目錄

南京大学 计算机科学与技术系 计算机系统基础 课程实验 2016

实验前阅读

最新消息

- 2017/02/25
  - ics2017 PA讲义见这里.
- 2016/12/29
  - PA4截止时间定为2017/01/11 23:59:59, 之后我们会对PA4的成绩进行统计, 把PA总成绩和理论课成绩汇总后上报到教务处, 因此截止时间之后不再接受任何提交. 另外任课老师没有允许我们公开PA部分的成绩, 如果你想知道成绩, 届时请向任课老师咨询.
  - 有偿收集课程感想, 具体请见这里. 为了及时统计课程感想的加分, 课程感想的截止

- 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(zihaoyu.x#gmail.com)

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.

```
$ docker stop ics-vm
$ docker commit ics-vm ics-intermediate-image
$ docker rm ics-vm
$ docker create -it --name=ics-vm -p 20022:22 --security-opt seccomp=unconf
$ docker start ics-vm
```

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)将指导学生实现一个功能完备(但经过简化)的x86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` 的机理. NEMU受到了QEMU的启发, 结合了GDB调试器的特性, 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4部分连贯的实验内容:

- 简易调试器
- 指令系统
- 存储管理
- 中断与I/O

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?)和历史相关的问题(Why is the linux kernel not implemented 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 4.4.7 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"上帝创造计算机"的故事.

上帝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, 就算是上帝也是花了一番功夫来准备的. 让我们来看看上帝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: 2016/09/04 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. 至于你信不信, 我反正信了.

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!

If you already have one copy of GNU/Linux, and you want to use your copy as the development environment, just use it! But if you encounter some troubles because of different GNU/Linux distribution or different version of the same distribution, please search the Internet for trouble-shooting.

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.

Download Docker-Toolbox from this website according to your host operating system, then install Docker with default settings. Note that if your host is GNU/Linux, you can install Docker by

```
apt-get install docker-engine
```

in Ubuntu or Debian. Different distribution uses different package tools. Please search the Internet for more information. Refer to Docker online Document for more information about installing Docker on GNU/Linux.

Preparing Dockerfile

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

- If your host is GNU/Linux, you can use the default terminal in the system.
- If your host is Windows or Mac, open `Docker Quickstart Terminal`. You can find it on the desktop or in the Start Menu. If it is opened for the first time, it will perform a series of operations for initialization. After the initialization, the terminal will output a success message like

  ```
  docker is configured to use the defualt machine with IP 192.168.99.100
  ```

Remember this IP address, since you will use it later. At the end, you will see the following prompt:

```
$ _
```

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
touch Dockerfile   # create an empty file with name "Dockerfile", make sure the "D" is
 capital
```

Now use the text editor in the host to open `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 32bit/debian

 # setting source list
RUN echo "deb http://mirrors.163.com/debian/ jessie main non-free contrib" > /etc/apt/
sources.list

 # installing tool `gosu`
ENV GOSU_VERSION 1.7
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recommends ca-certificates wg
et \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/$
GOSU_VERSION/gosu-$(dpkg --print-architecture)" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tianon/gosu/releases/downlo
ad/$GOSU_VERSION/gosu-$(dpkg --print-architecture).asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4
036A9C25BF357DD4 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +sx /usr/local/bin/gosu \
    && gosu nobody true

 # installing ssh server
RUN apt-get install -y openssh-server
 # defining a variable
ARG username=ics
ARG userpasswd=ics

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

 # setting login user
USER $username
WORKDIR /home/$username

 # setting `sudo` as an alias of `gosu`
RUN echo "alias sudo='gosu root'" >> ~/.bashrc

 # add search path to use `dpkg`
RUN echo "export PATH=$PATH:/usr/local/sbin:/usr/sbin:/sbin" >> ~/.bashrc

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

We choose the Debian distribution as the base image, since it can be quite small. Using 32bit system is to keep consistency with the textbook. Save the file and exit the editor.

Building Docker image

Go back to the `Docker Quickstart Terminal` and keep the Internet conntected. Type the following command to build our image:

```
# replace `jack` and `123456` with a username and password you prefer
docker build -t ics-image --build-arg username=jack --build-arg userpasswd=123456 .
```

This command will build an image with a tag `ics-image`, using the Dockerfile in the current directory(mydocker), while overwriting the variable `username` in Dockerfile. The `username` will be use in the container. Replace `jack` and `123456` in the above command with a username and password you prefer. In particular, if your host is GNU/Linux, all Docker commands should be executed with root privilege. If it is the first time you run this command, Docker will pull the base image `32bit/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       475.7 MB
32bit/debian        latest      fb434121fc77      4 hours ago         272 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 -it --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
- it is allocated with a pseudo-TTY
- it is interactive

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. The first thing to do is to determine the IP address of the machine over which the docker deamon runs.

- If your host is GNU/Linux, you will use the local loopback IP address `127.0.0.1`.
- If your host is Windows or Mac, the IP address is the one you remembered when you start the `Docker Quickstart Terminal`.

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@ip_addr
```

where `username` is the user name you told Dockerfile when building the image, `ip_addr` is the IP address mentioned above. For example:

```
ssh -p 20022 jack@192.168.99.100
```

If you are prompted with

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

enter "yes". Then enter the user password you told Dockerfile when building the image. 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 the IP address mentioned above, and change the port from `22` to `20022`. Leave other settings default, then click `Open` button. Enter the container user name and password you told Dockerfile when building the image. 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 is 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
```

However, you will receive an error message:

```
ping: icmp open socket: Operation not permitted
```

This is because `ping` requires superuser privilege to run inside a container.

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 `ping` with superuser privilege, use `sudo`. Note that this `sudo` is not the same as the one used in a real GNU/Linux system. If you are careful enough, you will find in the Dockerfile that the `sudo` command is set as an alias of gosu, which is installed while building the image. Inside a container, we should avoid using the real `sudo`, since it may cause some problems. We have already hidden the difference between them, and you do not have to worry about the details. For more detail information, please refer to the related Docker Doc.

If you find an operation requires superuser permission, append `sodu` before that operation. For example,

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

Now 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
```

This command requires root permission, too. And it requires Internet accessing. It costs some time for this command 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 gcc-doc        # GCC document
apt-get install gdb            # GNU debugger
apt-get install git            # reversion control system
apt-get install time        # we use the GNU time program instead of the build-in one
in bash
```

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 asked 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
```

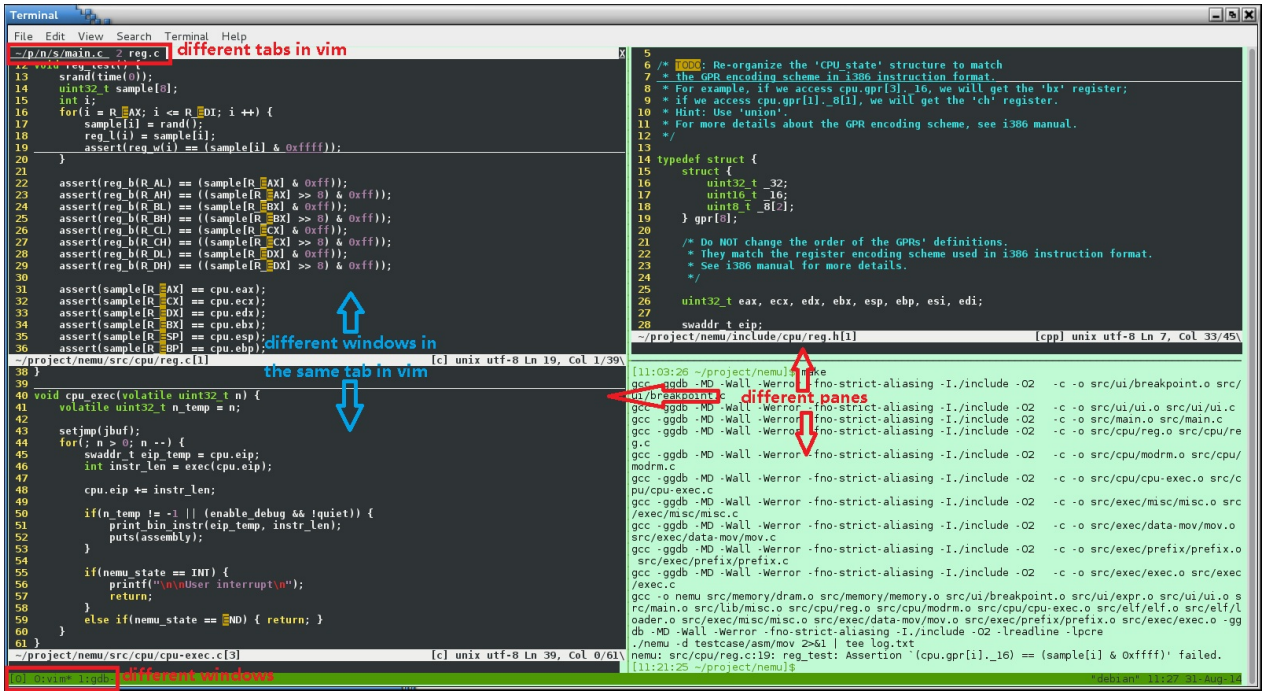
Append the following content to the file:

```
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 in the VM (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

Docker provides convenient ways to copy files between host and container. Open another `Docker Quickstart Terminal`. If your host is GNU/Linux, just open another terminal.

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

```
docker cp CONTAINER:SRC_PATH HOST_PATH
```

where

- `CONTAINER` is the name of the container
- `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 Windows path:

```
docker cp ics-vm:/home/ics/a.txt /d/temp
```

Note that you can not specify the path in Windows as `D:\temp`, since `:` in `docker cp` command has special meaning.

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

```
docker cp HOST_SRC_PATH CONTAINER:DEST_PATH
```

where

- `HOST_SRC_PATH` is the path of the host file to copy
- `CONTAINER` is the name of the container
- `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:

```
docker cp /d/myfolder ics-vm:/home/ics
```

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!

However, you may encounter error when you are trying to save the modification in `vim` in step 3 above:

```
E45: 'readonly' option is set (add ! to override)
```

According to the message, the file is read-only, but you may use `!` to force saving. Type

```
:w!
```

But you receive another error message this time (assuming the file name is `a.txt`):

```
"a.txt" E212: Can't open file for writing
```

It seems that you do not have the permission to write to this file. Type

```
:q!
```

to exit `vim` without saving. Back to shell, type

```
ls -l
```

to display detail information of the files. You will see a list like

```
total 4
-rw-r--r-- 1 root root   13 Sep  1  2016 a.txt
```

Here are some explanations of what the first column (for example, `-rw-r--r--`) of the list means. For more information about what each column means, please search the Internet.

You can see that the `a.txt` file is owned by root, and you do not have permission to modify it. This is because files copy into container will be owned by root. To change the owner of the copied files, issue the following command in the container:

```
sudo chown -R username a.txt
```

where `username` is your username in the container. This time you should modify the file successfully. So, remember to change the owner of files after copying them into the container.

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!

不过, 今年的实验配置中已经进行了相关的限制: 在container中不能直接切换到root账户.

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 https://github.com/NJU-ProjectN/ics2016
```

A directory called `ics2016` will be created. This is the project directory for PAs. Details will be explained in PA1.

Git configuration and usage

Issue the following commands:

```
git config --global user.name "151220000-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`.

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

Enter the project directory `ics2016`. 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 `config/Makefile.git`:

```
STU_ID=151220000            # your student ID
```

Run

```
git status
```

to see those files modified from the last commit:

```
On branch ics2016
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:   config/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/config/Makefile.git b/config/Makefile.git
index c9b1708..b7b2e02 100644
--- a/config/Makefile.git
+++ b/config/Makefile.git
@@ -1,4 +1,4 @@
-STU_ID = 151220000
+STU_ID = 151221234

  # 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. 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 `config/Makefile.git`, and you will find that `STU_ID` is still unchanged! By issuing `git log`, you will find the commit log you just created 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 above workflow 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

Before compiling the project, you should install the readline library:

```
apt-get install libreadline-dev
```

Now enter the project 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-ics2016 <tracer@njuics.org>
Date:   Sun Jul 26 14:30:31 2016 +0800

    > compile NEMU
    151220000
    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/ics2016/.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. Then issue

```
make submit
```

This command does 2 things:

1. Clean all unnecessary files for submission
2. 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 - 洞察世界的视点: 简易调试器

世界诞生的故事 - 第一章

上帝已经准备好了创造世界的工具, 同时也已经创造了计算机世界的原型, 包括寄存器和存储器, 这个世界已经可以很简单地运转起来了. 但在继续创造世界之前, 上帝还是有点不放心, 如何知道创造的世界有没有按照上帝设定的法则来运转呢? 为了解决自己的担忧, 上帝想到了一种办法.

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 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 - 2016/09/11 23:59:59
- 阶段2: 实现调试功能的表达式求值 - 2016/09/18 23:59:59
- 最后阶段: 实现所有要求, 提交完整的实验报告 - 2016/09/25 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, 但由于debian虚拟机中缺少GUI, 因此要运行LiteNES是一件比较困难的事情). 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!

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?

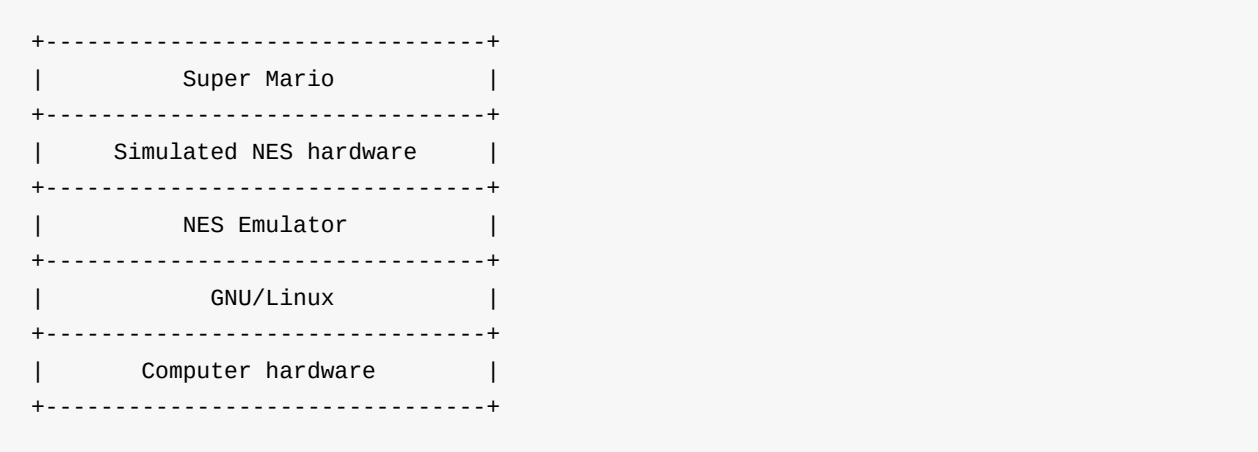
事实上, NEMU就是在做类似的事情! 它模拟了一个x86的世界(准确地说, 是x86的一个子集), 你可以在这个x86世界中执行程序. 换句话说, 你将要编写一个用来执行其它程序的程序! 为了更好地理解NEMU的功能, 下面将

- 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

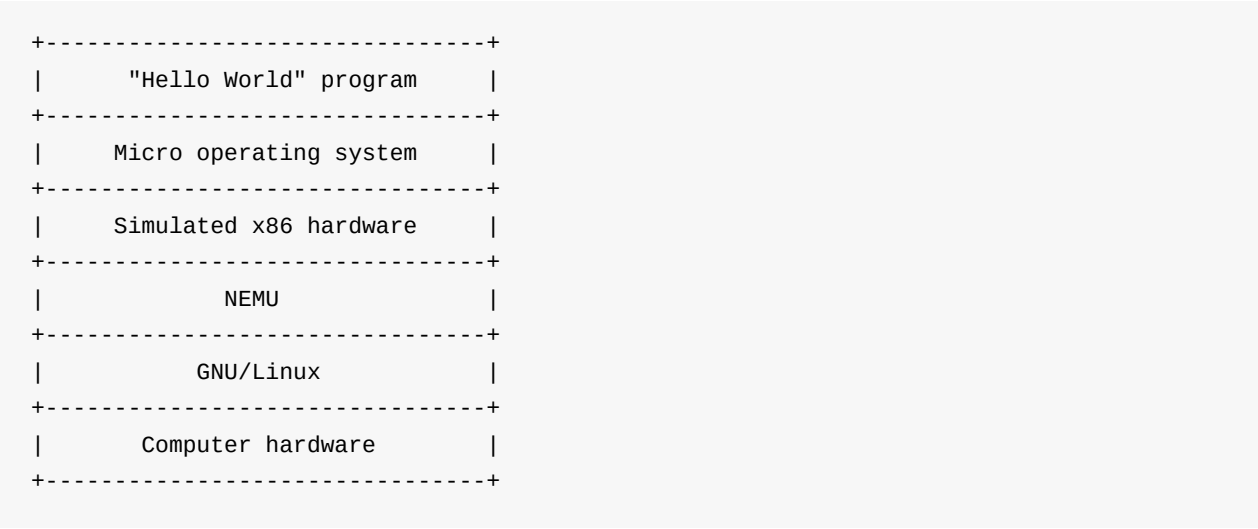
这三种情况进行比较.

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



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.



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. 不过, 我们还需要先在模拟出的x86硬件之上运行一个微型操作系统, 之后才让Hello World程序在这个微型操作系统上面运行. 为了方便叙述, 我们将在NEMU中运行的程序称为"用户程序".

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的真实情况有点复杂, 有兴趣的同学可以参考虚拟机和container的区别和Docker在Windows中的工作方式.

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C语言的数据对象来模拟, 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中使用结构体来模拟通用寄存器, 那么对这个结构体进行读写则相当于对通用寄存器进行读写.

这些数据对象之间相互协助的威力会让你感到吃惊! NEMU不仅仅能运行Hello World这样的小程序, 在PA的最后, 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, 你会觉得红白机模拟器不再是一件神奇的玩意儿, 甚至你会发现编写一个属于自己的红白机模拟器不再是遥不可及!

让我们来开始这段激动人心的旅程吧! 但请不要忘记:

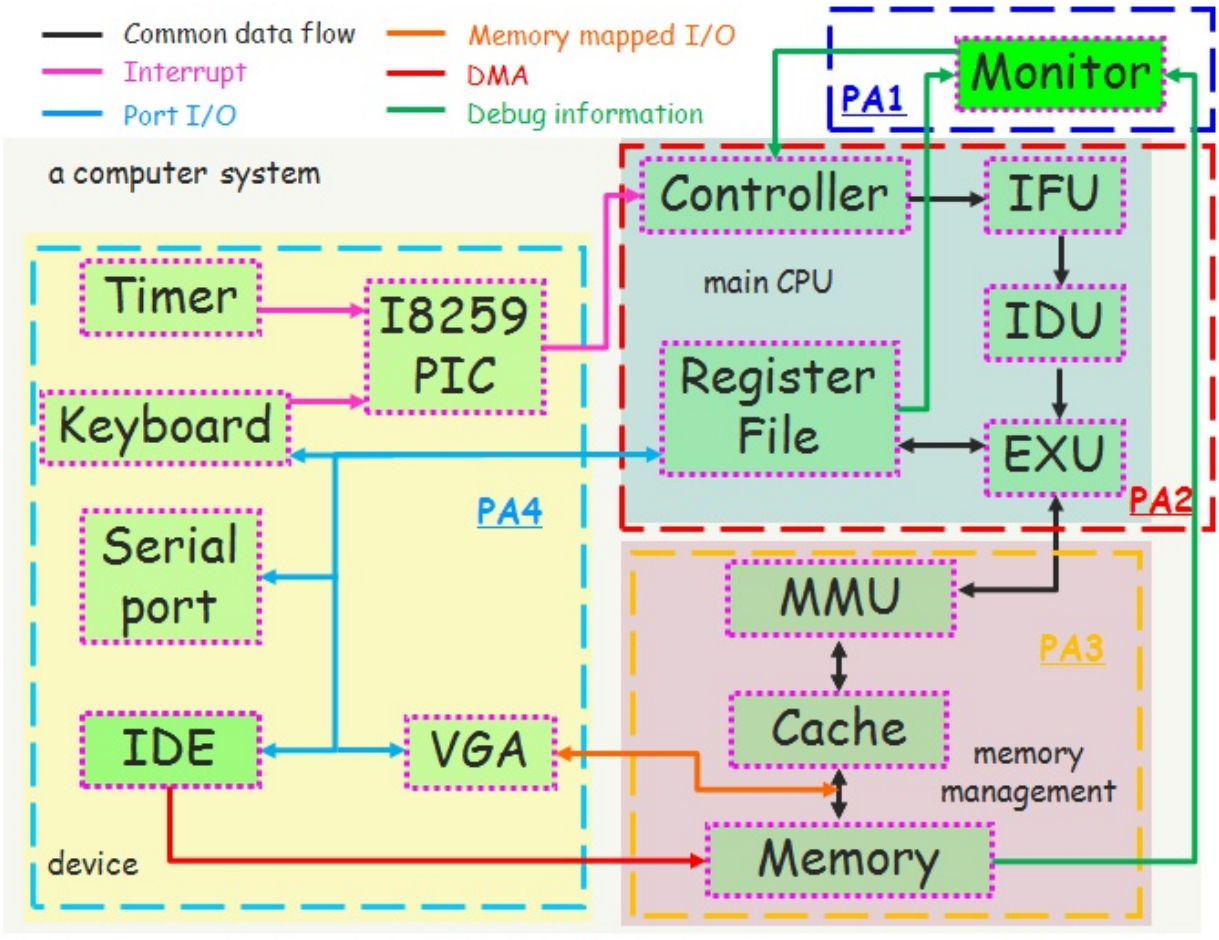
- 机器永远是对的
- 未测试代码永远是错的
- RTFM

RTFSC

拿到框架代码之后, 第一件事就是RTFSC. 不过框架代码内容众多, 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.

```
ics2016
├── config              # 包含Makefile的一些配置
├── game              # 包含打字小游戏和仙剑奇侠传两个游戏
├── kernel              # 微型操作系统内核
├── lib-common          # 公用的库
├── Makefile          # 提供工程的构建, 运行, 测试, 打包等功能
├── nemu              # NEMU
├── testcase          # 测试用例
└── test.sh             # 测试脚本
```

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. 下图给出了NEMU的结构.



NEMU主要由4个模块构成: monitor, CPU, 存储管理, 设备, 它们依次作为4个PA关注的主题. 其中, CPU, 存储管理, 设备这3个模块共同组成一个虚拟的计算机系统, 程序可以在其上运行; 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虽然不属于虚拟计算机系统, 但对PA来说, 它是必要的. 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                   # 译码相关
│   │   ├── exec                    # 执行相关
│   │   └── reg.h                    # 寄存器结构体的定义
│   ├── 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
│   │   └── memory.h                # 访问内存相关
│   ├── misc.h                       # 杂项
│   ├── monitor
│   │   ├── monitor.h
│   │   └── watchpoint.h            # 监视点相关
│   └── nemu.h
├── Makefile.part                    # 指示NEMU的编译和链接
└── src                              # 源文件
    ├── cpu
    │   ├── decode                   # 译码相关
    │   ├── exec                    # 执行相关
    │   └── reg.c                    # 寄存器相关
    ├── device                       # 设备相关
    ├── lib
    │   └── logo.c                   # "i386"的logo
    ├── main.c                       # 你知道的...
    ├── memory
    │   ├── burst.h
    │   ├── dram.c                   # DRAM工作方式的模拟
    │   └── memory.c                # 访问内存的接口函数
    └── monitor
        ├── cpu-exec.c               # 指令执行的主循环
        ├── debug                    # 简易调试器相关
        │   ├── elf.c                # ELF文件格式的解析
        │   ├── 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, 代码中完整实现了 `mov` 指令的功能(部分特殊的 `mov` 指令并未实现, 例如 `mov %eax, %cr3`), 并附带一个 `mov` 指令的用户程序(`testcase/src/mov.S`). 另外, 部分代码中会涉及一些硬件细节(例如 `nemu/src/cpu/decode/modrm.c`)和文件格式(例如 `nemu/src/monitor/debug/elf.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, 会进行一些和monitor相关的初始化工作, 包括打开日志文件, 读入ELF文件的符号表和字符串表, 编译正则表达式, 初始化监视点结构池. 这些初始化工作你几乎一个也看不懂, 但不要紧, 因为你现在根本不必关心它们的细节, 因此可以继续阅读代码. 之后代码会对寄存器结构的实现进行测试, 测试通过后会调用 `restart()` 函数(在 `nemu/src/monitor/monitor.c` 中定义), 它模拟了"计算机启动"的功能, 主要是进行一些和"计算机启动"相关的初始化工作, 包括

- 初始化ramdisk
- 读入入口代码entry
- 设置 `%eip` 的初值
- 初始化DRAM的模拟(目前不必关心)

在一个完整的计算机中, 程序的可执行文件应该存放在磁盘里, 但目前我们并没有实现磁盘的模拟, 因此NEMU先把内存开始的位置附近的一段区间作为磁盘来使用, 这样的磁盘有一个专门的名称, 叫ramdisk. 目前的ramdisk只用于存放将要在NEMU中运行的程序的可执行文件, 这个文件是运行NEMU的一个参数, 在运行NEMU的命令中指定, `init_ramdisk()` 函数把这个文件从真实磁盘读入到ramdisk.

入口代码entry的引入其实是一种简化. 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程序entry读入到一个固定的内存位置 `0x100000` , 并把这个内存位置作为 `%eip` 的初值. 这时内存的布局如下:

```
0                                0x100000
-------------------------------------------------------------------------
|                 |                   |                 |
|      ramdisk    |                   |      entry      |
|     (unused)    |                   |                 |
-------------------------------------------------------------------------
```

从0开始的一段物理内存被当作ramdisk来使用, 但这一阶段在NEMU中运行的程序并不需要使用ramdisk, 因此这段区间目前暂时不使用. 从 `0x100000` 开始的物理内存用于存放entry, 现在entry的内容就是将要在NEMU中运行的程序, NEMU的模拟执行将从这里开始. 在PA2中, 我们将会把kernel作为entry, kernel负责从ramdisk中读出将要运行的程序, 并把它加载到正确的内存位置.

`restart()` 函数执行完毕后, NEMU会进入用户界面主循环 `ui_mainloop()` (在 `nemu/src/monitor/debug/ui.c` 中定义), 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. 键入 `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()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)的功能是让CPU执行一条指令. 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.

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是定义在 `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inv的含义是invalid, 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()` 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.
- 用户程序执行了 `nemu_trap` 指令. 这是一条特殊的指令, 机器码为 `0xd6` . x86中并没有这条指令, 它是为了指示程序的结束而加入的. 在后续的实验中, 我们还会使用这条指令实现一些无法通过程序本身完成的, 需要NEMU帮助的功能.

退出 `cpu_exec()` 之后, NEMU将返回到 `ui_mainloop()` , 等待用户输入命令. 但为了再次运行程序, 你需要退出NEMU, 然后重新运行.

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?

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.
- 访问模拟的内存
  - 在程序运行的过程中, 总是使用 `swaddr_read()` 和 `swaddr_write()` 访问模拟的内存. swaddr, lnaddr, hwaddr分别代表虚拟地址, 线性地址, 物理地址, 这些概念将在PA3中用到, 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.

大致弄清楚NEMU的工作方式之后, 你就可以开始做PA1了. 需要注意的是, 上面描述的只是一个十分大概的过程, 如果你对这个过程有疑问, RTFSC.

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.

第三视点: 简易调试器

简易调试器是monitor的一项重要功能. 我们知道NEMU是一个用来执行其它用户程序的程序, 这意味着, NEMU可以随时了解用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调式NEMU的时候, 你将很难在NEMU中运行的用户程序中设置断点, 但对于NEMU来说, 这是一件不太困难的事情.

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
打印栈帧链(3)	`bt`	`bt`	打印栈帧链

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
- (3) 在PA2中实现

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.

寄存器结构体

寄存器是CPU中一个重要的组成部分, 在CPU中进行运算所用到的数据和结果都会存放在寄存器中. i386手册的第2.3节对i386中所用寄存器进行了简单的介绍. 在现阶段的NEMU中, 我们只会用到其中的两类寄存器: 首先是通用寄存器. 通用寄存器的结构如下图所示:

```
 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数据时能提供一些便利. 至于如何实现这样的结构, 当然是难不倒聪明的你啦!

第二类在NEMU中用到的寄存器就是 `EIP` , 也就是大名鼎鼎的程序计数器(Program Counter). 你在程序设计课上已经知道, 程序执行就是执行一行一行的C代码; 在计算机硬件的世界里, 程序执行也有类似的表现, 就是执行一条一条的指令. 但计算机怎么知道程序已经执行到哪里呢? 肩负着这一重要使命的就是程序计数器了, i386给它起了一个名字叫 `EIP` .

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

可别小看了这个32位的家伙, 你会在PA2中频繁地跟它打交道. 随着实验的推进, 更多的寄存器会加入到NEMU中.

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.

在 `nemu/src/cpu/reg.c` 中有一个 `reg_test()` 函数, 它会生成一些随机的数据, 来测试你的实现是否正确, 若不正确, 将会触发assertion fail. 实现正确之后, NEMU将不会在 `reg_test()` 中触发assertion fail, 同时会输出NEMU的命令提示符:

```
(nemu)
```

输入 `c` 之后, NEMU将会运行一个由 `mov` 指令组成的用户程序, 最后输出如下信息:

```
nemu: HIT GOOD TRAP at eip = 0x001002b1
```

这说明程序成功地结束运行. 键入 `q` 退出NEMU. 此时可以打开 `log.txt` 文件查看刚才程序执行的每一条指令.

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, 和用户程序的objdump结果进行对比(此时用户程序是 `mov` , 其dump结果在 `obj/testcase/mov.txt` 中), 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.

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 `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6000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)` , 这是因为表达式求值的核心功能和处理上述问题是不耦合的, 说得通俗点, 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
 pair of parentheses
"(4 + 3)) * ((2 - 1)"    // false, bad expression
"(4 + 3) * (2 - 1)"       // false, the leftmost '(' and the rightmost ')' are not ma
tched
```

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 `swaddr_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, 因此你会在PA2中实现它.

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

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 `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IA-32体系结构的变种my-IA-32, 除了 `int3` 指令的长度变成了2个字节之外, 其余指令和IA-32相同. 在my-IA-32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?

武功秘笈: 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:

- 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 `master` 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 - 不停计算的机器: 指令系统

世界诞生的故事 - 第二章

上帝已经创造了存储器, 但计算机还是不能计算. 为此, 上帝打算创造运算器, 向这个世界施以让人类叹为观止的神奇魔法 -- 计算.

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: 60小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 实现6个helper函数, 在NEMU中运行第一个C程序 - 2016/10/02 23:59:59
- 阶段2: 实现更多的指令 - 2016/10/16 23:59:59
- 阶段3: 完善简易调试器 - 2016/10/23 23:59:59
- 阶段4: 实现loader - 2016/10/30 23:59:59
- 最后阶段: 实现所有要求, 提交完整的实验报告 - 2016/11/06 23:59:59

提交说明: 见这里

武功秘笈阅读指南: x86指令系统简介

PA2的任务是在NEMU中实现x86指令系统(的子集), 你不可避免地需要了解x86指令系统的细节. i386手册有一章专门列出了所有指令的细节, 你需要在完成PA2的过程中反复阅读这一章的内容, 附录中的opcode map也很有用. 在这一小节中, 我们对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:

```
1000fe:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
100105:    ff 01 00
```

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.

在我们的PA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     |
+--------+-------------+-------------+
```

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RICS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RICS世界诞生, 到与CISC世界融为一体的故事, 体会一下RICS的诞生对计算机体系结构发展的里程碑意义.

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长度. 在IA-32中, 通常来说, 如果这个前缀没有出现, 操作数长度默认是32位; 当这个前缀出现的时候, 操作数长度就要改变成16位(也有相反的情况, 这个前缀的出现使得操作数长度从16位变成32位, 但这种情况在IA-32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` .

操作数长度前缀的由来

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长度前缀, 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, 才通过操作数长度前缀来指示操作数的长度. 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. 虽然我们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
```

现在NEMU中并没有加入段寄存器的功能, 因此这两种形式的 `mov` 指令还没有实现. 但现在你可以先忽略它们, 你将会在PA3中加入这两种形式.

<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NEMU现在还没有"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NEMU不打算实现异常处理的机制, 你可以不用关心这一条目.

RTFSC(2)

上一小节中的内容全部出自i386手册, 现在我们结合框架代码来理解上面的内容.

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()` 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, 它们是框架代码自带的用户程序mov(`testcase/src/mov.S`)中的两条指令(mov的反汇编结果在 `obj/testcase/mov.txt` 中):

```
100014:    b9 00 80 00 00            mov    $0x8000,%ecx
......
1000fe:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
100105:    ff 01 00
```

helper函数命名约定

对于每条指令的每一种形式, NEMU分别使用一个helper函数来模拟它的执行. 为了易于维护, 框架代码对helper函数的命名有一种通用的形式:

> 指令_形式_操作数后缀

例如对于helper函数 `mov_i2rm_b()` , 它模拟的指令是 `mov` , 形式是 `把立即数移动到寄存器或内存` , 操作数后缀是 `b` , 表示操作数长度是8位. 在PA2中, 你需要实现很多helper函数, 这种命名方式可以很容易地让你知道一个helper函数的功能.

一个特殊的操作数后缀是 `v` , 表示variant, 意味着光看操作码的首字节, 操作数长度还不能确定, 可能是16位或者32位, 需要通过 `ops_decoded.is_operand_size_16` 成员变量来决定. 其实这种helper函数做的事情, 就是在根据指令是否出现 `operand-size prefix` 来确定操作数长度, 从而决定最终的指令形式, 调用最终的helper函数来模拟指令的执行.

也有一些指令不需要区分形式和操作数后缀, 例如 `int3` , 这时可以直接用指令的名称来命名其helper函数. 如果你觉得上述命名方式不易看懂, 你可以使用其它命名方式, 我们不做强制要求.

简单mov指令的执行

我们先来剖析第一条 `mov $0x8000, %ecx` 指令的执行过程. 当NEMU执行到这条指令的时候(eip = 0x100014), 当前 `%eip` 的值被作为参数送进 `exec()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)中. 其中 `make_helper` 是个宏, 你需要编写一系列helper函数来模拟指令执行的过程, 而 `make_helper` 则定义了helper函数的声明形式:

```
#define make_helper(name) int name(swaddr_t eip)
```

从 `make_helper` 的定义可以看到, helper函数都带有一个参数 `eip` , 返回值类型都是 `int` . 从抽象的角度来说, 一个helper函数做的事情就是对参数 `eip` 所指向的内存单元进行某种操作, 然后返回这种操作涉及的代码长度. 例如 `exec()` 函数的功能是"执行参数 `eip` 所指向的指令, 并返回这条指令的长度"; 框架代码中还定义了一些获取指令中的立即数的helper函数, 它们的功能是"获取参数 `eip` 所指向的立即数, 并返回这个立即数的长度".

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)

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/helper.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()` 函数首先通过 `instr_fetch()` 取出指令的第一个字节, 然后根据取到的这个字节查看 `opcode_table` , 得到指令的helper函数, 从而调用这个helper函数来继续模拟这条指令的执行. 以 `mov $0x8000, %ecx` 指令为例, 首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb9` , 然后根据这个字节来索引 `opcode_table` , 找到了一个名为 `mov_i2r_v` 的helper函数, 这样就可以确定取到的是一条 `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. 我们还是以 `mov $0x8000, %ecx` 来说明这个过程, 但在这之前, 我们需要作一些额外的说明. 在上文的描述中, 我们通过这条指令的第一个字节 `0xb9` 找到了 `mov_i2r_v()` 的helper函数, 这个helper函数的定义在 `nemu/src/cpu/exec/data-mov/mov.c` 中:

```
make_helper_v(mov_i2r)
```

其中 `make_helper_v()` 是个宏, 它在 `nemu/include/cpu/exec/helper.h` 中定义:

```
#define make_helper_v(name) \
    make_helper(concat(name, _v)) { \
        return (ops_decoded.is_operand_size_16 ? concat(name, _w) : concat(name, _l))
(eip); \
    }
```

进行宏展开之后, `mov_i2r_v()` 的函数体如下:

```
int mov_i2r_v(swaddr_t eip) {
    return (ops_decoded.is_operand_size_16 ? mov_i2r_w : mov_i2r_l) (eip); \
}
```

它的作用是根据全局变量 `ops_decoded` (在 `nemu/src/cpu/decode/decode.c` 中定义)中的 `is_operand_size_16` 成员变量来决定操作数的长度, 然后从复用 `opcode` 的两个helper函数中选择一个进行调用. 全局变量 `ops_decoded` 用于存放一些译码的结果, 其中的 `is_operand_size_16` 成员和指令中的 `operand-size prefix` 有关, 而且会经常用到, 框架代码把类似于 `mov_i2r_v()` 这样的功能抽象成一个宏 `make_helper_v()` , 方便代码的编写. 关于 `is_operand_size_16` 成员的更多内容会在下文进行说明. 根据指令 `mov $0x8000, %ecx` 的功能, 它的操作数长度为4字节, 因此这里会调用 `mov_i2r_l()` 的helper函数. `mov_i2r_l()` 的helper函数在 `nemu/src/cpu/exec/data-mov/mov-template.h` 中定义, 它的函数体是通过宏展开得到的, 在这里我们直接给出宏展开的结果, 关于宏的使用请阅读相应的框架代码:

```
int mov_i2r_l(swaddr_t eip) {
    return idex(eip, decode_i2r_l, do_mov_l);
}
```

其中 `idex()` 函数的原型为

```
int idex(swaddr_t eip, int (*decode)(swaddr_t), void (*execute) (void));
```

它的作用是通过 `decode` 函数对参数 `eip` 指向的指令进行译码, 然后通过 `execute` 函数执行这条指令.

对于 `mov $0x8000, %ecx` 指令来说, 确定操作数其实就是确定寄存器 `%ecx` 和立即数 `$0x8000` . 在x86中, 通用寄存器都有自己的编号, `mov_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. 对于 `mov_i2r` 形式的指令来说, 立即数存放在指令的第二个字节, 可以很容易得到它. 然而很多指令都具有i2r的形式, 框架代码提供了几个函数(`decode_i2r_l()` 等), 专门用于进行对i2r形式的指令的译码工作. `decode_i2r_l()` 函数会把指令中的立即数信息和寄存器信息分别记录在全局变量 `ops_decoded` 中的 `src` 成员和 `dest` 成员中, `nemu/include/cpu/helper.h` 中定义了两个宏 `op_src` 和 `op_dest` , 用于方便地访问这两个成员.

立即数背后的故事

在 `decode_i_l()` 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 `0x8000` 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
+----+----+----+----+
| 00 | 80 | 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 `00 80 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 `0x8000` , 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.

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, 执行阶段就是真正完成指令的工作. 对于 `mov $0x8000, %ecx` 指令来说, 执行阶段的工作就是把立即数 `$0x8000` 送到寄存器 `%ecx` 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 `do_mov_l()` , 它是通过在 `nemu/src/cpu/exec/data-mov/mov-template.h` 中定义的 `do_execute()` 函数进行宏展开后得到的:

```
static void do_mov_l() {
    write_operand_l((&ops_decoded.dest), (&ops_decoded.src)->val);
    Assert(snprintf(assembly, 80, "movl %s,%s", (&ops_decoded.src)->str, (&ops_decoded
.dest)->str) < 80, "buffer overflow!");
}
```

其中 `write_operand_l()` 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.

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. 在NEMU中, 指令的长度是通过helper函数的返回值进行传递的, 最终会传回到 `cpu_exec()` 函数中, 完成对 `%eip` 的更新.

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 `ops_decoded.is_operand_size_16` 成员变量来记录操作数长度

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 `ops_decoded.is_operand_size_16` 成员变量做了标识之后, 越过前缀重新调用 `exec()` 函数, 此时取得了真正的操作码 `0xc7` , 通过查看 `opcode_table` 调用了helper函数 `mov_i2rm_v()` . 由于 `ops_decoded.is_operand_size_16` 成员变量进行过标识, 在 `mov_i2rm_v()` 中将会调用 `mov_i2rm_w()` 的helper函数. 到此为止才识别出本例中的指令是一条 `movw` 指令.

接下来是识别操作数. 同样地, 我们先给出 `mov_i2rm_w()` 函数的宏展开结果:

```
int mov_i2rm_w(swaddr_t eip) {
    return idex(eip, decode_i2rm_w, do_mov_w);
}
```

这里使用 `decode_i2rm_w()` 函数来进行译码的工作, 阅读代码, 你会发现它最终会调用 `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:

```
1000fe:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
100105:    ff 01 00
```

根据 `mov_i2rm` 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

```
int load_addr(swaddr_t eip, ModR_M *m, Operand *rm);
int read_ModR_M(swaddr_t eip, Operand *rm, Operand *reg);
```

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, 同时返回这一译码过程所需的字节数. 在上面的例子中, 为了计算出内存地址, 用到了 `ModR/M` 字节, `SIB` 字节和32位的 `displacement` , 总共6个字节, 所以 `read_ModR_M()` 返回6. 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, 这是通过 `do_mov_w()` 函数实现的. 执行结束后返回指令的长度, 最终在 `cpu_exec()` 函数中更新 `%eip` .

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_i2rm` 和 `add_rm2r` 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_i2rm` 和 `sub_i2rm` 等, 它们的译码阶段都是识别出一个立即数和一个 `rm` 操作数.
- 对于同一条指令同一种形式的不同长度, 它们的译码阶段和执行阶段都是非常类似的. 例如 `add_i2rm_b` , `add_i2rm_w` 和 `add_i2rm_l` , 它们都是识别出一个立即数和一个 `rm` 操作数, 然后把相加的结果存入 `rm` 操作数.

这意味着, 如果独立实现每条指令不同形式不同长度的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, 执行和操作数长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, 它把译码和执行的helper函数的指针作为参数, 依次调用它们, 这样我们就可以分别编写译码和执行的helper函数了. 实现操作数长度和译码, 执行这两者之间的解耦的是宏 `DATA_BYTE` , 它把不同操作数长度的共性抽象出来, 编写一份模板, 分别进行3次实例化, 就可以得到3分不同操作数长度的代码.

为了实现进一步的封装和抽象, 框架代码中使用了大量的宏, 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/exec/template-start.h	
SUFFIX	表示 DATA_BYTE 相应长度的后缀字母, 为 b, w, l 其中之一
DATA_TYPE	表示 DATA_BYTE 相应长度的无符号数据类型, 为 uint8_t, uint16_t, uint32_t 其中之一
DATA_TYPE_S	表示 DATA_BYTE 相应长度的有符号数据类型, 为 int8_t, int16_t, int32_t 其中之一
REG(index)	编码为 index, 长度为 DATA_BYTE 的GPR
REG_NAME(index)	编码为 index, 长度为 DATA_BYTE 的GPR的名称
MEM_R(addr)	从内存位置 addr 读出 DATA_BYTE 字节的数据
MEM_W(addr)	把长度为 DATA_BYTE 字节的数据 data 写入内存位置 addr
OPERAND_W(op, src)	把结果 src 写入长度为 DATA_BYTE 字节的目的操作数 op
MSB(n)	取出长度为 DATA_BYTE 字节的数据 n 的MSB位
nemu/include/cpu/helper.h	
make_helper(name)	名为 name 的helper函数的原型说明
op_src	全局变量 ops_decoded 中源操作数成员的地址
op_src2	全局变量 ops_decoded 中2号源操作数成员的地址
op_dest	全局变量 ops_decoded 中目的操作数成员的地址
nemu/include/cpu/exec/helper.h	
make_helper_v(name)	名为 name_v 的helper函数的定义, 用于根据指令的操作数长度前缀进一步确定调用哪一个helper函数
do_execute	用于模拟指令真正的执行操作的函数名

`make_instr_helper(type)`	名为 `指令_形式_操作数后缀` 的helper函数的定义, 其中 `type` 为指令的形式, 通过调用 `idex()` 函数来进行指令的译码和执行
`print_asm(...)`	将反汇编结果的字符串打印到缓冲区 `assembly` 中
`print_asm_template1()`	打印单目操作数指令的反汇编结果
`print_asm_template2()`	打印双目操作数指令的反汇编结果
`print_asm_template3()`	打印三目操作数指令的反汇编结果
`nemu/src/cpu/exec/*/*-template.h`	
`instr`	指令的名称, 被 `do_execute` , `make_instr_helper(type)` 和 `print_asm_template?()` 使用

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. 因此如果你对宏不太熟悉, 可能会对阅读框架代码带来困难. 我们准备了命令, 用来专门生成 `nemu/src/cpu/decode` 目录和 `nemu/src/cpu/exec` 目录下源文件的预处理结果, 键入

```
make cpp
```

会在这些目录中生成 `.i` 的预处理结果, 它们可以帮助你阅读框架代码, 调试与宏相关的错误. 键入

```
make clean-cpp
```

可以移除这些预处理结果.

源文件组织

最后我们来聊聊 `nemu/src/cpu/exec` 目录下源文件的组织方式.

```
nemu/src/cpu/exec
├── all-instr.h
├── arith
│   └── ...
├── data-mov
│   ├── mov.c
│   ├── mov.h
│   ├── mov-template.h
│   ├── xchg.c
│   ├── xchg.h
│   └── xchg-template.h
├── exec.c
├── logic
│   └── ...
├── misc
│   ├── misc.c
│   └── misc.h
├── prefix
│   ├── prefix.c
│   └── prefix.h
├── special
│   ├── special.c
│   └── special.h
└── string
    ├── rep.c
    └── rep.h
```

- `exec.c` 中定义了操作码表 `opcode_table` 和helper函数 `exec()` , `exec()` 根据指令的 `opcode` 首字节查阅 `opcode_table` , 并调用相应的helper函数来模拟相应指令的执行. 除此之外, 和2字节转义码相关的2字节操作码表 `_2byte_opcode_table` , 以及各种指令组表也在 `exec.c` 中定义.
- `all-instr.h` 中列出了所有用于模拟指令执行的helper函数的声明, 这个头文件被 `exec.c` 包含, 这样就可以在 `exec.c` 中的 `opcode_table` 直接使用各种helper函数了.
- 除了 `exec.c` 和 `all-instr.h` 两个源文件之外, 目录下还有若干子目录, 这些子目录分别存放用于模拟不同功能的指令的源文件. i386手册根据功能对所有指令都进行了分类, 框架代码中对相关文件的管理参考了手册中的分类方法(其中 `special` 子目录下模拟了和NEMU相关的功能, 与i386手册无关). 以 `nemu/src/cpu/exec/data-mov` 目录下与 `mov` 指令相关的文件为例, 我们对其文件组织进行进一步的说明:
  - `mov.h` 中列出了用于模拟 `mov` 指令所有形式的helper函数的声明, 这个头文件被 `all-instr.h` 包含.
  - `mov-template.h` 是 `mov` 指令helper函数定义的模板, `mov` 指令helper函数的函数体都在这个文件中定义. 模板的功能是通过宏来实现的: 对于一条指令, 不同操作数长度

的相近形式都有相似的行为, 可以将它们的公共行为用宏抽象出来. `mov-template.h` 的开头包含了头文件 `nemu/include/cpu/exec/template-start.h` , 结尾包含了头文件 `nemu/include/cpu/exec/template-end.h` , 它们包含了一些在模板头文件中使用的宏定义, 例如 `DATA_TYPE, REG()` 等, 使用它们可以编写出简洁的代码.
    - `mov.c` 中定义了 `mov` 指令的所有helper函数, 其中分三次对 `mov-template.h` 中定义的模板进行实例化, 进行宏展开之后就可以得到helper函数的完整定义了; 另外操作数后缀为 `v` 的helper函数也在 `mov.c` 中定义.

在PA2中, 你需要编写很多helper函数, 好的源文件组织方式可以帮助你方便地管理工程.

实现新指令

我们对实现一条新指令的流程进行整理. 如果指令的形式比较规整, 适合使用模板(适合大部分的指令), 可以按照以下流程来实现

1. 编写指令模板文件 `xxx-template.h`
    - 在文件头尾分别包含 `cpu/exec/template-start.h` 和 `cpu/exec/template-end.h`
    - 定义宏 `instr` 为指令名称
    - 定义函数 `static void do_execute()` , 实现该指令的通用执行过程
    - 定义helper函数
        - 若指令的译码方式在 `nemu/include/cpu/decode/decode.h` 中已经存在, 那么可以考虑使用宏 `make_instr_helper()` 来构造helper函数 (大部分helper函数都可以通过这种方式构造)
        - 否则
            - 可以考虑添加相应的译码函数
            - 或者不使用 `make_instr_helper()` , 而是直接使用 `make_helper()` 来定义helper函数, 在函数体中直接进行译码, 并调用 `do_execute()` (可以参考 `nemu/src/cpu/exec/data-mov/xchg-template.h` 中的 `xchg_a2r` 指令类型)
2. 编写指令实例化文件 `xxx.c`
    - 包含 `cpu/exec-helper.h`
    - 通过分别将宏 `DATA_BYTE` 定义成 `1` , `2` , `4` , 分别对指令模板文件 `xxx-template.h` 进行实例化
    - 若一个helper函数只会在某些操作数长度中用到, 可以在 `xxx-template.h` 中通过条件编译的功能来指定 (可以参考 `nemu/src/cpu/exec/data-mov/xchg-template.h` 中的 `xchg_a2r` 指令类型)
    - 必要时通过宏 `make_helper_v()` 定义相应的重载函数, 根据指令的操作数长度前缀确定调用哪一个helper函数
3. 编写指令头文件 `xxx.h`
    - 声明helper函数的原型
4. 在 `nemu/src/cpu/exec/all-instr.h` 中包含 `xxx.h`
5. 在 `nemu/src/cpu/exec/exec.c` 中的 `opcode_table` 中填写相应的helper函数

如果指令的形式不易抽象成模板(例如 `ret`), 那么可以不采取模板的方式实现, 直接在 `xxx.c` 中通过宏 `make_helper()` 定义函数体, 并编写译码和执行的过程.

初出茅庐: 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, 就是编写几条指令的helper函数, 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`testcase/src/mov-c.c`, 它做的事情十分简单, 对数组的某些元素进行赋值, 然后马上读出这些元素的值, 检查它们是否被正确赋值.

使用assertion进行验证

要怎么证明mov-c程序正确运行了呢? 你可能马上想到把元素的值输出到屏幕上看看. 但是, `输出一句话`是一件很复杂的事情(没错! 的确是一件很复杂的事情, 尽管你天天都在用), 由于现在NEMU的功能十分简陋, 不足以支持用户程序进行输出. 事实上, 做PA的最终目标之一, 就是让用户程序成功输出一句话, 回过头来你就能够理解, 程序要输出一句话其实也不容易.

既然用户程序不能输出数组元素, 那就用简易调试器中的扫描内存功能, 把数组元素所在的内存区域打印出来看看吧! 这是一个可行的方法, 但你很快就会因为把时间花费在人工检查当而感到厌倦了.

有没有一种方法能够让程序自动进行检查呢? 当然有! 那不就是帮你拦截了无数bug的assertion吗? assertion的功能就是当检查条件为假时, 马上终止程序的执行, 并汇报违反assertion的地方. 先别着急, 终止程序是需要操作系统的帮助的, 目前NEMU中并没有运行操作系统, 是不能直接使用标准库中的assertion功能的. 幸运的是, 框架代码早就已经考虑到这点了, 还记得在PA1中提到的`nemu_trap`这条特殊的指令吗? 我们只需要对这条特殊的指令稍作包装, 就可以把assertion的功能移植到用户程序中了!

移植后的assertion通过`nemu_assert()`来使用, 它是个宏, 在`lib-common/trap.h`中定义. `lib-common/trap.h`专门定义了一些用于测试的宏:

```
#define HIT_GOOD_TRAP \
    asm volatile(".byte 0xd6" : : "a" (0))

#define HIT_BAD_TRAP \
    asm volatile(".byte 0xd6" : : "a" (1))

#define nemu_assert(cond) \
    do { \
        if( !(cond) ) HIT_BAD_TRAP; \
    } while(0)
```

其中`HIT_GOOD_TRAP`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

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

的方式来给出指令, 汇编器将会报错, 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/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. `HIT_BAD_TRAP` 的功能是类似的, 这里就不再进行叙述了.

最后来看看 `nemu_assert()` , 它做的事情十分简单, 当条件为假时, 就执行 `HIT_BAD_TRAP` . 这样几行代码就实现了assertion的功能, 我们就可以在用户程序中使用assertion了.

上述三个宏都有相应的汇编版本, 在汇编代码中包含头文件trap.h, 你就可以使用它们了. 不过汇编版本的 `nemu_assert()` 功能比较简陋, 它只能判断某个通用寄存器是否与给定的一个立即数相等.

最后, 用户程序从 `main` 函数返回之后, 会通过 `HIT_GOOD_TRAP` 强行结束用户程序的运行, 同时也提示我们用户程序通过了所有的assertion.

main函数返回到哪里

查看testcase的相关代码, 你知道用户程序从 `main` 函数返回之后会跳转到哪里吗? 如果用户程序在GNU/Linux中运行, 问题的答案又是什么?

运行时环境与交叉编译

在让NEMU运行用户程序之前, 我们先来讨论NEMU需要为用户程序的运行提供什么. 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. 现在轮到你来为用户程序提供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. 目前, 我们约定NEMU提供的运行时环境有:

1. 物理内存有128MB(当然, 这是我们模拟出来的物理内存), 所有内存地址都是物理地址
2. 程序入口位于地址 `0x100000` , 程序总是从这里开始执行
3. `%ebp` 的初值为 `0` , `%esp` 的初值为 `0x8000000` 附近. 需要注意的是, 这个地址是物理内存的最大值, 是一个非法的物理地址, 不能直接访问

4. 程序通过 `nemu_trap` 结束运行
5. 不提供库函数的动态链接, 但提供静态链接, 故实际上对用户程序来说, 库函数的使用与运行时环境无关. 库函数的静态链接是通过框架代码中提供的函数库uclibc实现的, 相应的文件有 `lib-common/uclibc/lib/libc.a` 和 `lib-common/uclibc/include` 目录下的头文件, `Makefile` 中已经有相应的设置了. uclibc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, 其中一些函数甚至不需要任何运行时环境的支持(例如 `memcpy` 等), 这正好符合NEMU的情况. 这样, 你就可以在用户程序中使用一些不需要运行时环境支持的库函数了. 但类似于 `printf()` 这种需要运行时环境支持的库函数目前还是无法使用, 否则将会发生链接错误.

在让NEMU运行mov-c用户程序之前, 我们需要将用户程序的代码 `mov-c.c` 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 `mov-c.c` , 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并不能为用户程序提供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NEMU提供的运行时环境编译出能够在NEMU中运行的可执行文件. 框架代码已经把相应的配置准备好了: gcc先将源文件编译成目标文件, 然后让ld根据链接脚本 `testcase/user.ld` 将目标文件链接成可执行文件. 根据链接脚本 `testcase/user.ld` 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 `obj/testcase/start.o` 中的 `.text` 节开始, 然后接下来是其它目标文件的 `.text` 节, 未列出的节将以默认顺序往后依次放置. 这样, 可执行程序的 `0x100000` 处总是放置 `testcase/src/start.S` 的代码, 而不是其它代码, 保证用户程序总能从 `0x100000` 开始正确执行.

修改工程目录下的 `Makefile` 文件, 更换NEMU的用户程序:

```
--- Makefile
+++ Makefile
@@ -56,2 +56,2 @@
-USERPROG = obj/testcase/mov
+USERPROG = obj/testcase/mov-c
 ENTRY = $(USERPROG)
```

修改后, 键入

```
make run
```

使用新的用户程序运行NEMU, 你会发现NEMU输出以下信息:

```
invalid opcode(eip = 0x0010000a): e8 06 00 00 00 b8 00 00 ...

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!

nemu: nemu/src/cpu/exec/special/special.c:24: inv: Assertion `0' failed.
```

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.

实现最少的指令

要实现哪些指令才能让mov-c在NEMU中运行起来呢? 答案就在其反汇编结果(`obj/testcase/mov-c.txt`)中. 查看反汇编结果, 你发现只需要添加 `call` , `push` , `test` , `je` , `cmp` , `pop` , `ret` 七条指令就可以了. 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mov-c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
- `push` : 现在只需要实现 `PUSH r32` 的形式就可以了
- `test` : 在实现 `test` 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
+-------------------+---------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                   |V|R| |N|I O|O|D|I|T|S|Z| |A| |P| |C|
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | |0| |   | | | | | | |0| |0| |1| |
|                                   |M|F| |T|P L|F|F|F|F|F|F| |F| |F| |F|
+-------------------+---------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

关于EFLAGS中每一位的含义, 请查阅i386手册. 在NEMU中, 我们只会用到EFLAGS中以下的7个位: `CF` , `PF` , `ZF` , `SF` , `IF` , `DF` , `OF` . 其余位的功能可暂不实现. 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, 你就可以实现 `test` 指令了

- `je` : `je` 指令是 `jcc` 的一种形式
- `cmp` : 要注意被减数和减数的位置
- `pop` , `ret` : RTFM吧

运行用户程序mov-c

编写相应的helper函数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中运行用户程序mov-c, 你将会看到 `HIT GOOD TRAP` 的信息.

温馨提示

PA2阶段1到此结束.

融会贯通: 实现更多的指令

为了让NEMU支持大部分程序的运行, 目前你需要实现以下指令:

- Data Movement Instructions: `mov` `xchg` push pop leave movsx movzx cwtl/cltd(在i386手册中为 cwd/cdq)
- Binary Arithmetic Instructions: add adc sub sbb cmp `inc` `dec` `neg` `mul` `imul` `div` `idiv`
- Logical Instructions: `not` `and` `or` `xor` `sal(shl)` `shr` `sar` setcc test
- Control Transfer Instructions: jmp call ret jcc
- String and Character Translation Instructions: movs stos lods scas `rep`
- Miscellaneous Instructions: `lea` `nop`

你只需要实现上述带有下划线的指令, 它们不多不少都和以下的某些内容相关: EFLAGS, 堆栈, 整数扩展, 加减溢出判断. 我们需要针对 `movs`, `stos`, `lods` 和 `scas` 这四条字符串操作指令进行补充说明: 这四条指令的执行涉及到段寄存器, 但我们目前并没有在NEMU中实现段寄存器, 因此我们先忽略和段寄存器相关的描述. 例如i386手册中对 `movs` 指令有如下描述:

> Move DS:[(E)SI] to ES:[(E)DI]

目前我们只需要理解成

> Move [(E)SI] to [(E)DI]

即可.

框架代码已经把其它指令实现好了, 但并没有填写 `opcode_table` . 此外, 某些需要更新EFLAGS的指令, 以及有符号立即数的译码函数(在 `nemu/src/cpu/decode/decode-template.h` 中定义)并没有完全实现好(框架代码中已经插入了 `panic()` 作为提示), 你还需要编写相应的功能.

断点(2)

我们在PA1中介绍了如何通过监视点来模拟断点, 不过这种方法需要提前知道设置断点的地址, 下面来介绍一种不需要提前知道断点地址的设置方法.

在x86中有一条叫 `int3` 的特殊指令, 它是专门给调试器准备的, 一般的程序不应该使用这条指令. 当程序执行到int3指令的时候, CPU将会抛出一个含义为"程序触发了断点"的异常(exception), 操作系统会捕捉到这个异常, 然后操作系统会通过信号机制(signal), 向程序发送一个SIGTRAP信号, 这样程序就知道自己触发了一个断点. 如果你现在觉得上述过程很难理解, 不必担心, 你只需要知道

> 当程序执行到 `int3` 指令的时候, 调试器就能够知道程序触发了断点

设置断点, 其实就是在程序中插入 `int3` 指令. `int3` 指令的机器码为 `0xcc` , 长度为一个字节. 如果你看过PA1中关于断点的阅读材料, 你会发现断点真正的工作原理比较复杂, 根据KISS法则, 我们采用一种简单的方法来实现断点的功能. 在 `lib-common/trap.h` 中提供了一个函数 `set_bp()` , 它的功能就是马上执行 `int3` 指令. 当NEMU发现程序执行的是 `int3` 指令时, 输出一句话提示用户触发了断点, 最后返回到 `ui_mainloop()` 循环中等待用户的命令.

框架代码已经实现上述功能了. 要使用断点, 你只要在用户程序的源代码中调用 `set_bp()` 函数, 就可以达到设置断点的效果了. 你可以在 `testcase/src/mov-c.c` 中插入断点, 然后重新编译mov-c程序并运行NEMU来体会这种断点的设置方法. 需要注意的是, 这种简化的做法其实是对 `int3` 指令的滥用, 因为在真实的操作系统中, 一般的程序不应该使用 `int3` 指令, 否则它将会在运行时异常终止.

不过这种方法也有不足之处: 一行C代码可能会对应很多条机器指令, 因此上述方法并不能在任意位置设置断点. 例如我们熟知的函数调用语句, 其对应的机器指令分为压入实参和控制转移两部分, 我们很难使用 `set_bp()` 在程序压入实参后, 执行 `call` 指令前设置断点, 不过这对监视点来说就不在话下了.

测试用例

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 `testcase` 目录下准备了一些测试用例, 需要更换测试用例时, 修改工程目录下 `Makefile` 中的 `USERPROG` 变量, 改成测试用例的可执行文件(`obj/testcase/xxx` , 不是C源文件)即可.

实现更多的指令

你需要实现上文中提到的更多指令, 以支持 `testcase` 目录下更多程序的运行. 实现的时候尽可能使用框架代码中的宏(参考 `include/cpu/exec/helper.h` 和 `include/cpu/exec/template-start.h`), 它们可以帮助你编写出简洁的代码.

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 `testcase` 目录下的测试就可以了.

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. struct
3. string
4. hello-str

此外, matrix-mul需要运行较长时间, 运行过程中NEMU会输出大概400个 `.` , 请耐心等待.

`testcase` 目录下的大部分测试用例都可以直接在NEMU上运行, 除了以下几个测试用例:

- hello-inline-asm
- hello
- integral
- quadratic-eq
- print-FLOAT

其中运行hello-inline-asm和hello需要系统调用的支持, 在PA2中我们无法提供系统调用的功能, 这两个测试用例将会在PA4中用到, 目前你可以忽略它们; 要运行print-FLOAT需要使一些与运行时代码相关的小技巧, 我们在PA2的最后再来讨论如何运行它; 而integral和quadratic-eq涉及到浮点数的使用, 我们先来讨论如何处理浮点数.

实现binary scaling

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, 而且我们并不打算在用户程序中直接使用浮点指令. 为了在保持程序逻辑的同时不引入浮点指令, 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.

实现binary scaling

框架代码已经将测试用例中涉及浮点数的部分用 `FLOAT` 类型来模拟, 你需要实现一些和 `FLOAT` 类型相关的函数:

```
/* lib-common/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);
/* lib-common/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. `lib-common/FLOAT/FLOAT.c` 中的 `pow()` 函数目前不会用到, 我们会在PA4再提到它. 实现成功后, 你还需要在 `lib-common/FLOAT/Makefile.part` 中编写用于分别生成 `FLOAT.o` 和 `FLOAT_vfprintf.o` 的规则, 要求如下:

- 只编译不链接
- 使用 `-m32` , `-O2` 和 `-fno-builtin` 编译选项
- 添加 `lib-common` 目录作为头文件的搜索路径
- 把 `FLOAT.o` 和 `FLOAT_vfprintf.o` 生成到在 `obj/lib-common/FLOAT` 目录下, 若目录不存在, 可以先通过 `mkdir -p 目录路径名` 创建

`FLOAT_vfprintf.c` 中的代码会在运行print-FLOAT测试用例时被用到, 我们在PA2的最后再进一步说明, 目前你只需要通过相应的规则将其编译为目标文件, Makefile就会将其加入到 `FLOAT.a` 中, 将来链接的时候就能够找到相应的函数. 编写规则后, 修改工程目录下的 `Makefile` 文件:

```
--- Makefile
+++ Makefile
@@ -12,2 +12,2 @@
 LIBC = $(LIBC_LIB_DIR)/libc.a
-#FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT/FLOAT.a
+FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT/FLOAT.a
```

让 `FLOAT.a` 参与链接, 这样你就可以在NEMU中运行integral和quadratic-eq这两个测试用例了.

事实上, 我们并没有考虑计算结果溢出的情况, 不过我们的测试用例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.

编写自己的测试用例

从测试的角度来说, `testcase` 目录下的测试用例还不够完备, 很多指令可能都没有被覆盖到. 想象一下你编写了一个 `if` 语句, 但程序运行的时候根本就没有进入过这个 `if` 块中, 你怎么好意思说你写的这个 `if` 语句是对的呢? 因此我们鼓励你编写自己的测试用例, 尽可能地覆盖到你写的所有代码. 用户程序的来源有很多, 例如程序设计作业中的小程序, 或者是已经完成的数据结构作业等等. 但你需要注意NEMU提供的运行时环境, 用户程序不能输出, 只能通过 `nemu_assert()` 来进行验证. 你可以按照以下步骤编写一个测试用例:

- 先使用 `printf()` 根据数组的格式输出测试结果, 此时你编写的是一个运行在GNU/Linux下的程序, 直接用gcc编译即可.
- 运行程序, 得到了数组格式的输出, 然后把这些输出结果作为全局数组添加到源代码中, 你可以通过 `>>` 将输出重定向追加到源代码中.
- 去掉代码中的 `printf()` 和头文件, 包含 `trap.h` , 使用 `nemu_assert()` 进行验证.
- 把修改后的.c文件放到 `testcase/src` 目录下即可.

这样你就成功添加了一个测试用例了, 按照上文提到的步骤更换测试用例, 就可以使用你的测试用例进行测试了.

我们还鼓励你把测试用例分享给大家, 我们在提交网站中创建了一个"测试用例分享"的讨论版, 你可以在讨论版中分享你的测试用例, 同时也可以使用其它同学提供的测试用例进行测试. 你的程序通过越多的测试, 程序的健壮性就越好, 越有希望通过"在NEMU上运行仙剑奇侠传"的终极考验.

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 `jnz` 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 "jnz" carries a variable: jnz 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. 但NEMU现在连输出一句话的功能都无法向用户程序提供, 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.

第三视点: 简易调试器(2)

接下来, 我们将会对简易调试器的功能进行扩展, 为调试提供更多的手段.

运行用户程序add

在继续之前, 请保证用户程序add可以在NEMU中正确运行. 你将使用add程序来测试简易调试器的新功能.

添加变量支持

你已经在PA1中实现了简易调试器, 现在你已经将用户程序换成了C程序. 和之前的 `mov.S` 相比, C程序多了变量和函数的要素, 那么在表达式求值中如何支持变量的输出呢?

```
(nemu) p test_data
```

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

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

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

```
readelf -a add
```

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

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

其中每一行代表一个表项, 每一列列出了表项的一些属性, 现在我们只需要关心 `Type` 属性为 `OBJECT` 的表项就可以了. 仔细观察 `Name` 属性之后, 你会发现这些表项正好对应了 `add.c` 中定义的全局变量, 而相应的 `Value` 属性正好是它们的地址(你可以与 `add.txt` 中的反汇编结果进行对比), 而找到地址之后就可以找到这个变量了.

消失的符号

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

太好了, 我们可以通过符号表建立变量名和其地址之间的映射关系! 别着急, `readelf` 输出的信息是已经经过解析的, 实际上符号表中 `Name` 属性存放的是字符串在字符串表(string table)中的偏移量. 为了查看字符串表, 我们先查看 `readelf` 输出中Section Headers的信息:

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00100000 001000 00009a 00  AX  0   0  4
  [ 2] .eh_frame         PROGBITS        0010009c 00109c 000058 00   A  0   0  4
  [ 3] .data             PROGBITS        00100100 001100 000120 00  WA  0   0 32
  [ 4] .comment          PROGBITS        00000000 001220 00001c 01  MS  0   0  1
  [ 5] .shstrtab         STRTAB          00000000 00123c 00003a 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0013b8 0000a0 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 001458 00001e 00      0   0  1
```

从Section Headers的信息可以看到, 字符串表在ELF文件偏移为 `0x1458` 的位置开始存放. 在shell中可以通过以下命令直接输出ELF文件的十六进制形式:

```
hd add
```

查看输出结果的最后几行, 我们可以看到, 字符串表只不过是把标识符的字符串拼接起来而已. 现在我们就可以厘清符号表和字符串表之间的关系了:

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 7] .strtab           STRTAB          00000000 001458 00001e 00      0   0  1
                                                 |
                                  +--------------+     +----------------------+
                                  V                    V                      |
00001450  20 00 00 00 11 00 03 00  00 61 64 64 2e 63 00 61  | ........add.c.a|  |
00001460  64 64 00 6d 61 69 6e 00  61 6e 73 00 74 65 73 74  |dd.main.ans.test|  |
00001470  5f 64 61 74 61 00        ^           ^            |_data.|            |
                                  |           |                               |
                                  |           +-------------+                 |
                                  |                         |                 |
                                  +---------------------+   |                 |
Symbol table '.symtab' contains 10 entries:               |   |                 |
   Num:    Value  Size Type    Bind   Vis      Ndx Name   |   |                 |
     5: 00000000     0 FILE    LOCAL  DEFAULT  ABS 1      |   |                 |
     6: 00100084    22 FUNC    GLOBAL DEFAULT    1 7   ---+---+-----------------+
     7: 00100000   129 FUNC    GLOBAL DEFAULT    1 11     |   |
     8: 00100120   256 OBJECT  GLOBAL DEFAULT    3 16 ---+   |
     9: 00100100    32 OBJECT  GLOBAL DEFAULT    3 20 -------+
```

寻找"Hello World!"

在GNU/Linux下编写一个Hello World程序, 编译后通过上述方法找到ELF文件的字符串表, 你发现"Hello World!"字符串在字符串表中的什么位置? 为什么会这样?

一种解决方法已经呼之欲出了: 在表达式递归求值的过程中, 如果发现token的类型是一个标识符, 就通过这个标识符在符号表中找到一项符合要求的表项(表项的 `Type` 属性是 `OBJECT` , 并且将 `Name` 属性的值作为字符串表中的偏移所找到的字符串和标识符的命名一致), 找到标识符的地址, 并将这个地址作为结果返回. 在上述add程序的例子中:

```
(nemu) p test_data
0x100100
```

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

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

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

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

为表达式求值添加变量的支持

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

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

```
man 5 elf
```

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

丢失的信息

在用户程序中定义以下字符数组:

```
char str[] = "abcdefg";
```

尝试通过上述方式输出 `str[1]` 的值, 你发现有什么问题? 运用现有的信息, 你能够解决这个问题吗? 如果能, 请描述解决方法, 并尝试实现; 如果不能, 请解释为什么, 并尝试总结这背后反映的事实.

冗余的符号表

在GNU/Linux下编写一个Hello World程序, 然后使用 `strip` 命令丢弃可执行文件中的符号表:

```
gcc -o hello hello.c
strip -s hello
```

用 `readelf` 查看hello的信息, 你会发现符号表被丢弃了, 此时的hello程序能成功运行吗?

目标文件中也有符号表, 我们同样可以丢弃它:

```
gcc -c hello.c
strip -s hello.o
```

用 `readelf` 查看hello.o的信息, 你会发现符号表被丢弃了. 尝试对hello.o进行链接:

```
gcc -o hello hello.o
```

你发现了什么问题? 尝试对比上述两种情况, 并分析其中的原因.

打印栈帧链

我们知道函数调用会在堆栈上形成栈帧, 记录和这次函数调用有关的信息. 若干次连续的函数调用将会在堆栈上形成一条栈帧链, 为调试提供了很多有用的信息: `%eip` 可以让你知道程序现在的位置, 栈帧链则可以告诉你, 程序是怎么运行到现在的位置的. 我们需要在简易调试器中添加 `bt` 命令, 打印出栈帧链的信息, 如果你从来没有使用过 `bt` 命令, 请先在GDB中尝试.

在堆栈中形成的栈帧链结构如下:

```
      |   |   ......   | 4G
    stack +------------+
    frame |  prev_ebp  |
      |   +------------+ <--+
      |   | local_var  |    |
      v   | &temp_var  |    |
    ----- +------------+    |
      ^   | arguments  |    |
      |   +------------+    |
      |   |  ret_addr  |    |
    stack +------------+    |
    frame |  prev_ebp  | ---+
      |   +------------+ <--+
      |   | local_var  |    |
      v   | &temp_var  |    |
    ----- +------------+    |
      ^   | arguments  |    |
      |   +------------+    |
      |   |  ret_addr  |    |
    stack +------------+    |
    frame |  prev_ebp  | ---+
      |   +------------+ <--+
      |   | local_var  |    |
      v   | &temp_var  |    |
    ----- +------------+    |
      ^   | arguments  |    |
      |   +------------+    |
      |   |  ret_addr  |    |
    stack +------------+    |
    frame |  prev_ebp  | ---+
      |   +------------+ <-- %ebp
      |   |   ......   | 0
```

可以看到, `%ebp` 在栈帧链的组织中起到了非常重要的作用, 通过 `%ebp` , 我们就可以找到每一个栈帧的信息了. 聪明的你也许一眼就看出来, 这不就是程序设计课中学过的链表吗? 我们可以定义一个结构体来进一步厘清其中的奥妙:

```
typedef struct {
    swaddr_t prev_ebp;
    swaddr_t ret_addr;
    uint32_t args[4];
} PartOfStackFrame;
```

其中 `prev_ebp` 就类似于 `next` 指针, 不过我们没有将它定义成指针类型, 这是因为它表示的地址是用户程序的地址, 直接把它作为NEMU的地址来进行解引用就会发生错误, 所以这个结构体中的每一个成员都需要通过 `swaddr_read()` 来读取. `args` 成员数组表示函数的实参, 实际上实参的个数不一定是4个, 但我们仍然可以将它们强制打印出来, 说不定可以从中发现一些有用的调试信息. 链表的表头存储在 `%ebp` 寄存器中, 所以我们可以从 `%ebp` 寄存器开始, 像遍历链表那样逐一扫描并打印栈帧链中的信息. 链表通过 `NULL` 指示链表的结束, 在栈帧链中也是类似的. 还记得NEMU提供的运行时环境吗? `%ebp` 寄存器的初值为 `0` , 当我们发现栈帧中 `%ebp` 的信息为 `0` 时, 就表示已经达到最开始运行的函数了.

由于缺乏形参和局部变量的具体信息, 我们只需要打印地址, 函数名, 以及前4个参数就可以了, 打印格式可以参考GDB中 `bt` 命令的输出. 如何确定某个地址落在哪一个函数中呢? 这就需要符号表的帮助了:

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

对于 `Type` 属性为 `FUNC` 的表项, `Value` 属性指示了函数的起始地址, `Size` 属性指示了函数的大小, 通过这两个属性就可以确定函数的范围了. 由于函数的范围是互不相交的, 因此我们可以通过扫描符号表中 `Type` 属性为 `FUNC` 的每一个表项, 唯一确定一个地址在所的函数. 为了得到函数名, 你只需要根据表项中的 `Name` 属性在字符串表中找到相应的字符串就可以了.

打印栈帧链

为简易调试器添加 `bt` 命令, 实现打印栈帧链的功能. 实现之后, 在add的 `add()` 函数中设置断点, 触发断点之后, 在monitor中测试 `bt` 命令的实现是否正确.

%ebp是必须的吗?

使用优化选项编译代码的时候, gcc会对代码进行优化, 会将 `%ebp` 当作普通的寄存器来使用, 不再让其作为指示当前的栈帧, 更多的信息可以查阅 `man gcc` 中的 `-fomit-frame-pointer` 选项. 我们使用 `-O2` 来编译NEMU, 你可以对NEMU进行反汇编, 查看一些函数的代码. 在这种情况下, 代码要怎么找到函数调用的参数和局部变量?

另外优化 `%ebp` 寄存器之后, 就不能使用上述方法来打印栈帧链了. 如果你使用GDB对NEMU进行调试, 你会发现仍然可以使用bt命令来打印栈帧链. 你知道这是怎么做到的吗? 在优化 `%ebp` 寄存器之后, 为了打印栈帧链, 还需要哪些信息?

温馨提示

PA2阶段3到此结束.

三生万物: 实现加载程序的loader

loader是一个用于加载程序的模块, 实现了足够多的指令之后, 你也可以实现一个很简单的loader, 来加载其它用户程序.

可执行文件的组织

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, 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?

我们在PA1中已经提到了以下内容:

> 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NEMU先把内存开始的位置作为ramdisk来使用.

现在的ramdisk十分简单, 它只有一个文件, 也就是我们将要加载的用户程序, 访问内存位置0就可以得到用户程序的第一个字节. 这其实已经回答了上述第一个问题: 可执行文件位于内存位置0. 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.

代码和(静态)数据是程序的必备要素, 可执行文件中自然需要包含这两者. 但仅仅包含它们还是不够的, 我们还需要一些额外的信息来告诉我们"代码和数据分别有多少", 否则我们连它们两者的分界线在哪里都不知道. 这些额外的信息描述了可执行文件的组织形式, 不同组织形式形成了不同格式的可执行文件, 例如Windows主流的可执行文件是PE(Protable Executable)格式, 而GNU/Linux主要使用ELF(Executable and Linkable Format)格式, 因此一般情况下, 你不能在Windows下把一个可执行文件拷贝到GNU/Linux下执行, 反之亦然. ELF是GNU/Linux可执行文件的标准格式, 这是因为GNU/Linux遵循System V ABI(Application Binary Interface).

堆和栈在哪里?

我们提到了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?

如何识别不同格式的可执行文件?

如果你在GNU/Linux下执行一个从Windows拷过来的可执行文件, 将会报告"格式错误". 思考一下, GNU/Linux是如何知道"格式错误"的?

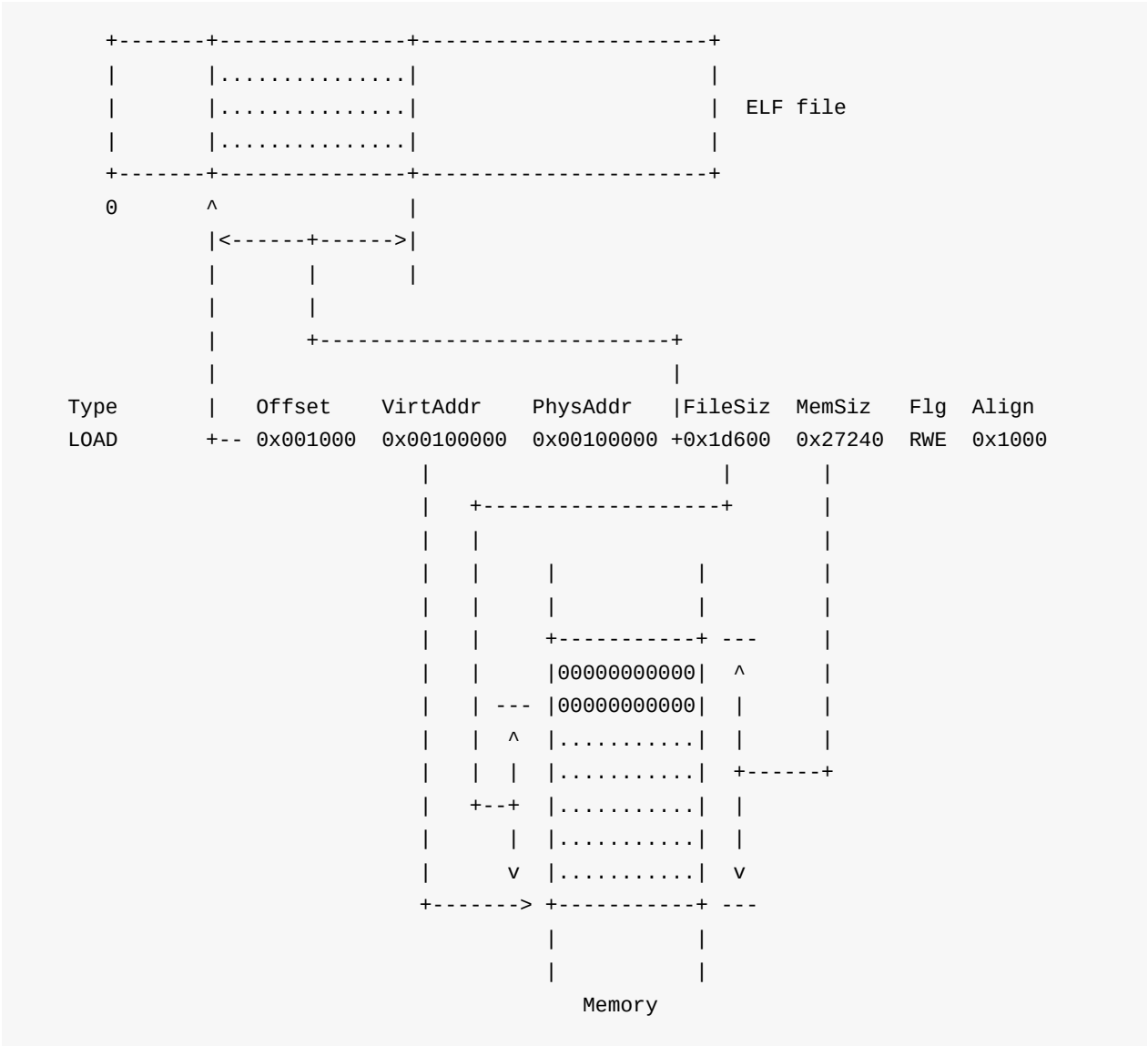
你应该已经学会使用 `readelf` 命令来查看ELF文件的信息了. ELF文件提供了两个视角来组织一个可执行文件, 一个是面向链接过程的section视角, 这个视角提供了用于链接与重定位的信息(例如符号表); 另一个是面向执行的segment视角, 这个视角提供了用于加载可执行文件的信息. 通过readelf命令, 我们还可以看到section和segment之间的映射关系: 一个segment可能由0个或多个section组成, 但一个section可能不被包含于任何segment中.

我们现在关心的是如何加载程序, 因此我们重点关注segment的视角. ELF中采用program header table来管理segment, program header table的一个表项描述了一个segment的所有属性, 包括类型, 虚拟地址, 标志, 对齐方式, 以及文件内偏移量和segment大小. 根据这些信息, 我们就可以知道需要加载可执行文件的哪些字节了, 同时我们也可以看到, 加载一个可执行文件并不是加载它所包含的所有内容, 只要加载那些与运行时刻相关的内容就可以了, 例如调试信息和符号表就不必加载. 由于运行时环境的约束, 在PA中我们只需要加载代码段和数据段, 如果你在GNU/Linux下编译一个Hello world程序并使用 `readelf` 查看, 你会发现它需要加载更多的内容.

冗余的属性?

使用 `readelf` 查看一个ELF文件的信息, 你会看到一个segment包含两个大小的属性, 分别是 `FileSiz` 和 `MemSiz` , 这是为什么? 再仔细观察一下, 你会发现 `FileSiz` 通常不会大于相应的 `MemSiz` , 这又是为什么?

我们通过下面的图来说明如何根据segment的属性来加载它:



你需要找出每一个program header的 `Offset` , `VirtAddr` , `FileSiz` 和 `MemSiz` 这些参数. 其中相对文件偏移 `Offset` 指出相应segment的内容从ELF文件的第 `Offset` 字节开始, 在文件中的大小为 `FileSiz` , 它需要被分配到以 `VirtAddr` 为首地址的虚拟内存位置, 在内存中它占用大小为 `MemSiz` . 但现在NEMU还没有虚拟地址的概念, 因此你只需要把 `VirtAddr` 当做物理地址来使用就可以了, 也就是说, 这个segment使用的内存就是 `[VirtAddr, VirtAddr + MemSiz)` 这一连续区间, 然后将segment的内容从ELF文件中读入到这一内存区间, 并将 `[VirtAddr + FileSiz, VirtAddr + MemSiz)` 对应的物理区间清零.

为什么要清零?

为什么需要将 `[VirtAddr + FileSiz, VirtAddr + MemSiz)` 对应的物理区间清零?

关于程序从何而来, 可以参考一篇文章: COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY. 如果你希望查阅更多与ELF文件相关的信息, 请参考

```
man 5 elf
```

在kernel中实现loader

理解了上述内容之后, 你就可以在kernel中实现loader了. 在这之前, 我们先对kernel作一些简单的介绍.

在PA中, kernel是一个单任务微型操作系统的内核, 其代码在工程目录的 `kernel` 目录下. kernel已经包含了后续PA用到的所有模块, 换句话说, 我们现在就要在NEMU上运行一个操作系统了(尽管这是一个十分简陋的操作系统), 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内核的组成. 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. 由于NEMU的功能是逐渐添加的, kernel也要配合这个过程, 你会通过 `kernel/include/common.h` 中的一些与实验进度相关的宏来控制kernel的功能. 随着实验进度的推进, 我们会逐渐讲解所有的模块, kernel做的工作也会越来越多. 因此在阅读kernel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.

另外需要说明的是, 虽然不会引起明显的误解, 但在引入kernel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.

在工程目录下执行 `make kernel` 来编译kernel. kernel的源文件组织如下:

```
kernel
├── include
│   ├── common.h
│   ├── debug.h
│   ├── memory.h
│   ├── x86
│   │   ├── cpu.h
│   │   ├── io.h
│   │   └── memory.h
│   └── x86.h
├── Makefile.part
└── src
    ├── driver
    │   ├── ide            # IDE驱动程序
    │   │   └── ...
    │   └── ramdisk.c    # ramdisk驱动程序
    ├── elf            # loader相关
    │   └── elf.c
    ├── fs            # 文件系统
    │   └── fs.c
    ├── irq            # 中断处理相关
    │   ├── do_irq.S        # 中断处理入口代码
    │   ├── i8259.c        # intel 8259中断控制器
    │   ├── idt.c        # IDT相关
    │   └── irq_handle.c    # 中断分发和处理
    ├── lib
    │   ├── misc.c        # 杂项
    │   ├── printk.c
    │   └── serial.c        # 串口
    ├── main.c
    ├── memory            # 存储管理相关
    │   ├── kvm.c        # kernel虚拟内存
    │   ├── mm.c        # 存储管理器MM
    │   ├── mm_malloc.o        # 为用户程序分配内存的接口函数, 不要删除它!
    │   └── vmem.c        # video memory
    ├── start.S            # kernel入口代码
    └── syscall            # 系统调用处理相关
        └── do_syscall.c
```

一开始 `kernel/include/common.h` 中所有与实验进度相关的宏都没有定义, 此时kernel的功能十分简单. 我们先简单梳理一下此时kernel的行为:

1. 第一条指令从 `kernel/src/start.S` 开始, 设置好堆栈之后就跳转到 `kernel/src/main.c` 的 `init()` 函数处执行.
2. 由于NEMU还没有实现分段分页的功能, 此时kernel会跳过很多初始化工作, 直接跳转到 `init_cond()` 函数处继续进行初始化.
3. 继续跳过一些初始化工作之后, 会通过 `Log()` 宏输出一句话. 需要说明的是, kernel中定义的 `Log()` 宏并不是NEMU中定义的 `Log()` 宏, kernel和NEMU的代码是相互独立的, 因此编译某一方时都不会受到对方代码的影响, 你在阅读代码的时候需要注意这一点. 在

kernel中, `Log()` 宏通过 `printk()` 输出. 阅读 `printk()` 的代码, 发现此时 `printk()` 什么都不做就直接返回了, 这是由于现在NEMU还不能提供输出的功能, 因此现在kernel中的 `Log()` 宏并不能成功输出.

4. 调用 `loader()` 函数加载用户程序, `loader()` 函数会返回用户程序的入口地址.
5. 跳转到用户程序的入口执行.

理解上述过程后, 你需要在 `kernel/src/elf/elf.c` 的 `loader()` 函数中定义正确ELF文件魔数, 然后编写加载segment的代码, 完成加载用户程序的功能. 你需要使用 `ramdisk_read()` 函数来读出ramdisk中的内容, `ramdisk_read()` 函数的原型如下:

```
void ramdisk_read(uint8_t *buf, uint32_t offset, uint32_t len);
```

它负责把从ramdisk中 `offset` 偏移处的 `len` 字节读入到 `buf` 中.

我们之前让用户程序直接在 `0x100000` 处运行, 而现在我们希望先从 `0x100000` 处运行kernel, 让kernel中的loader模块加载用户程序, 然后跳转到用户程序处运行. 为此, 我们需要修改用户程序的链接选项:

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

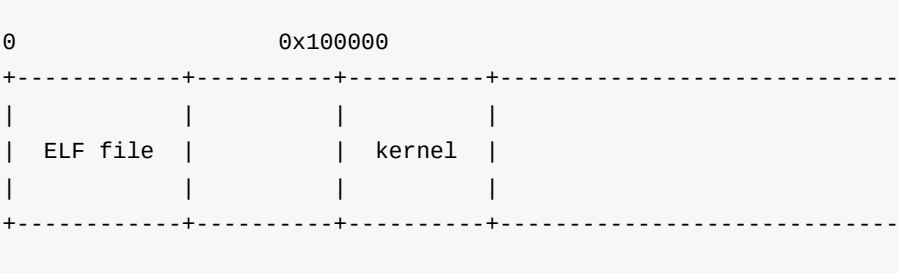
我们让用户程序从 `0x800000` 附近开始运行, 避免和kernel的内容产生冲突. 最后我们还需要修改工程目录下的 `Makefile` , 把kernel作为entry:

```
--- Makefile
+++ Makefile
@@ -56,2 +56,2 @@
 USERPROG = obj/testcase/mov-c
-ENTRY = $(USERPROG)
+ENTRY = $(kernel_BIN)
```

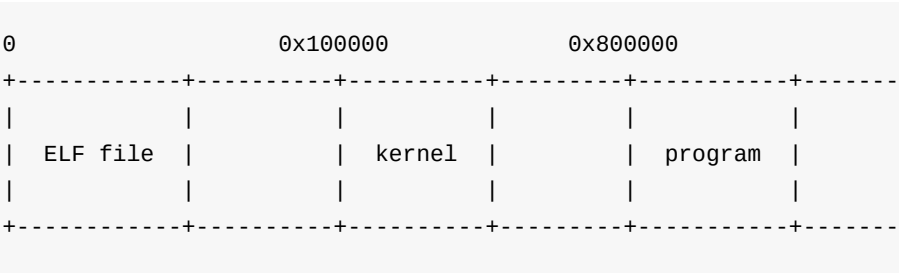
我们从运行时刻的角度来描述NEMU中物理内存的变化过程:

- 刚开始运行NEMU时, NEMU会进行一些全局的初始化操作, 其中有一项内容是初始化ramdisk(由 `nemu/src/monitor/monitor.c` 中的 `init_ramdisk()` 函数完成): 将用户程序的ELF文件从真实磁盘拷贝到模拟内存中地址为0的位置. 目前ramdisk中只有一个文件, 就是用户程序的ELF文件:

- 第二项初始化操作是将entry加载到内存位置 `0x100000` (由 `nemu/src/monitor/monitor.c` 中的 `load_entry()` 函数完成). 之前entry就是用户程序本身, 引入kernel之后, entry就变成kernel了. 换句话说, 在我们的PA中, kernel是由NEMU的monitor模块直接加载到内存中的. 这一点与真实的计算机有所不同, 在真实的计算机中, 计算机启动之后会首先运行BIOS程序, BIOS程序会将MBR加载到内存, MBR再加载别的程序... 最后加载kernel. 要模拟这个过程需要一个完善的模拟磁盘, 而且需要涉及较多的细节, 根据KISS法则, 我们不模拟这个过程, 而是让NEMU直接将kernel加载到内存. 这样, 在NEMU开始执行第一条指令之前, kernel就已经在内存中了:



- NEMU执行的第一条指令就是kernel的第一条指令. 如上文所述, kernel会完成一些自身的初始化工作, 然后从ramdisk中的EFL文件把用户程序加载到内存位置 `0x800000` 附近, 然后跳转到用户程序中执行:



实现loader

你需要在kernel中实现loader的功能, 让NEMU从kernel开始执行. kernel的loader模块负责把用户程序加载到正确的内存位置, 然后执行用户程序. 如果你发现你编写的loader没有正常工作, 你可以使用 `nemu_assert()` 和简易调试器进行调试.

实现loader之后, 你就可以使用 `test.sh` 脚本进行测试了, 在工程目录下运行

```
make test
```

脚本会将 `testcase` 中的测试用例逐个进行测试, 并报告每个测试用例的运行结果, 这样你就不需要手动切换测试用例了. 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.

温馨提示

PA2阶段4到此结束.

扭转乾坤: 改变程序的行为

print-FLOAT程序的功能是通过 `sprintf()` 函数的 `%f` 功能来格式化一个 `FLOAT` 类型的变量, 然后使用 `strcmp()` 函数来检查字符串是否正确. 等等, `%f` 不是用来格式化一个 `float` 类型的变量的吗? 那该如何实现让 `%f` 来格式化一个 `FLOAT` 类型的变量?

你可能会想修改uclibc的源代码, 然后重新编译. 这是一个能行的方法, 但需要进行库函数的交叉编译, 代价比较大, 有兴趣的同学可以到uclibc的官网上下载源代码并尝试修改.

既然不修改uclibc的源代码, 那么就从运行时的代码下手吧! 这正是这次任务最有魅力的地方: 你将要通过对 `libc.a` 进行攻击, 劫持相应的执行流, 来改变 `%f` 的行为!

为了方便调试, 我们先在GNU/Linux中实施攻击, 成功后再将程序移植到NEMU中运行. 框架代码已经为我们准备好生成相应文件的配置了(见 `testcase/Makefile.part`). 在工程目录下执行

```
make pa2-7
```

就会生成 `obj/testcase/print-FLOAT-linux` 这一可以在GNU/Linux中直接运行的可执行文件. 生成它时, `lib-common/uclibc/lib` 目录下的 `crt?.o` 会参与链接, 它们用于提供与GNU/Linux相兼容的运行时环境. 编译过程会通过gcc定义宏 `LINUX_RT` , 从而控制 `testcase/src/print-FLOAT.c` 采用 `printf()` 来输出结果. 代码期望首先通过 `init_FLOAT_vfprintf()` (在 `FLOAT.a` 中定义) 对 `libc.a` 进行劫持攻击, 之后通过 `%f` 进行格式化时就会执行我们编写的劫持目标代码, 该代码会根据 `FLOAT` 的编码方式进行格式化, 从而实现通过 `%f` 对 `FLOAT` 变量进行格式化的功能.

运行 `obj/testcase/print-FLOAT-linux` , 你会发现输出的结果为 `0.000000` , 这是因为我们还没有对 `libc.a` 进行劫持, 无法正确输出 `FLAOT` 类型的数据. 你的第一个任务就是要对 `libc.a` 中负责进行字符串格式化的 `_vfprintf_internal()` 函数进行劫持. 在劫持前, 格式化 `%f` 时 `vfprintf_internal()` 会运行如下代码:

```
else if (ppfs->conv_num <= CONV_A) {  /* floating point */
    ssize_t nf;
    nf = _fpmaxtostr(stream,
            (__fpmax_t)
            (PRINT_INFO_FLAG_VAL(&(ppfs->info),is_long_double)
             ? *(long double *) *argptr
             : (long double) (* (double *) *argptr)),
            &ppfs->info, FP_OUT );
    if (nf < 0) {
        return -1;
    }
    *count += nf;

    return 0;
} else if (ppfs->conv_num <= CONV_S) {  /* wide char or string */
```

其中会调用uclibc中的 `_fpmaxtostr()` 函数来实现 `float` 类型数据的格式化, 其函数原型说明如下:

```
extern ssize_t _fpmaxtostr(FILE * fp, __fpmax_t x, struct printf_info *info,
        __fp_outfunc_t fp_outfunc) attribute_hidden;
```

我们的目标是实现 `lib-common/FLOAT/FLOAT_vfprintf.c` 中的 `modify_vfprintf()` 函数, 它负责在运行时刻修改与上述c代码对应的二进制代码, 使其调用 `lib-common/FLOAT/FLOAT_vfprintf.c` 中的 `format_FLOAT()` 函数, 而不是 `_fpmaxtostr()` 函数. 要如何下手呢? 首先当然需要找到相应的二进制代码的位置.

知己知彼

对 `obj/testcase/print-FLOAT-linux` 进行反汇编, 找到与上述c代码对应的二进制代码的地址范围.

我们期望劫持后的代码如下:

```
else if (ppfs->conv_num <= CONV_A) {  /* floating point */
    ssize_t nf;
    nf = format_FLOAT(stream, *(FLOAT *) *argptr);
    if (nf < 0) {
        return -1;
    }
    *count += nf;

    return 0;
} else if (ppfs->conv_num <= CONV_S) {  /* wide char or string */
```

分析我们期望的代码, 我们只要做到以下几点即可:

- 将函数调用的目标改为 `format_FLOAT()`
- 设置好正确的函数调用参数
- 清理因为调用 `_fpmaxtostr()` 而留下的浮点指令

首先我们来修改函数调用的目标. RTFM后得知, `e8` 开头的是 `call REL32` 形式的指令, 因此后面跟的是一个相对于当前eip的偏移量. 因此, 我们只需要修改这一偏移量, 就可以达到修改函数调用目标的目的了. 为此, 我们需要得知这条 `call` 指令的地址. 假设从反汇编结果中得知该指令的地址为 `p`, 那么 `p+1` 就指向了我们需要修改的偏移量. 该如何修改这一偏移量呢? 其实我们只需要让它加上 `_fpmaxtostr()` 的首地址和 `format_FLOAT()` 的首地址之间的差即可. 由于 `format_FLOAT()` 就在 `lib-common/FLOAT/FLOAT_vfprintf.c` 中定义, 所以可以直接引用它来得到其首地址. 但 `_fpmaxtostr()` 却在 `libc.a` 中定义, 需要先通过外部引用来对其进行声明. 事实上, 框架代码中已经对其进行声明了.

狸猫换太子

我们知道 `_fpmaxtostr()` 是一个函数, 但框架代码的外部引用却把它声明成一个 `char` 类型的变量, 这样做会有问题吗? 思考一下, 在这种情况下, 编译器和链接器是如何处理这一声明的?

理解了上述内容之后, 你就可以在 `modify_vfprintf()` 中编写代码, 来修改函数调用的目标了. 不过可别高兴太早了, 我们编写代码重新编译之后, 由于代码的大小发生了变化, 可能会导致我们修改的那条 `call` 指令在链接重定位阶段后的地址不再是之前从反汇编结果中看到的 `p`! 这下可麻烦了, 难道每次编写完代码之后, 都要重新回过头来调整 `p` 的值? 更麻烦的是, 如果链接的用户程序改变了, `p` 也同样会改变! 这意味着, 这种把 `p` 写死的方法来修改指令是不能拓展到其它用户程序中的. 但天无绝人之路, 虽然这条 `call` 指令的位置会变化, 但它相对于 `_vfprintf_internal()` 函数首地址的偏移量是不变的! 这样, 我们只需要先计算出这一偏移量 `o`, 我们就可以通过 `_vfprintf_internal + o` 的方式找到这条 `call` 指令了. 由于 `_vfprintf_internal()` 函数的首地址是在链接重定位时确定的, 在运行时刻通过这种方式必定能找到 `call` 指令正确的位置.

偷龙转凤

在 `modify_vfprintf()` 中编写代码, 修改函数调用的目标, 使得 `_vfprintf_internal()` 在对 `%f` 进行格式化的时候调用 `format_FLOAT()`, 而不是 `_fpmaxtostr()`.

编写代码后, 编译并运行 `obj/testcase/print-FLOAT-linux`, 你会发现程序触发了段错误! 这是因为在GNU/Linux下, 程序的代码是只读的, 而我们的劫持需要在运行时刻修改程序的代码, 进行了违规的写操作, 从而触发了段错误. 为了使得我们的劫持可以顺利进行, 我们需要借助系统调用 `mprotect()` 的帮助. `mprotect()` 可以改变一段内存区间的访问权限, 详细信息请参考 `man 2 mprotect`. 我们需要在 `modify_vfprintf()` 修改代码之前为相应的内存区间设置可写权限:

```
#include <sys/mman.h>

mprotect((void *)((???) & 0xfffff000), 4096*2, PROT_READ | PROT_WRITE | PROT_EXEC);
```

其中, 你需要填写 `???` 中的内容, 内容为 `call` 指令所在地址的往前(backward) `100` 字节所在的地址. 往前 `100` 字节是因为我们接下来还需要修改这一区间的代码, 而 `100` 字节已经足够覆盖这一区间. 重新编译后运行, 如果你的代码实现正确, 你应该能看到目前通过 `%f` 输出了一个十六进制数.

但输出的这个十六进制数并不是我们在调用 `printf()` 时传入的 `FLOAT` 数据, 这是因为我们仅仅改变了函数调用的目标, `_vfprintf_internal()` 仍然按照调用 `_fpmaxtostr()` 那样压入参数, 但 `format_FLOAT()` 却以为 `vfprintf_internal()` 已经压入了正确的参数.

锱铢必较

你知道现在通过 `%f` 输出的十六进制数具体是怎么得到的吗?

接下来我们需要修改传入参数的代码了.

知己知彼(2)

阅读相应的汇编代码, 并回答以下问题:

- 调用 `_fpmaxtostr()` 的参数是如何压栈的?
- 变量 `argptr` 存放在哪一个寄存器中?
- 变量 `argptr` 指向的内容是什么?
- 变量 `stream` 的地址在哪里?
- 调用 `_fpmaxtostr()` 返回后会清理栈上若干字节的内容, 这若干字节都有些什么内容?

对比劫持前后的代码, 我们发现第一个参数 `stream` 都是一样的, 因此传入 `stream` 的代码不需要进行改动. 为了让 `format_FLOAT()` 拿到正确的实参 `f`, 我们需要在 `_vfprintf_internal()` 中将它的值放置到栈中正确的位置上. 原来的代码中是通过一条占用三个字节浮点指令来放置第二个参数的, 我们需要修改它, 将它换成一条 `push` 指令, 用于将 `argptr` 指向的内容压栈即可. 正所谓细节决定成败, 我们还需要处理以下的细节问题:

- 使用的 `push` 指令不能超过三个字节
- 如果使用的 `push` 指令不足三个字节, 剩下的字节要通过 `nop` 来覆盖, 保证原来那条三个字节的浮点指令"不留任何痕迹"
- 由于使用的 `push` 指令改变了栈的深度, 为了正确维护栈的深度, 我们还需要修改前一条用于分配栈空间大小的指令
- 由于 `format_FLOAT()` 只会使用两个参数, 传入的其余参数可以暂时忽略

偷龙转凤(2)

在 `modify_vfprintf()` 中编写代码, 修改 `_vfprintf_internal()` 的代码, 使其压入正确的参数, 让 `format_FLOAT()` 正确输出 `FLOAT` 数据的十六进制表示. 如果你仍然觉得毫无头绪, 你需要重新思考 `知己知彼(2)` 中的问题.

偷龙转凤(3)

现在的 `format_FLOAT()` 已经可以正确地获得 `FLOAT` 类型的实参了. 你需要修改 `format_FLOAT()` 的实现, 把 `FLOAT` 类型数据的十进制小数表示作为格式化的结果. 我们约定格式化结果通过截断的方式保留6位小数.

瞒天过海

在 `print-FLOAT.c` 中, 传入 `FLOAT` 类型参数时都用到了宏 `FLOAT_ARG()`. 尝试不使用这个宏, 而是直接传入 `FLOAT` 类型参数, 观察输出结果, 你发现了什么问题? 为什么会这样?

最后, 我们需要清除这段代码中的其它浮点指令, 来让print-FLOAT可以在NEMU中运行. 清除的方式很简单, 只需要用 `nop` 指令覆盖它们即可.

漏网之鱼

乍看之下, 如果不清除这些浮点指令, print-FLOAT-linux的运行也不会受到影响. 但如果通过 `%f` 进行格式化的次数足够多, 还是会出现问题的. 尝试在print-FLOAT-linux中进行10次含有 `%f` 的输出, 观察输出结果. 你知道为什么会出现这样的情况吗?

另外在NEMU中运行之前, 记得去掉代码中的 `mprotect()` 系统调用, 因为目前NEMU还不能支持系统调用的执行.

偷龙转凤(4)

在NEMU中运行print-FLOAT, 你会发现仍然遇到浮点指令. 仔细分析后, 你会发现这次遇到的浮点指令位于 `_ppfs_setargs()` 中. 你的最后一个任务就是编写 `lib-common/FLOAT/FLOAT_vfprintf.c` 中的 `modify_ppfs_setargs()` 函数, 让其对 `ppfs_setargs()` 函数的运行时二进制进行修改, 达到绕过上述浮点指令的目的.

主要的相关代码是一个switch-case语句, 通过不同的格式说明符来获取不同长度的数据. `%f` 对应的是一个64位的数据(`float` 类型不是32位吗?), 事实上也可以把它看成一个 `long long` 类型的数据来获取, 这样就能绕过 `float` 相应分支的浮点指令来获取64位的数据了. 你的修改目标就是在 `float` 分支的开头放置一条 `jmp` 指令, 跳转到 `long long` 分支. 更多的代码细节请阅读 `modify_ppfs_setargs()` 函数中的注释. 有了修改 `_vfprintf_internal()` 函数的经验, 这次的任务当然也难不倒聪明的你啦!

实现正确后, 你就可以在NEMU中成功运行用户程序print-FLOAT了.

不过在真实的操作系统中, 要进行这种劫持代码的攻击其实并非易事. 回顾我们在GNU/Linux的实现, 我们需要先通过 `mprotect()` 系统调用打开代码的可写权限, 才能进行劫持. 而一般的程序并不会主动打开代码的可写权限, 因此黑客们也在绞尽脑汁思考其它更巧妙的攻击方式. 不过可以肯定的是, 没有对计算机系统的深入理解, 是不可能写出这样的代码的.

和代码玩游戏

在程序设计基础课上, 你学会了如何对数据做一些基本操作, 学会了play with the data, 例如算阶乘, 用链表组织学生信息, 但你从来都不知道代码是什么东西. 而上述练习其实就是在play with the code, 你会发现修改程序的行为犹如探囊取物. 和代码玩游戏的最高境界就要数self-modifying code了, 它们能够在运行时刻修改自己, 但这种代码极难读懂, 维护更是难上加难, 因此它们成为了黑客们炫耀的一种工具. 事实上, 你刚才已经编写了self-modifying code!

这一切是否引起你的思考: 代码的本质是什么? 代码和数据之间的区别究竟在哪里?

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/helper.h` 中, 你会看到由 `static inline` 开头定义的 `instr_fetch()` 函数和 `idex()` 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 请描述你在工程目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `obj/nemu/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, 函数, 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 - 虚实交错的魔法: 存储管理

世界诞生的故事 - 第三章

上帝已经创造出了一个完整的处理器, 通过它, 这个世界已经可以完成多得连上帝自己也出乎意料的事情. 上帝觉得这个世界还不够完美, 于是他把目光转向存储器, 希望通过存储器来完善这个美妙世界的法则.

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cache和二级cache - 2016/11/20 23:59:59
- 阶段2: 实现IA-32分段机制 - 2016/11/27 23:59:59
- 阶段3: 实现IA-32分页机制 - 2016/12/04 23:59:59
- 最后阶段: 实现所有要求, 提交完整的实验报告 - 2016/12/11 23:59:59

提交说明: 见这里

Cache的故事

不可逾越的鸿沟

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

而CPU的寄存器采用的是SRAM, 是通过一个触发器来存储一个bit, 具体来说就是4-6个晶体管, 只要不断电, SRAM中的数据就不会丢失, 不需要定时刷新, 也不需要预充电, 读写速度随着主频的提升而提升.

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

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

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

近水楼台先得月

Cache工作方式实际上是局部性原理的应用:

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

访问, 这就是空间局部性. 例如顺序执行代码, 或者是扫描数组元素.

相应的:

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

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

- 一个内存区域可以被映射到多少个cache block? 少了容易冲突, 多了电路逻辑和功耗都会上升. 对这个问题的回答划分了不同的cache组织方式, 包括direct-mapped(直接映射), set associative(组相联)和fully associative(全相联).
- 冲突的时候, 需要替换哪一个cache block? 这个问题的回答涉及到替换算法, 最理想的情况是替换那个很长时间都没访问过的cache block, 这就是LRU算法. 但这对硬件实现来说太复杂了, 对于8-way set associative来说, 每一个set中的8个cache block都有 `8! = 40320` 种可能的访问情况, 编码至少需要16个bit, 译码则需要更大的代价, 电路逻辑和时延都会上升, 因此实际上会采用伪LRU算法, 近似记录cache block的访问情况, 从而降低硬件复杂度. 也有研究表明, 随机替换的效果也不会很差.
- 写cache的时候要不要每次都写回到内存? 这个问题涉及到写策略, write through(写通)策略要求每次cache的写操作都同时更新内存, cache中的数据和内存中的数据总是一致的; write back(写回)策略则等到cache block被替换才更新内存, 就节省了很多内存写操作, 但数据一致性得不到保证, 最新的数据有可能在cache中. 数据一致性在多核架构中是十分重要的, 如果一个核通过访问内存拿到了一个过时的数据, 用它来进行运算得到的结果就是错误的.
- 写缺失的时候要不要在cache中分配一个cache block? 分配就更容易引起冲突, 不分配就没有用到时间局部性.

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

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

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

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

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

理解DRAM的工作方式

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

在NEMU中实现cache

Cache的工作方式并不复杂(太复杂就不可能用硬件来实现了), 要在NEMU中模拟cache也并非难事. 从cache组织的角度来看, cache由若干cache block构成, 每一个cache block除了包含相应的存储空间之外, 还包括tag和一些标志位, 你可以很容易地使用结构体来表示一个cache的组织. 从操作的角度来看, 我们只需要提供cache的读和写两种操作就可以了.

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

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

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

} Type1;

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

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

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

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

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

关于cache具体如何工作, 课上都已经详细讲过, 这里就不另外叙述了. 一个需要注意的地方是"读写数据跨越cache block的边界", 这时候你需要通过两次读写cache的操作来完成它. 框架代码提供了一个专门用于读写不对齐内存区域的宏 `unalign_rw()` (在 `nemu/include/macro.h` 中定义), 使用它可以较方便地处理上述情况. 值得一提的是维基百科中的CPU cache页面, 里面除了课堂上讲过的知识, 还有诸多延伸, 值得仔细琢磨.

实现cache

在NEMU中实现一个cache, 它的性质如下:

- cache block存储空间的大小为64B
- cache存储空间的大小为64KB
- 8-way set associative
- 标志位只需要valid bit即可
- 替换算法采用随机方式
- write through

- no-write allocate

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

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

简易调试器(3)

为了方便调试, 你可以在monitor中添加如下命令:

```
cache ADDR
```

这条命令的功能是使用地址 `ADDR` 来查找cache, 当查找成功时, 输出相应cache block的内容和标志位; 查找失败时, 输出失败信息, 而不是读取DRAM来填写cache. 你可以根据你的实际需要添加或更改这条命令的功能.

此外, 由于简易调试器中使用 `swaddr_read()` 来进行内存读取操作(如 `x` , `p` 等), 使用它们会影响cache中的内容, 对cache的调试造成不便. 为此, 你可以修改cache处的代码, 加入对 `nemu_state` 进行判断的内容: 当 `nemu_state` 不为 `RUNNING` 时, 说明NEMU位于调试模式, 此时的cache也相应切换为no-read allocate, 即miss时完全不将数据装入cache, 而是直接从下一层存储结构中读到的数据直接返回. 这样就能保证cache的内容不受简易调试器的影响了.

观察cache的作用

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

矩阵乘法在工程应用中十分广泛, 如何让矩阵乘法算得更快也曾经成为一个研究热点. 从局部性原理的角度来进行优化的一个算法是Cannon算法, 有兴趣的同学可以看看它是怎么工作的.

NEMU作为一款模拟器, 它的好处在这里体现得淋漓尽致, 你可以轻而易举地修改一个cache的"构造", 马上重新开始统计新的数据, 而不需要做一个真正的cache才开始测试. 自从cache的概念被提出来, 无数的研究者提出了五花八门的cache, 学术界中研究cache的论文更是数不胜数, 但被工业界采纳的cache研究却寥寥无几, 究其原因只有一个 -- 纸上谈兵, 无法用硬件实现. NEMU作为一个教学实验, 旨在让大家巩固课堂知识, 并不要求大家实现

一个真正的cache, 但也希望大家能从中明白一个道理: 做事情要落到实处才有价值(理论工作除外). 如果你对cache的硬件实现感兴趣, 可以尝试用verilog写一个direct-mapped, 只有4项, 可综合的小cache.

实现二级cache

在NEMU中实现一个L2 cache, 它的性质如下:

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

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

Icache和Dcache(选做, 请谨慎尝试!)

现代处理器一般有两个L1 cache, 即Icache(指令cache)和Dcache(数据cache), 这是因为Icache只有读操作, 硬件上容易实现; 同时读指令比读数据重要得多, 如果读数据缺失, 现代CPU的乱序执行机制可以找其它合适的指令先执行, 但如果读指令也缺失, CPU就只能等了. 为了尽可能提高读指令的命中率, 将Icache和Dcache分开是一种不错的方法.

你也可以尝试在NEMU中分别实现Icache和Dcache, 让 `instr_fetch()` 函数访问Icache, 其余访问数据的接口函数访问Dcache. 但在这之前, 请你深思熟虑: 是不是这样就万事大吉了? 还有没有什么需要考虑的问题? 如果有的话, 要怎么解决?

温馨提示

PA3阶段1到此结束.

IA-32的故事

从Segmentation fault说起

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

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

编译运行后, 你会看到屏幕上输出了

```
Segmentation fault
```

段错误还有其它的类型, 例如

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

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

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

再深入一步, CPU是怎么捕获这些非法操作的? 答案就在接下来的故事中. 在这之前, 你可能会问"为什么类似cli的非法操作也会称为段错误?" 实际上这是有历史原因的, 在以前的CPU架构中没有这么强大的保护功能, 一般的非法操作都和分段机制有关(例如内存访问超越了段界限),

因此被称为Segmentation fault. 但这种称呼直到分页机制诞生之后也没有改变, 于是便一直沿用至今. 更多关于Segmentation fault的内容可以阅读这里.

混沌初开

故事追溯到IA-32诞生之前, 在那时, 8086曾经主宰了计算机的一切. 那是一个16位的世界, 所有寄存器都是16位的, 貌似最多只能访问 `2^16=64KB` 的内存. 但事实并非如此, 8086通过引入一系列的段寄存器(segment register)来开辟更广阔的世界:

> physical_address = (seg_reg << 4) + offset

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

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

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

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

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

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

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

这些约定一直沿用至今.

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

为什么要采用这种有重叠的寻址方式?

在 `seg_reg` 和 `offset` 的搭配下, 总共有32位的信息(每个寄存器分别是16位), 而要访问1MB的内存只需要20位的信息就足够了, 浪费了其中12位的信息导致了段之间的重叠. 为什么不充分利用这12位的信息? 或者说让段寄存器都变成4位?

这是一个open problem, 尝试一下你会提出什么理由来支持8086的设计. 如果你对这个问题感兴趣, 你可以到互联网上搜索相关内容.

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

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

NEMU的"实模式"

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

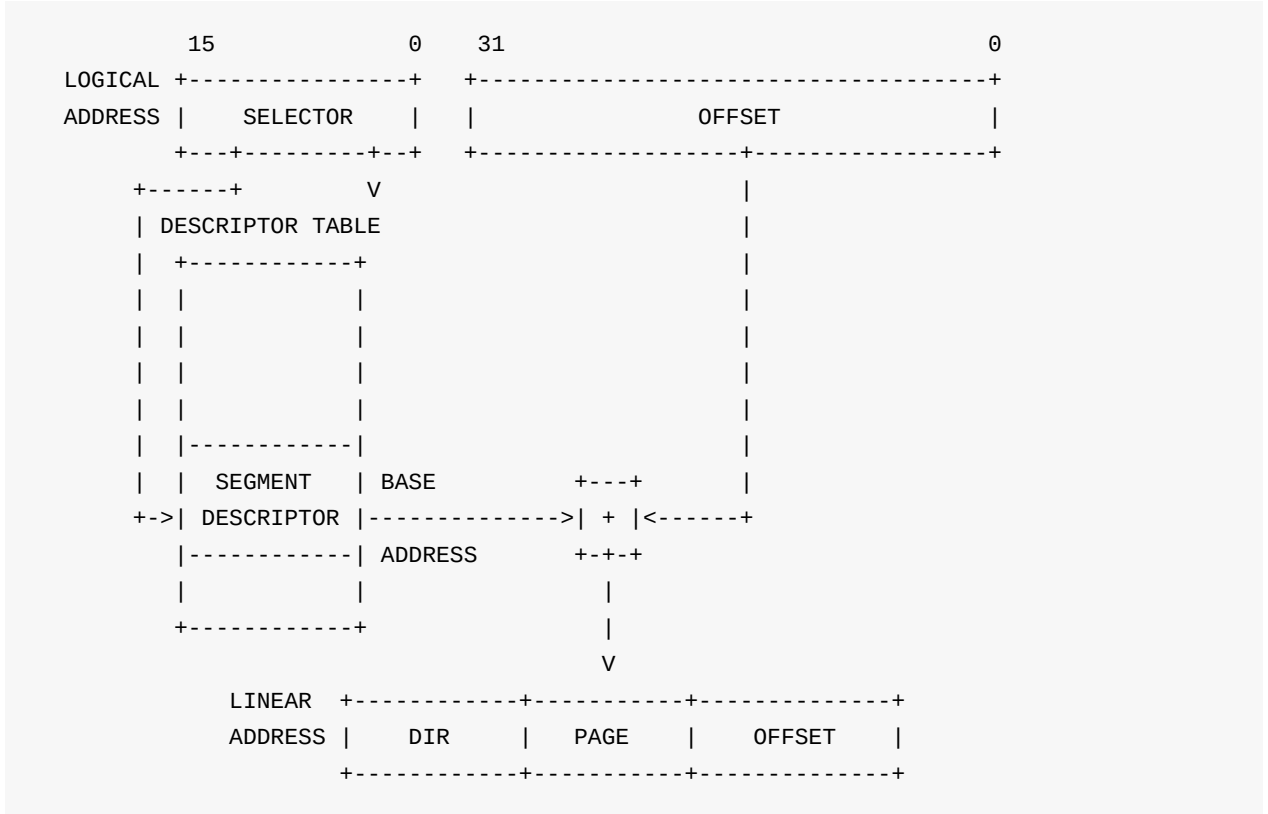
建立新秩序

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

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

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

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



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

- 1MB内存容量的瓶颈
- 恶意程序等安全问题

请记住, 分段过程搞得如此"复杂", 就是为了解决这两个历史遗留问题.

豁然开朗的视野

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

```
        DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

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

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

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

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

80386想出了一种同时解决这两个问题的方法, 那就是你经常使用的数组! 80386把内存中的某一段数据专门解释成一个数组, 名字叫GDT(Global Descriptor Table, 全局描述符表), 数组的一个元素就是一个段描述符. 这样一来就可以通过下标索引的方法来找到所有的段描述符啦. 于

是在80386的世界里, 原来的段寄存器就用来存放段描述符的索引, 另外还包含了一些属性, 这样的一个结构叫段选择符(Selector)(更正: i386手册相应的图中位域标识有误, 实际上RPL占2bit, TI占1bit):

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

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

GDT能有多大?

你能根据段选择符的结构, 计算出GDT最大能容纳多少个段描述符吗?

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

为什么是线性地址?

GDTR中存放的GDT首地址可以是虚拟地址吗? 为什么?

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

现在回去看那个好像很厉害的图, 你已经可以理解80386的分段机制了:

1. 通过段寄存器中的段选择符TI位决定在哪个表中进行查找
2. 根据GDTR或LDTR读出表的首地址
3. 根据段寄存器中的段选择符的index位在表中进行索引, 找到一个段描述符
4. 在段描述符中读出段的基地址, 和虚拟地址(也称逻辑地址)相加, 得出线性地址

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

如何提高寻找段描述符的效率?

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

耍一耍CPU

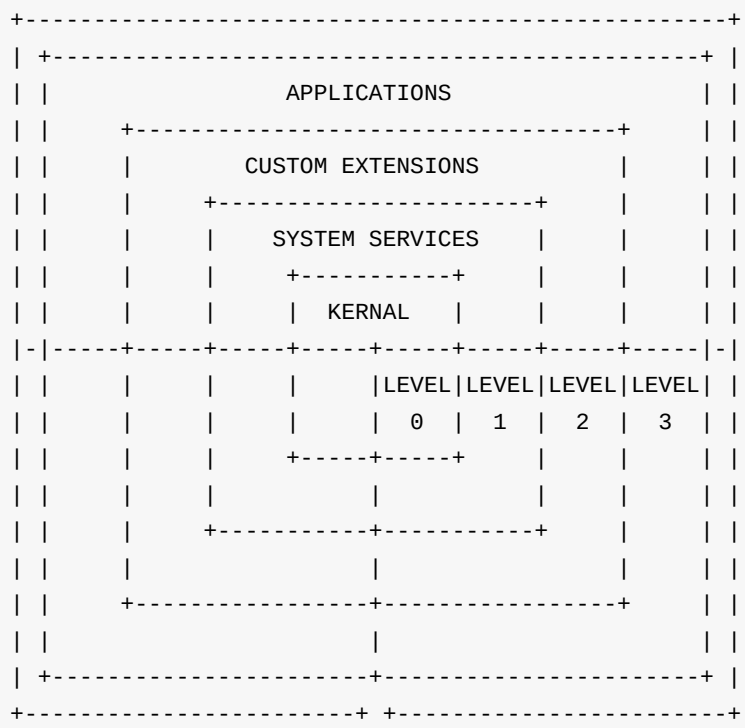
把一张图片的首地址和大小装载到GDTR(在NEMU中你确实可以这样做, 用 `xxd` 命令把一张图片的内容转化成一个数组, 然后修改kernel的代码, 把这个数组的首地址和大小装载到GDTR), 想象一下会发生什么? CPU会如何处理这个"GDT"? 为什么会这样?

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

等级森严的制度

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

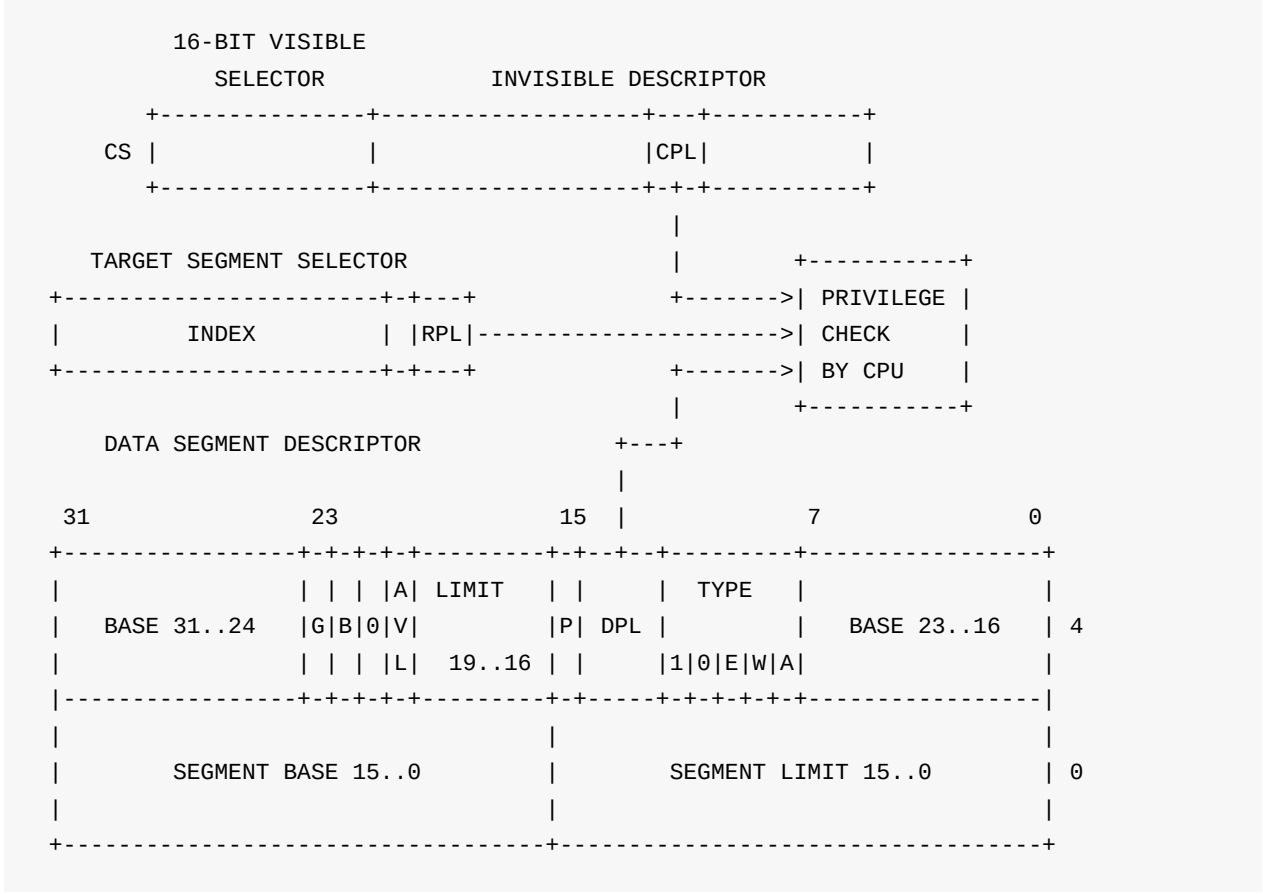
在80386的新世界里, 存在0, 1, 2, 3四个特权级, 0特权级最高, 3特权级最低. 特权级n所能访问的资源, 在特权级0~n也能访问. 不同特权级之间的关系就形成了一个环: 内环可以访问外环的资源, 但外环不能进入内环的区域, 因此也有"ring n"的说法来描述一个进程所在的特权级.



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

- 在段描述符中含有一个DPL域(Descriptor Privilege Level), 它描述了一个段所在的特权级
- 在段选择符中含有一个RPL域(Requestor's Privilege Level), 它描述了请求者所在的特权级
- CPL(Current Privilege Level)指示当前进程的特权级, 一般来说它和当前CS寄存器所指向的段描述符(也就是当前进程的代码段)的DPL相等

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



一次数据段的切换操作是合法的, 当且仅当

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

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

对RPL的补充

你可能会觉得RPL十分令人费解, 我们先举一个生活上的例子.

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

`requestor.RPL`), 但是你却没有权限直接对你的账户金额进行操作(`target_descriptor.DPL < current_process.CPL`), 因此你很有可能会被保安抓起来

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

通常情况下, 内核运行在ring 0, CPL为0, 因此有权限访问所有的段; 而用户进程运行在ring 3, CPL为3, 这就决定了它只能访问同样处在ring3的段. 这样, 只要操作系统内核将GDT, 以及下文将要提到的页表等重要的数据结构放在ring 0的段中, 恶意程序就永远没有办法访问到它们.

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

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

在NEMU中实现分段机制

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

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

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

- 在 `CPU_state` 结构中添加GDTR, CR0和各种段寄存器, 包括CS, DS, ES, SS, 它们的具体结构请参考i386手册. 80386中还引入了两个新的段寄存器GS和FS, 不过我们不会用到它们, 因此可以不模拟它们的功能. LDT我们也不会用到, 和LDT相关的内容也不必模拟. 你还需要在 `restart()` 函数中对CR0寄存器进行初始化, 让我们模拟的计算机在"开机"的

时候运行在"实模式"下.

- 添加 `lgdt` 指令.
- 添加 `opcode` 为 `0F 20` 和 `0F 22` 的 `mov` 指令, 使得我们可以设置/读出CR0. 设置CR0后, 如果发现CR0的PE位为1, 则进入IA-32保护模式, 从此所有虚拟地址的访问(包括 `swaddr_read()` 和 `swaddr_write()`)都需要经过段级地址转换.
- 为了实现段级地址转换, 你需要对 `swaddr_read()` 和 `swaddr_write()` 函数作少量修改. 以 `swaddr_read()` 为例, 修改后如下:

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

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

- 为了实现段寄存器的捆绑规则, 你还需要

1. 在 `Operand` 结构体中添加成员 `sreg` :

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

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

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

    - opcode为 `A0` , `A1` , `A2` , `A3` 的 `mov` 指令使用DS寄存器
    - 一些堆栈操作指令会隐式使用SS寄存器
    - `instr_fetch()` 总是使用CS寄存器
    - 在monitor中, `x` 和 `p` 命令读出内存时, 使用DS寄存器; `bt` 命令打印栈帧链时, 使用SS寄存器
    - 关于字符串操作指令使用的段寄存器, 请查阅i386手册
- 添加 `opcode` 为 `8E` 的 `mov` 指令, 使得我们可以设置段寄存器. 设置段寄存器时, 还需要将段的一些属性读入到段寄存器的描述符cache部分(在i386手册中被称为"隐藏部分", invisible part), 我们只需要读入段的base和limit就可以了, 其它属性在NEMU中不使用. 另外还有两点需要注意:
    1. GDTR中存放的GDT首地址是线性地址.
    2. IA-32中规定不能使用 `mov` 指令设置CS寄存器, 但切换到保护模式之后, 下一条指令的取指就要用到CS寄存器了. 解决这个问题的一种方式是在 `restart()` 函数中对CS寄存器的描述符cache部分进行初始化, 将base初始化为0, limit初始化为 `0xffffffff` 即可.
- 为了设置CS寄存器, 你需要实现 `ljmp` 指令, 即 `JMP ptr16:32` 形式的jmp指令, 其作用是"Jump intersegment, 6-byte immediate address", 更多信息请查阅i386手册.

在NEMU中实现分段机制

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

温馨提示

PA3阶段2到此结束.

迈进新时代

80386使用了升级版的段式存储管理, 听上去很不错, 但实际上并不是这样.

段式存储管理的缺点

回忆课堂内容, 段式存储管理有什么缺点? (说不定考试会出这道题喔 ^_^)

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

超越容量的界限

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

页式存储管理的优点

回忆课堂内容, 页式存储管理有什么优点? (说不定考试也会出这道题喔 ^_^)

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

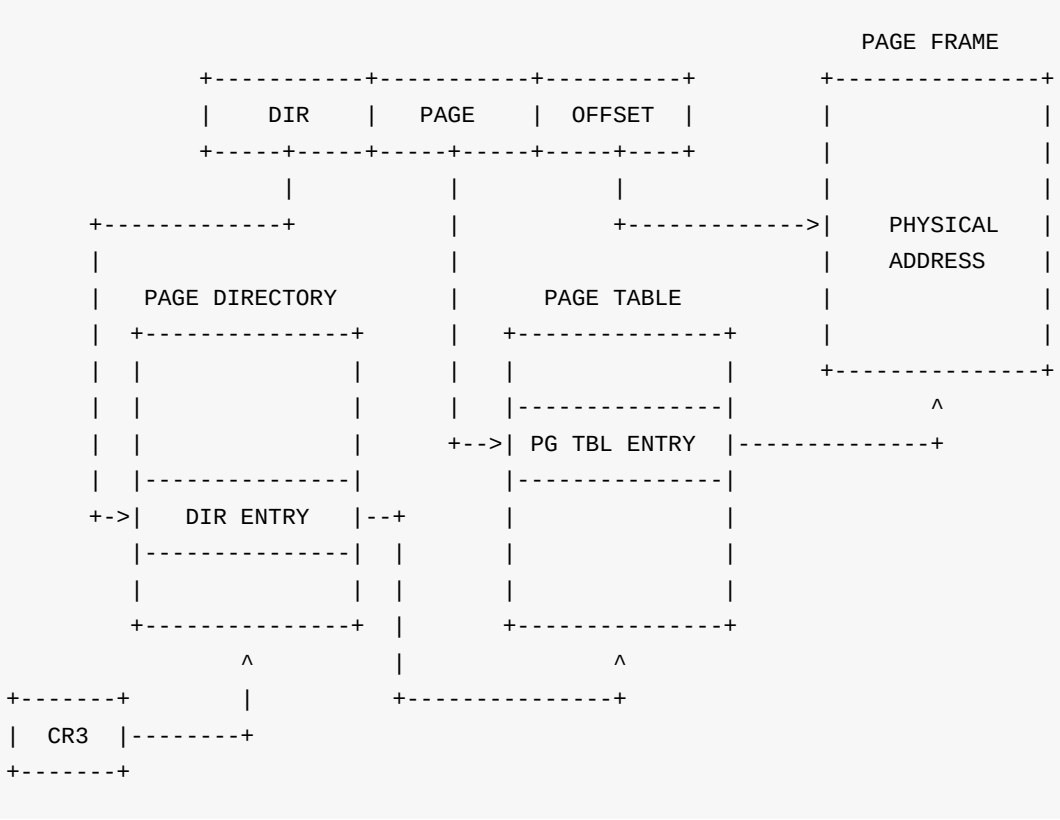
被"淘汰"的段式存储管理

gcc编译程序时也默认假设程序将来运行在扁平模式下. 你可以在 `kernel/src/start.S` 中进行以下尝试:

- 在kernel中为GDT增加一个新的段描述符, 它的基地址改为 `0x4000000` , 其它属性和数据段一样
- 把这个新的段描述符装载到SS寄存器, 作为新的堆栈段
- 把 `%esp` 的初值改成 `0x4000000`

修改后, 运行任意用户程序, 你会发现NEMU输出 `HIT BAD TRAP` 的信息. 分析相应的汇编代码, 你能厘清运行出错的根本原因吗?

我们也是先上一张图给大家一个感观上的认识:



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

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

一些问题

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

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

- present位表示物理页是否可用, 不可用的时候又分两种情况:

1. 物理页面由于交换技术被交换到磁盘中了, 这就是你在课堂上最熟悉的Page fault的情况之一了, 这时候可以通知操作系统内核将目标页面换回来, 这样就能继续执行了
2. 进程试图访问一个未映射的线性地址, 并没有实际的物理页与之相对应, 因此这就是一个非法操作咯

- R/W位表示物理页是否可写, 如果对一个只读页面进行写操作, 就会被判定为非法操作(还记得我们在PA2的最后在GNU/Linux下实施代码劫持攻击的时候引入的 `mprotect()` 系统调用吗?)
- 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. 我们知道段级地址转换是把虚拟地址转换成线性地址的过程, 页级地址转换是把线性地址转换成物理地址的过程, 撇去存储保护机制不谈, 我们可以把这两个过程分别抽象成两个数学函数:

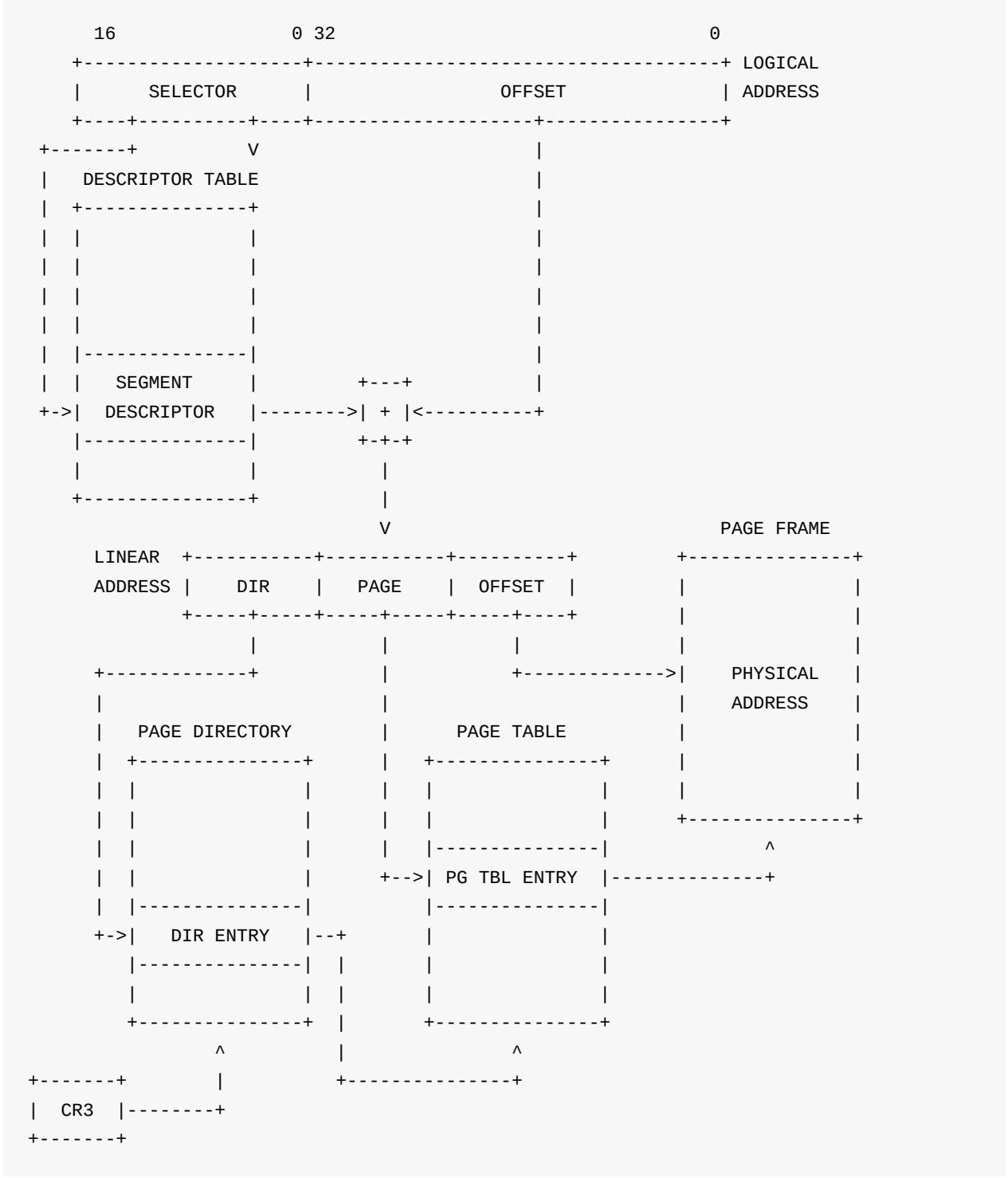
```
    y = seg(x) = seg.base + x
    y = page(x)
```

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 < HWADDR_MAX
=>    x < HWADDR_MAX - seg.base <= HWADDR_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. 这就已经把上面两点都解释清楚了.

最后我们用一张图来总结80386建立的新时代:



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

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

而"虚拟地址空间"和"物理地址空间"这两个在操作系统中无比重要的概念, 也只不过是这个复合函数的定义域和值域而已. 然而这个过程中涉及到的很多细节还是被我们忽略了. 理解80386这个新时代存在的意义和价值, 可以帮助你从更本质的角度来看待一些你在程序设计层次中感

到模糊不清, 或者甚至无法解释的问题, 这也是学习这些知识的价值所在.

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

在NEMU中实现分页机制

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

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

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

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

- 最后提醒一下页级地址转换时出现的一种特殊情况. 和之前实现cache的时候一样, 由于IA-32并没有严格要求数据对齐, 因此可能会出现数据跨越虚拟页边界的情况, 例如一条很长的指令的首字节在一个虚拟页的最后, 剩下的字节在另一个虚拟页的开头. 如果这两个虚拟页被映射到两个不连续的物理页, 就需要进行两次页级地址转换, 分别读出这两个物理页中需要的字节, 然后拼接起来组成一个完成的数据返回. MIPS作为一种RISC架构, 指令和数据都严格按照4字节对齐, 因此不会发生这样的情况, 否则MIPS CPU将会抛出异常,

> 可见软件灵活性和硬件复杂度是计算机科学中又一对tradeoff. 不过根据KISS法则, 你现在可以暂时不实现这种特殊情况的处理, 在判断出数据跨越虚拟页边界的情况之后, 先使用 `assert(0)` 终止NEMU, 等到真的出现这种情况的时候再进行处理.

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

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

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

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

团结力量大

上述文字提到了3个和内存地址相关的描述:

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

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

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

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

必答题

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

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

1. 首先让 `%esp` 加上 `KOFFSET` 转化成相应的虚拟地址, 然后通过间接跳转进入 `init_cond()` 函数继续进行初始化, 这样kernel就完全工作在虚拟地址中了.
2. 使用 `Log()` 宏输出一句话, 同样地, 现在的 `Log()` 宏还是不能成功输出.
3. 初始化MM. 这里的MM是指存储管理器(Memory Manager)模块, 它专门负责和用户进程分页相关的存储管理. 目前初始化MM的工作就是初始化用户进程的页目录, 同时将用户进程在 `0xc0000000` 以上的虚拟地址映射到和kernel一样的物理地址.

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

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

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

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

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

- 按照program header中的 `p_memsz` 属性, 为这一段segment分配一段不小于 `p_memsz` 的物理内存
- 根据虚拟地址 `p_vaddr` 和分配到的物理地址正确填写用户进程的页目录和页表
- 把ELF文件中的segment的内容加载到这段物理内存

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

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

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

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

不连续的物理页面

分配连续的物理页面是一个很强的要求, 当操作系统运行了一段时间, 并且运行的程序较多时, 空闲的物理页面会分布得十分零散, "分配一段很大的连续物理区间"这一要求通常很难被满足. 如果 `mm_malloc()` 允许分配不连续的物理页面, 你编写的 `loader()` 函数还能正确地加载程序吗? 思考一下可能会出现什么问题? 应该如何解决它?

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

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

此时用户进程已经完全运行在分页机制上了.

在NEMU中实现分页机制

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

简易调试器(4)

为了方便调试, 你可以在monitor中添加如下命令:

```
page ADDR
```

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

有本事就把我找出来!

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

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

暗藏杀机的分页机制(这些问题都有一定的难度, 请谨慎尝试)

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

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

```
--- kernel/src/main.c
+++ kernel/src/main.c
@@ -19,9 +19,11 @@
+volatile int x = 0;
 /* Initialization phase 1
  * The assembly code in start.S will finally jump here.
  */
 void init() {
 #ifdef IA32_PAGE
+     x = 1;
     /* We must set up kernel virtual memory first because our kernel thinks it
      * is located at 0xc0100000, which is set by the linking options in Makefile.
      * Before setting up correct paging, no global variable can be used. */
     init_page();
```

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

- 在刚刚调用 `init_page()` 的时候, 分页机制并没有开启. 但通过 `objdump` 查看kernel的代码, 你会发现 `init_page()` 函数在 `0xc0000000` 以上的地址, 为什么在没有开启分页机制的情况下调用位于高地址的 `init_page()` 却不会发生错误?
- 你已经画出 `init_page()` 函数中创建的映射了, 这个映射把两处虚拟地址映射到同一处物理地址, 请解释为什么要创建这样一个映射. 具体地, 在 `init_page()` 的循环中有这样两行代码:

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

尝试注释掉其中一行, 重新编译kernel并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.

- 在 `init()` 函数中, 我们通过内联汇编把 `%esp` 加上 `KOFFSET` 转化成相应的虚拟地址. 尝试把这行内联汇编注释掉, 重新编译kernel并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.
- 在 `init()` 函数中, 我们通过内联汇编间接跳转到 `init_cond()` 函数. 尝试把这行内联汇编注释掉, 改成通过一般的函数调用来跳转到 `init_cond()` 函数, 重新编译kernel并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.
- 在 `init_mm()` 函数中, 有一处代码用于拷贝 `0xc0000000` 以上的内核映射:

```
/* create the same mapping above 0xc0000000 as the kernel mapping does */
memcpy(&updir[KOFFSET / PT_SIZE], &kpdir[KOFFSET / PT_SIZE],
       (PHY_MEM / PT_SIZE) * sizeof(PDE));
```

尝试注释这处代码, 重新编译kernel并运行, 你会看到发生了错误. 请解释这个错误具体是怎么发生的.

为用户进程创建video memory映射(选做)

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

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

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

温馨提示

PA3阶段3到此结束.

近水楼台先得月(2)

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

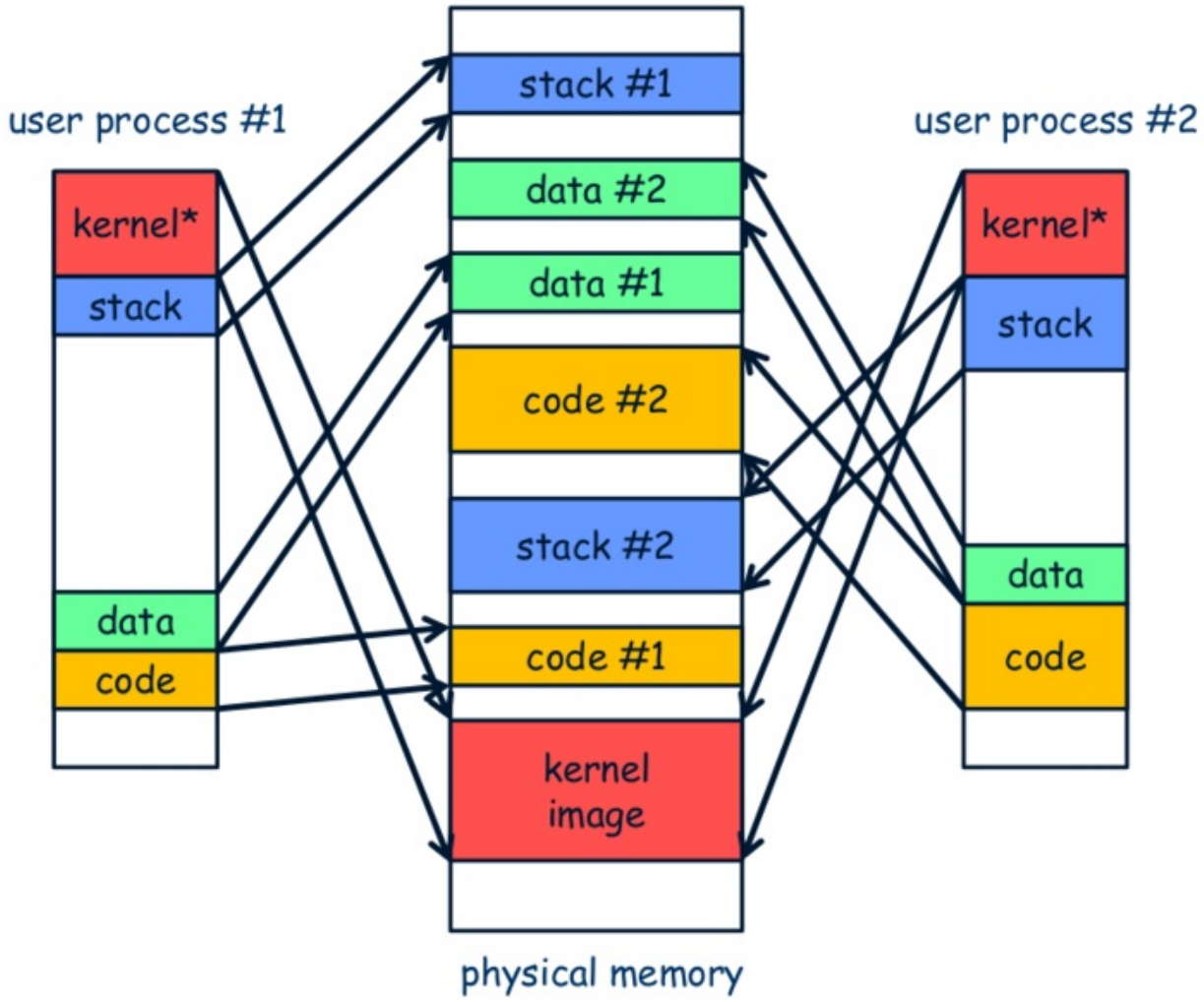
一个很自然的想法就是将页级地址转换的结果存起来, 在进行下一次的页级地址转换之前, 看看这个虚拟页是不是已经转换过了, 如果是, 就直接取出之前的结果, 这样就可以节省不必要的page walk了. 这不正好是cache的思想吗? 于是80386加入了一个特殊的cache, 叫TLB. 我们可以从CPU cache的知识来理解TLB的组织:

- TLB的基本单元是项, 一项存放了一次页级地址转换的结果(其实就是一个页表项, 包括物理页号和一些和物理页相关的标志位), 功能上相当于一个cache block.

- TLB项的tag由虚拟页号(即线性地址的高20位)来充当, 表示这一项对应于哪一个虚拟页号.
- TLB的项数一般不多, 为了提高命中率, TLB一般采用fully associative的组织方式.
- 由于页目录和页表一旦建立之后, 一般不会随意修改其中的表项, 因此TLB不存在写策略和写分配方式的问题.

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

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



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

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

实现TLB

在NEMU中实现一个TLB, 它的性质如下:

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

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

最后我们来聊聊cache和虚拟存储的关系. 课堂上我们提到的cache都是用物理地址来查找的, 引入虚拟存储的概念之后, 原则上我们也可以使用虚拟地址来查找. 从这个角度来考虑, 我们可以组合出4种cache:

- physical index, physical tag(PIPT)
- physical index, virtual tag(PIVT)
- virtual index, physical tag(VIPT)
- virtual index, virtual tag(VIVT)

其中PIPT就是我们在课堂上提到的情况, 用物理地址的中间部分作为index找到一个或若干个cache block, 然后用物理地址的高位部分作为tag检查是否匹配, 不过这要求每一次cache查找之前都需要经过地址转换. 但PIPT实现起来最简单, 其实你也已经实现PIPT了, 因此你不需要对代码作额外的改动. 而VIVT则可以先查找cache, 如果cache命中, 就不需要进行地址转换了,

可以直接取出数据. 不过VIVT和TLB一样, 需要维护进程和虚拟地址的一致性, 同时还有可能产生别名(aliasing)问题: 如果多个虚拟地址被映射到同一个物理地址, 就有可能造成物理上的一份数据以多个不同虚拟地址的拷贝存放在cache中, 这时如果对其中某一份拷贝进行写操作, 就要维护好这几份不同拷贝的数据一致性.

分析PIVT和VIPT

尝试根据cache和虚拟存储的知识, 分析PIVT和VIPT的性能和局限性. 特别地, 如果要在IA-32中实现一个VIPT cache, cache的组织方式有什么限制吗?

从一到无穷大

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

PC的足迹

PC的足迹系列博文对PC发展史作了简要的介绍, 上文提到的大部分术语都涵盖其中, 感兴趣的同学可以在茶余饭后阅读这些内容. 另外这里有一张x86系列发展的时间表, 在了解各种技术的同时, 不妨了解一下它们的时代背景.

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

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

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 - 来自外部的声音: 中断与I/O

世界诞生的故事 - 第四章

上帝已经创造出一个可以独立运行的美妙世界, 为了让这个世界更缤纷多彩, 上帝正在计划构建这个美妙世界和其它平行世界的通道.

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: 在NEMU中运行使用printf()输出的Hello程序 - 2016/12/18 23:59:59
- 阶段2: 移植打字小游戏 - 2016/12/25 23:59:59
- 最后阶段: 移植仙剑奇侠传, 并提交完整的实验报告 - 2017/01/11 23:59:59 (如无特殊原因, 迟交的作业将损失20%的成绩(即使迟了1秒), 请大家合理分配时间)

提交说明: 见这里

满足合理的需求

在之前的PA中, 用户进程都只能安分守己地在运行在NEMU上, 除了"计算"之外, 什么都做不了. 但像"输出一句话"这种合理的需求, 内核必须想办法满足它. 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, 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, 然后告诉操作系统内核. 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1个参数放在 `%eax` 中, 第二个参数放在 `%ebx` 中...). 为了让内核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, 然后陷入内核. 这个异常和非法操作产生的异常不同, 内核能够识别它是由系统调用产生的. 在GNU/Linux中, 这个异常通过int $0x80指令触发.

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 $14, %edx;"    // length
            "int $0x80");

    return 0;
}
```

用户进程执行上述代码, 就相当于告诉内核: 帮我把从 `str` 开始的14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GDT, 页表. 因此, 通常不存在一个系统调用用来获取GDT这些操作系统私有的信息.

事实上, 你平时使用的 `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 `fopen` 这个库函数用于创建并打开一个新文件, 或者

打开一个已有的文件, 在GNU/Linux中, 它封装了 `open` 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, `fopen` 封装了 `CreateFile` 系统调用, 如果在代码中直接使用 `CreateFile` 系统调用, 把代码放到GNU/Linux下编译就会产生链接错误, 即使链接成功, 程序运行的时候也会产生运行时错误.

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.

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, 缺页等. IA-32还向软件提供 `int` 指令, 让软件可以手动产生异常, 因此上文提到的系统调用也算是一种异常. 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.

为了方便叙述, 我们称触发异常之前用户进程的状态为A. 触发异常之后, CPU将会陷入内核, 跳转到操作系统事先设置好的异常处理代码, 处理结束之后再恢复A的执行. 可以看到, A的执行流程被打断了, 为了以后能够完美地恢复到被打断时的状态, CPU在处理异常之前应该先把A的状态保存起来, 等到异常处理结束之后, 根据之前保存的信息把计算机恢复到被打断之前的状态.

哪些内容表征了A的状态? 在IA-32中, 首先当然是EIP(instruction pointer)了, 它指示了A在被打断的时候正在执行的指令(或者下一条指令); 然后就是EFLAGS(各种标志位)和CS(代码段, CPL). 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. 此外, 通用寄存器(GPR, general propose register)的值对A来说还是有意义的, 而进行异常处理的时候又难免会使用到寄存器. 但硬件并不负责保存它们, 因此需要操作系统来保存它们的值.

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, 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.

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. 此外, IA-32提供了 `pusha` / `popa` 指令, 用于把通用寄存器的值压入/弹出堆栈, 但你需要注意压入的顺序, 更多信息请查阅i386手册.

等到异常处理结束之后, CPU将会根据堆栈上保存的信息恢复A的状态, 最后执行 `iret` 指令. `iret` 指令用于从中断或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. 这样用户进程就可以从A开始继续运行了, 在它看来, 这次时空之旅就好像没有发生过一样.

神奇的传送门

我们在上文提到, 用户进程触发异常之后将会陷入内核. "陷入内核"究竟是怎么样的一个过程? 具体要陷入到什么地方去? 要回答这些问题, 我们首先要认识IA-32中断机制.

在IA-32中, 异常事件的入口地址是通过门描述符(Gate Descriptor)来指示的. 门描述符有3种:

- 中断门(Interrupt Gate)
- 陷阱门(Trap Gate)
- 任务门(Task Gate)

关于中断会在下文作进一步的解释, 而任务门在实验中不会用到. 中断门和陷阱门的结构如下:

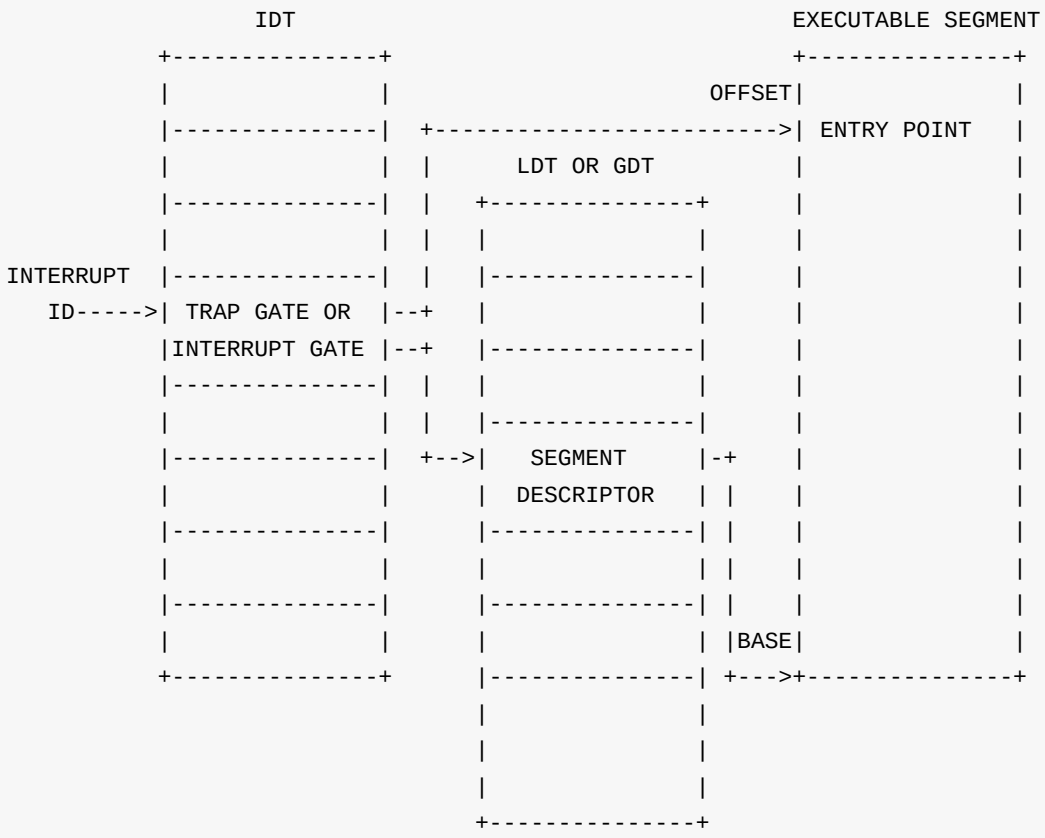
```
                            80386 INTERRUPT GATE
 31                23                 15                7                0
+-----------------+-----------------+---+---+---------+-----+-----------+
|           OFFSET 31..16           | P |DPL|0 1 1 1 0|0 0 0|(NOT USED) |4
+-----------------------------------+---+---+---------+-----+-----------+
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+

                            80386 TRAP GATE
 31                23                 15                7                0
+-----------------+-----------------+---+---+---------+-----+-----------+
|          OFFSET 31..16            | P |DPL|0 1 1 1 1|0 0 0|(NOT USED) |4
+-----------------------------------+---+---+---------+-----+-----------+
|             SELECTOR              |           OFFSET 15..0            |0
+-----------------+-----------------+-----------------+-----------------+
```

由于IA-32分段机制的存在, 我们必须通过段和段内偏移来表示跳转入口. 因此在中断门和陷阱门中, selector域用于指示目标段的段描述符, offset域用于指示跳转目标在段内的偏移. 这样, 如果能找到一个门描述符, 就可以根据门描述符中的信息计算出跳转目标了.

和分段机制类似, 为了方便管理各个门描述符, IA-32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(例如缺页异常为14号异常), 或者由 `int` 指令给出(例如 `int $0x80`). 最后, 为了找到IDT, IA-32中使用IDTR寄存器来存放IDT的首地址和长度. 操作系统需要事先把IDT准备好, 然后通过一条特殊的指令把IDT的首地址和长度装载到IDTR中, IA-32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:



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. 把门描述符中的selector域装入CS寄存器
5. 根据CS寄存器中的段选择符, 在GDT或LDT中进行索引, 找到一个段描述符, 并把这个段的一些信息加载到CS寄存器的描述符cache中
6. 在段描述符中读出段的基地址, 和门描述符中的offset域相加, 得出入口地址
7. 跳转到入口地址

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, 这只是一个大概的过程, 在真实的计算机上还会遇到很多特殊情况, 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.

特殊的原因

我们在上一小节中提到: "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". 究竟是什么特殊的原因, 使得EFLAGS, CS, EIP三个寄存器的值必须由硬件来保存? 尝试结合IA-32中断机制思考这个问题.

在计算机和谐社会中, 大部分神奇的传送门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` 指令需要进行特权级检查, 门描述符中的DPL域将会参与到特权级检查的过程中, 具体的检查规则我们就不展开讨论了, 需要了解的时候请查阅i386手册.

时空之旅大揭秘

让我们通过 `obj/testcase/hello-inline-asm` 这个具体的例子, 亲自体验一趟神秘的旅程. 我们会跟随 `hello-inline-asm` 程序一同探索这一时空之旅, 当我们从时空之旅结束归来, `hello-inline-asm` 程序也已经将信息传达出去, 我们的第一个任务也就完成了.

添加传送门

在踏上旅程之前, 你还需要在NEMU中实现IA-32中断机制, 只有这样才能敲开神奇的传送门.

一方面, 你需要在kernel中加入相关的代码, 你只需要在 `kernel/include/common.h` 中定义宏 `IA32_INTR` , 然后重新编译kernel就可以了. 重新编译后, kernel会在 `init_cond()` 函数中多进行两项初始化工作:

- 重新设置GDT.
- 设置IDT, 具体来说就是填写IDT中每一个门描述符, 设置完毕后通过 `lidt` 指令装载IDTR.

其它的工作和之前一样, 没有变化.

另一方面, 你需要在NEMU中添加中断机制的功能, 以便让上述代码成功执行. 具体的, 你需要:

- 添加IDTR和 `lidt` 指令, 注意IDTR中存放的IDT首地址是线性地址.
- 添加如下的 `raise_intr()` 函数来模拟IA-32中断机制的处理过程:

```
#include <setjmp.h></span>
extern jmp_buf jbuf;
```

void raise_intr(uint8_t NO) { /* TODO: Trigger an interrupt/exception with ``NO''.

```
 * That is, use ``NO'' to index the IDT.
 */</span>


<span class="hljs-comment">/* Jump back to cpu_exec() */</span>
longjmp(jbuf, <span class="hljs-number">1</span>);
```

}</code></pre>

其中 `longjmp()` 的功能相当于跨越函数的goto语句, 执行 `longjmp()` 会跳转到 `cpu_exec()` 中 `setjmp()` 的下一条指令, 这样从 `raise_intr()` 中"返回"之后就会马上继续执行 `cpu.eip` 所指向的指令, 也就是异常处理入口指令.

- 添加 `int` , `iret` , `cli` , `pusha` , `popa` 等指令. 关于 `int` 和 `iret` 指令, 实验中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. 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 s
    ize, a sign-extended value is pushed on the stack.
    ```

  - 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IA-32中断处理的代码放在 `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IA-32中断机制

你需要在NEMU中添加IA-32中断机制的支持, 如有疑问, 请查阅i386手册. 实现成功后在NEMU中运行 `hello-inline-asm` 程序, 你会看到在 `kernel/src/irq/irq_handle.c` 中的 `irq_handle()` 函数中触发了BAD TRAP. 这说明用户进程已经成功通过IA-32中断机制陷入内核, 你将会在下面的任务中修复"触发BAD TRAP"的问题.

神奇的longjmp

你可能会认为没有必要在 `raise_intr()` 函数中使用 `longjmp` , 直接一步一步返回到 `cpu_exec()` 中就可以了. 但如果需要实现处理器异常的识别和处理, `longjmp` 是最好的选择. 假设你打算在页级转换过程中, 当检测到页表项的present位为0时, 通过 `raise_intr(14)` 抛出缺页异常, 如果不使用 `longjmp` , 你将如何让代码从 `raise_intr()` 返回到 `cpu_exec()` ? 如果还需要实现无效指令异常的处理, 你又会怎么办?

你可以通过

```
man setjmp
man longjmp
```

查阅 `setjmp` 和 `longjmp` 的相关信息. 思考一下, 如果让你实现 `setjmp` 和 `longjmp` , 你将如何实现?

重新设置GDT

为什么要重新设置GDT? 尝试把 `init_cond()` 函数中的 `init_segment()` 注释掉, 编译后重新运行, 你会发现运行出错, 你知道为什么吗?

曲折的旅途

我们的第一次时空之旅被迫止于半路之中, 看来旅途并非我们想象中的那么顺利. 为了找到问题的原因, 我们需要仔细推敲途中的每一处细节.

用户进程执行 `int $0x80` 之后, CPU将会保存现场, 查阅kenrel设置好的IDT, 跳转到入口函数 `vecsys()` , 压入错误码和异常号 `#irq` , 跳转到 `asm_do_irq` . 在 `asm_do_irq` 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, 它记录了用户进程陷入内核时的状态, 注意到trap frame是在堆栈上构造的. 保存好用户进程的信息之后, kernel就可以随意使用通用寄存器了. 接下来代码将会把当前的 `%esp` 压栈, 并调用C函数 `irq_handle()` .

诡异的代码

`do_irq.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

框架代码在 `irq_handle()` 函数的开始处设置了 `panic()` , 你的任务是理解trap frame形成的过程, 然后重新组织 `kernel/include/irq.h` 中定义的 `TrapFrame` 结构体的成员, 使得这些成员声明的顺序和 `kernel/src/irq/do_irq.S` 中构造的trap frame保持一致. 实现正确之后, 去掉上述的 `panic()` , `irq_handle()` 以及后续代码就可以正确地使用trap frame了. 重新运行 `hello-inline-asm` 程序, 你会看到在 `kernel/src/syscall/do_syscall.c` 中的 `do_syscall()` 函数中触发了BAD TRAP.

`irq_handle()` 将会根据 `#irq` 确定异常事件的类型, 从而进行不同的处理. 用户进程执行的 `int $0x80` 会被识别成系统调用请求, 于是kernel会调用 `do_syscall()` 对相应的请求进行处理. 由于用户进程的所有现场信息都已经保存在trap frame中了, kernel很容易获取它们, `do_syscall()` 将根据用户进程之前设置好的系统调用号和系统调用参数进行处理. 我们找到了第二次触发BAD TRAP的原因: kernel并没有实现 `SYS_write` 系统调用的处理. 为了再次踏上旅程, 你需要在 `do_syscall()` 中添加 `SYS_write` 的系统调用.

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. 根据 `write` 系统调用的函数声明(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` 字节输出到屏幕上, 你需要思考如何从trap frame中获取这些信息.

现在又回到了最根本的问题了: 要怎么才能把内容输出到屏幕上? 在真实的计算机中, 这些内容最终是由设备来负责输出的, 但NEMU中还没有添加设备, 所以我们还是先借助NEMU的功能来完成输出吧. 我们使用以下内联汇编来"陷入"到NEMU中:

```
asm volatile (".byte 0xd6" : : "a"(2), "c"(buf), "d"(len));
```

这正是我们在NEMU中手工加入的 `nemu_trap` 指令. 接下来修改 `nemu/src/cpu/exec/special/special.c` 中 `nemu_trap()` 的helper函数, 如果 `%eax` 为2, 则输出 `%ecx` 为首地址的 `%edx` 字节的内容. 在NEMU中输出就是一件易如反掌的事情了, 不过需要注意, `%ecx` 表示的地址是虚拟地址, 聪明的你知道应该怎么做了吧.

回到kernel中, 假设执行完上述的 `nemu_trap` 指令后, "输出"成功了, 处理系统调用的最后一件事就是设置系统调用的返回值. GNU/Linux约定系统调用的返回值存放在 `%eax` 中, 所以我们只需要修改 `tf->eax` 的值就可以了. 至于 `write` 系统调用的返回值是什么, 请查阅 `man 2 write` .

系统调用处理结束后, 代码将会一路返回到 `do_irq.S` 的 `asm_do_irq()` 中. 接下来的事情就是恢复用户进程的现场, kernel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, 最后通过 `iret` 指令恢复用户进程的EIP, CS, EFLAGS. 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.

这样, 一次系统调用就执行完了, 时空之旅圆满结束.

实现系统调用

你需要在kernel中添加 `SYS_write` 系统调用, 让 `hello-inline-asm` 程序成功输出"Hello, world!"的信息.

运行hello程序

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. 至于格式化的功能, uclibc已经为我们实现好了.

我们这次的任务是在NEMU中执行 `obj/testcase/hello` 程序. `SYS_brk` 系统调用用于调整用户进程堆区的大小, kernel中已经实现了这个系统调用了. `hello` 程序编译通过之后, 你就可以在NEMU中运行它了.

在NEMU中输出"Hello world"

在NEMU中运行 `obj/testcase/hello` 程序.

温馨提示

PA4阶段1到此结束.

天外有天的世界

我们之前让NEMU伸出上帝之手, 用户进程才能输出一句话, 怎么说都有点作弊的嫌疑. 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. 你应该已经(或将要)在数字电路实验中看到键盘控制器和VGA控制器的verilog代码, 只要向设备控制器发送一些有意义的信号, 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, 例如键盘控制器有一个把按键的模拟信号转换成扫描码的部件, 而VGA控制器则有一个把像素颜色信息转换成显示器模拟信号的部件. 如果把控制设备工作的信号称为"设备指令", 设备控制器的工作就是负责接收设备指令, 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.

巴别之塔

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, 就需要向设备发送设备指令. 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设备指令?

对第一个问题的回答涉及到I/O的寻址方式. 一种I/O寻址方式是端口映射I/O(port-mapped I/O), CPU使用专门的I/O指令对设备进行访问, 并为设备中允许CPU访问的寄存器逐一编号, 这些编号叫端口号. 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, 它们是由设备控制器自己维护的, 它们没有端口号, CPU不能直接访问它们.

IA-32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 `kernel/src/irq/i8259.c` 的代码, 代码使用 `out` 指令与Intel 8259中断控制器进行通信, 给控制器发送命令字. 例如

```
movl $0x11, %eax
movb $0x20, %dl
outb %al, (%dx)
```

上述代码把数据0x11传送到0x20号端口所对应的设备寄存器中. 你要注意区分I/O指令和设备指令, I/O指令是CPU执行的, 作用是对设备寄存器进行读写; 而设备指令是设备来执行的, 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0x11这个数据传送到中断控制器的一个控制寄存器中, 中断控制器接收到0x11后, 把它解释成一条设备指令, 发现是一条初始化指令, 于是就会进入初始化状态; 但对CPU来说, 它并不关心0x11的含义, 只会老老实实地把0x11传送到0x20号端口, 至于设备接收到0x11之后会做什么, 那就是设备自己的事情了.

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

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

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

不可缓存(uncachable)的内存区域

我们知道使用cache可以提高访问内存的速度, 但是对于用作内存映射I/O的物理地址空间, 使用cache却可能造成致命的错误. 因此一般会将这部分物理地址空间设置为不可缓存, 这样, 每一次对它们的访问都不会经过cache, 而是老老实实地访问相应的"内存区域". 你知道为什么要这样做吗?

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?

你已经见识过IA-32引入的保护机制对构造计算机和谐社会所做出的贡献了. IA-32所倡导的和谐是全方面的, 自然也不希望恶意程序随意访问设备. 针对端口映射I/O, IA-32规定了可以使用I/O指令的最低特权级, 它用EFLAGS寄存器的IOPL位来表示. 如果当前进程的CPL在数值上大于IOPL, 使用 `in` / `out` 指令将会导致CPU抛出异常, 于是你在GNU/Linux中又看到了熟悉的段错误了.

内存映射I/O的保护机制

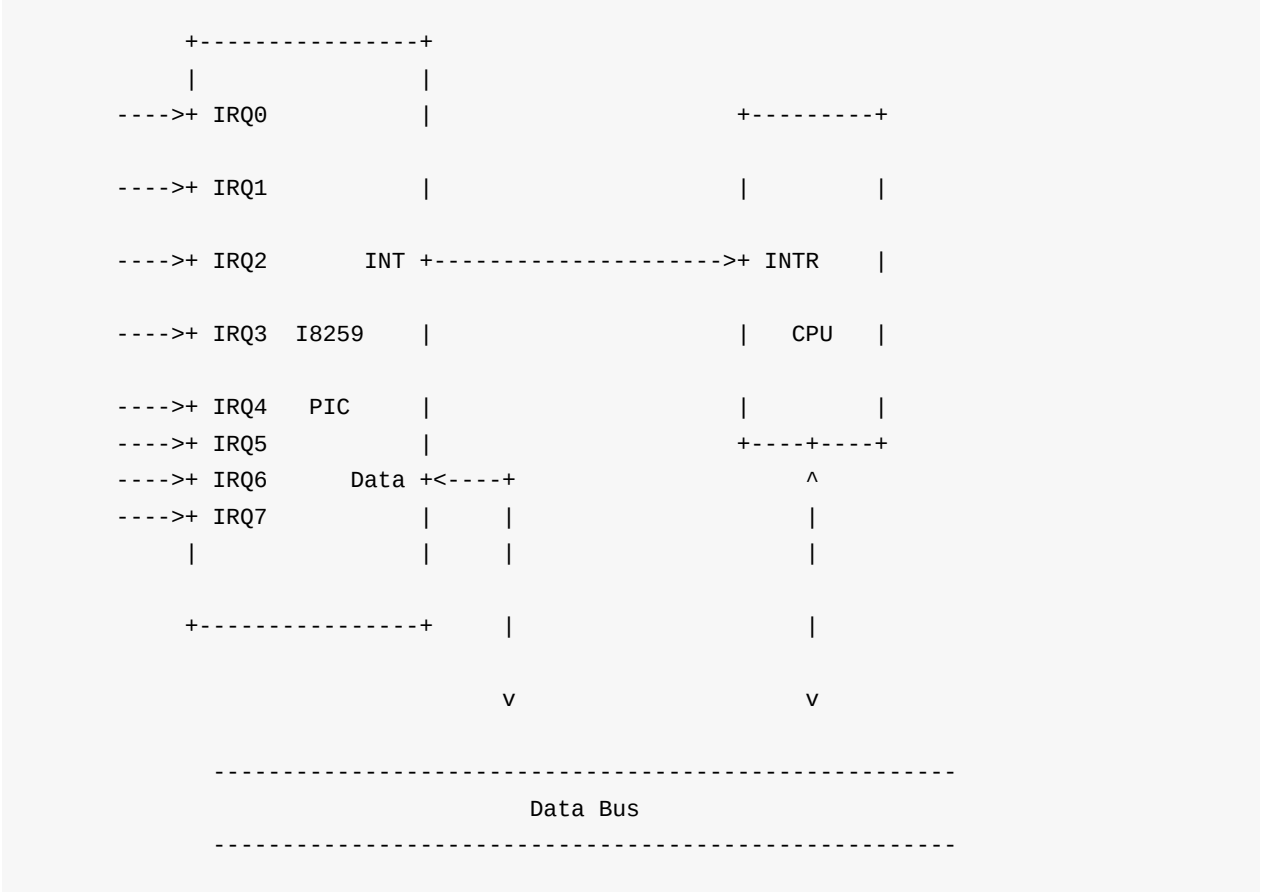
思考一下, IA-32怎么样对内存映射I/O进行保护? 尝试查阅i386手册对比你的想法.

来自外部的声音

让CPU一直监视设备的工作可不是明智的选择. 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汇报机制: 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, 主动向CPU汇报, 这时CPU才继续执行与磁盘相关的代码. 这样的汇报机制就是硬件中断. 硬件中断的实质是一个信号, 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.



如上图所示, i8259有8个不同的中断请求引脚IRQ0-IRQ7, 可以识别8种不同的硬件中断来源. 在i8259内部还有一些中断屏蔽逻辑和判优逻辑, 当多个中断请求引脚都有中断请求到来时, i8259会根据当前的设置选择一个未被屏蔽的, 优先级最高的中断请求, 根据该中断请求的所在的引脚生成一个中断号, 将中断号发送到数据总线上, 并将INT引脚置为高电平, 通知CPU有中断请求到来.

上面描述的是中断控制器最简单的工作过程, 在真实的机器中还有很多细节问题, 例如i8259的级联, 多个设备共享同一个IRQ引脚等, 这里就不详细展开了, 更多的信息可以查看i8259手册, 或者在互联网上搜索相关信息.

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, IA-32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. 一般来说, 中断处理会在IDT中找到中断门描述符, 异常处理会在IDT中找到陷阱门描述符. 它们的唯一区别就是, 穿过中断门的时候, EFLAGS中的IF位将会被清零, 达到屏蔽其它外部中断的目的; 而穿过陷阱门的时候, IF位将保持不变.

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, 内核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.

如果没有中断的存在, 计算机的运行就是完全确定的, 根据当前的指令和计算机的状态, 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; 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.

加入最后的拼图

设备代码介绍

框架代码中已经提供了设备的代码, 位于 `nemu/src/device` 目录下. 代码中提供了两种I/O寻址方式, i8259中断控制器和五种设备的模拟. 为了简化实现, 中断控制器和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/i8259.c` 是对Intel 8259中断控制器的功能模拟. 代码模拟了两块i8259芯片级联的情况, 从片的INT引脚连接到主片的IRQ2引脚. `i8259_raise_intr()` 是面向设备的接口, 当设备需要发出硬件中断时, 就会调用 `i8259_raise_intr()` , 代码会根据当前的中断请求状态选择一个优先级最高的中断, 并生成相应的中断号, 把中断号记录在 `intr_NO` 变量中, 然后把CPU的INTR引脚置为高电平, 通知CPU有硬件中断到来. CPU如果发现有硬件中断到来, 可以通过 `i8259_query_intr()` 查询当前优先级最高的中断号, 并调用 `i8259_ack_intr()` 向中断控制器确认收到中断信息, 中断控制器收到CPU的确认后会更新中断请求的状态, 清除刚才发送给CPU的中断请求. 为了简化, 中断控制器中没有实现中断屏蔽位的功能.
- `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.
- `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, 每当用户敲下/释放按键时, 将会把键盘扫描码放入数据寄存器, 然后发起键盘中断, CPU收到中断后, 可以通过端口I/O访问数据寄存器, 获得键盘扫描码.
- `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.
- `nemu/src/device/ide.c` 模拟了磁盘的功能. 磁盘初始化时会注册 `0x1F0` 处长度为8个字节的端口作为其寄存器, 并把NEMU运行时传入的测试文件当做虚拟磁盘来使用. 磁盘读写以扇区为单位, 进行读写之前, 磁盘驱动程序需要把读写的扇区号写入磁盘的控制寄存器, 然后往磁盘的命令寄存器中写入读/写命令字. 进行读操作时, 驱动程序可以从磁盘的

数据寄存器依次读出512个字节; 进行写操作时, 驱动程序需要向磁盘的数据寄存器依次写入512个字节.

- `nemu/src/device/vga.c` 模拟了VGA的功能. VGA初始化时会注册了两个用于更新调色板的端口, 并注册了从 `0xa0000` 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 `320x200x8` 的图形模式, 一个像素占8个bit的存储空间, 因此在一幅图中最多能够同时使用256种颜色.
- `nemu/src/device/vga-palette.c` 定义了VGA的默认调色板. 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, 其中VGA不使用alpha的信息. 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. 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.
- `nemu/src/device/sdl.c` 中是和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. 在 `init_sdl()` 函数中会进行一些和SDL相关的初始化工作, 包括创建窗口, 设置默认调色板等. 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操作, 包括发送100Hz的时钟中断, 以25Hz的频率刷新屏幕, 以及检测是否有按键按下/释放, 若有, 则发送键盘中断. 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.

如何检测多个键同时被按下

你应该从数字逻辑电路实验中认识到和扫描码相关的内容了: 当按下一个键的时候, 键盘控制器将会发送该键的通码(make code); 当释放一个键的时候, 键盘控制器将会发送该键的断码(break code), 其中断码的值为通码的值+0x80. 需要注意的是, 断码仅在键被释放的时候才发送, 因此你应该用"收到断码"来作为键被释放的检测条件, 而不是用"没收到通码"作为检测条件.

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. 根据键盘扫描码的特性, 你知道这些功能是如何实现的吗?

提高磁盘读写的效率

阅读 `kernel/driver/ide/disk.c` 中的代码, 理解kernel读写磁盘的方式. 以读操作为例, 你会发现磁盘中的每一个数据都先通过 `in` 指令读入到寄存器中, 然后再通过 `mov` 指令把读到的数据放回内存; 写操作也是类似的情况. 思考一下, 有什么办法能够提高磁盘读写的效率?

神奇的调色板

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?

我们提供的代码是模块化的, 为了让新加入的代码在NEMU中工作, 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_DEVICE` .
- 在cpu结构体中添加一个 `bool` 成员 `INTR` , 这个成员用于表示是否有外部中断到来, 它会在 `nemu/src/device/i8259.c` 中用到.
- 在 `init_monitor()` 函数中加入初始化设备和SDL的代码:

  ```
  init_device();
  init_sdl();
  ```

  代码在模拟某些设备的功能时用到了SDL库, 为了编译新加入的代码, 你需要先安装SDL库:

  ```
  apt-get install libsdl1.2-dev
  ```

  安装成功后, 修改 `nemu/Makefile.part` , 把SDL库加入链接对象:

  ```
  --- nemu/Makefile.part
  +++ nemu/Makefile.part
  @@ -4,1 +4,1 @@
  -nemu_LDFLAGS := -lreadline
  +nemu_LDFLAGS := -lreadline -lSDL
  ```

  之后重新编译.

配置X环境

如果你使用PA0中的Docker镜像, 你会发现编译后运行NEMU会提示以下错误:

```
(*) DirectFB/Core: Single Application Core. (2012-05-20 13:17)
(!) Direct/Util: opening '/dev/fb0' and '/dev/fb/0' failed
    --> No such file or directory
(!) DirectFB/FBDev: Error opening framebuffer device!
(!) DirectFB/FBDev: Use 'fbdev' option or set FRAMEBUFFER environment variable.
(!) DirectFB/Core: Could not initialize 'system_core' core!
    --> Initialization error!
nemu: src/device/sdl.c:57: init_sdl: Assertion `ret == 0' failed.
(!) [ 2095:    0.000] --> Caught signal 6 (unknown origin) <--
```

为了运行加入SDL库后的NEMU, 你还需要进行一些额外的配置.

下载X Server

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XServer, 你不需要额外下载.

为SSH打开X11转发功能

根据主机操作系统的类型, 你需要进行不同的操作:

- Mac用户和GNU/Linux用户. 在运行 `ssh` 时加入 `-X` 选项即可, `-X` 选项会为SSH连接打开X11转发功能:

  ```
  ssh -X jack@192.168.99.100
  ```

- 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, 你就可以顺利运行加入SDL库后的NEMU了, 运行时你会看到一个新窗口弹出.

使用设备代码

上述代码只是提供了I/O寻址方式的接口, 你还需要在NEMU中编写相应的代码来调用这些接口. 具体的, 你需要:

- 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.
- 在 `hwaddr_read()` 和 `hwaddr_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DRAM.

你还需要在NEMU中添加和硬件中断相关的代码:

- 在 `cpu_exec` ()中for循环的末尾添加轮询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:

```
if(cpu.INTR & cpu.eflags.IF) {
  uint32_t intr_no = i8259_query_intr();
  i8259_ack_intr();
  raise_intr(intr_no);
}
```

- 添加 `hlt` 指令. 这条指令十分特殊, 执行这条指令后, CPU直到硬件中断到来之前都不会执行下一条指令. 实现的时候, 只需要在相应的helper函数中通过一个循环不断查看INTR引脚, 直到满足响应硬件中断的条件才退出循环. 需要注意的是, 如果在关中断状态下执行 `hlt` 指令, 响应硬件中断的条件将永远得不到满足, CPU将一直处于停止工作的状态, 永远无法执行下一条指令. (温馨提示: 如果你发现在实现 `hlt` 指令的时候遇到了困难, 请参考本实验中的某一道蓝框题.)

最后你需要在kernel中加入相关的代码, 你只需要在 `kernel/include/common.h` 中定义宏 `HAS_DEVICE`, 然后重新编译kernel就可以了. 重新编译后, kernel会在 `init_cond()` 函数中多进行一些和设备相关的工作:

- 初始化i8259. 由于NEMU中的i8259模拟实现是不可编程的, 因此它没有注册端口I/O的回调函数, 故kernel中对i8259的初始化并没有实际效果.
- 初始化串口. 如果你的 `out` 指令实现正确, 初始化串口后, 你就可以在kernel中使用 `Log()` 进行输出了. 同时 `SYS_write` 系统调用也不需要通过"陷入"NEMU来输出了, 修改kernel中 `sys_write()` 的代码, 通过 `serial_printc()` 把 `SYS_write` 系统调用中 `buf` 的内容输出到串口.
- 初始化IDE驱动程序. `kernel/src/driver/ide` 中实现了IDE驱动程序. 初始化工作包括:
  - 初始化IDE驱动程序的高速缓存.
  - 把 `ide_writeback()` 函数加入到时钟中断的中断处理函数. 每次时钟中断到达的时候, `ide_writeback()` 将会被调用, 它负责每经过1秒将高速缓存中的脏块写回磁盘, 进行写数据的同步.
  - 把 `ide_intr()` 函数加入到磁盘中断的中断处理函数. 每次磁盘中断到达的时候, `ide_intr()` 将会被调用, 它负责设置 `has_ide_intr` 标志, 记录磁盘中断的到来.
- 此时内核的底层初始化操作已经全部完成, 可以打开中断.
- IDE驱动程序封装了磁盘读写的功能, 并向上层提供了 `ide_read()` 和 `ide_write()` 两个方便使用的接口来读写磁盘. 端口I/O的功能实现正确后, 我们已经可以使用"真正"的磁盘, 而不需要使用ramdisk了. 定义宏 `HAS_DEVICE` 后, `kernel/src/elf/elf.c` 中的一处代码会把磁盘开始的4096字节读入一个缓冲区中, 这4096字节已经包含了ELF头部和program

header table了. 你需要修改加载loader模块的代码, 从磁盘读入每一个segment的内容. 修改后, 把 `nemu/include/common.h` 中定义的宏 `USE_RAMDISK` 注释掉, ramdisk就可以退休了.

- 为用户进程创建video memory的虚拟地址空间. 在 `loader()` 函数中有一处代码会调用 `create_video_mapping()` 函数(在 `kernel/src/memory/vmem.c` 中定义), 为用户进程创建video memory的恒等映射, 即把从 `0xa0000` 开始, 长度为 `320 * 200` 字节的虚拟内存区间映射到从 `0xa0000` 开始, 长度为 `320 * 200` 字节的物理内存区间. 这是PA3中的一个选做任务, 如果你之前没有实现的话, 现在你需要面对它了. 具体的, 你需要定义一些页表(注意页表需要按页对齐, 你可以参考 `kernel/src/memory/kvm.c` 中的相关内容), 然后填写相应的页目录项和页表项即可. 注意你不能使用 `mm_malloc()` 来实现video memory映射的创建, 因为 `mm_malloc()` 分配的物理页面都在16MB以上, 而video memory位于16MB以内, 故使用 `mm_malloc()` 不能达到我们的目的. 如果创建地址空间和内存映射I/O的实现都正确, 你会看到屏幕上输出了一些测试时写入的颜色信息, 同时 `video_mapping_read_test()` 将会通过检查.

到此为止, 随着设备这最后一块拼图的加入, NEMU的基本功能都已经实现好了, 最后我们通过往NEMU中移植两个游戏来测试实现的正确性.

移植打字小游戏

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

打字小游戏来源于2013年oslab0的框架代码, 为了配合移植, 代码的结构做了少量调整, 同时去掉了和显存优化相关的部分, 并对浮点数用binary scaling进行了处理.

我们对游戏的初始化部分进行一些说明:

- 程序入口是 `lib-common/uclibc/lib/crt1.o` 中的 `_start()` 函数.
- `_start()` 函数会调用 `lib-common/uclibc/lib/libc.a` 中的 `__uClibc_main()` 函数, 进行一系列运行时环境相关的初始化工作. 其中会调用 `ioctl()` 系统调用来检查stdin, stdout, stderr是否为字符设备. 由于除此之外, 在NEMU中运行的程序不会再调用 `ioctl()`, 为了简单起见, 我们只让内核实现了 `ioctl()` 中我们目前需要的功能, 而其它功能并未实现.
- 运行时环境初始化结束后, 就会跳转到游戏入口 `game/src/common/main.c` 中的 `main()` 函数.
- `init_timer()` 函数用于设置100Hz的时钟频率, 但由于NEMU中的时钟模拟实现是不可编程的, 而且模拟实现的时钟的默认频率就是100Hz, 故此处的 `init_timer()` 函数并没有实际作用.
- 调用 `init_FLOAT_vfprintf()` 劫持 `vfprintf()` 函数, 之后就可以输出FLOAT类型变量帮助调试.
- 在游戏中, `add_irq_handle()` 是一个人为添加的系统调用, 其系统调用号是0, 用于注册一个中断处理函数. 已经注册的中断处理函数会在相应中断到来的时候被内核调用, 这样游戏代码就可以通过中断来控制游戏的逻辑了. 但在真实的操作系统中, 提供这样的系统调用是非常危险的: 恶意程序可以注册一个陷入死循环的中断处理函数, 由于操作系统处理中断的时候, 处理器一般都处于关中断状态, 若此时陷入了死循环, 操作系统将彻底崩溃.
- 使用 `Log()` 宏输出一句话. 在游戏中, 通过 `Log()` 宏输出的信息都带有 `{game}` 的标签, 方便和kernel中的 `Log()` 宏输出区别开来.
- 进入游戏逻辑主循环. 整个游戏都在中断的驱动下运行.

在工程目录下运行 `make game` 命令编译游戏, 然后修改 `Makefile` 文件, 把 `USERPROG` 变量设置为 `$(game_BIN)`, 让编译得到的可执行文件 `game` 作为NEMU的用户程序来运行.

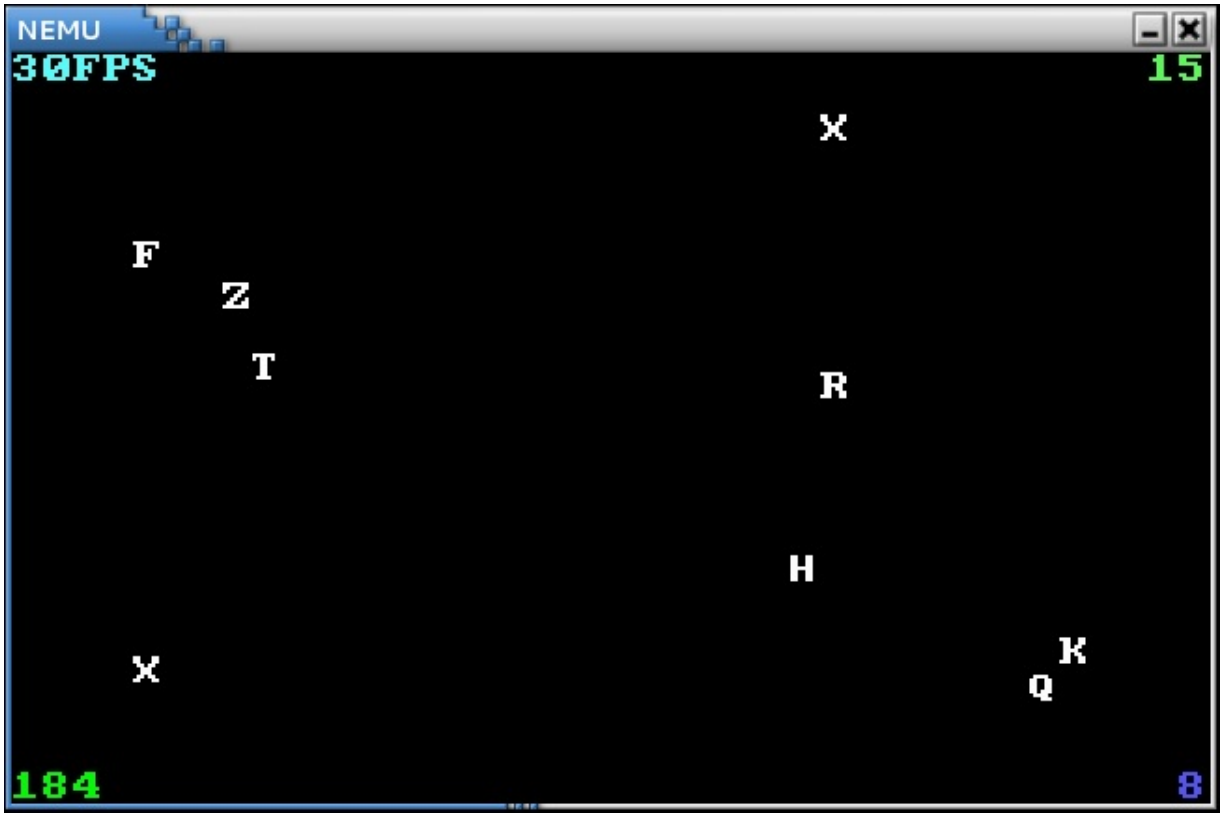
如果你之前的实现正确, 你将会看到打字游戏的画面, 但你会发现按键后出现system panic的信息, 这是因为kernel中没有对键盘中断进行响应, 认为键盘中断是一个非法的中断.

响应键盘中断

你需要找出引起system panic的原因, 然后根据你对中断响应过程的理解, 在kernel中添加相应的代码来响应键盘中断.

添加成功后, 按键后不再出现system panic的信息, 但游戏却没有对按键进行响应, 这是因为游戏并没有为键盘中断注册相应的中断处理函数, 你还需要在游戏初始化的时候为游戏注册键盘中断处理函数 `keyboard_event()` .

这些功能都实现之后, 你会得到一款完整的打字游戏.



必答题

游戏是如何工作的 在享受打字小游戏的乐趣的同时, 思考一下, 游戏究竟是如何工作的? 具体来说, 时钟中断到来/某个键被按下之后, 直到游戏逻辑更新(更新屏幕/寻找击中的字符)的这段时间, 计算机硬件(NEMU模拟出的CPU和设备)和软件(kernel和游戏代码)如何相互协助来支持游戏的运行? 这个问题涉及到硬件中断的处理过程, 你需要选择一种中断源(时钟/键盘), 并结合代码, 深入计算机层面来回答这个问题(能多详细就多详细).

让随机数更"随机"

游戏中使用了伪随机数生成函数 `rand()` , 但每次游戏重新开始运行之后, 生成函数产生的伪随机序列都是一样的. 你有办法让它们变得更"随机"吗?

奇怪的关中断(这个问题有难度)

框架代码给出的打字游戏中, game.c里有一段这样的代码:

```
while (true) {
    wait_intr();
    cli();
    if (now == tick) {
        sti();
        continue;
    }
    assert(now < tick);
    target = tick; /* now总是小于tick，因此我们需要“追赶”当前的时间 */
    sti();
    // ......
}
```

代码中有一处关中断操作, 但执行少量语句之后很快就重新打开中断了. 你能想明白这里的关中断有什么用意吗?

温馨提示

PA4阶段2到此结束.

通往高速的次元

如果打字小游戏已经在你的NEMU中跑起来了, 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, 打字小游戏会运行得比较慢, 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.

新版的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 obj/nemu/nemu obj/game/game
```

如果运行时发现类似如下错误:

```
/usr/bin/perf: line 24: exec: perf_3.16: not found
E: linux-tools-3.16 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?

移植仙剑奇侠传

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

把仙剑奇侠传移植到NEMU中的主要工作, 就是把应用层之下提供给仙剑奇侠传的所有API重新实现一遍, 因为这些API大多都依赖于操作系统提供的运行时环境, 我们需要根据NEMU和kernel提供的运行时环境重写它们. 主要包括以下四部分内容:

- C标准库
- 浮点数
- SDL库
- 文件系统

uclibc已经提供了C标准库的功能, 我们之前实现的 `FLOAT` 类型也已经解决了浮点数的问题, 因此我们可以很简单地对这两部分内容进行移植, 重点则落到了SDL库和文件系统的移植工作中.

关于浮点数有一点小小的补充. 在 `game/src/nemu-pal/battle/fight.c` 的代码中有一处调用了 `pow()` 函数, 用于根据角色的敏捷度(身法)计算在半即时战斗模式中角色的行动条变化量. 在SDLPAL中, 此处调用的 `pow()` 函数计算的是 `x^0.3` , 但 `FLOAT` 版本的通用 `pow()` 函数实现相对麻烦, 根据KISS法则, 我们把此处调用修改成 `x^(1/3)` , `lib-common/FLOAT.c` 中实现的 `pow()` 函数用来专门计算 `x^(1/3)` , 采用的是nth root algorithm.

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

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

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

重写SDL库的API

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

时钟相关

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

键盘相关

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

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

显示相关

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

- `SDL_Surface`
- `SDL_Rect`
- `SDL_BlitSurface`
- `SDL_FillRect`

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

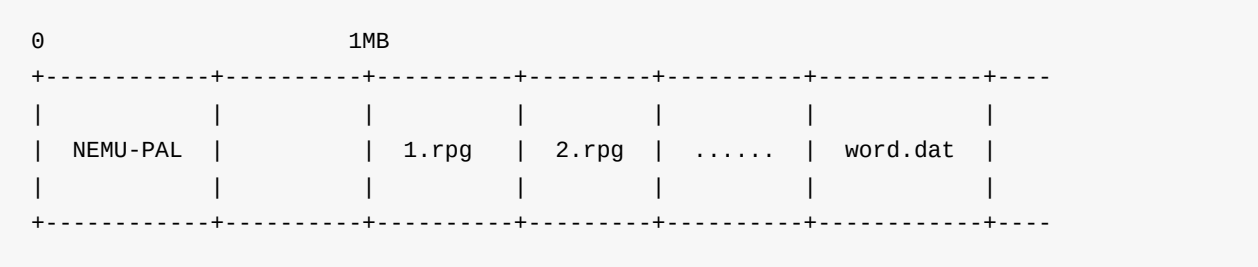
实现简易文件系统

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

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

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

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



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

```
typedef struct {
    bool opened;
    uint32_t offset;
} Fstate;
```

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

有了 `Fstate` 结构之后, 我们就可以实现以下的文件操作了:

```
int fs_open(const char *pathname, int flags);    /* 在我们的实现中可以忽略flags */
int fs_read(int fd, void *buf, int len);
int fs_write(int fd, void *buf, int len);
int fs_lseek(int fd, int offset, int whence);
int fs_close(int fd);
```

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

```
man 2 open
```

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

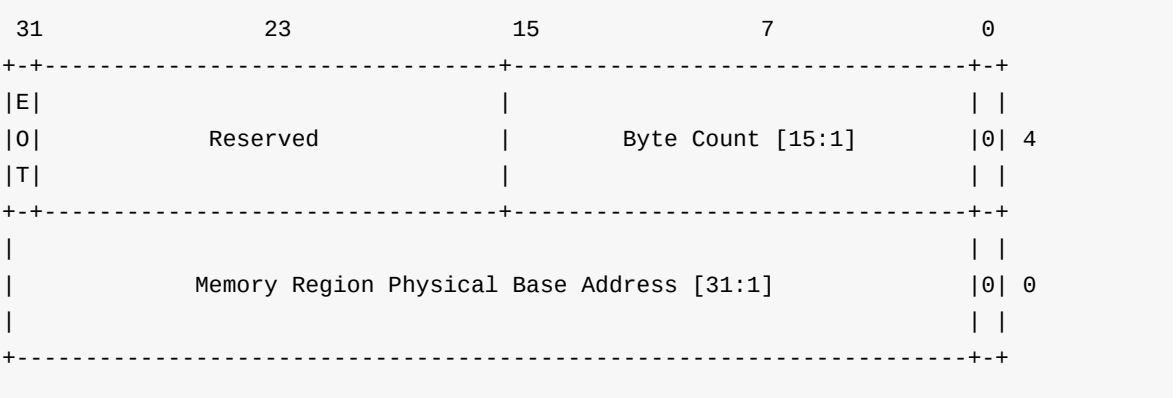
- 由于简易文件系统中每一个文件都是固定的, 不会产生新文件, 因此" `fs_open()` 没有找到 `pathname` 所指示的文件"属于异常情况, 你需要使用assertion终止程序运行.
- 使用 `ide_read()` 和 `ide_write()` 来进行文件的读写.
- 由于文件的大小是固定的, 在实现 `fs_read()` 和 `fs_lseek()` 的时候, 注意读写指针不要越过文件的边界.
- 除了写入 `stdout` 和 `stderr` 之外(即输出到串口), 其余对于 `stdin` , `stdout` 和 `stderr` 这三个特殊文件的操作可以直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

最后你还需要在kernel中编写相应的系统调用, 来调用相应的文件操作.

使用DMA方式读取磁盘内容(选做)

在kernel的框架代码中, IDE驱动程序让磁盘工作在PIO模式下. 在PIO模式下, CPU需要主动向磁盘读写数据. 现代的磁盘大多都支持DMA模式, DMA模式使得直接在磁盘和内存之间传输数据成为可能, 传输过程不需要CPU参与. 如果访问磁盘的频率很高, 使用DMA模式可以在一定程度上提高CPU工作的效率. NEMU-PAL常常需要从磁盘中读取数据文件, 我们可以尝试使用DMA模式来进行磁盘的读操作, 体会DMA的工作方式.

我们使用的DMA方式称为Bus Master DMA. 在介绍具体的编程步骤之前, 我们需要介绍PRD(Physical Region Descriptor)的概念. PRD的结构如下:



一个PRD用来描述一段按双字节对齐, 长度为偶数字节的物理内存区域, 其中 `Base Address` 是这段区域的首地址, `Byte Count` 是这段区域的长度. PRDT是一个数组, 数组的每一个元素是一个PRD, 特别地, 数组中的最后一个PRD的 `EOT` 位需要被置为 `1` , 表示"end of table". 在Bus Master DMA中, 一个PRD描述了一次DMA传输中内存位置的信息.

使用Bus Master DMA进行磁盘的读操作具体需要进行以下步骤:

- 在内存中准备PRDT.
- 将PRDT的首地址写入DMA控制器中的描述附表指针寄存器(Descriptor Table Pointer Register).
- 将待读扇区和长度等信息写入磁盘相应的寄存器.
- 向DMA控制器发送读命令.
- DMA控制器将根据传输请求将数据从磁盘传输在内存, 传送完成后, 磁盘会向CPU发送中断.

你需要做的事情是在 `kernel/src/driver/ide/dma.c` 中编写代码: 准备一个只有一项的PRDT, 填写这一表项的内容(缓冲区地址, 长度为512字节), 然后把PRDT的首地址装入相应的设备寄存器中. 需要注意的是, 这些地址都必须是物理地址, 你知道为什么不能是虚拟地址吗?

实现上述要求后, 在 `kernel/src/driver/ide/disk.c` 中定义宏 `USE_DMA_READ` , 重新编译并运行. 但你会发现IDE驱动并没有正常工作, 我们没有从IDE驱动中读出预期的数据. 这个问题和cache有关, 你能想明白为什么吗? 事实上在现代x86处理器的分页机制中, 页目录项和页表项有一位叫 `PCD` 的属性位, 当该属性位被置 `1` 时, 访问涉及到的物理页面时将不会经过cache. 一般来说, 操作系统会通过 `PCD` 位的功能把DMA用到的页面设置为不可缓存, 这样就不会出现上述问题了. 不过实现这一功能还需要对NEMU和kernel进行较多改动, 我们可以采取一种简单暴力的的方法: 把NEMU中的cache关掉, 让 `hwaddr_read()` 和 `hwaddr_write()` 直接访问dram.

由于我们的kernel是单任务的, 在发送DMA传输命令之后只能等待磁盘的中断, 无法切换到别的进程执行, 因此我们并不能完全体会到DMA的好处. 另外为了遵循KISS法则, 这一部分的代码省略了较多细节, 使得DMA部分的代码并不一定能在真机或完整的模拟器(如QEMU)中成功运行, 反之亦然. 即使这样, 这个练习仍然可以加深你对DMA的认识.

读写文件的具体过程

根据课堂上学习到的知识以及实验内容, 思考一下, 一个用户程序执行如下代码(其中 `fp` 是磁盘上某个文件的文件指针):

```
fprintf(fp, "Hello World!\n");
```

在这个过程中, 计算机是如何把字符串写入到磁盘的文件中的?

把移植仙剑奇侠传移植到NEMU

终于到了激动人心的时刻了! 根据上述讲义内容, 完成仙剑奇侠传的移植. 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, 来测试实现的正确性.

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.

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

到此为止, 我们已经把ISA层次中的指令系统, 存储管理, 中断/异常, I/O这四个方面全部都讨论完了; 仙剑奇侠传的移植工作也向你展示了ISA如何把软件硬件联系起来, 从而支持一个游戏的运行. 我们说"操作系统运行在ISA上", 其实是指操作系统的运行需要这四个方面的支持. 至于操作系统如何在利用ISA的同时给上层应用程序提供支撑和服务, 操作系统课程将会带你探索这个问题. ICS已经为操作系统之旅扫清了障碍, 准备好踏上新的旅程吧!

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一堆机械, 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?

问题的答案其实已经蕴涵在上图中了, 你能把"一堆沙子"(Physics, 材料)和"玩游戏"(Application, 应用)这两个看似毫不相关的概念联系起来吗? 如果不能, 你觉得还缺少些什么?

世界诞生的故事 - 终章

感谢你帮助上帝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, 你可以用上帝赋予你的创造力, 来改变这个美妙世界的轨迹, 书写故事新的篇章.

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.

```
#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:

  ```
  int a = 0x80000000;
  int b = a / -1;
  printf("%d\n", b);
  ```

- 代码段二:

  ```
  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(ics2016)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(如果你希望匿名吐槽, 请移步提交地址中的课程吐槽讨论区, 使用账号stu_ics登陆后进行吐槽)

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(http://cslab.nju.edu.cn/opsystem), 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.

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
```

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.

`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.

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(如果显示不正常,可以复制到文本编辑器中查看. 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.

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 th
e following addressing modes:
+  [*] means a disp32 with no base if MOD is 00. Otherwise, [*] means disp8[EBP] o
r disp32[EBP]. This provides the following addressing modes:
```

- 17.2.2.11 Instruction Set Detail中的DEC -- Decrement by 1

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

- 17.2.2.11 Instruction Set Detail中的INC -- Increment by 1

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

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

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

- 17.2.2.11 Instruction Set Detail中的MOV -- Move Data

```
@@ -?,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
 byte
19  /r       SBB r/m16,r16     2/6     Subtract with borrow word register from r/m
 word
19  /r       SBB r/m32,r32     2/6     Subtract with borrow dword register from r/
m dword
-1A  /r       SBB r8,r/m8       2/7     Subtract with borrow byte register from r/
m byte
-1B  /r       SBB r16,r/m16     2/7     Subtract with borrow word register from r/
m word
-1B  /r       SBB r32,r/m32     2/7     Subtract with borrow dword register from r
/m dword
+1A  /r       SBB r8,r/m8       2/7     Subtract with borrow r/m byte from byte re
gister
+1B  /r       SBB r16,r/m16     2/7     Subtract with borrow r/m word from word re
gister
+1B  /r       SBB r32,r/m32     2/7     Subtract with borrow r/m dword from dword
register
```

- 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
```

PA0提交情况

已经提交的压缩包

```
141130077    151220015    151220042    151220070    151220102    151220131    15122016
4
141130106    151220017    151220043    151220073    151220103    151220133    15122016
5
141150010    151220019    151220045    151220074    151220104    151220134    15122016
6
141170040    151220020    151220046    151220075    151220106    151220135    15122016
8
151110079    151220021    151220047    151220078    151220107    151220136    15122017
0
             151220022    151220049    151220081    151220110    151220137    15122017
1
151120033    151220023    151220050    151220082    151220111    151220138    15122017
2
151140042    151220024    151220052    151220085    151220113    151220139    15122017
4
151140083    151220025    151220053    151220086    151220114    151220140    15122017
5
151150010    151220026    151220054    151220087    151220116    151220142    15122017
6
151150051    151220027    151220055    151220088    151220117    151220146    15122017
7
151160018    151220028    151220056    151220090    151220119    151220149    15122017
9
151160030    151220030    151220057    151220091    151220120    151220150    15124200
2
151180115    151220031    151220059    151220092    151220121    151220151    15124201
1
151220003    151220033    151220060    151220093    151220122    151220152    15124201
2
151220004    151220034    151220061    151220094    151220123    151220153    15124203
5
151220005    151220035    151220062    151220095    151220124    151220155    15124206
0
151220007    151220036    151220063    151220096    151220125    151220156    15124206
2
151220008    151220037    151220064    151220097    151220126    151220157
151220009    151220038    151220065    151220098    151220127    151220158
151220010    151220039    151220066    151220099    151220128    151220160
151220013    151220040    151220068    151220100    151220129    151220161
151220014    151220041    151220069    151220101    151220130    151220162
```

如果你的学号没有出现在以上列表中, 可能是因为:

- 你还没有提交作业. 在课程结束之前, 我们仍然接受迟交的作业.
- 你没有使用规定的方式对工程进行打包.
- 你提交的压缩包命名不符合要求.

没有实验报告

```
151150010    151220040    151220064    151220092    151220114    151220153
151160018    151220041    151220073    151220101    151220138    151220162
151160030    151220046    151220078    151220102    151220142    151220176
151220036    151220059    151220082    151220103    151220152
```

如果你的学号出现在以上列表中, 可能是因为:

- 你没有提交实验报告.
- 你提交的实验报告命名或位置不符合要求.

没有编译记录

```
141130077
151220009
151220055
151220075
151220106
151220125
151220139
151220140
151220152
151220164
151220170
151242002
151242062
```

如果你的学号出现在以上列表中, 可能是因为:

- 你没有进行过编译.
- 编译时遇到index.lock的错误问题. 请确保 `git log` 能正确记录你的开发过程, 否则后续实验将视为没有提交.

报告中缺少实验进度的描述

有不少同学没有在实验报告中对进度进行简单的描述, 我们暂时不对缺少进度描述的报告进行惩罚. 若在后续PA中仍然缺少进度描述, 我们将按照提交说明中的描述进行处理: 该次实验将被视为没有完成.

PA1提交情况

已经提交的压缩包

141130077	151220020	151220048	151220078	151220110	151220139	151220166
141130106	151220021	151220049	151220079	151220111	151220140	151220168
141150010	151220022	151220050	151220081	151220112	151220141	151220169
151110079	151220023	151220052	151220082	151220113	151220142	151220170
151140042	151220024	151220053	151220083	151220114	151220143	151220171
151140083	151220025	151220054	151220085	151220116	151220144	151220172
151150010	151220026	151220055	151220086	151220117	151220145	151220173
151160018	151220028	151220056	151220087	151220119	151220146	151220174
151180107	151220030	151220057	151220088	151220120	151220147	151220175
151180115	151220031	151220059	151220091	151220122	151220149	151220176
151220003	151220033	151220060	151220092	151220123	151220150	151220177
151220004	151220034	151220061	151220094	151220125	151220151	151220179
151220005	151220035	151220062	151220095	151220126	151220152	151242002
151220007	151220036	151220063	151220096	151220127	151220153	151242011
151220008	151220037	151220064	151220097	151220128	151220154	151242012
151220009	151220038	151220065	151220099	151220129	151220155	151242035
151220010	151220039	151220066	151220100	151220130	151220156	151242041
151220012	151220040	151220068	151220101	151220131	151220157	151242060
151220013	151220041	151220069	151220102	151220133	151220158	151242062
151220014	151220042	151220072	151220103	151220134	151220160	
151220015	151220043	151220073	151220104	151220135	151220161	
151220016	151220045	151220074	151220106	151220136	151220162	
151220017	151220046	151220075	151220107	151220137	151220164	
151220019	151220047	151220076	151220109	151220138	151220165	

如果你的学号没有出现在以上列表中, 可能是因为:

- 你还没有提交作业. 在课程结束之前, 我们仍然接受迟交的作业.
- 你没有使用规定的方式对工程进行打包.
- 你提交的压缩包命名不符合要求.

以下同学提交的压缩包不符合要求, 将会被扣除PA1中10%的成绩:

```
151220070
151220077
151220093
151220159
```

没有工程

```
151220098
```

没有实验报告

```
151220033
151220075
151220098
151220109
151220147
151220152
151220173
151220176
```

如果你的学号出现在以上列表中, 可能是因为:

- 你没有提交实验报告.
- 你提交的实验报告命名或位置不符合要求.

按照约定, 以上同学的实验报告将被视为没有提交.

没有在**pa1**分支中进行开发

```
151160018
151220016
151220093
151220114
151220128
151220130
```

请以上同学在2016/10/14 23:59:59前联系yzh确认, 否则PA1将被视为没有提交.

其他异常

```
151160018  # 实验报告损坏
151220010  # 实验报告损坏
151220101  # ???
151220130  # ???
151220164  # 实验报告损坏
151242041  # git log损坏
```

请以上同学在2016/10/14 23:59:59前联系yzh确认异常情况, 否则...

课程感想收集说明(建议ICS期末考试结束后再写)

为了帮助我们改善课程, 我们需要大家提供一些反馈信息. 请大家以匿名的方式写一写课程感想.

感想内容

可以包括(并非强制要求)以下内容:

- 在学习ICS前后分别对"什么是计算机"的理解
- PA最后的完成情况, 每周在PA上花费的大概时间, 以及大概是怎么花费的
- 困难, 收获, 反思
  - 描述困难时请具体描述, 仅仅一句"看不懂"并不能帮助我们改善课程. 你可以具体描述看懂了哪些, 看不懂哪些, 对于看不懂的内容有什么想法等等, 这些详细的信息才能对我们有帮助.
- 对ICS的吐槽和建议
- 程序设计基础实验课对ICS的帮助, 或者说, 如果没有程序设计基础实验课, 你觉得你可能会在ICS课上多遇到哪些困难?
- PA课对完成PA的帮助
- 如果下学期的oslab也像PA那样具有一定的挑战性, 但最后也像PA那样可以做出一个完整的系统, 你愿意接受挑战吗?
- 对其它课程的吐槽和建议

你也可以写自己认为合适的其它内容.

感想要求

字数不限, 但有以下要求:

- 提交txt文本文件
- 内容眞实
- 感情眞挚
- 用语文明

违反上述要求的感想视为没有提交, 若发现抄袭现象, 抄袭双方(或团体)均视为没有提交. 若实在没有任何感想, 可以写"无感想".

提交方式

为了保证匿名特性(如果你愿意, 你也可以在感想中透露自己的个人信息), 感想通过ftp方式提交, 只接受 `txt` 文件, 文件名随机(例如 `2o4ursjer.txt`).

提交地址为ftp://172.25.46.144

- 账号 `ics_stu`
- 密码 `123456`
- 注意: 此网站不能在校外登陆

截止时间

感想提交截止时间: 2017/01/11 23:59:59

加分策略

为了鼓励大家认真写, 我们设定了一种加分策略. 设修读课程的总人数为 `N`, 其中认真写的人数为 `a`, 不那么认真写的人数为 `b`, 提交"无感想"的人数为 `c`, 没有提交感想的人数为 `d`, 则修读课程的每人都可以获得如下加分(加分算入PA部分):

```
( (a/N) ^ 2 + 0.3 * (b/N) ^ 1.5 + 0.05 * (c/N) ) * 4
```

例如, 修读课程的总人数为170, 其中认真写的人数为140, 不那么认真写的人数为15, 提交"无感想"的人数为10, 没有提交感想的人数为5, 则修读课程的每人都可以获得如下加分:

```
( (140/170) ^ 2 + 0.3 * (15/170) ^ 1.5 + 0.05 * (10/170) ) * 4 = 2.756
```

我们主要根据"是否能对我们提供有用的信息"来判断一份感想是否真实. 事实上, 只要你实事求是地具体描述, 对我们来说就是有用的信息. 加分不但取决于自己是否认真写, 还取决于其它同学是否认真写, 所以大家可以相互提醒和监督.