操作系统期末复习

Yilun Zhu

2025 年 6 月 18 日

前言

本复习资料为针对东南大学操作系统课程的复习资料，仅用于个人在期末考试中的整理复习，几乎涵盖了董恺老师上课提到的所有内容，并加以自己的理解进行的整理。

希望本复习资料能够帮助还未开始“预习”的大家在期末考试的一两天时间内掌握 OS 的基本内容，祝每一位看到这份资料的小伙伴们都能在期末中取得好成绩！

修订日志：

- 2025 年 6 月 15 日：由于时间原因，主播来不及整理 Chapter 2、Chapter 8、和 Chapter 13 的内容了，有机会的话后续会添加。

2025 年 6 月 18 日

考试注意事项

1. 答题的时候，遇到要罗列的重要知识点，可以先用中文把关键字写出来。
2. 考试一定要带尺，可能要画表格。
3. 考试范围是课本（Operating System Concepts）和 PPT 的交集，OSTEP 的一些内容也是不考的。
4. 每周三上午去办公室找 dk 是能够答疑的。
5. 平时 20 分、实验 20 分，期末 60 分；但是前面的 40 分不会大部分学生都给满（但是基本上都会给满，也就是 40 和 39.5 的区别）
6. 期末考试千万不能乱考。
7. 操作系统这个课很抽象，概念很多。
8. 判断题 20 个，每个 1 分，这个部分会很坑，很多人会错将近一半。董恺认为讲的绝对的一般都是错的，讲的不绝对的一般也都是对的。
9. 简答题 30 分，6 个 5 分题，没有题库，那个非常大（这里出现的内容会相对重要一些，是有机会满分的，不可能有概念完全没听过）
10. 注意，简答题考的内容可能不会纯概念，会考一些问答题本来应该出的题目。（mutex 等）
11. 最后的问答题的题型一定都见过，但是考的不会这么简单。
12. 第一题是 Process/Thread API，简单来说会问一共创建了多少个进程，考的复杂一点就会把进程和线程搞在一块考。
13. 第二题是 CPU Scheduling，历史上大家都是满分的。
14. 第三题是 Deadlock Avoidance，类似银行家算法。
15. 第四题可能有变化，一定和 page 相关，可能考 paging（比较复杂，董恺喜欢考，多级页表），也可能考 replacement（相对简单），不过董恺不太会出卷。
16. 第五题是 Semaphore，也不用完全把信号量小书都刷掉，确保 initialization 写对，后面写点东西，拿个 7，8 分什么的没那么困难。

目录

第一章　Introduction

1　In-class Contents

1.1　What Operating Systems Do

1.1.1　What is an Operating System

定义 1.1. *Program*：是计算机硬件和用户中间的一个软件。

Opearing system 的目标：

（1） 运行用户程序并且让解决用户问题变得更加简单

（2） 让计算机系统变得容易使用

（3） 用一个有效的方法去使用计算机硬件

所以说，我们也可以用 easy,convenience 和 efficiency 来评价一个 OS 的优劣。

1.1.2　Computer System Structure

计算机系统可以被分为以下四个组成部分：

（1） 硬件：CPU、内存、I/O

（2） OS：作为一个中间媒介来控制和管理硬件的使用，来实现软件的功能

（3） 应用程序：决定解决用户计算问题的方法（compiler 解决代码问题、web browsers 解决上网问题）

（4） Users：用户、机器和其他电脑

现在的计算机系统架构一般都支持 multi-user 和 multi-task。

现在，我们将从两个角度来看操作系统，一个是用户的角度，一个是系统的角度。

1.1.3 User View

用户想要的是 easy,convenience 和 good performance（efficiency），并不在意资源的利用率。

下面这段内容知道了解也就可以了。

但实际上吧，资源的分配会很大程度的影响用户的体验，比如说共享的电脑需要分配好资源让所有用户都高兴；专用系统的用户（如工作站）通常有独立的资源，但也会使用服务器的共享资源；手持设备资源有限，操作系统优化重点在于可用性和电池寿命；有些计算机几乎没有用户界面，比如嵌入式系统。

1.1.4 System View

注意以下的内容 very important。

从操作系统的角度来看：

考点 1. *OS 是一个资源分配器，它分配所有的资源，在高效和公平使用资源的相互冲突的请求之间做出决定。与此同时，OS 是一个控制程序，控制程序的执行，以防止错误和不当使用计算机。*

英文版本：

OS is a resource allocator, it manages all resources and decides between conflicting requests for efficient and fair resource use. Besides, OS is a control program, it controls execution of programs to prevent errors and improper use of the computer.

最后呢，操作系统没有明确的被广泛接受的定义，Microsoft 给出的定义是“当您订购操作系统时，供应商提供的所有产品”，但这是不对的。还有一种定义是“计算机上始终运行的一个程序”，这也是内核的定义，表明了其他一切东西都是系统程序（OS 自带）或者应用程序。

1.1.5 Computer-System Organization

定义 1.2. *Bootstrap program（引导程序）：开机或启动的时候被加载，一般存在 ROM 或者 EPROM 中，被称为固件，它可以初始化系统的所有东西，并且加在 OS Kernel 并开始执行。*

加载 OS kernel 的步骤是，首先通过运行 Bootstrap program，来加载、启动存在硬盘或者外存的 Bootstrap loader，再将 OS kernel 加载到内存当中，最后把控制权交给 OS kernel。

1.1.6 Von Neumann Model

冯诺依曼架构其实就是说了两件事情：

（1） 一个计算机系统是如何组成的：一个或多个 CPU、设备控制器通过公共总线连接，提供对共享内存的访问。

（2） 一个计算机系统是如何运行的：CPU 和设备的并发执行竞争内存周期。

1.1.7 Computer-System Operation

注意. *I/O 和 CPU 是可以并发执行的，什么意思呢，就是 I/O 和 CPU 可以同时进行不同的任务，如 A 任务在用 CPU，这个时候 B 任务就可以 I/O。*

那么 I/O 和 CPU 是怎么做到并发执行的呢？

（1） 每个设备控制器负责特定的设备类型。

（2） 每个设备控制器都有一个本地缓冲区。

（3） CPU 将数据从主存储器移动到本地缓冲区。

（4） I/O 从设备到控制器的本地缓冲区。

诶，你看是不是大家都不冲突呀（狗头）

好，那现在 CPU 和 I/O 可以实现数据沟通了，但是如何合作呢？

Device controller 通过 **interrupt** 告诉 CPU 已经完成操作了，你可以回去干活了。

例 1. *假设你让洗衣机洗衣服（这个动作就像 CPU 把任务交给设备控制器）。你不会一直站在旁边盯着洗衣机（这叫轮询）。相反，洗衣机会在洗完后响铃通知你（这就是中断）。你听到响铃（中断信号），就知道可以去处理下一步工作了。*

定义 1.3. *Interrupt：中断指的是处理器的输入信号，表示需要立即关注的事件，然后由 OS 来处理 interrupt。*

那么当一个中断发生的时候，通常会把 CPU 的控制权交给 interrupt service routine，也就是中断服务，通过查找中断向量表（也就是存储每个中端服务地址的表），让 CPU 去处理一下这个东西。

同时，中断机制必须存储被中断的指令的地址，就像打游戏存了一个副本，之后要找到我存的副本的位置。

注意. *Service rountines 都存在 memory 当中，同时中断服务和被 interrupt 的指令的地址都会被记录。*

另外，还有两种特殊的 interrupt，分别是 trap 和 exception，他们是由软件产生的，一般由错误（除以 0）或者是用户需求导致。

考点 2. *An operating system is interrupt driven.*（*interrupt 一般由 external devices 引起，注意 CPU 和 memory 不是 external devices，但是 GPU 是；当然 interrupt 也可以是软件引起的*）

怎么去理解这句话呢？其实也很简单，我们举一个例子：

例 2. *如果我们什么都不做，OS 会做什么呢？*

答：*Just wait and do nothing.*

那我们会怎么解决这些 Interrupts 呢？

首先 OS 会保存寄存器和 PC 的状态来记录中断时 CPU 的状态，然后通过轮询（效率低）或者中断向量系统（就是中断向量表）来处理 traps 和 exceptions，同时对于每种 interrupt 都会有一个单独的代码段来决定要采取什么行动。

以下内容知道就行了。

注意. *中断也是有 priority 的，低 priority 的要给高 priority 的让路；如果两个都是 high priority，要借助 scause。*

1.1.8 Storage Structure

（1） Main memory：是 CPU 唯一可以直接访问的较大的存储媒介，random access（不用像 tape 那样 traverse），但一般是易失的。

（2） Secondary storage（辅存，例如 disk）：是 main memory 的外延，并且提供较大的非易失的存储空间。

Hard disks：上面是磁道和扇区，由 disk controller 管理 device 和 computer 之间的联系。

Solid-state disks（固态硬盘）：比 hard disks 要快，而且是非易失的，也越来越受欢迎（虽然考试不考）。

1.1.9 I/O Structure

如果我们没有 interrupts 的话，当要做一个 I/O 的时候，CPU 必须要等 I/O 完成，这样的效率是很低的，因为 CPU 需要不断占用内存来查看 I/O 是不是完成了。而且同时只能处理一个 I/O 请求。

所以我们更多采取 Interrupt Driven 的 I/O structure。

啥意思呢？说白了就是、当用户程序发起I/O操作后，操作系统会将具体的I/O请求交由设备控制器执行，并将CPU控制权还给用户程序，使其可以继续执行其他任务而无需等待I/O完成。这种机制提高了资源利用率。当I/O操作完成时，设备控制器会通过中断通知CPU，此时控制权转交给操作系统的中断处理程序。操作系统处理完I/O相关事务后，可能恢复原来的用户程序，也可能调度其他就绪进程。总体而言，用户程序在正常执行时持有CPU控制权，而在I/O完成、系统调用或异常等情况下，操作系统接管控制权以执行必要的管理任务。

定义 1.4. *Device Driver*：一个软件，为 *device controller* 和 *kernel* 提供了一个独有的接口。

那么我们如何告诉OS说我要执行I/O了呢（读文件、写数据、打印）？
通过System call。

例 3. 当用户程序需要进行读文件操作时，会调用系统调用 *read()* 向操作系统发起 *I/O* 请求。操作系统接收到请求后，会检查设备状态，确保当前可以执行 *I/O* 操作。随后，设备驱动程序（*Device Driver*）会向对应的设备控制器（*Device Controller*）发送指令，启动读操作。在 *I/O* 操作进行过程中，*CPU* 的控制权通常会交还给用户程序或调度其他进程继续执行，以提高效率。当 *I/O* 操作完成后，设备控制器会通过中断机制通知操作系统。*CPU* 响应中断，转入内核态，由操作系统中的中断处理程序（通常由设备驱动提供）完成后续处理，如读取数据、更新状态或唤醒阻塞的进程。

注意. *system call* 和 *procedure call* 是不一样的，*syscall* 比如 *read()*，*write()*，而 *procedure call* 是函数调用，比如说 *def rgl()*。

我们前面说到OS要查看设备的状态，这个东西被称为Device-status table，其中包含设备类型、设备地址和当前的状态。同时看OS找到I/O状态表的索引去决定device的状态，也可以加入中断标志。

注意. 反正就注意一个事情，返回控制权的是 *Device Driver* 不是 *Device Controller*。

定义 1.5. *DMA(Direct Memory Access)*：核心思想就是 *disk* 和 *memory* 通过 *Device Controller* 直接交换数据，这个过程当中一般用网卡，不用 *CPU*。

在这个过程中，传输每个数据块才产生一个中断，不会每个字节都中断一下。
目的：提高数据传输率，加速CPU计算速度，保证设备与CPU之间的高效交互。

1.2 Computer-System Architecture

在过去，许多系统用的还是单通用处理器，当然也会有特殊目的的处理器，但现在流行用多处理器，被称为并行系统或者是紧密耦合系统。

这样做有什么好处呢？（有点类似 RAID）

（1） 增加吞吐量

（2） 规模经济：处理器共享资源-> 降低硬件成本

（3） 更加可靠：一个处理器故障，其他的好的就行

有两种类型的 multiprocessing：

（1） Asymmetric Multiprocessing（非对称）：由一个主处理器来进行调度和任务分配，而其他处理器都执行**特定的**任务。

（2） Symmetric Multiprocessing（对称，more popular）：每个 processor 做的事情都是一样的。

现在我们关注另一个话题，为啥那么 multicore（一个物理芯片多个核心）要比 multichip 好？

因为在一个芯片上的 communication 会更快，用更少的能量。

定义 1.6. *Blade Servers*（刀片服务器）：将多个计算单元集成到一个机箱上面，每个刀片就相当于一个独立的服务器，有自己的 *CPU*、内存和存储资源，同时多个 *server* 共享电源和散热系统，来减少能源消耗。

1.3 Operating-System Structure

这块是重点了兄弟，好好复习啊！

定义 1.7. *Batch system*（批处理系统）：处理大量的数据，执行过程完全自动化，不需要人工干预，也就是说，只要用户提交任务，系统就能按照预定的顺序去执行作业。

1.3.1 Multiprogramming

中文名：多道程序设计

核心思想：让 CPU 和 I/O 都忙起来

啥意思呢？就是在内存当中放了一系列的 jobs，CPU 在执行一个作业的时候，其他作业可以进行 I/O 操作。那我怎么选取一个 job 呢？这就和 CPU Scheduling 有关了，具体看第五章，这里我们用的是 **long term scheduling**。

1.3.2 Timesharing

别名：Multitasking

中文名：分时系统

核心思想：OS 虽然只能同时服务一个 user，完成一个 job，但是 OS 可以在 jobs 之间自由来去，让每个 program 都感觉自己独占了 CPU，也就实现了**交互式计算**。

那么怎么实现分时系统呢？

（1） 第一个指令的响应时间要在 1s 以内

（2） 每个用户至少有一个 program 在内存中运行，那么在运行的 program 就被称为 process 啦（打开 word 和运行 Edge 就是两个 process）

（3） 如果有许多 jobs 想要一起运行->CPU scheduling，这个是由 **short term schduling** 来决定的（可以让更快的先做或者让更紧急的先做，比如说游戏优先级高，所以用更多的 CPU 时间片，而后台下载任务的优先级比较低，所以占用的 CPU 资源应该更少）。

（4） 如果进程放不进内存来，那就要靠 swapping 了

（5） 虚拟内存是操作系统的一项重要技术，它使得程序可以运行，即使它们并没有完全被加载到物理内存中。操作系统通过将部分程序代码或数据保存在硬盘上，并使用内存管理技术（如分页），从而允许程序超出物理内存限制运行。

辨析. 比较一下 *Long-term scheduling* 和 *Short-term scheduling*：

长时间调度负责决定哪些作业可以进入内存并准备执行；短时间调度负责决定 *CPU* 将执行哪个作业。

辨析. 再比较一下 *multiprogramming* 和 *timesharing*：

多道程序设计（*Multiprogramming*）更关注提高 *CPU* 的利用率，通过加载多个作业到内存中，在作业等待 *I/O* 操作时切换任务，确保 *CPU* 始终有任务执行。

分时系统（*Timesharing*）则更关注用户交互和响应时间，通过非常快速的任务切换，使得多个用户或任务可以并发执行，并且每个任务能够得到快速反馈。

1.4 Operating-System Operations

定义 1.8. *Interrupt Driven*：分为 *hardware interrupt*（比如键盘输入、打印机完成任务，这种情况是不用等程序结束就能中断的）；还有一种是 *software interrupt(exception or trap)*，比如说除以 *0*，或者是请求 *OS* 服务，还包括无限循环等问题。

1.4.1 Dual Mode

定义 1.9. *Dual-mode*（双模式）：指的是 *User mode* 和 *kernel mode* 双模式，用来让 *OS* 保护自己和其他系统组件。

拓展. 其实现代的 *OS* 是有四个模式的，除了以上两个还有 *VMM(virtual machine manager)*，用来管理不同的 *OS*（虚拟机），还有一个 *monitor*（用作 *security,* 比如人脸识别）

与此同时，硬件还提供了一个 mode bit，来表示现在是 user mode(0) 还是 kernel mode(1)。

那“保护”是如何体现的呢？一些指令是特权指令，是只能在 kernel mode 才能运行的指令，比如说 set the time。

那如果我想要调用一些在 kernel mode 下才能使用的特权指令呢？啊哈，这个时候就要用到 **System call** 啦！它能够把模式切到 kernel mode，并且在访问内核结束之后把 mode 切回去。

1.4.2 Timer

定义 1.10. *Timer*（定时器）：防止无限循环或进程占用太多的资源。

核心思想：定时器保持一个由物理时钟递减的计数器。每当计时器减到零时，触发中断。

操作系统会通过设置这个计数器来控制中断的时机，这通常是一个需要特殊权限的操作（即特权指令）。

1.5 Operating-System Functions

在学 OS 的时候要知道一些关键的 idea：

（1） Virtualization：CPU 虚拟化（让每个 program 都觉得自己是在独占 CPU 而不是共享 CPU）、虚存（让 program 觉得自己有足够的内存）

（2） Concurrency（并发性）：同时解决一些问题来加速计算，但需要注意可能出现的问题（也就是同步问题等）

（3） Persistence：保证数据不会丢失

2 Exercise

Consider a multiprogramming environment, two programs A and B share the system simultaneously, and run as follows.

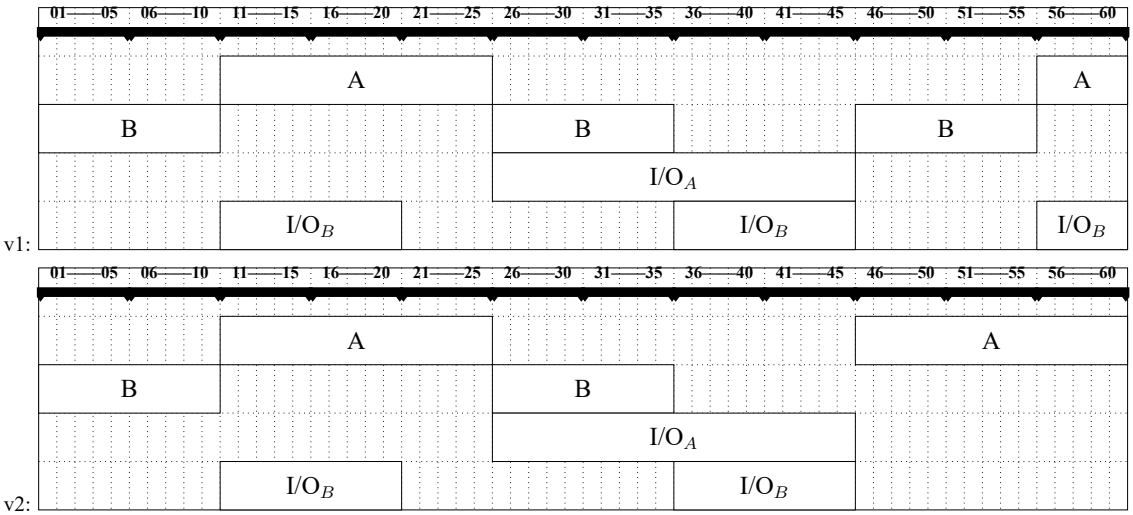
- For every 35 min, A runs on CPU for 15 min, then waits for I/O for 20 min.
- For every 20 min, B runs on CPU for 10 min, then waits for I/O for 10 min.

（1） Suppose B runs first, I/O_A and I/O_B are different devices, and the time of switching between A and B can be ignored. Draw the timeline for these two programs, and calculate CPU utilization within 60 minutes.

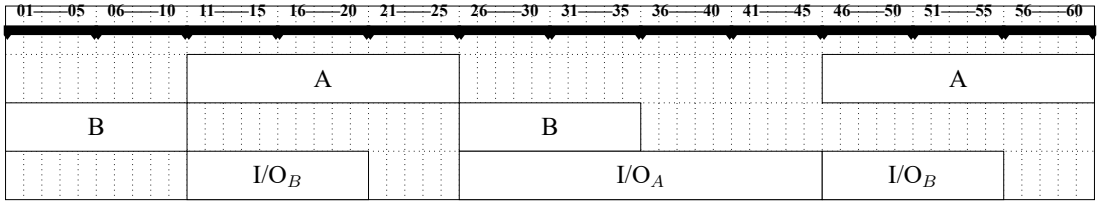
（2） What if I/O_A and I/O_B are the same device?

Keys：

Key to Q1: $CPU\ utilization = 50/60 = 83.33\%$



Key to Q2: $CPU\ utilization = 50/60 = 83.33\%$



3 Concepts Organization

1. What is the purpose of interrupts?

   My answer:The purpose of interrupts is to allow the CPU to respond to external events (such as I/O operations) while processing the current task, and to quickly handle these events, thereby improving system efficiency and supporting multi-tasking.

2. How does an interrupt differ from a trap?

   My answer:An interrupt is typically generated by hardware to signal an external event (such as I/O or a timer), while a trap is usually generated by software to handle exceptions, errors, or system calls.

3. **Can traps be generated intentionally by a user program? If so, for what purpose?**

   My answer:Yes, traps can be intentionally generated by a user program, typically to make a system call. This allows the program to request the operating system to perform privileged operations, such as file management, memory allocation, or process control.

4. Interrupt:generated by hardware, to enable CPU to respond to externel requests, like I/O.

5. Trap/Exception:generated by software, to do some system calls.

6. Multiprogramming:It allows multiple programs to run in memory and keeps the CPU and I/O devices busy, improving overall system efficiency by switching between tasks when one is waiting for I/O.

7. Multitasking(Time-sharing):The CPU time is divided between multiple tasks, giving each task a small time slice to ensure responsiveness, particularly in interactive systems.

8. Interrupt driven:Interrupt-driven I/O allows devices to notify the CPU when their I/O operations are complete through an interrupt, rather than the CPU continuously polling the device to check its status. This reduces unnecessary CPU usage and improves overall system efficiency.

9. Dual-mode:Dual-mode operation refers to the two modes of CPU operation: user mode and kernel mode. Programs typically execute in user mode, but when they need to perform privileged actions (e.g., accessing hardware or system resources), they must make system calls to transition to kernel mode. The purpose of dual-mode operation is to protect the system's resources and ensure security by preventing user programs from directly accessing or modifying critical system resources.

10. Privileged instructions: 太简单了，就是 kernel mode 有但是 user mode 没有的指令，防止系统资源被破坏或修改。

第二章　Operating_System

1　In-class Contents

1.1　System Calls

定义 1.1. *System Call*：用户程序和操作系统之间的接口。

定义 1.2. *Application Programming Interface(API)*：*API* 封装了一系列 *syscall*，只需要调用 *API* 就行，不用直接调用 *syscall*。

用 API 不直接用 system call 的一个重要原因就是，每个 OS 的底层 syscall 可能不同，但是它们的 API 可以是一样的。

那么怎么来执行一个 system call 呢？

Trap, trap-handler and return-from-trap.

那么在一个 trap 当中，我们运行 OS 的哪块代码呢？

Trap table 对应的 trap-handler.

trap 之前的信息会存在每个 process 自己的内核栈当中。

1.2　Policy & Mechanisum

我们分离了 policy（决定我们要做什么）和 mechanisum（我们应该如何做这件事情）。以 timer 为例，中断是我们的目标，但是我们可以通过 RR、SJF 等 mechanisum 来实现这个 policy。

2 Concepts Organization

1. System call: Programming interface to the services provided by the OS.
2. Policy and mechanisum: Mechanisums determine how to do something, policies decide what will be done.
3. Simple/ layered/ microkernel/ modules/ hybrid OS structures

第三章　Processes

1　In-class Contents

1.1　Warm-up

我们来看一个例子，来看看四个不同的进程是如何在一个 CPU 上运行的：

```
/* cpu.c */
int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                spin(1);
                printf("%s\n",str);
        }
        return 0;
}
```

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
```

```
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
```

```
C
A
B
D
C
A
C
B
D
...
```

我们可以发现输出的 ABCD 是没有顺序的，随便输出的。

我们来讲讲这个章节主要关注了哪些内容：

首先，如何实现 CPU 的虚拟化？OS 会通过一些低层硬件支持和软件机制（比如中断、特权指令）和一些高层策略（如调度策略）来实现 CPU 的虚拟化。

我们再来说一下 time-sharing 机制（上下文切换）+ 调度策略（优先级调度）：在这个过程当中，一个 process 正在运行，它会被停掉并且去跑另一个 process，以此类推。

注意. 进程之间是相互独立的，*OS* 的作用是让他们互不干扰。

最后，process(job) 是 OS 提供给用户的最基本的 abstraction 之一，process 和 job 在课本里是一样的。

1.2 Process Concept

定义 1.1. *Process*：一个进程就是在执行的程序，并且进程的执行是按顺序执行的。

当 program 执行后，它就会变成进程，进程里面包含代码、REG、内存等。

那么，进程到底由哪些东西组成呢？

（1） 程序代码，即 text section（文本段）

（2） 当前的运行状态，包括 program counter, instruction pointers, processor registers（存放变量值和临时数据）

（3） Stack：包含临时数据，例如函数参数、返回的地址、局部变量

（4） Data section: 存放全局变量

（5） Heap：动态分配的内存

辨析. 对于 *code* 来说，只有编译之后才会称为 *program*，*program* 是 *disk* 中的可执行文件，而只有 *program* 被加载到 *memory* 之后，*program* 才会变成 *process*。

考点 3. *Program* 只有 *code*、*data section*，而 *process* 还有 *stack* 和 *heap*。

注意. 一个 *program* 可以是多个 *processes*（例如服务器）

1.2.1 Loading: From Program To Process

那么，program 是怎么从 disk 加载到 memory 的呢？

（1） Loading：分为主动加载 (eagerly)（一次性把所有的代码和数据加载到内存当中）和懒加载 (lazily)（只加载需要的部分，剩下的按需加载）

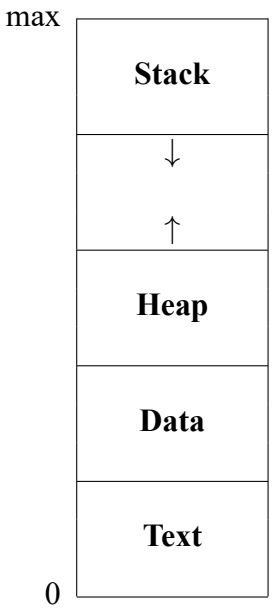
（2） Paging and swapping：将 program 分成一个个有固定 size 的 page，同时这些 page 可以通过 swap 进行换入换出。

（3） program 加载到内存之后，需要为它分配 stack 和 heap。

（4） IO Initialization（输入输出初始化）: 初始化进程的 I/O，例如打开标准输入输出、文件描述符、设备资源等.

现在我们来看看 program 加载到 memory 后变成 process 了是什么情况的，我们看下面的这段代码：

```
1  /* main.c */
2  int a = 0; // data
3  char *p1;
4  // 全局变量，不可以定义完之后再对它赋值，也就是说p1="hello
       "是非法的，但是直接char *p1="hello"是合法的。（全局变量
       存在data section里面）
5  int main(int argc, char *argv[]) {
6          int b;
7          char s[] = "abc"; // stack（由编译器自己管理，会自
               己释放空间）
8          char *p2;
9          char *p3 = "123456"; // stack
10         p1 = (char *)malloc(10); // heap（由用户管理，所以
               需要代码来释放空间）
11         p2 = (char *)malloc(20); // heap
12         return 0;
13 }
```



1.2.2 Process State

重点重点重点重点重点!!!!!!!、

我们的 Process 啊，一共有五个主要的状态：

（1） new:process 被创建了

（2） running: 指令在 CPU 被运行了

（3） waiting:process 在等待一些事件发生，比如 I/O

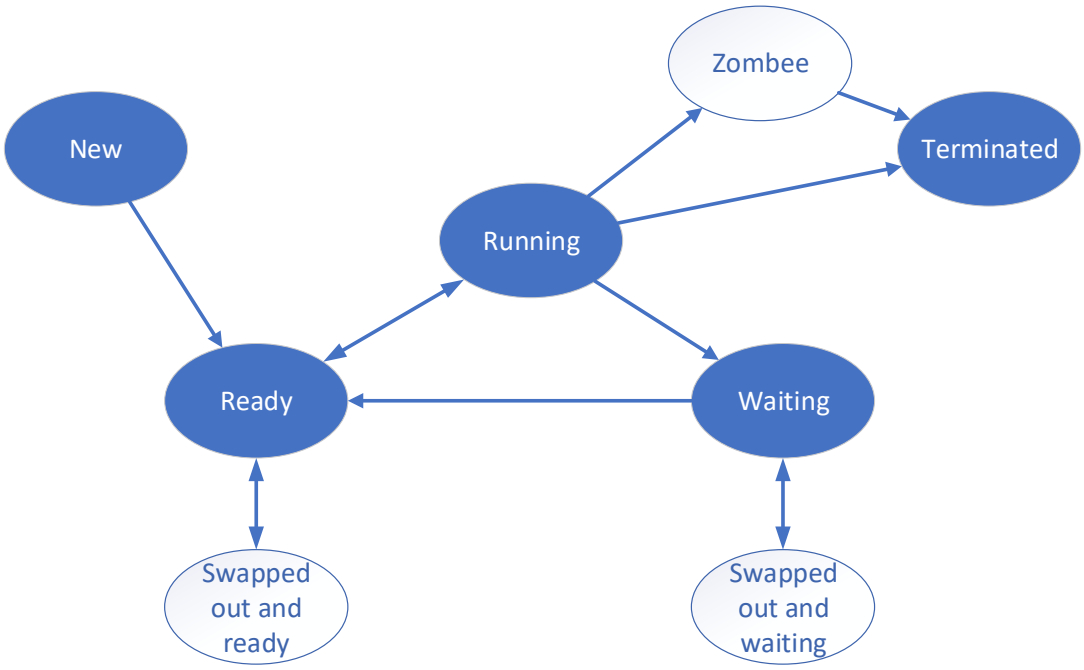
（4） ready:process 等待被分配到处理器，比如等待 cin,return,exit,ctrl+c

（5） terminated:process 完成运行

重点重点重点重点重点!!!!!!!

注意. *waiting time* 描述的是 *ready* 的时间，表示准备就绪等待调度。

超级无敌重点重点重点!!!!!!!!



我们一点点来讲 process state 的变化：

（1） new->ready: 进程创建完成，进入 ready 队列，这个过程由 long-term scheduler 处理（process 只要 create 了，就一定能跑，无非是什么时候让你跑）

（2） ready->running: 由 short-term scheduler 进行调度,process 被调度程序选中，获得 CPU 资源并开始运行

（3） running->ready: 进程 A 执行了一个时间片后，CPU 切到了进程 B，A 进入了 ready（这里是否是 short-term scheduler 还待考证）

（4） running->waiting：主要是内核在工作，通过 system call 把 running 的进程安排到 waiting，可能需要 wait for I/O 之类的，short-term scheduler 参与完成。

（5） waiting->ready: 由 short-term scheduler 完成调度，一般是 I/O 完成了

（6） running->zombee: 进程已经跑完了，但还有些信息

（7） running->terminated:Process 完成执行或者被强制终止

（8） ready->swapped out and ready: 将整个 process 给 swap out 出来，可能是因为系统负载更高，进程被暂时换出

（9） waiting->swapped out and waiting: 进程在等待 I/O，但由于内存紧张，被换出去了

（10） memory->disk 和 disk->memory 的 swapping 操作都是 medium-term scheduler 完成的。

（11） terminated 操作也是由 short-term scheduler 完成的。

1.2.3 Process Data Structure

那么问题来了，为什么我们用 process 而不用 program 呢？其实很简单，因为 program 就是一组指令的静态集合，他没有 heap、stack 啥的。

例 4. *那么，一个 program 可以停止然后再次被运行吗？*

答：不可以，除非我们能记录运行的 program 的机器状态 (machine state)，而 process 就是一个包含 machine state 的一个 running program。

定义 1.2. *Machine state: 包含 CPU registers，内存状态以及 I/O 状态。*

那么对于操作系统来说，它有哪些数据结构来追踪每个进程的状态呢？

一个是进程列表 Process lists（用于进程调度，决定哪个进程运行），里面有所有 ready/running/waiting 的 processes；那么这个 process list 里面的内容就是 Process Control Block(PCB)，每个 process 都有一个 PCB（包括 Program counter，process 的 state，process id 之类的东西，和 inode 很像），**PCB 就是 Process 的身份证**。

1.2.4 Process Execution

我们先来看一个直接执行协议的例子：

OS **(kernel mode)**	**Program**
create entry for process list	
allocate memory for program	
load program into memory	
set up stack with argc/argv	
clear registers	
execute call main()	
	run main()
	execute return from main
free memory of process	
remove from process list	

Direct execution 看上去很快，但是有两个问题：

（1） Protection: 没法确保 program 不会乱搞，比如说执行一些我们不想让他执行的东西

应对：使用双模式 (dual mode) 和 system call，也就是说，让 program 在 user mode 下运行，防止对系统进行误操作，那么如果 program 需要一些特殊的调用，就用 system call 就可以了。

（2） Time sharing: 我们想要做分时系统，那么就要保证 OS 能够把一个正在跑的进程切掉，换成另一个进程，所以我们需要有一个 timer 来做 context switch（时间片轮转），但是显然现在没有咯。

定义 1.3. *Context switch: 保存和恢复上下文（saving and restoring context）。*

那么要解决 Protection 的问题，看 PPT 的 19 到 21 页。

对于 Time sharing 的问题：在 processes 之间进行切换时，OS 通过 **timer interrupt** 获得 CPU 的控制权（hardware interrupt 能够做到强制获得 CPU 的控制权）；同时 OS 当中的 scheduler（**GPT 认为是 short-term scheduler**）决定到底换掉哪个 process。

补充一个概念：

定义 1.4. *Kernel Stack(k-stack): 理论上来说每个 process 都有一个 kernel stack，但董恺认为，kernel stack 看作一个更好理解，kernel stack 存储的是 kernel mode 下的临时信息，不包含 process 的 context，而 PCB 存的东西要比 kernel stack 是多一些的，在做 context switch 的时候，比方说 CPU 要从 process A 换到 process B，就要先把 CPU 的 Reg 里的内容（也就是 Process A 的内容）存到 kernel stack 当中，然后 kernel stack 再把这些内容写回 process A 的 PCB 里面，再把 process B 的 PCB 的东西存到 kernel stack 里面，kernel stack 再把 B 的数据写到 CPU 的 Reg 里面，这样就完成 context switch 啦！*

所以从上面的例子可以看出来，数据在 register、k-stack 和 PCB 里面都有一个备份。

现在有一个比较搞脑子的东西，讲的是两种 register saves/resotres:

（1） Timer interrupt（由 OS 控制，由 hardware 实现，通常发生在于 user mode 下运行 program）：在这种情况下，user registers 的内容会由 hardware 隐式存储到 kernel stack 当中。

（2） OS 决定 switch（可能是一些调度算法之类的，通常发生在于 kernel mode 下运行 program，那可能就是 syscall 之后进行了一些 switch），这个时候 OS 会把 kernel registers 里的东西存到 kernel stack，然后再从 kernel stack 存到 PCB 里面。

那么如果当一个 interrupt 发生时，另一个 interrupt 或者 trap 发生了怎么办，特别是当 kernel stack 的数据还没有存进 PCB 时？那很糟糕了兄弟。

但是一般性在处理一个 interrupt 的时候会 disable other interrupts。

当然，要是说每个 process 有多个 program counters 呢？就可以实现一个 process 在多个 locations 同时运行，也就是有多个执行路径->Threads。那么为了支持多线程，操作系统需要为每个线程保存相关的详细信息，这些信息会存在 PCB 中。

1.3 Process Scheduling

Process Scheduling 做的事情就是最大化 CPU 利用率，实现 process 之间的快速切换。那么对于 process scheduler 来说，要在一些 processes 之中选择下一个在 CPU 运行的 process。

与此同时，操作系统维护着几个不同的队列，用于跟踪进程的状态，帮助调度器选择下一个执行的进程，这样的队列一共有以下三种：

（1） Job queue（有 1 个）：系统中的所有 processes。

（2） Ready queue（有好几个）：所有在内存当中，等待执行的 Processes（也就是 ready 状态下的 processes）。

（3） Device queue（也有好几个）：所有正在等待执行 I/O 的 processes（也就是 waiting 状态下的 Processes）。

辨析. *Job queue 里面有所有 process，不管你有没有被加载到主存。然后，waiting state 的 process，也就是说 Device queue 里的 process 也是都在主存的。最后，不是说磁盘里只能有 program 不能有 process，如果内存紧张，就可以把 process 换到磁盘，它不会变成 program。*

1.3.1 Schedulers

定义 1.5. *Short-term scheduler(CPU scheduler): 决定哪个 process 下一个被执行，并且分配 CPU（process 的状态由 ready 到 running），因为它经常被调用，所以它需要响应得非常快。*

定义 1.6. *Long-term scheduler(job scheduler): 决定哪个 process 会被放到 ready queue（注意是 new->ready，不管 waiting->ready），因为它没那么经常被调用，所以会慢一些。但是 long-term scheduler 控制多道程序设计的程度 (degree of multiprogramming)*

诶？啥是多道程序设计的程度呢？

例 5. 比如说 *degree=10*，就代表着在 *main memory* 中有 *10* 个 *processes*，比方说有 *1* 个 *running*，*5* 个 *ready*，*4* 个 *waiting*，那么 *long-term scheduler* 可以做到增加这个 *degree*，但是不能减少，而 *short-term* 不能增加也不能减少-> 引入 *medium-term scheduler*（见后文）。

对于 processes 来说，可以分成以下两种：

（1） I/O-bound process: 花大部分时间做 I/O 而非计算，短时间的 CPU burst 较多，例如：读取大量数据，此时 I/O 不占 CPU，切到其他进程。

（2） CPU-bound process: 花大部分时间做计算，只有少量但是时间很长的 CPU burst，例如：数学计算、图像处理，因为很少但是时间长，所以中间可能会发生 CPU 饥饿。

所以对于 long-term scheduler 来说，他要做的事情就是很好地将这两类 process 做一个混合。

定义 1.7. *Medium-term scheduler:* 用来减少 *degree of multiple programming*，它做的事情是将 *process* 从 *memory* 里面拿走，放到 *disk* 里面，或者把 *process* 从 *disk* 里面拿回来，说的专业一点，叫 ***swapping***。

那为啥要降低多道程序的 degree 呢？防止 thrashing（第九章 Virtual Memroy->Thrashing）。

1.4 Operationgs on Processes

对于一个系统来说，它一定要具有 process creation、process termination 等功能。

1.4.1 Process Creation

如果一个 process 想要把任务分配出去，那么他就可以创建一些子进程（它也就被称为父进程），然后这些子进程还可以创建他们的子进程，这样就创建出了一个进程树。

那么在操作系统中如何对进程进行区分呢？我们还记得每个 process 都有一个 PCB 对吧，PCB 里面有一项叫做 pid(process identifier)，它叫进程标识符，就可以区分不同的进程。

那么在资源共享方面，父进程和子进程有以下这些资源共享的策略：

（1） parent 和 children 共享所有的资源

（2） children 共享 parent 的部分资源

（3） parent 和 children 不共享任何资源

在进程执行方面，父进程与子进程也有以下两种策略：

（1） parent 和 children 一起并发执行

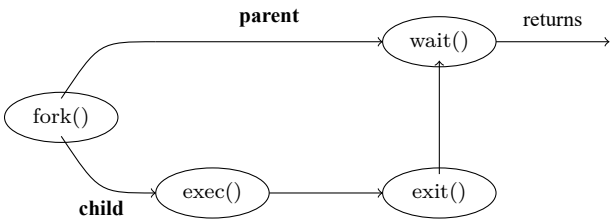
（2） parent 等待 children terminate 之后再执行

（3） 注意不会有 children 等 parent 的情况，因为这样不如 parent 再创建一个子进程

下面是重点重点重点!!!

那么复制完之后的地址空间呢？

对于子进程来说，他会复制父进程的地址空间（包括数据段、堆栈），同时子进程可以继续执行父进程的代码，同时也可以加载并执行新的程序。



例 6. 我们来举一个 *UNIX* 架构下的例子，首先有两个 *system call(API)*：*fork()*（创建新的进程，同时返回子进程的 *pid*）、*exec()*（中调用 *fork()* 之后，用一个新的 *program* 来替换当前的内存空间，包括 *data section*、*text section*、*heap*、*stack* 等）。

那么在上面这个图里面啊，比方说要发工资了 *parent* 干的事情是计算工资，而 *child* 做的事情是发工资。那么在 *fork()* 之后，因为一开始是复制的，在没有调用 *exec()* 之前，*child* 也以为他要做的事情是计算工资，后来才知道他要完成的工作是发钱。然后 *child* 运行完 *exit()* 之后，父进程才可以继续往下运行。

那么还要知道一个事情就是，当父进程运行了 fork() 之后，父进程和子进程都会从 fork 语句的下一句开始运行。我们来看下面的这段代码：

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main(int argc, char *argv[]) {
6         pid_t pid;
7         pid = fork();
```

```
8           // 注意在这个地方，对于父进程来说fork()返回的是子
                进程的pid，那么对于刚刚创建出来的子进程来说，
                fork()返回的是0，也就是pid在子进程中是0，所以可
                以说子进程是不知道自己的pid的。（我们约定创建出
                来的子进程的pid≥1）
9           if (pid < 0) { // error occurred
10                  fprintf(stderr, "Fork Failed");
11                  return 1;
12          }
13          else if (pid == 0) { // child process
14                  execlp("/bin/ls", "ls", NULL);
15          }
16          else { // parent process
17                  wait(NULL);
18                  printf("Child Complete");
19          }
20          return 0;
21  }
```

1.4.2 Process Termination

那么有进程创建，肯定也会有进程的终止对吧：

子进程的终止也能分为以下两种，一种是自己终止自己，另一种是父进程终止子进程：

（1） 当 process 执行完了它的最后一个语句的时候，它会调用 exit() 这个 system call 来让 OS 把自己删了（也就是 child terminates itself，他自杀了，像是下定了某种决心）。在这之后，父进程可以通过 wait() 这个 system call 来获取 child 的状态数据。最后，子进程的资源会被 OS 回收。

（2） 父进程也会调用 abort() 这个 system call 来终止子进程，可能是因为：子进程用超了分配给它的资源、分配给子进程的任务不再被需要了、父进程正在退出，而操作系统不允许子进程在父进程退出后继续运行。

我们再来详细讲讲说操作系统不允许子进程在父进程退出后继续运行这个事情：

定义 1.8. *Cascading termination*（级联终止）：父进程的终止会引发子进程（以及所有孙子进程等）的终止。同时，**子进程的终止不是 *parent process* 引起的，是由 *OS* 自动杀死的。**

当然 parent process 可以等待子进程完成，也就是用 wait() 这个 system call，一般用的代码是 pid=wait(&status)。

最后再讲两个 process 的特殊状态：

（1） Zombie: 父进程没有调用 wait() 来等待子进程的终止，那么子进程在终止后会进入“僵尸”状态。这意味着子进程已经结束，但它的状态信息还没有被父进程收集，导致它仍然占用一些系统资源。

（2） Orphan: 如果父进程提前终止，但没有调用 wait()，那么子进程就变成了“孤儿进程”。孤儿进程将由系统的初始化进程（也就是 init 进程）接管并继续执行。

注意. *zombie* 状态的进程已经结束了，但是 *orphan* 状态的进程是还没有结束的。

我们来看一个具体的代码来加深理解：

```c
/* p1.c */
int main(int argc, char *argv[]) {
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();
        if (rc < 0) {
                fprintf(stderr, "fork failed\n");
                exit(1);
        }
        // 子进程运行的代码
        else if (rc == 0) {
                printf ("hello, I am child (pid:%d)\n", (
                    int) getpid());
                char *myargs[3];
                myargs[0] = strdup("wc");
                myargs[1] = strdup("p1.c");
                myargs[2] = NULL;
                execvp(myargs[0], myargs);
                // 注意，子进程运行到上面execvp后，相当于
                    整个进程都被换掉了，后面任何东西都不会
                    执行，所以下面一行printf是不可能执行
                    的。
                printf("this shouldn't print out");
        }
        // 父进程运行的代码
```

```
21          else {
22                  int wc = wait(NULL);
23                  printf("hello , I am parent of %d (wc:%d) (
                        pid:%d)\n", rc, wc, (int) getpid());
24                  // 第一个传入的参数%d是rc的值，也就是子进
                        程的pid，第二个传入的参数为wc，同样是子
                        进程的pid（因为wc=wait()），最后一个传
                        入的参数是getpid()，也就是父进程的pid。
25          }
26          return 0;
27  }
```

```
1  prompt> ./p1
2  hello world (pid:29383)
3  hello, I am child (pid:29384)
4          29       107      1030     p1.c
5  hello, I am parent of 29384 (wc:29384) (pid:29383)
6  prompt>
```

1.5 Interprocess Communication

对于一个进程来说，他要么是 independent 的，要么是 cooperating 的（当然也可以这两种情况切来切去）：

如果是独立的，那很好理解，大家互不相关，但要是是合作的呢？他们之间是会彼此影响的，包括一些共享的数据，这样多个进程协同工作来完成一个任务，那么这样的 cooperating processes 需要什么呢？进程间通信（IPC），现在有两种 IPC 的 basic models：

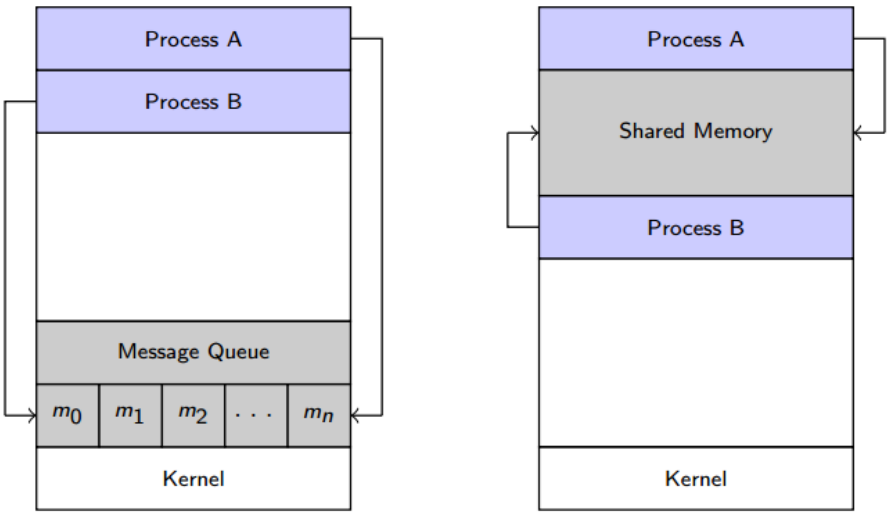
（1） Shared memory

（2） Message passing

1.5.1 Message Passing & Shared Memory

下面的东西确实会比较弯弯绕绕：

首先出场的是 message-passing model! 它给我们带来的是 pipe 和 message queue(Linux)!



那么在 message-passing model 当中，进程 A 和进程 B 之间通过发送和接收消息来进行通信。消息传递可以通过内核（Kernel）来实现，内核在这个过程中充当了一个消息交换的中介。

定义 1.9. *Message Queue:* 消息队列是一个存储消息的地方，消息可以通过该队列从一个进程传递到另一个进程。例如，进程 *A* 可以将消息 *m0*、*m1* 等放入消息队列中，进程 *B* 通过内核从队列中读取这些消息（同样遵循先进先出的顺序）。

这样做的优点是：提供了进程间的隔离，即使进程 A 和进程 B 的内存不共享，消息依然可以通过内核传递；但是缺点是：在高频率通信时，需要较多的上下文切换和内核的干预（要调用 system call 来访问消息队列）。**进程需要将数据复制一份送到消息队列。**

我们再来看 shared-memory model：进程 A 和进程 B 共享一个公共的内存区域。这意味着它们可以直接访问和修改该区域的内容，从而实现数据交换。

定义 1.10. *Shared Memory*：存在 *user memory space* 当中，不在内核，但是这块空间需要向 *kernel* 申请。进程 *A* 和进程 *B* 可以直接读取和写入共享内存中的数据，而不需要通过 *kernel* 传递信息，实际的数据交换发生在 *shared memory* 当中。

这样做的优点在于：共享内存的通信方式通常比消息传递更高效，因为它避免了消息的中介传递，进程之间直接通过共享区域交换数据，减少了开销。但缺点在于：共享内存需要进程之间协调和同步，以避免冲突或数据不一致的问题。对于不同进程的内存访问，需要考虑并发控制（如互斥锁、信号量等）。**进程可以直接在 shared memory 读取和写入数据。**

1.5.2 Producer-Consumer Problem

我们通过一个具体的案例（消费者-生产者问题）来比较一下这两种方案：

说简单点就是一个进程负责生产信息，另一个进程负责消费这些信息，比如说编译器是一个生产者，它将源代码编译为汇编代码；而汇编器是一个消费者，它接收并处理汇编代码，进一步生成机器代码或目标代码。

我们来看一下用 shared-memory solution 是怎么解决生产者消费者问题的：

```
/* Solution is correct, but can only use BUFFER_SIZE - 1
   elements */
#define BUFFER_SIZE 10
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

/* producer */
item next_produced;
while (true) {
        while (((in + 1) % BUFFER_SIZE) == out) ;
        // 表示缓冲区已满，等待
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}

/* consumer */
item next_consumed;
while (true) {
        while (in == out) ;
        // 表示缓冲区已空，等待
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

那么在这个案例当中，0 到 9 只是一个简单的索引，没有说 in=9 就一定是 buffer 为满，in=2，out=3 也是 buffer 为满。

Shared memory 的数据交流需要由进程自行管理数据访问，而不依赖 OS 的调度，也就是说，需要自己来编程。同时，面临一个主要的问题就是——同步（数据竞争 or 数据不一致问题）

接下来讲讲 Message Passing 的解决方法：

定义 1.11. *Message system*：进程之间进行交流但不使用共享变量，也就是说，它最大的优势是不用考虑同步的问题。

IPC 提供了两种操作：send(message) 和 receive(message)，这里的 message 的大小可以是固定的也可以是变化的。

那这个代码就会变得极其简单：

```
/* producer */
message next_produced;
while (true) {
        /* produce an item in next produced */
        send(next_produced);
}

/* consumer */
message next_consumed;
while (true) {
        receive(next_consumed);
        /* consume the item in next consumed */
}
```

1.5.3 Message Passing Vs. Shared Memory

这里和第一部分按理说是连起来的，但是中间穿插了一个生产者消费者问题也无伤大雅。

首先我们再来 detail 讲一下 message passing 背后到底干了什么事情。**个人感觉这块地方没有那么重要，知道就行了吧。（指直接通信和间接通信）**

通信分为直接通信 (direct communication) 和间接通信 (indirect communication)

对于直接通信，有两种命名方式：

（1） 对称地址：发送和接收方互相命名：

对于 process Q->send(P,message); 对于 process P->receive(Q,message)

（2） 非对称地址：只要发送方指定接收方就可以：

对于 process P->send(P,message); 对于 process Q->receive(id,message)，它能够接收任何 process 来的 message。

直接通信有这三个特质：

（1） 自动建立链接：每对想要进行通信的进程之间会自动建立一个通信链接。

（2） 链接关联性：每个通信链接仅与两个进程关联。

（3） 每对进程之间只有一个链接：每一对进程之间会存在且只存在一个通信链接。

那么我们再看看间接通信:间接通信的情况下,消息被送到或者从一个叫”mailbox”的地方接收，那么对于两个进程来说，或者多个进程来说，他们的代码就是简单的send(A,message) 和 receive(A,message)，A 是 mailbox。

间接通信也有三个特质：

（1） 链接的建立：当一对进程共享一个邮箱时，才会在这两个进程之间建立一个通信链接。

（2） 多个进程的链接：一个邮箱可以与多个进程建立关联，支持多个进程间的通信。

（3） 多个链接：对于每一对通信进程，可能会存在多个不同的链接，每个链接对应于一个邮箱。

下面的内容董恺稍微花了点时间讲：

对于消息的传输也有两种形式，一种是 blocking（阻塞的），另一种是 non-blocking（非阻塞的）：

（1） Blocking(synchronous：同步)：做个简单的类比，我们人有两条腿走路，一定是迈完左脚，要 block 一下，再等待右脚迈出来。

Blocking send（阻塞发送）：发送方被阻塞，直到消息被接收。

Blocking receive（阻塞接收）：接收方阻塞，直到有消息可以供我接收。

（2） Non-blocking(asynchronous：异步)：我在操场上跑 1000 米，我不带停的，所以这样很快。

Non-blocking send（非阻塞发送）：发送方一直在那里发送。

Non-blocking receive（非阻塞接收）：接收方接收到的可能是无效的消息或者是空消息。

当然，排列组合一下就可以发现，一共有四种搭配方法，那么怎么搭配都是可以的，当 send 和 receive 都是 blocking 的时候，我们称它是 rendezvous（会面）。

最后一个概念是 Buffering，也就是 message passing 会有一个 temporary queue，有以下三种情况：

（1） Zero capacity（刚生产出来就必须消费掉，虽然确保消息的传输，但是想想就知道这样的效率是很低的）

（2） Bounded capacity：有界容量

（3） Unbounded capacity：无界容量

最后的最后啊，我们对 message passing 和 shared memory 做一个比较：

注意. 记得我们所有的比较都是宏观的比较，不太需要涉及到具体的实现方案。

我们先来看看 Message passing 的优势（**稳定**）：

（1） Message passing 适合少量数据的交换，因为不会出现多个进程的冲突

（2） 如果想要在不同的计算机之间进行通信，message passing 会更容易实现

（3） 在多处理器系统中，共享内存可能面临缓存一致性问题，这可能导致性能下降。而消息传递通过明确的消息队列和通信方式，避免了这种问题。

再来看看 Shared memory 的优势（**快**）：

（1） 消息传递通常是通过系统调用实现的，这要求内核介入，增加了开销。而在共享内存系统中，系统调用只需要在初始化共享内存区域时进行，后续操作没有内核介入，效率更高。

从未来走向的角度说，如果要实现 IPC，message passing 会更加流行。

1.6 Communication in Client-Server Systems

定义 1.12. *Socket*（套接字）：用于网络通信的端点，通常由 *IP* 地址和端口号组合而成，是实现网络通信的基础。

例 7. 在 *UNIX* 环境下，用 *pipe()* 这个 *system call* 来实现 *message passing*，管道允许一个进程的输出与另一个进程的输入相连接。管道通常用于连接两个相关的进程，使得一个进程的输出直接作为另一个进程的输入。

管道还分为以下两种：

（1） Ordinary pipes（普通管道）：只能在父子进程之间进行通信的管道。通常情况下，父进程会创建管道，并且子进程通过该管道与父进程通信。普通管道无法在父进程外部访问。

（2） Named pipes（命名管道）：与普通管道不同，命名管道可以在没有父子进程关系的情况下进行访问。命名管道是给定名字的管道，允许不同行的进程之间通过文件系统中的路径进行通信。这使得命名管道在某些情况下比普通管道更灵活。

2 Exercise

例 8. 读下面的代码，*what is the output?*

```c
/* exercise.c */
int value = 5;
int main(int argc, char *argv[]) {
        pid_t pid;
        pid = fork();
        if (pid == 0) {
                printf("child process, value1 : %d\n",
                    value);
                value += 15;
                printf("child process, value2 : %d\n",
                    value);
        }
        else if (pid > 0) {
                printf("parent process, value3 : %d\n",
                    value);
                wait(NULL);
                printf("parent process, value4 : %d\n",
                    value);
        }
        exit(0);
}
```

Keys:

```
prompt> ./exercise
child process, value1 : 5
child process, value2 : 20
parent process, value3 : 5
parent process, value4 : 5
prompt>
```

```
prompt> ./exercise
parent process, value3 : 5
child process, value1 : 5
child process, value2 : 20
parent process, value4 : 5
```

```
prompt>
```

补充说明：一般不会出现先 value1，再 value3，再 value2，再 value4，虽然它是符合逻辑的。

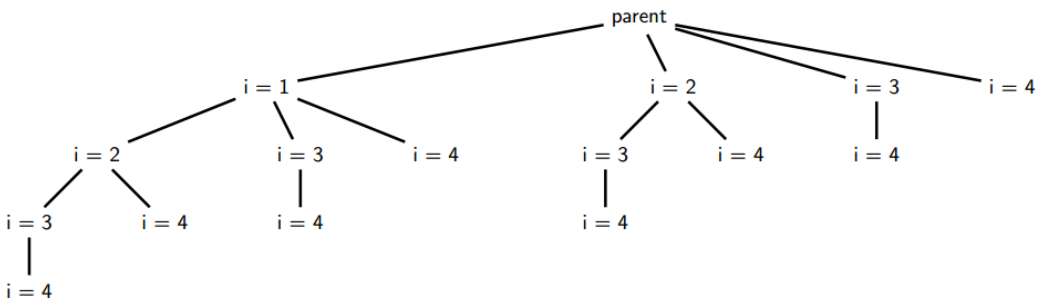
例 9. *Including the initial parent process, how many processes are created by the following program?*

```c
#include <stdio.h>
#include <unistd.h>

int main() {
        int i;
        for (i = 0; i < 4; i ++)
                fork();
        return 0;
}
```

一些小提示，考试的时候可以画树状图。

Keys:16 processes.



3 Concepts Organization

1. Processes:A program in execution; process execution must progress in sequential fashion.
2. Context switch:A context switch is the process of saving the state of a currently running process and loading the state of the next scheduled process, allowing the CPU to switch between tasks in a multitasking environment.
3. PCB:A struct that records the machine state of the process.
4. Long-term scheduling：Select which processes should be brought into the ready queue.
5. Medium-term scheduling:Used to decrease the degree of multiple programming.
6. Short-term scheduling:Select which process should be excuted next and allocates CPU.
7. Shared memory:A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
8. Message passing:Communication takes place by means of messages exchanged between the cooperating processes.
9. Pipes:The output of one process is connected to an in-kernel pipe. And the input of another process is connected to that same pipe.

第四章　Threads

1　In-class Contents

1.1　Warm-up

定义 1.1. *Thread:process* 的一部分，共享进程的内存、全局变量等资源；线程之间的通信比进程间更加高效，但容易发生数据竞争、同步等问题，也就是说，如果一个线程寄了，整个进程可能就没了。

本质是一个执行指令序列，允许程序同时进行多个操作。

我们来考虑一个事情，为什么一些 programs 会有多个执行点？比方说把两个大数组相加或者是输入一些元素的时候，program 可以把 task 分成几个 subtasks，再有 multiple threads 来完成这些 subtasks，听上去这样就可以加快任务的完成。

那么我们还有个问题，为什么用多线程而不用多进程呢？

1.2　Overview

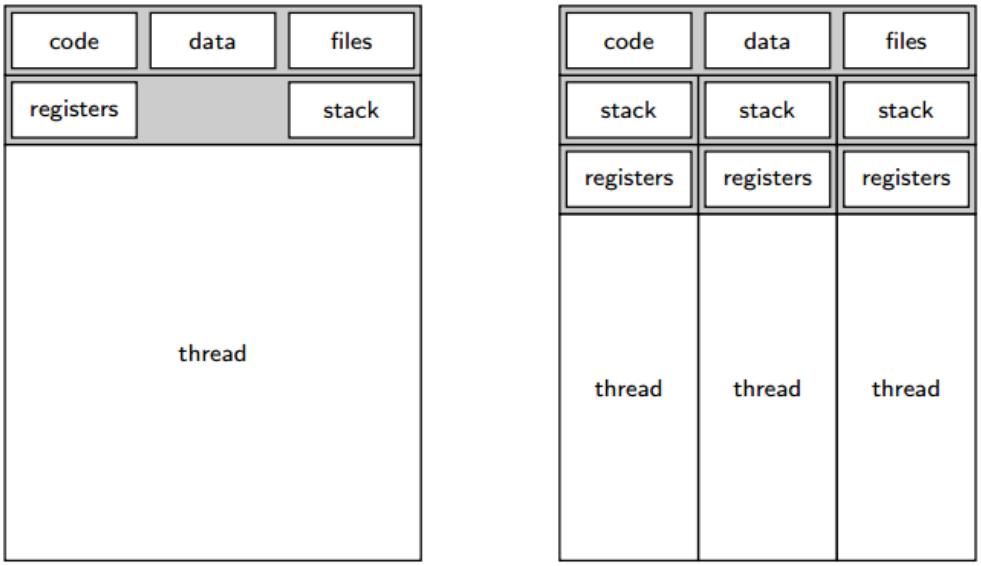
现代的应用程序大多都是多线程的，也就是说线程在应用程序中工作，完成不同的工作，比方说 word processor 有 update display、fetch data 和 spell checking 的工作，就会由多个线程一起完成。

那么为什么用 multithread 不用 multiple processes 呢？因为 multiple threads 能够减少开销，公用内存空间。

OS 的 kernel 基本上就是 mutithreaded 的。

1.2.1　Multithread

我们首先来看看 single-threaed process 和 multi-threaded process 的区别：



Single-Threaded Process **Multi-Threaded Process**

可以看到，一个 process 里的 threads 是共享 code、data 和 files 的，但他们有自己的 stack 和 registers，存放局部变量和线程的执行状态，例如 PC 和 IR。

那么这样做有什么好处呢？

（1） Responsiveness（响应性）：当程序的一部分被阻塞时，线程可以允许其他部分继续执行，这在用户界面中尤其重要。这样，程序能够更快地响应用户输入或操作，而不必等待阻塞的部分完成。（只有 multiple thread 才能这样继续执行）

（2） Resource Sharing：线程之间分享资源，比进程之间通过 shared memory 或者 message passing 更容易，开销也更小。

（3） Economy：创建新的进程只需要 stack 和 register 就可以了，创建 process 就很复杂了，而且在 thread switching 的开销比 context switching 要小很多，不用在 PCB、Kernel-stack 啥里面的搞来搞去。

（4） Scalability（可扩展性）：就这样吧，知道就行了；process can take advantage of multiprocessor architectures。

1.3 Multicore Programming

这个部分 dk 说没那么重要

Multicore 或者 Multiprocessor 在编程上有这些挑战：

（1） Dividing activities（分割活动）：将任务分成多个可以并行的小任务，那么要想好怎么去合理分配。

（2） Balance（工作负载）：尽量让每个处理器或者核心负载平衡

（3） Data splitting：将数据集分割成多个部分，以便在不同的线程或处理器之间共享

（4） Data dependency：确保任务按照正确的顺序执行

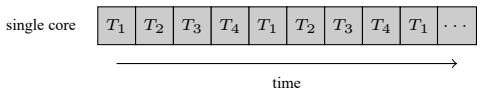
（5） Testing and debugging

两个我认为比较重要的概念：

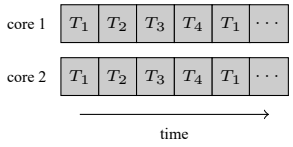
定义 1.2. *Parallelism*（并行）：并行性意味着一个系统能够同时执行多个任务。多核或多处理器系统通过在**不同的核心或处理器**上执行任务来实现并行处理。例如，在一个四核处理器上，程序可以将**四个不同的任务**分配到四个核心上，确保每个任务同时执行，从而加快整体执行速度。

定义 1.3. *Concurrency*（并发）：并发性指的是多个任务能够在同一时间段内有进展，但并不一定是同时执行。即使在单核处理器中，操作系统也可以通过**时间片轮转调度，使得多个任务交替进行**，给人一种“同时进行”的感觉。并发性适用于**单核处理器或核心**的情况，操作系统调度器通过快速切换任务，让它们看起来在同时执行。

看上去是不是不太直观？没关系。我们首先看看 Concurrency 的时候是什么样的：



再看看 Parallelism 是怎么样的：



Parallelism 有两种类型：

（1） Data parallelism（数据并行性）：在多个处理器核心上分配同一数据的不同子集，每个核心执行相同的操作。这样可以提高执行效率，特别适合于需要对大量数据进行相同计算的任务。

（2） Task parallelism（任务并行性）：将任务分配到不同的核心，每个核心执行不同的操作。每个线程执行独立的操作，适合于任务之间相互独立的情况。

现在有一个小问题，添加 core 的数量，我们到底能够提升多少的 performance？假设我们有 N 个 proceesing cores。

1.3.1 Amdahl's Law

具体看 PPT14 页，公式如下：

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

1.4 Multithreading Models

1.4.1 User Threads and Kernel Threads

首先要明确一个事情，或者这个事情能帮我更好去理解这些概念：**Thread 和 Process 几乎是一样的，只是它们的名字不一样罢了。**

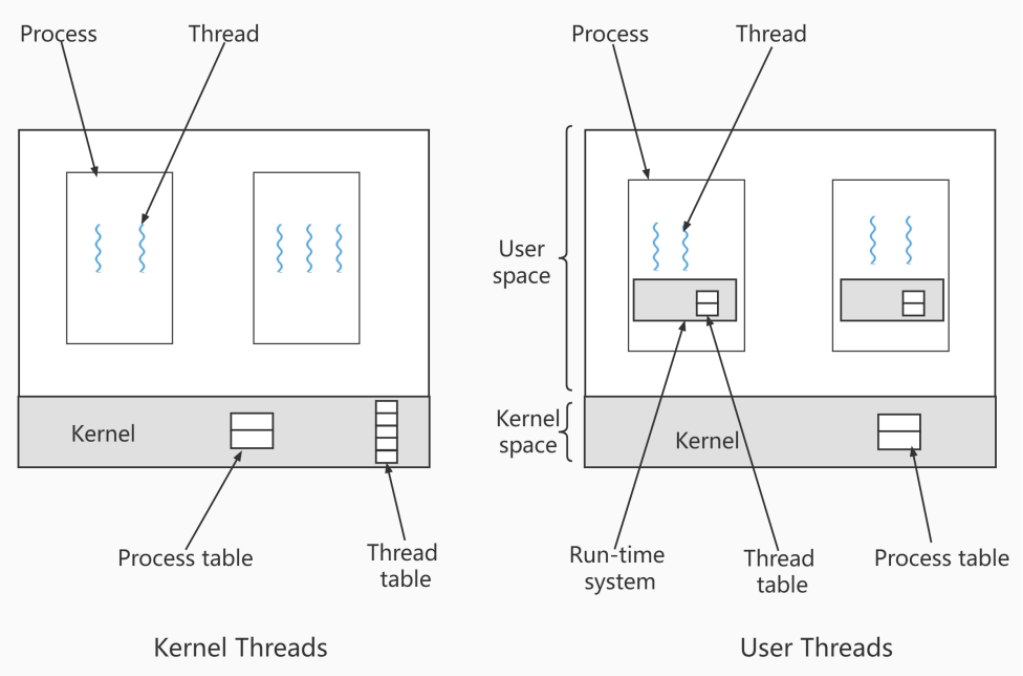
那么如何去支持这些 threads 呢，又是谁来支持这些 threads 呢？

别着急，我们先聊一下有哪些 thread 的类别（说是说类别，其实也就是它们的 library 的名称不同罢了，本质没啥区别）：

（1） User threads：由用户级线程库进行管理，而不是由操作系统内核直接管理，不会涉及 system call，完全在用户空间完成。但是这样做有一个缺点：如果一个线程阻塞（例如执行 I/O 操作），整个进程可能会被阻塞，因为操作系统看不到独立的线程调度，而只看到一个进程。

比方说，POSIX Pthreads、Windows threads 等就支持用户线程

（2） Kernel threads：从名字就能看出来，这类 threads 由操作系统内核直接支持和管理。内核负责创建、销毁、调度线程，并且可以处理线程的并发执行。几乎所有操作系统都支持 kernel threads。



我们先看看 Kernel threads 吧，这个看上去比较好理解：

内核线程是由操作系统内核完全管理的。线程的创建、调度、切换、同步等操作都**需要通过系统调用**来与内核交互，所以左边整张图都表示的是 kernel space 里面的情况。

同时，有一个 process table 管理所有 process 的信息（通常就是 PCB 啦），当然也会有一个 thread table，里面就是所有 thread 的信息，虽然书上没说，但是 GPT 认为是 TCB。

再看看右边这张图，这里表示的是一些 user threads。在用户线程的模型中，进程和线程主要是在用户空间中进行管理。进程表依然在内核空间中，**但线程表和调度工作主要在用户空间完成**，这种情况下 kernel 只会提供一些必要的支持，比如 I/O，但是线程的一些操作并不是由 kernel 完全搞定的。

考点 4. 进程可以通过以下几个方式管理用户级线程：

（*1*） *PCB*

（*2*） *TCB*

（*3*） **内核不会管，编程来管理 *Threads*（重要重要重要!!!）**

我们现在再来探讨一些问题：

（1） Why we need user threads? 没有内核干预，因此用户线程通常会更高效。

（2） Why user threads are not enough? 这就要说到 user thread 的缺点了：内核只识别进程，并且只对进程级别进行管理。如果一个用户线程被阻塞（例如等待 I/O 操作），整个进程也会被认为是阻塞状态。因此，同一进程中的所有线程都会被阻塞，即使其他线程并不需要等待。这就降低了系统的效率，特别是当进程内部有多个线程需要并发执行时。

（3） Why we need kernel threads? 诶嘿，这就要说到 kernel thread 的优点啦：当一个线程被阻塞时，内核会运行其他线程，而不会影响同一进程中的其他线程。操作系统能够独立调度每个线程，而不仅仅是整个进程（这才是真正的并行管理）。

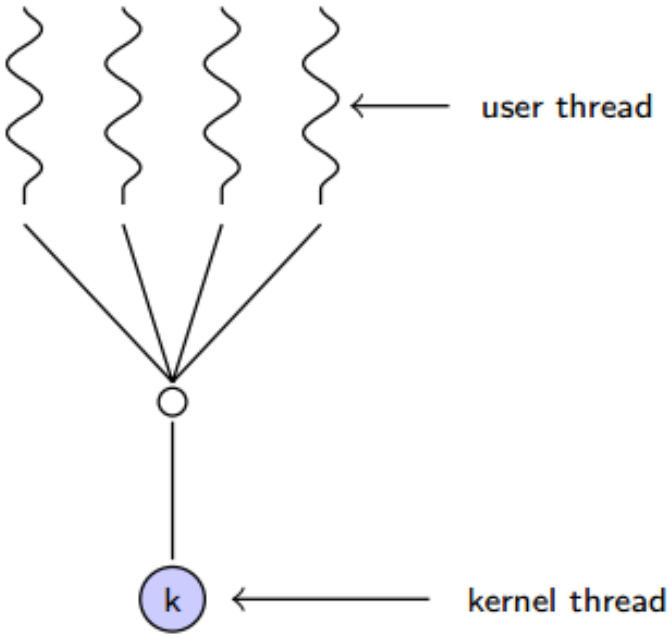
（4） Why kernel threads are not enough? 诶嘿（不是哥们，你戏有点太足了啊）很简单，kernel threads 比较慢呗。

那么 kernel thread 和 user thread 之间有什么关系呢？一共有下面三种关系：

我们首先要知道一件事情，**对于内核来说，它只能看到内核线程，而内核线程告诉用户线程决定执行的 user thread，而内核是不知道运行的 user thread 的情况的。**

1.4.2 Many-to-One

在这种模型中，多个用户级线程会被映射到一个内核级线程上。也就是说，**操作系统的内核只知道一个线程**，这个线程处理所有用户线程的调度和管理。

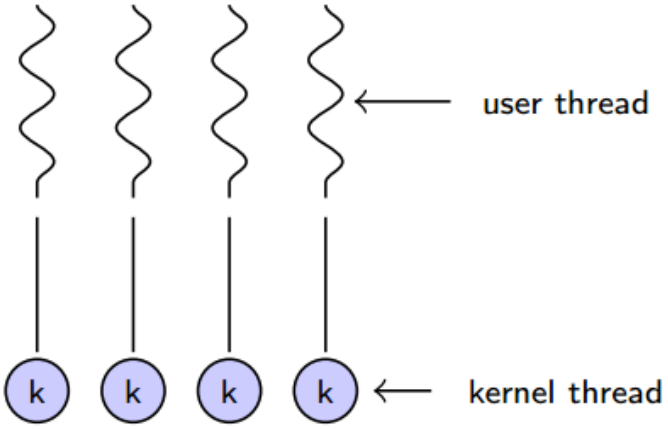


有什么问题呢？有一个 user thread block，那就全部 block 了，为什么呢？因为在这个模型当中，那个 kernel thread 就像是一个 process 一样，而 kernel thread 又只认那个正在运行的 thread，所以如果那个 thread block 了，那整个就垮了，因为内核不知道发生了什么，它也没办法切到其他 user thread 上面。

由于所有用户线程都在一个内核线程上运行，这意味着即使有多核 CPU，多个线程也无法真正地并行运行，除非系统支持其他线程模型。而且这个模型只能用于不支持 kernel thread 的操作系统，所以它很过时，没人用的。

1.4.3 One-to-One

从名字上就可以听出来，每个 user thread 对应一个 kernel thread。

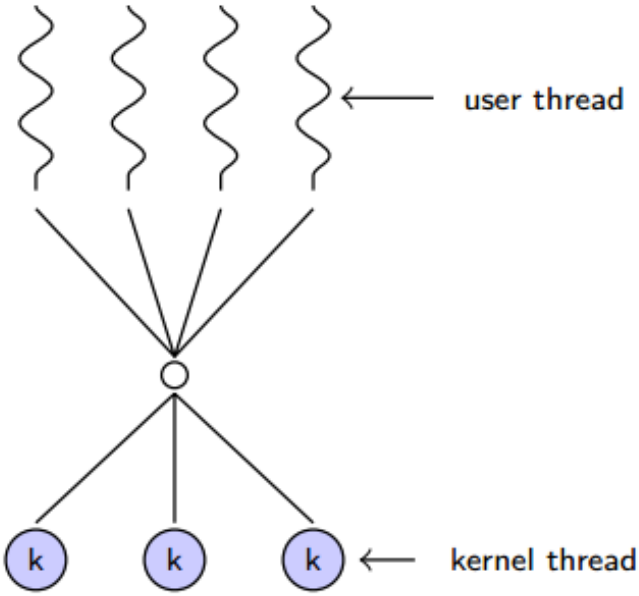


所以说在创建一个 user thread 的时候，也会创建一个 kernel thread。

这样做有什么好处呢？提高了并发性。因为在这种模型下，每个用户线程都可以独立执行，并且内核能够直接管理这些线程，但是代价就是开销有一点大，所以说每个进程能够有的线程数量是比较有限的。

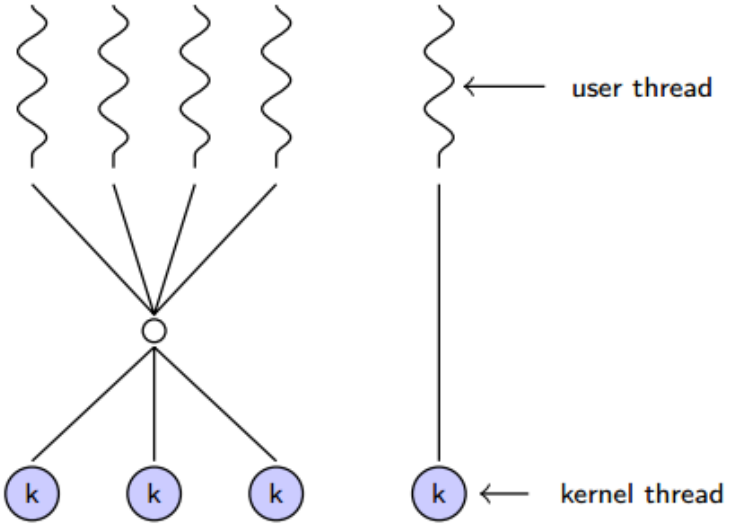
1.4.4 Many-to-Many

这就不能不提到我们 most flexible 的一个模型，顾名思义，many user threads 对应 many kernel threads，且一般 user thread 的数量大于 kernel thread 的数量。（GPT 认为，并发运行的线程数量等于 kernel thread 的数量）



1.4.5 Two-Level Model

你以为这样就结束了吗？并没有，其实还可以把 many-to-many 和 one-to-one 合在一起组成一个 two-level model。因为如果在 many-to-many 的模型当中，有三个 user thread block 了，那么整个模型也没用了，增加几个 one-to-one 可以保障高并发。



1.4.6 Conclusion

做一个小小的总结，那么这些模型当中，到底哪些在 modern systems 中更加受欢迎呢？

在服务器中，常用 many-to-many，因为它能可以更高效地处理大量并发线程。而在 PC（个人计算机）当中，更多用的是 one-to-one，因为这样可以更直接的分配系统资源。（这背后有一个小故事，在 LinuxThreads 当中，NGPT 推崇的是 M:M，而 NPTL 推崇的是 1:1，最后 NPTL 因为商业原因，而非技术原因胜出了）

1.5 Thread Libraries

定义 1.4. *Thread Library*：线程库为程序员提供了创建和管理 *threads* 的 *API*。

有以下两种实现线程库的方式：

（1） 完全在 user space 的线程库，操作系统不直接管理

（2） 由操作系统内核支持的内核级库，操作系统直接管理

这里呢，有三种主要的 thread libraries：**POSIX Pthreads**、Windows、Java，然后我们会重点看一下 POSIX Pthreads。

1.5.1 Pthread

首先，pthread 在 user-level 和 kernel-level 都有实现，且其提供的是一个规范（Specification），而不是具体的实现（Implementation）。换句话说，POSIX 定义了线程库的行为和功能，但它并不规定如何实现这一功能。不同的操作系统或线程库可以根据该规范来设计和实现自己的 Pthreads 库。最后 Pthreads 在 UNIX 类操作系统中常见。

我们首先来看看 Pthread 的创建：

```
1 #include <pthread.h>
2 int
3 pthread_create( pthread_t *      thread,
4         const   pthread_attr_t *attr,
5                 void *          (*start_routine) (void*),
6                 void *          arg);
```

我们来一点点看这四个参数：

*thread 是指向线程的指针，用来返回创建的线程。

*attr 指定线程的属性（比如栈的大小、优先级），那么如果不需要特殊属性的话直接传 NULL 就行。

(*start_routine) (void*) 这是线程入口函数，该函数将在线程启动时执行。

最后 *arg 就是传递的一些参数。

看上去很复杂对不对？不要着急，我们来看一个具体的例子就 ok：

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d␣%d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;
        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        ...
}
```

我们主要关注 pthread_create(&p, NULL, mythread, &args) 这行代码，可以看出来，我们在这里创建了一个进程 p，没有什么特殊的属性，创建完成之后就开始运行 *mythread 函数，传递的参数是 args，其中 args.a=10，args.b=10，*mythread 函数会输出 m->a 和 m->b，也就是 10 和 20。

那么我们已经知道怎么去创建一个线程了，现在我们再来看看我们如何等待一个线程执行完成：

```
#include <pthread.h>
int
pthread_join(    pthread_t *      thread,
```

```
4                    void **          value_ptr);
```

那么在这里显然 *thread 就是我们要等待的线程，**value_ptr 则用来接收线程退出时的返回值。pthread_create 创建的线程可以通过 return 语句返回一个值（通常是 void * 类型），pthread_join 允许获取这个返回值。

1.5.2 Java Threads

咱们再顺带提一下 Java Threads。在 Java 中，线程是由 Java 虚拟机（JVM）管理的，JVM 负责线程的创建、调度和生命周期管理。Java 程序通过 JVM 与操作系统提供的底层线程模型进行交互。

那么怎么创建 Java Threads 呢？有两种方式：继承 Thread 类或者实现 Runnable 接口。（不太重要，如果要看的话看 PPT35 页）

1.6 Implicit Threading

随着多核处理器的普及，程序需要使用更多的线程来提高并发性和性能。当线程数目增多时，程序的复杂性也相应增加。尤其是当程序员手动管理大量线程时，正确性和性能优化就变得越来越难。而显式线程管理需要程序员手动创建、同步和管理每个线程，这在处理大规模并发时非常复杂，容易引入竞态条件、死锁等问题。

所以呢，我们引入隐式线程。隐式线程的创建和管理不再依赖于程序员，而是由编译器和运行时库自动完成。程序员只需要指定程序的并行性需求，具体的线程创建、调度和同步交给编译器和运行时库来处理。

定义 1.5. *Thread Pool*：预先创建一定数量的线程，并将它们放入一个线程池中，等待执行任务。当任务到来时，线程池中的线程会取出任务进行执行，而不是每次都创建一个新线程。

用 Thread pools 有什么好处呢？

（1） 比创建新线程稍微快：使用线程池可以更快地处理任务，因为线程池中的线程已经创建好，并且准备好执行任务。与每次创建新线程相比，线程池避免了线程创建和销毁的开销。

（2） 线程池中线程的数量是有限的，程序可以通过设置池的大小来控制并发的线程数。这能有效避免创建过多线程导致的资源竞争和系统负担。

（3） 通过将任务执行与创建任务的机制分离，线程池允许在任务调度上使用不同的策略。例如，线程池可以周期性地执行任务，或者按照其他策略来调度任务。

1.7 Threading Issues

1.7.1 Semantics of fork() and exec()

现在我们来思考一个问题，如果说有一个 thread，要调用 fork()，那么它复制的是这个 process 的所有 threads，还是说新的 process 是单线程的呢？

答：新的 process 是单线程的，（一般情况下）只会有调用 fork() 的那个线程的内容。（当然有两个版本的 fork()，都是存在的）

注意. 线程调用 *fork()*，复制出来的是一个进程，不是线程。

说完了 fork()，我们再来看一下 exec()：一旦调用 exec()，当前进程空间完全被新程序替代，包括所有线程资源也会被清除，只留下主线程执行新程序。

一个小的讨论：如果说在 fork() 之后直接调用 exec()，那么应该复制 process 的所有线程还是只复制一个线程。

显而易见啊，你的资源都会被清除，肯定创建一个线程就够了。

1.7.2 Thread Cancellation

正常情况下线程会执行完自己的任务后自动退出。线程取消是指在外部主动发起让某个线程提前结束运行的行为。（可能是因为出现异常或者线程运行超时了吧）

有两种 cancel thread 的方式：

（1） Asynchronous Cancellation（异步取消）：直接把线程杀掉，但是可能会有些资源来不及释放造成 deadlock 或内存泄漏。

（2） Deferred Cancellation（延迟取消）：定期检查一个线程是不是该被杀掉，这样做更加安全，因为线程的资源能来得及释放。

然后呢，调用了 thread cancellation 也只是请求终止线程，但是真正的终止还要看线程的状态 (state)：

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

在这个表格当中，Mode（取消模式）：表示线程是否允许被取消，以及如何被取消；State（启用状态）：Disabled：取消请求挂起（pending），线程暂时不会响应；Enabled：线程准备好响应取消；Type（取消类型）：Deferred：线程在特定位置（取消点）才响应取消；Asynchronous：线程几乎立即响应取消请求。

注意. 对于一个线程来说，不出意外的话它只能是表中三个状态中的一个。（当然，可以是 ***Mode*** 为 ***Deferred***，***State*** 为 ***Disabled***，这个时候 ***Type*** 就没有，这是可以的）

那么如果 Mode 为 Off 的时候，你就没办法把这个线程删掉，除非 state 后面编程 enabled。

默认情况下会选用 Deferred 的 Mode 和 Type，那么线程只有到达 cancellation point 才会被 cancel 掉，这个时候就会释放资源，删掉线程。

最后，在 Linux 操作系统当中，线程的取消是用 signals 来完成的。

1.7.3 Signal Handling

定义 1.6. *Signal*：一种异步通知机制，用于通知进程发生了特定的事件（常见的事件包括 *ctrl+C*、除以 *0* 的非法访问等）；*signal* 本质是内核向进程发送的中断请求。

定义 1.7. *Signal Handler*：用来处理信号，一般有两种处理方式：

一种是 *Default handler*（默认处理器）：系统自动为每种信号定义了默认动作；另一种是 *User-defined handler*（用户自定义处理器）：为某些信号编写自定义响应函数。每个 *signal* 都会有一个 *default handler*，但如果程序员自定义了一个 *user-defined signal handler*，那它就会覆盖 *default handler*，这很好理解。

所以说信号的传递过程也非常的简单：某个事件发生，触发信号；信号被送到 process；signal handlers（default 或者 user-defined）来处理这些信号。

但是，很遗憾的是，上面只针对单线程的情况，signal 直接送到对应的 process 就结束了。

```
1  #include <stdio.h>
2  #include <signal.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  // 自定义信号处理函数
7  void handle_sigint(int sig) {
8      printf("\n␣␣收到␣SIGINT（信号编号␣%d），正在优雅退出
           ...\n", sig);
9      exit(0);  // 正常退出程序
10 }
11
12 int main() {
13     // 注册 signal handler
14     signal(SIGINT, handle_sigint);
```

```
15
16      printf("程序运行中，按␣Ctrl+C␣触发␣SIGINT␣信号...\n");
17
18      // 无限循环，等待信号到来
19      while (1) {
20          printf("□␣程序继续运行中...\n");
21          sleep(1);   // 每秒打印一次
22      }
23
24      return 0;
25  }
```

那么我们现在还得想想如果是一个 multi-threaded 的 process 该怎么办呢？有下面四个方法，由 process 来决定用哪个方案：

（1） 直接送到信号应用的线程:适用针对某个特定线程触发的信号（比如 pthread_kill (tid, sig)）

（2） 把信号送给所有线程：不太常见

（3） 某些线程可接收信号，其它线程忽略，例如通过线程屏蔽 sigprocmask() 控制。

（4） 分配一个专门的线程来负责接收和处理所有信号，其他线程则屏蔽信号。（最常见的方法）

还有一个小问题：每个线程能有自己的 handler 吗？

不可以！Handler 是进程级共享的，也就是说进程里的所有 threads 用的都是一样的 signal handler，如果你要搞特殊，你就屏蔽信号就可以了。

1.7.4 Thread-Local Storage

那么，我们如何在执行 counter++ 时实现并行性？只要让每个线程分别进行计数，最终汇总结果就可以了。

这时候就可以引入我们的 TLS(Thread-local storage) 啦，**它允许每个线程有自己的数据，这些数据不会被其他线程访问**。这是多线程编程中的一个重要概念，特别是当你使用线程池时（无法控制线程的创建过程），TLS 可以确保每个线程都有其独立的数据。

那么 TLS 和局部变量有什么区别呢？

局部变量只在函数调用期间有效，它们的作用范围仅限于当前函数。**TLS 则是跨函数调用有效的**，它可以保证每个线程在整个生命周期中都有独立的存储空间。

那么 TLS 和静态数据又有什么区别呢？

虽然 TLS 和静态数据都存储在程序的特定位置，但 TLS 是为每个线程单独分配的。每个线程都有自己的 TLS 数据，而静态数据是所有线程共享的。

拓展. *TLS 不在 data section，也不在 thread 的 stack 里面，一般存在 thread 的 TCB 里面。*

1.7.5 Scheduler Activations

还有最后一个 Issue，再坚持一下呀！

现在还有一个问题，在 M:M 或者是 Two-level 模型里，如何维护一个合适数量的 kernel threads？答：Communication（communicate 的内容包括用户线程要什么时候执行、内核什么时候有资源分配）. 但是问题是，谁知道有多少个 user threads 呢？只有用户自己知道，kernel 是不知道的。

定义 1.8. *Lightweight process：是一个内核级进程，因为用户进程不能直接运行在 CPU 上，所以要绑定一个 LWP 来实现调度。怎么实现的呢？首先操作系统通过短期调度决定 kernel space 的哪个 kernel thread 获得 CPU，然后 LWP 相当于把这些 CPU 时间带到 user sapce 里面，分给下面的 user thread。*

注意. *每个 LWP 对应一个 kernel thread。*

定义 1.9. *Scheduler activations（调度激活）：当发生关键事件（如线程阻塞、唤醒、CPU 可用等）时，内核会主动通知用户线程库*

定义 1.10. *Upcall：一种由内核空间的通信机制，这样线程库可以及时做出响应，比如：重新调度用户线程；释放或复用 LWP；创建更多 LWP 等。*

那么有 upcall 也一定有 downcall 对吧。

定义 1.11. *Downcall：比方说有两个 process A 和 B，这时候 A 要 B 为它服务，就叫了一个 procedure call 给 B，让 B 来处理，这时候 B 就会进入 kernel space 来处理这个问题。（说白了就是 system call）*

做一个小总结，我要运行 user thread 的时候，先把它们绑定到 LWP 上，实际上是由 kernel thread 执行的。

2 Exercise

例 10. 阅读下面的代码，输出的结果是什么？

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

typedef struct __myarg_t {
 int a;
 int b;
} myarg_t;

typedef struct __myret_t {
 int x;
 int y;
} myret_t;

void *mythread(void *arg) {
 myarg_t *m = (myarg_t *) arg;
 printf("%d␣%d\n", m->a, m->b);
 myret_t *r = Malloc(sizeof(myret_t));
 r->x = 1;
 r->y = 2;
 return (void *) r;
}

int main(int argc, char *argv[]) {
 int rc;
 // 这里似乎没有用到
 pthread_t p;
 myret_t *m;
 myarg_t args;
 args->a = 10;
 args->b = 20;
 Pthread_create(&p, NULL, mythread, &args);
 Pthread_join(p, (void **) &m);
```

```
35  printf("returned %d %d\n", m->x, m->y);
36  return 0;
37 }
```

Key:

其实很简单，首先 Pthread_create 调用 mythread 函数，输出 10 20；之后再有 Pthread_join 获取 mythread 的返回值，也就是 *r，输出 r->x 和 r->y，也就是 1 和 2。

所以答案就是：

10 20

returned 1 2

例 11. 阅读下面的代码，写出可能的输出（*Process* 和 *Thread* 章节的难度天花板，但同时也是一道真题）

```
1  /* kai.c */
2  #include <stdio.h>
3  #include <pthread.h>
4  void *helloFunc(void *ptr) {
5          int *data;
6          data = (int *) ptr;
7          printf("I'm Thread %d\n", *data);
8          return (void *) data;
9  }
10 int main(int argc, char *argv[]) {
11         pthread_t hThread[4];
12         int *rvals[4];
13         for (int i = 0; i < 4; i ++)
14                 pthread_create(&hThread[i], NULL,
                       helloFunc, (void *) &i);
15         for (int i = 0; i < 4; i ++) {
16                 pthread_join(hThread[i], (void **) &rvals[
                       i]);
17                 // printf("Thread %d returns %d\n", i, *
                       rvals[i])
18         }
19         return 0;
20 }
```

```
prompt> gcc -o kai kai.c -pthread -Wall
prompt> ./kai
```

Key:

虽然说创建了 4 个 Threads 是没有问题的，但是 pthread_create(&hThread[i], NULL, helloFunc, (void *) &i) 中传的 i 的值可能不是想象中的样子。在线程执行的时候，读取的是“当前”状态下的 i 的值，当然也可能出现后面创建的线程会先执行，但是因为 i 都是“当前”状态下的，所以输出的序列一定是升序的，那可能有哪些情况呢？

0,1,2,3;0,1,2,4;0,1,3,4...3,4,4,4;4,4,4,4

注意. 出现 *4* 的情况是，循环的时候 *i* 加到 *4* 了，但是不再进入下一个循环了。

例 12. *Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.*

1. *How many LWPs are created?*
2. *At some other moment, one kernel thread blocks (waiting for I/O completion)*

   T/F?*This kernel thread makes an upcall, and the attached LWP blocks the current user thread, and schedules another user thread to run.*

   T/F?*The attached LWP also blocks. The kernel makes an upcall and then allocates a new LWP to the application.*
3. *After returning from upcall handling, how many LWPs now?*
4. *If we wish to run this application efficiently, how many LWPs are needed for this application.*

Keys:

1. 因为是两个 kernel threads，所以显然就是 2 个 LWP 啦。
2. F/T

   这两道题显然一个是对的一个是错的。那么当一个 kernel thread 阻塞之后，正确的情况是，对应的 LWP 也会阻塞。然后 kernel 就会做一个 upcall 去和 user space 说，再分配一个新的 LWP。
3. 因为添加了 1 个，所以现在是 3 个。
4. 因为有 5 个任务，所以 5 个 LWP 是最好的。

3 Concepts Organization

1. Threads: A thread is a unit of execution within a process. Multiple threads can exist within a single process, sharing resources such as memory, data, and files, which allows them to communicate more efficiently compared to separate processes.
2. Concurrency: Supports more than one task making progress.
3. Parallelism: A system can perform more than one task simultaneously.
4. Kernel thread: A thread managed and scheduled by the operating system kernel, interacting directly with hardware.
5. User threads: A thread created and managed in user space, executing tasks through system calls but not interacting directly with hardware.
6. Multithreading models:many-to-one、one-to-one、many-to-many.
7. Thread Pools: A thread pool is a collection of pre-instantiated, idle threads that are ready to perform tasks.
8. Thread cancellation: Terminate a thread before it has finished.
9. Signal handling: Process signals with default handler or user-defined handler.
10. Thread-local storage: Allows each thread to have its own data.(Not global varaible but available between functions)
11. Lightweight process: LWP is a kernel-level abstraction for managing user threads. It allows user threads to interact with the kernel, which handles scheduling and resource allocation.

第五章　CPU_Scheduling

1　In-class Contents

1.1　Warm-up

Multiprogramming 当中，每个 Process 在内存中占的空间都是独立的，且只能由自己访问，除非 process 之间设置了信息交换的通道 (message passing, shared memeory)。

开销 user thread<kernel thread<process

考点 5. 谁决定哪个 *process* 下一个运行？*CPU Scheduler(in the kernel)*

谁来实现 *context switch(save and restore)*？*Dispatcher.*

考点 6. *CPU* 调度的最小单位是 *thread*，分配资源的最小单位是 *process*（因为线程不算有独立的资源）。

1.2　Basic Concepts

1.2.1　CPU Scheduler

由 Short-term scheduler 在 ready queue 中选一个 process，并且把 CPU 分配给它，那这样的 ready queue 的排序方式就有很多种啦，比如 FCFS、SJF 等等。

那什么时候需要 CPU 做出调度决定呢？

（1） Switch from running to waiting state（一般是要做 I/O）

nonpreemptive（非抢占式）：因为是 process 自愿将 CPU 让出来的。（**主动放弃 CPU cycle**）

（2） Switch from running to ready state（例如 timer interrupt）

preemptive（抢占式）：OS 强制剥夺了你的 CPU，让其他 process 运行（**被动放弃 CPU cycle**）

（3） Switch from waiting to ready（我的 I/O 做完了，我现在要重新回到 ready queue 准备运行）

preemptive：可能不是那么明显，可以这样想：对于在 running 的 process 来说，它的 CPU cycle 是不是很可能被这个从 waiting 到 ready 的 process 给抢掉呢？（有点间接不过）

（4） Terminate

nonpreemptive：自己结束的那肯定是非抢占了。

那么抢占会有些什么问题呢？

（1） 数据共享可能出现问题

（2） 如果内核模式正在执行系统调用，比如内存分配或文件写入，这时发生抢占可能会非常危险。

（3） 在操作系统关键活动（如调度器运行、进程切换）期间如果发生中断，也要谨慎处理

1.2.2 Dispatcher

调度器模块做的事情很简单，就是把 CPU 的控制权交给 short-term scheduler 选中的 process，一共完成了这三个步骤：

（1） switch context：保存当前 process 的 machine states，并且加载下一个进程的 machine states。（**调度最核心也是最耗时间的部分**）

（2） switching to user mode：进程在 kernel mode 完成调度后，切回用户模式运行用户级代码

（3） 跳转到用户程序上次执行中断/暂停的位置继续运行。

定义 1.1. *Dispatch latency*（调度延迟）：*dispatcher* 所花的从 *stop* 一个 *Process* 到让另一个 *process* 开始 *running* 的时间，影响因素包括 *context swtich* 的开销、*CPU* 模式切换的开销以及硬件和 *OS* 的工作效率。

拓展. *Read-time OS* 可以分为 *hard real-time*（立刻做出反应）和 *soft real-time*（尽快做出反应）

1.3 Scheduling Criteria

这里，全是重点!!!!!

5 个 scheduling criteria：

（1） CPU utilization：让 CPU 越忙越好（现代计算机基本上是 100%）

（2） Throughput（吞吐量）：单位时间内完成的进程的数量

（3） Turnaround time（周转时间）：一个进程从 arrive 到 complete 的时间，包括在 ready queue 等待的时间 (waiting time)、Running 的时间和 I/O 的时间。

（4） Waiting time：process 呆在 ready queue 的**时间总和**（也就是说，如果 process 从 running 再到 waiting 去做 I/O，再回到 ready state，后面的时间是要一起加上的）

（5） Response time：从请求提交到系统第一次给出响应的时间，不管你啥时候做完，也不管你后面会不会再回到 ready queue(for time-sharing environment)。比方说，拷贝一个电影，拷贝请求提交到那个拷贝窗口出现的时间，就是 respnse time。（时间片轮转 RR 就能保证较低的响应时间）

1.4 Simple Scheduling Algorithms

这块部分很重要啊，我们一点一点来剖析：

首先，我们很不要脸，我们做五个很强的假设：

（1） Each job runs for the same amount of time. A,B,C

（2） All jobs arrive at the same time. $T_{arrival}$

（3） Once started, each job runs to completion.

（4） All jobs only use the CPU (they perform no I/O)

（5） The run-time of each job is known.

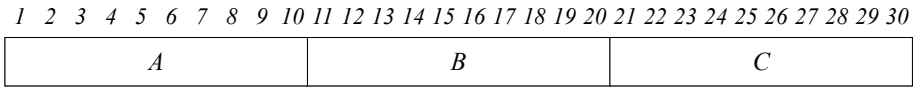
1.4.1 FCFS

First come, first served 调度算法（对于进程来说先来先服务），有的时候也被称为 FIFO(Firt in, first out)（对于存储来说是这样的）。

例 13. 假设说我们有 *3* 个进程，*A*，*B*，*C*，几乎同时到达。我们假设 *A* 比 *B* 比 *C* 早来那么一点点（只是做一个区分），同时假设每个任务要 *10s* 完成。

现在我们来算一算 *average turnaround time*。

先画一个 *Gantt Chart*：



那么就很容易算了啊，*A* 在 *10s* 结束，*B* 在 *20s* 结束，*C* 在 *30s* 结束；*average turnaround* 时间就是 *(10+20+30)/3=20*。

注意. 这个案例当中，*FCFS* 是 *Optimal solution*。

现在我们来放宽一下条件：

（1） ~~Each job runs for the same amount of time. A,B,C~~

（2） All jobs arrive at the same time. $T_{arrival}$

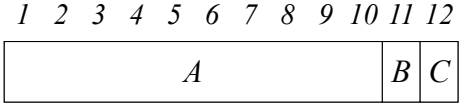
（3） Once started, each job runs to completion.

（4） All jobs only use the CPU (they perform no I/O)

（5） The run-time of each job is known.

这个情况下 FCFS 的表现怎么样呢？什么情况的 workload 会让 FCFS 表现更差呢？（其实让慢的先做就是最差的情况）

例 14. 现在 *A* 要运行 *10s*，*B* 和 *C* 只要运行 *1s*，其他都不变，我们再来算算 *turnaround time*：



turnaround time 是 *(10+11+12)/3=11*。

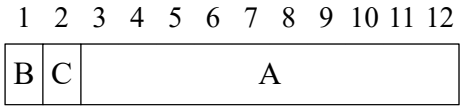
考点 7. *Convoy effect*（护航效应）：慢的 *process* 先做，然后是快的 *process* 做，这样显然是不好的。

那么其实要降低 turnaround time，一个直观的想法就是先让 B 和 C 做，A 最后做。

1.4.2 SJF

所以我们就有这样一个新的调度策略啦——Shortest Job First（也是这个情况下最优的）。

那对于上面那个例题，我们就可以再画一个新的 Gantt Chart：

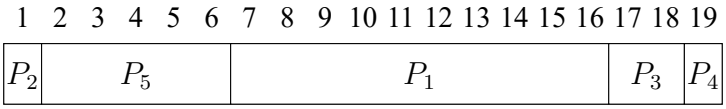


这时候的 turnaround time 就是 (12+1+2)/3=5。

考点 8. *另外，SJF 算法是一优先级调度算法的一个特例，也就是说 shorter job 有 higher priority*。

Procsss	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

我们来看上面这个表格，画一个 Gantt Chart（注意我们认为 1 比 2 的 priority 更高，以此类推）：



这个情况下是 turnaround time=(1+6+16+18+19)/5=12.

那这会有什么问题呢？

定义 1.2. *Starvation*：低 *prioriy* 的 *process* 可能永远不会执行。

注意. *只要最后这个 process 能执行，它就不会 starvation*。

那么怎么解决这个问题嘞？

定义 1.3. *Aging*：随着时间增加，*process* 的 *priority* 也会随之增加。

还有一个 SJF 的变种：HRRN(Highest Response Ratio Next)，但是吧，它不考，所以想了解的话看 PPT22 页就行。

现在我们再放宽一下条件：

（1） ~~Each job runs for the same amount of time. A,B,C~~

（2） ~~All jobs arrive at the same time. $T_{arrival}$~~

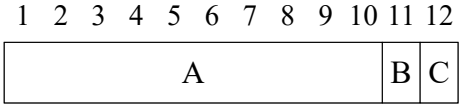
（3） Once started, each job runs to completion.

（4） All jobs only use the CPU (they perform no I/O)

（5） The run-time of each job is known.

那这个时候会不会让 SJF 表现的不那么好呢？其实是会的，我们看下面这个例子：

A 在 T=0 的时候到达，需要 10s，B 和 C 在 T=1 的时候到达，需要执行 1s。



这种情况下的 turnaround time 为 (10+(11-1)+(12-1))/3=10.33。

那么为了解决这个问题，我们需要放宽第三个条件：

（1） ~~Each job runs for the same amount of time. A,B,C~~

（2） ~~All jobs arrive at the same time. $T_{arrival}$~~

（3） ~~Once started, each job runs to completion.~~

（4） All jobs only use the CPU (they perform no I/O)

（5） The run-time of each job is known.

1.4.3 Preemptive SJF

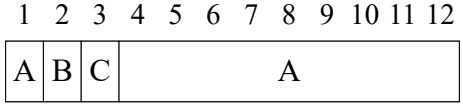
定义 1.4. *Preemptive: A job can preempt another job.*

所有的现代 schedulers 都是 preemptive 的。

现在我们来看看 Preemptive Shortest Job First(preemptive SJF) or Shortest Time-to-Completion First(STCF), of Shortest-Remaining-Time First(SRTF)

注意. 一定要关注是 *SJF* 还是 *Preemptive SJF*，很重要!!!

我们再回顾上面的那个例子：



这个时候的 average turnaround time=(12+(2-1)+(3-1))/3=5，这显然比前面的 10.33 表现要很多了对吧。

当然和 SJF 一样，会有这么一个问题：Starvation。

这个时候我们就不能只把目光放在 turnaround time 了，我们还要考虑考虑其他的指标，比如说——fairness（也就是让 response time 能够最少）。

定义 1.5. *Response time*：就是一个 *job arrive* 到它第一次被调度执行的时间。

Preemptive SJF 在 response time 做的不好-> 想到可以使用时间片轮转 Round Robin(RR) 调度策略，让每个 process 都能定期获得 CPU 的时间。

1.4.4 RR

RR 做的事情就是，它不要求一定要把一个任务完成，每个 process 轮流使用 CPU cycle，RR 有的时候被称为 time-slicing（时间分片）。

例 15. *Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds, the length of time slice is 1 second.*

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*

这种情况下的 *response time* 表现很好，是 *(0+1+2)/3=1*，但是也会有一个问题，就是频繁的上下文切换会导致开销有点高。所以这也就说明了，*time slice* 不是越小越好。

那么 RR 在 turnaround time 方面做的是不是就没那么好了？确实是这样，和 FCFS 和 SJF 相比，turnaround time 确实会长一些，那这只能说是 response time 和 turnaround time 的 trade-off 了。

1.4.5 Incorporating I/O

现在我们再放开一下条件，加一点 I/O 进去玩玩：

（1） ~~Each job runs for the same amount of time. A,B,C~~

（2） ~~All jobs arrive at the same time.~~ $T_{arrival}$

（3） ~~Once started, each job runs to completion.~~

（4） ~~All jobs only use the CPU (they perform no I/O)~~

（5） The run-time of each job is known.

例 16. *Assume two jobs, A and B. A runs for 1 second and then issues an I/O request which also takes 1 second, B simply uses the CPU for 5 seconds and performs no I/O.*

那么因为有 *I/O Requests*，我们可以把任务 *B* 拆成子任务，*Gantt Chart* 如下图所示：

	1	2	3	4	5	6	7	8	9	10	11
CPU	A	B	A	B	A	B	A	B	A	B	A
I/O		A		A		A		A		A	

最后呢，我们放宽所有条件：

（1） ~~Each job runs for the same amount of time. A,B,C~~

（2） ~~All jobs arrive at the same time.~~ $\sout{T_{arrival}}$

（3） ~~Once started, each job runs to completion.~~

（4） ~~All jobs only use the CPU (they perform no I/O)~~

（5） ~~The run-time of each job is known.~~

那么问题来了，我们没有所有信息，怎么去进行调度呢？我们如何做到 fairness 和 performance 的 trade-off 呢？

如果我们用的是 SJF 调度算法，那我们需要去“猜”下一个进程的 CPU burst 需要多久，然后选择最短的那个去执行。那么计算的公式是 $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$，然后 α 的取值一般为 0.5.

1.4.6 Multilevel Queue

现在我们来讲一下多级队列调度，具体是一种混合型调度策略。

说白了，ready queue 会被分成两个部分，一个是 foreground（前台，主要做交互式任务，例如移动鼠标），另一个部分是 background（后台，面向批处理任务，比如说大型计算）。

注意. 每个 *process* 会被永久分配到一个”队列”，要么是 *foreground* 里面，要么是 *background*，不能跨”队列”移动。（但是！**要注意一个事情是，本质上我们只有这样一个 *ready queue***）

每个“队列”有它自己的调度算法：对于 foreground 来说用的是 RR 调度算法，它更加公平一些，对于 background 来说用的是 FCFS 调度算法。

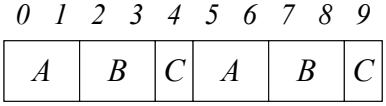
那么具体我们会怎么做调度呢？

（1） Fixed priority scheduling：先服务 foreground“队列”中的 process，然后服务 background，那么显然这可能会有 starvation 的可能。

（2） Time slice：为 foreground 和 background 分别分配 CPU 时间，比如说给 foreground 分配 80%，调度算法是 RR，给 background 分配 20%，调度算法是 FCFS。

例 17. 我们可以举一个小小的例子，以 *time slice* 为例：

假设有三个进程 *A*、*B*、*C*（都需要 *4s* 才能完成），*A*、*B* 在 *foreground queue* 当中，*RR* 的 *quantum=2*，分配 *80%* 的时间；*C* 在 *background queue* 当中，分配 *20%* 的时间。



1.5 Multilevel Feedback Queue

我们希望找一个算法，能够在不完全了解每个进程的执行情况的时候也能完成调度，这时候，诶嘿，我们就想到了用 Multi-level Feedback Queue(MLFQ)，目的是 optimize turnaround time 和 minimize response time。

当然，在这之前，我们先回顾一下 MLQ，它有多个不同的“队列”，并且队列间有不同的优先级。

对于 MLQ 来说有两个 basic rules:

（1） 如果 A 的优先级比 B 的优先级高，则 A 运行

（2） 如果 A 和 B 的优先级相同，那么 A 和 B 都用 RR 调度算法

在 MLQ 里，process 的 priority 是固定的，但这不是我们希望的样子，我们不希望看到 starvation，所以我们希望能够调整 process 的 priority，于是我们需要一个”feedback”，比如说如果一个进程占用 CPU 很久，它会被系统降级到低优先级队列。

注意. *MLFQ* 是行动驱动的调度策略，通过观察一个进程“是否经常使用完时间片”、“是否经常被抢占”等行为来调整其优先级。

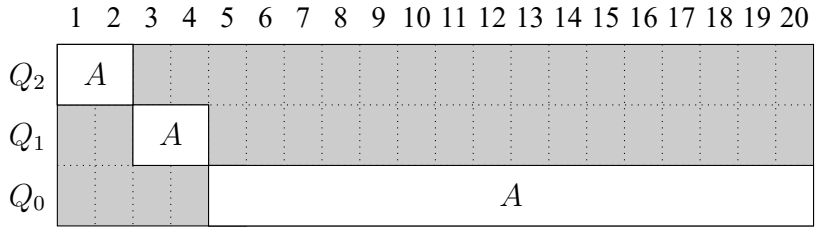
我们来看看一个新来的任务，MLFQ 会怎么进行调度：

首先，一个新的任务来的时候，scheduler 不知道它是个 long job 还是 short job，所以 scheduler 就假定它是一个 short job，并给它很高的优先级。同时在运行过程中观察这个 job 的行为，如果它频繁使用完时间片，那么说明它是个长任务，就会慢慢降低它的优先级；如果很快就完成了，那很好，说明它是一个短任务，也值得很高的优先级。

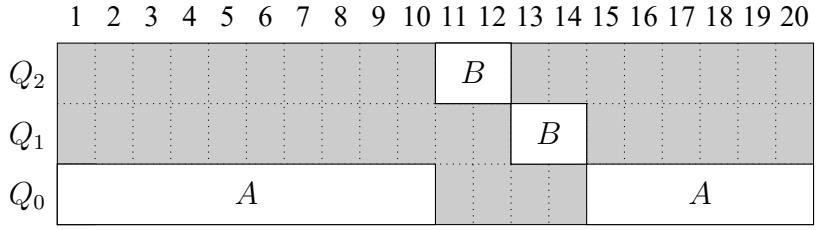
我们用两张图直观地来看一下 Priority 的变化：

注意. 再次注意，这实际上是一个 *Queue*！！！

对于一个 long-running job A:



那在这之后又来了一个短任务 B 呢？



好，那么好，我们现在来看看 MLFQ 的 Rules（MLQ+Feedback）:

（1） 如果 A 的优先级比 B 高，那么 A 先跑

（2） 如果 A 和 B 的优先级一样，那么 A 和 B 都用 RR 跑

（3） 如果有一个任务进到系统，那么它会被放在最高的优先级

（4） 如果一个任务在跑的时候**用完了一整个 time slice**，那么它的 priority 会降低

（5） 如果一个任务在 time slice 还没用完时自己放弃 CPU，那么它的 prioriy 就不变

那么这个版本的 MLFQ 的表现已经达到最好了吗？

显然它是有一些缺陷的，比如说：

（1） Starvation：很好理解啊，如果说一直有新的任务来，那么它们就一直是最高的优先级，那么低优先级的就永远无法执行。

（2） Gaming the scheduler: 也很好理解，就是在卡 bug 嘛，我就只用 99% 的 time slice 然后**主动放弃 CPU（可能是去做 I/O 了）**，这样我的 priority 就不会掉下去。

（3） Changeable program behaviors:process 最开始大量计算（CPU 密集），之后等待用户输入选项（变成交互型），但因为它已经被“打入冷宫”，所以响应变慢。（**所以，还应该有 priority 上升的策略**）

那么怎么解决这些问题呢？

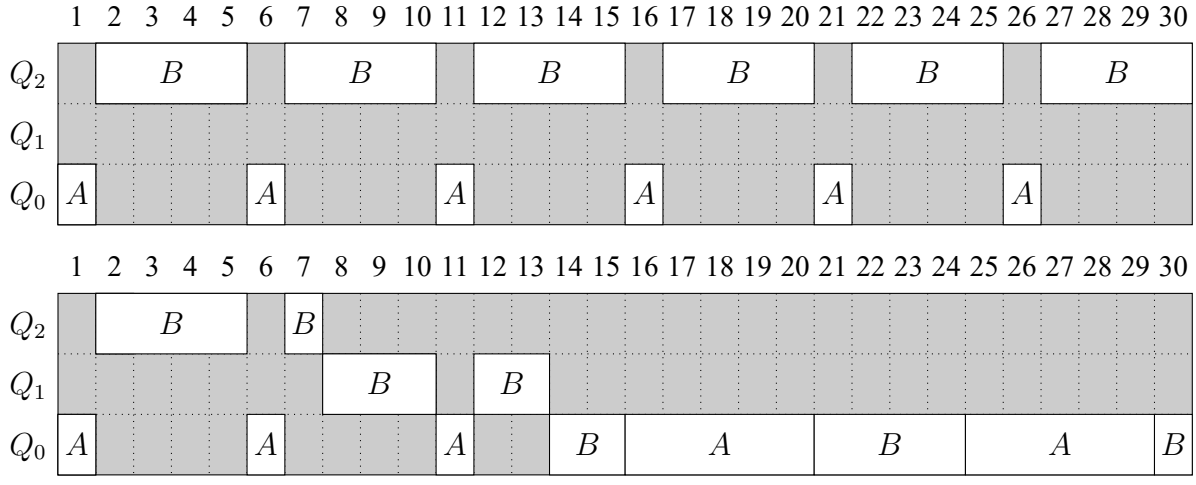
其实也很简单，就是过 S 秒之后，把系统中的所有 jobs 都变成 top priority，给所有任务一个机会，这可以解决什么问题呢？

一个是 processes 不会 starve；另一个是如果某个一开始是 CPU 密集型任务，后来行为变成了交互型（I/O 密集），那它也能被重新识别出来并享受交互任务的待遇。

同时，我们为了不让 process 卡 bug，我们在改变优先级的时候，用以下策略：

现在我们不管一个 process 分成多少次用分配给它的 CPU time，只要它用完了，就降级。

这样可以解决一个什么问题呢？我们看下面的 Gantt Chart，上面是可以卡 bug 的情况，下面是更新后的情况，假设 time slice=5



我们可以发现，更新之后，B 就不能卡 bug 了，它在用完了自己的 5 秒时间片之后，就降级了。

当然，最后还有一点，对于不同等级的”queue”，在做 RR 的时候，分配到是 time slice 可以是不一样的，等级越低，分配的 time slice 长度越长（因为等级低的大多是 CPU-Bound job 嘛）。

那么现在我们可以看看 MLFQ 的最终的五个 Rules:

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

1.6 Lottery Scheduling

这个部分不考!!!

现在呢，我们想要改变我们的目标，我们不再要求最快的 turnaround time 或者是 response time，我们只是想要保证每个 job 都能得到一定时间的 CPU cycle。那这个时候传统的调度算法就不是很好了，所以我们这里采用的是 Lottery Scheduling，在这种情况下，我们可以比方说给 Linux 分配 75% 的 CPU cycle，然后给 Windows25% 的 CPU cycle。

定义 1.6. *Proportional Share*：*Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.*（这个很重要，能够用来管理不同的虚拟机或者是操作系统）

那么要实现 Lottery Scheduling 其实也很简单，每个 process 都有一定数量的 lottery，系统抽到谁，谁就获得 CPU Cycle.

在抽奖的时候也会有下面三个特性：

（1） Ticket currency（票据货币）：用户或任务组拥有一批票，可以按自定义方式划分这些票给不同的子进程。

（2） Ticket transfer（票据转移）：很好玩吧，process 之间还可以暂时借 tickets 噢。

（3） Ticket inflation（票据膨胀）：process 可以暂时增加或者减少它的票。

那么，这个随机算法怎么去衡量它的公平性呢？我们就引入了这样一个公平性指标 U：它表示的是，对于两个有一样彩票，并且运行时间相同的 Process，它们实际上第一个任务完成的时间/第二个任务完成的时间的比值。那么我们拍拍脑袋就知道，U=1 的时候说明这个算法表现是最好的。

我们可以发现 Lottery Scheduling 是一个概率性调度算法，只有说 long-time jobs 才有可能体现公平性，没法保证 U=1（有点像大数定律）；那么既然有概率性的调度算法，那肯定也会有确定性的调度算法，也就是能够保证绝对的公平（U=1）。有一个这样的算法叫 Stride scheduling，每个 job 有一个固定的 stride（把它理解成和 tickets 成反比的东西就可以），然后每个 job 还有一个用来记录的 pass value，每次这个 process run 的时候，它的 process value 就会增加。最后呢，在进行调度的时候，每次都选择 pass 最小的那个 job 去工作。

我们来看下面的这个例子：

例 18. *Suppose three processes (A, B and C), with stride values of 100, 200 and 40, and all with pass values initially at 0.*

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	**Who Runs?**
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	···

那我问你，这个确定性的调度算法不是挺好的嘛，为什么我们还要用有随机性的 Lottery Scheduling 呢？

其实是这样的，一方面，每次在调度的时候都要去 check 一下”pass value”，再决定哪个 job 下一个运行；还有一个更加关键的问题，如果有一个新的 job 进来，它的 pass value 该是多少呢？是 0 吗？如果是 0 的话，它不就会独占 CPU 了吗？

1.7 Thread Scheduling

我们首先回顾一下，process 是 CPU 分配资源的最小单位，而 thread 是 CPU 调度的最小单位。

同时再回顾一下 user-level thread 和 kernel-level thread 的区别：

用户级线程（ULT）：线程由用户空间的线程库管理（如 pthread 库），操作系统内核并不知道这些线程的存在。而内核级线程（KLT）：由操作系统内核直接管理，调度器直接对它们进行调度。

同时，在支持线程的操作系统当中，调度的是 kernel-level threads，而不是 processes，这个时候内核线程的调度属于 system-contention scope，也就是说不管你是哪个进程的，所有的 kernel threads 一起争用 CPU cycles。

但是对于 user thread 就不一样了，对于 OS 来说，它只能看到 kernel threads 和 LWP，而 user thread 的调度是 user mode scheduler 来完成的（这个调度器在 user space 当中，其代码就是普通用户程序代码的一部分，由 thread library 实现）。具体步骤是 OS 把时间片分给 kernel thread/LWP，然后 kernel thread 来反向激活 user mode scheduler，然后再有 user mode scheduler 决定哪个 user thread 运行。

1.8 Multiple-Processor Scheduling

我们之前讲的东西都是"one queue"，那如果我们有 multiple processors/CPUs 呢？

我们都知道，Cache 这个东西的设计，是依赖局部性的，包括时间局部性和空间局部性。那么在 multi-processor 架构当中呢，每个 processor 都会有一个自己的 Cache，这时候就有一个叫做 Cache Affinity 的概念：

定义 1.7. *Cache Affinity(Processor Affinity)*：一个 *process*，如果在一个 *CPU* 上运行，那么让它继续在这个核心上运行，可以重用缓存数据，从而提高性能。从根本上来说，在执行一个 *process* 的时候，需要去 *memeory* 里面找它的指令并且执行，那么如果 *process* 在一个 *CPU* 里面执行过了，那么它的一些指令和数据很可能就会存在 *Cache* 里面，那么访问 *Cache* 肯定会比访问 *Memory* 快，也就是说，在运行过这个 *process* 的 *CPU* 运行肯定比一个“新”的 *CPU* 要快。

那么针对 Multiple-Processor Scheduling，有以下这两个算法：

1.8.1 Asymmetric Multiprocessing(ASMP)

在这个算法当中，所有的 CPU 共用一个 ready queue，我们结合一个例子来看：

如果说，我们一开始的 CPU 运行情况如下所示：

CPU_0	A	E	D	C	B
CPU_1	B	A	E	D	C
CPU_2	C	B	A	E	D
CPU_3	D	C	B	A	E

我们认为，执行的早的 job 和对应的 CPU 更有亲和力，所以 A 对 CPU0 最有亲和力，那么最后 4 个 CPU 的运行情况可能是：

CPU_0	A	E	A		
CPU_1	B		E	B	
CPU_2	C			E	C
CPU_3	D				E

但是我们会看到 job E 就会窜来窜去，就没有很好地利用到 Cache 的局部性。

1.8.2 Symmetric Multiprocessing

这个时候呢，对于每个 CPU 会有一个自己的 ready queue，那我们就可以把任务分给几个 queues。比方说每个 queue 都按照 RR 来调度，Queue0 负责完成任务 A 和 C，Queue1 负责完成任务 B 和任务 D，那 Gantt Chart 会是下面这样的：

CPU_0	A	C	A	C	A	C	B
CPU_1	B	D	B	D	B	D	

但是这又有一个问题（虽然上面的图已经解决了这个问题）：就是我们不知道每个 job 的 length 或者说 workload，这个时候只要做一个 migration 就可以了，在上面的例子中就是把 B 移到 CPU0 里面去完成，解决问题！

1.9 Read-Time CPU Scheduling

再坚持一下！就最后一点点了！

在实时 CPU 调度当中，我们也分成两种情况:

（1） Soft real-time systems: 对关键实时任务的调度不提供绝对时间保证，允许任务偶尔错过截止时间，系统整体仍能正常运行。（一般这个任务可以延时高一点）

（2） Hard real-time systems: 任务必须在绝对截止时间前完成，否则会导致严重后果。（这种情况下任务都可能是非常重要的）

同时也有两个会影响表现的 latencies:

（1） Interrupt Latency: 从 interrupt arrive 到开始 service 这个 Interrupt 的延时，和 response time 很相似

（2） Dispatch Latency:scheduling dispatcher 从把当前 process 拿下来，到换成另一个的时间（context switch 等花的时间）。

同时，dispatch latency 还要考虑一个冲突问题，就是低优先级进程正在执行内核态代码（如操作系统内核函数），高优先级进程无法立即抢占，需等待内核态代码执行完毕。那么怎么解决呢？当低优先级进程阻塞高优先级进程时，临时提升低优先级进程的优先级，使其尽快释放资源。（很像 priority inversion）

2 Exercise

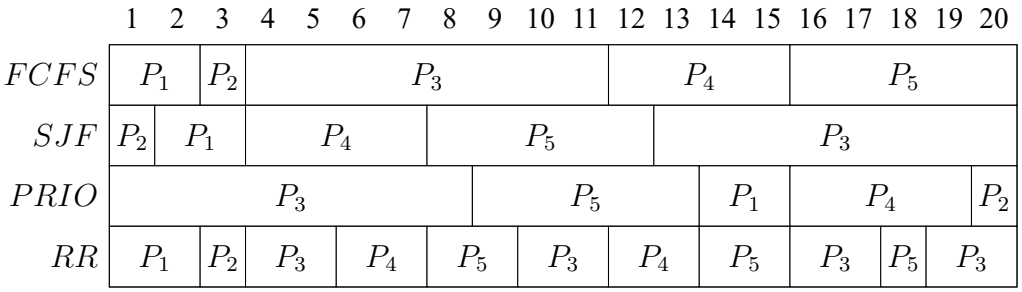
例 19. *Consider the following set of processes, with the length of the CPU burst given in milliseconds:*（一个小 *tip*：如果 *priority* 一样，一般谁先来，谁的 *priority* 相对高一点点）

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order of P1; P2; P3; P4; P5, all at time 0.

1. *Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).*
2. *What is the average turnaround time for each of the scheduling algorithms in part 1?*
3. *What is the waiting time of each process for each of these scheduling algorithms?*
4. *Which of the algorithms results in the minimum average waiting time (over all processes)?*

Keys:
Q1:



Q2:

	FCFS	SJF	PRIO	RR
$T_{turnaround}$ (ms)	$\frac{2+3+11+15+20}{5}$ =10.2	$\frac{1+3+7+12+20}{5}$ =8.6	$\frac{8+13+15+19+20}{5}$ =15	$\frac{2+3+20+13+18}{5}$ =11.2

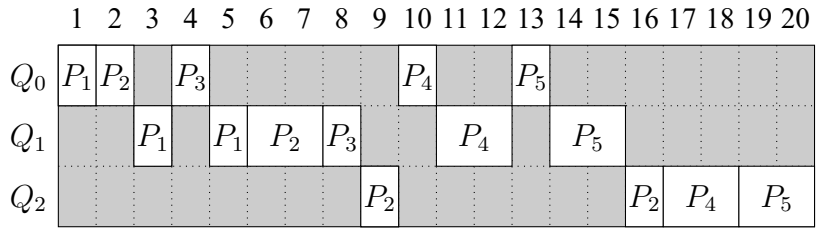
Q3 & Q4:

		P_1	P_2	P_3	P_4	P_5	Avg	
$T_{waiting}$ (ms)	FCFS	0	2	3	11	15	6.2	
	SJF	1	0	12	3	7	4.6	Minimum
	PRIO	13	19	0	15	8	11	
	RR	0	2	12	9	13	7.2	

例 20. *Given the table below showing the process information of a system, based on multilevel feedback queuing scheduling scheme. Suppose there are five levels in the system and within each level the FCFS scheduling is used. If a process is preempted, it will wait for being scheduled at the first place in the FCFS queue. Draw a Gantt chart to show the time of CPU allocated to each process until all processes are finished using time quantum $q = 2^i$, (where q = time allocated for a process to run in its turn, and i ranges from 0 to 4 indicating the i^{th} level queue).*

Process	Arrival Time	Service Time
P_1	0	3
P_2	1	5
P_3	3	2
P_4	9	5
P_5	12	5

Keys:



3 Concepts Organization

1. CPU burst: A period during which a process is executing instructions on the CPU without performing any I/O operations.

2. I/O burst: A period during which a process is waiting for or performing input/output operations, such as reading from disk, writing to a file, or interacting with a device. During this time, the CPU is not used by the process.

3. Scheduling timing: The moments or events at which the operating system decides to evaluate and possibly change the currently running process or thread.

4. Scheduler: Make the decision of which process runs next.

5. Dispatcher: It gives control of the CPU to the process selected by the short-term scheduler and performs the context switch.

6. Scheduling criteria: CPU utilization, Throughput, turnaround time, waiting time, response time.

7. FCFS: First come, First served.

8. (Preemptive)SJF: Shortest job first, then the next shortest, which is a special case of the general prioriy-scheduling algorithm.(Preemptive SJF:decide whether a job can preempt another job)

9. Priority: A numerical value assigned to a process to determine the order in which processes are selected for execution by the CPU scheduler. A process with higher priority is scheduled before processes with lower priority.

10. RR: Instead of running jobs to completion, RR runs a job for a time slice(or a scheduling quantum) and then switches to the next job in the run queue.

11. MLQ: Ready queue is partitioned into separate queues, such as foreground and background. Besides, process is permanently in a given queue, and each queue has its own scheduling algorithm.

12. MLFQ: A dynamic scheduling algorithm where processes can move between multiple priority queues based on their behavior and execution history. Unlike Multi-level Queue (MLQ), MLFQ allows a process to be promoted or demoted between queues, enabling better responsiveness and fairness for diverse workloads.

13. Gantt chart: A visual representation of the schedule of processes over time in a CPU scheduling scenario. It shows the order and duration in which processes are executed by the CPU.

14. Process/system-contention scope(two completely different concepts): Competiton of CPU cycle is within the process(all threads in system).

15. Symmetric multiprocessing: One queue per CPU, the system divides the jobs to several queues and each queue will likely follow a particular scheduling discipline, such as Round Robin.

16. Processor (cache) affinity: A process, when runs on a particular CPU, builds up a fair bit of state in the caches of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU.

第六章　Process_Synchronization

1　In-class Contents

1.1　Warm-up

我们先来看一个程序。

```
1  /* thread.c */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <common.h>
5
6  volatile int counter = 0;
7  int loops;
8
9  void *worker(void *arg) {
10  int i;
11  for (i = 0; i < loops; i++) {
12  counter++;
13  // 问题在于，多个线程同时访问counter，造成竞争，可能导致
       counter的值小于预期值。
14  }
15  return NULL;
16 }
17
18 int main(int argc, char *argv[]) {
19  if (argc != 2) {
20  fprintf(stderr, "usage: threads <value>\n");
21  exit(1);
22  }
23  loops = atoi(argv[1]);
```

```
24  pthread_t p1, p2;
25  printf("Initial value : %d\n", counter);
26
27  Pthread_create(&p1, NULL, worker, NULL);
28  Pthread_create(&p2, NULL, worker, NULL);
29  Pthread_join(p1, NULL);
30  Pthread_join(p2, NULL);
31  printf("Final value : %d\n", counter);
32  return 0;
33 }
```

```
1  prompt> gcc -o thread thread.c -Wall -pthread
2  prompt> ./thread 1000
3  Initial value : 0
4  Final value : 2000
5
6  prompt> ./thread 100000
7  Initial value : 0
8  Final value : 143012
9  prompt> ./thread 100000
10 Initial value : 0
11 Final value : 137298
```

从上面这个例子可以看出，loop 值增加的时候，Final value 的值会比两倍的 loop 小。这背后的原因是出现了并发错误（concurrency）。

1.2 Background

我们还是看上面的例子，问题出在"counter++" 这行代码，虽然它是一行，但是在汇编当中它是三行代码，所以在这中间如果出现 interrupt，那就会导致结果的 inconsisitency。

注意. *不完全是因为 multi-processors 的问题，只有一个 CPU 也会出错。*

这个事情的核心在于：我们的调度被 kernel control 而不是被 user control，kernel 不知道 user 想要的结果是什么样的。

那么其实有一种方式可以解决这个问题，如果我们有一个指令可以把自加操作合并成一个原子操作，那就没事了，但是这样做的代价是指令的执行会很慢，例如，将自加指令合并为"memory-add 0x8049a1c, $0x1"。

定义 1.1. *Race condition*(竞争条件): **最终的输出是不确定的。**多个 *processes(threads)* 同时竞争和控制同样的数据，那么这个数据的结果会取决于 *processes* 竞争这个数据的顺序。

1.3 The Critical-Section Problem

我们首先来对这个问题做一个定义：

假设我们的系统有 n 个 processes：$P_0, P_1, \cdots, P_{n-1}$。每个 Process 当中有这样几个代码部分 (segments)：entry、critical section（涉及变量的改变、写文件等）、exit、remain。那么理论上，我们在 critical section 运行 process 的时候，其他的 process 不应该在自己的 critical section 运行。

定义 1.2. *Critical Section:* 有可能导致不确定输出 *(race condition)* 的代码片段，在前面的例子中就是*"count++"*。

定义 1.3. *Critical section problem:* 我们做一些规定来避免 *race condition*。

具体来说，还是以上面的 count++ 为例，它可以拆成三条指令，我们希望这三条指令要么一起运行，要么都不运行。所以每个 process 在 entry section 的时候要询问是否能够进入 critical section，同时在执行完 critical section 的代码之后进入 exit section（告诉别的进程我用完这个 critical section 了，其他 process 可以进到它们的 critical section 了），最后是 remainder section（在这里做一些其他的事情）。

现在我们来看一个例子，看看它能不能解决 critical section problem。假设我们有两个 process，P_i 和 P_j。

```
// Pi
do {
    while (turn == j);
        critical section
    turn = j;
        remainder section
} while (true);
```

```
// Pj
do {
    while (turn == i);
        critical section
    turn = i;
        remainder section
} while (true);
```

看上去没有任何问题，能够满足每次只有一个 process 进到 critical section(mutual exclusion)，也满足有界等待 (bounded waiting)，因为 process 只会等待一次就能进入 critical section。但是问题在于**如果这两个 process 一个不想运行了，那另一个肯定也运行不了了 (progress)**。

1.3.1 Solution to Critical-Section Problem

下面的内容 very important!

现在我们来讲讲三个判定一个 solution 是否 correct 的标准：

（1） Mutual Exclusion（互斥）: 很好理解，一个 process 在执行 critical section 代码的时候，别的 process 不给执行。

（2） Progress（前进、空闲让进）: 保证当 critical section 没人执行的时候，如果有 process 想要执行 critical section，就保证能够执行，不能大家都堵死，**也就是不能出现选择一个 process 进入 critical section 的这个动作被无限延迟，等待的时间是有限的**。

（3） Bounded Waiting（有界等待）: 从一个进程做出请求进入 critical section 到这个请求被满足为止，其他 process 进到 critical section 的次数有限。**保证等待的次数有限，自己不会被饿死**。注意在判断这个的时候，只要找到一个调度序列，也就是代码的某一行被 interrupt，就能导致一个 process 进不去就行。

同时，在设计内核的时候，有两种情况，一种是 preemptive 的，可以强制某个进程释放 CPU，避免出现不想主动结束或者是阻塞的问题，但会出现 race condition 的情况；还有一种是 non-preemptive 的，它虽然不会有 race condition，但是它的响应能力太差。在实际当中，我们用的都是抢占式内核。

1.4 Mutex Locks

那么在解决同步问题的时候，我们需要的是 Hardware synchronization primitives（硬件支持同步语句）来实现将一串代码变成“原子操作”，但实际上一个 lock 本质就是一个变量。

在这里我们有两个函数 lock() 和 unlock()。

如果没人占用这把锁，调用 lock() 就可以获得这个锁，别人进不来，别人调用 lock() 也没用。同时呢，如果锁当前占有者 unlock() 了，那这下锁就又可以被使用了，但是下一个谁进去是没有保障的，取决于其他东西（比如说有这样一个队列），和锁本身无关。

注意. *lock()* 和 *unlock()* 一般是非原子性的，取决于 *OS* 的实现，*test and set* 这种就是原子性的。

那么我们怎么构建一个锁呢？有下面两类方法（都是实现 lock 的功能）：

（1） 第一类是没 special hardware support 的情况：Dekker's and Peterson's Algorithms、Lamport's Bakery Algorithm。

（2） 第二类是有 hardware support 的情况：Controlling Interrupts、The test-and-set instruction、The compare-and-swap instruction 等等。

1.4.1 Controlling Interrupts

一种很简单的想法就是，直接不允许 interrupt 就行：

```
1 void lock() {
2         DisableInterrupts();
3 }
4 void unlock() {
5         EnableInterrupts();
6 }
```

这里的 DisableInterrupts 和 EnableInterrupts 是两个 special hardware instructions。那么这是一个正确的解决 critical section problem 的思路吗？

（1） Mutual exclusion: 显然是的。

（2） Progress: 只要 enableinterrupt，就可以有 process 进去。

（3） Bounded waiting: 不满足，比如说有两个 process A 和 B，万一每次都是 A 抢到了 critical section 呢？没法保证 B 有机会执行。

它很简单、很快、也很方便，但是不是一个正确的解法，也会有下面这些问题（了解）：

（1） 关闭中断是一个特权操作，如果任何线程都能关闭中断，那这是很危险的，一个 process 占用了一个锁之后不放了，一直占用 CPU 你不傻了吗。

（2） diableinterrupts 只能在当前的 processor 使用，而不同的 CPU 处理自己的 interrupt，所以这么做没法阻止其他 CPU interrupt。

（3） 把中断关掉了，CPU 就没办法及时处理重要的中断请求了。

（4） 效率低。

1.4.2 Peterson's Solution

Important! 一定要知道这个方法的所有 details！虽然现代 OS 不会用，但是考试会考，不需要 hardware support。

注意. 这是一个两进程互斥问题的解决方案。

当然在这个方案里面有一个假设，也就是假设"load" 和"store" 的操作是原子操作，也就是它们不能被中断（从代码上看就是 turn=j 的这种赋值是原子操作）。

这两个 processes 共享两个变量：int turn 和 Boolean flag[2]。其中 turn 标识现在轮到谁进入 critical section，flag 数组则用来表示一个进程是否准备好进到 critical section，比方说 flag[i]=true 就代表着 P_i 已经准备好进到 critical section 了。

初始化的时候注意 flag[i] 和 flag[j] 都为 false。

注意. 三个变量已经是最简单的情况了，如果变量的数量少于三个，一定出错！

下面的内容，very very very important!

Peterson's Solution 的核心思想就是：首先我想进，但是我先问问你想不想进，如果你不想进，那我就进去了，如果你想进，那你先来，最后记得告诉我你不想进了，我才好进去。

从代码上来看：

```
// Pi的代码
do {
 flag[i] = true;
 turn = j;
 while (flag[j] && turn == j);
 critical section
 flag[i] = false;
 remainder section
} while (true);
```

```
// Pj的代码
do {
 flag[j] = true;
 turn = i;
 while (flag[i] && turn == i);
 critical section
 flag[j] = false;
 remainder section
} while (true);
```

好，那么好，我们现在来证明一下我们这样做是一个正确的解法，也就是满足那三个标准：

（1） Mutual Exclusion: **证明方法：假设一个 Process 进去了，看另一个 process 能不能进去。**假设 P_i 进去了，那么意味着要么 flag[j]=false，要么 turn=i。如果 turn=i，那么在 P_j 里面，肯定已经执行过 flag[j]=true，所以 P_j 要卡在 while 那个死循环那里。如果说 flag[j]=false，那对于 P_j，它一句都还没执行，肯定进不到 critical section。所以满足互斥。

（2） Progress: **证明方法：分情况讨论，一种情况是两种都从零开始，就是没人进去的情况；另一种情况是假设一个进程刚退出，看另一个进程能不能进去；还可以考虑一个进程执行完之后再也不动的情况。**如果两个进程都从零开始执行，因为 turn 只会有一个值，所以肯定有一个 Process 能进去。如果 P_i 刚运行完出来了，这个时候就会有 flag[i]=false，所以这个时候 P_j 就能进去，当然如果说 P_i 又绕了一圈回去，运行到 flag[i]=true 和 turn=j，这个时候 P_j 还是能够进去。

（3） Bounded Waiting: 我们看如果 P_i 进了一次，那下一次只要 P_j 运行到死循环的那个地方，那就是 P_j 进去。

考点 9. 做题的时候有一个小技巧，不满足互斥情况的往往满足前进；不满足前进情况的往往满足互斥和有界等待，当然也会有例外，比如 *bakery* 算法。

下面有几个例子，就不在这里赘述了，直接看 PPT 就好，但记得一定要看！

1.4.3 Bakery Algorithm

那么现在问题也就出现了，我们之前的算法是针对两个 process 来的，如果我现在有 n 个 processes 呢？

考点 10. 什么是 *Bakery Algorithm?*

简单来说就是：顾客进入店里 → 进程想进入临界区；他们从门口的取号机拿到一个号码（比如 *1, 2, 3...*）；所有人都按号码从小到大排队，谁号小谁先服务；正在被服务的人（临界区）之后，才轮到下一个；出来后“释放号码” → 表示可以让下一个进程继续。

但是这里可能会有一个问题，就是可能产生两个进程取的号是一样的情况，那这个时候我们决定下一个服务谁的时候就不能仅仅根据号来确定，而是要根据一个 pair(number, index of the process)，如果 number 相同的时候，谁的 index 小，就先服务谁。同时，number 的数组一定是递增的，虽然它可能不是严格递增的，例如：1,2,3,3,3,4,5...

那么在这个算法当中，我们有两个 shared data：

```
boolean choosing[n];    // initialized to false
int number[n];  // initialized to 0
```

```
do {
        choosing[i] = true;     // 表示我当前在取号，还没有取完
        number[i] = max(number[0], number[1], ···, number [n - 1]) + 1;
        // 看看别人都拿了哪些号，然后在别人拿的号的最大值的基础上加1，确保号是递增的，解决bounded waiting的问题。
        // 但是问题就在于可能i和j抢的是一样的号，因为这句代码不是原子操作
        choosing[i] = false; // 取号完成
        // 有这一步的意义就是，告诉别的进程，我现在的number为0到底是因为我本身就不想取号(choosing[i]=false)还是说我现在还在取号，你们等等我(choosing[i]=true)。
        for (j = 0; j < n; j ++) {
                while (choosing[j]);  // 有人还没取完，就得等等
                while ((number[j] != 0) && ((number[j], j) < (number[i], i)));
                // number[j]要是为0，那就说明这个进程不想进去；同时后面一句话表示的是号小的先进去。
        }
                critical section
        number[i] = 0;
                remainder section
} while (true);
```

那么 choosing[] 这个数组有什么用呢？

用来表示 process 是否在取号的过程中，如果没有 choosing[] 这个数组，就无法满足 mutual exclution。

为什么呢？比方说 process i 理论上要拿一个 3 的号，但是它一直卡在第三行取号的地方，刚做完 max()，号还没打印出来，但是这个时候 process j(i<j) 也进去了，

并且也打印了一个 3 的号，它走的比较快，发现自己是最小的号，就进去了，然后 process i 这时候把号打印出来，发现自己也是最小的（因为 number i=number j，但是 i<j），所以 process i 也进去了，就不满足 mutual exclusion 了。

注意. 不过 *Progress* 和 *Bounded waiting* 都是满足的！

但是呢，现在的主流还是硬件锁（LOL）。

1.4.4 Test And Set

我们先看看这段代码有什么问题？

```
typedef struct __lock_t { int flags; } lock_t;

void init(lock_t *mutex) {
        mutex->flag = 0;          // 0 -> lock is available,
            1 -> held
}

void lock(lock_t *mutex) {
        while (mutex->flag == 1)          //TEST the flag
                ;                 // spin-wait (do nothing)
        mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
        mutex->flag = 0;
}
```

可以发现这段代码不满足 mutual exclusion 和 bounded waiting，问题出在 while(mutex->flag==1) 这句话。所以这个时候我们就想到了用 hardware support 来完成这个事情。

```
int TestAndSet(int *old_ptr, int new) {
        int old = *old_ptr;
        *old_ptr = new;
        return old;
}
// 这里相当于在做一件事情，就是test old_ptr的值，如果是我
   要的值，就把它改掉，改成new的值。
```

这个时候我们再改一下前面我们说出错的地方，改成下面这个样子：

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *lock) {
        lock->flag = 0; // 0 -> available, 1 -> held
}

void lock(lock_t *lock) {
        while (TestAndSet(&lock->flag, 1) == 1);
        // spin-waiting
}

void unlock(lock_t *lock) {
        lock->flag = 0;
}
```

但是啊但是，这样的修改只能解决 mutual exclusion 的问题，也就是说，现在满足了 mutual exclusion 和 progress，但是 bounded waiting 的问题还没有得到解决。

考点 11. *Mutex lock 考察的天花板就是如何使用 test and set 满足 bounded waiting 的问题。*

1.4.5 Compare And Swap

现在我们来讲讲 compare and swap，其实它和 test and set 很像，compare 就是在 test 的过程，swap 就是 set 的另一种形式。

```
int CompareAndSwap(int *ptr, int expected, int new) {
        int actual = *ptr;
        if (actual == expected)
                *ptr = new;
        return actual;
}
```

和 test and set 的代码一样，可以这样来实现 mutual exclusion:

```
void lock(lock_t *lock) {
        while (CompareAndSwap(&lock->flag, 0, 1) == 1); //
            spin-waiting
}
```

同样的，这个方法也是满足 mutual exclusion、progress，但是不满足 bounded waiting。

但是！有一个不一样的地方：compare and swap 它更强大，它可以用来实现一个没有锁的同步！

这里补充一个内容，就是说我们看到代码里面有 spin-waiting，那么对于一个需要 spin-waiting 的 process 来说，就算时间片分给它了，它也干不了啥，就只能傻等着，这样的 performance 是比较差的。

我们来看看 compare and swap 是如何实现 lock-free synchronization 的：

首先，给一个 compare and swap 的代码，我们希望是实现无锁的加法的功能：

```
int CompareAndSwap(int *ptr, int expected, int new) {
        int actual = *ptr;
        if (actual == expected) {
         *ptr = new;
         return 1;
        }
        return 0;
}
```

然后是执行加法的函数：

```
void AtomicIncrement(int *counter, int amount) {
        do {
                int old = *counter;
        } while (CompareAndSwap(counter, old, old+amount)
           == 0);
}
```

那么这是一个原子操作，一定能在上下文切换之前完成，所以就不会出现切换到一个进程的时间片之后，他只能 spin waiting 的情况，也不会出现 deadlock。

如果在完成 old=*counter 的赋值之后，发现 counter 被改掉了，那就进行下一次循环，这个时候 old 又赋值为改之后的 counter 了，那其实一般情况就不会有问题哩。

1.4.6 Load-Linked and Store-Conditional

这个部分书上貌似没有，暂时跳过一下

1.4.7 Test And Set——Satisfying Bounded Waiting

这个部分非常非常重要！要能理解并且把代码背出来

```
/* C-like pseudo code */
Initially Boolean waiting[i] = false; lock = false;(false
    在这里也就是0)

lock() {
        waiting[i] = true; // 表示一个进程是否在尝试获得锁
        while (waiting[i] && (TestAndSet(lock, 1) == 1));
        // 那么开锁的条件就是lock=false或者waiting[i]=
            false。
        waiting[i] = false;
}

unlock() {
        j = (i + 1) % n; // 循环查找下一个等待锁的进程
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = false;
                // 如果大家都不要锁，饶了一圈回到我自己
                    了，那就释放锁
        else
                waiting[j] = false;
                // 如果有进程在等待锁，那就把锁传过去
}
```

现在又两个问题：一个是为什么这段代码就能够满足 bounded waiting 了呢？另一个问题是什么时候锁被释放了呢？

因为对于一个想要获取锁的进程来说，他只需要等一圈就可以了；只有所有进程都不要这个锁的时候，锁才会被释放，不然只是锁的传递。

1.4.8 Fetch And Add

是 Bakery 算法的硬件实现，更加聪明一些（把取号变成一个原子操作，也就是把取号的过程保护住），但是内容其实不在书上。

```
int FetchAndAdd(int *ptr) {
```

```
2         int old = *ptr;
3         *ptr = old + 1;
4         return old;
5 }
6
7 /* ticket lock */
8 typedef struct __lock_t { int ticket; int turn; } lock_t;
9 void lock_init(lock_t *lock) {
10        lock->ticket = 0;
11        lock->turn = 0;
12 }
13 void lock(lock_t *lock) {
14        int myturn = FetchAndAdd(&lock->ticket);
15        while (lock->turn != myturn);
16 }
17 void unlock(lock_t *lock) {
18        lock->turn = lock->turn + 1;
19 }
```

1.4.9 Spin-Waiting

现在我们来讲讲怎么解决 Spin-waiting 的问题：

可以用到一些 OS support（不是 hardware support），或者说提供一些 API，让 user 能够在写代码的时候避开需要 wait 的那些 thread，尽可能减少时间片的浪费。

比如说有一个 syscall 叫做 yield：process 自己把自己从 running state 搞到 ready state 去。这样做确实解决单核 CPU 的大部分时间浪费问题，但是**进程要切来切去还是很麻烦的**。

也就是说，现在是 scheduler 决定哪个 process/thread 下一个运行，这种情况下线程随机获得锁，没办法保证公平性，所以我们想对 scheduling 做一些控制，将等待锁的进程都放在一个队列里面，如果锁可用，就把队列的第一个进程放进来 running。

一个解决的方法是用 park() 和 unpark()，那么因为这个部分同样书上没有，同时也比较复杂，所以我们先不管。

考点 12. 我们要注意，不管怎么做，*spin* 都不是能够完全 *avoid* 掉的，只能说可以把它限制一下。

下面的内容 very important!

spin-waiting 这个东西并不是说它一直都是不好的，比方说这种情况：在多处理器系统当中，一个 thread 在一个 processor 运行的时候，另一个 thread 在另一个 processor 上可以做一会 spin waiting 来等第一个 thread 运行完它的 critical section，这样对于一个 thread 来说它就不需要被做 context switch 了，只要稍微等等就好。

最后讲了一个优先级反转和优先级继承的东西，有点像是 mutex lock 的特性：讲了两个概念，一个是 priority inversion（优先级反转:A higher-priority thread waiting for a lock held by lower-priority thread）：发生在具有**两个以上优先级（至少三个）**的系统中；另一个就是解决这个问题的 priority-inheritance（优先级继承）：低优先级的 process 换到更高的优先级，把资源释放掉之后，再恢复自己之前的优先级。

1.5 Locked Data Structures

dk 讲的很快很快，或许，不是很重要吧。

1.6 Condition Variables

我们前面学了这些东西，是不是以为 mutex lock 能解决所有问题呢？实际上不是这样的，我们先来看一下下面这个例子：

```
1  void *child(void *arg) {
2          pintf("child\n");
3          // Problem #1: how to indicate we are done?
4          // 我们没有任何机制能够通知主线程说我们已经完成了
               子进程的thread
5          return NULL;
6  }
7
8  int main(int argc, char *argv[]) {
9          printf("parent:␣begin\n");
10         pthread_t c;
11         pthread_create(&c, NULL, child, NULL);
12         // Problem #2: how to wait for child?
13         // 因为子线程没有通知我，所以我也不知道子线程是做
               完了还是没有做完。
14         printf("parent:␣end\n");
15         return 0;
16 }
```

那么这个问题能用互斥锁来解决吗？很明显不可以（因为这里都没有 critical section），这时候我们想到了可以用一个 shared variable 来做这个事情：

```
1  void *child(void *arg) {
2          pintf("child\n");
3          done = 1;
4          return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8          printf("parent: begin\n");
9          pthread_t c;
10         pthread_create(&c, NULL, child, NULL);
11         while (done == 0);
12         printf("parent: end\n");
13         return 0;
14 }
```

这个方法看上去可以，而且确实是可以，但是它很低效，因为对于父进程来说我得一直 spin-waiting 去 check done 的值。

这时候我们希望能够用条件变量 (condition variable)，它是一个显式的排序队列，有两种接口：wait()（让它自己睡去吧）和 signal()（唤醒一个满足条件的 thread），同时在用条件变量的时候，我们也会用到 mutex。

```
1  pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
2  pthread_cond_signal(pthread_cond_t *c);
```

我们来看看如何从 condition variable 解决我们上面说到的问题：

```
1  int done = 0;    // 共享变量，表示子进程是否已经完成
2  pthread_mutex_m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6   pthread_mutex_lock(&m);          // 加锁
7   done = 1;                                    // 表示子
       进程完成了
8   pthread_cond_signal(&c);         // 发出信号告诉主进程我已
       经完成了
9   pthread_mutex_unlock(&m);        // 解锁
10 }
```

```
11
12 void *child(void *arg) {
13  printf("child\n");
14  thr_exit();              // 通知主进程我已经完成嘞
15  return NULL;
16 }
17
18 void thr_join() {
19  pthread_mutex_lock(&m);          // 加锁
20  while (done == 0)
21   pthread_cond_wait(&c, &m);// 如果done一直为0，那也就说明
        子进程还没完成，运行这行代码的时候会释放锁m，同时我也
        会堵在这里等待子进程signal一下，如果c的值被signal了，
        主进程能重新获得锁m，最后再解锁就可以顺利出去了。
22  pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26  printf("parent:␣begin\n");
27  pthread_t p;
28  pthread_create(&p, NULL, child, NULL);
29  thr_join();
30  printf("parent:␣end\n");
31  return 0;
32 }
```

从上面的例子中，我们也可以看到，condition variable 会有一个 lock(m) 和一个 flag(done)。我们来分析一下如果两个少了一个该怎么办：

注意. *Condition Variable* 不可以先 *signal* 再 *wait*!

```
1 void thr_exit() {
2  pthread_mutex_lock(&m);
3  pthread_cond_signal(&c);
4  pthread_mutex_unlock(&m);
5 }
6
7 void thr_join() {
8  pthread_mutex_lock(&m);
```

```
 pthread_cond_wait(&c, &m);
 pthread_mutex_unlock(&m);
}

/* broken code #1 */
```

这里犯的错误就是可能子线程先 signal，父线程再 wait，那么父线程永远不能被唤醒，就寄了。

再看看这个情况：

```
void thr_exit() {
 done = 1;
 pthread_cond_signal(&c);
}


void thr_join() {
 if (done == 0)
  pthread_cond_wait(&c);
}


/* broken code #2 */
```

这个情况也是一样，可能先 if(done==0)，再子线程 done=1，并且 signal，这下父进程 wait，就没办法再被 signal 了，又寄了。

1.6.1 The Producer-Consumer Problem

一个人不停生产，喂给消费者作为输入，中间有一个缓冲区。

我们先来看一个简单的情况，只有一个 producer，一个 consumer 和 buffer_size 为 1 的情况（这是一个 correct solution）：

```
/* Let's begin with only 1 producer, 1 consumer,
   buffer_size = 1 */

cond_t cond;
mutex_t mutex
void *producer(void *arg) {
        for (int i = 0; i < loops; i ++) {
```

```
7                   pthread_mutex_lock(&mutex);
8                   if (count == 1) // 表示buffer里面有东西
                        了，等着
9                       pthread_cond_wait(&cond, &mutex);
10                  put();                  // produce
11                  pthread_cond_signal(&cond);
12                  pthread_mutex_unlock(&mutex);
13          }
14  }
15  void *consumer(void *arg) {
16          for (int i = 0; i < loops; i ++) {
17                  pthread_mutex_lock(&mutex);
18                  if (count == 0) // 表示buffer里面没东西
                        了，同样等着
19                      pthread_cond_wait(&cond, &mutex);
20                  get();                  // consume
21                  pthread_cond_signal(&cond);
22                  pthread_mutex_unlock(&mutex);
23          }
24  }
```

但是这个代码就不适用有很多 consumer 的情况啦（很好理解，直接看 PPT 就行）问题本质在于，我们只 check 了一次 count 的数值 (if 语句)，所以一个很自然的想法就是把 if 换成 while（我们以一个 producer 和两个 consumer 为例）：

```
1   /* Still broken */
2
3   cond_t cond;
4   mutex_t mutex
5   void *producer(void *arg) {
6           for (int i = 0; i < loops; i ++) {
7                   pthread_mutex_lock(&mutex);
8                   while (count == 1)      // use "while"
                        instead of "if"
9                       pthread_cond_wait(&cond, &mutex);
10                  put();                  // produce
11                  pthread_cond_signal(&cond);
12                  pthread_mutex_unlock(&mutex);
13          }
```

```
14 }
15 void *consumer(void *arg) {
16         for (int i = 0; i < loops; i ++) {
17                 pthread_mutex_lock(&mutex);
18                 while (count == 0)       // use "while"
                       instead of "if"
19                         pthread_cond_wait(&cond, &mutex);
20                 get();                   // consume
21                 pthread_cond_signal(&cond);
22                 pthread_mutex_unlock(&mutex);
23         }
24 }
```

但是很遗憾它还是错的，其实原因也很简单，producer 和 consumer 用的是同样的 condition，那么比方说我一个 consumer 消耗了一个资源，理论上我想要唤起的是 producer，但是不小心唤醒了另一个 consumer，那就炸了呀兄弟。

那么同样很自然的，我们可以想到用两个 conditon variables 来解决这个问题，producer 和 consumer 用的应该是不一样的：

```
1  /* buffer_size = 1 */
2
3  cond_t empty, fill;                      // two condition
       variables
4  mutex_t mutex
5  void *producer(void *arg) {
6          for (int i = 0; i < loops; i ++) {
7                  pthread_mutex_lock(&mutex);
8                  while (count == 1)
9                          pthread_cond_wait(&empty, &mutex);
10                 put();                   // produce
11                 pthread_cond_signal(&fill);              //
                       只signal fill, 说明现在buffer是满的
12                 pthread_mutex_unlock(&mutex);
13         }
14 }
15 void *consumer(void *arg) {
16         for (int i = 0; i < loops; i ++) {
17                 pthread_mutex_lock(&mutex);
18                 while (count == 0)
```

```
19                          pthread_cond_wait(&fill, &mutex);
20                  get();                  // consume
21                  pthread_cond_signal(&empty);    // 只
                        signal empty，说明现在buffer是空的
22                  pthread_mutex_unlock(&mutex);
23          }
24  }
```

在解决了这个问题之后，我们很自然地会去想，如果 buffer 的 size 不止是 1 呢？

```
1   /* dealing with bounded buffer instead of single buffer */
2
3   cond_t empty, fill;                 // still two
        condition variables
4   mutex_t mutex
5   void *producer(void *arg) {
6           for (int i = 0; i < loops; i ++) {
7                   pthread_mutex_lock(&mutex);
8                   while (count == MAX)    // remember:
                        always use "while"
9                           pthread_cond_wait(&empty, &mutex);
10                  put();                  // produce
11                  pthread_cond_signal(&fill);             //
                        注意这里的fill不是说真的满了，只是说现
                        在你consumer可以去消耗了
12                  pthread_mutex_unlock(&mutex);
13          }
14  }
15  void *consumer(void *arg) {
16          for (int i = 0; i < loops; i ++) {
17                  pthread_mutex_lock(&mutex);
18                  while (count == 0)
19                          pthread_cond_wait(&fill, &mutex);
20                  get();                  // consume
21                  pthread_cond_signal(&empty);    // 同样注
                        意这里的empty不是空了，而是告诉producer
                        你可以继续生产
22                  pthread_mutex_unlock(&mutex);
```

```
23          }
24  }
```

最后呢，我们讲一个好玩的情况（也算是 condition variable 的一个特性）：

比如说现在我们没有空间了，Ta 想要 100B 的空间，Tb 想要 10B 的空间，Tc 刚好要释放 50B 的空间，那么我们会唤醒哪一个线程呢？我们会用 pthread_cond_broadcast 把所有线程全部唤醒，再让他们看自己是不是符合条件。

1.7 Semaphores

Semaphore 比 condition variable 牛逼在哪里呢，condition variable 不能应对先 signal 再 wait 的情况，但是 semaphore 可以，它能解决所有 mutex 和 condition variable 的问题。对于 semaphore 的操作也是 wait() 和 signal()。

注意. 这里的 *wait* 和 *signal* 和 *condition variable* 的是不一样的，注意区分。*wait* 的时候，信号量*-1*，*signal* 的时候，信号量 *+1*。同时 *wait()* 和 *signal()* 都是原子操作。

我们来看这样一个例子：

```
1   typedef struct {
2           int value;
3           struct process *list;
4   } semaphore;
5
6   wait(semaphore *s) {
7           s->value --;
8           if (s->value < 0) {
9                   add this process to s->list;      // 进队
                        列，waiting to be signaled，记得如果s->
                        value为0的时候，刚好可以运行，不用进队
                        列。
10                  block();
11          }
12  }
13
14  signal(semaphore *s) {
15          s->value ++;
16          if (s->value <= 0) {
17                  remove a process P from s->list;          //
                        remove一个进程，然后把这个进程wake up
```

```
18                  wakeup(P);          // 类似不释放锁，直接把锁
                        传递给下一个进程
19              }
20      }
```

所以说，semaphore 的功能很强大：

$$Semaphore = Mutex + Integer + ConditionVariable$$

它最牛逼的地方还是在于，它可以先 signal() 再 wait()，condition variable 就不可以这么做。

1.7.1 Signaling

直接看 PPT 就行，非常简单兄弟。

1.7.2 Rendezvous

仍然很简单，还是看 PPT。

1.7.3 Mutex

依然很简单，自己解决。

注意一下 semaphore 的初始值很关键。

1.7.4 Multiplex

比方说我们最多允许 MAX 个进程访问 critical section，那么我们的代码应该这样写，同时 multiplex 这个信号量的初始值应该设为 MAX：

```
1   wait(multiplex);
2   count = count + 1;          // critical section
3   signal(multiplex);
```

注意. *Semaphore 更像是一种资源，如果是 condition variable，那初始值就是 0，代表现在没有资源；如果是 mutex，那初始值就是 1，也就是只有一张门卡；如果像上面那样，就说明有 MAX 张这样的门卡。*

1.7.5 Barrier

这里开始就有点难了哈（但还是我们能轻松理解的最后一个问题了），现在要求必须 n 个线程都到这个地方之后，一起进去，不能有一个先跑进去这样的。

如果只是想要完成这个任务，那么代码是比较简单的，**一定要背出来这些代码：**

```
/* initialization */
int count = 0;
semaphore mutex = 1;
semaphore barrier = 0;

wait(mutex);
count = count + 1;
signal(mutex);
if (count == n)
        signal(barrier);
wait(barrier);
signal(barrier);
critical point;
```

在这个例子当中，如果 n=10，也就是说，当有 10 个进程到这里的时候大家才能一起走，一共会有 11 次 signal(barrier)，最后 barrier 的值为 1。所以说这么做虽然是 correct，但好像不完全是 correct 的。

那我们之前也说了，这样做的 barrier 是一次性的，我们想让这样的 barrier 可以被重复利用（**虽然有一个很赖皮的做法，就是如果我要 10 个 barrier，我不复用 barrier，我直接搞 10 个也是可以的，但是考试的时候不能写的太简单**）：

```
// 这个代码不需要背出来，但是要能体会（现场写能写出来）
// 注意：真正意义的barrier就是有n为多少，就有多少个线程，
   不会有多出来的，现在解决的是之前进去barrier然后又绕回来
   冲进去的问题。
semaphore barrier1 = 0, barrier2 = 0, mutex = 1;

rendezvous;
wait(mutex);
count += 1;
if (count == n)
        for (int i = 0; i < n; i ++)
                signal(barrier1);
```

```
signal(mutex);
wait(barrier1);
critical point;
wait(mutex);
count -= 1;
if (count == 0)
        for (int i = 0; i < n; i ++)
                signal(barrier2);
signal(mutex);
wait(barrier2);
```

1.7.6 Pairing

这能算是考试难度的天花板了。（比如说两个氢和一个氧怎么配对呢？）

这个问题是这样的：在舞会当中，一个领舞来的时候，看看有没有伴舞在等着，如果有就去跳舞，如果没有就等着。对于伴舞也是一样，看看有没有领舞，如果有就跳舞，没有就等着。

解决方法如下：

```
/* initialization */
int num_l = 0, num_f = 0;
semaphore leader = 0, follower = 0, pairing = 0, mutex =
    1;
```

```
/* leader */
wait(mutex);
if (num_f > 0) {
        num_f --;
        signal(leader);
}
else {
        num_l ++;
        signal(mutex);
        wait(follower);
}
dance();
wait(pairing);
signal(mutex);
```

```
/* follower */
wait(mutex);
if (num_l > 0) {
        num_l --;
        signal(follower);
}
else {
        num_f ++;
        signal(mutex);
        wait(leader);
}
dance();
signal(pairing);
        // no signal(mutex);
```

值得注意的是，在 leader 和 follower 的最后两行，对于 leader 来说，它会被 pairing 卡住，对于 follower 来说，它会被 mutex 卡住。（保证不会有两个 follower 进场，也不会有两个 leader 进场）（获得两次锁，也会释放两次锁，一定要自己能够理解）

有一个问题，怎么就保证 leader 和 follower 成对跳舞呢？

还是依靠最后两行的代码，不能一个地方 signal 两个 semaphore。

最后我们来讲五个很经典的问题，每个都需要全面掌握。

1.7.7 The Producer-Consumer Problem

```
semaphore empty = MAX, full = 0, mutex = 1;
// MAX表示size of buffer

// empty+full永远是MAX
// 消耗一个empty，生产资源

// 在这里答案是用mutex的想法做的，当然也可以用condition
   variable做
void *producer(void *args) {
        for (int i = 0; i < loops; i ++) {
                wait(empty);
                wait(mutex);
                put();
```

```
13                  signal(mutex);
14                  signal(full);
15          }
16  }
17
18  // 消耗资源，生产一个empty
19  void *consumer(void *args) {
20          for (int i = 0; i < loops; i ++) {
21                  wait(full);
22                  wait(mutex);
23                  get();
24                  signal(mutex);
25                  signal(empty);
26          }
27  }
```

1.7.8 The Dining Philosophers

There are five "philosophers" sitting around a table. Between each pair of philosophers is a single chopstick (and thus, five total). The philosophers each have times where they think, and don't need any chopsticks, and times where they eat. In order to eat, a philosopher needs two chopsticks, both the one on their left and the one on their right. The basic loop of each philosopher is as follows, assuming each has a unique identifier $p \in [0, 4]$:

首先我们看一个不那么对的解决方法，但能够给我们一些启发：

```
1   /* a failed solution, why failed? */
2   // 因为大家如果都先去左边找筷子，大家都只能找到一支->
        deadlock
3
4   int left(int p) { return p; }
5   int right(int p) {return (p + 1) % 5; }
6
7   void putchopsticks() {
8           signal(chopsticks[left(p)]);
9           signal(chopsticks[right(p)]);
10  }
11
```

```
12 void getchopsticks() {
13         wait(chopsticks[left(p)]);
14         wait(chopsticks[right(p)]);
15 }
```

所以我们有下面的解决方案：一个人从右手拿，其他人正常从左手拿：

```
1  /* a correct solution */
2
3  int left(int p) { return p; }
4  int right(int p) {return (p + 1) % 5; }
5
6  void putchopsticks() {
7          signal(chopsticks[left(p)]);
8          signal(chopsticks[right(p)]);
9  }
10
11 void getchopsticks() {
12         // 这个地方就非常有意思了
13         if (p == 4) {
14                 wait(chopsticks[right(p)]);
15                 wait(chopsticks[left(p)]);
16         }
17         else {
18                 wait(chopsticks[left(p)]);
19                 wait(chopsticks[right(p)]);
20         }
21 }
```

1.7.9 The Readers-Writers Problem

写者可以修改数据，但只能有一个，但是读者是可以有很多个的，读者不能修改数据。

第一类读者写者问题（读者优先）：读者不能等着，除非写者已经获得了修改的权限。

```
1 semaphore write_mutex = 1;
2 semaphore readcount_mutex = 1;
3 int read_count = 0;
```

```
4
5  // 写者很简单，就是去竞争写的锁
6  // 考试的时候do，while没写不会扣分
7  void write() {
8   do {
9    wait(write_mutex);
10   /* writing */
11   signal(write_mutex);
12  }
13  while (true);
14 }
15
16 void read() {
17   entry(read,write)
18   /* reading */
19   exit(read,write)
20 }
```

当然在这种情况下，写者可能会饿死，因为 writer 都没有资格和 reader 抢一个锁：

```
1  semaphore readcount_mutex= 1;
2  semaphore writemutex=1;
3  int read_count = 0;
4  semaphore readmutex =1;
5
6  void write() {
7    // 和下一个要来的读者抢一下锁，让后来的reader都无法进入
8    wait(readmutex);
9    // 把后面来的那个读者给比下去了，现在就等着读者全部出去
       之后，就可以写了
10   wait(writemutex);
11   /* writing */
12   signal(writemutex);
13   signal(readmutex);
14 }
15
16 void read() {
17   // 同样和writer去竞争这个锁
```

```
18    wait(readmutex);
19    signal(readmutex);
20    entry(read,write)
21    /* reading */
22    exit(read,write)
23  }
```

第二类读者写者问题（写者优先）：一旦读者已经就位了，他必须要尽快写。

```
1  semaphore readcount_mutex= 1;
2  semaphore writemutex=1;
3  int write_count = read_count = 0;
4  semaphore writecount_mutex= 1;
5  semaphore readmutex=1;
6
7  entry(A,B){
8          wait(Acount_mutex);
9          A_count++;
10         if(A_count == 1) wait(B_mutex);
11         signal(Acount_mutex);
12 }
13
14 exit(A,B){
15         wait(Acount_mutex);
16         A_count--;
17         if(A_count == 0) signal(B_mutex);
18         signal(Acount_mutex)
19 }
20
21
22 void write() {
23   entry(write,read);
24   wait(writemutex);
25   /* writing */
26   signal(writemutex);
27   exit(write,read)
28 }
29
30 void read() {
```

```
31      wait(readmutex);
32      entry(read, write);
33      signal(readmutex);
34      /* reading */
35      exit(read, write);
36  }
```

1.8 Monitors

2 Concepts Organization

1. Race condition: Several processes (threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place. Result indeterminate.
2. Critical section: Multiple threads executing a segment of code, which can result in a race condition.
3. Critical section problem(three requirements): Mutual exclusion, progress, bounded waiting.
4. Peterson's Solution: A software-based algorithm for addressing critical section problems between two processes.
5. Interrupt masks: A technique where the CPU disables interrupts to prevent context switches during the execution of critical sections.
6. Test-and-set: An atomic hardware instruction used to implement mutual exclusion.(For more details, plz read the main text)
7. Compare-and-swap: An atomic hardware instruction used to implement mutual exclusion.(For more details, plz read the main text)
8. Spin-waiting: A thread repeatedly checks a condition (typically a lock variable) in a loop without relinquishing the processor(release the control of CPU cycle).
9. Mutex: A synchronization primitive used to prevent multiple threads from accessing a shared resource at the same time.
10. Semaphores: A counter that tracks the number of available units of a resource.
11. Condition variables: A synchronization primitive that allows threads to wait for certain conditions to become true.
12. Monitors:

第七章　Deadlocks

1　In-class Contents

1.1　Warm-up

我们先来看看 OS 中的两个类型的 bugs。

1.1.1　Non-deadlock Bugs

非死锁型错误：绝大多数这类错误是 atomicity violation（原子性违例）或者是 order violation（顺序违例），解决也很简单，用 mutex lock 或者是 signal 就可以解决。

（1） Atomicity violation bugs: 某个操作序列应该是原子的，但实际上在运行的时候被从中间打断了。

```
/* Thread 1 */
if (thd->proc_info) {
 ...
 // 假如说运行到这里的时候突然thread 2开始运行了，就把proc_info的内容置空了。
 fputs(thd->proc_info, ...);
 ...
}
```

```
/* Thread 2 */
thd->proc_info = NULL;
// 这个操作相当于破坏了thread 1里的atomicity
//
```

（2） Order violation Bugs: 因为执行顺序的问题而出的错误

```
/* Thread 1 */
void init() {
 ...
 mThread = PR_CreateThread(mMain, ...);
  // 按理说应该创建完mThread之后再运行mMain
 ...
}
```

```
/* Thread 2 */
void mMain(...) {
 ...
 mState = mThread->State;
 // Thread 1还没来得及初始化mThread，Thread 2就运行
    这个，那肯定获取不到mThread的State呀
 ...
}
//
```

1.1.2 Deadlock Bugs

那么为什么 Deadlock 会发生呢？

（1） 一个原因是 dependencies（相互依赖，比如模块 A 调用模块 B，同时模块 B 也会调用模块 A）：比如说虚拟内存和文件系统互相等待对方。

（2） 另一个原因是封装的问题：比如说 Java Vector 类里面的 AddAll()，函数，可以看下面的代码：v1 在调用 AddAll 的时候，会想要保护自己而给自己上锁，然后试图去开 v2 的锁，同时 v2 也给自己上锁，并且想开 v1 的锁，那么就死锁啦。

```
/* Thread 1 */
Vector v1, v2;
v1.AddAll(v2);

/* Thread 2 */
v2.AddAll(v1);
```

1.2 System Model

我们对我们的系统建一个模型：

我们认为 System 有很多 resources:$R_1, R_2, \cdots, R_m$；然后 resource types 包含 CPU cycles, memory space, I/O devices。

同时每个 resource type R_i 有 W_i 个实例；每个 process 用这些资源的方式包括：request, use, release。

1.3 Deadlock Characterization

下面的内容非常重要，讲产生 Deadlock 的四个必要非充分条件：

（1） Mutual exclusion：同时只能有一个 process 占用一个 resource，也就是说资源是不可以共享的。

（2） Hold and wait（一边占用，一边等待）：一个 process 至少占有一个 resource，并且正在等待别的 processes 释放它们的资源。（也就是说，自己有自己的资源的同时，还要抢别人的资源，也就是说，如果请求资源之前必须释放自己已经占有的资源的话，就不会发生死锁）。

（3） No preemption：process 占用的资源只能自己释放，不能被系统强制夺走（如果一个 resource 可以被抢占的话，那就要么没死锁，要么它和 deadlock 没有关系）。

（4） Circular wait：说的简单一点，就是 processes 互相等待彼此的 resources，形成一个圈；当然如果形成两个圈的话，那就产生两个死锁啦！

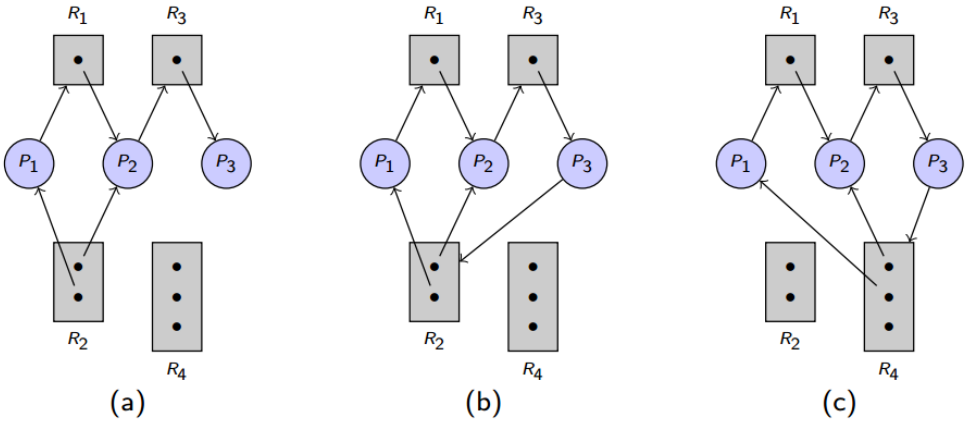
那么为什么是 necessary 而不是 sufficient 呢？看 PPT 吧，感觉 dk 讲的也不是非常清楚，举了一个反例是 multiplex，还有一个反例可以看后面的内容。

1.3.1 Resource Allocation Graph

那么如何描述一个 deadlock 呢？可以用 resource allocation graph。

定义 1.1. *Resource Allocation Graph*：包含一些 *vertices* 和一些 *edges*，其中 *vertices* 有两种（*Processes*、*Resources*），*edges* 也有两种（*Request edge* 和 *Assignment edge*）。

话不多说，我们来看几个例子：



首先看 (a) 图，图中没有形成环，所以没有 deadlock；其次看 (b) 图，图中有环，“所以”形成 deadlock；最后看 (c) 图，里面有环，好像要形成 deadlock 了？NOPE!!! 在这张图中，我们可以看见 R4 是有 3 分的，所以分出去两份之后，正好留一份能够满足 P3 的需求，所以 P3 是可以 make progress 的，所以没有 deadlock！另外再补充一些，这个 graph 表示的是当前时刻的资源分配情况，也就是说，当前时刻 P3 问 R4 要一份 resources，那下一刻的箭头可能就是 R4 指向 P3 了。

所以我们现在就可以总结出一些 basic facts：

（1） 如果一个图没有 cycle，那么肯定没有 deadlock。

（2） 如果一个图中存在 cycle，这时候要分类讨论：如果说每个 resource type 只有一个实例，那么就会有 deadlock；如果说每个 resource type 有多个实例，那就要看情况咯，就有 deadlock 的可能。

1.4 Methods for Handling Deadlocks

我们有四个方法来应对 deadlocks，分成两类：

一类是确保系统不会进入 deadlock 的状态：

- Deadlock prevention：把形成 deadlock 的 4 个必要条件一个个毙掉就可以嘞
- Deadlock avoidance：例如银行家算法，始终保持系统处于一个安全状态

另一类是允许系统进入 deadlock state 但是之后会 recover:

- Deadlock detection
- Recovery from deadlock

但是吧，你看上去我们要讲这么多的算法，实际上大部分 OS 都默认没有死锁的存在，因为死锁更多是用户/程序员该干的事情。

1.5 Deadlock Prevention

因为这块部分不是内核在做的事情，所以考试也就只会考考概念。

1.5.1 Circular Wait

做的事情很简单，就是给所有资源类型一个全局编号，然后所有进程必须按照资源编号升序请求资源。（本质：从代码层面 prevent deadlock）

当然这个升序可以是 total ordering（全序），也可以是 partial ordering（偏序），类似想法的还有哲学家问题。

例 21. *在给两个 mutex lock 加锁的时候，可以先给地址大的那个 lock 加锁。*

1.5.2 Hold and Wait

核心思想就是保证当一个 process 申请资源的时候，它不能占有任何其他资源。那么也就是说，对于一个 process 来说，要么 process 在执行前申请和分配所有需要的资源（同时也可能拿走一些其实本身并不需要的资源），或者就是在运行过程中只允许没有占用任何资源的 Process 去申请资源。”All resources of none”

但是啊，这也会有两个问题，一个问题是 low resource utilization（你很难搞到所有的资源），另一个问题是有可能 starvation（同样是你可能没法拿到所有资源，所以饥饿）。

当然，这个 solution 也是有问题的，问题就出在：

（1） 没法确保分装的函数能遵循上面的”rules”

（2） 并发率降低（没有资源，那就没办法运行，吞吐量也会随之降低）

1.5.3 No preemption

核心思想：如果一个 process 占用了一些资源，但是没办法运行，那就把它占用的这些资源都释放出去（在仿真允许 preemption 的感觉）。Process 只有在它重新获得了旧的资源，并且也获得了它需要的新的资源的时候，这个 process 才会继续运行。

```
1 top:
2         lock(L1);
3         if (trylock(L2) == -1) {
4                 unlock(L1);
5                 // 拿不到资源，那算了，我就把锁释放掉吧
6                 goto top;
```

```
7         }
```

当然，这个 solution 也是有问题的：

（1） 同样也是没办法封装的很好

（2） 可能出现 Livelock（活锁）：有可能两个线程都是占了一个锁，发现得不到另一个锁，就放弃，一直重复在干这个事情，最后卡住了。

解决方案：等一会就好啦

```
1 top:
2         lock(L1);
3         if (trylock(L2) == -1) {
4                 unlock(L1);
5                 sleep(rand()%10);
6                 goto top;
7         }
```

1.5.4 Mutual Exclusion

对于一些可共享的资源（比如只读文件），它们本身就是可以共享的，那很好，就不需要锁了。**但是!**，还有一些资源它们不是可以共享的，它们必须要有锁来进行管理，也就是说，**互斥是不可避免的**。

但是！我们可以用 compare and swap 这个 hardware support 来写代码，因为 compare and swap 可以实现一个 lock-free synchronization（当然，有 lock，才可能会有 deadlock，没有 lock，是不是就不会发生 deadlock 了呢）

这招被称为无等待并发 (wait-free concurrency)

```
1  /* atomically increment a value by a certain amount */
2  int CompareAndSwap(int *adderess, int expected, int new) {
3          if (*address == expected) {
4                  *address = new;
5                  return 1;
6          }
7          return 0;
8  }
9  void AtomicIncrement(int *value, int amount) {
10         do {
11                 int old = *value;
```

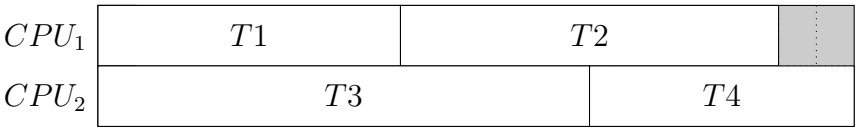
```
12          } while (CompareAndSwap(value, old, old + amout)
                == 0);
13  }
```

1.6 Deadlock Avoidance

我们之前说的是在代码上 Prevent deadlock（这是程序员应该做的事情），现在 OS 通过**调度**来 avoid deadlock。

例 22. *Assume we have two processors (CPU_1, CPU_2) and four threads (T_1, T_2, T_3, T_4) which must be scheduled upon them. Assume further we know that each thread will grab some locks L_1 or L_2 as follows.*

	T_1	T_2	T_3	T_4
L_1	*yes*	*yes*	*no*	*no*
L_2	*yes*	*yes*	*yes*	*no*



那么在这个例子中你可能会很奇怪，不对啊兄弟，你这个 *T1* 和 *T3* 难道不会抢资源嘛？不是这样的兄弟，对于 *T3* 来说，它不会等待任何锁，所以它也就不会阻塞 *T1* 的资源，也就是说，它们不满足 *deadlock* 的 *hold and wait* 和 *circular wait* 的条件。

下面的内容非常非常非常重要！因为这些都是 OS 在做的事情！

那么我们如何 avoid 呢？对于系统来说，它需要有一些先验信息，说白了，就是 OS 要看 Thread/Process 申请了内核资源之后，它是不是能够正常运行，大概步骤是这样的：

（1） 最简单也是最有效的方法就是让每个 process 声明它们每种 resource 要的最多的数量

（2） Deadlock-avoidance 算法动态检查资源分配的情况，来保证不会出现 circular-wait 的情况，从而 avoid deadlock。

（3） 资源分配情况是根据可分配资源和已分配的资源，还有 process 要求的最大资源数量一起决定的。

1.6.1 Safe State

当一个 process 想要一个空闲的资源的时候，系统需要看看资源如果分配给它之后，系统能不能处于一个"safe state"。

定义 1.2. *Safe State*：如果能构建一个 *process* 序列，比如说 $< P_1, P_2, \cdots, P_n >$，能够保证 *P1* 能够执行完，然后 *P1* 释放它的资源，再保证 *P2* 能够执行完，直到最后序列中的所有 *Process* 都能够顺利执行，那么说明这个 *system* 是处于一个 *safe state*。（如果两个 *process* 没有资源冲突，那也就无所谓这两个 *process* 的顺序）

注意. 我们找到的这个序列是在"最坏情况"下也能保证执行的序列，那么为什么会提到"最坏情况"呢？比方说一个 *Process*，它有一个条件语句，例如 *if():L1()*，那么它可能用到 *L1* 的资源，也可能用不到，但是我们的算法需要把这种情况考虑进去，*L1* 的资源该给还是要给，用不用再说。

例 23. 我们来看一个例子：*Consider a system with 12 magnetic tape drives and three processes:* P_0, P_1, P_2.

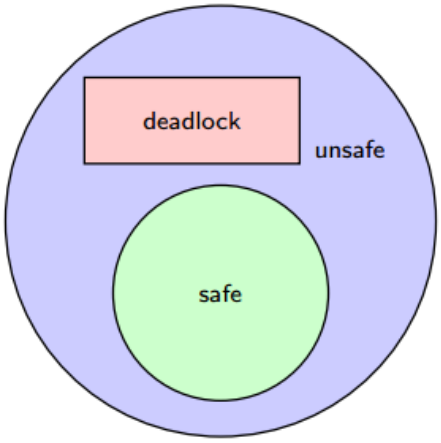
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- *At time* t_0*, the system is in a safe state. The sequence* $< P_1, P_0, P_2 >$ *satisfies the safety condition.*
- *Show by an example that a system can go from a safe state to an unsafe state.*
- *Suppose that, at time* t_1*, process* P_2 *requests and is allocated one more tape drive. The system is no longer in a safe state.*

具体讲讲的话，现在 P_1, P_0, P_2 加起来用了 *5+2+2=9* 个资源，那么我们还剩 *3* 个，可以先给 *P1*，完成后会剩下 *5* 个资源，再给 *P0* 运行，完成后会剩下 *10* 个资源，最后再让 *P2* 运行，就完成啦。

但是！如果说，一开始给 *P2* 多分配一个，也就是 *5*，*2*，*3*。这种情况下，*P1* 完成后，只有 *4* 个资源，*P0* 和 *P2* 都没法完成，那就进入了 *unsafe state*，就可能死锁咯。

注意. 有一件值得注意的事情是，如果一个系统处于 *safe state*，那么它肯定不会有 *deadlock* 的可能；但是，如果一个系统处于 *unsafe state*，它也只是有可能 *deadlock*。而我们 *deadlock avoidance* 做的事情是，保证一个系统不会进入 *unsafe state*，如果有可能进入 *unsafe state*，就拒绝请求。



那么有哪些 avoidance algorithms 呢？有下面这两种:

（1） 如果每个资源类型只有一个实例（比如说只有一个打印机，一个锁），这时候我们用 resource-allocation-graph algorithm。

（2） 如果每个资源类型有多种实例呢？这时候我们就要用银行家算法啦 (Banker's algorithm)。

1.6.2 Resource-Allocation-Graph Algorithm

在这里我们讲一个新的边：

定义 1.3. *Claim edge:* $P_i \dashrightarrow R_j$ *表示 Pi 可能会请求资源 Rj，但至少现在没有真的分配过去。*

那么到这里为止，我们就有三种边了：Request edge（P->R）、Assignment edge（R->P）、Claim edge（P->R）。

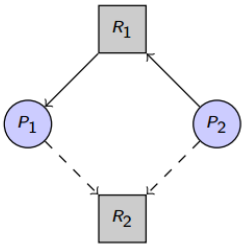
那么三条边之间怎么转换呢？

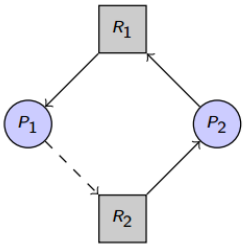
（1） 当一个 process 真的请求了一个 resource 时，claim edge 会变成 request edge。

（2） 当资源真的被分配给一个 process 时，request edge 会变成 assignment edge。

（3） 如果一个 process 把资源释放掉了，assignment edge 会变成 claim edge。

（4） 进程必须提前声明可能会请求哪些资源（就是声明边）





注意. 我们之前说，*resource-allocation-graph* 中出现环的时候，就是 *unsafe state*，这个时候 *claim edge* 是不会参与 *deadlock* 的判定的。但是只有确保 *resource allocation graph* 不会出现环的时候才能把一个 *request edge* 变成一个 *assignment edge*。（这个感觉不是很清晰，但我觉得不是很重要）

下面这个算法非常非常重要！

1.6.3 Banker's Algorithm

首先，银行家算法是针对多个实例的，然后我们做以下假定：

（1） 每个 process 都必须实现声明它要的 maximum use 的资源

（2） 当一个 process 需要一个资源的时候，它必须等待别人把资源给它

（3） 当一个 process 获得了它要的所有资源，它必须在有限的时间里面还回去

我们首先定义一下银行家算法当中要用到的量和数组：

- n Processes 的数量。
- m Resources 有 m 个类型。
- $Available$: 数组长度为 m. 如果 $Available[j] = k$，则说明有 k 个 R_j 是可以分配的。
- Max: $n \times m$ 的矩阵。如果 $Max[i, j] = k$，说明 P_i 最多会需要 k 个 R_j。
- $Allocation$: $n \times m$ 的矩阵。如果 $Allocation[i, j] = k$ 那么说明 P_i 已经被分配了 k 个 R_j 了。
- $Need$: $n \times m$ 的矩阵。如果 $Need[i, j] = k$, 那么 P_i 还需要 k 个 R_j 来完成它的任务。

$$Need[i, j] = Max[i, j] - Allocation[i, j]$$

同时，银行家算法又能分成下面两个部分，为了区分开来，额外用两个小标题来表示，但这两个部分都应归属银行家算法。

1.6.4 Safety Algorithm

其实这个算法做的事情很简单，看下面的例子就可以了：

（1） 首先我们有两个向量，一个 $Work$ 向量，长度为 m，就是 $Available$ 向量还有一个是 $Finish$ 向量，长度为 n，它初始化的时候都为 false，表示这个进程是否已经被完成了。

（2） 之后，我们找一个 i（也就是找那么一个 process），它要满足 Finish[i]=false，且它所需要的资源，也就是 Need[i]≤Work（也就是 Work 能满足 Process i 的需求）-> 这个步骤就相当于在构建 safe sequence。但是，如果没有这样的 process i 存在，那么跳到第 4 步。

（3） 这时候我们已经确保 process i 的要求是可以被满足的，所以我们可以做一个 Work=Work+Allocation[i]，相当于先把 work 的资源给 process i，它完成它的工作之后，把 Finish[i] 置为 True，同时，把资源全部还回去，在这之后跳到第 2 步重复进行。

（4） 最后，如果所有的 Finish[i] 都为 true，那么说明这个系统处于 safe state，反之，就是 unsafe 的。

1.6.5 Resource-Request Algorithm

这个算法做的事情就是利用 safe algorithm 算法来 avoid deadlock，其基本逻辑就是在分配资源的时候看每次分配是否 safe。

在这里我们引入一个向量 $Request_i$，它表示 process i 需要的资源，如果 $Request_i[j] = k$ 那也就说明 P_i 想要 k 个 R_j。

（1） **如果 $Request_i \leq Need_i$，就到第二步。否则就报错，因为 process 会超过它的 maximum claim。**

注意. 这个地方非常容易出错，也是一个扣分点，我们来打一个比方，如果 *Availabel=[100,5]*，*Need=[10,1]*，但是这个时候我申请的 *Request=[11,1]*，虽然说不会产生死锁，但是这是 *unsafe* 的，因为 *procee* 要求的资源超过 *maximum claim* 了。

（2） 如果 $Request_i \leq Available$，那就去第三步。否则 P_i 就得一直等着，因为资源不够它用的。

（3） 最后做一个分配资源的假想状态：$Available = Available - Request_i$、$Allocation_i = Allocation_i + Request_i$、$Need_i = Need_i - Request_i$。然后我们调用 safety 算

法，如果结果是 safe 的，那么就可以把资源分配给 P_i；如果结果是 unsafe 的，那就等着吧。

1.7 Deadlock Detection

这里和之前的区别在于，Deadlock 可能已经发生了，但是现在我想要把这样的 deadlock 给 detect 出来（虽然现在没有 OS 在搞这个事情，真的 deadlock 了还是重启吧）

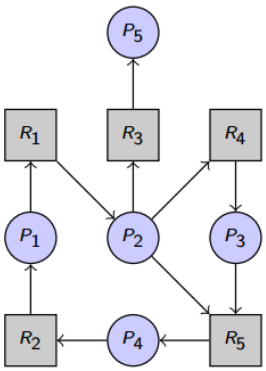
当然这也能分成两种情况，一种是每种 resource type 有一个实例的情况，还有一种显然就是有多个实例的情况。

1.7.1 Single Instance of Each Resource Type

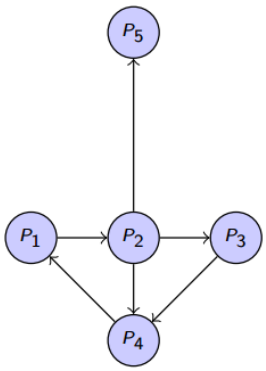
处理的方式和 resource-allocation graph algorithm 很像，我们在这里搞了一个 wait-for graph（就是 resource-allocation graph 的变形），节点都是 process，$P_i \to P_j$ 表达的意思就是 P_i 在等待 P_j 释放资源。

不过！有一个不一样的地方在于，如果 wait-for graph 有环的时候，**一定有一个 deadlock**（是充分条件，不是 possibility of deadlock）。

好，我们来看看如何从 Resource-Allocation Graph 变成 Wait-for Graph（虽然感觉这样好像也没啥必要，完全可以从 Resource-Allocation Graph 判断）：



Resource-Allocation Graph



Wait-for Graph

那么显然图中的有环的，所以有死锁。

1.7.2 Several Instances of a Resource Type

那有多个实例呢？我们有一个 Detection Algorithm，然后和 Safety Algorithm 很关键的一个不同点在于 Detection Alg. 用的是 Request 数组而不是 Need 数组，且它

更关注当下是不是发生 deadlock，也就是保证当前的进程能够前进，未来怎么样是不管的：

书上用的是 Finish 数组，但是 dk 认为应该是 Progress 数组，所以这里都改成 Progress 数组：

（1） 首先我们有两个向量，一个 $Work$ 向量，长度为 m，就是 $Available$ 向量还有一个是 $Progress$ 向量，长度为 n。（对 Progress 的初始化和 safety alg. 已经开始不一样了）如果进程 Pi 当前没有持有资源（Allocation[i] == 0），说明它不会阻塞死锁 → Progress[i] = true，否则，Progress[i] = false

（2） 之后，我们找一个 i（也就是找那么一个 process），它要满足 Finish[i]=false，且它**当前申请的资源**，也就是 **Request[i]≤Work**（当下我要的你能给，但不代表我以后要的你也能给）。如果没有这样的 process i 存在，那么跳到第 4 步。

（3） 同样我们也做一个 Work=Work+Allocation[i]，相当于先把 work 的资源给 process i，把 Progress 为 True，同时，把资源全部还回去（注意这个地方有点问题，因为对于 process i 来说，乐观情况下它会把资源还回去，但也有可能之后要的更多），在这之后跳到第 2 步重复进行。

（4） 最后，如果有 Progress[i]=false，那就陷入 deadlock 了。

那么最后还有一个问题，我们什么时候，以怎样的频率调用这个算法呢？这取决于：

（1） 一个 deadlock 发生的频率如何？

（2） 出现 deadlock 的话回滚多少进程？

如果检测地很随意，那很可能当你检测到的时候已经有很多环了，就会形成很多死锁，可以每次分配完资源后都 detect 一下，这下就能明确知道是哪步导致 deadlock 啦！

1.8 Recovery from Deadlock

那么，怎么解决 Deadlock 呢？有几种方式啊，一种是把所有 deadlock 的进程都杀了，还有一种方式是一个一个杀进程，直到死锁解开。被杀的那个进程叫做 victim。

还有一个办法是资源抢占，我们首先选择一个能够 minimize cost 的 victim，然后回退到 safe state，再 restart process。但是呢，有个问题，就是那个 process 可能很不幸啊，一直是 victim，那就很糟糕了。

2 Exercise

例 24. *A computer system has 3 types of resources A, B, and C with different numbers of instances. There are 4 running processes P_1, P_2, P_3, P_4. The total resources, the resource's $Allocation$ and Max matrices for the four processes are shown as follows:*

	$Allocation$			Max		
Process	A	B	C	A	B	C
P_1	1	3	1	6	5	3
P_2	0	2	2	3	5	3
P_3	2	0	0	3	5	2
P_4	0	1	3	2	4	3
total	6	9	6			

Keys to Q1:

	$Need$		
Process	A	B	C
P_1	5	2	2
P_2	3	3	1
P_3	1	5	2
P_4	2	3	0
$Available$	3	3	0

Keys to Q2:

（1） $Work = Available, Finish[i] = false, 0 \leq i < n.$

（2） Find an $i = 4$.

（3） $Work = Work + Allocation_i, Finish[4] = true$

	A	B	C
$Work$	3	4	3

（4） Go to step 2.

（5） Find an $i = 2$.

（6） $Work =< 3, 6, 5 >, Finish[2] = true$

（7） Go to step 2.

（8） Find an $i = 3$.

Safe sequence: $< P_4, P_2, P_3, P_1 >$

Keys to Q3:

注意. 很重要！考试的时候一定要查 *claim* 有没有超过 *max*！

（1） Pretend to grant.

（2） Current state:

	Allocation			*Need*		
Process	A	B	C	A	B	C
P_1	1	3	1	5	2	2
P_2	1	2	2	2	3	1
P_3	2	0	0	1	5	2
P_4	0	1	3	2	3	0
Total	6	9	6			
Avai.				2	3	0

（3） Find a safe sequence: $(2, 3, 0) \rightarrow P_4(2, 4, 3) \rightarrow P_2(3, 6, 5) \rightarrow P_3(5, 6, 5) \rightarrow P_1(6, 9, 6)$

（4） Safe $\rightarrow$ Yes!

Keys to Q4:

（1） Pretend to grant.

（2） Current state:

	Allocation			*Need*		
Process	A	B	C	A	B	C
P_1	2	3	1	4	2	2
P_2	0	2	2	3	3	1
P_3	2	0	0	1	5	2
P_4	0	1	3	2	3	0
Total	6	9	6			
Avai.				2	3	0

（3） Find a safe sequence: $(2,3,0) \rightarrow P_4(2,4,3) \rightarrow ?$

（4） Unsafe $\rightarrow$ No!

例 25. *Consider the following system snapshot using the data structures in the Banker's algorithm, with resources A, B, C, and D, and processes P_0 to P_4:*

	Max				$Allocation$				$Need$				$Available$			
Processes	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	6	0	1	2	4	0	0	1								
P_1	1	7	5	0	1	1	0	0								
P_2	2	3	5	6	1	2	5	4								
P_3	1	6	5	3	0	6	3	3								
P_4	1	6	5	6	0	2	1	2								
Total													3	2	1	1

（1） How many resources of type A, B, C, and D are there?

（2） What are the contents of the Need matrix? Fill in the above table.

（3） Is the system in a safe state? Provide your reasons.

（4） If a request from process P_4 arrives for additional resources of $(1,2,0,0)$, can the Banker's algorithm grant the request immediately? Show the new system state and other criteria.

Keys:

（1） How many resources of type A, B, C, and D are there?

- 9,13,10,11

（2） What are the contents of the Need matrix? Fill in the above table.（太简单了，跳过）

（3） Is the system in a safe state? Provide your reasons.

- Yes, $P_0(7,2,1,2) \rightarrow P_2(8,4,6,6) \rightarrow P3(8,10,9,9) \rightarrow \cdots$

（4） If a request from process P_4 arrives for additional resources of $(1,2,0,0)$, can the Banker's algorithm grant the request immediately? Show the new system state and other criteria.

- This state is NOT safe. P_0 can be satisfied, but the available resources at that point $(6,0,1,2)$ cannot satisfy the needs of the remaining processes.

3 Concepts Organization

1. Deadlock: A set of processes are all waiting for each other to release resources, and none of them can proceed.
2. Four necessary conditions: Mutual exclusion, hold and wait, no preemption, circular wait.
3. Resource-allocation graph: A directed graph that represents the allocation of resources to processes and the requests made by processes for resources in a system.
4. Deadlock prevention: A strategy that ensures the system will never enter a deadlock state by eliminating one or more of the necessary conditions for deadlock to occur.
5. Deadlock avoidance: A dynamic strategy that checks the current resource-allocation state before granting a request to ensure the system remains in a safe state.
6. Deadlock detection: A strategy that allows deadlocks to occur, but the system periodically checks for deadlock conditions and takes action when one is detected.
7. Deadlock recovery: A set of techniques used after a deadlock has been detected, to restore the system to a normal, non-deadlocked state.
8. Safe state: A state of a system in which there exists at least one sequence of process execution such that every process can finish execution.
9. Resource-allocation graph algorithm: A deadlock avoidance algorithm used in operating systems when there is only a single instance of each resource type.
10. The banker's algorithm: An algorithm that ensures that the system always remains in a safe state, where all processes can complete their execution without leading to a deadlock.
11. Wait-for graph algorithm: A deadlock detection method used in systems where each resource type has only one instance.
12. Detection algorithm: A method used in operating systems to determine whether a system has entered a deadlocked state.

第八章　Main_Memory

1　In-class Contents

Address binding 的功能是在把物理内存作为资源进行分割，保证 process 之间互不干扰，比如说每个 process 有自己的 0 号地址，这个是需要 OS 来管理的。那么如果说我的代码可以直接管理物理内存的内容，那这部分的代码完全不需要由 OS 管理。

同时有一个小插曲，我们现在绝大多数的 OS 用的都是 Paging，那么是不是说其实完全不需要 base and limit 呢？不是这样的，因为在启动的时候会有一小段代码需要用 base and limit 做简单的内存管理，同时，页表也需要 base and limit 去查。

1.1　Warm-up

Memory 有很多 API，包括 Stack Memory API 和 Heap Memory API。Stack 和 Heap 是 process 仅有的两种使用内存的方式。

```
1 void func() {
2         int x;
3         // Stack memory, 有生命周期, 由编译器隐式自动回收
4         int *x = (int *) malloc(sizeof(int));
5         // Heap memeory, 需要使用free()等去释放这些内存
              （灵活但需要小心管理）
6 }
```

我们再来看一段代码：

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "common.h"
5
6 int main(int argc, char *argv[]) {
```

```
7   int *p = malloc(sizeof(int));
8   assert(p != NULL);
9   printf("(%d) memory address of p: %08x\n", getpid(), (
        unsigned) p);
10  // (unsigned p表示p指向的地址)
11  *p = 0;
12  while (true) {
13    Spin(1);
14    *p = *p + 1;
15    printf("(%d) p: %d\n", getpid(), *p);
16  }
17  return 0;
18 }
19
20 /* For this example to work, address space randomization
     should be disabled */
```

现在有两种情况，一种情况是说只有一个 process 在运行这个程序，那么结果是这样的：

```
1 prompt> ./mem
2 (2134) memory address of p: 00200000
3 (2134) p: 1
4 (2134) p: 2
5 (2134) p: 3
6 ^C
```

还有一种情况是有两个 process 一起运行这个程序，那这个情况就比较复杂了：

```
1  prompt> ./mem &; ./mem &
2  [1] 24113
3  [2] 24114
4  (24113) memory address of p: 00200000
5  (24114) memory address of p: 00200000
6  (24113) p: 1
7  (24114) p: 1
8  (24113) p: 2
9  (24113) p: 3
10 (24114) p: 2
```

```
(24114) p: 3
...
```

从上面这段代码中，我们可以从两个进程的 memory address 都相等可以知道，这是 virtualization 在发挥作用，CPU 认为的地址其实是 virtual address，表示虚拟地址中堆的起始地址都是相同的。另外，我们可以从两个进程的 p 的改变情况可以得知，p 不是共享变量，两个 process 在改变各自的 p 的值。

1.2 Background

首先我们来讲一点背景知识：

（1） 程序从硬盘中加载到内存里面，然后 OS 为这个程序创建一个进程，然后 CPU 才可以调度它

（2） CPU 只能访问 Main Memory(only large storage) 和 Registers，其他的要借助 OS 等间接访问。

（3） Memory 会提供一些 interfaces(read(),write() 这种)，它做的事情只是按地址访问，只知道是读操作还是写操作，具体你做这个事情背后的原因它不管。

（4） 在一个 CPU cycle 中我们就能完成 register 的访问，非常快对吧

（5） 但是 Main memory 就不太行了，它相对比较慢，要很多 CPU cycle，就会导致 stall（滞留：CPU 在等待期间没法做其他事情）

（6） 啊哈，这时候我们就想着搞一个 Cache 吧，它的访问速度介于 main memory 和 CPU registers 之间，是不是就很棒呢。

1.2.1 Address Binding

在程序的不同阶段，有三种方式来表示地址：

（1） Symbolic（符号化）：在写源代码的时候，地址是符号化的，只是一个名字，没有具体的地址，比如说:”int count”，当我们对”count” 做 ++ 操作的时候，本质是对地址上的这个 value 进行操作。在这种情况下，我们通过符号表达去表示内存当中的一个具体位置。

（2） Logical（不知道具体的位置，但是知道逻辑地址）：这个时候，我们编译一下源代码，它就成为了可重定位的地址 (relocatable addresses)，这时候就会有一个可执行文件，例如 exe,img（镜像文件），当然可执行文件不是这里的重点。

比方说，我们再详细地说一下 count 的位置”14 bytes from beginning of this module”，那么我们拍脑袋一想就知道这是个相对地址，不是绝对地址。

（3） Physical：那么最后，我们还是要找到 physical memeory 的对不对，其实也非常简单，当我们把 image file 或者是这个 exe 文件加载到内存的时候，count 的位置现在就是 Physical address，也就是绝对位置，比如说是”74014”。

那么在这个过程中，我们可以知道每次发生 bind 的过程当中，就会有地址从一个地址空间映射到另一个地址空间 (Symbolic->Logical->Physical)。

1.2.2 Address Binding Time

那么我们知道了什么是 Address Binding 之后，我们应该什么时候做 Address Binding 呢？实际上，我们有下面三种情况：

（1） Compile time（编译时）：如果程序在编译时就已经知道将来会加载到内存中的具体地址，那么编译器可以**直接生成“绝对地址”的代码**。一旦内存位置发生变化（比如换个地方加载程序）就必须重新编译，因为地址已经写死在机器码里了。

例 26. 刚刚启动电脑的时候，会有些程序需要直接操作物理地址；比如说，在开机的时候，有一个*”Hello，这是 rgl 的计组课”*，那么这个内容是在固定的物理地址的。

（2） Load time（加载时）：如果编译时还不知道程序将来会被加载到哪个内存位置，编译器会**生成可重定位代码（relocatable code）**。等程序被加载进内存时，由加载器（Loader）把这些相对地址转换成具体的绝对地址。**程序加载进去地址就不变了！**

例 27. 只要不需要 *dynamic loading* 的情况都是。

（3） Execution time（运行时，最常见，也最灵活）：进程在运行时可能被移动到不同的内存位置，地址绑定必须等到程序真正运行时才完成。

例 28. $cout << ”Hello, world”;$

在这种情况下，可能需要硬件的支持来完成地址映射，比如 (base and limit registers，基址寄存器、界限寄存器)

那么为什么会有在 Execution time 的时候才绑定地址呢？其实就是因为，**现代计算机只能 load program 的一部分进到内存当中，所以只有在执行阶段才能知道具体的地址。**

1.2.3 Dynamic Loading & Linking

首先补充一个知识，program 的运行是 compile->link->load 的过程。

我们主要讲讲动态加载和动态链接之类的概念：

（1） Dynamic loading: 程序在启动时不会一次性把所有代码都加载进内存。某个函数或模块只有在被调用时才从磁盘加载到内存。

（2） Static linking: 编译链接时，所有用到的库文件和程序模块都被打包进一个最终的可执行文件里。一旦生成，就不依赖外部库文件了。缺点是文件体积大、更新库必须重新编译程序。

（3） Dynamic linking: 程序中调用的库函数不会在编译时直接包含。它们在程序运行时由操作系统在内存中动态链接。

比方说，Windows 会弹出来一个错误，叫做系统缺少某个 dll(dynamic link library) 所以游戏不能玩，这个时候我们还会用到一个东西，叫做 stub。

定义 1.1. *Stub*（存根）：一个很小的占位代码段。在使用动态链接时，程序中原本调用库函数的位置会先放一个 *stub*。程序运行时，这个 *stub* 会负责：查找库函数在内存中的位置；如果还没加载，就触发加载；然后跳转到真正的函数实现。

1.2.4 Address Space

在现代操作系统中，程序运行时使用的地址（逻辑地址）和实际内存中的地址（物理地址）是分开的；所以说把这两者绑定（address binding）在一起的过程，是实现内存管理的核心。

我们的 Address 有下面两种：

（1） Logical address(Virtual address): 由 CPU 产生，是在程序当中看到的地址。（可以看 warm-up example）

（2） Physical address: 内存芯片真正看到的、实际使用的地址。我们之前还说了 Memory 做的事情是按地址访问，执行读写操作，这个硬件接口在使用的时候，用的就是 Physical address（也就是 Absolute address）。

与此同时，在 compile-time address-binding 和 load-time address-binding 的情况下，logical address 和 physical adress 的完全一样的。只有在 execution-time address-binding 的情况下，两者的地址是不一样的，所以才会需要有一些算法来实现 Logical 和 Physical Address 之间的 Translate。

拓展. 为什么说 *Load-time binding* 的逻辑地址和物理地址也是一样的呢？因为它们加载到内存之后就不会发生改变了。

因为有这两类的 addresses，所以呢，也会有下面的两种 address space：logical address space（程序可以生成的逻辑地址的集合，比如说我是 32 位的机器，那么理论上就能生成 0 到 $2^32 - 1$ 的逻辑地址）和 physical address space（实际上能访问的物理地址，如果说机器是 8GB，那么物理地址空间就是 0 到 8GB-1）

1.3 Address Translation

好，那么好，我们之前说当我们使用的是 execution-time address-binding 的时候，需要做 address translation 对吧，那么我们现在就来讲讲具体是怎么 translation 的。

我们先来看这样一个例子：

```
1 void func() {
2         int x = 3000;
3         x = x + 3;
4 }
```

那么汇编语言是什么样的呢？（当然从计组当中的指令系统中我们会知道，这些在内存中都是一系列的 01 串）

```
1 128:    mov 0x0(%ebx), %eax     ; load 0+ebx into eax
2 132:    add $0x03, %eax         ; add 3 to eax register
3 135:    mov %eax, 0x0(%ebx)     ; store eax back to mem
```

我们现在来研究一下这背后具体一步步是怎么做的，假设 x 存在内存为 0x1000 处，%ebx 寄存器的值就是 0x1000：

（1） 获取地址为 128 的这个指令

（2） 执行这个指令：将地址为%ebx，也就是 0x1000 的数据加载到寄存器%eax 当中，所以现在%eax 寄存器的值为 3000

（3） 再获取地址为 132 的这条指令

（4） 执行这个指令，把%eax 寄存器的值加 3

（5） 最后获取地址为 135 的这条指令

（6） 执行这个指令，把 3003 写回地址为%ebx，也就是 0x1000 的地方

1.3.1 Base and Limit

书接上文，有一个问题啊，就是 CPU 只看到了三条指令的地址是 128，132，135，但这些地址都是 logical address，那我们怎么去找到物理地址，然后让 process 察觉不到呢？

那就是用 base and limit registers（基址寄存器和界限寄存器），用他们来定义程序的“逻辑地址空间”在物理内存中的位置和大小。

注意. *那么每次在 user mode 访问内存的时候，出于 protection 的考虑，CPU 会检查一下这样做是否合法，如果合法，那就可以通过下面的公式把物理地址算出来（这件事情是由硬件实现的，没有指令显示做这个事情）：*

$$physical\ address = virtual\ address + base$$

1.3.2 Memory Management Unit

base 和 limit 寄存器都是存在 chip 里面的寄存器，一般性每个 CPU 有一对。

下面讲一个很重要的概念：

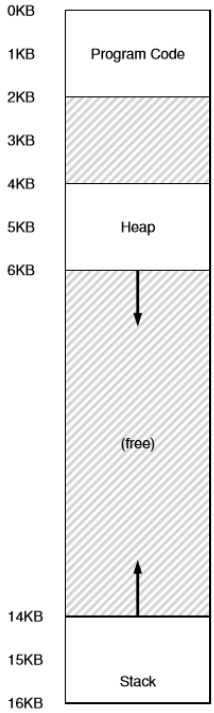
定义 1.2. *MMU: 负责完成地址翻译（逻辑地址-> 物理地址）以及检测是否越界（是否小于 Limit）*

1.3.3 Summary

这里又是一个很重要的重点!!!

Base-and-limit 能够有这些好处：Transparency（process 不会感知物理地址的变化）、Protection（Limit 会做保护，保证你不会出界）。那么有 Efficiency 吗？在时间上确实是这样的，因为全都由硬件完成，但是在空间是不是这样的，存在很多空间浪费，我们可以从下面这张图来看看。

这个问题被称为 Internal fragmentation，也就是说内部存在无法被使用的空间碎片，因为要加载一个 process 的时候，必须要给它分配 heap、stack 的空间，但是因为 heap、stack 的大小不确定，这个时候就只能尽可能开更大的空间来保证 process 的运行，那这样就空间浪费了嘞。



1.4 Segmentation

由于时间原因，只能将下面的重点内容分条陈述：

（1） 重要思想就是每个 segment 都有自己的 base 和 limit，同时有自己的 data, code, heap, stack 这种，而不是在整个 MMU 里面只有一对 base and bounds.

（2） 如果你试图访问一个 illegal 的 address，那你就会引发 segmentation fault。

（3） 当 OS 要做 context switch 的时候，base and limit 的信息也要存起来

（4） Segmentation 能解决 internal fragmentation 的问题，但会产生 external fragmentation 的问题。

1.5 Free Space Management

在分配空间的时候，我们会需要做 malloc() 和 free() 对吧，那么现在如果我有 5MB 的空间，申请了 3MB，就会把空间分成 3MB 和 2MB。如果那 3MB 的空间用完了，又会和边上的 2MB 重新合成 5MB，这很好理解。

在分配空间时，我们有四种方式：

- Best-fit(smallest fit)：找满足条件最小的 free space
- Worst-fit: 找满足条件的最大的 free space

- First-fit：找第一个满足条件的 free space
- Next-fit：从上一次被分配空间的地方开始，先看那个地方能不能分配，如果不能就到下一个 free space 找

注意. *First-fit 和 best-fit 表现都比 worst-fit 好，不管从时间角度还是空间角度。但是从空间利用角度，first-fit 和 best-fit 是差不多的，而且 first-fit 还更快。*

1.6 Paging

（1） Paging: 物理地址可以看作是一个数组，数组的内容是固定大小的 slots，被称为 page frame。

（2） 有什么好处呢？Flexibility（虚拟地址和物理地址都可以是零散的）和 Simplicity（找一个 page frame 非常快）。

（3） Physical Frame Number(PFN)

（4） Virtual Page Number(VPN)

（5） Page Table 就是 VPN 和 PFN 的映射，一般每个 process 有一个 page table（里面的内容被称为 page table entry，就是一堆二进制数）

（6） Page Table 不存在 MMU 里面，存在 kernel physical memory 里面

（7） Page-talbe base register 指向 page table 的为止，Page-table length register 表示 page table 的大小。这俩玩意是在 MMU 里面的。

（8） 那么 PTE 里面是什么东西呢？**注意，不需要 VPN**，因为在 virtual address 本来就是顺序的，所以 PTE 里面会有 PFN、valid bit（访问一个 valid bit 为 0 的页，可能就 page fault 了）、present bit（这个 page 是不是在 memory 里面）、dirty bit、reference bit(accedded bit)（这个 page 有没有被读/写过、LRU）、一些和缓存相关的 bits

1.7 Translation Lookaside Buffer

别告诉我这玩意你忘了！TLB！

现在的问题就是，pagign 有点慢，因为对于 segmantation 来说，CPU 直接用 base and limit 就解决了，paging 要查表，表还在内存当中，这非常不好！

（1） TLB 是 MMU 的一部分，它做的事情其实就是缓存，存最近的映射关系

（2） 由操作系统来处理 TLB missing-handling 的事情，因为操作系统可以选择任何适合的页表数据结构（链表、树、哈希表等）。这样硬件不必固定为某种结构，系统设计更灵活。

（3） TLB 和 page table 不太一样，它必须要 record VPN，原因也很显而易见。

（4） 但是有一个问题啊，TLB 服务很多 processes，如果你做了 context switch，理论上你的 VPN 和 PFN 就对不上了，有两个方式，一个方式是重写你的 TLB，当然这很费时间，还有一种方式就是加一位 ASID(Address space identifier) 在辨认你的这条映射是属于哪个 process 的。

（5） 当然，可能会出现两个 process 的某个 VPN 映射到同样的 PFN 上，这很好，可以共享代码。

（6） Effective Access Time(EAT)：这东西很重要，但是它不考 (LOL)

这个东西，very very very important!

那么我们怎么去选择 page size 呢，我们希望它是大点好还是小点好呢（虽然说到底，一般的 page size 都是 4KB）?

（1） 希望 small page:

Fragmentation（很好理解，让 internal fragmentation 最少）、locality/resolution（page 越小，越好刻画局部性和分辨率）

（2） 希望 large page:

Table size/page faults（很好理解，large page->less pages->less PTE->less possibility of page fault）、I/O overhead（这个不太好理解，首先 I/O 指的是内存和 disk 之间的数据交换，然后在 disk 当中，顺序读写的速度会远超随机读写，所以对于同样大小的空间，读一个大的 page 比读分散的小的 page 要快很多）、TLB reach/TLB size（TLB reach 指能 cover 多少内存，page 越大，cover 的肯定也就更多、TLB size 和 table size 同理）

1.8 Structure of the Page Table

（1） 核心思想就是，本来一个 process 一个 page table，现在根据它的逻辑段，一段一个 page table，那可能这个 process 有 code section、stack 啥的，最后一个 process 就对应了几个 page tables。

（2） 可以理解为，一个 process 里面的每个 segment 都有自己的 base and limit，还有自己的 page table。

（3） 但是现在问题也出现了，因为每个 segment 大小不一样，所以分配的 page table 的大小也不一样，那么其实提高了 OS 管理这些 page table 的难度。

（4） 那么为了解决这个问题，我们提出了 multi-level page tables，也就是用 paging and paging，核心思想就是，把有用的 entry 留下，没用的“扔掉”（也就是说明它是 not allocated 的）

（5） Page directory table(outer page table): 包含一系列 page directory entries(PDE)，每个 page 有这样一个 entry，PDE 里面含有 PFN 和一个 valid bit，那个 valid bit 为 1 就表示 page table 中至少有一个 page 是有内容的。

（6） 在 page 套 page 的模型当中，一个页是 4KB，对应了 1024 个 PTE。同时，base and limit 不再直接指向 Linear Page Table，而是指向 Page Directory Table。

（7） Page directory table 的一个 entry 指向一整个 page，也就是 1024 个 PTE。

（8） 但是这里有个问题，如果说我们没有在 TLB 找到，就要先访问外部表，再访问内部表，最后找到对应的内存：Time-Space Trade-off。

2 Exercise

3 Concepts Organization

1. Program loading and linking: The process of bringing a program into memory (loading) and resolving symbol references (linking) to make it executable.
2. Address binding: The mapping of symbolic or logical addresses to physical memory addresses.
3. Logical address space: The address space as perceived by a process.
4. Physical address space: The actual locations in main memory where data and instructions reside.
5. Swapping:
6. Base and limit:
7. MMU:
8. Address translation:
9. Free space management:
10. Segmentation:
11. External fragmentation: Total memory space exists to satisfy a request, but it is not contiguous.
12. Paging:
13. TLB:
14. Page size issues:
15. Internal fragmentation: Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
16. Paging and Segments:
17. Hierarchical page tables:
18. Hashed page tables:
19. Inverted page tables:

第九章　Virtual Memory

1　In-class Contents

1.1　Warm-up

定义 1.1. *Segmentation fault: 访问了不该访问的内存而引发的错误*

（1） free list 用于追踪哪些内存还没有被使用过

（2） OS 为了启动 process，需要先为其分配内存，再 set base/limit 寄存器

（3） 硬件用 base and limit 寄存器来翻译虚拟地址来取指令和完成 load/store 操作

（4） 如果要从一个 process A 切换到另一个 process B，OS 会把 A 的数据（包括 base/limit 寄存器的内容）存到 proc-struct(A) 当中，并将 B 中的内容都取出来

（5） 如果说 process 访问内存的时候出现 out-of-limit（由硬件检查，一般是 MMU），那么 OS 就会执行 trap handler，并回收 B 的空间

1.2　Background

一个现实问题：王者荣耀的容量的很大的，而且其中可能会有不常用的路径和大型数据结构（梦境大乱斗、人机对战），这个时候我只是想玩排位赛，可以让其他的内容自己去下载。

Benefits:

（1） 我的程序不会因为物理内存小而被限制

（2） 每个程序需要的内存更少-> 我就可以加载更多的程序-> 增加 CPU 利用率和吞吐量，同时响应时间并没有增加

（3） 更少的 I/O 切换操作（因为在物理内存中放的都是常用的部分，而且量比较少，所以不需要很多切出来切出去）

定义 1.2. *Virtual Memory*：介于 *Logical Memory(由 CPU 生成)* 和 *Physical Memory* 之间的存储层级，与 *Logical Memory* 很相似

辨析. 区分 *Virtual Memory* 和 *Logical Memory*：

1. *Logical Memory* 是 *CPU* 看到的，用户程序使用的；而 *Virtual Memory* 是硬件向下实现的部分，也就是 *Logical Memory* 的底层实现
2. *Logical Address = Virtual Address*（因为他们都用的是 *Hardware Interface*）

（1） 我每次只需要加载程序的一部分

（2） **逻辑地址空间远大于物理地址空间**

（3） 多个程序共享地址空间

（4） 允许更多程序并行运行

（5） 需要更少的 I/O

那我这时候想要 swap 呢？（通过 Backing store，也就是 disk）这样我就可以玩王者荣耀的其他模式了。

定义 1.3. *Memory Map*：将 *Virtual Memory* 映射到 *Physical Memory->Address Translation*

1.3 Swapping Mechanisms

定义 1.4. *Swap Space*：预留给 *backing store* 的空间，一般情况下，物理内存如果是 *4G*，*swap space* 要是物理内存的两倍，也就是 *8G*

OS 可以按照 page 的大小来访问（读/写）swap space，同时需要记住每一页对应的 disk 中的地址（这样才可以进行 swap 操作）

定义 1.5. *Present Bit*（在 *Page Table Entry* 里面）：如果是 *1*，那么 *page* 就在 *physical memory*，如果是 *0*，就在 *disk* 里面

定义 1.6. *Page Fault*：要找的 *page* 不在 *physical memory* 里面

辨析. *Present bit = 0* 只表示 *page* 不在 *memory*（*call page fault->invoke Page fault handler*，*handler* 来 *swap*）；但是 *Valid bit = 0*，都不在 *backing store* 里面（*call segmentation fault*）

Page fault handler 具体做什么事情呢？

（1） 找一个空的帧（frame）

（2） 把我要的 page 换进去（OS 是知道 page 在 disk 中的位置的）

（3） 将 Present bit 置为 1

（4） 重新运行导致 Page fault 的那个指令

1.4 Page Replacement

OS 要关注的两个事情：为每个 process 要分配多少 frame；当 frame 都用完了，应该怎么去做 Page replacement，使得 Page fault 能够降到最低？

定义 1.7. *Modify(Dirty) bit*：*Page* 被修改过了，那么在调出 *frame* 的时候需要写回 *disk*，如果没有被修改过，直接丢弃就可以了

Page replacement 能够做到通过 swapping，将很大的 Virtual memory 映射到较小的 Physical memory。

Basic Page Replacement 思想：

（1） 找到 disk 里面我要的 page 的位置

（2） 找到一个空闲的 frame（如果有那正好，如果没有，就要想办法替换；如果被替换的 frame 是 dirty 的，那还要先写回 disk 一下）

（3） 把我要的页面放进腾出来的空 frame，更新页面和 frame tables

（4） 重新运行导致中断的指令

注意. 一次 *Page fault* 会牵扯到两个 *pages*

1.4.1 Page-replacement Algorithm

目标：给定了每个 process 的 frames，想要降低 page fault

定义 1.8. *Reference string*：

- 只是 *page number*，不是 *address*!!!
- 如果 *string* 连续两个相同的字符，要删掉一个
- 算法 *page fault* 的结果和 *frames* 的数量有关（注意：并不是 *frame* 越多越好，后面会提到）

例 29. *7,0,1,2,0* 是一个 *reference string*，同时 *7,0,1,2,0* 和 *7,0,0,1,1,2,0* 是一样的，算 *page fault* 的时候记得先把重复的数字删掉

1.4.2 Optimal Algorithm

也被成为是 Belady's MIN algorithm，核心思想是替换掉最长时间才会用到的那个 page number（check the future）。

但是实际上我们并不能预知未来，所以没法用 Optimal Algorithm，我们更多用这种算法作为一种 benchmark 或者 baseline。

1.4.3 FIFO Algorithm

核心思想：先来的先被替换，用一个队列来追踪要被换到的 pages

注意. *Belady's Anomaly*（异常）：增加更多的 *frame* 反而可能导致 *page faults* 更多。（我们学过的算法里面，只有 *FIFO* 有这个问题）

例 30. *1,2,3,4,1,2,5,1,2,3,4,5 for 3 frames*（*9 faults*）*vs. for 4 frames*（*10 faults*）

1.4.4 LRU Algorithm

核心思想：替换掉最远使用的那个；是 most common use 的 algorithm

两种实现方法：

（1） Counter implementation：每个 page entry 有一个计数器，来记录 page 被换入到 frame 的时间，如果需要替换，找到那个数据最小的数（Confusing，我的理解是一开始给一个大的数，然后一点点减这个数）去替换掉

Problem：每次找到我想要替换的 page 很慢，因为要把每个 counter 遍历一遍

（2） Stack implementation：用一个双向链表来记录放进去的数

核心思想：每当有一个 page 被 refer 了，将这个 page 放在链表 head 的位置，替换掉的是原链表 Tail 位置的 page。

Problem：每次有 page 进来，要改变 6 个指针的值，那么维护 stack 是比较慢的，但是找那个 page 很快。

考点 13. *LRU* 和 *OPT* 算法是基于 *stack algorithms* 的算法，不会出现 *Belady's Anomaly*。（也就是说，只要是基于栈实现的，都不会有这个异常问题）

证明. *The set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames.*

1.4.5 LRU Approximation Algorithms

想要解决的问题：LRU 需要特殊的 Hardware support 并且还是比较慢，所以希望用近似算法来做。

定义 1.9. *Reference Bit*：表示这个 *page* 是否被访问过，*0* 就是“没有”被访问，*1* 就是被访问过了

在进行替换操作的时候，将 reference bit = 0 的替换掉，新加入的 page，设置其 reference bit = 1，问题在于这个方法没办法知道具体是谁更早被访问。

注意. *Reference bit = 0* 只是表示其近期没有被访问过，具体怎么变化的见 *Second-chance* 算法

定义 1.10. *Additional-Reference-Bits Algorithm*：用 *8* 个 *bit* 来表示 *Reference*，这样就能知道对于一个页面，谁更早被访问，谁更晚被访问

每次操作，将所有的 bits 往右移动一位，如果这个 page 被访问了，最高位置 1，否则为 0；在替换的时候，每次替换掉真值最小的那个 page。

例 31. *P1:00100100,P2:00100000*；*Obviously, P1* 比 *P2* 对应的值更大，所以要替换的话就替换 *P2*。

考点 14. *Second-chance (Clock) Algorithm*：*FIFO* 的思想，但是会循环遍历 *pages* 去找我要替换的 *page*，首先看 *Reference bit* 是否为 *0*，如果是，直接替换掉；如果为 *1*，将其至为 *0*，相当于给它第二次机会。

考点 15. *Enhanced Second-Chance Algorithm*：相比 *second-chance* 只用一位的 *reference bit*，这里用一对 *bits*（*reference, modify*）

替换顺序：*(0,0)->(0,1)->(1,0)->(1,1)*

核心思想：越不常用、且没有被修改的页面，越适合被替换掉

注意. 为什么会有 *(0,1)* 的被替换，因为 *reference bit = 0* 代表没有被访问，*modify bit = 1* 表示被修改了。

Answer：只是最近没有用，*modify bit* 从 *1* 变为 *0* 了，不代表说这个 *page* 从未被使用过。

1.4.6 Other Algorithms (Less Important)

（1） Counting Algorithms（Least Frequently Used & Most Frequently Used）

（2） Page-buffering Algorithms：

核心思想：一直保留空的 frame，这样可以直接把 page 读入，同时**方便的话，换出一个**

改进 1：记录哪些页面被修改过，这样当 backing store 空闲的时候，可以写回 disk，并将页面设为未被修改过的。

改进 2：不立即把空白帧的内容给清空掉，记录它装的页面内容

1.4.7 Conclusion & Comparision

所有的 OS 都会去“猜”未来可能的 page，但是可能猜错；而数据库能够帮助优化页面管理

定义 1.11. *Double Buffering*：*OS 有一份 page 的内容用于 I/O，同时 Application 有一份 page 的内容用于自己的工作-> 不必要的内存占用和冗余操作。*

例 32. *在放视频的时候，OS 为播放器分配内存缓冲区，存放视频数据，而应用程序（播放器）也需要将当前帧放在 bufferA，下一帧放在 bufferB-> 同一份数据被放在两个不同的地方。*

定义 1.12. *Raw Disk Mode*：*Applications 不经过 OS，直接获取 page 的内容。*

两个 Benchmark：一个之前提到的，是 OPT，另一个是 Random Algorithm（随便换）。

比较不同的 Page replacement 算法：

（1） The No-Locality Workload（page 随机到来）：OPT > RAND = LRU = FIFO

（2） The 80-20 Workload（80% 的访问集中在 process 里 20% 常用的 pages，另外 20% 的空间给 process80% 不那么常用的 pages）：OPT > LRU > CLOCK(和 LRU 差不太多)> FIFO = RAND

（3） The Looping-Sequential Workload（循环访问页面）：如果 frame 不够多，OPT > RAND > FIFO = LRU = 0；如果 frame 足够多，OPT = RAND = FIFO = LRU = 100%

1.5 Allocation of Frames

我们要解决的问题：我们要给每个 process 分配多少 frames。

每个 process 有最少需要的 frames 数量，比如 IBM 370 至少要 6 个 pages 才能实现 MVC 指令；frames 的最大值是整个系统所有的 frames。

1.5.1 Fixed Allocation

（1） Equal Allocation：均分

（2） Proportional Allocation：按 process 的大小的比例来分

1.5.2 Priority Allocation

根据 process 的优先级来分配，本质也是一种 proportional allocation，只是一个根据优先级来，一个根据 process 的 size 来。

1.5.3 Replacement Strategy

（1） Global replacement：从所有的 frame 里面去找一个 frame 替换，也就是不同 process 之间可以抢 frame 来用

Pros：更大的吞吐量

Cons：Process 自身无法控制自己的 Page fault

（2） Local replacement：每个 process 只能从分配给自己的 frame 里面去替换

Pros：表现更加 consistent

Cons：有的时候不需要那么多帧，那就有内存没被用上

1.5.4 NUMA:Non-Uniform Memory Access

CPU 访问 Memory 的速度也会有优先级，设置 lgroups（内存低延迟组），这样“离 CPU 更近”的内存会被更快访问。

1.6 Thrashing

定义 1.13. *Thrashing*（抖动、颠簸）：*A process is busy swapping pages in and out.*

例 33. 如果一个 *Process* 没有足够的 *page*，那么 *page fault* 会一直很高，所以就需要不停的从 *disk* 里面拿 *page* 来做 *replacement*，然而 *replacement* 的动作是一个 *I/O* 操作，这就导致了本来一个 *CPU-Bound* 的进程，因为频繁的 *I/O*，导致 *CPU* 一直空闲，操作系统认为它是一个 *I/O-Bound* 的进程，就会增加 *multiprogramming* 的程度（*Completed by long-term scheduler*）。那么系统添加的 *process* 变多了，每个 *process* 分配的 *frame* 更少了，那 *page fault* 又变多了，**凉性循环**。

所以说，当多道程序的程度增加，CPU Utilization 一开始是会增加的，直到发现没有 enought frames，CPU 利用率就减少了，开始**颠簸**。

定义 1.14. *Locality Model*（局部性模型）：*Process* 从一个局部性迁移到另一个局部性，且局部性之间可能有 *data* 和 *statements* 的重叠。

例 34. 程序执行到函数 *A*，访问 *A* 中的数据，之后执行到 *B*，就要访问 *B* 中的数据，这个过程就说局部性被迁移的过程。

理解：*Process* 的执行不是均匀访问整个内存，而是阶段性访问内存其中的一部分数据和指令。随着程序逻辑推进，不断迁移访问新的局部区域。

For Example：王者荣耀把游戏分为场景模块和战斗模块，这就是两个 *Locality*。

那么 Thrashing 是怎么发生的呢？说白了就是 Locality 的大小总和超出了系统的物理内存-> 我们怎么做？用 local or priority page replacement（个人理解：local 对应 PFF，priority 对应 working set）。

1.6.1 Working-Set Model

核心思想：看最近 Δ 时间的页面访问（被称为 WSS：working set size of Process P_i），来判断一个进程目前真正需要哪些页面。（Δ：working-set window）

如果 Δ 很小，那可能没法包含所有的局部性信息；Δ 很大，可能包含多个局部性；Δ 是无穷，那么包含整个 program，我要它也没用，失去局部性的概念了，所以 Δ 要合理选取。

定义 1.15. *Total Demand Frames(D)*：把所有的 *WSS* 加起来，表征 Δ 时间内系统所需要的页面需求量。

如果 D>m（物理内存大小）-> 寄了孩子，你要 thrashing 了

好，那么好，怎么解决这个问题？

Working-Set Model 给出了一个方案：如果 D>m，就把一些 process 给挂起来或者换出，来减少 the degree of multi-programming（这个事情是由 medium-term scheduler 完成的），同时，被换出去的 process 的什么状态（比如说是 ready）状态，之后换进来的时候也是什么状态。

现在我们遇到了第二个问题：精确记录每个页面是否在 Δ 时间内被访问的开销是很大的，所以我们希望找一个近似的算法来计算 WSS，同时不要很大的开销。我们采取的方法是通过 interval timer interrupt 和 reference bit 来做这个事情。

例 35. 假如 Δ=*10000*，我们可以每 *5000* 和时间单元中断一次，我们为每个 *page* 分配两个 *bit*，一个 *bit* 表示前 *5000* 时间单元它有没有被访问，另一个 *bit* 表示后 *5000* 时间单元它有没有被访问。那么我怎么来确定 *bit* 是 *1* 还是 *0* 呢？看 *page* 的 *reference bit*，如果前 *5000* 时间单元发现 *reference bit* 为 *1*，那么第一个 *bit* 就被置为 *1->(1,0)*，再把 *page* 的 *reference bit* 置 *0*；如果在后 *5000* 时间单元，*reference bit*

还为 *1*，那么第二个 *bit* 也为 *1->(1,1)*。不管最后这两个 *bit* 有几个 *1*，只要有一个，那么就可以说明这个 *page* 是在 *working set* 的。

注意. 这个方法是不够精确的，因为每 *5000* 时间单元采样一次，分辨率是比较低的 *(resolution)*，那么我们可以改进一下，比如说每 *1000* 时间单元采样一次（同时为每个 *page* 分配 *10* 个 *bits*），这样分辨率会高一些。

1.6.2 Page-Fault Frequency

核心思想：我去算一个 process 的 page-fault frequency(PFF)，然后设定一个 upper bound 和一个 lower bound，用 local replacement policy 来调整这个 process 分配的 frame。

这种方法比 WSS 更加直接，因为如果 PFF 很高，说明 process 缺页；如果 PFF 很低，说明 frame 很多，那么内存资源就被浪费了，系统的并发程度不高，那就减少 frame 的数量。

1.6.3 Conclusion

对于 PFF 方法来说，它通过调整单个进程的 frame 数量来尽量避免 Thrashing，而 WSS 则是通过调整系统运行的 process 数量来尽量避免 Thrashing。

他们之间也是有联系的，如果 working set 大，那么活跃的页面就多，很可能 frame 会不够，那就需要分配更多的 frame，如果分配多到能够覆盖 working set，PFF 就会下降（同时注意不能分配太多 frame，会导致 concurrency 下降的）。

1.7 Other Concepts & Issues

1.7.1 Demand Paging

常规情况下，我们是要把整个 process 加载到内存的，但是为了节省空间，我们使用“按需分页”的方式，当我们需要的时候，把一个页拿进内存。

这样做的好处有：

（1） 会减少 I/O，不会有没用的 I/O（这里听上去会容易疑惑，因为一个一个 page 搞进来会发生很多 page fault，导致更多的 I/O，但是，实际上如果把整个 process 搞进来，用的 I/O 更多）

（2） 需要更少的内存

（3） 响应更快（程序更早开始执行，不用全都加载进来）

（4） 容纳更多的用户（显然，并发率提高）

当我需要一个 page，首先看是不是无效的 reference，如果是无效的，直接 abort the request；如果是有效的，并且这个 page 不在 memory（很可能是因为这个 page 第一次被需要），这个时候就 page fault 并把这个 page 搞到内存中。

定义 1.16. *Lazy Swapper*：从来不会把 *page swap* 进内存，除非这个 *page* 被需要

似乎会增加一些 *page fault*，但可以大大减少不必要的内存加载和 *I/O* 操作。

定义 1.17. *Pager*：*Swapper that deals with pages.*

我们再深入探讨一下这种“swap”方式：

除了交换内容之外，Demand paging 有三个主要的工作，一个是需要 page-fault handler 来处理中断，这个操作相对是比较快的；把 page 搞进来，这是一个 I/O 操作，**是 Demand paging 最耗时间的一步**；还有一步是重新启动 process，也就是运行那个引起中断的语句，也不要啥时间。

在这种情况下，设 p 是 page fault rate，我们再来算一下这样情况下的 Effective Access Time：

$$EAT = (1-p) \times T_{memory\_access} + p \times (T_{page\_fault\_overhead} + T_{swap\_page\_out} + T_{swap\_page\_in}) \tag{9.1}$$

我们看 PPT60 页，会发现这样做的 EAT 还是很大的，所以我们希望能够改进一下这个算法。

这时候我们做一件事情，就是 Prepaging（预分页），先加载一些页面进来（当然不会把所有的 process 加载进来），目的是减少 page faults 的数量（可能加载一些关键的 page 吧）

同样也会有问题，如果预先准备的 page 没有被用到，那么同样 I/O 的时间被浪费了，空间也会被浪费。

那么我们要做的就说一件 Trade-off 的事情，我们要看到底是不用 Prepaging 时的 page fault 花的时间多，还是预先准备“没用”的 page 会花费更多时间。

下面的部分 dk 几乎很快带过去了，可能没有那么重要。

了解完 Demand Paging 的算法之后，我们尝试做一个优化：

定义 1.18. *Swap Space*：也是 *disk* 里面的存储，但是它的 *I/O* 速度比普通的 *file system I/O* 的速度要快，用大的块来存储，需要更少的管理

在进程启动，被加载到内存（可能是 Prepaging）的时候，这个进程的内容会复制一份到 swap space 里面，这样做的好处就是，我的 page in 和 page out 都可以在 swap space 里面做，就不用通过 disk 了。除此之外，如果要释放 frame 的时候，可以直接丢弃内存里 page 的内容（因为我有备份），但是注意丢弃之前要看一下是

不是要写回 swap space（比如堆栈这种和文件内容本身无关的内容或者是 page 被改过的内容）。

对于现在的系统来说，一般没有 swapping 的机制，用的是闪存。

现在我们再做一个优化，我们连页表都不全部加载进来，需要的时候再加载进来。

定义 1.19. *Inverted Page Table*（倒排页表）：不会有完整的 *page table*，完整的在 *disk* 里面，需要了再去找

例 36. 进程 *A* 有虚拟页号 *0,1,2,3*，这时候内存只加载了页号 *0,1*（说明倒排页表也只会有 *0,1* 页号到物理地址的映射），这个时候我要访问虚拟页号 *2*，首先 *2* 肯定不在 *physical memory*（$Page fault \times 1$），再去倒排页表里面找对应的映射，发现倒排页表里面也没有（$Page fault \times 1$），那只能去 *disk*（也就是 *backing store*）里面找有全部信息的那个页表。

1.7.2 Copy-on-Write

这是一个相对比较重要的概念，同时涉及 memory management 和 CPU management。

背景：有一个 process，它想要另一个 process 的 page，但是嫌 copy 太复杂太麻烦，就想让这两个进程指向同一个物理页。

定义 1.20. *Copy-on-Write(COW)*：这当 *OS* 需要将一个页面从一个地址空间 *copy* 到另一个地址空间的时候（典型情况：*fork()*），我们不是真正做一个复制，而是让两个进程指向同一个目标地址，并且让这个页面对于两个进程来说都是**只读**的。

但是啊，有个问题，如果有一个 process 的地址空间试图修改这个 page，那么 OS 就会意识到，并且只能真正地分配一块空间给你去修改。

那么 COW 有什么应用呢：

（1） 共享库：多个 process 共享一份只读数据

（2） 在 UNIX 系统，COW 允许父子进程共享内存的 page，分为两种情况：

fork()：有 COW；

vfork()：没有 COW，也就是子进程会直接共享父进程的地址空间，效率更高，但也更危险。

vfork 是 fork 的变种，它做的事情就是，在子进程生成之后，父进程挂起，子进程要立刻调用 exec()，把自己替换成全新的 process，一个和父子进程都没关系的 process，这样子才能保证父子进程的内容是不被修改的。

例 37. *UNIX* 的 *Shell* 在输入命令后，会调用 *vfork()* 在创建子进程。

1.7.3 Memory-Mapped Files

定义 1.21. *Memory-Mapping*（内存映射）：和 *swapping* 很像，本质做的事情就是使用 *mmap()* 来将一个已经打开的文件（例如 *stdin,stdout,stderror*）映射到 *process* 的虚拟内存，再返回一个指向文件在虚拟内存起始位置的指针 *p*

那它有什么好处呢？我们常规的例如调用 read()，write() 来对文件进行读写，本质输入 file I/O, 也就是和磁盘进行交互，那肯定是很慢的。现在我们用 Memory-mapping，把文件从 disk 映射到内存当中（一般是把一个 disk block 映射到内存的一个 page），这时候我们做的读写操作就是在和内存交互。

注意. 文件的读使用 *demand paging* 策略。

那么我们还可以知道，如果说我们要访问一个没有放进 memory 的文件的一部分，就会发生 page fault，之后我们就可以把对应的 disk block swap in 到内存，变成一个新的 page。同样，当我们要把这个 file 全部 swap out 的时候，再把内存映射的内容写回 disk。

1.7.4 Allocating Kernel Memory

背景：kernel 不知道用户是怎么使用 physical memory 的，只知道把空间分配出去就结束了，但是对于 kernel memory，kernel 是知道内存什么时候释放，是不是需要物理连续的。kernel memory 的空间一般来自 free-memory pool，也就是系统未被使用的内存区域（包括碎片去雨）。kernel 会申请不同大小的空间存放它的 data structures。同时有些 kernel memory 可能需要连续空间（例如 I/O，因为需要速率固定），这样不用 tranverse 就能直接定位到我要的内容。

现在，我们有两种方法来分配内核内存。

定义 1.22. *Buddy Allocator*（是所有内核内存分配的底层机制）：分配固定大小的段，里面是连续物理空间的页。

分配的大小只会是 2 的幂，具体怎么分配看 PPT68 页就行，非常容易理解。优势：能够快速合并小的块合成大的块。缺点：空间碎片化，多出来的会被浪费。

好，那么为了解决这样的空间碎片化的问题，我们提出了另一种内核空间的分配方式。

定义 1.23. *Slab Allocator*（现在更多说 *Slub Allocator*）：这种分配方法不会像 *buddy allocator* 给每个数据结构对应的 *object* 分配一块内存，而是按层级分成 *object->slab->cache*。

定义 1.24. *Slab*：是一个或者多个物理连续的页。

定义 1.25. *Cache*（只是一个名字，和 *CPU* 层级结构的 *Cache* 没有关系）：由一个或多个 *slab* 组成。

每个 Cache 存放一种 data structure，同时这个 Cache 存放的是这类 data structure 的所有实例。

例 38. 比如说这个 *data structure* 是 *PCB*（进程控制块），那么对应的 *Cache* 就会有系统所有的 *PCB*。

当 cache 被创建，里面填满空闲的对应的 objects；而当这个数据结构被存进来，对应的 object 被标记为 used。如果 slab 放满了 used objects，它就会开一个新的 slab 放新进来的 object，如果 slab 不够了，会分配新的 slab。

这么做的好处是不会存在碎片化空间，同时满足更快的内存请求（因为 kernel 知道每个 kernel data structure 的大小，在物理地址当中又是连续存放，可以用 index 来找，所以快）。

注意. *slab* 是连续的，*cache* 不一定，也就是说 *slab* 之间不一定物理连续；同时，一般情况下，*slab* 有空的地方优先填 *objects*。

那么把两个方法融合一下呢？就是原先是为每个 object 分配一块内存，现在是为每个 slab 分配一块内存，可以减少空间碎片化的程度。

定义 1.26. *I/O Interlock*：当执行 *I/O* 的时候，不能把正在执行的页面给 *swap out* 到 *disk* 里面去，需要把这些页面固定 *(Pin)* 在内存当中。

1.7.5 The Linux Address Space

分为 user portion 和 kernel portion，里面都是一些 code、heap、stack。kernel portion 里面就说一些 handler、system call 之类的。

那么为什么这么做呢？因为所有进程是共享一个内核空间（kernel portion）的，所以在切换进程的时候，user portion 的地址是要变的，但是 kernel portion 是不变的。

如果说我想要访问 kernel virtual pages 时，我需要 trap into the kernel（比如 read()，write()）然后将 CPU 切到特权模式。

两种 kernel virtual addresses（Difficult but useful）：

（1） Kernel logical addresses（内核最常用的虚拟地址），有以下这些特点：

使用 kmalloc 来分配这种类型的空间（类似用户态的 malloc()）

大部分 kernel 数据结构在这里（page tables, per-process kernel stacks），对于系统是很重要的数据结构。

不能被 swap 到 disk 里面（因为不能接收磁盘换入换出的延迟）

能够直接从虚拟地址映射到逻辑地址，有两个好处：1. 容易进行翻译；2. 方便用 DMA。

（2） Kernel virtual addresses：不是直接映射，使用 vmalloc 来分配空间，分配的量比较大，但是一般在物理上不连续，和用户层面的虚拟地址很像。

1.7.6 Page Table Structure

我们所知道的 page 都是 4KB 大小的，在 64 位操作系统当中，需要 4 次翻译才能找到目标 page，这是非常慢的。所以现在我们的电脑里面是有 4KB 的 page，也会有 2MB 的 page，这样子能让 TLB 表现更好，也需要更少的 page table entries。

1.7.7 Page Cache

和 kernel data structure 里的 Cache 很像，核心思想是把数据放在内存里面，这些实体被放在 page cache 的哈希表当中。

Page cache 的内容主要来自以下三个地方：

（1） memory-mapped files

（2） file data 和 metadata（元数据）（Chapter12,13 详细说明）

（3） 堆和栈

注意. 后面的内容都不是很重要，有个概念就行。

1.7.8 Page Replacement

问题：标准的 LRU 是比较有效的，但是有的时候表现会比较糟糕，比如说面对循环访问的大型文件的时候，可能比 RAND 糟糕。

这个时候，Linux 打算用两个队列来解决这个问题（本质上还是 LRU 的近似算法）。一个队列是不活跃队列，当一个 page 是第一次访问，就放在这个队列；另一个队列是活跃队列，当一个 page 被再次访问，那就放到这个队列里面。那么我们在做替换的时候，就会替换掉不活跃队列的内容。同时，Linux 也会定期把 page 从活跃队列里面搞到不活跃队列，保持活跃队列的大小大概是整个 page cache 的 2/3。

1.7.9 Security And Buffer Overflows

这里介绍了三种缓冲区溢出攻击和对应的解决方案。

```
1 int some_function(char*input){
2   char dest_buffer[100];
3   strcpy(dest_buffer,input);
4 }
```

例 39. 第一类问题：

问题声明：从上面的代码可以看到，*input* 的大小要是超过 *100*，那么 *buffer* 后面的内存区域就会被 *input* 的内容覆盖，恶意的程序员就可以在 *input* 里面加恶意数据来攻击。

应对方案：*NX(No-eXecute) bit in AMD*。这样做可以规定一块区域的代码是不可运行的（例如 *stack*）。

例 40. 第二类问题：*return-oriented programming(ROP*：返回导向编程，是很有名的攻击*)*

问题声明：这类问题源自 *return-to-libc*（*libc* 指的是 *C* 标准库）攻击。恶意的程序员做的事情是修改栈来改变 *return* 的地址，指向一个他们想要的指令（这些指令被称为 *gadgets*，和 *C Library* 相关）。

应对方案：*ASLR(address space layout randomization)*。具体来说，我们把代码、栈、堆的地址空间随机化，这样就能避免被攻击。

定义 1.27. *Speculative Execution*：*CPU* 会猜后面会执行的指令，并且会提前执行，那么在猜测的时候会用到 *cache*，不管有没有猜中，*cache* 的内容都不会被清理。

例 41. 第三类问题：*Meltdown and Spectre*（熔断和幽灵，也是很有名的攻击）

问题声明：正因为 *cache* 的内容不会被清理，所以这导致内存的内容会很脆弱，容易受到攻击。

应对方案：*KPTI(kernel page-table isolation)*：我们不会让每个 *process* 都能看到 *kernel* 的所有代码和数据结构，我们只会把最基本的放在 *process* 里面，这样做虽然能够解决一些安全问题（不能全部解决），但是会牺牲电脑的性能。因为当我们要进到 *kernel* 的时候，需要 *switch* 到 *kernel page table*，这是花时间的。

2 Concepts Organization

1. Demand paging: Loads pages into memory only when they are needed during execution.
2. Page fault: An event that occurs when a program accesses a page that is not currently in memory.
3. Effective access time: The average time to access memory.
4. Copy-on-write: Parent and child processes initially share the same memory pages, and a copy is made only when a write occurs.
5. Modify/dirty bit: A bit that indicates whether a page has been written
6. Reference string: A sequence of memory page numbers accessed by a process, used to evaluate page replacement algorithms.
7. Belady's anomaly: Increasing the number of page frames results in more page faults.
8. Page replacement algs: OPT, LRU, FIFO.
9. Equal allocation: A strategy that gives each process an equal number of page frames regardless of its size.
10. Proportional allocation: A strategy that allocates page frames to processes in proportion to their memory size or usage.
11. Priority allocation: A strategy that allocates page frames based on process priority levels.
12. Global replacement: A page replacement policy where any page in memory can be replaced, regardless of which process owns it.
13. Local replacement: A page replacement policy where only pages belonging to the faulting process can be replaced.
14. Thrashing: A process is busy swapping pages in and out.
15. Working-set model: A model that defines the set of pages a process needs to keep in memory to avoid page faults.

16. Page-fault frequency: Controlling the allocation of frames to avoid thrashing.
17. Memory-Mapped Files: A technique that allows files to be mapped into virtual memory so they can be accessed like regular memory.
18. Page size issues: Fragmentation, table size, page faults, I/O overhead, resolution, locality, TLB reach, TLB size.

第十章　Mass-Storage Structure

1　In-class Contents

1.1　Disk Structure

1.1.1　Moving-head Disk Mechanism

移动磁头磁盘机制一般由以下几个部分组成：

（1） platter：盘片

（2） spindle：转轴

（3） track(t)：磁道，是数据逻辑组织的最基本单位（也就是在整个组织结构中，第一个有规律划分的结构。具体来说，在进行读写操作的时候，是**磁道-> 柱面-> 扇区**；类比我要找一本书，是**书架-> 书架的一列-> 书架具体的隔间**）

（4） sector(s)：扇区，是读写的最小单元

注意. *sector 和 disk block 有很大关联但是不一样，一个 disk block 由多个 sector 组成。*

（5） cylinder(c)：柱面，物理意义：磁盘同时对一个 cylinder 进行读写操作，所以说，把文件分布在不同 platter 上时，读写是最快的。

（6） read-write head：读写头

（7） arm：磁盘臂，通过前后移动定位到不同的 track。

（8） arm assembly

1.1.2 Disk Interface

磁盘接口包含以下四个：

（1） Disk drives 是按照一个一位数组来分配地址的，数组元素是大小为 512B 的 logical block，**logical block 是最小的 transfer 单位**。所以 8 个 logical blocks 就是 4KB，就能和内存 (disk block) 对齐。

低级格式化（Low-lebel formatting）在物理媒介上创建 logical blocks。

（2） 那么这个一维数组也是顺序地映射到 sector 上，也就是说，1 sector = 1 logical block = 512B。

（3） 多扇区操作是可行的，许多文件系统会一次性读写 4KB 或更多的内容。

（4） 512B 的写操作的原子操作（也就是在单个扇区上操作），所以说这样的写操作要么全部完成，要么一点都没有完成。

1.1.3 Disk I/O

如何去计算一次 I/O 操作的时间和效率呢？

我们认为，一次磁盘的读写操作包括 seek（找到正确的 track）、rotation（等磁盘转到正确的 sector）和 transfer（读写操作）。所以定义 $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$，同时定义 $R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$，常用来比较不同的 drives。

其中 $Size_{Transfer}$ 表示对多少数据进行了读写操作。对于 $T_{I/O}$ 来说是越小越好，同时对于 $R_{I/O}$ 来说是越大越好。

现在呢，我们先来 detail 一下什么是 seek time(Not important)。

seek 一共有四个阶段：

（1） 加速：disk arm 加速动。

（2） 匀速 (coasting)：arm 全速前进。

（3） 减速：disk arm 开始减速。

（4） 停止：最后 read-write head 要到正确的轨道上。（是最重要的一部）

在这个过程中，加速和减速占大部分时间，不过为了简化问题，我们一般认为加速、减速、匀速都能认为是匀速阶段。

这个内容不考。

定义 1.1. *Seek Time*：*将 disk arm 移动到正确轨道的时间。平均的 disk-seek time 大约是 1/3 的 full seek time（遍历所有的 track）。*

好像很奇怪，为什么不是 1/2 是 1/3 呢？

例 42. 假设有 *track 1~track 10*，那么 *full seek time* 为 *9*。如果从 *track 1* 开始遍历，遍历到 *track* R_x，那么确实 *average time* 为 *9/2*。但是实际上，是从 *track* R_y 遍历到 *track* R_x，结果大概就是 *1/3* 啦。

定义 1.2. *Rotational delay*：等待我要的 *sector* 转到 *disk head* 底下，平均的 *rotational delay* 是 *full rotational delay* 的一半。

Rotation 和 seek（不包括 transfer）是 disk operations 里最耗时间的。

例 43. 我们现在可以解决一个之前没有说得很明白的问题，为什么 *I/O* 更喜欢 *large page size* 呢？

我们以 *40KB* 的 *page* 和 *4KB* 的 *page* 举一个例子来对比，这两种的 *rotation* 和 *seek* 的时间是不会变的，但是 *40KB* 的 *page* 的传输量要增加，也就是说，它的 *transfer time* 要增加，听上去不那么好？那么再考虑一下如果这 *40KB* 包含了 *2* 个要读写的 *4KB*，那么访问时间会是什么样的呢？

对于 *40KB* 的那个，只需要 *1 seek + 1 rotation + transfer(40KB)*。

对于两个 *4KB*，则需要 *2 seek + 2 rotation + transfer(8KB)*。

但是我们之前又说，*transfer* 其实不怎么占时间，所以综合来看 *40KB* 的 *I/O* 所花的时间是要少很多的。

1.1.4 Workload Assumptions

我们现在有两种 workload，一种是 Random workload, 就是每次随机访问 disk 上的 4KB（应用场景：类似数据库的频繁随机访问）；另一种是 Sequential workload，会连续读取一系列的 sectors（应用场景：读取类似视频的大量扇区数据）。

在这之后，我们对两种 disks 进行比较，一种是 Barracuda，其特点是 Average Seek 较长，但是 Cache 大（常作个人电脑、家庭存储，更适用 Sequential workload）。另一种是 IronWolf，它的特点和 Barracuda 相反，Average Seek 更快，但是 Cache 较小（常作为服务器，更适用 Random workload）。

我们再讨论一下在 Sequential workload 的情况下，$T_{I/O} = T_{transfer}$，因为这个过程当中是省略掉 seek 和 rotation 的，head 不需要重新定位，所以 sequential workload 的表现会比 random workload 的表现好得多。

定义 1.3. *Track skew*（磁道倾斜）：具体见 *PPT15* 页，核心思想是偏移几个 *block*，让 *head* 从外圈切进内圈的时间和 *rotate* 的时间可以相同，让顺序读取更加流畅。

1.1.5 Some Other Details

定义 1.4. *Multi-zoned disk drives*：在磁盘旋转时，外圈的磁道线速度要比内圈磁道线速度快（在 *const angular velocity,CAV* 的情况下），所以外圈的磁道可以容纳更多的 *sector*，内圈的相对少一点。但是呢，每个 *cylinder* 的区域的 *sector* 数量还是要统一的。

当然，现在也有了 *constant angular velocity(CAV)* 和 *constant linear velocity(CLV)* 两种形式。

定义 1.5. *Cache(track buffer)*：内存有一块小的地方，通常大概是 *64MB* 或者 *256MB*，这些地方可以被 *drive* 用来暂时存储被读和被写的 *data*。

例 44. 一般来说，*Random workload* 要的 *Cache* 少，*Sequential workload* 要的 *Cache* 大一些。

1.2 Disk Scheduling

Operating system 负责更有效地使用硬件，那么对于 disk drives 来说，这意味着能有更快的 access time 和 disk bandwidth（硬盘带宽）。

那么我们怎么做到最快完成这个任务呢？也就是 minimize $T_{I/O}$？其实如果我们给定了一个 disk，那么 I/O 操作的 rotation delay 和 transfer time 都是固定的，只有 seek time 是会变化的，所以我们的目标就变成了 minimize seek time。

我们再做一个假设，我们不考虑 seek 过程的加速减速变化，认为他们都是匀速的，所以 seek time 可以用 seek distance 来代替表示。

定义 1.6. *Disk bandwidth*= 传输的总字节数/完成 *I/O* 操作的总时间

Disk 的 I/O request 可能来自以下几个方面：

（1） OS

（2） System processes

（3） Users processes

这里想要表达的意思是，磁盘访问请求不只是来自用户，系统本身也会频繁进行 I/O 操作。

I/O 请求包括输入或输出模式、磁盘地址、内存地址和需要 transfer 的 sector 数量。OS 做的事情是为每个 disk 或者 device 管理请求队列。

注意. 空闲的 *disk* 可以直接完成 *I/O* 请求，只有忙的 *disk* 才会有等待队列。也就是说，优化算法只有太多请求排队的时候才有意义。

1.2.1 First Come First Serve(FCFS)

核心思想：很简单，根据 request queue，那个先来先 seek 哪个就行。

1.2.2 Shortest Seek Time First

核心思想：还是很简单，对于当前节点来说，哪个近就访问哪个。

But! 有一个问题，可能会出现 starvation，你一直在 0~50 访问，那么 100 多，200 多的就饿死了。

1.2.3 SCAN Algorithm(the elevator algorithm)

核心思想：我们先定一个方向，一条路走到黑（也就是走到边界），然后返回回来把路径上的 request 都完成。

注意. 别名电梯算法要知道，然后需要知道一开始的 *seek* 方向。

1.2.4 C-SCAN Algorithm(Circular-SCAN)

核心思想：在 SCAN 的基础上，先一条路走到黑，再走到另一个头（中间就算有 request 也不管），然后再回头 seek 所有的 request。

注意. 记得跨越两个边界的那条路要算上。

这么做有什么物理意义吗？有的兄弟，有的。

（1） 跨越两个边界的那条路实际上是加速再减速的过程，所以说不会太耗时间。

（2） C-SCAN 相比 SCAN 的响应更加平滑，新来的 request 也能更快被响应。(provide a more uniform wait time than SCAN)

1.2.5 LOOK & C-LOOK

核心思想：两个边界设置为 queue 里面最大的和最小的 request 就行了，不用一条路走到黑。

1.2.6 Other Issues

注意. 有个问题，两个 *SCAN* 算法和 *LOOK* 算法并不是最好的算法，因为他们只优化了 *seek time*，完全忽略了 *rotation delay*。

定义 1.7. *Short Positioning Time First(SPTF)*：*positioning time* 包括了 *seek* + *rotation*，这才是真正最优的（当然和 *disk drive* 本身的实现有关，因为涉及 *rotation*）。

考点 16. 操作系统把 *requests* 打包给 *disk drive*，让 *disk drive* 完成调度，所以对于 *OS* 来说，它是没什么工作的。

所以说，SSTF(or SPTF) 确实是最常见的算法，也很有吸引力，但问题是会有饥饿的问题。那么对于 SCAN 和 C-SCAN 来说，他们更加擅长解决有 heavy load 的情况，因为他们不太会有饥饿的问题（虽然一般不会有很重的 I/O）。

还有一个小问题，谁来完成这个调度呢？

在老的系统当中，OS 完成全部的调度。但是在现代系统当中，OS 只会做 I/O 的 merging，也就是把 requests 合并起来，而由 disk drive 来做例如 SPTF 这样的调度。

1.3 RAID Structure

RAID 在生活当中是很有用的。**（但是考试只考基本的概念）**

定义 1.8. *RAID*：一种技术去将几个磁盘搞在一起来构成一个更快、更大、更可靠的磁盘系统。

从外部来看，RAID 就像是“1”个 disk，里面是一系列可供读写的 block，这种透明性增加了 RAID 的可部署性。从内部看过去，RAID 由许多 disks，内存（包括易失性的，也有非易失性的）以及一个或更多的处理器来管理这个系统。

那么为什么我们用 RAID 而不是用单独的一个 Disk 呢？

（1） Perfoemance：几个 disk 并行读写，可以大幅加速 I/O 操作。

（2） Capacity：打个比方 5000RMB 可能可以买到 5 张 4GB 的 disk，但是买不到 1 张 20GB 的 disk。

（3） Reliability：在 disk system 中添加一些冗余，这样 RAIDs 可以接收磁盘内容的丢失并继续运行。

1.3.1 RAID Level 0

特点：条带化，完全没有 reduntant（所以本质并不是一个 RAID，更多作为一个 baseline）

RAID-0 是 performance 和 capacity 的上限，但是如果一个磁盘坏了，那么整个 RADI-0 的数据就木的了。

我们来对 RAID-0 做一个分析：

（1） Perfoemance：Perfect，并行读写利用到了所有的 disks。

（2） Capacity：Perfect，存储效率为 100%，假如有 N 个 disk，每个 disk 能存储 B 个数据块，那么 capacity 为 N×B。

（3） Reliability：没有冗余，可靠性较差，反而比单个 disk 更加糟糕。

现在我们来分析一下 RAID-0 的 performance：

例 45. *Single-request latency*（单次请求延时）：如果说我每次只访问一个 *block*，那对不起，这种情况下的访问速度和 *single disk* 没啥区别，*performance* 没有明显提升。

例 46. *Steady-state throughput*（持续吞吐量）：*But*！如果说我们有 *N* 个 *disk*，同时是 *sequential workload*，传输速率为 *S MB/s*，那么 *RAID* 的总传输速率就是 *N*×*S MB/S*。如果我们用的是 *random workload*，传输速率为 *R MB/S*（*R* 一般远小于 *S*），这时候 *RAID* 的总传输速率就是 *N*×*R MB/S*。

还有一个变体，按照默认的说法，stripe 是一行一行这么来的，这里可以改变 striping 的宽度，如果宽度为 2，就是每次分配两个数据块给每个 disk。

定义 1.9. *Chunk size*：表示 *striping* 的宽度。

chunk size 小的话，可以增加一个文件读写的并发程度，但是代价就是在每个 disk 上都会需要做 positioning，那么这对于整个读写系统来说可能更花时间。

1.3.2 RAID Level 1

这是最简单的 RAID，核心思想是**"Mirroring"**，也就是说我们会在一个磁盘系统中把每个 block 都拷贝一下，同时每份拷贝都会被安排在不一样的 disk，来应对 disk failure 的问题（增加 reliable）。

一种常见的分配方式是 RAID-10(RAID 1+0)，它做的事情就是先镜像，再让他们按照 stripes 去排列，也就是先做 RAID-1，再做 RAID-0。当然对应的也有 RAID-01(RAID 0+1), 其物理盘上的数据和 RAID-10 是一样的，但是从表现来说，RAID 1+0 会更好一些，具体看 PPT 的解析。

同样的，我们对 RAID-1 进行一个分析：

（1） Perfoemance：见后文，不考。

（2） Capacity：比 RAID-0 贵很多，因为我们相当于只用了一半的 capacity。

（3） Reliability：还是不错的，可以最多接受 N/2 个 disk failure（在 RAID 1+0 的情况下，要是一个 disk failure 了，可以去镜像盘里找）。

以下内容不考！不考！不考！

例 47. *Single-request latency*：*Read* 和在一个 *disk* 上是一样的，但是 *Write* 的时候要注意虽然是并行地写，但是写的时间取决于两个里面时间更长的那个。

例 48. *Steady-state throughput*：如果是 *Sequential*，读和写都是 *N×S/2 MB/s*，如果安排好的话理论上是能达到 *N×S* 的。*Random* 情况下读是 *N×R MB/s*，写是 *N×R/2 MB/s*。

1.3.3 RAID Level 4

我们发现 RAID-1 好像是可以解决 reliable 的问题，但是他需要的空闲位太多了，所以 RAID-4 提出的方法就是在 RAID-0 的基础上，加一个校验位 (parity block)，然后校验位的计算由其余非校验位通过异或运算得到。

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

同样的，我们对 RAID-4 做一个分析：

（1） Perfoemance：见后文。

（2） Capacity：N-1，因为有一个作为校验位了。

（3） Reliability：最多可以承受一个 disk failure。

现在再来讨论一下 performance：

例 49. *Single-request latency*：*Read* 和 *single disk* 一样；*write* 是 *single disk* 的两倍（因为每次写不仅要改变当前位置的数据，还要改变校验位）。

例 50. *Steady-state throughput*：*Sequential workload* 的读和写都是 *(N-1)×S MB/s*；*Random workload* 的读是 *(N-1)×R MB/s*，写是 *R/2 MB/s*，因为每次写只有两个 *disk* 可以并行操作，同时要两次读两次写，所以是 *R/2*。

对于 random writes，其实有两种方法对校验位进行计算，一种是 additive parity，也就是不看原来的 parity，更新了数据之后重新计算 parity，那这个显然是没那么高效的。另一种方法是 Subtractive parity，具体来说是只需要用被修改的 block 和 parity 就可以进行操作了，一共需要两次读和两次写。

RAID-4 有一个问题，就是每次并行操作的只能有两个 disk，假设 disk 4 里面都是 parity，那么 disk 4 会忙死，这样的效率是比较低的。

1.3.4 RAID Level 5

这个时候，我们就想到了 RAID-5，他和 RAID-4 也很像，做的事情就是让每个 disk 都有一个 block 作为 parity block。

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

同样的，对 RAID-5 进行分析（**啊，要疯掉了，这么多**），Capacity 和 Reliability 和 RAID-4 是一模一样的，现在来看 performance：

例 51. *Single-request latency*：还是和 *RAID-4* 一模一样。

例 52. *Steady-state throughput*：*Sequential workload*，和 *RAID-4* 一样，读和写都为 *(N-1)×S MB/s*。*Random* 情况就不一样了：

Read：*N×R MB/s*，因为 *pairty* 是均匀分布的，所以有的时候一个 *disk* 在工作，也有的时候这个 *disk* 充当 *parity*，所以是 *N* 不是 *N-1*。

Write：*(N/4)×R MB/s*，除以 *4* 是因为每个写操作的背后都是 *4* 个读写，同时这四个读写是可以一起并行的。

1.3.5 Conclusion

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \times B$	$N \times B/2$	$(N-1) \times B$	$(N-1) \times B$
Reliability	0	1 (for sure) $N/2$ (if lucky)	1	1
Throughput				
Sequential Read	$N \times S$	$(N/2) \times S$	$(N-1) \times S$	$(N-1) \times S$
Sequential Write	$N \times S$	$(N/2) \times S$	$(N-1) \times S$	$(N-1) \times S$
Random Read	$N \times R$	$N \times R$	$(N-1) \times R$	$N \times R$
Random Write	$N \times R$	$(N/2) \times R$	$(1/2) \times R$	$(N/4) \times R$
Latency				
Read	T	T	T	T
Write	T	T	2T	2T

上面内容都不考喔！

第十一章　File-System Interface

1　In-class Contents

1.1　Files and Directories

我们到现在学过了很多 abstractions，包括 process、thread、page 等，这里再额外介绍两个文件系统用到的：

定义 1.1. *File*：是一个线性字节数组，可供读写，每个文件都有一个 *low-level name*，被称为 *inode number*（索引节点编号，*int* 类型数据，不是 *file id*）（类似 *process* 有 *pid*，*thread* 有 *tid*），这样 *OS* 就能知道你是哪根葱了。

定义 1.2. *Directory*（目录）: 包含一个列表的 *entries*，每个 *entry* 是一个 *pair*，*pair* 的内容是用户可读的名称和一个 *low-level name*（类似 *page table* 做的 *translation:logical->physical*），可以指向一个文件，也可以指向其他目录（子目录），所以说每个目录也是会有 *inode number* 的。

定义 1.3. *Directory tree(hierarchy)*：在目录层级结构中，从根目录开始，用分隔符（一般为"/"）将父目录和子目录分开，直到找到我要的文件。

例 53. *home/user/hjm/pressure.doc*

这样的话一个文件就可以通过它的绝对路径被访问到。所以说 File system 提供了一个非常好的东西就是他的命名很方便，用户容易理解。

那么对于一个文件来说，通常是 x.y 的格式，前面是文件的名字，后面**通常**是它的扩展名，比如：rgl.ppt。

注意. 当然这只是一个约定俗成，应由用户决定怎么打开这些文件，*main.c* 文件也不一定真的是 *C source code*，不过修改扩展名一般会出现“改变扩展名可能导致文件无法正常打开”的提示。

1.2　File Interface

本质就是一些 System calls，那么我们一点一点来分析和认识。

1.2.1 Creating Files

首先在创建文件的时候，会用到 open() 这个 system call。

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

这里做的事情其实很简单，就是在当前工作目录中创建一个名为“foo”的文件，O_CREAT 指的是我要创建一个文件，O_WRONLY 指的是这是个只能写的文件，当然也可以设置为 read only，O_TRUNC 指的是如果这个文件已经存在了，那就会给它截断 (Truncate)，把里面的东西全部清空，当然也可以设置为在现有文件的基础上进行编辑。最后，open() 会返回一个 int 类型的数据 fd，也就是 file descriptor，是一个 low-level name，一般会被 read() 和 write() 的 syscall 调用到。

注意. *所有文件在进程里都表现为 file descriptor。*

1.2.2 Reading Files

这里我们举一个例子，来读一个文件中的内容：

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

这里做的事情就是将 hello 写在一个名为 foo 的文件当中，然后读取 foo 的内容并且打印出来。

现在我们来仔细分析一下 cat：

```
prompt> strace cat foo
...
# 打开foo这个文件，这里foo是一个只读文件，同时支持大文件的读。
open ("foo", O_RDONLY | O_LARGEFILE)      = 3
# 那么为什么返回的fd是3呢？因为0代表stdin，1代表stdout，2代表stderr。
# 现在开始读文件
read(3, "hello\n", 4096)                  = 6
# read里面有三个参数，第一个参数指的是读取的文件的fd；第二个参数是一个buffer，里面存放的是读取的内容，这里是hello和一个换行符；第三个参数是buffer的大小，设置的是4096B，也就是4KB，那就是page size，为什么这么设置？因为方便swap in和out
```

```
# read的返回值是读取的数据的大小，这里hello加换行符一共是6个Byte。
# 现在我们把读取的内容写到控制台当中
write(1, "hello\n", 6)                    = 6
# 第一个参数1表示stdout，这样我们才能把写的东西放到控制台中，返回的6同样表示内容的大小。
# 控制台输出hello
hello
# 因为我们一次性只读了4KB，还要进行下一次读写，发现buffer里面没有东西了，那说明我们已经读完了，就可以终止操作了。
read(3, "", 4096)                        = 0
close(3)                                 = 0
...
prompt>
```

其实，说白了 cat 用到的 syscall 也就 read、write、open、close。

1.2.3 Write Files

写文件和读文件差不多，唯一不同的就是和 stdout 没啥关系了。

1.2.4 Reading/Writing NOT Sequentially

现在我想实现一个功能，我的读写不再从头开始，而是说能够访问文件中的特定位置并进行读写。

这时候就要请出我们的 lseek() 这个 Syscall 了：

```
off_t lseek(int fildes, off_t offset, int whence);
```

第一个参数指的是 file descriptor，也就是我们要操作的文件；第二个参数是偏移量 offset，可以是正的可以是负的；第三个参数表示 lseek 怎么去 perform，具体分为以下三种：

（1） SEEK_SET：从头开始，加上一个偏移量

（2） SEEK_CUR：从当前位置开始，加上一个偏移量

（3） SEEK_END：从文件末尾开始，加上一个偏移量，那这里的偏移量大概率就是负的了

我们注意到 SEEK_CUR 是从当前位置开始，那么这个当前位置的更新又有两种方式：

（1） 隐式更新：前面提到的 read() 和 write()Syscall，假设读/写了 N 个 bytes，那当前的位置就是上一个位置＋ N（那么一般性是 4KB 的读和写）。

（2） 显示更新：就像 lseek 说的那样。

1.2.5 Writing Immediately

大部分情况下呢，我们要把数据写回 disk 并不是立刻执行的，write 请求什么时候响应取决于 disk，对于 file system 来说，他做的事情就是先把要写入的数据缓存在内存当中，把几个 write() 操作合并起来，等时机合适了再把它们全部写回 disk。

那么，万一我想立刻写回去，我不想等怎么办？

别急，我们有一个 Syscall 叫做 fsync()(file synchronize)。它可以强制把当前文件指向的文件中的所有数据立刻写回去，就相当于写 word 文档的时候 ctrl+S 保存一下。

注意. *在创建一个新文件的时候，注意要确保这个新文件是在目录里面的，这时候也要调用 fsync()。*

那么问题来了，万一我有些数据没有写入磁盘，电脑寄了，怎么办？

1.2.6 Renaming Files

我知道你很急，但是你先别急，我们先来讲讲 rename 这个 syscall，它是一个原子操作，可以改一个文件的名字，原子操作有什么好处呢？要么就改名成功，要么就失败，不会出现我断电了，都变成乱码了。

所以我们依据这个特性，可以用这串代码来解决断电数据丢失的问题：

```
1 // 创建一个叫foo.txt.tmp的临时文件，当然本来也有一个foo.
    txt文件的
2 int fd = open("foo.txt.tmp", O_WRONLY | O_CREAT | O_TRUNC)
    ;
3 write(fd, buffer, size);
4 fsync(fd);
5 close(fd);
6 // 运行到这里崩溃是没事的，东西至少还在foo.txt.tmp里面，但
    是之后记得要去恢复或者改名。
7 rename("foo.txt.tmp", "foo.txt");
```

```
8 // 完成保存操作
```

例 54. 我在写 *word* 文档，可以发现如果没有保存的话，隐藏的项目里面会有个副本，一旦我点了保存，那个副本就不见了，那个副本就是临时文件。

1.2.7 Geting Information about Files

File system 除了保存着文件的内容，还会保存关于文件本身的一些信息，这些信息被称为元数据 (metadata)，而这些信息会被存在 inode 里面（和 pcb 很像，被称为 file control block）。

如果说我们要看这些 metadata，可以用 stat() 或者 fstat()Syscall。

1.2.8 Permission Bits

我们用 9 个 bits 来表示一个文件能够被谁访问以及能够被如何访问，看下面的例子：

```
1 prompt> echo hello > file
2 prompt> ls -l foo.txt
3 -rw-r--r-- 1 kai kaigroup 0 Jul 13 16:29 foo.txt
4 # kai为文档的作者，rw-表示kai有读写权限；r--r--表示
    kaigroup和其他用户只有读的权限；1表示硬链接数；0表示文
    件大小，Jul 13 16：29表示最后修改的时间；最后一个是文档
    的名字
```

那么如何修改权限呢？

```
1 prompt> chmod 777 test.sh
```

首先我们看一个 7，7=4（读）+2（写）+1（执行）；那么三个 777，就对应的是 kai 的权限、kaigroup 的权限和 others 的权限。所以这个代码的作用就是让所有人有所有的权限。

1.2.9 Removing Files

当我们删除一个文件的时候，我们调用 rm() 这个 UNIX 中的指令，**不是一个 Syscall**！它的作用是调用一个 unlink() 的 Syscall，把我要删除的文件的名字给扔掉。那么为什么是这么做呢？我们这里留一个伏笔。

1.3 Directory Interface

1.3.1 Making Directories

在创建目录的时候，我们调用 mkdir()Syscall：

```
prompt> strace mkdir foo
...
# 这里做的事情是创建foo目录，并且设置权限为0777，也就是所有人都能rwe。
mkdir("foo", 0777)                          = 0
...
prompt>
```

在目录创建好之后，并不完全是空的，会默认有一个"."（表示当前目录的引用）和一个".."（表示父目录，也就是上一级目录的引用）

1.3.2 Reading Directories

简单来说，就是用 ls()Syscall 展示一堆东西。

1.3.3 Deleting Directories

调用 rmdir()Syscall，把目录下的所有文件都删除之后，再把这个目录删除（所有 OS 都是这么干的）。

1.4 Links

1.4.1 Hard Links

link()Syscall 需要两个参数，一个是旧的路径名字，一个是新的，即 link(oldpath, newpath)，这样可以多创建一个 entry，指向同一个 inode，也就是说，文件并没有被复制，只是增加了一个新名字来引用相同的内容（很像浅拷贝）。

同样的，如果删了其中一个名字，只要另一个名字还在，内容就不会被删掉；如果改其中一个文件的内容，另一个也会改变，因为它们指向的是一样的文件内容。

现在我们来看一个例子：

```
prompt> echo hello > file
prompt> cat file
hello
```

```
prompt> ln file file2
prompt> cat file2
hello
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

下面我们来解释为什么我们在删除的时候用的是 unlink() 而不是 remove() 或者 delete()。

在创建一个文档的时候，我们首先会创建一个 inode，里面有这个文件的内容，包括大小、blocks、**链接计数 (reference count)**（来跟踪有多少个文件名指向这个 inode）；其次，我们会链接一个面向用户的 human-readable 名字在目录下。

那么当我们 unlink 一个文档的时候，file system 会首先检查 reference count 是不是 0，如果不是 0，那没事，如果是 0，那很抱歉，兄弟你被“销户”了，文件被彻底删除掉。

所以这也就解释了为什么所谓的“文档删除”是 unlink()，而不是 remove()，也不是 delete()。

1.4.2 Symbolic Links

符号链接，也就是 Soft Links。

Hard links 是有一些限制的，因为我们没办法为一个目录创建 hard link，比方说，我们首先创建了一个 dir，又在 dir 这个目录下创建了一个 link，硬链接到 dir，这样就循环了，就坏了。

不仅如此，由于每个文件系统的 inode 都是独立的，所以说，我们没办法将链接链接到别的 disk 部分-> 找一个文件首先要找它所属的文件系统。

这时候，我们就引入了 Symbolic Link，见以下代码：

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt>
```

这个时候呢，file2 就指向了 file，而不是像硬链接那样指向 file 对应的 inode。

Symbolic link 的本质是一个文件，它有自己的 inode，是独立于原文件的，就比如：**快捷方式**。

当我们运行 ls 的时候：

```
prompt> ls -al
drwxr-x--- 2 kai kai 6 Jul 13 19:10 ./
drwxr-x--- 26 kai kai 4096 Jul 13 16:17 ../
-rw-r----- 1 kai kai 6 Jul 1319:10 file
lrwxrwxrwx 1 kai kai 4 Jul 1319:10 file2 -> file
# 我们来看这四行的最前面一个字母，"-"表示regular files，"d"表示directories，"l"表示soft links。在这个例子当中，symbolic link的大小是4B。
```

定义 1.4. *Dangling Reference*（悬空引用）：如果我们把 *file2* 指向的 *file* 给删了，就像我有一个游戏本体和快捷方式，我们把本体删了，快捷方式虽然还在，但是它指向不了任何东西。不过，如果我们删快捷方式，对游戏本体是没有影响的。

1.5 File System Interface

最后我们来提一下怎么创建 file system，在这里我们用的是 mkfs() 指令：

```
prompt> mkfs -t ext3 /dev/sda1
# 创建一个ext3类型的文件系统，并将其写入/dev/sda1设备（一个磁盘分区）
```

定义 1.5. *Mount*（挂载）：一种命令，将现有目录作为目标装载点，并在该点将新的文件系统粘贴到目录树上。

```
prompt> mount -t ext3 /dev/sda1 /home/users
# 任何访问/home/users的操作都会访问/dev/sda1上的内容，不再是原本的/home/users的内容，直到被卸载。
```

第十二章　File-System Implementation

1　In-class Contents

1.1　Warm-up

那当我们知道了有哪些 interface 之后，我们要开始 implement。那么既然要 implement，那肯定还要考虑 performance。

我们首先了解一下 File system 的特征，可以总结为：小文件多，但平均文件大小一直在增加；一小部分的大文件占据了绝大部分的空间；系统里会有成千上万个文件；即使你扩容了磁盘容量，也会有大约 50% 的使用率；目录不会有太多。

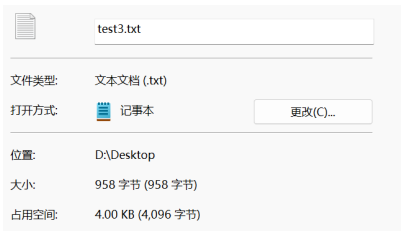
这时候我们来看一个例子：

Filename	Content	Description	**Size**	**Space**
test1	-	-	0	0
test2	“This is a test file.\r\n”	$\times 1$ (line)	22 bytes	0
test3	“This is a test file.\r\n”	$\times 40$ (lines)	878 bytes	4 KB
test4	“This is a test file.\r\n”	$\times(40-39)$ (lines)	22 bytes	4 KB

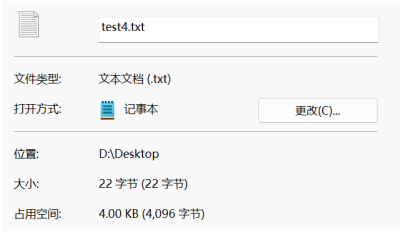
我们来验证一下：

(a) test1

(b) test2



(c) test3



(d) test4

诶？好像还真是这样，那为什么 test2 的空间是 0，但是 test4 的空间是 4KB 呢？

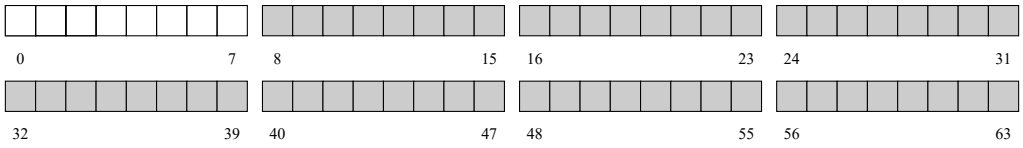
在这个章节中，我们会描述实现本地文件系统和目录结构的细节；描述远程文件系统的实施；讨论块分配和自由块算法以及权衡；最后讲一些固态盘上的文件系统。

1.2 Typical File System

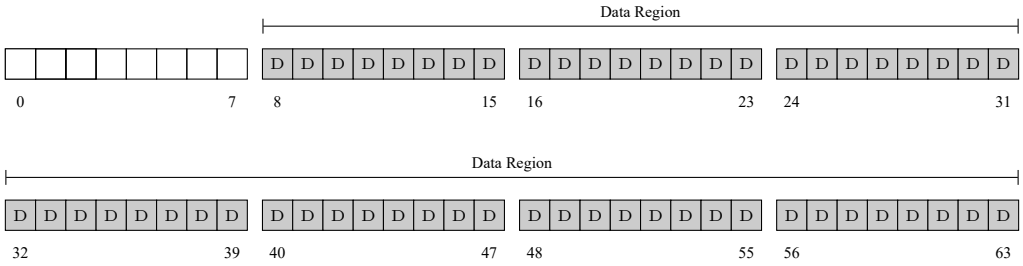
1.2.1 Very Simple File System

我们先讲一个 VSFS，是 UNIX file system 的简单版本，我们需要理解的有：Data structures（用了什么数据结构来管理数据和元数据）和 Access methods（进程发出的 syscall 如何映射到底层的数据结构）。

我们来举一个例子，我们找了一个 disk，有 64 个块，每个块的大小都是 4KB：

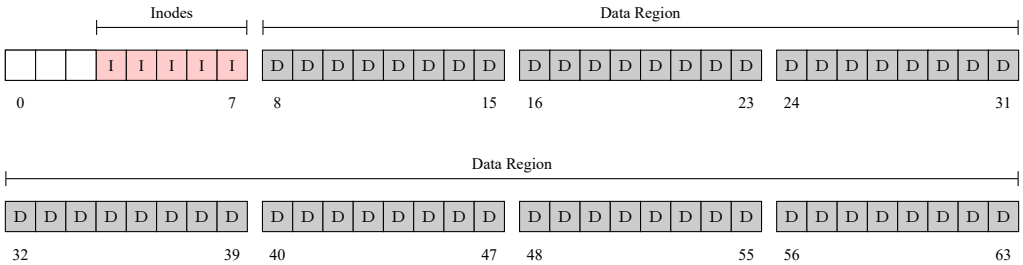


我们把最后 56 个 blocks 用来放 data，标记为 D，而前面的 8 个 blocks 用来放各种 data structures，这也就说明了为什么我买了一个 T 的盘，但是只有 900G 可以用：



现在我们聚焦前面 8 个 blocks，回顾一下第十一章，想想我们要放些什么数据结构。

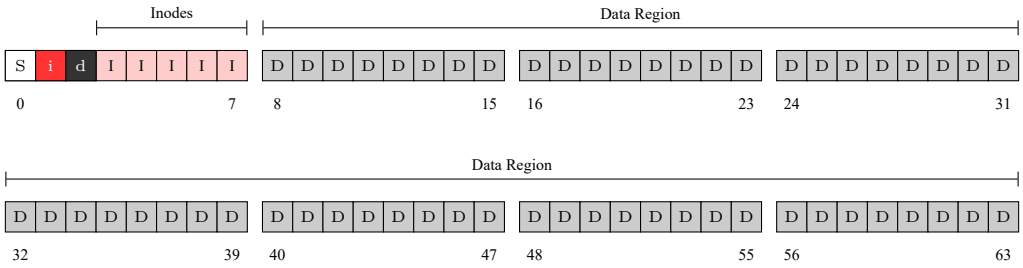
首先要放的是 inodes，inodes 组成了 inode table，存在了前面 8 个 blocks 的后面五个：



现在我们再假设一个 inode 要 256B，那么 5 个 4KB 的 blocks 一共能够存 5×4×1024/256=80 个 inodes，这也就代表了这个文件系统最多可以有 80 个文件（或者说是有 80 个有 inode 的东西）。

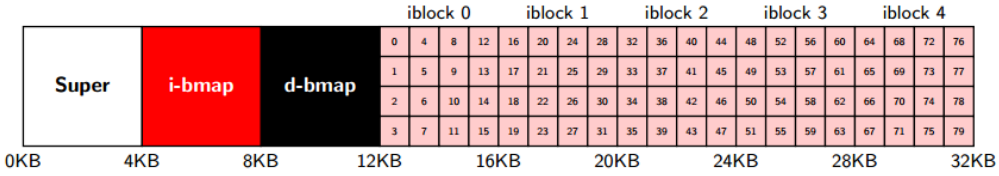
现在我们又有一个问题了，我们怎么知道哪个 inode 是空的或者哪个 data block 是空的呢？

我们引入了 inode bitmap 和 data bitmap，放在整个系统的 1 号 block 和 2 号 block，它们可以知道一个 inode 有没有被分配或者一个 data block 有没有被分配，这样的数据结构可以是一个 free list。



这时候我们发现 0 号 block 是空的，它被称为 Superblock。什么意思呢？它有整个文件系统的整体信息，包括文件系统有多少 inodes、data blocks 以及 inode table 的地址在哪里。所以当挂载这个文件系统的时候，OS 会首先去读超级块。

现在我们来聚焦一下前八个 blocks，看看里面都有什么：



我们之前也说了 inode 在文件系统中被称为 File Control Block(FCB)，是 index node 的简称。每个 inode 有 256B 的内容，那么为了在逻辑上能够更好表示 inode，我们给每个 inode 一个编号，被称为 inumber(0~79)。从这个图中我们也可以知道，i-bmap 会有 80 个 bits，来表示这 0~79 的 inodes 有没有被使用；而 d-bmap 则有 56 个 bits，因为一共有 56 个 data blocks，来表示对应的 data block 有没有被使用。

现在还有一个问题，给定一个 inumber，怎么去读这个 inumber 对应的 inode 里面的 metadata 呢？

很简单，我们首先找到 inode table 的首地址，那么在这里就是 12KB，假设我们要读 inumber=39 位置的数据，那么它在磁盘当中的地址就是 12KB+39×256B。

那 inode 里面有什么呢？以 Ext2 Inode 为例，有一坨东西，最重要的是一共有 15 个 pointers，一个 pointer4B，所以总共占据 60B 的空间，指向 disk 上的不同位置，也就是指向 disk block。

那么，我们来深入分析一下，一个 inode 如何去找到对应的 data blocks 呢？有一个很简单的方法，就是用 pointers **直接**去指向 data blocks，但这是有问题的，因为 15 个 data blocks 也就只有 60KB 的大小，这对于大型文件肯定是不能接受的。

所以我们的一个 solution 就是去使用**间接**指针，什么意思呢？就是用 pointer 指向一个存满了 pointers 的 data block。打个比方，一个 data block 4KB，那么就可以存 1024 个 pointers，再用这些 pointers 每个指向一个 data block，所以说最后可以用一个 pointer 指向 1024 个 data blocks，1024×4KB=4MB，15 个指针就可以指向 60MB 的空间，这是可以接受的。

那有了这样的一级的非直接指针，就会有二级间接指针 (4GB)、三级间接指针 (4TB)。或者说，我们用 extents(ext4) 来代替 pointers。

定义 1.1. *Extent*：本质还是一个指针，只是比方说指针指向的是第 *16* 个 *data block*，它要额外设置一个长度，假设为 *6*，那么这个 *pointer* 或者说这个 *extent* 最终指向的就是 *16*，*17*，*18*，*19*，*20*，*21* 这些 *blocks*。

这样子看上去还不错，节省一些空间，但是问题在于他不灵活，因为每个 extent 的 length 可能是不一样的，而且你只能连续访问 blocks，中间可能出现没有内容的情况。

那么在讲完了 VSFS 之后呢，我们来关注一下其他 File System。

1.2.2 FAT

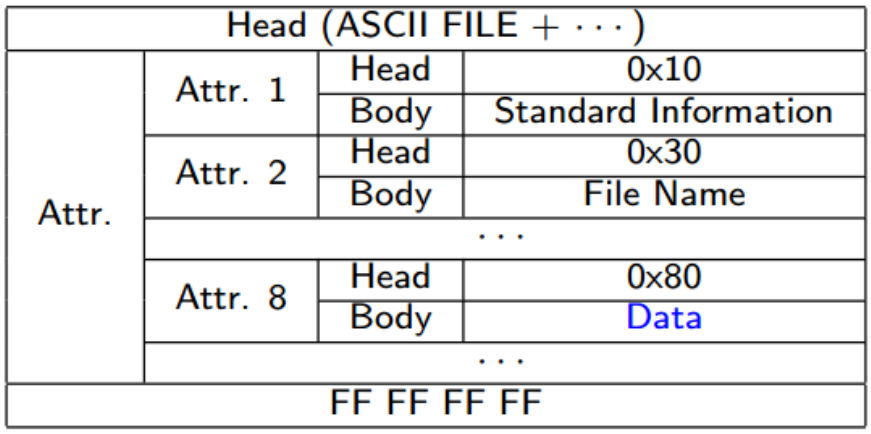
这个文件系统将 File-Allocation Table 作为 File Control Block，不支持大文件，但实现起来较为简单。

核心思想：类似链表，通过 start block 指向下一个 block 中的内容；在下一个 block 的最后又会有一个指针去指向再下一个 block 的内容。这样的话有一个问题，就是如果文件太大，要经过的 blocks 很多，效率是很低的。又况且访问的是 disk，效率更低。

1.2.3 NTFS

NTFS 被经常用在 Windows 操作系统下。

核心思想：用 MFT(master file table) 作为 File Control Block，和 inode 差不多，里面有文件的各种属性，例如文件名、文件的数据等。



我们来看一下这个“表格”，它本质是一个 entry，也就是说应该存在一行里面，这里为了方便展示，把它画成了一个表格。

注意. *Data 里面既可以存放小的数据，也可以存放 disk pointers*。

这时候我们再回顾一下 warm-up 的例子，做一个分析：

Filename	Content	Description	**Size**	**Space**
test1	-	-	0	0
test2	“This is a test file.\r\n”	$\times 1$ (line)	22 bytes	0
test3	“This is a test file.\r\n”	$\times 40$ (lines)	878 bytes	4 KB
test4	“This is a test file.\r\n”	$\times(40-39)$ (lines)	22 bytes	4 KB

我们先明确一下，Space 在这里指的是占用的 data block 的大小，和文件大小无关。

首先我们来解决第一个问题，test1 虽然没有内容，但是它有文件名，为什么 Size 和 Space 都为 0 呢？因为文件名是存放在 MFT 里面的 Attr.2 中间的，就不用占文件的存储。

再来解决一个问题，为什么 test2 的 Space 是 0 呢？因为 test2 的内容只有 22B，远小于 1KB，那就可以直接存在 MFT 的 Attr.8 里面。

那么为什么到 test3 的时候 Space 就是 4KB 了呢？因为 878B 已经很接近 1KB 了，MFT 不太够放了，只能给文件分配一个大小为 4KB 的 data block。

最后一个问题，test4 的 Space 和 test2 的 Space 为什么不一样呢？是因为 data block 已经分配出去了，只有文件全部删除完，才能够把那 4KB 收回来（但是噢，如果我分配的是 12KB，我的 Size 只有 3KB 了，那多分配的 8KB 是可以收回来的）。

考点 17. *非常中要的三个 Allocation Method*：

（1） *Contiguous allocation*：*把文件分配在 disk 的连续 blocks 上。（现在已经没有纯的 contiguous allocation 了，因为这样做的问题太大了，没有那么多连续空间给你）*

举例：*ext4(Linux)*、*ntfs(Windows)*，注意啊，它们两个本质是 *indexed allocation* 和 *contiguous allocation* 的结合，做的事情是尽可能连续，但也需要 *disk pointers* 去指向 *data block*。

（*2*） *Linked allocation*：就是链表找文件存放的内容，效果比较糟糕。

举例：*fat*

（*3*） *Indexed allocation*：由 *disk pointer* 指向 *data block*。

举例：*ext2,ext3(Very Simple File System)*

1.2.4 Directory Organization

Directory 的本质就是一个 translation table，里面有很多 entry name 和 inode number 的配对，现在我们来举一个例子：

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

我们先看一下这张表，strlen 表示的是文件名的长度，reclen(record length) 一般是 2 的多少次幂，表示 **entry 在目录文件中占的字节数量**。

那么如果我们要删掉一个文件 (unlink())，也就是把某个 entry 删了之后，它占用的空间还是存在的，因此我们需要做一些标记告诉 directory 这个 entry 没了，用的方法是把 inode number 置为 0。同时这也是为什么要用 reclen 的原因，reclen 指明了 entry 的长度，这样在扫描目录的时候可以跳过空的 entry，且要是插入一个新的 entry 时，可以复用空闲的空间。

最后，目录也有一个 inode，被存放在 inode table 里面，但是会对这个 inode 进行区分，会标记”directory” 和”regular file”。

1.2.5 Access Paths: Reading

现在呢，我想要打开一个文件，读一下，然后关闭它：

$$open(''/foo/bar'', O\_RDONLY)$$

对于 file system 来说，它必须遍历整个路径来定位到我要的 inode，所以读的过程大致为：

（1） 读根节点的 inode

（2） 在 inode 中找到指向 data blocks 的 pointers，里面包含着根目录的内容

（3） 找到 foo 的入口（也就是对应的 inum）

（4） 一个个文件夹往下找，直到找到目标文件的 inode，读取里面的 block 的内容

（5） 上面做的工作本质是在打开文件，那么读取文件还需要记住一个最近的访问时间变了，是需要在 inode 里面进行修改的

现在我们用一个流程图来理解一下（希望真的能理解吧）：

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
$open(bar)$			read			read				
				read						
							read			
					read					
$read()$					read					
								read		
					write					
$read()$					read					
									read	
					write					
$read()$					read					
										read
					write					

为什么在 read() 的过程中还有 write 呢？是因为最近的访问时间改变了。

1.2.6 Access Paths: Writing

写和读就不太一样了，写要更复杂一些，因为写的时候可能要 allocate 一个 block，那我们现在来看看 writer() 的流程：

（1） 首先读 data bitmap，找一个 0，也就是空的地方

（2） 找到这个空的地方，并把这个 0 写成 1

（3） 现在又要去 inode 里面搞一个 disk pointer 来指向这个新的 data block

（4） 最后要把东西写到这个 data block 里面

那么如果说我们现在要创建一个新的文档呢？那就不仅和文件有关了，还和目录有关了：

1. 首先我们要去 inode bitmap 里面找一个空的 inode
2. 找到这个空的 inode，将 0 置为 1
3. 初始化 inode
4. 链接 inode 和 user 能读的名字
5. 对于目录的 inode 还有一次读写

这个就比较复杂了，下面的流程图看看就行了：

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
$create$ $(/foo/bar)$		read write	read	read	read write	read	read write			
$write()$	read write				read write			write		
$write()$	read write				read write				write	
$write()$	read write				read write					write

好的小猪猪，文件系统理论上要考试的内容就到这里啦，不过有时间的话记得往下学喔！

2 Exercise

例 55. *Imagine a file system which uses inodes to manage files on disk. Each inode consists of a file name (4 bytes), user id (2 bytes), three timestamps (4 bytes each), protection bits (2 bytes), a reference count (2 bytes), a file type (2 bytes), and the file size (4 bytes). Additionally, the inode contains 13 direct indices, 1 index to a single indirect block, 1 index to a double indirect block, and one index to a triple indirect block. Each of these indices (block pointer) is 4 bytes. The file system also stores the first 356 bytes of each file in the inode.*

(1) Three major methods of allocating disk space are introduced in our textbook. What are these three allocation methods? Which one is used in the previous file system?

(2) Assume a disk sector is 512 bytes and that each indirect block fills a single sector. What is the maximum file size for this file system? Show your work clearly. You need not do the arithmetic to get full credit.

(3) Is there any benefit to including the first 356 bytes of the file in the inode? If so, what is the reason? If not, why not?

Keys：

(1) Contiguous Allocation、Linked Allocation、Indexed Allocation；Indexed Allocation.

(2) $(512/4)^3 * 512 + (512/4)^2 * 512 + (512/4)^1 * 512 + 13 * 512 +$ **356**（这里 356 很容易漏）

(3) Yes, Efficiency in both spatial and temporal. Most files are small.（在最开始的时候提到过，说大部分文件都是 2KB 左右的）For small files (≤356 bytes), do not need to access disk twice. save disk space (internal fragmentation within blocks).

第十三章　IO_Systems

1　In-class Contents

1.1　I/O Hardware

(1) Port(端口)：用过端口确定 I/O 设备是什么

(2) Bus: 电路

(3) Polling: 轮询

(4) Polling, DMA and interrupt are three main interaction models between I/O controllers and CPUs to accomplish a complete I/O transfer.

1.2　Application I/O Interface

1.3　Kernel I/O Subsystem

有一些相对重要的概念：

定义 1.1. *I/O Scheduling:* 优化 *I/O* 请求的顺序以提升性能（尤其是磁盘 *I/O*，可以使用更好的调度算法）

定义 1.2. *Buffering:* 当数据生产者和消费者速度不一致时（例如 *CPU* 和慢速磁盘），用缓冲区过渡。网络每次发送 *1500* 字节，而程序只写 *1* 字节，缓冲可以合并或拆分数据。

定义 1.3. *Caching:* 与缓冲不同：缓存用于重用数据，缓冲用于数据传输中转。

定义 1.4. *Spooling and Device Reservation:* 防止设备使用的冲突，比如说打印机同时只能处理一个文档。