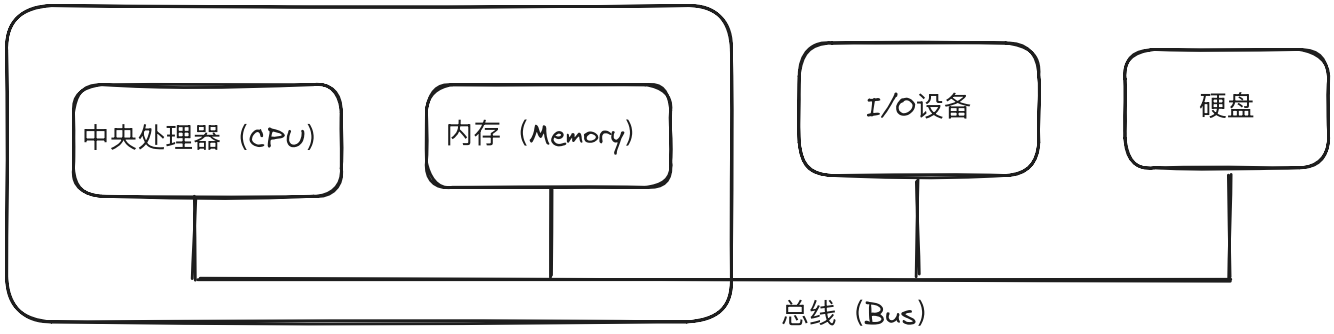
操作系统笔记

01-操作系统概述

01 计算机体系结构



CPU：负责进行运算

内存：存储运行时的所有信息

I/O设备：用户与计算机进行输入输出的桥梁

总线：数据传递的通道

02 指令：计算机运行的最小功能单元

操作码：指明该条指令的功能

操作数：指明操作的对象

执行程序时，指令和数据全部存放在内存中

03 计算机程序：计算机能够识别和执行的一组指令的集合

编写程序的步骤：

1. 确定核心任务：输入和输出
2. 算法设计：分而治之，形式化表达
3. 编码实现：机器语言->汇编语言->高级程序设计语言的演化
4. 测试与调试：确保程序正常运行

04 操作系统概念：硬件和应用软件间引入一层专门的软件

主要功能：

1. 管理系统的各个功能部件
2. 给上层应用软件提供易于理解和编程的接口

05 操作系统发展历史

1. 电子管时代：手工操作效率低下造成CPU等待时间过长
2. 晶体管时代：引入单道批处理，串行执行作业；无法实时调试，对输入/输出为主的作业CPU资源浪费
3. 集成电路时代：引入多道批处理系统，标志现代意义上的操作系统的出现，宏观上并发，微观上对单个设备串行；引入分时操作系统，分时提供服务，解决用户对响应时间的要求
4. 个人计算机时代与移动计算机时代：操作系统趋于成熟

06 操作系统的类型

1. 主机操作系统：批处理系统、事务处理系统、分时系统
2. 服务器操作系统：为客户机提供共享硬件和软件资源
3. 个人计算机操作系统：功能强大、UI界面、应用软件、多任务多硬件支持
4. 手持设备操作系统：安卓系统与iOS系统
5. 嵌入式操作系统：以应用为中心、计算机技术为基础，软硬件可裁剪，对功能、可靠性、成本、体积、功耗和应用环境有严格要求的专用计算机系统，将应用程序、操作系统和计算机硬件集成在一起
6. 实时操作系统：及时响应外部事件请求，在规定严格时间内完成对事件的处理，并控制所有实时设备和实时任务协调一致地工作

07 操作系统：系统资源的管理者

特权指令：受保护指令，只有操作系统有权使用的指令

CPU工作状态：内核态和用户态

程序状态字（Program Status Word，PSW）寄存器：指示处理器的状态

包括工作状态码、条件码、中断屏蔽码等
处理器状态的转换通过修改状态寄存器中的状态标志位

08 系统调用

将常用的内核功能封装为函数的形式供用户程序使用，实现过程为：

1. CPU执行仿管指令或陷阱指令时，引起中断，成为访管中断或陷阱中断

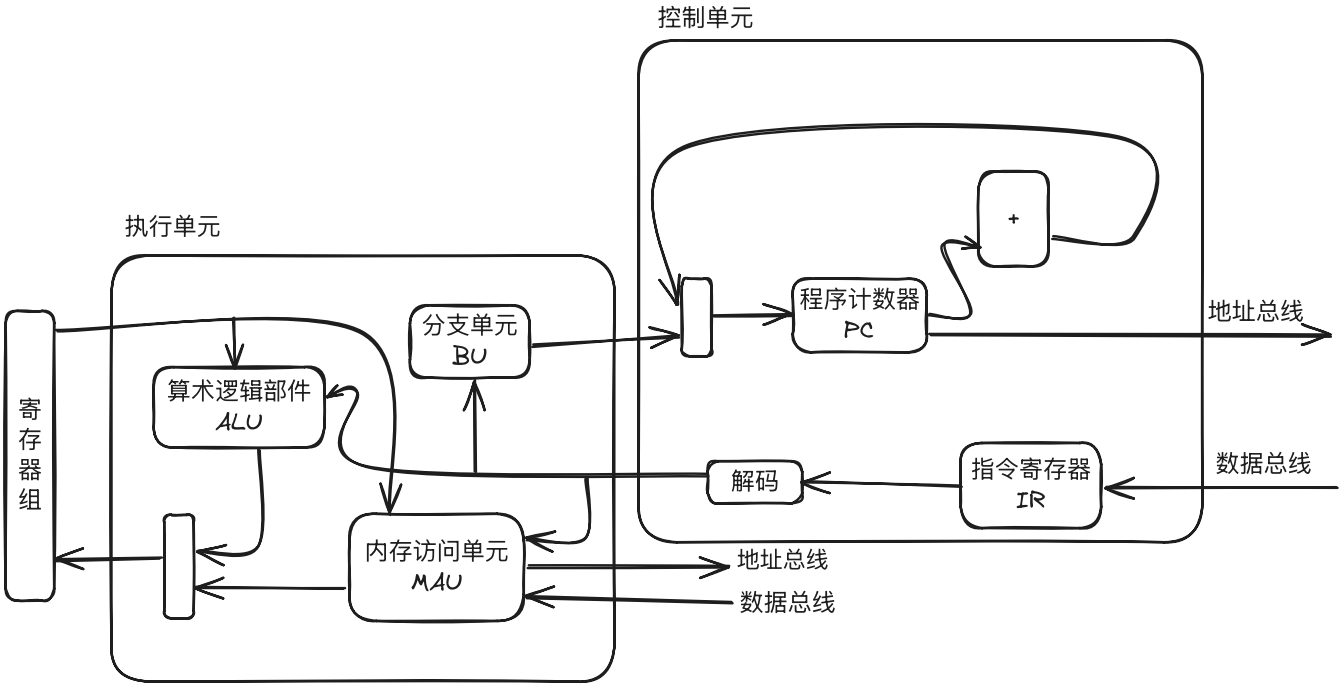
2. 处理器保存中断点的程序执行上下文环境，CPU状态被切换至内核态，其是CPU在中断时自动完成的
3. 处理器将控制权转移至相应的中断处理程序，调用相应的系统服务
4. 中断处理结束后，CPU恢复被中断程序的上下文环境，CPU恢复至用户态，回到中断点继续执行
   高级程序设计语言通过库函数间接地启动系统调用

02-进程

01 程序的执行

程序是计算机能够识别、执行的一组指令的集合

CPU结构图：



程序的执行流程为：

1. 执行所在的起始地址存放在PC中，控制单元通过地址总线将其发送给内存，内存在对应的内存单元中取出指令并通过数据总线返回给IR，并通过专门的解码硬件完成指令的分析
2. 在取得一个指令后，PC自动进行加法运算，从而进行下一条指令的获取
3. 若某指令为分支或跳转指令，则通过BU对PC寄存器的值进行修改并给出下一条指令的起始地址
4. 指令在完成分析后，通过MAU访问内存中的数据，通过寄存器组进行数据的暂存，通过ALU进行对应的逻辑运算，从而完成整个指令的执行

寄存器

寄存器是位于CPU内部的一些小型存储区域，用于暂时存放数据、指令和地址，其速度快但容量有限

02 进程的概念

为了提高计算机系统中各种资源的使用效率，设计出了多道程序（Multi-programming）技术，使得程序可以并发运行，并进一步提出了进程（Process）的概念

进程的组成部分：

1. 程序的代码
2. 程序的数据
3. CPU寄存器的值
4. 堆（Heap）：用于保存进程运行时动态分配的内存空间
5. 栈（Stack）：保存上下文信息和函数调用时产生的形参和局部变量
6. 系统资源：包括地址空间和打开的文件等
   程序是一个静态的概念，由代码和数据构成；进行是一个动态的概念，由程序和该程序的运行下上文
   一个程序在运行时对内存资源的需求，用代码段存放代码，用数据段存放全局变量，用栈存放局部变量，用堆存放动态变量

03 进程的特性

动态性

进程是一个正在运行的程序，是一个动态的概念，其在运行的过程中状态是不断变化的

独立性

进程是计算机系统资源的使用单位
每个进程都有属于自己的逻辑寄存器，其本质是一个个内存变量

并发性

进程宏观上并发，微观上串行执行

04 进程创建与终止

进程的创建

在一个已经存在的进程中，通过系统调用或者库函数来创建新进程，创建者可以是用户进程和系统进程

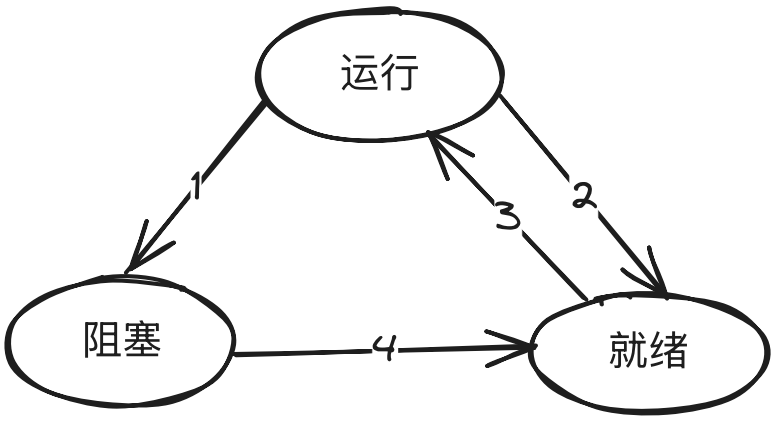
进程的终止

1. 自愿退出
2. 致命错误
3. 被其他进程所杀

05 进程的状态

1. 运行状态（Running）： 进程占有CPU
2. 就绪状态（Ready）： 已经具备了运行的条件，但CPU正忙，暂时不能运行
3. 阻塞状态（Blocked）： 因为等待某种事件的发生而暂时不能运行

进程状态转换：



1:进程在CPU上运行时，等待输入/输出操作，于是将运行态转换为阻塞态
2:由调度程序将进程由运行态转化为就绪态
3:由调度程序将进程由就绪态转化为运行态
4:等待事件发生，运行条件具备，由阻塞态转化为就绪态
就绪态等CPU，阻塞态等I/O

06 进程控制块（PCB）

程序=数据结构+算法
描述和管理进程的数据结构即为进程控制块（Process Control Block）
PCB主要包括进程管理、存储管理、I/O管理三方面

PCB主要内容：

进程管理	存储管理	I/O管理
CPU寄存器的值	基地址寄存器的值	I/O设备列表
进程的描述信息（进程号、进程状态等）	长度寄存器的值	当前工作目录
进程调度信息（优先级、调度参数）		进程打开文件列表

PCB是进程存在的唯一标志
对进程的组织和管理通过对PCB的组织和管理实现

07 状态队列与进程模型

由操作系统来维护一组队列，用于表示系统中所有进程的当前状态，其可通过链表的形式实现

多道程序系统中进程的运行模式：

1. 每个进程的状态是在不断变化的，在不同时刻用到的资源也是不一样的
2. 系统内核也在内存中，通过中断来实现在处理器上的运行
3. 进程之间的切换是由系统内核完成的
4. 系统内核不一定以独立进行的形式存在
5. 不同的硬件资源可以并行工作

03-线程

01 线程的引入

更小的能独立运行的基本单位：线程（Thread）
其满足：

1. 各线程之间可以并发地运行
2. 各线程之间可以共享相同的地址空间

线程是进程中的一条执行流程
进程=线程（代码的执行流程）+资源平台
线程共享进程的代码、全局变量、动态内存空间；独享CPU寄存器的值和栈

进程与线程的比较：

1. 进程是资源的分配单位，线程是CPU的调度单位
2. 进程拥有完整的资源平台，而线程只独享必不可少的资源
3. 线程同样具有三种基本状态
4. 线程能有效地减少并发执行的时间和空间开销

02 线程的实现

线程的数据结构TCB：

1. 线程自身的管理信息，如线程标识符、线程状态、调度信息
2. 进程资源平台上线程独享的资源，包括CPU寄存器的值和栈指针

线程的两种实现方式：用户线程和内核线程

用户线程

1. 用户线程的维护是由相应的进程通过线程库函数来完成的，用户空间需要增加一层线程管理的库函数
2. TCB由线程库函数来维护
3. 用户线程的切换由线程管理库函数来完成
4. **缺点：阻塞性的系统调用难以实现，一个线程发起系统调用，整个进程将被阻塞**

内核线程

1. TCB位于内核空间中，由内核程序维护，一般的用户程序不能访问
2. 线程的创建、终止、切换通过系统调用的方式进行，系统开销较大
3. 某线程由于发起系统调用而被阻塞，并不影响其他线程的运行
4. 进程内线程越多，获得的CPU时间可能就越多

03 线程库

线程库提供一组API函数，用于创建和管理线程，实现多线程编程

04-进程间通信与同步

01 进程间通信方式

低级通信

传递少量的控制信息，如信号量、信号

高级通信

1. 共享内存：操作系统负责把虚拟地址映射到物理内存，操作系统提供API函数，允许多个进程将自己的虚拟地址空间的部分共享，映射到相同的一块物理内存区域
2. 消息传递：操作系统维护消息机制，进程间建立通信链路，通过调用发送和接受操作来交换信息

3. 管道：管道通信以文件系统为基础，管道是指连接两个进程之间的一个打开的共享文件，专门用于进程之间的数据通信，发送进程从管道一段写入数据流，接受进程在管道另一段按照先进先出的顺序读出数据

02 进程的互斥

临界区：访问共享资源的代码段
互斥访问的三个要求：

1. 任何两个进程都不能同时进入临界区
2. 一个进程运行在临界区外时，不能妨碍其他进程进入临界区
3. 任何一个进程进入临界区的请求应该在有限时间内得到满足

03 基于关闭中断的互斥实现

一个进程进入临界区后，将中断关闭，因为操作系统是由中断驱动的，只有发生中断时，操作系统才能获得控制权，实现进程间切换

缺点：影响系统性能、不适合用户进程、不适用于多CPU系统

04 基于繁忙等待的互斥实现

加锁标志位法

设置共享变量lock，进入临界区前先访问lock，如有则修改lock并进入，若无则循环等待
缺点：共享变量lock本身会成为竞争访问的对象，并引出新的竞争状态

强制轮流法

每个进程严格地按照轮流的顺序进入临界区，使用共享变量turn作为标志
缺点：当一个进程有资格进入临界区单由于其他事项而并未进入，会阻止其他进程进入，违反了互斥访问的第二条要求

Peterson方法

该方法准确地来说是解决两个线程之间的互斥访问问题

```
#define FALSE 0
#define TRUE 1
#define N 2 //进程的个数
int turn;  //轮到谁?
int interested[N]; //兴趣数组 初始值均为FALSE
void enter_region(int process) //process为0或1
{
```

```
    int other; //另外一个进程的进程号
    other = 1 - process;
    interested[process] = TRUE; //表明本进程感兴趣
    turn = process; //设置标志位
    while(turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE; //本进程已离开临界区
}
```

设置双重判断，不仅检查turn（turn是在自己想要访问临界区时修改的），还要检查对方的兴趣数组，这样无论中断发生在何处，均可实现互斥访问
若同时进入while语句，则兴趣数组均为TRUE，而turn要么是1要么是0，两个进程不可能同时通过while语句判断

基于繁忙等待方法小结

缺点：

1. 浪费了CPU时间，存在这样一种情况：操作系统采用优先级调度算法，一个低优先级的进程处于临界区中，而一个高优先级的进程就绪并试图进入临界区，则高优先级进程被调度至CPU上运行，低优先级进程无法得到CPU而不能退出临界区，高优先级进程占据CPU但无法进入临界区，即发生了死锁。**对应解决方法**：当一个进程无法进入临界区时，将其阻塞并让出CPU
2. 任何时刻只能运行一个进程进入临界区

05 信号量

信号量（Semaphore）由荷兰计算机科学家Dijkstra提出

信号量由操作系统来维护，用户进程只能通过初始化和两个标准原语，P原语与V原语来访问信号量
原语是一种原子操作，在执行过程中不会被中断打断，可以一次执行完毕
当等待时间较短时，繁忙等待的方式更好，无需进程切换，没有额外的时间开销

```
typedef struct
{
    int count; //计数变量
    struct PCB *queue; //进程等待队列
}semaphore;

P(semaphore *S)
```

```
{
    S->count--; //申请一个资源
    if(S->count<0) //表示没有空闲资源
    {
    把该进程修改为等待状态;
    把该进程加入到等待队列 S->queue;
    调用进度调度器;
    }
}

V(semaphore *S)
{
    S->count++; //释放一个资源
    if(S->count<=0) //表示有进程被阻塞
    {
    从等待队列 S->queue中取出一个进程;
    把该进程修改为就绪状态，加入就绪队列
    }
}
```

07 进程的同步与经典IPC问题

互斥处理的是对共享资源的竞争问题，同步处理的是多个进程之间的时序问题
同步就是先后顺序，互斥就是同时竞争
考虑基于信号量的进程间同步问题，P（S）代表申请一个资源，V（S）代表释放一个资源

生产者与消费者问题

1. 对于生产者进程，每生产一个产品，需要检查缓冲区是否有空位，若是，则将产品送入缓冲区；否则，必须等待消费者取出一个产品后，才能将产品送入缓冲区
2. 对于消费者进程，在取产品时，必须先检查缓冲区是否有产品可取，若有，则取走并在必要时通知生产者进程；否则，必须等待生产者将一个新产品放入缓冲区后，才能去取

```
semaphore BufferNum; //初值为N
semaphore ProductNum; //初值为0
semaphore Mutex; //初值为1

void producer(void){
    int item;
    while(TRUE){
        item = produce_item();
        P(BufferNum); //是否有空闲缓冲区
        P(Mutex); //进入临界区
        insert_item(item); //把产品放入缓冲区
        V(Mutex); //离开临界区
```

```
            V(ProductNum); //释放产品资源
        }
    }

    void consumer(void){
        int item;
        while(TRUE){
            P(ProductNum); //是否有产品资源
            P(Mutex); //进入临界区
            item = remove_item(); //从缓冲区取走产品
            V(Mutex); // 离开临界区
            V(BufferNum); // 释放一个缓冲区资源
            consume_item(item);
        }
    }
```

生产者关注对象为缓冲区是否有空闲，消费者关注缓冲区是否有产品，两者关注对象不一样，用两个信号量分别表示，其和为N，信号量Mutex则用于控制对缓冲区的互斥访问，即同一时刻最多只能有一个对象处于缓冲区

哲学家就餐问题

哲学家就餐问题由**Dijkstra**提出，考虑有五位哲学家围坐在一张圆桌旁，每个哲学家左右均有一根筷子，共五个筷子，每一位哲学家的动作只有两种：思考和进餐。哲学家在感到饥饿时，会试图获得左右两边的筷子，每次只能取一根，两根筷子均拿到后，才能开始进餐
考虑一种方式：对于每一位哲学家，不能手里拿着一个筷子，同时又等待另一个筷子，关键在于每次虽然只能取一根，我们要使得每位哲学家在申请筷子时，要么不拿，要么就拿两根

```
#define N 5
#define LEFT (i+N-1) % N
#define RIGHT (i+1) % N
#define THINKING 0
#define HUNGRY 1
#define ETAING 2
int state[N]; //状态记录数组 该数组为临界区
semaphore mutex; //初值为1
semaphore S[N]; //初值为0

viod philosopher(int i){
    while(TRUE){
        think();
        take_sticks(i);
        eat();
        put_sticks(i);
    }
```

```
}

void take_sticks(int i){
    P(mutex); //该信号量主要用于对状态数组的互斥访问
    state[i] = HUNGRY; //将状态改为饥饿
    test(i); //试图获取筷子
    V(mutex); //退出临界区
    P(s[i]); //在test中若未获得两根筷子 此时阻塞
}

void test(int i){
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) //左右筷子均空闲
    {
        state[i] = EATING; //状态改为进餐
        V(s[i]); //表示一次获取两根筷子
    }
}

void put_sticks(int i){
    P(mutex);
    state[i] = THINKING; //完成进餐后状态改为思考
    test(LEFT); //放下筷子后 若左邻居在阻塞 则唤醒
    test(RIGHT); //右邻居一样
    V(mutex);
}
```

读写者问题

在任何时刻，可以允许多个进程来读，但如果有一个进程要修改数据，则在此期间，所有其他进程均不能访问
考虑一种读者优先的策略：

1. 若有新读者，无写者在写，则无论是否有写着在等待，均可以读；否则，等待写者写完
2. 若有新写者，若有读者在读或写者在写，均等待；否则，写者可以写

```
int rc = 0;
semaphore mutex;
semaphore db;

void writer(void){
    think_up_data();
    P(db); //没有读者且没有写者时才能申请到该资源 否则阻塞
    write_data_base();
    V(db);
```

```
}

void reader(void){
    P(mutex);//由于读者是多个 互斥访问计数单元rc
    rc ++;
    if(rc == 1 )P(db);//如果是第一个读者 则申请数据库资源 后续读者无需申请可以直接读
    V(mutex);
    read_data_base();
    P(mutex);
    rc --;
    if(rc == 0) V(db);//直到最后一个读者读完 才释放数据库资源
    V(mutex);
    use_data_read();//使用数据访问非临界区
}
```

05-进程调度

01 进程调度基本概念

进程分为CPU繁忙和I/O繁忙

进程调度发生的五种情形：

1. 一个新进程被创建时，选择立即执行新进程还是继续执行父进程
2. 当一个进程执行完毕时
3. 当一个进程因为I/O操作、信号量或其他原因而被阻塞时
4. 当一个I/O中断发生时
5. 在分时系统中，当一个时钟中断发生时

进程调度的两种方式：

1. 不可抢占调度方式
2. 可抢占调度方式

调度算法的目标与评价指标
用户角度：

1. 周转时间：从第一次提交到完成的时间$T_i = E_i - S_i$
2. 平均周转时间：$T = \frac{1}{N}\sum_{i=1}^{N} T_i$
3. 平均带权周转时间：$W = T = \frac{1}{N}\sum_{i=1}^{N} \frac{T_i}{r_i}$，其中$r_i$表示作业实际执行时间
4. 等待时间：作业在就绪队列中的等待时间
5. 响应时间：作业从提交到第一次得到响应的时间
   操作系统角度：

6. 吞吐量：单位时间内完成作业的个数
7. CPU利用率
8. 设备的均衡利用

02 先来先服务算法（FCFS）

不可抢占调度方式，各个进程按照到达的先后顺序执行
缺点：

1. 如果短进程位于长进程之后，会增大平均周转时间
2. 无法充分利用CPU繁忙和I/O繁忙作业间的互补关系

03 短作业优先法（SJF）

两种实现方案：

1. 不可抢占式：当前进程正在运行时，不能被打断，只有在运行完毕或被阻塞时，才会按照作业时间长短进行调度
2. 可抢占式：如果一个新进程到来，其运行时间小于当前正在运行进程的剩余时间，则抢占CPU进行运行，此种方式又称之为最短剩余时间优先（SRTF）

SJF能有效降低作业的平均周转时间，对于一批同时到达的作业，SJF能得到一个最小的平均周转时间

缺点：在一个进程运行前，并不能准确知道其运行时间；可能会导致饥饿状态

04 时间片轮转法（RR）

按照到达顺序排列成就绪队列，并依次分配时间片运行

优点：

1. 公平性：平均分配CPU时间
2. 活动性：每个进程最多等待$(n-1)q$
3. 较短的平均响应时间

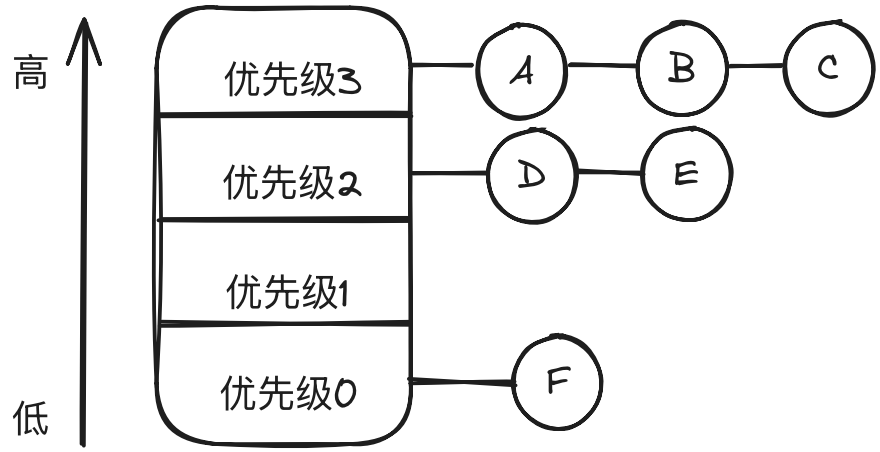
时间片太大退化为FCFS；时间片太小则增大了系统的管理开销，降低了CPU的使用效率
一般为20ms—50ms

05 优先级算法

给每个进程设置优先级，在就绪进程中选择优先级最高的运行
优先级算法可以分为可抢占方式和不可抢占方式

确定优先级的依据包括

1. 进程的类型：系统进程高于用户进程；交互式进程高于批处理进程
2. 对系统资源的需求：对CPU和内存需求较少的进程如I/O繁忙的进程优先级就较高（但实际事先并不能知道是否为I/O繁忙）
3. 用户需求：如用户级别等
   按照优先级分类作业，同一优先级采用RR：



缺点及解决方法：
饥饿问题：低优先级始终得不到CPU运行
解决方法：采用**动态优先级方式**，一方面，在每个时间片用完时，将当前正在运行的进程优先级降低；另一方面，如果一个进程等待时间不断延长，它的优先级会不断提高
动态优先级方式可以用于提高设备利用率：一个简单的实现方式是将进程的优先级设为$\frac{1}{f}$，其中f是该进程在上一个时间中所用CPU的时间比例

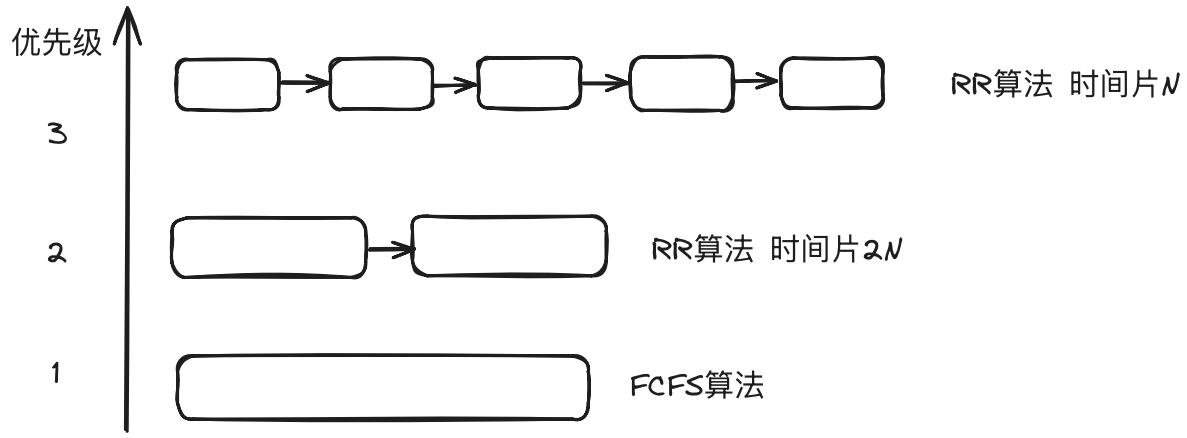
优先级反转问题：一个低优先级进程T1运行并获得了互斥锁，一个高优先级进程T2在到来后抢占CPU运行并试图获取互斥锁而被阻塞，这时一个优先级低于T2而高于T3的进程到来并抢占了CPU，T2优先级高却始终无法运行
解决方法：采用**优先级继承**的方法，如果一个高优先级进程在等待某个互斥锁，而该锁被某个低优先级的进程所占用，那么这个低优先级的进程将临时继承高优先级

06 多级反馈队列（Multilevel Feedback Queue）

引入多个就绪队列，根据进程在运行过程中的**反馈信息**，来动态的调整其所在的队列
多级反馈队列需要确定的参数：

1. 队列的个数
2. 每个队列采用的调度算法
3. 确定进程“升级”的方法
4. 确定进程“降级”的方法
5. 确定进程初始队列的方法

一个多级反馈队列的例子：



初始规则：新进程就绪，优先级设置为3，并加入队列3的末尾
降级规则：该进程在N内未能执行完毕，则优先级将为2并加入队列2的末尾；若在2N内为完成，则优先级降为1
升级规则：一个进程在时间片用完之前就被阻塞，则增加一个优先级
此外，只有较高优先级队列为空时，才调度优先级较低队列中的进程去执行，并且进程是抢占式的

优点：

1. 有效减少进程切换的次数
2. 为了提高系统吞吐量和缩短平均周转时间而照顾短进程
3. 为了获得较好的设备利用率和降低响应时间而照顾I/O繁忙进程
4. 不必估算执行时间，采用动态调节的方式

06-死锁

01 资源

死锁产生的根本原因是**对资源的竞争访问**
资源可分为两类：

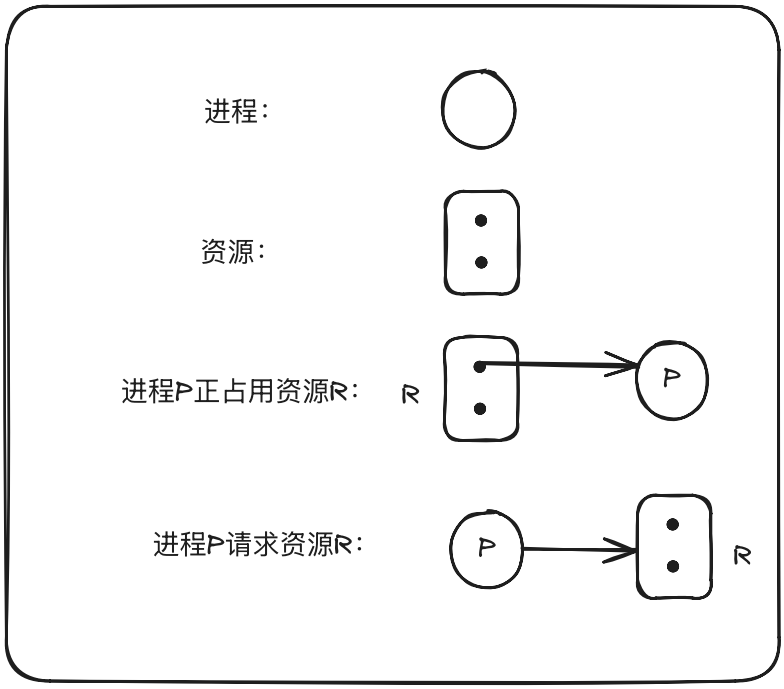
1. 可抢占资源：如CPU和内存
2. 不可抢占资源：如光盘刻录机
   对于可抢占的资源，可以通过重新分配资源的方法来避免死锁
   进程使用资源的三步骤：申请资源、使用资源、释放资源

02 死锁类型

Coffman提出了死锁发生的四个条件：

1. **互斥**条件：任何时刻每个资源最多只能被一个进程所使用
2. **请求和保持**条件：进程在占用若干个资源的同时又可以去访问新的资源
3. **不可抢占**条件：对已经占用的资源，只能等待进程主动释放
4. **环路等待**条件：存在一条多个进程所组成的环路链，每一个进程等待环路链中下一个进程所占的资源

Holt提出用资源分配图来描述死锁发生的四个条件，改进后的资源分配图为：



03 死锁的检测和解除

死锁的检测：设计算法判断有向图中是否存在环路
死锁解除的三种方法：

1. **剥夺资源**：强行对资源进行剥夺，对被剥夺资源的进程所造成的影响取决于资源自身的性质，会影响进程的正常进行
2. **进程回退**：定期将进程的状态信息保存在文件中，记载进程在不同时刻的状态信息，包括内存映像和所占资源状态，检测到死锁时，将其中一个资源拥有者回退，该方法安全但增加了系统的时间和空间开销
3. **撤销进程**：撤销一个或多个处于死锁状态的进程，抢夺资源并重新分配，对此应尽可能的选择那些能够安全地重新运行的进程

04 死锁的避免

假设系统提前得知每一个进程将在何时申请和释放某个资源，资源分配算法就可以判断出当前系统处于安全状态还是不安全状态
基于此，Dijkstra在1965年提出了避免死锁的调度算法——**银行家算法**：

算法所需四个数据结构：总资源向量、空闲资源向量、分配矩阵、请求矩阵
考虑：总资源向量$E=(3,12,14,14)$、空闲资源向量$A=(1,6,2,2)$
分配矩阵C：

	R1	R2	R3	R4
P1	0	0	3	2
P2	1	0	0	0
P3	1	3	5	4
P4	0	3	3	2
P5	0	0	1	4

请求矩阵R：

	R1	R2	R3	R4
P1	0	0	1	2
P2	1	7	5	0
P3	2	3	5	6
P4	0	6	5	2
P5	0	6	5	6

通过以下步骤判断初始状态T是否安全：

1. S1：在请求矩阵R中，寻找某一行R_i，其每个分量均小于等于A，若不存在则说明系统可能陷入死锁
2. S2：如果R_i存在，则满足其请求资源，并在运行结束后释放其资源
3. S3：重复上述步骤，直至所有进程均运行结束，则说明最初的状态T是安全的

05 死锁的预防

通过破坏死锁产生的四个必要条件之一来进行死锁的预防

破坏互斥条件

如进程在进程打印时，实际上是将任务提交给后台打印进程，由打印进程真正的使用打印机

破坏请求和保持条件

不允许进程在占用资源的同时又去申请新的资源，具体的两种实现方法：

1. 要求进程在运行前一次性请求所有资源，操作系统要么全部都给、要全全部不给
2. 要求进程在请求一个新资源时，先暂时释放其占用的各个资源

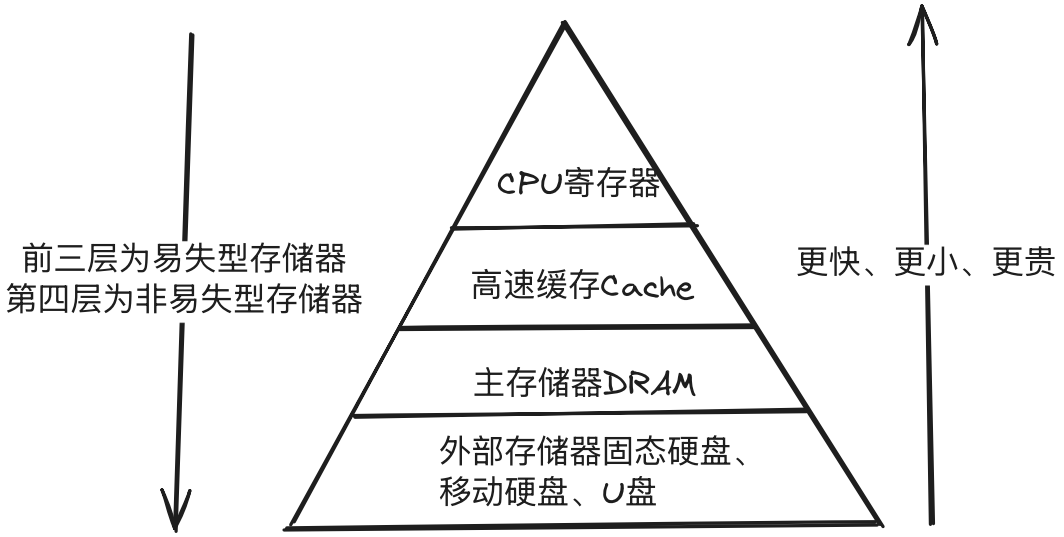
破坏环路等待条件

对系统中所有的资源进行编号，进程在申请资源时，必须严格地按照资源编号的递增次序进行

07-单道程序存储管理

01 存储管理概述

存储器层次结构：



CPU访问内存步骤：

1. 1.通过地址总线传输读取内存单元的地址
2. 2.通过控制总线向内存发送读命令
3. 3.内存取出对应内存单元的数据并通过数据总线传输给CPU

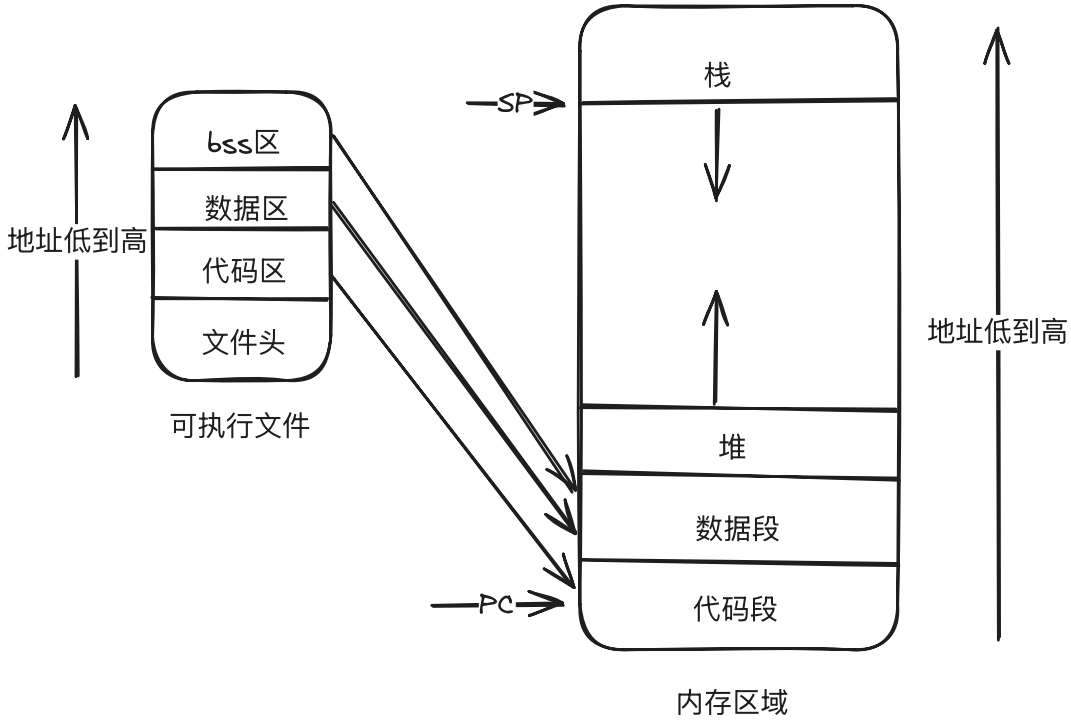
02 单道程序存储管理

内存划分为系统区和用户区，每次只装入一个程序
内存的回收：直接使用新程序覆盖
对于每个数据，需要考虑：存储位置、作用域、生存期

对于一个可执行文件，其内部结构为：

文件头	代码区	数据区	bss区	其他
int a_magic	mov da, ax	int func_static	int file_static[50];	

1. 文件头存放可执行文件的描述信息，如每个分区的起始地址和长度
2. 代码区存放编译后的可执行指令
3. 数据区存放全局变量和静态变量
4. bss区存放未赋初值的变量
   形参、局部变量和动态内存是在程序运行时直接在内存中
   程序装入内存示意图：
   如果发生了函数调用，就在栈中分配一块栈帧用于存放形参和局部变量；如果要访问动态内存空间，就访问堆



缺点：

5. 无法实现程序并发运行
6. 内存资源使用效率不高
7. 没有内存保护
8. 地址空间有限

08-分区存储管理

01 固定分区存储管理

内存空间管理

设置多个小分区、适当中等分区、少量大分区
采用**内存分配表**记录内存使用状况

分区号	起始地址	长度	状态	进程名
……	……	……	……	……

内存分配与回收

设置输入队列，当进程访问内存时，按照两种方法进行内存分配：

1. 最先匹配法：选择离队首最近的、能装入该分区的进程
2. 最佳匹配法：选择能够装入该分区的最大进程
   内存的回收仅需将内存分配表对应分区的状态改为空闲即可（即指示该分区可覆盖）

优点：易于实现、系统开销小
缺点：

1. 内存利用率不高，**内碎片**浪费（**进程所占分区内部未利用的空间**）
2. 分区总数固定，限制并发执行的程序个数
3. 缺乏内存保护
4. 每个进程的地址空间大小有限

02 可变分区存储管理

内存管理的数据结构

通过**分区链表**来记录内存使用情况
每部分包括：占用情况、起始地址、长度
并按照起始地址的递增顺序排列

分区匹配算法

1. 最先匹配法（First-fit）：从链表首结点开始寻找可分配分区，**其尽可能利用低地址部分的空闲区，保留较大的空闲区域**
2. 下次匹配法（Next-fit）：记录每次分配的空闲结点位置，下次分配从该结点开始往下找，到结尾时则重新回到开头，**其使空闲分区在内存中更加均匀，但较大的空闲分区不易保留**
3. 最佳匹配法（Best-fit）：将进程装入与其大小最接近的空闲分区，**但分割后剩余的空闲区域会很小，甚至成为外碎片**
4. 最坏匹配法（Worst-fit）：将最大的空闲分区分割一部分给请求进程，**其避免了空闲分区越分越小的问题**

分区回收算法

将该分区标记为空闲，并将相邻的空闲分区合并为一个空闲分区

外碎片与解决办法

解决外碎片可以采用内存紧缩技术：将所有进程都尽可能的网内存地址的低端移动
其需要大量CPU时间并且需要进行地址重定位

注：固定分区产生内碎片、可变分区产生外碎片

09-内存抽象与地址映射

01 内存抽象

进程并不能直接访问物理内存，每个进程都有一个相互独立、模仿物理内存的地址空间，进程通过该内存地址来访问内存，同时操作系统会为每个进程都创建一个虚拟的CPU

02 地址映射

目标代码采用**相对地址**的形式，其首地址为0，其余指令的地址相对于首地址来进行编址
操作系统需要将程序中的逻辑地址转换为运行时由机器直接寻址的物理地址，这个过程叫做**地址映射**或者**地址重定位**

地址映射的方式：

1. **静态地址映射**：一次性的实现逻辑地址到物理地址的转换，但转换后程序在内存中的位置不能再更改
2. **动态地址映射**：动态重定位，在程序运行的过程中，当需要访问内存单元时，才进行地址转换；即逐条执行指令时，来完成地址的转换
   地址转换工作通过专门的硬件**基地址寄存器**（重定位寄存器）来完成，当进程被调度运行时，其所在分区的起始地址被装入基地址寄存器中，程序运行需要访问内存单元时，硬件会自动地将其中的相对地址加上基地址寄存器中的内容，从而得到实际的物理地址

03 存储保护

在基地值寄存器上增加一个限长寄存器，其内存储分配给进程的内存分区的长度，当需要访问内存单元时，硬件会将相应的逻辑地址和限长寄存器中的值进行比较，以防止其**越界访问**

04 管理存储单元

CPU的组件中包括CPU和MMU（Memory Management Unit，存储管理单元），MMU专门负责将逻辑地址转换为对应的物理地址
在分区存储管理中，MMU包括基地址寄存器和限长寄存器
在页式存储管理中，MMU包括页表基地值寄存器和页表长度寄存器
在段式存储管理中，MMU包含段表基地值寄存器和段表长度寄存器

10-页式存储管理

01 基本原理

将物理内存划分为多个固定大小的内存块，称为物理页面或者页框
同时将逻辑地址空间也划分为大小相同的快，称为逻辑页面
页面的大小要求是2的整数次幂
逻辑页面在装入内存后，其存放位置不一定是连续的

02 数据结构

系统为每个进程建立一个**页表**，用于给出对应的物理页面（注意每个进程都有属于自己的页表）

内存块号	**M**	**N**	**……**
逻辑页号	0	1	……

同时，需要建立物理页面表，可以采用**位示图**或者**空闲页面链表法**
如256个页面，可以用8个字长为32的字段来表示，字的每一位用0/1来表示对应页面是否被占用，最后增加一个字段用于记录当前剩余的总空闲页面数

03 内存的分配与回收

内存分配过程如下：

1. 对于新进程计算其所需页面树N，查看位示图是否有N个空闲页面
2. 如果有，则申请一个页表，长为N，将页表的起始地址填入进程的PCB中
3. 分配N个空闲的物理页面并填入页表
4. 修改位示图，并将总空闲页面树减N

04 地址映射

地址映射的步骤：

1. 对于给定的逻辑地址，找到对应的逻辑页面号和页内偏移地址
2. 根据逻辑页面号查找对应的物理页表
3. 根据物理页面号和页内偏移地址，计算最终的物理地址
   对于一个逻辑地址，可以将其高位部分作为逻辑页面号，低位部分作为页内偏移地址

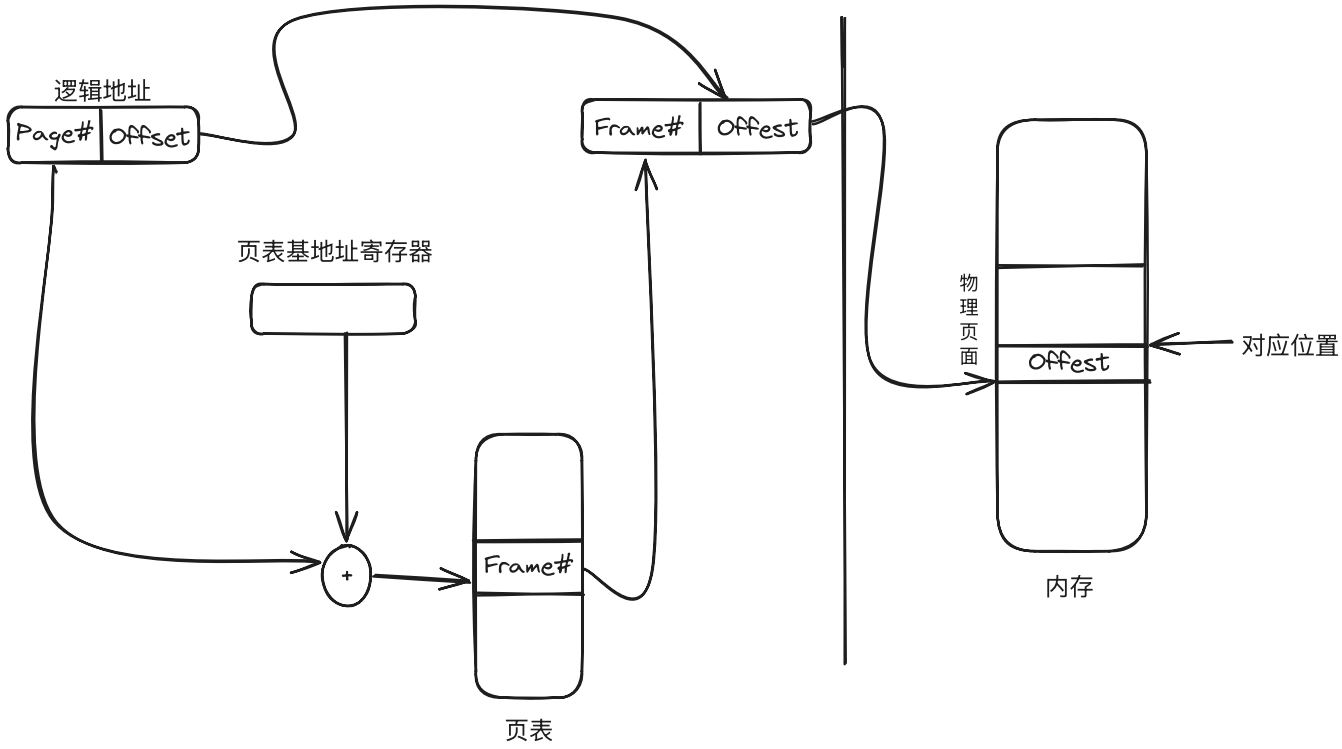
地址映射计算

假设在一个页式存储管理系统中，逻辑地址是16位，即需要16位2进制数编码来表示存储单元，每一个存储单元默认位一个字节，则存储大小为$2^{16}B = 64 * 2^{10}B = 64KB$，则逻辑地址的取值范围为0x0000～0xFFFF（$2^{16}B = (2^4)^4B = 16^4B$，每四位二进制数对应一个十六位进制数）
假设页面大小为4KB，则整个逻辑地址被划分为16个页面，对于逻辑页面0，其逻辑地址为0x0000~0x0FFF
$(0x0000 + 4KB = 0x0000 + 2^{12}B = 0x0000 + 16^3B = 0x0000 + 0x0FFF = 0x0FFF)$
则逻辑页面1的逻辑地址为0x1000～0x1FFF（以此类推）
那么对于每一个逻辑页面，其高4位代表逻辑页面号，低12位作为页内偏移地址
进一步的，若逻辑地址大小为M位，其页面大小为2^NB，则其高M-N为作为逻辑页面号，低N为作为页内偏移地址

若逻辑地址为16位，页面大小为1KB，对于逻辑地址0x2CBE，其高6位为逻辑页面号，低10位为页内偏移，即0x2CBE=0010 1100 1011 1110=001011 0010111110=00 1011｜00 1011 1110=0x0B｜0x0BE，即逻辑页面号为0x0B，页内偏移为0x0BE
若页面大小为2KB，则高5位为逻辑页面号，低11位为页内偏移，即0x2CBE=0010 1100 1011 1110=00101 10010111110=00 101｜100 1011 1110=0x05｜0x4BE，即逻辑页面号为0x05，页内偏移为0x0BE

若采用十进制表示逻辑地址，直接整除取余即可

根据逻辑页面号查找对应的物理页面号，需要页表，**页表保存在内存中**，其本质是一个数组，在内存中连续存放
为了访问页表，需要页表基地值寄存器和页表长度寄存器，分别指示页表在内存中的起始地址和大小，最终的计算过程为：



在映射过程中，操作系统主要负责数据内容维护，CPU负责地址映射的执行

05 优缺点

在每次访问内存时，需要访问两次内存，第一次访问页表，第二次才是真正访问数据或指令
解决办法：大多数程序在运行的一小段时间内，倾向于集中访问一小部分页面，因此增加一个快速查找硬件**TLB（Translation Lookaside Buffer）**，其用于存放最近一段时间内最常用的页表项，可以直接将逻辑页面号映射为对应的物理页面号，无需访问内存，因此缩短了页表的查询时间
优点：

1. 没有外碎片，且内碎片不会超过页面大小
2. 进程不必连续存放，从而提高了内存的利用率
3. 动态地址映射，在进程执行的过程中完成地址转换
   缺点：
4. 程序必须全部装入内存才能运行，若空闲页面较少，则无法执行（逻辑页面和物理页面要先一一对应）
5. 操作系统必须为每个进程维护一张页表，增大了系统开销

11-段式存储管理

01 基本原理

在逻辑地址空间中，对于程序的每一个逻辑单元设立一个完全独立的地址空间，称为“段”
好处在于，对于不同的逻辑单元可以设置不同的保护模式，便于存储保护；可以实现以段为单位的共享

02 具体实现

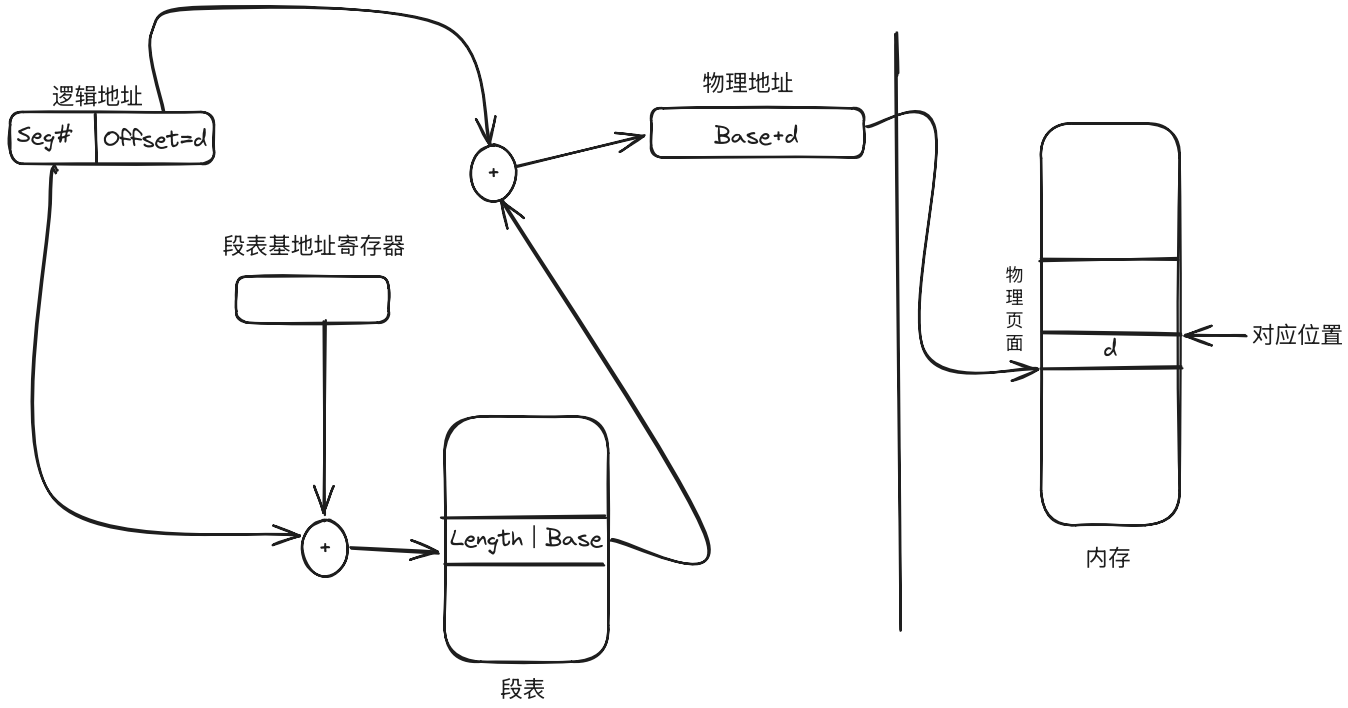
逻辑地址实现：段号和段内偏移组合在一起，高位部分表示段号，低位部分表示段内偏移；或者在指令中显式的给出段号和段内偏移

逻辑地址转为物理地址，通过段表实现。

段号	内存分区的起始地址	段长
……	……	……

段表直接给出内存分区的起始地址，而页表给出的是逻辑页号对应的物理页号
段的长度不固定，用新增的段长字段表示，页的长度是固定的

段表保存在内存中，通过**段表基地址寄存器**和**段表长度寄存器**来访问段表
映射过程具体为：



03 优缺点

优点：可以针对不同类型的段采取不同的保护方式；可以按段为单位来进行共享；不必连续存放，没有内碎片
缺点：必须全部装入内存才能够运行，而且存在外碎片

04 段式和页式储存管理的比较

1. 出发点不同：分页处于系统管理的需要；分段出于用户应用的需要；页式主要是为了减少碎片，提高内存使用效率；段式则是为了实现程序中各个逻辑单元的独立性，便于共享、保护和修改
2. 程序员关注角度不同：页式完全透明，操作系统维护页表，CPU访问页表，将逻辑地址映射为物理地址；段式的各个逻辑单元对程序员来说是知道的
3. 大小：段式大小不固定，页式大小固定
4. 逻辑地址：对于页式，其逻辑地址是一维的线性连续地址，同一程序的各个模块处于一个地址空间；段式则是二维的，包含段号和段内偏移

页式实际是对固定分区管理的一个扩展，不仅对内存分区，而且对程序分页，从而减少内碎片
段式实际是对可变分区的一个扩展，但不对一整个程序来分配一个内存空间，而是将逻辑空间按照功能划分为不同逻辑单元，对每个独立的逻辑单元按照可变分区的方法分配一个内存空间

12-虚拟存储技术

01 程序的局部性原理

时间局部性：数据和指令的两次相邻访问都集中在一个较短的时间内
空间局部性：正在访问的指令/数据和相邻的指令/数据都集中在一个较小的区域内

程序的局部性原理说明，在程序运行过程中，仅一小部分内容处于活跃状态，其余均在休眠
页式中的TLB和CPU中的高速缓存Cache都是基于程序的局部性原理

02 虚拟存储技术的原理

虚拟存储技术在页式货哦段式存储管理的基础上加以实现
其在装入程序时，只需将当前需要执行的部分页面或者段装入内存，即可让程序开始执行
如果需要执行的执行或者需要访问的数据不在内存中，称为缺页或者缺段，此时会产生一个硬件中断，调入相应的页或段，并调出暂时不使用的页面或者段
其特点是具有较大用户地址空间、部分交换、不连续性
相当于将内存作为磁盘的缓冲区

03 虚拟页式存储管理

Windows和Linux都采用这种技术
在页式的基础上增加了**请求调页**和**页面置换**

页表表项

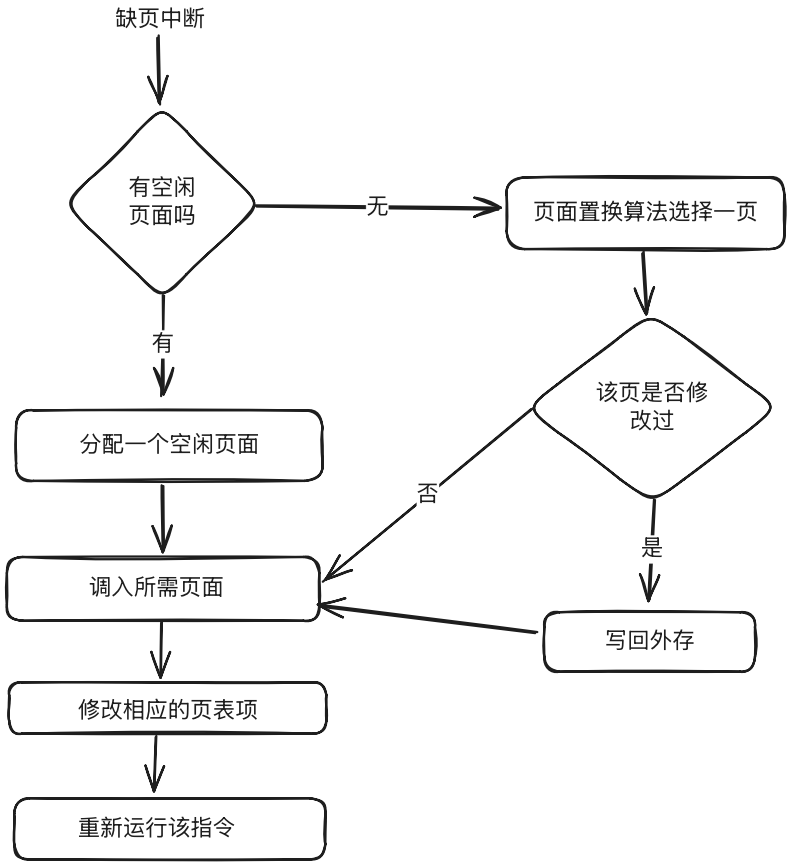
对于每个页表项，除了逻辑页号和对应的物理页号，还需要加入其他信息，包括驻留位（有效位）、保护位、修改位和访问位等

逻辑页号i	访问位	修改位	保护位	驻留位	物理页号

驻留位：1表示该页面现在在内存中，0则表示不在
保护位：表示运行对该页面进行何种类型的访问
修改位：表示该页面在内存中是否已经修改过（若未修改过，则可以直接将内容覆盖，提高交换效率，主要是数据页面，否则需要将其写回硬盘）
访问位：若被访问过则自动设置为1，主要用于页面置换算法

缺页中断处理流程

具体流程如图：



操作系统会创建和维护一个系统文件，用于存放各个进程被置换出去的页面，相当于一个临时的中转文件

04 页面置换算法

最优页面置换算法（OPT持Optimal Replacement）

对于每个保存在内存中的逻辑页面，计算下次访问前的等待时间，然后选择等待时间最长的页面作为被置换页面
理想算法，无法实现，但可以作为其他算法的性能评价依据

最近最久未使用算法（LRU持Least Recently Used）

选择在最近一段时间内最久未被使用的页面
其理论依据是程序的局部性原理
其需要记录各个页面使用时间的先后顺序，具体实现：系统维护一个**页面链表**；系统设置一个**活动页面栈**（使用则入栈，如栈中已经存在则抽出原有的，则栈底即为最久未使用）；使用**时间戳**
LRU的系统开销太大

最不常用算法（LFU持Least Frequently Used）

选择访问次数最少的页面置换，其考察访问的次数或频度
理论依据也是程序的局部性原理
需要设置一个访问计数器

先进先出算法（FIFO持First-In First-Out）

选择在内存中驻留时间最长的页面置换
可以使用队列实现（先进先出）

采用FIFO会发生**Belady现象**：出现分配的物理页面数增加，而缺页率反而提高的异常现象，其原因在于FIFO算法的置换特征与进程访问内存的动态特征相矛盾，也与置换算法的目标和要求不一致
FIFO置换出的页面不一定会是不访问的页面，甚至可能是经常要访问的页面

时钟页面置换算法（Clock）

将页面组织成环形链表，并需要访问位
考察指针指向的页面，若访问位为1，则将其设置为0，并考察下一位；若访问位为0，则选择该页面替换；若访问位为0，并且再次调用该页面，则访问位设置为1

LRU每发生一次页面访问时，都要动态调整各个页面之间的先后顺序，其性能较好，但系统开销较大；当内存中所有页面均未被访问过时，LRU就退化为FIFO；Clock是折中的算法

05 工作集模型

工作集模型描述一个程序在运行过程中的表现行为及其规律
使用二元函数$W(t,\Delta)$来表示，其中t指当前的执行时刻，Δ称之为工作集窗口（Working-set Window），是一个定长的页面访问窗口。W是在t时刻之前Δ窗口中所有页面组成的集合（即t时刻往前看定长步构成一个集合）
进程开始执行后，随着访问新页面逐步建立比较稳定的工作集

当内存访问的局部性区域的位置大致稳定时，工作集的大小也大致稳定
当局部性区域的位置发生改变时，工作集快速扩张和收缩过渡到下一个稳定值

驻留集，指当前时刻进程实际驻留在内存中的页面集合
工作集是进程在运行过程中固有的属性，而驻留集则取决于系统分配给进程的物理页面数和所采用的页面置换算法

当工作集是驻留集的子集时，进程的运行将非常顺利，直到工作集发生剧烈变动，发生缺页中断并从磁盘读入新的页面，从而过渡到另一个状态

当驻留集是工作集的子集时，进程将会造成很多的缺页中断，需要频繁地在内存和外存之间替换页面，从而使进程的运行速度非常慢，这种状态称为**抖动（Thrashing）**

（访问时间计算时需要注意，先访问TLB，未命中，再访问内存，如果此时发生缺页中断，在调入页面后，应该重新访问TLB，此时直接命中，无需访问内存中的页表，只需一次内存访问即可获得对应数据）

06 虚拟页式的设计问题

页面分配策略

关键在于确定**驻留集的大小**
固定分配策略：驻留集大小是固定不变的
可变分配策略：动态调整驻留集的大小，性能较好但增加系统开销

可变分配策略的具体实现——**缺页率算法**：
缺页率指的是缺页次数与内存访问次数之间的比率或者缺页平均时间的倒数，影响缺页率的因素主要有4点：

1. 页面置换算法的选择
2. 分配的物理页面数
3. 页面本身的大小
4. 程序的编制方法

对进程的缺页率设置一个上边界和下边界，超出上边界则分配物理页面防止抖动，低于下边界则减少物理页面数

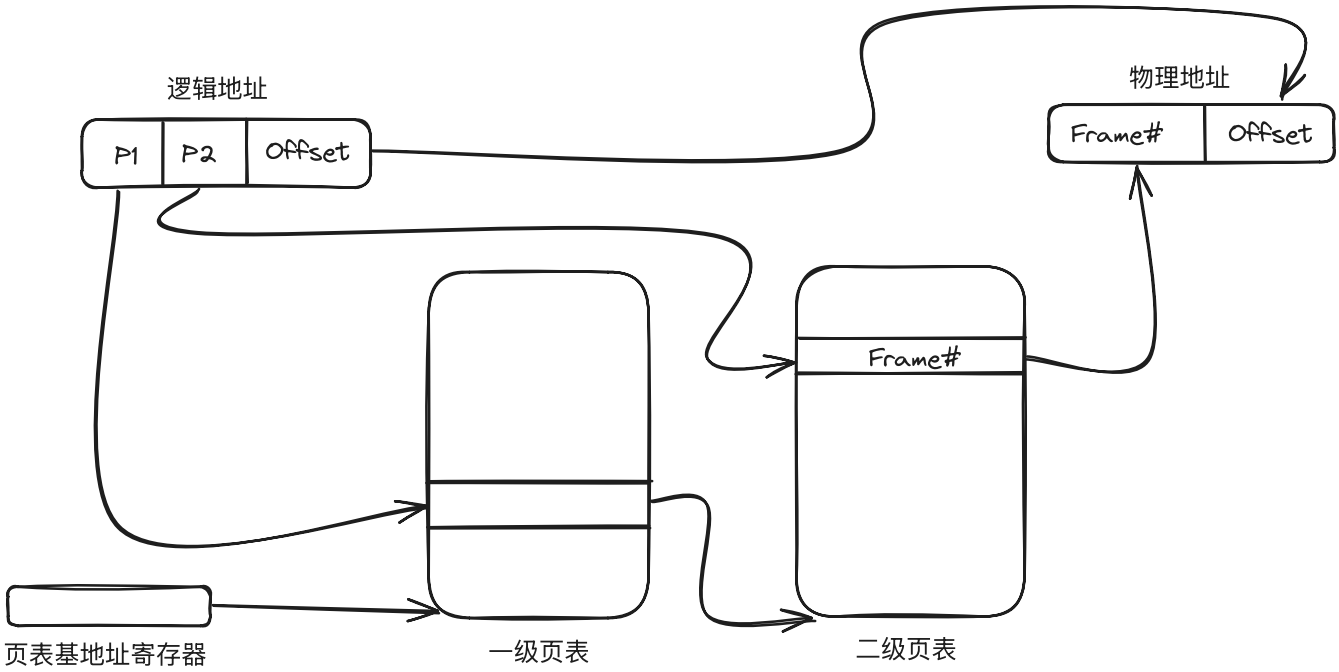
页面大小

从统计规律来看，内碎片的大小平均是半个页面
现代操作系统中，页面大小一般是4～64KB

多级页表

为了防止页表占据过多内存，设计了**多级页表**和**反置页表**

对于二级页表，若逻辑地址32位，页面大小4KB，则逻辑页面号20位，页内偏移12位
则可以将页表进一步分块，用一级页表存储二级页表位置
对于20位逻辑页面号：
10位的字段P1，用于指向第一级页表中所对应的页表项
10位的字段P2，用于指向第二级页表中所对应的页表项
具体映射过程：



页表基地值寄存器保存一级页表的起始地址，P1表明在一级页表中的偏移，一级页表对应表项中保存二级页表的起始地址，P2表明对应二级页表中的偏移，二级页表对于表项中保存对于的物理页面号，最终得到物理地址

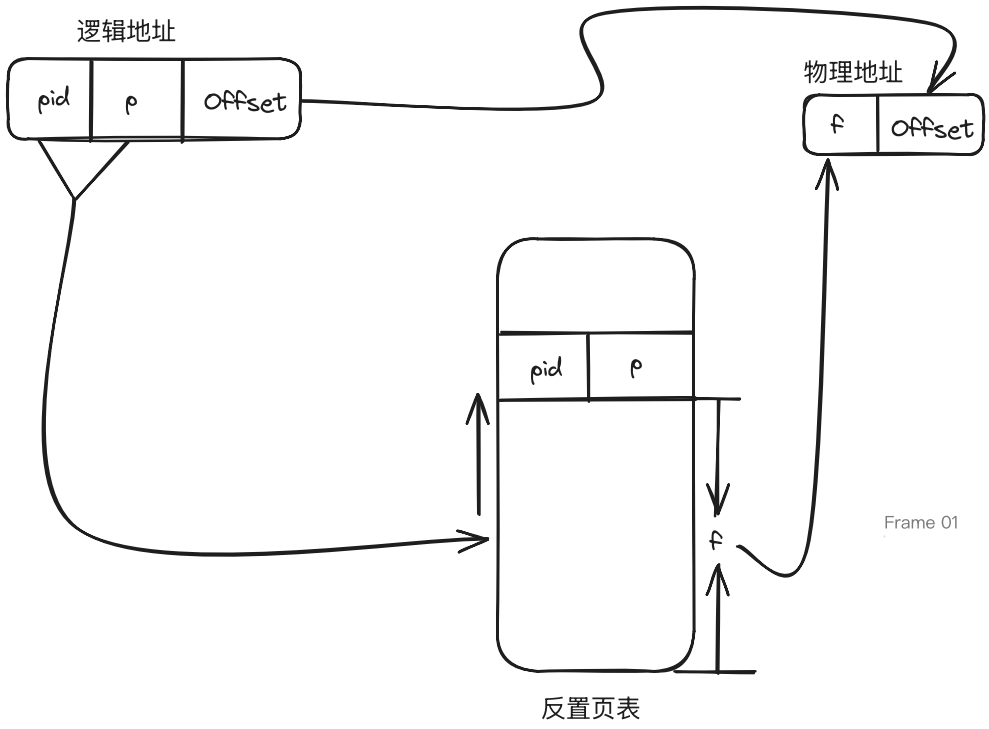
对于逻辑地址32位，大小为4GB，若页面大小为4KB，若采用单级页表项，共2^{20}个页表项
若其实际只使用12MB，实际只需要$3*2^{10}$个页面，则构造3个二级页表，每个页表有2^{10}个页表项，同时，对于一个一级页表，其也包含2^{10}个页表项（仅3个有用，但一级页表中保存的是对应二级页表的起始地址，而二级页表中保存的是对应的物理页面号）
可以对多余的页表项不在构造对应的二级页表，从而节省空间

若访问一级页表发生缺页中断，则首先需要创建一个二级页表，并将其起始地址装入一级页表对应页表项，一级页表就不会发生缺页中断；在访问到二级页表时，由于二级页表是空的，这时二级页表发生缺页中断，这时需要将对应的逻辑页面装入物理内存，并将对应物理页号装入二级页表对应表项

对于64位逻辑地址，12位表示页内偏移，剩余36位被划分为4个9位，作为每一级页表的索引

反置页表

只设置一张页表，用物理页号作为访问页表的索引，有多少个物理页面，就在页表中设置多少个页表项
其节省空间，但从逻辑页面号到物理页面号的转换变得更加复杂
具体过程为：



pid为进程标识符，p为逻辑页面号，根据pid和p在反置列表中依次查询，在第f个页表项找到对应的pid和p后，则说明该页保存在第f个物理页面中，从而计算出对应物理地址
即反置页表中第i个表项记录的是第i个物理页面对应的pid和p，这样确定物理页面需要逐个查找反置页表每一表项

13-IO硬件

01 设备分类

交互对象角度：人机交互设备（鼠标键盘）/计算机内部交互设备（磁盘传感器）/计算机之间通信设备（网卡）
交互方向角度：输入设备/输出设备/双向交互设备
数据组织方式角度：块设备/字符设备

02 设备控制器

设备控制器（Device Controller）将不同类型的硬件设备的内部实现细节封装，对外提供一个标准的、统一的接口
适配器与控制器：适配器一般是印刷电路卡的形式，可以插入主板的扩充槽；控制器一般是一组芯片，主要集成在主板或IO设备的内部

03 IO地址

每个设备控制器内部都有一些寄存器，用于和CPU进行通信，包括**控制寄存器、状态寄存器和数据寄存器**，除此之外，还有数据缓冲区
对设备控制器中的寄存器进行访问的方式有三种

IO独立编址

对每个寄存器分配唯一的一个I/O端口编号
优点：输入输出设备不会占用内存地址空间

内存映像编址

将设备控制器当中的每一个寄存器都映射为一个内存地址，进行统一编址
优点：编程方便，无需使用专用指令；无需专门的保护机制来防止用户执行IO操作（IO操作时一种特权指令，只能通过系统调用的方式）
缺点：不能对控制寄存器的内容进行Cache（CPU不访问，其值也可能发生变化）；每次都要判断访问的是内存还是IO

混合编址

对寄存器采用独立编址，对数据缓冲区采用内存映像编址

14-IO控制方式

IO控制就是在CPU上执行指令，与IO设备的设备控制器中的寄存器进行通信

01 程序循环检测方式

通过不断的检测IO设备当前的状态，来控制IO操作的完成，即繁忙等待或轮询方式
IO控制由CPU进行，IO操作由设备本身进行
缺点是CPU和IO设备的速度不匹配，浪费大量的CPU时间

02 中断驱动方式

中断技术依赖于硬件支持

> **中断类型**
>
> 所谓中断，是指由于某件事件的发生，改变了CPU上执行的指令的顺序，对应于CPU芯片内部或者外部的硬件电路所产生的电信号
> 中断可以分为**同步中断**和**异步中断**：
> 同步中断，也称为“异常”——1.CPU检测到的异常，包括Fault、Trap、Abort 2.程序主动设

置的异常，也称为软中断，用于实现系统调用服务
异步中断，简称为“中断”——由CPU以外的硬件设备发出的中断 1.可屏蔽中断，即IO中断 2.不可屏蔽中断，由掉电等硬件故障引起的硬件中断

中断控制器负责管理系统中的IO中断，只有其可以向CPU发送中断请求
发送请求时，会将编号放在地址总线上表明是哪个设备发送的中断；同时发送一个中断信号作为索引访问中断向量表（存放每个中断处理程序的起始地址）
在对应的中断处理程序开始运行后，会向中断控制器发送确认信号
（中断的处理需要额外的系统开销）

03 直接内存访问方式

内存直接访问（Direct Memory Access DMA）依赖于硬件支持
DMA控制器能独立于CPU直接访问系统总线，可以代替CPU指挥IO设备与内存之间的数据传输，从而节省CPU时间
DMA包括一个内存地址寄存器、一个字节计数器和多个控制寄存器，其指明了IO设备的端口地址、数据传送方向、传送单位和每一次传送的字节数
具体过程为：

1. 发生一次IO请求
2. CPU对DMA进行编程，指明将什么数据传送至内存的什么地方以及传送的字节总数
3. CPU启动IO
4. 磁盘控制器从磁盘驱动器中读取数据，保存在内部缓冲区中，并验证数据正确性（此期间CPU运行别的进程）
5. 磁盘完成读取后，向DMA启动数据传送
6. DMA通过数据总线向磁盘控制器发出读操作的请求信号，并将要写入的内存地址打在总线上
7. 磁盘控制器取出一字节并写入内存
8. 磁盘控制器向DMA发送确认信号，并继续写入直至写完
9. 数据全部传送完毕后，DMA向CPU发送中断，中断处理程序开始运行
   DMA在这个过程中代替CPU发送指令

15-IO软件

01 层次结构

为实现模块化管理，将各种设备管理软件组织成一系列层次，一般分为四层：

1. 用户空间的IO软件
2. 设备独立的系统软件
3. 设备驱动程序

4. 中断处理程序

1.中断处理程序

中断处理程序与设备驱动程序相互合作，其同步方式可以采用进程间的通信方式（信号量、PV原语）

2.设备驱动程序

设备驱动程序直接对设备控制器中的寄存器进行操作，其一般位于系统的内核空间之中
其负责把输入的抽象参数转化为控制设备所需要的具体参数

3.设备独立的系统软件

也称为内核IO子系统，其给上层应用统一接口、与设备驱动程序统一接口、提供与设备无关的数据块大小及缓冲技术等
应用程序与操作系统的接口：

1. 设备独立性：无需事先指定指定特定的设备类型
2. 统一命名：所有文件和设备都采用相同的命名方式（如路径名）
3. 阻塞与非阻塞IO：提供两类不同的API（进程调用IO操作后阻塞还是立即返回）

操作系统与IO设备的接口：

为实现设备独立性，操作系统将设备划分为三大类：块设备、字符设备和网络设备

接口映射：

操作系统负责将这两种接口映射成API函数

设备无关的数据块大小：

为实现设备独立性，操作系统可以掩盖不同的数据大小，并提供统一的数据块大小

缓冲技术：

内存中可以保存部分数据块，从而减少对磁盘的访问次数，提高访问速度
缓冲技术的实质是以空间换时间

4.用户空间的IO软件

主要分为两类：

1. 库函数：与用户程序进行链接的库函数

2. SPOOLing技术：Simultaneous Peripheral Operation On Line 假脱机技术，可以将独占设备转化为具有共享特征的虚拟设备
   SPOOLing技术的优点：
3. 高速的虚拟IO操作：其本质是在两个进程之间的一种通信，把数据从一个进程交给另一个进程，这种交换是在内存中进行的
4. 实现对独占设备的共享
   （后台打印程序就是一个SPOOLing进程）

16-磁盘

01 磁盘的硬件

磁盘是机械硬盘，由机械结构驱动
旋转轴上包含多个磁盘，磁头被固定在磁头臂上，可以沿磁盘半径方向移动
磁盘上的圆环形区域称为磁道（Track）
半径相同的所有磁道组成一个柱面（Cylinder）
对每一个磁道，被均匀的划分为一个个扇区（Sector）
读取顺序：

1. 移动磁头找到柱面（**柱面号**）
2. 根据磁道号选择磁头（**磁道号**）
3. 旋转找到对应的扇区（**扇区号**）
   磁盘访问以扇区为单位，即使读写一个字节，也要将整个扇区读入和写入

02 磁盘格式化

每一个扇区由三部分组成：

1. 相位编码：向硬件表明这是一个新扇区的开始（还包含柱面号、扇面号和扇区大小等信息）
2. 数据区：一般设定为512B
3. 纠错码：冗余信息用于纠错
   格式化三个步骤：
   低级格式化：分出扇区和磁道
   分区：把整个硬盘划分为若干个逻辑分区，每个逻辑分区都可以看作一个独立的硬盘
   高级格式化：Format 相应的逻辑分区上会生成一个引导块、空闲存储管理的数据结构、根目录和一个空白的文件系统

03 磁盘调度算法

磁盘调度需要的时间由三部分构成：

1. 柱面定位时间：机械运动找到正确的柱面
2. 旋转延迟时间：盘片旋转至正确的扇区
3. 数据传送时间：写入或读出数据
   盘片的旋转和数据的读取是同步进行的
   提高磁盘访问速度的方法：合理组织磁盘数据的存储位置/磁盘调度

先来先服务FCFS

按照访问请求到达的顺序来依次执行

最短定位时间优先算法（SSTF Shortest Seek Time First）

选择从当前的磁头位置出发，移动距离最短的那个访问请求去执行

电梯算法（Elevator Algorithm）/扫描算法（Scan）

先沿着一个方向移动，并依次执行这条路径上的所有访问请求，直至没有请求，然后再换一个方向继续进行
对于任意一组访问请求，磁头移动的总距离上界为柱面总数的两倍

04 出错处理

数据密度指单位长度的磁介质上能够存放的数据的位数
对于磁盘中的坏扇区有两种处理策略：

1. 由设备控制器处理，对整个磁盘进行测试，用列表记录坏扇区，并用备用扇区来替代
2. 由操作系统处理，操作系统对整个磁盘进行测试，获得坏扇区的列表，并在此基础上构造一个重映射表

17-固态硬盘

01 闪存

闪存（Flash Memory）是一种存储器，其基本单元电路是双层悬空栅MOS管，带电表示存入0，不带电表示存入1
Flash分为NOR Flash和NAND Flash
NOR Flash的设计目标是替代只读存储器ROM，一般用于存储不经常更新的程序代码，其速度快，提供完全的地址和数据总线，可以随机访问任何一个存储单元
NAND Flash的设计目标是匹敌磁介质的存储设备，其不能进行随机访问，所有操作必须以块为单位来进行

02 NAND Flash

NAND Flash的最小访问单位是Page，若干个Page组成一个Block（类似于磁盘中的数据块）
对于每一Page，其由两部分组成，一部分是有效容量，另一部分用来存放附加的校验信息
NAND Flash读取一页所需要的时间与页号和之前的访问请求无关

NAND的写操作：对于页中的每一个0/1，只能单独把1写成0，如果要把0写成1，要将**整块**全部写成1，再把相应的1写成0（擦除操作以块Block为单位）（在擦除前要用读操作先保存起来）

03 U盘

U盘是USB闪存驱动器，主要由控制存储电路和快闪存储器构成

04 SSD

SSD（Solid State Drive）即固态硬盘
SSD的写入速度比较慢，擦除次数有限
SSD控制器的主要功能是承上启下，实现数据的中转，将硬盘内部的闪存与外部的主机接口连接起来
SSD内部一般会有一块DRAM芯片用作缓存
将经常要使用的、以读为主的以及运行速度较慢的内容保存在固态硬盘以提高访问速度
将需要频繁更新的文件保存在机械硬盘，因为SSD擦除次数有限

18-文件

01 基本概念

将各种各样的信息组织成文件的形式，用文件作为信息的存储和访问单位
文件系统主要包括两个功能：目录功能和存储服务
目录功能负责将文件名映射为内部使用的标识符

文件是一种抽象机制

文件命名

文件名一般由文件名和扩展名组成
Windows中，会将文件的扩展名与某个应用程序关联起来

文件结构

普遍采用无结构方式，将文件看成一序列无结构的字节流（除可执行文件）

文件分类

1. 普通文件：ASCII文件/二进制文件
2. 目录文件：专用的特殊文件

文件属性

如保护信息、文件长度、只读标志位、系统标志位等等

02 文件使用

对文件的使用主要分为两种类型的操作：访问文件的**属性**/访问文件的**内容**

19-目录

01 目录结构

一级目录结构是一张线性表
二级目录结构分为根目录和用户子目录
多级目录结构除根目录外，每个目录下可以有目录也可以有文件
根目录的管理信息存放在磁盘空间中专门预留的一块区域

多级目录下指定文件或目录通过**相对路径名和绝对路径名**

02 文件系统的实现

文件在硬盘上的存放时以块Block为单位的，对于每一个文件，逻辑地址空间是连续的，如果最后一个逻辑块不够整块，那么也要占据一个完整的物理块

磁盘的扇区0称为主引导记录（Master Boot Record，MBR）用于启动计算机
MBR末尾有一个分区表，记录每一个分区的起始扇区和大小
操作系统保存在活动扇区中

对于一个具体的分区，其包括：

1. 引导块：其负责把该分区中存放的操作系统程序装入内存
2. 分区控制块：保存文件系统重要参数——标识符（表明文件系统类型）；块大小（每块物理块大小）；物理块个数
3. 空闲空间管理：空闲块的个数，空闲块的链表指针、位图等
4. I结点：记录每一个文件的管理信息
5. 根目录：包含其下所有文件各种说明信息
6. 普通的目录和文件：包含的主要内容

计算机加电后，主板上的BIOS启动执行，将MBR中的引导程序装入内存中执行，并查询硬盘分区表，并将活动分区中的引导块装入内存运行

03 文件的实现

文件由**元数据**和**一组数据块**组成

文件控制块（File Control Block持FCB）

FCB保存在硬盘上，主要包括文件的属性和文件内容在磁盘上的存储位置

文件物理结构

连续结构

类似于可变分区存储方案，将各逻辑块按照顺序存放在若干个连续的物理块之中
优点：易于实现，可以顺序读取速度快
缺点：会形成已占物理块和空闲物理块交错的情形，可以采用**存储紧缩技术**，把所有文件向一个方向移动；文件大小不能动态增长

链表结构

每个物理块中利用若干个字节作为指针指向下一个物理块
优点：其不存在外碎片问题，文件大小可以动态变化
缺点：只能顺序访问，不能随机访问；数据存储空间不再是2的整数次幂（指针占据）

带有文件分配表的链表结构

类似于页式存储管理中的反置页表
将每个物理块中的链表指针抽取出来，单独组成一个表哥，即文件分配表（FAT File Allocation Table），其使用时装入内存
其使用时必须把整个FAT保存在内存中
FCB记录文件所在的第一个物理块编号X1，FAT中的第X1项纪录第二个物理块编号X2，直至某项记录为-1
某个物理块未被使用，FAT中相应的表格项为空

索引结构

把文件中每个逻辑块对应的物理块编号直接记录在文件的FCB中，称为i结点（inode index-node）
只需将正在被使用的文件的索引结点装入内存即可
当需要访问文件中的第k个逻辑块时，首先通过该文件的目录项，查询其索引结点的存放位置，将该结点装入内存，再以k为下表查询里面的地址映射表，将第k个表项取出即为对应的物理块号

当文件过大时，地址映射表不够，可以引入间接索引，在地址映射表中存放某个地址，用于指向另一个地址映射表

04 目录的实现

目录项的内容

1. 直接法：目录项＝ 文件名+FCB（Windows）
2. 间接法：统一保存FCB，目录项＝ 文件名+FCB地址（UNIX）

长文件名问题

1. 目录项中将文件名长度固定为255个字符
2. 目录项长度可变，分为目录项的长度、文件的属性信息和文件名（可变）
3. 目录项长度固定，将长度可变的文件名统一放在目录文件的末尾

目录的搜索方法

1. 线性搜索
2. Hash表：另外构造一个Hash表，假设目录长度为N，Hash函数将文件名映射为0～N-1个值，在响应的Hash表项中，存放该文件所对应的目录项地址

05 系统调用实现

访问文件属性

实际是访问文件目录项

访问文件内容

外存：目录结构；文件控制块
内存：系统内打开文件表；进程内打开文件表（保存在PCB中）

打开文件

1. 文件系统调用目录结构方面的服务功能，查找到该文件所在的目录项
2. 根据文件的打开方式和共享说明等检查这次访问的合法性
3. 查找系统内打开文件表，检查文件是否已被其他进程打开（没有则FCB要读入内存）
4. 在进程打开文件表项中记录
5. 返回指针给系统调用，之后的文件操作通过该指针完成

关闭文件

1. 在进程打开文件表中，将该文件对应的表项删除
2. 系统打开文件表中，将该文件对应的表项共享计数值减1
3. 如果文件信息更新，则需要协会硬盘的目录结构，确定所有进程完成后将系统打开文件表中相应表项删除

读文件

通过系统调用函数实现，包含缓冲区

06 空闲空间管理

位图法

把磁盘每个物理块用一个位表示，系统维护一个空闲空间的列表
从位图第一个字出发，找到第一个不为0的字，磁盘的第一个空闲物理块编号为：值为0的字的个数✖ 字长+首个1的字内偏移量
位图本身存放在磁盘上

链表法

用链表表示空闲空间列表

索引法

用一些专门物理块记录空闲物理块编号，这些物理块用指针相连