# Overview and History

1. What are the Operating System Driven Factors?

大部分操作系统的历史都是由硬件和人的相对成本要素驱动和影响的。起初，硬件的开销要比人大的多，但从那时到现在，相对成本已经降低了。相对成本产生操作系统的目标。

起初：硬件昂贵，人力廉价→最大化利用硬件。

现在：硬件便宜，人力昂贵→让人使用电脑变得简单。

1. 在电脑很昂贵的时候出现的问题：
   1. 当程序员完成设置后，计算机是空闲的，高投资的利用率（utilization）却很低。
   2. 在任意给定的时间内，计算机的工作是由CPU与I/O设备其中一个来完成的，这样就导致剩下的部分的闲置
   3. 没有任何一项工作能使CPU和I/O设备都充分的工作，CPU和I/O设备的利用率都不高
2. 当电脑变得较便宜，新的问题：
   1. 让电脑更容易使用并提高人们的生产效率变得很重要。但是输入输出与电脑的不可交互性导致你必须等待电脑输出完才能继续工作。所以交互性变得十分重要！
      1. 解决：交互式的处理
   2. 以前的计算机调度者们致力于让一项任务尽可能的在CPU空闲状态下一样工作。不过现在，人们需要一个合理的计算机响应时间
      1. 解决：抢占式调度
   3. 人们在使用计算机的时候需要将数据和程序保留下来
      1. 解决：加入文件系统使得能够快速的使用数据。
   4. 当巨大的程序需要CPU运行时会得到一个非常糟糕的响应时间，因为机器会处于超负荷状态
      1. 解决：优先处理巨大的程序，大的程序获得更多的资源比那些小程序
3. 当电脑变得更便宜
   1. 人手一台电脑
4. What does a Modern Operating System do?

* Provides Abstractions 提供抽象方法
* Provides Standard Interface 提供标准接口
* Mediates Resource Usage 调节资源使用
* Consumes Resources 消耗资源

**Provides Abstractions：**

硬件的底层物理资源具有复杂而特殊的接口，操作系统提供抽象方法来描述这些接口

目标：使计算机的使用变得简单

示例：进程，无界内存文件，同步和通信机制

**Provides Standard Interface：**

目标：可移植。

**Mediates Resource Usage：**

目标：允许多个用户公平的、高效的、安全的共享资源

**Consumes Resources：**

操作系统消耗资源以提供上述功能。

1. What are the OS like?

操作系统是复杂的软件

1. 并发和异步使操作系统成为非常复杂的软件
2. 从根本上讲，操作系统具有不确定性，是事件驱动的
3. 操作系统很难构建，不可能完全的被调试
4. 操作系统很大，没有人能够完全理解整个系统，他比任何一个系统建造者存在的时间都长。

# Processes and Threads

1. 什么是进程

进程是一个在特定进程状态下的执行流

1. 执行流是一系列的指令
2. 进程状态决定了指令的作用
3. 进程是独立的：没有一个进程可以直接影响另一个进程的状态
4. 进程状态通常包含（并不仅限于）下列内容：
5. 进程状态的组成
   * 1. Register 寄存器
     2. Stack 栈
     3. Memory 内存
     4. Open file tables 打开文件表
     5. Signal management information 信号管理消息
6. 进程运行时的状态
   1. New 新建
   2. Running 运行
   3. Waiting 等待：进程等待某个事件的发生（I/O完成收到信号）
   4. Ready 就绪：进程等待分配处理器
   5. Terminated 终止：进程完成执行
7. 单进程系统
   1. 一个时间段内只有一个进程。如DOS。
   2. 问题：用户通常希望同一时间可以进行多个活动，但是单进程系统无法做到
   3. 单进程系统把东西放入诸如内存常住程序，以异步方式调用，但是始终有分离问题
   4. DOS系统的一个关键问题是没有内存保护机制——一个程序有可能向其他程序所使用的内存块中写入数据，导致预计的BUG
8. 多进程系统
   1. 同一时间可以有多个进程存在，允许系统清晰的区分不同活动区
9. 多进程系统和资源共享
   1. 让那个进程去使用机器物理资源？尤其是一个重要的资源：CPU
   2. 标准的解决方法是通过抢占式多任务处理方式——OS运行一个进程一段时间，然后CPU挂起进程，运行另一个进程
      1. 必须保存并且恢复进程状态
      2. 关键问题：公平。必须保证每个进程对于CPU资源的获取和使用都是平等的
10. 进程抽象化的实现
    1. 操作系统是如何实现进程抽象化：
       1. 使用上下文切换从一个进程切换到另一个进程
11. 如何实现上下文切换（就是某些书本中的进程切换）
    1. 问题：计算机如何实现上下文切换
       1. 一个处理器只有有限的物理资源。比如，他只有一个寄存器组（Register set）。但计算机中的每个进程都有自己的寄存器组
    2. 解决：在上下文切换时保存并恢复硬件状态。把状态保存在进程控制块（PCB）中
12. 基于硬件的进程控制器都存储什么
    1. 寄存器状态信息（Registers）——几乎所有的计算机都把它存在PCB中
    2. 进程状态标识符（Processor Status Word）
    3. 至于内存：
       1. 大多数计算机允许多个进程共存在机器的物理内存中
       2. 一些计算机需要通过内存管理单元（Memory Management Unit，MMU）的变动来实现上下文切换
       3. 不过一些早期的个人电脑直接在磁盘上（disk）切换所有进程的内存。
13. 操作系统是事件驱动的

操作系统从本质上是事件驱动的——他们等待一个事件的发生，作出适当的响应，然后继续等待下一个事件的发生

* + 1. 用户敲击一个按键。那个按键就会被重复在屏幕上
    2. 一个用户程序发出一个系统调用去读取一个文件。操作系统会计算出哪些磁盘块会被引进，然后发送一个请求给磁盘控制器读取磁盘块到内存中
    3. 磁盘控制器完成从磁盘块的读操作，然后产生中断。操作系统将读取的数据移动到用户程序并且重新启动用户程序

1. 事件驱动和线程

当构建一个具有不同序列活动的操作系统时，线程是一个重要机制。

1. 线程概念

一个线程是一个线程状态下的一个执行流

**线程和进程的区别：**

多线程可以共享他们的部分状态信息。并允许多个线程读写同一块内存（但是进程不可以直接访问其他进程的内存空间）。但是每个线程仍有它自己的寄存器和自己的栈；其他线程可以读写栈内内存。

每个用户的进程背后都有一个核心线程。

1. 线程控制块
   1. 每一个线程都有一个线程控制块（Thread control block，TCB）
   2. 线程控制块中都有什么？

只有寄存器。当线程切换的时候不需要向内存管理单元（MMU）做任何事情，因为所有线程都可以访问同一个内存。

1. 每一个用户进程都是由一个核心线程支持的
   1. 一个操作系统的每一个独立的活动区都会有一个单独的线程。操作系统会将独立的线程分给他的每一个进程，并且线程会代表进程来执行操作系统的活动。在这个场景下，我们说每个用户进程都是由核心线程来提供支持的
   2. 核心线程支持进程的例子：
      1. 当进程发出一个系统调用去读一个文件，进程的线程会接收到调用并计算出那些磁盘发生了访问，并且在需要的时候发出一个低等级的指令来转让磁盘空间。这个线程会等到磁盘完成读取文件信息之后挂起。
      2. 当进程开始一个远程TCO通信时，它的线程控制着发送网络包的低级细节。
2. 每一个活动区都有一个独立的线程的优点
   1. 每一个活动区都有一个线程允许程序员设计和那个活动相关的动作，而动作仅作为单个连续的动作和事件流
   2. 程序员不需要必须处理相同的线程之中的错综复杂的多个活动。
3. 为什么允许多个线程访问同一片内存
   1. 因为在操作系统内部，线程必须非常紧密的协调它们的活动
      1. 假如两个进程发出读取文件的系统调用在几乎相同的时间，必须确保操作系统适宜的连续处理磁盘请求
      2. 当一个进程分配一个内存，它的线程必须找到一些空闲的内存并且把它们发送给进程。必须保证多线程分配不相交的内存碎片
   2. 让多个线程共享相同的地址空间会使得更容易的去协调它们的活动——可以建立共同的数据结构来表述系统状态并且让那些线程利用（读或者写）那些数据结构来计算出当他们处理一个请求时应该做什么。
4. 线程处理异步
   1. 一个棘手的事情是线程必须处理异步事件。线程正在执行时异步事件任意的发生，有可能干扰到线程的活动，除非程序员做一些限制异步事件的工作、
   2. 例如：
      1. 一个中断发生，从一个线程传输控制信号给一个中断处理器
      2. 一个事件切片开关发生，从一个线程传输控制信号给另一个线程
      3. 两个在不同的进程上运行的线程操作同一片内存时
5. 异步的潜在问题

**异步事件如果不能被正确的控制，会导致错误的行为，**例如：

两个线程都要发出磁盘请求。第一个线程开始通过程序访问磁盘控制器（假定它是内存映射的，并且需要发送多个写的操作指定一个磁盘）。同时第二个线程在一个不同的进上运行并且也发出内存映射的

1. 同步线程
   1. 程序员需要协调多线程的活动以至于不让坏事情发生
   2. 关键机制：同步操作。相对于其他线程事件来说，这些操作允许线程控制他们的事件时间。

# 三．Synchronization

* + - 1. 概述
         1. 线程的创建和操作
         2. 竞争条件和临界断面
         3. 原子操作的概念
         4. 同步的抽象概念

信号

锁和条件变量

* + - 1. 一个线程接口
         1. class Thread {

public:

Thread(char\* debugName);

~Thread();

void Fork(void (\* func)(int), int arg);

void Yield();

void Finish();

}

* + - 1. 线程方法
         1. Thread（）这个构造方法创建一个新线程。它分配一个空间数据结构给TCB。
         2. Yield（）方法放弃CPU占用来让其他线程使用
         3. Finish（）方法停止调用线程
      2. Fork（）方法
         1. 真正让线程开始运行，必须告诉这个线程什么方法要开始运行和什么时候运行。Fork（）方法给线程那个函数，并且给那个函数一个参数
         2. Fork（）方法首先分配一个堆栈给线程。然后当线程开始运行时Fork（）创建TCB，它会调用方法并向这个方法传一个正确的参数。它会把线程放到运行队列的某处。然后Fork（）返回，线程继续调用它。
      3. TCB在运行函数时的设置
         1. 首先操作系统把TCB内的堆栈指针指向堆栈
         2. 然后操作系统设置TCB中的程序计数器（Program Counter，PC）为所要执行函数的第一个指令集的地址
         3. 操作系统继续设置TCB里保存第一个参数的寄存器成参数
         4. 当线程系统从TCB恢复状态（执行完TCB里的内容），函数会神奇的运行起来
      4. Runnable

系统维持一个可运行的线程的队列。无论处理器是否空闲，线程调度器都会从那个运行的队列中抓取一个线程然后运行它。

* + - 1. 并发线程的执行
         1. 从概念上说，线程是同步执行的。这是一个理解线程行为的最好方式。但是事实上，系统仅有有限的处理器，并且他不能立即让那些可运行的线程都同时跑起来。因此，必须在有限的处理器上实现多个通道来运行线程。
         2. 一个线程的例子：

int a = 0;

void sum(int p){

a++;

printf(“%d:a = %d\n”, p, a);

}

Void main (){

Thread \* t = new Thread(“child”);

t->Fork(sum, 1);

sum(0);

}

* + 1. 可能的结果。两个线程同时调用sum（）。要完全的明白，我们必须最原始的打开sum去了解他的每一个部件。

Sum（）最开始把读入数值一个放入寄存器

然后增加这个寄存器内放的数值

然后把寄存器内的值取回给a

然后读取字符串、p还有a进寄存器，并通过寄存器把值传递给printf函数

最后调用打印数据的printf函数

* + 1. 可能的输出结果

0 ：a = 1

1 ：a = 2

1 ：a = 1

0 ：a = 2

顺序执行两个线程

1. 0 ：a = 2

1 ：a = 2

1. 1 ：a = 2

0 ：a = 2

Printf的两个调用是在两个a++调用之后运行

1. 0 ：a = 2

1 ：a = 1

1. 1 ：a = 2

0 ：a = 1

第一个printf被调用时延时到第二个printf调用后面发生了

1. 0 ：a = 1

1 ：a = 1

1. 1 ：a = 1

0 ：a = 1

在a还是0的时候，两个a++同时发生了

1. 不确定的结果
   1. 当并发发生时，它的结果依赖于内部指令的交错顺序
   2. 结果是不确定性的——在你跑了好几遍程序之后，你也许会得到不同的结果
   3. 因此，很难去得到同样的bug
   4. 不确定的运行结果是让编写并行程序比编写串行程序更难的原因之一
2. 错误的结果

有可能程序员不是想要上列的程序结果。只想要在两个线程都跑完后的结果是2。要实现这个，必须要做原子增量操作。即必须防止交错的指令顺序导致的不可预期的增加。

1. 竞争条件和临界区
   1. 竞争条件是一种在程序的结果依赖于程序交错指令并发执行结果的情况。
   2. 临界区是包含竞争条件的并发程序的一部分
2. 原子操作的概念
   1. 原子操作是一种执行时不受其他任何操作干扰的操作。换句话说，他就像一个单元在工作
   2. 典型的说，建立复杂的原子操作是避免无序的原子序列。在例子中，原子操作就是单一的机器语言
   3. 更正式的说，假如多个原子操作执行，最终结果将保证和顺序执行的一样。
3. Sum的例子和原子操作

在我们上述的例子中，建立一个不受movl和addl机器指令影响的增量操作。需要该增量操作是原子的

1. 同步性（Synchronization）

使用同步操作来使编码序列原子化

1. 信号量
   1. 信号量是我们的第一个同步抽象例子，概念的说，它是一个支持两个原子操作P和V的计数器
   2. P将自动的等到计数器大于0，然后对计数器减1并返回它
   3. V会将计数器加1
2. 信号量接口

Class Semaphore {

Public:

Semaphore( char \* name, int value);

~Semaphore();

Void P();

Void V();

}

1. 使用信号量的Sum例子

Int a = 0;

Semaphore \* s;

Void sum(int p){

Int t；

S->P();

a++;

t = a;

s->V();

printf(("%d : a=%d\n", p, t);

}

Void main(){

Thread \* t = new Thread(“”);

s = new Semaphore(“s”, 1);

t->Fork(sum, 1);

sum(0);

}

1. 互斥（mutual exclusion）
   1. 我们在这里使用信号量实现互斥的机理。在互斥背后的想法是如果只有一个线程，那么它将被允许做什么
   2. 在这个场景下，只有一个线程接受变量a
   3. 用互斥原理来实现原子运算
   4. 执行原子运算的代码被叫做临界区
2. 生产者消费者问题
   1. 概念是生产者产生数据而消费者消费数据
   2. Unix管道拥有一个生产者和一个消费者。你也可以认为一个人在键盘上打字就类似于生产者，shell程序读取那些字符就像消费者
   3. 这里就有一个同步的问题：确保消费者不会走在生产者前面。但是我们希望生产者可以一直生产即使没有消费者消费（无限的缓冲区，Unbounded-Buffer）。
3. 无限缓冲区下，使用信号量的生产者消费者模型

Semaphore \* l;

Semaphore \* s;

Void consumer( int d ) {

While( 1 ) {

s->P();

l->P();

consume the next unit of data

l->V();

}

}

Void producer( int d ) {

While( 1 ) {

l->P();

produce the next unit of data

l->V();

s->V();

}

}

Void main() {

l = new Semaphore(“l”, 1);

s = new Semaphore(“s”, 0);

Thread \* t = new Thread(“c”);

t->Fork(consumer, 1);

t = new Thread(“p”);

t->Fork(producer, 1);

}

1. 有限缓冲区的生产者消费者模型
   1. 在真实的环境中，一个新的约束。如果我们让生产者不停的生产，消费者一点也不消费，那么我们必须把生产出来的数据放在什么地方。但是，没有机器有那么大的地方来放这么多的产品。因此如果可以的话我们想让生产者领先于消费者，哪怕仅仅一点点。我们需要实现一个仅能存住N条记录的有限缓冲区。如果缓冲区满了，生产者必须在他能够再往里面放入更多的东西之前等待。
2. 使用信号量实现有限缓冲区的生产者消费者模型

Semaphore \* l;

Semaphore \* full;

Semaphore \* empty;

Void consumer( int dummy ) {

While( 1 ){

full->P();

l->P();

consume the next unit of data

l->V();

empty->V();

}

}

Void producer ( int dummy ) {

While( 1 ){

empty->P();

l->P();

produce the next unit of data

l->V();

full->V();

}

}

Void main() {

l = new Semaphore(“l”, 1);

full = new Semaphore(“f”, 0);

empty = new Semaphore(“e”, N);

Thread \* t = new Thread(“c”);

t->Fork(consumer, 1);

t = new Thread(“p”);

t->Fork(producer, 1);

}

1. 操作系统中一个有限的缓冲区例子

一个你可能在操作系统中用到的消费者和生产者的例子，就是控制台。你或许会用信号量来确保你不会有在一个字符没有被敲入之前读取它的可能。

1. 锁和条件变量
   1. 信号变量是一个同步抽象概念
   2. 这里还有另一个同步的抽象概念叫做锁，一个仅仅关于互斥的特别的抽象概念，还有条件变量也是同步的抽象概念
2. 锁的接口

Class Lock() {

Public:

Lock(char \* name);

~Lock();

Void Acquire();

Void Release();

}

* 1. 锁的运算解释：
     1. 一个锁有两个状态：锁上和开锁
     2. Lock( name )：创建一个开始状态未开锁的锁
     3. Acquire()：利用原子操作等待锁的状态为开锁时，将状态设置成锁上状态
     4. Release()：利用原子操作把锁的状态从开锁状态到锁上状态

1. 上锁实现的需求
   1. 在同一时间，只有一个线程能够获得锁。（安全性，safety）
   2. 当多个线程尝试获取一个开锁状态的锁时，他们之中只有一个线程可以获得锁。（活跃性，liveness）
   3. 所有开锁过程在有限时间内完成。（活跃性，liveness）
2. 锁的实现具有令人满意的性能
   1. 效率：只占用一点点资源
   2. 公平：线程只有当他们需要锁时才会获取锁。
   3. 使用简单
3. 锁的使用
   1. 当使用锁的时候，通常会有多个线程访问锁的数据片段
   2. 当一个线程想要访问一片数据，它首先要获得锁。然后再执行访问操作，在这之后打开锁
   3. 因此，锁允许线程在每一片数据上执行复杂的原子操作
4. 用锁和信号量解决有限缓冲区生产者消费者模型

Lock \* l;

Semaphore \* full;

Semaphore \* empty;

Void consumer( int dummy ){

While( 1 ){

full->P();

l->Acquire();

consume the next unit of data

l->Release();

empty->V();

}

}

Void producer( int dummy ){

While( 1 ){

empty->P();

l->Acquire();

producer the next unit of data

l->Release();

full->V();

}

}

Void main (){

l = new Lock(“l”);

full = new Semaphore(“f”, 0);

empty = new Semaphore(“e”, N);

Thread \* t = new Thread(“c”);

t->Fork(consumer, 1);

t = new Thread(“p”, 1);

t->Fork(producer, 1);

}

1. 能够仅依靠锁就实现无限制缓冲区
   1. 一个问题：如果消费者想要消费一片数据在生产者生产数据之前，它就必须等待。但是锁不会允许消费者一直等到生产者生产数据。因此，消费者必须循环直到数据准备好了。这是糟糕的因为它会浪费CPU资源。
   2. 这里还有一个抽象的同步概念叫做条件变量，专门用来解决这种情景
2. 条件变量接口

Class Condition{

Public:

Condition( cha \* debugName );

~Condition();

Void Wait( Lock \* condiitionLock );

Void Signal( Lock \* conditionLock );

Void Broadcast( Lock \* conditionLock );

}

* 1. Condition(name)：创建一个条件变量
  2. Wait(Lock \* l)：原子的释放锁并等待。当Wait()返回锁时，锁可以被再获取
  3. Signal(Lock \* l)：原子的启用一个正在等待的线程，让他运行。当Signal()返回锁的时候，锁是一直被获取的
  4. Broadcast(Lock \* l)：（传说中的广播机制）原子的使所有等待状态中的线程运行。当Broadcast()返回锁的时候，锁是一直被获取的

1. 锁和条件变量的使用
   1. 具有代表性的，你用一个数据结构将锁和环境条件变量
   2. 在程序在数据结构上执行一个操作之前，它会获取锁
   3. 如果在它能够执行操作之前一直在等待，它使用环境条件变量去等待可以将数据机构变成它可以能够执行操作状态的另一个操作
   4. 在某些情况下，你需要不止一个条件变量
2. 使用锁和条件变量解决有限的缓冲区

Lock \* l;

Condition \* c;

Int avail = 0;

Void consumer( int dummy ){

While( 1 ){

l->Acquire();

while ( avail == 0 ){

c->Wait(l);

}

Consume the next unit of data

avail--;

l->Release();

}

}

Void producer( int dummy ){

While( 1 ){

l->Acquire();

produce the next unit of data

avail++;

c->Signal(l);

l->Release();

}

}

void main() {

l = new Lock("l");

c = new Condition("c");

Thread \*t = new Thread("c");

t->Fork(consumer, 1);

t = new Thread("c");

t->Fork(consumer, 2);

t = new Thread("p");

t->Fork(producer, 1);

}

# 死锁

1. 当获取超过一个锁时，死锁的可能

你可能需要写代码来获取超过一个锁。这便打开了死锁的可能

Lock \*l1, \*l2;  
void p() {  
 l1->Acquire(); l2->Acquire();  
 code that manipulates data that l1 and l2 protect  
 l2->Release(); l1->Release();  
}  
void q() {  
 l2->Acquire(); l1->Acquire();  
 code that manipulates data that l1 and l2 protect  
 l1->Release(); l2->Release();  
}

1. 死锁触发的必备条件
   1. 互斥：只有一个线程能够获得锁
   2. 占有并等待：至少有一个拥有锁的线程在等待另一个进程释放锁
   3. 不可被抢占：拥有锁的进程才能释放这个锁
   4. 循环等待：1等2,2等3……n等1
2. 避免死锁
   1. 锁排序，一直按照排序来获得锁。
   2. 消除循环等待条件。
3. 如果锁在不同的订单中需要被获取，应该做什么
   1. 大多数锁定抽象提供了一个操作,试图获取锁,但如果它不能返回。我们将调用TryAcquire操作。使用这个操作来试图获取你需要打乱顺序获取的锁。
   2. 如果操作成功，很好。一旦你得到了锁，就没有问题了
   3. 如果操作失败，你的代码将需要释放所有的锁。
4. 简单的算法实现死锁避免（资源角度）
   1. 每一条进程都告诉它将需要的最大资源数
   2. 正如进程运行，他请求资源但是从来不超过那个最大资源需求数
   3. 系统调度进程和分配资源在某种程度上会确保没有死锁的结果

# 实现同步操作

* + - 1. 测试和设定

测试和设定指令原子的检测内存地址是否为0，如果是，将内存地址设为1，如果内存地址为1，则不进行操作。他返回内存地址原先的值。你可以利用Test and Set 来实现锁：

① 锁的状态由内存地址实现。锁unlocked时地址为0，锁locked时地址为1 。

② 锁操作的实现：while(test-and-set(l)==1);

③ 开锁操作的实现：\*l = 0;

# CPU调度

1. 什么是CPU调度

在多线程环境下决定那一个线程运行

1. 为什么CPU调度很重要

因为CPU调度对系统的资源利用和系统全局表现有很大的影响

1. 为什么这是真的
   1. 顺便提一下，世界经历了一个曾由没有复杂调度系统的最受欢迎的系统（DOS, Mac）主导很长时间（80’s, 90’s）
   2. 他们都是单进程的，并且一个时间内运行一个进程直到用户指导它们运行另一个进程
   3. 更多较新的系统（Windows NT, Linux）回归到复杂的CPU调度算法
   4. 是什么导致这种现象
2. 大多数调度算法背后的基本假设
   1. 有一个进程池争夺CPU资源
   2. 进程是各自独立的抢夺资源
   3. 调度的工作就是把稀缺的CPU资源公平的分配给不同的进程，在某种程度上优化性能标准
3. 进程是如何运转的
   1. 首先是CPU/IO的脉冲循环。一个进程将运行一个周期（CPU脉冲），执行一些IO（IO脉冲）操作，然后运行另一个周期（下一个CPU脉冲）。
   2. IO操作持续多长时间？这个取决于进程
4. IO密集型的进程
   1. 执行大量的IO操作的进程
   2. 每一次IO操作都伴随着一个处理IO的简短CPU脉冲，然后更多的IO事件发生
5. CPU密集型进程
   1. 进程执行大量计算并且很少执行IO操作
   2. 会导致较长的CPU占用时间
6. 调度和CPU/IO脉冲（脉冲，burst，指的是对CPU的一次性占用时间）
   1. 调度要做的有代表性的事情就是转换CPU到两一个进程，当一个进程执行IO时
   2. 为什么？IO会花费很长时间，并且不想在等待IO完成时令CPU空闲
7. CPU脉冲时间分布
8. 进程运行状态
   1. Running（运行中）：进程在CPU上运行
   2. Ready（准备）：准备运行，但是并没有真正在CPU上运行
   3. Waiting（等待）：等待一些事件发生，像IO。
9. 什么时候调度决定替换
   1. 当进程从running状态切换到waiting状态时。可能是由于IO请求，还可能是等待子进程终止，或者是等待同步操作（如获得锁）完成。
   2. 当进程从running状态切换ready状态。比如，中断处理程序的完成。中断处理程序的一般例子：交互系统中的时钟中断。如果调度把进程切换到这个状态，新的进程会取代正在运行的进程继续运行。另一个一般的例子就是IO。
   3. 当进程从waiting切换到ready时。比如，IO完成或者锁的获取。
   4. 当一个进程终止
10. 如何评估一个调度算法
    1. CPU利用率
    2. 生产效率（吞吐量throughout）：单位时间完成的进程数
    3. 周转时间（Turnaround Time）：进程提交到完成总时间
    4. 等待时间（Waiting Time）：从ready到running之间的时间
    5. 响应时间（Response Time）：提交请求到第一个响应的时间
    6. 调度效率（Schedule Efficiency）
11. 批处理系统和交互式系统的巨大差别
    1. 批处理系统，典型地需要良好的生产效率或者提交时间
    2. 在交互式系统中，那些因素的通常都是重要的，不过响应时间通常是主要考虑的
    3. 一些其他的系统，生产效率或者提交时间都不是真正有作用的：一些进程理论上会永远的跑下去
12. 长程调度和短程调度的区别
    1. 长程调度是决定那一个程序可以进入到系统中处理
    2. 他们一旦运行起来，，他们可能会因为IO或者是优先权挂起
    3. 短程调度决定哪一个可运行的进程将被处理器执行
13. 先来先服务（先进先出算法，FCFS）调度算法
    1. 一个准备好的队列，操作系统运行队列最前面的，新的进程进入到队列的尾部（堆栈）
    2. 一个进程不会放弃CPU知道它被终止或者是执行IO操作
14. FCFS算法的性能
    1. 考虑FCFS算法的效率用三个计算密集型的进程。我们有三个进程：P1（用时24秒）， P2（用时3秒），P3（用时3秒）。
    2. 如果接受顺序是P1，P2，P3，那么
       1. 等待时间：（24+27）/3 = 17
       2. 周转时间：（24+27+30）/3 = 27
       3. 生产效率：（24+3+3）/3 = 10
    3. 如果进程进入顺序是P2，P3，P1的话
       1. 等待时间：（3+6）/3 = 3
       2. 周转时间：（3+6+30）/3 = 13
       3. 生产效率：（24+3+3）/3 = 10
15. 最短进程优先算法（SJF）调度算法
    1. SJF算法可以减少一些等待时间和周转时间的变异
    2. 巨大问题：调度程序如何指出一个进程将会运行多长时间
16. 长期调度
    1. 长期调度在批处理系统下，用户会给一个评估
    2. 如果它会太短，系统将会在当前进程完成之前删去工作。如果太长会拖延进程运行
17. 短期调度
    1. 必须有过去来预测未来
    2. 标准方法：用以前CPU区间每个进程时间延迟的指数平均数
    3. **sn+1 = w Tn + (1 - w)sn：Tn是CPU实际区间长度，sn是预测下一长度，W是计量因素（0<=W<=1），S0是默认值或系统平均值。**
18. 权重因素。

W告诉我们由过去到将来有多重

如果选择w =0.5，上一次观测与以前所有记录有相同价值

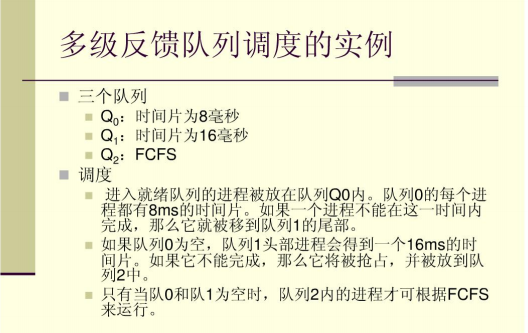
如果w = 1，只有最近的检测有价值

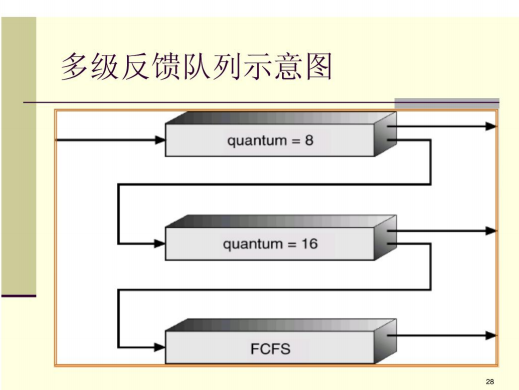
1. 抢占式的SJF vs 非抢占式的SJF
   1. 抢占式的SJF调度在进程准备好的时候会重新启动调度决策。如果一个新进程的优先权大于正在运行的进程，CPU会替换掉正在运行的进程并且执行那个新进程。
   2. 非抢占式的SJF调度只在运行的进程主动放弃CPU后执行调度决策。事实上，它允许每一条进程完成它的CPU周期
2. 一个例子
   1. 考虑4条进程P1（周期时间8），P2（周期时间4），P3（周期时间9），P4（周期时间5）在同一时间到达的顺序是P1，P2，P3，P4
   2. 假设在周期发生后，进程不能够再允许长时间执行（比如说，至少100时间）
   3. 一个抢占式的SJF调度会怎么做
   4. 一个非抢占式的SJF调度会怎么做
3. 优先调度
   1. 每一条进程都被给与一个优先权，然后CPU执行优先权最高的进程
   2. 如果多条进程都是一个优先级别，用一些其他的标准-------例如先来后到调度（FCFS）
   3. SJF是基于优先级的调度算法的一个例子。作为指数衰减的算法，一个给定的进程的优先顺序是随时间改变的
4. 一个例子
   1. 假设我们有5条进程P1（周期时间10），P2（周期时间1，优先级1），P3（周期时间2，优先级3），P4（周期时间1，优先级4），P5（周期时间5，优先级2）。越低的数字代表越高的优先级。
   2. 一个标准的优先调度会怎么做
5. 饥饿
   1. 优先调度的一个大问题：低优先级进程的饥饿或者阻塞
   2. 可以使用aging来避免--------提高长期处于低优先级缺还一直没有运行的进程的优先级
6. 交互式系统的调度
   1. 不能够让任何一个进程一直在CPU上运行直到它主动放弃---------必须在合理的时间内给用户一个回复
   2. 因此，使用一个叫做循环调度
      1. 有点类似于FCFS但是有抢占机制
      2. 时间段或者说是时间片
      3. 让队列中的第一个进程运行直到到达有效地时间段，然后运行队列中的下一个进程
7. 实现循环调度的时钟中断
   1. 当调度一个进程时，在时间段到期后设置一个离开定时器
   2. 如果进程在时间结束之前做IO操作的话，没问题——继续运行下一个进程
   3. 但是如果进程的量子时间到期，需要做一个上下文切换。保存正在运行的进程的状态并且运行下一个进程
8. RR（round-robin）调度如何良好的工作
   1. 当然，要先给一个合适的响应时间，当然也可以给一个糟糕的等待时间
   2. 考虑等待时间在循环调度下对于三个进程P1（周期24），P2（周期3），P3（周期4）循环周期为4
   3. 发生了什么？平均等待时间是多少？什么给了最佳等待时间？
9. 如果是一个很短的时间周期RR调度会发生什么
   1. 时间片很小时，可以看作n个进程各自拥有1/n个CPU，n为进程数
   2. 小量子时间片上下文切换的开销问题
10. 拥有一个小量子支持的硬件会怎么样

你有一些东西可以调用多线程或者超大线程

* + 1. 给CPU一群寄存器并且严格的管道执行
    2. 使进程一个接一个的流入管道中
    3. 像IO一样对待内存访问——挂起线程直到数据从内存中恢复
    4. 在此期间，执行其他线程。用估算隐藏潜在内存访问

1. 一个真正的大的量子会怎么样
   1. 相当于FCFS
   2. 经验法则——让80%的CPU中断比量子时间更短
2. 多队列调度
   1. 多队列调度就像RR调度算法，不过是拥有多个队列而已
   2. 典型地，把进程分给不同类别的队列，并且给每个队列一个类别
   3. 因此，只有系统、交互式的和分批的进程在这个规则下会有优先级
   4. 当然也可以把CPU的一个百分比分给每个队列
3. 多级反馈队列调度算法
   1. 多级反馈队列调度算法迹象多级调度算法，只不过进程能够在不同优先级的队列之间进行转移
   2. 能够习惯于给予IO限制和超越CPU约束进程的交互式进程CPU优先权
   3. 也能通过增加进程优先权来让进城有更长的时间片来防止饥饿





1. 传统Unix调度
   1. 另一个多队列反馈调度算法的例子就是Unix调度程序。我们将重温一个没有包含kernel优先级的简化版本
   2. 这个算法的特点是公平的分配CPU在进程之间，对于没有立刻使用大量CPU资源的进程就交出它的权限
   3. 进程会被给与一个基本的优先权，60。越低的数字代表着更高的优先权
   4. 系统时钟一秒钟生成50到100之间次数的中断，因此我们将假定一个数值60作为时钟每秒的中断
   5. 时钟中断处理程序会增加一个CPU使用字段在每次PCB中断进程时
   6. 系统总是运行更高优先级的进程。如果是两个优先级一样的进程，它将会运行那个准备时间更长的那个进程
   7. 每秒，它都会重新计算优先权和CPU字段给每一个进程，根据下面那个公式。
      1. CPU usage = CPU usage/2
      2. Priority = CPU usage/2 + base priority
   8. 因此，当一个进程没有立刻使用较多的CPU，它的优先级上升
   9. IO限制进程和交互式进程的优先权会更高，而CPU限制进程的优先权则会更低
   10. Unix也允许用户提供一个恰当的值给每个进程。恰当的值修改优先权的计算如下：
       1. Priority = CPU usage /2 + base priority + nice value
   11. 因此，你可以恰当的减小你的进程的优先级到其他进程
2. 多队列反馈调度算法是复杂的

# OS大杂烩

* + - 1. 什么时候一个进程需要调用OS功能函数

这里有几个例子：

读文件时。OS必须执行文件系统操作来要求读取相应数据从磁盘中

建立子进程时。OS必须让子进程执行相关的东西

发送包到互联网时。OS典型的做法是控制网络接口

* + - 1. 为什么进程能够直接的做这些工作
         1. 方便性：

在OS中实现一次并将其封装在接口中，以便每个人都能使用

* + - * 1. 可移植性：

OS输出一个公共的接口以便在不同硬件平台使用。

* + - * 1. 保护性：

如果给应用程序完全的调用硬盘或者网络或者其他一些的权力，它们会腐化其他应用中的数据，出于恶意的或者是bugs。必须让OS消除应用之间的安全问题。当然，应用程序一直要信任OS

* + - 1. 进程是如何调用OS函数的
         1. 通过生成一个系统调用
         2. 概念的，进程调用一个与该进程分离的子程序去执行必须的函数
         3. 但是OS必须在一个不同的保护域执行应用。代表性的，OS在一个只允许运行正确的操作的监督模式下执行。
      2. 一个系统调用指令
         1. 从用户模式下切换到监督模式，大多数机器提供一个系统调用指令
         2. 这个指令有一个例外的结果。硬件从用户模式（User mode）转换到核模式（Supervisor mode）会调用在操作系统中异常处理程序。
         3. 一个约定俗成的例子，进程用来与OS交互操作
      3. 一个例子——开放系统调用
         1. 系统调用通常以一个子程序调用开始。在这种情况下，当进程想要打开一个文件时，它仅仅在系统程序库的某处调用开型程序（open routine）。
      4. 程序库内部

在程序库内部，开放式子程序（open subroutine）执行一个会产生系统调用异常的系统调用指令。

* + - 1. 系统调用中的参数传递
         1. 开放式系统调用也需要获取参数——字符串的地址会告诉要打开的文件的名字
         2. 按照惯例，编译程序会将参数放入寄存器4在它生成代码去调用库中的开放式程序。因此，OS会检查出那个寄存器中的关于要打开的文件的名字的地址
         3. 更多例子：成功的参数会放入寄存器5、寄存器6等等中。任何来自系统调用的返回值都会放入寄存器2中
      2. 异常处理程序内部
         1. 在异常处理程序内部，OS指出会获取何种动作，执行何种动作，然后返回到用户程序中
      3. 另外几种异常
         1. 例如，如果程序企图使用一个NULL指针，硬件产生一个异常。OS将必须指出那种异常发生了并相应地处理它。另一种异常就是除以0
      4. 在中断中发生的类似的事
         1. 当中断发生，硬件会让OS进入监督模式下并执行一个中断处理程序
         2. 中断与异常的不同之处在于中断是由外部事件触发的而异常是由正在运行的程序触发的
      5. 目标文件格式
         1. 运行一个进程，OS必须从硬盘中加载一个可执行文件到内存。这类文件都包含什么？
         2. 代码运行，任何初始化的数据，未初始化的数据应该占多大空间的一个规格。可能会帮助调试器运行
         3. 编译器，链接器，OS必须同意可执行文件的格式
      6. Nachos目标文件格式

#define NOFFMAGIC 0xbadfad /\* magic number \*/  
typedef struct segment {  
 int virtualAddr; /\* location in virtual address \*/  
 int inFileAddr; /\* location in this file \*/  
 int size; /\* size of segment \*/  
} Segment;  
typedef struct noffHeader {  
 int noffMagic; /\* should be NOFFMAGIC \*/  
 Segment code; /\* executable code segment \*/  
 Segment initData; /\* initialized data segment \*/  
 Segment uninitData; /\* uninitialized data segment \*/  
} NoffHeader;

* + - 1. 当OS载入一个可执行文件是回做些什么
         1. 读取头文件
         2. 检查幻数是否需要匹配
         3. 指出这个进程需要多大空间。这包括了栈的空间，代码，初始化数据和为初始化数据。
         4. 如果它需要在物理内存中保持全部的进程，那么它会先找到他所需要的能够维持进程物理进程
         5. 然后从文件中读取代码段到物理内存中
         6. 然后从文件中读取初始化数据片段到内存中
         7. 它将栈和未初始化的内存致零
      2. I/0在操作系统中的概述
         1. 有两种基本的方式来管理IO

IO内存映射：IO设备上的控制寄存器映射到处理器的内存空间上。处理器控制设备通过读写IO映入的地址

IO程控化：处理器有一个特别的IO指令像IN和OUT。这些控制IO设备正常工作

* + - 1. 异步I/O
         1. 典型地，对于处理器来说IO是异步的
         2. 因此，处理器将开始一个IO运算（如写磁盘），然后离开，执行其他进程
         3. 当IO操作完成，它中断处理器。处理器适量关闭中断处理程序，无论任何活动需要它都会去处理

# 内存管理

内存管理算法的要点

支持主内存的共享

我们将集中于多进程共享同一块物理内存

内存管理的关键

保护。必须允许一个进程保护它的内存在另一个进程访问它的内存时。

命名。 进程是如何识别内存中共享的那一片的

透明度。共享是如何透明的。

效率。任何内存管理策略不应该带来太大的负载

为什么要在进程之间共享内存？

因为我们想要处理器实现多进程

分时使用系统，重叠计算和IO。因此必须实现多进程同时常驻在物理内存中。进程必须共享物理内存

内存检测在哪里完成

在硬件中完成，为了速率

当OS运行进程时，加载基本的界限寄存器

注意：有一个翻译过程。程序产生虚拟地址通过翻译变成物理地址。不过，再不会有一个保护问题：一个进程不能访问另一片内存，以为内它是外部地址空间。如果它尝试访问它，硬件将产生异常

内存分配

以一种动态将物理内存分配给进程的模型结束。

几种分配策略：最佳适配，首次适配，下次适配等

都由外部碎片提供

防止外部碎片的方法：分页

1. 用页帧的方式分配物理内存
2. 将应用程序的地址空间用页进行分割。页和帧同样大小，页存放在帧里
3. 当进程运行一个地址，动态的将保存数据的页翻译到物理页帧上。

分页中的虚拟地址：

1. 虚拟地址由两部分组成：页码和页偏移量
2. 页大小一般为2的指数倍

# 分页简介

分页的基本思想

1. 将物理内存分为固定大小的块叫做帧；将逻辑内存分为固定大小的块叫做页
2. 将应用程序的地址空间用页进行分割。页和帧同样大小，页存放在帧里
3. 当进程运行一个地址，动态的将保存数据的页翻译到物理页帧上。

动态加载：一个程序只有在调用时才被加载，不用的子程序不会被装入内存。

分页中的虚拟地址

1. 虚拟地址由两部分组成：页码和页偏移量
2. 页大小一般为2的指数倍
3. 为了访问给定地址的数据，系统会自动这样运行：

Extracts page number.   
Extracts offset.   
Translate page number to physical page frame id.   
Accesses data at offset in physical page frame.

分页中的地址翻译

简单的方法：利用页表：

Extract page number.   
Extract offset.   
Check that page number is within address space of process.   
Look up page number in page table.   
Add offset to resulting physical page number   
Access memory location.

地址翻译的问题

页表存在内存中，每次数据/指令的访问需要访问两次内存。

TLB加速地址翻译

1. 通过一个高速缓存（cache）来加速查找
2. 将最常用的表放在TLB中
3. TLB设计选择：全关联的、直接映射的、组关联的等
4. 可以给一个循环空间更大的直接映射

TLB查询如何工作

Extract page number.

Extract offset.

Look up page number in TLB.

If there, add offset to physical page number and access memory location.

Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

分页的内存分配

1. 在页帧中分配固定大小的物理内存简化了分配算法
2. OS能够跟踪空闲或者使用过的页，并在进程需要内存时给他分配页
3. 如果物理内存足够小，将不会有碎片产生

进程如何共享内存

1. OS让页表指向同一物理页帧
2. 用于快速内部处理交流机制
3. 允许快速透明共享

保护措施

1. 有很多种保护方式
2. 防止一个进程读写另一个进程的内存
3. 防止一个进程读另一个进程的内存
4. 防止一个进程读写自己的内存
5. 防止一个进程读自己的内存
6. How is this protection integrated into the above scheme?

答：通过在TLB中增加保护位来进行保护

虚拟内存简介

磁盘作为主存的拓展

虚拟内存的基本思想

当物理内存不够时，将页从物理内存中存到磁盘上，给新页腾出空闲帧

一些实际的考虑

1. 保留一定量的空闲帧。当空闲页帧数小于临界点时，选择一页将他存出
2. 页帧大小等于磁盘块大小

访问内存外的页

进程无法访问磁盘。因此，当进程尝试访问内存外的页时，OS到磁盘中将所需要的页读入新找到的空闲帧中，并重启进程。

虚拟内存的优点

1. 可以运行虚拟地址空间比实际物理内存大的程序
2. 程序内部共享内存
3. 可以灵活的在多个进程间共享

分页/虚拟内存的缺点

1. 额外的资源消耗
2. 储存页表的内存开销

解决办法：将页表再分页或者用更复杂的数据结构来存储虚拟到物理的翻译

1. 翻译开销

# 分页与虚拟内存

* + - 1. 页表结构

页表储存在实际机器的物理内存中

* + - 1. 几种页表

1. 线性页表
2. 二级页表
3. 三级页表
   * + 1. 页表的基本任务
4. 实现TLB重载
5. 维护常驻物理内存中的页的映射
   * + 1. 硬件与TLB关系

硬件只负责使用TLB，而软件处理TLB

* + - 1. 共享代码和数据

1. 不同进程指向同一物理页表，则共享内存
2. 一个进程写数据，另一个进程可以看见这些改变
3. 高效交流
4. 一个被编译的code保存在内存中，其他编译工具均可执行
5. 帮助内存利用
6. 共享代码必须出现在所有进程逻辑地址的同一位置
   * + 1. 重入代码的概念
7. 重入代码不可以修改自身，所以需要给每个进程的全局变量一个单独的copy
8. 所有Nachos内核代码需要重入，单独copy
9. 私有代码和数据页可以出现在逻辑地址空间的任何地方
   * + 1. 虚拟内存的主要思想
10. 主存作为后备存储器的高速缓存
11. 常用解决方案：页面请求
12. 一页可以常驻在磁盘或者主存中
    * + 1. 页面请求的第一个扩展
13. 第一个扩展是有效位
14. 每一个页表或者TLB入口都有一个有效位。如果设置了有效位，页存储在主存中；如果是无效位则存储在磁盘中。
    * + 1. 管理有效位和页表
15. OS从磁盘中转移页
16. 设置有效位和页表
    * + 1. 发生页错后OS将如何处理？
17. Trap to OS.
18. Save user registers and process state.
19. Determine that exception was page fault.
20. Check that reference was legal and find page on disk.
21. Find a free page frame.
22. Issue read from disk to free page frame.
23. Queue up for disk.
24. Program disk controller to read page.
25. Wait for seek and latency.
26. Transfer page into memory.
27. As soon as the controller is programmed, allocate CPU to another process. Must schedule the CPU, restore process state.
28. Take disk transfer completed interrupt.
29. Save user registers and process state.
30. Determine that interrupt was a disk interrupt.
31. Find process and update page tables.
32. Reschedule CPU.
33. Restore process state and resume execution.

给OS陷阱→保存用户寄存器和进程状态→确定页面错误→确定引用合法并在磁盘中找到页→找到空闲帧→声明从磁盘读入到帧→进入磁盘队列→磁盘控制器读取页→寻找和延迟等待→将页转移到内存中→当控制器编写完成，将CPU分配到另一个进程，并保存进程状态→获得磁盘转移完成中断→保存用户寄存器和进程状态→确定是磁盘中断→找到进程并更新页表→重新调度CPU→恢复进程状态并恢复运行

* + - 1. 有效访问时间（EAT）

EAT = (1-p)\*100 + p\*25\*10^6. P是页错率

* + - 1. 未来趋势

增加内存或者减小计算

* + - 1. 在哪里交换

1. 交换空间——为分页指派的一部分磁盘
2. 因为避免了正常文件系统的关联开销，交换空间操作比正常文件操作要快
   * + 1. 关于交换的更多声明
3. 不是为每一个进城数据提供后备存储器
4. 可运行的代码
5. 在未初始化的数据磁盘上的未引用的页
   * + 1. 核心映射
6. 要得到一个空闲帧，需要将一页写到磁盘中，并清除有效位
7. 核心映射记录每一个物理页帧，那个进程和虚拟页占用了页帧
   * + 1. 使TLB无效

当使一个页无效时，要同时清除TLB来防止过期的入口cache

* + - 1. 页置换算法

1. 长时间不用的页
2. 一个不需要写回后备存储器的干净的页
   * + 1. 使用位和修改位

使用修改位来降低页传输的开销——只有被修改过的页才写回磁盘。

* + - 1. TLB一致性

1. TLB入口与页表入口需要一致
2. 另一种方法综合这些位可以让TLB重载更快
   * + 1. 页置换算法
3. FIFO算法

缺点：可能弹出多次使用的页

1. Belady’s anomaly

更多地内存可能导致更多的页错

1. LRU算法（置换最长时间没有使用的页）
2. LRU近似算法
3. FIFO二次机会算法
   * + 1. 工作集的概念
4. 一个进程经常访问的页的集合
5. 工作集可能随时间改变
   * + 1. 系统颠簸
6. 如果物理内存太小以至于无法装下进程的工作集，就是系统颠簸
7. 颠簸的原因是进程一直忙于将页面换进换出
8. 颠簸是一个离散现象，通常有一个从不颠簸到颠簸的转换阶段
9. 当CPU利用率低时，OS会引入新的进程，最终工作集会变得比物理内存大，进程开始页操作，CPU利用率更低，引入更多进程，system开始颠簸
   * + 1. 页错率解决办法
10. 页错率高，分配更多的帧
11. 页错率低，丢弃一些帧给其他进程
12. 当所有进程页错率都高时，挂起其中一个进程
    * + 1. 页大小
13. 为什么不大些？
14. 传统，以及已经存在的代码
15. 增加的碎片
16. 为什么不小些？
17. 相同大小的工作集页小容易产生页错
18. 需要更多的TLB入口
19. 需要更多的页表

# 十二.文件系统简介

文件系统的重要性

最重要的信息都储存在文件系统中

什么是文件？

1. 可以随时访问，储存在稳定介质上的数据
2. 储存文件的位置：硬盘或者软盘、网络

目录和文件

为了组织文件以方便访问，OS将文件组织成树的结构，于是有了目录和文件

文件的意义

1. 文件的意义基于处理他的工具
2. OS处理可执行文件
3. 链接器处理目标文件

文件类型

一些系统支持很多不同类型的文件，并不同的处理不同类型的文件

文件属性

1. Name 文件名
2. Type 类型
3. Location 储存位置
4. Size 大小
5. Protection 保护
6. Time，data and user identification 时间，日期和用户标识

程序如何访问文件？

1. 顺序访问：打开文件，从头到尾进行读写
2. 直接访问：识别数据的起始位置
3. 间接访问：通过名字等标示符进行索引

文件被可以访问的方式不止一种

文件结构

对于某种访问方式可以适当组织文件结构以提高访问效率

1. 顺序排列，方便访问
2. 直接访问，用磁盘模块表
3. 间接访问，二级索引

多种文件格式的优缺点

优点：很容易找到可以运行的

缺点：导致OS变大，系统很难使用

目录结构

1. 为组织文件，系统提供继承文件系统管理。可以有文件或目录文件
2. 常见类型为树形结构

文件共享

1. 符号链接：指向存放位置
2. 硬连接：指向文件的物理数据

内存映射文件

数据储存在进程的地址空间中，一旦进程结束，数据也会消失。如果想保存数据，必须将他写入磁盘，然后需要时才从磁盘读取

为什么文件保护很必要

人们要共享文件，但并不是共享文件的所有方面

1. 只读不写
2. 不可读
3. 可以运行但不可修改

Unix安全模型

1. 拥有三种操作——读、写和执行
2. 每个文件都有所有者和组
3. 在“everybody”、“group”和“owner”层次上有不同的操作保护
4. 是简单公平的抢占式保护

磁盘对OS来说像什么？

对OS来说，磁盘就是一系列扇区。轨道的扇区在序列中，柱面的轨道在序列中，临近的柱面在序列中。OS在逻辑上把几个磁盘扇区连接起来以高效的增加磁盘块大小

# 十三.文件系统的实现

* + - 1. 连续分配

每个文件占用磁盘上一组连续的块。

* + - 1. 连续分配的优点

1. 简单快速的计算存有数据的磁盘——只需要记录文件的起始位置及长度
2. 访问文件很容易，几乎不需要需找时间
3. 直接访问也很快——只需要寻找然后读取
   * + 1. 连续分配的缺点
4. 文件长度很难增长
5. 需要整个的移动文件，即使文件很大
6. 外部碎片
7. 需要压缩，要很高的成本
   * + 1. 链接分配

所有文件存放在固定大小的块中。并像链表一样将临近的块链接起来

* + - 1. 链接分配的优点

1. 不需要更多的变量
2. 没有更多的外部碎片
3. 不需要压缩或重分配文件
   * + 1. 链接分配的缺点
4. 直接访问很麻烦——不得不一直读取下一个块的指针
5. 需要花费很长的寻找时间
6. 可靠性不强——一旦某个指针遗失了，问题就大了
   * + 1. FAT（File Allocation Table）文件分配表

每个分区的开始用于储存FAT，每个磁盘块在表中有一项，并可以通过块的号在表中索引。表中含有文件首块的块号，根据块号索引的FAT条目包含文件下一块的块号，这种链接会一直继续到最后一块，该块对应的FAT条目值为文件结束值。

* + - 1. FAT空闲块的分配

未使用的块用0值来表示，为文件分配一个新的块，只需要找到另一个值为0的FAT条目，用新块的地址替换文件结束值，用文件结束值替代0。

* + - 1. 索引分配

给所有的文件一个索引表。每个索引的入口指向包含真实数据的磁盘块。

* + - 1. 如何分配索引表

一个索引块通常为一个磁盘块，对于大文件可以将多个索引块连接起来。

* + - 1. 什么是索引节点？

关于文件的信息。每个索引节点都和一个文件相关。重要的域：

1. 模式（Mode）：包括保护信息和文件类型
2. 拥有者（owner）
3. 链接数量（Number of links）：指向索引节点的目录入口数量
4. 长度（Length）：文件有多少字节长
5. 块数量（Nblocks）：文件占用的磁盘块数量
6. 十个直接块指针组成的数组
7. 一个间接块指针
8. 一个二级间接块指针
9. 一个三级间接块指针

所以，一个文件由一个索引节点和它指向的磁盘块组成。

* + - 1. 超级块包含以下内容
         1. 文件系统的大小
         2. 文件系统Free block的数目
         3. 文件系统的可用Free block的列表
         4. 在free block列表中下一个free block的引索
         5. Inode列表的大小
         6. 文件系统中Free inode的数目
         7. Free inodes的缓存
         8. 在inode缓存中下一个free inode的引索
      2. 超级块和索引节点

1. 写文件时，可能需要分配更多的索引节点和磁盘块。超级块保持数据的轨道以帮助进程持久
2. 内核在内存中维护超级块，并定期将他写入磁盘。超级块也包含了重要的信息，所以将他复制到磁盘防止出错
   * + 1. inode分配
          1. 首先查看inode cache。Inode cache是一个free inode的stack，引索指向这个stack的最顶端
          2. 当OS分配一个inode，它仅仅渐减index。如果inode cache空了，它线性的搜索inode列表在磁盘中直到发现free inode
       2. 系统如何转变一个名字到inode
          1. 一个叫做namei的进程做这个事情