# 一.操作系统概述--By LZH

## (一)操作系统的概念、特征、功能和提供的服务

### <1>概念

操作系统是配置在计算机硬件上的第一层软件，是对硬件系统的首次扩充。其主要作用是管理好这些设备，提高他们的利用率和系统的吞吐量，并为用户和应用程序提供一个简单的接口，便于用户使用。(第一章第一段)

### <2>操作系统的特征 （1.3）

#### #1并发

计算机系统中同时存在多个运行的程序，需要OS管理和调度。

**并行性**:两个或者多个事件在**同一时刻**发生。

**并发性**:两个或者多个事件在**同一时间间隔**内发生。

#### #2共享

**“同时”访问**:有的资源允许一段时间内由多个进程"同时"对它们进行访问，"同时"在微观上意义上，进程对该资源的访问是交替进行的。

**互斥共享**:有的资源虽然可以供给多个进程使用，但是一段时间内只允许一个进程访问该资源，故建立进程对这类资源的互斥访问。

#### #3虚拟

**利用多道程序设计技术，让每个用户都觉得有一个计算机专门为他服务**

**时分复用**：利用某设备为一用户服务的空闲时间内，又转去为其他用户服务。通过利用处理机的空闲时间运行其它程序，提高处理的利用率

**空分复用**：利用存储器的空闲空间分区域存放和运行其他的多道程序，以此提高内存的利用率。

#### #4异步

程序的执行不是一贯到底，而是走走停停，向前推进的速度不可预知，这就是进程的异步性。

但是只要运行环境相同，OS需要保证程序运行的结果也要相同

### <3>操作系统的功能和提供的服务（1.4）

#### #1处理机管理功能

进程控制：为作业创建进程、撤销（终止）已结束的进程、控制进程在运行过程中的状态转换

进程同步：为多个进程（线程）的运行进行协调。

**进程通信：实现相互合作进程之间的信息交换。**

### #2存储器管理功能

内存分配

内存保护

地址映射

内存扩充

### #3.设备管理功能

缓冲管理：缓和CPU和I/O设备速度不匹配的矛盾。

设备分配：设备控制表、控制器控制表

设备处理：设备驱动程序，实现CPU和设设备控制器之间的通信。

### #4.文件管理系统

文件存储空间的管理

目录管理

文件的读 / 写管理和保护

### #5.操作系统与用户之间的接口

用户接口

程序接口

### #6.现代操作系统的新功能

系统安全

网络的功能和服务

支持多媒体

# (二)操作系统的发展与分类

## <1>操作系统的演变（1.2）

### #1单用户系统（’45-’55)-书上没有

■ 操作系统=装载器+通用子程序库

■ 问题：昂贵组件的低利用率

### #2单道批处理系统（’55-’65)

■ 顺序执行与批处理

■ 主要缺点:系统中的资源得不到充分利用，因为内存中只有一道程序

### #3多道批处理系统（’65-’80)

■ 保持多个工作在内存中并且在各工作间复用CPU

■ 多道程序交替执行，交替的条件是前一个正在执行的程序主动让出CPU的使用权。

■ 多道批处理系统主要考虑的是系统效率和系统的吞吐量(优点)。

■ 多道批处理系统缺点:平均周转时间长，无交互能力

### #4分时（'70-)

■ 定时中断用于工作对CPU的复用

■ 交互性和及时性是分时系统的主要特征。

### #5实时操作系统

■ 实时系统的正确性，不仅有计算的逻辑结果来决定，还取决于产生结果的时间。

■ 实时任务的类型：

周期性实时任务和非周期性实时任务，前者是指外部设备周期性地发出激励信号给计算机，使其周期性执行，以便周期性地控制某外部设备，后者无明显的周期性，但是都必须联系着一个截止时间。

硬实时和软实时，前者必须满足对截止时间的要求，后者对此没有严格要求

# (三)操作系统体系结构

### <1>传统结构

无结构

模块化

分层

自底向上的分层原则，确定每一步的设计都是建立在可靠的基础上。

### <2>现代结构

微内核结构

只将最基本的部分放入微内核中。

# (四)操作系统的启动流程

# (五)操作系统的运行环境

## <1>内核态与用户态

防止OS本身及相关数据遭到应用程序或无意的破坏，通常将处理机的执行状态分为：

系统态（内核态，内核态又称为管态）：高权限，能访问所有的寄存器。

用户态：低权限，能访问指定的寄存器。

**例子:**

**CPU执行操作系统代码的时候称为处理机处于管态。**

**函数调用并不会切换到内核态，而除零操作引发中断，中断和系统调用都会切换到内核态进行相应处理。**

# <2>中断、异常、系统调用 (切换到内核态)

## #1为什么需要中断、异常和系统调用

计算机的一些功能只有内核有权利访问，通过中断、异常和系统调用为应用程序提供方便。

在计算机运行中，内核是被信任的第三方

只有内核可以执行特权指令

方便应用程序

## #2中断和异常希望解决的问题(用于解决意外的情况)

当外设连接计算机时，会出现什么现象？(中断)

当应用程序处理意想不到的行为时，会出现什么现象？(异常)

## #3系统调用希望解决的问题

**用户应用程序是如何得到系统服务?**

**通过调用函数库，函数库又会调用对应的系统调用接口，从而得到系统服务。**

**系统调用和功能调用的不同之处是什么?**

系统调用时会有堆栈切换和特权级的转换，INT和IRET用于系统调用。

功能调用时没有堆栈切换，CALL和RET用于功能调用。

## #4无法预计用户在什么时候敲键盘\除法除以0

我们要解决用户程序如何来解决系统的服务。就好象说我们提供给银行对外提供服务，银行为了保证安全，它有很多的防护，这个防护又和对外提供服务这是有矛盾的。

为了方便用户来使用银行的服务，应该必须提供灵活的访问接口，又不能影响到银行的安全。

操作系统内核也是一样的，我们需要来通过系统调用来提供一个接口，让应用程序既方便的使用内核提供的服务，又不至于用户的行为对我内核的安全产生影响

## #5定义

**系统调用 (system call)**

- 应用程序 **主动** 向操作系统发出的服务器请求
- **系统调用是应用程序向操作系统发出服务请求并获得操作系统服务的唯一通道和结果（操作系统与用户的接口）。**

**异常 (exception)**

- 非法指令或其他原因导致当前 **指令执行失败**
    - 如: 内存出错后的处理请求

**中断(hardware interrupt)**

- 来自硬件设备的处理请求

**中断向量地址**

- 即存储中断向量的存储单元地址，中断服务例行程序入口地址

# <3>中断、异常和系统调用的比较

## #1源头

- 中断：外设
- 异常：应用程序意想不到的行为
- 系统调用：应用程序请求操作系统提供服务

## #2响应方式

- 中断：异步
- 异常：同步
- 系统调用：异步或同步

## #3处理机制

- **中断：持续，对用户应用程序是透明的**
- **异常：杀死或重新执行意想不到的应用程序指令**
- **系统调用：等待和持续**

## #4中断(此处为三者总称)处理机制

- **硬件处理**
  - 在CPU初始化时设置 **中断使能** 标志
    - 依据内部或外部事件设置**中断标志**
    - 依据 **中断向量** 调用相应 **中断服务** 例程
  - **注释:1)** 在许可外界打扰CPU的执行之前，CPU是不会对外界的任何中断请求发出响应的。

    **2)** 生成中断标志，通常是一个电平的上升沿或者说是一个高电平，CPU会记录这个事件。

    我有一个中断标志，表示出现了一个中断，什么设备产生的，需要知道中断源的编号。
- **软件**
  - 现场保持 **(编译器)**
  - 中断服务处理 **(服务例程)**
  - 清除中断标记 **(服务例程)**
  - 现场恢复 **(编译器)**
  - 注: 都到了**中断向量表**，中断--中断服务例程，异常--异常服务例程，系统调用--总共占有一个中断编号，不同系统调用功能由系统调用表来表示的。由系统调用表的不同，选中不同的系统调用实现。
  - 如果是系统调用，由于系统调用的量很大，他在中断向量表里只占一个中断编号，不同的系统调用的功能由系统调用表实现。

    需要保存上下文信息。

## #5中断(此处为三者总称)嵌套

- **硬件中断服务例程可以被打断**
    - 不同**硬件中断源**可能**硬件中断处理**时出现
    - **硬件中断服务例程**中需要**临时禁止中断** 请求
    - 中断请求会保持到cpu做出响应
    - **中断处理例程**（也可称为中断处理程序）需要**执行打开中断，关闭中断等特权指令**，而这些指令只能在**内核态**下才能正确执行，所以中断处理例程位于操作系统内核中。
- **异常服务例程可被打断**
    - 异常服务例程执行时可能出现硬件中断
- **异常服务例程可嵌套**
- 
    - 异常服务例程可能出现缺页

### #6中断例子

我正在处理一个请求的时候，又来了一个请求这时候我怎么办，那我们说在操作系统的里头呢?

它是硬件的中断，它是允许被打断的，也就是说我正在处理一个中断的时候，可以允许你再出现其他的中断。

如果两个中断源不同，那这时候我可以通过优先级的高低，让一个往后推一段，或者说让一个暂停下来，那使得我可以同时在做交替在做处理。

中断服务例程里头，并不是说我任何一个时刻都可以做任何一个处理，它会在一定的时间里禁止中断请求。

比如说我电源有问题，那可能其他的问题就变得不重要了，这时候我在做电源的处理的时候，我就会禁止掉其他中断。中断服务请求会一直保持到CPU做出响应。

比如说我在这里头虚拟存储里头，它访问到的存储单元的数据不存在，我正在从硬盘上倒数据进来。倒的过程当中，它会用到磁盘I/O，这时候也会再有磁盘设备的中断，这时候是允许它可以做嵌套的。

### #7注意点

系统调用是提供给应用程序使用的，由用户态发出，进入内核态执行。外部中断随时可能发生；应用程序执行时可能发生缺页；进程切换完全由内核来控制。

## <4>系统调用

1. **操作系统服务的编程接口**
2. **通常由高级语言编写(C或者C++)**
3. **程序访问通常是通过高层次的API接口而不是直接进行系统调用**
4. 三种最常用的应用程序编程接口(API)

- Win32 API 用于WIndows
- POSIX API 用于POSIX-based systems(包括UNIX，LINUX，Mac OS X)
- Java API 用于java虚拟机(JVM)

# 二.进程管理

## (一)进程与线程

# <1>进程概念

## #1定义

进程是指一个具有一定**独立功能的程序**在一个**数据集合**上的一次**动态执行**过程。

## #2组成

进程包含了正在运行的一个程序的**所有状态信息**：

1. 代码
2. 数据
3. 状态寄存器

- CPU状态CR0，指令指针IP

4. 通用寄存器

- AX,BX,CX

5. 进程占用系统资源

- 打开文件、已分配内存

## #3特点

1. 动态性:可动态地创建、结束进程
2. 并发性:进程可以被独立调度并占用处理机运行
3. 独立性:不同进程的工作不相互影响
4. 制约性:因访问共享数据/资源或进程间同步而产生制约

## #4进程与程序的联系

1. 进程是操作系统处于执行状态程序的抽象

- 程序 = 文件 (静态的可执行文件)
- 进程 = 执行中的程序 = 程序 + 执行状态

2. 同一个程序的多次执行过程对应为不同进程

- 如命令“ls”的多次执行对应多个进程

3. 进程执行需要的资源

- 内存：保存代码和数据
- CPU：执行指令

## #5进程与程序的区别

1. 进程是动态的，程序是静态的

- 程序是有序代码的集合
- 进程是程序的执行，进程有核心态/用户态

2. 进程是暂时的，程序的永久的

- 进程是一个状态变化的过程

- 程序可长久保存

3. 进程与程序的组成不同

- 进程的组成包括程序、数据和进程控制块

## <2>进程控制块（PCB，Process Control Block）

### #1概念

1. 操作系统管理控制进程运行所用的信息集合
2. 操作系统用PCB来描述进程的基本情况以及运行变化的过程
3. PCB是进程存在的唯一标志

- 每个进程都在操作系统中有一个对应的PCB

### #2进程控制块的使用

1. 进程创建

- 生成该进程的PCB

2. 进程终止

- 回收它的PCB

3. 进程的组织管理

- 通过对PCB的组织管理来实现

### #3进程控制块PCB的内容

1. 进程标识信息
2. 处理机现场保存
3. 进程控制信息

- 调度和状态信息
  - 调度进程和处理机使用情况
- 进程间通信信息
  - 进程间通信相关的各种标识
- 存储管理信息
  - 指向进程映像存储空间数据结构
- 进程所用资源
  - 进程使用的系统资源，如打开文件等
- 有关数据结构的连接信息
  - 与PCB相关的进程队列
  - 进程状态的变化体现于其进程控制块所在的链表，通过进程队列实现。

### #4进程控制块的组织方式

1. 链表形式:同一状态的进程的PCB组成一个链表，多个状态对应多个不同的链表

- 各个状态的进程形成不同的链表:就绪链表，阻塞链表

2. 索引表形式:同一状态的进程归于一个索引表(由索引指向PCB)，多个状态对应多个不同的索引表

- 进程状态的变化体现于其进程控制块所在的链表，通过进程队列实现。

# <3>进程状态与转换

## #1进程的生命周期划分

1. 进程创建
2. 进程执行
3. 进程等待
4. 进程抢占
5. 进程唤醒
6. 进程结束

## #2导致进程创建的情况

1. 系统初始化时
2. 用户请求创建一个新进程
3. 正在运行的进程执行了创建进程的系统调用

## #3进程执行

内核选择一个就绪的进程，让它占用处理机并运行

如何选择？处理机调度算法

## #4进程进入等待(阻塞)的情况

只有进程本身才知道何时需要等待某种事件的发生，即导致其进入等待状态的一定是进程本身内部原因所导致的，不是外部原因所导致的。

1. 请求并等待系统服务,无法马上完成
2. 启动某种操作，无法马上完成
3. 需要的数据没有到达

## #5进程被抢占的情况

1. 高优先级的进程变成就绪状态
2. 调度算法为每个进程设置的时间片，进程执行的时间片用完了，操作系统会抢先让下一个进程投入运行。

## #6唤醒进程的情况

**进程只能被别的进程或者操作系统给唤醒。**

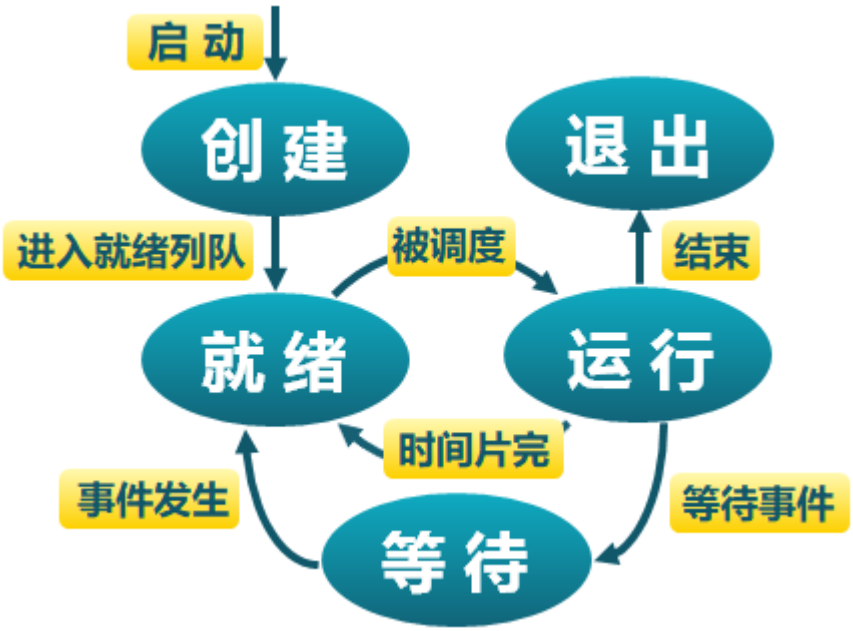
1. 被阻塞进程需要的资源可被满足
2. 被阻塞进程等待的事件到达

## #7进程结束的情况

1. 正常退出(自愿的)
2. 错误退出(自愿的)
3. 致命错误(强制性的)
4. 被其他进程所杀(强制性的)

5. **进程退出了，但还没被父进程回收，此时进程处于zombie态**

## <3>三状态进程模型




### #1每种状态的含义

**1.运行状态(Running)**

- 进程正在处理机上运行

**2.就绪状态(Ready)**

- 进程获得了除了处理机之外的所有所需的资源，得到处理机即可运行

**3.等待状态(有成阻塞状态Blocked)**

- 进程正在等待某一事件的出现而暂停运行

**4.创建状态(New)**

- 一个进程正在被创建，还没被转到就绪状态之前的状态，是一个过渡状态。也就是在分配资源和相应的数据结构

**5.退出状态(Exit)**

- 一个进程反正在从系统中消失时的状态，这是因为进程结束或者由于其他原因所致(也就是系统正在回收资源)

### #2各种状态的变迁

**1.NULL->创建(启动)**

一个新进程被产生出来执行一个程序

**2.创建->就绪(进入就绪队列)**

当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态

**3.就绪->运行(被调度)**

处于就绪状态的进程被进程调度程序选中后，就分配到处理机上运行

**4.运行->结束(结束)**

当进程表示它已经完成或者因出错，当前运行进程会由操作系统作结束处理(回收资源)

**5.运行->就绪(时间片完或者被抢先)**

处于运行状态的进程在其运行期间，由于分配给它的处理时间片用完而让出处理机

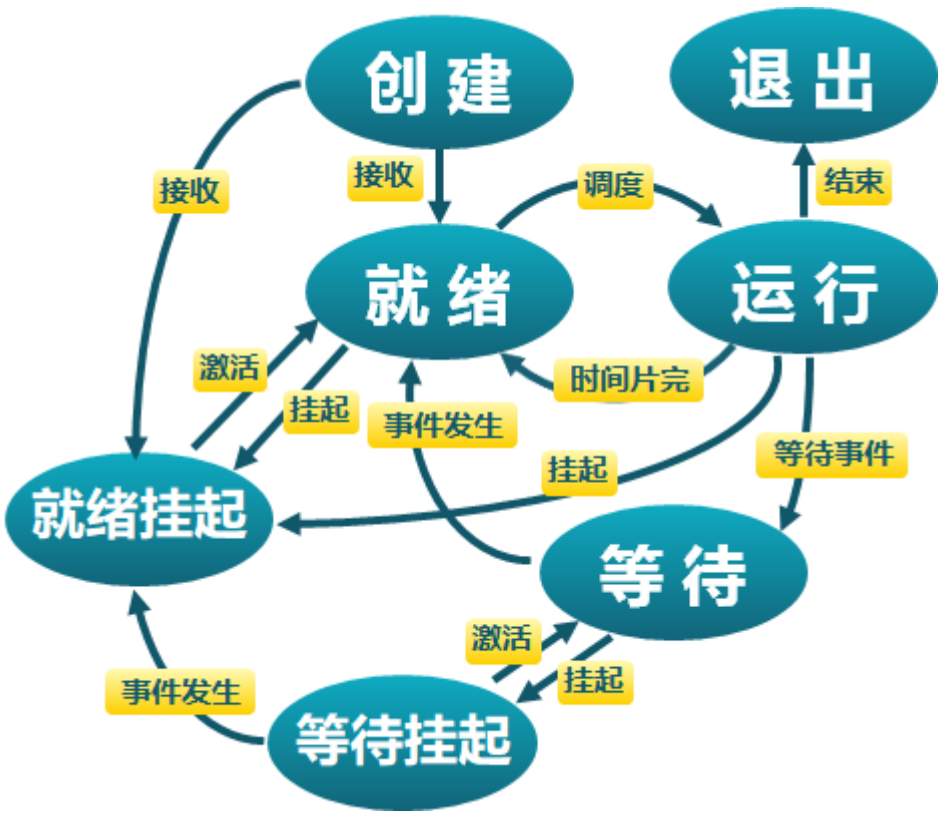
**6.运行->等待(等待事件)**

当进程请求某资源且必须等待时

**7.等待(阻塞)->就绪(事件发生)**

当进程要等待某事件到来时，它从阻塞状态变到就绪状态。

## <4>进程挂起




### #1作用

处在挂起状态的进程映像在磁盘上，目的是减少进程占用内存

### #2每种状态的含义

**1.等待挂起状态**

进程在外存并等待某事件的出现(多加了一个关于进程的位置信息)

**2.就绪挂起状态**

进程在外存，但只要进入内存，即可运行

(无法进入内存原因:内存空间不够或者进程本身优先级不够高)

### #3从内存到外存的变迁

**0.挂起:把一个进程从内存转到外存**

**1.等待->等待挂起:**

没有进程处于就绪状态或者就绪进程要求更多内存资源

**2.就绪->就绪挂起:**

当有高优先级等待(系统认为会很快就绪的)进程和低优先级就绪进程

**3.运行->就绪挂起:**

对抢先式分时系统，当有高优先级等待挂起进程因为事件出现而进入就绪挂起(比如内存不够)

### #4在外存时的状态变迁

**1.等待挂起->就绪挂起**

当有等待挂起进程因为相关事件出现

### #5激活:把一个进程从外存转到内存

**1.就绪挂起->就绪**

没有就绪进程或者挂起就绪进程优先级高于就绪进程

**2.等待挂起->等待**

当一个进程释放足够内存，并有高优先级等待挂起进程

### #6状态队列

**1.由操作系统来维护一组队列，表示系统中所有进程的当前状态**

**2.不同队列表示不同状态**

就绪队列、各种等待队列

**3.根据进程状态不同，进程PCB加入相应队列**

进程状态变化时，它所在的PCB会从一个队列

换到另一个

## <5>进程通信

### #1基本概念

进程通信是进程之间的信息交换，是进程进行通信和同步的机制。

### #2消息传递系统

**进程不借助任何共享存储区或数据结构，而是以格式化的消息（message）为单位，将数据封装在消息中，并利用操作系统提供一组通信命令（原语）完成信息传递和数据交换。**

## #2.1消息传递系统中实现方式

### 1.直接消息传递系统(消息缓冲队列(教材中))

发送进程利用OS所提供的通信指令，直接把消息放到目标进程

#### [1]直接通信原语

（1）对称寻址方式；该方式要求发送和接受进程必须以显示方式提供对方的标识符。

系统提供命令:

```
send(receiver,message);//发送一个消息给接受进程receiver
receive(sender,message);//接受进程sender发来的消息
```

（2）非对称寻址方式；在接受程序原语中，不需要命名发送进程。

系统提供命令:

```
send(P,message);//发送一个消息给接受进程P
receive(id,message);//接受来自任何进程的消息，id变量可以设置为发送进程的id或者名字
```

#### [2]消息格式

（1）定长（消息长度）

（2）变长（消息长度）

#### [3]进程的同步方式(同步机制，进程之间)

（1）发送阻塞，接收阻塞

（2）发送不阻塞，接收阻塞

（3）发送不阻塞，接收不阻塞

#### [4]对应通信链路的属性

建立方式：（1）显示建立链接命令；（2）发送命令自动建立链路

通信方式（1）单向（2）双向

### 2.直接消息传递系统的实例--消息缓冲队列

#### [1-1]数据结构--消息缓冲区

```
type struct message_buffer   {
    int sender;                          发送者进程标识符
    int size;                            消息长度
    char *text;                          消息正文
    struct message_buffer *next;         指向下一个消息缓冲区的指针
}
```

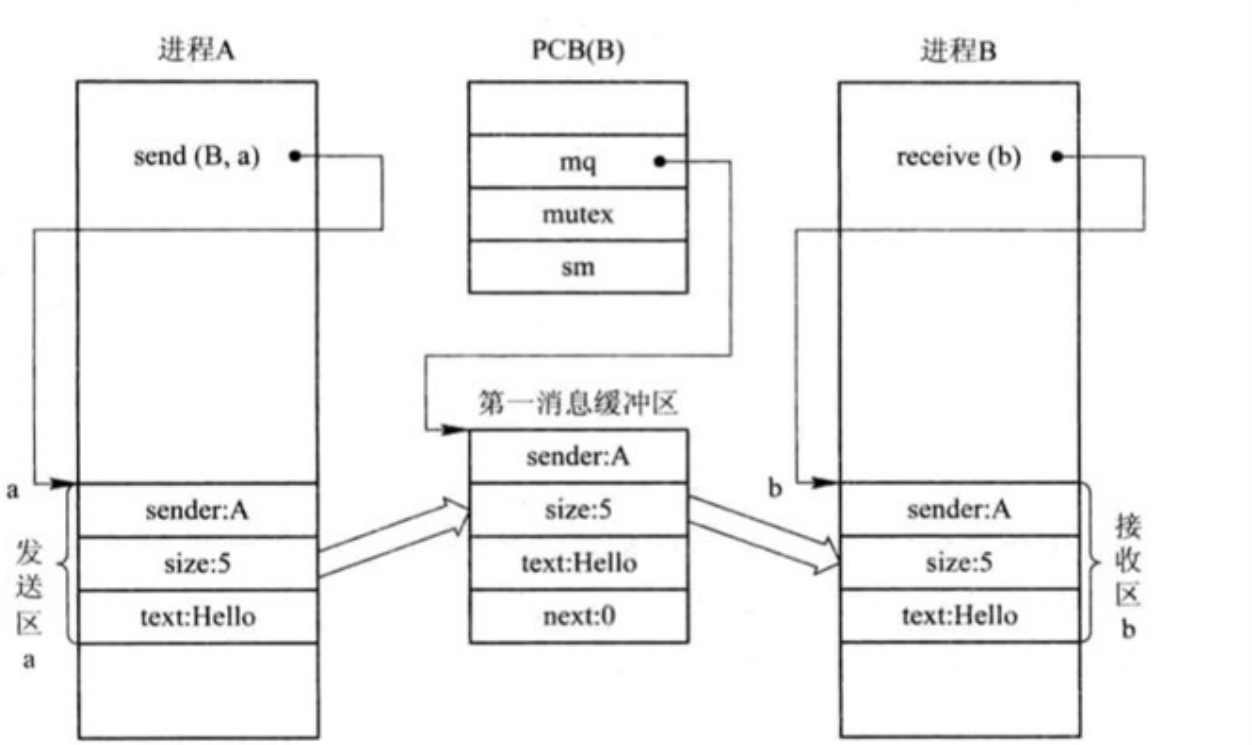
**[1-2]数据结构--PCB中关于通信的数据项**

(增加了消息队列的队首指针，互斥和资源信号量)

```
type struct processcontrol_block     {
    ...
    struct message_buffer *mq       ;      消息队列队首指针
    semaphore mutex;                       消息队列互斥信号量
    semaphore sm;                          消息队列资源信号量
    ...
}PCB;
```

**[2]发送原语**

发送原语首先根据发送区a中的消息长度a.size来申请一个缓冲区i，接着把a中的信息复制到缓冲区i中。获得接受进程内部标识符j,然后将i挂在j.mq上，由于该队列属于临界资源，所以执行insert前后都要执行wait和signal操作。




其中

```
mq//消息队列
mutex//对消息队列的互斥访问
sm//消息的资源信号量
```

发送原语可描述如下：

```
void send(receiver, a)  {                  receiver 为接收进程标识符，a 为发送区首址；
    getbuf(a.size, i);                     根据 a.size 申请缓冲区；
    copy(i.sender, a.sender);              将发送区 a 中的信息复制到消息缓冲区 i 中；
    i.size=a.size;
    copy(i.text, a.text);                  将发送进程的消息复制到消息缓冲区中
    i.next=0;
    getid(PCBset, receiver.j);             获得接收进程内部的标识符；
    wait(j.mutex);
    insert(&j.mq, i);                      将消息缓冲区插入消息队列；
    signal(j.mutex);
    signal(j.sm);                          将消息缓冲区挂到接收进程的消息队列mq上
}
```

**[3]接受原语**

调用接受原语receive(b),从自己的消息缓冲队列mq中摘下第一个消息缓冲区i,并将其中的数据复制到以b为首地址的指定消息接收区内。

```
void receive(b)  {
    j = internal name;                     j 为接收进程内部的标识符；
    wait(j.sm);
    wait(j.mutex);
    remove(i.mq, i);                       将消息队列中第一个消息移出；
    signal(j.mutex);
    copy(b.sender, i.sender);              将消息缓冲区 i 中的信息复制到接收区 b；
    b.size =i.size;
    copy(b.text, i.text);
    releasebuf(i);                         释放消息缓冲区；
}
```

**3.间接消息传递系统(信箱)**

发送和 接收进程，通过共享中间实体（邮箱 ）完成通信。该实体建立在随机存储器的公用缓冲区上，用来暂存信息。

**[1]信箱结构--数据结构**

信箱头：用于存放信箱的描述信息，如信箱标识符等

信箱体：由若干个可以存放信息的信箱格组成，信箱格数目和大小是在创建信箱时确定的。


**[2]信箱通信原语**

（1）邮箱的创建和撤销

（2）消息的发送和接收

```
Send(mailbox,message);//将一个消息发送到指定邮箱
Receive(mailbox,message);//从指定邮箱中接受一个消息
```

**[3]邮箱的类型：**

（1）私用邮箱：只有创建者才能接收消息

（2）公用邮箱：操作系统创建

（3）共享邮箱：创建进程指明接收进程

**[4]使用邮箱通讯时，发送进程和接收进程之间的关系：**

（1）一对一：专用通信链路

（2）多对一：客户 / 服务器

（3）一对多：广播

（4）多对多：公共邮箱

## #3管道通信

### 1.概念

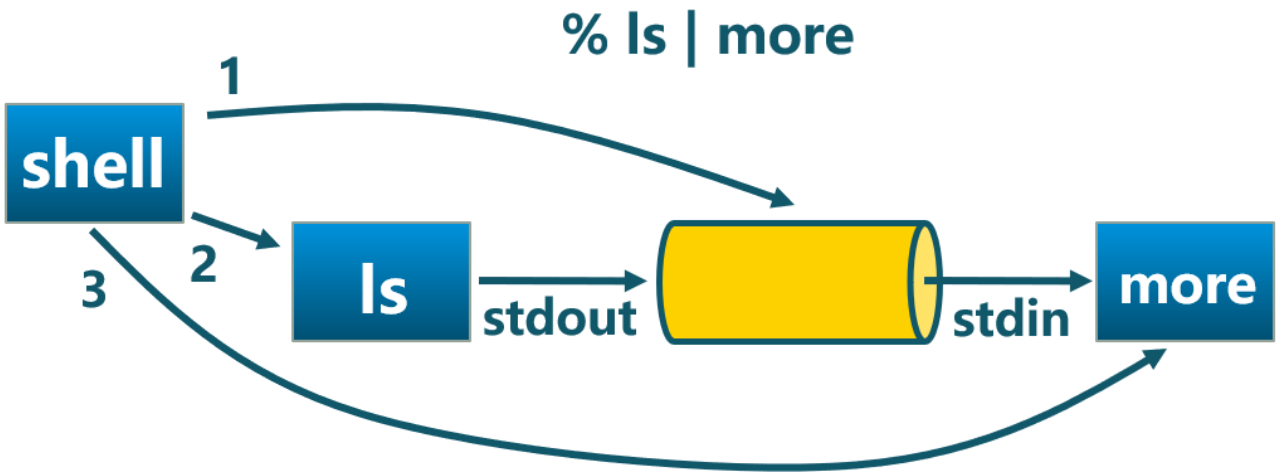
- 管道是进程间基于内存文件的通信机制;
  - 管道在父进程创建子进程过程中继承文件描述符;
  - 缺省文件描述符：0 stdin, 1 stdout, 2 stderr;
- 进程不关心另一端
  - 创建一个管道时，只关心通信的管道是谁，不关心另一端是谁放入的数据
  - 数据可能从键盘、文件、程序读取
  - 数据可能写入到终端、文件、程序

### 2.与管道相关的系统调用

- 读管道：read(fd,buffer, nbytes)

    - scanf()是基于它实现的
- 写管道：write(fd,buffer, nbytes)
    - printf()是基于它实现的
- 创建管道：pipe(rgfd)
    - 结果生成的rgfd是2个文件描述符组成的数组
    - rgfd**[0]是读**文件描述符
    - rgfd**[1]是写**文件描述符

**3.一个管道的例子**




其中

- shell
    - 在ls和more之间创建管道
    - 为ls创建一个进程, 设置stdout为管道写端
    - 为more创建一个进程,设置stdin为管道读端

## #4共享存储器

**1.共享内存概念**

共享内存是把同一个物理内存区域同时映射到多个进程的内存地址空间的通信机制

**2.在进程里时**

每个进程都有私有内存地址空间

每个进程的内存地址空间需明确设置共享内存段

**3.在线程里时**

同一进程中的线程总是共享相同的内存地址空间

**4.优点**

快速、方便地共享数据；

最快的通信方法；

一个进程写另外一个进程立即可见；

没有系统调用干预；

没有数据复制；

**5.不足**

不提供同步，必须用额外的同步机制来协调数据访问，比如由程序员提供同步

### #5嵌套字

**1.概念**

一个嵌套字就是一个通信标识类型的数据结构，包含通信目的的地址，端口号，传输层协议等

**2.分类**

基于文件型:两个进程都运行在同一台机器上,嵌套字基于本地文件系统支持

基于网络型:非对称通信方式，需要发送者提供接收者的命名

**3.优点**

不仅使用与同一台计算机内部的进程通信，而且适用于网络环境中不同计算机之间的进程通信。

## <6>线程概念与多线程模型

### #1为什么引入进程和线程?

在OS中引入进程是为了让多个程序能并发执行，来提高资源利用率和系统吞吐量。

在OS中映入线程是为了减少程序在并发执行时所付出的时空开销，使得OS有更高的并发性。

### #2线程的概念

线程**是进程的一部分**，描述指令流执行状态，它是**进程中的指令执行流的最小单元**，是**CPU调度的基本单位。**

**PCB变化**

- 进程的资源分配角色：进程由一组相关资源构成，包括地址空间（代码段、数据段）、打开的文件等各种资源
- 线程的处理机调度角色：线程描述在进程资源环境中的指令流执行状态

### #3.线程 = 进程 - 共享资源

- 线程的优点：
  - 一个进程中可以同时存在多个线程
  - 各个线程之间可以并发地执行
  - 各个线程之间可以共享地址空间和文件等资源
- 线程的缺点：
  - 一个线程崩溃，会导致其所属进程的所有线程都崩溃

### #4.线程与进程的比较

- 进程是资源分配单位，线程是CPU调度的单位
- 进程拥有一个完整的资源平台，而线程只独享指令流执行的必要资源，如寄存器和栈

- 原来和进程执行有关的状态都转为线程的状态，所以线程具有就绪、等待和运行三种基本状态和状态之间的转换关系
- 线程能减少并发执行的时间和空间开销
    - 线程的创建时间比进程短
    - 线程的结束时间比进程短
    - 同一进程内的线程切换时间比进程短
    - 同一进程的各个线程之间共享内存和文件资源，**可以不通过内核进程直接通信**

### #5.线程的三种实现方式

- 用户线程：在用户空间实现(函数库实现，不依赖内核)
    - POSIX Pthreads，Mach C-threads，Solaris threads
- 内核线程：在内核中实现(通过系统调用实现，由内核维护内核线程的线程控制块)
    - Windows，Solaris，Linux
- 轻量级进程：在内核中实现，支持用户线程
    - Solaris (LightWeight Process)

# (二)处理机调度

## <1>调度的基本概念

### #1处理机调度

调度的实质是一种资源分配，处理机调度是对处理机资源进行分配。

处理机调度决定系统运行时的性能：系统吞入量、资源利用率、作业周转时间、作业响应时间等....

### #2处理机调度的层次

高级调度（又称长程调度或者作业调度） - 》作业级

低级调度（又称短程调度或者进程调度） - 》进程（线程）级

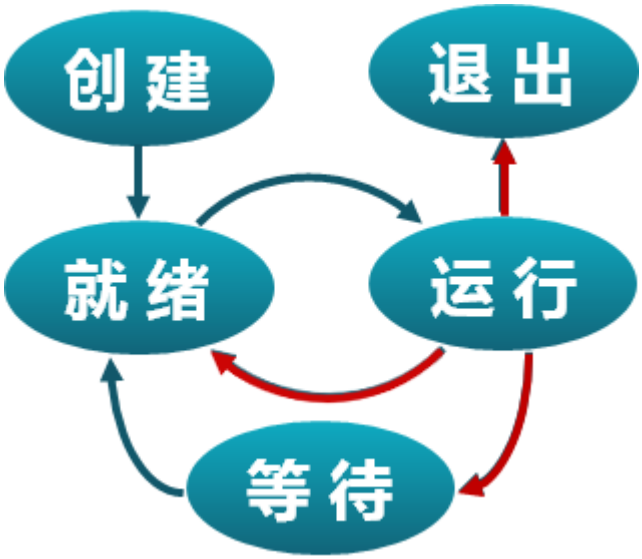
中级调度（内存调度） - 》内存

### #3处理机调度算法功能--CPU的时分复用

- 处理机调度算法
    - 从**就绪队列中挑选**下一个占用CPU运行的进程
    - 从多个**可用CPU中挑选**就绪进程可使用的CPU资源
- 调度程序：挑选就绪进程的内核函数
    - 调度策略：依据什么原则挑选进程/线程?
    - 调度时机：什么时候进行调度?

## <2>调度时机、切换与过程

在进程/线程的生命周期中的什么时候进行调度?




### #1内核运行调度程序的条件

- 进程从运行状态切换到等待状态
- 进程被终结(退出)了

### #2非抢占系统

- 当前进程主动放弃CPU时

### #3可抢占系统

- 中断请求被服务例程响应完成时
- 当前进程被抢占
  - 进程时间片用完
  - (高优先级)进程从等待切换到就绪

## <3>调度的基本准则

### #1处理机调度算法的共同目标

- 资源利用率：CPU处于忙状态的时间百分比（包括I/O）
  - $CPU_{利用率} = \frac{CPU_{有效工作时间}}{CPU_{有效工作时间}+CPU_{空闲等待时间}}$
- 公平性：各个进程都能合理的使用CPU，不会发送进程饥饿状态
- 平衡性：使各个资源经常处于繁忙状态
- 策略强制执行

### #2批处理系统的目标

- 平均周转时间T短(周转时间是指作业到达时间开始一直到作业完成的时间)
  - $T = \frac{1}{n}\left[\sum_{i=0}^{n} T_i\right]$
- 带权的平均周转时间
- $T = \frac{1}{n}\left[\sum_{i=0}^{n} \frac{T_i}{T_s}\right]$
- 其中$T_i$为作业周转时间，$T_s$为系统为其提供服务的时间

- 系统吞吐量高
- 处理机利用率高

### #3分时系统的目标

- 响应时间快
- 均衡性

### #4实时系统的目标

- 截止时间的保证
- 可预测性
- 哪些系统是实时系统?
    - 硬实时操作系统的代表：VxWorks
    - 软实时操作系统的代表：各种实时Linux

## <4>调度方式

## <5>典型调度算法

### #1先来先服务调度算法(FCFS)

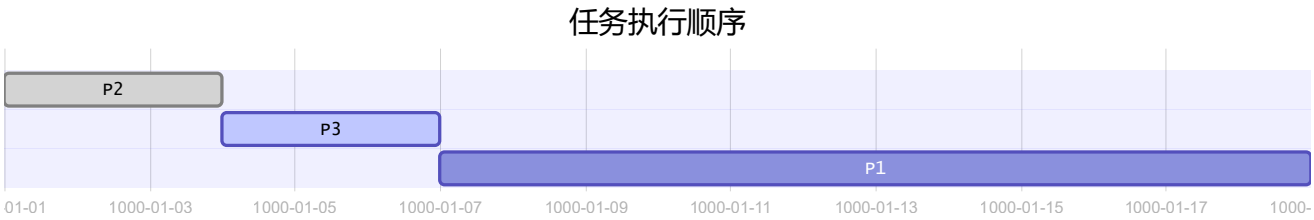
**1.依据进程进入就绪状态的先后顺序排列**

**2.周转时间(从到达时间~作业结束时间)**

比如3个进程，计算时间为12，3，3，到达顺序为P1,P2,P3（假设同一时刻到达）


则周转时间=(12+15+18)/3=15

如果到达顺序为P2,P3,P1




则测试周转时间=(3+6+18)/3=9

**3.优点**

简单

**4.缺点**

平均等待时间波动比较大，比如短进程可能排在长进程后面

## #2短进程（短作业、短线程）优先调度算法(SPN,SJF)

**1.概念**

选择就绪队列中**执行时间最短的进程占用CPU**进入运行状态

**2.排序**

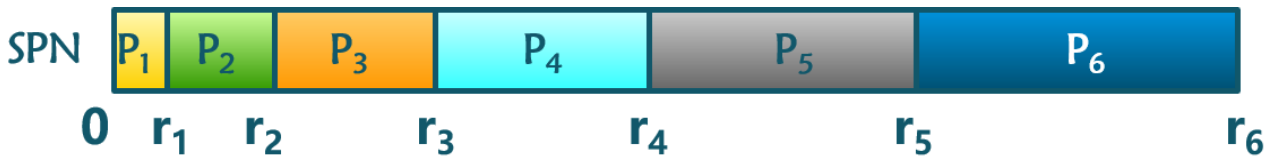
就绪队列**按照预期的执行时间长度**来排序

**3.SPN的可抢占改进--SRT(短剩余时间优先算法)**

新进程所需要的执行时间比当前正在执行的进程剩余的执行时间还要短，那么允许它抢先。

**4.SPN具有最优平均周转时间**

SPN算法中一组进程的平均周转时间




此时周转时间$=(r_1+r_2+r_3+r_4+r_5+r_6)/6$

修改进程执行顺序可能减少平均等待时间吗?


周转时间$=(r_1+r_2+r_4-c_3+r_5-c_3+r_4+c_4+c_5+r_6)/6$

$=(r_1+r_2+r_3+r_4+r_5+r_6+(c_4+c_5-2c_3))/6$

$c_i$表示进程$P_i$的执行时间(不一定等于周转时间)

**5.缺点**

- **可能导致饥饿**
  - 连续的短进程流会使长进程无法获得CPU资源(在就绪队列中)
- **需要预知未来**
  - 如何预估下一个CPU计算的持续时间?
  - 简单的解决办法：询问用户
    - 用户欺骗就杀死相应进程
    - 用户不知道怎么办？（用历史的执行时间来预估未来的执行时间）
- 人机无法交互

- 未考虑作业的紧迫程度，不能保证紧迫性作业能得到及时处理

## #3最高响应比优先算法(HRRN)

**1.排序选择**

**选择就绪队列中响应比R值最高的进程**

**即按照R值来排序**

R = （w+s)/s

w: 等待时间(waiting time)

s: 执行时间(service time)

**2.优点**

- 在短进程优先算法的基础上改进
- 不可抢占
- 关注进程的等待时间
- 防止无限期推迟(w越大，优先级越高)

## #4时间片轮转调度算法(依靠时钟中断)

**1.思想**

时间片结束时，按照FCFS算法切换到下一个就绪进程

每隔(n-1)个时间片进程，进程执行一个时间片q

**2.时间片为20的RR算法示例**

- **示例: 4个进程的执行时间如下**

| | |
|---|---|
| P1 | 53 |
| P2 | 8 |
| P3 | 68 |
| P4 | 24 |

甘特图如下:


等待时间 $P_1=(68-20)+(112-88)=72$

$P_2=(20-0)=20$

$P_3=(28-0)+(88-48)+(125-108)=85$

$P_4=(48-0)+(108-68)=88$

平均等待时间 = (72+20+85+88)/4=66.25

**3.进程切换时机**

进程在一个时间片内**已执行完**，立刻调度，并将该进程从就绪队列中删除

进程在一个时间片内**未执行完**，立刻调度，并将该进程放入就绪队列末尾

**4.RR的时间片长度**

- RR开销主要在于额外的上下文切换
- 如果时间片太大
  - 那么等待时间过长
  - 极限情况下RR就退化为FCFS
- 如果时间片太小
  - 反应迅速，会产生大量的上下文切换
  - 从而增加了系统的开销，影响系统吞吐量
- 时间片长度选择目标
  - 选择合适的时间片长度
  - 经验规则:维持上下文开销处于1%

## #4.1比较FCFS和RR例子

**0.等待时间=周转时间−执行时间**

■ 示例: 4个进程的执行时间如下

| | |
|---|---|
| P1 | 53 |
| P2 | 8 |
| P3 | 68 |
| P4 | 24 |

假设上下文切换时间为零

FCFS和RR各自的平均等待时间是多少?

| 时间片 | $P_1$ | $P_2$ | $P_3$ | $P_4$ | 平均等待时间 |
|---|---|---|---|---|---|
| RR(q=1) | 84 | 22 | 85 | 57 | 62 |
| RR(q=5) | 82 | 20 | 85 | 58 | 61.25 |
| RR(q=8) | 80 | 8 | 85 | 56 | 57.25 |
| RR(q=10) | 82 | 10 | 85 | 68 | 61.25 |
| RR(q=20) | 72 | 20 | 85 | 88 | 66.25 |
| BestFCFS | 32 | 0 | 85 | 8 | 31.25 |
| WorstFCFS | 68 | 145 | 0 | 121 | 83.5 |

**1.最好FCFS相当于短进程优先**

**2.最坏FCFS相等于长进程优先**

## #5多级队列调度算法(MQ)

**1.就绪队列被划分成多个独立的子队列**

如：前台(交互)、后台(批处理)

**2.每个队列拥有自己的调度策略(进程不能在队列间移动)**

如：前台–RR、后台–FCFS

**3.队列间的调度**

- 如果固定优先级
  - 先处理前台，然后处理后台
  - 可能导致饥饿
- 如果时间片轮转
  - 每个队列都得到一个确定的能够调度其进程的CPU总时间
  - 如：80%CPU时间用于前台，20%CPU时间用于后台

## #6多级反馈队列调度算法(MLFQ)

**1.调度机制**

**设置多个就绪队列，为每个队列赋予不同的优先级，每个队列采用FCFS算法，按队列优先级调度**

- 进程可在不同队列间移动的多级队列算法
  - 队列时间片大小随优先级级别增加而增加(倍增)，即时间片越小队列优先级越高
  - 如进程在当前的时间片没有完成，则降到下一个优先级队列

**2.MLFQ算法的特征**

- CPU密集型进程的优先级下降很快，并且时间片会分得很大
- I/O密集型进程停留在高优先级

# (三)同步与互斥

## <1>进程同步的基本概念

### #1进程同步

**对多个进程在执行顺序**上进行**调节**，使**并发执行**的诸程序之间能按照一定的规则（时序）**共享系统资源**，并能够很好的相互合作，从而使**程序**的执行具有**可再现性。**

### #2两种形式的制约

间接相互制约关系（互斥）：由于共享系统资源导致

直接相互制约关系（同步）：为完成同一任务而合作

### #3.临界资源（Critical resource, 进程需要互斥访问的资源）

比如打印机，磁带机，producer-consumer问题

### #4.临界区（Critical Section，进程中访问临界资源的代码）

```
enter section       //进入区
    critical section //临界区
exit section        //退出区
    remainder seciton//剩余区
```

- 临界区
  - 进程中**访问临界资源**的一段需要**互斥执行**的**代码**
- 进入区(entry section)
  - 检查可否进入临界区的一段代码
  - 如可进入，**设置相应"正在访问临界区"标志**
- 退出区(exit section)
  - 清除“正在访问临界区”标志
- 剩余区(remainder section)
  - 代码中的其余部分,与同步互斥无关的代码

### #5.同步机制规则(临界区的访问规则)

空闲让进：无进程时，任何进程可进去

忙则等待：有进程在临界区时，其他进程均不能进入临界区

有限等待：有点等待时间，不能无限等待

让权等待（可选）：不能进入临界区的进程，应释放CPU

## #6.实现临界区互斥的基本方法

软件实现方法，硬件实现方法。

**方法一:禁用硬件中断**

- 没有中断，没有上下文切换，因此没有并发
  - 硬件将中断处理延迟到中断被启用之后
  - 现代计算机体系结构都提供指令来实现禁用中断

```
local_irq_save(unsigned long flags); //关中断
critical section                     //临界区
local_irq_restore(unsigned long flags); //使能中断
```

- 进入临界区
  - 禁止所有中断，并保存标志
- 离开临界区
  - 使能所有中断，并恢复标志
- 缺点
  - 关中断后，进程无法被停止，可能导致其他进程饥饿或者系统停止
  - 临界区可能很长，因为无法确定响应中断所需要的时间，并且可能存在硬件影响
  - 不适用于多CPU系统，因为一个处理器上关中断不影响其他处理器执行临界区代码

**方法二:软件方法-Peterson算法**

满足线程$T_i$和$T_j$之间互斥的经典的基于软件的方法

- 共享变量

```
int turn;   //表示允许进入临界区的线程ID
boolean flag[]; //表示进程请求进入临界区
```

- 进入区代码

```
flag[i]=true;
turn=j;
while(flag[j] && turn == j);    //有一个条件不满足就进入临界区，否则一直等
/*
*此时如果同时有两个进程进入临界区
*那么先写的那个进程能进入(后一个不满足)，后的不能(都满足)
*/
```

- 退出区代码

```
flag[i]=false;
```

线程$T_i$的代码

```
do{
    flag[i]=true; //线程i请求进入临界区
    turn=j;
    while(flag[j] && turn == j);
        CRITICAL SECTION    //临界区
    flag[i]=false;
        REMAINDER SECTION   //退出区
}while(true);
```

**方法二:软件方法-Dekkers算法(两个进程)**

```
flag[0]=false;
flag[1]=false;
turn=0;
do{
    flag[i]=true; //线程i请求进入临界区
    while(flag[j]==true){
        if(turn!=i){
            flag[i]=false;
            while(turn!=i){}
            flag=true;
        }
    }
    CRITICAL SECTION //临界区
    turn=j;
    falsg[i]=false;
    REMAINDER SECTION //退出区
}while(true);
```

**方法三:更高级的抽象方法**

**1.概念**

基于硬件提供了一些**同步原语**，比如中断禁用，原子操作指令等

操作系统提供更高级的编程抽象来简化进程同步，例如：锁、信号量，用硬件原语来构建

**2.例如使用TS指令实现自旋锁(spinlock)**

```
class Lock{
    int value=0;
}
//忙等待锁
Lock::Acquire(){
    while(test-and-set(value))
        ;//spin
}

Lock::Release(){
    value=0;
}
```

#### 3.原子操作指令锁的特征

- 优点
  - 适用于单处理器或者共享主存的多处理器中任意数量的进程同步
  - 简单并且容易证明
  - 支持多临界区
- 缺点
  - 忙等待消耗处理器时间
- 可能导致饥饿
  - 进程离开临界区时有多个等待进程的情况
- 死锁
  - 有一个拥有临界区的低优先级进程
  - 同时有一个请求访问临界区的高优先级进程获得处理器并等待临界区

### #7基于软件的解决方法的分析

**复杂,需要两个进程间的共享数据项**

**需要忙等待,浪费CPU时间**

# <2>信号量(semaphore)

## #1.概念

信号量是操作系统提供的一种协调共享资源访问的方法

#### 1.信号量是一种抽象数据类型

由一个整形变量**sem**（共享资源的数目）和两个原子操作组成

```
//P操作--申请使用资源
P()(Prolaag （荷兰语尝试减少）) -》wait
sem--;//可用资源减少一
if sem<0,进入等待，否则继续  //可用资源用完了，需要等待其他线程释放资源

//V操作--释放可用资源
V()(Verhoog （荷兰语增加）) -》signal
sem++;
if sem<=0,唤醒一个等待进程
```

**2.信号量的特性**

- 信号量(sem)是**被保护的整数**变量
  - **初始化完成后，只能通过P()和V()操作修改**
- 由**操作系统保证，PV操作是原子操作(无法被打断)。**
- **P() 可能阻塞**（由于没有资源进入等待状态），**V()不会阻塞**(V操作只会释放资源)
- 通常假定信号量是“公平的”
  - **线程不会被无限期阻塞在P()操作**
  - 假定**信号量**等待按先进先出排队(**等待队列按照FCFS排列**)
- 信号量不能避免死锁问题

**3.自旋锁能否实现先进先出?**

不能。因为自旋锁需要占用CPU，随时检查，有可能临界区的使用者退出时刚修改完，下一个进入者进入时资源才变成有效，就无法实现先进先出。

## #2信号量的实现

```
classSemaphore{
    int sem;    //共享资源数目
    WaitQueue q;  //等待队列
}

Semaphore::P(){
    sem--;
    if(sem<0){
        //资源用完了
        Add this thread t to q;
        block(p);  //阻塞
    }
}

Semaphore::V() {
    sem++;
    if (sem<=0) {
        //此时前面仍有等待线程
        //从对应的等待队列里把相应的线程放入就绪队列
        Remove a thread t from q;
        wakeup(t);
    }
}
```

```
24
```

## #3信号量的使用

### 1.信号量分类

- 可分为两种信号量
  - 二进制信号量（AND型）：资源数目为**0或1**
  - 资源信号量(记录型):资源数目为**任何非负值**
  - 两者等价
    - 基于一个可以实现另一个

### 2.信号量的使用

- 互斥访问
  - 临界区的互斥访问控制
- 条件同步
  - 线程间的事件等待

### 3.用信号量实现临界区的互斥访问

**每个临界区设置一个信号量，其初值为1**

```
mutex = new Semaphore(1); //信号量初始化为1

//控制临界区的访问
mutex->P();     //信号量计数--
Critical Section;
mutex->V();     //释放资源，信号量计数++
```

**注意:**

**初始化如果是同步互斥，看资源数目，如果是条件同步，为0或者1**

**必须成对使用P()操作和V()操作**

**P()操作保证互斥访问临界资源**

**PV操作不能次序错误、重复或遗漏**(但不要求P在V之前或者之后)

执行时不可中断

**问题:**

**不申请直接释放**，出现多个线程进入临界区的情况

**只申请不释放**，缓冲区没有线程，但是谁也进不去临界区

### 4.用信号量实现条件同步

```
//此时的条件同步设置一个信号量，初始化为0
condition =new Semaphore(0);

```

```
//实现一个条件等待，线程A要等到线程B执行完X模块后才能执行N模块


//线程A
---M---
    condition->P();
---N---

//线程B
---x---
    condition->V();
---Y---


    //在B里释放信号量，使其0->1,如果B先执行完X模块，则A可以直接往下执行;
    //如果A先执行完就等待
```

## #4生产者-消费者问题(信号量)


- 有界缓冲区的生产者-消费者问题描述
  - 一个或多个**生产者**在生成数据后**放**在一个缓冲区里
  - 一个或多个**消费者**从缓冲区**取**出数据处理
  - 任何时刻只能有**一个**生产者或消费者**可访问缓冲区**
- 问题分析
  - 任何时刻只能有一个线程操作缓冲区（**互斥访问**）
  - 缓冲区空时，消费者必须等待生产者（**条件同步**）
  - 缓冲区满时，生产者必须等待消费者（**条件同步**）
- 用信号量描述每个约束
  - 二进制信号量mutex
  - 资源信号量fullBuffers--等待有数据
  - 资源信号量emptyBuffers--缓冲区有空闲
  - **两个资源相加=缓冲区总大小**
- 实现

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);//一开始缓冲区中没有数据
    emptyBuffers = new Semaphore(n);//全都是空缓冲区
}
```

```
BoundedBuffer::Deposit(c) {//生产者
    emptyBuffers->P(); //检查是否有空缓冲区
    mutex->P(); //申请缓冲区
    Add c to the buffer;
    mutex->V();
    fullBuffers->V();//释放该资源，相等于缓冲区中多了一个数据
}
```

```
BoundedBuffer::Remove(c) {//消费者
    fullBuffers->P();//检查缓冲区中是否有数据
    mutex->P();
    Remove c from buffer;
    mutex->V();
    emptyBuffers->V();//释放空缓冲区资源
}
```

**两次P操作的顺序有影响吗?**

**交换顺序会出现死锁，原因在于自我检查空和满**

# <3>管程

## #1概念

**管程是一种用于多线程互斥访问共享资源的程序结构**

- 采用面向对象方法，简化了线程间的同步控制
- 任一时刻最多只有一个线程执行管程代码
- 正在管程中的线程可临时放弃管程的互斥访问，等待事件出现时恢复

**管程的使用**

- 在对象/模块中，收集相关共享数据
- 定义访问共享数据的方法

## #2管程的组成

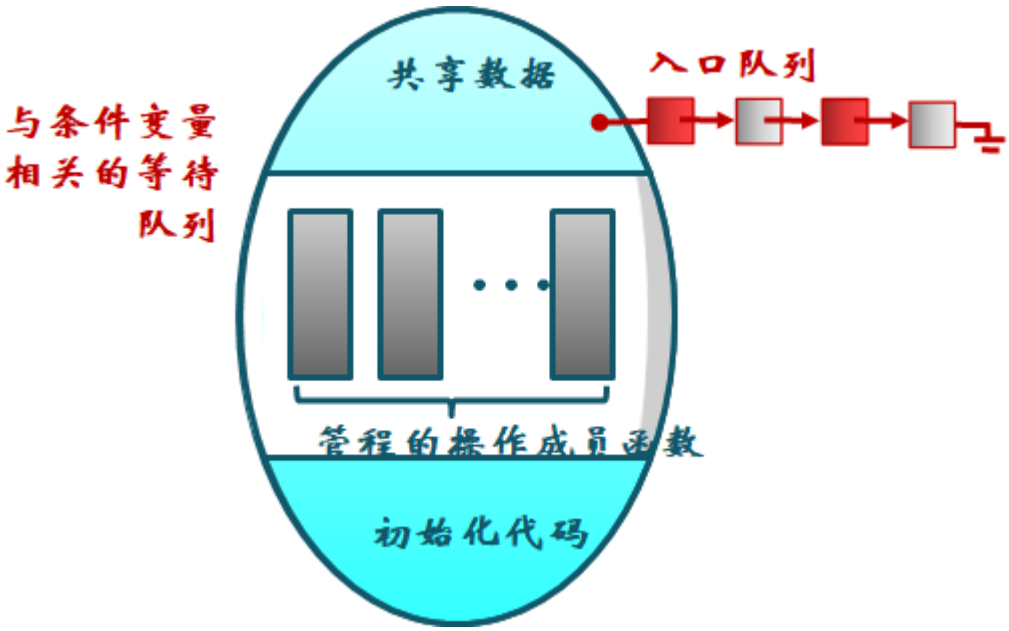
**1.(在入口队列加)一个锁**

控制管程代码的互斥访问

**2.0或者多个条件变量**

管理共享数据的并发访问

如果是0个，就等同与一个临界区，如果是多个就是管程所特有的




**3.条件变量**

- 条件变量是管程内的等待机制
  - **进入管程的线程因资源被占用而进入等待状态**
  - **每个条件变量表示一种等待原因，对应一个等待队列**
- **Wait()操作--等待操作**
  - **将自己阻塞在等待队列中**
  - **同时唤醒一个等待者或释放管程的互斥访问(即允许另外一个线程进入管程)**
- **Signal()操作--释放操作**
  - 将等待队列中的一个线程**唤醒**
  - 如果**等待队列为空**，则等**同空操作**

**4.条件变量的实现**

```
Class Condition{
    int numWaiting=0;
    //条件变量初值为0，如果在信号量里和资源数目一致
    WaitQueue q;
}
```

```
Condition::Wait(lock){
    numWaiting++;  //等待数目++
    Add this thread t to q;//将自己放入等待队列当中
    release(lock);  //释放管程的互斥访问权限
    shedule();//执行调度，切换线程need mutex
    require(lock);//请求访问权限
}
```

```
Condition::Signal(){
    if(numWaiting>0){//等待队列不为空,即有另外的线程等待这个条件变量上，每个变量对应一个队列
        Remove a thread t from q;//将此线程从等待队列移动到就绪队列中
        wakeup(t);//唤醒进程need mutex
        numWaiting--;//等待数目--
    }
}
```

### #4生产者-消费者问题

```
classBoundedBuffer {
    …
    Lock lock;//一个入口等待队列
    int count = 0; //写入缓冲区的数据的数目
    Condition notFull, notEmpty;//两个条件变量
}
```

```
BoundedBuffer::Deposit(c) {//生产者
    lock->Acquire();    //管程进入权申请
    while (count == n)  //n个缓冲区中都有数据了--对应**1
        notFull.Wait(&lock);//就放弃管程使用权，等在notfull条件变量上
    Add c to the buffer;
    count++;
    notEmpty.Signal();
    lock->Release();    //管程进入权释放
}
```

```
BoundedBuffer::Remove(c) {//消费者
    lock->Acquire();    //管程进入权申请
    while (count == 0)    //如果没数据
      notEmpty.Wait(&lock);//就放弃管程使用权，等在非空条件上
    Remove c from buffer;
    count--;
    notFull.Signal();   //读出一个数据后就释放notfull--对应**1
    lock->Release();    //管程进入权释放
}
```

**做法:**

1.初始化

2.管程中写法

- 先写管程进去权申请和释放lock

## <4>同步问题

### #1哲学家就餐问题

**1.问题描述:**

5个哲学家围绕一张圆桌而坐，桌子上放着5支叉子，每两个哲学家之间放一支

哲学家的动作包括思考和进餐，进餐时需同时拿到左右两边的叉子，思考时将两支叉子放回原处

如何保证哲学家们的动作有序进行？如：不出现有人永远拿不到叉子

**2.信号量解决**

```
#define   N   5                       // 哲学家个数
semaphore fork[5];                    // 信号量初值为1
void   philosopher(int   i)           // 哲学家编号：0 - 4
    while(TRUE)
    {
        think( );                     // 哲学家在思考
        if (i%2 == 0) {
            P(fork[i]);         // 去拿左边的叉子
            P(fork[(i + 1) % N]);    // 去拿右边的叉子
        } else {
            P(fork[(i + 1) % N]);    // 去拿右边的叉子
            P(fork[i]);              // 去拿左边的叉子
        }
        eat( );                       // 吃面条中….
        V(fork[i]);             // 放下左边的叉子
        V(fork[(i + 1) % N]);          // 放下右边的叉子
    }
//没有死锁，可有多人同时就餐
```

## #2读者 - 写者问题（信号量）

**1.问题描述**

- 共享数据的两类使用者
  - 读者：只读取数据，不修改
  - 写者：读取和修改数据
- 读者-写者问题描述：对共享数据的读写
  - “读 - 读”允许
    - 同一时刻，允许有多个读者同时读
  - “读 - 写”互斥
    - 没有写者时读者才能读
    - 没有读者时写者才能写
  - “写 - 写”互斥
    - 没有其他写者时写者才能写

**2.解决**

用信号量描述每个约束

- 信号量WriteMutex
  - 控制**读写操作的互斥**
  - 初始化为1
- 读者计数Rcount
  - 正在进行**读操作的读者数目**

    - 初始化为0
- 信号量CountMutex
    - 控制对**写者计数的互斥修改**
    - 初始化为1,同一时间只有一个可以写

```
semphoare WriteMutex=1;
int Rcount=0;
semphoare CountMutex=1;
```

```
void Writer(){
    P(WriteMutex);
    write;
    V(WriteMutex);
}
```

```
void Reader(){
    P(CountMutex);
    if (Rcount == 0)
        P(WriteMutex);
    ++Rcount;
    V(CountMutex);
    read;
    P(CountMutex);
    --Rcount;
    if (Rcount == 0)
    V(WriteMutex);
    V(CountMutex)
}
//此实现中，读者优先
```

**3.优先策略**

读者优先策略

只要有读者正在读状态，后来的读者都能直接进入

如读者持续不断进入，则写者就处于饥饿

写者优先策略

只要有写者就绪，写者应尽快执行写操作

如写者持续不断就绪，则读者就处于饥饿

如何实现?

## #3读者 - 写者问题（写者优先）

用信号量描述每个约束

- 信号量WriteMutex
    - 控制**读写操作的互斥**
    - 初始化为1

- 读者计数Rcount
  - 正在进行**读操作的读者数目**
  - 初始化为0
- 信号量CountMutex
  - 控制对**计数变量Rcount的互斥修改**
  - 初始化为1,同一时间只有一个可以写
- 信号量ReadMutex与x
  - 控制读者进入
  - x防止大量读者被阻塞
- 写者计数WRcount
  - 正在进行写操作的写者数目
  - 初始化为0
- 信号量WRMutex
  - 控制计数变量WRcount的互斥修改
  - 初始化为1，同一时间只有一个可以修改

```
semaphore WriteMutex=1,ReadMutex=1,x=1;
int Rcount=0,WRcount=0;
semaphore CountMutex=1,WRMutex=1;
```

```
void Reader(){
    P(x);
    P(ReadMutex);
    P(CountMutex);
    if(Rcount==0)
        P(WriteMutex);
    ++Rcount;
    V(CountMutex);
    V(ReadMutex);
    V(x);
    read;
    P(CountMutex);
    Rcount--;
    if(Rcount==0)
       V(WriteMutex);
    V(CountMutex);
}
```

```
void Writer(){
    P(WRMutex);
    if(WRcount==0)
        P(ReadMutex);
    WRcount++;
    V(WRMutex);
    P(WriteMutex);
    Write;
    V(WriteMutex);
    P(WRMutex);
```

```
    WRcount--;
    if(WRcount==0)
        V(ReadMutex);
    V(WRMutex);
}
```

# <5>死锁

## #0.资源分类

### 1.可重用资源（Reusable Resource）

- 资源不能被删除且在任何时刻只能有一个进程使用
- 进程释放资源后，其他进程可重用
- 可重用资源示例
  - 硬件:处理器，I/O通道，存储器，设备等
  - 软件:文件、数据库和信号量等数据结构
- 可能出现死锁

### 2.消耗资源(Consumable resource)

- 临时性资源，在进程运行期间，由进程动态的创建和销毁
- 资源创建和销毁
- 消耗资源示例
  - 在I/O缓冲区的中断、信号、消息等

### 3.可抢占的资源

- **某进程获得资源后，该资源可以再被其他进程抢占。**
- 例如：处理机、内存
- **不会引起死锁**

### 4.不可抢占的资源

- **一旦系统把资源分配给某一进程，必须等待进程自行释放该资源。**
- 例如：打印机、刻录机
- **可能引起死锁**

### 5.前驱图(资源分配图)

顶点

- 进程
- 资源

有向边

- 资源请求，P操作请求某资源
- 资源分配，便是某资源已经分配给某资源

## #1.死锁的概念

如果一组进程中每一个进程都在等待仅由该组进程中的其它进程才能引发的事件，那该组进程是死锁的。

## #2死锁产生的原因

- **竞争不可抢占的资源**
- **竞争可消费的资源**
- **进程推进顺序不当引起的死锁**

## #3产生死锁的必备条件(同时满足)

- 互斥
  - 任何时候只能有一个进程使用一个(非共享)资源实例
- 持有并等待
  - 一个进程至少一个资源，并且正在等待获取其他进程的资源
- 非抢占
  - 资源只能在进程使用后资源释放，不可以强行剥夺
- 循环等待
  - 存在等待进程集合{p0~pn}
  - p0等p1
  - ~~
  - pn-1等pn
  - pn等p0的资源

## #4.死锁处理策略

**1.预防死锁**

**确保系统永远不会进去死锁状态**

预防是采用某种策略，限制并发进程对资源的请求，使得系统在任何时刻都不满足死锁的必要条件（四个）

**2.避免死锁**

在使用前进行判断，只允许不会出现死锁的进程请求资源

**3.死锁检测和恢复(Deadlock Detection & Recovery)**

在检测到运行系统进入死锁状态后，进行恢复

**4.由应用进程处理死锁**

通常操作系统忽略死锁,大多数操作系统（包括UNIX）的做法

## #5.死锁预防具体做法

- 互斥
  - 把互斥的共享资源封装为可以同时访问的
- 持有并等待
  - 要求进程在请求资源时，不占优其他任何资源
  - 或者仅仅允许进程在开始执行时，一次请求所有需要的资源
  - 这样资源的利用率低
- 非抢占

    - 如果进程请求不能立即分配，就释放占有的资源
    - 或者，只有在能够同时获得所有资源的时候，才执行分配操作
- 循环等待
    - 对资源排序，要求进程按顺序请求资源

## #6.死锁避免

**0.安全状态**

在死锁避免方法中，把系统的状态分为安全状态和不安全状态。

当系统处于安全状态时，可以避免发生死锁。反之，可能发生死锁。

**1.安全状态概念**

- **当进程请求资源时，系统判断分配后是否处于安全状态**
- 系统处于安全状态
    - 针对**所有已占用进程，存在安全序列**
- **序列**$< P_1 , P_2 , \dots , P_N >$**是安全**的
    - $P_i$**要求的资源**$\leq$**当前可用资源+所有**$P_j$ **持有资源,其中**$j < i$
    - 如$P_i$的资源请求不能立即分配，则$P_i$等待所有$P_j$（$j < i$）完成
    - $P_i$完成后，$P_i + 1$可得到所需资源，执行并释放所分配的资源
    - 最终整个序列的所有$P_i$都能获得所需资源

**2.安全状态和死锁的关系**

系统处于安全状态，一定没有死锁

系统处于不安全状态，可能出现死锁，**避免死锁就是确保系统不会进入不安全状态**

## #7银行家算法（Banker's Algorithm）

**0.概念**

- 银行家算法是一个**避免死锁**产生的算法。以银行借贷分配策略为基础，判断并**保证系统处于安全状态**
- 客户在**第一次**申请贷款时，**声明**所需**最大资金量**，在满足所有贷款要求并完成项目时，**及时归还**
- 在客户**贷款**数量**不超过**银行**拥有的最大值**时，银行家**尽量满足**客户需要
- 类比
    - 银行家 $\leftrightarrow$ 操作系统
    - 资金 $\leftrightarrow$ 资源
    - 客户 $\leftrightarrow$ 申请资源的线程

**1.数据结构**

$n$ = **线程数量**, $m$ = **资源类型数量**

$Max$（**总需求量**）:$n \times m$矩阵

线程$T_i$**最多请求**类型$R_j$的资源$Max[i,j]$个实例

$Available$（**剩余空闲量**）：长度为$m$的向量

**当前有** $Available[j]$个类型为$R_j$的资源实例**可用**

$Allocation$（**已分配量**）：$n \times m$矩阵

线程$T_i$**当前分配**了$Allocation[i,j]$个$R_j$的实例

$Need$（**未来需要量**）：$n \times m$矩阵

线程$T_i$**未来需要** $Need[i,j]$个$R_j$资源实例

满足公式:

$Need[i,j] = Max[i,j] - Allocation[i,j]$

**2.银行家算法**

**初始化:**

$Request_i$ 线程$T_i$的资源请求向量

$Request_i[j]$ 线程$T_i$请求资源$R_j$的实例

**循环:**

1.如果 $Request_i \leq Need[i]$, 转到步骤2。否则, 拒绝资源申请, 因为线程已经超过了其最大要求

2.如果$Request_i \leq Available$, 转到步骤3。否则,$T_i$必须**等待**,因为资源不可用

3.通过安全状态判断来确定是否分配资源给$T_i$：

**生成一个需要判断状态是否安全的资源分配环境**

$Available = Available - Request_i$;//剩余-请求

$Allocation[i] = Allocation[i] + Request_i$;//已分配+请求

$Need[i] = Need[i] - Request_i$;//未来需要-请求

**调用安全状态判断:**

**如果返回结果是安全，将资源分配给$T_i$**

**如果返回结果是不安全，系统会拒绝$T_i$的资源请求**

**3.安全性算法--安全状态判断**

```
1.Work 和Finish 分别是长度为m和n的向量初始化:
   Work = Available //当前资源剩余空闲量
   Finish[i] = false for i: 1,2, …, n. //线程i没结束

2.寻找线程Ti:
   (a) Finish[i] = false//接下来找出Need比Work小的线程i
   (b) Need[i]≤Work
    若找到，执行3;
    没有找到满足条件的Ti，转4。

3.Work = Work + Allocation[i]//线程i的资源需求量小于当前剩余空闲资源量，所以配置给它的资源再回收
Finish[i] = true 转2

4.如所有线程Ti满足Finish[i] == true, //所有线程的Finish为True,表明系统处于安全状态
```

```
15      则系统处于安全状态
```

**4.死锁的检测**

允许系统进入死锁状态

维护系统的资源分配图

定期调用死锁检测算法来搜索图中是否存在死锁

出现死锁时，用死锁恢复机制进行恢复

# 三.存储器管理

## (一)知识点

### #1.用户源程序变为一个可在内存中执行的程序需经过哪些步骤？（P131-132）

- 编译(由编译程序对源程序进行编译，形成目标模块),
- 连接(由链接程序将编译后的模块和库函数链接),
- 装入(由装入程序装入内存

### #2.程序装入的方式 （P132-133）

- 绝对装入:给出绝对地址，将装入模块直接装入指定内存即可，其逻辑地址和实际内存地址完全相同，无需进行地址变换。只适用于单道程序环境。
- 可重定位装入方式:在装入时逻辑地址和实际地址不同，需要对逻辑地址进行改变，其地址变换在装入时一次性完成，之后不在变换。
- 动态装入:在装入模块到内存后，不立即把模块中的逻辑地址转换为物理地址，将转换推迟到程序运行时才进行，其装入地址都是逻辑地址，需要重定位寄存器的支持。

### #3.重定位、静态重定位、动态重定位（P132-133）

- 重定位：装入时对目标程序中指令和数据地址的修改过程
- 静态重定位：地址变换在装入时一次性完成，之后不在变换
- 动态重定位：将地址转换推迟到程序真正要运行时才进行

### #4.内存的连续分配方式有哪些？（P135-144）

- 单一连续分配:整个内存的用户空间由该程序独占
- 固定分区分配：将整个用户空间划分为多个固定大小(分区大小可等可不等)的区域，每个区域中只装入一道作业
- 动态分区分配：根据进程的实际需要，动态分配内存空间
- 动态可重定位分区分配：根据进程的实际需要，动态分配内存空间，并且一个程序必须被装入连续内存空间中

### #5.基于顺序搜索的动态分区分配算法有哪些，算法的主要思想是什么？（P139-140）

- FF,首次适应算法，要求空闲分区链按照地址递增次序链接，从首地址开始查找，直到找到一个大小可满足的空闲分区
- NF,循环首次算法，从上次找到的空闲分区的下一个空闲分区开始查找，直到找到一个大小可满足的空闲分区

- BF，最佳适应算法，找到满足要求且是最小的空闲分区分配，为了加快查找，要求空闲分区按照容量递增排序
- WF，最坏适应算法，找到满足要求并且是最大的空闲分区分配，要求空闲分区按照容量递减顺序排序，查找时只看第一个分区是否满足

## #6.对换（P145-147）

**1.定义:**

将内存中暂时不能运行的程序调到磁盘的对换区，同时将磁盘上能运行的程序调入内存

**2.对换类型:**

整体对换(进程对换)，以进程为单位，需要系统支持功能**对对换空间的管理**，**进程的换入**和**进程的换出**

页面(分段)对换，以进程的一个页面或者分段为单位，有称部分对换，**其目的是支持虚拟存储系统**

**3.对换空间的管理**

文件区用于存放各类文件，对换区用于存放从进程换出的进程

- 目标
  - 对文件区的目标--提高文件存储空间的利用率，然后才是提高对文件的访问速度
  - 对对换空间管理的目标--提高进程换入换出的速度，然后才是文件存储空间的利用率
- 数据结构
  - 空闲分区链，每个表目中包含对换区的首地址和大小，分别用盘块号和盘块数目表示

## #7基本分页管理原理、地址变换过程（P147-155）

**1.原理**

- 将用户程序的地址空间分为若干个固定大小的区域，称为“页面”或者“页”
- 页内碎片：进程的页面没有装满，形成的不可用碎片
- 页面大小：通常为1KB~8KB
- 地址结构：页面号P+位移量W(即为页内地址)，P可算地址空间的页面数目，W可算每页的大小
  - $P = INT[\frac{A}{L}]$,$d = [A]MODL$
  - 其中P为页面号，L为页面大小，d为页内地址，A为逻辑地址
- 页表：分页系统中允许将进程的各个页离散地存储在内存的任一物理块中，为每个进程建立一张页面映像表，简称页表，实现从页面号到物理块号的地址映射

**2.地址变换机构**

实现从逻辑地址到物理地址的变换，借助页表来完成

- 基本地址变化机构
  - 只设置一个页表寄存器(PTR,Page-Table Register),存放页表在内存中的起始地址和长度
  - 访问数据时，将有效地址(相对地址)分为页号和页内地址，用页号检索页表，检索之前页号和页表长度比较，防止地址越界。如果没有越界，则将页表起始地址转换为该表项在页表中的位置，得到物理块号，装入物理地址寄存器，同时，将页内地址送入物理地址寄存器的块内地址字段中。
  - 页表在内存中，CPU存取一个数据时候需要两次访存，第一次是访问内存中的页表，找到对应页的物理块号，将块号和页内偏移量拼接形成物理地址。第二次访问是从第一次所得地址中获得所需要的数据(或者向其中写入数据)

- 具有快表的地址变换机构
    - 提高访问速度，设置联想寄存器(Associative Memory),也就是快表，用于存放当前访问的页表项。这样可以直接从快表中读出该页所对应的物理块号。如果快表已满，则OS必须找到不再需要的页表项将其换出。

**3.访问内存的有效时间**

从进程发出指定逻辑地址的访问请求，经过地址变换，到内存中找到对应的实际物理单元取出数据的总时间。

## #8分段系统的基本原理、地址变换过程（P155-160）

**1.段存储管理方式的引入原因**

主要满足用户和程序员以下需求：

**[1]、方便编程**

用户把自己的作业按照逻辑管理划分为若干段，每个段都是从0开始编址，并有自己的名字和长度。因此，希望要访问的逻辑地址是由段名（段号）和段内偏移量（段内地址）决定的。

**[2]、信息共享**

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。分页系统中的页只是存放信息的物理单位（块），并无完整的意义,段却是信息的逻辑单位。为了实现段的共享，希望存储管理能与用户程序分段的组织方式相适应。

**[3]、信息保护**

**[4]、动态增长**

有些段，会随着程序的使用不断增长。而事先又无法确切地知道数据段会增长到多大。

**[5]、动态链接**

动态链接是指在作业运行前，并不把几个目标程序段链接起来。要运行时，先将主程序所对应的目标程序装入内存并启动运行，当运行过程中有需要调用某段时，才将该段调入内存并进行链接。可见动态链接也要求以段作为管理的单位。

**2.分段系统的基本原理**

段号+段内地址

段号可算一个作业最长有多少个段，段内地址可算每个段的最大长度

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，每个段定义了一组逻辑信息

**3.段表**

在系统中为**每个进程建立一段映射表**，简称“段表”。每个段在表中占有一个表项。其中记录了**该段在内存中的起始地址（基址）和段的长度。**段表可以存放在一组寄存器中，以提高访问速度，但更常见的是将段表放在内存中。

在配置了段表后，执行中的进程可通过查找段表找到每个段所对应的内存区。

段表是用于实现从逻辑段到物理内存区的映射。

**4.地址变换机构**

为了实现进程逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表起始地址和段表长度TL。在进行地址变换时，系统将逻辑地址中的段号S与段表长度TL进行比较。若S>TL，表示段号太大。访问越界，于是产生越界中断信号；若未越界，则根据段表的起始地址和该段的段号+段内地址从而到的要访问的内存物理地址。

段表放在内存中时，每次访问一个数据都要两次方寸，解决方法和分页系统类似，设置一个联想寄存器，来保存最近常用的段表项。

### #9分页与分段的主要区别（P158）

a)、**页是信息的物理单位**，分页是为实现离散分配方式，其目的是消减内存的外零头，提高内存的利用率；分段中的**段则是信息的逻辑单位**，它含有一组其意义相对完整的信息，分段的目的是为了能更好地满足用户的需要。

b)、**页的大小固定且由系统决定**，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；**而段的长度却不固定**，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

c)、**分页的作业地址空间是一维的**，分页完全是系统行为，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；**而分段的作业地址空间则是二维的**，分段是用户行为，程序员在标识一个地址时，既要给出段名，又要给出段内地址。

### #10段页式存储管理的基本原理、地址变换过程（P160-162）

**1.基本原理**

先将用户程序分段，在段内进行分页，为每一个段赋予一个段名。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成。

**2.地址变换过程**

配置一个段表寄存器，其中存放段表起始地址和段表长TL。比较段号与TL是否越界，从段表寄存器中获取段表始址找到段表，根据段表内的页表始址找到对应的页表，在根据页表的存储块找到内存中的物理块，从而获取物理地址。

**3.访存次数**

段页式系统中，为了获得一条指令或数据，须三次访问内存：

① 访问内存中的段表，从中取得页表始址

② 访问内存中的页表，从中取出该页所在的物理块号，并与页内地址形成物理地址

③ 访问真正从第二次访问所得的地址中，取出指令或者数据

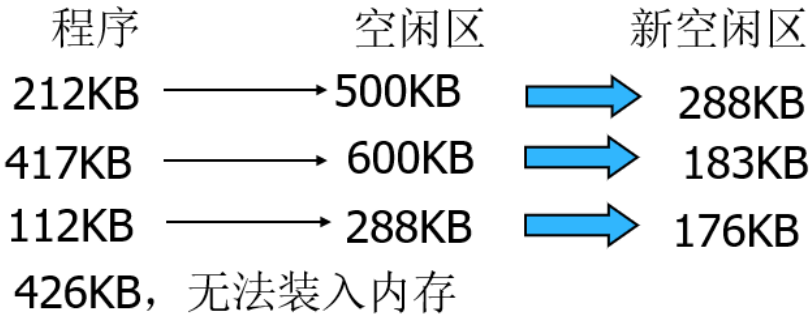
多次访问内存，执行速度降低，因此在地址变换机构中增设一个高速缓冲寄存器。每次访问它时，都须同时利用段号和页号去检索高速缓存，若找到匹配的表项，便可以从中得到相应页的物理块号，用来与页内地址一起形成物理地址；若未找到匹配表项，则仍需要再三次访问内存。

# (二)练习题

## <1>练习题 1

在可变分区存储管理下，按地址排列的内存空闲区为：100KB、500KB、200KB、300KB和600KB。现有若干用户程序，其所需内存依次分别为212KB、417KB、112KB和426KB，分别用首次适应算法、最佳适应算法、最坏适应算法，将它们装入到内存的哪些空闲分区？哪个算法能最有效利用内存？

解：采用首次适应算法




类似的分析可知，最坏适应算法也不能将426KB的程序装入内存，而最佳适应算法可将程序全部装入内存。

## <2>练习题 2

可变分区存储管理中，作业的撤离必定会修改内存的“空闲区表”，试画出因作业撤离修改“空闲区表”的四种情况。

根据回收区的首地址，在空闲分区表（链）找到插入点，此时可能出现4种情况之一（假设空闲分区表按地址从低到高顺序排列）：

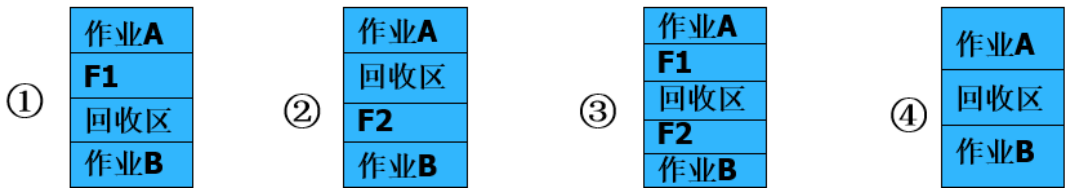
**解:**

①回收区与插入点的前一个空闲分区F1相邻接：将回收区与前一区合并，不必增加新表项，只需修改F1的大小为两者之和。

**②回收区与高地址F2分区邻接：此时将回收分区与该分区合并，回收区的首地址为新分区的首地址，大小为两者之和。**

**③回收区与前后分区F1和F2都邻接：将此3个分区合并，F1（前邻接区）的首地址为新分区的首址，大小为三者之和，取消F2表项。**

④回收区与任何空闲区都不邻接：在插入点建立一个新表项，填写回收区的首地址和大小。插入到空闲区表的适当位置（后移插入点后的各个表项）




## <3>练习题 3

在采用页式存储管理的系统中，某作业的逻辑地址空间为4页（每页4096字节），且已知该作业的页表如下表。试求出逻辑地址14688所对应的物理地址。

| 页 号 | 内存块号 |
| --- | --- |
| 0 | 2 |
| 1 | 4 |
| 2 | 7 |
| 3 | 9 |

页号P=INT(14688/4096)
=3
页内偏移d=14688%4096
=2400
物理地址=9×4096+2400
=39264

# 四.虚拟存储器管理

## (一)知识点

### <1>局部性原理（P165）

#### #1概念

程序在执行时出现的局部性规律，即在一较短时间内，程序的执行仅仅局限于某个部分，相应地，它所访问的存储空间也局限与某个区域。

#### #2表现

空间局限性:程序在一段时间内所访问地址可能集中在一定的范围内，其典型情况是程序的顺序执行

时间局限性：程序中的某条指令(数据)被执行(访问)，不久后该指令可能再次被执行(访问)。产生的典型原因是在程序中存在大量的循环操作。

### <2>虚拟存储器的定义与特征（P166）

#### #1定义

虚拟存储器，是指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。

#### #2特征

**1.多次性**

程序被允许分成多次装入内存

**2.对换性**

允许将暂不使用的代码和数据从内存中调至外存的对换区

**3.虚拟性**

能从逻辑上扩充内存容量，使得用户看到的内存容量远大于实际内存容量

# <3>请求分页存储管理方式的原理与硬件（P168-174）

## #1原理

在分页系统的基础上增加了请求调页功能和页面置换功能所形成的页式虚拟存储系统，允许用户程序只装入少数页面的程序和数据即可启动运行，通过上述功能将即将运行的页面调入内存，同时将暂不运行的页面换出到外存。

## #2硬件支持

### 1.请求分页的页表机制

在纯分页的页表机制上增加其他字段形成的数据结构，用于将逻辑地址转换为物理地址

### 2.缺页中断机构

每当用户程序需要的页面不再内存中，就产生缺页中断

### 3.地址变换机构

# <4>页面置换算法（Opt，FIFO，LRU）,缺页率计算（P174-181）

## #1-Optimal算法

选择的被淘汰页面是以后永远不使用的，或者是在最长(未来)时间内不再被访问的

## #2-FIFO

总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面进行淘汰

## #3-LRU

选择最近最久未使用的页面进行淘汰

### 0.硬件支持

LRU需要为每个内存页面配置一个移位寄存器，来记录进程在内存中的各个页面使用情况

LRU还需要用一个特殊的栈来保存当前使用的各个页面的页面号

# <5>请求分段存储管理方式的原理与硬件（P185-186）

## #1原理

请求分段系统中，程序在运行之前，只需要调入少数几个分段(不必调入所有的分段)就可以启动运行。当所访问的段不在内存中时，可请求OS将所缺的段调入内存。

## #2硬件支持

### 1.请求段表机制

在请求分段式管理中所需要的主要数据结构是请求段表

### 2.缺段中断机构

每当发现程序要访问的断不再内存中，就有缺段中断机构产生一个中断信号，由OS将所需段调入内存。 在一条指令的执行期间产生和处理中断，可能发生多次中断，但不可能出现一条指令被分割在两个段中。

#### 3.地址变换机构

被访问的段不再内存时，需要地址变换。具体做法是先将所缺的段调入内存，并修改段表，然后利用段表进行地址变换。

## <6>分段的共享与保护（P187-189）

### #1共享段表

记录了共享段的段号，段长，内存起始地址，状态(存在位)，外村起始地址，共享进程计数(count)

### #2共享段的分配与回收

#### 1.共享段的分配

当第一个使用共享段的进程提出请求时，由系统为该共享段分配一物理区，并调入该共享段，同时修改相应的段表（该段的内存地址）和共享段表，把 count 置为 1。当其它进程需要调用此段时，不需再调入，只需修改相应的段表和共享段表，再执行 count :=count+1 操作。

#### 2.共享段的回收

当共享共享段的某进程不再使用该共享段时，修改相应的段表和共享段表，执行 count :=count-1 操作。当最后一共享此段的进程也不再需要此段时，则系统回收此共享段的物理区，同时修改共享段表（删除该表项）。

### #3分段管理的保护

#### 1.地址越界保护

先利用段表寄存器中的段表长度与逻辑地址中的段号比较，若段号超界则产生越界中断。 再利用段表项中的段长与逻辑地址中的段内位移进行比较，若段内位移大于段长，也会产生越界中断。 注：在允许段动态增长的系统中，允许段内位移大于段长。

#### 2.访问控制保护（存取控制保护）

在段表中设置存取控制字段，用于规定对该段的访问方式。

#### 3.环保护机构

环保护机构是一种功能较完善的保护机制。在该机制中规定：低编号的环具有高优先权。 OS 核心处于 0 环内；某些重要的实用程序和操作系统服务占居中间环；而一般的应用程序则被安排在外环上。 在环系统中，程序的访问和调用应遵循一定的规则：

- 一个程序可以访问同环或较低特权环的数据
- 一个程序可以调用同环或较高特权环的服务

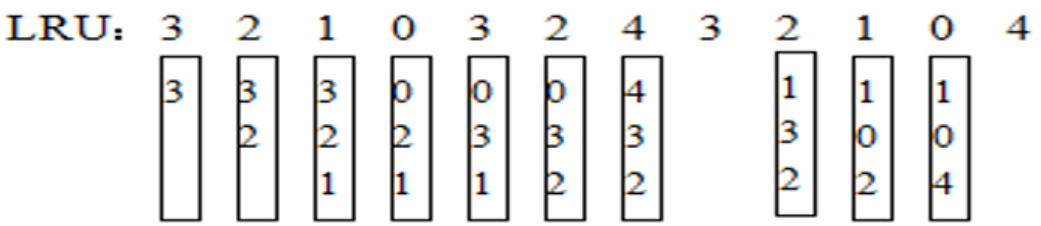
## <7>产生抖动的原因

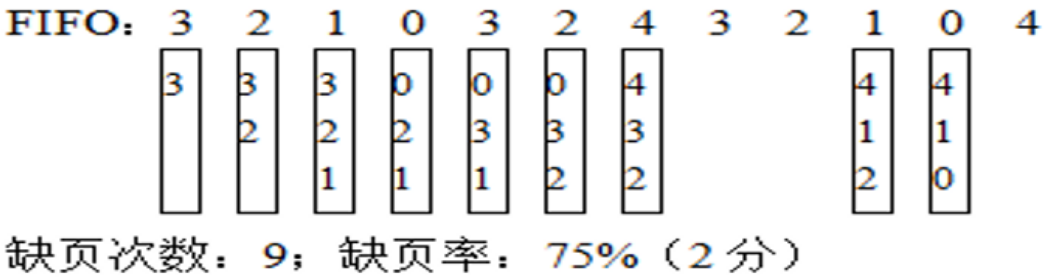
同时在系统中运行的进程太多，由此分配给每一个进程的物理块太少，不能满足进程正常运行的基本要求，使得每个进程在运行时，频繁地出现缺页，必须请求系统调页，这样使得进程大部分时间用于页面置换，处理机效率急剧下降。

# (二)练习题

## <1>练习题1

在一个请求分页存储管理系统中，进程P共有5页，页号为0至4。若对该进程页面的访问序列为3，2，1，0，3，2，4，3，2，1，0，4，试用最佳（Optimal）置换算法、LRU置换算法和FIFO置换算法，计算当分配给该进程的物理块数为3时，**访问过程中发生的缺页次数和缺页率。**




# 五.输入输出系统

## （一）知识点

### <1>按设备的共享属性可将设备分为什么？（P192）

独占设备，进程互斥地访问这类设备，即系统一旦将该类设备分配给某进程，便由该进程独占，直到用完释放。

共享设备，指一段时间内允许多个进程同时访问的设备

### <2>通道类型？（P200-201）

#### #1.字节多路通道(Byte Multiplexor Channel)

一种按照字节交叉工作的通道，不适用于连接高速设备，通常含有许多非分配型子通道，每个子通道连接一个I/O设备，这些子通道按照时间片轮转方式共享主通道。

#### #2.数组选择通道(Block Selector Channel)

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，直到程序释放通道。利用率很低，但是传输效率很高。

#### #3.数组多路通道(Block Multiplexor Channel)

数组多路通道结合了数组选择通道和字节多路通道，能使得各个子通道(设备)分时并行操作，含有多个非分配子通道。

### <3>设备控制器的组成--三部分组成（P198-199）

### #0设备控制器的功能

1. 接受和识别命令
2. 数据交换
3. 标识和报告设备的状态
4. 地址识别
5. 数据缓冲区
6. 差错控制

### #1设备控制器与处理机的接口

用于实现CPU和设备控制器之间的通信，该接口含有三类信号线：数据线，地址线和控制线

### #2设备控制器与设备的接口

用于连接一个或者多个设备

### #3I/O逻辑

用于实现对设备的控制

## <4>中断处理程序的处理过程（P204-205）

1. 测定是否有未响应的中断信号
2. 保护被中断进程的CPU环境
3. 装入相应的设备处理程序
4. 中断处理
5. 恢复CPU的现场并退出中断

## <5>SPOOLING、组成、特点（P220-222）

### 0.定义

在多道程序中，利用一道程序模拟脱机输入时的外围控制机功能，再利用另一道程序模拟脱机输出时外围控制机的功能，从而在主机的直接控制下，实现脱机输入输出。 在联机情况下实现的同时外围操作的技术，即为SPOOLing技术，假脱机技术。

### #1组成

1. 输入井和输出井
2. 输入缓冲区和输出缓冲区
3. 输入进程和输出进程
4. 井管理程序

### #2特点

1. 提高了I/O的速度
2. 将独占设备改造为共享设备
3. 实现了虚拟设备功能

## <6>引入缓冲区的原因？（P224-225）

1. 缓和CPU和I/O设备间速度不匹配的矛盾

- cpu的运算速度远远高于I/O设备的速度

2. 减少对CPU的中断频率，放宽对CPU中断响应时间的限制

- 随着传输速率的提高，需要配置位数更多的寄存器进行缓冲

3. 解决数据粒度不匹配的问题

- 缓冲区可以用于解决生产者和消费者之间交换的数据粒度(数据单元大小)不匹配的问题

4. 提高CPU和I/O设备之间的并行性

- 先放入缓冲，便可进行下一次操作，提高系统吞吐量和设备利用率

## <7>磁盘访问时间包括什么？（P232-233）

### #1寻道时间$T_s$

指的是把磁头移动到指定磁道上花费的时间，该时间是启动磁头的时间$s$和磁头移动$n$条磁道所花费的时间之和，即为 $T_s = m * n + s$ 其中m为一个常速，与磁盘驱动器的速度有关，一般磁盘，m为0.2；高速磁盘，m<=0.1

### #2旋转延迟时间$T_\tau$

指的是指定扇区移动到磁头下所花费的时间 比如硬盘为15000 r/min,则每转需要4ms,从而平均旋转延时$T_\tau$为2ms

### #3传输时间$T_t$

指的是把数据从磁盘读出或者向磁盘写入数据所花费的时间，其大小和每次读写的字节数b和旋转速度有关:
$T_t = \frac{b}{r*N}$ 其中r为磁盘每秒的转速，N为一条磁道上的字节数 当一次读写的字节数相当于半条磁道上的字节数时，$T_t$和$T_\tau$相同，因此，可将访问时间$T_a$表示为: $T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$

## <8>磁盘调度算法（FCFS, SSTF, SCAN, CSCAN），计算平均寻道长度（P233-235）

### #1FCFS

根据进程请求访问磁盘的先后顺序进行调度

### #2SSTF，最短寻道时间算法

选择这样的进程，其要求访问的磁道与当前磁头所在磁道距离最近，使得每次的寻道时间最短，但是不能保证平均寻道时间最短

### #3SCAN算法--电梯调度算法

不仅考虑将要访问的磁道和当前磁道间的距离，更优先考虑的是磁头当前的移动方向。 比如，当磁头向外移动时，SCAN算法考虑的下一个磁道应该是当前移动方向上距离最近的，直到最后再反向移动

### #4CSCAN算法

规定磁头单向移动，当磁头移动到最外磁道并访问后，磁头立即返回到最里面的欲访问磁道，即将最小磁道号和最大磁道号构成循环进行扫描

# (二)练习题

例题：假设一个磁盘的每个盘面上有200个磁道，现有多个进程要求对存储在该盘面上的数据进行访问，按照这些请求到达的先后次序，它们的访问位置分别处于54、57、39、18、90、162、154、38、180号磁道上。假定当前磁头在100号磁道上。请给出按“先来服务（FCFS）”、“最短寻道时间优先(SSTF)”和“扫描（SCAN）”算法进行磁盘调度时各自实际的访问次序，并计算它们的平均寻道长度。(注：当采用扫描算法时，规定当前磁头正在向磁道号增加的方向上移动)

解:

```
FCFS算法访问次序:54、57、39、18、90、162、154、38、180
          平均寻道长度：55.3
SSTF算法访问次序:90、57、54、39、38、18、154、162、180
          平均寻道长度：27.1
SCAN算法访问次序:154、162、180、90、57、54、39、38、18
          平均寻道长度：26.8
```

# 六.文件与磁盘管理

## (一)知识点

### <1>数据项、记录与文件（P237-238）

**数据项**是最低级的数据组织方式，可分为基本数据项和组合数据项

**记录**是一组关于数据项的集合，用于描述一个对象在某方面的属性

**文件**是指由创建者所定义的，具有文件名的一组相关元素的集合，可分为有结构文件和无结构文件两种，每个文件都有文件属性。

**文件属性**

- 文件类型
- 文件长度
- 文件的物理位置
- 文件的建立时间 三者层次关系
- 文件包含记录，记录包含数据项

### <2>文件的逻辑结构及分类（P242-248）

#### 1.定义

**文件的逻辑结构**，从用户观点出发看到的文件组织形式，即文件是由逻辑记录组成，是用户可以直接处理的数据及其结构，独立于文件的物理特性，又称为文件组织

**文件的物理结构**，又称为文件的存储结构，指系统将文件存储在外存上的存储组织形式

#### 2.分类

- 按是否有结构分为有结构和无结构文件
    - 有结构文件，又称为记录式文件，每个记录都用于描述实体集上的一个实体，各个记录有相同或者不相同的数据项，记录长度有定长和变长记录
    - 无结构文件，指由字符流构成的文件，又称为流式文件，

- 按文件的组织方式分为顺序，索引，索引顺序文件
    - 顺序文件，**指一系列记录按照某种顺序排列的文件**，记录可以定长和变长
        - 顺序文件有串结构和顺序结构
        - 顺序文件最适用于对文件记录进行批量存取时
        - 顺序文件不适用与交互应用的场景，并且增删记录困难
    - 索引文件，指可变长记录文件建立一张索引表，**为每个记录设置一个表项**，来加速对记录的检索速度
        - 索引文件按照关键字建立索引，也可以具有多个索引表
        - 索引文件提高了对文件的查找速度，插入和删除记录很方便
        - 索引文件增加了存储开销
    - 索引顺序文件，顺序文件和索引文件结合的产物，此时为每个文件建立一个索引表，不是每个记录设置一个表项，而是**为一组记录中的第一个记录建立一个表项**
        - 索引顺序文件客服了变长记录的顺序文件不能随机访问和不便于记录的删除和插入的缺点
        - 索引顺序文件引入了文件索引表来实现对文件的随机访问
        - 索引顺序文件增加了溢出文件，用于记录新增，删除和修改的记录

# <3>文件控制块与索引节点（P249-251）

## #1文件控制块FCB(File Control Block)

为了方便管理，含有三类信息

**1.基本信息类**

- 文件名
- 文件物理位置，
- 文件逻辑结构，流式文件等
- 文件物理结构，顺序文件等

**2.存取控制信息类**

- 文件主的存取权限
- 核准用户的存取权限
- 一般用户的存取权限

**3.使用信息类**

- 建立文件的相关日期和时间
- 文件上一次修改的日期和时间
- 当前使用信息
    - 当前已打开该文件的进程数目
    - 是否被其他进程锁住
    - 文件在内存中是否已经被修改但没有拷贝到盘上

## #2.索引节点

**磁盘索引节点**，每个文件有唯一的磁盘索引节点，包含内容

- 文件主标识符，即拥有该文件的个人或者小组的标识符
- 文件类型，包括正规文件，目录文件或者特别文件
- 文件存取权限，指的是各类用户的存取权限

- 文件物理地址，每个索引节点含13个地址项，即iaddr(0)~iaddr(12),以直接或者间接的方式给出数据文件所在盘块的编号
- 文件长度，以字节为单位的文件长度
- 文件连接计数，表明在本文件系统中所有指向该文件的指针计数
- 文件存取时间，指出本文件最近被进程存取的时间，最近被修改的时间，以及索引节点最近被修改的时间

**内存索引节点**，存放在内存中的节点，文件打开时，将磁盘索引节点拷贝到内次索引节点中，包含内容

- 索引节点编号，用于标识内存索引节点
- 状态，指示i节点是否上锁或者修改
- 访问计数，每当有一个进程要访问此节点时，访问计数加一，访问后减一
- 文件所属文件系统的逻辑设备号
- 链接指针，设置有分别指向空闲链表和散列队列的指针

# <4>外存的组织方式（P268-278）

**外存的组织形式分为连续组织方式，链接组织方式，索引组织方式**

## #1.连续组织方式

为每个文件分配一组相邻接的盘块 优点:

- 顺序访问容易
- 顺序访问速度快

缺点:

- 要求为一个文件分配连续的存储空间
- 必须事先知道文件的长度
- 不能灵活删除和插入记录
- 对于动态增长文件，很难为其分配空间

## #2.链接组织方式

为文件分配多个不连续的盘块，通过每个盘块上的链接指针将同属于一个文件的离散盘块形成链表，由此形成链接文件。

优点:

- 消除了磁盘的外部碎片，提高了外存的利用率
- 对插入，删除和修改记录非常容易
- 能适应文件的动态增长，无需事先知道文件的大小

缺点:

- 不能支持高效的直接存取
- FAT需要占用较大的内存空间

链接方式:

- 隐式链接，文件目录的每个目录项中都必须含有指向链接文件第一个盘块和最后一个盘块的指针
    - 只适用于顺序访问，对于随即访问效率极低
    - 此时为了提高检索速度和减少指针的开销，可以将几个盘块组成一个簇
- 显示链接，把用于链接文件各物理块的指针显示地存放在内存中的一张链接表中

- 提高了检索速度，减少了访问磁盘的速度
- 需要文件分配表(FAT),把偶才能分配给文件的所有盘块号

## #3索引组织方式

**1.单级索引组织方式-小文件**

为每个文件分配一个索引表(块)，把分配给该文件的所有盘块号都记录在表(块)中，在建立一个文件时，只要在为之建立的目录项中填入指向改索引块的指针。

优点:

- 支持直接访问
- 不会产生外部碎片
- 当访问文件较大时，索引组织方式优于链接组织方式

缺点:

- 对于小文件采用索引组织方式，，本身占有盘块极少，索引块利用率低

**2.多级索引组织方式-大文件**

为每个文件分配一个索引表(块)，把分配给该文件的所有盘块号都记录在表(块)中，在建立一个文件时，只要在为之建立的目录项中填入指向改索引块的指针。对于一个大文件，所分配的块已满，必须再分配一个索引块。

优点:

- 加快了对大文件的查找速度

缺点:

- 访问一个盘块时，其需要启动磁盘的次数随着索引级数增加而增加，对于小文件也是如此

**3.增量式索引组织方式--兼顾小中大文件（混合上种组织方式，针对性设置组织方式）**