--最后的胜利(1.2-1.8)

Header

期末考时搜集整合几篇博客+笔记而成，深度的知识区折叠了

删除了部分考试章节中没有的部分，同时偷懒了一点，毕竟目的是考试

由于相比于上一次计算机组成原理时间紧迫得很(只有5天复习)，因此部分内容只能等到日后复盘整理

我想指出的是，这里为了拟合期末复习的知识点作了大量的删节，同时也发现大佬作的总结更加完善

发现自己所学不过是OS的冰山一角，大量的知识点被无情的删除了，仅仅因为期末考不考

呜呼！仅仅为了通过考试的作品完全称不上良好，以下内容仅供参考

口述考点

时间短，题目精

50分卷面及格，其中30分讲过了，20分我这个笔记里面有

进程同步不会很难，吃水果和抽烟变形，加上读者优先

标注略的大概是不考的题目

1OS引论

基本信息

定义

操作系统是控制和管理计算机硬件和软件资源、合理地组织和管理计算机的工作流程以方便用户使用的程序的集合.

发展目标

1. 方便性
2. 有效性
    - 提高系统资源利用率
    - 提高系统吞吐量
3. 可扩充性
4. 开放性

地位

操作系统为建立用户与计算机之间的接口，为裸机配置的一种系统软件

操作系统是裸机上的第一层软件，位于系统硬件之上，所有其它软件之下（是其它所有软件的共同环境）

历史

- 未配置OS的计算机系统
    - 人工操作方式

      用户独占全机 CPU等待人工操作 严重降低了计算机资源的利用率
    - 脱机输入/输出(Off-Line I/O)方式

      减少了CPU的空闲时间 提高了I/O速度 效率仍然不理想
- 批处理

    - 单道批处理系统
        - 自动性、顺序性、单道性
    - 多道批处理系统
        - 多道、宏观上并行、微观上串行．资源利用率高，系统吞吐量大；但是用户响应时间长，无人机交互能力
- 分时
    - 时间片轮转；同时性、交互性、独立性、及时性
    - 若干用户联机操作
- 实时
    - 在事先定义好的时间内对事件进行响应处理；及时性、可靠性

人工操作方式

手工把纸带装入纸带输入机，启动输入机输入计算机，启动计算机计算，取走结果。用户即使操作员，又是操作员。该操作方式下计算机工作特点 ：

(1) 用户独占全机，资源利用率低

(2) CPU等待人工操作，CPU利用率低

批处理阶段

为了提高计算机的利用率，人们提出了批处理的概念。批处理是指把一批作业按照一定的顺序排好队，计算机按照一定的顺序自动运行，直到运行完所有的作业为止。

- 单道批处理系统 (Simple Batch Processing)

    > 也有资料称之为脱机批处理，脱机输入/输出

    把系统的输入/输出工作交给一台外围计算机。用户把程序和数据输入到外围计算机，外围计算机把输入的数据和程序转存到磁带上，然后把磁带送到主机，主机把程序和数据装入内存，执行程序，把结果送到磁带上，再由外围计算机把磁带上的结果输出到打印机上。

    提高了CPU利用率和输入输出的速度。但是，用户仍然需要等待作业完成才能获得结果，CPU仍然需要等待人工操作。

- 多道批处理系统（Multi-Programming Batch Processing）

  多道程序设计技术的出现给计算机系统的管理带来了巨大挑战，如：内存共享与保护问题，处理器调度问题，I/O设备的共享及竞争问题等，为解决这些问题，形成了现代操作系统的雏形。所以说多道程序设计技术是操作系统形成的标志。

  多道程序在内存中同时驻留，它们共享CPU和其他资源。当一个作业需要等待某事件发生时，CPU可以立刻切换到另一个作业上去执行，从而提高了CPU的利用率。

  但是，多道程序环境下，程序的执行是异步的，程序的执行速度不可预知，因此，需要采用某种措施来保证程序的执行顺序。

  为了保证程序的执行顺序，需要引入作业调度和内存管理两个概念。

分时操作系统（Time-Sharing System）

分时操作系统是一种允许多个用户通过终端同时连接到一台计算机上的操作系统。分时系统的基本特征是：多路性、独立性、及时性、交互性。

分时系统的基本原理是：把CPU的时间分成很多个时间片，每个时间片分配给一个用户，用户在这个时间片内独占CPU。由于CPU的时间片很短，用户感觉好像计算机在同一时间为他服务一样，这就是分时系统的基本原理。

分时系统的出现，使得用户不再需要等待作业完成才能获得结果，而是可以实时地与计算机交互。

分时系统不能优先处理一些紧急任务，因此引入了实时操作系统。

实时操作系统（Real-Time System）

在限定时间内完成某些紧急任务，而不需要时间片排队。

实时操作系统分为硬实时操作系统和软实时操作系统。

- 硬实时操作系统：某个动作必须绝对地在规定的时间内完成，否则就会产生严重的后果。例如：导弹控制系统、核反应堆控制系统等。
- 软实时操作系统：某个动作必须在规定的时间内完成，但是如果不能完成，后果并不严重。例如：飞机订票系统，银行管理系统等。

特性原理

特征

并发和共享是多用户(多任务)OS的两个最基本的特征。它们又是互为存在的条件

- 并发
  - 区别并行和并发

    并发是进程宏观一起运行，微观上交替运行，而并行是指同时运行
- 共享
  - 1.互斥共享方式（互斥共享的资源称为临界资源）
    - 一段时间内只允许一个进程使用
  - 2.同时访问方式（一段时间内允许多个进程同时访问）
    - 允许一个时间段由多个进程同时（宏观）访问，微观上是交替访问
- 虚拟

  把物理实体变为若干个逻辑的对应

  两种虚拟技术
  - 时分复用技术
  - 空分复用技术
- 异步

  进程的执行走走停停，以不可预知的速度向前推进

运行机制

另见CPU

时钟管理

帮助处理进程

中断

广义中断 = 异常(内部) + 狭义中断(IO/外部)

- 外中断

  > 由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

- 异常

  > 由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

系统调用

从用户态转换到核心态，这是由硬件完成的

输入输出必然在核心态执行

导致用户从用户态切到内核态的操作有：某个东西导致了异常中断、I/O等等

在用户程序中使用系统调用。如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。

系统调用是指用户自编程序通过操作系统提供的接口，向操作系统发出服务请求，从而获得操作系统的服务。是用户态到核心态的唯一入口。

体系结构模型

略

- 微内核 (micro kernel)

> 基本思想：内核尽可能小，只实现操作系统的基本功能；将更多的操作系统功能放在核心之外，作为独立的服务进程运行。
>
> 其特点是：内核简单，安全可靠，可移植性好。

- 单内核（monolithic kernel）

> 基本思想：将核心分为若干个模块，模块间的通信通过调用其它模块中的函数来实现。
>
> 其特点为：执行效率高，但可移植性相对减弱。

2硬件

CPU

构成

- 运算器
- 控制器
- 寄存器

> 重要寄存器
>
> - PC
> - IR
> - PSW

状态

- 特权指令：只能由操作系统内核程序运行的指令，例如：关中断、修改时钟中断向量等。
- 非特权指令：用户自编程序可以运行的指令，例如：加法、减法、乘法、除法等。

在具体实现上，根据程序的不同，CPU所处状态分为核心态和用户态。

- 核心态：又称为管态、内核态。CPU执行操作系统内核程序时所处的状态，此时CPU可以运行特权指令。
- 用户态：又称为目态。CPU执行用户自编程序时所处的状态，此时CPU不可以运行特权指令。

在CPU中设置程序状态寄存器（PSW）来标志处理器当前处于的状态。

CPU处于核心态可以调用除了访管指令以外的所有指令

> 从用户态转到核心态会用到访管指令. 访管指令在用户态使用，所以它不可能是特权指令，也不可能是管态(核心态)使用的

功能

取指令然后执行

取值周期 <-> 执行周期

存储器体系结构

程序的存储器访问局部性原理　　能提高存储器体系效能的关键

金字塔结构

- 寄存器

- Cache
  - 内存
    - 外存(硬盘)
      - 磁带、光盘...

3进程

程序执行特征

顺序执行

- 顺序性
- 封闭性
- 可再现性

并发执行

- 间断性
- 失去封闭性
- 不可再现性

定义

进程是进程实体的运行过程

在系统中能独立运行并作为资源分配的基本单位

基本特征

基本特性是动态、并发

- 并发
  - 多个进程实体同存于内存中，且在一段时间内同时运行。
- 动态
  - 进程的实质是进程实体的一次执行过程，因此，动态性是进程最基本的特征。
- 独立
  - 进程实体是一个独立运行、独立分配资源和独立接受调度的基本单位。
- 异步
  - 各进程是按各自独立的、不可预知的速度向前推进的

分类

从操作系统角度

- 系统进程
- 用户进程

组成

PCB	Process Control Block，**进程控制块。包含着进程的描述和控制信息，是进程存在的唯一标志**
程序	“纯代码”部分，也称为“文本段”，描述了进程要完成的功能，是进程执行时不可修改的部分

PCB	**Process Control Block，进程控制块。包含着进程的描述和控制信息，是进程存在的唯一标志**
数据	进程执行时用到的数据（全局变量，静态变量，常量）
工作区	即堆栈区，用于内存的动态分配，保存局部变量，传递参数、函数调用地址等

和程序区别

进程是程序的一次执行(动态)

程序是完成任务功能的指令的有序序列(静态)

进程具有并行特征（独立性、异步性），程序则没有

基本状态

进程的三种基本状态

- 就绪状态ready
- 执行状态running
- 阻塞状态block

(展开的话五种)

①创建态。进程刚被创建时的状态，尚未进入就绪态。

②就绪态。进程获得了除处理机外的所有资源，只需获得处理机就可以运行，如果多个进程处于就绪态，则它们放在就绪队列中。

③运行态。进程获得处理机运行。在单处理机同一时刻只能有一个进程在处理机上运行

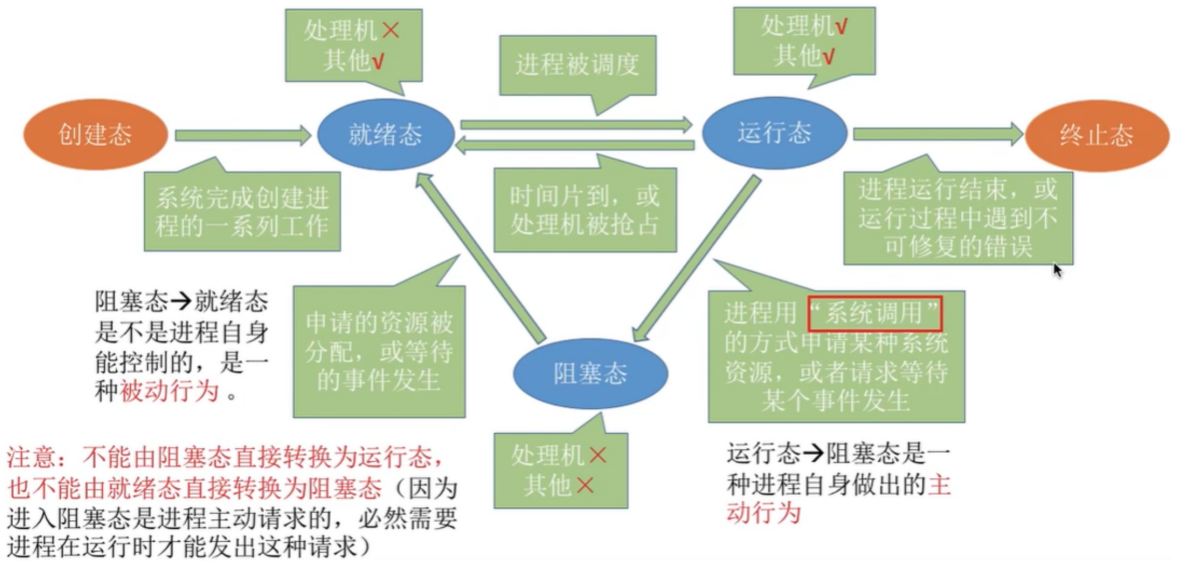
④阻塞态。进程正在等待某一资源而暂停运行的状态，如等待某资源或等待输入输出完成，如果有多个阻塞进程，则将他们存入阻塞队列中。

⑤结束态。进程正在从系统中消失，可能正常结束，也可能被动结束。系统需要将该进程置为结束态回收相关资源。

状态转换

- 就绪状态 -> 运行状态：分配到CPU
- 运行状态 -> 就绪状态：**时间片用完**，或被更高优先级的进程抢占CPU
- 运行状态 -> 阻塞状态：**等待某事件发生**，例如需要分配资源，申请I/O操作，但系统暂时不能进行分配
- 阻塞状态 -> 就绪状态：等待的事件发生

> 阻塞态->运行态和就绪态->阻塞态这二种状态转换不可能发生



补充

“挂起”和“阻塞”的区别

两种状态都是暂时不能获得CPU的服务，但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中

控制原语

- create原语：创建新进程
- 撤销原语：终止进程
- 阻塞原语block：将进程由运行态转变为阻塞态，是由被阻塞进程自我调用实现的
- 唤醒原语：将进程从等待队列中移出，并置其状态为就绪态，是由一个被唤醒进程合作或被其他相关的进程调用实现的

学习技巧:进程控制会导致进程状态的转换。无论哪个原语，要做的无非三类事情:

1．更新PCB中的信息（如修改进程状态标志、将运行环境保存到PCB、从PCB恢复运行环境）

a.所有的进程控制原语一定都会修改进程状态标志

b.剥夺当前运行进程的CPU使用权必然需要保存其运行环境

c.某进程开始运行前必然要恢复期运行环境

2.将PCB插入合适的队列

3.分配/回收资源

进程控制块PCB

包含信息

1. 进程标识符
2. 处理机状态
3. 进程调度
4. 进程控制

组织方式

1. 链接

   相同状态组织成一个队列

2. 索引

   建立索引表，记录PCB地址

控制

系统对进程的控制通过操作系统内核中的原语实现

原语

由若干条指令构成的可完成特定功能的程序段，必须在管态下运行，它是一个“原子操作(atomic operation)”过程，执行过程不能被中断

进程控制的基本原语

- 创建进程：创建一个新进程，为其分配资源，初始化PCB，将其插入就绪队列
- 撤销进程：终止一个进程，释放其占有的资源，回收其PCB
- 阻塞进程：将一个进程由运行状态转换为阻塞状态
- 唤醒进程：将一个进程由阻塞状态转换为就绪状态
- 挂起进程：内存不足时使用
- 激活进程：解除挂起状态

层次结构

- 父进程
- 子进程

终止

引起进程终止的事件

1. 正常结束
2. 异常结束
3. 外界干预

阻塞 – 主动行为

唤醒 – 被动行为

调度

概念

周转时间= 完成时间 – 到达时间

带权周转时间 = 周转时间 / 运行(服务)时间

等待时间 = 周转时间 – 运行时间 –（IO操作）

平均周转时间=作业周转总时间/作业个数;

平均带权周转时间=带权周转总时间/作业个数;

调度层级

- 高级(作业调度)
  - 从辅存中选择作业送入内存，每个作业只调入一次，调出一次
- 低级(进程调度)
  - 从就绪队列选进程分配处理机
- 中级(内存调度)

- 提高内存利用率和系统吞吐量
  > 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就绪，将不能运行的调到外存挂起

调度方式

- 非剥夺方式
- 剥夺方式

调度准则

(略)用于比较的特征准则有

- CPU使用率
- 吞吐量：一个单位时间内完成进程的数量
- 周转时间：从进程提交到进程完成的时间称为周转时间
- 平均周转时间：多个作业周转时间平均值
- 带权周转时间：周转时间/运行时间
- 平均带权周转时间：多个作业带权周转时间的平均值
- 等待时间：为最就绪队列中等待所花的时间，调度算法不影响进程运行时间，只影响在队列中的等待时间
- 响应时间：从提交请求到响应请求的时间

调度算法

算法 + 原理

先进先出

（先来先服务）

- 简单，但效率低；对长作业比较有利，但对短作业不利（相对SPF和高响应比），若一个长作业先到达系统，就会使后面的许多短作业等待很长时间

短作业（最短CPU运行期）优先

- 从就绪队列中选择一个或几个估计运行时间最短的进程，将处理机分配给它，使之立即执行，直到完成或发生某事件而阻塞时，才释放处理机。

  ①算法对长作业不利，SJF调度算法中长作业的周转时间会增加。如果有一个长作业进人系统的后备队列，由于调度程序总是优先调度短作业，将导致长作业长期不被调度，可能会出现“饥饿”现象。

  ②没有考虑作业 的紧迫程度，因而不能保证紧迫性作业会得到及时处理。

最高响应比优先

- 先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

  响应比 = (等待时间 + 要求服务时间)/要求服务时间

  ①作业的等待时间相同时，要求服务时间越短，响应比越高，有利于短作业。

  ②要求服务时间相同时，作业的响应比由其等待时间决定，等待时间越长，其响应比越高，所以算法实现的又 是先来先服务。

  ③对于长作业，作业的响应比可以随等待时间的增加而提高，等待时间足够长时，其响应比便可升到很高，也可获得处理机。因此，克服了饥饿状态，兼顾了长作业。

- 非抢占

优先级

- 从就绪队列中选择优先级最高的进程，根据新的更高优先级进程能否抢占正在执行的进程，可分为非剥夺式优先级调度算法和剥夺式优先级调度算法

时间片轮转

- 进程调度程序总是按到达时间的先后次序选择就绪队列中的第一个进程执行，即先来先服务的原则，但仅运行一个时间片。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

  ①若时间片足够大，以至所有进程均能在一个时间片执行完成，则退化为FCFS算法

  ②若时间片很小，则处理机切换频繁，开销增大，进程使用时间减少

前后台

- 将分时用户程序放在前台，将批处理程序放在后台；对前台按照时间片轮转，前台空闲时候才把CPU分配给后台程序的进程。这样既能让分时用户进程得到及时响应， 又提高了系统资源的利用率

多级反馈队列轮转

- 时间片轮转调度算法和优先级调度算法的综合与发展，通过动态调整进程优先级和时间片大小，多级反馈队列调度算法可以兼顾多方面的系统目标

线程

引入　简化线程间的通信，以小的开销来提高进程内的并发程度(轻装运行)

引入线程后，进程只作为系统资源的分配单元，线程作为处理机的分配单元

与进的区别联系

- **线程**：操作系统进行**运行调度的最小单位**
- **进程**：系统进行**资源分配和调度的基本单位**

- 二者都用于实现程序的并发执行

- 进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位
- 进程有自己独立的地址空间和资源；线程共享所属进程的地址空间和资源，但有自己的栈和寄存器，线程之间的切换和通信开销较小
- 进程的并发性较差，一个进程崩溃可能导致整个系统崩溃；线程的并发性较好，一个线程崩溃只会影响该线程所属的进程
- 进程可以拥有多个线程，一个进程至少有一个线程；线程不能拥有其他线程，但可以创建和销毁其他线程

实现机制

分类

- 用户级线程
- 内核支持线程

两者比较

- 线程调度和切换速度
- 系统调用
- 执行时间

4进程同步

基本概念

进程间的相互作用

程序设计的核心

同步

需要相互合作，协同工作的进程间的关系　　直接制约关系

互斥

临界资源，　互斥访问

多个进程因争用临界资源而互斥执行　　间接制约关系

- 临界资源：一段时间内只允许一个进程访问的资源

信号量和P、V操作

- 信号量(semapore)：一个整形变量，通过初始化以及P、V操作来访问
  - 公有信号量：用于进程间的互斥，初值通常为1
  - 私有信号量：用于进程间的同步，初值通常为0或n
- P操作：荷兰语中的"proberen"，意为"测试"，用于申请临界资源
  - 若申请成功，信号量 $s = s - 1$ ，进程继续执行
  - 若申请失败，进程进入阻塞状态，进入S的等待队列等待信号量变为非0
- V操作：荷兰语“verhogen”—“增量/升高”之意，意味着释放/增加一个单位资源

P、V操作通常被设置为原语操作，不可分割，不可中断，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了　**互斥量（Mutex）**　，0 表示临界区已经加锁，1 表示临界区解锁。

```
typedef int semaphore;
semaphore mutex = 1;
void P1() {
    down(&mutex);
    // 临界区
    up(&mutex);
}

void P2() {
```

```
    down(&mutex);
    // 临界区
    up(&mutex);
}
```

尽量减少将访问临界区的锁拿在手上的时间，拿了什么锁就赶紧干什么事，不要再去干其他的了否则容易死锁(经验之谈)

在考试中如果来不及仔细分析，可以加上互斥信号量，保证各进程一定会互斥地访问缓冲区。但需要注意的是，`实现互斥的P操作一定要在实现同步的P操作之后`，否则可能引起·`“死锁”`

管程

管程(monitor)是关于共享资源的一组数据结构和在这组数据结构上的一组相关操作。管程将共享变量以及对共享变量能够进行的所有操作集中在一个模块中，以过程调用的形式提供给进程使用。

同步机制规则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

① 空闲让进，当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效的利用临界资源。

② 忙则等待，当已有进程进入临界区时，表明临界资源正在被访问，因而其他视图进入临界区的进程必须等待，以保证对临界资源的互斥访问。

③ 有限等待，对要求访问临界资源的进程 ，应保证在有限时限内能进入自己的临界区，以免陷入死等状态。

④ 让权等待，当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入忙等状态。

进程同步机制

信号量机制

- 整型信号量
- 记录型信号量
  - 由于整型信号量没有遵循让权等待原则，记录型允许负数，即阻塞链表
- AND型信号量
- 信号量集
  - 理解:AND型号量的wait和signal仅能对信号施以加1或减1操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。当一次需要N个单位时，便要进行N次wait操作，这显然是低效的，甚至会增加死锁的概率。此外，在有些情况下，为确保系统的安全性，当所申请的资源数量低于某一下限值时，还必须进行管制，不予以分配。因此，当进程申请某类临界资源时，在每次分配前，都必须测试资源数量，判断是否大于可分配的下限值，决定是否予以分配
  - 操作
    - `Swait(S1, t1, d1…Sn, tn, dn)`
    - `Ssignal(S1, d1…Sn, dn)`
  - 特殊情况

经典进程同步问题

注意区分是同步还是互斥问题

互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。
同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

- 生产者-消费者问题
- 读者-写者问题
- 哲学家进餐问题
- 嗜睡的理发师问题

生消

这是一个非常重要而典型的问题，进程同步60%以上的题目都是生产者消费者的改编

问题描述：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去进行消费。使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 `mutex` 来控制对缓冲区的互斥访问。（缓冲区用循环队列就可以模拟）

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：`empty` 记录空缓冲区的数量，`full` 记录满缓冲区的数量。其中，`empty` 信号量是在生产者进程中使用，当 `empty` 不为 `0` 时，生产者才可以放入物品；`full` 信号量是在消费者进程中使用，当 `full` 信号量不为 `0` 时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 `down(mutex)` 再执行 `down(empty)`。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 `down(empty)` 操作，发现 `empty = 0`，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 `up(empty)` 操作，`empty` 永远都为 `0`，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

顺序执行的时候显然没有任何问题，然而在并发执行的时候，就会出现差错，比如共享变量 `counter`，会出现冲突。解决的关键是将counter作为临界资源来处理。

```
##define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer() {
    while(TRUE) {
        int item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer() {
    while(TRUE) {
        down(&full);
        down(&mutex);
        int item = remove_item();
        consume_item(item);
        up(&mutex);
        up(&empty);
    }
}
```

实际例子：吃水果问题

桌子上有一个盘子，可以放一个水果。爸爸每次放一个苹果，妈妈每次放一个桔子；女儿每次吃一个苹果，儿子每次吃一个桔子。

```
semaphore S =  1, SA = 0, SO = 0; // 盘子的互斥信号量、苹果和桔子的互斥信号量

father(){
  while(1){
    have an apple;
    P(S);
    put an apple;
    V(SA);
  }
}

mother(){
```

```
  while(1){
    have an orange;
    P(S);
    put an orange;
    V(SO);
  }
}

daughter(){
  while(1){
    P(SA);
    get an apple;
    V(S);
    eat an apple;
  }
}

son(){
  while(1){
    P(SO);
    get an orange;
    V(S); // 吃完之后要V(S)才能接着让爸爸妈妈接着放
    eat an orange;
  }
}
```

典型问题：哲学家就餐问题

这是由Dijkstra提出的典型进程同步问题

5个哲学家坐在桌子边，桌子上有5个碗和5支筷子，哲学家开始思考，如果饥饿了，就拿起两边筷子进餐（两支筷子都拿起才能进餐），用餐后放下筷子，继续思考

多个临界资源的问题

只考虑筷子互斥：可能会产生死锁，比如所有人都同时拿起右边筷子，左边无限等待
再考虑吃饭行为互斥：同时只让一个人吃饭，肯定不会冲突或死锁，但是资源比较浪费

解决方法1：允许4位哲学家同时去拿左边的筷子

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; // 筷子信号量
semaphore eating = 4; // 允许四个哲学家可以同时拿筷子

void philosopher(int i){ // 第i个哲学家的程序
  thinking();
  P(eating);
```

```
  P(chopstick[i]); // 请求左边的筷子
  P(chopstick[(i+1)%5]); // 请求右边的筷子
  eating();
  V(chopstick[(i+1)%5]);
  V(chopstick[i]);
  V(eating);
}
```

解决方式2：奇数位置的哲学家先左后右，偶数位置的哲学家先右后左

典型问题：读者写者问题

读进程：Reader进程
写进程：Writer进程

允许多个进程同时读一个共享文件，因为读不会使数据混乱；但同时仅仅允许一个写者在写

写-写互斥，写-读互斥，读-读允许

纯互斥问题，没有同步关系没有先后之分

需要多加一个读者计数器，并且修改这个时要同步，所以要再加一个变量保证修改count时的互斥

```
semaphore rmutex = 1, wmutex = 1; // readcount的互斥信号量，数据的互斥信号量
int readcount= 0;

Reader(){ // 每次进入和退出因为涉及readcount变化，要保证同时只有一个，所以分别设置rmutex
  while(1){
    P(rmutex); // 抢readcount信号量，防止多个reader同时进入 导致readcount变化不同
    if(readcount==0){
      P(wmutex); // 第一个进来的读者，抢公共缓冲区
    }
    readcount += 1;
    V(rmutex); // 其他reader可以进来了

    perform read operation;

    P(rmutex); // 再抢一次，使每次只有一个退出
    readcount -= 1;
    if(readcount==0){
      V(wmutex); // 最后一个reader走了，释放公共缓冲区
    }
    V(rmutex);
  }
```

```
}

Writer(){ // 写者很简单，只需要考虑wmutex公共缓冲区
  while(1){
    P(wmutex);
    perform write operation;
    V(wmutex);
  }
}
```

上面的算法可能会导致写者可能会被插队，如果是RWR，中间的W被堵了，结果后面的R还能进去，W还要等后面的R读完

变形：让写者优先，如果有写者，那么后来的读者都要阻塞，实现完全按照来的顺序进行读写操作。
解决方法：再增加一个wfirst信号量，初始为1，在读者的Read()和之前的阶段和写者部分都加一个P(wfirst)作为互斥入口，结尾V(wfirst)释放即可

变形：让写者真正优先，可以插队
解决方法：增加一个writecount统计，也就是加一个写者队列，写者可以霸占这个队列一直堵着

例子：汽车过窄桥问题

来往各有两条路，但是中间有一个窄桥只能容纳一辆车通过，方向一样的车可以一起过

```
semaphore mutex1=1, mutex2=1, bridge=1;
int count1=count2=1;

Process North(i){
  while(1){
    P(mutex1);
    if(count1==0){
      P(bridge); // 第一辆车来了就抢bridge让对面的车进不来
    }
    count1++;
    V(mutex1);

    cross the bridge;

    P(mutex1);
    count1--;
    if(count1==0){
      V(bridge); // 最后一辆车走了就释放bridge让对面的车可以进来
    }
```

```
    V(mutex1);
  }
}

Process South(i){
  while(1){
    P(mutex2);
    if(count2==0){
      P(bridge);
    }
    count2++;
    V(mutex2);

    cross the bridge;

    P(mutex2);
    count2--;
    if(count2==0){
      V(bridge);
    }
    V(mutex2);
  }
}
```

哲学家进餐问题

五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

下面是一种错误的解法，如果所有哲学家同时拿起左手边的筷子，那么所有哲学家都在等待其它哲学家吃完并释放自己手中的筷子，导致死锁。

```
#define N 5

void philosopher(int i) {
    while(TRUE) {
        think();
        take(i);        // 拿起左边的筷子
        take((i+1)%N); // 拿起右边的筷子
        eat();
```

```
        put(i);
        put((i+1)%N);
    }
}
```

为了防止死锁的发生，可以设置两个条件：

- 必须同时拿起左右两根筷子；
- 只有在两个邻居都没有进餐的情况下才允许进餐。

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
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。
仅当一个哲学家左右两支筷子都可用时才允许他抓起筷子。

```
#define N 5
#define LEFT (i + N - 1) % N // 左邻居
#define RIGHT (i + 1) % N    // 右邻居
#define THINKING 0
#define HUNGRY   1
#define EATING   2
typedef int semaphore;
int state[N];                // 跟踪每个哲学家的状态
semaphore mutex = 1;         // 临界区的互斥，临界区是 state 数组，对其修改需要互斥
semaphore s[N];              // 每个哲学家一个信号量

void philosopher(int i) {
    while(TRUE) {
        think(i);
        take_two(i);
        eat(i);
        put_two(i);
    }
}

void take_two(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    check(i);
    up(&mutex);
    down(&s[i]); // 只有收到通知之后才可以开始吃，否则会一直等下去
```

```
}

void put_two(i) {
    down(&mutex);
    state[i] = THINKING;
    check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始吃了
    check(RIGHT);
    up(&mutex);
}

void eat(int i) {
    down(&mutex);
    state[i] = EATING;
    up(&mutex);
}

// 检查两个邻居是否都没有用餐，如果是的话，就 up(&s[i])，使得 down(&s[i]) 能够得到通知并继续执行
void check(i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

读者-写者问题

允许多个进程同时对数据进行读操作，但是不允许读和写以及写和写操作同时发生。

一个整型变量 count 记录在对数据进行读操作的进程数量，一个互斥量 count_mutex 用于对 count 加锁，一个互斥量 data_mutex 用于对读写的数据加锁。

```
typedef int semaphore;
semaphore count_mutex = 1;
semaphore data_mutex = 1;
int count = 0;

void reader() {
    while(TRUE) {
        down(&count_mutex);
        count++;
        if(count == 1) down(&data_mutex); // 第一个读者需要对数据进行加锁，防止写进程访问
        up(&count_mutex);
        read();
        down(&count_mutex);
```

```
        count--;
        if(count == 0) up(&data_mutex);//最后一个读者要对数据进行解锁，防止写进程无法访问
        up(&count_mutex);
    }
}

void writer() {
    while(TRUE) {
        down(&data_mutex);
        write();
        up(&data_mutex);
    }
}
```

进程通信

目前的高级通信机制可归结为三大类

- 共享存储

  要互斥的访问共享空间

  - 基于共享数据结构
  - 基于共享存储区
- 消息传递
- 管道通信
  - 一个管道只能实现半双工通信

死锁

如果一组进程中的每一个进程都在等待仅由该进程中的其他进程才能引发的事件，那么该组进程是死锁的

原因

1. 竞争资源
    - 可剥夺和非剥夺性资源
    - 竞争非剥夺性资源
    - 竞争临时性资源
2. 进程推进顺序非法

条件

原因之上更近一步

- 互斥
- 请求保持
- 不可抢占（不剥夺）
- 循环等待（环路等待）

他们不是相互独立的，前三者是必要，环路等待(循环等待)是死锁可能展现的结果

解决

- 预防
- 避免

- 检测
- 解除

预防死锁和避免死锁这两种方法都是通过施加某些限制条件，来预防发生死锁，两者主要差别在于：为预防死锁所施加的限制条件较严格，往往会影响进程的并发执行，而未避免死锁所施加的限制条件就相对宽松，这给进程的运行提供了较宽松的环境，有利于进程的并发执行。

预防

静态方法，在进程执行前采取的措施，通过设置某些限制条件，去破坏产生死锁的四个条件之一，防止发生死锁

- 破坏"请求和保存"条件

  所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源

  优点

  简单，易行，安全

  缺点

  - 资源被严重浪费，严重地恶化了资源的利用率
  - 使进程经常会发生饥饿现象

- 破坏"不可抢占"条件

  当一个已经保存了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请

- 破坏"循环等待（环路等待）"条件

  对系统所以资源类型进行线性排序，并赋予不同的序号。所有进程对资源的请求必须严格按资源序号递增的次序提出

避免

动态的方法，在进程执行过程中采取的措施，不需事先采取限制措施破坏产生死锁的必要条件，而是在进程申请资源时用某种方法去防止系统进入不安全状态，从而避免发生死锁

安全状态

- 系统安全状态

  > 某时刻，对于并发执行的n个进程，若系统能够按照某种顺序如<p1,p2...pn>来为每个进程分配所需资源，直至最大需求，从而使每个进程都可顺利完成，则认为该时刻系统处于安全状态，这样的序列为安全序列
  >
  > 不按照安全顺序进行分配资源，则可能发生由安全状态向不安全状态的转换

- 不安全状态：不存在一种顺序使得每个进程都能够顺利完成。不安全状态不一定发生死锁，但死锁一定是不安全状态

银行家算法

- 单个资源
  - 每个进程有一个贷款额度
  - 总的资源可能不能满足所有的贷款额度
  - 跟踪分配的和仍然需要的资源
  - 每次分配时检查安全性
- 多个资源
  - 两个矩阵：已分配和仍然需要

银行家算法步骤：

检查此次申请是否超过了之前声明的最大需求数
检查此时系统剩余的可用资源是否还能满足这次请求
试探着分配，更改各数据结构
用安全性算法检查此次分配是否会导致系统进入不安全状态

数据结构：

```
长度为m 的一维数组Available 表示还有多少可用资源
n * m 矩阵Max 表示各进程对资源的最大需求数
n * m 矩阵Allocation 表示已经给各进程分配了多少资源
Max - Allocation = Need 矩阵表示各进程最多还需要多少资源
用长度为m 的一位数组Request 表示进程此次申请的各种资源数
```

安全性算法步骤：

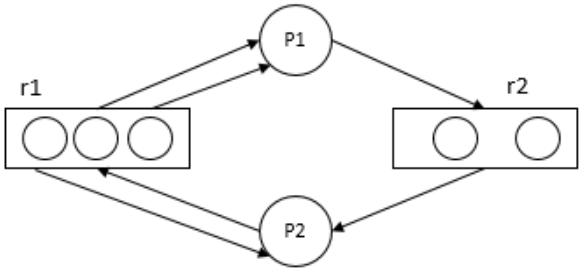
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，
并把该进程持有的资源全部回收。
不断重复上述过程，看最终是否能让所有进程都加入安全序列。

检测

- 资源分配图
  - 简化步骤
    - 选择一个没有阻塞的进程p
    - 将p移走，包括它的所有请求边和分配边
    - 重复步骤1，2，直至不能继续下去
- 死锁定理
  - 利用资源分配图

    若一系列简化以后不能使所有的进程节点都成为孤立节点
- 检测算法

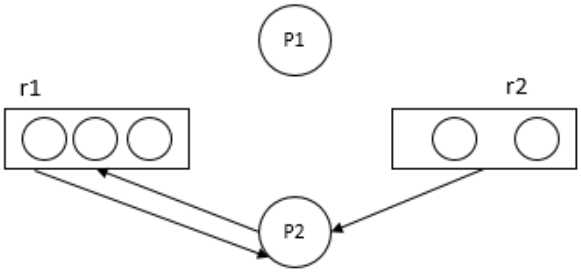
资源分配图



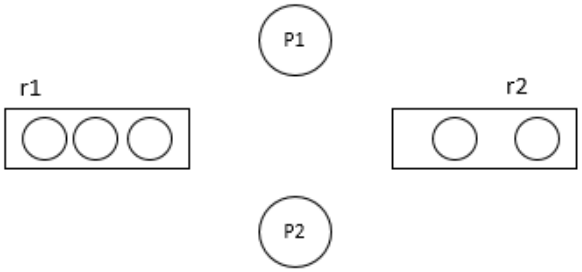
说明：p1进程获得了两个r1资源，请求一个r2资源，p2进程获得一个r1资源和一个r2资源，并且又申请了一个r1资源。

利用资源分配图可以检测系统是否处于死锁状态。

① 在资源分配图中，找到一个既不阻塞又不独立的进程结点pi，在顺利情况下，可获得所需资源而继续运行，直至运行完毕，再释放其所占用的全部资源。



② p1释放资源后，p2可获得资源继续运行，执行p2完成后又释放其所占用的资源。



③ 在进行一系列化简后，若能消去图中所有的边，使得所有的进程结点都成为孤立结点，则称该图是可以完全简化的，若不能通过任何过程使改图完全简化，则改图是不可完全简化的。

所有的简化顺序，都将得到相同的不可简化图，S为死锁状态的充分条件是：**当且仅当S状态的资源分配图是不可完全简化的，该充分条件称为死锁定理。**

1. 圆(椭圆)表示一个进程；

2. 方块表示一个资源类，其中的圆点表示该类型资源中的单个资源；
3. 从资源指向进程的箭头表示资源被分配给了这个进程；
4. 从进程指向资源的箭头表示进程申请一个这类资源；

临时性资源，即可消耗的资源。如信号，邮件等。特点是没有固定数量，不需要释放

含临时性资源的资源分配图规则：

1. 圆表示一个进程；
2. 方块表示一个资源类，其中的圆点表示该类型资源中的单个资源；
3. 由进程指向资源的箭头表示该进程申请这种资源，一个箭头只表示申请一个资源；
4. 由资源类指向进程的箭头表示该进程产生这种资源，一个箭头可表示产生一到多个资源，每个资源类至少有一个生产者进程。

检测

1. 检查有无环路，如果没有环路，则系统一定不会发生死锁；
2. 如果有环路，则检查环路上的资源类是否都只有一个资源，如果是，则系统一定会发生死锁；
3. 如果环路上的资源类都有多个资源，查找即非阻塞又非独立的进程，消去与之有关的所有有向边
4. 如果最终能够消去所有的有向边，则系统不会发生死锁，否则系统会发生死锁

死锁定理：如果一个系统状态为死锁状态，当且仅当资源分配图是不可完全化简。也即，如果资源图中所有的进程都成为孤立结点，则系统不会死锁；否则系统状态为死锁状态

解除

- 抢占资源
- 终止（（或撤销））进程

5存储

定义

碎片

内存碎片分为 内部碎片和外部碎片

- 【内部碎片】
  已经被分配出去却不能被其利用的内存空间(分太多了)

  常见于固定分配方式
- 【外部碎片】
  由于太小了因此无法分配给申请内存空间的新进程的内存空闲区域(太碎了分不出去)

  一般出现在动态分配方式中
  - **分段**管理会产生外部碎片，**分页**管理只有少量的内部碎片

消除

外部碎片通过**紧凑技术**消除

操作系统不时地对进程进行移动和整理。但是这需要动态重定位寄存器地支持，且相对费时。紧凑地过程实际上类似于Windows系统中地磁盘整理程序，只不过后者是对外存空间地紧凑

内存交换

可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出连续的 256MB 空间，于是新的 200MB 程序就可以装载进来

回收内存时要尽可能地将相邻的空闲空间合并

目的

方便用户　　将逻辑和物理地址分开

常规存储管理方式的特征

- 一次性
- 驻留性

覆盖技术

略

交换技术

略

交换技术就是把暂时不用的某个程序及数据的一部分或全部从内存中移到外存中去，腾出必要的内存空间；或把指定的程序或数据从外存中读到相应的内存中，并将控制权转给它

- 换出　　把处于等待（阻塞）状态的进程从内存移到外存，将内存空间腾出来
- 换入　　将准备好竞争CPU运行的进程从外存转移到内存

重定位/地址映射

将地址空间中使用的逻辑地址转换为内存空间中的物理地址的地址转换

根据地址转换的时间和手段，把重定位分为静态重定位和动态重定位两种

程序操作

步骤

1. 编译

   源程序 ->目标模块 (Object modules) --------Compiler

   由编译程序对用户源程序进行编译，形成若干个目标模块

2. 链接

   一组目标模块 ->装入模块 (Load Module) ----------Linker

   由链接程序将编译后形成的一组目标模板以及它们所需要的库函数链接在一起，形成一个完整的装入模块

3. 装入

   装入模块 ->内存 --------Loader

   由装入程序将装入模块装入内存

链接

- 静态链接方式
- 装入时动态链接
- 运行时动态链接

装入

- 绝对装入方式
- 可重定位装入方式

  一个程序通常需要占用连续的内存空间，程序装入内存后不能移动。不易实现共享

- 动态运行时的装入方式

  > 装入模块装入内存后，并不立即把装入模块中的逻辑地址转换为物理地址，而是把这种地址转换推迟到程序真正要执行时才进行
  >
  > 可以将一个程序分散存放于不连续的内存空间，可以移动程序，有利用实现共享。

地址绑定

（略）

将指令和数据绑定到内存地址有三种情况：

- 编译时：生成绝对代码，编译时就知道了进程在内存中的驻留地址，所生成的编译代码就可以从该位置往后扩展（开始位置发生变化就要重新编译）
- 加载时：生成可重定位代码
- 执行时：绑定延迟到执行时才进行

- 逻辑地址（相对地址，虚拟地址）：用户程序经过编译、汇编后形成目标代码，目标代码通常采用相对地址的形式，其首地址为`0`，其余地址都相对于首地址而编址
- 物理地址（绝对地址，实地址）：内存中储存单元的地址，可直接寻址

编译和加载时的地址绑定方法生成相同逻辑地址和物理地址，而执行时的地址绑定方案生成不同的逻辑和物理地址，这种情况通常称逻辑地址为虚拟地址

运行时从虚拟地址到物理地址的映射由**内存管理单元**来完成

管理方式

碎片

- 内碎片：占用分区之内未被利用的空间
- 外碎片：占用分区之间难以利用的空闲分区（通常是小空闲分区）

连续分配

- 单一连续分配

  （最简单）
- 分区分配
  - 固定分区
  - 可变分区

固定分区

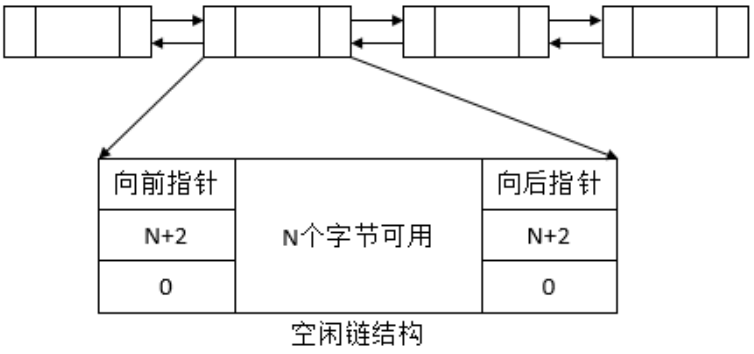
可变分区

指在作业装入内存时，从可用的内存中划出一块连续的区域分配给它，且分区大小正好等于该作业的大小。可变式分区中分区的大小和分区的个数都是可变的，而且是根据作业的大小和多少动态地划分

分区分配DS

- 已分分区表 + 空闲分区表
- 空闲分区链

空闲分区表（在系统中设置一张空闲分区表，用于记录每个空闲分区的情况，每个空闲分区占一个表目，表目中包括分区序号、分区始址、分区大小等，在前面已有介绍）、空闲分区链（为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的向前指针；在分区尾部设置一向后指针，这样，可以将空闲分区链接成一个双向链），为了检索方便，在分区尾部重复设置状态为和分区大小表目，当分区被分配出去以后，把状态为从0改成1，此时前后指针都失去意义（已经不再空闲链表中）。



空闲链结构

分区分配AL

算法	思想	空闲分区	优点	缺点
首次适应算法	从前往后找到第一个满足要求的空闲分区	地址递增排序	简单，快速	产生大量无法利用的小碎片
循环首次（邻近）适应算法	从上次分配的空闲分区开始找，直到找到第一个满足要求的空闲分区	地址递增排序（循环）	简单，快速	产生大量无法利用的小碎片
最佳适应算法	从所有满足要求的空闲分区中找到最小的一个	容量递增排序	大分区得以保留	产生小碎片。开销大，回收分区要重新对空闲分区排队
最坏适应算法	从所有满足要求的空闲分区中找到最大的一个	容量递减排序	减少小碎片	大分区不能保留。开销大，回收分区要重新对空闲分区排队

- 首次适应算法（first fit,FF）
  - 顺序找，找到一个满足的就分配，但是可能存在浪费
  - 这种方法目的在于减少查找时间
  - 空闲分区表（空闲区链）中的空闲分区要按地址由低到高进行排序
- 循环首次适应算法（next fit，NF）
  - 相对上面那种，不是顺序，类似哈希算法中左右交叉排序
  - 空闲分区分布得更均匀，查找开销小
  - 从上次找到的空闲区的下一个空闲区开始查找，直到找到第一个能满足要求的的空闲区为止，并从中划出一块与请求大小相等的内存空间分配给作业。
- 最佳适应算法（best fit，BF）

    - 找到最合适的，但是大区域的访问次数减少
    - 这种方法能使外碎片尽量小。
    - 空闲分区表（空闲区链）中的空闲分区要按大小从小到大进行排序，自表头开始查找到第一个满足要求的自由分区分配。
- 最差适应算法（worst fit，WF）
    - 相对于最好而言，找最大的区域下手，导致最大的区域可能很少，也造成许多碎片
    - 空闲分区按大小由大到小排序

总结

首次适应不仅最简单，通常也是最好最快，不过首次适应算法会使得内存低地址部分出现很多小的空闲分区，而每次查找都要经过这些分区，因此也增加了查找的开销。邻近算法试图解决这个问题，但实际上，它常常会导致在内存的末尾分配空间分裂成小的碎片，它通常比首次适应算法结果要差。

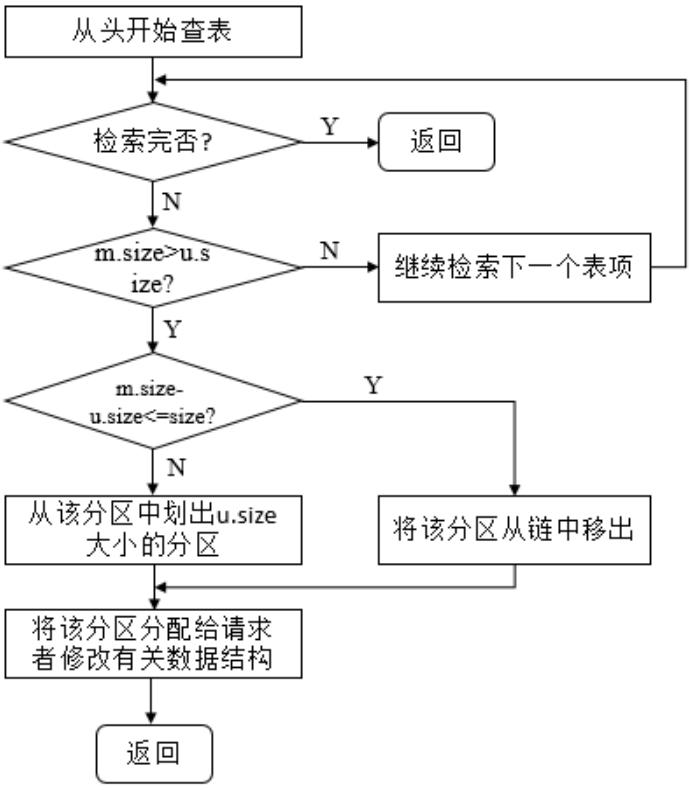
最佳导致大量碎片，最坏导致没有大的空间。

首次适应比最佳适应要好，他们都比最坏好。

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。**算法开销小**，回收分区后一般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多太小的、难以利用的碎片；**算法开销大**，回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完，不利于大进程；**算法开销大**（原因同上）
邻近适应	由首次适应演变而来，每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列（可排列成循环链表）	不用每次都从低地址的小分区开始检索。**算法开销小**（原因同首次适应算法）	会使高地址的大分区也被用完

分配操作

系统利用某种分配算法，从空闲分区链（表）中找到所需大小的分区，其流程图如下

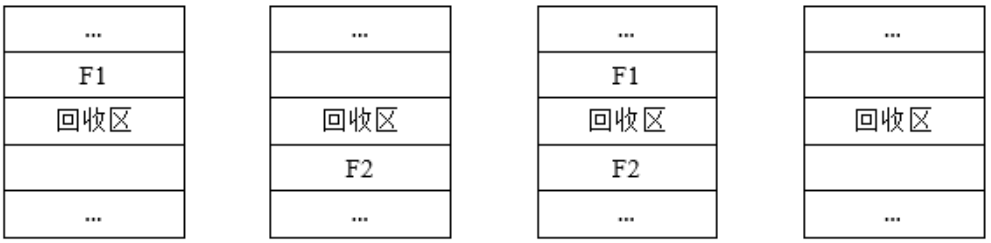


说明：size表示事先规定的不再切割的剩余分区的大小。空闲分区表示为m.size，请求分区的大小为u.size。

回收操作

当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链（表）中找到相应的插入点，此时会出现如下四种情况之一

1. **回收分区与插入点的前一个空闲区F1相邻接**，此时将回收区与插入点的前一分区合并，不必为回收区分配新表项，只需要修改前一分区F1的大小。
2. **回收分区与插入点的后以空闲分区F2相邻接**，此时将两分区合并，形成新的空闲分区，用回收区的首址作为新空闲区的首址，大小为两者之和。
3. **回收区同时与插入点的前、后两个分区邻接**，此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。
4. **回收区既不与F1邻接，也不与F2邻接**，这时为回收区单独建立一个新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。



优缺点

紧凑

动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，**增加了紧凑功能**，通常，在找不到足够大的空闲分区来满足用户需求时进行紧凑。

离散分配

（非连续分配）

连续分配方式会形成很多碎片，为之进行紧凑操作的开销非常大，如果允许一个进程直接分散地装入到许多不相邻接的分区中，则无须进行紧凑操作，基于这一思想产生了离散分配方式，如果离散分配的基本单位是页，则称为分页存储管理方式，若为段，则为分段存储管理方式。

将程序分为若干块，分别放在不同的空闲区中，从而使得存储空间的利用率得到提升。

各种方式的访问内存次数

- 基本两个　　2次

  > 第一次从内存中找到页表，然后找到页面的物理块号，加上页内偏移得到实际物理地址；第二次根据第一次得到的物理地址访问内存取出数据

- 基本两个加快表命中　　1次
- 多级基本两个　　多一级加一次

- 多级基本两个加快表　　多一级加一次，多一个快表命中减一次
- 段页式　　3次
- 段页式加快表　　3次，多一次TLB命中减一次

基本分页

方便系统内存利用率提升

概念

- 页面
  - 将一个进程的逻辑地址空间分成若干个大小相等的片
- 页框（frame）
  - 内存空间分成与页面相同大小的存储块
- 地址结构
  - 页号P + 位移量W(0-31)
- 页表
  - 在分页系统中，允许将进程的各个页离散地存储在内存在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每一个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表
  - 页表的作用是实现从页面号到物理块号的地址映射
- 地址变换机构

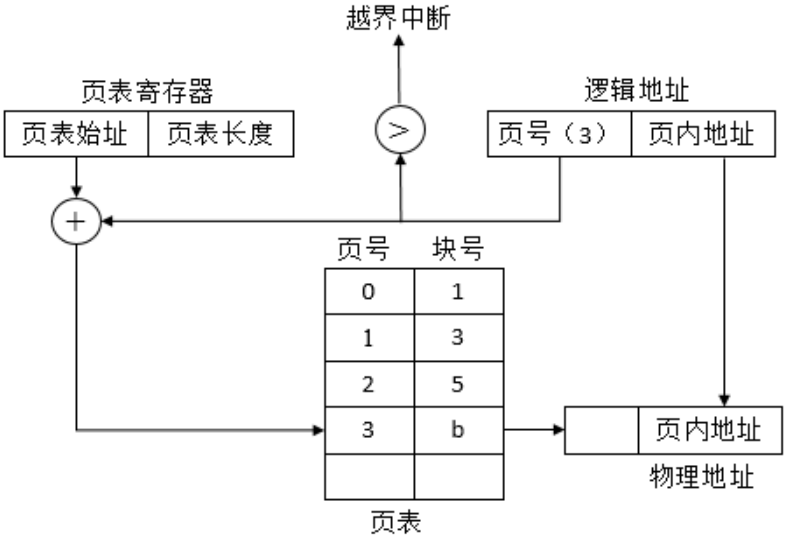
  页表寄存器PTR(Page-Table Register)　　存放页表在内存的始址 + 页表长度 长度 + 始址

  工作原理　　页号(判断地址越界)->查PTR->获得页表信息查页表(在内存中)->转换为块号->拼接块号和块内地址(拼)->找到内存块

采用分页技术不会产生外部碎片，但是会有内部碎片，产生的原因是进程所需内存不是页的整数倍，最后一页可能装不满最后一个帧（物理块），导致产生页内碎片

分页的一个重要特点是用户视角的内存和实际物理内存的分离，用户看到的是一整块内存用于一个进程，但实际的物理内存则是分散的，逻辑地址到物理地址的映射用户不知道。

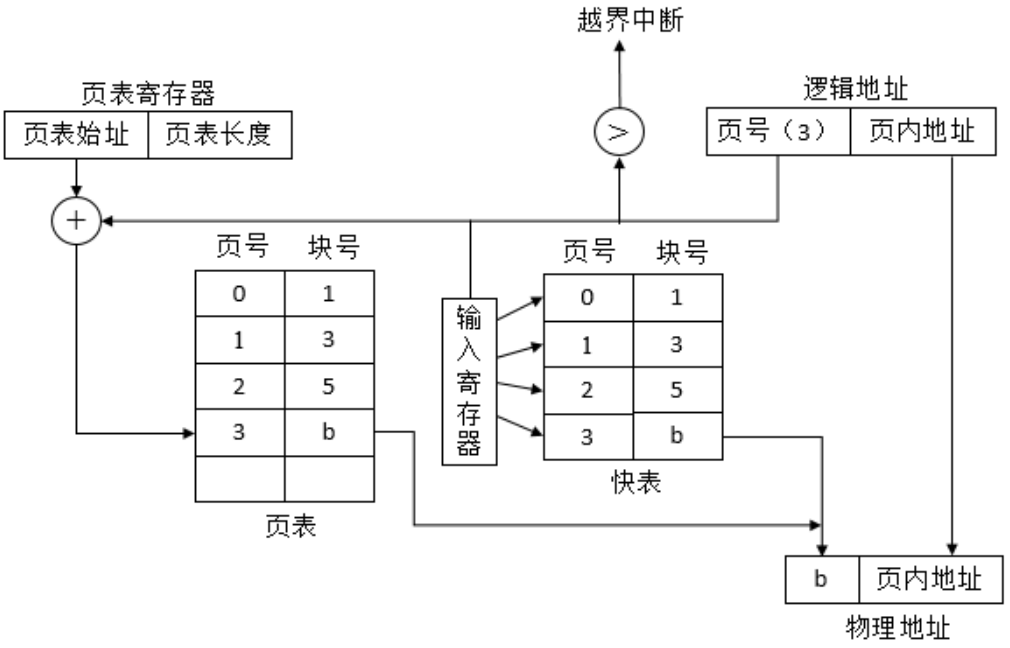
用户进程不能访问该进程非占用的内存，即无法访问页表规定之外的内存



+快表

有快表情况下地址变换过程为

1. CPU给出有效地址
2. 地址变换机构自动地将页号送入高速缓存，确定所需要的页是否在快表中。
3. 若是，则直接读出该页所对应的物理块号，送入物理地址寄存器；
4. 若快表中未找到对应的页表项，则需再访问内存中的页表
5. 找到后，把从页表中读出的页表项存入快表中的一个寄存器单元中，以取代一个旧的页表项。



多级页表

将页表进行分页，每个页面的大小与内存物理块的大小相同，并为它们进行编号，可以离散地将各个页面分别存放在不同的物理块中

两级页表的分页系统访问一次需要三次访存

有效访问时间

定义：从进程发出指定逻辑地址的访问请求，经过地址变换，到在内存中找到对应的实际物理地址所需要花费的总时间

基本页式管理下EAT的计算

1、无快表

假设一次访问内存的时间为t，则无快表情况下

EAT=t+t=2t

2、有快表

假设一次访问快表的时间为λ，快表命中率a，一次访问内存的时间为t

EAT= a(λ+t)+(1−a)(λ+t+t)

缺页率计算题

缺页中断率的定义：缺页中断次数/进程访问次数

大小计算题

例子

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?

$4GB = 2^{32}B$, $4KB = 2^{12}B$

因此4GB的内存总共会被分为 $2^{32} / 2^{12} = 2^{20}$ 个内存块，因此内存块号的范围应该是$0 \sim 2^{20} - 1$ 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（每个字节8个二进制位，3个字节共24个二进制位）

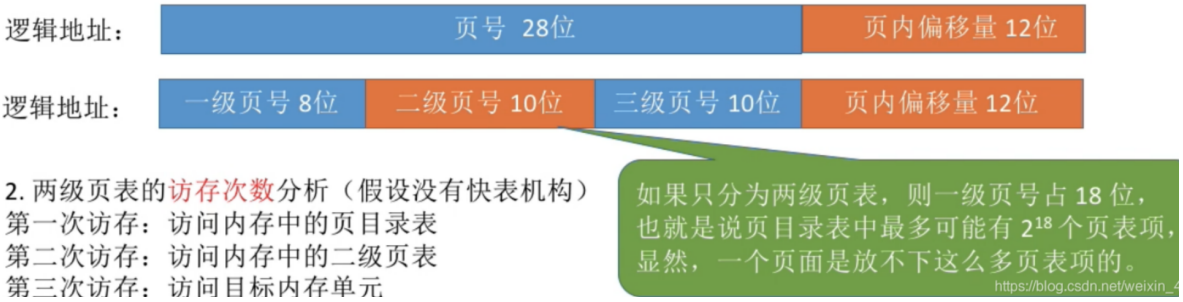
各页表项会按顺序连续地存放在内存中

如果该页表在内存中存放的起始地址为X，则M号页对应的页表项一定是存放在内存地址为 $X + 3 * M$ 因此，页表中的**“页号”可以是“隐含”的。**只需要知道页表存放的起始地址和页表项长度，即可找到各个页号对应的页表项存放的位置

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

例：某系统按字节编址，采用 40 位逻辑地址，页面大小为 4KB，页表项大小为 4B，假设采用纯页式存储，则要采用（）级页表，页内偏移量为（）位？

页面大小 = 4KB =2^{12}B，按字节编址，因此页内偏移量为12位
页号 = 40 - 12 = 28 位
页面大小 = 2^{12}B，页表项大小 = 4B，则每个页面可存放 2^{12} / 4 = 2^{10} 个页表项
因此各级页表最多包含 2^{10} 个页表项，需要 10 位二进制位才能映射到 2^{10} 个页表项，因此每一级的页表对应页号应为10位。总共28位的页号至少要分为三级



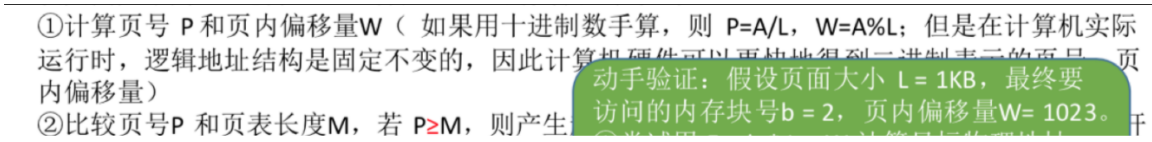
2. 两级页表的访存次数分析（假设没有快表机构）
第一次访存：访问内存中的页目录表
第二次访存：访问内存中的二级页表
第三次访存：访问目标内存单元

地址转换题

计算题

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。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。

注意:页面大小是2的整数幂
设页面大小为L，逻辑地址A到物理地址E的变换过程如下:



例:若页面大小L为1K字节，页号2对应的内存块号b=8，将逻辑地址A=2500转换为物理地址E。
等价描述：某系统按字节寻址，逻辑地址结构中，页内偏移量占10位(说明一个页面的大小为2^10B = 1KB)，页号2对应的内存块号 b=8，将逻辑地址A=2500转换为物理地址E。

①计算页号、页内偏移量
页号P=A/L = 2500/1024 = 2; 页内偏移量W= A%L = 2500%1024 = 452

②根据题中条件可知，页号2没有越界，其存放的内存块号b=8

③物理地址E=b*L+W=8 * 1024+ 425 = 8644

在分页存储管理(页式管理)的系统中，只要确定了每个页面的大小，逻辑地址结构就确定了。因此，页式管理中地址是一维的。

即，只要给出一个逻辑地址，系统就可以自动地算出页号、页内偏移量两个部分，并不需要显式地告诉系统这个逻辑地址中，页内偏移量占多少位。

某虚拟存储器的用户空间共有32个页面，每页1K，主存16K。假定某时刻系统为用户的第0、1、2、3页分配的物理块号为5、10、4、7，而该用户作业的长度为6页，试将十六进制的虚拟地址0A5C、103C、1A5C转换为物理地址，转换过程中是否会发生中断？发生何种中断？

（1）该系统的逻辑地址有15位，其中高5位为页号，低10位为页内地址；物理地址14位，其中高4位为块号，低10位为块内地址。逻辑地址(0A5C)16表示为二进制数地址为000 1010 0101 1100，即其页号为00010，即2，故页号合法；从页表中找到对应的内存块号为4，即0100；与页内地址10 0101 1100拼接形成物理地址010010 0101 1100，即（125C)16
（2）逻辑地址（103C）16的页号为4，页号合法，但该页未装入内存，故产生缺页中断。
（3）逻辑地址（1A5C）16的页号为6，为非法页号，故产生越界中断。

某虚拟存储器的用户编程空间共32个页面，每页为1KB，内存为16KB。假定某时刻一用户页表中已调入内存的页面的页号和物理块号的对照表如下表，求逻辑地址357所对应的物理地址为？2456所对应的物理地址为？

页号	物理块号
0	5
1	10
2	4

```
页号 = 有效地址÷每页字节数（页号=商，页内偏移量=余数）
块号：对应页号找表中的物理块号
块长：每页字节数
地址转换公式：物理地址 = 块号×块长+块内地址(页内偏移量)
```

357：357÷1024=0…357 页号=0 页内偏移量=357

357所对应的物理地址：5*1024+357=5477

2456：2456÷1024=2…408 页号=2 业内偏移量=408

2456所对应的物理地址：4*1024+408=4504

基本分段

方便用户

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段是一组完整的逻辑信息，每个段都有自己的名字，都是从零开始编址的一段连续的地址空间，各段长度是不等的。

内存空间被动态的划分为若干个长度不相同的区域，称为物理段，每个物理段由起始地址和长度确定

概念

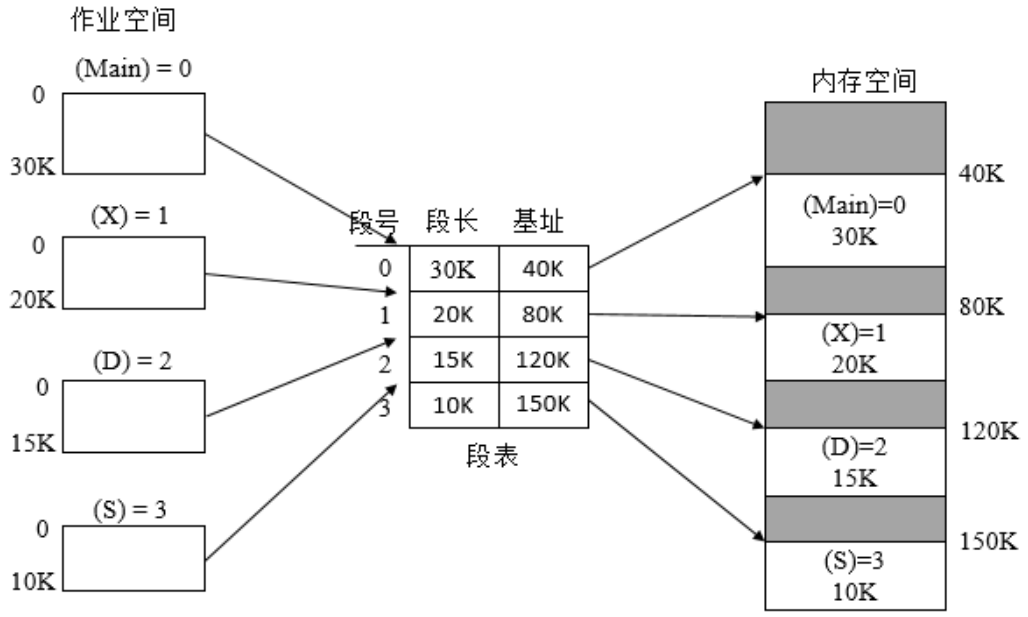
格式　段号+段内地址

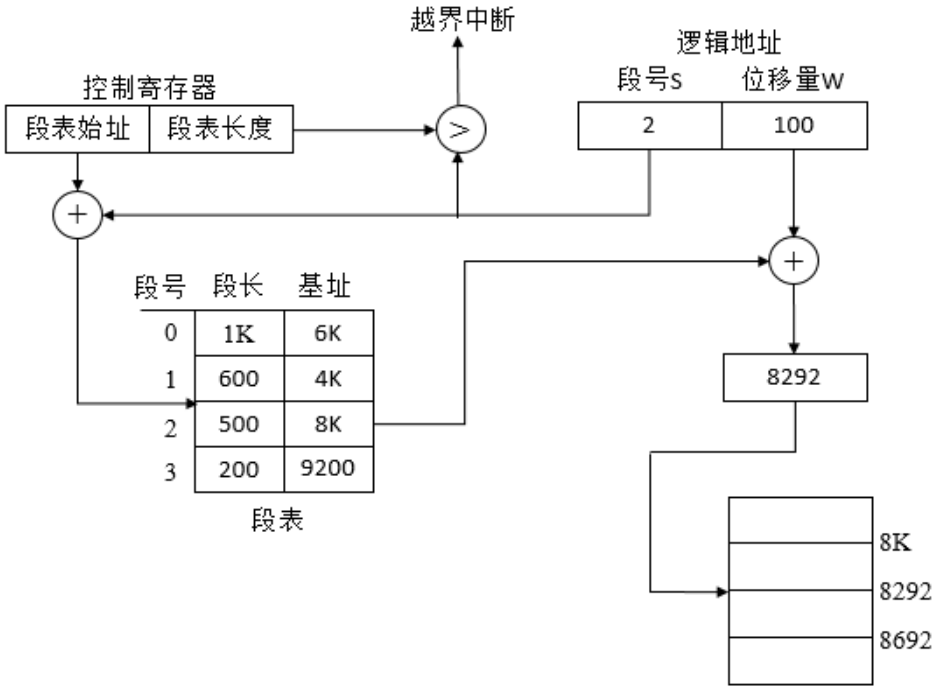
- 段表

  段表实现了从逻辑段到物理内存区的映射

- 地址变换机构

  段表寄存器(+)





分段分页区别

在段页式系统中，以段为单位管理用户的虚拟空间，以页为单位管理内存空间

> 分页是系统管理的需要，分段是用户应用的需要。一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处

- 页是信息的物理单位　　段是信息的逻辑单位
- 页的大小固定由系统决定　　段的大小不固定由用户程序决定
- **从用户角度看**

> **分页的地址空间是一维的** + **透明的**　**分段的地址空间是**二维的（要标识一个地址出了给出段内地址外还需给出段号） + **可见的**

补充

> 在一个进程中，段表只有一个，而页表可能有多个
> 分段比分页更容易实现信息的共享和保护
> 单级页表和分段访问一个逻辑地址均需要两次访存

段页式

用户程序按段划分，物理内存按页划分，即以页为单位进行分配

基本原理

段号S	页号P	页内地址W

- 段表记录了每一页的页表的起始地址和页表长度
- 页表记录了每一段所对应的逻辑页号与内存块号的对应关系，每一段有一个页表

通过三次访问取指令或数据

1. 访问内存中的段表，获得页表地址
2. 访问内存中的页表，获得该页所在的物理块号
3. 真正根据所得的物理地址取出指令或者数据

虚存

- 请求分页
- 请求分段

6虚拟存储器

概念

定义

具有请求调入功能和置换功能，能从**逻辑**上对内存容量加以扩充的一种存储器系统

局部性原理

（核心）

程序在执行时将呈现出局部性特征，即在一较短的时间内，程序的执行仅局限于某个部分，相应地，它所访问的存储空间也局限于某个区域

- 时间局限性
- 空间局限性

基本原理

- 在程序装入时不必将其全部读入内存，而只需要将当前需要执行的部分页或段读入内存就可以开始执行
- 在程序执行过程中，如果需要用到的指令或者数据不在内存中，则通知操作系统把需要的页或段调入内存，继续执行
- 操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段–具有请求调入和置换功能，只需程序的一部分在内存就可执行

优点

- 大程序：可在较小的可用内存中执行较大的用户程序；
- 大的用户空间：提供给用户可用的虚拟内存空间通常大于物理内存(`real memory`)
- 并发：可在内存中容纳更多程序并发执行；
- 易于开发：不必影响编程时的程序结构

- 以CPU时间和外存空间换取昂贵内存空间，这是操作系统中的资源转换技术

实现

- 虚拟页式（请求分页）
- 虚拟段式（请求分段）
- 虚拟段页式（请求分段页）

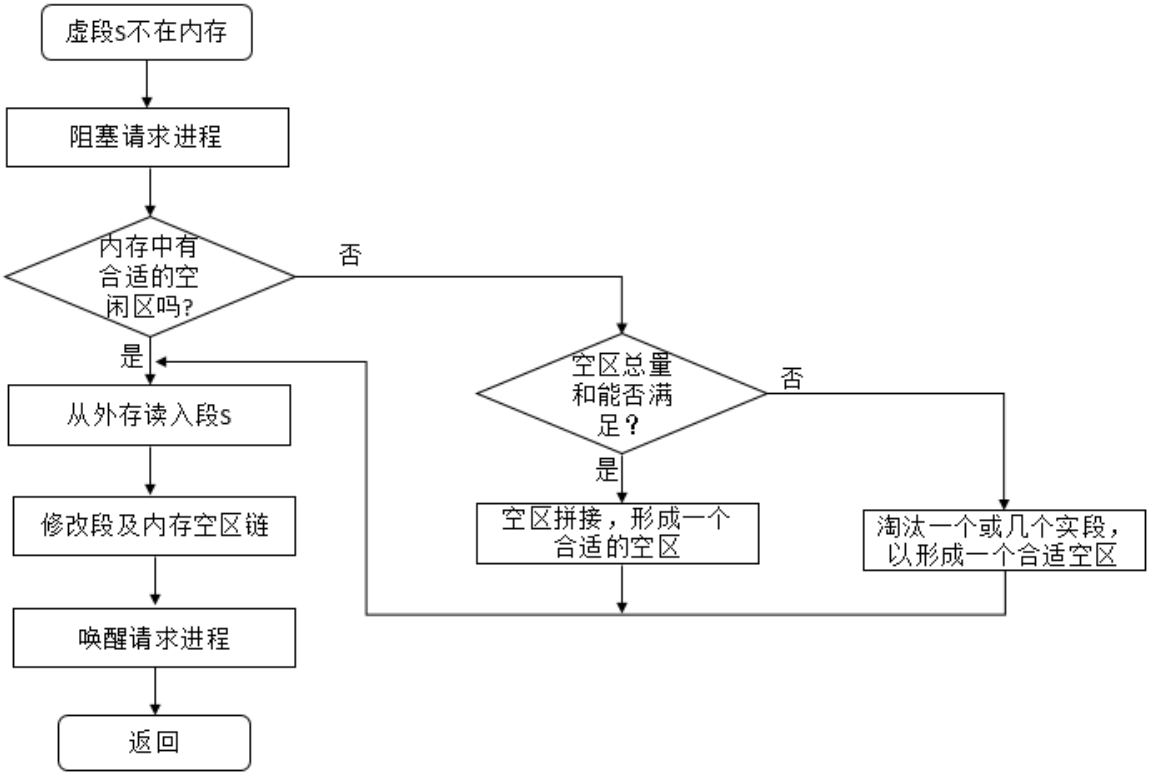
特征

- 离散性
  - 指在内存分配时采用离散的分配方式，它是虚拟存储器的实现的基础
- 多次性
  - 指一个作业被分成多次调入内存运行，即在作业运行时没有必要将其全部装入，只须将当前要运行的那部分程序和数据装入内存即可。多次性是虚拟存储器最重要的特征
- 对换性
  - 指允许在作业的运行过程中在内存和外存的对换区之间换进、换出。
- 虚拟性
  - 指能够从逻辑上扩充内存容量，使用户所看到的内存容量远大于实际内存容量。

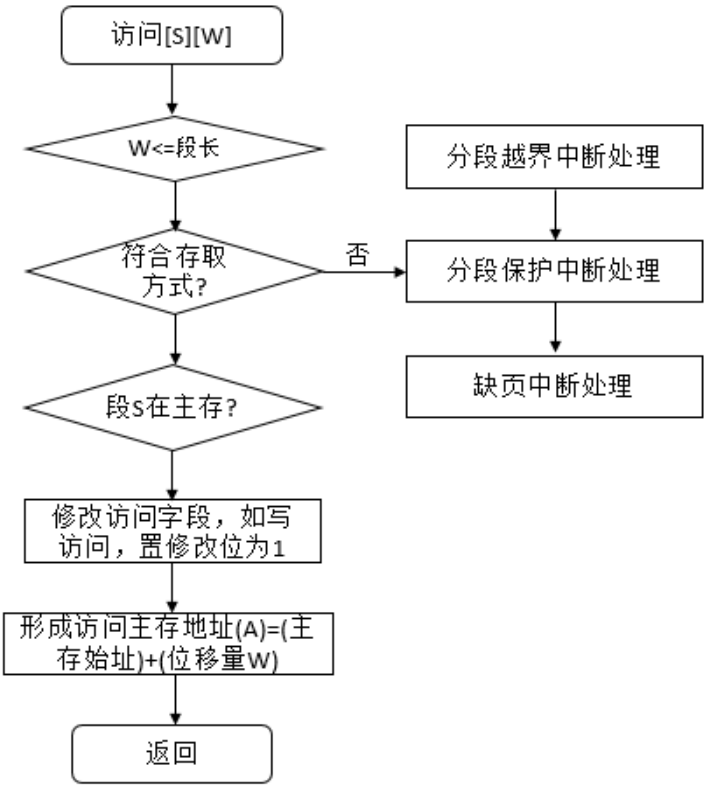
请求分页虚拟存储

硬件机构

缺页中断机构


虚段s不在内存
阻塞请求进程
内存中有
合适的空
闲区吗?
否
是
空区总量
和能否满
足?
否
是
从外存读入段s
修改段及内存空区链
唤醒请求进程
返回
空区拼接，形成一个
合适的空区
淘汰一个或几个实段，
以形成一个合适空区

地址变换机构



页面置换算法

- 最佳置换算法(不能实现)
- FIFO
- 最近最久未使用（LRU)
    - 寄存器支持
    - 特殊的栈结构
- 简单clock
    - 对访问位A的判断
- 改进clock
    - 增加对修改位M思维判断

LRU

每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：

- R=0, M=0
- R=0, M=1
- R=1, M=0
- R=1, M=1

- 未访问过且未修改过
- 未访问过且修改过
- 访问过且未修改过
- 访问过且修改过

当发生缺页中断时，NRU 算法随机地从类编号最小的非空类中挑选一个页面将它换出。

NRU 优先换出已经被修改的脏页面（R=0，M=1），而不是被频繁使用的干净页面（R=1，M=0）。

Clock

- 改进Second Chance在替换时移动页的开销
- 算法：把所有页框组织成环形链表
  - 用一个表针指向最老的页
  - 发生缺页时，按表针走动方向来检查页
- 第二次机会：
  - 如果R位为1，将其置为0，且表针向前移一格
  - 如果R位为0，替换它

双表针的Clock算法

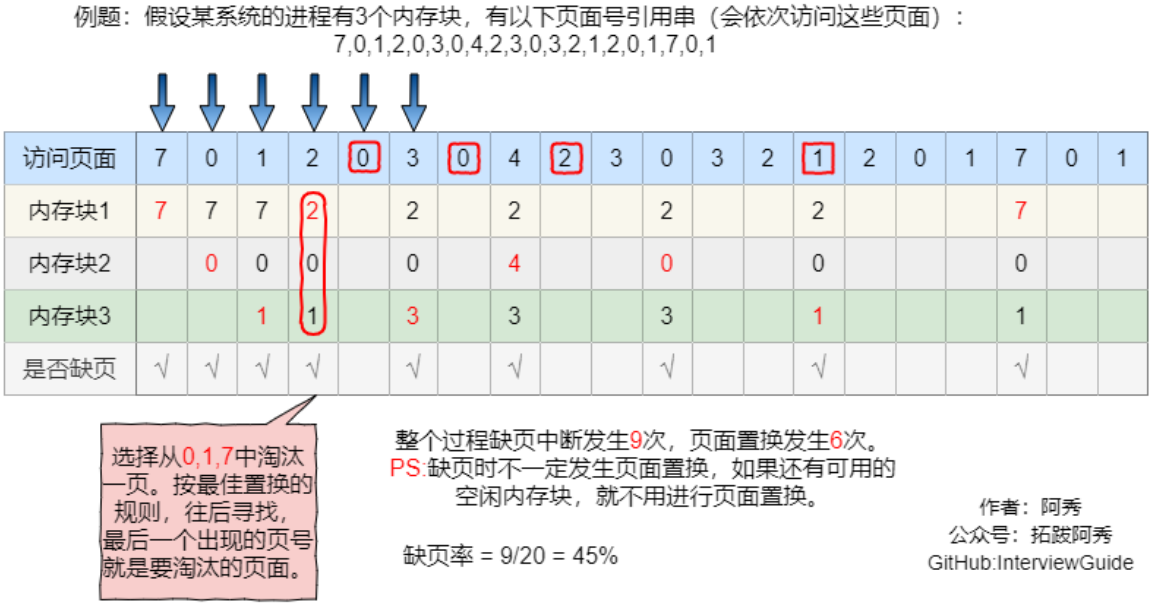
- 对Clock的替换加以控制
- 方法：增加一个表针

    - 前表针扫描页，把R位清0
    - 后表针扫描页，把R位为0的页加入替换页链表
    - 扫描方向相同，扫描速度相同
- 替换控制
    - 扫描速度：
        - 控制替换速度
        - 空闲内存多，扫描速度慢
        - 空闲内存少，扫描速度快
    - 表针间距：
        - 页框再次被访问的时间窗口
        - 控制页在内存里的最长停留时间

页面置换算法总结

1、最佳置换法(OPT)

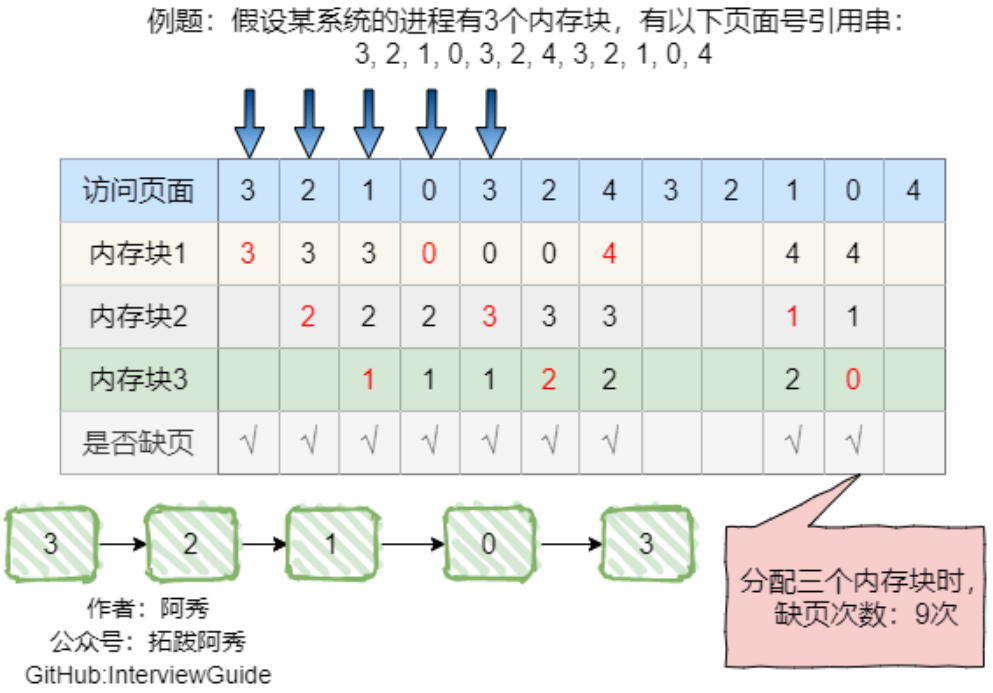
最佳置换算法(OPT，Optimal)：每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。



最佳置换算法可以保证最低的缺页率，但实际上，只有在进程执行的过程中才能知道接下来会访问到的是哪个页面。操作系统无法提前预判页面访问序列。因此，最佳置换算法是无法实现的

2、先进先出置换算法(FIFO)

先进先出置换算法(FIFO) :每次选择淘汰的页面是最早进入内存的页面
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队列的最大长度取决于系统为进程分配了多少个内存块。



Belady异常—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。

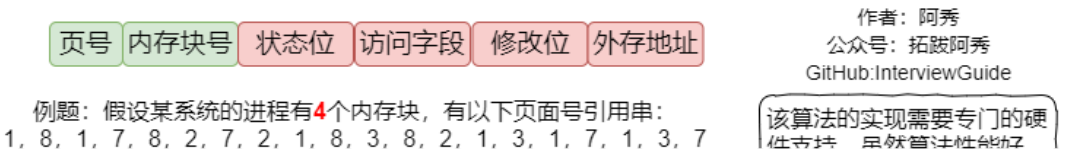
只有FIFO算法会产生Belady异常，而LRU和OPT算法永远不会出现Belady异常。另外，FIFO算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能差

FIFO的性能较差，因为较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出，并且有Belady现象。所谓Belady现象是指：采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

3、最近最久未使用置换算法(LRU)

最近最久未使用置换算法(LRU, least recently used) :每次淘汰的页面是最近最久未使用的页面
实现方法:赋予每个页面对应的页表项中，用访问字段记录该页面自上次被访问以来所经历的时间t(该算法的实现需要专门的硬件支持，虽然算法性能好，但是实现困难，开销大)。当需要淘汰一个页面时，选择现有页面中t值最大的，即最近最久未使用的页面。

LRU性能较好，但需要寄存器和栈的硬件支持。LRU是堆栈类算法，理论上可以证明，堆栈类算法不可能出现Belady异常。



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。

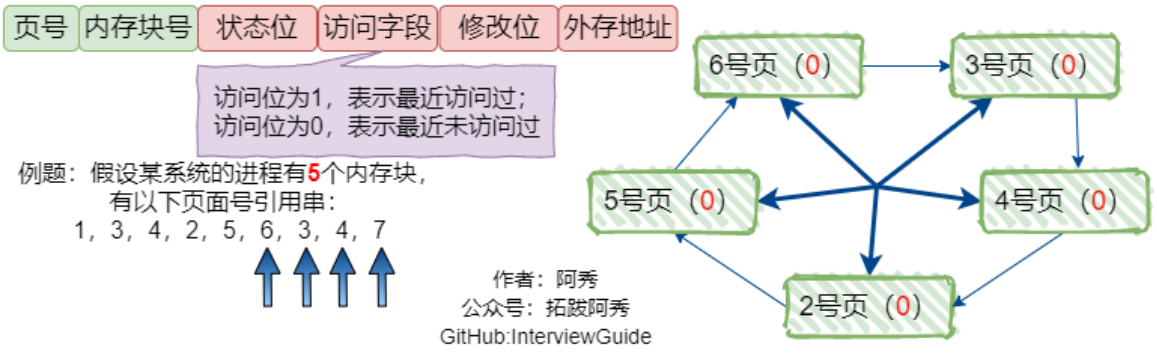
4、时钟置换算法(CLOCK)

最佳置换算法性OPT能最好，但无法实现；先进先出置换算法实现简单，但算法性能差；最近最久未使用置换算法性能好，是最接近OPT算法性能的，但是实现起来需要专门的硬件支持，算法开销大。

所以操作系统的设计者尝试了很多算法，试图用比较小的开销接近LRU的性能，这类算法都是CLOCK算法的变体，因为算法要循环扫描缓冲区像时钟一样转动。所以叫clock算法。

时钟置换算法是一种性能和开销较均衡的算法，又称CLOCK算法，或最近未用算法(NRU，Not Recently Used)

简单的CLOCK算法实现方法:为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位置为1。当需要淘汰一个页面时，只需检查页的访问位。如果是0，就选择该页换出;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(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)



5、改进型的时钟置换算法

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。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。

改进型的Clock算法需要综合考虑某一内存页面的访问位和修改位来判断是否置换该页面。在实际编写算法过程中，同样可以用一个等长的整型数组来标识每个内存块的修改状态。访问位A和修改位M可以组成一下四种类型的页面。

算法规则：将所有可能被置换的页面排成一个循环队列

> 第一轮：从当前位置开始扫描到第一个（A =0，M = 0）的帧用于替换。表示该页面最近既未被访问，又未被修改，是最佳淘汰页
> 第二轮：若第一轮扫描失败，则重新扫描，查找第一个（A =1，M = 0）的帧用于替换。本轮将所有扫描过的帧访问位设为0。表示该页面最近未被访问，但已被修改，并不是很好的淘汰页。
> 第三轮：若第二轮扫描失败，则重新扫描，查找第一个（A =0，M = 1）的帧用于替换。本轮扫描不修改任何标志位。表示该页面最近已被访问，但未被修改，该页有可能再被访问。
> 第四轮：若第三轮扫描失败，则重新扫描，查找第一个A =1，M = 1）的帧用于替换。表示该页最近已被访问且被修改，该页可能再被访问。

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

算法规则：将所有可能被置换的页面排成一个循环队列

第一轮: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。(第一优先级:最近没访问，且没修改的页面)

第二轮: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
(第二优先级: 最近没访问，但修改过的页面)

第三轮: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(第三优先级:最近访问过，但没修改的页面)

第四轮: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。(第四优先级:最近访问过，且修改过的页面)
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

6、总结

	算法规则	优缺点
OPT	优先淘汰最长时间内不会被访问的页面	缺页率最小，性能最好;但无法实现
FIFO	优先淘汰最先进入内存的页面	实现简单;但性能很差，可能出现Belady异常
LRU	优先淘汰最近最久没访问的页面	性能很好;但需要硬件支持，算法开销大
CLOCK (NRU)	循环扫描各页面 第一轮淘汰访问位=0的，并将扫描过的页面访问位改为1。若第一轮没选中，则进行第二轮扫描。	实现简单，算法开销小;但未考虑页面是否被修改过。
改进型CLOCK (改进型NRU)	若用(访问位，修改位)的形式表述，则 第一轮:淘汰(0,0) 第二轮:淘汰(0,1)，并将扫描过的页面访问位都置为0 第三轮:淘汰(0，0) 第四轮:淘汰(0，1)	算法开销较小，性能也不错

内存分配策略

最小物理块数　　即能保证进程正常运行所需的最小物理块数

物理块分配策略

两两结合，就是少了 固定分配全局置换 - 固定+

- 固定分配局部置换
- 可变分配全局置换
- 可变分配局部置换

缺页中断

在请求分页系统中，每当所要访问的页面不在内存时，就会产生缺页中断

调页策略

- 请求调页
- 预调页

抖动问题

即刚被换出的页很快又要被访问，需要将它重新调入，此时又需要再选一页调出

原因　　进程太多了

只有先进先出算法会出现

预防方法

- 局部置换策略
- 工作集算法
- L=S准则
- 挂起一些进程

请求分段虚拟存储

简单

- 共享段
  - 可重入代码/纯代码 – 允许多个进程同时访问

7用户接口

简单 – 略

概念

OS接口

- 命令接口
  - 脱机命令
  - 交互式命令
- 程序接口
- 图形接口

系统调用

类似中断处理

略

8文件

存储介质

- 顺序存储设备　　磁带
- 直接存储设备　　磁盘

结构

逻辑结构

- 纪录式(有结构)文件
- 流式(无结构)文件

物理结构

(讲直接存储设备)

见外存分配方式

- 连续文件
- 连接文件
  - 隐式连接
  - 显式连接
    - FAT
- 索引文件
- 直接文件
- 物理文件 of NTFS文件系统

存放形式

存放形式 + 对应存取方法

顺序结构	链接结构	索引结构
顺序	顺序	顺序
随机		随机

分配方式

文件的物理结构与外存分配方式有关，在采用连续分配方式时的文件物理结构是顺序式的文件结构，在采用链接分配方式将形成链接式文件结构，而索引分配方式将形成索引式文件结构。

连续分配

连续分配要求为每个文件分配一组相邻接的盘块，一组盘块地址定义了磁盘上的一段线性地址。采用连续分配方式时，**可把逻辑文件中的记录顺序地存储到邻接的各物理盘块中，这样所形成的文件结构称为顺序文件结构**

链接分配

如果将一个逻辑文件存储到外存上，并不要求为整个文件分配一块连续的空间，而是可以将文件装到多个离散的盘块中，这样就可以消除连续分配的缺点。采用链接分配方式时，可通过在每个盘块上的链接指针，**将同属于一个文件的多个离散盘块链接成一个链表**，把这样形成的物理文件称为链接文件。链接分配采取离散分配方式，消除了外部碎片，故而显著地提高了外存空间的利用率，并且对文件的增、删、改、查十分方便。链接方式可分为隐式链接和显示链接两种形式。

隐式链接

在文件目录的每个目录项中，都须含有**指向链接文件第一个盘块和最后一个盘块的指针。**

显示链接

把用于链接文件各物理块的指针，显式的放在内存的一张链接表中，该表在整个磁盘仅设置一张。此为文件分配表FAT (File Allocation Table)

索引分配

在打开某个文件时，只需要把该文件占用的盘块号的编号调入内存即可，完全没有必要把整个FAT调入内存，为此，应该**将每个文件所对应的盘块号集中地放在一起**，索引分配方式就是基于这种想法所形成的一种分配方式。其为每个文件分配一个索引块（表），再把分配给该文件的所有盘块号都记录在该索引块中，因而该索引块就是一个含有许多磁盘块号的数组。在建立一个文件时，只需要在位为之建立的目录项中填上指向该索引块的指针（单级索引）。

混合索引分配方式

目录

文件控制块（FCB）

- 文件名+inode(属性)

简单的文件目录

- 单级文件目录
  - 查找慢
  - 不允许重名
  - 不便于实现文件共享
- (二)多级文件目录
  - 提高检索速度，从M*N到M+N

FCB

- FCB的有序集合称为文件目录，一个FCB就是一个文件目录项
- 逻辑结构的组织形式取决于用户，物理结构的组织形式取决于储存介质特性
- 文件目录项（FCB）不包括FCB的物理位置

改进(分解)

索引节点

(M+1)/2 > (N+1)/2 + 1 才可以改进(前的平均查找次数大于改进后+1)

平均访问盘块数

> 一个盘块存放几个目录项?

目录占用盘块数 M

(M + 1) / 2

存储空间管理

空闲块表法

空闲表法属于连续分配方式，它与内存的动态分配方式雷同，它为每个文件分配一块连续的存储空间，即**系统也为外存上所有空闲区建立一张空闲表，每个空闲区对应于一个空闲表项**，其中包括表项序号、该空闲区的第一个盘块号、该区的空闲盘块号等信息，再将所有空闲区按其起始盘块号递增排列。

序号	第一空闲盘块号	空线盘块数
1	2	4
2	9	3
3	15	5
4	-	-

空闲盘区的分配与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。系统在对用户所释放的存储空间进行回收时，也采取类似于内存回收的方法，即考虑回收区是否与空闲表中插入点的前区和后区相邻接，对相邻接者应该予以合并。当文件较小时，采用连续分配方式，当文件较大时，可采用离散分配方式。

空闲链表法

- 空闲盘块链
- 空闲盘区链

位示图法

类似页式存储系统

成组连接法

略

使用

共享

- 符号链
- 索引节点

安全性

影响因素

- 人为
- 系统
- 自然

防止系统因素

坏块管理

磁盘容错

1. 第一级容错技术
    - 双份目录 + 双份文件分配表
    - 热修复重定向 + 写后读校验
2. 第二级容错技术
    - 磁盘镜像
    - 磁盘双工
3. 廉价磁盘冗余阵列

> - RAID 0~6级

备份

一致性

略

磁盘调度

IO时间

- 查找(可改进)
- 等待
- 传输

时间关系

寻道时间

读写头沿径向移动，移到要读取的扇区所在磁道的上方

旋转延迟时间

通过盘片的旋转，使得要读取的扇区转到读写头的下方

= 旋转一周所需时间 / 2

例

一个7200（转 /每分钟）的硬盘，每旋转一周所需时间为60×1000(秒->毫秒)÷7200＝8.33毫秒，则平均旋转延迟时间为8.33÷2＝4.17毫秒（最多旋转1圈，最少不用旋转，平均情况下，需要旋转半圈）

存取时间

平均寻道时间与平均旋转延迟时间之和称为平均存取时间

移臂调度AL

- FCFS　先来先服务
  - 优点：公平，简单
  - 缺点：可能导致某些进程的请求长期得不到满足
- SSTF　最短寻道时间优先
- SCAN　扫描算法/电梯调度
  - 扫描算法不仅考虑到欲访问的磁道与当前磁道间的距离，更优先考虑的是磁道当前的移动方向
  - 可防止低优先级进程出现“饥饿”的现象
- CSCAN　循环扫描/圆形电梯调度
  - 仅在一个方向上处理请求
  - 规定磁头单向移动，这里不算跳的距离
- NStepScan　N步扫描算法
  - 子队列依次处理
- FSCAN
  - 使用子队列简化N步SCAN

9设备

层次关系

通道–> 控制器 –> 设备

> 通道控制控制器，设备在控制器控制下工作

为了实现设备独立性应该设置一张逻辑设备表

设备独立性

基本含义　　应用程序独立于具体物理设备

设备类型

直接存取设备　　磁盘

顺序存取设备　　磁带

设备控制器

主要功能：控制一个或多个I/O设备，以实现I/O设备和计算机之间的数据交换

基本功能

- 接收和识别命令
- 数据交换
  - 实现CPU与控制器，控制器与设备间的数据交换
- 标识和报告设备的状态
- 地址识别
- 数据缓冲区
- 差错控制

通道

是一种特殊的处理机，具有通过执行通道程序完成I/O操作的指令

特点

- 指令单一(局限于与I/O操作相关的指令)
- 与CPU共享内存

目的

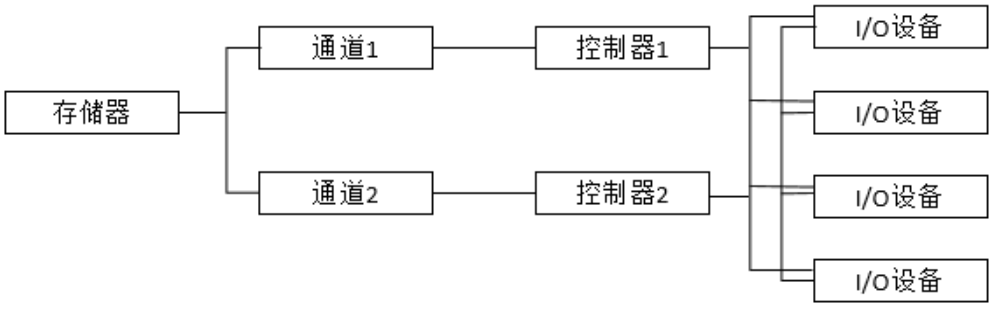
建立独立的I/O操作(组织，管理和结束)，使由CPU处理的I/O工作转由通道完成（解放CPU，实现并行)

基本过程

- CPU向通道发出I/O指令->通道接收指令->从内存取出通道程序处理I/O->向CPU发出中断

性能瓶颈

- 原因是通道不足
- 解决办法：增加设备到主机间的通路，而不增加通道



类型

- 字节多路通道
  - 低中速连接子通道时间片轮转方式共享主通道
  - 字节多路通道不适于连接高速设备，这推动了按数组方式进行数据传送的数组选择通道的形成。
- 数组选择通道
  - 这种通道可以连接多台高速设备，但只含有一个分配型子通道，在一段时间内只能执行一道通道程序， 控制一台设备进行数据传送， 直至该设备传送完毕释放该通道。这种通道的利用率很低。
- 数组多路通道
  - 含有多个非分配型子通道，前两种通道的组合，通道利用率较好

I/O控制方式

宗旨

减少CPU对I/O控制的干预， 充分利用CPU完成数据处理工作

四个方式从上往下CPU干预程度越来越低，CPU利用率越来越高

- 程序查询
- IO中断
- DMA（直接存储器访问）
- IO通道

缓冲

引入原因

- 缓和CPU与I/O设备间速度不匹配的矛盾
- 减少对CPU的中断频率，放宽对CPU中断响应时间的限制
- 提高CPU和I/O设备之间的并行性

类型

- 单缓冲
- 双缓冲
- 循环缓冲
- 缓冲池

缓冲池

（理解为更大的缓冲区）

组成

三类缓冲区(空 + 满(输入数据) + 满(输出数据))链接在一起组成

- 空缓冲队列（emq）
- 输入队列（inq）
- 输出队列（outq）

Getbuf+Putbuf过程

- 收容：缓冲池接收外界数据
- 提取：外界从缓冲池获得数据

工作方式

- 收容输入
- 提取输入
- 收容输出
- 提取输出

设备驱动程序

主要任务是接受来自它上一层的与设备无关软件的抽象请求，并执行这个请求

功能

- 接收由I/O进程发来的命令和参数， 并将命令中的抽象要求转换为具体要求
- 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，设置设备的工作方式。
- 发出I/O命令，如果设备空闲，便立即启动I/O设备去完成指定的I/O操作；如果设备处于忙碌状态，则将请求者的请求块挂在设备队列上等待。

- 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。
- 对于设置有通道的计算机系统，驱动程序还应能够根据用户的I/O请求，自动地构成通道程序。

DMA控制器组成

- 主机与DMA控制器的接口
- DMA控制器与块设备的接口
- I/O控制逻辑

设备分配

数据结构

显然，在有通道的系统中，一个进程只有获得了通道，控制器和所需设备三者之后，才具备了进行I/O操作的物理条件

- 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
- 逻辑设备表LUT

考虑因素

- 设备固有属性
- 设备分配算法
- 设备分配中的安全性

独占设备的分配程序

基本分配程序

- 分配设备
- 分配控制器
- 分配通道

改进的分配程序

- 增加设备独立性
- 考虑多通路情况

虚拟设备

Spooling假脱机系统

是对脱机输入/输出系统的模拟

组成

- 输入/输出井
- 输入/输出缓冲区
- 输入/输出进程

共享打印机

工作流程

- 由输出进程在输出井中为之申请一块空闲盘块区，将要打印的Data存入
- 输出进程为用户进程申请一张空白的用户请求打印表，并将用户的打印要求填入，再讲其挂到请求打印队列上

> 并不会真正立刻把设备给对象