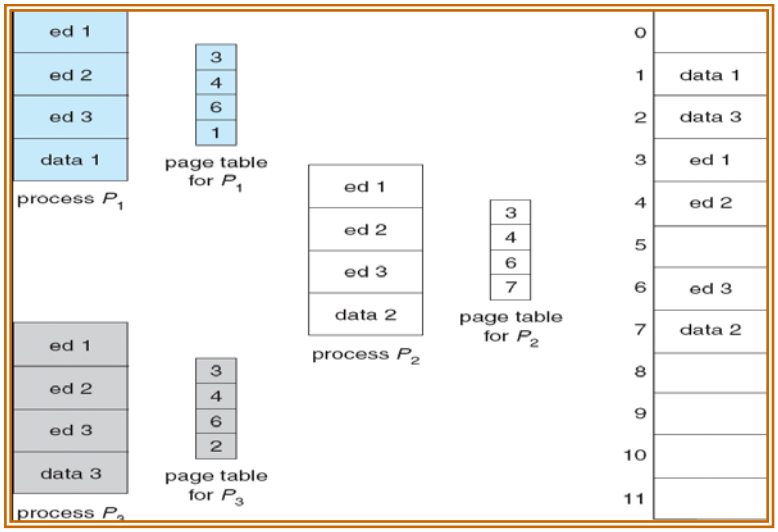
b、将逻辑地址的页号部分和页表长度比较，若比页表长度大则判断无效。

（7）共享页：

实质：利用页表指向同一部分帧实现共享。

条件：①代码是可重入代码（运行多次内容不变，也叫纯码）②共享的代码应该完全一样（在逻辑地址空间里面的地址也一样）。

例如下图中进程 P1，P2 共享 0,1,2 三个页面



8.3.4 *分段

引入：分页内存管理方案的缺点：物理内存与用户想象的内存不一致；内存共享时可能将一个模块分成了多个页面，分享困难。

（1）主体思想：

支持用户视角的内存管理方案。

作业分段，内存按动态分区管理。

内存分配以段为单位，每段都有段名和长度。

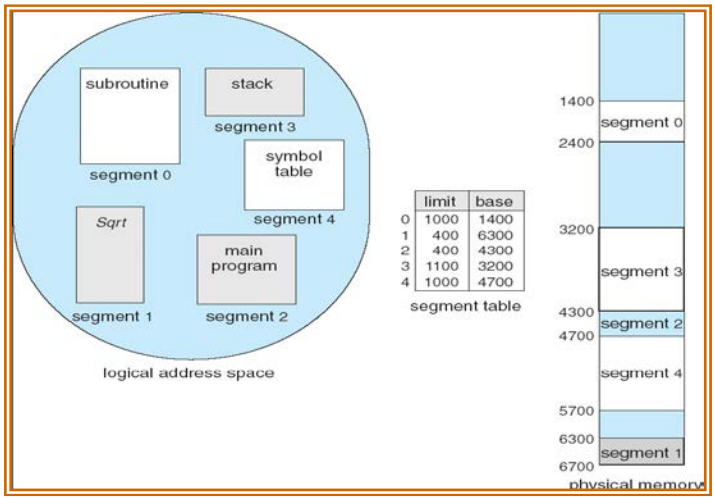
作业在内存中不连续，但是在段内连续。

逻辑地址格式：段号，段内偏移量。

段表描述段之间的内存位置关系，每个条目包含该段的起始位置和长度，段表基址寄存器（**Segment-table base register , STBR**）保存段表在内存中的地址，段表长度寄存器（**Segment-table length register , STLR**）保存一个程序使用的段的个数。

碎片：会产生外碎片。

（2）基本结构如下：

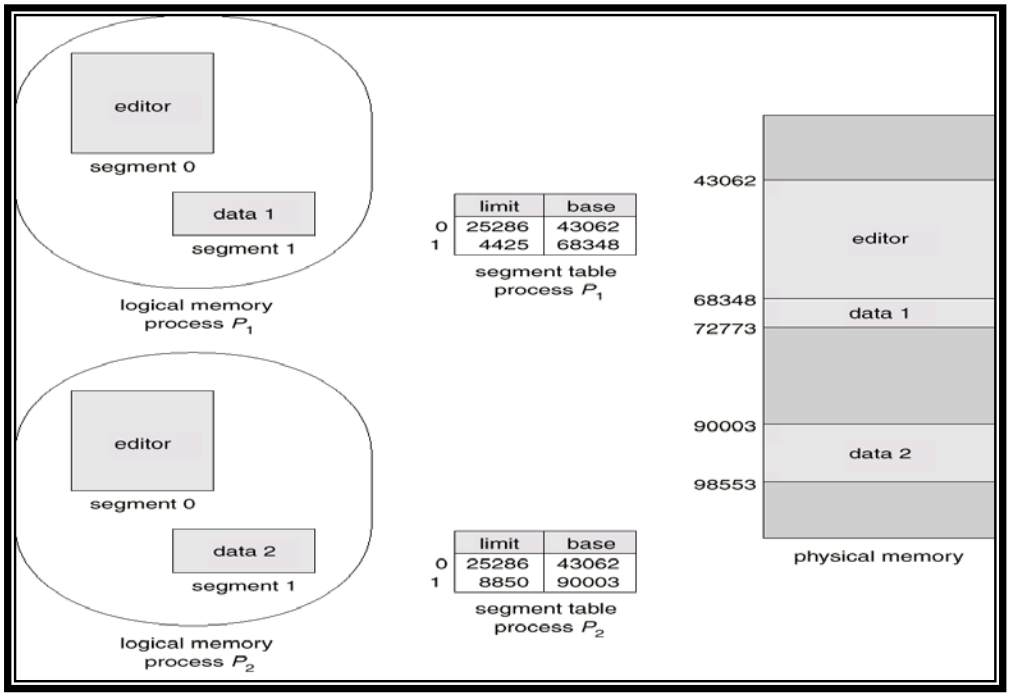


（3）内存保护的方法：①设置 valid/invalid 标志位②利用段表长度与段号进行比较判断段号是否有效，然后根据段表判断段内偏移量是否有效。

（4）共享：

条件：代码可重入；逻辑地址必须相同。

下图示例：进程 P1，P2 共享了 0 号段。



（5）分段和分页的区别：

① 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外碎片，提高内存的利用率；或者说，分页仅仅是由于系统管理的需要，而不是用户的需要；段是信息的逻辑单位，它包含有一组其意义完整的信息。分段的目的是为了能更好地满足用户的需要；

② 页的大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因为一个系统只有一种大小的页面；段的长度不固定，决定于用户所编写的程序，通常由编译程序对源程序进行编译时，根据信息的性质来划分；

③ 分页的作业地址空间是一维的，即单一的线性地址空间，程序员只须一个地址记忆符，即可表示一个地址；分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址；

4.分段与分页结合：

（1）主体思想：

作业先分段，段内分页，内存划分为与页相同大小的帧。

每个段都有一个段名；

内存分配以页为单位；

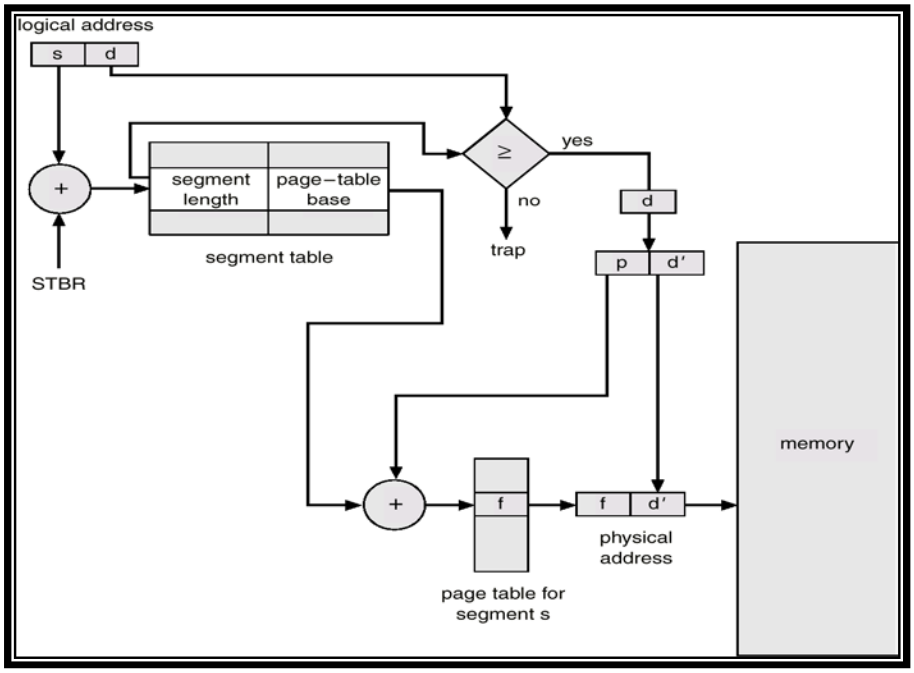
每个作业有一个段表，记录段在内存的起始地址以及段长；

每个段还对应一个页表；

逻辑地址的格式：段号 ，段内偏移量（页号，页内偏移量）

会产生内碎片

（2）地址转换如下图：



8.4 页表结构

8.4.1 层次页表（Hierarchical Paging）

引入：页表可能非常大，内存可能找不到如此大的连续空间，所以将页表划分成更小的部分进行存放，即将页表再分页。

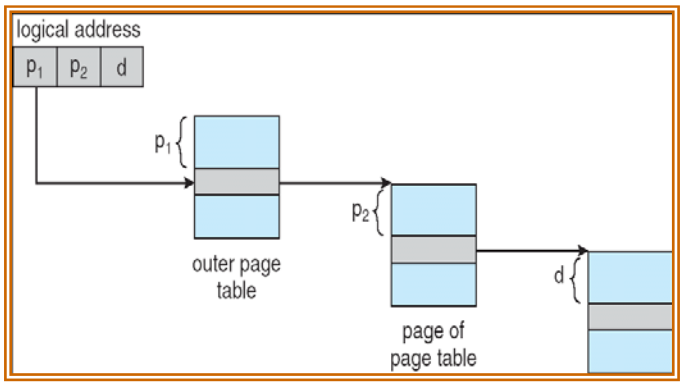
以二级分页算法为例：

以页面大小为 4KB 的 64 位系统：为例：若不采取层次页表，则页号部分占 20 位，页内偏移量占 12 位，这样的页表是超级大的，不方便存储。若采用层次页表，则：

假定页表项占 4 个字节，则由于每个帧大小为 4KB，则每个帧内页表项个数为 1K，即占 10 位，所以逻辑地址形式如下：

outer page	inner page	offset
p_1	p_2	d
10	10	12

地址转换过程如下：



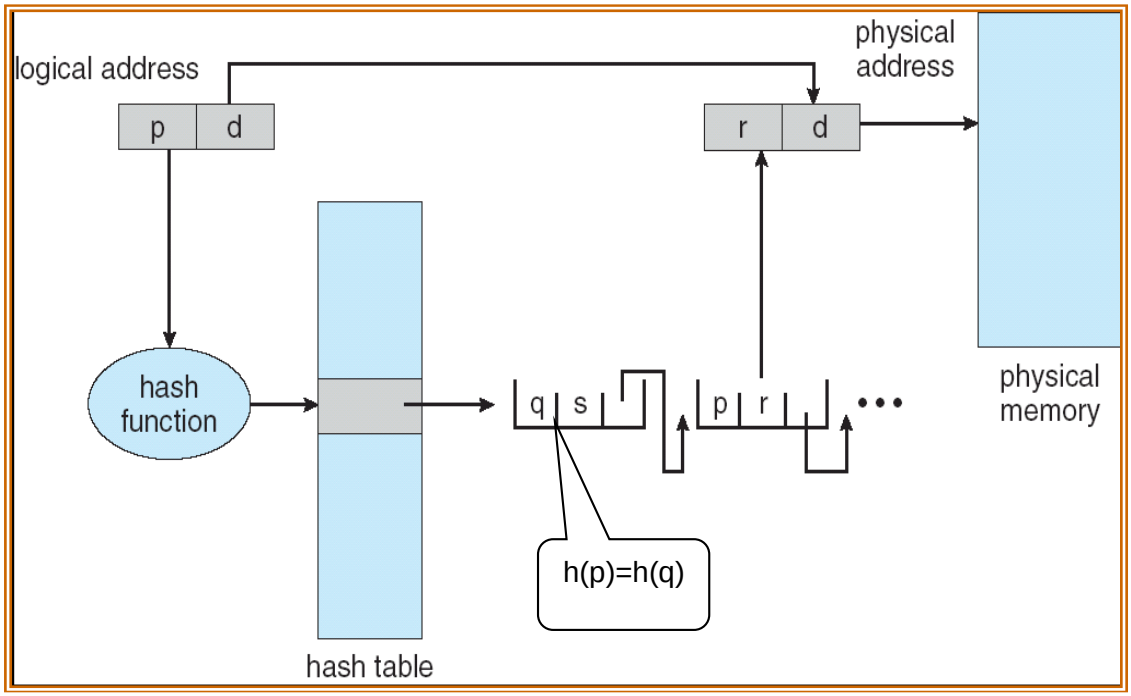
页面大小为 4KB 的 64 位系统，页表项占 4 个字节，分别采用采用二级、三级分页算法，逻辑地址形式如下：

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

8.4.2 哈希页表（Hashed Page Tables）

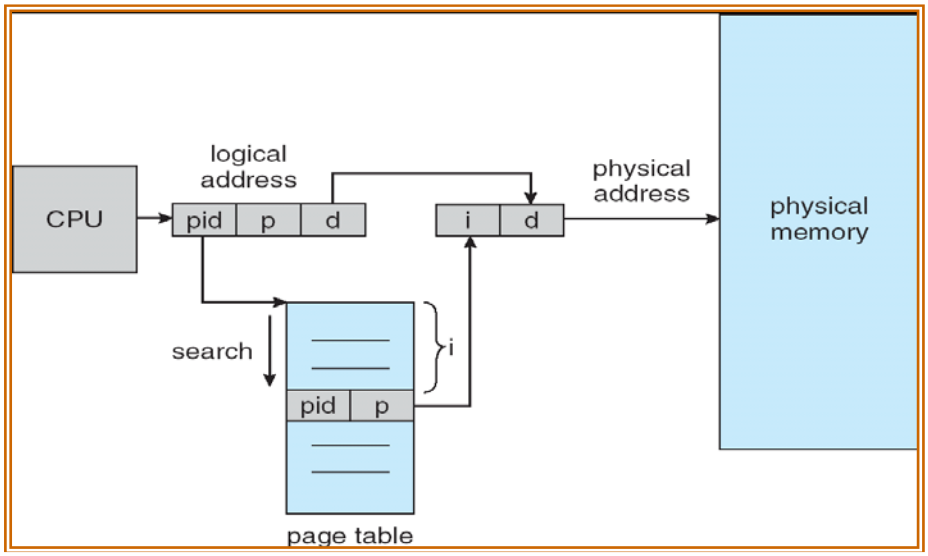
地址映射如下：虚拟地址的虚拟页号根据哈希函数找到哈希表的某一条目（哈希表的每一条目都存在一个链表元素，链表用于处理碰撞，链表的每个元素都包含三部分：虚拟页号，帧号，指向下一个元素的指针），将虚拟页号与链表中的每个元素的虚拟页号比较，找到匹配的元素即获得对应帧号，即可形成物理地址。



8.4.3 反向列表

逻辑页号来确定物理帧号为正向列表，而由物理帧确定对应的页号为反向列表。反向列表的每个条目包括对应页号以及拥有该页的进程信息。

地址形成过程如下：其中 pid 用来表示地址空间的标识符。



8.5 例题

8.5.0 重点概念

动态链接和静态链接；分段、分页管理的思想、地址转换过程、存储保护方法、内存共享。

8.5.1

有一采用分区存储管理的 OS，用户区主存在 512KB，空闲块链入空块表，分配时截取空块的前半部分（小地址部分）。初始时全部空闲。在执行了如下申请、释放操作序列后：

request(300kB),request(100kB),release(300KB),request(150KB),
request(50KB),request(90KB)

（1）、采用首次（最先）适配，空块表中有哪些空块（指出大小及始址）

（2）、采用最佳适配，空块表中有哪些空块（指出大小及始址）

（3）、若随后又要申请 80KB，针对上述两种情况，会产生什么后果？

参考答案：

（1）采用首次适应算法产生的空闲块：

块 1：首址 290KB，长度 10KB

块 2：首址 400KB，长度 112KB

（2）采用最佳适应算法产生的空闲块：

块 1：首址 240KB，长度 60KB

块 2：首址 450KB，长度 62KB

（3）随后又要申请 80KB，对于首次适应算法，分配成功；对于最佳适应算法，分配失败；

8.5.2

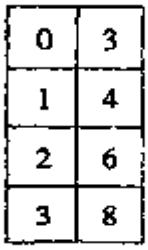
在某个采用页式存储管理的系统中，现有 J1，J2，J3 共 3 个作业同驻主存。其中 J2 有 4 个页面，被分别装入到主存的第 3，4，6，8 块中。假定页面和存储块的大小均为 1024 字节，主存容量为 10K 字节。

(1)写出 J2 的页面映像表；

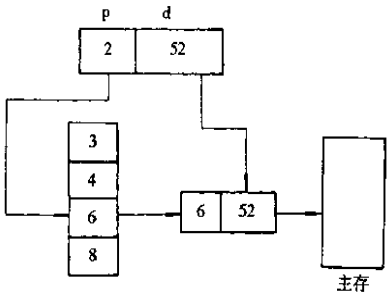
(2)当 J2 在 CPU 上运行时，执行到其地址空间第 500 号处遇到一条传指令 MOV 2100,3100

请用地址变换图计算出 MOV 指令中两个操作数的物理地址。

参考答案：该题已知条件很多，但实质还是给出逻辑地址，要求转换成物理地址。首先得出页表项如图 1 所示，页面大小为 1024 字节，可得页内位移为 10 位。2100/1024=2…52,故逻辑地址 2100 页号为 2，页内位移 52，同理，3100 页号为 3．页内位移 28。转换过程如图 2 所示，逻辑地址 3100 的转换过程同理。



（图 1）



（图 2）

3．考虑下面的段表：

段号	基地址	段长
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1592	96

计算下面的逻辑地址对应的物理地址：（逻辑地址的形式为（段号,段内偏移量））

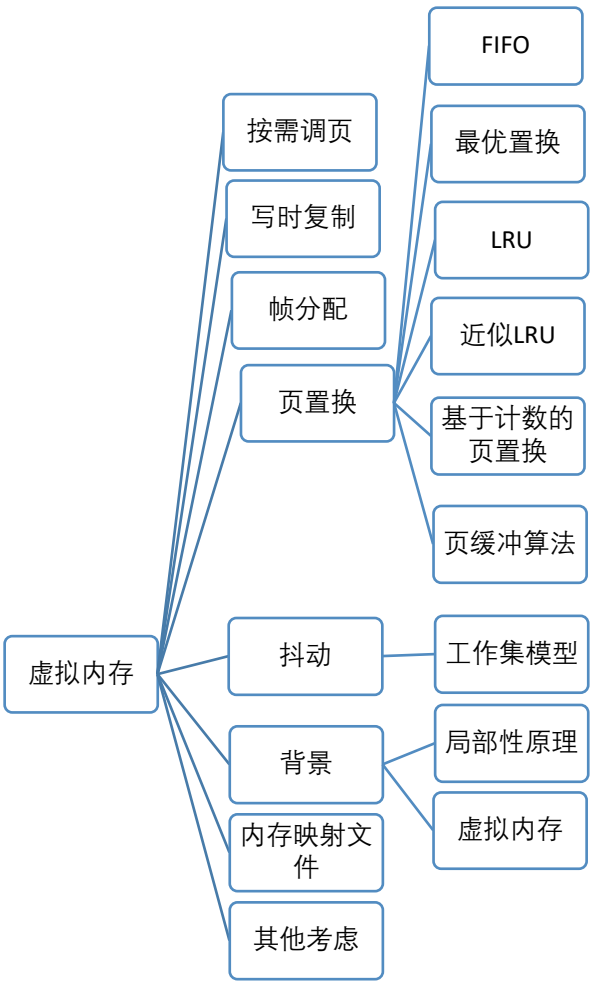
a. 0,430
b. 1,10
c. 2,500
d. 3,400
e. 4,112

参考答案：判定逻辑地址是否合法的顺序：段号是否合法，然后段内偏移量是否越界

a.物理地址 = 219+430=649
b.物理地址 = 2300+10=2310
c.因为段内偏移量 500>段长 100，地址越界；
d.物理地址 = 1327+400=1727
e.因为段内偏移量 112>段长 96，地址越界；

第九章 虚拟内存

作者：徐卫霞



9.1 背景

引言：之前学的分段、分页等内存管理方案都是在作业执行前就将作业全部装入内存，直到作业结束才释放，但是一些数据在作业运行时并不一定会用到（如异常处理）。之前的存储管理方法**浪费内存空间、降低了内存利用率、降低了系统并发度、减少了系统吞吐量、增加了 I/O 时间**。

9.1.1 局部性原理

时间局部性：若程序的某条指令被执行（或数据被访问），则不久后它可能被再次执行。

空间局部性：一旦程序访问了某个存储单元，那么它附近的存储单元也将被访问。（例如数组的顺序访问）

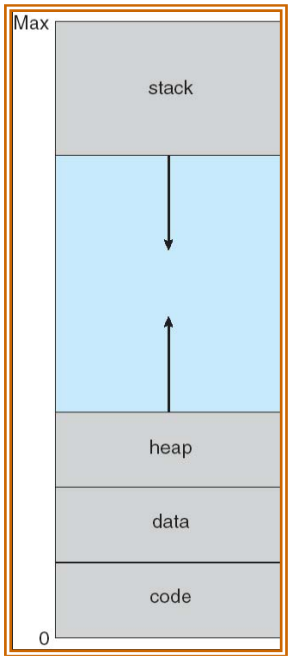
9.1.2 虚拟内存（虚拟存储器）

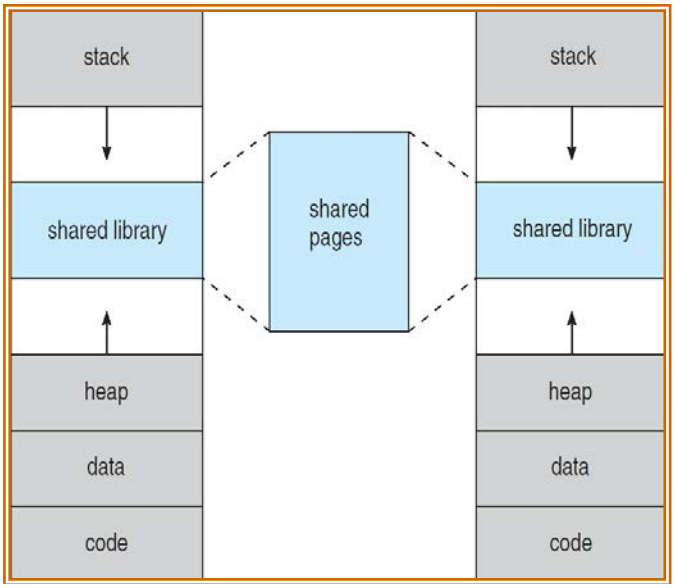
（1）定义：仅把作业的一部分装入内存即可执行，具有请求调入功能和置换功能，能在逻辑上对内存空间加以扩充的一种存储器系统。

（2）思想：根据局部性原理，我们只需要将当前要执行的那部分页或段先放入内存执行，其他先放在磁盘中。若程序想要访问的页（或段）未在内存中（称为缺页或缺段），操作系统调用请求调页（段）功能将所要访问的放入内存。若此时内存已满，则需要利用置换功能将内存中暂时不用的页（段）调出内存，将新的页（段）调入内存。

（3）优点：使一个大程序可以在小内存中运行；用户在一个虚拟内存中编程，使得编程不再受内存容量限制；内存可以装入更多程序并发执行；提高系统的并发度和吞吐量，减少了 I/O 时间。

（4）进程的虚拟内存空间是进程如何在内存中存放的逻辑视图，如左下图所示：





（5）共享库：通过将共享对象映射到一个虚拟地址空间，系统库可以被多个进程共享，如右上图所示。

9.2 按需调页（Demand paging，请求页式）

9.2.1 定义

调用程序时在需要时才调入相应的页，使用懒惰交换（只有在需要页时才调入）。

9.2.2 优点

I/O 时间低，所需内存减少，更快响应，并发度更高

9.2.3 硬件支持

（1）页表：需要设置 valid/invalid 位（标明该页是否在内存中）：

Valid：该页号有效在内存中

Invalid：该页号无效（不在该进程中）或有效但不在内存中。

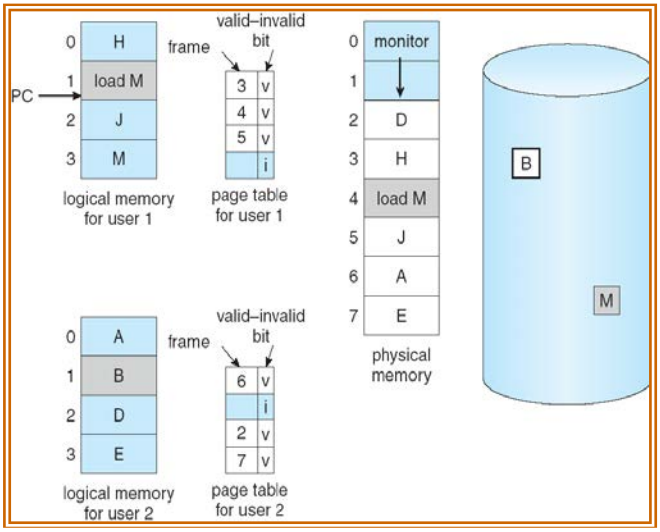
（2）次级存储器：用于保存不在内存中的页，通常为快速磁盘，常称为交换设备，用于交换的这部分磁盘叫交换空间（swap space）。

9.2.4 页面错误（page fault，也称页面失效或缺页中断）

（1）定义：所需页不在内存中（标志位为 invalid），引发页面错误。

（2）执行：页面错误发生后，系统对比进程页表和系统页表确定该页是否合法（是否在该进程内），若非法则中止，若合法则找到一个空闲帧，调用一个磁盘操作将所需页放入该空闲帧中并修改进程内部表和页表（其中标志位由 i 变为 v，表示该页已在内存），重启当前指令。

例：下图所示：需求页为 3 号页，若内存未满，则找到一个空闲帧放入所需页。但是此处内存已满，故需要替换出一个页，将 3 号页放入空闲帧（需要进行页面置换）。

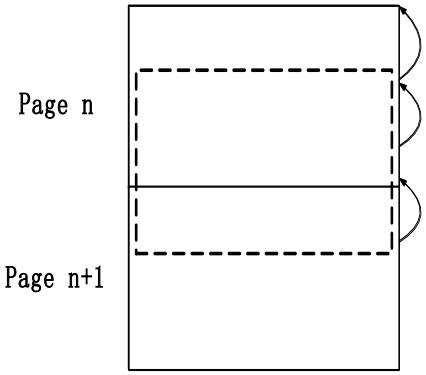


（3）缺页中断和一般中断的区别：普通中断是在 CPU 完成对该指令的执行后检查和处理中断，而缺页中断是在指令执行期间，发现所需指令或数据不在内存时产生和处理的。

（4）重启指令有时并不可行，例如：相互重叠的区域中进行块移动指令

该指令将虚框内的数据移动到 Page n 中，由于虚框跨越两个页面，若指令开始执行时第 n+1 页不在内存，当将虚框内第 n+1 页中的数据移动到第 n 页时，产生缺页

中断；当将第 n+1 页装入到主存后，虚框中第 n 页的内容已被修改，重启指令重新进行传送，将导致错误；（复制的是已经修改过的数据）如下图所示：



9.2.5 纯粹按需调页（pure demand paging）

所有页都不在内存中就开始执行该进程，会一直产生缺页中断直到所需页均在内存中。

9.2.6 性能

缺页中断率 p 越接近 0，性能越高。

有效访问时间（Effective Access Time，EAT）=（1-p）*内存访问时间+p*平均页错误处理时间。

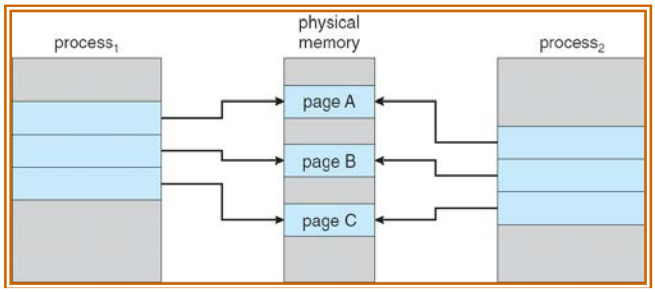
页错误处理时间包括处理页错误中断时间，页调入、调出时间，重启进程时间等。

9.3 写时复制（Copy-on-Write）

9.3.1 定义

允许父子进程在初始时共享同一页面，当任何一个进程对页进行写操作时创建一个共享页的副本。

下图示例：



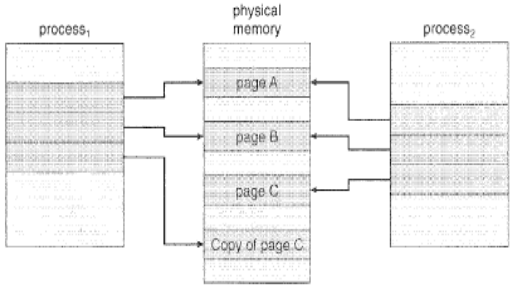


Figure 9.8 After process 1 modifies page C.

9.3.2 分配空闲帧

操作系统提供空闲缓冲池，当需要复制时从中取出一个空闲帧进行写入。操作系统采用按需填零（空闲帧在被分配前先填零来消除之前的内容）技术分配这些空闲帧。

9.4 *页面置换（Page Replacement）

9.4.1 定义

内存已满，当前需求的页无法放入内存，需要将内存里的某个页置换出去形成空闲帧才能放入。

9.4.2 基本页置换

（1）基本页置换下的页错误处理程序如下：

查找所需页在磁盘中的位置；查找空闲帧，若找到则使用，若未找到，则使用页置换算法来选择一个“牺牲”帧（victim frame），将“牺牲”帧的内容写入磁盘，改变帧表和页表；将所需页写入空闲帧，改变页表和帧表；重启用户进程。

（2）优化：加入修改位（modify bit）或脏位（dirty bit）：当无空闲帧时页错误处理程序需要两次页传输，加入修改位后，当某页被修改后该页的修改位被修改，若该页被替换则需要回写到磁盘，两次页传输。若未改写则不需回写，所需页直接覆盖原数据即可，这样就只需一次页传输，降低了传输时间。

（3）要实现按需调页，必须开发帧分配算法和页置换算法。

（4）最小页错误率算法：用于评价页置换算法（帧分配算法也适用）

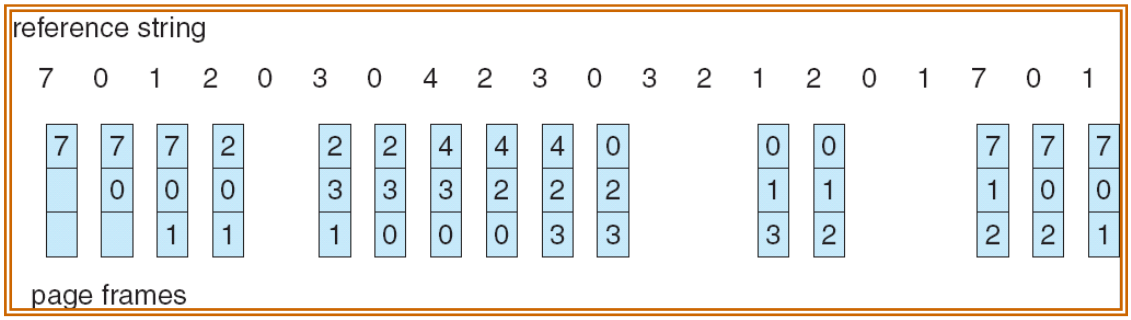
思想：根据特定内存引用序列，执行某个置换算法，得到页错误率。内存引用序列叫做引用串。

9.4.3 FIFO 页置换（FIFO Page Replacement）

（1）思想：最先到达的页会被首先作为“牺牲”页。。

（2）实现：系统维护一个按照为页面分配物理帧的顺序排序的先进先出队列，后分配的总放在队尾，最先置换队首页。

（3）示例：



页面错误率=15/20=75%

（4）评价：

优点：容易理解和实现

缺点：性能不是很好，可能置换出去的是很久之前初始化并且一直在用的变量。存在 Bélády 异常。

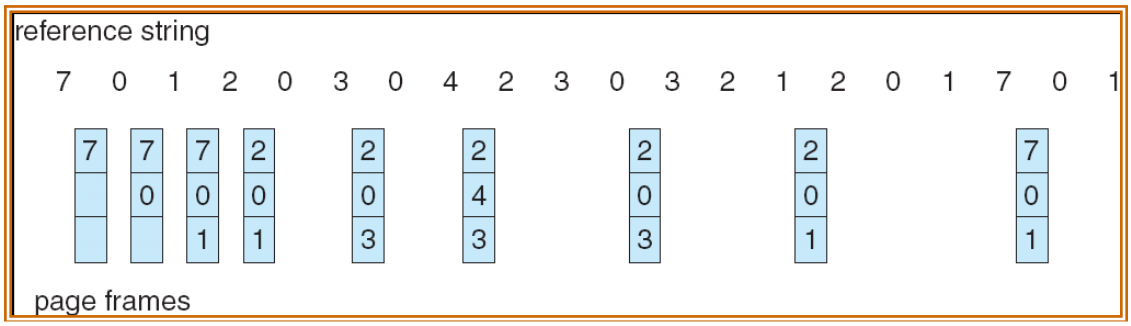
Belady 异常：引用串为（1,2,3,4,1,2,5,1,2,3,4,5）时给进程分配 4 个帧产生的页面错误率比分配 3 个帧还多。

9.4.4 最优置换（Optimal Algorithm, OPT）

（1）思想：进行页置换时选择未来最长时间不会用到的页作为“牺牲”页（选择“牺牲”页时“往将来看”）。

（2）实现：因为很难知道未来会用到哪些页，所以难以实现。

（3）示例：



页面错误数为 9，页面错误率为 9/20=45%；

（4）评价：

优点：顾名思义，它是最优的页置换算法，最小的页面错误率。

缺点：由于需要知道未来的情况，所以难以实现。这个算法一般用来作为对其他算法的评价。

9.4.5 LRU 置换（Least-Recently-Used）

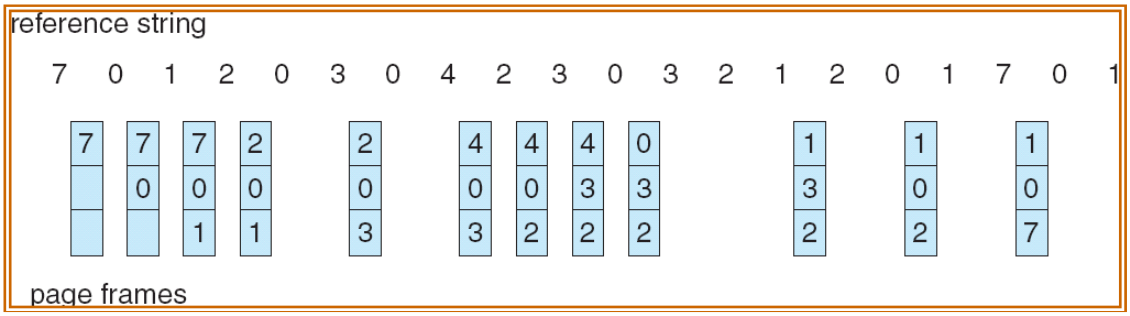
（1）思想：根据时间局部性原则，最近访问到的页面近期也会被访问，而最近未被访问的页面近期也不会被访问，所以可以优先置换掉最近最少使用的页（“往之前看”）。

（2）实现：

①计数器：为每个页表关联一个时间域，为 CPU 增加一个逻辑时钟或计数器，每次内存引用计数器加一，时钟寄存器内容复制到相应页所对应页表项的使用时间域中。每次置换使用时间最小的页（最早被使用的页）。此方法需要搜索页表查找时间最小的页，并动态维护页表的使用时间域，必须考虑时钟溢出。

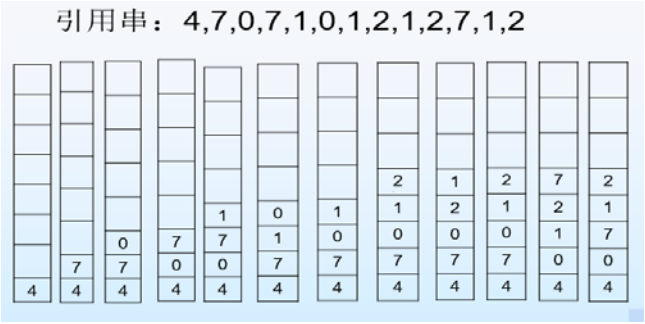
②使用页码栈，每次引用一个页就把相应的页放入栈顶，在栈底选择“牺牲”页。可使用双向链表实现。

（3）示例：



页面错误数为 12，页面错误率为 12/20=60%

用页码栈实现的的 LRU 示例：



（4）评价：

优点：和最优置换一样，都没有 Belady 异常（它们属于同一种算法，叫做栈算法）。效率较高，是一种经常被使用的页置换算法。

缺点：需要硬件支持（计数器、时钟等）

9.4.6 近似 LRU 置换

很少有系统可以提供足够的硬件来支持真正的 LRU 置换，当时 LRU 总体效率较高，所以可以利用近似 LRU 置换。

（1）思想：给页表的每个页表项增加引用位，每次引用该页时引用位被置数。根据引用位来进一步选择“牺牲”页。

（2）分类：

①附加引用位算法：为每个页保留一个 8 位的字节（相当于 8 个引用位），规定时间间隔内时钟定时器产生中断，将该时间间隔内每页的引用位复制到它们所对应的 8 位字节的高位，而将其他位引用右移，最低位抛弃。这样 8 位字节内保存的是该页在 8 个时间周期内的引用情况。例如若为 00000000，则该页在 8 个时间周期内都未被引用，若为 11111111，这说明该页在这 8 个时间周期内都至少被引用过一次。把它们当做无符号数来判断的话，则数值越小说明近期越最少使用，选择“牺牲”页时选择数值最小的。也可以利用 FIFO 的原则来进行选择“牺牲”页。

②二次机会算法：基础算法是 FIFO，且只有一个引用位。要选择“牺牲”页时，检测引用位，若为 1，则将其置为 0，并将它的到达时间置为当前，若为 0 则置换出去。

实现方法：利用循环队列，指针指向下次要置换的页。当要寻找“牺牲”页时，指针向前移动时它将引用位为 1 的置为 0，直到找到引用位为 0 的页将其置换出去，新页插入到该位置。

③增强二次机会算法：增加引用位和修改位并将其当做有序对来考虑。因此有四种情况：（引用位，修改位）

(0,0) -最佳的置换页

(0,1) –该页被修改过，所以若要置换掉它需要将其回写。

(1,0) –可能会被再次引用

(1,1) - 可能会被再次引用，若被置换需要将其回写。

置换方法：可使用时钟算法，检查所指页属于哪种类型，置换最低级别的页。与时钟算法主要区别在于给已经修改过的页更高的优先级，从而降低所需 I/O 数量。

9.4.7 基于计数的页置换

（1）思想：给每个页保留一个用于记录其引用次数的计数器。

（2）分类：

①最不经常使用页置换算法（**(Least Frequently Used**）：

思想：计数小的是访问次数少的，应该予以淘汰，故置换掉计数最小的页。

问题：一个页在进程开始时使用很多，但之后不再使用，但由于开始使用次数很多导致计数很大，故之后也会在内存中。

优化：定时对计数器右移一位，形成指数衰减的平均使用次。

②最常使用页置换算法（**Most Frequently Used**）：

思想：访问次数少的可能是刚装入内存的，而访问次数多的可能装入内存时间较长，应该予以淘汰，故置换掉计数最大的页。

9.4.8 页缓冲算法

（1）思想：系统维护一个空闲帧缓冲池，当有页面错误时所需页直接从池中选择一个帧来写入，使得进程尽可能快的被重启，而此算法也会选择“牺牲”页，将该帧回写后放入池中备用。

（2）扩展：

①当调页设备空闲时选择一个被改写过的页写回到磁盘上，并更新其修改位。

②保留一个空闲帧池，但是这些帧所对应的页号要记住，由于这些帧的内容未被修改，所以当再次需要这些页时可以直接从池中取出包含所需页的帧。

9.5 帧分配

帧的最少数量：分配给进程的帧的最少数量是由体系结构决定的，最大数量是由物理内存的数量决定的。

帧的分配算法：

（1）平均分配：若将 n 个帧分配给 m 个进程，每个进程分配到 n/m（向下取整）个帧，例如 93 个帧分配给 5 个进程，每个进程分得 18 个帧，剩余 3 个放入空闲帧缓冲池中。

（2）比例分配：按照进程的虚拟内存大小或优先级大小或两者结合的标准按比例分配帧。

（3）全局置换和局部置换

①全局置换：

特点：进程可从所有帧中选择一个置换帧，不管这个帧是否已分配给了别的进程。

缺点：进程无法控制其页错误率。

优点：进程可以利用其它进程不常用的内存，所以系统吞吐量更大，更为常用。

②局部置换：进程只能从分配给自己的帧中选择置换帧。

（4）优先级分配：当一个进程出现页面错误时：

①选择一个已分配给自己的帧作为置换帧

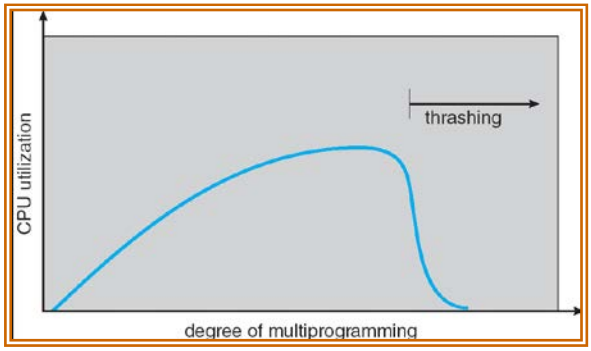
②从优先级比它低的进程中选择置换帧。

9.6 抖动（Thrashing，颠簸）

定义：进程没有“足够”的帧，需要进行页调度，而调度出去的页再次被需要，所以又很快被调回来，导致进程用于页调度的时间多于执行的时间。

9.6.1 系统抖动的原因

进程没有“足够”的帧，它会不断地进行页调度，导致 CPU 的利用率不高，而系统检测到 CPU 利用率不高就会加入新的进程执行，新的进程也需要帧，所以它也会不断地进行页调度从而拉低 CPU 的利用率，形成一个恶性循环。如下图：



9.6.2 解决方案

（1）采用局部置换或优先级置换算法：

思想：利用局部置换算法可以保证一个进程抖动时不会从其他进程所占用的帧中选择置换帧，也就不会干扰其他进程导致其他进程跟着抖动。优先级置换算法保证进程只能从比他优先级低的进程中选择置换帧，减少了从其他进程中调度帧，减少了对其他进程的干扰。

缺点：若进程抖动，则它大部分时间用于等待调度设备，这会使得调度设备的等待队列边长。相应的，系统的页面错误处理时间也会变长，所以其他进程的有效访问时间也会变长。

（2）工作集模型

①原理：

局部：一个经常使用页的集合，一个进程由多个不同的局部构成，局部可能交叠。

局部模型：进程由多个局部构成，进程执行时从一个局部移到另一个局部

原理：为了防止抖动，需要提供给进程所需的足够多的帧。对于正在运行的进程，如果分配给它的帧不足以容纳它的局部时会出现抖动。所以可以根据它的局部所占内存大小分配帧。

例如一个循环对数组中的数据进行处理，程序包括三个页面，数据包括四个页面，共七个页面构成一个局部。若分配给该局部的帧少于 7 个会出现抖动。故：**分配给每个进程的帧数应不少于该进程的当前局部**

②思想：基于局部性原理，每个进程近期使用的帧数作为将要使用的帧数的近似值。

③概念：

工作集窗口：用Δ表示，是固定数目的页引用集合，例如 100 个页引用。

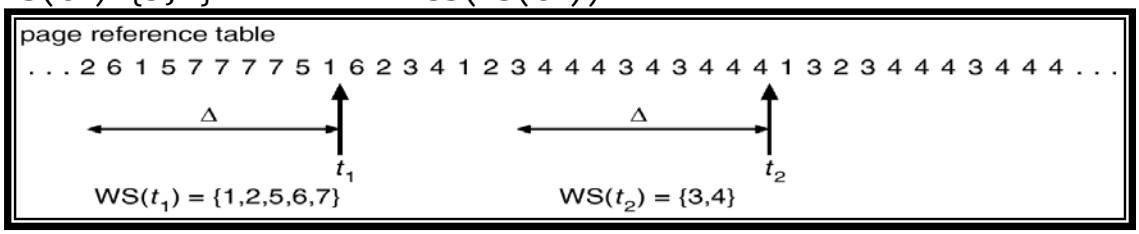
工作集合：最近Δ个引用的页集合，用 WS 表示。若一个页正在使用中，那么它就在工作集合内。工作集合是程序局部的近似。工作集合的精度和Δ的大小选择有关，若Δ很小则无法涵盖整个局部，若很大则可能涵盖多个局部。

WSS：表示该工作集所包含的页的数目**.** 因为工作集是程序的一个局部的近似，因此工作集中包含的页面数 WSS 是一个局部所需要的页框数，也可以认为是一个程序在近段时间内最少需要的页框数（帧数）。如果一个进程的可用页框数少于 WSS，也就是说分配给该进程的页框数不能包含进程的某个局部，就会引起颠簸（或抖动）。假定系统中每个进程的工作集所包含页的数目为 WSSi，则 D=ΣWSSi（D 为总的帧需求量），若 D 大于可用帧数，则会出现抖动。

例：下图中

WS(t1)={1,2,5,6,7}　　WSS(WS(t1))=5

WS(t2)={3,4}　　WSS(WS(t2))=2



④应用：

操作系统跟踪每个进程的工作集合，并为进程分配大于其工作集合的帧数，若还有空闲帧，则启动另一进程；若所有 WSSi 总和大于可用帧数，则选择暂定一个进程，释放其所占内存给其他进程。

⑤评价：

优点：防止了抖动，尽可能的提高了多道程序度，优化了 CPU 使用率。

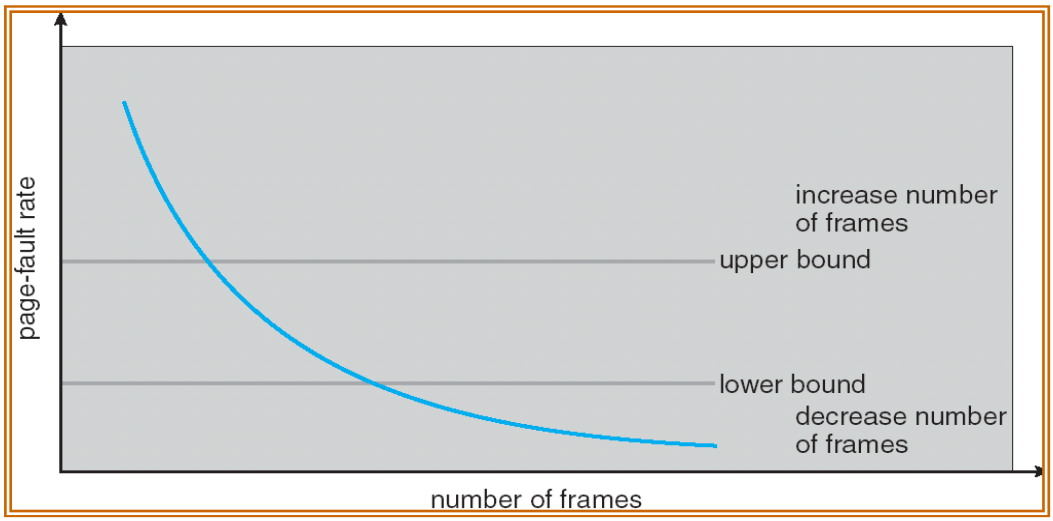
缺点：工作集合是移动窗口，难以跟踪。

⑥实现：通过固定定时中断和引用位，可以近似模拟工作集合模型。

（3）页错误频率策略（**Page-Fault Frequency Scheme**）

思想：为期望的页错误频率设置上限、下限，当进程的页错误频率大于上限时需要给该进程分配更多的帧，而当进程的页错误频率小于下限时从该进程中移走帧。其

中当页错误率增加而无可用帧时可能需要暂停一个进程，并将其所占资源释放分配给其他高页错误率的进程。如图：



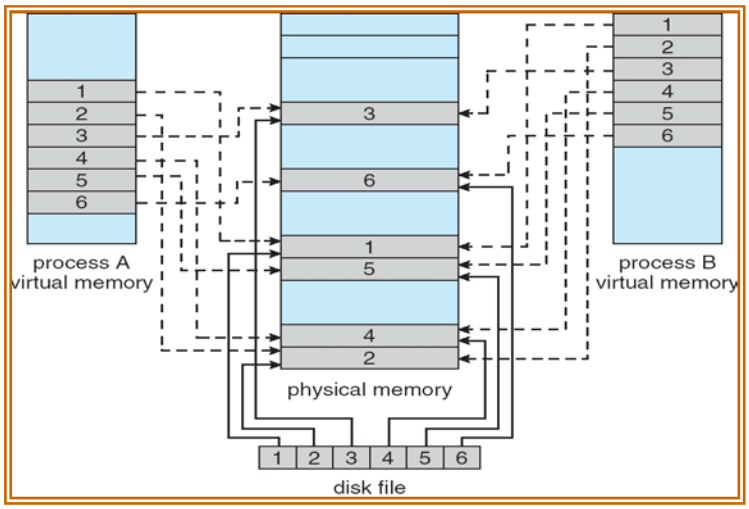
9.7 内存映射文件（Memory-Mapped Files）

9.7.1 基本机制

定义：利用虚拟内存技术将文件 I/O 当做普通的内存访问来对待，称为文件的内存映射。

思想：文件的内存映射可将一块磁盘块映射到内存的一页（或多页）。开始的文件访问按普通的请求页式来进行，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。之后的文件读写按照普通内存访问执行。对映射到内存中的文件进行写可能不会被立即写回到磁盘上的文件中，有些操作系统定期建厂文件的内存映射页是否改变以选择是否更新到物理文件。文件的关闭会导致内存映射的数据回写到磁盘，并从进程的虚拟内存中删除。

共享：多个进程允许将同一个文件映射到各自的虚拟内存中，以允许数据共享。如下图：



9.7.2 内存映射 I/O（Memory-Mapped I/O）

每个 I/O 控制器包括存放命令及传送数据的寄存器，通常，专用 I/O 指令允许寄存器和系统内存之间进行数据传递。

内存映射 I/O：一组内存地址专门映射到设备寄存器，对这些内存地址的读写如同对设备寄存器的读写。

9.8 其他考虑

9.8.1 预调页

纯按需调页系统的显著特性是当进程开始时会出现大量页错误，预调页目的在于阻止这种大量的初始调页，采用同时将所需的所有页一起调入内存。类似于指令预取，在进程引用前将需要的所有或部分页准备好。例如总线空闲时将执行页面之后的页面装入内存（局部性原理）。

9.8.2 页大小

页大小的确定需要考虑以下四点：

（1）页表：对于给定虚拟内存，页越大，相应的页表和页的数量越小，所以较大的页比较理想。

（2）内存利用率：较小的页可以更好地利用内存。

（3）页读写所需时间：为最小化 I/O 时间，需要较大页。

（4）页错误：为降低页错误数量，需要较大页。

9.8.3 TLB 范围

定义：TLB 范围指通过 TLB 可以访问的内存量，等于 TLB 的条目与页大小的乘积。

增加 TLB 范围（提高命中率）的方法：

① 增加页的大小或提供多种页大小：增加页的大小可能会有碎片产生；提供多种页的支持，需要操作系统而不是硬件来管理 TLB，较为常用。

② 增加 TLB 的条目数：价格昂贵，而且可能也不足以满足某些大型程序的需要。

9.8.4 反向列表

为帧创建的页表（8.5.3 详解）

9.9 例题解析

重点概念

虚拟内存；请求页式（缺页中断）、页置换、抖动

9.9.1

假定有一个按需调页存储器，页表放在寄存器中。处理一个页错误，当有空的帧可用或者被置换的帧没被修改过时要用 8ms，当被置换的帧被修改过时用 20ms，存储器读取时间 100ns，假定被置换的页中有 70%被修改过，则有效访问时间不超过 200ns 时最大可以接受的页错误率是多少？

参考答案：0.2 ns = (1-P)×100ns+(0.3P)×8 ms+ (0.7P)×20ms

P =0.000006

9.9.2

引用串为（1，2，3，4，2，1，5，6，2，1，2，3，7，6，3，2，1，2，3，6），当内存块数量分别为 3 时，试问 FIFO、LRU 这两种置换算法的缺页次数各是多少？（所有内存开始时都是空的，凡第一次用到的页面都产生一次缺页中断）

参考答案：

所有内存块最初都是空的，所以第一次用到的页面都产生一次缺页。
当内存块数量为 3 时：

FIFO

	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
	1	1	1	4	4	4	6	6	6	3	3	3	2	2	2	6				
		2	2	2	1	1	1	2	2	2	7	7	7	1	1	1				
			3	3	3	5	5	5	1	1	1	6	6	6	3	3				

发生缺页中断的次数为 16。

LRU

	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
	1	1	1	4	4	5	5	5	1	1	7	7	2	2	2					
		2	2	2	2	2	6	6	6	3	3	3	3	3	3					
			3	3	1	1	1	2	2	2	2	6	6	1	6					

发生缺页中断的次数为 15。

9.9.3

假设一个具有以下时间利用率的按需调页系统：

CPU 利用率　20%
分页磁盘　97.7%
其他 I/O 设备　5%

试说明下面哪一项可能提高 CPU 利用率，为什么？

a.安装更快的 CPU
b.安装更大的分页磁盘
c.提高多道程序度
d.安装更多内存
e.加入预约式页面调度算法预取页
f.增加页面大小
g.安装更快的硬盘，或对多个硬盘用多个控制器

参考答案：

该系统 CPU 利用率较低，而分页磁盘利用率极高，说明出现了抖动。要提高 CPU 利用率，就需要消除抖动，而抖动是因为进程没有“足够”的帧。所以 a.安装更快的 CPU、b. 安装更大的分页磁盘是不可能提高 CPU 利用率的。

c. 提高多道程序度会使得更多进程在系统中争夺帧，所以也不可以，需要降低多道程序度。

d. 安装更多内存可以使得更多页可以驻留内存，减缓抖动，从而提高 CPU 利用率。

e. CPU 可以更快的获得数据，所以提高了 CPU 利用率。但是只有页面调度适用于预取时（即某些访问是顺序的）才会提高利用率。

f．如果按顺序访问数据，增加页面大小将导致更少的页面错误。 如果数据访问或多或少是随机的，则可能会发生更多的页面调度操作，因为较少的页面可以保存在内存中，并且每页错误传输更多的数据。 所以这种变化可能会降低利用率，也可能增加。

g．安装更快硬盘或者对多个硬盘用多个控制其可以提高响应速度和磁盘吞吐量，CPU 会更快的得到数据，提高了 CPU 利用率。

第四部分 存储管理（Storage Management）

存储管理这一部分主要涉及了文件系统和 IO 系统的两部分概念，从第 10 章的文件系统的接口对外（用户和程序人员）表现的描述，到第 11 章操作系统内部如何从数据结构和算法方面实现文件系统，再到第 12 章磁盘从硬件上如何实现文件的物理存储和调度，这一章总的看脉络很清晰，但是如果学习时陷于一章就会云里雾里，所以要学会打通这几章的知识。这一部分的重点在于对文件系统的理解和对 IO 系统概念的把握。

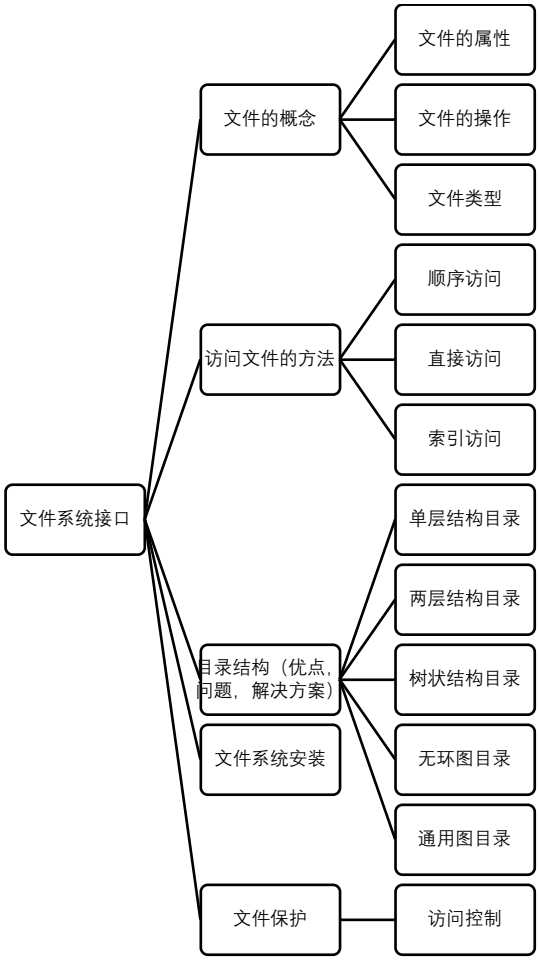
第十章 文件系统接口（File System Interface）

作者：孙吉鹏

冯诺依曼计算机体系结构里计算机需要有存储器，我们大量的信息需要让计算机“长久记住”以便使用。存在磁盘上不同用途的信息那么多，不分开根本搞不懂谁是谁，所以需要区分，把每个用途的信息需要单独管理起来，这种抽象集合就是文件。因为外存访问太慢了而且文件又那么多，所以我们需要建目录来管理。

为了便于理解，部分实现层的概念放在第 11 章讲。

10.1 知识框架



10.2 内容概述

10.2.1 文件的概念

文件是记录在外存相关信息的具有名称的集合，从用户看，文件是逻辑外存的最小分配单元，即数据除非在文件中，否则不能写到外存。

(1) 文件属性

名称，标识符，类型，位置，保护（访问权限），时间，日期等等。

(2) 文件操作

创建，读，写，删除，文件内重定位，截短（删除内容保留属性）

(3) 文件类型

实现技术就是拓展名

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

文件类型的作用？

1. 便于让操作系统知道是否识别支持当前类型
2. 通过文件类型确定哪种程序可以操作当前类型文件
3. 确定文件的内部组织结构

10.2.2 文件的访问方法

(1) 顺序访问

基于文件的磁带模型，文件信息按顺序，一个记录一个记录的处理，如编辑器和编译器。

(2) 直接访问

基于文件的磁盘模型，文件由固定长度的逻辑记录构成，读写顺序没有限制，如对表文件的查询。

(3) 索引访问

先搜索文件索引，再直接访问文件。

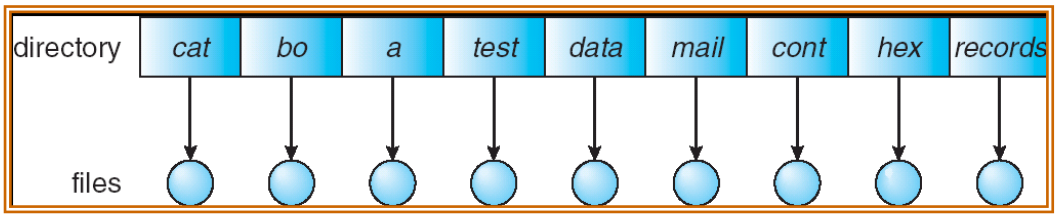
10.2.3 目录结构

文件系统实现对文件的“按名存取”

文件系统需要建立这样一种数据结构，以实现文件名与文件物理位置之间的映射关系，体现这种对应关系的数据结构称为文件目录。目录的每个叶节点保证通过文件名检索到，又包含这个文件物理位置的信息。

(1) 单层结构目录

所有文件都包含在同一层目录中。



优点：

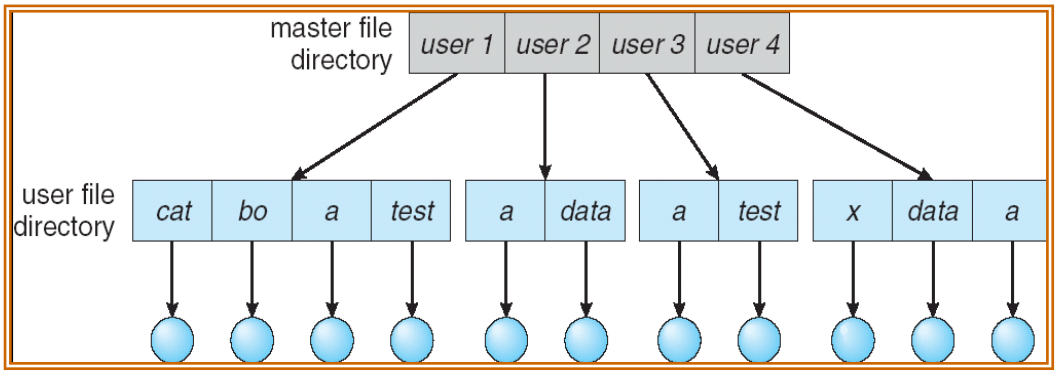
便于理解，便于支持，搜索高效

问题：

命名重复问题，用户分组问题，不同用户间文件共享问题

(2) 双层结构目录

为每个用户建立独立目录，避免不同用户间命名重复的问题。第一层目录叫主文件目录（Master File Directory，MFD），第二层用户的目录叫用户文件目录（User File Directory，UFD），Windows 系列采用的就是**支持扩展**的双层目录结构。



优点：

命名问题解决，搜索高效

缺点：

不同用户间文件共享问题（有的系统简单地不允许本地用户文件被其他用户访问）；系统文件多次备份的问题

解决方案：

定义一个特殊用户包含所有系统文件，当本地 UFD 搜索不到文件时默认去特殊用户下搜索。

(3) 树状结构目录

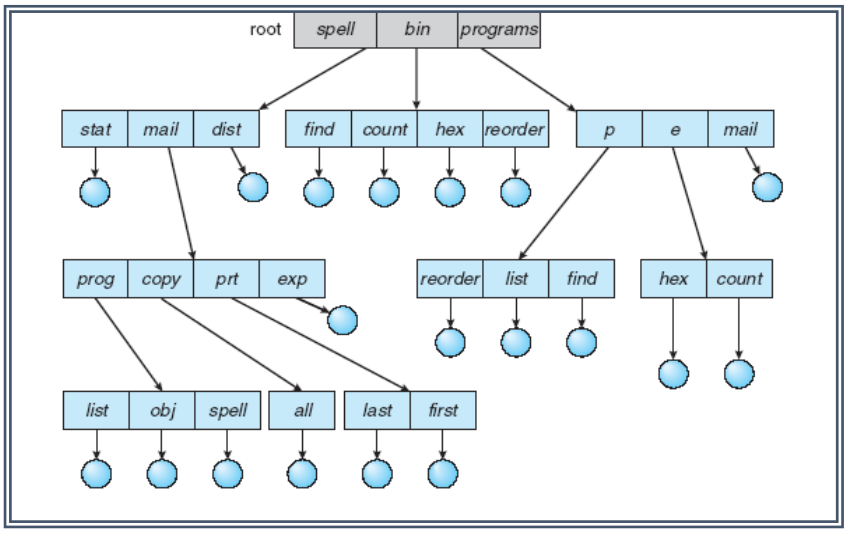
将目录结构扩展为任意高度的树，不再按照用户区分不同子目录，所以可以通过路径访问到其他用户下的文件。

路径又有绝对路径和相对路径之分。

绝对路径：从根开始给出路径上的目录名直到目标文件。

相对路径：从当前目录开始定义路径

（pwd 命令显示当前路径，cd 命令切换当前路径）



优点：

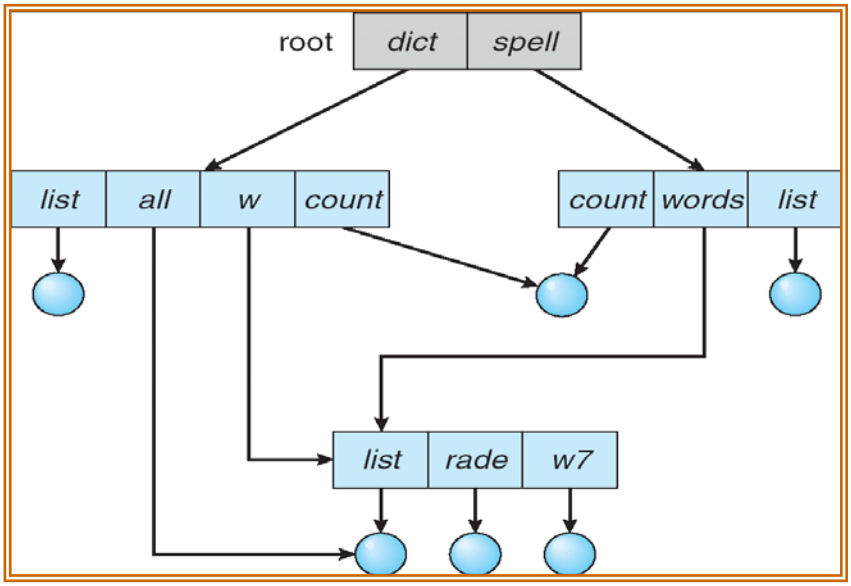
易于管理，搜索高效

缺点：

多用户对同一文件的共享问题

(4)无环图目录

允许目录含有共享子目录和文件，共享文件（或目录）不同于文件的复制，对于一个共享文件，只存在一个真正的文件。当多用户在一个组工作时，只需要把共享文件目录设为每个用户目录（UFD）下的子目录就可完成分组共享。



优点：

实现了文件共享

问题：

（1） 同一文件拥有多个绝对路径，遍历时重复计数。

（2） 删除共享文件时，会留下悬空指针（dangling pointers）

解决方案：

（1） 提出符号链接（symbolic link）又称软链接概念，即指向共享目录的指针，用绝对路径名实现，通过路径名定位真的文件获得解析，与共享文件唯一真正

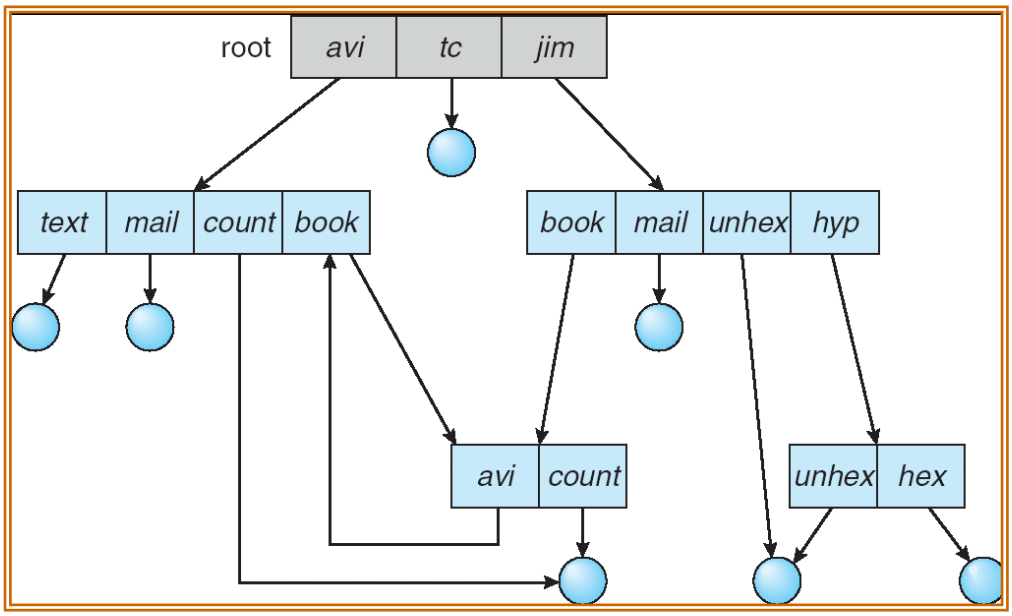
的绝对路径获得区别。即它存的不是真正的文件节点，而是指向真正文件节点的指针。解决了共享计数问题。

（2） 引入链接概念后，当实际文件被删除时，链接保留，当用户通过链接访问文件时再告知用户链接已经失效，Windows，Unix 都采用此种处理方式。

(5) 通用图目录

优先考虑无环图的实现模式，遍历算法实现简单。

但当目录中允许环存在时，存在重复遍历问题和删除时出现因为自身引用，导致引用计数无法为 0 而永远无法被删除的问题。



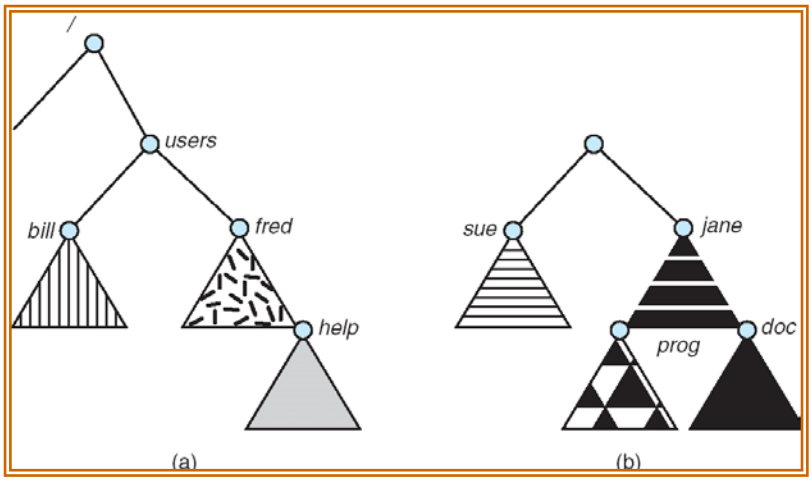
解决方案：

（1） 在无限循环的遍历中设置最大遍历次数，如果超过则自动跳出。

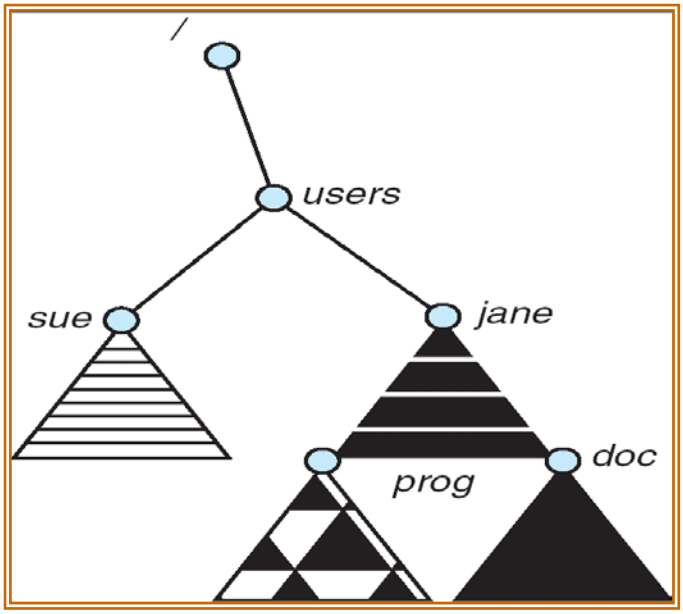
（2） 遍历两遍，第一遍标记所有可以访问到的空间，第二遍将没有访问到的空间加入空闲空间链表。（自引用的孤立节点没法访问到）

10.2.4 文件系统的安装（Mount）

文件系统在被进程使用之前必须安装，具体说目录可以建立在多个文件系统（卷 Volume）上，这些“子”文件系统必须使它们在这个多文件系统目录的命名空间内可以被访问到。比如 U 盘作为一个新的文件系统（Windows 里的新加卷）必须要安装，成为本地系统的某个盘（如 H 盘）一样。



如图（a）为原文件系统，（b）为要安装的文件系统



安装后会 隐藏原来/users 下的文件目录，访问安装后的目录，当卸载（unmount）这一卷时，则恢复原有的文件目录。

10.2.5 文件保护

文件的拥有者规定文件的访问权限（访问用户；访问操作）

Unix 通过对 Read，Write，Execute 三种基本操作，对 owner，group，public 三组用户进行授权。对应相应的 3*3 位二进制表示，1 标示允许，0 标示拒绝。

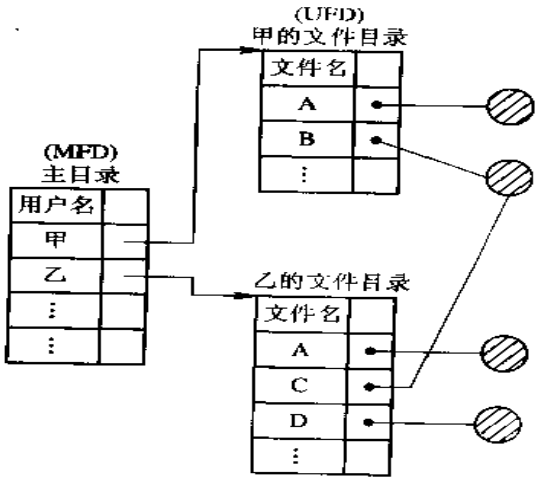
如 100 010 001 表示对拥有者可读，对同组用户可写，对公共用户可执行。

10.3 例题解析

10.3.1

若有甲、乙两个用户，甲用户有文件 A、B，乙用户有文件 A、C、D，甲用户的文件 A 与乙用户的文件 A 不是同一个文件。甲用户的文件 B 与乙用户的 C 是同一个文件。请设计一个目录组织方案，并画图说明。

解：由于本问题是有两个用户，并且存在文件重名和别名问题，因此目录组织方案应采用二级目录结构，如下图所示。



10.3.2

打开文件列表用来维护当前打开文件的信息。操作系统应该为每个用户维护一个单独的列表还是维护一个包括同时被所有用户访问的文件的引用数的全局的列表？如果相同文件被两个不同进程或用户访问，打开文件列表应该有两个不同的条目么？

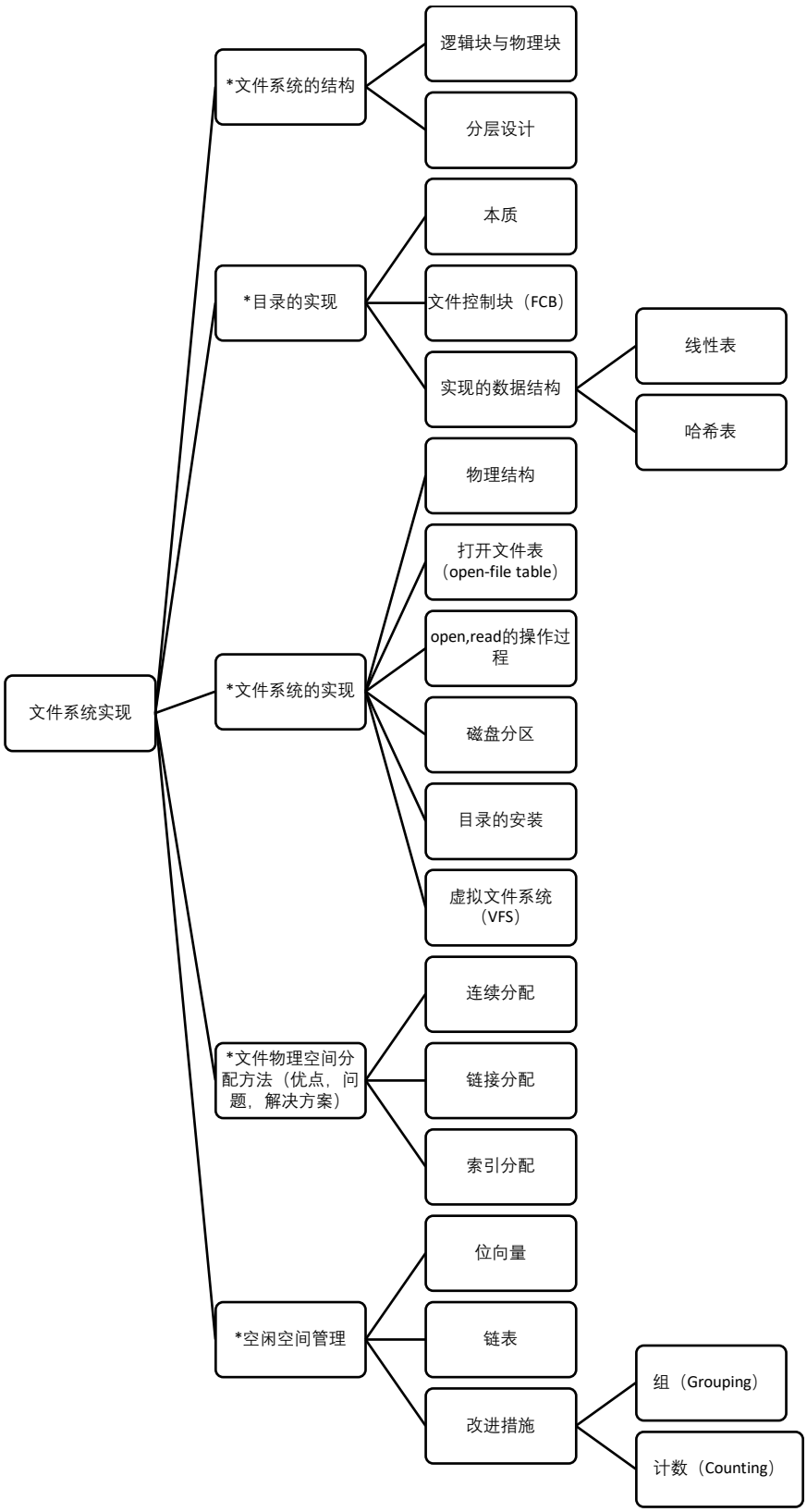
这个题暗示着第 11 章的两种打开文件表（per-process open-file table & system-wide open-file table）的登场，也解释了为什么要有两种打开文件表的原因。

当一个进程想提交对某文件的删除操作时，为了保证实现当只有所有对该文件的引用都被撤销时这一文件才被从磁盘上删除，必须记录当前文件被引用的数量，这就要求只为相同文件记录一个全局的表，当有进程打开引用该文件时计数就加 1，一个进程对该文件关闭时，计数就减 1。单独为每个进程维护一个表，无法实现全局的计数删除。

当多个进程同时操作同一文件时，它们对文件的操作位置不同，这时就需要为每个进程维护一个打开文件表来记录当前进程对该文件操作的位置指针。

第十一章 文件系统的实现（File System Implementation）

11.1 知识框架



11.2 学习指导

这一章的学习要从实现角度出发思考问题，多问为什么。

之前提了那么多文件系统的概念，首先文件系统在硬盘上到底是什么结构？

为了实现用户端通过目录对一个文件的按名打开，目录需要被提前加载进内存，那么目录是怎么实现的？目录的叶结点存的是什么？

当打开一个文件后对文件进行操作，难道每次都要按照打开时找到文件的方法通过相对费时的目录查找找到目标文件的操作位置么？那是怎么高效实现的呢？用户只知道文件的逻辑位置那它到底存在磁盘的什么物理位置呢？一个文件的所有数据都连续分布在磁盘上吗？如果不是它又是怎样映射的呢？

当新建一个文件时操作系统该干什么？分配物理存储空间时又是怎么知道哪块是空闲的呢？

带着这些问题去探索，这一章其实很清晰。

作者：孙吉鹏

11.3 文件系统的结构

文件系统有两个设计问题：

(1) 如何定义文件系统对用户的接口？

即文件及其属性，所允许的操作，组织文件的目录结构等，在第 10 章已经探讨。

(2) 创建怎样的数据结构和算法将**逻辑文件系统**映射到**物理外存设备**上？本章进行探讨。

11.3.1 逻辑块与磁盘块

(1) 逻辑块

文件系统将磁盘视为一个逻辑空间，该逻辑空间就象一个大的数组，数组的每个元素是文件系统操作的基本单位—逻辑块。比如数据库表里的每行记录就是一个逻辑记录。

逻辑块是从 0 开始编址且逻辑块是连续的

逻辑块的大小一般是扇区大小的 2^n 倍

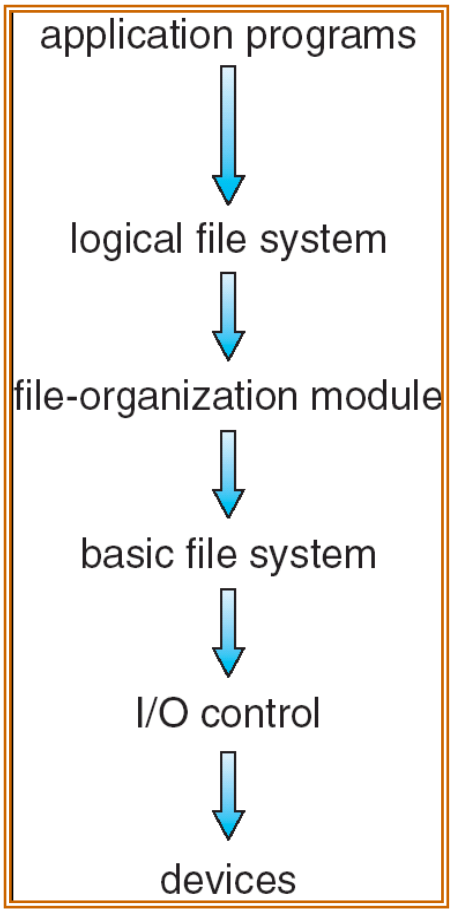
(2) 物理块

物理块是数据在磁盘上的存取单位，也就是每进行一次 I/O 操作，最小传输的数据大小，一般情况下，一个物理块对应一个扇区 （有时也对应多个）

为了让整数 n 次 IO 读进来一个逻辑块，所以把逻辑块的大小一般设计成扇区大小的 2^n 倍。

11.3.2 文件系统的分层设计

为了实现从逻辑地址到物理磁盘地址的转换，采用分层设计，上层（逻辑文件系统）负责管理文件逻辑数据，中间层（文件组织模块）负责将逻辑地址转化为基本文件系统的物理块地址，下层（基本文件系统）对底层驱动程序发送物理地址读写指令。



逻辑文件系统（logical file system）通过文件控制块（FCB）管理文件的所有不包括具体数据（数据块内容）的结构数据，即元数据（metadata），维护文件的结构。

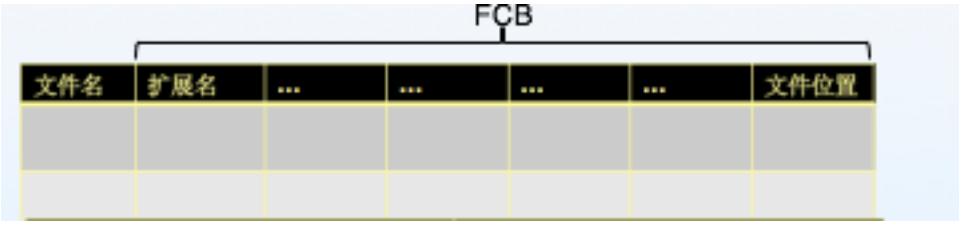
文件组织模块(file-organization module) 通过逻辑文件系统传入的 FCB，使用 FCB 中文件分配类型和文件的位置信息，算出基本文件系统所用的物理块地址。

11.4 目录的实现

11.4.1 目录的本质

Unix 中将目录视为文件，称为目录文件，目录文件中的内容是目录表，在打开文件前就将目录文件信息加载进内存，根据查询对应文件名的一行记录找到文件位置信息。

两种常用的目录表



DOS 等系统采用，如 FAT 文件系统

实现简单

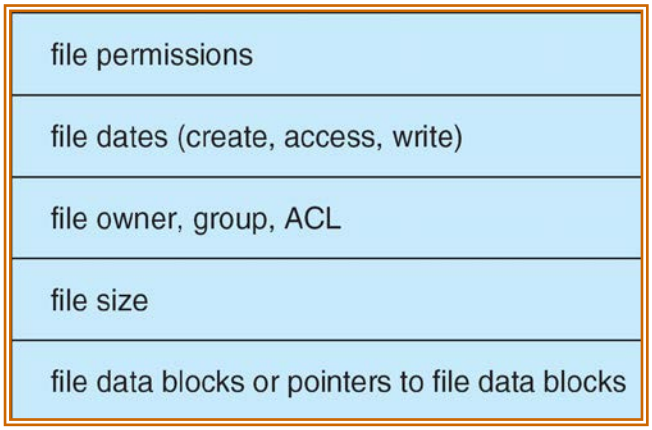
不利于实现文件共享（第二种可以实现共享索引节点号（Unix 系统里的 FCB）来实现文件的共享，而这种方法不是获得指向 FCB 的指针而是 FCB 的具体内容，所以不易共享）

文件名	索引结点号

Unix 采用
将 FCB 与文件名分开
目录表简单（名号目录项）
便于文件共享

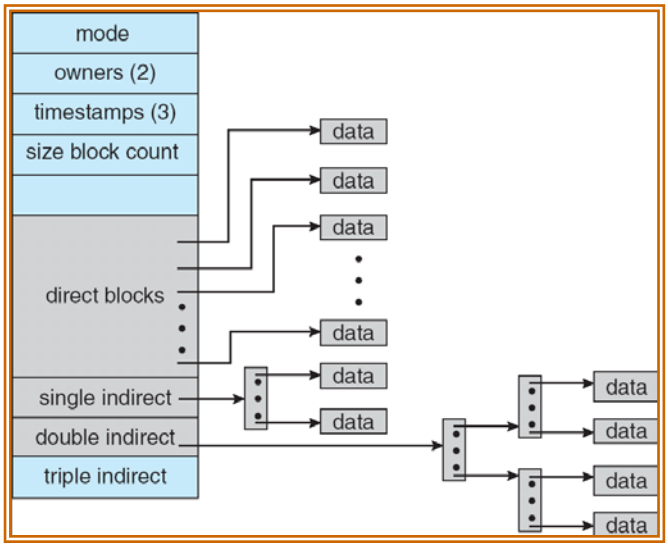
11.4.2 文件控制块（FCB）

FCB 是种数据结构，它包含了一个文件的所有结构信息，通过目录表里的叶结点中文件名和 FCB 号的一一映射，我们可以根据文件名获取该文件的 FCB，我们后期所有对单一文件的操作所需要的信息本质上都来源于 FCB，一般 FCB 的结构



注意，FCB 中包含对该文件数据块物理地址的指针。

Unix 里由 inode 充当 FCB 这个角色，它的在数据块地址索引的设计上采用了多级索引结构，很适合不同大小文件的灵活使用。



11.4.3 目录实现的数据结构

(1) 线性列表（Linear List）

将目录文件表每行的信息用线性表组织起来，易于编程但是查找，创建新文件困难。

(2) 哈希表（Hash Table）

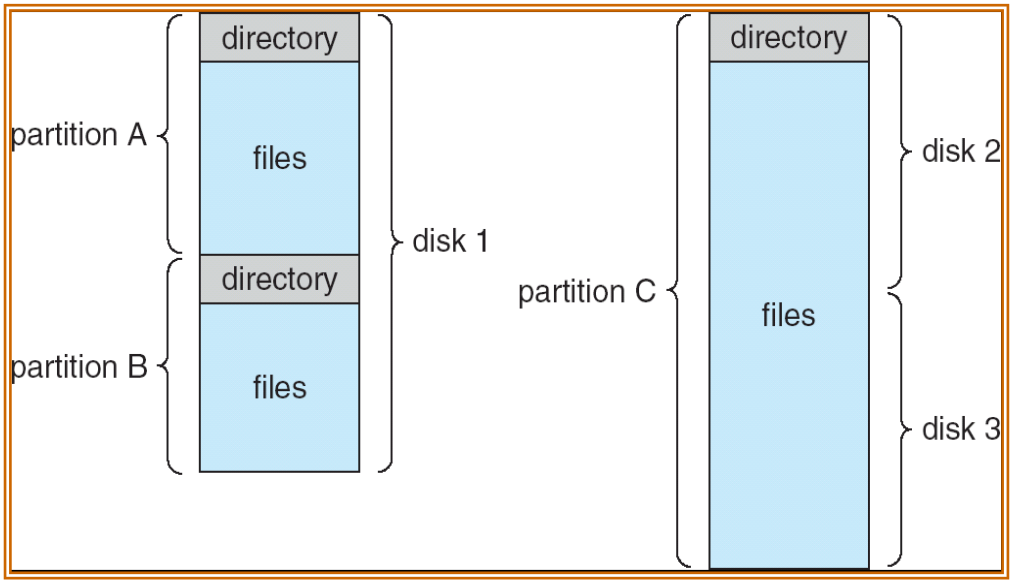
根据哈希函数易于查找，但是会出现冲突（collision），出现溢出块（类比数据库的哈希索引）降低效率但还是要比线性表全表查找快。

11.5 文件系统的实现

11.5.1 文件系统的物理结构

(1) 磁盘上

磁盘可以整体用作一个操作系统，也可以在将一个磁盘分为几个磁盘分区或片（partition），每个分区上创立一个文件系统。这些分区可以组合成卷（Volume）的结构，在卷上创立文件系统，也就是说卷上的文件系统可以使用多个子文件系统（分区上的文件系统），这就开阔了之前安装（mount）的概念，当时为了直观举 U 盘的例子，其实每个分区都是一个“U 盘”。



磁盘上，文件系统包含以下结构：

a. 卷的引导控制块（per volume boot control disk）

该卷如果有操作系统，则存放引导信息，没有则为空，Unix 又称引导块（Boot Block）

b. 卷控制块（per volume volume control block）

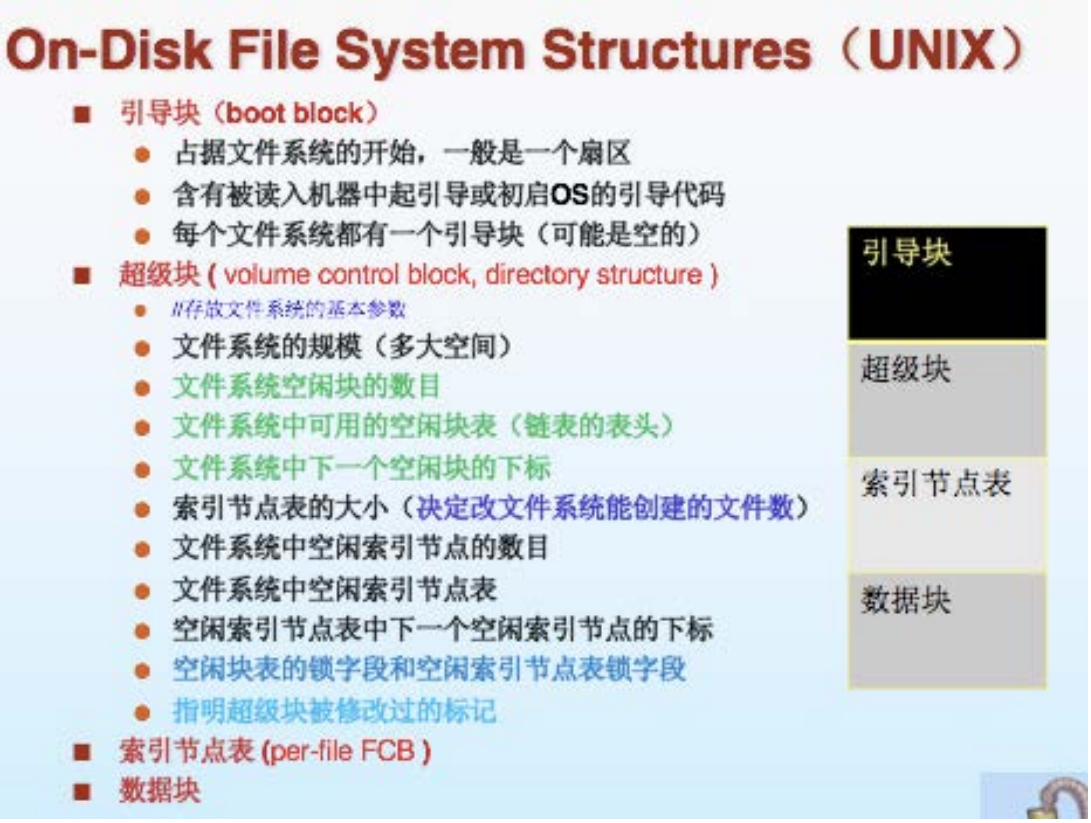
包括卷的详细信息，包括卷分区的块数，块的大小，空闲块的数量和指针，空闲 FCB 的数量和指针，Unix 里又称超级块（Super Block）

c. 每个文件系统的目录结构（Directory structure per file system）

就是目录表，一个文件系统一个，卷上的“大”文件系统也需要有一个目录结构，目录就存在这里，在想打开该文件系统的文件时，这一块需要先加载入内存。

d. 每个文件的 FCB（per file）

e. 具体文件的数据块



(2)内存中，也就是说操作系统对文件系统的认识，它了解磁盘上的文件系统只能通过以下形式，当一个文件系统安装时装入这些结构，卸载时删除这些结构。

a. 安装表（mount table）

包含指向安装设备文件系统超级块的指针

b. 目录结构缓存（directory-structure cache）

实现目录对文件的查找

c. 系统范围内的打开文件表（system-wide open-file table）

每个打开文件的 FCB 副本和其他信息

d. 每个进程的打开文件表(per-process open-file table)

一个指向系统范围打开文件表中对应文件条目的指针和其他信息。

11.5.2 打开文件表（open-file table）

如上一章例题第 2 题里讲的一样，维护两个打开文件表是有原因的，详见上一章的答案解析。

11.5.3 *Unix 中 open() 和 read() 的操作实现

(1) open()操作：

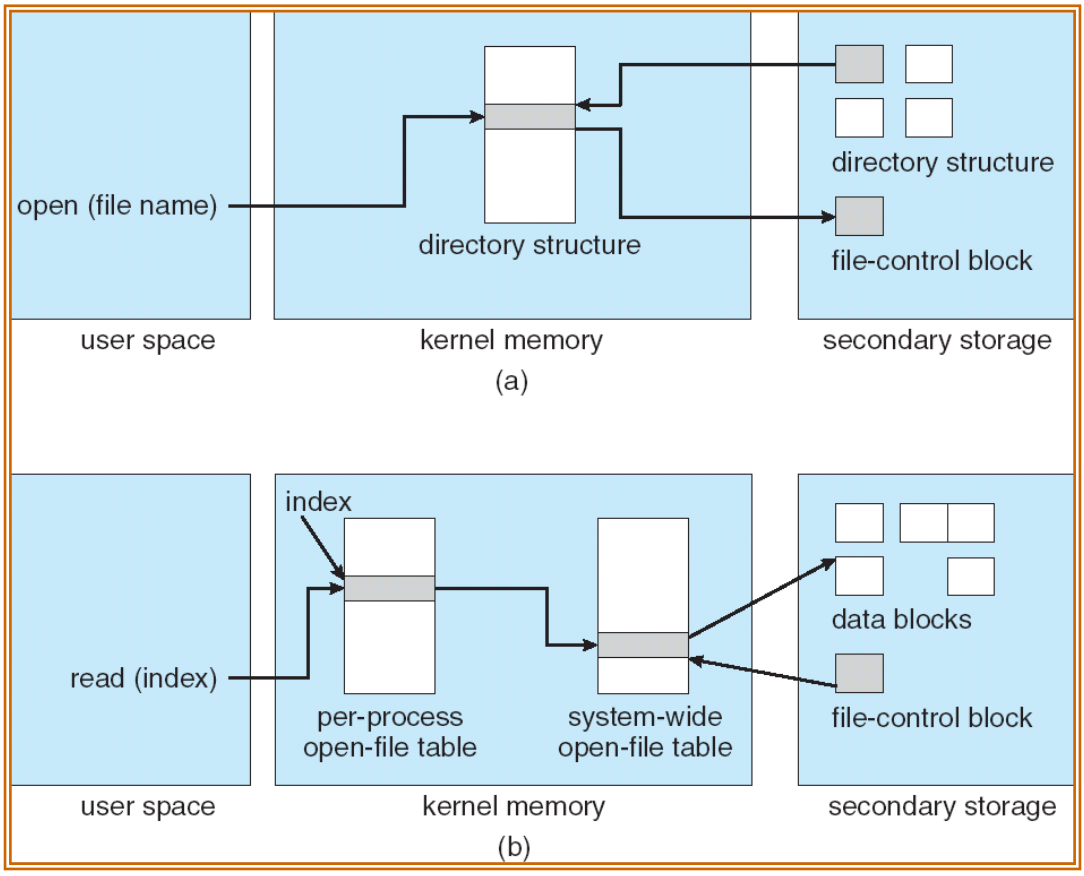
用户给 open()传入一个文件的逻辑路径名，这时先将该文件系统的目录结构加载进内存，根据文件名，操作系统会首先对系统范围内的打开文件表进行搜索（节省时间）。

如果该文件已经被其他进程打开了，则直接将该进程的打开文件表中的指针指向系统范围打开文件表的这一项，同时，系统打开文件表该文件引用计数加 1。

如果该文件在系统范围文件表中不存在，说明该文件第一次打开，则对该文件系统的目录表进行搜索，依次查找到叶结点，叶结点包含了一个该文件控制节点（inode）号，即控制节点的物理位置指针，将这个指针返回给用户，同时在系统范围打开文件表中新注册一行这个文件的信息，将该进程的打开文件表中指针指向这条新信息，open()操作的任务就完成了。

(2) read()操作

通过 open()操作返回的该文件的索引节点号从进程的打开文件表中的指针找到系统的打开文件表中该文件的 inode（FCB）物理位置指针，将该 FCB 读入内存，通过 FCB 中文件存储类型和存储地址的信息算出数据块的存储地址，将数据块读入即可完成 read()操作。



（a）open( ) (b) read( )

11.5.4 磁盘分区

分区可以有生分区（raw）和熟分区(cooked)

生分区：

没有文件系统的磁盘分区，可以用作页帧对换区（swap space）；数据库自己建文件系统管理自治一样

熟分区：

装有文件系统的磁盘分区，采取了逻辑格式化 Format

逻辑格式化 Format 可以创建文件系统

逻辑格式化 Format 做了哪些工作？

a. 划分磁盘块：将一定的扇区组织成磁盘块

b. 创建文件系统，建立文件系统在磁盘上的布局及建立文件系统所使用的数据结构，如引导块、超级块、目录表、FCB 表(索引结点表)、文件分配表(如 FAT)、空闲块索引表等

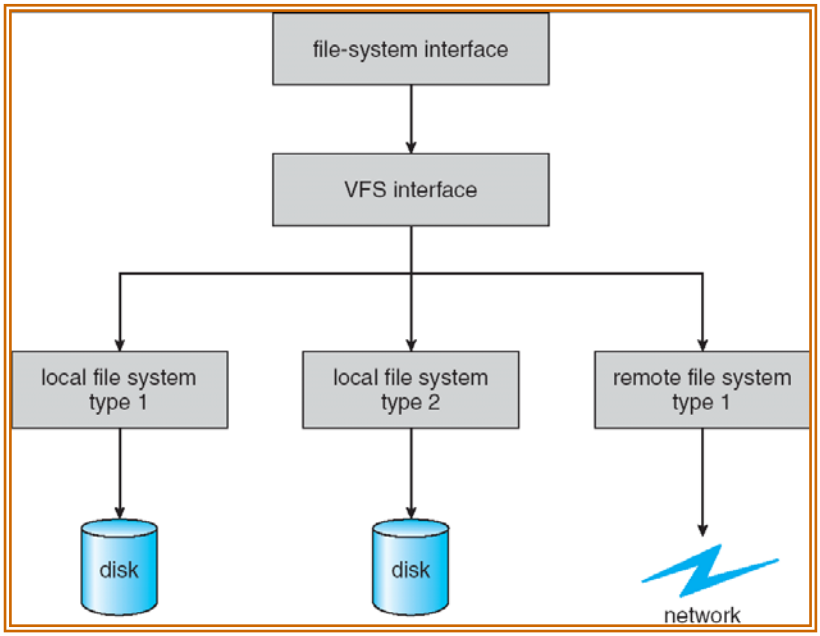
11.5.5 再看目录的安装

将主文件系统位于内存的目录结构中对应安装点（mount point）的 inode 上加一位标记，并将指向新的文件系统的超级块的指针加入到主文件系统的安装表（mount table）里，当遍历到对应 inode 节点时去安装表里找指针，遍历新加入的文件系统，从而实现文件系统里的无缝切换。

11.5.6 虚拟文件系统（VFS）

虚拟文件系统(VFS)是物理文件系统与文件系统服务之间的一个接口层（VFS Interface），它对每个物理文件系统的所有细节进行抽象，并为这些不同的文件系统提供了一个**统一的系统调用接口**。

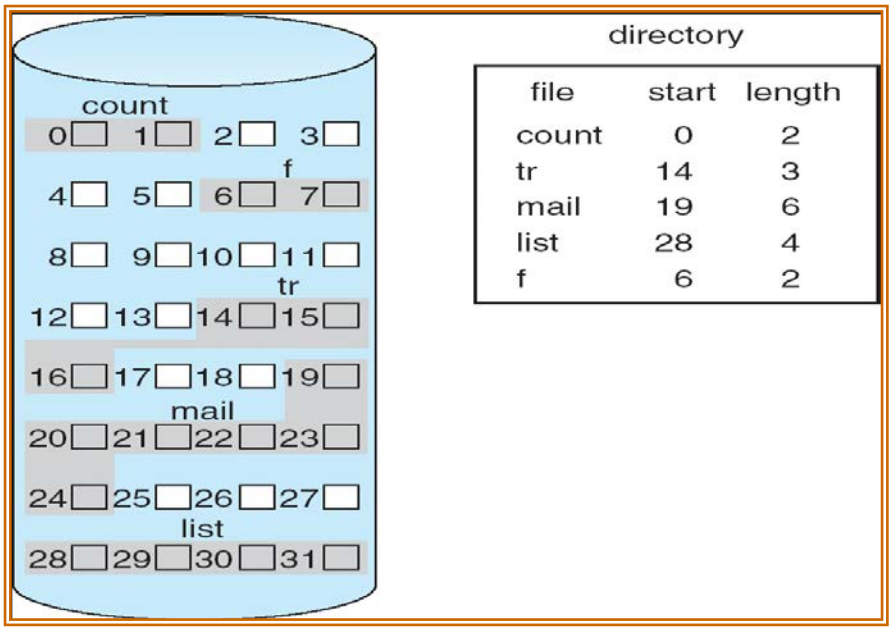
严格说来，VFS 并不是一种实际的文件系统。它只存在于内存中，不存在于任何外存空间。VFS 在系统启动时建立，在系统关闭时消亡。



11.6 文件物理空间的分配方法

11.6.1 连续分配（Contiguous Allocation）

每个文件在磁盘上占有一组连续的块，可以用第一块磁盘地址和连续块的数量来定义。



优点：

实现简单，随机存取速度快，效率高，适合文件内容不进行变动的情况（swap space）

问题：

难以进行文件扩展，需要提前声明文件大小

产生外碎片

分配方法：best fit; first fit; worst fit; next fit

例题：

逻辑地址和物理地址转化问题

假设每个磁盘块的大小(block size)为 512 bytes

给定要访问文件的逻辑地址 LA，则

LA 除以 512（block size），商为 Q，余数为 R，S 为文件的第一个磁盘块号

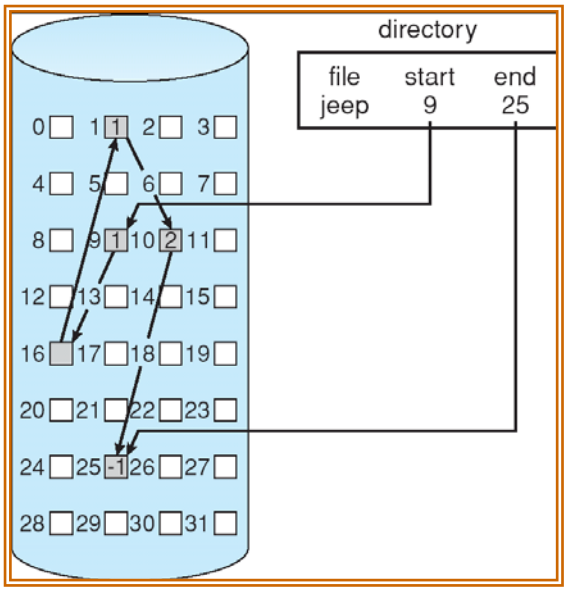
(LA，Q，R，S 的含义)

访问的物理块号 = S + Q

访问的块内偏移地址 = R

11.6.2 链接分配（Linked Allocation）

每个文件是磁盘块的链表，磁盘块分布在磁盘的任何地方，目录块包含文件第一块和最后一块的指针。



优点：

实现简单，只需要首地址

没有外碎片问题

文件可以扩展，不需要提前声明文件大小

问题：

每个文件块都有指针，占用空间

无法实现随机读取

可靠性差，一个中间数据块中指针的丢失都会导致链的断裂

例题：

逻辑地址和物理地址转化问题

假设每个磁盘块的大小(block size)为 512 bytes，其中一个字节作为存储指针的地址，511 字节存储实际数据。

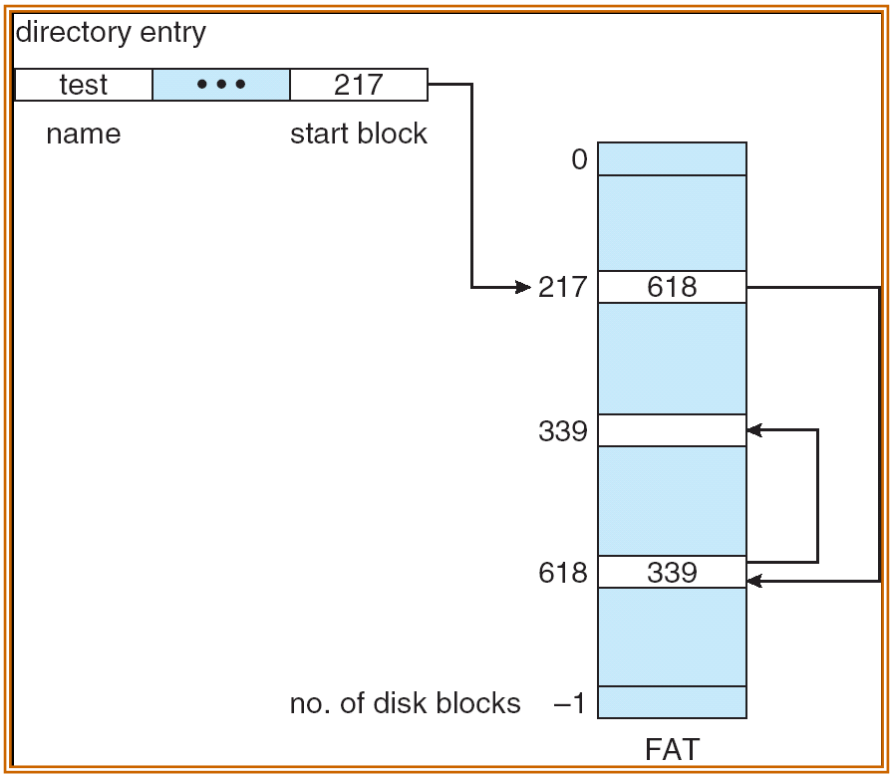
给定要访问文件的逻辑地址 LA，则

LA 除以（512-1）（data block size），商为 Q，余数为 R

Q 为要访问链上的第 Q 个节点块，R 为块内偏移量

加入文件分配表（File Allocation Table）的链接分配改进版

每个卷的开始部分用来存储该卷的 FAT，每块在该表中都有一项，该表可以通过块号码进行索引，FAT 使用跟链表相似，目录条目含有文件首块的块号，根据块号索引的 FAT 条目包含文件下一块的块号，这条链一直继续直到最后一块，该块对应的 FAT 条目的值为文件的结束值，该项存一个特殊的结尾符表示文件的结束。



优点：

通过一下 IO 一次性读入 FAT 表进入内存，就可实现对文件任意位置的随机访问（链表移动工作放在了内存中对 FAT 表项的移动）

问题：

如果不采用缓存（Cache）将 FAT 表读入内存，会导致每次访问都要先访问 FAT 表，导致访问时间的浪费。

例题：

某磁盘文件区 256GB ，回答下列问题：（列出解题步骤）

1、如果采用 FAT32 文件系统，假定每个磁盘块大小为 1KB，问 FAT 表需要占用几个磁盘块？

2、如果采用 FAT32 文件系统，每个磁盘块最小可以是多少字节？

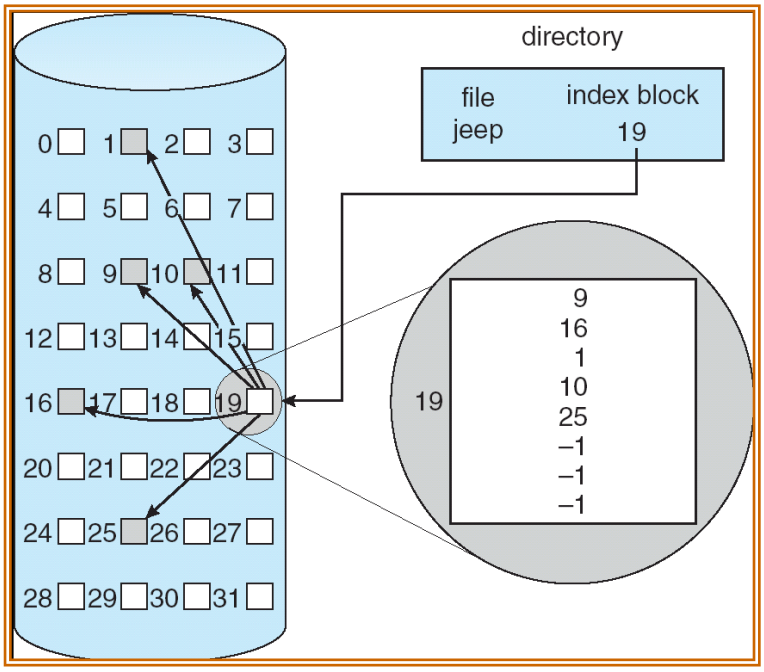
3、如果采用 FAT16 系统，在磁盘空间不变的情况下，每个磁盘块最小可以是多少字节？

答案：

1、
磁盘总字节数：$2^{30}*2^8 = 2^{38}$字节
磁盘总块数：$2^{38}/2^{10} = 2^{28}$ 块
FAT表所占空间（字节数）： 由于每个FAT表项占用32位，因此FAT所需字节数为：2^{28} *4 字节=2^{30} 字节
采用FAT所占盘块数为： 2^{30} / 2^{10} 块=2^{20}块
2、磁盘总盘字节数： 2^{38}字节
磁盘的总块数上限： 2^{32}块 (由于每个FAT表项占用32位，因此可以采用32位对磁盘块进行编址)
每个磁盘块最小可以为： 2^{38}/ 2^{32}=2^6字节，即64bytes
3、
磁盘总盘字节数： 2^{38}字节
磁盘的总块数上限： 2^{16}块 （采用16位对磁盘块进行编址）
每个磁盘块的大小最小可以为： 2^{38}/ 2^{16}=2^{22} 字节= 2^2 * 2^{20} 字节，即4MB

11.6.3 索引分配（Indexed Allocation）

把所有指针放到统一的索引块上，上面存放着磁盘块地址的数组，索引块的第 i 个条目指向了文件的第 i 个块



优点：
随机访问
可扩展，文件不需要提前声明大小
没有外碎片
问题：
浪费空间（比如一个文件只有两块大小，采用链接分配只浪费一个指针空间，采用索引分配则浪费 1 块索引块的空间）
例题：
逻辑地址和物理地址转化问题
假设每个磁盘块的大小(block size)为 512 bytes， 文件大小 128K bytes (2^{17} bytes)，一个磁盘块可存储 512 bytes (2^9 bytes)，因此该文件需要 256 个磁盘块存储其文件内容($2^{17}/2^9$=256)。如果每个索引项占用一个字节，作为索引块的一个磁盘块可以有 512 个索引项)

LA 除以 512（block size），商为 Q，余数为 R（Q<512(索引块中最大索引项数)）

访问的块号为索引块中的第 Q 行的块号

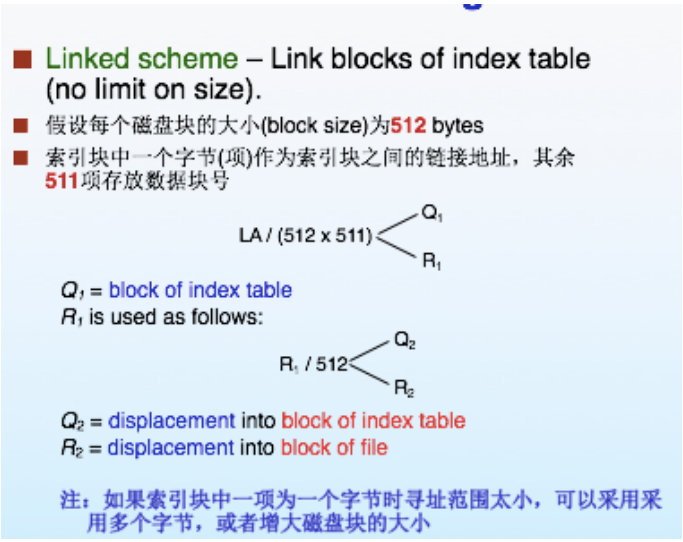
块内偏移量为 R

当所需索引块多于一块时，需要提供管理组织索引块的机制，有三种方案。

(1) 链接方案

将索引块链接起来，每块留出一个字节来存放下一块索引块的指针

例题：



Q1 得出应该访问第几块索引块

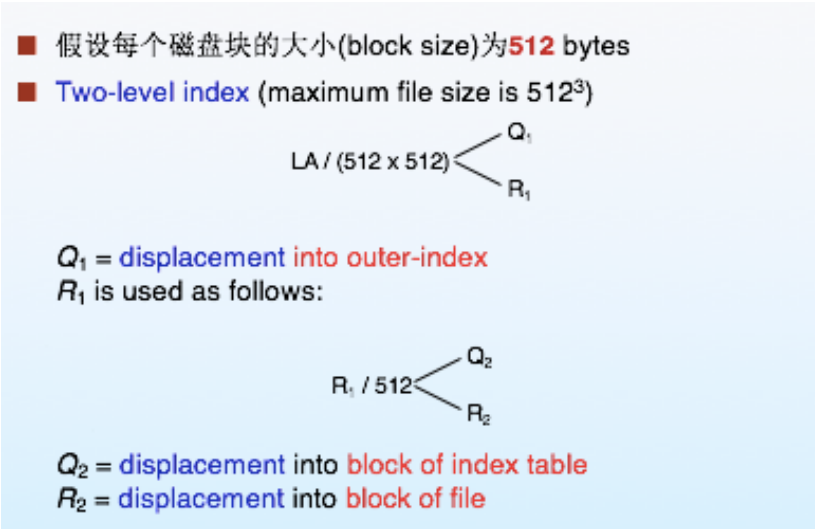
Q2 得出应该访问该索引块的第几行上对应的数据块

R2 得出块内偏移量

(2) 多层索引方案

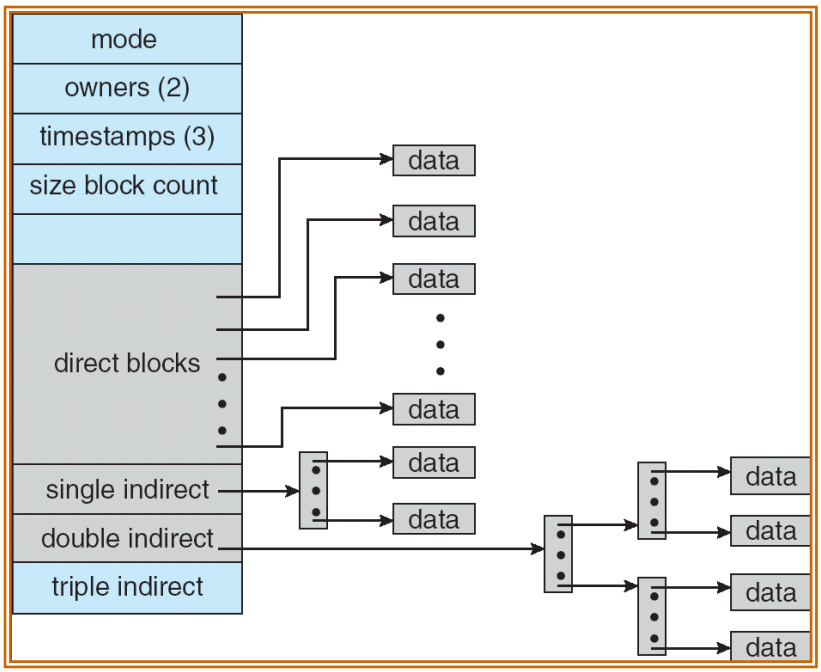
将一个链接块索引指向第二层的索引块，第二层的索引块再指向文件数据块，扩展性极强。

例题：



(3) 组合方案

Unix 中既可以直接一层索引，也可以有两层，三层，提高了文件分配大小的灵活性。



11.7 空闲空间管理

11.7.1 位向量（Bit Vector）

将空闲空间表现为位图（bit map），每块空闲空间用一位表示，如果空闲则为 1，已分配则为 0。

优点：

实现相对简单

容易产生连续空间分配的文件

问题：

如果磁盘块数过多，对应的超级块空闲空间分配位图也很大。

例题：

某磁盘文件区 16GB ,每个磁盘块大小为 1KB，回答下列问题：（列出解题步骤）

如果空闲存储空间采用位示图（位向量）管理方法，那么位示图需要占用多少个盘块？

答案：

磁盘总盘字节数 (16GB)：$16^* 2^{30} = 2^{34}$字节

磁盘总块数：2^{34} 字节/2^{10} (字节/块)= 2^{24} 块

存储位示图需要的位数：2^{24} 位 (每块对应1位)

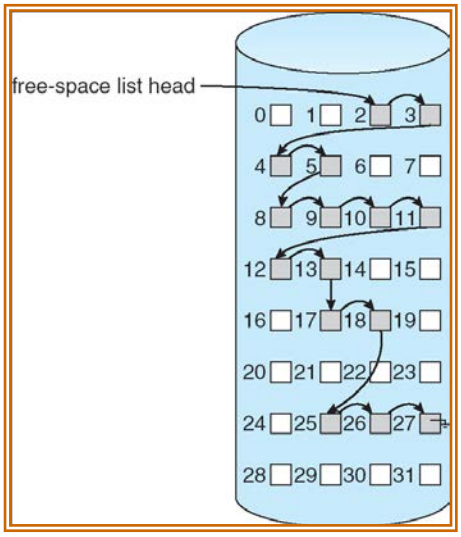
存储位示图需要的字节数：2^{24}位 /8(位/字节)=2^{21}字节

位示图所占盘块数：2^{21}字节/2^{10} (字节/块)=2^{11}块 (2k块)

11.7.2 链表（Linked List）

将所有空闲磁盘块用链表链接起来，并将指向第一个空闲块的头指针保存在磁盘的超级块（Super block）上，同时也缓存在内存中；

(类似于页式管理中的空闲帧列表)



优点：

查询时占用内存空间小（读进一块即可，优于位向量法）

问题：

可靠性

不易找到大量空闲块，查找费时

11.7.3 改进措施

(1) 组（Grouping）

对于空闲链表而言，可以将 n 个空闲块的地址存在第一个空闲块中，之后的 n-1 块确实为空，最后一块包含另外 n 个空闲块的地址。

可以快速找到大量空闲块的地址

(2) 计数（Counting）

建立一个类似于分区内存管理中的分区表；

每个表项记录第一块的地址和与第一块连续的空闲块的数量；

优点：

容易找到连续的空间

需要额外的空间，但表的总长度会更短，因为连续块的数量往往大于 1.

11.8 例题解析（大量例题在内容详解里）

11.8.1

一个文件的逻辑记录大小为 125 字节，共有 20 个逻辑记录，文件系统采用链接方式将这个文件存储到磁盘上，磁盘分块大小为 512 字节，请问：

(1)采用什么方法可有效地利用磁盘空间?

(2)若用户要读包含第 1285 字节的逻辑记录，文件系统将如何工作?

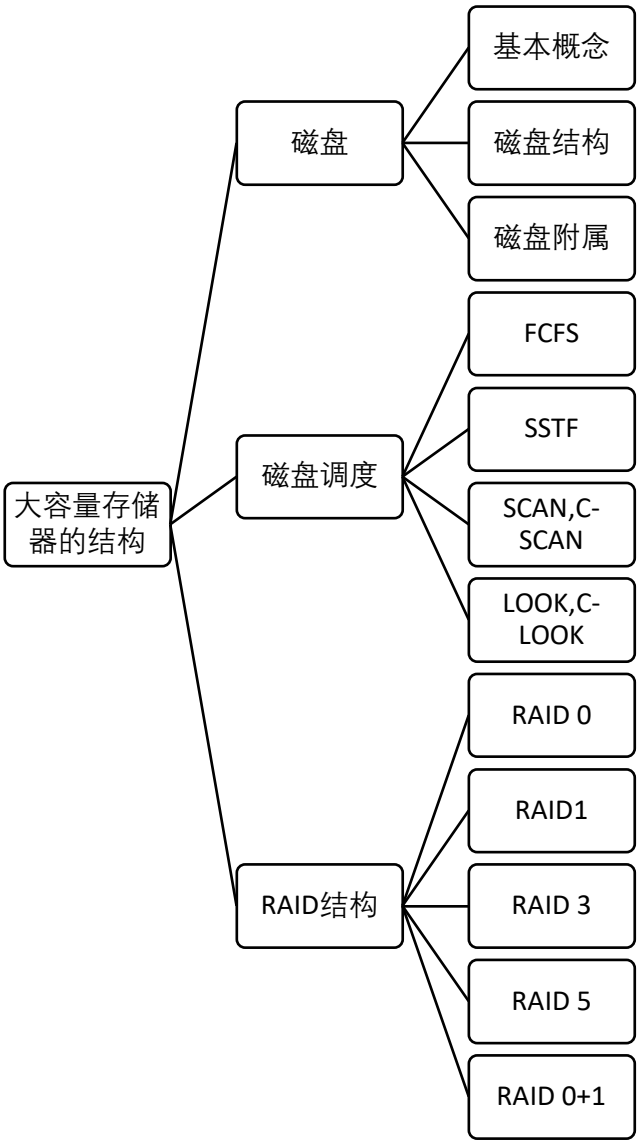
答：(1)可采用记录的成组方法，因[512/125]=4，所以成组的逻辑记录个数应为 4。20 个逻辑记录共需要 5 个存储块(0~4)。

(2)首先，由[1285/(125×4)]=2,可知包含 1285 字节的逻辑记录在链接结构的第 2 块上。将该块读入主存缓冲区；然后由 1285 mod (125×4)=285，且[285/125]=2，可知文件系统从主存缓冲区中取出第 2 个记录传输给用户。

(注意：记录号和存储块号都是从 0 开始排序的。)

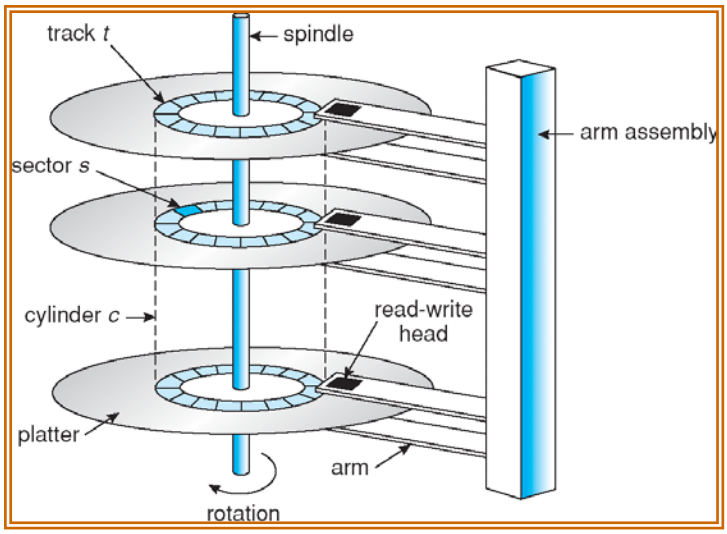
第十二章 大容量存储器的结构

作者：林子童



12.1 磁盘

12.1.1 基本概念



magnetic disk 磁盘
disk arm 磁臂
read-write head 读写头
track 磁道
cylinder 柱面
platter 磁盘面
sector 扇区
spindle 转轴
arm assembly 机械臂杆

磁头与磁臂相连，磁臂能将所有磁头作为一个整体而一起移动。磁盘片表面被逻辑划分为圆形磁道，磁道进一步划分为扇区。统一磁臂位置的磁道集合形成柱面。

磁盘速度有两部分：

传输速率：驱动器和计算机之间的数据传输速率

定位时间/随机访问时间：由寻道时间和旋转等待时间组成

寻道时间：磁臂将磁头移动到包含目标扇区的柱面的时间

旋转等待时间：磁盘需要将目标扇区转动到磁头下的时间

12.1.2 磁盘结构

现代磁盘驱动器可以看做一个一维逻辑块的数组，逻辑块是最小的传输单位，一维逻辑块数组按顺序映射到磁盘的扇区，理论上能将逻辑块号转换为磁盘内的柱面号、磁道号和扇区号。但是事实上执行这种转换并不容易，因为：

①绝大多数磁盘都有一些缺陷扇区，映射必须要用磁盘上其他空闲扇区替代

②磁盘是圆的，磁道离中心越远长度越长。为此有两种解决方式，一种是增加外部扇区数并且随着磁头外移驱动器也会增加速度；另一种方式时降低外道的磁道密度

12.1.3 磁盘附属

计算机访问磁盘存储的方式：

(1) 主机附属存储：通过本地 I/O 端口访问的存储

(2) 网络附属存储(NAS)：客户通过远程进程调用接口来访问 NAS

(3) 存储区域网络(SAN)：服务器与存储单元之间有私有网络（采用存储协议而不是网络协议）

12.2 *磁盘调度

操作系统任务之一就是有效使用硬件。对磁盘驱动器来说就是要有较快的访问速度和较宽的磁盘带宽。访问时间主要包括寻道时间和旋转延迟

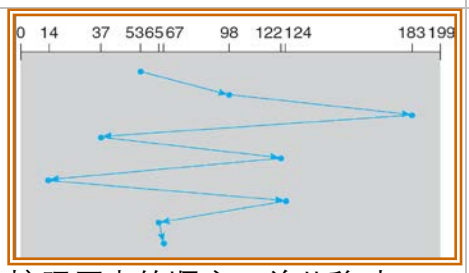
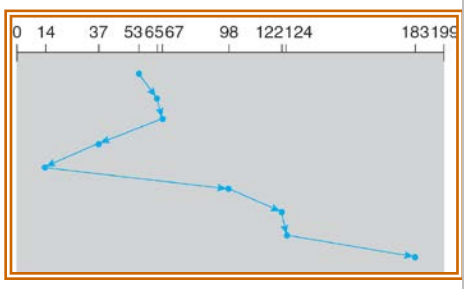
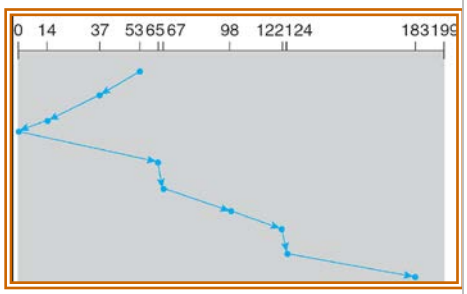
概念解析：

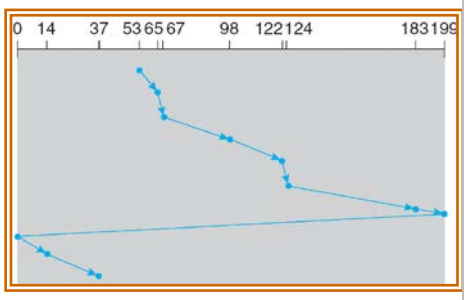
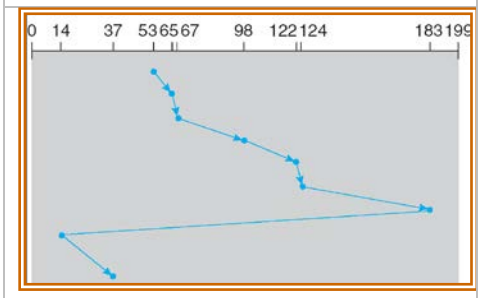
磁盘带宽：所传递的总的字节数除以从服务请求开始到最后传递结束的总时间

当一个进程需要对磁盘进行 I/O 操作时，它就会向操作系统发出一个系统调用，如果所需的磁盘驱动器和控制器忙，新的服务请求就会被加到该磁盘驱动器的待处理请求队列上。对于如何选择处理请求，操作系统有多个调度算法可供使用。

下面以一个请求顺序为例讨论磁盘的各种调度算法。

磁盘队列对 I/O 各柱面上的请求顺序为：98,183,37,122,14,124,65，磁头开始于位置 53，假定磁盘的最外道号为 199，最内道号为 0

算法	算法描述	调度图示和调度顺序	优缺点
FCFS	First Come First Served 先来先服务，根据进程请求访问磁盘的的先后次序进行调度。	 按照原来的顺序，总共移动 640 柱面	算法比较公平，但是因为磁臂摆动幅度大导致平均寻道距离大，效率低
SSTF	shortest-seek-time-first 最短寻道时间优先算法（贪心算法）每次都选择处理距离当前磁头位置最近的待处理请求	 53,65,67,37,14,98,122,124,183，总共移动 236 个柱面	提高了性能但不是最优。基本与 SJF 最短作业优先调度一样，因为新请求随时会抵达，所以可能导致磁头一直在一个小范围摆动，产生饥饿。
SCAN	又称电梯调度算法。磁臂从磁盘的一端向另一端移动，磁头经过每个柱面时处理位于该柱面上的请求。到达**另一端**时改变移动方向继续处理	 53,37,14,**0**,65,67,98,122,124,183，共移动 236 个柱面	避免了饥饿现象。但当磁头到达一端折返往回移动时，在该端等待服务的请求可能很少，等待的时间也比较短，因为该区域的请求刚刚被访问过；而另一端等待服务的请求可能很多，等待的时间也可能很长

C-SCAN	circular SCAN，SCAN 调度变种，要求磁头单项移动，即磁头从磁盘一边移动到另一端并处理请求，当磁头移动到另一端时会马上返回磁盘开始，返回时不处理请求	 53,65,67,98,122,124,183,**199,0**,14,37，总共移动 382 个柱面	提供更为均匀的等待时间，与 SCAN 相比比较公平，但需要移动更多个柱面
LOOK	SCAN 和 C-SCAN 是使磁头在整个磁盘宽度内进行移动，但这并不容易实现。通常，磁头只会移动到一个方向上最远的请求为止	 53,37,14,65,67,98,122,124,183，总共移动 208 柱面	与 SCAN 和 C-SCAN 相比提高了效率。
C-LOOK		 53,65,67,98,122,124,183,14,37，总共移动 322 柱面	

磁盘调度算法的选择：SSTF 或 LOOK 被认为是比较合理的默认算法，SCAN 和 C-SCAN 适合磁盘负荷较大的。除了算法以外，目录和索引的位置也很重要，因为它们经常被访问，一般放在中央柱面。另外需要注意，这里所描述的调度算法只考虑了寻道时间，旋转等待的时间和寻道时间几乎一样，但是操作系统比较男调度以改善旋转等待。

12.3 RAID 结构

12.3.1 独立磁盘冗余技术(RAID) Redundant Array of Independent Disks

首先了解两个基础知识：

(1) 通过冗余改善可靠性

可靠性问题解决方案是引入冗余，即存储额外信息，在磁盘出错的时候可以用来重新修补损坏信息。引入冗余最简单的方法就是复制每个磁盘，这种技术称为镜像。但是价格比较昂贵

(2) 通过并行处理改善性能

对于磁盘镜像，读请求处理速度加倍，因为有两个位置可以处理读。可见，对于多个磁盘，可以通过数据分散改善传输率。

①位级分散：在多个磁盘上分散每个字节的各个位。如 8 个磁盘，每个磁盘存储一位，就会有 8 倍的传输率，因为每个磁盘每秒所能处理的访问数不变，但每次访问都能在同样时间读到单个磁盘系统 8 倍的数据，

②块级分散：一个文件的块可以分散在多个磁盘上，读块只访问一个磁盘，可允许多个读访问并行处理，对于大量的读操作传输速度较高；大量写传输速度也高，因为数据和奇偶可以并行写。小于块大小的数据独立写不能并行进行

镜像提高可靠性却昂贵；分散提供高数据传输率却未改善可靠性。RAID 采用并行交叉存取及数据冗余校验来融合二者优点。

RAID 技术将数据按位级或者按块级分散写入到多个磁盘上，多个磁盘可以同时读写存取操作，实现数据的并行存取，提高了系统的性能；再结合数据冗余技术及校验技术，例如磁盘镜像、其它校验技术（CRC、Hamming 等），又提高系统的可靠性；

12.3.2 RAID 级别

如图，P 为差错纠正位，C 为数据的第二副本

(1) RAID 0 无冗余磁盘阵列：

数据块级分散，并行交叉存取，无数据冗余；磁盘不容错

(2) RAID 1 磁盘镜像

(3) RAID 2 内存方式的差错纠正代码结构

奇偶校验：内存系统每个字节都有一个相关奇偶位以记录字节中置 1 的个数是偶数还是奇数。如果字节的一个位损坏或者奇偶位损坏就会出现不匹配然后被内存系统所检测

RAID 2 采用了位级分散，即每一个字节的第 n 位位于第 n 个磁盘上，除了奇偶校验位以外，差错纠正方案还会存储两个或多个额外位，当单个位出错时可用来重新构造数据

(4) RAID 3 位交织奇偶结构

与内存系统不同，磁盘控制器能检测到一个扇区是否正确读取，因为采用位级分散，所以每一位都在不同的磁盘上，单个奇偶位就足够进行差错检测，加上磁盘控制器检测的某个磁盘出错，就完全可以完成纠正了。因此级别 2 在实际中并不使用。

级别 3 的问题在于如果奇偶校验盘损坏，数据就无法恢复。

另一个性能问题时关于奇偶校验的奇偶位需要计算和写。为了减少这样的性能损失，RAID 存储阵列包括一个专门奇偶计算硬件的控制器使 CPU 奇偶计算转换到 RAID 阵列。

(5) RAID 4 块交织奇偶结构

块级分散，采用与 RAID0 一样的块级分散，另外在一独立磁盘上保存其他 N 个磁盘相应的奇偶块。允许磁盘无缝加到 RAID 集合：加入的磁盘块初始化为 0RAID 集合仍然正确

但是若奇偶校验盘出错数据无法恢复；多个磁盘出错，数据无法恢复。

(6) RAID 5 块交织分布奇偶结构

将数据和奇偶分布在所有 N+1 块磁盘上，即第 n 块磁盘的奇偶放在第(n mod 5)+1 块磁盘上，避免了级别 4 对单个奇偶盘使用过度，是最常见的奇偶校验 RAID 系统

(7) RAID 6 P+Q 冗余方案

与级别 5 类似，也是将校验位分布在不同的磁盘上，但是不再使用奇偶校验，而是用公式生成 P、Q。P 为所有数值的异或，Q 是乘了系数的异或，其中系数是 n 个线性无关的数值。当只有一位出错时，用 P 就可以恢复（同 RAID5）；当有两位出错时，可以用 P、Q 联立求解。

核心思想就是有两份检验数据以保证两位数据出错也能保障数据安全

(8) RAID 0+1 RAID 0 和 RAID 1 的组合，有更高的性能和可靠性，同时也更昂贵

适用性：

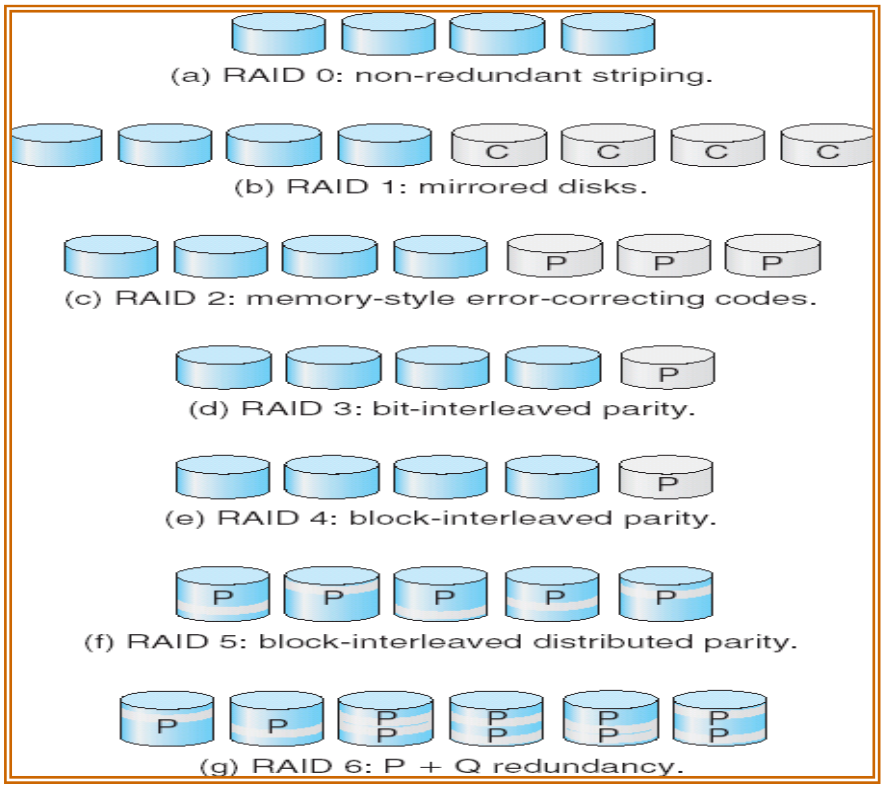
RAID 0－适于性能要求高但数据可靠性要求不高的应用；

RAID 1－适于可靠性要求高和快速恢复的应用；

RAID (0+1) 和 (1+0)－适于性能和可靠性要求都高的应用；例如小型数据库服务器；

RAID 5－用于存储量大的数据；

RAID 6－较少有支持；



12.4 例题

对比达到一个 RAID 5 级的组织 与所取得的一个 RAID 1 级安排的吞吐量如下：

a:在单一块上读取操作;

b:在多个毗连区块读取操作

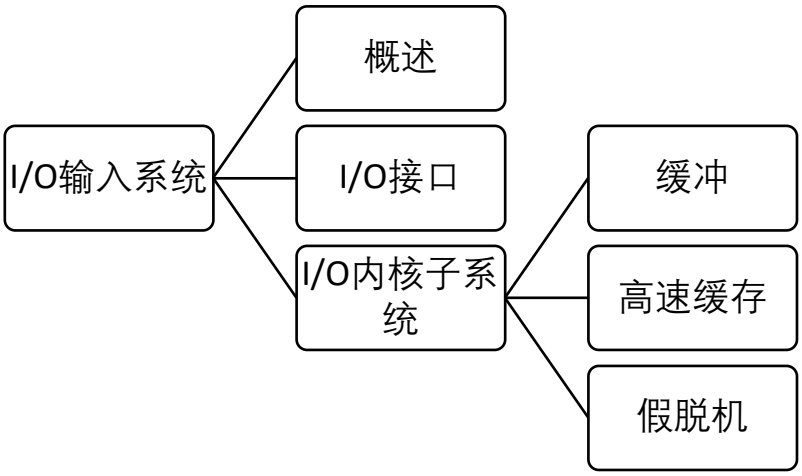
参考答案：

a)吞吐量的数额取决于在 RAID 系统里磁盘的数量。一个 RAID 5 由为每套的奇偶块的四张块延长的 5 个磁盘所组成，它可能同时支持四到五次操作。一个 RAID 1 级，包括两个磁盘可以支持两个同步行动。当然，考虑到磁盘头的位置，RAID 级别 1 有更大的灵活性的副本块可查阅，并可以提供性能优势。

b)RAID 5 为访问多个毗连区块提供更大的带宽，因为邻近的区块可以同时访问。这种带宽的改善在 RAID 级别 1 中是不可能的。

第十三章 I/O 输入系统

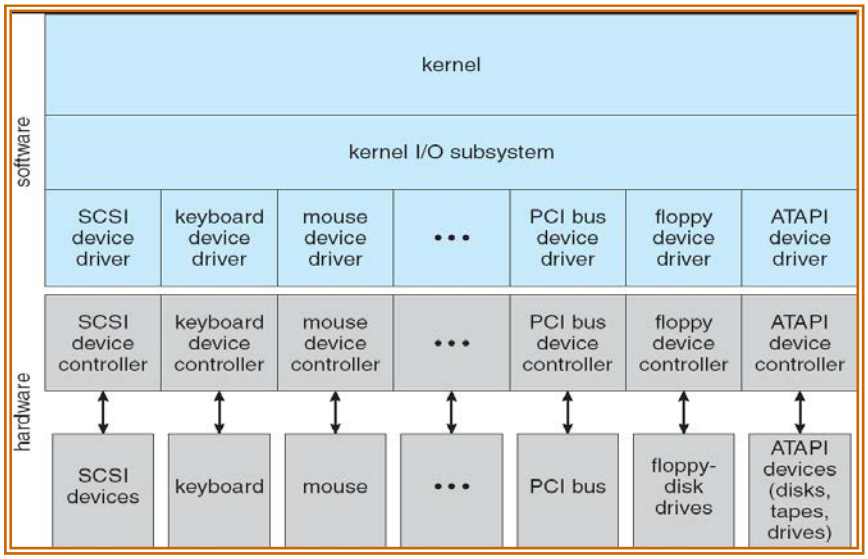
作者：林子童 鲍伟



13.1 概述

计算机有两个主要任务：I/O 操作与计算处理，许多情况下主要任务是 I/O 操作，比如浏览网页或编辑文件。操作系统在计算机 I/O 方面的作用是管理和控制 I/O 操作和 I/O 设备。

I/O 设备技术呈现两个矛盾的趋势：硬件软件接口日益增长的标准化与 I/O 设备日益增长的多样性。为了封装不同设备的细节与特点，操作系统内核设计成使用设备驱动程序模块的结构。**设备驱动程序** device driver 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。



13.2 I/O 应用接口

操作系统的组织技术与接口的目的是使 I/O 设备可以按统一的标准方式来对待，这里的方式包括抽象、封装与软件分层，具体就是从详细而不同的 I/O 设备中抽象出一些通用类型，每个通用类型都可以通过一组标准函数（即接口）来访问。设备驱动程序层的作用是为内核 I/O 子系统隐藏设备控制器之间的差异。

应用程序可以使用内核提供的统一接口访问 I/O 设备，方便了内核及应用程序的设计与编码，但也导致应用程序无法使用设备的具体特性，降低了设备的性能。绝大多数操作系统存在后门，允许应用程序将任何命令透明地传递到设备控制器，如 Unix 的系统调用 ioctl()能使应用程序访问由设备驱动程序所实现的一切功能而不需要再设计新的系统调用。

下面介绍设备差异：

1. 块与字符设备
2. 网络设备
3. 时钟与定时器
4. 阻塞与非阻塞 I/O

13.3 I/O 内核子系统

13.3.1 I/O 调度

调度一组 I/O 就是确定一个合适的顺序来执行这些请求。应用程序发布的系统调用顺序不一定总是最佳选择，调度能改善系统整体性能，如磁盘调度，重新安排服务顺序增加性能就是 I/O 调度的核心。

操作系统开发人员通过为每个设备维护一个请求队列来实现调度。

(1) 缓冲 buffer

缓冲区是用来保存两个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。使用缓冲的优点：

a. 处理数据流中生产者与消费者之间的速度差异。使用双缓冲机制，一个缓冲接收生产者数据，另一个缓冲写给消费者，若生产者比较慢，那么当第一个缓冲写满了以后两个缓冲调换，完成了生产者与消费者的分离解耦。

b. 协调传输数据大小不一致的设备。如网络传输中发送端将大消息分成若干网络包，接收端将它们放在重组缓冲区内。

c. 支持应用程序 I/O 的复制语义。当用作 buffer 的页面被修改后，复制出一个新的页面，原来的页面用于 write()，新的页面用于修改。

(2) 高速缓存 cache

高速缓存是可以保留数据副本的高速存储器。

缓冲与高速缓存差别：缓冲可能是数据项的唯一副本；高速缓存只是提供了一个驻留在其他地方的数据在高速存储器上的一个副本。

(3) 假脱机 spooling 与设备预留 device reservation

这两种技术的目的：解决独占设备的并发访问问题以提高设备利用率

脱机输入是利用专门的外围控制机将低速 I/O 设备上的数据预先输入到磁盘上，然后主机从磁盘上直接读取输入数据，脱机输出相反

spooling 技术，即外部设备联机并行操作，是利用进程与磁盘对脱机输入输出工作过程的模拟，因此又称假脱机。这种技术将慢速的字符设备（独占设备）虚拟成共享的虚拟设备，从而提高独占设备利用率和进程的推进速度。

13.4 例题精析

13.4.1 非阻塞和阻塞 I/O 是什么，主要有什么不同，分别用在哪里？

阻塞 I/O，当应用程序发出一个阻塞系统调用时，应用程序挂起，应用程序从运行队列转入等待队列。等系统调用完成之后再回到就绪队列，在合适的时候继续运行。绝大多数操作系统为应用程序提供的都是阻塞系统调用，因为它代码更加简单，更容易理解。

非阻塞 I/O，一个非阻塞调用在程序执行过长时间并不中止应用程序，它会很快返回，其返回值表示已经传输了多少字节。在多个程序协作完成 I/O 的工作时，可能会使用非阻塞 I/O。