一．Overview and History

1. 什么是操作系统？

   不能精确地定义，操作系统是人们为了利用计算机解决问题才逐渐产生的

2. 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。

3. 初期电脑的使用

   a) 电脑很庞大，买、运行、维护的成本很高

   b) 单用户模式，交互模式（interactive mode）

   c) 交互方式低级

   d) 接口基本是硬件

4. 一个初期电脑使用的问题

   操作外部 I/O 设备的代码十分复杂

   解决方案 建一个 subroutine library 来进行 IO 设备的操作——OS 的雏形

5. 在电脑很昂贵的时候出现的问题：

   a) 当程序员完成设置时，计算机是空闲的，高投资的利用率（utilization）却很低。

      解决方案一 专人专用来设置

      解决方案二 建立 batch monitor

   b) 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

      解决方案 overlap 计算和 I/O。Buffering 和中断控制被加入 subroutine library。

   c) 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

      解决方案 multiprogramming、上下文切换、重要问题是保护：一个任务不能影响另一个任务的结果

6. 当电脑变得较便宜，新的问题：
   a) 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！
      i. 解决：交互式的分时处理 interactive timesharing
   b) 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
      i. 解决：抢占式调度
   c) 人们在使用计算机的时候需要将数据和程序保留下来
      i. 解决：加入文件系统使得能够快速地使用数据。
   d) 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
      i. 解决：优先处理巨大的程序
7. 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

   示例：一个处理器 多进程、一块物理内存多程序、一块磁盘多用户多文件、多程序分享带宽和存储空间

   Consumes Resources：

   操作系统消耗资源以提供上述功能。
8. 抽象方法效果好/坏

   示例 :timesharing 虚拟内存 hierarchical 和网络文件系统(networked file systems)

   有时工作得很好

   但是分时机制有时会使计算机负载过重

   虚拟内存会出现颠簸现象

9. What are the OS like?

   操作系统是复杂的软件

   ① 并发（concurrency）和异步（asynchrony）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。

   计算机科学面临的一个大难题就是如何构建一个可靠的大型的软件系统

   OS 是现存的为数不多的大型软件系统。我们可以从学习 OS 来增强如何构建大型软件系统。

二．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（execution stream）

   ① 执行流是一系列的指令

   ② 进程状态（process state）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

2. 进程状态通常包含（并不仅限于）下列内容：

   ① 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

   ② 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

3. 进程是一个重要的 OS 抽象

4. 单进程系统（uniprogramming）
    a) 一个时间段内只有一个进程。如 DOS。
    b) 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
    c) 单进程系统把东西放入诸如内存常驻程序（memory-resident program），以异步方式调用，但是始终有分离问题
    d) 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
5. 多进程系统
    a) 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
6. 多进程系统和资源共享
    a) 让哪个进程去使用机器物理资源？尤其是一个重要的资源：CPU
    b) 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
7. 进程抽象化的实现
    a) 操作系统是如何实现进程抽象化：
        i. 使用上下文切换（context switch）从一个 running 进程切换到另一个 running 进程
8. 如何实现上下文切换（就是某些书本中的进程切换）
    a) 问题：计算机如何实现上下文切换
        i. 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
    b) 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（Process Control Block）中
9. PCB 都存储什么
    a) 寄存器状态信息（Registers）——几乎所有的计算机都把它存在 PCB 中
    b) 进程状态标识符（Processor Status Word）
    c) 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。
10. 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

11. 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。

12. 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。

13. 线程控制块 TCB

a) 每一个线程都有一个线程控制块（Thread control block，TCB）

b) 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。

14. 每一个用户进程都是由一个核心线程支持的

a) 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它的每一个进程，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

b) 核心线程支持进程的例子：

i. 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。

ii. 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。

15. 每一个活动区都有一个独立的线程的优点

a) 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

b) 程序员不需要处理相同的线程之中的错综复杂的多个活动。

16. 为什么允许多个线程访问同一片内存

a) 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
   i. 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
   ii. 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
b) 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。

17. 线程处理异步
a) 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、
b) 例如：
   i. 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
   ii. 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
   iii. 两个在不同的进程上运行的线程操作同一片内存时

18. 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处理器上运行并且也发出内存映射的写操作给磁盘控制器。磁盘控制器会发生混乱。

19. 同步线程
a) 程序员需要协调多线程的活动以至于不让坏事情发生
b) 关键机制:同步操作。These operations allow threads to control the timing of their events relative to events in other threads. Appropriate use allows programmers to avoid problems like the ones outlined above.

三．Synchronization

1. 概述
a) 线程的创建和操作
b) 竞争条件和临界区（critical sections）
c) 原子操作的概念
d) 同步的抽象概念

i. 信号量

ii. 锁和条件变量

2. 一个线程接口

a) class Thread {

```
public:
    Thread(char* debugName);
    ~Thread();
    void Fork(void (* func)(int), int arg);
    void Yield();
    void Finish();
}
```

3. 线程函数

a) Thread（ ）这个构造函数创建一个新线程。它分配一个空间数据结构给 TCB。

b) Yield（ ）方法放弃 CPU 占用来让其他线程使用

c) Finish（ ）方法停止调用线程

4. Fork（ ）

a) 真正让线程开始运行，必须告诉这个线程什么函数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

b) **Fork（ ）方法首先分配一个栈给线程。**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。

有两个不同的线程。一个是系统的线程，另一个是程序中创建的线程。

5. TCB 在运行函数时的设置（Fork（ ）的作用之一）

a) 首先操作系统把 TCB 内的栈指针指向栈

b) 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

c) 操作系统继续设置 TCB 里的寄存器保存第一个参数

d) 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

6. Runnable

系统维持一个可运行的线程的队列。如果处理器空闲，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。

7. 并发线程的执行

a) 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，OS 仅有有限的处理器，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。

b) 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);
}
```

ii. 可能的结果。两个线程同时调用 sum。要完全的明白，我们必须了解 sum 最原始的每一个操作。

1. Sum（ ）最开始把读入数值一个放入寄存器
2. 然后增加这个寄存器内放的数值
3. 然后把寄存器内的值取回给 a
4. 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
5. 最后调用打印数据的 printf 函数

iii. 可能的输出结果

1. 0 ：a = 1
   1 ：a = 2
2. 1 ：a = 1
   0 ：a = 2
   顺序执行两个线程

1. 0 ：a = 2
   1 ：a = 2
2. 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

2. 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

2. 1 ：a = 1

   0 ：a = 1

   在 a 还是 0 的时候，两个 a++同时发生了

8. 不确定的结果
   a) 当并发发生时，它的结果依赖于内部指令的交错顺序
   b) 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
   c) 因此，很难去得到同样的 bug
   d) 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
9. 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。
10. 竞争条件（race condition）和临界区
    a) 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。
    b) 临界区是包含竞争条件的并发程序的一部分
11. 原子操作的概念
    a) 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
    b) 原子操作是建立在一系列原始操作之上。在例子中，原始操作就是独立的机器语言。
    c) 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。
12. Sum 的例子和原子操作

    在我们上述的例子中，build an increment operation up out of movl and addl machine instructions 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。
13. 同步（Synchronization）

    使用同步操作来使编码序列原子化
14. 信号量（semaphore）
    a) 信号量是我们的第一个同步抽象，概念的说，它是一个支持两个原子操作 P 和 V 的计数器
    b) P 将一直等到计数器大于 0，然后对计数器减 1 并返回。

c) V 会将计数器加 1

15. 信号量接口

```
Class Semaphore {
    Public:
        Semaphore( char * name, int value);
        ~Semaphore();
        Void P();
        Void V();
}
```

16. 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);
}
```

17. 互斥（mutual exclusion）

a) 我们在这里使用信号量实现互斥的机理。在互斥的中心思想是在某个时刻只有一个线程可以被允许执行操作

b) 在这个情况下，只有一个线程接受变量 a

c) 用互斥原理来实现原子操作

d) 执行原子运算的代码被叫做临界区（critical section）

18. 生产者消费者问题

a) 概念是生产者产生数据而消费者消费数据

b) 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

c) 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）。

19. 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);
}
```

20. 有限缓冲区的生产者消费者模型

a) 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。

21. 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);
    }
```

22. 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。

23. 锁和条件变量

    a) 信号变量是一个同步抽象概念

    b) 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

24. 锁的接口

```
Class Lock() {
    Public:
        Lock(char * name);
        ~Lock();
        Void Acquire();
        Void Release();
}
```

    a) 锁的运算解释：

        i. 一个锁有两个状态：locked 和 unlocked

        ii. Lock(name)：创建一个 unlocked 的锁

        iii. Acquire()：等待锁的状态为开锁（unlocked）时，将状态设置成锁上状态（locked）

        iv. Release()：利用原子操作把锁的状态从 locked 状态到 unlocked 状态

25. 上锁实现的需求（Requirements for a locking implementation）

    a) 在同一时间，只有一个线程能够获得锁。（安全性，safety）

    b) 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）

    c) All unlocks complete in finite time 所有解锁过程在有限时间内完成 不然锁就一直被占用了。（活跃性，liveness）

26. 锁的实现具有令人满意的性能

    a) 效率：只占用一点点资源

    b) 公平：线程得到锁的顺序与他们请求锁的顺序一致

    c) 使用简单

27. 锁的使用

a) 当使用锁的时候，通常会有多个线程访问锁的数据片段
b) 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
c) 因此，锁允许线程在每一片数据上执行复杂的原子操作

28. 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);
```

}

29. 仅依靠锁实现无限制缓冲区
    a) 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。
    b) 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
30. 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 );
}
```

    a) Condition(name)：创建一个条件变量
    b) Wait(Lock * l)：释放锁并等待。当 Wait()返回锁时，锁可以被再获取
    c) Signal(Lock * l) :启用一个正在等待的线程 ,让他运行。当 Signal()返回锁的时候，锁是一直锁上的
    d) 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
31. 锁和条件变量的使用
    a) 用一个数据结构将锁和条件变量联系起来
    b) 在程序在数据结构上执行一个操作之前，它会获取锁
    c) 如果在它能够执行操作之前需要一直等待 ,使用环境条件变量去等待直到有另一个操作可以将数据结构变成它可以能够继续执行的状态
    d) 在某些情况下，你需要不止一个条件变量
32. 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);
    }
```

33. The Laundromat Example

四．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();
}
```

2. 死锁触发的必备条件
   a) 互斥：只有一个线程能够获得锁
   b) 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
   c) 不可被抢占：拥有锁的进程才能释放这个锁
   d) 循环等待：1 等 2,2 等 3……n 等 1
3. 避免死锁
   a) 锁排序，一直按照排序来获得锁。
   b) 消除循环等待条件。
4. 如果锁在不同的订单中需要被获取，应该做什么
   a) 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。
   b) 如果操作成功，很好。一旦你得到了锁，就没有问题了
   c) 如果操作失败，你的代码将需要释放所有的锁。
5. 死锁预防
   a) 对于互斥——让每个进程请求资源时都尽量快地得到资源
   b) 对于占有并等待——保证当某个进程请求资源时并没有同时持有资源
   c) 对于循环等待——对资源排序
6. 简单的算法实现死锁避免（资源角度）
   a) 每一条进程都告诉它将需要的最大资源数
   b) 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
   c) 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

五． 实现同步操作

1. 怎样实现类似锁的同步时序操作

   利用算法实现同步操作十分慢且类中，所以很多机器是通过硬件支持同步操作的——同步指令。

2. 单处理器

   单处理器机器 中断是使指令失去原子性的唯一因素

   所以在临界区前关闭中断机制 在临界区后开启中断机制

   还是比较有效率的

3. 测试和设定（Test-and-Set）

   测试和设定指令检测内存地址的值是否为 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;

   但是存在 busy-waiting（忙碌等待）的问题 当线程 1 持有锁 l1 同时另一个线程 2 想得到 l1 此时线程 2 就会一直空转直到线程 1 释放 l1

4. Test-and-Set 单处理器

   ```
   while (test-and-set(l) == 1) {
     currentThread->Yield();
   }
   ```

   等待队列：把锁给队列中的第一个线程 这样，队列中的线程发出获得锁的请求不会超过一次

5. Test-and-Set 多处理器

   可能会解锁在另一个处理器上运转的锁 没准需要多运转一会儿 期待另一个处理器释放锁

   cost for each suspension algorithm——三个主要消耗时间 :spinning ,suspending，resuming

   If the lock will be free in less than the suspend and resume time, spin until acquire the lock.

   If the lock will be free in more than the suspend and resume time, suspend immediately.

显然，上述说法实现不了，我们无法预知未来。

6. Test-and-Set RISC 机器

   在精简指令集计算机上，test-and-set 和 swap 实现起来比较困难。

   利用非阻塞的抽象操作：Load Linked(LL)，Store Conditional(SC)来实现同步。

   LL：Load memory location into register and mark it as loaded by this processor. A memory location can be marked as loaded by more than one processor.

   SC：if the memory location is marked as loaded by this processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.

7. Non-blocking

六．CPU 调度

1. 什么是 CPU 调度

   在多线程环境下决定哪一个线程运行

2. 为什么 CPU 调度很重要

   因为 CPU 调度对系统的资源利用（resource utilization）和系统全局表现有很大的影响。

   a) 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（late 80’s, early 90’s）
   b) 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. 未来会发生什么

   更多较新的系统（Windows NT, Linux）回归到复杂的 CPU 调度算法

4. 大多数调度算法背后的基本假设

   a) 有一个进程池（pool）争夺 CPU 资源
   b) 进程是各自独立的，并且在各自抢夺资源
   c) 调度的工作就是把稀缺的 CPU 资源“公平”（根据一些对公平的定义）的分配给不同的进程，在某种程度上优化性能标准

5. 大致来讲，这些基本假设在崩塌

   a) CPU 不稀缺
   b) 很多应用是在多处理器互相配合的基础上设计的

所以，对于调度是调解和分配相互竞争的进程的看法是有些过时的

6. 进程是如何运转的
    a) 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）。
    b) IO 操作持续多长时间？这个取决于进程
7. IO 密集型的进程（IO bound processes）
    a) 执行大量的 IO 操作的进程
    b) 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
8. CPU 密集型进程（CPU bound processes）
    a) 进程执行大量计算并且很少执行 IO 操作
    b) 会导致较长的 CPU 占用时间
9. 调度和 CPU/IO 脉冲（脉冲，burst，指的是对 CPU 的一次性占用时间）
    a) 调度经常要做的事情就是当一个进程执行 IO 时，switch CPU 到另一个进程，
    b) 为什么？IO 会花费很长时间，并且不想在等待 IO 时让 CPU 空闲
10. CPU 脉冲时间分布

    观察整个系统的 CPU 脉冲时间，有指数或超指数分步
11. 进程运行状态（running states）
    a) Running（运行）：进程在 CPU 上运行
    b) Ready（就绪）：准备运行，但是并没有真正在 CPU 上运行
    c) Waiting（等待）：等待一些事件发生，像 IO。
12. 什么时候调度决定替换
    a) 当进程从 running 状态切换到 waiting 状态：

       IO 请求，等待子进程终止，等待同步操作（如获得锁）完成。
    b) 当进程从 running 状态切换 ready 状态：

       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。（If scheduler switches processes in this case, it has preempted the running process.？）**另一个一般的例子就是 IO 完成。
    c) 当进程从 waiting 切换到 ready：

       IO 操作完成或者锁的获取。
    d) 当一个进程终止
13. 如何评估一个调度算法
    a) CPU 利用率
    b) 吞吐量（throughout）：单位时间完成的进程数

c) 周转时间（Turnaround Time）：进程提交到完成总时间
d) 等待时间（Waiting Time）：从 ready 到 running 之间等待的时间
e) 响应时间（Response Time）：提交请求到第一次响应请求的时间
f) 调度效率（Schedule Efficiency）

14. 批处理系统（Batch system）和交互式系统（Interactive system）的巨大差别
a) 批处理系统，典型地需要良好的吞吐量或者 Turnaround Time
b) 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首要考虑的
c) 一些其他的系统，吞吐量或者 Turnaround Time 都不是真正有作用的：一些进程理论上会永远的跑下去

15. 长程调度和短程调度的区别
a) 长程调度是决定哪一个程序可以进入到系统中处理，他们一旦运行起来，他们可能会因为 IO 或者是其他优先权挂起（从缓存中选出等待执行的进程加载进内存，使其进入 ready 状态）
b) 短程调度决定哪一个可运行的进程将被处理器执行（从 ready 状态的进程队列中选出一个来执行，分配 CPU 资源）

16. 先到先服务（FCFS）调度算法
a) 一个就绪队列，操作系统运行队列最前面的进程，新的进程进入到队列的尾部
b) 一个进程不会放弃 CPU 知道它终止或者是执行 IO 操作

17. FCFS 算法的性能
a) 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）。
b) 如果接受顺序是 P1，P2，P3，那么
i. 等待时间（waiting time）：（24+27）/3 = 17
ii. 周转时间（turnaround time）：（24+27+30）/3 = 27
iii. 吞吐量（throughput）：（24+3+3）/3 = 10
c) 如果进程进入顺序是 P2，P3，P1 的话
i. 等待时间（waiting time）：（3+6）/3 = 3
ii. 周转时间（turnaround time）：（3+6+30）/3 = 13
iii. 吞吐量（throughput）：（24+3+3）/3 = 10

18. 最短进程优先算法（SJF）调度算法
a) SJF 算法可以减少一些等待时间和周转时间的变异 事实上，这是平均等待时间最小的算法
b) 巨大问题：调度程序如何指出一个进程将会运行多长时间

19. 对于长期调度：用户预测执行时间并上报

a) 长期调度在批处理系统下，用户会给一个评估

b) 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

20. 对于短期调度：利用过去预测未来

a) 必须有过去来预测未来

b) 标准方法：用以前 CPU 区间每个进程时间延迟的指数平均数

c) $S_{n+1} = w\,T_n + (1 - w)S_n$: T_n 是 CPU 实际区间长度，S_n 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。

21. 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

22. 抢占式的 SJF vs 非抢占式的 SJF

a) 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级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。

b) 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，它允许每一个 running 进程完成它的 CPU 周期

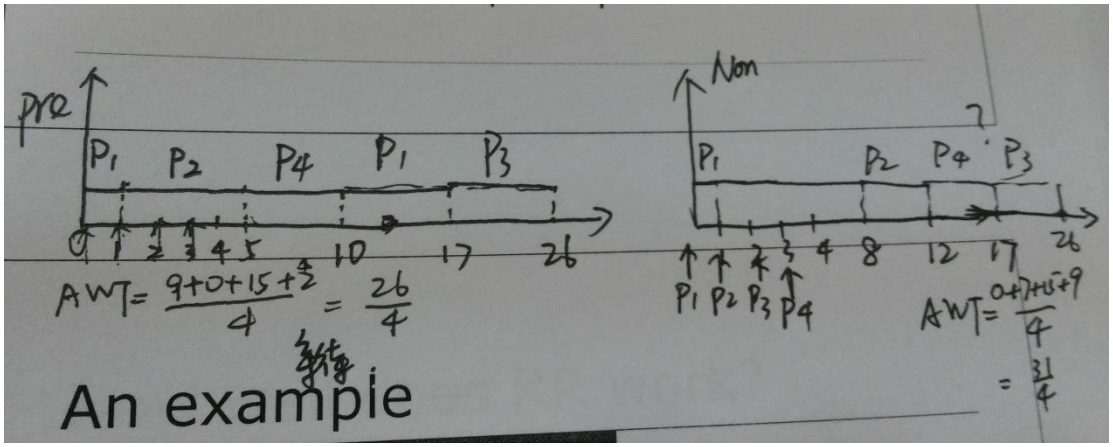
23. 一个例子

a) 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） 到达的顺序是 P1，P2，P3，P4 间隔 1 个时间单位陆续到达

b) 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）

c) 一个抢占式的 SJF 调度会怎么做

d) 一个非抢占式的 SJF 调度会怎么做



24. 优先调度

a) 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

b) 如果多条进程都是一个优先级别，用一些其他的标准-------例如先到先服务调度（FCFS）

c) **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

25. 一个例子

a) 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。**

b) 一个标准的优先调度会怎么做

26. 饥饿（starvation）

a) 优先调度的一个大问题：低优先级进程的饥饿或者阻塞

b) 可以使用 aging 来避免——逐渐提高系统中长时间等待的进程的优先级

27. 交互式系统的调度

a) 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

b) 因此，使用一个叫做循环调度（round-robin scheduling）RR 调度

1) 有点类似于 FCFS 但是有抢占机制
2) 有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

28. 实现循环调度需要时钟中断

a) 当调度一个进程时，在时间段到期后设置一个离开定时器

b) 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

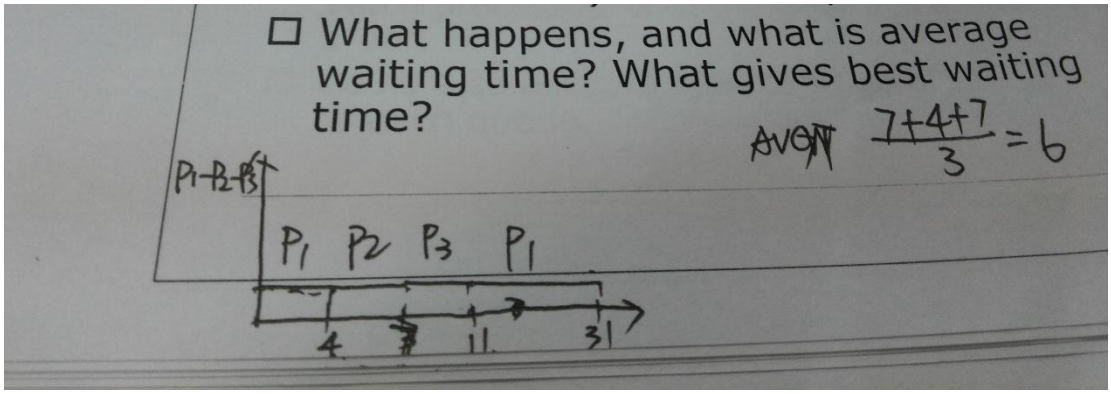
c) 但是如果进程的时间段到期，需要做一个上下文切换。保存正在运行的进程的状态并且运行下一个进程

29. RR（round-robin）调度如何良好的工作

a) 响应时间是好的，但有时候会有糟糕的等待时间

b) 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

c) 发生了什么？平均等待时间是多少？什么给了最佳等待时间？



30. 如果是一个很短的时间周期 RR 调度会发生什么

a) 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

b) 小量子时间片上下文切换的开销问题

31. 拥有一个小量子支持的硬件会怎么样

多线程或者超线程

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

32. 时间周期很长会怎么样

a) 相当于 FCFS

b) 经验法则——让 80%的 CPU 中断比时间段（time quantum）更短

33. 多队列调度（Multilevel Queue Scheduling）

a) 多队列调度就像 RR 调度算法，不过是拥有多个队列而已

b) 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

c) 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

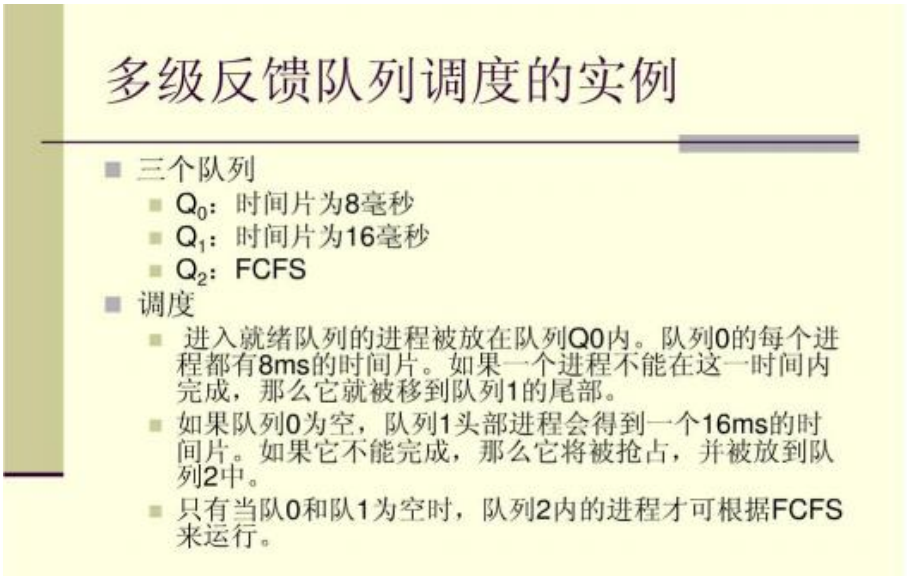
d) 当然也可以用 CPU 百分比形式分给每个队列(Could also allocate a percentage of the CPU to each queue）

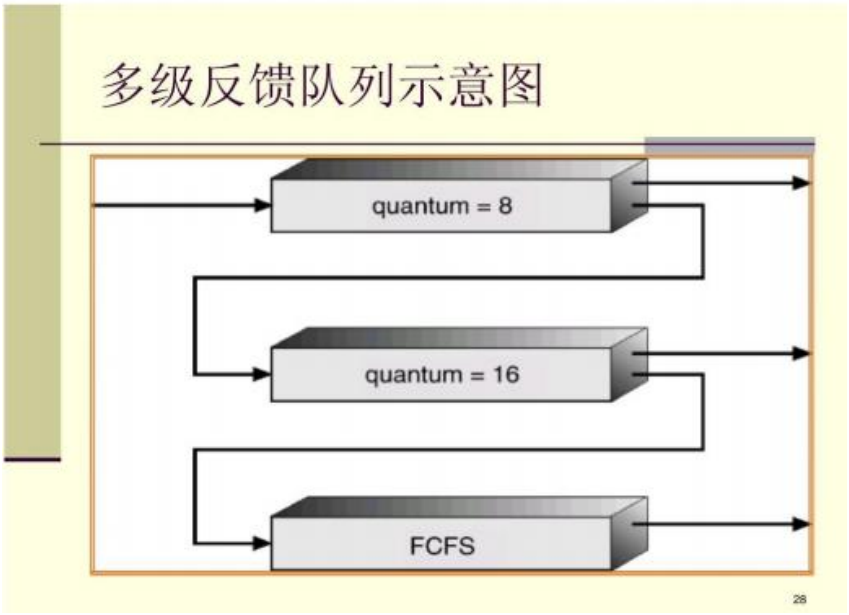
34. 多级反馈队列调度算法

a) 多级反馈队列调度算法类似多级队列调度算法 ,只不过进程能够在不同优先级的队列之间进行转移

b) 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

c) 也能通过增加等待时间过程的进程的优先级来防止饥饿现象的产生





35. 传统 Unix 调度

a) 另一个**多级队列反馈调度算法**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

b) 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

c) 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

d) 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

e) 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

f) 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

g) 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。

   i. CPU usage = CPU usage/2

   ii. Priority = CPU usage/2 + base priority

h) 因此，当一个进程没有立刻使用较多的 CPU，它的优先级上升

i) 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

j) 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：

   i. Priority = CPU usage /2 + base priority + nice value

k) 因此，你可以恰当的减小你的进程的优先级到其他进程

36. Convoy effect

37. 多队列反馈调度算法是复杂的

七．OS 集锦

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：

   ① 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读取数据

   ② 建立子进程时。OS 必须为子进程的建立进行一些设置

   ③ 发送包到互联网时。OS 控制网络接口

2. 为什么进程不能够直接的做这些工作

   a) 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

   b) 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。应用是不包括专门适配硬件的代码的

   c) 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。让 OS 做这些操作可以消除应用这些安全问题。当然，应用程序需要信任 OS。

3. 进程是如何调用 OS 函数的

   a) 通过生成一个系统调用

   b) 概念上，进程调用一个与该进程分离的子程序（subroutine）去执行必须的函数

   c) 但是 OS 必须在一个不同的保护域执行应用。通常的，OS 在 supervisor mode 中运行，这是一个一个只允许运行正确的操作的监督模式。（Typically, OS executes in supervisor mode, which allows it to do things like manipulate the disk directly.）

4. 系统调用指令（system call instruction）

   a) 从用户模式（user mode）下切换到监督模式（supervisor mode），大多数机器提供一个系统调用指令。

   b) 这个指令引起一个异常（exception）。硬件从用户模式（User mode）转换到核模式（Supervisor mode）会调用操作系统中的异常处理程序。

   c) 一个约定俗成的例子，进程用来与 OS 交互操作

5. 一个例子——开放系统调用

   a) 系统调用通常以一个普通的子程序调用开始。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）。

```
/* Open the Nachos file "name",
 * and return an "OpenFileId" that can
 * be used to read and write to the file.
 */
OpenFileId Open(char *name);
```

6. 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。

   ```
   Open:
           addiu $2,$0,SC_Open
           syscall
           j       $31
           .end Open
   ```

   约定俗成，Open subroutine 在 register 2 中放一个数字（在本例中时 SC Open）。操作系统可以通过 register 2 中的数字来知道需要进行哪一个系统调用。

7. 系统调用中的参数传递
   a) 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
   b) 按照惯例，编译器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的时候。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
   c) 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
8. 异常处理程序内部
   a) 在异常处理程序内部，OS 会分析出应该获取何种动作，执行何种动作，然后返回到用户程序中。
9. 另外几种异常
   a) 例如，如果程序企图使用一个 NULL 指针，硬件产生一个异常。OS 会指出发生了哪种异常并相应地处理它。另一种异常就是除以 0。这些异常都会引起 mode 的切换。
10. 在中断中发生的类似的事
    a) 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
    b) 中断与异常的不同之处在于：
    中断是由外部事件触发的（比如磁盘 IO 操作完成、新的字符串被输入控制台等）
    异常是由正在运行的程序触发的。
11. 目标文件格式

a) 运行一个进程，OS 必须从硬盘中加载一个可执行文件(executable file)到内存。这类文件都包含什么？

b) 包含 代码，初始化的数据，未初始化的数据会占多大空间的具体说明。可能会帮助调试器运行的信息。

c) 编译器，链接器，OS 必须对可执行文件的格式达成共识。

12. 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;
```

13. 当 OS 载入一个可执行文件时会做些什么

a) 读取头文件

b) 检查幻数（magic number）是否匹配

c) 指出这个进程需要多大空间。这包括了栈的空间，代码，初始化数据和未初始化数据。

d) 如果它需要在物理内存中保持全部的进程，那么它会先找到足够的物理内存空间。

e) 然后从文件中读取代码段到物理内存中

f) 然后从文件中读取初始化数据片段到内存中

g) 它将栈和未初始化的内存置零

14. I/0 在操作系统中的概述

a) 有两种基本的方式来管理 IO

i. IO 内存映射（memory mapped IO）: IO 设备上的控制寄存器映射到处理器的内存空间上。处理器通过在 IO 设备映射到的内存空间地址上读和写控制设备。

ii. IO 程控化(programmed IO):处理器有一个特别的 IO 指令像 IN 和 OUT。这些指令直接控制 IO 设备使之正常工作。

15. 设备驱动接口（Device driver interface）

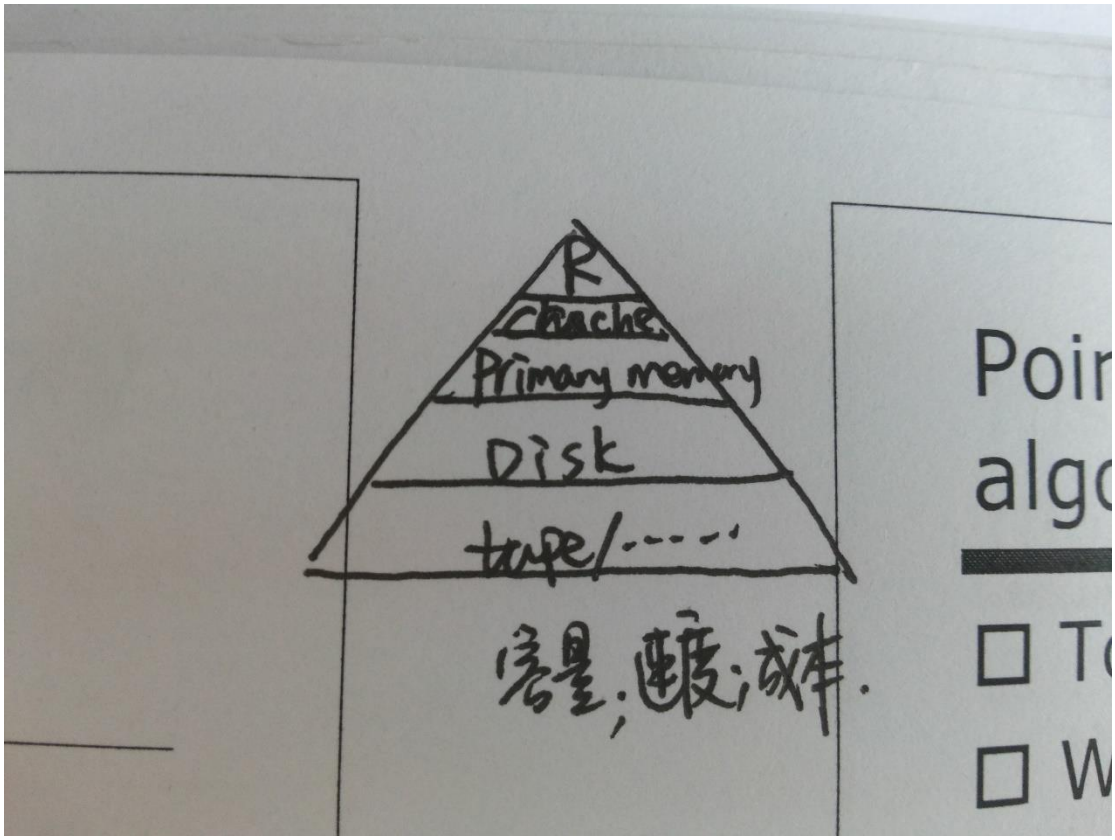
通过写低级、复杂的代码去控制设备是十分有技巧难度的工作

OS 封装了这些代码并形成标准接口。剩下的操作系统的别的部分就可以使用这些接口，而无需处理复杂的 IO 代码了。

例子：Unix 有 block device driver interface

16. 异步 I/O
    a) 典型地，对于处理器来说 IO 是异步的
    b) 因此，处理器将开始一个 IO 运算（如写磁盘扇区），然后离开，执行其他进程
    c) 当 IO 操作完成，它中断处理器。处理器接过来然后继续做应该做的操作。

八．内存管理



1. 内存管理算法的要点
    a) 支持内存（主存 main memory）的共享
    b) 我们将集中于多进程共享同一块物理内存
2. 内存管理的关键
    a) 保护。必须允许一个进程保护它的内存，防止另一个进程访问它的内存。

b) 识别（identify）。 进程是如何识别内存中共享内存区域

c) 透明度。共享过程的透明度

d) 效率。任何内存管理策略不应该带来太大的负载（performance burden）

3. 为什么要在进程之间共享内存？

a) 因为我们想要处理器实现多进程

b) 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

4. 内存管理的历史发展

a) 第一台计算机，将一个程序加载进入机器并进行执行运算，没有分享的需求。OS 只是一个子程序库，没有保护机制。

b) 为了提高处理器的利用率，也注意到了 IO 占用的时间很长，打算采取多任务编程。一个进程会持续执行知道它进行 IO 操作，OS 会让另一个进程跑。

那进程间怎么共享内存呢？

i. 将两个进程都加载进内存，在 OS 的控制下切换。

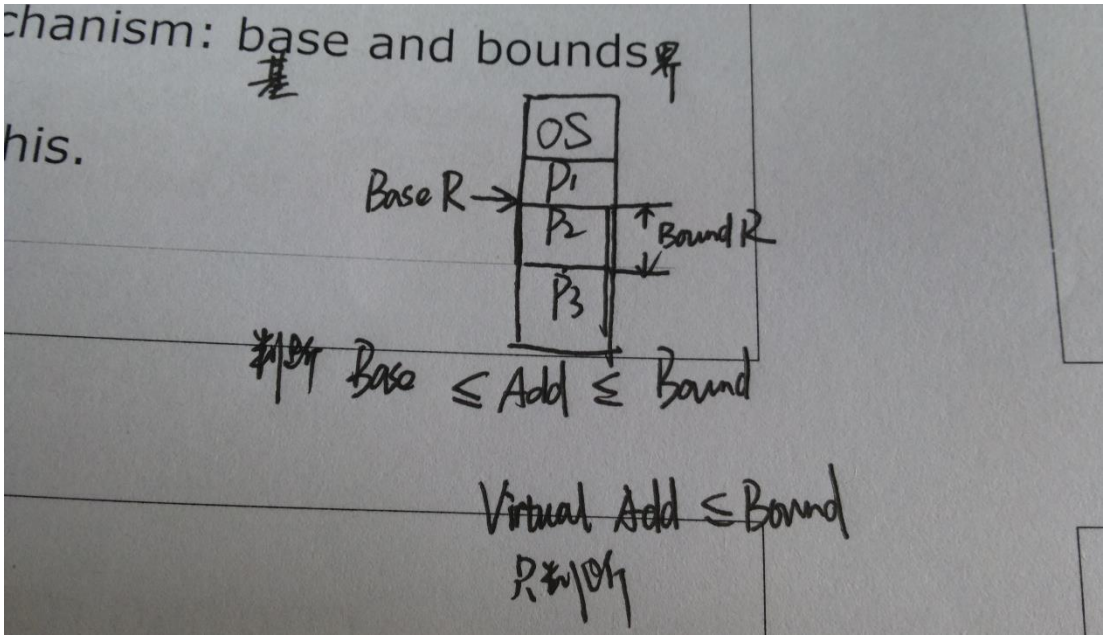
保护性的缺失 一个进程的 bug 会杀死另一个进程

ii. 把整个进程空间交换到磁盘

性能问题

iii. 对每一个内存引用进行访问检查 看它是否在属于该进程的内存区域内

通常机制：base and bounds registers



5. 内存检测在哪里完成
   a) 在硬件中完成，为了速率
   b) 当 OS 运行进程时，加载这个进程的 base and bounds registers
   c) 注意：有一个翻译过程（physical address=virtual address + base register）。程序产生虚拟地址通过翻译变成物理地址。不过，不会有保护问题：一个进程不能访问另一片内存，因为这超出了它的地址空间，自动认为是外部地址空间。如果它尝试访问另一片内存，硬件将产生异常。
6. 内存分配
   a) 将物理内存动态分配给进程（当进程进入或结束）的模型。
   b) 许多分配策略：最佳适配（best fit），首次适配（first fit）等
   c) 都会产生外部碎片（external fragmentation）。最坏的情况，实际上内存具有足够的空间来加载进程，但都是碎片化的空间。
7. 防止外部碎片的方法：分页（paging）

   将物理内存以一个个固定大小的块的方式分配给进程。这个固定大小的块叫做帧(page frame）

   ① 用页帧（page frame）的方式分配物理内存。提供给应用单一的线性地址空间的抽象。
   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 当进程得到一个地址，将这个地址翻译成物理页帧地址。
8. 分页中的虚拟地址：
   ① 虚拟地址由两部分组成：页码（page number）和页偏移量（offset）
   ② 页大小一般为 2 的指数倍
   ③ 为了得到地址所对应的的数据，OS 会做一下操作
      i. 得到页的号码
      ii. 得到偏移量
      iii. 将页码翻译成物理页帧的 id
      iv. 通过偏移量在物理页帧中找到数据
9. OS 是如何翻译的呢

   页表（page table）

   页表是一个线性数组，用虚拟页码编号，其中存的是对于的物理页帧的 id

   - 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.

10. 分页的保护机制

    一个进程不能访问另一片内存区域，除非它的页表指向那块区域。所以，如果两个进程的页表指向两个不同的物理页，进程之间不会存在内存分享。

11. 分页的内存分配

    物理空间的固定大小分配大大简化了分配算法。OS 只要记录下来空的以及被占用的页。不存在外部碎片。

12. 内部碎片（internal fragmentation）依旧存在

九．分页简介

1. 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 提供给应用一个抽象的单一线性地址空间。

   ③ 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

   ④ 当进程运行一个地址，动态地将地址翻译为保存数据的物理页帧。

   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。

2. 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.

3. 分页中的地址翻译

   简单的方法：利用页表（每个进程有一个属于自己的页表）

   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.

4. 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。

   一次是查页表，还有一次是真正的内存访问得到数据。

5. TLB（Translation Lookaside Buffer 转换后备缓冲器）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：全相联的（fully associative）、直接映射的（direct mapped）、组关联的（set associative）等

   ④ 利用循环空间可以使直接映射的范围更大

6. TLB 查询如何工作

   Extract page number.

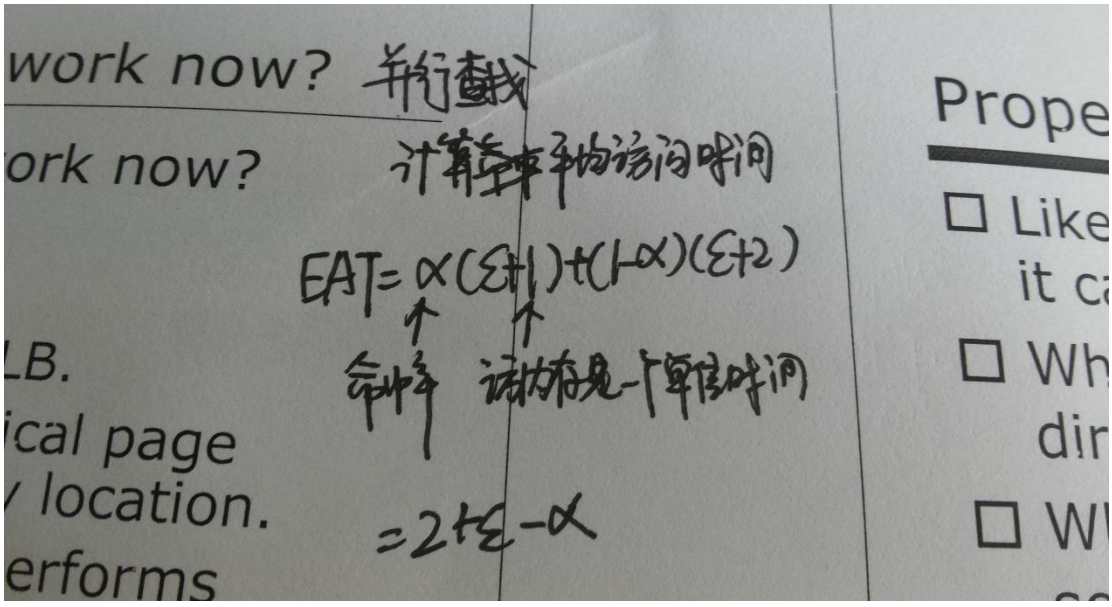
   Extract offset.

   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.

   命中率以及平均查找时间



7. 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 只要跟踪空闲或者使用过的页，并在进程需要内存时给它分配页

③ 不会有外部碎片产生

8. 仍然有内部碎片
9. 进程如何共享内存
   ① OS 让页表指向同一物理页帧
   ② 有用且快速的内部进程间通信机制
   ③ 允许快速且透明的共享
10. 保护措施

    有很多种保护方式：

    1) 防止一个进程读写另一个进程的内存
    2) 防止一个进程读另一个进程的内存
    3) 防止一个进程读写部分自己的内存
    4) 防止一个进程读部分自己的内存
11. 阻止一个进程读写内存

    阻止读与写：OS 拒绝建立虚拟地址与被保护的物理内存之间的映射。当程序视图访问这片内存时，OS 会产生错误。如果用户程序捕捉到错误，可以采取行动使程序正常。

    阻止写可以读：在 TLB 中设置写保护位（write protect bit）。如果程序试图写。产生异常。如果只是读，没有错误。
12. 虚拟内存简介

    磁盘作为主存的拓展
13. 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。然后之后需要之前的页时，再从磁盘里交换回来。
14. 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，选择一页将他存到磁盘上。可以 overlap IO 和存出页。
    ② 页帧大小等于磁盘块大小。写数据是以磁盘块为单位的。
    ③ **Do you need to allocate disk space for a virtual page before you swap it out? (Not if always keep one page frame free) Why did BSD do this? At some point OS must refuse to allocate a process more memory because has no swap space. When can this happen? (malloc, stack extension, new process creation).** **?**
15. 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

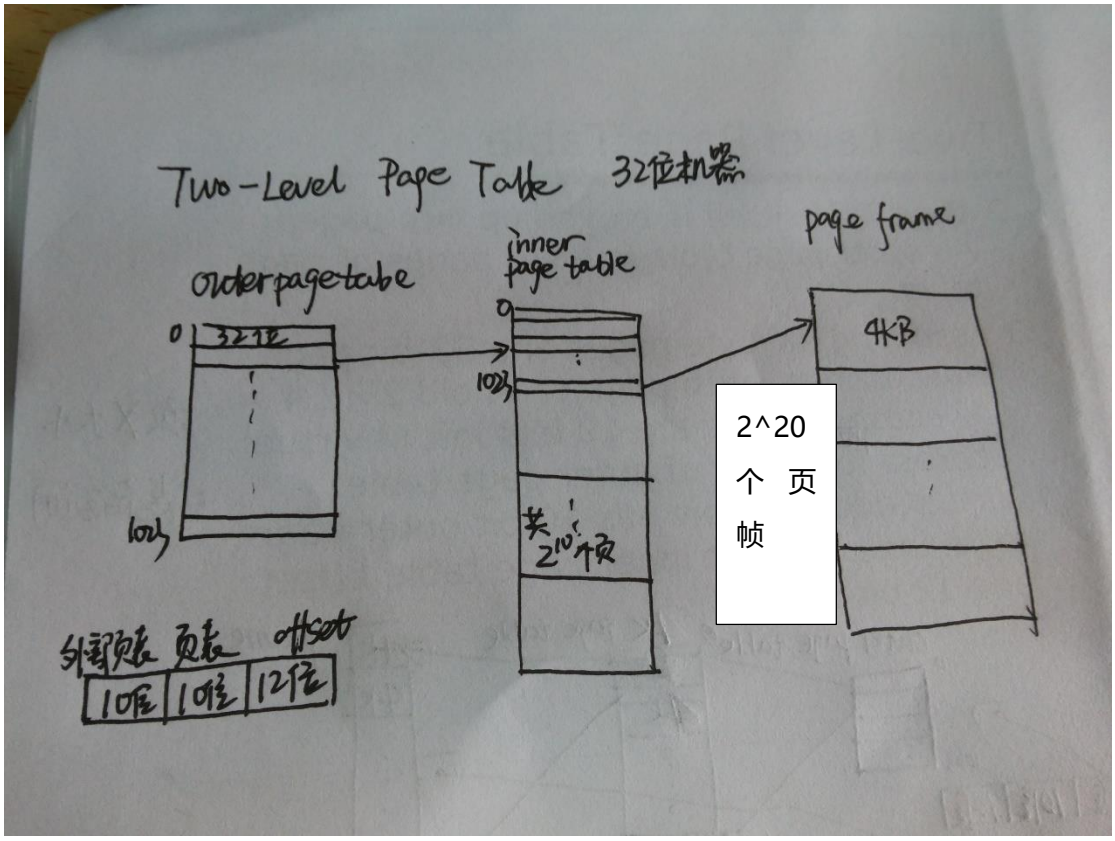
读入新找到的空闲帧中，并重启进程。

16. 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 程序间灵活共享内存（可能这些程序的地址空间的总数超过机器的物理内存）
    ③ **Supports a wide range of user-level stuff - See Li and Appel paper.** ?
17. 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
    在极端情况下，页表会占相当大的一部分虚拟内存。
    解决办法：将页表再分页/用更复杂的数据结构来存储虚拟到物理的翻译
    ③ 翻译开销

十．分页与虚拟内存

1. 页表结构
   页表储存在机器的物理内存中。
   页表占多少空间？页表怎么分配空间？（连续分配？块分配？给页表分页？）
2. 页表与 TLB 未命中
   若 TLB 中失靶，那 OS 就要去内存里的页表里找。如何设计页表使之命中率更高。
3. 稀疏地址空间的抽象
4. 几种页表
   ① 线性页表
   对于页表分配，面临可变大小的分配问题。
   - 页表会占据很大一部分空间。32 位机器，4KB 大小为一页，有 $2^{32} / 2^{12} = 2^{20}$ 项，每项占 32 位的话，需要 4MB 空间来存页表。
   - 对于稀疏地址空间的支持性不好，浪费很大
   - TLB 失靶处理很简单

   ② 二级页表
   对页表再分页。再多一外部页表。



二级页表寻找步骤

- ☐ Find physical page containing outer page table for process.
- ☐ Extract top 10 bits of address.
- ☐ Index outer page table using extracted bits to get physical page number of page table page.
- ☐ Extract next 10 bits of address.
- ☐ Index page table page using extracted bits to get physical page number of accessed page.
- ☐ Extract final 12 bits of address - the page offset.
- ☐ Index accessed physical page using 12 bit page offset.

对二级页表的评估

- 消除了页表的可变大小分配问题。
- 依旧存在内部碎片问题。
- 如果页表还是占了很大空间。can page the pages of the page table 再分页。但是外部的页表要存在内存中。
- 对稀疏地址空间的支持性好。
- 增加了 TLB 失靶后的时间。需要两次查表寻找。

③ 三级页表

64 位机器的选择。外部页表可能大于一个页的大小。

SPARC 使用三级页表。

5. 反向页表

每个物理内存页帧都有对应的一项 ,其中标明拥有该页的进程以及页帧对应的虚拟地址。

TLB 失靶时，在反向页表结构中寻找虚拟地址对应的物理页帧。可以用 hashing/ Associative table of recently accessed entries 加快查找速度。

若需要的页不在内存中，需要去磁盘中寻找。类似标准的页表。

6. 页表的基本任务

① 实现 TLB 重载

② 维护常驻物理内存中的页的映射

7. 硬件与 TLB 关系

硬件只负责使用 TLB，而软件处理 TLB

8. 共享代码和数据

① 不同进程的页表项指向同一物理页帧，则共享内存。一个进程写数据，另一个进程可以看见这些改变：高效交流

② 一个被编译的 code 保存在内存中，其他编译工具均可执行：帮助提高内存利用

③ 共享代码必须出现在所有进程逻辑地址的同一位置

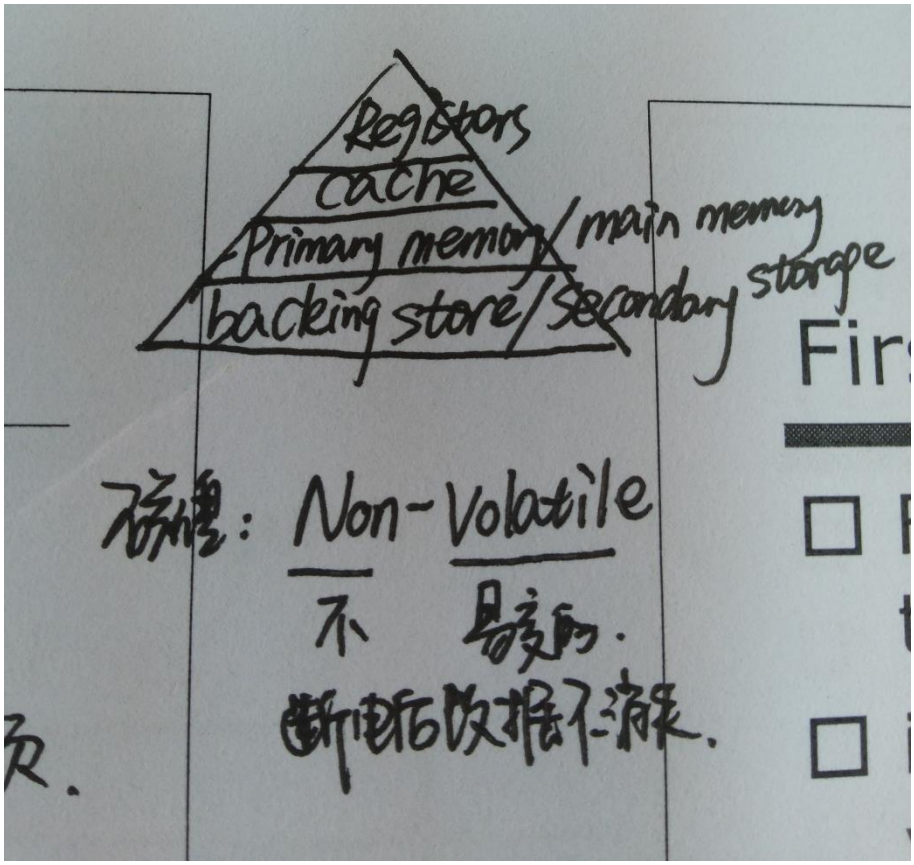
9. 重入代码（reentrant code）的概念

① 重入代码不可以修改自身，需要给每个进程的全局变量一个单独的 copy

② 所有 Nachos 内核代码需要重入，单独 copy

③ 私有代码和数据页可以出现在逻辑地址空间的任何地方

10. 虚拟内存的主要思想



① 主存作为后备存储（backing memory）的高速缓存

② 常用解决方案：按需分页（demand paging）

③ 一页可以常驻在磁盘或者主存中

11. 按需分页的第一个扩展

① 有效位（valid bit）

② 每一个页表或者 TLB 的项都有一个有效位。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。

12. 管理有效位和页表

① OS 负责磁盘与内存间页的转移

② 设置有效位和页表的建立

13. 发生页错后 OS 将如何处理？

① Trap to OS.

② Save user registers and process state.

③ Determine that exception was page fault.

④ Check that reference was legal and find page on disk. //

⑤ Find a free page frame.

⑥ Issue read from disk to free page frame.

⑦ Queue up for disk.

⑧ Program disk controller to read page.

⑨ Wait for seek and latency.

⑩ Transfer page into memory.

⑪ As soon as the controller is programmed, allocate CPU to another process. Must schedule the CPU, restore process state.

⑫ Take disk transfer completed interrupt.

⑬ Save user registers and process state.

⑭ Determine that interrupt was a disk interrupt.

⑮ Find process and update page tables.

⑯ Reschedule CPU.

⑰ Restore process state and resume execution.

OS 获得控制权→保存用户寄存器和进程状态→确定异常是发生了页错→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→安排磁盘控制器读取页→寻找和延迟等待→将页转移到内存中→当控制器 programmed 完成，将 CPU 分配到另一个进程(不是当前这个发生页错的进程 ,这个进程现在处于 waiting state) , 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→找到原来的进程并更新页表→重新调度 CPU→恢复进程状态并恢复运行（那个发生页错的进程）

14. 有效访问时间（Effective Access Time）

    EAT = (1-p)*100 + p*25*10^6. P 是页错率

    If we want overall effective access time to be 110, less than one memory reference out of 2.5 * 106 can fault.

15. 未来趋势

    增加内存或者减小计算

16. 在哪里交换

    ① 交换空间（swap space）——为分页指派的专门一部分磁盘

    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

    ③ 任何实现了文件系统的设备上，分页系统都能正常工作。可以在没有硬盘的工作站（workstation）利用文件系统进行远程分页。

17. 关于交换的更多声明

    ① 后备存储不是把每一个进程数据都存储下来

    ② 可运行的代码 . Can just use executable file on disk as backing store. (Problem: recompilation). ？

③ 在未初始化的数据段上的未引用的页。Just zero the page on first access - no need to access backing store. Called zero on demand paging.

18. 页置换算法（page replacement algorithm）
    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
19. 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。
20. Use bit & Dirty bit

    硬件上提供了两个位来帮助 OS 建立更合理的页替换法则。

    Use bit： 当页被访问时设置该位为 1

    Dirty bit：当页被修改时设置该位为 1
21. TLB 一致性
    ① TLB 项与页表项需要一致
    ② 另一种方法在软件中综合这些位让 TLB 重载更快
22. 页置换算法
    ① FIFO 算法

       缺点：可能弹出多次使用的页
    ② Belady’ s anomaly

       增加内存可能导致更多的页错
    ③ LRU 算法（Least Recently Used 置换最长时间没有使用的页）

       注意运算。
    ④ LRU 近似算法

       走得很慢的时钟。很多页会被标上相同的时间，精确度较低的 LRU 算法。通过 Use bits 实现。
    ⑤ FIFO 二次机会算法

       建立一个交换页的队列。当队首的页的 use bit=1，表示这个页最近被使用过，就把 use bit 清零，然后把这个页放到队尾，仍使它存在内存中，队列指针指向下一个页。若 use bit=0，替换。
    ⑥ 加强的 FIFO 二次机会算法
       - use = 0, dirty = 0: Best page to replace.
       - use = 0, dirty = 1: Next best - has not been recently used.
       - use = 1, dirty = 0: Next best - don't have to write out.
       - use = 1, dirty = 1: Worst.

23. 提前把页 free up

    Keep a list of modified pages, and write pages out to disk whenever paging device is idle. Then clear the dirty bits.

    Increases probability that page will be clean when it is ejected, so don't have to write page out.

24. 空页帧池

25. 如果硬件没有实现 use bit & dirty bit

    如果硬件支持 valid bit & readonly bit，OS 可以实现 use bit & dirty bit 功能。

    实现 use bit：每次清空 use bit 时清空 valid bit。OS 随时跟踪页的真实状态，当发生页错时，OS 发现这页是在内存中的，那就设置 use bit 和 valid bit。

    实现 dirty bit：在 clean 的页上设置 readonly bit。OS 随时跟踪页的真实状态，当第一次出现尝试写内容发生的异常时，OS 设置 dirty bit 同时清空 readonly bit。

    这个策略的好处：

    with this scheme, don't have to rewrite page table entries when eject an entry from TLB.

26. 工作集的概念

    ① 一个进程经常访问的页的集合

    ② 工作集可能随时间改变

    ③ 每个进程在每个时刻都有一个工作集

27. 预留空间

    在 backing memory 中需要预留出一定的空间。

28. 什么需要被初始化？

数据段/未初始化的数据段的页可以直接置零/可以用可执行文件来储存代码/ Makes sense to preload much of code segment to make startup go faster. Of course, must be prepared to page even during startup - what if init data segment does not fit in available physical memory?

29. 系统颠簸

    ① 如果物理内存太小以至于无法装下所有运行中的进程的工作集，就是系统颠簸

    ② 颠簸的原因是进程一直忙于将页面换进换出

    ③ 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

    ④ 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

    ⑤ 消除颠簸的方法：降低 multiprogramming/将所有进程都交换到 backing store

里然后挂起进程

30. 页错率解决办法
    ① 首先要有一个 ideal page fault frequency
    ② 页错率高，分配更多的帧
    ③ 页错率低，分一些帧给其他进程
    ④ 当所有进程页错率都高时，挂起其中一个进程
31. 页大小
    ① 为什么不大些？
        1) 传统，以及已经存在的代码
        2) 增加内部碎片
    ② 为什么不小些？
        1) 相同大小的工作集，页小容易产生页错
        2) 需要更多的 TLB 项
        3) 需要更大的页表

十二.文件系统简介

1. 文件系统的重要性
   最重要的信息都储存在文件系统中
2. 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
3. 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
4. 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
5. 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
6. Unix 的文件
   在 Unix 系统中，文件的意义取决于一系列字节（文件开头）。
   唯独的例外：目录和 symbolic link 是显式的指出的

7. DOS 的文件

   OS 只识别.com/.exe/.bat 后缀的文件

   其他后缀的文件 OS 不识别，是别的程序识别的

8. 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

9. 程序如何访问文件？

   注：打开（open）一个文件，就是将其从磁盘中加载进入内存

   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 索引访问：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。

10. 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 顺序访问->顺序排列，方便访问

    ② 直接访问->磁盘块表(disk block table)

    ③ 索引访问->二级索引（two-level index）

11. 多种文件格式的优缺点

    优点：很容易找到可以合适的打开程序

    缺点：导致 OS 变大，系统很难使用

12. 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 绝对（absolute）路径和相对（relative）路径

13. 文件共享

    有时树的不同部分需要共享，树就变成了图。Unix 支持两种链接。

    ① 符号链接（symbolic link）：指向存放位置

    ② 硬连接（hard link）：指向文件本身

Linux链接分两种，一种被称为硬链接（Hard Link），另一种被称为符号链接（Symbolic Link）。默认情况下，ln命令产生硬链接。
--硬连接
硬 连接指通过索引节点来进行连接。在Linux的文件系统中，保存在磁盘分区中的文件不管是什么类型都给它分配一个编号，称为索引节点号(Inode Index)。在Linux中，多个文件名指向同一索引节点是存在的。一般这种连接就是硬连接。硬连接的作用是允许一个文件拥有多个有效路径名，这样用户 就可以建立硬连接到重要文件，以防止“误删”的功能。其原因如上所述，因为对应该目录的索引节点有一个以上的连接。只删除一个连接并不影响索引节点本身和 其它的连接，只有当最后一个连接被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除。

【软连接】
另外一种连接称之为符号连接（Symbolic Link），也叫软连接。软链接文件有类似于Windows的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

14. Soft link =symbolic link 的使用

    双方共享文件。建立源目录。是文件系统中十分有用的结构工具。

    Symbolic link 是包含一个文件名字的文件。使用到它的时候，自动把其包含的文件名宏替换。（ ln13 p22 ）
15. Graph structure introduces complications
16. 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。

    内存映射文件是指将一部分的文件映射到进程的地址空间上 ,然后进程可以读和写普通的内存一样来读写文件。
17. 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
18. 访问控制的两种标准机制
    - Access lists
    - Capabilities
19. 基于组的文件保护（ group-based ）

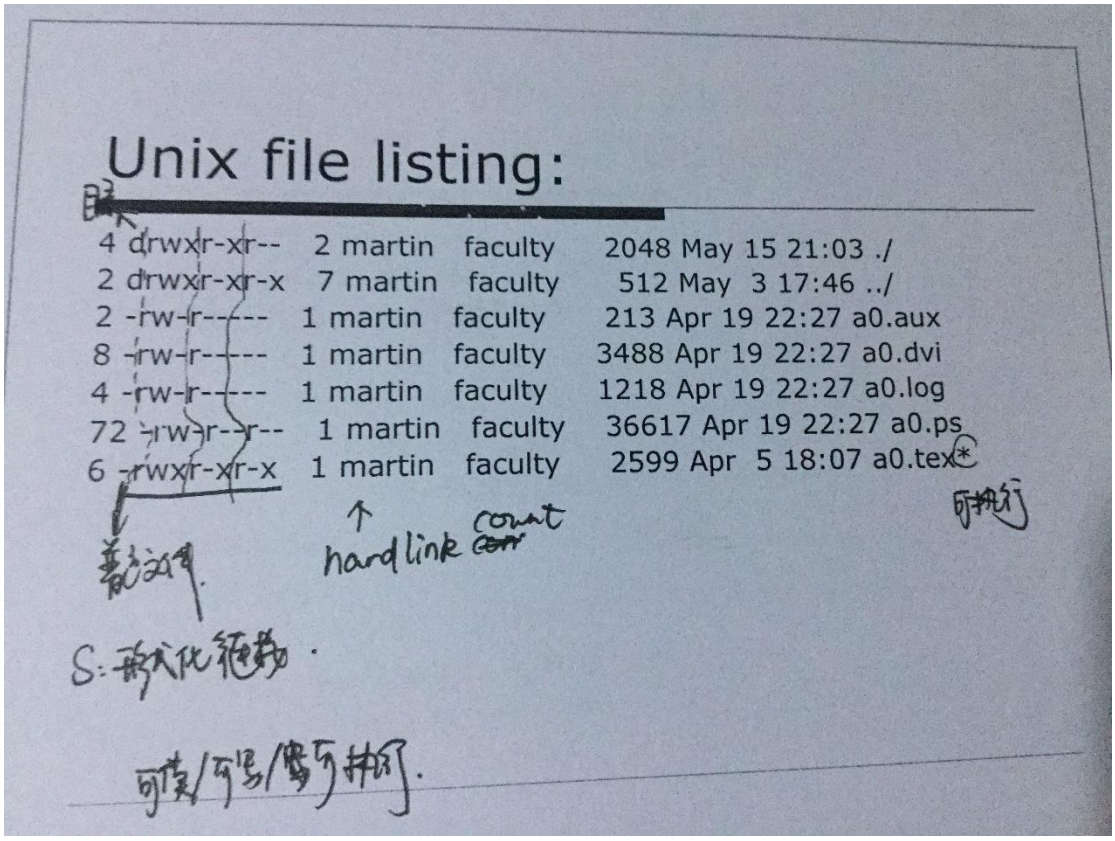
20. Unix 安全模型

    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 是简单原始的保护策略



21. 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

22. OS 怎么访问磁盘

磁盘控制器。OS 对磁盘控制器发出指令。In effect, disk is just a big array of fixed-size chunks. Job of the OS is to implement file system abstractions on top of these chunks.

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。

2. 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，几乎不需要寻找时间
   ③ 直接访问也很快——只需要寻找然后读取
3. 连续分配的缺点
   ① 文件长度很难增长
   ② 需要整个的移动文件，即使文件很大
   ③ 外部碎片
   ④ 需要压缩，要很高的成本
4. 链接分配

   所有文件存放在固定大小的块中。并像链表一样将相邻的块链接起来
5. 链接分配的优点
   ① 文件大小可以任意增长减小
   ② 没有更多的外部碎片
   ③ 不需要压缩或重分配文件
6. 链接分配的缺点
   ① 直接访问很麻烦——不得不一直读取下一个块的指针（因为指针存在不同的磁盘块中）
   ② 需要花费很长的寻找时间
   ③ 可靠性不强——一旦某个指针遗失了，问题就大了
7. FAT（File Allocation Table）文件分配表

   用一个表来存下一个块的指针（磁盘块序号）。还是需要一个个找下一个指针，但是至少不用再去每个磁盘块里找下一个指针了。可以把 FAT table 缓存过来，在内存中进行这个遍历的过程。
8. FAT 空闲块的分配

   文件中最后一个磁盘块指针有 EOF 值。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。
9. 索引分配

   给所有的文件一个索引表。每个索引的项指向包含真实文件数据的磁盘块。支持快速的直接文件访问，顺序访问的效率也不低。

   每个文件有一个专属的索引表（类似页表），存在文件系统中。
10. 如何分配索引表

    像别的文件一样被存在文件系统中。连续分配、链接分配、多级索引分配都可以用。

11. 典型的 unix 文件系统的分布（磁盘块大小为 4KB）

- First 8KB - label + boot block.
- Next 8KB - Superblock plus free inode and disk block cache.
- Next 64KB - inodes. Each inode corresponds to one file.
- Until end of file system - disk blocks. Each disk block consists of a number of consecutive sectors.

12. 索引节点（iNode）

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

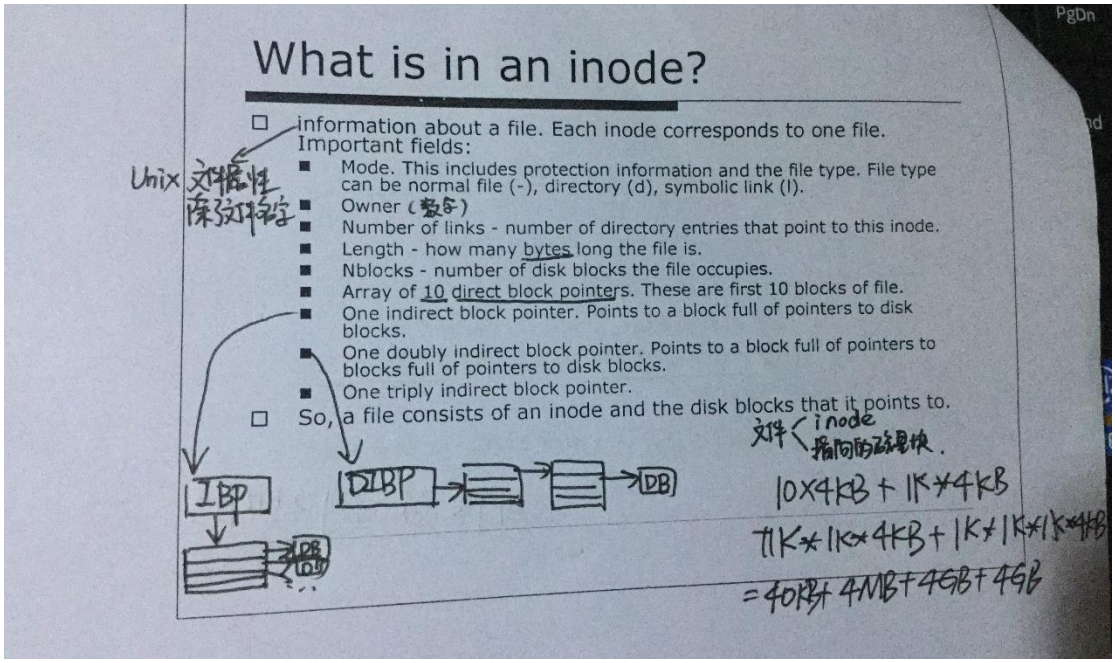
⑥ 十个直接块指针组成的数组

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。



13. 目录

是一系列键值对（name，iNode number）

14. 超级块（superblock）和索引节点（iNode）

    超级块帮助分配 iNode 和磁盘块。它存储了核心信息，需要备份到磁盘上。

15. 超级块包含以下内容

    a) 文件系统的大小
    b) 文件系统 Free block 的数目
    c) 文件系统的可用 Free block 的列表
    d) 在 free block 列表中下一个 free block 的索引
    e) Inode 列表的大小
    f) 文件系统中 Free inode 的数目
    g) Free inodes 的缓存
    h) 在 inode 缓存中下一个 free inode 的索引

16. 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

17. inode 分配

    a) 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
    b) 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

18. Inode 释放
19. Disk bloc 分配
20. Disk block 释放
21. iNode 与 disk block 的释放比较
22. 系统如何转变一个名字到 inode

    a) 一个叫做 namei 的进程做这个事情