# 操作系统

## 一、操作系统概述

### 1. 基本概念

计算机系统自下而上可粗分为四个部分：硬件、操作系统、应用程序和用户。操作系统是指控制和管理整个计算机系统的硬件和软件资源，并合理地组织调度计算机的工作和资源的分配，以提供给用户和其它软件方便的接口和环境的程序集合。

#### 操作系统的基本特征

操作系统的基本特征包括并发、共享、虚拟和异步。最基本的特征是并发和共享。

- 并发是指两个或多个事件在**同一时间间隔内**发生（并行是指多个事件在同一时刻发生），操作系统的并发性是指计算机系统中同时存在多个运行着的程序。引入进程的目的是使程序能够并发执行，但微观上这些程序还是分时地交替执行，也即操作系统的并发性是通过分时得以实现的。
- 共享即资源共享，可分为互斥共享方式和同时访问方式。互斥共享如打印机，一段时间内允许一个进程访问该资源。同时共享是宏观上的同时，微观上仍是分时共享。
- 虚拟是指将一个物理实体变为若干个逻辑上的对应物。操作系统利用了多种虚拟技术，分别来实现虚拟处理器、虚拟内存和虚拟外部设备。可以归纳为两种：时分复用技术和空分复用技术。
- 异步是指进程的执行不是一贯到底，而是走走停停。

#### 操作系统的功能

操作系统应具备以下几个方面的功能：处理机管理（进程管理）、存储器管理、文件管理和设备管理，还必须向用户提供**接口**，以及**扩充机器**（隐藏硬件）。

- 处理机管理的主要功能有：进程控制、进程同步、进程通信、死锁处理、处理机调度等。
- 存储器管理的主要功能有：内存分配、地址映射、内存保护与共享、内存扩充等。
- 文件管理的主要功能有：文件存储空间的管理、目录管理、文件读写管理和保护等。
- 设备管理的主要功能有：缓冲管理、设备分配、设备处理、虚拟设备等。

操作系统提供的接口有两类：命令接口、程序接口。

- 命令接口：用户利用命令接口来组织和控制作业的执行。有两种控制方式：联机控制方式、脱机控制方式，对应地有联机控制接口、脱机控制接口。
    - 联机控制接口又称交互式命令接口，用于分时或实时系统，具备和用户交互的能力。
    - 脱机控制接口又称批处理命令接口，用于批处理系统，作业运行时用户不能干预。
- 程序接口：程序接口由一组系统调用命令组成，编程人员可以使用这些系统调用命令来请求操作系统为其提供服务。**系统调用命令简称系统调用**。如trap（用户态转入内核态）、fork（创建进程）、execve（执行进程）、exit（结束进程）、read、write（读写文件）、mkdir、rmdir（目录）等等。

操作系统将裸机改造成功能更强、使用更方便的机器。通常把覆盖了软件的机器称为**扩充机器**，又称为虚拟机。

### 2.操作系统的发展和分类

操作系统的发展历程如下：

手工操作阶段→脱机处理→早期批处理（单道批处理）→多道批处理→分时操作系统→实时操作系统→网络操作系统→分布式操作系统→个人计算机操作系统。

此外还有嵌入式操作系统、服务器操作系统、多处理器操作系统等。

1. 手工操作阶段：所有的工作都需要人工干预，此阶段无操作系统。
2. 单道批处理系统：系统对作业的处理是成批进行的，但内存中始终只保持一道作业。主要特征有：自主性（自动装入下一道）、顺序性、单道性。

   单道批处理系统下，高速的CPU需要等待低速的I/O设备，资源利用率低。
3. 多道批处理系统：允许多个程序同时进入内存并交替在CPU中运行，从而尽量让系统处于忙的状态。主要特征有：多道、宏观上并行、微观上串行。

   多道批处理系统需要进行多个程序之间的切换和各种资源的调度，设计和实现都复杂很多。
4. 分时操作系统：把处理器的运行时间分成很短的时间片，轮流分配给各联机作业使用。主要特征有：同时性（或称多路性，允许多个终端同时使用）、交互性、独立性（用户之间独立）、及时性（用户请求在很短时间内获得响应）。
5. 实时操作系统：为了在某个时间限制内完成某项紧急任务，产生了实时操作系统。分为硬实时系统和软实时系统，硬实时系统如飞机的控制系统、导弹制导系统等，指令必须实时执行；软实时系统允许偶尔违反时间限制，如飞机订票系统、银行管理系统等。
6. 网络操作系统和分布式操作系统：网络操作系统实现计算机之间的数据传送，分布式操作系统则是协同若干台计算机完成同一任务。
7. 个人计算机操作系统：Windows、Linux和MacOS等。

# 3.操作系统的运行环境

操作系统划分了用户态（目态）和核心态（管态），核心态时运行操作系统内核程序，用户态时运行用户自编程序或系统外层程序，一些特权指令被禁止，如置中断指令、送程序状态字到程序状态字寄存器等。

用户态转向核心态的例子：

- 系统调用
- 中断
- 用户程序产生一个错误状态
- 用户程序企图执行一条特权指令

从核心态转向用户态由一条指令实现，该指令也是特权指令，一般是中断返回指令。而用户态转向核心态时用到的**访管指令**（自动放弃CPU的使用权），不是特权指令。

操作系统**内核**包含一些与硬件关联紧密的模块、运行频率较高的程序（进程管理、存储器管理和设备管理等），这部分内容都工作在核心态。核心态指令实际上包括系统调用指令和一些针对对时钟、中断、原语的操作指令。

## 系统调用

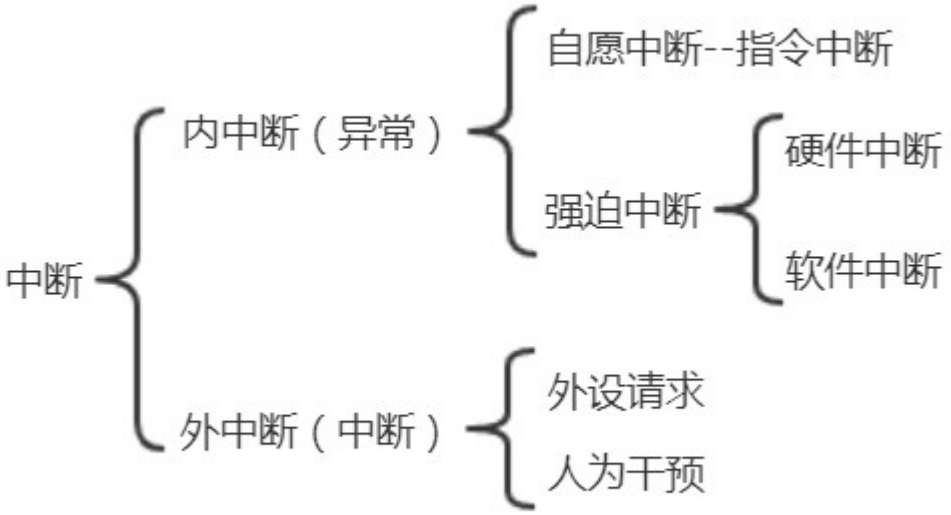
在用户程序中，凡是与资源有关的操作，都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成。系统调用都运行在核心态。系统调用按功能可分为：设备管理、文件管理、进程管理、进程通信、内存管理。

## 时钟管理

时钟的功能：计时、通过时钟中断实现进程切换。在分时操作系统中，采用时间片轮转调度来实现进程切换；实时系统中，按截止时间控制运行来实现进程切换；皮处理系统中，通过时钟管理来衡量一个作业的运行程度。

### 中断机制

现代操作系统是靠中断驱动的软件。CPU由用户态进入核心态的唯一途径是通过中断和异常实现。异常又称为内中断、例外或陷入（trap）。




### 原语

原语是内核的组成部分，用于系统中的设备驱动、CPU切换、进程通信等。原语程序不可中断，直接方法是关中断。

## 4. 大内核与微内核

随着体系结构和应用需求的不断发展，操作系统内核越来越大，内核代码维护难度增加，提出了微内核的体系结构。微内核有效分离了内核与服务、服务与服务，维护的代价降低。但微内核需要频繁切换核心态和用户态，增加了操作系统的开销。

# 二、进程管理

## 1. 进程

### 进程的概念

多道程序系统中，为使多个程序并发运行时不失封闭性，引入了进程的概念。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独立单位。**

系统为描述进程的基本情况和运行状态，控制和管理进程，设置了一个专门的数据结构：进程控制块PCB。创建进程实际是创建PCB，撤销进程实际是撤销PCB，PCB是进程存在的唯一标志。由程序段、数据段和PCB三部分构成进程映像（进程实体），进程映像是静态的，进程是动态的。

### 进程的特征

- 动态性：进程是程序的一次执行，具有一定的生命周期。动态性是进程最基本的特征。
- 并发性：引入进程的目的就是为了使程序之间并发执行，以提高资源利用率。
- 独立性：进程是一个能独立运行、独立获得资源和独立接受调度的基本单位。
- 异步性：进程具有执行的间断性。
- 结构性：进程实体由程序段、数据段和PCB三部分组成。

### 进程的状态与转换

- 运行状态：进程在处理机上运行。在单处理机环境下，每一时刻最多只有一个进程处于运行状态。
- 就绪状态：进程已获得处理机之外的一切资源，处于准备运行的状态，一旦得到处理机即可运行。
- 阻塞状态：又称等待状态，进程在等待某资源可用或等待输入输出完成，而暂停运行。
- 创建状态：进程正在被创建。首先申请一个空白的PCB，并向PCB写入一些控制和管理信息；然后系统为该进程分配运行时所需的资源；最后该进程转入就绪状态。
- 结束状态：系统首先置该进程为结束状态，然后再进行资源释放和回收等工作。

进程状态之间的一些转换如下：

- 就绪状态→运行状态：就绪状态的进程获得处理机资源，进入运行。
- 运行状态→阻塞状态：进程以系统调用的形式请求操作系统提高服务，如I/O操作，这是一个主动的行为。
- 运行状态→就绪状态：运行中的进程在时间片用完之后，让出处理机，转为就绪状态。
- 阻塞状态→就绪状态：阻塞状态下的进程获得了处理机外的资源，转为就绪状态。这是一个被动的行为，需要其它进程协助完成。

只有就绪状态才能转为运行状态。

### 进程控制

进程的控制使用原语实现。

#### 进程创建

操作系统创建一个新进程（创建原语）的过程如下：

1. 为新进程分配一个唯一的标识号，并申请一个空白的PCB。PCB的数量有限，若PCB申请失败，则进程创建失败。
2. 为新进程的程序、数据以及用户栈分配必要的内存。如果资源不足，则等待。

3. 初始化PCB，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。
4. 将新进程插入就绪队列，等待被调度。

允许父进程创建子进程，子进程继承父进程所拥有的资源。当子进程被撤销时，将这些资源归还父进程；当父进程被撤销时，子进程一并被撤销。

**进程终止**

进程终止分为正常结束和异常结束。撤销原语的执行过程如下：

1. 根据被终止的进程的标识符，从其PCB中读取该进程的状态。
2. 若该进程仍处于运行状态，立即终止其执行，将处理机资源分配给其它进程。
3. 若该进程有子进程，将终止所有的子进程。
4. 将该进程所拥有的全部资源，归还父进程或操作系统。
5. 将PCB从队列中删除。

**进程的阻塞与唤醒**

正在执行的进程，在等待某些数据到达或等待某些操作完成而无事可做时，由系统执行阻塞原语，进程会自发地使自己的状态变为阻塞状态。阻塞原语的执行过程：

1. 找到要被阻塞的进程的标识号对应的PCB。
2. 若该进程处于运行状态，则保护现场，将其转为阻塞状态，停止运行。
3. 将该PCB插入到相应事件的等待队列中去。

唤醒原语的执行过程：

1. 在该事件的等待队列中找到相应进程的PCB。
2. 将其从等待队列中移出，转为就绪状态。
3. 将该PCB插入到就绪队列，等待被调度执行。

**进程的切换**

进程切换是指处理机从一个进程的运行转到另一个进程的运行。过程如下：

1. 保存处理机上下文，包括程序计数器和其它寄存器。
2. 更新PCB信息。
3. 把进程的PCB移入相应的队列。
4. 选择另一个进程执行，并更新其PCB。
5. 更新内存管理的数据结构。
6. 恢复处理机上下文。

注意：调度是指决定资源分配给哪个进程的行为，是一种决策行为；切换是指实际分配的行为，是执行行为。

## 进程的组成

**PCB**

PCB常驻内存，包括以下内容：

1. 进程描述信息：进程标识符PID和用户标识符UID，UID决定该进程所归属的用户。
2. 进程控制和管理信息：进程当前状态、进程优先级、代码运行入口地址等。
3. 资源分配清单：用于说明有关内存地址空间的状况，打开文件的列表和I/O设备信息。
4. 处理机相关信息：处理机中各寄存器的值。

为了方便进程的调度和管理，各进程的PCB需要用适当的方法组织起来，常用的组织方式有：链接方式、所以方式。

- 链接方式将同一状态的PCB链接成一个队列，不同的状态对应不同的队列，阻塞队列根据阻塞原因的不同排成多个阻塞队列。
- 索引方式将同一状态的PCB组织在一个索引表中，不同的状态对应不同的索引表。

### 进程的通信

- 低级通信方式：PV操作；
- 高级通信方式：共享存储、消息传递、管道通信。

#### 共享存储

通信的进程之间有一块可直接访问的共享空间，使用同步互斥工具，对共享空间进行控制，达到通信的目的。共享空间必须通过特殊的系统调用实现，分为基于数据结构的共享（低级）、基于存储区的共享（高级）。

#### 消息传递

进程间的数据传递以格式化的消息为单位，通过系统提高的发送消息和接收消息两个原语进行数据交换。分为直接通信方式和间接通信方式。

#### 管道通信

管道是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。管道机制必须提供三个方面的协调能力：互斥、同步、确定双方的存在。管道是一个固定大小的缓冲区，采用版双工通信。管道中的数据一旦被读取，就被抛弃，释放空间以便写入更多数据；当读进程比写进程工作的快时，管道变空，随后的读取调用将默认地被阻塞。

## 2. 线程

### 线程的概念

为了更好地使多道程序并发执行，提高系统资源的利用率，而引入了线程。

线程可以理解为轻量级进程，由线程ID、线程控制块、程序计数器、寄存器集合和堆栈组成，每个线程的线程ID都是唯一的。

线程是进程的一个实体，一个进程可以创建多个线程，一个线程也可以创建另一个线程。线程自身不拥有系统资源，同一个进程的多个线程共享该进程所拥有的全部系统资源。

- 引入线程之后，进程只作为除CPU以外的系统资源的分配单位，即**进程是拥有资源的基本单位**；线程则作为处理机的分配单位，即**线程是独立调度的基本单位，**也是程序执行的最小单元。
- 由于线程自己不不拥有系统资源，创建和撤销线程不需要系统为之分配或回收资源，只需要少量的系统开销用于创建线程控制块。
- 由于归属于同一进程的多个线程之间共享该进程的存储，因此线程之间的通信与同步非常容易实现，线程间可以直接读写进程数据段（如全局变量）来实现通信。
- 线程之间的切换只需要保存少量的寄存器，开销很小。同一进程内的线程切换不会引起进程的切换，不同进程内的线程切换，必定会引起进程切换。

### 线程的实现方式

线程的实现可分为两类：用户级线程、内核级线程。

- 在用户级线程中，管理线程的所有工作都由应用程序来完成，内核意识不到线程的存在。
- 在内核级线程中，管理线程的所有工作都由内核完成，应用程序只有一个到内核级线程的编程接口。内核为进程及其内部的线程维护上下文信息，以及调度线程。

在一些系统中，会使用组合方式的多线程实现。线程创建、调度和同步都在应用程序中进行，一个应用程序的多个用户线程被映射到一些内核级线程上。

- 多对一模型：将多个用户级线程映射到一个内核级线程。
- 一对一模型：将一个用户级线程映射到一个内核级线程。
- 多对多模型：将n个用户级线程映射到一个内核级线程。

## 3. 处理器调度

处理机调度就是对处理机进行分配，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。

一个作业从提交开始直到结束，往往要经历以下三级调度：

- 作业调度：又称高级调度。为处于外存的一个作业分配系统资源、建立相应的进程，使其获得竞争处理机的权利。作业调度就是内存与辅存之间的调度，频率很低。多道批处理系统中大多配有作业调度。
- 内存调度：又称中级调度。使暂时不能运行的进程，调至外存等待，此时的进程状态处于挂起状态。挂起状态的进程等到运行条件满足，由中级调度将其重新调入内存，转为就绪状态。
- 进程调度：又称低级调度。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。进程调度是最基本的，不可或缺，进程调度的频率最高。

### 调度的时机

现代操作系统中，不能进行进程调度和切换的情况有以下几种：

- 在处理中断的过程中。
- 进程在操作系统内核程序临界区。临界区指访问临界资源的代码，临界资源一次指允许一个进程访问。
- 其它需要完全屏蔽中断的原子操作过程中，如加锁、解锁、中断现场保护和恢复等。

在上述过程中发生了引起调度的条件时，并不能马上进行调度和切换，应置系统的请求调度标志，上述过程结束后才进行相应的调度与切换。

应该进行进程调度与切换的情况有：

- 当发生引起调度条件，且当前进程无法继续运行下去。
- 当中断或自陷处理结束后。

### 进程调度的方式

- 非剥夺调度方式：正在运行的进程因某个事件而进入阻塞状态时，处理机分配给其它进程。非剥夺调度方式下，一旦处理机分配给一个进程，该进程就会保持处理机直到终止或转为其它状态。
- 剥夺调度方式：当一个进程正在执行时，若有某个更重要或紧迫的进程需要处理机，则立即暂停正在执行的进程，将处理机分配给这个更为重要或紧迫的进程。剥夺必须按照一定的原则，主要有：优先权、短进程优先、时间片原则。

### 调度的基本准则

- 提高CPU利用率：应尽可能使CPU处于忙的状态。
- 提高系统的吞吐量：系统吞吐量是指单位时间内CPU完成作业的数量。短进程优先可以提高系统的吞吐量。
- 降低周转时间、等待时间、响应时间：

- 周转时间=作业完成时间-作业提交时间；带权周转时间指作业周转时间与作业实际运行时间的比值。
- 平均带权周转时间=（作业1的带权周转时间+……+作业n的带权周转时间）/ n。
- 等待时间指进程处于等待处理机的时间之和。衡量一个调度算法的优劣，往往只需考虑等待时间。
- 响应时间指才用户提交请求到系统首次产生响应所用的时间。

## 典型的调度算法

**先来先服务FCFS调度算法**

FCFS调度算法属于非剥夺调度算法。从后备队列中选择最先进入队列的作业/进程进行处理。

FCFS调度算法简单，但效率低。对长作业有利，对短作业不利。

**短作业优先SJF调度算法**

SJF调度算法属于非剥夺调度算法，从后备队列中选择最短作业/短进程进行优先处理。

SJF调度算法的平均等待时间、平均周转时间最少，但对长作业不利（导致长作业长期不被调用，饥饿现象）、未考虑作业的紧迫度。作业时间的长短由用户提供的估计执行时间而定，而用户会故意缩短自己估计的作业时间，因此此算法实际并不一定能做到短作业优先调度。

**优先级调度算法**

优先级调度算法从后备队列中选择优先级最高的作业/进程进行处理。按照新的最高优先级的进程是否能够正在执行的进程，可分为非剥夺式优先级调度算法、剥夺式优先级调度算法；按照进程的优先级是否能够改变，可分为静态优先级、动态优先级。

**高响应比优先级调度算法**

响应比=（等待时间+要求服务时间）/ 要求服务时间

高响应比优先级调度算法主要用于作业调度，每次进行进行作业调度时，先计算后备作业队列中的每个作业的响应比，选择响应比最高的作业投入运行。

根据响应比公式可知，等待时间越长，要求服务时间越短，其响应比越高。这样一方面利于短作业优先执行，另一方面避免了长作业的饥饿现象。

**时间片轮转调度算法**

时间片轮转调度算法是一种剥夺调度算法，主要用于分时系统，进程调度按照先来先服务的原则，将处理机分配给最先进入就绪队列的进程，但其仅能运行一个时间片。若其在使用完该时间片后未完成运行，它必须释放处理机给其它进程（被剥夺），被剥夺的进程返回就绪队列的尾端重新排队。

**多级反馈队列调度算法**

多级反馈队列调度算法是前几种算法的结合，其实现思想是：

- 设置多个就绪队列，按照1~n的顺序队列的优先级降低。
- 时间片大小不同，优先级越高的队列中的进程，将获得越小的时间片。
- 当一个新进程进入就绪队列时，应将其放入第1级队列的末尾，按先来先服务的原则排队等待调度。当该进程执行完一个时间片后，若其仍未完成，则转入第2级队列的末尾，按先来先服务的原则排队等待调度。依次类推，直到该进程转入第n级队列，便在第n级队列中轮转。
- 只有当第1级队列为空时，才调度第2级队列中的进程运行，依次类推。每次调度都是如此。

多级反馈队列调度算法实现了新作业优先、短作业优先，长作业不会长期不被执行。

# 4. 进程同步

一次仅允许一个进程使用的资源称为临界资源。将对临界资源的访问过程分成四部分：

- 进入区：进程检查是否可以进入临界区，若可以进入，还需设置**正在访问临界区**的标志，以组织其它进程进入。
- 临界区：进程访问临界资源。访问临界资源的代码也被称为临界区。
- 退出区：清除正在访问临界区的标志。
- 剩余区：代码的其它部分。

**同步**是指多个进程为完成某一项工作，需要等待其它进程的信息或是传递给其它进程信息，而产生的工作次序上的制约关系。同步又称为直接制约关系。

**互斥**是指多个进程为竞争临界资源而产生的制约关系。互斥遵循空闲让进、忙则等待、有限等待（等待时间必须是有限时间）、让权等待（进程等待时要让出处理机）的原则。互斥又称为间接制约关系。

## 实现临界区互斥的方法

### 软件实现方法

0. 锁变量：用一个公用bool型变量来表示临界区是否可用，称为锁变量。锁变量会可能会导致两个进程进入临界区，当进程A检查锁变量发现临界资源可用时，下一步是将锁变量设置为临界资源不可用；但若此时正好处于时间片的末尾，进程A被调度离开处理机，进程B进入处理机，检查锁变量发现临界资源可用，进程B设置锁变量进入临界区；此时进程调度将进程B替换为进程A，进程A不会再检查锁变量，直接设置锁变量进入临界区，这样就导致了两个进程都处于临界区的错误。
1. 单标志法：用一个公用整型变量（turn）来表示谁进入了临界区。每个进程都有一个唯一的整型编号，只有当该变量与进程的编号相同时，该进程才能进入临界区。以两个互斥的进程为例，

```
# 进程0:
while(turn != 0); // 进入区，等待自己的编号
critical section; // 临界区
turn = 1;  // 退出区，此时设置公用变量，使得另一进程可以访问临界区
remainder section; // 剩余区
# 进程1:
while(turn != 1); // 进入区，等待自己的编号
critical section; // 临界区
turn = 0;  // 退出区，此时设置公用变量，使得另一进程可以访问临界区
remainder section; // 剩余区
```

   从代码中可以看出，即使进程0执行完进入区就被调度出处理机，进程1也是无权访问临界资源的，达到了互斥的目的。但是若公用变量为0，而进程0不需要访问临界区，则公用变量将无法更改，需要访问临界资源的进程1也无法进入临界区。
2. 双标志法先检查：用一个bool型数组（flag[]）来表示各进程是否处于临界区。每个进程对应一个元素，当元素值为FALSE时，对应进程不在临界区；当元素值为TRUE时，对应进程处于临界区。以两个互斥的进程为例：

```
# 进程0:
while(flag[1]); // 进入区，等待互斥进程不在临界区
flag[0] = TRUE; // 进入区
critical section; // 临界区
flag[0] = FALSE;  // 退出区
remainder section; // 剩余区
# 进程1:
while(flag[0]); // 进入区，等待互斥进程不在临界区
flag[1] = TRUE; // 进入区
critical section; // 临界区
flag[1] = FALSE;  // 退出区
remainder section; // 剩余区
```

双标志法先检查不像单标志法那样交替使用临界区，不会出现一个进程无法进入临界区的问题。但是当按照下面的顺序调度进程时，可能会出现两个进程同时进入临界区的错误。

```
while(flag[1]); // 若此时flag[0] = FALSE, flag[1] = FALSE
while(flag[0]);
flag[0] = TRUE;
flag[1] = TRUE;
```

3. 双标志法后检查：此算法与双标志先检查法类似，但是是先置位后检查。以两个互斥的进程为例：

```
# 进程0:
flag[0] = TRUE; // 进入区，先置位
while(flag[1]); // 进入区，等待互斥进程不在临界区
critical section; // 临界区
flag[0] = FALSE;  // 退出区
remainder section; // 剩余区
# 进程1:
flag[1] = TRUE; // 进入区，先置位
while(flag[0]); // 进入区，等待互斥进程不在临界区
critical section; // 临界区
flag[1] = FALSE;  // 退出区
remainder section; // 剩余区
```

后检查法虽然不会像先检查法那样会导致两个进程同时进入临界区，但是 两个进程可能会发现对方都是TRUE，而互相谦让导致饥饿现象。

4. Peterson's算法：在双标志后置位法的基础上，增加了公用变量turn。想进入临界区的进程先设置自己的标志，再更改turn标志。以两个互斥的进程为例：

```
# 进程0:
flag[0] = TRUE; turn = 1; // 进入区，先置位
while(flag[1] && turn == 1); // 进入区，等待互斥进程不在临界区
critical section; // 临界区
flag[0] = FALSE;  // 退出区
remainder section; // 剩余区
# 进程1:
flag[1] = TRUE; turn = 0; // 进入区，先置位
while(flag[0] && turn == 0); // 进入区，等待互斥进程不在临界区
critical section; // 临界区
flag[1] = FALSE;  // 退出区
remainder section; // 剩余区
```

```
Peterson's算法利用flag解决临界资源的互斥访问，利用turn解决饥饿现象。
```

**硬件实现方法**

1. 中断屏蔽方法：当一个进程正在使用处理机执行它的临界区代码时，会屏蔽所在CPU的中断（CPU只在发生中断时才引起进程切换），防止切换到其它进程。执行完临界区后在打开中断。

   这种方法限制了处理机交替执行的能力，执行效率降低。

2. 硬件指令方法：采用原子操作指令，其执行时会屏蔽内存总线的中断，防止其它进程操作内存。

   (1). TSL指令或TestAndSet指令：其功能是读取指定标志，并把它设置为真。将每个临界资源设置一个共享bool变量lock，lock为TRUE时表示该资源正在被占用，lock为FALSE时表示该资源未被占用。

   ```
   while(TestAndSet(&lock));
   // lock为TRUE时，表示其它进程正在访问临界资源，会一直循环；一旦其它进程退出临界区，lock变为FALSE，不在继续循环，并置lock为TRUE
   critical section; // 临界区
   lock = FALSE; // 退出区
   remainder section; // 剩余区
   ```

   (2).XCHG指令或Swap指令：其功能是交换两个字的内容。将每个临界资源设置一个共享bool变量lock，同时每个进程设置一个局部bool变量key，用于与lock交换信息。

   ```
   key = TRUE; // 进程想要访问临界资源时，将key置TRUE
   while(key != FALSE)
       Swap(&lock, &key);
   // 对于上述循环，如果有其它进程访问临界区，lock值为TRUE，此时交换key与lock，其值都为TURE，会一直循环；当使用临界资源的进程进入退出区，将lock置FALSE，此时交换key与lock的值，key变为FALSE，退出循环。同时交换使得lock变为TRUE，阻止其它进程。
   critical section; // 临界区
   lock = FALSE; // 退出区
   remainder section; // 剩余区
   ```

   硬件指令方法中进程在等待临界资源时一直循环，会耗费处理机资源，不能实现让权等待。多个进程访问同一个临界资源时，可能会使某个进程因调度进处理机的时机不在其它进程的退出区，而一直无法访问临界资源，造成饥饿现象。

软件硬件两种方式中，只有Peterson's算法、TSL、XCHG是正确的。

## 信号量与PV操作

信号量机构是一种功能较强的机制，用来解决互斥和同步问题。它只有两个标准的原语：P操作、V操作（首字母来自荷兰语）。

1. 设一个整型信号量S，P操作和V操作可描述为：

```
P(S) {
    while(S <= 0); // 当S非正时，不断循环测试
    S--;
}
V(S) {
    S++;
}
```

由于P操作需要不断循环测试，因此这种方式未遵循让权等待。

2. 设一个记录型信号量S，S的结构为

```
typedef struct {
    int value;
    struct process *L; // 进程链表L,记录等待临界资源的进程
} semaphore;
```

P操作和V操作可描述为：

```
P(semaphore S) {
    S.value--;
    if (S.value < 0) { // 此时临界资源是否已被分配完
        add this process to S.L; // 将该进程添加到等待队列
        block(S.L); // 阻塞等待队列中的进程
    }
}
V(semaphore S) {
    S.value++;
    if (S.value <= 0) { // 当该进程释放临界资源时，检测是否有等待的进程
        remove a process P from S.L; // 将进程P从等待队列中移出
        wakeup(P); // 唤醒进程P，进程P将继续执行进入临界区
    }
}
```

这种方式下可以实现让权等待。

### 使用信号量实现同步

设分布位于进程0、进程1的两条指令x、y，y的执行需要x先执行完，x的执行与否通过一个信号量S表征。

```
semaphore S = 0;

# 进程0
x;
V(S); // 此时指令x已经执行完，设置信号量

# 进程1
P(S); // 检测信号量，等待指令x执行完
y;
```

**使用信号量实现互斥**

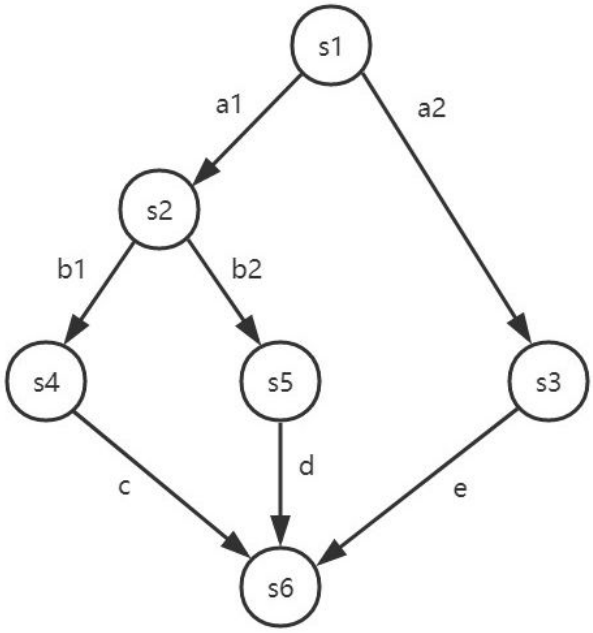
设进程0、进程1互斥访问一个临界资源，使用一个信号量S实现。S的初始值为1，即可用临界资源数量为1。

```
semaphore S = 1;

# 进程0
P(S);
critical section; // 临界区
V(S);

# 进程1
P(S);
critical section; // 临界区
V(S);
```

只有有一个进程进入临界区，就会把S减1，S变为0，阻止其它进程进入临界区。

**使用信号量实现前驱关系**

如下图的前驱关系，通过信号量实现。




```
# 程序段S1
all statement;
V(a1); V(a2); // S1运行完成，通过信号量通知S2、S3

# 程序段S2
P(a1); // 等待S1运行完成
all statement;
V(b1); V(b2); // S2运行完成，通过信号量通知S4、S5

# 程序段S3
P(a2); // 等待S1运行完成
all statement;
V(e); // S3运行完成，通过信号量通知S6

# 程序段S4
P(b1); // 等待S2运行完成
all statement;
V(c); // S4运行完成，通过信号量通知S6

# 程序段S5
P(b2); // 等待S2运行完成
all statement;
V(d); // S5运行完成，通过信号量通知S6

# 程序段S6
P(c); P(d); P(e); // 等待S3、S4、S5运行完成
all statement;
```

## 管程

管程将进程间共享的硬件资源和软件资源抽象成一个数据结构，用少量的信息描述其资源特性，而忽略资源本身的细节。管程包含该数据结构，以及用于初始化该数据结构、操作该数据结构的方法。

上述的数据结构局部于管程，对其访问只能通过调用管程提供的方法。并且管程限制每次只允许一个进程进入管程。

## 经典同步问题

### 生产者-消费者问题

问题描述：一组生产者和一组消费者共享一个初始为空、大小为n的仓库，每次只允许一个人操作仓库，

- 只有仓库没满，生产者才能放入产品，否则等待；
- 只有仓库不空，消费者才能取出产品，否则等待。

```
semaphore mutex = 1; // 临界区（仓库）互斥信号
semaphore empty = n; // 空余位置数量
semaphore full = 0; // 存入商品数量

# 生产者
while(1) {
    produce an item;
    P(empty); // 检查是否有空余位置，若没有就等待
    P(mutex);
    add this item to buffer; // 临界区
    V(mutex);
```

```
    V(full);
}

# 消费者
while(1) {
    P(full); // 是否有存下的商品可用，若没有就等待
    P(mutex);
    remove an item from buffer; // 临界区
    V(mutex);
    V(empty);
    consume this item;
}
```

问题描述：桌上有一个盘子，只能存放一个水果。爸爸只向盘子里放苹果，妈妈只向盘子里放橘子，儿子只等着吃盘子里的苹果，女儿只等着吃盘子里的橘子。

这个问题中爸爸和妈妈构成互斥关系，爸爸和儿子、妈妈和女儿构成分布构成同步关系。

```
semaphore plate = 1; // 使用盘子的互斥信号
semaphore apple = 0, orange = 0; // 同步信号

# 爸爸
while(1) {
    P(plate); // 互斥地使用盘子
    put an apple on the plate;
    V(apple);
}

# 妈妈
while(1) {
    P(plate); // 互斥地使用盘子
    put an orange on the plate;
    V(orange);
}

# 儿子
while(1) {
    P(apple); // 等待盘子中有苹果
    take an apple from the plate;
    V(plate); // 爸爸-儿子这一同步进程完成，释放盘子的使用权
}

# 女儿
while(1) {
    P(orange); // 等待盘子中有橘子
    take an orange from the plate;
    V(plate); // 妈妈-女儿这一同步进程完成，释放盘子的使用权
}
```

**读者-写者问题**

问题描述：有读者、写者两种多个进程，共享一个文件。要求：

- 同时允许多个读者进程对该文件操作；
- 每次只允许一个写者进程对该文件操作；
- 写者进程对文件操作前，必须保证其它任何读写进程都已退出对该文件的操作；
- 写者进程在完成文件操作前，不允许其它任何读写进程对文件操作。

```
int count = 0; // 记录读者进程的数量
semaphore mutex = 1; // 保护更新count变量时互斥
semaphore rw = 1; // 保证读写互斥地访问文件

# 写者进程
while(1) {
    P(rw);
    writing; // 因为写者进程与其它任何进程都互斥
    V(rw);
}

# 读者进程
while(1) {
    P(mutex);
    if (count == 0)
        P(rw); // 阻塞写者进程
    // 若count不为0，说明已有读者进程在操作共享文件，已经阻塞了写者进程
    count++;
    V(mutex);
    reading;
    P(mutex);
    count--;
    if (count == 0)
        V(rw); // 如果没有读者进程，则释放共享文件的使用权
    V(mutex);
}
```

上述算法读者进程有更高的优先级，若有实现读写进程有相同的优先级，可增加一个信号量。

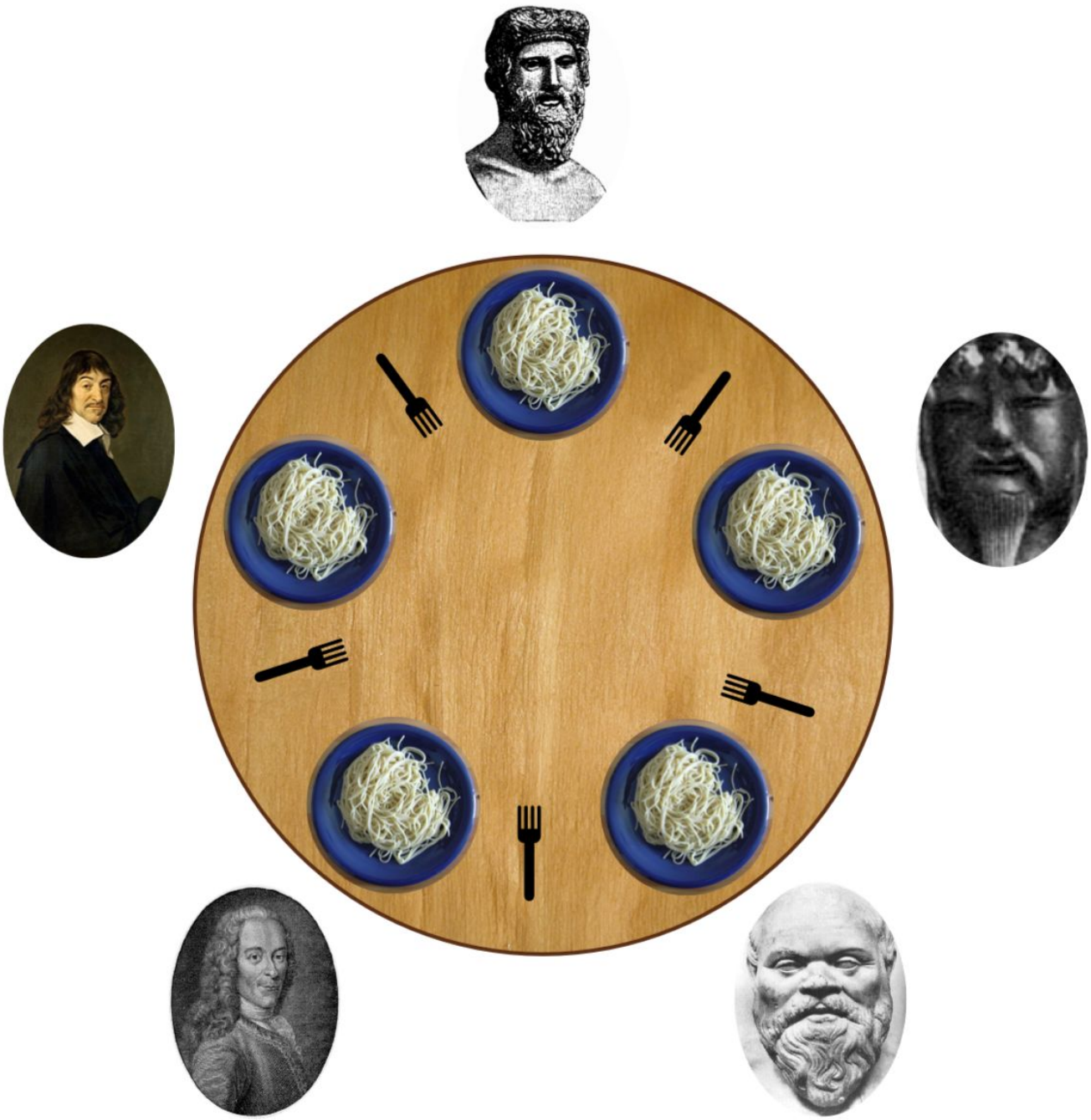
```
int count = 0; // 记录读者进程的数量
semaphore mutex = 1; // 保护更新count变量时互斥
semaphore rw = 1; // 保证读写互斥地访问文件
semaphore w = 1; // 用于实现优先级

# 写者进程
while(1) {
    P(w); // 在无写者进程时，要求对共享文件操作
    P(rw);
    writing; // 因为写者进程与其它任何进程都互斥
    V(rw);
    V(w);
}

# 读者进程
while(1) {
    P(w); // 如果此时有写者进程请求，便阻塞了新的读者进程进入
    P(mutex);
```

```
    if (count == 0)
        P(rw); // 阻塞写者进程
    // 若count不为0，说明已有读者进程在操作共享文件，已经阻塞了写者进程
    count++;
    V(mutex);
    V(w);
    reading;
    P(mutex);
    count--;
    if (count == 0)
        V(rw); // 如果没有读进程，则释放共享文件的使用权
    V(mutex);
}
```

**哲学家进餐问题**

问题描述：如下图，五位哲学家饥饿时，会依次拿起左右两把叉子进食，进食完会将叉子放回原处。如果哲学家无法获得两个叉子，则等待。

五位哲学家之间是互斥关系，可能会造成每位哲学家左手都有叉子，但是无法获得右手的叉子的死锁。



设哲学家的编号为0~4，编号i的哲学家的左手边的叉子编号为i，右手边的叉子的编号为[(i+1) % 5]。

```
semaphore chopstick[5] = {1, 1, 1, 1, 1};
semaphore mutex = 1;

# 每位哲学家
while(1) {
    P(mutex);
    P(chopstick[i]);
    P(chopstick[(i+1)%5]);
    V(mutex);
    // 将拿两个叉子放在一起，避免了拿起一个叉子就被调度，而导致的死锁
    eat;
    V(chopstick[i]);
    V(chopstick[(i+1)%5]);
```

```
}
```

**吸烟者问题**

问题描述：有三个吸烟者和一个供应者，有三种材料：烟草、纸、胶水。三个吸烟者每人分别有三个材料中的一种；供应者有三种材料，但是他每次只会将其中的两种放到桌子上。此时拥有最后一种材料的吸烟者会将三种材料制作称烟并抽掉，并告诉供应者已完成。供应者继续将材料放到桌上，一直重复。

```
int random; // 随机数，产生供应者的行为
semaphore offer1 = 0; // 定义信号量，对应烟草和纸的组合
semaphore offer2 = 0; // 定义信号量，对应烟草和胶水的组合
semaphore offer3 = 0; // 定义信号量，对应纸和胶水的组合
semaphore finish = 0; // 定义信号量，抽烟是否完成

# 供应者进程
while(1) {
    random = (0,1,2)中随机一个;
    if (random == 0) V(offer1);
    else if (random == 1) V(offer2);
    else V(offer3);
    P(finish);
}

# 拥有烟草的吸烟者
while(1) {
    P(offer3);
    make a cigarette and smoke it;
    V(finish);
}

# 拥有纸的吸烟者
while(1) {
    P(offer2);
    make a cigarette and smoke it;
    V(finish);
}

# 拥有胶水的吸烟者
while(1) {
    P(offer1);
    make a cigarette and smoke it;
    V(finish);
}
```

## 5. 死锁

死锁产生的原因：

- 对少量不可剥夺的系统资源竞争；
- 进程推进顺序非法。

死锁产生的必要条件。以下四个条件必须同时满足才会造成死锁，

- 互斥条件：某种系统资源每次只允许一个进程占用。
- 不剥夺条件：进程未使用完该资源，不能被其它进程强行夺走，只能在使用完之后主动释放。
- 请求和保持条件：进程需要两项资源，并且已经获得了其中一项，而另一项正在被其它进程使用。此时该进程因请求另一项资源而被阻塞，但是不会释放已获得的资源。
- 循环等待条件：存在循环等待链。该链中每个进程已获得的资源同时被链中下一个进程所请求，此时若链中每个进程的保持自己已有的资源，则每个进程都无法获得请求的另一个资源而无限等待。

循环等待不等同于死锁，在循环等待中如果链上某个进程所请求的资源被链外的一个进程释放提供，循环等待就被解除了。

## 死锁的处理

**死锁预防**

死锁预防只需破坏四个必要条件中的一个即可。

- 破坏互斥条件：对应临界资源不可行。
- 破坏不剥夺条件：保持有不剥夺资源的进程在等待时，必须释放保持的资源。这种方法下进程会反复申请资源增加系统开销，常用于状态易于保存和恢复的资源，如CPU寄存器、内存。
- 破坏请求和保持条件：采用预先静态分配方法，即进程运行前一次申请完它需要的所有资源，否则不投入运行。由于个别资源可能仅在进程某个时间点使用一次，其它时间空闲，这种方法会造成资源的浪费。
- 破坏循环等待条件：采用顺序资源分配法。首先系统给所有的资源编号，并规定进程在申请资源时必须按编号递增的顺序申请，同类资源一次申请完，这样就无法形成循环。这种方法限制了新设备的加入，造成资源的浪费，同时增加了编程的难度。

**死锁避免**

系统安全状态：系统能够按某种顺序，为每个进程分配其所需的资源，满足其对资源的最大需求，使得每个进程都能顺序地完成的状态。此时的进程执行顺序称为安全序列。如果系统无法找到一个**安全序列**，则系统处于不安全状态。

例子：设系统有3个进程P1、P2、P3，分别需要10台、4台、9台磁带机。系统一共有12台磁带机，其中P1保持有5台，P2保持有2台，P3保持有2台，剩余3台未分配。此时3台磁带机无法满足P1、P3的需求，满足P2的需要。可以按照如下顺序执行：

- 分配2台磁带机给P2，P2运行完成后释放所有的磁带机，空闲的磁带机变为5台；
- 分配5台磁带机给P1，P1运行完成后释放所有的磁带机，空闲的磁带机变为10台；
- 分配7台磁带机给P3，P3运行完成。

此时的安全序列为P2$\rightarrow$P1$\rightarrow$P3，系统处于安全状态。

**银行家算法：**

- 当进程首次申请资源时，要测试该进程对资源的最大需求量。如果系统现存的空闲资源能够满足，则按当前的申请量进行分配，否则推迟分配。
- 当进程执行中继续申请资源时，要先测试该进程已占用的资源数+本次想申请的资源数是否超过了该进程对资源的最大需求量。t
    - 若超过则拒绝分配；
    - 若没有超过则测试系统现存的空闲资源能否满足该进程尚需的最大资源量，能满足则分配，否则推迟分配。

**银行家算法的过程：**

- 可利用资源矢量$Available$：含有$m$个元素的数组，每个元素代表一个资源的可用数目。

- 最大需求矩阵$Max$：为$n \times m$的矩阵，对应$n$个进程$m$种资源。$Max[i,j]=K$表示进程$i$对资源$j$的最大需求为$K$。
- 分配矩阵$Allocation$：为$n \times m$的矩阵，$Allocation[i,j]=K$表示进程$i$已经分配得到资源$j$的数量为$K$。
- 需求矩阵$Need$：为$n \times m$的矩阵，$Need[i,j]=K$表示进程$i$需要资源$j$的数量为$K$。

上述矩阵存在关系：$Need=Max-Allocation$。

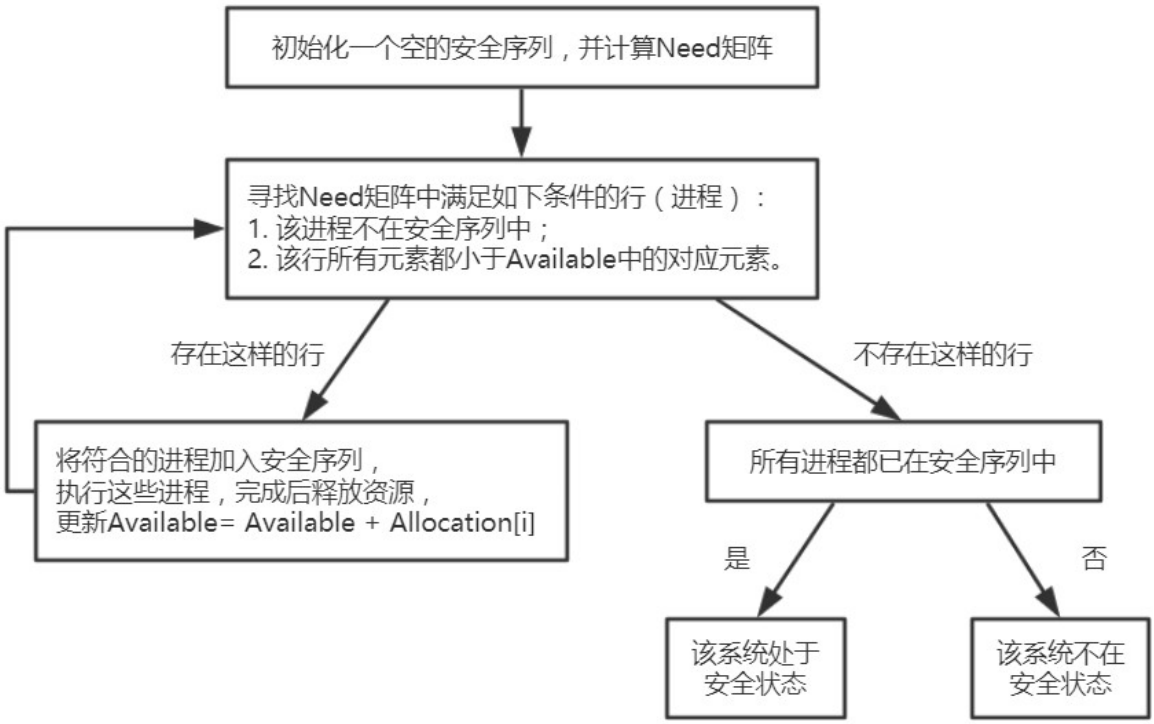
设进程$P_i$的资源请求矢量为$Request_i$，此时进程$P_i$申请$j$类资源，具体步骤如下：

1. 计算$Need$矩阵：$Need=Max-Allocation$；
2. 测试$Request_i[j] \leq Need[i,j]$，不成立则拒绝资源的申请；
3. 测试$Request_i[j] \leq Available[i,j]$，不成立则推迟分配；
4. 系统按照$Request_i[j]$的数量**试探分配**给$P_i$资源，并更新数据：

   $Available=Available-Request_i$

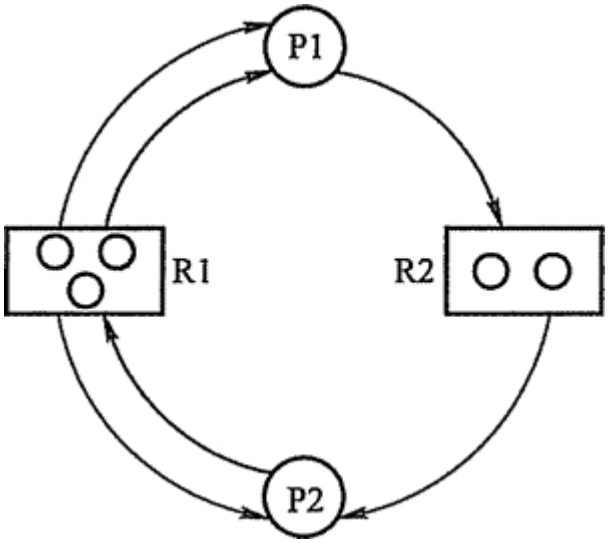
   $Allocation[i,j]=Allocation[i,j]+Request_i[j]$

   $Need[i,j]=Need[i,j]-Request_i[j]$
5. 执行系统安全算法，测试系统在此次分配后是否仍处于安全状态。若安全，则将资源分配给进程；否则，作废此次试探分配，恢复原来的资源分配状态，让进程等待。安全性算法如下：




**死锁检测和解除**

系统死锁可利用资源分配图描述。如下图




圆圈表示一个进程，方框表示一类资源，方框中的小圈表示一个资源。从进程到资源的有向边为请求边，从资源到进程的有向边为分配边。上图中P1已经得到2个R1资源，请求一个R2资源；P2已经得到一个R1资源和一个R2资源，请求一个R1资源。

判断系统是否死锁转变为简化资源分配图。若资源分配图中经过简化所有的边都可以消去，则系统不会死锁。按照如下方法简化：

1. 找到可以满足其所有请求的点（进程），如P1请求一个R2资源，R2有一个空闲的资源，可以满足P1的请求。此时P1可以完成运行，消去所有和P1相关的边。
2. 由于上述进程释放了一些资源，则可以唤醒原来阻塞的进程。再次寻找可以满足其所有请求的点，消去和其相关的边。重复此步骤，直到无法继续进行。

若最后所有的边都被消去，称此资源分配图是可以完全简化的，表示系统不会死锁。否则系统会发生死锁。

**死锁定理：**系统S状态为死锁的条件是当且仅当S状态的资源分配图是不可完全简化的。

一旦检测出死锁，应采取措施解除死锁。

- 资源剥夺法：挂起某些死锁的进程，并抢占它的资源重新分配给其它死锁进程。
- 撤销进程法：强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。撤销按照进程优先级和代价高低判断。
- 进程回退法：让一个或多个进程回退到足以回避死锁的状态，此时进程自动放弃资源。此种方法下需要系统保持进程的历史信息，设置还原点。

# 三、内存管理

## 1. 内存管理的概念

内存管理的功能：

- 内存空间的分配与回收
- 地址转换：将逻辑地址转换成物理地址
- 内存空间的扩充：从逻辑上扩充内存
- 存储保护：保证各道程序在各自的存储空间运行，互补干扰

### 程序装入和链接

源程序变为内存中可执行的程序，需要以下几个步骤：编译、链接、装入。编译形成目标模块，链接形成装入模块。

程序的链接有三种方式：

- 静态链接：程序运行前，先将各目标模块和其所需库函数链接成一个完整的可执行程序，不再拆开。
- 装入时动态链接：目标模块在装入内存时，边装入边链接。
- 运行时动态链接：在程序执行到该目标模块时才对它进行链接。其优点是便于修改和更新，便于实现对目标模块的共享。

装入模块装入内存有三种方式：

- 绝对装入：编译时已经确定程序在内存中的位置，编译产生的代码中的地址为绝对地址，不需要对地址进行转换。只适用于单道程序环境中。
- 可重定位装入：多道程序环境中，目标模块的起始地址多是从0开始的，此时程序中的地址都是相对地址。装入程序根据内存的当前状况，将装入模块装入内存的合适位置，其中的地址将全部修改为绝对地址。修改地址的过程称为重定位，这种方式又称静态重定位。

  这种方式下装入一个程序时必须分配其要求的全部内存空间，并且装入后无法移动位置，也不能再申请空间。
- 动态运行时装入：又称动态重定位。装入程序把装入模块装入内存时并不立即修改地址，而是在程序真正要执行时才进行相对地址到绝对地址的转换。

  这种方式下可以将程序分配到不连续的内存空间中；在程序运行之前可以只装入其部分代码即可投入运行，运行期间程序可根据需要申请新的内存。

### 逻辑地址空间和物理地址空间

编译后每个模块都是从0开始编址，称为目标模块的相对地址/逻辑地址。程序员只需要直到逻辑地址即可。

内存中的实际地址称为绝对地址/物理地址。将逻辑地址转换为物理地址，称为地址重定位。

### 内存保护

内存分配前，需要保护操作系统不受用户进程的影响，用户进程之间也不能相互影响。需要内存保护，有两种方法：

- 在CPU中设置一对上下限寄存器，存放用户作业在主存中的上下限地址。每当CPU访问一个地址时，与两个寄存器中的地址比对，判断是否越界。
- 采用重定位寄存器（或基址寄存器）和界地址寄存器（或称限长寄存器），重定位寄存器中存有用户作业的最小的物理地址，界地址寄存器中存有用户作业的最大的逻辑地址。每个逻辑地址必小于界地址寄存器的值，否则发生越界。未发生越界的逻辑地址加上重定位寄存器的值，就是该逻辑地址对应的物理地址，再将此物理地址提交给内存即可。

### 覆盖与交换

覆盖与交换技术是多道程序环境下用来扩充内存的方法。交换技术用于不同进程之间，而覆盖技术则用于同一个程序或进程中。现代操作系统通过虚拟内存技术替代了覆盖技术，而交换技术仍在使用。

#### 覆盖

将用户空间分成一个固定区和若干个覆盖区，将活跃的部分放在固定区，其余部分按调用关系分段，将即将访问的段放入覆盖区，其余段放在外存。在某段需要被调用前，在将其调入覆盖区，替换覆盖区中的原有的段。

覆盖技术的特点是打破了必须将一个进程的全部信息装入主存后才能运行的限制。

#### 交换

交换是将处于等待状态的程序从内存移到外存，腾出内存空间，称为换出；将准备好竞争CPU的程序从外存移到内存，称为换入。

## 2. 连续分配管理方式

### 单一连续分配

此方式下内存分为系统区和用户区，系统区供操作系统使用，通常在地址低位；用户区是除系统区之外的内存空间。这种方式内存中只有一道程序，无需内存保护。

- 优点是简单、无外部碎片，可以采用覆盖技术。
- 缺点是只能用于单用户单任务的操作系统中，有内部碎片，存储器利用率极低。

### 固定分区分配

此方式下用户内存被划分成若干个固定大小的区域，每个分区只装入一道作业，当有空闲分区时便从外存的后备作业队列中选择适当大小的作业装入分区。

- 分区大小相等
- 分区大小不等划分为多个较小的分区，适量中等的分区和少量大分区。

为了便于内存分配，通常将分区按大小排队，并建立一个分区说明表，每一个表项包括每个分区的起始地址、大小及状态（是否已分配）。

固定分区分配这种方式为最简单的多道程序存储分配，无外部碎片。但是有时可能程序太大而无法放入一个分区，需要使用覆盖技术；固定的分区大小会造成内部碎片，浪费内存空间。引出此方式下的存储空间利用率低。

### 动态分区分配

动态分区分配又称可变分区分配。这种方式下是在进程装入内存时，根据进程大小动态建立分区，使分区的大小适合进程的需要。对空闲内存的跟踪，可以使用位图来标记没有分配的内存单元，或者使用链表将空闲的内存块链接起来。

动态分区分配在开始时会很好地利用内存，但是之后会导致小的内存碎片。随着进程的不断换入换出，碎片越来越多，内存的利用率下降。解决此问题可以通过紧凑技术，即操作系统对进程进程移动和整理，消除碎片。

**动态分区的分配策略**

当进程装入或换入内存时，操作系统需要决定分配哪个内存块给进程。主要有以下算法：

- 首次适应算法：空闲分区按照**地址递增**的顺序排列，分配内存时按顺序查找，找到第一个大小能满足进程的空闲分区，分配给进程使用。
- 最佳适应算法：空闲分区按照**容量递增**的顺序排列，分配内存时按顺序查找，找到第一个大小能满足进程的空闲分区，分配给进程使用。也即挑出满足要求且最小的空闲分区。
- 最坏适应算法：空闲分区按照**容量递减**的顺序排列，分配内存时按顺序查找，找到第一个大小能满足进程的空闲分区，分配给进程使用。也即挑出满足要求且最大的空闲分区。又称最大适应算法。
- 邻近适应算法：在首次适应算法的基础上，查找时从上一次查找结束的地方开始。又称循环首次适应算法。

首次适应算法是最好的，而最佳适应算法性能通常很差，最坏适应算法性能最坏。

# 3. 非连续分配管理方式

**抖动/颠簸**是指刚刚换出的页面马上又换入主存，刚刚换入的页面马上又换出主存。发生抖动时会大大降低系统的效率。

**工作集/驻留集**是指某段时间内，进程要访问的页面集合。为防止抖动现象，需要选择合适的工作集大小。操作系统需要跟踪每个进程的工作集，并为进程分配大于其工作集的物理块数；若还有空闲物理块，则再调入一个进程到内存；若所有工作集之和大于可以物理块数，则操作系统会暂停一个进程，防止抖动。

## 基本分页存储管理方式（一维）

分页存储管理的思想是：把主存空间划分成大小相等的块，一般大小为2的整数幂。块相对较小，作为主存的基本单位。每个进程也已块为单位进行划分，进程在执行时已块为单位逐个申请主存中的空间。分页管理不会产生外部碎片，每个进程平均产生半个块大小的内部碎片，内存利用率高。

进程中的块称为页，内存中的块称为页框，外存也以同样的大小划分，直接称为块。取页的大小为4KB，分页后的逻辑地址可以分为两部分：页号P和页内偏移量W，页号为12~31位，页内偏移量（页内地址）为0~11位。

为了便于内存中找到进程每个页对应的物理块，系统为每个进程建立了一张页表。页表存储在内存中，每个页表项包含两个部分：页号和页号对应的物理块号。页表实现了页号到物理块号的地址映射，转换时将逻辑地址的页号换成物理块号即可转换为物理地址。

**基本地址转换机构**

系统中通常设置一个页表寄存器PTR，用于存放页表在内存中的起始地址和页表长度。进程未执行时，页表起始地址和页表长度都保存在进程控制块PCB中，进程执行时，两项内容存入PTR。设页面大小为L，逻辑地址A到物理地址E的转换过程如下：

1. 计算逻辑地址对应的页号P = A / L和页内偏移量W = A % L，也即将逻辑地址分成两部分；
2. 比较页号P和页表长度M（两个都是偏移量），若P大于等于M，则发生越界中断，否则继续执行；
3. 查找页号P对应的物理块号b。b = 页表起始地址（也是第0页对应的块号） + 页号P * 每页包含的物理块数；
4. 计算物理地址E = b * L + W。

该过程由硬件自动完成。

注意：如果以4KB为页的大小，则32位地址中有20位可以作为页号，共1M页，那么每个页表项的长度最少为3B = 24 > 20。页表也是存放在页中的，取页表项长度为4B，则每个页中可以存放1K个页表项，仍需要1K页用于存放页表。

如果地址转换的过程不够快，则访存速度必然降低；如果页表过大，占用了过多的页，则内存的利用率降低。

**具有快表的地址转换机构**

在地址转换机构中增设一个快表TLB，又称联想寄存器。快表是一个具有并行查找能力的高速缓冲存储器，用来存放当前访问的若干页表项，根据程序访问的局部性原理，加速地址转换的过程，一般TLB的命中率达90%以上。相对的内存中的页表也被称为慢表。

具有TLB的机构中，地址转换的过程如下：

1. 由硬件将逻辑地址分开为页号和页内偏移量。页号送入高速缓存，将其与TLB内的页表项进行比较。
2. 若匹配到对应的页表项，页直接从TLB中取出该页对应的页框号，与页内偏移量拼接成物理地址使用。
3. 若每页匹配到，则需要访问内存中的页表，计算物理地址。同时将该页表项替换入TLB中。

有些处理机设计为快表和慢表同时查找，如果在快表中匹配到则停止慢表的查找。

**两级页表**

随着内存容量的不断增大，页表的程度也在增长。为了压缩页表，引入二级分页。规定顶级页表只能存入一个页，如上述的4KB的分页只能保存1K个页表项，顶级页表的长度不能大于1K。此时1K一级页号需要10位地址，4KB的分页需要12位存储页内偏移量，所有二级页号也是10位，正好可以放入一个分页。

也即两级页表方式下，32位逻辑地址中，一级页号占用最高的10位， 二级页号占用中间的10位，页内偏移量占用最低的12位。

## 基本分段存储管理方式（二维）

分段存储管理的思想是：按照用户进程中的自然段划分逻辑空间，也即段的大小根据进程确定，不是固定的长度。但是一个程序可以根据需要分成多个段，段内连续，段间不连续。

分段后将逻辑地址分为两部分：段号S和段内偏移量W。内存分页对应用户是透明的，但在段式系统中，段号和段内偏移量必须由用户提供，这个工作一般由编译器完成。

每个进程都有一个段表，包含段号、段长、本段在内存中的起始地址，用于逻辑地址到物理地址的转换。系统中设置一个段表寄存器，用于存放段表初始地址F和段表长度M。进程执行时会将该进程对应的两项内容存入寄存器。

1. 将逻辑地址分成段号和段内偏移量；
2. 比较段号S和段表长度M，若S大于等于M，则产生越界中断；否则继续执行；
3. 该段的起始物理地址b = 段表起始地址F + 段号S * 每段包含的物理块数；
4. 物理地址E = b + W。

**段的共享与保护**

在分段系统中，段的共享是由两个作业的段表中相应表项指向被共享的段的同一物理副本来实现的。当一个作业正从共享的段中读取数据时，必须防止另一个作业修改此共享段中的数据。

分段管理的包含方法有两种：存取控制、地址越界保护。

## 段页式管理方式（二维）

在段页式系统中，作业的地址空间首先被分成若干个逻辑段，每段有自己的段号，然后将每一段分成若干个大小固定的页。对内存空间的管理仍和分页存储管理一样，以块为单位进行。

此时作业的逻辑地址分为三部分：段号S、页号P、页内偏移量W。为了实现地址转换，系统为每个进程建立了一个段表，而每个分段有一张页表。注意一个进程只有一个段表，但是可以有多个页表。段表项中包括段号、页表长度、页表起始地址，页表项中包括页号和块号。在进行地址转换时，首先通过段表查到页起始地址，然后通过页表找到对应的物理块号，最后形式物理地址。

# 4. 虚拟内存管理

## 虚拟内存的概念

传统存储管理方式具有以下两个特征：作业必须一次性全部装入内存，作业装入内存后就一直驻留在内存中。很多作业运行时根本不用的程序和数据占用了大量的内存空间，浪费宝贵的内存资源。

虚拟存储器基于程序局部性原理，操作系统将内存暂时不使用的内容换出到外存中，从而腾出空间存放将要调入内存的信息。主要有以下三个特征：多次性（作业允许被分成多次调入内存）、对换性（作业运行时无需常驻内存，进行换入换出）、虚拟性（从逻辑上扩充内存，即把外存虚拟成内存的一部分）。

### 虚拟内存技术的实现

虚拟内存的实现需要建立在离散分配的内存管理方式基础上。主要有以下三种方式：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。

## 请求分页管理方式

实现请求分页需要一定容量的内存和外存的计算机系统，以及页表机制、缺页中断机制和地址变换机制。

### 页表机制

请求分页系统的页表机制**不同于基本分页系统**。请求分页系统在一个作业运行之前不要求全部一次性调入内存，作业运行过程中，必然会出现要访问的页不在内存的情况。

发现和解决这种页面不在内存的情况是请求分页系统必须解决的两个基本问题。为此在请求分页页表项中增加了四个字段，页表项包括：页号、物理块号、状态位P、访问字段A、修改位M、外存地址。

- 状态位P：该页是否已调入内存。
- 访问字段A：该页在一段时间内被访问的次数，供置换算法参考。
- 修改位M：该页调入内存后是否被修改过。
- 外存地址：该页在外存上的地址，通常是物理块号。

### 缺页中断机制

当访问的页不在内存中时，便产生一个缺页中断，请求操作系统将所缺的页调入内存。缺页中断的中断处理程序会查找内存是否有空闲的块，若有空闲的块，则将所缺的页装入该块；若没有空闲的块，则需要替换掉某页。

缺页中断需要在指令执行中产生和处理中断信号，属于内部中断。且一条指令的执行期间，可能会产生多次中断。

### 地址变换机制

请求分页系统中的地址变换过程如下：

1. 但程序请求访问某一页时，首先判断页号是否越界，越界则产生越界中断；
2. CPU检索快表，如果对应页表项不在快表中，则检查慢表。
    - 如果找到了对应页表项，则通过页表项将该页的逻辑地址转换成物理地址，同时将该页表项放入快表，根据实际情况修改页表项的访问字段和修改位。
    - 如果没有找到对应的页表项，则产生缺页中断，执行缺页中断处理程序。

3. CPU保护现场。从外存中找到所缺的页，准备将其装入内存。
    - 如果此时内存未满，则将其直接装入内存，修改页表。
    - 如果此时内存已满，则需要进行页面替换，选择一个页面将其换出。如果换出的页面发生该修改，还需将其写回外存。
4. 缺页中断处理结束，CPU恢复现场，通过访问页表进程地址转换，同时将该页表项放入快表，修改页表项的访问字段和修改位等。

## 页面置换算法

**最佳置换算法OPT**

最佳置换算法选择以后永久不会再使用或者最长时间内不再访问的页，将其换出。这种算法可以保持最低的缺页率，但是目前无法预测哪个页面在未来不会被访问，因此该算法无法实现。

**先进先出FIFO页面置换算法**

优先换出最早进入内存的页面，也即在内存中驻留时间最长的页面。

FIFO算法会产生当所分配的物理块数（内存页框数）增大而缺页率不减反增的异常现象，称为Belady异常。

**最近最久未使用LRU置换算法**

根据页表项中的访问字段，选择过去一段时间内未访问过的页面，将其换出。

LRU性能较好， 但是需要寄存器和硬件的支持。LRU算法不会出现Belady异常。

**时钟CLOCK置换算法/最近未用NRU算法**

简单的CLOCK算法为每一页增加一个使用位，当该页调入内存或者被访问时，使用位都置1；操作系统将所有的候选替换页集合看作一个循环缓冲区，并有一个指针指向其中一页；当需要替换一页时，操作系统从指针指向的页开始，若该页使用位为1则将其置0，若该页使用位为0则将其替换，无论何种情况，指针会指向循环缓冲区的下一页；若该循环缓冲区中的所有页的使用位都为1，在一次循环后，起始位置的页的使用位已经变为0，将其替换即可。

CLOCK算法又称为最近未用算法，其性能接近LRU。

通过增加使用位的数目可以使得CLOCK算法更高效，在使用位u的基础上增加一个修改位m，得到**改进型CLOCK算法**。具体的操作步骤如下：

1. 操作系统从指针指向的页开始，选择(u = 0, m = 0)的页进行替换。这次扫描过程中不修改使用位和修改位；
2. 若上一步没有找到符合的页，则重新扫描，选择(u = 0, m = 1)的页进行替换，**同时将经过的页的使用位u改为0**；
3. 若上一步没有找到符合的页，则跳回步骤1。若步骤1仍未找到符合的页，则继续执行步骤2，此时一定有符合的页。

改进型CLOCK算法优先替换未被访问过的页，如果所有的页都被访问过，则替换未被修改过的页。

## 页面分配策略

当操作系统给一个进程的存储量越小，则驻留在主存中的进程数越多，可以提高处理机的时间利用率。但是一个进程在主存中的页数过少，缺页率会较高；页数过多，给该进程分配更多的主存空间也不会降低缺页率。

页面的分配现代操作系统通常采用以下三种策略：

- 固定分配局部置换：每个进程分配得的物理块数在运行过程中是不会改变的，若进程发生缺页，则从该进程的内存中替换出一页。这种策略很难确定一个进程所需的合适的物理块数目，太少会频繁缺页，太多会降低

主存利用率。

- 可变分配全局置换：为每个进程分配一定数目的物理块的同时，操作系统自身也保持一个空闲物理块队列。当某进程缺页时，系统从空闲物理块队列分配一个物理块给该进程，将需要的页装入。这种策略会盲目地为进程增加物理块，降低系统的多道程序并发能力。
- 可变分配局部置换：为每个进程分配一定数目的物理块的同时，操作系统自身也保持一个空闲物理块队列。当某个进程缺页时，从该进程的内存中替换出一页；只有一个进程频繁缺页时，系统从空闲物理块队列中分配一块给该进程使用，直至其缺页率趋于适当程度；若一个进程的缺页率特别低是，可以适当地减少分配给该进程的物理块。

**调入页面的时机**

- 预调页策略：一次调入若干个相邻的页可能比一次调入一页更加高效，前提是预测准确。目前预调页成功率越50%，故这种策略仅用于进程首次调入内存时，由程序员指出应该先调入哪些页。
- 请求调页策略：进程在运行中需要访问的页面不在内存中而提出要求，由系统将所需页面调入内存。目前虚拟存储器中大多使用此策略，进程运行过程中使用此策略。

两种调页策略同时使用，处于进程的不同时期。

**从何处调入页面**

请求分页系统的外存分为两部分：对换区和文件区。对换区采用连续分配，文件区采用离散分配，对换区的I/O速度比文件区快。

从何处调入页面有三种情况：

- 系统拥有足够的对换区：全部从对换区调入所需页面。为此，进程运行前需将与该进程相关的全部文件复制到对换区。
- 系统缺少足够的对换区：不会被修改的文件直接从文件区调入，可能修改的部分，换出时放入对换区，以后需要时再从对换区调入。
- UNIX方式：与进程相关的文件都在文件区，未运行过的页面，需要从文件区调入；曾经运行又被调出的页面，放入对换区，再次使用时从对换区调入。进程请求的共享页面若已被其它进程调入，则无须再从对换区调入。

# 四、文件管理

## 1. 文件系统概念

在系统运行时，计算机以进程为基本单位；在用户进程输入、输出中，以文件为单位。

文件是指由创建者所定义的一组信息的集合。文件包含如下属性：名称、标识符（唯一标签）、类型、位置（指针）、大小、保护（访问控制信息）、日期时间和用户标识等，所有这些信息都保存在目录结构中。

文件的基本操作有：

- 创建文件：分配空间，并在目录中创建条目，记录文件名称、文件所在位置和其它一些信息。
- 写文件：系统为每个文件维护一个写指针，指明写入的位置。
- 读文件：系统为每个文件维护一个读指针，指明读取的位置。
- 删除文件：删除文件的目录项，回收文件的存储空间。
- 截断文件：保持文件的属性不变，长度变为0。
- 文件打开与关闭：文件打开时将文件属性和磁盘地址表加载到内存中；文件关闭时首先将缓冲区的内容写入文件，后释放其内存。

文件的基本类型可以分为字符型文件和二进制文件。

### 文件逻辑结构

文件的逻辑结构是从用户观点出发看到的文件组织形式，与存储介质的特性无关。按照逻辑结构文件分为两种：

- 无结构文件：又称流式文件，以字节为单位，将数据按顺序组织成记录并保存。无结构文件对记录的访问只能通过穷举搜索的方式进行。
- 有结构文件：有结构文件按记录的组织形式可分为，
    - 顺序文件：文件中的记录一个接一个地顺序排列，每条记录可以是定长或者变长的，可以顺序存储或者以链表形式存储，访问记录时顺序搜索文件。
        - 串结构：记录按时间顺序排列。
        - 顺序结构：按找某个关键字的顺序排列。这种方式下增加和删除单个记录十分困难。
    - 索引文件：索引文件含有一张索引表，记录着每个的索引号、记录长度、指向该记录的指针。索引表本身是一个定长记录的顺序文件，通过索引可以成百上千倍地提高访问速度。
    - 索引顺序文件：将顺序文件的所有记录分成若干组，建立一个索引表记录每组中第一个记录的位置，即索引表项中包含指向每组第一个记录的指针。索引顺序文件提高了顺序文件的查找效率。
    - 直接文件或散列文件：给定记录的键值或通过Hash函数转换的键值直接决定记录的物理地址，这种文件没有顺序性。散列文件有很高的存取速度，但是回引起冲突，即不同关键字的散列函数值相同。

### 目录结构

目录在用户所需文件名和文件之间提高一种映射，实现文件按名存取；目录提高了对文件的检索速度，提供用于控制访问文件的信息。

#### 文件控制块

操作系统为实现目录管理，引入了文件控制块FCB的数据结构。文件控制块主要包含：

- 文件基本信息：文件名、文件物理位置、文件逻辑结构、文件物理结构等。

- 存取控制信息：文件的存取权限等。
- 使用信息：文件建立的时间、修改时间等。

文件控制块的有序集合称为文件目录，创建一个新文件时，操作系统将分配一个FCB并存放在文件目录中，成为目录项。

检索目录文件时只需使用文件名即可，不需要其它文件信息，因此有的操作系统（UNIX）将文件名和其它文件描述信息分开，文件描述信息单独形成一个称为索引结点的数据结构，简称 **i 结点**。此时文件目录中的每个目录项仅包含文件名和指向 i 结点的指针。

一个FCB的大小为64字节，大小为1KB的盘块可以存放16个FCB；UNIX中一个目录项仅为16字节，一个盘块中可以存放64个目录项。这样查找文件的时间可以减少到原来的1/4，大大节省了系统开销。

UNIX系统的**索引结点包含**：

- 文件主标识符：拥有该文件的个人或小组的标识符。
- 文件类型：普通文件、目录文件、特别文件。
- 文件存取权限：各类用户对该文件的存取权限。
- 文件物理地址：数据文件所在盘块的编号。
- 文件长度
- 文件存取时间：本文件最近被进程存取的时间、最近被修改的时间、索引结点最近被修改的时间。
- 文件被打开：磁盘索引结点复制到内存中以便于使用。

内存索引结点又增加了以下内容：

- 索引结点编号
- 状态：是否上锁或被修改。
- 访问计数
- 逻辑设备号
- 链接指针：设置分别指向空闲链表和散列队列的指针。

**目录结构**

目录的操作：搜索、创建文件、删除文件、显示目录、修改目录。

以下几种目录结构：

- 单级目录结构

  整个文件系统中只建立一张目录表，每个文件占用一个目录项。

  访问一个文件时，先按文件名在目录表中找到相应的FCB，经合法性检查后执行相应的操作。创建一个文件时，先检索所有目录项，确保没有重名，然后在该目录表中增设一项，把FCB的全部信息存入该项。删除一个文件时，先从目录表中找到该文件的目录项，回收该文件所占的存储空间，然后清除该目录项。

- 两级目录结构

  将文件目录分为主文件目录MFD和用户文件目录UFD，主文件目录项记录用户名和相应用户文件目录所在的存储位置；用户文件目录项记录用户文件的FCB信息。这种结构解决了不同用户的同名文件的重名问题，在目录上实现了不同用户的访问限制，提高了文件的安全性。

- 多级目录结构

  将两级目录结构的层次关系加以推广，形成多级目录结构，即树形目录结构。由“/”分隔符连接的绝对路径或相对路径标识文件。

  树形目录结构可以很方便地对文件进行分类，更有效地进行文件的管理和保护。但是查找文件需要逐级访问中级结点，增加了磁盘开销。

- 无环图目录结构

  树形目录结构不适合实现文件共享，在目录基础上增加一些指向同一结点的有向边，此时此目录结构形成一个有向无环图。

  无环图目录结构方便实现文件的共享，但使得系统的管理变得更加复杂。

## 文件共享

以下的硬链接和软链接都属于静态链接。两个进程同时对一个文件进行操作，这样的共享称为动态链接。

**基于索引结点的共享方式（硬链接）**

将文件的物理地址及其它文件属性等信息放入索引结点中，在目录项中只设置文件名和指向该文件索引结点的指针，只要将该目录项放入不同用户的目录即可实现共享。同时在索引结点中设置一个链接计数count，用于统计链接到本索引结点上的用户目录项的数目。每当一个用户删除该文件时，count减1，只有count减至0时，系统才完全删除该文件。

**利用符号链实现文件共享（软链接）**

为实现某个文件的共享，由系统创建一个LINK类型的新文件，与需要共享文件的文件名相同。新文件被链接到其它共享用户的目录中，其内容只有共享文件的路径名。这样的链接称为符号链接，新文件中的路径名称为符号链。

当文件拥有这删除文件后，其它用户使用符号链接访问该文件会发生错误，因为只有文件拥有者才有指向文件的指针，其它用户只拥有路径名。但是如果其它用户在此目录上创建了同名文件，符号链接又能够使用了，只不过文件已不是原来那个文件，也可能引发错误。

使用符号链实现文件共享，操作系统需要先读取LINK文件，才能对共享文件操作，增加了磁盘的开销。

硬链接的查找速度比软链接快。

## 文件保护

文件保护通过口令保护、加密保护、访问控制等方式实现。

现代操作系统将访问控制列表与用户、组和其它成员访问控制方案一起使用。多级目录结构需要保护子目录，这与文件保护的机制时不同的。

**访问控制**

访问类型：读、写、执行、添加（在文件结尾添加新内容）、删除、列表清单（列出文件名和属性）、重命名、复制、编辑等。

访问控制最常用的方法是根据用户身份进行控制，而实现基于身份访问的最普通的方法是为每个文件和目录增加一个访问控制列表ACL，以规定每个用户及其所允许的访问类型。

精简的访问列表采用拥有者、组、其它用户三种用户类型，只需要用三个域来列出这三种用户的访问权限即可。UNIX系统采用此方法。

**口令**

口令指用户建立一个文件时提供的一个口令，系统为其建立FCB时附上相应口令，同时告诉允许共享该文件的其它用户。用户请求访问时必须提供相应口令。缺点是口令之间存储在系统内部不过安全。

**密码**

密码指用户对文件进行加密，访问文件时需要使用密钥。这种方法保密性强，但是需要花费解密和加密的时间。

# 2. 文件系统实现

现代操作系统的文件系统类型：FAT32、NTFS、ext2、ext3、ext4等。

## 文件系统的层次结构

1. 用户调用接口：文件系统为用户提供的与文件和目录有关的调用。
2. 文件目录系统：管理文件目录，主要功能有管理活跃文件目录表、管理读写状态信息表、管理用户进程的打开文件表、管理和组织在存储设备上的文件目录结构、调用下一级存取控制模块。
3. 存取控制验证：将用户的访问要求与FCB中的访问控制权限进行比较，以确认访问的合法性。
4. 逻辑文件系统与文件信息缓冲区：将用户读写的逻辑记录转换成文件逻辑结构内的相应块号。
5. 物理文件系统：将上述相应块号转换成实际的物理地址，即逻辑地址转换为物理地址。
6. 分配模块：分配和回收辅存空间。
7. 设备管理程序模块：分配设备、分配设备读写缓冲区、磁盘调度、启动设备、处理设备中断、释放设备读写缓冲区、释放设备等。

## 目录实现

目录实现的基本方法有线性列表和哈希表两种，线性列表实现对应线性查找，哈希表实现对应散列查找。

目录查询需要大量的I/O操作，因此操作系统将当前使用的文件目录复制到内存，以后使用该文件目录只要在内存中操作，降低了磁盘的操作次数，提供了系统速度。

#### 线性列表

使用存储文件名和数据块指针的线性表实现。创建新文件时必须搜索目录确认没有同名文件，然后在目录表后增加一个目录项。删除文件时根据给定的文件名搜索目录表，释放分配给该文件的存储空间，可以将目录项标记为不再使用，或者将它移动到空闲目录项表上。

线性列表的优点是实现简单，但是使用较为费时。

#### 哈希表

根据文件名获得一个值，并返回一个指向线性列表中元素的指针。这种方法的优点是查找迅速，插入和删除也比较简单，但是需要一些预备措施来避免冲突。

## 文件实现

#### 文件分配方式（对非空闲空间的管理）

文件分配方式常用的有三种：连续分配、链接分配、索引分配。

1. 连续分配：每个文件在磁盘上占用一组连续的块，支持顺序访问和直接访问。

   连续分配实现简单，存取速度快。缺点是文件长度很难动态增加，一旦增加就需要大量移动盘块；且反复增删文件会产生外部碎片。因此连续分配方式适合固定长度的文件。

2. 链接分配：分为隐式链接和显示链接两种。
   - 隐式链接：每个文件对应的都是一个磁盘块组成的链表，具体为每个磁盘块都有一个指向下一个磁盘块的指针，这些指针对用户透明。目录项包含文件第一块的指针和最后一块的指针。

     隐式链接的缺点是无法直接访问磁盘块，必须通过指针顺序访问，速度慢；每个盘块的指针占用了存储空间；一旦其中的一个指针丢失，则整个文件丢失。
   - 显示链接：为整个磁盘建立一张显示的链接表，称为文件分配表，保存了磁盘上每一个文件的所有链接指针，文件目录项中有指向该文件物理块指针链的头指针。

3. 索引分配：系统将文件的所有盘块的盘块号集中放在一起组成一张**索引块**，每个文件都有索引块，是一个磁盘块地址的数组，一般为一物理块大小。创建文件时，文件索引块的所有指针都为空，当写入某物理块时，将其地址写入索引块的对应条目。索引分配支持直接访问，且没有外部碎片。其缺点是索引块的分配增加了系统开销。

   由于每个文件必须有一个索引块，索引块的大小应尽可能小，以减少存储空间占用，但是索引块太小就无法支持大文件。可以用以下机制解决：

   - 链接方案：通过物理块指针将多个索引块链接起来，使一个文件支持更多的索引条目。
   - 多层索引：即建立指向索引块的索引块，为支持更大的文件可以建立多层索引。
   - 混合索引：将多种索引方式相集合的分配方式。

**文件存储空间管理（对空闲空间的管理）**

文件存储空间的划分和初始化：一般来说一个文件存储在一个文件卷中，文件卷分为目录区和文件区，目录区存放文件控制信息FCB，文件区存放文件数据信息。现代操作系统存在很多不同的文件管理模块，以应对不同的文件格式的逻辑卷。文件卷在提供文件服务前，必须由对应的文件程序进行初始化，划分好目录区和文件区，建立空闲空间管理表格及存放逻辑卷信息的超级块。

文件存储管器的空间管理实质上是对空闲块的组织和管理，包括空闲块的组织、分配与回收等问题。

1. 空闲表法

   空闲表法属于连续分配方式，为每个文件分配一块连续的存储空间。系统为外存上的所有空闲区建立一张空闲盘块表，每个空闲区对应一个空闲表项，其中包括表项序号、该空闲区第一个盘块号、该空闲区的盘块数量等信息；空闲表项按照其起始盘块号递增的顺序排列。

   空闲盘区的分配和内存的动态分配类似，同样采用首次适应算法、循环首次适应算法等。系统回收用户所释放的存储空间，也采用类似内存回收的方法。

2. 空闲链表法

   将所有空闲盘区形成一条空闲链表，可分为空闲盘块链和空闲盘区链。空闲盘块链以空闲盘块为单位，用户创建文件请求分配存储空间时，系统从链首开始，一次摘下适当数目的盘块分配给用户；删除文件时，系统回收存储空间插入链尾。空闲盘区链以空闲盘区为单位，每个空闲盘区包含一个指向下一个空闲盘区的指针和本盘区的大小，分配盘区的方法和动态分区分配类似，通常采用首次适应算法；回收盘区时，将回收区与相邻的空闲盘区合并。

3. 位示图法

   位示图（二维表）利用二进制位来表示磁盘中一个盘块的使用情况，磁盘上所有的盘块都有一个二进制位与之对应。

   盘块的分配：顺序扫描位示图，从中找出一个或一组其值为0的二进制位；将所找到的二进制位转换成相应的盘号。修改位示图中分配盘区的对应二进制位为1。

   计算盘块号的方法：位示图中的行号为$i$（0~n），列号为$j$（0~n），盘块号为$b = n(i-1)+j$。

   盘块的回收：将回收盘块号转换为位示图中的行号和列号；修改位示图。

   计算位示图行列号的方法：$i = (b-1) \text{ div } (n+1), j = (b-1) \bmod (n+1)$。

4. 成组链接法

   空闲表法和空闲链表法都不适用于大型文件系统，在UNIX系统中采用成组链接法，这种方法结合了空闲表法和空闲链接法，其大致思想是：将顺序的n个空闲扇区（1~n）地址保存在第0个空闲扇区内，其后第n个空闲扇区内则保存另一组顺序空闲扇区（n+1~2n）的地址，如此类推，直至所有空闲扇区均予以链接。系统只需要保存一个指向第0个空闲扇区的指针即可，这种方式可以迅速找到大批空闲块地址。

表示文件存储器空闲空间的位向量表或第一个成组链块以及卷中的目录区、文件区划分信息都需要存放在辅存储器中，一般放在卷头位置，在UNIX系统中称为超级块。在对卷中文件操作前，需要预先将超级块读入系统空闲的主存，并且经常保持主存中超级块与辅存卷中的超级块的一致性。

# 3. 磁盘组织与管理

## 磁盘的结构

磁道、柱面、扇区（盘块）。磁盘的地址用“柱面号-盘面号-扇区号”表示。

## 磁盘调度算法

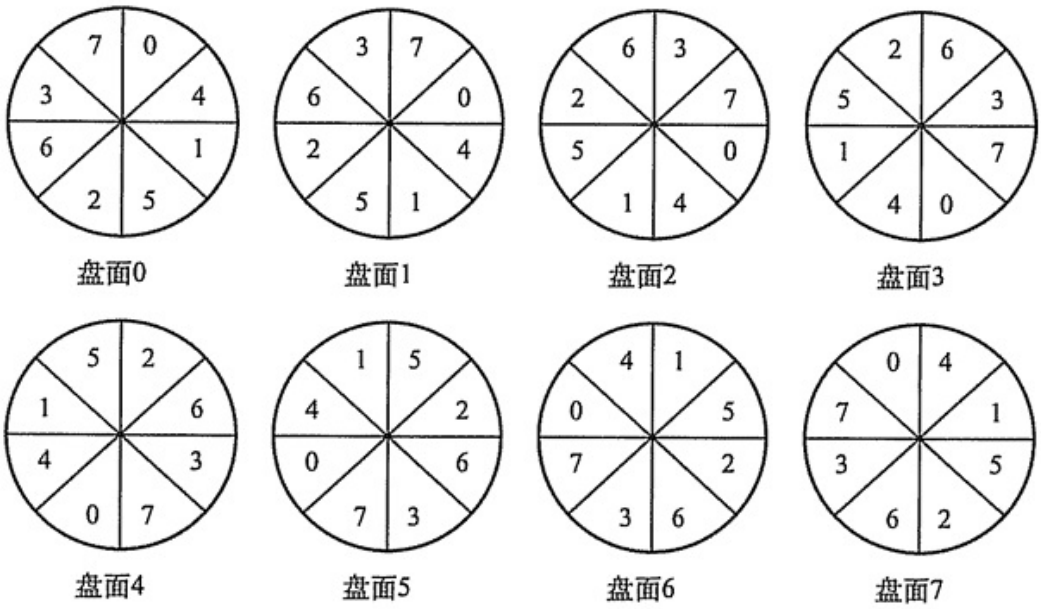
### 读写时间

- 寻找时间/寻道时间$T_s$：磁头移动到指定磁道所需时间，包含跨越$n$条磁道的时间和磁臂启动时间$s$，即$T_s = m \times n + s$。其中$m$与磁盘驱动器速度有关，约为0.2ms；磁臂启动时间约为2ms。
- 延迟时间$T_r$：磁头找到磁道后定位到指定扇区的时间，与磁盘的转速$r$有关，平均为$T_r = \frac{1}{2r}$。
- 传输时间$T_t$：从磁盘读出或向磁盘写入数据所经历的时间，取决于每次所读写的字节数$b$和磁盘的转速，设每个磁道上的字节数为$N$，则$T_t = \frac{b}{rN}$。
- 总平均存取时间$T_a$为：$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$。

### 调度算法

- 先来先服务FCFS算法：FCFS算法根据进程访问磁盘的先后顺序进程调度。该算法是一种最简单的调度算法，具有公平性；但大量进程竞争使用磁盘时，这种算法性能上接近于随机调度。
- 最短寻找时间优先（SSTF）算法：SSTF算法优先处理磁道与当前磁头所在磁道最近的访问请求，使访问时间最小。但是这并不难保证平均寻找时间最小，且当某磁道附件的访问请求频繁时远的磁道就长时间得不到访问，造成饥饿现象。
- 扫描（SCAN）算法：又称为电梯算法，SCAN算法优先处理当前磁头移动方向上距离最近的磁道的访问请求，实际就是在SSTF算法上增加了方向因素。SCAN算法对最近扫描过的区域不公平（再近也不回头），在访问局部性上不如FCFS和SSTF算法。
- 循环扫描（C-SCAN）算法：在SCAN算法的基础上增加了磁头单向移动的限定。磁头当移动到一端后直接快速返回起始端，实际中改进为磁头移动到最远的请求，完成该请求后立即返回起始端，而不是到磁盘端点，这种改进的算法称为LOOK算法和C-LOOK算法。

由于**磁头**在读写一个物理块后，需要经过短暂的处理时间才能开始下一块的读写，若扇区连续编号，则连续读写多个扇区时无法避免磁头的处理时间。因此可以对盘面扇区进行**交替编号**，对盘片组中的不同盘面**错位命名**，不同盘面的磁头交替工作，连续访问一系列扇区时可以避免磁头的处理时间降低访问速率。




## 磁盘的管理

- 磁盘初始化：
  - 低级格式化：将空白盘分成扇区，每个扇区的数据结构有头、数据区域、尾部组成。
  - 逻辑格式化：创建文件系统，包括空闲和已分配的空间、初始为空的目录。
- 引导块：计算机启动时运行自举程序，它初始化CPU、寄存器、设备控制器、内存等，接着启动操作系统。自举程序需要找到磁盘上的操作系统内核，装入内存，并转到操作系统的起始地址，从而开始操作系统的运行。

  自举程序一般在ROM中保存很小的自举装入程序，完整功能的自举程序保存在磁盘的启动块上，拥有启动块的磁盘称为启动磁盘或系统磁盘。
- 坏块：磁盘中会产生坏的扇区，简单的磁盘会在逻辑格式化时将坏块表明，不再使用。复杂的磁盘则会在低级格式化时初始化一个磁盘坏块链表，并在磁盘的使用过程中不断更新；同时低级格式化时将一些扇区保留下来，对操作系统透明，以后使用时替换坏掉的扇区，这种称为扇区备用。

# 五、I/O管理

## 1. I/O管理概述

### I/O设备

按使用特性可以分为人机交互类外部设备、存储设备、网络通信设备。按传输速率分为低速设备（鼠标键盘）、中速设备（打印机）、高速设备（磁盘）。按信息交换的单位分类分为块设备、字符设备。

### I/O控制方式

- 程序直接控制方式

  CPU循环检查I/O设备的状态，这种方式下造成了CPU资源的浪费。

- 中断驱动方式

  允许I/O设备主动打断CPU的运行并请求服务，使得CPU无须浪费资源来监控I/O设备的状态。

- DMA方式

  高速I/O设备与内存直接建立数据通路，以块为单位交换数据，解放CPU。CPU只需启动DMA和DMA结束时参与，节省了CPU资源。

- 通道控制方式

  I/O通道指专门负责输入输出的处理机，CPU需要进程I/O操作时，只需要向I/O通道发送一条I/O指令即可。

### I/O子系统的层次结构

整个I/O系统可以看成具有四个层次的系统结构，各层次如下：

- 用户层I/O软件：实现与用户交互的接口，用户可以直接调用在用户层提供的、与I/O操作有关的库函数。
- 设备独立性软件：用于实现用户程序与设备驱动器的统一接口、设备命令、设备保护、以及设备分配与释放等，同时为设备管理和数据传递提供必要的存储空间。

  为了实现设备独立性而引入了逻辑设备和物理设备两个概念，在应用程序中使用逻辑设备名来请求使用某类设备；系统实际执行中必须将逻辑设备名映射成物理设备名使用。使用逻辑名的好处是增加设备分配的灵活性，易于实现I/O重定向（更换I/O设备，但不必改变应用程序）。

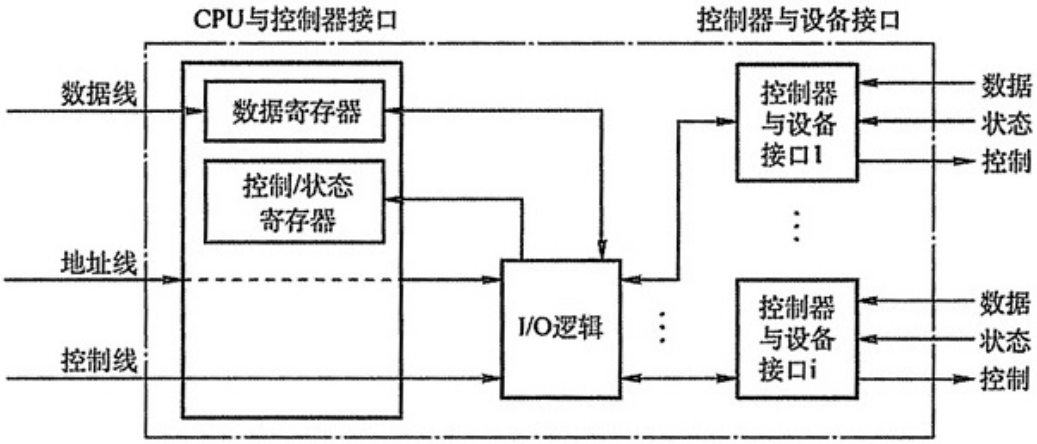
  设备独立性软件主要功能可分为以下两个方面：执行所有设备的公有操作；向用户层或文件层提供统一接口。

  设备驱动程序：与硬件直接相关，负责具体实现系统对设备发出的操作命令，驱动I/O设备工作的驱动程序。
- 中断处理程序：用于保存中断进程的CPU环境，转入相应的中断处理程序进程处理；处理完并恢复被中断进程的现场后，返回被中断的进程。
- 硬件设备：硬件设备包括设备本身和设备控制器，设备控制器的主要功能为：
  - 接收和识别CPU或通道发来的命令。
  - 实现数据交换，包括设备和控制器之间的数据交换；通过数据总线或通道，控制器和主存之间的数据传输。
  - 发现和记录设备及本身的状态信息。
  - 设备地址识别。

为实现上述功能，设备控制器包含以下组成部分：

- 设备控制器与CPU的接口：三类信号线-数据线、地址线和控制线；两类寄存器-数据寄存器和控制/状态寄存器。
- 设备控制器与设备的接口。
- I/O控制逻辑：对从CPU收到的I/O命令进行译码。

具体结构如下：




# 2. I/O核心子系统

I/O子系统是操作系统内核的一部分，提供的服务主要有：I/O调度、缓冲与高速缓冲、设备分配与回收、假脱机、设备保护和差错处理等。

## I/O调度

I/O调度就是确定一个好的顺序来执行I/O请求，以改善系统的整体性能，使进程之间公平地共享设备访问。操作系统为每个I/O设备维护一个请求队列来，通过重新安排队列的顺序以改善系统总体效率和应用程序的平均响应时间。

## 高速缓存与缓冲区

### 磁盘高速缓存

操作系统通过使用磁盘高速缓存技术来提高磁盘的I/O速度，具体是指利用内存中的存储空间来暂存从磁盘中读出的一系列盘块中的信息，其逻辑上属于磁盘，物理上属于内存。

### 缓冲区Buffer

缓冲区的主要目的：

- 缓和CPU和I/O设备间的速度不匹配的矛盾。
- 减少对CPU的中断频率，放宽对CPU中断响应时间的限制。
- 解决基本数据单元大小（数据粒度）不匹配的问题。
- 提供CPU和I/O设备之间的并行性。

实现方法有：

- 采用硬件缓冲器。

- 采用位于内存中的缓冲区。

根据系统设置的缓冲区的个数，缓冲技术可以分为：

- 单缓冲

  在设备和处理机之间设置一个缓冲区，两者之间交换数据时，先将数据放入缓冲区然后需要数据的设备或处理机从缓冲区取走数据。

- 双缓冲

  在设备和处理机之间设置两个缓冲区，两个缓冲区交替工作，避免了传送大量数据时的等待时间。两个缓冲区还可以实现双向数据传输。

- 循环缓冲

  多个大小相等的缓冲区组成一个循环的链表，设备和处理机交换数据时，缓冲区链表按顺序使用，避免了设备等待缓冲区变空的时间。

- 缓冲池

  由多个公用的缓冲区组成，按其使用情况可分为三个队列：空缓冲队列、装满输入数据的缓冲队列、装满输出数据的缓冲队列；还有四种缓冲区：用于收容输入数据的工作缓冲区、用于提取输入数据的工作缓冲区、用于收容输出数据的工作缓冲区、用于提取输出数据的工作缓冲区。

  当输入进程需要输入数据时，从空闲缓冲队列的对首摘取一个空缓冲区，把它作为收容输入数据的工作缓冲区，然后把输入数据放入其中，装满后再将它挂到输入队列队尾。其它的情况类似。

## 设备分配与回收

### 设备分配

从设备的特性来看，设备分配采用以下三种方式：

- 独占式使用设备：对应独占设备，一次只允许一个进程使用，等该设备被释放后才允许其它进程申请使用。如打印机。
- 分时式共享使用设备：对应共享设备，通过分时共享提高设备的利用率。如磁盘。
- 以SPOOLing方式使用外部设备：对应虚拟设备，SPOOLing技术为假脱机I/O技术，实质就是对I/O操作进行批处理。

设备分配相关的**主要数据结构**有设备控制表DCT、控制器控制表COCT、通道控制表CHCT、系统设备表SDT。

- DCT：一个设备控制表代表一个设备，其中的表项就是设备的属性。每个设备控制表中都有一个表项存放指向对应控制器控制表的指针。
- COCT：采用通道控制方式的系统中，每个控制器控制表都有一个表项存放指向相应通道控制表的指针。
- CHCT：一个通道可以为多个设备控制器服务，因此通道控制表中都有一个指针指向一个表，这个表上记录了该CHCT提供服务的设备控制器。
- SDT：整个系统只有一个系统设备表，它记录了已链接到系统中的所有物理设备的情况。

设备**分配方式**分为静态分配和动态分配。静态分配指作业开始工作前，系统将其申请的所有设备、控制器一次性全部分配给该作业，一旦分配这些设备、控制器就一直为该作业所有，直到作业被撤销。动态分配是进程在执行过程中根据需要申请分配，用完后立即释放；常用的动态设备分配算法有先请求先分配、优先级高者先分配等。

静态分配不会造成死锁，动态分配会造成死锁。

设备分配的**安全性**指设备分配中应防止发生进程死锁。安全的分配方式是每当进程发出I/O请求后便进入阻塞状态；不安全的分配方式是进程发出I/O申请后继续执行，需要时由发出第二个、第三个请求等。

为了实现设备的独立性，在应用程序中使用逻辑设备名请求某类设备，在系统中设置一张逻辑设备表LUT，用于将逻辑设备名映射成物理设备名。逻辑设备表中每个表项包含逻辑设备名、物理设备名和设备驱动程序入口地址。系统中可以采用两种方式建立逻辑设备表：在整个系统中设置一张逻辑设备表、为每个用户设置一张逻辑设备表。

## SPOOLing技术

SPOOLing技术是一种将独占设备改造成共享设备的技术。

在磁盘上设置两个存储区域：输入井和输出井。输入井模拟脱机输入时的磁盘，用于收容I/O设备输入的数据；输出井模拟脱机输出时的磁盘，用于收容用户程序的输出数据。

在内存中设置两个缓冲区：输入缓冲区和输出缓冲区。输入缓冲区用于暂存输入设备送来的数据，以后再传送到输入井；输出缓冲区用于暂存输出井送来的数据，以后再传送到设备。

输入进程将用户要求的数据从输入机通过输入缓冲区再送到输入井，当CPU需要输入数据时，从输入井直接读入内存。输出进程把用户要求输出的数据先从内存送到输出井，待输出设备空闲，再将输出井中的数据通过输出缓冲区送到输出设备。

SPOOLing技术提高了I/O的速度，将独占设备改造成共享设备，实现了虚拟设备功能。