操作系统 Operating System

RandomStar

2021 年 1 月 26 日

目录

1 导论 Introduce

1.1 什么是操作系统?

1.1.1 操作系统的定义

Definition 1.1 操作系统是用户和计算机硬件之间的中间程序，可以用于执行用户程序，使计算机易于使用，充分调用计算机的资源

操作系统从不同的角度 (用户视角、系统视角、软件分层) 看来有不同的作用，并且承担着不同的角色:

- 用户视角：操作系统是用户和计算机硬件之间的接口，接口又分为命令级接口和程序级接口
- 系统视角：操作系统是计算机系统资源的管理者，可以执行程序、分配资源
- 从软件分层的观点来看，操作系统是扩充裸机的第一层系统软件
- 计算机中一直运行着的程序就是操作系统的**内核 (kernel)**

1.1.2 操作系统的特性

操作系统的设计目的是易于使用、注重速度和资源的利用率。不同的操作系统可能具有如下这些不同的特性：

- 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 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
- 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
- multi-tasking 多任务系统，一次可以执行多个任务

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

1.2 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。中断是通过中断向量来实现的，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字寄存器中的内容，但是子程序调用不会保存。

1.3 双模态操作 dual-mode operation

- 用户态 (user mode)：操作系统在执行用户程序时候的模式，，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)：执行操作系统程序时候的模式，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、外围设备的中断

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。

1.4 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

1.5 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

1.6 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

- 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

- 操作系统的 API 封装了一系列的系统调用，比如 POSIX API 和 Java API
- 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*

1.7 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。

2 进程和线程 Process&Thread

2.1 进程

2.1.1 进程的定义

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**，类似的概念还有 job、task、user program 等等。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
- 程序计数器 program counter
- 寄存器
- 数据区，存储全局变量
- 堆，存放动态分配的变量
- 栈区，存放临时变量

2.1.2 进程的状态

- 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

值得注意的是，进程的不是任意两个状态之间都可以随意切换的，比如处于 waiting 状态的进程，不能直接变成 running，需要先变成 ready 之后再等待 CPU 空闲才可以变成 running 状态。

running 状态的进程发生了中断会转化成 ready 状态，而如果等待 I/O 或者特定事件就会变成 waiting 状态，我自己的理解是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。

就拿一个正在上班的码农作为例子，平时上班就是 running，如果退休了或者加班猝死了就是 terminated，如果自己想要更高的学历或者更高的工资待遇或者想要远离加班而主动辞职，那就是转换成了 waiting，这表明你在等待更好的机会，而如果是被血汗工厂强制辞退或者优化了，自己其实还可以正常干活，那就是转换到了正在待业的 ready 状态，只要有工作了随时可以上岗，waiting 需要一段时间的沉淀才能再次变成 ready，而不能直接从 waiting 又变回 running，因为知道自己还没有准备好。而 CPU 的调度并不关心你这个进程，而只追求最高的 CPU 利用率，尽量减小空转的时间，它不会专门为一个特定的进程来服务，除非这个进程优先级足够高，可以一直占用 CPU，但是这会使得其他进程一直处于 starvation 状态。

2.1.3 进程控制块

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，寄存器信息，调度信息，内存管理信息，I/O 统计信息等信息

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

2.1.4 进程的调度

进程的调度需要在 ready，running 和 waiting 三种状态之间切换，进程的调度可以分为

- 长程调度：又叫作业调度，已经被现代操作系统抛弃了
- 短程调度：又叫 CPU 调度，CPU 可以选择下一个执行的进程
- 中程调度：从磁盘调度到主存中

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 CPU 型进程

Definition 2.1 *上下文切换：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。*

2.1.5 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**

Definition 2.2 *fork() 函数是 Linux 下创建进程的系统调用，可以将当前进程分为两个进程，一个是父进程，一个是子进程。*

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

```
1 int main() {
2     int pid1 = fork();
3     if(pid1 < 0) {
```

```
4           cout << "Failed!" << endl;
5       } else if(pid1 == 0) {
6           // chidren proccess
7       } else {
8           // parent process
9       }
10  }
```

Theorem 2.2 *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。

2.1.6 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。

Theorem 2.3 *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。

2.1.7 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

Definition 2.3 僵尸进程 *(zombie process)*：子进程终止但是父进程尚未调用 *wait*，此时子进程就变成僵尸进程

Definition 2.4 孤儿进程 *(orphan process)*：父进程终止了但是还没有调用 *wait*，导致子进程成为孤儿，并且没有孤儿院

2.1.8 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独立进程和合作进程。而进程间的通信又分为直接通信和间接通信。

Definition 2.6 共享内存模型和消息模型是两种常见的进程通信模型。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

2.2 线程

2.2.1 线程的定义

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

2.2.2 多线程模型

- 用户级线程：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：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

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：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：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：多个用户线程对应多个内核线程，比如 Windows NT/2000

此外还有二级模型，就是多对多 + 一对一的缝合版。

3 CPU 调度

3.1 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

Theorem 3.1 调度发生的时间节点：从运行切换到暂停的时候，从运行切换到就绪的时候，从等待切换到就绪的时候，进程终止的时候

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：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：从进程提交到首次被响应所需要的时间
  - ★ 等待时间 Waiting Time：进程在就绪队列中等待的时间之和 (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

3.2 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，按照进程的提交顺序来安排 CPU 的调度，先来的直到结束或者阻塞才让出 CPU，特点是实现比较简单。**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
- HRRN 算法：最高响应比优先，是 SJF 的变式，响应比 R=(等待时间 + 要执行的时间)/要执行的时间
- 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：优先级高的来了会直接武装夺取 CPU 使用权

- 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 slice，在时间片结束之后会发生中断，将改进程切换到**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
- 多级队列调度：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

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。是进程的阶级固化导致的后果，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。

4 同步 Sychronized

4.1 基本定义

Definition 4.1 并行的进程或者并发的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，这时候就需要一定的机制来维护

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。

一个 count++ 的操作翻译成汇编指令可能是执行了下面这样一系列操作

```
register = counter;
register = register + 1;
counter = register;
```

Example 4.1.1 有两个进程 *P1,P2* 分别执行下面的程序，其中 *total* 是两个进程的共享变量且初始值为 *0*，假设两个进程并发执行，并且可以自由交叉，则进程结束之后 *total* 的可能最小值是多少？

```
P1: {
    for (int count = 1; count <= 50; count++) {
        total += 1;
    }
}
P2: {
    for (int count = 1; count <= 50; count++) {
        total += 2;
    }
}
```

分析：上面提到的汇编指令操作中，到最后一步才会把寄存器中的值写回 total 中，这要进行合理控制，使得一个进程结束之前另一个进程运行 49 次，这样一来这 49 次就不会影响 total 的最终结果，因为两个进程分别用两个不同的寄存器对 total 进行操作，因此最终影响到 total 的可以只有一次 +1 和一次 +2，最后结果就是 3

4.1.1 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
- 饥饿 Starvation 资源分配不公平导致进程长时间得不到资源

4.1.2 临界资源和临界区

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，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

4.2 面包房算法 Bakery Algorithm

处理 N 个进程的同步，进入临界区之前，进程会收到一个标号 number，标号最小的进程有权利进入临界区如果两个进程有相同的标号就按照 FCFS 原则，共享的数据区包含一个 bool 类型数组 choosing 和 int 型数组 number，若 choosing[i]=true 表示进程 i 正在获取编号。

4.3 信号量 Semaphores

信号量是一种数据类型，只有初始化，wait 和 signal 三种操作，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

Theorem 4.1 记录型信号量的值可以是任意的正整数，表示资源的个数。二值信号量的值只能是 *0* 或者 *1*，也叫做同步互斥锁。

信号量的数据结构可以表示为

```
1 typedef struct {
2     int value;
3     struct process *list;
4 } semaphore;
```

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**: *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

```
wait(semaphore *S) {
    S->value--;
    if(S->value) {
        add to S->list;
        block();
    }
}

singal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove from S->list;
        wakeup(P);
    }
}
```

4.4 同步的三个经典问题

- 生产-消费者问题：也叫做有限缓冲区问题，主要形式就是生产者写入缓冲区，消费者读取缓冲区

  ★ 十多个协作进程之间通过消息队列进行通信的过程的抽象

  ★ 假设缓冲区有 N 个，需要设置三个信号量 mutex=1,full=0,empty=N

- 读者-写者问题：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。第二读写者问题会让写者就绪的时候尽可能多地满足写者的要求。

- 哲学家吃饭问题：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，总之是个很莫名其妙的问题。

5 死锁 DeadLock

5.1 基本概念

5.1.1 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。

5.1.2 产生死锁的必要条件

产生死锁有四个必要条件：

- 互斥：一个资源在一段时间里只能被一个进程使用
- 占有并等待：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：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：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

5.2 资源分配图 Resource Allocation Graph

5.2.1 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，请求边从 P 指向 R，表示进程请求这个资源，分配边从 R 指向 P，表示资源被分配给这个进程。

5.2.2 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：

- 没有环的肯定没有死锁
- 有环并且可以用的只有一个实例，产生死锁
- 有环但是还有多个可以分配的实例，就不会产生死锁

5.3 死锁的处理

处理死锁的三种方法：在程序的开发阶段预防和避免死锁，在程序运行的时候进行死锁的检测和解除，鸵鸟方法，即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

5.3.1 预防死锁

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

5.3.2 避免死锁

在分配资源之前先判断是否可能产生死锁，如果可能产生死锁就先不分配资源，具体的算法有资源分配图算法和银行家算法。

5.4 死锁的避免算法

5.4.1 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。*

如果系统处于安全状态，很明显就不会产生死锁，否则就有可能产生死锁，而避免死锁的策略就是让系统永远处于安全状态，实现这个目标需要一定的算法，对于每个资源只有一个实例的情况可以使用资源分配图算法，对于每个资源有多个实例的情况可以使用银行家算法。

5.4.2 资源分配图算法

Definition 5.2 *资源分配图算法：用虚线来标识需求边，在一定条件下可以转换成请求边，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。*

5.4.3 银行家算法 Banker's Algorithm

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
- max[i,j]=k 表示第 i 个进程需要资源 j 的 k 个实例
- 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
- 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 $1 \leq i \leq n$
2: 找到一个下标 i 使得 finish[i]=false 并且 $need[i] \leq work[i]$ 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

5.5 死锁的检测

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。

6 主存 Main Memory

6.1 基本概念

一个刻在 DNA 里的常识：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，冯诺依曼体系下的计算机的存储层级如下图所示：

6.1.1 可执行程序如何产生

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：

- 编译：由编译程序将用户的源代码编译成若干模块
- 链接：由链接程序将编译后形成的一组目标模块和所需的库函数链接在一起，形成一个完整的可装入模块，这一过程中会形成程序的逻辑地址。
- 装入：由装入程序将装入模块装入内存中运行

6.1.2 逻辑地址和物理地址

逻辑地址也叫做相对地址和虚地址，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。物理地址也叫实地址和绝对地址，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，进而确定一个逻辑地址空间。

6.1.3 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 种，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。

6.1.4 载入和链接

程序的链接有三种方式：

- 静态链接：生成可执行程序的时候就和所需库函数进行链接，之后不再改变
- 装入时的动态链接：一边装入内存，一边进行动态链接
- 运行时动态链接：执行过程中在进行链接

而装入也有三种方式：

- 绝对装入：在编译的时候产生绝对地址的目标代码，按照这个绝对地址将程序装入内存，只适用于单通道环境

- 可重定位装入：也叫做静态重定位，可以根据内存使用情况修改装入时的起始位置，但是装入之后就不能改变
- 动态运行时装入：也叫做动态重定位，在执行的时候才进行逻辑地址和物理地址之间的转换

通过使用扩动态载入，扩展的库只在需要的时候装载入内存而不是在进程的一开始就装载进入内存，提高了内存空间的利用率。而动态链接会将库文件预装载入内存中，当需要调用库函数的时候会提供动态链接库的物理地址。

6.2 交换技术

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。

移动操作系统不支持交换，因为是基于闪存，有写次数的限制

6.3 连续分配

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：

- 基寄存器保存最小的物理地址
- 有限寄存器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

6.3.1 分区式管理

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。所谓的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
- Next-Fit 类似于 First-Fit，但是搜索的位置从上一次查找结束的地方开始，找到了一个足够大的区域就分配

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

6.3.2 内存碎片 Fragmentation

在分区式管理中会产生很多内存碎片，一种是一个进程内部的空闲的内存，称为内部碎片 (inner fragmentation)，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来进行管理，但是消灭内存碎片需要非常大的 cost

6.4 页式存储管理

6.4.1 页

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。页式存储管理中，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。

Theorem 6.1 *外存中也和内存一样划分出若干相同大小的块 (block)，为了方面地址的转换，大小都是二的幂次。*

6.4.2 地址结构

分页式的存储管理的逻辑地址结构由页号 P 和页内偏移量 W 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

6.4.3 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 来存储页表的起始地址 F 和页表长度 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。

6.4.4 快表 TLB 和有效访问时间

快表 (TLB，translation look-aside buffers) 也叫联想寄存器，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(计算机系统原理里面学过 cache 没有命中就叫做失配，TLB 同样也会失配)，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：

$$EAT = (x + y)\alpha + (x + 2y)(1 - \alpha) \tag{1}$$

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

6.4.5 页表的类型

x86 架构中，逻辑地址是 32 位，而页的大小是 4KB，则很显然页表项最多有 1M 个，这就需要每个进程的页表需要连续的 4MB，这很明显是不现实的，因此我们需要对普通的页表进行改进，常见的页表有如下几种类型：

- 分级页表 Hierarchical Page Table，常见的有 Linux 系统的四级页表和 Windows 的二级页表，基本的原理是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

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

6.5 段式存储管理

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。

7 虚拟内存 Virtual Memory

7.1 基本概念

7.1.1 局部性原理

计算机中的两个局部性原理：空间的局部性和时间的局部性，意思是内存中的一段内容被访问，那么它附近的一些内容被访问的可能性非常大，而接下来的时间里这一段内容被访问的可能性也非常大。

7.1.2 虚拟内存的基本概念

一个进程的地址空间中，从高到低依次是栈区、堆、数据和代码区，而在整个内存中更高的地址空间有操作系统的内核程序和环境变量，以及各种参数。而这些内存被划分成了若干不同的帧。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) 和请求段式管理两种。

7.2 按需调页

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。

7.2.1 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) 来标识当前页是否有效，如果无效表明这一页没有被调入内存中，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

7.2.2 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

$$EAT = (1 - p) * x + p * y \tag{2}$$

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，缺页中断服务在操作系统中被细分成了十多个步骤，这里就不一一深究了。

7.2.3 进程创建

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。

7.3 页的置换 Page Replacement

当没有空闲的物理帧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。

7.3.1 页面置换算法的流程

页面置换的时候首先需要找到磁盘上需要换进内存的页面，然后去内存中找到一个空闲的帧或者可以被淘汰的帧，如果选择了淘汰的帧就需要将其写入磁盘中，然后修改页表，之后将需要换如的帧换入内存并修改页表，然后启动进程。

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**修改位**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。

7.3.2 页面置换算法

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 最近最久未使用算法：将最近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，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
- 近似 LRU 算法，有如下几种实现方式

★ 可以考虑使用一个引用位，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

★ 附加引用位算法 (Additional-Reference-Bits Algorithm)：每页使用一个 8bit 的引用位，被访问的时候左边的最高位设置成 1，定期将所有页的引用位右移并在最高位补 0，数值最小的就是被替换的

★ 二次机会算法，也叫 clock 算法，需要一个引用位，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

★ 增强二次机会算法，使用一个引用位和一个修改位，替换优先级 (0, 0)-(0, 1)-(1, 0)-(1, 1)

- LFU 最不经常使用算法
- MFU 最多引用算法
- Page Buffering 页面缓冲算法：通过被置换的页面的缓冲，有机会找回刚才被置换的页面，在 VAX/VMS 系统中使用

7.3.3 如何减少换页的次数

复习的时候遇到一道题，说一个按需调页的操作系统中，如果 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，因此解决的方法是换用更大的物理内存，或者降低多道程序的 degree，这样才能让每个进程获得更多的物理内存，而其他方法，比如换 CPU，换磁盘都是没有用的。

7.4 零散的点

7.4.1 帧分配

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，平均分配算法就是每个进程分到等量的帧，按比例分配就是按照进程的大小来分配相应的物理帧
- 优先级分配：按照一定的优先级进行帧的分配

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，而局部置换只能置换该进程拥有的帧。而帧的分配策略有上述提到的固定和优先级分配，和帧的置换策略进行组合之后形成了三种不同的置换和分配策略：固定分配局部置换、可变分配全局置换、可变分配局部置换

7.4.2 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。

7.4.3 内存映射文件

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。

7.4.4 内核内存分配

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，一些内核内存必须要是连续的。

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。

7.4.5 预调页

在程序开始运行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，就好像选调生选了一大批只想养老而不干事的人一样。

8 文件系统 File System

8.1 文件系统接口

8.1.1 文件

文件是存储在某种介质的并且具有文件名的一组信息的集合，分为数据文件和代码文件。文件有很多的属性，包括文件名、文件类型、文件大小等等。操作系统中的文件也是一种抽象数据类型，可以进行创建、读写和删除，文件具有一定的结构，可以分为以下三种

- 流文件结构：字符流或者字节流
- 记录文件结构；按照行或者定长、变长的段来存储文件
- 复杂结构：格式化文档、可重定位加载文件等等

8.1.2 文件的访问方式

打开文件需要：文件指针，指向上次的文件读写的位置；文件打开次数的计数器，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中删除这个文件名；还需要知道文件的磁盘位置和拥有访问权限。而文件的存取方式又可以分为顺序存取，直接存取。

- 顺序存取：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：可以直接读写 n 个字符

8.1.3 文件目录

磁盘可以划分为若干个分区，每个分区都有目录和文件，而目录可以用于搜索、创建、删除和重命名一个文件，并遍历所有的文件，目录可以分为单级目录、多级目录、树形目录、无环图目录、通用图目录等等。

树形目录的优点是搜索速度快，拥有一个根目录，搜索的路径可以分为绝对路径和相对路径两种。而无环图目录允许 **不同的项目有共享的子目录和文件，同一个文件或者目录可以出现在两个不同的目录中**

共享文件的实现方式是创建一个链接，链接实际上是一种指向另一个文件或者目录的指针，但是无环图中虽然允许创建链接但必须做到无环，删除链接并不影响原本的文件而只是删除这个链接，当原文件被删除的时候链接也就失效了。链接信息的维护可以使用一个文件引用表，在删除所有的链接之前不能删除文件，UNIX 系统中采用了硬链接的方式在 inode 号中记录文件的被引用量。

8.2 文件系统的实现

8.2.1 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，文件系统时操作系统中以文件方式管理计算机软件资源以及被管理的文件和数据结构的集合。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：知道文件对应的逻辑块和物理块
- 基本文件系统：向合适的设备驱动程序发送一般命令就可以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

8.2.2 常见的文件系统

- FAT 是 MS-DOS 系统使用的文件系统，FAT32 是 Windows98 中使用的文件系统，NTFS 是最新的 Windows 文件系统
- ext 系列是 Linux 操作系统中使用的文件系统
- NFS 是网络文件系统
- VFS 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接口，即系统调用
- Linux 内核使用一个 proc 伪文件系统

8.2.3 文件控制块 FBC

文件控制块 (File Control Block) 是文件系统中的一个重要数据结构，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。

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。

8.2.4 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。

8.2.5 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：每个文件分配一系列连续的磁盘块，优点是简单但是浪费空间
  - ★ 允许块的随机访问，实现非常简单但是浪费空间
  - ★ 文件的 size 不能增长否则就要换块，可以采用 defragmentation 机制
  - ★ 存在一个逻辑地址向物理地址转换的机制，可以用逻辑地址/块大小得到块号和块内便宜
  - ★ 可以基于长度进行分配

- 连接分配：每个文件由若干个磁盘的 block 组成，这些 block 串成一个链表，实现方法是每个 block 包含数据和指向下一个 block 的指针
  - ★ 不允许随机访问，访问的时候必须按照链表上排列的顺序访问
  - ★ 每个 block 中数据的量要减去指针的长度
  - ★ FAT 文件系统就是使用了连接分配，比如 FAT32 有一个引导区，文件分配表维护了一系列文件的链表
  - ★ 具体的 FAT 相关内容好像比较复杂，考试应该不会考，先不复习了
- 索引分配：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
- 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

9 大容量存储系统 Mass-Storage Systems

9.1 常见的大容量存储器

计算机中的存储器结构分成三个等级，这件事情已经是多次强调过的了，寄存器，cache 和内存是易失存储，位于第一级，而第二级存储主要有非易失性内存和磁盘驱动器，第三级存储主要有光盘和磁带，这些存储器都是非易失的。常见的大容量存储器和特点有：

- 磁盘：可拆卸，通过总线连接到计算机，总线的类型很多，有 EIDE,ATA,SATA,USB,SCSI 等等
- 固态驱动器 SSD，也叫做固态硬盘，由控制端元和存储单元 (FLASH 或者 DRAM) 组成
- 磁带，Liner Tape File System(LTFS) 为磁带提供了一种通用而开放的文件系统

9.2 磁盘 Disk

9.2.1 磁盘的结构

磁盘的地址被分为若干个逻辑块，组成一个一维数组，一般逻辑块的大小是 512 字节，逻辑块的数组被映射到扇区中，扇区 0 是第一个磁盘上的第一个山区，磁盘由多个 cylinder 组成，每个 cylinder 又分为多个磁道 track，每个磁道又分成多个扇区 sector，使用一个读写头来进行信息的读写。当然还有很多其他的存储技术，包括 Host-attached storage 主机连接存储、Network-Attached Storage 网络连接存储和虚拟化存储技术等等。

9.2.2 磁盘的访问

磁盘的访问时间主要受三个因素的映像：

- 寻道时间 Seek time 是将读写头移动到对应扇区中所需要的时间
- 旋转延迟 Rotational Latency 是扇区旋转到读写头对齐所需要的额外的时间
- 传输时间 Transfer Time 是完成读写所需要的时间

其中旋转延迟可以根据磁盘的转速来计算，一般情况下按照平均需要旋转半圈来计算旋转延迟

9.2.3 磁盘的调度算法

操作系统为磁盘的连续访问提供了一系列的调度算法，包括：

- FCFS 先来先服务算法：按照访问的顺序一个个调度磁盘
- SSTF 最短寻道时间优先算法：每次都找离上一次调度最近的磁盘 (即寻道时间最短)
- 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

- C-SCAN 循环扫描算法：和扫描算法的区别在于扫描触底之后立马从一端转移到另一端，然后开始扫描，等于说将磁盘是以循环的排列方式来扫描
- LOOK 算法：SCAN 的优化，再往一个方向扫描的时候最后一个磁盘访问结束就立马返回，不触底
- C-LOOK 算法：C-SCAN 算法的优化，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

扫描类算法在大规模的系统中运用的时候效果更好，磁盘调度算法的性能取决于磁盘访问请求的数量和类型。

9.2.4 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 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

9.3 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 Windows 系统中用 pagefile.sys 文件来维护交换空间的信息，而在 Unix 和 Linux 系统中有单独的交换空间的磁盘分区。

9.4 RAID 结构

RAID 结构的全称是 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，RAID 可以分为 0-5 级。

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，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
- RAID1：有镜像磁盘
- RAID2：有 Memory-style 的纠错码
- RAID3：bit-interleaved parity 奇偶校验
- RAID4：按块进行条带化
- RAID5：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。

10 I/O 系统

I/O 设备的日益多样化和软硬件接口的标准化的矛盾愈演愈烈，因此操作系统内核设计城使用设备驱动程序模块的结构，为不同的 I/O 设备提供了统一的接口。I/O 系统有总线 I/O 系统和主机 I/O 系统两种组成。I/O 设备具有一定的寻址方式，被直接 I/O 指令和内核映射 I/O 所使用。

10.1 I/O 方式

10.1.1 轮询 Polling

CPU 定期检查设备的状态，来确定是否到了下一次 I/O 的时间，设备具有一定的状态 (ready, busy, error 等等)，可以用状态位来标识，CPU 访问的时候就直接查看状态位来了解设备的情况。

10.1.2 中断 Interrupt

CPU 有一条**中断请求线**，由 I/O 设备出发，设备控制器通过中断请求线发送信号而引起中断，不会检查接口设备的状态位。

中断请求可以分为可屏蔽 maskable 和非屏蔽 unmaskable 两种，非屏蔽的中断请求用于处理不可恢复内存等事件，可屏蔽中断可以由 CPU 在执行关键的不可中断指令序列前面加以屏蔽。系统里的中断由一定的优先级。高优先级的中断可以抢占低优先级的中断，中断可以用于处理各种异常，系统调用就是中断驱动的。

10.1.3 直接内存访问 DMA

不需要 CPU 参与，但是需要 DMA 控制器，内存和 I/O 设备进行数据的传输。

10.1.4 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。

10.2 杂七杂八的东西

10.2.1 系统调用和设备驱动

I/O 相关的系统调用位不同设备的 I/O 提供了统一的接口，驱动设备对内和隐藏了不同 I/O 控制器的差异。

10.2.2 块设备和字符设备

I/O 设备被 Linux 作为块设备或者字符设备处理，块设备包括磁盘驱动器，可以进行 read，write 和 seek 等操作，允许 raw I/O 和文件系统访问，已经内存映射我呢见访问。字符设备包括鼠标键盘和串口，包含 get 和 put 等命令。

10.2.3 阻塞和非阻塞的 I/O

阻塞的 I/O 是在 I/O 完成之间进程一直挂起，而非阻塞的 I/O 会立即返回，比如 buffered I/O，可以通过多线程机制来实现，而异步的 I/O 指的是进程和 I/O 操作同时运行。

10.3 内核中的 I/O 子系统

内核中的 I/O 子系统负责文件和设备命令名的管理、访问的控制，文件系统空间的分配，缓冲，I/O 调度，设备状态监控，驱动程序的配置和初始化。

10.3.1 I/O 调度

操作系统通过为每个设备维护一个请求队列来实现调度，调度方式根据实际的需要确定，可以 FCFS 也可以使用优先级调度，也可以用缓冲、高速缓冲、假脱机等技术实现。

10.3.2 缓冲

解决了 CPU 和 I/O 设备的速度失配问题，需要管理缓冲区的创建、分配和释放。分为单缓冲、多缓冲、缓冲池等等。

10.3.3 假脱机 SPOOLing

可以用来保存设备中的输出的缓冲区，比如打印机不能接受交叉数据流，就可以使用 SPOOLing 使得多个电脑可以同时使用一台打印机。

10.4 文件的读取

进程从磁盘中读取一个文件的过程：确定保存文件的设备，转换名字到设备的表示法，把数据从磁盘读到缓冲区中，通知请求进程数据现在是有效的，把控制权返回给进程。

10.4.1 STREAM

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

10.5 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

11 Linux 操作系统与内核

11.1 Linux 操作系统概述

11.1.1 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

11.1.2 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

11.2 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切换到内核态并访问对应的内核资源。

11.2.1 地址空间

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。

11.2.2 上下文

- 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

11.3 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。

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。

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、紫禁城和兄弟进程的指针，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。

11.4 Linux 内存管理

最早的操作系统采用实模式来进行内存管理，即直接使用物理地址，而操作系统发展到了现在都采用了保护模式，即使用虚拟地址来进行内存的管理，一系列地址的转换机制是：程序编译生成的二进制文件中有包含了逻辑地址的机器语言指令，这些逻辑地址通过段机制转换成虚拟地址，虚拟地址一般都是一个无符号 32 位整数。

11.4.1 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，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。

11.4.2 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 mm_types.h 这个头文件中。**

11.4.3 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。

11.4.4 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

11.4.5 物理内存的管理

11.5 Linux 文件系统

11.5.1 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。

11.5.2 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：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：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

11.5.3 虚拟文件系统 VFS

Linux 的虚拟文件系统 (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。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：对应文件系统的超级块或者文件控制块

- 索引节点对象：对应文件控制块 FCB
- 目录项对象：描述各个目录，组成路径
- 文件对象：存储打开的文件和进程的关联信息

超级块对象是文件系统中描述整理组织和结构的信息体，实再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。

虚拟文件系统中也有 inode 结构，但是物理文件系统的 inode 在外村而 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。

11.5.4 文件系统的注册

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，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，具体的太细了我觉得没必要了解的这么清楚。

11.5.5 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
- 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

11.5.6 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：

11.5.7 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

12 Linux 内核实验

听说实验中用到的各类指令考试的时候也会考到，所以这里记录一下，防止忘记

12.1 添加内核模块

好像没什么特别要记住的，主要就是内核模块编译后的.ko 文件可以安装、查看和卸载，具体的指令有如下几条：

- 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

12.2 添加系统调用

实验中用到的一些重要指令如下：

- make mrproper 清除目录下所有配置文件和之前产生的.o 文件
- make menuconfig 进行.config 文件的配置
- sudo make modules 编译各个可加载模块，再用 make modules_install 来安装到标准的模块目录
- sudu make install 安装编译好的内核模块
- tar -xvf linux-4.8.xz 解压压缩包
- xz -d patch-3.13.xz | patch -p1 给内核源码打补丁
- dmesg 查看内核日志，也可以到/log/kern 目录下去查看

12.3 添加一个加密文件系统

实验中用到的一些重要指令如下：

- cat /proc/filesystems | grep myext2 查看文件系统是否加载成功
- 挂载和卸载文件系统的一系列操作：
  - ★ mount -t myext2 -o loop ./myfs /mnt 挂载文件系统
  - ★ mount | grep /mnt 查看目录下面挂载的文件
  - ★ umount /mnt 卸载 mnt 目录下面的文件系统
- dd if=/dev/zero of=myfs bs=1M count=1 文件数据读写命令，if 和 of 分别表示输入输出文件，bs 表示读出块的大小，count 表示块的个数

- /sbin/mkfs.ext2 myfs 的意思是执行了一个创建文件的 shell 脚本

实验中的一些细节：

- 修改内核代码时候用到的一些内核源码头文件：
  - ★ /lib/modules/$(uname -r)/build/include/asm-generic/bitops.h 中添加 asm-generic/bitops/myext2-atomic.h
  - ★ /lib/modules/$(uname -r)/build/arch/x86/include/asm/bitops.h 中添加 asm-generic/bitops/myext2-atomic-setbit.h
  - ★ /lib/modules/$(uname -r)/build/include/uapi/linux/magic.h 中添加宏定义 MYEXT2_SUPER_MAGIC 0xEF53
- magic number 位于文件 magic.h 中，而这个文件在内核中的位置是 include/uapi/linux/magic.h
- 这个实验的第五步要修改加密解密的相关函数，值得注意的是操作文件数据的时候要先从用户态读入内核态，再拷贝到用户态中，并需要修改 file_operations 结构体