操作系统

参考书

- 张尧学，史美林，计算机操作系统教程，清华大学出版社

课程主要内容介绍

操作系统的功能

- 管理系统软硬件资源、
- 扩展计算机的功能、
- 向用户提供服务。

- ♦用户接口
- ♦进程管理
- ♦处理机管理
- ♦存储管理
- ♦文件系统
- ♦外部设备管理

第一章 绪论

1.1 什么是操作系统

返回

1.1.1 操作系统的地位和目标

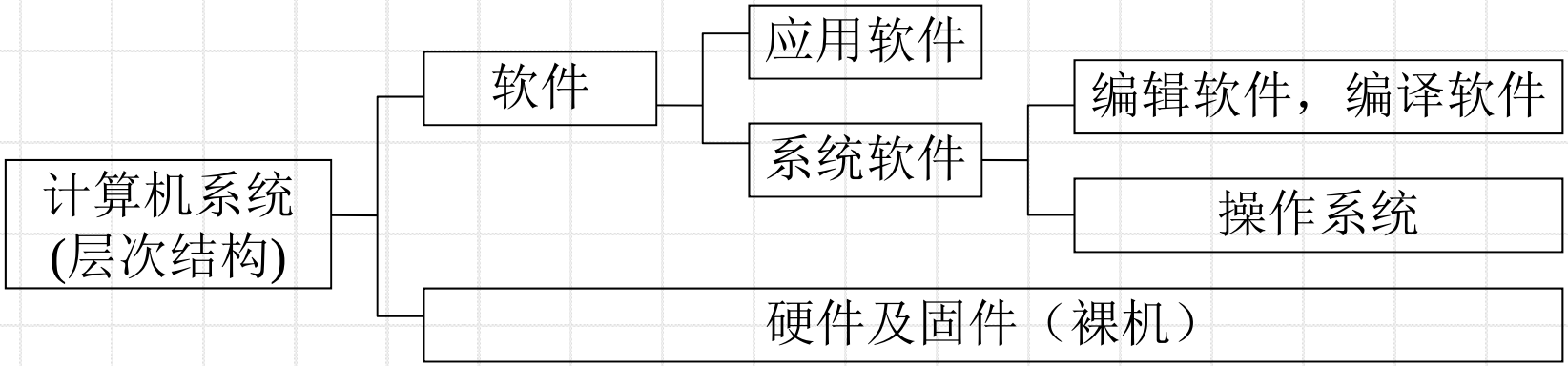
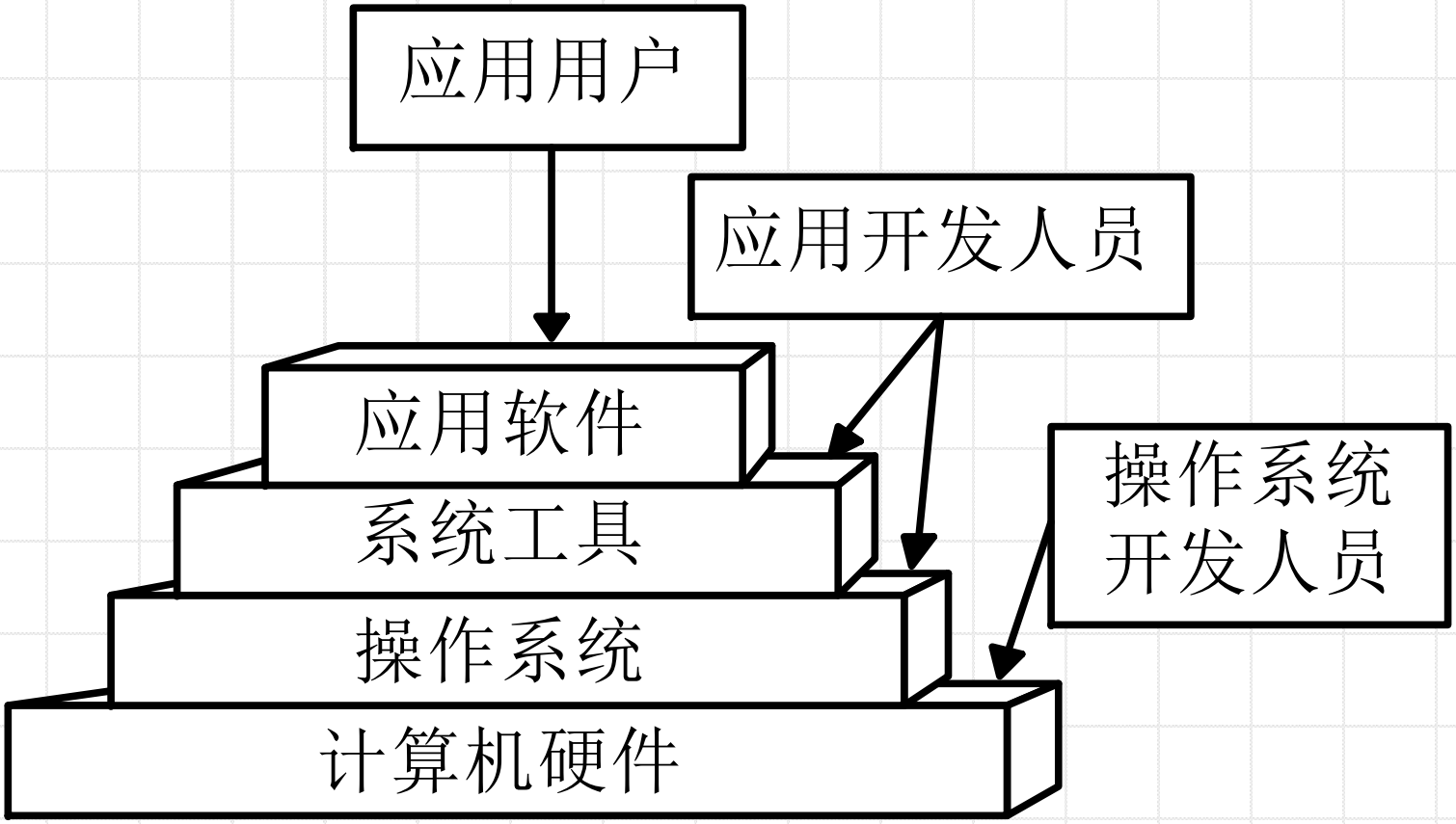


图1.1　计算机系统的组成

操作系统在计算机系统中的地位



操作系统的地位：紧贴系统硬件之上，所有其他软件之下（是其他软件的共同环境）

引入操作系统的目标

- 有效性（系统管理人员的观点）：管理和分配硬件、软件资源，合理地组织计算机的工作流程
- 方便性（用户的观点）：提供良好的、一致的用户接口，弥补硬件系统的类型和数量差别
- 可扩充性（开放的观点）：硬件的类型和规模的扩充、操作系统本身的功能和管理策略的扩充、多个系统之间的资源共享和互操作

1.1.2操作系统的作用

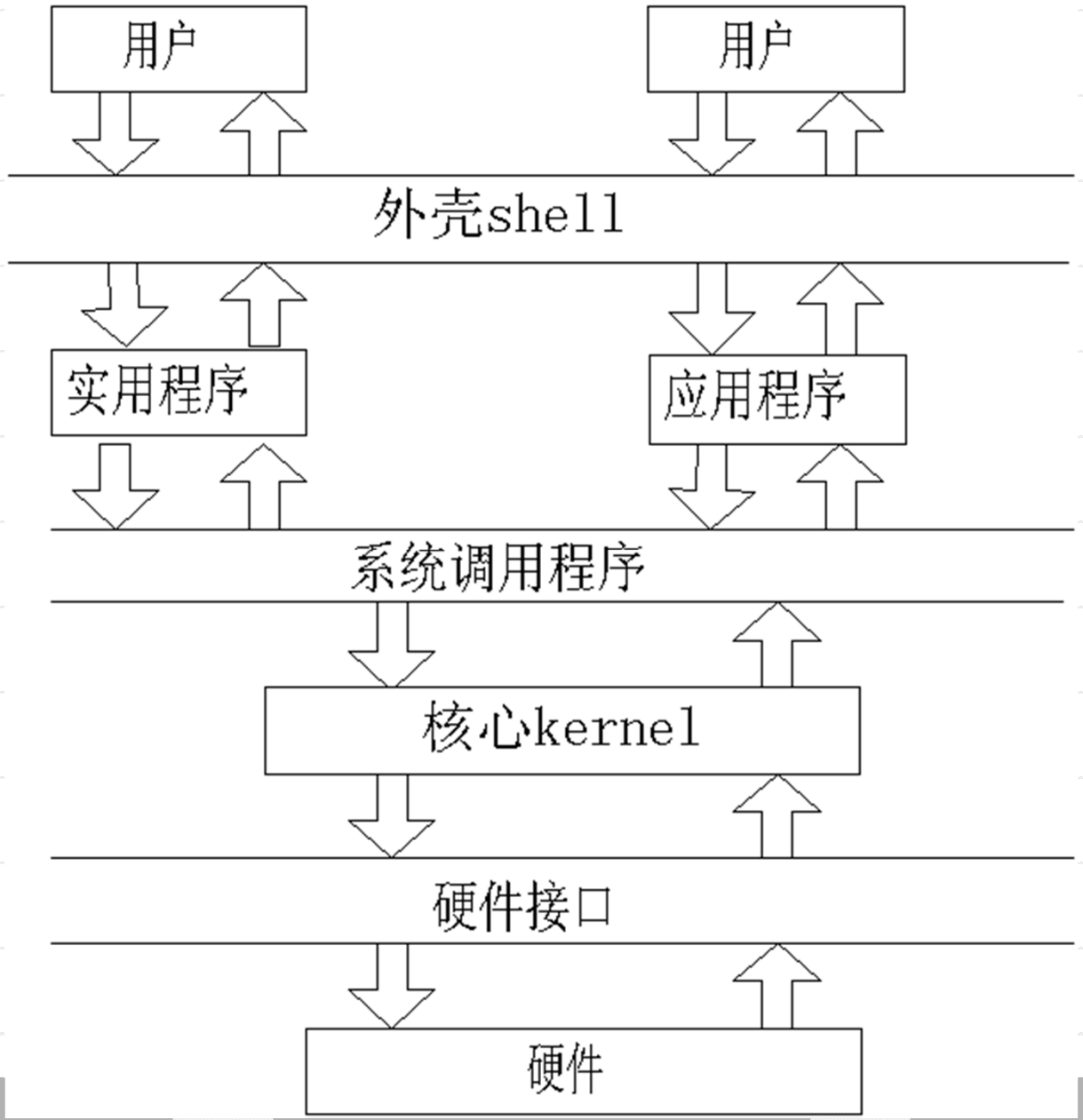
- 计算机硬件、软件管理者
  - 管理对象资源：CPU、存储器、外部设备、信息（数据和软件）；
  - 管理的内容：资源的当前状态（数量和使用情况）、资源的分配、回收和访问操作，相应管理策略（包括用户权限）。
- 用户使用软硬件的接口
  - 系统命令（命令行、菜单式、命令脚本式、图形用户接口GUI）；供一般用户使用。
  - 系统调用（形式上类似于过程调用，在应用编程中使用）；供程序员使用。

- 扩展机（extended Machine)、虚拟机(virtual machine)
  - 在裸机上添加：设备管理、文件管理、存储管理（针对内存和外存）、处理机管理（针对CPU）,让用户使用更加方便；
  - 另外，为合理组织工作流程：作业管理、进程管理。

操作系统的非形式化定义（关键点）：系统软件，程序模块的集合，资源管理和用户接口功能

操作系统的组成

- 核心(kernel)：负责管理计算机系统的资源,记录它们的状态,负责分配、使用和释放。
- 外壳(shell)：计算机系统与用户之间的接口，用户通过它来使用整个计算机。外壳也称为命令解释程序。


用户
用户
外壳shell
实用程序
应用程序
系统调用程序
核心kernel
硬件接口
硬件

1.1.3研究操作系统的几种观点

- 资源管理观点
- 抽象机层次观点
- 进程的观点
- 用户观点

操作系统的定义：操作系统是一种管理计算机系统资源，控制系统工作流程，方便用户使用计算机的系统软件。

1.1.4 操作系统举例

- MS OS: MS DOS, MS Windows 3.x, Windows 95, Windows NT, Windows 2000,
- UNIX: BSD, SRV4, OSF1, SCO UNIX, IBM AIX, SUN Solaris, Linux,HP UX, IBM OS400
- NOS: Novell Netware
- Mainframe: VSE, MVS,OS390

1.2 操作系统的发展历史

返回

1.2.1 推动操作系统发展的主要动力

“需求推动发展”

(1)器件的发展：CPU的位宽度（指令和数据）、快速外存

(2)提高资源的利用率和系统性能：计算机发展的初期，计算机系统昂贵，用作集中计算

(3)方便用户：用户上机、调试程序，分散计算时的事务处理和非专业用户（商业和办公、家庭）

1.2.2 手工操作

1946～50年代（电子管），集中计算（计算中心），计算机资源昂贵；

- 工作方式
  - 用户：用户既是程序员，又是操作员；用户是计算机专业人员；
  - 编程语言：为机器语言；
  - 输入输出：纸带或卡片；
- 计算机的工作特点
  - 用户独占全机：不出现资源被其他用户占用，资源利用率低；
  - CPU等待用户：计算前，手工装入纸带或卡片；计算完成后，手工卸取纸带或卡片；CPU利用率低；

- 主要矛盾
  - 计算机处理能力的提高，手工操作的低效率（造成浪费）；
  - 用户独占全机的所有资源；
- 提高效率的途径
  - 专门的操作员，批处理

1.2.3 单道批处理系统 (simple batch processingm, ,uniprogramming)

50年代末～60年代中（晶体管）：利用磁带把若干个作业分类编成作业执行序列，每个批作业由一个专门的监督程序（Monitor）自动依次处理。可使用汇编语言开发。

- 批处理中的作业的组成：
  - 用户程序
  - 数据
  - 作业说明书（作业控制语言）
- 批：
  - 供一次加载的磁带或磁盘，通常由若干个作业组装成，在处理中使用一组相同的系统软件（系统带）

两种批处理方式(1)
联机批处理

- 用户提交作业：以纸带或卡片为介质；
- 操作员合成批作业：结果为磁带介质；
- 批作业处理：对批作业中的每个作业进行相同的处理：
  - 从磁带读入用户作业和编译链接程序，编译链接用户作业，生成可执行程序；
  - 启动执行；
  - 执行结果输出。
- 这时的问题：慢速的输入输出处理仍直接由主机来完成。输入输出时，CPU处于等待状态。

两种批处理方式(2)
脱机批处理

利用卫星机完成输入输出功能。主机与卫星机可并行工作。

- 卫星机：完成面向用户的输入输出（纸带或卡片），中间结果暂存在磁带或磁盘上。
- 作业控制命令由监督程序(monitor)来执行，完成如装入程序、编译、运行等操作。
- 优点：同一批内各作业的自动依次更替，改善了主机CPU和I/O设备的使用效率，提高了吞吐量。
- 缺点：磁带或磁盘需要人工装卸，作业需要人工分类，监督程序易遭到用户程序的破坏（由人工干预才可恢复）。

通道和中断技术

60年代初，发展了通道技术和中断技术，这些技术的出现使监督程序在负责作业运行的同时提供I/O控制功能。

- 通道：用于控制I/O设备与内存间的数据传输。启动后可独立于CPU运行，实现CPU与I/O的并行。
  - 通道有专用的I/O处理器，可与CPU并行工作
  - 可实现 I/O联机处理
- 中断是指CPU在收到外部中断信号后，停止原来工作，转去处理该中断事件，完毕后回到原来断点继续工作。
  - 中断处理过程：中断请求，中断响应，中断点（暂停当前任务并保存现场），中断处理例程，中断返回（恢复中断点的现场并继续原有任务
  - 可处理算术溢出和非法操作码，死循环（利用时钟中断进行超时限定）
- 监督程序发展为执行系统(executive system)，常驻内存

单道批处理的主要问题

- CPU和I/O设备使用忙闲不均（取决于当前作业的特性）。
  - 对计算为主的作业，外设空闲；
  - 对I/O为主的作业，CPU空闲；

1.2.4 多道批处理系统 (multiprogramming system)

60年代中～70年代中（集成电路），利用多道批处理提高资源的利用率。

- 多道批处理的运行特征
  - 多道：内存中同时存放几个作业；
  - 宏观上并行运行：都处于运行状态，但都未运行完；
  - 微观上串行运行：各作业交替使用CPU；

在当前运行的作业需作I/O处理时，CPU转而执行另一个作业。

多道批处理系统的特点

- 优点：
  - 资源利用率高：CPU和内存利用率较高；
  - 作业吞吐量大：单位时间内完成的工作总量大；
- 缺点：
  - 用户交互性差：整个作业完成后或中间出错时，才与用户交互，不利于调试和修改；
  - 作业平均周转时间长：短作业的周转时间显著增长；

批处理：交互性差——提高对CPU利用率；
分时处理：用户与应用程序随时交互，控制程序运行，适于商业和办公事务处理——缩短响应时间

多道批处理技术

- 同步与互斥
  - 资源的共享与竞争
- 内存管理技术
  - 覆盖技术、虚拟内存技术、对换技术
- 内存保护技术
  - 保护各程序所占用的内存

1.2.5 分时系统 (time-sharing system)

70年代中期至今

- “分时”的含义分时是指多个用户分享使用同一台计算机。多个程序分时共享硬件和软件资源。
  - 多个用户分时：单个用户使用计算机的效率低，因而允许多个应用程序同时在内存中，分别服务于不同的用户。有用户输入时由CPU执行，处理完一次用户输入后程序暂停，等待下一次用户输入——时走时停
  - 前台和后台程序(foreground & background)分时：后台程序不占用终端输入输出，不与用户交互——现在的图形用户界面(GUI)，除当前交互的程序(输入焦点)之外，其他程序均作为后台
- 通常按时间片(time slice)分配：各个程序在CPU上执行的轮换时间。

抢先式和非抢先式调度

- 抢先式和非抢先式(preemptive & non-preemptive)：取决于程序出让CPU是OS强迫还是程序主动的.
  - 抢先式：OS强迫程序出让CPU；
  - 非抢先式：只能由程序主动出让CPU；

分时系统的特点

- 人机交互性好：在调试和运行程序时由用户自己操作。
- 共享主机：多个用户同时使用。
- 用户独立性：对每个用户而言好象独占主机。

现在的许多操作系统都具有分时处理的功能，在分时系统的基础上，操作系统的发展开始分化，如实时系统、通用系统、个人系统等。

1.2.6 实时系统(real-time system)

用于工业过程控制、军事实时控制、金融等领域，包括实时控制、实时信息处理

- 要求：要求响应时间短，必须在一定范围之内；系统可靠性高
- 硬实时：用于过程控制。如武器系统，工业生产。其时间要求可达到毫秒以至微秒级。
- 软实时：用于事务处理。如联网订票系统、银行交易系统。一般要求秒级或毫秒级。

目前的操作系统，通常具有分时、实时和批处理功能，又称作通用操作系统。可适用于计算、事务处理等多种领域，能运行在多种硬件平台上，如 UNIX系统、Windows NT等。——通用化、小型化

1.3 操作系统的分类

操作系统分类主要讨论操作系统的内部特征。

1.3.1 批处理操作系统

1.3.2 分时操作系统

1.3.3 实时操作系统

1.3.4 多处理操作系统

1.3.5 网络操作系统

1.3.6 分布式操作系统

1.3.7 个人计算机操作系统

返回

1.3.1 批处理操作系统 (Batch Processing Operation System)

- 作业的处理流程（四种状态）
  - 作业提交：作业输入到磁盘后备区；
  - 作业收容：输入完毕，等待执行；
  - 作业执行：分配资源，执行；
  - 作业完成：作业的输出，释放资源；

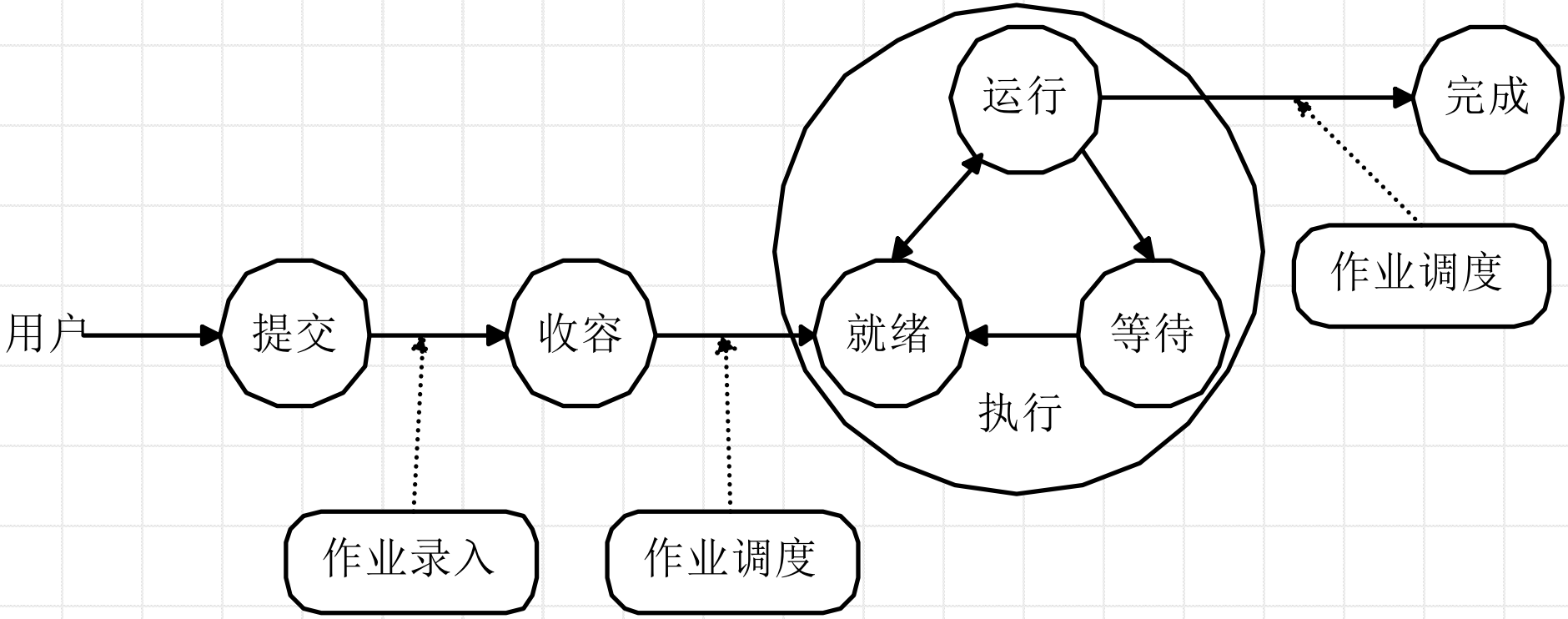


图1 批处理系统中作业处理及状态

单道(uniprogramming)和多道批处理的比较

	单道	多道
内存使用	每次一个作业	每次多个作业（充分利用内存）
作业次序	顺序，先进先出	无确定次序

批处理的主要特征

- 用户脱机使用计算机：作业提交后直到获得结果之前，用户无法与作业交互。
- 作业成批处理

（以下仅限多道批处理）

- 多道程序并行：充分利用系统资源。

多道批处理系统上的技术

- 作业调度：作业的现场保存和恢复——上下文切换
- 资源共享：资源的竞争和同步——互斥(exclusion)和同步(synchronization)机制
- 内存使用：提高内存使用效率（为当前由CPU执行的程序提供足够的内存）——覆盖(overlay)，交换(swap)和虚拟存储(virtual memory)
- 内存保护：系统存储区和各应用程序存储区不可冲突——存储保护
- 文件非顺序存放、随机存取

1.3.2 分时操作系统 (Time Sharing Operating System)

分时的定义

把计算机的系统资源（尤其是CPU时间）进行时间上的分割，每个时间段称为一个时间片（time slice），每个用户依次轮流使用时间片。

分时系统的特征

- 多用户：多个用户同时工作。
  - 共享系统资源，提高了资源利用率。节省维护开支，可靠性高：终端——至今仍在使用。促进了计算机的普遍应用，提高资源利用率：远地用户通过终端（较便宜）联机使用。
- 独立性：各用户独立操作，互不干扰。
- 交互性：系统能及时对用户的操作进行响应，显著提高调试和修改程序的效率：缩短了周转时间。

1.3.3 实时操作系统 (Real Time Operating System)

实时操作系统主要用于过程控制、事务处理等有实时要求的领域，其主要特征是实时性和可靠性。

- 实时系统的特征
  - 及时性：对用户的请求或外部事件处理必须在指定时间内完成。
  - 过载保护：缓冲区排队，丢弃某些任务，动态调整任务周期；
    - 过载是指进入系统的任务数目超出系统的处理能力。
  - 高度可靠性和安全性：容错能力（如故障自动复位）和冗余备份（双机，关键部件）；
  - 专用性：它与应用程序以及外界环境关系密切。

1.3.4 多处理操作系统 (Multi-processor Operating System)

多处理操作系统的出现是为了提高计算机系统性能和可靠性。提高性能有两条途径：提高各个组成部分的速度、增大处理的并行程度。1975年前后，出现多处理机系统(multi-processor)。

- 多处理机系统的特点
  - 增加系统的吞吐量：N个处理器加速比达不到N倍（额外的调度开销，算法的并行化）
  - 提高系统可靠性：故障时系统降级运行

多处理机系统的类型

- 紧密耦合(tightly-coupled)：各处理机之间通过快速总线或开关阵列相连，共享内存，整体系统由一个统一的OS管理（一个OS核心）。
- 松散耦合(loosely-coupled)：各处理机带有各自的存储器、I/O设备和操作系统，通过通道或通信线路相连。每个处理机上独立运行OS。

多处理操作系统的类型

要运行OS本身、I/O处理（如中断响应）、应用程序

- 非对称式多处理(Asymmetric Multiprocessing, ASMP)：又称主从模式(Master-slave mode)。
  - 主处理器：只有一个，运行OS。管理整个系统的资源，为从处理器分配任务；
  - 从处理器：可有多个，执行应用程序或I/O处理。
  - 特点：不同性质任务的负载不均，可靠性不够高，不易移植（通常要求硬件也是"非对称"）。
- 对称式多处理(Symmetric Multiprocessing, SMP)：OS交替在各个处理器上执行。任务负载较为平均，性能调节容易——"傻瓜式"

1.3.5 网络操作系统 (NOS, Network Operating System)

网络操作系统是在通常**操作系统功能**的基础上提供**网络通信和网络服务功能**的操作系统。网络操作系统为网上计算机进行方便而有效的**网络资源共享**，提供网络用户所需各种服务的软件和相关规程的集合。

网络功能与操作系统的结合程度是网络操作系统的重要性能指标。早期的作法是通常操作系统**附加网络软件**，过渡到网络功能成为操作系统的**有机组成部分**。它们的区别在于：网络功能的强弱、使用是否方便等。

网络操作系统的功能

- 通常操作系统的功能：处理机管理、存储器管理、设备管理、文件管理等；
- 网络通信功能：通过网络协议进行高效、可靠的数据传输；
- 网络资源管理：协调各用户使用；
- 网络服务：文件和设备共享，信息发布；
- 网络管理：安全管理、故障管理、性能管理等；
- 互操作：直接控制对方比交换数据更为困难；

1.3.6 分布式操作系统 (Distributed Operating System)

分布式系统：处理和控制的分散（相对于集中式系统）

分布式系统是以计算机网络为基础的，它的基本特征是处理上的分布，即功能和任务的分布。

分布式操作系统的所有系统任务可在系统中任何处理机上运行，自动实现全系统范围内的任务分配并自动调度各处理机的工作负载。
LOAD BALANCE

分布式操作系统与网络操作系统的比较

- 耦合程度：
  - 分布式系统是紧密耦合系统：分布式OS是在各机上统一建立的"OS同质"，直接管理CPU、存储器和外设；统一进行全系统的管理；
  - 网络通常容许异种OS互连，各机上各种服务程序需按不同网络协议"协议同质"。
- 并行性：
  - 分布式OS可以将一个进程分散在各机上并行执行"进程迁移"；
  - 网络则各机上的进程独立。
- 透明性：用户是否知道或指定资源在哪个机器上（如CPU、内存或外设）。
  - 分布式系统的网络资源调度对用户透明，用户不了解所占有资源的位置；
  - 网络操作系统中对网络资源的使用要由用户明确指定；
- 健壮性：分布式系统要求更强的容错能力（工作时系统重构）

1.3.7 个人计算机操作系统 (Personal Computer Operating System)

针对单用户使用的个人计算机进行优化的操作系统。

- 个人计算机操作系统的特征
  - 应用领域：事务处理、个人娱乐，
  - 系统要求：使用方便、支持多种硬件和外部设备（多媒体设备、网络、远程通信）、效率不必很高。
- 常用的个人计算机操作系统
  - 单用户单任务：MS DOS
  - 单用户多任务：OS/2, MS Windows 3.x, Windows 95, Windows NT, Windows 2000 Professional
  - 多用户多任务：UNIX(SCO UNIX, Solaris x86, Linux, FreeBSD)

1.4 操作系统的功能

1.4.1 处理机管理

1.4.2 存储管理

1.4.3 设备管理

1.4.4 信息管理

1.4.5 用户接口

返回

1.4.1 处理机(CPU)管理—实质是对进程的管理

完成处理机资源的分配调度等功能。处理机调度的单位可为进程或线程。

- 进程控制：创建、撤销、挂起、改变运行优先级等——主动改变进程的状态
- 进程同步与互斥：协调并发进程之间的推进步骤，以协调资源共享；——交换信息能力弱
- 进程通信：进程之间传送数据，以协调进程间的协作；——交换信息能力强，也可以用来协调进程之间的推进
- 进程调度：进程的运行切换，以充分利用处理机资源和提高系统性能

1.4.2 存储管理

管理目标：提高利用率、方便用户使用、提供足够的存储空间、方便进程并发运行。

- 存储分配与回收
- 存储保护：保证进程间互不干扰、相互保密；如：访问合法性检查、甚至要防止从"垃圾"中窃取其他进程的信息；
- 地址映射（变换）：进程逻辑地址到内存物理地址的映射；
- 内存扩充（覆盖、交换和虚拟存储）：逻辑上的扩充，提高内存利用率、扩大进程的内存空间；

1.4.3 设备管理

设备管理的目标是：方便的设备使用、提高CPU与I/O设备利用率；

- 设备操作：利用设备驱动程序（通常在内核中）完成对设备的操作。
- 设备分配与回收：在多用户间共享I/O设备资源。
  - 虚拟设备(virtual device)：设备由多个进程共享，每个进程如同独占该设备。
- 缓冲区管理：匹配CPU和外设的速度，提高两者的利用率

1.4.4 文件管理(信息管理)

解决软件资源的存储、共享、保密和保护，操作系统中负责这一功能的部分称为文件系统。

- 文件存储空间管理：解决如何存放信息，以提高空间利用率和读写性能。
- 目录管理：解决信息检索问题。
- 文件的读写管理和存取控制：解决信息安全问题。系统设口令：“哪个用户”、用户分类：“哪个用户组”、文件权限：“针对用户或用户组的读写权”

1.4.5用户接口

目标：提供一个友好的用户访问操作系统的接口。操作系统向上提供两种接口：

- 系统命令：供用户用于组织和控制自己的作业运行。命令行、菜单式或GUI"联机"；命令脚本"脱机"
- 编程接口：供用户程序和系统程序调用操作系统功能：系统调用。

1.5 操作系统的结构

随着操作系统的发展，功能越强，OS自身代码量越大——采用良好的结构：有利于保证正确性以及自身修改和扩充。

1.5.1模块化结构

1.5.2 分层结构或虚拟机

1.5.3 客户/服务器模型或微内核结构

返回

操作系统的原则

- 可维护性：容易修改与否称为可维护性；有三种可能的维护：
  - 改错性维护：改正已发现的错误；
  - 适应性维护：修改软件，使之适应新的运行环境（硬件环境和软件环境）；如：操作系统的移植。
  - 完善性维护：增加新功能；
- 可靠性：可靠性包括两方面：
  - 正确性：正确实现所要求的功能和性能；
  - 稳健性：对意外（故障和误操作）作出适当的处理；
- 可理解性：易于理解，以方便测试、维护和交流；
- 性能：有效地使用系统资源；尽可能快地响应用户请求；

1.5.1模块化结构

整个系统按功能进行设计和模块划分。系统是一个单一的、庞大的的软件系统。这种结构思想来源于服务功能观点，而不是资源管理的观点。

- 模块结构的特点：模块由众多服务过程（模块接口）组成，可以随意调用其他模块中的服务过程
  - 优点：具有一定灵活性，在运行中的高效率
  - 缺点：功能划分和模块接口难保正确和合理；模块之间的依赖关系（功能调用关系）复杂（调用深度和方向），降低了模块之间的相对独立性——不利于修改

1.5.2 分层结构或虚拟机 layered system or virtual machine

从资源管理观点出发，划分层次。在某一层次上代码只能调用低层次上的代码，使模块间的调用变为有序性。系统每加一层，就构成一个比原来功能更强的虚拟机。有利于系统的维护性和可靠性。

分层结构的特点

- 优点：
  - 功能明确，调用关系清晰（高层对低层单向依赖），有利于保证设计和实现的正确性
  - 低层和高层可分别实现（便于扩充）；高层错误不会影响到低层；避免递归调用
- 缺点：降低了运行效率

各系统对具体划分多少层次有不同的看法。

分层原则

- 被调用功能在低层：如文件系统管理——设备管理——设备驱动程序
- 活跃功能在低层：提高运行效率
- 资源管理的公用模块放在最低层：如缓冲区队列、堆栈操作
- 存储器管理放在次低层：便于利用虚拟存储功能
- 最低层的硬件抽象层：与机器特点紧密相关的软件放在最低层。如Windows NT中的HAL——单处理、多处理
- 资源分配策略放在最上层，便于修改或适应不同环境

调用跨越的层次：相邻层（最严格）、所有下层、部分下层

1.5.3 客户/服务器模型或微内核结构 client-server model or microkernel

把操作系统分成若干分别完成一组特定功能的服务进程，等待客户提出请求；而系统内核只实现操作系统的基本功能(如：虚拟存储、消息传递)。

- 微内核(micro-kernel)：将更多操作系统功能放在核心之外，作为独立的服务进程运行；
  - 服务进程（或称作“保护子系统”）
  - 客户进程（系统客户和应用客户）——需支持多进程
- 本地过程调用 (LPC, Local Procedure Call)：一种进程之间请求-应答式的消息（Message）传递机制。
- 消息：是一定格式的数据结构。①发起调用，送出请求消息②请求消息到达并进行处理③送出回答消息④整理回答消息，返回结果；如：对文件create, read, write

微内核模式的特点

- 优点：
  - 良好的扩充性：只需添加支持新功能的服务进程即可
  - 可靠性好：调用关系明确，执行转移不易混乱
  - 便于网络服务，实现分布式处理：以同样的调用形式，在下层可通过核心中的网络传送到远方服务器上 (远地过程调用 RPC, Remote Procedure Call)
- 缺点：
  - 消息传递比直接调用效率要低一些 (但可以通过提高硬件性能来补偿)

RPC的过程：RPC应用程序－－RPC Stub(client)－－Network－－RPC Server－－进行本地调用

1.6 常用的操作系统

1.6.1 MS DOS

1.6.2 MS Windows 3.x, Windows 95, Windows NT, Windows 2000

1.6.3 UNIX

返回

1.6.1 MS DOS

IBM PC, CPU 8088/8086, BIOS
单用户单任务，简单分层结构，16位

MS DOS的历史

- 1981年：PC-DOS 1.1：IBM PC，只支持软盘的个人操作系统；
- 1983年：DOS2.0：PC XT，支持硬盘和目录的层次结构，并提供丰富的系统命令；
- 1984年：DOS3.0：PC AT (Intel 80286 CPU)，它把286作为一个快速的8086使用；
- 1987年：DOS3.3：提供对IBM PS/2的支持（如3.5"软驱），提供了更多的应用；
- 1988年：DOS4.0：支持大于32M的硬盘；
- 1991年：DOS5.0：改进对扩展内存的支持；

MS DOS的结构

- DOS BIOS(Basic Input/Output System)：由一组与硬件相关的设备驱动程序组成，实现基本的输入/输出功能；
- DOS核心：提供一套独立于硬件的系统功能：内存管理、文件管理、字符设备和输入/输出、实时时钟等；
- 命令处理程序：对用户命令进行分析和执行；

MS DOS的特点

- 字符用户界面。作业管理：命令行，批处理程序（BAT文件），菜单式。编程时通过软中断调用(int 21h)来使用系统功能。不区分用户。
- "准多任务"：通过内存驻留程序TSR(Terminated and Stay Resident)来实现，通过时钟中断或键盘中断"热键hotkey"来激活其他任务。
- 不支持虚拟存储，没有存储保护。采用段式分配(内存块)，可直接访问的最大地址空间为1MB。其余的内存只能通过作为扩展内存(XMS)或扩充内存(EMS)来使用。
  - XMS是段式分配，通过内存数据搬移来使用XMS区域
  - EMS是页式分配，通过页面的映射来使用EMS区域
  - 或者用支持保护方式的编程工具
- 文件系统为FAT(File Allocation Table)格式（磁盘卷，多级目录，文件名 8+3 个字符；分区容量最大为2GB）；有文件属性，没有区分用户的访问权限保护。
- 设备驱动程序在系统起动时加载。分为字符设备和块设备。

1.6.2 MS Windows 3.x, Windows 95, Windows NT, Windows 2000

CPU 80386
单用户多任务（分时系统），16位/16和32位混合/32位

Windows的历史

- 1990年：Windows 3.0（成功版本），16位OS，借见Apple Macintosh给出友好的用户界面；
- 1993年：Windows NT 3.1, 32位OS，支持DOS和Windows应用程序；
- 1999年12月：Windows 2000(Professional, Server, Advanced Server)，32位OS；

Windows 2000的特点

- 支持对称多处理机
- 真正的32位操作系统：除16位应用的支持代码，没有16位的代码；
- 完全的代码可重入(reentrant)：同一段代码可由多个应用同时访问；
- 图形用户界面GUI（和字符用户界面）。
- 抢先式多任务和多线程。支持动态链接。
- 虚拟存储：段页式(有存储保护)。
- 兼容16位Windows应用。
- 文件系统：NTFS(HPFS)　NFS，支持安全控制。
- 设备驱动程序：VxD(virtual driver)。
- 可移植：适用于多种硬件平台。
- 容错能力。
- 面向对象特性：用对象来表示所有资源。

1.6.3 UNIX

多用户多任务，16/32/64位

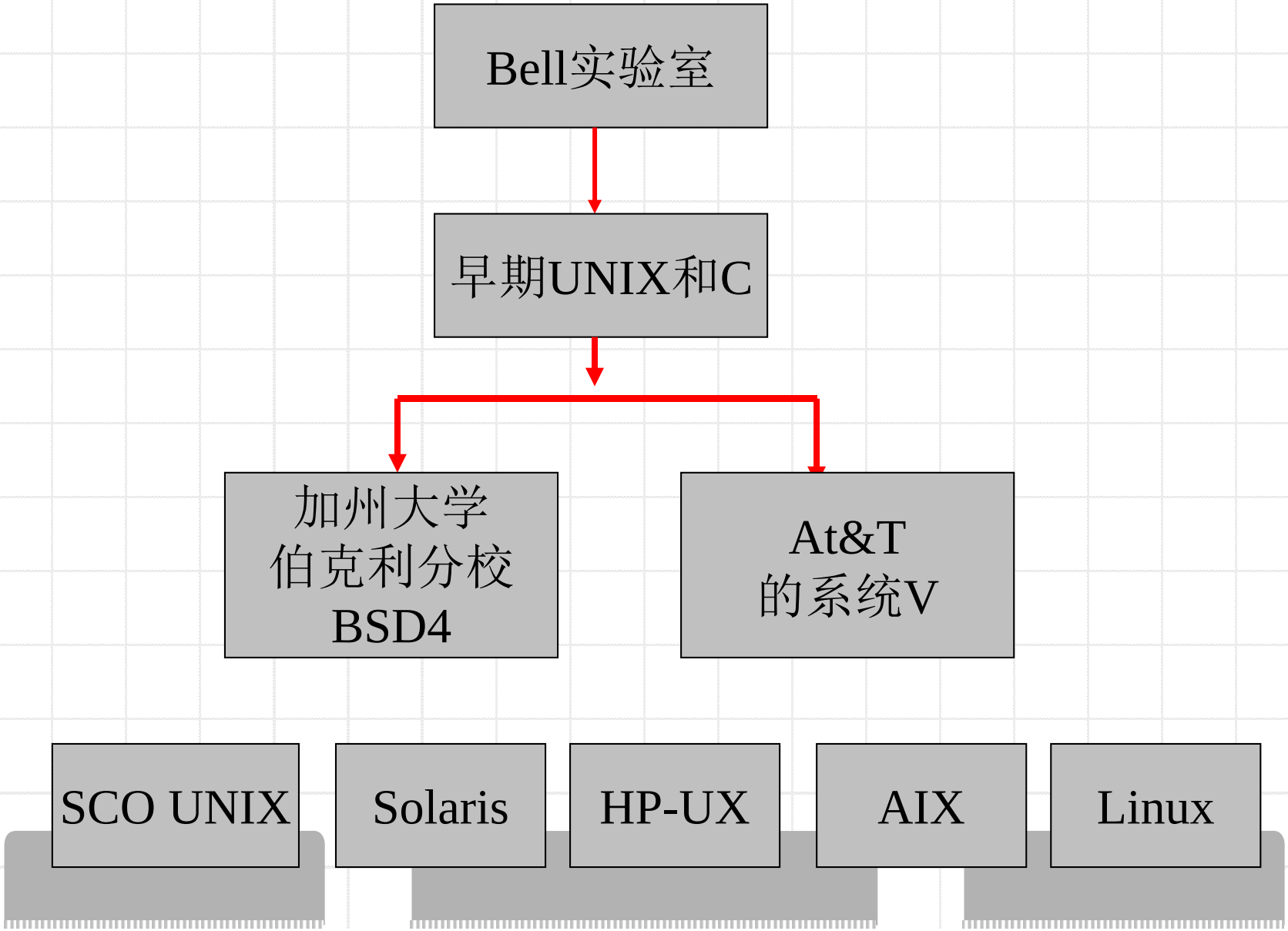
，

BSD, SVR4（模块式结构）, OSF/1（微内核结构）

UNIX的历史

- 1965年：MIT的Multics，由于规模和进展而没有达到目标；
- 1969年：AT&T，PDP-11上的16位操作系统；
- 1974年：UNIX系统正式发表(第五版)，在大学得到使用和好评；
- 1980年：University of California at Berkeley为VAX11发表BSD4.0；以后，UNIX就以AT&T和Berkeley为主分别开发，有多种变种；
- 1989年：UI (UNIX International)发表UNIX system V Res4.0；使BSD和System V在用户界面上统一；
- 1991年芬兰大学生Linus Benedict Torralds开发了第一个Linux版本。
- 1994年：Linux 1.0，现在的最新内核版本是2.4

UNIX系统：可运行UNIX应用软件的操作系统。


Bell实验室
早期UNIX和C
加州大学
伯克利分校
BSD4
At&T
的系统V
SCO UNIX
Solaris
HP-UX
AIX
Linux

UNIX系统的特点

- 字符用户界面和图形用户界面GUI(X Window)。
- 抢先式多任务，多线程。支持动态链接。支持对称式多处理。
- 虚拟存储：段页式，有存储保护。
- 文件系统：多级目录，文件卷可以在子目录下动态装卸。无文件属性，可有别名(NFS)。
- 采用设备文件的形式（读写，参数控制）。设备驱动程序修改后需要重新编译连接生成内核。
- 支持多种硬件平台。
- 易移植：主要代码用C语言写成；
- 变种很多，很难标准化。

小结

- OS地位、目的、作用和组成
- OS发展：主要动力
- OS分类：批处理、分时、实时、（通用）、多处理、网络和分布式、PC
- OS的结构：模块——层次——Client-Server
- OS的特征和服务
- OS功能

作业

- 由以下任选一题
  - 什么是操作系统的基本功能？
  - 什么是批处理、分时和实时操作系统，及其特征？

第二章 作业管理和用户接口

在这一章中，我们讨论OS向上提供的用户接口，即系统命令接口和系统调用接口。系统命令接口可完成用户作业的组织和控制。

2.1 作业组织和控制

2.2 作业管理举例

2.3 系统调用(SYSTEM CALL)

2.1 作业组织和控制

返回

2.1.1 作业和作业处理过程

1. 作业的概念
2. 作业的组成
3. 作业的处理过程
4. 作业输入方式
5. 作业控制表(JCB, Job Control Block)

1. 作业的概念

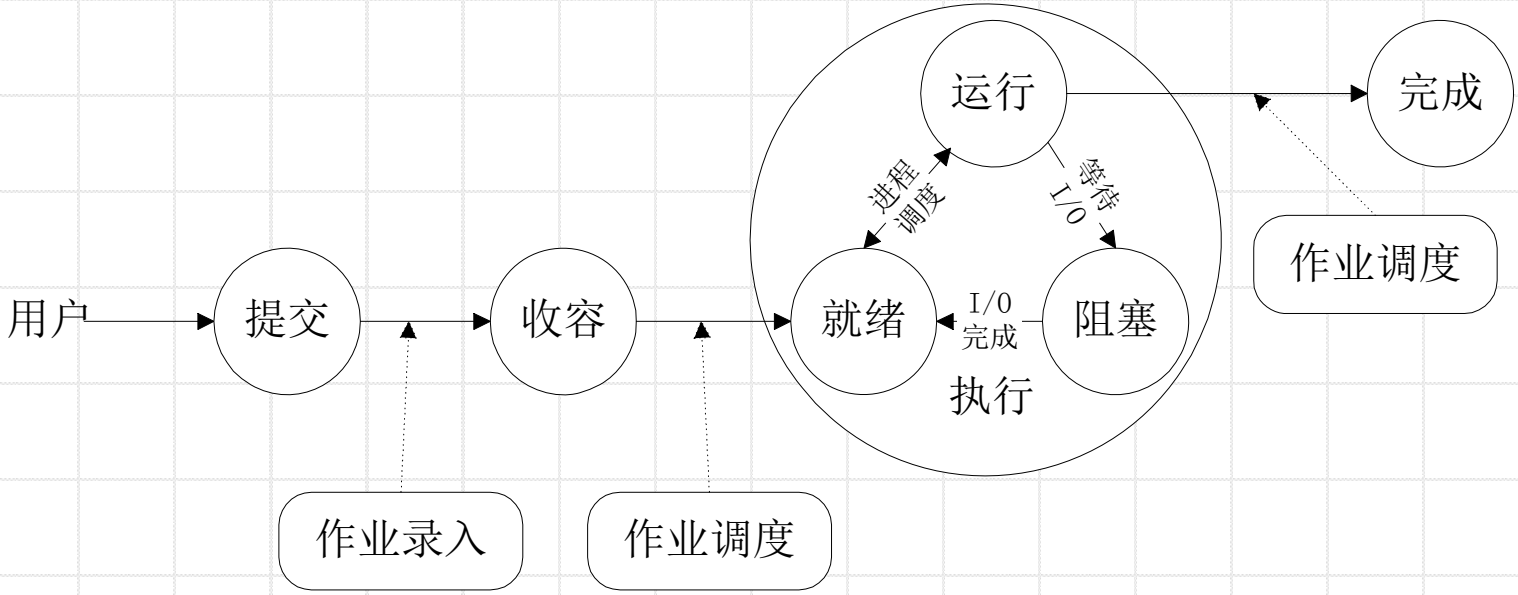
- 一个作业是指在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作。
  - 用户的观点：在一次业务处理过程中，从输入程序和数据到输出结果的全过程。作业步：形成中间结果文件。
  - 系统的观点（针对作业进行资源分配）：作业由程序及数据（作业体）和作业说明书（作业控制语言）
- 作业由不同的顺序相连的作业步组成。
- 作业步是在一个作业的处理过程中，计算机所做的相对独立的工作。

2. 作业的组成

作业由程序、数据和作业说明书三部分组成。

- 程序数据是用户要求的业务处理工作
- 作业说明书包括作业基本情况、作业控制、作业资源要求的描述；它体现用户的控制意图。如：预计运行时间、要求的资源情况、执行优先级等。
  - 作业基本情况：用户名、作业名、编程语言、最大处理时间等；
  - 作业控制描述：作业控制方式、作业步的操作顺序、作业执行出错处理；
  - 作业资源要求描述：处理时间、优先级、内存空间、外设类型和数量等；

3. 作业的处理过程



- 作业提交：
  - 作业的输入；
- 作业执行：
  - 先到"就绪"，经调度"运行"，有I/O请求"等待"，I/O完成到"就绪"
- 作业完成：
  - 作业的输出；

4. 作业输入方式

- 无通道处理方式：
  - 联机（降低了CPU效率）：由主机直接控制输入/输出；I/O与作业处理不能并行。
  - 脱机（人工干预）：通过磁带或磁盘在外围处理机与主机之间交换作业（需人工移动）；用于主机不太快的情况。
  - 直接耦合：通过共享外存在外围处理机与主机之间交换作业；用于高速主机。
- 有通道处理：输入/输出由主机和通道来承担。
  - 假脱机(spooling, Simultaneously Peripheral Operation On Line)：系统把作业处理的全过程划分为相对独立的三个部分--输入流、处理流和输出流。spooling-in/spooling-out进程：控制输入/输出；
  - 网络：在网络上一台机器进行作业输入/输出，在另一台主机上运行；

5. 作业控制表(JCB, Job Control Block)

在运行过程中，系统对作业进行管理的必要信息。

- 作业名
- 估计执行时间
- 优先数（用于调度）
- 作业说明书文件名
- 程序类型（需调用的系统程序）
- 资源要求：（静态，或中间可以随作业步变化——效率不高；动态分配
- 作业状态：提交、后备、执行、就绪、等待、完成；

2.1.2 作业调度

检查系统是否满足作业的资源要求，并一定算法选取作业。作业调度也称为宏观调度。

- 作业调度算法的评价因素
  - 作业吞吐量：运行尽可能多的作业；
  - 充分利用资源：CPU忙、I/O设备忙；
  - 对各作业公平、合理，使用户满意：执行时间长短、等待时间等；

2. 作业调度算法

实际的算法可能会是多种算法的综合。

- 先来先服务（FCFS）：按照作业进入系统的先后次序进行调度，先进入系统者先调度；即启动等待时间最长的作业。
  - 优点：实现简单、公平
  - 缺点：没考虑资源利用率和作业的特殊性
- 短作业优先（SJF）：以要求运行时间长短进行调度，即启动要求运行时间最短的作业。
  - 优点：易于实现，强调了资源的充分利用，保证了系统的最大吞吐量（单位时间里处理作业的个数）。
  - 缺点：不公平，会造成长作业长期等待。
  - 结论：假设系统中所有作业同时到达，可以证明采用SJF能得到最短的作业平均周转时间。

高响应比优先（HRF）：响应比最高的作业优先启动。

响应比=（等待时间+估计运行时间）/ 估计运行时间

该算法是FCFS和SJF的结合，克服了两种算法的缺点

优点: 公平，吞吐率大

缺点: 增加了计算，增加了开销

高优先级优先：由用户指定作业优先级，优先级高的作业先启动。

资源均衡型调度：把作业分类，作业调度从不同类型作业中去调度作业

根据作业对资源要求分类：I/O型、CPU型和均衡型

2.1.3 作业控制语言

脱机作业控制：用户输入作业说明书，整个作业的运行由系统控制。

联机作业控制：通过人-机会话方式控制作业运行。用户登录（控制台登录或远程登录），由系统自动执行一些命令脚本后，并进入shell（字符或GUI界面），接受用户的命令和操作，最后退出系统。

1. 命令行
2. 环境变量

1. 命令行

- 命令行：一行可有一个或多个命令，每次一行，包含一个或多个命令。
  - shell给出提示符时可输入，以回车键提交。如：
  - “ls -a -l”列出当前目录文件列表；
  - "gunzip mp1.tar.gz; tar -xvf mp1.tar; rm -r -f mp1.tar"为解压缩后再展开。
- 命令格式：一个命令可有命令参数，格式包括选项/开关 (option/switch)或参数(argument)。
  - 如UNIX系统： cp -r doc /tmp——argv[0], argv[1], ...（含子目录的文件复制：/tmp为目标地址）

命令分类：内部命令和外部命令

内部命令：直接由shell本身完成，功能简单、使用频繁；如：DOS的copy命令。

外部命令：运行相应的可执行文件，在使用时加载。如：DOS的xcopy命令。

命令简化：利用参数替换可简化命令输入，通配符(?, *)用于匹配一组文件名

如：UNIX的cp命令：当前目录上有两个"1.tar"和"2.tar"时，"cp *.tar /tmp"等同于"cp 1.tar /tmp; cp 2.tar /tmp"

2. 环境变量

环境变量(environment variable)——应用进程地址空间中的特殊变量区。

- 环境变量也可以作为shell参数，如命令提示符的式样，外部命令的查找目录路径等。
  - 如：PATH=/bin:/usr/bin:/usr/sbin:.
  - HOME=/home/thisuser
- 环境变量是传递命令参数的另一种途径。如：
  - cd $HOME; =cd /home/thisuser
- 环境变量可按名字访问，可以新建、赋值或撤销。
- 有效范围：只对本进程里的环境变量能够直接进行操作。此外，在执行新进程时（UNIX中是exec()调用），可以给出环境变量的初始值，通常就是直接复制当前进程的所有环境变量——继承。

2.2 作业管理举例

2.2.1 MS DOS的作业管理

2.2.2 UNIX的作业管理

返回

2.2.1 MS DOS的作业管理

2.2.1.1. DOS命令处理程序

command.com（或其他程序如DOS shell）：驻留内存，在系统运行期间不再退出。为了给应用程序的执行提供更大内存空间，又分为常驻部分和暂驻部分（可被应用程序覆盖）

- 命令分类：
  - 内部命令：如dir, cd, copy
  - 外部命令：如format, xcopy
- 命令行选项
  - 命令行选项通常是：/option；如："/?"选项可显示各命令的命令行选项列表。
  - 通配符：由外部命令自己处理。如：xcopy *.c . ——argv[1]="*.c"

- 输入输出重定向和管道(pipe)
  - <, >, >>, |, 基于临时文件
  - "<"为输入重定向，如："find "string" < temp.txt"将显示文件"temp.txt"中有"string"串的行；"more < temp.txt"将逐屏显示输出文件"temp.txt"的内容；
  - “>”为输出重定向，“>>”为添加输出重定向。如：
    - “dir > temp.txt”将把“dir”命令在屏幕上的输出保存在新文件“temp.txt”中；
    - 而"dir >> temp.txt"将屏幕输出追加在文件"temp.txt"的结尾。
  - 管道"|"是将前一个命令的屏幕输出作为后一个命令的键盘输入。如："dir | sort"将把"dir"命令的输出按行进行排序。
- 环境变量
  - set PATH=c:\tools;%PATH%——原PATH=c:\dos 则后PATH=c:\tools;c:\dos
- 系统引导时加载
  - 系统引导时加载：autoexec.bat

2.2.1.2. DOS批处理：由command.com执行

有简单的变量替换，有条件转移和跳转、循环和注释语句rem

- 循环：循环执行命令。
  - FOR /R [[drive:]path] %variable IN (set) DO command [command-parameters]
  - 遍历根在[drive:]path上的目录树，在树的每个目录中执行 FOR 语句。如果在 /R 后没有指定任何目录规范，那么假设为当前目录。如果 set 仅是一个句号(.)，那么它 将仅列出目录树。

关于循环的实例

如下面批处理将显示当前目录及其子目录所有文件名(含路径名)；

```
for /R %f in (*.*) do echo %f
```

如下面批处理将显示当前目录及其子目录所有后缀为ppt的文件名(含路径名)；

```
for /R %f in (*.ppt) do echo %f
```

可能的显示结果：

C:\users\work\2001-02-20 chapter1.ppt

C:\users\work\temp\2001-02-18 chapter1.ppt

C:\users\work\temp\2000-08-09 Linux Lecture\2000-08-15 Linux.ppt

分支

IF [NOT] ERRORLEVEL number command
IF [NOT] string1==string2 command
IF [NOT] EXIST filename command

NOT　　只有在条件为假时，才需运行命令。
ERRORLEVEL number　　如果最近程序运行返回的退出码等于或大于指定的号码时，则设定条件为真。
command　　当符合条件时，指定要运行的命令。
string1==string2　　当指定字符串匹配时，设置条件为真。
EXIST filename　　当指定文件名存在时，设置条件为真。

如：

```
IF ERRORLEVEL 1 goto okay
:okay echo okay
```

2.2.2 UNIX的作业管理

2.2.2.1. shell命令处理程序

2.2.2.2. shell批处理

2.2.2.1. shell命令处理程序

1. shell的类型
2. 初始化文件
3. 基本特征
4. 输入输出重定向
5. 管道
6. 后台执行
7. 环境变量和内部变量
8. 别名
9. 常用的外部命令

1. shell的类型

UNIX上有许多种shell，主要功能是相同的，在细节上有一些区别。几种shell 都有它们的优点和缺点。

- Bourne shell(/bin/sh)
  - Bourne shell 的作者是 Steven Bourne。它是 UNIX 最初使用的shell 并且在每种 UNIX 上都可以使用。Bourne shell 在 shell 编程方面相当优秀，但在处理与用户的交互方面作得不如其他几种 shell。
  - Bourne shell 最大的缺点在于它处理用户的输入方面。在 Bourne shell 里键入命令会很麻烦，尤其当你键入很多相似的命令时。
- C shell(/bin/csh)
  - C shell 由 Bill Joy 所写，它更多的考虑了用户界面的友好性。它支持象命令补齐（command-line completion）等一些 Bourne shell 所不支持的特性。普遍认为C shell 的编程接口做的不如 Bourne shell, 但 C shell 被很多 C 程序员使用因为 C shell的语法和 C语言的很相似，这也是C shell名称的由来。

- Korn shell (ksh)
  - Korn shell (ksh) 由 Dave Korn 所写。它集合了C shell 和 Bourne shell 的优点并且和 Bourne shell 完全兼容。
- Bourne Again shell (bash)
  - Bourne Again shell (bash)是 Bourne shell 的扩展。bash 与 Bourne shell 完全向后兼容，并且在 Bourne shell 的基础上增加和增强了很多特性。bash 也包含了很多 C 和 Korn shell 里的优点。bash 有很灵活和强大的编程接口，同时又有很友好的用户界面。 bash 有几种特性使命令的输入变得更容易。
- 其他shell
  - 除了这些 shell 以外，许多其他的 shell 程序吸收了这些原来的 shell 程序的优点而成为新的 shell 。如：
  - tcsh (csh 的扩展)
  - Public Domain Korn shell (pdksh, ksh 的扩展)

2. 初始化文件

- B Shell执行.profile
- C Shell执行.cshrc和.login两个文件
- K Shell执行.profile和$ENV两个文件
- (sh) .profile与(csh) .login对应，进行注册时的初始化；而在csh在非注册启动时都读入.cshrc；

3. 基本特征

- 内部命令： 如cd, exec——区分大小写，exec的功能是执行一个命令；
- 外部命令： 如ls, mkdir
- 命令行选项通常是： -option
  - 如："ls -a -l"中的-a表示列出所有文件，-l表示列出所有信息。
- 通配符： 由shell处理后再传递给外部命令。
  - 如： cat *.c 则argv[1]="a.c", argv[2]="b.c"， 而 cat "*.c" 则argv[1]="*.c"(cat的功能是读入所有文件，并显示)

4. 输入输出重定向

基于内核的缓冲区

- "<"为标准输入重定向；
- ">"和">>"为标准输出重定向；
- "2>"和"2>>"为标准错误输出重定向（2表示标准错误输出的设备号，只对sh有意义）；
- " >&"是标准输出和标准错误输出重定向；

行输入重定向：用定界符间的内容作为标准输入。
如：下面命令的标准输入为邮件内容。

```
mail user2 << WARNING
...
WARNING
```

5. 管道

管道可以把一系列命令连接起来。第一个命令的输出会通过管道传给第二个命令而作为第二个命令的输入，第二个命令的输出又会作为第三个命令的输入，以此类推。而管道行中最后一个命令的输出才会显示在屏幕上（如果命令行里使用了输出重定向的话，将会放进一个文件里）。通过管道，可以将多个简单程序组合完成复杂的功能。

- 如："ls -l | wc -l"可给出文件数目。
- 如：“cat sample.text | grep "High" | wc -l” 这个管道将把 cat 命令（列出一个文件的内容）的输出送给grep命令。grep 命令在输入里查找单词 High，grep命令的输出则是所有包含单词 High的行，这个输出又被送给 wc命令。带 -l选项的wc命令将统计输入里的行数。

6. 后台执行

后台执行：cmd &;

- 如："xterm -display 202.204.5.88:0.0 &"为在后台启动一个xterm窗口，并显示到主机202.204.5.88上。

7. 环境变量和内部变量

内部变量不能被子进程继承（如同C里的局部变量）；改环境变量就会自动改内部变量，反之不然。

- "set"可给出内部变量列表，"env"可给出环境变量列表。继承只对环境变量有效。
- sh: PATH=/usr/bin:$PATH ; export PATH——注意：在export前为内部变量，之后为环境变量。
- csh: setenv PATH /usr/bin:$PATH--注意：在csh中环境变量的赋值(setenv)没有等号，而内部变量的赋值(set)有等号。

8. 别名

alias 的格式:

alias aliasname=string ….

把 aliasname 直接用来取代后面的 string ， 如有任何跟在后面的 argument 将会出现的其后。利用该功能，使用者可以将常用却冗长的指令以其他的名字存起。

如："alias dir='ls -a -l'"为"ls -a -l"定义了一个别名"dir"；

9. 常用的外部命令

- man查看手册
- echo, wc, grep, sed, awk（用于文本扫描和处理）, sort, cut（对每行进行特定删除处理）字符串操作;
- pwd, ls, mkdir, rmdir, cp, rm, mv, ln文件和目录操作;
- chmod, chown, chgrp（修改文件所在的用户组）文件权限和属主;
- cat, more, tail（显示文件的最后部分）查看文件;
- test, expr检测和数值计算;
- vi全屏幕编辑;

总结

- 命令
  - Command -/options para
- 环境变量
  - Set, env
- 初始化文件
  - Autoexec.bat, .profile, .cshrc, .login
  - 进入系统之后第一个执行的文件
  - 设置环境变量
- 程序相关内容
  - 输入输出重定向
    - 改变标注输入和标准输出
  - 管道的使用
    - 改变标注输入输出
  - 程序控制语言(循环,分支)
    - Errorlevel number

2.3 系统调用(SYSTEM CALL)

系统调用是操作系统提供给软件开发人员的唯一接口，开发人员可利用它使用系统功能。OS核心中都有一组实现系统功能的过程（子程序），系统调用就是对上述过程的调用。

返回

2.3.1 系统调用及其功能

每个操作系统都提供几百种系统调用，包括：外存文件与目录的读写，各种I/O设备的使用，在程序中启动另一个程序，查询和统计系统资源使用情况等等。

1. 系统调用的功能

1) 设备管理：
设备的读写和控制；

Ioctl　设备配置
Open　设备打开
Close　设备关闭
Read　读设备
Write　写设备

2) 文件管理：文件读写和文件控制；
Open　文件打开
Close　文件关闭
Read　读文件
Write　写文件
seek　读写指针定位
Creat　文件创建
Stat　读文件状态
Mount 安装文件系统
chmod 修改文件属性

3) 进程控制：创建、中止、暂停等控制；
Fork　创建进程
Exit　进程自我终止
Wait　阻塞当前进程
Sleep　进程睡眠
Getpid 读父进程标识

4) 进程通信：消息队列、共享存储区、socket等通信渠道的建立、使用和删除；

5) 存储管理：内存的申请和释放；

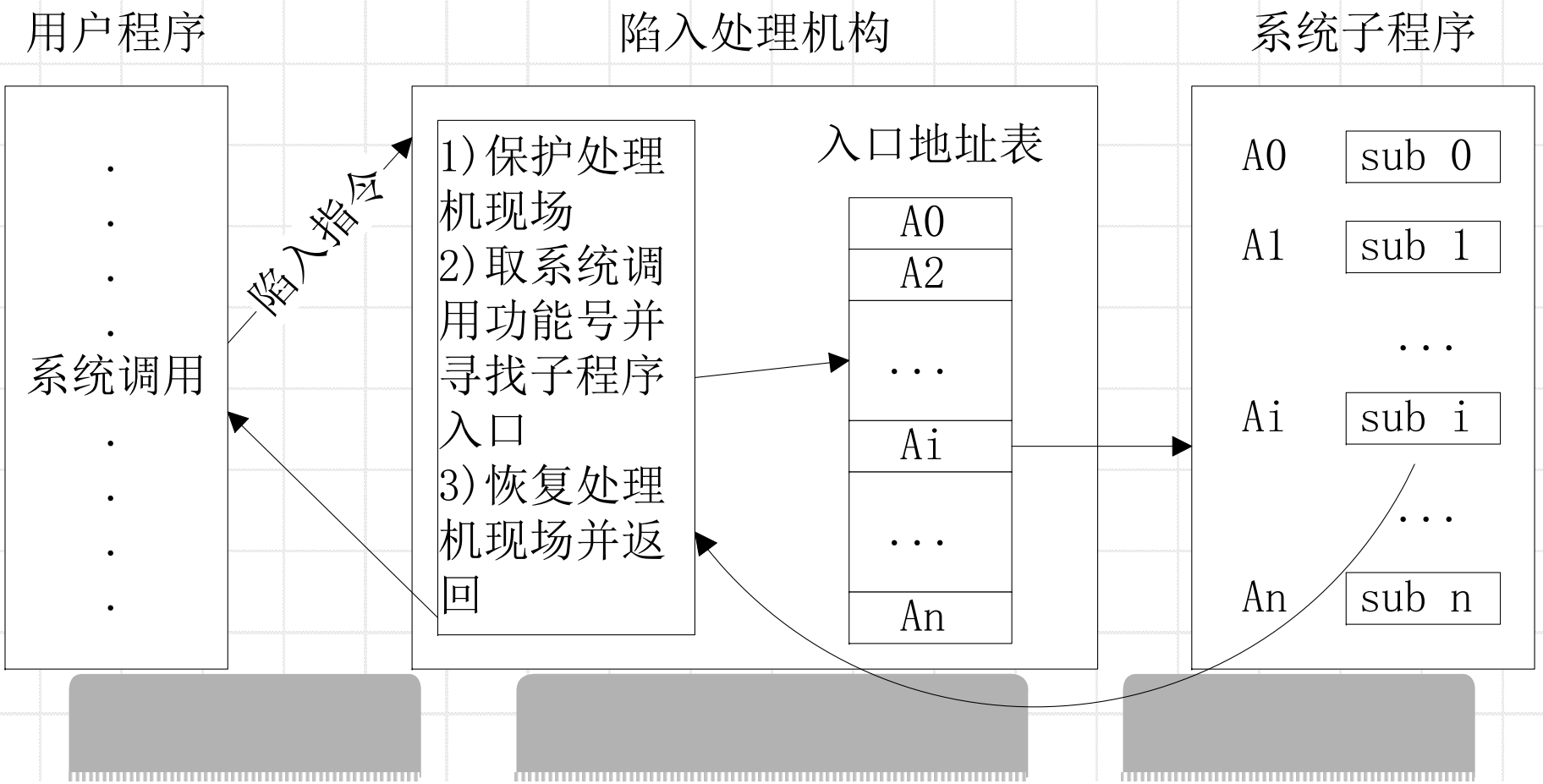
6) 系统管理：设置和读取时间、读取用户和主机标识等；
gtime　读取时间
Stime　设置时间
getuid　读取用户标识

2. 通过系统调用接口使用系统命令

- 通过系统调用接口也可使用系统命令。
  - C语言里的system()函数可调用shell来完成命令
  - 如 UNIX系统： system("cp -r doc /tmp")

2.3.2 系统调用的实现过程

实际上系统调用语句本身是硬件提供的（机器指令），但其所调用的功能是操作系统提供的。每种机器的机器指令集中都有一条系统调用指令。



- 设置系统调用号和参数。
  - 调用号作为指令的一部分（如早期UNIX），或装入到特定寄存器里（如：DOS int 21h，AH=调用号。）
  - 指令自带；或者参数装入到特定寄存器里；或以寄存器指针指向参数表（内存区域）。
- 执行trap(int)指令：入口的一般性处理，查入口跳转表，跳转到相应功能的过程。
  - 保护CPU现场(将PC与PSW入栈)，改变CPU执行状态（处理机状态字PSW切换，地址空间表切换）
  - 将参数取到核心空间
- 执行操作系统内部代码；
- 执行iret指令：将执行结果装入适当位置（类似于参数带入），恢复CPU现场（以栈顶内容置PSW和PC）。

2.3.4 系统调用与普通过程调用的相同点和不同点

- 相同点
  - 改变指令流程
  - 重复执行和公用
  - 改变指令流程后需要返回原处
- 不同点
  - 系统调用是动态调用，而CALL调用方式是静态调用；
  - 执行状态不同
  - 进入方式不同
  - 与进程调度的关系不同：
  - 嵌套或递归调用

1. 系统调用是动态调用，而CALL调用方式是静态调用；

系统调用是动态调用，程序中不包含被调用代码，好处：
（1）用户程序长度缩短
（2）当OS升级时，调用方不必改变
系统调用方式的调用地址和返回地址都是不固定的
系统调用指令中不包含调用地址，只包含功能号，是按功能号（在可执行目标程序中）调用的。在操作系统内部，由系统调用处理程序通过系统调用分支表（OS的一个数据结构）将功能号转换为相应的指令地址。
系统调用返回指令中不包括返回地址，通过栈保存和弹出返回地址。系统调用返回地址不固定，因为用户程序在不同的地方调用OS。
CALL调用方式是静态调用，被调用代码与调用代码在同一程序之内。CALL调用方式，其调用地址是固定的，包含在调用语句中；返回地址是不固定的（同一子程序可能被不同处多次调用），在程序执行过程中通过栈的实现来保存和弹出返回地址。

2. 执行状态不同

调用和返回经历了不同的系统状态。通常核心和应用程序的代码分别运行在CPU的不同的状态下（系统态/核心态/管态和用户态/目态），所用地址空间也不同——核心的代码可以直接访问应用进程的地址空间，反之不然。

- 状态切换：系统调用、中断、异常
- trap陷入
- 特权指令（访问关键寄存器、停机指令）和I/O敏感指令（中断屏蔽、端口读写）

3. 进入方式不同

利用int或trap指令进行系统调用；利用call或jmp指令进入普通的过程调用；

- CALL指令的内部实现过程：
  - 返回地址压栈（即该CALL指令所在的地址）；
  - 将该CALL指令中所含的地址（即被调用代码所在地址）送入PC
- RET指令的内部实现过程：
  - 从栈顶弹出返回地址送入程序计数器PC

4. 与进程调度的关系不同

采用抢先式调度的系统，在系统调用返回时，要进行重新调度的检查——是否有更高优先级的任务就绪（创建或唤醒）。

5. 嵌套或递归调用

对系统调用，一般不允许在同一个进程中发生嵌套或递归（不同进程可以重入同一个系统调用）。

小结

- 作业组织和控制：脱机、联机（命令行）
- 系统调用：与普通过程调用的区别、与高级语言函数库的区别、实现过程
- 作业管理举例：DOS、UNIX(shell)
- 图形用户接口(GUI)：概述、X Window、MS Windows、事件驱动模式

作业

1） 在Windows2000下用批处理命令实现UNIX的grep命令的主要功能。

2） 在Linux下用Bash实现Windows的dir命令的主要功能。

第三章 进程管理

为了描述程序在并发执行时对系统资源的共享，我们需要一个描述程序执行时动态特征的概念，这就是进程。在本章中，我们将讨论进程概念、进程控制和进程间关系。

3.1 进程(PROCESS)

3.1.1 程序的顺序执行和并发执行

3.1.2 进程的定义和描述

3.1.3 进程的状态转换

3.1.4 操作系统代码的执行

返回

3.1.1 程序的顺序执行和并发执行

- 程序的执行有两种方式：顺序执行和并发执行。
  - 顺序执行是单道批处理系统的执行方式，也用于简单的单片机系统；
  - 现在的操作系统多为并发执行，具有许多新的特征。引入并发执行的目的是为了提高资源利用率。

- 顺序执行的特征
  - 顺序性：按照程序结构所指定的次序（可能有分支或循环）
  - 封闭性：独占全部资源，计算机的状态只由于该程序的控制逻辑所决定
  - 可再现性：初始条件相同则结果相同。如：可通过空指令控制时间关系。
- 并发执行的特征
  - 间断(异步)性："走走停停"，一个程序可能走到中途停下来，失去原有的时序关系；
  - 失去封闭性：共享资源，受其他程序的控制逻辑的影响。如：一个程序写到存储器中的数据可能被另一个程序修改，失去原有的不变特征。
  - 失去可再现性：失去封闭性 —>失去可再现性；外界环境在程序的两次执行期间发生变化，失去原有的可重复特征。

并发执行的条件：达到封闭性和可再现性

并发执行失去封闭性的原因是共享资源的影响，去掉这种影响就行了。1966年，由Bernstein给出并发执行的条件。（这里没有考虑执行速度的影响。）

- 程序 P(i) 针对共享变量的读集和写集 R(i)和W(i)
- 条件：任意两个程序P(i)和P(j)，有：
  - $R(i) \cap W(j) = \Phi$;
  - $W(i) \cap R(j) = \Phi$;
  - $W(i) \cap W(j) = \Phi$;

前两条保证一个程序的两次读之间数据不变化；最后一条保证写的结果不丢掉。

现在的问题是这个条件不好检查。

3.1.2 进程的定义和描述

1. 进程的定义

一个具有一定独立功能的程序对一个数据集合在处理机上执行过程和资源分配的基本单位。

- 它对应虚拟处理机、虚拟存储器和虚拟外设等资源的分配和回收；
- 引入多进程，提高了对硬件资源的利用率，但又带来额外的空间和时间开销，增加了OS 的复杂性；

2. 进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制。
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

3. 处理机调度器（dispatcher）

处理机调度器是操作系统中的一段代码，它完成如下功能：

- 把处理机从一个进程切换到另一个进程；
- 防止某进程独占处理机；

4. 进程与内存

- 动态性：进程具有动态的地址空间，地址空间上包括：
  - 代码（指令执行和CPU状态的改变）
  - 数据（变量的生成和赋值）
  - 系统控制信息（进程控制块的生成和删除）
- 独立性：各进程的地址空间相互独立，除非采用进程间通信手段；
- 并发性、异步性：“虚拟的各种设备和资源”
- 结构化：代码段、数据段和核心段（在地址空间中）；程序文件中通常也划分了代码段和数据段，而核心段通常就是OS核心（由各个进程共享，包括各进程的PCB）
- 两种状态：
  - 用户态时不可直接访问受保护的OS代码；
  - 核心态时执行OS代码，可以访问全部进程空间。

5. 进程控制块 (PCB, process control block)

进程控制块是由OS维护的用来记录进程相关信息的一块内存。

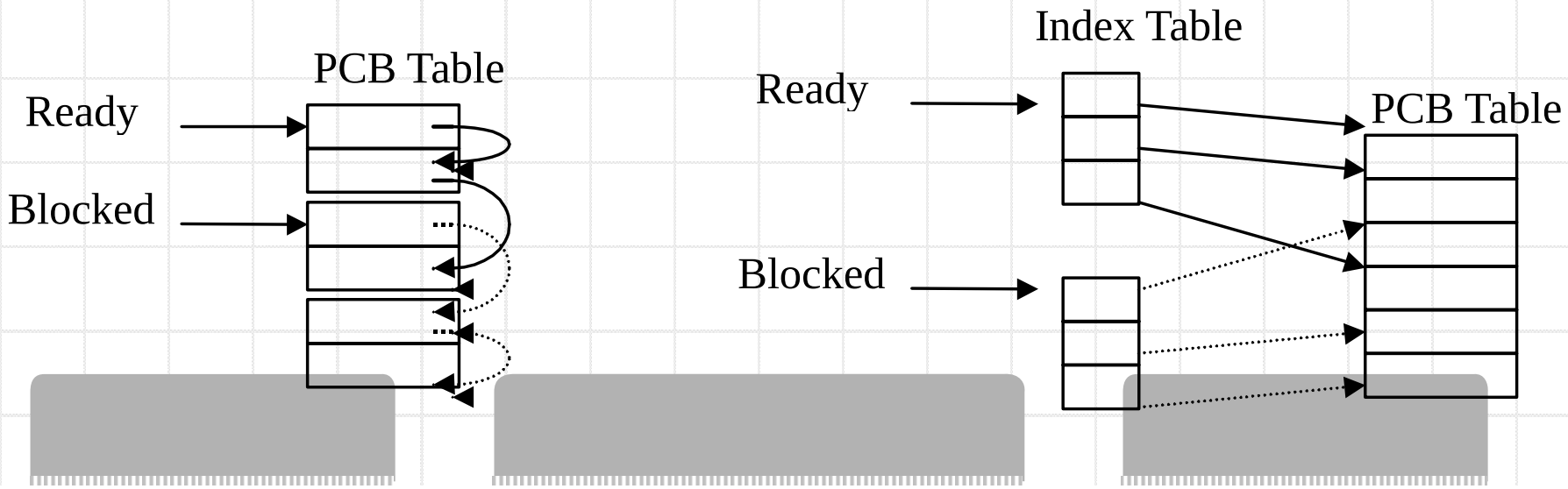
- 每个进程在OS中的登记表项（可能有总数目限制），OS据此对进程进行控制和管理（PCB中的内容会动态改变），不同OS则不同
- 处于核心段，通常不能由应用程序自身的代码来直接访问，而要通过系统调用，或通过UNIX中的进程文件系统(/proc)直接访问进程映象(image)。文件名为进程标识（如：00316），权限为创建者可读写。

进程控制块的内容

- 进程描述信息：
  - 进程标识符(process ID)，唯一，通常是一个整数；
  - 进程名，通常基于可执行文件名（不唯一）；
  - 用户标识符(user ID)；进程组关系(process group)
- 进程控制信息：
  - 当前状态；
  - 优先级(priority)；
  - 代码执行入口地址；
  - 程序的外存地址；
  - 运行统计信息（执行时间、页面调度）；
  - 进程间同步和通信；阻塞原因
- 资源占用信息：虚拟地址空间的现状、打开文件列表
- CPU现场保护结构：寄存器值（通用、程序计数器PC、状态PSW，地址包括栈指针）

6. PCB的组织方式

- 链表：同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
  - 各状态的进程形成不同的链表：就绪链表、阻塞链表
- 索引表：同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表
  - 各状态的进行形成不同的索引表：就绪索引表、阻塞索引表



7.进程上下文

进程上下文是对进程执行活动全过程的静态描述。进程上下文由进程的用户地址空间内容、硬件寄存器内容及与该进程相关的核心数据结构组成。

- 用户级上下文：进程的用户地址空间（包括用户栈各层次），包括用户正文段、用户数据段和用户栈；
- 寄存器级上下文：程序寄存器、处理机状态寄存器、栈指针、通用寄存器的值；
- 系统级上下文：
  - 静态部分（PCB和资源表格）
  - 动态部分：核心栈（核心过程的栈结构，不同进程在调用相同核心过程时有不同核心栈）

3.1.3 进程的状态转换

4.1.3.1 两状态进程模型

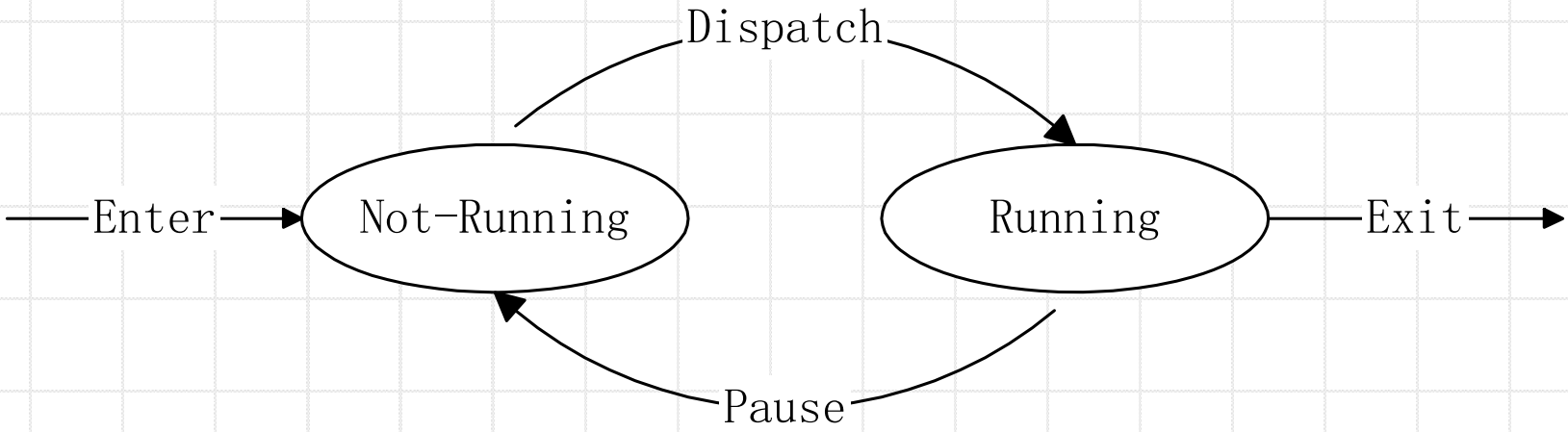
4.1.3.2 五状态进程模型

4.1.3.3 挂起进程模型

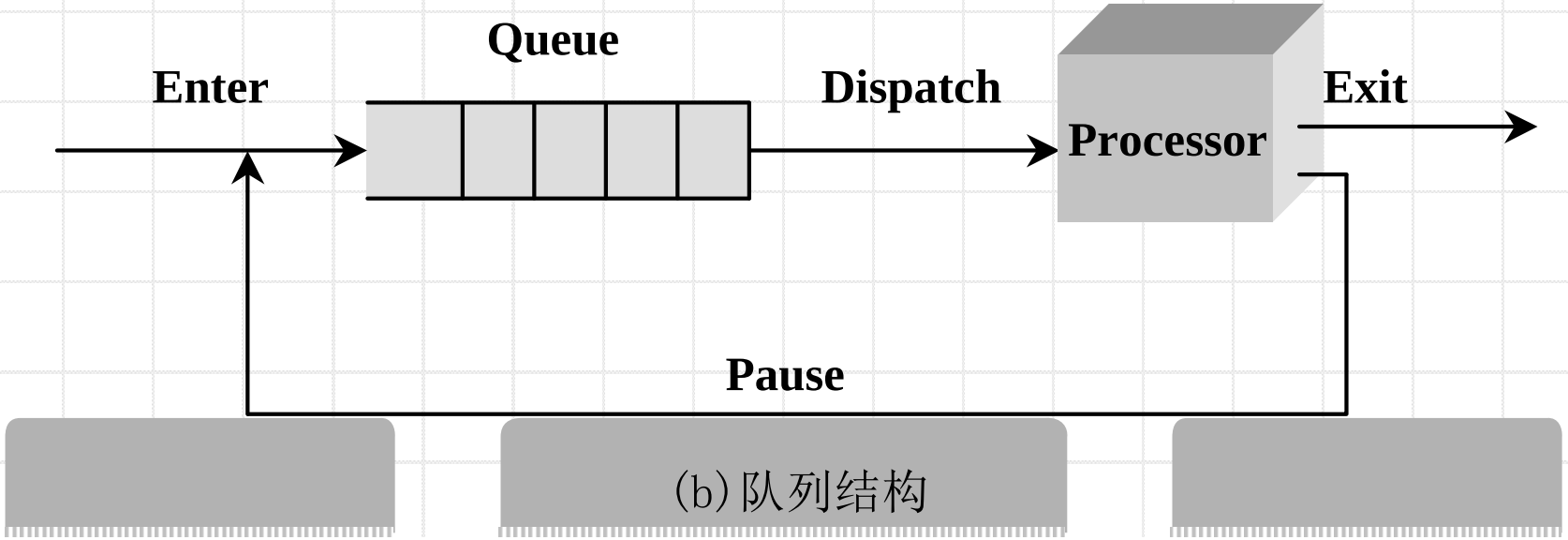
进程状态

- 进程生命周期
  - 进程划分为各种不同的状态
- 基本
  - 一般三状态

3.1.3.1 两状态进程模型



(a)状态变迁图



(b)队列结构

1. 状态

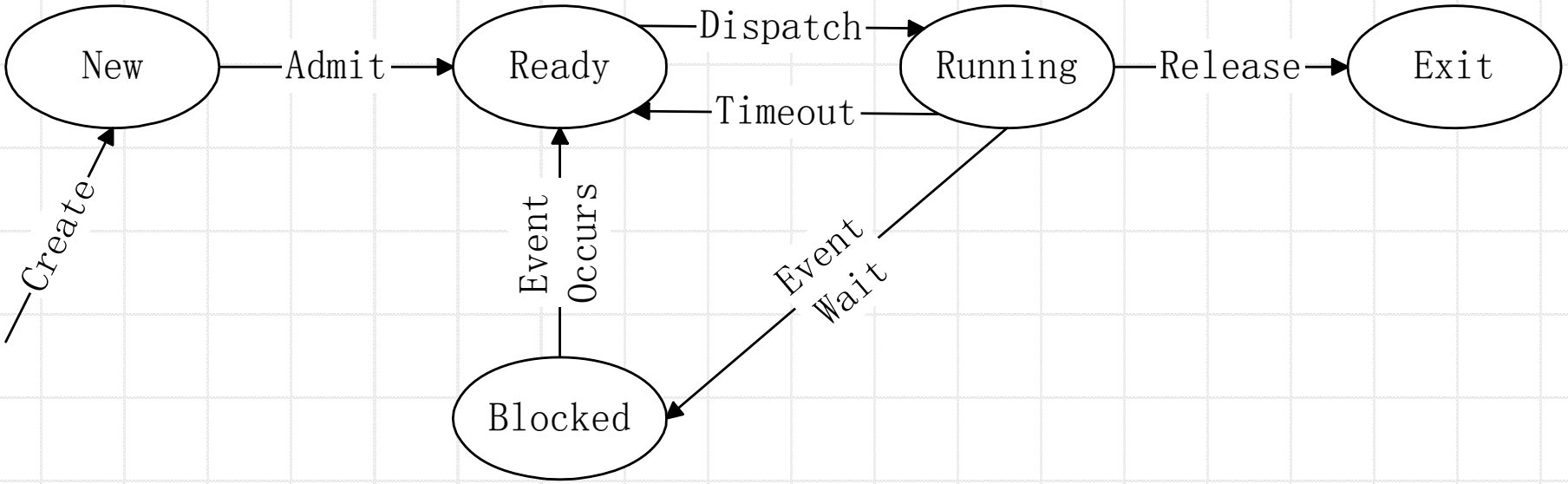
- 运行状态(Running)：占用处理机资源；
- 暂停状态(Not-Running)：等待进程调度分配处理机资源；

2. 转换

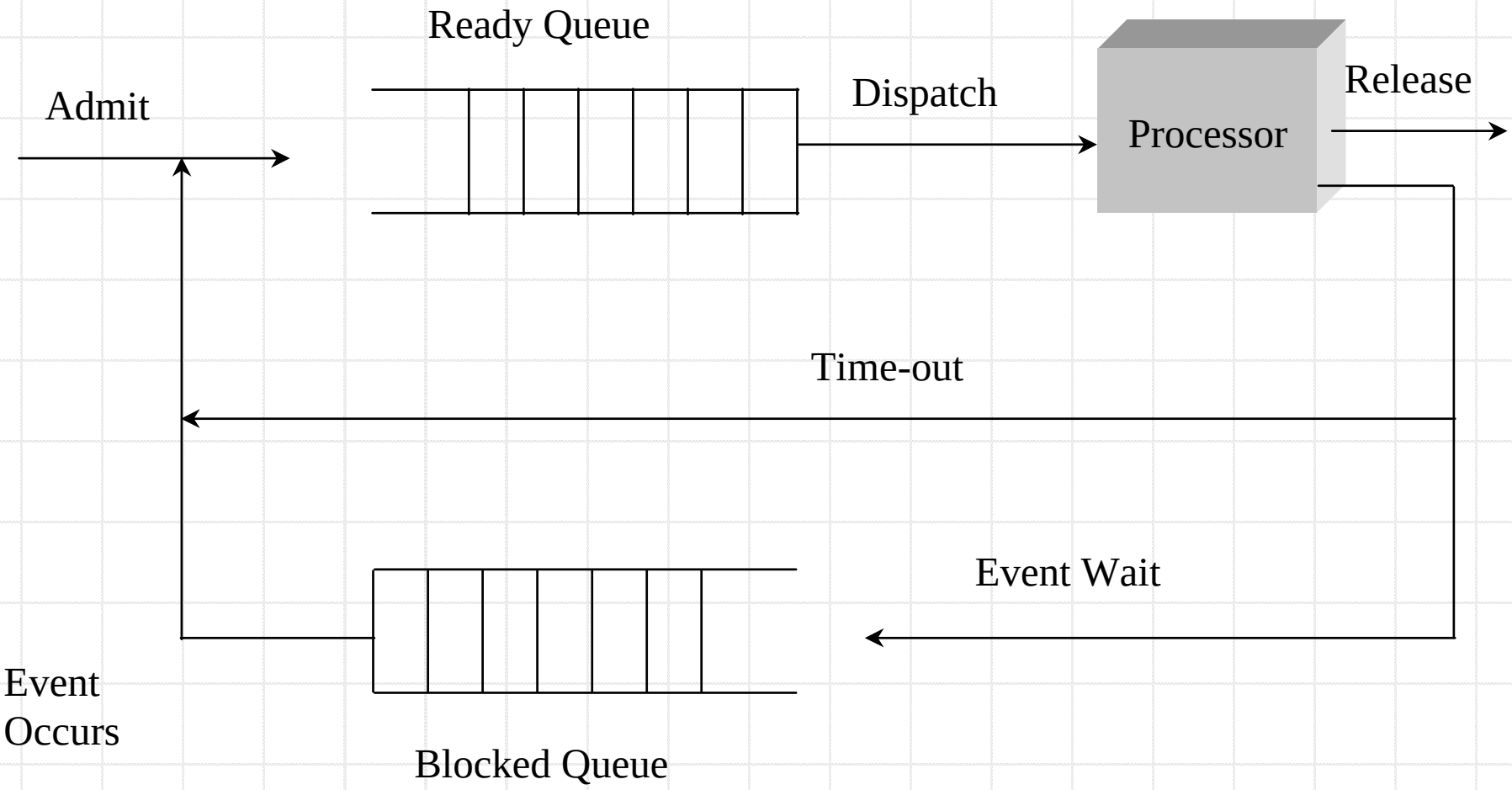
- 进程创建(Enter)：系统创建进程，形成PCB，分配所需资源，排入暂停进程表(可为一个队列)；
- 调度运行(Dispatch)：从暂停进程表中选择一个进程(要求已完成I/O操作)，进入运行状态；
- 暂停运行(Pause)：用完时间片或启动I/O操作后，放弃处理机，进入暂停进程表；
- 进程结束(Exit)：进程运行中止；

3.1.3.2 五状态进程模型

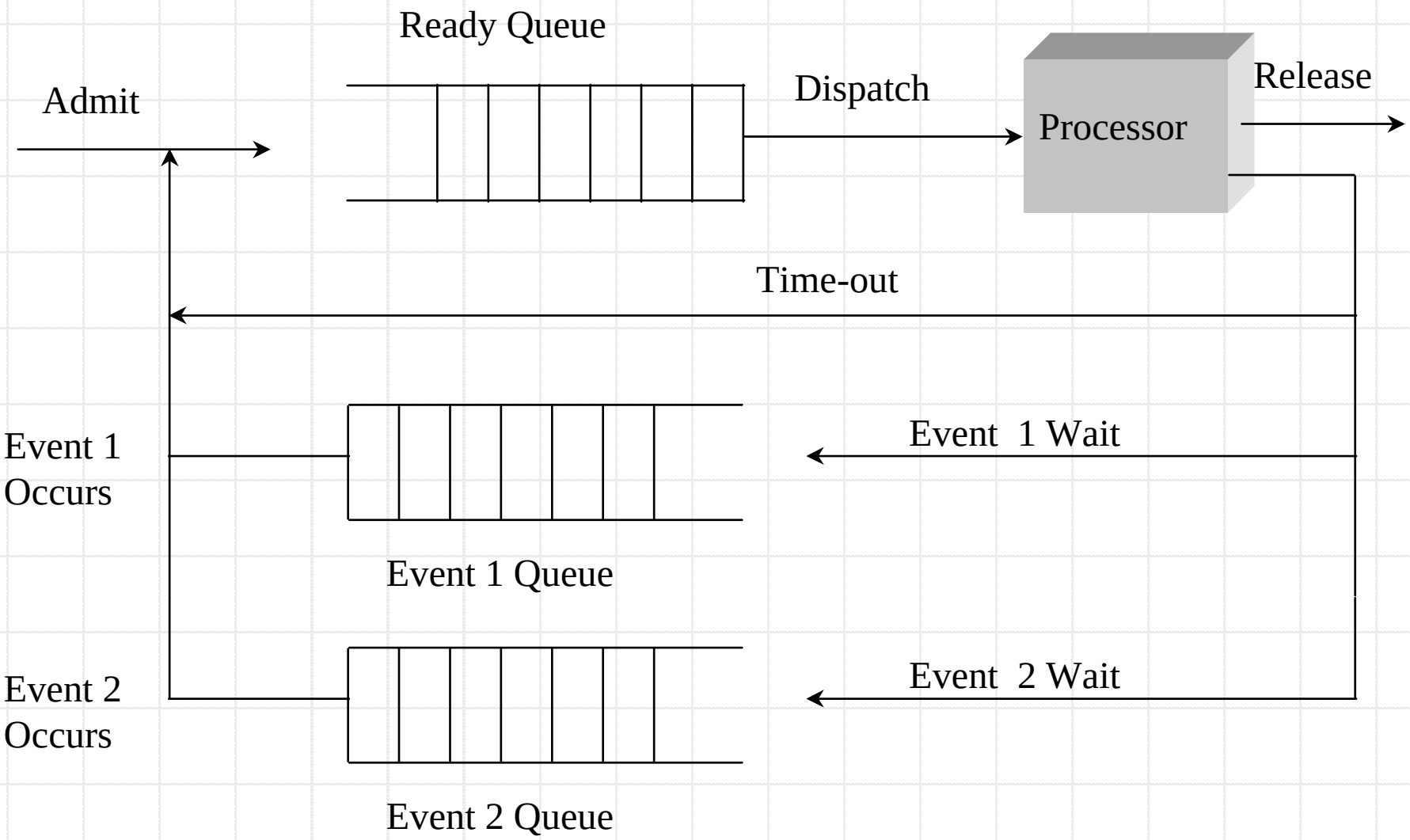
两状态模型无法区分暂停进程表中的可运行和阻塞，五状态模型就是对暂停状态的细化。



五状态进程模型(状态变迁)



五状态进程模型(单队列结构)



五状态进程模型(多队列结构)

1. 状态

- 运行状态(Running)：占用处理机资源；处于此状态的进程的数目小于等于CPU的数目。
  - 在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的idle进程（相当于空操作）。
- 就绪状态(Ready)：进程已获得除处理机外的所需资源，等待分配处理机资源；只要分配CPU就可执行。
  - 可以按多个优先级来划分队列，如：时间片用完—>低优，I/O完成—>中优，页面调入完成—>高优
- 阻塞状态(Blocked)：由于进程等待某种条件（如I/O操作或进程同步），在条件满足之前无法继续执行。该事件发生前即使把处理机分配给该进程，也无法运行。如：等待I/O操作的完成。

- 创建状态(New)：进程刚创建，但还不能运行(一种可能的原因是OS对并发进程数的限制)；如：分配和建立PCB表项（可能有数目限制）、建立资源表格（如打开文件表）并分配资源，加载程序并建立地址空间表。
- 结束状态(Exit)：进程已结束运行，回收除PCB之外的其他资源，并让其他进程从PCB中收集有关信息（如记帐，将退出码exit code传递给父进程）。

2. 转换

- 创建新进程：创建一个新进程，以运行一个程序。可能的原因为：用户登录、OS创建以提供某项服务、批处理作业。
- 收容(Admit, 也称为提交)：收容一个新进程，进入就绪状态。由于性能、内存、进程总数等原因，系统会限制并发进程总数。
- 调度运行(Dispatch)：从就绪进程表中选择一个进程，进入运行状态；
- 释放(Release)：由于进程完成或失败而中止进程运行，进入结束状态；
  - 运行到结束：分为正常退出Exit和异常退出abort（执行超时或内存不够，非法指令或地址，I/O失败，被其他进程所终止）
  - 就绪或阻塞到结束：可能的原因有：父进程可在任何时间中止子进程；

- 超时（Timeout）：由于用完时间片或高优先进程就绪等导致进程暂停运行；
- 事件等待（Event Wait）：进程要求的事件未出现而进入阻塞；可能的原因包括：申请系统服务或资源、通信、I/O操作等；
- 事件出现（Event Occurs）：进程等待的事件出现；如：操作完成、申请成功等；

注：对于五状态进程模型，一个重要的问题是当一个事件出现时如何检查阻塞进程表中的进程状态。当进程多时，对系统性能影响很大。一种可能的作法是按等待事件类型，排成多个队列。

3.1.3.3 挂起进程模型

- 这个问题的出现是由于进程优先级的引入，一些低优先级进程可能等待较长时间，从而被对换至外存。这样做的目的是：
  - 提高用户满意度：为优先级高的进程提供足够的资源
  - 为运行进程提供足够内存：资源紧张时，暂停某些进程，如：CPU繁忙（或实时任务执行），内存紧张
  - 用于调试：在调试时，挂起被调试进程（从而对其地址空间进行读写）

3.2 进程控制

3.2.1 进程控制的功能

3.2.2 进程的创建和退出

3.2.3 进程的阻塞和唤醒

3.2.4 UNIX进程管理举例

3.2.1 进程控制的功能

完成进程状态的转换。

原语(primitive)：由若干条指令构成的“原子操作(atomic operation)”过程，作为一个整体而不可分割——要么全都完成，要么全都不做。许多系统调用就是原语。

注意：系统调用并不都是原语。进程A调用read()，因无数据而阻塞，在read()里未返回。然后进程B调用read()，此时read()被重入。系统调用不一定一次执行完并返回该进程，有可能在特定的点暂停，而转入到其他进程。

3.2.2 进程的创建和退出

1. 创建

	产生新进程	不产生新进程
复制现有进程的上下文	fork（新进程的系统上下文会有不同）	—
加载程序	spawn（等待子进程完成，）	exec（加载新程序并覆盖自身）

- 创建方式：
  - 系统统一创建
    - 不被继承：进程标识符，无父进程标识符
  - 父进程创建
    - 继承(inherit)：子进程可以从父进程中继承用户标识符、环境变量、打开文件、文件系统的当前目录、控制终端、已经连接的共享存储区、信号处理例程入口表等
    - spawn创建并执行一个新进程；新进程与父进程的关系可有多种：覆盖(_P_OVERLAY)、并发(_P_NOWAIT or _P_NOWAITO)、父进程阻塞(_P_WAIT)、后台(_P_DETACH)等。

2. 撤销

- 也称为“终止”或主程序返回：调用exit()可终止进程。
- 释放资源：
  - 释放内外存空间
  - 关闭所有打开文件
  - 释放共享内存段和各种锁定lock

3.2.3 UNIX进程的阻塞和唤醒

- 阻塞：
  - 暂停一段时间(sleep)；
  - 暂停并等待信号(pause)；
  - 等待子进程暂停或终止(wait)；
- 唤醒：发送信号到某个或一组进程(kill)

- 调用wait挂起本进程以等待子进程的结束，子进程结束时返回。父进程创建多个子进程且已有某子进程退出时，父进程中wait函数在第一个子进程结束时返回。
  - 其调用格式为"pid_t wait(int *stat_loc);"；返回值为子进程ID。
  - waitpid(9)等待指定进程号的子进程的返回并修改状态；
  - waitid()等待子进程修改状态；
- 调用pause挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。
  - 其调用格式为"int pause(void);"；

- 调用sleep将在指定的时间seconds内挂起本进程。其调用格式为："unsigned sleep(unsigned seconds);"；返回值为实际的挂起时间。
- 调用kill可发送信号sig到某个或一组进程pid。其调用格式为："int kill(pid_t pid, int sig);"。信号的定义在文件"/usr/ucbinclude/sys/signal.h"中。命令"kill"可用于向进程发送信号。如："kill -9 100"将发送SIGKILL到ID为100的进程；该命令将中止该进程的执行。

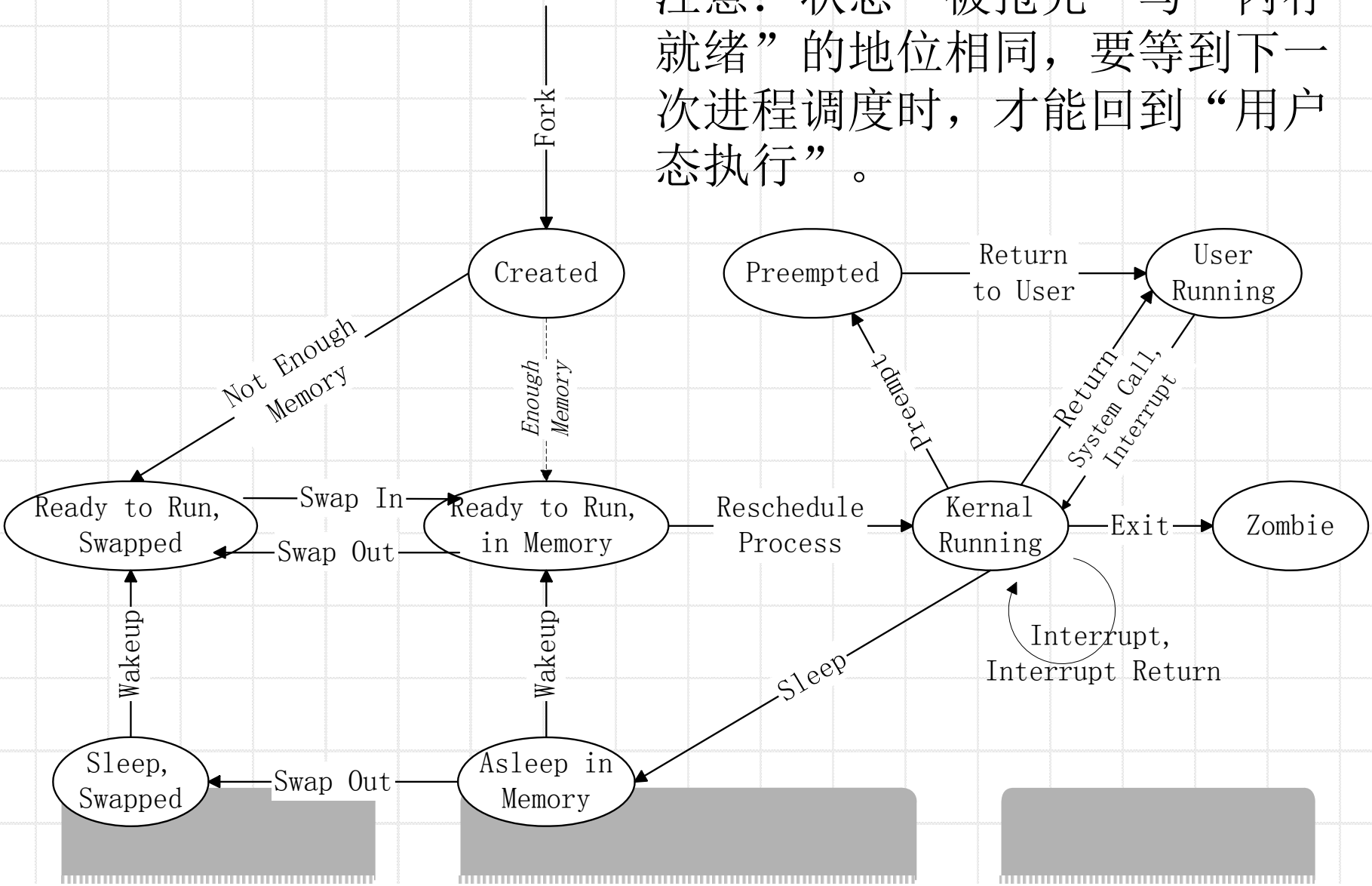
3.2.4 UNIX进程管理举例

1. 进程结构

- 进程上下文：指进程的用户地址空间内容、寄存器内容及与进程相关的核心数据结构；包括三部分"上下文"：用户级、寄存器级、系统级
- 用户级上下文：正文段即代码(text)；数据段(data)；栈段(user stack)：用户态执行时的过程调用；共享存储区(shared memory)
  - 把地址空间的段称为"区(region)"：进程区表和系统区表（前者索引指向后者）
- 系统上下文：
  - proc结构：总在内存，内容包括阻塞原因；
  - user结构：可以调出到外存，进程处于执行状态时才用得着，各种资源表格；
  - 进程区表：从虚拟地址到物理地址的映射；
  - 核心栈：核心态执行时的过程调用的栈结构；

2. 进程状态转换

注意：状态“被抢先”与“内存就绪”的地位相同，要等到下一次进程调度时，才能回到“用户态执行”。



3. 进程控制

- 父子进程的fork()返回值不同，Parent PID的值不同 getppid()
- fork()创建子进程之后，执行返回父进程，或调度切换到子进程以及其他进程
- fork创建一个新进程（子进程），子进程是父进程的精确复制。在子进程中返回为0；在父进程中，返回子进程的标识。子进程是从fork调用返回时在用户态开始运行的。父进程的返回点与子进程的开始点是相同的。
- exec用一个新进程覆盖调用进程。它的参数包括新进程对应的文件和命令行参数。成功调用时，不再返回；否则，返回出错原因。

- 退出：exit()向父进程给出一个退出码（8位的整数）。父进程终止时如何影响子进程：
  - 子进程从父进程继承了进程组ID和终端组ID（控制终端），因此子进程对发给该进程组或终端组的信号敏感。终端关闭时，以该终端为控制终端的所有进程都收到SIGHUP信号。
  - 子进程终止时，向父进程发送SIGCHLD信号，父进程截获此信号并通过wait3()系统调用来释放子进程PCB。

- 阻塞：暂停一段时间sleep；暂停并等待信号pause；等待子进程暂停或终止wait：可以取得子进程的退出码（16位整数：exit退出时exitCode*256，信号终止时coreFlag*128 + signalNumber，信号停止时signalNumber*256 + 0x7f）
- 唤醒：发送信号到某个或一组进程kill：使得接收方从阻塞的系统调用中返回（如read返回并给出失败值-1），并随即调用相应的信号处理例程
- 调试：设置执行断点(breakpoint)，读写进程映象中的数据（从而修改进程的上下文）。系统调用ptrace()，信号SIGTRAP（由调试方发送到被调试方）ptrace允许父进程控制子进程的运行，可用于调试。子进程在遇到signal时暂停，父进程可访问core image。

实例

演示子进程与父进程的关系和fork、exec、wait的使用；

程序main.c

功能是进行10次循环，创建2个子进程。循环到第3次时，等待子进程结束。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
void perror(const char *s);
#include <errno.h>
int errno;
int global;
```

```
main() {
  int local,i;
  pid_t child;

  if ((child=fork()) == -1)
  {       // 创建失败
     printf("Fork Error.\n");
  }
  if (child == 0)
  {// 子进程
          printf("Now it is in child process.\n");
          if (execl(“b",“b",NULL) == -1)
          {       // 加载程序失败
             perror("Error in child process");
          }
          global=local + 2;
          exit();
  }
```

```
// 父进程
 printf("Now it is in parent process.\n");
 for (i=0; i<10; i++)
 {
        sleep(2);
        printf("Parent:%d\n",i);
        if (i==2)
        {
             if ((child=fork()) == -1)
             {       // 创建失败
                  printf("Fork Error.\n");
             }
             if (child == 0)
             {// 子进程
                  printf("Now it is in child process.\n");
                  if (execl(“b",“b",NULL) == -1)
                  {       // 加载程序失败
                       perror("Error in child process");
                  }
                  global=local + 2;
                  exit();
             }
        }
        if (i==3)
```

程序test.c

功能是进行10次循环。

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
int global;
main() {
  int local;
  int i;
  pid_t CurrentProcessID, ParentProcessID;
  CurrentProcessID=getpid();
  ParentProcessID=getppid();
  printf("Now it is in the program TEST.\n");
  for (i=0; i<10; i++)
    {
          sleep(2);
          printf("Parent: %d, Current: %d, Nunber:%d\n",ParentProcessID,
                CurrentProcessID,i);
    }
  global=local + 1;
  exit();
}
```

结果

Now it is in parent process.
Now it is in child process.
Now it is in the program TEST.
Parent:0
Parent: 7072, Current: 7073, Nunber:0
Parent:1
Parent: 7072, Current: 7073, Nunber:1
Parent:2
Parent: 7072, Current: 7073, Nunber:2
Now it is in child process.
Now it is in the program TEST.
Parent: 7072, Current: 7073, Nunber:3
Parent:3
Parent: 7072, Current: 7074, Nunber:0
Parent: 7072, Current: 7073, Nunber:4
Parent: 7072, Current: 7074, Nunber:1
Parent: 7072, Current: 7073, Nunber:5
Parent: 7072, Current: 7074, Nunber:2
Parent: 7072, Current: 7073, Nunber:6
Parent: 7072, Current: 7074, Nunber:3
Parent: 7072, Current: 7073, Nunber:7
Parent: 7072, Current: 7074, Nunber:4
Parent: 7072, Current: 7073, Nunber:8
Parent: 7072, Current: 7074, Nunber:5
Parent: 7072, Current: 7073, Nunber:9
Child process ID: 7073
Parent: 7072, Current: 7074, Nunber:6
Parent:4
Parent: 7072, Current: 7074, Nunber:7
Parent:5
Parent: 7072, Current: 7074, Nunber:8
Parent:6
Parent: 7072, Current: 7074, Nunber:9
Parent:7
Parent:8
Parent:9

UNIX进程创建

Program A

```
int global;
main() {
  int local;

  if ((child=fork()) == -1) {
    创建失败
  }
  if (child == 0) {
    子进程

    global=local + 2;
    exit();
  }
  父进程
  global=local + 1;
  exit();
}
```

```
if (execlp("B",…) == -1) {
  加载程序失败
}
```

Program B

```
main() {

  exit();
}
```

3.4 进程互斥和同步

3.4.1 互斥算法

3.4.2 信号量(semaphore)

3.4.3 经典进程同步问题

3.4.1 互斥算法

3.4.1.1临界资源

3.4.1.2临界区的访问过程

3.4.1.3同步机制应遵循的准则

3.4.1.4进程互斥的软件方法

3.4.1.1临界资源

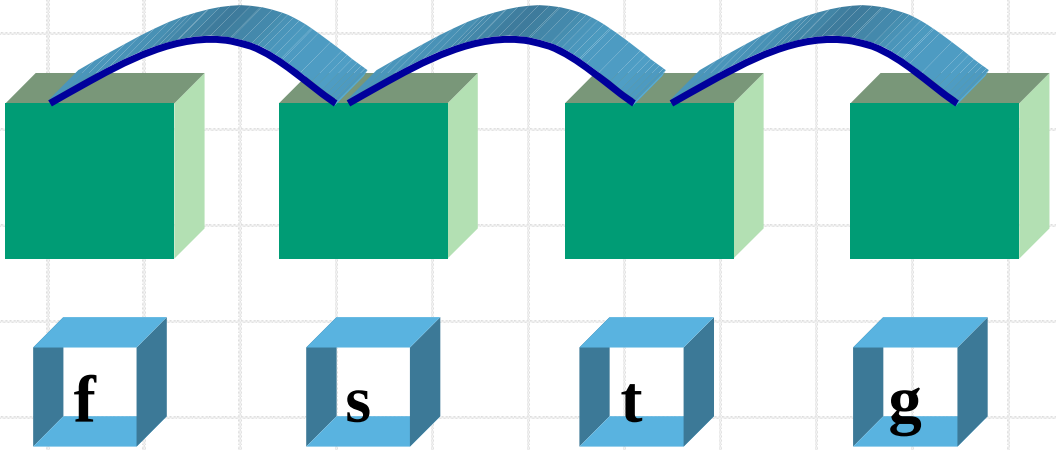
硬件或软件（如外设、共享代码段、共享数据结构），多个进程在对其进行访问时（关键是进行写入或修改），必须互斥地进行——有些共享资源可以同时访问，如只读数据;
临界区：不允许进程交叉执行的一段程序称为临界区

- 进程间资源访问冲突
  - 共享变量的修改冲突
  - 操作顺序冲突
- 进程间的制约关系
  - 间接制约：进行竞争——独占分配到的部分或全部共享资源，“互斥”
  - 直接制约：进行协作——等待来自其他进程的信息，“同步”。由此带来的进行运行时间延长。

一个飞机订票系统，两个终端，运行T1、T2进程

```
T1 :                    T2:
...                     ...
Read(x);                Read(x);
if x>=1 then            if x>=1 then
    x:=x-1;                 x:=x-1;
write(x);               write(x);
...                     ...
```

共享变量的修改冲突



有3个进程：get, copy和put，它们对4个存储区域f、s、t和g进行操作。

操作顺序冲突

	f	s	t	g	结果
初始状态	3,4,...,m	2	2	(1,2)	
g,c,p	4,5,...,m	3	3	(1,2,3)	正确
g,p,c	4,5,...,m	3	3	(1,2,2)	错误
c,g,p	4,5,...,m	3	2	(1,2,2)	错误
c,p,g	4,5,...,m	3	2	(1,2,2)	错误
p,c,g	4,5,...,m	3	2	(1,2,2)	错误
p,g,c	4,5,...,m	3	3	(1,2,2)	错误

有6种可能的操作顺序，只有一种结果是正确的！

3.4.1.2临界区的访问过程

```
entry section
  critical section
exit section
  remainder section
```

- 临界区(critical section)：进程中访问临界资源的一段代码。
- 进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应"正在访问临界区"标志
- 退出区(exit section)：用于将"正在访问临界区"标志清除。
- 剩余区(remainder section)：代码中的其余部分。

3.4.1.3同步机制应遵循的准则

- 空闲则入：其他进程均不处于临界区；
- 忙则等待：已有进程处于其临界区；
- 有限等待：等待进入临界区的进程不能"死等"；
- 让权等待：不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

3.4.1.4进程互斥的软件方法

算法1：单标志

♦有两个进程Pi, Pj，其中的Pi

```
while (turn != i);
    critical section
turn = j;
    remainder section
```

设立一个公用整型变量 turn：描述允许进入临界区的进程标识

在进入区循环检查是否允许本进程进入：turn为i时，进程Pi可进入；

在退出区修改允许进入进程标识：进程Pi退出时，改turn为进程Pj的标识j；

- 缺点：强制轮流进入临界区，没有考虑进程的实际需要。容易造成资源利用不充分：在Pi出让临界区之后，Pj使用临界区之前，Pi不可能再次使用临界区；

算法2：双标志、先检查

```
while (flag[j]);                    〈a〉
flag[i] = TRUE;                     〈b〉
    critical section
flag[i] = FALSE;
    remainder section
```

- 设立一个标志数组flag[]：描述进程是否在临界区，初值均为FALSE。
  - 先检查，后修改：在进入区检查另一个进程是否在临界区，不在时修改本进程在临界区的标志；
  - 在退出区修改本进程在临界区的标志；

- 优点：不用交替进入，可连续使用；
- 缺点：Pi和Pj可能同时进入临界区。按下面序列执行时，会同时进入："Pi<a> Pj<a> Pi<b> Pj<b>"。即在检查对方flag之后和切换自己flag之前有一段时间，结果都检查通过。这里的问题出在检查和修改操作不能连续进行。

算法3：双标志、后检查

```
flag[i] = TRUE;                <b>
while (flag[j]);               <a>
  critical section
flag[i] = FALSE;
  remainder section
```

- 类似于算法2，与互斥算法2的区别在于先修改后检查。可防止两个进程同时进入临界区。

- 缺点：Pi和Pj可能都进入不了临界区。按下面序列执行时，会都进不了临界区："Pi<a> Pj<a> Pi<b> Pj<b>"。即在切换自己flag之后和检查对方flag之前有一段时间，结果都切换flag，都检查不通过。

算法4(Peterson's Algorithm)：
先修改、后检查、后修改者等待

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

- 结合算法1和算法3，是正确的算法
- turn=j;描述可进入的进程（同时修改标志时）
- 在进入区先修改后检查，并检查并发修改的先后：
  - 检查对方flag，如果不在临界区则自己进入——空闲则入
  - 否则再检查turn：保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入——先到先入，后到等待

3.4.2 信号量(semaphore)

前面的互斥算法都存在问题，它们是平等进程间的一种协商机制，需要一个地位高于进程的管理者来解决公有资源的使用问题。OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公有资源的有效手段。

信号量代表可用资源实体的数量。

4.4.2.1 信号量和P、V原语

4.4.2.2 信号量集

3.4.2.1 信号量和P、V原语

- 1965年，由荷兰学者Dijkstra提出（所以P、V分别是荷兰语的pass(proberen)和increment(verhogen)），是一种卓有成效的进程同步与互斥机制。
- 每个信号量sem除一个整数值s.count（计数）外，还有一个进程等待队列s.queue，其中是阻塞在该信号量的各个进程的标识
  - 信号量只能通过初始化和两个标准的原语来访问——作为OS核心代码执行，不受进程调度的打断
  - 初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"）——若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数
- "二进制信号量(binary semaphore)"：只允许信号量取0或1值

1. P原语wait(s)

```
--s.count;                  //表示申请一个资源;
if (s.count <0)             //表示没有空闲资源;
{
    调用进程进入等待队列s.queue;
    阻塞调用进程;
}
Else { critical section, V}
```

2. V原语signal(s)

V原语通常唤醒进程等待队列中的头一个进程

```
++s.count;                  //表示释放一个资源;
if (s.count <= 0)//表示有进程处于阻塞状态;
{
    从等待队列s.queue中取出一个进程P;
    进程P进入就绪队列;
}
```

3.利用信号量实现互斥

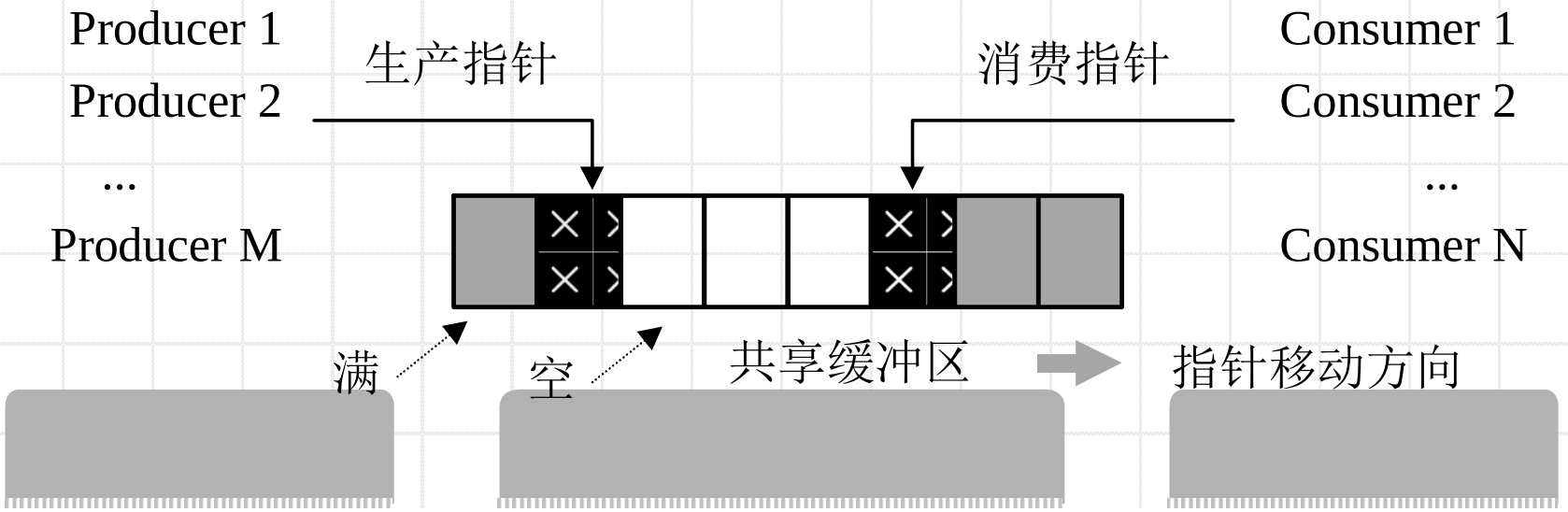
```
P(mutex);
    critical section
V(mutex);
    remainder section
```

- 为临界资源设置一个互斥信号量mutex(MUTual Exclusion)，其初值为1；在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间
- 必须成对使用P和V原语：遗漏P原语则不能保证互斥访问，遗漏V原语则不能在使用临界资源之后将其释放（给其他等待的进程）；P、V原语不能次序错误、重复或遗漏

3.4.3 经典进程同步问题

问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。

同步：多个进程，因直接制约而互发消息进行相互合作、互相等待，使得各进程之间按着一定的速度执行。



- 采用信号量机制：
  - full是"满"数目，初值为0，empty是"空"数目，初值为N。实际上，full和empty是同一个含义：full + empty == N
  - mutex用于访问缓冲区时的互斥，初值是1
- 每个进程中各个P操作的次序是重要的：先检查资源数目，再检查是否互斥——否则可能死锁(为什么？)

Producer

```
P(empty);
P(mutex);    //进入区
  one unit --> buffer;
V(mutex);
V(full);     //退出区
```

Consumer

```
P(full);
P(mutex);    //进入区
  one unit <-- buffer;
V(mutex);
V(empty);    //退出区
```

3.5 进程间通信
(IPC, INTER-PROCESS COMMUNICATION)

返回

3.5.0 进程间通信的类型

1. 低级通信和高级通信

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。优点的速度快。缺点是：
  - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
  - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：能够传送任意数量的数据，包括三类：共享存储区、管道、消息。

返回

2. 直接通信和间接通信

- 直接通信：信息直接传递给接收方，如管道。
  - 在发送时，指定接收方的地址或标识，也可以指定多个接收方或广播式地址；
  - 在接收时，允许接收来自任意发送方的消息，并在读出消息的同时获取发送方的地址。
- 间接通信：借助于收发双方进程之外的共享数据结构作为通信中转，如消息队列 MQ。通常收方和发方的数目可以是任意的。

3. 高级通信的特征

- 通信链路(communication link)：
  - 点对点/多点/广播
  - 单向/双向
  - 有容量（链路带缓冲区）/无容量（发送方和接收方需自备缓冲区）
- 数据格式：
  - 字节流(byte stream)：各次发送之间的分界，在接收时不被保留，没有格式；
  - 报文(datagram/message)：各次发送之间的分界，在接收时被保留，通常有格式（如表示类型），定长/不定长报文，可靠报文/不可靠报文。
- 收发操作的同步方式
  - 发送阻塞（直到被链路容量或接收方所接受）和不阻塞（失败时立即返回）
  - 接收阻塞（直到有数据可读）和不阻塞（无数据时立即返回）
  - 由事件驱动收发：在允许发送或有数据可读时，才做发送和接收操作

3.5.1 信号(signal)

信号相当于给进程的“软件”中断；进程可发送信号，指定信号处理例程；它是单向和异步的。

4.5.1.1 UNIX信号

3.5.1.1 UNIX信号

1. 信号类型

- 一个进程向另一个进程或进程组（或自己）发送（kill系统调用）：发送者必须具有接收者同样的有效用户ID，或者发送者是超级用户身份
- 某些键盘按键，如：中断字符（通常是Ctrl+C或Del）、暂停字符（如Ctrl+Z）
- 硬件条件，如：除数为零、浮点运算错、访问非法地址等异常条件
- 软件条件，如：Socket中有加急数据到达

2. 对信号的处理

- 进程可以设置信号处理例程（signal系统调用），在接收到信号时就被调用，称为"捕获"该信号。信号处理例程的参数是接收到信号的编号。
- 进程也可以忽略指定的信号(SIG_IGN)。
  - 只有SIGKILL信号（无条件终止进程）和SIGSTOP（使进程暂停）不能被忽略。
  - 在库函数system()的实现中，通过fork和exec加载新程序之后，在父进程中对SIGINT和SIGQUIT都要忽略，然后wait直到子进程终止，才恢复对SIGINT和SIGQUIT的原有处理例程。
- 进程创建后为信号设立了默认处理例程(SIG_DFL)，如：终止并留映象文件(core)

3.5.2 共享存储区(shared memory)

相当于内存，可以任意读写和使用任意数据结构（当然，对指针要注意），需要进程互斥和同步的辅助来确保数据一致性

1. UNIX的共享存储区

- 创建或打开共享存储区(shmget)：依据用户给出的整数值key，创建新区或打开现有区，返回一个共享存储区ID。
- 连接共享存储区(shmat)：连接共享存储区到本进程的地址空间，可以指定虚拟地址或由系统分配，返回共享存储区首地址。
- 拆除共享存储区连接(shmdt)：拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl)：对共享存储区进行控制。如：共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)；

返回

3.5.3 管道(pipe)

管道是一条在进程间以字节流方式传送的通信通道。它由OS核心的缓冲区（通常几十KB）来实现，是单向的；常用于命令行所指定的输入输出重定向和管道命令。在使用管道前要建立相应的管道，然后才可使用。

Command1 | command2

返回

1. UNIX管道

- 通过pipe系统调用创建无名管道，得到两个文件描述符，分别用于写和读。
  - int pipe(int fd[2]);
  - 文件描述符fd[0]为读端，fd[1]为写端；
  - 通过系统调用write和read进行管道的写和读；
  - 进程间双向通信，通常需要两个管道；
  - 只适用于父子进程之间或父进程安排的各个子进程之间；
- UNIX中的命名管道，可通过mknod系统调用建立：指定mode为S_IFIFO
  - int mknod(const char *path, mode_t mode, dev_t dev);

3.5.4 消息(message)

与窗口系统中的“消息”不同。通常是不定长数据块。消息的发送不需要接收方准备好，随时可发送。

返回

1. UNIX消息

- 消息队列(message queue)：每个message不定长，由类型(type)和正文(text)组成
- UNIX消息队列API：
  - msgget依据用户给出的整数值key，创建新消息队列或打开现有消息队列，返回一个消息队列ID；
  - msgsnd发送消息；
  - msgrcv接收消息，可以指定消息类型；没有消息时，返回-1；
  - msgctl对消息队列进行控制，如删除消息队列；
- 通过指定多种消息类型，可以在一个消息队列中建立多个虚拟信道
- 注意：消息队列不随创建它的进程的终止而自动撤销，必须用msgctl(msgqid, IPC_RMID, 0)。

3.5.5 套接字(socket)

- 双向的，数据格式为字节流（一对一）或报文（多对一，一对多）；主要用于网络通信；
- 支持client-server模式和peer-to-peer模式，本机或网络中的两个或多个进程进行交互。提供TCP/IP协议支持
- UNIX套接字(基于TCP/IP)：send, sendto, recv, recvfrom;
- 在Windows NT中的规范称为"Winsock"(与协议独立，或支持多种协议)：WSASend, WSASendto, WSARecv, WSARecvfrom;

返回

3.6 死锁问题(DEADLOCK)

3.6.1 概述

3.6.2 死锁排除

返回

概述

- 定义：指并发的各进程之间彼此等待对方拥有的资源。
- 本质原因：资源的个数小于进程的数量
- 死锁条件：
  - 互斥条件：排他性控制
  - 不剥夺
  - 部分分配：得到的资源，在申请新资源是不释放
  - 环路：环形链，每个等待上一个资源

死锁排除

- 预防：限制并发进程对资源的请求
  - 打破互斥和不可剥夺
  - 打破部分分配
  - 打破环路
- 避免：预测死锁的可能性，并避免
- 死锁检测与恢复
  - 用有限状态机或者PetriNet来检测请求和保持的环路
  - 通过强迫释放的手段，释放资源

第四章 处理机调度

处理机管理的工作是对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源。这里的主要问题是处理机调度算法和调度算法特征分析。

DISPATCH

4.1 引言

4.1.1 调度的类型(scheduling)

4.1.2 调度的性能准则

4.1.3 进程调度

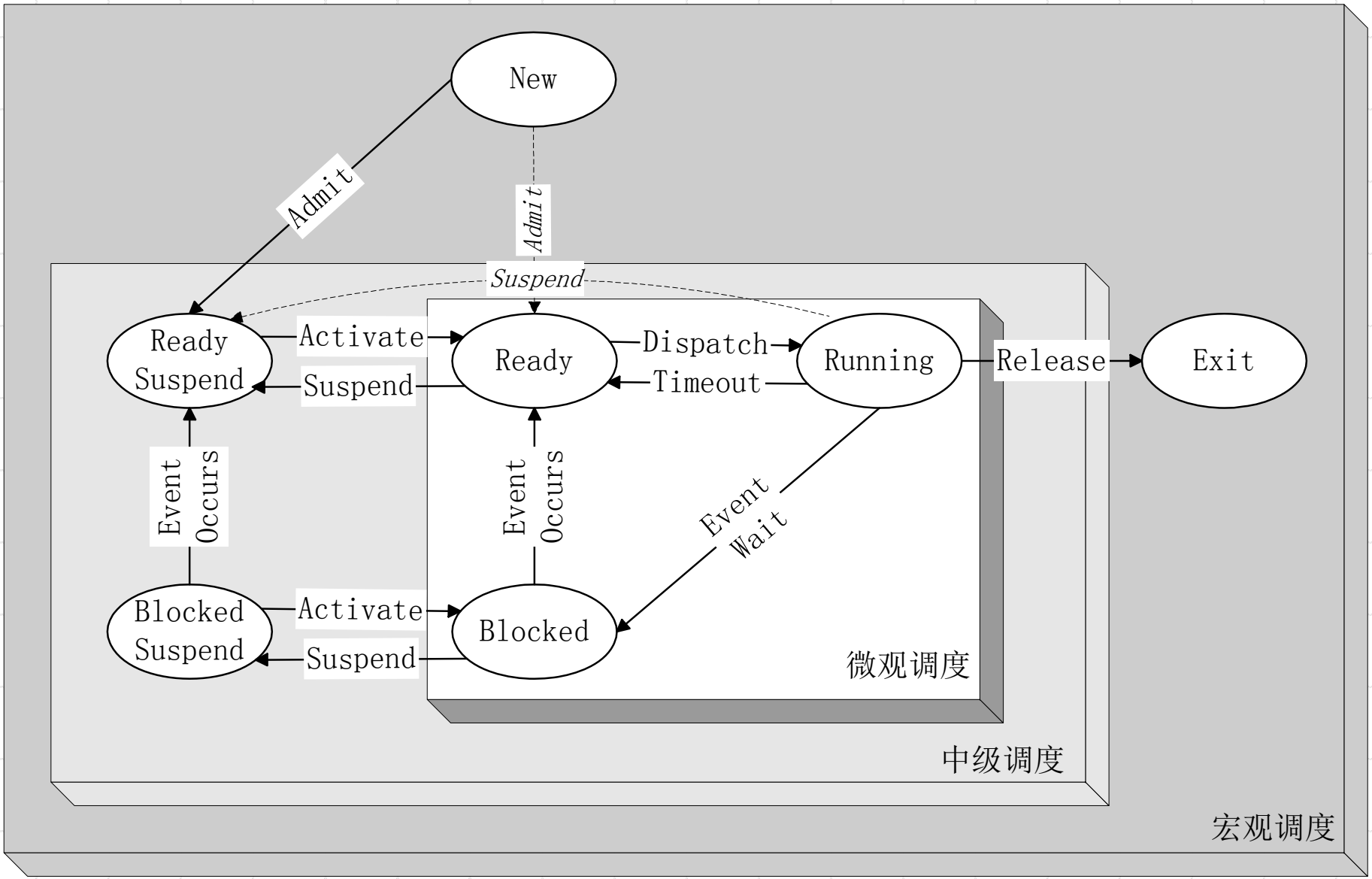
返回

4.1.1 调度的类型(scheduling)

从处理机调度的对象、时间、功能等不同角度，我们可把处理机调度分成不同类型。

1. 按照调度的层次

- 作业：又称为"宏观调度"、"高级调度"。从用户工作流程的角度，一次提交的若干个流程，其中每个程序按照进程调度。时间上通常是分钟、小时或天。
- 内外存交换：又称为"中级调度"。从存储器资源的角度。将进程的部分或全部换出到外存上，将当前所需部分换入到内存。指令和数据必须在内存里才能被CPU直接访问。
- 进程或线程：又称为"微观调度"、"低级调度"。从CPU资源的角度，执行的单位。时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。


New
Admit
Admit
Suspend
Ready
Suspend
Activate
Suspend
Ready
Dispatch
Timeout
Running
Release
Exit
Event
Occurs
Event
Occurs
Event
Wait
Blocked
Suspend
Activate
Suspend
Blocked
微观调度
中级调度
宏观调度

处理机调度的层次

2. 按照OS的分类

- 批处理调度——应用场合：大中型主机集中计算，如工程计算、理论计算（天气预报）
- 分时调度、实时调度：通常没有专门的作业调度
- 多处理机调度

4.1.2 调度的性能准则

我们可从不同的角度来判断处理机调度算法的性能，如用户的角度、处理机的角度和算法实现的角度。实际的处理机调度算法选择是一个综合的判断结果。

1. 面向用户的调度性能准则

- 周转时间：作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——批处理系统
  - 平均周转时间T
  - 平均带权周转时间（带权周转时间W是 T(周转)/T(CPU执行)〕
- 响应时间：用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统
- 截止时间：开始截止时间和完成截止时间——实时系统，与周转时间有些相似。
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间。
- 优先级：可以使关键任务达到更好的指标。

2. 面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，而每个周转时间是1小时，则吞吐量是2个作业/小时
- 处理机利用率：——大中型主机
- 各种设备的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——大中型主机

3. 调度算法本身的调度性能准则

4.1.3 进程调度

- 功能：调度程序(dispatcher)
  - 记录所有进程的运行状况（静态和动态）
  - 当进程出让CPU或调度程序剥夺执行状态进程占用的CPU时，选择适当的进程分派CPU
  - 完成上下文切换
- 进程的上下文切换过程：
  - 用户态执行进程A代码——进入OS核心（通过时钟中断或系统调用）
  - 保存进程A的上下文，恢复进程B的上下文（CPU寄存器和一些表格的当前指针）
  - 用户态执行进程B代码
- 注：上下文切换之后，指令和数据快速缓存cache通常需要更新，执行速度降低

4.2 调度算法

通常将作业或进程归入各种就绪或阻塞队列。有的算法适用于作业调度，有的算法适用于进程调度，有的两者都适应。

4.2.1 先来先服务

4.2.2 短作业优先

4.2.3 时间片轮转算法

4.2.4 多级队列算法

4.2.5 优先级算法

4.2.6 多级反馈队列算法

返回

4.2.1 先来先服务 (FCFS, First Come First Service)

这是最简单的调度算法，按先后顺序进行调度。

- 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
- 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）。
- 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU。最简单的算法。

2. FCFS的特点

- 比较有利于长作业，而不利于短作业。
- 有利于CPU繁忙的作业，而不利于I/O繁忙的作业。

4.2.2 短作业优先 (SJF, Shortest Job First)

又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。

当一个进程P Ready，

读取进程的预计执行时间 PT

在队列Q中，找到两个进程，I与I+1

IT<PT<（I+1）T

选择队列首位的进程进入CPU

对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业。

2. SJF的特点

- 优点：
  - 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；
  - 提高系统的吞吐量；
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行；
  - 未能依据作业的紧迫程度来划分执行的优先级；
  - 难以准确估计作业（进程）的执行时间，从而影响调度性能。

3. SJF的变型

- "最短剩余时间优先"SRT(Shortest Remaining Time)
  - 允许比当前进程剩余时间更短的进程来抢占
  - P1 30, P2 25　P1 30 20 10　P2 25 0 25
- "最高响应比优先"HRRN(Highest Response Ratio Next)
  - 响应比R = (等待时间 / 要求执行时间) 执行时间
  - 是FCFS和SJF的折衷

4.2.3 时间片轮转(Round Robin)算法

前两种算法主要用于宏观调度，说明怎样选择一个进程或作业开始运行，开始运行后的作法都相同，即运行到结束或阻塞，阻塞结束时等待当前进程放弃CPU 。本算法主要用于微观调度，说明怎样并发运行，即切换的方式；设计目标是提高资源利用率。

其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；

1. 时间片轮转算法

- 将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- 每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
- 在一个时间片结束时，发生时钟中断。
- 调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
- 进程可以未使用完一个时间片，就出让CPU（如阻塞）。

2.时间片长度的确定

- 时间片长度变化的影响
  - 过长—>退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
  - 过短—>用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长。
- 对响应时间的要求：
  - T(响应时间)=N(进程数目)*q(时间片)
- 时间片长度的影响因素：
  - 就绪进程的数目：数目越多，时间片越小（当响应时间一定时）
  - 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则使响应时间，平均周转时间和平均带权周转时间延长。

4.2.4 多级队列算法 (Multiple-level Queue)

本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；

- 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
- 每个作业固定归入一个队列。
- 各队列的不同处理：不同队列可有不同的优先级、时间片长度、调度策略等。如：系统进程、用户交互进程、批处理进程等。

4.2.5 优先级算法(Priority Scheduling)

本算法是多级队列算法的改进，平衡各进程对响应时间的要求。适用于作业调度和进程调度，可分成抢先式和非抢先式；

5.2.5.1 静态优先级

5.2.5.2 动态优先级

5.2.5.3 线性优先级调度算法（SRR, Selfish Round Robin）

4.2.5.1 静态优先级

创建进程时就确定，直到进程终止前都不改变。通常是一个整数。

P=F(X1,X2,X3,X4)

- 依据：
  - 进程类型（系统进程优先级较高）
  - 对资源的需求（对CPU和内存需求较少的进程，优先级较高）
  - 用户要求（紧迫程度和付费多少）

4.2.5.2 动态优先级

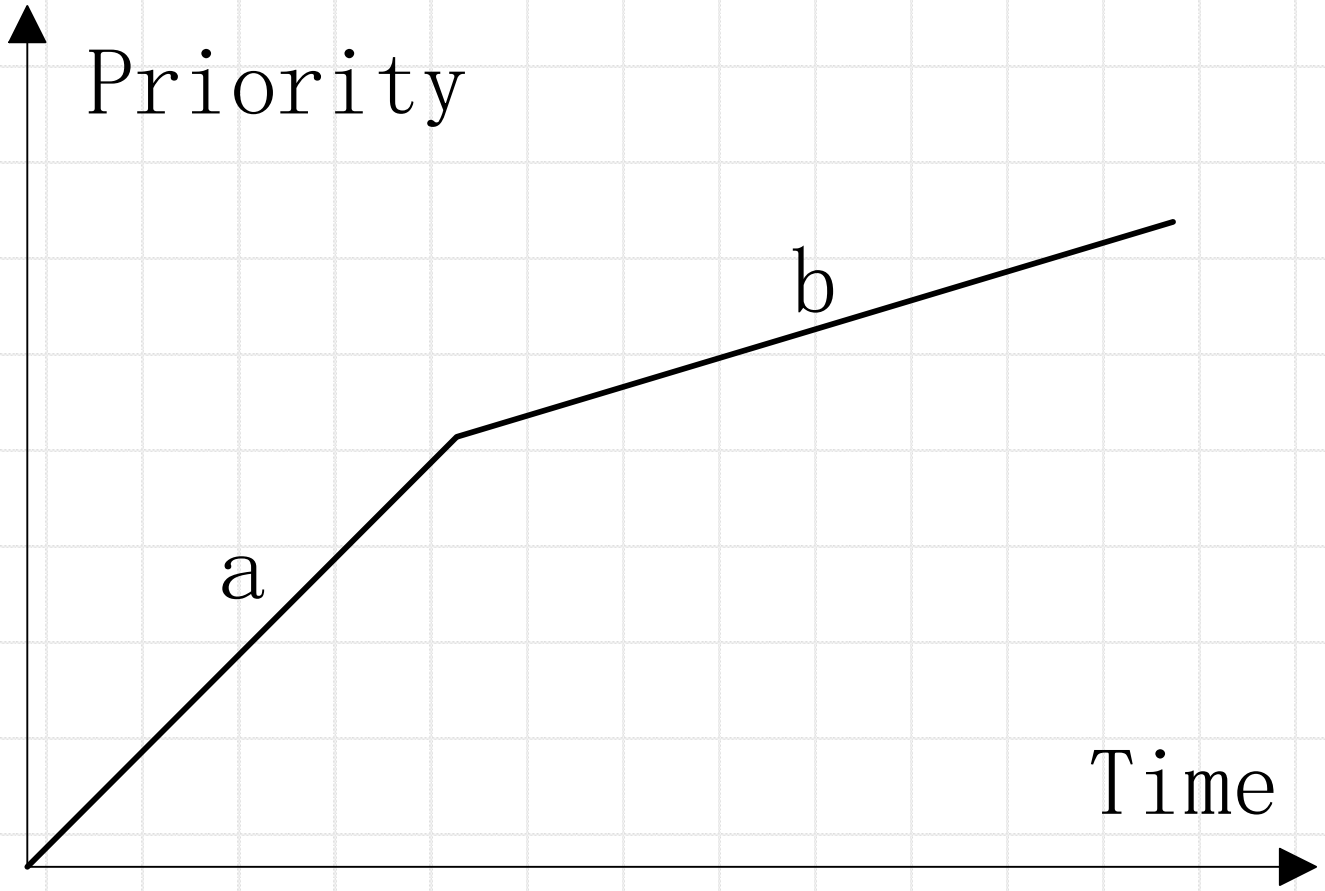
在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：

- 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
- 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

4.2.5.3 线性优先级调度算法（SRR, Selfish Round Robin）

本算法是优先级算法的一个实例，它通过进程优先级的递增来改进长执行时间进程的周转时间特征。

- 就绪进程队列分成两个：
  - 新创建进程队列：按FCFS方式排成；进程优先级按速率a增加；
  - P11　(T11)*a ,P12 T12*a,P13 T13*a
  - 享受服务队列：已得到过时间片服务的进程按FCFS方式排成；进程优先级按速率b增加；
  - P21 c21+T21*b, P22　c22+ T22*b , P23 c23+T23*b ,P24 c24+T24*b
  - T11*a==c24+T24*b
  - P21 c21+T21*b, P22　c22+ T22*b , P23 c23+T23*b ,P24 c24+T24*b,
  - P21 c21 +0, P22　c22+ T22*b , P23 c23+T23*b ,P24 c24+T24*b,
- 新创建进程等待时间的确定：当新创建进程优先级与享受服务队列中最后一个进程优先级相同时，转入享受服务队列；



SRR算法的优先级变化

2. SRR算法与FCFS算法和时间片轮转算法的关系

- 当b>a>0时，享受服务队列中永远只有一个进程；SRR算法退化成FCFS算法；
- 当a>b=0时，SRR算法就是时间片轮转算法；

4.2.6 多级反馈队列算法 (Round Robin with Multiple Feedback)

- 多级反馈队列算法是时间片轮转算法和优先级算法的综合和发展。优点：
  - 为提高系统吞吐量和缩短平均周转时间而照顾短进程
  - 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
  - 不必估计进程的执行时间，动态调节

1. 多级反馈队列算法

- 设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列1的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍
- 新进程进入内存后，先投入队列1的末尾，按FCFS算法调度；若按队列1一个时间片未能执行完，则降低投入到队列2的末尾，同样按FCFS算法调度；如此下去，降低到最后的队列，则按"时间片轮转"算法调度直到完成。
- 仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

2. 几点说明

- I/O型进程：让其进入最高优先级队列，以及时响应I/O交互。通常执行一个小时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列。
- 计算型进程：每次都执行完时间片，进入更低级队列。最终采用最大时间片来执行，减少调度次数。
- I/O次数不多，而主要是CPU处理的进程：在I/O完成后，放回优先I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降。
- 为适应一个进程在不同时间段的运行特点，I/O完成时，提高优先级；时间片用完时，降低优先级；

4.3 调度算法性能分析

调度算法的性能通常是通过实验或计算得到的。

- FCFS, Round Robin, 线性优先级算法SRR(Selfish Round Robin)
- 周转时间
  - 长作业时：T(FCFS) < T(SRR) < T(RR)（运行时间是主要因素）
  - 短作业时：T(RR) < T(SRR) < T(FCFS)（等待时间是主要因素）

返回

4.4 实时调度

4.4.1 实时调度的特点

4.4.2 实时调度算法

返回

4.4.1 实时调度的特点

- 要求更详细的调度信息：如，就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时）。
- 采用抢先式调度。
- 快速中断响应，在中断处理时（硬件）关中断的时间尽量短。
- 快速上下文切换：相应地采用较小的调度单位（如线程）。

4.4.2 实时调度算法

- 静态表调度算法（Static table-driven scheduling）：适用于周期性的实时应用。通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。这种方法的特点是有效但不灵活。
- 静态优先级调度算法（Static priority-driven scheduling）：把通用的优先级调度算法用于实时系统，但优先级的确定是通过静态分析（运行时间、到达频率）完成的。

- 动态分析调度算法（Dynamic planning-based scheduling）：在任务下达后执行前进行调度分析，要求满足实时性要求。
- 无保障动态调度算法（Dynamic best effort scheduling）：在任务下达时分配优先级，开始执行，在时限到达时未完成的任务被取消。用于非周期性任务的实时系统。

第五章 存储管理

存储管理是指存储器资源（主要指内存并涉及外存）的管理。

- 存储器资源的组织（如内存的组织方式）
- 地址变换（逻辑地址与物理地址的对应关系维护）
- 虚拟存储的调度算法

5.1 引言

返回

5.1.1 存储组织

- 存储器的功能是保存数据，存储器的发展方向是高速、大容量和小体积。
  - 内存在访问速度方面的发展：DRAM、SDRAM、SRAM等；
  - 硬盘技术在大容量方面的发展：接口标准、存储密度等；
- 存储组织是指在存储技术和CPU寻址技术许可的范围内组织合理的存储结构。
  - 其依据是访问速度匹配关系、容量要求和价格。
  - “寄存器-内存-外存”结构
  - “寄存器-缓存-内存-外存”结构；
- 微机中的存储层次组织：
  - 访问速度越慢，容量越大，价格越便宜；
  - 最佳状态应是各层次的存储器都处于均衡的繁忙状态（如：缓存命中率正好使主存读写保持繁忙）；

返回

存储层次结构

- 快速缓存：
  - Data Cache
  - TLB(Translation Lookaside Buffer)
- 内存： DRAM, SDRAM等；
- 外存： 软盘、硬盘、光盘、磁带等；

5.1.2 存储管理的功能

- 存储分配和回收：分配和回收算法及相应的数据结构。
- 地址变换：
  - 可执行文件生成中的链接技术
  - 程序加载(装入)时的重定位技术
  - 进程运行时硬件和软件的地址变换技术和机构
- 存储共享和保护：
  - 代码和数据共享
  - 地址空间访问权限（读、写、执行）
- 存储器扩充：存储器的逻辑组织和物理组织；
  - 由应用程序控制：覆盖；
  - 由OS控制：交换（整个进程空间 SWAP），虚拟存储的请求调入和预调入（部分进程空间）

返回

5.1.3 重定位方法

- 重定位：在可执行文件装入时需要解决可执行文件中地址（指令和数据）和内存地址的对应。由操作系统中的装入程序loader来完成。
- 程序在成为进程前的准备工作
  - 编辑：形成源文件(符号地址)
  - 编译：形成目标模块(模块内符号地址解析)
  - 链接：由多个目标模块或程序库生成可执行文件(模块间符号地址解析)
  - 装入：构造PCB，形成进程(使用物理地址)

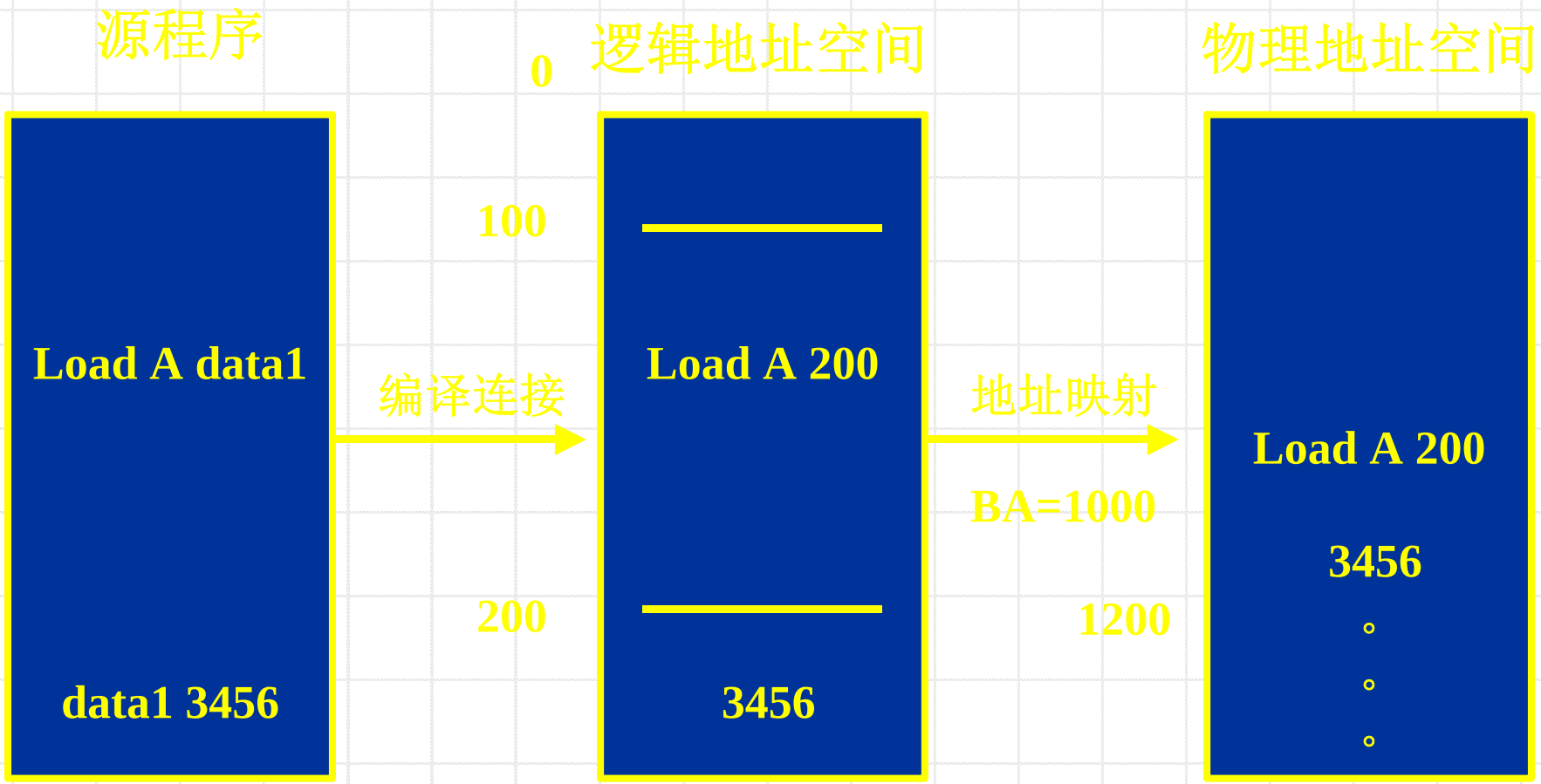
  A.C, B.C, C.C:::A.OBJ, B.OBJ, C.OBJ Compile

  LINK::::
- 重定位方法：
  - 绝对装入
  - 可重定位装入
  - 动态装入

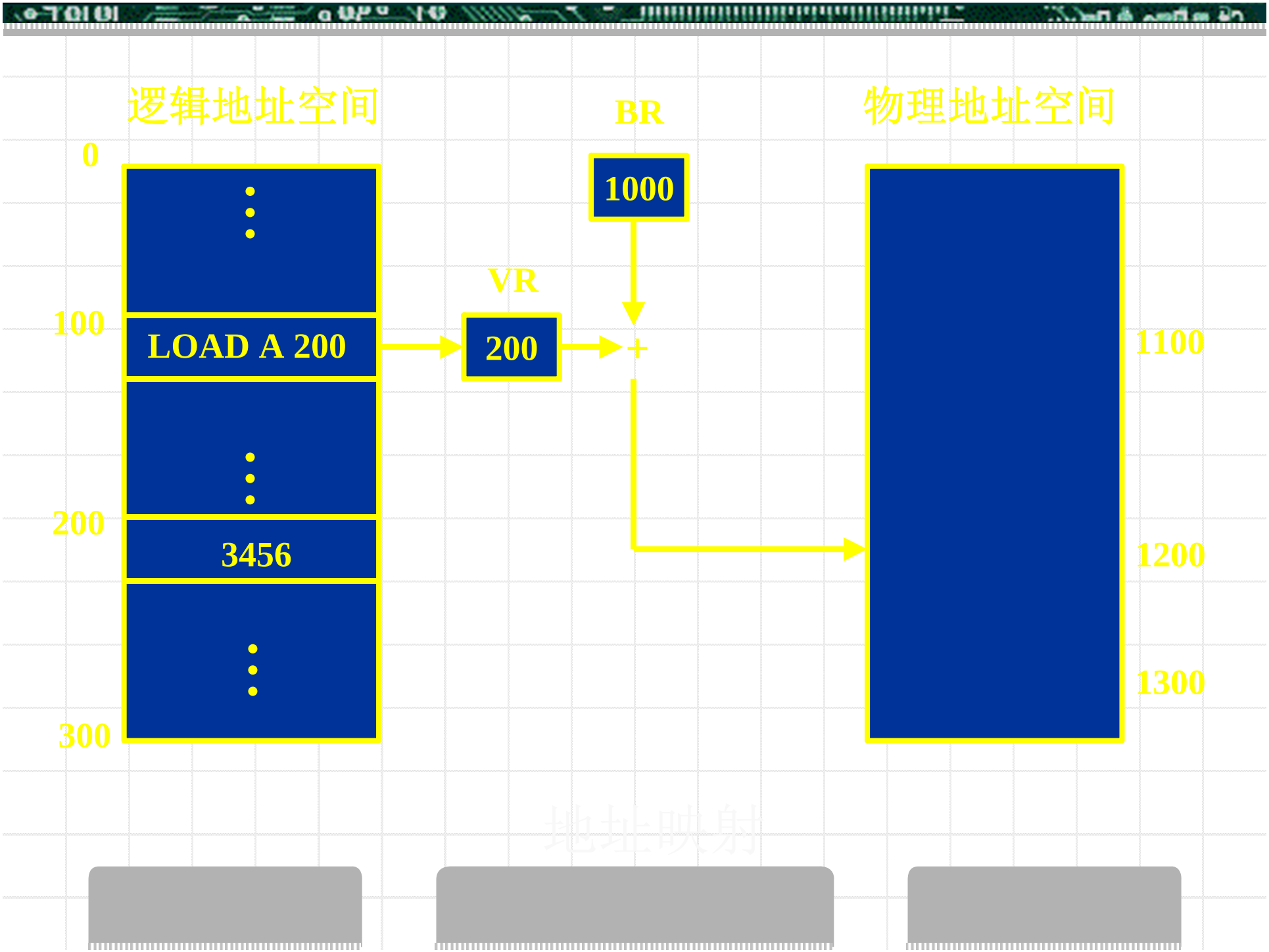
返回

1.逻辑地址、物理地址和地址映射

- 逻辑地址（相对地址，虚地址）：用户的程序经过汇编或编译后形成目标代码，目标代码通常采用相对地址的形式。
  - 其首地址为0，其余指令中的地址都相对于首地址来编址。
  - 不能用逻辑地址在内存中读取信息。
- 物理地址（绝对地址，实地址）：内存中存储单元的地址。物理地址可直接寻址。
- 地址映射：将用户程序中的逻辑地址转换为运行时由机器直接寻址的物理地址。
  - 当程序装入内存时,操作系统要为该程序分配一个合适的内存空间，由于程序的逻辑地址与分配到内存物理地址不一致,而CPU执行指令时，是按物理地址进行的，所以要进行地址转换。



逻辑地址、物理地址和地址映射


逻辑地址空间
BR
物理地址空间
0
1000
VR
100
LOAD A 200
200
+
1100
200
3456
1200
1300
300
地址映射

2. 绝对装入(absolute loading)

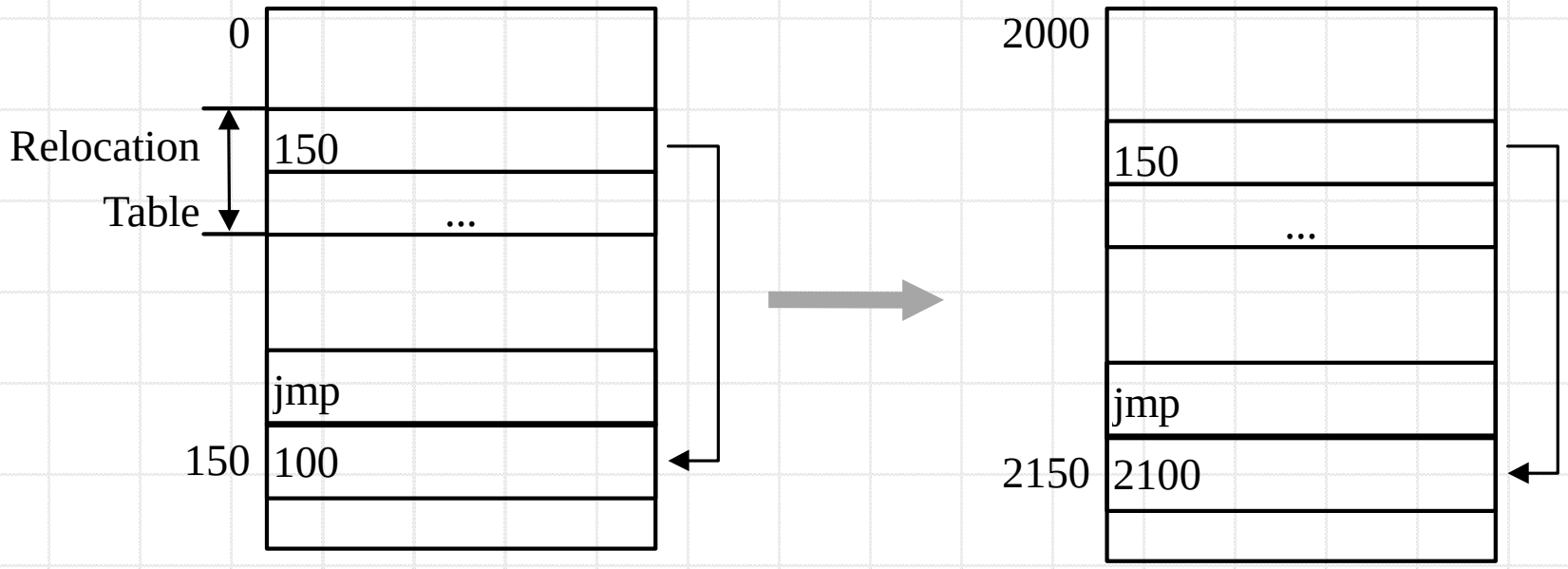
在可执行文件中记录内存地址，装入时直接定位在上述(即文件中记录的地址)内存地址。

- 优点：装入过程简单。
- 缺点：过于依赖于硬件结构，不适于多道程序系统。

3. 可重定位装入(relocatable loading)

在可执行文件中，列出各个需要重定位的地址单元和相对地址值。当用户程序被装入内存时，一次性实现逻辑地址到物理地址的转换，以后不再转换（一般在装入内存时由软件完成）。即：装入时根据所定位的内存地址去修改每个重定位地址项，添加相应偏移量。

- 优点：不需硬件支持，可以装入有限多道程序（如MS DOS中的TSR）。
- 缺点：一个程序通常需要占用连续的内存空间，程序装入内存后不能移动。不易实现共享。



说明：重定位表中列出所有修改的位置。如：重定位表的150表示相对地址150处的内容为相对地址(即100为从0起头的相对位置)。在装入时，要依据重定位后的起头位置(2000)修改相对地址。

重定位修改：重定位表中的150->绝对地址2150(=2000+150)

内容修改：内容100变成2100(=100+2000))。

4. 动态装入(dynamic run-time loading)

在可执行文件中记录虚拟内存地址，装入和执行时通过硬件地址变换机构，完成虚拟地址到实际内存地址的变换。

- 优点：
  - OS可以将一个程序分散存放于不连续的内存空间，可以移动程序，有利用实现共享。
  - 能够支持程序执行中产生的地址引用，如指针变量（而不仅是生成可执行文件时的地址引用）。
- 缺点：需要硬件支持（通常是CPU），OS实现较复杂。它是虚拟存储的基础。

5.1.4 链接

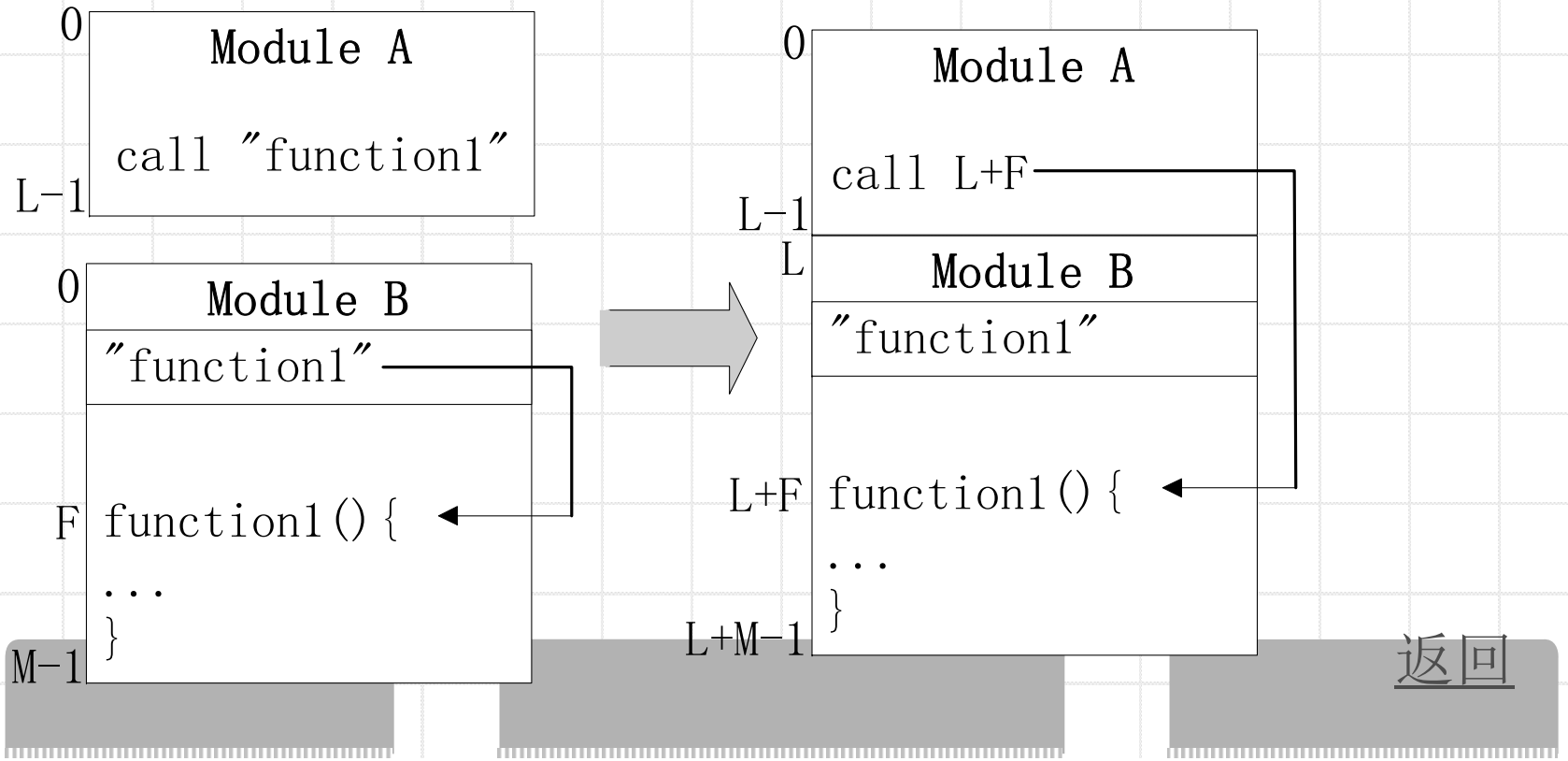
链接是指多个目标模块在执行时的地址空间分配和相互引用。

返回

5.1.4.1 链接方法

1. 静态链接(static-linking)

静态链接是在生成可执行文件时进行的。在目标模块中记录符号地址(symbolic address)，而在可执行文件中改写为指令直接使用的数字地址。



返回

2. 动态链接(dynamic-linking)

在装入或运行时进行链接。通常被链接的共享代码称为动态链接库(DLL, Dynamic-Link Library)或共享库(shared library)。

- 优点
  - 共享：多个进程可以共用一个DLL，节省内存，减少文件交换。
  - 部分装入：一个进程可以将多种操作分散在不同的DLL中实现，而只将当前操作相应的DLL装入内存。
  - 便于局部代码修改：即便于代码升级和代码重用；只要函数的接口参数（输入和输出）不变，则修改函数及其DLL，无需对可执行文件重新编译或链接。
  - 便于运行环境适应：调用不同的DLL，就可以适应多种使用环境和提供不同功能。如：不同的显示卡只需厂商为其提供特定的DLL，而OS和应用程序不必修改。
- 缺点：
  - 链接开销：增加了程序执行时的链接开销；
  - 管理开销：程序由多个文件组成，增加管理复杂度

5.2 分区存储管理

返回

5.2.1 原理

- 把内存分为一些大小相等或不等的分区(partition)，每个应用进程占用一个或几个分区。操作系统占用其中一个分区。
- 特点：适用于多道程序系统和分时系统
  - 支持多个程序并发执行
  - 难以进行内存分区的共享。
- 问题：可能存在内碎片和外碎片。
  - 内碎片：占用分区之内未被利用的空间
  - 外碎片：占用分区之间难以利用的空闲分区（通常是小空闲分区）。

返回

- 分区的数据结构：分区表，或分区链表
  - 可以只记录空闲分区，也可以同时记录空闲和占用分区
  - 分区表中，表项数目随着内存的分配和释放而动态改变，可以规定最大表项数目。
  - 分区表可以划分为两个表格：空闲分区表，占用分区表。从而减小每个表格长度。空闲分区表中按不同分配算法相应对表项排序。

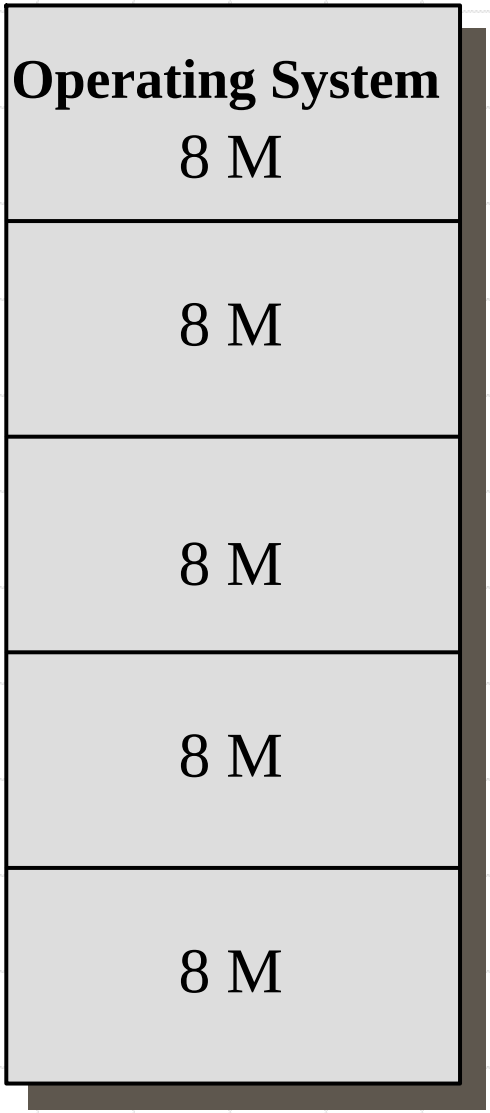
- 内存紧缩(compaction)：将各个占用分区向内存一端移动。使各个空闲分区聚集在另一端，然后将各个空闲分区合并成为一个空闲分区。
  - 对占用分区进行内存数据搬移占用CPU时间
  - 如果对占用分区中的程序进行"浮动"，则其重定位需要硬件支持。
  - 紧缩时机：每个分区释放后，或内存分配找不到满足条件的空闲分区时

5.2.2 固定分区(fixed partitioning)

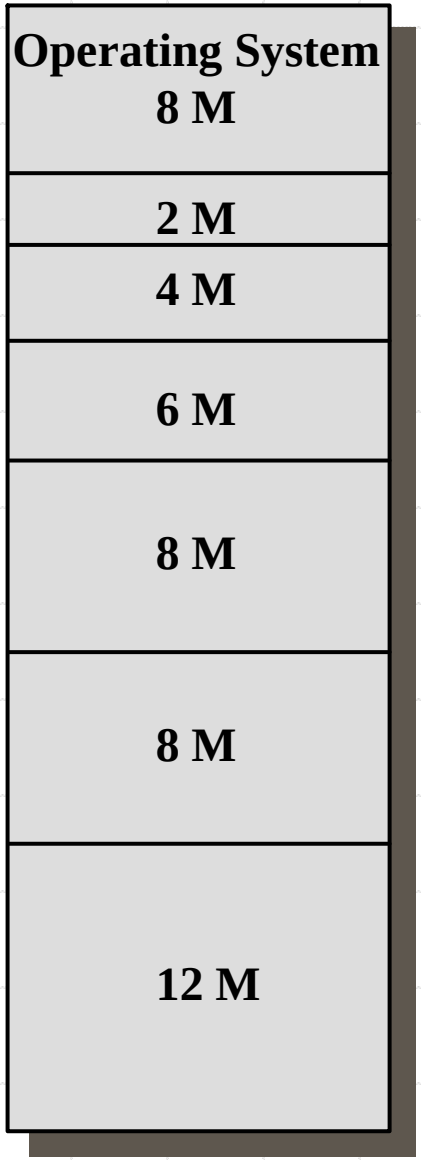
把内存划分为若干个固定大小的连续分区。

- 分区大小相等：只适合于多个相同程序的并发执行（处理多个类型相同的对象）。
- 分区大小不等：多个小分区、适量的中等分区、少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

返回



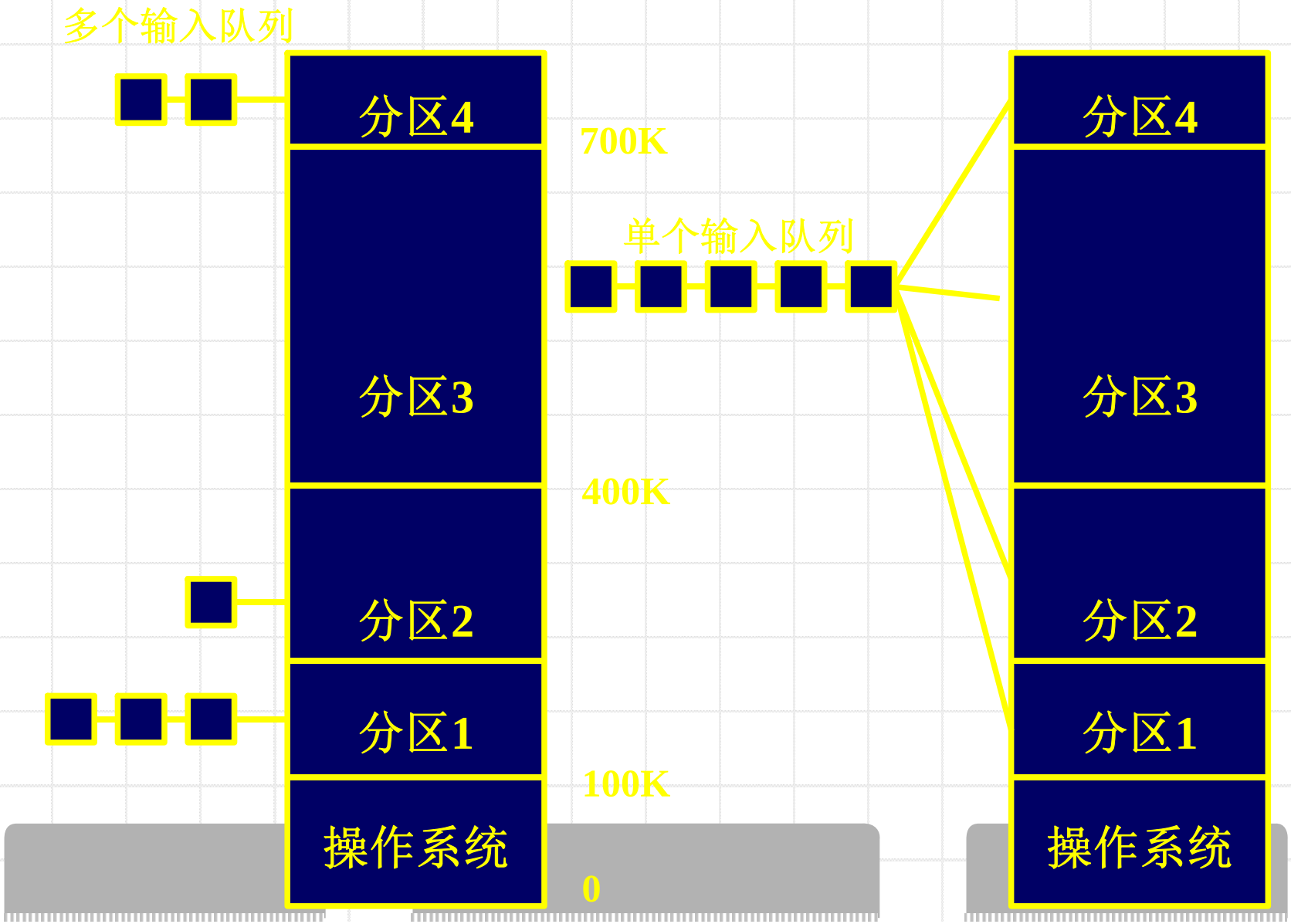
固定分区(大小相同)



固定分区(多种大小)

- 优点：易于实现，开销小。
- 缺点：
  - 内碎片造成浪费
  - 分区总数固定，限制了并发执行的程序数目。
- 可以和覆盖、交换技术配合使用。
- 采用的数据结构：分区表——记录分区的大小和使用情况

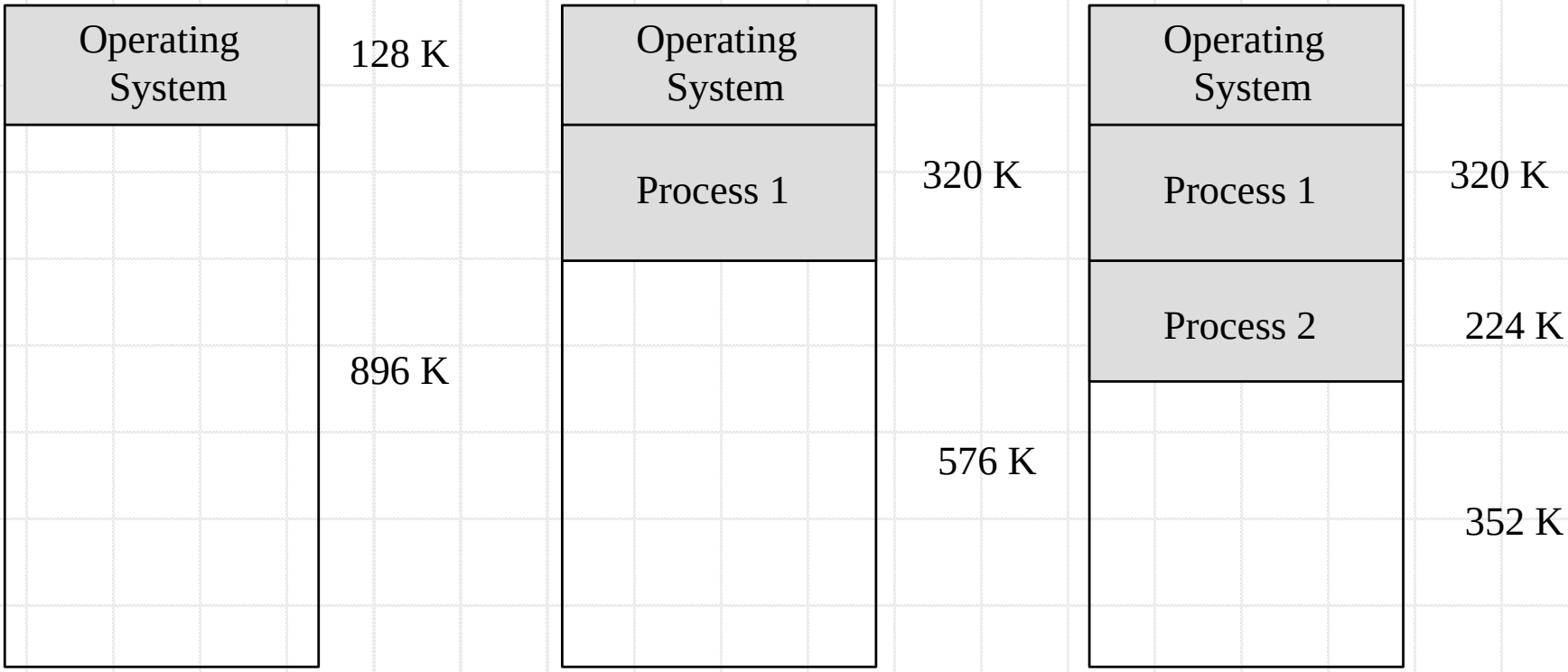
固定分区的进程队列：多队列适用于大小不等的固定分区；单队列适用于大小相同的固定分区，也可用于大小不等的情况。

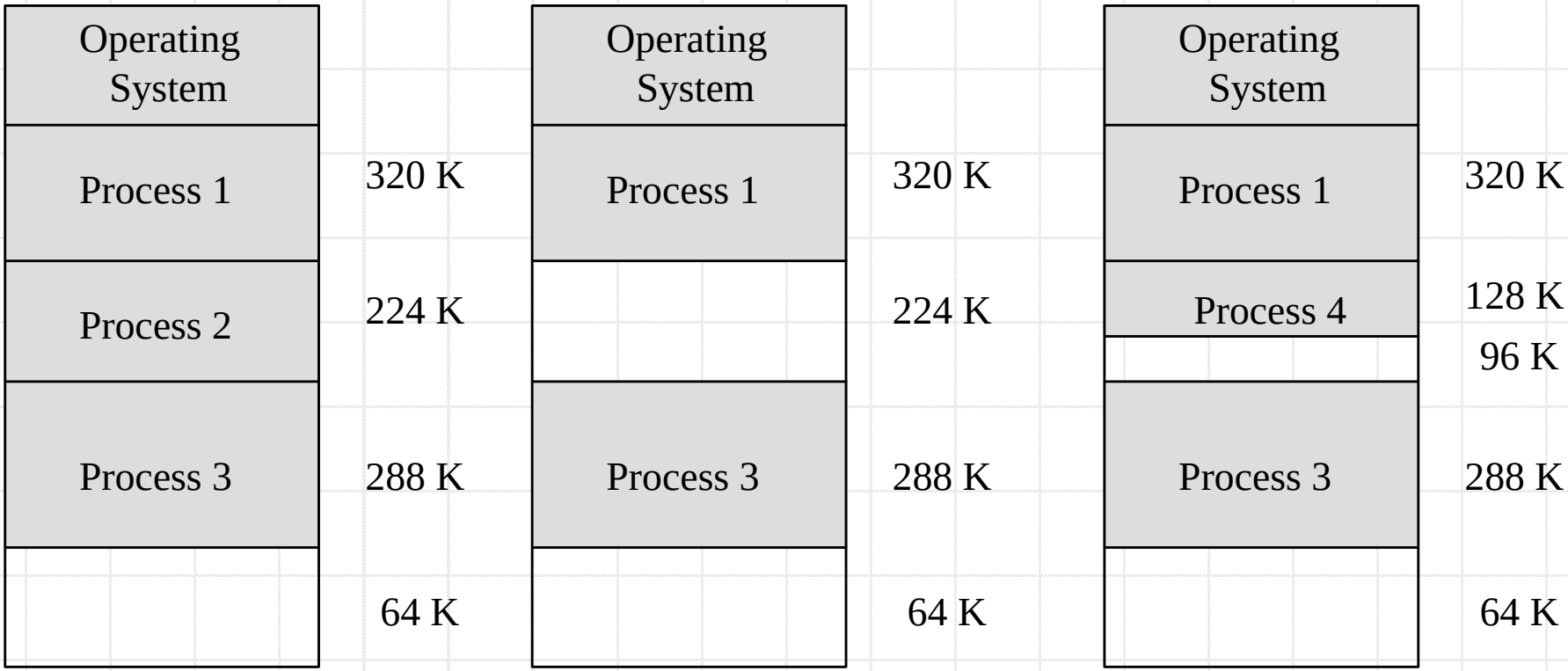


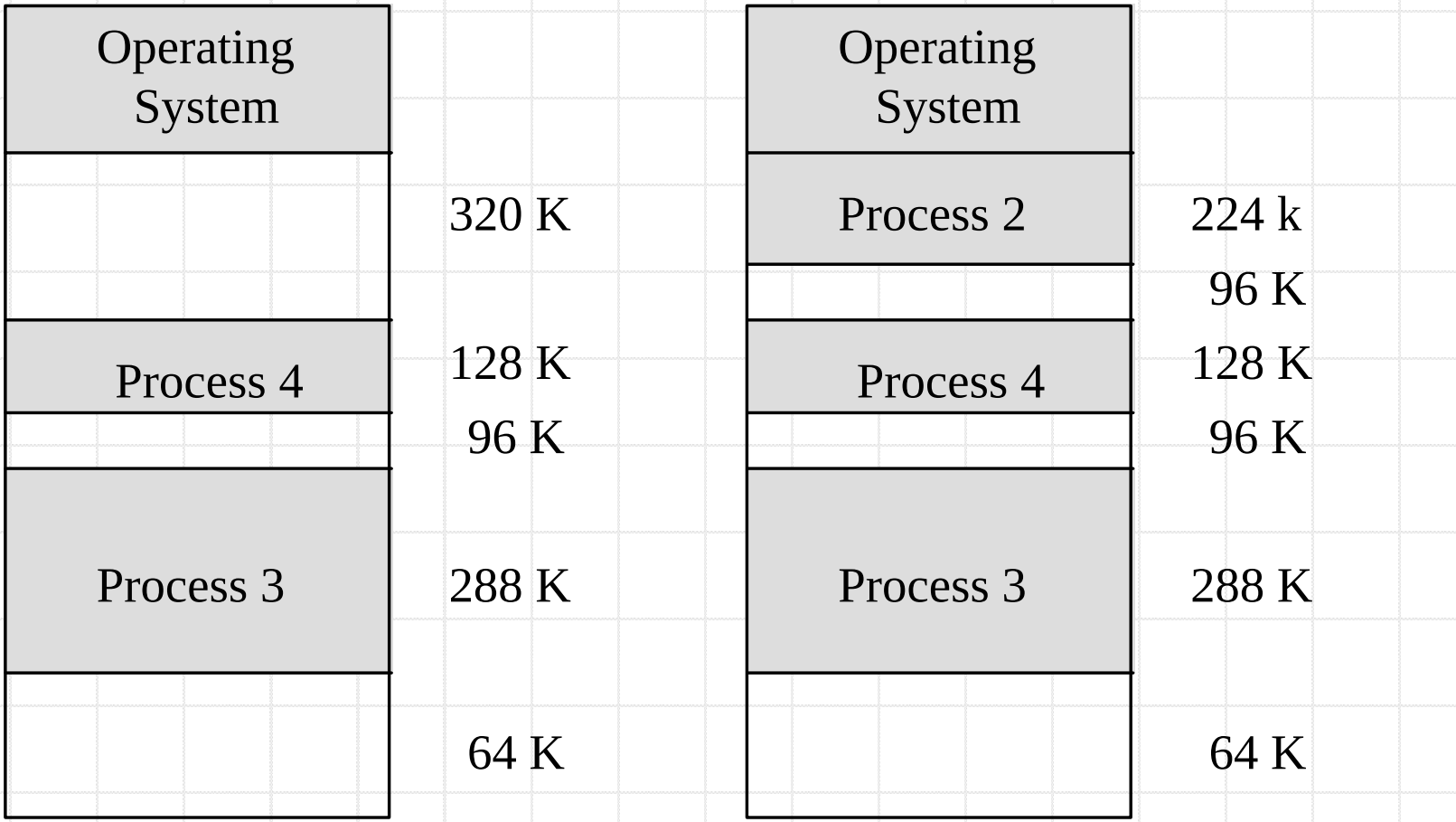
5.2.3 动态分区(dynamic partitioning)

- 动态创建分区：在装入程序时按其初始要求分配，或在其执行过程中通过系统调用进行分配或改变分区大小。
- 优点：没有内碎片。
- 缺点：有外碎片；如果大小不是任意的，也可能出现内碎片。

返回


Operating System
128 K
896 K
Operating System
Process 1
320 K
576 K
Operating System
Process 1
320 K
Process 2
224 K
352 K


Operating System
Process 1
Process 2
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 3
320 K
224 K
288 K
64 K
Operating System
Process 1
Process 4
Process 3
320 K
128 K
96 K
288 K
64 K


Operating System
320 K
Process 4
128 K
96 K
Process 3
288 K
64 K
Operating System
Process 2
224 k
96 K
Process 4
128 K
96 K
Process 3
288 K
64 K

5.2.4 分区分配算法

- 分区分配算法：寻找某个空闲分区，其大小需大于或等于程序的要求。若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区的先后次序通常是从内存低端到高端。
- 分区释放算法：需要将相邻的空闲分区合并成一个空闲分区。(这时要解决的问题是：合并条件的判断和合并时机的选择)

返回

- 最先匹配法(first-fit)：按分区的先后次序，从头查找，找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。
  - 但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。
- 下次匹配法(next-fit)：按分区的先后次序，从上次分配的分区起查找（到最后分区时再回到开头），找到符合要求的第一个分区
  - 该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大的空闲分区不易保留。
- 最佳匹配法(best-fit)：找到其大小与要求相差最小的空闲分区
  - 从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。较大的空闲分区可以被保留。
- 最坏匹配法(worst-fit)：找到最大的空闲分区
  - 基本不留下小空闲分区，但较大的空闲分区不被保留。

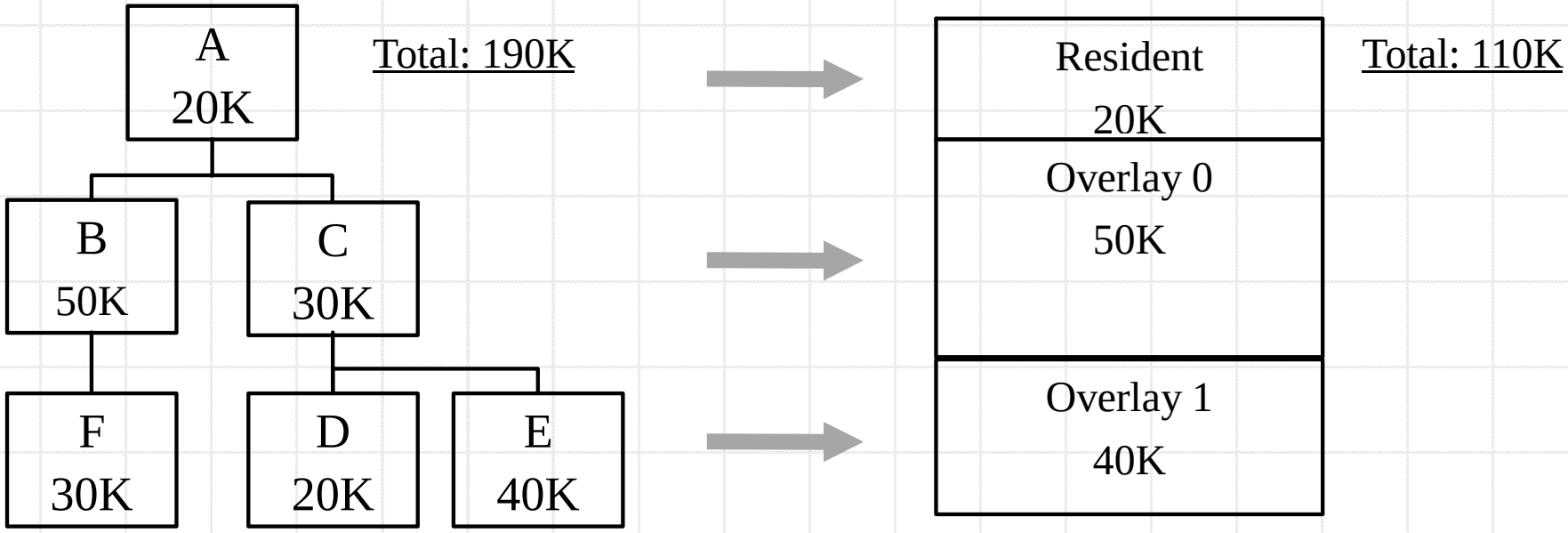
5.3 覆盖和交换技术

返回

5.3.1 覆盖(overlay)

- 引入：其目标是在较小的可用内存中运行较大的程序。常用于多道程序系统，与分区存储管理配合使用。
- 原理：一个程序的几个代码段或数据段，按照时间先后来占用公共的内存空间。
  - 将程序的必要部分（常用功能）的代码和数据常驻内存；
  - 可选部分（不常用功能）在其他程序模块中实现，平时存放在外存中（覆盖文件），在需要用到时才装入到内存；
  - 不存在调用关系的模块不必同时装入到内存，从而可以相互覆盖。(即不同时用的模块可共用一个分区)

返回



注：另一种覆盖方法：(100K)

- A(20K)占一个分区：20K；
- B(50K)、D(20K)和E(40K)共用一个分区：50K；
- F(30K)和C(30K)共用一个分区：30K；

- 缺点：
  - 编程时必须划分程序模块和确定程序模块之间的覆盖关系，增加编程复杂度。
  - 从外存装入覆盖文件，以时间延长来换取空间节省。
- 实现：函数库（操作系统对覆盖不得知），或操作系统支持

5.3.2 交换(swapping)

- 引入：多个程序并发执行，可以将暂时不能执行的程序送到外存中，从而获得空闲内存空间来装入新程序，或读入保存在外存中而目前到达就绪状态的进程。交换单位为整个进程的地址空间。常用于多道程序系统或小型分时系统中，与分区存储管理配合使用。又称作"对换"或"滚进/滚出(roll-in/roll-out)"；
  - 程序暂时不能执行的可能原因：处于阻塞状态，低优先级（确保高优先级程序执行）；
- 原理：暂停执行内存中的进程，将整个进程的地址空间保存到外存的交换区中（换出swap out），而将外存中由阻塞变为就绪的进程的地址空间读入到内存中，并将该进程送到就绪队列（换入swap in）。

返回

- 优点：增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构
- 缺点：对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。
- 考虑的问题：
  - 程序换入时的重定位；
  - 减少交换中传送的信息量，特别是对大程序；
  - 对外存交换区空间的管理：如动态分区方法；

5.4 页式和段式存储管理

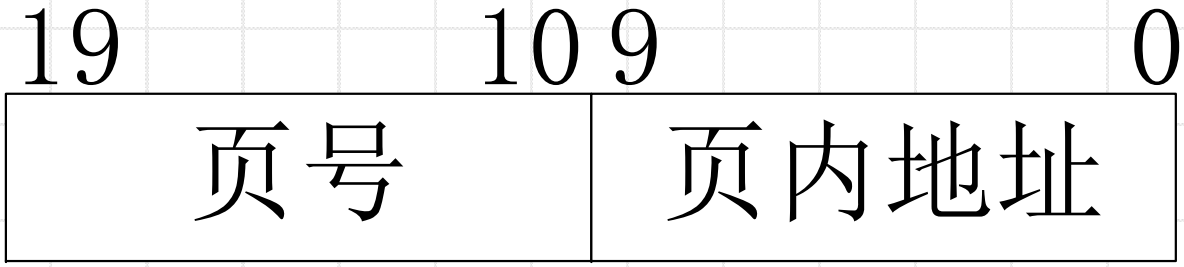
页式和段式存储管理是通过引入进程的逻辑地址，把进程地址空间与实际存储位置分离，从而增加存储管理的灵活性。

返回

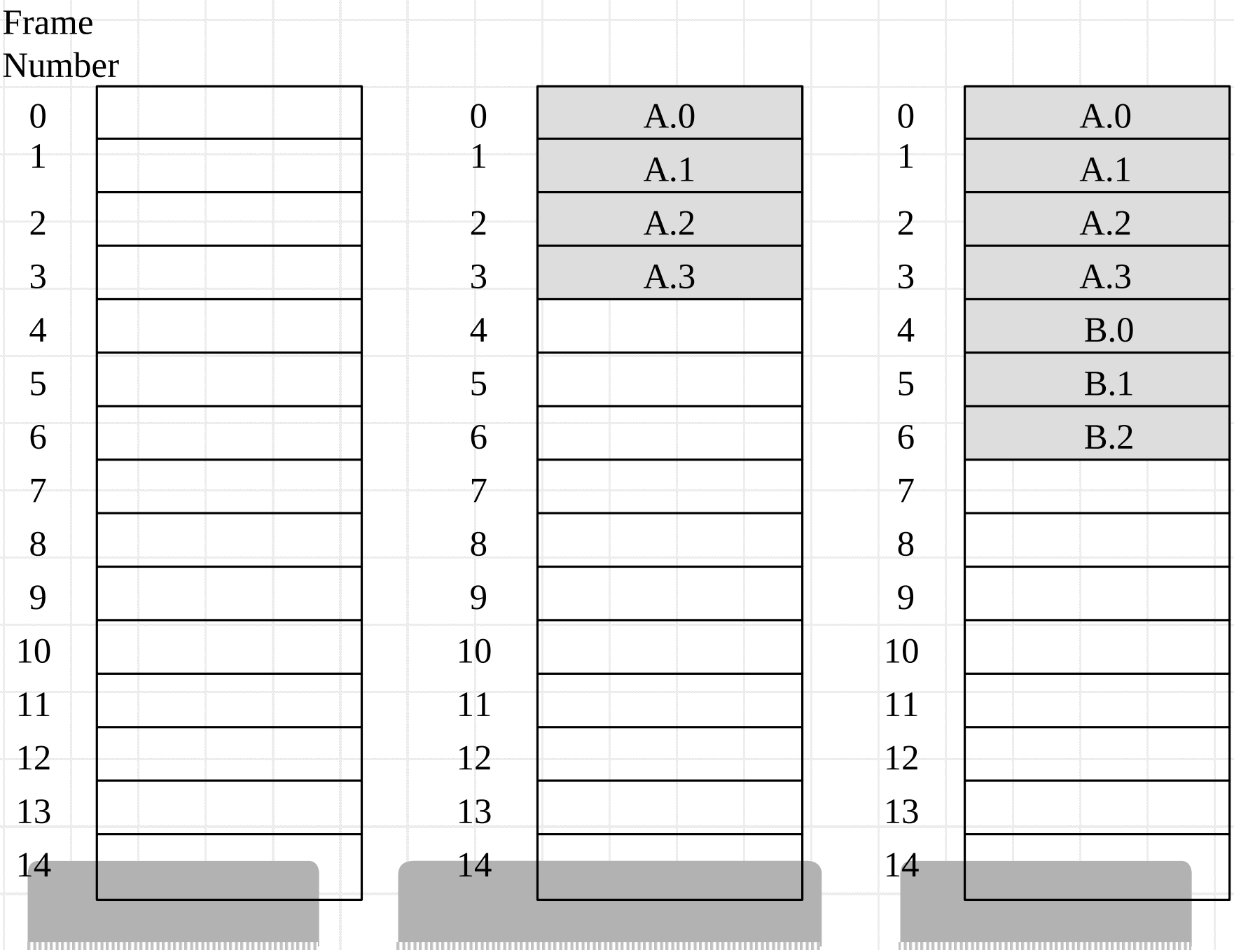
5.4.1 简单页式(simple paging)

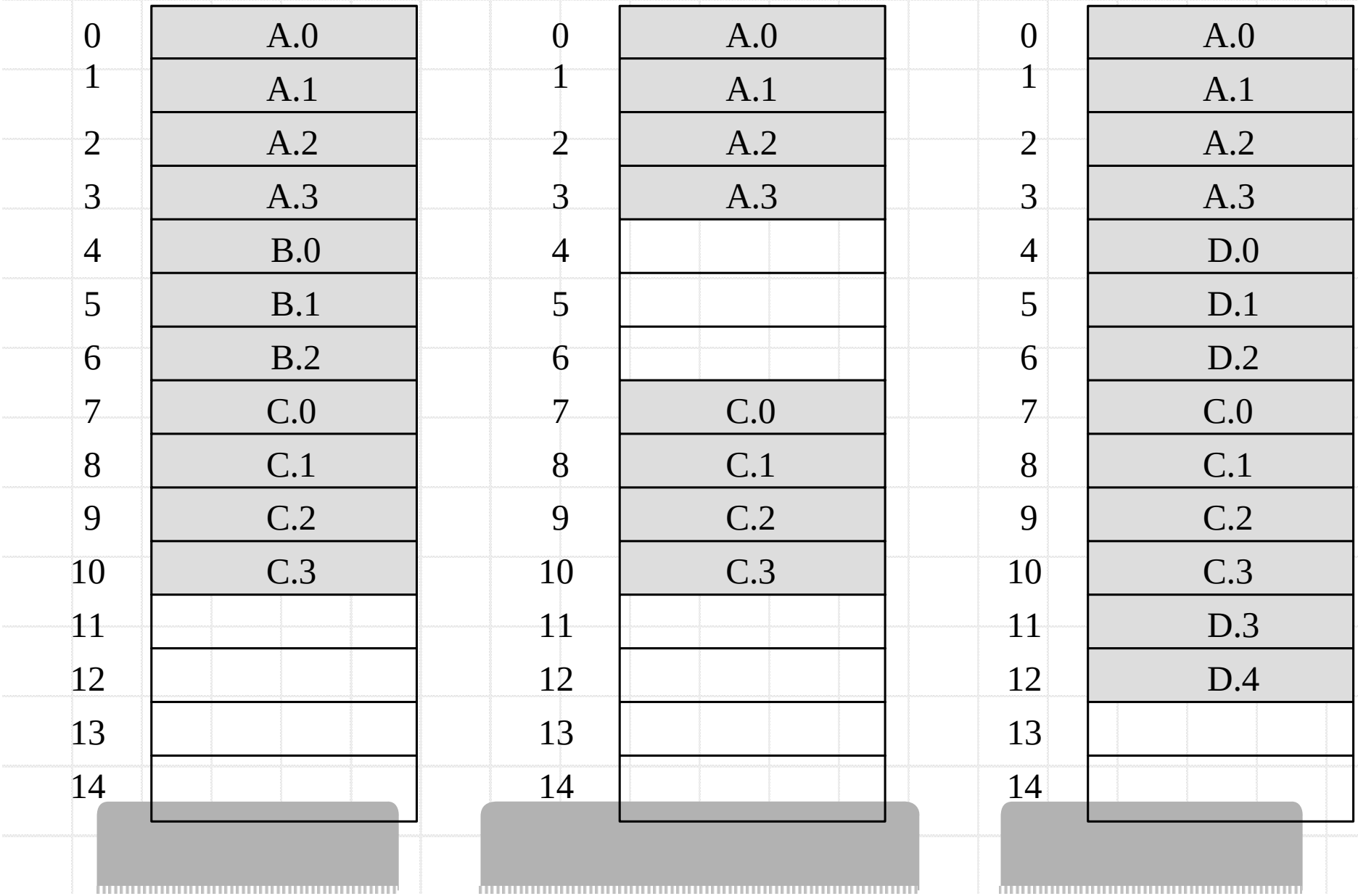
1. 简单页式管理的基本原理

将程序的逻辑地址空间和物理内存划分为固定大小的页或页面(page or page frame)，程序加载时，分配其所需的所有页，这些页不必连续。需要CPU的硬件支持。



返回

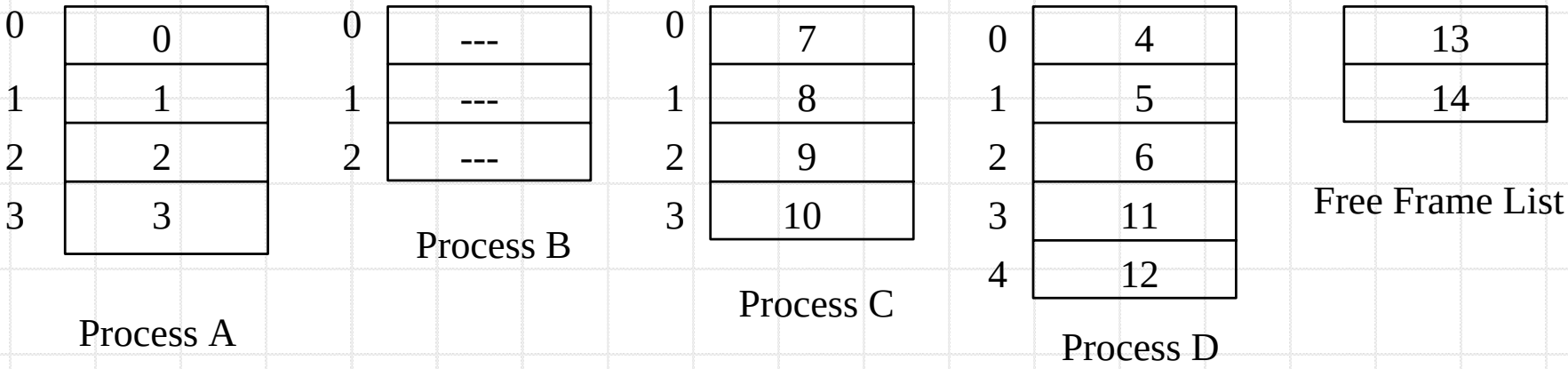

Frame Number
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
0
A.0
1
A.1
2
A.2
3
A.3
4
5
6
7
8
9
10
11
12
13
14
0
A.0
1
A.1
2
A.2
3
A.3
4
B.0
5
B.1
6
B.2
7
8
9
10
11
12
13
14

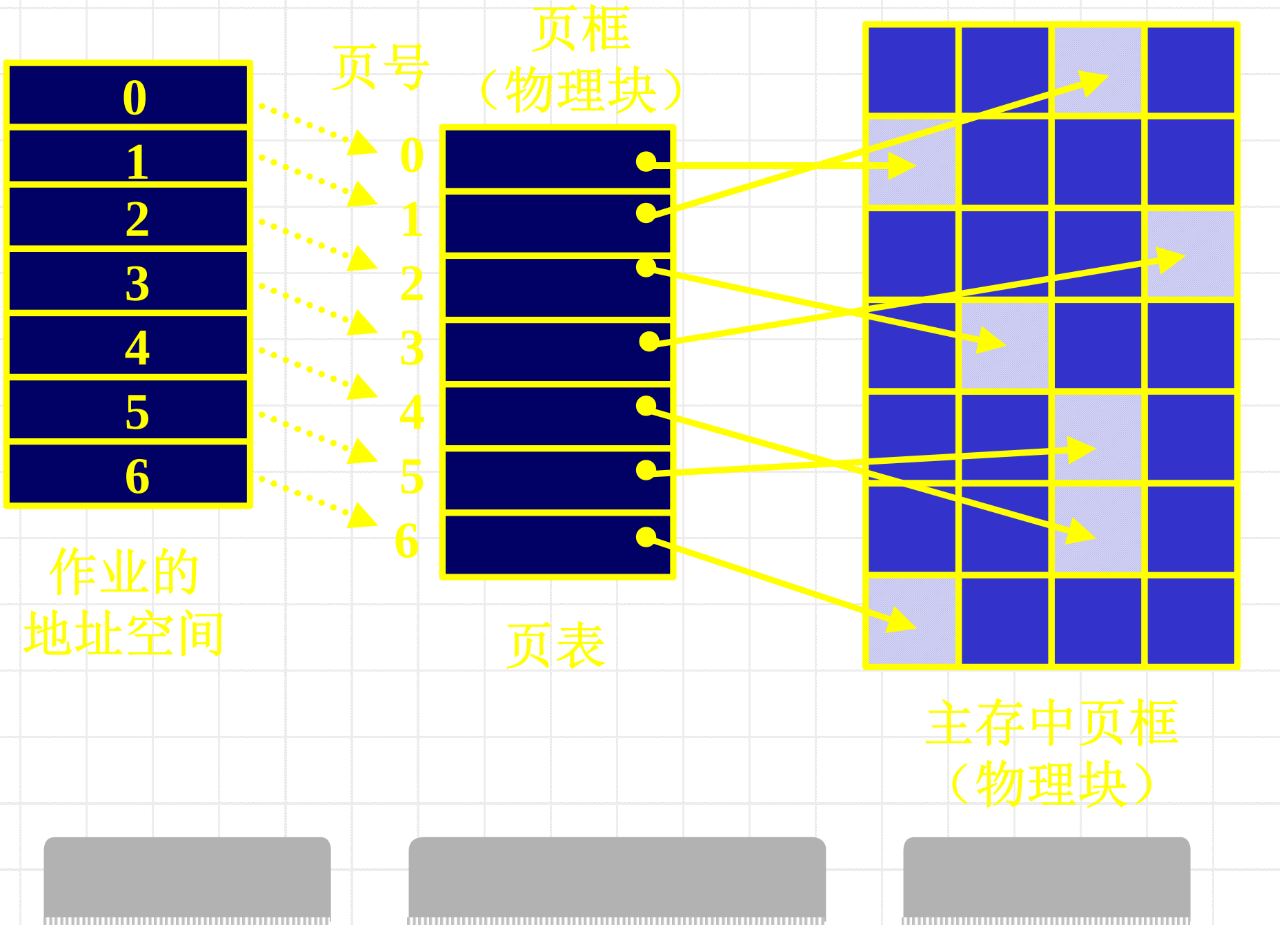

0 A.0
1 A.1
2 A.2
3 A.3
4 B.0
5 B.1
6 B.2
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
0 A.0
1 A.1
2 A.2
3 A.3
4
5
6
7 C.0
8 C.1
9 C.2
10 C.3
11
12
13
14
0 A.0
1 A.1
2 A.2
3 A.3
4 D.0
5 D.1
6 D.2
7 C.0
8 C.1
9 C.2
10 C.3
11 D.3
12 D.4
13
14

- 优点：
  - 没有外碎片，每个内碎片不超过页大小。
  - 一个程序不必连续存放。
  - 便于改变程序占用空间的大小。即随着程序运行而动态生成的数据增多，地址空间可相应增长。
- 缺点：程序全部装入内存。

2. 简单页式管理的数据结构

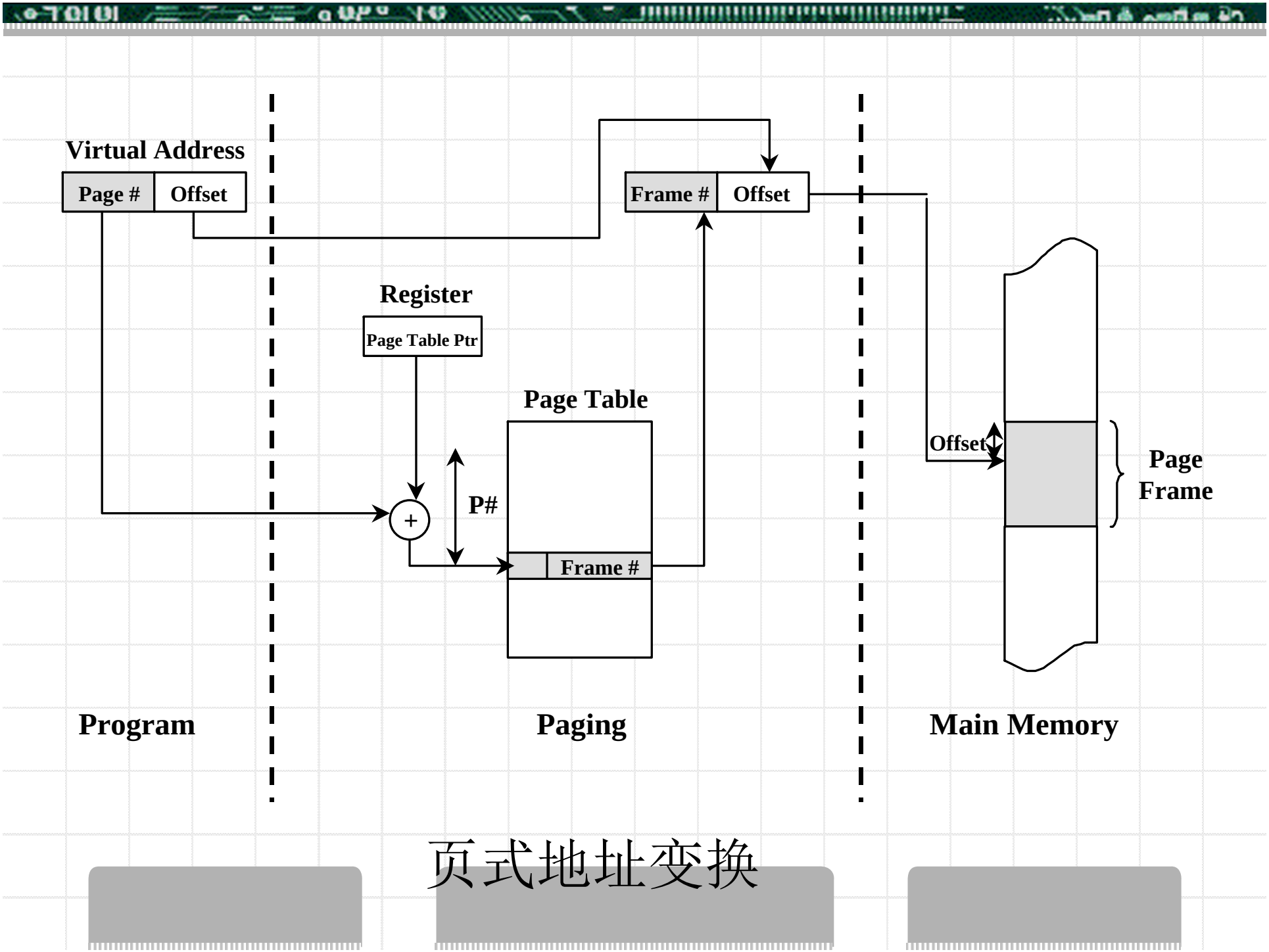
- 进程页表：每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序；
  - 逻辑页号（本进程的地址空间）—>物理页面号（实际内存空间）；
- 物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况。
  - 数据结构：位示图，空闲页面链表；
- 请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换，也可以结合到各进程的PCB里；


0
1
2
3
0
1
2
Process A
0
1
2
Process B
0
1
2
3
Process C
0
1
2
3
4
Process D
13
14
Free Frame List

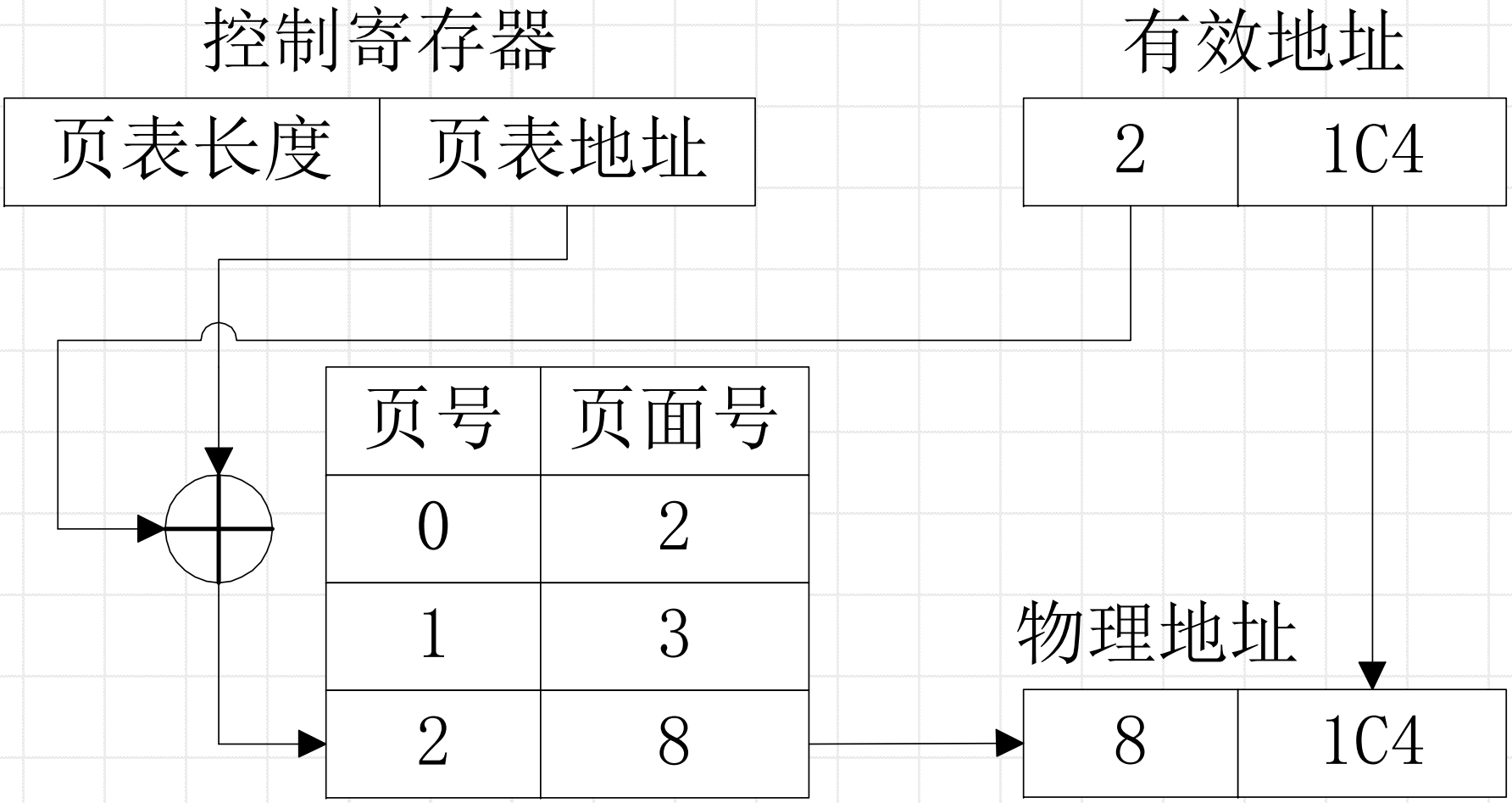

0
1
2
3
4
5
6
作业的
地址空间
页号
0
1
2
3
4
5
6
页框
（物理块）
页表
主存中页框
（物理块）

3. 简单页式管理的地址变换

- 指令所给出地址分为两部分：逻辑页号，页内偏移地址—>查进程页表，得物理页号—>物理地址
  - 为缩短查找时间，可以将页表从内存装入到关联存储器(TLB, Translation Lookaside Buffer)，按内容查找(associative mapping)，即逻辑页号—>物理页号


Virtual Address
Page #
Offset
Frame #
Offset
Register
Page Table Ptr
Page Table
P#
Frame #
Offset
Page
Frame
Program
Paging
Main Memory

页式地址变换



页式地址变换举例

5. 页的大小

- 通常是：几KB到几十KB。
  - 小—>内碎片小；大—>页表短，管理开销小，交换时对外存I/O效率高。
  - 和目前计算机的物理内存大小有关：4MB到256MB，不太大。

5.4.2动态页式管理

- 请求页式管理
  - 执行指令或者访问数据不在内存中
  - 发生缺页中断，从外存调入内存
- 预调入方式
  - 页调入预计算
  - 调入界面排序
  - 按次序调入
- 主要动作
  - 调入与置换

5.4.3 置换算法

- 功能：需要调入页面时，选择内存中哪个物理页面被置换。称为replacement policy。
- 目标：把未来不再使用的或短期内较少使用的页面调出，通常只能在局部性原理指导下依据过去的统计数据进行预测；
- 页面锁定(frame locking)：用于描述必须常驻内存的操作系统的关键部分或时间关键(time-critical)的应用进程。实现方法为在页表中加上锁定标志位(lock bit)。

返回

- 最佳算法(OPT, optimal)
- 最近最久未使用算法(LRU, Least Recently Used)
- 先进先出算法(FIFO)
- 轮转算法(clock)
- 最不常用算法(LFU, Least Frequently Used)
- 页面缓冲算法(page buffering)

1. 最佳算法(OPT, optimal)

选择“未来不再使用的”或“在离当前最远位置上出现的”页面被置换。这是一种理想情况，是实际执行中无法预知的，因而不能实现。可用作性能评价的依据。

2. 最近最久未使用算法 (LRU, Least Recently Used)

选择内存中最久未使用的页面被置换。这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间的先后关系，硬件开销太大。硬件机构如：

- 一个特殊的栈：把被访问的页面移到栈顶，于是栈底的是最久未使用页面。
- 每个页面设立移位寄存器：被访问时左边最高位置1，定期右移并且最高位补0，于是寄存器数值最小的是最久未使用页面。

3. 先进先出算法(FIFO)

选择建立最早的页面被置换。可以通过链表来表示各页的建立时间先后。性能较差。较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出。并且有Belady现象。

- Belady现象：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多，缺页率反而提高的异常现象。
- Belady现象的描述：一个进程P要访问M个页，OS分配N个内存页面给进程P；对一个访问序列S，发生缺页次数为PE（S,N）。当N增大时，PE(S, N)时而增大，时而减小。
- Belady现象的原因：FIFO算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的。

Belady现象的例子

进程P有5页程序访问页的顺序为：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5;

如果在内存中分配3个页面，则缺页情况如下：
12次访问中有缺页9次;

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺 页	x	x	x	x	x	x	x	√	√	x	X	√

如果在内存中分配4个页面，则缺页情况如下：
12次访问中有缺页10次；

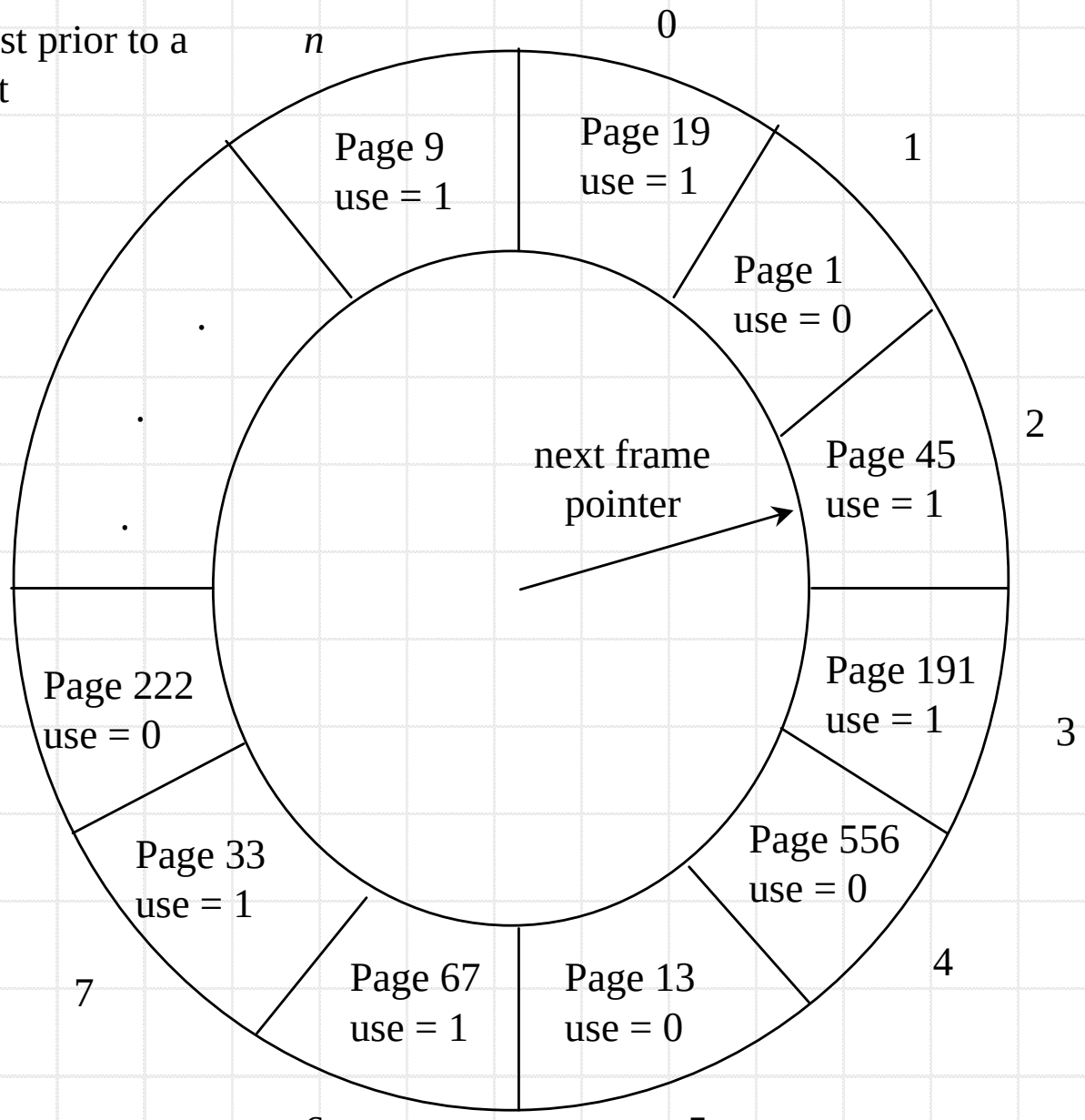
FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺 页	x	x	x	x	√	√	x	x	x	x	x	x

4. 轮转算法(clock)

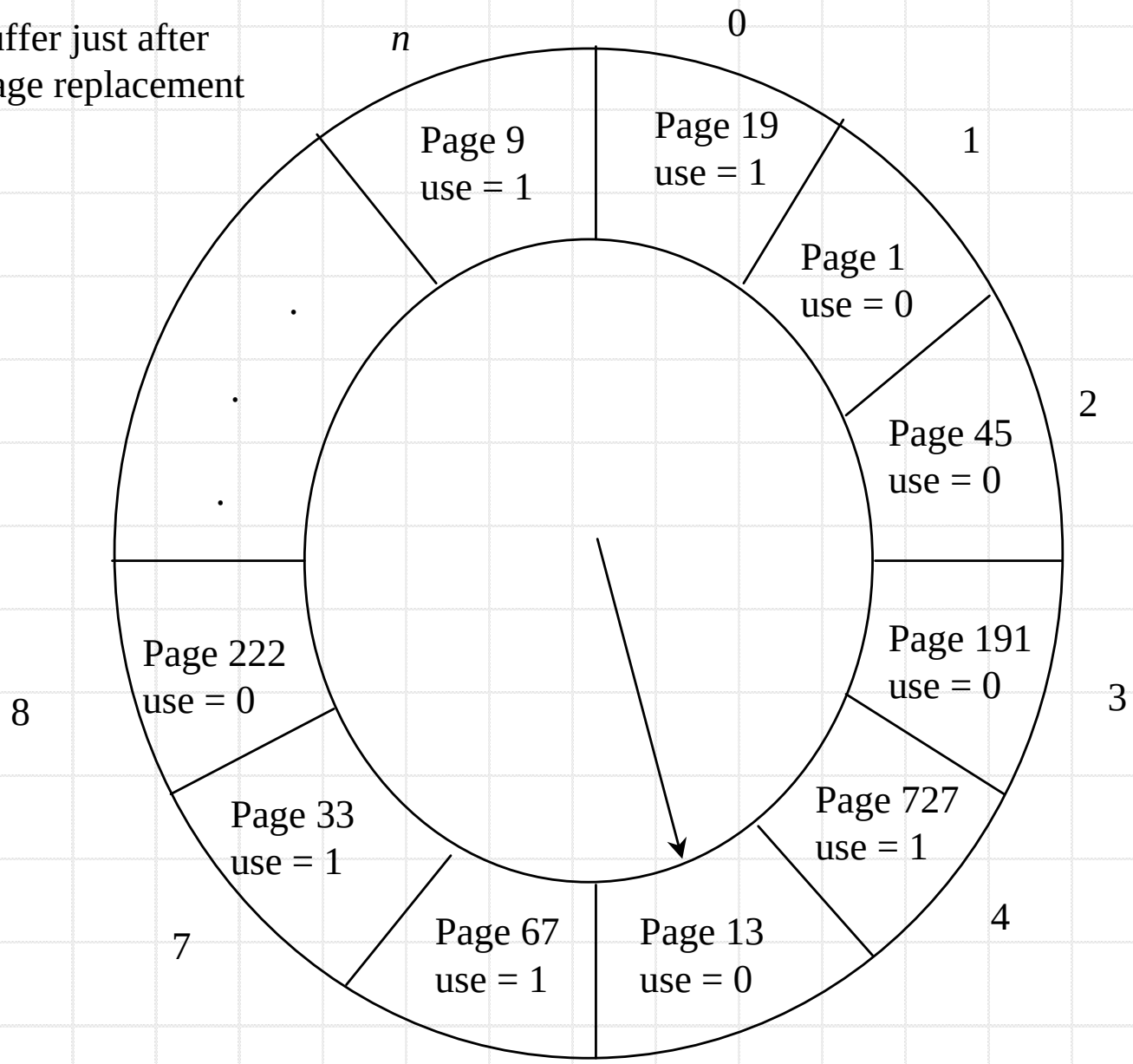
也称最近未使用算法(NRU, Not Recently Used)，它是LRU(最近最久未使用算法)和FIFO的折衷。

- 每页有一个使用标志位(use bit)，若该页被访问则置user bit=1。
- 置换时采用一个指针，从当前指针位置开始按地址先后检查各页，寻找use bit=0的页面作为被置换页。
- 指针经过的use bit=1的页都修改user bit=0，最后指针停留在被置换页的下一个页。

State of buffer just prior to a page replacement



State of buffer just after
the next page replacement



5. 最不常用算法(LFU, Least Frequently Used)

- 选择到当前时间为止被访问次数最少的页面被置换；
- 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1；
- 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零；

5. 页面缓冲算法(page buffering)

它是对FIFO算法的发展，通过被置换页面的缓冲，有机会找回刚被置换的页面；

- 被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。
  - 如果页面未被修改，就将其归入到空闲页面链表的末尾
  - 否则将其归入到已修改页面链表。

- 需要调入新的物理页面时，将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。
- 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，只需较小开销，而被访问的页面可以返还作为进程的内存页。
- 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表，这样能大大减少I/O操作的次数。

7. 置换算法举例

某程序在内存中分配三个页面，初始为空，页面走向为4，3，2，1，4，3，5，4，3，2，1，5。

OPT	4	3	2	1	4	3	5	4	3	2	1	5
页1	4	3	2	1	1	1	5	5	5	2	1	1
页2		4	3	3	3	3	3	3	3	5	5	5
页3			4	4	4	4	4	4	4	4	4	4
		x	x	x	x			x			x	x	

共缺页中断7次

LRU	4	3	2	1	4	3	5	4	3	2	1	5
页1	4	3	2	1	4	3	5	4	3	2	1	5
页2		4	3	2	1	4	3	5	4	3	2	1
页3			4	3	2	1	4	3	5	4	3	2
		x	x	x	x	x	x	x			x	x	x

共缺页中断10次

FIFO	4	3	2	1	4	3	5	4	3	2	1	5
页1	4	3	2	1	4	3	5	5	5	2	1	1
页2		4	3	2	1	4	3	3	3	5	2	2
页3			4	3	2	1	4	4	4	3	5	5
		x	x	x	x	x	x	x			x	x

共缺页中断9次

5.4.4 简单段式(simple segmentation)

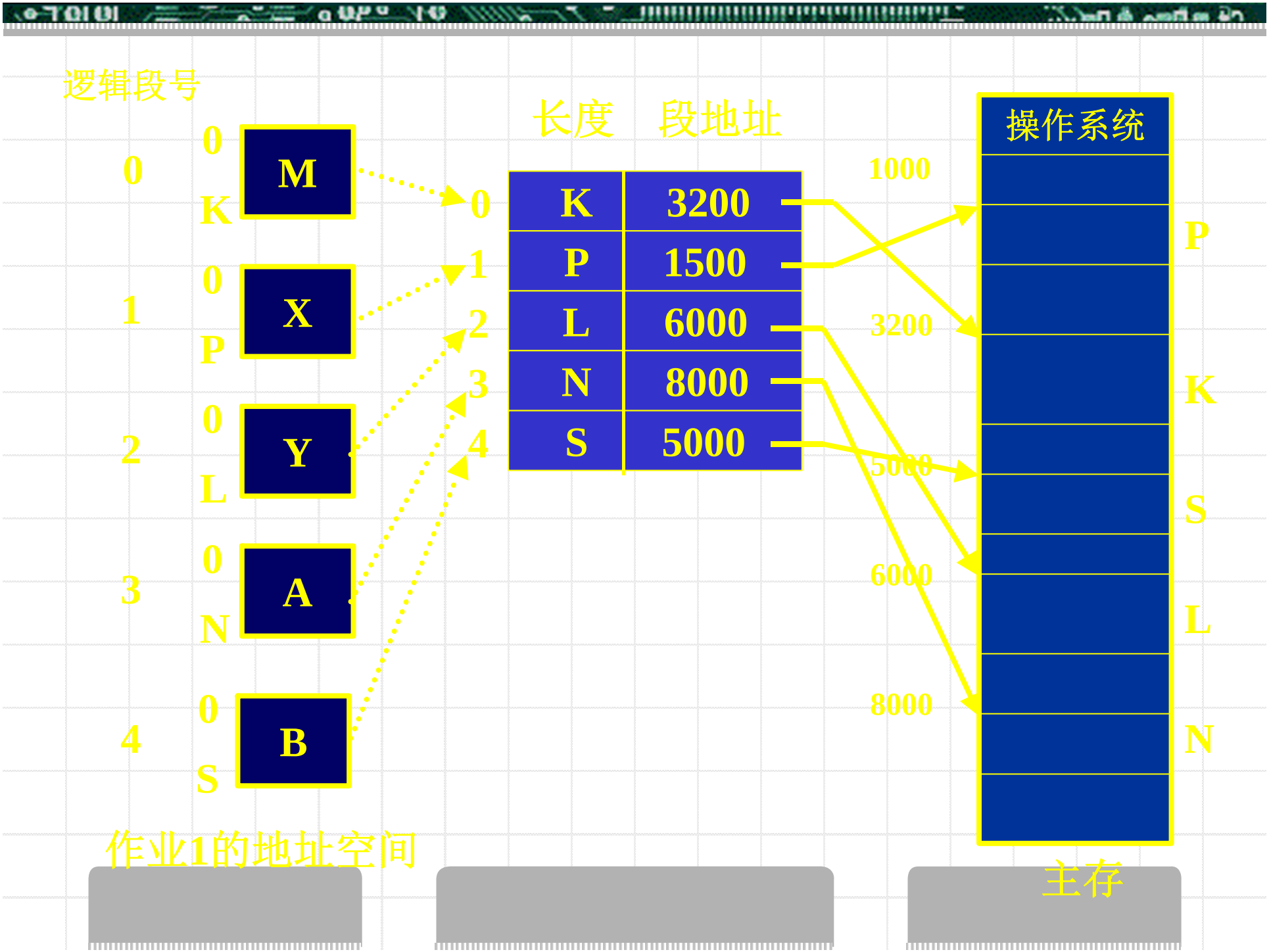
页式管理是把内存视为一维线性空间；而段式管理是把内存视为二维空间，与进程逻辑相一致。

1. 简单段式管理的基本原理

将程序的地址空间划分为若干个段(segment)，程序加载时，分配其所需的所有段（内存分区），这些段不必连续；物理内存的管理采用动态分区。需要CPU的硬件支持。

返回

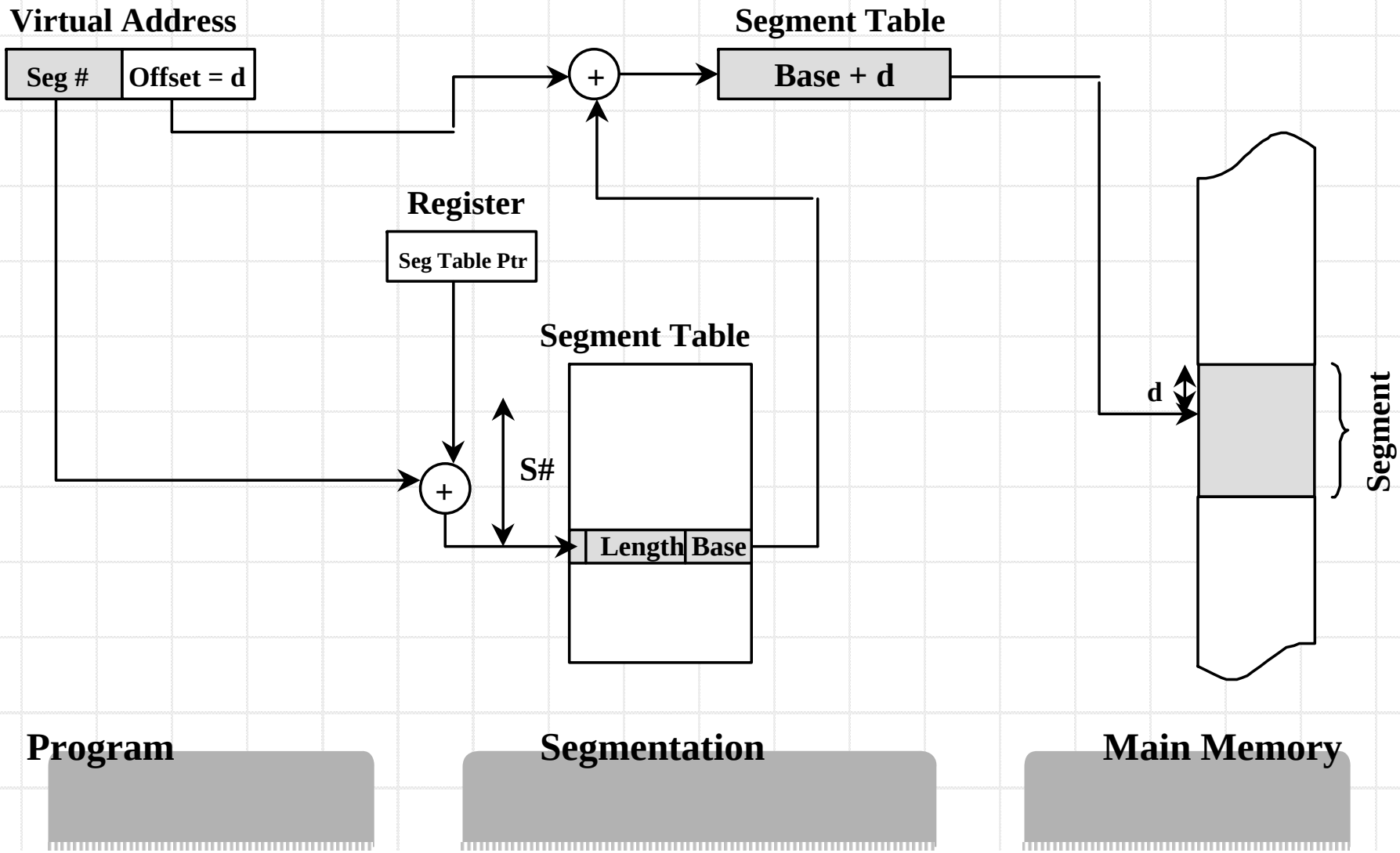
- 程序通过分段(segmentation)划分为多个模块，如代码段、数据段、共享段。
  - 可以分别编写和编译
  - 可以针对不同类型的段采取不同的保护
  - 可以按段为单位来进行共享，包括通过动态链接进行代码共享
- 优点：
  - 没有内碎片，外碎片可以通过内存紧缩来消除。
  - 便于改变进程占用空间的大小。
- 缺点：
  - 进程全部装入内存。

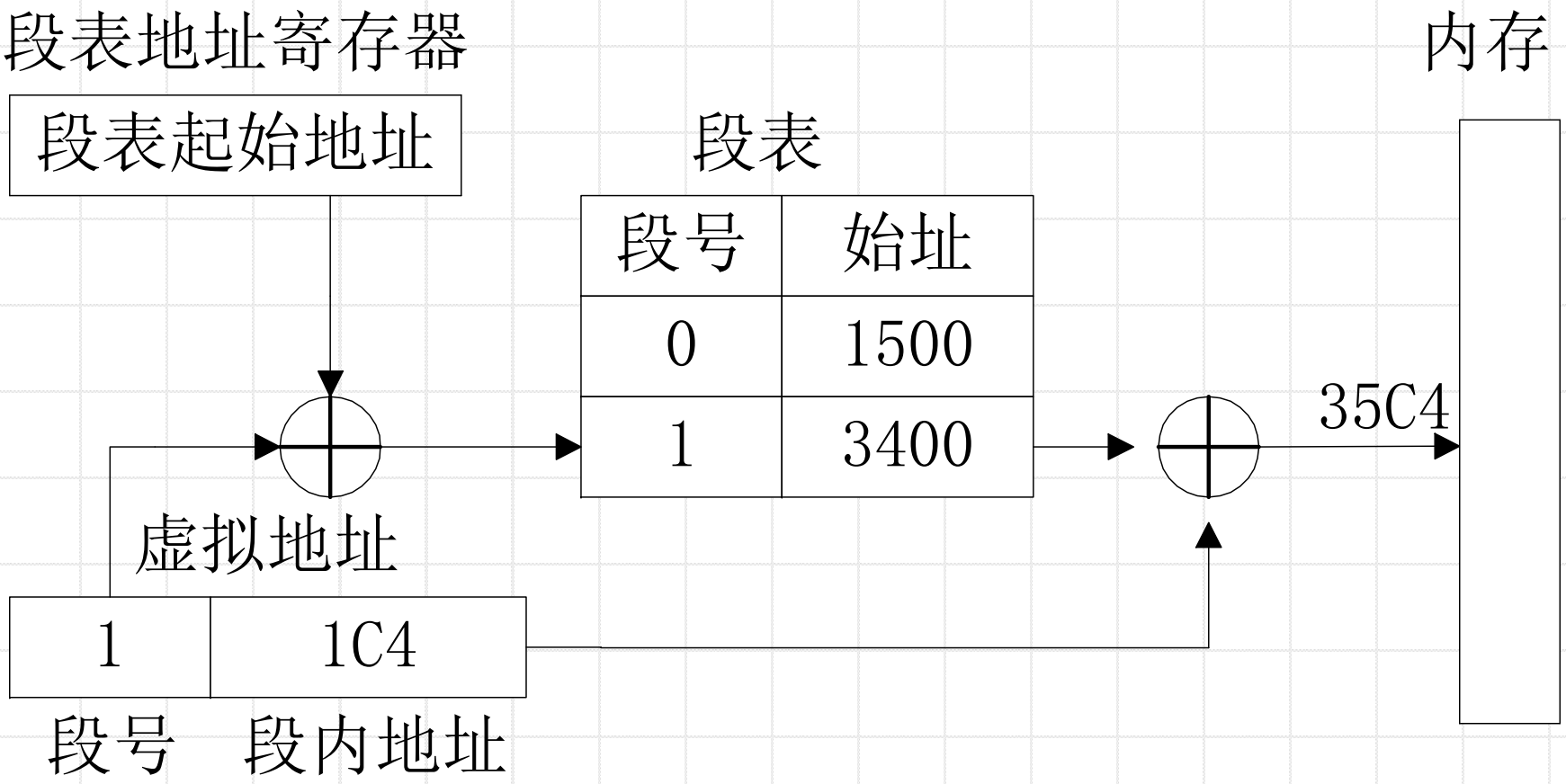

逻辑段号
0
K
M
1
P
X
2
L
Y
3
N
A
4
S
B
作业1的地址空间
长度
段地址
0
1
2
3
4
K
3200
P
1500
L
6000
N
8000
S
5000
操作系统
1000
3200
5000
6000
8000
P
K
S
L
N
主存

2. 简单段式管理的数据结构

- 进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(base address)和段长度
- 系统段表：系统内所有占用段
- 空闲段表：内存中所有空闲段，可以结合到系统段表中

3. 简单段式管理的地址变换





段式地址变换举例

5.4.5 页式管理和段式管理的比较

- 分页是出于系统管理的需要，分段是出于用户应用的需要。
  - 一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。
- 页大小是系统固定的，而段大小则通常不固定。
- 逻辑地址表示：
  - 分页是一维的，各个模块在链接时必须组织成同一个地址空间；
  - 分段是二维的，各个模块在链接时可以每个段组织成一个地址空间。
- 通常段比页大，因而段表比页表短，可以缩短查找时间，提高访问速度。

返回

5.4.5 局部性原理

- 局部性原理(principle of locality)：指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性：一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - 空间局部性：当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。

返回

- 局部性原理的具体体现
  - 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。
  - 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。
  - 程序中存在相当多的循环结构，它们由少量指令组成，而被多次执行。
  - 程序中存在相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。

第六章 文件系统

信息是计算机系统中的重要资源。操作系统中的一个重要组成部分，文件系统，就负责信息的组织、存储和访问。

文件系统的功能就是提供高效、快速和方便的信息存储和访问功能。本章的主要内容就是信息的组织。

6.1 引言

返回

6.1.1 文件管理的目的

- 方便的文件访问和控制：以符号名称作为文件标识，便于用户使用；
- 并发文件访问和控制：在多道程系统中支持对文件的并发访问和控制；
- 统一的用户接口：在不同设备上提供同样的接口，方便用户操作和编程；
- 多种文件访问权限：在多用户系统中的不同用户对同一文件会有不同的访问权限；
- 优化性能：存储效率、检索性能、读写性能；
- 差错恢复：能够验证文件的正确性，并具有一定的差错恢复能力；

返回

6.1.2 文件系统的基本概念

文件系统是操作系统中管理文件的机构，提供文件存储和访问功能。FAT, FAT32, NTFS, NFS

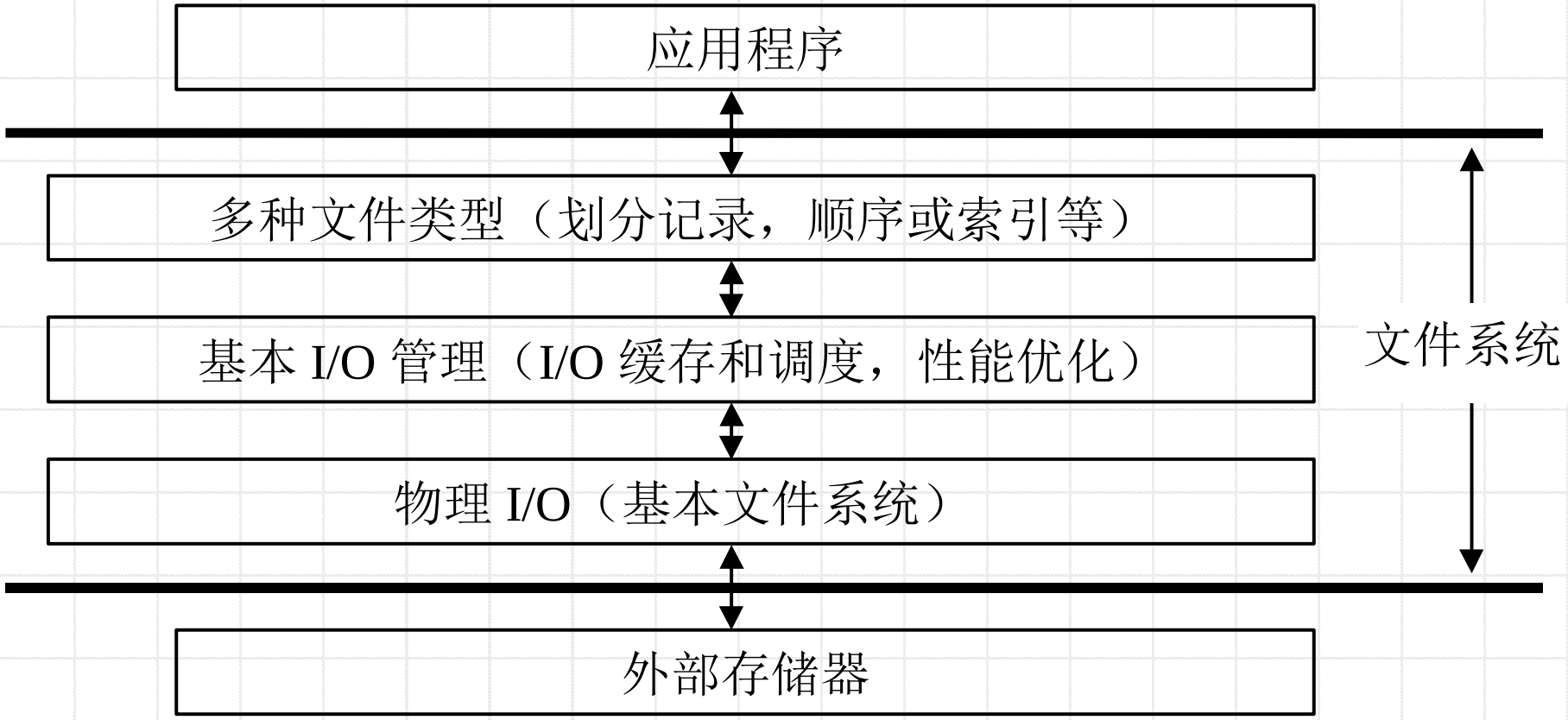
目录是由文件说明索引组成的用于文件检索的特殊文件。

文件是具有符号名的数据项的集合。文件名是文件的标识符号。文件包括两部分：

- 文件体：文件本身的信息；
- 文件说明：文件存储和管理信息；如：文件名、文件内部标识、文件存储地址、访问权限、访问时间等；

返回

6.1.3 文件系统的结构和功能元素



返回

2. 文件管理的服务功能元素

(文件系统向上层用户提供的服务)

- 文件访问：文件的创建、打开和关闭，文件的读写；
- 目录管理：用于文件访问和控制的信息，不包括文件内容
- 文件结构管理：划分记录，顺序，索引
- 访问控制：并发访问和用户权限
- 限额(quota)：限制每个用户能够建立的文件数目、占用外存空间大小等
- 审计(auditing)：记录对指定文件的使用信息（如访问时间和用户等），保存在日志中

3. 文件系统的实现功能元素

(文件系统要实现的功能模块)

- 文件的分块存储：与外存的存储块相配合
- I/O缓冲和调度：性能优化
- 文件定位：在外存上查找文件的各个存储块
- 外存存储空间管理：如分配和释放。主要针对可改写的外存如磁盘。
- 外存设备访问和控制：包括由设备驱动程序支持的各种基本文件系统如硬盘，软盘，CD ROM等

6.2 文件的组织(file organization)

文件组织讨论文件的内部逻辑结构，主要考虑因素是文件存储性能和访问性能。

返回

6.2.1文件的组织

文件的组织是指从用户观点出发讨论文件内部的逻辑结构(logical structure)或用户访问模式；它可以独立于在外存上的物理存储。

- 文件逻辑结构的设计要求：
  - 访问性能：便于检索；便于修改
  - 存储性能：向物理存储转换方便，节省空间
- 文件的不同组织层次：域、记录、文件

返回

6.2.2 文件的组织类型

1. 无结构文件

文件体为字节流，不划分记录，顺序访问，每次读写访问可以指定任意数据长度。当前操作系统中常用的文件组织。

2. 累积文件(pile)

文件体为无结构记录序列，通过特定分隔符来划分记录，各记录大小和组成可变。新记录总是添加到文件末尾。如日志log，或电子邮件的邮箱文件(mailbox)。检索必须从头开始。

返回

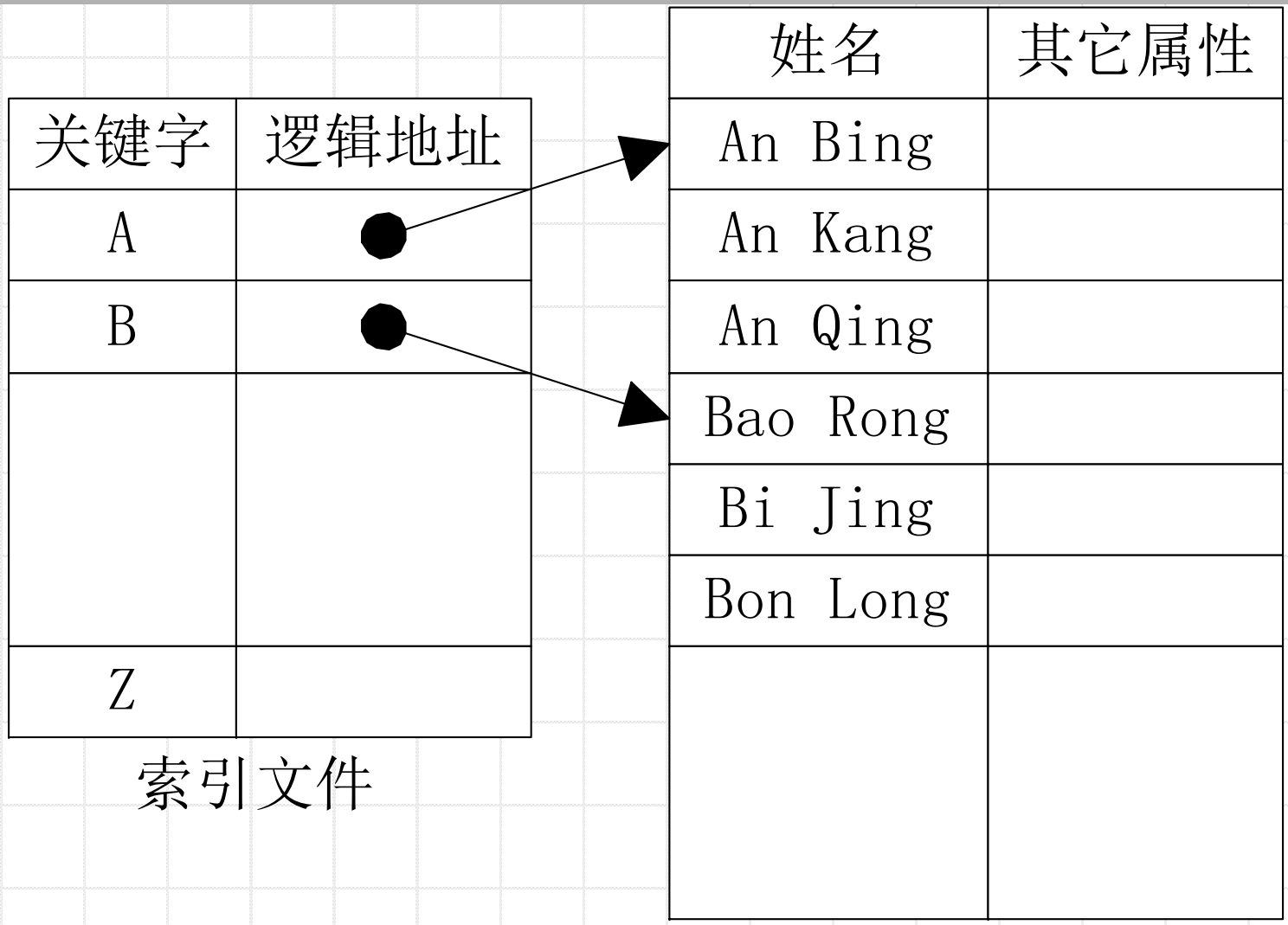
3. 顺序文件(sequential file)

文件体为大小相同的排序记录序列。它由一个主文件和一个临时文件组成。记录大小相同，按某个关键字域(key field)排序，存放在主文件(master file)中。新记录暂时保存在日志或事务文件(log file or transaction file)中，定期归并入主文件。

4. 索引顺序文件(indexed-sequential file)

在顺序文件（主文件main file）的基础上，另外建立索引(index)和溢出文件(overflow file)。这样做的目的是加快顺序文件的检索速度。

- 在索引文件中，可将关键字域中的取值划分若干个区间（如A~Z可以划分为A到Z共26个区间），每个区间对应一个索引项，后者指向该区间的开头记录。新记录暂时保存在溢出文件中，定期归并入主文件。
- 通过划分层次，在记录数量较大时，比顺序文件大大缩短检索时间。顺序文件是N/2(这时可使用折半查找)，而索引顺序文件（一级索引）是i/2 + N/(2*i)，其中i为索引长度。索引还可以是多级的。如：有1000,000条记录的顺序文件的平均检索长度为500,000，而在添加一个有1000条索引项的索引文件后，平均检索长度为1000。


关键字
逻辑地址
A
B
Z
索引文件
姓名
其它属性
An Bing
An Kang
An Qing
Bao Rong
Bi Jing
Bon Long
顺序文件

索引顺序文件

5. 索引文件(indexed file)

记录大小不必相同，不必排序，存放在主文件(primary file)中。索引文件与索引顺序文件的区别在于主文件不排序。另外建立索引，每个索引项指向一个记录，索引项按照记录中的某个关键字域排序。对同一主文件，可以针对不同的关键字域相应建立多个索引。索引文件的记录项通常较小，查找速度快，便于随机访问(random access)。

6. 哈希文件或直接文件(hashed file or direct file)

记录大小相同。由主文件和溢出文件组成。记录位置由哈希函数确定。检索时给出记录编号，通过哈希函数计算出该记录在文件中的相对位置。访问速度快，但在主文件中有空闲空间。

6.3 文件目录

目录是由文件说明索引组成的用于文件检索的特殊文件。文件目录的内容主要是文件访问的控制信息（不包括文件内容）。

返回

6.3.1 目录内容

目录的内容是文件属性信息(properties)，其中的一部分是用户可获取的。

1. 基本信息

- 文件名：字符串，通常在不同系统中允许不同的最大长度。可以修改。有些系统允许同一个文件有多个别名(alias)；
- 别名的数目；
- 文件类型：可有多种不同的划分方法，如：
  - 有无结构（记录文件，流式文件）
  - 内容（二进制，文本）
  - 用途（源代码，目标代码，可执行文件，数据）
  - 属性attribute（如系统，隐含等）
  - 文件组织（如顺序，索引等）

返回

2. 地址信息

- 存放位置：包括哪个设备或文件卷volume，以及各个存储块位置；
- 文件长度（当前和上限）：以字节、字或存储块为单位。可以通过写入或创建、打开、关闭等操作而变化。

3. 访问控制信息

文件所有者（属主）：通常是创建文件的用户，或者改变已有文件的属主；

访问权限（控制各用户可使用的访问方式）：如读、写、执行、删除等；

4. 使用信息

- 创建时间
- 最后一次读访问的时间和用户
- 最后一次写访问的时间和用户

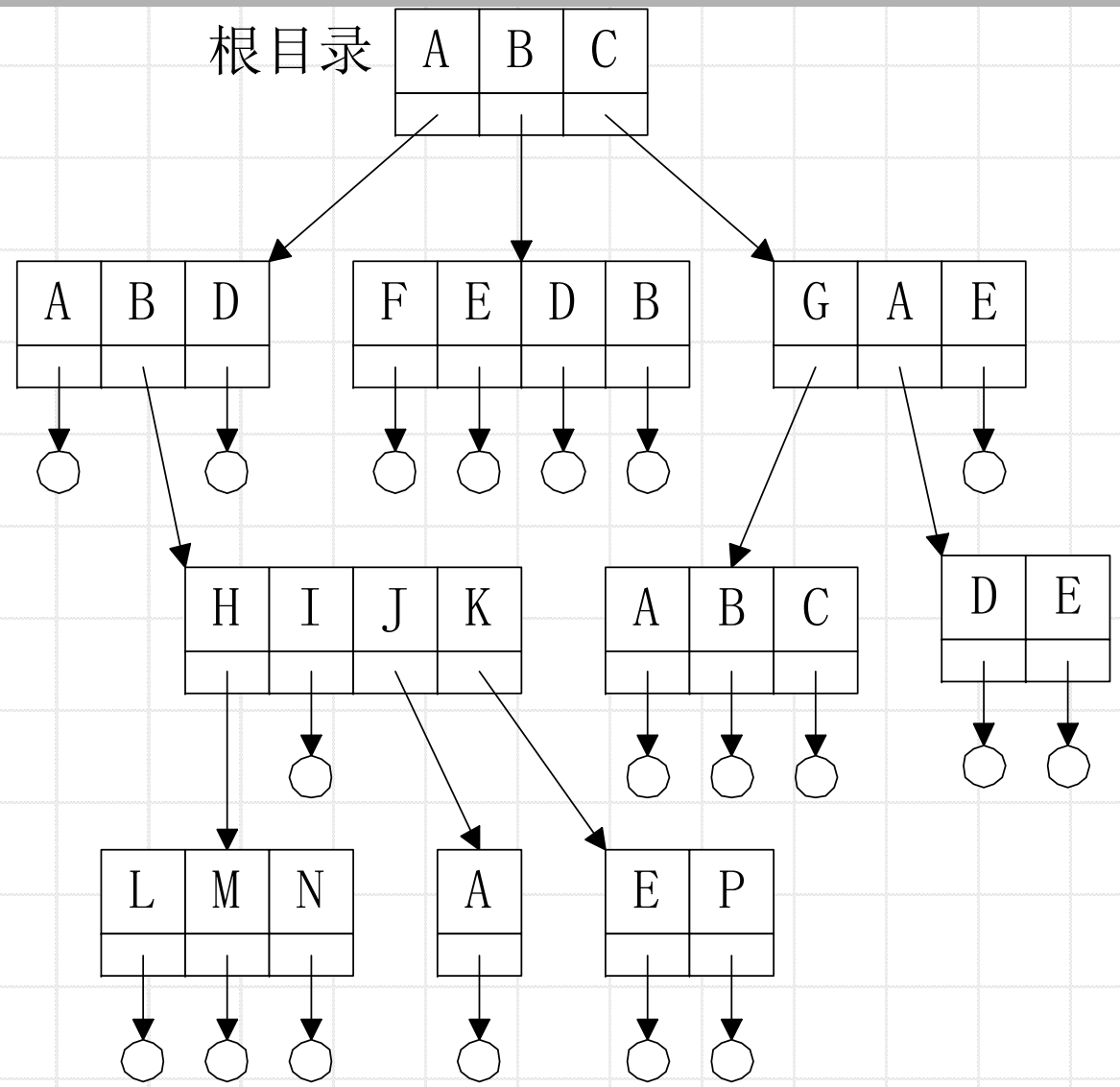
6.3.2 目录结构类型

目录结构讨论目录的组织结构，设计目标是检索效率。

- 一级目录：整个目录组织是一个线性结构，系统中的所有文件都建立在一张目录表中。它主要用于单用户操作系统。它具有如下的特点：
  - 结构简单；
  - 文件多时，目录检索时间长；
  - 有命名冲突：如重名(多个文件有相同的文件名) 或别名(一个文件有多个不同的文件名)
- 二级目录：在根目录下，每个用户对应一个目录（第二级目录）；在用户目录下是该用户的文件，而不再有下级目录。适用于多用户系统，各用户可有自己的专用目录。

返回

- 多级目录：或称为树状目录(tree-like)。在文件数目较多时，便于系统和用户将文件分散管理。适用于较大的文件系统管理。目录级别太多时，会增加路径检索时间。
  - 目录名：可以修改。
  - 目录树：中间结点是目录，叶子结点是目录或文件。
  - 目录的上下级关系：当前目录(current directory, working directory)、父目录(parent directory)、子目录(subdirectory)、根目录(root directory)等；
  - 路径(path)：每个目录或文件，可以由根目录开始依次经由的各级目录名，加上最终的目录名或文件名来表示；



多级目录组织

6.4 文件和目录的使用

这一部分讨论操作系统提供的与文件系统相关的API。

返回

6.4.1 文件访问

文件访问是指围绕文件内容读写进行的文件操作。

- 打开open：为文件读写所进行的准备。给出文件路径，获得文件句柄(file handle)，或文件描述符(file descriptor)。需将该文件的目录项读入到内存中。
- 关闭close：释放文件描述符，把该文件在内存缓冲区的内容更新到外存上。
- 复制文件句柄dup：用于子进程与父进程间的文件共享，复制前后的文件句柄有相同的文件名、文件指针和访问权限；

返回

- 读read、写write和移动文件读写指针lseek：系统为每个打开文件维护一个读写指针(read-write pointer)，它是相对于文件开头的偏移地址(offset)。读写指针指向每次文件读写的开始位置，在每次读写完成后，读写指针按照读写的数据量自动后移相应数值。
- 执行exec：执行一个可执行文件；
- 修改文件的访问模式（fcntl和ioctl）：提供对打开文件的控制，如：文件句柄复制、读写文件句柄标志、读写文件状态标志、文件锁定控制、流（stream）的控制；

6.4.2 文件控制

文件控制是指围绕文件属性控制进行的文件操作。

- 创建（creat和open）：给出文件路径，获得新文件的文件句柄；
- 删除unlink：对于symbolic link和hard link，删除效果是不同的；
- 获取文件属性（stat和fstat）：stat的参数为文件名，fstat的参数为文件句柄；
- 修改文件名rename；
- 修改文件属主chown，修改访问权限chmod：与相应系统命令类似；
- 文件别名控制：创建symlink或link，读取链接路径readlink；

返回

6.4.3 目录管理

目录管理是指目录访问和目录属性控制。

- 进行文件访问和控制时，由操作系统自动更新目录内容
- 目录创建mkdir，删除rmdir，修改目录名rename。只适用于超级用户：mknod（建立文件目录项）和unlink（删除目录项）
- 修改当前目录chdir；

返回

6.4.4 伪文件(pseudo file)

伪文件是指具有文件某些特征的系统资源或设备，它们的访问和控制方式与文件类似。

- 特点
  - 内容并不保存在外存上，而是在其他外部设备上或内存里
  - 随文件类型的不同，适用于某些文件访问和控制的系统调用，如：open, read, write, close, chmod, chown。
  - 创建时使用特定的系统调用，如：创建管道pipe，创建管套socket，创建设备文件mknod
- 类型
  - 设备：字符设备或块设备，可以直接访问设备中的字节数据或数据块。如终端、硬盘、内存等。在UNIX中称为特殊文件(special file)。
  - 进程间通信：本计算机或通过网络。如：管道，管套等。

返回

6.5 外存存储空间管理

讨论如何高效地进行数据存储

6.5.1 文件存储空间分配(file allocation)

6.5.2 外存空闲空间管理方法(free space management)

6.5.3 文件卷

6.5.1 文件存储空间分配(file allocation)

1. 新创建文件的存储空间（文件长度）分配方法

- 预分配(preallocation)：创建时(这时已知文件长度)一次分配指定的存储空间，如文件复制时的目标文件。
- 动态分配(dynamic allocation)：需要存储空间时才分配（创建时无法确定文件长度），如写入数据到文件。

返回

2. 文件存储单位：簇（cluster）

文件的存储空间通常由多个分立的簇组成，而每个簇包含若干个连续的扇区(sector)。

- 簇的大小
  - 两个极端：大到能容纳整个文件，小到一个外存存储块；
  - 簇较大：提高I/O访问性能，减小管理开销；但簇内碎片浪费问题较严重；
  - 簇较小：簇内的碎片浪费较小，特别是大量小文件时有利；但存在簇编号空间不够的问题（如FAT12、16、32）；

- 簇的分配方法：两种
  - 簇大小可变，其上限较大：I/O访问性能较好，文件存储空间的管理困难（类似于动态分区存储管理）
  - 簇大小固定，较小：文件存储空间使用灵活，但I/O访问性能下降，文件管理所需空间开销较大
- 文件卷容量与簇大小的关系
  - 文件卷容量越大，若簇的总数保持不变即簇编号所需位数保持不变，则簇越大。缺点：簇内碎片浪费越多
  - 文件卷容量越大，若簇大小不变，则簇总数越多，相应簇编号所需位数越多。如簇编号长度为12、16、32二进制位，即构成FAT12、FAT16、FAT32。

3. 文件存储分配数据结构

采用怎样的数据结构来记录一个文件的各个部分的位置。

- 连续分配(contiguous)：只需记录第一个簇的位置，适用于预分配方法。可以通过紧缩(compact)将外存空闲空间合并成连续的区域。
- 链式分配(chained)：在每个簇中有指向下一个簇的指针。可以通过合并(consolidation)将一个文件的各个簇连续存放，以提高I/O访问性能。
- 索引分配(indexed)：文件的第一个簇中记录了该文件的其他簇的位置。可以每处存放一个簇或连续多个簇（只需在索引中记录连续簇的数目）。

6.5.2 外存空闲空间管理(free space management)方法

外存空闲空间管理的数据结构通常称为磁盘分配表(disk allocation table)，分配的基本单位是簇。文件系统可靠性包括检错和差错恢复。空闲空间的管理方法：三种，均适用于上述几种文件存储分配数据结构；

- 位示图(bitmap)：每一位表示一个簇，取值0和1分别表示空闲和占用。
- 空闲空间链接(chained free space)：每个空闲簇中有指向下一个空闲簇的指针，所有空闲簇构成一个链表。不需要磁盘分配表，节省空间。每次申请空闲簇只需取出链表开头的空闲簇即可。
- 空闲空间索引(indexed free space)：在一个空闲簇中记录其他几个空闲簇的位置。

注：可以上述方法结合，应用于不同的场合。如：位示图应用于索引结点表格，链接和索引结合应用于文件区的空闲空间。

返回

6.5.3 文件卷

- 磁盘分区(partition)：通常把一个物理磁盘的存储空间划分为几个相互独立的部分，称为"分区"。一个分区的参数包括：磁盘参数（如每道扇区数和磁头数），分区的起始和结束柱面等。
- 文件卷(volume)：或称为"逻辑驱动器(logical drive)"。在同一个文件卷中使用同一份管理数据进行文件分配和外存空闲空间管理，而在不同的文件卷中使用相互独立的管理数据。
  - 一个文件不能分散存放在多个文件卷中，其最大长度不超过所在文件卷的容量。
  - 通常一个文件卷只能存放在一个物理外设上（并不绝对），如一个磁盘分区或一盘磁带。

返回

- 格式化(format)：在一个文件卷上建立文件系统，即：
  - 建立并初始化用于进行文件分配和外存空闲空间管理的管理数据。
  - 通常，进行格式化操作使得一个文件卷上原有的文件都被删除。
- 扩展文件卷集(extended volume set)：一个文件卷由一个或几个磁盘上的多个磁盘分区依次连接组成。可以容纳长度大于磁盘分区容量的文件。
  - 实例：Windows NT中的扩展文件卷集。

- 磁盘交叉存储(disk interleaving)：将一个文件卷的存储块依次分散在多个磁盘上。如4个磁盘，则磁盘0上是文件卷块0, 4, 8, …，磁盘1上是文件卷块1, 5, 9, …。
  - 优点：提高I/O效率。如果需要访问一个文件的多个存储块，而它们分散在多个磁盘上，则可以并发地向多个磁盘发出请求，并可在此基础上提供文件系统的容错功能。关键：磁盘访问时间大部分由旋转等待时间组成。
  - 需要相应硬件设备：如多个硬盘连接在同一个或不同的SCSI接口上，或者两个硬盘连接在一个或不同的IDE接口上（两个硬盘连接在同一个IDE接口上，不能提高I/O效率）
  - 实例：Windows NT中的条带卷(stripe set)，每个文件卷块的大小是64KB。
  - 类似例子：在虚拟存储器中建立多个交换区，分散在多个磁盘上

6.6 文件系统举例

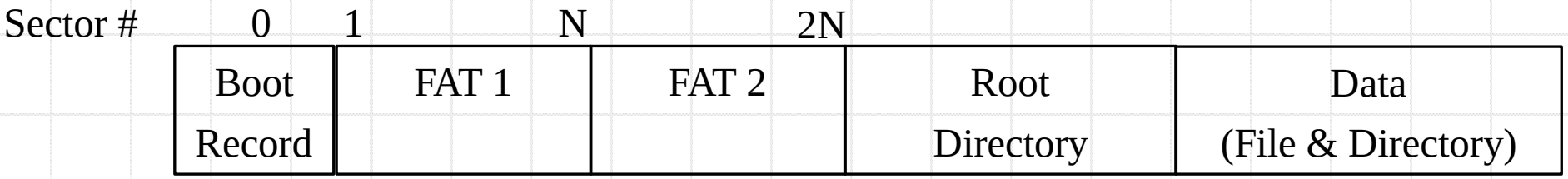
6.6.1 MS DOS的文件系统

返回

6.6.1 MS DOS的文件系统

多级目录，不支持文件别名，无用户访问权限控制

1. 磁盘文件卷结构



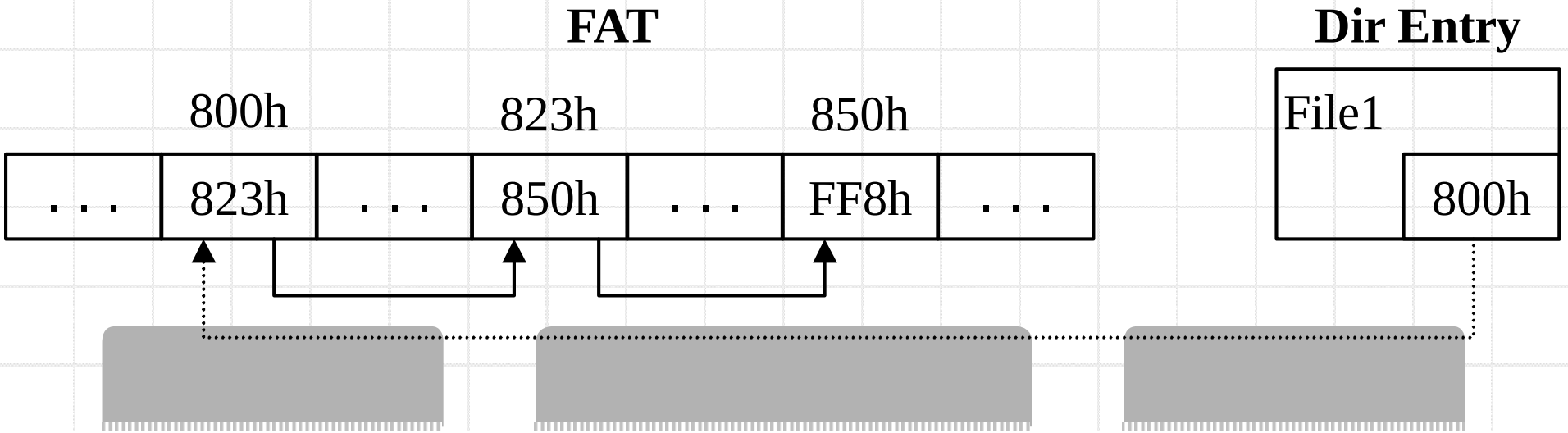
Volume Structure in MS DOS

返回

- 文件卷(volume)信息：记录在引导记录的扇区中。包括：簇大小，根目录项数目，FAT表大小，磁盘参数（每道扇区数，磁头数），文件卷中的扇区总数，簇编号长度等
  - 逻辑扇区号：三元组（柱面号，磁头号，扇区号）—>一个文件卷中从0开始对每个扇区编号，优点：屏蔽了物理磁盘参数的不同
  - 允许同时访问的文件卷数目上限可以由config.sys文件中的LASTDRIVE= 语句指定
  - 簇(cluster)：由若干个扇区组成。在一个文件卷中从0开始对每个簇编号。

- FAT表：两个镜像，互为备份。文件卷中的每个簇均对应一个FAT表项，文件分配采用链式分配方法。
  - 每个FAT表项所占位数是簇编号的位数，其值是（以FAT12为例）：
    - 0：表示该簇空闲
    - FF7h：物理坏扇区
    - FF8h~FFFh：表示该簇是文件的最后一个簇
    - 其他值：表示该簇被文件占用，而且表项中的值是文件下一个簇的编号。
  - FAT表大小占文件卷容量的比例：
    - 簇编号位数/ (8*512*每个簇的扇区数)

- 目录：是目录项的顺序文件(即大小相同的排序记录序列)，不对目录项排序。
  - 若目录中包含的文件数目较多，则搜索效率低。
  - 每个目录项大小为32字节，其内容包括：文件名（8+3个字符），属性（包括文件、子目录和文件卷标识），最后一次修改时间和日期，文件长度，第一个簇的编号。
  - 在目录项中，若第一个字节为 E5h，则表示空目录项；若为05h，则表示文件名的第一个字符为 E5h。
  - 文件名不区分大小写



2. 打开文件管理

- 系统文件表(SFT, System File Table)和任务文件表(JFT, Job File Table)：
  - SFT包含系统的所有打开文件，可以由几个表项依次连接组成。
  - JFT包含该任务（进程）的所有打开文件。JFT表项内容是到SFT表项的索引。
  - SFT的表项数目可由 config.sys文件中的 FILES= 来语句指定，默认是8。