实验三（PA3）存储管理

实验简介(请认真阅读以下内容，若有违反，后果自负)
预计平均耗时/代码量: 45 小时/500 行 实验内容: 本次实验的阶段性安排如下： ● 阶段 1: 实现 cache（必做）和二级 cache（选做） - ● 阶段 2: 实现 IA-32 分段机制 - ● 阶段 3：实现 IA-32 分页机制 - 提交说明：见这里 评分依据: 代码实现占 70%, 实验报告占 30%。代码实现中完成阶段 1 占 40%, 阶段 2 占 20%，阶段 3 占 40%，选做任务不计入成绩。

进行本实验前，请在工程目录下执行以下命令进行分支整理，否则将影响成绩：

```
git commit --allow-empty -am "before starting PA3"
git checkout master
git merge pa2
git checkout -b pa3
```

普通阅读，无需提交。

必做任务，需要提交

选做任务，可以不提交

思考题，需要提交

提交要求，请仔细阅读。

Cache 的故事

不可逾越的鸿沟

随着集成电路技术的发展，CPU 越来越快，另一方面，DRAM 的速度却受限于它本身的工作原理，我们先简要解释一下这两者的差别。DRAM 的存储空间可以看成若干个二维矩阵(若干个 bank)，矩阵中的每个元素包含一个晶体管和一个电容，晶体管充当开关的作用，功能上相当于读写使能；电容用来存储一个 bit，当电容的电量大于 50%，就认为是 1，否则就认为是 0。但是电容是会漏电的，如果不进行任何操作的话，电容中的电量就会不断下降，1 最终会变成 0，存储数据就丢失了。为了避免这种情况，DRAM 必须定时刷新，读出存储单元的每一个 bit，如果表示 1，就往里面充电。DRAM 每次读操作都会读出二维矩阵中的一行，由于电容漏电的特性，在将一行数据读出之前，还要对这一行的电容进行预充电，防止读出的过程中有的电容电量下降到 50%以下而被误认为是 0。

而 CPU 的寄存器采用的是 SRAM，是通过一个触发器来存储 1 个 bit，具体来说就是 4-6 个晶体管，只要不断电，SRAM 中的数据就不会丢失，不需要定时刷新，也不需要预充电，读写速度随着主频的提升而提升。

由于 RISC 架构的指令少，格式规整，硬件的逻辑不算特别复杂，CPU 做出来之后，芯片上还多出了很多面积。为了把这些面积利用起来，架构师们提出了 cache 的概念，把剩下的面积用于 SRAM，同时也为了弥补 CPU 和 Memory 之前性能的鸿沟。CISC 的运气就没那么好了，指令多,格式不规整，硬件逻辑十分复杂，在芯片上一时间腾不出地方来放 cache，所以你在 i386 手册上找不到和 cache 相关的内容。当 CISC 架构师们意识到复杂的电路逻辑已经成为了提高性能的瓶颈时，他们才向 RISC 取经，把指令分解成微指令来执行：

```
                          R[EAX] <- M[var]
addl $1, var        =>    R[EAX] <- R[EAX] + 1
                          M[var] <- R[EAX]
```

这样就减少了硬件的逻辑，让微指令的执行流水化的同时，也可以腾出面积来做 cache 了，不过这些都是后话了。

近水楼台先得月

Cache 工作方式实际上是局部性原理的应用：

- 如果程序访问了一个内存区间，那么这个内存区间很有可能在不久的将来会被再次访问，这就是时间局部性。例如循环执行一小段代码，或者是对一个变量进行读写(addl $1, var 需要将 var 变量从内存中读出，加 1 之后再写回内存)。

- 如果程序访问了一个内存区间，那么这个内存区间的相邻区间很有可能在不久的将来会被访问，这就是空间局部性。例如顺序执行代码，或者是扫描数组元素。

相应的：

- 为了利用时间局部性，cache 将暂时存放从内存读出的数据，当 CPU 打算再次访问这些数据的时候，它不需要去访问内存，而是直接在 cache 中读出即可。就这样把数据一放，那些循环次数多达成千上万的小循环的执行速度马上提高了成千上万倍。
- 为了利用空间局部性，cache 从内存中读数据的时候，并不是 CPU 要多少读多少，而是一次多读点。Cache 向内存进行读写的基本单位是 cache block(块)。现代的 cache 设计还会在空闲的时候进行预取(prefetch)，当 CPU 一直在计算的时候，cache 会趁这段时间向内存拿点数据，将来 CPU 正好需要的话就不用再花时间拿了。

这听起来很不错，有了 cache，只要 CPU 访问 cache 的时候命中，就不需要把大量时间花费在访存上面了。不过为了保证 cache 的命中率，cache 本身也需要处理很多问题，例如：

- 一个内存区域可以被映射到多少个 cache block？少了容易冲突，多了电路逻辑和功耗都会上升。故形成了不同的 cache 组织方式，包括 direct-mapped(直接映射)，set associative(组相联)和 fully associative(全相联)。
- 冲突的时候，需要替换哪一个 cache block？这个问题的回答涉及到替换算法，最理想情况是替换那个很长时间都没访问过的 cache block，这就是 LRU 算法。但这对硬件实现来说太复杂了，译码则需要更大的代价，电路逻辑和时延都会上升，因此实际上会采用伪 LRU 算法，近似记录 cache block 访问情况，从而降低硬件复杂度。也有研究表明，随机替换效果也不会很差。
- 写 cache 的时候要不要每次都写回到内存？这个问题涉及到写策略，write through(写通)策略要求每次 cache 的写操作都同时更新内存，cache 中的数据和内存中的数据总是一致的；write back(写回)策略则等到 cache block 被替换才更新内存，就节省了很多内存写操作，但数据一致性得不到保证，最新的数据有可能在 cache 中。数据一致性在多核架构中是十分重要的，如果一个核通过访问内存拿到了一个过时的数据，用它来进行运算得到的结果就是错误的。
- 写缺失的时候要不要在 cache 中分配一个 cache block？分配就更容易引起冲突，不分配就没有用到时间局部性。

这些问题并没有完美的回答，任何一个选择都是 tradeoff，想获得好处势必要付出相应的代价，计算机就是这样一个公平的世界。

另一个值得考虑的问题是如何降低 cache 缺失的代价。一种方法是采用多级的 cache 结构，当 L1 cache 发生缺失时，就去 L2 cache 中查找，只有当 L2 cache 也发生缺失时，才去访问内存。L2 cache 通常比 L1 cache 要大，所以查找所花时间要多一些，但怎么说也比访问内存要快。还有一种方法是采用 victim cache，被替换的 cache block 先临时存放在 victim cache 中，等到要访问那个不幸被替换的 cache block 的时候，可以从 victim cache 中找回来。实验数据表明，仅仅是一个大小只有 4 项的 victim cache，对于 direct-mapped 组织方式的 cache 有十分明显的性能提升，有时候可以节省高达 90%的访存。

上面叙述的只是 CPU cache，事实上计算机世界到处蕴含着 cache 的思想。使用 printf 并没有及时输出，是因为每次只输出一个字符需要花很大的代价，因此程序会将内容先放在输出缓存区，等到缓冲区满了再输出，这其实有点 write back 的影子。像内存，磁盘这些相对于 CPU 来说的"低速"硬件，都有相应的硬件 cache 来提高性能。例如现代的 DRAM 一般都包含以下两种功能：

- 每个 bank 中都有一个行缓存，读出一行的时候会把数据放到行缓存中，如果接下来的访存操作的目的数据正好在行缓存中，就直接对行缓存进行操作，而不需要再进行预充电。
- 采用 burst(突发读写)技术，每次读写 DRAM 的时候不仅读写目的存储单元，把其相邻的存储单元也一同进行读写，这样对于一些物理存储连续的操作(例如数组)，一次 DRAM 操作就可以读写多个存储单元了。

明白 cache 存在的价值之后，你就不难理解这些技术的意义了。可惜的是，DRAM 仍旧摆脱不了定时刷新的命运。

理解 DRAM 的工作方式

NEMU 的框架代码已经模拟了 DRAM 的行缓存和 burst，尝试结合 nemu/src/memory/dram.c 中的代码来理解它们。想一想，为什么编译器为变量分配存储空间的时候一般都会对齐？访问一个没有对齐的存储空间会经历怎么样的过程？

在 NEMU 中实现 cache

Cache 的工作方式并不复杂(太复杂就不可能用硬件来实现了)，要在 NEMU 中模拟 cache 也并非难事。从 cache 组织的角度来看，cache 由若干 cache block 构成，每一个 cache block 除了包含相应的存储空间之外，还包括 tag 和一些标志位，你可以很容易地使用结构体来表示一个 cache 的组织。从操

作的角度来看，我们只需要提供 cache 的读和写两种操作就可以了。

这有点像面向对象的基本思维方式：把对象的特性抽象成一种类型。虽然 C 语言并不是面向对象的语言，但我们还是可以借助 C 语言来模拟面向对象中"封装"的特性：

```
/* define a "class" */
typedef struct Type1 {
  /* define "attributes" (members) */
  int attr1;
  char attr2;
  /* ... */

  /* define "methods" (operations) */
  int (* op1) (struct Type1 *this, int arg);
  void (* op2) (struct Type1 *this, int arg1, char arg2);
  /* ... */

} Type1;

/* define an "object" of this "class" */
Type1 obj;

/* define methods */
int add(Type1 *this, int n) {
  return this->attr1 + n;
}

/* install methods */
obj.op1 = add;

/* call methods "op1" */
obj.op1(&obj, 123);
```

函数指针使得同一个类型的不同对象的同一个操作可以有不同的具体实现，换句话说就是，把多个不同的函数抽象成统一的接口. 相信你已经在 PA2 中领教过函数指针的威力了。采用面向对象的思想，要定义多个不同的对象会变得十分方便。

关于 cache 具体如何工作，课上都已经详细讲过，这里就不另外叙述了。一个需要注意的地方是"读写数据跨越 cache block 的边界"，这时候你需要通过两次读写 cache 的操作来完成它。框架代码提供了一个专门用于读写不对齐内存区域的宏 unalign_rw() (在 nemu/include/macro.h 中定义)，使用它可以较方便地处理上述情况。

必做任务 1：实现一级 Cache

在 NEMU 中实现一个 cache，它的性质如下：

- cache block 存储空间的大小为 64B
- cache 存储空间的大小为 64KB
- 8-way set associative
- 标志位只需要 valid bit 即可
- 替换算法采用随机方式
- write through
- not write allocate

你还需要在 restart()函数中对 cache 进行初始化，将所有 valid bit 置为无效即可。实现后，修改 nemu/src/memory/memory.c 中的 hwaddr_read()和 hwaddr_write()函数，让它们读写 cache，当 cache 缺失时才读写 DRAM。

我们建议你将 cache 实现成可配置的，一份代码可以适用于各种参数的 cache，以减少重复代码. 这也是对 cache 知识的一次很好的复习。

观察 Cache 的作用

实现 cache 后，让 NEMU 运行 matrix-mul 的测试用例。你可以声明一个计时变量来模拟访存的代价，单位是 CPU 周期。每当 cache 命中时，计时变量增加 2；cache 缺失时，计时变量增加 200。为了避免溢出，最好将计时变量声明成 uint64_t 类型。当 matrix-mul 运行结束时，观察它总共运行了多少"时间"。尝试修改 cache 的各种参数(例如把 cache 总大小改成 256B)，重新运行 matrix-mul，观察它的"运行时间"。也可以统计更多性能指标，例如命中率等。

如何让矩阵乘法算得更快一直一个研究热点。从局部性原理的角度来进行优化的一个算法是 Cannon 算法，有兴趣的同学可以看看它是怎么工作的。

NEMU 作为一款模拟器，它的好处是可以轻而易举地修改一个 cache 的"构造",而不需要做一个真正的 cache 才开始测试。自从 cache 的概念被提出来，学术界提出了五花八门、数不胜数的 cache,但被工业界采纳的 cache 却寥寥无几，究其原因只有一个--纸上谈兵,无法用硬件实现。 NEMU 作为一个教学实验,并不要求大家实现一个真正的 cache，但也希望大家能明白一个道理：做事情要落到实处才有价值(理论工作除外)。如果对 cache 的硬件实现感兴趣,可以尝试用 verilog 写一个 direct-mapped，只有 4 项，可综合的小 cache。

选做任务 1：实现二级 Cache

在 NEMU 中实现一个 L2 cache，它的性质如下：

- cache block 存储空间的大小为 64B
- cache 存储空间的大小为 4MB
- 16-way set associative
- 标志位包括 valid bit 和 dirty bit
- 替换算法采用随机方式
- write back
- write allocate

你还需要在 restart()函数中对 cache 进行初始化，将所有 valid bit 置为无效即可. 把之前实现的 cache 作为 L1 cache，修改它缺失的操作，让它读写 L2 cache，当 L2 cache 缺失时才读写 DRAM。

温馨提示

PA3 阶段 1 到此结束。

IA-32 的故事

从 Segmentation fault 说起

相信你一定写过一些触发了 Segmentation fault(段错误)的程序，例如数组访问越界，空指针引用等等，这些错误的共同点是访问了非法的内存区域. 我们很容易在 Linux 下编写一个触发段错误的程序：

```
#include <stdio.h>
int main() {
  printf("%d\n", *(int *)NULL);
  return 0;
}
```

编译运行后，你会看到屏幕上输出了

```
int main() {
  asm volatile ("cli");
  while(1);
  return 0;
}
```

编译运行后，你同样会看到段错误的信息。这个恶意程序试图执行用于关中断的指令，如果执行成功，操作系统将无法响应外部中断，除非恶意程序主动放弃执行，它将独占整个系统的所有资源；从用户的角度来看，他会看到电脑死机。因此，cli 指令不应该让一般的程序随意执行，要执行它需要有一定的权限。上述恶意程序因为执行了无权限的操作，而被操作系统杀死，从而保证系统的安全。

那么，操作系统究竟是如何知道一个程序执行了非法操作(非法访问内存，执行非法指令等)？这是因为现代的 CPU 带有保护机制，当 CPU 捕获到这些非法操作的时候，它会抛出异常通知操作系统，操作系统会进行相应的处理，一般是杀死那个执行非法操作的程序. 如果你使用过 Online Judge，应该会看到过 Run-time Error 的信息，这是因为 Online Judge 的守护进程知道你提交的程序要被杀死了。

再深入一步, CPU 是怎么捕获这些非法操作的？答案就在接下来的故事中。在这之前，你可能会问"为什么类似 cli 的非法操作也会称为段错误？" 实际上这是有历史原因的，在以前的 CPU 架构中没有这么强大的保护功能，一般的非法操作都和分段机制有关(例如内存访问超越了段界限)，因此被称为 Segmentation fault。但这种称呼直到分页机制诞生之后也没有改变，于是便一直沿用至今。

混沌初开

故事追溯到 IA-32 诞生之前，在那时, 8086 曾经主宰了计算机的一切。那是

一个 16 位的世界，所有寄存器都是 16 位的，貌似最多只能访问 2^16=64KB 的内存。但事实并非如此，8086 通过引入一系列的段寄存器(segment register)来开辟更广阔的世界：

physical_address = (seg_reg << 4) + offset

其中 seg_reg 为某个段寄存器的值，offset 为寻址的偏移量，由通用寄存器或者立即数给出。例如当数据段寄存器 DS 的值为 0x8765，AX 寄存器的值为 0x1234，那么一次[DS:AX]的寻址将会访问物理地址

[DS:AX] = [0x8765:0x1234] = (0x8765 << 4) + 0x1234 = 0x87650 + 0x1234 = 0x88884

这条规则将每一次内存寻址都和某一个段寄存器绑定起来，通过借助段寄存器的信息，计算机可以访问 1MB 的内存，当然，8086 需要有 20 根地址线。后来人们把 8086 的这种寻址模式成为实模式(real mode)，因为程序能够感知到一次内存访问的真实物理地址。

然而，seg_reg 和 offset 并不是可以随便搭配的，8086 中有一些捆绑的约定

- 取指令总是使用 CS(code segment)寄存器和 IP(instruction pointer)绑定
- 大部分的内存数据访问都是使用 DS(data segment)寄存器，例如 mov %ax,(%bx)，用寄存器传输语言(RTL)来描述就是

```
M[DS:BX] <- R[AX]
```

但也可以显式指定使用 ES(extra segment)

- 堆栈相关的内存访问总是使用 SS(stack segment)寄存器，包括使用 push 和 pop 指令，或者使用 SP(stack pointer)和 BP(base pointer)进行寻址
- 一些字符串处理指令(例如 movsb)会默认使用 ES

这些约定一直沿用至今。

这样，内存就被分成 65536 个有重叠的，大小为 64KB 的段，一个段的基地址由 (seg_reg << 4)给出。如果要求段之间不能相互重叠，那就只有 16 个符合要求的段。不过这对刚刚破壳而出的计算机技术来说，已经是一个十分伟大的贡献了。借助 8086 的分段机制，计算机已经可以做很多事情，你也许很难想象当年风靡全球的吃豆子游戏竟然可以在 8086 中跑起来。

随着需求的增长，程序需要使用越来越多的内存，固定使用一个段的内存已经变得不可行了。幸好 8086 允许程序改变段寄存器的值，以达到使用更多内存的目的。因此以前的程序员在编写规模稍大的程序时，经常需要在不同的段之间

来回切换。虽然麻烦，但总比没有好。

细心的你应该会发现，8086 这种段寄存器不加保护的做法毫无安全可言，你甚至可以轻而易举地编写一个用来清除内存上除了自身以外所有数据的恶意程序。由于当时的程序员大多都十分纯洁，因此整个计算机时代也在 8086 下度过了一段十分和谐的时光。但即使没有恶意程序的困扰，1MB 的内存却将要成为计算机性能的瓶颈。当你听到了以前广为流传的"640KB 内存已经足够"，"4GB 大得简直无法想象"这类说法时，难免会呵呵地付诸一笑。你能想象你现在盯着的无所不能的机器，30 年前竟然连你现在电脑上那张桌面墙纸都放不下吗？

> **NEMU 的“实模式”**
>
> 目前 NEMU 的运行模式和 8086 的实模式有点类似，程序使用的内存地址都是实际的物理地址，不同的是，寄存器和地址都是 32 位的，而且没有分段机制。需要说明的是，这其实是 KISS 法则的产物，只是让你可以在 PA2 中将精力集中在指令系统的实现，x86 中并不存在这样的运行模式。

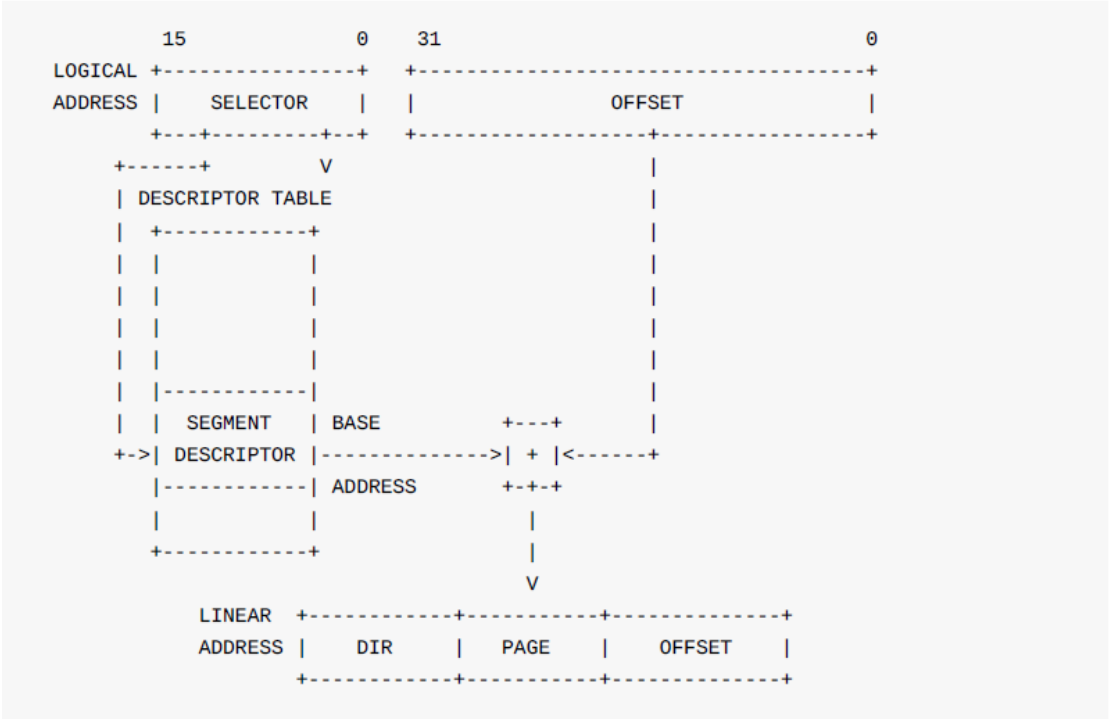
建立新秩序

16 位的 8086 已经满足不了人类了，这时 32 位的世界应运而生，它结束了内存容量成为瓶颈的 8086 时代，带领着计算机技术进入了 IA-32 的新纪元. 这个世界的名字叫 80386。

在 80386 制定的新秩序下，所有通用寄存器的长度都升级到 32 位，地址也变成了 32 位，这意味着寻址的范围扩大到了 2^32 = 4GB. "4GB 怎么可能用得完？" 要知道，这个新世界刚建立的时候,内存还都只是 1MB 的，因为在 8086 的世界里，再大的内存也是浪费。

既然一个通用寄存器的长度就已经是 32 位，这已经足够访问 4GB 的内存空间了，是不是就可以把段寄存器去掉呢？理论上是的，但在现实中，工业界还得考虑一个关系到产品生死存亡的问题：兼容。要知道，不支持兼容的产品注定是要被市场和历史抛弃的(Intel 的 IA-64 就是这样被无情抛弃了)。于是 80386 中带有了一个神奇的开关，只有触发了这个开关，才能踏入这个全新的世界。这个开关放在一个叫 CR0(control register 0)的寄存器中的 PE 位，计算机可以决定自己留在哪个世界。

如果计算机没有打开这个神奇的开关，那么段寄存器的作用和寻址方式都和 8086 一模一样。但在 80386 的世界里，分段的寻址方式发生了很大的改变。首先来感受一下 80386 中建立的新秩序：



怎么样？是不是看上去很厉害的样子？在进行进一步解释之前，我们先来消除你心中最大的疑问：为什么要把分段的过程搞得如此复杂？还记得 8086 那个混沌的时代吗？随着历史的发展,8086 暴露出了两个急需解决的问题：

- 1MB 内存容量的瓶颈
- 恶意程序等安全问题

请记住，分段过程搞得如此"复杂"，就是为了解决这两个历史遗留问题。

豁然开朗的视野

为了解决内存容量瓶颈, 80386 已经使用了 32 位的 CPU 架构。在新的分段机制里面，我们自然也希望段的基地址是 32 位的，同时也希望段的大小可以自由设定(而不像 8086 中是固定的 64KB)， 还希望能够设定粒度大小，段类型等各种属性... 这无非是希望分段机制用起来可以更加灵活(例如给一个小程序分配一个很大的段是没有必要的)，而段寄存器只有 16 位，连 32 位的基地址都放不下。别着急，这都是在 80386 的预料之中, 80386 把一个段的各种属性放在一起，组成一个段描述符(Segment Descriptor)。所谓段描述符，就是用来描述一个段的属性的数据结构，如果能有办法找到一个段描述符，就可以找到相应的段了。一个用于描述代码段和数据段的段描述符结构如下：

```
        DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

 31                23                15                7               0
+-----------------+-+-+-+-+---------+-+-----+-+-----+-+-----------------+
|                 | | | |A|         | |     | |     | |                 |
|   BASE 31..24   |G|X|O|V| LIMIT   |P| DPL |1| TYPE|A|  BASE 23..16    | 4
|                 | | | |L| 19..16  | |     | |     | |                 |
|-----------------+-+-+-+-+---------+-+-----+-+-----+-+-----------------|
|                                     |                                 |
|        SEGMENT BASE 15..0           |       SEGMENT LIMIT 15..0       | 0
|                                     |                                 |
+-------------------------------------+---------------------------------+

          A       - ACCESSED
          AVL     - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
          DPL     - DESCRIPTOR PRIVILEGE LEVEL
          G       - GRANULARITY
          P       - SEGMENT PRESENT
```

段描述符竟然有 64 位！段寄存器根本放不下，于是只好把它们放到内存里了。那计算机要怎么找到内存中的一个段描述符呢？聪明的你应该马上想到：用指针！但你没有觉得哪里不对吗？

- 在 80386 里面，指针都是 32 位的，段寄存器还是放不下啊！
- 即使段寄存器能够放下一个 32 位的指针，计算机想切换到其它段的时候，怎么知道其它段描述符在哪里呢？

80386 想出了一种同时解决这两个问题的方法，那就是你经常使用的数组！80386 把内存中的某一段数据专门解释成一个数组，名字叫 GDT(Global Descriptor Table，全局描述符表)，数组的一个元素就是一个段描述符。这样一来就可以通过下标索引的方法来找到所有的段描述符啦。于是在 80386 的世界里，原来的段寄存器就用来存放段描述符的索引，另外还包含了一些属性，这样的一个结构叫段选择符(Selector)(**更正：i386 手册相应的图中位域标识有误，实际上 RPL 占 2bit, TI 占 1bit**)：

```
 15                      3 2   0
+-------------------------+-+---+
|                         |T|   |
|          INDEX          | |RPL|
|                         |I|   |
+-------------------------+-+---+

 TI  - TABLE INDICATOR
 RPL - REQUESTOR'S PRIVILEGE LEVEL
```

思考题 1：GDT 能有多大
你能根据段选择符的结构，计算出 GDT 最大能容纳多少个段描述符吗？

剩下的问题就是，怎么找到这个 GDT 呢？由于 GDT 是全局唯一的，问题就很好解决了：在 80386 中引入一个寄存器 GDTR，专门用来存放 GDT 的首地址和长度。需要注意的是，这个首地址是线性地址，使用这个地址的时候不需要再次经过分段机制的地址转换。最后 80386 和操作系统约定，让操作系统事先把 GDT 准备好，然后通过一条特殊的指令把 GDT 的首地址和长度装载到 GDTR 中，计算机就可以开启上述的分段机制了。

思考题 2：为什么是线性地址
GDTR 中存放的 GDT 首地址可以是虚拟地址吗？为什么？

事实上，80386 还允许每个进程拥有自己的描述符表，称为 LDT(Local Descriptor Table，局部描述符表)，它的结构和 GDT 一模一样，同样地也有一个 LDTR 来存放 LDT 的位置(实际上存放的是 LDT 段在 GDT 中的索引，详细信息请查阅 i386 手册)。为了指示 CPU 在哪一个描述符表里面做索引，在段选择符中有 1 个 TI 位专门来做这件事。但由于现代操作系统弱化了分段的使用，故通常不会使用 LDT，只使用 GDT 就足够了。

现在回去看那个好像很厉害的图，你已经可以理解 80386 的分段机制了：

1. 通过段寄存器中的段选择符 TI 位决定在哪个表中进行查找。

2. 根据 GDTR 或 LDTR 读出表的首地址。

3. 根据段寄存器中的段选择符的 index 位在表中进行索引，找到一个段描述符。

4. 在段描述符中读出段的基地址，和虚拟地址(也称逻辑地址)相加，得出线性地址。

至于在什么情况下使用哪一个段寄存器，80386 继承了 8086 中段寄存器捆绑约定，具体内容请阅读上文，或者查阅 i386 手册。

思考题 3：如何提高寻找段描述符的效率？

在上述 4 个步骤中，如果段寄存器的内容没有改变，前 3 个步骤的结果都是一样的。注意到对 GDT 或 LDT 做索引是要访问内存的，如果每次寻址都需要重复前 3 步，就会产生很多不必要的内存访问。你能想到有什么办法来避免这些不必要的内存访问吗？请查阅 i386 手册，对比一下你的想法和 80386 的实现是否一样。

绕了一大圈，其实还是回到了 base + offset 的分段寻址方式上，但对这个过程的理解让你看到了在真实的计算机上是如何进行最朴素的段式存储管理。你不需要记住段描述符这些数据结构的细节(需要了解的时候可以查阅 i386 手册)，更重要的是从问题驱动的角度去理解"为什么要弄得这么复杂"。

等级森严的制度

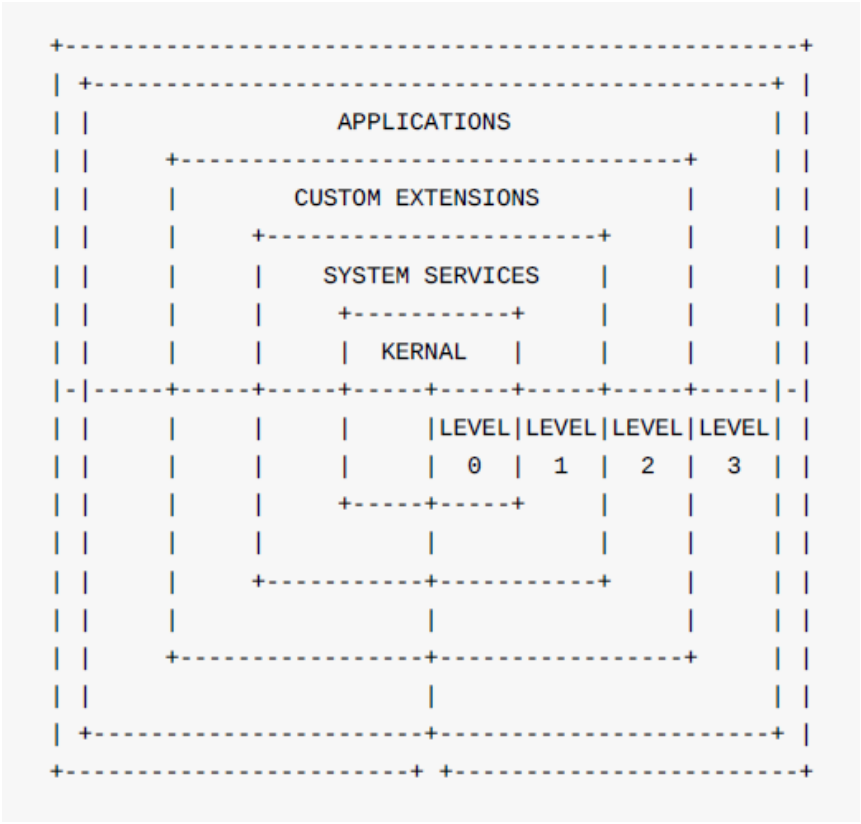
为了构建计算机和谐社会，80386 的前身 80286 就已经引入了保护模式(protected mode)和特权级(privilege level)的概念。但由于 80286 并不能很好地兼容 8086 操作系统和程序，因此 80286 并未得到广泛使用。80386 继续发扬保护模式的思想：简单地说，只有高特权级的进程才能去执行一些系统级别的指令(例如之前提到的 cli 指令等)，如果一个特权级低的进程尝试执行一条它没有权限执行的指令，CPU 将会抛出一个异常。一般来说，最适合担任系统管理员的角色就是操作系统内核了，它拥有最高的特权级，可以执行所有指令；而除非经过允许，运行在操作系统上的用户进程一般都处于最低的特权级，如果它试图破坏社会的和谐，它将会被判"死刑"。

在 80386 的新世界里，存在 0, 1, 2, 3 四个特权级，0 特权级最高，3 特权级最低。特权级 n 所能访问的资源，在特权级 0~n 也能访问。不同特权级之间的关系就形成了一个环：内环可以访问外环的资源，但外环不能进入内环的区域，因此也有"ring n"的说法来描述一个进程所在的特权级。

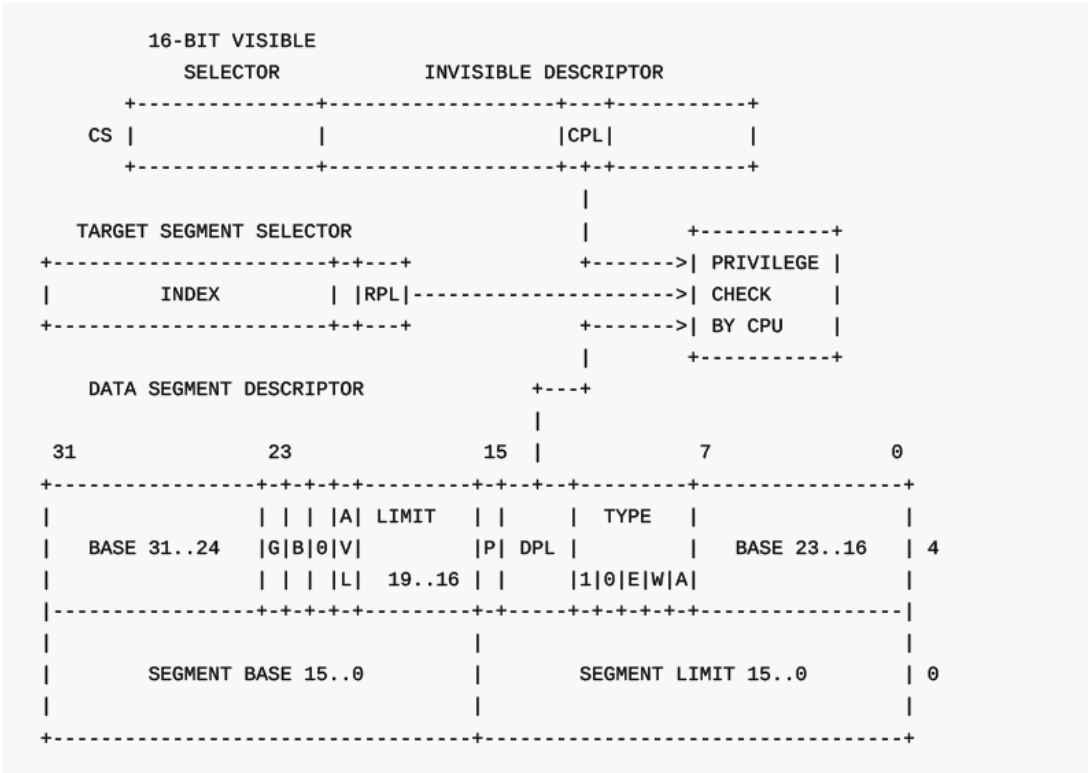
虽然 80386 提供了 4 个特权级，但大多数通用的操作系统只会使用 0 级和 3 级：内核处在 ring 0，一般的程序处在 ring 3，这就已经起到保护的作用了。那 CPU 是怎么判断一个进程是否执行了无权限操作呢？在这之前，我们还得了解一下 80386 中引入的与特权级相关的概念：

- 在段描述符中含有一个 DPL 域(Descriptor Privilege Level)，它描述了一个段所在的特权级。
- 在段选择符中含有一个 RPL 域(Requestor's Privilege Level)，它描述了请求者所在的特权级。
- CPL(Current Privilege Level)指示当前进程的特权级，一般来说它和当前

CS 寄存器所指向的段描述符(也就是当前进程的代码段)的 DPL 相等。



80386 会在段寄存器更新的时候(也就是切换到另一个段的时候)进行特权级的检查。我们以数据段的切换为例：



一次数据段的切换操作是合法的，当且仅当

```
target_descriptor.DPL >= requestor.RPL        # <1>
target_descriptor.DPL >= current_process.CPL    # <2>
```

两式同时成立，注意这里的 >= 是数值上的(numerically greater)。<1>式表示请求者有权限访问目标段，<2>式表示当前进程也有权限访问目标段。如果违反了上述其中一式，此次操作将会被判定为非法操作，CPU 将会抛出异常，通知操作系统进行处理。

对 RPL 的补充

你可能会觉得 RPL 十分令人费解，我们先举一个生活上的例子。

- 假设你到银行找工作人员办理取款业务，这时你就相当于 requestor，你的账户相当于 target_descriptor，工作人员相当于 current_process。业务办理成功是因为
  - ✧ 你有权限访问自己的账户(target_descriptor.DPL >= requestor.RPL)
  - ✧ 工作人员也有权限对你的账户进行操作(target_descriptor.DPL >= current_process.CPL)
- 如果你想从别人的账户中取钱，虽然工作人员有权限访问别人的账户(target_descriptor.DPL >= current_process.CPL)，但是你却没有权限访问(target_descriptor.DPL < requestor.RPL)，因此业务办理失败。
- 如果你打算亲自操作银行系统来取款，虽然账户是你的(target_descriptor.DPL >= requestor.RPL)，但是你却没有权限直接对你的账户金额进行操作(target_descriptor.DPL < current_process.CPL)，因此你很有可能会被保安抓起来。

在计算机中也存在类似的情况：用户进程(requestor)想对它自己拥有的数据(位于 target_descriptor 所描述的段中)进行一些它没有权限的操作，它就要请求有权限的进程(current_process，通常是操作系统内核)来帮它完成这个操作，于是就会出现"内核代表用户进程进行操作"的场景，但在真正进行操作之前，也要检查这些数据是不是真的是用户进程有权使用的数据。

通常情况下，内核运行在 ring 0，CPL 为 0，因此有权限访问所有的段；而用户进程运行在 ring 3，CPL 为 3，这就决定了它只能访问同样处在 ring3 的段。这样，只要操作系统内核将 GDT，以及下文将要提到的页表等重要的数据结构放

在 ring 0 的段中，恶意程序就永远没有办法访问到它们。

上述的规则只是针对切换数据段的行为，在不同的场景下有不同的规则，这里就不一一列举了，需要了解的时候可以查阅 i386 手册。可以看到在 80386 的保护模式下，通过特权级的概念可以有效辨别出进程的非法操作，让恶意程序无所遁形，为构建计算机和谐社会作出了巨大的贡献。

遗憾的是，根据 KISS 法则，我们并不打算在 NEMU 中引入 IA-32 保护机制。我们让所有用户进程都运行在 ring 0，虽然所有用户进程都有权限执行所有指令，不过由于 PA 中的用户程序都是我们自己编写的，一切还是在我们的控制范围之内。但我们最好在 NEMU 的代码中尽可能插入 assertion，以便及时捕捉一些本来应该由 IA-32 保护机制捕捉的错误(例如段描述符的 present 位为 0)。

在 NEMU 中实现分段机制

理解 IA-32 分段机制之后，你需要在 NEMU 中实现它。一方面，你需要在 kernel 中加入切换到保护模式的代码，你只需要在 kernel/include/common.h 中定义宏 IA32_SEG，然后重新编译 kernel 就可以了。重新编译后，kernel/src/start.S 的行为如下：

1. 设置 GDTR
2. 将 CR0 的 PE 位置 1，切换到保护模式
3. 使用 ljmp 设置 CS 寄存器
4. 设置 DS, ES, SS 寄存器
5. 为 C 代码设置堆栈
6. 跳转到 init()函数继续进行初始化工作

你需要根据 IA-32 分段机制理解上述代码。另一方面，你需要在 NEMU 中添加分段机制的功能，以便让上述代码成功执行。具体的，你需要：

- 在 CPU_state 结构中添加 GDTR，CR0 和各种段寄存器，包括 CS，DS，ES，SS，其具体结构请参考 i386 手册。80386 中还引入了两个新的段寄存器 GS 和 FS，不过我们不会用到它们，因此可以不模拟它们的功能。LDT 我们也不会用到，和 LDT 相关的内容也不必模拟。你还需要在 restart()函数中对 CR0 寄存器进行初始化，让模拟的计算机在"开机"的时候运行在"实模式"下。
- 添加 lgdt 指令。
- 添加 opcode 为 0F 20 和 0F 22 的 mov 指令，使得我们可以设置/读出 CR0。设置 CR0 后，如果发现 CR0 的 PE 位为 1，则进入 IA-32 保护模式，从此所有虚拟地址的访问(包括 swaddr_read()和 swaddr_write())都需要经过段级

地址转换。

- 为了实现段级地址转换，你需要对 swaddr_read()和 swaddr_write()函数作少量修改。以 swaddr_read()为例，修改后如下：

```
uint32_t swaddr_read(swaddr_t addr, size_t len, uint8_t sreg) {
  assert(len == 1 || len == 2 || len == 4);
  lnaddr_t lnaddr = seg_translate(addr, len, sreg);
  return lnaddr_read(lnaddr, len);
}
```

其中 sreg 记录了当前段级地址转换所用到的段寄存器的编码，关于段寄存器的编码，请查阅 i386 手册。你需要理解段级地址转过的过程，然后实现 seg_translate()函数。再次提醒，在 NEMU 中，只有进入保护模式之后才会进行段级地址转换。

- 为了实现段寄存器的捆绑规则，你还需要

1. 在 Operand 结构体中添加成员 sreg:

```
--- nemu/include/cpu/decode/operand.h
+++ nemu/include/cpu/decode/operand.h
@@ -8,12 +8,15 @@
 typedef struct {
  uint32_t type;
  size_t size;
  union {
    uint32_t reg;
-    swaddr_t addr;
+    struct {
+      swaddr_t addr;
+      uint8_t sreg;
+    };
    uint32_t imm;
    int32_t simm;
  };
  uint32_t val;
  char str[OP_STR_SIZE];
 } Operand;
```

2. 修改 read_ModR_M()中的代码，以确定是和 DS, SS 中的哪一个进行捆绑，然后设置 rm->sreg，这样 swaddr_read()和 swaddr_write()就可以使用正确的段寄存器了。

3. 修改宏 MEM_W()和 MEM_R()，以及所有调用 swaddr_read()和 swaddr_write()的代码，为它们添加段寄存器的参数。特别地:

- opcode 为 A0 , A1, A2, A3 的 mov 指令使用 DS 寄存器。
- 一些堆栈操作指令会隐式使用 SS 寄存器。
- instr_fetch()总是使用 CS 寄存器。
- 在 monitor 中，x 和 p 命令读出内存时，使用 DS 寄存器；bt 命令打印栈帧链时，使用 SS 寄存器。
- 关于字符串操作指令使用的段寄存器，请查阅 i386 手册。

- 添加 opcode 为 8E 的 mov 指令，使得我们可以设置段寄存器。设置段寄存器时，还需要将段的一些属性读入到段寄存器的描述符 cache 部分(在 i386 手册中被称为"隐藏部分"，invisible part)，我们只需要读入段的 base 和 limit 就可以了，其它属性在 NEMU 中不使用. 另外还有两点需要注意：

1. GDTR 中存放的 GDT 首地址是线性地址。

2. IA-32 中规定不能使用 mov 指令设置 CS 寄存器，但切换到保护模式之后，下一条指令的取指就要用到 CS 寄存器了。解决这个问题的一种方式是在 restart() 函数中对 CS 寄存器的描述符 cache 部分进行初始化，将 base 初始化为 0, limit 初始化为 0xffffffff 即可。

- 为了设置 CS 寄存器，你需要实现 ljmp 指令，即 JMP ptr16:32 形式的 jmp 指令，其作用是"Jump intersegment, 6-byte immediate address"，更多信息请查阅 i386 手册。

必做任务 2：在 NEMU 中实现分段机制

根据上述的讲义内容，在 NEMU 中模拟 IA-32 分段机制，如有疑问，请查阅 i386 手册。在 libcommon/x86-inc 目录下的头文件中定义了一些和 x86 相关的宏和结构体，你可以在 NEMU 中包含这些头文件来使用它们。

温馨提示

PA3 阶段 2 到此结束。

迈进新时代

80386 使用了升级版的段式存储管理，听上去很不错，但实际上并不是这样。

> 思考题 4：段式存储管理的缺点
>
> 参考手册内容，段式存储管理有什么缺点？(说不定考研会出这道题喔 ^_^)

尽管这个升级版的段式存储管理是 80386 提出的，但在手册上也提到可以想办法"绕过"它来提高性能：将段的基地址设成 0，长度设成 4GB，这就是 i386 手册中提到的"扁平模式"，这样虚拟地址就和分段之后得到的线性地址一样了。当然，这里的"绕过"并不是简单地将分段机制关掉(事实上也不可能关掉)，毕竟段级保护机制的特性是计算机法制社会最重要的特征，抛弃它是十分不明智的。

超越容量的界限

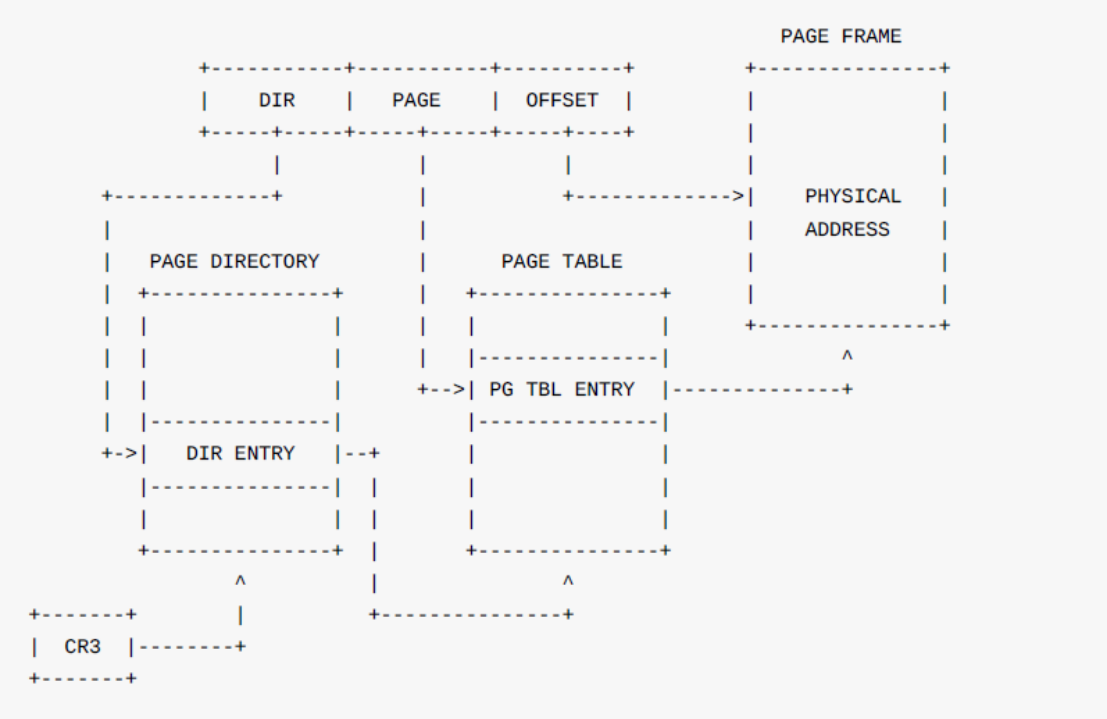
为了克服分段机制的缺点，80386 作出了计算机发展史上又一个具有里程碑意义的贡献：提供了分页机制。当然，80386 刚建立的时候，它不能强迫大家使用分页机制，因此 80386 也提供了一个神奇的开关，只有打开了开关才能启用分页机制。这个神奇的开关在 CR0 寄存器中的 PG 位，分页机制只能在保护模式下启用，这也算是给 8086 时代的程序一个过渡的选择。

> 思考题 5：页式存储管理的优点
>
> 回忆课堂内容，页式存储管理有什么优点？(说不定考研会出这道题喔 ^_^)

正是因为这些优点，在现代的通用操作系统中，分页机制基本上"取代"了分段机制，成为计算机存储管理的主要方式。

> 被"淘汰"的段式存储管理
>
> gcc 编译程序时也默认假设程序将来运行在扁平模式下。你可以在 kernel/src/start.S 中进行以下尝试：
>
> - 在 kernel 中为 GDT 增加一个新的段描述符，它的基地址改为 0x4000000，其它属性和数据段一样。
> - 把这个新的段描述符装载到 SS 寄存器，作为新的堆栈段。
> - 把%esp 的初值改成 0x4000000。
>
> 修改后，运行任意用户程序，你会发现 NEMU 输出 HIT BAD TRAP 的信息. 分析相应的汇编代码，你能厘清运行出错的根本原因吗？

我们也是先上一张图给大家一个感观上的认识：



80386 采用了二级页表的结构，为了方便叙述，80386 给第一级页表取了个新名字叫"页目录"。虽然听上去很厉害，但其实原理都是一样的。每一张页目录和页表都有 1024 个表项，每个表项的大小都是 4 字节，除了包含页表(或者物理页)的基地址，还包含一些标志位信息。因此，一张页目录或页表的大小是 4KB，要放在寄存器中是不可能的，因此它们也是放在内存中。为了找到页目录，80386 提供了一个 CR3(control register 3)寄存器，专门用于存放页目录的基地址。这样，页级地址转换就从 CR3 开始，一步一步地进行，最终将线性地址转换成真正的物理地址，这个过程称为一次 page walk。同样地，CPU 也不知道什么叫"页表"，它不能识别一段数据是不是一个页表，因此页表的结构也需要操作系统事先准备好。

思考题 6：一些问题

- 80386 不是一个 32 位的世界吗，为什么表项中的基地址信息只有 20 位，而不是 32 位？
- 手册上提到表项(包括 CR3)中的基地址都是物理地址，物理地址是必须的吗？能否使用虚拟地址或线性地址？
- 为什么不采用一级页表？或者说采用一级页表会有什么缺点？

我们不打算给出分页过程的详细解释，请你结合 i386 手册的内容和课堂上的知识，尝试理解 IA-32 分页机制，这也是作为分页机制的一个练习。i386 手册中包含你想知道的所有信息，包括这里没有提到的表项结构，地址如何划分等。

页级转换的过程并不总是成功的，因为 80386 也提供了页级保护机制，实现保护功能就要靠表项中的标志位了。我们对一些标志位作简单的解释：

- present 位表示物理页是否可用，不可用的时候又分两种情况：

  1. 物理页面由于交换技术被交换到磁盘中了，这就是你在课堂上最熟悉的 Page fault 的情况之一了，这时候可以通知操作系统内核将目标页面换回来，这样就能继续执行了。

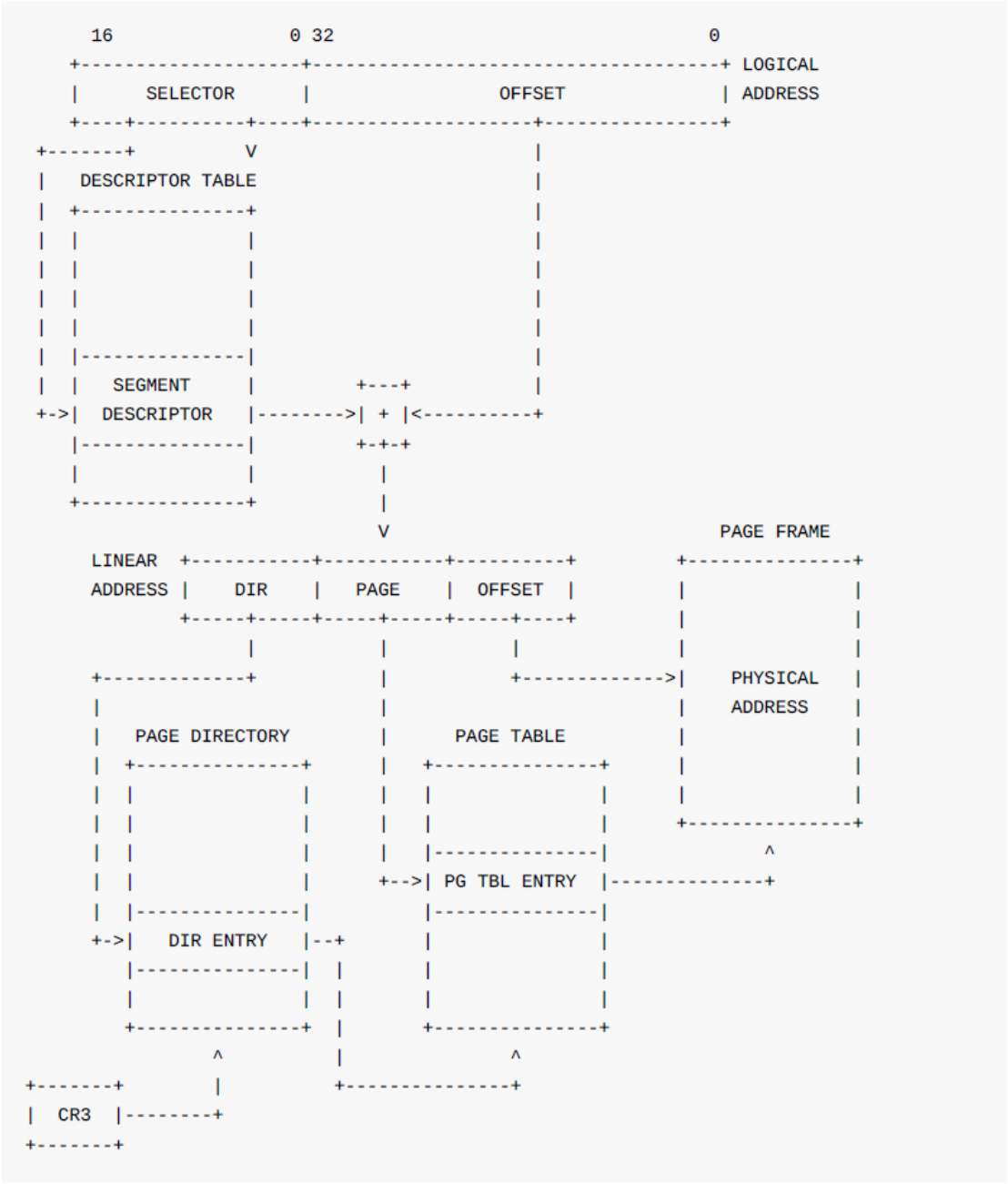
  2. 进程试图访问一个未映射的线性地址，并没有实际的物理页与之相对应，因此这就是一个非法操作咯。

- R/W 位表示物理页是否可写，如果对一个只读页面进行写操作，就会被判定为非法操作。

- U/S 位表示访问物理页所需要的权限，如果一个 ring 3 的进程尝试访问一个 ring 0 的页面，当然也会被判定为非法操作。

思考题 7：空指针真的是“空”吗？
程序设计课上老师告诉你，当一个指针变量的值等于 NULL 时，代表空，不指向任何东西。仔细想想，真的是这样吗？当程序对空指针解引用的时候，计算机内部具体都做了些什么？你对空指针的本质有什么新的认识？

最后我们用一张图来总结 80386 建立的新时代:



我的妈呀，还真够复杂的. 不过仔细看看，你会发现这只不过是把分段和分页结合起来罢了，用数学函数来理解，也只不过是个复合函数：

```
hwaddr = page(seg(swaddr))
```

而"虚拟地址空间"和"物理地址空间"这两个在操作系统中无比重要的概念，也只不过是这个复合函数的定义域和值域而已。然而这个过程中涉及到的很多细节还是被我们忽略了。理解 80386 这个新时代存在的意义和价值，可以帮助你从更本质的角度来看待一些你在程序设计层次中感到模糊不清，或者甚至无法解释的问题，这也是学习这些知识的价值所在。

思考题 8：在扁平模式下如何进行保护

现代操作系统一般使用扁平模式来"绕过"IA-32 分段机制。在扁平模式中，ring 0 和 ring 3 的段区间都是 [0, 4G)，这意味着放在 ring 0 中的 GDT，页表这些重要的数据结构对于处在 ring 3 的恶意程序来说竟是一览无余！扁平模式已经不能阻止恶意程序访问这些重要的数据结构了，为何恶意程序仍然不能为所欲为？

在 NEMU 中实现分页机制

理解 IA-32 分页机制之后，你需要在 NEMU 中实现它。具体的，你需要：

- 添加 CR3 寄存器。
- 为 CR0 寄存器添加 PG 位的功能。装载 CR0 后，如果发现 CR0 的 PE 位和 PG 位均为 1，则开启 IA-32 分页机制，从此所有线性地址的访问(包括 lnaddr_read(), lnaddr_write())都需要经过页级地址转换。在 restart()函数中对 CR0 寄存器进行初始化时，PG 位要被置 0。
- 为了实现页级地址转换，你需要对 lnaddr_read()和 lnaddr_write()函数作少量修改。以 lnaddr_read()为例，修改后如下：

```
uint32_t lnaddr_read(lnaddr_t addr, size_t len) {
  assert(len == 1 || len == 2 || len == 4);
  if (data cross the page boundary) {
    /* this is a special case, you can handle it later. */
    assert(0);
  }
  else {
    hwaddr_t hwaddr = page_translate(addr);
    return hwaddr_read(hwaddr, len);
  }
}
```

你需要理解页级地址转过的过程，然后实现 page_translate()函数。另外由于我们不打算实现保护机制，在 page_translate()函数的实现中，你务必使用 assertion 检查页目录项和页表项的 present 位，如果发现了一个无效的表项，及时终止 NEMU 的运行，否则调试将会异常困难。这通常是由于你的实现错误引起的，请检查实现的正确性。再次提醒，只有进入保护模式并开启分页机制之后才会进行页级地址转换。

- 最后提醒一下页级地址转换时出现的一种特殊情况。和之前实现 cache 的时候一样，由于 IA-32 并没有严格要求数据对齐，因此可能会出现数据跨越虚拟页边界的情况，例如一条很长的指令的首字节在一个虚拟页的最后，剩下的字节在另一个虚拟页的开头。如果这两个虚拟页被映射到两个不连续的物

理页，就需要进行两次页级地址转换，分别读出这两个物理页中需要的字节，然后拼接起来组成一个完成的数据返回。MIPS 作为一种 RISC 架构，指令和数据都严格按照 4 字节对齐，因此不会发生这样的情况，否则 MIPS CPU 将会抛出异常，可见软件灵活性和硬件复杂度是计算机科学中又一对 tradeoff. 不过根据 KISS 法则，你现在可以暂时不实现这种特殊情况的处理，在判断出数据跨越虚拟页边界的情况之后，先使用 assert(0) 终止 NEMU，等到真的出现这种情况的时候再进行处理。

另一方面，你需要在 kernel 中加入分页管理相关的代码，你只需要在 kernel/include/common.h 中定义宏 IA32_PAGE，然后修改 kernel/Makefile.part 中的链接选项：

```
--- kernel/Makefile.part
+++ kernel/Makefile.part
@@ -8,1 +8,1 @@
-kernel_LDFLAGS = -m elf_i386 -e start -Ttext=0x00100000
+kernel_LDFLAGS = -m elf_i386 -e start -Ttext=0xc0100000
```

上述修改让 kernel 的代码从虚拟地址 0xc0100000 开始(由于 kernel 中设定的段描述符并没有改变段级地址转换的结果，也就是说在 kenrel 中，虚拟地址和线性地址的值是一样的，因此在描述 kernel 行为的时候，我们不另外区分虚拟地址和线性地址的概念)，这样做是为了配合分页机制的加入，更多的内容会在下文进行解释。修改后，重新编译 kernel 就可以了。

重新编译后，kernel 会发生较大的变化，现在我们来逐一解析它们。首先是 kernel/src/start.S 的代码会有少量变化，其中涉及到地址的部分都被 va_to_pa() 的宏进行过处理，这是因为程序中使用的地址都是虚拟地址，但 NEMU 会把 kernel 加载到物理地址 0x100000 处，而在 start.S 中运行的时候并没有开启分页机制，此时若直接使用虚拟地址则会发生错误。

团结力量大

上述文字提到了 3 个和内存地址相关的描述:

- Makefile.part 中的链接选项让 kernel 从虚拟地址 0xc0100000 开始。
- NEMU 把 kernel 加载到物理地址 0x100000 处。
- start.S 中 va_to_pa()的宏让地址相关的部分都减去 KOFFSET。

请你进一步思考，这三者是如何相互配合，最终让 kernel 成功在 NEMU 中运行的？尝试分别修改其中一方(例如把 Makefile.part 中的链接选项-Ttext 修改回 0x00100000，把 nemu/src/monitor/monitor.c 的 ENTRY_START 宏修改成 0x200000，或者把 start.S 中的 KOFFSET 宏修改成 0xc0000001)，编译后重新运行，并思考为什么修改后 kernel 会运行失败。

kernel 最大的变化在 C 代码中。跳转到 init()函数之后，第一件事就是为 kernel 自己创建虚拟地址空间，并开启分页机制。这是通过 init_page()函数(在 kernel/src/memory/kvm.c 中定义)完成的，具体的工作有:

1. 填写页目录项和页表项。
2. 将页目录基地址装载到 CR3 寄存器。
3. 将 CR0 的 PG 位置 1，开启分页机制。

思考题 9：地址映射

结合 kernel 的框架代码理解分页机制，阅读 init_page()函数的代码，它建立了一个从虚拟地址到物理地址的映射。请结合此处代码描述这个映射具体是怎么样的，并尝试画出这个映射：画两个矩形，左边代表虚拟地址，右边代表物理地址，并标上 0 和 4G，然后画出哪一段虚拟地址对应哪一段物理地址。你可以先用纸笔来画，然后拍照片，粘贴在实验报告中。

开启分页机制后，kernel 就可以使用虚拟地址了。接下来 kernel 的初始化工作如下:

1. 首先让%esp 加上 KOFFSET 转化成相应的虚拟地址，然后通过间接跳转进入 init_cond()函数继续进行初始化，这样 kernel 就完全工作在虚拟地址中了。
2. 使用 Log()宏输出一句话，同样地，现在的 Log()宏还是不能成功输出。
3. 初始化 MM。这里的 MM 是指存储管理器(Memory Manager)模块，它

专门负责和用户进程分页相关的存储管理。目前初始化 MM 的工作就是初始化用户进程的页目录，同时将用户进程在 0xc0000000 以上的虚拟地址映射到和 kernel 一样的物理地址。

在 loader()函数中加载 ELF 可执行文件是另一个需要理解的难点。由于我们开启了分页机制，将来用户进程将运行在分页机制之上，这意味着用户进程使用的地址都是虚拟地址，kernel 要先为用户进程创建虚拟地址空间。因此我们就可以为用户进程提供更方便的运行时环境了：

- 代码和数据位于 0x8048000 附近。
- 有自己的虚拟地址空间，最大是 4GB，不过这需要 kernel 实现交换技术. 虽然我们不打算实现交换技术，但"虚拟地址作为物理地址的抽象"这一好处已经体现出来了：原则上用户程序可以运行在任意的虚拟地址，不受物理内存容量的限制. 我们让程序的代码从 0x8048000 附近开始，这个地址已经超过了物理地址的最大值(NEMU 提供的物理内存是 128MB)，但分页机制保证了程序能够正确运行。这样，链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址，它们只要使用虚拟地址就可以了，而虚拟地址和物理地址之间的映射则全部交给 kernel 的 MM 来管理。
- %ebp 的初值为 0, %esp 的初值为 0xc0000000，堆栈大小为 1MB。
- 程序通过 nemu_trap 结束运行。
- 提供 uclibc 库函数的静态链接。

这一运行时环境规定的进程地址空间和 GNU/Linux 十分相似，我们可以不显式指定用户程序链接参数中的代码段位置，链接器会默认将代码段放置在 0x8048000 附近：

```
--- testcase/Makefile.part
+++ testcase/Makefile.part
@@ -8,2 +8,2 @@
 testcase_START_OBJ := $(testcase_OBJ_DIR)/start.o
-testcase_LDFLAGS := -m elf_i386 -e main -Ttext-segment=0x00800000
+testcase_LDFLAGS := -m elf_i386 -e main
```

为了向用户进程提供这一升级版的运行时环境，你需要对加载过程作少量修改。此时 program header 中的 p_vaddr 就真的是用户进程的虚拟地址了，但它并不在 kernel 的虚拟地址空间中，所以 kernel 不能直接访问它. kernel 要做的事情就是：

- 按照 program header 中的 p_memsz 属性，为这一段 segment 分配一段不小于 p_memsz 的物理内存。

- 根据虚拟地址 p_vaddr 和分配到的物理地址正确填写用户进程的页目录和页表。
- 把 ELF 文件中的 segment 的内容加载到这段物理内存。

这一切都是为了让用户进程在将来可以正确地运行：用户进程在将来使用虚拟地址访问内存，在 kernel 为用户进程填写的页目录和页表的映射下，虚拟地址被转换成物理地址，通过这一物理地址访问到的物理内存，恰好就是用户进程想要访问的数据。物理内存并不是可以随意分配的，如果把 kernel 正在使用的物理页分配给用户进程，将会发生致命的错误。NEMU 模拟的物理内存只有 128MB，我们约定低 16MB 的空间专门给 kernel 使用，剩下的 112MB 供用户进程使用，即第一个可分配给用户进程的物理页首地址是 0x1000000。

不过这一部分的内容实现起来会涉及很多细节问题，出现错误是十有八九的事情。为了减轻大家的负担，我们已经为大家准备了一个内存分配的接口函数 mm_malloc()，其函数原型为：

```
uint32_t mm_malloc(uint32_t va, int len);
```

它的功能是为用户进程分配一段以虚拟地址 va 开始，长度为 len 的连续的物理内存区间，并填写为用户进程准备的页目录和页表，然后返回这一段物理内存区间的首地址。由于 mm_malloc()的实现和操作系统实验有关，我们没有提供 mm_malloc()函数的源代码，而是提供了相应的目标文件 mm_malloc.o，Makefile 中已经设置好相应的链接命令了，你可以在 loader()函数中直接调用它，完成上述内存分配的功能。你不必为此感到沮丧，我们已经为你准备了另外一个简单的分页小练习（见下文）。有兴趣的同学可以尝试编写自己的 mm_malloc()函数，也可以尝试通过 mm_malloc.o 破解相应的源代码。

加载 ELF 可执行文件之后，kernel 还会为用户进程分配堆栈，堆栈从虚拟地址 0xc0000000 往下生长，大小为 1MB。这样用户进程的地址空间就创建好了，更新 CR3 后，此时 kernel 已经运行在刚刚创建好的地址空间上了。接下来做的事情和之前差不多：

- loader()将返回用户程序的入口地址。
- 把%esp 设置成 0xc0000000。
- 跳转到用户程序的入口执行。

此时用户进程已经完全运行在分页机制上了。

必做任务 3：在 NEMU 中实现分页机制

根据上述的讲义内容，在 NEMU 中模拟 IA-32 分页机制，如有疑问，请查阅 i386 手册。在 lib-common/x86-inc 目录下的头文件中定义了一些和 x86 相关的宏和结构体，你可以在 NEMU 中包含这些头文件来使用它们。

选做任务 2：简易调试器

为了方便调试，你可以在 monitor 中添加如下命令：

```
page ADDR
```

这条命令的功能是输出地址 ADDR 的页级地址转换结果，当地址转换失败时，输出失败信息。你可以根据你的实际需要添加或更改这条命令的功能。

思考题 10：有本事就把我找出来！

在 kernel/src/memory/kvm.c 中，为了提高效率，我们使用了内联汇编来填写页表项，同时给出了相应的 C 代码作为参考。如果你曾经尝试用 C 代码替换内联汇编，编译后重新运行，你会看到发生了错误。事实上，作为参考的 C 代码中隐藏着一个小小的 bug，这个 bug 的藏身之术十分高超，以至于几乎不影响你对 C 代码的理解。聪明的你能够让这个嚣张的 bug 原形毕露吗？

这一部分的内容算是整个 PA 中最难理解的了（但代码量十分少），其中涉及到很多课本上没有提到的细节。不过上文的讨论还是忽略了很多细节的问题，分页机制中也总是暗藏杀机，如果你喜欢一些挑战性的工作，你可以在完成这一部分的实验内容之后尝试完成下面的蓝框题。当你把这些问题都弄明白之后，你就不会再害怕由于分页机制而造成的错误了。

思考题 11：暗藏杀机的分页机制（这 5 个问题都有一些难度哦~）

尝试回答下列问题之前，你需要保证你的分页机制实现正确，用户进程可以在你实现的分页机制上运行。否则一些问题的原因可能会被你造成的错误所掩盖，会让你百思不得其解。

- 在 init() 函数中有一处注释"Before setting up correct paging, no global variable can be used". 尝试在 main.c 中定义一个全局变量，然后在调用 init_page() 前使用这个全局变量：

```
--- kernel/src/main.c
+++ kernel/src/main.c
@@ -19,9 +19,11 @@
+volatile int x = 0;
 /* Initialization phase 1
  * The assembly code in start.S will finally jump here.
  */
 void init() {
 #ifdef IA32_PAGE
+     x = 1;
```

重新编译并运行，你发现了什么问题？请解释为什么在开启分页之前不能使用全局变量，但却可以使用局部变量(在 init_page() 函数中使用了局部变量)。细心的你会发现，在开启分页机制之前，init_page() 中仍然使用了一些全局变量，但却没有造成错误，这又是为什么？

- 在刚刚调用 init_page()的时候，分页机制并没有开启。但通过 objdump 查看 kernel 的代码，你会发现 init_page()函数在 0xc0000000 以上的地址，为什么在没有开启分页机制的情况下调用位于高地址的 init_page()却不会发生错误？

- 你已经画出 init_page()函数中创建的映射了，这个映射把两处虚拟地址映射到同一处物理地址，请解释为什么要创建这样一个映射。具体地，在 init_page()的循环中有这样两行代码：

```
pdir[pdir_idx].val = make_pde(ptable);
pdir[pdir_idx + KOFFSET / PT_SIZE].val = make_pde(ptable);
```

- 在 init() 函数中，我们通过内联汇编把 %esp 加上 KOFFSET 转化成相应的虚拟地址. 尝试把这行内联汇编注释掉，重新编译 kernel 并运行，你会看到发生了错误. 请解释这个错误具体是怎么发生的。

- 在 init() 函数中，我们通过内联汇编间接跳转到 init_cond() 函数. 尝试把这行内联汇编注释掉，改成通过一般的函数调用来跳转到 init_cond() 函数，重新编译 kernel 并运行，你会看到发生了错误. 请解释这个错误具体是怎么发生的。

选做任务 3：为用户进程创建 video memory 映射

video memory 是一段有特殊功能的内存区间，我们会在 PA4 中作进一步解释。在 loader()函数中有一处代码会调用 create_video_mapping()函数(在 kernel/src/memory/vmem.c 中定义)，为用户进程创建 video memory 的恒等映射，即把从 0xa0000 开始，长度为 320 * 200 字节的虚拟内存区间映射到从 0xa0000 开始，长度为 320 * 200 字节的物理内存区间。有兴趣的同学可以实现 create_video_mapping()函数，具体的，你需要定义一些页表(注意页表需要按页对齐，你可以参考 kernel/src/memory/kvm.c 中的相关内容)，然后填写相应的页目录项和页表项即可。注意你不能使用 mm_malloc()来实现 video memory 映射的创建，因为 mm_malloc()分配的物理页面都在 16MB 以上，而 video memory 位于 16MB 以内，故使用 mm_malloc()不能达到我们的目的。

实现完成后，让 kernel 调用 video_mapping_write_test(), video_mapping_read_test()和 video_mapping_clear()(把相应的函数调用移出条件编译块即可)，这些代码会在为用户进程创建地址空间之前向 video memory 写入一些测试数据，创建地址空间之后读出这些数据并检查.如果你的 create_video_mapping()实现有问题，你将不能通过检查。

这个 video memory 的映射将会在 PA4 中用到，现在算是一个简单的分页小练习，帮助你更好地理解 IA-32 分页机制。你现在可以选择忽略这个问题，但你会在 PA4 中重新面对它。

温馨提示

PA3 阶段 3 到此结束。

近水楼台先得月(2)

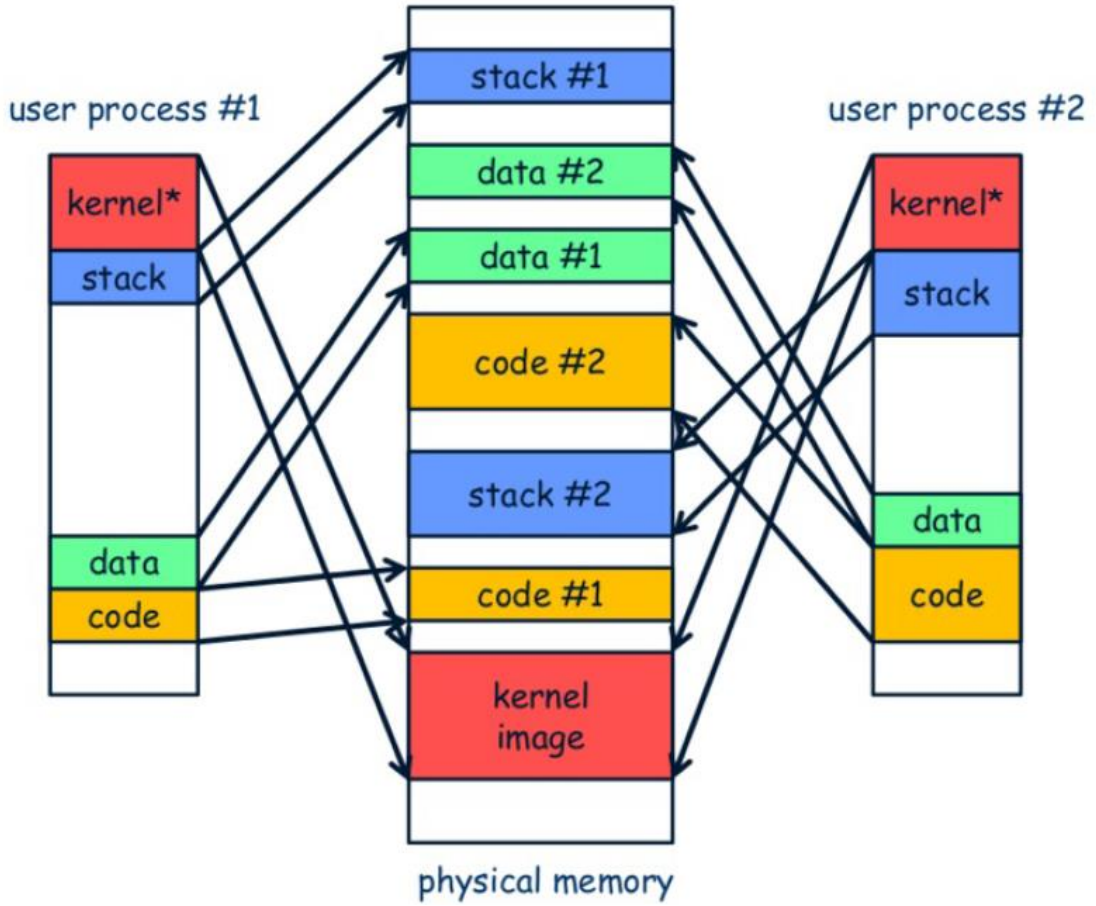
细心的你会发现，在不改变 CR3，页目录和页表的情况下，如果连续访问同一个虚拟页的内容，页级地址转换的结果都是一样的。事实上，这种情况太常见了，例如程序执行的时候需要取指令，而指令的执行一般都遵循局部性原理，大多数情况下都在同一个虚拟页中执行。但进行 page walk 是要访问内存的，如果有方法可以避免这些没有必要的 page walk，就可以提高处理器的性能了。

一个很自然的想法就是将页级地址转换的结果存起来，在进行下一次的页级地址转换之前，看看这个虚拟页是不是已经转换过了，如果是，就直接取出之前的结果，这样就可以节省不必要的 page walk 了。这不正好是 cache 的思想吗？于是 80386 加入了一个特殊的 cache，叫 TLB。我们可以从 CPU cache 的知识来理解 TLB 的组织：

- TLB 的基本单元是项，一项存放了一次页级地址转换的结果(其实就是一个页表项，包括物理页号和一些和物理页相关的标志位)，功能上相当于一个 cache block。
- TLB 项的 tag 由虚拟页号(即线性地址的高 20 位)来充当，表示这一项对应于哪一个虚拟页号。
- TLB 的项数一般不多，为了提高命中率, TLB 一般采用 fully associative 的组织方式。
- 由于页目录和页表一旦建立之后，一般不会随意修改其中的表项，因此 TLB 不存在写策略和写分配方式的问题。

实践表明，大小 64 项的 TLB，命中率可以高达 90%，有了 TLB 之后，果然大大节省了不必要的 page walk。

听上去真不错！不过在现代的多任务操作系统中，如果仅仅简单按照上述方式来使用 TLB，却会导致致命的后果。我们知道，现在的计算机可以"同时"运行多个进程，这里的"同时"其实只是一种假象，并不是指在物理时间上的重叠，而是操作系统很快地在不同的进程之间来回切换，切换的频率大约是 10ms 一次，一般的用户是感觉不到的。而让多个进程"同时"运行的一个基本条件，就是不同的进程要拥有独立的存储空间，它们之间不能相互干扰. 这一条件是通过分页机制来保证的，操作系统为每个进程分配不同的页目录和页表，虽然两个进程可能都会从 0x8048000 开始执行，但分页机制会把它们映射到不同的物理页，从而做到存储空间的隔离。当然，操作系统进行进程切换的时候也需要更新 CR3 的内容，使得 CR3 寄存器指向新进程的页目录，这样才能保证分页机制将虚拟地址映射到新进程的物理存储空间。



现在问题来了，假设有两个进程，对于同一个虚拟地址 0x8048000，操作系统已经设置好正确的页目录和页表，让 1 号进程映射到物理地址 0x1234000，2 号进程映射到物理地址 0x5678000，同时假设 TLB 一开始所有项都被置为无效。这时 1 号进程先运行，访问虚拟地址 0x8048000，查看 TLB 发现未命中，于是进行 page walk，根据 1 号进程的页目录和页表进行页级地址转换，得到物理地址 0x1234000，并填充 TLB。假设此时发生了进程切换，轮到 2 号进程来执行，它也要访问虚拟地址 0x8048000，查看 TLB，发现命中，于是不进行 page walk，而是直接使用 TLB 中的物理页号，得到物理地址 0x1234000. 2 号进程竟然访问了 1 号进程的存储空间，但 2 号进程和操作系统对此都毫不知情!

出现这个致命错误的原因是，TLB 没有维护好进程和虚拟地址映射关系的一致性, TLB 只知道有一个从虚拟地址 0x8048000 到物理地址 0x1234000 的映射关系，但它并不知道这个映射关系是属于哪一个进程的。找到问题的原因之后，解决它也就很容易了，只要在 TLB 项中增加一个域 ASID(address space ID)，用于指示映射关系所属的进程即可, MIPS 处理器就是这样做的。IA-32 的做法则比较"野蛮"，在每次更新 CR3 时强制冲刷 TLB 的内容，由于进程切换必定伴随着 CR3 的更新,因此一个进程运行的时候, TLB 中不会存在其它进程的映射关系。

必做任务 4：实现 TLB

在 NEMU 中实现一个 TLB，它的性质如下：

- TLB 总共有 64 项
- fully associative
- 标志位只需要 valid bit 即可
- 替换算法采用随机方式

你还需要在 restart()函数中对 TLB 进行初始化，将所有 valid bit 置为无效即可。在 page_translate()的实现中，先查看 TLB，如果命中，则不需要进行 page walk；如果未命中,则需要进行 page walk，并填充 TLB。另外不要忘记，更新 CR3 的时候需要强制冲刷 TLB 中的内容。由于 TLB 对程序来说是透明的，所以 kernel 不需要为 TLB 的功能添加新的代码。

从一到无穷大

80386 作为 IA-32 系列的鼻祖，实现了一款真正的 32 位 CPU 架构，并且提供了带有分页机制的保护模式，成为后续 x86 系列 CPU 发展的框架. 随着电子技术的发展，时钟频率越来越高，计算机的速度也越来越快；另一方面，集成电路的集成度越来越高，这意味着同样大小的芯片上可以容纳更多部件，实现更加复杂的控制逻辑. 于是各种各样的技术被开发出来，从多级流水线，cache，超标量，到乱序执行，SIMD，超线程（这些概念我们都会在计算机专业“计算机组成与体系结构”的课上接触到）... 每一个新名词的出现都能把计算机的速度往上推。仅仅就单核的主频而言，就已经从 80386 的 30MHz 提升到 Intel Core i7 的 3GHz；x86 的 SSE 技术也已经在 15 年里更新了 5 代... 同时在 25 年前被认为是天文数字的 4GB 也已经满足不了人类的需求了，于是 x86-64 横空出世，又一次解决了内存容量的瓶颈问题；随着频率的增加，工艺和散热问题逐渐显现出来，这意味着 3GHz 已经快要到达单核时钟频率的极限了，于是多核架构应运而生... 然而大数据时代的来临又给计算机技术的发展送出了一张挑战书：据 IBM 统计，在 2012 年，每天大约产生 2.5EB(1EB = 10^6TB)的数据... 这一切都表明，在技术和需求的相互作用下，计算机世界正在往更快，更好的方向高速发展。

PC 的足迹
PC 的足迹系列博文对 PC 发展史作了简要的介绍，上文提到的大部分术语都涵盖其中，感兴趣的同学可以在茶余饭后阅读这些内容。

另一方面，8086 虽然是 x86 系列的第一款产品，但却并不是计算机发展史的源头。在 8086 之前，有 8080，8008，甚至更早的 4040 和 4004... 以及第一台通用计算机 ENIAC。如果你愿意突破硬件的障壁，你还能继续往前追溯：冯诺依曼体系结构，图灵机，计算理论，数理逻辑，布尔代数...。

"一"究竟起源于何处？"无穷"又会把我们带到怎么样的世界？思考这些问题，你会发现 CS/CE/EE 的世界有太多值得探索的地方了。

温馨提示
PA3 到此结束。

任务自查表

序号	是否已完成
必做任务 1	
必做任务 2	
必做任务 3	
必做任务 4	
选做任务 1	
选做任务 2	
选做任务 3	

实验提交说明

提交地址

阶段性工程提交至自建 git 远程仓库

最终工程和实验报告提交智慧树平台

提交方式

1. 实验报告命名为“学号-PA3.pdf”，上传至智慧树平台；

2. 教师将通过脚本自动检查运行结果，因此除非实验手册特别说明，不要修改工程中的任何脚本，包括 Makefile、test.sh。

- 我们会清除中间结果，使用原来的编译选项重新编译(包括 -Wall 和 -Werror)，若编译不通过，本次实验你将得 0 分(编译错误是最容易排除的错误，我们有理由认为你没有认真对待实验)。

3. 学生务必使用 make submit 命令将整个工程打包，导出到本地机器，再上传到智慧树平台。

- make submit 命令会用你的学号来命名压缩包，然后修改压缩包的名称为“学号-PA3.zip”。此外，不要修改压缩包内工程根目录的命名。为了防止出现编码问题，压缩包中所有文件名都不要包含中文。

git 版本控制

请使用 git 管理你的项目，并合理地手动提交的 git 记录。请你不定期查看自己的 git log，检查是否与自己的开发过程相符。git log 是独立完成实验的最有力证据，完成了实验内容却缺少合理的 git log，会给抄袭判定提供最有力的证据。

如果出现 git head 损毁现象，说明你拷贝了他人的代码，也按抄袭处理。

实验报告内容

你必须在实验报告中描述以下内容：

- 实验进度。按照“任务自查表”格式在线制作表格并填写。缺少实验进度的描述，或者描述与实际情况不符，将被视为没有完成本次实验。

- 思考题

你可以自由选择报告的其它内容。你不必详细地描述实验过程，但我们鼓励你在报告中描述如下内容：

- 你遇到的问题和对这些问题的思考、解决办法
- 实验心得

如果你实在没有想法，你可以提交一份不包含任何想法的报告，我们不会强求。但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图，却没有相应的详细解释(让我们明显看出是凑字数的)

来让你的报告看起来十分丰富，编写和阅读这样的报告毫无任何意义，你也不会因此获得更多的分数，同时还可能带来扣分的可能。