COA2023-programming07

> 出神入化

1 实验要求

分别实现实模式、分段式、段页式三种内存的地址转换与数据加载功能。

1.1 地址转换

在MMU类中，实现三个地址转换的方法，将逻辑地址转换为线性地址再转换为物理地址。

```
private String toRealLinearAddr(String logicAddr)
private String toSegLinearAddr(String logicAddr)
private String toPagePhysicalAddr(String linearAddr)
```

1.2 数据加载

在Memory类中，实现三个数据加载方法。

```
public void real_load(String pAddr, int len)
public void seg_load(int segIndex)
public void page_load(int vPageNo)
```

1.3 融合cache与TLB

将cache与TLB融合到MMU中。

2 相关资料

2.1 实模式与保护模式

实模式和**保护模式**都是CPU的工作模式的一种。

实模式出现于早期8088CPU时期。当时由于CPU的性能有限，一共只有 20 位地址线（所以地址空间只有1MB），以及一些 16 位的寄存器。为了能够通过这些16 位的寄存器去构成 20 位的主存地址，通常需要用两个寄存器，第一个寄存器表示基址，第二个寄存器表示偏移量。那么两个 16 位的值如何组合成一个 20位的地址呢？实模式采用的方式是把基址先向左移 4 位，然后再与偏移量相加。即：

- 物理地址 = 基址 << 4 + 偏移量

现在，大部分通用计算机系统启动后总是先进入实模式，对系统进行初始化，然后才转入保护模式进行操作。在保护模式里，虚拟存储器的机制就可以运作起来，你们将会在接下来的操作系统课程中对这部分内容得到更深入的理解。

2.2 虚拟存储器

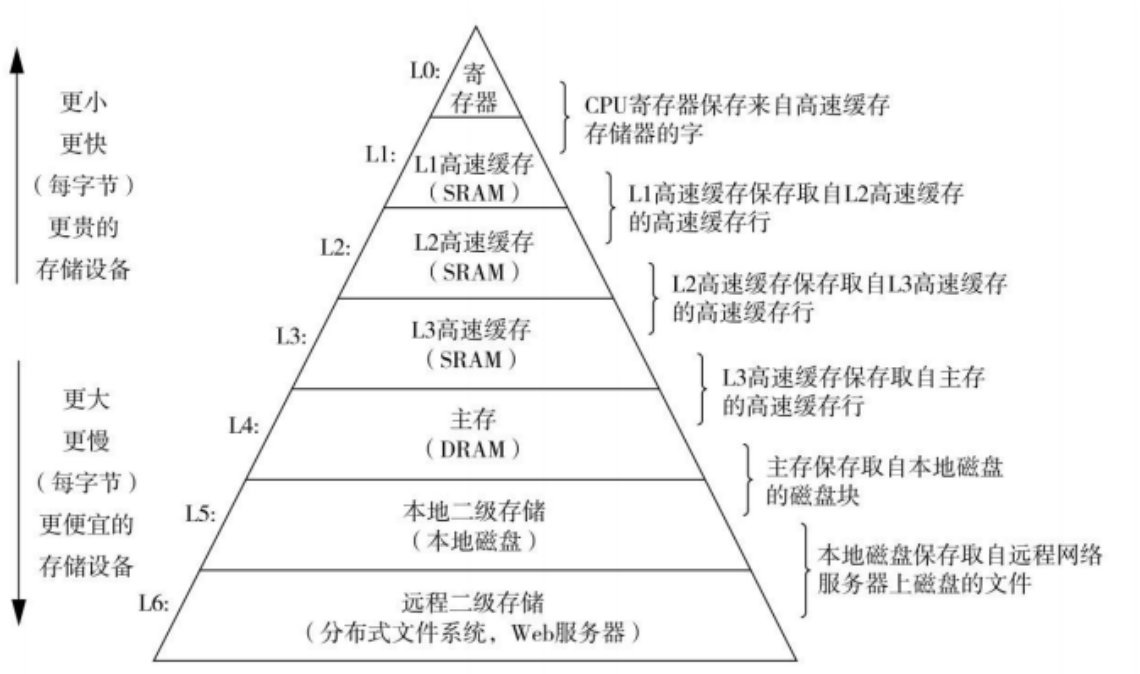
目前，在服务器、台式机和笔记本等各类通用计算机系统中都采用虚拟存储技术。在采用虚拟存储技术的计算机中，指令执行时，通过存储器管理部件（Memonr Mangemet Unit, 简称**MMU**）将指令中的逻辑地址转化为主存的物理地址。在地址转换过程中由硬件检查是否发生了访问信息不在主存、地址越界、访问越权等情况。

- 若发现信息不在主存，则由操作系统将数据从硬盘读到主存。
- 若发生地址越界或访问越权，则由操作系统进行相应的异常处理。

由此可以看出，虚拟存储技术既解决了编程空间受限的问题，又解决了多道程序共享主存带来的安全等问题。在虚拟存储系统中，CPU执行指令时所给出的是指令或操作数的虚拟地址，需要通过MMU将虚拟地址转换为主存的物理地址才能访问主存，MMU包含在CPU芯片中。

下图为存储器的层次结构。我们之前已经学习过，cache是主存的缓存。其实，在这个层次结构中，上层都可以看作下层的缓存，即主存可以看作磁盘的缓存。因此，要实现虚拟存储器，也必须考虑交换块大小、映射问题、替换问题、写一致性问题等。

根据对这些问题解决方法的不同，虚拟存储器分成三种不同类型：分页式、分段式和段页式。



下面以我们熟悉的IA-32架构（即x86架构）为例。IA-32 采用段页式虚拟存储管理方式，存储空间采用逻辑地址、线性地址和物理地址来进行描述。IA-32中的逻辑地址由 48 位组成，包含 16 位的段选择符和32 位的段内偏移量。为了便于多用户、多任务下的存储管理，IA-32 采用在分段基础上的分页机制。分段过程实现将逻辑地址转换为线性地址，分页过程再实现将线性地址转换为物理地址。

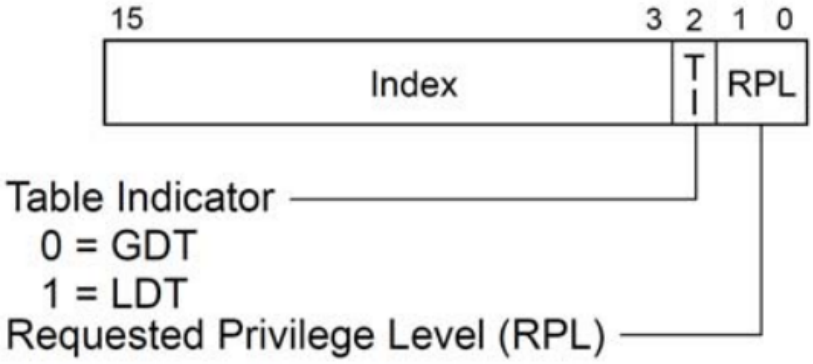
2.3 逻辑地址到线性地址的转换

为了说明逻辑地址到线性地址的转换过程，首先简要介绍段选择符、段描述符、段描述符表等基本概念。

2.3.1 段选择符和段寄存器

段选择符格式如下图所示，其中TI和RPL字段我们暂时不用关心。高 13 位的索引值用来确定当前使用的段描述符在描述符表中的位置，表示是其中的第几个段表项。

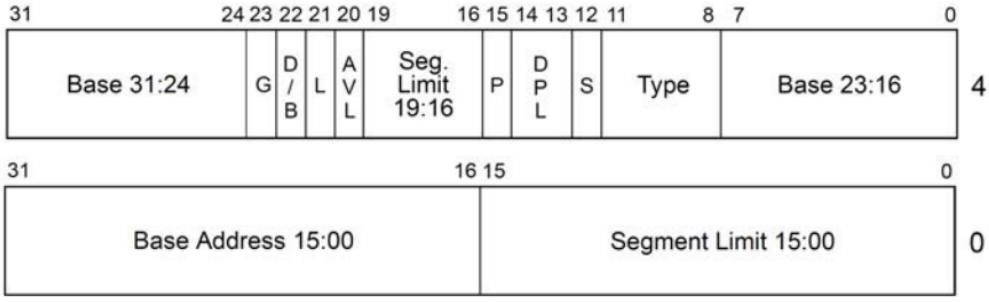
段选择符存放在段寄存器中，共有 6 个段寄存器: CS、SS、DS、ES、FS和GS。其中，CS是代码段寄存器，指向程序代码所在的段。SS是栈段寄存器，指向栈区所在的段。DS是数据段寄存器，指向程序的全局静态数据区所在的段。其他 3 个段寄存器可以指向任意的数据段。



2.3.2 段描述符

段描述符是一种数据结构，实际上就是分段方式下的段表项。一个段描述符占用 8 个字节，其一般格式如下图所示，包括 32 位的基地址（Base）、 20 位的限界（SegLimit）和一些属性位。属性位比较多，我们只介绍属性位G。

属性位G表示粒度大小。G=1说明段以页（4KB）为基本单位，G=0则段以字节为基本单位。由于界限为20 位，所以当G=0时，最大的段为1B * 2^20 = 1MB。当G=1时，最大的段为4KB * 2^20 = 4GB。



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

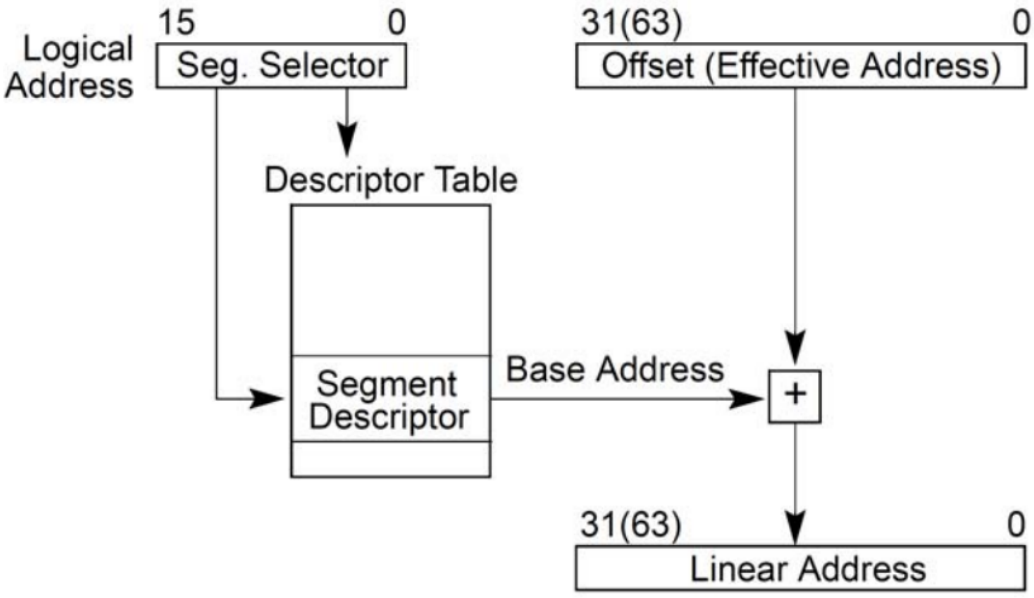
2.3.3 段描述符表

段描述符表实际上就是分段方式下的段表，由若干个段描述符组成，主要有三种类型：全局描述符表（GDT）、局部描述符表（LDT）和中断描述符表（IDT）。其中，GDT只有一个，用来存放系统内每个任务都可能访问的描述符，LDT和IDT我们暂时不用关心。

2.3.4 逻辑地址向线性地址的转换过程

逻辑地址向线性地址的转换过程如下图所示。 48 位的逻辑地址包含 16 位的段选择符和 32 位的段内偏移量。MMU首先通过段选择符内的 13 位索引值，从段描述符表中找到对应的段描述符，从中取出 32 位的基地址，与逻辑地址中 32 位的段内偏移量相加，就得到 32 位线性地址。

MMU在计算线性地址的过程中，可以根据段限界和段的存取权限（也称访问权限）判断是否地址越界或访问越权（访问越权是指对指定单元所进行的操作类型不符合存取权限，例如，对存取权限为“只读”的页面进行了写操作），以实现存储保护。



2.4 线性地址到物理地址的转换

IA-32采用段页式虚拟存储管理方式，通过分段方式完成逻辑地址到线性地址的转换后，再进一步通过分页方式将线性地址转换为物理地址。分页方式老师在上课时已经做出了详细的讲解，这里不再赘述。

3 实验攻略

3.1 实验概述

本次实验为虚拟存储器的模拟，需要大家完成实模式、分段式、段页式的模拟，然后将cache、TLB、memory、disk融合成一个完整的存储器系统。此外，我们还需要完成MMU的模拟。MMU是CPU的部件之一，CPU想要访问内存必须先经过MMU。

在本次作业中，大家主要的任务在两个类上面：MMU类和Memory类，它们分别是干什么的呢？

- MMU类：

1. 地址转换，将逻辑地址转换为线性地址再转化为物理地址，然后通过物理地址去访问Memory。
2. 权限检查，在地址转换的过程中检查权限，若越权访问则抛出异常
3. 通知加载，若在地址转换的过程中发现需要访问的信息不在主存，则需要通知主存加载数据。

- Memory类：

1. 数据加载，就是从磁盘中加载相应的数据进入内存。
2. 数据读取，就是通过MMU传入的物理地址去读取自己的数据并返回。

那么，逻辑地址、线性地址、物理地址分别是什么地址呢?

- 逻辑地址：指令中给出的地址， 48 位。CPU在运行指令时，如果想要访问内存，它并不是用直接使用内存的地址访问，而是给出一个由 16 位段寄存器和 32 位段内偏移量拼起来的 48 位的逻辑地址。

> 比如，如果CPU想知道当前指令的地址，他给出的逻辑地址应该是(CS:EIP)。其中，CS是 16 位代码段寄存器，EIP是 32 位指令指针寄存器（也就是程序计数器PC）。

- 线性地址：逻辑地址到物理地址的中间层， 32 位。如果没有启用分页机制，那么线性地址就等于物理地址。如果启用了分页机制，那么线性地址需要通过再一次变换才能得到物理地址。
- 物理地址：内存中的地址， 32 位。

MMU类的地址转换功能主要在addressTranslation方法上，这个方法是本次作业最重要也是最需要理解的一个方法，大家需要去认真阅读，接下来的所有讲解都将围绕addressTranslation方法展开。

接下来，我们将对源码进行一个全面的解读。

3.2 代码导读

3.2.1 代码结构

```
j.
| .gitignore
| pom.xml
| README.md
|
└─src
├─main
| └─java
| ├─cpu
| | ├─alu
| | |
| | ├─fpu
| | |
| | └─mmu
| |         MMU.java        # need to write
| |
| |
| ├─memory
| | |       Memory.java # need to write
| | |
| | ├─cache
| | |
| | └─disk
| |
| └─util
|
└─test
```

3.2.2 实模式的模拟

Memory

在Memory类里面，我们设置了两个字段如下：

```
public static boolean SEGMENT = false;
public static boolean PAGE = false;
```

这两个字段为开启分段分页的标志，当两个字段均为false时，代表此时系统运行在实模式下。

MMU

在MMU类中，addressTranslation方法对实模式的处理如下：

```
if (!Memory.SEGMENT) {
    // 实模式：线性地址等于物理地址
    linearAddr = toRealLinearAddr(logicAddr);
    memory.real_load(linearAddr, length); // 从磁盘中加载到内存
    physicalAddr = linearAddr;
}
```

这三行代码其实就是三个步骤。

- 第一步，将逻辑地址转换为线性地址；
- 第二步，从磁盘加载数据；
- 第三步，线性地址无需转换直接等于物理地址。

这样，MMU就完成了它在实模式下的使命，转化为物理地址之后就可以拿去访问主存了。

3.2.3 分段式的模拟

Memory

当Memory类的SEGMENT字段设为true，PAGE设为false时，系统就进入了分段式。根据2.3的内容，此时系统需要有全局描述符表和段描述符。我们模拟了一个简单的段描述符的结构如下：

```
private static class SegDescriptor {
    private char[] base = new char[32];  // 32位基地址
    private char[] limit = new char[20]; // 20位限长
    private boolean validBit = false;  // 有效位
    private boolean granularity = false;  // 粒度
}
```

其中，粒度为true表示段以页（4KB）为基本单位，为false表示段以字节为基本单位。

然后，我们模拟了全局描述符表GDT，并提供了访问它的get方法：

```
private SegDescriptor[] GDT = new SegDescriptor[8 * 1024];  // 全局描述符表
private SegDescriptor getSegDescriptor(int index) {
    if (GDT[index] == null) {
        GDT[index] = new SegDescriptor();
    }
    return GDT[index];
}
```

为了降低实验的难度，我们简化了分段机制。`我们规定，除了测试需要，每个由MMU装载进入GDT的段，其段基址均为全 0 ，其限长均为全 1 ，未开启分页时粒度为false，开启分页后粒度为true。` 这样简化之后，就不会存在多个段冲突的问题，也降低了出现段错误的概率。

MMU

在MMU类中，addressTranslation方法对分段式的处理如下：

```
if (!Memory.SEGMENT) {
    // 实模式
    ...
} else {
    // 分段模式
    int segIndex = getSegIndex(logicAddr);
    if (!memory.isValidSegDes(segIndex)) {
        // 缺段中断，该段不在内存中，内存从磁盘加载该段索引的数据
        memory.seg_load(segIndex);
    }
    linearAddr = toSegLinearAddr(logicAddr);
    // 权限检查
    int start = Integer.parseInt(transformer.binaryToInt(linearAddr));
    int base = chars2int(memory.getBaseOfSegDes(segIndex));
    int limit = chars2int(memory.getLimitOfSegDes(segIndex));
    if ((start < base) || (start + length > base + limit)) {
        throw new SecurityException("Segmentation Fault");
    }
    if (!Memory.PAGE) {
        // 分段模式：线性地址等于物理地址
        physicalAddr = linearAddr;
    }
}
```

这段代码其实可以分成四个步骤。

- 第一步，查GDT，判断段是否在内存，若不在内存则加载段；
- 第二步，将逻辑地址转换为线性地址；
- 第三步，检查是否访问越界，若越界则抛出段错误；
- 第四步，线性地址直接等于物理地址。

这样，MMU就完成了它在分段式下的使命。

3.2.4 段页式的模拟

Memory

当Memory类的SEGMENT字段和PAGE字段均设为true时，系统就进入了段页式。我们模拟了页表与页表项如下：

```
private final PageItem[] pageTbl = new PageItem[Disk.DISK_SIZE_B /
Memory.PAGE_SIZE_B];
private static class PageItem {
    private char[] pageFrame; // 物理页框号
    private boolean isInMem = false;  // 装入位
}
private PageItem getPageItem(int index) {
    if (pageTbl[index] == null) {
        pageTbl[index] = new PageItem();
    }
    return pageTbl[index];
}
```

在开启分页后，有页表的存在貌似已经足够。但这会导致一个问题，在加载页的时候，我们需要知道新的页应该加载到哪个物理页框上去，如果仅有页表的存在，我们就需要遍历整个页表来找出空闲的物理页框号，这个过程比较复杂。而且，在现代操作系统中，每个进程都对应着一张页表，而系统中往往运行着非常多的进程，把每个进程的页表都遍历一遍显然不太现实。

因此，一些计算机系统采用了反向页表来来解决这个问题。为了降低实验难度，我们不模拟反向页表，而是简单地使用了一个有效位数组：

```
private boolean[] pageValid = new boolean[MEM_SIZE_B / PAGE_SIZE_B];
```

这个数组记录了每个物理页框的使用情况，大家可以选择使用这个数组，也可以选择使用遍历页表的方式（毕竟我们只有一个页表）。

MMU

在MMU类中，addressTranslation方法对段页式的处理如下：

```
if (!Memory.SEGMENT) {
    // 实模式
    ...
} else {
    // 分段模式
    ...
    if (!Memory.PAGE) {
    ...
    } else {
        // 段页式
        int offset =
Integer.parseInt(transformer.binaryToInt(linearAddr.substring(20, 32)));
        int pages = (length - offset + Memory.PAGE_SIZE_B - 1) /
Memory.PAGE_SIZE_B;
        if (offset > 0) pages++;
        int endvPageNo = startvPageNo + pages - 1;
        for (int i = startvPageNo; i <= endvPageNo; i++) {
            if (!memory.isValidPage(i)) {
                // 缺页中断，该页不在内存中，内存从磁盘加载该页的数据
```

```
                memory.page_load(i);
            }
        }
        physicalAddr = toPagePhysicalAddr(linearAddr);
    }
}
```

这段代码在分段模式已经将逻辑地址转换为线性地址的基础之上，又做了两件事情：

- 第一步，计算出需要访问的虚拟页号（有很多行代码是在处理数据跨页的情况，可以忽略），查页表判断该虚页是否在内存，若不在内存则加载页；
- 第二步，将线性地址转换为物理地址。

这样，MMU就完成了它在段页式下的使命。

3.2.5 TLB的模拟

我们模拟了一个大小为 256 行、使用全相联映射、FIFO替换策略的TLB。TLB类的源码比较简单，相关注释也非常详细，大家可以自行阅读。

需要特别提醒的是，TLB只是页表的缓存，因此TLB行号并不等于虚页号。

在你已经结合源代码充分理解上述内容后，接下来就可以开始快乐地编码啦！

3.3 实现指导

3.3.1 实模式的实现

实模式的加载与地址转换十分简单，在此略去。需要注意的是，在2.1的计算公式中，实模式应该是使用 32 位地址的，而MMU使用 48 位逻辑地址仅仅是为了与保护模式兼容。在实际计算中，我们把高 16 位看作基址，低 32 位看作偏移量（实际有用的只有低 16 位）。

3.3.2 分段式的实现

首先，你需要在Memory类中实现seg_load段加载方法。seg_load方法需要干两件事情：

- 从磁盘上加载该段的数据到内存。如何从磁盘上读取一整段呢？你应该使用段基址作为访问磁盘的地址，用段限长（即段大小）作为读取的长度。至于段基址和段限长是多少，参考我们3.2.3的规定。那么加载过来之后写到内存的哪里呢？由于分段式下每个段大小只有1MB，不会超出内存大小，所以我们默认把数据放在物理地址为 0 的地方。
- 除了加载数据，你还需要填好全局描述符表GDT，需要填入的内容还是按照3.2.3的规定进行填写。下面为该规定的原文。

> 我们规定，除了测试需要，每个由MMU装载进入GDT的段，其段基址均为全 0 ，其限长均为全 1 ，未开启分页时粒度为false，开启分页后粒度为true。

然后，你需要实现在MMU类中实现toSegLinearAddr段级地址转换方法。

- 在分段式下，逻辑地址转线性地址应该要查全局描述符表GDT，按照2.3.4的流程进行计算。
- 注意，不要以为可以偷懒直接把逻辑地址的前 16 位去掉（反正段基址都全为 0 了），我们在测试的时候会强制创建一些段基址不为 0 的段来测试大家是否正确实现了查全局描述符表这一操作。

提醒一下大家，访问GDT尽可能使用getSegDescriptor方法，而不要直接访问GDT数组。在getSegDescriptor方法中，我们已经处理好了空指针的情况，而如果直接访问GDT数组，可能会出现空指针，是不安全的。

> 有同学可能会疑惑，既然段基址都为 0 ，那分段和查段表还有什么意义呢?
>
> 正如2.2所述，系统在地址转换过程中会检查是否发生了地址越界、访问越权等情况。分段其实最主要的功能就是提供存储保护，不同段之间不能越权互相访问数据，否则会发生段错误，也就是熟悉的"Segmentation Fault"。我们的MMU在地址转换过程中也会检查是否发生地址越界的情况，若越界则抛出段错误。
>
> Linux操作系统为了使它能够移植到绝大多数流行的处理器平台，就是把所有段基址设为全 0 的、段限长设为全 1 的。但即使Linux操作系统这么做了，在进行地址转换的时候，它也是需要去查GDT的。可以说，查GDT是所有系统的必备步骤。因此，我们即使规定了段基址为全 0 ，我们也需要考察大家是否正确进行了查表操作。

3.3.3 段页式的实现

首先，你需要在Memory类中实现page_load页加载方法。page_load方法也是需要干两件事情：

- 从磁盘上加载该页数据到内存。如何在磁盘上读取该页数据呢？你应该使用该虚页的起始地址作为作为访问磁盘的地址，用页大小作为读取的长度。至于该虚页的起始地址是多少，可以直接根据虚页号得到。那么加载过来之后写到内存的哪里呢？这就需要你找出一个空闲的物理页框然后放下去啦。
- 除了加载数据，你还需要填好页表，如果你使用有效位数组的话还需要填好有效位数组。

然后，你需要修改seg_load方法。

- 在段页式存储管理下，从磁盘加载数据应该是以页为单位的，不再是以段为单位。因为开启分页之后，一个段应该是4GB（怎么算出来？），加载这么多数据内存可吃不消。因此开启分页之后，seg_load应该跳过加载数据这一步，它的作用在开启分页之后仅仅是填写GDT，加载数据的任务应该交给page_load来完成。
- 强烈建议搞清楚addressTransition方法从头到尾在干什么，这会帮助你理解seg_load和page_load的关系。

最后，你需要实现在MMU类中实现toPagePhysicalAddr页级地址转换方法。

- 在段页式下，线性地址转物理地址需要查页表，然后进行虚拟页号到物理页号的替换，具体流程可以参考课件。

与GDT同理，访问页表的时候尽可能使用getPageItem方法。

3.3.4 cache与TLB的融合

第一步，你需要将cache融合进MMU中。

这一步相对简单。需要注意，由于cache是memory的缓存，所以任何涉及到访问主存数据的地方都要添加对cache的调用。

第二步，你需要将TLB融合进MMU中。

这一步相对麻烦，即使TLB中的方法我们已经帮助大家实现好了。由于TLB是页表的缓存，所以任何涉及到访问页表的地方都要添加对TLB的调用。下面作一点提示。

在addressTranslation方法中判断是否缺页并进行加载页的时候，原代码如下：

```
if (!memory.isValidPage(i)) {
    // 缺页中断，该页不在内存中，内存从磁盘加载该页的数据
    memory.page_load(i);
}
```

在启用TLB之后，需要改动的地方有：

1. 启用TLB之后，判断是否缺页的工作应该首先交给TLB来完成。
2. 如果发生缺页，page_load方法会进行填页表，填页表之后不要忘记填TLB。

同时，在之前你自己实现的toPagePhysicalAddr方法里面，从页表中取物理页框号的时候，也应该改成从TLB中取物理页框号，这个时候也需要调用你自己设计的TLB的方法。参考结构如下

```
if (TLB.isAvailable) {
访问TLB取物理页框号;
} else {
访问页表取物理页框号;
}
```

至此，你已经完成了全部工作(・ω・)ノ

3.3.5 关于磁盘的一些补充

分页机制的基本思想其实就是，为每个进程都提供一个独立的、极大的虚拟地址空间，将主存里放不下的页放到磁盘上去。在真实的计算机磁盘中，会有专门的地方来存放虚页：在Windows下是PageFile.Sys文件，在Linux下是swap分区。

而我们的模拟磁盘简化了这个设计，因为在我们这个系统中不会出现多个进程。所以我们可以简单地用磁盘文件的前 4 个GB来存放虚页。也就是说，我们将磁盘文件的前 4 个GB（为什么是4GB？因为线性地址是 32 位的，实际上我们的磁盘文件只有64MB）直接作为进程的虚拟地址空间。

因此，无论是在实模式还是分段式还是段页式，当你访问磁盘的时候，请保证你用来访问磁盘的是线性地址。上面所说的用来加载段的时候用的段基址、加载页的时候用的虚页起始地址，都属于线性地址。

3.4 彩蛋

作为存储模块的最终章，我们设计了一个彩蛋供大家体会各个内存部件之间的关系。

细心观察的同学可以发现，我们在Memory类中新增了一个timer字段，在这个字段开启设为true后，所有访问内存以及访问页表的操作都将产生 10 毫秒的延时，而访问cache与TLB的操作则不会产生延时。由于在真实的计算机系统中，访问cache和TLB的速度要比访问主存的速度快得多。因此，我们设计了这个10 毫秒的延时，来模拟这个速度上的差异。

在我们提供的测试用例中有一个EasterEgg类，这个类将不断的访问同一段数据，来模拟计算机的时间局部性原理。在EasterEgg类中的init方法里面有如下代码：

```
Memory.timer = true;
Cache.isAvailable = true;
TLB.isAvailable = true;
```

大家可以修改两个isAvailable字段，自主设置cache和TLB是否启用，来观察他们速度上的差异。以下是提供的运行结果，仅供参考，此处的处理器型号是Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.GHz。

大家可以尽情修改彩蛋，比如修改延时的时间、修改cache和tlb是否启用等。也可以在彩蛋里面单步进入，看看从mmu开始是怎么一步步访问disk、memory、TLB和cache的。

最后，留给大家一个问题，为什么在我们的模拟下，只开启cache和只开启TLB的运行时间差那么远？

(・ω・)ノ