
PA 2017 实验指导

本教程是针对 2017 年秋季《计算机系统基础》Programming Assignment (PA) 实验的指导。本教程所使用的框架代码可以通过以下方式获取。

`git clone https://github.com/ics-pa/pa2017.git`

拿到框架代码后, 1) 修改 `Makefile.git` 中的 `STU_ID` 填入学号; 2) 在项目根目录执行 `make clean`。在提交时, 使用 `make submit` 命令打包项目代码并上传压缩包 (详细内容请参见 PA 0)。

- 本实验教程是 PA 课堂教学的配套教程, 其中理论部分主要依赖《计算机系统基础》理论课的教授。整个 PA 实验由四个主要部分组成, 分别是:

PA 1 - 数据的表示、存取和运算

PA 2 - 程序的执行

PA 3 - 存储管理

PA 4 - 异常、中断与 I/O

这四个阶段大致对应模拟器中对 CPU、MMU 和 Device 的模拟, 涵盖计算机系统基础课程中的主要内容, 并尽力做到和理论课程的讲授进度的同步。

- 除 PA 0 和可选任务以外, 每一阶段的 PA 在教程中都会以如下方式展开:

1. 路线图: 指明当前完成的阶段的进度;
2. 预备知识: 简要讲述和回顾这一阶段实验所对应的理论知识, 如果还不清楚的, 请结合理论课内容进行复习;
3. 代码导读和实验理解: 详细讲述实验所涉及的代码, 并对实验中所需要完成的内容进行讲解和提示;
4. 实验过程及要求: 罗列实验步骤和要求, 在每一阶段开始做之前, 可以先看看这里的要求。

- 教程所涉及的主要参考资料

1. 课本

2. i386 手册

➤ <http://css.csail.mit.edu/6.858/2014/readings/i386.pdf>

3. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

➤ <https://software.intel.com/en-us/articles/intel-sdm>

4. x86 Instruction Set Reference (i386 手册上没有的指令来这里找)

➤ <http://www.felixcloutier.com/x86/>

Don't panic :-)

目录

PA 2017 实验指导	1
目录	3
PA 0 实验环境配置	4
PA 1 数据的表示、存取和运算	8
PA 1-1 数据在计算机内的存储	9
PA 1-2 整数的表示、存储和运算	13
PA 1-3 浮点数的表示和运算	17
PA 2 程序的执行	24
PA 2-1 指令解码与执行	25
PA 2-2 装载程序的 loader	43
PA 2-3 可选任务：完善调试器	48
PA 3 存储管理	58
PA 3-1 Cache 的模拟	59
PA 3-2 保护模式	63
PA 3-3 虚拟地址转换	71
PA 4 异常、中断与 I/O	76
PA 4-1 异常和中断的响应	77
PA 4-2 外设与 I/O	83
PA 4-3 可选任务：游戏移植	89
I386 手册勘误	94
Git 入门教程	98

PA 0 实验环境配置

在正式开始 PA 实验之前，需要完成实验环境的搭建并下载框架代码。本节首先介绍实验环境的搭建方法。PA 实验使用 GNU/Linux 平台开展实验，项目代码使用 C 语言编写并用 gcc 编译。同时依赖于一些开发库如 readline 和 SDL。为了避免在开发过程中遇到太多麻烦，建议使用本教程所推荐的配置展开 PA 实验。

总体而言，本教程目前所提供的实验和框架代码的原始开发环境为：

类型	名称	版本
虚拟机	VMware® Workstation 12 Player	12.5.6 build-5528349
客户操作系统	Debian GNU/Linux 9 (i386)	9.0
桌面环境	MATE	1.16.2
编译器	gcc	6.3.0
readline 开发包	libreadline-dev	7.0-3
SDL 开发包	libsdl1.2-dev	1.2.15+dfsg1-4

安装虚拟机

如果你当前正使用的宿主操作系统(Host OS)不是恰好为 32 位的 GNU/Linux 系统(绝大多数情况下不是)，那么推荐安装虚拟机来搭建实验环境。本教程推荐使用 **VMware Player** 作为虚拟机平台，也可以使用 **VirtualBox**。

VMware Player 下载地址：

<https://www.vmware.com/products/player/playerpro-evaluation.html>

在安装过程中，忽略序列号的输入环节，使用邮箱注册后，就可以免费使用。

或，VirtualBox 下载地址：

<https://www.virtualbox.org/wiki/Downloads>

在宿主系统中成功安装虚拟机软件后，新建虚拟机并安装客户操作系统（Guest OS）。本实验所使用的客户操作系统为 GNU/Linux，推荐使用 Debian 9 操作系统。下载地址为

<https://www.debian.org/distrib/netinst>

选择 Small CDs or USB sticks 项下的 i386 体系结构镜像，按照说明进行安装，语言选择为 US English。为了将来运行的方便，在安装过程中，推荐安装桌面环境。框架代码的测试环境为 MATE 桌面系统，这是老的 Gnome 2 的一个分支，比较符合一些老用户的使用习惯，你也可以尝试使用其他的桌面环境（比如 xfce）。以 MATE 为例，在图形化的桌面环境中，命令行终端可以通过 **Applications -> System Tools -> MATE Terminal** 打开。

配置客户操作系统

在操作系统安装完毕后，需要进行的第一个操作是将当前用户加入到 `sudoer list`。所需要进行的具体操作步骤为：

```
username@debian:~$ su
Password: <input your password>
root@debian:/home/username# adduser username sudo
root@debian:/home/username# exit
```

完成上述操作后，Log Out 系统并再次登入，就可以在普通用户的身份下使用 `sudo cmd` 的方式执行特权命令了。

在此提醒大家注意：千万不要使用 `root` 用户身份来进行开发和日常的使用，应当使用普通用户身份进行工作，遇到需要执行特权命令时，使用 `sudo`。

如果你发现系统中没有 `sudo` 命令，那么可以在使用 `su` 切换到 `root` 用户后，使用命令

```
root@debian: apt-get install sudo
```

来安装 `sudo`。

安装完操作系统后，推荐继续安装 VMware 提供的 `vmware-tools`。`vmware-tools` 针对客户操作系统提供了一些很方便的功能，如自适应调整分辨率、共享文件夹等。在安装过程中 VMware Player 就会提示下载安装。`vmware-tools` 镜像下载完成后，在 VMware Player 的菜单中选择 `install vmware tools...`，就可以看到在客户操作系统中挂载了一个光盘驱动器，其中就包含 `vmware-tools` 的安装文件。

在安装之前，需要额外安装编译环境和 Linux 头文件包通过执行以下命令来安装相应的依赖

```
sudo apt-get install linux-headers-4.9.0-3-all-i386 net-tools

sudo apt-get install build-essential
```

安装好必要的依赖后，将 `vmware-tools` 光盘中的 `VMwareTools-10.1.6-5214329.tar.gz` 压缩包拷贝出来并解压，在解压得到的目录下执行命令

```
sudo ./vmware-install.pl
```

在第一步可能会提示是否安装 `open-vm-tools`，请忽略，并坚持安装 `vmware-tools`。如果没有遇到其他问题，一路选择默认选项即可。在完成安装之后，你将获得包括屏幕分辨率自动调整、与宿主系统共享剪贴板、共享文件夹等功能，有助于更方便地使用客户操作系统的功能。

你可以在宿主系统中创建一个文件夹，并在 `vmware` 软件菜单中通过 `Virtual Machine Settings -> Options -> Shared Folders` 页面指定创建的文件夹为共享文件夹，在客户系统中共享文件夹位于 `/mnt/hgfs/` 目录下。有了共享文件夹，就可以方便地在虚拟机和宿主机之间传递文件了。

搭建开发环境

正如在本节开头表格中所列举的那样，实验的开展需要安装编译环境并安装一些工具和开发包。简单来说，可以通过下面这行命令来安装所需软件（如果你选择安装 `vmware-tools` 并成功，那么 `build-essential` 应该

已经装好了)：

```
sudo apt-get install build-essential libreadline-dev libsdl1.2-dev vim git
```

这一行命令会帮助我们安装包括 `gcc`、`make` 在内的编译环境，安装 `readline` 和 `SDL` 开发包（注意 `SDL` 的版本，我们使用老的 `1.x` 版本而不是新的 `2.x` 版本），并安装必要的工具如 `vim` 和 `git`。

对 `vim` 不熟悉的同学可以参照其官网¹提供的内容，鸟哥的 Linux 私房菜²也是一个不错的初学者入口。当然善用万能的 Google 和百度能够解决你面临的几乎所有困难。

PA 使用 `git` 来管理项目代码，在本教程最后有一个入门教程。注意在提交时我们只接受在 `master` 分支中的代码。

获取并理解框架代码

在成功完成搭建实验环境后，下一步就是获取框架代码并着手准备开展实验了。在 Terminal 中执行以下命令获取 PA 框架代码：

```
git clone https://github.com/ics-pa/pa2017.git
```

- 拿到框架代码后，1) 修改 `Makefile.git` 中的 `STU_ID` 填入学号；2) 在项目根目录执行 `make clean`。

下面我们来对框架代码进行整体理解。整个框架代码包含三大块内容：`nemu`、`testcase`、`kernel`，另外还有一个包含了库文件和配置文件的 `include` 文件夹。其中，`nemu` 就是我们要重点实现的模拟器相关的代码；`testcase` 中包含了一系列以用户程序的形式呈现的测试用例；而 `kernel` 则是一个简单的操作系统，用于配合之后的相关功能实现。在 PA1 的阶段，我们仅需要关注 `nemu` 中的项目代码和测试用例；在 PA2 初期我们引入 `testcase`，并在 PA2 的后期引入 `kernel`。PA 整体的框架代码结构树节选如下图所示（更完整的树形结构可以通过 `tree` 命令查看，你或许需要通过 `apt-get` 来安装 `tree` 命令）：

```
pa2017/
├── game                // 包含游戏相关代码
├── include             // PA 整体依赖的一些文件
│   ├── config.h       // 一些配置用的宏
│   └── newlib          // 公共用的库
├── kernel              // 一个微型操作系统内核
├── Makefile            // 帮助编译和执行工程的 Makefile
├── Makefile.git        // 和 git 有关的部分
├── nemu                // NEMU
│   └── src
│       └── main.c      // NEMU 入口
└── testcase            // 测试用例
```

实验任务

- 你需要完成教程中的所有任务，除明确注明是可选任务的额外加分项以外，其余都是必做任务；

¹ <http://www.vim.org/>

² http://cn.linux.vbird.org/linux_basic/0310vi.php

-
- 我们会设置 4 个打分点，分别是 PA 1 到 PA 4 的最终阶段截止之时；在此之前的每个小阶段，我们需要你按时提交一个可编译的版本，如果在小阶段的截止时间之前没有提交可编译版本，会扣 10% 的分数。但在最终打分时间点之前，我们不检查逻辑正确性；
 - 在提交时，使用 `make submit` 命令打包项目代码并上传压缩包；
 - 当然，不允许抄袭！我们会做交叉检验。一经查实，该阶段得分清零。

PA 1 数据的表示、存取和运算

计算机的世界是一个由 0（低电平）和 1（高电平）构成的世界，所有的信息都被表示成 01 串形式的数据。我们把世界上所有需要表示的信息大概分一分，发现可以分为两大类：非数值型和数值型。

非数值型大体上对应的是用于表征类别或个体的信息，比如汉字、英文字母、阿拉伯数字、每个学生个体等等，我们可以将每一个需要表示的对象赋予一个编号，并且让大家都遵守统一的编号规则就行了。比如 ASCII 编码，UTF-8 编码等等。由于编码规则没有太多道理可言³，大家认可就行，因此我们在此不给予太多关注。当然，在 PA 2 中我们其实会非常隐晦地重点关注这一类数据，那就是指令。选择了一套指令集体系结构（Instruction Set Architecture, ISA）就意味着我们接受了一套指令的编码规范。在本课程中，我们选择了 IA-32 这套 ISA。

与非数值型相对应的是数值型的信息，用于进行科学计算。纵观我们需要进行的科学计算，无非就是两种类型的数：整数和实数（复数总可以用两个实数来表示）。而在整数和实数上又定义了加减乘除等各种运算规则。计算机顾名思义就是要进行计算的，因此我们要重点关注的就是以整数和实数为代表的数值型信息的表示、存取和运算。

在 PA 1 中，配合理论课的节奏，我们先通过数据的存储开始简单了解 NEMU 的总体结构，进而从整数和浮点数的表示、存取和运算出发，开始构建一台完整计算机的旅程。

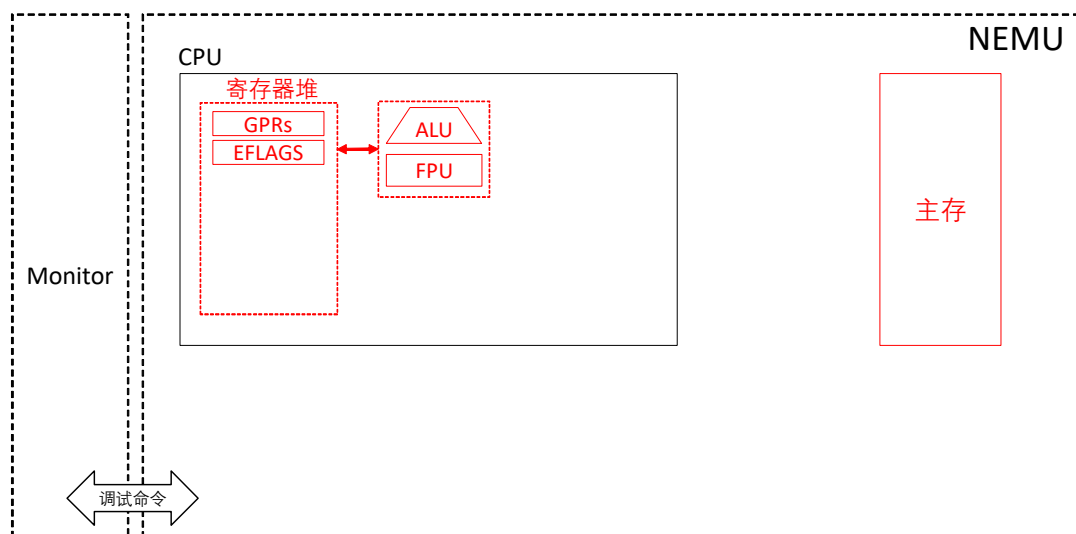


图 1-1 PA 1 路线图

³ 我们不把话说太绝对，非数值型的编码一般还是有那么一点道理的，比如 B 在字母表上在 A 的后面一个，那么 B 的编码就比 A 的编码大 1。

PA 1-1 数据在计算机内的存储

在构建一台能计算的机器之前，我们首先要尝试将计算的对象——数据妥善地存放在计算机中。

§1-1.1 预备知识

§1-1.1.1 数据存储和读写

在计算机内部，数据的存储往往采用层次化的存储器体系结构。在 NEMU 中，我们也根据这种层次化的方法，将存储器体系结构划分为寄存器、高速缓存（cache）、主存储器（RAM）、辅助存储器（硬盘）这四个层次，这四个层次的关系基本是越来越慢，越来越便宜（存储量自然越来越大），离 CPU 核心越来越远。在当前的实验中，为了使得 NEMU 具备最基本的运行能力，我们需要构建其中的两个层次，即，寄存器和主存储器（在本教程术语中，主存等于内存）。由于我们选择了 IA-32 指令集体系结构，那么我们的数据存储器件和规范就遵照对应的 x86 体系结构（在本教程术语中 x86 等于 i386）。

数据存储的最小单位是比特，即可以存储 1 位 0 或 1。8 个比特构成了内存中可编址的最小单位，字节（byte）。2 个字节 16 比特构成一个字（word）。4 个字节构成一个双字（double word）。

§1-1.1.2 寄存器

寄存器是位于 CPU 内部的存储器，可以由 CPU 直接访问。为了实现最基本的数据存储以支持马上要展开的运算和指令执行的功能，我们需要模拟 8 个通用寄存器（General Purpose Register, GPR）和程序计数器（PC）。我们模拟的是 32 位机器，因此这八个通用寄存器均为 32 位，依次为：

eax, ecx, edx, ebx, esp, ebp, esi, edi

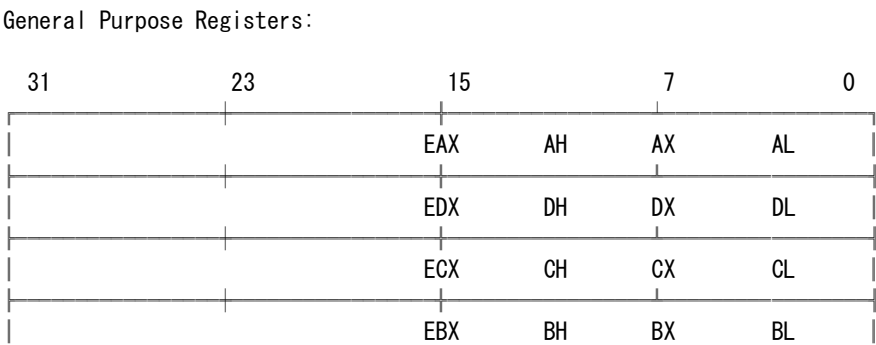
为了实现向下兼容，这八个通用寄存器的低 16 位分别对应于

ax, cx, dx, bx, sp, bp, si, di

这 8 个通用寄存器。同时，ax, cx, dx, bx 这四个通用寄存器的高 8 位和低 8 位又分别对应于：

ah, al, ch, cl, dh, dl, bh, bl

这 8 个通用寄存器。可见，x86 体系结构的通用寄存器结构可以用下图来表示（摘录自 i386 手册 Figure 2-5）：



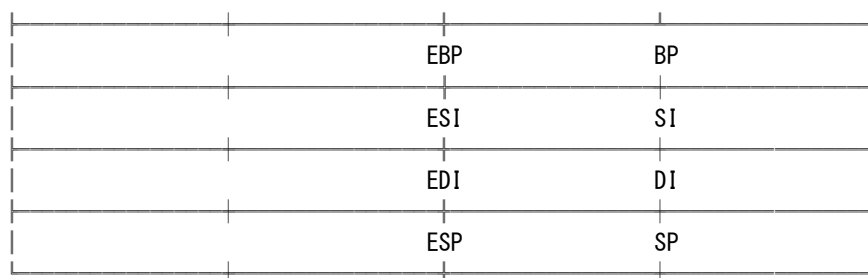


图 1-2 通用寄存器结构图

程序计数器也是 32 位，在 NEMU 的模拟 CPU 中，由 eip 寄存器来表示。

§1-1.1.3 主存

主存从器件来看就是插在机器主板上的内存条，通过存储器总线和 CPU 相连。以目前常用的 DRAM 芯片技术来说，主存的内部构成也有一点小复杂（参见课本第 233 页 6.2 节的内容）。但是，在本教程中，我们无需关心其复杂的内部结构，而只关注它对外所展现出来的存取接口。

我们只需要理解主存就是一个由字节构成的数组，数组的下标就是内存地址，而数组中某下标对应的元素就是主存中该地址对应的字节数据。这么介绍虽然有些本末倒置的意味，但是鉴于大家先学了 C 语言编程，这么类比应该能够比较快速地理解主存对外所展现出来的形象。

x86 采用小端方式存储数据，它规定了超过一个字节的数据的存储规则：低有效字节放在低地址，高有效字节放在高地址。

§ 1-1.2 代码导读和实验理解

§1-1.2.1 寄存器模拟

NEMU 中模拟寄存器文件的代码位于 `nemu/include/cpu/reg.h` 头文件中，对应于 `CPU_STATE` 这个结构体。

在源文件 `nemu/src/cpu/cpu.c` 中定义了全局变量 `CPU_STATUS cpu`，这样，在程序的其它模块中，我们就可以通过类似 `x = cpu.eax` 这样的语句，来访问寄存器的值了。

但是，目前框架代码中的寄存器结构体实现并不完善。我们希望以 `cpu.eax` 形式访问的和以 `cpu.gpr[0]_32` 形式访问的是同一个模拟寄存器，同时 `cpu.gpr[0]_16` 能够访问到 `cpu.eax` 的低 16 位，`cpu.gpr[0]_8[0]` 访问到 `cpu.eax` 的低 8 位等等。在 `nemu/src/main.c` 中的 `main()` 函数里，我们一开始就执行了 `reg_test()` 测试用例。你需要调整寄存器结构体的实现方式，使其满足上述要求，并通过 `reg_test()` 测试用例。

作为实验的开篇，我们在此给出一个参考答案：

```
union {
    union {
        union {
            uint32_t _32;
            uint16_t _16;
            uint8_t _8[2];
        };
        uint32_t val;
    };
};
```

```

    } gpr[8];
    struct { // do not change the order of the registers
        uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
    };
};

```

§1-1.2.2 主存模拟

在头文件 `include/memory/memory.h` 中，我们定义宏：

```
#define MEM_SIZE_B 128*1024*1024
```

约定 NEMU 拥有 128MB 字节的内存。紧接着，在源文件 `src/memory/memory.c` 中，我们定义数组：

```
uint8_t hw_mem[MEM_SIZE_B];
```

如此一来，我们的模拟主存就构建完成了。但是仅有主存物理器件的模拟尚且不够，我们还需要为它创建读写接口。创建读写接口的目的是为了对模块提供的功能进行封装，以免其它外部模块直接接触模块的内部核心数据，产生不可预料的后果。在头文件 `include/memory/memory.h` 中，我们声明了六个读写函数：

```

// 物理地址读写
uint32_t paddr_read(paddr_t addr, size_t len);
void paddr_write(paddr_t addr, size_t len, uint32_t data);

// 线性地址读写
uint32_t laddr_read(laddr_t addr, size_t len);
void laddr_write(laddr_t addr, size_t len, uint32_t data);

// 虚拟地址读写
uint32_t vaddr_read(vaddr_t addr, uint8_t sreg, size_t len);
void vaddr_write(vaddr_t addr, uint8_t sreg, size_t len, uint32_t data);

```

——原教程和框架代码对 `vaddr_read()` 和 `vaddr_write()` 的定义不一致，此处已更正，感谢 16 级许翔同学

这六个函数的第一个 `xaddr_t` 类型的参数都是待读写的内存地址，第二个 `size_t` 类型的参数 (`len`) 都是待读写的数据的长度（字节数），对于虚拟地址读写来说，仅可以是 1、2、4 中的某一个值。对于所有的 `read` 函数，其返回的 `uint32_t` 类型数据即为模拟内存中从 `addr` 开始连续 `len` 个字节的数据内容，若 `len` 小于 4，在返回值高位补 0。对于所有的 `write` 函数，其第三个 `uint32_t` 类型的参数 (`data`) 即为要写入的数据，写入位置为模拟内存中从 `addr` 开始连续 `len` 个字节，若 `len` 小于 4，则将 `data` 高位截断。注意 NEMU 采用小端方式存储数据。

在介绍存储管理之前，NEMU 工作在类似实地址模式下。此时，虚拟地址等于线性地址等于物理地址，因此，在源文件 `src/memory/memory.c` 中，虚拟地址和线性地址的读写函数简单实现为：

```

uint32_t paddr_read(paddr_t paddr, size_t len) {
    return hw_mem_read(paddr, len);
}
void paddr_write(paddr_t paddr, size_t len, uint32_t data) {

```

```

hw_mem_write(paddr, len, data);
}

uint32_t laddr_read(laddr_t laddr, size_t len) {
    return paddr_read(laddr, len);
}

void laddr_write(laddr_t laddr, size_t len, uint32_t data) {
    paddr_write(laddr, len, data);
}

uint32_t vaddr_read(vaddr_t vaddr, uint8_t sreg, size_t len) {
    assert(len == 1 || len == 2 || len == 4);
    return laddr_read(vaddr, len);
}

void vaddr_write(vaddr_t vaddr, uint8_t sreg, size_t len, uint32_t data) {
    assert(len == 1 || len == 2 || len == 4);
    laddr_write(vaddr, len, data);
}

```

随着实验进展到 PA3，我们会在这里进行大量的实验。在目前阶段，我们对于内存读写的接口无需进行修改。

§1-1.3 实验过程及要求

1. 修改 `CPU_STATUS` 结构体中的通用寄存器结构体；
2. 使用 `make` 编译项目；
3. 在项目根目录通过 `./nemu/nemu` 命令执行 `nemu` 并通过 `reg_test()` 测试用例。

在实验报告中，简要叙述：

1. C 语言中的 `struct` 和 `union` 关键字都是什么含义，寄存器结构体的参考实现为什么把部分 `struct` 改成了 `union`？

PA 1-2 整数的表示、存储和运算

我们首先考察整数的表示、存储和运算问题。在计算机中，一切都是离散的。用于表示各类信息的数据，尽管其含义可能千差万别，但最终在计算机内的表示总可以在形式上归于整数形式。因此，从整数开始讨论计算机内数据的表示、存储和运算，是一个较为理想的切入点。

§1-2.1 预备知识

在计算机中，采用二进制表示任何的数。整数可分为无符号整数和带符号整数两种。带符号整数的表示可分为原码表示和补码表示两种方法。数据的存储方式又可分为大端方式和小端方式两种。具体的背景知识请参见理论课的内容，在实验指导中不再赘述。

§1-2.2 代码导读和实验理解

§1-2.2.1 EFLAGS 寄存器

在进行整数的算术和逻辑运算之前，我们需要先为运算提供必要的支撑。在 CPU 内部有一个刻画当前 CPU 状态的寄存器称为标志寄存器 EFLAGS。其结构请参见 i386 手册第 2.3.4 节。EFLAGS 中包含一系列重要的标志位如我们现在就要用到的 CF、PF、ZF、SF、DF、OF 和将来要用到的 IF。在 `nemu/include/cpu/reg.h` 头文件中我们已经给出了 EFLAGS 寄存器的一个实现，其中使用到了 C 语言的位域 (bit field)，如不理解请上网搜索。

§1-2.2.2 整数的算术和逻辑运算

在整数表示的基础上，我们就能够来模拟 CPU 对整数的算术和逻辑运算功能了。在 CPU 内部，这一功能通过算术逻辑部件 (Arithmetic Logic Unit, ALU) 来实现。在 NEMU 中，对应于头文件 `nemu/include/cpu/alu.h` 和源文件 `nemu/src/cpu/alu.c`。在模拟这些运算功能时，我们的最终目标是要配合 x86 指令模拟的需求，为相应的算术和逻辑运算指令提供封装好的函数。因此，尽管目前还没有进展到对指令和指令的执行进行模拟的步骤，我们仍需从指令模拟的需求出发，对运算函数的需求进行刻画。

整数的加减操作

首先，我们需要模拟整数的加减运算。在采用补码表示法后，带符号和无符号的整数加减法可以统一使用无符号整数加减法来执行。因此，我们只需要实现无符号整数加减法即可。同时，考虑是否带进位和借位的加减法，我们一共需要实现四个无符号加减法函数，如下表所示。

函数名	功能描述
<code>uint32_t alu_add(uint32_t src, uint32_t dest)</code>	返回 <code>dest + src</code> ，并根据运算结果设置各标志位 (AF

	可以不模拟，下同)
<code>uint32_t alu_adc(uint32_t src, uint32_t dest)</code>	返回 <code>dest + src + CF</code> ，并设置各标志位
<code>uint32_t alu_sub(uint32_t src, uint32_t dest)</code>	返回 <code>dest - src</code> ，并设置各标志位
<code>uint32_t alu_sbb(uint32_t src, uint32_t dest)</code>	返回 <code>dest - src - CF</code> ，并设置各标志位

在实现上述函数时，我们仅需要使用 C 语言本身所提供的运算符即可。运算的具体语义可参照 i386 手册中对应指令（指令名称和函数名对应）的说明。在进行各标志位的设置时，参考 i386 手册附录 C（Appendix C Status Flag Summary）的相关内容以及对应的指令的详细说明。再次申明，作为简化的 x86 模拟器，我们目前不对 **AF** 标志位进行操作。

在 `nemu/src/cpu/test/alu_test.c` 源文件中，包含了一些针对整数算术和逻辑运算的单元测试用例。在目前这个阶段，只需要通过 `alu_test_add()`、`alu_test_adc()`、`alu_test_sub()`、`alu_test_sbb()` 这四个测试用例即可。仔细观察这四个测试用例，可以发现框架代码所采用的测试方法是构造一系列的测试输入（test input），然后将被测程序（我们的实现）包含运算结果和标志位在内的输出与一个“黄金版本”（golden version）的输出进行比较。框架代码通过内联汇编的方式来实现所谓的“黄金版本”。当然，聪明的各位不难想到可以直接采用测试程序中所使用的内联汇编法来实现整数的算术和逻辑运算操作。虽然此法简单高效正确，但无助于我们更深入地理解运算过程以及标志位设置的条件。

在此我们明确禁止各位采用内联汇编的方法来实现整数的算术和逻辑运算。

在 `alu_test.c` 源文件中我们只提供了一些样例测试输入。你可以通过增加测试输入的方法来提高测试覆盖率，提高找到潜在 bug 的可能性。

整数的移位操作

整数的移位操作从方向上可以分为左移和右移两种，从对符号位的处理方式上又可以分为逻辑移位和算术移位两种，因此一共可以分为算术左移（SAL）、逻辑左移（SHL）、算术右移（SAR）和逻辑右移（SHR）四种。可以参照 i386 手册对应的指令说明和附录 C（Appendix C Status Flag Summary）的相关内容来具体了解这四种操作具体的语义和符号位的设置方法。在框架代码中，移位操作对应下表所示的四个函数。

函数名	功能描述
<code>uint32_t alu_sal(uint32_t src, uint32_t dest, size_t data_size)</code>	返回将 <code>dest</code> 算术左移 <code>src</code> 位后的结果， <code>data_size</code> 用于指明操作数长度（比特数），可以是 8、16、32 中的一个用于判断标志位的取值，标志位设置参照手册说明
<code>uint32_t alu_shl(uint32_t src, uint32_t dest, size_t data_size)</code>	同 <code>alu_sal()</code>
<code>uint32_t alu_sar(uint32_t src, uint32_t dest, size_t data_size)</code>	返回将 <code>dest</code> 算术右移 <code>src</code> 位后的结果（高位补符）， <code>data_size</code> 用于指明操作数长度，标志位设置参照手册说明
<code>uint32_t alu_shr(uint32_t src, uint32_t dest, size_t data_size)</code>	返回将 <code>dest</code> 逻辑右移 <code>src</code> 位后的结果（高位补零）， <code>data_size</code> 用于指明操作

	数长度，标志位设置参照手册说明
--	-----------------

同样的，在完成了对应的移位操作函数后，可以通过执行测试用例来检测程序中的问题。

注意手册中提到移位指令只在单次移位时才会对 OF 位进行设置，在 NEMU 中我们忽略这个细节，不对移位操作后的 OF 位进行测试。

特别说明：针对上面四个移位操作，约定只影响 dest 操作数的低 data_size 位，而不影响其高 32 - data_size 位。标志位的设置根据结果的低 data_size 位来设置。

——感谢 16 级何峰彬、张明超同学的建议

整数的逻辑运算

整数的逻辑运算包括整数的与 (and)、或 (or)、非 (not)、异或 (xor) 操作。在我们的实验中，目前只需要实现与、或、异或这三个操作，非操作由于太过简单，可以在之后的指令实现中直接实现：

函数名	功能描述
<code>uint32_t alu_and(uint32_t src, uint32_t dest)</code>	返回两个操作数与的结果
<code>uint32_t alu_or(uint32_t src, uint32_t dest)</code>	返回两个操作数或的结果
<code>uint32_t alu_xor(uint32_t src, uint32_t dest)</code>	返回两个操作数异或的结果

在实现相应的函数后，同样可以通过执行测试用例来检查实现中可能存在的 bug。

整数的乘除运算

最后我们需要实现整数的乘除运算。整数的乘除运算按照是否带符号可以分为两组，无符号乘除法和有符号乘除法。现代计算机中有专门的乘法器来实现整数的乘除运算，在这里我们简化这一实现方案，将乘法也归于 ALU 的功能。

对于乘法而言，其能乘数和被乘数最大位数为 32 位，所得乘积的最大位数为 64 位。根据 i386 手册的描述，对于乘法指令而言，当乘积为 64 位（32 位）时，其高 32 位（16 位）置于 EDX（DX）寄存器中，而低 32 位（16 位）置于 EAX（AX）寄存器中。在目前阶段，我们暂不考虑乘积在寄存器中的存放方式。而仅仅实现两个 32 位整数乘法得到一个 64 位整数的运算操作。

而对于除法而言，当其除数为 32 位时，其被除数取最大位数为 64 位。同样根据 i386 手册，对于除法指令而言，当被除数为 64 位时，其高 32 位放在 EDX 寄存器中，而低 32 位放在 EAX 寄存器中。与乘法操作类似，在目前阶段，我们不关心在除法的除数和被除数在寄存器中如何存放，而仅关心两个 64 位整数相除得到一个 32 位整数的运算。

为了实现对乘除运算的支持，需要实现 6 个函数，如下表所示：

函数名	功能描述
<code>uint64_t alu_mul(uint32_t src, uint32_t dest, size_t data_size)</code>	返回两个操作数无符号乘法的乘积，data_size 为操作数长度（比特数），在

<code>int64_t alu_imul(int32_t src, int32_t dest, size_t data_size)</code>	设置标志位时有用
<code>uint32_t alu_div(uint64_t src, uint64_t dest, size_t data_size)</code>	返回两个操作数带符号乘法的乘积
<code>int32_t alu_idiv(int64_t src, int64_t dest, size_t data_size)</code>	返回无符号除法 <code>dest / src</code> 的商，遇到 <code>src</code> 为 0 直接报错退出程序
<code>uint32_t alu_mod(uint64_t src, uint64_t dest)</code>	返回带符号除法 <code>dest / src</code> 的商，遇到 <code>src</code> 为 0 直接报错退出程序
<code>int32_t alu_imod(int64_t src, int64_t dest)</code>	返回无符号模运算 <code>dest % src</code> 的结果
	返回带符号模运算 <code>dest % src</code> 的结果

最后两个模运算操作是 NEMU 新增的，在 IA32 指令集中并不存在。事实上，除法运算会在给出商的同时给出除法的余数，具体内容请参照 i386 手册中有关除法指令的描述。在当前阶段，我们将除法和模运算分开设置，方便将来在实现除法指令时调用。

注意手册对于 `imul` 指令对标志位设置的描述较为模糊，因此在框架代码中，我们针对 `imul` 操作的标志位设置不进行测试。

§1-2.3 实验过程及要求

1. 实现 `nemu/src/cpu/alu.c` 中的各个整数运算函数；
2. 使用 `make` 命令编译项目；
3. 使用 `./nemu/nemu` 命令执行 NEMU 并通过各个整数运算测试用例。

PA 1-3 浮点数的表示和运算

§1-3.1 预备知识

NEMU 是模拟的 x86 体系结构，浮点数的表示和运算参照 IEEE 754 标准的规定。在上一节所实践的整数表示方法中，小数点约定永远在最右侧，因此可以说是一种数的定点表示方法。也可以通过约定小数点在某一特定位置来表示带小数的数字。而本节的实验中，我们参照 x86 体系结构的现行标准，实现 IEEE 754 标准的浮点数表示和运算方法。于定点数表述方法不同，浮点数表示的一个数字，其小数点所在的位置是不定（浮动）的，因此称之为浮点数。

根据 IEEE 754 标准的规定，单精度（32 位）浮点数由最高位 1 位符号位（*sign*），8 位阶码部分（*exponent*）和 23 位尾数部分（*fraction*）构成，其具体的表示方法如下：

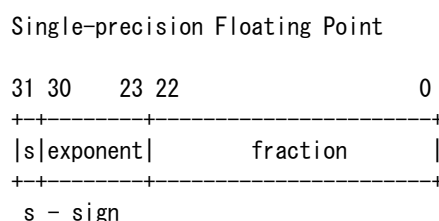


图 1-3 IEEE 754 浮点数结构

在单精度浮点数的表示中，8 位阶码部分采用移码形式，其偏置常数为 127；对于规格化数而言，23 位的尾数部分实际上表示了 24 位的有效数字，其最高位缺省为 1。尾数加上最高位可能存在的缺省的 1（非规格化浮点数没有缺省的 1）后构成的有效数字称为 *significand*。在 PA 实验中，我们仅关注单精度浮点数的表示和实现。双精度（64 位）浮点数的表示方法与单精度浮点数非常类似，只是阶码部分扩展为 11 位（偏置常数 1023），尾数部分扩展为 52 位。在以后的教程中，如无特殊说明，我们所称的浮点数均指单精度浮点数。

§1-3.2 代码导读和实验理解

§1-3.2.1 浮点数的表示

框架代码提供了 `FLOAT` 数据结构来方便地表示浮点数，其定义在头文件 `include/cpu/reg_fpu.h` 中，具体代码如下：

```
typedef union {
    struct {
        uint32_t fraction    :23;
        uint32_t exponent    :8;
        uint32_t sign       :1;
    };
    float fval;
    uint32_t val;
}FLOAT;
```

在拥有了 CPU 中寄存器结构体的实现的基本知识后，相信不难理解这段代码。在 FLOAT 结构中，我们在同一段 32 位的内存中，同时表示了一个浮点数类型以及其用 32 位整数所表示的值。在接下来的实验中，我们将实践针对浮点数的加减乘除操作，而这些操作都将在浮点数的 32 位整形表示的基础上展开。当然，我们可以利用 FLOAT 类型中的 fval 方便地实现浮点数的各类运算，这也正是我们在浮点数测试用例里实现黄金版本的方法。但出于教学目的，我们在此声明：

在实现模拟的浮点数的运算时，禁止使用将其整数表示通过类型转换成浮点数表示的方法来实现浮点数的运算操作。

§1-3.2.2 在 NEMU 中模拟浮点数运算

在 x86 体系结构中，浮点数的运算功能由浮点数协处理器提供。在 NEMU 中，这一系列操作则在模拟的 FPU 中实现。框架代码已经实现了对 FPU 的基本器件和所提供指令的模拟，即使在后面的指令实现实验中也不涉及浮点指令的模拟。对于框架代码模拟细节感兴趣的同学可以参见 Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, Chapter 8 的相关内容，以及在线手册 x86 Instruction Set Reference 中对相关指令的描述。在本实验中，我们仅将注意力集中在对浮点数运算的模拟上。

在头文件 include/cpu/fpu.h 中，框架代码声明了四个供内部使用的浮点数运算函数，具体如下表所示：

函数名	功能描述
uint32_t internal_float_add(uint32_t b, uint32_t a)	输入存储在 32 位无符号整型变量中的两个浮点数 a 和 b，返回存储在 32 位无符号整型变量中的浮点数运算结果 a + b
uint32_t internal_float_sub(uint32_t b, uint32_t a)	返回 a - b，注意变量次序
uint32_t internal_float_mul(uint32_t b, uint32_t a)	返回 a * b
uint32_t internal_float_div(uint32_t b, uint32_t a)	返回 a / b

NEMU 中模拟浮点数运算的基本思想是对符号位、阶码和尾数运用整数运算来模拟浮点数计算的过程。

浮点数的加减法运算

首先谈浮点数的加法运算。浮点数加法的大致过程可参照课本 2.7.7 节的相关内容展开。本教程结合课本内容，给出实现浮点数加法的大致思路。

第一步，判断边界条件。在进行加减运算之前，我们首先要判断一些包括参与运算的数字是 ± 0 、 $\pm \infty$ 、NaN 的边界条件。在框架代码中我们已经针对各种运算给出了对边界条件的判断。

第二步，提取尾数并对阶。当处理的浮点数 FLOAT f 是规格化数时尾数需要加上隐藏位 1，否则隐藏位为 0。上述过程可以用如下伪代码来描述：

// 提取尾数过程，假设对应的 32 位无符号整型的值存储在 uint32_t uf 中
FLOAT f;

```
f.val = uf;
uint32_t significand = f.fraction;
if (/* f 是规格化浮点数 */)
    significand |= 0x800000; // 加上隐藏位 1
// else do nothing
```

对参与加法的两个浮点数 `fa` 和 `fb` 都提取了其尾数后，需要执行对阶操作。参照课本描述，对阶的原则是：小阶向大阶看齐，阶小的那个数的尾数右移，右移的位数等于两个阶的差的绝对值。假设参与运算的两个浮点数为 `FLOAT fa, fb`。且 `fa` 的阶小于 `fb` 的阶。那么 `fa` 的尾数需要右移的位数为：

```
shift = fb.exponent - fa.exponent
```

这里有一个特殊情况就是需要处理非规格化浮点数的情况，对于非规格化浮点数而言，其阶码为全 0 而尾数为非 0。对于全 0 的阶码，其表示的真实值为 $0.f \times 2^{-126}$ ，不能理解为阶码为 $0-127 = -127$ 。因此，为了在数值上保证右移位数的正确性，需要对阶码进行判断（此时已经经过第一步，排除了浮点数为 0 的情况）。

```
shift = (fb.exponent == 0 ? fb.exponent + 1 : fb.exponent) - (fa.exponent == 0 ? fa.exponent + 1 : fa.exponent)
```

相加后和临时的阶码为 `b.exponent`。

在对 **fa** 进行对阶的过程中，尾数每右移一位都会在最高位补 0 并导致最低有效位移出。为了提高运算结果的精度，我们不能将移出的位直接丢弃。根据 IEEE 754 的规定，所有浮点数运算的中间结果右边都必须至少保留两位附加位，依次是保护位 (guard, G) 和舍入位 (round, R)。为进一步提高精度，舍入位右侧还有一个粘位 (sticky, S)，只要舍入位右边有任何非 0 数字，粘位就置为 1，否则置为 0。因此，在运算的中间过程中，实际上尾数应该是隐藏位 (1 位) + 小数部分 (23 位) + G (1 位) + R (1 位) + S (1 位) = 27 位：

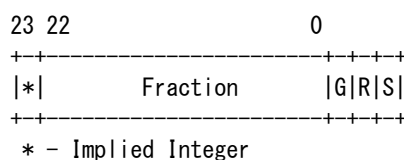


图 1-4 框架代码中带 GRS bits 的浮点数尾数结构

在尾数右边的 `GRS bits` 也需要参与运算来保持精度，如果采用额外的变量来存储 `GRS bits`，那实现起来就略显烦琐。在这里提供一种简便的思路，如第二步中的代码段所示，我们可以用一个 32 位无符号整型变量 `significand` 来暂存尾数部分。事实上，尾数部分即使算上隐藏位和 `GRS bits` 也就是 27 位。因此，我们可以将 `significand` 左移 3 位，为 `GRS bits` 空出位置来。此时等同于约定运算中间结果的 `fraction` 部分为 $23+3=26$ 位。

总结上述提取尾数并对阶的过程，我们将步骤整理如下：

1. 提取尾数至至少为 32 位（乘法中用 64 位较为便利）的无符号整型变量中，并根据是否是规格化数补上隐藏位得到 `significand`；
2. 将参与运算的两个尾数都左移 3 位，为 `GRS bits` 空出位置；
3. 确定阶小的那个数需要右移的位数 `shift`；
4. 将阶小的那个数的尾数右移执行对阶操作，移出部分自然进入 `G` 和 `R` 两位，`S` 位的取值需根据规则额

外操作。

第三步，尾数相加。浮点数的尾数部分采用的是原码表示，而浮点数的符号位保存在最高位。在做尾数相加时，只需按照符号位的取值将原码表示的尾数转变为补码表示，并做无符号整型加法即可。如果你采用了第二步中建议的左移留出 GRS bits 的方法，那此时这三位自然参与运算。

执行尾数相加后，再根据尾数之和的补码取值，判断结果的正负号。根据结果的符号设置运算结果浮点数的符号，并将尾数从补码转回原码表示。

第四步，尾数规格化。完成尾数相加后，我们接下来需要对尾数进行规格化。框架代码给出了一个尾数规格化的内联函数，其原型为：

```
inline uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

其中 `sign` 是运算结果的符号，`exp` 为待规格化的阶码，是一个带符号的整数，采用带符号的阶码是为了方便应对在乘除法运算时可能获取负数阶码的情况，`sig_grs` 为待规格化的尾数，从变量名中可以反映出这个尾数中是包含 GRS bits 的，同时也包含最高位的隐藏位 (Implied Integer)，采用 64 位整型临时存放尾数是为了方便乘除法的实现。对于加减法而言，在尾数规格化的过程中，我们可以根据最终 Implied Integer 部分的取值来判断是需要进行左规还是右规。具体规则如下：

如果 `sig_grs >> (23 + 3) > 1`，则需要右规直至 `sig_grs >> (23 + 3) == 1`；反之如果 `sig_grs >> (23 + 3) == 0` 且不是非规格化浮点数 (`exp > 0`)，则需要左规直至 `sig_grs >> (23 + 3) == 1`。注意这里的 `23 + 3` 的取值是因为我们在临时尾数的最低三位保留了 GRS bits 的缘故，也就是上文所述的等同于中间结果的 fraction 部分约定为 26 位。在规格化过程中，根据规格化的方向确定尾数右移还是左移，并对阶码进行增减操作。右规过程中需要保留粘位。同时在每次右移前都要检查阶码上溢 (`exp >= 0xff`) 的情形。在左规过程中，则有可能发生结果出现非规格化浮点数的情况 (阶码变为 0)，此时需要将尾数额外右移 1 位以对应非规格化浮点数阶码是 2^{-126} 的约定 (否则单纯从数值上看阶码全 0 对应 2^{-127})。

假设在上述规格化过程中没有发生阶码上溢或下溢，则完成规格化后需要对 GRS bits 进行舍入操作。舍入的方法采用就近舍入 (中间值舍入到偶数)，三位 GRS bits 对应的中间值为 4。确定舍入后，完成尾数最后三位的右移操作，丢弃 GRS bits。最后，由于在舍入过程中可能涉及尾数加一。因此还要再判断一次是否需要右规。对于规格化浮点数，不要忘记丢弃最高的隐藏位。

完成上述步骤后，我们就能够确定最终的运算结果 (包括符号位、阶码和尾数) 了，利用 `float` 类型可以方便得到浮点数对应的 32 位整型表示。为了方便理解上述过程，我们用伪代码给出尾数规格化函数的实现方案如下：

```
inline uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs) {
    if (/* 需要进行右规 */) {
        while (/* 需要右规 且 未发生阶码上溢 */) {
            /* 右规一次 */
        }
        if (/* 发生了阶码上溢 */) {
            /* 根据符号将结果置为 +∞ 或 -∞ */
        }
    } else if (/* 需要进行左规 */) {
        while (/* 需要左规 且 阶码大于 0 */) {
            /* 左规一次 */
        }
        if (/* 发生了阶码==0 */) {
```

```

        /* 右移一次化为非规格化浮点数 */
    }
} else if( /* 两个非规格化数运算后得到了一个规格化数 */ ){
    exp++;
}

if( /* 规格化过程中未发生溢出 */ ){
    /* 根据 sig_grs 最后三位 GRS bits 的取值决定舍入，采取就近舍入到偶数的方式 */
    /* 移除 sig_grs 最后三位保留的 GRS bits*/
    if( /* 舍入后破坏了规格化 */ ){
        /* 再进行规格化并判断溢出 */
    }
}

// 假设最后的结果的符号、阶码、尾数保留在 sign, exp, sig_grs 中
FLOAT f;
f.sign = sign;
f.exponent = (uint32_t)(exp & 0xff);
f.fraction = sig_grs; // here only the lowest 23 bits are kept
return f.val;
}

```

参照上述过程, 在 `src/cpu/fpu.c` 中实现浮点数加法函数 `internal_float_add()` 和规格化函数 `internal_normalize()` 并通过位于 `src/cpu/fpu_test.c` 中的测试用例 `fpu_test_add()` 即可。再次强调, 我们不接受测试用例中将无符号整型表示的浮点数先转成 `float` 类型后运算的实现方案。

在实现了加法运算后, 减法运算只需将减数符号置反后做加法即可。有一些额外的边界条件需要判断, 为节省时间, 框架代码已给出相应的实现。

浮点数的乘除运算

在充分掌握了浮点数的加法运算后, 浮点数的乘除运算就十分简单了。其基本步骤和加法类似, 相比加法运算, 还可以免去对阶的过程。这里只作几点提示:

1. 尾数的中间结果用 `uint64_t` 表示更为便利;
2. 阶码在做加法(乘法)和减法(除法)时, 需要考虑对偏置常数(单精度浮点数为 127)所带来的影响, 同时必须考虑处理非规格化浮点数时阶码全 0 其实在表示的数值上等于阶码为 1 (表示 2^{-126}) 的情形;
3. (特别说明: 如果你觉得这一段难以理解, 可以参考脚注的说法⁴) 在做尾数除法之前, 为提高运算精度, 先将被除数(假设为 `fa`) 尾数存放在 64 位临时变量中并尽可能左移(直至最高位为 1), 同时将除数(假设为 `fb`) 的尾数尽可能的右移。经过以上处理后再将被除数的尾数除以除数的尾数。假设在上一步左移加右移的次数为 `shift`, 此时对应于结果的阶码应该是 `fa.exponent - fb.exponent + 127 -`

⁴ 如前所述, 加入 GRS bits 后, 等同于约定中间结果 `fraction` 为 26 位。对于除法而言, 将被除数左移 `shift` 位后除以除数得到的 `fraction` 为 `shift` 位, 为了保证中间结果 `fraction` 为 26 位, 将多余的 `shift - 26` 位从阶码中减去, 最后的阶码等于 `fa.exponent - fb.exponent + 127 - (shift - 26)`。对于乘法而言, 用两个 `fraction` 为 23 位的尾数相乘得到的乘积尾数 `fraction` 为 46 位, 为了保证中间结果的 `fraction` 依然是 26 位, 我们将多出来的 `46 - 26 = 20` 位归于阶码, 因此阶码就是 `fa.exponent + fb.exponent - 127 - 20`。在此感谢 15 级姚荣春助教在审阅过程中给出的建议!

(shift - 23) + 3。最后减去从 shift 中减 23 是因为用整数模拟小数除法时，我们模拟的是小数除法 1.aaa / 1.bbb，而实际进行的运算是整数除法 1aaa / 1bbb，为了使得最后的结果符合小数除法的运算规则，应该将被除数尾数左移 23 位再进行除法运算。而为了提高精度，我们左移了 shift 位，多移动了 shift - 23 位，在此处补上。另外最后三位依照先前的约定是 GRS bits，原则上我们应该配合约定将中间结果左移 3 位再提交给规格化函数。我们省略左移过程，改为通过阶码加 3 来进行补偿。类似的，在进行乘法时，两个整数表示的尾数通过乘法获取 64 位的乘积后，结果的阶码要减去 23，并保留 GRS bits，最终的阶码为 fa.exponent + fb.exponent - 127 - 23 + 3（注意：尽管我们做了这么多的努力去提高精度，还是有可能在测试中出现尾数的最后一位与标准结果差 1 的情况，我们在测试用例的 assertion 中列举了这种例外情况）；

4. 在 internal_normalize() 函数中，右规的条件应增加一种情况，即 $\text{exp} < 0$ 的情况，对应于乘除法获取的结果临时阶码非常小（小于 -126）的情况，此时的一次右规也可以称为一次逐级下溢，试图将结果变为非规格化数。此时在右规的 while 循环退出后，需要判断得到的结果是否是规格化浮点数；同时也需要判断是否退出 while 循环后阶码仍为负数的情形（此时发生阶码下溢）。补充之后 internal_normalize() 规格化函数的伪代码如下：

```
inline uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs) {
    if (/* 需要进行右规 (或 扩充条件 exp < 0) */){
        while (/* (需要右规 且 未发生阶码上溢) 或 (扩充条件 exp < 0 且 尾数没有被右规至舍入后
为 0, 即, sig_grs > 0x4) */){
            /* 右规一次 或称 逐级下溢一次 */
        }
        if (/* 发生了阶码上溢 */){
            /* 根据符号将结果置为 +∞ 或 -∞ */
        }
        if (/* 阶码为全 0, 得到了非规格化的浮点数 */){
            /* 右移一次化为非规格化浮点数 */
        }
        if (/* exp 仍然小于 0, 发生阶码下溢 */){
            /* 根据符号将结果置为 +0 或 -0 */
        }
    } else if (/* 需要进行左规 且 阶码大于 0 */){
        while (/* 需要左规 且 阶码大于 0 */){
            /* 左规一次 */
        }
        if (/* 发生了阶码==0 */){
            /* 右移一次化为非规格化浮点数 */
        }
    } else if (/* 两个非规格化数运算后得到了一个规格化数 */){
        exp++;
    }

    if (/* 规格化过程中未发生溢出 */){
        /* 根据 sig_grs 最后三位 GRS bits 的取值决定舍入, 采取就近舍入到偶数的方式 */
        /* 移除 sig_grs 最后三位保留的 GRS bits */
        if (/* 舍入后破坏了规格化 */){
            /* 再进行规格化并判断溢出 */
        }
    }

    // 假设最后的结果的符号、阶码、尾数保留在 sign, exp, sig_grs 中
}
```

```
    FLOAT f;
    f.sign = sign;
    f.exponent = (uint32_t) (exp & 0xff);
    f.fraction = sig_grs; // here only the lowest 23 bits are kept
    return f.val;
}
```

浮点数的乘除法对应于 `src/cpu/fpu.c` 中 `internal_float_mul()`和 `internal_float_div()`两个函数。所有的测试用例都位于 `src/cpu/fpu_test.c` 中。`test_fpu()`函数会在 NEMU 启动时调用。

§1-3.3 实验过程及要求

1. 实现 `nemu/src/cpu/fpu.c` 中的各个整数运算函数；
2. 将 `internal_normalize()`函数补完；
3. 使用 `make` 命令编译项目；
4. 使用 `./nemu/nemu` 命令执行 NEMU 并通过各个浮点数运算测试用例。

在实验报告中，回答以下问题：

为浮点数加法和乘法各找两个例子：1) 对应输入是规格化或非规格化数，而输出产生了阶码上溢结果为正（负）无穷的情况；2) 对应输入是规格化或非规格化数，而输出产生了阶码下溢结果为正（负）零的情况。是否都能找到？若找不到，说出理由。

PA 2 程序的执行

通过 PA 1 的努力，我们已经使得 NEMU 具备了初步的数据表示、存储和运算能力，可以说是搭建了一个功能强大的计算器。但是，我们还不能让我们的机器按照我们的意愿去进行工作。那要怎样做才行呢？作为计算机方向的专业选手，我们已经知道答案：写程序！那程序到底是什么，计算机是怎样运行程序的？这就是我们通过这一阶段的 PA 需要解决的问题。

在基本的理论学习之后，我们了解到，其实所谓程序就是由一条又一条称为指令的 01 串所构成的序列。CPU 能够解读这些由 01 串编码好的指令并将其转换成对应的操作。硬件的设计者和软件开发者之间达成一项协议：什么样的 01 串对应什么样的指令，会产生什么样的操作。这项协议的具体内容就构成了所谓的指令集体系结构（Instruction Set Architecture, ISA），也就是 i386 手册所论述的东西。

在真实的机器上，ISA 是通过半导体芯片来实现的，这就是我们能够看得见摸得着的硬件。而 NEMU 则是一个软件实现的模拟器，它能够和硬件实现的机器一样用来运行程序吗？答案是肯定的，其原因就在于：对于上层的程序而言，不管是怎么实现的，只要有人实现了 ISA 的功能，我就能通过编排指令序列来完成需要的计算功能⁵。因此，我们只要用软件来造一台能够读懂指令并模拟其执行过程的“软”机器就可以了。这就相当于给应用程序创造一个绝对逼真的虚拟现实环境，程序在其中分不清是真是幻，于是对程序而言模拟器和硬件机器就没有区别。

⁵ 当然，除了运算以外，ISA 中的有些指令还设计到对包括键盘、显示器等外部设备的输入输出功能也需要我们在 NEMU 中进行模拟，这就是我们在 PA 4 里要做的工作。

PA 2-1 指令解码与执行

在完成了基本的运算功能后，我们希望计算机能够按照我们的命令来执行各种运算。在这一阶段，我们就要赋予 NEMU 这样的能力。

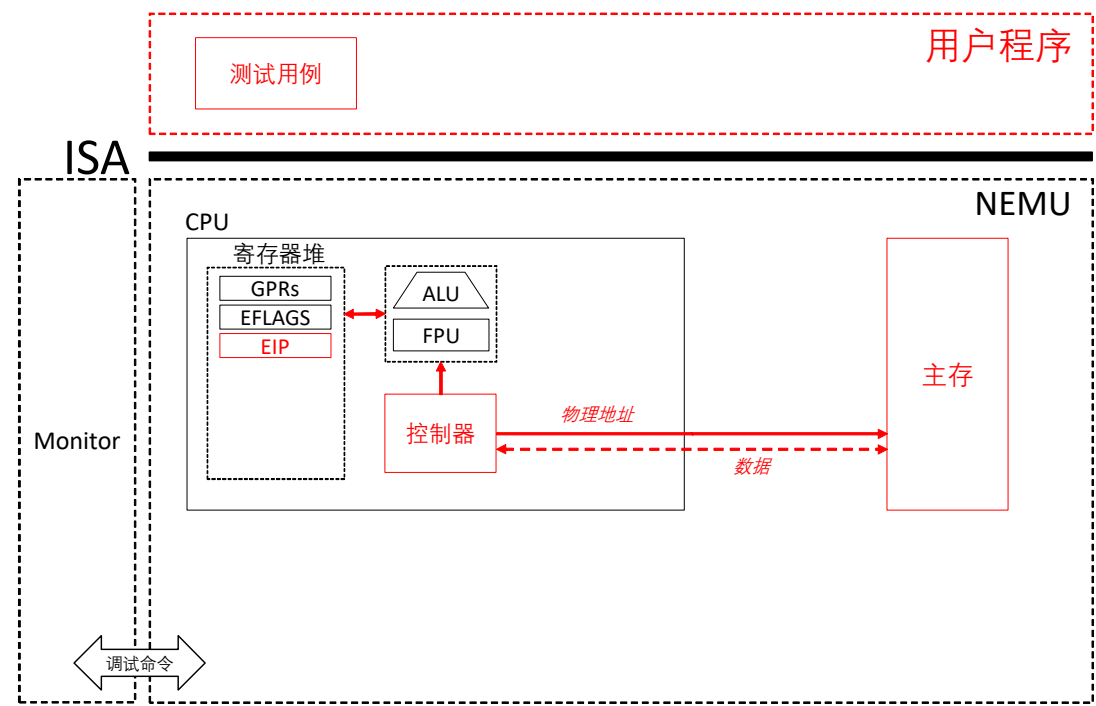


图 2-1 PA 2-1 路线图

§2-1.1 预备知识

在计算机中，一切信息都按照某种规则编码为由 01 串组成的数据形式，在 PA 1 中所介绍的整数和浮点数是如此，指挥机器运行的指令也是如此。每一条指令都指明了计算机所需要进行的一步操作（操作码）和操作的对象（操作数）。若我们使用机器语言写程序，那么一个程序就是一个指令的序列。这个序列规定了计算机解决问题所需要执行的各个步骤。若我们使用高级语言（如 C 语言）写程序，那么往往需要使用编译器将高级语言程序编译成指令序列后再交给机器去执行。

机器执行程序的过程就是按序执行事先编制好的指令序列的过程。

§2-1.1.1 指令的格式

NEMU 模拟的是一个简化的 IA-32 体系结构，采用同样的指令集体系结构。

指令的格式

一条指令从其基本的构成上来说，需要包含两部分内容：

1. 指令的操作码 (opcode)：用于指明指令所对应的行为是什么，如数据移动、加法运算、跳转等；
2. 指令的操作数 (operand)：对于涉及数据操作的指令，我们还需要指明指令所操作的对象是什么。操作数可能是指令中直接给出的一个数字，称为立即数 (immediate)；也可以是一个寄存器编号；或者是一个内存地址。

在 IA-32 体系结构中，一条指令的格式可以由下图来表示：

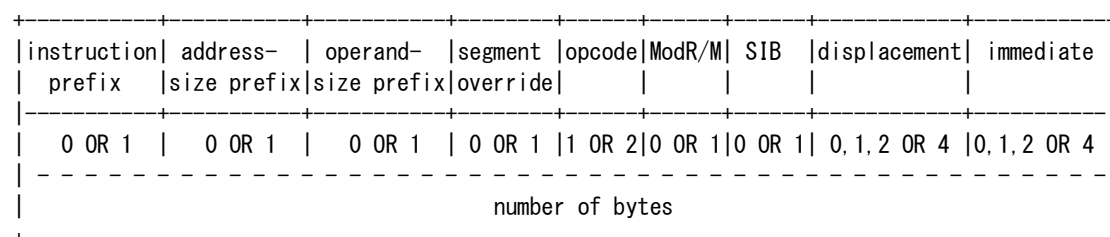


图 2-2 指令格式

其中包含的信息如下：

1. 指令中必须的部分是操作码 (opcode)，它指出指令对应要执行的功能是什么；
2. 在操作码前面可能会出现一些前缀 (prefix)，用于指明指令采用的地址长度 (address-size) 或操作数长度 (operand-size) 等。在 NEMU 中，我们只考虑操作数长度前缀，其值为 0x66。当 opcode 所对应的操作数长度可变时 (16 位或 32 位)，若 opcode 前面出现 0x66，则操作数长度临时改变为 16 位，否则为 32 位。操作数长度是 8 位的指令往往其 opcode 区别于其它操作数长度的同类指令，不会产生混淆；
3. ModR/M、SIB、displacement 这三个域通过各种组合构成了除 1) 立即数寻址，2) 在 opcode 中直接指定使用某一个寄存器，以外的所有可能的操作数寻址方式。结合课本第 93 页图 3.4 了解所有可能的寻址方式；
4. Immediate 部分指定了指令中可能出现的立即数，对应立即数寻址方式。

指令的解码和执行

针对 IA-32 指令集，我们从实用主义的角度，通过一个例子结合 i386 手册来介绍指令的解码过程。等一条指令解码完成，我们就清楚其详细的含义，模拟其对应的执行就很直接了。

假设当前我们的 PC 指向下面这串代码段数据 (16 进制) 的第一个字节 (0xC7)：

8b 94 83 00 11 00 00 8b 45 f4

第一步，看第一个字节。发现不是 0x66，那么 0x8b 就是 opcode。

第二步，使用 `opcode` 去 i386 手册的 Appendix A Opcode Map 部分查表。发现 `0x8b` 对应的操作为 `MOV Gv, Ev`。请注意 i386 手册使用的是 Intel 格式的汇编，而我们的教程和 NEMU 中使用的是 AT&T 格式的汇编，其主要区别是操作数顺序正好颠倒，在 Intel 格式中，源操作数在右边，目的操作数在左边。因此这条指令的含义是将一个 `Ev` 源操作数 `MOV` 到 `Gv` 这个目的操作数中去。

在 Appendix A Opcode Map 的开头部分，我们了解到 `Ev` 和 `Gv` 分别指代一个变长的（`v` 指明操作数可能是 16 位或 32 位）由 `Mod/RM` 字节所指定的寄存器或内存地址，和一个变长的由 `Mod/RM` 字节的 `reg` 域指定的通用寄存器。由于我们在第一步就了解到 `opcode` 之前不存在 `0x66` 操作数长度前缀，因此操作数长度确定为 32 位。

第三步，如果不清楚 `MOV` 的具体细节，到 i386 手册的 Chapter 17 部分仔细查找，定位到第 345 页的说明。仔细阅读其中的内容，其中包含了每一条指令所涉及的每一项细节且没有歧义。当然，其中难免存在一些笔误，在本教程的最后部分，我们附上一份勘误，其中包含了目前已经发现的手册中的错误。

通过以上三步，我们已经完成了对 `opcode` 的解码，知道这条指令要做什么了，接下来，就是进一步确定其操作数的过程：

第四步，解码操作数。在第二步中我们就已经了解到，`opcode` 后面跟着的是 `ModR/M` 字节。于是确定 `0x94` 是 `ModR/M` 字节。`ModR/M` 字节的结构请查阅 i386 手册手册第 17.2.1 节。其解析方式对应 Table 17-3。通过查表，我们发现，`0x94` 这个 `ModR/M` 字节的 `reg` 域为二进制 `010`，结合 `Gv` 的说明，得知目的操作数为 2 号通用寄存器，即，`%edx`。而进一步组合 `Mod` 和 `R/M` 域，我们得知源操作数形式为 `disp32[--] [-]`，还需要解析 `SIB` 字节。

`0x83` 于是确定为 `SIB` 字节。`SIB` 字节结构请查阅 i386 手册手册第 17.2.1 节。通过查表 Table 17-4，得知 `base` 为 `%ebx`，`scaled index` 为 `%eax * 4`。注意手册中有关 `ModR/M` 和 `SIB` 字节的解析表都存在一些笔误，请同样查阅教程最后的勘误部分。

根据 `ModR/M` 字节所指出的操作数格式，最后我们还跟着一个 32 位的偏移量 `disp32`。所以后续的四字节内容 `00 11 00 00` 是 `disp32`。小端方式，对应值 `0x1100`。从而得知源操作数存储在主存位于地址 `0x1100 + %ebx + %eax * 4` 的 4 字节（32 位）存储区域中。

至此，这条指令中所有信息都已清楚，指令解码完成。通过上述一步一步地解码操作码和操作数，我们得知原来代码段的这一串数据：

```
8b 94 83 00 11 00 00
```

对应用 AT&T 格式汇编书写的指令：

```
mov 0x1100(%ebx, %eax, 4), %edx
```

而后面的数据，就是属于下一条指令的内容了。在解码并执行完当前指令后，只需将程序计数器指向下一条指令的首地址，并重复上述过程便可以执行指令的序列了。

§2-1.1.2 CPU 顺序执行指令序列的过程

当程序员写好程序，并将对应的指令序列存储到主存的某一段位置后，便可以请求机器来执行这个程序了。在目前阶段，我们不考虑如流水线、乱序发射等复杂的执行方式，仅考虑最基本的顺序执行指令序列的情形。无论是多复杂的执行方式，都可以看做是对执行效率的优化，不改变某一个程序执行的基本逻辑：按照程序员指定的指令序列去一条接一条的执行指令，从而完成某一项工作。

当程序在内存中就位后，机器便可以开始执行程序了。若机器和程序员约定好第一条指令的位置，那么机器就能够在时钟脉冲的激励下，从第一条指令开始，一条接一条地解码并执行每一条指令（参照上一小节中所述过程）。这个过程可以理解成一个循环：

1. 获取当前要执行的指令，约定当前指令的位置存放在程序计数器（PC）中；
2. 对指令的操作码、源操作数和目的操作数进行解码；
3. 取源操作数；
4. 执行解码好的指令；
5. 存目的操作数；
6. 计算下一条指令的地址，更新 PC 并跳转到第 1 步。

以上过程可以结合课本第 201 页的图 5.1 进行理解。

§2-1.2 代码导读和实验理解

在掌握了基本的预备知识后，就可以在 NEMU 中模拟指令的解码和执行过程，进而执行各种各样的用户程序了。

§2-1.2.1 测试用例

使用 make run 和 make test 运行测试用例

框架代码提供两条命令来编译和执行测试用例，分别是 `make run` 和 `make test`。

在项目根目录控制台键入 `make run` 之后，`make` 程序会搜索根目录下的 `Makefile` 文件，并执行其中的 `run` 目标。这个目标有两个依赖目标 `nemu` 和 `testcase`，分别对应编译 NEMU 和测试用例的操作。若编译成功，会执行命令 `./nemu/nemu -run <testcase_name>`，其中的 `<testcase_name>` 对应的是测试用例的名称，也就是 `testcase/src/` 文件夹下各个测试用例程序去除后缀之后的文件名。

在执行命令 `./nemu/nemu -run <testcase_name>` 时，会首先进入 NEMU 的 `main()` 函数。在 `main()` 函数中，NEMU 首先根据执行参数配置好测试用例的文件路径，并执行 `single_run()` 函数。`single_run()` 函数对应 NEMU 的一次执行。在 `single_run()` 函数中，会完成一系列的初始化工作，最后进入 `ui_mainloop()` 函数。`ui_mainloop()` 函数是 NEMU 中 `monitor` 的一部分。`Monitor` 提供了一个基于命令行的交互式界面，我们可以在这个交互式界面中通过键入各种命令来实现包括执行测试用例、设置断点和监视点、查看寄存器和内存等程序状态信息的功能。下表总结了 `monitor` 所提供的命令及其含义：

命令	格式	使用举例	说明
帮助	<code>help</code>	<code>help</code>	打印帮助信息
继续运行	<code>c</code>	<code>c</code>	继续运行被暂停的程序

退出	q	q	退出当前正在运行的程序
单步执行	si [N]	si 10	单步执行 N 条指令，N 缺省为 1
打印程序状态	info <r/w>	info r info w	打印寄存器状态 打印监视点信息
表达式求值 *	p EXPR	p \$eax + 1	求出表达式 EXPR 的值 (EXPR 中可以出现数字, 0x 开头的十六进制数字, \$开头的寄存器, *开头的指针解引用, 括号对, 和算术运算符)
扫描内存 *	x N EXPR	x 10 0x10000	以表达式 EXPR 的值为起始地址, 以十六进制形式连续输出 N 个 4 字节
设置监视点 *	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时, 暂停程序运行
设置断点 *	b EXPR	b main	在 EXPR 处设置断点。除此以外, 框架代码还提供了宏 BREAK_POINT, 可以插入到用户程序中, 起到断点的作用
删除监视点或断点	d N	d 2	删除第 N 号监视点或断点

* 表达式 EXPR 求值的功能在可选任务 PA 2-3.1 中实现, 在此之前你可能无法使用这些命令, 但你可以使用一些比较简单的实现, 如约定 EXPR 一定是一个 16 进制数字这样的方式来开启这些命令。

进入交互式界面后, 可以通过 c 命令执行测试用例。

与 make run 类似, make test 也提供了编译和运行测试用例的功能, 区别是它会自动循环执行 nemu/src/main.c 中的 testcases 数组所列举的测试用例。只有当触发断点时才会进入 monitor 的交互式界面。

在实现指令的过程中, 可能会遇到大量的 bug 需要调试。要学会善用断点和测试用例。虽然我们最后会以框架代码所提供的测试用例作为基准, 但是这并不意味着在实验过程中不能增加或修改测试用例。你可以不断地构造新的测试用例, 也可以剪裁已有的测试用例, 来缩小被测试代码的范围, 从而更加准确地定位 bug 并修复之。程序中的逻辑错误到其真正引起一个能被发现的软件故障可能会经过很长的执行过程。有兴趣的同学可以查阅一下软件 bug 从内部 fault 到 error 到外部 failure 传递过程的相关资料。因此, 我们希望程序在一出现内部 fault 的时候 (执行到出 bug 的那行代码) 就立马抛出错误引起我们的关注。那么就需要我们大量运用单元测试并通过 assert 语句检查执行的结果是否符合预期。PA 1 中的 reg_test() 和 alu_test() 给出了单元测试的良好例子。前一个通过对比预期输出的方式来检查可能存在的错误, 而后者通过和一个黄金版本的输出进行比对的方式来检查可能的错误。同时这两者每次都是针对非常小的一个程序单元进行高强度的测试, 这样方便我们尽早地查出 bug, 并将问题的范围控制在很小的区域内。在通过了 alu_test() 之后, 在本阶段调用 alu.c 中的函数实现相应的算术和逻辑运算指令并设置标志位寄存器时就有充足的信心了。

你可以使用 monitor 所提供的命令来执行测试, 也可以在代码中插入 BREAK_POINT 宏或者 printf() 语句来打印输出内部状态。后面这种方法称为插桩 (instrumentation), 很适合初学者使用。但是当进入成熟的项目开发时, 这种方法却往往不值得鼓励, 这是因为插桩的方法临时修改了目标代码的行为, 而在项目发布

时又需要被移除，导致被测代码和发布的代码有所不同，可能会导致程序出现不确定的行为。有一类 bug 叫做 Heisenbug，有兴趣的同学可以查阅相关的资料。

Testing & Debugging 是每一个程序员和程序员都需要掌握的重要的能力，唯有通过不断地练习方能掌握，虽然很希望在这门课上多讲一点，但是时间不允许，关键在于实践出真知。为避免大家把时间浪费在无意义的猜测上，我们在本课程所涉及的编码范围内提出两条调试公理：

1. 机器永远是对的，错的只能是我们的程序；
2. 未测试代码永远是错的，出了 bug 不要感到意外。

此外，框架代码虽然经过了大量测试，但是我们依然无法保证其正确性。因为测试无法证明一个软件是正确的（形式化验证可能是一种办法，但目前的技术仍旧代价高昂）！即使一个函数的参数只有一个 `uint32_t` 类型，我们也难以在可接受的代价下穷举其所有可能的输入和输出，更不用说复杂的参数组合和函数调用关系了，执行测试只能通过触发 bug 从而发现存在的错误。因此，如果你找出了框架代码的 bug 并提出修复方案，我们会给予奖励。

NEMU 执行测试用例

在这一阶段，框架代码提供了一些测试用例（testcase）来测试指令的实现情况。测试用例的源代码位于 `testcase/src/` 文件夹下。使用 `make` 编译测试用例后，会在 `testcase/bin/` 文件夹下为每个测试用例生成两个文件：1) ELF 可执行目标文件（不带后缀），和 2) 内存镜像文件（带 `.img` 后缀）。在实现 ELF 文件的装载功能前，我们通过直接将测试用例的镜像文件拷贝到模拟内存中的方式来加载测试用例。

NEMU 约定程序的第一条指令存储在内存 `0x30000` 处。因此当 NEMU 初始化时会做两件事情：

1. 把测试用例镜像文件的内容直接拷贝到内存从 `LOAD_OFF = 0x30000` 处开始的连续区域内；
2. 将 EIP 初始化为 `INIT_EIP = 0x30000`。

第一步对应的代码位于 `nemu/src/main.c` 中 `single_run()` 函数的 `load_image(image_path, LOAD_OFF)` 一句。同样位于 `single_run()` 函数中的 `load_image(elf_path, 0)` 一句则是将测试用例的 ELF 可执行目标文件装载到内存从 `0x0` 开始的连续区域内，留待后续在 PA 2-2 中使用。我们约定在实现硬盘之前，即，`HAS_DEVICE_IDE` 宏被定义之前，我们使用内存物理地址 `0x00000 - 0x30000` 的区域作为内存模拟的硬盘（RAM Disk）。当进行到 PA4 阶段实现了对磁盘的模拟后，我们就不再需要在内存中模拟磁盘空间，这一段内存就等于弃之不用了。此时内存的划分方式如下图所示：

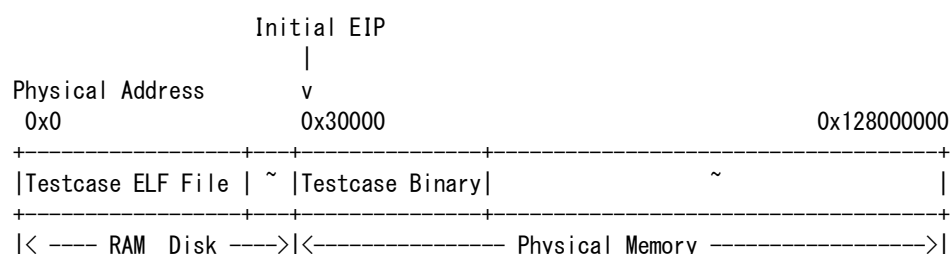


图 2-3 带有 RAM Disk 时的内存划分方式

第二步对应的初始化代码在 `nemu/src/cpu/cpu.c` 中的 `void init_cpu()` 函数中。在完成初始化后，CPU 就可以从第一条指令开始，按照上述指令执行的过程，循环往复地执行每一条指令，直至退出。

§2-1.2.1 指令循环的模拟

CPU 执行一条指令的过程大致包含取指令、译码、执行、计算下一条指令地址这几个步骤。指令地址总是存放在程序计数器（PC）中，在 NEMU 中，指令地址总是放在 EIP 寄存器中。

在真实的机器中，CPU 在时钟脉冲的激励下不断地重复指令执行的过程来实现对整个指令序列的执行。在 NEMU 中，我们简化这一措施，采用 `while` 循环的方式来模拟循环往复地执行指令的过程。指令循环的实现对应 `nemu/src/cpu/cpu.c` 中 `void exec(uint32_t)` 函数中的 `while` 循环。其中，`exec()` 函数的参数为需要执行的指令条数，当满足条件时，CPU 将不断地执行指令。在 `exec()` 函数的 `while` 循环中，语句 `len = exec_inst()` 调用了函数 `exec_inst()`。通过阅读代码注释，可以了解到 `exec_inst()` 函数的功能是执行 EIP 指向的指令，并返回指令的长度。因此，语句 `len = exec_inst()` 以及其后续的语句 `cpu.eip += len` 的含义就非常清晰了。

在 `exec_inst()` 函数中，语句 `opcode = instr_fetch(cpu.eip, 1)` 的作用比较直观，即，到当前 EIP 所指向的内存地址中取长度为 1 字节的数据，该数据就是待执行的指令。接下来语句 `int len = opcode_entry[opcode](cpu.eip, opcode)` 的功能理解起来需要一些技巧。按照注释，其功能为解码并执行指令，但其形式却是数组访问，何解？

为了回答上述问题，我们需要对 `opcode_entry` 数组进行深刻地理解。该数组的定义位于 `nemu/src/cpu/decode/opcode.c` 源文件中。我们看到 `opcode_entry` 数组的类型是 `instr_func`。从 `nemu/include/cpu/instr_helper.h` 头文件中，能够找到 `instr_func` 的类型定义为：

```
typedef int (*instr_func)(uint32_t eip, uint8_t opcode)
```

可见 `instr_func` 类型定义为一个函数指针，指向一个包含两个参数，返回一个 `int` 型数据的函数。结合 `exec_inst()` 函数中的数组访问语句 `int len = opcode_entry[opcode](cpu.eip, opcode)`，现在能够理解其实该数组访问就是调用 `opcode_entry` 中第 `opcode` 个函数，并把当前的 EIP 和指令操作码传递给该函数，获取其返回的整型数据，该返回值的解释应为指令的长度（注意在某些特殊情况下，如 `jmp` 指令中，我们会直接修改 EIP 的值指向跳转地址，此时根据实际情况，会将返回的指令长度置为 0，避免导致目的跳转地址在 `eip += len` 过程中发生偏差）。

既然每一条指令都能够用其操作码来唯一标识，那么，将指令对应的执行函数放在 `opcode_entry` 数组中的 `opcode` 位置，NEMU 所模拟的 CPU 就能够顺利地通过数组访问的方式，执行该指令了。再结合在 `exec()` 函数中 `while` 循环不断执行指令并指向下一条指令的操作，我们就能够按照顺序，执行程序指令序列了。

§2-1.2.2 模拟指令解码和执行

在对框架代码的指令执行方式有了初步了解后，我们可以开始着手模拟指令的实现了。在上一节介绍的内容中，我们已经完成了取指令和指令操作码译码的过程，接下来就是要针对每一条具体的指令，完成计算源操作数地址、取操作数、执行数据操作、计算目的操作数地址、存结果、计算下一条指令地址这一系列的过程。

一条简单 mov 指令的实现

我们先来考察一条最简单的 mov 指令，其功能是将一个一字节的立即数存入一个寄存器。假设该指令对应的机器码（16 进制）如下：

B1 01

通过查阅 i386 手册，我们知道，这是一条 mov 指令，其源操作数以立即数的方式给出，为 0x01，而目的操作数则在 opcode 的低 3 位给出，是一个通用寄存器，可以通过操作 `opcode & 0x07` 的方式得到。在此例中，目的操作数为 1，对应 `%ecx` 寄存器。指令所需要实现的功能就是将跟在 opcode 之后的一个 8 位立即数放到 opcode 低 3 位所给出的寄存器中。

回忆在介绍 `opcode_entry` 数组时的指令实现函数的声明为 `int instr_func(uint32_t eip, uint8_t opcode)`，那么在实现上述指令时，我们只需要实现一个对应的 `instr_func` 即可。下面的代码给出了这条 mov 指令的实现方法：

```
int mov_i2r_b(uint32_t eip, uint8_t opcode) {
    uint8_t imm = instr_fetch(eip + 1, 1); // 获取立即数
    uint8_t regidx = opcode & 0x7; // 获取寄存器编号
    cpu.gpr[regidx]._8[0] = imm; // 完成 mov 动作
    return 2; // 返回指令长度
}
```

可见上述代码准确反映出了这条指令从取操作数，执行数据操作，存结果，计算下一条指令地址等一系列工作。接下来，我们将这条指令的实现加入到框架代码中。首先，要创建一个头文件，给出这条指令函数的声明。例如创建头文件 `nemu/include/cpu/instr/mov.h`，并在里面写入以下内容并保存：

```
#ifndef __INSTR_MOV_H__
#define __INSTR_MOV_H__

int mov_i2r_b(uint32_t eip, uint8_t opcode);

#endif
```

编辑 `nemu/include/cpu/instr.h` 头文件，在里面 include 上述头文件：

```
include "cpu/instr/mov.h"
```

在 `nemu/src/cpu/opcode.c` 源文件中已经 include 了 `cpu/instr.h` 头文件，现在只需在 `opcode_entry` 数组中对应的位置填入该函数的名称，就能够顺利将这个函数的实现加入到框架代码中去。具体而言，可以在 `opcode_entry` 数组中对应 0xB0 到 0xB7 的位置都填上这个函数的名称。在这个数组中，这些位置的元素就是对应这个函数的函数指针。等到 CPU 执行到对应的指令时，结合上一节讲的指令循环的相关内容，应该不难想象通过访问 `opcode_entry` 数组中 0xB0 到 0xB7 其中一个位置的元素，就会调用刚刚实现的 `mov_i2r_b` 函数，完成 mov 操作并使得 EIP 加 2 得到下一条指令的起始地址。

虽然上述代码很好地实现了这一条 `mov` 指令的功能，但是由于其对于操作数的寻址和操作都采取硬编码的方式实现，在构建其它 `mov` 指令的时候，很可能会出现实现一条指令写一套代码的现象。这种编程方式会导致大量的代码重复，并且使得代码凌乱不堪难以维护。因此，下面我们介绍框架代码中针对操作数和操作数寻址所提供的一系列封装函数，以期用比较统一的风范，来实现不同的指令。

§2-1.2.3 操作数和操作数寻址

在指令实现的过程中，一个核心的功能就是实现对操作数的寻址。在上面介绍的简单 `mov` 指令中，源操作数和目的操作数的寻址十分简单，但对于一些操作数寻址较为复杂的指令，书写起来就相对复杂了。为了方便这一功能的实现，NEMU 对操作数相关的代码进行了封装。本节就详细介绍框架代码中有关的内容。

OPERAND 数据结构及其读写操作封装

NEMU 中所有的操作数都封装在一个叫做 `OPERAND` 的数据结构中。该数据结构的声明在头文件 `nemu/include/cpu/operand.h` 中。仔细阅读该头文件，理解其中每个字段的含义，具体内容本教程不再赘述。对应 `OPERAND`，有两个主要的方法来封装操作数的读写操作，分别为：

```
void operand_read(OPERAND *opr);
void operand_write(OPERAND *opr);
```

阅读代码的注释，可知这两个函数的作用分别是将地址为 `opr->addr` 的操作数的值从读到 `opr->val` 中，以及将 `opr->val` 写到地址 `opr->addr` 里。这两个函数对应的实现在源文件 `src/cpu/operand.c` 中，通过阅读代码，可以理解操作数的读写过程会根据操作数的类型 (`opr->type`) 来决定对应的寻址方式，并根据操作数的长度 (`opr->data_size`) 来决定最后读写的位数。下表总结了 `OPERAND` 字段所代表的不同地址表示方式。

opr->addr 类型	与 opr->addr 构成的地址表示方式
OPR_MEM	opr->addr 中保存的是一个内存地址（位于数据段或栈段）
OPR_IMM	opr->addr 中保存的是一个立即数的内存地址（位于代码段）
OPR_REG	opr->addr 中保存的是一个通用寄存器的编号，随着 opr->data_size 的不同，其对应关系略有不同。参照 <code>nemu/include/cpu/reg.h</code> 中定义的 <code>enum</code> 数据结构。
OPR_SREG	opr->addr 中保存的是一个段寄存器的编号，参照 <code>nemu/include/cpu/reg.h</code> 中定义的 <code>enum</code> 数据结构。
OPR_CREG	opr->addr 中保存的是一个控制寄存器的编号 <code>CRx</code> ， <code>opr->addr = x</code> 。

利用 `OPERAND` 数据结构，我们可以重构上述 `mov_i2r_b` 函数的实现如下：

```
int mov_i2r_b(uint32_t eip, uint8_t opcode) {
    OPERAND imm, r;      // 创建源操作数和目的操作数局部变量
```

```

    imm.type = OPR_IMM;    // 配置源操作数类型
    imm.addr = eip + 1;    // 配置源操作数地址
    imm.data_size = 8;     // 配置源操作数长度

    r.data_size = 8;       // 配置目的操作数类型
    r.type = OPR_REG;      // 配置目的操作数类型
    r.addr = opcode & 0x7; // 配置目的操作数类型

    operand_read(&imm);    // 读源操作数的值
    r.val = imm.val;        // 将源操作数的值赋给目的操作数
    operand_write(&r);     // 写入目的操作数，完成 mov 动作

    return 2;              // 返回指令长度
}

```

一眼看去这一段代码似乎比先前的实现复杂多了。但是请注意，我们这里所给出的例子是一个非常简单的 `mov` 指令的实现。马上我们就会了解到，在 IA32 体系结构中，即便是 `mov` 指令，也存在着多种多样的形式，其操作数的类型可能是立即数、内存地址或寄存器；其操作数长度可能是 8 位 (byte)、16 位 (word)、32 位 (double word)。若不使用 `OPERAND` 对操作数结构和读写操作封装，对于上述每一种情况，可能都要结合具体情况书写详细的寻址和操作过程。而使用 `OPERAND` 封装，则每次只需要修改 `OPERAND` 所对应的 `type` 和 `data_size`，再赋予正确的 `addr`，就能够在很少的代码改动下，以一种规整的代码风格，完成对不同类型 `mov` 的实现了。这种对需要反复调用的功能的封装和抽象，可以有效地提高书写代码的效率，并降低错误的发生。但是，正如我们后续要谈到的，这只是我们抽象代码的第一层，为了使得我们的代码更简洁，框架代码设计了一套简单的机制来帮助我们用尽可能少的代码来书写尽可能多的功能。

接下来，我们把代码功能抽象的事情先放一放，仍然回到操作数寻址的问题上来。

ModR/M 字节的解析

在上述简单 `mov` 指令的例子中，指令所涉及的操作数类型十分单纯，寻址方式也非常简单。正如我们在预备知识一节中所看到的那样，在 IA32 所包含的众多指令的操作数寻址过程中，涉及到很多更为复杂的内存和寄存器寻址方式。这些复杂的寻址方式大量使用到对 `ModR/M` 和 `SIB` 字节的解析。在本节和接下来的一节中，我们具体阐述框架代码对于 `ModR/M` 字节和 `SIB` 字节的解析方式。

通过理论课的学习，结合在预备知识中介绍的例子，我们已经掌握了通过查阅 i386 手册的方式来解析 `ModR/M` 字节的方法。结合 `OPERAND` 的经验阐述，我们显然不会愿意在每一次需要 `ModR/M` 解析功能时，把详细的代码在指令的实现函数中写一遍。因此，框架代码对这部分功能也进行了封装。

在头文件 `nemu/include/cpu/modrm.h` 中，声明了 `ModR/M` 字节的结构定义和四个函数：

```

int modrm_rm(uint32_t eip, OPERAND * rm);
int modrm_r_rm(uint32_t eip, OPERAND * r, OPERAND * rm);
int modrm_opcode_rm(uint32_t eip, uint8_t * opcode, OPERAND * rm);
int modrm_opcode(uint32_t eip, uint8_t * opcode);

```

这四个函数分别对应指令希望通过解析 ModR/M 字节所获得的数据的四中不同类型组合，已涵盖实验中所涉及的所有指令。当指令的 opcode 确定时，就已经能够通过查阅手册确定：1) 是否需要解析 ModR/M 字节；2) 通过解析 ModR/M 字节应当获得什么操作数类型的信息。因此，在指令对应的函数实现中，根据需要调用上述四个函数中的一个即可。以上四个函数的返回类型都为整型，其含义是为了完整解析 ModR/M 字节所包含的信息所一共扫描的字节个数，包含 ModR/M 字节本身，也包含后续可能出现的 SIB 字节和 displacement 字节。每个函数的第一个参数都为 eip，其含义是 ModR/M 字节在内存中的地址，若当前 eip 指向 opcode，则 ModR/M 字节的地址为 eip + 1。

上述四个函数对应的实现在源文件 `src/cpu/decode/modrm.c` 中，其实现方法是对手册内容的直白翻译，相信不难理解。值得注意的是，这四个函数只负责对操作数的地址和类型进行填写，不会帮助我们完成操作数值的读写，因此在调用过后，不要忘记调用 `operand_read` 或 `operand_write` 函数。

ModR/M 字节的解析方式，在 i386 手册中有详细的说明，请参阅手册中第 17.2.1 节中的描述并结合 Table 17-3 的内容进行理解。请关注教程最后针对 Table 17-3 的勘误。

SIB 字节的解析

通过查阅 i386 手册第 17.2.1 节 Table 17-3，可以发现当 ModR/M 字节的 Mod 部分取 00、01、或 10 且其 R/M 部分取 100 时，ModR/M 字节后跟 SIB 字节。此时指令操作数的寻址方式为基址+比例变址+位移。

在框架代码头文件 `include/cpu/sib.h` 中，声明了 SIB 字节的结构定义和一个函数：

```
int parse_sib(uint32_t eip, uint32_t mod, OPERAND *opr);
```

该函数的功能就是根据 SIB 字节的内容，对操作数的地址进行解析。其返回类型为整型，含义是为了完整解析 SIB 字节所包含的信息所一共扫描的字节个数，包含 SIB 字节本身，也包含后续可能出现的 displacement 字节。其第一个参数为 SIB 字节在内存中的地址，若当前 eip 指向 ModR/M 字节，则 SIB 字节的地址为 eip + 1。第二个参数 mod 为 ModR/M 字节中的 mod 部分。最后一个参数则指向需要确定地址的操作数。其对应的实现在源文件 `nemu/src/cpu/decode/sib.c` 中。parse_sib() 函数不会在实现指令时被显式调用，而会在解析 ModR/M 的过程中根据需要调用。

SIB 字节的解析方式请参阅手册中第 17.2.1 节中的描述并结合 Table 17-4 的内容进行理解。请关注教程最后针对 Table 17-4 的勘误。

复杂寻址方式下 mov 指令的实现

在充分了解了框架代码中针对操作数的寻址和读写操作等接口的基础上，我们已经准备好实现复杂寻址方式下的指令的实现了。假设某指令的机器码（16 进制）如下：

```
C7 05 48 11 10 00 02 00 00 00
```

通过查阅 i386 手册，我们知道这条指令译码结果对应 AT&T 汇编语句：

```
movl $0x02 0x101148
```

所执行的操作为将一个 16 位或 32 位的立即数 mov 到一个由 ModR/M 字节所表达的 16 位或 32 位内存地址或寄存器 (r/m) 中。通过使用上面提到框架代码对操作数寻址和操作所提供的一系列函数封装，我们给出这一条指令的实现方案如下：

```

// 宏展开后这一行即为 int mov_i2rm_v(uint32_t eip, uint8_t opcode) {
make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;

    rm.data_size = data_size;
    int len = 1;                // opcode 长度 1 字节
    len += modrm_rm(eip + 1, &rm); // 读 ModR/M 字节, rm 的 type 和 addr 会被填写

    imm.type = OPR_IMM;        // 填入立即数类型
    imm.addr = eip + len;      // 找到立即数的地址
    imm.data_size = data_size; // 右侧的 data_size 为一个 uint8_t 类型全局变量

    operand_read(&imm);        // 执行 mov 操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8; // opcode 长度 + ModR/M 字节扫描长度 + 立即数长度
}

```

参照上文中对简单 `mov` 指令的实现方案，不难理解这一段代码的含义。遵循在 `include/cpu/instr/mov.h` 中声明函数，并在 `opcode_entry` 数组中添加函数指针的方法，也不难将这条指令添加到 NEMU 的指令集中去。在这里，我们就这段代码中的两个要点进行简单说明：

首先，`make_instr_func` 是一个宏，定义在 `include/cpu/instr_helper.h` 头文件中。其作用仅仅是将实现指令的函数名扩展成符合框架代码规范的指令函数的声明格式。可以在实现指令的过程中广泛使用，有效缩减编码长度，增加代码可理解性。

第二，代码块中特别注明的 `data_size` 是一个全局变量，其定义在 `src/cpu/instr/data_size.c` 源文件中，配合该源文件中的 `data_size_16()` 函数使用，用于在指令操作码前出现 `0x66` 前缀时，将操作数长度临时修改为 16 位。配合 `opcode_entry` 数组中第 `0x66` 号元素的函数指针（指向 `data_size_16()` 函数），不难理解其工作方式。

通过以上的讲解，我们对于利用框架代码中所提供的一系列函数来完成对各个指令的模拟已经有了一个可行的方案。参照上面的代码，实现诸如 `mov_i2rm_b`、`mov_r2rm_b`、`mov_r2rm_v` 等指令对应的函数，并将其添加到 `opcode_entry` 数组中已不在话下。事实上，在框架代码第一遍构建的时候，正是采用上述方法完成了对绝大部分指令的模拟。即使是在采用了我们接下来要谈到的抽象措施之后，对于一些不方便抽象的函数，我们任然保留了这种实现方法。因此，充分理解上述对操作数寻址和读写操作的封装，对于理解并运用框架代码实现指令，具有重要意义。

接下来，我们针对上述构建方法在代码简洁性方面的不足展开讨论，介绍框架代码所提供的更为高级的函数抽象功能。

§2-1.2.4 利用框架代码实践 KISS 原则

KISS（Keep It Simple and Stupid）原则，是在工程实践中的一条重要准则。再厉害的工程师，在面对复杂的代码时，恐怕都会觉得头疼不已。在我们的这个项目中，KISS 原则也是需要坚守的。否则不用说刚刚开始接触工程的新手们，即使是拥有多年经验的老司机也难免会翻车。下面，我们结合框架代码来具体阐述一下。

正如上一节提到的，通过运用框架代码中对操作数寻址和读写的封装，我们可以以一种统一的风范，编写所有的指令。例如，上一节提到的 `mov` 指令，针对它的多种变型，如 `mov_i2rm_b`、`mov_i2rm_v`、`mov_r2rm_b`、`mov_r2rm_v`，我们不难写出如下的代码：

```
make_instr_func(mov_i2rm_b) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = 8; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov 操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + 1;
}

make_instr_func(mov_i2rm_v) {
    OPERAND rm, imm;

    imm.data_size = rm.data_size = data_size; // 指定操作数长度

    int len = 1;
    len += modrm_rm(eip + 1, &rm); // 操作数寻址
    imm.type = OPR_IMM;
    imm.addr = eip + len;

    operand_read(&imm); // mov 操作
    rm.val = imm.val;
    operand_write(&rm);

    return len + data_size / 8;
}

make_instr_func(mov_r2rm_v) {
    OPERAND r, rm;

    rm.data_size = r.data_size = data_size; // 指定操作数长度

    int len = 1;
    len += modrm_r_rm(eip + 1, &r, &rm); // 操作数寻址

    operand_read(&r); // mov 操作
    rm.val = r.val;
    operand_write(&rm);

    return len;
}

make_instr_func(mov_rm2r_b) {
```

```

OPERAND r, rm;

r.data_size = rm.data_size = 8; // 指定操作数长度

int len = 1;
len += modrm_rm(eip + 1, &r, &rm); // 操作数寻址

operand_read(&rm); // mov 操作
r.val = rm.val;
operand_write(&r);

return len;
}

```

按照这个套路写下去，我们可以很顺利地实现所有指令的模拟函数。但是，通过观察上面的代码，我们发现每个函数长得都差不多。每个函数所做的事情都是配置操作数，执行 `mov` 操作，返回长度。除了操作数类型和操作数长度有所不同，`mov` 操作的过程完全就是一模一样，返回的长度也不过是结合操作数长度做相应的调整。

在 IA32 的指令集中，这种情况并不少见。观察 i386 手册 Appendix A 中的 One-Byte Opcode Map 的前三行，我们发现 `add`、`or`、`adc`、`sbb` 等许多指令都包含操作重复只有操作数类型和长度变化的情形。以加法和减法指令为例，如果按照上述风范写代码，那就会是这样的一个过程：1) 先在源文件 `add.c` 里写一条指令的函数比如 `add_r2rm_b`；2) 不断拷贝这一段代码，做少量修改后得到诸如 `add_r2rm_v`、`add_rm2r_b`、`add_rm2r_v` 等一系列指令的实现；3) 把 `add.c` 整个拷贝一个副本并重命名为 `sub.c`，把 `sub.c` 里所有的函数名字前缀都改成 `sub`，并且把操作从加法变到减法。重复上述第三步动作，很快我们就能得到 `or`、`adc`、`sbb` 等指令的实现。

这种大量代码克隆的方法虽然挺快也挺容易理解的，但会导致我们工程的代码非常臃肿。更严重的是，如果等我们都拷贝完了，发现最初的 `add_r2rm_b` 函数中有一个小小的 bug，岂不是要把这个过程再重复一遍？可见此法 Stupid 有余而 Simple 不足，不符合 KISS 原则。为此框架代码提供更为高层的抽象方式，以 KISS 原则为指导，力图在不降低可理解性的前提下，使得我们的代码尽可能地简洁。接下来我们介绍框架代码为精简代码所提供的一系列宏。

用于精简指令实现的宏

C 语言中的宏本质上就是字符串替换，在预处理阶段就被处理。我们可以通过使用宏来有效精简代码并增加代码的可读性。框架代码通过构建一系列的宏定义来精简指令实现。这些宏的定义都可以在头文件 `include/cpu/instr_helper.h` 中找到。在这里，我们就这些宏设计背后的思考过程进行讲解。

首先，观察上一节中的指令函数实现，我们发现，对于 `mov` 指令，其操作数都可以分为源操作数和目的操作数两个。进一步观察 IA32 指令集体系结构中的指令，我们发现，大多数指令也都包含两个操作数，如 `add`、`or` 等等。对于实现指令的函数而言，源操作数和目的操作数起什么名字并不重要。同时，由于 NEMU 模拟器是单线程的，不可能出现指令并发执行的情况。因此，我们可以把源操作数和目的操作数对应的 `OPERAND` 从局部变量变为全局变量。这样可以节省栈空间，同时规范源操作数和目的操作数的变量名称，方便我们进一步抽象。在框架代码的 `nemu/src/cpu/decode/operand.c` 源文件中，我们定义了两个 `OPERAND` 类型的全局变量，`opr_src` 和 `opr_dest`，用于表示源操作数和目的操作数，为所有的指令实现函数共享。

第二，使用统一命名的全局变量代替局部变量后，观察上一节的 `mov` 指令实现。不难看出，注释为“`mov` 操

作”的那三行代码，会在所有的函数中变成一模一样的代码，即：

```
operand_read(&opr_src); // mov 操作
opr_dest.val = opr_src.val;
operand_write(&opr_dest);
```

既然这三行代码在所有 `mov` 指令的实现函数中都一样，那不妨将其提取出来，抽象成一个执行函数好了。假定我们定义执行函数名为 `instr_execute_2op`，那么针对 `mov` 指令的执行函数就可以写成如下：

```
static void instr_execute_2op() {
    operand_read(&opr_src);
    opr_dest.val = opr_src.val;
    operand_write(&opr_dest);
}
```

其中的 `static` 关键字是为了将 `instr_execute_2op()` 函数的作用域限制在该 `c` 文件中，以便我们能够在别的指令对应的 `c` 文件中复用该函数名。于是，所有 `mov` 指令的实现函数中，对应 `mov` 操作的部分都可以变为对 `instr_execute_2op()` 函数的调用了。如此一来，不同 `mov` 指令的实现函数越来越相似了。

第三，继续观察上一节中 `mov` 指令的实现函数。不难发现，每个指令的实现都遵循同样的套路：

1. 指定源操作数和目的操作数的长度。该长度可以从指令名称中的操作数长度后缀来确定，`b` 就是 8 位；`v` 就是 `data_size`，表示由 `0x66` 前缀确定的位数；类似地还可以定义 `w` 和 `l` 后缀，分别代表 16 位和 32 位。
2. 解码操作数地址。源操作数和目的操作数的类型也可以从指令名称中间的操作数类型部分来确定，如，`rm2r` 就是 `rm` 类型的源操作数 `mov` 到 `r` 类型的目的操作数。所有指令中的操作数类型有限，每种操作数解码地址的方法也都固定，比如遇到 `rm2r` 或 `r2rm` 类型，那必然使用 `modrm_r_rm()` 函数去解码地址；
3. 执行指令的数据操作。比如 `mov` 指令就是做数据转移，`add` 指令就是做加法等；
4. 返回指令长度。即指令 `opcode` 本身 1 字节，再加上操作数地址解码过程中扫描过的字节数。

在第二点中，我们已经对上述第 3 步要做的操作封装了一个叫做 `instr_execute_2op()` 的函数。延续这个思路，我们对第 1 点中要做的工作也封装成一系列 `decode_data_size` 函数，如，`decode_data_size_b`、`decode_data_size_v` 等等，在这些函数中对全局操作数变量 `opr_src` 和 `opr_dest` 赋予相应的以比特计的操作数长度。在框架代码中，我们用宏代替函数，定义了一系列的 `decode_data_size` 宏，使用宏的好处是减少函数调用次数，提高程序性能。

第四，正如套路中的第 2 步所述，指令的源操作数和目的操作数类型一给出，我们就知道用什么办法去解码操作数地址。既然操作数类型组合有限，那不妨也来封装成函数。这里的思路和 `instr_execute_2op()` 与 `decode_data_size` 的设计一样，封装成一系列的 `decode_operand` 操作。具体过程就不赘述了，查阅头文件 `nemu/include/cpu/instr_helper.h`，稍加思考应该能明白。

综上所述，我们实现一条（双目）指令所需要的所有信息包括：指令名称（`inst_name`）、源操作数类型（`src_type`）、目的操作数类型（`dest_type`）、操作数长度后缀（`suffix`）。除此以外，我们约定实现（双目）

指令操作的函数统一命名为 `static void instr_execute_2op()`。于是我们可以讲上述套路变成代码了：

1. 实现指令的函数名称就命名为 `inst_name_src_type2dest_type_suffix`
2. 指定返回长度 `len = 1`，意思是 opcode 占一字节
3. 指定操作数长度 `decode_data_size_suffix`
4. 操作数地址解码 `decode_operand_src_type2dest_type`，在解码过程中，顺便把 `len` 加上解码过程中扫描的字节个数
5. 调用指令操作函数 `instr_execute_2op()`
6. 返回 `len`

通过以上讲解，应当能读懂双目指令对应的宏 `make_instr_impl_2op` 设计：

```
#define make_instr_impl_2op(inst_name, src_type, dest_type, suffix) \
    make_instr_func(concat7(inst_name, _, src_type, 2, dest_type, _, suffix)) {\
        int len = 1; \
        concat(decode_data_size_, suffix) \
        concat3(decode_operand, _, concat3(src_type, 2, dest_type)) \
        instr_execute_2op(); \
        return len; \
    }
```

其中的一系列 `concat` 宏就是简单的把几个字符串组合成一个字符串。

除此以外，框架代码在头文件 `include/cpu/instr_helper.h` 中还设计了针对单目指令的宏和针对条件执行的宏，请阅读相关代码并进行理解。

最后，通过使用框架代码中的宏，上一节四条 `mov` 指令的实现就可以写成：

```
static void instr_execute_2op() { // 所有 mov 指令共享的执行方法
    operand_read(&opr_src);
    opr_dest.val = opr_src.val;
    operand_write(&opr_dest);
}

make_instr_impl_2op(mov, i, rm, b)
make_instr_impl_2op(mov, i, rm, v)
make_instr_impl_2op(mov, r, rm, v)
make_instr_impl_2op(mov, rm, r, v)
```

对比上一节的实现和这一节中的实现方法，可见：1) 后者通过宏展开能够得到几乎一样的代码（唯一区别就是操作数 `OPERAND` 从局部变量变成全局变量并改换名称）；2) 后者的实现没有冗余的代码克隆，非常简洁，实现过程统一，犯错机会较少。

我们鼓励大家在充分理解框架代码的基础上，尽可能地用宏所提供的设施，来高效地实现指令函数。当然这套宏的设计也不是万能的，我们没有针对三个操作数的指令给出框架，同时也这些宏不适用于一些内部逻辑非常复杂的指令，如 `call` 指令等。更进一步地，每个人对于代码要如何抽象，如何实践 KISS 原则或许都有自己的见解，我们鼓励大家大胆尝试。但有一点提醒大家，既然是学习，那就要扎实地学，如果无法理解这一套宏，那宁肯先不用也不要依样画葫芦就稀里糊涂过了。不妨以最笨的方法写它十几二十条指令再回过头来看，可能体会更深刻。

根据框架代码的构筑经验，适用和不适用宏的指令分别是：

- 适用宏的指令：`adc, add, and, bt, cbw, cmov, cmp, dec, inc, jcc`，大多数的 `mov, not, or, pop, push, sar, sbb, setcc, shl, shr, sub, test, xor`
- 不适用的指令：`call, cld, cmps, div, idiv, mul, imul, cld, clc, sahf, hlt, int, jmp, lea, leave, rep, ret, stos, x87`

特别说明：在框架代码的 `nemu/src/cpu/instr/idiv.c` 中提供的针对 `idiv` 指令的实现，有一个小的疏漏。所有调用 `alu_imod()` 函数时第一个参数使用了 `sign_ext()` 进行扩展，需要修改成 `sign_ext_64()`。如果你 clone 下来的代码已经修改过了，则可以不用理会这个问题。

使用 `print_asm()` 函数输出指令信息

在 `nemu/include/cpu/instr_helper.h` 中定义了四个用于输出指令信息的 `print_asm()` 函数，函数名最后的数字指出需要打印几个操作数。如使用框架代码中预定义的宏来实现指令，那么已经由框架代码负责调用合适的 `print_asm()` 函数。若不使用框架代码中的宏，那么在指令实现函数中，调用合适的 `print_asm()` 函数即可。注意遇到跳转指令，`print_asm()` 函数要在改变 `eip` 之前调用。`print_asm()` 函数中的 `len` 参数用于指出指令的长度，若不知道该怎么设置，可以暂时设置为一个较大的值。

有的时候我们希望能将执行过程中的输出同时打印到一个 `log` 文件中以方便慢慢查看。这个时候可以采用以下命令：

```
make run | tee log.txt
```

具体的含义请自行搜索。

§2-1.3 实验过程及要求

在这一阶段要实现较多的指令，需要通过除了 `hello-inline` 和 `echo` 以外的所有测试用例，基本的步骤为：



1. 修改 `make run` 中的 `<testcase_name>`，指定要执行的测试用例。或使用 `make test` 自动执行各个测试用例；
2. 保证编译通过并开始执行测试用例；
3. 若遇到 `invalid opcode` 错误，则使用 `objdump -d testcase/bin/<testcase_name>` 查看测试用例反汇编结果，看看到底是缺了哪条指令，查阅 `i386` 手册详细了解这条指令。实现这条指令并将其加入 `opcode_entry` 数组，替换对应位置上的 `inv` 指令（框架代码已经提供了一部分指令的实现，只是没有和 `opcode_entry` 连接起来，比如 `mov.S` 所需要的所有指令）；

重复上述过程，直至通过所有这一阶段要求的测试用例（见到传说中的 **Hit Good Trap**），建议按照 `main.c`

中的 `testcases` 数组中给定的顺序来执行测试用例。

注意：test-float 测试用例是唯一的一个例外，它理应 **Hit Bad Trap**，请在实验报告中简述为什么。

在实验报告中，回答下面的问题：

1. 使用 `hexdump` 命令查看测试用例的文件，所显示的文件的内容对应模拟内存的哪一个部分？指令在机器中表示的形式是什么？
2. 如果去掉 `instr_execute_2op()` 函数前面的 `static` 关键字会发生什么情况？为什么？
3. 为什么 test-float 会 fail？以后在写程序的时候要注意什么？

注意：push imm8 指令需要对立即数进行符号扩展，这一点在 i386 手册中没有说明，在 IA-32 手册中关于 push 指令有如下说明：

If the source operand is an immediate and its size is less than the operand size, a sign-extended value is pushed on the stack。

——感谢 15 级何知涵助教在审阅过程中给出的提醒！

PA 2-2 装载程序的 loader

在上一节中所展开的实现指令的实验中，测试用例的可执行目标文件是通过交叉编译并使用 objcopy 直接得到对应的内存镜像加载到内存中执行的。在真实的计算机系统中，这是一种效率极低的存储可执行文件的方法。现代类 UNIX 操作系统，如 Linux，主要使用可执行可链接格式 (Executable and Linkable Format, 简称 ELF) 来存储目标文件。本节我们就目标文件的装载展开讨论。

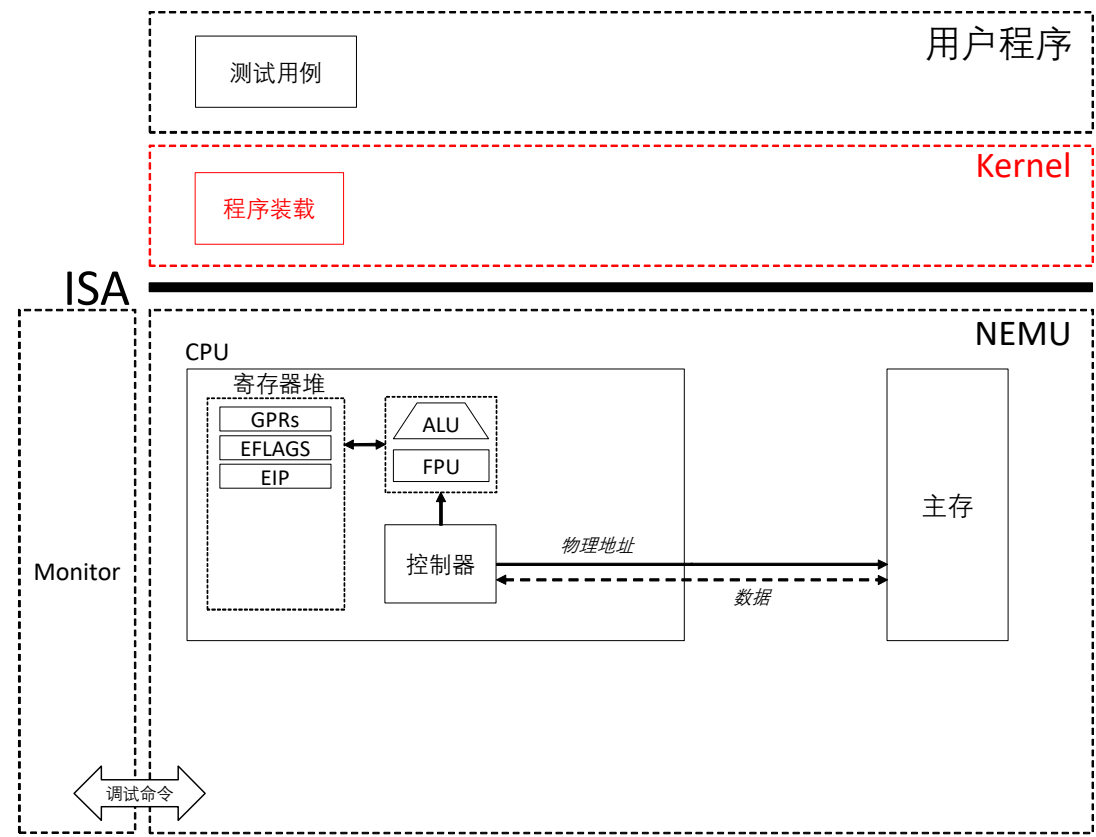


图 2-4 PA 2-2 路线图

§2-2.1 预备知识

§2-2.1.1 ELF 文件和 ELF Header

要实现 ELF 文件装载，需要先熟悉 ELF 可执行目标文件的结构。简单来说，对应课本 pg. 171，图 4.5 所示的结构，目前我们可以将 ELF 可执行目标文件看作由三个部分组成：ELF 头、程序头表、其余的 ELF 文件体。

ELF Header	Program Header Table	Rest of the ELF File
------------	----------------------	----------------------

图 2-5 ELF 文件结构简图

ELF 头的结构可以通过在控制台中执行 `man elf` 命令进行查阅，描述如下：

```
The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:

#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;        // 程序头表在 ELF 文件中的偏移量
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;    // 程序头表中每个表项的大小
    uint16_t      e_phnum;        // 程序头表中包含表项的个数
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

其中和可执行文件装载相关的三个成员已经给出注释了。事实上，我们只需要使用其中的两个，即，`e_phoff` 和 `e_phnum` 就能够顺利地实现装载。

§2-2.1.1 ELF 文件的装载

装载的过程简言之就是将 ELF 文件中的程序和数据段等需要装载到内存中的 segment 拷贝到内存中合适位置的过程。在 ELF 文件中存储了一个数组，叫做程序头表（program header table），其在 ELF 文件中偏移量由 ELF Header 中的 `e_phoff` 域给出。程序头表中每一项的结构可以通过 `man elf` 命令进行查看，摘录如下：

```
typedef struct {
    uint32_t      p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    uint32_t      p_filesz;
    uint32_t      p_memsz;
    uint32_t      p_flags;
    uint32_t      p_align;
} Elf32_Phdr;
```

其中，`p_type` 指定了表项的类型，对于类型为 `PT_LOAD` 类型的表项，我们需要对其进行装载。装载过程可以简述为，对于 `p_type == PT_LOAD` 的表项，将 ELF 文件中起始于 `p_offset`，大小为 `p_filesz` 字节的数据拷贝到内存中起始于 `p_vaddr` 的位置，并将内存中剩余的 `p_memsz - p_filesz` 字节的内容清零。其过程可以由下图来表示：

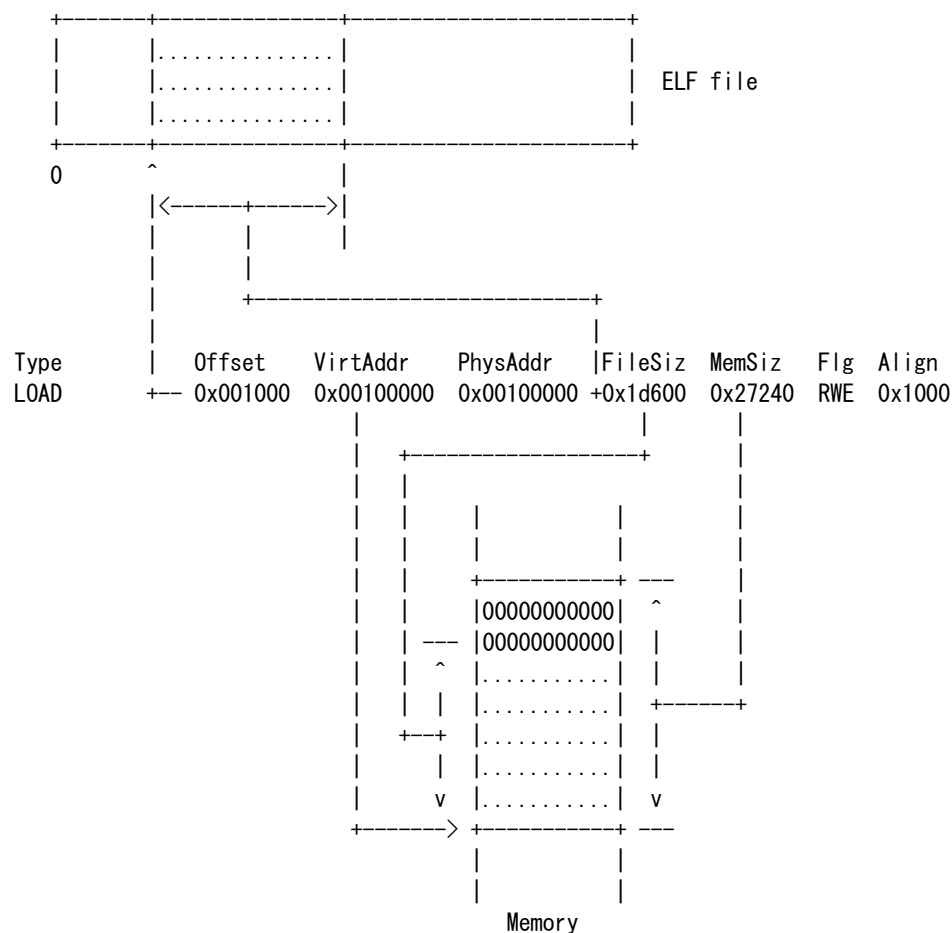


图 2-6 ELF 文件的装载

为了方便地理解一个可执行文件的程序头表的内容，可以通过 `readelf` 命令查看 ELF 文件的内容，`readelf` 提供了两个视角，一个是面向链接过程的 `section` 视角 (`readelf -S`)，另一个是面向执行的 `segment` 视角 (`readelf -l`)。在这里我们关注后一个视角即可。通过对比 `readelf` 所打印出的程序头表和 `Elf32_Phdr` 所示的程序头表表项结构，不难理解其中的含义。所谓装载的过程即为扫描程序头表，对所有类型为 `PT_LOAD` 的表项执行上图中所示的装载过程。

总结预备知识中的内容，ELF 文件装载的过程如下：

1. 读入位于 ELF 文件最开始位置（偏移量为 0）处的 ELF 头，并根据其中的值 `e_phoff` 定位程序头表在 ELF 文件中的位置；
2. 顺序扫描程序头表中的每一个表项，遇到需要装载的表项时，根据表项描述的内容将相应的数据拷贝到内存相应位置。

§2-2.2 代码导读和实验理解

§2-2.2.1 引入 Kernel

可执行目标文件的格式是由操作系统定义的。自然地，在这里我们也将这一功能实现在一个简单地操作系统之中。从这一节开始，我们引入一个非常精简的操作系统——Kernel，来实现和操作系统有关的系统功能的实现。Kernel 的代码位于和 NEMU 平行的目录中名为 kernel 的文件夹之下。

我们的实验主要围绕 NEMU 展开，核心是系统的功能模拟。引入 Kernel 主要是为了完成以下三个方面的功能：

1. 实现可执行目标文件（ELF 格式）的装载
2. 实现存储管理对段表和页表的初始化
3. 实现中断处理和 I/O 相关的操作

以上三点功能正好贯穿从 PA2 后期到 PA4 的所有内容。在本实验中，我们关心的是第一点功能。

§2-2.2.2 使用 make testkernel 执行测试用例

在之前使用 `make run` 或 `make test` 执行测试用例时，NEMU 会直接将测试用例的镜像加载到内存 `0x30000` 开始的模拟内存中，并从第一条指令开始执行。此时的内存划分方式请参照 PA 2-1 中的论述。

在引入 Kernel 之后，我们通过命令 `make testkernel` 来加载 Kernel 进而执行测试用例。此时的内存区段划分与 `make run` 和 `make test` 时的情形非常类似，物理地址 `0x00000 ~ 0x30000` 区段存放的是测试用例的 ELF 文件（`testcase/bin/<testcase>`），而从物理地址 `0x30000` 开始存放的是 Kernel 的代码和数据：

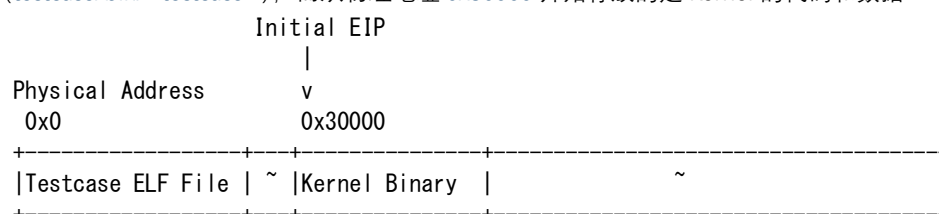


图 2-7 装入 Kernel 后的内存划分

而在本实验中，我们要实现 Kernel 中位于 `kernel/src/elf/elf.c` 中的 `loader()` 函数。我们约定此时测试用例的起始位置位于物理地址 `0x60000`。此约定隐含假设 Kernel 的代码和数据能够存放于物理地址 `0x30000 ~ 0x60000` 的区间。若空间不够，则需要将测试用例的起始位置设置为更靠后的值。此时内存的划分方和 `loader()` 的功能可由图 2-8 来描述。

为配合上述过程，需要修改 `testcase/Makefile` 中 `LDFLAGS` 的 `-Ttext` 参数，将其从 `0x30000` 修改为 `0x60000`。实现 `loader()` 的方案请参照预备知识中描述的过程。

在完成了程序的装载后，`loader()` 函数将返回所装载程序（测试用例）的入口地址，该地址由 ELF 头中的 `e_entry` 域给出。这个域的取值由我们在测试用例的 `Makefile` 中的链接参数给出，此时我们设置为 `0x60000`。当 `loader()` 返回后，在 `kernel/src/main.c` 中，我们会记录该入口地址，并在稍后跳转到该地址开始执行测试

用例。

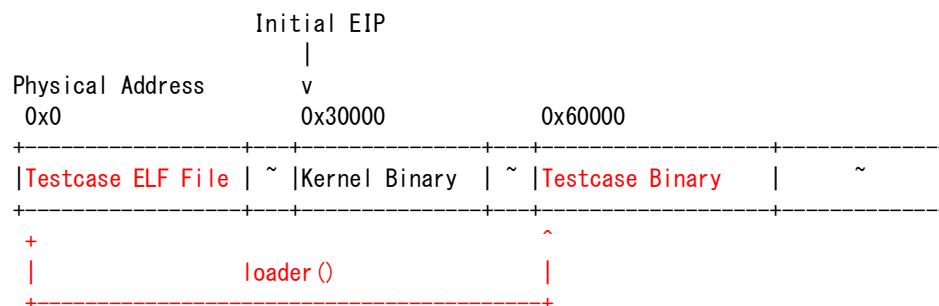


图 2-8 Kernel 装载测试用例的 ELF 文件

§2-2.3 实验过程及要求

1. 修改 `testcase/Makefile` 中 `LD_FLAGS` 并 `make clean` ;
2. 实现 Kernel 中的 `loader()` ;
3. 使用 `make testkernel` 执行测试用例并通过。

在实验报告中，回答如下问题：

1. 为什么在装载时要把内存中剩余的 `p_memsz - p_filesz` 字节的内容清零？

PA 2-3 可选任务：完善调试器

在 `nemu/src/monitor/ui.c` 中，框架代码提供了一系列用于帮助调试 NEMU 的调试和执行命令。目前这些命令所提供的功能都比较初级。在这一个可选任务中，我们考虑进一步完善调试器的功能。具体而言，我们试图完成以下两个功能：

1. 表达式求值；
2. 添加变量和函数名支持。

通过第一项任务，我们得以一窥编译器的设计原理；而第二项任务涉及到符号表的解析，属于 ELF 文件解析的一部分。为配合平行试验，这一节的教程内容组织方式按照问题来组织。

§2-3.1 表达式求值

§2-3.1.1 预备知识和代码导读

给你一个表达式的字符串

```
"5 + 4 * 3 / 2 - 1"
```

你如何求出它的值？表达式求值是一个很经典的问题，以至于有很多方法来解决它。我们在所需知识和难度两方面做了权衡，在这里使用如下方法来解决表达式求值的问题：

1. 首先识别出表达式中的单元
2. 根据表达式的归纳定义进行递归求值

词法分析

"词法分析"这个词看上去很高端，说白了就是做上面的第 1 件事情，"识别出表达式中的单元"。这里的"单元"是指有独立含义的子串，它们正式的称呼叫 `token`。具体地说，我们需要在上述表达式中识别出 `5`, `+`, `4`, `*`, `3`, `/`, `2`, `-`, `1` 这些 `token`。你可能会觉得这是一件很简单的事情，但考虑以下的表达式：

```
"0xc0100000+ ($eax +5)*4 - *( $ebp + 8) + number"
```

它包含更多的功能，例如十六进制整数(`0xc0100000`)，小括号，访问寄存器(`$eax`)，指针解引用(第二个 `*`)，访问变量(`number`)。事实上，这种复杂的表达式在调试过程中经常用到，而且你需要在空格数目不固定(0 个或多个)的情况下仍然能正确识别出其中的 `token`。当然你仍然可以手动进行处理(如果你喜欢挑战性的工作的话)，一种更方便快捷的做法是使用正则表达式。正则表达式可以很方便地匹配出一些复杂的 `pattern`，是程序员必须掌握的内容，如果你从来没有接触过正则表达式，请到查阅相关资料。在实验中，你只需要了解正则表达式的一些基本知识就可以了(例如元字符)。

学会使用简单的正则表达式之后，你就可以开始考虑如何利用正则表达式来识别出 `token` 了。我们先来处理一种简单的情况 -- 算术表达式，即待求值表达式中只允许出现以下的 `token` 类型：

1. 十进制整数
2. `+, -, *, /`
3. `(,)`
4. 空格串(一个或多个空格)

首先我们需要使用正则表达式分别编写用于识别这些 `token` 类型的规则。在框架代码中，一条规则是由正则表达式和 `token` 类型组成的二元组。框架代码中已经给出了 `+` 和空格串的规则，其中空格串的 `token` 类型是 `NOTYPE`，因为空格串并不参加求值过程，识别出来之后就可以将它们丢弃了；`+` 的 `token` 类型是 `'+'`，事实上 `token` 类型只是一个整数，只要保证不同的类型的 `token` 被编码成不同的整数就可以了；框架代码中还有一条用于识别双等号的规则，不过我们现在可以暂时忽略它。

这些规则会在 NEMU 初始化的时候被编译成一些用于进行 `pattern` 匹配的内部信息，这些内部信息是被库函数使用的，而且它们会被反复使用，但你不必关心它们如何组织。但如果正则表达式的编译不通过，NEMU 将会触发 `assertion fail`，此时你需要检查编写的规则是否符合正则表达式的语法。

给出一个待求值表达式，我们首先要识别出其中的 `token`，进行这项工作的是 `make_token()` 函数。`make_token()` 函数的工作方式十分直接，它用 `position` 变量来指示当前处理到的位置，并且按顺序尝试用不同的规则来匹配当前位置的字符串。当一条规则匹配成功，并且匹配出的子串正好是 `position` 所在位置的时候，我们就成功地识别出一个 `token`，`Log()` 宏会输出识别成功的信息。你需要做的是将识别出的 `token` 信息记录下来(一个例外是空格串)，我们使用 `Token` 结构体来记录 `token` 的信息：

```
typedef struct token {
    int type;
    char str[32];
} Token;
```

其中 `type` 成员用于记录 `token` 的类型。大部分 `token` 只要记录类型就可以了，例如 `+, -, *, /`，但对于有些 `token` 类型是不够的：如果我们只记录了一个十进制整数 `token` 的类型，在进行求值的时候我们还是不知道这个十进制整数是多少，这时我们应该将 `token` 相应的子串也记录下来，`str` 成员就是用来做这件事情的。需要注意的是，`str` 成员的长度是有限的，当你发现缓冲区将要溢出的时候，要进行相应的处理(思考一下，你会如何处理?)，否则将会造成难以理解的 bug。`tokens` 数组用于按顺序存放已经被识别出的 `token` 信息，`nr_token` 指示已经被识别出的 `token` 数目。

如果尝试了所有的规则都无法在当前位置识别出 `token`，识别将会失败，这通常是待求值表达式并不合法造成的，`make_token()` 函数将返回 `false`，表示词法分析失败。

作为表达式求值的第一步，你需要完成词法分析的功能，具体要求参见“实验要求”。

递归求值

把待求值表达式中的 `token` 都成功识别出来之后，接下来我们就可以进行求值了。需要注意的是，我们现在是在对 `tokens` 数组进行处理，为了方便叙述，我们称它为“`token 表达式`”。例如待求值表达式

`"4 + 3*(2 - 1)"`

的 `token 表达式`为

NUM	'+'	NUM	'*'	'('	NUM	'-'	NUM	')'
"4"		"3"			"2"		"1"	

图 2-9 tokens 数组举例

根据表达式的归纳定义特性，我们可以很方便地使用递归来进行求值。首先我们给出算术表达式的归纳定义：

```

<expr> ::= <number>           # 一个数是表达式
        | "(" <expr> ")"       # 在表达式两边加个括号也是表达式
        | <expr> "+" <expr>    # 两个表达式相加也是表达式
        | <expr> "-" <expr>    # 接下来你全懂了
        | <expr> "*" <expr>
        | <expr> "/" <expr>

```

上面这种表示方法就是大名鼎鼎的 BNF，任何一本正规的程序设计语言教程都会使用 BNF 来给出这种程序设计语言的语法。

根据上述 BNF 定义，一种解决方案已经逐渐成型了：既然长表达式是由短表达式构成的，我们就先对短表达式求值，然后再对长表达式求值。这种十分自然的解决方案就是分治法的应用，就算你没听过这个高大上的名词，也不难理解这种思路。而要实现这种解决方案，递归是你的不二选择。

为了在 token 表达式中指示一个子表达式，我们可以使用两个整数 *p* 和 *q* 来指示这个子表达式的开始位置和结束位置。这样我们就可以很容易把求值函数的框架写出来了：

```

eval(p, q) {
    if(p > q) {
        /* Bad expression */
    }
    else if(p == q) {
        /* Single token.
         * For now this token should be a number.
         * Return the value of the number.
         */
    }
    else if(check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of parentheses.
         * If that is the case, just throw away the parentheses.
         */
        return eval(p + 1, q - 1);
    }
    else {
        /* We should do more things here. */
    }
}

```

其中 `check_parentheses()` 函数用于判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配，如果不匹配，这个表达式肯定是不符合语法的，也就不需要继续进行求值了。我们举一些例子来说明 `check_parentheses()` 函数的功能：

```

"(2 - 1)"           // true
"(4 + 3 * (2 - 1))" // true
"4 + 3 * (2 - 1)"    // false, the whole expression is not surrounded by a matched pair of parentheses
"(4 + 3) * ((2 - 1))" // false, bad expression
"(4 + 3) * (2 - 1)"  // false, the leftmost '(' and the rightmost ')' are not matched

```

至于怎么检查左右括号是否匹配, 就留给聪明的你来思考吧!

上面的框架已经考虑了 BNF 中算术表达式的开头两种定义, 接下来我们来考虑剩下的情况(即上述伪代码中最后一个 `else` 中的内容). 一个问题是, 给出一个最左边和最右边不同时是括号的长表达式, 我们要怎么正确地把它分裂成两个子表达式? 我们定义 **dominant operator** 为表达式人工求值时, 最后一步进行运行的运算符, 它指示了表达式的类型(例如当最后一步是减法运算时, 表达式本质上是一个减法表达式). 要正确地对一个长表达式进行分裂, 就是要找到它的 **dominant operator**. 我们继续使用上面的例子来探讨这个问题:

```

"4 + 3 * ( 2 - 1 )"
/*****/
case 1:
    "+"
    /  \
    "4"  "3 * ( 2 - 1 )"

case 2:
    "*"
    /  \
    "4 + 3" "( 2 - 1 )"

case 3:
    "-"
    /  \
    "4 + 3 * ( 2" "1 )"

```

图 2-10 dominant operator 举例

上面列出了 3 种可能的分裂, 注意到我们不可能在非运算符的 **token** 处进行分裂, 否则分裂得到的结果均不是合法的表达式. 根据 **dominant operator** 的定义, 我们很容易发现, 只有第一种分裂才是正确的, 这其实也符合我们人工求值的过程: 先算 4 和 $3 * (2 - 1)$, 最后把它们的结果相加. 第二种分裂违反了算术运算的优先级, 它会导致加法比乘法更早进行. 第三种分裂破坏了括号的平衡, 分裂得到的结果均不是合法的表达式.

通过上面这个简单的例子, 我们就可以总结出如何在一个 **token 表达式** 中寻找 **dominant operator** 了:

1. 非运算符的 **token** 不是 **dominant operator**.
2. 出现在一对括号中的 **token** 不是 **dominant operator**. 注意到这里不会出现有括号包围整个表达式的情况, 因为这种情况已经在 `check_parentheses()` 相应的 `if` 块中被处理了.
3. **dominant operator** 的优先级在表达式中是最低的. 这是因为 **dominant operator** 是最后一步才进行的运算符.
4. 当有多个运算符的优先级都是最低时, 根据结合性, 最后被结合的运算符才是 **dominant operator**. 一个例子是 $1 + 2 + 3$, 它的 **dominant operator** 应该是右边的 `+`.

要找出 dominant operator, 只需要将 token 表达式全部扫描一遍, 就可以按照上述方法唯一确定 dominant operator.

找到了正确的 dominant operator 之后, 事情就变得很简单了, 先对分裂出来的两个子表达式进行递归求值, 然后再根据 dominant operator 的类型对两个子表达式的值进行运算即可. 于是完整的求值函数如下:

```
eval(p, q) {
    if(p > q) {
        /* Bad expression */
    }
    else if(p == q) {
        /* Single token.
         * For now this token should be a number.
         * Return the value of the number.
         */
    }
    else if(check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of parentheses.
         * If that is the case, just throw away the parentheses.
         */
        return eval(p + 1, q - 1);
    }
    else {
        op = the position of dominant operator in the token expression;
        val1 = eval(p, op - 1);
        val2 = eval(op + 1, q);
        switch(op_type) {
            case '+': return val1 + val2;
            case '-': /* ... */
            case '*': /* ... */
            case '/': /* ... */
            default: assert(0);
        }
    }
}
```

由于 ICS 不是算法课, 我们已经把递归求值的思路和框架都列出来了, 你需要做的是理解这一思路, 然后在框架中填充相应的内容. 实现表达式求值的功能之后, p 命令也就不难实现了.

需要注意的是, 上述框架中并没有进行错误处理, 在求值过程中发现表达式不合法的时候, 应该给上层函数返回一个表示出错的标识, 告诉上层函数“求值的结果是无效的”. 例如在 check_parentheses() 函数中, $(4 + 3) * ((2 - 1)$ 和 $(4 + 3) * (2 - 1)$ 这两个表达式虽然都返回 false, 因为前一种情况是表达式不合法, 是没有办法成功进行求值的; 而后一种情况是一个合法的表达式, 是可以成功求值的, 只不过它的形式不属于 BNF 中的 “(<expr>)”, 需要使用 dominant operator 的方式进行处理, 因此你还需要想办法把它们区别开来.

当然, 你也可以在发现非法表达式的时候使用 assert(0) 终止程序, 不过这样的话, 你在使用表达式求值功能的时候就要十分谨慎了.

调试中的表达式求值

实现了算术表达式的求值之后，你可以很容易把功能扩展到复杂的表达式。我们用 BNF 来说明需要扩展哪些功能：

```
<expr> ::= <decimal-number>
          | <hexadecimal-number>      # 以"0x"开头
          | <reg_name>                  # 以"$"开头
          | "(" <expr> ")"
          | <expr> "+" <expr>
          | <expr> "-" <expr>
          | <expr> "*" <expr>
          | <expr> "/" <expr>
          | <expr> "==" <expr>
          | <expr> "!=" <expr>
          | <expr> "&&" <expr>
          | <expr> "||" <expr>
          | "!" <expr>
          | "*" <expr>                  # 指针解引用
```

它们的功能和 C 语言中运算符的功能是一致的，包括优先级和结合性，如有疑问，请查阅相关资料。需要注意的是指针解引用(dereference)的识别，在进行词法分析的时候，我们其实没有办法把乘法和指针解引用区别开来，因为它们都是 `*`。在进行递归求值之前，我们需要将它们区别开来，否则如果将指针解引用当成乘法来处理的话，求值过程将会认为表达式不合法。其实要区别它们也不难，给你一个表达式，你也能将它们区别开来，实际上，我们只要看 `*` 前一个 token 的类型，我们就可以决定这个 `*` 是乘法还是指针解引用了，不信你试试？我们在这里给出 `expr()` 函数的框架：

```
if(!make_token(e)) {
    *success = false;
    return 0;
}
/* TODO: Implement code to evaluate the expression. */
for(i = 0; i < nr_token; i++) {
    if(tokens[i].type == '*' && (i == 0 || tokens[i - 1].type == certain type)) {
        tokens[i].type = Deref;
    }
}
return eval(?, ?);
```

其中的 `certain type` 就由你自己来思考啦！其实上述框架也可以处理负数问题，如果你之前实现了负数，`*` 的识别对你来说应该没什么困难了。

另外和 GDB 中的表达式相比，我们做了简化，简易调试器中的表达式没有类型之分，因此我们需要额外说明两点：

1. 为了方便统一，我们认为所有结果都是 `uint32_t` 类型。
2. 指针也没有类型，进行指针解引用的时候，我们总是从内存中取出一个 `uint32_t` 类型的整数，同时记得使用 `vaddr_read()` 来读取内存。

§2-3.1.2 实验要求

实现表达式的词法分析

你需要完成以下的内容:

1. 为算术表达式中的各种 `token` 类型添加规则, 你需要注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能.
2. 在成功识别出 `token` 后, 将 `token` 的信息依次记录到 `tokens` 数组中.

实现表达式的递归求值

你需要实现上文 BNF 中列出的功能. 一个要注意的地方是词法分析中编写规则的顺序, 不正确的顺序会导致一个运算符被识别成两部分, 例如 `!=` 被识别成 `!` 和 `=`. 关于变量的功能, 它需要涉及符号表和字符串表的查找, 因此你会在下一阶段中实现它.

上面的 BNF 并没有列出 C 语言中所有的运算符, 例如各种位运算, `<=` 等等. `==`, `!=` 和逻辑运算符很可能在使用监视点的时候用到, 因此要求你实现它们. 如果你在将来的使用中发现由于缺少某一个运算符而感到使用不方便, 到时候你再考虑实现它.

在完成上述两个任务的时候, 你可以尝试分步走的办法: 首先针对简单的算术表达式实现其词法和求值功能, 再扩展到更为复杂的表达式.

§2-3.1.3 延伸话题: 从表达式求值窥探编译器

你在程序设计课上已经知道, 编译是一个将高级语言转换成机器语言的过程. 但你是否曾经想过, 机器是怎么读懂你的代码的? 回想你实现表达式求值的过程, 你是否有什么新的体会?

事实上, 词法分析也是编译器编译源代码的第一个步骤, 编译器也需要从你的源代码中识别出 `token`, 这个功能也可以通过正则表达式来完成, 只不过 `token` 的类型更多, 更复杂而已. 这也解释了你为什么可以在源代码中插入任意数量的空白字符(包括空格, `tab`, 换行), 而不会影响程序的语义; 你也可以将所有源代码写在一行里面, 编译仍然能够通过.

一个和词法分析相关的有趣的应用是语法高亮. 在程序设计课上, 你可能完全没有想过可以自己写一个语法高亮的程序, 事实是, 这些看似这么神奇的东西, 其实也没那么复杂, 你现在确实有能力来实现它: 把源代码看作一个字符串输入到语法高亮程序中, 在循环中识别出一个 `token` 之后, 根据 `token` 类型用不同的颜色将它的内容重新输出一遍就可以了. 如果你打算将高亮的代码输出到终端里, 你可以使用 ANSI 转义码的颜色功能.

在表达式求值的递归求值过程中, 逻辑上其实做了两件事情: 第一件事是根据 `token` 来分析表达式的结构(属于 BNF 中的哪一种情况), 第二件事才是求值. 它们在编译器中也有对应的过程: 语法分析就好比分析表达式的结构, 只不过编译器分析的是程序的结构, 例如哪些是函数, 哪些是语句等等. 当然程序的结构要比表达式的结构更复杂, 因此编译器一般会使用一种标准的框架来分析程序的结构, 理解这种框架需要更多

的知识，这里就不展开叙述了。另外如果你有兴趣，可以看看 C 语言语法的 BNF。

和表达式最后的求值相对的，在编译器中就是代码生成。ICS 理论课会有专门的章节来讲解 C 代码和汇编指令的关系，即使你不了解代码具体是怎么生成的，你仍然可以理解它们之间的关系，这是因为 C 代码天生就和汇编代码有密切的联系，高水平 C 程序员的思维甚至可以在 C 代码和汇编代码之间相互转换。如果要深究代码生成的过程，你也不难猜到是用递归实现的：例如要生成一个函数的代码，就先生成其中每一条语句的代码，然后通过某种方式将它们连接起来。

我们通过表达式求值的实现来窥探编译器的组成，是为了落实一个道理：学习汽车制造专业不仅仅是为了学习开汽车，是要学习发动机怎么设计。我们也强烈推荐你在将来修读“编译原理”课程，深入学习“如何设计发动机”。

§2-3.2 添加变量和函数名支持

§2-3.2.1 预备知识和代码导读

你已经在上一阶段中实现了简易调试器。同时在这一阶段现在你已经将用户程序换成了 C 程序。和之前的 `mov.S` 相比，C 程序多了变量和函数的要素，那么在表达式求值中如何支持变量的输出呢？

`(nemu) p test_data`

换句话说，我们怎么从 `test_data` 这个字符串找到这个变量在运行时刻的信息？下面我们就来讨论这个问题。

符号表(symbol table)是可执行文件的一个 section，它记录了程序编译时刻的一些信息，其中就包括变量和函数的信息。为了完善调试器的功能，我们首先需要了解符号表中都记录了哪些信息。

以 `add` 这个用户程序为例，使用 `readelf` 命令查看 ELF 可执行文件的信息：

`readelf -a add`

你会看到 `readelf` 命令输出了很多信息，这些信息对了解 ELF 的结构有很好的帮助，我们建议你在课后仔细琢磨。目前我们只需要关心符号表的信息就可以了，在输出中找到符号表的信息：

```
Symbol table '.symtab' contains 10 entries:
  Num:   Value   Size Type   Bind   Vis     Ndx Name
   0: 00000000    0 NOTYPE  LOCAL DEFAULT UND
   1: 00100000    0 SECTION LOCAL DEFAULT 1
   2: 0010009c    0 SECTION LOCAL DEFAULT 2
   3: 00100100    0 SECTION LOCAL DEFAULT 3
   4: 00000000    0 SECTION LOCAL DEFAULT 4
   5: 00000000    0 FILE    LOCAL DEFAULT ABS add.c
   6: 00100084   22 FUNC    GLOBAL DEFAULT 1 add
   7: 00100000  129 FUNC    GLOBAL DEFAULT 1 main
   8: 00100120  256 OBJECT GLOBAL DEFAULT 3 ans
   9: 00100100   32 OBJECT GLOBAL DEFAULT 3 test_data
```

图 2-11 符号表举例

这个标识符在符号表中找到一项符合要求的表项(表项的 `Type` 属性是 `OBJECT` , 并且将 `Name` 属性的值作为字符串表中的偏移所找到的字符串和标识符的命名一致), 找到标识符的地址, 并将这个地址作为结果返回. 在上述 `add` 程序的例子中:

```
(nemu) p test_data
```

```
0x100100
```

需要注意的是, 如果标识符是一个基本类型变量, 简易调试器和 GDB 的处理会有所不同: 在 GDB 中会直接返回基本类型变量的值, 但我们在表达式求值中并没有实现类型系统, 因此我们无法区分一个标识符是否基本类型变量, 所以我们统一输出变量的地址. 如果对于一个整型变量 `x` , 我们可以通过以下方式输出它的值:

```
(nemu) p *x
```

而对于一个整型数组 `A` , 如果想输出 `A[1]` 的值, 可以通过以下方式:

```
(nemu) p *(A + 4)
```

§2-3.2.2 实验要求

为表达式求值添加变量的支持

根据上文提到的方法, 向表达式求值添加变量的支持, 为此, 你还需要在表达式求值的词法分析和递归求值中添加对变量的识别和处理. 框架代码提供的 `load_elf_tables()` 函数已经为你从可执行文件中抽取出符号表和字符串表了, 其中 `strtab` 是字符串表, `symtab` 是符号表, `nr_symtab_entry` 是符号表的表项数目, 更多的信息请阅读 `nemu/src/monitor/elf.c` . (感谢 16 级张航帆同学发现的教程笔误。)

头文件 `<elf.h>` 已经为我们定义了与 ELF 可执行文件相关的数据结构, 为了使用符号表, 请查阅

```
man 5 elf
```

实现之后, 你就可以在表达式中使用变量了. 在 NEMU 中运行 `add` 程序, 并打印全局数组某些元素的值.

消失的符号?

在实验报告中, 回答下面这个问题:

我们在 `add.c` 中定义了宏 `NR_DATA` , 同时也在 `add()` 函数中定义了局部变量 `c` 和形参 `a, b` , 但你会发现符号表中找不到和它们对应的表项, 为什么会这样? 思考一下, 什么才算是一个符号(symbol)?

PA 3 存储管理

通过 PA 2 的实验，我们已经创造了一台能够进行各种复杂运算的机器。在以后学习了包括图灵机在内的知识后，我们就能够了解到在 PA 2 中所实现机器的强大之处：它的计算能力和现今所有的计算机等价。各种超级计算机只是算得比咱快，但我们的机器说不可计算的问题，超级计算机甚至是量子计算机也没办法。

做计算理论的科学家们到这一阶段往往就心满意足了，但我们却还不能止步不前。我们希望机器能够算得更快，于是针对速度的瓶颈之一：访存操作进行加速，于是便设计了 cache。我们希望在机器上运行的程序和数据得到更多保护，于是便加入了分段保护机制。我们希望同时运行的多个程序之间互不干扰，于是便产生了虚拟地址空间的做法。

PA 3-1 Cache 的模拟

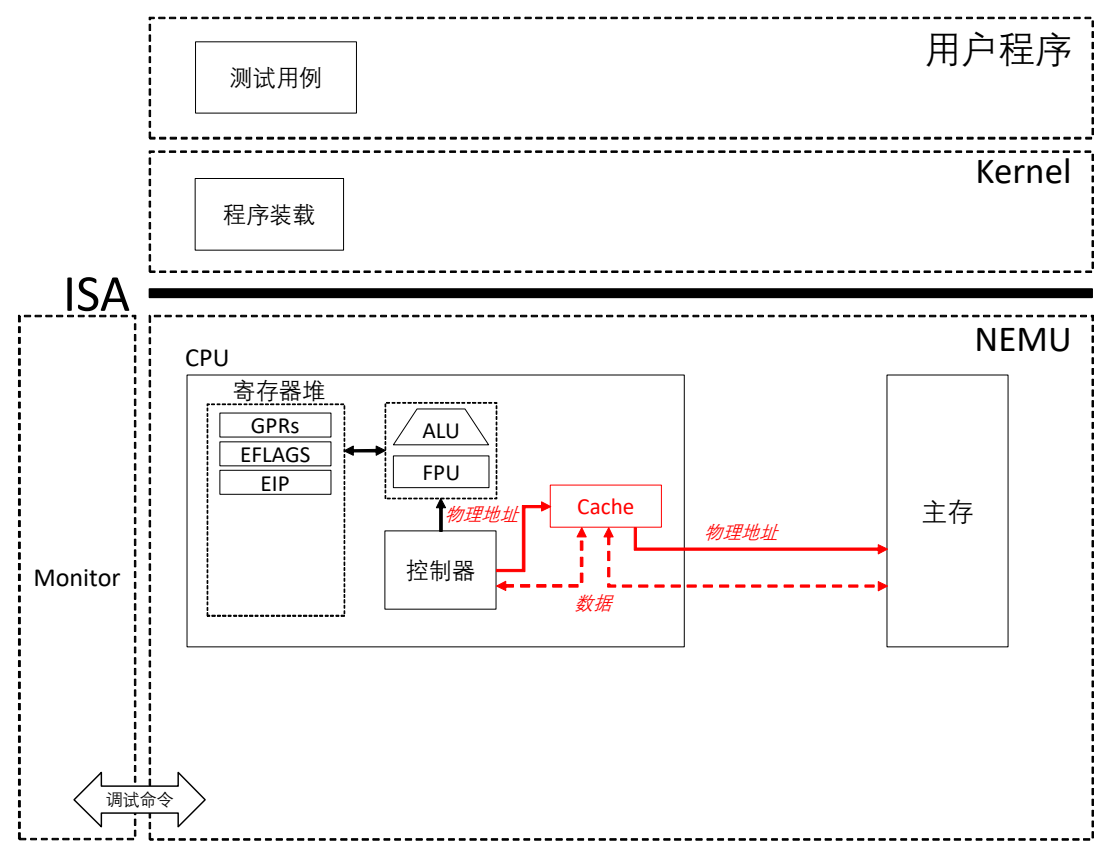


图 3-1 PA 3-1 Cache 模拟路线图

在之前的阶段中，我们将指令和数据存放在主存中。在程序的执行过程中，CPU 先通过访存操作获取指令和源操作数，执行操作后将结果通过访存操作写入目的操作数。每一条指令的执行都需要进行至少一次访存。

随着大规模集成电路和半导体工艺的进步，CPU 变得越来越快。而采用动态随机存取存储器（Dynamic Random Access Memory，DRAM）实现的主存由于其工作原理的限制，其速度难以得到进一步提高。简单来说，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，由于电容会漏电的特性，在将一行数据读出之前，还要对这一行的电容进行预充电，防止在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。

同时，CPU 和主存之间需要通过系统总线通过 I/O 桥接器连接内存总线来交换信息。CPU 每次发起对主存的读写都将引起一系列的操作称为总线事务。以读事务为例，首先，CPU 将地址放到系统总线上；I/O 桥接器将信号传递到内存总线；主存获取内存总线上的地址信号，读取 DRAM 得到数据字，并将数据写到内存总线；I/O 桥接器将数据传递到系统总线；最后 CPU 才能读取通过系统总线传来的数据。

由此可见，CPU 的每次访存都需要进过一系列的信号转换，最后到一个速度受限的 DRAM 上去完成数据的

存取。由于每一条指令的执行都至少需要进行一次访存。如果不采取有效措施，那么系统速度就会卡在 DRAM 这一个瓶颈上，CPU 造的再快也没有效果。那如何提高访存的效率呢？通过上面的论述，我们可以自然地想到从两个地方着手：

1. 采用比 DRAM 更快的器件，于是我们想到采用无需刷新的静态随机存取存储器 (Static Random Access Memory, SRAM) 来代替 DRAM；
2. 让存储部件离 CPU 更近一点，我们把 SRAM 直接放到 CPU 的芯片里面。

事实表明，这是一种很成功的做法，能够使这一部分 SRAM 的存取速度相比 DRAM 提高十几甚至上百倍，CPU 能够在几个时钟周期内就完成对 SRAM 数据的访问。我们将这一部分造在 CPU 中的 SRAM 称为高速缓存 (Cache)。

那我们还要 DRAM 干什么？为什么不在 CPU 芯片内做一个很大的，比如 4GB，的 SRAM 来代替 DRAM？因为造不起。SRAM 的相对访问时间是 DRAM 的十分之一，而相对花费却在千倍以上。因此，这一部分造在 CPU 内部的 SRAM 只能提供非常有限的存储空间，用来放置一些被频繁使用的数据。CPU 访问主存前先访问 Cache，如果数据在 Cache 中（称为 Cache 命中），则抄近路从 Cache 取数据；若不在（称为 Cache 缺失），那只能老老实实去访问主存。因此，如何提高 Cache 的命中率是 Cache 设计中的一个核心问题。

§3-1.1 预备知识

使有限的 Cache 能够高效存储相对近乎无限的主存空间中的数据并保证较高的命中率，利用的是程序访问地址的**局部性原理 (locality)**：

1. 时间局部性：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）；
2. 空间局部性：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。

相应地，我们希望利用程序访问地址的局部性来高效的缓存数据：

1. 利用时间局部性原理，Cache 将缓存从主存中读出的数据，这样下次再访问的时候就不需要再次访存，而只需从 Cache 中读取即可；
2. 利用空间局部性原理，每次 Cache 缓存数据的时候并不是 CPU 要多少就缓存多少，而是多读一点。Cache 和主存之间交换数据的基本单元在主存中称为块 (block)，而在 Cache 中则称为行 (line) 或槽 (slot)。

在确立了基本思想之后，要使得 Cache 能够真正地实现出来，我们还需要解决一系列的具体问题：

1. 主存中的块与 Cache 中的槽如何对应？在这里就要考虑到查找的效率和 Cache 存储空间使用效率的权衡。于是就产生了直接映射、全相联映射和组相联映射这三种方式；
2. 当新访问的主存块映射到 Cache 中已经被占用的槽时怎么办？于是便产生了不同的替换策略如先进先出、最近最少用、最不经常用和随机替换算法等；
3. 当 Cache 槽中的数据和主存对应块的数据产生不一致时怎么办？这种不一致只会由对 Cache 的写操作引起，于是针对写操作的不同处理方法就形成了全写法和回写法两类方法。

-
4. 写操作时 Cache 缺失时怎么处理？根据是否将内存块调入 Cache 就形成了包括写分配法和非写分配法两种基本的策略。

对于上述问题的回答没有一个统一的最优答案，在工程实践过程中，每一个策略组合的选择都有其优点和缺陷。每一个设计都是面对多种可能因素的一个权衡。

在现代处理器设计中，不仅仅只有一个 cache，还有针对指令的指令 cache 和针对数据的数据 cache，同时 cache 也往往分为多级，其目的就在于充分利用 CPU 芯片上有限的空间，在成本和性能间不断取得最优的权衡结果。课本第 6.4.7 节简要介绍了相关的知识。

§3-1.2 代码导读和实验理解

在框架代码中我们并没有给出和 cache 实现相关的样例，你需要自行进行添加。在 `include/config.h` 中通过定义宏 `CACHE_ENABLED` 开启对 cache 的模拟。在 `nemu/src/memory/memory.c` 中，在 `paddr_read()` 和 `paddr_write()` 函数里，增加相应的代码来通过 cache 实现对物理内存的读写：

```
uint32_t paddr_read(paddr_t paddr, size_t len) {
    uint32_t ret = 0;
    #ifndef CACHE_ENABLED
        ret = cache_read(paddr, len, &L1_dcache);
    #else
        ret = hw_mem_read(paddr, len);
    #endif
    return ret;
}

void paddr_write(paddr_t paddr, size_t len, uint32_t data) {
    #ifndef CACHE_ENABLED
        cache_write(paddr, len, data, &L1_dcache);
    #else
        hw_mem_write(paddr, len, data);
    #endif
}
```

在 `cache_read()` 和 `cache_write()` 函数中，封装通过 cache 读写物理内存的逻辑。如 cache 命中，则直接从 cache 进行读（写的话根据是否采用直写法进行相应操作）；如 cache 缺失，则通过 `hw_mem_read()` 和 `hw_mem_write()` 接口将主存块调入 cache 后再写（写缺失的处理根据是否采用写分配法来确定）。

由于 NEMU 是用软件模拟的 cache，因此不可能在物理上达到真正 cache 硬件的加速性能。为此，你可以通过模拟的计时器来测试 cache 的性能。如增加一个全局的计时器变量，当 cache 命中时，为访问时间加上 1；而 cache 缺失时，则为访问时间加上 10。最后等测试用例执行完后，比较一下采用 cache 和不采用 cache 的模拟执行时间的区别。

§3-1.3 实验过程及要求

在 `include/config.h` 中定义宏 `CACHE_ENABLED` 并 `make clean`；

在 NEMU 中实现一个 cache，它的性质如下：

1. cache block 存储空间的大小为 64B

-
2. cache 存储空间的大小为 64KB
 3. 8-way set associative
 4. 标志位只需要 valid bit 即可
 5. 替换算法采用随机方式
 6. write through
 7. not write allocate

你还需要在 `nemu/src/memory/memory.c` 的 `init_mem()` 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 `memory.c` 中的 `paddr_read()` 和 `paddr_write()` 函数，让它们读写 cache，当缺失时由 cache 负责调用 `hw_mem_read()` 和 `hw_mem_write()` 读写 DRAM。

PA 3-2 保护模式

在之前的阶段中，NEMU 始终工作在类似于“实模式”的状态下，简言之，就是程序直接通过物理地址访问主存。从本小节开始，我们开始让 NEMU 具备现代计算机的内存管理功能。

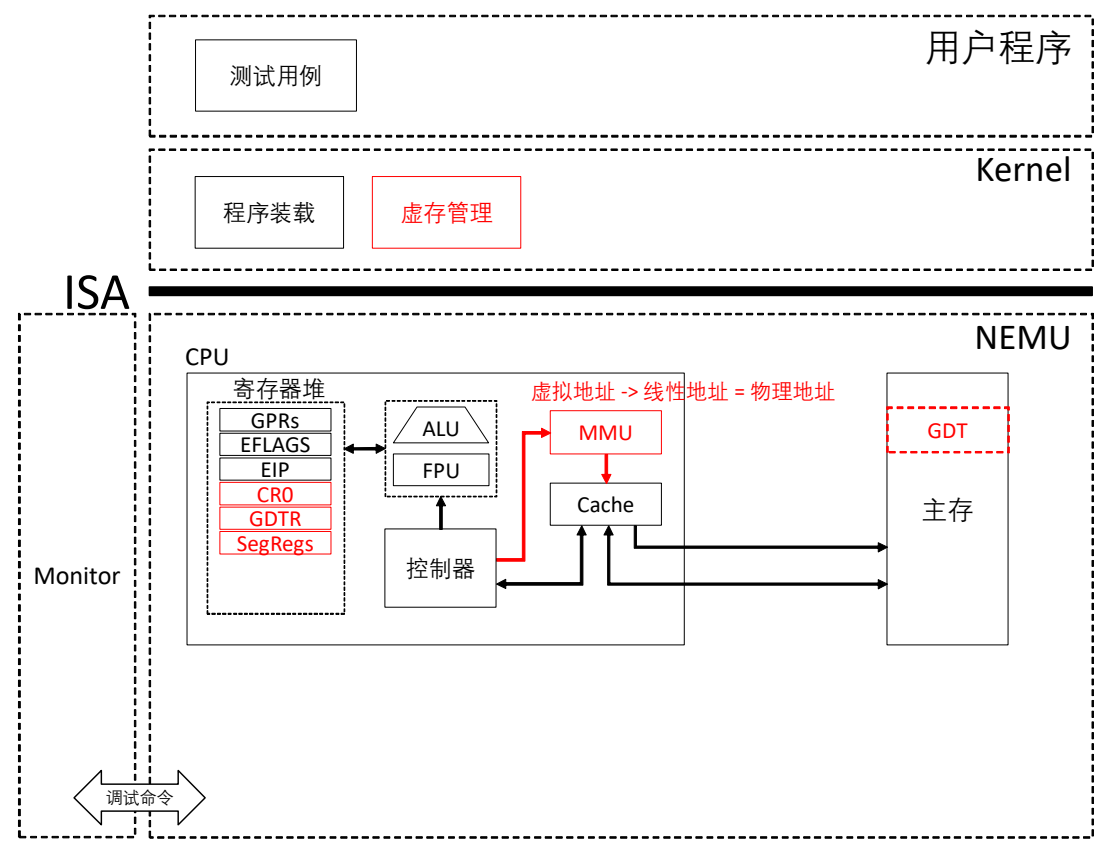


图 3-2 PA 3-2 实现分段机制路线图

§3-2.1 预备知识

§3-2.1.1 8086 的实模式

在最早的 8086 时代，所有的寄存器都是 16 位的。若地址也是 16 位的，那么能够寻址的内存大小为 $2^{16} \text{ B} = 64\text{KB}$ 。为了扩展可寻址的空间，8086 引入了一系列 16 位的段寄存器。引入段寄存器后，实际的地址计算方式如下：

$$\text{physical address} = (\text{seg_reg} \ll 4) + \text{offset}$$

其中，`seg_reg` 是某一个段寄存器的值，而 `offset` 则是程序给出的 16 位的地址偏移量，加起来物理地址的位数总共为 20 位。如此，便可以寻址 $2^{20} \text{ B} = 1\text{MB}$ 的地址空间，在 8086 的时代，1MB 的内存空间已经是非常巨大了。尽管目前看来，当初的实模式已经不能满足现代计算机的需求。但是有很多设计却一直保留到今天：

首先，约定各段寄存器和 offset 之间的绑定关系如下：

偏移量（offset）类型	绑定的段寄存器
代码段（对应 eip）	CS
数据访问	DS
堆栈访问（对应 esp 和 ebp）	SS
特殊类型访问（如 movs）	ES

第二，在系统初始化时，先进入实模式。当完成初始化后，可由程序控制使得机器转入保护模式执行。此举的主要目的就是为了实现新机器对老程序的向下兼容。尽管使得机器设计会变得更加复杂，但是作为一款成功的商业产品，向下兼容是保证新产品能够被市场接受的一个重要条件。

§3-2.1.2 80386 的保护模式

保护模式下的寻址过程

在保护模式下，寻址方式会产生变化。简单来说，程序给出的 32 位地址不再直接解释为物理地址，而是相对于某一个段的偏移量（offset）。真正的物理地址由下式给出：

$$\text{physical address} = \text{linear address} = \text{base address} + \text{offset}$$

其中的 base address 是一个 32 位的地址，对应某个段的基地址；而 offset 则是程序给出的 32 位段内偏移量。在这里我们引入了一个新概念叫线性地址（linear address），这个概念直到我们介绍分页机制的时候才会用到。在现阶段，线性地址就等于物理地址（physical address）。

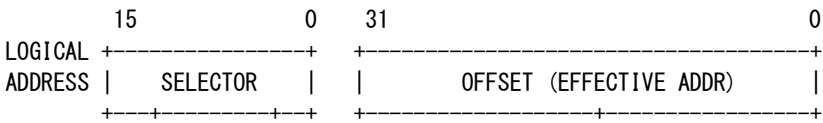
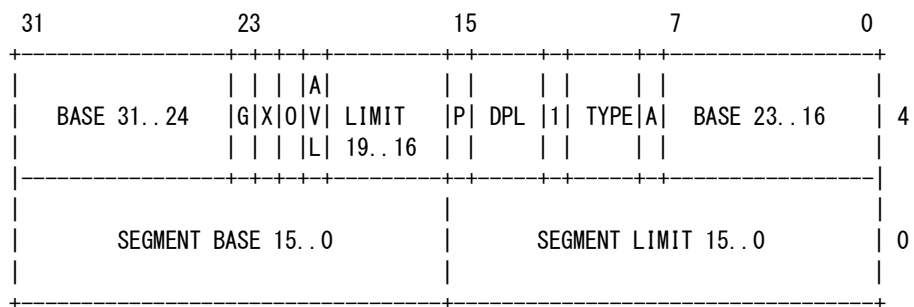


图 3-3 48 位逻辑地址结构

当开启保护模式后，NEMU 中运行的程序访问内存时给出的就不简单是 32 位的物理地址了，而是由一个 16 位的段选择符加上 32 位的段内偏移量（有效地址）所构成的 48 位的逻辑地址（或称虚拟地址）。由于 32 位的段内偏移量是由程序直接给出，那唯一的问题就是如何通过 16 位的段选择符来获取 32 位的段基地址（base address）。在 80386 中，这一过程通过查表来实现。计算机与操作系统约定，若要开启保护模式，则操作系统需要事先在内存中准备好一个表，叫做“段表”，其中存储好每个段的首地址（base address）、段的长度（limit）等相关的信息。段表由一系列连续的段表项构成，其中每个段表项都是一个 64 位的数据结构称为段描述符，其结构如下：

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



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

图 3-4 64 位段描述符结构

于是整个段表就是多个段表项构成的一个数组：

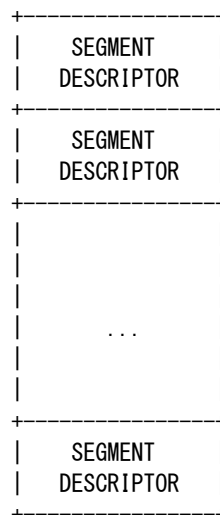


图 3-5 段表

在 48 位的逻辑地址中，包含了 16 位的段选择符，一个段选择符的结构如下：



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

图 3-6 16 位段选择符结构

如上图所示的段选择符存储在段寄存器的可见部分，由三个部分组成：高 13 位是一个 `index`，用于指出所访问段的段描述符是段表中的第几项（数组下标）；`TI` 用于指出选择哪一个段描述符表，`TI` 为 0 时表示选择全局描述符表（GDT），`TI` 为 1 时表示选择局部描述符表（LDT），在 NEMU 中，我们只关注 GDT；最后 `RPL` 则与访问权限控制有关，在 NEMU 中，我们不模拟权限管理，但会在本节最后进行一些讨论。

结合虚拟地址、段选择符和段表的相关概念，在分段机制中，将虚拟地址转换成线性地址（此时即为物理地址）的过程可描述如下：

1. 根据段选择符中的 `TI` 位选择 GDT 或 LDT（NEMU 中永远是 GDT）；
2. 根据段选择符中的 `index` 部分到 GDT 中找到对应位置上的段描述符；
3. 读取段描述符中的 `base` 部分，作为 32 位段基址，加上 32 位段内偏移量获取最终的物理地址。

在此过程中，我们还需要一项关键信息，那就是段表 GDT 的首地址。这一项关键信息保存在 CPU 中的一个特殊寄存器 `GDTR` 中。该寄存器中保存了 GDT 的首地址（线性地址）和界限，由操作系统在系统初始化时填入，并对用户程序不可见。上述过程也可以由下图说明：

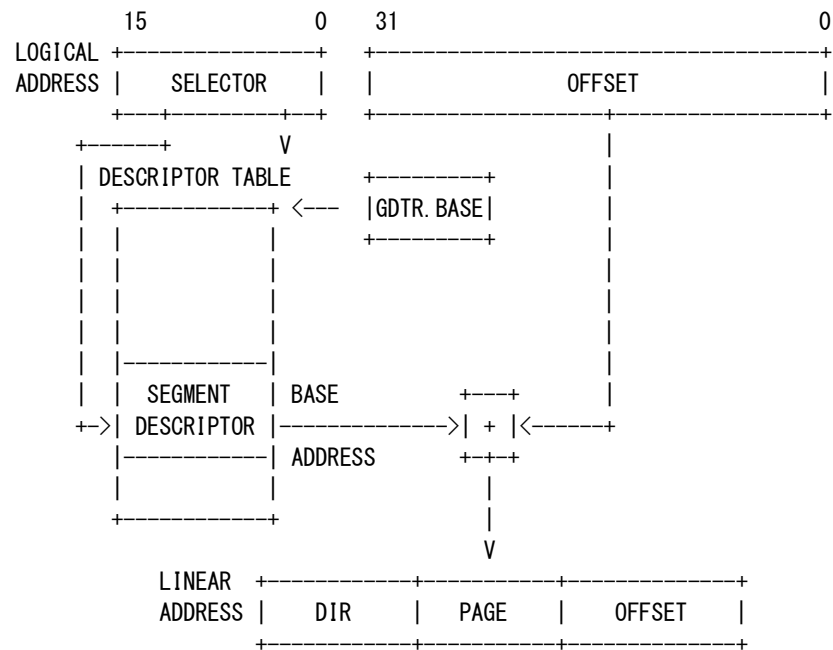


图 3-7 从逻辑地址到线性地址（此时即为物理地址）的转换

从实模式切换到保护模式

如前所述，计算机在刚启动时进入的是实模式。在实模式下，操作系统需要完成包括初始化段表（如 `GDT`）和描述符表寄存器（如 `GDTR`）。在初始化完成后，操作系统通过将 0 号控制寄存器（`CR0`）中的 `PE` 位置为 1 的方式，来通知机器进入保护模式。在此之前，`CR0` 中的 `PE` 初始化为 0。`CR0` 寄存器的结构请自行参阅 i386 手册的相关内容。

§3-2.2 代码导读和实验理解

在 NEMU 中实现保护模式需要 Kernel 和 NEMU 配合完成保护模式的初始化。在 PA 中，为了开启保护模式

相关的功能，我们首先需要在 `include/config.h` 头文件中添加宏定义 `#define IA32_SEG`。

§3-2.2.1 Kernel 的行为改变

在重新 `make` 之后（可能要先执行 `make clean`），这将引起 Kernel 和 NEMU 行为的一些变化。对于 Kernel 而言，其行为改变发生在 `kernel/start/start.S` 中：

1. 通过 `lgdt` 设置全局描述符表
2. 将 `CR0` 寄存器的 `PE` 位置为 `1` 开启保护模式
3. 使用 `ljmp` 指令装载 `CS` 段寄存器
4. 通过 `mov` 指令初始化 `DS, ES, SS` 段寄存器；
5. 转入后续执行。

我们约定 NEMU 工作在扁平模式，即所有段的基址都为 `0x0`，界限为全 `1`。这一点体现在 `start.S` 所准备的段表 `gdt` 中。

§3-2.2.2 NEMU 的进化

为了实现保护模式，NEMU 首先需要模拟相应的器件。在 `nemu/include/cpu/reg.h` 头文件中，我们要为 `CPU_STATE` 结构添加上必要的器件模拟，其中包括 `GDTR`、`CR0` 和长度为 6 的段寄存器数组 `SegReg[]`。其中，`GDTR` 的类型为一个 `struct`，包含 16 位的 `limit` 和 32 位的 `base` 两个部分。`CR0` 寄存器的结构参照 i386 手册，其中最关键的是要包含 `PE` 位，后续到分页机制阶段，则需要利用到 `CR0` 中的 `PG` 位。你可以参照 `EFLAGS` 的实现方法来实现 `CR0`。段寄存器结构体类型 `SegReg` 除了在其可见部分保存了段选择符以外，还在其不可见部分缓存了对应段描述符中的关键信息如 `base` 和 `limit`，可参照 i386 手册或课本第 274 页图 6.38 的描述。参考实现代码：

```
typedef struct {
    uint32_t limit :16;
    uint32_t base :32;
}GDTR;

typedef union {
    struct {
        uint32_t pe :1;
        uint32_t mp :1;
        uint32_t em :1;
        uint32_t ts :1;
        uint32_t et :1;
        uint32_t reserve :26;
        uint32_t pg :1;
    };
    uint32_t val;
}CR0;

typedef struct {
```

```

// the 16-bit visible part, i.e., the selector
union {
    uint16_t val;
    struct {
        uint32_t rpl :2;
        uint32_t ti :1;
        uint32_t index :13;
    };
};

// the invisible part, i.e., cache part
struct {
    uint32_t base;
    uint32_t limit;
    uint32_t type :5;
    uint32_t privilege_level :2;
    uint32_t soft_use :1;
};
}SegReg;

```

如果采用上述命名约定，在实现了各个寄存器的类型后，在为 `CPU_STATE` 结构中就可以通过添加类似如下的代码来实现对器件的模拟了。

```

GDTR gdt; // GDTR
union { // segment registers
    SegReg segReg[6];
    struct { SegReg es, cs, ss, ds, fs, gs; };
};
CR0 cr0; // control register 0

```

在实现了对器件的模拟之后，可以对 NEMU 在访问内存时的行为进行相应的调整。所涉及的代码主要包含在 `nemu/src/memory/` 文件夹下。

首先，NEMU 需要针对保护模式中的分段机制提供地址翻译的功能。这部分功能的代码包含在 `nemu/src/memory/mmu/segment.c` 的 `segment_translate()` 函数中。这个函数包含两个参数，分别是 32 位的有效地址（即段内偏移量）`offset`，和段寄存器的编号 `sreg`。要实现这个函数的功能是非常简单的：

1. 使用 `sreg` 作为编号查询 CPU 中的 `SegReg` 段寄存器数组；
2. 读出其隐藏部分的 `base`，将其与 `offset` 相加并返回结果即可。

第二，段寄存器的隐藏部分怎么来？我们注意到在系统初始化完成后，每个段的 `base` 和 `limit` 一般不会发生变化（在扁平模式下更是如此）。因此，只需要在每个段寄存器在初始化时将对应段的 `base` 和 `limit` 等信息装入段寄存器的隐藏部分即可一劳永逸。那么段寄存器在何时初始化？观察 Kernel 的行为，发现只有 `ljmp` 和目的操作数为段寄存器的 `mov` 指令才会引起段寄存器内容发生变化（在后面实现到中断的时候，我们会发现 CPU 在响应中断和异常时也会引起段寄存器的内容发生改变，但这是后话了）。因此，在实现 `ljmp` 和 `mov` 指令时，需要同时完成对段寄存器隐藏部分的加载，在框架代码中，这个过程可以通过调用 `load_sreg()`

函数来实现。`load_sreg()`函数定义在 `nemu/src/memory/mmu/segment.c` 中，其参数是需要装载的段寄存器的编号，需要完成的工作是根据段寄存器在 `ljmp` 和 `mov` 指令的操作数中给出的 `index` 查询段表并完成隐藏部分的装填。注意 GDTR 中保存的基地址是线性地址。

在 `nemu/include/memory/mmu/segment.h` 中，我们已经准备好了段表项的数据结构 `SegDesc`。在 `load_sreg()` 中，建议通过 `assert` 针对扁平模式下 `base` 等于 0，`limit` 为全 1，`granularity` 为 1 等条件进行检查，以尽早暴露可能的错误。

第三，有了上述功能的支持，我们就可以在 `memory.c` 中为 NEMU 添加分段机制了。修改 `vaddr_read()` 和 `vaddr_write()` 函数的实现，当 CR0 的 PE 位为 1 时，通过我们实现好的 `segment_translate()` 函数将逻辑地址翻译成线性地址再进行下一层的内存访问。在这里，将 `vaddr_read()` 和 `vaddr_write()` 传入的参数 `vaddr` 直接作为 `offset` 传给 `segment_translate()` 函数即可。此时 `vaddr` 已经转义为有效地址了，请理解这一区别。

最后，不要忘记在 CPU 初始化时（在 `nemu/src/cpu/cpu.c` 的 `init_cpu()`），将 CR0 和 GDTR 初始化为全 0。

§3-2.3 实验过程及要求

1. 在 `include/config.h` 头文件中添加宏定义 `IA32_SEG` 并 `make clean`；
2. 在 `CPU_STATE` 中添加对 GDTR、CR0 的模拟以及在 `init_cpu()` 中进行初始化为 0；
3. 在 `CPU_STATE` 中添加对 6 个段寄存器的模拟在 `init_cpu()` 中进行初始化为 0，注意除了要模拟其 16 位的可见部分，还要模拟其隐藏部分，顺序不能有错；
4. 实现包括 `lgdt`、针对控制寄存器和段寄存器的特殊 `mov` 以及 `ljmp` 指令；
5. 实现 `segment_translate()`、`load_sreg()` 函数，并在 `vaddr_read()` 和 `vaddr_write()` 函数中添加保护模式下的虚拟地址向线性地址转换的过程；
6. 通过 `make testkernel` 执行并通过各测试用例。

在实验报告中，请回答如下问题：

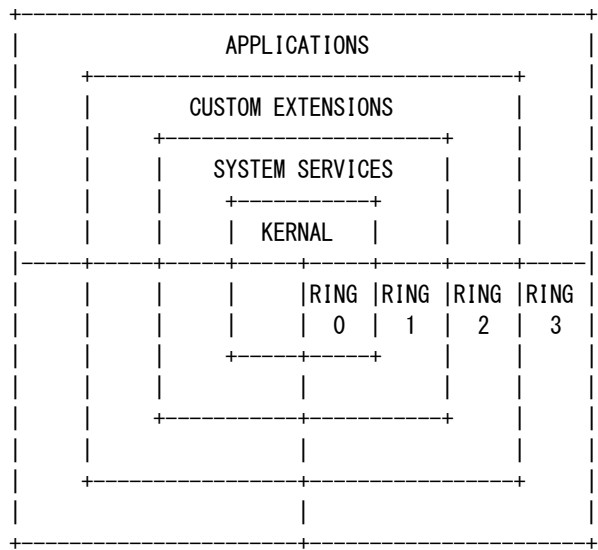
1. NEMU 在什么时候进入了保护模式？
2. 在 GDTR 中保存的段表首地址是虚拟地址、线性地址、还是物理地址？为什么？

§3-2.4 延伸阅读：特权等级

在上述实现分段机制的实验中，我们仅仅讨论了使用段表完成从逻辑地址到线性地址的转换。而事实上当工作在扁平模式下的时候，这种转换没有起到什么作用，其保护的意味并不明显。然而在我们编写程序的过程中，如果访问了错误的内存地址，机器还是抛出“段错误”，这又是怎么做到的呢？这就需要谈谈 80386 中所引入的特权等级的概念。回忆图 3-4 中所给出的段描述符结构和图 3-6 中给出的段选择符结构，我们发现它们分别拥有 2 比特的 DPL（Descriptor Privilege Level）和 RPL（Requestor's Privilege Level）域，分别指出了段所在的特权级，以及请求者所在的特权级。除此以外，还有一个 CPL（Current Privilege Level）用于指出当前进程的特权级，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。

既然标识特权等级的 DPL、RPL 和 CPL 都是两位的，自然就一共能够表示 0、1、2、3 四个特权等级，其中

特权等级 0 为最高，等级 3 为最低。低等级能够访问的资源，在高等级中都能够访问。于是按照能够获取相应等级的程序类型来划分，就可以将特权等级划分成不同的环。在相应的术语体系中，分别称之为 ring 0 一直到 ring 3。Ring 0 一般只给操作系统内核使用，而 ring 3 则属于普通的应用程序。在大多数的通用操作系统中，往往只使用 ring 0 和 ring 3 就足够了。



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。以数据段为例，当一个程序要访问内存中的数据时，它必须要事先装载 DS, ES, FS, GS, SS 中的某一个段寄存器。此时，处理器会自动进行特权级的检查：当且仅当从数值上

```
target_descriptor.DPL >= requestor.RPL
```

```
target_descriptor.DPL >= current_process.CPL
```

这两个条件同时满足时，段的切换才是合法的。在实际的执行过程中，requestor 一般是用户进程，而 current_process 则往往是操作系统内核。当用户进程需要针对数据进行一些自己没有权限的操作时，就会委托操作系统内核来帮助其完成操作。这时处理就会自动执行上述两条检查：1) 确保用户进程有权对数据进行操作 (target_descriptor.DPL >= requestor.RPL)；2) 用户进程所委托的对象有权对数据进行操作 (target_descriptor.DPL >= current_process.CPL)。

违反上述第一条规则的例子是：用户进程试图对操作系统的核心数据（比如段表）进行修改。而违反第二条规则的例子则是：操作系统内核（可能被攻击后）委托一个用户进程来对自己的核心数据进行操作。

80386 所引入的保护机制还有很多，感兴趣的同学可以参见 i386 手册 Chapter 6 的相关内容。在 PA 实验中，我们不强调对保护机制的实现，因此所有的程序都工作在最高权限，ring 0 中。目前我们只要理解相关的概念，并且知道在现行的机器上，不同的程序拥有不同的权限即可。到操作系统实验的阶段，会引入针对不同特权等级的实验。

PA 3-3 虚拟地址转换

在上一节中，我们实现从逻辑地址到线性地址的转换。在实现分页机制之前，线性地址就当做物理地址使用。而自从 80386 开始，计算机又提供了一种全新的存储管理方式，那就是分页机制。在分页机制下，每一个进程都拥有独立的存储空间。同时，每一个进程独立的存储空间又具有相同的地址划分方式。此时，线性地址就需要通过进一步的转换，才能获得最终要访问的物理地址。

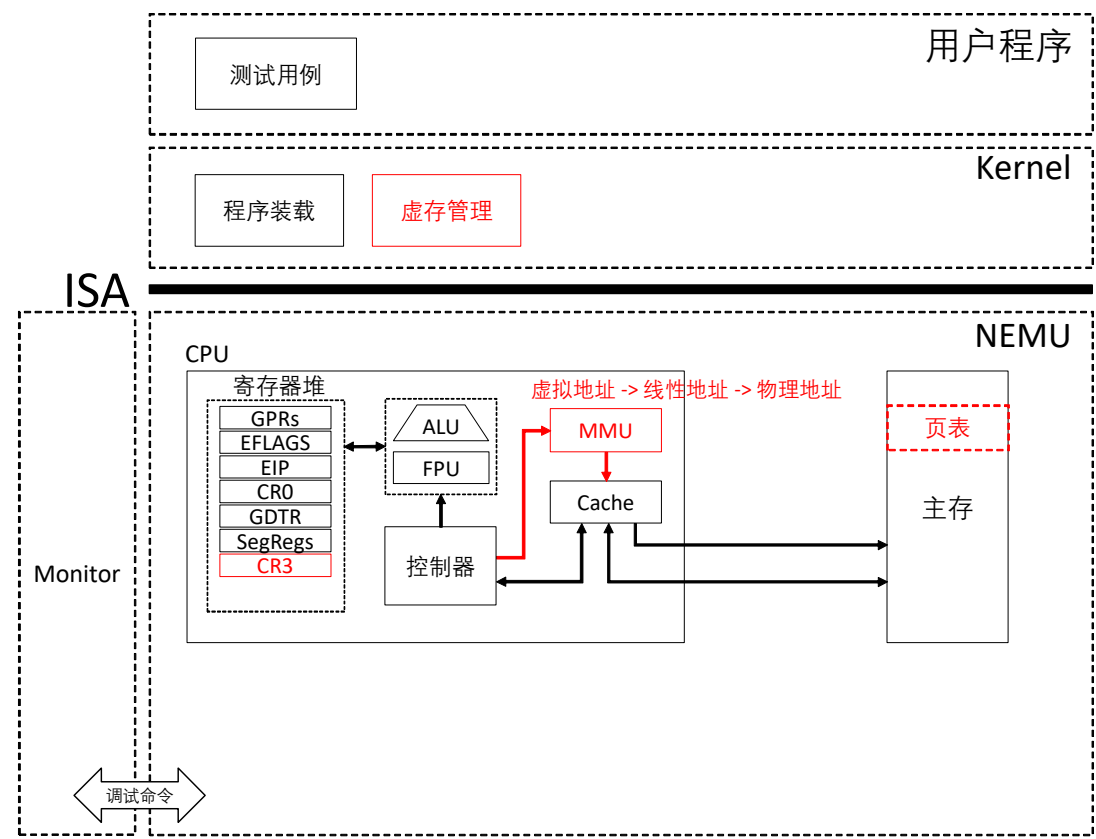


图 3-8 PA 3-3 虚拟存储模拟路线图

§3-3.1 预备知识

在采用虚拟存储技术的计算机系统中，每一个进程都拥有自己独立的虚拟地址空间。对于 32 位的系统而言，每个进程独立的虚拟地址空间就是 0x00000000 ~ 0xFFFFFFFF 共 4GB。在 80386 这样的分页式虚拟存储中，主存地址和虚拟地址空间都被划分成大小相等的页面，在虚拟地址空间中的页面称为虚拟页，而对应的在主存空间中的页面称为物理页。在 x86 系统中，规定每个页面的大小为 $2^{12} = 4KB$ 。于是一个进程的整个虚拟地址空间就被划分成 $1M = 2^{20} = 1024 \times 1024$ 个虚拟页。

虚拟页（无论属于哪一个进程）和物理页之间存在一一对应的关系。这种对应关系由进程的页表来维护。与段表类似，页表也是一个由页表项构成的数组。每个页表项是一个占 32 位的结构体，包含了装入位、修改位、使用位、访问权限、禁止缓存等标志位，最关键地，它包含了某一个虚拟页映射到主存中的哪一个物理页这个关键信息（课本上称为存放位置）。某一个页表项的结构如下图所示：

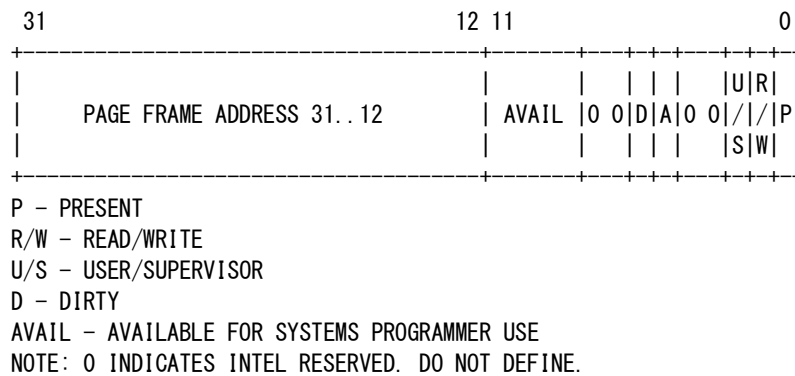


图 3-9 32 位页表项结构

与段描述符中的段基地址不同，32 位系统中，页表项中的物理页存放位置指向的是主存中对应的 20 位物理页号（或称页框号）。若物理页号为 0，则表示该页没有调入主存。这里体现出分页机制和分段机制一个很大的区别：在分页机制下，所寻址的物理页不一定在主存中，而可能位于磁盘上。这种允许用磁盘来扩展存储主存中数据的方式大大提高了系统的处理能力。

每个进程都有自己独立的页表来描述该进程的虚拟地址空间和物理地址空间之间的映射关系。从较为直观的角度来说，一个 32 位的线性地址可以分为两个部分：高 20 位指出该线性地址属于哪一个虚拟页，而低 12 位则指出该地址具体指向的字节在该虚拟页内的偏移量是多少。若参照段表的组织方式，将所有 2^{20} 个页表项都按顺序存放在一个数组中。那么地址转换的过程就可以简单地表达为：使用线性地址中的高 20 位作为索引，查找页表；提取对应页表项中的物理页号，加上线性地址中低 12 位指出的页内偏移量；便能够获取对应的物理地址了。

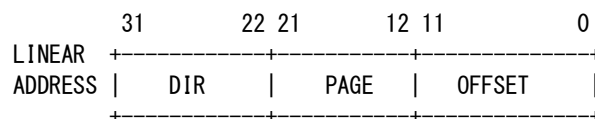
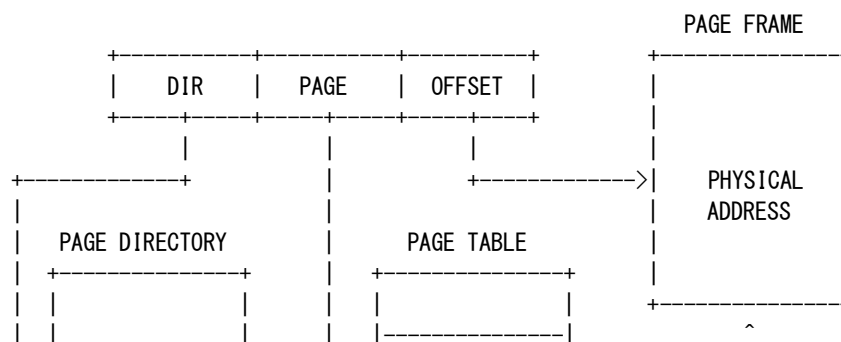


图 3-10 32 位线性地址结构

基本的设计思路就是这样，只不过在具体实现的时候还会遇到一些小麻烦。 2^{20} 个 32 位的页表项如果要组成一个连续的数组，则需要开辟 4MB 也就是 1024 个连续的 4KB 大小的物理页来进行存储。这显然是难以实现的。因此，一个简单的处理方案就是把页表拆成两级。第一级称为页目录，由 1024 个类似页表项的页目录项构成。每个页目录项包含一个物理页号，其所指向的物理页中存储了 1024 个第二级的页表项。如此， 2^{20} 个页表项就被拆分成两级，可以存储在 $1 + 1024$ 个物理页中。这样就大大增加了存储页表的灵活性。在二级页表结构下，一个 32 位的线性地址就可以被分成三个部分，高 10 位的页目录索引，中间 10 位的页表索引和最低 12 位的页内偏移量。

线性地址向物理地址转换的过程可以由下图表示。



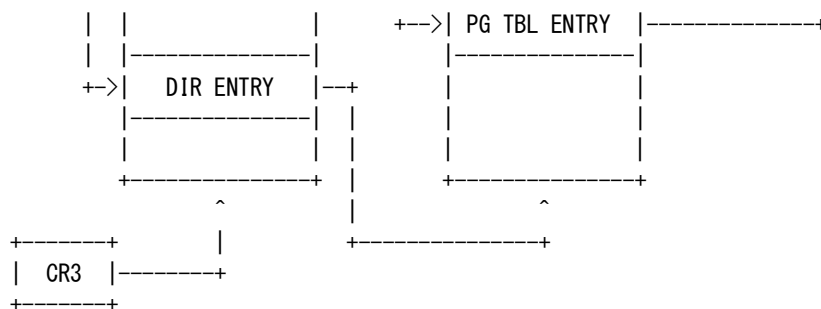


图 3-11 线性地址向物理地址转换的过程

在图中我们注意到还有一个 CR3 寄存器，该寄存器用于保存页目录基地址（物理地址）。具体结构请参见 i386 手册。

§3-3.2 代码导读和实验理解

通过修改 `include/config.h` 头文件在其中定义宏 `IA32_PAGE` 来开启对分页机制的模拟。

§3-3.2.1 修改 Makefile 中的链接选项

需要对 Kernel 和 testcase 的 `Makefile` 也要进行相应的调整。具体而言：

1. 修改 `kernel/Makefile`，将 `LDFLAGS` 中的 `-Ttext=0x30000` 修改为虚拟地址空间中的位置。在 Linux 系统中，约定高于虚拟地址 `0xc0000000` (7 个 0) 的空间为操作系统内核区。为配合这一约定，我们将 Kernel 在虚拟地址空间中的起始地址设为在原物理地址的基础上加上 `0xc0000000`，即，修改 `-Ttext=0x30000` 为 `-Ttext=0xc0030000`；
2. 修改 `testcase/Makefile`，去掉 `LDFLAGS` 中之前对于 `-Ttext` 的设置，使得测试用例的只读数据和代码段从虚拟地址 `0x8048000` 以上开始，符合 Linux 虚拟地址空间的约定。

§3-3.2.2 Kernel 行为的改变

在开启 `IA32_PAGE` 后 Kernel 的行为会发生较大的改变。具体而言：

1. 在 `kernel/start/start.S` 中定义的 `va_to_pa` 宏将产生实质性的作用，会将输入的虚拟地址减去 `KOFFSET=0xc0000000`，从而转换成物理地址。这里使用到了我们先前对于 Kernel 地址偏移量的约定；
2. 在 `kernel/src/main.c` 源文件中的 `init()` 函数里，会调用 `init_page()` 函数来初始化 Kernel 的页表；
3. 页表初始化完成后，将 `ESP` 寄存器的值加上 `KOFFSET` 使其指向虚拟地址空间中的栈，并且使用 `ljmp` 来将 `EIP` 置为下一条指令在虚拟地址空间中的地址（注意此处有一个和编译器有关的诡异程序行为变化。若采用 `gcc-4.x` 编译，`ljmp` 会自动跳转到虚拟地址空间。而采用 `gcc-6` 编译的话，跳转地址仍然停留在原先的物理地址。在框架代码中我们做了一个简单的修补，即将跳转地址加上 `KOFFSET`，请同学们根据编译器的具体编译结果选择是否加上 `KOFFSET`）；
4. 在进入 `init_cond()` 函数继续初始化后，将调用 `init_mm()` 函数为用户进程初始化页表的内核映射部分；
5. `loader()` 的行为要发生相应的改变，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：

- 1) 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；
- 2) 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；
- 3) 把 ELF 文件中这一段 segment 的内容加载到这段物理内存；

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。

§3-3.2.3 NEMU 的升级

在开启分页机制之后，NEMU 的行为也会发生一定的改变。

首先，当然是要在 `CPU_STATE` 中添加 `CR3` 寄存器，其结构比较简单直接，查阅 i386 手册即可。

第二，改变 NEMU 对线性地址读写的处理方法。其改变集中在 `memory.c` 中对线性地址读写的处理上。在 `nemu/include/memory/mmu/page.h` 中，框架代码准备了页目录和页表项的数据结构供程序使用。查询页目录和页表将线性地址翻译为物理地址的过程对应 `nemu/src/memory/mmu/page.c` 的 `page_translate()` 函数，你需要填上 `#ifndef TLB_ENABLED` 直至 `#else` 之间的部分，这部分对应使用线性地址中的 `dir` 和 `page` 部分去查询页目录和页表，然后将页框号加上 `offset` 获取物理地址的过程。注意 `CR3` 寄存器和页目录项中保存的基地址都是物理地址。

`laddr_read()` 和 `laddr_write()` 的行为也要发生改变：当 `CR0` 的 `PG` 位被置为 1 时，需要按照上一小节中所述的方法，查询两级页表，将线性地址转换成物理地址再加以访问。注意在 `laddr_read()` 和 `laddr_write()` 中要处理地址访问跨越页边界的情形，若发生跨越页边界，则应当将一次读写拆分成两次物理地址读写来进行。在 NEMU 中，我们不会发生缺页的情况，因此，在进行 `page_translate()` 时，建议添加对页目录项和页表项中 `present==1` 的检查，若出现 `present` 为 0，则一定是页级地址转换出了问题。若不做相应检查，代码可能会变得非常难以调试。

§3-3.2.4 TLB 简述

由于页表是放在主存中的，那意味着每次进行页级地址转换时我们都要去主存访问页表，将线性地址转换为物理地址后再进行访问。这显然引入了可观的额外开销。其解决方案也非常直接，针对页表再做一个 cache。

于是便有了快表 TLB。在框架代码中我们给出了 TLB 的一个模拟实现，大家可以结合课堂内容对该实现进行理解。在实验课中，我们就不针对 TLB 设计更深入的实验内容了。

可以在 `include/config.h` 中通过定义宏 `TLB_ENABLED` 来开启 NEMU 对快表的模拟，有兴趣的同学可以观察学习相应的代码。

§3-3.3 实验过程及要求

1. 修改 Kernel 和 testcase 中 `Makefile` 的链接选项；
2. 在 `include/config.h` 头文件中定义宏 `IA32_PAGE` 并 `make clean`；
3. 在 `CPU_STATE` 中添加 `CR3` 寄存器；
4. 修改 `laddr_read()`和 `laddr_write()`，适时调用 `page_translate()`函数进行地址翻译；
5. 修改 Kernel 的 `loader()`，使用 `mm_malloc` 来完成对用户进程空间的分配；
6. 通过 `make testkernel` 执行并通过各测试用例。

在实验报告中，请回答以下问题：

1. Kernel 的虚拟页和物理页的映射关系是什么？请画图说明；
2. 以某一个测试用例为例，画图说明用户进程的虚拟页和物理页间映射关系又是怎样的？Kernel 映射为哪一段？你可以在 `loader()`中通过 `Log()`输出 `mm_malloc` 的结果来查看映射关系，并结合 `init_mm()`中的代码绘出内核映射关系。
3. “在 Kernel 完成页表初始化前，程序无法访问全局变量”这一表述是否正确？在 `init_page()`里面我们对全局变量进行了怎样的处理？

PA 4 异常、中断与 I/O

通过前三个阶段的 PA，我们已经基本构建了一个能够运算的机器的所有功能。目前为止，NEMU 只能够进行正常的控制流执行。在最后阶段，我们添加异常控制流的支持并使得 NEMU 能够实现和外设的 I/O。最终，我们希望在 NEMU 模拟器上能够运行类似仙剑奇侠传这样的小游戏。

PA 4-1 异常和中断的响应

在之前的程序执行过程中，我们都遵循着指令按序执行，遇到跳转指令则跳转到目的地址的次序来执行指令。这种执行指令的序列所得到的是正常控制流。而在程序正常执行的过程中，CPU 经常会遇到来自内部的异常事件或者外部的中断事件而打断原来程序的执行，转而执行操作系统提供的针对这些事件的处理程序。此时形成的控制流就称为异常控制流。这些打断程序正常执行的事件就称为异常或中断。

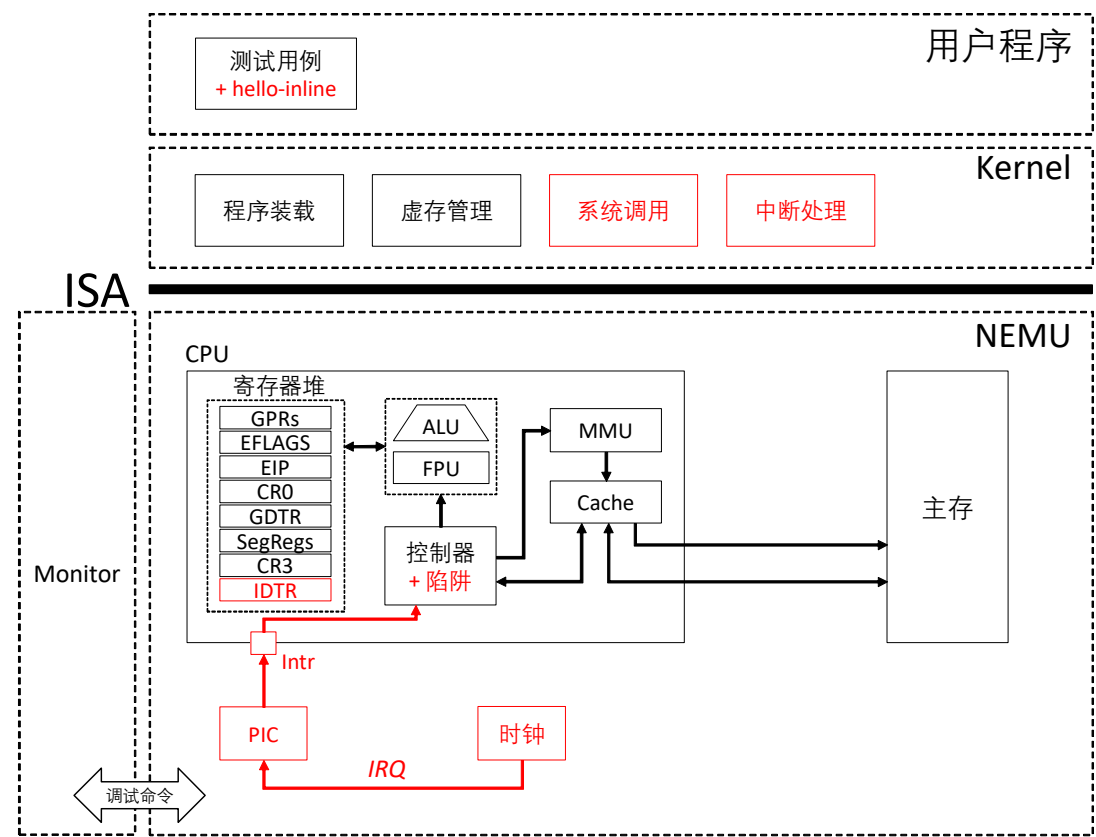


图 4-1 PA 4-1 异常和中断响应路线图

§4-1.1 预备知识

§4-1.1.1 异常和中断的类型和两阶段处理

从 80286 开始，Intel 统一把由 CPU 内部产生的意外事件，即，“内中断”称为异常；而把来自 CPU 外部的中断请求，即，“外中断”称为中断。而内部异常又分为三类：

1. 故障：与指令执行相关的意外事件，如“除数为 0”、“页故障”等；
2. 陷阱：往往用于系统调用；
3. 终止：指令执行过程中出现的严重错误。

在本实验中，我们对于内部异常，只关注“陷阱”这一类。对于“故障”和“终止”这两类异常不做模拟，若遇到

相应的情况，在 NEMU 中直接通过 `assert(0)` 强行停止模拟器运行。

异常和中断的响应和处理过程可分为两个阶段：第一阶段，CPU 对异常或中断进行响应，打断现有程序运行并调出处理程序；第二阶段，由操作系统提供的异常或中断处理程序处理完异常事件后返回用户程序继续执行。

§4-1.1.2 由硬件执行的第一阶段

其中第一阶段，CPU 对异常和中断的响应过程分为以下三个步骤：

1. 保护断点和程序状态：依次将 `EFLAGS`、`CS`、`EIP` 寄存器压栈；
2. 关中断：当异常事件是外部中断时，清除 `EFLAGS` 中的 `IF` 位；否则不清除；
3. 识别异常和中断事件并转相应的处理程序执行：根据指令或硬件给出的异常和中断类型号，查询中断描述符表（`IDT`）得到处理程序的入口地址并跳转执行；

以上三个步骤中的前两个步骤比较简单直观，第三步需要借助中断描述符表来完成。

开机后系统首先工作在实模式下。此时系统采用中断向量的方式，由 BIOS 在主存的物理地址 `0x00000 ~ 0x003FF` 的区域建立中断向量表并提供中断服务程序。当系统完成初始化进入保护模式后，中断向量表随即失效，转而使用操作系统提供的中断描述符表来处理各类异常事件。在此处我们本着简化的原则，只针对中断描述符表展开讨论。

与段表的构造非常类似，中断描述符表是一个由门描述符所构成的数组。该数组中的每一个元素（或称表项）都是一个门描述符。门描述符分为三大类：中断门描述符、陷阱门描述符和任务门描述符。具体每一种门描述符的形式请自行参阅 i386 手册。

与 `GDT` 类似，门描述符表（`IDT`）的首地址储存在一个特殊的寄存器 `IDTR` 中。当一个异常或中断到来时，CPU 根据异常或中断号，该异常或中断号可能由硬件给出（如，14 号页故障），也可能由程序给出（如，`int 0x80`）。得到异常或中断号后，CPU 根据该号码查询 `IDT`，从对应的门描述符中提取出处理程序的入口地址（虚拟地址，`selector + offset`），并跳转处理程序继续执行。更为详细的过程请仔细查阅 i386 手册 Chapter 9 的相关内容

完成上述第三步后，程序的下一条指令就位于操作系统提供的处理程序了，于是进入第二阶段，由操作系统软件完成的对异常和中断的处理。

§4-1.1.3 由软件执行的第二阶段

CPU 通过根据异常和中断号查询 `IDT` 获得处理程序的入口地址。跳转到入口地址后，操作系统提供的处理程序会完成第二阶段的处理：

1. 根据操作处理过程的需要，通过 `pusha` 等指令保存程序执行的现场；
2. 处理相应的异常或中断；
3. 处理完成后，Kernel 使用 `popa` 等指令恢复现场；
4. 通过 `iret` 指令恢复最初被保护的程序断点和状态信息，返回原程序被中断的指令（或下一条，根据保

护断点时具体保存的 EIP 决定) 继续执行。

以上便是操作系统软件在第二阶段处理异常和中断的过程，不管是内部异常还是外部中断，其过程都非常类似。

§4-1.2 代码导读和实验理解

框架代码已经实现了针对异常和中断的大部分代码。为了在实验中开启针对异常和中断的响应，我们首先需要在 `include/config.h` 头文件中定义宏 `IA32_INTR`。定义了这个宏之后，会引起 Kernel 和 NEMU 的一些行为变化，有一些作为实验的组成部分，需要我们完善当前的框架代码。

§4-1.2.1 Kernel 初始化过程

在开启 `IA32_INTR` 之后会引起 Kernel 的行为变化，

首先，在 `start.S` 的第一句，会通过 `cli` 指令关中断，直至 IDT 初始化完成后才会通过 `sti` 指令开中断。这是因为在 IDT 完成初始化前，Kernel 无法对中断进行正确的响应。

接着，Kernel 会在 `init_cond()` 函数中完成两项工作：

首先、重新初始化 GDT。因为在开启了分页机制后，原来在 `start.S` 中初始化的 GDT 已经无法访问了。在开启中断机制之前，由于可以通过各个段寄存器的隐藏部分来获取相关信息而无需查询段表，所以我们的代码没有出现问题。而开启中断机制后，在穿越门描述符时，会引起上下文切换，即将门描述的 `selector` 装载到 CS 寄存器，如此就会重新引起访问 GDT 的需要。

第二、设置 IDT。这是 Kernel 为异常和中断处理所做的核心准备工作。具体来说就是填写 IDT 中每一个门描述符，设置完毕后通过 `lidt` 指令装载 IDTR。相关的代码在 `kernel/src/irq/` 文件夹下，涉及 IDT 的初始化以及在 §4-1.1.3 中提到的第二阶段异常和中断请求处理函数的实现。

完成这两项工作后，Kernel 已经准备好要处理各类异常和中断事件了。而此时我们的 NEMU 尚未做好充分的准备。

§4-1.2.2 为 NEMU 添加异常和中断支持

为软件提供支持

为了配合 Kernel 实现异常和中断处理功能，在 NEMU 中我们也要展开相应的工作。

首先，在 Kernel 初始化 IDT 的 `init_idt()` 函数的结尾处，Kernel 要将初始化的 IDT 首地址通过 `lidt` 指令装载到 IDTR 中以供 CPU 查询使用。因此，需要添加 IDTR 和 `lidt` 指令，注意 IDTR 中存放的 IDT 首地址是线性地址；

第二，在 Kernel 处理异常和中断并返回的过程中（观察 `do_irq.S` 源文件），需要使用包括 `pusha`、`popa`、`iret` 等指令，需要添加到 NEMU 提供的指令集中。

同时，针对用户程序，需要在 NEMU 中添加 `int`、`cli` 等指令。

实现由硬件执行的第一阶段响应过程

§4-1.1.2 中提到的硬件响应异常和中断分三个主要步骤。在 NEMU 中，这三个步骤都由位于源文件 `nemu/src/cpu/intr.c` 中的 `raise_intr()` 函数来实现，其原型为：

```
void raise_intr(uint8_t intr_no) {
    // Trigger an exception/interrupt with 'intr_no'
    // 'intr_no' is the index to the IDT
    // Push EFLAGS, CS, and EIP
    // Find the IDT entry using 'intr_no'
    // Clear IF if it is an interrupt
    // Set EIP to the entry of the interrupt handler
}
```

对于内部通过 `int` 指令产生的自陷，会通过位于同一源文件中的 `raise_sw_intr()` 函数调用 `raise_intr()` 来唤出操作系统。而对于来自外部的中断，则需要 CPU 在每个指令执行的结束后查看中断引脚信号和 IRQ 请求号后调用 `raise_intr()` 来跳转到处理程序。

利用自陷实现系统调用

运行 `hello-inline` 测试用例。在该测试用例中，我们使用自陷指令 `int $0x80` 来唤出操作系统。这一条自陷指令起到的作用就是系统调用。系统调用的参数依照约定储存在通用寄存器中，其中 `eax = 4` 指出系统调用号为 4，含义是 `SYS_Write`；`ebx = 1` 是文件描述符，指出写的目标是标准输出 `stdout`；`ecx` 中保存的是待输出字符串的首地址；而 `edx` 中则保存待输出字符串的长度。

到系统执行到 `int $0x80` 自陷指令后，即获取 `intr_no = 0x80` 并执行 `raise_sw_intr()`。请注意 `raise_sw_intr()` 在调用 `raise_intr()` 前的行为。进行必要的保存和查询后，跳转到 IDT 第 0x80 号陷阱门所指向的 Kernel 准备好的处理程序。处理程序依照上述过程，保存现场并根据通用寄存器中保存的参数执行相应的处理。请依照上述描述结合框架代码，深入理解 `int 0x80` 指令所引起的控制流变化。在实现外部设备之前，我们利用 `nemu_trap` 来帮助我们实现在控制台输出字符串的功能。

在保存参数的过程中，Kernel 通过构造 `TrapFrame` 的方式将系统调用的参数传递给处理函数。在这里有 `push %esp` 一句指令，请理解它的意思。

对外部中断的响应

内部异常是在指令执行的过程中在 CPU 内部检测到的，一旦由 CPU 检测到，那么立即可以通过 `raise_sw_intr()` 来启动异常处理流程。与之相对应的，由外部设备，如时钟、键盘等引起的需要 CPU 处理的事件就对应于外部中断处理。Linux 系统处理的中断包含三种类型：I/O 中断：由 I/O 外设所发出的中断请求；时钟中断：由时钟产生的中断请求；处理器中断：多处理器系统中由其他处理器发出的中断请求。

外部中断的事件如何到达 CPU？在 Intel CPU 上设置了两个专门的引脚，分别接可屏蔽中断请求线 INTR 和不可屏蔽中断请求线 NMI。当引脚被置为高电平，即逻辑值 1，的时候，意味着有一个中断事件到来了，需要 CPU 引起关注。如果处于开中断状态，CPU 会在每一条指令执行结束后，查看中断引脚的值，若发现有中断需要处理，则查询中断号并调用相应的处理程序。

对于 I/O 中断，每一个能够发出中断请求 IRQ 的外设都有一根 IRQ 线。所有外设的 IRQ 线都连到一个可编程中断控制器（PIC，i8259 芯片）对应的 IRQ 引脚上。PIC 中每一个 IRQ 引脚都有一个对应的编号 i ， $i = 0, 1, 2, \dots$ 。当某一个外设需要发起中断时，就将自己的 IRQ 线置为 1，PIC 接收到信号后会进行一些判断，如优先级排队，判断是否被屏蔽等。若未被屏蔽，PIC 将 CPU 的 INTR 置为 1 发起中断请求。同时，PIC 会保存引起中断的中断请求号以便 CPU 来查询。按照 IA-32 的约定，32 号以上的中断类型为可屏蔽中断和软中断。因此，PIC 保留的中断号就是引起中断的引脚号加上 32，即， $\text{intr_no} = \text{irq_no} + \text{IRQ_BASE}$ 。其中的 IRQ_BASE 取值为 32，其取值的依据是根据 IA-32 的约定而来，可根据课本第 304 页的表 7.1 来进行理解。

在 NEMU 中，本着 KISS 原则我们简化了 i8259 的实现，略去了其可编程的部分。在实现外部 I/O 设备之前，我们将时钟作为一种特殊的外设，来尝试实现 CPU 对外部中断的响应和 Kernel 对中断的处理。

NEMU 中与时钟中断相关的代码主要涉及 `nemu/src/device/sdl.c` 和 `nemu/src/device/dev/timer.c` 这两个源文件。在 `sdl.c` 源文件中，我们使用 SDL 库实现包括时钟、图形显示、键盘输入捕获在内的功能。在 `init_sdl()` 函数中，我们开启了一个线程，其对应的执行函数为 `NEMU_SDL_Thread()`。在该线程负责显示以及键盘输入相关设备的初始化，并以 100Hz 的频率循环触发时钟中断、刷新屏幕、扫描键盘输入直至程序退出。在实现设备功能之前，我们只关注和时钟有关的内容。在 `NEMU_SDL_Thread()` 线程内，我们会以 100Hz 的频率循环调用 `timer_intr()` 函数触发时钟中断。`timer_intr()` 函数定义在源文件 `timer.c` 中，该函数通过 i8259 PIC 提交时钟中断请求信号。时钟中断请求的 `irq_no` 为 0。PIC 接收到信号后，在 `irq_no` 的基础上加上 IRQ_BASE 得到新中断号 `intr_no`，并将 CPU 的 INTR 引脚置为 1。

CPU 在结束当前指令的执行后，查看 EFLAGS 的 IF 位和 INTR 引脚，若是开中断状态且有中断到来，则向 PIC 查询中断号。利用中断号调用在上一小节中实现的 `raise_intr()` 函数，注意传参时使用的 `irq_no` 是通过 PIC 增加了 IRQ_BASE 后得到的值。如此，能够实现将时钟中断一直送达 Kernel 中时钟中断处理程序处的目的。这一过程在代码中实现在 `nemu/src/cpu/cpu.c` 的 `exec()` 函数中 `while` 循环体的末尾，`do_intr()` 函数中，你需要根据提示在合理的地方调用这个函数。

通过上述过程，我们得以一窥外部设备将异常事件送入 CPU 并引发操作系统进行处理的过程。当然，如键盘、磁盘这样的外部设备，除了提醒 CPU 有事件到达之外，还需要完成设备和 CPU、主存之间的数据交换才能实现其相应的功能。这就涉及到下一节我们要谈到的 I/O 设备的模拟。在这里，我们只要把中断请求送入 CPU，并正确唤出操作系统的处理函数就算成功了。

在成功添加时钟中断后，会触发 Kernel 中的 `panic`。找到这个 `panic`，并理解控制流是如何从 `NEMU_SDL_Thread()` 到达这个 `panic` 处的。理解之后，可删除 `panic` 继续执行。

§4-1.3 实验过程及要求

§4-1.3.1 通过自陷实现系统调用

1. 在 `include/config.h` 中定义宏 `IA32_INTR` 并 `make clean`；
2. 在 `nemu/include/cpu/reg.h` 中定义 `IDTR` 结构体，并在 `CPU_STATE` 中添加 `idtr`；
3. 实现包括 `lidt`、`cli`、`sti`、`int`、`pusha`、`popa`、`iret` 等指令；
4. 在 `nemu/src/cpu/intr.c` 中实现 `raise_intr()` 函数；
5. 执行 `hello-inline` 测试用例并看到屏幕输出

nemu trap output: Hello, world!

除了上述代码实验，在实验报告中还需要回答如下问题：

1. 详细描述从测试用例中的 `int $0x80` 开始一直到 `HIT_GOOD_TRAP` 为止的详细的系统行为（完整描述控制的转移过程，即相关函数的调用和关键参数传递过程），可以通过文字或画图的方式来完成；
2. 在描述过程中，回答 `kernel/src/irq/do_irq.S` 中的 `push %esp` 起什么作用，画出在 `call irq_handle` 之前，系统栈的内容和 `esp` 的位置，指出 `TrapFrame` 对应系统栈的哪一段内容。

§4-1.3.2 响应时钟中断

1. 在 `include/config.h` 中定义宏 `HAS_DEVICE_TIMER` 并 `make clean`；
2. 在 `nemu/include/cpu/reg.h` 的 `CPU_STATE` 中添加 `uint8_t intr` 成员，模拟中断引脚；
3. 在 `nemu/src/cpu/cpu.c` 的 `init_cpu()` 中初始化 `cpu.intr = 0`；
4. 在 `nemu/src/cpu/cpu.c` 的 `exec()` 函数 `while` 循环体，每次执行完一条指令后调用 `do_intr()` 函数查看并处理中断事件；
5. 执行 `make testkernel`；
6. 触发 Kernel 中的 `panic`，找到该 `panic` 并移除。

在实验报告中，需要回答：

1. 详细描述 NEMU 和 Kernel 响应时钟中断的过程和先前的系统调用过程不同之处在哪里？相同的地方又在哪里？可以通过文字或画图的方式来完成。

PA 4-2 外设与 I/O

在之前的几个阶段中，我们模拟了 CPU 的运算、指令执行；存储管理；异常和中断响应的相关功能。我们的模拟器功能已日趋完备！现在，我们可以向一台完整的计算机迈出最后的一步了。我们要在模拟器中增加与外部设备进行 I/O 的功能。如此我们的模拟器就能够实现包括键盘输入、屏幕输出等功能，能够与用户互动起来，完成除了运算以外更加丰富的功能。

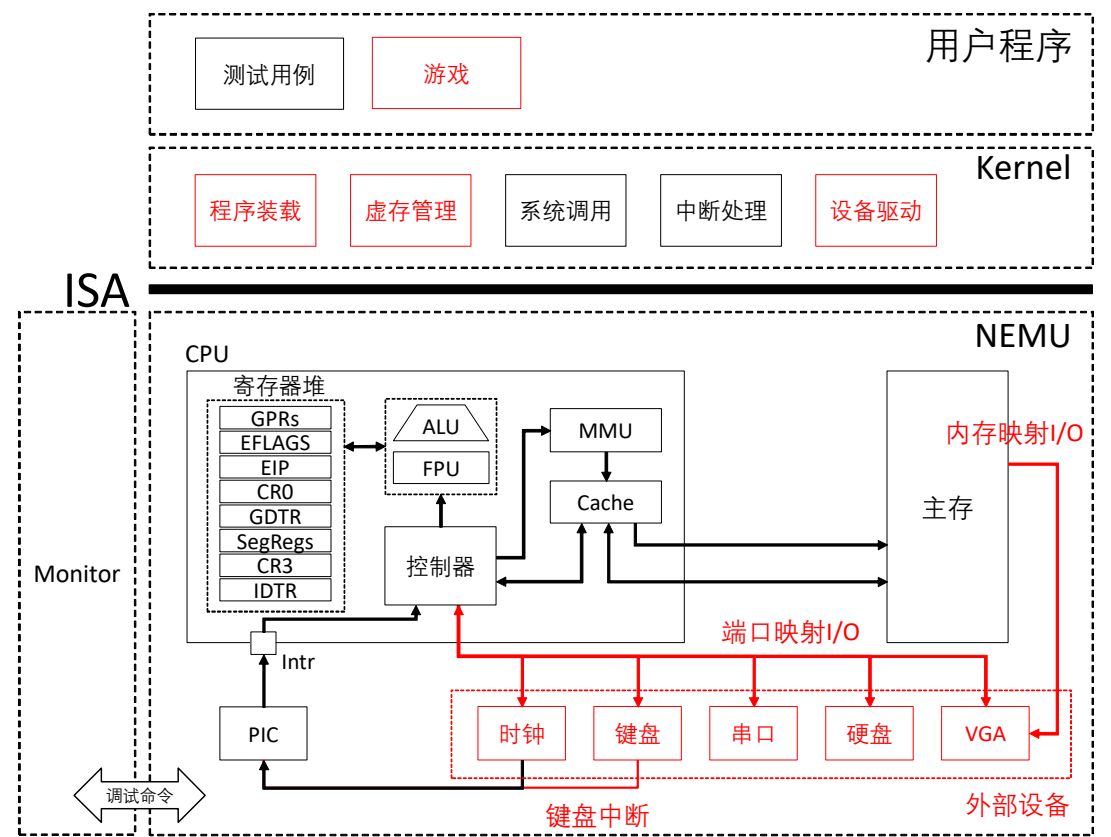


图 4-2 PA 4-2 外设与 I/O 路线图

§4-2.1 预备知识

要完成与外设的 I/O，核心要解决两个问题：

1. 与谁进行 I/O？
2. I/O 的内容是什么？

对于第一个问题的回答涉及到 I/O 寻址的方式。一种 I/O 寻址方式是端口映射 I/O（Port-mapped I/O）。端口映射 I/O 的方式相当于打电话：先拨号，再通话。在计算机中有一系列的设备控制器，这些控制器一头通过 I/O 总线与主机相连，另一头则连接着被控制的外设。设备控制器中包含一系列的寄存器：控制寄存器，用于存放主机送来的控制信号；状态寄存器，用于存放设备状态如就绪和错误信息；数据缓冲寄存器，用于临时存放主机和设备间需要交换的数据信息。通常，把以上三类寄存器统称为 I/O 端口。IA-32 共定义了 65536 个 8 位的 I/O 端口。在端口映射的 I/O 方式下，CPU 通过专门的 I/O 指令 `in (ins)` 和 `out (outs)`

来对某一个端口进行读和写。市场上的计算机绝大多数都是 IBM PC 兼容机，IBM PC 兼容机对常见设备端口号的分配有专门的规定。当然我们也可以自定义端口号与外设之间的分配和对应关系。简言之，通过对某一特定端口的读写，就可以完成 CPU 和某特定外设之间的数据交换。至于交换的数据是控制命令、状态还是数据，则不是 CPU 所关心的了。

另一种 I/O 寻址方式是内存映射 I/O(memory-mapped I/O)。这种寻址方式将一部分物理内存映射到 I/O 设备空间中，使得 CPU 可以通过普通的访存指令来访问设备。这种物理内存的映射对 CPU 是透明的，CPU 觉得自己是在访问内存，但实际上可能是访问了相应的 I/O 空间。这样以后，访问设备的灵活性就大大提高了。一个例子是物理地址区间 [0xa0000, 0xc0000)，这段物理地址区间被映射到 VGA 内部的显存，读写这段物理地址区间就相当于对读写 VGA 显存的数据。例如：

```
memset((void *)0xa0000, 0, SCR_SIZE);
```

会将显存中一个屏幕大小的数据清零，即往整个屏幕写入黑色像素，作用相当于清屏。

对于第二个问题，即具体怎么通信，则不是我们要关心的内容了。正如在端口映射 I/O 最后我们谈到的那样，CPU 只会忠实地将程序给出的字节交给指定的端口，或从指定端口读出字节。至于这些字节数据如何解释，是软件（设备驱动程序）该关心的事情，和计算机无关。计算机通过上述 I/O 寻址方式打通了 CPU 与外设之间的桥梁，接下来就是设备制造商和设备用户之间通过约定各种通信和控制协议来进行下一步的沟通了。

§4-2.2 代码导读和实验理解

目前在 NEMU 中我们模拟了四种外设：串口（Serial）、硬盘（IDE）、显示器（VGA）和键盘（Keyboard）。所有和设备相关的代码都可以在 `nemu/include/device` 和 `nemu/src/device` 文件夹下找到。

§4-2.2.1 端口映射 I/O 模拟

在 `nemu/src/device/io/port_io.c` 源文件中有一个 `pio_handler_table` 数组，其中维护了端口和外设之间的映射关系，即，端口号 `port` 和处理函数 `handler` 之间的对应关系。同时提供了两个接口函数 `pio_write()` 和 `pio_read()` 供 `out` 和 `in` 指令调用。`pio_handler_table` 数组建立了某一个端口及其处理函数（`handler`）之间的对应关系，当对某一个端口进行读写时，会调用该端口对应的处理函数来模拟设备的相应动作。在 `nemu/include/device/port_io.h` 头文件中我们提供了宏 `make_pio_handler` 来帮助声明处理函数。

为了使得 CPU 能够通过端口映射的 I/O 与外设进行通信，需要实现 `in` 和 `out` 两条指令。

模拟串口

对串口进行模拟时，我们约定规定串口的端口为从 0x3F8 开始的连续 8 个端口。对 0x3F8 端口进行 1 字节的写操作对应应在控制台上输出一个字符（ASCII 码编码）。

在 NEMU 中，要开启对串口的模拟，需要在 `include/config.h` 中定义宏 `HAS_DEVICE_SERIAL`。定义该宏后，NEMU 会在每次 `restart` 时初始化 `serial`。最关键的，它会改变 Kernel 进行 `fs_write()` 时的行为。在此之前，Kernel 都是通过调用 `opcode` 为 0x82 的 `nemu_trap` 指令来帮助完成输出的。这条指令在真实的机器中并不存在，没有输出设备也不可能实现字符串的输出，借用 `nemu_trap` 指令来帮助输出是 NEMU 为了调试

和实验方便而设计的一个权宜之计，因此每次通过 `nemu_trap` 指令来帮助输出时我们都会在控制台上打出醒目的 `nemu trap output:` 这一标记。

在实现了 `serial` 之后，我们就可以停止使用 `nemu_trap` 指令的输出功能了，在 `fs_write()` 中调用 `serial_printc()` 来通过向 `0x3F8` 端口写字节的方式在控制台上一个一个地输出字符。自然地，`nemu trap output:` 这一标记也就消失了。在对应的实验中，我们需要你帮助实现 `serial_printc()` 函数的功能，将传入的参数 `ch` 通过 `serial` 输出到控制台上。在 `kernel/include/x86/io.h` 中框架代码已经封装好了 `in` 和 `out` 指令的调用，请选择合适的函数调用。

在完成对 `serial` 的模拟后，再次运行 `hello-inline` 测试用例，可以看到输出内容前的 `nemu trap output:` 标记消失了。在充分理解了上一阶段实验的 `hello-inline` 测试用例的工作原理后，相信对于输出的过程已经有了全面的认识，对于实现模拟串口后输出方式的改变，也应当能够充分理解。

模拟硬盘

在对硬盘展开模拟时，我们将注意力主要集中在端口映射的 I/O 这一方式上，采用 DMA 方式的磁盘读写操作暂不涉及，有兴趣的同学可以自学。在端口映射的 I/O 方式下，约定硬盘设备对应的端口号为 `0x1F0` 开始的连续 8 个端口，其中 `0x1F0` 为数据端口，当对磁盘进行读写时，从 `0x1F0` 开始的 4 个字节都可以作为数据端口使用；`0x1F7` 为控制和状态端口，用于标识硬盘处于读 (`0x20`)、写 (`0x30`) 或空闲 (`0x40`) 状态；在进行读写之前，需要进行磁盘准备工作，向 `0x1F3 ~ 0x1F6` 这四个端口写入要读或写的扇区号（小端方式），每个扇区大小为 512 字节。在使用磁盘进行读写时，首先向磁盘 `0x1F3 ~ 0x1F6` 控制寄存器写入要读写的扇区号；第二向 `0x1F7` 控制寄存器写入读或写控制命令；最后可以通过读写数据端口实现对磁盘对应扇区 512 字节的读写操作。

在 NEMU 中，通过在 `include/config.h` 中定义宏 `HAS_DEVICE_IDE` 来开启对硬盘的模拟。在 `nemu/src/device/dev/ide.c` 中实现了对硬盘的模拟。其中的 `init_ide()` 函数供 NEMU 在初始化时调用，将模拟的硬盘设备关联到 Linux 系统中的一个文件，比如测试用例的 ELF 文件。对于模拟硬盘的读写就对应文件的读写。`handler_ide()` 处理函数实现了硬盘对控制信号的响应，响应的方式根据端口对应的数据或控制寄存器值的约定实现，请结合代码和上面一小段的描述进行理解。

在 Kernel 中的 `kernel/src/driver/ide` 文件夹中，包含了对硬盘驱动程序的封装。`disk.c` 源文件中提供的 `ide_read()` 和 `ide_write()` 函数是磁盘驱动程序对外提供的接口，供其它模块调用实现对磁盘的读写。其中实现了对磁盘的 cache 等功能，具体在教程中就不展开论述了。感兴趣的同学可以结合代码深入理解相应的功能。

在实现了硬盘的模拟之后，我们可以让 ram disk 退休了。在定义了 `HAS_DEVICE_IDE` 宏之后，在 NEMU 每次重启时都会通过 `init_ide()` 将测试用例的 ELF 文件挂载到模拟硬盘。随着 ram disk 的退休和 ide disk 的开启，需要改变 Kernel 装载程序的 `loader()`，使其从 ide disk 装载 ELF 文件。当然在 NEMU 中，Kernel 还是通过 image 镜像的方式进行加载的。到操作系统实验部分，大家会进一步了解，其实在开机之后内存中只会加载一个很小的引导程序 image。通过引导程序装载 Kernel 之后，再由 Kernel 装载用户程序。在计算机系统基础实验课中，我们省略了前面的步骤，将整个 Kernel 的镜像作为一个大引导程序整个装载到内存中。

模拟 VGA 显示器的控制端口

VGA 的显示采用内存映射 I/O 的方式，但是也有一些控制命令需要通过端口进行发送。在 NEMU 中，我们

已经实现了 VGA 这一部分的功能，只需结合代码大致理解即可。VGA 实验的重点在于下一节中要实现的内存映射 I/O 的模拟。

模拟键盘

在 NEMU 中，通过在 `include/config.h` 中定义宏 `HAS_DEVICE_KEYBOARD` 来开启对键盘的模拟。对键盘展开模拟时，键盘事件首先在 `nemu/src/device/sdl.c` 中由 `NEMU_SDL_Thread()` 线程捕获。NEMU 捕获两类事件：键盘按下和抬起。当检测到相应事件后，将对应键的扫描码作为参数传送给 `keyboard.c` 中的模拟键盘函数。模拟键盘缓存扫描码，并通过中断请求的方式通知 CPU 有按键或抬起的事件，键盘的中断请求号为 1。CPU 收到中断请求后调用 Kernel 的中断响应程序。在响应程序中，Kernel 会查找是否有应用程序注册了对键盘事件的响应，若有，则通过调用注册的响应函数的方式来通知应用程序。此时在应用程序的键盘响应函数中，可以通过 `in` 指令从键盘的数据端口读取扫描码完成数据交换。键盘数据端口约定为 `0x60`，键盘扫描码的编码方式参照这个⁶约定。

使用 SDL 来捕获键盘事件需要有一个窗口，在窗口中进行的键盘输入才会被 SDL 捕获。因此在开启键盘模拟后，会跳出一个全黑的窗口。在开启 VGA 模拟之前，这个窗口不会输出任何信息，只能接收来自键盘的输入信息。

在实现了模拟键盘后，我们能开启最后一个测试用例 `echo`。运行 `echo`，每次在窗口中键入一个英文字母，就会在标准输出中打印一个对应的大写字母。在 `echo` 启动后，它会向 Kernel 注册监听键盘事件，并在每一个键盘事件到来后，判断是否进行输出。在 `echo.c` 测试用例的代码中，我们其实在一个 `.c` 文件中包含了原本应该分散在应用程序、库文件、操作系统头文件、设备驱动程序中的各种功能。从软件工程和程序设计角度来说，`echo` 的写法不是很好：各个抽象层次、功能模块的代码全都紧耦合在一起。以后写代码千万不能照着这种泛型去写！但是仅仅从课程教学的角度来说，它在尽可能少的代码中包装了尽可能多的知识点。读懂 `echo` 的代码并回答实验问题，是对中断响应、系统调用和端口映射 I/O 的一次全面的回顾。

可以通过关闭窗口或在控制台 `Ctrl-c` 的方式退出 `echo`。

S4-2.2.2 内存映射 I/O 模拟

可以通过在 `include/config.h` 中定义宏 `HAS_DEVICE_VGA` 来开启对 VGA 的模拟。对显示器（VGA）进行模拟时，我们采用内存映射 I/O 的方式来向显示器发送待显示的数据。我们约定内存从物理地址 `0xa0000` 开始，长度为 `320 * 200` 字节的区间为显存区间。

在 `nemu/src/memory/memory.c` 中，为物理地址读写的函数添加是否是 `mm_io` 的判断。通过 `is_mmio()` 函数判断一个物理地址是否被映射到 I/O 空间。如果是，`is_mmio()` 会返回映射号，否则返回 -1。内存映射 I/O 的访问需要调用 `mmio_read()` 或 `mmio_write()`，调用时需要提供映射号。如果不是内存映射 I/O 的访问，就访问主存。

为用户进程创建 video memory 的虚拟地址空间。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。具体的，你需要定义一些页表项（注意页表需要按页对齐，你可以参考

⁶ http://www.mouseos.com/os/doc/scan_code.html

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。如果创建地址空间和内存映射 I/O 的实现都正确, 你会看到屏幕上输出了一些测试时写入的颜色信息, 同时 video_mapping_read_test() 将会通过检查。

§4-2.3 实验过程及要求

§4-2.3.1 完成串口的模拟

1. 在 include/config.h 中定义宏 HAS_DEVICE_SERIAL 并 make clean ;
2. 实现 in 和 out 指令 ;
3. 实现 serial_printc() 函数 ;
4. 运行 hello-inline 测试用例, 对比实现串口前后的输出内容的区别。

§4-2.3.2 通过硬盘加载程序

1. 在 include/config.h 中定义宏 HAS_DEVICE_IDE 并 make clean ;
2. 修改 Kernel 中的 loader(), 使其通过 ide_read() 和 ide_write() 接口实现从模拟硬盘加载用户程序 ;
3. 通过 make testkernel 执行测试用例, 验证加载过程是否正确。

§4-2.3.3 完成键盘的模拟

1. 在 include/config.h 中定义宏 HAS_DEVICE_KEYBOARD 并 make clean ;
2. 通过 make testkernel 运行 echo 测试用例 ;

在实验报告中, 结合代码详细描述 :

1. 注册监听键盘事件是怎么完成的 ?
2. 从键盘按下一个键到控制台输出对应的字符, 系统的执行过程是什么? 如果涉及与之前报告重复的内容, 简单引用之前的内容即可。

§4-2.3.4 实现 VGA 的 MMIO

1. 在 include/config.h 中定义宏 HAS_DEVICE_VGA ;
2. 在 nemu/src/memory/memory.c 中添加 mm_io 判断和对应的读写操作 ;
3. 在 kernel/src/memory/vmem.c 中完成显存的恒等映射 ;
4. 通过 make testkernel 执行测试用例, 观察输出测试颜色信息, 并通过 video_mapping_read_test()。

PA 4-3 可选任务：游戏移植

如果通过了上述所有的阶段，那么我们就可以为我们的 PA 加入最后的拼图，移植打字小游戏和仙剑奇侠传了。

§4-3.1 移植打字小游戏

框架代码中的 `game` 目录下包含两款游戏，共用的部分存放在 `game/src/common` 目录下，游戏各自的逻辑分别存放在 `game/src/typing` 和 `game/src/nemu-pal` 中。可以通过修改 `game/Makefile` 中的 `GAME` 变量在两个游戏之间切换(需要重新编译)。

打字小游戏来源于 2013 年 oslab0 的框架代码，为了配合移植，代码的结构做了少量调整，同时去掉了和显存优化相关的部分。

我们对游戏的初始化部分进行一些说明：

游戏入口是 `game/src/common/main.c` 中的 `game_init()` 函数。

`init_timer()` 函数用于设置 100Hz 的时钟频率，但由于 NEMU 中的时钟模拟实现是不可编程的，而且模拟实现的时钟的默认频率就是 100Hz，故此处的 `init_timer()` 函数并没有实际作用。

在游戏中，`add_irq_handle()` 是一个人为添加的系统调用，其系统调用号是 0，用于注册一个中断处理函数。已经注册的中断处理函数会在相应中断到来的时候被内核调用，这样游戏代码就可以通过中断来控制游戏的逻辑了。但在真实的操作系统中，提供这样的系统调用是非常危险的：恶意程序可以注册一个陷入死循环的中断处理函数，由于操作系统处理中断的时候，处理器一般都处于关中断状态，若此时陷入了死循环，操作系统将彻底崩溃。

使用 `Log()` 宏输出一句话。在游戏中，通过 `Log()` 宏输出的信息都带有 `{game}` 的标签，方便和 `kernel` 中的 `Log()` 宏输出区别开来。

进入游戏逻辑主循环。整个游戏都在中断的驱动下运行。

在工程目录下运行 `make testgame` 命令编译游戏。

如果前面的实现都正确，那么原则上你可以直接运行打字小游戏。

§4-3.2 移植仙剑奇侠传

原版的仙剑奇侠传是针对 Windows 平台开发的，因此它并不能在 GNU/Linux 中运行(你知道为什么吗?)，也不能在 NEMU 中运行。网友 weimingzhi 开发了一款基于 SDL 库，跨平台的仙剑奇侠传，工程叫 SDLPAL。你可以通过 `git clone` 命令把 SDLPAL 克隆到本地，然后把仙剑奇侠传的数据文件(我们已经把数据文件上传到提交网站上)放在工程目录下 (`game/src/nemu-pal/`)，执行 `make` 编译 SDLPAL，编译成功后就可以玩了。更多的信息请参考 SDLPAL 工程中的 README 说明。

把仙剑奇侠传移植到 NEMU 中的主要工作，就是把应用层之下提供给仙剑奇侠传的所有 API 重新实现一遍，因为这些 API 大多都依赖于操作系统提供的运行时环境，我们需要根据 NEMU 和 `kernel` 提供的运行时环境重写它们。主要包括以下三部分内容：

-
1. C 标准库
 2. SDL 库
 3. 文件系统

`newlib` 已经提供了 C 标准库的功能, 因此我们可以很简单地对这两部分内容进行移植, 重点则落到了 SDL 库和文件系统的移植工作中.

我们把待移植的仙剑奇侠传称为 NEMU-PAL. NEMU-PAL 在 SDLPAL 的基础上经过少量修改得到, 包括去掉了声音, 修改了 `game/src/nemu-pal/device/input.c` 中和按键相关的处理, 把我们关心的和 SDL 库的实现整理到 `game/src/nemu-pal/hal` 目录下, 一些我们不必关心的实现则整理到 `game/src/nemu-pal/unused` 目录下.

为了编译 NEMU-PAL, 你需要修改 `game/Makefile` 中的 `GAME` 变量, 从打字小游戏切换到 NEMU-PAL. 然后把仙剑奇侠传的数据文件放在 `game/src/nemu-pal/data` 目录下, 工程目录下执行 `make testgame` 即可.

下面来谈谈移植工作具体要做什么. 在这之前, 请确保你已经理解打字小游戏的工作方式.

重写 SDL 库的 API

在 SDLPAL 中, SDL 库负责时钟, 按键, 显示和声音相关的处理. 由于在 NEMU 中没有模拟声卡的实现, NEMU-PAL 已经去掉了和声音相关的部分. 其余三部分的内容被整理到 `game/src/nemu-pal/hal` 目录下, 其中 HAL(Hardware Abstraction Layer)是硬件抽象层的意思, 和硬件相关的功能将在 HAL 中被打包, 提供给上层使用.

时钟相关

1. `SDL_GetTicks()` 用于返回用毫秒表示的当前时间(从运行游戏时开始计算).
2. `SDL_Delay()` 用于延迟若干毫秒.
3. `jiffy` 变量记录了时钟中断到来的次数, 通过它可以实现上述和时钟相关的控制功能.

键盘相关

键盘通常都支持"重复按键", 即若一直按着某一个键不松开, 键盘控制器将会不断发送该键的扫描码. 但是 SDLPAL(包括待移植的 NEMU-PAL)的游戏逻辑是在基于"非重复按键"的特性编写的, 即若一直按着某一个键不松开, SDLPAL 只会收到一次该键的扫描码. 因此 HAL 需要把键盘的"重复按键"特性屏蔽起来, 向上层提供"非重复按键"的特性.

1. 实现这一抽象的方法是记录按键的状态. 你需要在键盘中断处理函数 `keyboard_event()` 中编写相应代码, 根据从键盘控制器得到的扫描码记录按键的状态.
2. `process_keys()` 函数会被 NEMU-PAL 轮询调用. 每次调用时, 寻找一个刚刚按下或刚刚释放的按键, 并调用相应的回调函数, 然后改变该按键的状态. 若找到这样的按键, 函数马上返回 `true`; 若找不到, 函数返回 `false`. 注意返回之前需要打开中断.
3. 代码中提供了数组的实现方式用于记录按键的状态, 你也可以使用其它方式来实现上述抽象.

显示相关

SDL 中包含很多和显示相关的 API, 为了重写它们, 你首先需要了解它们的功能. 通过 `man` 命令查阅以下内容:

- `SDL_Surface`
- `SDL_Rect`
- `SDL_BlitterSurface`
- `SDL_FillRect`
- `SDL_UpdateRect`

在 `game/src/nemu-pal/include/hal.h` 中已经定义了相关的结构体, 你需要阅读 `man`, 了解相关成员的功能, 然后实现 `game/src/nemu-pal/hal/video.c` 中相应函数的功能. 你可以忽略 `man` 中提到的"锁"等特性, 我们并不打算在 NEMU-PAL 中实现这些特性.

实现简易文件系统

对于大部分游戏来说, 游戏用到的数据所占的空间比游戏逻辑本身还大, 因此这些数据一般都存储在磁盘中. IDE 驱动程序已经为我们屏蔽了磁盘的物理特性, 并提供了读写接口, 使得我们可以很方便地访问磁盘某一个位置的数据. 但为了易于上层使用, 我们还需要提供一种更高级的抽象, 那就是文件.

文件的本质就是字节序列, 另外还由一些额外的属性构成. 在这里, 我们只讨论磁盘上的文件. 这样, 那些额外的属性就维护了文件到磁盘存储位置的映射. 为了管理这些映射, 同时向上层提供文件操作的接口, 我们需要在 kernel 中实现一个文件系统.

不要被"文件系统"四个字吓到了, 我们需要实现的文件系统并不是那么复杂, 这得益于 NEMU-PAL 的一些特性: 对于大部分数据文件来说, NEMU-PAL 只会读取它们, 而不会对它们进行修改; 唯一有可能进行文件写操作的, 就只有保存游戏进度, 但游戏存档的大小是固定的. 因此我们得出了一个重要的结论: 我们需要实现的文件系统中, 所有文件的大小都是固定的. 既然文件大小是固定的, 我们自然也可以把每一个文件分别固定在磁盘中的某一个位置. 这些很好的特性大大降低了文件系统的实现难度, 当然, 真实的文件系统远远比这个简易文件系统复杂.

我们约定磁盘的最开始用于存放 NEMU-PAL 游戏程序, 从 1MB 处开始一个挨着一个地存放数据文件:

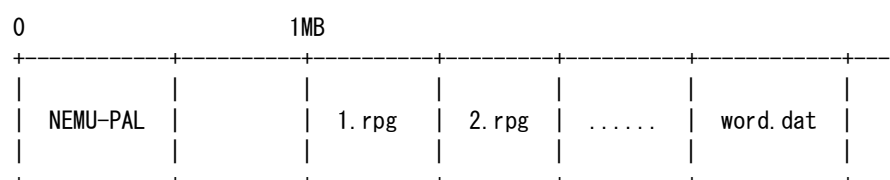


图 4-3 简易文件系统

`kernel/src/fs/fs.c` 中已经列出了所有数据文件的信息, 包括文件名, 文件大小和文件在磁盘上的位置. 但若只有这些信息, 文件系统还是不能表示文件在读写时的动态信息, 例如读写位置的指针等. 为此, 文件系统需要为那些打开了的文件维护一些动态的信息:

```
typedef struct {
```

```
bool opened;
uint32_t offset;
} Fstate;
```

在这里, 我们只需要维护打开状态 `opened` 和读写指针 `offset` 即可. 由于这个简易文件系统文件数目是固定的, 我们可以为每一个文件静态分配一个 `Fstate` 结构, 因此我们只需要定义一个长度为 `NR_FILES + 3` 的 `Fstate` 结构数组即可. 这里的 3 包括 `stdin`, `stdout`, `stderr` 三个特殊的文件, 磁盘中的第 `k` 个文件固定使用第 `k + 3` 个 `Fstate` 结构. 这样, 我们就可以把 `Fstate` 结构在数组中的下标作为相应文件的文件描述符(`fd`, `file descriptor`)返回给用户层了.

有了 `Fstate` 结构之后, 我们就可以实现以下的文件操作了:

```
int fs_open(const char *pathname, int flags); /* 在我们的实现中可以忽略 flags */

int fs_read(int fd, void *buf, int len);

int fs_write(int fd, void *buf, int len);

int fs_lseek(int fd, int offset, int whence);

int fs_close(int fd);
```

这些文件操作实际上是相应的系统调用在内核中的实现, 你可以通过 `man` 查阅它们的功能, 例如

`man 2 open`

其中 2 表示查阅和系统调用相关的 `man page`. 实现这些文件操作的时候注意以下几点:

1. 由于简易文件系统中每一个文件都是固定的, 不会产生新文件, 因此“`fs_open()` 没有找到 `pathname` 所指示的文件”属于异常情况, 你需要使用 `assertion` 终止程序运行.
2. 使用 `ide_read()` 和 `ide_write()` 来进行文件的读写.
3. 由于文件的大小是固定的, 在实现 `fs_read()` 和 `fs_lseek()` 的时候, 注意读写指针不要越过文件的边界.
4. 除了写入 `stdout` 和 `stderr` 之外(即输出到串口), 其余对于 `stdin`, `stdout` 和 `stderr` 这三个特殊文件的操作可以直接忽略.

最后你还需要在 `kernel` 中编写相应的系统调用, 来调用相应的文件操作, 同时修改 `game/src/common/lib/syscall.c` 中的代码, 为用户进程开放系统调用接口. 在完成了所有的工作, `fix` 了所有导致程序崩溃的 `bug` 之后, 你将能够成功运行仙剑奇侠传!



I386 手册勘误

17.2.1 Table 17-3:

@@ -?,2 +?,2 @@									
disp8[EDX]	010	42	4A	52	5A	62	6A	72	7A
-disp8[EPX]	011	43	4B	53	5B	63	6B	73	7B
+disp8[EBX]	011	43	4B	53	5B	63	6B	73	7B

17.2.1 Table 17-4:

@@ -?,2 +?,2 @@									
Base =		0	1	2	3	4	5	6	7
- r32		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
+ r32		EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
@@ -?,2 +?,2 @@									
[ECX*2]	001	48	49	4A	4B	4C	4D	4E	4F
-[ECX*2]	010	50	51	52	53	54	55	56	57
+ [EDX*2]	010	50	51	52	53	54	55	56	57
@@ -?,2 +?,2 @@									
[EDX*4]	010	90	91	92	93	94	95	96	97
-[EBX*4]	011	98	89	9A	9B	9C	9D	9E	9F
+ [EBX*4]	011	98	99	9A	9B	9C	9D	9E	9F
@@ -?,2 +?,2 @@									
NOTES:									
- [*] means a disp32 with no base if MOD is 00, [ESP] otherwise.									
This provides the following addressing modes:									
+ [*] means a disp32 with no base if MOD is 00. Otherwise, [*] means disp8[EBP] or disp32[EBP].									
This provides the following addressing modes:									

17.2.2.11 Instruction Set Detail 中的 DEC -- Decrement by 1

@@ -?,2 +?,2 @@			
FF /1	DEC r/m16	2/6	Decrement r/m word by 1
-	DEC r/m32	2/6	Decrement r/m dword by 1
+FF /1	DEC r/m32	2/6	Decrement r/m dword by 1

17.2.2.11 Instruction Set Detail 中的 INC -- Increment by 1

@@ -?,2 +?,2 @@			
FF /0	INC r/m16		Increment r/m word by 1
-FF /6	INC r/m32		Increment r/m dword by 1
+FF /0	INC r/m32		Increment r/m dword by 1

17.2.2.11 Instruction Set Detail 中的 Jcc -- Jump if Condition is Met

@@ -?,2 +?,2 @@				
72	cb	JB rel8	7+m,3	Jump short if below (CF=1)
-76	cb	JBE rel8	7+m,3	Jump short if below or (CF=1 or ZF=1)
+76	cb	JBE rel8	7+m,3	Jump short if below or equal (CF=1 or ZF=1)
@@ -?,2 +?,2 @@				
7C	cb	JL rel8	7+m,3	Jump short if less (SF!=OF)
-7E	cb	JLE rel8	7+m,3	Jump short if less or equal (ZF=1 and SF!=OF)
+7E	cb	JLE rel8	7+m,3	Jump short if less or equal (ZF=1 or SF!=OF)
@@ -?,2 +?,2 @@				
0F	8C cw/cd	JL rel16/32	7+m,3	Jump near if less (SF!=OF)
-0F	8E cw/cd	JLE rel16/32	7+m,3	Jump near if less or equal (ZF=1 and SF!=OF)
+0F	8E cw/cd	JLE rel16/32	7+m,3	Jump near if less or equal (ZF=1 or SF!=OF)

17.2.2.11 Instruction Set Detail 中的 MOV -- Move Data

@@ -?,7 +?,7 @@				
A3		MOV moffs32,EAX	2	Move EAX to (seg:offset)
-B0 + rb		MOV reg8,imm8	2	Move immediate byte to register
-B8 + rw		MOV reg16,imm16	2	Move immediate word to register
-B8 + rd		MOV reg32,imm32	2	Move immediate dword to register
-Ciiii		MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
-C7		MOV r/m16,imm16	2/2	Move immediate word to r/m word
-C7		MOV r/m32,imm32	2/2	Move immediate dword to r/m dword
+B0 + rb ib		MOV reg8,imm8	2	Move immediate byte to register
+B8 + rw iw		MOV reg16,imm16	2	Move immediate word to register
+B8 + rd id		MOV reg32,imm32	2	Move immediate dword to register
+C6 ib		MOV r/m8,imm8	2/2	Move immediate byte to r/m byte
+C7 iw		MOV r/m16,imm16	2/2	Move immediate word to r/m word
+C7 id		MOV r/m32,imm32	2/2	Move immediate dword to r/m dword

17.2.2.11 Instruction Set Detail 中的 MUL -- Unsigned Multiplication of AL or AX

@@ -?,2 +?,2 @@				
Flags Affected				
-OF and CF as described above; SF, ZF, AF, PF, and CF are undefined				
+OF and CF as described above; SF, ZF, AF, PF are undefined				

17.2.2.11 Instruction Set Detail 中的 PUSH -- Push Operand onto the Stack

@@ -?,3 +?,3 @@				
FF	/6	PUSH m32	5	Push memory dword
-50 + /r		PUSH r16	2	Push register word
-50 + /r		PUSH r32	2	Push register dword
+50 + rw		PUSH r16	2	Push register word
+50 + rd		PUSH r32	2	Push register dword

17.2.2.11 Instruction Set Detail 中的 SETcc - Byte Set on Condition

@@ -?,2 +?,2 @@				
0F	94	SETE r/m8	4/5	Set byte if equal (ZF=1)
-0F	9F	SETG r/m8	4/5	Set byte if greater (ZF=0 or SF=OF)
+0F	9F	SETG r/m8	4/5	Set byte if greater (ZF=0 and SF=OF)
@@ -?,3 +?,3 @@				
0F	9C	SETLE r/m8	4/5	Set byte if less (SF!=OF)
-0F	9E	SETLE r/m8	4/5	Set byte if less or equal (ZF=1 and SF!=OF)
-0F	96	SETNA r/m8	4/5	Set byte if not above (CF=1)
+0F	9E	SETLE r/m8	4/5	Set byte if less or equal (ZF=1 or SF!=OF)
+0F	96	SETNA r/m8	4/5	Set byte if not above (CF=1 or ZF=1)
@@ -?,2 +?,2 @@				
0F	9D	SETNL r/m8	4/5	Set byte if not less (SF=OF)
-0F	9F	SETNLE r/m8	4/5	Set byte if not less or equal (ZF=1 and SF!=OF)
+0F	9F	SETNLE r/m8	4/5	Set byte if not less or equal (ZF=0 and SF=OF)

——感谢 16 级许翔、闫雨呼同学的细心发现

17.2.2.11 Instruction Set Detail 中的 SHLD -- Double Precision Shift Left

@@ -?,2 +?,2 @@
 Flags Affected
 -OF, SF, ZF, PF, and CF as described above; AF and OF are undefined
 +OF, SF, ZF, PF, and CF as described above; AF is undefined

17.2.2.11 Instruction Set Detail 中的 SHLR -- Double Precision Shift Right

@@ -?,2 +?,2 @@
 Flags Affected
 -OF, SF, ZF, PF, and CF as described above; AF and OF are undefined
 +OF, SF, ZF, PF, and CF as described above; AF is undefined

Appendix A. Opcode Map 中的 One-Byte Opcode Map

68 的位置应对应于 PUSH Iv

——感谢 16 级张航帆同学提供的信息

Git 入门教程

PA 实验采用 Git 来管理项目代码，和 SVN 类似，Git 是一个强大的版本控制工具。我们将代码托管在 GitHub 仓库，大家通过 `git clone` 命令就可以方便地获取框架代码。同时每次进行 `make` 的时候都会自动进行 `commit`，方便我们追踪你的实验进程。

在平时的实验中，你也可以使用 git 来管理你的代码。Git 带来的好处显而易见，你可以创建分支来进行试验性的内容，在通过测试后再合并到主分支。你也可以回滚对项目进行的改动，防止因为一时手残而带来的终身遗憾。网上有关 git 的资料和教程有很多，本教程只给出一点简单的入门信息，同时强调一点：

提交时只认 master 分支的代码！

相关资料推荐：

- Git document（其中的 GitHub Cheat Sheet 很好用）：<https://git-scm.com/doc>
- git - 简明指南：<http://www.runoob.com/manual/git-guide/>
- `man git`, `git help`, ...

简单 Git 入门

安装 git

```
apt-get install git
```

安装好之后，你需要先进行一些配置工作。在终端里输入以下命令

```
git config --global user.name "Zhang San"      # your name
git config --global user.email "zhangsan@foo.com"  # your email
git config --global core.editor vim              # your favourite editor
git config --global color.ui true
```

经过这些配置，你就可以开始使用 git 了。

在实验中，你会通过 `git clone` 命令下载我们提供的框架代码，里面已经包含一些 git 记录，因此不需要额外进行初始化。如果你想在别的实验/项目中使用 git，你首先需要切换到实验/项目的目录中，然后输入

```
git init
```

进行初始化。

查看存档信息

使用

```
git log
```

查看目前为止所有的存档。

使用

```
git status
```

可以得知与当前存档相比，哪些文件发生了变化。

存档

你可以像以前一样编写代码。等到你的开发取得了一些阶段性成果，你应该马上进行“存档”（当然在 PA 中，每次 make 都会自动帮你存档一次）。

首先你需要使用 `git status` 查看是否有新的文件或已修改的文件未被跟踪，若有，则使用 `git add` 将文件加入跟踪列表，例如

```
git add file.c
```

会将 `file.c` 加入跟踪列表。如果需要一次添加所有未被跟踪的文件，你可以使用

```
git add -A
```

但这样可能会跟踪了一些不必要的文件，例如编译产生的 `.o` 文件，和最后产生的可执行文件。事实上，我们只需要跟踪代码源文件即可。为了让 `git` 在添加跟踪文件之前作筛选，你可以编辑 `.gitignore` 文件(你可以使用 `ls -a` 命令看到它),在里面给出需要被 `git` 忽略的文件和文件类型。

把新文件加入跟踪列表后，使用 `git status` 再次确认。确认无误后就可以存档了，使用

```
git commit
```

提交工程当前的状态。执行这条命令后，将会弹出文本编辑器，你需要在第一行中添加本次存档的注释，例如“fix bug for xxx”。你应该尽可能添加详细的注释，将来你需要根据这些注释来区别不同的存档。编写好注释之后，保存并退出文本编辑器，存档成功。你可以使用 `git log` 查看存档记录，你应该能看到刚才编辑的注释。

读档

如果你在进过了一些修改后发现自己犯下了严重的错误并希望回到过去的某个时间点，那么便可以使用读档功能来回到过去。首先使用 `git log` 来查看已有的存档，并决定你需要回到哪个过去。每一份存档都有一个 `hash code`，例如 `b87c512d10348fd8f1e32ddea8ec95f87215aaa5`，你需要通过 `hash code` 来告诉 `git` 你希望读哪一个档。使用以下命令进行读档：

```
git reset --hard b87c
```

其中 `b87c` 是上文 `hash code` 的前缀: 你不需要输入整个 `hash code`. 这时你再看看你的代码, 你已经成功地回到了过去!

但事实上, 在使用 `git reset` 的 `hard` 模式之前, 你需要再三确认选择的存档是不是你的真正目标. 如果你读入了一个较早的存档, 那么比这个存档新的所有记录都将被删除! 这意为着你不能随便回到"将来"了.

分支

当然还是有办法来避免上文提到的副作用的, 这就是 `git` 的分支功能. 使用命令

```
git branch
```

查看所有分支. 其中 `master` 是主分支, 使用 `git init` 初始化之后会自动建立主分支.

读档的时候使用以下命令

```
git checkout b87c
```

而不是 `git reset`. 这时你将处于一个虚构的分支中, 你可以

- 查看 `b87c` 存档的内容
- 使用以下命令切换到其它分支

```
git checkout 分支名
```

- 对代码的内容进行修改, 但你不能使用 `git commit` 进行存档, 你需要使用

```
git checkout -B 分支名
```

把修改结果保存到一个新的分支中, 如果分支已存在, 其内容将会被覆盖

不同的分支之间不会相互干扰, 这也给项目的分布式开发带来了便利. 有了分支功能, 你就可以在一个分支的多个存档, 或者是多个分支之间来回穿梭了。