1. 为什么要学习OSD?

> OS forms the foundation of modern computing

1. **Abstractions**: Modern Software 是如何使用 Hardware的?
2. **Resource Management**： 如何去做 resource isolation?
3. 如何在兼顾**实现以上两点的同时**，确保 **High Performace** ?

在这门课上，主要通过两种方式来学习OS

- 1. Conceptual Learning
- 2. Intensive Programming

在笔记的书写中，很多时候并不能很好、贴切的翻译一些英文句子和词汇，所以就直接使用了来自课件、书上的原话。

2. OS 导论

OS, 是一个抽象（abstract）和 管理（manage）硬件资源的 **软件**
从高到低排序： 用户 -> 应用程序 -> 操作系统 -> 硬件

本章将是这门课内容的导论。section 2以后的章节将对对于各个主题，进行更深层次的讨论和总结。

2.1 硬件

OS管理的硬件主要有

1. memory： 基本实体都是DRAM，对应OS中的Memory，使用malloc()作为memory controller, 来分配内存
2. Disk：对应OS中的File System, 通过SATA传输，使用read()/write() 进行控制
3. Nework Adapter: 对应OS中的网络模块， 使用send()/recv() 进行传输

2.2 OS提供了什么?

1. **Software library (abstraction)** between applications and hardware to make the hardware easier to use
    - Simple, uniform view of diverse hardware devices
2. **Mechanisms and policies for resource management**, to provision and isolate hardware across many applications
    - Effective multi-tenant（多租户） and multi-application systems

2.3 Abstrction

2.3.1 现代操作系统通常为哪些资源提供哪些抽象?

> CPU：进程/线程
> 内存：地址空间
> 存储：文件

2.3.2 操作系统提供抽象的好处?

> 允许应用程序重用公共资源
> 让不同的设备看起来相同 （内存、主板、硬盘）
> 提供更高级别或更有用的功能

2.3.3 挑战

1. What are the correct abstractions?
2. How much of the hardware capabilities should be exposed?

2.4 System Calls

- 系统调用允许用户告诉操作系统在硬件上执行什么操作
- 操作系统提供标准软件接口 （API）
- 典型的操作系统会导出几百个系统调用
- 运行程序、访问内存、访问硬件设备.....

2.5 Resource Management

> Want fair and efficient use of hardware across applications

2.5.1 Advantages of OS providing resource management:

1. Protect applications from one another
2. Provide efficient access to resources (cost, time, energy)
3. Provide fair access to resources

2.5.2 Challenges

1. What are the correct **mechanisms**?
2. What are the correct **policies**?

2.6 Virtualization

> Make each application believe it has each hardware resource to itself
> 这门课主要关注：CPU和Memory

2.6.1 Virtualizing CPU

- 系统有大量的虚拟CPU： 将一个物理CPU转换为实际上无穷多数量的CPU。允许很多程序同时执行
- 可以理解成，当我们在命令行同时执行多个c文件的时候，这时候就可以看成是多个程序同时执行，并实际上使用同一个cpu，但是这些c文件，他们认为自己独占cpu。

2.6.2 Virtualizing Memory

- 物理内存其实是一个byte数组
- 一个程序将自身所有的数据结构存放在内存中
- 读取内存 （load）：
    - 指定一个能够访问数据的地址

- 写入内存 （store）：
    - 指定要写入特定地址的数据

虚拟化内存涉及到的一些Mechanism：

1. Virtual-to-Physical Memory Mapping
2. Page-Fault

Demo1： 用c分配内存

```
#include <stdio.h>

int main(int argc, char *argv[1]) {
    int a = 0;
    printf("%d\n", a);//0
    //aa指针指向a的地址
    int *aa = &a;
    //加了*是指针，对应一个数据的地址。那么指针加上指针，就是再次得到值。
    *aa += 1;
    printf("%x\n", aa);//a16738d4
    printf("%d\n", a);//1

    //创建p指针，分配内存。
    int *p = malloc(sizeof(int));
    //复制给该片区域
    *p = 100;
    printf("address of p: %x\n", p); //fbf156b0
    printf("value stored in p: %d\n", *p);//100
    printf("address of p: %x\n", &(*p));//fbf156b0
}
```

每个进程都有自己的私有虚拟内存空间（private virtual memory space），OS则会映射这些address space到物理内存中

- 对一个正在运行的程序的内存引用，不会影响到其他程序的address space。
- 物理内存是一个由OS管理的共享资源

2.7 Concurrency

> Concurrency（并发）： Events are occurring simultaneously and may interact with one another
> OS必须能够处理并发事件

比较简单的处理方式： 直接隔离他们，阻止他们交互，从而达到Hide Concurrency的效果。
但是，这样子指标不治本。因为很多时候有些任务，就是需要进程间交互才能执行的。这时候，才是真正想办法来处理了。
常见做法有：

1. 为进程提供抽象（锁(lock)、信号（semaphores）、条件变量(condition variables)、共享内存(shared memory)、关键部分(critical sections)）
2. 如果用了锁，则需要确保进程不会死锁
3. 让交互线程(interaction threads)必须协调对共享数据(shared data)的访问

多线程Demo：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

volatile int counter = 0;
int loop;
char *name;

void *worker() {
    for (int i = 0; i < loop; i++) {
        counter++;
    }
}

int main(int argc, char *argv[]) {
    loop = atoi(argv[1]);
    pthread_t p1, p2;
    printf("initial value: %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value: %d\n", counter); // 20.
}
```

可以试着执行之下上面代码。试着改变参数大小。可以发现，如果参数比较小（100~1000），那么最终结果就是参数值*2. 但是，当参数变大后，结果就不对了。

```
./mte 10
initial value: 0
Final value: 20

./mte 1000
initial value: 0
Final value: 2000

./mte 10000
initial value: 0
Final value: 14692

./mte 1000000
initial value: 0
Final value: 1007083
```

这是因为并发的情况并没有处理好。p1和p2这两个线程同时执行，并且同时对一个counter进行增加操作。数字小的时候，p1或p2执行的很快，因此并不会出现**p2更新了数字，p1读了旧的数字并将该数字更新到了和p2更新后同样值的情况。**参考数据库的脏读。

有点抽象，举个例子：

```
p2:  读出counter: 10000  然后+ 1 = 10001
p1:  读出counter: 10000  然后+ 1 = 10001;
```

那这样就等于白操作了。

2.8 Persistence

> Persistence: Access information permanently

- 信息的寿命比任何一个线程都要长
- 机器出现未预期的**重启、断电。**这时候就体现出持久化的重要性了

持久化需要：

1. 确保出现unexpected failure的时候，信息能够正确的存储
2. 提供Abstraction使得进程不知道数据是如何存储的
3. 因为磁盘IO很慢，我们需要对持久化进行优化

2.8.1 OS 在持久化过程中做了什么?

1. 搞清楚新的数据在disk的那一块存放
2. 向底层（underlying）存储设备（storage device）发出I/O 请求
3. 文件系统（FileSystem）在IO期间处理崩溃

2.8.2 两种持久化策略：

1. Journaling
    - 日志文件系统： 一种文件系统，在将这些更改提交到主文件系统之前，使用日志记录对文件系统的更改
    - 运行流程：
        - 对文件系统的更改首先记录在日志中
        - 更改成功写入日志后，将其提交到主文件系统
        - 当系统出现崩溃或故障时，可以通过重放日志来恢复文件系统。日志中记录的任何不完整或不一致的操作都可以完成或撤消，以维护文件系统的完整性。
        - 崩溃后恢复速度更快
        - 数据完整性得到保证
    - 例子： ext3、ext4
2. Copy on write
    - 写入时不直接对原位置的数据进行修改，而是写入新的位置
    - 原数据将一直保存，直到写入完成
    - 运行流程：
        1. 当需要读取数据时，首先从当前位置读取数据
        2. 在新位置制作数据的副本
        3. 对副本进行修改
        4. 最后将只想原数据的指针更新指向新数据的副本
    - 优点：
        1. 简化了崩溃后的恢复。因为在修改完成前的原始数据不会受到影响
        2. 提供了任何给定时间点的一致数据快照
    - 例子： 一些现代化的文件系统，如ZFS、Btrfs

3 CPU Virtualization

> 从本章开始，我们正式开始讨论虚拟化的第一个主题： CPU 虚拟化
> 本章会解决以下问题：
>
> 1. What is a process?
> 2. Why is **limited direct execution** a good approach for virtualizing the CPU?
> 3. What execution state must be saved for a process?
> 4. What 3 modes could a process in?

3.1 What is process?

> Process: An **execution stream**（执行流） in the context（上下文） of a **process state**

execution stream
* A stream of executing instructions
* Running piece of code
* Thread of control (其实和execution stream一个意思。单线程的话就是执行流嘛。然后多线程的话，每个线程都有自己的执行流。)

执行流指的是计算机程序正在执行的指令的顺序流。代表着一段指令是活动的。

- 说人话，就是正在跑的程序，这个程序得是活的，而不是在磁盘上的静态文件。在跑的过程中，CPU会直接处理这些指令序列

补充一下： CPU全名**central processing unit**。别学了半天OS，CPU到底是个啥都不知道。。

process state
* Everything that the running code can affect or affected by
* Register
  * Heap, General Purpose, floating point, stack pointer, program counter
* Memory Spaces
* 例子： Open files

- process is not program!
  - program是静态的代码和数据
  - process是动态的代码和数据
  - 可以有多个process运行同一个program

3.2 Process Management Segments

- OS 会为每个进程分配memory。
- 这个memory，包含了很多的块(segment)
  - 从上到下来看
  - 最高层是 stack，用于存储局部变量。也包括command line arguments（位于顶部）， 以及环境变量
  - 再往下走是Heap。用于动态的memory。 在stack和heap之间，有一块unused segment，用于stack/heap的扩容
  - .bss Global Uniniialized Variables（全局未初始化变量）
  - .data Global Initialized Variables（全局已初始化变量）

- 最底层才是code。Read-Only

3.3 Process vs. Thread

很明显，进程肯定不是线程

但是他们还是有相似之处的，我们可以把线程看作一个**轻量版** 的process（Light weight Process LWP）。

Thread is a execution stream that **shares an address space**
一个进程里存在多个线程

举个共享内存空间的例子。
对于进程来说，两个相同程序的不同进程，当他们都在access同一个地址值的时候，他们得到的结果不一样 （内存虚拟化）
而对于线程，一个进程内的多个线程，当他们都在access同一个地址值的时候，他们得到的结果是一样的

对于进程，我们的目标是： **让每个进程都感觉，自己是独占cpu的。**

3.4 Resources Sharing

了解了目标后，我们先看一下OS是怎么分配资源(resources)的

首先， 资源分配通分为两种：

1. time sharing
2. space sharing

> time sharing 主要针对的是**单CPU**的情况。 使多个用户或进程能够共享单个处理器。
> 具体做法是：将可用的处理时间划分为多个小的时间间隔（timeslice），并且将这些时间片分配给各个用户/进程。从而打到各个用户/进程公平分配资源的效果，并且在他们各自分配的时间片内，能够独享cpu

- 关si键词： 公平、快速切换（illusion of multaneous execution）

> space sharing 关注的是**空间**上的共享/复用。每一个进程都被分配CPU容量的一小部分。

- 关键词：并行（Parallel）执行， 同时执行（real simultanous execution）， 资源隔离（resource isolation）

在共享时，我们主要注意两点：

1. cannot perform restricted operation
2. should not run forever or make the entire system slow => performance

###3.5 Provide Good CPU Performance

1. Direct Execution
    - 直接让用户操作硬件： CPU只负责创建并初始化进程，之后的控制权就回到起点（比如main()），交给用户手中了。

这样子用户进程的权限就过大了。因此Direct Execution肯定是有问题的：

1. 进程（Process）可以做一些受限（restricted）的事情
    - 比如读取/写入别的进程的数据
2. 进程可以永久执行（缓慢，有漏洞，恶意（malicious））
    - OS 需要在进程之间交换(swap)的能力

3. OS很多操作很慢： 比如I/O 还是进程切换能力

根据以上问题，我们可以给出一个折中的解决方案: 让OS和Hardware保留一些控制权。

3.5.1 Restricted Ops

Q1: 我们如何确保用户进程不会单方面（unilaterally）执行受限(restricted)的操作呢?

解决方法： （权限级别/分离）privilege levels/separation

- 如果想要直接和设备进行交互， 那就用kernal mode，这样就不受限制了。
- 对于用户进程，就在user mode下执行。如果试图和设备进行直接交互，就会进陷阱（trap），然后software interrupt
- 如果用户进程想和设备进行交互的话，可以通过以下方式：
    1. System Calls (由OS实现的方法)
    2. Change Privilege Level（权限级别）through system call（trap）

System call demo: 1 direct system call & 1 system call provided by libc function.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>

int main(int argc, char *argv[]) {
    long ID1, ID2;
    ID1 = syscall(SYS_getpid);
    printf("direct system call, pid = %ld\n", ID1);//direct system call, pid =
3565265
    ID2 = getpid();
    printf("libc wrapped system call, pid = %ld\n", ID2);//libc wrapped system
call, pid = 3565265
}
```

3.6 System Call Table and Trap Table

> 前面讲到过，系统调用表和陷阱表之间存在映射。用于管理用户引发的系统事件。
> 主要得知道怎么通过汇编，对系统调用表和陷阱表进行分析，从而得出系统时间的结果。

CPU使用EAX寄存器（register）的内容作为源操作数。
窍门就是看$后面的数字内容，找系统调用表和陷阱表上数字对应的操作就行。

比如这道题：

```
Suppose the trap table on a machine looks like the following:

1 - illegal; call OS process kill routine

2- run OS system call routine

Suppose the OS's system call table looks like the following.

1 - sys_read()
```

```
2 - sys_write()

What happens when an application performs the following instructions?

movl $1, %eax

int $1
```

前面的1是system call表里的，后面的int $1是trap表里的。所以结果就是kill runtime.
movl -> 放进eax， int -> 产生软件中断。

总结一下，user processes 是不被允许直接 perform：

1. arbitrary memory access
2. Disk I/O
3. Special x86 instructions like lidt. (Inteterrupted Discription Table)

如果user processes做了上面的事情，那么大概率就是进了trap，然后kill routine。

3.7 How to take CPU away?

> OS 需要实现多任务处理（multitasking）。先记住一个词**上下文切换**。后面会重点围绕这个概念来讲

- Mechanism： To switch between classes
- Policy: To decide which process to run at what time.

Mechaism 和 Policy的区别?

> Policy: Decision-maker to optimize some workload performance metric
>
> - Which Process to run? When to run? => scheduler
> Mechanism: Low-level code that implements the decision
> - How ? => Dispatcher

在这里，我们先将Dispatcher

Dispatch 的逻辑很简单

```
while (1) {
    run process A for some time slice
    stop process A and save its time
    load context of another process
}
```

问题是：
1. dispatcher怎么在一段时间后，重新获得控制?
2. 哪些execution context必须被保存并恢复?

3.7.1 Q1:dispatcher怎么在一段时间后，重新获得控制?

> Option 1 : Cooperative Muti-tasking
> 通过trap，将CPU移交给操作系统的信任线程
>
> - 例如： System Call， page fault（想要换取的page不在main memory里）或者 error
> - yield()
>   但是，这样子的话也有问题。设想一个process，它没有任何越过自身权限的操作（I/O），也没有进任何的Trap，也没有自己叫yield(),那么最终，这个process就会一直占着整个机器。唯一办法只有重启。

这时候就有了第二种选项

> Option 2: Regain control without cooperation
> 我们可以启用周期性的，时钟。进入OS的时候，我们启动时钟，，时钟时间到后，硬件会生成timer interrupt
> 用户也不能屏蔽掉timer interrupt，因为他们没有权限这样子做。

3.7.2 Context save在哪里?

> process control block (PCB) 也可以叫做 process descriptor (PD)
> 每个进程都会有PCB

3.7.2.1 PCB 存储了哪些信息

1. PID
2. Process state (I.e., running, ready, or blocked)
3. Execution state (all registers, PC, stack pointer) -- Context
4. Scheduling priority
5. Accounting information (parent and child processes)
6. Credentials (which resources can be accessed, owner)
7. Pointers to other allocated resources (e.g., open files)

3.7.2.2. Context 保存/切换流程

1. 进程A从user mode 转换到 kernal mode，权限提升。OS决定从A转到B
2. 在kernal stack上保存A的上下文（PC， registers， kernal stack pointer）
3. 将stack pointer指向进程B的kernal stack
4. 从B的kernal stack中恢复上下文

```
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};
```

一些进程在执行不需要Cpu的任务的时候，OS会switch到哪些需要CPU的进程
为了完成这个功能，OS必须关注进程的状态

```
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Running: 占用cpu
Ready： 等待CPU
Blocked：正在等待同步或者I/O.

4. Scheduling

4.1 两种创建Process的方式

1. New process from sratch (从0构建)

- 步骤
  - 从memory中加载指定代码和数据；并创建空的call stack
  - create and initialzie pcb
  - put process on ready list.

好处：定制化，no waste work
坏处： 很难涵盖所有可能的options for setup. 比如WindowesNT有10个参数，这咋搞。

2. 第二个方式： Clone an existing project and change it.
   Fork(): 克隆调用者
   - 停止目前的进程，保存他的状态
   - 复制代码的stack，code ,data, pcb
   - 将PCB放进ready list
     Exec (char *file): exec覆盖调用进程
   - 替换掉目前的代码和数据

demo: Base shell program

```
while (1) {
    Char *cmd = getcmd();
    Int retval = fork();
    If (retval == 0) {
        // This is the child process
        // Setup the child's process environment here
        // E.g., where is standard I/O, how to handle signals?
        exec(cmd);
        // exec does not return if it succeeds
        printf("ERROR: Could not execute %s\n", cmd);
        exit(1);
    } else {
        // This is the parent process; Wait for child to finish
        int pid = retval;
        wait(pid);
    }
}
```

4.2 Dispatcher 和 Scheduling 区别

> Dispatcher是一个low level 的mechanism 而 Scheduling是policy

Scheduling: Policy to determine which process gets CPU when

重点： How to transition? (“mechanism”)
When to transition? (“policy”)

一个用于理解的例子：文件IO Process alternates between CPU and I/O process moves between ready and blocked queues

一些术语
Workload: set of **job** descriptions (arrival time, run_time)
* Job: View as current CPU burst of a process

Metric: measurement of quality of schedule
* Minimize turnaround time
Do not want to wait long for job to complete
Completion_time – arrival_time （process/thread complete - process/thread add to runqueue）
* Minimize response time
* Schedule interactive jobs promptly so users see output quickly
* Initial_schedule_time – arrival_time （process/thread add to runqueue - process/thread scheduled）
* Maximize throughput
* Want many jobs to complete per unit of time
* Maximize resource utilization
* Keep expensive devices busy
Minimize overhead
* Reduce number of context switches
Maximize fairness
* All jobs get same amount of CPU over some time interval

4.3 一些 Scheduler

> 在考虑Workload的情况下，我们需要考虑这些是否达成
>
> 1. Each job runs for the same amount of time
> 2. All jobs **arrive at the same time**
> 3. All jobs only use the CPU (no I/O)
> 4. Run-time of each job is known

4.3.1 FIFO

FIFO: First In, First Out
- also called FCFS (first come first served)
- 根据arrive time来干活
- ABC同时到，先干A，A好了的同时马上干B
- turn_around = completion_time - arrival_time

计算题： 算一下avg_turn_around_time:

显然FIFO是不满足第一条的， 每个Job跑的时间都不一样。从而导致，metric中的turnaround会很高

如果第一个来的job非常的time-consuming，那么就会导致，后面那些原本能够很快就完成的job，被堵着（阻塞）。

这里的平均turnout_time就达到了 (60 + 70 + 80) / 3 = 70s 而原本有的任务只需要10s就可以完成。

根据FIFO的缺点，我们就可以设计出第二种似乎更好的Scheduler。 Shortest Job First（SJF）

4.3.2 SJF

> 选择run_time最小的job

计算1： 对于这个例子，平均turnout time是多少呢

(10 + 20 + 80) / 3 = 36.7

计算2： 对于这个例子，平均turnout time是多少呢

turn_around = completion_time - arrival_time
记住这个公式，就很好算了。

可以看出，FIFO和SJF都是非抢占式（non-preemptive）的。只有当任务执行完毕，或者优先级不够的情况下，才会让出CPU。

抢占式（preemptive）的CPU则相反。就算你的job已经在执行了，万一来了个比你正在执行的job更牛的job，则会schedule更牛的job，你正在运行的job失去了CPU，在旁边等着

4.3.3 STCF

> Shortest time-to-completion First
> 永远执行会完成最快的任务。

计算：平均turnaround time

(80 - 0) + (20 - 10) + (30 - 10) = 80 + 10 + 20 = 110
110/3 = 36.6s
很明显在这种情况下，抢占式的turnout time会比非抢占式的快很多

4.3.3.1 Response Time

> 有时候，从**任务到达**到**任务开始**的这段时间也很重要。我们将这段时间称为：Response Time
> response_time = first_run_time - arrival_time

job b在10s的时候到。
turnaround time = 30 - 10 = 20s
response time = 20 - 10 = 10s

4.3.4 RR

> Round Robin
> 在response time方面比STCF、SJF、FIFO做的都好
> 因为它会每一段时间就会交替状态为Ready的进程，从而每个job第一次开始run的时间不会差距不会特别大。

当前，有得必有失。RR在turnaround time上非常慢。因为job是交替执行的，原本能很快结束的job被迫得隔一段时间就让出CPU。
通常，我们选择RR的原因是因为我们不知道每个job的run time。选择rr的目的和stcf、sjf一样，都是为了让能够最快完成的进程，有机会最早做到。

4.3.5 MLFQ

> Multi Level Feedback Queue

不同类型的job，要求也不一样

- 交互性的（interactive）program，需要更快的response time
- 批处理（batch）program，需要更快的turnaround time。

MLFQ 基于多层的RR（Round Robin）实现。
每层都有更高的优先级，并会抢占低优先级的层数。

4.3.5 决定优先级

两种方式能够用来决定优先级。

1. History

> 使用进程过去的behavior来预测未来的behavior
> 根据此进程过去的 CPU 突发（作业），猜测 CPU 突发（作业）的行为方式

4.3.6 MLFQ Rules

1. If priority(A) > priority(B), A runs
2. If priority(A) == priority(B), A and B runs in RR
3. Process start at the top priority
4. If job uses the full timeslice, then demote process

MLFQ 的小缺陷
低优先级的任务可能永远都不会被scheduled。因此，我们需要每隔一段时间就去将所有的job放到最高优先级的队列中。

4.3.6 Lottery Schedudling

就和它的名字一样。彩票。
目标： fair share

- 只关心能否公平地分享CPU

Fair Scheduler:: Guarantee that each job obtain a certain percentage of CPU time. Not care about response time or turnaroud time
实现逻辑也很简单： 给processes 一张彩票，谁中了谁就run。更高的优先级说明拿到了更多的彩票。

Ticket: 代表了一个process可以占用多少份额的资源。

比如 Process A拿了75张票，那就占用75%的CPU。 Proess B拿了25张票，那就占用25%的CPU
然后Scheduler就会在这100张票里去抽，抽中哪一个数字，就执行手里握着那个数字的线程任务。

lottery 算法的实现

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t *current = head;

while (current) {
    counter += current->tickets;
    if (counter > winner)
        break;
    current = current->next;
}
// current is the winner
```

4.3.7 Stride Scheduling

目标同样是为了防止任何线程monopoly CPU。

The basic idea is assign each process a 'stride', which represents its priority or share of the CPU time.

系统中的每个进程都被分配了一个唯一的步长值。步幅与进程的优先级成反比。优先级较高的进程分配较小的步幅，优先级较低的进程分配较大的步幅。目标是让优先级较高的进程更频繁地访问 CPU。

同样，进程会被放进一个队列。所有进程自身的counter从0开始。每次自己被执行了，就将自身的counter 翻一翻（counter + stride）。 Scheduler会选择具有最小counter的process。

- Stride Scheduling同样也存在问题：
    - 对于新增job，很难确保公平性和优先性。因为每个job，在初始状态下，counter都是0.问题是，当这些线程执行了一段时间后，counter就会变得很大，起码会和0差很多。那么新的job加进来，scheduler必然会判定一直让新job占用cpu，这样就不公平了
    - 就算不考虑新增。如果你想手动更改process的优先级，又该怎么做呢

4.3.8 Complete Fair Scheduling

很显然Stride Scheduling可以确保一定的完整性，但是不能完全确保完全公平

我们的目标是无论线程数量的大小，无论何时添加、更改线程，都能保证其公平地使用资源

Complete Fair Scheduling(CFS) 自从Linux 2.6.23 版本被应用。 O(logN) runtime
原本是MLFQ
Process now ordered by the amount of CPU time they use
取代了队列，转而使用红黑树

- CFS核心概念
    - 使用一个counter记录累计执行时间（cummulative execution time）
    - Schedule process with **least** runtime

下面这些都是gpt对其的介绍

- 虚拟运行时间：
    - CFS为每个可运行的进程维护一个“虚拟运行时间”。虚拟运行时间表示一个进程等待执行的时间相对于其他进程。较小的虚拟运行时间值表示更高的优先级。
- 调度决策：

    - 选择具有最小虚拟运行时间的进程进行执行。这确保了等待时间较长或累积CPU时间较短的进程被优先考虑。
- 时间量子：
    - CFS不使用固定的时间片或时间量子，而是根据可运行进程的数量和它们的虚拟运行时间动态调整时间量子。
- 动态时间量子计算：
    - CFS根据进程的权重计算每个进程的时间量子。权重是分配给每个进程的值，表示它在CPU中的份额。较高的权重导致较大的时间量子。
- 权重和Nice值：
    - 进程根据其优先级被分配权重。用户进程可以使用“nice”值进行优先级调整，该值范围从-20到+19。较低的nice值表示较高的优先级。权重与nice值成反比。
- 平衡机制：
    - CFS采用平衡机制来维持随时间的公平性。它定期检查运行队列，如果检测到不平衡，则重新分配负载。这有助于确保没有进程在CPU时间上被不公平地耗尽。
- 稳态公平性：
    - CFS旨在实现稳态公平性，这意味着在更长的时间内，每个进程都能获得其公平份额的CPU时间，而不考虑短期波动。
- 红黑树数据结构：
    - CFS中的运行队列使用红黑树数据结构实现。这允许根据它们的虚拟运行时间有效地插入和删除进程。

5. Muti-core Scheduling

> 由于多核处理器的兴起，多核调度成为必须。因为单单添加CPU的数量不会让一个应用程序执行地更快。我们需要重写应用程序以保证其能**并行**执行

5.1 单CPU + 缓存 + main memory

CPU分出一篇区域给缓存，存放popular data found in main memory. 速度块，容量小
Main memory存放所有数据，从main memory获取数据的速度会比cache慢很多

5.2 多CPU情况

简单来说，每个CPU都有一个缓存区域。那两个CPU加起来就有两块缓存区域了。

同步不同缓存区域的内存看起来很简单： CPU0 将memory中的 数据放进了自己的缓存区，CPU1会读取CPU0的缓存区，同步数据。

但是，有一种特殊情况：
1. CPU 0 读内存放进缓存，CPU1 读到了共享。
2. CPU 0 更新了缓存中的数据 同时 CPU 1 被schedule了
3. 此时 CPU 1 中保存的还是过去的数据。

数据就不一致了

解决方法很简单： Bus Snooping
* 每个cache都会通过观察bus来注意到memory的更新
* 当CPU注意到自身memory中的数据更新了，就会注意到这个变化

5.3 Cache Affinity

> 无论是多CPU还是单CPU，Scheduler都会尝试将一个进程放在同一个CPU上执行。因为当CPU在执行该线程时，会往缓存里加很多关于改进程的状态。那么等到改进程下次启动时，就会更快一点，因为缓存里已经有信息了

实现方式：将所有需要scheduled的任务全部放在一个队列里面。每个CPU就从这个**Globally Shared**的队列里面拿job。

坏处：

1. 锁
2. 扩展性缺少
3. Cache Affinity
4. 实现起来复杂

5.4 Multi-queue Multiprocessor Scheduling (MQMS)

- Contains Multiple Scheduling queues
  - 每条队列都有自己的Scheduling Discipline
  - 当job进入系统的时候，只会放在一条队列上， 从而避免了信息共享（information sharing）和 同步（synchronization）的问题

MQMS with Round Robin

MQMS的问题： 需要通过跨内核迁移进程来平衡跨内核的负载

6. Virtualizing Memory

目标：

- Transparency
  - Processes are not aware that memory is shared
  - Works regardless of number and/or location of processes
- Protection
  - Cannot corrupt OS or other processes
- Privacy
  - Cannot read data of other processes
- Efficiency
  - Do not waste memory resources (minimize fragmentation（碎片化）)
- Sharing
  - Cooperating processes can share portions of address space

6.1 Abstraction: Address Space

> Address space: Each process has set of addresses that map to bytes

问题是：OS是如何让每个Process觉得，自己有专用的地址空间的?

** 回顾 Addresss Space 中都有什么？ **

1. 静态：

- Code
- Global Variables

2. 动态：
   - Stack
   - Heap

6.1.1 为什么进程需要动态地分配资源?

> 1. 不知道编译时需要的内存量. 静态分配内存时必须悲观, 为最坏的情况分配足够的资源;不高效地使用存储
> 2. 对于那些递归的步骤，不知道会嵌套多少次
> 3. 进程中会有复杂的数据结构，需要我们手动分配资源。

6.1.2 Stack用在哪里?

> 操作系统将Stack用于过程调用帧(procedure call frames), 存放local varibles and parameters
> 局部变量例子

```
#include <stdio.h>

void foo(int z);

int main(int argc, char *argv[]) {
    int A = 0;
    foo(A);
    printf("A: %d\n", A); //A: 0
}

void foo(int z) {
    int A = 2;
    z = 5;
    printf("A: %d, Z: %d\n", A, z); //A: 2, Z: 5
}
```

6.1.3 Heap用在哪里?

> 任何位置的malloc(), new() 都会跑到heap里
>
> - Heap memory consists of allocated area and free area
> - Order of Allocation and free is unpredictable.

pro：所有数据结构都是这样用的
cons:
* Allocation can be slow
* End up with small chunks of free spaces -- Fragmentation(碎片化)

6.1.4 OS在managing heap中的作用

> OS gives big chunk of free memory to process
> OS provides library manages individual allocations

代码中各个数据结构在address space中的分布

6.2 如何虚拟化Memory

> 问题：如何同时运行多个进程

我们知道进程中的addressess是硬编码进进程的binary里的。然而，因为进程已经产生了自己独享memory的错觉，当多个进程同时运行时，很有可能出现**多个进程试图操作同一内存区域**的问题。

解决这些问题的方法有：

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

6.3 Time Sharing of Memory

> 属于比较少见的做法，甚至概念也很少提及

OS通过在进程不在运行时，将CPU registers放入memory，给了进程一种有很多虚拟cpu的幻觉
同样，我们也可以在进程不在运行的时候，把内存放进磁盘。这样进程就会感觉独占memory。

但是，我们知道Disk I/O是一个非常费时、低效的过程。所以这个法子性能非常差

6.4 Space Sharing - Statis Relocation

> 比Time Sharing会好一点

OS在进程加载进内存之前，重写每个程序，这样子不同的进程就会使用不同的地址和指针了。

但是没有保护，可能会出现地址出问题的情况，进而：
1.破坏OS
2.破坏其他进程
3.并且会产生虚有的IO
4.没有隐私
5.一旦分配地址之后，就不能移动地址了。从而不能够allocate new proceess

6.5 Space Sharing - Dynamic Relocation

> Goal： Protect processes from one another

Require hardware support
  * Memory Management Unit (MMU)

MMU会在每个内存引用处动态地改变进程的地址
  * 进程生成逻辑（虚拟）地址，交给MMU
  * MMu 将其转换成物理（真实）地址，存入内存

MMU两种运行模式：
  * Priviledged:
    1.有OS运行
    2.可以操作MMU中内容和

3.一般是通过trap、system call来运行

```
* User mode:
    * 主要是为了translate 虚拟地址到物理地址
```

MMU为了转移，有base register。
base: start location for address space

可以注意到user mode 里面，往logical address上面加了base。
个人理解： 是为了限制user，只能访问base上面的地址。

6.5.1 Dynamic Relocation with Base Register

Idea: 在虚拟内存转换成物理内存的过程中，设一个偏移量。
将这个偏移量存在base register上。
每个进程的偏移量都不一样。

6.5.2 Base Register + Bound

> 单凭Base Register, 如果一个线程不停的往上/下 加减他的memory address，那么总有一天，他会和别的进程的memory space发生碰撞。这样一来，protection又没了

- Bound register： size of this process's virtual address space
    - 有时候会是 base + size （一个进程最大的物理地址）

OS will kill the process if process loads/stores beyond bounds

Tips:
Register中包含
1. base 32bit
2. bound 32bit
3. mode 1bit

判定流程

Interrupt示例

6.5.3 Base + Bound时对进程的管理

> 谈到进程管理，主要涉及到的时上下文切换（context-switch）
> 当执行上下文切换的时候，我们要将Base & Bound Register 添加到 Process Control Block中的context中

- 步骤
    - Change Privileged mode
    - save base & bound registers of old process
    - load base & bound registers of new process
    - Change to user mode to jumo to new process

从上面的步骤，我们可以看出，user process不能操作base & bound registers，并且也不能切换到Privileged mode

6.5.4 Base + Bound + Dynamic Relocation 的好处：

1. Provides protection (both read and write) across address spaces
2. Supports dynamic relocation
    - Can place addresses places initally at locations different from assumed in the memory
    - Can move addresses spaces if needed
3. Simple, inexpensive implementation
4. Fast
    - add and compare in parallel

6.5.5 Base + Bound + Dynamic Relocation 的坏处：

```
* 每个进程都必须连续地分配在物理内存中
* 必须得分配一些不能被process用到的内存。
* No Paritial Sharing.(Implication of isolation)
```

6.6 Segmentation Addressing（分段寻址）

> Base and Bound 的加强版
> 将地址空间分割成若干个逻辑块， 每个逻辑块对应地址空间中的逻辑实体

- code， heap， stack
  特性：
    1. 每个实体都可以placed seperately on the physical address
    2. 每个实体都可以扩展和伸缩
    3. 每个实体都受到了保护（seperate read/write/exec bits）

> 在操作系统（OS）中，分段寻址（Segment Addressing）是一种内存管理的方法，其中内存被划分为多个不同大小的段（segments），每个段用于存储特定类型的数据或执行特定的任务。每个段都有一个唯一的标识符，称为段描述符，它包含有关段的信息，例如起始地址、段的大小和访问权限等。
>
> 以下是关于操作系统中分段寻址的解释：
>
> 段描述符：
> 在分段寻址中，每个段都由一个段描述符表示。段描述符存储了与段相关的信息，如起始地址、段的大小、访问权限、以及其他控制信息。这些描述符通常存储在特殊的表中，例如全局描述符表（Global Descriptor Table，GDT）或局部描述符表（Local Descriptor Table，LDT）。
>
> 逻辑地址：
> 逻辑地址由两个部分组成：段选择器和偏移量。段选择器用于选择段描述符，而偏移量表示从选定段的起始地址开始的位置。通过组合这两部分，可以构成完整的逻辑地址。
>
> 段寻址过程：
> 当程序引用一个逻辑地址时，操作系统通过分段寻址来确定对应的物理地址。首先，根据逻辑地址中的段选择器，操作系统找到相应的段描述符。然后，使用段描述符中的起始地址和逻辑地址中的偏移量来计算物理地址。

> 优势：
> 分段寻址的主要优势之一是可以更灵活地管理内存，因为不同的段可以有不同的大小和访问权限。这有助于更好地组织和保护内存中的数据和代码。
>
> 保护机制：
>
> 通过分段寻址，可以实现内存保护机制。每个段描述符都包含有关访问权限的信息，如读、写、执行权限。这样，操作系统可以确保程序只能访问其具有权限的段。
>
> 虚拟内存：
> 分段寻址也有助于实现虚拟内存。不同的段可以映射到物理内存的不同区域，而不同的程序可以共享相同的段，使得虚拟内存的管理更加灵活。
>
> 需要注意的是，现代操作系统通常采用更先进的内存管理机制，如分页寻址（Paging），而不仅仅是分段寻址。这些机制可以更好地支持虚拟内存、内存共享和更高级的内存保护。

Segmentation Addressing的缺点

- Fragmentation，太多scattered的segment

6.7 Paging

> 前面说过，分段寻址的问题在于fragmentation太多。我们的目标就是**降低对于连续空间的要求**

- reduce external fragmentation
- grow segment as needed

思路：我们可以将address space和physical memory分成固定大小的page

- Size 2^n， e.g. 4kb 8kb 16kb etc..
- Physical Page: page frame

一些计算题公式

1. 已知number of bits for vpn（virtual page number）， 问多少virtual page?

> 2^number of bits for vpn

2. 已知bits in virtual address and bits for offset, 问virtual page number有多少bits?

> bits in virtual address - bits for offset

3. 已知Page Size， 问offset需要多少bits?

> log2(page_size)