Table of Contents

南京大学 计算机科学与技术系 计算机系统基础 课程实验 2017

实验前阅读

最新消息

- 2017/12/04
  - PA4每阶段的提交时间已经更新
- 2017/11/06
  - PA3每阶段的提交时间已经更新
- 2017/10/09
  - PA2每阶段的提交时间已经更新
- 2017/10/01
  - 由于提交网站的校外域名正在进行维护, PA1截止日期延后到2017/10/08 23:59:59

- 实验前请先仔细阅读本页面以及为什么要学习计算机系统基础.
- 如果你在实验过程中遇到了困难, 并打算向我们寻求帮助, 请先阅读提问的智慧这篇文章.
- 如果你发现了实验讲义和材料的错误或者对实验内容有疑问或建议, 请通过邮件的方式联系余子濠(zhuoyu.x#gmail.com)

```
--- nemu/Makefile.git
+++ nemu/Makefile.git
@@ -8,2 +8,2 @@
 default: git
-- -@git add . -A --ignore-errors
+ -@git add .. -A --ignore-errors
```

调试公理

- The machine is always right. (机器永远是对的)
  - Corollary: If the program does not produce the desired output, it is the programmer's fault.
- Every line of untested code is always wrong. (未测试代码永远是错的)
  - Corollary: Mistakes are likely to appear in the "must-be-correct" code.

jyy曾经将它们作为fact提出. 事实上无数程序员(包括你的学长学姐)在实践当中一次又一次验证了它们的正确性, 因此它们在这里作为公理出现. 你可以不相信调试公理, 但你可能会在调试的时候遇到麻烦.

成长是一个痛苦的过程

PA是充满挑战性的, 在实验过程中, 你会看到自己软弱的一面: 没到deadline就不想动手的拖延症, 打算最后抱大腿的侥幸, 面对英文资料的恐惧, 对不熟悉工具的抵触, 遇到问题就请教大神的懒惰, 多次失败而想放弃的念头, 对过去一年自己得过且过的悔恨, 对完成实验的

绝望, 对将来的迷茫... 承认自己的软弱, 是成长的第一步; 对这样的自己的不甘, 是改变自己的动力. 做PA不仅仅是做实验, 更重要的是认识并改变那个软弱的自己. 即使不能完成所有的实验内容, 只要你坚持下来, 你就是非常了不起的! 你会看到成长的轨迹, 看到你正在告别过去的自己.

小百合系版"有像我一样不会写代码的cser么?"回复节选

- 我们都是活生生的人, 从小就被不由自主地教导用最小的付出获得最大的得到, 经常会忘记我们究竟要的是什么. 我承认我完美主义, 但我想每个人心中都有那一份求知的渴望和对真理的向往, "大学"的灵魂也就在于超越世俗, 超越时代的纯真和理想 -- 我们不是要讨好企业的毕业生, 而是要寻找改变世界的力量. -- jyy
- 教育除了知识的记忆之外, 更本质的是能力的训练, 即所谓的training. 而但凡training就必须克服一定的难度, 否则你就是在做重复劳动, 能力也不会有改变. 如果遇到难度就选择退缩, 或者让别人来替你克服本该由你自己克服的难度, 等于是自动放弃了获得training的机会, 而这其实是大学专业教育最宝贵的部分. -- etone
- 这种"只要不影响我现在survive, 就不要紧"的想法其实非常的利己和短视: 你在专业上的技不如人, 迟早有一天会找上来, 会影响到你个人职业生涯的长远的发展; 更严重的是, 这些以得过且过的态度来对待自己专业的学生, 他们的survive其实是以透支南大教育的信誉为代价的 -- 如果我们一定比例的毕业生都是这种情况, 那么过不了多久, 不但那些混到毕业的学生也没那么容易survive了, 而且那些真正自己刻苦努力的学生, 他们的前途也会受到影响. -- etone

实验方案

理解"程序如何在计算机上运行"的根本途径是从"零"开始实现一个完整的计算机系统. 南京大学计算机科学与技术系 `计算机系统基础` 课程的小型项目 (Programming Assignment, PA)将提出x86架构的一个教学版子集n86, 指导学生实现一个功能完备的n86模拟器NEMU(NJU EMUlator), 最终在NEMU上运行游戏"仙剑奇侠传", 来让学生探究"程序在计算机上运行"的基本原理. NEMU受到了QEMU的启发, 并去除了大量与课程内容差异较大的部分. PA包括一个准备实验(配置实验环境)以及5部分连贯的实验内容:

- 简易调试器
- 冯诺依曼计算机系统
- 异常控制流
- 分时多任务
- 程序性能优化

实验环境

- CPU架构: IA-32
- 操作系统: GNU/Linux
- 编译器: GCC
- 编程语言: C语言

如何获得帮助

在学习和实验的过程中, 你会遇到大量的问题. 除了参考课本内容之外, 你需要掌握如何获取其它参考资料.

但在此之前, 你需要适应查阅英文资料. 和以往程序设计课上遇到的问题不同, 你会发现你不太容易搜索到相关的中文资料. 回顾计算机科学层次抽象图, 计算机系统基础处于程序设计的下层. 这意味着, 懂系统基础的人不如懂程序设计的人多, 相应地, 系统基础的中文资料也会比程序设计的中文资料少.

如何适应查阅英文资料? 方法是尝试并坚持查阅英文资料.

搜索引擎, 百科和问答网站

为了查找英文资料, 你应该使用下表中推荐的网站:

	搜索引擎	百科	问答网站
推荐使用	这里和这里有google搜索镜像	http://en.wikipedia.org	http://stackoverflow.com
不推荐使用	~~http://www.baidu.com~~	~~http://baike.baidu.com~~	~~http://zhidao.baidu.com~~ ~~http://bbs.csdn.net~~

一些说明:

- 一般来说, 百度对英文关键词的处理能力比不上Google.
- 通常来说, 英文维基百科比中文维基百科和百度百科包含更丰富的内容. 为了说明为什么要使用英文维基百科, 请你对比词条 `前束范式` 分别在百度百科, 中文维基百科和英文维基百科中的内容.
- stackoverflow是一个程序设计领域的问答网站, 里面除了技术性的问题(What is ":-!!" in C code?)之外, 也有一些学术性(Is there a regular expression to detect a valid regular expression?)和一些有趣的问题(What is the “-->” operator in C++?).

官方手册

官方手册包含了查找对象的所有信息, 关于查找对象的一切问题都可以在官方手册中找到答案. 通常官方手册的内容十分详细, 在短时间内通读一遍基本上不太可能, 因此你需要懂得"如何使用目录来定位你所关心的问题". 如果你希望寻找一些用于快速入门的例子, 你应该使用搜索引擎.

这里列出一些本课程中可能会用到的手册:

- Intel 80386 Programmer's Reference Manual (人手一本的i386手册)
- GCC 6.3.0 Manual
- GDB User Manual
- GNU Make Manual
- System V ABI for i386
- On-line Manual Pager (即man, 这里有一个入门教程)

GNU/Linux入门教程

jyy为我们准备了一个GNU/Linux入门教程, 如果你是第一次使用GNU/Linux, 请阅读这里.

PA0 - 世界诞生的前夜: 开发环境配置

世界诞生的故事 - 序章

PA讲述的是一个"先驱创造计算机"的故事.

先驱打算创造一个计算机世界. 但巧妇难为无米之炊, 为了更方便地创造这个世界, 就算是先驱也是花了一番功夫来准备的. 让我们来看看他们都准备了些什么工具.

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 10小时

截止时间: 2017/09/10 23:59:59

提交说明: 见这里

对, 你没有看错, 除了一些重要的信息之外, PA0的实验讲义都是英文!

随着科学技术的发展, 在国际学术交流中使用英语已经成为常态: 顶尖的论文无一不使用英文来书写, 在国际上公认的计算机领域经典书籍也是使用英文编著. 顶尖的论文没有中文翻译版; 如果需要获取信息, 也应该主动去阅读英文材料, 而不是等翻译版出版. "我是中国人, 我只看中文"这类观点已经不符合时代发展的潮流, 要站在时代的最前沿, 阅读英文材料的能力是不可或缺的.

阅读英文材料, 无非就是"不会的单词查字典, 不懂的句子反复读". 如今网上有各种词霸可解燃眉之急, 但英文阅读能力的提高贵在坚持. "刚开始觉得阅读英文效率低", 是所有中国人都无法避免的经历. 如果你发现身边的大神可以很轻松地阅读英文材料, 那是因为他们早就克服了这些困难. 引用陈道蓄老师的话: 坚持一年, 你就会发现有不同; 坚持两年, 你就会发现大有不同.

撇开这些高大上的话题不说, 阅读英文材料和你有什么关系呢? 有! 因为在PA中陪伴你的, 就是没有中文版的i386手册, 当然还有`man`: 如果你不愿意阅读英文材料, 你是注定无法独立完成PA的.

作为过渡, 我们为大家准备了全英文的PA0. PA0的目的是配置实验环境, 同时熟悉GNU/Linux下的工作方式. 其中涉及的都是一些操作性的步骤, 你不必为了完成PA0而思考深奥的问题.

你需要独立完成PA0, 请你认真阅读讲义中的每一个字符, 并按照讲义中的内容进行操作: 当讲义提到要在互联网上搜索某个内容时, 你就去互联网上搜索这个内容. 如果遇到了错误, 请认真反复阅读讲义内容, 机器永远是对的. 如果你是第一次使用GNU/Linux, 你还需要查阅大量资料或教程来学习一些新工具的使用方法, 这需要花费大量的时间(例如你可能需要花

费一个下午的时间, 仅仅是为了使用`vim`在文件中键入两行内容). 这就像阅读英文材料一样, 一开始你会觉得效率很低, 但随着时间的推移, 你对这些工具的使用会越来越熟练. 相反, 如果你通过"投机取巧"的方式来完成PA0, 你将会马上在PA1中遇到麻烦. 正如etone所说, 你在专业上的技不如人, 迟早有一天会找上来.

另外, PA0的讲义只负责给出操作过程, 并不负责解释这些操作相关的细节和原理. 如果你希望了解它们, 请在互联网上搜索相关内容.

PA0 is a guide to GNU/Linux development environment configuration. You are guided to install a GNU/Linux development environment. All PAs and Labs are done in this environment. If you are new to GNU/Linux, and you encounter some troubles during the configuration, which are not mentioned in this lecture note (such as "No such file or directory"), that is your fault. Go back to read this lecture note carefully. Remember, the machine is always right!

Installing Docker

Docker is an implementation of the lightweight virtualization technology. Virtual machines built by this technology is called "container". By using Docker, it is very easy to deploy GNU/Linux applications.

If you already have one copy of GNU/Linux distribution different from that we recommend, and you want to use your copy as the development environment, we still encourage you to install docker on your GNU/Linux distribution to use the same GNU/Linux distribution we recommend over docker to avoid issues brought by platform disparity. Refer to Docker online Document for more information about installing Docker for GNU/Linux. It is OK if you still insist on your GNU/Linux distribution. But if you encounter some troubles because of platform disparity, please search the Internet for trouble-shooting.

It it also OK to use traditional virtual machines, such as VMWare or VirtualBox, instead of Docker. If you decide to do this and you do not have a copy of GNU/Linux, please install Debian 9 distribution in the virtual machine. Also, please search the Internet for trouble-shooting if you have any problems about virtual machines.

Download Docker from this website according to your host operating system, then install Docker with default settings. Reboot the system if necessary. If your operating system can not meet the requirement of installing Docker, please upgrade your operating system. Do not install `Docker Toolbox` instead. It seems not very stable in Windows since it is based on VirtualBox.

Preparing Dockerfile

`Dockerfile` is the configuration file used to build a Docker image. Now we are going to prepare a Dockerfile with proper contentby using the terminal working environment.

- If your host is GNU/Linux or Mac, you can use the default terminal in the system.
- If your host is Windows, open `PowerShell`.

Type the following commands after the prompt, one command per line. Every command is issued by pressing the `Enter` key. The contents after a `#` is the comment about the command, and you do not need to type the comment.

```
mkdir mydocker     # create a directory with name "mydocker"
cd mydocker        # enter this directory
```

Now use the text editor in the host to new a file called `Dockerfile`.

- Windows: Type command `notepad Dockerfile` to open Notepad.
- MacOS: Type command `open -e Dockerfile` to open TextEdit.
- GNU/Linux: Use your favourite editor to open Dockerfile.

Now copy the following contents into Dockerfile:

```
 # setting base image
FROM debian

 # new a directory for sshd to run
RUN mkdir -p /var/run/sshd

 # installing ssh server
RUN apt-get update
RUN apt-get install -y openssh-server

 # installing sudo
RUN apt-get install -y sudo

 # make ssh services use IPv4 to let X11 forwarding work correctly
RUN echo AddressFamily inet >> /etc/ssh/sshd_config

 # defining user account imformation
ARG username=ics
ARG userpasswd=ics

 # adding user
RUN useradd -ms /bin/bash $username && (echo $username:$userpasswd | chpasswd)

 # adding user to sudo group
RUN adduser $username sudo

 # setting running application
CMD /usr/sbin/sshd -D
```

We choose the Debian distribution as the base image, since it can be quite small. Change `username` and `userpasswd` above to your favourite account settings. Save the file and exit the editor.

For Windows user, `notepad` will append suffix `.txt` to the saved file. This is unexpected. Use the following command to rename the file.

```
mv Dockerfile.txt Dockerfile     # rename the file to remove the suffix in Windows
```

Building Docker image

Keep the Internet conntected. Type the following command to build our image:

```
docker build -t ics-image .
```

This command will build an image with a tag `ics-image`, using the Dockerfile in the current directory (mydocker). In particular, if your host is GNU/Linux, all Docker commands should be executed with root privilege, or alternatively you can add your account to the group `docker` before executing any docker commands. If it is the first time you run this command, Docker will pull the base image `debian` from Docker Hub. This will cost several minutes to finish.

After the command above finished, type the following command to show Docker images:

```
docker images
```

This command will show information about all Docker images.

```
REPOSITORY          TAG         IMAGE ID          CREATED             SIZE
ics-image           latest      7d9495d03763      4 minutes ago       210 MB
debian              latest      fb434121fc77      4 hours ago         100 MB
```

If you see a repository with name `ics-image`, you are done with building image.

Now we can remove the directory mentioned above.

```
cd ..           # go back to the parent directory
rm -r mydocker  # remove the `mydocker` directory
```

Creating Debian container

After building the image, now we can create a container. Type the following command:

```
docker create --name=ics-vm -p 20022:22 ics-image
```

This command will create a container with the following property:

- the name of the container is `ics-vm`
- the Docker image is `ics-image`, which we just built
- the default SSH port (`22`) in the container is bound to port `20022` in the docker host

If the above command fails because a container with the same name already exists, type the following command to remove the existing container:

```
docker rm ics-vm
```

Then create the container again.

To see whether the container is created successfully, type the following command to show containers:

```
docker ps -a
```

This command will show information about all Docker containers. If you see a container with name `ics-vm`, you are done with creating container.

First Exploration with GNU/Linux

To start the container, type the following command in the terminal:

```
docker start ics-vm
```

This command will start the container with name `ics-vm`, which is created by us. By defualt, `ics-vm` will start in detach mode, running the SSH deamon instructed at the end of the Dockerfile. This means we can not interact with it directly. To login the container, we should do the SSH configuration first.

SSH configuration

According to the type of your host operating system, you will perform different configuration.

For GNU/Linux and Mac users

You will use the build-in `ssh` tool, and do not need to install an extra one. Open a terminal, run

```
ssh -p 20022 username@127.0.0.1
```

where `username` is the user name in Dockerfile. By default, it is `ics`. If you are prompted with

```
Are you sure you want to continue connecting (yes/no)?
```

enter "yes". Then enter the user password in Dockerfile. If everything is fine, you will login the container via SSH successfully.

For Windows users

Windows has no build-in `ssh` tool, and you have to download one manually. Download the latest release version of `putty.exe` here. Run `putty.exe`, and you will see a dialog is invoked. In the input box labeled with `Host Name (or IP address)`, enter `127.0.0.1`, and change the port to `20022`. To avoid entering IP address and port every time you login, you

can save these information as a session. Leave other settings default, then click `Open` button. Enter the container user name and password in Dockerfile. If everything is fine, you will login the container via SSH successfully.

First exploration

After login via SSH, you will see the following prompt:

```
username@hostname:~$
```

This prompt shows your username, host name, and the current working directory. The username should be the same as you set in the Dockerfile before building the image. The host name is generated randomly by Docker, and it is unimportant for us. The current working directory is `~` now. As you switching to another directory, the prompt will change as well. You are going to finish all the experiments under this environment, so try to make friends with terminal!

Where is GUI?

Many of you always use operating system with GUI, such as Windows. The container you just created is without GUI. It is completely with CLI (Command Line Interface). As you entering the container, you may feel empty, depress, and then panic...

Calm down yourself. Have you wondered if there is something that you can do it in CLI, but can not in GUI? Have no idea? If you are asked to count how many lines of code you have coded during the 程序设计基础 course, what will you do?

If you stick to Visual Studio, you will never understand why `vim` is called 编辑器之神. If you stick to Windows, you will never know what is Unix Philosophy. If you stick to GUI, you can only do what it can; but in CLI, it can do what you want. One of the most important spirits of young people like you is to try new things to bade farewell to the past.

GUI wins when you do something requires high definition displaying, such as watching movies. But in our experiments, GUI is unnecessary. Here are two articles discussing the comparision between GUI and CLI:

- Why Use a Command Line Instead of Windows?
- Command Line vs. GUI

Now you can see how much disk space Debian occupies. Type the following command:

```
df -h
```

You can see that Debian is quite "slim".

Why Windows is quite "fat"?

Installing a Windows operating system usually requires much more disk space as well as memory. Can you figure out why the Debian operating system can be so "slim"?

To shut down the container, first type `exit` command to terminate the SSH connection. Then go back to the host terminal, stop the container by:

```
docker stop ics-vm
```

And type `exit` to exit the host terminal.

Installing Tools

In GNU/Linux, you can download and install a software by one command (which may be difficult to do in Windows). This is achieved by the package manager. Different GNU/Linux distribution has different package manager. In Debian, the package manager is called `apt`.

You will download and install some tools needed for the PAs from the network mirrors. Before using the network mirrors, you should check whether the container can access the Internet.

Checking network state

By the default network setting of the container will share the same network state with your host. That is, if your host is able to access the Internet, so does the container. To test whether the container is able to access the Internet, you can try to ping a host outside the university LAN:

```
ping www.baidu.com -c 4
```

You should receive reply packets successfully:

```
PING www.a.shifen.com (220.181.111.188) 56(84) bytes of data.
64 bytes from 220.181.111.188: icmp_seq=1 ttl=51 time=5.81 ms
64 bytes from 220.181.111.188: icmp_seq=2 ttl=51 time=6.11 ms
64 bytes from 220.181.111.188: icmp_seq=3 ttl=51 time=6.88 ms
64 bytes from 220.181.111.188: icmp_seq=4 ttl=51 time=4.92 ms

--- www.a.shifen.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 4.925/5.932/6.882/0.706 ms
```

If you get an "unreachable" message, please check whether you can access www.baidu.com in the host system.

Updating APT package information

Now you can tell `apt` to retrieve software information from the sources:

```
apt-get update
```

However, you will receive an error message:

```
E: Could not open lock file /var/lib/apt/lists/lock - open (13: Permission denied)
E: Unable to lock directory /var/lib/apt/lists/
```

This is because `apt-get` requires superuser privilege to run.

> **Why some operations require superuser privilege?**
>
> In a real GNU/Linux, shutting down the system also requires superuser privilege. Can you provide a scene where bad thing will happen if the shutdown operation does not require superuser privilege?

To run `apt-get` with superuser privilege, use `sudo`. If you find an operation requires superuser permission, append `sodu` before that operation. For example,

```
sudo apt-get update
```

Enter your password you set previously in the `Dockerfile`. Now `apt-get` should run successfully. Since it requires Internet accessing, it may cost some time to finish.

Installing tools for PAs

The following tools are necessary for PAs:

```
apt-get install build-essential    # build-essential packages, include binary utilitie
s, gcc, make, and so on
apt-get install gdb                # GNU debugger
apt-get install git                # reversion control system
apt-get install libreadline-dev    # a library to use compile the project later
apt-get install libsdl2-dev        # a library to use compile the project later
apt-get install qemu-system-x86    # QEMU
```

The usage of these tools is explained later.

Configuring vim

```
apt-get install vim
```

`vim` is called 编辑器之神. You will use `vim` for coding in all PAs and Labs, as well as editing other files. Maybe some of you prefer to other editors requiring GUI environment (such Visual Studio). However, you can not use them in some situations, especially when you are accessing a physically remote server:

- the remote server does not have GUI installed, or
- the network condition is so bad that you can not use any GUI tools.

Under these situations, `vim` is still a good choice. If you prefer to `emacs`, you can download and install `emacs` from network mirrors.

Learning vim

You are going to be asked to modify a file using `vim`. For most of you, this is the first time to use `vim`. The operations in `vim` are quite different from other editors you have ever used. To learn `vim`, you need a tutorial. There are two ways to get tutorials:

- Issue the `vimtutor` command in terminal. This will launch a tutorial for `vim`.This way is recommended, since you can read the tutorial and practice at the same time.
- Search the Internet with keyword "vim 教程", and you will find a lot of tutorials about `vim`. Choose some of them to read, meanwhile you can practice with the a temporary file by

  ```
  vim test
  ```

PRACTICE IS VERY IMPORTANT. You can not learn anything by only reading the tutorials.

Some games operated with vim

Here are some games to help you master some basic operations in `vim`. Have fun!

- Vim Adventures
- Vim Snake
- Open Vim Tutorials
- Vim Genius

The power of vim

You may never consider what can be done in such a "BAD" editor. Let's see two examples.

The first example is to generate the following file:

```
1
2
3
.....
98
99
100
```

This file contains 100 lines, and each line contains a number. What will you do? In `vim`, this is a piece of cake. First change `vim` into normal state (when `vim` is just opened, it is in normal state), then press the following keys sequentially:

```
i1<ESC>q1yyp<C-a>q98@1
```

where `<ESC>` means the ESC key, and `<C-a>` means "Ctrl + a" here. You only press no more than 15 keys to generate this file. Is it amazing? What about a file with 1000 lines? What you do is just to press one more key:

```
i1<ESC>q1yyp<C-a>q998@1
```

The magic behind this example is recording and replaying. You initial the file with the first line. Then record the generation of the second. After that, you replay the generation for 998 times to obtain the file.

The second example is to modify a file. Suppose you have such a file:

```
aaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeffffffffffffffffffffffff
ggggggggggggggggggggggggghhhhhhhhhhhhhhhhhhhhhhh
iiiiiiiiiiiiiiiiiiiiiiijjjjjjjjjjjjjjjjjjjjjjj
```

You want to modify it into:

```
bbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaa
ddddddddddddddddddddddddddcccccccccccccccccccccc
ffffffffffffffffffffffffeeeeeeeeeeeeeeeeeeeeeee
hhhhhhhhhhhhhhhhhhhhhhhhggggggggggggggggggggggg
jjjjjjjjjjjjjjjjjjjjjjjjiiiiiiiiiiiiiiiiiiiiiiii
```

What will you do? In `vim`, this is a piece of cake, too. First locate the cursor to first "a" in the first line. And change `vim` into normal state, then press the following keys sequentially:

```
<C-v>24l4jd$p
```

where `<C-v>` means "Ctrl + v" here. What about a file with 100 such lines? What you do is just to press one more key:

```
<C-v>24l99jd$p
```

Although these two examples are artificial, they display the powerful functionality of `vim`, comparing with other editors you have used.

Enabling syntax highlight

`vim` provides more improvements comparing with `vi`. But these improvements are disabled by default. Therefore, you should enable them first.

We take syntax highlight as an example to illustrate how to enable the features of `vim`. To do this, you should modify the `vim` configuration file. The file is called `vimrc`, and it is located under `/etc/vim` directory. We first make a copy of it to the home directory by `cp` command:

```
cp /etc/vim/vimrc ~/.vimrc
```

And switch to the home directory if you are not under it yet:

```
cd ~
```

If you use `ls` to list files, you will not see the `.vimrc` you just copied. This is because a file whose name starts with a `.` is a hidden file in GNU/Linux. To show hidden files, use `ls` with `-a` option:

```
ls -a
```

Then open `.vimrc` using `vim`:

```
vim .vimrc
```

After you learn some basic operations in `vim` (such as moving, inserting text, deleting text), you can try to modify the `.vimrc` file as following:

```
--- before modification
+++ after modification
@@ -17,3 +17,3 @@
 " Vim5 and later versions support syntax highlighting. Uncommenting the next
 " line enables syntax highlighting by default.
-"syntax on
+syntax on
```

We present the modification with GNU diff format. Lines starting with `+` are to be inserted. Lines starting with `-` are to be deleted. Other lines keep unchanged. If you do not understand the diff format, please search the Internet for more information.

After you are done, you should save your modification. Exit `vim` and open the `vimrc` file again, you should see the syntax highlight feature is enabled.

Enabling more vim features

Modify the `.vimrc` file mentioned above as the following:

```
--- before modification
+++ after modification
@@ -21,3 +21,3 @@
 " If using a dark background within the editing area and syntax highlighting
 " turn on this option as well
-"set background=dark
+set background=dark
@@ -31,5 +31,5 @@
 " Uncomment the following to have Vim load indentation rules and plugins
 " according to the detected filetype.
-"if has("autocmd")
-"  filetype plugin indent on
-"endif
+if has("autocmd")
+  filetype plugin indent on
+endif
@@ -37,10 +37,10 @@
 " The following are commented out as they cause vim to behave a lot
 " differently from regular Vi. They are highly recommended though.
 "set showcmd            " Show (partial) command in status line.
-"set showmatch          " Show matching brackets.
-"set ignorecase         " Do case insensitive matching
-"set smartcase          " Do smart case matching
-"set incsearch          " Incremental search
+set showmatch          " Show matching brackets.
+set ignorecase         " Do case insensitive matching
+set smartcase          " Do smart case matching
+set incsearch          " Incremental search
 "set autowrite          " Automatically save before commands like :next and :make
-"set hidden             " Hide buffers when they are abandoned
+set hidden             " Hide buffers when they are abandoned
 "set mouse=a            " Enable mouse usage (all modes)
```

You can append the following content at the end of the `.vimrc` file to enable more features. Note that contents after a double quotation mark `"` are comments, and you do not need to include them. Of course, you can inspect every features to determine to enable or not.

```
setlocal noswapfile " 不要生成swap文件
set bufhidden=hide " 当buffer被丢弃的时候隐藏它
colorscheme evening " 设定配色方案
set number " 显示行号
set cursorline " 突出显示当前行
set ruler " 打开状态栏标尺
set shiftwidth=4 " 设定 << 和 >> 命令移动时的宽度为 4
set softtabstop=4 " 使得按退格键时可以一次删掉 4 个空格
set tabstop=4 " 设定 tab 长度为 4
set nobackup " 覆盖文件时不备份
set autochdir " 自动切换当前目录为当前文件所在的目录
set backupcopy=yes " 设置备份时的行为为覆盖
set hlsearch " 搜索时高亮显示被找到的文本
set noerrorbells " 关闭错误信息响铃
set novisualbell " 关闭使用可视响铃代替呼叫
set t_vb= " 置空错误铃声的终端代码
set matchtime=2 " 短暂跳转到匹配括号的时间
set magic " 设置魔术
set smartindent " 开启新行时使用智能自动缩进
set backspace=indent,eol,start " 不设定在插入状态无法用退格键和 Delete 键删除回车符
set cmdheight=1 " 设定命令行的行数为 1
set laststatus=2 " 显示状态栏 (默认值为 1, 无法显示状态栏)
set statusline=\ %<%F[%1*%M%*%n%R%H]%=\ %y\ %0(%{&fileformat}\ %{&encoding}\ Ln\ %l,\
Col\ %c/%L%) " 设置在状态行显示的信息
set foldenable " 开始折叠
set foldmethod=syntax " 设置语法折叠
set foldcolumn=0 " 设置折叠区域的宽度
setlocal foldlevel=1 " 设置折叠层数为 1
nnoremap <space> @=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')<CR> " 用空格键来开关折叠
```

If you want to refer different or more settings for `vim`, please search the Internet. In addition, there are many plug-ins for `vim` (one of them you may prefer is `ctags`, which provides the ability to jump among symbol definitions in the code). They make `vim` more powerful. Also, please search the Internet for more information about `vim` plug-ins.

More Exploration

Learning to use basic tools

After installing tools for PAs, it is time to explore GNU/Linux again! Here is a small tutorial for GNU/Linux written by jyy. If you are new to GNU/Linux, read the tutorial carefully, and most important, try every command mentioned in the tutorial. Remember, you can not learn anything by only reading the tutorial. Besides, 鸟哥的Linux私房菜 is a book suitable for freshman in GNU/Linux.

Write a "Hello World" program under GNU/Linux

Write a "Hello World" program, compile it, then run it under GNU/Linux. If you do not know what to do, refer to the GNU/Linux tutorial above.

Write a Makefile to compile the "Hello World" program

Write a Makefile to compile the "Hello World" program above. If you do not know what to do, refer to the GNU/Linux tutorial above.

Now, stop here. Here is a small tutorial for GDB. GDB is the most common used debugger under GNU/Linux. If you have not used a debugger yet (even in Visual Studio), blame the 程序设计基础 course first, then blame yourself, and finally, read the tutorial to learn to use GDB.

Learn to use GDB

Read the GDB tutorial above and use GDB following the tutorial. In PA1, you will be required to implement a simplified version of GDB. If you have not used GDB, you may have no idea to finish PA1.

RTFM

The most important command in GNU/Linux is `man` - the on-line manual pager. This is because `man` can tell you how to use other commands. Here is a small tutorial for `man`. Remember, learn to use `man`, learn to use everything. Therefore, if you want to know something about GNU/Linux (such as shell commands, system calls, library functions, device files, configuration files...), RTFM.

Installing tmux

`tmux` is a terminal multiplexer. With it, you can create multiple terminals in a single screen. It is very convenient when you are working with a high resolution monitor. To install `tmux`, just issue the following command:

```
apt-get install tmux
```

Now you can run `tmux`, but let's do some configuration first. Go back to the home directory:

```
cd ~
```

New a file called `.tmux.conf`:

```
vim .tmux.conf
```

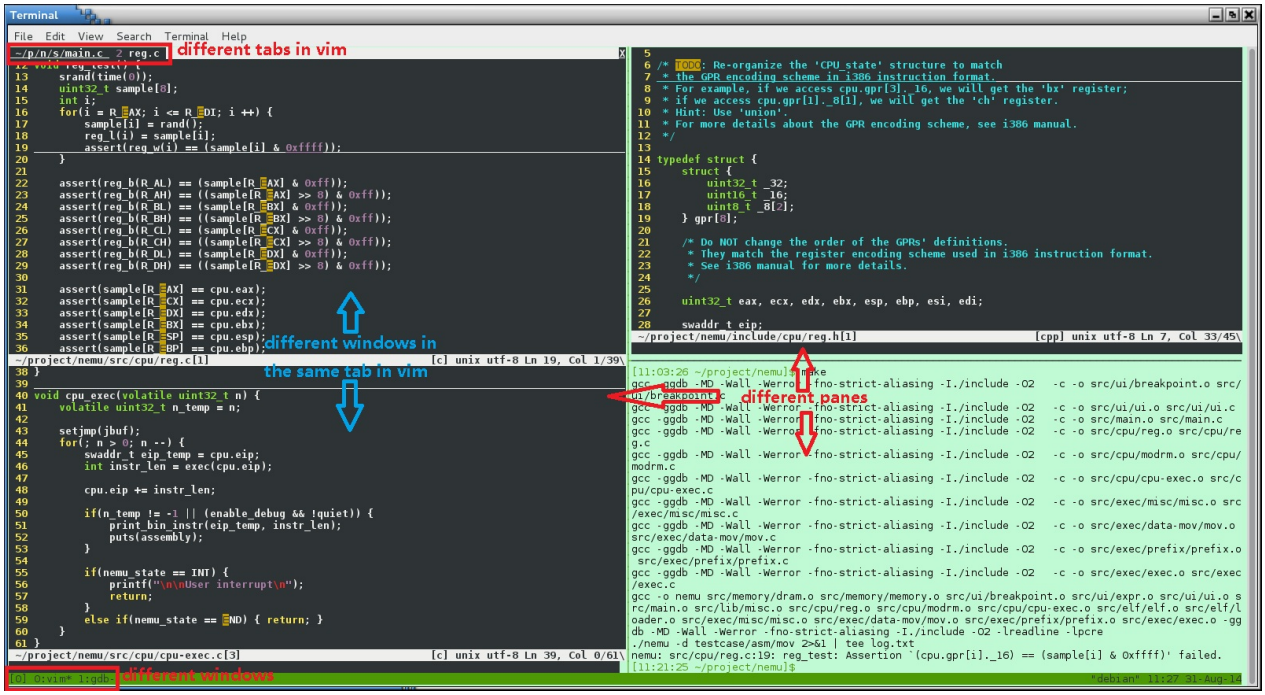
Append the following content to the file:

```
setw -g c0-change-trigger 100
setw -g c0-change-interval 250

bind-key c new-window -c "#{pane_current_path}"
bind-key % split-window -h -c "#{pane_current_path}"
bind-key '"' split-window -c "#{pane_current_path}"
```

The first two lines of settings control the output rate of `tmux`. Without them, `tmux` may become unresponsive when lots of contents are output to the screen. The last three lines of settings make `tmux` "remember" the current working directory of the current pane while creating new window/pane.

Maximize the terminal windows size, then use `tmux` to create multiple normal-size terminals within single screen. For example, you may edit different files in different directories simultaneously. You can edit them in different terminals, compile them or execute other commands in another terminal, without opening and closing source files back and forth. You can scroll the content in a `tmux` terminal up and down. For how to use `tmux`, please search the Internet. The following picture shows a scene working with multiple terminals within single screen. Is it COOL?



Things behind scrolling

You should have used scroll bars in GUI. You may take this for granted. So you may consider the original un-scrollable terminal (the one you use when you just log in) the hell. But think of these: why the original terminal can not be scrolled? How does `tmux` make the terminals scrollable? And last, do you know how to implement a scroll bar?

GUI is not something mysterious. Remember, behind every elements in GUI, there is a story about it. Learn the story, and you will learn a lot. You may say "I just use GUI, and it is unnecessary to learn the story." Yes, you are right. The appearance of GUI is to hide the story for users. But almost everyone uses GUI in the world, and that is why you can not tell the difference between you and them.

Transferring Files Between host and container

With the SSH port, we can easily copy files between host and container.

For GNU/Linux and Mac users

You will use the build-in scp tool, and do not need to install an extra one. To copy file from container to host, issue the following command in the host terminal:

```
scp -P 20022 username@127.0.0.1:SRC_PATH HOST_PATH
```

where

- `username` is the user name in Dockerfile. By default, it is `ics`.
- `SRC_PATH` is the path of the file in container to copy
- `HOST_PATH` is the path of the host to copy to

For example, the following command will copy a file in the container to a host path:

```
scp -P 20022 ics@127.0.0.1:/home/ics/a.txt .
```

To copy file from host to container, issue the following command in the host terminal:

```
scp -P 20022 HOST_SRC_PATH username@127.0.0.1:DEST_PATH
```

where

- `HOST_SRC_PATH` is the path of the host file to copy
- `username` is the user name in Dockerfile. By default, it is `ics`.
- `DEST_PATH` is the path in the container to copy to

For example, the following command will copy a folder in Windows into the container:

```
scp -P 20022 hello.c ics@127.0.0.1:/home/ics
```

For Windows users

Windows has no build-in `scp` tool, and you have to download one manually. Download the latest release version of `pscp.exe` here. Change the current directory of PowerShell to the one with `pscp.exe` in it. Then use the following commands to transfer files.

```
./pscp -P 20022 username@127.0.0.1:SRC_PATH HOST_PATH
./pscp -P 20022 HOST_SRC_PATH username@127.0.0.1:DEST_PATH
```

The explanation of these commands is similar to `scp` above. Refer to them for more information.

Have a try!

1. New a text file with casual contents in the host.
2. Copy the text file to the container.
3. Modify the content of the text file in the container.
4. Copy the modified file back to the host.

Check whether the content of the modified file you get after the last step is expected. If it is the case, you are done!

Acquiring Source Code for PAs

Getting Source Code

Go back to the home directory by

```
cd ~
```

Usually, all works unrelated to system should be performed under the home directory. Other directories under the root of file system (`/`) are related to system. Therefore, do NOT finish your PAs and Labs under these directories by `sudo` .

不要使用root账户做实验!!!

从现在开始, 所有与系统相关的配置工作已经全部完成, 你已经没有使用root账户的必要. 继续使用root账户进行实验, 会改变实验相关文件的权限属性, 可能会导致开发跟踪系统无法正常工作; 更严重的, 你的误操作可能会无意中损坏系统文件, 导致虚拟机/容器无法启动! 往届有若干学长因此而影响了实验进度, 甚至由于损坏了实验相关的文件而影响了分数. 请大家引以为鉴, 不要贪图方便, 否则后果自负!

如果你仍然不理解为什么要这样做, 你可以阅读这个页面: Why is it bad to login as root? 正确的做法是: 永远使用你的普通账号做那些安分守己的事情(例如写代码), 当你需要进行一些需要root权限才能进行的操作时, 使用 `sudo` .

Now acquire source code for PA by the following command:

```
git clone -b 2017 https://github.com/NJU-ProjectN/ics-pa.git ics2017
```

A directory called `ics2017` will be created. This is the project directory for PAs. Details will be explained in PA1.

Issue the following commands to perform `git` configuration:

```
git config --global user.name "161220000-Zhang San" # your student ID and name
git config --global user.email "zhangsan@foo.com"   # your email
git config --global core.editor vim                 # your favorite editor
git config --global color.ui true
```

You should configure `git` with your student ID, name, and email. Before continuing, please read this `git` tutorial to learn some basics of `git` .

Enter the project directory `ics2017`, then run

```
git branch -m master
bash init.sh
```

to initialize all the subprojects. This script will pull 4 subprojects from github. We will explain them later. Besides, the script will also add some environment variables into the bash configuration file `~/.bashrc`. These variables are defined by absolute path to support the compilation of the subprojects. Therefore, DO NOT move your project to another directory once the initialization finishes, else these variables will become invalid. Particularly, if you use shell other than `bash`, please set these variables in the corresponding configuration file manually.

Git usage

We will use the `branch` feature of `git` to manage the process of development. A branch is an ordered list of commits, where a commit refers to some modifications in the project.

You can list all branches by

```
git branch
```

You will see there is only one branch called "master" now.

```
* master
```

To create a new branch, use `git checkout` command:

```
git checkout -b pa0
```

This command will create a branch called `pa0`, and check out to it. Now list all branches again, and you will see we are now at branch `pa0`:

```
  master
* pa0
```

From now on, all modifications of files in the project will be recorded in the branch `pa0`.

Now have a try! Modify the `STU_ID` variable in `nemu/Makefile.git`:

```
STU_ID=161220000            # your student ID
```

Run

```
git status
```

to see those files modified from the last commit:

```
On branch pa0
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   nemu/Makefile.git

no changes added to commit (use "git add" and/or "git commit -a")
```

Run

```
git diff
```

to list modifications from the last commit:

```
diff --git a/nemu/Makefile.git b/nemu/Makefile.git
index c9b1708..b7b2e02 100644
--- a/nemu/Makefile.git
+++ b/nemu/Makefile.git
@@ -1,4 +1,4 @@
-STU_ID = 161220000
+STU_ID = 161221234

  # DO NOT modify the following code!!!
```

You should see the `STU_ID` is modified. Now add the changes to commit by `git add`, and issue `git commit`:

```
git add .
git commit
```

The `git commit` command will call the text editor. Type `modified my STU_ID` in the first line, and keep the remaining contents unchanged. Save and exit the editor, and this finishes a commit. Now you should see a log labeled with your student ID and name by

```
git log
```

Now switch back to the `master` branch by

```
git checkout master
```

Open `nemu/Makefile.git`, and you will find that `STU_ID` is still unchanged! By issuing `git log`, you will find that the commit log you just created has disappeared!

Don't worry! This is a feature of branches in `git`. Modifications in different branches are isolated, which means modifying files in one branch will not affect other branches. Switch back to `pa0` branch by

```
git checkout pa0
```

You will find that everything comes back! At the beginning of PA1, you will merge all changes in branch `pa0` into `master`.

The workflow above shows how you will use branch in PAs:

- before starting a new PA, new a branch `pa?` and check out to it
- coding in the branch `pa?` (this will introduce lot of modifications)
- after finish the PA, merge the branch `pa?` into `master`, and check out back to `master`

Compiling and Running NEMU

Now enter `nemu/` directory, and compile the project by `make`:

```
make
```

If nothing goes wrong, NEMU will be compiled successfully.

What happened?

You should know how a program is generated in the 程序设计基础 course. But do you have any idea about what happened when a bunch of information is output to the screen during `make` is executed?

To perform a fresh compilation, type

```
make clean
```

to remove the old compilation result, then `make` again.

To run NEMU, type

```
make run
```

However, you will see an error message:

```
nemu: nemu/src/cpu/reg.c:21: reg_test: Assertion `(cpu.gpr[check_reg_index(i)]._16) ==
 (sample[i] & 0xffff)' failed.
```

This message tells you that the program has triggered an assertion fail at line 21 of the file `nemu/src/cpu/reg.c`. If you do not know what is assertion, blame the 程序设计基础 course. If you go to see the line 21 of `nemu/src/cpu/reg.c`, you will discover the failure is in a test function. This failure is expected, because you have not implemented the register structure correctly. Just ignore it now, and you will fix it in PA1.

To debug NEMU with gdb, type

```
make gdb
```

Development Tracing

Once the compilation succeeds, the change of source code will be traced by `git`. Type

```
git log
```

If you see something like

```
commit 4072d39e5b6c6b6837077f2d673cb0b5014e6ef9
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Sun Jul 26 14:30:31 2017 +0800

    > compile NEMU
    161220000
    user
    Linux debian 3.16.0-4-686-pae #1 SMP Debian 3.16.7-3 i686 GNU/Linux
     14:30:31 up  3:44,  2 users,  load average: 0.28, 0.09, 0.07
    3860572d5cc66412bf85332837c381c5c8c1009f
```

this means the change is traced successfully.

If you see the following message while executing make, this means the tracing fails.

```
fatal: Unable to create '/home/user/ics2017/.git/index.lock': File exists.

If no other git process is currently running, this probably means a
git process crashed in this repository earlier. Make sure no other git
process is running and remove the file manually to continue.
```

Try to clean the compilation result and compile again:

```
make clean
make
```

If the error message above always appears, please contact us as soon as possible.

开发跟踪

我们使用`git`对你的实验过程进行跟踪, 不合理的跟踪记录会影响你的成绩. 往届有学长"完成"了某部分实验内容, 但我们找不到相应的git log, 最终该部分内容被视为没有完成. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据. 因此, 请你注意以下事项:

- 请你不定期查看自己的git log, 检查是否与自己的开发过程相符.
- 提交往届代码将被视为没有提交.
- 不要把你的代码上传到公开的地方.
- 总是在工程目录下进行开发, 不要在其它地方进行开发, 然后一次性将代码复制到工程目录下, 这样`git`将不能正确记录你的开发过程.
- 不要修改`Makefile`中与开发跟踪相关的内容.
- 不要删除我们要求创建的分支, 否则会影响我们的脚本运行, 从而影响你的成绩
- 不要清除git log

偶然的跟踪失败不会影响你的成绩. 如果上文中的错误信息总是出现, 请尽快联系我们.

Local Commit

Although the development tracing system will trace the change of your code after every successful compilation, the trace record is not suitable for your development. This is because the code is still buggy at most of the time. Also, it is not easy for you to identify those bug-free traces. Therefore, you should trace your bug-free code manually.

When you want to commit the change, type

```
git add .
git commit --allow-empty
```

The `--allow-empty` option is necessary, because usually the change is already committed by development tracing system. Without this option, `git` will reject no-change commits. If the commit succeeds, you can see a log labeled with your student ID and name by

```
git log
```

To filter out the commit logs corresponding to your manual commit, use `--author` option with `git log`. For details of how to use this option, RTFM.

Submission

Finally, you should submit your project to the submission website. To submit PA0, put your report file (ONLY `.pdf` file is accepted) under the project directory.

```
ics2017
├── 161220000.pdf   # put your report file here
├── init.sh
├── Makefile
├── nanos-lite
├── navy-apps
├── nemu
└── nexus-am
```

Then go back to the project directory, issue

```
make submit
```

This command does 3 things:

1. Cleanup unnecessary files for submission
2. Cleanup unnecessary files in git
3. Create an archive containing the source code and your report. The archive is located in the father directory of the project directory, and it is named by your student ID set in Makefile.

If nothing goes wrong, transfer the archive to your host. Open the archive to double check whether everything is fine. And you can manually submit this archive to the submission website.

RTFSC and Enjoy

If you are new to GNU/Linux and finish this tutorial by yourself, congratulations! You have learn a lot! The most important, you have learn searching the Internet and RTFM for using new tools and trouble-shooting. With these skills, you can solve lots of troubles by yourself during PAs, as well as in the future.

In PA1, the first thing you will do is to RTFSC. If you have troubles during reading the source code, go to RTFM:

- If you can not find the definition of a function, it is probably a library function. Read `man` for more information about that function.
- If you can not understand the code related to hardware details, refer to the i386 manual.

By the way, you will use C language for programming in all PAs. Here is an excellent tutorial about C language. It contains not only C language (such as how to use `printf()` and `scanf()`), but also other elements in a computer system (data structure, computer architecture, assembly language, linking, operating system, network...). It covers most parts of this course. You are strongly recommended to read this tutorial.

Finally, enjoy the journey of PAs, and you will find hardware is not mysterious, so does the computer system! But remember:

- The machine is always right.
- Every line of untested code is always wrong.
- RTFM.

Reminder

This ends PA0. And there is no 必答题 in PA0.

PA1 - 开天辟地的篇章: 最简单的计算机

世界诞生的故事 - 第一章

先驱已经准备好了创造计算机世界的工具. 为了迈出第一步, 他们运用了一些数字电路的知识, 就已经创造出了一个最小的计算机 -- 图灵机. 让我们来看看其中的奥妙.

在进行本PA前, 请在工程目录下执行以下命令进行分支整理, 否则将影响你的成绩:

```
git commit --allow-empty -am "before starting pa1"
git checkout master
git merge pa0
git checkout -b pa1
```

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 30小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 实现单步执行, 打印寄存器状态, 扫描内存 - 2017/09/17 23:59:59
- 阶段2: 实现调试功能的表达式求值 - 2017/09/24 23:59:59
- 最后阶段: 实现所有要求, 提交完整的实验报告 - 2017/10/01 23:59:59

提交说明: 见这里

在开始愉快的PA之旅之前

PA的目的是要实现NEMU, 一款经过简化的x86全系统模拟器. 但什么是模拟器呢?

你小时候应该玩过红白机, 超级玛丽, 坦克大战, 魂斗罗... 它们的画面是否让你记忆犹新? (希望我们之间没有代沟...) 随着时代的发展, 你已经很难在市场上看到红白机的身影了. 当你正在为此感到苦恼的时候, 模拟器的横空出世唤醒了你心中尘封已久的童年回忆. 红白机模拟器可以为你模拟出红白机的所有功能. 有了它, 你就好像有了一个真正的红白机, 可以玩你最喜欢的红白机游戏. 这里是jyy移植的一个小型项目LiteNES, PA工程里面已经带有这个项目, 你可以在如今这个红白机难以寻觅的时代, 再次回味你儿时的快乐时光, 这实在是太神奇了!

配置X Server

Docker container中默认并不带有GUI, 为了运行LiteNES, 你需要根据主机操作系统的类型, 你需要下载不同的X Server:

- Windows用户. 点击这里下载, 安装并打开Xming.
- Mac用户. 点击这里进入XQuartz工程网站, 下载, 安装并打开XQuartz.
- GNU/Linux用户. 系统中已经自带X Server, 你不需要额外下载.

然后根据主机操作系统的类型, 为SSH打开X11转发功能:

- Mac用户和GNU/Linux用户. 在运行 `ssh` 时加入 `-X` 选项即可:

  ```
  ssh -X -p 20022 username@127.0.0.1
  ```

- Windows用户. 在使用 `PuTTY` 登陆时, 在 `PuTTY Configuration` 窗口左侧的目录中选择 `Connection -> SSH -> X11`, 在右侧勾选 `Enable X11 forwarding`, 然后登陆即可.

通过带有X11转发功能的SSH登陆后, 在 `nexus-am/apps/litenes` 目录下执行 `make run`, 即可在弹出的新窗口中运行基于LiteNES的超级玛丽(具体操作请参考该目录下的 `README.md`).

事实上, 我们在PA进行到中期时也需要进行图像的输出, 因此你务必完成X Server的配置.

你被计算机强大的能力征服了, 你不禁思考, 这到底是怎么做到的? 你学习完程序设计基础课程, 但仍然找不到你想要的答案. 但你可以肯定的是, 红白机模拟器只是一个普通的程序, 因为你还是需要像运行Hello World程序那样运行它. 但同时你又觉得, 红白机模拟器又不像一个普通的程序, 它究竟是怎么模拟出一个红白机的世界, 让红白机游戏在这个世界中运行的呢?

事实上, NEMU就是在做类似的事情! 它模拟了一个x86(准确地说, n86, 是x86的一个子集)的世界, 你可以在这个x86世界中执行程序. 换句话说, 你将要在PA中编写一个用来执行其它程序的程序! 为了更好地理解NEMU的功能, 下面将

- 在GNU/Linux中运行Hello World程序

- 在GNU/Linux中通过红白机模拟器玩超级玛丽
- 在GNU/Linux中通过NEMU运行Hello World程序

这三种情况进行比较.

```
+--------------------------------+
|      "Hello World" program     |
+--------------------------------+
|            GNU/Linux           |
+--------------------------------+
|       Computer hardware        |
+--------------------------------+
```

上图展示了"在GNU/Linux中运行Hello World程序"的情况. GNU/Linux操作系统直接运行在计算机硬件上, 对计算机底层硬件进行了抽象, 同时向上层的用户程序提供接口和服务. Hello World程序输出信息的时候, 需要用到操作系统提供的接口, 因此Hello World程序并不是直接运行在计算机硬件上, 而是运行在操作系统(在这里是GNU/Linux)上.

```
+--------------------------------+
|          Super Mario           |
+--------------------------------+
|     Simulated NES hardware     |
+--------------------------------+
|          NES Emulator          |
+--------------------------------+
|            GNU/Linux           |
+--------------------------------+
|       Computer hardware        |
+--------------------------------+
```

上图展示了"在GNU/Linux中通过红白机模拟器玩超级玛丽"的情况. 在GNU/Linux看来, 运行在其上的红白机模拟器NES Emulator和上面提到的Hello World程序一样, 都只不过是一个用户程序而已. 神奇的是, 红白机模拟器的功能是负责模拟出一套完整的红白机硬件, 让超级玛丽可以在其上运行. 事实上, 对于超级玛丽来说, 它并不能区分自己是运行在真实的红白机硬件之上, 还是运行在模拟出来的红白机硬件之上, 这正是"虚拟化"的魔术.

```
+--------------------------------+
|      "Hello World" program     |
+--------------------------------+
|     Simulated x86 hardware     |
+--------------------------------+
|              NEMU              |
+--------------------------------+
|           GNU/Linux            |
+--------------------------------+
|       Computer hardware        |
+--------------------------------+
```

上图展示了"在GNU/Linux中通过NEMU执行Hello World程序"的情况. 在GNU/Linux看来, 运行在其上的NEMU和上面提到的Hello World程序一样, 都只不过是一个用户程序而已. 但NEMU的功能是负责模拟出一套x86硬件, 让程序可以在其上运行. 事实上, 上图只是给出了对NEMU的一个基本理解, 很多细节会在后续PA中逐渐补充. 为了方便叙述, 我们将在NEMU中运行的程序称为"客户程序".

NEMU是什么?

上述描述对你来说也许还有些晦涩难懂, 让我们来看一个ATM机的例子.

ATM机是一个物理上存在的机器, 它的功能需要由物理电路和机械模块来支撑. 例如我们在ATM机上进行存款操作的时候, ATM机都会吭哧吭哧地响, 让我们相信确实是一台真实的机器. 另一方面, 现在第三方支付平台也非常流行, 例如支付宝. 事实上, 我们可以把支付宝APP看成一个虚拟的ATM机, 在这个虚拟的ATM机里面, 真实ATM机具备的所有功能, 包括存款, 取款, 查询余额, 转账等等, 都通过支付宝APP这个程序来实现.

同样地, NEMU就是一个虚拟出来的计算机系统, 物理计算机中的基本功能, 在NEMU中都是通过程序来实现的. 要虚拟出一个计算机系统并没有你想象中的那么困难. 我们可以把计算机看成由若干个硬件部件组成, 这些部件之间相互协助, 完成"运行程序"这件事情. 在NEMU中, 每一个硬件部件都由一个程序相关的数据对象来模拟, 例如变量, 数组, 结构体等; 而对这些部件的操作则通过对相应数据对象的操作来模拟. 例如NEMU中使用数组来模拟内存, 那么对这个数组进行读写则相当于对内存进行读写.

我们可以把实现NEMU的过程看成是开发一个支付宝APP. 不同的是, 支付宝具备的是真实ATM机的功能, 是用来交易的; 而NEMU具备的是物理计算机系统的功能, 是用来执行程序的. 因此我们说, NEMU是一个用来执行其它程序的程序.

你或许还对虚拟机和模拟器这两个相似的概念感到疑惑, 毕竟它们都表示用程序的功能来实现某些东西. 虚拟机就是用程序虚拟出来的机器; 而模拟器的范围则更加广泛, 可以用程序来模拟天体运动, 大气环流, 分子碰撞等等, 然而这些模拟的对象并不是一个计算机系统. 当我们用模拟器来模拟一个计算机系统的时候, 它和虚拟机在本质上并没有太大的差异. 所以我们说NEMU是个x86模拟器, 或者说NEMU是个x86的虚拟机, 其实可以认为是同一个意思: NEMU是用程序来实现一个计算机系统的功能, 并不是一个物理上的计算机.

初识虚拟化

假设你在Windows中使用Docker安装了一个GNU/Linux container, 然后在container中完成PA, 通过NEMU运行Hello World程序. 在这样的情况下, 尝试画出相应的层次图.

嗯, 事实上在Windows中运行Docker container的眞实情况有点复杂, 有兴趣的同学可以参考虚拟机和container的区别.

NEMU的威力会让你感到吃惊! 它不仅仅能运行Hello World这样的小程序, 在PA的后期, 你将会在NEMU中运行仙剑奇侠传(很酷! %>_<%). 完成PA之后, 你在程序设计课上对程序的认识会被彻底颠覆, 你会觉得计算机不再是一个神秘的黑盒, 甚至你会发现创造一个属于自己的计算机不再是遥不可及!

让我们来开始这段激动人心的旅程吧! 但请不要忘记:

- 机器永远是对的
- 未测试代码永远是错的
- RTFM

开天辟地的故事

先驱希望创造一个计算机的世界, 并赋予它执行程序的使命. 让我们一起来帮助他们, 体验创世的乐趣.

大家都上过程序设计课程, 知道程序就是由代码和数据组成. 例如一个求 `1+2+...+100` 的程序, 大家不费吹灰之力就可以写出一个程序来完成这件事情. 不难理解, 数据就是程序处理的对象, 代码则描述了程序希望如何处理这些数据. 先不说仙剑奇侠传这个庞然大物, 为了执行哪怕最简单的程序, 最简单的计算机又应该长什么样呢?

为了执行程序, 首先要解决的第一个问题, 就是要把程序放在哪里. 显然, 我们不希望自己创造的计算机只能执行小程序. 因此, 我们需要一个足够大容量的部件, 来放下各种各样的程序, 这个部件就是存储器. 于是, 先驱创造了存储器, 并把程序放在存储器中, 等待着CPU去执行.

等等, CPU是谁? 你也许很早就听说过它了, 不过现在还是让我们来重新介绍一下它吧. CPU是先驱最伟大的创造, 从它的中文名字"中央处理器"就看得出它被赋予了至高无上的荣耀: CPU是负责处理数据的核心电路单元, 也就是说, 程序的执行全靠它了. 但只有存储器的计算机还是不能进行计算. 自然地, CPU需要肩负起计算的重任, 先驱为CPU创造了运算器, 这样就可以对数据进行各种处理了. 如果觉得运算器太复杂, 那就先来考虑一个加法器吧.

先驱发现, 有时候程序需要对同一个数据进行连续的处理. 例如要计算 `1+2+...+100` , 就要对部分和 `sum` 进行累加, 如果每完成一次累加都需要把它写回存储器, 然后又把它从存储器中读出来继续加, 这样就太不方便了. 同时天下也没有免费的午餐, 存储器的大容量也是需要付出相应的代价的, 那就是速度慢, 这是先驱也无法违背的材料特性规律. 于是先驱为CPU创造了寄存器, 可以让CPU把正在处理中的数据暂时存放在其中. 为了兼容x86, 我们选择了一个稍微有点复杂的寄存器结构:

```
 31              23              15              7               0
+----------------+----------------+----------------+----------------+
|                                EAX       AH       AX      AL       |
|----------------+----------------+----------------+----------------|
|                                EDX       DH       DX      DL       |
|----------------+----------------+----------------+----------------|
|                                ECX       CH       CX      CL       |
|----------------+----------------+----------------+----------------|
|                                EBX       BH       BX      BL       |
|----------------+----------------+----------------+----------------|
|                                EBP                BP               |
|----------------+----------------+----------------+----------------|
|                                ESI                SI               |
|----------------+----------------+----------------+----------------|
|                                EDI                DI               |
|----------------+----------------+----------------+----------------|
|                                ESP                SP               |
+----------------+----------------+----------------+----------------+
```

其中

- `EAX`, `EDX`, `ECX`, `EBX`, `EBP`, `ESI`, `EDI`, `ESP` 是32位寄存器;
- `AX`, `DX`, `CX`, `BX`, `BP`, `SI`, `DI`, `SP` 是16位寄存器;
- `AL`, `DL`, `CL`, `BL`, `AH`, `DH`, `CH`, `BH` 是8位寄存器. 但它们在物理上并不是相互独立的, 例如 `EAX` 的低16位是 `AX`, 而 `AX` 又分成 `AH` 和 `AL`. 这样的结构有时候在处理不同长度的数据时能提供一些便利.

寄存器的速度很快, 但容量却很小, 和存储器的特性正好互补, 它们之间也许会交织出新的故事呢, 不过目前我们还是顺其自然吧.

为了让强大的CPU成为忠诚的奴仆, 先驱还设计了"指令", 用来指示CPU对数据进行何种处理. 这样, 我们就可以通过指令来控制CPU, 让它做我们想做的事情了.

有了指令以后, 先驱提出了一个划时代的设想: 能否让程序来自动控制计算机的执行? 为了实现这个设想, 先驱和CPU作了一个简单的约定: 当执行完一条指令之后, 就继续执行下一条指令. 但CPU怎么知道现在执行到哪一条指令呢? 为此, 先驱为CPU创造了一个特殊的计数器, 叫"程序计数器"(Program Counter, PC), 它在x86中的名字叫 `EIP`.

```
 31              23              15              7               0
+----------------+----------------+----------------+----------------+
|                     EIP (INSTRUCTION POINTER)                      |
+----------------+----------------+----------------+----------------+
```

从此以后, 计算机就只需要做一件事情:

```
while (1) {
  从EIP指示的存储器位置取出指令;
  执行指令;
  更新EIP;
}
```

这样, 我们就有了一个足够简单的计算机了. 我们只要将一段指令序列放置在存储器中, 然后让PC指向第一条指令, 计算机就会自动执行这一段指令序列, 永不停止. 这个全自动的执行过程实在是太美妙了! 事实上, 开拓者图灵在1936年就已经提出类似的核心思想, "计算机之父"可谓名不虚传. 而这个流传至今的核心思想, 就是"存储程序". 为了表达对图灵的敬仰, 我们也把上面这个最简单的计算机称为"图灵机"(Turing Machine, TRM). 或许你已经听说过"图灵机"这个作为计算模型时的概念, 不过在这里我们只强调作为一个最简单的真实计算机需要满足哪些条件:

- 结构上, TRM有存储器, 有PC, 有寄存器, 有加法器
- 工作方式上, TRM不断地重复以下过程: 从PC指示的存储器位置取出指令, 执行指令, 然后更新PC

咦? 存储器, 计数器, 寄存器, 加法器, 这些不都是数字电路课上学习过的部件吗? 也许你会觉得难以置信, 但先驱说, 你正在面对着的那台无所不能的计算机, 就是由数字电路组成的! 不过, 我们在程序设计课上写的程序是C代码. 但如果计算机真的是个只能懂0和1的巨大数字电路, 这个冷冰冰的电路又是如何理解凝结了人类智慧结晶的C代码的呢? 先驱说, 计算机诞生的那些年还没有C语言, 大家都是直接编写对人类来说晦涩难懂的机器指令, 那是他们所见过的最早的对电子计算机的编程方式了. 后来人们发明了高级语言和编译器, 能把我们写的高级语言代码进行各种处理, 最后生成功能等价的, CPU能理解的指令. CPU执行这些指令, 就相当于是执行了我们写的代码. 今天的计算机本质上还是"存储程序"这种天然愚钝的工作方式, 是经过了无数计算机科学家们的努力, 我们今天才可以轻松地使用计算机.

RTFSC

既然TRM那么简单, 就让我们在NEMU里面实现一个TRM吧.

不过我们还是先来介绍一下框架代码. 框架代码内容众多, 其中包含了很多在后续阶段中才使用的代码. 随着实验进度的推进, 我们会逐渐解释所有的代码. 因此在阅读代码的时候, 你只需要关心和当前进度相关的模块就可以了, 不要纠缠于和当前进度无关的代码, 否则将会给你的心灵带来不必要的恐惧.

```
ics2017
├── init.sh        # 初始化脚本
├── Makefile       # 用于工程打包提交
├── nanos-lite     # 微型操作系统内核
├── navy-apps      # 应用程序集
├── nemu           # NEMU
└── nexus-am       # 抽象计算机
```

目前我们只需要关心NEMU的内容, 其它内容会在将来进行介绍. NEMU主要由4个模块构成: monitor, CPU, memory, 设备. 我们已经在上一小节简单介绍了CPU和memory的功能, 设备会在PA2中介绍, 目前不必关心. monitor位于这个虚拟计算机系统之外, 主要用于监视这个虚拟计算机系统是否正确运行. monitor从概念上并不属于一个计算机的必要组成部分, 但对NEMU来说, 它是必要的基础设施. 它除了负责与GNU/Linux进行交互(例如读写文件)之外, 还带有调试器的功能, 为NEMU的调试提供了方便的途径. 缺少monitor模块, 对NEMU的调试将会变得十分困难.

代码中 `nemu` 目录下的源文件组织如下(部分目录下的文件并未列出):

```
nemu
├── include                     # 存放全局使用的头文件
│   ├── common.h                # 公用的头文件
│   ├── cpu
│   │   ├── decode.h            # 译码相关
│   │   ├── exec.h              # 执行相关
│   │   ├── reg.h               # 寄存器结构体的定义
│   │   └── rtl.h               # RTL指令
│   ├── debug.h                 # 一些方便调试用的宏
│   ├── device                  # 设备相关
│   ├── macro.h                 # 一些方便的宏定义
│   ├── memory                  # 访问内存相关
│   ├── monitor
│   │   ├── expr.h              # 表达式求值相关
│   │   ├── monitor.h
│   │   └── watchpoint.h        # 监视点相关
│   └── nemu.h
├── Makefile                    # 指示NEMU的编译和链接
├── Makefile.git                # git版本控制相关
├── runall.sh                   # 一键测试脚本
└── src                         # 源文件
    ├── cpu
    │   ├── decode              # 译码相关
    │   ├── exec                # 执行相关
    │   ├── intr.c              # 中断处理相关
    │   └── reg.c               # 寄存器相关
    ├── device                  # 设备相关
    ├── main.c                  # 你知道的...
    ├── memory
    │   └── memory.c            # 访问内存的接口函数
    ├── misc
    │   └── logo.c              # "i386"的logo
    └── monitor
        ├── cpu-exec.c          # 指令执行的主循环
        ├── diff-test
        ├── debug               # 简易调试器相关
        │   ├── expr.c          # 表达式求值的实现
        │   ├── ui.c            # 用户界面相关
        │   └── watchpoint.c    # 监视点的实现
        └── monitor.c
```

为了给出一份可以运行的框架代码, 代码中实现了 `mov` 指令的功能, 并附带一个 `mov` 指令序列的默认客户程序. 另外, 部分代码中会涉及一些硬件细节(例如 `nemu/src/cpu/decode/modrm.c`). 在你第一次阅读代码的时候, 你需要尽快掌握NEMU的框架, 而不要纠缠于这些细节. 随着PA的进行, 你会反复回过头来探究这些细节.

大致了解上述的目录树之后, 你就可以开始阅读代码了. 至于从哪里开始, 就不用多费口舌了吧.

需要多费口舌吗?

嗯... 如果你觉得提示还不够, 那就来一个劲爆的: 回忆程序设计课的内容, 一个程序从哪里开始执行呢?

如果你不屑于回答这个问题, 不妨先冷静下来. 其实这是一个值得探究的问题, 你会在将来重新审视它.

对vim的使用感到困难?

在PA0的强迫之下, 你不得不开始学习使用vim. 如果现在你已经不再认为vim是个到处是bug的编辑器, 就像简明vim练级攻略里面说的, 你已经通过了存活阶段. 接下来就是漫长的修行阶段了, 每天学习一两个vim中的功能, 累积经验值, 很快你就会发现自己已经连升几级. 不过最重要的还是坚持, 只要你在PA1中坚持使用vim, PA1结束之后, 你就会发现vim的熟练度已经大幅提升! 你还可以搜一搜vim的键盘图, 像英雄联盟中满满的快捷键, 说不定能激发起你学习vim的兴趣.

NEMU开始执行的时候, 首先会调用 `init_monitor()` 函数(在 `nemu/src/monitor/monitor.c` 中定义) 进行一些和monitor相关的初始化工作, 我们对其中几项初始化工作进行一些说明. `reg_test()` 函数(在 `nemu/src/cpu/reg.c` 中定义)会生成一些随机的数据, 对寄存器实现的正确性进行测试. 若不正确, 将会触发assertion fail.

实现正确的寄存器结构体

我们在PA0中提到, 运行NEMU会出现assertion fail的错误信息, 这是因为框架代码并没有正确地实现用于模拟寄存器的结构体 `CPU_state`, 现在你需要实现它了(结构体的定义在 `nemu/include/cpu/reg.h` 中). 关于i386寄存器的更多细节, 请查阅i386手册. Hint: 使用匿名union.

然后通过调用 `load_img()` 函数(在 `nemu/src/monitor/monitor.c` 中定义)读入带有客户程序的镜像文件. 我们知道内存是一种RAM, 是一种易失性的存储介质, 这意味着计算机刚启动的时候, 内存中的数据都是无意义的; 而BIOS是固化在ROM中的, 它是一种非易失性的存储介质, BIOS中的内容不会因为断电而丢失. 因此在真实的计算机系统中, 计算机启动后首先会把控制权交给BIOS, BIOS经过一系列初始化工作之后, 再从磁盘中将有意义的程序读入内存中执行. 对这个过程的模拟需要了解很多超出本课程范围的细节, 我们在这里做了简化, 让monitor直接把一个有意义的客户程序镜像guest prog读入到一个固定的内存位置 `0x100000`, 这个程序是运行NEMU的一个参数, 在运行NEMU的命令中指定, 缺省时将把上文提到的 `mov` 程序作为客户程序(参考 `load_default_img()` 函数). 这时内存的布局如下:

```
0            0x100000
---------------------------------------------
|                 |                 |
|                 |    guest prog   |
|                 |                 |
---------------------------------------------
                  ^
                  |
                 eip
```

接下来调用 `restart()` 函数(在 `nemu/src/monitor/monitor.c` 中定义), 它模拟了"计算机启动"的功能, 进行一些和"计算机启动"相关的初始化工作, 一个重要的工作就是将 `%eip` 的初值设置为刚才我们约定的内存位置 `0x100000` , 这样就可以让CPU从我们约定的内存位置开始执行程序了.

monitor的其它初始化工作我们会在后续实验内容中介绍, 目前可以不必关心它们的细节, 最后通过调用 `welcome()` 函数输出欢迎信息和NEMU的编译时间.

monitor的初始化工作结束后, NEMU会进入用户界面主循环 `ui_mainloop()` (在 `nemu/src/monitor/debug/ui.c` 中定义), 输出NEMU的命令提示符:

```
(nemu)
```

代码已经实现了几个简单的命令, 它们的功能和GDB是很类似的. 输入 `c` 之后, NEMU开始进入指令执行的主循环 `cpu_exec()` (在 `nemu/src/monitor/cpu-exec.c` 中定义). `cpu_exec()` 模拟了CPU的工作方式: 不断执行指令. `exec_wrapper()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)的功能 让CPU执行当前 `%eip` 指向的一条指令, 然后更新 `%eip` . 已经执行的指令会输出到日志文件 `log.txt` 中, 你可以打开 `log.txt` 来查看它们.

究竟要执行多久?

在 `cmd_c()` 函数中, 调用 `cpu_exec()` 的时候传入了参数 `-1` , 你知道这是什么意思吗?

执行指令的相关代码在 `nemu/src/cpu/exec` 目录下. 其中一个重要的部分定义在 `nemu/src/cpu/exec/exec.c` 文件中的 `opcode_table` 数组, 在这个数组中, 你可以看到框架代码中都已经实现了哪些指令. 其中 `EMPTY` 代表对应的指令还没有实现(也可能是x86中不存在该指令). 在以后的PA中, 随着你实现越来越多的指令, 这个数组会逐渐被它们代替. 关于指令执行的详细解释和 `exec_wrapper()` 相关的内容需要涉及很多细节, 目前你不必关心, 我们将会在PA2中进行解释.

温故而知新

`opcode_table` 到底是个什么类型的数组? 如果你感到困惑, 你需要马上复习程序设计的知识了. 这里有一份十分优秀的C语言教程. 事实上, 我们已经在PA0中提到过这份教程了, 如果你觉得你的程序设计知识比较生疏, 而又没有在PA0中阅读这份教程, 请你务必阅读它.

NEMU将不断执行指令, 直到遇到以下情况之一, 才会退出指令执行的循环:

- 达到要求的循环次数.
- 客户程序执行了 `nemu_trap` 指令. 这是一条特殊的指令, 机器码为 `0xd6` . 如果你查阅i386手册, 你会发现x86中并没有这条指令, 它是为了在NEMU中让客户程序指示执行的结束而加入的.

当你看到NEMU输出以下内容时:

```
nemu: HIT GOOD TRAP at eip = 0x00100026
```

说明客户程序已经成功地结束运行. 退出 `cpu_exec()` 之后, NEMU将返回到 `ui_mainloop()` , 等待用户输入命令. 你为了再次运行程序, 你需要键入 `q` 退出NEMU, 然后重新运行.

谁来指示程序的结束?

在程序设计课上老师告诉你, 当程序执行到 `main()` 函数返回处的时候, 程序就退出了, 你对此深信不疑. 但你是否怀疑过, 凭什么程序执行到 `main()` 函数的返回处就结束了? 如果有人告诉你, 程序设计课上老师的说法是错的, 你有办法来证明/反驳吗? 如果你对此感兴趣, 请在互联网上搜索相关内容.

最后我们聊聊代码中一些值得注意的地方.

- 三个对调试有用的宏(在 `nemu/include/debug.h` 中定义)
  - `Log()` 是 `printf()` 的升级版, 专门用来输出调试信息, 同时还会输出使用 `Log()` 所在的源文件, 行号和函数. 当输出的调试信息过多的时候, 可以很方便地定位到代码中的相关位置
  - `Assert()` 是 `assert()` 的升级版, 当测试条件为假时, 在assertion fail之前可以输出一些信息
  - `panic()` 用于输出信息并结束程序, 相当于无条件的assertion fail

  代码中已经给出了使用这三个宏的例子, 如果你不知道如何使用它们, RTFSC.
- 内存通过在 `nemu/src/memory/memory.c` 中定义的大数组 `pmem` 来模拟. 在客户程序运行的过程中, 总是使用 `vaddr_read()` 和 `vaddr_write()` 访问模拟的内存. vaddr, paddr分别代表虚拟地址和物理地址. 这些概念在将来会用到, 但从现在开始保持接口的一致性可以在将来避免一些不必要的麻烦.

理解框架代码

你需要结合上述文字理解NEMU的框架代码. 需要注意的是, 阅读代码也是有技巧的, 如果你分开阅读框架代码和上述文字, 你可能会觉得阅读之后没有任何效果. 因此, 你需要一边阅读上述文字, 一边阅读相应的框架代码.

如果你不知道"怎么才算是看懂了框架代码", 你可以先尝试进行后面的任务. 如果发现不知道如何下手, 再回来仔细阅读这一页面. 理解框架代码是一个螺旋上升的过程, 不同的阶段有不同的重点. 你不必因为看不懂某些细节而感到沮丧, 更不要试图一次把所有代码全部看明白.

讲义中的知识点很多, 在实验的不同阶段对同一个知识点的理解也会有所不同. 我们建议你在完成相应阶段的任务之后回过头来重新阅读一遍讲义的内容, 你很可能会有不一样的收获.

事实上, TRM的实现已经都蕴含在上述的介绍中了.

- 存储器是个在 `nemu/src/memory/memory.c` 中定义的大数组
- PC和通用寄存器都在 `nemu/include/cpu/reg.h` 中的结构体中定义
- 加法器在... 嗯, 框架代码这部分的内容有点复杂, 不过它并不影响我们对TRM的理解, 我们还是在PA2里面再介绍它吧
- TRM的工作方式通过 `cpu_exec()` 和 `exec_wrapper()` 体现

在NEMU中, 我们只需要一些很简单的C语言知识就可以理解最简单的计算机的工作方式, 真应该感谢先驱啊.

基础设施: 简易调试器

基础设施 - 提高项目开发的效率

基础设施是指支撑项目开发的各种工具和手段. 原则上基础设施并不属于课本上知识的范畴, 但是作为一个有一定规模的项目, 基础设施的好坏甚至会影响到项目的推进, 这是你在程序设计课上体会不到的.

事实上, 你已经体会过基础设施给你带来的便利了. 我们的框架代码已经提供了Makefile来对NEMU进行一键编译. 现在我们来假设我们没有提供一键编译的功能, 你需要通过手动键入 `gcc` 命令的方式来编译源文件: 假设你手动输入一条 `gcc` 命令需要10秒的时间(你还需要输入很多编译选项, 能用10秒输入完已经是非常快的了), 而NEMU工程下有30个源文件, 为了编译出NEMU的可执行文件, 你需要花费多少时间? 然而你还需要在开发NEMU的过程中不断进行编译, 假设你需要编译500次NEMU才能完成PA, 一学期下来, 你仅仅花在键入编译命令上的时间有多少?

有的项目即使使用工具也需要花费较多时间来构建. 例如硬件开发平台 `vivado` 一般需要花费半小时到一小时不等的时间来生成比特文件, 也就是说, 你编写完代码之后, 可能需要等待一小时之后才能验证你的代码是否正确. 这是因为, 这个过程不像编译程序这么简单, 其中需要处理很多算法上的NPC问题. 为了生成一个还不错的比特文件, `vivado` 需要付出比 `gcc` 更大的代价来解决这些NPC问题. 这时候基础设施的作用就更加重要了, 如果能有工具可以帮助你一次进行多个方面的验证, 就会帮助你节省下来无数个"一小时".

Google内部的开发团队非常重视基础设施的建设, 他们把可以让一个项目得益的工具称为Adder, 把可以让多个项目都得益的工具称为Multiplier. 顾名思义, 这些工具可以成倍提高项目开发的效率. 在学术界, 不少科研工作的目标也是提高开发效率, 例如自动bug检测和修复, 自动化验证, 易于开发的编程模型等等. 在PA中, 基础设施也会体现在不同的方面, 我们会在将来对其它方面进行讨论.

你将来肯定会参与比PA更大的项目, 如何提高项目开发的效率也是一个很重要的问题. 希望在完成PA的过程中, 你能够对基础设施有新的认识: 有代码的地方, 就有基础设施. 随着知识的积累, 将来的你或许也会投入到这些未知的领域当中, 为全世界的开发者作出自己的贡献.

简易调试器是NEMU中一项非常重要的基础设施. 我们知道NEMU是一个用来执行其它客户程序的程序, 这意味着, NEMU可以随时了解客户程序执行的所有信息. 然而这些信息对外面的调试器(例如GDB)来说, 是不容易获取的. 例如在通过GDB调试NEMU的时候, 你将很难在NEMU中运行的客户程序中设置断点, 但对于NEMU来说, 这是一件不太困难的事情.

为了提高调试的效率, 同时也作为熟悉框架代码的练习, 我们需要在monitor中实现一个具有如下功能的简易调试器 (相关部分的代码在 `nemu/src/monitor/debug` 目录下), 如果你不清楚命令的格式和功能, 请参考如下表格:

命令	格式	使用举例	说明
帮助(1)	`help`	`help`	打印命令的帮助信息
继续运行(1)	`c`	`c`	继续运行被暂停的程序
退出(1)	`q`	`q`	退出NEMU
单步执行	`si [N]`	`si 10`	让程序单步执行 `N` 条指令后暂停执行, 当 `N` 没有给出时, 缺省为 `1`
打印程序状态	`info SUBCMD`	`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` 支持的运算请见调试中的表达式求值小节
扫描内存(2)	`x N EXPR`	`x 10 $esp`	求出表达式 `EXPR` 的值, 将结果作为起始内存 地址, 以十六进制形式输出连续的 `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, 暂停程序执行
删除监视点	`d N`	`d 2`	删除序号为 `N` 的监视点

备注:

- (1) 命令已实现
- (2) 与GDB相比, 我们在这里做了简化, 更改了命令的格式

总有一天会找上门来的bug

你需要在将来的PA中使用这些功能来帮助你进行NEMU的调试. 如果你的实现是有问题的, 将来你有可能会面临以下悲惨的结局: 你实现了某个新功能之后, 打算对它进行测试, 通过扫描内存的功能来查看一段内存, 发现输出并非预期结果. 你认为是刚才实现的新功能有问题, 于是对它进行调试. 经过了几天几夜的调试之后, 你泪流满面地发现, 原来是扫描内存的功能有bug!

如果你想避免类似的悲惨结局, 你需要在实现一个功能之后对它进行充分的测试. 随着时间的推移, 发现同一个bug所需要的代价会越来越大.

解析命令

NEMU通过 `readline` 库与用户交互, 使用 `readline()` 函数从键盘上读入命令. 与 `gets()` 相比, `readline()` 提供了"行编辑"的功能, 最常用的功能就是通过上, 下方向键翻阅历史记录. 事实上, shell程序就是通过 `readline()` 读入命令的. 关于 `readline()` 的功能和返回值等信息, 请查阅

```
man readline
```

从键盘上读入命令后, NEMU需要解析该命令, 然后执行相关的操作. 解析命令的目的是识别命令中的参数, 例如在 `si 10` 的命令中识别出 `si` 和 `10` , 从而得知这是一条单步执行10条指令的命令. 解析命令的工作是通过一系列的字符串处理函数来完成的, 例如框架代码中的 `strtok()` . `strtok()` 是C语言中的标准库函数, 如果你从来没有使用过 `strtok()` , 并且打算继续使用框架代码中的 `strtok()` 来进行命令的解析, 请务必查阅

```
man strtok
```

另外, `cmd_help()` 函数中也给出了使用 `strtok()` 的例子. 事实上, 字符串处理函数有很多, 键入以下内容:

```
man 3 str<TAB><TAB>
```

其中 `<TAB>` 代表键盘上的TAB键. 你会看到很多以str开头的函数, 其中有你应该很熟悉的 `strlen()` , `strcpy()` 等函数. 你最好都先看看这些字符串处理函数的manual page, 了解一下它们的功能, 因为你很可能会用到其中的某些函数来帮助你解析命令. 当然你也可以编写你自己的字符串处理函数来解析命令.

另外一个值得推荐的字符串处理函数是 `sscanf()` , 它的功能和 `scanf()` 很类似, 不同的是 `sscanf()` 可以从字符串中读入格式化的内容, 使用它有时候可以很方便地实现字符串的解析. 如果你从来没有使用过它们, RTFM, 或者到互联网上查阅相关资料.

单步执行

单步执行的功能十分简单, 而且框架代码中已经给出了模拟CPU执行方式的函数, 你只要使用相应的参数去调用它就可以了. 如果你仍然不知道要怎么做, RTFSC.

打印寄存器

打印寄存器就更简单了, 执行 `info r` 之后, 直接用 `printf()` 输出所有寄存器的值即可. 如果你从来没有使用过 `printf()` , 请到互联网上搜索相关资料. 如果你不知道要输出什么, 你可以参考GDB中的输出.

扫描内存

扫描内存的实现也不难, 对命令进行解析之后, 先求出表达式的值. 但你还没有实现表达式求值的功能, 现在可以先实现一个简单的版本: 规定表达式 `EXPR` 中只能是一个十六进制数, 例如

```
x 10 0x100000
```

这样的简化可以让你暂时不必纠缠于表达式求值的细节. 解析出待扫描内存的起始地址之后, 你就使用循环将指定长度的内存数据通过十六进制打印出来. 如果你不知道要怎么输出, 同样的, 你可以参考GDB中的输出.

实现了扫描内存的功能之后, 你可以打印 `0x100000` 附近的内存, 你应该会看到程序的代码, 和默认镜像进行对比, 看看你的实现是否正确.

实现单步执行, 打印寄存器, 扫描内存

熟悉了NEMU的框架之后, 这些功能实现起来都很简单, 同时我们对输出的格式不作硬性规定, 就当做是熟悉GNU/Linux编程的一次练习吧.

不知道如何下手? 嗯, 看来你需要再阅读一遍RTFSC小节的内容了. 不敢下手? 别怕, 放手去写! 编译运行就知道写得对不对. 代码改挂了, 就改回来呗; 代码改得面目全非, 还有 `git` 呀!

温馨提示

PA1阶段1到此结束.

表达式求值

数学表达式求值

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:

- 十进制整数
- `+`, `-`, `*`, `/`
- `(`, `)`
- 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 `TK_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 `+`, `-`, `*`, `/`, 但这对于有些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, 你会如何进行处理?), 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.

系统设计的黄金法则 -- KISS法则

这里的 `KISS` 是 `Keep It Simple, Stupid` 的缩写, 它的中文翻译是: 不要在一开始追求绝对的完美.

你已经学习过程序设计基础, 这意味着你已经学会写程序了, 但这并不意味着你可以顺利地完成PA, 因为在现实世界中, 我们需要的是可以运行的system, 而不是求阶乘的小程序. NEMU作为一个麻雀虽小, 五脏俱全的小型系统, 其代码量达到3000多行(不包括空行). 随着

PA的进行, 代码量会越来越多, 各个模块之间的交互也越来越复杂, 工程的维护变得越来越困难, 一个很弱智的bug可能需要调好几天. 在这种情况下, 系统能跑起来才是王道, 跑不起来什么都是浮云, 追求面面俱到只会增加代码维护的难度.

唯一可以把你从bug的混沌中拯救出来的就是KISS法则, 它的宗旨是从易到难, 逐步推进, 一次只做一件事, 少做无关的事. 如果你不知道这是什么意思, 我们以上文提到的 `str` 成员缓冲区溢出问题来作为例子. KISS法则告诉你, 你应该使用 `assert(0)` , 就算不"得体"地处理上述问题, 仍然不会影响表达式求值的核心功能的正确性. 如果你还记得调试公理, 你会发现两者之间是有联系的: 调试公理第二点告诉你, 未测试代码永远是错的. 与其一下子写那么多"错误"的代码, 倒不如使用 `assert(0)` 来有效帮助你减少这些"错误".

如果把KISS法则放在软件工程领域来解释, 它强调的就是多做单元测试: 写一个函数, 对它进行测试, 正确之后再写下一个函数, 再对它进行测试... 一种好的测试方式是使用assertion进行验证, `reg_test()` 就是这样的例子. 学会使用assertion, 对程序的测试和调试都百利而无一害.

KISS法则不但广泛用在计算机领域, 就连其它很多领域也视其为黄金法则, 这里有一篇文章举出了很多的例子, 我们强烈建议你阅读它, 体会KISS法则的重要性.

实现算术表达式的词法分析

你需要完成以下的内容:

- 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.
- 在成功识别出token后, 将token的信息依次记录到 `tokens` 数组中.

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" |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

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 pa
ir 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, 我们要怎么正确地将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 )"
```

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:

- 非运算符的token不是dominant operator.
- 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.

- 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.
- 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);
    }
  }
}
```

实现算术表达式的递归求值

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.

实现带有负数的算术表达式的求值(选做)

在上述实现中, 我们并没有考虑负数的问题, 例如

```
"1 + -1"
"--1"    /* 我们不实现自减运算, 这里应该解释成 -(-1) = 1 */
```

它们会被判定为不合法的表达式. 为了实现负数的功能, 你需要考虑两个问题:

- 负号和减号都是 `-`, 如何区分它们?
- 负号是个单目运算符, 分裂的时候需要注意什么?

你可以选择不实现负数的功能, 但你很快就要面临类似的问题了.

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:

- 为了方便统一, 我们认为所有结果都是`uint32_t`类型.
- 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.

实现更复杂的表达式求值

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, 我们在PA中暂不实现.

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.

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到一行里面, 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".

温馨提示

PA1阶段2到此结束.

监视点

监视点的功能是监视一个表达式的值何时发生变化. 如果你从来没有使用过监视点, 请在GDB中体验一下它的作用.

简易调试器允许用户同时设置多个监视点, 删除监视点, 因此我们最好使用链表将监视点的信息组织起来. 框架代码中已经定义好了监视点的结构体(在 `nemu/include/monitor/watchpoint.h` 中):

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;

  /* TODO: Add more members if necessary */


} WP;
```

但结构体中只定义了两个成员: `NO` 表示监视点的序号, `next` 就不用多说了吧. 为了实现监视点的功能, 你需要根据你对监视点工作原理的理解在结构体中增加必要的成员. 同时我们使用"池"的数据结构来管理监视点结构体, 框架代码中已经给出了一部分相关的代码(在 `nemu/src/monitor/debug/watchpoint.c` 中):

```
static WP wp_pool[NR_WP];
static WP *head, *free_;
```

代码中定义了监视点结构的池 `wp_pool`, 还有两个链表 `head` 和 `free_`, 其中 `head` 用于组织使用中的监视点结构, `free_` 用于组织空闲的监视点结构, `init_wp_pool()` 函数会对两个链表进行了初始化.

实现监视点池的管理

为了使用监视点池, 你需要编写以下两个函数(你可以根据你的需要修改函数的参数和返回值):

```
WP* new_wp();
void free_wp(WP *wp);
```

其中 `new_wp()` 从 `free_` 链表中返回一个空闲的监视点结构, `free_wp()` 将 `wp` 归还到 `free_` 链表中, 这两个函数会作为监视点池的接口被其它函数调用. 需要注意的是, 调用 `new_wp()` 时可能会出现没有空闲监视点结构的情况, 为了简单起见, 此时可以通

过 `assert(0)` 马上终止程序. 框架代码中定义了32个监视点结构, 一般情况下应该足够使用, 如果你需要更多的监视点结构, 你可以修改 `NR_WP` 宏的值.

这两个函数里面都需要执行一些链表插入, 删除的操作, 对链表操作不熟悉的同学来说, 这可以作为一次链表的练习.

温故而知新(2)

框架代码中定义 `wp_pool` 等变量的时候使用了关键字 `static` , `static` 在此处的含义是什么? 为什么要在此处使用它?

实现了监视点池的管理之后, 我们就可以考虑如何实现监视点的相关功能了. 具体的, 你需要实现以下功能:

- 当用户给出一个待监视表达式时, 你需要通过 `new_wp()` 申请一个空闲的监视点结构, 并将表达式记录下来. 每当 `cpu_exec()` 执行完一条指令, 就对所有待监视的表达式进行求值(你之前已经实现了表达式求值的功能了), 比较它们的值有没有发生变化, 若发生了变化, 程序就因触发了监视点而暂停下来, 你需要将 `nemu_state` 变量设置为 `NEMU_STOP` 来达到暂停的效果. 最后输出一句话提示用户触发了监视点, 并返回到 `ui_mainloop()` 循环中等待用户的命令.
- 使用 `info w` 命令来打印使用中的监视点信息, 至于要打印什么, 你可以参考GDB中 `info watchpoints` 的运行结果.
- 使用 `d` 命令来删除监视点, 你只需要释放相应的监视点结构即可.

实现监视点

你需要实现上文描述的监视点相关功能, 实现了表达式求值之后, 监视点实现的重点就落在了链表操作上. 如果你仍然因为链表的实现而感到调试困难, 请尝试学会使用assertion.

在同一时刻触发两个以上的监视点也是有可能的, 你可以自由决定如何处理这些特殊情况, 我们对此不作硬性规定.

断点

断点的功能是让程序暂停下来, 从而方便查看程序某一时刻的状态. 事实上, 我们可以很容易地用监视点来模拟断点的功能:

```
w $eip == ADDR
```

其中 `ADDR` 为设置断点的地址. 这样程序执行到 `ADDR` 的位置时就会暂停下来.

调试器设置断点的工作方式和上述通过监视点来模拟断点的方法大相径庭. 事实上, 断点的工作原理, 竟然是三十六计之中的"偷龙转凤"! 如果你想揭开这一神秘的面纱, 你可以阅读这篇文章. 了解断点的工作原理之后, 可以尝试思考下面的两个问题.

一点也不能长?

我们知道 `int3` 指令不带任何操作数, 操作码为1个字节, 因此指令的长度是1个字节. 这是必须的吗? 假设有一种x86体系结构的变种my-x86, 除了 `int3` 指令的长度变成了2个字节之外, 其余指令和x86相同. 在my-x86中, 文章中的断点机制还可以正常工作吗? 为什么?

"随心所欲"的断点

如果把断点设置在指令的非首字节(中间或末尾), 会发生什么? 你可以在GDB中尝试一下, 然后思考并解释其中的缘由.

NEMU的前世今生

你已经对NEMU的工作方式有所了解了. 事实上在NEMU诞生之前, NEMU曾经有一段时间并不叫NEMU, 而是叫NDB(NJU Debugger), 后来由于某种原因才改名为NEMU. 如果你想知道这一段史前的秘密, 你首先需要了解这样一个问题: 模拟器(Emulator)和调试器(Debugger)有什么不同? 更具体地, 和NEMU相比, GDB到底是如何调试程序的?

i386手册

在以后的PA中, 你需要反复阅读i386手册. 鉴于有同学片面地认为"看手册"就是"把手册全看一遍", 因而觉得"不可能在短时间内看完", 我们在PA1的最后来聊聊如何科学地看手册.

学会使用目录

了解一本书都有哪些内容的最快方法就是查看目录, 尤其是当你第一次看一本新书的时候. 查看目录之后并不代表你知道它们具体在说什么, 但你会对这些内容有一个初步的印象, 提到某一个概念的时候, 你可以大概知道这个概念会在手册中的哪些章节出现. 这对查阅手册来说是极其重要的, 因为我们每次查阅手册的时候总是关注某一个问题, 如果每次都需要把手册从头到尾都看一遍才能确定关注的问题在哪里, 效率是十分低下的. 事实上也没有人会这么做, 阅读目录的重要性可见一斑. 纸上得来终觉浅, 还是来动手体会一下吧!

尝试通过目录定位关注的问题

假设你现在需要了解一个叫`selector`的概念, 请通过i386手册的目录确定你需要阅读手册中的哪些地方.

怎么样, 是不是很简单? 虽然你还是不明白`selector`是什么, 但你已经知道你需要阅读哪些地方了, 要弄明白`selector`, 那也是指日可待的事情了.

逐步细化搜索范围

有时候你关注的问题不一定直接能在目录里面找到, 例如"CR0寄存器的PG位的含义是什么". 这种细节的问题一般都是出现在正文中, 而不会直接出现在目录中, 因此你就不能直接通过目录来定位相应的内容了. 根据你是否第一次接触CR0, 查阅这个问题会有不同的方法:

- 如果你已经知道CR0是个control register, 你可以直接在目录里面查看"control register"所在的章节, 然后在这些章节的正文中寻找"CR0".
- 如果你对CR0一无所知, 你可以使用阅读器中的搜索功能, 搜索"CR0", 还是可以很快地找到"CR0"的相关内容. 不过最好的方法是首先使用搜索引擎, 你可以马上知道"CR0是个control register", 然后就可以像第一种方法那样查阅手册了.

不过有时候, 你会发现一个概念在手册中的多个地方都有提到. 这时你需要明确你要关心概念的哪个方面, 通常一个概念的某个方面只会在手册中的一个地方进行详细的介绍. 你需要在这多个地方中进行进一步的筛选, 但至少你已经过滤掉很多与这个概念无关的章节了. 筛选也是有策略的, 你不需要把多个地方的所有内容全部阅读一遍才能进行筛选, 小标题, 每段的第一句

话, 图表的注解, 这些都可以帮助你很快地了解这一部分的内容大概在讲什么. 这不就是高中英语考试中的快速阅读吗? 对的, 就是这样. 如果你觉得目前还缺乏这方面的能力, 现在锻炼的好机会来了.

搜索和筛选信息是一个trail and error的过程, 没有什么方法能够指导你在第一遍搜索就能成功, 但还是有经验可言的. 搜索失败的时候, 你应该尝试使用不同的关键字重新搜索. 至于怎么变换关键字, 就要看你对问题核心的理解了, 换句话说, 怎么问才算是切中要害. 这不就是高中语文强调的表达能力吗? 对的, 就是这样.

事实上, 你只需要具备一些基本的交际能力, 就能学会查阅资料, 和资料的内容没有关系, 来一本"民法大全", "XX手机使用说明书", "YY公司人员管理记录", 照样是这么查阅. "查阅资料"是一种与领域无关的基本能力, 无论身处哪一个行业都需要具备, 如果你不想以后工作的时候被查阅资料的能力影响了自己的前途, 从现在开始就努力锻炼吧!

必答题

你需要在实验报告中回答下列问题:

- 理解基础设施 我们通过一些简单的计算来体会简易调试器的作用. 首先作以下假设:
  - 假设你需要编译500次NEMU才能完成PA.
  - 假设这500次编译当中, 有90%的次数是用于调试.
  - 假设你没有实现简易调试器, 只能通过GDB对运行在NEMU上的客户程序进行调试. 在每一次调试中, 由于GDB不能直接观测客户程序, 你需要花费30秒的时间来从GDB中获取并分析一个信息.
  - 假设你需要获取并分析20个信息才能排除一个bug.

  那么这个学期下来, 你将会在调试上花费多少时间?

  由于简易调试器可以直接观测客户程序, 假设通过简易调试器只需要花费10秒的时间从中获取并分析相同的信息. 那么这个学期下来, 简易调试器可以帮助你节省多少调试的时间?

  事实上, 这些数字也许还是有点乐观, 例如就算使用GDB来直接调客户程序, 这些数字假设你能通过10分钟的时间排除一个bug. 如果实际上你需要在调试过程中获取并分析更多的信息, 简易调试器这一基础设施能带来的好处就更大.
- 查阅i386手册 理解了科学查阅手册的方法之后, 请你尝试在i386手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - EFLAGS寄存器中的CF位是什么意思?
  - ModR/M字节是什么?
  - mov指令的具体格式是怎么样的?
- shell命令 完成PA1的内容之后, `nemu/` 目录下的所有.c和.h文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 目前 `2017` 分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到"过

去"?) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, `nemu/` 目录下的所有 `.c` 和 `.h` 文件总共有多少行代码?

- 使用man 打开工程目录下的 `Makefile` 文件, 你会在 `CFLAGS` 变量中看到gcc的一些编译选项. 请解释gcc中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror` ?

温馨提示

PA1到此结束. 请你编写好实验报告(不要忘记在实验报告中回答必答题), 然后把命名为 `学号.pdf` 的实验报告文件放置在工程目录下, 执行 `make submit` 对工程进行打包, 最后将压缩包提交到指定网站.

PA2 - 简单复杂的机器: 冯诺依曼计算机系统

世界诞生的故事 - 第二章

先驱已经创造了图灵机. 但区区几个数字电路模块搭成的如此简单的机器, 又能做些什么呢? 先驱说, 一切无限的可能, 都蕴含其中.

在进行本PA前, 请在工程目录下执行以下命令进行分支整理, 否则将影响你的成绩:

```
git commit --allow-empty -am "before starting pa2"
git checkout master
git merge pa1
git checkout -b pa2
```

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 40小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 在NEMU中运行第一个C程序 `dummy` - 2017/10/15 23:59:59
- 阶段2: 实现更多的指令, 在NEMU中运行所有 `cputests` - 2017/10/29 23:59:59
- 最后阶段: 运行打字小游戏, 提交完整的实验报告 - 2017/11/05 23:59:59

提交说明: 见这里

不停计算的机器

在PA1中, 我们已经见识到最简单的计算机TRM的工作方式:

```
while (1) {
  从EIP指示的存储器位置取出指令;
  执行指令;
  更新EIP;
}
```

接下来我们就来谈谈这个过程, 也就是, CPU究竟是怎么执行一条指令的. 对于大部分指令来说, 执行它们都可以抽象成取指-译码-执行的指令周期. 为了使描述更加清晰, 我们借助指令周期中的一些概念来说明指令执行的过程.

取指(instruction fetch, IF)

取指令要做的事情自然就是将 `%eip` 指向的指令从内存读入到CPU中, 其实就是一次内存的访问.

译码(instruction decode, ID)

在取指阶段, 计算机拿到了将要执行的指令. 让我们也来目睹一下指令的风采, 睁大眼睛一看, 竟然是个0和1组成的比特串!

```
10111001 00110100 00010010 00000000 00000000
```

这究竟是什么鬼... 不过想想, 计算机也只是个巨大的数字电路, 它也只能理解0和1了. 但是, 这样的计算机又是如何理解这让人一头雾水的比特串的呢?

让我们先来回想一下指令是做什么的. 我们知道CPU是用来处理数据的, 指令则是用来指示CPU具体对什么数据进行什么样的处理. 也就是说, 我们只要让CPU从上面那串神秘的比特串中解读出处理的对象和处理的操作, CPU就知道我们想让它做什么了. 所以相应地, CPU需要从指令中解读出"操作数"和"操作码"两部分信息.

于是, 为了让计算机明白指令的含义, 先驱想到了一个办法, 那就是你在数字电路课上学习过的查找表! CPU拿到一条指令之后, 可以通过查表的方式得知这条指令的操作数和操作码. 这个过程叫译码.

当然, 译码逻辑实际上也并非只有一张查找表那么简单, 还需要根据不同的指令通过多路选择器选择不同的操作数. 回想一下, 计算机现在已经有存储器和寄存器了, 它们都可以存放操作数, 指令中也可以存放立即数. 也可能还有二次译码的处理... 不过无论再怎么复杂, 我们只需要

知道, 这个过程终究也只是一些数字电路的事情, 毕竟所有需要的信息都在指令里面了, 没什么神秘的操作.

执行(execute, EX)

经过译码之后, CPU就知道当前指令具体要做什么了, 执行阶段就是真正完成指令的工作. 现在TRM只有加法器这一个执行部件, 必要的时候, 只需要往加法器输入两个源操作数, 就能得到执行的结果了. 之后还要把结果写回到目的操作数中, 可能是寄存器, 也可能是内存.

更新 `%eip`

执行完一条指令之后, CPU就要执行下一条指令. 在这之前, CPU 需要更新 `%eip` 的值, 让 `%eip` 加上刚才执行完的指令的长度, 即可指向下一条指令的位置.

于是, 计算机不断地重复上述四个步骤, 不断地执行指令, 直到永远.

也许你会疑惑, 这个只能做加法的TRM, 究竟还能做些什么呢? 对于采用补码表示的计算机, 能做加法自然就能做减法. 如果再添加一条条件跳转指令 `jne r, addr`：当寄存器 `r` 不为 `0` 时, `%eip` 跳转到 `addr` 处, TRM就大不一样了. 科学家证明了, 只要有 `inc`, `dec`, `jne` 这三条指令, 就可以实现"所有"的算法! (这里的"所有"是指可计算理论中的"所有可计算的算法") 也就是说, 现代计算机可以解决的纯粹的计算问题, 这个只有三条指令的TRM也可以解决. 例如通过 `jne` 和 `dec` 的组合可以实现循环, 循环执行 `inc` 可以实现任意数的加法, 循环执行加法可以实现乘法... 甚至科学家还证明了, 仅仅通过这三条指令, 就可以编写一个和NEMU功能等价的程序! 这下可不得了了, 没想到这个弱不禁风的TRM竟然深藏着擎天撼地的威力! 不过, 虽然这个只有三条指令的TRM可以解决所有可计算的问题, 但却低效得让人无法忍受. 为此, 先驱决定往TRM中加入更多高效的指令.

RTFM

我们在上一小节中已经在概念上介绍了一条指令具体如何执行, 其中有的概念甚至显而易见得难以展开. 不过x86这一庞然大物背负着太多历史的包袱, 但当我们决定往TRM中添加各种高效的x86指令时, 也同时意味着我们无法回避这些繁琐的细节.

首先你需要了解指令确切的行为, 为此, 你需要阅读i386手册中指令集相关的章节. 这里有一个简单的阅读教程.

> **RISC - 与CISC平行的另一个世界**
>
> 你是否觉得x86指令集的格式特别复杂? 这其实是CISC的一个特性, 不惜使用复杂的指令格式, 牺牲硬件的开发成本, 也要使得一条指令可以多做事情, 从而提高代码的密度, 减小程序的大小. 随着时代的发展, 架构师发现CISC中复杂的控制逻辑不利于提高处理器的性能, 于是RISC应运而生. RISC的宗旨就是简单, 指令少, 指令长度固定, 指令格式统一, 这和KISS法则有异曲同工之妙. 这里有一篇对比RISC和CISC的小短文.
>
> 另外值得推荐的是这篇文章, 里面讲述了一个从RISC世界诞生, 到与CISC世界融为一体的故事, 体会一下RISC的诞生对计算机体系结构发展的里程碑意义.

RTFSC(2)

下面我们来介绍NEMU的框架代码是如何执行指令的.

数据结构

首先先对这个过程中的两个重要的数据结构进行说明.

- `nemu/src/cpu/exec/exec.c` 中的 `opcode_table` 数组. 这就是我们之前提到的译码查找表了, 这一张表通过操作码opcode来索引, 每一个opcode对应相应指令的译码函数, 执行函数, 以及操作数宽度.
- `nemu/src/cpu/decode/decode.c` 中的 `decoding` 结构. 它用于记录一些全局译码信息供后续使用, 包括操作数的类型, 宽度, 值等信息. 其中的 `src` 成员, `src2` 成员和 `dest` 成员分别代表两个源操作数和一个目的操作数. `nemu/include/cpu/decode.h` 中定义了三个宏 `id_src` , `id_src2` 和 `id_dest` , 用于方便地访问它们.

执行流程

然后对 `exec_wrapper()` 的执行过程进行简单介绍.

- 首先将当前的 `%eip` 保存到全局译码信息 `decoding` 的成员 `seq_eip` 中, 然后将其地址被作为参数送进 `exec_real()` 函数中. `seq` 代表顺序的意思, 当代码从 `exec_real()` 返回时, `decoding.seq_eip` 将会指向下一条指令的地址. `exec_real()` 函数通过宏 `make_EHelper` 来定义:

  ```
  #define make_EHelper(name) void concat(exec_, name) (vaddr_t *eip)
  ```

  其含义是"定义一个执行阶段相关的helper函数", 这些函数都带有一个参数 `eip` . NEMU通过不同的helper函数来模拟不同的步骤.

在 `exec_real()` 中:

- 首先通过 `instr_fetch()` 函数(在 `nemu/include/cpu/exec.h` 中定义)进行取指, 得到指令的第一个字节, 将其解释成 `opcode` 并记录在全局译码信息 `decoding` 中.
- 根据 `opcode` 查阅译码查找表, 得到操作数的宽度信息, 并通过调用 `set_width()` 函数将其记录在全局译码信息 `decoding` 中.
- 调用 `idex()` 对指令进行进一步的译码和执行

`idex()` 函数会调用译码查找表中的相应的译码函数进行操作数的译码. 译码函数统一通过宏 `make_DHelper` 来定义(在 `nemu/src/cpu/decode/decode.c` 中):

```
#define make_DHelper(name) void concat(decode_, name) (vaddr_t *eip)
```

它们的名字主要采用i386手册附录A中的操作数表示记号, 例如 `I2r` 表示将立即数移入寄存器, 其中 `I` 表示立即数, `2` 表示英文 `to` , `r` 表示通用寄存器, 更多的记号请参考i386手册. 译码函数会把指令中的操作数信息分别记录在全局译码信息 `decoding` 中

这些译码函数会进一步分解成各种不同操作数的译码的组合, 以实现操作数译码的解耦. 操作数译码函数统一通过宏 `make_DopHelper` 来定义 (在 `nemu/src/cpu/decode/decode.c` 中, `decode_op_rm()` 除外):

```
#define make_DopHelper(name) void concat(decode_op_, name) (vaddr_t *eip, Operand *op,
 bool load_val)
```

它们的名字主要采用i386手册附录A中的操作数表示记号. 操作数译码函数会把操作数的信息记录在结构体 `op` 中, 如果操作数在指令中, 就会通过 `instr_fetch()` 将它们从 `eip` 所指向的内存位置取出. 为了使操作数译码函数更易于复用, 函数中的 `load_val` 参数会控制 是否需要将该操作数读出到全局译码信息 `decoding` 供后续使用. 例如如果一个内存操作数是源操作数, 就需要将这个操作数从内存中读出来供后续执行阶段来使用; 如果它仅仅是一个目的操作数, 就不需要从内存读出它的值了, 因为执行这条指令并不需要这个值, 而是将新数据写入相应的内存位置.

`idex()` 函数中的译码过程结束之后, 会调用译码查找表中的相应的执行函数来进行真正的执行操作. 执行函数统一通过宏 `make_EHelper` 来定义, 它们的名字是指令操作本身. 执行函数通过RTL来描述指令真正的执行功能(RTL将在下文介绍). 其中 `operand_write()` 函数 (在 `nemu/src/cpu/decode/decode.c` 中定义) 会根据第一个参数中记录的类型的不同进行相应的写操作, 包括写寄存器和写内存.

从 `idex()` 返回后, `exec_real()` 最后会通过 `update_eip()` 对 `%eip` 进行更新.

上文已经把一条指令在NEMU中执行的流程进行了大概的介绍. 如果觉得上文的内容不易理解, 可以结合这个例子来RTFSC. 但这个例子中会描述较多细节, 阅读的时候需要一定的耐心.

立即数背后的故事

在 `decode_op_I()` 函数中通过 `instr_fetch()` 函数获得指令中的立即数. 别看这里就这么一行代码, 其实背后隐藏着针对字节序的慎重考虑. 我们知道x86是小端机, 当你使用高级语言或者汇编语言写了一个32位常数 `0x1234` 的时候, 在生成的二进制代码中, 这个常数对应的字节序列如下(假设这个常数在内存中的起始地址是x):

```
x   x+1  x+2  x+3
+----+----+----+----+
| 34 | 12 | 00 | 00 |
+----+----+----+----+
```

而大多数PC机都是小端架构(我们相信没有同学会使用IBM大型机来做PA), 当NEMU运行的时候,

```
op_src->imm = instr_fetch(eip, 4);
```

这行代码会将 `34 12 00 00` 这个字节序列原封不动地从内存读入 `imm` 变量中, 主机的CPU会按照小端方式来解释这一字节序列, 于是会得到 `0x1234`, 符合我们的预期结果.

Motorola 68k系列的处理器都是大端架构的. 现在问题来了, 考虑以下两种情况:

- 假设我们需要将NEMU运行在Motorola 68k的机器上(把NEMU的源代码编译成Motorola 68k的机器码)
- 假设我们需要编写一个新的模拟器NEMU-Motorola-68k, 模拟器本身运行在x86架构中, 但它模拟的是Motorola 68k程序的执行

在这两种情况下, 你需要注意些什么问题? 为什么会产生这些问题? 怎么解决它们?

事实上不仅仅是立即数的访问, 长度大于1字节的内存访问都需要考虑类似的问题. 我们在这里把问题统一抛出来, 以后就不再单独讨论了.

结构化程序设计

细心的你会发现以下规律:

- 对于同一条指令的不同形式, 它们的执行阶段是相同的. 例如 `add_I2E` 和 `add_E2G` 等, 它们的执行阶段都是把两个操作数相加, 把结果存入目的操作数.
- 对于不同指令的同一种形式, 它们的译码阶段是相同的. 例如 `add_I2E` 和 `sub_I2E` 等, 它们的译码阶段都是识别出一个立即数和一个 `E` 操作数.
- 对于同一条指令同一种形式的不同操作数宽度, 它们的译码阶段和执行阶段都是非常类似的. 例如 `add_I2E_b` , `add_I2E_w` 和 `add_I2E_l` , 它们都是识别出一个立即数和一个 `E` 操作数, 然后把相加的结果存入 `E` 操作数.

这意味着, 如果独立实现每条指令不同形式不同操作数宽度的helper函数, 将会引入大量重复的代码. 需要修改的时候, 相关的所有helper函数都要分别修改, 遗漏了某一处就会造成bug, 工程维护的难度急速上升. 一种好的做法是把译码, 执行和操作数宽度的相关代码分离开来, 实现解耦, 也就是在程序设计课上提到的结构化程序设计.

在框架代码中, 实现译码和执行之间的解耦的是 `idex()` 函数, 它依次调用 `opcode_table` 表项中的译码和执行的helper函数, 这样我们就可以分别编写译码和执行的helper函数了. 实现操作数宽度和译码, 执行这两者之间的解耦的是 `id_src` , `id_src2` 和 `id_dest` 中的 `width` 成员, 它们记录了操作数宽度, 译码和执行的过程中会根据它们进行不同的操作, 通过同一份译码函数和执行函数实现不同操作数宽度的功能.

为了易于使用, 框架代码中使用了一些宏, 我们在这里把相关的宏整理出来, 供大家参考.

宏	含义
`nemu/include/macro.h`	
`str(x)`	字符串 `"x"`
`concat(x, y)`	token `xy`
`nemu/include/cpu/reg.h`	
`reg_l(index)`	编码为 `index` 的32位GPR
`reg_w(index)`	编码为 `index` 的16位GPR
`reg_b(index)`	编码为 `index` 的8位GPR
`nemu/include/cpu/decode.h`	
`id_src`	全局变量 `decoding` 中源操作数成员的地址
`id_src2`	全局变量 `decoding` 中2号源操作数成员的地址
`id_dest`	全局变量 `decoding` 中目的操作数成员的地址
`make_Dhelper(name)`	名为 `decode_name` 的译码函数的原型说明
`nemu/src/cpu/decode.c`	
`make_Dophelper(name)`	名为 `decode_op_name` 的操作数译码函数的原型说明
`nemu/include/cpu/exec.h`	
`make_Ehelper(name)`	名为 `exec_name` 的执行函数的原型说明
`print_asm(...)`	将反汇编结果的字符串打印到缓冲区 `decoding.assembly` 中
`suffix_char(width)`	操作数宽度 `width` 对应的后缀字符
`print_asm_template1(instr)`	打印单目操作数指令 `instr` 的反汇编结果
`print_asm_template2(instr)`	打印双目操作数指令 `instr` 的反汇编结果
`print_asm_template3(instr)`	打印三目操作数指令 `instr` 的反汇编结果

强大的宏

如果你知道C++的"模板"功能, 你可能会建议使用它, 但事实上在这里做不到. 我们知道宏是在编译预处理阶段进行处理的, 这意味着宏的功能不受编译阶段的约束(包括词法分析, 语法分析, 语义分析); 而C++的模板是在编译阶段进行处理的, 这说明它会受到编译阶段的限制. 理论上来说, 必定有一些事情是宏能做到, 但C++模板做不到. 一个例子就是框架代码中的拼接宏 `concat()`, 它可以把两个token连接成一个新的token; 而在C++模板进行处理的时候, 词法分析阶段已经结束了, 因而不可能通过C++模板生成新的token.

计算机世界处处都是tradeoff, 有好处自然需要付出代价. 由于处理宏的时候不会进行语法检查, 因为宏而造成的错误很有可能不会马上暴露. 例如以下代码:

```
#define N 10;
int a[N];
```

在编译的时候, 编译器会提示代码的第2行有语法错误. 但如果你光看第2行代码, 你很难发现错误, 甚至会怀疑编译器有bug. 那宏到底要不要用呢? 一种客观的观点是, 在你可以控制的范围中使用. 这就像goto语句一样, 当你希望在多重循环中从最内层循环直接跳出所有循环, goto是最方便的做法. 但如果代码中到处都是goto, 已经严重影响到代码段的可读性了, 这种情况当然是不可取的.

用RTL表示指令行为

NEMU使用RTL(寄存器传输语言)来描述x86指令的行为. 这样做的好处是可以提高代码的复用率, 使得指令模拟的实现更加规整. 同时RTL也可以作为一种IR(中间表示)语言, 将来可以很方便地引入即时编译技术对NEMU进行优化, 即使你在PA中不一定有机会感受到这一好处.

下面我们对NEMU中使用的RTL进行一些说明, 首先是RTL寄存器的定义. RTL寄存器是RTL指令专门使用的寄存器. 在NEMU中, RTL寄存器统一使用 `rtlreg_t` 来定义, 而 `rtlreg_t` (在 `nemu/include/common.h` 中定义)其实只是一个 `uint32_t` 类型:

```
typedef uint32_t rtlreg_t;
```

在NEMU中, RTL寄存器只有以下这些

- x86的八个通用寄存器(在 `nemu/include/cpu/reg.h` 中定义)
- `id_src` , `id_src2` 和 `id_dest` 中的访存地址 `addr` 和操作数内容 `val` (在 `nemu/include/cpu/decode.h` 中定义). 从概念上看, 它们分别与MAR和 MDR有异曲同工之妙
- 临时寄存器 `t0~t3` (在 `nemu/src/cpu/decode/decode.c` 中定义)
- 0寄存器 `tzero` (在 `nemu/src/cpu/decode/decode.c` 中定义), 它只能读出 `0` , 不能写入

有了RTL寄存器, 我们就可以定义RTL指令对它们进行的操作了. 在NEMU中, RTL指令有两种(在 `nemu/include/cpu/rtl.h` 中定义). 一种是RTL基本指令, 它们的特点是在即时编译技术里面可以只使用一条机器指令来实现相应的功能, 同时也不需要使用临时寄存器, 可以看做是最基本的x86指令中的最基本的操作. RTL基本指令包括:

- 立即数读入 `rtl_li`
- 算术运算和逻辑运算, 包括寄存器-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)` 和立即数-寄存器类型 `rtl_(add|sub|and|or|xor|shl|shr|sar|slt|sltu)i`
- 内存的访存 `rtl_lm` 和 `rtl_sm`
- 通用寄存器的访问 `rtl_lr_(b|w|l)` 和 `rtl_sr_(b|w|l)`

第二种RTL指令是RTL伪指令, 它们是通过RTL基本指令或者已经实现的RTL伪指令来实现的, 包括:

- 带宽度的通用寄存器访问 `rtl_lr` 和 `rtl_sr`
- EFLAGS标志位的读写 `rtl_set_(CF|OF|ZF|SF|IF)` 和 `rtl_get_(CF|OF|ZF|SF|IF)`
- 其它常用功能, 如数据移动 `rtl_mv` , 符号扩展 `rtl_sext` 等

其中大部分RTL伪指令还没有实现, 必要的时候你需要实现它们. 有了这些RTL指令之后, 我们就可以方便地通过若干条RTL指令来实现每一条x86指令的行为了.

实现新指令

对译码, 执行和操作数宽度的解耦实现以及RTL的引入对NEMU中实现一条新的x86指令提供了很大的便利, 为了实现一条新指令, 你只需要

1. 在 `opcode_table` 中填写正确的译码函数, 执行函数以及操作数宽度
2. 用RTL实现正确的执行函数, 需要注意使用RTL伪指令时不要把临时变量中有意义的值覆盖了

框架代码把绝大部分译码函数和执行函数都定义好了, 你可以很方便地使用它们.

如果你读过上文的扩展阅读材料中关于RISC与CISC融为一体的故事, 你也许会记得CISC风格的x86指令最终被分解成RISC风格的微指令在计算机中运行, 才让x86在这场扩日持久的性能大战中得以存活下来的故事. 如今NEMU在经历了第二次重构之后, 也终于引入了RISC风格的RTL来实现x86指令, 这也许是冥冥之中的安排吧.

运行第一个C程序

说了这么多, 现在到了动手实践的时候了. 你在PA2的第一个任务, 就是实现若干条指令, 使得第一个简单的C程序可以在NEMU中运行起来. 这个简单的C程序的代码是 `nexus-am/tests/cputest/tests/dummy.c` , 它什么都不做就直接返回了. 在 `nexus-am/tests/cputest` 目录下键入

```
make ARCH=x86-nemu ALL=dummy run
```

编译 `dummy` 程序, 并启动NEMU运行它. 事实上, 并不是每一个程序都可以在NEMU中运行, `nexus-am/` 子项目专门用于编译出能在NEMU中运行的程序, 我们在下一小节中会再来介绍它.

在NEMU中运行 `dummy` 程序, 你会发现NEMU输出以下信息:

```
invalid opcode(eip = 0x0010000a): e8 01 00 00 00 90 55 89 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010000a is not implemented.
2. Something is implemented incorrectly.
Find this eip value(0x0010000a) in the disassembling result to distinguish which case
it is.

If it is the first case, see
 _ ____   ___    __    __  __                          _
(_)___ \ / _ \  / /   |  \/  |                        | |
 _  __) | (_) |/ /_   | \  / | __ _ _ __   _   _  __ _| |
| ||__ < > _ <| '_ \  | |\/| |/ _  | '_ \| | | |/ _  | |
| |___) | (_) | (_) | | |  | | (_| | | | | |_| | (_| | |
|_|____/ \___/ \___/  |_|  |_|\__,_|_| |_|\__,_|\__,_|_|

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!
```

这是因为你还没有实现以 `0xe8` 为首字节的指令, 因此, 你需要开始在NEMU中添加指令了.

要实现哪些指令才能让 `dummy` 在NEMU中运行起来呢? 答案就在其反汇编结果(`nexus-am/tests/cputest/build/dummy-x86-nemu.txt`)中. 查看反汇编结果, 你发现只需要添加 `call` , `push` , `sub` , `xor` , `pop` , `ret` 六条指令就可以了. 每一条指令还有不同的形式, 根据KISS法则, 你可以先实现只在 `dummy` 中出现的指令形式, 通过指令的 `opcode` 可以确定具体的形式.

这里要再次强调, 你务必通过i386手册来查阅指令的功能, 不能想当然. 手册中给出了指令功能的完整描述(包括做什么事, 怎么做的, 有什么影响), 一定要仔细阅读其中的每一个单词, 对指令功能理解错误和遗漏都会给以后的调试带来巨大的麻烦.

- `call` : `call` 指令有很多形式, 不过在PA中只会用到其中的几种, 现在只需要实现 `CALL rel32` 的形式就可以了. `%eip` 的跳转可以通过将 `decoding.is_jmp` 设为 `1` , 并将 `decoding.jmp_eip` 设为跳转目标地址来实现, 这时在 `update_eip()` 函数中会把跳转目标地址作为新的 `%eip` , 而不是顺序意义下的下一条指令的地址
- `push` , `pop` : 现在只需要实现 `PUSH r32` 和 `POP r32` 的形式就可以了, 它们可以很容易地通过 `rtl_push` 和 `rtl_pop` 来实现
- `sub` : 在实现 `sub` 指令之前, 你首先需实现EFLAGS寄存器. 你只需要在寄存器结构体中添加EFLAGS寄存器即可. EFLAGS是一个32位寄存器, 它的结构如下:

```
31                  23                  15              7             0
+-------------------+-------------------+-------+-+-+-+-+-+-+-------+-+-+
|                                         |O| |I| |S|Z|       | |C|
|                     X                   | |X| |X| | |   X   |1| |
|                                         |F| |F| |F|F|       | |F|
+-------------------+-------------------+-------+-+-+-+-+-+-+-------+-+-+
```

在NEMU中, 我们只会用到EFLAGS中以下的5个位: `CF` , `ZF` , `SF` , `IF` , `OF` , 标记成 `X` 的位不必关心, 它们的功能可暂不实现. 关于EFLAGS中每一位的含义, 请查阅i386手册. 添加EFLAGS寄存器需要用到结构体的位域(bit field)功能, 如果你从未听说过位域, 请查阅相关资料. 关于EFLGAS的初值, 我们遵循i386手册中提到的约定, 你需要在i386手册的第10章中找到这一初值, 然后在 `restart()` 函数中对EFLAGS寄存器进行初始化. 实现了EFLAGS寄存器之后, 再实现相关的RTL指令, 之后你就可以通过这些RTL指令来实现 `sub` 指令了

- `xor` , `ret` : RTFM吧

运行第一个客户程序

在NEMU中通过RTL指令实现上文提到的指令, 具体细节请务必参考i386手册. 实现成功后, 在NEMU中运行客户程序 `dummy` , 你将会看到 `HIT GOOD TRAP` 的信息.

温馨提示

PA2阶段1到此结束.

程序, 运行时环境与AM

现代指令系统

我们已经成功在TRM上运行 `dummy` 程序了, 然而这个程序什么都没做就结束了, 一点也不过瘾啊. 为了让NEMU支持大部分程序的运行, 你还需要实现更多的指令:

- Data Movement Instructions: ~~`mov`~~, `push`, `pop`, `leave`, `cltd` (在i386手册中为 `cdq`), ~~`movsx`~~, ~~`movzx`~~
- Binary Arithmetic Instructions: `add`, `inc`, `sub`, `dec`, `cmp`, `neg`, ~~`adc`~~, ~~`sbb`~~, ~~`mul`~~, ~~`imul`~~, ~~`div`~~, ~~`idiv`~~
- Logical Instructions: `not`, `and`, `or`, `xor`, `sal(shl)`, `shr`, `sar`, ~~`setcc`~~, `test`
- Control Transfer Instructions: ~~`jmp`~~, ~~`jcc`~~, `call`, `ret`
- Miscellaneous Instructions: ~~`lea`~~, ~~`nop`~~

框架代码已经实现了上述删除线标记的指令, 但并没有填写 `opcode_table`. 此外, 某些需要更新EFLAGS的指令并没有完全实现好(框架代码中已经插入了 `TODO()` 作为提示), 你还需要编写相应的功能.

运行时环境与AM

但并不是有了足够的指令就能运行更多的程序. 我们之前提到"并不是每一个程序都可以在NEMU中运行", 现在我们来解释一下背后的缘由.

从直觉上来看, 让TRM来支撑一个功能齐全的操作系统的运行还是比较勉强的. 这给我们的感觉就是, 计算机也有一定的"功能强弱"之分, 计算机越"强大", 就能跑越复杂的程序. 换句话说, 程序的运行其实是对计算机的功能有需求的. 在你运行Hello World程序时, 你敲入一条命令(或者点击一下鼠标), 程序就成功运行了, 但这背后其实隐藏着操作系统开发者和库函数开发者的无数汗水. 一个事实是, 应用程序的运行都需要运行时环境的支持, 包括加载, 销毁程序, 以及提供程序运行时的各种动态链接库(你经常使用的库函数就是运行时环境提供的)等. 为了让客户程序在NEMU中运行, 现在轮到你来提供相应的运行时环境的支持了. 不用担心, 由于NEMU目前的功能并不完善, 我们必定无法向用户程序提供GNU/Linux般的运行时环境.

我们先来讨论一下程序执行究竟需要些什么.

1. 程序需要有地方存放代码和数据, 于是需要内存
2. 程序需要执行, 于是需要CPU以及指令集
3. 对于需要运行结束的程序, 需要有一种结束运行的方法

事实上, 可以在TRM上运行的程序都对计算机有类似的需求. 我们把这些计算机相关的需求抽象成统一的API提供给程序, 这样程序就不需要关心计算机硬件相关的细节了. 这正是AM(Abstract machine)项目的意义所在.

什么是AM?

你或许会觉得NEMU与AM的关系有点模糊不清, 让我们还是来看ATM机的例子.

说起ATM机, 你脑海里一定会想起一个可以存款, 取款, 查询余额, 转账的机器. 我们不妨把你脑海里的这个机器的模型称为抽象ATM机. 从用户的角度来说, 用户对ATM机的功能是有期望的: 要能存款, 取款, 查询余额, 转账.

从银行的角度来说, 不同银行的ATM机千差万别: 存款的加密方式, 交易时使用的自定义通信协议, 余额在银行系统里面的表示和组织方式... 不同银行的ATM机之间存在这么多细节上的差异, 怎么样才能让用户方便地使用ATM机呢? 那就是, 为不同银行的ATM机分别实现上文提到的抽象ATM机的功能: 只要ATM机实现了存款, 取款和查询余额的这组统一的功能, 和用户对抽象ATM机的认识匹配上, 用户就可以方便地使用这台ATM机, 而不必关心ATM机的上述细节.

在NEMU和AM的关系中, 程序就像是用户, AM就像是抽象ATM机, 我们实现NEMU这个计算机就像是造一台新的(虚拟的)ATM机, 也就像我们在PA1中提到的, 写一个支付宝APP. 同样的道理, 程序对计算机的功能是有期望的: 要能计算, 输入输出... 这些功能的期望组成了一台抽象计算机AM, 它刻画了一台真实计算机应该具备的功能. 但不同计算机的硬件配置各不相同, ISA也千差万别, 怎么样才能让程序方便地运行呢? 那就是, 为不同的计算机分别实现AM的功能: 只要计算机实现了AM定义的一组统一的API, 就能和程序对计算机的功能期望匹配上, 程序就可以方便地在计算机上运行, 而不必关心计算机的底层细节.

有兴趣折腾的同学还可以来理解一下真机, NEMU和AM这三者的关系. 我们会发现, 无论是真实的ATM机还是支付宝APP, 都符合我们对的抽象ATM机的认知: 它们都能存款, 取款, 查询余额, 转账. 也正因为如此, 支付宝APP刚推出的时候, 我们才能很容易上手: 虽然支付宝APP是个虚拟的ATM机, 但我们还是可以很容易根据我们对抽象ATM机的认知来使用它.

回到NEMU的例子中来, 我们还是用ATM机的例子来比喻: 真机就像是一台真实的ATM机, NEMU这个虚拟机就像是一个支付宝APP, AM还是我们概念上的抽象ATM机. 只要一台机器实现了AM的功能(能计算, 能输出输入...), 程序都可以在上面运行, 不必关心这台机器是真实的, 还是用程序虚拟出来的.

用一句话来总结这三者的关系: AM在概念上定义了一台抽象计算机, 它从运行程序的视角刻画了一台计算机应该具备的功能, 而真机和NEMU都是这台抽象计算机的具体实现, 只是真机是通过物理上存在的数字电路来实现, NEMU是通过程序来实现.

如果你对面向对象程序设计有一些初步的了解, 解释起来就更简单了:

> AM是个抽象类, 真机和虚拟机是由AM这个抽象类派生出来的两个子类, 而x86真机和NEMU则分别是这两个子类的实例化.

AM作为一个计算机的抽象模型, 可以将一个现代计算机从逻辑上划分成以下模块

```
AM = TRM + IOE + ASYE + PTE + MPE
```

- TRM(Turing Machine) - 图灵机, 为计算机提供基本的计算能力
- IOE(I/O Extension) - 输入输出扩展, 为计算机提供输出输入的能力
- ASYE(Asynchronous Extension) - 异步处理扩展, 为计算机提供处理中断异常的能力
- PTE(Protection Extension) - 保护扩展, 为计算机提供存储保护的能力
- MPE(Multi-Processor Extension) - 多处理器扩展, 为计算机提供多处理器通信的能力 (MPE超出了ICS课程的范围, 在PA中不会涉及)

不同程序对计算机的功能需求也不完全一样, 例如只进行纯粹计算任务的程序在TRM上就可以运行; 要运行小游戏, 仅仅是TRM就不够了, 因为小游戏还需要和用户进行交互, 因此还需要IOE; 要运行一个现代操作系统, 还要在此基础上加入ASYE和PTE. 我们知道ISA是计算机系统中的软硬件接口, 而从上述AM的模块划分可以看出, AM描述的恰恰就是ISA本身, 它是不同ISA的抽象.

感谢AM项目的诞生, 让NEMU和程序的界线更加泾渭分明, 同时使得PA的流程更加明确:

```
(在NEMU中)实现硬件功能 -> (在AM中)提供软件抽象 -> (在APP层)运行程序
(在NEMU中)实现更强大的硬件功能 -> (在AM中)提供更丰富的软件抽象 -> (在APP层)运行更复杂的程序
```

这个流程其实与PA1中开天辟地的故事遥相呼应: 先驱希望创造一个计算机的世界, 并赋予它执行程序的使命. 亲自搭建NEMU(硬件)和AM(软件)之间的桥梁来支撑程序的运行, 是"理解程序如何在计算机上运行"这一终极目标的不二选择.

RTFSC(3)

我们来简单介绍一下AM项目的代码. 代码中 `nexus-am` 目录下的源文件组织如下(部分目录下的文件并未列出):

```
nexus-am
├── am                              # AM相关
│   ├── am.h
│   ├── arch                        # 不同体系结构-平台的AM实现
│   │   ├── native
│   │   └── x86-nemu                # x86-nemu的AM实现
│   │       ├── img                 # 构建/运行二进制文件/镜像的脚本
│   │       │   ├── boot
│   │       │   │   ├── Makefile
│   │       │   │   └── start.S     # 程序入口
│   │       │   ├── build           # 构建脚本
│   │       │   ├── loader.ld       # 链接脚本
│   │       │   └── run             # 运行脚本
│   │       ├── include
│   │       ├── README.md
│   │       └── src
│   │           ├── asye.c          # ASYE
│   │           ├── ioe.c           # IOE
│   │           ├── pte.c           # PTE
│   │           ├── trap.S
│   │           └── trm.c           # TRM
│   └── Makefile
├── apps                            # 直接运行在AM上的应用
├── libs                            # 可以直接运行在AM上的库
├── Makefile
├── Makefile.app
├── Makefile.check
├── Makefile.compile
├── Makefile.lib
├── README.md
├── SPEC.md                         # AM接口规范说明
└── tests                           # 直接运行在AM上的测试
```

整个AM项目分为三大部分:

- `nexus-am/am` - 不同计算机架构的AM实现, 在PA中我们只需要关注 `nexus-am/am/arch/x86-nemu` 即可
- `nexus-am/tests` 和 `nexus-am/apps` - 一些功能测试和直接运行AM上的应用程序
- `nexus-am/libs` - 一些体系结构无关的, 可以直接运行在AM上的库, 方便应用程序的开发

在让NEMU运行客户程序之前, 我们需要将客户程序的代码编译成可执行文件. 需要说明的是, 我们不能使用gcc的默认选项直接编译, 因为默认选项会根据GNU/Linux的运行时环境将代码编译成运行在GNU/Linux下的可执行文件. 但此时的NEMU并不能为客户程序提供GNU/Linux的运行时环境, 在NEMU中运行上述可执行文件会产生错误, 因此我们不能使用gcc的默认选项来编译用户程序.

解决这个问题的方法是交叉编译, 我们需要在GNU/Linux下根据AM的运行时环境编译出能够在NEMU中运行的可执行文件. 为了不让链接器ld使用默认的方式链接, 我们还需要提供描述AM运行时环境的链接脚本. AM的框架代码已经把相应的配置准备好了:

- gcc将AM实现的源文件编译成目标文件, 然后通过ar将这些目标文件打包成一个归档文件作为一个库, 把不同计算机架构的AM实现通过库的方式提供给程序
- gcc把在AM上运行的应用程序源文件编译成目标文件
- 必要的时候通过gcc和ar把程序依赖的运行库也打包成归档文件
- 执行脚本文件 `nexus-am/am/arch/x86-nemu/img/build` , 在脚本文件中
  - 将程序入口 `nexus-am/am/arch/x86-nemu/img/boot/start.S` 编译成目标文件
  - 最后让ld根据链接脚本 `nexus-am/am/arch/x86-nemu/img/loader.ld` , 将上述目标文件和归档文件链接成可执行文件

根据这一链接脚本的指示, 可执行程序重定位后的节从 `0x100000` 开始, 首先是 `.text` 节, 其中又以 `nexus-am/am/arch/x86-nemu/img/boot/start.o` 中自定义的 `entry` 节开始, 然后接下来是其它目标文件的 `.text` 节. 这样, 可执行程序的 `0x100000` 处总是放置 `nexus-am/am/arch/x86-nemu/img/boot/start.S` 的代码, 而不是其它代码, 保证客户程序总能从 `0x100000` 开始正确执行. 链接脚本也定义了其它节(包括 `.rodata` , `.data` , `.bss`)的链接顺序, 还定义了一些关于位置信息的符号, 包括每个节的末尾, 栈顶位置, 堆区的起始和末尾.

我们对编译得到的可执行文件的行为进行简单的梳理:

1. 第一条指令从 `nexus-am/am/arch/x86-nemu/img/boot/start.S` 开始, 设置好栈顶之后就跳转到 `nexus-am/am/arch/x86-nemu/src/trm.c` 的 `_trm_init()` 函数处执行.
2. 在 `_trm_init()` 中调用 `main()` 函数执行程序的主体功能.
3. 从 `main()` 函数返回后, 调用 `_halt()` 结束运行.

阅读 `nexus-am/am/arch/x86-nemu/src/trm.c` 中的代码, 你会发现只需要实现很少的API就可以支撑起程序在TRM上运行了:

- `_Area _heap` 结构用于指示堆区的起始和末尾
- `void _putc(char ch)` 用于输出一个字符
- `void _halt(int code)` 用于结束程序的运行
- `void _trm_init()` 用于进行TRM相关的初始化工作

这是因为, TRM所需要的指令集和内存已经被编译器考虑进去了: 编译器认为, 硬件需要提供具体的指令集实现和可用的内存, 编译生成的程序里面只需要包含"使用的指令"和"程序的内存映象"这两方面的信息, 程序就可以在硬件上运行了, 所以我们不需要在 `trm.c` 里面提供"使用指令集"和"使用内存"的API. 关于AM定义的API, 可以阅读 `nexus-am/README.md` 和 `nexus-am/SPEC.md` .

堆和栈在哪里?

> 我们知道代码和数据都在可执行文件里面, 但却没有提到堆(heap)和栈(stack). 为什么堆和栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆和栈又是怎么来的? AM的代码是否能给你带来一些启发?

把 `_putc()` 作为TRM的API是一个很有趣的考虑, 我们在不久的将来再讨论它, 目前我们暂不打算运行需要调用 `_putc()` 的程序.

最后来看看 `_halt()`. `_halt()` 里面是一条内联汇编语句, 内联汇编语句允许我们在C代码中嵌入汇编语句. 这条指令和我们常见的汇编指令不一样(例如 `movl $1, %eax`), 它是直接通过指令的编码给出的, 它只有一个字节, 就是 `0xd6` . 如果你在 `nemu/src/cpu/exec/exec.c` 中查看 `opcode_table` , 你会发现, 这条指令正是那条特殊的 `nemu_trap` ! 这其实也说明了为什么要通过编码来给出这条指令, 如果你使用以下方式来给出指令, 汇编器将会报错:

```
asm volatile("nemu_trap" : : "a" (0))
```

因为这条特殊的指令是我们人为添加的, 标准的汇编器并不能识别它. 如果你查看objdump的反汇编结果, 你会看到 `nemu_trap` 指令被标识为 `(bad)` , 原因是类似的: objdump并不能识别我们人为添加的 `nemu_trap` 指令. `"a"(0)` 表示在执行内联汇编语句给出的汇编代码之前, 先将 `0` 读入 `%eax` 寄存器. 这样, 这段汇编代码的功能就和 `nemu/src/cpu/exec/special.c` 中的helper函数 `nemu_trap()` 对应起来了. 此外, `volatile` 是C语言的一个关键字, 如果你想了解关于 `volatile` 的更多信息, 请查阅相关资料.

运行更多的程序

未测试代码永远是错的, 你需要足够多的测试用例来测试你的NEMU. 我们在 `nexus-am/tests/cputest` 目录下准备了一些测试用例. 首先我们让AM项目上的程序默认编译到 `x86-nemu` 的AM中:

```
--- nexus-am/Makefile.check
+++ nexus-am/Makefile.check
@@ -7,2 +7,2 @@
-ARCH ?= native
+ARCH ?= x86-nemu
 ARCH = $(shell ls $(AM_HOME)/am/arch/)
```

然后在 `nexus-am/tests/cputest/` 目录下执行

```
make ALL=xxx run
```

其中 `xxx` 为测试用例的名称(不包含 `.c` 后缀).

上述 `make run` 的命令最终会调用 `nexus-am/am/arch/x86-nemu/img/run` 来启动NEMU. 为了使用GDB来调试NEMU, 你需要修改这一 `run` 脚本的内容:

```
--- nexus-am/am/arch/x86-nemu/img/run
+++ nexus-am/am/arch/x86-nemu/img/run
@@ -3,1 +3,1 @@
-make -C $NEMU_HOME run ARGS="-l `dirname $1`/nemu-log.txt $1.bin"
+make -C $NEMU_HOME gdb ARGS="-l `dirname $1`/nemu-log.txt $1.bin"
```

然后再执行上述 `make run` 的命令即可. 无需GDB调试时, 可将上述 `run` 脚本改回来.

实现更多的指令

你需要实现上文中提到的更多指令, 以通过上述测试用例.

你可以自由选择按照什么顺序来实现指令. 经过PA1的训练之后, 你应该不会实现所有指令之后才进行测试了. 要养成尽早做测试的好习惯, 一般原则都是"实现尽可能少的指令来进行下一次的测试". 你不需要实现所有指令的所有形式, 只需要通过这些测试即可. 如果将来仍然遇到了未实现的指令, 就到时候再实现它们.

需要注意的是, `push imm8` 指令需要对立即数进行符号扩展, 这一点在i386手册中并没有明确说明. 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.

由于部分测试用例需要实现较多指令, 建议按照以下顺序进行测试:

1. 其它
2. string
3. hello-str

基础设施(2)

测试与调试

理解指令的执行过程之后, 添加各种指令更多的是工程实现. 工程实现难免会碰到bug, 实现不正确的时候如何快速进行调试, 其实也属于基础设施的范畴. 思考一下, 译码查找表中有那么多指令, 每一条指令又通过若干RTL指令实现, 如果其中实现有误, 我们该如何发现呢?

直觉上这貌似不是一件容易的事情, 不过让我们来讨论一下其中的缘由. 假设我们不小心把译码查找表中的某一条指令的译码函数填错了, NEMU执行到这一条指令的时候, 就会使用错误的译码函数进行译码, 从而导致执行函数拿到了错误的源操作数, 或者是将正确的结果写入了错误的目的操作数. 这样, NEMU执行这条指令的结果就违反了它原来的语义, 接下来就会导致跟这条指令有依赖关系的其它指令也无法正确地执行. 最终, 我们就会看到客户程序访问内存越界, 陷入死循环, 或者HIT BAD TRAP, 甚至是NEMU触发了段错误.

调试的工具与原理

我们可以从上面的这个例子中抽象出一些软件工程相关的概念:

- Fault: 实现错误的代码, 例如填写了错误的译码函数
- Error: 程序执行时不符合预期的状态, 例如客户程序的指令没有被正确地执行
- Failure: 能直接观测到的错误, 例如HIT BAD TRAP, 段错误等

调试其实就是从观测到的failure一步一步回溯寻找fault的过程, 找到了fault之后, 我们就很快知道应该如何修改错误的代码了. 但从上面的例子也可以看出, 调试之所以不容易, 恰恰是因为:

- fault不一定马上触发error
- 触发了error也不一定马上转变成可观测的failure
- error会像滚雪球一般越积越多, 当我们观测到failure的时候, 其实已经距离fault非常遥远了

理解了这些原因之后, 我们就可以制定相应的策略了:

- 尽可能把fault转变成error. 这其实就是测试做的事情, 所以 `nexus-am/tests/` 目录下提供了各种各样的测试用例. 但并不是有了测试用例就能把所有fault都转变成error了, 因为这取决于测试的覆盖度. 要设计出一套全覆盖的测试并不是一件简单的事情, 越是复杂的系统, 全覆盖的测试就越难设计. 至少, 框架代码中提供的测试用例的覆盖度还是很有限的. 但是, 如何提高测试的覆盖度, 是学术界一直以来都在关注的问题.
- 尽早观测到error的存在. 观测到error的时机直接决定了调试的难度: 如果等到触发failure的时候才发现error的存在, 调试就会比较困难; 但如果能在error刚刚触发的时候就观测到它, 调试难度也就大大降低了. 事实上, 你已经见识过一些有用的工具了:
  - `-Wall`, `-Werror`: 在编译时刻把潜在的fault直接转变成failure. 这种工具的作用很有

限, 只能寻找一些在编译时刻也觉得可疑的fault, 例如 `if (p = NULL)` , 但也是代价最低的.

- `assert()` : 在运行时刻把error直接转变成failure. `assert()` 是一个很简单却又非常强大的工具, 只要在代码中定义好程序应该满足的特征, 就一定能在运行时刻将不满足这些特征的error拦截下来. 例如链表的实现, 我们只需要在代码中插入一些很简单的 `assert()` (例如指针不为空), 就能够几乎告别段错误. 事实上, 客户程序之所以会HIT BAD TRAP, 其实也是因为违背了我们设置的 `nemu_assert()` . 但是, 编写这些 `assert()` 其实需要我们对程序的行为有一定的了解, 同时在程序特征不易表达的时候, `assert()` 的作用也较为有限.
- `printf()` : 通过输出的方式观察潜在的error. 这是用于回溯fault时最常用的工具, 用于观测程序中的变量是否进入了错误的状态. 在NEMU中我们提供了输出更多调试信息的宏 `Log()` , 它实际上封装了 `printf()` 的功能. 但由于 `printf()` 需要根据输出的结果人工判断是否正确, 在便利程度上相对于 `assert()` 的自动判断就逊色了不少.
- GDB: 随时随地观测程序的任何状态. 调试器是最强大的工具, 但你需要在程序行为的茫茫大海中观测那些可疑的状态, 因此使用起来的代价也是最大的.

根据上面的分析, 我们就可以总结出一些调试的建议:

- 总是使用 `-Wall` 和 `-Werror`
- 尽可能多地在代码中插入 `assert()`
- `assert()` 无法捕捉到error时, 通过 `printf()` 输出可疑的变量, 期望能观测到error
- `printf()` 不易观测error时, 通过GDB理解程序的细致行为

Differential Testing

如果你在程序设计课上听说过上述这些建议, 相信你几乎不会遇到过运行时错误. 然而回过头来看上文提到的指令实现的bug, 我们会发现, 这些工具还是不够用: 我们很难通过 `assert()` 来表达指令的正确行为来进行自动检查, 而 `printf()` 和GDB实际上并没有缩短error和failure的距离.

如果有一种方法能够表达指令的正确行为, 我们就可以基于这种方法来进行类似 `assert()` 的检查了. 那么, 究竟什么地方表达了指令的正确行为呢? 最直接的, 当然就是i386手册了, 但是我们恰恰就是根据i386手册中的指令行为来在NEMU中实现指令的, 同一套方法不能既用于实现也用于检查. 如果有一个i386手册的参考实现就好了. 嘿! 我们用的真机不就是根据i386手册实现出来的吗? 我们让在NEMU中执行的每条指令也在真机中执行一次, 然后对比NEMU和真机的状态, 如果NEMU和真机的状态不一致, 我们就捕捉到error了!

这实际上是一种非常奏效的测试方法, 在软件测试领域称为differential testing. 我们刚才提到了"状态", 那"状态"具体指的是什么呢? 我们在PA1中已经认识到, 计算机就是一个数字电路. 那么, "计算机的状态"就恰恰是那些时序逻辑部件的状态, 也就是寄存器和内存的值. 其实仔细思考一下, 计算机执行指令, 就是修改这些时序逻辑部件的状态的过程. 要检查指令的实现是否正确, 只要检查这些时序逻辑部件中的值是否一致就可以了! Differential testing可以非常及时地

捕捉到error, 第一次发现NEMU的寄存器或内存的值与真机不一样的时候, 就是因为当时执行的指令实现有误导致的. 这时候其实离error非常接近, 防止了error进一步传播的同时, 要回溯找到fault也容易得多.

多么美妙的功能啊! 背后还蕴含着计算机本质的深刻原理! 但很遗憾, 不要忘记了, 真机上是运行了操作系统GNU/Linux的, 而NEMU中的测试程序是运行在AM上的. 就如前文所说, 它们提供的运行时环境是不一样的, 我们无法在GNU/Linux中运行基于 `x86-nemu` 的AM程序. 所以, 我们需要的不仅是一个i386手册的正确实现, 而且需要在上面能正确运行基于 `x86-nemu` 的AM程序.

事实上, QEMU就是一个不错的参考实现. 它是一个虚拟出来的完整的x86计算机系统, 而NEMU的目标只是虚拟出x86的一个子集, 能在NEMU上运行的程序, 自然也能在QEMU上运行. 因此, 为了通过differential testing的方法测试NEMU实现的正确性, 我们让NEMU和QEMU逐条指令地执行同一个客户程序. 双方每执行完一条指令, 就检查各自的寄存器和内存的状态, 如果发现状态不一致, 就马上报告错误, 停止客户程序的执行.

NEMU的框架代码已经准备好相应的功能了, 在 `nemu/include/common.h` 中定义宏 `DIFF_TEST` 之后, 重新编译NEMU后运行, 你会发现NEMU多输出了 `Connect to QEMU successfully` 的信息. 定义了宏 `DIFF_TEST` 之后, monitor会多进行以下初始化工作, 你不需要了解这些工作的具体细节, 只需要知道这是在为了让QEMU进入一个和NEMU同等的状态就可以了.

- 调用 `init_difftest()` 函数(在 `nemu/src/monitor/diff-test/diff-test.c` 中定义)来启动QEMU. 需要注意的是, 框架代码让QEMU运行在后台, 因此你将看不到QEMU的任何输出.
- 在 `load_img()` 的最后将客户程序拷贝一份副本到QEMU模拟的内存中.
- 在 `restart()` 中调用 `init_qemu_reg()` 函数(在 `nemu/src/monitor/diff-test/diff-test.c` 中定义), 来把QEMU的通用寄存器设置成和NEMU一样.

进行了上述初始化工作之后, QEMU和NEMU就处于相同的状态了. 接下来就要进行逐条指令执行后的状态对比了, 实现这一功能的是 `difftest_step()` 函数(在 `nemu/src/monitor/diff-test/diff-test.c` 中定义). 它会在 `exec_wrapper()` 的最后被调用, 在NEMU中执行完一条指令后, 就在 `difftest_step()` 中让QEMU执行相同的指令, 然后读出QEMU中的寄存器. 你需要添加相应的代码, 把NEMU的8个通用寄存器和eip与从QEMU中读出的寄存器的值进行比较, 如果发现值不一样, 就输出相应的提示信息, 并将 `diff` 标志设置为 `true` . 在 `difftest_step()` 的最后, 如果检测到 `diff` 标志为 `true` , 就停止客户程序的运行.

实现differential testing

在 `difftest_step()` 中添加相应的代码, 实现differential testing的核心功能. 实现正确后, 你将会得到一款无比强大的测试工具.

体会到differential testing的强大之后, 不妨思考一下: 作为一种基础设施, differential testing能帮助你节省多少调试的时间呢?

咦? 我们不需要对内存的状态进行比较吗? 事实上, NEMU是通过一套GDB协议与QEMU通信来获取QEMU的状态的, 但是通过这一协议还是不好获取指令修改的内存位置, 而对比整个内存又会带来很大的开销, 所以我们就不对内存的状态进行比较了. 事实上, NEMU中的简化实现也会导致某些寄存器的状态与QEMU的结果不一致, 例如EFLAGS, NEMU只实现了EFLAGS中的少量标志位, 同时也简化了某些指令对EFLAGS的更新. 另外, 一些特殊的系统寄存器也没有完整实现. 因此, 我们实现的differential testing并不是完整地对比QEMU和NEMU的状态, 但是不管是内存还是标志位, 只要客户程序的一条指令修改了它们, 在不久的将来肯定也会再次用到它们, 到时候一样能检测出状态的不同. 同时框架中也准备了`is_skip_nemu`和`is_skip_qemu`这两个变量, 用于跳过少量不易进行对比的指令. 因此, 我们其实牺牲了一些比较的精度, 来换取性能的提升, 但即使这样, 由于differential testing需要与QEMU进行通信, 这还是会把NEMU的运行速度拉低上百倍. 因此除非是在进行调试, 否则不建议打开differential testing的功能来运行NEMU.

一键回归测试

在实现指令的过程中, 你需要逐个测试用例地运行. 但在指令实现正确之后, 是不是意味着可以和这些测试用例说再见呢? 显然不是. 以后你还需要在NEMU中加入新的功能, 为了保证加入的新功能没有影响到已有功能的实现, 你还需要重新运行这些测试用例. 在软件测试中, 这个过程称为回归测试.

既然将来还要重复运行这些测试用例, 而手动重新运行每一个测试显然是一种效率低下的做法. 为了提高效率, 我们提供了一个用于一键回归测试的脚本. 在`nemu/`目录下运行

```
bash runall.sh
```

来自动批量运行`nexus-am/tests/cputest/`中的所有测试, 并报告每个测试用例的运行结果. 如果一个测试用例运行失败, 脚本将会保留相应的日志文件; 当使用脚本通过这个测试用例的时候, 日志文件将会被移除.

NEMU的本质

你已经知道, NEMU是一个用来执行其它程序的程序. 在可计算理论中, 这种程序有一个专门的名词, 叫通用程序(Universal Program), 它的通俗含义是: 其它程序能做的事情, 它也能做. 通用程序的存在性有专门的证明, 我们在这里不做深究, 但是, 我们可以写出NEMU, 可以用Docker/虚拟机做实验, 乃至我们可以在计算机上做各种各样的事情, 其背后都蕴含着通用程序的思想: NEMU和各种模拟器只不过是通用程序的实例化, 我们也可以毫不夸张地说, 计算机就是一个通用程序的实体化. 通用程序的存在性为计算机的出现奠定了理论基础, 是可计算理论中一个极其重要的结论, 如果通用程序的存在性得不到证明, 我们就没办法放心地使用计算机, 同时也不能义正辞严地说"机器永远是对的".

我们编写的NEMU最终会被编译成x86机器代码, 用x86指令来模拟x86程序的执行. 事实上在30多年前(1983年), Martin Davis教授 就在他出版的"Computability, complexity, and languages: fundamentals of theoretical computer science"一书中 提出了一种仅有三种指令的程序设计语言L语言, 并且证明了L语言和其它所有编程语言的计算能力等价. L语言中的三种指令分别是:

```
V = V + 1
V = V - 1
IF V != 0 GOTO LABEL
```

用x86指令来描述, 就是 `inc` , `dec` 和 `jne` 三条指令. 假设除了输入变量之外, 其它变量的初值都是0, 并且假设程序执行到最后一条指令就结束, 你可以仅用这三种指令写一个计算两个正整数相加的程序吗?

```
    # Assume a = 0, x and y are initialized with some positive integers.
    # Other temporary variables are initialized with 0.
    # Let "jne" carries a variable: jne v, label.
    # It means "jump to label if v != 0".
    # Compute a = x + y used only these three instructions: inc, dec, jnz.
    # No other instructions can be used.
    # The result should be stored in variable "a".
    # Have a try?
```

令人更惊讶的是, Martin Davis教授还证明了, 在不考虑物理限制的情况下(认为内存容量无限多, 每一个内存单元都可以存放任意大的数), 用L语言也可以编写出一个和NEMU类似的通用程序! 而且这个用L语言编写的通用程序的框架, 竟然还和NEMU中的 `cpu_exec()` 函数如出一辙: 取指, 译码, 执行... 这其实并不是巧合, 而是模拟(Simulation)在计算机科学中的应用.

早在Martin Davis教授提出L语言之前, 科学家们就已经在探索什么问题是可以计算的了. 回溯到19世纪30年代, 为了试图回答这个问题, 不同的科学家提出并研究了不同的计算模型, 包括Gödel, Herbrand和Kleen研究的递归函数, Church提出的λ-演算, Turing提出的图灵机, 后来发现这些模型在计算能力上都是等价的; 到了40年代, 计算机就被制造出来了. 后来甚至还有人证明了, 如果使用无穷多个算盘拼接起来进行计算, 其计算能力和图灵机等价! 我们可以从中得出一个推论, 通用程序在不同的计算模型中有不同的表现形式. NEMU作为一个通用程序, 在19世纪30年代有着非凡的意义. 如果你能在80年前设计出NEMU, 说不定"图灵奖"就要用你的名字来命名了. 计算的极限这一篇科普文章叙述了可计算理论的发展过程, 我们强烈建议你阅读它, 体会人类的文明(当然一些数学功底还是需要的). 如果你对可计算理论感兴趣, 可以选修宋方敏老师的计算理论导引课程.

把思绪回归到PA中, 通用程序的性质告诉我们, NEMU的潜力是无穷的. 为了创造出一个缤纷多彩的世界, 你觉得NEMU还缺少些什么呢?

捕捉死循环(有点难度)

NEMU除了作为模拟器之外, 还具有简单的调试功能, 可以设置断点, 查看程序状态. 如果让你为NEMU添加如下功能

> 当用户程序陷入死循环时, 让用户程序暂停下来, 并输出相应的提示信息

你觉得应该如何实现? 如果你感到疑惑, 在互联网上搜索相关信息.

温馨提示

PA2阶段2到此结束. 此阶段需要实现较多指令, 你有两周的时间来完成所有内容.

输入输出

我们已经成功运行了各个 `cputest` 中的测试用例, 但这些测试用例都只能默默地进行纯粹的计算. 回想起我们在程序设计课上写的第一个程序 `hello` , 至少也输出了一句话. 事实上, 输入输出是计算机与外界交互的基本手段, 如果你还记得计算机刚启动时执行的BIOS程序的全称是Basic Input/Output System, 你就会理解输入输出对计算机来说是多么重要了. 在真实的计算机中, 输入输出都是通过I/O设备来完成的.

设备的工作原理其实没什么神秘的. 你会在不久的将来在数字电路实验中看到键盘模块和VGA模块相关的verilog代码. 噢, 原来这些设备也一样是个数字电路! 事实上, 只要向设备发送一些有意义的数字信号, 设备就会按照这些信号的含义来工作. 让一些信号来指导设备如何工作, 这不就像"程序的指令指导CPU如何工作"一样吗? 恰恰就是这样! 设备也有自己的状态寄存器(相当于CPU的寄存器), 也有自己的功能部件(相当于CPU的运算器). 当然不同的设备有不同的功能部件, 例如键盘有一个把按键的模拟信号转换成扫描码的部件, 而VGA则有一个把像素颜色信息转换成显示器模拟信号的部件. 这些控制设备工作的信号称为"命令字", 可以理解成"设备的指令", 设备的工作就是负责接收命令字, 并进行译码和执行... 你已经知道CPU的工作方式, 这一切对你来说都太熟悉了. 唯一让你觉得神秘的, 就要数设备功能部件中的模/数转换, 数/模转换等各种有趣的实现. 遗憾的是, 我们的课程并没有为我们提供实践的机会, 因此它们成为了一种神秘的存在.

我们希望计算机能够控制设备, 让设备做我们想要做的事情, 这一重任毫无悬念地落到了CPU身上. CPU除了进行运算之外, 还需要与设备协作来完成不同的任务. 要控制设备工作, 就需要向设备发送命令字. 接下来的问题是, CPU怎么区分不同的设备? 具体要怎么向一个设备发送命令字?

对第一个问题的回答涉及到I/O的编址方式. 我们知道内存有地址的概念, 类似地, 我们也可以给I/O设备中允许CPU访问的寄存器逐一编址. I/O编址的目的就是让CPU可以区分不同的设备, 尽管这种区分的方式在我们来看非常笨拙: 只是让不同的设备报个数而已.

一种I/O编址方式是端口映射I/O(port-mapped I/O), CPU使用专门的I/O指令对设备进行访问, 并把设备的地址称作端口号. 有了端口号以后, 在I/O指令中给出端口号, 就知道要访问哪一个设备的哪一个寄存器了. 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. 设备中可能会有一些私有寄存器, 它们是由设备自己维护的, 它们没有端口号, CPU不能直接访问它们.

x86提供了 `in` 和 `out` 指令用于访问设备, 其中 `in` 指令用于将设备寄存器中的数据传输到CPU寄存器中, `out` 指令用于将CPU寄存器中的数据传送到设备寄存器中. 一个例子是 `nexus-am/am/arch/x86-nemu/src/trm.c` 中 `serial_init()` 的代码, 代码使用 `out` 指令给串口发送命令字. 例如

```
movl $0x0, %al
movl $0x3f9, %edx
outb %al, (%dx)
```

上述代码把数据0x0传送到0x3f9号端口所对应的设备寄存器中. 你要注意区分I/O指令和命令字, I/O指令是CPU执行的, 作用是对设备寄存器进行读写; 而命令字是设备来执行的, 作用和设备相关, 由设备来解释和执行. CPU执行上述代码后, 会将0x0这个数据传送到串口的一个寄存器中, 串口接收到0x0后, 把它解释成一条命令, 发现是一条关中断命令, 于是就会进入关中断状态; 但对CPU来说, 它并不关心0x0的含义, 只会老老实实地把0x0传送到0x3f9号端口. 至于设备接收到0x0之后会做什么, 那就是设备自己的事情了. 事实上, 设备的行为都会在相应的文档里面有清晰的定义, 驱动开发者需要阅读设备的相关文档, 编写相应的命令字序列来对设备进行期望的操作. 在PA中我们无需了解这些细节, 只需要知道, 我们可以通过阅读相关文档, 编写相应的程序在CPU上运行来操作设备即可.

端口映射I/O把端口号作为I/O指令的一部分, 这种方法很简单, 但同时也是它最大的缺点. 指令集为了兼容已经开发的程序, 是只能添加但不能修改的. 这意味着, 端口映射I/O所能访问的I/O地址空间的大小, 在设计I/O指令的那一刻就已经决定下来了. 所谓I/O地址空间, 其实就是所有能访问的设备的地址的集合. 随着设备越来越多, 功能也越来越复杂, I/O地址空间有限的端口映射I/O已经逐渐不能满足需求了. 有的设备需要让CPU访问一段较大的连续存储空间, 如VGA的显存, 24色加上Alpha通道的1024x768分辨率的显存就需要3MB的编址范围. 于是内存映射I/O(memory-mapped I/O)应运而生.

内存映射I/O这种编址方式非常巧妙, 它是通过不同的物理内存地址给设备编址的. 这种编址方式将一部分物理内存"重定向"到I/O地址空间中, CPU尝试访问这部分物理内存的时候, 实际上最终是访问了相应的I/O设备, CPU却浑然不知. 这样以后, CPU就可以通过普通的访存指令来访问设备. 这也是内存映射I/O得天独厚的好处: 物理内存的地址空间和CPU的位宽都会不断增长, 内存映射I/O从来不需要担心I/O地址空间耗尽的问题. 从原理上来说, 内存映射I/O唯一的缺点就是, CPU无法通过正常渠道直接访问那些被映射到I/O地址空间的物理内存了. 但随着计算机的发展, 内存映射I/O的唯一缺点已经越来越不明显了: 现代计算机都已经是64位计算机, 物理地址线都有48根, 这意味着物理地址空间有256TB这么大, 从里面划出3MB的地址空间给显存, 根本就是不痛不痒. 正因为如此, 内存映射I/O成为了现代计算机主流的I/O编址方式: RISC架构只提供内存映射I/O的编址方式, 而PCI-e, 网卡, x86的APIC等主流设备, 都支持通过内存映射I/O来访问.

内存映射I/O的一个例子是NEMU中的物理地址区间 `[0x40000, 0x80000)` . 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 *)0x40000, 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. 可以看到, 内存映射I/O的编程模型和普通的编程完全一样: 程序员可以直接把I/O设备当做内存来访问. 这一特性也是深受驱动开发者的喜爱.

理解volatile关键字

也许你从来都没听说过C语言中有 `volatile` 这个关键字, 但它从C语言诞生开始就一直存在. `volatile` 关键字的作用十分特别, 它的作用是避免编译器对相应代码进行优化. 你应该动手体会一下 `volatile` 的作用, 在GNU/Linux下编写以下代码:

```
void fun() {
    volatile unsigned char *p = (void *)0x8049000;
    *p = 0;
    while(*p != 0xff);
    *p = 0x33;
    *p = 0x34;
    *p = 0x86;
}
```

然后使用 `-O2` 编译代码. 尝试去掉代码中的 `volatile` 关键字, 重新使用 `-O2` 编译, 并对比去掉 `volatile` 前后反汇编结果的不同.

你或许会感到疑惑, 代码优化不是一件好事情吗? 为什么会有 `volatile` 这种奇葩的存在? 思考一下, 如果代码中的地址 `0x8049000` 最终被映射到一个设备寄存器, 去掉 `volatile` 可能会带来什么问题?

加入IOE

NEMU框架代码中已经提供了设备的代码, 位于 `nemu/src/device` 目录下. 代码提供了以下模块的模拟:

- 端口映射I/O和内存映射I/O两种I/O编址方式
- 串口, 时钟, 键盘, VGA四种设备

为了简化实现, 所有设备都是不可编程的, 只实现了在NEMU中用到的功能. 我们对代码稍作解释.

- `nemu/src/device/io/port-io.c` 是对端口I/O的模拟. 其中 `PIO_t` 结构用于记录一个端口I/O映射的关系, 设备会初始化时会调用 `add_pio_map()` 函数来注册一个端口I/O映射关系, 返回该映射关系的I/O空间首地址. `pio_read()` 和 `pio_write()` 是面向CPU的端口I/O读写接口. 由于NEMU是单线程程序, 因此只能串行模拟整个计算机系统的工作, 每次进行I/O读写的时候, 才会调用设备提供的回调函数(callback), 更新设备的状态. 内存映射I/O的模拟和端口I/O的模拟比较相似, 只是内存映射I/O的读写并不是面向CPU的, 这一点会在下文进行说明.

- `nemu/src/device/device.c` 含有和SDL库相关的代码, NEMU使用SDL库来模拟计算机的标准输入输出. `init_device()` 函数首先对以上四个设备进行初始化, 其中在初始化VGA时还会进行一些和SDL相关的初始化工作, 包括创建窗口, 设置显示模式等. 最后还会注册一个100Hz的定时器, 每隔0.01秒就会调用一次 `device_update()` 函数.
  `device_update()` 函数主要进行一些设备的模拟操作, 包括以50Hz的频率刷新屏幕, 以及检测是否有按键按下/释放. 需要说明的是, 代码中注册的定时器是虚拟定时器, 它只会在NEMU处于用户态的时候进行计时: 如果NEMU在 `ui_mainloop()` 中等待用户输入, 定时器将不会计时; 如果NEMU进行大量的输出, 定时器的计时将会变得缓慢. 因此除非你在进行调试, 否则尽量避免大量输出的情况, 从而影响定时器的工作.

我们提供的代码是模块化的, 要在NEMU中加入IOE, 你只需要在原来的代码上作少量改动: 在 `nemu/include/common.h` 中定义宏 `HAS_IOE` . 定义后, `init_device()` 函数会对设备进行初始化. 重新编译后, 你会看到运行NEMU时会弹出一个新窗口, 用于显示VGA的输出(见下文).

另一方面, 我们还需要在AM中实现相应的API为程序提供IOE的抽象 (在 `nexus-am/am/arch/x86-nemu/src/ioe.c` 中定义):

- `unsigned long _uptime()` 用于返回系统启动后经过的毫秒数
- `int _read_key()` 用于返回按键的键盘码, 若无按键, 则返回 `_KEY_NONE`
- `_Screen _screen` 结构用于指示屏幕的大小
- `void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h)` 用于将 `pixels` 指定的矩形像素绘制到屏幕中以 `(x, y)` 和 `(x+w, y+h)` 两点连线为对角线的矩形区域
- `void _draw_sync()` 用于将之前的绘制内容同步到屏幕上 (在NEMU中绘制内容总是会同步到屏幕上, 因而无需实现此API)
- `void _ioe_init()` 用于进行IOE相关的初始化工作, 调用后程序才能正确使用上述IOE相关的API

下面我们来逐一介绍如何在AM中添加IOE的功能来支撑程序的运行.

串口

串口是最简单的输出设备. `nemu/src/device/serial.c` 模拟了串口的功能. 其大部分功能也被简化, 只保留了数据寄存器和状态寄存器. 串口初始化时会注册 `0x3F8` 处长度为8个字节的端口作为其寄存器, 但代码中只模拟了其中的两个寄存器的功能. 由于NEMU串行模拟计算机系统的工作, 串口的状态寄存器可以一直处于空闲状态; 每当CPU往数据寄存器中写入数据时, 串口会将数据传送到主机的标准输出.

事实上, 我们之前提到的 `_putc()` 函数, 就是通过串口输出的. 然而AM却把 `_putc()` 放在TRM, 而不是IOE中, 这让人觉得有点奇怪. 的确, 可计算理论中提出的最原始的TRM并不包含输出的能力, 但对于一个现实的计算机系统来说, 输出是一个最基本的功能, 没有输出, 用户甚至无法知道程序具体在做什么. 因此在AM中, `_putc()` 的加入让TRM具有输出字符的能力, 被扩充后的TRM更靠近一个实用的机器, 而不再是只会计算的数学模型.

`nexus-am/am/arch/x86-nemu/src/trm.c` 中已经提供了串口的功能. 为了让程序使用串口进行输出, 你还需要在NEMU中实现端口映射I/O.

运行Hello World

实现 `in` , `out` 指令, 在它们的helper函数中分别调用 `pio_read()` 和 `pio_write()` 函数. 由于NEMU中有一些设备的行为是我们自定义的, 与QEMU中的标准设备的行为不完全一样(例如NEMU中的串口总是就绪的, 但QEMU中的串口并不是这样), 这导致在NEMU中执行 `in` 和 `out` 指令的结果与QEMU可能会存在不可调整的偏差. 为了使得differential testing可以正常工作, 我们在这两条指令中调用了相应的函数来设置 `is_skip_qemu` 标志, 来跳过与QEMU的检查.

实现后, 在 `nexus-am/am/arch/x86-nemu/src/trm.c` 中定义宏 `HAS_SERIAL` , 然后在 `nexus-am/apps/hello` 目录下键入

```
make run
```

在NEMU中运行基于AM的hello程序. 如果你的实现正确, 你将会看到程序往终端输出了10行 `Hello World!` (请注意不要让输出埋没在Log的海洋中).

需要注意的是, 这个hello程序和我们在程序设计课上写的第一个hello程序所处的层次是不一样的: 这个hello程序是可以说是直接运行在裸机上, 可以在AM的抽象下直接输出到设备(串口); 而我们在程序设计课上写的hello程序位于操作系统之上, 不能直接操作设备, 只能通过操作系统提供的服务进行输出, 输出的数据要经过很多层抽象才能到达设备层. 我们会在PA3中进一步体会操作系统的作用.

时钟

有了时钟, 程序才可以提供时间相关的体验, 例如游戏的帧率, 程序的快慢等.

`nemu/src/device/timer.c` 模拟了i8253计时器的功能. 计时器的大部分功能都被简化, 只保留了"发起时钟中断"的功能(目前我们不会用到). 同时添加了一个自定义的RTC(Real Time Clock), 初始化时将会注册 `0x48` 处的端口作为RTC寄存器, CPU可以通过I/O指令访问这一寄存器, 获得当前时间(单位是ms).

实现IOE

实现 `_uptime()` 后, 在NEMU中运行 `timetest` 程序 (在 `nexus-am/tests/timetest` 目录下, 编译和运行方式请参考上文, 此后不再额外说明). 如果你的实现正确, 你将会看到程序每隔1秒输出一句话.

native作为AM

"native"是指操作系统默认的运行时环境, 例如我们通过 `gcc hello.c` 编译程序时, 就会编译到GNU/Linux提供的运行时环境. 事实上, native也可以看做一个简单的AM, 目前只支持TRM和IOE. 但很快你就会看到, native也已经可以支撑很多程序的运行了.

看看NEMU跑多快

有了时钟之后, 我们就可以测试一个程序跑多快, 从而测试计算机的性能. 尝试在NEMU中依次运行以下benchmark(已经按照程序的复杂度排序, 均在 `nexus-am/apps` 目录下; 另外跑分时请注释掉 `nemu/include/common.h` 中的 `DEBUG` 和 `DIFF_TEST` 宏, 以获得较为真实的跑分):

- dhrystone
- coremark
- microbench

成功运行后会输出跑分. 跑分以 `i7-6700 @ 3.40GHz` 的处理器为参照, `100000` 分表示与参照机器性能相当, `100` 分表示性能为参照机器的千分之一. 除了和参照机器比较之外, 也可以和小伙伴进行比较. 如果把上述benchmark编译到native(编译和运行时添加 `ARCH=native` 参数), 还可以比较native的性能.

另外, microbench提供了两个不同规模的测试集 `test` 和 `ref`. 其中 `ref` 测试集规模较大, 用于跑分测试, 默认会编译 `ref` 测试集; `test` 测试集规模较小, 用于正确性测试, 需要在运行 `make` 时显式指定编译 `test` 测试集:

```
make INPUT=TEST
```

键盘

键盘是最基本的输入设备. 一般键盘的工作方式如下: 当按下一个键的时候, 键盘将会发送该键的通码(make code); 当释放一个键的时候, 键盘将会发送该键的断码(break code). `nemu/src/device/keyboard.c` 模拟了i8042通用设备接口芯片的功能. 其大部分功能也被简化, 只保留了键盘接口. i8042初始化时会注册 `0x60` 处的端口作为数据寄存器, 注册 `0x64` 处的端口作为状态寄存器. 每当用户敲下/释放按键时, 将会把相应的键盘码放入数据寄存器, 同时把状态寄存器的标志设置为 `1`, 表示有按键事件发生. CPU可以通过端口I/O访问这些寄存器, 获得键盘码. 在AM中, 我们约定通码的值为 `断码 + 0x8000`.

如何检测多个键同时被按下

在游戏中, 很多时候需要判断玩家是否同时按下了多个键, 例如RPG游戏中的八方向行走, 格斗游戏中的组合招式等等. 根据键盘码的特性, 你知道这些功能是如何实现的吗?

实现IOE(2)

实现 `_read_key()` 后, 在NEMU中运行 `keytest` 程序(在 `nexus-am/tests/keytest` 目录下). 如果你的实现正确, 在程序运行时弹出的新窗口中按下按键, 你将会看到程序输出相应的按键信息.

VGA

VGA可以用于显示颜色像素, 是最常用的输出设备. `nemu/src/device/vga.c` 模拟了VGA的功能. VGA初始化时注册了从 `0x40000` 开始的一段用于映射到video memory的物理内存. 在NEMU中, video memory是唯一使用内存映射I/O方式访问的I/O空间. 代码只模拟了 `400x300x32` 的图形模式, 一个像素占32个bit的存储空间, R(red), G(green), B(blue), A(alpha)各占8 bit, 其中VGA不使用alpha的信息. 如果你对VGA编程感兴趣, 这里有一个名为FreeVGA的项目, 里面提供了很多VGA的相关资料.

神奇的调色板

现代的显示器一般都支持24位的颜色(R, G, B各占8个bit, 共有 `2^8*2^8*2^8` 约1600万种颜色), 为了让屏幕显示不同的颜色成为可能, 在8位颜色深度时会使用调色板的概念. 调色板是一个颜色信息的数组, 每一个元素占4个字节, 分别代表R(red), G(green), B(blue), A(alpha)的值. 引入了调色板的概念之后, 一个像素存储的就不再是颜色的信息, 而是一个调色板的索引: 具体来说, 要得到一个像素的颜色信息, 就要把它的值当作下标, 在调色板这个数组中做下标运算, 取出相应的颜色信息. 因此, 只要使用不同的调色板, 就可以在不同的时刻使用不同的256种颜色了.

在一些90年代的游戏中, 很多渐出渐入效果都是通过调色板实现的, 聪明的你知道其中的玄机吗?

添加内存映射I/O

在 `paddr_read()` 和 `paddr_write()` 中加入对内存映射I/O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 `pmem` .

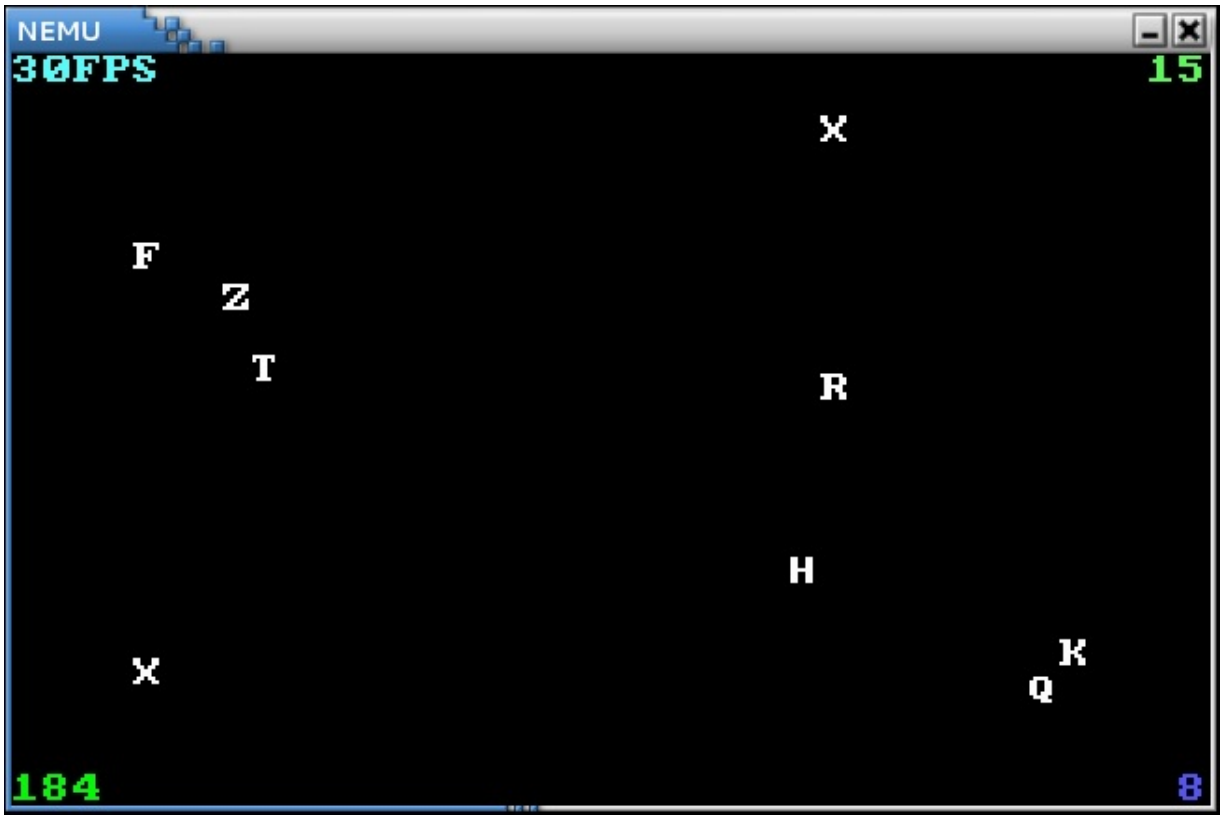
实现后, 在NEMU中运行 `videotest` 程序(在 `nexus-am/tests/videotest` 目录下). 如果内存映射I/O实现正确, 你会看到新窗口中输出了一些颜色信息.

实现IOE(3)

事实上, 刚才输出的颜色信息并不是 `videotest` 输出的画面, 这是因为框架代码中的 `_draw_rect()` 并未正确实现其功能. 你需要实现正确的 `_draw_rect()` . 实现后, 在NEMU中重新运行 `videotest` . 如果你的实现正确, 你将会看到新窗口中输出了相应的动画效果.

运行打字小游戏

在NEMU和AM中都完整实现IOE后, 我们就可以运行打字小游戏了(在 `nexus-am/apps/typing` 目录下). 打字小游戏来源于2013年NJUCS 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, 并去掉了浮点数.



有兴趣折腾的同学可以尝试在NEMU中运行litenes(在 `nexus-am/apps/litenes` 目录下). 没错, 我们在PA1的开头给大家介绍的红白机模拟器, 现在也已经可以在NEMU中运行起来了!

事实上, 我们已经实现了一个冯诺依曼计算机系统! 你已经在导论课上学习到, 冯诺依曼计算机系统由5个部件组成: 运算器, 控制器, 存储器, 输入设备和输出设备. 何况这些咋听之下让人云里雾里的名词, 现在都已经跃然"码"上: 你已经在NEMU中把它们都实现了! 再回过头来审视这一既简单又复杂的计算机系统: 说它简单, 它只不过在TRM的基础上添加了IOE, 本质上还是"取指->译码->执行"的工作方式, 甚至只要具备一些数字电路的知识就可以理解构建计算机的可能性; 说它复杂, 它却已经足够强大来支撑这么多酷炫的程序, 实在是让人激动不已啊! 那些看似简单但又可以折射出无限可能的事物, 其中承载的美妙规律容易使人们为之陶醉, 为之折服. 计算机, 就是其中之一.

必答题

你需要在实验报告中用自己的语言, 尽可能详细地回答下列问题.

- 编译与链接 在 `nemu/include/cpu/rtl.h` 中, 你会看到由 `static inline` 开头定义的各种

RTL指令函数. 选择其中一个函数, 分别尝试去掉 `static` , 去掉 `inline` 或去掉两者, 然后重新进行编译, 你会看到发生错误. 请分别解释为什么会发生这些错误? 你有办法证明你的想法吗?

- 编译与链接
  1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问重新编译后的NEMU含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?
  2. 添加上题中的代码后, 再在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问此时的NEMU含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.
  3. 修改添加的代码, 为两处 `dummy` 变量进行初始化: `volatile static int dummy = 0;` 然后重新编译NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)
- 了解Makefile 请描述你在 `nemu` 目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `nemu/build/nemu` . (这个问题包括两个方面: `Makefile` 的工作方式和编译链接的过程.) 关于 `Makefile` 工作方式的提示:
  - `Makefile` 中使用了变量, 包含文件等特性
  - `Makefile` 运用并重写了一些implicit rules
  - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
  - RTFM

温馨提示

PA2到此结束. 请你编写好实验报告(不要忘记在实验报告中回答必答题), 然后把命名为 `学号.pdf` 的实验报告文件放置在工程目录下, 执行 `make submit` 对工程进行打包, 最后将压缩包提交到指定网站.

PA3 - 穿越时空的旅程: 异常控制流

世界诞生的故事 - 第三章

冯诺依曼计算机果然功力深厚, 竟然能向冷冰冰的门电路赋予新的生命. 但为了应对各种突发情况, 先驱对计算机进行了改进.

在进行本PA前, 请在工程目录下执行以下命令进行分支整理, 否则将影响你的成绩:

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

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 50小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 实现第一个系统调用 - 2017/11/19 23:59:59
- 阶段2: 实现简易文件系统 - 2017/11/26 23:59:59
- 最后阶段: 运行仙剑奇侠传, 提交完整的实验报告 - 2017/12/03 23:59:59

提交说明: 见这里

操作系统 - 更方便的运行时环境

我们在PA2中已经实现了一个冯诺依曼计算机系统, 并且已经在AM上把打字游戏运行起来了. 有了IOE, 几乎能把各种小游戏移植到AM上来运行了. 但说起运行仙剑奇侠传, 我们目前暂时还无能为力. 这主要是因为, 仙剑奇侠传算得上是一个较为复杂的游戏了, 它会使用文件来管理游戏相关的数据. 一提到文件, 就已经超出了AM的能力范围了, 因为AM只是计算机的一种抽象模型, 是用来描述计算机如何构成的, 作为运行时环境对程序的支撑能力也很有限, 显然文件的概念并不属于AM.

事实上, 我们每天都使用的文件, 其实是操作系统提供的一种服务. 仔细想想, 我们使用的绝大部分程序, 都是在操作系统上运行的, 这是因为操作系统的层次比ISA和AM都要高, 自然能提供更丰富的抽象和更方便的运行时环境. 如果要让开发者在AM上进行开发, 估计各种游戏都早已陷入无尽跳票的死循环中了.

因此, 为了运行规模更大的程序, 我们需要操作系统的支持. 噢, 可别被操作系统这个庞然大物吓到了, 我们只需要一个支持文件操作的操作系统, 就可以支撑仙剑奇侠传的运行了. 感觉还是比较复杂啊, 我们还是先做一件最简单的事情: 先实现一个足够简单的操作系统, 来支撑dummy程序的运行.

RTFSC(4)

框架代码中已经为大家准备好了Nanos-lite的代码. Nanos-lite是南京大学操作系统Nanos的裁剪版, 是一个为PA量身订造的操作系统. 换句话说, 我们现在就要在NEMU上运行一个操作系统了(尽管这是一个比较简陋的操作系统), 同时也将带领你根据课堂上的知识剖析一个简单操作系统的组成. 这不仅是作为对这些抽象知识的很好的复(预)习, 同时也是为以后的操作系统实验打下坚实的基础, 而对PA来说最重要的是, 体会操作系统对运行程序的意义所在.

Nanos-lite已经包含了后续PA用到的所有模块, 由于NEMU的功能是逐渐添加的, Nanos-lite也要配合这个过程, 你会通过 `nanos-lite/src/main.c` 中的一些与实验进度相关的宏来控制Nanos-lite的功能. 随着实验进度的推进, 我们会逐渐讲解所有的模块, Nanos-lite做的工作也会越来越多. 因此在阅读Nanos-lite的代码时, 你只需要关心和当前进度相关的模块就可以了, 不要纠缠于和当前进度无关的代码.

```
nanos-lite
├── include
│   ├── common.h
│   ├── debug.h
│   ├── fs.h
│   ├── memory.h
│   └── proc.h
├── Makefile
└── src
    ├── device.c   # 设备抽象
    ├── fs.c       # 文件系统
    ├── initrd.S   # ramdisk设备
    ├── irq.c      # 中断异常处理
    ├── loader.c   # 加载器
    ├── main.c
    ├── mm.c       # 存储管理
    ├── proc.c     # 进程调度
    ├── ramdisk.c  # ramdisk驱动程序
    └── syscall.c  # 系统调用处理
```

需要提醒的是, Nanos-lite是运行在AM之上的, AM的API在Nanos-lite中都是可用的. 因此我们会有以下说法:

```
Nanos-lite是NEMU的客户程序, 它运行在AM之上.
仙剑是Nanos-lite的用户程序, 它运行在Nanos-lite之上.
```

另外, 虽然不会引起明显的误解, 但在引入Nanos-lite之后, 我们还是会在某些地方使用"用户进程"的概念, 而不是"用户程序". 如果你现在不能理解什么是进程, 你只需要把进程作为"正在运行的程序"来理解就可以了. 还感觉不出这两者的区别? 举一个简单的例子吧, 如果你打开了记事本3次, 计算机上就会有3个记事本进程在运行, 但磁盘中的记事本程序只有一个. 进程是操作系统中一个重要的概念, 有关进程的详细知识会在操作系统课上进行介绍.

一开始, 在 `nanos-lite/src/main.c` 中所有与实验进度相关的宏都没有定义, 此时Nanos-lite的功能十分简单. 我们来简单梳理一下Nanos-lite目前的行为:

1. 通过 `Log()` 输出hello信息和编译时间. 需要说明的是, Nanos-lite中定义的 `Log()` 宏并不是NEMU中定义的 `Log()` 宏. Nanos-lite和NEMU是两个独立的项目, 它们的代码不会相互影响, 你在阅读代码的时候需要注意这一点. 在Nanos-lite中, `Log()` 宏通过 `klib` 中的 `printk()` 输出, 最终会调用TRM的 `_putc()` .
2. 初始化ramdisk. 在一个完整的模拟器中, 程序应该存放在磁盘中. 但目前我们并没有实现磁盘的模拟, 因此先把Nanos-lite中的一段内存作为磁盘来使用. 这样的磁盘有一个专门的名字, 叫ramdisk.
3. 调用 `init_device()` 对设备进行一些初始化操作. 目前 `init_device()` 会直接调用 `_ioe_init()` .
4. 调用 `loader()` 函数加载用户程序, 函数会返回用户程序的入口地址. 其中 `loader()` 函数

并未实现, 我们会在下文进行说明.

5. 跳转到用户程序的入口执行.

加载操作系统的第一个用户程序

loader是一个用于加载程序的模块. 我们知道程序中包括代码和数据, 它们都是存储在可执行文件中. 加载的过程就是把可执行文件中的代码和数据放置在正确的内存位置, 然后跳转到程序入口, 程序就开始执行了. 更具体的, 为了实现 `loader()` 函数, 我们需要解决以下问题:

- 可执行文件在哪里?
- 代码和数据在可执行文件的哪个位置?
- 代码和数据有多少?
- "正确的内存位置"在哪里?

为了回答第一个问题, 我们还要先说明一下用户程序是从哪里来的. 由于用户程序运行在操作系统之上, 不能与AM所提供的运行时环境相适配了, 因此我们不能把编译到AM上的程序放到操作系统上运行. 为此, 我们准备了一个新的子项目Navy-apps, 专门用于编译出操作系统的用户程序.

```
navy-apps
├── apps            # 用户程序
│   ├── init
│   ├── litenes
│   ├── lua
│   ├── nterm
│   ├── nwm
│   └── pal         # 仙剑奇侠传
├── fsimg           # 根文件系统
├── libs            # 库
│   ├── libc        # Newlib C库
│   ├── libfont
│   ├── libndl
│   └── libos       # 系统调用的用户层封装
├── Makefile
├── Makefile.app
├── Makefile.check
├── Makefile.compile
├── Makefile.lib
├── README.md
└── tests           # 一些测试
```

其中, `navy-apps/libs/libc` 中是一个名为Newlib的项目, 它是一个专门为嵌入式系统提供的C库, 库中的函数对运行时环境的要求极低. 这对Nanos-lite来说是非常友好的, 我们不需要为了配合C库而在Nanos-lite中实现额外的功能. 用户程序的入口位于 `navy-apps/libs/libc/start.c` 中的 `_start()` 函数, 它会调用用户程序的 `main()` 函数, 从 `main()` 函数返回后会调用 `exit()` 结束运行.

我们要在Nanos-lite上运行的第一个用户程序是 `navy-apps/tests/dummy/dummy.c` . 首先我们让Navy-apps项目上的程序默认编译到 `x86` 中:

```
--- navy-apps/Makefile.check
+++ navy-apps/Makefile.check
@@ -1,1 +1,1 @@
-ISA ?= native
+ISA ?= x86
```

然后在 `navy-apps/tests/dummy` 下执行

```
make
```

就会在 `navy-apps/tests/dummy/build/` 目录下生成dummy的可执行文件. 编译Newlib时会出现较多warning, 我们可以忽略它们. 为了避免和Nanos-lite的内容产生冲突, 我们约定目前用户程序需要被链接到内存位置 `0x4000000` 处, Navy-apps已经设置好了相应的选项(见 `navy-apps/Makefile.compile` 中的 `LDFLAGS` 变量).

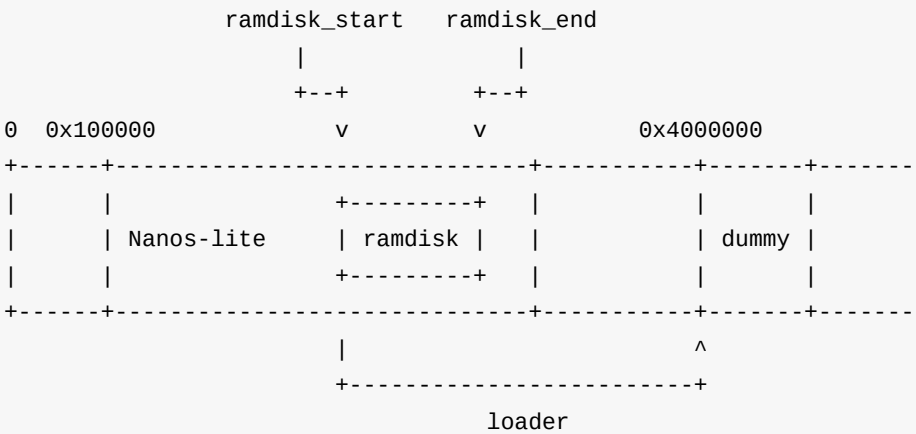
在 `nanos-lite/` 目录下执行

```
make update
```

`nanos-lite/Makefile` 中会将其生成ramdisk镜像文件 `ramdisk.img` , 并包含进Nanos-lite成为其中的一部分(在 `nanos-lite/src/initrd.S` 中实现). 现在的ramdisk十分简单, 它只有一个文件, 就是我们将要加载的用户程序, 这其实已经回答了上述第一个问题: 可执行文件位于ramdisk偏移为0处, 访问它就可以得到用户程序的第一个字节.

为了回答剩下的问题, 我们首先需要了解可执行文件是如何组织的. 你应该已经在课堂上学习过ELF文件格式了, 它除了包含程序本身的代码和静态数据之外, 还包括一些用来描述它们的组织信息. 事实上, 我们的loader目前并没有必要去解析并加载ELF文件. 为了简化, `nanos-lite/Makefile` 中已经把用户程序运行所需要的代码和静态数据通过 `objcopy` 工具从ELF文件中抽取出来了, 整个ramdisk本身就已经存放了loader所需要加载的内容. 最后, "正确的内存位置", 也就是我们上文提到的约定好的 `0x4000000` 了.

所以, 目前的loader只需要做一件事情: 将ramdisk中从0开始的所有内容放置在 `0x4000000` , 并把这个地址作为程序的入口返回即可. 我们把这个简化了的loader称为raw program loader. 我们通过内存布局来理解loader目前需要做的事情:



框架代码提供了一些ramdisk相关的函数(在 `nanos-lite/src/ramdisk.c` 中定义), 你可以使用它们来实现loader的功能:

```
// 从ramdisk中`offset`偏移处的`len`字节读入到`buf`中
void ramdisk_read(void *buf, off_t offset, size_t len);

// 把`buf`中的`len`字节写入到ramdisk中`offset`偏移处
void ramdisk_write(const void *buf, off_t offset, size_t len);

// 返回ramdisk的大小, 单位为字节
size_t get_ramdisk_size();
```

真实操作系统中的loader远比我们目前在Nanos-lite中实现的loader要复杂. 事实上, Nanos-lite的loader设计其实也向我们展现出了程序的最为原始的状态: 一个凝结着人类智慧设计的精妙算法, 承载着人类劳动收集的宝贵数据的...比特串! 加载程序其实就是把这一无比珍贵的比特串放置在正确的位置, 但这看似平凡无比的比特串当中又蕴含着"存储程序"的划时代思想: 当操作系统将控制权交给它的时候, 计算机以把它解释成指令并逐条执行, 却让这一比特串真正发挥出它足以改变世界的潜能.

实现loader

你需要在Nanos-lite中实现loader的功能, 来把用户程序加载到正确的内存位置, 然后执行用户程序.

需要注意的是, 每当ramdisk中的内容需要更新时, 你都需要在 `nanos-lite/` 目录下手动执行

```
make update
```

来更新Nanos-lite中的ramdisk内容, 然后再通过

```
make run
```

来在NEMU上运行带有最新版ramdisk的Nanos-lite.

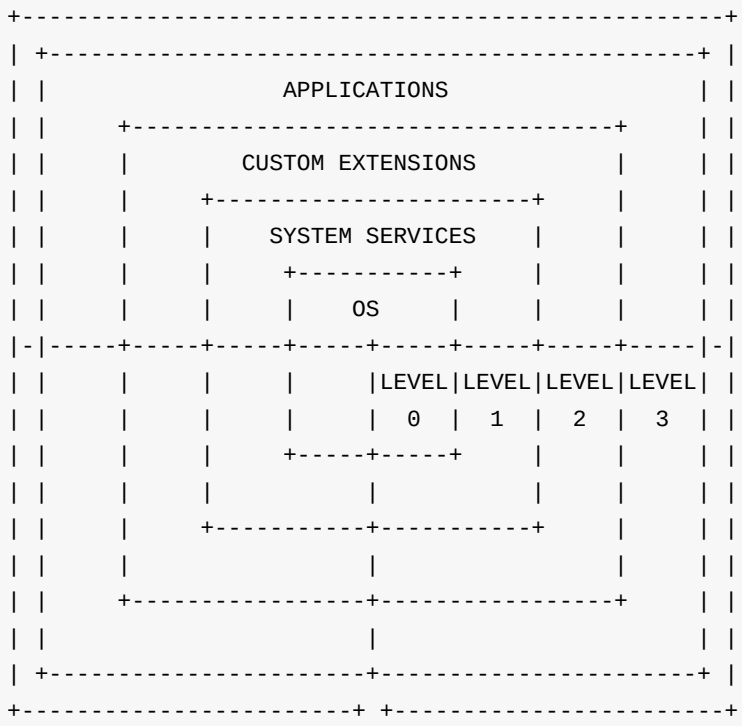
实现正确后, 你会看到dummy程序执行了一条未实现的 `int` 指令, 这说明loader已经成功加载dummy, 并且成功地跳转到dummy中执行了. 未实现的 `int` 指令我们会接下来的内容中进行说明.

等级森严的制度

我们在dummy程序中碰到了一条看似奇怪的 `int` 指令. 为了解释它, 我们还需要了解它背后折射出来的计算机和谐社会的故事.

为了构建计算机和谐社会, i386强化了保护模式(protected mode)和特权级(privilege level)的概念: 简单地说, 只有高特权级的进程才能去执行一些系统级别的操作, 如果一个特权级低的进程尝试执行它没有权限执行的操作, CPU将会抛出一个异常. 一般来说, 最适合担任系统管理员的角色就是操作系统了, 它拥有最高的特权级, 可以执行所有操作; 而除非经过允许, 运行在操作系统上的用户进程一般都处于最低的特权级, 如果它试图破坏社会的和谐, 它将会被判"死刑".

在i386中, 存在0, 1, 2, 3四个特权级, 0特权级最高, 3特权级最低. 特权级n所能访问的资源, 在特权级0~n也能访问. 不同特权级之间的关系就形成了一个环: 内环可以访问外环的资源, 但外环不能进入内环的区域, 因此也有"ring n"的说法来描述一个进程所在的特权级.



虽然80386提供了4个特权级, 但大多数通用的操作系统只会使用0级和3级: 操作系统处在ring 0, 一般的程序处在ring 3, 这就已经起到保护的作用了. 那CPU是怎么判断一个进程是否执行了无权限操作呢? 在这之前, 我们还要简单地了解一下i386中引入的与特权级相关的概念:

- DPL(Descriptor Privilege Level)属性描述了一段数据所在的特权级
- RPL(Requestor's Privilege Level)属性描述了请求者所在的特权级
- CPL(Current Privilege Level)属性描述了当前进程的特权级,

一次数据的访问操作是合法的, 当且仅当

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

两式同时成立, 注意这里的 `>=` 是数值上的(numerically greater). <1>式表示请求者有权限访问目标数据, <2>式表示当前进程也有权限访问目标数据. 如果违反了上述其中一式, 此次操作将会被判定为非法操作, CPU将会抛出异常, 跳转到一个约定好的代码位置, 然后通知操作系统进行处理.

对RPL的补充

你可能会觉得RPL十分令人费解, 我们先举一个生活上的例子.

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

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

通常情况下, 操作系统运行在ring 0, CPL为0, 因此有权限访问所有的数据; 而用户进程运行在ring 3, CPL为3, 这就决定了它只能访问同样处在ring3的数据. 这样, 只要操作系统将其私有数据放在ring 0中, 恶意程序就永远没有办法访问到它们. 这些保护相关的概念和检查过程都是通过硬件实现的, 只要软件运行在硬件上面, 都无法逃出这一天网. 硬件保护机制使得恶意程序永远无法全身而退, 为构建计算机和谐社会作出了巨大的贡献.

这是多美妙的功能! 遗憾的是, 上面提到的很多概念其实只是一带而过, 真正的保护机制也还需要考虑更多的细节. i386手册中专门有一章来描述保护机制, 就已经看出来这并不是简单说说而已. 根据KISS法则, 我们并不打算在NEMU中加入保护机制. 我们让所有用户进程都运行在ring 0, 虽然所有用户进程都有权限执行所有指令, 不过由于PA中的用户程序都是我们自己编写的, 一切还是在我们的控制范围之内. 毕竟, 我们也已经从上面的故事中体会到保护机制的本质了: 在硬件中加入一些与特权级检查相关的门电路, 如果发现了非法操作, 就会抛出一个异常, 让CPU跳转到一个固定的地方, 并进行后续处理.

操作系统的义务

既然操作系统位于ring 0享受着至高无上的权利, 自然地它也需要履行相应的义务, 那就是: 管理系统中的所有资源, 为用户进程提供相应的服务. 举一个银行的例子, 如果银行连最基本的取款业务都不能办理, 是没有客户愿意光顾它的. 但同时银行也不能允许客户亲自到金库里取款, 而是需要客户按照规定的手续来办理取款业务. 同样地, 操作系统并不允许用户进程直接操作显示器硬件进行输出, 否则恶意程序就很容易往显示器中写入恶意数据, 让屏幕保持黑屏, 影响其它进程的使用. 因此, 用户进程想输出一句话, 也要经过一定的合法手续向操作系统进行申请, 这一合法手续就是系统调用.

我们到银行办理业务的时候, 需要告诉工作人员要办理什么业务, 账号是什么, 交易金额是多少, 这无非是希望工作人员知道我们具体想做什么. 用户进程执行系统调用的时候也是类似的情况, 要通过一种方法描述自己的需求, 然后告诉操作系统. 用来描述需求最方便的手段就是使用通用寄存器了, 用户进程将系统调用的参数依次放入各个寄存器中(第一个参数放在 `%eax` 中, 第二个参数放在 `%ebx` 中...). 为了让操作系统注意到用户进程提交的申请, 系统调用通常都会触发一个异常, 然后陷入操作系统. 在GNU/Linux中, 系统调用产生的异常通过 `int $0x80` 指令触发. 这个异常和上文提到的非法操作产生的异常不同, 操作系统能够识别它是由系统调用产生的.

Navy-apps已经为用户程序准备好了系统调用的接口了. `navy-apps/libs/libos/src/nanos.c` 中定义的 `_syscall_()` 函数已经蕴含着上述过程:

```
int _syscall_(int type, uintptr_t a0, uintptr_t a1, uintptr_t a2) {
  int ret;
  asm volatile("int $0x80": "=a"(ret): "a"(type), "b"(a0), "c"(a1), "d"(a2));
  return ret;
}
```

上述内联汇编会先把系统调用的参数依次放入 `%eax` , `%ebx` , `%ecx` , `%edx` 四个寄存器中, 然后执行 `int $0x80` 手动触发一个特殊的异常. 操作系统捕获这个异常之后, 发现是一个系统调用, 就会调出相应的处理函数进行处理, 处理结束后设置好返回值, 然后返回到上述的内敛汇编中. 内联汇编最后从 `%eax` 寄存器中取出系统调用的返回值, 并返回给调用该接口的函数, 告知其系统调用执行的情况(如是否成功等).

我们可以在GNU/Linux下编写一个程序, 来手工触发一次 `write` 系统调用:

```
const char str[] = "Hello world!\n";

int main() {
  asm volatile ("movl $4, %eax;"      // system call ID, 4 = SYS_write
                "movl $1, %ebx;"      // file descriptor, 1 = stdout
                "movl $str, %ecx;"    // buffer address
                "movl $13, %edx;"      // length
                "int $0x80");

  return 0;
}
```

如果你在64位操作系统上运行它, 你需要在编译的时候加入 `-m32` 参数来生成32位的代码. 用户进程执行上述代码, 就相当于告诉操作系统: 帮我把从 `str` 开始的13字节写到1号文件中去. 其中"写到1号文件中去"的功能相当于输出到屏幕上.

虽然操作系统需要为用户进程服务, 但这并不意味着操作系统需要把所有信息都暴露给用户程序. 有些信息是用户进程没有必要知道的, 也永远不应该知道, 例如一些与内存管理相关的数据结构. 如果一个恶意程序获得了这些信息, 可能会为恶意攻击提供了信息基础. 因此, 通常不存在一个系统调用来获取这些操作系统的私有数据.

穿越时空的旅程

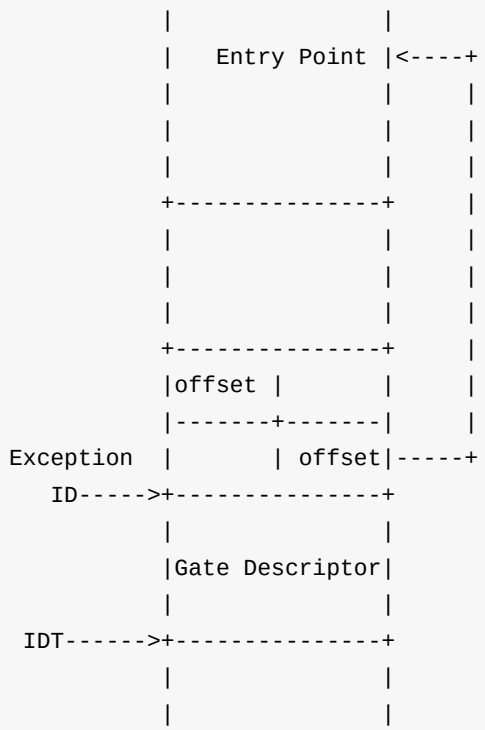
异常是指CPU在执行过程中检测到的不正常事件, 例如除数为零, 无效指令, 权限不足等. i386还向软件提供 `int` 指令, 让软件可以手动产生异常, 因此前面提到的系统调用也算是一种特殊的异常. 那触发异常之后都发生了些什么呢? 我们先来对这一场神秘的时空之旅作一些简单的描述.

我们之前提到, CPU检测到异常之后, 就会跳转到一个地方, 这个过程是由位于硬件层次i386中断机制支撑的. 在i386中, 上述跳转的目标通过门描述符(Gate Descriptor)来指示. 门描述符是一个8字节的结构体, 里面包含着不少细节的信息, 我们在NEMU中简化了门描述符的结构, 只保留存在位P和偏移量OFFSET:

```
 31                23                15                7                0
+-----------------+-----------------+---+-----------------------------+
|          OFFSET 31..16            | P |          Don't care         |4
+-----------------------------------+---+-----------------------------+
|             Don't care            |           OFFSET 15..0          |0
+-----------------+-----------------+-----------------+-----------------+
```

P位来用表示这一个门描述符是否有效, OFFSET用来指示跳转目标.

为了方便管理各个门描述符, i386把内存中的某一段数据专门解释成一个数组, 叫IDT(Interrupt Descriptor Table, 中断描述符表), 数组的一个元素就是一个门描述符. 为了从数组中找到一个门描述符, 我们还需要一个索引. 对于CPU异常来说, 这个索引由CPU内部产生(例如除零异常为0号异常), 或者由 `int` 指令给出(例如 `int $0x80`). 最后, 为了在内存中找到IDT, i386使用IDTR寄存器来存放IDT的首地址和长度. 我们需要通过软件代码事先把IDT准备好, 然后通过一条特殊的指令 `lidt` 在IDTR中设置好IDT的首地址和长度, 这一中断处理机制就可以正常工作了. 现在是万事俱备, 等到异常的东风一刮, CPU就会按照设定好的IDT跳转到目标地址:



但感觉有什么不太对劲? 异常处理结束之后, 我们要怎么返回异常之前的状态呢? 为了方便叙述, 我们称触发异常之前状态为S. 为了以后能够完美地恢复到S, 在开始真正的处理异常之前应该先把S保存起来, 等到异常处理结束之后, 才能根据之前保存的信息把计算机恢复到S的样子. 哪些内容表征了S? 首先当然是EIP了, 它指示了S正在执行的指令(或者下一条指令); 然后就是EFLAGS(各种标志位)和CS(代码段寄存器, 里面包含CPL的信息). 由于一些特殊的原因, 这三个寄存器的内容必须由硬件来保存. 要将这些信息保存到哪里去呢? 一个合适的地方就是进程的堆栈. 触发异常时, 硬件会自动将EFLAGS, CS, EIP三个寄存器的值保存到堆栈上.

于是, 触发异常后硬件的处理如下:

1. 依次将EFLAGS, CS, EIP寄存器的值压入堆栈
2. 从IDTR中读出IDT的首地址
3. 根据异常(中断)号在IDT中进行索引, 找到一个门描述符
4. 将门描述符中的offset域组合成目标地址
5. 跳转到目标地址

需要注意的是, 这些工作都是硬件自动完成的, 不需要程序员编写指令来完成相应的内容. 事实上, 这只是一个简化后的过程, 在真实的计算机上还要处理很多细节问题, 在这里我们就不深究了. i386手册中还记录了处理器对中断号和异常号的分配情况, 并列出了各种异常的详细解释, 需要了解的时候可以进行查阅.

在计算机和谐社会中, 大部分门描述符都不能让用户进程随意使用, 否则恶意程序就可以通过 `int` 指令欺骗操作系统. 例如恶意程序执行 `int $0x2` 来谎报电源掉电, 扰乱其它进程的正常运行. 因此执行 `int` 指令也需要进行特权级检查, 但PA中就不实现这一保护机制了, 具体的检查规则我们也就不展开讨论了, 需要了解的时候请查阅i386手册.

加入ASYE

在AM的模型中, 异常处理的能力被划分到ASYE模块中. 老规矩, 我们还是分别从NEMU和AM两个角度来体会硬件和软件如何相互协助来支持ASYE的功能.

准备IDT

首先是要准备一个有意义的IDT, 这样以后触发异常时才能跳转到正确的目标地址. 具体的, 你需要在NEMU中添加IDTR寄存器和 `lidt` 指令. 然后在 `nanos-lite/src/main.c` 中定义宏 `HAS_ASYE` , 这样以后, Nanos-lite会多进行一项初始化工作: 调用 `init_irq()` 函数, 这最终会调用位于 `nexus-am/am/arch/x86-nemu/src/asye.c` 中的 `_asye_init()` 函数. `_asye_init()` 函数会做两件事情, 第一件就是初始化IDT:

1. 代码定义了一个结构体数组 `idt` , 它的每一项是一个门描述符结构体
2. 在相应的数组元素中填写有意义的门描述符, 例如编号为 `0x80` 的门描述符就是将来系统调用的入口地址. 需要注意的是, 框架代码中还是填写了完整的门描述符(包括上文中提到的don't care的域), 这主要是为了在QEMU中进行differential testing时也能跳转到正确的入口地址. QEMU实现了完整的中断机制, 如果只填写简化版的门描述符, 就无法在QEMU中正确运行. 但我们无需了解其中的细节, 只需要知道代码已经填写了正确的门描述符即可.
3. 在IDTR中设置 `idt` 的首地址和长度

`_asye_init()` 函数做的第二件事是注册一个事件处理函数, 这个事件处理函数由 `_asyn_init()` 的调用者提供. 关于事件处理函数, 我们会在下文进行更多的介绍.

触发异常

为了测试是否已经成功准备IDT, 我们还需要真正触发一次异常, 看是否正确地跳转到目标地址. 具体的, 你需要在NEMU中实现 `raise_intr()` 函数(在 `nemu/src/cpu/intr.c` 中定义) 来模拟上文提到的i386中断机制的处理过程:

```
void raise_intr(uint8_t NO, vaddr_t save_addr) {
  /* TODO: Trigger an interrupt/exception with ``NO''.
   * That is, use ``NO'' to index the IDT.
   */

}
```

需要注意的是:

- PA不涉及特权级的切换, 查阅i386手册的时候你不需要关心和特权级切换相关的内容.
- 通过IDTR中的地址对IDT进行索引的时候, 需要使用 `vaddr_read()` .
- PA中不实现分段机制, 没有CS寄存器的概念. 但为了在QEMU中顺利进行differential

testing, 我们还是需要在cpu结构体中添加一个CS寄存器, 并在 `restart()` 函数中将其初始化为 `8` .
- 由于中断机制需要对EFLAGS进行压栈, 为了配合differential testing, 我们还需要在 `restart()` 函数中将EFLAGS初始化为 `0x2` .
- 执行 `int` 指令后保存的 `EIP` 指向的是 `int` 指令的下一条指令, 这有点像函数调用, 具体细节可以查阅i386手册.
- 你需要在 `int` 指令的helper函数中调用 `raise_intr()` , 而不要把中断机制的代码放在 `int` 指令的helper函数中实现, 因为在后面我们会再次用到 `raise_intr()` 函数.

实现中断机制

你需要实现上文提到的 `lidt` 指令和 `int` 指令, 并实现 `raise_intr()` 函数.

实现正确后, 重新在Nanos-lite上运行dummy程序, 如果你看到在 `vecsys()` (在 `nexum-am/am/arch/x86-nemu/src/trap.S` 中定义)附近触发了未实现指令, 说明你的中断机制实现正确.

保存现场

成功跳转到入口函数 `vecsys()` 之后, 我们就要在软件上开始真正的异常处理过程了. 但是, 进行异常处理的时候不可避免地需要用到通用寄存器, 然而看看现在的通用寄存器, 里面存放的都是异常触发之前的内容. 这些内容也是现场的一部分, 如果不保存就覆盖它们, 将来就无法恢复异常触发之前的状态了. 但硬件并不负责保存它们, 因此需要通过软件代码来保存它们的值. i386提供了 `pusha` 指令, 用于把通用寄存器的值压入堆栈.

`vecsys()` 会压入错误码和异常号 `#irq` , 然后跳转到 `asm_trap()` . 在 `asm_trap()` 中, 代码将会把用户进程的通用寄存器保存到堆栈上. 这些寄存器的内容连同之前保存的错误码, `#irq` , 以及硬件保存的EFLAGS, CS, EIP, 形成了trap frame(陷阱帧)的数据结构. 我们知道栈帧记录了函数调用时的状态, 而相应地, 陷阱帧则完整记录了用户进程触发异常时现场的状态, 将来恢复现场就靠它了.

对比异常与函数调用

我们知道进行函数调用的时候也需要保存调用者的状态: 返回地址, 以及调用约定(calling convention)中需要调用者保存的寄存器. 而进行异常处理之前却要保存更多的信息. 尝试对比它们, 并思考两者保存信息不同是什么原因造成的.

注意到trap frame是在堆栈上构造的. 接下来代码将会把当前的 `%esp` 压栈, 并调用C函数 `irq_handle()` (在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中定义).

诡异的代码

`trap.S` 中有一行 `pushl %esp` 的代码, 乍看之下其行为十分诡异. 你能结合前后的代码理解它的行为吗? Hint: 不用想太多, 其实都是你学过的知识.

重新组织TrapFrame结构体

你的任务如下:

- 实现 `pusha` 指令, 你需要注意压栈的顺序, 更多信息请查阅i386手册.
- 理解trap frame形成的过程, 然后重新组织 `nexus-am/am/arch/x86-nemu/include/arch.h` 中定义的 `_RegSet` 结构体的成员, 使得这些成员声明的顺序和 `nexus-am/am/arch/x86-nemu/src/trap.S` 中构造的trap frame保持一致.

实现正确之后, `irq_handle()` 以及后续代码就可以正确地使用trap frame了. 重新在Nanos-lite上运行dummy程序, 你会看到在 `nanos-lite/src/irq.c` 中的 `do_event()` 函数中触发了BAD TRAP:

```
[src/irq.c,5,do_event] {kernel} system panic: Unhandled event ID = 8
```

事件分发

`irq_handle()` 的代码会把异常封装成事件, 然后调用在 `_asye_init()` 中注册的事件处理函数, 将事件交给它来处理. 在Nanos-lite中, 这一事件处理函数是 `nanos-lite/src/irq.c` 中的 `do_event()` 函数. `do_event()` 函数会根据事件类型再次进行分发. 我们刚才触发了一个未处理的8号事件, 这其实是一个系统调用事件 `_EVENT_SYSCALL` (在 `nexus-am/am/am.h` 中定义). 在识别出系统调用事件后, 需要调用 `do_syscall()` (在 `nanos-lite/src/syscall.c` 中定义)进行处理.

系统调用处理

我们终于正式进入系统调用的处理函数中了. `do_syscall()` 首先通过宏 `SYSCALL_ARG1()` 从现场 `r` 中获取用户进程之前设置好的系统调用参数, 通过第一个参数 - 系统调用号 - 进行分发. 但目前Nanos-lite没有实现任何系统调用, 因此触发了panic.

添加一个系统调用比你想象中要简单, 所有信息都已经准备好了. 我们只需要在分发的过程中添加相应的系统调用号, 并编写相应的系统调用处理函数 `sys_xxx()`, 然后调用它即可.

回过头来看 `dummy` 程序, 它触发了一个号码为 `0` 的 `SYS_none` 系统调用. 我们约定, 这个系统调用什么都不用做, 直接返回 `1`.

处理系统调用的最后一件事就是设置系统调用的返回值. 我们约定系统调用的返回值存放在系统调用号所在的寄存器中, 所以我们只需要通过 `SYSCALL_ARG1()` 来进行设置就可以了.

恢复现场

系统调用处理结束后, 代码将会一路返回到 `trap.S` 的 `asm_trap()` 中. 接下来的事情就是恢复用户进程的现场. `asm_trap()` 将根据之前保存的trap frame中的内容, 恢复用户进程的通用寄存器 (注意trap frame中的 `%eax` 已经被设置成系统调用的返回值了), 并直接弹出一些不再需要的信息, 最后执行 `iret` 指令. `iret` 指令用于从异常处理代码中返回, 它将栈顶的三个元素来依次解释成EIP, CS, EFLAGS, 并恢复它们. 用户进程可以通过 `%eax` 寄存器获得系统调用的返回值, 进而得知系统调用执行的结果. 在它看来, 这次时空之旅就好像没有发生过一样.

实现系统调用

你需要:

1. 在 `do_event()` 中识别出系统调用事件 `_EVENT_SYSCALL`, 然后调用 `do_syscall()`.
2. 在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中实现正确的 `SYSCALL_ARGx()` 宏, 让它们从作为参数的现场 `reg` 中获得正确的系统调用参数寄存器.
3. 添加 `SYS_none` 系统调用.
4. 设置系统调用的返回值.
5. 实现 `popa` 和 `iret` 指令.

重新运行dummy程序, 如果你的实现正确, 你会看到dummy程序又触发了一个号码为 `4` 的系统调用. 查看 `nanos-lite/src/syscall.h`, 你会发现它是一个 `SYS_exit` 系统调用. 这说明之前的 `SYS_none` 已经成功返回, 触发 `SYS_exit` 是因为dummy已经执行完毕, 准备退出了.

你需要实现 `SYS_exit` 系统调用, 它会接收一个退出状态的参数, 用这个参数调用 `_halt()` 即可. 实现成功后, 再次运行dummy程序, 你会看到GOOD TRAP的信息.

需要提醒的是, ASYE还有其它的API, 但我们暂时不会用到, 现在可以先忽略它们.

温馨提示

PA3阶段1到此结束.

在操作系统上运行Hello World

成功运行dummy程序后, 我们已经把系统调用的整个流程都摸清楚了.

标准输出

Navy-apps中提供了一个hello测试程序(`navy-apps/tests/hello`), 它首先通过 `write()` 来输出一句话, 然后通过 `printf()` 来不断输出. 为了运行它, 我们只需要再实现 `SYS_write` 系统调用即可. 根据 `write` 的函数声明(参考 `man 2 write`), 在 `do_syscall()` 中识别出系统调用号是 `SYS_write` 之后, 检查 `fd` 的值, 如果 `fd` 是 `1` 或 `2` (分别代表 `stdout` 和 `stderr`), 则将 `buf` 为首地址的 `len` 字节输出到串口(使用 `_putc()` 即可). 最后还要设置正确的返回值, 否则系统调用的调用者会认为 `write` 没有成功执行, 从而进行重试. 至于 `write` 系统调用的返回值是什么, 请查阅 `man 2 write` . 另外不要忘记在 `navy-apps/libs/libos/src/nanos.c` 的 `_write()` 中调用系统调用接口函数.

事实上, 我们平时使用的 `printf()` , `cout` 这些库函数和库类, 对字符串进行格式化之后, 最终也是通过系统调用进行输出. 这些都是"系统调用封装成库函数"的例子. 系统调用本身对操作系统的各种资源进行了抽象, 但为了给上层的程序员提供更好的接口(beautiful interface), 库函数会再次对部分系统调用再次进行抽象. 例如 `fwrite()` 这个库函数用于往文件中写入数据, 在GNU/Linux中, 它封装了 `write()` 系统调用. 另一方面, 系统调用依赖于具体的操作系统, 因此库函数的封装也提高了程序的可移植性: 在Windows中, `fwrite()` 封装了 `WriteFile()` 系统调用, 如果在代码中直接使用 `WriteFile()` 系统调用, 把代码放到GNU/Linux下编译就会产生链接错误.

并不是所有的库函数都封装了系统调用, 例如 `strcpy()` 这类字符串处理函数就不需要使用系统调用. 从某种程度上来说, 库函数的抽象确实方便了程序员, 使得他们不必关心系统调用的细节.

实现 `SYS_write` 系统调用之后, 我们已经为"使用 `printf()` "扫除了最大的障碍了, 因为 `printf()` 进行字符串格式化之后, 最终会通过 `write()` 系统调用进行输出. 这些工作, Navy-apps中的newlib库已经为我们准备好了.

在Nanos-lite上运行Hello world

实现 `write()` 系统调用, 然后把Nanos-lite上运行的用户程序切换成hello程序并运行:

- 切换到 `navy-apps/tests/hello/` 目录下执行 `make` 编译hello程序
- 修改 `nanos-lite/Makefile` 中ramdisk的生成规则, 把ramdisk中的唯一的文件换成hello程序:

```
--- nanos-lite/Makefile
+++ nanos-lite/Makefile
@@ -9,2 +9,2 @@
OBJCOPY_FLAG = -S --set-section-flags .bss=alloc,contents -O binary
-OBJCOPY_FILE = $(NAVY_HOME)/tests/dummy/build/dummy-x86
+OBJCOPY_FILE = $(NAVY_HOME)/tests/hello/build/hello-x86
```

- 在 `nanos-lite/Makefile` 下执行 `make update` 更新ramdisk
- 重新编译Nanos-lite并运行

堆区管理

如果你在Nanos-lite中的 `sys_write()` 中通过 `Log()` 观察 `write` 系统调用的调用情况, 你会发现用户程序通过 `printf()` 输出的时候是逐个字符地调用 `write` 来输出的. 事实上, 用户程序在第一次调用 `printf()` 的时候会尝试通过 `malloc()` 申请一片缓冲区, 来存放格式化的内容. 若申请失败, 就会逐个字符进行输出.

`malloc()` / `free()` 库函数的作用是在用户程序的堆区中申请/释放一块内存区域. 堆区的使用情况是由libc来进行管理的, 但堆区的大小却需要通过系统调用向操作系统提出更改. 这是因为, 堆区的本质是一片内存区域, 当需要调整堆区大小的时候, 实际上是在调整用户程序可用的内存区域. 事实上, 一个用户程序可用的内存区域要经过操作系统的分配和管理的. 想象一下, 如果一个恶意程序可以不经过操作系统的同意, 就随意使用其它程序的内存区域, 将会引起灾难性的后果. 当然, 目前Nanos-lite只是个单任务操作系统, 不存在多个程序的概念. 在PA4中, 你将会对这个问题有更深刻的认识.

调整堆区大小是通过 `sbrk()` 库函数来实现的, 它的原型是

```
void* sbrk(intptr_t increment);
```

用于将用户程序的program break增长 `increment` 字节, 其中 `increment` 可为负数. 所谓program break, 就是用户程序的数据段(data segment)结束的位置. 我们知道可执行文件里面有代码段和数据段, 链接的时候 `ld` 会默认添加一个名为 `_end` 的符号, 来指示程序的数据段结束的位置. 用户程序开始运行的时候, program break会位于 `_end` 所指示的位置, 意味着此时堆区的大小为0. `malloc()` 被第一次调用的时候, 会通过 `sbrk(0)` 来查询用户程序当前program break的位置, 之后就可以通过后续的 `sbrk()` 调用来动态调整用户程序program break的位置了. 当前program break和和其初始值之间的区间就可以作为用户程序的堆区, 由 `malloc()` / `free()` 进行管理. 注意用户程序不应该直接使用 `sbrk()` , 否则将会扰乱 `malloc()` / `free()` 对堆区的管理记录.

在Navy-apps的Newlib中, `sbrk()` 最终会调用 `_sbrk()` , 它在 `navy-apps/libs/libos/src/nanos.c` 中定义. 框架代码让 `_sbrk()` 总是返回 `-1` , 表示堆区调整失败, 于是 `printf()` 会认为无法在堆区中申请用于格式化的缓冲区, 只好逐个字符地输出. 但如果堆

区总是不可用, Newlib中很多库函数的功能将无法使用, 因此现在你需要实现 `_sbrk()` 了. 为了实现 `_sbrk()` 的功能, 我们还需要提供一个用于设置堆区大小的系统调用. 在GNU/Linux中, 这个系统调用是 `SYS_brk` , 它接收一个参数 `addr` , 用于指示新的program break的位置. `_sbrk()` 通过记录的方式来对用户程序的program break位置进行管理, 其工作方式如下:

1. program break一开始的位置位于 `_end`
2. 被调用时, 根据记录的program break位置和参数 `increment` , 计算出新program break
3. 通过 `SYS_brk` 系统调用来让操作系统设置新program break
4. 若 `SYS_brk` 系统调用成功, 该系统调用会返回 `0` , 此时更新之前记录的program break的位置, 并将旧program break的位置作为 `_sbrk()` 的返回值返回
5. 若该系统调用失败, `_sbrk()` 会返回 `-1`

上述代码是在用户层的库函数中实现的, 我们还需要在Nanos-lite中实现 `SYS_brk` 的功能. 由于目前Nanos-lite还是一个单任务操作系统, 空闲的内存都可以让用户程序自由使用, 因此我们只需要让 `SYS_brk` 系统调用总是返回 `0` 即可, 表示堆区大小的调整总是成功.

实现堆区管理

根据上述内容在Nanos-lite中实现 `SYS_brk` 系统调用, 然后在用户层实现 `_sbrk()` . 你可以通过 `man 2 sbrk` 来查阅libc中 `brk()` 和 `sbrk()` 的行为, 另外通过 `man 3 end` 来查阅如何使用 `_end` 符号.

需要注意的是, 调试的时候不要在 `_sbrk()` 中通过 `printf()` 进行输出, 这是因为 `printf()` 还是会尝试通过 `malloc()` 来申请缓冲区, 最终会再次调用 `_sbrk()` , 造成死递归. 你可以通过 `sprintf()` 先把调试信息输出到一个字符串缓冲区中, 然后通过 `write` 系统调用进行输出.

如果你的实现正确, 你将会在Nanos-lite中看到 `printf()` 将格式化完毕的字符串通过一次 `write` 系统调用进行输出, 而不是逐个字符地进行输出.

缓冲区与系统调用开销

你已经了解系统调用的过程了. 事实上, 如果通过系统调用千辛万苦地陷入操作系统只是为了输出区区一个字符, 那就太不划算了. 于是有了batching的技术: 将一些简单的任务累积起来, 然后再一次性进行处理. 缓冲区是batching技术的核心, libc中的输入输出函数正是通过缓冲区来将输入输出累积起来, 然后再通过一次系统调用进行处理. 例如通过一个1024字节的缓冲区, 就可以通过一次系统调用直接输出1024个字符, 而不需要通过1024次系统调用来逐个字符地输出. 显然, 后者的开销比前者大得多.

有兴趣的同学可以在GNU/Linux上编写相应的程序, 来粗略测试一下一次 `write` 系统调用的开销, 然后和这篇文章对比一下.

简易文件系统

我们的ramdisk已经提供了读写接口, 使得我们可以很方便地访问某一个位置的数据. 目前ramdisk中只有一个文件, 使用起来没什么繁琐的地方. 但如果文件的数量增加之后, 我们就要知道哪个文件在ramdisk的什么位置. 这对Nanos-lite来说貌似没什么困难的地方, 但对用户程序来说, 它怎么知道文件位于ramdisk的哪一个位置呢? 更何况文件会动态地增删, 用户程序并不知情. 这说明, 把ramdisk的读写接口直接提供给用户程序来使用是不可行的. 操作系统还需要在存储介质的驱动程序之上为用户程序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, 我们先讨论普通意义上的文件. 这样, 那些额外的属性就维护了文件到ramdisk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Nanos-lite中实现一个文件系统.

不要被"文件系统"四个字吓到了, 我们对文件系统的需求并不是那么复杂:

- 每个文件的大小是固定的
- 写文件时不允许超过原有文件的大小
- 文件的数量是固定的, 不能创建新文件
- 没有目录

既然文件的数量和大小都是固定的, 我们自然可以把每一个文件分别固定在ramdisk中的某一个位置. 这些简化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.

我们约定文件从ramdisk的最开始一个挨着一个地存放:

```
0
+-------------+---------+----------+-----------+--
|    file0    |  file1  |  ......  |   filen   |
+-------------+---------+----------+-----------+--
 \           / \       /            \         /
  +  size0  +   +size1+              + sizen +
```

为了记录ramdisk中各个文件的名字和大小, 我们还需要一张"文件记录表". Nanos-lite的Makefile已经提供了维护这些信息的脚本, 先对 `nanos-lite/Makefile` 作如下修改:

```
--- nanos-lite/Makefile
+++ nanos-lite/Makefile
@@ -34,2 +34,2 @@
-update: update-ramdisk-objcopy src/syscall.h
+update: update-ramdisk-fsimg src/syscall.h
  @touch src/initrd.S
```

然后运行 `make update` 就会自动编译Navy-apps里面的所有程序, 并把 `navy-apps/fsimg/` 目录下的所有内容整合成ramdisk镜像, 同时生成这个ramdisk镜像的文件记录表 `nanos-lite/src/files.h` . 需要注意的是, 并不是Navy-apps里面的所有程序都能在Nanos-lite上运行,

有些程序需要更多系统调用的支持才能运行, 例如NWM和NTerm, 我们并不打算在PA中运行这些程序.

"文件记录表"其实是一个数组, 数组的每个元素都是一个结构体:

```
typedef struct {
  char *name;          // 文件名
  size_t size;         // 文件大小
  off_t disk_offset;   // 文件在ramdisk中的偏移
} Finfo;
```

在我们的简易文件系统里面, 这三项信息都是固定不变的. 其中的文件名和我们平常使用的习惯不太一样: 由于我们的简易文件系统中没有目录, 我们把目录分隔符 `/` 也认为是文件名的一部分, 例如 `/bin/hello` 是一个完整的文件名. 这种做法其实也隐含了目录的层次结构, 对于文件数量不多的情况, 这种做法既简单又奏效.

有了这些信息, 就已经可以实现最基本的文件读写操作了:

```
ssize_t read(const char *filename, void *buf, size_t len);
ssize_t write(const char *filename, void *buf, size_t len);
```

但在真实的操作系统中, 这种直接用文件名来作为读写操作参数的做法却所有缺陷. 例如, 我们在用 `less` 工具浏览文件的时候:

```
cat file | less
```

`cat` 工具希望把文件内容写到 `less` 工具的标准输入中, 但我们却无法用文件名来标识 `less` 工具的标准输入! 实际上, 操作系统中确实存在不少"没有名字"的文件. 为了统一管理它们, 我们希望通过一个编号来表示文件, 这个编号就是文件描述符(file descriptor). 一个文件描述符对应一个正在打开的文件, 由操作系统来维护文件描述符到具体文件的映射. 于是我们很自然地通过 `open()` 系统调用来打开一个文件, 并返回相应的文件描述符

```
int open(const char *pathname, int flags, int mode);
```

在Nanos-lite中, 由于简易文件系统中的文件数目是固定的, 我们可以简单地把文件记录表的下标作为相应文件的文件描述符返回给用户程序. 在这以后, 所有文件操作都通过文件描述符来标识文件:

```
ssize_t read(int fd, void *buf, size_t len);
ssize_t write(int fd, const void *buf, size_t len);
int close(int fd);
```

另外, 我们也不希望每次读写操作都需要从头开始. 于是我们需要为每一个已经打开的文件引入偏移量属性 `open_offset` , 来记录目前文件操作的位置. 每次对文件读写了多少个字节, 偏移量就前进多少.

```
--- nanos-lite/src/fs.c
+++ nanos-lite/src/fs.c
@@ -3,5 +3,6 @@
 typedef struct {
   char *name;          // 文件名
   size_t size;         // 文件大小
   off_t disk_offset;   // 文件在ramdisk中的偏移
+  off_t open_offset;   // 文件被打开之后的读写指针
 } Finfo;
```

事实上在真正的操作系统中, 把偏移量放在文件记录表中维护会导致用户程序无法实现某些功能. 但解释这个问题需要理解一些超出课程范围的知识, 我们在此就不展开叙述了. 而且由于Nanos-lite是一个精简版的操作系统, 上述问题暂时不会出现, 为了简化实现, 我们还是把偏移量放在文件记录表中进行维护.

偏移量可以通过 `lseek()` 系统调用来调整:

```
off_t lseek(int fd, off_t offset, int whence);
```

为了方便用户程序进行标准输入输出, 操作系统准备了三个默认的文件描述符:

```
#define FD_STDIN 0
#define FD_STDOUT 1
#define FD_STDERR 2
```

它们分别对应标准输入 `stdin` , 标准输出 `stdout` 和标准错误 `stderr` . 我们经常使用的printf, 最终会调用 `write(FD_STDOUT, buf, len)` 进行输出; 而scanf将会通过调用 `read(FD_STDIN, buf, len)` 进行读入.

`nanos-lite/src/fs.c` 中定义的 `file_table` 会包含 `nanos-lite/src/files.h` , 其中前面还有6个特殊的文件, 前三个分别是 `stdin` , `stdout` 和 `stderr` 的占位表项, 它们只是为了保证我们的简易文件系统和约定的标准输入输出的文件描述符保持一致, 例如根据约定 `stdout` 的文件描述符是 `1` , 而我们添加了三个占位表项之后, 文件记录表中的 `1` 号下标也就不会分配给其它的普通文件了. 后面三个是特殊的文件, 我们会在后面来介绍它们, 目前可以先忽略它们.

根据以上信息, 我们就可以在文件系统中实现以下的文件操作了:

```
int fs_open(const char *pathname, int flags, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t 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manual page. 实现这些文件操作的时候注意以下几点:

- 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.
- 为了简化实现, 我们允许所有用户程序都可以对所有已存在的文件进行读写, 这样以后, 我们在实现 `fs_open()` 的时候就可以忽略 `flags` 和 `mode` 了.
- 使用 `ramdisk_read()` 和 `ramdisk_write()` 来进行文件的真正读写.
- 由于文件的大小是固定的, 在实现 `fs_read()` , `fs_write()` 和 `fs_lseek()` 的时候, 注意偏移量不要越过文件的边界.
- 除了写入 `stdout` 和 `stderr` 之外(用 `_putc()` 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.
- 由于我们的简易文件系统没有维护文件打开的状态, `fs_close()` 可以直接返回 `0` , 表示总是关闭成功.

最后你还需要在Nanos-lite和Navy-apps的libos中添加相应的系统调用, 来调用相应的文件操作.

让loader使用文件

我们之前是让loader来直接调用 `ramdisk_read()` 来加载用户程序. ramdisk中的文件数量增加之后, 这种方式就不合适了, 我们首先需要让loader享受到文件系统的便利.

你需要先实现 `fs_open()` , `fs_read()` 和 `fs_close()` , 这样就可以在loader中使用文件名来指定加载的程序了, 例如"/bin/hello". 我们还需要让 `fs_read()` 知道文件的大小, 我们可以在文件系统中添加一个辅助函数

```
size_t fs_filesz(int fd);
```

它用于返回文件描述符 `fd` 所描述的文件的大小.

实现之后, 以后更换用户程序只需要修改传入 `loader()` 函数的文件名即可, 无需更新ramdisk的内容(除非ramdisk上的内容确实需要更新, 例如重新编译了Navy-apps的程序).

实现完整的文件系统

实现 `fs_write()` 和 `fs_lseek()`, 然后运行测试程序 `/bin/text`. 这个测试程序用于进行一些简单的文件读写和定位操作. 如果你的实现正确, 你将会看到程序输出 `PASS!!!` 的信息.

温馨提示

PA3阶段2到此结束.

一切皆文件

我们已经提供了完整的文件系统, 用户程序已经可以读写普通的文件了. 想想我们在AM上运行的打字游戏, 读入按键/查询时钟/更新屏幕其实也是用户程序的合理需求, 操作系统也需要提供支持. 一种最直接的方式, 就是为每个功能单独提供一个系统调用, 用户程序通过这些系统调用, 就可以直接使用相应的功能了. 然而这种做法却存在不少问题:

- 首先, 设备的类型五花八门, 其功能更是数不胜数, 要为它们分别实现系统调用来给用户程序提供接口, 本身就已经缺乏可行性了;
- 此外, 由于设备的功能差别较大, 若提供的接口不能统一, 程序之间的交互就会变得困难.

我们在上一小节中提到, 文件的本质就是字节序列. 事实上, 计算机系统中到处都是字节序列(如果只是无序的字节集合, 计算机要如何处理?), 我们可以轻松地举出很多例子:

- 内存是以字节编址的, 天然就是一个字节序列, 因而我们之前使用的ramdisk作为字节序列也更加显而易见了
- 管道(shell命令中的 `|`)是一种先进先出的字节序列, 本质上它是内存中的一个队列缓冲区
- 磁盘也可以看成一个字节序列: 我们可以为磁盘上的每一个字节进行编号, 例如第x柱面第y磁头第z扇区中的第n字节, 把磁盘上的所有字节按照编号的大小进行排列, 便得到了一个字节序列
- socket(网络套接字)也是一种字节序列, 它有一个缓冲区, 负责存放接收到的网络数据包, 上层应用将socket中的内容看做是字节序列, 并通过一些特殊的文件操作来处理它们
- 操作系统的一些信息可以以字节序列的方式暴露给用户, 例如CPU的配置信息
- 操作系统提供的一些特殊的功能, 如随机数生成器, 也可以看成一个无穷长的字节序列
- 甚至一些非存储类型的硬件也可以看成是字节序列: 我们在键盘上按顺序敲入按键的编码形成了一个字节序列, 显示器上每一个像素的内容按照其顺序也可以看做是字节序列...

既然文件就是字节序列, 那很自然地, 上面这些五花八门的字节序列应该都可以看成文件. Unix就是这样做的, 因此有"一切皆文件"(Everything is a file)的说法. 这种做法最直观的好处就是为不同的事物提供了统一的接口: 我们可以使用文件的接口来操作计算机上的一切, 而不必对它们进行详细的区分: 例如 `nanos-lite/Makefile` 中通过管道把各个shell工具的输入输出连起来, 生成文件记录表

```
wc -c $(FSIMG_FILES) | grep -v 'total$$' | sed -e 's+ $(FSIMG_PATH)+ +' |
  awk -v sum=0 '{print "\x7b\x22" $$2 "\x22\x2c " $$1 "\x2c " sum "\x7d\x2c";sum += $$
1}' > src/files.h
```

以十六进制的方式查看磁盘上的内容

```
head -c 512 /dev/sda | hd
```

查看CPU的配置信息

```
cat /proc/cpuinfo | vim -
```

而

```
#include "/dev/urandom"
```

则会将urandom中的内容包含到源文件中: 由于urandom是一个长度无穷的字节序列, 提交一个包含上述内容的程序源文件将会令一些检测功能不强的Online Judge平台直接崩溃.

"一切皆文件"的抽象使得我们可以通过标准工具很容易完成一些在Windows下不易完成的工作, 这其实体现了Unix哲学的部分内容: 每个程序采用文本文件作为输入输出, 这样可以使程序之间易于合作. GNU/Linux继承自Unix, 也自然继承了这种优秀的特性. 为了向用户程序提供统一的抽象, Nanos-lite也尝试将IOE抽象成文件.

首先当然是来看输出设备. 串口已经被抽象成 `stdout` 和 `stderr` 了, 我们无需担心. 至于VGA, 程序为了更新屏幕, 只需要将像素信息写入VGA的显存即可. 于是, Nanos-lite需要做的, 便是把显存抽象成文件. 显存本身也是一段存储空间, 它以行优先的方式存储了将要在屏幕上显示的像素. Nanos-lite和Navy-apps约定, 把显存抽象成文件 `/dev/fb` (fb为frame buffer之意), 它需要支持写操作和lseek, 以便于用户程序把像素更新到屏幕的指定位置上.

除此之外, 用户程序还需要获得屏幕大小的信息, 然后才能决定如何更好地显示像素内容. Nanos-lite和Navy-apps约定, 屏幕大小的信息通过 `/proc/dispinfo` 文件来获得, 它需要支持读操作. `/proc/dispinfo` 内容的一个例子如下:

```
WIDTH:640
HEIGHT:480
```

需要注意的是, `/dev/fb` 和 `/proc/dispinfo` 都是特殊的文件, 文件记录表中有它们的文件名, 但它们的实体并不在ramdisk中. 因此, 我们需要在 `fs_read()` 和 `fs_write()` 的实现中对它们进行"重定向", 以 `fs_write()` 为例:

```
ssize_t fs_write(int fd const void *buf, size_t len) {
  // ...
  switch (fd) {
    case FD_STDOUT:
    case FD_STDERR:
      // call _putc()
      break;

    case FD_FB:
      // write to frame buffer
      break;

    default:
      // write to ramdisk
      break;
}
```

把VGA显存抽象成文件

你需要在Nanos-lite中

- 在 `init_fs()` (在 `nanos-lite/src/fs.c` 中定义)中对文件记录表中 `/dev/fb` 的大小进行初始化, 你需要使用IOE定义的API来获取屏幕的大小.
- 实现 `fb_write()` (在 `nanos-lite/src/device.c` 中定义), 用于把 `buf` 中的 `len` 字节写到屏幕上 `offset` 处. 你需要先从 `offset` 计算出屏幕上的坐标, 然后调用IOE的 `_draw_rect()` 接口.
- 在 `init_device()` (在 `nanos-lite/src/device.c` 中定义)中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中. 实际的屏幕大小信息已经记录在AM的IOE接口中, 你需要在Nanos-lite中获取它们.
- 实现 `dispinfo_read()` (在 `nanos-lite/src/device.c` 中定义), 用于把字符串 `dispinfo` 中 `offset` 开始的 `len` 字节写到 `buf` 中.
- 在文件系统中添加对 `/dev/fb` 和 `/proc/dispinfo` 这两个特殊文件的支持.

让Nanos-lite加载 `/bin/bmptest` , 如果实现正确, 你将会看到屏幕上显示ProjectN的Logo.

最后我们来看输入设备. 输入设备有键盘和时钟, 我们需要把它们的输入包装成事件. 一种简单的方式是把事件以文本的形式表现出来, 我们定义以下事件, 一个事件以换行符 `\n` 结束:

- `t 1234` : 返回系统启动后的时间, 单位为毫秒;
- `kd RETURN` / `ku A` : 按下/松开按键, 按键名称全部大写, 使用AM中定义的按键名

我们采用文本形式来描述事件有两个好处, 首先文本显然是一种字节序列, 这使得事件很容易抽象成文件; 此外文本方式使得用户程序可以容易可读地解析事件的内容. Nanos-lite和Navy-apps约定, 上述事件抽象成文件 `/dev/events` , 它需要支持读操作, 用户程序可以从中一次读出

一个输入事件. 需要注意的是, 由于时钟事件可以任意时刻进行读取, 我们需要优先处理按键事件, 当不存在按键事件的时候, 才返回时钟事件, 否则用户程序将永远无法读到按键事件.

Bug说明

`nexus-am/libs/klib/build/klib-x86-nemu.a` 中的 `sprintf()` 返回结果字符串长度时额外计算了末尾的 `\0` , 与 `man sprintf` 中的说明不符. 可以使用 `strlen()` 来计算结果字符串的长度来避免这个问题.

把设备输入抽象成文件

你需要在Nanos-lite中

- 实现 `events_read()` (在 `nanos-lite/src/device.c` 中定义), 把事件写入到 `buf` 中, 最长写入 `len` 字节, 然后返回写入的实际长度. 其中按键名已经在字符串数组 `names` 中定义好了. 你需要借助IOE的API来获得设备的输入.
- 在文件系统中添加对 `/dev/events` 的支持.

让Nanos-lite加载 `/bin/events` , 如果实现正确, 你会看到程序输出时间事件的信息, 敲击按键时会输出按键事件的信息.

运行仙剑奇侠传

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, 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.

我们的框架代码已经把SDLPAL移植到Navy-apps中了. 移植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Navy-apps提供的运行时环境重写它们. 主要包括以下三部分内容:

- C标准库
- 浮点数
- SDL库

Navy-apps中的newlib已经提供了C标准库的功能, 我们无需额外移植. 关于浮点数的移植工作, 我们会在PA5中再来讨论, 目前先忽略它. 为了移植SDL库相关的代码, Navy-apps把时钟, 键盘, 显示的功能封装成NDL(NJU DirectMedia Layer)多媒体库, 其中封装了我们之前实现的 `/dev/fb` 和 `/dev/events` 的读写. 为了用NDL的API来替代原来SDL的相应功能, 移植工作需

要对SDLPAL进行了少量修改, 包括去掉了声音, 修改了和按键相关的处理, 把我们关心的与NDL相关的功能整理到 `hal/hal.c` 中, 一些我们不必关心的实现则整理到 `unused/` 目录下. 框架代码已经把这些移植工作都做好了, 目前你不需要编写额外的代码来进行移植.

在NEMU中运行仙剑奇侠传

终于到了激动人心的时刻了! 我们已经通过文件的抽象向仙剑奇侠传提供了所有它需要的功能了. 从提交网站上下载仙剑奇侠传的数据文件, 并放到 `navy-apps/fsimg/share/games/pal/` 目录下, 更新ramdisk之后, 在Nanos-lite中加载并运行 `/bin/pal` .

在我们提供的数据文件中包含一些游戏存档, 可以读取迷宫中的存档, 与怪物进行战斗. 但战斗需要进行一些浮点数相关的计算, 而NEMU目前没有实现浮点数, 因而不能成功进行战斗. 我们会在PA5中再来解决浮点数的问题, 目前我们先暂时不触发战斗, 可以先通过"新的故事"进行游戏.

不再神秘的秘技

网上流传着一些关于仙剑奇侠传的秘技, 其中的若干条秘技如下:

1. 很多人到了云姨那里都会去拿三次钱, 其实拿一次就会让钱箱爆满! 你拿了一次钱就去买剑把钱用到只剩一千多, 然后去道士那里, 先不要上楼, 去掌柜那里买酒, 多买几次你就会发现钱用不完了.
2. 不断使用乾坤一掷(钱必须多于五千文)用到财产低于五千文, 钱会暴增到上限, 如此一来就有用不完的钱了

3. 当李逍遥等级到达99级时, 用5~10只金蚕王, 经验点又跑出来了, 而且升级所需经验会变回初期5~10级内的经验值, 然后去打敌人或用金蚕王升级, 可以学到灵儿的法术(从五气朝元开始); 升到199级后再用5~10只金蚕王, 经验点再跑出来, 所需升级经验也是很低, 可以学到月如的法术(从一阳指开始); 到299级后再用10~30只金蚕王, 经验点出来后继续升级, 可学到阿奴的法术(从万蚁蚀象开始).

假设这些上述这些秘技并非游戏制作人员的本意, 请尝试解释这些秘技为什么能生效.

必答题

文件读写的具体过程 仙剑奇侠传中有以下行为:

- 在 `navy-apps/apps/pal/src/global/global.c` 的 `PAL_LoadGame()` 中通过 `fread()` 读取游戏存档
- 在 `navy-apps/apps/pal/src/hal/hal.c` 的 `redraw()` 中通过 `NDL_DrawRect()` 更新屏幕

请结合代码解释仙剑奇侠传, 库函数, libos, Nanos-lite, AM, NEMU是如何相互协助, 来分别完成游戏存档的读取和屏幕的更新.

温馨提示

PA3到此结束. 请你编写好实验报告(不要忘记在实验报告中回答必答题), 然后把命名为 `学号.pdf` 的实验报告文件放置在工程目录下, 执行 `make submit` 对工程进行打包, 最后将压缩包提交到指定网站.

PA4 - 虚实交错的魔法: 分时多任务

世界诞生的故事 - 第四章

先驱已经创造了一个足够强大的计算机, 甚至能支撑操作系统和真实应用程序的运行. 但这还不够, 先驱决定向计算机施以虚拟化的魔法.

在进行本PA前, 请在工程目录下执行以下命令进行分支整理, 否则将影响你的成绩:

```
git commit --allow-empty -am "before starting pa4"
git checkout master
git merge pa3
git checkout -b pa4
```

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 30小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 实现分页机制 - 2017/12/17 23:59:59
- 阶段2: 实现上下文切换 - 2017/12/24 23:59:59
- 最后阶段: 实现真正的分时多任务, 并提交完整的实验报告 - 2017/12/31 23:59:59

提交说明: 见这里

虚拟地址空间

通过Nanos-lite的支撑, 我们已经在NEMU中成功把仙剑奇侠传跑起来了! 这说明我们亲自构建的NEMU这个看似简单的机器, 同样能支撑眞实程序的运行, 丝毫不逊色于眞实的机器! 不过, 我们目前还是只能在这个机器上同时运行一个程序, 这是因为Nanos-lite目前还只是一个单任务的操作系统. 那为了同时运行多个程序, 我们的NEMU和Nanos-lite还缺少些什么呢?

我们知道, 现在的计算机可以"同时"运行多个进程. 这里的"同时"其实只是一种假象, 并不是指在物理时间上的重叠, 而是操作系统很快地在不同的进程之间来回切换. 切换的频率大约是10ms一次, 一般的用户是感觉不到的. 而让多个进程"同时"运行的一个基本条件, 就是不同的进程要拥有独立的存储空间, 它们之间不能相互干扰.

一个很自然的想法, 就是让操作系统的loader直接把不同的程序加载到不同的内存位置就可以了. 我们在PA3中提到操作系统有管理系统资源的义务, 在多任务操作系统中, 内存作为一种资源自然也是要被管理起来: 操作系统需要记录内存的分配情况, 需要运行一个新程序的时候, 就给它分配一片空闲的内存位置, 把它加载到这一内存位置上即可.

这个方法听上去很可靠, 但对程序来说就不是这么简单了. 回想我们编译Navy-apps中的程序时, 我们都把它们链接到0x4000000的内存位置. 这意味着, 如果我们正在运行仙剑奇侠传, 同时也想运行hello程序, 仙剑奇侠传的内容将会被hello程序所覆盖! 最后的结果是, 仙剑奇侠传无法正确运行, 从而也无法实现"多个程序同时运行"的美好愿望.

或者, 我们可以尝试把不同的程序链接到不同的内存位置. 然而新问题又来了, 我们在编译链接的时候, 怎么能保证程序将来运行的时候它所用到的内存位置是空闲的呢? 况且, 我们还希望一个程序能同时运行多个进程实例, 例如在浏览器中同时打开多个页面浏览不同的网页. 这是多么合理的需求啊! 然而这种方式却没法实现.

所以如果要解决这个问题, 我们的方法就需要满足一个条件: 在程序被加载之前, 我们不能对程序被加载到的内存位置有任何提前的假设. 很自然地, 为了实现多任务, 我们必须在系统栈的某些层次满足这个条件.

一种方式是从程序本身的性质入手. 事实上, 编译器可以编译出PIC(position-independent code, 位置无关代码). 所谓PIC, 就是程序本身的代码不对将来的运行位置进行任何假设, 这样的程序可以被加载到任意内存位置也能正确运行. PIC程序不仅具有这一灵活的特性, 还能在一定程度上对恶意的攻击程序造成了干扰: 恶意程序也无法提前假设PIC程序运行的地址. 也正是因为这一安全相关的特性, 最近的不少GNU/Linux的发行版上配置的gcc都默认生成PIC程序. 多神奇的功能啊! 然而, 天下并没有免费的午餐, PIC程序之所以能做到位置无关, 其实是要依赖于程序中一个叫GOT(global offset table, 全局偏移量表)的数据结构. 要正确运行PIC程序, 操作系统中的动态加载器需要在加载程序的时候往GOT中填写正确的内容. 但是, 先不说GOT具体如何填写, 目前Nanos-lite中的loader是个raw program loader, 它无法在可执行文件中找到GOT的位置. 因此, 在Nanos-lite上运行PIC程序目前并不是一个可行的方案.

我们要寻求另一种解决方案了. 既然我们无法运行PIC程序, 我们还是只能让程序链接到一个固定的内存位置. 问题貌似又回到原点了. 我们来仔细琢磨一下我们的需求: 我们需要在让程序认为自己在某个固定的内存位置的同时, 把程序加载到不同的内存位置去执行. 这个看似自相矛盾的需求, 其实里面正好蕴藏着那深刻的思想. 说是自相矛盾, 是因为思维定势会让我们觉得, "固定的内存位置"和"不同的内存位置"必定无法同时满足; 说是蕴藏着深刻的思想, 我们不妨换一个角度来想想, 如果这两个所谓的"内存位置"并不是同一个概念呢?

为了让这个问题的肯定回答成为可能, 虚拟内存的概念就诞生了. 所谓虚拟内存, 就是在真正的内存(也叫物理内存)之上的一层专门给程序使用的抽象. 有了虚拟内存之后, 程序只需要认为自己运行在虚拟地址上就可以了, 真正运行的时候, 才把虚拟地址映射到物理地址. 这样, 我们只要把程序链接到一个固定的虚拟地址, 加载程序的时候把它们加载到不同的物理地址, 并维护好虚拟地址到物理地址的映射关系, 就可以实现我们那个看似不可能的需求了!

绝大部分多任务操作系统就是这样做的. 不过在讨论具体的虚拟内存机制之前, 我们先来探讨最关键的一个问题: 程序运行的时候, 谁来把虚拟地址映射成物理地址呢? 我们在PA1中已经了解到指令的生命周期:

```
while (1) {
  从EIP指示的存储器位置取出指令;
  执行指令;
  更新EIP;
}
```

如果引入了虚拟内存机制, EIP就是一个虚拟地址了, 我们需要在访问存储器之前完成虚拟地址到物理地址的映射. 尽管操作系统管理着计算机中的所有资源, 在计算机看来它也只是一个程序而已. 作为一个在计算机上执行的程序而言, 操作系统不可能有能力干涉指令执行的具体过程. 所以让操作系统来把虚拟地址映射成物理地址, 是不可能实现的. 因此, 在硬件中进行这一映射是唯一的选择了: 我们在处理器和存储器之间添加一个新的硬件模块MMU(Memory Management Unit, 内存管理单元), 它是虚拟内存机制的核心, 肩负起这一机制最重要的地址映射功能. 需要说明的是, 我们刚才提到的"MMU位于处理器和存储器之间"只是概念上的说法. 事实上, 虚拟内存机制在现代计算机中是如此重要, 以至于MMU在物理上都实现在处理器芯片内部了.

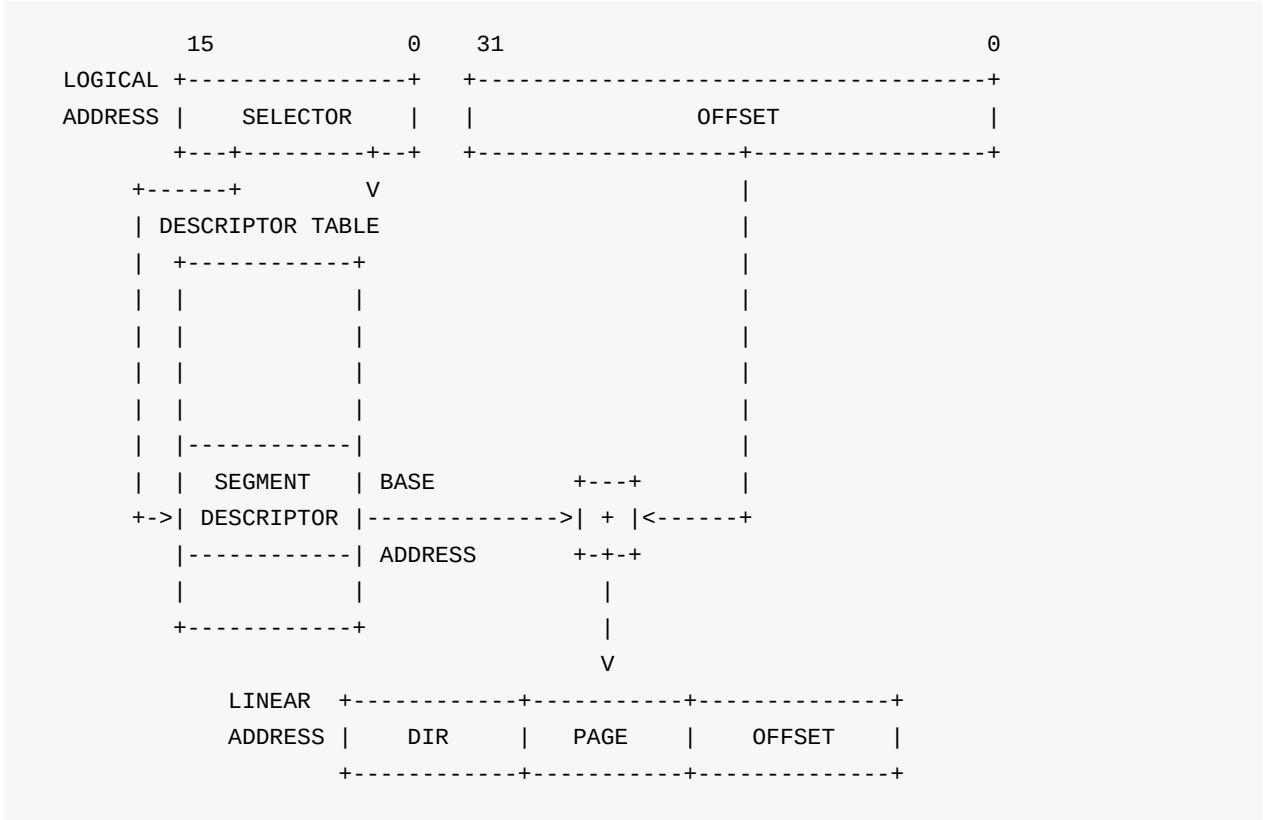
但是, 只有操作系统才知道具体要把虚拟地址映射到哪些物理地址上. 所以, 虚拟内存机制是一个软硬协同才能生效的机制: 操作系统负责进行物理内存的管理, 加载程序的时候决定要把程序的虚拟地址映射到那些物理地址; 等到程序真正运行之前, 还需要配置MMU, 把之前决定好的映射落实到硬件上, 程序运行的时候, MMU就会进行地址转换, 把程序的虚拟地址映射到操作系统希望的物理地址.

分段

关于MMU具体如何进行地址映射, 目前主要有两种主流的方式. 最简单的方法就是, 物理地址=虚拟地址+偏移量. 这种最朴素的方式就是段式虚拟内存管理机制, 简称分段机制. 直觉上来理解, 就是把物理内存划分成若干个段, 不同的程序就放到不同的段中运行, 程序不需要关心自己具体在哪一个段里面, 操作系统只要让不同的程序使用不同的偏移量, 程序之间就不会相互干扰了.

分段机制在硬件上的实现可以非常简单, 只需要在MMU中实现一个段基址寄存器就可以了. 操作系统在运行不同程序的时候, 就在段基址寄存器中设置不同的值, MMU会把程序使用的虚拟地址加上段基址, 来生成真正用于访问内存的物理地址, 这样就实现了"让不同的程序使用不同的段"的目的. 作为教学操作系统的Minix就是这样工作的.

实际上, 处理器中的分段机制有可能复杂得多. 例如i386为了兼容它的前身8086, 引入了段描述符, 段选择符, 全局描述符表(GDT), 全局描述符表寄存器(GDTR)等概念, 段描述符中除了段基址之外, 还描述了段的长度, 类型, 粒度, 访问权限等等的属性, 为了弥补段描述符的性能问题, 又加入了描述符cache等概念... 我们可以目睹一下i386分段机制的风采:



咋看之下真是眼花缭乱, 让人一头雾水.

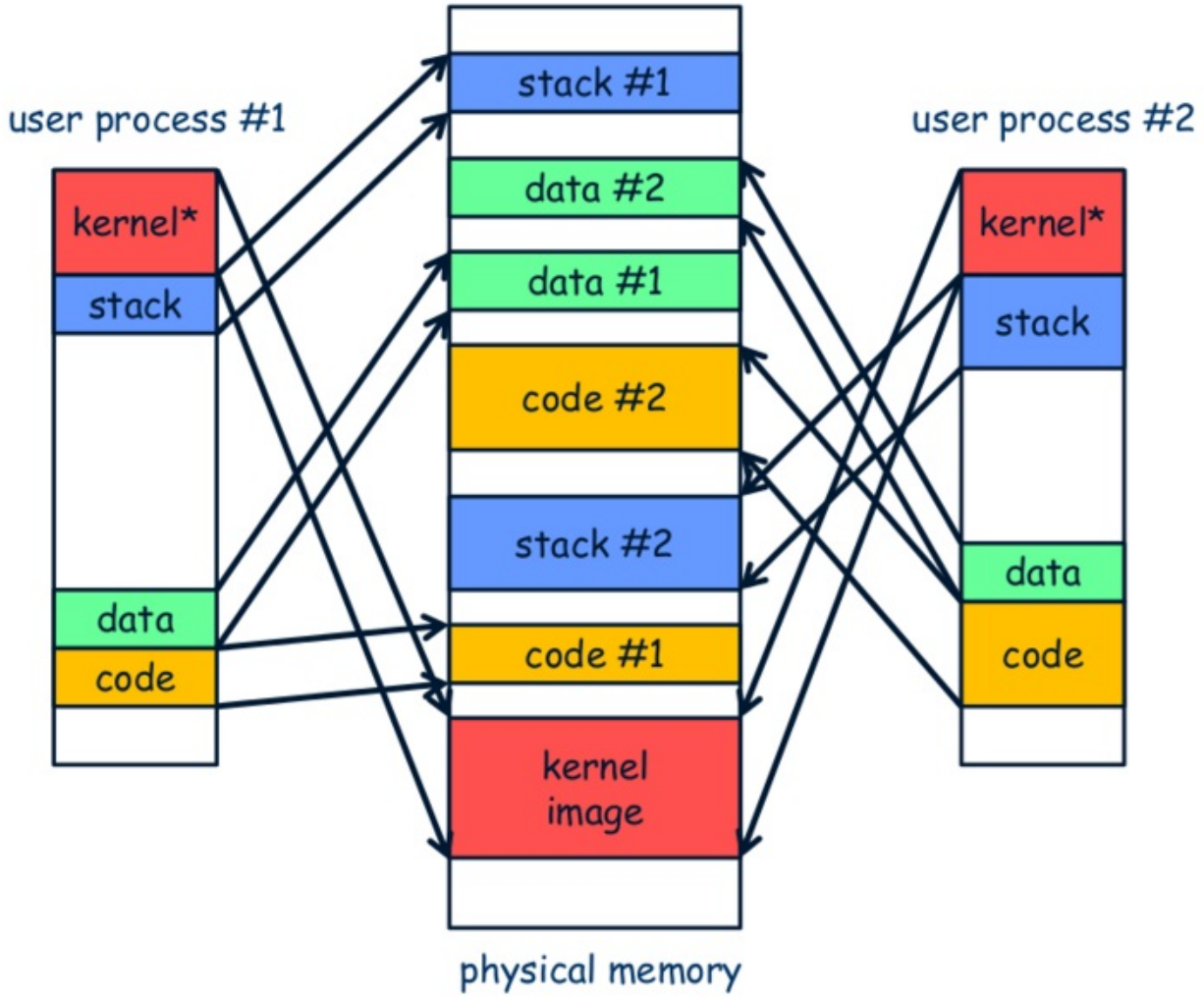
在NEMU中, 我们需要了解什么呢? 什么都不需要. 现在的绝大部分操作系统都不再使用分段机制, 就连i386手册中也提到可以想办法"绕过"它来提高性能: 将段基地址设成0, 长度设成4GB, 这样看来就像没有段的概念一样, 这就是i386手册中提到的"扁平模式". 当然, 这里的"绕过"并不是简单地将分段机制关掉(事实上也不可能关掉), 我们在PA3中提到的i386保护机制中关于特权级的概念, 其实就是i386分段机制提供的, 抛弃它是十分不明智的. 不过我们在NEMU中也没打算实现保护机制, 因此i386分段机制的各种概念, 我们也不会加入到NEMU中来.

超越容量的界限

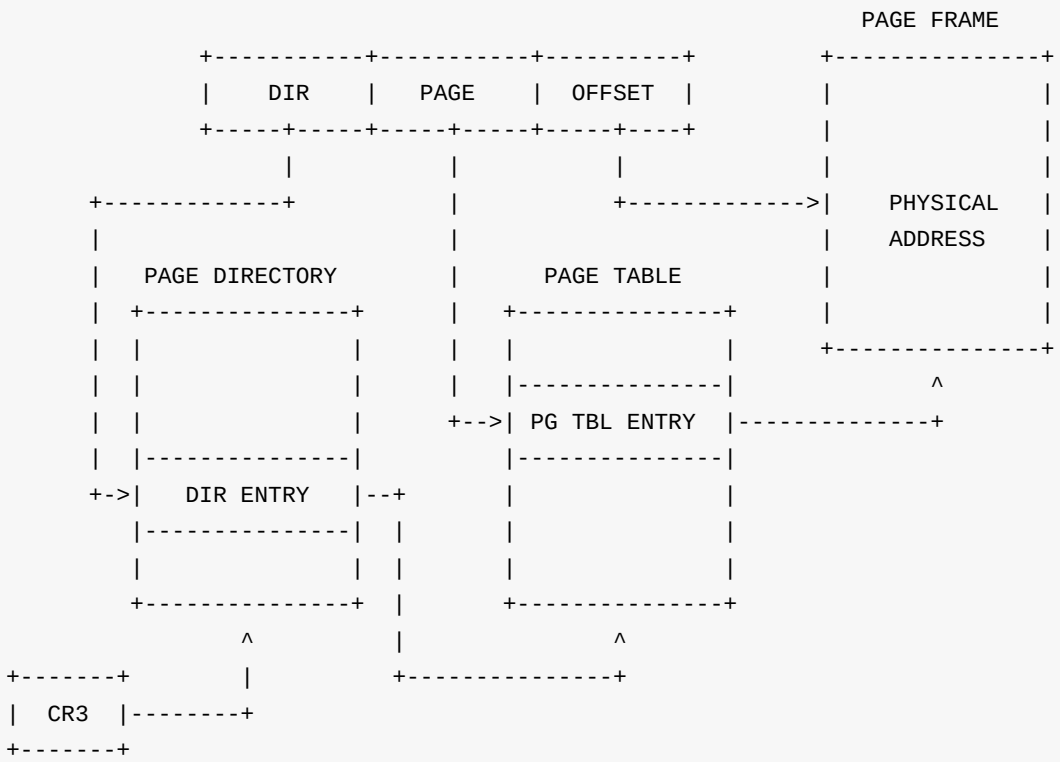
现代操作系统不使用分段还是有一定的道理的. 有研究表明, Google数据中心中的1000台服务器在7分钟内就运行了上千个不同的程序, 其中有的是巨大无比的家伙(Google内部开发程序的时候为了避免不同计算机上的动态库不兼容的问题, 用到的所有库都以静态链接的方式成为程序的一部分, 光是程序的代码段就有几百MB甚至上GB的大小, 感兴趣的同学可以阅读这篇文章), 有的只是一些很小的测试程序. 让这些特征各异的程序都占用连续的存储空间并不见得有什么好处: 那些巨大无比的家伙们在一次运行当中只会触碰到很小部分的代码, 其实没有必要分配那么多内存把它们全部加载进来; 另一方面, 小程序运行结束之后, 它占用的存储空间就算被释放了, 也很容易成为"碎片空洞" - 只有比它更小的程序才能把碎片空洞用起来. 分段机制的简单朴素, 在现实情况中也许要付出巨大的代价.

事实上, 我们需要一种按需分配的虚存管理机制. 之所以分段机制不好实现按需分配, 就是因为段的粒度太大了, 为了实现这一目标, 我们需要反其道而行之: 把连续的存储空间分割成小片段, 以这些小片段为单位进行组织, 分配和管理. 这正是分页机制的核心思想.

在分页机制中, 这些小片段称为页面, 在虚拟地址空间和物理地址空间中也分别称为虚拟页和物理页. 分页机制做的事情, 就是把一个个的虚拟页分别映射到相应的物理页上. 显然, 这一映射关系并不像分段机制中只需要一个段基址寄存器就可以描述的那么简单. 分页机制引入了一个叫"页表"的结构, 页表中的每一个表项记录了一个虚拟页到物理页的映射关系, 来把不必连续的页面重新组织成连续的虚拟地址空间. 因此, 为了让分页机制支撑多任务操作系统的运行, 操作系统首先需要以物理页为单位对内存进行管理. 每当加载程序的时候, 就给程序分配相应的物理页(注意这些物理页之间不必连续), 并为程序准备一个新的页表, 在页表中填写程序用到的虚拟页到分配到的物理页的映射关系. 等到程序运行的时候, 操作系统就把之前为这个程序填写好的页表设置到MMU中, MMU就会根据页表的内容进行地址转换, 把程序的虚拟地址空间映射到操作系统所希望的物理地址空间上.



i386是x86史上首次引进分页机制的处理器, 它把物理内存划分成以4KB为单位的页面, 同时也采用了二级页表的结构. 为了方便叙述, i386给第一级页表取了个新名字叫"页目录". 虽然听上去很厉害, 但其实原理都是一样的. 每一张页目录和页表都有1024个表项, 每个表项的大小都是4字节, 除了包含页表(或者物理页)的基地址, 还包含一些标志位信息. 因此, 一张页目录或页表的大小是4KB, 要放在寄存器中是不可能的, 因此它们要放在内存中. 为了找到页目录, i386提供了一个CR3(control register 3)寄存器, 专门用于存放页目录的基地址. 这样, 页级地址转换就从CR3开始一步一步地进行, 最终将虚拟地址转换成真正的物理地址, 这个过程称为一次page walk.



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

一些问题

- i386不是一个32位的处理器吗, 为什么表项中的基地址信息只有20位, 而不是32位?
- 手册上提到表项(包括CR3)中的基地址都是物理地址, 物理地址是必须的吗? 能否使用虚拟地址?
- 为什么不采用一级页表? 或者说采用一级页表会有什么缺点?

页级转换的过程并不总是成功的, 因为i386也提供了页级保护机制, 实现保护功能就要靠表项中的标志位了. 我们对一些标志位作简单的解释:

- present位表示物理页是否可用, 不可用的时候又分两种情况:
  1. 物理页面由于交换技术被交换到磁盘中了, 这就是你在课堂上最熟悉的Page fault的情况之一了, 这时候可以通知操作系统内核将目标页面换回来, 这样就能继续执行了
  2. 进程试图访问一个未映射的线性地址, 并没有实际的物理页与之相对应, 因此这就是一个非法操作咯
- R/W位表示物理页是否可写, 如果对一个只读页面进行写操作, 就会被判定为非法操作
- U/S位表示访问物理页所需要的权限, 如果一个ring 3的进程尝试访问一个ring 0的页面, 当然也会被判定为非法操作

空指针真的是"空"的吗?

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

和分段机制相比，分页机制更灵活，甚至可以使用超越物理地址上限的虚拟地址. 现在我们从数学的角度来理解这两点. 撇去存储保护机制不谈，我们可以把这分段和分页的过程分别抽象成两个数学函数：

```
y = seg(x) = seg.base + x
y = page(x)
```

可以看到，`seg()`函数只不过是做加法. 如果仅仅使用分段机制，我们还要求段级地址转换的结果不能超过物理地址上限：

```
   y = seg(x) = seg.base + x < PMEM_MAX
=> x < PMEM_MAX - seg.base
=> x <= PMEM_MAX
```

我们可以得出这样的结论：仅仅使用分段机制，虚拟地址是无法超过物理地址上限的. 而分页机制就不一样了，我们无法给出`page()`具体的解析式，是因为填写页目录和页表实际上就是在用枚举自变量的方式定义`page()`函数，这就是分页机制比分段机制灵活的根本原因. 虽然"页级地址转换结果不能超过物理地址上限"的约束仍然存在，但我们只要保证每一个函数值都不超过物理地址上限即可，并没有对自变量的取值作明显的限制，当然自变量本身也就可以比函数值还大. 这就已经把分页的"灵活"和"允许使用超过物理地址上限"这两点特性都呈现出来了.

i386采用段页式存储管理机制. 不过仔细想想，这只不过是把分段和分页结合起来罢了，用数学函数来理解，也只不过是个复合函数：

```
paddr = page(seg(swaddr))
```

而"虚拟地址空间"和"物理地址空间"这两个在操作系统中无比重要的概念，也只不过是这个复合函数的定义域和值域而已.

最后，支持分页机制的处理器能识别什么是页表吗？我们以一个页面大小为1KB的一级页表的地址转换例子来说明这个问题：

```
pa = (pg_table[va >> 10] & ~0x3ff) | (va & 0x3ff);
```

可以看到，处理器并没有表的概念：地址转换的过程只不过是一些访存和位操作而已. 这再次向我们展示了计算机的本质：一堆美妙的，蕴含着深刻数学道理和工程原理的... 门电路！然而这些小小的门电路操作却成为了今天多任务操作系统的基础，支撑着千千万万程序的运行，真不愧

是人类的文明.

加入PTE

在AM的模型中, 由PTE模块来负责提供存储保护的能力. 为了在Nanos-lite中实现一个多任务操作系统, 我们需要在NEMU和AM中添加PTE的支持. 我们的第一个目标是首先让仙剑奇侠传运行在分页机制上, 然后再考虑多任务的支持.

准备内核页表

由于页表位于内存中, 但计算机启动的时候, 内存中并没有有效的数据, 因此我们不可能让计算机启动的时候就开启分页机制. 操作系统为了启动分页机制, 首先需要准备一些内核页表. 框架代码已经为我们实现好这一功能了(见 `nexus-am/am/arch/x86-nemu/src/pte.c` 的 `_pte_init()` 函数). 只需要在 `nanos-lite/src/main.c` 中定义宏 `HAS_PTE` , Nanos-lite在初始化的时候首先就会调用 `init_mm()` 函数(在 `nanos-lite/src/mm.c` 中定义)来初始化MM. 这里的MM是指存储管理器(Memory Manager)模块, 它专门负责分页相关的存储管理.

目前初始化MM的工作有两项, 第一项工作是将TRM提供的堆区起始地址作为空闲物理页的首地址, 将来会通过 `new_page()` 函数来分配空闲的物理页. 第二项工作是调用AM的 `_pte_init()` 函数, 填写内核的页目录和页表, 然后设置CR3寄存器, 最后通过设置CR0寄存器来开启分页机制. 这样以后, Nanos-lite就运行在分页机制之上了. 调用 `_pte_init()` 函数的时候还需要提供物理页的分配和回收两个回调函数, 用于在AM中获取/释放物理页. 为了简化实现, MM中采用顺序的方式对物理页进行分配, 而且分配后无需回收.

为了在NEMU中实现分页机制, 你需要添加CR3寄存器和CR0寄存器, 以及相应的操作它们的指令. 对于CR0寄存器, 我们只需要实现PG位即可. 如果发现CR0的PG位为1, 则开启分页机制, 从此所有虚拟地址的访问(包括 `vaddr_read()` , `vaddr_write()`)都需要经过分页地址转换. 为了让differential testing机制正确工作, 在 `restart()` 函数中我们需要对CR0寄存器初始化为 `0x60000011` , 但我们不必关心其含义.

然后你需要对 `vaddr_read()` 和 `vaddr_write()` 函数作少量修改. 以 `vaddr_read()` 为例, 修改后如下:

```
uint32_t vaddr_read(vaddr_t addr, int len) {
    if (data cross the page boundary) {
        /* this is a special case, you can handle it later. */
        assert(0);
    }
    else {
        paddr_t paddr = page_translate(addr);
        return paddr_read(paddr, len);
    }
}
```

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

最后提醒一下页级地址转换时出现的一种特殊情况. 由于i386并没有严格要求数据对齐, 因此可能会出现数据跨越虚拟页边界的情况, 例如一条很长的指令的首字节在一个虚拟页的最后, 剩下的字节在另一个虚拟页的开头. 如果这两个虚拟页被映射到两个不连续的物理页, 就需要进行两次页级地址转换, 分别读出这两个物理页中需要的字节, 然后拼接起来组成一个完成的数据返回. MIPS作为一种RISC架构, 指令和数据都严格按照4字节对齐, 因此不会发生这样的情况, 否则MIPS CPU将会抛出异常, 可见软件灵活性和硬件复杂度是计算机科学中又一对tradeoff. 不过根据KISS法则, 你现在可以暂时不实现这种特殊情况的处理, 在判断出数据跨越虚拟页边界的情况之后, 先使用 `assert(0)` 终止NEMU, 等到真的出现这种情况的时候再进行处理.

在NEMU中实现分页机制

根据上述的讲义内容, 在NEMU中实现i386分页机制, 如有疑问, 请查阅i386手册.

让用户程序运行在分页机制上

成功实现分页机制之后, 你会发现仙剑奇侠传也同样成功运行了. 但仔细想想就会发现这其实不太对劲: 我们在 `_asye_init()` 中创建了内核的虚拟地址空间, 之后就再也没有切换过这一虚拟地址空间. 也就是说, 我们让仙剑奇侠传也运行在内核的虚拟地址空间之上! 这太不合理了, 虽然NEMU没有实现ring 3, 但用户进程还是应该有自己的一套虚拟地址空间. 更可况, Navy-apps之前让用户程序链接到 `0x4000000` 的位置, 是因为之前Nanos-lite并没有对空闲的物理内存进行管理; 现在引入了分页机制, 由MM来负责所有物理页的分配. 这意味着, 如果将来MM把 `0x4000000` 所在的物理页分配出去, 仙剑奇侠传的内容将会被覆盖! 因此, 目前仙剑奇侠传看似运行成功, 其实里面暗藏杀机.

正确的做法是, 我们应该让用户程序运行在操作系统为其分配的虚拟地址空间之上. 为此, 我们需要对工程作一些变动. 首先需要将 `navy-apps/Makefile.compile` 中的链接地址 `-Ttext` 参数改为 `0x8048000` , 这是为了避免用户程序的虚拟地址空间与内核相互重叠, 从而产生非预期的错误. 同样的, `nanos-lite/src/loader.c` 中的 `DEFAULT_ENTRY` 也需要作相应的修改. 这时, "虚拟地址作为物理地址的抽象"这一好处已经体现出来了: 原则上用户程序可以运行在任意的虚拟地址, 不受物理内存容量的限制. 我们让用户程序的代码从 `0x8048000` 附近开始, 这个地址已经超过了物理地址的最大值(NEMU提供的物理内存是128MB), 但分页机制保证了程序能够正确运行. 这样, 链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址, 它们只要使用虚拟地址就可以了, 而虚拟地址和物理地址之间的映射则全部交给操作系统的MM来管理.

然后, 我们让Nanos-lite通过 `load_prog()` 函数(在 `nanos-lite/src/proc.c` 中定义)来进行用户程序的加载:

```
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,2 +33,1 @@
-  uintptr_t entry = loader(NULL, "/bin/pal");
-  ((void (*)(void))entry)();
+  load_prog("/bin/dummy");
```

我们先运行dummy, 是因为让仙剑奇侠传成功运行在虚拟地址空间上还需要进行一些额外的工作. `load_prog()` 函数首先会通过 `_protect()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义) 创建一个用户进程的虚拟地址空间, 这个虚拟地址空间除了内核映射之外就没有其它内容了. 框架代码在调用 `_protect()` 的时候用到了一个 `PCB` 的结构体, 我们会在后面再介绍它, 目前只需要知道虚拟地址空间的信息被存放在 `PCB` 结构体的 `as` 成员中即可. 然后 `load_prog()` 会调用 `loader()` 函数加载用户程序. 需要注意的是, 此时 `loader()` 不能直接把用户程序加载到内存位置 `0x8048000` 附近了, 因为这个地址并不在内核的虚拟地址空间中, 内核不能直接访问它. `loader()` 要做的事情是, 获取用户程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 把这一物理页映射到用户程序的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

这一切都是为了让用户进程在将来可以正确地运行: 用户进程在将来使用虚拟地址访问内存, 在loader为用户进程准备的映射下, 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. 为了提供映射一页的功能, 你需要在AM中实现 `_map()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义). 它的函数原型如下

```
void _map(_Protect *p, void *va, void *pa);
```

功能是将虚拟地址空间 `p` 中的虚拟地址 `va` 映射到物理地址 `pa` . 通过 `p->ptr` 可以获取页目录的基地址. 若在映射过程中发现需要申请新的页表, 可以通过回调函数 `palloc_f()` 向Nanos-lite获取一页空闲的物理页.

从 `loader()` 返回后, `load_prog()` 会调用 `_switch()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义), 切换到刚才为用户程序创建的地址空间. 最后跳转到用户程序的入口, 此时用户程序已经完全运行在分页机制上了.

让用户程序运行在分页机制上

根据上述的讲义内容, 在PTE中实现 `_map()` , 然后修改 `loader()` 的内容, 通过 `_map()` 在用户程序的虚拟地址空间中创建虚拟页, 并把用户程序加载到虚拟地址空间上.

实现正确后, 你会看到dummy程序最后输出GOOD TRAP的信息, 说明它确实在虚拟地址空间上成功运行了.

内核映射的作用

在 `_protect()` 函数中创建虚拟地址空间的时候, 有一处代码用于拷贝内核映射:

```
for (int i = 0; i < NR_PDE; i ++) {
  updir[i] = kpdirs[i];
}
```

尝试注释这处代码, 重新编译并运行, 你会看到发生了错误. 请解释为什么会发生这个错误.

在分页机制上运行仙剑奇侠传

之前我们让 `mm_brk()` 函数直接返回 `0`, 表示用户程序的堆区大小修改总是成功, 这是因为在实现分页机制之前, `0x4000000` 之上的内存都可以让用户程序自由使用. 现在用户程序运行在虚拟地址空间之上, 我们还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中:

```
int mm_brk(uint32_t new_brk) {
  if (current->cur_brk == 0) {
    current->cur_brk = current->max_brk = new_brk;
  }
  else {
    if (new_brk > current->max_brk) {
      // TODO: map memory region [current->max_brk, new_brk)
      // into address space current->as


      current->max_brk = new_brk;
    }

    current->cur_brk = new_brk;
  }

  return 0;
}
```

你需要填充上述TODO处的代码, 其中 `current` 是一个特殊的指针, 我们会在后面介绍它. 你需要注意 `_map()` 参数是否需要按页对齐的问题(这取决于你的 `_map()` 实现). 为了简化, 我们也不实现堆区的回收功能了.

实现正确后, 仙剑奇侠传就可以正确在分页机制上运行了.

温馨提示

PA4阶段1到此结束.

上下文切换

我们已经可以让用户程序运行在相互独立的虚拟地址空间上了, 我们只需要再加入上下文切换的机制, 就可以实现一个眞正的分时多任务操作系统了! 所谓上下文, 其实可以看作是程序运行时候的状态. 聪明的你应该马上能想起来, 我们在PA3中遇到的陷阱帧, 不就正好保存了程序的状态了吗? 没错, 要实现上下文切换, 就是要实现在不同程序的陷阱帧之间的切换!

具体地, 假设程序A运行的过程中触发了系统调用, 陷入到内核. 根据 `asm_trap()` 的代码, A的陷阱帧将会被保存到A的堆栈上. 本来系统调用处理完毕之后, `asm_trap()` 会根据A的陷阱帧恢复A的现场. 神奇的地方来了, 如果我们先不着急恢复A的现场, 而是先将栈顶指针切换到另一个程序B的堆栈上, 接下来的恢复现场操作将会恢复成B的现场: 恢复B的通用寄存器, 弹出#irq和错误码, 恢复B的EIP, CS, EFLAGS. 从 `asm_trap()` 返回之后, 我们已经在运行程序B了!

那程序A到哪里去了呢? 别担心, 它只是被暂时"挂起"了而已. 在被挂起之前, 它已经把现场的信息保存在自己的堆栈上了, 如果将来的某一时刻栈顶指针被切换到A的堆栈上, 代码将会根据A的"陷阱帧"恢复A的现场, A将得以唤醒并执行. 所以, 上下文切换其实就是不同程序之间的堆栈切换!

我们只要稍稍借助数学归纳法, 就可以让我们相信这个过程对于正在运行的程序来说总是正确的. 那么, 对于刚刚加载完的程序, 我们要怎么切换到它来让它运行起来呢? 答案很简单, 我们只需要在程序的堆栈上人工初始化一个陷阱帧, 使得将来切换的时候可以根据这个人工陷阱帧来正确地恢复现场即可.

在讨论具体如何初始化陷阱帧之前, 我们先来看一个关键的问题: 我们要如何找到别的程序的陷阱帧呢? 注意到陷阱帧是在堆栈上的形成的, 但堆栈那么大, 受到函数调用形成的栈帧的影响, 每次形成陷阱帧的位置并不是固定的. 自然地, 我们需要一个指针 `tf` 来记录陷阱帧的位置, 当想要找到别的程序的陷阱帧的时候, 只要寻找这个程序相关的 `tf` 指针即可.

事实上, 有不少信息都是进程相关的, 除了刚才提到的陷阱帧位置 `tf` 之外, 还有我们之前遇到的虚拟地址空间, 以及用户进程堆区的位置. 对于用户进程, 还需要有一个堆栈. 为了方便对进程进行管理, 操作系统使用一种叫进程控制块(PCB, process control block)的数据结构, 为每一个进程维护一个PCB. Nanos-lite的框架代码中已经定义了我们所需要使用的PCB结构(在 `nanos-lite/include/proc.h` 中定义):

```
typedef union {
  uint8_t stack[STACK_SIZE] PG_ALIGN;
  struct {
    _RegSet *tf;
    _Protect as;
    uintptr_t cur_brk;
    uintptr_t max_brk;
  };
} PCB;
```

Nanos-lite使用一个联合体来把其它信息放置在进程堆栈的底部. 代码为每一个进程分配了一个32KB的堆栈, 已经足够使用了, 不会出现栈溢出导致PCB中的其它信息被覆盖的情况. 在进行上下文切换的时候, 只需要把PCB中的tf指针返回给ASYE的 `irq_handle()` 函数即可, 剩余部分的代码会根据上下文信息恢复现场. 在GNU/Linux中, 进程控制块是通过 `task_struct` 结构来定义的.

因此, 我们要做的事情, 就是在用户进程的堆栈上初始化一个陷阱帧. 具体来说, 就是如何初始化陷阱帧中的每一个域, 因此你需要仔细思考陷阱帧中的每一个域对一开始运行的用户进程有什么影响. 提醒一下, 为了保证differential testing的正确运行, 我们还是把陷阱帧中的 `cs` 设置为 `8` . 这件事情是通过PTE提供的 `_umake()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义)来实现的, 它的原型是

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack,
  void *entry, char *const argv[], char *const envp[]);
```

`_umake()` 是专门用来创建用户进程的现场的, 但由于NEMU并没有实现ring 3, Nanos-lite也对用户进程作了一些简化, 因此目前 `_umake()` 只需要实现以下功能: 在 `ustack` 的底部初始化一个以 `entry` 为返回地址的陷阱帧. `p` 是用户进程的虚拟地址空间, 在简化之后, `_umake()` 不需要使用它. `argv` 和 `envp` 分别是用户进程的 `main()` 函数参数和环境变量, 目前Nanos-lite暂不支持, 因此我们可以忽略它们. 但是, Navy-apps中程序的入口函数是 `navy-apps/libs/libc/src/start.c` 中的 `_start()` 函数, `_start()` 函数认为它是有参数的, 因此我们还需要在陷阱帧之前设置好 `_start()` 函数的栈帧, 这是为了 `_start()` 开始执行的时候, 可以访问到正确的栈帧. 我们只需要把这一栈帧中的参数设置为 `0` 或 `NULL` 即可, 至于返回地址, 我们永远不会从 `_start()` 返回, 因此可以不设置它.

因此, `_umake()` 函数需要在栈上初始化如下内容, 然后返回陷阱帧的指针, 由Nanos-lite把这一指针记录到用户进程PCB的 `tf` 中:

```
|               |
+---------------+ <---- ustack.end
|  stack frame  |
|   of _start() |
+---------------+
|               |
|   trap frame  |
|               |
+---------------+ <--+
|               |    |
|               |    |
|               |    |
|               |    |
+---------------+    |
|       tf      | ---+
+---------------+ <---- ustack.start
|               |
```

我们之前让Nanos-lite在加载用户程序后通过函数调用跳转到用户程序中执行. 事实上, 这并不是一个合理的方式, 从安全的角度来说, 高特权级的代码是不能直接跳转到低特权级的代码中执行的, 真实硬件的保护机制甚至会抛出异常来阻止这种情况的发生. 合理的做法是, 当操作系统初始化工作结束之后, 就会通过自陷指令触发一次上下文切换, 切换到第一个用户程序中来执行. 真实的操作系统就是这样做的.

为了测试 `_umake()` 的正确性, 我们也先通过自陷的方式触发第一次上下文切换. 内核自陷的功能与ISA相关, 是由ASYE的 `_trap()` 函数提供的. 在 `x86-nemu` 的AM中, 我们约定内核自陷通过指令 `int $0x81` 触发. ASYE的 `irq_handle()` 函数发现触发了内核自陷之后, 会包装成一个 `_EVENT_TRAP` 事件. Nanos-lite收到这个事件之后, 就可以返回第一个用户程序的现场了.

实现内核自陷

修改Nanos-lite的如下代码:

```
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,3 +33,5 @@
   load_prog("/bin/pal");

+  _trap();
+
   panic("Should not reach here");
--- nanos-lite/src/proc.c
+++ nanos-lite/src/proc.c
@@ -17,4 +17,4 @@
   // TODO: remove the following three lines after you have implemented _umake()
-  _switch(&pcb[i].as);
-  current = &pcb[i];
-  ((void (*)(void))entry)();
+  // _switch(&pcb[i].as);
+  // current = &pcb[i];
+  // ((void (*)(void))entry)();
```

并在ASYE添加相应的代码, 使得 `irq_handle()` 可以识别内核自陷并包装成 `_EVENT_TRAP` 事件, Nanos-lite接收到 `_EVENT_TRAP` 之后可以输出一句话, 然后直接返回即可, 因为真正的上下文切换还需要正确实现 `_umake()` 之后才能实现. 实现正确之后, 你会看到Nanos-lite触发了 `main()` 函数中最后的panic. 如果你不知道应该怎么做, 请参考你对PA3必答题中关于系统调用部分的回答.

上下文切换只是AM的工作, 而具体切换到哪个进程的上下文, 是由操作系统来决定的, 这项任务叫做进程调度. 进程调度是由 `schedule()` 函数(在 `nanos-lite/src/proc.c` 中定义)来完成的, 它用于返回将要调度的进程的上下文. 因此, 我们需要一种方式来记录当前正在运行哪一个进程, 这样我们才能在 `schedule()` 中返回另一个进程的现场, 以实现多任务的效果. 这一工作是通过 `current` 指针(在 `nanos-lite/src/proc.c` 中定义)实现的, 它用于指向当前运行进程的PCB. 这样, 我们就可以在 `schedule()` 中通过 `current` 来决定接下来要调度哪一个进程了. 不过在调度之前, 我们还需要把当前进程的上下文信息的位置保存在PCB当中:

```
// save the context pointer
current->tf = prev;

// always select pcb[0] as the new process
current = &pcb[0];

// TODO: switch to the new address space,
// then return the new context
```

目前 `schedule()` 只需要总是切换到第一个用户进程即可, 即 `pcb[0]` . 注意它的上下文是在加载程序的时候通过 `_umake()` 创建的, 在 `schedule()` 中才决定要切换到它, 然后在ASYE的 `asm_trap()` 中才真正地恢复这一上下文. 在 `schedule()` 返回之前, 还需要切换到新进程的虚

拟地址空间. 这样, 等到从异常处理的代码返回之后, 我们就已经正确地在仙剑奇侠传的虚拟地址空间中运行仙剑奇侠传的代码了!

实现上下文切换

根据讲义的上述内容, 实现以下功能:

- PTE的 `_umake()` 函数
- Nanos-lite的 `schedule()` 函数
- Nanos-lite收到 `_EVENT_TRAP` 事件后, 调用 `schedule()` 并返回其现场
- 修改ASYE中 `asm_trap()` 的实现, 使得从 `irq_handle()` 返回后, 先将栈顶指针切换到新进程的陷阱帧, 然后才根据陷阱帧的内容恢复现场, 从而完成上下文切换的本质操作

实现成功后, Nanos-lite就可以通过内核自陷触发上下文切换的方式运行仙剑奇侠传了.

分时多任务

我们已经实现了虚拟内存和上下文切换机制, Nanos-lite已经能支持分时多任务了! 这时候, 我们就可以加载第二个用户程序了:

```
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,3 +33,4 @@
   load_prog("/bin/pal");
+  load_prog("/bin/hello");

   _trap();
```

我们让仙剑奇侠传和hello程序分时运行. 需要注意的是, 我们目前只允许最多一个需要更新画面的进程参与调度, 这是因为多个这样的进程分时运行会导致画面被相互覆盖, 影响画面输出的效果. 在真正的图形界面操作系统中, 通常由一个窗口管理进程来统一管理画面的显示, 需要显示画面的进程与这一管理进程进行通信, 来实现更新画面的目的. 但这需要操作系统支持进程间通信的机制, 这已经超出了ICS的范围, 而且Nanos-lite作为一个裁剪版的操作系统, 也不提供进程间通信的服务. 因此我们进行了简化, 最多只允许一个需要更新画面的进程参与调度即可.

为此, 我们还需要修改调度的代码, 让 `schedule()` 轮流返回仙剑奇侠传和hello的现场:

```
current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
```

最后, 我们还需要选择一个时机来触发进程调度. 目前比较合适的时机就是处理系统调用之后: 修改 `do_event()` 的代码, 在处理完系统调用之后, 调用 `schedule()` 函数并返回其现场.

分时运行仙剑奇侠传和hello程序

根据讲义的上述内容, 添加相应的代码来实现仙剑奇侠传和hello程序之间的分时运行.

实现正确后, 你会看到仙剑奇侠传一边运行的同时, hello程序也会一边输出.

但我们会发现, 和hello程序分时运行之后, 仙剑奇侠传的运行速度有了明显的下降. 这其实再次向我们展现了"分时"的本质: 程序之间只是轮流使用处理器, 它们并不是真正意义上的"同时"运行. 为了让仙剑奇侠传尽量保持原来的性能, 我们可以在调度的时候进行一些修改.

优先级调度

我们可以修改`schedule()`的代码, 来调整仙剑奇侠传和hello程序调度的频率比例, 使得仙剑奇侠传调度若干次, 才让hello程序调度1次. 这是因为hello程序做的事情只是不断地输出字符串, 我们只需要让hello程序偶尔进行输出, 以确认它还在运行就可以了.

温馨提示

PA4阶段2到此结束.

来自外部的声音

我们终于实现了分时多任务了, 进程在系统调用返回之前, 将会触发 `schedule()` 进行进程的上下文切换. 嗯, 这套机制运行得非常顺利. 然而, 如果被调度的是一个有bug的, 意外陷入了死循环的程序, 又或者是个根本就没打算使用系统调用的恶意程序, 我们的操作系统将会如何?

非常遗憾, 这是一个致命的漏洞. 产生这个致命问题的原因, 是我们将上下文切换的触发条件寄托在程序的行为之上: 触发了系统调用, 才能触发上下文切换. 我们知道程序被调度的时候, 整个计算机都会被它所控制, 无论是计算, 访存, 还是输入输出, 都是由程序来决定的. 为了修复这个漏洞, 我们必须寻找一种程序也无法控制的机制.

回想起我们考试的时候, 在试卷上如何作答都是我们来控制的, 但等到铃声一响, 无论我们是否完成答题, 都要立即上交试卷. 我们希望的恰恰就是这样一种效果: 时间一到, 无论正在运行的进程有多不情愿, 操作系统都要进行上下文切换. 而解决问题的关键, 就是时钟. 我们在IOE中早就已经加入了时钟了, 然而这还不能满足我们的需求, 我们希望时钟能够主动地通知处理器, 而不是被动地等着处理器来访问.

这样的通知机制, 在计算机中称为硬件中断. 作为与程序行为无关的机制, 硬件中断除了可以成为上下文切换的根基之外, 还有其它好处. 例如, 我们目前实现的IOE中, 都是让CPU轮询设备的状态, 但让CPU一直监视设备的工作并不是明智的选择. 以磁盘为例, 磁盘进行一次读写需要花费大约5毫秒的时间, 但对于一个2GHz的CPU来说, 它需要花费10,000,000个周期来等待磁盘操作的完成. 这对CPU来说无疑是巨大的浪费, 因此我们迫切需要一种通知机制: 在磁盘读写期间, CPU可以继续执行与磁盘无关的代码; 磁盘读写结束后, 主动通知CPU, 这时CPU才继续执行与磁盘相关的代码. 这里的通知机制也就是硬件中断. 硬件中断的实质是一个数字信号, 当设备有事件需要通知CPU的时候, 就会发出中断信号. 这个信号最终会传到CPU中, 引起CPU的注意.

第一个问题就是中断信号是怎么传到CPU中的. 支持中断机制的设备控制器都有一个中断引脚, 这个引脚会和CPU的INTR引脚相连, 当设备需要发出中断请求的时候, 它只要将中断引脚置为高电平, 中断信号就会一直传到CPU的INTR引脚中. 但计算机上通常有多个设备, 而CPU引脚是在制造的时候就固定了, 因而在CPU端为每一个设备中断分配一个引脚的做法是不现实的.

为了更好地管理各种设备的中断请求, IBM PC兼容机中都会带有Intel 8259 PIC(Programmable Interrupt Controller, 可编程中断控制器). 中断控制器最主要的作用就是充当设备中断信号的多路复用器, 即在多个设备中断信号中选择其中一个信号, 然后转发给CPU.

第二个问题是CPU如何响应到来的中断请求. CPU每次执行完一条指令的时候, 都会看看INTR引脚, 看是否有设备的中断请求到来. 一个例外的情况就是CPU处于关中断状态. 在x86中, 如果EFLAGS中的IF位为0, 则CPU处于关中断状态, 此时即使INTR引脚为高电平, CPU也不会响

应中断. CPU的关中断状态和中断控制器是独立的, 中断控制器只负责转发设备的中断请求, 最终CPU是否响应中断还需要由CPU的状态决定.

如果中断到来的时候, CPU没有处在关中断状态, 它就要马上响应到来的中断请求. 我们刚才提到中断控制器会生成一个中断号, CPU将会保存中断现场, 然后根据这个中断号在IDT中进行索引, 找到并跳转到入口地址, 进行一些和设备相关的处理. 这个过程和之前提到的异常处理十分相似.

对CPU来说, 设备的中断请求何时到来是不可预测的, 在处理一个中断请求的时候到来了另一个中断请求也是有可能的. 如果希望支持中断嵌套 -- 即在进行优先级低的中断处理的过程中, 响应另一个优先级高的中断 -- 那么堆栈将是保存中断现场信息的唯一选择. 如果选择把现场信息保存在一个固定的地方, 发生中断嵌套的时候, 第一次中断保存的现场信息将会被优先级高的中断处理过程所覆盖, 从而造成灾难性的后果.

灾难性的后果(这个问题有点难度)

假设硬件把中断信息固定保存在内存地址 `0x1000` 的位置, AM也总是从这里开始构造trap frame. 如果发生了中断嵌套, 将会发生什么样的灾难性后果? 这一灾难性的后果将会以什么样的形式表现出来? 如果你觉得毫无头绪, 你可以用纸笔模拟中断处理的过程.

在NEMU中, 我们只需要添加时钟中断这一种中断就可以了. 由于只有一种中断, 我们也不需要通过中断控制器进行中断的管理, 直接让时钟中断连接到CPU的INTR引脚即可, 我们也约定时钟中断的中断号是 `32` . 时钟中断通过 `nemu/src/device/timer.c` 中的 `timer_intr()` 触发, 每10ms触发一次. 触发后, 会调用 `dev_raise_intr()` 函数(在 `nemu/src/cpu/intr.c` 中定义). 你需要:

- 在cpu结构体中添加一个 `bool` 成员 `INTR` .
- 在 `dev_raise_intr()` 中将INTR引脚设置为高电平.
- 在 `exec_wrapper()` 的末尾添加轮询INTR引脚的代码, 每次执行完一条指令就查看是否有硬件中断到来:

```
#define TIMER_IRQ 32

if (cpu.INTR & cpu.eflags.IF) {
  cpu.INTR = false;
  raise_intr(TIMER_IRQ, cpu.eip);
  update_eip();
}
```

- 修改 `raise_intr()` 中的代码, 在保存EFLAGS寄存器后, 将其IF位置为 `0` , 让处理器进入关中断状态.

在软件上, 你还需要:

- 在ASYE中添加时钟中断的支持, 将时钟中断打包成 `_EVENT_IRQ_TIME` 事件.

- Nanos-lite收到 `_EVENT_IRQ_TIME` 事件之后, 直接调用 `schedule()` 进行进程调度, 同时也可以去掉系统调用之后调用的 `schedule()` 代码了.
- 为了可以让处理器在运行用户进程的时候响应时钟中断, 你还需要修改 `_umake()` 的代码, 在构造现场的时候, 设置正确的EFLAGS.

添加时钟中断

根据讲义的上述内容, 添加相应的代码来实现真正的分时多任务.

为了证明时钟中断确实在工作, 你可以在Nanos-lite收到 `_EVENT_IRQ_TIME` 事件后用 `Log()` 输出一句话.

需要注意的是, 添加时钟中断之后, differential testing机制就无法正确工作了. 这是因为, 我们无法给QEMU注入时钟中断, 无法保证QEMU与NEMU处于相同的状态. 不过, differential testing作为一个强大的工具用到这时候, 指令实现的正确性也基本上得到相当大的保证了.

如果没有中断的存在, 计算机的运行就是完全确定的. 根据计算机的当前状态, 你完全可以推断出下一条指令执行后, 甚至是执行100条指令后计算机的状态. 正是中断的不可预测性, 给计算机世界带来了不确定性的乐趣. 而在分时多任务操作系统中, 中断更是操作系统赖以生存的根基: 只要中断的东风一刮, 操作系统就会卷土重来, 一个故意执行死循环的恶意程序就算有天大的本事, 此时此刻也要被请出CPU, 从而让其它程序得到运行的机会, 因此, 上下文切换的本质其实是中断驱动的堆栈切换; 如果没有中断, 一个陷入了死循环的程序将使操作系统万劫不复. 但另一方面, 中断的存在也不得不让操作系统在一些问题的处理上需要付出额外的代价, 最常见的问题就是保证某些操作的原子性: 如果在一个原子操作进行到一半的时候到来了中断, 数据的一致性状态将会被破坏, 成为了潜伏在系统中的炸弹; 而且由于中断到来是不可预测的, 重现错误可能需要付出比修复错误更大的代价... 即使这样, 中断对现代计算机作出的贡献是不可磨灭的, 由中断撑起半边天的操作系统也将长久不衰.

必答题

分时多任务的具体过程 请结合代码, 解释分页机制和硬件中断是如何支撑仙剑奇侠传和hello程序在我们的计算机系统(Nanos-lite, AM, NEMU)中分时运行的.

温馨提示

PA4到此结束. 请你编写好实验报告(不要忘记在实验报告中回答必答题), 然后把命名为 `学号.pdf` 的实验报告文件放置在工程目录下, 执行 `make submit` 对工程进行打包, 最后将压缩包提交到指定网站.

编写不朽的传奇

最后, 我们再来做一些小小的修改, 来展示我们亲手搭建的计算机系统.

展示你的计算机系统

让Nanos-lite加载第3个用户程序 `/bin/videotest` , 并在Nanos-lite的 `events_read()` 函数中添加以下功能: 当发现按下F12的时候, 让游戏在仙剑奇侠传和videotest之间切换. 为了实现这一功能, 你还需要修改 `schedule()` 的代码: 通过一个变量 `current_game` 来维护当前的游戏, 在 `current_game` 和hello程序之间进行调度. 例如, 一开始是仙剑奇侠传和hello程序分时运行, 按下F12之后, 就变成videotest和hello程序分时运行.

万变之宗 - 重新审视计算机

什么是计算机? 为什么看似平淡无奇的机械, 竟然能够搭建出如此缤纷多彩的计算机世界? 那些酷炫的游戏画面, 究竟和冷冰冰的电路有什么关系? 看着仙剑奇侠传运行的画面, 不妨思考一下, NEMU和AM分别如何支撑仙剑奇侠传的运行?

世界诞生的故事 - 终章

感谢你帮助先驱创造了这个美妙的世界! 同时也为自己编写了一段不朽的传奇! 也希望你可以和我们分享成功的喜悦! ^_^

故事到这里就告一段落了, PA也将要结束, 但对计算机的探索并没有终点. 如果你想知道这个美妙世界后来的样子, 可以翻一翻IA-32手册. 又或许, 你可以通过从先驱身上习得的创造力, 来改变这个美妙世界的轨迹, 书写故事新的篇章.

PA5 - 从一到无穷大: 程序与性能

世界诞生的故事 - 外传

先驱已经创造了一个功能齐全现代计算机, 终于可以来思考计算机系统领域中扩日持久的终极问题了: 如何让程序跑得更快?

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

PA5为选做实验, 不计入PA成绩.

浮点数的支持

我们已经在PA3中把仙剑奇侠传运行起来了, 但却不能战斗, 这是因为还有一些浮点数相关的工作需要处理. 现在到了处理的时候了. 要在NEMU中实现浮点指令也不是不可能的事情. 但实现浮点指令需要涉及x87架构的很多细节, 根据KISS法则, 我们选择了一种更简单的方式: 我们通过整数来模拟实数的运算, 这样的方法叫binary scaling.

我们先来说明如何用一个32位整数来表示一个实数. 为了方便叙述, 我们称用binary scaling方法表示的实数的类型为 `FLOAT` . 我们约定最高位为符号位, 接下来的15位表示整数部分, 低16位表示小数部分, 即约定小数点在第15和第16位之间(从第0位开始). 从这个约定可以看到, `FLOAT` 类型其实是实数的一种定点表示.

```
31  30                 16                   0
+----+------------------+--------------------+
|sign|      integer     |      fraction      |
+----+------------------+--------------------+
```

这样, 对于一个实数 `a` , 它的 `FLOAT` 类型表示 `A = a * 2^16` (截断结果的小数部分). 例如实数 `1.2` 和 `5.6` 用 `FLOAT` 类型来近似表示, 就是

```
1.2 * 2^16 = 78643 = 0x13333
+----+------------------+--------------------+
| 0  |         1        |        3333        |
+----+------------------+--------------------+


5.6 * 2^16 = 367001 = 0x59999
+----+------------------+--------------------+
| 0  |         5        |        9999        |
+----+------------------+--------------------+
```

而实际上, 这两个 `FLOAT` 类型数据表示的数是:

```
0x13333 / 2^16 = 1.19999695
0x59999 / 2^16 = 5.59999084
```

对于负实数, 我们用相应正数的相反数来表示, 例如 `-1.2` 的 `FLOAT` 类型表示为:

```
-(1.2 * 2^16) = -0x13333 = 0xfffecccd
```

比较FLOAT和float

`FLOAT` 和 `float` 类型的数据都是32位, 它们都可以表示2^32个不同的数. 但由于表示方法不一样, `FLOAT` 和 `float` 能表示的数集是不一样的. 思考一下, 我们用 `FLOAT` 来模拟表示 `float` , 这其中隐含着哪些取舍?

接下来我们来考虑 `FLOAT` 类型的常见运算, 假设实数 `a` , `b` 的 `FLOAT` 类型表示分别为 `A` , `B` .

- 由于我们使用整数来表示 `FLOAT` 类型, `FLOAT` 类型的加法可以直接用整数加法来进行:

  ```
  A + B = a * 2^16 + b * 2^16 = (a + b) * 2^16
  ```

- 由于我们使用补码的方式来表示 `FLOAT` 类型数据, 因此 `FLOAT` 类型的减法用整数减法来进行.

  ```
  A - B = a * 2^16 - b * 2^16 = (a - b) * 2^16
  ```

- `FLOAT` 类型的乘除法和加减法就不一样了:

  ```
  A * B = a * 2^16 * b * 2^16 = (a * b) * 2^32 != (a * b) * 2^16
  ```

  也就是说, 直接把两个 `FLOAT` 数据相乘得到的结果并不等于相应的两个浮点数乘积的 `FLOAT` 表示. 为了得到正确的结果, 我们需要对相乘的结果进行调整: 只要将结果除以 `2^16` , 就能得出正确的结果了. 除法也需要对结果进行调整, 至于如何调整, 当然难不倒聪明的你啦.
- 如果把 `A = a * 2^16` 看成一个映射, 那么在这个映射的作用下, 关系运算是保序的, 即 `a <= b` 当且仅当 `A <= B` , 故 `FLOAT` 类型的关系运算可以用整数的关系运算来进行.

有了这些结论, 要用 `FLOAT` 类型来模拟实数运算就很方便了. 除了乘除法需要额外实现之外, 其余运算都可以直接使用相应的整数运算来进行. 例如

```
float a = 1.2;
float b = 10;
int c = 0;
if (b > 7.9) {
  c = (a + 1) * b / 2.3;
}
```

用 `FLOAT` 类型来模拟就是

```
FLOAT a = f2F(1.2);
FLOAT b = int2F(10);
int c = 0;
if (b > f2F(7.9)) {
  c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

其中还引入了一些类型转换函数来实现和 `FLOAT` 相关的类型转换.

仙剑奇侠传的框架代码已经用 `FLOAT` 类型对浮点数进行了相应的处理. 你还需要实现一些和 `FLOAT` 类型相关的函数:

```
/* navy-apps/apps/pal/include/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);
/* navy-apps/apps/pal/src/FLOAT/FLOAT.c */
FLOAT f2F(float a);
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
```

其中 `F_mul_int()` 和 `F_div_int()` 用于计算一个 `FLOAT` 类型数据和一个整型数据的积/商, 这两种特殊情况可以快速计算出结果, 不需要将整型数据先转化成 `FLOAT` 类型再进行运算.

事实上, 我们并没有考虑计算结果溢出的情况, 不过仙剑奇侠传中的浮点数结果都可以在 `FLOAT` 类型中表示, 所以你可以不关心溢出的问题. 如果你不放心, 你可以在上述函数的实现中插入assertion来捕捉溢出错误.

实现binary scaling

实现上述函数来在仙剑奇侠传中对浮点数操作进行模拟. 实现正确后, 你就可以在仙剑奇侠传中成功进行战斗了.

通往高速的次元

恭喜你! 你亲手一砖一瓦搭建的计算机世界可以运行真实的程序, 确实是一个了不起的成就! 不过通常来说, 仙剑奇侠传会运行得比较慢, 现在是时候对NEMU进行优化了.

说起优化, 不知道你有没有类似的经历: 辛辛苦苦优化了一段代码, 结果发现程序的性能并没有得到明显的提升. 事实上, Amdahl's law早就看穿了这一切: 如果优化之前的这段代码只占程序运行总时间的很小比例, 即使这段代码的性能被优化了成千上万倍, 程序的总体性能也不会有明显的提升. 如果把上述情况反过来, Amdahl's law就会告诉我们并行技术的理论极限: 如果一个任务有5%的时间只能串行完成(例如初始化), 那么即使使用成千上万个核来进行并行处理, 完成这个任务所需要的时间最多快20倍.

跑题了... 总之, 盲目对代码进行优化并不是一种合理的做法. 好钢要用在刀刃上, Amdahl's law给你最直接的启示, 就是要优化hot code, 也就是那些占程序运行时间最多的代码. KISS法则告诉你, 不要在一开始追求绝对的完美, 一个原因就是, 在整个系统完成之前, 你根本就不知道系统的性能瓶颈会出现在哪一个模块中. 你一开始辛辛苦苦追求的完美, 对整个系统的性能提升也许只是九牛一毛, 根本不值得你花费这么多时间. 从这方面来说, 我们不得不承认KISS法则还是很有先见之明的.

那么怎样才能找到hot code? 一边盯着代码, 一边想"我认为...", "我觉得...", 这可不是什么靠谱的做法. 最可靠的方法当然是把程序运行一遍, 对代码运行时间进行统计. Profiler(性能剖析工具)就是专门做这些事情的.

GNU/Linux内核提供了性能剖析工具perf, 可以方便地收集程序运行的信息. 通过运行 `perf record` 命令进行信息收集:

```
perf record nemu/build/nemu nanos-lite/build/nanos-lite-x86-nemu.bin
```

如果运行时发现类似如下错误:

```
/usr/bin/perf: line 24: exec: perf_4.9: not found
E: linux-tools-4.9 is not installed.
```

请安装 `linux-tools` :

```
apt-get install linux-tools
```

通过 `perf record` 命令运行NEMU后, `perf` 会在NEMU的运行过程中收集性能数据. 当NEMU运行结束后, `perf` 会生成一个名为 `perf.data` 的文件, 这个文件记录了收集的性能数据. 运行命令 `perf report` 可以查看性能数据, 从而得知NEMU的性能瓶颈.

性能瓶颈的来源

Profiler可以找出实现过程中引入的性能问题, 但却几乎无法找出由设计引入的性能问题. NEMU毕竟是一个教学模拟器, 当设计和性能有冲突时, 为了达到教学目的, 通常会偏向选择易于教学的设计. 这意味着, 如果不从设计上作改动, NEMU的性能就无法突破上述取舍造成的障壁. 纵观NEMU的设计, 你能发现有哪些可能的性能瓶颈吗?

天下武功唯快不破

相信你也已经在NEMU中运行过microbench, 发现NEMU的性能连真机的1%都不到. 使用 `perf` 也没有发现能突破性能瓶颈的地方. 那NEMU究竟慢在哪里呢?

回想一下, 执行程序, 其实就是不断地执行程序的每一条指令: 取指, 译码, 执行, 更新 `eip` ... 在真机中, 这个过程是通过高速的门电路来实现的. 但NEMU毕竟是个模拟器, 只能用软件来实现"执行指令"的过程: 执行一次 `exec_wrapper()` , 客户程序才执行一条指令, 但运行NEMU的真机却需要执行上百条native指令. 这也是NEMU性能不高的根本原因. 为了方便叙述, 我们将"客户程序的指令"称为"客户指令". 因此, 作为软件模拟器的NEMU注定无法摆脱"用n条native指令模拟一条客户指令"的命运. 要提高NEMU的性能, 我们就只能想办法减小 `n` 了.

事实上, 模拟器的这种工作方式称为解释执行: 每一条客户指令的执行都需要经历完整的指令生命周期. 但仔细回顾一下计算机的本质, 执行指令的最终结果就是改变计算机的状态(寄存器和内存), 而真正改变状态的动作, 只有指令生命周期中的"执行"阶段, 其它阶段都是为状态的改变作铺垫: 取指是为了取到指令本身, 译码是为了看指令究竟要怎么执行, 更新 `eip` 只是为了执行下一条指令. 而且, 每次解释执行的时候, 这些辅助阶段做的事情都是一样的. 例如

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
```

每次执行到这条指令的时候, 取指都是取到相同的比特串, 译码总是发现"要将0x1234移动到 `eax` 中", 更新 `eip` 后其值也总是 `0x100005` . 反正执行客户指令的结果就是改变计算机的状态, 这不就和执行

```
mov $0x1234,(cpu.eax的地址)
```

这条native指令的效果一样吗?

这正是即时编译(JIT) 的思想: 通过生成并执行native指令来直接改变(被模拟)计算机的状态. 这里的编译不再是"生成机器指令"的含义了, 而是更广义的"语言转换": 把客户程序中执行的机器语言转换成与之行为等价的native机器语言. "即时"是指"这一编译的动作并非事先进行", 而是"在客户程序执行的过程中进行", 这样的好处是, 不会被执行到的客户指令, 就不需要进行编译.

为了生成native指令, 我们至少也要知道相应的客户指令要做什么. 因此, 我们至少也要对客户指令进行一次取指和译码. 通常情况下, 客户指令不会发生变化, 因此编译成的native指令也不会发生变化. 这意味着, 我们只需要对客户指令进行一次编译, 生成相应的native指令, 然后存放起来, 将来碰到相同的客户指令, 就不必重新编译, 而是可以找到之前编译的结果直接执行了. 这恰恰就是cache的思想: 我们将native指令序列组织成一个TB(translation block), 用客户程序的 `eip` 来索引; 这个cache由一系列的TB组成; 执行客户程序的时候, 先用 `eip` 索引cache, 若

命中, 说明相应的客户指令已经被编译过了, 此时可以不必重新编译, 直接取出相应的TB并执行; 若缺失, 说明相应的客户指令还没有被编译过, 此时才需要对客户指令进行取指和译码, 并编译成相应的TB, 更新cache, 以便于将来多次执行.

一个值得考虑的问题是, 每次编译多少条客户指令比较合适? 若一次编译一条客户指令, 则会导致每执行完一条客户指令就需要重新对cache进行索引, 查看下一条客户执行是否被编译过. 细心的你会发现, 在即时编译模式中, 一条客户指令被编译过, 当且仅当它被执行过. 也就是说, NEMU在执行完一条客户执行之后, 都会去检查下一条指令有没有被执行过. 我们知道, 顺序执行是程序最基本的执行流之一. 我们很容易想到, 在一个顺序执行的模块中, 如果其中的一条指令被执行过, 那就意味着整个模块的每一条指令都已经被执行过. 这说明, 若一次编译一条客户指令, "检查下一条指令有没有被执行过"大多数时候是一个冗余的动作. 为了避免这些冗余的动作, 我们可以一次编译一个顺序执行的模块, 这样的模块称为基本块.

要如何进行编译呢? 我们知道, x86的指令集非常复杂, 如果要考虑每一条x86客户指令如何编译到native指令, 就太麻烦了. 嘿, 我们在PA2中引入的RTL就是用来解决这个问题的: RTL只有少数的基本指令, 我们只需要考虑如何将少数的RTL基本指令编译到native指令就可以了! 引入RTL还有另一个好处, 就是方便NEMU的移植:

```
           +-----+
  x86 ---> |     | ---> mips
 mips ---> | RTL | ---> arm
riscv ---> |     | ---> x86
           +-----+
|           |   |          |
+ front-end +   + back-end +
```

以RTL为分界, 我们可以把即时编译模式的NEMU分为两部分: 前端用于将客户程序的机器语言编译成RTL, 后端负责将RTL编译成native机器语言. 这样以后, 要在NEMU中支持一种新的客户程序架构x, 只需要增加相应的前端模块来将x编译成RTL即可; 要让NEMU运行在一种新的架构y, 只需要增加相应的后端模块来将RTL编译成y即可.

于是, 即时编译模式的NEMU的工作方式如下:

```
while (1) {
  tb = query_cache(cpu.eip);
  if ( cache miss ) {
    tb = jit_translate(cpu.eip); // translate a basic block
    update_cache(cpu.eip, tb);
  }

  jump_to(tb); // cpu.eip will be updated while executing tb
}
```

关于`jit_translate()`如何工作, 可以参考这篇讲述QEMU中的JIT如何实现的文章.

什么? 这就没有了?

事实上, 实现JIT的坑非常多, yzh也还没全踩完, 也就还没总结出好的要点. 即使把坑都踩过了, 拖延症患者yzh也不一定有时间把这些要点整理成讲义.

不过如果你看到这里, 相信你也有一定能力来面对这些坑了. 踩坑其实是非常非常宝贵的经验, 也是做这些项目的意义所在: 通过做项目, 知道了以前永远也不可能知道的东西. 上述文章提到, QEMU的早期版本可以做到只比真机性能慢4倍. 看着microbench的分数越来越高, 了解每一项技术带来的性能提升及其背后揭示的原理, 这些都是最好的回报, 也是系统方向科研人员所追求的奥义.

开源项目的大门已经向你敞开: 不妨尝试一下阅读QEMU的源代码 (虽然现在的QEMU已经不是上述那篇十几年前的文章所说的那个样子了). NEMU的架构也不够完美: 欢迎和yzh交流你实现JIT所踩过的坑.

这道蓝框题之后的讲义内容, 你, 也许就是作者.

为什么要学习计算机系统基础

一知半解

你已经学过 `程序设计基础` 课程了，对于C和C++程序设计已有一定的基础，但你会发现，你可能还是不能理解以下程序的运行结果：

数组求和

```
int sum(int a[ ], unsigned len) {
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

当 `len = 0` 时，执行 `sum` 函数的for循环时会发生 `Access Violation` ，即"访问违例"异常. 但是，当参数 `len` 说明为 `int` 型时， `sum` 函数能正确执行，为什么？

整数的平方

若 `x` 和 `y` 为 `int` 型，当 `x = 65535` 时，则 `y = x*x = -131071` . 为什么？

多重定义符号

```
/*---main.c---*/
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main() {
    p1();
    printf("d=%d,x=%d\n",d,x);
    return 0;
}

/*---p1.c---*/
double d;
void p1() {
    d=1.0;
}
```

在上述两个模块链接生成的可执行文件被执行时，`main` 函数的 `printf` 语句打印出来的值是：`d=0,x=1072693248` . 为什么不是 `d=100,x=200` ？

奇怪的函数返回值

```
double fun(int i) {
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

从 `fun` 函数的源码来看，每次返回的值应该都是 `3.14` ，可是执行 `fun` 函数后发现其结果是：

- `fun(0)` 和 `fun(1)` 为 `3.14`
- `fun(2)` 为 `3.1399998664856`
- `fun(3)` 为 `2.00000061035156`
- `fun(4)` 为 `3.14` 并会发生 `访问违例` 这是为什么？

时间复杂度和功能都相同的程序

```
void copyij(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
void copyji(int src[2048][2048], int dst[2048][2048]) {
      int i,j;
      for (j = 0; j < 2048; j++)
          for (i = 0; i < 2048; i++)
              dst[i][j] = src[i][j];
}
```

上述两个功能完全相同的函数，时间复杂度也完全一样，但在Pentium 4处理器上执行时，所测时间相差大约21倍. 这是为什么？猜猜看是 `copyij` 更快还是 `copyji` 更快？

网友贴出的一道百度招聘题

请给出以下C语言程序的执行结果，并解释为什么.

```c
#include <stdio.h>
int main() {
    double a = 10;
    printf("a = %d\n", a);
    return 0;
}
```

该程序在IA-32上运行时, 打印结果为 `a=0` ; 在x86-64上运行时, 打印出来的 `a` 是一个不确定值. 为什么?

整数除法

以下两个代码段的运行结果是否一样呢?

- 代码段一: `c int a = 0x80000000; int b = a / -1; printf("%d\n", b); C`
- 代码段二:

  ```c
  int a = 0x80000000;
  int b = -1;
  int c = a / b;
  printf("%d\n", c);
  ```

  代码段一的运行结果为 `-2147483648` ; 而代码段二的运行结果为 `Floating point exception` , 显然, 代码段二运行时被检测到了"溢出"异常. 看似同样功能的程序为什么结果完全不同?

类似上面这些例子还可以举出很多. 从这些例子可以看出, 仅仅明白高级语言的语法和语义, 很多情况下是无法理解程序执行结果的.

站得高, 看得远

国内很多学校老师反映, 学完高级语言程序设计后会有一些学生不喜欢计算机专业了, 这是为什么? 从上述给出的例子应该可以找到部分答案, 如果一个学生经常对程序的执行结果百思不得其解, 那么他对应用程序开发必然产生恐惧心理, 也就对计算机专业逐渐失去兴趣. 其实, 程序的执行结果除了受编程语言的语法和语义影响外, 还与程序的执行机制息息相关. `计算机系统基础` 课程主要描述程序的底层执行机制, 因此, 学完本课程后同学们就能很容易地理解各种程序的执行结果, 也就不会对程序设计失去信心了.

我们还经常听到学生问以下问题: 像地质系这些非计算机专业的学生自学JAVA语言等课程后也能找到软件开发的工作, 而我们计算机专业学生多学那么多课程不也只能干同样的事情吗? 我们计算机专业学生比其他专业自学计算机课程的学生强在哪里啊? 现在计算机学科发展这么快, 什么领域都和计算机相关, 为什么我们计算机学科毕业的学生真正能干的事也不多呢? ...

确实, 对于大部分计算机本科专业学生来说, 硬件设计能力不如电子工程专业学生, 行业软件开发和应用能力不如其他相关专业学生, 算法设计和分析基础又不如数学系学生. 那么, 计算机专业学生的特长在哪里? 我们认为计算机专业学生的优势之一在于计算机系统能力, 即具备计算机系统层面的认知与设计能力、能从计算机系统的高度考虑和解决问题.

随着大规模数据中心(WSC)的建立和个人移动设备(PMD)的大量普及使用, 计算机发展进入了后PC时代, 呈现出"人与信息世界及物理世界融合"的趋势和网络化, 服务化, 普适化和智能化的鲜明特征. 后PC时代WSC, PMD和PC等共存, 使得原先基于PC而建立起来的专业教学内容已经远远不能反映现代社会对计算机专业人才的培养要求, 原先计算机专业人才培养强调"程序"设计也变为更强调"系统"设计.

后PC时代, 并行成为重要主题, 培养具有系统观的, 能够进行软, 硬件协同设计的软硬件贯通人才是关键. 而且, 后PC时代对于大量从事应用开发的应用程序员的要求也变得更高. 首先, 后PC时代的应用问题更复杂, 应用领域更广泛. 其次, 要能够编写出各类不同平台所适合的高效程序, 应用开发人员必需对计算机系统具有全面的认识, 必需了解不同系统平台的底层结构, 并掌握并行程序设计技术和工具.

下图描述了计算机系统抽象层的转换.

从图中可以看出, 计算机系统由不同的抽象层构成, "计算"的过程就是不同抽象层转换的过程, 上层是下层的抽象, 而下层则是上层的具体实现. 计算机学科主要研究的是计算机系统各个不同抽象层的实现及其相互转换的机制, 计算机学科培养的应该主要是在计算机系统或在系统某些层次上从事相关工作的人才.

相比于其他专业, 计算机专业学生的优势在于对系统深刻的理解, 能够站在系统的高度考虑和解决应用问题, 具有系统层面的认知和设计能力, 包括:

- 能够对软, 硬件功能进行合理划分
- 能够对系统不同层次进行抽象和封装
- 能够对系统的整体性能进行分析和调优

- 能够对系统各层面的错误进行调试和修正
- 能够根据系统实现机理对用户程序进行准确的性能评估和优化
- 能够根据不同的应用要求合理构建系统框架等

要达到上述这些在系统层面上的分析, 设计, 检错和调优等系统能力, 显然需要提高学生对整个计算机系统实现机理的认识, 包括:

- 对计算机系统整机概念的认识
- 对计算机系统层次结构的深刻理解
- 对高级语言程序, ISA, OS, 编译器, 链接器等之间关系的深入掌握
- 对指令在硬件上执行过程的理解和认识
- 对构成计算机硬件的基本电路特性和设计方法等的基本了解等 从而能够更深刻地理解时空开销和权衡, 抽象和建模, 分而治之, 缓存和局部性, 吞吐率和时延, 并发和并行, 远程过程调用(RPC), 权限和保护等重要的核心概念, 掌握现代计算机系统中最核心的技术和实现方法.

  `计算机系统基础` 课程的主要教学目标是培养学生的系统能力, 使其成为一个"高效"程序员, 在程序调试, 性能提升, 程序移植和健壮性等方面成为高手; 建立扎实的计算机系统概念, 为后续的OS, 编译, 体系结构等课程打下坚实基础.

实践是检验眞理的唯一标准

旷日持久的计算机教学只为解答三个问题：

- (theory, 理论计算机科学) 什么是计算?
- (system, 计算机系统) 什么是计算机?
- (application, 计算机应用) 我们能用计算机做什么?

除了纯理论工作之外, 计算机相关的工作无不强调动手实践的能力. 很多时候, 你会觉得理解某一个知识点是一件简单是事情, 但当你眞正动手实践的时候, 你才发现你的之前的理解只是停留在表面. 例如你知道链表的基本结构, 但你能写出一个正确的链表程序吗? 你知道程序加载的基本原理, 但你能写一个加载器来加载程序吗? 你知道编译器, 操作系统, CPU的基本功能, 但你能写一个编译器, 操作系统, CPU吗? 你甚至会发现, 虽然你在程序设计课上写过很多程序, 但你可能连下面这个看似很简单的问题都无法回答:

终极拷问

当你运行一个Hello World程序的时候, 计算机究竟做了些什么?

很多东西说起来简单, 但做起来却不容易, 动手实践会让你意识到你对某些知识点的一知半解, 同时也给了你深入挖掘其中的机会, 你会在实践过程中发现很多之前根本没有想到过的问题(其实科研也是如此), 解决这些问题反过来又会加深你对这些知识点的理解. 理论知识和动手实践相互促进, 最终达到对知识点透彻的理解.

目前也有以下观点:

> 目前像VS, Eclipse这样的IDE功能都十分强大, 点个按钮就能编译, 拖动几个控件就能设计一个GUI程序, 为什么还需要学习程序运行的机理?
>
> PhotoShop里面的滤镜功能繁多, 随便点点就能美化图片, 为什么还需要学习图像处理的基本原理?

像"GUI程序开发", "PhotoShop图片美化"这样的工作也确实需要动手实践, 但它们并不属于上文提到的计算机应用的范畴, 也不是计算机本科教育的根本目的, 因为它们强调的更多是技能的培训, 而不是对"计算机能做什么"这个问题的探索, 这也是培训班教学和计算机本科教学的根本区别. 但如果你对GUI程序运行的机理了如指掌, 对图像处理基本原理的理解犹如探囊取物, 上述工作对你来说根本就不在话下, 甚至你还有能力参与Eclipse和PhotoShop的开发.

而对这些原理的透彻理解, 离不开动手实践.

宋公语录

学汽车制造专业是要学发动机怎么设计, 学开车怎么开得过司机呢?

实验提交说明

实验阶段

- 为了尽可能避免拖延症影响实验进度, 我们采用分阶段方式进行提交, 强迫大家每周都将实验进度往前推进. 在阶段性提交截止前, 你只需要提交你的工程, 并且实现的正确性不影响你的分数, 即我们允许你暂时提交有bug的实现. 在最后阶段中, 你需要提交你的工程和完整的实验报告, 同时我们也会检查实现的正确性.
- 如无特殊原因, 迟交的作业将损失30%的成绩(即使迟了1秒), 请大家合理分配时间.
- 但是, 如果你完全没有开始进行某阶段的实验内容, 请你不要进行相应的提交, 因为这会影响我们的工作. 一旦发现这种情况, 我们将会额外扣除你 `发现次数*10%` 的PA总成绩.

学术诚信

如果你确实无法独立完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数.

下表说明了你可能采取的各种策略的收益:

	并非完全没有完成相应内容	完全没有完成相应内容	抄袭
按时提交	100%(获得完成部分的全部分数)	`-发现次数*10%`	0%, 并通知辅导员
未按时提交	70%(迟交惩罚)	`-发现次数*10%`	0%, 并通知辅导员
不提交	10%(学术诚信奖励)	10%(学术诚信奖励)	10%(学术诚信奖励)

总的来说, 最好的策略是: 做了就交, 没做就不要交.

提交地址

http://cslabcms.nju.edu.cn

提交方式

把实验报告放到工程目录下之后, 使用 `make submit` 命令直接将整个工程打包即可. 请注意:

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

- 我们会使用脚本进行批量解压缩. `make submit` 命令会用你的学号来命名压缩包, 不要修改压缩包和工程根目录(ics2017)的命名, 否则脚本将不能正确工作. 另外为了防止出现编码问题, 压缩包中的所有文件名都不要包含中文.
- 我们只接受pdf格式, 命名只含学号的实验报告, 不符合格式的实验报告将视为没有提交报告. 例如 `141220000.pdf` 是符合格式要求的实验报告, 但 `141220000.docx` 和 `141220000张三实验报告.pdf` 不符合要求, 它们将不能被脚本识别出来.
- 如果你需要多次提交, 请先手动删除旧的提交记录(提交网站允许下载, 删除自己的提交记录)

git版本控制

我们鼓励你使用git管理你的项目, 如果你提交的实验中包含均匀合理的, 你手动提交的git记录(不是开发跟踪系统自动提交的), 你将会获得本次实验20%的分数奖励(总得分不超过本次实验的上限). 这里有一个十分简单的git教程, 更多的git命令请查阅相关资料. 另外, 请你不定期查看自己的git log, 检查是否与自己的开发过程相符. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据.

实验报告内容

你必须在实验报告中描述以下内容:

- 实验进度. 简单描述即可, 例如"我完成了所有内容", "我只完成了xxx". 缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验.
- 必答题.

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

- 你遇到的问题和对这些问题的思考
- 对讲义中蓝框思考题的看法
- 或者你的其它想法, 例如实验心得, 对提供帮助的同学的感谢等

认真描述实验心得和想法的报告将会获得分数的奖励; 蓝框题为选做, 完成了也不会得到分数的奖励, 但它们是经过精心准备的, 可以加深你对某些知识的理解和认识. 因此当你发现编写实验报告的时间所剩无几时, 你应该选择描述实验心得和想法. 如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求. 但请不要

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

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

Linux入门教程

以下内容引用自jyy的操作系统实验课程网站, 并有少量修改和补充. 如果你是第一次使用Linux, 请你一边仔细阅读教程, 一边尝试运行教程中提到的命令.

探索命令行

Linux命令行中的命令使用格式都是相同的:

```
命令名称 参数1 参数2 参数3 ...
```

参数之间用任意数量的空白字符分开. 关于命令行, 可以先阅读一些基本常识. 然后我们介绍最常用的一些命令:

- `ls` 用于列出当前目录(即"文件夹")下的所有文件(或目录). 目录会用蓝色显示. `ls -l` 可以显示详细信息.
- `pwd` 能够列出当前所在的目录.
- `cd DIR` 可以切换到 `DIR` 目录. 在Linux中, 每个目录中都至少包含两个目录: `.` 指向该目录自身, `..` 指向它的上级目录. 文件系统的根是 `/`.
- `touch NEWFILE` 可以创建一个内容为空的新文件 `NEWFILE`, 若 `NEWFILE` 已存在, 其内容不会丢失.
- `cp SOURCE DEST` 可以将 `SOURCE` 文件复制为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件复制到该目录下.
- `mv SOURCE DEST` 可以将 `SOURCE` 文件重命名为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件移动到该目录下.
- `mkdir DIR` 能够创建一个 `DIR` 目录.
- `rm FILE` 能够删除 `FILE` 文件; 如果使用 `-r` 选项则可以递归删除一个目录. 删除后的文件无法恢复, 使用时请谨慎!
- `man` 可以查看命令的帮助. 例如 `man ls` 可以查看 `ls` 命令的使用方法. 灵活应用 `man` 和互联网搜索, 可以快速学习新的命令.

`man` 的功能不仅限于此. `man` 后可以跟两个参数, 可以查看不同类型的帮助(请在互联网上搜索). 例如当你不知道C标准库函数 `freopen` 如何使用时, 可以键入命令

```
man 3 freopen
```

学会使用man

如果你是第一次使用 `man`，请阅读这里. 这个教程除了说明如何使用 `man` 之外，还会教你在使用一款新的命令行工具时如何获得帮助.

消失的cd

上述各个命令除了 `cd` 之外都能找到它们的manpage，这是为什么？如果你思考后仍然感到困惑，试着到互联网上寻找答案.

下面给出一些常用命令使用的例子，你可以键入每条命令之后使用 `ls` 查看命令执行的结果：

```
$ mkdir temp       # 创建一个目录temp
$ cd temp          # 切换到目录temp
$ touch newfile    # 创建一个空文件newfile
$ mkdir newdir     # 创建一个目录newdir
$ cd newdir        # 切换到目录newdir
$ cp ../newfile .  # 将上级目录中的文件newfile复制到当前目录下
$ cp newfile aaa   # 将文件newfile复制为新文件aaa
$ mv aaa bbb       # 将文件aaa重命名为bbb
$ mv bbb ..        # 将文件bbb移动到上级目录
$ cd ..            # 切换到上级目录
$ rm bbb           # 删除文件bbb
$ cd ..            # 切换到上级目录
$ rm -r temp       # 递归删除目录temp
```

更多的命令行知识

仅仅了解这些最基础的命令行知识是不够的. 通常，我们可以抱着如下的信条：只要我们能想到的，就一定有方便的办法能够办到. 因此当你想要完成某件事却又不知道应该做什么的时候，请向Google求助. 如果你想以Linux作为未来的事业，那就可以去图书馆或互联网上找一些相关的书籍来阅读.

统计代码行数

第一个例子是统计一个目录中(包含子目录)中的代码行数. 如果想知道当前目录下究竟有多少行的代码，就可以在命令行中键入如下命令：

```
find . | grep '\.c$\|\.h$' | xargs wc -l
```

如果用 `man find` 查看 `find` 操作的功能，可以看到 `find` 是搜索目录中的文件. Linux中一个点 `.` 始终表示Shell当前所在的目录，因此 `find .` 实际能够列出当前目录下的所有文件. 如果在文件很多的地方键入 `find .` ,将会看到过多的文件，此时可以按 `CTRL + c` 退出.

同样，用 `man` 查看 `grep` 的功能——"print lines matching a pattern". `grep` 实现了输入的过滤，我们的 `grep` 有一个参数，它能够匹配以 `.c` 或 `.h` 结束的文件. 正则表达式是处理字符串非常强大的工具之一，每一个程序员都应该掌握其相关的知识. 有兴趣的同学可以首先阅读一个基础的教程，然后看一个有趣的小例子: 如何用正则表达式判定素数. 正则表达式还可以用来编写一个30行的java表达式求值程序(传统方法几乎不可能)，聪明的你能想到是怎么完成的吗? 上述的 `grep` 命令能够提取所有 `.c` 和 `.h` 结尾的文件.

刚才的 `find` 和 `grep` 命令，都从标准输入中读取数据，并输出到标准输出. 关于什么是标准输入输出，请参考这里. 连接起这两个命令的关键就是管道符号 `|` . 这一符号的左右都是Shell命令，`A | B` 的含义是创建两个进程 `A` 和 `B` ，并将 `A` 进程的标准输出连接到 `B` 进程的标准输入. 这样，将 `find` 和 `grep` 连接起来就能够筛选出当前目录(`.`)下所有以 `.c` 或 `.h` 结尾的文件.

我们最后的任务是统计这些文件所占用的总行数，此时可以用 `man` 查看 `wc` 命令. `wc` 命令的 `-l` 选项能够计算代码的行数. `xargs` 命令十分特殊，它能够将标准输入转换为参数，传送给第一个参数所指定的程序. 所以，代码中的 `xargs wc -l` 就等价于执行 `wc -l aaa.c bbb.c include/ccc.h ...` ，最终完成代码行数统计.

统计磁盘使用情况

以下命令统计 `/usr/share` 目录下各个目录所占用的磁盘空间:

```
du -sc /usr/share/* | sort -nr
```

`du` 是磁盘空间分析工具，`du -sc` 将目录的大小顺次输出到标准输出，继而通过管道传送给 `sort` . `sort` 是数据排序工具，其中的选项 `-n` 表示按照数值进行排序，而 `-r` 则表示从大到小输出. `sort` 可以将这些参数连写在一起.

然而我们发现，`/usr/share` 中的目录过多，无法在一个屏幕内显示. 此时，我们可以再使用一个命令: `more` 或 `less` .

```
du -sc /usr/share/* | sort -nr | more
```

此时将会看到输出的前几行结果. `more` 工具使用空格翻页，并可以用 `q` 键在中途退出. `less` 工具则更为强大，不仅可以向下翻页，还可以向上翻页，同样使用 `q` 键退出. 这里还有一个关于less的小故事.

在Linux下编写Hello World程序

Linux中用户的主目录是 `/home/用户名称` , 如果你的用户名是 `user` , 你的主目录就是 `/home/user` . 用户的 `home` 目录可以用波浪符号 `~` 替代, 例如临时文件目录 `/home/user/Templates` 可以简写为 `~/Templates` . 现在我们就可以进入主目录并编辑文件了. 如果 `Templates` 目录不存在, 可以通过 `mkdir` 命令创建它:

```
cd ~
mkdir Templates
```

创建成功后, 键入

```
cd Templates
```

可以完成目录的切换. 注意在输入目录名时, `tab` 键可以提供联想.

你感到键入困难吗

你可能会经常要在终端里输入类似于

```
cd AVeryVeryLongFileName
```

的命令, 你一定觉得非常烦躁. 回顾上面所说的原则之一: 如果你感到有什么地方不对, 就一定有什么好办法来解决. 试试 `tab` 键吧.

Shell中有很多这样的小技巧, 你也可以使用其他的Shell例如zsh, 提供更丰富好用的功能. 总之, 尝试和改变是最重要的.

进入正确的目录后就可以编辑文件了, 开源世界中主流的两大编辑器是 `vi(m)` 和 `emacs` , 你可以使用其中的任何一种. 如果你打算使用 `emacs` , 你还需要安装它

```
apt-get install emacs
```

`vi` 和 `emacs` 这两款编辑器都需要一定的时间才能上手, 它们共同的特点是需要花较多的时间才能适应基本操作方式(命令或快捷键), 但一旦熟练运用, 编辑效率就比传统的编辑器快很多.

进入了正确的目录后, 输入相应的命令就能够开始编辑文件. 例如输入

```
vi hello.c
或emacs hello.c
```

就能开启一个文件编辑. 例如可以键入如下代码(对于首次使用 `vi` 或 `emacs` 的同学, 键入代码可能会花去一些时间, 在编辑的同时要大量查看网络上的资料):

```
#include <stdio.h>
int main(void) {
  printf("Hello, Linux World!\n");
  return 0;
}
```

保存后就能够看到 `hello.c` 的内容了. 终端中可以用 `cat hello.c` 查看代码的内容. 如果要将它编译, 可以使用 `gcc` 命令:

```
gcc hello.c -o hello
```

`gcc` 的 `-o` 选项指定了输出文件的名称, 如果将 `-o hello` 改为 `-o hi` , 将会生成名为 `hi` 的可执行文件. 如果不使用 `-o` 选项, 则会默认生成名为 `a.out` 的文件, 它的含义是assembler output. 在命令行输入

```
./hello
```

就能够运行改程序. 命令中的 `./` 是不能少的, 点代表了当前目录, 而 `./hello` 则表示当前目录下的 `hello` 文件. 与Windows不同, Linux系统默认情况下并不查找当前目录, 这是因为Linux下有大量的标准工具(如 `test` 等), 很容易与用户自己编写的程序重名, 不搜索当前目录消除了命令访问的歧义.

使用重定向

有时我们希望将程序的输出信息保存到文件中, 方便以后查看. 例如你编译了一个程序 `myprog` , 你可以使用以下命令对 `myprog` 进行反汇编, 并将反汇编的结果保存到 `output` 文件中:

```
objdump -d myprog > output
```

`>` 是标准输出重定向符号, 可以将前一命令的输出重定向到文件 `output` 中. 这样, 你就可以使用文本编辑工具查看 `output` 了.

但你会发现, 使用了输出重定向之后, 屏幕上就不会显示 `myprog` 输出的任何信息. 如果你希望输出到文件的同时也输出到屏幕上, 你可以使用 `tee` 命令:

```
objdump -d myprog | tee output
```

使用输出重定向还能很方便地实现一些常用的功能, 例如

```
> empty                    # 创建一个名为empty的空文件
cat old_file > new_file  # 将文件old_file复制一份, 新文件名为new_file
```

如果 `myprog` 需要从键盘上读入大量数据(例如一个图的拓扑结构), 当你需要反复对 `myprog` 进行测试的时候, 你需要多次键入大量相同的数据. 为了避免这种无意义的重复键入, 你可以使用以下命令:

```
./myprog < data
```

`<` 是标准输入重定向符号, 可以将前一命令的输入重定向到文件 `data` 中. 这样, 你只需要将 `myprog` 读入的数据一次性输入到文件 `data` 中, `myprog` 就会从文件 `data` 中读入数据, 节省了大量的时间.

下面给出了一个综合使用重定向的例子:

```
time ./myprog < data | tee output
```

这个命令在运行 `myprog` 的同时, 指定其从文件 `data` 中读入数据, 并将其输出信息打印到屏幕和文件 `output` 中. `time` 工具记录了这一过程所消耗的时间, 最后你会在屏幕上看到 `myprog` 运行所需要的时间. 如果你只关心 `myprog` 的运行时间, 你可以使用以下命令将 `myprog` 的输出过滤掉:

```
time ./myprog < data > /dev/null
```

`/dev/null` 是一个特殊的文件, 任何试图输出到它的信息都会被丢弃, 你能想到这是怎么实现的吗? 总之, 上面的命令将 `myprog` 的输出过滤掉, 保留了 `time` 的计时结果, 方便又整洁.

使用Makefile管理工程

大规模的工程中通常含有几十甚至成百上千个源文件(Linux内核源码有25000+的源文件), 分别键入命令对它们进行编译是十分低效的. Linux提供了一个高效管理工程文件的工具: GNU Make. 我们首先从一个简单的例子开始, 考虑上文提到的Hello World的例子, 在 `hello.c` 所在目录下新建一个文件 `Makefile` , 输入以下内容并保存:

```
hello:hello.c
    gcc hello.c -o hello    # 注意开头的tab, 而不是空格

.PHONY: clean

clean:
    rm hello    # 注意开头的tab, 而不是空格
```

返回命令行, 键入 `make` , 你会发现 `make` 程序调用了 `gcc` 进行编译. `Makefile` 文件由若干规则组成, 规则的格式一般如下:

```
目标文件名:依赖文件列表
    用于生成目标文件的命令序列    # 注意开头的tab, 而不是空格
```

我们来解释一下上文中的 `hello` 规则. 这条规则告诉 `make` 程序, 需要生成的目标文件是 `hello` , 它依赖于文件 `hello.c` , 通过执行命令 `gcc hello.c -o hello` 来生成 `hello` 文件.

如果你连续多次执行 `make` , 你会得到"文件已经是最新版本"的提示信息, 这是 `make` 程序智能管理的功能. 如果目标文件已经存在, 并且它比所有依赖文件都要"新", 用于生成目标的命令就不会被执行. 你能想到 `make` 程序是如何进行"新"和"旧"的判断的吗?

上面例子中的 `clean` 规则比较特殊, 它并不是用来生成一个名为 `clean` 的文件, 而是用于清除编译结果, 并且它不依赖于其它任何文件. `make` 程序总是希望通过执行命令来生成目标, 但我们给出的命令 `rm hello` 并不是用来生成 `clean` 文件, 因此这样的命令总是会被执行. 你需要键入 `make clean` 命令来告诉 `make` 程序执行 `clean` 规则, 这是因为 `make` 默认执行在 `Makefile` 中文本序排在最前面的规则. 但如果很不幸地, 目录下已经存在了一个名为 `clean` 的文件, 执行 `make clean` 会得到"文件已经是最新版本"的提示. 解决这个问题的方法是在 `Makefile` 中加入一行 `PHONY: clean` , 用于指示" `clean` 是一个伪目标". 这样以后, `make` 程序就不会判断目标文件的新旧, 伪目标相应的命令序列总是会被执行.

对于一个规模稍大一点的工程, `Makefile` 文件还会使用变量, 函数, 调用Shell命令, 隐含规则等功能. 如果你希望学习如何更好地编写一个 `Makefile` , 请到互联网上搜索相关资料.

综合示例: 教务刷分脚本

使用编辑器编辑文件 `jw.sh` 为如下内容(另外由于教务网站的升级改版, 目前此脚本可能不能实现正确的功能):

```
#!/bin/bash
save_file="score" # 临时文件
semester=20102 # 刷分的学期, 20102代表2010年第二学期
jw_home="http://jwas3.nju.edu.cn:8080/jiaowu" # 教务网站首页地址
jw_login="http://jwas3.nju.edu.cn:8080/jiaowu/login.do" # 登录页面地址
jw_query="http://jwas3.nju.edu.cn:8080/jiaowu/student/studentinfo/achievementinfo.do?m
ethod=searchTermList&termCode=$semester" # 分数查询页面地址

name="09xxxxxxx" # 你的学号
passwd="xxxxxxxx" # 你的密码

# 请求jw_home地址, 并从中找到返回的cookie. cookie信息在http头中的JSESSIONID字段中
cookie=`wget -q -O - $jw_home --save-headers | \
    sed -n 's/Set-Cookie: JSESSIONID=\([0-9A-Z]\+\);.*$/\1/p'`
# 用户登录, 使用POST方法请求jw_login地址, 并在POST请求中加入userName和password
wget -q -O - --header="Cookie:JSESSIONID=$cookie" --post-data \
    "userName=${name}&password=${passwd}" "$jw_login" &> /dev/null
# 登录完毕后, 请求分数查询页面. 此时会返回html页面并输出到标准输出. 我们将输出重定向到文件"tmp"中.
wget -q -O - --header="Cookie:JSESSIONID=$cookie" "$jw_query" > tmp
# 获取分数列表. 因为教务网站的代码实在是实现得不太规整, 我们又想保留shell的风味, 所以用了比较繁琐的s
ed和awk处理. list变量中会包含课程名称的列表.
list=`cat tmp | sed -n '/<table.*TABLE_BODY.*>/,/<\/table>/p' \
        | sed '/<--/,/-->/d' | grep td \
        | awk 'NR%11==3' | sed 's/^.*>\(.*\)<.*$/\1/g'`
# 对list中的每一门课程, 都得到它的分数
for item in $list; do
    score=`cat tmp | grep -A 20 $item | awk "NR==18" | sed -n '/^.*\..*$/p'`
    score=`echo $score`
    if [[ ${#score} != 0 ]]; then # 如果存在成绩
        grep $item $save_file &>/dev/null # 查找分数是否显示过
        if [[ $? != 0 ]]; then # 如果没有显示过
        # 考虑到尝试的同学可能没有安装notify-send工具, 这里改成echo  -- yzh
            # notify-send "新成绩：$item $score" # 弹出窗口显示新成绩
            echo "新成绩：$item $score" # 在终端里输出新成绩
            echo $item >> $save_file # 将课程标记为已显示
        fi
    fi
done
```

运行这个例子需要在命令行中输入 `bash jw.sh`, 用bash解释器执行这一脚本. 如果希望定期运行这一脚本, 可以使用Linux的标准工具之一: `cron`. 将命令添加到crontab就能实现定期自动刷新.

为了理解这个例子, 首先需要一些HTTP协议的基础知识. HTTP请求实际就是来回传送的文本流——浏览器(或我们例子中的爬虫)生成一个文本格式的HTTP请求, 包括header和content, 以文本的形式通过网络传送给服务器. 服务器根据请求内容(header中包含请求的URL以及浏览器等其他信息), 生成页面并返回.

用户登录的实现, 就是通过HTTP头中header中的cookie实现的. 当浏览器第一次请求页面时, 服务器会返回一串字符, 用来标识浏览器的这次访问. 从此以后, 所有与该网站交互时, 浏览器都会在HTTP请求的header中加入这个字符串, 这样服务器就"记住"了浏览器的访问. 当完成登录操作(将用户名和密码发送到服务器)后, 服务器就知道这个cookie隐含了一个合法登录的帐号, 从而能够根据帐号信息发送成绩.

得到包含了成绩信息的html文档之后, 剩下的事情就是解析它了. 我们用了大量的 `sed` 和 `awk` 完成这件事情, 同学们不用去深究其中的细节, 只需知道我们从文本中提取出了课程名和成绩, 并且将没有显示过的成绩显示出来.

我们讲解这个例子主要是为了说明新环境下的工作方式, 以及实践Unix哲学:

- 每个程序只做一件事, 但做到极致
- 用程序之间的相互协作来解决复杂问题
- 每个程序都采用文本作为输入和输出, 这会使程序更易于使用

一个Linux老手可以用脚本完成各式各样的任务: 在日志中筛选想要的内容, 搭建一个临时HTTP服务器(核心是使用 `nc` 工具)等等. 功能齐全的标准工具使Linux成为工程师, 研究员和科学家的最佳搭档.

man快速入门

这是一个man的使用教程, 同时给出了一个如何寻找帮助的例子.

初识man

你是一只Linux菜鸟. 因为课程实验所迫, 你不得不使用Linux, 不得不使用十分落后的命令行. 实验内容大多数都要在命令行里进行, 面对着一大堆陌生的命令和参数, 这个链接中的饼图完美地表达了你的心情.

不行! 还是得认眞做实验, 不然以后连码农都当不上了! 这样的想法鞭策着你, 因为你知道, 就算是码农, 也要有适应新环境和掌握新工具的能力. "还是先去找man吧." 于是你在终端里输入 `man` , 敲了回车. 只见屏幕上输出了一行信息:

> What manual page do you want?

噢, 原来命令行也会说人话! 你明白这句话的意思, `man` 在询问你要查询什么内容. 你能查询什么内容呢? 既然 `man` 会说人话, 还是先多了解 `man` 吧. 为了告诉 `man` 你想更了解ta, 你输入

```
man man
```

敲了回车之后, `man` 把你带到了一个全新的世界. 这时候, 你又看到了一句人话了, 那是 `man` 的独白, ta告诉你, ta的眞实身份其实是

> an interface to the on-line reference manuals

接下来, ta忽然说了一大堆你听不懂的话, 似乎是想告诉你ta的使用方法. 可是你还没做好心理准备啊, 于是你无视了这些话.

寻找帮助

很快, 你已经看到"最后一行"了. 难道man的世界就这么狭小? 你仔细一看, "最后一行"里面含有一些信息:

> Manual page man(1) line 1 (press h for help or q to quit)

原来可以通过按 `q` 来离开这个世界啊, 不过你现在并不想这么做, 因为你想多了解 `man` , 以后可能会经常需要 `man` 的帮助. 为了更了解ta, 你按了 `h` .

这时你又被带到了新的世界, 世界的起点是"SUMMARY OF LESS COMMANDS", 你马上知道, 这个世界要告诉你如何使用 `man` , 你十分激动. 于是你往下看, 这句话说"带有'*'标记的命令可以在前面跟一个数, 这时命令的行为在括号里给出". 这是什么意思? 你没看懂, 还是找个带'*'的命令试试吧. 你继续往下看, 看到了两个功能和相应的命令:

- 第一个是展示帮助, 原来除了 `h` 之外, `H` 也可以看到帮助, 而且这里把帮助的命令放在第一个, 也许 `man` 想暗示你, 找到帮助是十分重要的.
- 第二个命令是退出. "哈哈, 知道怎么退出之后, 就不用通过重启来退出一个命令行程序啦", 你心想. 但你现在还是不想退出, 还是再看看其它的吧.

继续往下看, 你看到了用于移动的命令. 果然, 你还是可以在这个世界里面移动的. 第一个用于移动的功能是往下移动一行, 你看到有5种方法可以实现:

```
e  ^E  j  ^N  CR
```

`e` 和 `j` 你看懂了, 就是按 `e` 或者 `j` . 但 `^E` 是什么意思呢? 你尝试找到 `^` 的含义, 但是你没找到, 还是让我告诉你吧. 在上下文和按键有关的时候, `^` 是Linux中的一个传统记号, 它表示 `ctrl+` . 还记得Windows下 `ctrl+c` 代表复制的例子吗? 这里的 `^E` 表示 `ctrl+E` . `CR` 代表回车键, 其实 `CR` 是控制字符(ASCII码小于32的字符)的一个, 这里有一段关于控制字符的问答.

你决定使用 `j` , 因为它像一个向下的箭头, 而且它是右手食指所按下的键. 其实这点和 `vim` 的使用是类似的, 如果你不能理解为什么 `vim` 中使用 `h` , `j` , `k` , `l` 作为方向键, 这里有一个初学者的提问, 事实上, 这是一种touch typing.

你按下了 `j` , 发现画面上的信息向下滚动了一行. 你看到了 `*` , 想起了 `*` 标记的命令可以在前面跟一个数. 于是你试着输入 `10j` , 发现画面向下滚动了10行, 你第一次感觉到在这个"丑陋"的世界中也有比GUI方便的地方. 你继续阅读帮助, 并且尝试每一个命令. 于是你掌握了如何通过移动来探索 `man` 所在的世界.

继续往下翻, 你看到了用于搜索的命令. 你十分感动, 因为使用关键字可以快速定位到你关心的内容. 帮助的内容告诉你, 通过按 `/` 激活前向搜索模式, 然后输入关键字(可以使用正则表达式), 按下回车就可以看到匹配的内容了. 帮助中还列出了后向搜索, 跳到下一匹配处等功能. 于是你掌握了如何使用搜索.

探索man

你一边阅读帮助, 一边尝试新的命令, 就这样探索着这个陌生的世界. 你虽然记不住这么多命令, 但你知道你可以随时来查看帮助. 掌握了一些基本的命令之后, 你按 `q` 离开了帮助, 回到了 `man` 的世界. 现在你可以自由探索 `man` 的世界了. 你向下翻, 跳过了看不懂的 `SYNOPSIS`

小节, 在 `DESCRIPTION` 小节看到了人话, 于是你阅读这些人话. 在这里, 你看到整个manual分成9大类, 每个manual page都属于其中的某一类; 你看到了一个manual page主要包含以下的小节:

- NAME - 命令名
- SYNOPSIS - 使用方法大纲
- CONFIGURATION - 配置
- DESCRIPTION - 功能说明
- OPTIONS - 可选参数说明
- EXIT STATUS - 退出状态, 这是一个返回给父进程的值
- RETURN VALUE - 返回值
- ERRORS - 可能出现的错误类型
- ENVIRONMENT - 环境变量
- FILES - 相关配置文件
- VERSIONS - 版本
- CONFORMING TO - 符合的规范
- NOTES - 使用注意事项
- BUGS - 已经发现的bug
- EXAMPLE - 一些例子
- AUTHORS - 作者
- SEE ALSO - 功能或操作对象相近的其它命令 你还看到了对 `SYNOPSIS` 小节中记号的解释, 现在你可以回过头来看 `SYNOPSIS` 的内容了. 但为了弄明白每个参数的含义, 你需要查看 `OPTIONS` 小节中的内容.

你想起了搜索的功能, 为了弄清楚参数 `-k` 的含义, 你输入 `/-k` , 按下回车, 并通过 `n` 跳过了那些 `OPTIONS` 小节之外的 `-k` , 最后大约在第254行找到了 `-k` 的解释: 通过关键字来搜索相关功能的manual page. 在 `EXAMPLES` 小节中有一个使用 `-k` 的例子:

```
man -k printf
```

你阅读这个例子的解释: 搜索和 `printf` 相关的manual page. 你还是不太明白这是什么意思, 于是你退出 `man` , 在命令行中输入

```
man -k printf
```

并运行, 发现输出了很多和 `printf` 相关的命令或库函数, 括号里面的数字代表相应的条目属于manual的哪一个大类. 例如 `printf (1)` 是一个shell命令, 而 `printf (3)` 是一个库函数. 要访问库函数 `printf` 的manual page, 你需要在命令行中输入

```
man 3 printf
```

当你想做一件事的而不知道用什么命令的时候, `man` 的 `-k` 参数可以用来列出候选的命令, 然后再通过查看这些命令的manual page来学习怎么使用它们.

接下来, 你又开始学习 `man` 的其它功能...

开始旅程

到这里, 你应该掌握 `man` 的用法了. 你应该经常来拜访ta, 因为在很多时候, ta总能给你提供可靠的帮助.

在这个励志的故事中, 你学会了:

- 阅读程序输出的提示和错误信息
- 通过搜索来定位你关心的内容
- 动手实践是认识新事物的最好方法
- 独立寻找帮助, 而不是一有问题就问班上的大神

于是, 你就这样带着 `man` 踏上了Linux之旅...

git快速入门

光玉

想象一下你正在玩Flappy Bird, 你今晚的目标是拿到100分, 不然就不睡觉. 经过千辛万苦, 你拿到了99分, 就要看到成功的曙光的时候, 你竟然失手了! 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我? 为什么不让我存档?"

想象一下你正在写代码, 你今晚的目标是实现某一个新功能, 不然就不睡觉. 经过千辛万苦, 你终于把代码写好了, 保存并编译运行, 你看到调试信息一行一行地在终端上输出. 就要看到成功的曙光的时候, 竟然发生了段错误! 你仔细思考, 发现你之前的构思有着致命的错误, 但之前正确运行的代码已经永远离你而去了. 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我?" 你绝望地倒在屏幕前... 这时, 你发现身边渐渐出现无数的光玉, 把你包围起来, 耀眼的光芒令你无法睁开眼睛... 等到你回过神来, 你发现屏幕上正是那份之前正确运行的代码! 但在你的记忆中, 你确实经历过那悲痛欲绝的时刻... 这一切真是不可思议啊...

人生如戏, 戏如人生

人生就像不能重玩的Flappy Bird, 但软件工程领域却并非如此, 而那不可思议的光玉就是"版本控制系统". 版本控制系统给你的开发流程提供了比朋也收集的更强大的光玉, 能够让你在过去和未来中随意穿梭, 避免上文中的悲剧降临你的身上.

没听说过版本控制系统就完成实验, 艰辛地排除万难, 就像游戏通关之后才知道原来游戏可以存档一样, 其实玩游戏的时候进行存档并不是什么丢人的事情.

在实验中, 我们使用 `git` 进行版本控制. 下面简单介绍如何使用 `git` .

游戏设置

首先你得安装 `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".

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

如果你遇到了上文提到的让你悲痛欲绝的情况, 现在你可以使用光玉来救你一命了. 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.

第三视点

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, 你就可以像第三视点那样在一个世界的不同时间(一个分支的多个存档), 或者是多个平行世界(多个分支)之间来回穿梭了.

更多功能

以上介绍的是 `git` 的一些基本功能, `git` 还提供很多强大的功能, 例如使用 `git diff` 比较同一个文件在不同版本中的区别, 使用 `git bisect` 进行二分搜索来寻找一个bug在哪次提交中被引入...

其它功能的使用请参考 `git help` , `man git` , 或者在网上搜索相关资料.

x86指令系统简介

i386手册有一章专门列出了所有指令的细节, 附录中的opcode map也很有用. 我们需要实现的n86架构是x86的子集, 在这里, 我们先对x86指令系统作一些简单的梳理. 当你对x86指令系统有任何疑惑时, 请查阅i386手册, 关于指令系统的一切细节都在里面.

i386手册勘误

我们在这个页面列出目前找到的错误, 如果你在做实验的过程中也发现了新的错误, 请帮助我们更新勘误信息.

指令格式

x86指令的一般格式如下:

```
+-----------+-----------+-----------+--------+------+------+------+------------+------
-----+
|instruction| address-  |  operand- |segment |opcode|ModR/M| SIB  |displacement| immed
iate |
|  prefix   |size prefix|size prefix|override|      |      |      |            |
     |
|-----------+-----------+-----------+--------+------+------+------+------------+------
-----|
|   0 OR 1  |  0 OR 1   |   0 OR 1  | 0 OR 1 |1 OR 2|0 OR 1|0 OR 1| 0,1,2 OR 4 |0,1,2
OR 4 |
| - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - -|
|                                    number of bytes
     |
+-------------------------------------------------------------------------------------
-----+
```

除了opcode(操作码)必定出现之外, 其余组成部分可能不出现, 而对于某些组成部分, 其长度并不是固定的. 但给定一条具体指令的二进制形式, 其组成部分的划分是有办法确定的, 不会产生歧义(即把一串比特串看成指令的时候, 不会出现两种不同的解释). 例如对于以下指令:

```
100017:    66 c7 84 99 00 e0 ff ff 01 00      movw   $0x1,-0x2000(%ecx,%ebx,4)
```

其组成部分的划分如下:

```
+-----------+-----------+-----------+--------+------+------+------+------------+------
-----+
|instruction| address-  |  operand- |segment |opcode|ModR/M| SIB  |displacement| immed
iate |
|  prefix   |size prefix|size prefix|override|      |      |      |            |
     |
|-----------+-----------+-----------+--------+------+------+------+------------+------
-----|
|                            66                c7     84     99    00 e0 ff ff    01
00   |
+-----------------------------------------------------------------------------------
-----+
```

凭什么 `0x84` 要被解释成 `ModR/M` 字节呢? 这是由 `opcode` 决定的, `opcode` 决定了这是什么指令的什么形式, 同时也决定了 `opcode` 之后的比特串如何解释. 如果你要问是谁来决定 `opcode`, 那你就得去问Intel了.

在n86中, `address-size prefix` 和 `segment override prefix` 都不会用到, 因此NEMU也不需要实现这两者的功能.

编码的艺术

对于以下5个集合:

1. 所有 `instruction prefix`
2. 所有 `address-size prefix`
3. 所有 `operand-size prefix`
4. 所有 `segment override prefix`
5. 所有 `opcode` 的第一个字节

它们是两两不相交的. 这是必须的吗? 这背后反映了怎样的隐情?

另外我们在这里先给出 `ModR/M` 字节和 `SIB` 字节的格式, 它们是用来确定指令的操作数的, 详细的功能会在将来进行描述:

```
ModR/M byte
7    6    5    4    3    2    1    0
+--------+-------------+-------------+
|  mod   | reg/opcode  |     r/m     |
+--------+-------------+-------------+


SIB (scale index base) byte
7    6    5    4    3    2    1    0
+--------+-------------+-------------+
|   ss   |    index    |    base     |
+--------+-------------+-------------+
```

事实上, 一个字节最多只能区分256种不同的指令形式. 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们. x86中有主要有两种方法来解决这个问题:

- 一种方法是使用转义码(escape code). x86中有一个2字节转义码 `0x0f` , 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况). 后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时,表示需要再读入一个字节才能决定具体的指令形式.
- 另一种方法是使用 `ModR/M` 字节中的扩展opcode域来对 `opcode` 的长度进行扩充. 有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式. x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展opcode域来区分.

指令集细节

要实现一条指令, 首先你需要知道这条指令的格式和功能, 格式决定如何解释, 功能决定如何执行. 而这些信息都在instruction set page中, 因此你务必知道如何阅读它们. 我们以 `mov` 指令的opcode表为例来说明如何阅读:

```
     Opcode       Instruction       Clocks        Description

< 1> 88 /r        MOV r/m8,r8       2/2           Move byte register to r/m byte
< 2> 89 /r        MOV r/m16,r16     2/2           Move word register to r/m word
< 3> 89 /r        MOV r/m32,r32     2/2           Move dword register to r/m dword
< 4> 8A /r        MOV r8,r/m8       2/4           Move r/m byte to byte register
< 5> 8B /r        MOV r16,r/m16     2/4           Move r/m word to word register
< 6> 8B /r        MOV r32,r/m32     2/4           Move r/m dword to dword register
< 7> 8C /r        MOV r/m16,Sreg    2/2           Move segment register to r/m word
< 8> 8D /r        MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
< 9> A0           MOV AL,moffs8     4             Move byte at (seg:offset) to AL
<10> A1           MOV AX,moffs16    4             Move word at (seg:offset) to AX
<11> A1           MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
<12> A2           MOV moffs8,AL     2             Move AL to (seg:offset)
<13> A3           MOV moffs16,AX    2             Move AX to (seg:offset)
<14> A3           MOV moffs32,EAX   2             Move EAX to (seg:offset)
<15> B0 + rb ib   MOV r8,imm8       2             Move immediate byte to register
<16> B8 + rw iw   MOV r16,imm16     2             Move immediate word to register
<17> B8 + rd id   MOV r32,imm32     2             Move immediate dword to register
<18> C6 /0 ib (*) MOV r/m8,imm8     2/2           Move immediate byte to r/m byte
<19> C7 /0 iw (*) MOV r/m16,imm16   2/2           Move immediate word to r/m word
<20> C7 /0 id (*) MOV r/m32,imm32   2/2           Move immediate dword to r/m dword

------------------------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative
to the segment base. The 8, 16, and 32 refer to the size of the data. The
address-size attribute of the instruction determines the size of the
offset, either 16 or 32 bits.
------------------------------------------------------------------------------

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.
```

上表中的每一行给出了`mov`指令的不同形式, 每一列分别表示这种形式的opcode, 汇编语言格式, 执行所需周期, 以及功能描述. 由于NEMU关注的是功能的模拟, 因此`Clocks`一列不必关心. 另外需要注意的是, i386手册中的汇编语言格式都是Intel格式, 而objdump的默认格式是AT&T格式, 两者的源操作数和目的操作数位置不一样, 千万不要把它们混淆了! 否则你将会陷入难以理解的bug中.

首先我们来看`mov`指令的第一种形式:

```
     Opcode       Instruction       Clocks        Description
< 1> 88 /r        MOV r/m8,r8       2/2           Move byte register to r/m byte
```

- 从功能描述可以看出, 它的作用是"将一个8位寄存器中的数据传送到8位的寄存器或者内存中", 其中`r/m`表示"寄存器或内存".
- Opcode一列中的编码都是用十六进制表示, `88`表示这条指令的opcode的首字节

是 `0x88` , `/r` 表示后面跟一个 `ModR/M` 字节, 并且 `ModR/M` 字节中的 `reg/opcode` 域解释成通用寄存器的编码, 用来表示其中一个操作数.

- 通用寄存器的编码如下:

二进制编码	000	001	010	011	100	101	110	111
8位寄存器	AL	CL	DL	BL	AH	CH	DH	BH
16位寄存器	AX	CX	DX	BX	SP	BP	SI	DI
32位寄存器	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

- Instruction一列中, `r/m8` 表示操作数是8位的寄存器或内存, `r8` 表示操作数是8位寄存器, 按照Intel格式的汇编语法来解释, 表示将8位寄存器(`r8`)中的数据传送到8位寄存器或内存(`r/m8`)中, 这和功能描述是一致的. 至于 `r/m` 表示的究竟是寄存器还是内存, 这是由 `ModR/M` 字节的 `mod` 域决定的: 当 `mod` 域取值为 `3` 的时候, `r/m` 表示的是寄存器; 否则 `r/m` 表示的是内存. 表示内存的时候又有多种寻址方式, 具体信息参考i386手册中的表格17-3.

看明白了上面的第一种形式之后, 接下来的两种形式也就不难看懂了:

```
< 2> 89 /r        MOV r/m16,r16     2/2          Move word register to r/m word
< 3> 89 /r        MOV r/m32,r32     2/2          Move dword register to r/m dword
```

但你会发现, 这两种形式的 `Opcode` 都是一样的, 难道不会出现歧义吗? 不用着急, 还记得指令一般格式中的 `operand-size prefix` 吗? x86正是通过它来区分上面这两种形式的. `operand-size prefix` 的编码是 `0x66` , 作用是指示当前指令需要改变操作数的宽度. 在i386中, 通常来说, 如果这个前缀没有出现, 操作数宽度默认是32位; 当这个前缀出现的时候, 操作数宽度就要改变成16位 (也有相反的情况, 这个前缀的出现使得操作数宽度从16位变成32位, 但这种情况在i386中极少出现). 换句话说, 如果把一个开头为 `89 ...` 的比特串解释成指令, 它就应该被解释成 `MOV r/m32,r32` 的形式; 如果比特串的开头是 `66 89...` , 它就应该被解释成 `MOV r/m16,r16` .

操作数宽度前缀的由来

i386是从8086发展过来的. 8086是一个16位的时代, 很多指令的16位版本在当时就已经实现好了. 要踏进32位的新时代, 兼容就成了需要仔细考量的一个重要因素.

一种最直接的方法是让32位的指令使用新的操作码, 但这样1字节的操作码很快就会用光. 假设8086已经实现了200条16位版本的指令形式, 为了加入这些指令形式的32位版本, 这种做法需要使用另外200个新的操作码, 使得大部分指令形式的操作码需要使用两个字节来表示, 这样直接导致了32位的程序代码会变长. 现在你可能会觉得每条指令的长度增加一个字节也没什么大不了, 但在i386诞生的那个遥远的时代(你可以在i386手册的封面看到那个时代), 内存是一种十分珍贵的资源, 因此这种使用新操作码的方法并不是一种明智的选择.

Intel想到的解决办法就是引入操作数宽度前缀, 来达到操作码复用的效果. 当处理器工作在16位模式(实模式)下的时候, 默认执行16位版本的指令; 当处理器工作在32位模式(保护模式)下的时候, 默认执行32位版本的指令. 当某些需要的时候, 才通过操作数宽度前缀来指示操作数的宽度. 这种方法最大的好处就是不需要引入额外的操作码, 从而也不会明显地使得程序代码变长. 虽然在NEMU里面可以使用很简单的方法来模拟这个功能, 但在真实的芯片设计过程中, CPU的译码部件需要增加很多逻辑才能实现.

到现在为止, <4>-<6>三种形式你也明白了:

```
< 4> 8A /r        MOV r8,r/m8       2/4           Move r/m byte to byte register
< 5> 8B /r        MOV r16,r/m16     2/4           Move r/m word to word register
< 6> 8B /r        MOV r32,r/m32     2/4           Move r/m dword to dword register
```

<7>和<8>两种形式的mov指令涉及到段寄存器:

```
< 7> 8C /r        MOV r/m16,Sreg    2/2           Move segment register to r/m word
< 8> 8D /r        MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
```

n86去掉了段寄存器的实现, 我们可以忽略这两种形式的 `mov` 指令.

<9>-<14>这6种形式涉及到一种新的操作数记号 `moffs` :

```
< 9> A0           MOV AL,moffs8     4             Move byte at (seg:offset) to AL
<10> A1           MOV AX,moffs16    4             Move word at (seg:offset) to AX
<11> A1           MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
<12> A2           MOV moffs8,AL     2             Move AL to (seg:offset)
<13> A3           MOV moffs16,AX    2             Move AX to (seg:offset)
<14> A3           MOV moffs32,EAX   2             Move EAX to (seg:offset)
-------------------------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative
to the segment base. The 8, 16, and 32 refer to the size of the data. The
address-size attribute of the instruction determines the size of the
offset, either 16 or 32 bits.
-------------------------------------------------------------------------------
```

NOTES中给出了 `moffs` 的含义, 它用来表示段内偏移量, 但n86没有"段"的概念, 目前可以理解成"相对于物理地址0处的偏移量". 这6种形式是 `mov` 指令的特殊形式, 它们可以不通过 `ModR/M` 字节, 让 `displacement` 直接跟在 `opcode` 后面, 同时让 `displacement` 来指示一个内存地址.

<15>-<17>三种形式涉及到两种新的操作数记号:

```
<15> B0 + rb ib   MOV r8,imm8       2            Move immediate byte to register
<16> B8 + rw iw   MOV r16,imm16     2            Move immediate word to register
<17> B8 + rd id   MOV r32,imm32     2            Move immediate dword to register
```

其中:

- `+rb`, `+rw`, `+rd` 分别表示8位, 16位, 32位通用寄存器的编码. 和 `ModR/M` 中的 `reg` 域不一样的是, 这三种记号表示直接将通用寄存器的编号按数值加到 `opcode` 中 (也可以看成通用寄存器的编码嵌在 `opcode` 的低三位), 因此识别指令的时候可以通过 `opcode` 的低三位确定一个寄存器操作数.
- `ib`, `iw`, `id` 分别表示8位, 16位, 32位立即数

最后3种形式涉及到一种新的操作码记号 `/digit`, 其中 `digit` 为 `0` ~ `7` 中的一个数字:

```
<18> C6 /0 ib (*) MOV r/m8,imm8     2/2          Move immediate byte to r/m byte
<19> C7 /0 iw (*) MOV r/m16,imm16   2/2          Move immediate word to r/m word
<20> C7 /0 id (*) MOV r/m32,imm32   2/2          Move immediate dword to r/m dword

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.
```

上述形式中的 `/0` 表示一个 `ModR/M` 字节, 并且 `ModR/M` 字节中的 `reg/opcode` 域解释成扩展opcode, 其值取 `0`. 对于含有 `/digit` 记号的指令形式, 需要通过指令本身的 `opcode` 和 `ModR/M` 中的扩展opcode共同决定指令的形式, 例如 `80 /0` 表示 `add` 指令的一种形式, 而 `80 /5` 则表示 `sub` 指令的一种形式, 只看 `opcode` 的首字节 `80` 不能区分它们.

注: 在i386手册中, 这3种形式的 `mov` 指令并没有 `/0` 的记号, 在这里加入 `/0` 纯粹是为了说明 `/digit` 记号的意思. 但同时这条指令在i386中也比较特殊, 它需要使用 `ModR/M` 字节来表示一个寄存器或内存的操作数, 但 `ModR/M` 字节中的 `reg/opcode` 域却没有用到 (一般情况下, `ModR/M` 字节中的 `reg/opcode` 域要么表示一个寄存器操作数, 要么作为扩展opcode), i386手册也没有对此进行特别的说明, 直觉上的解释就是"无论 `ModR/M` 字节中的 `reg/opcode` 域是什么值, 都可以被CPU识别成这种形式的 `mov` 指令". x86是商业CPU, 我们无法从电路级实现来考证这一解释, 但对编译器生成代码来说, 这条指令中的 `reg/opcode` 域总得有个确定的值, 因此编译器一般会把这个值设成 `0`. 在NEMU的框架代码中, 对这3种形式的 `mov` 指令的实现和i386手册中给出 `Opcode` 保持一致, 忽略 `ModR/M` 字节中的 `reg/opcode` 域, 没有判断其值是否为 `0`. 如果你不能理解这段话在说什么, 你可以忽略它, 因为这并不会影响实验的进行.

到此为止, 你已经学会了如何阅读大部分的指令集细节了. 需要说明的是, 这里举的 `mov` 指令的例子并没有完全覆盖i386手册中指令集细节的所有记号, 若有疑问, 请参考i386手册.

除了opcode表之外, `Operation`, `Description` 和 `Flags Affected` 这三个条目都要仔细阅读, 这样你才能完整地掌握一条指令的功能. `Exceptions` 条目涉及到执行这条指令可能产生的异常, 由于n86不打算加入异常处理的机制, 你可以不用关心这一条目.

i386手册勘误

17.2.1 ModR/M and SIB Bytes中的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 ModR/M and SIB Bytes中的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 fo
llowing addressing modes:
+  [*] means a disp32 with no base if MOD is 00. Otherwise, [*] means disp8[EBP] or di
sp32[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!=O
F)
+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

```
@@ -?,14 +?,14 @@
 8C  /r       MOV r/m16,Sreg    2/2           Move segment register to r/m word
-8D  /r       MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
+8E  /r       MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
 A0           MOV AL,moffs8     4             Move byte at (seg:offset) to AL
 A1           MOV AX,moffs16    4             Move word at (seg:offset) to AX
 A1           MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
 A2           MOV moffs8,AL     2             Move AL to (seg:offset)
 A3           MOV moffs16,AX    2             Move AX to (seg:offset)
 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
-Ciiiiii      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OR -- Logical Inclusive OR

```
@@ -?,6 +?,6 @@
 08  /r       OR r/m8,r8        2/6       OR byte register to r/m byte
 09  /r       OR r/m16,r16      2/6       OR word register to r/m word
 09  /r       OR r/m32,r32      2/6       OR dword register to r/m dword
-0A  /r       OR r8,r/m8        2/7       OR byte register to r/m byte
-0B  /r       OR r16,r/m16      2/7       OR word register to r/m word
-0B  /r       OR r32,r/m32      2/7       OR dword register to r/m dword
+0A  /r       OR r8,r/m8        2/7       OR r/m byte to byte register
+0B  /r       OR r16,r/m16      2/7       OR r/m word to word register
+0B  /r       OR r32,r/m32      2/7       OR r/m dword to dword register
```

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 REP/REPE/REPZ/REPNE/REPNZ -- Repeat Following String Operation

```
@@ -?,13 +?,13 @@
    service pending interrupts (if any);
    perform primitive string instruction;
    CountReg <- CountReg - 1;
    IF primitive operation is CMPB, CMPW, SCAB, or SCAW
    THEN
-       IF (instruction is REP/REPE/REPZ) AND (ZF=1)
+       IF (instruction is REP/REPE/REPZ) AND (ZF=0)
        THEN exit WHILE loop
        ELSE
-          IF (instruction is REPNZ or REPNE) AND (ZF=0)
+          IF (instruction is REPNZ or REPNE) AND (ZF=1)
           THEN exit WHILE loop;
           FI;
       FI;
    FI;
```

17.2.2.11 Instruction Set Detail中的SBB -- Integer Subtraction with Borrow

```
@@ -?,6 +?,6 @@
 18  /r       SBB r/m8,r8       2/6     Subtract with borrow byte register from r/m by
te
 19  /r       SBB r/m16,r16     2/6     Subtract with borrow word register from r/m wo
rd
 19  /r       SBB r/m32,r32     2/6     Subtract with borrow dword register from r/m d
word
-1A  /r       SBB r8,r/m8       2/7     Subtract with borrow byte register from r/m by
te
-1B  /r       SBB r16,r/m16     2/7     Subtract with borrow word register from r/m wo
rd
-1B  /r       SBB r32,r/m32     2/7     Subtract with borrow dword register from r/m d
word
+1A  /r       SBB r8,r/m8       2/7     Subtract with borrow r/m byte from byte regist
er
+1B  /r       SBB r16,r/m16     2/7     Subtract with borrow r/m word from word regist
er
+1B  /r       SBB r32,r/m32     2/7     Subtract with borrow r/m dword from dword regi
ster
```

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)
```

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
+SF, ZF, PF, and CF as described above; AF and OF are 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
+SF, ZF, PF, and CF as described above; AF and OF are undefined
```

17.2.2.11 Instruction Set Detail中的SUB - Integer Subtraction

```
@@ -?,6 +?,6 @@
 28  /r      SUB r/m8,r8      2/6      Subtract byte register from r/m byte
 29  /r      SUB r/m16,r16    2/6      Subtract word register from r/m word
 29  /r      SUB r/m32,r32    2/6      Subtract dword register from r/m dword
-2A  /r      SUB r8,r/m8      2/7      Subtract byte register from r/m byte
-2B  /r      SUB r16,r/m16    2/7      Subtract word register from r/m word
-2B  /r      SUB r32,r/m32    2/7      Subtract dword register from r/m dword
+2A  /r      SUB r8,r/m8      2/7      Subtract r/m byte from byte register
+2B  /r      SUB r16,r/m16    2/7      Subtract r/m word from word register
+2B  /r      SUB r32,r/m32    2/7      Subtract r/m dword from dword register
```

17.2.2.11 Instruction Set Detail中的XOR - Logical Exclusive OR

```
@@ -?,6 +?,6 @@
 30  /r      XOR r/m8,r8      2/6      Exclusive-OR byte register to r/m byte
 31  /r      XOR r/m16,r16    2/6      Exclusive-OR word register to r/m word
 31  /r      XOR r/m32,r32    2/6      Exclusive-OR dword register to r/m dword
-32  /r      XOR r8,r/m8      2/7      Exclusive-OR byte register to r/m byte
-33  /r      XOR r16,r/m16    2/7      Exclusive-OR word register to r/m word
-33  /r      XOR r32,r/m32    2/7      Exclusive-OR dword register to r/m dword
+32  /r      XOR r8,r/m8      2/7      Exclusive-OR r/m byte to byte register
+33  /r      XOR r16,r/m16    2/7      Exclusive-OR r/m word to word register
+33  /r      XOR r32,r/m32    2/7      Exclusive-OR r/m dword to dword register
```

mov指令执行例子剖析

在PA1中, 你已经阅读了monitor部分的框架代码, 了解了NEMU执行的粗略框架. 但现在, 你需要进一步弄明白, 一条指令是怎么在NEMU中执行的, 即我们需要进一步探究 `exec_wrapper()` 函数中的细节. 为了说明这个过程, 我们举了两个 `mov` 指令的例子, 它们是NEMU自带的客户程序 `mov` 中的两条指令:

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
......
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

简单mov指令的执行

对于大部分指令来说, 执行它们都可以抽象成取指-译码-执行的指令周期. 为了使描述更加清晰, 我们借助指令周期中的一些概念来说明指令执行的过程. 我们先来剖析第一条 `mov $0x1234, %eax` 指令的执行过程.

取指(instruction fetch, IF)

要执行一条指令, 首先要拿到这条指令. 指令究竟在哪里呢? 还记得冯诺依曼体系结构的核心思想吗? 那就是"存储程序, 程序控制". 你以前听说这两句话的时候可能没有什么概念, 现在是实践的时候了. 这两句话告诉你, 指令在存储器中, 由PC(program counter, 在x86中就是 `%eip`)指出当前指令的位置. 事实上, `%eip` 就是一个指针! 在计算机世界中, 指针的概念无处不在, 如果你觉得对指针的概念还不是很熟悉, 就要赶紧复习指针这门必修课啦. 取指令要做的事情自然就是将 `%eip` 指向的指令从内存读入到CPU中. 在NEMU中, 有一个函数 `instr_fetch()` (在 `nemu/include/cpu/exec.h` 中定义)专门负责取指令的工作.

译码(instruction decode, ID)

在取指阶段, CPU拿到的是指令的比特串. 如果想知道这串比特串究竟代表什么意思, 就要进行译码的工作了. 我们可以把译码的工作作进一步的细化: 首先要决定具体是哪一条指令的哪一种形式, 这主要是通过查看指令的 `opcode` 来决定的. 对于大多数指令来说, CPU只要看指令的第一个字节就可以知道具体指令的形式了. 在NEMU中, `exec_real()` 函数首先通过 `instr_fetch()` 取出指令的第一个字节, 将其解释成 `opcode` 并记录在全局译码信息 `decoding` 中. 然后通过 `set_width()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)记录默认的

操作数宽度. 若操作数宽度结果为 `0`, 表示光看操作码的首字节, 操作数宽度还不能确定, 可能是16位或者32位, 需要通过 `decoding.is_operand_size_16` 成员变量来决定. 这其实实现了"操作数宽度前缀"的相关功能, 关于 `is_operand_size_16` 成员的更多内容会在下文进行说明.

返回后, `exec_real()` 接下来会根据取到的 `opcode` 查看 `opcode_table`, 得到指令的译码helper函数和执行helper函数, 并将其作为参数调用 `idex()` 函数来继续模拟这条指令的执行.
`idex()` 函数的原型为

```
void idex(vaddr_t *eip, opcode_entry *e);
```

它的作用是通过 `e->decode` 函数(若不为 `NULL`)对参数 `eip` 指向的指令进行译码, 然后通过 `e->execute` 函数执行这条指令.

以 `mov $0x1234, %eax` 指令为例, 首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb8`, 然后将这个字节作为opcode来索引 `opcode_table`, 发现这一指令的操作数宽度是 `4` 字节, 并通过 `set_width()` 函数记录. 接着按照同样的方式来索引 `opcode_table`, 确定取到的是一条 `mov` 指令, 它的形式是将立即数移入寄存器(move immediate to register).

事实上, 一个字节最多只能区分256种不同的指令形式. 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们. x86中有主要有两种方法来解决这个问题(在PA2中你都会遇到这两种情况):

- 一种方法是使用转义码(escape code), x86中有一个2字节转义码 `0x0f`, 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况). 后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时, 表示需要再读入一个字节才能决定具体的指令形式.
- 另一种方法是使用 `ModR/M` 字节中的扩展opcode域来对 `opcode` 的长度进行扩充. 有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式. x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展opcode域来区分.

决定了具体的指令形式之后, 译码工作还需要决定指令的操作数. 事实上, 在确定了指令的 `opcode` 之后, 指令形式就能确定下来了, CPU可以根据指令形式来确定具体的操作数. 对于 `mov $0x1234, %eax` 指令来说, 确定操作数其实就是确定寄存器 `%eax` 和立即数 `$0x1234`. 在x86中, 通用寄存器都有自己的编号, `I2r` 形式的指令把寄存器编号也放在指令的第一个字节里面, 我们可以通过位运算将寄存器编号抽取出来; 立即数存放在指令的第二个字节, 可以很容易得到它. 需要说明的是, 由于立即数是指令的一部分, 我们还是通过 `instr_fetch()` 函数来获得

它. 总的来说, 由于指令变长的特性, 指令长度和指令形式需要一边取指一边译码来确定, 而不像RISC指令集那样可以泾渭分明地处理取指和译码阶段, 因此你会在NEMU的实现中看到译码函数里面也会有 `instr_fetch()` 的操作.

执行(execute, EX)

译码阶段的工作完成之后, CPU就知道当前指令具体要做什么了, 执行阶段就是眞正完成指令的工作. 对于 `mov $0x1234, %eax` 指令来说, 执行阶段的工作就是把立即数 `$0x1234` 送到寄存器 `%eax` 中. 由于 `mov` 指令的功能可以统一成"把源操作数的值传送到目标操作数中", 而译码阶段已经把操作数都准备好了, 所以只需要针对 `mov` 指令编写一个模拟执行过程的函数即可. 这个函数就是 `exec_mov()` , 它是通过 `make_EHelper` 宏来定义的:

```
make_EHelper(mov) {
    write_operand((id_dest, &id_src->val);
    print_asm_template2(mov);
}
```

其中 `write_operand()` 函数会根据第一个参数中记录的类型的不同进行相应的写操作, 包括写寄存器和写内存. `print_asm_template2()` 是个宏, 用于输出带有两个操作数的指令的汇编形式.

更新 `%eip`

执行完一条指令之后, CPU就要执行下一条指令. 在这之前, CPU需要更新 `%eip` 的值, 让 `%eip` 指向下一条指令的位置. 为此, 我们需要确定刚刚执行完的指令的长度. 事实上, 在 `instr_fetch()` 中, 每次取指都会更新它的 `eip` 参数, 而这个参数就是在 `exec_wrapper()` 调用 `exec_real()` 时传入的 `decoding.seq_eip` . 因此当 `exec_wrapper()` 执行完一条指令调用 `update_eip()` 时, `decoding.seq_eip` 已经正确指向下一条指令了, 这时候直接更新 `%eip` 即可.

复杂mov指令的执行

对于第二个例子 `movw $0x1, -0x2000(%ecx,%ebx,4)` , 执行这条执行还是分取指, 译码, 执行三个阶段.

首先是取指. 这条mov指令比较特殊, 它的第一个字节是 `0x66` , 如果你查阅i386手册, 你会发现 `0x66` 是一个 `operand-size prefix` . 因为这个前缀的存在, 本例中的 `mov` 指令才能被CPU识别成 `movw` . NEMU使用 `decoding.is_operand_size_16` 成员变量来记录操作数宽度前缀是否出现, `0x66` 的helper函数 `operand_size()` 实现了这个功能. `operand_size()` 函数对 `decoding.is_operand_size_16` 成员变量做了标识之后, 越过前缀重新调用 `exec_real()` 函数, 此时取得了眞正的操作码 `0xc7` . 由于 `decoding.is_operand_size_16` 成员变量进行过标识, 在 `set_width()` 函数中将会确定操作数长度为 `2` 字节.

接下来是识别操作数. 根据操作码 `0xc7` 查看 `opcode_table` , 调用译码函数 `decode_mov_I2E()` , 这个译码函数又分别调用 `decode_op_I()` 和 `decode_op_rm()` 来取出操作数. 阅读代码, 你会发现 `decode_op_rm()` 最终会调用 `read_ModR_M()` 函数. 由于本例中的 `mov` 指令需要访问内存, 因此除了要识别出立即数之外, 还需要确定好要访问的内存地址. x86通过 `ModR/M` 字节来指示内存操作数, 支持各种灵活的寻址方式. 其中最一般的寻址格式是

> displacement(R[base_reg], R[index_reg], scale_factor)

相应内存地址的计算方式为

> addr = R[base_reg] + R[index_reg] * scale_factor + displacement

其它寻址格式都可以看作这种一般格式的特例, 例如

> displacement(R[base_reg])

可以认为是在一般格式中取 `R[index_reg] = 0, scale_factor = 1` 的情况. 这样, 确定内存地址就是要确定 `base_reg` , `index_reg` , `scale_factor` 和 `displacement` 这4个值, 而它们的信息已经全部编码在 `ModR/M` 字节里面了.

我们以本例中的 `movw $0x1, -0x2000(%ecx,%ebx,4)` 说明如何识别出内存地址:

```
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

根据 `mov_I2E` 的指令形式, `0xc7` 是 `opcode` , `0x84` 是 `ModR/M` 字节. 在i386手册中查阅表格17-3得知, `0x84` 的编码表示在 `ModR/M` 字节后面还跟着一个 `SIB` 字节, 然后跟着一个32位的 `displacement` . 于是读出 `SIB` 字节, 发现是 `0x99` . 在i386手册中查阅表格17-4得知, `0x99` 的编码表示 `base_reg = ECX, index_reg = EBX, scale_factor = 4` . 在 `SIB` 字节后面读出一个32位的 `displacement` , 发现是 `00 e0 ff ff` , 在小端存储方式下, 它被解释成 `-0x2000` . 于是内存地址的计算方式为

> addr = R[ECX] + R[EBX] * 4 - 0x2000

框架代码已经实现了 `load_addr()` 函数和 `read_ModR_M()` 函数 (在 `nemu/src/cpu/decode/modrm.c` 中定义), 它们的函数原型为

```
void load_addr(swaddr_t *eip, ModR_M *m, Operand *rm);
void read_ModR_M(swaddr_t *eip, Operand *rm, bool load_rm_val, Operand *reg, bool load
_reg_val);
```

它们将变量 `eip` 所指向的内存位置解释成 `ModR/M` 字节, 根据上述方法对 `ModR/M` 字节和 `SIB` 字节进行译码, 把译码结果存放到参数 `rm` 和 `reg` 指向的变量中. 虽然i386手册中的表格17-3和表格17-4内容比较多, 仔细看会发现, `ModR/M` 字节和 `SIB` 字节的编码都是有规律可

循的, 所以 `load_addr()` 函数可以很简单地识别出计算内存地址所需要的4个要素(当然也处理了一些特殊情况). 不过你现在可以不必关心其中的细节, 框架代码已经为你封装好这些细节, 并且提供了各种用于译码的接口函数.

本例中的执行阶段就是要把立即数写入到相应的内存位置. 译码阶段已经把操作数准备好了, 执行函数 `exec_mov()` 会完成数据移动的操作, 最终在 `update_eip()` 函数中更新 `%eip` .