操作系统

目录

概述

操作系统的定义

- 是系统最基本最核心的软件，属于系统软件。
- 控制和管理整个计算机的硬件和软件资源。
- 合理的组织、调度计算机的工作与资源的分配。
- 为用户和其它软件提供方便的接口和环境。

操作系统的功能

计算机系统资源的管理者

管理软硬件资源、合理的组织、调度计算机的工作与资源的分配。

处理器（CPU）管理

在多道程序环境下，CPU的分配和运行都以进程（或线程）为基本单位，因此对CPU的管理可理解为对进程的管理。进程管理的主要功能包括进程控制、进程同步、进程通信、死锁处理、处理机调度等。

存储器管理

为多道程序的运行提供良好的环境，方便用户使用及提高内存的利用率，主要包括内存分配与回收、地址映射、内存保护与共享和内存扩充等功能。

文件管理

计算机中所有的信息都是以文件的形式存在的，操作系统中负责文件的管理的部分称为文件系统，文件管理包括文件存储空间的管理、目录管理及文件读写管理和保护等。

设备管理

设备管理的主要任务是完成用户的I/O请求，方便用户使用各种设备，并提高设备的利用率，主要包括缓存管理、设备分配、设备处理和虚拟设备等功能。

用户与计算机硬件系统之间的接口

为了让用户方便、快捷、可靠的操作计算机硬件并执行自己的程序，操作系统提供了用户接口。

操作系统提供的用户接口分为三类：命令接口、程序接口、图形用户接口。

命令接口

用户可以直接使用的，利用这些操作命令来组织和控制作业的执行。

命令接口分为两类：联机命令接口和脱机命令接口。

- 联机命令接口：又称交互式命令接口，适用于分时或实时系统的接口，由一组键盘操作命令组成。用户输入一条指令，操作系统就执行一条指令。
- 脱机命令接口：又称批处理接口，使用于批处理系统，由一组作业控制命令组成。用户输入一堆指令，操作系统运行一堆指令。在操作系统运行这些命令时用户不可干预。

批处理（$Batch$），也称为批处理脚本。顾名思义，批处理就是对某对象进行批量的处理，通常被认为是一种简化的脚本语言，它应用于DOS和$Windows$系统中。批处理文件的扩展名为bat。

程序接口

用户通过程序间接使用的，编程人员可以使用它们来请求操作系统服务。由一组**系统调用**（也称**广义指令**）组成。

用户通过在程序中使用这些系统调用来请求操作系统为其提供服务（也称为系统调用），只能通过用户程序间接调用，如使用各种外部设备、申请分配和回收内存及其它各种要求。

图形用户接口

即图形用户界面GUI。指采用图形方式显示的计算机操作用户界面。图形接口不算操作系统的一部分，但是其调用的系统调用命令是操作系统的一部分。

作为扩充机器（虚拟机）

没有任何软件支持的计算机称为裸机。

覆盖了软件的机器称为扩充机器或虚拟机。

操作系统的特征

操作系统具有并发、共享、虚拟、异步四个特征，其中并发和共享是两个最基本的特征，两者互为存在条件。

并发

- 并发：两个或多个事件在同一时间间隔内发生，这些事件在宏观上是同时发生的，在微观上是交替发生的，操作系统的并发性指系统中同时存在着多个运行的程序。
- 并行：两个或多个事件在同一时刻发生。
- 一个单核（CPU）同一时刻只能执行一个程序，因此操作系统会协调多个程序使他们交替进行（这些程序在宏观上是同时发生的，在微观上是交替进行的）。
- 操作系统是伴随着“多道程序技术”出现的，因此操作系统和并发是一同诞生的。
- 在如今的计算机中，一般都是多核CPU的，即在同一时刻可以并行执行多个程序，比如计算机是八核的，计算机可以在同一时刻并行执行八个程序，但是事实上计算机执行的程序并不止八个，因此并发技术是必须存在的，并发性必不可少。

共享

- 资源共享即共享，是指系统中的资源可以供内存中多个并发执行的进程共同使用。
- 共享分为两类：互斥共享和同时共享。

互斥共享

- 计算机中的某个资源在一段时间内只能允许一个进程访问，别的进程没有使用权。
- 临界资源（独占资源）：在一段时间内只允许一个进程访问的资源，计算机中大多数物理设备及某些软件中的栈、变量和表格都属于临界资源，它们被要求互斥共享。
- 比如QQ和微信视频。同一段时间内摄像头只能分配给其中一个进程。

同时共享

- 计算机中的某个资源在在一段时间内可以同时允许多个进程访问。
- 同时共享通常要求一个请求分为几个时间片段间隔的完成，即交替进行，微观上“分时共享”。
- 这里的同时指在宏观上是同时的，在微观上是交替进行访问的，只是CPU处理速度很快，我们感觉不到，在宏观上感觉是在同时进行。
- 举个例子：比如QQ在发送文件A，微信在发送文件B，宏观上两个进程A和B都在访问磁盘，在我们看来是同时进行的，但是在微观上两个进程A和B是交替进行访问磁盘的，只是时间太短，CPU处理速度太快，我们感觉不到。
- 有时候多个进程可能真的是在同时进行资源访问，比如玩游戏时可以放音乐，游戏声音和音乐声音都能听见。

虚拟

虚拟技术

- 虚拟是把一个物理上的实体变为若干逻辑上的对应物。
- 物理实体（前者）是实际存在的；而后者是虚的，是用户感觉上的事务。
- 虚拟技术：用于实现虚拟的技术。
- 虚拟处理器（CPU）：通过多道程序设计技术，采用让多道程序并发执行的方法，分时来使用一个CPU，实际物理上只有一个CPU，但是用户感觉到有多个CPU。
- 虚拟存储器：从逻辑上扩充存储器容量，用户感觉到的但实际不存在的存储器。
- 虚拟设备：将一台物理设备虚拟为逻辑上的多台设备，使多个用户在同一时间段内访问同一台设备，即同时共享，用户宏观上感觉是同时的，但实际上是微观交替访问同一台设备的。
- 操作系统的虚拟技术科归纳为：
  - 时分复用技术：如处理器的分时共享。
  - 空分复用技术：如虚拟存储器。

异步

- 异步指多道程序环境允许多个程序并发执行，但由于资源有限，如CPU只有一个，进程的执行并不是一贯到底的，而是走走停停的，它以不可预知的速度向前推进。
- 比如A进程正在占用CPU计算，B进程这时也想占用CPU计算，B进程只有等，等A进程算完了，A进程去访问磁盘资源了，这时B进程再占用CPU进行计算，B进程还没计算完，A进程从磁盘取出资源了，A进程发现B这时在占用CPU，这时A进程就需要等待，等B算完后再继续到CPU中进行计算。由于每个进程占用资源的时间不固定，所以进程的执行以不可预知的速度前进。
- 操作系统必须在运行环境相同的条件下保证运行结果的一致性。

多道程序设计

多道程序设计：是指在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制之下，相互穿插的运行。 两个或两个以上程序在计算机系统中同处于开始到结束之间的状态。这就称为多道程序设计。多道程序技术运行的特征：多道、宏观上并行、微观上串行。

- 制约性：共享资源和相互协同产生了竞争。
- 间断性：为了竞争公平，需要间断释放资源重新竞争。
- 共享性。
- ~~封闭性~~。
- ~~顺序性：单道程序设计特征~~。

操作系统的发展阶段

手动操作阶段

无操作系统。

- 缺点：
  - 用户独占全机。
  - 人机速度矛盾。

单道批处理系统

- 引入脱机输入/输出技术（磁带），并用监督程序复杂控制作业的输入输出。
- 特点：
  - 自动性。
  - 顺序性。
  - 单道性。

- 优点：
  - 缓解了一定的人机速度矛盾，资源利用率得到提升。
- 缺点：
  - 内存只能有一道程序运行。只能串行执行。
  - CPU大量时间用于等待I/O的完成，资源利用率低。

多道批处理系统

- 操作系统正式诞生，引入中断技术，从而能并发执行程序。
- 特点：
  - 多道。
  - 宏观并行。
  - 微观串行。
- 优点：
  - 多道程序并发执行，共享计算机资源。
  - 资源利用率提升，CPU和其他资源基本上忙碌，系统吞吐量增大。
- 缺点：
  - 用户响应时间长。
  - 无法人机交互，不能控制作业执行。
- 问题：
  - 处理器分配。
  - 多道程序内存分配。
  - I/O设备分配。
  - 程序数据组织存放。

操作系统的分类

由于无法人机交互，操作系统进一步发展，出现了不同种类的操作系统。

分时操作系统

- 计算机以时间片为单位轮流为各个用户/作业服务，每个用户可以通过终端与计算机交互。
- 特点：
  - 同时性。
  - 交互性。
  - 独立性。
  - 及时性。
- 优点：
  - 用户请求可以及时响应，解决人机交互问题。

  - 运行多个用户共同使用一台计算机，且用户对计算机的操作相互独立，感受不到其他人存在。
- 缺点：
  - 不能处理紧急任务，操作系统对于每个用户都是公平的，循环给出时间片。
- 非抢占式优先级高者优先算法。

实时操作系统

- 能优先响应紧急任务，不用等待时间片排队。
- 特点：
  - 及时性。
  - 可靠性。
- 优点：
  - 能优先处理紧急任务，使用率更高。
  - 对于紧急事件能有效高速处理，可靠性较高。
- 分类：
  - 硬实时系统：对某个动作必须绝对在指定时间内完成，如导弹系统、股票系统、自动驾驶系统等。
  - 软实时系统：对某个动作可以接受偶尔违反规定且伤害较小，如订票系统，旅店管理系统。
- 抢占式优先级高者优先算法。

其他操作系统

- 网络操作系统：伴随计算机网络发展，能连接网络中各个计算机从而能传输数据，实现网络中各种资源的共享和计算机之间的通信，如$Windows\ NT$。
- 分布式操作系统：具有分布性和并行性，系统中各个计算机地位相同，任何工作可以分布在这些计算机上，并行写协同完成任务。
- 个人计算机操作系统：方便个人使用的操作系统，如$Windows\ 10$、$MacOS$等。

操作系统运行环境

系统开机后，操作系统的程序会被自动加载到内存中的系统区，这段区域是RAM。

操作系统运行机制

两种指令

- 指令就是处理器能识别和执行的最基本命令。所以指令能控制处理器，需要给指令进行控制，对于危险的指令要更高的权限。
- 特权指令：

  - 指具有特殊权限的指令，只用于操作系统或其他系统软件，只能操作系统使用，普通用户不能直接使用。
  - 如有关对I/O设备使用的指令、有关访问程序状态的指令、存取特殊寄存器指令。
  - 具体而言如清内存、置时钟、输入输出、分配系统资源、修改虚存的段表和页表，修改用户的访问权限等。
- 非特权指令：
  - 可以被用户自由使用的指令。
  - 如读取时钟、从内存中取数、将运算结果装入内存、算术运算、寄存器清零等。

相关特权指令说明：

- 置时钟：若在用户态下执行“置时钟指令”，则一个用户进程可在时间片还未到之前把时钟改回去，从而导致时间片永远不会用完，进而导致该用户进程一直占用CPU，这显然不合理。
- 输入输出：涉及中断指令，而中断处理由系统内核负责，工作在核心态。

处理机状态

操作系统根据处理器状态来判断是否可以使用特权指令。

操作系统用程序状态字寄存器PSW中的某标志位来标识当前处理器处于什么状态，如0为用户态，1为核心态。

- 用户态（目态）：只能执行非特权指令。
- 核心态（管态）：可以执行特权指令。

计算机通过**硬件**完成操作系统由用户态到核心态的转换，这是通过中断机制来实现的。发生中断事件时（有可能是用户程序发出的系统调用），触发中断，硬件中断机制将计算机状态置为核心态。

计算机通过**操作系统程序**从核心态到用户态的转换，执行完成中断后操作系统自动转换。

程序类别

根据程序所可以使用指令的权限，程序分为两种：

- 内核程序：是系统的管理者，既可以运行特权指令也可以运行非特权指令，运行在核心态。
- 应用程序：为了保证安全，普通应用程序只能执行非特权指令，运行在用户态。

操作系统的内核

操作系统的程序既然分为内核程序和应用程序，就说明有些程序是更重要的。

内核是计算机上配置的底层软件，是操作系统最基本最核心的部分，而实现操作系统内核功能的程序就是内核程序，其他的就是非内核功能。

内核功能分为：

- 时钟管理：
  - 进程等的计时功能。
  - 时钟中断实现进程切换。
- 中断处理：
  - 负责实现中断机制，提高CPU利用率。
  - 只有一小部分功能属于内核，负责保护和恢复现场，转移控制权给相关处理程序。
- 原语：
  - 是一种特殊的公用程序，是最接近硬件的部分。
  - 这种程序的运行具有原子性，运行时间短，调用次数频繁。
  - 定义原语的直接方法是关闭中断，让所有操作一致完成再打开中断。
  - 如设备驱动、CPU切换、进程通信等。
- 系统资源管理功能：可能包含在内核中也可能不包含：
  - 进程管理。
  - 存储器管理。
  - 设备管理。

中断

中断是内核所必要的基本功能。

中断机制的概念

- 并发批处理就需要中断机制。发生中断就表示需要操作系统介入，开展管理工作。
- 用户态道核心态之间的转换是通过中断实现的，且是唯一的途径，通过硬件的控制，类似10标志位。
- 而核心态道用户态之间的切换只用执行一个特权指令，将程序状态字PSW的标志位设置为用户态。

1. 当中断发生时，CPU立刻进入核心态。
2. 当中断发生后，当前运行的进程暂停，并由操作系统内核对中断进行处理。
3. 对于不同的中断信号会进行不同的处理。

中断的分类

- 内中断（异常、例外、陷入）信号来自CPU内部，与当前执行的命令有关，必须立刻处理，且对于无法恢复故障的需要终止进程。
  - 自愿中断：指令中断。如系统调用时使用的访管指令（陷入指令、$trap$指令）引起的**访管中断**。
  - 强迫中断：
    - 硬件故障。（缺页、主存故障造成机器校验中断）
    - 软件中断。（整数除0）
- 外中断（中断）信号来自CPU外部，与当前执行的命令无关。
  - 外设请求：如I/O操作完成时发出的中断信号。
  - 时钟中断：时间片已到。
  - 人工干预。

另一种分类方式内中断分为：

- 陷阱、陷入。
- 故障。
- 终止。

外中断的处理过程

1. 关中断。CPU响应中断后，首先要保护程序的现场状态，在保护现场的过程中，CPU不应响应更高级中断源的中断请求。否则，若现场保存不完整，在中断服务程序结束后，也就不能正确地恢复并继续执行现行程序。
2. 保存断点。为保证中断服务程序执行完毕后能正确地返回到原来的程序，中断隐指令将原来的程序的断点（即程序计数器PC）保存起来。由硬件自动完成。
3. 中断服务程序寻址。其实质是取出中断服务程序的入口地址送入程序计数器PC。
4. 保存现场和屏蔽字。进入中断服务程序后，首先要**操作系统**保存现场，现场信息一般是指程序状态字寄存器$PSWR$和某些通用寄存器的内容，即中断屏蔽字。
5. 开中断。允许更高级中断请求得到响应。
6. 执行中断服务程序。这是中断请求的目的。各中断向量统一存放在中断向量表中，该表由操作系统初始化，硬件找到该中断信号对应的中断向量，中断向量指明中断服务程序入口地址，然后根据其开始执行。
7. 关中断。保证在恢复现场和屏蔽字时不被中断。
8. 恢复现场和屏蔽字。将现场和屏蔽字恢复到原来的状态。
9. 开中断、中断返回。中断服务程序的最后一条指令通常是一条中断返回指令，使其返回到原程序的断点处，以便继续执行原程序。

第一步到第三步是CPU进入中断周期后，由硬件自动完成（中断隐指令），第四到九条是中断服务程序（软件）完成。

终端服务程序本身可能是用户程序，但是具体进入中断进行处理的程序一定是系统程序。

中断处理和子程序调用都会压栈保存数据。子程序调用只需保存程序断点PC内容，即该指令的下一条指令的地址；中断处理不仅要保存断点（PC的内容），还要保存程序状态字寄存器（PSW）的内容。

系统调用

系统调用的概念

系统调用是程序接口的组成部分，用于应用软件调用，也称为**广义指令**。可以认为是一种可供应应用程序调用的特殊函数，应用程序可以发出系统调用请求来获得操作系统的服务。

通过编译，系统调用需要触发访管指令，所以形成一系列参数与访管指令。

系统调用的过程

为什么要使用系统调用来处理应用程序的请求？如果不同进程争用有限的资源，没有良好的处理机制就会混乱。所以操作系统提供系统调用提供统一处理的过程规范，应用程序通过系统调用发出请求，操作系统在根据请求协调管理。

系统调用本质就是指令中断，所以需要特权指令，从而系统调用的相关处理只能在**核心态**下进行。

1. 传递系统调用参数。
2. 执行陷入指令（用户态）。
3. 执行系统调用相应服务程序（核心态）。
4. 返回用户程序。

- 陷入指令也称为访管指令，在用户态执行（因此不是特权指令），执行陷入指令后立刻引发一个内中断，从而CPU进入核心态。
- 发出系统调用请求的是在用户态，而对系统调用的相应处理在核心态下进行。
- 陷入指令是**唯一一个**只能在用户态执行，而不能在核心态执行的指令。

系统调用的类别

- 设备管理：设备的请求、释放、启动。
- 文件管理：文件的读、写、创建、删除。
- 进程管理：进程的创建、撤销、阻塞、唤醒。
- 进程通信：进程之间的消息传递、信号传递。

- 内存管理：内存的分配、回收。

系统调用与库函数的区别

层次	使用关系
普通应用程序	可直接使用系统调用，也可以使用库函数。有的库函数涉及系统调用有些则不涉及
编程语言	向上提供库函数，有时会将系统调用封装为库函数以隐藏系统调用细节，从而使上层系统调用更方便
操作系统	向上提供系统调用
裸机	无

操作系统的体系结构

按功能关系分类可以分为分层式架构和模块化架构；按照内核架构分类可以分为宏内核和微内核。

分层法

层0为硬件，层N为用户接口。每层只能调用紧邻其低层的功能和服务。

优点：

- 便于系统调试和验证，简化系统设计和实现。从底层向上调试。
- 易扩充和维护。

缺点：

- 分层困难，不够灵活。
- 依赖复杂，效率较低。

模块化

将系统功能划分为独立模块，通过接口进行通信。重点是划分模块的方式。

标准：

- 内聚性，模块内部各部分间联系的紧密程度。内聚性越高，模块独立性越好。
- 耦合度，模块间相互联系和相互影响的程度。耦合度越低，模块独立性越好。

优点：

- 提高了操作系统设计的正确性、可理解性和可维护性。
- 增强了操作系统的可适应性。

- 加速了操作系统的开发过程。

缺点：

- 模块间的接口规定很难满足对接口的实际需求。
- 各模块设计者齐头并进，每个决定无法建立在上一个已验证的正确决定的基础上，因此无法找到一个可靠的决定顺序。

宏内核

除了基本功能外还包含进程管理等其他功能。

优点：高性能，速度快。

缺点：

- 内核代码庞大，结构混乱，难以维护。
- 可靠性低，单个服务崩溃全系统崩溃。
- 占用空间大。
- 可移植性差。

微内核

内核功能只包含基本的进程（线程）管理、低级存储器管理、中断和陷入处理。

通过信息传递机制传输数据。

优点：

- 扩展性和灵活性。
- 可靠性和安全性。
- 可移植性。
- 分布式计算。

缺点：因为很多功能不在内核中，需要频繁在核心态和用户态之间切换，效率较低。

外核

外核为一个程序，在内核态中运行，为虚拟机分配资源并管理。

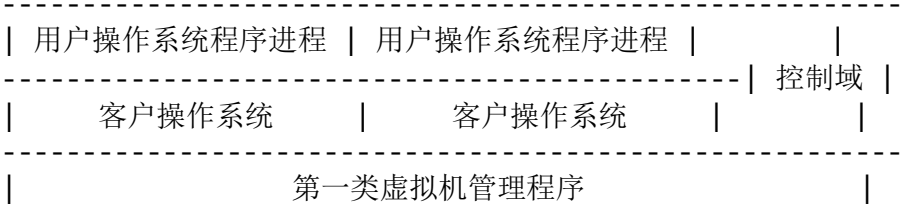
优点：

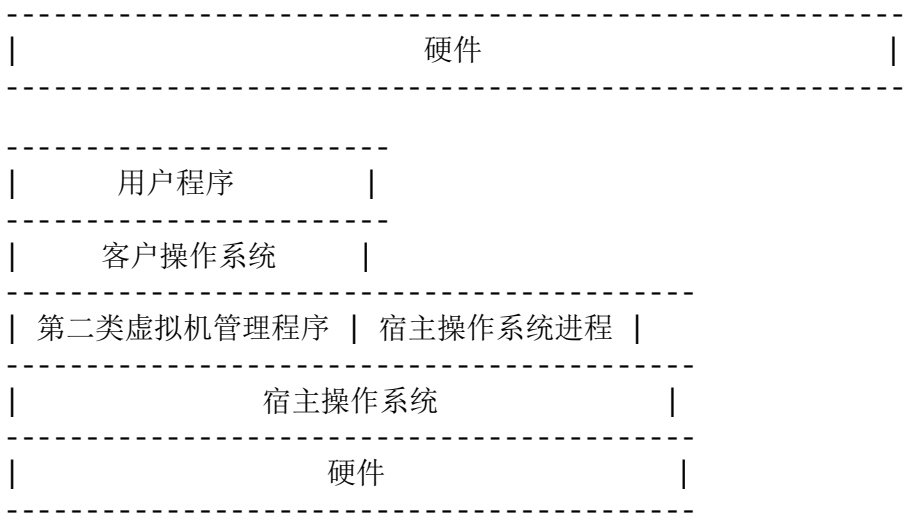
- 减少了映射层，不需要表格重映磁盘地址。
- 将多道程序与操作系统代码分离。

操作系统引导

1. 激活CPU。充电，$CS:IP$指向$FFFF0H$。
2. 执行JMP指令跳转到$BIOS$（基本输入输出系统）。$BIOS$是一组固化到计算机内主板上一个ROM芯片上的程序，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，它可从$CMOS$中读写系统设置的具体信息。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。此外，$BIOS$还向作业系统提供一些系统参数。系统硬件的变化是由$BIOS$隐藏，程序使用$BIOS$功能而不是直接控制硬件。现代作业系统会忽略$BIOS$提供的抽象层并直接控制硬件组件。
3. 登记$BIOS$中断例程入口地址。读取ROM中的$boot$程序，将指令寄存器置为$BIOS$的第一条指令，即开始执行$BIOS$的指令。
4. 硬件自检，并识别已连接的外设。检查硬件是否出现故障。如有故障，主板会发出不同含义的蜂鸣，启动中止。如无故障，屏幕会显示CPU、内存、硬盘等信息。
5. 加载带有操作系统硬盘。硬件自检后，$BIOS$开始读取$Boot\ Sequence$（通过$CMOS$里保存的启动顺序，或者通过与用户交互的方式），把控制权交给启动顺序排在第一位的存储设备，然后CPU将该存储设备引导扇区的内容加载到内存中。
6. 在ROM中加载主引导记录MBR。硬盘以特定的标识符区分引导硬盘和非引导硬盘。如果发现一个存储设备不是可引导盘，就检查下一个存储设备。如无其他启动设备，就会死机。主引导记录MBR的作用是告诉CPU去硬盘的哪个主分区去找操作系统。
7. MBR扫描硬盘分区表，加载硬盘活动分区。MBR包含硬盘分区表，硬盘分区表以特定的标识符区分活动分区和非活动分区。主引导记录扫描硬盘分区表，进而识别含有操作系统的硬盘分区（活动分区）。找到硬盘活动分区后，开始加载硬盘活动分区，将控制权交给活动分区。
8. MBR加载分区引导记录PBR。即该分区的启动程序（操作系统引导扇区），读取活动分区的第一个扇区，这个扇区称为分区引导记录（PBR），其作用是寻找并激活分区根目录下用于引导操作系统的程序（启动管理器）。
9. 加载启动管理器。分区引导记录搜索活动分区中的启动管理器，加载启动管理器。
10. 加载操作系统。

虚拟机





进程管理

进程与线程

进程概念

- 程序指一个指令序列。
- 进程为了满足操作系统的并发性和共享性。
- 进程和程序的根本区别在于其动态性。（注意：所以进程不能与程序相等）
- 系统为每个运行的程序配置一个数据结构，称为**进程控制块 PCB**，用来描述进程的各种信息，如程序代码存放位置。
- PCB、程序段、数据段三个部分构成了进程实体（进程映像），简称为进程。
- 创建进程就是创建PCB，销毁进程就是销毁PCB，PCB是进程存在的唯一标志。
- 进程映像是静态的，而进程是动态的。
- 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。
- 系统资源这里指处理机、存储器和其他设备服务于某个进程的时间单位。

进程结构

- 程序段（代码段）：包括程序代码，程序运行时使用、产生的运算数据，往往是只读的。
- 数据段：存放程序运行过程中处理的各种数据。如全局变量、静态变量、宏定义的常量。
- PCB：操作系统用PCB来管理进程，所以PCB包含所有管理进程所需的信息。

- 进程描述信息：
  - 进程标识符PID：标识进程。
  - 用户标识符UID：标识进程归属的用户，用于共享和保护服务。
- 进程控制和管理信息：
  - 进程当前状态：进程状态信息，作为处理机发分配调度的依据。
  - 进程优先级：进程抢占处理机的优先级。
  - 代码运行入口地址。
  - 程序外存地址。
  - 进入内存时间。
  - 处理机占用时间。
  - 信号量使用。
- 资源分配清单：说明有关内存地址空间或虚拟地址空间的状况，打开文件的列标与使用的输入输出设备信息。
  - 代码段指针。
  - 数据段指针。
  - 堆栈段指针。
  - 文件描述符。
  - 键盘。
  - 鼠标。
- 处理机相关信息：主要处理机中各种寄存器值。
  - 通用寄存器值。
  - 地址寄存器值。
  - 控制寄存器值。
  - 标志寄存器值。
  - 状态字。

- 堆：用来存放动态分配的变量，如调用 `malloc` 函数动态地向高地址分配空间。
- 栈：用来实现函数调用，保存局部变量、函数传递参数，从用户空间的高地址向低地址增长。

进程特征

- 动态性：是进程最基本的特征，进程是程序的一次执行过程，是动态地产生、变化和消亡的。
- 并发性：内存中有多个进程实体，各进程可并发执行。
- 独立性：进程是资源分配、接受调度、能独立运行、独立获得资源、独立接受调度的基本单位。

- 异步性：各进程按各自独立的、不可预知的速度向前推进，操作系统要提供“进程同步机制”来解决异步问题。
- 结构性：每个进程都会配置一个PCB。结构上看，进程由程序段、数据段、PCB组成。

进程组织

当一个系统中存在多个PCB时，需要以适当的方式组织PCB。

- 链接方式：
  - 按照进程状态将PCB分为多个队列。如执行指针（指针数为最大执行并行数）、就绪队列指针（一般会把优先级高的进程放在队头）、阻塞队列指针（许多操作系统会根据阻塞原因的不同将其分为多个阻塞队列）。
  - 操作系统持有指向各个队列的指针。
- 索引方式：
  - 根据进程状态的不同建立索引表。如执行指针、就绪表指针、阻塞表指针。
  - 操作系统持有指向各个索引表的指针。

进程与程序

执行一条命令或运行一个应用程序时，进程和程序之间形成一对一的关系。进程在执行过程中可以加载执行不同的应用程序，从而形成一对多的关系；以不同的参数或数据多次执行同一个应用程序时，形成多对一的关系；并发地执行不同的应用程序时，形成多对多的关系。

进程状态

基本进程状态

具有三种基本状态和其他两种状态：

- 运行态：占有CPU并已经在CPU上运行。
- 就绪态：已经具备运行条件（除处理机外的一切所需资源），但是由于没有空闲CPU，导致暂时不能运行。
- 阻塞态（等待态）：因等待某一事件而暂时不能运行，只有分配其他资源到位才能考虑分配CPU。
- 创建态（新建态）：为进程分配所需的内存空间等系统资源，将转为就绪态。
  - 申请一个空白PCB。
  - 向PCB填写一些控制和管理进程的信息。
  - 系统为进程分配运行所需资源。

  - 进程转为就绪态。
- 终止态（结束态）：进程运行结束或出现错误导致进程被撤销，操作系统需要回收进程资源，撤销PCB等工作，以防止内存泄漏。
- 挂起态：暂时不能获得服务，进程映像调到外存等待（对应进程的PCB还在内存中等待并允许被修改）。分为：
  - 就绪挂起态：准备好后在外存中，只要允许可以随时进入内存。
  - 阻塞挂起态：等待事件在外存中，必须等待一定时间发生。

进程状态转换

- 创建态到就绪态：系统完成创建进程等工作并准备好处理机等资源。
- 就绪态到运行态：占有CPU资源，进程被调用。
- 运行态到就绪态：时间片到或处理机（包含CPU和内存的一系列硬件）被抢占。
- 运行态到阻塞态：进程主动用系统调用的方式申请某种系统资源（由用户态程序调用操作系统内核），或请求等待某个事件发生。
- 阻塞态到就绪态：申请的资源被分配或等待的事件发生，是被动发生。
- 运行态到终止态：进程运行结束或运行过程中遇到不可修复的错误。
- 就绪挂起态到就绪态：激活。
- 就绪态到就绪挂起态：挂起。
- 阻塞挂起态到阻塞态：激活。
- 阻塞态到阻塞挂起态：挂起。
- 阻塞挂起态到就绪挂起态：阻塞事件发生。
- 运行态到就绪挂起态：运行中内存空间不足，或优先级更高的进程进入队列。
- 创建态到就绪挂起态：创建后发现内存空间不足。

进程控制

进程控制概念

- 进程控制指对系统中的所有进程实施有效的管理，具有创建新进程、撤销已有进程、实现进程状态转换功能。
- 如进程组织所说，通过将PCB指针放入各种状态的进程队列中转换进程状态来实现控制。
- 进程控制由**原语**实现。
  - 特点是执行期间不能中断。
  - 这种不能中断的操作就是原子操作。
  - 原语采取“关中断指令”（不监听外部中断信息）和“开中断指令”（开始监听外部中断信息）实现。

- 关/开中断指令的权限很大，所以必然是核心态下执行的特权指令。从而原语必然运行在核心态。

父子进程

允许一个进程创建另一个进程。此时创建者称为父进程，被创建的进程称为子进程。子进程可以继承父进程所拥有的资源。当子进程被撤销时，应将其从父进程那里获得的资源归还给父进程。此外，在撤销父进程时，必须同时撤销其所有的子进程。

父进程创建子进程后，父进程与子进程同时执行（并发）。主程序调用子程序后，主程序暂停在调用点，子程序开始执行，直到子程序返回，主程序才开始执行。

父进程和子进程可以共享一部分资源，但是不能共享虚拟地址空间，因为这是逻辑地址。且其有不同的PCB用于区分进程。

进程控制原语

前四种是必要的。

- 过程创建：
  - 创建原语：
    - 分配一个唯一的进程标识号PID，申请空白PCB，若申请失败，则创建失败。
    - 为新进程分配所需资源，若资源不足，则进入阻塞态。
    - 初始化PCB，主要包括标志信息、处理机状态信息、处理机控制信息、进程优先级。
    - 将PCB插入就绪队列。
  - 引起进程创建的事件：
    - 用户登录：分时系统中，用户登录成功，系统会为其建立一个新进程。
    - 作业调度：多道批处理系统中，有新的作业进入内存时，会为其创建一个新进程。
    - 提供服务：用户向系统提出请求时，会创建一个新进程来处理请求。
    - 应用请求：由用户进程主动请求创建一个子进程。
- 进程终止：
  - 撤销原语：
    - 根据被终止进程的标识符，从PCB集合中找到终止进程的PCB，读取进程的状态。
    - 若进程正在运行，则立刻剥夺其CPU交给其他进程。

        - 若是普通的终止，父进程终止时会将其子进程交给$init$进程收养。
        - 若是整个进程组的进程，则终止其所有子进程。
        - 将进程所有的资源交给父进程或操作系统。
        - 从所在队列或链表中删除PCB。
    - 引起进程终止的事件：
        - 正常结束：任务完成。
        - 异常结束：内部异常，如存储区越界、保护错、运行超时等。
        - 外界干预：外部请求而终止运行，如操作员或操作系统干预、父进程请求、父进程终止。
- 进程阻塞：
    - 阻塞原语：
        - 找到要阻塞的进程对应的PCB。
        - 保护进程运行现场，将PCB状态信息设置为阻塞态，暂停进程。
        - 将PCB插入事件等待队列，将处理机资源调度给其他就绪态进程。
    - 引起进程阻塞的事件：
        - 需要等待系统分配资源。
        - 需要等待相互合作的其他进程完成工作。
- 进程唤醒：
    - 唤醒原语：
        - 在事件等待队列中找到PCB。
        - 将PCB从等待队列中移除，设置进程为就绪态。
        - 将PCB插入就绪队列，等待被调度。
    - 引起进程唤醒的事件：
        - 等待的事件的发生（因何事被阻塞因何事被唤醒）。
- 进程切换：
    - 切换原语：
        - 将处理机上下文信息，包括程序计数器PC和其他寄存器信息存入PCB。
        - PCB移入相应队列。
        - 选择另一个进程执行，并更新其PCB。
        - 更新内存管理的数据结构。
        - 根据PCB恢复新进程所需的运行环境。
    - 引起进程切换的事件：
        - 当前时间片到。

- 有更高优先级进程到达。
- 当前进程主动阻塞。
- 当前进程终止。

进程通信

进程通信是进程之间的信息交换。

为了保证安全，一个进程不能直接访问另一个进程的地址空间，但是又必须有进程通信。PV操作是低级通信方式，高级通信方式有：

- 共享存储：分配一个可以共同使用的共享空间，进程对其的使用必须是互斥的（使用同步互斥工具如PV操作）。
  - 基于数据结构：共享空闲只能放固定的数据结构如数组等。速度慢，限制多，是低级通信方式。
  - 基于存储：内存中划出一块共享存储区，数据的形式、存放位置都由进程控制而非操作系统。速度更快，限制少，是高级通信方式。
- 信息传递：进程间的数据交换以格式化的消息为单位。进程通过操作系统提供的“发送消息/接收消息”两个原语进行数据交换。
  - 消息包括消息头和消息体，消息头包括：发送进程ID、接受进程ID、消息类型、消息长度等格式化的信息。
  - 消息传递包括：
    - 直接通信方式：消息直接挂到接受进程的信息缓冲队列上。
    - 间接通信方式：消息先发送到中间实体（信箱）中，因此也称为信箱通信方式，如计算机网络中的电子邮件系统。
- 管道通信：是消息传递的特殊方式，指用于链接各自一个的读写进程的一个共享文件，又名$pipe$文件，其实就是内存中开辟一个大小**固定**的缓冲区。
  - 只能使用半双工的通信，某一时间段内只能单向传输，若要双向同时通信则必须设置两个管道。
  - 进程需要互斥访问管道，需要满足互斥、同步、确定对方存在。
  - 数据以字符流的形式写入管道，当满时，写进程的$write(\ \ )$系统调用将被阻塞，等待读进程将数据取走。当读进程将数据全部取走后，管道变空，此时读进程的$read(\ \ )$系统调用将被阻塞。
  - 如果没有**写满**则不允许读，如果没有**读空**则不允许写。所以读写都可能被堵塞。
  - 数据一旦被读出就被管道抛弃，所以读进程只能**至多**有一个，否则可能会读错。
- 共享文件（文件系统）：利用操作系统提供的文件共享功能实现进程之间的通信。
- 信号量：也需要信号量来解决文件共享操作中的同步和互斥问题。

由于不同进程拥有不同代码段和数据段，全局变量是针对同一个进程而言，所以全局变量不能用来交换数据。

线程

线程概念

传统而看，进程是程序的一次执行，但是程序的多个功能不能由一个程序顺序处理就能完成，所以一个进程需要同时进行多个任务，所以就引入了线程来增加并发度。

由线程ID、程序计数器PC、寄存器集合和堆栈组成。

线程是CPU**执行**的基本单元，是程序执行流的最小单位，是处理机的分配单元。进程只作为除CPU之外的系统资源的分配单元，即打印机等都是分配给线程而不是进程。

线程与进程

- 系统分配调度：
  - 传统进程机制中，进程是资源分配、调度的基本单位。
  - 引入线程后，进程是**资源分配**的基本单位，线程是**调度**的基本单位。
  - 同一进程中，线程切换不会导致进程切换，在不同进程进行线程切换才会引起进程切换。
- 拥有资源：
  - 进程都是拥有资源的基本单位，线程除了必备的资源不拥有系统资源。
  - 线程可以拥有同进程的所有资源。
  - 同进程的所有线程共享进程地址空间。
- 并发性：
  - 传统进程机制中，只能进程间并发。
  - 引入线程后，各线程间也能并发，提升了并发度。
- 系统开销：
  - 传统的进程间并发，需要切换进程的运行环境，系统开销很大。
  - 线程间并发，如果是同一进程内的线程切换，则不需要切换进程环境，系统开销小。
  - 引入线程后，并发所带来的系统开销减小。
  - 同一进程多个线程共享进程地址空间，所以线程同步通信非常容易。
- 地址空间和其他资源：
  - 进程间不可见，同一进程的线程间可见。
- 通信：

- 进程间通信IPC需要进程同步和互斥来保证一致性。
- 线程间可以直接读写进程数据段如全局变量完成。

线程属性

- 线程是处理机调度的单位。
- 多CPU计算机中，各个线程可占用不同的CPU。
- 每个线程都有一个线程ID、线程控制块TCB（记录线程执行的寄存器和栈等现场状态）。
- 线程也有就绪、阻塞、运行三种基本状态。
- 线程几乎不拥有系统资源。
- 同一进程的不同线程间共享进程的资源。
- 由于共享内存地址空间，同一进程中的线程间通信甚至无需系统干预。
- 同一进程中的线程切换，不会引起进程切换。
- 不同进程中的线程切换，会引起进程切换。
- 切换同进程内的线程，系统开销很小。
- 切换进程，系统开销较大。

TCB

线程控制块TCB是与进程的控制块PCB相似的子控制块，只是TCB中所保存的线程状态比PCB中保存少而已。

线程实现

线程库（$thread\ library$）是为程序员提供创建和管理线程的API。线程库中的线程都是单例模式，所以不同的进程使用线程库中的同名线程都是同一个线程。实现线程库主要的方法有如下两种：

11. 在用户空间中提供一个没有内核支持的库。这种库的所有代码和数据结构都位于用户空间中。这意味着，调用库内的一个函数只导致用户空间中的一个本地函数的调用。
12. 实现由操作系统直接支持的内核级的一个库。对于这种情况，库内的代码和数据结构位于内核空间。调用库中的一个API函数通常会导致对内核的系统调用。

用户级线程ULT：

- 用户级线程由应用程序通过线程库实现。所有的线程管理工作都由应用程序负责（包括线程切换）。
- 用户级线程中，线程切换可以在用户态下即可完成，无需操作系统干预。
- 在用户看来，是有多个线程。但是在操作系统内核看来，并意识不到线程的存在。用户级线程对用户不透明，对操作系统透明。
- 所以操作系统无法操作用户级线程，就不会给用户级线程分配TLB。

优点：

- 线程切换不需要转换到内核空间，节省了模式切换的开销。
- 调度算法可以是进程专用的，不同的进程可根据自身的需要，对自己的线程选择不同的调度算法。
- 用户级线程的实现与操作系统平台无关，对线程管理的代码是属于用户程序的一部分。

缺点：

- 系统调用的阻塞问题，当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都被阻塞。
- 不能发挥多处理机的优势，内核每次分配给一个进程的仅有一个CPU，因此进程中仅有一个线程能执行。

内核级线程（内核支持的线程）KLT：

- 内核级线程的管理工作由操作系统内核完成。
- 线程调度、切换等工作都由内核负责，因此内核级线程的切换必然需要在核心态下才能完成。
- 内核级线程就是从操作系统内核视角看能看到的线程。

优点：

- 能发挥多处理机的优势，内核能同时调度同一进程中的多个线程并行执行。
- 如果进程中的一个线程被阻塞，内核可以调度该进程中的其他线程占用处理机，也可运行其他进程中的线程。
- 内核支持线程具有很小的数据结构和堆栈，线程切换比较快、开销小。
- 内核本身也可采用多线程技术，可以提高系统的执行速度和效率。

缺点：

- 同一进程中的线程切换，需要从用户态转到核心态进行，系统开销较大。这是因为用户进程的线程在用户态运行，而线程调度和管理是在内核实现的。

组合方式：

- 有些系统使用组合方式的多线程实现。
- 在组合实现方式中，内核支持多个内核级线程的建立、调度和管理，同时允许用户程序建立、调度和管理用户级线程。
- 一些内核级线程对应多个用户级线程，这是用户级线程通过时分多路复用内核级线程实现的。
- 同一进程中的多个线程可以同时在多处理机上并行执行，且在阻塞一个线程时不需要将整个进程阻塞。

多线程模型

组合方式同时支持用户线程和内核线程，对于用户级线程如何映射到内核级线程的问题出现了“多线程模型”问题。

- 多对一模型：
  - 多个用户级线程映射到一个内核级线程。每个用户进程只对应一个内核级线程。多个用户级线程共用一个线程控制块。
  - 优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高。
  - 缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行。
- 一对一模型：
  - 一个用户级线程映射到一个内核级线程。每个用户进程有与用户级线程同数量的内核级线程。操作系统给每个用户线程建立一个线程控制块。
  - 优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。
  - 缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。
- 多对多模型：
  - 在同时支持用户级线程和内核级线程的系统中，可以使用二者结合的方式，将n个用户级线程映射到m个内核级线程上（$n \geq m$）。
  - 克服了多对一模型并发度不高的缺点，又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点。

处理机调度

处理机调度的概念

- 处理机：包括中央处理器，主存储器，输入输出接口，加接外围设备就构成完整的计算机系统。处理机是处理计算机系统中存储程序和数据，并按照程序规定的步骤执行指令的部件。
- 处理机调度：在多道程序设计系统中，内存中有多道程序运行，他们相互争夺处理机这一重要的资源。处理机调度就是从就绪队列中，按照一定的算法选择一个进程并将处理机分配给它运行，以实现进程并发地执行。

处理机调度的层次

	工作内容	发生位置	发生频率	对进程状态的影响
高级调度（作业调度）	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存（面向作业）	一个作业一个调入一次调出	无→创建态→就绪态
中级调度（内存调度）	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存（面向进程）	中等	挂起态→就绪态（阻塞挂起→阻塞态）
低级调度(进程调度）	按照某种规则，从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态

高级调度

即作业调度：

- 由于内存空间有限，有时无法将用户提交的作业全部放入内存，因此就需要确定某种规则来决定将作业从外存调入内存的顺序。一般一个作业包含多个进程。
- 高级调度按一定的原则从外存上处于后备队列的作业中挑选一个(或多个）作业，给他们分配内存等必要资源，并建立相应的进程（建立PCB），以使它（们）获得竞争处理机的权利。
- 高级调度是辅存（外存）与内存之间的调度。每个作业只调入一次，调出一次。作业调入时会建立相应的PCB，作业调出时才撤销PCB。
- 多道批处理系统多具备，其他系统一般不需要。
- 高级调度主要是指调入的问题，因为只有调入的时机需要操作系统来确定，但调出的时机必然是作业运行结束才调出。
- 调动执行频率低。

中级调度

即内存调度：

- 引入了虚拟存储技术之后，可将暂时不能运行的进程调至外存等待。等它重新具备了运行条件且内存又稍有空闲时，再重新调入内存。
- 目的是为了提高内存利用率和系统吞吐量。
- 暂时调到外存等待的进程状态为挂起状态。值得注意的是，PCB并不会一起调到外存，而是会常驻内存。PCB中会记录进程数据在外存中的存放位置，进程状态等信息，操作系统通过内存中的PCB来保持对各个进程的监控、管

理。被挂起的进程PCB会被放到内存里的挂起队列中，当条件具备转为就绪态。

- 中级调度就是要决定将哪个处于挂起状态的进程重新调入内存。
- 一个进程可能会被多次调出、调入内存，因此中级调度发生的频率要比高级调度更高。

低级调度

即进程调度：

- 其主要任务是按照某种方法和策略从就绪队列中选取一个进程，将处理机分配给它。
- 进程调度是操作系统中最基本的一种调度，在一般的操作系统中都必须配置进程调度。
- 进程调度的频率很高，一般几十毫秒一次。

进程调度

进程调度是最低级的调度也是其他调度的基础，是内核程序。

调度程序

即调度器，用于调度和分派CPU的组件。

- 排队器。
- 分派器。
- 上下文切换器。

进程调度的时机

需要进行进程调度和切换：

- 当前运行程序主动放弃处理机（非剥夺调度）：
  - 进程正常终止。
  - 出现异常终止。
  - 进程主动请求阻塞（如等待I/O）。
- 当前运行程序被动放弃处理机（剥夺式调度）：
  - 分配时间片用完。
  - 有更紧急的事件需要处理（如I/O中断）。
  - 优先级更高进程进入就绪队列。
  - 中断处理结束。
  - 自陷处理结束。

不能进行进程调度和切换：

- 在处理中断的过程中。中断处理过程复杂，很难做到在中断处理过程中进行进程切换；与硬件密切相关，不属于某一进程。
- 进程在操作系统**内核程序临界区**中。
  - 临界资源：一个时间段内只允许一个进程使用的资源。各进程需要互斥地访问临界资源。
  - 临界区：访问临界资源的那段代码。
  - 内核程序临界区一般是用来访问某种内核数据结构的，比如进程的就绪队列（由各就绪进程的PCB组成）。
  - 进程访问时会上锁，而如果还没退出临界区（还没解锁）就进行进程调度但是进程调度相关的程序也需要访问就绪队列，但此时就绪队列被锁住了，因此无法顺利进行进程调度。
  - 内核程序临界区访问的临界资源如果不尽快释放的话，极有可能影响到操作系统内核的其他管理工作。因此在访问内核程序临界区期间不能进行调度与切换。
  - 而如果是普通程序临界区时，如在打印机打印完成之前，进程一直处于临界区内，临界资源不会解锁。但打印机又是慢速设备，此时如果一直不允许进程调度的话就会导致CPU一直空闲，所以为了保证效率进程在操作系统普通程序临界区时运行进程调度。
- 在原子操作过程中（原语）。原子操作不可中断，更不能切换进程，要一气呵成（如修改PCB中进程状态标志，并把PCB放到相应队列）。

进程切换往往在调度完成后立刻发生，它要求保存原进程当前切换点的现场信息，恢复被调度进程的现场信息。

现场切换时，操作系统内核将原进程的现场信息推入当前进程的内核堆栈来保存它们，并更新堆栈指针。

内核完成从新进程的内核栈中装入新进程的现场信息、更新当前运行进程空间指针、重设PC寄存器等相关工作之后，开始运行新的进程。

进程调度的方式

针对操作系统是否可以剥夺进程处理机，进程调度方式分为：

- 非剥夺调度方式，又称非抢占方式：
  - 只允许进程主动放弃处理机。在运行过程中即便有更紧迫的任务到达，当前进程依然会继续使用处理机，直到该进程终止或主动要求进入阻塞态。
  - 实现简单，系统开销小但是无法及时处理紧急任务。
  - 适合于早期的批处理系统。
- 剥夺调度方式，又称抢占方式：

- 当一个进程正在处理机上执行时，如果有一个更重要或更紧迫的进程需要使用处理机，则立即暂停正在执行的进程，将处理机分配给更重要紧迫的那个进程。
- 可以优先处理更紧急的进程，也可实现让各进程按时间片轮流执行的功能（通过时钟中断）。
- 适合于分时操作系统、实时操作系统。

闲逛进程

在进程切换时，如果系统中没有就绪进程，就会调度闲逛进程（$idle$）运行，如果没有其他进程就绪，该进程就一直运行，并在执行过程中测试中断。闲逛进程的优先级最低，没有就绪进程时才会运行闲逛进程，只要有进程就绪，就会立即让出处理机。

闲逛进程不需要CPU之外的资源，它不会被阻塞。

线程调度

- 用户级线程调度：
  - 由于内核并不知道线程的存在，所以内核还是和以前一样，选择一个进程，并给予时间控制。
  - 由进程中的调度程序决定哪个线程运行。
  - 用户级线程的线程切换在同一进程中进行，仅需少量的机器指令。
- 内核级线程调度：
  - 内核选择一个特定线程运行，通常不用考虑该线程属于哪个进程。
  - 对被选择的线程赋予一个时间片，如果超过了时间片，就会强制挂起该线程。
  - 内核级线程的线程切换需要完整的上下文切换、修改内存映像、使高速缓存失效，这就导致了若干数量级的延迟。

调度算法

指作业与进程的调度算法。

- 批处理系统算法：
  - $FCFS$。
  - $SJF/SPF/SRTN$。
  - $HRRN$。
- 交互式系统算法：
  - RR。
  - PS。
  - MFQ。

	先来先服务	短作业优先	高响应比优先	时间片轮转	多级反馈队列
能否是可抢占	否	是	是	是	队列内算法不一定
优点	公平，实现简单	平均等待时间最少，效率最高	兼顾长短作业	兼顾长短作业	兼颐长短作业，有较好的响应时间，可行性强
缺点	不利于短作业	长作业会饥饿，估计时间不易确定	计算响应比的开销大	平均等待时间较长，上下文切换浪费时间	无
适用于	无	作业调度，批处理系统	无	分时系统	通用
默认决策模式	非抢占	非抢占	非抢占	抢占	抢占

CPU繁忙型更接近于长作业，少I/O所以少中断。而I/O繁忙型需要大量I/O，等待输入输出数据时会阻塞从而重新排队，所以更接近短作业。

算法评价指标

- CPU利用率：CPU忙碌时间占总时间的比例。其中利用率=CPU忙碌（运行）时间÷进程运行总时间。
- 系统吞吐量：单位时间内完成作业的数量。系统吞吐量=总共完成多少道作业÷总时间。
- 周转时间：从作业被提交到系统开始到作业完成为止的时间间隔。
  - 它包括四个部分：作业在外存后备队列上等待作业调度（高级调度）的时间、进程在就绪队列上等待进程调度（低级调度）的时间、进程在CPU上执行的时间、进程等待I/O操作完成的时间。后三项在一个作业的整个处理过程中，可能发生多次。
  - （作业）周转时间=作业完成时间-作业提交时间。
  - 平均周转时间=各作业周转时间之和÷作业数。
  - 带权周转时间=作业周转时间÷作业实际运行的时间=(作业完成时间-作业提交时间)÷作业实际运行的时间（≥ 1）。是一个比值，越靠近1越合理。
  - 平均带权周转时间=各作业带权周转时间之和÷作业数。
- 等待时间：指进程或作业处于等待处理机状态时间之和。

  - 对于进程来说，等待时间就是指进程建立后等待被服务的时间之和，在等待I/O完成的期间其实进程也是在被服务的，所以不计入等待时间。
  - 对于作业来说，不仅要考虑建立进程后的等待时间，还要加上作业在外存后备队列中等待的时间。
  - 一个作业总共需要被CPU服务多久，被I/O设备服务多久一般是确定不变的，因此调度算法其实只会影响作业或进程的等待时间。当然，与前面指标类似，也有“平均等待时间”来评价整体性能。
  - 如果一个进程到达后要么等待要么运行，则等待时间=周转时间-运行时间。
  - 如果一个进程又有计算又有I/O操作，则等待时间=周转时间-运行时间-I/O操作时间。
- 响应时间：从用户提交请求到首次产生响应所用的时间。主要用于交互式系统。

先来先服务

即$FCFS$算法。

- 算法思想：主要从“公平”的角度考虑。
- 算法规则：按照作业/进程到达的先后顺序进行服务。
- 用于作业/进程调度：用于作业调度时，考虑的是哪个作业先到达后备队列；用于进程调度时，考虑的是哪个进程先到达就绪队列。
- 是否可抢占：非抢占式的算法。
- 特点：
  - 优点：公平、算法实现简单。
  - 缺点：排在长作业(进程）后面的短作业需要等待很长时间，带权周转时间很大，对短作业来说用户体验不好。
  - 即$FCFS$算法对长作业有利，对短作业不利。
  - 不能作为分时系统和实时系统的主要调度策略。
  - 利于CPU繁忙型作业，不利于I/O繁忙型作业（即适用于长作业类型）。
- 是否会导致饥饿：不会。

短作业优先

即SJF算法。

- 算法思想：追求最少的平均等待时间，最少的平均周转时间、最少的平均平均带权周转时间。
- 算法规则：最短的作业/进程优先得到服务（所谓“最短”，是指要求服务时间最短）。

- 用于作业/进程调度：即可用于作业调度，也可用于进程调度。用于进程调度时称为“短进程优先(SPF，$Shortest\ Process\ First$）算法”。
- 特点：
  - 优点：“最短的”平均等待时间、平均周转时间。
  - 缺点：
    - 不公平。对短作业有利，对长作业不利。
    - 可能产生饥饿现象。
    - 未考虑作业紧迫性。
    - 另外，作业/进程的运行时间是由用户提供的，并不一定真实，不一定能做到真正的短作业优先。
- 是否可抢占：SJF和SPF是非抢占式的算法。但是也有抢占式的版本———最短剩余时间优先算法（$SRTN, Shortest\ Remaining\ Time\ Next$）。
- 是否会导致饥饿：会。如果源源不断地有短作业/进程到来，可能使长作业/进程长时间得不到服务，产生“饥饿”现象。如果一直得不到服务，则称为“饿死”。

13. 如果题目中未特别说明，所提到的“短作业/进程优先算法”默认是非抢占式的。
14. 很多书上都会说“SJF调度算法的平均等待时间、平均周转时间最少”，严格来说，这个表述是错误的，不严谨的。最短剩余时间优先算法得到的平均等待时间、平均周转时间还要更少。应该加上一个条件“在所有进程同时可运行时，采用SJF调度算法的平均等待时间、平均周转时间最少”，或者说“在所有进程都几乎同时到达时，采用SJF调度算法的平均等待时间、平均周转时间最少”。如果不加上述前提条件，则应该说“抢占式的短作业/进程优先调度算法（最短剩余时间优先，$SRNT$算法）的平均等待时间、平均周转时间最少”。
15. 虽然严格来说，SJF的平均等待时间、平均周转时间并不一定最少，但相比于其他算法（如$FCFS$），SJF依然可以获得较少的平均等待时间、平均周转时间。
16. 如果选择题中遇到“SJF算法的平均等待时间、平均周转时间最少”的选项，那最好判断其他选项是不是有很明显的错误，如果没有更合适的选项，那也应该选择该选项。

高响应比优先

即$HRRN$算法。

- 算法思想：要综合考虑作业/进程的等待时间和要求服务的时间，是$FCFS$和SJF的综合。
- 算法规则：在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务。响应比=(等待时间+要求服务时间)/要求服务时间（响应比≥ 1）。

- 用于作业/进程调度：即可用于作业调度，也可用于进程调度，但是主要用于作业调度。
- 是否可抢占：非抢占式的算法。因此只有当前运行的作业/进程主动放弃处理机时，才需要调度，才需要计算响应比。
- 特点：
  - 综合考虑了等待时间和运行时间（要求服务时间）。
  - 等待时间相同时，要求服务时间短的优先(SJF的优点）。
  - 要求服务时间相同时，等待时间长的优先（$FCFS$的优点）。
  - 对于长作业来说，随着等待时间越来越久，其响应比也会越来越大，从而避免了长作业饥饿的问题。
- 是否会导致饥饿：不会。

时间片轮转

即RR算法。

- 算法思想：公平地、轮流地为各个进程服务，让每个进程在一定时间间隔内都可以得到响应。
- 算法规则：按照各进程到达就绪队列的顺序，轮流让各个进程执行一个时间片（如$100ms$）。若进程未在一个时间片内执行完，则剥夺处理机，将进程重新放到就绪队列队尾重新排队。
- 用于作业/进程调度：用于进程调度（只有作业放入内存建立了相应的进程后，才能被分配处理机时间片）。
- 是否可抢占：若进程未能在时间片内运行完，将被强行剥夺处理机使用权，因此时间片轮转调度算法属于抢占式的算法。由时钟装置发出时钟中断来通知CPU时间片已到。
- 特点：
  - 优点：
    - 公平。
    - 响应快，适用于分时操作系统。
  - 缺点：
    - 由于高频率的进程切换，因此有一定开销。
    - 不区分任务的紧急程度。
- 是否会导致饥饿：不会。
- 常用于分时操作系统，更注重响应时间，所以不怎么关系周转时间。

- 如果时间片太大，每个进程在一个时间片内就可以完完成，则时间片轮转算法会退化为先来先服务算法，增大进程响应时间，所以时间片不能太大。
- 如果时间片太小，进程切换会频繁发生，需要保存现场恢复环境，增加时间开销。
- 时间片分片因素：系统响应时间、就绪队列中的进程数目、系统处理能力。
- 一般设计时间片段时要让切换进程的开销不超过1%。

优先级

也称为优先权调度算法，即PS算法。

- 算法思想：随着计算机的发展，特别是实时操作系统的出现，越来越多的应用场景需要根据任务的紧急程度来决定处理顺序。
- 算法规则：调度时选择优先级最高的作业/进程。I/O繁忙型作业要优于计算繁忙型作业（因为I/O操作需要及时完成，其无法长时间保存输入输出数据），系统进程的优先权应高于用户进程优先权。
- 用于作业/进程调度：既可用于作业调度，也可用于进程调度。甚至，还会用于在之后会学习的I/O调度中。
- 是否可抢占：抢占式、非抢占式都有。做题时的区别在于：
  - 非抢占式只需在进程主动放弃处理机时进行调度即可。
  - 抢占式还需在就绪队列变化时，检查是否会发生抢占，若优先级更高的进程进入就绪队列，则立刻暂停正在运行的进行。
- 特点：
  - 优点：用优先级区分紧急程度、重要程度，适用于实时操作系统。可灵活地调整对各种作业/进程的偏好程度。
  - 缺点：若源源不断地有高优先级进程到来，则可能导致饥饿。
- 是否会导致饥饿：会。
- 优先数与优先级的关系要看具体情况，如$Windows$优先级与优先数成正比，$UNIX$中成反比。
- 优先级调度算法中就绪队列未必只有一个，可以按照不同优先级来组织。
- 可以把优先级更高的进程排在队头位置。
- 根据优先级是否可以动态改变，分为：
  - 静态优先级：创建进程时确定，一直保持不变。依据：进程类型、进程对资源的要求、用户要求。

  - 动态优先级：创建进程时有初始值，之后根据情况动态调整优先级。依据有进程占用CPU时间的长短、就绪进程等待CPU时间的长短。
- 设置进程优先级：
  - 系统进程高于用户进程。
  - 前台进程高于后台进程。即交互性进程高于非交互性进程。
  - 操作系统更偏好I/O型进程（I/O繁忙型进程）而不是计算型进程（CPU繁忙型进程），I/O设备和CPU可以并行工作。如果优先让I/O繁忙型进程优先运行的话，则越有可能让I/O设备尽早地投入工作，则资源利用率、系统吞吐量都会得到提升。
- 调整动态优先级：
  - 若进程在就绪队列中等待了很长时间，则提升其优先级。
  - 若进程占用处理机很长时间，则降低其优先级。
  - 若进程频繁进行I/O操作，则提升其优先级。

多级反馈队列

即MFQ算法。

- 算法思想：对时间片轮转调度算法和优先级调度算法的折中权衡，动态调整进程优先级和时间片大小。
- 算法规则：
  1. 设置多级就绪队列，各级队列优先级从1到k依次递减，时间片从小到大依次变大一倍。
  2. 新进程到达时先进入第1级队列队尾，按$FCFS$原则排队等待被分配时间片，若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经是在最下级的队列，则重新放回该队列队尾。若是被剥夺，则回退到该队列队尾。
  3. 只有第k级队列为空时，才会为$k+1$级队头的进程分配时间片，当又有新进程进入优先级较高的队列则立刻抢占给更够优先级的进程。
- 用于作业/进程调度：用于进程调度。
- 是否可抢占：抢占式的算法。在k级队列的进程运行过程中，若更上级的队列（$1\sim k-1$级）中进入了一个新进程，则由于新进程处于优先级更高的队列中，因此新进程会抢占处理机，原来运行的进程放回k级队列队尾。
- 优缺点：
  - 对各类型进程相对公平（$FCFS$的优点）。
  - 每个新到达的进程都可以很快就得到响应（RR的优点）。
  - 短进程只用较少的时间就可完成(SPF的优点）。
  - 不必实现估计进程的运行时间（避免用户作假）。

  - 可灵活地调整对各类进程的偏好程度，比如CPU密集型进程、I/O密集型进程（拓展：可以将因I/O而阻塞的进程重新放回原队列，这样I/O型进程就可以保持较高优先级）
- 是否会导致饥饿：会。

各就绪队列的调度算法也可能不是时间片调度算法而是别的，但是基本上都是差不多的计算方式。

进程切换

上下文切换

切换CPU到另一个进程需要保存当前进程状态并恢复另一个进程的状态，这个就是上下文切换。进程切换主要需要完成上下文切换的任务。上下文切换实质上是指处理机从一个进程的运行转到另一个进程上运行，在这个过程中，进程的运行环境产生了实质性的变化。

进程切换的过程主要完成了：

- 对原来运行进程各种数据的保存。
- 对新的进程各种数据的恢复。（如程序计数器、程序状态字、各种数据寄存器等处理机现场信息，这些信息一般保存在进程控制块）。

过程如下：

17. 挂起一个进程，保存CPU上下文，包括程序计数器和其他寄存器。
18. 更新PCB信息。
19. 把进程的PCB移入相应的队列，如 I 就绪、在某事件阻塞等队列。
20. 选择另一个进程执行，并更新其PCB。
21. 跳转到新进程PCB中的程序计数器所指向的位置执行。
22. 恢复处理机上下文。

进程切换消耗

进程切换是有代价的，因此如果过于频繁的进行进程调度、切换，必然会使整个系统的效率降低，使系统大部分时间都花在了进程切换上，而真正用于执行进程的时间减少。

可以使用多个寄存器组，这样切换时只需要简单改变当前寄存器组的指针。

进程调度与进程切换区别

- 狭义的进程调度指的是从就绪队列中选中一个要运行的进程。（这个进程可以是刚刚被暂停执行的进程，也可能是另一个进程，后一种情况就需要进程切换）。
- 进程切换是指一个进程让出处理机，由另一个进程占用处理机的过程。

- 广义的进程调度包含了选择一个进程和进程切换两个步骤。
- 调度是一个决策，而切换是一个实际行为。

模式切换与进程切换区别

模式切换指从用户态切换到核心态或相反，其进程中断或异常，本身没有改变当前进程所以没有进程切换。

进程切换只能在核心态中完成。

进程同步与互斥

进程同步与互斥的基本概念

进程同步

- 同步也称为直接制约关系。
- 在多道程序环境下，进程是并发执行的，不同进程之间存在着不同的相互制约关系。为了协调进程之间的相互制约关系，如等待、传递信息等，引入了进程同步的概念。进程同步是为了解决进程的异步问题。
- 异步性：进程具有异步性的特征。指各并发执行的进程以各自独立的、不可预知的速度向前推进。

进程互斥

- 互斥也称间接制约关系。
- 进程互斥指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源。
- 资源共享方式分为：
  - 互斥共享方式：允许多个进程使用，但是同一个时间段内只允许一个进程访问该资源。
  - 同时共享方式：允许一个时间短内由多个进程在宏观上同时对其访问。

临界资源

- 我们把一个时间段内只允许一个进程使用的资源称为临界资源。许多物理设备（比如摄像头、打印机）都属于临界资源。此外还有许多变量、数据、内存缓冲区等都属于临界资源。
- 对临界资源的访问，必须互斥地进行，从逻辑上分为四个部分：
  - 进入区：负责检查是否可进入临界区，若可进入则应设置**正在访问临界资源**的标志（上锁），以阻止其他进程同时进入临界区。
  - 临界区（临界段）：实际访问临界资源的代码。
  - 退出区：负责解除**正在访问临界资源**的标志（解锁）。

- 剩余区：做其他处理。
- 进入区和退出区是实现互斥的代码段。

互斥访问的原则

1. 空闲让进。临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区。
2. 忙则等待。当已有进程进入临界区时，其他试图进入临界区的进程必须等待。
3. 有限等待。对请求访问的进程，应保证能在有限时间内进入临界区（保证不会饥饿）。
4. 让权等待。当进程不能进入临界区时，应立即释放处理机，防止进程忙等待。

进程互斥软件实现

即在进入区设置并检查一些标志来表明是否有进程在临界区，若有则在进入区通过循环检查进行等待，进程离开临界区后则在退出区修改标志。

单标志法

算法思想：两个进程在访问完临界区后会把使用临界区的权限转交给另一个进程。也就是说每个进程进入临界区的权限只能被另一个进程赋予。所以设置一个公用整型变量 `turn` 用来表示允许进入临界区的进程编号，若 `turn=0` 则允许P_0进入临界区。

```
// turn 表示当前允许进入临界区的进程号
int turn = 0;

// P0 进程
// 进入区
while (turn != 1); // ①
// 临界区
critical section; // ②
// 退出区
turn = 1; // ③
// 剩余区
remainder section; // ④

// P1 进程
// 进入区
while(turn != 0); // ⑤
// 临界区
critical section; // ⑥
// 退出区
turn = 0; // ⑦
```

```
// 剩余区
remainder section;// ⑧
```

优点：

- turn 的初值为0，即刚开始只允许0号进程进入临界区。
- 若P_1先上处理机运行，则会一直卡在⑤，无法使用临界资源。
- 直到P_1的时间片用完，发生调度，切换P_0上处理机运行。
- 而代码①不会卡住P_0，P_0可以正常访问临界区，在P_0访问临界区期间即时切换回P_1，P_1依然会卡在⑤。
- 只有P_0在退出区将 turn 改为 1 后，P_1才能进入临界区。
- 因此，该算法可以实现同一时刻最多只允许一个进程访问临界。

缺点：

- turn 表示当前允许进入临界区的进程号，而只有当前允许进入临界区的进程在访问了临界区之后，才会修改 turn 的值。
- 也就是说，对于临界区的访问，一定是按$P_0 \rightarrow P1 \rightarrow P0 \rightarrow P1 \rightarrow \cdots$这样轮流访问。
- 如果一个进程一直不访问临界区，那么临界资源会被这个进程一直占用。
- 违背了空闲让进的原则。

双标志先检查法

算法思想：设置一个布尔型数组 flag[]，数组中各个元素用来标记各进程想进入临界区的意愿，比如 flag[0]=ture 意味着0号进程 P_0现在想要进入临界区。每个进程在进入临界区之前先检查当前有没有别的进程想进入临界区，如果没有，则把自身对应的标志 flag[i]设为 true，之后开始访问临界区。

```
// 表示进入临界区意愿的数组
bool flag[2];
//刚开始设置为两个进程都不想进入临界区
flag [0] = false;
flag [1] = false;

// P0 进程
// 进入区
while (flag[1]); // ①
flag[0] = true; // ②
// 临界区
critical section; // ③
// 退出区
flag [0] = false; // ④
// 剩余区
remainder section;
```

```
// P1 进程
// 进入区
while (flag[0]); // ⑤
flag[1] = true; // ⑥
// 临界区
critical section; // ⑦
// 退出区
flag[1] = false; // ⑧
// 剩余区
remainder section;
```

优点：不需要交替进入。

缺点：若按照①⑤②⑥③⑦……的顺序执行，若P_0和P_1同时检查，发现可以访问，P_0和P_1将会同时访问临界区，违反忙则等待原则。即在检查对方的$flag$后和切换自己的$flag$前之间有一段时间，结果都会检查通过，检查和修改操作不能一次性进行。

双标志后检查法

算法思想：双标志先检查法的改版。前一个算法的问题是先“检查”后“上锁”，但是这两个操作又无法一气呵成，因此导致了两个进程同时进入临界区的问题。因此，人们又想到先“上锁”后“检查”的方法，来避免上述问题。即先将自己标志位设置为 true 再检查对方的标志位，若对方也为 true 则等待，否则进入临界区。

```
// 表示进入临界区意愿的数组
bool flag[2];
//刚开始设置为两个进程都不想进入临界区
flag [0] = false;
flag [1] = false;

// P0 进程
// 进入区
flag[0] = true; // ①
while (flag[1]); // ②
// 临界区
critical section; // ③
// 退出区
flag [0] = false; // ④
// 剩余区
remainder section;

// P1 进程
// 进入区
flag[1] = true; // ⑤
while (flag[0]); // ⑥
```

```
// 临界区
critical section; // ⑦
// 退出区
flag[1] = false; // ⑧
// 剩余区
remainder section;
```

若进程同时想进入临界区，按照①⑤②⑥③⑦……的顺序执行，则都发现对方标志位是 true，P_0和P_1都不能访问临界区都卡死。

因此，双标志后检查法虽然解决了“忙则等待”的问题，但是又违背了“空闲让进”和“有限等待”原则，会因各进程都长期无法访问临界资源而产生“饥饿”现象。

Peterson 算法

算法思想：双标志后检查法中，两个进程都争着想进入临界区，但是谁也不让谁，最后谁都无法进入临界区。$Gary\ L.Peterson$想到了一种方法，如果双方都争着想进入临界区，那可以让进程尝试“孔融让梨”，主动让对方先使用临界区。每个进程再设置自己的标志后再设置一个变量 turn 不允许进入标志的值，再同时检测另一个进程的状态标志位与不允许进入标志。

```
// 表示进入临界区意愿的数组，初始值都是 false
bool flag[2];
// turn 表示优先让哪个进程进入临界区
int turn = 0;

// P0 进程:
// 进入区
flag[0] = true; // ①
turn = 1; // ②
// 当对方也想进且本进程已经让出优先权就等待
while (flag[1] && turn==1); //③
// 临界区
critical section; // ④
// 退出区
flag[0] = false; // ⑤
// 剩余区
remainder section;

// P1 进程:
// 进入区
flag[1] = true; // ⑥ 表示自己想进入临界区
turn = 0; // ⑦ 可以优先让对方进入临界区
while (flag[0] && turn==0); // ⑧ 对方想进，且最后一次是自己“让梨"，那自己就循环等待
// 临界区
critical section; // ⑨
```

```
// 退出区
flag[1] = false; // ⑩ 访问完临界区，表示自己已经不想访问临界区了
// 剩余区
remainder section;
```

如果出现两个进程并发的情况，则其中一个进程必然被卡住，另一个进程必然会执行，所以不存在饥饿和死锁问题。

*Peterson*算法用软件方法解决了进程互斥问题，遵循了空闲让进、忙则等待、有限等待三个原则，但是依然未遵循让权等待的原则。

进程互斥硬件实现

称为**低级方法**，或**元方法**。

中断屏蔽

- 利用“开/关中断指令”实现（与原语的实现思想相同，即在某进程开始访问临界区到结束访问为止都不允许被中断，也就不能发生进程切换，因此也不可能发生两个同时访问临界区的情况）。
- 关中断后即不允许当前进程被中断，也必然不会发生进程切换。
- 直到当前进程访问完临界区，再执行开中断指令，才有可能有别的进程上处理机并访问临界区。
- 优点：简单、高效。
- 缺点：
  - 不适用于多处理机。
  - 只适用于操作系统内核进程，不适用于用户进程（因为开/关中断指令只能运行在内核态，这组指令如果能让用户随意使用会很危险）。

硬件指令

*TS*指令，也有地方称*TestAndSetLock*指令，或*TSL*指令。

*TSL*指令是用硬件实现的原子操作，执行的过程不允许被中断，只能一气呵成。

读出指定标志后把该标志设置为真。

```
// 布尔型共享变量 lock 表示当前临界区是否被加锁
// true 表示已加锁，false 表示未加锁
bool TestAndSet (bool *lock){
  bool old;
  old = *lock; // old 用来存放 lock 原来的值
  *lock = true; // 无论之前是否已加锁，都将 lock 设为 true
  return old; // 返回 lock 原来的值
}

// 以下是使用 TSL 指令实现互斥的算法逻辑
```

```
while (TestAndSet (&lock)); // ""上锁"并"检查"
临界区代码段...
lock = false; // "解锁""
剩余区代码段...
```

- 每个临界资源都有一个共享布尔变量 lock，true 表示被占用，false 表示空闲，是初值。
- 若刚开始 lock 是 false，则TSL返回的 old 值为 false，while 循环条件不满足，直接跳过循环，进入临界区。
- 若刚开始 lock 是 true，则执行TSL后 old 返回的值为 true，while 循环条件满足，会一直循环，直到当前访问临界区的进程在退出区进行“解锁”。
- $lock$负责唤醒处于就绪态的进程。

相比软件实现方法，TSL指令把“上锁”和“检查”操作用硬件的方式变成了一气呵成的原子操作。

- 优点：
  - 实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞。
  - 适用于多处理机环境。
- 缺点：
  - 不满足“让权等待”原则。
  - 暂时无法进入临界区的进程会占用CPU并循环执行TSL指令，从而导致“忙等”。

$Swap$指令有的地方也叫$Exchange$指令，或简称$XCHG$指令。

$Swap$指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。

```
// Swap 指令的作用是交换两个变量的值
Swap (bool *a,bool *b){
  bool temp;
  temp = *a;
  *a=*b;
  *b = temp;
}

// 以下是用Swap 指令实现互斥的算法逻辑
// lock 表示当前临界区是否被加锁
// 上锁
bool old = true;
while (old == true){
  Swap (&lock,&old);
}
临界区代码段...
// 解锁
```

```
lock = false;
剩余区代码段...
```

逻辑上来看$Swap$和TSL并无太大区别，都是先记录下此时临界区是否已经被上锁（记录在 old 变量上），再将上锁标记 lock 设置为 true，最后检查 old，如果 old 为 false 则说明之前没有别的进程对临界区上锁，则可跳出循环，进入临界区。

信号量

之前软硬件实现的进程互斥都无法解决让权等待问题，所以$Dijkstra$提出实现进程互斥和同步的方法——信号量机制。

信号量机制的基础概念

- 用户进程可以通过使用操作系统提供的一对原语来对信号量进行操作，从而很方便的实现了进程互斥、进程同步。
- 信号量其实就是一个变量（可以是一个整数，也可以是更复杂的记录型变量），可以用一个信号量来表示系统中某种资源的数量，比如：系统中只有一台打印机，就可以设置一个初值为1的信号量。
- 原语是一种特殊的程序段，其执行只能一气呵成，不可被中断。原语是由关中断/开中断指令实现的。软件解决方案的主要问题是由“进入区的各种操作无法一气呵成”，因此如果能把进入区、退出区的操作都用“原语”实现，使这些操作能“一气呵成”就能避免问题。
- 一对原语：wait(S)原语和 signal(S)原语（S 表示整型量），可以把原语理解为我们自己写的函数，函数名分别为 wait 和 signal，括号里的信号量 S 其实就是函数调用时传入的一个参数。
- $wait$、$signal$原语常简称为P、V操作（来自荷兰语$proberen$和$verhogen$）。因此，做题的时候常把$wait(S)$、$signal(S)$两个操作分别写为$P(S)$、$V(S)$。

整型信号量

用一个整数型的变量作为信号量，用来表示系统中某种资源的数量。

与普通整数变量的区别:对信号量的操作只有三种，即初始化、P操作、V操作。

```
// 加入某计算机系统中有一台打印机，打印机被争用，如果有 n 台打印机则 S 初始为为 n
//初始化整型信号量 S，表示当前系统中可用的打印机资源数
#define n 1;
int S = n;
// wait 原语，相当于“进入区”
void wait (int &S){
  while (S <= 0); // 如果资源数不够，就一直循环等待
  S=S-1; // 如果资源数够，则占用一个资源
```

```
}

// signal 原语,相当于“退出区”
void signal (int &S) {
  S=S+1; // 使用完资源后，在退出区释放资源
}

// 进程
...
wait(S); // 进入区，申请资源
使用打印机资源... // 临界区，访问资源
signal(S); // 退出区，释放资源
```

“检查”和“上锁”一气呵成，避免了并发、异步导致的问题。

存在的问题：只要信号量$S \leq 0$就不断测试，不满足“让权等待”原则，会发生“忙等”。

记录型信号量

整型信号量的缺陷是存在“忙等”问题，因此人们又提出了“记录型信号量”，即用记录型数据结构表示的信号量。

```
// 记录型信号量的定义
typedef struct {
  int value;// 剩余资源数
  struct process *L; // 等待队列
}semaphore;

// 某进程需要使用资源时，通过wait 原语申请
void wait (semaphore S) {
  S.value--;
  // 如果剩余资源数不够,使用 block 原语使进程从运行态进入阻塞态,并把挂到信号量 S 的等待队列（即阻塞队列）中。
  if (S.value < 0) {
    block(S.L);
  }
}

// 进程使用完资源后，通过 signal 原语释放
void signal (semaphore S) {
  S.value++;
  // 释放资源后，若还有别的进程在等待这种资源，则使用 wakeup 原语唤醒等德队列中一个进程,该进程从阻塞态变为就绪态
  if (S.value <= 0) {
    wakeup(S.L);
  }
}
```

- 在考研题目中 wait(S)、signal(S)也可以记为 P(S)、V(S)，这对原语可用于实现系统资源的“申请”和“释放”。
- S.value 的初值表示系统中某种资源的数目。
- 对信号量 S 的一次P操作意味着进程请求一个单位的该类资源，因此需要执行 S.value--，表示资源数减1，当 S.value<0 时表示该类资源已分配完毕，因此进程应调用 block 原语进行自我阻塞（当前运行的进程从运行态变为阻塞态），主动放弃处理机，并插入该类资源的等待队列 S.L 中。可见，该机制遵循了“让权等待”原则，不会出现“忙等”现象。
- 对信号量 S 的一次*V*操作意味着进程释放一个单位的该类资源，因此需要执行 S.value++，表示资源数加1，若加1后仍是 S.value<=0，表示依然有进程在等待该类资源，因此应调用 wakeup 原语唤醒等待队列中的第一个进程（被唤醒进程从阻塞态→就绪态）。

信号量机制实现进程互斥

互斥信号量默认设置为1。

1. 分析并发进程的关键活动，划定临界区（如对临界资源打印机的访问就应放在临界区）。
2. 设置互斥信号量 mutex，初值为1，表示一次只能有一个进程访问。
3. 互斥信号量取值为0或1，0代表上锁，1代表释放。
4. 在临界区之前执行 P(mutex)。
5. 在临界区之后执行 V(mutex)。
6. 注意：对不同的临界资源需要设置不同的互斥信号量。

```
// 信号量机制实现互斥
// 要会自己定义记录型信号量,但如果题目中没特别说明,可以把信号量的声明简写成这种形式
semaphore mutex=1; // 初始化信号量

P1(){
  ...
  P(mutex); // 使用临界资源前需要加锁
  临界区代码段...
  V(mutex); // 使用临界资源后需要解锁
  ...
}

P2(){
  ...
  P(mutex);
  临界区代码段...
  V(mutex);
  ...
}
```

信号量机制实现进程同步

进程同步：要让各并发进程按要求有序地推进。

同步信号量设置应该分为两种情况，如果期望消息未产生则设置为0，若已经产生，则应该设为非0的正整数。

1. 分析什么地方需要实现“同步关系”，即必须保证“一前一后”执行的两个操作（或两句代码）。如先P_1后P_2。
2. 设置同步信号量 S，初始为0。
3. 同步信号量为整数，释放加一，占用减一。
4. 在“前操作”之后执行 V(S)。
5. 在“后操作”之前执行 P(S)。

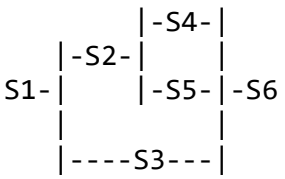
```
// 信号量机制实现同步
// 初始化同步信号量，初始值为0
semaphore S=0;
P1(){
  代码 1;
  代码 2;
  V(S);
  代码 3;
}
P2(){
  P(S);
  代码 4;
  代码 5;
  代码 6;
}
```

- 若先进行进程P_1：
  - 执行到 V(S)操作，则 S++后 S=1。
  - 之后当进行P_2，执行到 P(S)操作时，由于 S=1，表示有可用资源，会执行 S--，S 的值变回0，P_2进程不会执行 block 原语，而是继续往下执行代码4。
- 若先进行进程P_2：
  - 执行到 P(S)操作，由于 S=0，S--后 S=-1，表示此时没有可用资源，因此P操作中会执行 block 原语，主动请求阻塞。
  - 时间片用完后进行进程P_1，之后当执行完代码2，继而执行 V(S)操作，S++，使 S 变回0。
  - 由于此时有进程在该信号量对应的阻塞队列中，因此会在V操作中执行 wakeup 原语，唤醒P_2进程。这样P_2就可以继续执行代码4了。

信号量机制实现前驱关系

前驱关系其实是多组同步。

进程P_1中有句代码S_1，P_2中有句代码S_2……P_6中有句代码S_6。这些代码要求按如下前驱图所示的顺序来执行：



其实每一对前驱关系都是一个进程同步问题（需要保证一前一后的操作）因此：

1. 要为每一对前驱关系各设置一个同步变量。
2. 在“前操作”之后对相应的同步变量执行V操作。类似表示当前动作S_i已经完成。
3. 在“后操作”之前对相应的同步变量执行P操作。类似检测前一个动作S_i是否完成。

令S_1-S_2之间信号量为$a=0$，S_1-S_3之间的信号量$b=0$，S_2-S_4之间信号量为$c=0$，S_2-S_5之间信号量为$d=0$，S_4-S_6之间信号量为$e=0$，S_5-S_6之间信号量为$f=0$，S_3-S_6之间信号量为$g=0$。

每个S_i操作都设置一个进程P_i。

每一条线段靠近根的对当前信号量进行V操作，靠近尾的对当前信号量进行P操作。

再将每个代码结点旁边的操作聚拢在一起，就是每个进程所应该执行的操作。

```
P1() {
  ...
  S1;
  V(a);
  V(b);
  ...
}
P2() {
  ...
  P(a);
  S2;
  V(c);
  V(d);
  ...
}
P3() {
  ...
```

```
  P(b);
  S3;
  V(g);
  ...
)
P4() {
  ...
  P(c);
  S4;
  V(e);
  ...
}
P5() {
  ...
  P(d);
  S5;
  V(f);
  ...
}
P6() {
  ...
  P(e);
  P(f);
  P(g);
  S6;
  ...
}
```

信号量数值

- $value > 0$表示某类可用资源的数量。每次P操作，意味着请求分配一个单位的资源。
- $value <= 0$表示某类资源已经没有，或者说还有因请求该资源而被阻塞的进程。
- $value <= 0$时的绝对值，表示等待进程数目。

进程同步与互斥应用

PV操作题目分析步骤：

1. 关系分析。找出题目中描述的各个进程，分析它们之间的同步、互斥关系。
2. 整理思路。根据各进程的操作流程确定P、V操作的大致顺序。
3. 设置信号量。设置需要的信号量，并根据题目条件确定信号量初值。
4. 互斥信号量初值一般为1，同步信号量的初始值要看对应资源的初始值是多少。

生产者消费者问题

- 系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用。（这里的“产品”理解为某种数据）
- 生产者、消费者共享一个初始为空、大小为n的缓冲区。
- 只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。
- 只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。
- 缓冲区是临界资源，各进程必须互斥地访问。

关系分析：

- 由缓冲区是临界资源，所以对其访问是互斥关系。
- 缓冲区满时，生产者要等待消费者取走产品，消费者取必须在生产者放之前，所以是同步关系。
- 缓冲区空时（即没有产品时），消费者要等待生产者放入产品，消费者取必须在生产者放之后，所以是同步关系。
- 所以分析得到一共两个实体缓冲区与产品和两个进程，生产者每次要消耗P一个空闲缓冲区，并生产V一个产品。消费者每次要消耗P一个产品，并释放一个空闲缓冲区V。往缓冲区放入/取走产品需要互斥。

所以设置三个变量，不能合并：

```
semaphore mutex = 1; // 互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n; // 同步信号量，表示空闲缓冲区的数量，交给生产者判断
semaphore full = 0; // 同步信号量，表示产品的数量，也即非空缓冲区的数量，交给消费者判断

// 生产者
producer() {
  // 不断循环
  while(1){
    生产一个产品;
    P(empty); // 获取一个空闲缓冲区
    // 进入临界区
    P(mutex); // 上锁缓冲区
    把产品放入缓冲区;
    V(mutex); // 解锁缓冲区
    // 离开临界区
    V(full); // 增加一个产品
  }
}
// 消费者
consumer() {
  while (1){
```

```
    P(full); //消耗一个产品（非空缓冲区）
    P(mutex); // 上锁缓冲区
    从缓冲区取出一个产品;
    V(mutex); // 解锁缓冲区
    V(empty); // 增加一个空闲缓冲区
    使用产品;
  }
}
```

若调换P操作顺序会怎么样？（$empty$和$full$的P操作必然在$mutex$的P操作之前，如果先上锁再获取会有问题）

- 若此时缓冲区内已经放满产品，则 `empty=0，full=n`。
- 则生产者进程执行 `P(mutex)`使 `mutex` 变为 `0` 上锁，再执行 `P(empty)`，由于已没有空闲缓冲区，因此生产者被阻塞。
- 由于生产者阻塞，因此切换回消费者进程。消费者进程执行 `P(mutex)`，由于 `mutex` 为 `0`，即生产者还没释放对临界资源的“锁”，因此消费者也被阻塞。
- 这就造成了生产者等待消费者对产品消费来释放空闲缓冲区，而消费者又等待生产者解锁临界区的情况，生产者和消费者循环等待被对方唤醒，出现“死锁”。
- 同样的，若缓冲区中没有产品，即 `full=0，empty=n`，按先消费者后生产者的顺序执行也会发生死锁。
- 因此，实现互斥的P操作一定要在实现同步的P操作之后。可以理解为要先拿到这个空间再上锁，若没有就拿到就上锁，那么其他进程也用不了这个空间。

而由于V操作是释放不会导致进程阻塞，所以两个V操作可以交换顺序。

- 生产者消费者问题是一个互斥、同步的综合问题。
- 对于初学者来说最难的是发现题目中隐含的两对同步关系。
- 有时候是消费者需要等待生产者生产，有时候是生产者要等待消费者消费，这是两个不同的“一前一后问题”，因此也需要设置两个同步信号量。前生产者V后消费者P的关系就是 `full`，前消费者V后生产者者P的关系就是 `empty`。

多生产者多消费者问题

桌子上有一只盘子，每次只能向其中放入一个水果。爸爸专向盘子中放苹果，妈妈专向盘子中放橘子，儿子专等着吃盘子中的橘子，女儿专等着吃盘子中的苹果。只有盘子空时，爸爸或妈妈才可向盘子中放一个水果。仅当盘子中有自己需要的水果时，儿子或女儿可以从盘子中取出水果。用PV操作实现上述过程。

关系分析：

- 父母分别为两个生产者进程，子女分别为两个消费者进程。
- 实体有三个，盘子、苹果、橘子。
- 盘中是一个大小为1，初始为空的缓冲区，对缓冲区盘子需要互斥使用。
- 只有父亲放入苹果女儿才能取出，所以是同步关系。
- 只有母亲放入橘子儿子才能取出，所以是同步关系。
- 只有盘子空时，父亲或母亲才能放水果，所以也是同步关系。
- 而儿子女儿之间没有同步和互斥关系，所以不用管。

所以对于互斥关系设置一个互斥信号量 `mutex=1`，对于苹果设置为 `apple=0`，对于橘子设置为 `orange=0`，对于向拍子放水果设置 `plate=1`（也可以设置为0，后面的处理不同）。

```
// 实现互斥访问盘子（缓冲区）
semaphore mutex = 1;
// 盘子中有几个苹果
semaphore apple = 0;
// 盘子中有几个橘子
semaphore orange= 0;
// 盘子中还可以放多少个水果
semaphore plate = 1;

// 先准备一个苹果，放苹果之前，先判断盘子里是否为空（P 一下盘子，检查盘子中还可以放多少个水果），如果盘子为空进行加锁，然后再将苹果放入进去（V 一下苹果，数量+1）
dad () {
  while (1){
    准备一个苹果;
    P(plate);
    P(mutex);
    把苹果放入盘子;
    V(mutex);
    V(apple);
  }
}

// 先准备一个橘子，放橘子之前，先判断盘子里是否为空（P 一下盘子，检查盘子中还可以放多少个水果），如果盘子为空进行加锁，然后再将橘子放入进去（V 一下橘子，数量+1）
mom() {
  while (1){
    准备一个橘子;
    P(plate);
    P(mutex);
    把橘子放入盘子;
    V(mutex);
    V(orange);
```

```
  }
}

// 拿苹果之前，先判断盘子里有没有苹果（P 一下苹果，若没有苹果，自己被阻塞），
如果有就锁定盘子，取出苹果，解锁，然后告诉父母，盘子为空了（V 一下盘子）
daughter() {
  while (1){
    P(apple);
    P(mutex);
    从盘中取出苹果;
    V(mutex);
    V(plate);
    吃掉苹果;
  }
}

// 拿橘子之前，先判断盘子里有没有橘子（P 一下橘子，若没有橘子，自己被阻塞），
如果有就锁定盘子，取出橘子，解锁，然后告诉父母，盘子为空了（V 一下盘子）
son(){
  while (1){
    P(orange);
    P(mutex);
    从盘中取出橘子;
    V(mutex);
    V(plate);
    吃掉橘子;
  }
}
```

P(plate/apple/orange)和 V(plate/apple/orange)实现了同步，而 P(mutex)和 V(mutex)实现互斥。

如果不使用互斥量会怎么样？

- 刚开始，儿子、女儿进程即使上处理机运行也会被阻塞（$apple$和$orange$刚开始都是0，所以被阻塞）。如果刚开始是父亲进程先上处理机运行，则父亲 P(plate)，可以访问盘子→母亲 P(plate)，阻塞等待盘子→父亲放入苹果 V(apple)→女儿进程被唤醒，其他进程即使运行也都会阻塞，暂时不可能访问临界资源（盘子）→女儿 P(apple)，访问盘子，V(plate)，等待盘子的母亲进程被唤醒→母亲进程访问盘子（其他进程暂时都无法进入临界区）。
- 即使不设置专门的互斥信号量 mutex，也不会出现多个进程同时访问盘子的现象。
- 原因在于本题中的缓冲区大小为 1，在任何时刻，apple、orange、plate 三个同步信号量中最多只有一个是1。因此在任何时刻，最多只有一个进程

的P操作不会被阻塞，并顺利地进入临界区，此时互斥关系全部变成同步关系。

- 而如果缓冲区大于1，则父母两个可以同时访问临界区，可能导致数据覆盖，所以必须使用互斥信号量。

解决“多生产者-多消费者问题”的关键在于理清复杂的同步关系：

- 在分析同步问题（一前一后问题）的时候不能从单个进程行为的角度来分析，要把“一前一后”发生的事看做是两种“事件”的前后关系。
- 比如，如果从单个进程行为的角度来考虑的话，我们会有以下结论：
- 如果盘子里装有苹果，那么一定要女儿取走苹果后父亲或母亲才能再放入水果，如果盘子里装有橘子，那么一定要儿子取走橘子后父亲或母亲才能再放入水果。
- 这么看是否就意味着要设置四个同步信号量分别实现这四个“一前一后”的关系了?
- 正确的分析方法应该从“事件”的角度来考虑，我们可以把上述四对“进程行为的前后关系”抽象为一对“事件的前后关系”。
- 盘子变空事件→放入水果事件。“盘子变空事件”既可由儿子引发，也可由女儿引发;“放水果事件”既可能是父亲执行，也可能是母亲执行。这样的话，就可以用一个同步信号量解决问题了。

读者写者问题

有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程（读进程或写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求:

1. 允许多个读者可以同时对文件执行读操作。
2. 只允许一个写者往文件中写信息。
3. 任一写者在完成写操作之前不允许其他读者或写者工作写。
4. 写者执行写操作前，应让已有的读者和写者全部退出。

具有两类进程：写进程、读进程。

互斥关系：写进程-写进程、写进程-读进程。读进程与读进程不存在互斥问题。

写者进程和任何进程都互斥，设置一个互斥信号量 `rw`，在写者访问共享文件前后分别执行P、V操作。

读者进程和写者进程也要互斥，因此读者访问共享文件前后也要对 `rw` 执行P、V操作。

如果所有读者进程在访问共享文件之前都执行$P(rw)$操作，那么会导致各个读进程之间也无法同时访问文件。所以读者写者问题的核心思想——怎么处理该读者共享的问题呢？即读同步。

P(rw)和 V(rw)其实就是对共享文件的“加锁”和“解锁”。既然各个读进程需要同时访问，而读进程与写进程又必须互斥访问，那么我们可以让第一个访问文件的读进程“加锁”，让最后一个访问完文件的读进程“解锁”。可以设置一个整数变量 count 来记录当前有几个读进程在访问文件，这个也需要保持读进程互斥，避免多个读进程同时操作导致计数错误。

```
// 用于实现对文件的互斥访问。表示当前是否有进程在访问共享文件，1 代表空闲
semaphore rw = 1;
// 记录当前有几个读进程在访问文件
int count = 0;
// 用于保证对 count 变量的互斥访问
semaphore mutex = 1;

writer() {
  while(1){
    P(rw); // 写之前”加锁”
    写文件...
    V(rw); // 写之后"解锁”
  }
}

reader() {
  while(1){
    P(mutex); // 各读进程互斥访问 count
    if(count==0){
      P(rw); //第一个读进程负责"加锁”
    }
    count++; // 访问文件的读进程数+1
    V(mutex); // 释放对 count 的锁
    读文件...
    P(mutex); // 各读进程互斥访问 count
    count--; // 访问文件的读进程数-1
    if(count==0){
      v(rw); // 最后一个读进程负责”解锁"
    }
    V(mutex); // 释放对 count 的锁
  }
}
```

这个算法中读进程是优先的，（因为只要有读进程读写进程就永远无法写而读进程可以一直读），所以写进程可能饥饿。

若希望写进程优先，则当读进程读共享文件时，有写进程访问就立刻禁止后续读进程的请求，当前所有读进程都执行完毕后立刻执行写进程，只有无写进程执行再执行读进程。

所以需要再添加一个信号量并进行一对*PV*操作。

```
// 用于实现对文件的互斥访问。表示当前是否有进程在访问共享文件，1 代表空闲
semaphore rw = 1;
// 记录当前有几个读进程在访问文件
int count = 0;
// 用于保证对 count 变量的互斥访问
semaphore mutex = 1;
// 用于实现"写优先"
semaphore w=1;

writer() {
  while(1){
    P(w); // 当无其他写进程时进入写
    P(rw); // 写之前”加锁”
    写文件...
    V(rw); // 写之后"解锁”
    V(w); // 恢复对共享文件的可读可写
  }
}

reader() {
  while(1){
    P(w); // 当无写进程时进入，若当前有写进程就不允许有新的读进程读，在这里堵塞
    P(mutex); // 各读进程互斥访问 count
    if(count==0){
      P(rw); //第一个读进程负责"加锁”
    }
    count++; // 访问文件的读进程数+1
    V(mutex); // 释放对 count 的锁
    V(w); // 恢复对共享文件的可读可写，w 的 V 操作位置无所谓
    读文件...
    P(mutex); // 各读进程互斥访问 count
    count--; // 访问文件的读进程数-1
    if(count==0){
      v(rw); // 最后一个读进程负责”解锁"
    }
    V(mutex); // 释放对 count 的锁
  }
}
```

当前实现的是实际上是按照进程进入顺序来执行，是**读写公平法**，若有多个读进程下一个执行的进程仍是读进程，没有真正实现写进程最优先。

- 读者写者问题为我们解决复杂的互斥问题提供了一个参考思路。
- 其核心思想在于设置了一个计数器 count 用来记录当前正在访问共享文件的读进程数。我们可以用 count 的值来判断当前进入的进程是否是第一个/最后一个读进程，从而做出不同的处理。
- 另外，对 count 变量的检查和赋值不能一气呵成导致了一些错误，如果需要实现“一气呵成”，自然应该想到用互斥信号量对 count 进行PV操作。
- 最后，还要认真体会我们是如何解决“写进程饥饿”问题的。
- 绝大多数的考研PV操作大题都可以用之前介绍的几种生产者消费者问题的思想来解决，如果遇到更复杂的问题，可以想想能否用读者写者问题的这几个思想来解决。

哲学家进餐问题

一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭。哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左、右两根筷子（一根一根地拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。

系统中有5个哲学家进程，5位哲学家与左右邻居对其中间筷子的访问是互斥关系。

这个问题中只有互斥关系，但与之前遇到的问题不同的是，每个哲学家进程需要同时持有两个临界资源才能开始吃饭。如何避免临界资源分配不当造成的死锁现象，是哲学家问题的精髓。

信号量设置。定义互斥信号量数组 `chopstick[5]={1,1,1,1,1}`用于实现对5个筷子的互斥访问。并对哲学家按$0 \sim 4$编号，哲学家i左边的筷子编号为i，右边的筷子编号为$(i+1)\%5$。

为了防止死锁：

1. 可以对哲学家进程施加一些限制条件，比如最多允许四个哲学家同时进餐。这样可以保证至少有一个哲学家是可以拿到左右两只筷子的。
2. 要求奇数号哲学家先拿左边的筷子，然后再拿右边的筷子，而偶数号哲学家刚好相反。用这种方法可以保证如果相邻的两个奇偶号哲学家都想吃饭，那么只会有其中一个可以拿起第一只筷子，另一个会直接阻塞。这就避免了占有一支后再等待另一只的情况。
3. 同时仅允许当一个哲学家左右两支筷子都可用时才允许他抓起筷子。（一个时刻只有一个哲学家才能尝试拿筷子）

使用第三种方法：

```
semaphore chopstick[5]={1,1,1,1,1}; // 设置五根筷子的信号量
semaphore mutex = 1; // 互斥地取筷子

//i 号哲学家的进程
Pi() {
  while(1){
    P(mutex); // 取筷子前获取互斥量
    P(chopstick[i]); // 拿左
    P(chopstick[(i+1)%5]); // 拿右
    V(mutex); // 释放互斥量
    吃饭...
    V(chopstick[i]); // 放左
    V(chopstick[(i+1)%5]); // 放右
    思考...
  }
}
```

- 哲学家进餐问题的关键在于解决进程死锁。
- 这些进程之间只存在互斥关系，但是与之前接触到的互斥关系不同的是，每个进程都需要同时持有两个临界资源，因此就有“死锁”问题的隐患。
- 如果在考试中遇到了一个进程需要同时持有多个临界资源的情况，应该参考哲学家问题的思想，分析题中给出的进程之间是否会发生循环等待，是否会发生死锁。

抽烟者问题

假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但是要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草、第二个拥有纸、第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者进程一个信号告诉完成了，供应者就会放另外两种材料再桌上，这个过程一直重复（让三个抽烟者轮流地抽烟）。

桌子可以抽象为容量为1的缓冲区，要互斥访问。

组合一：纸+胶水；组合二：烟草+胶水；组合三：烟草+纸。

同步关系（从事件的角度来分析）：桌上有组合一→第一个抽烟者取走东西；桌上有组合二→第二个抽烟者取走东西；桌上有组合三→第三个抽烟者取走东西；发出完成信号→供应者将下一个组合放到桌上。

抽烟者抽烟与供应者准备烟互斥，同理由于缓冲区为1，所以互斥变量 mutex 可有可无，因为四个信号量同时只能有一个为1，天然互斥。

对于同步关系分别设置 offer1=0、offer2=0、offer3=0、finish=0。

```
semaphore offer1 = 0; // 桌上组合一的数量
semaphore offer2 = 0; // 桌上组合二的数量
semaphore offer3 = 0; // 桌上组合三的数量
semaphore finish = 0; // 抽烟是否完成
int i = 0; // 用于实现三个抽烟者轮流抽烟

provider() {
  while(1){
    // 根据 i 选择提供材料
    if(i==0){
      将组合一放桌上;
      V(offer1);
    }
    else if(i==1){
      将组合二放桌上;
      V(offer2);
    }
    else if(i==2){
      将组合三放桌上;
      V(offer3);
    }
    // 向下一个抽烟者提供
    i=(i+1)%3;
    // 等待抽烟者反馈
    P(finish);
  }
}

smoker1() {
  while(1){
    P(offer1); // 占有组合
    从桌上拿走组合一;
    卷烟;
    抽掉;
    // 反馈抽烟完成
    V(finish);
  }
}

smoker2() {
  while(1){
    P(offer2); // 占有组合
    从桌上拿走组合而;
    卷烟;
    抽掉;
```

```
    // 反馈抽烟完成
    V(finish);
  }
}

smoker3() {
  while(1){
    P(offer3); // 占有组合
    从桌上拿走组合三;
    卷烟;
    抽掉;
    // 反馈抽烟完成
    V(finish);
  }
}
```

- 吸烟者问题可以为我们解决“可以生产多个产品的单生产者”问题提供一个思路。
- 值得吸取的精华是“轮流让各个吸烟者吸烟”必然需要“轮流的在桌上放上组合一、二、三”，注意体会我们是如何用一个整型变量i实现这个“轮流”过程的。
- 若一个生产者要生产多种产品（或者说会引发多种前驱事件），那么各个V操作应该放在各自对应的“事件”发生之后的位置。

管程

此前一般使用信号量机制来完成进程互斥同步，但是编写程序困难，易出错。

管程是进程同步工具，解决信号量机制大量同步操作分散的问题。

管程的概念

管程是一种特殊的软件模块，有这些部分组成：

1. 局部于管程的共享数据结构（临界区）与其说明。
2. 对该数据结构进行操作的一组过程或函数。
3. 对局部于管程的共享数据设置初始值的语句。
4. 管程名字。

管程的基本特征：

1. 局部于管程的数据只能被局部于管程的过程所访问。
2. 一个进程只有通过调用管程内的过程才能进入管程访问共享数据。（即管程中定义的共享数据结构只能被管程定义的函数所修改）
3. 每次仅允许一个进程在管程内执行某个内部过程。（在一个时刻内一个函数只能被一个进程使用）

4. 管程是被进程调用的，管程是语法范围，无法创建和撤销。

条件变量

通过条件变量来实现阻塞进程。由于一个进程被阻塞的原因可能有多个，所以管程中设置多个条件变量，每个条件变量保存一个等待队列，用于记录因该条件变量而阻塞的所有进程。对条件变量只有两个操作：`wait` 和 `signal`。所以管程调用这两个操作时都**不用**判断条件，直接阻塞或唤醒。

- `x.wait`：当 x 对应的条件不满足时，正在调用管程的进程调用 `x.wait` 将自己插入 x 条件的等待队列，并释放管程。此时其他进程可以使用该管程。
- `x.signal`：x 对应的条件发生了变化，则调用 `x.signal`，唤醒一个因 x 条件而阻塞的进程。

```
monitor Demo {
  共享数据结构 S;
  condition x; //定义一个条件变量x
  init code(){}
  take away(){
    if(S<=0) x.wait();
    // 资源不够,在条件变量x上阻塞等待
    资源足够，分配资源，做一系列相应处理;
  }
  give_back(){
    归还资源，做一系列相应处理;
    if(有进程在等待) x.signal; //唤醒一个阻塞进程
  }
}
```

条件变量与信号量：

- 相似点：条件变量的*wait/signal*操作类似于信号量的*PV*操作，可以实现进程的阻塞/唤醒。
- 不同点：条件变量是“没有值”的，仅实现了“排队等待”功能（所以一旦调用就不用判断，直接阻塞或释放）；而信号量是“有值”的，信号量的值反映了剩余资源数，而在管程中，剩余资源数用共享数据结构记录。

处理同步互斥问题

1. 需要在管程中定义共享数据（如生产者消费者问题的缓冲区）。
2. 需要在管程中定义用于访问这些共享数据的“入口”——其实就是一些函数（如生产者消费者问题中，可以定义一个函数用于将产品放入缓冲区，再定义一个函数用于从缓冲区取出产品）。
3. 只有通过这些特定的“入口”才能访问共享数据。
4. 管程中有很多“入口”，但是每次只能开放其中一个“入口”，并且只能让一个进程或线程进入（如生产者消费者问题中，各进程需要互斥地访问共享缓冲

区。管程的这种特性即可保证一个时间段内最多只会有一个进程在访问缓冲区。注意:这种互斥特性是由编译器负责实现的，程序员不用关心）。

5. 可在管程中设置条件变量及等待/唤醒操作以解决同步问题。可以让一个进程或线程在条件变量上等待（此时，该进程应先释放管程的使用权，也就是让出“入口”）。可以通过唤醒操作将等待在条件变量上的进程或线程唤醒。

死锁

死锁的概念

死锁的定义

在并发环境下，各进程因竞争资源而造成的一种互相等待对方手里的资源，导致各进程都阻塞，都无法向前推进的现象，就是“死锁”。发生死锁后若无外力干涉，这些进程都将无法向前推进。

死锁、饥饿、死循环

都是进程无法顺利向前推进的现象（故意设计的死循环除外）。

	区别
死锁	死锁一定是“循环等待对方手里的资源”导致的，因此如果有死锁现象，那至少有两个或两个以上的进程同时发生死锁。另外，发生死锁的进程一定处于阻塞态
饥饿	可能只有一个进程发生饥饿。发生饥饿的进程既可能是阻塞态（如长期得不到需要的I/O设备），也可能是就绪态（长期得不到处理机）
死循环	可能只有一个进程发生死循环。死循环的进程可以上处理机运行（可以是运行态），只不过无法像期待的那样顺利推进。

死锁和饥饿问题是由于操作系统分配资源的策略不合理导致的，而死循环是由代码逻辑的错误导致的。

死锁和饥饿是管理者（操作系统）的问题，死循环是被管理者的问题。

死锁发生的条件

1. 系统资源的竞争。
2. 进程推进顺序非法。

产生死锁必须同时满足一下四个条件，只要其中任一条件不成立，死锁就不会发生：

- 互斥条件：只有对必须互斥使用的资源的争抢才会导致死锁（如哲学家的筷子、打印机设备）。像内存、扬声器这样可以同时让多个进程使用的资源是不会导致死锁的（因为进程不用阻塞等待这种资源）。
- 不剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。
- 请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。
- 循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。
  - 发生死锁时一定有循环等待，但是发生循环等待时未必死锁（循环等待是死锁的必要不充分条件）。
  - 如果同类资源数大于1，则即使有循环等待，也未必发生死锁。但如果系统中每类资源都只有一个，那循环等待就是死锁的充分必要条件了。

死锁发生的情况

1. 对系统资源的竞争。各进程对不可剥夺的资源（如打印机）的竞争可能引起死锁，对可剥夺的资源（CPU）的竞争是不会引起死锁的。
2. 进程推进顺序非法。请求和释放资源的顺序不当，也同样会导致死锁。例如，并发执行的进程$P1$、$P2$分别申请并占有了资源$R1$、$R2$，之后进程$P1$又紧接着申请资源$R2$，而进程$P2$又申请资源$R1$，两者会因为申请的资源被对方占有而阻塞，从而发生死锁。
3. 信号量的使用不当也会造成死锁。如生产者-消费者问题中，如果实现互斥的P操作在实现同步的P操作之前，就有可能导致死锁。（可以把互斥信号量、同步信号量也看做是一种抽象的系统资源）

总之，对不可剥夺资源的不合理分配，可能导致死锁。

死锁的处理策略

1. 预防死锁。破坏死锁产生的四个必要条件中的一个或几个。
2. 避免死锁。用某种方法防止系统进入不安全状态，从而避免死锁（银行家算法）。
3. 死锁的检测和解除。允许死锁的发生，不过操作系统会负责检测出死锁的发生，然后采取某种措施解除死锁。

	资源分配策略	各种可能模式	主要优点	主要缺点
死锁预防	保守，宁可资源闲置	一次请求所有资源，资源剥夺，资源按序分配	适用于突发式处理的进程，不必进行剥夺	效率低，进程初始化时间延长；剥夺次数过多；不便灵活申请新资源

	资源分配策略	各种可能模式	主要优点	主要缺点
死锁避免	是“预防”和“检测”的折中（在运行时判断是否可能死锁）	寻找可能的安全允许顺序	不必进行剥夺	必须知道将来的资源需求；进程不能被长时间阻塞
死锁检测	宽松，只要允许就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间，允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

预防死锁

预防死锁是不允许死锁发生的静态策略。

破坏互斥条件

如果把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态。

比如：$SPOOLing$技术。操作系统可以采用$SPOOLing$技术把独占设备在逻辑上改造成共享设备。比如，用$SPOOLing$技术将打印机改造为共享设备，将多个进程的请求合并为一个输出进程。

该策略的缺点：并不是所有的资源都可以改造成可共享使用的资源。并且为了系统安全，很多地方还必须保护这种互斥性。因此，很多时候都无法破坏互斥条件。

破坏不剥夺条件

1. 当某个进程请求新的资源得不到满足时，它必须立即释放保持的所有资源，待以后需要时再重新申请。也就是说，即使某些资源尚未使用完，也需要主动释放，从而破坏了不可剥夺条件。
2. 当某个进程需要的资源被其他进程所占有的时候，可以由操作系统协助，将想要的资源强行剥夺。这种方式一般需要考虑各进程的优先级（比如剥夺调度方式，就是将处理机资源强行剥夺给优先级更高的进程使用）。

该策略的缺点：

1. 实现起来比较复杂。
2. 释放已获得的资源可能造成前一阶段工作的失效。因此这种方法一般只适用于易保存和恢复状态的资源，如CPU，而不能用于打印机。
3. 反复地申请和释放资源会增加系统开销，降低系统吞吐量。
4. 若采用方案一，意味着只要暂时得不到某个资源，之前获得的那些资源就都需要放弃，以后再重新申请。如果一直发生这样的情况，就会导致进程饥饿。

破坏请求和保持条件

可以采用**静态分配方法**，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源了。

该策略实现起来简单，但也有明显的缺点：

1. 有些资源可能只需要用很短的时间，因此如果进程的整个运行期间都一直保持着所有资源，就会造成严重的资源浪费，资源利用率极低。
2. 该策略也有可能导致某些进程饥饿。

破坏循环等待条件

可采用**顺序资源分配法**。首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源（即编号相同的资源）一次申请完。

原理分析：一个进程只有已占有小编号的资源时，才有资格申请更大编号的资源。按此规则，已持有大编号资源的进程不可能逆向地回来申请小编号的资源，从而就不会产生循环等待的现象。

该策略的缺点：

1. 不方便增加新的设备，因为可能需要重新分配所有的编号。
2. 进程实际使用资源的顺序可能和编号递增顺序不一致，会导致资源浪费。
3. 必须按规定次序申请资源，用户编程麻烦。

避免死锁

预防死锁是不允许死锁发生的动态策略。

安全序列与不安全状态

- 所谓安全序列，就是指如果系统按照这种序列分配资源，则每个进程都能顺利完成。只要能找出一个安全序列，系统就是安全状态。当然，安全序列可能有多个。
- 如果分配了资源之后，系统中找不出任何一个安全序列，系统就进入了不安全状态。这就意味着之后可能所有进程都无法顺利的执行下去。当然，如果有进程提前归还了一些资源，那系统也有可能重新回到安全状态，不过我们在分配资源之前总是要考虑到最坏的情况。
- 如果系统处于安全状态，就一定不会发生死锁。如果系统进入不安全状态，就可能发生死锁（处于不安全状态未必就是发生了死锁，但发生死锁时一定是在不安全状态）。

银行家算法

可以在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，以此决定是否答应资源分配请求。这也是“银行家算法”的核心思想。

若每一轮检查都从最小编号开始会更快得到安全序列。

同时还有一个更快的方法得到安全序列：将剩余资源数与最多还需要对比，满足条件的进程全部加入安全序列（而非一个个），然后把归还的资源相加，进行下一轮的比较。

使用代码实现：

假设系统中有n个进程，m种资源。

用一个长度为m的一维数组 $Available$表示当前系统中还有多少可用资源。`Available[j]=K` 表示系统中有R_j类资源K个。

每个进程在运行前先声明对各种资源的最大需求数，则可用一个$n \times m$的矩阵（可用二维数组实现）表示所有进程对各种资源的最大需求数。不妨称为最大需求矩阵Max，`Max[i,j]=K` 表示进程P_i最多需要K个资源R_j。

同理，系统可以用一个$n \times m$的分配矩阵$Allocation$表示对所有进程的资源分配情况。`Allocation[i,j]=K` 表示进程P_i当前已经分片到R_j类资源的数目为K。

$Max - Allocation = Need[i,j]$矩阵，表示各进程最多还需要多少各类资源。

某进程P_i向系统申请资源，可用一个长度为m的一维数组R_i表示本次申请的各种资源量。

可用银行家算法预判本次分配是否会导致系统进入不安全状态：

1. 如果 `Ri[j] <= Need[i,j]`（$0 \le j \le m$）则转向步骤二，否则因为其需要的资源数已经大于最大值，认为出错。
2. 如果 `Ri[j] <= Available[i]`（$0 \le j \le m$），便转向步骤三看，否则表示尚无足够资源，P_i必须等待。
3. 系统试探着把资源分配给进程P_i，并修改相应的数据（并非真的分配，修改数值只是为了做预判）。
   - `Available = Available - Ri;`
   - `Allocation[i,j] = Allocation[i,j] + Ri[j];`
   - `Need[i,j] = Need[i,j] - Ri[j];`
4. 操作系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式分配，否则，恢复相应数据，让进程阻塞等待。

安全性算法，设置工作向量$Work$，有m个元素，表示系统中的剩余可用资源数目。在执行安全性算法开始时，`Work = Available`。

1. 初始时安全序列为空。
2. 从 Need矩阵中找出符合下面条件的行：该行对应的进程不在安全序列中，而且该行小于等于$Work$向量，找到后，把对应的进程加入安全序列；若找不到，则执行步骤四。
3. 进程P_i进入安全序列后，可顺利执行，直至完成，并释放分配给它的资源，因此应执行 `Work = Work + Allocation[i]`，其中 `Allocation[i]`表示进程P_i代表的在$Allocation$矩阵中对应的行，返回步骤二。 若此时安全序列中已有所有进程，则系统处于安全状态，否则系统处于不安全状态。

检测解除死锁

允许死锁的产生。

检测死锁

为了能对系统是否已发生了死锁进行检测，必须：

1. 某种数据结构来保存资源的请求和分配信息。
2. 提供一种算法，利用上述信息来检测系统是否已进入死锁状态。

检测数据结构就是**资源分配图**：

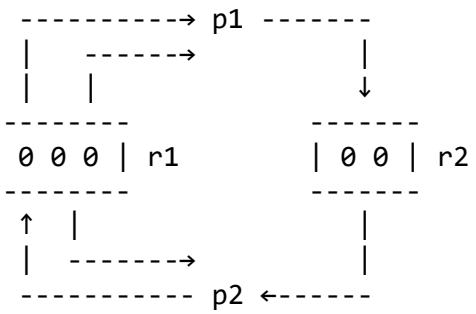
- 两种结点：
  - 进程结点（圆圈）：对应一个进程。
  - 资源结点（方框）：对应一类资源，一类资源可能有多个。
- 两种边：
  - 进程结点→资源结点（请求边）：表示进程想申请几个资源（每条边代表一个）。
  - 资源节点→进程结点（分配边）：表示已经为进程分配了几个资源（每条边代表一个）。
- 如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。
- 如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。
- 相应的，这些被激活的进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程。
- 如果按上述过程分析，最终能消除所有边，就称这个图是可完全简化的。此时一定没有发生死锁（相当于能找到一个安全序列）。
- 如果最终不能消除所有边，那么此时就是发生了死锁。
- 最终还连着边的那些进程就是处于死锁状态的进程。

总结上面的描述，所以检测死锁的算法：在资源分配图中，找出既不阻塞又不是孤点的进程P_i（即找出一条有向边与它相连，且该有向边对应资源的申请数量小于等

于系统中已有空闲资源数量。若所有的连接该进程的边均满足上述条件，则这个进程能继续运行直至完成，然后释放它所占有的所有资源）。消去它所有的请求边和分配边，使之称为孤立的结点。进程P_i所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。

1. 先看系统还剩下多少资源没分配，再看有哪些进程是不阻塞（“不阻塞”即：系统有足够的空闲资源分配给它）的。
2. 把不阻塞的进程的所有边都去掉，形成一个孤立的点，再把系统分配给这个进程的资源回收回来。
3. 看剩下的进程有哪些是不阻塞的，然后又把它们逐个变成孤立的点。
4. 最后，所有的资源和进程都变成孤立的点。这样的图就叫做“可完全简化”。

死锁定理：根据步骤一中的方法进行一系列简化后，若能消去途中所有的边，则称该图是可完全简化的。若是不能完全简化，则系统死锁。

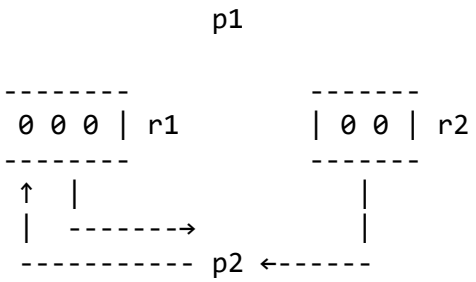


首先先看$R1$资源，它有三个箭头是向外的，因此它一共给进程分配了三个资源，此时，$R1$没有空闲的资源剩余。

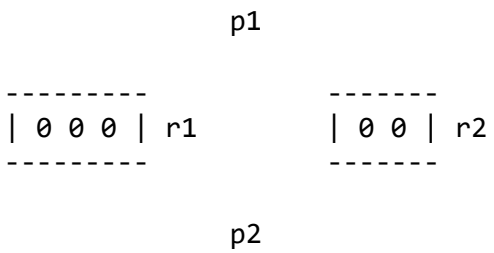
再看$R2$资源，它有一个箭头是向外的，给进程分配了一个资源，此时，$R2$还剩余一个空闲的资源没分配。

看完资源，再来看进程，先看进程$P2$，它只申请一个$R1$资源，但此时$R1$资源已经用光了，所以，进程$P2$进入阻塞状态，因此，进程$P2$暂时不能化成孤立的点。

再看进程$P1$，它只申请一个$R2$资源，此时，系统还剩余一个$R2$资源没分配，因此，可以满足$P1$的申请。这样，进程$P1$便得到了它的全部所需资源，所以它不会进入阻塞状态，可以一直运行，等它运行完后，我们再把它的所有的资源释放。相当于可以把$P1$的所有的边去掉，变成一个孤立的点：



进程$P1$运行完后，释放其所占有的资源（两个$R1$资源和一个$R2$资源），系统回收这些资源后，空闲的资源便变成两个$R1$资源和一个$R2$资源，由于进程$P2$一直在申请一个$R1$资源，所以此时，系统能满足它的申请。这样，进程$P2$便得到了它的全部所需资源，所以它不会进入阻塞状态，可以一直运行，等它运行完后，我们再把它的所有的资源释放。相当于可以把$P2$的所有的边都去掉，化成一个孤立的点：



由于这个资源分配图可完全简化，因此，不会产生死锁。

解除死锁

一旦检测出死锁的发生，就应该立即解除死锁。

并不是系统中所有的进程都是死锁状态，用死锁检测算法化简资源分配图后，还连着边的那些进程就是死锁进程。

解除死锁的主要方法有：

1. 资源剥夺法：挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。
2. 撤销进程法（或称终止进程法）：强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。
3. 进程回退法：让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。

确定挂起或撤销的进程的指标：

1. 进程优先级。
2. 已执行多长时间。
3. 还要多久能完成。
4. 进程已经使用了多少资源
5. 进程是交互式的还是批处理式的。

内存管理

内存管理基础知识

内存概念

- 内存是用于存放数据的硬件。程序执行前需要先放到内存中才能被CPU处理。用于缓冲速度。
- 内存地址从0开始，每个地址对应一个存储单元。
- 存储单元的大小不定：
  - 按字节编址，则每个存储单元大小为$1B$，八个二进制位。
  - 按字编址，每个存储单元大小为一个字，根据计算机的字长确定大小，若字长为16位，则一个字大小为16个二进制位。

内存管理功能

内存分配与回收

操作系统负责内存空间的分配与回收。即后面的普通内存管理内容。

地址映射

操作系统需要提供某种技术从逻辑上对内存空间进行扩充。并且操作系统需要提供地址转换功能，通过内存管理部件MMU，负责程序的逻辑地址与物理地址的转换。

- 相对地址（逻辑地址）：从0号开始，程序员只用知道逻辑地址，可有相同逻辑地址。
- 绝对地址（物理地址）：内存物理单元的集合，是地址转换的最终地址。从逻辑地址到物理地址就是**地址重定位**。

内存保护

操作系统需要提供内存保护功能。保证各进程在各自存储空间内运行，互不干扰：

- 在CPU中设置一对上、下限寄存器，存放进程的上、下限地址。进程的指令要访问某个地址时，CPU检查是否越界。
- 采用重定位寄存器（又称基址寄存器，用于地址相加）和界地址寄存器（又称限长寄存器，用于地址比较）进行越界检查。重定位寄存器中存放的是进程的起始物理地址。界地址寄存器中存放的是进程的最大逻辑地址。

内存共享

只有只读区域才能共享，这种代码就是**可重入代码**（纯代码）允许多个进程同时访问但是不允许任何进程修改的代码。可以复制副本后自己修改。

减少了对程序段的调入/调出，因此减少了对换数量。

程序载入过程

- 程序运行过程：
    1. 编译：由编译程序将用户源代码编译成若干个目标模块（编译就是把高级语言翻译为机器语言）。
    2. 链接：由链接程序将编译后形成的一组目标模块，以及所需库函数链接在一起，形成一个完整的装入模块。
    3. 装入（装载）：由装入程序将装入模块装入内存运行。
    4. 执行。

链接

将独立的逻辑地址合并为完整的逻辑地址：

6. 静态链接：在**编程**时先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。
7. 装入时动态链接：将各目标模块**装入**内存时，边装入边链接的链接方式。
8. 运行时动态链接：在程序**执行**中需要该目标模块时，才对它进行链接，通过硬件转换机制进行转化。其优点是便于修改和更新，便于实现对目标模块的共享。

动态链接与程序逻辑结构有关，所以段式管理有利于动态链接。

装入

将逻辑地址转换为物理地址：

9. 绝对装入（单道程序阶段、未产生操作系统）：在**编译**时，如果知道程序将放到内存中的哪个位置，编译程序将产生绝对地址的目标代码。装入程序按照装入模块中的地址，将程序和数据装入内存。
    - 绝对装入只适用于单道程序环境。
    - 程序中使用的绝对地址，可在编译或汇编时给出，也可由程序员直接赋予。
    - 通常情况下都是编译或汇编时再转换为绝对地址。
10. 静态重定位（可重定位装入）（多道批处理操作系统）：编译、链接后的装入模块的地址都是从0开始的，指令中使用的地址、数据存放的地址都是相对于起始地址而言的逻辑地址。可根据内存的当前情况，使用单独的装入程序将装入模块装入到内存的适当位置。**装入**时对地址进行“重定位”，将逻辑地址变换为物理地址（地址变换是在装入时一次完成的）。
    - 静态重定位的特点是在一个作业装入内存时，必须分配其要求的全部内存空间，如果没有足够的内存，就不能装入该作业。

    - 作业一旦进入内存后，在运行期间就不能再移动，也不能再申请内存空间。
11. 动态重定位（动态运行时装入）（现代操作系统）：编译、链接后的装入模块的地址都是从0开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要**执行**时才进行。因此装入内存后所有的地址依然是逻辑地址。这种方式需要一个重定位寄存器的支持。
    - 采用动态重定位时允许程序在内存中发生移动。
    - 可将程序分配到不连续的存储区中：在程序运行前只需裂入它的部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存。
    - 便于程序段的共享，可以向用户提供一个比存储空间大得多的地址空间。

普通内存管理

分为：

- 连续分配管理：
    - 单一连续分配。
    - 固定分区分配。
    - 动态分区分配。
- 非连续分配管理（离散分配方式）：
    - 基本分页存储管理。
    - 基本分段存储管理。
    - 段页式存储管理。

连续分配是指为用户进程分配的必须是一个连续的内存空间。而非连续分配反之。

- 内部碎片，是已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间。
- 外部碎片，是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域。

单一连续分配

- 内存被分为系统区和用户区。系统区通常位于内存的低地址部分，用于存放操作系统相关数据；用户区用于存放用户进程相关数据。
- 内存中只能有一道用户程序，用户程序独占整个用户区空间。
- 优点：
    - 实现简单。
    - 无外部碎片。

  - 可以采用覆盖技术扩充内存。
  - 因为只有一道程序，所以肯定不会访问越界，不一定需要采取内存保护（如早期的PC操作系统$Ms-Dos$）。
- 缺点：
  - 只能用于单用户、单任务的操作系统中。
  - 有内部碎片。即分配给某进程的内存区域中，如果有些区域没有用上的部分
  - 存储器利用率极低。

固定分区分配

- 将整个用户空间划分为若干个固定大小的分区，在每个分区中只装入一道作业。
- 分区的方式：
  - 分区大小相等：缺乏灵活性，但是很适合用于用一台计算机控制多个相同对象的场合。
  - 分区大小不等：增加了灵活性，可以满足不同大小的进程需求。根据常在系统中运行的作业大小情况进行划分（比如划分多个小分区、适量中等分区、少量大分区）。
- 记录分区的方法：操作系统需要建立一个数据结构——**分区说明表**，来实现各个分区的分配与回收。每个表项对应一个分区，通常按分区大小排列。每个表项包括对应分区的天小、起始地址、状态（是否已分配）。
- 分区分配过程：当某用户程序要装入内存时，由操作系统内核程序根据用户程序大小检索该表，从中找到一个能满足大小的、未分配的分区，将之分配给该程序，然后修改状态为“已分配”。
- 优点：
  - 实现简单。
  - 无外部碎片。
- 缺点：
  - 当用户程序太大时，可能所有的分区都不能满足需求，此时不得不采用覆盖技术来解决，但这又会降低性能。
  - 会产生内部碎片，内存利用率低。

动态分区分配

- 动态分区分配又称为可变分区分配。这种分配方式不会预先划分内存分区，而是在进程**装入**内存时根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要。因此系统分区的大小和数目是可变的。
- 记录分区的方法：
  - 空闲分区表：每空闲分区对应表项。表项中包含分区号、分区大小、分区起始地址、分区状态等信息。

  - 空闲分区链：每个分区的起始部分和末尾部分分别设置前向指针和后向指针。起始部分处还可记录分区大小等信息。
- 分配分区：
  - 当选择的分区分区大小大于分配空间，则分区大小相减，并修改起始地址。
  - 当选择的分区分区大小等于分配空间，则删除该表项。
- 回收分区：
  - 若回收区的后面或前面有一个相邻的空闲分区则合并为一个。
  - 若回收区的后面和前面都有一个相邻的空闲分区则合并为一个。
  - 若回收区的后面或前面都没有一个相邻的空闲分区，则增加一个表项。
- 动态分区分配会导致外部碎片，可用通过**紧凑**（拼接）技术来移动进程位置合并空闲空间。

动态分区分配算法

为了解决动态分区分配方式中如何从多个空闲分区中选择一个分区分配

首次适应算法（$First\ Fit$）：

- 算法思想：每次都从低地址开始查找，找到第一个能满足大小的空闲分区。
- 如何实现：空闲分区以**地址**递增的次序排列。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。
- 优点：
  - 首次适应算法每次都要从头查找，每次都需要检索低地址的小分区。但是这种规则也决定了当低地址部分有更小的分区可以满足需求时，会更有可能用到低地址部分的小分区，也会更有可能把高地址部分的大分区保留下来。
  - 算法开销小，每次分区后不需要对分区队列重新排序。
- 缺点：
  - 造成低分区大量内存碎片。
  - 都要重复经过已经分配的底层区间，增加查找开销。

最佳适应算法（$Best\ Fit$）：

- 算法思想：由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，即，优先使用更小的空闲区。
- 如何实现：空闲分区按**容量**递增次序链接。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。当分配完后需要重新调整空闲分区链（或空闲分区表）。
- 优点：容易保存大分区。

- 缺点：
  - 每次都选最小的分区进行分配，会留下越来越多的、很小的、难以利用的内存块。因此这种方法会产生**最多**的外部碎片且很难查找回收。
  - 算法开销大，每次分区外需要对分区队列进程重新排序。

最坏适应算法（$Worst\ Fit$）或最大适应算法（$Largest\ Fit$）：

- 算法思想：为了解决最佳适应算法的问题——即留下太多难以利用的小碎片，可以在每次分配时优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。
- 如何实现：空闲分区按**容量**递减次序链接。每次分配内存时顺序查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。当分配完后需要重新调整空闲分区链（或空闲分区表）。
- 优点：可用减少难以利用的小碎片。
- 缺点：
  - 每次都选最大的分区进行分配，虽然可以让分配后留下的空闲区更大，更可用，但是这种方式会导致较大的连续空闲区被迅速用完。如果之后有“大进程”到达，就没有内存分区可用了。
  - 算法开销大，每次分区外需要对分区队列进程重新排序。

临近适应算法（$Next\ Fit$）：

- 算法思想：首次适应算法每次都从链头开始查找的。这可能会导致低地址部分出现很多小的空闲分区，而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。如果每次都从上次查找结束的位置开始检索，就能解决上述问题。
- 如何实现：空闲分区以地址递增的顺序排列（可排成一个循环链表）。每次分配内存时从上次查找结束的位置开始查找空闲分区链（或空闲分区表），找到大小能满足要求的第一个空闲分区。
- 优点：减少了检索空闲分区的次数，提高了效率。
- 缺点：邻近适应算法的规则可能会导致无论低地址、高地址部分的空闲分区都有相同的概率被使用，也就导致了高地址部分的大分区更可能被使用，划分为小分区，最后导致无大分区可用。

所以综合效果来看，首次适应算法>最佳适应法?临近适应法>最大适应法。

首次适应法和临近适应法只用简单查找，最佳适应法和最大适应法都需要对可用块进行排序和遍历。

基本分页存储管理

分页

- 将内存空间分为一个个大小相等的分区（比如每个分区$4KB$），每个分区就是一个“页框”（$Page\ Frame$），或称“页帧”、“内存块”、“物理块”。每个页框有一个编号，即“页框号”（或者“内存块号”、“页帧号”、“物理块号”）页框号从0开始。
- 将用户进程的地址空间也分为与页框大小相等的一个个区域，称为“页”（$Page$）或“页面”。每个页面也有一个编号，即“页号”，页号也是从0开始。所以页面不同于页框，是进程的逻辑概念。（进程的最后一个页面可能没有一个页框那么大。）
- 页框不能太大，否则可能产生过大的内部碎片，也不能太小，否则回页面数过大，页表过长占用内存，同时增加硬件地址转换的开销，降低页面换入换出的效率。
- 外存中也同样的单位进行划分，直接称为“块”（$Block$）。
- 操作系统以页框为单位为各个进程分配内存空间。进程的每个页面分别放入一个页框中。也就是说，进程的页面与内存的页框有一一对应的关系。
- 各个页面不必连续存放，也不必按先后顺序来，可以放到不相邻的各个页框中。
- 为了方便计算页号、页内偏移量、页面大小一般设为2的整数幂。
- 如果每个页面大小为2^kB，用二进制数表示逻辑地址，则末尾k位即为页内偏移量，其余部外就是页号。
- 由于是对程序根据内存大小进行分页，所以只对硬件和操作系统是可见的，对于连接装配程序、编译系统、用户都是透明的。

地址结构

- 分为页面偏移量W和页号P两个部分。
- 长度为32位，$0 \sim 11$是页内地址，即页内偏移量，每页大小为$4KB$。
- $12 \sim 31$为页号，代表每个进程内的页的顺序，地址空间最多允许2^{20}页。

页表

- 因为允许将进程的各个页离散地存储在内存不同的物理块中，但系统应能保证进程的正确运行，即能在内存中找到每个页面所对应的物理块，所以为了能知道进程的每个页面在内存中存放的位置，操作系统要为每个进程建立一张页表，其中页表大小也与页面一样被页框约束。
- 一个进程对应一张页表和一个逻辑地址。
- 进程的每一页对应一个页表项。
- 每个页表项由“页号”和“块号”组成。完成从逻辑的页号向物理的块号的映射。
- 页表记录进程页面和实际存放的内存块之间的对应关系。

- 当进程未执行，则页表始址和页表长度在其PCB中，执行时将其装入PTR（页表始址寄存器）中从而进驻内存。

假设某系统物理内存大小为$4GB$，页面大小为$4KB$，则每个页表项至少应该为多少字节？

- 首先$4GB = 2^{32}B$，$4KB = 2^{12}B$。
- 因此$4GB$的内存总共会被分为$2^{32} \div 2^{12} = 2^{20}$个内存块，因此内存块号的范围应该是$0 \cdots 2^{20} - 1$，因此至少要20个二进制位才能表示这么多的内存块号，因此至少要3个字节才够，因为每个字节8位，$3 \times 8 = 24 > 20$。
- 所以最少为三个字节。
- 深入来看，因为每页表项会顺序连续存储在内存中，若该页表在内存中存放的起始地址是X，则M号页对应的页表项存放在内存地址$X + 3 \times M$。
- 同时因为页面大小为$4KB$，所以每个页框（即可用存放的最大值）大小为$4 \times 1024 \div 3 = 1365$个页表项，但是此时每页会余下$4 \times 1024 \bmod 3 = 1$页内碎片，所以一定会在中间空出内存，所以1365号页在页框约束下会在新的下一个页框存储，表项会存放在$X + 3 \times 1365 + 1$处。这时候地址公式就不管用了。
- 而如果每个页表项占4字节，则每个页框刚好能放下1024个页表项，从而没有余数，能减少查找的麻烦。
- 所以理论上$3B$就能表示内存块的范围，但是为了方便页表查找（对齐），实际上会多一些字节，使得每个页面能装下整数个页表项。

页式基本地址变换机构

- 可用借助页表进行转换，通常会在系统中设置一个页表寄存器（PTR），存放页表在内存中的起始地址F和页表长度M（即这个进程里有多少页）。进程未执行时，页表的始址和页表长度放在进程控制块（PCB）中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。
- 在页式存储管理的系统中时，只用确定页面大小和逻辑结构就能得到物理地址。

基本地址变换机构需要先查询页表，再查询内存两次操作：

12. 要算出逻辑地址A对应的页号P与页内偏移量W。页号P=逻辑地址$A \div$页面长度L（取除法的整数部分）。页内偏移量W=A逻辑地址%页面长度L（取除法的余数部分）。
13. 检测页号P是否越界。如果页号P大于等于页表长度M，则内中断（因为页号从0开始，页表长度至少为1，从而$P = M$页会越界）。
14. 根据页表寄存器中的页表项地址PA=页表起始地址F+页号P×页表项长度PL，得到页表中对应的页表项，从而确定页面存放的内存块号B。
    - 页表长度指的是这个页表中总共有几个页表项，即总共有几个页。
    - 页表项长度指的是每个页表项占多大的存储空间。

- 页面大小指的是一个页面占多大的存储空间。

15. 最后物理地址E=内存块号B×页面大小L+页内偏移量W（如果内存块号和业内偏移量用二进制表示，则直接拼接起来就是最终物理地址了）。

快表地址变换机构

- 时间局部性：如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行;如果某个数据被访问过，不久之后该数据很可能再次被访问（因为程序中存在大量的循环）。
- 空间局部性：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问（因为很多数据在内存中都是连续存放的）。
- 快表，又称相联寄存器（TLB），是一种访问速度比内存快很多的高速缓冲存储器，用来存放当前访问的若干页表项，以加速地址变换的过程，此是其物理特性。与此对应，内存中的页表常称为慢表。
- 由于查询快表的速度比查询页表的速度快很多，因此只要快表命中，就可以节省很多时间。
- 因为局部性原理，一般来说快表的命中率可以达到90%以上。
- 快表的地址变换过程：
    1. CPU给出逻辑地址，由某个硬件算得页号、页内偏移量，将页号与快表中的所有页号进行比较。
    2. 如果找到匹配的页号（即命中），说明要访问的页表项在快表中有副本，则直接从中取出该页对应的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表命中，则访问某个逻辑地址仅需一次访存即可。
    3. 如果没有找到匹配的页号，则需要访问内存中的页表，找到对应页表项，得到页面存放的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。同时将其存入快表，以便后面可能的再次访问。但若快表已满，则必须按照一定的页面置换算法对旧的页表项进行替换。因此，若快表未命中，则访问某个逻辑地址需要两次访存。

多级页表

单级页表的缺点：

16. 单级页表的所有表项都必须连续存储，实现起来比较困难。
17. 进程在一段时间内只需要访问少数页面就可用正常运行，无需整个页表都常驻内存。

将页表进行分组离散地放入内存块，并为离散分组的页表再建立一张页表，称为页目录表、外层页表或顶层页表，页目录表页保存序号和内存块号两项。两级页表结构的逻辑地址结构分为一级页号、二级页号和页内偏移量三项。

地址转换过程：

18. 按照地址结构将逻辑地址拆分成三部分。
19. 从PCB中读出页目录表始址，再根据一级页号查页目录表，找到下一级页表在内存中的存放位置（物理地址）。
20. 根据二级页号查表，找到最终想访问的内存块号。
21. 结合页内偏移量得到物理地址。

注意点：

- 若采用多级页表机制，则各级页表的大小不能超过一个页面。因为顶级页表只能有一个，否则一个页面放不下页表项。如果一个页面框放不下就需要多个页面框，而如果需要多个页面框就会导致多个页面框有相同的页号，就不能区分出哪个是顶级页表。
- 两级页表的访存次数分析（假设没有快表机构）：
    1. 第一次访存：访问内存中的页目录表。
    2. 第二次访存：访问内存中的二级页表。
    3. 第三次访存：访问目标内存单元。

多级页表会使用页表基址寄存器（$PTBR$）来存储页表基址，此时$PTBR$会存储当前进程的一级页表的物理地址。

基本分段存储管理

分段

- 进程的地址空间：按照程序自身的逻辑关系划分为若干个不等长的段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址。
- 内存分配规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻。
- 分段系统的逻辑地址结构由段号（段名S）和段内地址（段内偏移量W）所组成。
- 段号的位数决定了每个进程最多可以分几个段段内地址位数决定了每个段的最大长度是多少。
- 页式系统中逻辑地址的页号和页内偏移量对用户透明，但是段式系统段号和段内偏移量必须用户显式提供，一般由编译程序完成。
- 程序分多个段，各段离散地装入内存，为了保证程序能正常运行，就必须能从物理内存中找到各个逻辑段的存放位置。为此，需为每个进程建立一张段映射表，简称“段表”：
    4. 每个段对应一个段表项，其中记录了段号、该段在内存中的起始位置（又称“基址”）和段的长度。
    5. 各个段表项的长度是相同的。

- 即使段是共享的，但是不同的程序中使用，其段号也是不同的。

某系统按字节寻址，采用分段存储管理，逻辑地址结构为（段号16位，段内地址16位），因此用16位即可表示最大段长。物理内存大小为$4GB$（可用32位表示整个物理内存地址空间)。因此，可以让每个段表项占$16+32=48$位，即$6B$。由于段表项长度相同，因此段号可以是隐含的，不占存储空间。若段表存放的起始地址为R，则K号段对应的段表项存放的地址为$R+K\times 6$，段号并不占内存空间。

分段的优点：

- 方便共享和保护。
- 方便编程。
- 方便动态链接和增长。

段式存储地址变换过程

22. 根据逻辑地址得到段号S和段内地址W。
23. 根据段号S是否大于等于段表寄存器的段表长度M，判断上是否越界，若是则越界中断。（段表长度至少为1）。
24. 查询段表寄存器中的段表，找到对应的段表项，段表项分为段长C和段基址B，段表项的存放地址SA=段表始值F+段号S×段表项长度SL。
25. 检测段内地址W是否大于等于段长C（这也是段式存储与页式存储的最大不同，段式存储不定长），若是则越界中断。
26. 计算得到物理地址E=段基址B+段内地址W。

分段分页管理的区别

	与信息的关系	主要目的	是否对用户可见	长度	用户进程地址空间
分页	信息的物理单位	实现离散分配，提高内存利用率	仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的	页的大小固定且由系统决定	是一维的，程序员只需给出一个记忆符即可表示一个地址
分段	信息的逻辑单位	更好地满足用户需求	一个段通常包含着一组属于一个逻辑模块的信息。分段对用户是可见的，用户编程时需要显式地给出段名	不固定，决定于用户编写的程序	二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

- 分段比分页更容易实现信息的共享和保护。
- 只需让各进程的段表项指向同一个段即可实现共享。

- 不能被修改的代码称为纯代码或可重入代码（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的（比如，有一个代码段中有很多变量，各进程并发地同时访问可能造成数据不一致）。
- 分页时页面不是按逻辑模块划分的。这就很难实现共享。

在计算地址时，段式存储的基址往往以十进制的位置给出，计算的结果需要与偏移量相加，最后转换为十六进制，页式存储的基址往往以十六进制的位置给出，计算的结果需要与偏移量直接拼接。

段页式存储管理

	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外,段式管理会产生外部碎片

段页式存储的过程

27. 将进程按逻辑模块分段，再将各段分页（如每个页面$4KB$）。
28. 再将内存空间分为大小相同的内存块/页框/页帧/物理块。
29. 进程前将各页面分别装入各内存块中。

段页式系统逻辑地址结构

- 由段号S、页号P、页内地址W（页内偏移量）组成。段号的位数决定了每个进程最多可以分几个段。页号位数决定了每个段最大有多少页。页内偏移量决定了页面大小、内存块大小是多少。
- 系统含有一个段表寄存器，指出作业的段表始址和段表长度。
- “分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。
- 因此段页式管理的地址结构是二维的。

地址表分为段表和页表，段表只能有一个，页表可以有多个：

- 每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个段表项长度相等，段号是隐含的。
- 每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等，页号是隐含的。
- 一个进程对应一个段表，对应一个或多个页表项，一个段表项就相当于只存一个的页表寄存器。

段页式存储地址变换过程

30. 根据逻辑地址得到段号S、页号P和页内偏移量W。
31. 根据段号S是否大于等于段表寄存器的段表长度M，判断上是否越界，若是则越界中断。（段表长度至少为1）。
32. 查询段表寄存器中的段表，找到对应的段表项。段表项分为段号S和页表长度L、段基址F和页表存放块号D。段表项的存放地址SA=段表始值F+段号S×段表项长度SL。
33. 检测页号P是否大于等于页表项长度PL，若是则越界中断。
34. 根据页表存放块号D和页号P查询页表，找到对应页表项。页表项分为页号P和内存块号B。
35. 计算得到物理地址E=内存块号B+页内偏移量W。
36. 也可引入快表机构，用段号和页号作为查询快表的关键字。若快表命中则仅两次访存。

内存空间扩充

内存空间的扩充（用容量小的内存运行大的程序）有三种技术：

37. 覆盖技术。
38. 交换技术。
39. 虚拟存储技术。使用覆盖和交换可以实现虚拟存储。

前两种技术不是重点。

覆盖和交换技术本质是通过不断换入换出数据，以时间换空间，所以对外存对换区的管理应该以提高交换速度减少交换时间为目标。

覆盖技术

覆盖技术在同一个程序或进程中执行。可以使用在单一连续分配和固定分区分配的方式。

- 覆盖技术的思想：将程序分为多个段（多个模块）。常用的段常驻内存，不常用的段在需要时调入内存。
- 内存中分为一个“固定区”和若干个“覆盖区”。
- 需要常驻内存的段放在“固定区”中，调入后就不再调出（除非运行结束）。
- 不常用的段放在“覆盖区”，需要用到时调入内存，用不到时调出内存。
- 按照自身代码逻辑结构，让那些不可能同时被访问的程序段共享同一个覆盖区。
- 缺点：
  - 必须由程序员声明覆盖结构，操作系统完成自动覆盖。
  - 对用户不透明，增加了用户编程负担。

交换技术

交换技术在不同进程或作业之间进行的。

- 交换（对换）技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）。
- 交换的位置：具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式;对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式（学过文件管理章节后即可理解）。总之，对换区的I/O速度比文件区的更快。
- 交换的时机：交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。例如在发现许多进程运行时经常发生缺页，就说明内存紧张，此时可以换出一些进程；如果缺页率明显下降、就可以暂停换出。
- 交换进程的选择：
  - 可优先换出阻塞进程。
  - 可换出优先级低的进程。
  - 为了防止优先级低的进程在被调入内存后很快又被换出，考虑进程在内存的驻留时间。
- 暂时换出外存等待的进程状态是挂起状态。
- 处理机调度的中级调度（内存调度）就是交换技术的实现。进程的PCB常驻内存。

覆盖用于同一个进程或程序中，交换用于不同进程或作业之间。

虚拟内存管理

虚拟存储技术使用局部性原理，用于对内存空间进行扩充，基于覆盖和交换技术。

传统存储管理特性：

- 一次性：作业必须一次性全部装入内存后才能开始运行。这会造成两个问题：
  1. 作业很大时，不能全部装入内存，导致大作业无法运行。
  2. 当大量作业要求运行时，由于内存无法容纳所有作业，因此只有少量作业能运行，导致多道程序并发度下降。
- 驻留性：一旦作业被装入内存，就会一直驻留在内存中，直至作业运行结束。事实上，在一个时间段内，只需要访问作业的一小部分数据即可正常运行，这就导致了内存中会驻留大量的、暂时用不到的数据，浪费了宝贵的内存资源。

高速缓冲技术的思想：将近期会频繁访问到的数据放到更高速的存储器中，暂时用不到的数据放在更低速存储器中。所以虚拟存储器技术不具备一次性和驻留性。

虚拟内存的基本概念

虚拟内存定义

- 基于局部性原理，在程序装入时，可以将程序中很快会用到的部分装入内存，暂时用不到的部分留在外存，就可以让程序开始执行。
- 在程序执行过程中，当所访问的信息不在内存时，由操作系统负责将所需信息从外存调入内存，然后继续执行程序。
- 若内存空间不够，由操作系统负责将内存中暂时用不到的信息换出到外存。
- 在操作系统的管理下，在用户看来似乎有一个比实际内存大得多的内存，这就是虚拟内存。

虚拟内存容量

- 虚拟内存的最大容量是由计算机的**地址结构**（CPU寻址范围）确定的=CPU寻址范围。
- 虚拟内存的实际容量=min(内存和外存容量之和，CPU寻址范围)。
- 某计算机地址结构为32位，按字节编址，内存大小为512MB，外存大小为2GB。则虚拟内存的最大容量为$2^{32}B = 4GB$，而实际内存是$2GB + 512MB$。

虚拟内存特征

- 多次性：无需在作业运行时一次性全部装入内存，而是允许被分成多次调入内存。
- 对换性：在作业运行时无需一直常驻内存，而是允许在作业运行过程中，将作业换入、换出。
- 虚拟性：从逻辑上扩充可内存的容量，使用户看到的内存容量，远大于实际的容量。

虚拟内存种类

虚拟内存实现需要基于离散分配的内存管理方式基础上。所以根据传统的非连续分配存储管理，可以将虚拟存储的实现分为：

- 请求分页存储管理。
- 请求分段存储管理。
- 请求段页式存储管理。

其主要区别是：在程序执行过程中，当所访问的信息不在内存时，由操作系统负责将所需信息从外存调入内存，然后继续执行程序；若内存空间不够，由操作系统负责将内存中暂时用不到的信息换出到外存。所以操作系统需要提供请求调页或请求调段的功能与页面置换或段置换的功能。

不管哪种方式，都需要有一定的硬件支持。一般需要的支持有以下几个方面：

- 一定容量的内存和外存。
- 页表机制（或段表机制），作为主要的数据结构。
- 中断机构，当用户程序要访问的部分尚未调入内存时，则产生中断。
- 地址变换机构，逻辑地址到物理地址的变换。

请求分页管理方式

是最常用的实现虚拟存储器的方式。基于基本分页系统，增加了请求调页功能和页面置换功能。

页表机制

- 与基本分页管理相比，请求分页管理中，为了实现“请求调页”，操作系统需要知道每个页面是否已经调入内存；如果还没调入，那么也需要知道该页面在外存中存放的位置。
- 当内存空间不够时，要实现“页面置换”，操作系统需要通过某些指标来决定到底换出哪个页面。有的页面没有被修改过，就不用再浪费时间写回外存；有的页面修改过，就需要将外存中的旧数据覆盖。因此，操作系统也需要记录各个页面是否被修改的信息。

其中基本分页存储管理的页表项分为隐藏的页号与内存块号，所以请求分页管理存储的页表项分为：

- 页号（隐藏）。
- 内存块号（物理块号）。
- 状态位P：表示是否已调入内存，0表示未调入，1表示已调入，供访问时参考。
- 访问字段A：可记录最近被访问过几次，或记录上次访问的时间，供置换算法选择换出页面时参考。
- 修改位M：页面调入内存后是否被修改过，0表示没有，1表示修改过。
- 外存地址：页面在外存中的存放地址，一般是物理块号，供调入参考。

缺页中断机构

缺页中断是因为当前执行的指令想要访问的目标页面未调入内存而产生的，因此属于内中断的故障。一条指令在执行期间可能出现多次缺页中断。

40. 在请求分页系统中，每当要访问的页面不在内存时（状态位为0），便产生一个缺页中断，然后由操作系统的缺页中断处理程序处理中断。
41. 此时缺页的进程阻塞，放入阻塞队列。
42. 如果内存中有空闲块，则为进程分配一个空闲块，将所缺页面装入该块，并修改页表中相应的页表项，如内存块号。

43. 如果内存中没有空闲块，则由页面置换算法选择一个页面淘汰，若该页面在内存期间被修改过，则要将其写回外存。未修改过的页面不用写回外存。
44. 调页完成后再将其唤醒，放回就绪队列。

地址变换机构

与普通页表的地址变换机构不同的是，增加了请求调页、页面置换和请求修改内容三个部分。

45. 要算出逻辑地址A对应的页号P与页内偏移量W。页号P=逻辑地址A ÷页面长度L（取除法的整数部分）。页内偏移量W=A逻辑地址%页面长度L（取除法的余数部分）。
46. 检测页号P是否越界。如果页号P大于等于页表长度M，则内中断（因为页号从0开始，页表长度至少为1，从而$P = M$页会越界）。
47. 在快表中查找对应页号，如果找到匹配的页号（即命中），说明要访问的页表项在快表中有副本，则直接从中取出该页对应的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表命中，则访问某个逻辑地址仅需一次访存即可。
48. 快表中有的页面一定是在内存中的。若某个页面被换出外存，则快表中的相应表项也要删除，否则可能访问错误的页面。
49. 如果没有找到匹配的页号，则需要访问内存中的页表。
50. 找到对应页表项后，若对应页面未调入内存，则产生缺页中断，之后由操作系统的缺页中断处理程序进行处理，包括调页与页面置换。
51. 根据页表寄存器中的页表项地址PA=页表起始地址F+页号P×页表项长度PL，得到页表中对应的页表项，从而确定页面存放的内存块号B。
52. 最后物理地址E=内存块号B×页面大小L+页内偏移量W（如果内存块号和业内偏移量用二进制表示，则直接拼接起来就是最终物理地址了）。

简单而言，在具有快表机构的请求分页系统中，访问一个逻辑地址时，若发生缺页，则地址变换步骤是：查快表（未命中）——查慢表（发现未调入内存）——调页（调入的页面对应的表项会直接加入快表）——查快表（命中）——访问目标内存单元。

页面置换算法

假设一个例子，系统为某进程分配了三个内存块，并考虑到有一下页面号引用串（会依次访问这些页面）：7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1。使用算法如何置换？

最佳置换算法

即OPT算法。

每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。这是不可能实际实现的，因为不可能预测本进程所有页面请求，一般用来评价其他算法效果。

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块 1	7	7	7	2		2		2			2			2				7		
内存块 2		0	0	0		0		4			0			0				0		
内存块 3			1	1		3		3			3			1				1		
是否缺页	√	√	√	√		√		√			√			√				√		

所以缺页中断发生了9次，页面置换发生了6次，缺页率$= 9 \div 20 = 45\%$。

先进先出置换算法

即$FIFO$算法。

每次选择淘汰的页面是最早进入内存的页面。只考虑进入时间而不考虑访问次数。

实现方法：把调入内存的页面根据调入的先后顺序排成一个队列，需要换出页面时选择队头页面即可。队列的最大长度取决于系统为进程分配了多少个内存块。

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块 1	7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
内存块 2		0	0	0		3	3	3	2	2	2			1	1			1	0	0
内存块 3			1	1		1	0	0	0	3	3			3	2			2	2	1
是否缺页	√	√	√	√		√	√	√	√	√	√			√	√			√	√	√

缺页了15次，置换了12次，所以缺页率为$15 \div 20 = 75\%$。

如给定一串页面号：3,2,1,0,3,2,4,3,2,1,0,4，使用$FIFO$算法进行置换，会发现分配三个内存块缺页次数为9次，而分配四个内存块缺页次数为10次。

这种当为进程分配的**物理块数**增大时，缺页次数不减反增的异常现象就是 **Belady 异常**。但是如果物理块尺寸增大一倍而块数不变，则在程序顺序执行时缺页中断次数会减少。

只有$FIFO$算法会产生$Belady$异常，使用队列实现，是队列类算法。另外，$FIFO$算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此该算法性能差。

最近最久未使用置换算法

即LRU算法。

每次淘汰的页面是最近最久未使用的页面。（是检查过去的，而OPT是检查未来的）

所以这里就需要使用页表中访问字段这一项，用来记录该页面自上次被访问以来所经历的时间t，当需要淘汰时，就选择t值最大的，即最近最久未使用的页面。

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块 1	7	7	7	2		2		4	4	4	0			1		1		1		
内存块 2		0	0	0		0		0	0	3	3			3		0		0		
内存块 3			1	1		3		3	2	2	2			2		2		7		
是否缺页	√	√	√	√		√		√	√	√	√			√		√		√		

所以缺页中断发生了12次，页面置换发生了9次，缺页率$= 12 \div 20 = 60\%$。

在手动做题时，若需要淘汰页面，可以逆向检查此时在内存中的几个页面号。在逆向扫描过程中最后一个出现的页号就是要淘汰的页面。

该算法的实现要对所有页面进行排序，需要增加额外的LRU位（根本原因），所以需要专门的硬件支持，如寄存器和栈，虽然算法性能好，最接近OPT算法，但是实现困难，开销大。

时钟置换算法

即$CLOCK$算法。是一种性能和开销较均衡的算法，又称最近未用算法NRU。即对LRU的简化，只用管最近未使用即可，不用管最久。

实现方法：

53. 为每个页面设置一个访问位，初始化全部为1。访问位为1，表示最近访问过；访问位为0，表示最近没访问过。
54. 再将内存中的页面都通过链接指针链接成一个循环队列。
55. 当某页被访问时，其访问位置设为1。
56. 当内存满需要淘汰一个页面时，只需检查页的访问位。
57. 如果是0，表示进入内存后一直没有访问过，就选择该页换出。
58. 如果是1，则将它置为0，暂不换出，继续检查下一个页面，若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描。
59. 第二轮扫描中一定会有访问位为0的页面，因此简单的$CLOCK$算法选择一个淘汰页面，最多会经过两轮扫描。

值得注意的是：访问和置换是不同的，扫描指针用于置换，只有缺页中断才会发生指向的变化，而访问和修改数据是另一个指针，会随着访问而不断移动，且是访问指针会影响访问位而不是扫描指针。

假设系统为某进程分配了五个内存块，并考虑到有以下页面号引用串：1,3,4,2,5,6,3,4,7。

- 首先第一步，由于有五个内存块，所以前五个页面全部正常进入，将1,3,4,2,5链接成为循环队列，代表置入内存的内存块，此时访问位全部为1，表示全部被访问并置入内存。
- 然后访问到6，需要置换出一个页面，所以循环访问块链，由于都访问过，所以扫描指针依次扫描1,3,4,2,5，将访问位全部改为0。
- 第二轮循环开始，扫描指针指向第一个的1访问位为0，则将1换出，6换入，访问位置为1，变为6,3,4,2,5，扫描指针指向3。
- 接着访问指针转动，访问3和4，由于都在链中，所以将其访问位都置为1。
- 访问指针指向是7，因为7不在内存中，需要换出页面。
- 扫描指针从上次的3开始扫描，3、4访问位为1，而2访问位为0，所以7换入2换出，7的访问位置为1，变为6,3,4,7,5，扫描指针指向5。
- 访问结束。

改进型时钟置换算法

简单的时钟置换算法仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过，就不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。

因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进型的时钟置换算法的思想。

需要利用到修改位，修改位$= 0$，表示页面没有被修改过；修改位$= 1$，表示页面被修改过。

为方便讨论，用（访问位，修改位）的形式表示各页面状态。如(1,1)表示一个页面近期被访问过，且被修改过。

算法规则：

60. 将所有可能被置换的页面排成一个循环队列。初始的(访问位，修改位)可能是任何的10组合，这和简单时钟算法初始为全0或全1不同。
61. 第一轮（没访问没修改）：从当前位置开始扫描到第一个(0,0)的帧用于替换。本轮扫描不修改任何标志位。
62. 第二轮（没访问有修改）：若第一轮扫描失败，则重新扫描，查找第一个(0,1)的帧用于替换。本轮将所有扫描过的帧访问位设为0。
63. 第三轮（有访问没修改）：若第二轮扫描失败，则重新扫描，查找第一个(0,0)的帧用于替换。本轮扫描不修改任何标志位。

64. 第四轮（有访问有修改）：若第三轮扫描失败，则重新扫描，查找第一个(0,1)的帧用于替换。需要第四轮扫描只有全部页面都被访问都被修改过这一种情况。
65. 由于第二轮已将所有帧的访问位设为0，因此经过第三轮、第四轮扫描一定会有一个帧被选中，因此改进型$CLOCK$置换算法选择一个淘汰页面最多会进行四轮扫描。

- (0,0)：最近没有使用使用也没有修改，最佳状态。
- (0,1)：修改过但最近没有使用，将会被写。
- (1,0)：使用过但没有被修改，下一轮将再次被用。
- (1,1)：使用过也修改过，下一轮页面置换最后的选择。

页面分配策略

页面分配、置换策略

- 驻留集：指请求分页存储管理中给进程分配的物理块的集合（即允许进入内存运行的最大物理块数量）。
- 在采用了虚拟存储技术的系统中，驻留集大小一般小于进程的总大小。
- 若驻留集太小，会导致缺页频繁，系统要花大量的时间来处理缺页，实际用于进程推进的时间很少；驻留集太大，又会导致多道程序并发度下降，资源利用率降低。所以应该选择一个合适的驻留集大小。

页面分配策略：

- 固定分配：操作系统为每个进程分配一组固定数目的物理块，在进程运行期间不再改变。即，驻留集大小不变。
- 可变分配：先为每个进程分配一定数目的物理块，在进程运行期间，可根据情况做适当的增加或减少。即，驻留集大小可变。

页面置换策略：

- 局部置换：发生缺页时只能选进程自己的物理块进行置换。
- 全局置换：可以将操作系统保留的空闲物理块分配给缺页进程，也可以将别的进程持有的物理块置换到外存，再分配给缺页进程。

所以结合页面分配策略与页面置换策略，可以得到：

- 固定分配局部置换：
  - 系统为每个进程分配一定数量的物理块，在整个运行期间都不改变。若进程在运行中发生缺页，则只能从该进程在内存中的页面中选出一页换出，然后再调入需要的页面。
  - 这种策略的缺点是：很难在刚开始就确定应为每个进程分配多少个物理块才算合理。

  - 采用这种策略的系统可以根据进程大小、优先级、或是根据程序员给出的参数来确定为一个进程分配的内存块数。
- 可变分配全局置换：
  - 刚开始会为每个进程分配一定数量的物理块。操作系统会保持一个空闲物理块队列。当某进程发生缺页时，从空闲物理块中取出一块分配给该进程；若已无空闲物理块，则可选择一个未锁定的页面换出外存，再将该物理块分配给缺页的进程。
  - 采用这种策略时，只要某进程发生缺页，都将获得新的物理块，仅当空闲物理块用完时，系统才选择一个未锁定（即可以调出内存）的页面调出。被选择调出的页可能是系统中任何一个进程中的页，因此这个被选中的其他进程拥有的物理块会减少，其他进程缺页率会增加。
- 可变分配局部置换：
  - 刚开始会为每个进程分配一定数量的物理块。当某进程发生缺页时，只允许从该进程自己的物理块中选出一个进行换出外存。
  - 如果进程在运行中频繁地缺页，系统会为该进程多分配几个物理块，直至该进程缺页率趋势适当程度；反之，如果进程在运行中缺页率特别低，则可适当减少分配给该进程的物理块。
- 没有固定分配全局置换策略，因为全局置换意味着进程拥有的物理块数量必然会改变，因此不可能是固定分配。

调入页面

调入页面时机：

- 预调页策略（运行前）：根据局部性原理，一次调入若干个相邻的页面可能比一次调入一个页面更高效。但如果提前调入的页面中大多数都没被访问过，则又是低效的。因此可以预测不久之后可能访问到的页面，将它们预先调入内存，但目前预测成功率只有50%左右。故这种策略主要用于进程的首次调入，由程序员指出应该先调入哪些部分。
- 请求调页策略（运行时）：进程在运行期间发现缺页时才将所缺页面调入内存。由这种策略调入的页面一定会被访问到，但由于每次只能调入一页，而每次调页都要磁盘I/O操作，因此I/O开销较大。

调入页面位置：

- 系统拥有足够的对换区空间：页面的调入、调出都是在内存与对换区之间进行，这样可以保证页面的调入、调出速度很快。在进程运行前，需将进程相关的数据从文件区复制到对换区。
- 系统缺少足够的对换区空间：凡是不会被修改的数据都直接从文件区调入，由于这些页面不会被修改，因此换出时不必写回磁盘，下次需要时再从文件区调入即可。对于可能被修改的部分，换出时需写回磁盘对换区，下次需要时再从对换区调入。

- *UNIX*方式：运行之前进程有关的数据全部放在文件区，故未使用过的页面，都可从文件区调入。若被使用过的页面需要换出，则写回对换区，下次需要时从对换区调入。

抖动（颠簸现象）

- 刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。
- 产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数（分配给进程的物理块不够）。所以需要合适的物理块数量。
- 工作集：指在某段时间间隔里，进程实际访问页面的集合。
- 所有的页面置换算法都可能存在抖动。
- 可以撤销部分进程来减缓抖动。

工作集

- 窗口尺寸就是驻留集的大小，约束工作集大小，工作集大小小于等于窗口尺寸，实际应用中，操作系统可以统计进程的工作集大小，根据工作集大小给进程分配若干内存块。如窗口尺寸为5，经过一段时间的监测发现某进程的工作集最大为3，那么说明该进程有很好的局部性，可以给这个进程分配3个以上的内存块即可满足进程的运行需要。
- 驻留集大小不能小于工作集大小，否则会产生抖动现象。
- 基于局部性原理可知，进程在一段时间内访问的页面与不久之后会访问的页面是有相关性的。因此，可以根据进程近期访问的页面集合（工作集）来设计一种页面置换算法——选择一个不在工作集中的页面进行淘汰。

文件管理

文件系统

文件系统不仅管理普通文件，对于*UNIX*系统所有设备都被视为特殊文件。

文件系统基础

文件基本概念

用户输入输出以文件为基本单位。

- 数据项：最低级数据组织形式。
  - 基本数据项：用于描述一个对象的某种属性的一个值，是数据中可命名的最小逻辑数据单位，即原子数据。
  - 组合数据项：多个基本数据项构成。
- 记录：一组相关数据项的集合，用于表述一个对象在某方面的属性。

- 文件：一组有意义的信息/数据的集合。
- 操作系统以“块”为单位为文件分配存储空间，外存中的数据读入内存同样以块为单位。

文件基本属性

操作系统通过文件控制块FCB来保护文件元数据：

- 文件名：由创建文件的用户决定文件名，主要是为了方便用户找到文件，同一目录下不允许有重名文件。
- 标识符：一个系统内的各文件标识符唯一，对用户来说毫无可读性，因此标识符只是操作系统用于区分各个文件的一种内部名称。
- 类型：指明文件的类型。
- 位置：文件存放的路径（让用户使用）、在外存中的地址（操作系统使用，对用户不可见）。
- 大小：指明文件大小。
- 创建时间。
- 上次修改时间。
- 文件所有者信息。
- 保护信息：对文件进行保护的访问控制信息。

文件控制块

- 目录本身就是一种有结构文件，由一条条记录组成。每条记录对应一个在该放在该目录下的文件。
- 目录文件中的一条记录就是一个**文件控制块**（FCB）。FCB的有序集合就是文件目录，一个FCB就是一个文件目录项。
- FCB中包含了文件的基本信息（文件名、物理地址、逻辑结构、物理结构等），存取控制信息（是否可读/可写、禁止访问的用户名单等），使用信息（如文件的建立时间、修改时间等）。最重要最基本的就是文件名和文件存放物理地址。
- FCB实现了文件名与文件之间的映射，使得用户（用户程序）可以实现按名存取。

索引结点

- 其实在查找各级目录的过程中只需要用到“文件名”这个信息，只有文件名匹配时，才需要读出文件的其他信息。因此可以考虑让目录表“瘦身”来提升效率。
- 所以目录只包含文件名与索引结点指针，除了文件名之外的文件描述信息都存放在索引结点之中。索引结点是对FCB的改进。

- 当找到文件名对应的目录项时，才需要将索引结点调入内存，索引结点中记录了文件的各种信息，包括文件在外存中的存放位置，根据“存放位置”即可找到文件。
- 假设一个FCB是$64B$，磁盘块的大小为$1KB$，则每个盘块中只能存放16个FCB。若一个文件目录中共有640个目录项，则共需要占用$640 \div 16 = 40$个盘块。因此按照某文件名检索该目录，使用折半查找平均需要查询320个目录项，平均需要启动磁盘20次（每次磁盘I/O读入一块)。
- 若使用索引结点机制，文件名占$14B$，索引结点指针站$2B$，则每个盘块可存放64个目录项，那么按文件名检索目录平均只需要读入$320 \div 64 = 5$个磁盘块。显然，这将大大提升文件检索速度。
- “磁盘索引结点”：存放在外存中的索引结点，每个文件有一个唯一的磁盘索引结点。
  - 文件主标识符。
  - 文件类型。
  - 文件存取权限。
  - 文件物理地址。
  - 文件长度。
  - 文件链接计数。
  - 文件存取时间。
- “内存索引结点”：存放在内存中的索引结点，文件打开后将磁盘索引结点复制到内存中。
  - 相比之下内存索引结点中需要增加一些信息：
  - 索引结点编号。
  - 状态。
  - 访问计数。
  - 逻辑设备号。
  - 链接指针。

文件操作

文件基本操作

- 创建$create$。
  1. 文件系统为文件找到空间。
  2. 在目录中为新文件创建条目，记录文件名称、位置和其他信息。
- 删除$delete$：找到对应目录项，使其为空，并回收该文件所占存储空间。
- 截断：允许文件所有属性不变，删除文件内容，即设置长度归零并释放空间。
- 读文件$read$：执行一个系统调用，指明文件名和位置。系统维护一个读位置的指针，读时就更新读指针。需要的参数：

3. 文件描述符。
4. 缓冲区首址。
5. 传输字节数。

- 写文件$write$：执行一个系统调用，指明文件名和内容。系统通过文件名搜索位置。系统需要为文件维护一个写位置的指针，当写时就更新写指针。
- 文件重定位（文件寻址）：按条件搜索目录，将当前文件位置设置为给定值，不读写。

文件打开

- 在很多操作系统中，在对文件进行操作之前，要求用户先使用$open$系统调用“打开文件”，需要提供的几个主要参数：
  6. 文件存放路径。
  7. 文件名。
  8. 要对文件的操作类型（如：r只读；rw读写等）。
- 操作系统在处理$open$系统调用时，主要做了几件事：
  9. 根据文件存放路径找到相应的目录文件，从目录中找到文件名对应的的目录项，并检查该用户是否有指定的操作权限。
  10. 将目录项FCB复制到内存中的“打开文件表”（活跃文件目录表）中。并将对应表目的编号返回给用户。之后用户使用打开文件表的编号来指明要操作的文件。
- 打开文件表分为两种：
  11. 系统打开文件表，只有一张，包括编号、文件名、外存地址、打开计数器（多少个进程打开了次文件）等。
  12. 每一个进程的打开文件表，包括编号、文件名、读写指针、访问权限、系统表索引号等。
- 打开文件时并不会把文件数据直接读入内存，而是提供索引号。“索引号”也称“文件描述符”或“文件句柄”。

每个打开文件都有如下关联信息：

- 文件指针。系统跟踪上次的读写位置作为当前文件位置的指针，这种指针对打开文件的某个进程来说是唯一的，因此必须与磁盘文件属性分开保存。
- 文件打开计数。文件关闭时，操作系统必须重用其打开文件表条目，否则表内空间会不够用。因为多个进程可能打开同一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件。计数器跟踪打开和关闭的数量，计数为0时，系统关闭文件，删除该条目。
- 文件磁盘位置。绝大多数文件操作都要求系统修改文件数据。该信息保存在内存中，以免为每个操作都从磁盘中读取。

- 访问权限。每个进程打开文件都需要有一个访问模式（创建、只读、读写、添加等）。该信息保存在进程的打开文件表中，以便操作系统能够允许或拒绝之后的I/O请求。
- 系统范围的打开文件表，包括每个打开文件的FCB复制和其他信息。
- 单个进程的打开文件表，包括一个指向系统范围内已打开文件表中合适条目和其他信息的指针。

文件关闭

- 用户使用$close$系统调用“打开文件”，需要提供的几个主要参数：
  13. 文件存放路径。
  14. 文件名。
- 操作系统在处理$close$系统调用时，主要做了几件事：
  15. 将进程的打开文件表相应表项删除。
  16. 回收分配给该文件的内存空间等资源。
  17. 系统打开文件表的打开计数器$count$减1，若$count = 0$，则删除对应表项。

文件逻辑结构

分为无结构文件与有结构文件。

- 无结构文件（如文本文件）：
  - 由一些二进制或字符流组成，又称“流式文件”。
  - 以$Byte$为单位。
  - 只能通过穷举进行搜索。
  - 管理简单，适用于字符流的无结构方式，如源程序文件、目标代码文件等。
- 有结构文件（如数据库表）：由一组相似的记录（每条记录又若干个数据项组成，数据项又包含多个属性，是文件系统中最基本的数据单位）组成，又称“记录式文件”。每条记录有一个数据项可作为关键字（如ID）。

有结构文件可以根据各条记录的长度（占用的存储空间）是否相等，可分为定长记录和可变长记录两种。

有结构文件的逻辑结构：

- 顺序文件。
- 索引文件。
- 索引顺序文件。

顺序文件

文件中的记录一个接一个地顺序排列（逻辑上），记录可以是定长的或可变长的。各个记录在物理上可以顺序存储或链式存储。

根据关键字与顺序之间的关系：

- 串结构：记录之间的顺序与关键字无关。一般按记录存入时间决定记录的顺序。
- 顺序结构：记录之间的顺序按关键字顺序排列。

如何访存顺序文件：

- 链式存储：无论是定长/可变长记录，都无法实现随机存取，每次只能从第一记录开始依次往后查找。
- 顺序存储：
  - 可变长记录：无法实现随机存取。每次只能从第一个记录开始依次往后查找。因为需要显式地给出记录长度。
  - 定长记录：
    - 可实现随机存取。记录长度为L，则第i个记录存放的相对位置是$i \times L$。
    - 若采用串结构，只能从头开始查找，无法快速找到某关键字对应的记录。
    - 若采用顺序结构，可以快速找到某关键字对应的记录（如折半查找）。

顺序文件插入删除文件比较麻烦。一般实际实现时会隔一个时间段集中将修改写入到文件中。

索引文件

- 为了解决顺序文件查找问题，建立一张索引表存放每个文件所存放的块盘地址，以加快文件检索速度。每条记录对应一个索引项。
- 记录可以离散存放，而索引表必须连续存放。
- 索引表包含索引号，长度，指针三个部分。其本身是定长长度的顺序文件，所以可以快速找到索引，通过指针随机访问。
- 可将关键字作为索引号内容，若按关键字顺序排列，则还可以支持按照关键字折半查找。
- 每当要增加/删除一个记录时，需要对索引表进行修改。由于索引文件有很快的检索速度，因此主要用于对信息处理的及时性要求比较高的场合。
- 可以使用不同的数据项对一组数据建立多个索引表。
- 索引表其实是索引顺序结构的。

- 对于N条记录的顺序文件，查找关键字记录需要$N/2$次，在索引顺序文件中，将其分为$\sqrt{N}$组，每组$\sqrt{N}$条记录，所以索引表查找需要$\frac{\sqrt{N}}{2}$次，主文件顺序查找也需要$\frac{\sqrt{N}}{2}$次，所以一共需要查找$\sqrt{N}$次。

索引顺序文件

- 因为每个记录对应一个索引表项，因此索引表可能会很大。同时若数据长度本身远小于索引表项长度，则空间利用率会很低。
- 索引顺序文件是索引文件和顺序文件思想的结合。索引顺序文件中，同样会为文件建立一张索引表，但不同的是并不是每个记录对应一个索引表项，而是一组记录对应一个索引表项。
- 每个分组就是一个顺序文件，分组内的记录不需要按关键字排序。
- 索引项页不需要按关键字顺序排列，从而能便利插入删除。
- 索引表其实是索引串结构的。
- 为了进一步提高索引效率，可以建立多级索引表。

散列文件

也称为直接文件。给定记录的键值或通过散列函数转换的键值直接决定记录的物理地址。这种映射结构不同于顺序文件或索引文件，没有顺序的特性。

散列文件有很高的存取速度，但是会引起冲突，即不同关键字的散列函数值相同。

文件物理结构

文件物理结构即文件分配方式，是对非空间磁盘块的管理。

文件块与磁盘块

- 类似于内存分页，磁盘中的存储单元也会被分为一个个“块/磁盘块/物理块”。很多操作系统中，磁盘块的大小与内存块、页面的大小相同。
- 内存与磁盘之间的数据交换（即读/写操作、磁盘I/O）都是以“块”为单位进行的。即每次读入一块，或每次写出一块。
- 在内存管理中，进程的逻辑地址空间被分为一个一个页面。同样的，在外存管理中，为了方便对文件数据的管理，文件的逻辑地址空间也被分为了一个一个的文件“块”。于是文件的逻辑地址也可以表示为（逻辑块号，块内地址）的形式。
- 用户通过逻辑地址来操作自己的文件， 操作系统要负责实现从逻辑地址到物理地址的映射，即文件的物理结构或文件分配方式。

连续分配

- 每个文件在磁盘上占有一组连续的块。
- （逻辑块号，块内地址）→（物理块号，块内地址）。只需转换块号就行，块内地址保持不变。

- 文件目录中记录存放的起始块号和长度（总共占用几个块）。
- 用户给出要访问的逻辑块号，操作系统找到该文件对应的目录项（FCB），物理块号=起始块号＋逻辑块号。
- 优点：
  - 可以直接算出逻辑块号对应的物理块号，因此连续分配支持顺序访问和直接访问（即随机访问）。
  - 连续分配的文件在顺序读写时速度最快。
- 缺点：
  - 不便于拓展。
  - 存储利用率低，产生大量外部碎片。

链接分配

- 采取离散分配的方式，可以为文件分配离散的磁盘块。分为隐式链接和显式链接两种。
- 隐式链接：
  - 类似链表。
  - FCB目录中记录了文件存放的起始块号和结束块号，当然也可以增加一个字段来表示文件的长度。
  - 除了文件的最后个磁盘块之外，每个磁盘块中都会保存指向下一个盘块的指针，这些指针对用户是透明的。
  - 优点：
    - 方便文件拓展。
    - 不会有硬盘碎片。
  - 缺点：
    - 只支持顺序访问，不支持随机访问，查找效率低。
    - 指向下一个盘块的指针也需要耗费少量的存储空间。
    - 存储稳定性不足，若是中间毁坏后面的数据也会丢失。
- 显式链接：
  - 类似静态链表。
  - 把用于链接文件各物理块的指针显式地存放在一张表中。即文件分配表（FAT，$File\ Allocation\ Table$）。FAT包含物理块号和下一块指针两项。
  - FCB目录中只需记录文件的起始块号。
  - 一个磁盘仅设置一张FAT。开机时，将FAT读入内存，并常驻内存。所以地址转换不需要读取内存，从而效率更高。
  - FAT的各个表项在物理上连续存储，且每一个表项长度相同，因此“物理块号”字段可以是隐含的。

- 地址转换：目录项中找到起始块号，若$i > 0$，则查询内存中的文件分配表FAT，往后找到i号逻辑块对应的物理块号。逻辑块号转换成物理块号的过程不需要读磁盘操作。
- 优点：
  - 很方便文件拓展。
  - 不会有外部碎片问题，外存利用率高。
  - 支持随机访问。
  - 相比于隐式链接来说，地址转换时不需要访问磁盘，因此文件的访问效率更高。
- 缺点：文件分配表的需要占用一定的存储空间。

索引分配

- 允许文件离散地分配在各个磁盘块中，系统会为每个文件建立一张索引表，索引表中记录了文件的各个逻辑块对应的物理块（索引表的功能类似于内存管理中的页表——建立逻辑页面到物理页之间的映射关系）。索引表存放的磁盘块称为索引块。文件数据存放的磁盘块称为数据块。
- 索引表包含逻辑块号和物理块号两项。逻辑块号可以是隐含的。
- 每一个文件都有一个索引表。
- 地址转换：从目录项中可知索引表存放位置，将索引表从外存读入内存，并查找索引表即可只i号逻辑块在外存中的存放位置。
- 优点：
  - 很方便文件拓展，不会有碎片问题，外存利用率高。
  - 支持随机访问。
- 缺点：文件索引表的需要占用一定的存储空间。

索引块必须不能太大，但是索引块太小就无法支持大文件，解决方案：

- 链接方案：
  - 分配多个索引块并链接起来。
  - 文件FCB中只需要记录第一个索引块的块号。
  - 缺点：查找效率低。
- 多层索引：
  - 建立类似多级页表的多层索引，使上层索引块指向下一层的索引块。
  - FCB中只需要记录顶层索引块就可以了。
  - 若采用多层索引，则各层索引表大小不能超过一个磁盘块。理由同多级页表。
  - 采用K层索引结构，且顶级索引表未调入内存，则访问一个数据块只需要$K+1$次读磁盘操作。
  - 缺点：即使是小文件页需要$K+1$次读磁盘操作。
- 混合索引：

- 多种索引分配方式的结合。例如，一个文件的顶级索引表中，既包含直接地址索引（直接指向数据块），又包含一级间接索引（指向单层索引表）、还包含两级间接索引（指向两层索引表）等。
- 优点：对于小文件，只需较少的读磁盘次数就可以访问目标数据块。（一般计算机中小文件更多）。

如一共十三个地址项，前十个为直接地址，后面三个为一级间址、二级间址、三级间址。每个盘块的大小为$4KB$，索引块大小为$4B$。

所以文件不大于$4 \times 10 = 40KB$时可以直接读出。一个一级间址块中能包含$4KB \div 4B = 1K = 2^{10}$个索引，所以能索引出$2^{10} \times 4KB = 4MB$的空间。同理，若文件长度大于$4MB + 40KB$时可以使用二级间址块，索引出$4GB$的空间，三级间址能索引出$4TB$的空间。

	访问第 n 条记录	优点	缺点
连续分配	需访问磁盘 1 次	顺序存取时速度快，文件定长时可根据文件起始地址及记录长度进行随机访问	文件存储要求连续的存储空间，会产生碎片，不利于文件的动态扩充
链接分配	需访问磁盘 n 次	可解决外存的碎片问题，提高外存空间的利用率，动态增长较方便	只能按照文件的指针链顺序访问，查找效率低，指针信息存放消耗外存空间
索引分配	m 级需访问磁盘 m+1 次	可以随机访问，文件易于增删	索引表增加存储空间的开销，索引表的查找策略对文件系统效率影响较大

文件保护

口令保护和加密保护是为了防止用户文件被他人存取，而访问控制用于控制用户对文件的访问方式。

口令保护

- 为文件设置一个“口令”，用户请求访问该文件时必须提供“口令”。
- 口令一般存放在文件对应的FCB或索引结点中。用户访问文件前需要先输入“口令”，操作系统会将用户提供的口令与FCB中存储的口令进行对比，如果正确，则允许该用户访问文件。
- 优点：保存口令的空间开销不多，验证口令的时间开销也很小。
- 缺点：正确的“口令”存放在系统内部，不够安全。

加密保护

- 使用某个“密码”对文件进行加密，在访问文件时需要提供正确的“密码”才能对文件进行正确的解密。

- 系统并不保存原始数据，而是加密过的数据。
- 加密方式有异或加密等。
- 优点：保密性强，不需要在系统中存储“密码”。
- 缺点：编码/译码，或者说加密/解密要花费一定时间。

访问控制

- 在每个文件的FCB（或索引结点）中增加一个访问控制列表（$Access Control\ List$，ACL），该表中记录了各个用户可以对该文件执行哪些操作。
- 访问类型包括：
  - 读：从文件中读数据。
  - 写：向文件中写数据。
  - 执行：将文件装入内存并执行。
  - 添加：将新信息添加到文件结尾部。
  - 删除：删除文件，释放空间。
  - 列表清单：列出文件名和文件属性。
- 优点：可以使用复杂的访问方法。
- 缺点：长度无法余集且可能导致复杂的空间管理。
- 有的计算机可能会有很多个用户，因此访问控制列表可能会很大，可以用精简的访问列表解决这个问题。
- 精简的访问列表：以“组”为单位，标记各“组”用户可以对文件执行哪些操作。当某用户想要访问文件时，系统会检查该用户所属的分组是否有相应的访问权限。所以系统也需要管理分组的信息。
- 用户类型：
  - 拥有者：创建文件用户。
  - 组：一组需要共享文件且拥有类似访问的用户。
  - 其他：系统内其他所有用户。
- 若想要让某个用户能够获取某种权限，需要把该用户放入有该权限的分组即可。
- 如果对某个目录进行了访问权限的控制，那也要对目录下的所有文件进行相同的访问权限控制。

Linux 文件权限

命令格式 `chmod` 权限 文件名。

一共九位，从左至右分为三组，$1-3$位数字代表文件所有者的权限，$4-6$位数字代表同组用户的权限，$7-9$数字代表其他用户的权限。每一组的第一位表示可读，第二位表示可写，第三位表示可执行。

如755，第一组的$7=4+2+1$表示所有者可读可写可执行，$5=4+0+1$表示同组用户和其他用户可读可执行但是不可写。

目录系统

目录结构

单级目录结构

整个系统中只建立一张目录表，每个文件占一个目录项。

- 实现了“按名存取”，但是不允许文件重名。
- 在创建一个文件时，需要先检查目录表中有没有重名文件，确定不重名后才能允许建立文件，并将新文件对应的目录项插入目录表中。
- 不适用于多用户操作系统。

两级目录结构

分为主文件目录（MFD，$Master\ File\ Directory$）和用户文件目录（UFD，$User\ Flie\ Directory$）：

- 主文件目录记录用户名及相应用户文件目录的存放位置。
- 用户文件目录由该用户的文件FCB组成。
- 允许不同用户的文件重名。文件名虽然相同，但是对应的其实是不同的文件。
- 两级目录结构允许不同用户的文件重名，也可以在目录上实现实现访问限制（检查此时登录的用户名是否匹配）。
- 两级目录结构依然缺乏灵活性，用户不能对自己的文件进行分类。

多级目录结构

即树形目录结构：

- 不同目录下的文件可以重名。
- 用户（或用户进程）要访问某个文件时要用文件路径名标识文件，文件路径名是个字符串。各级目录之间用“/”隔开。
- 从根目录出发的路径称为绝对路径。
- 很多时候，用户会连续访问同一目录内的多个文件，显然，每次都从根目录开始查找，是很低效的。因此可以设置一个“当前目录”出发的相对路径。
- 树形目录结构可以很方便地对文件进行分类，层次结构清晰，也能够更有效地进行文件的管理和保护。
- 但是，树形结构不便于实现文件的共享，且查找文件需要逐级访问影响速度。为此，提出了“无环图目录结构”。

无环图目录结构

- 在树形目录结构的基础上，增加一些指向同一节点的有向边，使整个目录成为一个有向无环图。可以更方便地实现多个用户间的文件共享。
- 可以用不同的文件名指向同一个文件，甚至可以指向同一个目录（共享同一目录下的所有内容）。
- 需要为每个共享结点设置一个共享计数器，用于记录此时有多少个地方在共享该结点。用户提出删除结点的请求时，只是删除该用户的FCB、并使共享计数器减1，并不会直接删除共享结点。
- 共享文件不同于复制文件。在共享文件中，由于各用户指向的是同一个文件，因此只要其中一个用户修改了文件数据，那么所有用户都可以看到文件数据的变化。
- 实现了文件共享，但是使得文件管理更复杂。

目录操作

目录基本操作

- 搜索：首先要找到对应目录项。
- 创建文件。
- 删除文件。
- 创建目录。
- 删除目录：包括不删除非空目录和可删除非空目录两种。
- 移动目录。
- 显示目录。
- 修改目录。

目录实现

- 线性列表。
- 哈希表。

文件目录共享

多个用户共享同一个文件或目录，意味着系统中只有“一份”文件数据。并且只要某个用户修改了该文件的数据，其他用户也可以看到文件数据的变化。

文件共享方式分为：

- 基于索引结点的共享方式：硬链接。
- 基于符号链的共享方式：软链接。

索引结点中设置一个链接计数变量$count$，用于表示链接到本索引结点上的用户目录项数。

硬链接

- 在文件目录中提到，索引结点，是一种文件目录瘦身策略。由于检索文件时只需用到文件名，因此可以将除了文件名之外的其他信息放到索引结点中。这样目录项就只需要包含文件名、索引结点指针。
- 不同目录下对于同一个文件的索引结点的命名可以是不同的。
- 创建硬链接时$count + 1$。

若某个用户决定“删除”该文件：

- 则只是要把用户目录中与该文件对应的目录项删除，且索引结点的$count$值减1。
- 若$count > 0$，说明还有别的用户要使用该文件，暂时不能把文件数据删除，否则会导致指针悬空。
- 若$count = 0$，则系统需要删除文件。

软链接

- 软链接就是共享时建立一个$Link$类型的文件，文件记录了要共享的文件的存放路径或任意一条硬链接路径，类似$Windows$系统的快捷方式。
- 文件拥有者才拥有指向其索引结点的指针，而共享该文件的其他用户只有该文件的路径名。
- 当访问共享文件时，先判断这个文件属于$Link$类型文件，然后根据其中记录的路径层层查找路径找到索引结点。
- 创建软链接时$count$直接复制。

若某个用户决定“删除”该文件：

- 由于删除操作对软链接不可见，所以$count$值不变。
- 当以后通过符号链接再次访问时发现文件不存在再直接删除软链接。

优点：

- 网络共享只用提供文件所在机器的网络地址和文件路径。
- 若共享文件被删除了，则软链接失效。删除软链接则无影响。

缺点：

- 当共享文件删除，而其他用户新建一个相同路径的文件则原链接指向的是新文件而不是原文件。
- 因为软链接访问共享文件时需要查询多层目录，所以有多层I/O操作，从而软链接访问速度慢于硬链接。

硬链接和软链接都是静态共享方式，都存在一个共同的问题，每个共享文件都有几个文件名，从而每增加一条文件名，当遍历整个文件系统时会多次遍历到该共享文件。

多个进程同时对同一个文件操作称为动态共享。

文件系统管理

系统层次结构

66. 设备。
67. 相关模块
    - 辅助分配模块：管理辅存空间。即负责分配和回收存储空间。
    - 设备管理模块：管理设备。直接与硬件交互，负责和硬件直接相关的一些管理工作。如分配设备、分配设备缓冲区、磁盘调度、启动设备、释放设备等。
68. 物理文件系统：物理地址转换。这一层需要把上一层提供的文件逻辑地址转换为实际的物理地址。
69. 逻辑文件系统与文件信息缓冲区：逻辑地址转换。用户指明想要访问文件记录号，这一层需要将记录号转换为对应的逻辑地址。文件信息缓冲区用来在调入索引表到内存时暂存索引表的内容。
70. 存取控制模块：文件保护。为了保证文件数据的安全，还需要验证用户是否有访问权限。
71. 文件目录系统：管理文件目录。用户是通过文件路径来访问文件的，因此这一层需要根据用户给出的文件路径找到相应的FCB或索引结点。所有和目录、目录项相关的管理工作都在本层完成，如管理活跃的文件目录表、管理打开文件表等。
72. 用户接口：向用户提供文件目录相关功能接口。这层就是用于处理用户发出的系统调用请求（$read$、$write$、$open$、$close$等系统调用）。
73. 用户/应用程序。

假设某用户请求删除文件“D:/工作目录/学生信息.$xlsx$”的最后100条记录：

74. 用户需要通过操作系统提供的接口发出上述请求——用户接口。
75. 由于用户提供的是文件的存放路径，因此需要操作系统一层一层地查找目录，找到对应的目录项——文件目录系统
76. 不同的用户对文件有不同的操作权限，因此为了保证安全，需要检查用户是否有访问权限—-存取控制模块（存取控制验证层）。
77. 验证了用户的访问权限之后，需要把用户提供的“记录号”转变为对应的逻辑地址——逻辑文件系统与文件信息缓冲区。
78. 知道了目标记录对应的逻辑地址后，还需要转换成实际的物理地址——物理文件系统。
79. 要删除这条记录，必定要对磁盘设备发出请求——设备管理程序模块。
80. 删除这些记录后，会有一些盘块空闲，因此要将这些空闲盘块回收——辅助分配模块。

外存空闲空间管理

包含文件系统的分区称为卷。

同一个卷中存放数据的空间（文件区）和FCB的空间（目录区）是分离的。

存储空间划分与初始化

- 将物理磁盘划分为一个个文件卷（逻辑卷、逻辑盘）。
- 将各个文件卷初始化为：
  - 目录区：主要存放文件目录信息FCB、用于磁盘存储空间管理的信息。
  - 文件区：用于存放普通文件数据。
- 有的系统支持超大型文件，支持由多格物理磁盘组成一个文件卷。

空闲表法

是连续分配方式。

- 拥有一个空闲盘块表，包括空闲区间的起始位置（第一个空闲盘块号）和空闲空间长度（空闲盘块数）。
- 适用于连续分配方式。
- 如何分配磁盘块：与内存管理中的动态分区分配很类似，为一个文件分配连续的存储空间。同样可采用首次适应、最佳适应、最坏适应等算法来决定要为文件分配哪个区间。
- 如何回收磁盘块：与内存管理中的动态分区分配很类似，当回收某个存储区时需要有四种情况：
  1. 回收区的前后都没有相邻空闲区。
  2. 回收区的前后都是空闲区。
  3. 回收区前面是空闲区.
  4. 回收区后面是空闲区。
  5. 总之，回收时需要注意表项的合并问题。

空闲链表法

- 空闲盘块法：
  - 以盘块为单位组成一条空闲链。空闲盘块中存储着下一个空闲盘块的指针。
  - 操作系统保存着链头、链尾指针。
  - 如何分配：若某文件申请K个盘块，则从链头开始依次摘下K个盘块分配，并修改空闲链的链头指针。
  - 如何回收：回收的盘块依次挂到链尾，并修改空闲链的链尾指针。
  - 适用于离散分配的物理结构。为文件分配多个盘块时可能要重复多次操作。

- 空闲盘区法：
  - 以盘区为单位组成一条空闲链。由连续的几个盘块组成一个盘区。每一个盘区内的第一个盘块内记录了盘区的长度和下一个盘区的指针。
  - 操作系统保存着链头、链尾指针。
  - 如何分配：若某文件申请K个盘块，则可以采用首次适应、最佳适应等算法，从链头开始检索，按照算法规则找到一个大小符合要求的空闲盘区，分配给文件。若没有合适的连续空闲块，也可以将不同盘区的盘块同时分配给一个文件，注意分配后可能要修改相应的链指针、盘区大小等数据。
  - 如何回收：若回收区和某个空闲盘区相邻，则需要将回收区合并到空闲盘区中。若回收区没有和任何空闲区相邻，将回收区作为单独的一个空闲盘区挂到链尾。
  - 离散分配、连续分配都适用。为一个文件分配多个盘块时效率更高。

位示图法

- 每个二进制位对应一个盘块。如“0”代表盘块空闲，“1”代表盘块已分配。
- 位示图一般用连续的“字”来表示，如一个字的字长是16位，则一共有16列，字中的每一位对应一个盘块。因此可以用（字号，位号）对应一个盘块号。当然有的题目中也描述为（行号，列号）。
- 若盘块号、字号、位号从0开始，若n表示字长，则(字号,位号)=(i,j)的二进制位对应的盘块号$b=ni+j$。
- b号盘块对应的字号$i=b\div n$，位号$j=b\%n$。
- 如何分配：若文件需要K个块，
  6. 顺序扫描位示图，找到K个相邻或不相邻的“0”。
  7. 根据字号、位号算出对应的盘块号，将相应盘块分配给文件。
  8. 将相应位设置为“1”。
- 如何回收：
  9. 根据回收的盘块号计算出对应的字号、位号。
  10. 将相应二进制位设为“0”。
- 离散分配、连续分配都适用。

成组链接法

- 空闲表法、空闲链表法不适用于大型文件系统，因为空闲表或空闲链表可能过大。$UNIX$系统中采用了成组链接法对磁盘空闲块进行管理。
- 文件卷的目录区中专门用一个磁盘块作为**超级块**，当系统启动时需要将超级块读入内存。并且要保证内存与外存中的超级块数据一致。
- 将空闲块分成若干组，如每100个空闲块为一组，每组的第一空闲块登记了下一组空闲块的物理盘块号和空闲块总数。组内索引，组间链接。
- 一个分组的块号不需要连续。

- 如何分配：
    11. 检测第一个分组的块数是否足够。
    12. 若足够则分配空闲块并修改对应数据。
    13. 若刚好相等或不足，则分配时要先将其数据复制到超级块中，超级块充当链头的作用。
- 如何回收：
    14. 如果块内回收后余留，则修改数据。
    15. 若回收数量大于等于余下，需要将超级块中的数据复制到新回收的块中，并修改超级块的内容，让新回收的块成为第一个分组。

文件系统分布

文件系统存放在磁盘上，多数磁盘划分分区，每个分区都有独立的文件系统。

文件系统结构

- 主引导记录（*Master Boot Record*，*MBR*），位于磁盘的 0 号扇区，用来引导计算机，*MBR*后面是分区表，该表给出每个分区的起始和结束地址。表中的一个分区被标记为活动分区，当计算机启动时，BIOS 读入并执行*MBR*。*MBR*做的第一件事是确定活动分区，读入它的第一块，即引导块。
- 引导块（*boot block*），*MBR*执行引导块中的程序后，该程序负责启动该分区中的操作系统。为统一起见，每个分区都从一个引导块开始，即使它不含有一个可启动的操作系统，也不排除以后会在该分区安装一个操作系统。*Windows*系统称之为分区引导扇区。除了从引导块开始，磁盘分区的布局是随着文件系统的不同而变化的。
- 超级块（*super block*），包含文件系统的所有关键信息，在计算机启动时，或者在该文件系统首次使用时，超级块会被载入内存。超级块中的典型信息包括分区的块的数量、块的大小、空闲块的数量和指针、空闲的*FCB*数量和*FCB*指针等。
- 文件系统中空闲块的信息，可以使用位示图或指针链接的形式给出。后面也许跟的是一组 i 结点，每个文件对应一个结点，i 结点说明了文件的方方面面。接着可能是根目录，它存放文件系统目录树的根部。最后，磁盘的其他部分存放了其他所有的目录和文件。

文件系统内存结构

内存中的信息用于管理文件系统并通过缓存来提高性能。

- 内存中的安装表（*mount table*），包含每个已安装文件系统分区的有关信息。
- 内存中的目录结构的缓存包含最近访问目录的信息。对安装分区的目录，它可以包括一个指向分区表的指针。
- 整个系统的打开文件表，包含每个打开文件的*FCB*副本及其他信息。

- 每个进程的打开文件表，包含一个指向整个系统的打开文件表中的适当条目的指针与其他信息。

虚拟文件系统

即VFS，为用户程序提供了文件系统操作的统一接口（统一调用函数）屏蔽了不同文件系统的差异。

$Linux$实现VFS提供四个对象：

- 超级块对象：表示已安装或挂载的特定文件系统。
- 索引结点对象：表示一个特定的文件。
- 目录项对象：表示特定的目录项。
- 文件对象：表示一个与进程相关的已打开文件。

VFS可以提高系统性能，只存在内存中，系统启动建立，关闭时消亡。

设备管理

I/O 概述

I/O 设备基本概念

I/O 设备定义

- “I/O”就是“输入/输出”（$Input/Output$）。I/O设备就是可以将数据输入到计算机，或者可以接收计算机输出数据的外部设备，属于计算机中的硬件部件。
- $UNIX$系统将外部设备抽象为一种特殊的文件，用户可以使用与文件操作相同的方式对外部设备进行操作。
- 计算机系统为每台设备确定一个编号以便区分和识别设备，这个确定的编号称为设备的**绝对号**。

I/O 设备分类

- 按使用特性分类：
  - 人机交互类外部设备：数据传输速度慢，如鼠标、键盘。
  - 存储设备：数据传输速度块，如移动硬盘。
  - 网络通信设备：数据传输速度介于二者之间，如调制解调器。
- 按传输速率分类：
  - 低速设备：鼠标、键盘。
  - 中速设备：打印机。

  - 高速设备：硬盘。
- 按信息交换的单位分类：
  - 块设备：有结构设备，传输速率较高，可寻址，即对它可随机地读/写任一块，如硬盘、磁盘。
  - 字符设备：无结构设备，传输速率较慢，不可寻址，在输入/输出时常采用中断驱动方式，键盘、鼠标。

I/O 控制器

- I/O设备由机械部件和电子部件（I/O控制器或设备控制器）组成。
- I/O设备的机械部件主要用来执行具体I/O操作。如鼠标/键盘的按钮，显示器的LED屏，移动硬盘的磁臂、磁盘盘面。
- I/O设备的电子部件通常是一块插入主板扩充槽的印刷电路板。
- CPU无法直接控制I/O设备的机械部件，因此I/O设备还要有一个电子部件作为CPU和I/O设备机械部件之间的“中介”，用于实现CPU对设备的控制。

I/O 控制器功能

- 接受和识别CPU发出的命令：如CPU发来的$Read/Write$命令，I/O控制器中会有相应的**控制寄存器**来存放命令和参数。
- 向CPU报告设备的状态：I/O控制器中会有相应的**状态寄存器**，用于记录I/O设备的当前状态。如1表示空闲，0表示忙碌。
- 数据交换：I/O控制器中会设置相应的**数据寄存器**。输出时，数据寄存器用于暂存CPU发来的数据，之后再由控制器传送设备。输入时，数据寄存器用于暂存设备发来的数据，之后CPU从数据寄存器中取走数据。
- 地址识别：类似于内存的地址，为了区分设备控制器中的各个寄存器，也需要给各个寄存器设置一个特定的“地址”。I/O控制器通过CPU提供的“地址”来判断CPU要读/写的是哪个寄存器。

I/O 接口

负责与CPU和设备进行通信：

- CPU与控制器的接口：用于实现CPU与控制器之间的通信。CPU通过控制线发出命令，通过地址线指明要操作的设备，通过数据线来取出（输入）数据,或放入（输出）数据。
- I/O逻辑：负责接收和识别CPU的各种命令（如地址译码），并负责对设备发出命令。
- 控制器与设备的接口：用于实现控制器与设备之间的通信，包括数据、状态和控制。
- 一个I/O控制器可能会对应多个设备。

I/O 端口

- 数据寄存器。

- 控制寄存器。
- 状态寄存器：可能有多个（如每个控制/状态寄存器对应一个具体的设备），且这些寄存器都要有相应的地址，才能方便CPU操作。

实现CPU与I/O端口通信，使用不同编址方式：

- 统一编址：有的计算机会让这些寄存器占用内存地址的一部分，称为**内存映像I/O**，优点是简化了指令。可以采用对内存进行操作的指令来对控制器进行操作。
- 独立编址：另一些计算机则采用I/O专用地址，即**寄存器独立编址**，缺点是需要设置专门的指令来实现对控制器的操作，不仅要指明寄存器的地址，还要指明控制器的编号。

I/O 控制方式

程序直接控制方式

- 完成读写的流程：
    16. CPU向控制器发出读指令。于是设备启动，并且状态寄存器设为1（未就绪）。
    17. 轮询检查控制器的状态，其实就是在不断地执行程序的循环，若状态位一直是1，说明设备还没准备好要输入的数据，于是CPU会不断地轮询。
    18. 输入设备准备好数据后将数据传给控制器，并报告自身状态。
    19. 控制器将输入的数据放到数据寄存器中，并将状态改为0（已就绪）。
    20. CPU发现程序已就绪，即可将数据寄存器中的内容读入CPU的寄存器中，再把CPU寄存器中的内容放入内存。
    21. 若还要继续读入数据，则CPU继续发出读指令。
- CPU干预的频率：很频繁，I/O操作开始之前、完成之后需要CPU介入，并且在等待I/O完成的过程中CPU需要不断地轮询检查。
- 数据传送的单位：每次读/写一个字。
- 数据的流向：
    - 读操作（数据输入）：I/O设备→CPU→内存（指的是CPU的寄存器）。
    - 写操作（数据输出）：内存→CPU→I/O设备。
    - 每个字的读/写都需要CPU的帮助。
- 主要缺点和主要优点：
    - 优点：实现简单，在读/写指令之后，加上实现循环检查的一系列指令即可，因此才称为“程序直接控制方式”。
    - 缺点：

- CPU和I/O设备只能串行工作。
- CPU需要一直轮询检查，长期处于“忙等”状态，CPU利用率低。

中断驱动方式

- 引入中断机制。由于I/O设备速度很慢，因此在CPU发出读/写命令后，可将等待I/O的进程阻塞，先切换到别的进程执行。当I/O完成后，控制器会向CPU发出一个中断信号，CPU检测到中断信号后，会保存当前进程的运行环境信息，转去执行中断处理程序处理该中断。处理中断的过程中，CPU从I/O控制器读一个字的数据传送到CPU寄存器，再写入主存。接着，CPU恢复等待I/O的进程（或其他进程）的运行环境，然后继续执行。
  - CPU会在每个指令周期的末尾检查中断。
  - 中断处理过程中需要保存、恢复进程的运行环境，这个过程是需要一定时间开销的。可见，如果中断发生的频率太高，也会降低系统性能。
- CPU干预的频率：每次I/O操作开始之前、完成之后需要CPU介入。等待I/O完成的过程中CPU可以切换到别的进程执行。
- 数据传送的单位：每次读/写一个字。
- 数据的流向：
  - 读操作（数据输入）:I/O设备→CPU→内存（指的是CPU的寄存器）。
  - 写操作（数据输出）:内存→CPU→I/O设备。
- 主要缺点和主要优点：
  - 优点：
    - 与“程序直接控制方式”相比，在“中断驱动方式”中，I/O控制器会通过中断信号主动报告I/O已完成，CPU不再需要不停地轮询。
    - CPU和I/O设备可并行工作，CPU利用率得到明显提升。
  - 缺点：
    - 每个字在I/O设备与内存之间的传输，都需要经过CPU。
    - 频繁的中断处理会消耗较多的CPU时间。

DMA 方式

- 即直接存储器存取方式，主要用于块设备的I/O控制。
- 改进方面：
  - 数据的传送单位是“块”。不再是一个字、一个字的传送。
  - 数据的流向是从I/O设备直接放入**内存**，或者从内存直接到设备。不再需要CPU中转。
  - 仅在传送一个或多个数据块的开始和结束时，才需要CPU干预。

- 完成读写的流程：
  22. CPU向I/O模块发出读或写块的命令。CPU指明此次要进行的操作，如读操作说明要读入多少数据、数据要存放在内存的什么位置、数据在外部设备上的地址。
  23. CPU转向其他工作，DMA控制器根据CPU所给参数完成工作。
  24. 完成工作后DMA控制器向CPU发送一个中断信号。CPU处理中断。
- DMA控制器结构：与I/O控制器结构类似：
  - 主机控制器接口：
    - DR（$Data\ Register$，数据寄存器）：暂存从设备到内存，或从内存到设备的数据。
    - MAR（$Memory\ Address\ Register$，内存地址寄存器）：在输入时，$MAR$表示数据应放到内存中的什么位置，输出时$MAR$表示要输出的数据放在内存中的什么位置。
    - DC（$Data\ Counter$，数据计数器）：表示剩余要读/写的字节数。
    - CR（$Command\ Register$，命令/状态寄存器）：用于存放CPU发来的I/O命令，或设备的状态信息。
  - I/O控制逻辑：用于实现设备控制。
  - 块设备控制器接口。
- CPU干预的频率：仅在传送一个或多个数据块的**开始和结束**时，才需要CPU干预。
- 数据传送的单位：每次读/写一个或多个块（注意每次读写的只能是连续的多个块，且这些块读入内存后在内存中也必须是连续的）。
- 数据的流向（不再需要经过CPU）：
  - 读操作（数据输入）:I/O设备→内存。
  - 写操作（数据输出）:内存→I/O设备。
- 主要缺点和主要优点：
  - 优点：
    - 数据传输以“块”为单位，CPU介入频率进一步降低。
    - 数据的传输不再需要先经过CPU再写入内存，数据传输效率进一步增加。
    - CPU和I/O设备的并行性得到提升。
  - 缺点：
    - CPU每发出一条I/O指令，只能读/写一个或多个连续的数据块。
    - 如果要读/写多个离散存储的数据块，或者要将数据分别写到不同的内存区域时，CPU要分别发出多条I/O指令，进行多次中断处理才能完成。

DMA的控制器与CPU分时使用内存，通常采用以下三种方法：停止CPU访内存、周期挪用、DMA与CPU交替访内存。（计算机组成原理会具体讲到）

- 数据块传送方式：在I/O接口电路中设置一个比较大的数据缓冲区，一般能存放一个数据块，I/O接口电路与内存之间的数据交换以数据块为单位。总线仲裁器判定究竟是DMA控制器还是CPU能获得总线的使用权。
- 周期挪用方式：当I/O接口没有DMA请求时，CPU按程序要求访问内存；一旦I/O接口有DMA请求，则I/O接口挪用一个或几个周期。缺点是：数据输入或输出过程中实际占用了CPU时间。
- 交替访存方式：CPU与DMA控制器交替访问内存。不需要总线使用权的申请、建立和归还过程。 效率高，但实现起来有困难，基本上不被使用。

通道控制方式

- 通道：一种硬件，可以理解为是简版的CPU，因为与CPU相比，通道可以执行的指令很单一，并且通道程序是放在主机内存中的，也就是说通道与CPU共享内存。
- 通道可以识别并执行一系列通道指令。
- 通道控制设备控制器、设备控制器控制设备工作。
- 字节多路通道：用于连接大量低俗或中速设备。它通常含有许多非分配型子通道，其数量可达几十到几百个，每个通道连接一台I/O设备，并控制该设备的I/O操作。这些子通道按时间片轮转方式共享主通道。各个通道循环使用主通道，各个通道每次完成其I/O设备的一个字节的交换，然后让出主通道的使用权。这样，只要字节多路通道扫描每个子通道的速率足够快，而连接到子通道上的设备的速率不太高时，便不至于丢失信息。
- 完成读写的流程：
    25. CPU向通道发出I/O指令。指明通道程序在内存中的位置，并指明要操作的是哪个I/O设备。之后CPU就切换到其他进程执行。启动时无论成功与否都需要回答CPU。
    26. 通道执行内存中的通道程序（其中指明了要读入/写出多少数据，读/写的数据应放在内存的什么位置等信息），类似任务清单。
    27. 通道执行完规定的任务后，此时向CPU发出中断信号，之后CPU对中断进行处理。
- CPU干预的频率：极低，通道会根据CPU的指示执行相应的通道程序，只有完成一组数据块的读/写后才需要发出中断信号，请求CPU干预。
- 数据传送的单位：每次读/写一组数据块。
- 数据的流向（(在通道的控制下进行）：
    - 读操作（数据输入）:I/O设备→内存。
    - 写操作（数据输出）:内存→I/O设备。
- 主要缺点和主要优点：
    - 优点：CPU、通道、I/O设备可并行工作，资源利用率很高。

- 缺点：实现复杂，需要专门的通道硬件支持。

方式区别

DMA方式与中断控制方式的区别：

- 中断控制方式在每个数据传送完成后中断CPU，而DMA控制方式则在所要求传送的一批数据全部传送结束时中断CPU。
- 中断控制方式的数据传送在中断处理时由CPU控制完成，而DMA控制方式则在DMA控制器的控制下完成。不过，在DMA控制方式中，数据传送的方向、存放数据的内存始址及传送数据的长度等仍然由CPU 控制。
- DMA方式以存储器为核心，中断控制方式以CPU为核心。因此DMA方式能与CPU并行工作。DMA方式传输批量的数据，中断控制方式的传输则以字节为单位。

DMA方式与通道方式的区别：

- 在DMA控制方式中，在DMA控制器控制下设备和主存之间可以成批地进行数据交换而不用CPU干预，这样既减轻了CPU的负担，又大大提高了I/O数据传送的速度。
- 通道控制方式与DMA控制方式类似，也是一种以内存为中心实现设备与内存直接交换数据的控制方式。
- 不过在通道控制方式中，CPU只需发出启动指令，指出通道相应的操作和I/O设备，该指令就可以启动通道并使通道从内存中调出相应的通道程序执行。
- 与DMA控制方式相比，通道控制方式所需的CPU干预更少，并且一个通道可以控制多台设备，进一步减轻了CPU的负担。
- 另外，对通道来说，可以使用一些指令灵活改变通道程序，这一点DMA控制方式无法做到。

I/O 软件层次结构

I/O软件结构层次：

81. 中断处理程序。
82. 设备驱动程序。
83. 设备独立性软件。
84. 用户层软件。

每一层会利用其下层提供的服务，实现某些功能，并屏蔽实现的具体细节，向高层提供服务（“封装思想”）。

其中中断处理程序、设备驱动程序、设备独立性软件属于操作系统的内核部分，即I/O系统或I/O核心子系统，需要使用核心态进行运行，而用户层软件用户态就可以运行。

用户层软件

- 用户层软件实现了与用户交互的接口，用户可直接使用该层提供的、与I/O操作相关的库函数对设备进行操作。
- 用户层软件将用户请求翻译成格式化的I/O请求，并通过“系统调用”请求操作系统内核的服务。

设备独立性软件

- 设备独立性软件，又称设备无关性软件。与设备的硬件特性无关的功能几乎都在这一层实现。
- 主要实现的功能：
    1. 向上层提供统一的调用接口（如$Read/Write$系统调用）。
    2. 设备保护。原理类似与文件保护。设备被看做是一种特殊的文件，不同用户对各个文件的访问权限是不一样的，同理，对设备的访问权限也不一样。
    3. 差错处理。
    4. 设备的分配与回收。
    5. 数据缓冲区管理。可以通过缓冲技术屏蔽设备之间数据交换单位大小和传输速度的差异。
    6. 建立逻辑设备名到物理设备名的映射关系，根据设备类型选择调用相应的驱动程序。（逻辑设备名）
        - 用户或用户层软件发出I/O操作相关系统调用的系统调用时，需要指明此次要操作的I/O设备的逻辑设备名（如去学校打印店打印时，需要选择扣印机1/打印机2/打印机3，其实这些都是逻辑设备名）。
        - 设备独立性软件需要通过“逻辑设备表”（LUT，$Logical\ UnitTable$）来确定逻辑设备对应的物理设备，并找到该设备对应的设备驱动程序。
        - 操作系统系统可以采用两种方式管理逻辑设备表（LUT）：
            1. 整个系统只设置一张LUT，这就意味着所有用户不能使用相同的逻辑设备名，因此这种方式只适用于单用户操作系统。
            2. 为每个用户设置一张LUT，各个用户使用的逻辑设备名可以重复，适用于多用户操作系统。系统会在用户登录时为其建立一个用户管理进程，而LUT就存放在用户管理进程的PCB中。

设备驱动程序

- 主要负责对硬件设备的具体控制，将上层发出的一系列命令（如$Read/Write$）转化成特定设备“能听得懂”的一系列操作。包括设置设备寄存器、检查设备状态等。
- 不同的I/O设备有不同的硬件特性，具体细节只有设备的厂家才知道，因此厂家需要根据设备的硬件特性设计并提供相应的驱动程序。
- 驱动程序一般会以一个独立进程的方式存在。

中断处理程序

- 当I/O任务完成时，I/O控制器会发送一个中断信号，系统会根据中断信号类型找到相应的中断处理程序并执行。中断处理程序的处理流程如下：
    7. 从控制器读出设备状态。
    8. 如果正常结束则从设备中读入一个字的数据并由CPU放入内存缓冲区中。交给上一层处理。
    9. 如果异常结束则根据异常原语进行处理。

应用程序 I/O 接口

即I/O系统与高层软件之间的接口

- 字符设备接口。
- 块设备接口。
- 网络设备接口。
- 阻塞或非阻塞I/O。

I/O 核心子系统

I/O 核心子系统概述

实现功能：

- 用户层软件：假脱机技术。
- 设备独立性软件：
    - I/O调度：用某种算法确定一个好的顺序来处理各个I/O请求。
    - 设备保护：将设备看作文件，具有FCB。
    - 设备分配与回收。
    - 缓冲区管理（即缓冲与高速缓存）。

缓冲区管理

磁盘高速缓存

利用内存中的存储空间暂存从磁盘中读出的一系列盘块中的信息，逻辑上是属于磁盘，物理上驻留在内存中。

分为两种形式：在内存中开辟出一个单独的存储空间作为磁盘高速缓存，大小固定；另一种把未利用的内存空间作为缓冲池，供请求分页系统和磁盘I/O时共享。

缓冲区概念

- 缓冲区是一个存储区域，可以由专门的硬件寄存器组成，也可利用内存作为缓冲区。
- 使用硬件作为缓冲区的成本较高，容量也较小，一般仅用在对速度要求非常高的场合（如存储器管理中所用的联想寄存器，由于对页表的访问频率极高，因此使用速度很快的联想寄存器来存放页表项的副本）。
- 一般情况下，更多的是利用内存作为缓冲区，“设备独立性软件”的缓冲区管理就是要组织管理好这些缓冲区。
- 缓冲区作用：
  - 缓和CPU与I/O设备之间速度不匹配的矛盾。
  - 减少对CPU的中断频率，放宽对CPU中断相应时间的限制。
  - 解决数据粒度不匹配的问题。
  - 提高CPU与I/O设备之间的并行性。

单缓冲

- 假设某用户进程请求某种块设备读入若干块的数据。若采用单缓冲的策略，操作系统会在主存中为其分配一个缓冲区（若题目中没有特别说明，一个缓冲区的大小就是一个块）。
- 当缓冲区数据非空时，不能往缓冲区冲入数据，只能从缓冲区把数据传出；当缓冲区为空时，可以往缓冲区冲入数据，但必须把缓冲区充满以后，才能从缓冲区把数据传出。
- 常考题型：计算每处理一块数据平均需要多久？技巧：假定一个初始状态（如工作区满、缓冲区空），分析下次到达相同状态需要多少时间，这就是处理一块数据平均所需时间。
- 处理一块数据平均耗时$\max(C,T)+M$。其中C代表处理时间、M代表传输时间、T代表输入时间。

双缓冲

- 假设某用户进程请求某种块设备读入若干块的数据。若采用双缓冲的策略，操作系统会在主存中为其分配两个缓冲区（若题目中没有特别说明，一个缓冲区的大小就是一个块）。

- 一块存，一块可以取，处理一块数据平均耗时$\max(T,(C+M))$。其中C代表处理时间、M代表传输时间、T代表输入时间。
- 若两个相互通信的机器设置双缓冲区，则同一时刻可以实现双向的数据传输。否则单缓冲区只能单向数据传输。

循环缓冲

- 将多个大小相等的缓冲区链接成一个循环队列。
- in指针，指向下一个可以冲入数据的空缓冲区。
- out指针，指向下一个可以取出数据的满缓冲区。

缓冲池

- 缓冲池由系统中共用的缓冲区组成。这些缓冲区按使用状况可以分为：
  - 空缓冲队列、
  - 装满输入数据的缓冲队列（输入队列）。
  - 装满输出数据的缓冲队列（输出队列）。
- 根据一个缓冲区在实际运算中扮演的功能不同，又设置了四种工作缓冲区：
  - 用于收容输入数据的工作缓冲区（hin）。
  - 用于提取输入数据的工作缓冲区（sin）。
  - 用于收容输出数据的工作缓冲区（$hout$）。
  - 用于提取输出数据的工作缓冲区（$sout$）。
- 输入进程请求输入数据：从空缓冲队列中取出一块作为收容输入数据的工作缓冲区（hin）。冲满数据后将缓冲区挂到输入队列队尾。
- 计算进程想要一块输入数据：从输入队列中取得一块冲满输入数据的缓冲区作为提取输入数据的工作缓冲区（cin）。缓冲区读空后挂到空缓冲区队列。
- 计算进程将准备好的数据冲入缓冲区：从空缓冲队列中取出一块作为收容输出数据的工作缓冲区（$hout$）。数据冲满后将缓冲区挂到输出队列队尾。
- 输出进程请求输出数据：从输出队列中取得一块冲满输出数据的缓冲区作为提取输出数据的工作缓冲区（$sout$）。缓冲区读空后挂到空缓冲区队列。

	高速缓存	缓冲区
存放数据	存放的是低速设备上的某些数据的复制数据，即高速缓存上有的，低速设备上面必然有	存放的是低速设备传递给高速设备的数据（或相反），而这些数据在低速设备（或高速设备）上却不一定有备份，这些数据再从缓冲区传送到高速设备（或低速设备）
目的	高速缓存存放的是高速设备经常要访问的数据，若高速设备要访问的数据不在高速缓存	高速设备和低速设备的通信都要经过缓冲区，高速设备永远不会直接去访问低速设备

高速缓存	缓冲区
中，则高速设备就需要访问低速设备	

设备分配回收

设备分配方法：

- 静态分配：主要是独占设备的分配，在用户作业开始前，由系统一次性分配，直到作业完成才会取消占用，不会死锁。
- 动态分配：主要是共享设备的分配，进程执行期间进行，通过系统调用提出设备请求，可能死锁。
- $SPOOLing$技术：主要是虚拟设备的分配，实现了虚拟设备，让设备可同时被分配给多个进程。

设备分配考虑因素

- 固有属性：
  - 独占设备：同时只能被一个进程访问。如打印机、磁带机。
  - 共享设备：同时能被多个进程访问，不会死锁。如磁盘。可寻址和可随机访问。
  - 虚拟设备：将独占设备改造成共享的虚拟设备。
- 分配算法：
  - 先请求先分配。
  - 优先级高者优先。
- 安全性：
  - 安全分配方式：
    - 为进程分配一个设备后就将进程阻塞，本次I/O完成后才将进程唤醒。一个时段内每个进程只能使用一个设备。
    - 优点：破坏了“请求和保持”条件，不会死锁。
    - 缺点：对于一个进程来说，CPU和I/O设备只能串行工作。
  - 不安全分配方式：
    - 进程发出I/O请求后，系统为其分配I/O设备，进程可继续执行，之后还可以发出新的I/O请求。只有某个I/O请求得不到满足时才将进程阻塞。一个进程可以同时使用多个设备。
    - 优点：进程的计算任务和I/O任务可以并行处理，使进程迅速推进。
    - 缺点：有可能发生死锁（死锁避免、死锁的检测和解除）。
- 独立性。

设备分配数据结构

- 一个通道可控制多个设备控制器，每个设备控制器可控制多个设备。
- 设备控制表（DCT），系统为每个设备配置一张DCT，用于记录设备情况：
  - 设备类型：如打印机/扫描仪/键盘。
  - 设备标识符：即物理设备名，系统中的每个设备的物理设备名唯一。
  - 设备状态：忙碌/空闲/故障……。
  - 指向控制器表的指针：每个设备由一个控制器控制，该指针可找到相应控制器的信息。
  - 重复执行次数或时间：当重复执行多次I/O操作后仍不成功，才认为此次I/O失败。
  - 设备队列的队首指针：指向正在等待该设备的进程队列（由进程PCB组成队列）。
- 控制器控制表（$COCT$），每个设备控制器都会对应一张$COCT$。操作系统根据$COCT$的信息对控制器进行操作和管理，$CHCT$和$COCT$是一对多的关系：
  - 控制器标识符：各个控制器的唯一ID。
  - 控制器状态：忙碌/空闲/故障……
  - 指向通道表的指针：每个控制器由一个通道控制，该指针可找到相应通道的信息。
  - 控制器队列的队首指针：指向正在等待该控制器的进程队列（由进程PCB组成队列）。
  - 控制器队列的队尾指针：指向控制器的队尾。
- 通道控制表（$CHCT$），每个通道都会对应一张$CHCT$。操作系统根据$CHCT$的信息对通道进行操作和管理：
  - 通道标识符：各个通道的唯一 ID。
  - 通道状态：忙碌/空闲/故障……
  - 与通道连接的控制器表首址：可通过该指针找到该通道管理的所有控制器相关信息（$COCT$）。
  - 通道队列的队首指针：指向正在等待该通道的进程队列（由进程PCB组成队列）。
  - 通道队列的队尾指针：指向通道队列的队尾。
- 系统设备表（SDT），整个系统只有一张，记录了系统中全部设备的情况，每个设备对应一个表目：
  - 设备类型：打印机/扫描仪/键盘。
  - 设备标识符：物理设备名。
  - DCT（设备控制表）。
  - 驱动程序入口：对应设备驱动的程序地址。

设备名映射

为了提高分配灵活性，使用了设备独立性，使用逻辑设备名来使用设备。

所以系统中设置了一张逻辑设备表LUT。包括设备逻辑名、物理逻辑名、设备驱动程序入口。

有两种方式设置LUT：

- 系统只有一张，所以只适合单用户系统。
- 每个用户一张，当用户登录，系统就为用户创建一个进程并建立一张LUT并放入PCB中。

优点：

- 逻辑设备表（LUT）建立了逻辑设备名与物理设备名之间的映射关系。
- 某用户进程第一次使用设备时使用逻辑设备名向操作系统发出请求，操作系统根据用户进程指定的设备类型（逻辑设备名）查找系统设备表，找到一个空闲设备分配给进程，并在LUT中增加相应表项。
- 如果之后用户进程再次通过相同的逻辑设备名请求使用设备，则操作系统通过LUT表即可知道用户进程实际要使用的是哪个物理设备了，并且也能知道该设备的驱动程序入口地址。

设备分配过程

85. 根据进程请求的物理设备名查找SDT（物理设备名是进程请求分配设备时提供的参数）。
86. 根据SDT找到DCT，若设备忙碌则将进程PCB挂到设备等待队列中，不忙碌则将设备分配给进程。
87. 根据DCT找到$COCT$，若控制器忙碌则将进程PCB挂到控制器等待队列中，不忙碌则将控制器分配给进程。
88. 根据$COCT$找到$CHCT$，若通道忙碌则将进程PCB挂到通道等待队列中，不忙碌则将通道分配给进程。
89. 只有设备、控制器、通道三者都分配成功时，这次设备分配才算成功，之后便可后动I/O设备进行数据传送。

缺点:

90. 用户编程时必须使用“物理设备名”，底层细节对用户不透明，不方便编程。
91. 若换了一个物理设备，则程序无法运行。
92. 若进程请求的物理设备正在忙碌，则即使系统中还有同类型的设备，进程也必须阻塞等待这个名字设备。

改进方法：建立逻辑设备名与物理设备名的映射机制，用户编程时只需提供逻辑设备名。

93. 根据进程请求的逻辑设备名查找SDT（用户编程时提供的逻辑设备名其实就是“设备类型”）。
94. 查找SDT，找到用户进程指定类型的、并且空闲的设备，将其分配给该进程。操作系统在逻辑设备表（LUT）中新增一个表项。
95. 根据DCT找到$COCT$，若控制器忙碌则将进程PCB挂到控制器等待队列中，不忙碌则将控制器分配给进程。
96. 根据$COCT$找到$CHCT$，若通道忙碌则将进程PCB挂到通道等待队列中，不忙碌则将通道分配给进程。

假脱机技术

也称为$SPOOLing$技术。

脱机技术

- 脱机处理是一种计算机技术，是指在不受主机控制的外部设备上进行数据处理，或与实时控制系统、主机不直接相连的数据处理。常用于主机速度不高的数据处理中提高设备的利用率。
- 如输入输出设备速度慢，远小于CPU处理速率，而CPU要处理就必须等待输入输出设备，导致CPU资源浪费，而脱机技术能让数据更快进出CPU，从而速度加快。

假脱机技术实现

- “假脱机技术”，又称“$SPOOLing$技术”是用软件的方式模拟脱机技术。
- $SPOOLing$系统组成：
  - 输入井：在磁盘上，磁盘固定区域，模拟脱机输入时的磁带，用于收容I/O设备输入的数据。
  - 输出井：在磁盘上，磁盘固定区域，模拟脱机输出时的磁带，用于收容用户进程输出的数据。
  - 输入进程：在内存中，模拟脱机输入时的外围控制机。
  - 输出进程，在内存中，模拟脱机输出时的外围控制机。
  - 输入缓冲区与输出缓冲区：在内存中，用于暂存输入的数据转存到输入井中，或暂存输出的数据转存到输出井中。
- 由预输入程序、井管理程序、缓输出程序管理。
- 用户进程实际分配的是外存区，即虚拟设备。
- 必要条件：
  - 大容量、高速度的外存作为输入井和输出井。
  - $SPOOLing$软件。
- 由于需要对设备进行虚拟，所以需要独占设备。

共享打印机

打印机是一种独占式设备，而通过假脱机技术变成共享设备。

当多个用户进程提出输出打印的请求时，系统会答应它们的请求，但是并不是真正把打印机分配给他们，而是由假脱机管理进程处理：

97. 在磁盘输出井中为进程申请一个空闲缓冲区（也就是说，这个缓冲区是在磁盘上的），并将要打印的数据送入其中。
98. 为用户进程申请一张空白的打印请求表，并将用户的打印请求填入表中（其实就是用来说明用户的打印数据存放位置等信息的），再将该表挂到假脱机文件队列（打印任务队列）上。
99. 当打印机空闲时，输出进程会从文件队列的队头取出一张打印请求表，并根据表中的要求将要打印的数据从输出井传送到输出缓冲区，再输出到打印机进行打印。用这种方式可依次处理完全部的打印任务。

磁盘系统

磁盘概念

磁盘结构

- 磁盘由空气过滤片、主轴、音圈马达、永磁铁、磁盘、磁头、磁头臂组成。
- 磁盘的表面由一些磁性物质组成，可以用这些磁性物质来记录二进制数据。
- 磁盘由多个盘片构成，盘片保存数据的一面就是盘面。一个盘片有一个或两个盘面。
- 磁盘的盘面被划分成一个个磁道。这样的一个“圈”就是一个磁道。
- 一个磁道又被划分成一个个扇区，每个扇区就是一个“磁盘块”。各个扇区存放的数据量相同。
- 最内侧磁道上的扇区面积最小，因此数据密度最大。硬盘的存储能力也受限于最内侧的最大记录密度。
- 磁头用于读写盘面数据，需要把磁头移动到想要读/写的扇区所在的磁道。磁盘会转起来，让目标扇区从磁头下面划过，才能完成对扇区的读/写操作。
- 每个盘面对应一个磁头。
- 所以磁头都是链接在一个磁臂上，所有磁头的移动方向都是一致的。
- 所有盘面中相对位置相同的磁道组成柱面。
- 可用（柱面号，盘面号，扇区号）来定位任意一个磁盘块。
- 地址读取方式：
    1. 根据“柱面号”移动磁臂，让磁头指向指定柱面。
    2. 激活指定盘面对应的磁头。

3. 磁盘旋转的过程中，指定的扇区会从磁头下面划过，这样就完成了对指定扇区的读/写。

- 许多操作系统位改善磁盘访问时间，不以扇区为单位，而是以**簇**为单位近空间分配。

磁盘类别

磁盘的分类：

- 磁头可以移动的称为活动头磁盘。磁臂可以来回伸缩来带动磁头定位磁道。
- 磁头不可移动的称为固定头磁盘。这种磁盘中每个磁道有一个磁头。
- 盘片可以更换的称为可换盘磁品。
- 盘片不可更换的称为固定盘磁品。

固态硬盘SSD基于闪存技术，是一种容量更大的U盘。

- 由一个或多个闪存芯片和闪存翻译层构成。
- 一个闪存由B块组成，每块由P页组成。存取以页为单位，一般每页$512B \sim 4KB$，每块$32 \sim 128$页，每块$16KB \sim 512KB$。
- 闪存磨损速度很快，为了弥补寿命缺陷其磨损均衡技术分为两种：
  - 动态磨损均衡：自动选择较新块写入。
  - 静态磨损均衡：SSD自动检测并数据分配。

磁盘操作时间

- 寻找时间（寻道时间）$T_s = s + m \times n$：在读/写数据前，将磁头移动到指定磁道所花的时间：
  - 启动磁头臂是需要时间的。假设耗时为s。
  - 移动磁头也是需要时间的。假设磁头匀速移动，每跨越一个磁道耗时为m，总共需要跨越n条磁道。
- 延迟时间$T_r = \frac{1}{2} \times \frac{1}{r} = \frac{1}{2r}$：通过旋转磁盘，使磁头定位到目标扇区所需要的时间。
  - 磁盘转速为r（单位：转/秒，或转/分）。
  - $\frac{1}{r}$就是转一圈需要的时间。
  - 找到目标扇区平均需要转半圈，因此再乘以$\frac{1}{2}$。
- 传输时间$T_t = \frac{b}{rN}$：从磁盘读出或向磁盘写入数据所经历的时间。
  - 假设磁盘转速为r，此次读/写的字节数为b，每个磁道上的字节数为N。

  - 每个磁道要可存N字节的数据，因此b字节的数据需要$\frac{b}{N}$个磁道才能存储。而读/写一个磁道所需的时间刚好又是转一圈所需要的时间$\frac{1}{r}$，$T_t = \frac{1}{r} \times \frac{b}{N}$。
- 操作时间=寻道时间T_s+延迟时间T_r+传输时间T_t。
- 延迟时间和传输时间都与磁盘转速相关，且为线性相关。而转速是硬件的固有属性，因此操作系统也无法优化延迟时间和传输时间，操作系统只能通过磁盘调度算法优化寻道时间。

磁盘调度算法

磁盘调度算法用来优化寻道时间。

假设某磁盘的磁道为0 ~ 200号，磁头的初始位置是100号磁道，此时磁头正在往磁道号增大的方向，有多个进程先后陆续地请求访问55、58、39、18、90、160、150、38、184号磁道。

先来先服务算法

- 即$FCFS$算法，根据进程请求访问磁盘的先后顺序进行调度。
- 优点：
  - 公平。
  - 如果请求访问的磁道比较集中的话，算法性能还算过的去。
- 缺点：如果有大量进程竞争使用磁盘，请求访问的磁道很分散，则$FCFS$在性能上很差，寻道时间长。

按照$FCFS$的规则，移动顺序为55、58、39、18、90、160、150、38、184号磁道。一共移动了498个磁道。平均寻找长度为55.3。

最短寻找时间优先算法

- 即$SSTF$算法，会优先处理的磁道是与当前磁头最近的磁道。可以保证每次的寻道时间最短，但是并不能保证总的寻道时间最短。（其实就是贪心算法的思想，只是选择眼前最优，但是总体未必最优）。
- 优点：性能较好，平均寻道时间短。
- 缺点：
  - 可能产生“饥饿”现象。产生饥饿的原因在磁头可能在一个小区域内来回来去地移动。
  - 对于全局而言未必最优。

按照$SSTF$的规则，首先需要将号码排序，100在90和150之间，离90更近，所以向更小方向移动，移动顺序为90、58、55、39、38、18、150、160、184。一共移动了248个磁道。平均寻找长度为27.5。

扫描算法

- 也叫电梯算法，即$SCAN$算法，规定只有磁头移动到最外侧磁道的时候才能往内移动，移动到最内侧磁道的时候才能往移动。
- 优点：
  - 性能较好，平均寻道时间较短。
  - 不会产生饥饿现象。
- 缺点：
  - 只有到达最边上的磁道时才能改变磁头移动方向，事实上，处理了最大和最小的磁道的访问请求之后就不需要再往两边移动磁头了。
  - 对于各个位置磁道的响应频率不平均。（如假设此时磁头正在往右移动，且刚处理过90号磁道，那么下次处理90号磁道的请求就需要等磁头移动很长一段距离，而响应了184号磁道的请求之后，很快又可以再次响应184号磁道的请求了）。

对于例题，按照$SCAN$规则，因为磁头向增大的方向移动，所以应该最开始访问150号，而$SCAN$规定只有到了最边上的磁道才能改变磁头移动方向，所以移动到184号后还要移动到200号才能往小方向移动磁头，移动顺序为150、160、184、200、90、58、55、39、38、18。一共移动了282个磁道，平均寻找长度为31.3。

LOOK 调度算法

- 为了解决必须移动到两边磁道的缺点，$LOOK$规定如果在磁头移动方向上已经没有别的请求，就可以立即改变磁头移动方向。
- 优点：比起$SCAN$算法来,不需要每次都移动到最外侧或最内侧才改变磁头方向，使寻道时间进一步缩短。

对于例题，按照$LOOK$规则，到了184就可以立刻回头，所以移动顺序为150、160、184、90、58、55、39、38、18。一共移动了250个磁道，平均寻找长度为27.5。

循环扫描算法

- 即$C-SCAN$算法，为了解决每个位置磁道的响应频率不平均，规定只有磁头朝某个特定方向移动时才处理磁道访问请求，而返回时直接快速移动至起始端而不处理任何请求。
- 优点：比起$SCAN$来，对于各个位置磁道的响应频率很平均。
- 缺点：
  - 平均寻道时间更长。
  - 只有到达最边上的磁道时才能改变磁头移动方向，事实上，处理了184号磁道的访问请求之后就不需要再往右移动磁头了。
  - 磁头返回时其实只需要返回到18号磁道即可，不需要返回到最边缘的磁道。

对于例题，按照$C-SCAN$规则，先访问150号，然后移动到200达到边缘，立刻返回到0的位置不处理，从0开始向右扫描，所以移动顺序为150、160、184、200、0、18、38、39、55、58、90。一共移动了390个磁道，平均寻找长度为43.3。

C-LOOK 调度算法

- 为了解决必须移动到两边磁道的缺点，$C-LOOK$基于$C-SCAN$，规定如果在磁头移动方向上已经没有别的请求，就可以立即改变磁头移动方向，且磁头只用返回到有磁道访问请求的位置即可。
- 优点：比起$C-SCAN$算法来，不需要每次都移动到最外侧或最内侧才改变磁头方向，使寻道时间进一步缩短.

对于例题，按照$C-LOOK$规则，先访问150号，然后移动到184，立刻返回到最靠左的18号，开始向右扫描，所以移动顺序为150、160、184、18、38、39、55、58、90。一共移动了322个磁道，平均寻找长度为35.8。

延迟时间处理

磁头在读取一块内容后需要一段时间处理，由于盘片不断旋转，从而在处理本块内容时，回错过相邻扇区的处理，从而只能等待下次再旋转到此扇区进行读取，所以如果逻辑上相邻的扇区在物理上也相邻，则读入几个连续的逻辑扇区，可能需要很长的延迟时间。

磁盘地址结构设计

- 为什么磁盘的物理地址是（柱面号，盘面号，扇区号）而不是（盘面号，柱面号，扇区号）。
- 读取地址连续的磁盘块时，采用（柱面号，盘面号，扇区号）的地址结构可以减少磁头移动消耗的时间。
- （柱面号，盘面号，扇区号）若要连续读取物理地址(000,00,000) ~ (000,01,111)的扇区,读取完(000,00,000) ~ (000,00,111)由于柱面号/磁道号相同，只是盘面号不同，因此不需要移动磁头臂。只需要激活相邻盘面的磁头即可。
- 若物理地址结构是（盘面号，柱面号，扇区号），且需要连续读取物理地址(00,000,000) ~ (00,001,111)的扇区，则(00,000,000) ~ (00,000,111)转两圈可读完，之后再读取物理地址相邻的区域，即(00,001,000) ~ (00,001,111)，需要启动磁头臂，将磁头移动到下一个磁道，花费时间更多。

交替编号

让逻辑上相邻的扇区在物理上有一定的间隔，可以使读取连续的逻辑扇区所需要的延迟时间更小。

错位命名

若相邻的盘面相对位置相同处扇区编号相同，则会出现不能连续读取的问题。所以使用错位命名，同一柱面不同盘面的扇区编号不同，从而有足够的时间来处理。

磁盘的管理

磁盘初始化

100. 进行低级格式化（物理格式化），将磁盘的各个磁道划分为扇区。一个扇区通常可分为头、数据区域（如512B大小）、尾三个部分组成。管理扇区所需要的各种数据结构一般存放在头、尾两个部分，包括扇区校验码（如奇偶校验、CRC循环冗余校验码等，校验码用于校验扇区中的数据是否发生错误）。
101. 将磁盘分区，每个分区由若干柱面组成（即分为我们熟悉的C盘、D盘、E盘）。将每个分区的起始扇区和大小都记录在磁盘主引导记录MBR的分区表中。
102. 进行逻辑格式化，创建文件系统。包括创建文件系统的根目录、初始化存储空间管理所用的数据结构（如位示图、空闲分区表）。其中操作系统将相邻的多个扇区组合称为簇（$Linux$为块），文件内容占用空间为簇的整数倍。

引导块

- 计算机开机时需要进行一系列初始化的工作，这些初始化工作是通过执行初始化程序（自举程序）完成的。
- 初始化程序可以放在ROM（只读存储器）中。ROM中的数据在出厂时就写入了，并且以后不能再修改。
- 一般ROM因为无法修改所以只会存放很小的自举装入程序。
- 完整的自举程序放在磁盘的启动块（即引导块/启动分区）上，启动块位于磁盘的固定位置。
- 开机时计算机先运行自举装入程序出通过执行该程序就可找到引导块，并将完整的自举程序读入内存，完成初始化。
- 拥有启动分区的磁盘称为启动磁盘或系统磁盘（C盘）。
- 引导控制块记录系统从该分区引导操作系统所需要的信息，若没有操作系统这块内容为空，一般为分区的第一块。UFS为引导块，$NTFS$为分区引导扇区。
- 分区控制块包括分区详细信息。UFS为超级块，而$NTFS$称为主控文件表。

坏块管理

- 坏了、无法正常使用的扇区就是“坏块”。这属于硬件故障，操作系统是无法修复的。应该将坏块标记出来，以免错误地使用到它。

- 对于简单的磁盘，可以在逻辑格式化时（建立文件系统时）对整个磁盘进行坏块检查，标明哪些扇区是坏扇区，比如在FAT表上标明。在这种方式中，坏块对操作系统不透明。
- 对于复杂的磁盘，磁盘控制器（磁盘设备内部的一个硬件部件）会维护一个坏块链表。
- 在磁盘出厂前进行低级格式化（物理格式化）时就将坏块链进行初始化。
- 操作系统会保留一些“备用扇区”，用于替换坏块。这种方案称为扇区备用。且这种处理方式中，坏块对操作系统透明。