操作系统 Operating System

Wei Chow

目录

[1 导论 Introduce 2](#_Toc154864845)

[2 进程和线程 Process&Thread 6](#_Toc154864846)

[3 CPU 调度 12](#_Toc154864847)

[4 同步 Sychronized 15](#_Toc154864848)

[5 死锁 DeadLock 20](#_Toc154864849)

[6 主存 Main Memory 24](#_Toc154864850)

[7 虚拟内存 Virtual Memory 31](#_Toc154864851)

[8 文件系统 File System 36](#_Toc154864852)

[9 大容量存储系统 Mass-Storage Systems 41](#_Toc154864853)

[10 I/O 系统 43](#_Toc154864854)

[11 Linux 操作系统与内核 44](#_Toc154864855)

[12 一些题目 52](#_Toc154864856)

**写在前面**

本笔记主要在原pdf的基础上，按照ws老师的ppt进行更改（基本涵盖了ws老师的所有内容）

季江民的复习课可能会透题

Final Exam = 50％

**3张A4可以打印**

季江民班的平时作业提供大量选择题，其题目部分来自于往年卷，又因为每年的出题老师都会参照往年卷，故考试中出现一些季江民班布置的选择题，在考前，其他班的学生可以刷下一季江民班的平时作

2023.1 只考了一些Linux常用命令，和一些Linux"常识"，没有考实验，s最后大题也考了王道原题（内存管理，文件系统各考了一道王道原题）。只有2分，不重要

每年都有一些题型取自于王道的大题, 复习的时候务必关注。王道做上几道大题就行，os选择应该和王道没啥重合

2023

50个选择，70分

3个简单题，一个10分(同步算法，内存里计算为主的)

（生产者消费者）

（不考monitor）

导论 Introduce

* 1. 什么是操作系统**?**

计算机中一直运行着的程序就是操作系统的内核 **(kernel)**

bootstrap program位于内核用于把bootstrap导入内存然后启动(初始化代码)，放在固定位置。启动时最先运行的程序就是 bootstrap program

**DMA**

Direct memory access (DMA) is a feature of computer systems that allowscertain hardware subsystems to access main system memory (random-access memory), independent of the central processing unit (CPU) .

DMA的传输模式的以下三种：突发模式(Burst mode)、循环挪用(Cycle stealing mode)模式、透明模式.DMA时CPU不能访问内存。突发模式就是在一个连续的时间序列内完成一块完整数据的传输。当cpu将系统总线(AMBA总线)控制权交给DMA控制器，cpu输入空闲状态，直到DMAC将数据块所有数据传输完成并将总线控制权交还给cpu。这个模式叫做块传输模式，也叫做突发模式。

并发是同一段时间内发生，并行是同一时刻

OS structre

* + - * batch processing 批处理系统，特点是操作系统在处理一个事务的时候不能和用户发生交互，**交互性差**
      * time-sharing（multi-tasking） 分时系统，可以多个用户共享一台计算机，操作系统轮流为多个用户服务

其中 CPU 频繁切换作业，以便用户可以在每个作业运行时与其进行交互，从而创建交互式计算（交互性

* + - * multi-programming 多道程序设计系统，通过安排作业（编码与数据）使得 CPU 总有一个执行作业， 从而提高 CPU 利用率，可能在内存上保留多个任务的信息

操作系统的发展历程大致上是：no software–resident monitors–multi-programming–multi-tasking

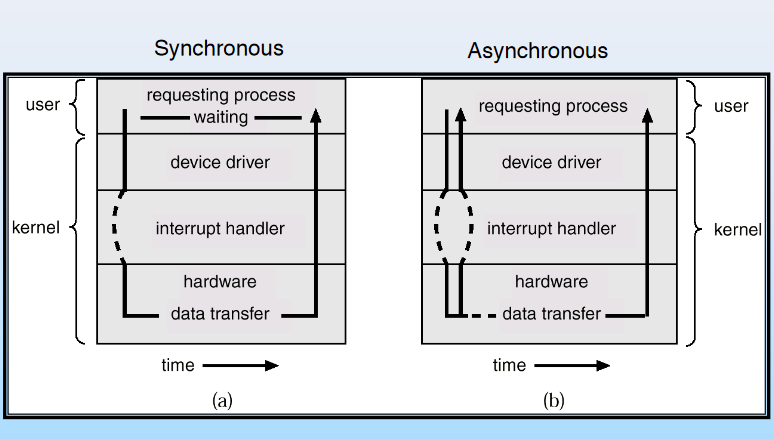
* 1. 中断 Interrupt

**Definition 1.2** 中断是指系统发生某个同步*/*异步的事件之后，处理机暂停正在执行的程序，转而执行该事件的处理程序的过程。中断是通过interrupt vector来实现的，中断向量中会记录所有服务例程的地址。中断的分类如下：

* 外部中断：分为 I/O 中断和时钟中断
* 内部中断 (异常 exception)：由 CPU 产生，包括系统调用、缺页异常、断点指令等等

**Theorem 1.1** 操作系统是中断驱动的

**Theorem 1.2** 中断处理会保存程序状态寄存器中的内容，但是子程序调用不会保存。



* 1. 双模态操作 dual-mode operation
* 用户态(user mode)：只能访问为其分配的寄存器和存储空间，只能执行普通指令，用户程序和操作系统以外的服务程序都运行在用户态中，使用用户栈
* 内核态(kernel mode)：可以访问所有的系统资源，执行特权指令，可以直接操作和管理硬件设备。操作系统的内核程序运行的时候处于内核态，使用内核栈

**Theorem 1.3** 三种从用户态切换到内核态的方式：系统调用、异常、中断

**Theorem 1.4** 关于 *mode:* 不可能发生在用户态的事件是：**进程切换**，因为需要调度处理器和系统资源。**缺页**会导致用户态转换到内核态。而**系统调用和中断**既可以发生在用户态又可以发生在内核态。

* 1. **Linux** 系统概述

Linux 操作系统是一种类Unix 的操作系统，遵守 GNU 的GPL/LGPL/AGPL 协议，是一个可以供多人使用的抢占式多任务操作系统,Linux 的内核是单一体系结构的，使用了一种全新的内核模块机制，用户可以根据需要，在不需要重新编译内核的情况下动态地装入/移除内核的模块

内核模块在内核态运行，全称是 Loadable Kernel Module(动态加载模块)，实际上是一种目标对象文件，在没有进行链接的时候不能独立运行，内核的代码在运行的时候可以连接到系统中，这就是一种Linux 内核中的动态扩充机制

加载内核模块对应的指令是 **insmod**，查看内核模块的指令是 lsmod，卸载内核模块的指令时 **rmmod**， 内核模块运行的时候生成的结果会存放在**/var/log/kern.log** 文件中

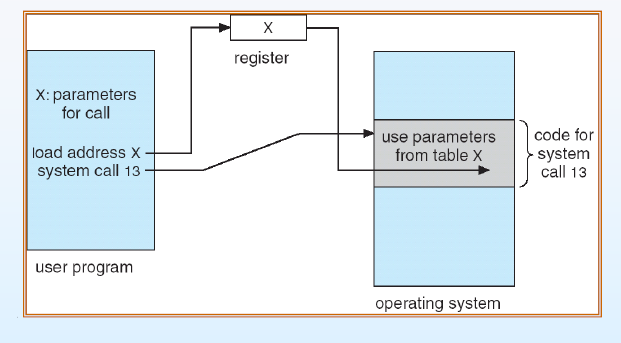
* 1. 操作系统服务
* 用户接口：操作系统提供的接口有分为命令接口和程序接口两类，命令接口包括命令行 CLI 和图形用户界面 GUI，而程序接口有 system call
* 程序执行：将程序装载到内存中执行
* I/O 操作：操作文件和 I/O 设备进行输入输出
* 进程之间的信息交流
* 错误检测：检测 CPU，内存，I/O 设备出现的错误
* 系统资源的分配
* 用户操作的记录：形成日志
  1. 系统调用 system call

**Definition 1.3** 系统调用是操作系统提供服务的**程序接口 Programming interface**

* 系统调用是进程和 OS 内核的程序接口，使用**高级语言来编写 （Application Program Interface (API) rather than direct system call use）**
* 操作系统的 API 封装了一系列的系统调用，每个系统调用对应了一个封装例程 (wrapper routine)
* 系统调用是在**内核**完成的，用户态的函数是在编程语言的**库函数**中实现的。像 C 语言中的printf，fopen， malloc 都是 C 语言的库函数
* 执行系统调用的几个过程：执行 trap 指令，传递系统调用操作，执行相关操作，返回用户态

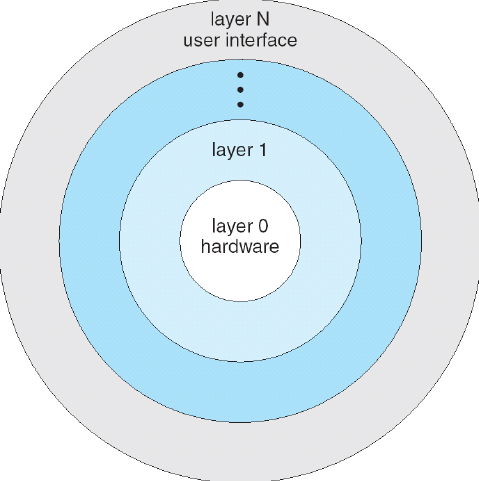
**Theorem 1.5** 系统调用中**三种参数传递的方式***:* 通过**寄存器**传递参数，通过**栈**传递，通过**内存中的 block 和**

**table**等数据结构存储参数，并在系统调用中把块地址作为参数放在寄存器中（基于栈和block的方法不限制参数长度）， 下图是通过table传播的一个例子



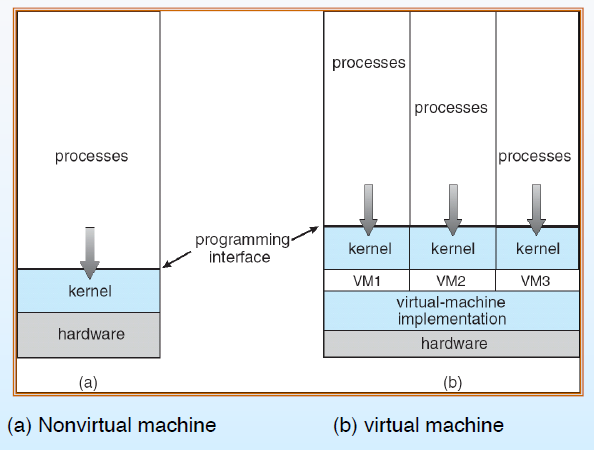
* 1. 内核结构

操作系统分成了若干层次，最外层是用户接口，最内层是硬件



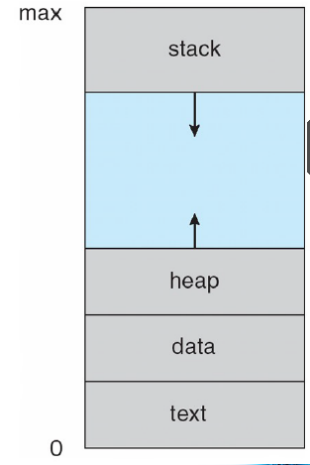
* 单内核结构 (宏内核结构 Monolithic Kernels)：内核的全部代码打包在一个文件中，优点是**效率很高**， 缺点是维护和修改内核非常困难，容易出 bug
* 微内核接口(Micro Kernels)：内核的最基本功能由中央内核实现，其他的功能都委托给独立的进程来实现，这些进程和内核通过一定的接口来通信，这些即内核的模块。更安全，可扩展，分布式计算

虚拟机将分层方法发挥到了极致。它将硬件和操作系统内核视为所有硬件



进程和线程 Process&Thread

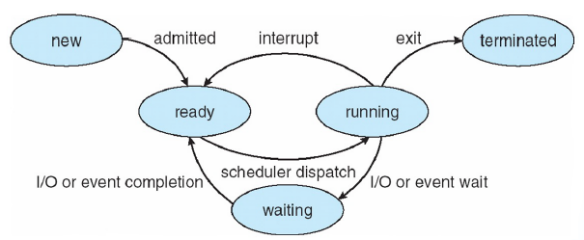
* 1. 进程(process, job)

进程就是正在执行中的程序，放在外存的程序就不是进程。进程是计算机工作的基本单位，进程包含了一系列的指令和资源，是一个抽象实体，在执行任务的时候，要分配和释放各类资源

进程包括以下内容:

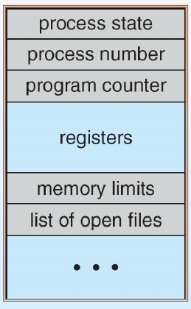
* + - * 程序代码 text section
      * 数据区data section，存储全局变量
      * 堆heap(构造为完全二叉树，父亲需要都比儿子大或者小)，存放动态分配的变量
      * 栈区satck，存放临时变量

进程的state

* + - * new 新建，进程刚被创建出来
      * running 运行、执行，进程正在被执行的状态
      * ready 就绪，进程准备好被分配给一个 CPU 执行
      * waiting 等待，等待一些事件结束之后再执行，也叫做阻塞状态
      * terminated 中止，执行结束

**Theorem 2.1** 可以引起进程状态变化的操作包括：程序中的操作，比如系统调用；操作系统的操作，比如调度的决策；外部操作，比如中断

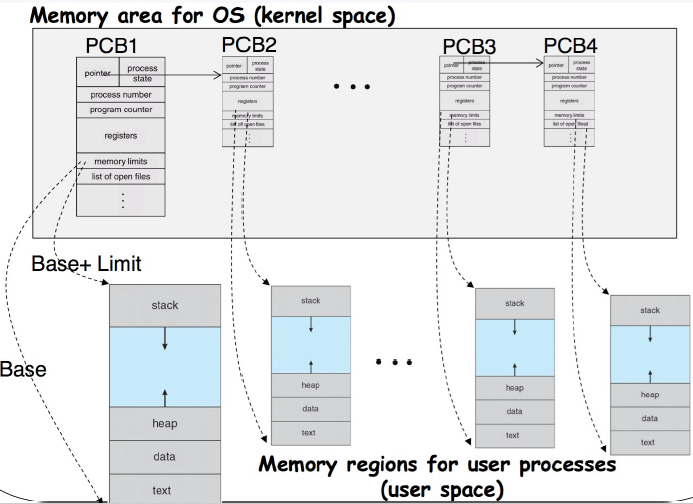
自己主动需要某些资源，比如输入输出的数据或者某个事件的时候，进程就会变成waiting 状态，而如果是因为 CPU 等外部因素让正在 running 的进程停止，比如时间片轮转的时候时间到了，或者优先级不是最高的时候，进程就会被调度成 ready 状态。而 terminated 则是因为发生意外导致了进程终止或者进程自己正常运行之后结束。

* 1. 进程控制块 Process Control Block (PCB)

在操作系统中，进程用进程控制块 (process control block) 这样的数据结构来表示，包含进程的状态(run, etc)，programm counter（下一条指令的地址）， 寄存器信息，调度信息，内存管理信息， 统计信息，I/O状态信息

在Linux 内核中，进程的信息用一个 C 语言的结构体 **task\_struct** 表示，定义在头文件

<linux/sched.h>中，一系列进程构成一个双向链表



* + 1. 进程的调度 Process Scheduling （进程调度不需要额外的硬件）

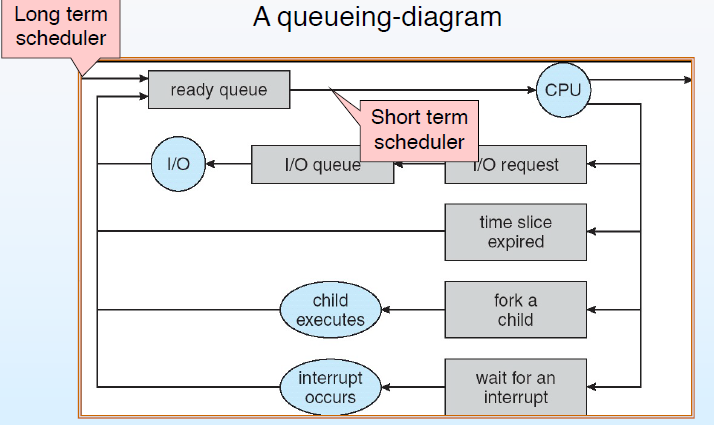
程序运行再main memory中

进程的调度需要在 ready(**ready queen** in main memory, ready)，running **(job quene**)和 waiting(**device quene**,等待I/O等设备)

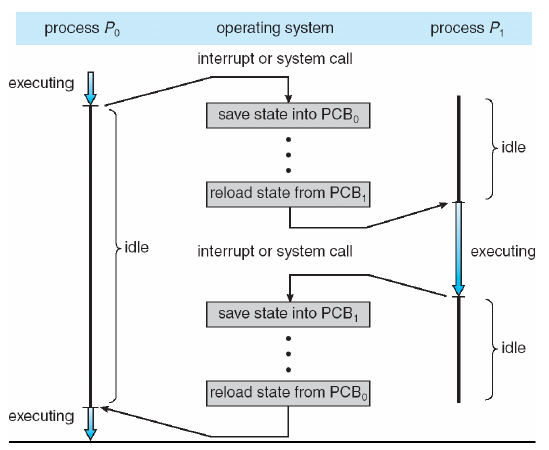
三种状态之间切换，进程的调度可以分为

* + - * **长程调度(job scheduler)**：选择应将哪些进程放入内存(ready quene),不频繁。决定degree of multiprogramming
      * **短程调度(CPU scheduler)**：CPU 可以选择下一个执行的进程，非常频繁

一种进程分类的方式是将进程根据主要耗时的操作类型来分，即分为 **I/O 型进程**和(花大量时间在I/O上，而不是计算上) **CPU 型进程**



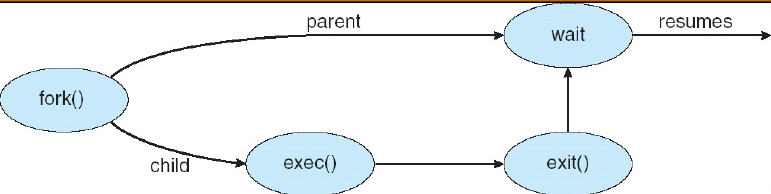
**Definition 2.1 上下文切换Context Switch**：当 *CPU* 切换到另一个进程的时候，系统必须要保留上一个进程的状态，同时又要载入的进程状态，这一过程被称为上下文切换，在 PCB 中有专门存储上下文信息的地方。



* + 1. 进程的创建与 **fork** 系统调用

父进程可以创建子进程，并可以形成一棵**进程树**，可以用命令 pstree 查看，每个进程都有唯一的标识号 **pid(process identifier)**

父子的resource的关系可以是全部共享，部分共享，不共享



**Definition 2.2** *fork()* 函数是 *Linux* 下创建进程的系统调用，将当前进程分为父进程和子进程。

fork 被调用之后系统先给新的进程分配资源，然后把原来的进程的数字复制到新的进程中，函数的返回值是一个pid，调用 fork 之后两个进程的除了返回值pid 不同以外没有任何不同，其中**父进程返回的是子进程的 pid**，**子进程返回的是 0**

*fork* 系统调用可能只会复制当前的调用它的线程，也可能会复制进程中所有的线程。

* + 1. 进程的替换与 **exec** 系统调用

系统调用 exec 是以新的进程去代替原来的进程，但进程的 pid 保持不变。因此，exec 系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。

*exec* 系统调用可能会替换掉整个进程，但是不会只替换掉调用它的子线程里的内容。

* + 1. 进程的终止与 **wait** 系统调用

**Theorem 2.4** 进程的终止分为：正常结束，异常结束和外界干预，可以用 *exit(0)* 来退出进程

系统调用 **wait()**

* + - * 通过wait 系统调用可以让父进程等待子进程结束之后再执行，wait 调用会返回进程的状态信息和结束进程的 pid
      * 父进程一旦调用了 wait 就立即阻塞自己，由 wait 函数分析是否当前进程的某个子进程已经退出了
      * 如果找到了退出的子进程就会将其销毁，如果没有找到已经退出的子进程，wait 就会一直阻塞父进程

Cascade termination 父进程terminates导致子进程全部abort

* + 1. 进程间通信

**Definition 2.5** 进程根据是否可以和其他进程进行通信和交互分为Independent独立进程和Cooperating合作进程。而进程间的通信又分为直接通信(link自动建立，有且仅有一个link)和间接通信（使用相同mailbox的时候建立link，一个link可能练剑多个进程，每个pair进程会分享多个link）。

|  |  |
| --- | --- |
|  |  |

**Definition 2.6** 共享内存模型shared memory和消息模型message passing是两种常见的Interprocess Communication (IPC)。共享内存顾名思义就是通过共享的内存来进行数据的交互，而消息模型则是通过消息队列的数据结构来缓存进程之间的通信内容

**Theorem 2.5** 常见的通信机制有：信号 *signal*，共享内存，管道 *pipe*，消息 *message* 和套接字 *socket*，*Linux*

中还有文件锁和 POSIX 线程，互斥锁 mutex 等等。

消息的传送又可以具体分为阻塞的和非阻塞的

阻塞的消息传递(blocking, synchronous)就是发送方进程需要阻塞直到消息被接受，而接收方需要阻塞到有消息收到，

非阻塞(non-blocking, asynchronous)就是一种异步的通信模式。当发送方和接受方都是阻塞的时候，就可以认为二者达到了一个 rendezvous。发送方可以一直发，接收方收到valid message or null

常用的Communication in Client-Server Systems的3种方法：Sockets，remote Procedure Calls, Remote Method Invocation (Java)

* 1. 线程

**Definition 2.7** 进程是操作系统中的资源的拥有单位和调度单位，拥有虚拟的地址空间，控制了一些资源， 有状态，优先级和调度。进程是资源拥有的单位，而调度的单位则称为**线程 *(thread)***，线程是进程中的一个执行单元或者可以调度的实体。

线程的特点

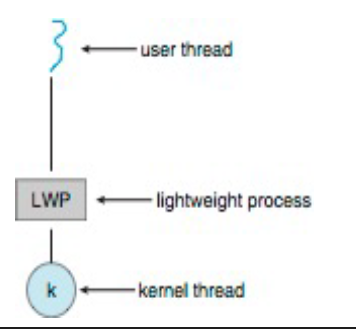
* + - * 有执行状态，不运行的时候会保存上下文
      * 每个进程有一个执行栈，有局部变量的静态存储区
      * **线程可以取用所在进程的资源**，但是不拥有系统资源
      * 可以创建、撤销另一个进程
      * 一个进程中的多个线程可以并发执行，相比于进程系统的开销小，切换快

线程和进程的比较：

1. 进程通常是独立的，而**线程则是进程的子集（同一process的不同thread分享code, data, files）**
2. 进程比线程携带更多的状态信息、 而进程中的多个线程共享进程状态以及内存和其他资源
3. 进程有独立的地址空间，而线程共享其地址空间
4. 进程只能通过系统提供的进程间通信机制进行交互
5. 同一进程**中线程间的上下文**切换通常比进程间的上下文切换快
   * 1. 多线程模型
        + 用户级线程（User Threads）：不依赖于 OS 内核，利用线程库提供创建、同步、调度和管理的函数来控制用户线程。用户不了解用户线程的存在，线程切换不需要内核的特权。一个线程发起系统调用而阻塞的时候，整个进程会等待
        + 内核级线程（Kernel Threads）：依赖于 OS 内核，在内核的内部需求进行创建和撤销，线程的切换和上下文信息维护都由内核完成

**lightweight process (LWP)** : an intermediate data structure between user and kernel

threads. 是一个虚拟处理器，进程可以在LWP上安排用户线程运行;每个 LWP 都附属于内核线程



multithreaded programming的好处：Responsiveness, Resource Sharing, Economy, Scalability

而多线程模型可以分为多对一模型、一对一模型、多对多模型，具体的特点如下：

* + - * 多对一模型（Many-to-One）：多个用户级线程对应于一个内核线程，通过用户级的运行时库来实现，OS 感觉不到用户线程的存在

★ 优点有不需要 OS 的支持，可以调整调度策略来满足应用程序 (用户级) 的需求

★ 缺点是当一个线程阻塞的时候整个进程都会阻塞

* + - * 一对一模型（One-to-One）：一个用户级线程对应于一个内核线程，线程的创建、同步和调度需要 OS 内核来进行，比如 Windows XP 和 Linux 都是一对一模型

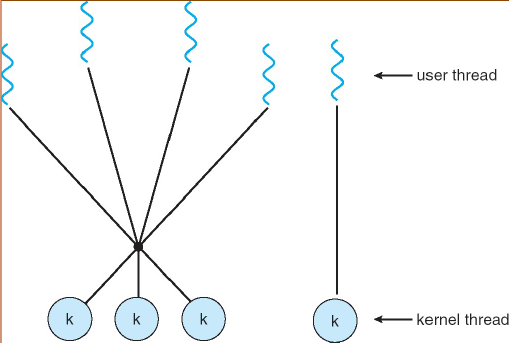
★优点有多个进程可以并行，当一个线程阻塞的时候其他的线程依然可以执行

★ 缺点是线程操作的开销更高，OS 内核必须有更好的 scale

* + - * 多对多模型（Many-to-Many）：多个用户线程对应多个内核线程，方便，比如 Windows NT/2000

|  |  |  |
| --- | --- | --- |
| Many-to-One Model | One-to-one Model | Many-to-Many Model |
|  |  |  |

此外还有二级模型，就是多对多 + 一对一的缝合版



**Thread Pool**: 在pool中可以创造很多thread，并且使得thread在pool中等待

优势：

1. 使用现有线程处理请求通常比创建新线程稍微快一些。

2. 允许将应用程序中的线程数量限制在池的大小范围内。

3. 将要执行的任务与创建任务的机制分离，可以采用不同的策略来运行任务。例如，任务可以被安排定期运行。

**thread cancellation**

Asynchronous cancellation 立即终止目标线程

Deferred cancellation 允许目标线程周期性地检查是否应该取消

CPU 调度

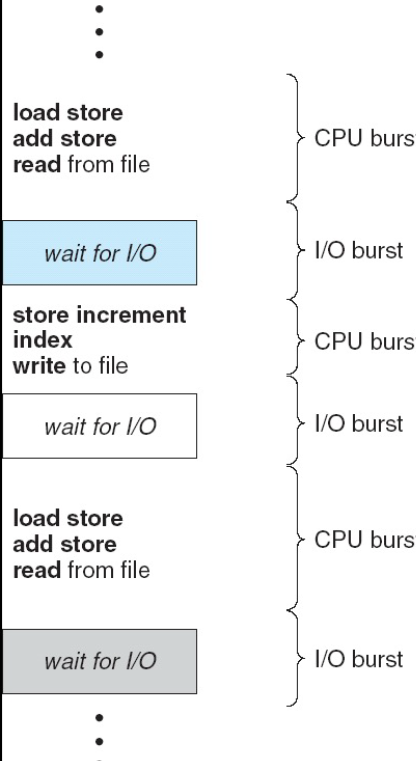
* 1. 基本概念

CPU 调度 = 处理机调度 = 进程调度，进程运行消耗的时间主要是 CPU 时间和 I/O 时间，调度主要分为长程、中程和短程三种(之前已经提过了) 调度Dispatcher 的时候需要进行上下文的切换，并将操作系统切换到用户态

■ Maximum CPU utilization obtained with multiprogramming

■ CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait. CPU burst followed by I/O burst

■ CPU burst distribution：A burst is an instance of breaking



**Theorem 3.1** 调度发生的时间节点：Switches **from running to waiting** state（非抢占式nonpreemptive）, from **running to ready state**（preemptive）, **Switches from waiting to ready**（preemptive）, **Terminates（nonpreemptive）**



调度根据是否可以抢占又分为

* 非抢占式调度：进程一旦被调度了就一直运行，直到进程终止或者阻塞，处理机才执行另一个进程
* 抢占式调度preemptive：程序运行的时候可以基于某种原则，剥夺进程的 CPU 使用权，分配给别的进程调度的评价标准
* 面向用户的评价标准

★ 周转时间 Turn-around Time：进程从提交到完成所经历的时间

★ 响应时间 Response time ：从进程提交到首次被响应所需要的时间

★ 等待时间 Waiting Time：进程在ready quene中等待的时间之和 (一个进程可以有多段等待时间相加)

* 面向系统的性能准则

★ 吞吐量 Throughput：系统在单位时间内完成的进程数

★ CPU 利用率：CPU 工作时间占总时间的比重，我们需要让 CPU 尽可能地忙碌

* 1. 调度算法

以下列举了操作系统中比较常见的一些调度算法

* FCFS 算法：先来先服务，先来的直到结束或者阻塞才让出。有利于长进程而不利于短进程，对 **CPU** 型进程友好
* SJF 算法：预计执行时间(CPU burst)最短的进程优先，但事实上很难实现，因为运行时间不好估计。它的抢占式调度版本Preemptive SJF也叫 **Shortest-Remaining-Time-First (SRTF)**，如果新来的比当前执行的花的时间少就抢占

SJF is Optimal in average waiting time

* Priority 调度：给每个进程分配优先级，优先级高的进程先执行

★ 静态优先级和动态优先级策略：静态优先级会在进程一提交就确定优先级，动态优先级则会对优先级随情况调整

★ 抢占式的优先级调度：优先级高的来了会夺取CPU 使用权

**Definition 3.1** 饥饿 *(Starvation)*：在优先级调度中，指优先级低的进程长时间得不到 *CPU* 的使用权的情况。优先级高的进程霸占 *CPU*，解决的方法是老化 *(Aging)*，即采用动态优先级策略，让进程的优先级随着时间的推移而提高。

* RR 算法 (Roung Robin)：先将系统中的进程按照 FCFS 排成队列，每次调度将 CPU 分配给队列顶端的进程执行一个时间片**Quantum**，在时间片结束之后会发生中断，将该进程切换到就绪状态并放到队列末尾，然后进行上下文切换，执行下一个进程。时间片够大的时候就是FCFS

* 多级队列调度 Multilevel Queue：将就绪队列也分为若干个子队列，不同的队列可能有不同的优先级、时间片长度和调度策略

Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing; others execute only user code.

Symmetric multiprocessing(SMP) – each processor is self-scheduling. Multiple processors might access and update a common data structure

|  |  |
| --- | --- |
|  |  |

同步 Sychronized

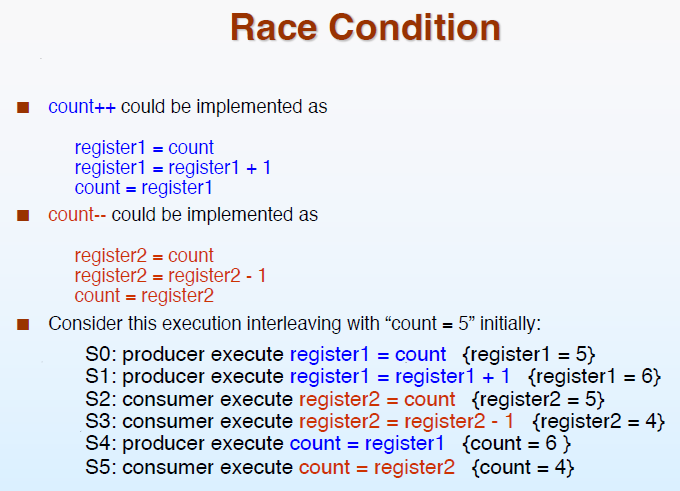
（每年期末的最后一道大题基本都是这方面）

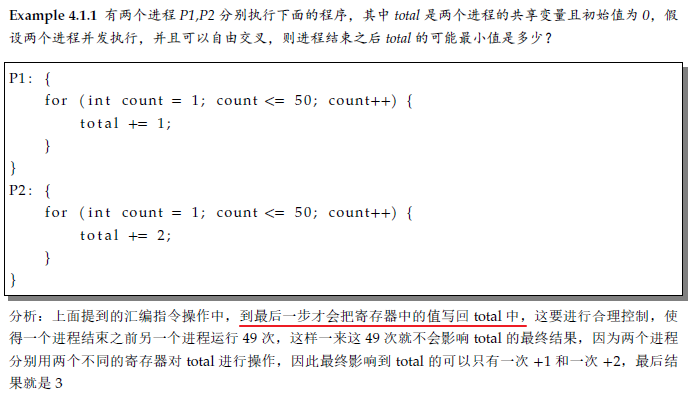
* 1. 基本定义

**Definition 4.1** 并行的进程或者并发的Concurrent线程在访问共享的数据区域的时候可能会出现数据的不一致性data inconsistency，这时候就需要一定的机制mechanisms来维护

**Definition 4.2** 有限缓冲区问题：生产者往有限的缓冲区写入数据，消费者从中读取数据，如果我们用一个*count* 来表示缓冲区中数据的个数，两种操作分别对应的就是 *count++* 和 *count–*，但是这两种操作都需要满足**原子性**，即操作的时候不能被中断，如果一个生产者和一个消费者同时想要改变 *count* 就会造成数据的不一致性。

|  |  |
| --- | --- |
| Producer | Consumer |
|  |  |





* + 1. 竞争条件Race Condition

多个进程都需要访问和修改一块共享数据区，此时数据的状态取决于最后一块访问这一区域的过程。为了防止竞争条件的出现，必须要将进程同步。竞争条件面临的三个主要问题

* + - * 互斥 Mutual Exclusion 多个进程不能同时得到一个资源
      * 死锁 Dead Lock 进程之间互不相让，导致永远得不到资源(多个信号量交叉等待)
      * 饥饿 Starvation 资源分配不公平导致进程长时间得不到资源
    1. 临界问题 **Critical-section problem**

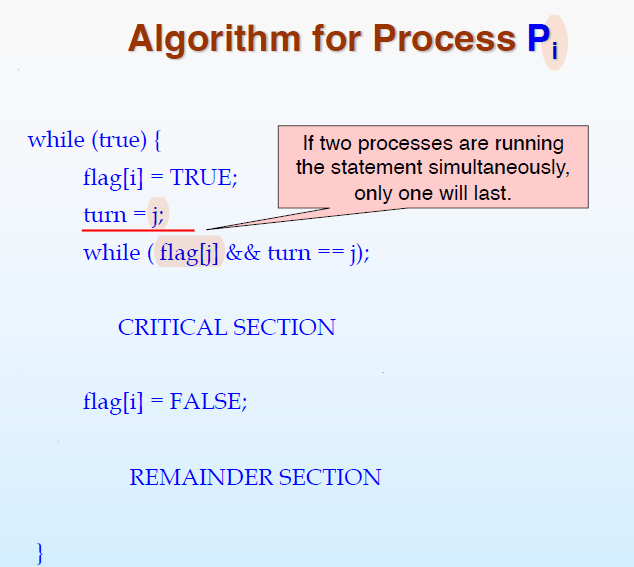
**Definition 4.3** 临界资源 *Critical Resource*，一次只允许一个进程访问的资源，比如消息缓冲队列

**Definition 4.4** 临界区 *Critical Section*，进程中访问临界资源的代码片段

同步问题的 solution 必须满足以下几个条件

* + - * 互斥 Mutual Exclusion（正确），一个进程在访问临界资源的时候其他进程不能访问
      * 空间让进 Progress，如果临界资源没有进程在访问，并且有进程想要进入临界区域，就应该允许其进入
      * 有限等待 Bounded Waiting，进程需要在等待有限时间之后可以访问资源，即不能产生死锁
    1. Peterson’s Solution

假设有原子操作LOAD和SOTRE, 然后可以定义proccess为



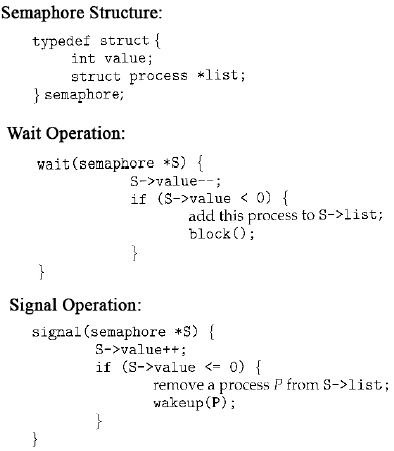
理论可行但是事实上不可行，因为现实中不存在原子操作

4.1.4 同步硬件 Synchronization Hardware：（这两具体的内容考试不会考）

1. 对于uniprocessor(只有一个CPU)，只要disable中断
2. 其余可以通过在硬件中写入原子操作(non-interruptable来避免)，比如TestAndSet Instruction， Swap这两个方法写入硬件后，可以实现Peterson’s solution但是只允许每个资源访问一次，所以也不好
   1. 信号量 **Semaphores**

信号量是一种数据类型，只有初init，**wait**(--) 和 **signal(++)** 2种原子操作，信号量可以分为整型信号量、记录型信号量,AND 型信号量和二值信号量。 注意大题中，信号量的定义就有很多分

信号量的数据结构和具体原子操作



该数据结构中list 指向等待队列的下一个进程，记录型信号量中value 的值大于 0 表示资源的个数，等于0的时候表示没有资源可以用或者不允许进入临界区，负数表示在等待队列中的进程个数或者等待进入临界区的进程

**Theorem 4.2** 信号量的两种**原子操作**（原子操作不能中断，一旦开始只能执行完）：*block()* 是将进程放置到等待队列上面，即从 *running* 切换到 *waiting*， 而 *wakeup()* 则是讲 *waiting* 转换成 *ready* 状态。

wait 操作和 signal 操作的实现方式

**同步的大题，初始值，算法**

* 1. 同步的三个经典问题
* 生产-消费者问题 *Bounded-Buffer Problem*：也叫做有限缓冲区问题，生产者写入缓冲区，消费者读取缓冲区。临界是buffer满了生产者休眠；buffer为空消费者休眠

假设缓冲区有 N 个，需要设置三个信号量 (排它锁)**mutex**=1, (两个临界信号量) **full**=0, **empty**=N

|  |  |
| --- | --- |
| 生成者 | 消费者 |
|  |  |

* 读者-写者问题 *Readers and Writers Problem*：只读取数据的成为读者，会修改数据内容的就是写者，一般来说允许多个读者一起读， 但是写者只能有且仅有一个

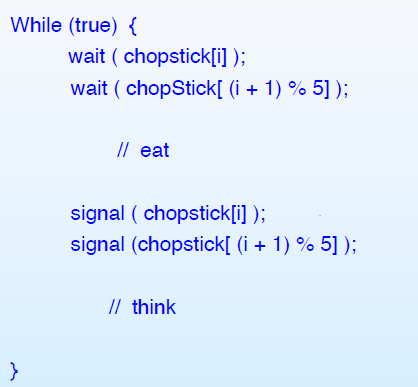
两个信号量 mutex=1, wrt=1。 然后一个计数器readcount=0

|  |  |
| --- | --- |
| write | read |
|  |  |

但是这里有个不能解决一个问题，bounded-wait writer会变成一个饥饿问题（只会信号量无法解决）

* 哲学家吃饭问题 *Dining-Philosophers Problem*：一群哲学家围成一圈吃饭，每两个哲学家之间有一根筷子，一个人吃饭需要拿起左右两边的筷子，哲学家在吃饭和思考之间切换状态

五个信号量 chopstick[5]=1(均为1)



如果都是左撇子（都选择eat，并且先左边），会导致死锁问题

ppt上后面的内容，不考

死锁 DeadLock

* 1. 基本概念
     1. 定义

死锁是指多个进程因为共享资源而造成互相等待的局面，若无外力作用，这些进程将永远阻塞不能继续执行。因为程序是人写的，只会按照确定的逻辑执行，而不会主动回退。

* + 1. 产生死锁的必要条件

产生死锁有四个必要条件：

* + - * 互斥 **Mutual Exclusion**：一个资源在一段时间里只能被一个进程使用
      * 占有并等待 **hold and wait**：进程已经占有了一个资源又提出新的要求，而该资源又被其他的进程占有
      * 资源不可抢占/剥夺 **No preemption**：资源只能等待占有它的进程使用完之后释放，不能去抢占资源
      * 循环等待 **Circular Wait**：一系列的进程都占有了前一个进程需要的资源，并在等待后一个进程释放资源
  1. 资源分配图 **Resource Allocation Graph**
     1. 资源分配图的组成

|  |  |
| --- | --- |
| 表示资源的分配情况，由若干节点V 和边E 构成，是**有向图**。节点分为两类，**P** 表示各个进程，**R** 表示资源，一个资源节点可以有多个实例。而边也分成请求边和分配边两种，请求边request edge从P 指向R，表示进程请求这个资源，分配边assignment edge从 R 指向 P，表示资源被分配给这个进程。 |  |

* + 1. 死锁的判定

资源分配图中有环就表明可能存在死锁，但是当一个资源有多个实例的时候也可能不会死锁，需要结合实例个数和是否有环来判定是否存在死锁：

* + - * 1 没有环的肯定没有死锁
      * 2 有环并且only one instance per resource type，产生死锁
      * 3 有环但是several instances per resource type，就不会产生死锁

|  |  |
| --- | --- |
| 2 | 3 |
|  |  |

* 1. 死锁的处理

处理死锁的三种方法：(never进入死锁)在程序的开发阶段预防和避免死锁，（死锁recover）在程序运行的时候进行死锁的检测和解除，（ 鸵鸟方法）假装死锁从来没发生过，这一方法被 Unix，Linux 和 Windows 操作系统所使用

* + 1. 预防死锁 Deadlock Prevention

预防死锁主要需要从产生死锁的四个必要条件出发，阻断一切产生死锁的可能性，归结起来主要有如下四点：

* + - * 互斥：必须给不能共享的资源加锁
      * 占有并等待：必须确保一个进程请求一个资源的时候不占有其他任何资源
      * 资源不能被抢占
      * 循环等待：资源必须有序地申请，找到需要的数量从小到大的顺序，可以破坏循环等待很显然这要的要求是比较苛刻的，也难以达到
  1. 死锁的避免算法
     1. 安全状态和安全序列

**Definition 5.1** 如果系统存在一种调度方式使得一系列进程可以申请到所有所需资源并完成运行，那么就称系统处于安全状态，而这个进程序列也叫做安全序列safe sqeuence，让系统处于安全状态有以下两种方法。

如果系统处于safe state，很明显就不会产生死锁；但unsafe的时候也不一定会死锁

* + 1. 资源分配图算法(用于解决Single instance of a resource type)

**Definition 5.2** 资源分配图算法：用虚线来标识需求边Request edge(claim edge是可能需要，当真的需要资源的时候会转换为request edge)，在一定条件下可以转换成请求边assignment edge(assignment edge释放了就又会重新变为claim edge)，如果进程 *P* 需要资源 *R*，就作一条 *P* 指向 *R* 的需求边，只有当这条边变成分配边 *R-P* 而不会使资源分配图形成环的时候才允许这次申请。

* + 1. 银行家算法 **Banker’s Algorithm** （用于解决Multiple instances of a resource type）

假设有 n 个进程和 m 种资源需要进行分配调度，我们首先定义如下数据结构：

* + - * available[m]=k 表示第 m 种资源有 k 个实例 (m)
      * max[i,j]=k 表示第 i 个进程需要资源 j 的 k 个实例 (n\*m)
      * allocation[i,j]=k 表示第 i 个进程已经被分配了资源 j 的 k 个实例 (n\*m)
      * **need[i,j]=max[i,j]-allocation[i,j]** 表示第 i 个进程**还需要**资源 j 的 k 个实例

**Algorithm 1** 安全算法

1: 令 work=available，**finish[i]=false** for 1 ≤ *i* ≤ *n*

2: 找到一个下标 i 使得 finish[i]=false 并且 *need*[*i*] ≤ *work*[*i*] 即每种资源的需求都不超过可分配的数量，如果没有符合条件的就进入第 4 步

3: work=work+allocation[i],finish[i]=true 即第i 个进程在分配到资源之后成功执行，并且释放了所有占有的资源

4: 如果所有的 finish[i] 都变成了 true，则系统处于安全状态，否则就是不安全的

**Algorithm 2** 资源分配算法

1: 令request[i,j]=k 表示进程i 需要资源j 的k 个实例，必须要保证 request[i] 不超过need[i] 和available[i]

否则就出现错误了

2: 假装将第 i 个进程所需要的资源都分配给了它，即进行了如下操作： available-= request[i]

allocation[i] += request[i]

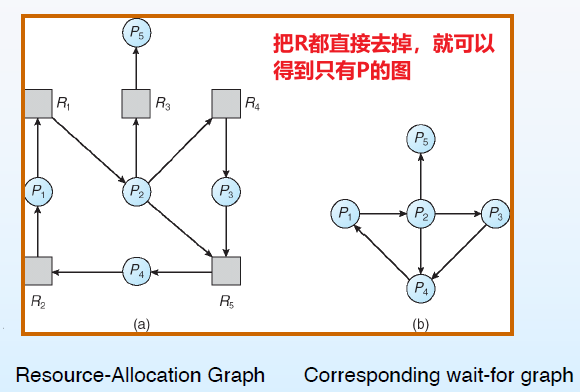
need[i] -= request[i]

3: 调用安全算法来检测进行了上面的操作之后系统是否处于安全状态，如果是就将资源分配给 Pi，否则 Pi

必须进入等待状态

* 1. 死锁的检测 Deadlock Detection

每种资源只有一个实例的时候可以使用等待图，如果有环就表明系统存在死锁，如果是多个实例，可以用类似于安全算法的方式来检测是否存在死锁(一般只考银行家算法，这个多个实例的就不管了)。



* 1. 死锁的恢复

1. 进程中止 Process Termination

杀死所有

一个个杀知道恢复

1. 资源抢占 Resource Preemption

Rollback – return to some safe state, restart process for that state.

Starvation – same process may always be picked as victim, include number of rollback in cost factor.

主存 Main Memory

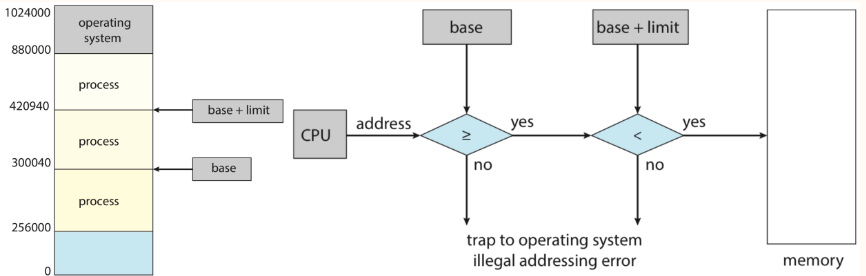
* 1. 基本概念

程序一定要从磁盘装在入内存之后才能运行，内存和寄存器是 CPU 可以直接访问的存储部分，而 cache 位于内存和寄存器之间

* + 1. 内存保护

每一个进程在内存中都应当有一块连续的内存空间，而单个进程应当只能访问自己的内存空间，而不能访问其他进程的内存空间。这就是内存保护的基本要求。

我们通过引入 base 和 limit 两个寄存器来实现框定进程的内存空间，当前进程的内存空间始于 base 寄存器中存储的地址，终于 base + limit 对应的地址，即：



两个特殊的寄存器只能由内核通过特定的特权指令来修改。而内存的保护，通过内存管理单元(memory management unit, MMU)来实现，MMU 会在每次访问内存时，检查访问的地址是否在 base 和 limit 寄存器所定义的范围内，如果不在，则会产生一个异常，中断程序的执行。

* + 1. 地址绑定 Binding of Instructions and Data to Memory

用户的源代码需要经过一系列步骤才能变成可执行程序，并装入内存运行，具体有如下几个步骤：

具体来说有三个阶段：编译时间(compile time)，装载时间(load time)和执行时间(execution time)。而内存也分三种：符号地址(symbolic addresses)，可重定位地址(relocatable addresses)（类似于一种相对量）和绝对地址(absolute addresses)。

通常来说，在 compile time，compiler 会将代码中的 symbol 转为 relocatable addresses；而如果在 compile time 就知道了进程最终会被安置在何处，那么在 compile time 就将 symbol 转为 absolute addresses 也是可能的，只不过如果此时起始地址发生改变，就需要重新编译。

而一般在 load time，relocatable addresses 会转为 absolute addresses，当进程起始地址发生改变时，我们只需要重新装载即可。

如果进程在 execution time 时，允许被移动，那么可能从 relocatable addresses 转为 absolute addresses 这一步就需要延迟到 execution time 来执行。绝大多数操作系统都使用这种方案。

* + 1. 逻辑地址（虚拟地址）和物理地址

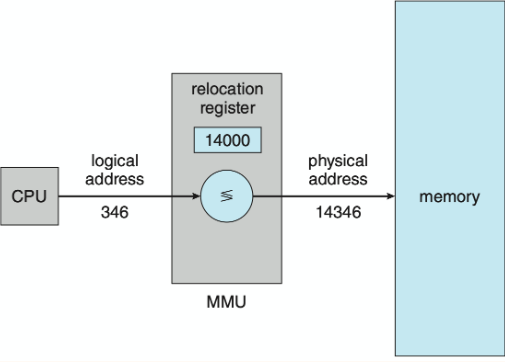
逻辑地址，由 CPU 生成，用户程序汇编之后生成的目标代码通常采用相对地址的形式，将首地址作为 0，而其他的指令的地址都用相对于首地址的偏移量来决定。逻辑地址不能用于读取内存中的信息。物理地址，可以用来直接寻址。

虚拟地址的实现需要基地址寄存器和限长寄存器，一对基地址寄存器和限长寄存器定义了逻辑地址空间，其中基地址寄存器存放地址的首地址，而限长寄存器存储整个地址空间的长度

* + 1. 内存管理单元 **MMU**

内存管理单元 MMU 是一种将虚拟地址映射成为物理地址的硬件设备，在 MMU 中，重定位寄存器中的值被加到每一个由用户进程生成的地址中。

除了是实现虚拟地址->物理地址的映射外，MMU 还负责内存访问的保护。TBL 也属于 MMU 的一部分。下图是Dynamic relocation using a relocation register：



* + 1. 动态装载

由于引入了多道技术，操作系统的内存中可能同时存在多个进程。为了更加灵活地使用内存资源，我们引入动态装载(dynamic loading)机制。

动态装载指的是，如果一个例程还没有被调用，那么它会以可重定位装载格式(relocatable load format)5存储在磁盘上；当它被调用时，就动态地被装载到内存中。即，例程只有在需要的时候才被载入内存。对于大量但不经常需要访问的代码片段（例如错误处理代码），这种方式可以节省大量的内存空间——这种只有偶尔会被访问的代码也不应当长久地占有内存。

需要注意的是，动态装载并不需要操作系统的支持，而是由开发者来负责实现

* + 1. 动态链接和共享库

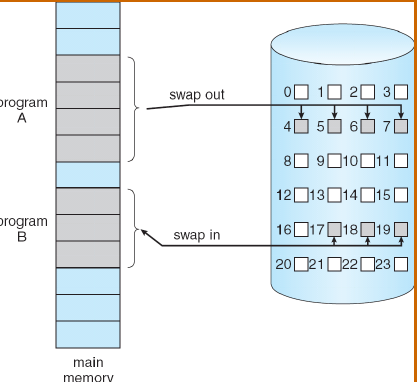
我们在总览#链接器和装载器中已经谈论过动态链接了。而能被动态链接的库就被称为动态链接库(dynamically linked libraries, DDLs)，由于它们可以被多个进程共享，所以也被称为共享库(shared libraries)。

区别于动态装载，动态链接需要操作系统的支持。

* 1. 交换技术 Swapping

交换技术是指一个进程可以暂时被移出内存并保存在后备存储器中，之后再次装载进入内存执行。后备存储器是一个可以存储所有内存映像的磁盘，可以提供对这些内存映像的直接访问，在 Linux 中表现为交换区，而在 Windows 的系统中是一个 pagefile.sys 文件。

系统运行的过程中，优先级低的进程会被调出，而优先级高的会被调入内存，这一过程需要使用交换变量，主要是transfer 的过程耗时。系统会维护一个准备队列来存放准备运行的进程，其内存映像都存储在磁盘上面。一般来说，系统在正常运行的时候是不会去交换的，只有当可用内存数量低于一个阈值的时候才会启动自动交换，在可用内存增加到一定数量的时候停止换出。



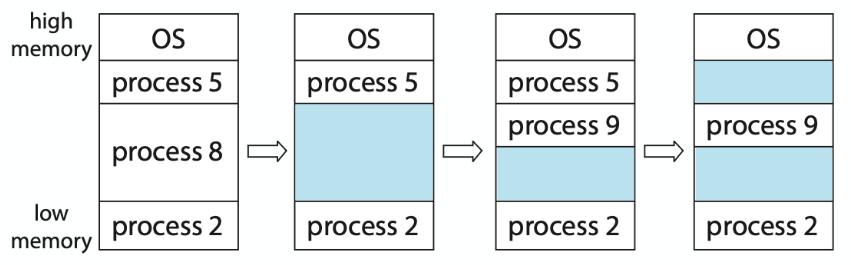
**Lazy swapper**： 直到一个page要被用到才真正换到memory中

* 1. 连续分配 Contiguous Allocation

主存往往被分为两个单独的分区，**常驻操作系统 (Resident operating system)** 保存在具有中断向量的低内存中，**用户进程**被保存在高内存中。

**浮动寄存器 (relocation register)** 被用来保护用户进程，将进程和进程之间隔离开，并防止进程更改操作系统中的数据和代码，其中：

* 基寄存器Base register保存最小的物理地址
* 有限寄存器Limit register包含了逻辑地址的区间，每个逻辑地址必须要处于这个区间中
* MMU 是动态映射逻辑地址



分区式管理的基本思想时将内存划分成若干个连续的区域，称为分区，每个分区只能存放一个进程，分区的方式有固定分区和动态分区。固定分区就是将内存分成若干块固定长度的区域，区域和区域之间存在着未使用的内存空间。

而动态分区需要动态地划分内存，在程序装入内存的时候把连续的一块内存区域分配给该进程，并且大小正好适合该进程的需要，存在下面几种动态分配算法：

* + - * First-Fit 分配第一个发现的可用的足够大的内存分区给进程
      * Best-Fit 分配所有可用块中最小的并且足够大的内存分区给进程
      * Worst-Fit 分配最大的一个分区给进程。在速度和内存利用上都比前两种要差

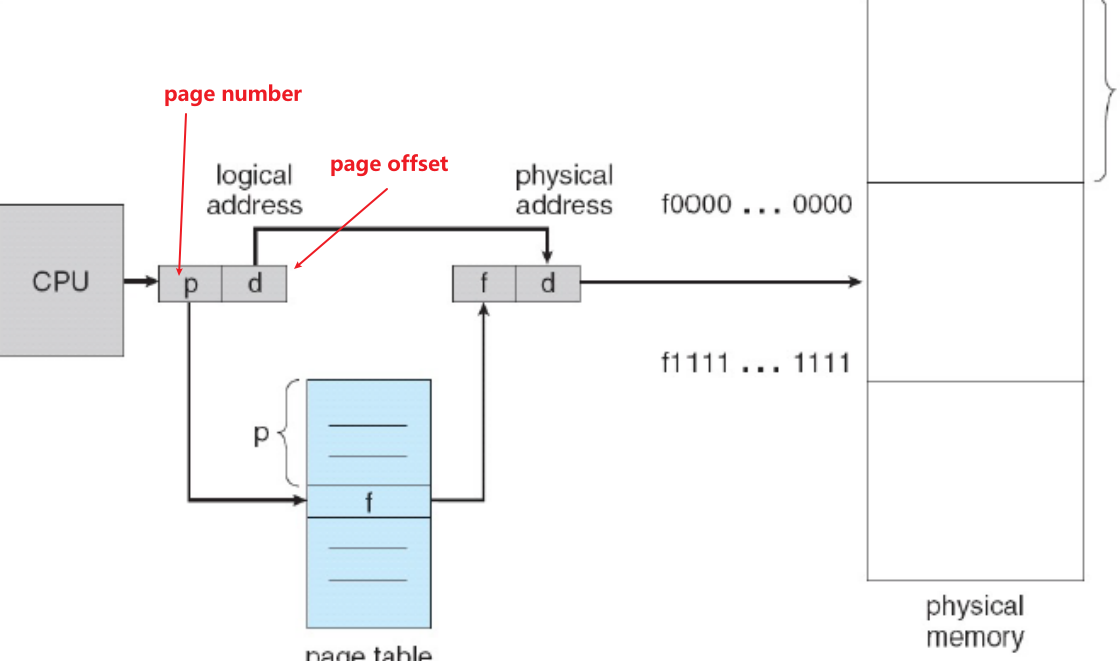
很明显，best-fit 和 worst-fit 算法需要遍历所有内存之后才可以做出决定

一个进程内部的空闲的内存（固定分区的时候出现），称为**内部碎片 (inner frag- mentation)**，相邻的两个内存空间之间空闲的内存段叫做**外部碎片 (outer fragmentation)**，碎片的存在会减少可运行的进程的个数。可以通过拼接和压紧compaction来进行管理，但是消灭内存碎片开胸啊很高

* 1. 页式存储管理

页式存储管理可以使得一个进程内部的逻辑地址所处的物理空间可以不连续，只要在内存有空的地方就可以给进程分配物理内存。

物理内存被分为若干个固定大小的帧 (Frame)，而逻辑地址也被分成了若干个大小相同的页 (Page)，一般来说页和帧的大小相同，都是 4KB，运行一个含有 n 页的程序需要找到内存中 n 个空闲的帧。



逻辑地址结构由页号 p 和页内偏移量 d 组成，因为帧和页都是 4KB，因此是 32 位的地址，地址结构决定了内存的寻址空间有多大，对于逻辑地址空间大小为 2*m*，页的大小是 2*n* 字节的情况，**页号有 m-n 位**，偏移量有 **n 位**

* + 1. 页表 **Page Table**

页表位于内存中，记录了页面在内存中对应的物理块号，页表由若干页表项组成，每一项由页号和物理内存的块号组成。进程执行的时候，通过查找该表，可以找到物理块号，即逻辑地址空间通过页表的转换变成了物理地址空间。

同时，页表也有**页表寄存器 PTR** 来存储页表的(base)起始地址 F 和(length)页表长度 M，两个寄存器分别叫做 **PTBR** 和**PTLR**，每次内存的访问都需要经过逻辑地址空间到物理地址空间的转换，因此页表的访问速率如果太慢就会导致程序运行的效率降低，同时，页表如果太大，就会导致内存的利用率降低。

Page table 中还会有vaild位表示是否有效

* + 1. 快表 **TLB**

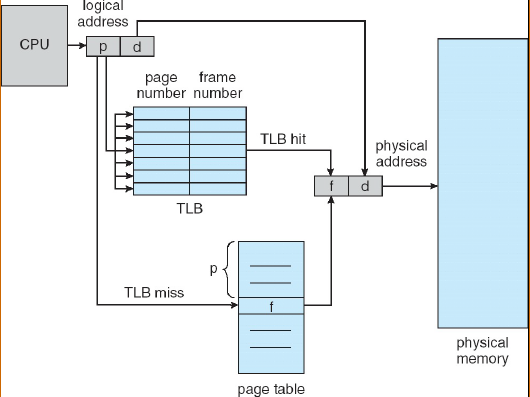
**TLB**，translation look-aside buffers，相当于页表的cache，存放了一些当前访问的若干项页表项，支持并行的快速访问，相对应的内存中的页表也叫做慢表。有了 TLB 之后，内存访问的工作原理就变成了：

* + - * CPU 给出程序的逻辑地址之后，要访问对应的物理地址时，先去查询 TLB
      * 若查到了对应的物理块号，就加上偏移量之后直接去内存访问对应的物理块
      * 若没有在TLB 中查到就在页表中继续查询对应的物理块号，然后将查到的物理块号存储到 TLB 中，以便下次继续访问

但是 TLB 的访问不是一定可以成功的，存在一个命中率 *α* (percentage of times that a page number is found in the associative registers)，假设快表 TLB 的访问需要的时间是 x，内存访问的时间是 y，则有效访问时间 (EAT) 的计算公式是：

*EAT* = (*x* + *y*)*α* + (*x* + 2*y*)(1 − *α*)

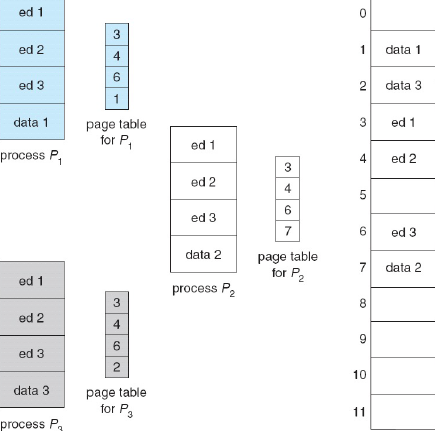
解释起来也很简单，因为访问内存和访问TLB 的时间存在数量级的差距，而TLB 中命中的时候只需要直接访问内存即可，因此耗时 x+y，而失配的时候要多访问一次内存去查页表，因为需要的时间是 x+2y



提升Page Size会导致更多的碎片，可以通过提供多种不同的Page Size来解决

* + 1. **共享页**

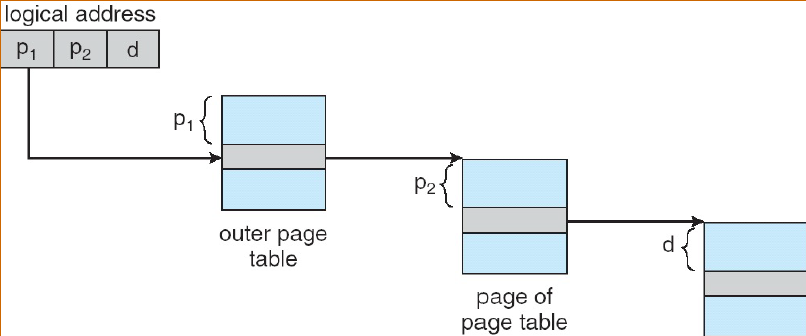
虚拟地址与物理地址的映射并非需要是单射，换句话来说，多个页可以对应同一个帧，这就是共享页(shared page)。共享页可以用来提高代码重用率，例如，多个进程可能会使用同一个库，那么这个库就可以被共享，而不需要每个进程都各自在物理内存中准备一份。共享库就通常是使用共享页来实现的

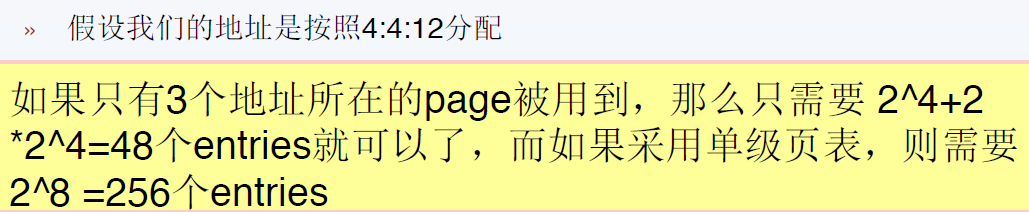


* + 1. 页表的类型 Structure of the Page Table

这些方法减少了存储页表所需要的内存空间，但是加大了内存的访问时间

**分级页表Hierarchical Page Table（最重要）**，将页号继续拆分成页号 + 偏移量，用新的页号作为原本页号的索引，层层查找页表

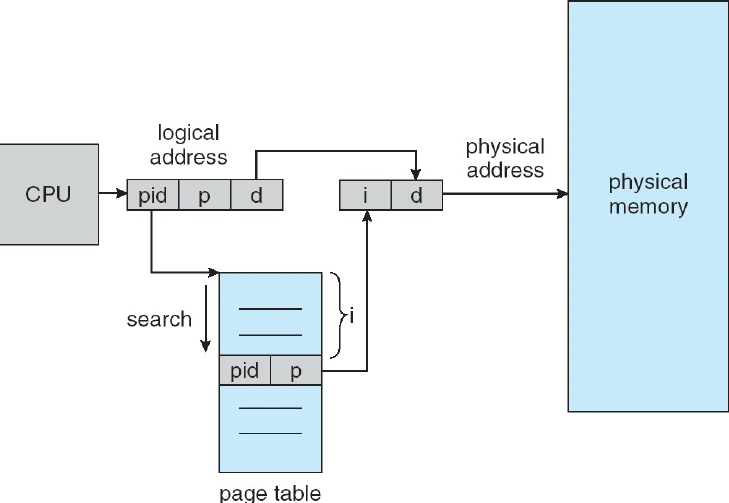




**哈希页表 Hashed Page Table**，将逻辑地址用哈希函数映射到页表中

**反向页表Inverted Page Table**，逻辑地址中包含进程的 pid，页表中通过 pid 和页号共同查询物理地址号。整个系统中只有一张表

* Has one entry for every frame of memory
* Records which page is in that frame
* Is indexed by frame number not page number!

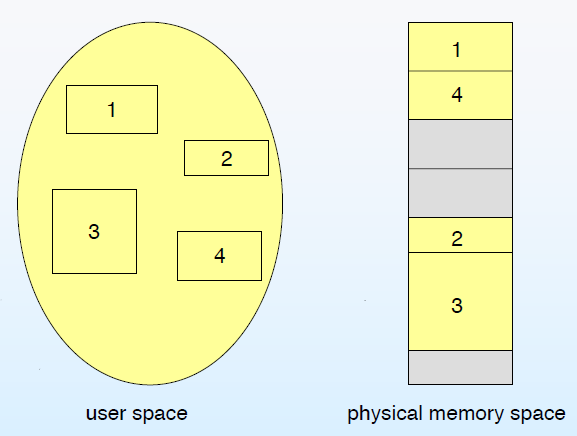


Search is slow, so put page table entries into a hash table.

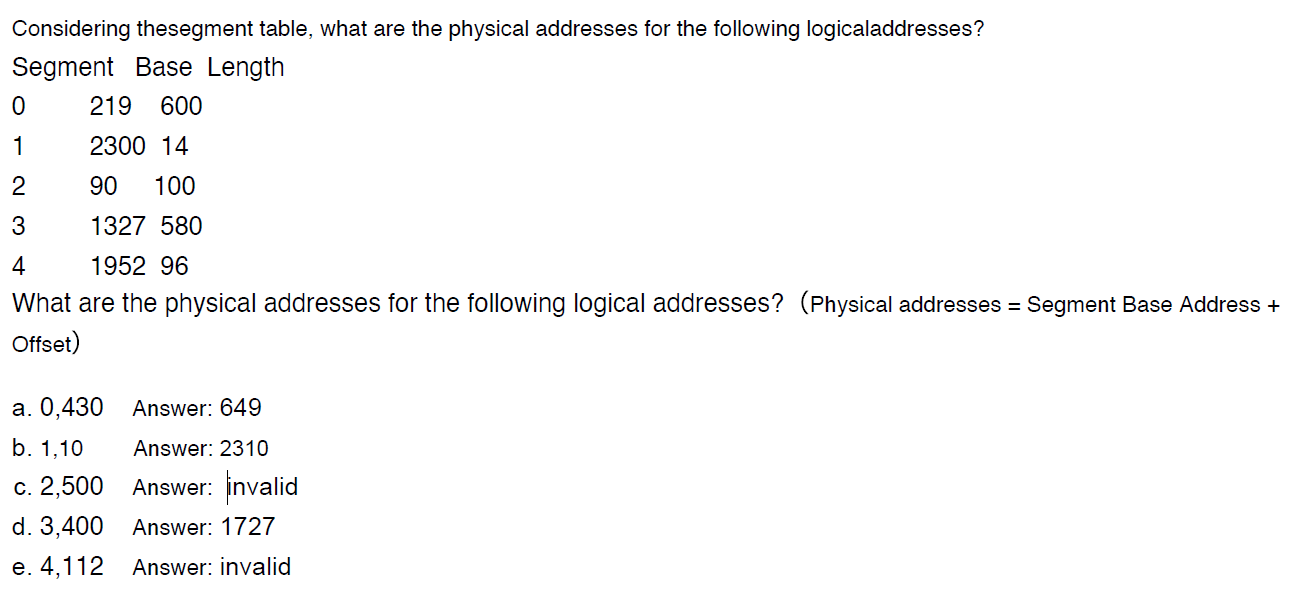
TLB can be used to speed up hash-table reference.

* 1. 段式存储管理 Segmentation

可以将一个进程分成若干段，包括代码段、数据段、栈区、本地变量区等等，每个段中从 0 开始编地址， 并分配一段连续的地址空间，逻辑地址分为段号和段内偏移量。并且每个进程都有一张表示逻辑地址和物理地址映射关系的段表，其中每个段表项对应进程中的某一段，表中的项记录了段号、段长和本段在内存中的起始地址。每个段长度不一样，共享内存的时候分段更方便



映射方法和页表一致，就是基地址＋偏移量



分页+分段有很小概率考，具体可以看王道

虚拟内存 Virtual Memory

* 1. 基本概念

虚拟内存的基本思路就是将一个程序正在运行的部分放入物理内存中，而不运行的部分则放在磁盘中，通过这种方式可以将物理内存和逻辑内存区分开，使得硬盘的一部分也变成了“逻辑内存”，“扩大”了可用的内存容量，使得虚拟内存的空间要比物理内存大。虚拟内存的管理模式分为请求调页 (Demand Paging) 和请求段式管理(Demand segmentation)两种。

* 1. 按需调页 Demand Paging

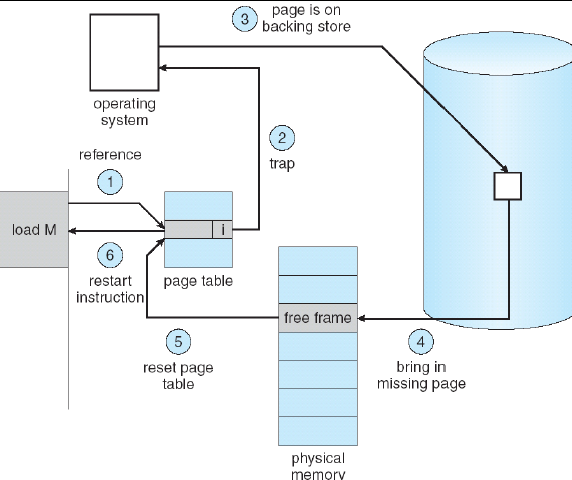
基本的想法是只在需要的时候将进程的页调入内存中运行，可以减少 I/O 和内存的使用，程序的响应更快，支持更多用户同时使用。

* + 1. 页表的结构

在使用虚拟内存技术的时候页表需要有一个有效位 (valid bit) 为1则在memory中，否则存在一个 page fault，在请求调页系统中，每个页表项的页号对应如下内容：

* + - * 物理块号
      * 状态位 P，表示该页是否已经调入内存中，供程序访问页的时候参考
      * 访问字段 A，用于记录本也在一段时间内被访问的次数，或者已经有多久没有被访问
      * 修改位 R/W，表示页在调入内存后是否被修改过
      * 外存地址，指出该页在外存中对应的存放地址
    1. 缺页 **Page Fault**

程序第一次访问页表的中的某一项的时候肯定会导致缺页，因为这个时候这一页还没有调入内存中， valid 位是无效，这会使得操作系统进入缺页中断之中，此时操作系统会把页调入内存中并修改表项，然后重新运行刚才的程序 (因为没有访问到对应的页)

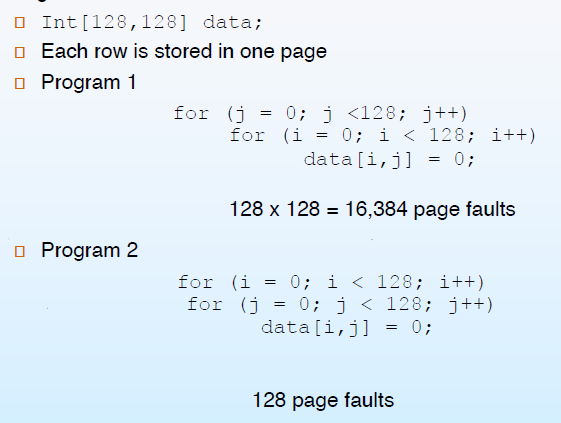


缺页率在 0-1 之间，内存访问有效时间 EAT 的计算方式是

*EAT* = (1 − *p*) ∗ *x* + *p* ∗ *y*

其中x 表示物理内存的访问时间，y 表示缺页中断服务所需的时间总和

Page fault和程序结构也有关系：



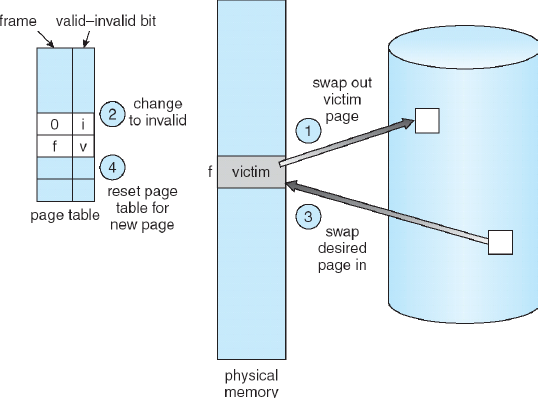
* + 1. 进程创建 **Process Creation**

虚拟内存允许进程创建的时候进行**写时拷贝(COW copy-on-write)**，这一技术允许父进程和子进程一开始的时候共享一些页，如果有进程需要改变页的内容，才会 copy 一份这个页供两个进程分别使用，这一技术使得进程的创建更加有效率。

* 1. 页的置换 **Page Replacement**

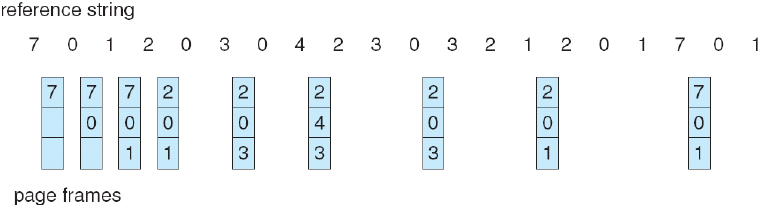
当没有空闲的物理帧frame的时候就需要进行页的置换，将一部分没有用的页换出，把有需要的页调入内存中，这一过程就是页的置换。页的置换的目标是尽可能减小置换之后缺页的概率。

将页面换出时需要将原本物理帧中存访的页调出时，我们可以使用1bit的dirty位来标识这一页是否被修改过，如果修改过就需要写入磁盘中，而如果没有修改过就不用再写一次而是直接覆盖原本的内容即可。



我们使用一个引用串 (Reference String) 来表示一系列对页面的引用，引用串由要引用的一系列页号组成。常见的页面置换算法有：

* + - * FIFO 先进先出算法，缺点是会引发 Belady’s Anomaly(更多的帧导致更多的缺页)
      * OPT 最佳页面置换算法：选择未来不再使用的或者当前最不需要用的页替换，使用的时候需要往后搜索引用串，观察最晚被再次引用到的现有页



* + - * LRU 最近最少使用算法：将最近没有引用到的帧换出，但是需要记录页面的使用时间，硬件的开销太大，该算法的关键就在于找到最久未使用的帧，实现方法有这样两种

★ 使用计数器，每个页有一个 counter，每次页被引用就把时钟拷贝到计数器中，当页中的内容需要改变的时候就通过 couter 来确定

★ 使用栈stack，用一个双向的链表来维护一个页号的栈，被引用的时候就移动到栈顶，优势在于确定替换页面的时候不需要搜索

* + - * LRU Approximation 算法，有如下几种实现方式

★ 可以考虑使用一个**引用位Reference bit**，初始化为 0，使用过之后就变成 1，优先替换引用位是 0 的页，很明显这个算法非常局限

★ **二次机会算法Second chance**，也叫 **clock** 算法，如果被替换的页的引用位是 1 就将引用位变成0，然后保留这一页，按时钟顺序往下找，直到找到一个引用位是 0 的进行替换，也就是说在引用位全是 1 的时候可能会转一圈

* 1. 其他
     1. 帧分配 **Allocation of frames**

帧的分配分为固定分配和优先级分配两种，用于给每个进程分配一定的物理帧

* + - * 固定分配：有平均分配算法和按比例分配算法两种，**平均分配算法 Equal allocation**就是每个进程分到等量的帧，**按比例分配 Proportional allocation** 就是按照进程的大小来分配相应的物理帧
      * **优先级分配**：按照一定的优先级比例而非大小比例分配。如果一个进程发生了缺页，可以置换自己的帧，也可以置换出一个优先级更低的进程的帧。置换的策略有全局置换和局部置换两种，**全局置换**就是可以置换所有可以置换的帧，问题是不能预测page fault rate 而**局部置换**只能置换该进程拥有的帧，问题是 low throughput: 因为不能用其他进程的空闲帧。
    1. 抖动 **Thrashing**

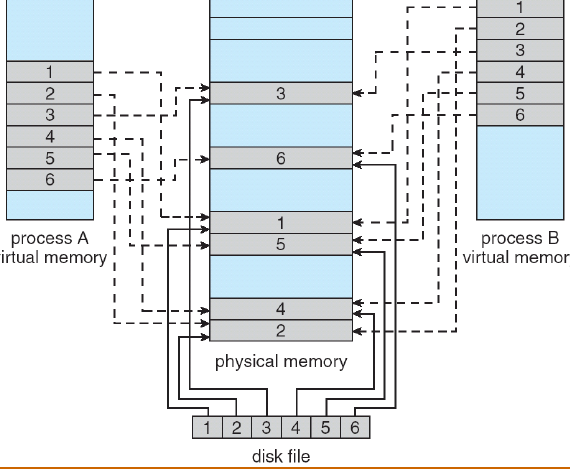
如果有一个进程**没有足够的页数**，会导致缺页率非常高，降低 CPU 的使用效率，抖动指的就是一个进程频繁换进换出的行为，当计算机的两个局部性失效的时候就容易出现抖动的现象。解决的措施就是分配更多的memory， 以及降低 degree of multiprogramming。（表现为，CPU 和其他设备的利用率很低，而用于交换空间的磁盘利用率非常高，这就反应出虽然 CPU 不太忙碌，但是进程一直在按需调页，说明分配给每个进程的物理内存空间不够）

另一个解决思路是working set，没做到过题，感觉不重要。可以看[xy](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit3-Part2/#working-set)了解一下

* + 1. 内存映射文件 **Memory-Mapped Files**

内存映射文件的读写允许文件读写被作为一个routine memory access 来进行，可以将一个磁盘块映射到内存中的一页，一个文件一开始用按需调页的方式读入，文件的子序列的读写被作为普通的内存访问来进行，允许多个进程将同一个文件映射到内存中。

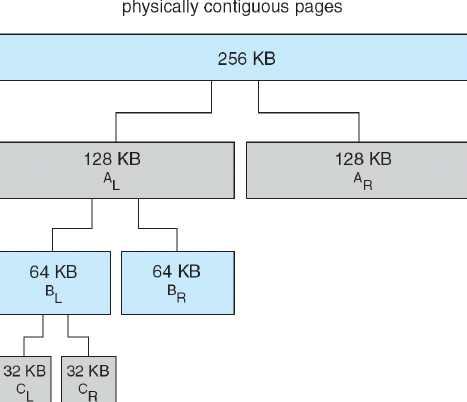
简化了文件调用，不再使用**read()**, **write()** 这些**system call**



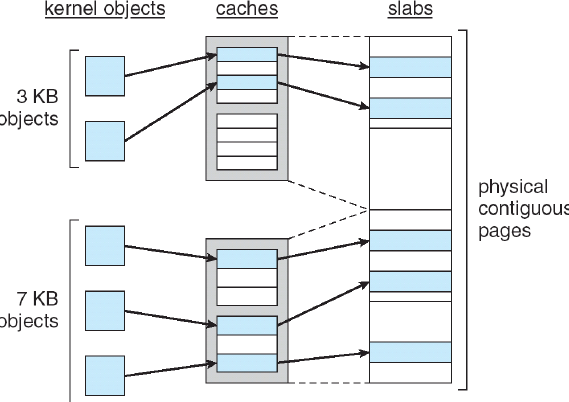
* + 1. 内核内存分配 Allocating Kernel Memory

内核的内存分配**和用户的不同**，通常用一个空闲的内存池，内核需要多种不同大小的内存（减少碎片），一些内核内存必须要是连续的。因此不适用paging的方法了

**Definition 7.1** *Buddy* *System*：使用一系列物理地址连续的定长段来分配内存，并且内存的分配以 *2* 的幂为单位。



**Definition 7.2** *Slab Allocator*：Slab 分配的大概思路是预先了解到 kernel 内的常见数据结构（被称为各种 object）的大小，并预先准备好对应粒度的小内存块，注册到每类 object 的 cache 里。当一个 object 需要使用内存时，就查询对应的 cache 里是否有空闲的内存块，如果有就分配给它，如果没有就向 Buddy system 申请。相当于预先把内存分成了苹果小盒、冰箱大箱、超大车库，然后根据要放的东西的大小选择合适的空间来放。



* + 1. 预调页 prepaging

在被引用之前预先调如一定的页，可以减少一开始会产生的大量缺页，但是如果预调入的页没有被引用的话就会导致内存和 I/O 的浪费

* + 1. I/O interlock

被I/O用到的帧必须在memory中

文件系统 File System

* 1. 文件系统接口
     1. 文件

文件是连续的逻辑地址空间。文件的信息一般保存在directory structure里面（存在disk里面）。文件结构分为以下三种

* + - * None：sequence of words, bytes
      * Simple record structure；按照行或者定长、变长的段来存储文件
      * Complex Structures：格式化文档Formatted document、可重定位加载文件Relocatable load file

关于文件分类可以看[这里](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit6/" \l "%E6%96%87%E4%BB%B6%E7%B1%BB%E5%9E%8B)简单了解

* + 1. 文件的访问方式

search the directory structure on disk for entry Fi, andmmove the content of entry to memory 一旦打开就返回一个 open-file table 中entry的指针

move the content of entry Fi in memory to directory structure on disk

打开文件需要：**文件指针File pointer**，指向上次的文件读写的位置；**文件打开次数的计数器File-open count**，记录文件被打开的次数，当最后一个进程关闭文件的时候就要从一张文件记录表open-file table中删除这个文件名；**文件的磁盘位置Disk location of the file**和访**问权限Access rights**

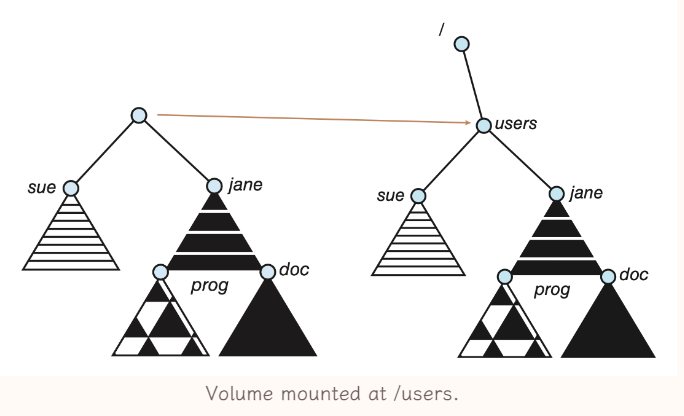
* + - * 顺序存取 Sequential Access：按照顺序读取文件的内容，有一个指向当前位置的指针，每读写一次都前进一格，也可以将指针 reset
      * 直接存取 Direct Access：可以直接读写 n 个字符
    1. 文件目录 Directory Structure

具体可以看[这里](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit6/#%E7%9B%AE%E5%BD%95%E7%BB%93%E6%9E%84)。需要补充的是树形目录，需要额外的一位bit来记录file(0), directory(1)

对于无环文件目录，允许了子文件和子文件夹共享

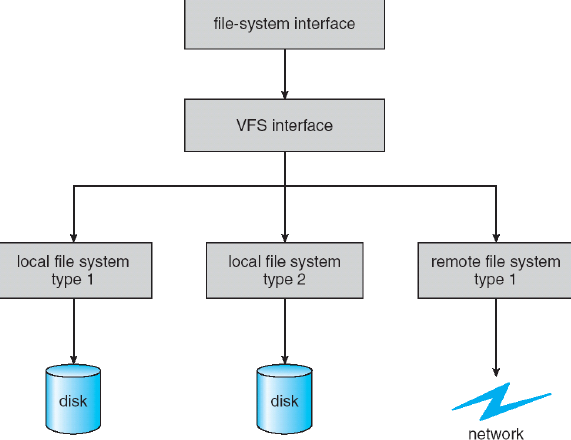
* 1. 文件系统的实现
     1. 文件系统的层次结构

按照计算机存储器层级架构的划分，文件系统位于二级存储中，即被存储在**磁盘**中。文件系统挂载(mount)是指将一个文件系统的根目录挂载到另一个文件系统的某个目录（被称为 **mount** point），使得这个目录下的文件可以访问到被挂载的文件系统中的文件。只有被挂载了，一个文件系统才能被访问



文件系统的物理设备由设备驱动器进行控制，而文件系统的分层如下：

* + - * 逻辑文件系统：管理各类文件的元数据，即文件系统的所有数据结构而没有实际的文件数据，根据给定的符号文件名来管理目录结构，逻辑文件系统通过文件控制块来维护文件的结构
      * 文件组织模块：把逻辑块转为物理块
      * 基本文件系统：对磁盘上的物理块进行读写
      * I/O 控制：由设备驱动程序和中断处理程序组成，实现内存和磁盘之间的 I/O
    1. 常见的文件系统
       - FAT File-allocation table 是 MS-DOS 系统使用的文件系统
       - NFS 是**网络文件系统**， 是一种分布式的文件共享方法
       - VFS Virtual File Systems **虚拟文件系统**是物理文件系统和服务之间的接口，给用户和程序提供了统一的API.定义一个network-wide unique structure称为vnode



* + 1. 文件控制块 **FCB**

文件控制块 (File Control Block) ，存储文件的各类元数据，包括文件的**权限**、时间信息、大小、所有者或者数**据块的指针**，打开一个文件的系统调用 **open** 实际上就是把对应的 **FCB** 调入内存中，要注意的是内存中的 FCB 不包含文件名的信息。(inode in UFS, master file table entry in NTFS)

内存中维护了分区表、目录结构、系统打开文件表和进程打开文件表等结构。

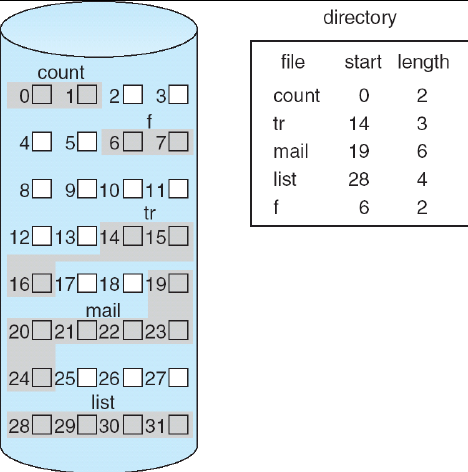
* + 1. 文件目录的实现

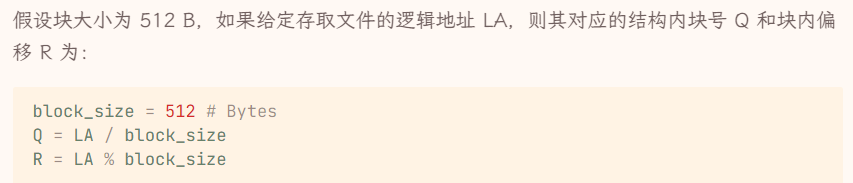
文件目录的检索可以采用线性检索法、哈希表法和索引等等，线性检索法实现简单而耗时较长，哈希表减少了搜索时间但是可能会引发冲突collisions和overflow。

* + 1. 文件物理结构与磁盘分配

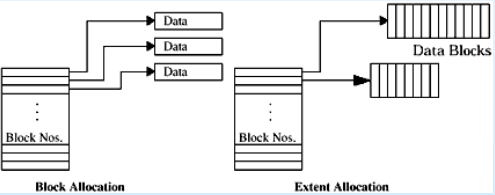
磁盘也被分为若干大小相同的块，一个文件可能会需要多个磁盘的块，这就需要对磁盘块进行分配，主要的分配方式如下：

* + - * 连续分配Contiguous allocation：每个文件分配一系列连续的磁盘块，简单，并且支持随机访问但是浪费空间，文件size不能增长

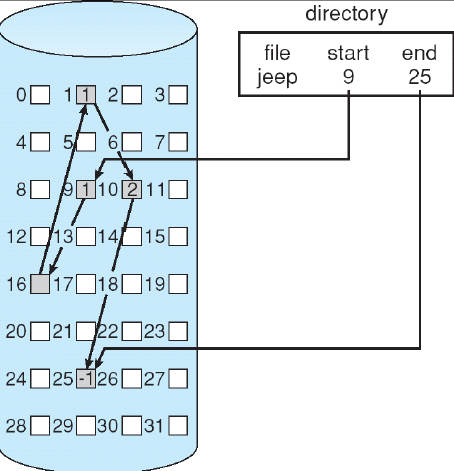


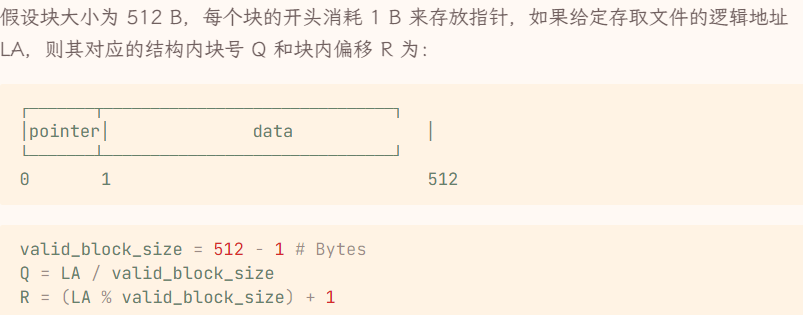


* + - * Extent-based 是一种改进的连续分配方式，分配磁盘块在**extents**(范围)中, 一个文件可能由一个或者多个extent

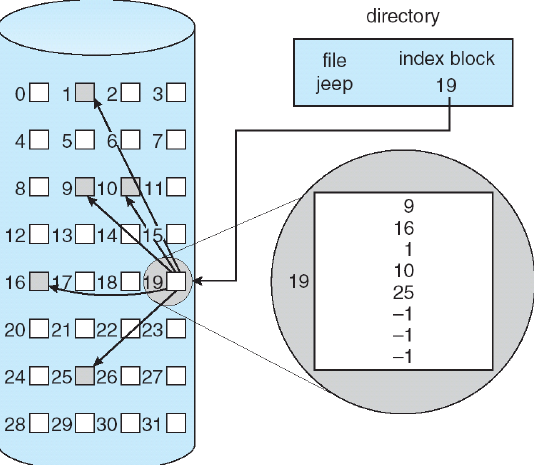


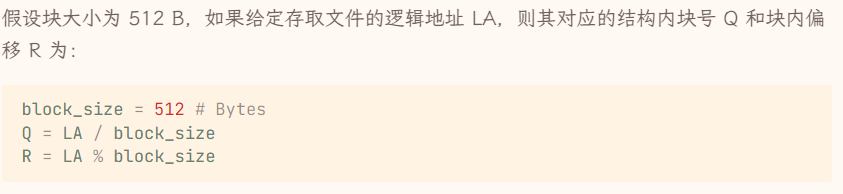
* + - * 链接分配 Linked allocation ：每个 block 作为一个链节，维护存储信息以外还需要维护指向下一个 block 的指针。简单，FCB 中只需要记录起始地址即可。不支持随机访问，可靠性较低





* + - * 索引分配Indexed allocation：使用索引块专门存放所有的指针，每个文件使用一个索引块，包含所有该文件所使用的块的地址，可以分为一级索引和二级索引等形式，二级索引就是用一级索引块存储二级索引的信息，索引块组成的链表被称为链接索引。好处是可以调整大小，坏处是随机访问比较慢





|  |  |
| --- | --- |
|  |  |

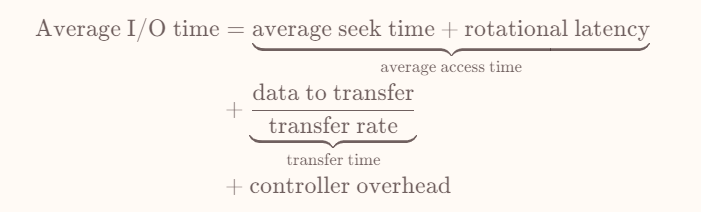
* + - * Linux 系统的文件系统一般采用混合分配的机制，也就是以某种分配方式为主体，多种分配方式并存
    1. 文件系统性能与安全

看看[这个](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit6/" \l "%E8%99%9A%E6%8B%9F%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F)就行，不重要

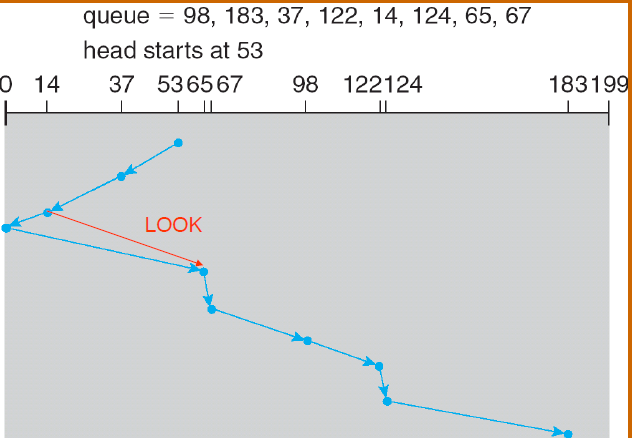
大容量存储系统 Mass-Storage Systems

* 1. 磁盘 **Disk**
     1. 磁盘的结构

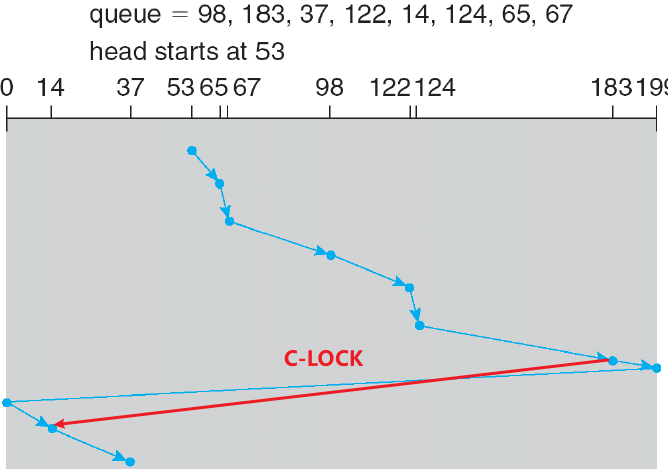
磁盘是二级存储，见[结构](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit5/#%E7%A1%AC%E7%9B%98)部分



* + 1. 磁盘的调度算法 看[这个的调度算法部分](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit5/#%E8%B0%83%E5%BA%A6)即可，需要补充的是
       - SCAN 扫描算法：先往一个方向扫描过去，然后再回过头反向扫描，扫描的方向取决于开始之前一次的方向，并且要先到达边界再转向。LOCK算法时对其的优化



* + - * C-SCAN 循环扫描算法：将磁盘是以循环的排列方式来扫描。C-LOCK 是不触底就反弹的 C-SCAN，到了最后一个请求就直接折返， 并不死磕到底



* + 1. 磁盘的管理

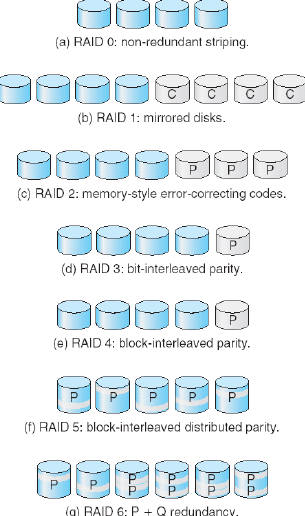
磁盘的低级格式化 (Formatting，也叫做物理格式化) 是指将一个磁盘分成若干个扇区可以使得磁盘控制器机型读写，操作系统需要在磁盘中维护一系列数据结构来管理磁盘。可以将磁盘划分成若干个分区，每个分区由一系列 **柱面cylinder** 组成，也可以用逻辑格式化或者构建文件系统的方式来管理磁盘，为了提高项目， 磁盘的 I/O 以块为单位进行，而文件的读写以簇为单位进行。

磁盘在使用过程中会损坏，出现 bad block

* + 1. 交换空间管理

交换空间 (Swap Space) 是指虚拟内存中使用的内存扩展，是磁盘上的一个分区，在 Unix 和 Linux 系统中有单独的交换空间的磁盘分区。

* 1. **RAID** 结构

**RAID Redundant Arrays of Inexpensive (independent) Disks(冗余廉价磁盘阵列)**，是将多个物理硬盘组合成一个逻辑磁盘，用来提供更好的存储性能和数据备份技术，RAID 提高了磁盘fail 所需的平均时间，通过冗余（**性能**）来提升**可靠性**

磁盘的**条带化**是指使用一组磁盘作为一个存储单元，RAID 机制提高了存储系统性能的表现（读写速度提升），并且通过Mirroring 或者 shadowing 来保持每个磁盘的备份，

* RAID0：没有冗余的条带化，提升n倍性能（n是容器个数），如果如果bit坏了都是灾难性的
* RAID1：有镜像磁盘，保证安全性，提升2倍性能
* RAID2：有 Memory-style 的纠错码(ECC校验，汉明码)，只能允许单磁盘错误，并且校验码计算特别麻烦
* RAID3：bit-interleaved parity 奇偶校验。只用一个盘，它的每个行的数值来存之前每行的序列的就校验码。比RAID2快并且占位小，但也只允许一个disk错
* RAID4：按块进行条带化。前几个RAID都是按照bit来条带化的。RAID4方便文件读写
* RAID5：RAID4基础上把奇偶校验均匀分布在每个磁盘上
* RAID6: P+Q 冗余，纠错码

通常情况下会有少量热备盘没有被分配，当其他硬盘失效的时候就会自动替换并在上面重建数据。

注意xy笔记上的非易失性内存和存储介质管理两个大块ws的ppt上都没有，应该不用管

I/O 系统

[Xy的笔记已经足够](https://note.isshikih.top/cour_note/D3QD_OperatingSystem/Unit4/#io-%E6%A6%82%E8%BF%B0)，额外需要注意的是

三种 **I/O** 方式的区别和特点：

轮询非常耗时，因为访问的时候设备可能还在busy，中断可以做到并行，**DMA 不需要CPU** 因此效率高，耗时短，三种 I/O 方式中轮询对硬件设备的要求最低。

**STREAM** 是一个用户进程和 I/O 设备之间的全双控通信通道。包括头接口、驱动器终端接口和若干个

STREAM 模块

提高 **I/O** 性能的办法

* 减少上下文切换、数据的 copy
* 减少中断，使用更大的数据传输或者是更优秀的控制器
* 使用 DMA
* 平衡 CPU，内存，I/O 设备和总线的吞吐量

Linux 操作系统与内核

* 1. **Linux** 操作系统概述
     1. **Linux** 操作系统的基本层次结构

Linux 系统分为用户空间和内核空间，用户空间主要包括各类应用程序和 GNU 的 C 库 (glibc)，内核空间主要包括系统调用接口、内核和与计算机架构有关的内核代码，最底层就是硬件平台，其中：

* + - * glibc 提供了连接内核的系统调用接口，提供了用户态和内核态的切换机制
      * Linux 内核中由系统调用接口、进程管理、内存管理、虚拟文件系统、网络堆栈和设备驱动程序等等
      * Linux 内核是宏内核结构，运行在单独的内核地址空间，并且具有模块化设计，是抢占式的内核，支持内核线程
    1. **Linux** 系统的启动过程
       - BIOS 加载操作系统引导装入程序，由操作系统引导装入程序和加载内核
       - 内核代码的解压缩和初始化
       - 生成 init 进程，系统初始化，生成各种终端进程
       - 用户登录系统
  1. 系统调用

系统调用是内核向用户进程提供服务的唯一方法，应用程序调用操作系统的功能模块，用户可以通过系统调用从用户态(LINUX 3)切换到内核态(LINUX 0)并访问对应的内核资源。

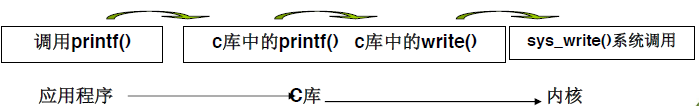
地址空间 space

每个进程的虚拟地址空间又可以分为用户空间和内核空间，在用户态下只能访问用户空间，而在内核态下都是可以访问，内核空间在每个进程的虚拟地址空间中都是固定的。

上下文 context

* + - * 用户级上下午：代码、数据、用户栈和共享存储区
      * 寄存器上下文：通用寄存器、程序寄存器 eip、处理机状态寄存器 eflags，栈指针 eps
      * 系统及上下文：进程控制块，内存管理信息，核心栈等等

系统调用的实现是在内核完成的，而用户态的函数是在函数库中实现的



* 1. Linux 进程管理

Linux 是一个多任务多用户的操作系统，这里说的一个任务就是指一个进程，存放在磁盘上的可执行文件的代码和数据的集合称为可执行映像，一个程序装入系统中运行的时候就形成了一个进程。Linux没有真正的线程的概念，。但Linux通过clone()系统调用支持轻量级(Lightweight process)进程(线程)，两个轻量级进程可以共享如:地址空间、打开文件等资源; Linux还支持内核线程，内核线程永远在内核态运行，没有用户空间。Linux中运行**ps命令**，你能得到当前系统中进程的列表

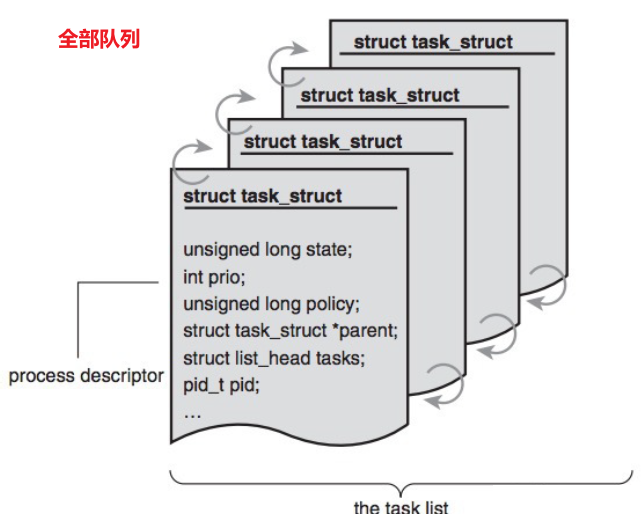
进程由正文段、用户数据段和系统数据段、堆栈段组成，其中：

* 正文段存放着进程要执行的指令代码，具有只读的属性
* 用户数据段时进程在运行过程中处理数据的集合，可以直接访问
* 系统用的数据段存放了进程的控制信息，比如 PCB

**11.3.1 task\_struct 结构体**

这个结构体是 Linux 中的 **进程控制块 PCB** 的具体实现，进程的 task\_structure **是进程存在的唯一标志**， 在一个进程被创建的时候系统就会创建一个 task 结构，在进程结束的时候就会将其撤销，在 Linux 系统中有一个数组来存放所有的task，因此task 的数量有一定的上限，可以用命令**ulimit -u** 来显示用户可以同时执行的最大进程个数。改结构被定义在include/linux/sched.h文件

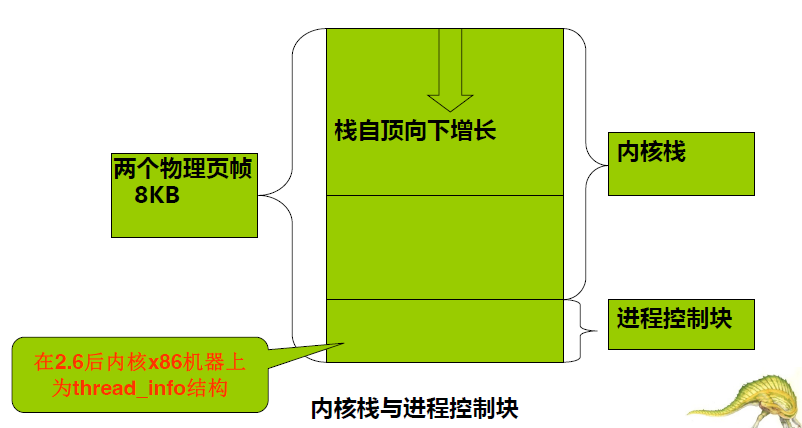
task 结构中进程都有 32 位的无符号整数的标识符 pid，除此之外还有很多奇怪的标识号，比如用户标识符 UID 和组标识符 GID，结构体维护了一系列亲属关系之间的指针，每个进程维护了指向了祖先进程、父进程、子进程和兄弟进程的指针，而进程之间又前后连接，每个进程的 task 结构都有指向上一个进程和下一个进程的task 的指针，此外task 结构还存储了进程的调度信息、时间信息、虚拟内存信息、文件信息、进程通信信息和进程状态等关键信息



**11.3.2 task\_struct 的存放**

进程有两个栈，分别是用户模式栈和内核模式栈，分别在对应的模态下使用，Linux 内核中定义了一个 current 宏来获取当前正在执行的进行，可运行队列的存放方式是一个双向链表，其中第一个进程就是init\_task，可以用 for\_each\_task 宏来遍历整个进程链表。进程的等待队列表示一组正在睡眠的进程。

PCB和进程的内核栈连续分配在一起，内核为它们分配两个连续的物理页



Run队列=cpu的核数

* 1. **Linux** 内存管理

Linux操作系统采用了请求式分页虚拟存储管理方法。系统为每个进程提供了4GB的虚拟内存空间。各个进程的虚拟内存彼此独立。

* + 1. **Linux** 的内存空间

一般来说 32 位寻址空间的Linux 系统**内核空间是 1GB**，**用户空间是 3GB**，每个进程最多拥有 3GB 的地址空间（内核空间和用户空间大小的划分由宏定义PAGE\_OFFSET决定，在文件src/include/asmi386/page.h）



而内核空间由所有的进程共享，其存访的是内核代码和数据 (即内核映像)，不可以被回收或者换出。进程的用户空间中存访的是用户程序的代码和数据，内核空间映射到物理内存总是**从最低地址 (0x00000000)开始**，使之在内核空间与物理内存之间建立简单的线性映射关系，当系统启动的时候，Linux 的内核映像被装入物理地址 (**0x00100000**) 的地方，即 1M 开始的区间。

程序编译之后形成的二进制文件有代码段和数据段，进程运行的时候必须占有独立的堆栈空间。

* + 1. **Linux** 中的内存管理数据结构

Linux 中定义了一系列的结构体来管理内存，大致的架构如下：

**mm\_struct** 这个结构体对进程的整个用户空间进行描述，也被称为内存表述符，而**vm\_area\_structs** 结构对用户空间中的各个区间 (虚存区) 进行描述，这个结构是虚存空间中一个连续的区域，这个区域中的信息具有相同的操作和访问特性，各个区间不重叠，按照地址的次序连接在一起，数目比较多的使用将使用红黑树来保证搜索的速度，并且有指向进程的mm 结构体和虚拟区域开始、终止位置的指针，用vm\_flags 标志这一区域的操作特性。这两个结构体都定义在 include/linux/mm\_types.h这个头文件中。

* + 1. **Linux** 的分页内存管理机制

Linux 总是假定处理器支持**三级页表**的结构，并在 64 位体系结构上采用三级页表的机制，在三级分页管理中虚拟地址分成页目录 PGD、页中间目录 PMD 、页表 PTE 和页内偏移四个部分，2.6.11 版本以后的内核采用了四级页表。

Linux 系统采用的虚拟内存系统通过维护的一系列表，由内存管理单元 MMU 来实现从虚拟地址到物理地址的转换。

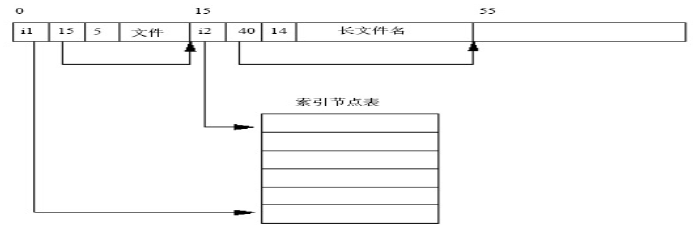
* + 1. **Linux** 的缺页异常

导致缺页的原因可能有编程错误和操作系统故意引发的异常，缺页异常的处理函数是 **do\_page\_fault()** 这个函数有两个参数，一个是指向发生异常时寄存器存访的子 hi 的指针，另一个是错误码，由三位二进制组成，分别记录了访问的物理页帧是否存在，错误的类型 (读写运行错误) 和程序的运行状态 (内核态还是用户态)，可以根据错误码的值确定下一个步骤：

* + - * 如果错误码的值表示为写错误，则需要检查该区间是否允许写，不允许则进行出错处理
      * 如果允许写就属于写时拷贝
      * 如果错误码的值表示页面不存在，就需要进行按需调页
  1. **Linux** 文件系统
     1. 概述

Linux 采用了多级目录的树形层次结构来管理文件，最上层是根目录**/**，Linux 的文件系统不管是什么类型都会安装到一个目录下面，并隐藏掉目录中原有的内容，这个目录就叫做安装目录或者**安装点**。Linux 系统的根目录下面有很多个不同的目录，每个目录下面可能挂载了不同的文件系统。

Linux 文件系统将文件名和文件的控制信息分开管理，文件控制信息单独组成一个**inode 数据结构**，每个文件对应一个 inode，文件系统各个目录中的项主要由文件名和 inode 号组成。



* + 1. **Linux** 文件分类
       - 普通文件：文件名不超过 255 个字符，不能使用保留字进行命名
       - 目录文件：目录 + 目录下面所包含的项组成，只能由系统来修改
       - 字符设备文件和块设备文件（所有的设备都会映射为文件）：将对设备的 I/O 当作文件的读写，提供了同意的接口，如 fd0，tty，hda， sda 等等。显示器、键盘就是字符设备文件
       - 管道文件(pipe)：用于进程之间进行通信
       - 链接文件：又称为符号链接文件，用于文件共享
       - socket 文件：用于进行 socket 通信

**ls -l** 或**ls -ld**命令显示文件的访问权限

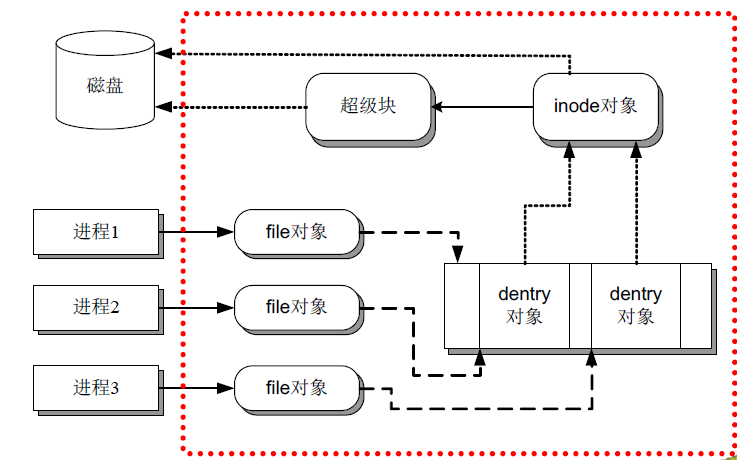
**/proc** 虚拟文件系统，在这里可以获取系统状态信息并且修改系统的某些配置信息。如内存情况在/**proc/meminfo**文件中，使用命令**cat /proc/meminfo**

* + 1. 虚拟文件系统 **VFS**

Linux 的虚拟文件转换系统 (VFS) 存在于内存中，提供了一种用于不同物理文件系统之间的转换机制，像ext2 这些是存在于外存的物理文件系统，VFS 对于不同的物理文件系统提供了统一的API，统一了各种不同的物理文件系统。VFS并不是一种实际的文件系统，ext等物理文件系统存在于外存，而VFS只存在于内存

虚拟文件系统有一个通用文件系统模型，有四种数据对象组成：

* + - * 超级块对象 superblock：对应文件系统的超级块或者文件控制块。超级块对象是文件系统中描述整理组织和结构的信息体，在不同的文件系统安装的时候建立的，在文件系统卸载的时候删除，在建立索引节点的时候就需要操作超级块的对象。
      * 索引节点对象 inode object：对应文件控制块 FCB。虚拟文件系统中也有 inode 结构，但是物理文件系统的 inode 在外存而 VFS 的 inode 在内存中，并且在需要时才会建立，是物理设备上的文件或者目录的 inode 在内存中的统一映像，组成一个双向链表，并且每个摁键除了 inode 之外还有一个目录项 dentry 的数据结构，用于实现文件的快速定位和缓冲作用。
      * 目录项对象 dentry object：**描述各个目录**，组成路径
      * 文件对象 file object：存储打开的文件和进程的**关联信息**



**cp /floppy/TEST /tmp/test**

/floppy是MS-DOS的磁盘的一个挂载点（安装点）

/tmp是ext2文件系统中的一个目录

* + 1. 文件系统的注册

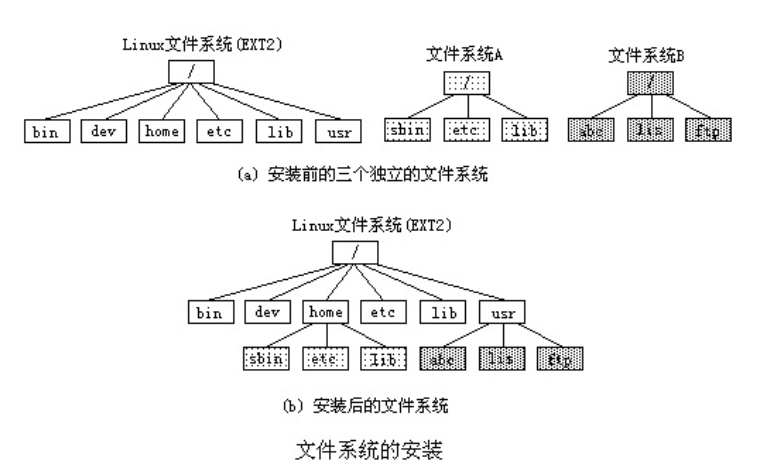
Linux 支持的文件系统必须注册后才能使用，文件系统不再使用时则予以注销。文件系统需要在内核中进行注册，在内核中注册文件有两种方式，一种是在系统引导的时候直接在 VFS 上面注册，一种是将文件系统注册为可以卸载的模块，每种注册的文件都登记在**file\_system\_type** 结构体中，一系列的file\_system\_type 结构体组成一个链表，称为注册链表。链表的表头由全局变量 file\_system 给出，在VFS 中用一个双向链表来进行维护

* + 1. 文件系统的安装和 **mount** 命令

每种文件系统占用一个独立的磁盘分区，具有各自的树形结构，Linux 系统中的文件系统安装要用到

mount 命令，mount 命令的具体用法是：

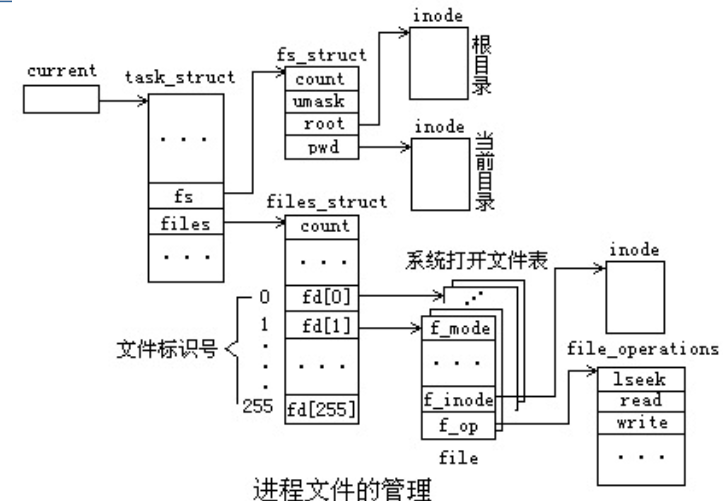
* + - * **mount + [-t fstype] [-o options] device dirname** 比如 mount –t vfat /dev/hda5 /mnt/win
      * fstype 是文件系统的类型，常见的由 ios9660、vfat、ntfs 等等
      * device 是设备文件，dir 是挂载的目录
      * options 是文件或者设备的挂载方式，常见的有：loop 把一个文件当作块设备挂载，rw 采用读写方式挂载



* + 1. 文件系统的管理

对于已经打开的文件可以通过打开文件表和私有数据结构进行管理，系统打开文件表是一个内核中的双向链表，每一项都是一个file 结构体，进程打开一个文件就是建立一个file 结构并将其加入链表，有全局变量指向系统打开文件表的表头。

Linux 系统中的每个进程也会对文件进行管理，**fs\_struct** 记录了进程所在文件系统的根目录和当前目录，**files\_struct** 记录了进程的打开文件表。其中进程的打开文件表中用一个数组来存储所有指向 file 结构体的指针，进程文件管理可以用下面这张图来表示：



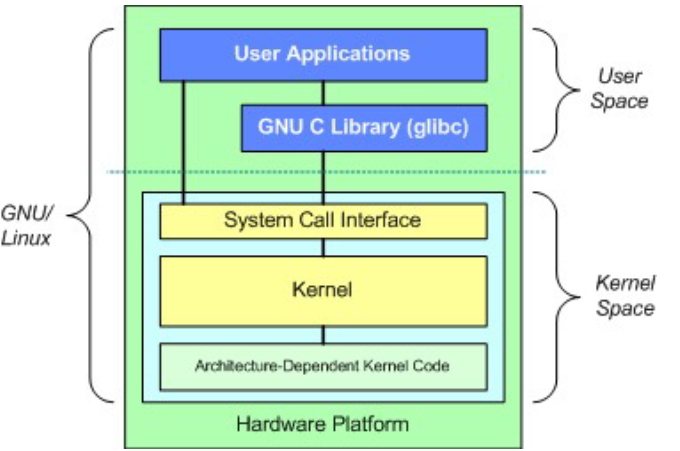
* + 1. **open** 系统调用

Linux 系统打开文件时都需要调用open 系统调用，在内核态对应的内核函数是sys\_open()，需要两个参数，第一个是路径名，第二个参数是文件的访问标志，sys\_open() 的调用过程是：

* + - * 调用 getname() 从进程地址空间读取文件路径
      * 调用 get\_unused\_fd() 找到一个未使用的 fd 位置
      * 调用 filp\_open() 函数，先调用 open\_namei() 函数找到目标节点所对应的 dentry 对象，然后调用

dentry\_open() 申请一个 file 对象的空间并初始化

* + - * 调用 fd\_install() 函数，将文件对象装入当前进程的文件打开表，然后返回文件的 fd
  1. Linux 内核

Linux操作系统的内核是**单内核(monolithic kernel)**的。Linux操作系统使用了一种全新的内核模块机制。用户可以根据需要，在不需要对内核重新编译的情况下，模块能动态地装入内核或从内核移出。

**GUN C Library(glibc)**提供了连接内核的系统调用接口，还提供了在用户空间应用程序和内核之间进行转换的机制。

模块（动态可加载内核模块（LoadableKernel Module **LKM**）在内核空间运行，实际上是一种目标对象文件，没有链接，不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中取下，从而可以动态扩充内核的功能。并且模块不依赖于固定的硬件平台。模块的目标代码一旦被链接到内核，它的作用域

和静态链接的内核目标代码完全等价。

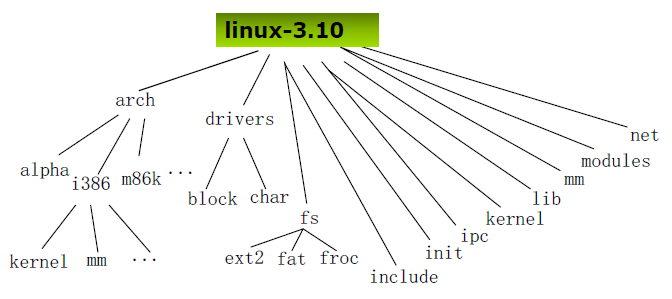
* insmod xxx.ko 安装模块
* lsmod [regex] 查看模块，可以输入一个正则表达式进行匹配
* rmmod xxx.ko 卸载模块
* cat /proc/kallsyms 显示内核符号和模块符号表的信息，可以读取/proc/kallsyms文件

内核模块是如何被调入内核工作的? 1. 当操作系统内核需要的扩展功能不存在时，内核模块

管理守护进程kmod执行modprobe去加载内核模块。2. modprobe遍历文

/lib/modules/$(version)/modules.dep 来判断是否有其它内核模块需要在该模块加载前被加载。3. 最后modprobe调用insmod先加载被依赖的模块，然后加载该被内核要求的模块。

Linux内核代码可以安装在~/linux-x.x.x目录下，



一些题目

1. A user-level process cannot modify its own page table entries √
2. Remote Procedure Calls provide identical semantics to local calls. ×