Table of Contents

南京大学 计算机科学与技术系 计算机系统基础 课程实验 2018

实验前阅读

最新消息(请每天至少关注一次)

- 2018/12/16
  - PA4和PA5已发布
- 2018/12/12
  - 由于i386手册的印刷错误较多, 一定程度上影响理解, 我们在github上开放了一个repo, 欢迎大家贡献修改建议.
- 2018/12/08
  - PA4.1和PA4.2已发布
- 2018/11/25
  - PA3已全部发布
- 2018/11/03
  - PA3.1和PA3.2已发布
- 2018/10/09
  - 我们修复了docker中运行native程序时 `mmap` 失败的问题, 详细信息请见这里. 如果你在此时间之前创建container, 请手动运行以下命令修复该bug:

```
docker stop ics-vm
docker commit ics-vm ics-image:fix-mmap
docker rm ics-vm
docker create --name=ics-vm -p 20022:22 --tmpfs /dev/shm:exec --privileged=true ic
s-image:fix-mmap
docker start ics-vm
```

- 2018/10/04
  - PA2已全部发布
- 2018/10/03
  - PA2.2已发布
- 2018/09/22
  - PA2.1已发布
- 2018/09/12
  - PA1已全部发布, 提交填写的task请见这里
- 2018/09/09

    - PA1.1已发布
    - 框架代码已发布
    - 我们在2018/09/09 10:28修复了如下bug. 如果你在此时间之前获得源代码, 请手动修复该bug; 如果你在此时间之后获得源代码, 你不需要进行额外的操作.

```
--- Makefile
+++ Makefile
@@ -12,3 +12,3 @@
submit: clean
git gc
-  STUID=$(STUID) STUNAME=$(STDNAME) bash -c "$$(curl -s http://moon.nju.edu.cn/people/yyjiang/teach/submit.sh)"
+  STUID=$(STUID) STUNAME=$(STUNAME) bash -c "$$(curl -s http://moon.nju.edu.cn/people/yyjiang/teach/submit.sh)"
```

- 2018/09/04
    - PA0已发布, 请大家按照讲义内容安装系统, 并注意截止时间
    - ~~框架代码未发布, 请保持关注~~
- 2018/03/12
    - ~~本讲义目前处于测试阶段, 在2018年秋季学期开始前, 将被视为往届讲义材料. 如果你是修读2018年秋季ICS课程的学生, 请勿使用本讲义代替2018年秋季ICS课程的PA实验讲义, 提交本实验内容将被视为没有提交.~~

如何求助

- 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)

小百合系版"有像我一样不会写代码的cser么?"回复节选

- 我们都是活生生的人, 从小就被不由自主地教导用最小的付出获得最大的得到, 经常会忘记我们究竟要的是什么. 我承认我完美主义, 但我想每个人心中都有那一份求知的渴望和对真理的向往, "大学"的灵魂也就在于超越世俗, 超越时代的纯真和理想 -- 我们不是要讨好企业的毕业生, 而是要寻找改变世界的力量. -- jyy
- 教育除了知识的记忆之外, 更本质的是能力的训练, 即所谓的training. 而但凡training就必须克服一定的难度, 否则你就是在做重复劳动, 能力也不会有改变. 如果遇到难度就选择退缩, 或者让别人来替你克服本该由你自己克服的难度, 等于是自动放弃了获得training的机会, 而这其实是大学专业教育最宝贵的部分. -- etone

- 这种"只要不影响我现在survive, 就不要紧"的想法其实非常的利己和短视: 你在专业上的技不如人, 迟早有一天会找上来, 会影响到你个人职业生涯的长远的发展; 更严重的是, 这些以得过且过的态度来对待自己专业的学生, 他们的survive其实是以透支南大教育的信誉为代价的 -- 如果我们一定比例的毕业生都是这种情况, 那么过不了多久, 不但那些混到毕业的学生也没那么容易survive了, 而且那些真正自己刻苦努力的学生, 他们的前途也会受到影响. -- etone

实验方案

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

- 简易调试器
- 冯诺依曼计算机系统
- 批处理系统
- 分时多任务
- 程序性能优化

实验环境

- CPU架构: x64
- 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	这里有google搜索镜像	http://en.wikipedia.org	http://stackoverflow.com
不推荐使用	~~http://www.baidu.com~~	~~http://baike.baidu.com~~	~~http://zhidao.baidu.com~~ ~~http://bbs.csdn.net~~

一些说明:

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

官方手册

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

这里列出一些本课程中可能会用到的手册:

- Intel 80386 Programmer's Reference Manual(PDF)(HTML)(人手一本的i386手册)
- System V ABI for i386
- C99 Standard
- GCC 6.3.0 Manual
- GDB User Manual
- GNU Make Manual
- 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.

许可协议

PA0 - 世界诞生的前夜: 开发环境配置

世界诞生的故事 - 序章

PA讲述的是一个"先驱创造计算机"的故事.

先驱打算创造一个计算机世界. 但巧妇难为无米之炊, 为了更方便地创造这个世界, 就算是先驱也是花了一番功夫来准备的. 让我们来看看他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: 2018/09/09 23:59:59

提交说明: 见这里

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!

信息框说明

讲义中会出现一些信息框, 根据其颜色和左上角的图标可以得知信息的类别. 例如本信息框就是一些提示相关的内容. 其它类别主要还有

实验进度相关的提示

扩展阅读

选做思考题

实验必做内容

实验进度相关的必读信息

重要性超越实验的原则与方法

对, 你没有看错, 除了一些重要的信息之外, **PA0**的实验讲义都是英文!

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

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

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

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

你需要独立完成PA0, 请你认真阅读讲义中的每一个字符, 并按照讲义中的内容进行操作: 当讲义提到要在互联网上搜索某个内容时, 你就去互联网上搜索这个内容. 如果遇到了错误, 请认真反复阅读讲义内容, 机器永远是对的. 如果你是第一次使用GNU/Linux, 你还需要查阅大量资料或教程来学习一些新工具的使用方法, 这需要花费大量的时间(例如你可能需要花费一个下午的时间, 仅仅是为了使用`vim`在文件中键入两行内容). 这就像阅读英文材料一样, 一开始你会觉得效率很低, 但随着时间的推移, 你对这些工具的使用会越来越熟练. 相反, 如果你通过"投机取巧"的方式来完成PA0, 你将会马上在PA1中遇到麻烦. 正如etone所说, 你在专业上的技不如人, 迟早有一天会找上来.

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

Installing Docker

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

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

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

必须使用64位的GNU/Linux

> 如果你打算使用已有的GNU/Linux平台, 请确保它是64位版本. 今年PA的新增特性会依赖于64位平台.

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

Preparing Dockerfile

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

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

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

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

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

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

Now copy the following contents into Dockerfile:

```
 # setting base image
FROM debian

RUN apt-get update

 # Set the locale
RUN apt-get install -y locales
RUN sed -i -e 's/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/' /etc/locale.gen
RUN dpkg-reconfigure --frontend=noninteractive locales
RUN update-locale LANG=en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8

 # new a directory for sshd to run
RUN mkdir -p /var/run/sshd
 # installing ssh server
RUN apt-get install -y openssh-server

 # installing sudo
RUN apt-get install -y sudo

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

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

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

 # adding user to sudo group
RUN adduser $username sudo

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

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

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

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

Building Docker image

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

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

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

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

```
docker images
```

This command will show information about all Docker images.

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

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

Now we can remove the directory mentioned above.

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

Creating Debian container

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

```
docker create --name=ics-vm -p 20022:22 --tmpfs /dev/shm:exec --privileged=true ics-im
age
```

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
- the container will get extended privileges (for GDB to run)
- mount `/dev/shm` with an executable flag

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

```
docker rm ics-vm
```

Then create the container again.

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

```
docker ps -a
```

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

First Exploration with GNU/Linux

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

```
docker start ics-vm
```

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

SSH configuration

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

For GNU/Linux and Mac users

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

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

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

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

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

For Windows users

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

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

First exploration

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

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

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

Where is GUI?

> Many of you always use operating system with GUI, such as Windows. The container you just created is without GUI. It is completely with CLI (Command Line Interface). As you entering the container, you may feel empty, depress, and then panic...
>
> Calm down yourself. Have you wondered if there is something that you can do it in CLI, but can not in GUI? Have no idea? If you are asked to count how many lines of code you have coded during the 程序设计基础 course, what will you do?
>
> If you stick to Visual Studio, you will never understand why `vim` is called 编辑器之神. If you stick to Windows, you will never know what is Unix Philosophy. If you stick to GUI, you can only do what it can; but in CLI, it can do what you want. One of the most important spirits of young people like you is to try new things to bade farewell to the past.
>
> GUI wins when you do something requires high definition displaying, such as watching movies. But in our experiments, GUI is unnecessary. Here are two articles discussing the comparision between GUI and CLI:
>
> - Why Use a Command Line Instead of Windows?
> - Command Line vs. GUI

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

```
df -h
```

You can see that Debian is quite "slim".

Why Windows is quite "fat"?

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

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

```
docker stop ics-vm
```

And type `exit` to exit the host terminal.

Installing Tools

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

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

Checking network state

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

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

You should receive reply packets successfully:

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

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

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

Updating APT package information

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

```
apt-get update
```

However, you will receive an error message:

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

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

Why do some operations require superuser privilege?

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

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

```
sudo apt-get update
```

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

Installing tools for PAs

The following tools are necessary for PAs:

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

The usage of these tools is explained later.

Configuring vim

```
apt-get install vim
```

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

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

Under these situations, `vim` is still a good choice. Another reason to choose `vim` is that, `vim` can greatly improve your coding efficiency. 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.

为什么上课不讲GNU/Linux的使用?

你可能会想: 这是我第一次接触GNU/Linux, 为什么上课不讲讲怎么用?

因为说明书不是用来讲的, 是用来一边看一边操作的; 你对这些工具也不是靠听来掌握的, 而是自己动手去尝试. 你在大学课堂上应该接受到的是那些一脉相承的知识, 然后去思考这些知识背后的原则和思想, 将来有能力将这些原则和思想应用到新的领域.

我们设计这些实验内容, 是为了让你明白, 自己有能力去看教程学习新的工具; 以及, 以后接触新事物的时候, 你不应该等着别人来给你讲, 而应该自己主动去找教程来学习如何使用.

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:

```
aaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeffffffffffffffffffffffff
ggggggggggggggggggggggggghhhhhhhhhhhhhhhhhhhhhhhh
iiiiiiiiiiiiiiiiiiiiiiiijjjjjjjjjjjjjjjjjjjjjjjj
```

You want to modify it into:

```
bbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaa
ddddddddddddddddddddddddcccccccccccccccccccccccc
ffffffffffffffffffffffffeeeeeeeeeeeeeeeeeeeeeeee
hhhhhhhhhhhhhhhhhhhhhhhhggggggggggggggggggggggggg
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. If you do not understand the diff format, please search the Internet for more information.

为什么要STFW?

你或许会想, 我问别人是为了节省我的时间.

但现在是互联网时代了, 在网上你能得到各种信息: 比如diff格式这种标准信息, 网上是100%能搜到的; 就包括你遇到的问题, 很大概率也是别人在网上求助过的. 如果对于一个你本来只需要在搜索引擎上输入几个关键字就能找到解决方案的问题, 你都没有付出如此微小的努力, 而是首先想着找人来帮你解决, 占用别人宝贵的时间, 你将是这个时代的"失败者".

于是有了STFW (Search The F***ing Web) 的说法, 它的意思是, 在向别人求助之前自己先尝试通过正确的方式使用搜索引擎独立寻找解决方案.

正确的STFW方式能够增加找到解决方案的概率, 包括

- 使用Google搜索引擎搜索一般性问题
- 使用英文维基百科查阅概念
- 使用stack overflow问答网站搜索程序设计相关问题

如果你没有使用上述方式来STFW, 请不要抱怨找不到解决方案而开始向别人求助, 你应该想, "噢我刚才用的是百度, 接下来我应该试试Google". 关于使用Google, 在学校可以尝试设置IPv6, 或者设置"科学上网", 具体设置方式请STFW.

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.

为什么要这么麻烦?

搞了半天, 你发现其实也就是改动一个字符而已, 干嘛不直接说清楚呢?

这是为了"入乡随俗": 我们希望你了解怎么用计算机思维精简准确地表达我们想做的事情. diff格式是一种描述文件改动的常用方式. 实际上, 计算机的世界里面有很多约定俗成的"规矩", 当你慢慢去接触去了解这些规矩的时候, 你就会在不知不觉中明白计算机世界是怎么运转的.

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
```

提高开发效率的编辑器

程序设计课上你学会了使用Visual Studio, 然后你可能会认为, 程序员就是这样写代码的了. 其实并不是, 程序员会追求那些提高效率的方法. 不是GUI不好, 而是你只是用记事本的操作方式来写代码. 所以你需要改变, 去尝试一些可以帮助你提高开发效率的工具.

在GNU/Linux中, 与记事本的操作方式相比, 学会 `vim` 的基本操作就已经可以大大提高开发效率. 还有各种插件来增强 `vim` 的功能, 比如可以在代码中变量跳转的 `ctags` 等等. 你可以花点时间去配置一下 `vim` , 具体配置方式请STFW. 总之, "编辑器之神"可不是浪得虚名的.

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.

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.

为什么要RTFM?

RTFM是STFW的长辈, 在互联网还不是很流行的年代, RTFM是解决问题的一种有效方法. 这是因为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手册中找到答案.

你或许会觉得翻阅手册太麻烦了, 所以可能会在百度上随便搜一篇博客来尝试寻找解决方案. 但是, 你需要明确以下几点:

- 你搜到的博客可能也是转载别人的, 有可能有坑
- 博主只是分享了他的经历, 有些说法也不一定准确
- 搜到了相关内容, 也不一定会有全面的描述

最重要的是, 当你尝试了上述方法而又无法解决问题的时候, 你需要明确"我刚才只是在尝试走捷径, 看来我需要试试RTFM了".

Write a "Hello World" program under GNU/Linux

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

Write a Makefile to compile the "Hello World" program

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

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

Learn to use GDB

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

嘿! 别偷懒啊!

> 上文让你写个"Hello World"程序, 然后写个Makefile来编译它, 并且看教程学习一下GDB的基本使用呢!

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:

```
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}"
```

These 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 STFW.

又要没完没了地**STFW**了?

对.

PA除了让大家巩固ICS理论课的知识之外, 还承担着一个重要的任务: 把大家培养成一个素质合格的CSer. 事实上, 一个素质合格的CSer, 需要具备独立搜索解决方案的能力. 这是IT企业和科研机构对程序员的一个基本要求: 你将来的老板很可能会把一个任务直接丢给你, 如果你一遇到困难就找人帮忙, 老板就会认为你没法创造价值.

PA在尝试让你重视这些业界和学术界都看重的基本要求, 从而让你锻炼这些能力和心态: 遇到问题了, 第一反应不是赶紧找个大神帮忙搞定, 而是"我来试试STFW和RTFM, 看能不能自己解决". 所以PA不是按部就班的中学实验, 不要抱怨讲义没写清楚导致你走了弯路, 我们就是故意的: 我们会尽量控制路不会太弯, 只要你摆正心态, 你是有能力去独立解决这些问题的. 重要的是, 你得接受现实: 你走的弯路, 都在说明你的能力有待提升, 以后少走弯路的唯一方法, 就是你现在认真把路走下去.

提问的智慧

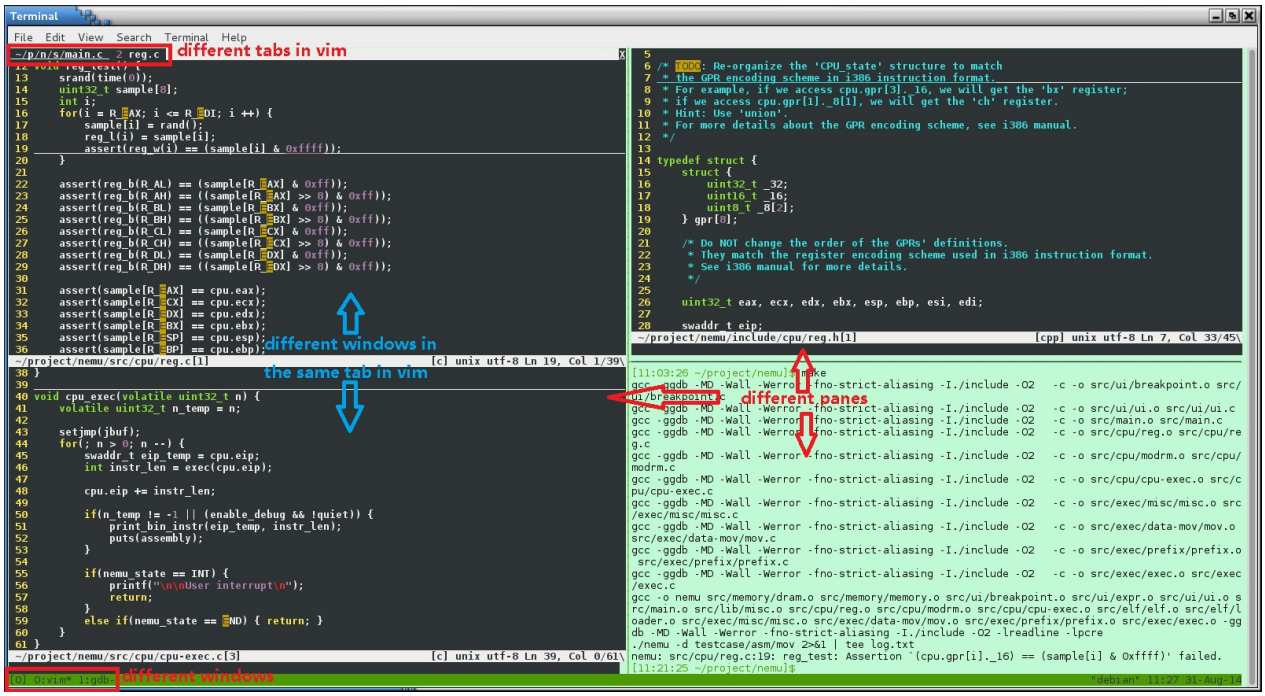
一个素质合格的CSer需要具备的另一个标准是, 懂得如何提问.

相信大家作为CSer, 被问如何修电脑的事情应该不会少. 比如你有一个文科小伙伴, 他QQ跟你说一句"我的电脑出问题了", 让你帮他修. 然后你得问东问西才了解具体的问题, 接着你让他尝试各种方案, 让他给你尝试的反馈. 如果你有10个这样的小伙伴, 相信你肯定受不了了. 这下你多少能体会到助教的心情了吧.

事实上, 如果希望能提高得到回答的概率, 提问者应该学会如何更好地提问. 文科小伙伴确实不是学习计算机专业, 你可以选择原谅他; 但你是CSer, 至少你得在问题中描述具体的现象以及你做过的尝试, 而不是直接丢一句"我的程序挂了", 就等着别人来救场. 如果你不知

道如何更好地提问, 请务必阅读提问的智慧这篇文章.

The following picture shows a scene working with multiple terminals within single screen. Is it COOL?



为什么要使用tmux?

这其实是一个"使用正确的工具做事情"的例子.

计算机天生就是为用户服务的, 只要你有任何需求, 你都可以想, "有没有工具能帮我实现?". 我们希望每个终端做不同的事情, 能够在屏幕上一览无余的同时, 还能在终端之间快速切换. 事实上, 通过STFW和RTFM你就可以掌握如何使用一款正确的工具: 你只要在搜索引擎上搜索"Linux 终端 分屏", 就可以搜到 `tmux` 这个工具; 然后再搜索"tmux 使用教程", 就可以学习到 `tmux` 的基本使用方法; 在终端中输入 `man tmux` , 就可以查阅关于 `tmux` 的任何疑问.

当然, 学习不是零成本的. 往届有学长提出一种零学习成本的分屏方式: 打开4个putty窗口分别登陆ssh, 然后4个窗口分别拖动到屏幕的4个角落, 发现用Alt+Tab快捷键不好选择窗口(因为4个窗口的外貌都差不多), 就使用鼠标点击的方式来切换. 然后形容安装学习 `tmux` 是"脱裤子放屁 -- 多此一举".

`tmux` 的初衷就是为用户节省上述的操作成本. 如果你抱着不愿意付出任何学习成本的心态, 就无法享受到工具带来的便利.

Things behind scrolling

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

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

Transferring Files Between host and container

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

For GNU/Linux and Mac users

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

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

where

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

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

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

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

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

where

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

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

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

For Windows users

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

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

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

Have a try!

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

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

Acquiring Source Code for PAs

Getting Source Code

Go back to the home directory by

```
cd ~
```

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

不要使用root账户做实验!!!

使用root账户进行实验, 会改变实验相关文件的权限属性, 可能会导致开发跟踪系统无法正常工作; 更严重的, 你的误操作可能会无意中损坏系统文件, 导致虚拟机/容器无法启动! 往届有若干学长因此而影响了实验进度, 甚至由于损坏了实验相关的文件而影响了分数. 请大家引以为鉴, 不要贪图方便, 否则后果自负!

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

Now acquire source code for PA by the following command:

```
git clone -b 2018 https://github.com/NJU-ProjectN/ics-pa.git ics2018
```

A directory called `ics2018` will be created. This is the project directory for PAs. Details will be explained in PA1.

Issue the following commands to perform `git` configuration:

```
git config --global user.name "171220000-Zhang San" # your student ID and name
git config --global user.email "zhangsan@foo.com"   # your email
git config --global core.editor vim                 # your favorite editor
git config --global color.ui true
```

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

Enter the project directory `ics2018`, then run

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

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

Git usage

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

You can list all branches by

```
git branch
```

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

```
* master
```

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

```
git checkout -b pa0
```

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

```
  master
* pa0
```

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

Now have a try! Modify the `STUID` and `STUNAME` variables in `nemu/Makefile.git`:

```
STUID = 171220000  # your student ID
STUNAME = 张三     # your Chinese name
```

Run

```
git status
```

to see those files modified from the last commit:

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

    modified:   nemu/Makefile.git

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

Run

```
git diff
```

to list modifications from the last commit:

```
diff --git a/nemu/Makefile.git b/nemu/Makefile.git
index c9b1708..b7b2e02 100644
--- a/nemu/Makefile.git
+++ b/nemu/Makefile.git
@@ -1,4 +1,4 @@
-STUID = 171220000
-STUNAME = 张三
+STUID = 171221234
+STUNAME = 李四

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

You should see `STUID` and `STUNAME` are 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 info` in the first line, and keep the remaining contents unchanged. Save and exit the editor, and this finishes a commit. Now you should see a log labeled with your student ID and name by

```
git log
```

Now switch back to the `master` branch by

```
git checkout master
```

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

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

```
git checkout pa0
```

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

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

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

Compiling and Running NEMU

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

```
make
```

If nothing goes wrong, NEMU will be compiled successfully.

未找到**xxx**命令

你有可能会遇到这个错误信息, 好吧确实是讲义疏忽了. 那就正好当作一个练习吧: 你需要把缺少的工具装上.

至于怎么装, 当然是STFW了.

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-ics2018 <tracer@njuics.org>
Date:   Sun Jul 26 14:30:31 2018 +0800

    compile
    171220000
    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/ics2018/.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 about 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.

```
ics2018
├── 171220000.pdf   # put your report file here
├── init.sh
├── Makefile
├── nanos-lite
├── navy-apps
├── nemu
├── nexus-am
└── README.md
```

Double check whether everything is fine. Then go back to the project directory, issue

```
make submit
```

This command does 3 things:

1. Cleanup unnecessary files for submission
2. Cleanup unnecessary files in git
3. Download a script from the server and run it to submit everything to the specific website

The script will ask you to enter the task name. Enter `PA0` for this submission. If everything is fine, the script will output a `SUCCESS` message.

> 又报错了
>
> 我知道, 那你说该怎么办呢?

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 STFW and RTFM for using new tools and trouble-shooting. (反思一下, 你真的做到了吗?) 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:

- STFW
- RTFM
- RTFSC

Reminder

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

PA1 - 开天辟地的篇章: 最简单的计算机

世界诞生的故事 - 第一章

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

代码管理

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:

- task PA1.1: 实现单步执行, 打印寄存器状态, 扫描内存 - 2018/09/16 23:59:59
- task PA1.2: 实现算术表达式求值 - 2018/09/23 23:59:59
- task PA1.3: 实现所有要求, 提交完整的实验报告 - 2018/09/30 23:59:59

提交说明: 见这里

在开始愉快的PA之旅之前

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

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

配置X Server

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

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

然后根据主机操作系统的类型, 为SSH打开X11转发功能:

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

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

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

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

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

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

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

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

这三种情况进行比较.

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

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

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

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

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

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

NEMU是什么?

上述描述对你来说也许还有些晦涩难懂, 让我们来看一个ATM机的例子.

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

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

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

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.

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

让我们来开始这段激动人心的旅程吧!

做**PA**的正确方式

PA除了给大家展示"程序如何在计算机中执行"这一终极目标之外, 还加入了很多科学的做事原则. PA在尝试制造场景让大家体会这些原则的重要性, 这也是作为一个素质合格的CSer的必修课. 如果你只是仅仅把PA作为一个编程大任务, 我们相信你确实吃了亏.

PA像一个值得打二周目的游戏, 在二周目的过程中, 你会对这些原则有更深刻的理解. 同时讲义中也准备了一些适合在二周目思考的问题, 希望大家玩得开心!

随时记录实验心得

我们已经在你学长学姐的实验报告中多次看到类似的悔恨: 因为没有及时记录实验心得而在编写实验报告的时候忘记了自己经历趣事的细节. 为了和助教们分享你的各种实验经历, 我们建议你在实验过程中随时记录实验心得, 比如自己踩过的大坑, 或者是调了一周之后才发现的一个弱智bug, 等等.

我们相信, 当你做完PA回过头来阅读这些心得的时候, 就会发现这对你来说是一笔宝贵的财富.

开天辟地的篇章

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

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

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

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

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

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

其中

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

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

计算机可以没有寄存器吗? (建议二周目思考)

> 如果没有寄存器, 计算机还可以工作吗? 如果可以, 这会对硬件提供的编程模型有什么影响呢?
>
> 就算你是二周目来思考这个问题, 你也有可能是第一次听到"编程模型"这个概念. 不过如果一周目的时候你已经仔细地阅读过i386手册, 你会记得确实有这么个概念的. 所以, 如果想知道什么是编程模型, RTFM吧.

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

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

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

从此以后, 计算机就只需要做一件事情:

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

这样, 我们就有了一个足够简单的计算机了. 我们只要将一段指令序列放置在存储器中, 然后让PC指向第一条指令, 计算机就会自动执行这一段指令序列, 永不停止.

这个全自动的执行过程实在是太美妙了! 事实上, 开拓者图灵在1936年就已经提出类似的核心思想, "计算机之父"可谓名不虚传. 而这个流传至今的核心思想, 就是"存储程序". 为了表达对图灵的敬仰, 我们也把上面这个最简单的计算机称为"图灵机"(Turing Machine, TRM). 或许你已经听说过"图灵机"这个作为计算模型时的概念, 不过在这里我们只强调作为一个最简单的真实计算机需要满足哪些条件:

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

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

计算机的状态模型

我们知道, 时序逻辑电路里面有"状态"的概念. 那么, 对于TRM来说, 是不是也有这样的概念呢? 具体地, 什么东西表征了TRM的状态? 在状态模型中, 执行指令和执行程序, 其本质分别是什么?

RTFSC

既然TRM那么简单, 就让我们在NEMU里面实现一个TRM吧.

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

```
ics2018
├── init.sh        # 初始化脚本
├── Makefile       # 用于工程打包提交
├── nanos-lite     # 微型操作系统内核
├── navy-apps      # 应用程序集
├── nemu           # NEMU
├── nexus-am       # 抽象计算机
└── README.md
```

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

代码中 `nemu/` 目录下的源文件组织如下(并未列出所有文件):

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

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

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

需要多费口舌吗?

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

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

对vim的使用感到困难?

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

其实最主要的还是, 你需要有尝试新事物的精神.

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

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

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

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

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

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

```
(nemu)
```

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). 实现正确后, NEMU将不会再触发assertion fail, 而是输出上文提到的命令提示符.

关于i386寄存器的更多细节, 请查阅i386手册. Hint: 使用匿名union.

什么是匿名union?

你有这个疑问是很正常的, 但你接下来应该意识到要去STFW了.

reg_test()是如何测试你的实现的?

阅读 `reg_test()` 的代码, 思考代码中的 `assert()` 条件是根据什么写出来的.

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

究竟要执行多久?

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

潜在的威胁 (建议二周目思考)

> "调用 `cpu_exec()` 的时候传入了参数 `-1` ", 这一做法属于未定义行为吗? 请查阅C99手册确认你的想法.

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

温故而知新

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

NEMU将不断执行指令, 直到遇到以下情况之一, 才会退出指令执行的循环:

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

当你看到NEMU输出以下内容时:

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

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

谁来指示程序的结束?

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

有始有终 (建议二周目思考)

对于GNU/Linux上的一个程序, 怎么样才算开始? 怎么样才算是结束? 对于在NEMU中运行的程序, 问题的答案又是什么呢?

与此相关的问题还有: NEMU中为什么要有 `nemu_trap` ? 为什么要有monitor?

最后我们聊聊代码中一些值得注意的地方.

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

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

理解框架代码

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

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

事实上, TRM的实现已经都蕴含在上述的介绍中了.

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

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

基础设施: 简易调试器

基础设施 - 提高项目开发的效率

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

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

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

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

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

真实故事

yzh的小组里曾经发生过一件由于基础设施不完善而导致在论文投稿截止前科研成果质量不佳的事件.

> 这一科研工作需要运行各种不同的测试来验证效果, 在两台4核8线程的PC机上运行完所有的测试需要花费约24小时. 每次对设计进行改动之后, 都需要重新运行所有测试, 可以说是, 改一行代码, 要过24小时后才能得到结果. 事实上, 在投稿截止3个月之前, 我们已经得知有一台112核的服务器可以使用. 如果在这台服务器上部署测试环境, 预计可以使测试总时间减少到原来的1/5 (多核服务器的主频比PC机低1倍, 架构也落后两代, 1/5是一个综合考虑之后的数字).
>
> 但是带领这一项目的同学并没有意识到基础设施的重要性: 他一直在PC机上进行测试. 事实上, 测试总时间减少到原来的1/5, 其实意味着改进设计的机会是原来的5倍. 结果到投稿截止之前, 设计还在修改, 测试还在反复运行, 最终只能无奈地采用一个质量有待提高的设计版本提交论文.

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

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

命令	格式	使用举例	说明
帮助(1)	`help`	`help`	打印命令的帮助信息
继续运行(1)	`c`	`c`	继续运行被暂停的程序
退出(1)	`q`	`q`	退出NEMU
单步执行	`si [N]`	`si 10`	让程序单步执行 `N` 条指令后暂停执行, 当 `N` 没有给出时, 缺省为 `1`
打印程序状态	`info SUBCMD`	`info r` `info w`	打印寄存器状态 打印监视点信息
表达式求值	`p EXPR`	`p $eax + 1`	求出表达式 `EXPR` 的值, `EXPR` 支持的运算请见调试中的表达式求值小节
扫描内存(2)	`x N EXPR`	`x 10 $esp`	求出表达式 `EXPR` 的值, 将结果作为起始内存 地址, 以十六进制形式输出连续的 `N` 个4字节
设置监视点	`w EXPR`	`w *0x2000`	当表达式 `EXPR` 的值发生变化时, 暂停程序执行
删除监视点	`d N`	`d 2`	删除序号为 `N` 的监视点

备注:

- (1) 命令已实现

- (2) 与GDB相比, 我们在这里做了简化, 更改了命令的格式

总有一天会找上门来的**bug**

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

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

解析命令

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

```
man readline
```

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

```
man strtok
```

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

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

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

如何测试字符串处理函数?

> 你可能会抑制不住编码的冲动: 与其RTFM, 还不如自己写. 如果真是这样, 你可以考虑一下, 你会如何测试自己编写的字符串处理函数?
>
> 如果你愿意RTFM, 也不妨思考一下这个问题, 因为你会在PA2中遇到类似的问题.

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STFW.

单步执行

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

打印寄存器

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

扫描内存

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

```
x 10 0x100000
```

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

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

实现单步执行, 打印寄存器, 扫描内存

> 熟悉了NEMU的框架之后, 这些功能实现起来都很简单, 同时我们对输出的格式不作硬性规定, 就当做是熟悉GNU/Linux编程的一次练习吧.
>
> 不知道如何下手? 嗯, 看来你需要再阅读一遍RTFSC小节的内容了.

我怕代码写错了啊, 怎么办?

2014年图灵奖得主Michael Stonebraker在一次访谈中提到, 他当时花了5年时间开发了世界上第一个关系数据库系统Ingres, 其中90%的时间用于将它运行起来. 也就是说, 在开发过程中, 有90%的时间系统都是运行不起来的, 是有bug的, 需要调试.

所以, 接受现实吧: 代码出错是很正常的, 你需要从当年程序设计实验里感受到的那种"代码可以一次编译通过成功运行"的幻觉中清醒过来. 重要的是, 我们需要使用正确的方法和工具来帮助我们测试和调试, 最终让程序运行起来. 一个例子是版本控制工具`git`, 它可以跟踪代码的变化, 从而发现bug是何时引入的, 而且能够在必要的时候回退到上一个程序可以运行的版本.

总之, 只有掌握正确的方法和工具, 才能真正驱散心中对bug的恐惧.

好像有点不对劲

细心的你会发现, 和默认镜像进行对比的时候, 扫描内存的结果貌似有点不太一样. 你知道这是为什么吗?

温馨提示

PA1阶段1到此结束.

表达式求值

在TRM中, 寄存器和内存中的值唯一地确定了计算机的一个状态. 因此从道理上来说, 打印寄存器和扫描内存这两个功能一定可以帮助我们调试出所有的问题. 但为了方便使用, 我们还希望简易调试器能帮我们计算一些带有寄存器和内存的表达式. 所以你需要在简易调试器中添加表达式求值的功能. 为了简单起见, 我们先来考虑数学表达式的求值实现.

数学表达式求值

给你一个表达式的字符串

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

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

1. 首先识别出表达式中的单元
2. 根据表达式的归纳定义进行递归求值

词法分析

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

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

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

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

- 十进制整数

- `+`, `-`, `*`, `/`
- `(`, `)`
- 空格串(一个或多个空格)

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

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

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

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

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

如果尝试了所有的规则都无法在当前位置识别出token, 识别将会失败, 框架代码会输出当前token的位置(当表达式过长导致在终端里输出需要换行时, `^`可能无法指示正确的位置, 此时建议通过输出的`position`值来定位token的位置). 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.

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.

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.

这两条公理的意思是: 抱怨是没有用的, 接受代码有bug的现实, 耐心调试.

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.

如何调试

- 不要使用"目光调试法", 要思考如何用正确的工具和方法帮助调试
  - 程序设计课上盯着几十行的程序, 你或许还能在大脑中模拟程序的执行过程; 但程序规模大了之后, 很快你就会放弃的
  - 我们学习计算机是为了学习计算机的工作原理, 而不是学习如何像计算机那样机械地工作
- 使用 `assert()` 设置检查点, 拦截非预期情况
  - 例如 `assert(p != NULL)` 就可以拦截由空指针解引用引起的段错误
- 结合对程序执行路径的理解, 使用 `printf()` 查看程序执行的情况(注意字符串要换行)
  - `printf()` 输出任意信息可以检查代码可达性: 输出了相应信息, 当且仅当相应的代码块被执行
  - `printf()` 输出变量的值, 可以检查其变化过程与原因
- 使用GDB查看程序的任意状态
  - 打印变量, 断点, 监视点, 函数调用链...
  - 不会用GDB? 快去STFW找教程啊! 事实上, 我们已经在PA0中提供了一份简单教程了

如果你突然觉得上述方法很有道理, 说明你在程序设计课上没有受到该有的训练.

系统设计的黄金法则 -- KISS法则

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

你已经学习过程序设计基础, 这意味着你已经学会写程序了, 但这并不意味着你可以顺利地完成PA, 因为在现实世界中, 我们需要的是可以运行的system, 而不是求阶乘的小程序. NEMU作为一个麻雀虽小, 五脏俱全的小型系统, 其代码量达到3000多行(不包括空行). 随着PA的进行, 代码量会越来越多, 各个模块之间的交互也越来越复杂, 工程的维护变得越来越困难, 一个很弱智的bug可能需要调好几天. 在这种情况下, 系统能跑起来才是王道, 跑不起来什么都是浮云, 追求面面俱到只会增加代码维护的难度.

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

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

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

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 matched
```

至于怎么检查左右括号是否匹配, 就留给聪明的你来思考吧!

上面的框架已经考虑了BNF中算术表达式的开头两种定义, 接下来我们来考虑剩下的情况(即上述伪代码中最后一个 `else` 中的内容). 一个问题是, 给出一个最左边和最右边不同时是括号的长表达式, 我们要怎么正确地将它分裂成两个子表达式? 我们定义"主运算符"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(例如当一个表达式的最后一步是减法运算时, 它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, 就是要找到它的主运算符. 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. 根据主运算符的定义, 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表达式中寻找主运算符了:

- 非运算符的token不是主运算符.
- 出现在一对括号中的token不是主运算符. 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.

- 主运算符的优先级在表达式中是最低的. 这是因为主运算符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主运算符. 一个例子是 `1 + 2 + 3`, 它的主运算符应该是右边的 `+`.

要找出主运算符, 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主运算符.

找到了正确的主运算符之后, 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, 然后再根据主运算符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 主运算符 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);
    }
  }
}
```

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> ")"`, 需要使用主运算符的方式进行处理, 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.

最后, 为了方便统一, 我们认为所有结果都是`uint32_t`类型.

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.

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.

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".

如何测试你的代码

你将来是要使用你自己实现的表达式求值功能来帮助你来进行后续的调试的, 这意味着程序设计课上那种"代码随便测试一下就交上去然后就可以撒手不管"的日子已经一去不复返了. 测试需要测试用例, 通过越多测试, 你就会对代码越有信心. 但如果让你来设计测试用例, 设计十几个你就会觉得没意思了, 有没有一种方法来自动产生测试用例呢?

一种常用的方法是随机测试. 首先我们需要来思考如何随机生成一个合法的表达式. 事实上, 表达式生成比表达式求值要容易得多. 同样是上面的BNF, 我们可以很容易写出生成表达式的框架:

```
void gen_rand_expr() {
  switch (choose(3)) {
    case 0: gen_num(); break;
    case 1: gen('('); gen_rand_expr(); gen(')'); break;
    default: gen_rand_expr(); gen_rand_op(); gen_rand_expr(); break;
  }
}
```

你应该一眼就能明白上述代码是如何工作的: 其中 `uint32_t choose(uint32_t n)` 是一个很简单又很重要的函数, 它的作用是生成一个小于 `n` 的随机数, 所有随机生成的内容几乎都是通过它来选择的.

有了这些随机表达式作为测试输入, 我们怎么知道输出对不对呢? 如果要我们把这些表达式手动算一遍, 那就太麻烦了. 如果可以在生成这些表达式的同时, 也能生成它们的结果, 这样我们就能得到类似OJ的测试用例啦! 但我们在NEMU中实现的表达式求值是经过了一些简化的, 所以我们需要一种满足以下条件的"计算器":

- 进行的都是无符号运算

- 数据宽度都是32bit
- 溢出后不处理

嘿! 如果我们把这些表达式塞到如下C程序的源文件里面:

```
#include <stdio.h>
int main() {
  unsigned result = ???; // 把???替换成表达式
  printf("%u", result);
  return 0;
}
```

然后用gcc编译它并执行, 让它输出表达式的结果, 这不就是我们想要的"计算器"吗?

还真能这样做! 我们已经准备好这个表达式生成器的框架代码了(在 `nemu/tools/gen-expr/gen-expr.c` 中). 你需要实现其中的 `void gen_rand_expr()` 函数, 将随机生成的表达式输出到缓冲区 `buf` 中. `main` 函数中的代码会调用你实现的 `gen_rand_expr()`, 然后把 `buf` 中的随机表达式放入上述C程序的代码中. 剩下的事情就是编译运行这个C程序了, 代码中使用了 `system()` 和 `popen()` 等库函数来实现这一功能. 最后, 框架代码将这个C程序的打印结果和之前随机生成的表达式一同输出, 这样就生成了一组测试用例.

表达式生成器如何获得C程序的打印结果?

> 代码中这部分的内容没有任何注释, 聪明的你也许马上就反应过来: 竟然是个RTFM的圈套! 阅读手册了解API的具体行为可是程序员的基本功. 如果觉得去年一整年的程序员都白当了, 就从现在开始好好锻炼吧.

不过实现的时候, 你很快就会发现需要面对一些细节的问题:

- 如何保证表达式进行无符号运算?
- 如何随机插入空格?
- 如何生成长表达式, 同时不会使 `buf` 溢出?
- 如何过滤求值过程中有除0行为的表达式?

这些问题大多都和C语言相关, 就当作是C语言的又一个编程练习吧.

为什么要使用无符号类型? (建议二周目思考)

> 我们在表达式求值中约定, 所有运算都是无符号运算. 你知道为什么要这样约定吗? 如果进行有符号运算, 有可能会发生什么问题?

除0的确切行为

如果生成的表达式有除0行为, 你编写的表达式生成器的行为又会怎么样呢?

过滤除0行为的表达式

乍看之下这个问题不好解决, 因为框架代码只负责生成表达式, 而检测除0行为至少要对表达式进行求值. 结合前两个蓝框题的回答(前提是你对它们的了解都足够深入了), 你就会找到解决方案了, 而且解决方案不唯一喔!

实现表达式生成器

根据上文内容, 实现表达式生成器. 实现后, 就可以用来生成表达式求值的测试用例了.

```
./gen-expr 100 > input
```

将会生成100个测试用例到 `input` 文件中, 其中每行为一个测试用例, 其格式为

```
结果 表达式
```

再稍微改造一下NEMU的 `main()` 函数, 让其读入 `input` 文件中的测试表达式后, 直接调用 `expr()` , 并与结果进行比较. 为了容纳长表达式的求值, 你还需要对 `tokens` 数组的大小进行修改.

随着你的程序通过越来越多的测试, 你会对你的代码越来越有信心.

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.

扩展表达式求值的功能

你之前已经实现了算术表达式的求值, 但这些表达式都是常数组成的, 它们的值不会发生变化. 这样的表达式在监视点中没有任何意义, 为了发挥监视点的功能, 你首先需要扩展表达式求值的功能.

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>              # 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:

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

扩展表达式求值的功能

你需要实现上述BNF中列出的功能. 上述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 `&&` 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.

测试的局限

我们之前实现了一个表达式生成器, 但给表达式求值加入了寄存器使用和指针解引用这两个功能之后, 表达式生成器就不能满足我们的所有需求了. 这是因为在C程序中, 寄存器的语义并不存在, 而指针解引用的语义则与NEMU大不相同.

这里想说明的是, 测试也会有局限性, 没有一种技术可以一劳永逸地解决所有问题. 前沿的研究更是如此: 它们很多时候只能解决一小部分的问题. 然而这个表达式生成器还是给你带来了很大的信心, 去思考如何方便地测试, 哪怕是进行一部分的测试, 也是有其价值的.

实现监视点

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

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

  /* TODO: Add more members if necessary */


} WP;
```

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

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

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

实现监视点池的管理

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

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

其中 `new_wp()` 从 `free_` 链表中返回一个空闲的监视点结构, `free_wp()` 将 `wp` 归还到 `free_` 链表中, 这两个函数会作为监视点池的接口被其它函数调用. 需要注意的是, 调用 `new_wp()` 时可能会出现没有空闲监视点结构的情况, 为了简单起见, 此时可以通过 `assert(0)` 马上终止程序. 框架代码中定义了32个监视点结构, 一般情况下应该足够使用, 如果你需要更多的监视点结构, 你可以修改 `NR_WP` 宏的值.

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

温故而知新(2)

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

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

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

实现监视点

你需要实现上文描述的监视点相关功能, 实现了表达式求值之后, 监视点实现的重点就落在了链表操作上.

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.

调试工具与原理

在实现断点的过程中, 你很有可能会碰到段错误. 如果你因此而感觉到无助, 你应该好好阅读这一小节的内容.

我们来简单梳理一下段错误发生的原因. 首先, 机器永远是对的. 如果程序出了错, 先怀疑自己的代码有bug. 比如由于你的疏忽, 你编写了 `if (p = NULL)` 这样的代码. 但执行到这行代码的时候, 也只是 `p` 被赋值成 `NULL` , 程序还会往下执行. 然而等到将来对 `p` 进行了解引用的时候, 才会触发段错误, 程序彻底崩溃.

我们可以从上面的这个例子中抽象出一些软件工程相关的概念:

- Fault: 实现错误的代码, 例如 `if (p = NULL)`
- Error: 程序执行时不符合预期的状态, 例如 `p` 被错误地赋值成 `NULL`
- Failure: 能直接观测到的错误, 例如程序触发了段错误

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

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

理解了这些原因之后, 我们就可以制定相应的策略了:

- 尽可能把fault转变成error. 这其实就是测试做的事情, 所以我们在上一节中加入了表达式生成器的内容, 来帮助大家进行测试, 后面的实验内容也会提供丰富的测试用例. 但并不是有了测试用例就能把所有fault都转变成error了, 因为这取决于测试的覆盖度. 要设计出一套全覆盖的测试并不是一件简单的事情, 越是复杂的系统, 全覆盖的测试就越难设计. 但是, 如何提高测试的覆盖度, 是学术界一直以来都在关注的问题.

你会如何测试你的监视点实现?

我们没有提供监视点相关的测试, 思考一下, 你会如何测试?

当然, 将来边用边测也是一种说得过去的方法, 就看你对自己代码的信心了.

- 尽早观测到error的存在. 观测到error的时机直接决定了调试的难度: 如果等到触发failure的时候才发现error的存在, 调试就会比较困难; 但如果能在error刚刚触发的时候就观测到它, 调试难度也就大大降低了. 事实上, 你已经见识过一些有用的工具了:
  - `-Wall`, `-Werror`: 在编译时刻把潜在的fault直接转变成failure. 这种工具的作用很有限, 只能寻找一些在编译时刻也觉得可疑的fault, 例如 `if (p = NULL)`. 不过随着编译器版本的增强, 编译器也能发现代码中的一些未定义行为. 这些都是免费的午餐, 不吃就真的白白浪费了.
  - `assert()`: 在运行时刻把error直接转变成failure. `assert()` 是一个很简单却又非常强大的工具, 只要在代码中定义好程序应该满足的特征, 就一定能在运行时刻将不满足这些特征的error拦截下来. 例如链表的实现, 我们只需要在代码中插入一些很简单的 `assert()` (例如指针解引用时不为空), 就能够几乎告别段错误. 但是, 编写这些 `assert()` 其实需要我们对程序的行为有一定的了解, 同时在程序特征不易表达的时候, `assert()` 的作用也较为有限.
  - `printf()`: 通过输出的方式观察潜在的error. 这是用于回溯fault时最常用的工具, 用于观测程序中的变量是否进入了错误的状态. 在NEMU中我们提供了输出更多调试信息的宏 `Log()`, 它实际上封装了 `printf()` 的功能. 但由于 `printf()` 需要根据输出的结果人工判断是否正确, 在便利程度上相对于 `assert()` 的自动判断就逊色了不少.
  - GDB: 随时随地观测程序的任何状态. 调试器是最强大的工具, 但你需要在程序行为的茫茫大海中观测那些可疑的状态, 因此使用起来的代价也是最大的.

强大的GDB

尝试在GDB中运行触发段错误的程序, 你能获得什么有用的信息吗?

根据上面的分析, 我们就可以总结出一些调试的建议:

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

如果你在程序设计课上听说过上述这些建议, 相信你几乎不会遇到过运行时错误.

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.

如何提高断点的效率 (建议二周目思考)

如果你在运行稍大一些的程序(如microbench)的时候使用断点, 你会发现设置断点之后会明显地降低NEMU执行程序的效率. 思考一下这是为什么? 有什么方法解决这个问题吗?

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

一点也不能长?

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

随心所欲的断点

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

NEMU的前世今生

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

i386手册

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

学会使用目录

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

尝试通过目录定位关注的问题

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

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

逐步细化搜索范围

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

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

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

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

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

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

必答题

你需要在实验报告中回答下列问题:

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

  那么这个学期下来, 你将会在调试上花费多少时间?

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

  事实上, 这些数字也许还是有点乐观, 例如就算使用GDB来直接调客户程序, 这些数字假设你能通过10分钟的时间排除一个bug. 如果实际上你需要在调试过程中获取并分析更多的信息, 简易调试器这一基础设施能带来的好处就更大.

- 查阅i386手册 理解了科学查阅手册的方法之后, 请你尝试在i386手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - EFLAGS寄存器中的CF位是什么意思?
  - ModR/M字节是什么?
  - mov指令的具体格式是怎么样的?
- shell命令 完成PA1的内容之后, `nemu/` 目录下的所有.c和.h和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 `pa1` 分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到"过去"?) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, `nemu/` 目录下的所有 `.c` 和 `.h` 文件总共有多少行代码?

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

温馨提示

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

PA2 - 简单复杂的机器: 冯诺依曼计算机系统

世界诞生的故事 - 第二章

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

代码管理

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

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

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 40小时

截止时间: 本次实验的阶段性安排如下:

- task PA2.1: 在NEMU中运行第一个C程序 `dummy` - 2018/10/07 23:59:59
- task PA2.2: 实现更多的指令, 在NEMU中运行所有 `cputest` - 2018/10/28 23:59:59
- task PA2.3: 运行打字小游戏, 提交完整的实验报告 - 2018/11/04 23:59:59

提交说明: 见这里

不停计算的机器

在PA1中, 我们已经见识到最简单的计算机TRM的工作方式:

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

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

取指(instruction fetch, IF)

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

译码(instruction decode, ID)

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

```
10111001 00110100 00010010 00000000 00000000
```

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

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

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

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

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

执行(execute, EX)

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

更新 `%eip`

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

于是, 计算机不断地重复上述四个步骤, 不断地执行指令, 直到永远.

也许你会疑惑, 这个只能做加法的TRM, 究竟还能做些什么呢? 对于采用补码表示的计算机, 能做加法自然就能做减法. 如果再添加一条条件跳转指令 `jnz r, addr`：当寄存器 `r` 不为 `0` 时, `%eip` 跳转到 `addr` 处, TRM就大不一样了. 例如通过 `jnz` 和 `dec` 的组合可以实现循环, 循环执行 `inc` 可以实现任意数的加法, 循环执行加法可以实现乘法, 函数调用可以看成一种特殊的跳转, 递归本质上就是函数调用... 这下可不得了了, 没想到这个弱不禁风的TRM竟然深藏着擎天撼地的威力! 不过, 虽然这个只有三条指令的TRM可以解决所有可计算的问题, 但却低效得让人无法忍受. 为此, 先驱决定往TRM中加入更多高效的指令.

RTFM

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

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

i386手册勘误

由于PDF版本的i386手册的印刷错误较多, 一定程度上影响理解, 我们在github上开放了一个repo, 用于提供修复印刷错误的版本. 同时我们也为修复错误后的版本提供在线的HTML版本.

如果你在做实验的过程中也发现了新的错误, 欢迎帮助我们修复这些错误.

RISC - 与CISC平行的另一个世界

你是否觉得x86指令集的格式特别复杂? 这其实是CISC的一个特性, 不惜使用复杂的指令格式, 牺牲硬件的开发成本, 也要使得一条指令可以多做事情, 从而提高代码的密度, 减小程序的大小. 随着时代的发展, 架构师发现CISC中复杂的控制逻辑不利于提高处理器的性能, 于是RISC应运而生. RISC的宗旨就是简单, 指令少, 指令长度固定, 指令格式统一, 这和KISS法则有异曲同工之妙. 这里有一篇对比RISC和CISC的小短文.

另外值得推荐的是这篇文章, 里面讲述了一个从RISC世界诞生, 到与CISC世界融为一体的故事, 体会一下RISC的诞生对计算机体系结构发展的里程碑意义.

RTFSC(2)

下面我们来介绍NEMU的框架代码是如何执行指令的.

数据结构

首先先对这个过程中的两个重要的数据结构进行说明.

- `nemu/src/cpu/exec/exec.c` 中的 `opcode_table` 数组. 这就是我们之前提到的译码查找表了, 这一张表通过操作码opcode来索引, 每一个opcode对应相应指令的译码函数, 执行函数, 以及操作数宽度.

- `nemu/src/cpu/decode/decode.c` 中的 `decoding` 结构. 它用于记录一些全局译码信息供后续使用, 包括操作数的类型, 宽度, 值等信息. 其中的 `src` 成员, `src2` 成员和 `dest` 成员分别代表两个源操作数和一个目的操作数. `nemu/include/cpu/decode.h` 中定义了三个宏 `id_src` , `id_src2` 和 `id_dest` , 用于方便地访问它们.

执行流程

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

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

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

在 `exec_real()` 中:

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

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

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

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

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

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

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

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

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

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

立即数背后的故事

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

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

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

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

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

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

- 假设我们需要将NEMU运行在Motorola 68k的机器上(把NEMU的源代码编译成

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

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

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

结构化程序设计

细心的你会发现以下规律:

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

这意味着, 如果独立实现每条指令不同形式不同操作数宽度的helper函数, 将会引入大量重复的代码. 需要修改的时候, 相关的所有helper函数都要分别修改, 遗漏了某一处就会造成bug, 工程维护的难度急速上升.

Copy-Paste - 一种糟糕的编程习惯

事实上, 第一版PA发布的时候, 框架代码就恰恰是引导大家独立实现每一个helper函数. 大家在实现指令的时候, 都是把已有的代码复制好几份, 然后进行一些微小的改动(例如把 `<<` 改成 `>>`). 当你发现这些代码有bug的时候, 噩梦才刚刚开始. 也许花了好几天你又调出一个bug的时候, 才会想起这个bug你好像之前在哪里调过. 你也知道代码里面还有类似的bug, 但你已经分辨不出哪些代码是什么时候从哪个地方复制过来的了. 由于当年的框架代码没有足够重视编程风格, 导致学生深深地陷入调试的泥淖中, 这也算是PA的一段黑历史了.

这种糟糕的编程习惯叫Copy-Paste, 经过上面的分析, 相信你也已经领略到它的可怕了. 事实上, 周圆圆教授的团队在2004年就设计了一款工具CP-Miner, 来自动检测操作系统代码中由于Copy-Paste造成的bug. 这个工具还让周圆圆教授收获了一篇系统方向顶级会议OSDI的论文, 这也是她当时所在学校UIUC史上的第一篇系统方向的顶级会议论文.

不过, 之后周圆圆教授发现, 相比于操作系统, 应用程序的源代码中Copy-Paste的现象更加普遍. 于是她们团队把CP-Miner的技术应用到应用程序的源代码中, 并创办了PatternInsight公司. 很多IT公司纷纷购买PatternInsight的产品, 并要求提供相应的定制服

务.

这个故事折射出, 大公司中程序员的编程习惯也许不比你好多少, 他们也会写出Copy-Paste这种难以维护的代码. 但反过来说, 重视编码风格这些企业看中的能力, 你从现在就可以开始培养.

一种好的做法是把译码, 执行和操作数宽度的相关代码分离开来, 实现解耦, 也就是在程序设计课上提到的结构化程序设计. 在框架代码中, 实现译码和执行之间的解耦的是 `idex()` 函数, 它依次调用 `opcode_table` 表项中的译码和执行的helper函数, 这样我们就可以分别编写译码和执行的helper函数了. 实现操作数宽度和译码, 执行这两者之间的解耦的是 `id_src` , `id_src2` 和 `id_dest` 中的 `width` 成员, 它们记录了操作数宽度, 译码和执行的过程中会根据它们进行不同的操作, 通过同一份译码函数和执行函数实现不同操作数宽度的功能.

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

宏	含义
`nemu/include/macro.h`	
`str(x)`	字符串 `"x"`
`concat(x, y)`	token `xy`
`nemu/include/cpu/decode.h`	
`id_src`	全局变量 `decoding` 中源操作数成员的地址
`id_src2`	全局变量 `decoding` 中2号源操作数成员的地址
`id_dest`	全局变量 `decoding` 中目的操作数成员的地址
`make_Dhelper(name)`	名为 `decode_name` 的译码函数的原型说明
`nemu/src/cpu/decode.c`	
`make_Dophelper(name)`	名为 `decode_op_name` 的操作数译码函数的原型说明
`nemu/include/cpu/exec.h`	
`make_Ehelper(name)`	名为 `exec_name` 的执行函数的原型说明
`print_asm(...)`	将反汇编结果的字符串打印到缓冲区 `decoding.assembly` 中
`suffix_char(width)`	操作数宽度 `width` 对应的后缀字符
`print_asm_template[1|2|3](instr)`	打印单/双/三目操作数指令 `instr` 的反汇编结果

用RTL表示指令行为

我们知道, x86指令作为一种CISC指令集, 不少指令的行为都比较复杂. 但我们会发现, i386手册会用一些更简单的操作来表示指令的具体行为. 这说明, 复杂的x86指令还是能继续分解成一些更简单的操作的组合. 如果我们先实现这些简单操作, 然后再用它们来实现x86指令, 不就可

以进一步提高代码的复用率了吗?

在NEMU中, 我们使用RTL(寄存器传输语言)来描述这些简单的操作. 下面我们对NEMU中使用的RTL进行一些说明, 首先是RTL寄存器的定义. 在NEMU中, RTL寄存器统一使用 `rtlreg_t` 来定义, 而 `rtlreg_t` (在 `nemu/include/common.h` 中定义)其实只是一个 `uint32_t` 类型:

```
typedef uint32_t rtlreg_t;
```

在NEMU中, RTL寄存器只有以下这些

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

有了RTL寄存器, 我们就可以定义RTL指令对它们进行的操作了. 在NEMU中, RTL指令有两种(在 `nemu/include/cpu/rtl.h` 中定义). 一种是RTL基本指令, 它们的特点是不需要使用临时寄存器, 可以看做是最基本的x86指令中的最基本的操作. RTL基本指令包括(我们使用了一些简单的正则表达式记号):

- 立即数读入 `rtl_li`
- 寄存器传输 `rtl_mv`
- 32位寄存器-寄存器类型的算术/逻辑运算, 包括 `rtl_(add|sub|and|or|xor|shl|shr|sar|i?mul_[lo|hi]|i?div_[q|r])` , 这些运算的定义用到了 `include/util/c_op.h` 中的C语言运算
- 被除数为64位的除法运算 `rtl_i?div64_[q|r]`
- guest内存访问 `rtl_lm` 和 `rtl_sm`
- host内存访问 `rtl_host_lm` 和 `rtl_host_sm`
- 关系运算 `rtl_setrelop` , 具体可参考 `src/cpu/exec/relop.c`
- 跳转, 包括直接跳转 `rtl_j` , 间接跳转 `rtl_jr` 和条件跳转 `rtl_jrelop`
- 终止程序 `rtl_exit`

上述RTL基本指令在 `nemu/include/cpu/rtl.h` 中定义时, 添加了 `interpret_` 前缀, 这是为了给PA5作准备, 在 `nemu/include/cpu/rtl-wrapper.h` 的作用下, 其它代码中使用到这些RTL基本指令时会自动添加 `interpret_` 前缀. 因此你在代码中使用它们的时候, 只需要编写 `rtl_xxx` 即可.

神秘的host内存访问 (建议二周目思考)

为什么需要有host内存访问的RTL指令呢?

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

- 32位寄存器-立即数类型的算术/逻辑运算, 包括 `rtl_(add|sub|and|or|xor|shl|shr|sar|i?mul_[lo|hi]|i?div_[q|r])_i`
- 通用寄存器访问 `rtl_lr` 和 `rtl_sr`
- EFLAGS标志位的读写 `rtl_set_(CF|OF|ZF|SF)` 和 `rtl_get_(CF|OF|ZF|SF)`
- 其它常用功能, 如按位取反 `rtl_not` , 符号扩展 `rtl_sext` 等

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

小型调用约定

我们定义RTL基本指令的时候, 约定了RTL基本指令不需要使用RTL临时寄存器. 但某些RTL伪指令需要使用临时寄存器存放中间结果, 才能实现其完整功能. 这样可能会带来寄存器覆盖的问题, 例如如下RTL指令序列:

```
(1) rtl_mv(&t0, &t1);
(2) rtl_sext(&t1, &t2, 1);  // use t0 temporarily
(3) rtl_add(&t2, &t0, &t1);
```

如果实现 `(2)` 的时候恰好使用到了 `t0` 作为临时寄存器, 在 `(3)` 中使用的 `t0` 就不再是 `(1)` 的结果了, 从而产生非预期的结果.

为了尽可能避免上述问题, 我们有两条约定:

- 实现RTL伪指令的时候, 尽可能不使用 `dest` 之外的寄存器存放中间结果. 由于 `dest` 最后会被写入新值, 其旧值肯定要被覆盖, 自然也可以安全地作为RTL伪指令的临时寄存器.
- 实在需要使用临时寄存器的时候, 使用 `at` . `at` 全称是 `assembly temporary` , 是MIPS ABI中定义的一个特殊寄存器: 编译器并不会使用它, 它可以在编写汇编代码的时候安全地作为可使用的临时寄存器. 在这里, 我们借鉴它的功能来作如下约定: 不要在RTL伪指令的内部实现之外使用 `at` . 这样, `at` 就可以安全地作为RTL伪指令的临时寄存器了.

仔细体会上述约定, 你也许会发现, 这和课上学习的调用约定有那么一点点相似之处. 如果把RTL伪指令看成一个函数调用, 我们刚才其实在讨论, 在这个"函数"里面究竟可以使用哪些RTL寄存器. 在调用约定中, 有些寄存器对被调用函数来说, 使用它们之前是需要先保存的. 但我们的RTL编程模型中并没有"栈"的概念, 所以在RTL中我们就不设置所谓的"被调用者保存寄存器"了. 从某种程度上来说, 这样的"小型调用约定"很难支撑大规模RTL指令的编写. 不过幸好, 在用RTL来实现x86指令的时候, 这一"小型调用约定"已经足够使用了.

计算机系统中的约定与未定义行为

上述例子其实折射出计算机系统工作的一种基本原则: 遵守约定.

我们定义了RTL寄存器和相应的RTL指令, 基于这些定义, 原则上可以编写出任意的RTL指令序列, 这些RTL序列最终也会按照它们原本的语义来执行. 但光靠这些定义, 我们无法避免上述RTL寄存器相互覆盖造成错误的问题. 所以我们又提出一些新的约定, 来避免这个问题. 当然, 你也可以自己提出一套新的约定(比如用 `t1` 替代上述 `at` 的作用).

违反约定会发生什么呢? 最常见的就是程序无法得到正确的结果. 比如当两套约定不兼容的RTL代码放在一起的时候, 它们都分别违反了对方的约定 (你的RTL覆盖了我的 `at` , 我的RTL覆盖了你的 `t1`). 当然也有可能恰好没有覆盖各自约定使用的寄存器, 撞大运地得到正确的运行结果. 总之, 违反约定的具体行为会怎么样, 还需要具体问题具体分析, 很难明确地说清楚.

既然说不清楚, 那就干脆不说吧, 于是有了未定义行为的概念: 只要遵守约定, 就能保证程序具有遵守约定后的特性; 如果违反, 不按照说好的来, 那就不保证行为是正确的.

计算机系统就是这样工作的: 计算机系统抽象层之间的接口其实也是一种约定, 比如指令就是软件和硬件的一种接口, 所以有了i386手册来规范每一条指令的行为, 编译器需要根据i386手册中的约定来生成可以正确执行的代码. 如果编译器不按照手册约定来生成代码, 那么编译出的程序的行为就是未定义的.

引入未定义行为还有一个好处是, 给约定的实现方式一定的自由度. 例如, C语言标准规定, 整数除法的除数为0时, 结果是未定义的. x86的除法指令在检测到除数为0时, 就会向CPU抛出一个异常信号. 而MIPS的除法指令则更简单暴力: 首先在MIPS指令集手册中声明, 除数为0时, 结果未定义; 然后在硬件上实现除法器的时候, 对除0操作就可以视而不见了. 然而给定一个除法器电路, 就算除数为0, 电路的输出也总会有一个值, 至于具体的值是什么, 就看造化了, 反正C语言标准规定除0的行为本身就是未定义的, 让除法指令随便返回一个值, 也不算违反约定.

未定义行为其实离你很近, 比如你经常使用的 `memcpy()` , 如果源区间和目的区间有重叠时, 它的行为会怎么样? 如果你从来没有思考过这个问题, 你应该去 `man` 一下, 然后思考一下为什么会这样. 还有一种有人欢喜有人愁的现象是基于未定义行为的编译优化: 既然源代码的行为是未定义的, 编译器基于此进行各种奇葩优化当然也不算违反约定. 这篇文章列举了一些让你大开眼界的花式编译优化例子, 看完之后你就会刷新对程序行为的理解了.

所以这就是为什么我们强调要学会RTFM. RTFM是了解接口行为和约定的过程: 每个输入的含义是什么? 查阅对象的具体行为是什么? 输出什么? 有哪些约束条件必须遵守? 哪些情况下会报什么错误? 只有完全理解并遵守它们, 才能正确无误地使用查阅的对象, 大至系统设计原则, 小到一个 `memcpy()` 的行为, 都蕴含着约定与遵守的法则. 理解这些法则, 也是理解计算机系统的不二途径.

RTL寄存器中值的生存期

在程序设计课上, 我们知道C语言中不同的变量有不同的生存期: 有的变量的值会一直持续到程序结束, 但有的变量却很快消亡. 在上述定义的RTL寄存器中, 其实也有不同的生存期. 尝试根据生存期给RTL寄存器分类.

尽管目前这个分类结果并没有什么用处, 但其实将来在PA5中设计RTL优化方案的时候, 生存期的性质会给我们提供很大的优化机会.

实现新指令

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

1. 在 `opcode_table` 中填写正确的译码函数, 执行函数以及操作数宽度
2. 用RTL实现正确的执行函数, 需要注意使用RTL伪指令时要遵守上文提到的小型调用约定

框架代码把绝大部分译码函数和执行函数都定义好了, 你可以很方便地使用它们.

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

运行第一个C程序

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

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

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

在NEMU中运行 `dummy` 程序, 你会发现NEMU输出以下信息:

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

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

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

for more details.

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

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

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

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

- `call` : `call` 指令有很多形式, 不过在PA中只会用到其中的几种, 现在只需要实现 `CALL rel32` 的形式就可以了. 至于跳转地址, 框架代码里面已经有不少提示了, 也就算作是RTFSC的一个练习吧.
- `push` , `pop` : 现在只需要实现 `PUSH r32` 和 `POP r32` 的形式就可以了, 它们可以很容易地通过 `rtl_push` 和 `rtl_pop` 来实现
- `sub` : 在实现 `sub` 指令之前, 你首先需实现EFLAGS寄存器. 你只需要在寄存器结构体中添加EFLAGS寄存器即可. EFLAGS是一个32位寄存器, 但在NEMU中, 我们只会用到EFLAGS中以下的5个位: `CF` , `ZF` , `SF` , `IF` , `OF` , 它们的功能可暂不实现. 关于EFLAGS中每一位的含义, 请查阅i386手册. 实现了EFLAGS寄存器之后, 再实现相关的RTL指令, 之后你就可以通过这些RTL指令来实现 `sub` 指令了
- `xor` , `ret` : RTFM吧

运行第一个客户程序

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

温馨提示

PA2阶段1到此结束.

程序, 运行时环境与AM

运行时环境

我们已经成功在TRM上运行 `dummy` 程序了, 然而这个程序什么都没做就结束了, 一点也不过瘾啊. 为了让NEMU支持大部分程序的运行, 你还需要实现更多的指令. 但并不是有了足够的指令就能运行更多的程序. 我们之前提到"并不是每一个程序都可以在NEMU中运行", 现在我们来解释一下背后的缘由.

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

当然, 我们先来考虑最简单的运行时环境是什么样的. 换句话说, 为了运行最简单的程序, 我们需要提供什么呢? 其实答案已经在PA1中了: 只要把程序放在正确的内存位置, 然后让 `%eip` 指向第一条指令, 计算机就会自动执行这个程序, 永不停止.

不过, 虽然计算机可以永不停止地执行指令, 但一般的程序都是会结束的, 所以运行时环境需要向程序提供一种结束运行的方法. 聪明的你已经能想到, 我们在PA1中提到的那条人工添加的 `nemu_trap` 指令, 就是让程序来结束自己的运行的.

所以, 只要有内存, 有结束运行的方式, 加上实现正确的指令, 就可以支撑最简单程序的运行了. 而这, 也可以算是最简单的运行时环境了.

将运行时环境封装成库函数

我们刚才讨论的运行时环境是直接位于计算机硬件之上的, 因此运行时环境的具体实现, 也是和机器相关的. 以程序结束为例, NEMU中是使用人为添加的 `nemu_trap` 指令, 但如果我们自己用verilog设计了一个CPU, 有可能是通过一条 `mycpu_trap` 指令来结束程序, 它和 `nemu_trap` 指令有很大概率是不一样的. 而结束运行是程序共有的需求, 为了让 `n` 个程序运行在 `m` 个机器上, 难道我们要维护 `n*m` 份代码? 有没有更好的方法呢?

对于同一个程序, 如果能把 `m` 个版本不同的部分都转换成相同的代码, 我们就只需要维护一个版本就可以了. 而实现这个目标的杀手锏, 就是你在程序设计课上学过的抽象! 我们只需要定义一个结束程序的API, 比如 `void _halt()` , 它对不同机器上程序的不同结束方式进行了抽象: 程序只要调用 `_halt()` 就可以结束运行, 而不需要关心自己运行在哪一个机器上. 经过抽象之后, 之前 `m` 个版本的程序, 现在都统一通过 `_halt()` 来结束运行, 我们就只需要维护这一个通

过 `_halt()` 来结束运行的版本就可以了. 然后, 不同的机器分别实现自己的 `_halt()` , 就可以支撑 `n` 个程序的运行! 这样以后, 我们就可以把程序和机器解耦了: 我们只需要维护 `n+m` 份代码 (`n` 个程序和 `m` 个机器相关的 `_halt()`), 而不是之前的 `n*m` .

这个例子也展示了运行时环境的一种普遍的存在方式: 库. 通过库, 运行程序所需要的公共要素被抽象成API, 不同的机器只需要实现这些API, 也就相当于实现了支撑程序运行的运行时环境, 这提升了程序开发的效率: 需要的时候只要调用这些API, 就能使用运行时环境提供的相应功能.

究竟有什么实质性的好处

> 也许上面的文字并不能"忽悠"清醒的你: 现在不就只有NEMU这一个机器吗(`m=1`)? 哪里需要抽象?
>
> 目前确实是这样. 但不妨思考一下, 如果有多个机器, 这样的抽象还会带来哪些好处呢? 你很快就会体会到这些好处了.

AM - 直接运行在计算机上的运行时环境

一方面, 正如上文提到, 应用程序的运行都需要运行时环境的支持; 另一方面, 只进行纯粹计算任务的程序在TRM上就可以运行, 更复杂的应用程序对运行时环境必定还有其它的需求: 例如你之前玩的超级玛丽需要和用户进行交互, 至少需要运行时环境提供输入输出的支持. 要运行一个现代操作系统, 还要在此基础上加入更高级的功能.

如果我们把这些需求都收集起来, 将它们抽象成统一的API提供给程序, 这样我们就得到了一个可以支撑各种程序运行在各种机器上的库了! 具体地, 每个机器都按照它们的特性实现这组API; 应用程序只需要直接调用这组API即可, 无需关心自己将来运行在哪个机器上. 由于这组统一抽象的API代表了程序运行对机器的需求, 所以我们把这组API称为抽象计算机.

AM(Abstract machine)项目就是这样诞生的. 作为一个向程序提供运行时环境的库, AM根据程序的需求把库划分成以下模块

```
AM = TRM + IOE + CTE + VME + MPE
```

- TRM(Turing Machine) - 图灵机, 最简单的运行时环境, 为程序提供基本的计算能力
- IOE(I/O Extension) - 输入输出扩展, 为程序提供输出输入的能力
- CTE(Context Extension) - 上下文扩展, 为程序提供上下文管理的能力
- VME(Virtual Memory Extension) - 虚存扩展, 为程序提供虚存管理的能力
- MPE(Multi-Processor Extension) - 多处理器扩展, 为程序提供多处理器通信的能力 (MPE超出了ICS课程的范围, 在PA中不会涉及)

AM给我们展示了程序与计算机的关系: 利用计算机硬件的功能实现AM, 为程序的运行提供它们所需要的运行时环境. 感谢AM项目的诞生, 让NEMU和程序的界线更加泾渭分明, 同时使得PA的流程更加明确:

```
(在NEMU中)实现硬件功能 -> (在AM中)提供运行时环境 -> (在APP层)运行程序
(在NEMU中)实现更强大的硬件功能 -> (在AM中)提供更丰富的运行时环境 -> (在APP层)运行更复杂的程序
```

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

AM的诞生和ProjectN的故事

在AM诞生之前, ProjectN的各个主要部件就已经存在了:

- NEMU - NJU EMUlator (系统基础实验)
- Nanos - Nanjing U OS (操作系统实验)
- NOOP - NJU Out-of-Order Processor (组成原理实验)
- NCC - NJU C Compiler (编译原理实验)

但我们一直没想好, 如何把这些部件集成到一个完整的教学生态系统中.

在2017年春季的计算机系统综合实验课程中, jyy首先提出AM的思想, 把程序和机器解耦. 解耦之后, AM就成了ProjectN的一把关键的钥匙: 只要实现了AM, 我们就可以在NEMU和NOOP上运行各种AM程序; 只要在AM上实现Nanos, 我们就可以把Nanos运行在NEMU和NOOP上; 只要NCC把程序编译到AM上, 我们就可以在NOOP上运行NCC编译的程序.

经过几个月的尝试, 我们很快就相信, 这条路是对的. 于是临时决定将2017年秋季的PA进行大改版, 借鉴AM的思想来设计开发NEMU, 期望大家能更好地理解"程序如何在计算机上运行". 因此2017年秋季版本的NEMU, 也算是第一次正式作为一个子项目收录到ProjectN教学生态系统中.

我们已经连续两年组队参加计算机系统设计大赛"龙芯杯", 在大赛上展示我们独有的ProjectN生态系统, 均获得第二名的好成绩. 我们在大赛中探索出来的好方法, 也会反馈到PA中. 这些离你其实并不遥远, 我们在PA中传递出来的做事方法和原则, 都是大赛得奖的黄金经验.

如果你对AM和ProjectN感兴趣, 欢迎联系jyy或yzh.

穿越时空的羁绊

有了AM, 我们就可以把课程之间的实验打通, 做一些以前做不到的有趣的事情了. 比如今年春季的操作系统课上, 你的学长学姐在AM上编写了他们自己的小游戏. 在今年PA的后期, 你将有机会把学长学姐们编写的游戏无缝地移植到NEMU上, 作为最终系统展示的一部分, 想想都是一件激动人心的事情.

为什么要有AM? (建议二周目思考)

操作系统也有自己的运行时环境. AM和操作系统提供的运行时环境有什么不同呢? 为什么会有这些不同?

RTFSC(3)

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

```
nexus-am
├── am                              # AM相关
│   ├── am.h
│   ├── arch                        # 不同机器的AM实现
│   │   ├── native
│   │   └── x86-nemu                # x86-nemu的AM实现
│   │       ├── img                 # 构建/运行二进制文件/镜像的脚本
│   │       │   ├── boot
│   │       │   │   ├── Makefile
│   │       │   │   └── start.S     # 程序入口
│   │       │   ├── build           # 构建脚本
│   │       │   ├── loader.ld       # 链接脚本
│   │       │   └── run             # 运行脚本
│   │       ├── include
│   │       ├── README.md
│   │       └── src
│   │           ├── cte.c           # CTE
│   │           ├── ioe.c           # IOE
│   │           ├── trap.S
│   │           ├── trm.c           # TRM
│   │           └── vme.c           # VME
│   └── Makefile
├── apps                            # 直接运行在AM上的应用
├── libs                            # 可以直接运行在AM上的库
├── Makefile
├── Makefile.app
├── Makefile.check
├── Makefile.compile
├── Makefile.lib
└── tests                           # 直接运行在AM上的测试
```

整个AM项目分为三大部分:

- `nexus-am/am` - 不同机器的AM API实现, 目前我们只需要关注 `nexus-am/am/arch/x86-nemu` 即可
- `nexus-am/tests` 和 `nexus-am/apps` - 一些功能测试和直接运行在AM上的应用程序
- `nexus-am/libs` - 一些机器无关的函数库, 方便应用程序的开发

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

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

堆区是给程序自由使用的一段内存区间, 为程序提供动态分配内存的功能. TRM的API只提供堆区的起始和末尾, 而堆区的分配和管理需要程序自行维护. 当然, 程序也可以不使用堆区, 例如 `dummy` .

堆和栈在哪里?

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

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

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

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

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

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

解决这个问题的方法是交叉编译, 我们需要在GNU/Linux下根据AM的运行时环境编译出能够在 `x86-nemu` 这个新环境中运行的可执行文件. 为了不让链接器ld使用默认的方式链接, 我们还需要提供描述 `x86-nemu` 的运行时环境的链接脚本. AM的框架代码已经把相应的配置准备好了:

- gcc将 `x86-nemu` 的AM实现的源文件编译成目标文件, 然后通过ar将这些目标文件作为一个库, 打包成一个归档文件
- gcc把应用程序源文件(如 `nexus-am/tests/cputest/tests/dummy.c`)编译成目标文件
- 必要的时候通过gcc和ar把程序依赖的运行库(如 `nexus-am/libs/klib`)也打包成归档文件
- 执行脚本文件 `nexus-am/am/arch/x86-nemu/img/build` , 在脚本文件中
  - 将程序入口 `nexus-am/am/arch/x86-nemu/img/boot/start.S` 编译成目标文件
  - 最后让ld根据链接脚本 `nexus-am/am/arch/x86-nemu/img/loader.ld` , 将上述目标文件和归档文件链接成可执行文件

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

我们对编译得到的可执行文件的行为进行简单的梳理:

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

有了TRM这个简单的运行时环境, 我们就可以很容易地在上面运行各种"简单"的程序了. 当然, 我们也可以运行"不简单"的程序: 我们可以实现任何复杂的算法, 甚至是各种理论上可计算的问题, 都可以在TRM上解决.

运行更多的程序

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

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

然后在 `nexus-am/tests/cputest/` 目录下执行

```
make ALL=xxx run
```

其中 `xxx` 为测试用例的名称(不包含 `.c` 后缀).

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

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

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

实现更多的指令

你需要实现更多的指令, 以通过上述测试用例.

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

框架代码已经实现了部分指令, 但并没有填写 `opcode_table` . 此外, 部分函数的功能也并没有完全实现好(框架代码中已经插入了 `TODO()` 作为提示), 你还需要编写相应的功能.

由于 `string` 和 `hello-str` 还需要实现额外的内容才能运行(具体在下文介绍), 目前可以先使用其它测试用例进行测试.

push imm8指令行为补充

需要注意的是, `push imm8` 指令需要对立即数进行符号扩展, 这一点在i386手册中并没有明确说明. 在IA-32手册中关于 `push` 指令有如下说明:

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

指令名对照

AT&T格式反汇编结果中的少量指令, 与i386手册中列出的指令名称不符, 如 `cltd`. 除了STFW之外, 你有办法在手册中找到对应的指令吗? 如果有的话, 为什么这个办法是有效的呢?

实现常用的库函数

我们已经在TRM上运行了不少简单的程序了, 但如果想在TRM上编写一些稍微复杂的程序, 我们就会发现有点不方便. 目前TRM这个最简单的运行时环境只提供了堆区和 `_halt()`, 但我们平时经常使用的像 `memcpy()` 这样的库函数却没有提供. 既然没有提供, 那就让我们来实现一下吧.

既然叫得起库函数, 那说明很多程序都可以用到它们, 所以我们可以像AM那样, 把它们组织成一个库. 然而和AM不同的是, 这些库函数的具体实现可以是和机器无关的: 与 `_halt()` 不同, 在NEMU上, 或者在你将来用verilog实现的CPU上, 甚至是其它的机器, `memcpy()` 都可以通过相同的方式来实现. 所以, 如果在AM中来实现这些常用的库函数, 就会引入不必要的重复代码.

一种好的做法是, 把运行时环境分成两部分: 一部分是机器相关的运行时环境, 也就是我们之前介绍的AM; 另一部分是机器无关的运行时环境, 类似 `memcpy()` 这种常用的函数应该归入这部分. 所以 `nexus-am/libs` 用于收录机器无关的库函数.

在PA中, 我们只要关注 `nexus-am/libs/klib` 就可以了. `klib` 是 `kernel library` 的意思, 用于提供一些兼容libc的基础功能. 框架代码在 `nexus-am/libs/klib/src/string.c` 和 `nexus-am/libs/klib/src/stdio.c` 中列出了将来可能会用到的库函数, 但并没有提供相应的实现.

实现字符串处理函数

根据需要实现 `nexus-am/libs/klib/src/string.c` 中列出的字符串处理函数, 让测试用例 `string` 可以成功运行. 关于这些库函数的具体行为, 请务必RTFM.

免责声明

有一些库函数可能在将来才会使用, 目前你可以选择暂时不实现它们. 但如果将来你因为忘记实现它们而导致程序出错时, 请不要抱怨讲义没有提醒你什么时候应该去实现哪个库函数.

这其实是一个代码管理的问题, 在项目中, 这种情况还是比较常见的. 比如你一下子定义了一堆API, 但不一定来得及马上把它们全部实现. 反过来, 你应该思考, 有没有更好的方法可以在你用到某个没有实现的函数的时候提醒你, 而不是让你经历一段没有必要的调试过程才发现竟然是个让你哭笑不得的原因呢?

为了运行测试用例 `hello-str`, 你还需要实现库函数 `sprintf()`. 和其它库函数相比, `sprintf()` 比较特殊, 因为它的参数数目是可变的. 为了获得数目可变的参数, 你可以使用C库 `stdarg.h` 中提供的宏, 具体用法请查阅 `man stdarg`.

实现sprintf

实现 `nexus-am/libs/klib/src/stdio.c` 中的 `sprintf()`, 具体行为可以参考 `man 3 printf`. 目前你只需要实现 `%s` 和 `%d` 就能通过 `hello-str` 的测试了, 其它功能(包括位宽, 精度等)可以在将来需要的时候再自行实现.

基础设施(2)

AM作为基础设施

编写klib, 然后在NEMU上运行 `string` 程序, 看其是否能通过测试. 表面上看, 这个做法似乎没什么不妥当, 然而如果测试不通过, 你在调试的时候肯定会思考: 究竟是klib写得不对, 还是NEMU有bug呢? 如果这个问题得不到解决, 调试的难度就会上升: 很有可能在NEMU中调了一周, 最后发现是klib的实现有bug.

你之所以会思考这个问题, 是因为软件(klib)和硬件(NEMU)都是你编写的, 它们的正确性都是不能保证的. 大家在中学的时候都学习过控制变量法: 如果能把其中一方换成是认为正确的实现, 就可以单独测试另一方的正确性了! 比如我们在真机上对klib进行测试, 如果测试没通过, 那就说明是klib的问题, 因为我们可以相信真机的硬件实现永远是对的; 相反, 如果测试通过了, 那就说明klib没有问题, 而是NEMU有bug.

一个新的问题是, 我们真的可以很容易地把软件移植到其它硬件上进行测试吗? 聪明的你应该想起来AM的核心思想了: 把程序和机器解耦. AM的思想保证了运行在AM之上的代码(包括klib)都是机器无关的, 这恰恰增加了代码的可移植性. 想象一下, 如果 `string.c` 的代码中有一条只能在NEMU中执行的 `nemu_trap` 指令, 那么它就无法在真机上运行.

`nexus-am` 中有一个特殊的AM叫 `native` , 是用GNU/Linux默认的运行时环境来实现的AM API. 例如我们通过 `gcc hello.c` 编译程序时, 就会编译到GNU/Linux提供的运行时环境, 你在PA1的开头试玩的超级玛丽, 也是编译到 `native` 上并运行. 和 `x86-nemu` 相比, `native` 有如下好处:

- 直接运行在真机上, 可以相信真机的行为永远是对的
- 就算软件有bug, 在 `native` 上调试也比较方便(例如可以使用GDB)

因此, 与其在 `x86-nemu` 中直接调试软件, 还不如在 `native` 上把软件调对, 然后再换到 `x86-nemu` 中运行, 来对NEMU进行测试. 在 `nexus-am` 中, 我们可以很容易地把程序编译到另一个AM上运行, 例如在 `nexus-am/tests/cputest/` 目录下执行

```
make ALL=string ARCH=native run
```

即可将 `string` 程序编译到 `native` 并运行. 由于我们会将程序编译到不同的AM中, 因此你需要注意 `make` 命令中的 `ARCH` 参数.

如何生成native的可执行文件

阅读相关Makefile和脚本文件, 尝试理解AM项目是如何生成 `native` 的可执行文件的.

与NEMU中运行程序不同, 由于 `cputest` 中的测试不会进行任何输出, 我们只能通过程序运行的返回值来判断测试是否成功. 如果 `string` 程序通过测试, 终端将不会输出任何信息; 如果测试不通过, 终端将会输出

```
make[1]: *** [run] Error 1
```

当然也有可能输出段错误等信息.

奇怪的错误码

为什么错误码是 `1` 呢? 你知道 `make` 程序是如何得到这个错误码的吗?

更新框架代码

我们在2018/10/06 17:00:00对 `nexus-am` 项目中的部分文件进行了更新. 如果你在此时间之前获得框架代码, 请根据这里手动修改代码; 如果你在此时间之后获得框架代码, 你不需要进行额外的操作.

别高兴太早了, 框架代码编译到 `native` 的时候默认链接到glibc, 我们需要把这些库函数的调用链接到我们编写的klib来进行测试. 我们可以通过在 `nexus-am/libs/klib/include/klib.h` 中控制是否定义宏 `__NATIVE_USE_KLIB__` , 来控制是否把库函数链接到klib. 若不定义这个宏, 库函数将会链接到glibc, 可以作为正确的参考实现来进行对比.

好了, 现在你就可以在 `native` 上测试/调试你的klib实现了, 还可以使用 `_putc()` 进行字符输出来帮助你调试. 实现正确后, 再将程序编译到 `x86-nemu` (记得移除调试时插入的 `_putc()`), 对NEMU进行测试.

编写可移植的程序

为了不损害程序的可移植性, 你编写程序的时候不能再做一些机器相关的假设了, 比如"指针的长度是4字节"将不再成立, 因为在 `native` 上指针长度是8字节, 按照这个假设编写的程序, 在 `native` 上运行很有可能会触发段错误.

当然, 解决问题的方法还是有的, 至于要怎么做, 老规矩, STFW吧.

Differential Testing

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

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

我们已经在PA1中讨论过调试的方法, 然而对于指令实现的bug, 我们会发现, 这些调试的方法还是不太奏效: 我们很难通过 `assert()` 来表达指令的正确行为来进行自动检查, 而 `printf()` 和GDB实际上并没有缩短error和failure的距离.

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

这实际上是一种非常奏效的测试方法, 在软件测试领域称为differential testing(后续简称DiffTest). 我们刚才提到了"状态", 那"状态"具体指的是什么呢? 我们在PA1中已经认识到, 计算机就是一个数字电路. 那么, "计算机的状态"就恰恰是那些时序逻辑部件的状态, 也就是寄存器和内存的值. 其实仔细思考一下, 计算机执行指令, 就是修改这些时序逻辑部件的状态的过程. 要检查指令的实现是否正确, 只要检查这些时序逻辑部件中的值是否一致就可以了! DiffTest可以非常及时地捕捉到error, 第一次发现NEMU的寄存器或内存的值与真机不一样的时候, 就是因为当时执行的指令实现有误导致的. 这时候其实离error非常接近, 防止了error进一步传播的同时, 要回溯找到fault也容易得多.

多么美妙的功能啊! 背后还蕴含着计算机本质的深刻原理! 但很遗憾, 不要忘记了, 真机上是运行了操作系统GNU/Linux的, 而NEMU中的测试程序是运行在 `x86-nemu` 上的, 我们无法在 `native` 中运行编译到 `x86-nemu` 的AM程序. 所以, 我们需要的不仅是一个i386手册的正确实现, 而且需要在上面能正确运行 `x86-nemu` 的AM程序.

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

为了方便实现DiffTest, 我们在DUT(Design Under Test, 测试对象)和REF(Reference, 参考实现) 之间定义了如下的一组API:

```
// 从DUT host memory的`src`处拷贝`n`字节到REF guest memory的`dest`处
void difftest_memcpy_from_dut(paddr_t dest, void *src, size_t n);
// 获取REF的寄存器状态到`r`
void difftest_getregs(void *r);
// 设置REF的寄存器状态为`r`
void difftest_setregs(const void *r);
// 让REF执行`n`条指令
void difftest_exec(uint64_t n);
// 初始化REF的DiffTest功能
void difftest_init();
```

其中寄存器状态 `r` 要求寄存器的值按照某种顺序排列, 若未按要求顺序排列, `difftest_getregs()` 和 `difftest_setregs()` 的行为是未定义的. REF需要实现这些API, DUT会使用这些API来进行DiffTest. 在这里, REF和DUT分别是QEMU和NEMU.

NEMU的框架代码已经准备好相应的功能了. 首先在 `nemu/tools/qemu-diff/` 目录下执行 `make`, 将用于与QEMU进行DiffTest的API编译成动态库 `qemu-so`. 然后在 `nemu/include/common.h` 中定义宏 `DIFF_TEST`, 重新编译NEMU后运行, `nemu/Makefile` 中已经设置了相应的参数, 把 `qemu-so` 作为NEMU的一个参数传入. 定义了宏 `DIFF_TEST` 之后, `nemu/src/monitor/diff-test/diff-test.c` 中的 `init_difftest()` 会额外进行以下初始化工作:

- 打开动态库文件 `ref_so_file`, 此时也就是我们之前编译好的 `qemu-so`.
- 从动态库中分别读取上述API的符号.
- 对REF的DIffTest功能进行初始化, 此时会启动QEMU, 并输出 `Connect to QEMU successfully` 的信息. 代码还会对QEMU的状态进行一些初始化工作, 但你不需要了解这些工作的具体细节. 需要注意的是, 我们让QEMU运行在后台, 因此你将看不到QEMU的任何输出.
- 将DUT的guest memory拷贝到REF中.
- 将DUT的寄存器状态拷贝到REF中.

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

实现DiffTest

上文在介绍API约定的时候, 提到了寄存器状态 `r` 需要把寄存器按照某种顺序排列. `qemu-diff` 作为REF, 已经满足API的这一约束. 你首先需要RTFSC, 从中找出这一顺序, 并检查你的NEMU实现是否已经满足约束.

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

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

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

基础设施 - 龙芯杯获胜的秘诀

DiffTest的思想非常简单: 找一个正确的实现, 跟它对比结果. 事实上, 你在PA1实现的表达式生成器, 里面也蕴含着DiffTest的思想: C程序就是REF. 框架代码提供的测试用例也是: 这些测试都会先在 `native` 上运行, 得到正确的结果, 你其实是把 `native` 作为REF, 来对比程序运行的结果(HIT GOOD/BAD TRAP).

当然, 这里介绍的DiffTest的粒度就更细致了: 我们不仅仅是对比程序运行的结果, 而是对比每条指令的行为. 这样可以帮助我们快速发现并定位指令实现的bug. 我们在龙芯杯上继承了这套思想, 把用verilog写的CPU作为DUT, 用已经实现好的NEMU作为REF, 很快就可以发现并修复verilog中的bug. 借助DiffTest, 我们在第二届龙芯杯大赛中书写了

> 一周正确实现一个全乱序执行处理器, 并在上面运行操作系统Nanos和仙剑奇侠传

的神话.

这再次体现了基础设施的重要性: 完善的基础设施使得CPU设计变得高效简单, 甚至完成了前人无法完成的任务. 有了基础设施, 令人望而却步的组成原理实验也可以脱胎换骨, 浴火重生: 你几乎不需再要看那些让你晕头转向的波形来调试硬件代码了. 最近硬件设计领域也掀起一股敏捷开发的热潮, 基础设施在其中扮演的角色就不言而喻了. 如果你对龙芯杯感兴趣, 欢迎联系我们, 和我们一同探索基础设施完善的方向.

一键回归测试

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

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

```
bash runall.sh
```

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

NEMU的本质

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

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

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

用x86指令来描述, 就是 `inc` , `dec` 和 `jne` 三条指令.

令人更惊讶的是, Martin Davis教授还证明了, 在不考虑物理限制的情况下(认为内存容量无限多, 每一个内存单元都可以存放任意大的数), 用L语言也可以编写出一个和NEMU类似的通用程序! 而且这个用L语言编写的通用程序的框架, 竟然还和NEMU中的 `cpu_exec()` 函数

如出一辙: 取指, 译码, 执行... 这其实并不是巧合, 而是模拟(Simulation)在计算机科学中的应用.

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

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

捕捉死循环(有点难度)

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

> 当用户程序陷入死循环时, 让用户程序暂停下来, 并输出相应的提示信息

你觉得应该如何实现? 如果你感到疑惑, 在互联网上搜索相关信息.

温馨提示

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

输入输出

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

设备的工作原理其实没什么神秘的. 你会在不久的将来在数字电路实验中看到键盘控制器模块和VGA控制器模块相关的verilog代码. 噢, 原来这些设备也一样是个数字电路! 事实上, 只要向设备发送一些有意义的数字信号, 设备就会按照这些信号的含义来工作. 让一些信号来指导设备如何工作, 这不就像"程序的指令指导CPU如何工作"一样吗? 恰恰就是这样! 设备也有自己的状态寄存器(相当于CPU的寄存器), 也有自己的功能部件(相当于CPU的运算器). 当然不同的设备有不同的功能部件, 例如键盘有一个把按键的模拟信号转换成扫描码的部件, 而VGA则有一个把像素颜色信息转换成显示器模拟信号的部件. 这些控制设备工作的信号称为"命令字", 可以理解成"设备的指令", 设备的工作就是负责接收命令字, 并进行译码和执行... 你已经知道CPU的工作方式, 这一切对你来说都太熟悉了.

既然设备是用来进行输入输出的, 所谓的访问设备, 说白了就是从设备获取数据(输入), 比如从键盘控制器获取按键扫描码, 或者是向设备发送数据(输出), 比如向显存写入图像的颜色信息. 但是, 如果万一用户没有敲键盘, 或者是用户想调整屏幕的分辨率, 怎么办呢? 这说明, 除了纯粹的数据读写之外, 我们还需要对设备进行控制: 比如需要获取键盘控制器的状态, 查看当前是否有按键被按下; 或者是需要有方式可以查询或设置VGA控制器的分辨率. 所以, 在程序看来, 访问设备 = 读出数据 + 写入数据 + 控制状态.

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除了进行计算之外, 还需要访问设备, 与其协作来完成不同的任务. 那么在CPU看来, 这些行为究竟意味着什么呢? 具体要从哪里读数据? 把数据写入到哪里? 如何查询/设置设备的状态? 一个最本质的问题是, CPU和设备之间的接口, 究竟是什么?

答案也许比你想象中的简单很多: 既然设备也有寄存器, 一种最简单的方法就是把设备的寄存器作为接口, 让CPU来访问这些寄存器. 比如CPU可以从/往设备的数据寄存器中读出/写入数据, 进行数据的输入输出; 可以从设备的状态寄存器中读出设备的状态, 询问设备是否忙碌; 或者往设备的命令寄存器中写入命令字, 来修改设备的状态.

那么, CPU要如何访问设备寄存器呢? 我们先来回顾一下CPU是如何访问CPU自己的寄存器的: 首先给这些寄存器编个号, 比如 `eax` 是 `0` , `ecx` 是 `1` ... 然后在指令中引用这些编号, 电路上会有相应的选择器来对相应的寄存器进行读写. 对设备寄存器的访问也是类似的: 我们也可以给设备中允许CPU访问的寄存器逐一编号, 然后通过指令来引用这些编号. 设备中可能会有一些私有寄存器, 它们是由设备自己维护的, 它们没有这样的编号, CPU不能直接访问它们.

这就是所谓的I/O编址方式, 因此这些编号也称为设备的地址. 常用的编址方式有两种.

端口I/O

一种I/O编址方式是端口映射I/O(port-mapped I/O), CPU使用专门的I/O指令对设备进行访问, 并把设备的地址称作端口号. 有了端口号以后, 在I/O指令中给出端口号, 就知道要访问哪一个设备寄存器了. 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.

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

```
movl $0x41, %al
movl $0x3f8, %edx
outb %al, (%dx)
```

上述代码把数据0x41传送到0x3f8号端口所对应的设备寄存器中. CPU执行上述代码后, 会将0x41这个数据传送到串口的一个寄存器中, 串口接收之后, 发现是要输出一个字符 `A`; 但对CPU来说, 它并不关心设备会怎么处理0x41这个数据, 只会老老实实地把0x41传送到0x3f8号端口. 事实上, 设备的API及其行为都会在相应的文档里面有清晰的定义, 在PA中我们无需了解这些细节, 只需要知道, 驱动开发者可以通过RTFM, 来编写相应程序来访问设备即可.

有没有一种熟悉的感觉?

> API, 行为, RTFM... 没错, 我们又再次看到了计算机系统设计的一个例子: 设备向CPU暴露设备寄存器的接口, 把设备内部的复杂行为(甚至包含一些模拟电路的特性)进行抽象, CPU只需要使用这一接口访问设备, 就可以实现期望的功能.
>
> 计算机系统处处蕴含抽象的思想, 只要理解其中的原理, 再加上RTFM的技能, 你就能掌握计算机系统的全部!

内存映射I/O

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

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

内存映射I/O的一个例子是NEMU中的物理地址区间 `[0x40000, 0x80000)`. 这段物理地址区间被映射到VGA内部的显存, 读写这段物理地址区间就相当于对读写VGA显存的数据. 例如

```
memset((void *)0x40000, 0, SCR_SIZE);
```

会将显存中一个屏幕大小的数据清零, 即往整个屏幕写入黑色像素, 作用相当于清屏. 可以看到, 内存映射I/O的编程模型和普通的编程完全一样: 程序员可以直接把I/O设备当做内存来访问. 这一特性也是深受驱动开发者的喜爱.

理解volatile关键字

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

```
void fun() {
  extern unsigned char _end;  // _end是什么?
  volatile unsigned char *p = &_end;
  *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 `p` 指向的地址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?

NEMU中的设备

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

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

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

- `nemu/src/device/io/port-io.c` 是对端口I/O的模拟. 其中 `PIO_t` 结构用于记录一个端口I/O映射的关系, 设备会初始化时会调用 `add_pio_map()` 函数来注册一个端口I/O映射关系, 返回该映射关系的I/O空间首地址. `pio_read_[l|w|b]()` 和 `pio_write_[l|w|b]()` 是面向CPU的端口I/O读写接口. 由于NEMU是单线程程序, 因此只能串行模拟整个计算机系统的工作, 每次进行I/O读写的时候, 才会调用设备提供的回调函数(callback), 更新设备的状态. 内存映射I/O的模拟和端口I/O的模拟比较相似, 只是内存映射I/O的读写并不是面向CPU的, 这一点会在下文进行说明.
- `nemu/src/device/device.c` 含有和SDL库相关的代码, NEMU使用SDL库来实现设备的模拟. `init_device()` 函数首先对以上四个设备进行初始化, 其中在初始化VGA时还会进行一些和SDL相关的初始化工作, 包括创建窗口, 设置显示模式等. 最后还会注册一个100Hz的定时器, 每隔0.01秒就会调用一次 `device_update()` 函数. `device_update()` 函数主要进行一些设备的模拟操作, 包括以50Hz的频率刷新屏幕, 以及检测是否有按键按下/释放. 需要说明的是, 代码中注册的定时器是虚拟定时器, 它只会在NEMU处于用户态的时候进行计时: 如果NEMU在 `ui_mainloop()` 中等待用户输入, 定时器将不会计时; 如果NEMU进行大量的输出, 定时器的计时将会变得缓慢. 因此除非你在进行调试, 否则尽量避免大量输出的情况, 从而影响定时器的工作.

我们提供的代码是模块化的, 要在NEMU中加入设备的功能, 你只需要在 `nemu/include/common.h` 中定义宏 `HAS_IOE` . 定义后, `init_device()` 函数会对设备进行初始化. 重新编译后, 你会看到运行NEMU时会弹出一个新窗口, 用于显示VGA的输出(见下文). 需要注意的是, 终端显示的提示符 `(nemu)` 仍然在等待用户输入, 此时窗口并未显示任何内容.

将设备访问抽象成IOE

设备访问的具体实现是机器相关的, 比如NEMU的VGA显存位于物理地址区间 `[0x40000, 0x80000)` , 但对 `native` 的程序来说, 这是一个不可访问的非法区间, 因此 `native` 程序需要通过别的方式来实现类似的功能. 自然地, 设备访问这一机器相关的功能, 应该归入AM中. 与TRM不同, 设备访问是为机器提供输入输出的功能, 因此我们把它们划入一类新的API, 名字叫IOE(I/O Extension).

要如何对不同机器的设备访问抽象成统一的API呢? 回想一下在程序看来, 访问设备其实想做什么: 访问设备 = 读出数据 + 写入数据 + 控制状态. 进一步的, 控制状态本质上也是读/写设备寄存器的操作, 所以访问设备 = 读/写操作.

对, 就是这么简单! 所以IOE为每个设备定义了如下的数据结构(见 `nexus-am/am/am.h`):

```
typedef struct _Device {
  uint32_t id;
  const char *name;
  size_t (*read) (uintptr_t reg, void *buf, size_t size);
  size_t (*write)(uintptr_t reg, void *buf, size_t size);
} _Device;
```

其中 `id` 是设备的唯一ID, `name` 是设备的名字(非必须), `read()/write()` 分别是设备读/写操作的实现, 用于从设备的 `reg` 寄存器中读出 `size` 字节的内容到缓冲区 `buf` 中, 或者往设备的 `reg` 寄存器中写入缓冲区 `buf` 中的 `size` 字节的内容. 需要注意的是, 这里的 `reg` 寄存器并不是上文讨论的设备寄存器, 因为设备寄存器的编号是机器相关的. 在IOE中, 我们希望采用一种机器无关的"抽象寄存器", 这个 `reg` 其实是一个功能编号, 我们约定在不同的机器中, 同一个功能编号的含义也是相同的, 这样就实现了设备寄存器的抽象.

另一个API是

```
_Device *_device(int n);
```

用于返回编号为 `n` 的设备的数据结构. 机器中的可用设备从 `1` 开始按顺序编号, 若不存在编号为 `n` 的设备, 则返回 `NULL` . 有了这个API, 程序就可以枚举机器中的每一个设备了. 最后一个API是 `_ioe_init()` , 它用于进行IOE相关的初始化操作.

`nexus-am/am/arch/x86-nemu/src/ioe.c` 中已经实现了时钟, 键盘和VGA的设备结构体, 但它们的读写函数(在 `nexus-am/am/arch/x86-nemu/src/devices/` 目录下定义)并未实现.

`nexus-am/am/amdev.h` 中定义了常见设备的ID:

```
#define _DEV_INPUT   0x0000ac02 // AM Virtual Input Device
#define _DEV_TIMER   0x0000ac03 // AM Virtual Timer
#define _DEV_VIDEO   0x0000ac04 // AM Virtual Video Controller
```

头文件中还定义了其它设备的ID, 但在PA中暂不使用.

头文件中还定义了这些常见设备的抽象寄存器编号. 这些定义是机器无关的, 每个机器在实现各自的IOE API时, 都需要遵循这些定义(约定). 这样, 我们就可以基于这套IOE的API, 来实现一些常用的输入输出功能了:

```
// 返回系统启动后经过的毫秒数
uint32_t uptime();
// 在`rtc`结构中返回当前时间, PA中不会用到
void get_timeofday(void *rtc);
// 返回按键的键盘码, 若无按键, 则返回`_KEY_NONE`
int read_key();
// 将`pixels`指定的矩形像素绘制到屏幕中以`(x, y)`和`(x+w, y+h)`两点连线为对角线的矩形区域
void draw_rect(uint32_t *pixels, int x, int y, int w, int h);
// 将之前的绘制内容同步到屏幕上
void draw_sync();
// 返回屏幕的宽度
int screen_width();
// 返回屏幕的高度
int screen_height();
```

经过了IOE的抽象, 上述功能的实现都可以是机器无关的, 因此可以将它们归入klib中(在 `nexum-am/libs/klib/src/io.c` 中定义), 供其它程序使用.

下面我们来逐一介绍NEMU中每个设备的功能.

串口

串口是最简单的输出设备. `nemu/src/device/serial.c` 模拟了串口的功能. 其大部分功能也被简化, 只保留了数据寄存器和状态寄存器. 串口初始化时会分别注册 `0x3F8` 和 `0x3FD` 处长度为1个字节的端口, 分别作为数据寄存器和状态寄存器. 由于NEMU串行模拟计算机系统的工作, 串口的状态寄存器可以一直处于空闲状态; 每当CPU往数据寄存器中写入数据时, 串口会将数据传送到主机的标准输出.

事实上, 在 `x86-nemu` 中, 我们之前提到的 `_putc()` 函数, 就是通过串口输出的. 然而AM却把 `_putc()` 放在TRM, 而不是IOE中, 这让人觉得有点奇怪. 的确, 可计算理论中提出的最原始的TRM并不包含输出的能力, 但对于一个现实的计算机系统来说, 输出是一个最基本的功能, 没有输出, 用户甚至无法知道程序具体在做什么. 因此在AM中, `_putc()` 的加入让TRM具有输出字符的能力, 被扩充后的TRM更靠近一个实用的机器, 而不再是只会计算的数学模型.

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

运行Hello World

实现 `in`, `out` 指令, 在它们的helper函数中分别调用 `pio_read_[l|w|b]()` 和 `pio_write_[l|w|b]()` 函数. 实现后, 在 `nexus-am/apps/hello/` 目录下键入

```
make run
```

在 `x86-nemu` 中运行基于AM的hello程序. 如果你的实现正确, 你将会看到程序往终端输出了10行 `Hello World!` (请注意不要让输出埋没在调试信息中).

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

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

实现**printf**

有了 `_putc()`, 我们就可以在klib中实现 `printf()` 了.

你之前已经实现了 `sprintf()` 了, 它和 `printf()` 的功能非常相似, 这意味着它们之间会有不少重复的代码. 你已经见识到Copy-Paste编程习惯的坏处了, 思考一下, 如何简洁地实现它们呢?

时钟

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

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

`nexus-am/am/amdev.h` 中为时钟定义了两个抽象寄存器:

- `_DEVREG_TIMER_UPTIME`, AM系统启动时间. 从中读出 `_UptimeReg` 结构体, `(hi << 32LL) | lo` 是系统启动的毫秒数.
- `_DEVREG_TIMER_DATE`, AM实时时钟(RTC). 从中读出 `_DateReg` 结构体, 包含年月日时分秒. PA中暂不使用.

实现IOE

在 `nexus-am/am/arch/x86-nemu/src/devices/timer.c` 中实现 `_DEVREG_TIMER_UPTIME` 的功能. 实现后, 在 `x86-nemu` 中运行 `timetest` 程序(在 `nexus-am/tests/timetest/` 目录下, 编译和运行方式请参考上文, 此后不再额外说明). 如果你的实现正确, 你将会看到程序每隔1秒往终端输出一句话. 由于没有实现 `_DEVREG_TIMER_DATE` , 测试总是输出2018年0月0日0时0分0秒, 这属于正常行为, 可以忽略.

看看NEMU跑多快

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

- dhrystone
- coremark
- microbench

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

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

```
make INPUT=TEST
```

先完成, 后完美 - 抑制住优化代码的冲动

计算机系统的设计过程可以概括成两件事:

1. 设计一个功能正确的完整系统 (先完成)
2. 在第1点的基础上, 让程序运行得更快 (后完美)

看到跑分之后, 你也许会忍不住去思考如何优化你的NEMU. 上述原则告诉你, 时机还没到. 一个原因是, 在整个系统完成之前, 你很难判断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, 最后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. 比如你可能在PA1中花时间去优化表达式求值的算法, 你可以以此作为一个编程练习, 但如果你的初衷是为了优化性能, 你的付出绝对是没有效果的: 你得输入多长的表达式才能让你明显感觉到新算法的性能优势?

此外, PA作为一个教学实验, 只要性能不是差得无法接受, 性能都不是你需要考虑的首要目标, 实现方案点到为止即可. 相比之下, 通过设计一个完整的系统来体会程序如何运行, 对你来说才是最重要的.

事实上, 除了计算机, "先完成, 后完美"的原则也适用于很多领域. 比如企业方案策划, 大家可以在一个完整但哪怕很简单的方案上迭代; 但如果一开始就想着把每一个点都做到完美, 最后很可能连一份完整的方案也拿不出手. 论文写作也一样, 哪怕是只有完整的小标题, 大家都可以去检查文章的整体框架有无逻辑漏洞; 相反, 就算文章配有再漂亮的实验数据, 在有漏洞的逻辑面前也无法自圆其说.

随着你参与越来越大的项目, 你会发现让完整的系统正确地跑起来, 会变得越来越难. 这时候, 遵循"先完成, 后完美"的原则就显得更重要了: 很多问题也许会等到项目趋于完整的时候才会暴露出来, 舍弃全局的完整而换来局部的完美, 大多时候只会南辕北辙.

键盘

键盘是最基本的输入设备. 一般键盘的工作方式如下: 当按下一个键的时候, 键盘将会发送该键的通码(make code); 当释放一个键的时候, 键盘将会发送该键的断码(break code). `nemu/src/device/keyboard.c` 模拟了i8042通用设备接口芯片的功能. 其大部分功能也被简化, 只保留了键盘接口. i8042初始化时会注册 `0x60` 处的端口作为数据寄存器. 每当用户敲下/释放按键时, 将会把相应的键盘码放入数据寄存器, CPU可以通过端口I/O访问数据寄存器, 获得键盘码; 当无按键可获取时, 将会返回 `_KEY_NONE` . 在AM中, 我们约定通码的值为 `断码 | 0x8000` .

如何检测多个键同时被按下?

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

`nexus-am/am/amdev.h` 中为键盘定义了一个抽象寄存器:

- `_DEVREG_INPUT_KBD` , AM键盘控制器. 从中读出 `_KbdReg` 结构体, `keydown = 1` 为按下按键, `keydown = 0` 为释放按键. `keycode` 为按键的断码, 没有按键时, `keycode` 为 `_KEY_NONE` .

实现IOE(2)

在 `nexus-am/am/arch/x86-nemu/src/devices/input.c` 中实现 `_DEVREG_INPUT_KBD` 的功能. 实现后, 在 `x86-nemu` 中运行 `keytest` 程序(在 `nexus-am/tests/keytest/` 目录下). 如果你的实现正确, 在程序运行时弹出的新窗口中按下按键, 你将会看到程序输出相应的按键信息, 包

括按键名, 键盘码, 以及按键状态.

VGA

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

神奇的调色板

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

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

`nexus-am/am/amdev.h` 中为VGA定义了两个抽象寄存器:

- `_DEVREG_VIDEO_INFO` , AM显示控制器信息. 从中读出 `_VideoInfoReg` 结构体, 其中 `width` 为屏幕宽度, `height` 为屏幕高度. 另外假设AM运行过程中, 屏幕大小不会发生变化.
- `_DEVREG_VIDEO_FBCTL` , AM帧缓冲控制器. 向其写入 `_FBCtlReg` 结构体, 向屏幕 `(x, y)` 坐标处绘制 `w*h` 的矩形图像. 图像像素按行优先方式存储在 `pixels` 中, 每个像素用32位整数以 `00RRGGBB` 的方式描述颜色.

实现IOE(3)

我们在讲义中并未介绍NEMU中的VGA设备如何将屏幕大小的信息暴露给CPU, 但框架代码中已经实现了相应的功能, 你需要RTFSC, 然后在 `nexus-am/am/arch/x86-nemu/src/devices/video.c` 中实现 `_DEVREG_VIDEO_INFO` 的功能. 这一任务可以与下文的"添加内存映射I/O"任务一同测试.

添加内存映射I/O

在NEMU中的 `paddr_read()` 和 `paddr_write()` 中加入对内存映射I/O的判断. 通过 `is_mmio()` 函数判断一个物理地址是否被映射到I/O空间, 如果是, `is_mmio()` 会返回映射号, 否则返回 `-1` . 内存映射I/O的访问需要调用 `mmio_read()` 或 `mmio_write()` , 调用时需要提供映射号. 如果不是内存映射I/O的访问, 就访问 `pmem` .

实现后, `_DEVREG_VIDEO_FBCTL` 中添加如下测试代码:

```
--- nexus-am/am/arch/x86-nemu/src/devices/video.c
+++ nexus-am/am/arch/x86-nemu/src/devices/video.c
@@ -20,6 +20,9 @@
 size_t video_write(uintptr_t reg, void *buf, size_t size) {
   switch (reg) {
     case _DEVREG_VIDEO_FBCTL: {
       _FBCtlReg *ctl = (_FBCtlReg *)buf;
+      int i;
+      int size = screen_width() * screen_height();
+      for (i = 0; i < size; i ++) fb[i] = i;

       if (ctl->sync) {
```

然后在 `x86-nemu` 中运行 `videotest` 程序(在 `nexus-am/tests/videotest/` 目录下). 如果 `_DEVREG_VIDEO_INFO` 和内存映射I/O均实现正确, 你会看到新窗口中输出了全屏的颜色信息.

实现IOE(4)

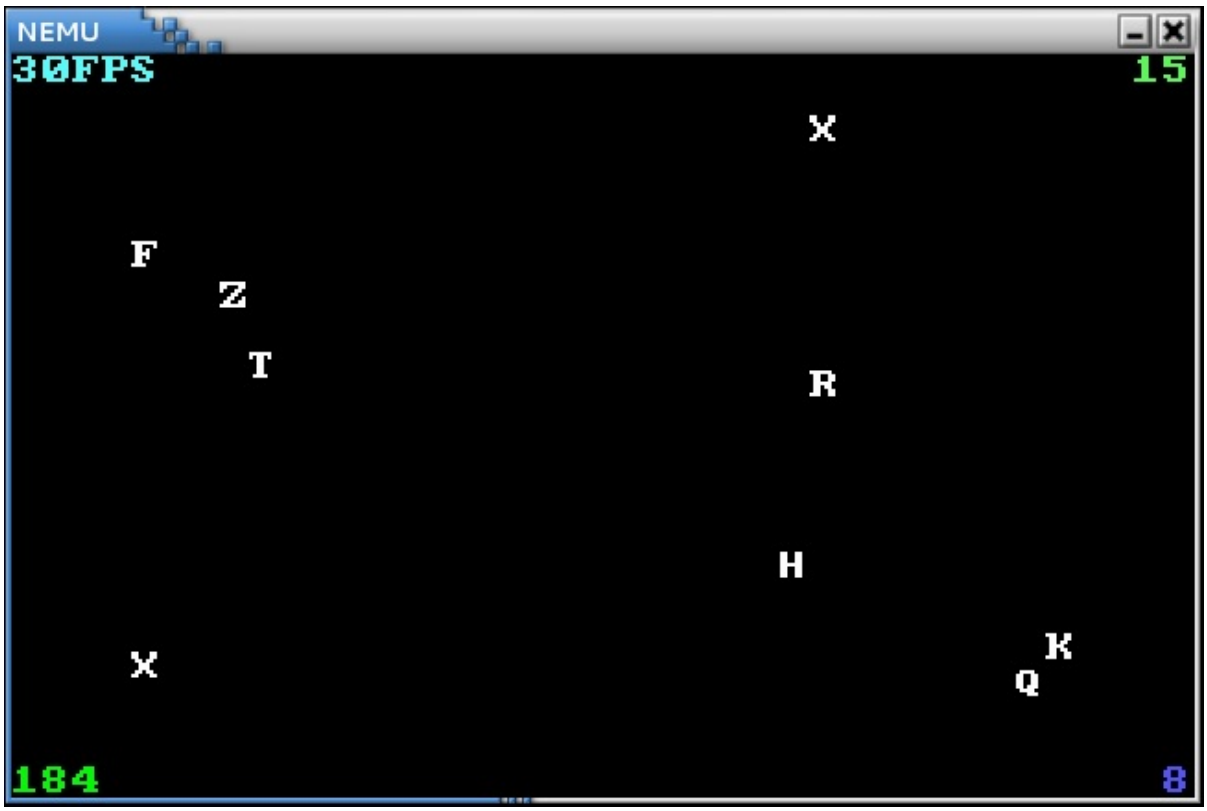
事实上, 刚才输出的颜色信息并不是 `videotest` 输出的画面, 这是因为上述测试代码并未正确实现 `_DEVREG_VIDEO_FBCTL` 的功能. 你需要正确地实现 `_DEVREG_VIDEO_FBCTL` 的功能. 实现后, 在 `x86-nemu` 中重新运行 `videotest` . 如果你的实现正确, 你将会看到新窗口中输出了相应的动画效果.

可展示的计算机系统

展示你的计算机系统

完整实现IOE后, 我们就可以运行一些酷炫的程序了:

- 幻灯片播放(在 `nexus-am/apps/slider/` 目录下). 程序将每隔5秒切换 `images/` 目录下的图片.
- 打字小游戏(在 `nexus-am/apps/typing/` 目录下). 打字小游戏来源于2013年NJUCS oslab0的框架代码. 为了配合移植, 代码的结构做了少量调整, 同时去掉了和显存优化相关的部分, 并去掉了浮点数.



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

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

必答题

你需要在实验报告中用自己的语言, 尽可能详细地回答下列问题.

- 编译与链接 在 `nemu/include/cpu/rtl.h` 中, 你会看到由 `static inline` 开头定义的各种RTL指令函数. 选择其中一个函数, 分别尝试去掉 `static` , 去掉 `inline` 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?
- 编译与链接
  1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问重新编译后的NEMU含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?

2. 添加上题中的代码后, 再在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问此时的NEMU含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.
3. 修改添加的代码, 为两处 `dummy` 变量进行初始化: `volatile static int dummy = 0;` 然后重新编译NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)

- 了解Makefile 请描述你在 `nemu/` 目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `nemu/build/nemu` . (这个问题包括两个方面: `Makefile` 的工作方式和编译链接的过程.) 关于 `Makefile` 工作方式的提示:
  - `Makefile` 中使用了变量, 包含文件等特性
  - `Makefile` 运用并重写了一些implicit rules
  - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
  - RTFM

温馨提示

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

PA3 - 穿越时空的旅程: 批处理系统

世界诞生的故事 - 第三章

冯诺依曼计算机果然功力深厚, 竟然能向冷冰冰的门电路赋予新的生命. 为了让计算机自动运行一组程序, 先驱对计算机进行了改进.

代码管理

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: 30小时

截止时间: 本次实验的阶段性安排如下:

- task PA3.1: 实现自陷操作 `_yield()` 及其过程 - 2018/11/18 23:59:59
- task PA3.2: 实现用户程序的加载和系统调用, 支撑TRM程序的运行 - 2018/11/25 23:59:59
- task PA3.3: 运行仙剑奇侠传并展示批处理系统, 提交完整的实验报告 - 2018/12/09 23:59:59

提交说明: 见这里

批处理系统

我们在PA2中已经实现了一个冯诺依曼计算机系统, 并且已经在AM上把打字游戏运行起来了. 有了IOE, 几乎能把各种小游戏移植到AM上来运行了. 这些小游戏在计算机上的运行模式有一个特点, 它们会独占整个计算机系统: 我们可以在NEMU上一直玩打字游戏, 不想玩的时候, 我们就会退出NEMU, 然后重新运行超级玛丽来玩.

事实上, 早期的计算机就是这样工作的: 系统管理员给计算机加载一个特定的程序(其实是上古时期的打孔卡片), 计算机就会一直执行这个程序, 直到程序结束或者是管理员手动终止, 然后再由管理员来手动加载下一个程序. 当年的程序也远远不如你玩的超级玛丽这么酷炫, 大多都是一些科学计算和物理建模的任务(比如弹道轨迹计算).

后来人们就想, 每次都要管理员来手动加载新的程序, 这太麻烦了. 能不能让管理员事先准备好一组程序, 让计算机执行完一个程序之后, 就自动执行下一个程序呢? 这就是批处理系统的思想, 有了批处理系统之后, 就可以解放管理员的双手了. 而批处理系统的关键, 就是要有一个后台程序, 当一个前台程序执行结束的时候, 后台程序就会自动加载一个新的前台程序来执行.

这样的一个后台程序, 其实就是操作系统. 对, 你没有听错, 这个听上去好像什么都没做的后台程序, 就是操作系统! 说起操作系统, 也许你会马上想到安装包都有几个GB的Windows. 但实际上, 历史上最早投入使用的操作系统GM-NAA I/O在1956年就诞生了, 而它的一个主要任务, 就是上文提到的"自动加载新程序".

什么是操作系统? (建议二周目思考)

> 这可是个大问题, 我们也鼓励你学习完操作系统课程之后再回来重新审视它.

最简单的操作系统

那么, 我们也来介绍一下在PA中使用的最简单的操作系统吧, 它的名字叫Nanos-lite. Nanos-lite是南京大学操作系统Nanos的裁剪版, 是一个为PA量身订造的操作系统. 通过编写Nanos-lite的代码, 你将会认识到操作系统是如何使用机器提供的接口, 来支撑程序的运行的. 这也符合PA的终极目标.

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

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

需要提醒的是, Nanos-lite是运行在AM之上, AM的API在Nanos-lite中都是可用的. 虽然操作系统对我们来说是一个特殊的概念, 但在AM看来, 它只是一个使用AM API的普通C程序而已, 和超级玛丽没什么区别. 同时, 你会再次体会到AM的好处: Nanos-lite的实现可以是机器无关的, 这意味着, 你可以像开发klib那样, 在 `native` 上调试你编写的Nanos-lite.

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

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

1. 通过 `Log()` 输出hello信息和编译时间. 需要说明的是, Nanos-lite中定义的 `Log()` 宏并不是NEMU中定义的 `Log()` 宏. Nanos-lite和NEMU是两个独立的项目, 它们的代码不会相互影响, 你在阅读代码的时候需要注意这一点. 在Nanos-lite中, `Log()` 宏通过你在 `klib` 中编写的 `printf()` 输出, 最终会调用TRM的 `_putc()` .
2. 初始化ramdisk. 一般来说, 程序应该存放在永久存储的介质中(比如磁盘). 但要在NEMU中对磁盘的模拟略显复杂, 因此先把Nanos-lite中的一段内存作为磁盘来使用. 这样的磁盘有一个专门的名字, 叫ramdisk.
3. 调用 `init_device()` 对设备进行一些初始化操作. 目前 `init_device()` 会直接调用 `_ioe_init()` .
4. `init_fs()` 和 `init_proc()` , 分别用于初始化文件系统和创建进程, 目前均为空函数, 可以忽略它们.

5. 调用 `panic()` 结束Nanos-lite的运行.

更新Makefile

> 我们在2018/11/03 16:00:00对框架代码的 `nanos-lite/Makefile` 进行了更新. 如果你在此时间之前获得框架代码, 请根据这里手动更新该文件; 如果你在此时间之后获得框架代码, 你不需要进行额外的操作.

由于Nanos-lite本质上也是一个AM程序, 我们可以采用相同的方式来编译/运行Nanos-lite. 在 `nanos-lite/` 目录下执行

```
make ARCH=x86-nemu update
make ARCH=x86-nemu run
```

即可. 另外如前文所说, 你也可以将Nanos-lite编译到 `native` 上并运行, 来帮助你进行调试.

框架代码提供的这个操作系统还真的什么都没做! 回顾历史, 要实现一个最简单的操作系统, 就要实现以下两点功能:

- 用户程序执行结束之后, 可以跳转到操作系统的代码继续执行
- 操作系统可以加载一个新的用户程序来执行

来自操作系统的新需求

仔细思考, 我们就会发现, 上述两点功能中其实蕴含着一个新的需求: 程序之间的执行流切换. 我们知道函数调用一般是在一个程序内部发生的(动态链接库除外), 属于程序内部的执行流切换, 使用call指令即可实现. 而上述两点需求需要在操作系统和用户程序之间进行执行流的切换. 不过, 执行流切换的本质, 也只是把 `%eip` 从一个值修改成另一个值而已(黑客眼中就是这么理解的). 那么, 我们能否也使用call指令来实现程序之间的执行流切换呢?

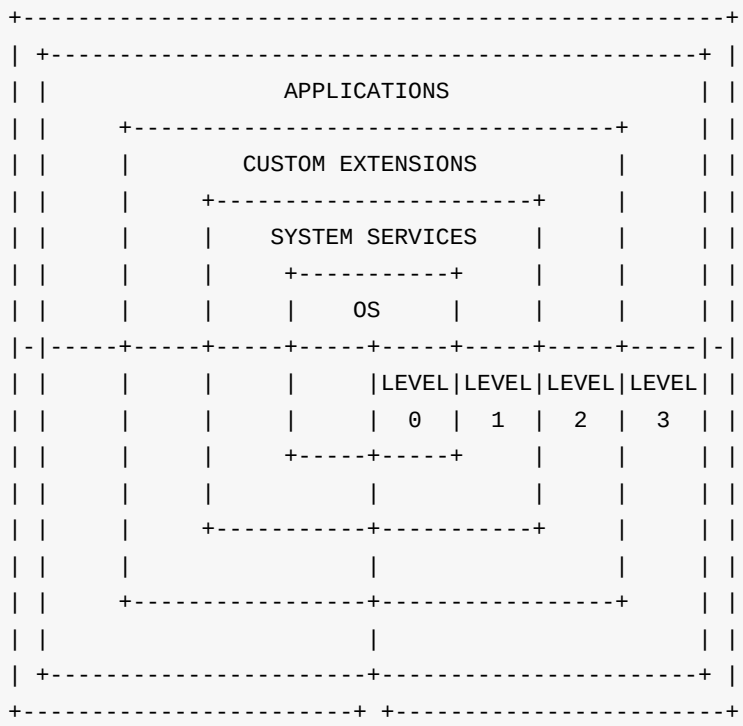
也许在GM-NAA I/O诞生的那个年代, 大家说不定还真是这样做的: 操作系统就是一个库函数, 用户程序退出的时候, 调用一下这个特殊的库函数就可以了, 就像我们在AM的程序中调用 `_halt()` 一样. 不过, 后来人们逐渐认识到, 操作系统和其它用户程序还是不太一样的: 一个用户程序出错了, 操作系统可以运行下一个用户程序; 但如果操作系统崩溃了, 整个计算机系统都将无法工作. 所以, 我们还是希望能把操作系统保护起来, 尽量保证它可以正确工作.

在这个需求面前, 用call指令来进行操作系统和用户进程之间的切换就显得太随意了. 操作系统的本质也是一个程序, 也是由函数构成的, 但无论用户程序是无意还是有心, 我们都不希望它可以把执行流切换到操作系统中的任意函数. 我们所希望的, 是一种可以限制入口的执行流切换方式, 显然, 这种方式是无法通过程序代码来实现的.

等级森严的制度

为了阻止程序将执行流切换到操作系统的任意位置, 硬件中逐渐出现保护机制相关的功能, 比如i386中引入了保护模式(protected mode)和特权级(privilege level)的概念. 简单地说, 只有高特权级的程序才能去执行一些系统级别的操作, 如果一个特权级低的程序尝试执行它没有权限执行的操作, CPU将会抛出一个异常信号, 来阻止这一非法行为的发生. 一般来说, 最适合担任系统管理员的角色就是操作系统了, 它拥有最高的特权级, 可以执行所有操作; 而除非经过允许, 运行在操作系统上的用户程序一般都处于最低的特权级, 如果它试图破坏社会的和谐, 它将会被判"死刑".

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



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

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

一次数据的访问操作是合法的, 当且仅当

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

两式同时成立, 注意这里的 `>=` 是数值上的(numerically greater).

<1>式表示请求者有权限访问目标数据, <2>式表示当前进程也有权限访问目标数据. 如果违反了上述其中一式, 此次操作将会被判定为非法操作, CPU将会抛出异常信号, 并跳转到一个和操作系统约定好的内存位置, 交由操作系统进行后续处理.

对RPL的补充

你可能会觉得RPL十分令人费解, 我们先举一个生活上的例子.

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

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

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

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

分崩离析的秩序

特权级保护是现代计算机系统的一个核心机制, 但并不是有了这一等级森严的制度就在高枕无忧了, 黑客们总是会绞尽脑汁去试探这一制度的边界. 最近席卷计算机领域的, 就要数2018年1月爆出的Meltdown和Spectre这两个大名鼎鼎的硬件漏洞了. 这两个史诗级别的漏洞之所以震惊全世界, 是因为它们打破了特权级的边界: 恶意程序在特定的条件下可以以极高的效率窃取操作系统的信息. Intel的芯片被爆都有Meltdown漏洞, 而Spectre漏洞则是危害着所有架构的芯片, 无一幸免, 可谓目前为止体系结构历史上影响最大的两个漏洞了. 如果你执行`cat /proc/cpuinfo`, 你应该会在`bugs`信息中看到这两个漏洞的影子.

Meltdown和Spectre给过去那些一味追求性能的芯片设计师敲响了警钟: 没有安全, 芯片跑得再快, 也是徒然. 有趣的是, 直接为这场闹剧买单的, 竟然是各大云计算平台的工程师们: 漏洞被爆出的那一周时间, 阿里云和微软Azure的工程师连续通宵加班, 想尽办法给云平台打上安全补丁, 以避免客户的数据被恶意窃取.

不过作为教学实验, 安全这个话题离PA还是太遥远了, 甚至性能也不是PA的主要目标. 这个例子想说的是, 真实的计算机系统非常复杂, 远远没到完美的程度, 这些漏洞的出现从某种程度上也说明了, 复杂程度已经到了人们没法一下子想明白每个模块之间的相互影响了; 但计算机背后的原理都是一脉相承的, 在一个小而精的教学系统中理解这些原理, 然后去理解, 去改进真实的系统, 这也是做PA的一种宝贵的收获.

穿越时空的旅程

有了强大的硬件保护机制, 用户程序将无法把执行流切换到操作系统的任意代码了. 但为了实现最简单的操作系统, 硬件还需要提供一种可以限制入口的执行流切换方式. 这种方式就是自陷指令, 程序执行自陷指令之后, 就会陷入到操作系统预先设置好的跳转目标.

i386提供 `int` 指令作为自陷指令, 它的工作过程十分特别, 整个过程是由i386中断机制支撑的. i386中断机制不具体区分CPU异常和自陷, 甚至是将在PA4最后介绍的硬件中断, 而是对它们进行统一的处理. 在i386中, 上述跳转目标是通过门描述符(Gate Descriptor)来指示的. 门描述符是一个8字节的结构体, 里面包含着不少细节的信息, 我们在NEMU中简化了门描述符的结构, 只保留存在位P和偏移量OFFSET:

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

P位来用表示这一个门描述符是否有效, OFFSET用来指示跳转目标. 有了门描述符, 用户程序就只能跳转到门描述符中OFFSET所指定的位置, 再也不能随心所欲地跳转到操作系统的任意代码了.

为了方便管理各个门描述符, i386把内存中的某一段数据专门解释成一个数组, 叫IDT(Interrupt Descriptor Table, 中断描述符表), 数组的一个元素就是一个门描述符. 为了从数组中找到一个门描述符, 我们还需要一个索引. 对于CPU异常来说, 这个索引由CPU内部产生(例如除零异常为0号异常),或者由 `int` 指令给出(例如 `int $0x80`). 最后, 为了在内存中找到IDT, i386使用IDTR寄存器来存放IDT的首地址和长度. 操作系统的代码事先把IDT准备好, 然后执行一条特殊的指令 `lidt` , 来在IDTR中设置好IDT的首地址和长度, 这一中断处理机制就可以正常工作了. 现在是万事俱备, 等到程序执行自陷指令或者触发异常的时候, CPU就会按照设定好的IDT跳转到目标地址:

不过, 我们将来还是有可能需要返回到程序的当前状态来继续执行的, 比如通过 `int3` 触发的断点异常. 这意味着, 我们需要在进行异常处理之前保存好程序当前的状态. 于是, 触发异常后硬件的处理如下:

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

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

特殊的原因? (建议二周目思考)

> 当前的EFLAGS, CS, EIP必须由硬件来保存吗? 能否通过软件来保存? 为什么?

由于IDT中的目标地址是硬件和操作系统约定好的, 接下来的处理过程将会由操作系统来接管, 操作系统将视情况决定是否终止当前程序的运行(例如触发段错误的程序将会被杀死). 若决定不杀死当前程序, 等到异常处理结束之后, 就根据之前保存的信息恢复程序的状态. `iret` 指令用于从异常处理过程中返回, 它将栈顶的三个元素来依次解释成EIP, CS, EFLAGS, 并恢复它们.

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

将上下文管理抽象成CTE

我们刚才提到了程序的状态, 在操作系统中有一个等价的术语, 叫"上下文". 因此, 硬件提供的上述在操作系统和用户程序之间切换执行流的功能, 在操作系统看来, 都可以划入上下文管理的一部分.

与IOE一样, 上下文管理的具体实现也是机器相关的: 比如MIPS中会通过 `syscall` 指令来进行自陷, `native` 中也可以通过一些神奇的库函数来模拟相应的功能, 而上下文的具体内容, 在不同的机器上也显然不一样(比如寄存器就已经不一样了). 于是, 我们可以将上下文管理的功能划入到AM的一类新的API中, 名字叫CTE(ConText Extension).

接下来的问题是, 如何将不同机器的上下文管理功能抽象成统一的API呢? 换句话说, 我们需要思考, 操作系统的处理过程其实需要哪些信息?

- 首先当然是引发这次执行流切换的原因, 是程序除0, 非法指令, 还是触发断点, 又或者是程序自愿陷入操作系统? 根据不同的原因, 操作系统都会进行不同的处理.
- 然后就是程序的上下文了, 在处理过程中, 操作系统可能会读出上下文中的一些寄存器, 根据它们的信息来进行进一步的处理. 例如操作系统读出EIP所指向的非法指令, 看看其是否能被模拟执行. 事实上, 通过这些上下文, 操作系统还能实现一些神奇的功能, 你将会在PA4中了解更详细的信息.

用软件模拟指令

> 在一些嵌入式场景中, 处理器对低功耗的要求非常严格, 很多时候都会去掉浮点处理单元FPU. 这时候如果软件要指令一条浮点指令, 处理器就会抛出一个非法指令的异常. 有了异常处理的机制, 我们就可以在异常处理的程序中模拟这条非法指令的执行了, 原理和PA2中的指令执行过程非常类似. 在不带FPU的MIPS, ARM和RISC-V处理器中, 都可以通过这种方式来执行浮点指令.

所以, 我们只要把这两点信息抽象成一种统一的表示方式, 就可以定义出CTE的API了. 对于切换原因, 我们只需要定义一种统一的描述原因的方式即可. CTE定义了名为"事件"的如下数据结构(见 `nexus-am/am/am.h`):

```
typedef struct _Event {
  int event;
  uintptr_t cause, ref;
  const char *msg;
} _Event;
```

其中 `event` 表示事件编号, `cause` 和 `ref` 是一些描述事件的补充信息, `msg` 是事件信息字符串. 在PA中, 目前只会用到 `event` . 然后, 我们只要定义一些统一的事件编号(见 `nexus-am/am/am.h`), 让每个机器在实现各自的CTE API时, 都统一通过上述结构体来描述执行流切换的原因, 就可以实现切换原因的抽象了.

对于上下文, 我们只能将描述上下文的结构体类型名统一成 `_Context` , 至于其中的具体内容, 就无法进一步进行抽象了. 这主要是因为不同机器之间上下文信息的差异过大, 比如MIPS有32个通用寄存器, 就从这一点来看, MIPS和x86的 `_Context` 注定是无法抽象成完全统一的结构的. 所以在AM中, `_Context` 的具体成员也是由不同的机器自己定义的, 比如 `x86-nemu` 的 `_Context` 结构体在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中定义. 因此, 在操作系统中, 对 `_Context` 成员的直接引用, 都属于机器相关的行为, 会损坏操作系统的可移植性. 不过大多数情况下, 操作系统并不需要单独访问 `_Context` 结构中的成员. 必要的时候, CTE也可以提供一些统一的接口, 来让操作系统通过这些接口来访问, 从而保证操作系统的相关代码与机器无关.

最后还有另外两个统一的API:

- `int _cte_init(_Context* (*handler)(_Event ev, _Context *ctx))` 用于进行CTE相关的初始化操作. 其中它还接受一个来自操作系统的事件处理回调函数的指针, 当发生事件时, CTE将会把事件和相关的上下文作为参数, 来调用这个回调函数, 交由操作系统进行后续处理.
- `void _yield()` 用于进行自陷操作, 会触发一个编号为 `_EVENT_YIELD` 事件. 在 `x86-nemu` 中, 我们约定自陷操作通过 `int $0x81` 触发.

CTE中还有其它的API, 目前不使用, 故暂不介绍它们.

接下来, 我们将尝试在Nanos-lite中触发一次自陷操作, 来梳理过程中的细节.

准备IDT

首先是准备一个有意义的IDT, 将来切换执行流时才能跳转到正确的目标地址. 这显然是机器相关的行为(IDT是x86特有的结构), 因此我们把这一行为放入CTE中, 而不是让Nanos-lite直接来准备IDT. 你需要在 `nanos-lite/include/common.h` 中定义宏 `HAS_CTE` , 这样以后, Nanos-lite会多进行一项初始化工作: 调用 `init_irq()` 函数, 这最终会调用位于 `nexus-am/am/arch/x86-nemu/src/cte.c` 中的 `_cte_init()` 函数. `_cte_init()` 函数会做两件事情, 第一件就是初始化IDT:

1. 代码定义了一个结构体数组 `idt`, 它的每一项是一个门描述符结构体
2. 在相应的数组元素中填写有意义的门描述符, 例如编号为 `0x81` 的门描述符中就包含自陷操作的入口地址. 需要注意的是, 框架代码中还是填写了完整的门描述符(包括上文中提到的don't care的域), 这主要是为了在QEMU中进行DiffTest时也能跳转到正确的入口地址. QEMU实现了完整的中断机制, 如果只填写简化版的门描述符, 就无法在QEMU中正确运行. 但我们无需了解其中的细节, 只需要知道代码已经填写了正确的门描述符即可.
3. 通过 `lidt` 指令在IDTR中设置 `idt` 的首地址和长度

`_cte_init()` 函数做的第二件事是注册一个事件处理回调函数, 这个回调函数由Nanos-lite提供, 更多信息会在下文进行介绍.

触发自陷操作

为了测试是否已经准备正确的IDT, 我们还需要真正触发一次自陷操作, 看是否正确地跳转到目标地址. 定义了宏 `HAS_CTE` 后, Nanos-lite会在 `panic()` 前调用 `_yield()` 来触发自陷操作. 为了支撑这次自陷操作, 你需要在NEMU中实现 `raise_intr()` 函数(在 `nemu/src/cpu/intr.c` 中定义)来模拟上文提到的i386中断机制的处理过程:

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

}
```

需要注意的是:

- PA不涉及特权级的切换, RTFM的时候你不需要关心和特权级切换相关的内容.
- 通过IDTR中的地址对IDT进行索引的时候, 需要使用 `vaddr_read()`.
- PA中不实现分段机制, 没有CS寄存器的概念. 但为了在QEMU中顺利进行DiffTest, 我们还是需要在cpu结构体中添加一个CS寄存器, 并在 `restart()` 函数中将其初始化为 `8`.
- 由于i386中断机制需要对EFLAGS进行压栈, 为了配合DiffTest, 我们还需要在 `restart()` 函数中将EFLAGS初始化为 `0x2`.
- 执行 `int` 指令后保存的EIP指向的是 `int` 指令的下一条指令, 这有点像函数调用, 具体细节请RTFM.
- 你需要在 `int` 指令的helper函数中调用 `raise_intr()`, 而不要把中断机制的代码放在 `int` 指令的helper函数中实现, 因为在后面我们会再次用到 `raise_intr()` 函数.

实现i386中断机制

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

实现正确后, 重新运行Nanos-lite, 如果你看到在 `vectrap()` (在 `nexum-am/am/arch/x86-nemu/src/trap.S` 中定义)附近触发了未实现指令, 说明你实现的i386中断机制已经跳转到正确的自陷入口.

保存上下文

成功跳转到自陷入口函数 `vectrap()` 之后, 我们就要在软件上开始真正的异常处理过程了. 但是, 进行异常处理的时候不可避免地需要用到通用寄存器, 然而看看现在的通用寄存器, 里面存放的都是执行流切换之前的内容. 这些内容也是上下文的一部分, 如果不保存就覆盖它们, 将来就无法恢复这一上下文了. 但硬件并不负责保存它们, 因此需要通过软件代码来保存它们的值. i386提供了 `pusha` 指令, 用于把通用寄存器的值压栈.

`vectrap()` 会压入错误码和异常号 `#irq` , 然后跳转到 `asm_trap()` . 在 `asm_trap()` 中, 代码将会把当前的通用寄存器保存到栈上, 并通过一条 `pushl $0` 指令在栈上占位, 它是为PA4准备的, 目前可以忽略它. 这些内容连同之前保存的错误码, `#irq` , 以及硬件保存的EFLAGS, CS, EIP, 形成了完整的上下文, 将来恢复上下文的时候就靠它了.

对比异常处理与函数调用

我们知道进行函数调用的时候也需要保存调用者的状态: 返回地址, 以及calling convention中需要调用者保存的寄存器. 而CTE在保存上下文的时候却要保存更多的信息. 尝试对比它们, 并思考两者保存信息不同是什么原因造成的.

注意到上下文是在栈上构造的. 接下来代码将会把当前的 `%esp` 压栈, 并调用C函数 `irq_handle()` (在 `nexus-am/am/arch/x86-nemu/src/cte.c` 中定义).

诡异的代码

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

重新组织_Context结构体

你的任务如下:

- 实现 `pusha` 指令, 你需要注意压栈的顺序, 详情请RTFM.
- 理解上下文形成的过程, 然后重新组织 `nexus-am/am/arch/x86-nemu/include/arch.h` 中定义的 `_Context` 结构体的成员, 使得这些成员的定义顺序和 `nexus-am/am/arch/x86-nemu/src/trap.S` 中构造的上下文保持一致.

实现之后, 你可以在 `irq_handle()` 中通过 `printf` 输出上下文 `tf` 的内容, 然后通过简易调试器观察触发自陷时的寄存器状态, 从而检查你的 `_Context` 实现是否正确.

事件分发

`irq_handle()` 的代码会把执行流切换的原因打包成事件, 然后调用在 `_cte_init()` 中注册的事件处理回调函数, 将事件交给Nanos-lite来处理. 在Nanos-lite中, 这一回调函数是 `nanos-lite/src/irq.c` 中的 `do_event()` 函数. `do_event()` 函数会根据事件类型再次进行分发. 不过我们在这里会触发一个未处理的1号事件:

```
[src/irq.c,5,do_event] {kernel} system panic: Unhandled event ID = 1
```

这是因为CTE的 `irq_handle()` 函数并未正确识别出自陷事件. 根据 `_yield()` 的定义, `irq_handle()` 函数需要将自陷事件打包成编号为 `_EVENT_YIELD` 的事件.

实现正确的事件分发

你需要:

1. 在 `irq_handle()` 中通过异常号识别出自陷事件, 并打包成编号为 `_EVENT_YIELD` 的事件.
2. 在 `do_event()` 中识别出自陷事件 `_EVENT_YIELD`, 然后输出一句话即可, 无需进行其它操作.

重新运行Nanos-lite, 如果你的实现正确, 你会看到识别到自陷事件之后输出的信息,

恢复上下文

代码将会一路返回到 `trap.S` 的 `asm_trap()` 中, 接下来的事情就是恢复程序的上下文. `asm_trap()` 将根据之前保存的上下文内容, 恢复通用寄存器, 并直接弹出一些不再需要的信息, 最后执行 `iret` 指令, 返回到Nanos-lite触发自陷的代码位置, 然后继续执行. 在它看来, 这次时空之旅就好像没有发生过一样.

恢复上下文

你需要实现 `popa` 和 `iret` 指令. 重新运行Nanos-lite, 如果你的实现正确, 你会看到在 `do_event()` 中输出的信息, 并且最后仍然触发了 `main()` 函数末尾设置的 `panic()`.

温馨提示

PA3阶段1到此结束.

用户程序和系统调用

有了自陷指令, 用户程序就可以将执行流切换到操作系统指定的入口了. 现在我们来解决如何加载用户程序的问题.

加载第一个用户程序

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

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

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

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

其中, `navy-apps/libs/libc` 中是一个名为Newlib的项目, 它是一个专门为嵌入式系统提供的C库, 库中的函数对运行时环境的要求极低. 这对Nanos-lite来说是非常友好的, 我们不需要为了配合C库而在Nanos-lite中实现额外的功能. 用户程序的入口位于 `navy-apps/libs/libc/src/platform/crt0.c` 中的 `_start()` 函数, 它会调用用户程序的 `main()` 函数, 从 `main()` 函数返回后会调用 `exit()` 结束运行.

我们要在Nanos-lite上运行的第一个用户程序是 `navy-apps/tests/dummy/dummy.c` . 为了避免和Nanos-lite的内容产生冲突, 我们约定目前用户程序需要被链接到内存位置 `0x4000000` 处, Navy-apps已经设置好了相应的选项(见 `navy-apps/Makefile.compile` 中的 `LDFLAGS` 变量).

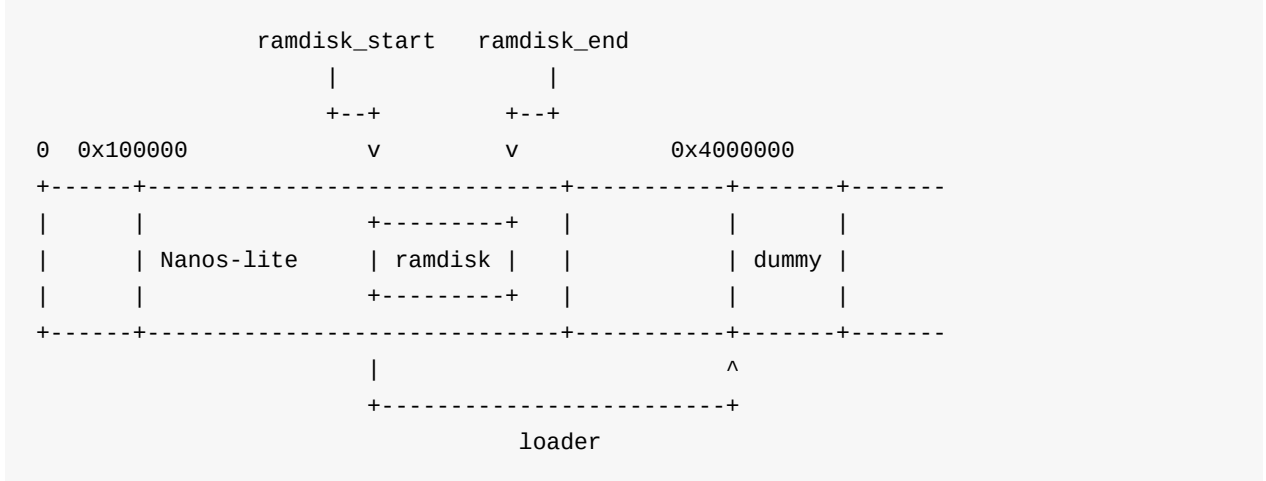
在 `nanos-lite/` 目录下执行

```
make ARCH=x86-nemu update
```

将会生成ramdisk镜像文件 `ramdisk.img` , 并包含进Nanos-lite成为其中的一部分(在 `nanos-lite/src/initrd.S` 中实现). 执行上述命令时会编译Navy-apps项目中的Newlib, 过程中会出现较多warning, 我们可以忽略它们. 现在的ramdisk十分简单, 它只有一个文件, 就是我们将要加载的用户程序 `dummy` , 这其实已经回答了上述第一个问题: 可执行文件位于ramdisk偏移为0处, 访问它就可以得到用户程序的第一个字节.

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

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



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

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

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

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

真实操作系统中的loader远比我们目前在Nanos-lite中实现的loader要复杂. 事实上, Nanos-lite的loader设计其实也向我们展现出了程序的最为原始的状态: 比特串! 加载程序其实就是把这一毫不起眼的比特串放置在正确的位置, 但这其中又折射出"存储程序"的划时代思想: 当操作系统将控制权交给它的时候, 计算机把它解释成指令并逐条执行. loader让计算机的生命周期突破程序的边界: 一个程序结束并不意味着计算机停止工作, 计算机将终其一生履行执行程序的使命.

实现loader

你需要在Nanos-lite中实现loader的功能, 来把用户程序加载到正确的内存位置, 然后执行用户程序. `loader()` 函数在 `nanos-lite/src/loader.c` 中定义, 其中的 `pcb` 参数目前暂不使用, 可以忽略, 而因为ramdisk中目前只有一个文件, `filename` 参数也可以忽略.

实现后, 在 `init_proc()` 中调用 `naive_uload(NULL, NULL)`, 它会调用你实现的loader来加载第一个用户程序, 然后跳转到用户程序中执行. 如果你的实现正确, 你会看到执行 `dummy` 程序时在Nanos-lite中触发了一个未处理的1号事件. 这说明loader已经成功加载dummy, 并且成功地跳转到dummy中执行了. 关于未处理的事件, 我们会在下文进行说明.

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

```
make ARCH=x86-nemu update
```

来更新Nanos-lite中的ramdisk内容, 然后重新编译Nanos-lite来使用最新的ramdisk.

将Nanos-lite编译到native

你可以在 `native` 上测试你的Nanos-lite实现是否正确. 但是需要注意, 你需要同时生成一份与 `native` 相对应的ramdisk:

```
make ARCH=native update
make ARCH=native run
```

让 `x86-nemu` 版本的Nanos-lite使用 `native` 版本的ramdisk, 或者让 `native` 版本的Nanos-lite使用 `x86-nemu` 版本的ramdisk, 都会使得用户程序无法正确运行在Nanos-lite上.

操作系统的运行时环境

加载程序之后, 我们就来谈谈程序的运行. 回顾PA2, 我们已经知道, 程序的运行需要运行时环境的支撑. 而操作系统希望加载并运行程序, 自然有责任来提供运行时环境的功能. 在PA2中, 我们根据具体实现是否与机器相关, 将运行时环境划分为两部分. 但对于运行在操作系统上的程序, 它们就不需要直接与机器交互了. 那么在操作系统看来, 它应该从什么角度来看这些运行时环境呢?

注意到运行时环境的部分功能是需要使用资源的, 比如申请内存需要使用物理内存, 更新屏幕需要使用帧缓冲. 在PA2中, 我们的计算机系统是被一个程序独占的, 它可以想怎么玩就怎么玩, 玩坏了也是它一个程序的事情. 而在现代的计算机系统中, 可能会有多个程序并发, 甚至同时使用计算机系统中的资源. 如果每个程序都直接使用这些资源, 各自都不知道对方的使用情况, 很快整个系统就会乱套了: 比如我覆盖了你的画面, 你覆盖了我的内存空间...

所以需要有一个角色来对系统中的资源进行统一的管理: 程序不能擅自使用资源了, 使用的时候需要向资源管理者提出申请. 既然操作系统位于ring 0享受着至高无上的权利, 自然地它也需要履行相应的义务: 作为资源管理者管理着系统中的所有资源, 操作系统还需要为用户程序提供相应的服务. 这是操作系统从诞生那一刻就被赋予的使命: 我们之前提到GM-NAA I/O的一个主要任务就是加载新程序, 而它的另一个主要功能, 就是为程序提供输入输出的公共接口.

系统调用的必要性

> 对于批处理系统来说, 系统调用是必须的吗? 如果直接把AM的API暴露给批处理系统中的程序, 会不会有问题呢?

既然受到硬件保护的操作系统要提供服务, 那必定会有相应的接口提供给用户程序. 用户程序只能通过这一接口来请求服务, 这一接口就是系统调用. 系统调用把整个运行时环境分成两部分, 一部分是操作系统内核区, 另一部分是用户区. 那些会访问系统资源的功能会放到内核区中实现, 而用户区则保留一些无需使用系统资源的功能(比如 `strcpy()`), 以及用于请求系统资源相关服务的系统调用接口.

在这个模型之下, 用户程序只能在用户区安分守己地"计算", 任何超越纯粹计算能力之外的任务, 都需要通过系统调用向操作系统请求服务. 如果用户程序尝试进行任何非法操作, CPU就会向操作系统抛出一个异常信号, 并交由操作系统进行处理.

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

系统调用

那么, 触发一个系统调用的具体过程是怎么样的呢?

现实生活中的经验可以给我们一些启发: 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. 用户程序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.

说起"告诉操作系统", 你应该马上想起来, 这是通过自陷指令来实现的. 在GNU/Linux中, 用户程序通过自陷指令 `int $0x80` 指令触发系统调用, Nanos-lite也沿用这个约定. 我们在 `x86-nemu` 的CTE中通过 `int $0x81` 实现 `_yield()` , 虽然它们在CTE中引发了不同的事件, 但从上下文保存到事件分发, 它们的过程都是非常相似的. 既然我们通过自陷指令来触发系统调用, 那么对用户程序来说, 用来向操作系统描述需求的最方便手段就是使用通用寄存器了, 因为执行自陷指令之后, 执行流就会马上切换到操作系统事先设置好的入口, 通用寄存器也会作为上下文的一部分被保存起来. 系统调用处理函数只需要从上下文中获取必要的信息, 就能知道用户程序发出的服务请求是什么了.

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

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

上述内联汇编会先把系统调用的参数依次放入 `%eax` , `%ebx` , `%ecx` , `%edx` 四个寄存器中, 然后执行自陷指令 `int $0x80` . `x86-nemu` 的CTE会将这个自陷操作打包成一个系统调用事件 `_EVENT_SYSCALL` , 并交由Nanos-lite继续处理.

识别系统调用

目前 `dummy` 已经通过 `_syscall_()` 直接触发系统调用, 你需要让Nanos-lite识别出系统调用事件 `_EVENT_SYSCALL` .

你可能需要进行多处的代码修改, 当你为你的代码无法实现正确 而感到疑惑时, 请检查这个过程中的每一个细节.

Nanos-lite收到系统调用事件之后, 就会调出系统调用处理函数 `do_syscall()` 进行处理. `do_syscall()` 首先通过宏 `GPR1` 从上下文 `c` 中获取用户进程之前设置好的系统调用参数, 通过第一个参数 - 系统调用号 - 进行分发. 但目前Nanos-lite没有实现任何系统调用, 因此触发了panic.

添加一个系统调用比你想象中要简单, 所有信息都已经准备好了. 我们只需要在分发的过程中添加相应的系统调用号, 并编写相应的系统调用处理函数 `sys_xxx()` , 然后调用它即可. 回过头来看 `dummy` 程序, 它触发了一个 `SYS_yield` 系统调用. 我们约定, 这个系统调用直接调用CTE的 `_yield()` 即可, 然后返回 `0` .

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

经过CTE, 执行流会从 `do_syscall()` 一路返回到用户程序的上述内联汇编中. 内联汇编最后从 `%eax` 寄存器中取出系统调用的返回值, 并返回给 `_syscall()` 的调用者, 告知其系统调用执行的情况(如是否成功等).

实现**SYS_yield**系统调用

你需要:

1. 在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中实现正确的 `GPR?` 宏, 让它们从上下文 `c` 中获得正确的系统调用参数寄存器.
2. 添加 `SYS_yield` 系统调用.
3. 设置系统调用的返回值.

重新运行dummy程序, 如果你的实现正确, 你会看到dummy程序又触发了一个号码为 `0` 的系统调用. 查看 `nanos-lite/src/syscall.h` , 你会发现它是一个 `SYS_exit` 系统调用. 这说明之前的 `SYS_yield` 已经成功返回, 触发 `SYS_exit` 是因为dummy已经执行完毕, 准备退出了.

实现**SYS_exit**系统调用

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

操作系统之上的**TRM**

我们已经实现了两个很简单的系统调用了, 那么在当前的Nanos-lite上, 用户程序还可以做什么呢? 你也许想起我们在PA2中是如何对程序的需求分类的了, 那就是AM! 最基本的, TRM向我们展示了, 为了满足程序的基本计算能力, 需要有哪些条件:

- 机器提供基本的运算指令

- 能输出字符
- 有堆区可以动态申请内存
- 可以结束运行

基本的运算指令还是得靠机器提供, 也就是你在PA2中已经实现的指令系统. 至于结束运行, `SYS_exit` 系统调用也已经提供了. 为了向用户程序提供输出字符和内存动态申请的功能, 我们需要实现更多的系统调用.

标准输出

在GNU/Linux中, 输出是通过 `SYS_write` 系统调用来实现的. 根据 `write` 的函数声明(参考 `man 2 write`), 你需要在 `do_syscall()` 中识别出系统调用号是 `SYS_write` 之后, 检查 `fd` 的值, 如果 `fd` 是 `1` 或 `2` (分别代表 `stdout` 和 `stderr`), 则将 `buf` 为首地址的 `len` 字节输出到串口(使用 `_putc()` 即可). 最后还要设置正确的返回值, 否则系统调用的调用者会认为 `write` 没有成功执行, 从而进行重试. 至于 `write` 系统调用的返回值是什么, 请查阅 `man 2 write` . 另外不要忘记在 `navy-apps/libs/libos/src/nanos.c` 的 `_write()` 中调用系统调用接口函数.

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

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

在Nanos-lite上运行Hello world

Navy-apps中提供了一个 `hello` 测试程序(`navy-apps/tests/hello`), 它首先通过 `write()` 来输出一句话, 然后通过 `printf()` 来不断输出.

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

- 修改 `nanos-lite/Makefile` 中ramdisk的生成规则, 把ramdisk中的唯一的文件换成hello程序:

```
--- nanos-lite/Makefile
+++ nanos-lite/Makefile
@@ -13,3 +13,3 @@
OBJCOPY_FLAG = -S --set-section-flags .bss=alloc,contents -O binary
-OBJCOPY_APP = $(NAVY_HOME)/tests/dummy
+OBJCOPY_APP = $(NAVY_HOME)/tests/hello
OBJCOPY_FILE = $(OBJCOPY_APP)/build/$(notdir $(OBJCOPY_APP))-$(ISA)
```

- 在 `nanos-lite/` 目录下执行 `make ARCH=x86-nemu update` 更新ramdisk
- 重新编译Nanos-lite并运行

堆区管理

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

调整堆区大小是通过 `sbrk()` 库函数来实现的, 它的原型是

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

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

在Navy-apps的Newlib中, `sbrk()` 最终会调用 `_sbrk()` , 它在 `navy-apps/libs/libos/src/nanos.c` 中定义. 框架代码让 `_sbrk()` 总是返回 `-1` , 表示堆区调整失败, 事实上, 用户程序在第一次调用 `printf()` 的时候会尝试通过 `malloc()` 申请一片缓冲区, 来存放格式化的内容. 若申请失败, 就会逐个字符进行输出. 如果你在Nanos-lite中的 `sys_write()` 中通过 `Log()` 观察其调用情况, 你会发现用户程序通过 `printf()` 输出的时候, 确实是逐个字符地调用 `write()` 来输出的.

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

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

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

实现堆区管理

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

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

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

缓冲区与系统调用开销

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

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

实现了这两个系统调用之后, 原则上所有TRM上能运行的程序, 现在都能在Nanos-lite上运行了. 不过我们目前并没有严格按照AM的API来将相应的系统调用功能暴露给用户程序, 毕竟与AM相比, 对操作系统上运行的程序来说, libc的接口更加广为人们所用, 我们也就不必班门弄斧了.

温馨提示

PA3阶段2到此结束.

简易文件系统

要实现一个完整的批处理系统, 我们还需要向系统提供多个程序. 我们之前把程序以文件的形式存放在ramdisk之中, 但如果程序的数量增加之后, 我们就要知道哪个程序在ramdisk的什么位置. 我们的ramdisk已经提供了读写接口, 使得我们可以很方便地访问某一个位置的内容, 这对Nanos-lite来说貌似没什么困难的地方; 另一方面, 用户程序也需要处理数据, 它们处理的数据也可能会组织成文件, 那么对用户程序来说, 它怎么知道文件位于ramdisk的哪一个位置呢? 更何况文件会动态地增删, 用户程序并不知情. 这说明, 把ramdisk的读写接口直接提供给用户程序来使用是不可行的. 操作系统还需要在存储介质的驱动程序之上为用户程序提供一种更高级的抽象, 那就是文件.

文件的本质就是字节序列, 另外还由一些额外的属性构成. 在这里, 我们先讨论普通意义上的文件. 这样, 那些额外的属性就维护了文件到ramdisk存储位置的映射. 为了管理这些映射, 同时向上层提供文件操作的接口, 我们需要在Nanos-lite中实现一个文件系统.

不要被"文件系统"四个字吓到了, 我们对文件系统的需求并不是那么复杂:

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

既然文件的数量和大小都是固定的, 我们自然可以把每一个文件分别固定在ramdisk中的某一个位置. 这些简化的特性大大降低了文件系统的实现难度. 当然, 真实的文件系统远远比这个简易文件系统复杂.

我们约定文件从ramdisk的最开始一个挨着一个地存放:

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

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

```
--- nanos-lite/Makefile
+++ nanos-lite/Makefile
@@ -40,2 +40,2 @@
-update: update-ramdisk-objcopy src/syscall.h
+update: update-ramdisk-fsimg src/syscall.h
  @touch src/initrd.S
```

然后运行`make ARCH=x86-nemu update`就会自动编译Navy-apps里面的所有程序, 并把`navy-apps/fsimg/`目录下的所有内容整合成ramdisk镜像, 同时生成这个ramdisk镜像的文件记录表`nanos-lite/src/files.h`. 需要注意的是, 并不是Navy-apps里面的所有程序都能在Nanos-lite上运行, 有些程序需要更多系统调用的支持才能运行, 例如NWM和NTerm, 我们并不打算在PA中运行这些程序.

"文件记录表"其实是一个数组, 数组的每个元素都是一个结构体:

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

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

有了这些信息, 就已经可以实现最基本的文件读写操作了:

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

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

```
cat file | less
```

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

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

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

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

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

```
--- nanos-lite/src/fs.c
+++ nanos-lite/src/fs.c
@@ -6,4 +6,5 @@
 typedef struct {
   char *name;          // 文件名
   size_t size;         // 文件大小
   off_t disk_offset;   // 文件在ramdisk中的偏移
+  off_t open_offset;   // 文件被打开之后的读写指针
```

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

偏移量可以通过 `lseek()` 系统调用来调整:

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

为了方便用户程序进行标准输入输出, 操作系统准备了三个默认的文件描述符:

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

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

`nanos-lite/src/fs.c` 中定义的 `file_table` 会包含 `nanos-lite/src/files.h`, 其中前面还有3个特殊的文件: `stdin`, `stdout` 和 `stderr` 的占位表项, 它们只是为了保证我们的简易文件系统和约定的标准输入输出的文件描述符保持一致, 例如根据约定 `stdout` 的文件描述符是 `1`, 而我们添加了三个占位表项之后, 文件记录表中的 `1` 号下标也就不会分配给其它的普通文件了.

根据以上信息, 我们就可以在文件系统中实现以下的文件操作了:

```
int fs_open(const char *pathname, int flags, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
```

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

```
man 2 open
```

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

- 由于简易文件系统中每一个文件都是固定的, 不会产生新文件, 因此"`fs_open()` 没有找到 `pathname` 所指示的文件"属于异常情况, 你需要使用assertion终止程序运行.
- 为了简化实现, 我们允许所有用户程序都可以对所有已存在的文件进行读写, 这样以后, 我们在实现 `fs_open()` 的时候就可以忽略 `flags` 和 `mode` 了.
- 使用 `ramdisk_read()` 和 `ramdisk_write()` 来进行文件的真正读写.
- 由于文件的大小是固定的, 在实现 `fs_read()`, `fs_write()` 和 `fs_lseek()` 的时候, 注意偏移量不要越过文件的边界.
- 除了写入 `stdout` 和 `stderr` 之外(用 `_putc()` 输出到串口), 其余对于 `stdin`, `stdout` 和 `stderr` 这三个特殊文件的操作可以直接忽略.
- 由于我们的简易文件系统没有维护文件打开的状态, `fs_close()` 可以直接返回 `0`, 表示总是关闭成功.

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

让loader使用文件

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

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

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

它用于返回文件描述符 `fd` 所描述的文件的大小.

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

实现完整的文件系统

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

一切皆文件

AM中的IOE向我们展现了程序进行输入输出的需求. 那么在Nanos-lite上, 如果用户程序想访问设备, 要怎么办呢? 一种最直接的方式, 就是让操作系统为每个设备单独提供一个系统调用, 用户程序通过这些系统调用, 就可以直接使用相应的功能了. 然而这种做法却存在不少问题:

- 首先, 设备的类型五花八门, 其功能更是数不胜数, 要为它们分别实现系统调用来给用户程序提供接口, 本身就已经缺乏可行性了;
- 此外, 由于设备的功能差别较大, 若提供的接口不能统一, 程序和设备之间的交互就会变得困难. 所以我们需要有一种方式对设备的功能进行抽象, 向用户程序提供统一的接口.

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

- 内存是以字节编址的, 天然就是一个字节序列, 因而我们之前使用的ramdisk作为字节序列也更加显而易见了
- 管道(shell命令中的 `|`)是一种先进先出的字节序列, 本质上它是内存中的一个队列缓冲区
- 磁盘也可以看成一个字节序列: 我们可以为磁盘上的每一个字节进行编号, 例如第x柱面第y磁头第z扇区中的第n字节, 把磁盘上的所有字节按照编号的大小进行排列, 便得到了一个字节序列
- socket(网络套接字)也是一种字节序列, 它有一个缓冲区, 负责存放接收到的网络数据包, 上层应用将socket中的内容看做是字节序列, 并通过一些特殊的文件操作来处理它们. 比如你之前使用的 `qemu-diff` 就是通过socket与QEMU进行通信的, 而操作socket的方式就是 `fgetc()` 和 `fputc()`
- 操作系统的一些信息可以以字节序列的方式暴露给用户, 例如CPU的配置信息

- 操作系统提供的一些特殊的功能, 如随机数生成器, 也可以看成一个无穷长的字节序列
- 甚至一些非存储类型的硬件也可以看成是字节序列: 我们在键盘上按顺序敲入按键的编码形成了一个字节序列, 显示器上每一个像素的内容按照其顺序也可以看做是字节序列...

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

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

以十六进制的方式查看磁盘上的内容

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

查看CPU是否有Meltdown漏洞

```
cat /proc/cpuinfo | grep 'meltdown'
```

而

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

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

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

虚拟文件系统

为了实现一切皆文件的思想, 我们之前实现的文件操作就需要进行扩展了: 我们不仅需要对普通文件进行读写, 还需要支持各种"特殊文件"的操作. 至于扩展的方式, 你是再熟悉不过的了, 那就是抽象!

我们对之前实现的文件操作API的语义进行扩展, 让它们可以支持任意文件(包括"特殊文件")的操作:

```
int fs_open(const char *pathname, int flags, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
```

这组扩展语义之后的API有一个酷炫的名字, 叫VFS(虚拟文件系统). 既然有虚拟文件系统, 那相应地也应该有"真实文件系统", 这里所谓的真实文件系统, 其实是指具体如何操作某一类文件. 比如在Nanos-lite上, 普通文件通过ramdisk的API进行操作; 在真实的操作系统上, 真实文件系统的种类更是数不胜数: 比如熟悉Windows的你应该知道管理普通文件的NTFS, 目前在GNU/Linux上比较流行的则是EXT4; 至于特殊文件的种类就更多了, 于是相应地有 `procfs` , `tmpfs` , `devfs` , `sysfs` , `initramfs` ... 这些不同的真实文件系统, 它们都分别实现了这些文件的具体操作方式.

所以, VFS其实是对不同种类的真实文件系统的抽象, 它用一组API来描述了这些真实文件系统的抽象行为, 屏蔽了真实文件系统之间的差异, 上层模块(比如系统调用处理函数)不必关心当前操作的文件具体是什么类型, 只要调用这一组API即可完成相应的文件操作. 有了VFS的概念, 要添加一个真实文件系统就非常容易了: 只要把真实文件系统的访问方式包装成VFS的API, 上层模块无需修改任何代码, 就能支持一个新的真实文件系统了.

又来了

阅读上述文字的时候, 如果你想起了AM的概念, 这就对了, 因为VFS背后的思想, 也是抽象.

在Nanos-lite中, 实现VFS的关键就是 `Finfo` 结构体中的两个读写函数指针:

```
typedef struct {
  char *name;          // 文件名
  size_t size;         // 文件大小
  off_t disk_offset;   // 文件在ramdisk中的偏移
  off_t open_offset;   // 文件被打开之后的读写指针
  ReadFn read;         // 读函数指针
  WriteFn write;       // 写函数指针
} Finfo;
```

其中 `ReadFn` 和 `WriteFn` 分别是两种函数指针, 它们用于指向真正进行读写的函数, 并返回成功读写的字节数. 有了这两个函数指针, 我们只需要在文件记录表中对不同的文件设置不同的读写函数, 就可以通过 `f->read()` 和 `f->write()` 的方式来调用具体的读写函数了.

用C语言模拟面向对象编程

> VFS的实现展示了如何用C语言来模拟面向对象编程的一些基本概念: 例如通过结构体来实现类的定义, 结构体中的普通变量可以看作类的成员, 函数指针就可以看作类的方法, 给函数指针设置不同的函数可以实现方法的重载...
>
> 这说明, OOP中那些看似虚无缥缈的概念也没比C语言高级到哪里去, 只不过是OOP的编译器帮我们做了更多的事情, 编译到了机器代码之后, OOP也就不存在了. Object-Oriented Programming With ANSI-C 这本书专门介绍了如何用ANSI-C来模拟OOP的各种概念和功能. 在GNU/Linux的内核代码中, 很多地方也有OOP的影子.

不过在Nanos-lite中, 由于特殊文件的数量很少, 我们约定, 当上述的函数指针为 `NULL` 时, 表示相应文件是一个普通文件, 通过ramdisk的API来进行文件的读写, 这样我们就不需要为大多数的普通文件显式指定ramdisk的读写函数了.

操作系统之上的IOE

有了VFS, 要把IOE抽象成文件就非常简单了.

首先当然是来看最简单的输出设备: 串口. 在Nanos-lite中, `stdout` 和 `stderr` 都会输出到串口. 之前你可能会通过判断 `fd` 是否为 `1` 或 `2` , 来决定 `sys_write()` 是否写入到串口. 现在有了VFS, 我们就不需要让系统调用处理函数关心这些特殊文件的情况了: 我们只需要在 `nanos-lite/src/device.c` 中实现 `serial_write()` , 然后在文件记录表中设置相应的写函数, 就可以实现上述功能了. 需要说明的是, `serial_write()` 中的 `offset` 参数可以忽略, 因为对串口来说, `offset` 是没有意义的. 另外Nanos-lite也不打算支持 `stdin` 的读入, 因此在文件记录表中设置相应的报错函数即可.

把串口抽象成文件

> 根据上述内容, 让VFS支持串口的写入.

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

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

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

至于具体的屏幕大小, 你需要通过IOE的相应API来获取.

更新框架代码

我们在2018/11/08 21:00:00修复了在 `native` 上运行 `bmptest` 发生段错误的bug. 如果你在此时间之前获得框架代码, 请根据这里手动更新该文件; 如果你在此时间之后获得框架代码, 你不需要进行额外的操作.

把VGA显存抽象成文件

你需要在Nanos-lite中:

- 在 `init_fs()` (在 `nanos-lite/src/fs.c` 中定义)中对文件记录表中 `/dev/fb` 的大小进行初始化.
- 实现 `fb_write()` (在 `nanos-lite/src/device.c` 中定义), 用于把 `buf` 中的 `len` 字节写到屏幕上 `offset` 处. 你需要先从 `offset` 计算出屏幕上的坐标, 然后调用IOE的 `draw_rect()` .
- 在 `init_device()` (在 `nanos-lite/src/device.c` 中定义)中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中.
- 实现 `dispinfo_read()` (在 `nanos-lite/src/device.c` 中定义), 用于把字符串 `dispinfo` 中 `offset` 开始的 `len` 字节写到 `buf` 中.
- 在VFS中添加对 `/dev/fb` 和 `/proc/dispinfo` 这两个特殊文件的支持.

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

最后我们来看输入设备. 输入设备有键盘和时钟, 它们对系统来说本质上就是到来了一个事件. 一种简单的方式是把事件以文本的形式表现出来, 我们定义以下事件, 一个事件以换行符 `\n` 结束:

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

我们采用文本形式来描述事件有两个好处, 首先文本显然是一种字节序列, 这使得事件很容易抽象成文件; 此外文本方式使得用户程序可以容易可读地解析事件的内容. Nanos-lite和Navy-apps约定, 上述事件抽象成文件 `/dev/events` , 它需要支持读操作, 用户程序可以从中一次读出一个输入事件. 需要注意的是, 由于时钟事件可以任意时刻进行读取, 我们需要优先处理按键事件, 当不存在按键事件的时候, 才返回时钟事件, 否则用户程序将永远无法读到按键事件.

把设备输入抽象成文件

你需要在Nanos-lite中

- 实现 `events_read()` (在 `nanos-lite/src/device.c` 中定义), 把事件写入到 `buf` 中, 最长写入 `len` 字节, 然后返回写入的实际长度. 其中按键名已经在字符串数组 `names` 中定

义好了. 你需要借助IOE的API来获得设备的输入.
- 在VFS中添加对 `/dev/events` 的支持.

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

最后, 为了方便用户程序的使用, Navy-apps把上述这些特殊文件的功能封装成NDL(NJU DirectMedia Layer)多媒体库, 具体的API请阅读 `navy-apps/libs/libndl/` 目录下的代码. 有了NDL库, 用户程序就可以很方便地进行I/O操作了.

运行仙剑奇侠传

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

我们的框架代码已经把SDLPAL移植到Navy-apps中了. 移植的主要工作就是把仙剑奇侠传调用的所有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Nanos-lite和Navy-apps提供的运行时环境重写它们. 重写内容主要包括以下三部分:

- C标准库
- 浮点数
- SDL库

Navy-apps中的Newlib已经提供了C标准库的功能, 我们无需额外移植. 关于浮点数的移植工作, 我们会在PA5中再来讨论, 目前先忽略它. 为了用NDL的API来替代原来SDL的相应功能, 移植工作需要对SDLPAL进行了少量修改, 包括去掉了声音, 修改了和按键相关的处理, 把我们关心的与NDL相关的功能整理到 `hal/hal.c` 中, 一些我们不必关心的实现则整理到 `unused/` 目录下. 框架代码已经把这些移植工作都做好了, 目前你不需要编写额外的代码来进行移植.

在NEMU中运行仙剑奇侠传

现在是万事俱备, 终于到了激动人心的时刻了! 从课程网站上下载仙剑奇侠传的数据文件, 并放到 `navy-apps/fsimg/share/games/pal/` 目录下, 更新ramdisk之后, 在Nanos-lite中加载并运行 `/bin/pal` .

不过由于我们的疏忽, 游戏默认跳过了片头动画. 你可以进行以下修改来加入片头动画的播放:

```
--- navy-apps/apps/pal/src/main.c
+++ navy-apps/apps/pal/src/main.c
@@ -552,5 +552,5 @@
    //
    // Show the trademark screen and splash screen
    //
-   //PAL_TrademarkScreen();
-   //PAL_SplashScreen();
+   PAL_TrademarkScreen();
+   PAL_SplashScreen();
```

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. 但与怪物进行战斗需要进行一些浮点数相关的计算, 而NEMU目前没有实现浮点数, 因而不能成功进行战斗. 我们会在PA5中再来解决浮点数的问题, 目前我们先暂时不触发战斗, 可以先通过"新的故事"进行游戏.

不要在仙剑奇侠传中按p键

在SDLPAL中, `p` 键是用于截屏的快捷键, 会把截屏结果保存到一个新文件中. 但我们的简易文件系统并不支持新文件的创建, 从而导致 `panic` , 这属于正常现象.

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.

基础设施(3)

如果你的仙剑奇侠传无法正确运行, 借助AM, 你应该可以很快确认是硬件还是软件bug. 如果是硬件bug, 你也许会陷入绝望之中: 基于QEMU的DiffTest速度太慢了! 有什么方法可以加快DiffTest的速度呢?

自由开关DiffTest模式

目前每次DiffTest都是从一开始进行, 但如果这个bug在很久之后才触发, 那么每次都从一开始进行DiffTest是没有必要的. 如果我们怀疑bug在某个函数中触发, 那么我们更希望DUT首先按照正常模式运行到这个函数, 然后开启DiffTest模式, 再进入这个函数. 这样, 我们就节省了前期大量的不必要的比对开销了.

为了实现这个功能, 关键是要在DUT运行中的某一时刻开始进入DiffTest模式. 而进入DiffTest模式的一个重要前提, 就是让DUT和REF的状态保持一致, 否则进行比对的结果就失去了意义. 我们又再次提到了状态的概念, 你应该再熟悉不过了: 计算机的状态就是计算机中的时序逻辑部件的状态. 这样, 我们只要在进入DiffTest模式之前, 把REF的寄存器和内存设置成和DUT一样, 它们就可以从一个相同的状态开始进行对比了.

幸运的是, DiffTest的API已经把大部分的工作都准备好了. 为了控制DUT是否开启DiffTest模式, 我们还需要在简易调试器中添加如下两个命令:

- `detach` 命令用于退出DiffTest模式, 之后DUT执行的所有指令将不再与REF进行比对. 实现方式非常简单, 只需要让 `difftest_step()`, `difftest_skip_dut()` 和 `difftest_skip_ref()` 直接返回即可.
- `attach` 命令用于进入DiffTest模式, 之后DUT执行的所有指令将逐条与REF进行比对. 为此, 你还需要将DUT中 `[0, 0x7c00)` 和 `[0x100000, PMEM_SIZE)` 的内存内容分别设置到REF相应的内存区间中, 并将DUT的寄存器状态也同步到REF中. 因为QEMU中在 `0x7c00` 附近会有GDT相关的代码, 覆盖这段代码会使得QEMU无法在保护模式下运行, 导致后续无法进行DiffTest, 因此我们要绕过相应的内存区间, 具体细节可以参考 `nemu/tools/qemu-diff/src/diff-test.c` 中的 `difftest_init()` 函数.

这样以后, 你就可以通过以下方式来在客户程序运行到某个目标位置的时候开启DiffTest了:

- 去掉运行NEMU的 `-b` 参数(在 `nexus-am/am/arch/x86-nemu/img/run` 中), 使得我们可以在客户程序开始运行前键入命令
- 键入 `detach` 命令, 退出DiffTest模式
- 通过单步执行, 监视点, 断点等方式, 让客户程序通过正常模式运行到目标位置

- 键入 `attach` 命令, 进入DiffTest模式, 注意设置REF的内存需要花费约数十秒的时间
- 之后就可以在DiffTest模式下继续运行客户程序了

不过上面的方法还有漏网之鱼, 具体来说, 我们还需要处理EFLAGS和IDTR这两个寄存器, 否则, 不一致的EFLAGS会导致接下来的 `jcc` 或者 `setcc` 指令在QEMU中的执行产生非预期结果, 而不一致的IDTR将会导致在QEMU中执行的系统调用因无法找到正确的目标位置而崩溃. 为了解决EFLAGS的一致性问题, 我们可以修改 `qemu-diff` 的代码, 让 `DIFFTEST_REG_SIZE` 覆盖EFLAGS, 这样就可以在DiffTest和寄存器访问相关的API中处理EFALGS了, 但不要忘记检查寄存器排列的顺序, 同时在 `difftest_step()` 中跳过EFLAGS相关的检查, 因为NEMU中对EFLAGS的实现是不完整的.

而为了解决IDTR的问题, 我们需要使用一种稍微复杂的方法, 这是因为GDB协议中没有定义IDTR寄存器的访问, 这意味着我们无法通过 `difftest_setregs()` 来修改QEMU中的IDTR. 既然无法直接修改, 我们就让QEMU执行一段指定的代码来设置IDTR吧. 具体地, 我们只要让QEMU执行一条 `lidt` 指令就可以了, 通过这条指令将一个IDT描述符的内容装载到IDTR中. 所以我们首先需要准备一个长度为6字节的IDT描述符, 内容就是当前NEMU中IDTR的内容. 我们需要将这个描述符放置到QEMU中一个安全的地方, 不要被其它内容覆盖的同时, 也不要覆盖其它内容. 一个合适的内存位置就是 `0x7e00` , 我们可以将IDT描述符放在REF的这个内存位置. 然后再准备一条 `lidt (0x7e00)` 指令, 把这条指令放在REF的内存位置 `0x7e40` , 然后将REF的eip设置成这个内存位置, 并让REF执行一条指令. 顺利的话, QEMU就设置好IDTR了, 将来QEMU就可以成功执行系统调用了.

更新框架代码

我们在2018/11/24 18:00:00修复了一个原则上不算是bug的bug. 如果你在此时间之前获得框架代码, 请根据这里手动更新该文件; 如果你在此时间之后获得框架代码, 你不需要进行额外的操作.

实现可自由开关的DiffTest

根据上述内容, 在简易调试器中添加 `detach` 和 `attach` 命令, 实现正常模式和DiffTest模式的自由切换. 如果你不知道怎么做, 请参考我们在PA2中讨论的DiffTest API.

上述文字基本上把涉及的主要内容都介绍过了, 不过留了一些小坑, 关键就看你对状态的理解是否到位了.

快照

更进一步的, 其实连NEMU也没有必要每次都从头开始执行. 我们可以像仙剑奇侠传的存档系统一样, 把NEMU的状态保存到文件中, 以后就可以直接从文件中恢复到这个状态继续执行了. 在虚拟化领域中, 这样的机制有一个专门的名字, 叫快照. 如果你用虚拟机来做PA, 相信你对这

个名词应该不会陌生. 在NEMU中实现快照是一件非常简单的事情, 我们只需要在简易调试器中添加如下命令即可:

- `save [path]`, 将NEMU的当前状态保存到`path`指示的文件中
- `load [path]`, 从`path`指示的文件中恢复NEMU的状态

在NEMU中实现快照

关于NEMU的状态, 我们已经强调过无数次了, 快去实现吧. 另外, 由于我们可能会在不同的目录中执行NEMU, 因此使用快照的时候, 建议你通过绝对路径来指示快照文件.

展示你的批处理系统

在PA3的最后, 你将会向Nanos-lite中添加一些简单的功能, 来展示你的批处理系统.

Navy-apps准备了一个开机菜单程序, 在Nanos-lite中加载`/bin/init`就可以运行它了. 不过为了运行它, 你还需要在VFS中添加一个特殊文件`/dev/tty`, 只要让它往串口写入即可. 这个开机菜单程序中准备了若干项选择, 你可以通过键入数字来选择运行相应的程序. 不过你会发现, 键入数字之后就会触发HIT BAD TRAP, 这是因为我们需要实现一个新的系统调用.

这个系统调用就是`SYS_execve`, 它的作用是结束当前程序的运行, 并启动一个指定的程序. 这个系统调用比较特殊, 如果它执行成功, 就不会返回到当前程序中, 具体信息可以参考`man execve`. 为了实现这个系统调用, 你只需要在相应的系统调用处理函数中调用`naive_uload()`就可以了. 目前我们只需要关心`filename`即可, `argv`和`envp`这两个参数可以暂时忽略.

添加开机菜单

你需要实现`SYS_execve`系统调用. 你已经实现过很多系统调用了, 需要注意哪些细节, 这里就不啰嗦了.

链接错误

你可能会(也可能不会)在链接`init`的时候遇到"`_fork()`和`_wait()`的引用未定义"的错误. 这是由于框架代码的疏忽造成的. 为了修复这个错误, 你可以在`navy-apps/libs/libos/src/nanos.c`中定义这两个函数, 函数体留空即可.

展示你的批处理系统

有了开机菜单程序之后, 就可以很容易地实现一个有点样子的批处理系统了. 你只需要修改 `SYS_exit` 的实现, 让它调用 `SYS_execve` 来再次运行 `/bin/init` , 而不是直接调用 `_halt()` 来结束整个系统的运行. 这样以后, 在一个用户程序结束的时候, 操作系统就会自动再次运行开机菜单程序, 来让用户选择一个新的程序来运行.

运行超级玛丽(建议二周目思考)

开机菜单中有两个和LiteNES相关的选项, 分别是运行功夫和超级玛丽. 但目前运行它们可能会触发错误, 你知道这是为什么吗?

如果你修复了上述错误, 再选择运行超级玛丽, 你可能会发现最后运行的还是功夫, 你知道这又是为什么吗? 为了让这两个选项运行不同的游戏, 你需要对系统(而不是LiteNES本身)进行哪些改动呢?

体会一下就行了

由于性能原因, LiteNES的运行会非常缓慢. 实验并不要求你流畅地运行LiteNES, 体会一下就行了.

到这里为止, 我们基本上实现了一个"现代风"的批处理系统了. 如果你还记得PA1一开始提到的红白机, 现在我们实现的功能, 就类似红白机开机之后的游戏选择菜单, 甚至比红白机的功能还强大, 毕竟红白机上面并没有运行操作系统. 我们通过在机器中添加CTE来支持用户程序和操作系统之间的执行流切换, 并基于此实现系统调用机制, 向用户程序提供了全新的运行时环境. 系统是用来运行程序的, 我们又通过系统调用给用户程序开放更多的功能, 包括AM中不存在的文件系统. 就靠这几个看似功能简陋的系统调用, 我们就已经能支撑真实游戏仙剑奇侠传的运行了. 真实的游戏和系统固然会更复杂, 但现在的你, 是否觉得对它们多了一些理解呢?

必答题

文件读写的具体过程 仙剑奇侠传中有以下行为:

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

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

温馨提示

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

PA4 - 虚实交错的魔法: 分时多任务

世界诞生的故事 - 第四章

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

代码管理

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:

- task PA4.1: 实现基本的多道程序系统 - 2018/12/16 23:59:59
- task PA4.2: 实现支持虚存管理的多道程序系统 - 2018/12/23 23:59:59
- task PA4.3: 实现抢占式分时多任务系统, 并提交完整的实验报告 - 2018/12/30 23:59:59

提交说明: 见这里

甩锅声明

从PA4开始, 讲义中就不再提供滴水不漏的代码指导了, 部分关键的代码细节需要你自己去思考和尝试(我们故意省略的). 我们会在讲义中将技术的原理阐述清楚, 你需要首先理解这些原理, 然后根据理解来阅读并编写相应的代码.

一句话总结, 与其抱怨讲义写得不清楚, 还不如自己多多思考. 现在都到PA4了, 为了让成绩符合正态分布, 拿高分总需要多付出点努力吧. 如果你之前都是"一切以完成实验优先"而不去深入理解系统如何工作, 现在到了该吃亏的时候了.

多道程序

通过Nanos-lite的支撑, 我们已经在NEMU中成功运行了一个批处理系统, 并把仙剑奇侠传跑起来了! 这说明我们亲自构建的NEMU这个看似简单的机器, 同样能支撑真实程序的运行, 丝毫不逊色于真实的机器! 不过, 这个批处理系统目前还是只能同时运行一个程序, 只有当一个程序结束执行之后, 才会开始执行下一个程序.

这也正是批处理系统的一个缺陷: 如果当前程序正在等待输入输出, 那么整个系统都会因此而停顿. 和CPU的性能相比, 输入输出是非常缓慢的: 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. 但事实上, 与其让系统陷入无意义的等待, 还不如用这些时间来进行一些有意义的工作. 一个简单的想法就是, 在系统一开始的时候加载多个程序, 然后运行第一个; 当第一个程序需要等待输入输出的时候, 就切换到第二个程序来运行; 当第二个程序也需要等待的时候, 就继续切换到下一个程序来运行, 如此类推.

这就是多道程序(multiprogramming)系统的基本思想. 多道程序的想法看着很简单, 但它也是一种多任务系统, 这是因为它已经包含了多任务系统的基本要素. 换句话说, 要把批处理的Nanos-lite改造成一个多道程序操作系统, 我们只需要实现以下两点就可以了:

- 在内存中可以同时存在多个进程
- 在满足某些条件的情况下, 可以让执行流在这些进程之间切换

术语变更

既然是多任务系统, 系统中就运行的程序就不止一个了. 现在我们就可以直接使用"进程"的概念了.

第一点的实现很简单, 我们只要让loader把不同的进程加载在不同的内存位置就可以了, 加载进程的过程本质上就是一些内存拷贝的操作, 因此并没有什么困难的地方. 甚至我们可以在Nanos-lite中直接定义一些测试函数来作为程序, 因为程序本质上就是一些有意义的指令序列,

目前我们不必在意这些指令序列到底从哪里来. 不过, 一个需要注意的地方是栈, 我们需要为每个进程分配各自的栈空间.

为什么需要使用不同的栈空间?

如果不同的进程共享同一个栈空间, 会发生什么呢?

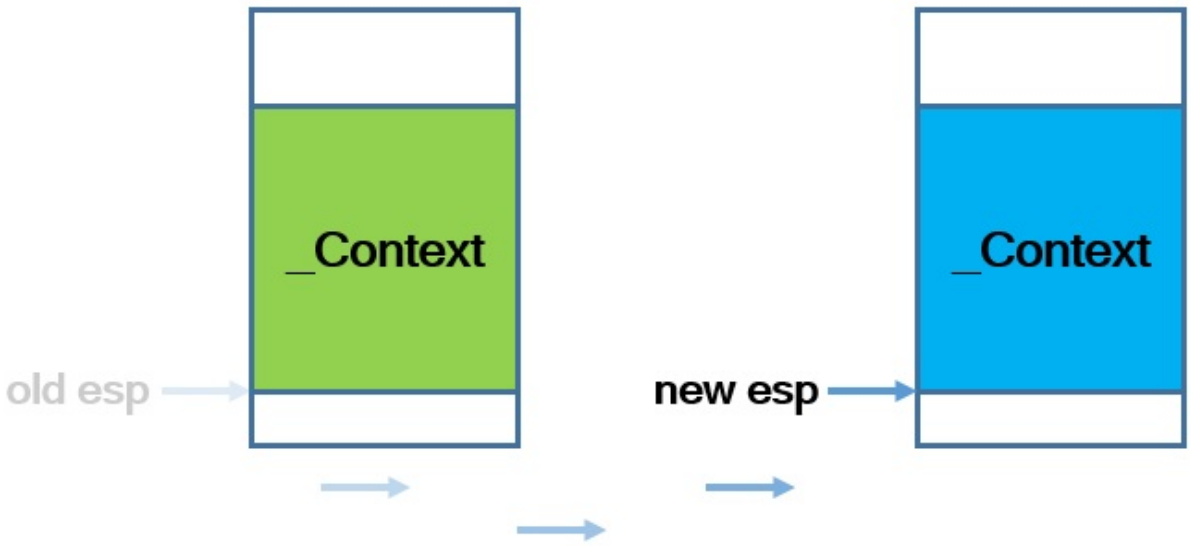
反而需要深思熟虑的是第二点, 怎么让执行流在进程之间切换表面上看并不是一件直观的事情.

上下文切换

在PA3中, 我们已经提到了操作系统和用户进程之间的执行流切换, 并介绍了"上下文"的概念: 上下文本质上就是进程的状态. 换句话说, 我们现在需要考虑的是, 如何在多个用户进程之间进行上下文切换.

基本原理

事实上, 有了CTE, 我们就有一种很巧妙的方式来实现上下文切换了. 具体地, 假设进程A运行的过程中触发了系统调用, 陷入到内核. 根据 `asm_trap()` 的代码, A的上下文结构(`_Context`)将会被保存到A的栈上. 本来系统调用处理完毕之后, `asm_trap()` 会根据栈上保存的上下文结构来恢复A的上下文. 神奇的地方来了, 如果我们先不着急恢复A的上下文, 而是先将栈顶指针切换到另一个进程B的栈上, 那会发生什么呢? 由于B的栈上存放了之前B保存的上下文结构, 接下来的操作就会根据这一结构来恢复B的上下文: 恢复B的通用寄存器, 弹出#irq和错误码, 恢复B的EIP, CS, EFLAGS. 从 `asm_trap()` 返回之后, 我们已经在运行进程B了!



那进程A到哪里去了呢? 别担心, 它只是被暂时"挂起"了而已. 在被挂起之前, 它已经把上下文结构保存到自己的栈上了, 如果将来的某一时刻栈顶指针被切换到A的栈上, 代码将会根据栈上的上下文结构来恢复A的上下文, A将得以唤醒并执行. 所以, 上下文切换其实就是不同进程之间的栈切换!

进程控制块

但是, 我们要如何找到别的进程的上下文结构呢? 注意到上下文结构是保存在栈上的, 但栈空间那么大, 受到函数调用形成的栈帧的影响, 每次保存上下文结构的位置并不是固定的. 自然地, 我们需要一个 `cp` 指针(context pointer)来记录上下文结构的位置, 当想要找到其它进程的上下文结构的时候, 只要寻找这个进程相关的 `cp` 指针即可.

事实上, 有不少信息都是进程相关的, 除了刚才提到的上下文指针 `cp` 之外, 上文提到的栈空间也是如此. 为了方便对这些进程相关的信息进行管理, 操作系统使用一种叫进程控制块(PCB, process control block)的数据结构, 为每一个进程维护一个PCB. Nanos-lite的框架代码中已经定义了我们所需要使用的PCB结构(在 `nanos-lite/include/proc.h` 中定义):

```
typedef union {
  uint8_t stack[STACK_SIZE] PG_ALIGN;
  struct {
    _Context *cp;
  };
} PCB;
```

PCB中还定义了其它成员, 目前可以忽略它们, 我们会在将来介绍它们.

更新变量名

> 由于我们的疏忽, 框架代码中的上下文指针仍然使用了旧的变量名 `tf`, 为了和讲义保持一致, 你可以把变量名修改成 `cp`. 在 `nanos-lite/src/loader.c` 的代码中也有对 `tf` 成员的两处引用, 你还需要修改它们.

Nanos-lite使用一个联合体来把其它信息放置在进程堆栈的底部. 代码为每一个进程分配了一个32KB的堆栈, 已经足够使用了, 不会出现栈溢出导致未定义行为. 在进行上下文切换的时候, 只需要把PCB中的 `cp` 指针返回给CTE的 `irq_handle()` 函数即可, 剩余部分的代码会根据上下文结构恢复现场. 我们只要稍稍借助数学归纳法, 就可以让我们相信这个过程对于正在运行的进程来说总是正确的.

创建上下文

那么, 对于刚刚加载完的进程, 我们要怎么切换到它来让它运行起来呢? 答案很简单, 我们只需要在进程的栈上人工创建一个上下文结构, 使得将来切换的时候可以根据这个结构来正确地恢复上下文即可. 具体来说, 我们还需要思考如何初始化上下文结构中的每一个成员, 因此我们需要仔细思考这些成员对一开始运行的进程有什么影响. 提醒一下, 为了保证DiffTest的正确运行, 我们还是把上下文结构中的 `cs` 设置为 `8`.

创建上下文是通过CTE提供的 `_kcontext()` 函数 (在 `nexus-am/am/arch/x86-nemu/src/cte.c` 中定义)来实现的, 它的原型是

```
_Context *_kcontext(_Area stack, void (*entry)(void *), void *arg);
```

你需要在 `stack` 的底部创建一个以 `entry` 为返回地址的上下文结构(目前你可以先忽略 `arg` 参数). 然后返回这一结构的指针, 由Nanos-lite把这一指针记录到进程PCB的 `cp` 中:

```
|               |
+---------------+ <---- stack.end
|               |
|    context    |
|               |
+---------------+ <--+
|               |    |
|               |    |
|               |    |
|               |    |
+---------------+    |
|       cp      | ---+
+---------------+ <---- stack.start
|               |
```

进程调度

上下文的创建和切换是CTE的工作, 而具体切换到哪个进程的上下文, 是由操作系统来决定的, 这项任务叫做进程调度. 进程调度是由 `schedule()` 函数(在 `nanos-lite/src/proc.c` 中定义)来完成的, 它用于返回将要调度的进程上下文. 因此, 我们需要一种方式来记录当前正在运行哪一个进程, 这样我们才能在 `schedule()` 中返回另一个进程的上下文, 以实现多任务的效果. 这一工作是通过 `current` 指针(在 `nanos-lite/src/proc.c` 中定义)来实现的, 它用于指向当前运行进程的PCB. 这样, 我们就可以在 `schedule()` 中通过 `current` 来决定接下来要调度哪一个进程了. 不过在调度之前, 我们还需要把当前进程的上下文指针保存在PCB当中:

```
// save the context pointer
current->cp = prev;

// always select pcb[0] as the new process
current = &pcb[0];

// then return the new context
return current->cp;
```

目前我们让 `schedule()` 总是切换到第一个用户进程, 即 `pcb[0]` . 注意它的上下文是通过 `_kcontext()` 创建的, 在 `schedule()` 中才决定要切换到它, 然后在CTE的 `asm_trap()` 中才真正地恢复这一上下文.

实现上下文切换

根据讲义的上述内容, 实现以下功能:

- CTE的 `_kcontext()` 函数
- Nanos-lite的 `schedule()` 函数
- 在Nanos-lite收到 `_EVENT_YIELD` 事件后, 调用 `schedule()` 并返回其现场
- 修改CTE中 `asm_trap()` 的实现, 使得从 `irq_handle()` 返回后, 先将栈顶指针切换到新进程的上下文结构, 然后才恢复上下文, 从而完成上下文切换的本质操作

为了测试实现的正确性, 框架代码提供了一个测试函数 `hello_fun()` (在 `nanos-lite/src/proc.c` 中定义). 你需要在 `init_proc()` 中单独创建一个以 `hello_fun` 为返回地址的上下文:

```
void init_proc() {
  context_kload(&pcb[0], (void *)hello_fun);
  switch_boot_pcb();
}
```

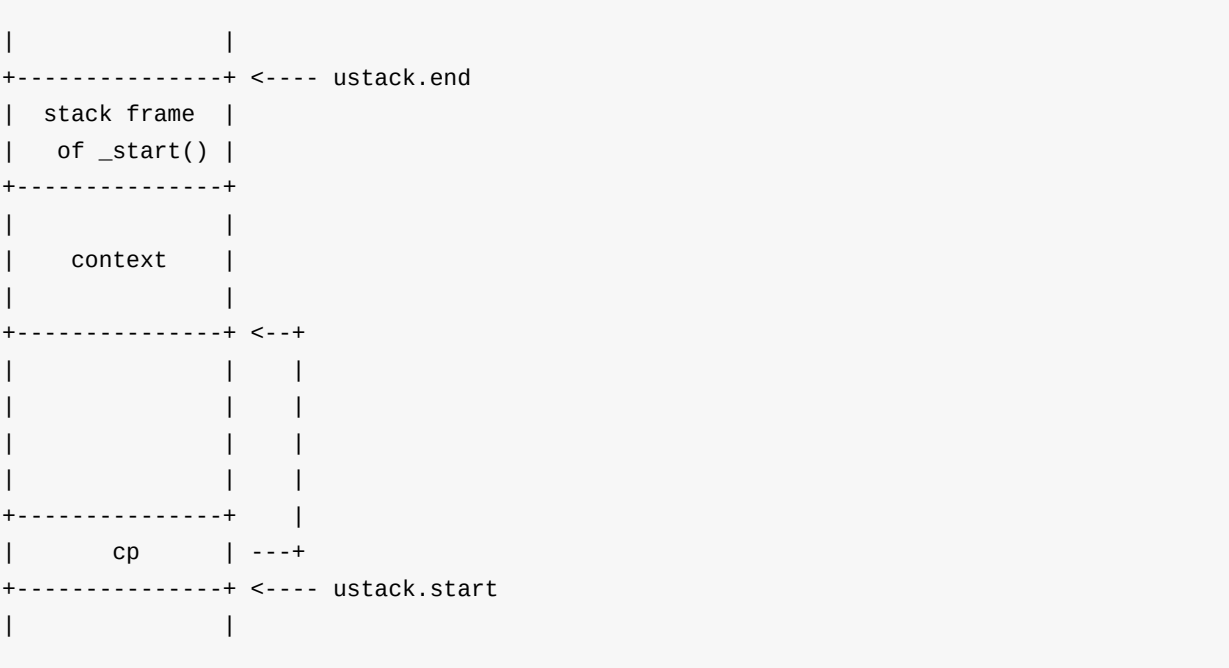
其中, `context_kload()` 在 `nanos-lite/src/loader.c` 中定义, 它会调用CTE的 `kcontext()` 来创建一个上下文, 而调用 `switch_boot_pcb()` 则是为了初始化 `current` 指针. 如果你的实现正确, 你将会看到 `hello_fun()` 中的输出信息.

创建用户进程上下文

创建用户进程的上下文则需要一些额外的考量. 我们知道, 用户进程的 `main()` 函数是有参数的, ABI规定, `main()` 函数的参数需在用户进程开始运行之前要由运行时环境准备好. 为了遵循这一约定, AM额外准备了一个API `_ucontext()` , 专门用于创建用户进程的上下文. `_ucontext()` 的原型是

```
_Context *_ucontext(_Protect *p, _Area ustack, _Area kstack, void *entry, void *arg);
```

其中, 参数 `p`, `kstack` 和 `arg` 目前不会用到, 可以忽略它们. 与 `_kcontext()` 相比, 除了在栈上创建必要的上下文信息之外, `_ucontext()` 还需要在栈上准备一个栈帧, 用于存放 `main()` 函数的参数信息. 这个栈帧将来会被 `navy-apps/libs/libc/src/platform/crt0.c` 中的 `_start()` 函数使用, 它会把参数信息传给 `main()` 函数.



目前我们只需要把这一栈帧中的参数设置为 `0` 或 `NULL` 即可, 至于返回地址, 我们永远不会从 `_start()` 返回, 因此可以不设置它.

实现 `_ucontext()` 后, 我们就可以加载用户进程了. 我们添加一个开机画面进程, 同时修改调度的代码, 让其轮流返回两个进程的上下文:

```
// init_proc()
context_uload(&pcb[1], "/bin/init");

// schedule()
current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
```

最后, 我们还需要在 `serial_write()`, `events_read()` 和 `fb_write()` 的开头调用 `_yield()`, 来模拟设备访问缓慢的情况. 添加之后, 访问设备时就要进行上下文切换, 从而实现多道程序系统的功能.

实现多道程序系统

根据上述内容, 实现多道程序系统. 如果你的实现正确, 你将可以一边运行仙剑奇侠传的同时, 一边输出hello信息.

bug修复

我们在2018/12/09 16:30:00修复了`native`运行多道程序时触发段错误的bug. 如果你在此时间之前获得框架代码, 请根据这里手动更新该文件; 如果你在此时间之后获得框架代码, 你不需要进行额外的操作.

一山不能藏二虎?

尝试把`hello_fun()`换成Navy-apps中的`hello`:

```
-context_kload(&pcb[0], (void *)hello_fun);
+context_uload(&pcb[0], "/bin/hello");
```

你发现了什么问题? 为什么会这样? 思考一下, 答案很快揭晓!

温馨提示

PA4阶段1到此结束.

程序和内存位置

我们已经实现了一个多道程序的多任务系统, 但却在尝试运行第二个用户进程的时候遇到了问题. 如果你还记得编译Navy-apps中的程序时, 我们都把它们链接到 `0x4000000` 的内存位置, 你就会知道问题的原因了: 如果我们正在运行仙剑奇侠传, 同时也想运行hello程序, 它们的内容就会被相互覆盖!

出现上述问题的根本原因是, "在内存中可以同时存在多个进程"的条件被打破了. 显然我们需要对内存进行管理, 而不是让多个进程随意使用. 我们在PA3中提到操作系统有管理系统资源的义务, 在多任务操作系统中, 内存作为一种资源自然也是要被管理起来. 操作系统需要记录内存的分配情况, 需要运行一个新进程的时候, 就给它分配一片空闲的内存位置, 把它加载到这一内存位置上即可.

不过, 这片空闲的内存位置是操作系统的加载器在加载时刻指定的, 但进程代码真的可以在这一内存位置上正确运行吗?

绝对代码

一般来说, 程序的内存位置是在链接时刻(link time)确定的(Navy-apps中的程序就是这样), 以前的程序员甚至在程序中使用绝对地址来进行内存访问, 这两种代码称为绝对代码(absolute code). 绝对代码会假设程序对象(函数和数据)位于某个固定的位置, 比如链接时重定位之后, 程序就会认为某个变量 `a` 位于内存位置 `0x4001234` , 如果我们把程序加载到别的位置, 加载后变量 `a` 的实际位置就改变了, 但程序中的绝对代码还是认为变量 `a` 仍然位于内存位置 `0x4001234` , 访问 `0x4001234` 也就无法访问真正的变量 `a` . 显然, 绝对代码只能在固定的内存位置才能正确运行.

但操作系统在加载时刻分配的空闲内存位置, 并不总是能让这种程序正确运行. 因此, 这个问题的一个解决方案, 就是让操作系统记录程序的加载位置, 当一个程序试图加载到一个已经被使用的内存位置时, 加载将会失败, 操作系统将返回一个错误. 为了避免加载失败, 一个方法是为每个程序维护多个不同加载地址的版本, 期望其中有一个版本可以被成功加载.

可重定位代码

但这样太麻烦了. 为什么一定要提前确定一个程序的加载位置呢? 如果我们把链接时的重定位阶段往后推迟, 不就可以打破绝对代码的限制了吗?

于是有程序员开发了一类"自重定位(self-relocation)"的特殊程序, 这种程序可以在开始运行的时候, 先把自己重定位到其它内存位置, 然后再开始真正的运行. 这种重定位类型称为"运行时(run time)重定位", 但这也并没有真正解决问题, 因为程序在运行时刻并不知道重定位的目标内存位置是否空闲.

既然只有操作系统才知道内存是否空闲, 那就干脆让加载器来进行重定位吧, 于是有了"加载时(load time)重定位"的说法. 具体地, 加载器会申请一个空闲的内存位置, 然后将程序加载到这个内存位置, 并把程序重定位到这个内存位置, 之后才会执行这个程序. 今天的GNU/Linux就是通过这种方式来插入内核模块的.

位置无关代码

从某种程度上来说, 加载时重定位会带来额外的开销: 如果一个程序要被重复执行多次, 那么就要进行多次的加载时重定位. 早期的计算机速度较慢, 大家觉得加载时重定位的开销还是不能忽略的, 而且加载时重定位需要对整个程序进行处理, 如果程序比较大的话, 开销就更明显了.

有没有方法可以节省重定位的开销, 甚至不进行重定位呢? 但链接时的重定位又可能会产生绝对代码, 这并不是我们所希望的. 如果程序中的所有寻址, 都是针对程序位置来进行相对寻址操作, 这样的程序就可以被加载到任意位置执行, 而不会出现绝对代码的问题了,

这就是PIC(position-independent code, 位置无关代码)的基本思想. 今天的动态库都是PIC, 这样它们就可以被加载到任意的内存位置了. 此外, 如果一个可执行文件全部由PIC组成, 那么它有一个新名字, 叫PIE(position-independent executable, 位置无关可执行文件). 编译器可以通过特定的选项编译出PIE. 和一般的程序不同, PIE还能在一定程度上对恶意的攻击程序造成了干扰: 恶意程序也无法提前假设PIE运行的地址. 也正是因为这一安全相关的特性, 最近的不少GNU/Linux的发行版上配置的gcc都默认生成PIE. 不过, 使用相对寻址会使得程序的代码量增大, 性能也会受到一些影响, 但对于早期的计算机来说, 内存是一种非常珍贵的资源, 降低性能也是大家不愿意看到的, 因此对于PIC和PIE, 大家也会慎重考虑.

实现基于PIE的loader(建议二周目思考)

> 天下并没有免费的午餐, PIE之所以能做到位置无关, 其实是要依赖于程序中一个叫GOT(global offset table, 全局偏移量表)的数据结构. 要正确运行PIE, 操作系统中的加载器需要在加载程序的时候往GOT中填写正确的内容.
>
> 有兴趣的同学可以让Nanos-lite的loader支持PIE, 当然这需要了解一些ELF相关的细节, 具体细节可以参考ABI手册.

如果世界上的所有程序都是可重定位的, 或者是PIC, 内存覆盖的问题就不攻自破了. 但总有一些包含绝对代码的程序, 考虑到兼容问题, 还需要想办法运行它们. 有没有更好的, 一劳永逸的方案呢?

虚实交错的魔法

我们刚才是从程序本身来考量, 自然无法绕开绝对代码的问题. 为了解决这个问题, 我们需要从另一个方面 - 内存 - 来思考. 我们知道程序会经历编译, 链接, 加载, 运行这四个阶段, 绝对代码经过编译链接之后, 程序看到的内存地址就会确定下来了, 加载运行的时候就会让程序使用这一内存地址, 来保证程序可以正确运行. 一种尝试是把程序看到的内存和它运行时候真正使用的内存解耦开来. 这就是虚拟内存的思想.

所谓虚拟内存, 就是在真正的内存(也叫物理内存)之上的一层专门给进程使用的抽象. 有了虚拟内存之后, 进程只需要认为自己运行在虚拟地址上就可以了, 真正运行的时候, 才把虚拟地址映射到物理地址. 这样, 我们只要把程序链接到一个固定的虚拟地址, 加载的时候把它们加载到不同的物理地址, 并维护好虚拟地址到物理地址的映射关系, 就可以一劳永逸地解决上述问题了!

那么, 在进程运行的时候, 谁来把虚拟地址映射成物理地址呢? 我们在PA1中已经了解到指令的生命周期:

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

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

但是, 只有操作系统才知道具体要把虚拟地址映射到哪些物理地址上. 所以, 虚拟内存机制是一个软硬协同才能生效的机制: 操作系统负责进行物理内存的管理, 加载进程的时候决定要把进程的虚拟地址映射到哪些物理地址; 等到进程真正运行之前, 还需要配置MMU, 把之前决定好的映射落实到硬件上, 进程运行的时候, MMU就会进行地址转换, 把进程的虚拟地址映射到操作系统希望的物理地址. 注意到这个映射是进程相关的: 不同的进程有不同的映射, 这意味着对不同的进程来说, 同一个虚拟地址可能会被映射到不同的物理地址. 这恰好一劳永逸地解决了内存覆盖的问题. 绝大部分多任务操作系统就是这样做的.

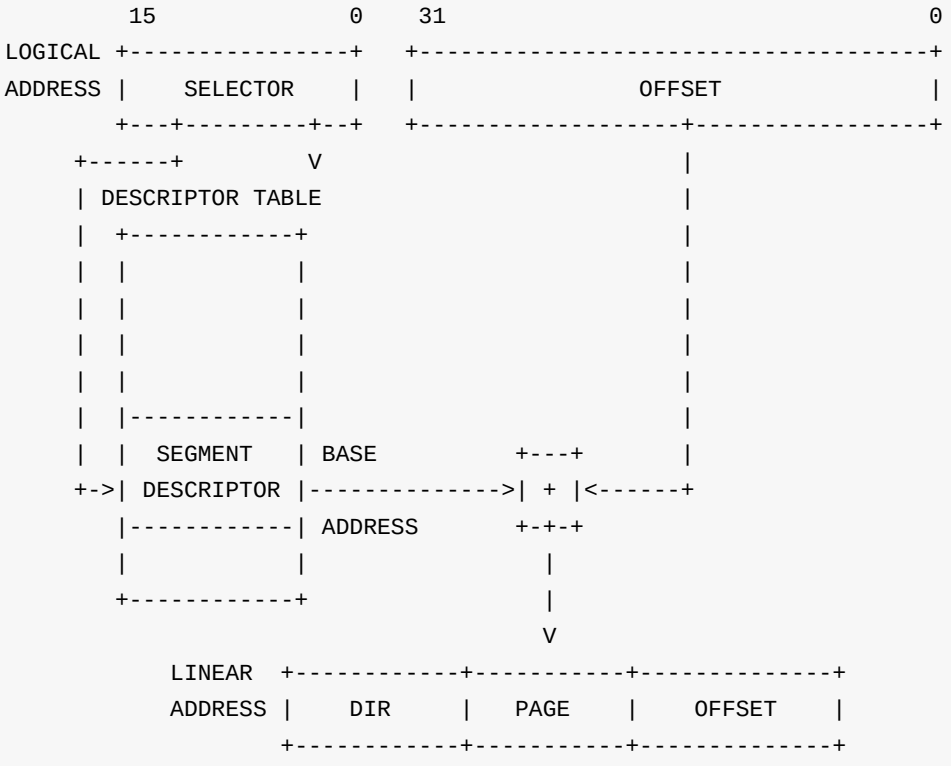
分段

关于MMU具体如何进行地址映射, 目前主要有两种主流的方式. 最简单的方法就是, 物理地址=虚拟地址+偏移量. 这种最朴素的方式就是段式虚拟内存管理机制, 简称分段机制. 从直觉上来理解, 就是把物理内存划分成若干个段, 不同的进程就放到不同的段中运行, 进程不需要关心自

已具体在哪一个段里面, 操作系统只要让不同的进程使用不同的偏移量, 进程之间就不会相互干扰了.

分段机制在硬件上的实现可以非常简单, 只需要在MMU中实现一个段基址寄存器就可以了. 操作系统在运行不同进程的时候, 就在段基址寄存器中设置不同的值, MMU会把进程使用的虚拟地址加上段基址, 来生成真正用于访问内存的物理地址, 这样就实现了"让不同的进程使用不同的段"的目的. 作为教学操作系统的Minix就是这样工作的, 一些简单的嵌入式系统和实时系统, 也是通过分段机制来进行虚存管理.

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



咋看之下真是眼花缭乱, 让人一头雾水.

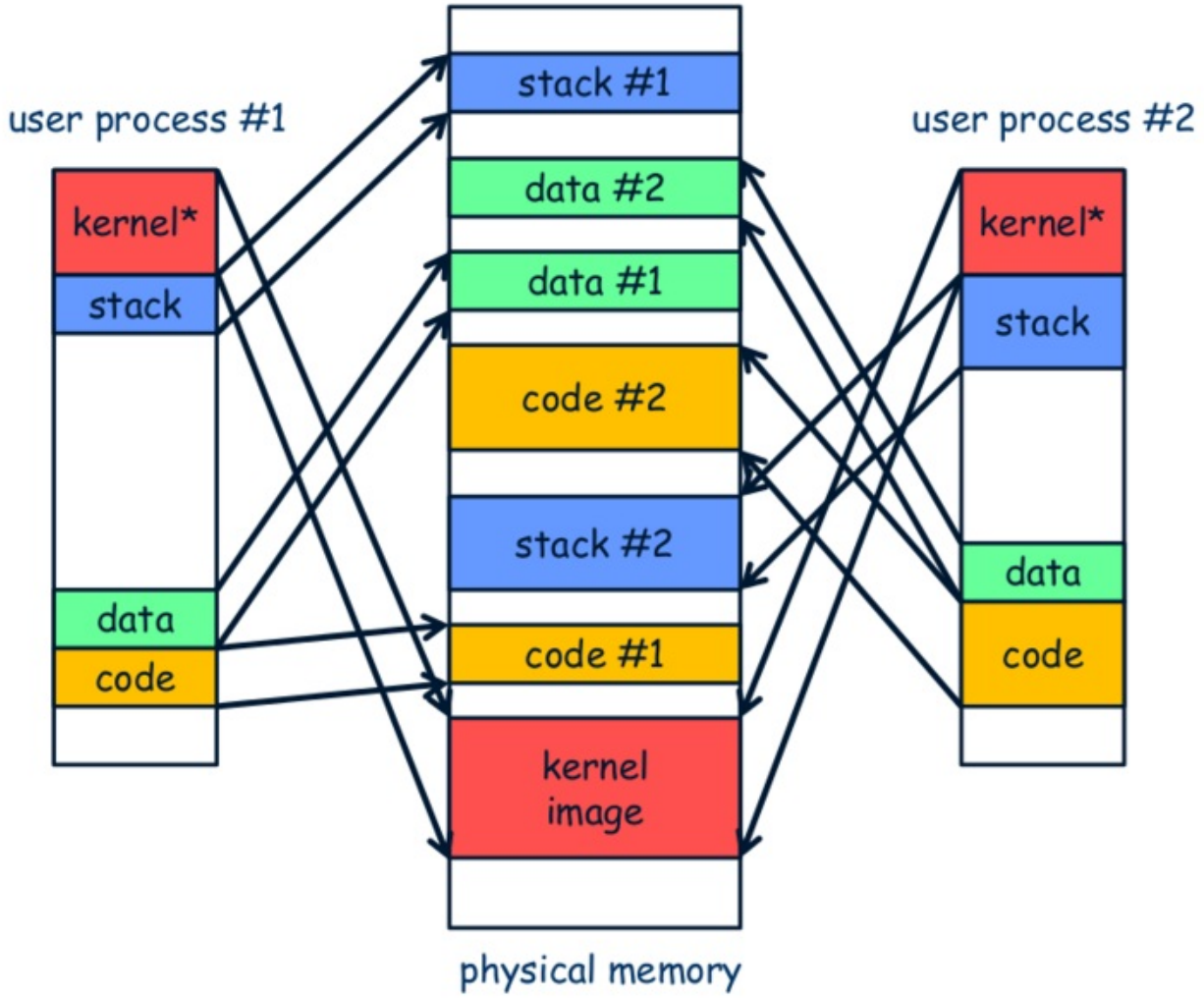
在NEMU中, 我们需要了解什么呢? 什么都不需要. 现在的大部分操作系统都不再使用分段机制, 就连i386手册中也提到可以想办法"绕过"它来提高性能: 将段基地址设成0, 长度设成4GB, 这样看来就像没有段的概念一样, 这就是i386手册中提到的"扁平模式". 当然, 这里的"绕过"并不是简单地将分段机制关掉(事实上也不可能关掉), 我们在PA3中提到的i386保护机制中关于特权级的概念, 其实就是i386分段机制提供的, 抛弃它是十分不明智的. 不过我们在NEMU中也没打算实现保护机制, 因此i386分段机制的各种概念, 我们也不会加入到NEMU中来.

超越容量的界限

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

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

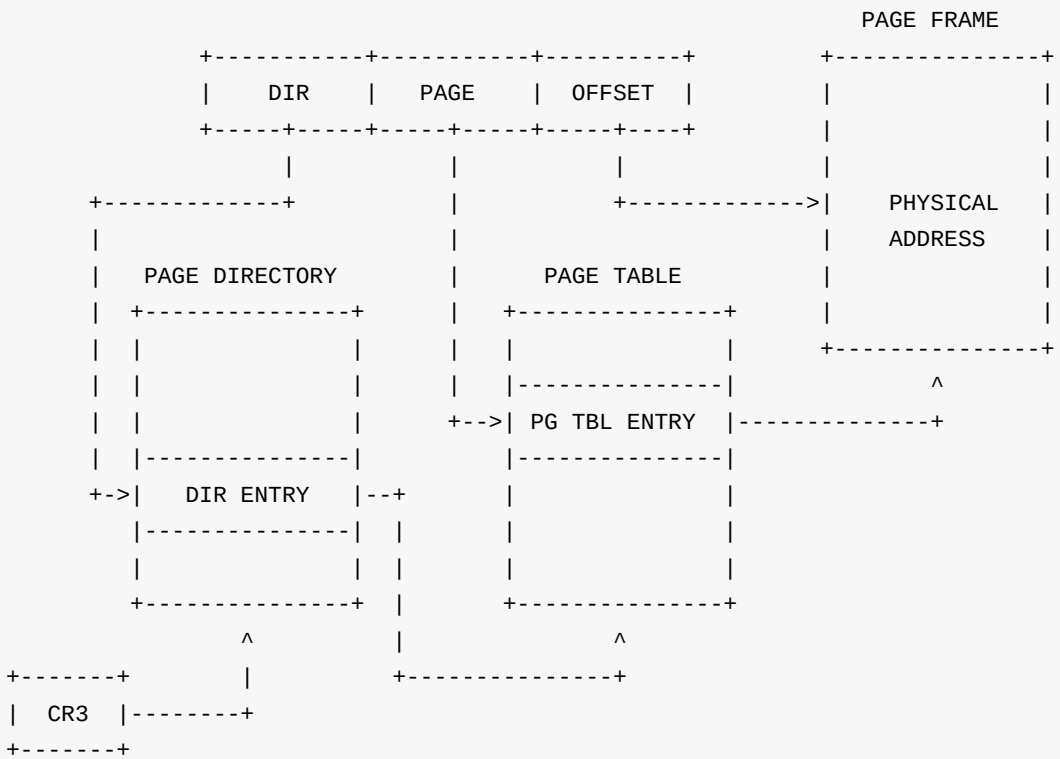
在分页机制中, 这些小片段称为页面, 在虚拟地址空间和物理地址空间中也分别称为虚拟页和物理页. 分页机制做的事情, 就是把一个个的虚拟页分别映射到相应的物理页上. 显然, 这一映射关系并不像分段机制中只需要一个段基址寄存器就可以描述的那么简单. 分页机制引入了一个叫"页表"的结构, 页表中的每一个表项记录了一个虚拟页到物理页的映射关系, 来把不必连续的页面重新组织成连续的虚拟地址空间. 因此, 为了让分页机制支撑多任务操作系统的运行, 操作系统首先需要以物理页为单位对内存进行管理. 每当加载程序的时候, 就给程序分配相应的物理页(注意这些物理页之间不必连续), 并为程序准备一个新的页表, 在页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



虚存管理中PIC的好处

我们之前提到, PIC的其中一个好处是可以将代码加载到任意内存位置执行. 如果配合虚存管理, PIC还有什么新的好处呢? (Hint: 动态库已经在享受这些好处了)

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



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

理解分页细节

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

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

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

空指针真的是"空"的吗?

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

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

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

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

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

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

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

```
paddr = page(seg(vaddr))
```

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

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

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

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

将虚存管理抽象成VME

虚存管理的具体实现自然是机器相关的, 比如在i386中用于存放页目录基地址的CR3, 在其它机器上很大概率并不叫这个名字, 访问这个寄存器的指令自然也各不相同. 再者, 不同机器中页面的大小可能会有差异, 页表项的结构也不尽相同, 更不用说有的机器还可能有多于两级的页表结构了. 于是, 我们可以将虚存管理的功能划入到AM的一类新的API中, 名字叫VME(Virtual Memory Extension).

老规矩, 我们来考虑如何将虚存管理的功能抽象成统一的API. 换句话说, 虚存机制的本质究竟是什么呢? 我们在上文已经讨论过这个问题了: 虚存机制, 说白了就是个映射(或函数). 也就是说, 本质上虚存管理要做的事情, 就是在维护这个映射. 但这个映射应该是每个进程都各自维护一份, 因此我们需要如下的两个API:

- `int _protect(_Protect *p)` 用于创建一个默认的地址空间
- `void _unprotect(_Protect *p)` 用于销毁指定的地址空间

其中 `_Protect` 是一个结构体类型, 定义了地址空间描述符的结构(在 `nexus-am/am/am.h` 中定义):

```
typedef struct _Protect {
  size_t pgsize;
  _Area area;
  void *ptr;
} _Protect;
```

其中 `pgsize` 用于指示页面的大小, `area` 表示虚拟地址空间中用户态的范围, `ptr` 则用于指示具体的映射. 在PA中, 目前只会用到 `ptr` .

有了地址空间, 我们还需要有相应的API来维护它们. 于是很自然就有了如下的API:

```
int _map(_Protect *p, void *va, void *pa, int prot);
```

它用于将地址空间 `p` 中虚拟地址 `va` 所在的虚拟页, 以 `prot` 的权限映射到 `pa` 所在的物理页. 当 `prot` 中的present位为 `0` 时, 表示让 `va` 的映射无效.

VME的主要功能已经通过上述三个API抽象出来了. 最后还有另外两个统一的API:

- `int _vme_init(void *(*pgalloc)(size_t size), void (*pgfree)(void *))` 用于进行VME相

关的初始化操作. 其中它还接受两个来自操作系统的页面分配回调函数的指针, 让AM在必要的时候通过这两个回调函数来申请/释放一页物理页.

- `_Context *_ucontext(_Protect *p, _Area ustack, _Area kstack, void *entry, void *args)` 用于创建用户进程上下文. 我们之前已经介绍过这个API, 但加入虚存管理之后, 我们需要对这个API的实现进行一些改动, 具体改动会在下文介绍.

下面我们来介绍Nanos-lite如何使用 `x86-nemu` 的VME.

在分页机制上运行Nanos-lite

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

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

为了在NEMU中实现分页机制, 你需要添加CR3寄存器和CR0寄存器, 以及相应的操作它们的指令. 对于CR0寄存器, 我们只需要实现PG位即可. 如果发现CR0的PG位为1, 则开启分页机制, 从此所有虚拟地址的访问(包括 `vaddr_read()` , `vaddr_write()`)都需要经过分页地址转换. 为此, 你需要对 `vaddr_read()` 和 `vaddr_write()` 函数作少量修改. 以 `vaddr_read()` 为例, 修改后如下:

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

你需要理解分页地址转过的过程, 然后编写 `page_translate()` 函数. 另外由于我们不打算实现保护机制, 在 `page_translate()` 函数的实现中, 你务必使用assertion检查页目录项和页表项的present位, 如果发现了一个无效的表项, 及时终止NEMU的运行, 否则调试将会异常困难. 这通

常是由于你的实现错误引起的, 请检查实现的正确性. 再次提醒, 只有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. 由于i386并没有严格要求数据对齐, 因此可能会出现数据跨越虚拟页边界的情况, 例如一条很长的指令的首字节在一个虚拟页的最后, 剩下的字节在另一个虚拟页的开头. 如果这两个虚拟页被映射到两个不连续的物理页, 就需要进行两次页级地址转换, 分别读出这两个物理页中需要的字节, 然后拼接起来组成一个完成的数据返回. MIPS作为一种RISC架构, 指令和数据都严格按照4字节对齐, 因此不会发生这样的情况, 否则MIPS CPU将会抛出异常, 可见软件灵活性和硬件复杂度是计算机科学中又一对tradeoff. 不过根据KISS法则, 你现在可以暂时不实现这种特殊情况的处理, 在判断出数据跨越虚拟页边界的情况之后, 先使用 `assert(0)` 终止NEMU, 等到真的出现这种情况的时候再进行处理.

在NEMU中实现分页机制

根据上述的讲义内容, 在NEMU中实现i386分页机制, 如有疑问, 请查阅i386手册.

让DiffTest支持分页机制

为了让DiffTest机制正确工作, 你需要

- 在 `restart()` 函数中我们需要对CR0寄存器初始化为 `0x60000011` , 但我们不必关心其含义.
- 实现分页机制中accessed位和dirty位的功能
- 处理 `attach` 命令时, 需要将CR0和CR3也同步到REF中
- 对快照功能进行更新

这毕竟是个选做题而已, 实现细节就不提示了, 遇到困难就自己思考一下解决方案吧.

在分页机制上运行用户进程

成功实现分页机制之后, 你会发现仙剑奇侠传也同样成功运行了. 但仔细想想就会发现这其实不太对劲: 我们在 `_vme_init()` 中创建了内核的虚拟地址空间, 之后就再也没有切换过这一虚拟地址空间. 也就是说, 我们让仙剑奇侠传也运行在内核的虚拟地址空间之上! 这太不合理了, 虽然NEMU没有实现ring 3, 但用户进程还是应该有自己的一套虚拟地址空间. 更可况, Navy-apps之前让用户程序链接到 `0x4000000` 的位置, 是因为之前Nanos-lite并没有对空闲的物理内存进行管理; 现在引入了分页机制, 由MM来负责所有物理页的分配. 这意味着, 如果将来MM把 `0x4000000` 所在的物理页分配出去, 仙剑奇侠传的内容将会被覆盖! 因此, 目前仙剑奇侠传看似运行成功, 其实里面暗藏杀机.

正确的做法是, 我们应该让用户进程运行在操作系统为其分配的虚拟地址空间之上. 为此, 我们需要对工程作一些变动. 首先需要将 `navy-apps/Makefile.compile` 中的链接地址 `-Ttext` 参数改为 `0x8048000` , 这是为了避免用户进程的虚拟地址空间与内核相互重叠, 从而产生非预期的错误. 同样的, `nanos-lite/src/loader.c` 中的 `DEFAULT_ENTRY` 也需要作相应的修改. 这时, "虚拟地址作为物理地址的抽象"这一好处已经体现出来了: 原则上用户进程可以运行在任意的虚拟地址, 不受物理内存容量的限制. 我们让用户进程的代码从 `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), 但分页机制保证了进程能够正确运行. 这样, 链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址, 它们只要使用虚拟地址就可以了, 而虚拟地址和物理地址之间的映射则全部交给操作系统的MM来管理.

为此, 我们需要对创建用户进程的过程进行较多的改动. 我们首先需要在加载用户进程之前为其创建地址空间. 由于地址空间是进程相关的, 我们将 `_Protect` 结构体作为PCB的一部分. 这样以后, 我们只需要在 `context_uload()` 的开头调用 `_protect()` , 就可以实现地址空间的创建. 目前这个地址空间除了内核映射之外就没有其它内容了, 具体可以参考 `nexus-am/am/arch/x86-nemu/src/vme.c` .

不过, 此时 `loader()` 不能直接把用户进程加载到内存位置 `0x8048000` 附近了, 因为这个地址并不在内核的虚拟地址空间中, 内核不能直接访问它. `loader()` 要做的事情是, 获取程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 通过 `_map()` 把这一物理页映射到用户进程的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

这一切都是为了让用户进程在将来可以正确地运行: 用户进程在将来使用虚拟地址访问内存, 在loader为用户进程维护的映射下, 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. 因此, 你需要在AM中实现 `_map()` 函数(在 `nexus-am/am/arch/x86-nemu/src/vme.c` 中定义), 你可以通过 `p->ptr` 获取页目录的基地址. 若在映射过程中发现需要申请新的页表, 可以通过回调函数 `pgalloc_usr()` 向Nanos-lite获取一页空闲的物理页.

最后, 为了让这一地址空间生效, 我们还需要将它落实到MMU中. 具体地, 我们希望在CTE恢复进程上下文的时候来切换地址空间. 为此, 我们需要将进程的地址空间描述符指针加入到上下文中. 框架代码已经实现了这一功能(见 `nexus-am/am/arch/x86-nemu/include/arch.h`), 但你还需要

- 修改 `_ucontext()` 的实现, 在创建的用户进程上下文中设置地址空间描述符指针
- 在 `irq_handle()` 的开头调用 `get_cur_as()` (在 `nexus-am/am/arch/x86-nemu/src/vme.c` 中定义), 来将当前的地址空间描述符指针保存到上下文中
- 在 `irq_handle()` 返回前调用 `_switch()` (`nexus-am/am/arch/x86-nemu/src/vme.c` 中定义) 来切换地址空间, 将调度目标进程的地址空间落实到MMU中

在分页机制上运行用户进程

根据上述的讲义内容, 对创建用户进程的过程进行相应改动, 让用户进程在分页机制上成功运行.

为了测试实现的正确性, 我们先单独运行dummy(记得修改调度代码), 并先在 `exit` 的实现中调用 `_halt()` 结束系统的运行, 这是因为让其它程序成功运行还需要进行一些额外的改动. 如果你的实现正确, 你会看到dummy程序最后输出GOOD TRAP的信息, 说明它确实在分页机制上成功运行了.

内核映射的作用

在 `_protect()` 函数中创建地址空间的时候, 有一处代码用于拷贝内核映射:

```
for (int i = 0; i < NR_PDE; i ++) {
  updir[i] = kpdirs[i];
}
```

尝试注释这处代码, 重新编译并运行, 你会看到发生了错误. 请解释为什么会发生这个错误.

为了在分页机制上运行仙剑奇侠传, 我们还需要考虑堆区的问题. 之前我们让 `mm_brk()` 函数直接返回 `0`, 表示用户进程的堆区大小修改总是成功, 这是因为在实现分页机制之前, `0x4000000` 之上的内存都可以让用户进程自由使用. 现在用户进程运行在分页机制之上, 我们还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中, 这样才能保证运行在分页机制上的用户进程可以正确地访问新申请的堆区.

为了识别堆区中的哪些空间是新申请的, 我们还需要记录堆区的位置. 由于每个进程的堆区使用情况是独立的, 我们需要为它们分别维护堆区的位置, 因此我们在PCB中添加 `cur_brk` 和 `max_brk` 两个成员, 来分别记录当前的program break位置, 以及program break曾经达到的最大位置. 引入 `max_brk` 是为了简化实现: 我们可以不实现堆区的回收功能, 而是只为当前新program break超过 `max_brk` 部分的虚拟地址空间分配物理页.

在分页机制上运行仙剑奇侠传

根据上述内容, 实现 `nanos-lite/src/mm.c` 中的 `mm_brk()` 函数. 你需要注意 `_map()` 参数是否需要按页对齐的问题(这取决于你的 `_map()` 实现).

实现正确后, 仙剑奇侠传就可以正确在分页机制上运行了.

native的VME实现

尝试阅读 `native` 的VME实现, 你发现 `native` 是如何实现VME的? 为什么可以这样做?

支持虚存管理的多道程序

绕了一大圈引入了虚存管理, 现在我们终于回来了: 我们可以支持多个用户进程的并发运行了.

支持虚存管理的多道程序

让Nanos-lite加载仙剑奇侠传和hello这两个用户进程. 如果你的实现正确, 你将可以一边运行仙剑奇侠传的同时, 一边输出hello信息. 和之前不一样, 这次的hello信息是由用户进程输出的.

修复缺页错误 (选做)

尝试通过 `context_kload()` 来加载在Nanos-lite中定义的 `hello_fun()` 函数, 来替换hello用户进程, 你应该会观察到缺页错误. 尝试定位并修复这个问题.

需要注意的是, 我们目前只允许最多一个需要更新画面的进程参与调度, 这是因为多个这样的进程并发运行会导致画面被相互覆盖, 影响画面输出的效果. 在真正的图形界面操作系统中, 通常由一个窗口管理进程来统一管理画面的显示, 需要显示画面的进程与这一管理进程进行通信, 来实现更新画面的目的. 但这需要操作系统支持进程间通信的机制, 这已经超出了ICS的范围, 而且Nanos-lite作为一个裁剪版的操作系统, 也不提供进程间通信的服务. 因此我们进行了简化, 最多只允许一个需要更新画面的进程参与调度即可.

不过我们会发现, 和之前相比, 在分页机制上运行的仙剑奇侠传的性能有了明显的下降. 尽管NEMU在串行模拟MMU的功能, 并不能完全代表硬件MMU的真实运行情况, 但这也说明了虚存机制确实会带来额外的运行时开销. 由于这个原因, 60年代工程师普遍对虚存机制有所顾虑, 不敢轻易在系统中实现虚存机制. 但"不必修改程序即可让多个程序并发运行"的好处越来越明显, 以至于虚存机制成为了现代计算机系统的标配.

支持开机菜单程序的运行 (选做)

尝试运行开机菜单程序, 你应该会发现问题. 尝试定位并修复这个问题.

由于我们没有提供任何提示, 这算是最难的一道选做题了, 供愿意挑战极限的同学尝试.

这一阶段的内容算是整个PA中最难的了, 连选做题的难度也和之前不是一个量级的. 这也展示了构建系统的挑战: 随着一个系统趋于完善, 模块之间的交互会越来越复杂, 代码看似避繁就简却可谓字字珠玑, 牵一发而动全身. 不过这是工程复杂度上升的必然规律, 等到代码量到了一定程度, 就算是开发一个应用程序, 也会面临类似的困难. 但我们要如何理解规模日趋复杂的项目代码呢?

答案是抽象. 所以你在PA中看到各种各样的API, 我们并不是随随便便定义它们的, 它们确实蕴含了模块行为的本质, 从而帮助我们更容易地从宏观的角度理解整个系统的行为, 就算是调试, 这些API对我们梳理代码的行为也有巨大的帮助. 当你在理解, 实现, 调试这些API的过程中, 你对整个系统的认识也会越来越深刻. 如果你确实独立完成到这里, 相信你以后也不会畏惧高复杂度的项目了.

温馨提示

PA4阶段2到此结束.

分时多任务

多道程序成功地实现了进程的并发执行, 但这种并发不一定是公平的. 如果一个进程长时间不触发I/O操作, 多道程序系统并不会主动将控制权切换到其它进程, 这样其它进程就得不到运行的机会. 想象一下, 如果你在开黑的时候, Windows突然在后台进行自动更新, 队友就打电话问你是不是掉线了, 你一定会非常不爽. 所以多道程序系统更多还是用在批处理的场景当中, 它能保证CPU满负荷运转, 但并不适合用于交互式的场景.

如果要用于交互式场景, 系统就要以一定的频率在所有进程之间来回切换, 保证每个进程都能及时得到响应, 这就是分时多任务. 从触发上下文切换的角度看, 分时多任务可以分成两类. 第一类是协同多任务, 它的工作方式基于一个约定: 用户进程周期性地主动让出CPU的控制权, 从而让其它进程得到运行的机会. 这件事需要操作系统提供一个特殊的系统调用, 那就是我们在PA3中实现的 `SYS_yield` . 在PA3中看似没什么用的 `SYS_yield` , 其实是协同多任务操作系统中上下文切换的基础.

说是"协同", 是因为这个机制需要所有进程一起合作, 共同遵守这个约定, 整个系统才能正确工作. 一些简单的嵌入式操作系统或者实时操作系统会采用协同多任务, 因为这些系统上运行的程序都是固定的那么几个, 让它们共同遵守约定来让出CPU并不困难. 但试想一下, 如果有一个恶意进程故意不遵守这个约定, 不调用 `SYS_yield` , 或者无意陷入了死循环, 整个系统将会被这个进程独占. 某些上古时期的Windows版本就采用了协同多任务的设计, 操作系统经常会被一些有bug的程序弄垮.

之所以协同多任务会出现这样的问题, 是系统将上下文切换的触发条件寄托在进程的行为之上. 我们知道调度一个进程的时候, 整个计算机都会被它所控制, 无论是计算, 访存, 还是输入输出, 都是由进程的行为来决定的. 为了修复这个漏洞, 我们必须寻找一种无法由进程控制的机制.

来自外部的声音

回想起我们考试的时候, 在试卷上如何作答都是我们来控制的, 但等到铃声一响, 无论我们是否完成答题, 都要立即上交试卷. 我们希望的恰恰就是这样一种效果: 时间一到, 无论正在运行的进程有多不情愿, 操作系统都要进行上下文切换. 而解决问题的关键, 就是时钟. 我们在IOE中早就已经加入了时钟了, 然而这还不能满足我们的需求, 我们希望时钟能够主动地通知处理器, 而不是被动地等着处理器来访问.

这样的通知机制, 在计算机中称为硬件中断. 硬件中断的实质是一个数字信号, 当设备有事件需要通知CPU的时候, 就会发出中断信号. 这个信号最终会传到CPU中, 引起CPU的注意.

第一个问题就是中断信号是怎么传到CPU中的. 支持中断机制的设备控制器都有一个中断引脚, 这个引脚会和CPU的INTR引脚相连, 当设备需要发出中断请求的时候, 它只要将中断引脚置为高电平, 中断信号就会一直传到CPU的INTR引脚中. 但计算机上通常有多个设备, 而CPU

引脚是在制造的时候就固定了, 因而在CPU端为每一个设备中断分配一个引脚的做法是不现实的.

为了更好地管理各种设备的中断请求, IBM PC兼容机中都会带有Intel 8259 PIC(Programmable Interrupt Controller, 可编程中断控制器). 中断控制器最主要的作用就是充当设备中断信号的多路复用器, 即在多个设备中断信号中选择其中一个信号, 然后转发给CPU.

第二个问题是CPU如何响应到来的中断请求. CPU每次执行完一条指令的时候, 都会看看INTR引脚, 看是否有设备的中断请求到来. 一个例外的情况就是CPU处于关中断状态. 在x86中, 如果EFLAGS中的IF位为0, 则CPU处于关中断状态, 此时即使INTR引脚为高电平, CPU也不会响应中断. CPU的关中断状态和中断控制器是独立的, 中断控制器只负责转发设备的中断请求, 最终CPU是否响应中断还需要由CPU的状态决定.

如果中断到来的时候, CPU没有处在关中断状态, 它就要马上响应到来的中断请求. 我们刚才提到中断控制器会生成一个中断号, CPU将会保存中断上下文, 然后根据这个中断号在IDT中进行索引, 找到并跳转到入口地址, 进行一些和设备相关的处理. 这个过程和之前提到的异常处理十分相似.

对CPU来说, 设备的中断请求何时到来是不可预测的, 在处理一个中断请求的时候到来了另一个中断请求也是有可能的. 如果希望支持中断嵌套 -- 即在进行优先级低的中断处理的过程中, 响应另一个优先级高的中断 -- 那么堆栈将是保存中断上下文信息的唯一选择. 如果选择把上下文信息保存在一个固定的地方, 发生中断嵌套的时候, 第一次中断保存的上下文信息将会被优先级高的中断处理过程所覆盖, 从而造成灾难性的后果.

灾难性的后果(这个问题有点难度)

> 假设硬件把中断信息固定保存在内存地址 `0x1000` 的位置, AM也总是从这里开始构造上下文. 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.

抢占多任务

分时多任务的第二类就是抢占多任务, 它基于硬件中断(通常是时钟中断)强行进行上下文切换, 让系统中的所有进程公平地轮流运行. 在抢占多任务操作系统中, 中断是其赖以生存的根基: 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,

在NEMU中, 我们只需要添加时钟中断这一种中断就可以了. 由于只有一种中断, 我们也不需要通过中断控制器进行中断的管理, 直接让时钟中断连接到CPU的INTR引脚即可, 我们也约定时钟中断的中断号是 `32` . 时钟中断通过 `nemu/src/device/timer.c` 中的 `timer_intr()` 触发, 每10ms触发一次. 触发后, 会调用 `dev_raise_intr()` 函数(在 `nemu/src/cpu/intr.c` 中定义). 你需要:

- 在cpu结构体中添加一个 `bool` 成员 `INTR` .
- 在 `dev_raise_intr()` 中将INTR引脚设置为高电平.
- 在 `exec_wrapper()` 的末尾添加轮询INTR引脚的代码, 每次执行完一条指令就查看是否有硬件中断到来:

```
#define IRQ_TIMER 32

if (cpu.INTR & cpu.eflags.IF) {
  cpu.INTR = false;
  raise_intr(IRQ_TIMER, cpu.eip);
  update_eip();
}
```

- 修改 `raise_intr()` 中的代码, 在保存EFLAGS寄存器后, 将其IF位置为 `0` , 让处理器进入关中断状态.

在软件上, 你还需要:

- 在CTE中添加时钟中断的支持, 将时钟中断打包成 `_EVENT_IRQ_TIMER` 事件.
- Nanos-lite收到 `_EVENT_IRQ_TIMER` 事件之后, 调用 `_yield()` 来强制当前进程让出CPU, 同时也可以去掉我们之前在设备访问中插入的 `_yield()` 了.
- 为了可以让处理器在运行用户进程的时候响应时钟中断, 你还需要修改 `_ucontext()` 的代码, 在构造上下文的时候, 设置正确的EFLAGS.

实现抢占多任务

根据讲义的上述内容, 添加相应的代码来实现抢占式的分时多任务.

为了测试时钟中断确实在工作, 你可以在Nanos-lite收到 `_EVENT_IRQ_TIMER` 事件后用 `Log()` 输出一句话.

硬件中断与基础设施

目前的各种基础设施并不支持硬件中断, 具体地, 对DiffTest来说, 我们无法直接给QEMU注入时钟中断, 从而无法保证在中断到来时QEMU与NEMU处于相同的状态; 此外, `native` 的AM目前也并未实现中断相关的功能, 抢占多任务的操作系统在 `native` 上将无法运行.

不过, 伴你一路走来的基础设施, 相信也已经帮你排除了绝大部分的bug了, 接下来的一小段路就试试靠自己吧. 如果实在搞不定, 可以想想怎么造个有用的轮子.

其实原则上这些功能都是可以实现的, 只是PA4发布的deadline要来了, 只好赶紧上线(溜). 有兴趣的同学可以思考一下如何实现它们.

基于时间片的进程调度

在抢占多任务操作系统中, 由于时钟中断以固定的速率到来, 时间被划分成长度均等的时间片, 这样系统就可以进行基于时间片的进程调度了.

优先级调度

> 我们可以修改 `schedule()` 的代码, 给仙剑奇侠传分配更多的时间片, 使得仙剑奇侠传调度若干次, 才让hello程序调度1次. 这是因为hello程序做的事情只是不断地输出字符串, 我们只需要让hello程序偶尔进行输出, 以确认它还在运行就可以了.

我们会发现, 给仙剑奇侠传分配更多的时间片之后, 其运行速度有了一定的提升. 这其实再次向我们展现了"分时"的本质: 程序之间只是轮流使用处理器, 它们并不是真正意义上的"同时"运行.

真实系统中的调度问题比上面仙剑奇侠传的例子要复杂得多. 比如阿里巴巴的双十一, 全国一瞬间向阿里巴巴的服务器发起购物请求, 这些请求最终会被转化成上亿个进程在成千上万台服务器中被处理. 在数据中心中如何调度数量如此巨大的进程, 来尽可能提高用户的服务质量, 是阿里巴巴一直都面临的严峻挑战.

抢占和并发

如果没有中断的存在, 计算机的运行就是完全确定的. 根据计算机的当前状态, 你完全可以推断出下一条指令执行后, 甚至是执行100条指令后计算机的状态. 中断作为计算机的一种外部输入, 除了作为抢占多任务操作系统的根基之外, 其不确定性也给计算机带来了不少好玩的特性. 比如GNU/Linux内核会维护一个entropy pool, 用于收集系统中的熵(不确定性). 每当中断到来的时候, 就会给这个pool添加一些熵. 通过这些熵, 我们就可以生成真正的随机数了, `/dev/random` 就是这样做的. 有了真正的随机数, 恶意程序的攻击也变得相对困难了(比如ASLR), 系统的安全也多了一分保障.

但另一方面, 中断的存在也不得不让程序在一些问题的处理上需要付出额外的代价. 由于中断随时可能会到来, 如果两个进程有一个共享的变量 `v` (比如迅雷多线程下载共享同一个文件缓冲区), 一个进程 `A` 往 `v` 中写 `0`, 刚写完中断就到来, 但当下次 `A` 再次运行的时候, `v` 的值就可能不再是 `0` 了. 从某种程度上来说, 这也是并发惹的祸: 可能进程 `B` 在并发地修改 `v` 的值, 但 `A` 却不知情. 这种由于并发造成的bug, 还带有不确定性的烙印: 如果中断没到来, 就不会触发这个bug了. 所以这种bug又叫Heisenbug, 和测不准原理类似, 你想调试它的时候, 它可能就不出现了.

不但是用户进程之间可能会有共享变量, 操作系统内核更是并发bug的重灾区. 比如并发写同一个文件的多个用户进程会共享同一个文件偏移量, 如果处理不当, 可能就会导致写数据丢失. 更一般地, 用户进程都会并发地执行系统调用, 操作系统还需要保证它们都能按照系统调用的语义正确地执行.

Nanos-lite与并发bug(建议二周目/学完操作系统课思考)

思考一下, 目前Nanos-lite的设计会导致并发bug吗? 为什么?

啊, 还是不剧透那么多了, 大家到了操作系统课再慢慢体(xiang)会(shou)乐(tong)趣(ku)吧, 也许到时候你就会想念PA的好了. 不过也用不着太沮丧, 有问题的地方, 就有解决方案; 万一没有, 那就想一个.

编写不朽的传奇

在PA的最后, 我们来做些好玩的事情.

Navy-apps之上的AM

一个环境只要能支撑AM API的实现, AM就可以运行在这一环境之上. 我们在PA3中已经使用libc和libndl实现了TRM和IOE的对应功能, 但并没有严格按照AM的API来实现它们. 如果把libc和libndl的功能按照AM的API包装成一台机器, 那么AM上的各种应用也可以运行在Navy-apps之上了.

框架代码已经准备好相应的功能了. 在 `nexus-am/am/arch/x86-navy` 目录下已经准备了用于运行在Navy-apps之上的AM的框架, 但并未实现任何API. 你需要在 `x86-navy` 中实现TRM和IOE的API, 这样就能支持绝大多数AM应用在Navy-apps上运行了.

我们在 `navy-apps/apps/am-apps` 目录下准备了一个 `Makefile` 脚本, 其中的 `APPS` 和 `TESTS` 变量可以用于控制对哪些AM应用进行编译, 这些AM应用会被编译到 `navy-apps/fsimg/bin/*-am` , 最后被包含到Nanos-lite的ramdisk中. 为了方便地使用这个脚本, 我们可以把 `am-apps` 加入到Navy-apps的编译列表中:

```
--- navy-apps/Makefile
+++ navy-apps/Makefile
@@ -3,2 +3,2 @@
-APPS = init pal litenes
+APPS = init pal litenes am-apps
 TESTS = bmp dummy events hello text
```

然后在 `nanos-lite` 目录下更新ramdisk即可. 这样以后, 你就可以在Nanos-lite中加载这些AM应用了; 如果你之前成功运行了开机菜单程序, 可以修改 `navy-apps/apps/init/init.cpp` 中的代码, 将这些AM应用加入到菜单中.

实现Navy-apps上的AM (选做)

为 `x86-navy` 实现TRM和IOE, 然后在Navy-apps中运行AM应用.

不过在运行microbench的时候会出现 `_start()` 并不位于 `0x8048000` 的情况, 你需要在编译选项中加入 `-fno-reorder-functions` 来修复这个问题. 什么? 不知道在哪个文件中加入? 那就自己了解一下项目的结构吧, 毕竟PA都快结束了, 现在还不知道项目结构就有点说不过去了.

如何在Navy-apps上运行Nanos-lite?

既然能在Navy-apps上运行基于AM的打字游戏, 那么为了炫耀, 在Navy-apps上运行Nanos-lite也并不是不可能的. 思考一下, 如果想在Navy-apps上实现CTE和VME, 我们还需要些什么呢?

诞生于未来的游戏

AM的精彩之处不仅在于可以方便地支持新机器, 加入新应用也是顺手拈来. 你的学长学姐在他们的OS课上编写了一些基于AM的小游戏, 由于它们的API并未发生改变, 我们可以很容易地把这些小游戏移植到PA中来. 当然下学期的OS课你也可以这样做.

我们在

```
https://github.com/NJU-ProjectN/oslab0-collection
```

中收录了部分游戏, 你可以在`ics2018`工程目录下通过`git clone`获得游戏代码. 由于我们已经设置好了`AM_HOME`环境变量, 你就可以直接编译并运行了. 为了将它们自动收录到Nanos-lite的ramdisk中, 你可以进行以下修改:

```
--- navy-apps/am-apps/Makefile
+++ navy-apps/am-apps/Makefile
@@ -3,2 +3,2 @@
-APPS = coremark dhrystone hello microbench slider typing litenes
+APPS = coremark dhrystone hello microbench slider typing litenes ../../oslab0-collect
ion
 TESTS = videotest
```

这样以后, 你就可以根据上一小节的方法来将这些游戏运行在Navy-apps之上了.

RTFSC???

机智的你也许会想: 哇塞, 下学期的oslab0我不就有优秀代码可以参考了吗? 不过我们已经对发布的代码进行了某种特殊的处理. 在沮丧之余, 不妨思考一下, 如果要你来实现这一特殊的处理, 你会如何实现? 这和PA1中的表达式求值有什么相似之处吗?

展示你的计算机系统

目前Nanos-lite中最多可以运行4个进程, 我们可以把这4个进程全部用满. 具体地, 我们可以加载仙剑奇侠传, 打字游戏, slider和hello程序, 然后通过一个变量`fg_pcb`来维护当前的前台程序, 让前台程序和hello程序分时运行. 具体地, 我们可以在Nanos-lite的`events_read()`函数中

让 `F1`, `F2`, `F3` 这3个按键来和3个前台程序绑定, 例如, 一开始是仙剑奇侠传和hello程序分时运行, 按下 `F3` 之后, 就变成slider和hello程序分时运行. 如果你没有实现Navy-apps之上的AM, 可以加载3份仙剑奇侠传, 让它们分别读取不同的存档进行游戏.

特别地, 如果你之前成功运行了开机菜单程序, 现在就可以加载3份开机菜单程序并绑定各自的按键. 在VME的支持下, 你可以在这3个开机菜单程序之间自由切换, 就好像在同时玩3个红白机游戏一样, 很酷!

展示你的计算机系统

添加前台程序及其切换功能, 展示你亲手创造的计算机系统.

必答题

分时多任务的具体过程 请结合代码, 解释分页机制和硬件中断是如何支撑仙剑奇侠传和hello程序在我们的计算机系统(Nanos-lite, AM, NEMU)中分时运行的.

温馨提示

PA4到此结束. 请你编写好实验报告(不要忘记在实验报告中回答必答题), 然后把命名为 `学号.pdf` 的实验报告文件放置在工程目录下, 执行 `make submit` 对工程进行打包, 最后将压缩包提交到指定网站.

欢迎二周目

你在一周目的PA中学到了什么? 二周目的时候就知道了: 二周目很顺利的部分就是你已经掌握的内容.

世界诞生的故事 - 终章

感谢你帮助先驱创造了这个美妙的世界! 同时也为自己编写了一段不朽的传奇! 也希望你可以和我们分享成功的喜悦! ^_^

故事到这里就告一段落了, PA也将要结束, 但对计算机的探索并没有终点. 如果你想知道这个美妙世界后来的样子, 可以翻一翻IA-32手册. 又或许, 你可以通过从先驱身上习得的创造力, 来改变这个美妙世界的轨迹, 书写故事新的篇章.

从一到无穷大

事实上, 计算机系统的工作只做两件事情:

- make things work
- make things work better

PA让大家体会了一次make things work的全过程. 具体地, PA以AM为主线, 从零开始展示了硬件, 运行时环境(操作系统)和应用程序之间同步演进的关系:

- TRM - 硬件有指令和内存, 几乎无需任何运行时环境, 就能运行所有的可计算算法
- IOE - 硬件添加输入输出, 运行时环境提供设备访问的抽象, 就能运行单一的交互式程序
- CTE - 硬件添加异常处理, 运行时环境添加上下文管理, 就可以支持批处理系统, 让用户程序顺序地自动运行
- VME - 硬件添加虚存机制, 运行时环境添加虚存管理, 就可以支持分时多任务, 让多个用户程序分时运行

这就是PA的全部内容.

当然, 还有各种原则和方法

除了"程序如何在计算机中执行", PA还展示了很多做事的原则和方法, 现在的你能回想起哪些内容呢?

如果你想不起来, 或者无法回忆起"程序如何在计算机中执行"的每一处细节, 那么也许你应该尝试一下独立完成二周目了. 坚持独立完成, 你将会对这些问题有全新的理解.

提示

既然你看到这里了, 就给你一个也许算不上奖励的奖励吧.

目前加入硬件中断之后, 各种基础设施就失效了. yzh在测试的时候曾经遇到过"加入硬件中断之后Nanos-lite随机触发assertion fail"的情况. 为了解决这个bug, yzh运用了以下原则: 对同一个用户进程来说, 是否加入硬件中断并不会影响其执行流. 然后把这个原则用于DiffTest中: 把加入中断前的用户进程指令序列作为REF, 加入中断后的用户进程指令序列作为DUT, 将两者进行比较, 总共花费了yzh约半小时就修复了这个原因很弱智的bug.

这个例子展示了自制轮子所需的必要条件:

- 明白"程序如何在计算机中执行"(中断对程序的影响)
- 随心所欲根据自己的需要对代码进行改造(如何抓取用户进程的指令序列)

如果你也在加入硬件中断之后遇到了调试困难, 但并没有想到上面的方法(无论这个方法对你是否有效), 那么也许你应该尝试一下独立完成二周目了.

但为了make things work better, 各种各样的技术被开发出来.

交互时代

就硬件而言, 摩尔定律预测了集成电路工艺的发展规律, 电路的集成度会越来越高. 这对计算机来说可谓是免费的午餐: 即使硬件架构师和软件设计师什么都不做, 芯片中的电路也会越来越小, 越来越快, 越来越省电. 电路工艺的发展直接推动了工作站和PC机的诞生, 搭载小型芯片的PC机开始走进寻常百姓家. 这意味着计算机不再像40-60年代那样只能进行核弹模拟, 密码破译等军事任务了, 它需要与普通用户进行交互, 于是需要一个支持交互的操作系统来方便用户使用, 当时最著名的就要数Unix和DOS了. 伴随着Unix的出现, 还有另一个流传至今的项目, 那就是C语言. 有了C语言编译器, 一些专业用户就可以开发自己的程序了. 办公和编程, 算是那个时代最普遍的应用了.

见证着摩尔定律的快速增长, 显然硬件架构师们并不会置身事外. 电路越来越小, 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. 从硬件微结构设计来看, 提升性能的一条路线是提高并行度, 从多级流水线, 超标量, 到乱序执行, SIMD, 超线程... CPU的并行度被逐渐挖掘出来; 提升性能的另一条路线是利用局部性, 于是有了cache, TLB, 分支预测, 以及相应的替换算法和预取算法... 在这些技术的支撑下, CPU等待数据的时间大幅降低, 从而能投入更多的时间来进行计算. 每当这些新名词出现, 都能把计算机的性能往上推. 计算机越来越快, 就能做越来越复杂的事情了. 一个例子是大大降低了计算机使用门槛的GUI, PC机上的分时多任务操作系统越来越普遍, 音乐, 视频, 游戏等休闲娱乐的应用开始大规模出现.

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.

上面的微结构技术都是在让一个处理器的性能变得更强(scale up), 而大型机和超级计算机早就用上了这些技术了, 它们在另一个维度上让自己变得更强: 通过片上互联技术(interconnect)把更多的处理器连接起来, 让它们协同工作(scala out). 于是有了SMP, ccNUMA, MPP, 以及各种互联拓扑结构. 同时为了高效应对多处理器架构中的数据共享问题, cache一致性和内存一致性也逐渐进入人们的视野. 为了适配片上互联技术, 可以管理众多处理器资源的集中式操作系统被设计出来, 随之而来的还有新的编程模型, 用于在这些强大的计算机上开发并运行应用. 过去诸如大气模拟, 物理建模等科学计算应用, 如今可以更快更好地在这些超级计算机上运行. 但片上互联技术要扩展规模较为困难, 大规模的处理器互联成本非常昂贵, 一般只有超级计算机能承受得起.

互联时代

如果没有互联网的普及, 计算机的发展估计就这样到头了. TCP/IP协议的标准化解决了互联网的传输控制问题, 让互联网从局域网迈向全球, 真正的因特网从此诞生. 因特网让人们变成了地球村的村民, 只要一根网线, 就可以轻而易举地访问到世界上任何一个角落的计算机. 另一方

面, HTTP和URL的出现则真正让因特网广为人们所知, 网页浏览, 电子邮件, 文件传输, 即时语音等网络应用开始流行, 通过PC机的屏幕, 用户可以看到全世界.

除了PC机, 互联网的出现也让机器之间的互联成为了可能, 于是有了集群. 而集群需要一个可以管理众多计算机资源的分布式操作系统, 相应的分布式编程模型也开始出现, 过去在一个计算机上运行的任务, 现在可以编写相应的分布式版本来在集群中运行. 集群的出现一下子解决了系统扩展性的成本问题, 只要几个廉价的计算机和几根网线, 就能组成一个集群. 从此, 互联网公司如雨后春笋般诞生, 人类开始迈进互联时代.

集群的规模进一步扩大, 就成了数据中心. 由于集群实在太容易构建了, 有大量的机器却缺少可以运行的程序. 另一方面, 摩尔定律并没有停下脚步, 为了进一步挖掘处理器的并行度, 多核架构开始出现. 集群和多核架构造成了计算资源的供过于求, 数据中心的利用率非常低. 于是虚拟化技术诞生了, 在hypervisor的支持下, 一个计算机可以同时运行多个不同的操作系统. 这还不够, 为了进一步提高集群的利用率, 互联网公司开始以虚拟机为单位出租给客户, 让客户在其上自由运行自己的系统和应用. 后来, 硬件厂商在芯片中新增了虚拟化的硬件支持, 降低了虚拟化技术的性能开销, 虚拟机的性能和真机越来越接近. 云计算的大门从此打开, 越来越多的应用开始移动到云端.

此外, iPhone的诞生重新定义了手机的概念, 手机从过去的通讯工具变成了一台名副其实的计算机. 于是手机处理器和手机操作系统再次进入人们的视野, 和PC机相比, 手机上的处理器和操作系统还需要面对功耗问题的严峻考验. 手机应用开始大量涌现, 包括即时通信和移动支付在内的各种手机应用已经改变了用户的习惯. 各种应用也开始采用端云结合的运行模式, 现在我们在手机上点击一下微信应用的图标, 登录请求就会通过网络传输到云端, 被数据中心中的上百台服务器进行处理, 最后我们才会看到登录界面.

融合时代

互联网把人和计算机连接起来, 但这并没有结束. 摩尔定律还在发展, 处理器面积进一步减小, 以前把设备接入计算机, 现在可以把计算机嵌入到设备中了. 于是各种小型设备和物品也可以接入到互联网中, 这就是物联网.

随着接入互联网的人机物越来越多, 这些终端产生的数据也日益增加, 大数据时代拉开了序幕: 据IBM统计, 在2012年, 每天大约产生2.5EB(1EB = 10^6TB)的数据. 为了处理这些大数据, 人工智能和机器学习的概念再次流行起来, 但这一下子带来了海量的数据处理任务, 于是GPU和FPGA也加入到这场大数据的持久战当中来. 还有一支奇兵就是ASIC, 和CPU不同, ASIC是一个专用电路, 可以对某一应用进行深度优化. 以AI芯片为例, 与CPU相比, AI芯片可以提升成千上万倍的性能. AlphaGo的横空出世让人们感受到了人工智能的强大, 现在人工智能芯片也已经开始集成到手机中了, 人们也逐渐发现照片越拍越漂亮, 应用越用越智能.

在这个时代, 貌似没什么是不可能的. 计算机, 用户, 网络, 设备, 云, 数据, 智能... 这些概念之间的边界变得越来越模糊, 直接和应用打交道的用户, 也越来越感受不到操作系统和硬件的存在. 这反而说明了, 那些一脉相承的计算机基本原理已经在时代的发展中留下了挥之不去的烙印.

万变之宗 - 重新审视计算机

什么是计算机? 为什么看似平淡无奇的机械, 竟然能够搭建出如此缤纷多彩的计算机世界? 那些酷炫的游戏画面, 究竟和冷冰冰的电路有什么关系?

"一"究竟起源于何处? "无穷"又会把我们带到怎么样的未来? 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.

PA5 - 天下武功唯快不破: 程序与性能

世界诞生的故事 - 外传

先驱已经创造了一个功能齐全现代计算机, 终于可以来思考计算机系统领域中扩日持久的终极问题了: 如何让程序跑得更快?

提交要求

PA5为选做实验, 不计入PA成绩.

浮点数的支持

我们已经在PA3中把仙剑奇侠传运行起来了, 但却不能战斗, 这是因为还有一些浮点数相关的工作需要处理. 现在到了处理的时候了. 要在NEMU中实现浮点指令也不是不可能的事情. 但实现浮点指令需要涉及x87架构的很多细节, 根据KISS法则, 我们选择了一种更简单的方式: 我们通过整数来模拟实数的运算, 这样的方法叫binary scaling.

我们先来说明如何用一个32位整数来表示一个实数. 为了方便叙述, 我们称用binary scaling方法表示的实数的类型为 `FLOAT` . 我们约定最高位为符号位, 接下来的15位表示整数部分, 低16位表示小数部分, 即约定小数点在第15和第16位之间(从第0位开始). 从这个约定可以看到, `FLOAT` 类型其实是实数的一种定点表示.

```
31  30                 16                   0
+----+------------------+--------------------+
|sign|      integer     |      fraction      |
+----+------------------+--------------------+
```

这样, 对于一个实数 `a` , 它的 `FLOAT` 类型表示 `A = a * 2^16` (截断结果的小数部分). 例如实数 `1.2` 和 `5.6` 用 `FLOAT` 类型来近似表示, 就是

```
1.2 * 2^16 = 78643 = 0x13333
+----+------------------+--------------------+
| 0  |         1        |        3333        |
+----+------------------+--------------------+


5.6 * 2^16 = 367001 = 0x59999
+----+------------------+--------------------+
| 0  |         5        |        9999        |
+----+------------------+--------------------+
```

而实际上, 这两个 `FLOAT` 类型数据表示的数是:

```
0x13333 / 2^16 = 1.19999695
0x59999 / 2^16 = 5.59999084
```

对于负实数, 我们用相应正数的相反数来表示, 例如 `-1.2` 的 `FLOAT` 类型表示为:

```
-(1.2 * 2^16) = -0x13333 = 0xfffecccd
```

比较FLOAT和float

`FLOAT` 和 `float` 类型的数据都是32位, 它们都可以表示2^32个不同的数. 但由于表示方法不一样, `FLOAT` 和 `float` 能表示的数集是不一样的. 思考一下, 我们用 `FLOAT` 来模拟表示 `float` , 这其中隐含着哪些取舍?

接下来我们来考虑 `FLOAT` 类型的常见运算, 假设实数 `a` , `b` 的 `FLOAT` 类型表示分别为 `A` , `B` .

- 由于我们使用整数来表示 `FLOAT` 类型, `FLOAT` 类型的加法可以直接用整数加法来进行:

  ```
  A + B = a * 2^16 + b * 2^16 = (a + b) * 2^16
  ```

- 由于我们使用补码的方式来表示 `FLOAT` 类型数据, 因此 `FLOAT` 类型的减法用整数减法来进行.

  ```
  A - B = a * 2^16 - b * 2^16 = (a - b) * 2^16
  ```

- `FLOAT` 类型的乘除法和加减法就不一样了:

  ```
  A * B = a * 2^16 * b * 2^16 = (a * b) * 2^32 != (a * b) * 2^16
  ```

  也就是说, 直接把两个 `FLOAT` 数据相乘得到的结果并不等于相应的两个浮点数乘积的 `FLOAT` 表示. 为了得到正确的结果, 我们需要对相乘的结果进行调整: 只要将结果除以 `2^16` , 就能得出正确的结果了. 除法也需要对结果进行调整, 至于如何调整, 当然难不倒聪明的你啦.
- 如果把 `A = a * 2^16` 看成一个映射, 那么在这个映射的作用下, 关系运算是保序的, 即 `a <= b` 当且仅当 `A <= B` , 故 `FLOAT` 类型的关系运算可以用整数的关系运算来进行.

有了这些结论, 要用 `FLOAT` 类型来模拟实数运算就很方便了. 除了乘除法需要额外实现之外, 其余运算都可以直接使用相应的整数运算来进行. 例如

```
float a = 1.2;
float b = 10;
int c = 0;
if (b > 7.9) {
  c = (a + 1) * b / 2.3;
}
```

用 `FLOAT` 类型来模拟就是

```
FLOAT a = f2F(1.2);
FLOAT b = int2F(10);
int c = 0;
if (b > f2F(7.9)) {
  c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

其中还引入了一些类型转换函数来实现和 `FLOAT` 相关的类型转换.

仙剑奇侠传的框架代码已经用 `FLOAT` 类型对浮点数进行了相应的处理. 你还需要实现一些和 `FLOAT` 类型相关的函数:

```
/* navy-apps/apps/pal/include/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);
/* navy-apps/apps/pal/src/FLOAT/FLOAT.c */
FLOAT f2F(float a);
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
```

其中 `F_mul_int()` 和 `F_div_int()` 用于计算一个 `FLOAT` 类型数据和一个整型数据的积/商, 这两种特殊情况可以快速计算出结果, 不需要将整型数据先转化成 `FLOAT` 类型再进行运算.

事实上, 我们并没有考虑计算结果溢出的情况, 不过仙剑奇侠传中的浮点数结果都可以在 `FLOAT` 类型中表示, 所以你可以不关心溢出的问题. 如果你不放心, 你可以在上述函数的实现中插入assertion来捕捉溢出错误.

实现binary scaling

实现上述函数来在仙剑奇侠传中对浮点数操作进行模拟. 实现正确后, 你就可以在仙剑奇侠传中成功进行战斗了.

通往高速的次元

恭喜你! 你亲手一砖一瓦搭建的计算机世界可以运行真实的程序, 确实是一个了不起的成就! 不过通常来说, 仙剑奇侠传会运行得比较慢, 现在是时候对NEMU进行优化了.

说起优化, 不知道你有没有类似的经历: 辛辛苦苦优化了一段代码, 结果发现程序的性能并没有得到明显的提升. 事实上, Amdahl's law早就看穿了这一切: 如果优化之前的这段代码只占程序运行总时间的很小比例, 即使这段代码的性能被优化了成千上万倍, 程序的总体性能也不会有明显的提升. 如果把上述情况反过来, Amdahl's law就会告诉我们并行技术的理论极限: 如果一个任务有5%的时间只能串行完成(例如初始化), 那么即使使用成千上万个核来进行并行处理, 完成这个任务所需要的时间最多快20倍.

跑题了... 总之, 盲目对代码进行优化并不是一种合理的做法. 好钢要用在刀刃上, Amdahl's law给你最直接的启示, 就是要优化hot code, 也就是那些占程序运行时间最多的代码. KISS法则告诉你, 不要在一开始追求绝对的完美, 一个原因就是, 在整个系统完成之前, 你根本就不知道系统的性能瓶颈会出现在哪一个模块中. 你一开始辛辛苦苦追求的完美, 对整个系统的性能提升也许只是九牛一毛, 根本不值得你花费这么多时间. 从这方面来说, 我们不得不承认KISS法则还是很有先见之明的.

那么怎样才能找到hot code呢? 一边盯着代码, 一边想"我认为...", "我觉得...", 这可不是什么靠谱的做法. 最可靠的方法当然是把程序运行一遍, 对代码运行时间进行统计. Profiler(性能剖析工具)就是专门做这些事情的.

GNU/Linux内核提供了性能剖析工具perf, 可以方便地收集程序运行的信息. 通过运行 `perf record` 命令进行信息收集:

```
perf record nemu/build/nemu nanos-lite/build/nanos-lite-x86-nemu.bin
```

如果运行时发现类似如下错误:

```
/usr/bin/perf: line 24: exec: perf_4.9: not found
E: linux-tools-4.9 is not installed.
```

请安装 `linux-tools` :

```
apt-get install linux-tools
```

通过 `perf record` 命令运行NEMU后, `perf` 会在NEMU的运行过程中收集性能数据. 当NEMU运行结束后, `perf` 会生成一个名为 `perf.data` 的文件, 这个文件记录了收集的性能数据. 运行命令 `perf report` 可以查看性能数据, 从而得知NEMU的性能瓶颈.

性能瓶颈的来源

Profiler可以找出实现过程中引入的性能问题, 但却几乎无法找出由设计引入的性能问题. NEMU毕竟是一个教学模拟器, 当设计和性能有冲突时, 为了达到教学目的, 通常会偏向选择易于教学的设计. 这意味着, 如果不从设计上作改动, NEMU的性能就无法突破上述取舍造成的障壁. 纵观NEMU的设计, 你能发现有哪些可能的性能瓶颈吗?

天下武功唯快不破

相信你也已经在NEMU中运行过microbench, 发现NEMU的性能连真机的1%都不到. 使用 `perf` 也没有发现能突破性能瓶颈的地方. 那NEMU究竟慢在哪里呢?

回想一下, 执行程序, 其实就是不断地执行程序的每一条指令: 取指, 译码, 执行, 更新 `eip` ... 在真机中, 这个过程是通过高速的门电路来实现的. 但NEMU毕竟是个模拟器, 只能用软件来实现"执行指令"的过程: 执行一次 `exec_wrapper()` , 客户程序才执行一条指令, 但运行NEMU的真机却需要执行上百条native指令. 这也是NEMU性能不高的根本原因. 为了方便叙述, 我们将"客户程序的指令"称为"客户指令". 因此, 作为软件模拟器的NEMU注定无法摆脱"用n条native指令模拟一条客户指令"的命运. 要提高NEMU的性能, 我们就只能想办法减小 `n` 了.

事实上, 模拟器的这种工作方式称为解释执行: 每一条客户指令的执行都需要经历完整的指令生命周期. 但仔细回顾一下计算机的本质, 执行指令的最终结果就是改变计算机的状态(寄存器和内存), 而真正改变状态的动作, 只有指令生命周期中的"执行"阶段, 其它阶段都是为状态的改变作铺垫: 取指是为了取到指令本身, 译码是为了看指令究竟要怎么执行, 更新 `eip` 只是为了执行下一条指令. 而且, 每次解释执行的时候, 这些辅助阶段做的事情都是一样的. 例如

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
```

每次执行到这条指令的时候, 取指都是取到相同的比特串, 译码总是发现"要将0x1234移动到 `eax` 中", 更新 `eip` 后其值也总是 `0x100005` . 反正执行客户指令的结果就是改变计算机的状态, 这不就和执行

```
mov $0x1234,(cpu.eax的地址)
```

这条native指令的效果一样吗?

这正是即时编译(JIT)的思想: 通过生成并执行native指令来直接改变(被模拟)计算机的状态. 这里的编译不再是"生成机器指令"的含义了, 而是更广义的"语言转换": 把客户程序中执行的机器语言转换成与之行为等价的native机器语言. "即时"是指"这一编译的动作并非事先进行", 而是"在客户程序执行的过程中进行", 这样的好处是, 不会被执行到的客户指令, 就不需要进行编译.

为了生成native指令, 我们至少也要知道相应的客户指令要做什么. 因此, 我们至少也要对客户指令进行一次取指和译码. 通常情况下, 客户指令不会发生变化, 因此编译成的native指令也不会发生变化. 这意味着, 我们只需要对客户指令进行一次编译, 生成相应的native指令, 然后存放起来, 将来碰到相同的客户指令, 就不必重新编译, 而是可以找到之前编译的结果直接执行了. 这恰恰就是cache的思想: 我们将native指令序列组织成一个TB(translation block), 用客户程序的 `eip` 来索引; 这个cache由一系列的TB组成; 执行客户程序的时候, 先用 `eip` 索引cache, 若

命中, 说明相应的客户指令已经被编译过了, 此时可以不必重新编译, 直接取出相应的TB并执行; 若缺失, 说明相应的客户指令还没有被编译过, 此时才需要对客户指令进行取指和译码, 并编译成相应的TB, 更新cache, 以便于将来多次执行.

一个值得考虑的问题是, 每次编译多少条客户指令比较合适? 若一次编译一条客户指令, 则会导致每执行完一条客户指令就需要重新对cache进行索引, 查看下一条客户执行是否被编译过. 细心的你会发现, 在即时编译模式中, 一条客户指令被编译过, 当且仅当它被执行过. 也就是说, NEMU在执行完一条客户执行之后, 都会去检查下一条指令有没有被执行过. 我们知道, 顺序执行是程序最基本的执行流之一. 我们很容易想到, 在一个顺序执行的模块中, 如果其中的一条指令被执行过, 那就意味着整个模块的每一条指令都已经被执行过. 这说明, 若一次编译一条客户指令, "检查下一条指令有没有被执行过"大多数时候是一个冗余的动作. 为了避免这些冗余的动作, 我们可以一次编译一个顺序执行的模块, 这样的模块称为基本块.

要如何进行编译呢? 我们知道, x86的指令集非常复杂, 如果要考虑每一条x86客户指令如何编译到native指令, 就太麻烦了. 嘿, 我们在PA2中引入的RTL就是用来解决这个问题的: RTL只有少数的基本指令, 我们只需要考虑如何将少数的RTL基本指令编译到native指令就可以了! 引入RTL还有另一个好处, 就是方便NEMU的移植:

```
           +-----+
  x86 ---> |     | ---> mips
 mips ---> | RTL | ---> arm
riscv ---> |     | ---> x86
           +-----+
|           |   |          |
+ front-end +   + back-end +
```

以RTL为分界, 我们可以把即时编译模式的NEMU分为两部分: 前端用于将客户程序的机器语言编译成RTL, 后端负责将RTL编译成native机器语言. 这样以后, 要在NEMU中支持一种新的客户程序架构x, 只需要增加相应的前端模块来将x编译成RTL即可; 要让NEMU运行在一种新的架构y, 只需要增加相应的后端模块来将RTL编译成y即可.

于是, 即时编译模式的NEMU的工作方式如下:

```
while (1) {
  tb = query_cache(cpu.eip);
  if ( cache miss ) {
    tb = jit_translate(cpu.eip); // translate a basic block
    update_cache(cpu.eip, tb);
  }

  jump_to(tb); // cpu.eip will be updated while executing tb
}
```

关于`jit_translate()`如何工作, 可以参考这篇讲述QEMU中的JIT如何实现的文章.

什么? 这就没有了?

事实上, 实现JIT的坑非常多, yzh也还没全踩完, 也就还没总结出好的要点. 即使把坑都踩过了, 拖延症患者yzh也不一定有时间把这些要点整理成讲义.

不过如果你看到这里, 相信你也有一定能力来面对这些坑了. 踩坑其实是非常非常宝贵的经验, 也是做这些项目的意义所在: 通过做项目, 知道了以前永远也不可能知道的东西. 上述文章提到, QEMU的早期版本可以做到只比真机性能慢4倍. 看着microbench的分数越来越高, 了解每一项技术带来的性能提升及其背后揭示的原理, 这些都是最好的回报, 也是系统方向科研人员所追求的奥义.

开源项目的大门已经向你敞开: 不妨尝试一下阅读QEMU的源代码 (虽然现在的QEMU已经不是上述那篇十几年前的文章所说的那个样子了). NEMU的架构也不够完美: 欢迎和yzh交流你实现JIT所踩过的坑.

这道蓝框题之后的讲义内容, 你, 也许就是作者.

为什么要学习计算机系统基础

一知半解

你已经学过 `程序设计基础` 课程了，对于C和C++程序设计已有一定的基础. 但你会发现，你可能还是不能理解以下程序的运行结果：

数组求和

```
int sum(int a[ ], unsigned len) {
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

当 `len = 0` 时，执行 `sum` 函数的for循环时会发生 `Access Violation` ，即"访问违例"异常. 但是，当参数 `len` 说明为 `int` 型时， `sum` 函数能正确执行，为什么？

整数的平方

若 `x` 和 `y` 为 `int` 型，当 `x = 65535` 时，则 `y = x*x = -131071` . 为什么？

多重定义符号

```
/*---main.c---*/
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main() {
    p1();
    printf("d=%d,x=%d\n",d,x);
    return 0;
}

/*---p1.c---*/
double d;
void p1() {
    d=1.0;
}
```

在上述两个模块链接生成的可执行文件被执行时，`main` 函数的 `printf` 语句打印出来的值是: `d=0,x=1072693248`. 为什么不是 `d=100,x=200` ?

奇怪的函数返回值

```
double fun(int i) {
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

从 `fun` 函数的源码来看, 每次返回的值应该都是 `3.14` , 可是执行 `fun` 函数后发现其结果是:

- `fun(0)` 和 `fun(1)` 为 `3.14`
- `fun(2)` 为 `3.1399998664856`
- `fun(3)` 为 `2.00000061035156`
- `fun(4)` 为 `3.14` 并会发生 `访问违例` 这是为什么?

时间复杂度和功能都相同的程序

```
void copyij(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
void copyji(int src[2048][2048], int dst[2048][2048]) {
      int i,j;
      for (j = 0; j < 2048; j++)
          for (i = 0; i < 2048; i++)
              dst[i][j] = src[i][j];
}
```

上述两个功能完全相同的函数, 时间复杂度也完全一样, 但在Pentium 4处理器上执行时, 所测时间相差大约21倍. 这是为什么? 猜猜看是 `copyij` 更快还是 `copyji` 更快?

网友贴出的一道百度招聘题

请给出以下C语言程序的执行结果, 并解释为什么.

```
#include <stdio.h>
int main() {
    double a = 10;
    printf("a = %d\n", a);
    return 0;
}
```

该程序在IA-32上运行时, 打印结果为 `a=0` ; 在x86-64上运行时, 打印出来的 `a` 是一个不确定值. 为什么?

整数除法

以下两个代码段的运行结果是否一样呢?

- 代码段一: `c int a = 0x80000000; int b = a / -1; printf("%d\n", b); C`
- 代码段二:

  ```
  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. 不要修改压缩包和工程根目录的命名, 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.
- 支持多次提交, 我们会按最新的提交来批改

git版本控制

我们鼓励你使用git管理你的项目, 如果你提交的实验中包含均匀合理的, 你手动提交的git记录(不是开发跟踪系统自动提交的), 你将会获得本次实验20%的分数奖励(总得分不超过本次实验的上限). 这里有一个十分简单的git教程, 更多的git命令请查阅相关资料. 另外, 请你不定期查看自己的git log, 检查是否与自己的开发过程相符. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据.

实验报告内容

你必须在实验报告中描述以下内容:

- 实验进度. 简单描述即可, 例如"我完成了所有内容", "我只完成了xxx". 缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验.
- 必答题.

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

- 你遇到的问题和对这些问题的思考
- 对讲义中蓝框思考题的看法
- 或者你的其它想法, 例如实验心得, 对提供帮助的同学的感谢等

认真描述实验心得和想法的报告将会获得分数的奖励; 蓝框题为选做, 完成了也不会得到分数的奖励, 但它们是经过精心准备的, 可以加深你对某些知识的理解和认识. 因此当你发现编写实验报告的时间所剩无几时, 你应该选择描述实验心得和想法. 如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求. 但请不要

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

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

Linux入门教程

以下内容引用自jyy的操作系统实验课程网站, 并有少量修改和补充. 如果你是第一次使用Linux, 请你一边仔细阅读教程, 一边尝试运行教程中提到的命令.

探索命令行

Linux命令行中的命令使用格式都是相同的:

```
命令名称 参数1 参数2 参数3 ...
```

参数之间用任意数量的空白字符分开. 关于命令行, 可以先阅读[一些基本常识][linux basics]. 然后我们介绍最常用的一些命令:

- `ls` 用于列出当前目录(即"文件夹")下的所有文件(或目录). 目录会用蓝色显示. `ls -l` 可以显示详细信息.
- `pwd` 能够列出当前所在的目录.
- `cd DIR` 可以切换到 `DIR` 目录. 在Linux中, 每个目录中都至少包含两个目录: `.` 指向该目录自身, `..` 指向它的上级目录. 文件系统的根是 `/`.
- `touch NEWFILE` 可以创建一个内容为空的新文件 `NEWFILE`, 若 `NEWFILE` 已存在, 其内容不会丢失.
- `cp SOURCE DEST` 可以将 `SOURCE` 文件复制为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件复制到该目录下.
- `mv SOURCE DEST` 可以将 `SOURCE` 文件重命名为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件移动到该目录下.
- `mkdir DIR` 能够创建一个 `DIR` 目录.
- `rm FILE` 能够删除 `FILE` 文件; 如果使用 `-r` 选项则可以递归删除一个目录. 删除后的文件无法恢复, 使用时请谨慎!
- `man` 可以查看命令的帮助. 例如 `man ls` 可以查看 `ls` 命令的使用方法. 灵活应用 `man` 和互联网搜索, 可以快速学习新的命令.

`man` 的功能不仅限于此. `man` 后可以跟两个参数, 可以查看不同类型的帮助(请在互联网上搜索). 例如当你不知道C标准库函数 `freopen` 如何使用时, 可以键入命令

```
man 3 freopen
```

学会使用man

如果你是第一次使用 `man`, 请阅读这里. 这个教程除了说明如何使用 `man` 之外, 还会教你在使用一款新的命令行工具时如何获得帮助.

消失的cd

上述各个命令除了 `cd` 之外都能找到它们的manpage, 这是为什么? 如果你思考后仍然感到困惑, 试着到互联网上寻找答案.

下面给出一些常用命令使用的例子, 你可以键入每条命令之后使用 `ls` 查看命令执行的结果:

```
$ mkdir temp      # 创建一个目录temp
$ cd temp         # 切换到目录temp
$ touch newfile   # 创建一个空文件newfile
$ mkdir newdir    # 创建一个目录newdir
$ cd newdir       # 切换到目录newdir
$ cp ../newfile . # 将上级目录中的文件newfile复制到当前目录下
$ cp newfile aaa  # 将文件newfile复制为新文件aaa
$ mv aaa bbb      # 将文件aaa重命名为bbb
$ mv bbb ..       # 将文件bbb移动到上级目录
$ cd ..           # 切换到上级目录
$ rm bbb          # 删除文件bbb
$ cd ..           # 切换到上级目录
$ rm -r temp      # 递归删除目录temp
```

更多的命令行知识

仅仅了解这些最基础的命令行知识是不够的. 通常, 我们可以抱着如下的信条: 只要我们能想到的, 就一定有方便的办法能够办到. 因此当你想要完成某件事却又不知道应该做什么的时候, 请向Google求助. 如果你想以Linux作为未来的事业, 那就可以去图书馆或互联网上找一些相关的书籍来阅读.

统计代码行数

第一个例子是统计一个目录中(包含子目录)中的代码行数. 如果想知道当前目录下究竟有多少行的代码, 就可以在命令行中键入如下命令:

```
find . | grep '\.c$\|\.h$' | xargs wc -l
```

如果用 `man find` 查看 `find` 操作的功能, 可以看到 `find` 是搜索目录中的文件. Linux中一个点 `.` 始终表示Shell当前所在的目录, 因此 `find .` 实际能够列出当前目录下的所有文件. 如果在文件很多的地方键入 `find .`,将会看到过多的文件, 此时可以按 `CTRL + c` 退出.

同样，用 `man` 查看 `grep` 的功能——"print lines matching a pattern". `grep` 实现了输入的过滤，我们的 `grep` 有一个参数，它能够匹配以 `.c` 或 `.h` 结束的文件. 正则表达式是处理字符串非常强大的工具之一，每一个程序员都应该掌握其相关的知识. 有兴趣的同学可以首先阅读一个基础的教程，然后看一个有趣的小例子: 如何用正则表达式判定素数. 正则表达式还可以用来编写一个30行的java表达式求值程序(传统方法几乎不可能)，聪明的你能想到是怎么完成的吗? 上述的 `grep` 命令能够提取所有 `.c` 和 `.h` 结尾的文件.

刚才的 `find` 和 `grep` 命令，都从标准输入中读取数据，并输出到标准输出. 关于什么是标准输入输出，请参考这里. 连接起这两个命令的关键就是管道符号 `|` . 这一符号的左右都是Shell命令，`A | B` 的含义是创建两个进程 `A` 和 `B` ，并将 `A` 进程的标准输出连接到 `B` 进程的标准输入. 这样，将 `find` 和 `grep` 连接起来就能够筛选出当前目录(`.`)下所有以 `.c` 或 `.h` 结尾的文件.

我们最后的任务是统计这些文件所占用的总行数，此时可以用 `man` 查看 `wc` 命令. `wc` 命令的 `-l` 选项能够计算代码的行数. `xargs` 命令十分特殊，它能够将标准输入转换为参数，传送给第一个参数所指定的程序. 所以，代码中的 `xargs wc -l` 就等价于执行 `wc -l aaa.c bbb.c include/ccc.h ...` ，最终完成代码行数统计.

统计磁盘使用情况

以下命令统计 `/usr/share` 目录下各个目录所占用的磁盘空间:

```
du -sc /usr/share/* | sort -nr
```

`du` 是磁盘空间分析工具，`du -sc` 将目录的大小顺次输出到标准输出，继而通过管道传送给 `sort` . `sort` 是数据排序工具，其中的选项 `-n` 表示按照数值进行排序，而 `-r` 则表示从大到小输出. `sort` 可以将这些参数连写在一起.

然而我们发现，`/usr/share` 中的目录过多，无法在一个屏幕内显示. 此时，我们可以再使用一个命令: `more` 或 `less` .

```
du -sc /usr/share/* | sort -nr | more
```

此时将会看到输出的前几行结果. `more` 工具使用空格翻页，并可以用 `q` 键在中途退出. `less` 工具则更为强大，不仅可以向下翻页，还可以向上翻页，同样使用 `q` 键退出. 这里还有一个关于less的小故事.

在Linux下编写Hello World程序

Linux中用户的主目录是 `/home/用户名称` , 如果你的用户名是 `user` , 你的主目录就是 `/home/user` . 用户的 `home` 目录可以用波浪符号 `~` 替代, 例如临时文件目录 `/home/user/Templates` 可以简写为 `~/Templates` . 现在我们就可以进入主目录并编辑文件了. 如果 `Templates` 目录不存在, 可以通过 `mkdir` 命令创建它:

```
cd ~
mkdir Templates
```

创建成功后, 键入

```
cd Templates
```

可以完成目录的切换. 注意在输入目录名时, `tab` 键可以提供联想.

你感到键入困难吗?

你可能会经常要在终端里输入类似于

```
cd AVeryVeryLongFileName
```

的命令, 你一定觉得非常烦躁. 回顾上面所说的原则之一: 如果你感到有什么地方不对, 就一定有什么好办法来解决. 试试 `tab` 键吧.

Shell中有很多这样的小技巧, 你也可以使用其他的Shell例如zsh, 提供更丰富好用的功能. 总之, 尝试和改变是最重要的.

进入正确的目录后就可以编辑文件了, 开源世界中主流的两大编辑器是 `vi(m)` 和 `emacs` , 你可以使用其中的任何一种. 如果你打算使用 `emacs` , 你还需要安装它

```
apt-get install emacs
```

`vi` 和 `emacs` 这两款编辑器都需要一定的时间才能上手, 它们共同的特点是需要花较多的时间才能适应基本操作方式(命令或快捷键), 但一旦熟练运用, 编辑效率就比传统的编辑器快很多.

进入了正确的目录后, 输入相应的命令就能够开始编辑文件. 例如输入

```
vi hello.c
或emacs hello.c
```

就能开启一个文件编辑. 例如可以键入如下代码(对于首次使用 `vi` 或 `emacs` 的同学, 键入代码可能会花去一些时间, 在编辑的同时要大量查看网络上的资料):

```
#include <stdio.h>
int main(void) {
  printf("Hello, Linux World!\n");
  return 0;
}
```

保存后就能够看到 `hello.c` 的内容了. 终端中可以用 `cat hello.c` 查看代码的内容. 如果要将它编译, 可以使用 `gcc` 命令:

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

`gcc` 的 `-o` 选项指定了输出文件的名称, 如果将 `-o hello` 改为 `-o hi` , 将会生成名为 `hi` 的可执行文件. 如果不使用 `-o` 选项, 则会默认生成名为 `a.out` 的文件, 它的含义是assembler output. 在命令行输入

```
./hello
```

就能够运行改程序. 命令中的 `./` 是不能少的, 点代表了当前目录, 而 `./hello` 则表示当前目录下的 `hello` 文件. 与Windows不同, Linux系统默认情况下并不查找当前目录, 这是因为Linux下有大量的标准工具(如 `test` 等), 很容易与用户自己编写的程序重名, 不搜索当前目录消除了命令访问的歧义.

使用重定向

有时我们希望将程序的输出信息保存到文件中, 方便以后查看. 例如你编译了一个程序 `myprog` , 你可以使用以下命令对 `myprog` 进行反汇编, 并将反汇编的结果保存到 `output` 文件中:

```
objdump -d myprog > output
```

`>` 是标准输出重定向符号, 可以将前一命令的输出重定向到文件 `output` 中. 这样, 你就可以使用文本编辑工具查看 `output` 了.

但你会发现, 使用了输出重定向之后, 屏幕上就不会显示 `myprog` 输出的任何信息. 如果你希望输出到文件的同时也输出到屏幕上, 你可以使用 `tee` 命令:

```
objdump -d myprog | tee output
```

使用输出重定向还能很方便地实现一些常用的功能, 例如

```
> empty                    # 创建一个名为empty的空文件
cat old_file > new_file  # 将文件old_file复制一份, 新文件名为new_file
```

如果 `myprog` 需要从键盘上读入大量数据(例如一个图的拓扑结构), 当你需要反复对 `myprog` 进行测试的时候, 你需要多次键入大量相同的数据. 为了避免这种无意义的重复键入, 你可以使用以下命令:

```
./myprog < data
```

`<` 是标准输入重定向符号, 可以将前一命令的输入重定向到文件 `data` 中. 这样, 你只需要将 `myprog` 读入的数据一次性输入到文件 `data` 中, `myprog` 就会从文件 `data` 中读入数据, 节省了大量的时间.

下面给出了一个综合使用重定向的例子:

```
time ./myprog < data | tee output
```

这个命令在运行 `myprog` 的同时, 指定其从文件 `data` 中读入数据, 并将其输出信息打印到屏幕和文件 `output` 中. `time` 工具记录了这一过程所消耗的时间, 最后你会在屏幕上看到 `myprog` 运行所需要的时间. 如果你只关心 `myprog` 的运行时间, 你可以使用以下命令将 `myprog` 的输出过滤掉:

```
time ./myprog < data > /dev/null
```

`/dev/null` 是一个特殊的文件, 任何试图输出到它的信息都会被丢弃, 你能想到这是怎么实现的吗? 总之, 上面的命令将 `myprog` 的输出过滤掉, 保留了 `time` 的计时结果, 方便又整洁.

使用Makefile管理工程

大规模的工程中通常含有几十甚至成百上千个源文件(Linux内核源码有25000+的源文件), 分别键入命令对它们进行编译是十分低效的. Linux提供了一个高效管理工程文件的工具: GNU Make. 我们首先从一个简单的例子开始, 考虑上文提到的Hello World的例子, 在 `hello.c` 所在目录下新建一个文件 `Makefile` , 输入以下内容并保存:

```
hello:hello.c
    gcc hello.c -o hello    # 注意开头的tab, 而不是空格

.PHONY: clean

clean:
    rm hello    # 注意开头的tab, 而不是空格
```

返回命令行, 键入 `make` , 你会发现 `make` 程序调用了 `gcc` 进行编译. `Makefile` 文件由若干规则组成, 规则的格式一般如下:

```
目标文件名:依赖文件列表
    用于生成目标文件的命令序列    # 注意开头的tab, 而不是空格
```

我们来解释一下上文中的 `hello` 规则. 这条规则告诉 `make` 程序, 需要生成的目标文件是 `hello` , 它依赖于文件 `hello.c` , 通过执行命令 `gcc hello.c -o hello` 来生成 `hello` 文件.

如果你连续多次执行 `make` , 你会得到"文件已经是最新版本"的提示信息, 这是 `make` 程序智能管理的功能. 如果目标文件已经存在, 并且它比所有依赖文件都要"新", 用于生成目标的命令就不会被执行. 你能想到 `make` 程序是如何进行"新"和"旧"的判断的吗?

上面例子中的 `clean` 规则比较特殊, 它并不是用来生成一个名为 `clean` 的文件, 而是用于清除编译结果, 并且它不依赖于其它任何文件. `make` 程序总是希望通过执行命令来生成目标, 但我们给出的命令 `rm hello` 并不是用来生成 `clean` 文件, 因此这样的命令总是会被执行. 你需要键入 `make clean` 命令来告诉 `make` 程序执行 `clean` 规则, 这是因为 `make` 默认执行在 `Makefile` 中文本序排在最前面的规则. 但如果很不幸地, 目录下已经存在了一个名为 `clean` 的文件, 执行 `make clean` 会得到"文件已经是最新版本"的提示. 解决这个问题的方法是在 `Makefile` 中加入一行 `PHONY: clean` , 用于指示" `clean` 是一个伪目标". 这样以后, `make` 程序就不会判断目标文件的新旧, 伪目标相应的命令序列总是会被执行.

对于一个规模稍大一点的工程, `Makefile` 文件还会使用变量, 函数, 调用Shell命令, 隐含规则等功能. 如果你希望学习如何更好地编写一个 `Makefile` , 请到互联网上搜索相关资料.

综合示例: 教务刷分脚本

使用编辑器编辑文件 `jw.sh` 为如下内容(另外由于教务网站的升级改版, 目前此脚本可能不能实现正确的功能):

```
#!/bin/bash
save_file="score" # 临时文件
semester=20102 # 刷分的学期, 20102代表2010年第二学期
jw_home="http://jwas3.nju.edu.cn:8080/jiaowu" # 教务网站首页地址
jw_login="http://jwas3.nju.edu.cn:8080/jiaowu/login.do" # 登录页面地址
jw_query="http://jwas3.nju.edu.cn:8080/jiaowu/student/studentinfo/achievementinfo.do?m
ethod=searchTermList&termCode=$semester" # 分数查询页面地址

name="09xxxxxxx" # 你的学号
passwd="xxxxxxxx" # 你的密码

# 请求jw_home地址, 并从中找到返回的cookie. cookie信息在http头中的JSESSIONID字段中
cookie=`wget -q -O - $jw_home --save-headers | \
    sed -n 's/Set-Cookie: JSESSIONID=\([0-9A-Z]\+\);.*$/\1/p'`
# 用户登录, 使用POST方法请求jw_login地址, 并在POST请求中加入userName和password
wget -q -O - --header="Cookie:JSESSIONID=$cookie" --post-data \
    "userName=${name}&password=${passwd}" "$jw_login" &> /dev/null
# 登录完毕后, 请求分数查询页面. 此时会返回html页面并输出到标准输出. 我们将输出重定向到文件"tmp"中.
wget -q -O - --header="Cookie:JSESSIONID=$cookie" "$jw_query" > tmp
# 获取分数列表. 因为教务网站的代码实在是实现得不太规整, 我们又想保留shell的风味, 所以用了比较繁琐的s
ed和awk处理. list变量中会包含课程名称的列表.
list=`cat tmp | sed -n '/<table.*TABLE_BODY.*>/,/<\/table>/p' \
        | sed '/<--/,/-->/d' | grep td \
        | awk 'NR%11==3' | sed 's/^.*>\(.*\)<.*$/\1/g'`
# 对list中的每一门课程, 都得到它的分数
for item in $list; do
    score=`cat tmp | grep -A 20 $item | awk "NR==18" | sed -n '/^.*\..*$/p'`
    score=`echo $score`
    if [[ ${#score} != 0 ]]; then # 如果存在成绩
        grep $item $save_file &>/dev/null # 查找分数是否显示过
        if [[ $? != 0 ]]; then # 如果没有显示过
        # 考虑到尝试的同学可能没有安装notify-send工具, 这里改成echo  -- yzh
            # notify-send "新成绩：$item $score" # 弹出窗口显示新成绩
            echo "新成绩：$item $score" # 在终端里输出新成绩
            echo $item >> $save_file # 将课程标记为已显示
        fi
    fi
done
```

运行这个例子需要在命令行中输入 `bash jw.sh`, 用bash解释器执行这一脚本. 如果希望定期运行这一脚本, 可以使用Linux的标准工具之一: `cron`. 将命令添加到crontab就能实现定期自动刷新.

为了理解这个例子, 首先需要一些HTTP协议的基础知识. HTTP请求实际就是来回传送的文本流——浏览器(或我们例子中的爬虫)生成一个文本格式的HTTP请求, 包括header和content, 以文本的形式通过网络传送给服务器. 服务器根据请求内容(header中包含请求的URL以及浏览器等其他信息), 生成页面并返回.

用户登录的实现, 就是通过HTTP头中header中的cookie实现的. 当浏览器第一次请求页面时, 服务器会返回一串字符, 用来标识浏览器的这次访问. 从此以后, 所有与该网站交互时, 浏览器都会在HTTP请求的header中加入这个字符串, 这样服务器就"记住"了浏览器的访问. 当完成登录操作(将用户名和密码发送到服务器)后, 服务器就知道这个cookie隐含了一个合法登录的帐号, 从而能够根据帐号信息发送成绩.

得到包含了成绩信息的html文档之后, 剩下的事情就是解析它了. 我们用了大量的 `sed` 和 `awk` 完成这件事情, 同学们不用去深究其中的细节, 只需知道我们从文本中提取出了课程名和成绩, 并且将没有显示过的成绩显示出来.

我们讲解这个例子主要是为了说明新环境下的工作方式, 以及实践Unix哲学:

- 每个程序只做一件事, 但做到极致
- 用程序之间的相互协作来解决复杂问题
- 每个程序都采用文本作为输入和输出, 这会使程序更易于使用

一个Linux老手可以用脚本完成各式各样的任务: 在日志中筛选想要的内容, 搭建一个临时HTTP服务器(核心是使用 `nc` 工具)等等. 功能齐全的标准工具使Linux成为工程师, 研究员和科学家的最佳搭档.

man快速入门

这是一个man的使用教程, 同时给出了一个如何寻找帮助的例子.

初识man

你是一只Linux菜鸟. 因为课程实验所迫, 你不得不使用Linux, 不得不使用十分落后的命令行. 实验内容大多数都要在命令行里进行, 面对着一大堆陌生的命令和参数, 这个链接中的饼图完美地表达了你的心情.

不行! 还是得认眞做实验, 不然以后连码农都当不上了! 这样的想法鞭策着你, 因为你知道, 就算是码农, 也要有适应新环境和掌握新工具的能力. "还是先去找man吧." 于是你在终端里输入 `man` , 敲了回车. 只见屏幕上输出了一行信息:

> What manual page do you want?

噢, 原来命令行也会说人话! 你明白这句话的意思, `man` 在询问你要查询什么内容. 你能查询什么内容呢? 既然 `man` 会说人话, 还是先多了解 `man` 吧. 为了告诉 `man` 你想更了解ta, 你输入

```
man man
```

敲了回车之后, `man` 把你带到了一个全新的世界. 这时候, 你又看到了一句人话了, 那是 `man` 的独白, ta告诉你, ta的眞实身份其实是

> an interface to the on-line reference manuals

接下来, ta忽然说了一大堆你听不懂的话, 似乎是想告诉你ta的使用方法. 可是你还没做好心理准备啊, 于是你无视了这些话.

寻找帮助

很快, 你已经看到"最后一行"了. 难道man的世界就这么狭小? 你仔细一看, "最后一行"里面含有一些信息:

> Manual page man(1) line 1 (press h for help or q to quit)

原来可以通过按 `q` 来离开这个世界啊, 不过你现在并不想这么做, 因为你想多了解 `man` , 以后可能会经常需要 `man` 的帮助. 为了更了解ta, 你按了 `h` .

这时你又被带到了新的世界, 世界的起点是"SUMMARY OF LESS COMMANDS", 你马上知道, 这个世界要告诉你如何使用 `man` , 你十分激动. 于是你往下看, 这句话说"带有'*'标记的命令可以在前面跟一个数, 这时命令的行为在括号里给出". 这是什么意思? 你没看懂, 还是找个带'*'的命令试试吧. 你继续往下看, 看到了两个功能和相应的命令:

- 第一个是展示帮助, 原来除了 `h` 之外, `H` 也可以看到帮助, 而且这里把帮助的命令放在第一个, 也许 `man` 想暗示你, 找到帮助是十分重要的.
- 第二个命令是退出. "哈哈, 知道怎么退出之后, 就不用通过重启来退出一个命令行程序啦", 你心想. 但你现在还是不想退出, 还是再看看其它的吧.

继续往下看, 你看到了用于移动的命令. 果然, 你还是可以在这个世界里面移动的. 第一个用于移动的功能是往下移动一行, 你看到有5种方法可以实现:

```
e  ^E  j  ^N  CR
```

`e` 和 `j` 你看懂了, 就是按 `e` 或者 `j` . 但 `^E` 是什么意思呢? 你尝试找到 `^` 的含义, 但是你没找到, 还是让我告诉你吧. 在上下文和按键有关的时候, `^` 是Linux中的一个传统记号, 它表示 `ctrl+` . 还记得Windows下 `ctrl+c` 代表复制的例子吗? 这里的 `^E` 表示 `ctrl+E` . `CR` 代表回车键, 其实 `CR` 是控制字符(ASCII码小于32的字符)的一个, 这里有一段关于控制字符的问答.

你决定使用 `j` , 因为它像一个向下的箭头, 而且它是右手食指所按下的键. 其实这点和 `vim` 的使用是类似的, 如果你不能理解为什么 `vim` 中使用 `h` , `j` , `k` , `l` 作为方向键, 这里有一个初学者的提问. 事实上, 这是一种touch typing.

你按下了 `j` , 发现画面上的信息向下滚动了一行. 你看到了 `*` , 想起了 `*` 标记的命令可以在前面跟一个数. 于是你试着输入 `10j` , 发现画面向下滚动了10行, 你第一次感觉到在这个"丑陋"的世界中也有比GUI方便的地方. 你继续阅读帮助, 并且尝试每一个命令. 于是你掌握了如何通过移动来探索 `man` 所在的世界.

继续往下翻, 你看到了用于搜索的命令. 你十分感动, 因为使用关键字可以快速定位到你关心的内容. 帮助的内容告诉你, 通过按 `/` 激活前向搜索模式, 然后输入关键字(可以使用正则表达式), 按下回车就可以看到匹配的内容了. 帮助中还列出了后向搜索, 跳到下一匹配处等功能. 于是你掌握了如何使用搜索.

探索man

你一边阅读帮助, 一边尝试新的命令, 就这样探索着这个陌生的世界. 你虽然记不住这么多命令, 但你知道你可以随时来查看帮助. 掌握了一些基本的命令之后, 你按 `q` 离开了帮助, 回到了 `man` 的世界. 现在你可以自由探索 `man` 的世界了. 你向下翻, 跳过了看不懂的 `SYNOPSIS` 小节, 在 `DESCRIPTION` 小节看到了人话, 于是你阅读这些人话. 在这里, 你看到整个manual分成9大类, 每个manual page都属于其中的某一类; 你看到了一个manual page主要包含以下的小节:

- NAME - 命令名
- SYNOPSIS - 使用方法大纲
- CONFIGURATION - 配置
- DESCRIPTION - 功能说明
- OPTIONS - 可选参数说明
- EXIT STATUS - 退出状态, 这是一个返回给父进程的值
- RETURN VALUE - 返回值
- ERRORS - 可能出现的错误类型
- ENVIRONMENT - 环境变量
- FILES - 相关配置文件
- VERSIONS - 版本
- CONFORMING TO - 符合的规范
- NOTES - 使用注意事项
- BUGS - 已经发现的bug
- EXAMPLE - 一些例子
- AUTHORS - 作者
- SEE ALSO - 功能或操作对象相近的其它命令

你还看到了对 `SYNOPSIS` 小节中记号的解释, 现在你可以回过头来看 `SYNOPSIS` 的内容了. 但为了弄明白每个参数的含义, 你需要查看 `OPTIONS` 小节中的内容.

你想起了搜索的功能, 为了弄清楚参数 `-k` 的含义, 你输入 `/-k` , 按下回车, 并通过 `n` 跳过了那些 `OPTIONS` 小节之外的 `-k` , 最后大约在第254行找到了 `-k` 的解释: 通过关键字来搜索相关功能的manual page. 在 `EXAMPLES` 小节中有一个使用 `-k` 的例子:

```
man -k printf
```

你阅读这个例子的解释: 搜索和 `printf` 相关的manual page. 你还是不太明白这是什么意思, 于是你退出 `man` , 在命令行中输入

```
man -k printf
```

并运行, 发现输出了很多和 `printf` 相关的命令或库函数, 括号里面的数字代表相应的条目属于manual的哪一个大类. 例如 `printf (1)` 是一个shell命令, 而 `printf (3)` 是一个库函数. 要访问库函数 `printf` 的manual page, 你需要在命令行中输入

```
man 3 printf
```

当你想做一件事的而不知道用什么命令的时候, `man` 的 `-k` 参数可以用来列出候选的命令, 然后再通过查看这些命令的manual page来学习怎么使用它们.

接下来, 你又开始学习 `man` 的其它功能...

开始旅程

到这里, 你应该掌握 `man` 的用法了. 你应该经常来拜访ta, 因为在很多时候, ta总能给你提供可靠的帮助.

在这个励志的故事中, 你学会了:

- 阅读程序输出的提示和错误信息
- 通过搜索来定位你关心的内容
- 动手实践是认识新事物的最好方法
- 独立寻找帮助, 而不是一有问题就问班上的大神

于是, 你就这样带着 `man` 踏上了Linux之旅...

git快速入门

光玉

想象一下你正在玩Flappy Bird, 你今晚的目标是拿到100分, 不然就不睡觉. 经过千辛万苦, 你拿到了99分, 就要看到成功的曙光的时候, 你竟然失手了! 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我? 为什么不让我存档?"

想象一下你正在写代码, 你今晚的目标是实现某一个新功能, 不然就不睡觉. 经过千辛万苦, 你终于把代码写好了, 保存并编译运行, 你看到调试信息一行一行地在终端上输出. 就要看到成功的曙光的时候, 竟然发生了段错误! 你仔细思考, 发现你之前的构思有着致命的错误, 但之前正确运行的代码已经永远离你而去了. 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我?" 你绝望地倒在屏幕前... 这时, 你发现身边渐渐出现无数的光玉, 把你包围起来, 耀眼的光芒令你无法睁开眼睛... 等到你回过神来, 你发现屏幕上正是那份之前正确运行的代码! 但在你的记忆中, 你确实经历过那悲痛欲绝的时刻... 这一切真是不可思议啊...

人生如戏, 戏如人生

人生就像不能重玩的Flappy Bird, 但软件工程领域却并非如此, 而那不可思议的光玉就是"版本控制系统". 版本控制系统给你的开发流程提供了比朋也收集的更强大的光玉, 能够让你在过去和未来中随意穿梭, 避免上文中的悲剧降临你的身上.

没听说过版本控制系统就完成实验, 艰辛地排除万难, 就像游戏通关之后才知道原来游戏可以存档一样, 其实玩游戏的时候进行存档并不是什么丢人的事情.

在实验中, 我们使用 `git` 进行版本控制. 下面简单介绍如何使用 `git` .

游戏设置

首先你得安装 `git` :

```
apt-get install git
```

安装好之后, 你需要先进行一些配置工作. 在终端里输入以下命令

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

经过这些配置, 你就可以开始使用 `git` 了.

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

```
git init
```

进行初始化.

查看存档信息

使用

```
git log
```

查看目前为止所有的存档.

使用

```
git status
```

可以得知, 与当前存档相比, 哪些文件发生了变化.

存档

你可以像以前一样编写代码. 等到你的开发取得了一些阶段性成果, 你应该马上进行"存档".

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

```
git add file.c
```

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

```
git add -A
```

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

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

```
git commit
```

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

读档

如果你遇到了上文提到的让你悲痛欲绝的情况, 现在你可以使用光玉来救你一命了. 首先使用 `git log` 来查看已有的存档, 并决定你需要回到哪个过去. 每一份存档都有一个hash code, 例如 `b87c512d10348fd8f1e32ddea8ec95f87215aaa5` , 你需要通过hash code来告诉 `git` 你希望读哪一个档. 使用以下命令进行读档:

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

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

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

第三视点

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

```
git branch
```

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

读档的时候使用以下命令

```
git checkout b87c
```

而不是 `git reset` . 这时你将处于一个虚构的分支中, 你可以

- 查看 `b87c` 存档的内容
- 使用以下命令切换到其它分支

```
git checkout 分支名
```

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

```
git checkout -B 分支名
```

  把修改结果保存到一个新的分支中, 如果分支已存在, 其内容将会被覆盖

不同的分支之间不会相互干扰, 这也给项目的分布式开发带来了便利. 有了分支功能, 你就可以像第三视点那样在一个世界的不同时间(一个分支的多个存档), 或者是多个平行世界(多个分支)之间来回穿梭了.

更多功能

以上介绍的是 `git` 的一些基本功能, `git` 还提供很多强大的功能, 例如使用 `git diff` 比较同一个文件在不同版本中的区别, 使用 `git bisect` 进行二分搜索来寻找一个bug在哪次提交中被引入...

其它功能的使用请参考 `git help` , `man git` , 或者在网上搜索相关资料.

x86指令系统简介

i386手册有一章专门列出了所有指令的细节, 附录中的opcode map也很有用. 我们需要实现的n86架构是x86的子集, 在这里, 我们先对x86指令系统作一些简单的梳理. 当你对x86指令系统有任何疑惑时, 请查阅i386手册, 关于指令系统的一切细节都在里面.

i386手册勘误

由于PDF版本的i386手册的印刷错误较多, 一定程度上影响理解, 我们在github上开放了一个repo, 用于提供修复印刷错误的版本. 同时我们也为修复错误后的版本提供在线的HTML版本.

如果你在做实验的过程中也发现了新的错误, 欢迎帮助我们修复这些错误.

指令格式

x86指令的一般格式如下:

```
+-----------+-----------+-----------+--------+------+------+------+------------+------
-----+
|instruction| address-  |  operand- |segment |opcode|ModR/M| SIB  |displacement| immed
iate |
|  prefix   |size prefix|size prefix|override|      |      |      |            |
     |
|-----------+-----------+-----------+--------+------+------+------+------------+------
-----|
|   0 OR 1  |  0 OR 1   |   0 OR 1  | 0 OR 1 |1 OR 2|0 OR 1|0 OR 1| 0,1,2 OR 4 |0,1,2
OR 4 |
| - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - -|
|                                   number of bytes
     |
+-----------------------------------------------------------------------------------
-----+
```

除了opcode(操作码)必定出现之外, 其余组成部分可能不出现, 而对于某些组成部分, 其长度并不是固定的. 但给定一条具体指令的二进制形式, 其组成部分的划分是有办法确定的, 不会产生歧义(即把一串比特串看成指令的时候, 不会出现两种不同的解释). 例如对于以下指令:

```
100017:    66 c7 84 99 00 e0 ff ff 01 00      movw   $0x1,-0x2000(%ecx,%ebx,4)
```

其组成部分的划分如下:

```
+-----------+-----------+-----------+--------+------+------+------+------------+------
-----+
|instruction| address-  |  operand- |segment |opcode|ModR/M| SIB  |displacement| immed
iate |
|  prefix   |size prefix|size prefix|override|      |      |      |            |
     |
|-----------+-----------+-----------+--------+------+------+------+------------+------
-----|
|                           66                c7     84     99    00 e0 ff ff    01
00   |
+-----------------------------------------------------------------------------------
-----+
```

凭什么 `0x84` 要被解释成 `ModR/M` 字节呢? 这是由 `opcode` 决定的, `opcode` 决定了这是什么指令的什么形式, 同时也决定了 `opcode` 之后的比特串如何解释. 如果你要问是谁来决定 `opcode` , 那你就得去问Intel了.

在n86中, `address-size prefix` 和 `segment override prefix` 都不会用到, 因此NEMU也不需要实现这两者的功能.

编码的艺术

对于以下5个集合:

1. 所有 `instruction prefix`
2. 所有 `address-size prefix`
3. 所有 `operand-size prefix`
4. 所有 `segment override prefix`
5. 所有 `opcode` 的第一个字节

它们是两两不相交的. 这是必须的吗? 这背后反映了怎样的隐情?

另外我们在这里先给出 `ModR/M` 字节和 `SIB` 字节的格式, 它们是用来确定指令的操作数的, 详细的功能会在将来进行描述:

```
ModR/M byte
7    6    5    4    3    2    1    0
+--------+-------------+-------------+
|  mod   | reg/opcode  |     r/m     |
+--------+-------------+-------------+


SIB (scale index base) byte
7    6    5    4    3    2    1    0
+--------+-------------+-------------+
|   ss   |    index    |    base     |
+--------+-------------+-------------+
```

事实上, 一个字节最多只能区分256种不同的指令形式. 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们. x86中有主要有两种方法来解决这个问题:

- 一种方法是使用转义码(escape code). x86中有一个2字节转义码 `0x0f` , 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况). 后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时,表示需要再读入一个字节才能决定具体的指令形式.
- 另一种方法是使用 `ModR/M` 字节中的扩展opcode域来对 `opcode` 的长度进行扩充. 有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式. x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展opcode域来区分.

指令集细节

要实现一条指令, 首先你需要知道这条指令的格式和功能, 格式决定如何解释, 功能决定如何执行. 而这些信息都在instruction set page中, 因此你务必知道如何阅读它们. 我们以 `mov` 指令的opcode表为例来说明如何阅读:

```
     Opcode       Instruction       Clocks        Description

< 1> 88 /r        MOV r/m8,r8       2/2           Move byte register to r/m byte
< 2> 89 /r        MOV r/m16,r16     2/2           Move word register to r/m word
< 3> 89 /r        MOV r/m32,r32     2/2           Move dword register to r/m dword
< 4> 8A /r        MOV r8,r/m8       2/4           Move r/m byte to byte register
< 5> 8B /r        MOV r16,r/m16     2/4           Move r/m word to word register
< 6> 8B /r        MOV r32,r/m32     2/4           Move r/m dword to dword register
< 7> 8C /r        MOV r/m16,Sreg    2/2           Move segment register to r/m word
< 8> 8D /r        MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
< 9> A0           MOV AL,moffs8     4             Move byte at (seg:offset) to AL
<10> A1           MOV AX,moffs16    4             Move word at (seg:offset) to AX
<11> A1           MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
<12> A2           MOV moffs8,AL     2             Move AL to (seg:offset)
<13> A3           MOV moffs16,AX    2             Move AX to (seg:offset)
<14> A3           MOV moffs32,EAX   2             Move EAX to (seg:offset)
<15> B0 + rb ib   MOV r8,imm8       2             Move immediate byte to register
<16> B8 + rw iw   MOV r16,imm16     2             Move immediate word to register
<17> B8 + rd id   MOV r32,imm32     2             Move immediate dword to register
<18> C6 /0 ib (*) MOV r/m8,imm8     2/2           Move immediate byte to r/m byte
<19> C7 /0 iw (*) MOV r/m16,imm16   2/2           Move immediate word to r/m word
<20> C7 /0 id (*) MOV r/m32,imm32   2/2           Move immediate dword to r/m dword

------------------------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative
to the segment base. The 8, 16, and 32 refer to the size of the data. The
address-size attribute of the instruction determines the size of the
offset, either 16 or 32 bits.
------------------------------------------------------------------------------

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.
```

上表中的每一行给出了`mov`指令的不同形式, 每一列分别表示这种形式的opcode, 汇编语言格式, 执行所需周期, 以及功能描述. 由于NEMU关注的是功能的模拟, 因此`Clocks`一列不必关心. 另外需要注意的是, i386手册中的汇编语言格式都是Intel格式, 而objdump的默认格式是AT&T格式, 两者的源操作数和目的操作数位置不一样, 千万不要把它们混淆了! 否则你将会陷入难以理解的bug中.

首先我们来看`mov`指令的第一种形式:

```
     Opcode       Instruction       Clocks        Description
< 1> 88 /r        MOV r/m8,r8       2/2           Move byte register to r/m byte
```

- 从功能描述可以看出, 它的作用是"将一个8位寄存器中的数据传送到8位的寄存器或者内存中", 其中`r/m`表示"寄存器或内存".
- Opcode一列中的编码都是用十六进制表示, `88`表示这条指令的opcode的首字节

是 `0x88` , `/r` 表示后面跟一个 `ModR/M` 字节, 并且 `ModR/M` 字节中的 `reg/opcode` 域解释成通用寄存器的编码, 用来表示其中一个操作数.

- 通用寄存器的编码如下:

二进制编码	000	001	010	011	100	101	110	111
8位寄存器	AL	CL	DL	BL	AH	CH	DH	BH
16位寄存器	AX	CX	DX	BX	SP	BP	SI	DI
32位寄存器	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

- Instruction一列中, `r/m8` 表示操作数是8位的寄存器或内存, `r8` 表示操作数是8位寄存器, 按照Intel格式的汇编语法来解释, 表示将8位寄存器(`r8`)中的数据传送到8位寄存器或内存(`r/m8`)中, 这和功能描述是一致的. 至于 `r/m` 表示的究竟是寄存器还是内存, 这是由 `ModR/M` 字节的 `mod` 域决定的: 当 `mod` 域取值为 `3` 的时候, `r/m` 表示的是寄存器; 否则 `r/m` 表示的是内存. 表示内存的时候又有多种寻址方式, 具体信息参考i386手册中的表格17-3.

看明白了上面的第一种形式之后, 接下来的两种形式也就不难看懂了:

```
< 2> 89 /r        MOV r/m16,r16     2/2          Move word register to r/m word
< 3> 89 /r        MOV r/m32,r32     2/2          Move dword register to r/m dword
```

但你会发现, 这两种形式的 `Opcode` 都是一样的, 难道不会出现歧义吗? 不用着急, 还记得指令一般格式中的 `operand-size prefix` 吗? x86正是通过它来区分上面这两种形式的. `operand-size prefix` 的编码是 `0x66` , 作用是指示当前指令需要改变操作数的宽度. 在i386中, 通常来说, 如果这个前缀没有出现, 操作数宽度默认是32位; 当这个前缀出现的时候, 操作数宽度就要改变成16位 (也有相反的情况, 这个前缀的出现使得操作数宽度从16位变成32位, 但这种情况在i386中极少出现). 换句话说, 如果把一个开头为 `89 ...` 的比特串解释成指令, 它就应该被解释成 `MOV r/m32,r32` 的形式; 如果比特串的开头是 `66 89...` , 它就应该被解释成 `MOV r/m16,r16` .

操作数宽度前缀的由来

i386是从8086发展过来的. 8086是一个16位的时代, 很多指令的16位版本在当时就已经实现好了. 要踏进32位的新时代, 兼容就成了需要仔细考量的一个重要因素.

一种最直接的方法是让32位的指令使用新的操作码, 但这样1字节的操作码很快就会用光. 假设8086已经实现了200条16位版本的指令形式, 为了加入这些指令形式的32位版本, 这种做法需要使用另外200个新的操作码, 使得大部分指令形式的操作码需要使用两个字节来表示, 这样直接导致了32位的程序代码会变长. 现在你可能会觉得每条指令的长度增加一个字节也没什么大不了, 但在i386诞生的那个遥远的时代(你可以在i386手册的封面看到那个时代), 内存是一种十分珍贵的资源, 因此这种使用新操作码的方法并不是一种明智的选择.

Intel想到的解决办法就是引入操作数宽度前缀, 来达到操作码复用的效果. 当处理器工作在16位模式(实模式)下的时候, 默认执行16位版本的指令; 当处理器工作在32位模式(保护模式)下的时候, 默认执行32位版本的指令. 当某些需要的时候, 才通过操作数宽度前缀来指示操作数的宽度. 这种方法最大的好处就是不需要引入额外的操作码, 从而也不会明显地使得程序代码变长. 虽然在NEMU里面可以使用很简单的方法来模拟这个功能, 但在真实的芯片设计过程中, CPU的译码部件需要增加很多逻辑才能实现.

到现在为止, <4>-<6>三种形式你也明白了:

```
< 4> 8A /r        MOV r8,r/m8       2/4           Move r/m byte to byte register
< 5> 8B /r        MOV r16,r/m16     2/4           Move r/m word to word register
< 6> 8B /r        MOV r32,r/m32     2/4           Move r/m dword to dword register
```

<7>和<8>两种形式的mov指令涉及到段寄存器:

```
< 7> 8C /r        MOV r/m16,Sreg    2/2           Move segment register to r/m word
< 8> 8D /r        MOV Sreg,r/m16    2/5,pm=18/19  Move r/m word to segment register
```

n86去掉了段寄存器的实现, 我们可以忽略这两种形式的 `mov` 指令.

<9>-<14>这6种形式涉及到一种新的操作数记号 `moffs` :

```
< 9> A0           MOV AL,moffs8     4             Move byte at (seg:offset) to AL
<10> A1           MOV AX,moffs16    4             Move word at (seg:offset) to AX
<11> A1           MOV EAX,moffs32   4             Move dword at (seg:offset) to EAX
<12> A2           MOV moffs8,AL     2             Move AL to (seg:offset)
<13> A3           MOV moffs16,AX    2             Move AX to (seg:offset)
<14> A3           MOV moffs32,EAX   2             Move EAX to (seg:offset)
-------------------------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relative
to the segment base. The 8, 16, and 32 refer to the size of the data. The
address-size attribute of the instruction determines the size of the
offset, either 16 or 32 bits.
-------------------------------------------------------------------------------
```

NOTES中给出了 `moffs` 的含义, 它用来表示段内偏移量, 但n86没有"段"的概念, 目前可以理解成"相对于物理地址0处的偏移量". 这6种形式是 `mov` 指令的特殊形式, 它们可以不通过 `ModR/M` 字节, 让 `displacement` 直接跟在 `opcode` 后面, 同时让 `displacement` 来指示一个内存地址.

<15>-<17>三种形式涉及到两种新的操作数记号:

```
<15> B0 + rb ib   MOV r8,imm8       2            Move immediate byte to register
<16> B8 + rw iw   MOV r16,imm16     2            Move immediate word to register
<17> B8 + rd id   MOV r32,imm32     2            Move immediate dword to register
```

其中:

- `+rb`, `+rw`, `+rd` 分别表示8位, 16位, 32位通用寄存器的编码. 和 `ModR/M` 中的 `reg` 域不一样的是, 这三种记号表示直接将通用寄存器的编号按数值加到 `opcode` 中 (也可以看成通用寄存器的编码嵌在 `opcode` 的低三位), 因此识别指令的时候可以通过 `opcode` 的低三位确定一个寄存器操作数.
- `ib`, `iw`, `id` 分别表示8位, 16位, 32位立即数

最后3种形式涉及到一种新的操作码记号 `/digit`, 其中 `digit` 为 `0` ~ `7` 中的一个数字:

```
<18> C6 /0 ib (*) MOV r/m8,imm8     2/2          Move immediate byte to r/m byte
<19> C7 /0 iw (*) MOV r/m16,imm16   2/2          Move immediate word to r/m word
<20> C7 /0 id (*) MOV r/m32,imm32   2/2          Move immediate dword to r/m dword

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.
```

上述形式中的 `/0` 表示一个 `ModR/M` 字节, 并且 `ModR/M` 字节中的 `reg/opcode` 域解释成扩展opcode, 其值取 `0`. 对于含有 `/digit` 记号的指令形式, 需要通过指令本身的 `opcode` 和 `ModR/M` 中的扩展opcode共同决定指令的形式, 例如 `80 /0` 表示 `add` 指令的一种形式, 而 `80 /5` 则表示 `sub` 指令的一种形式, 只看 `opcode` 的首字节 `80` 不能区分它们.

注: 在i386手册中, 这3种形式的 `mov` 指令并没有 `/0` 的记号, 在这里加入 `/0` 纯粹是为了说明 `/digit` 记号的意思. 但同时这条指令在i386中也比较特殊, 它需要使用 `ModR/M` 字节来表示一个寄存器或内存的操作数, 但 `ModR/M` 字节中的 `reg/opcode` 域却没有用到 (一般情况下, `ModR/M` 字节中的 `reg/opcode` 域要么表示一个寄存器操作数, 要么作为扩展opcode), i386手册也没有对此进行特别的说明, 直觉上的解释就是"无论 `ModR/M` 字节中的 `reg/opcode` 域是什么值, 都可以被CPU识别成这种形式的 `mov` 指令". x86是商业CPU, 我们无法从电路级实现来考证这一解释, 但对编译器生成代码来说, 这条指令中的 `reg/opcode` 域总得有个确定的值, 因此编译器一般会把这个值设成 `0`. 在NEMU的框架代码中, 对这3种形式的 `mov` 指令的实现和i386手册中给出 `Opcode` 保持一致, 忽略 `ModR/M` 字节中的 `reg/opcode` 域, 没有判断其值是否为 `0`. 如果你不能理解这段话在说什么, 你可以忽略它, 因为这并不会影响实验的进行.

到此为止, 你已经学会了如何阅读大部分的指令集细节了. 需要说明的是, 这里举的 `mov` 指令的例子并没有完全覆盖i386手册中指令集细节的所有记号, 若有疑问, 请参考i386手册.

除了opcode表之外, `Operation`, `Description` 和 `Flags Affected` 这三个条目都要仔细阅读, 这样你才能完整地掌握一条指令的功能. `Exceptions` 条目涉及到执行这条指令可能产生的异常, 由于n86不打算加入异常处理的机制, 你可以不用关心这一条目.

mov指令执行例子剖析

在PA1中, 你已经阅读了monitor部分的框架代码, 了解了NEMU执行的粗略框架. 但现在, 你需要进一步弄明白, 一条指令是怎么在NEMU中执行的, 即我们需要进一步探究 `exec_wrapper()` 函数中的细节. 为了说明这个过程, 我们举了两个 `mov` 指令的例子, 它们是NEMU自带的客户程序 `mov` 中的两条指令:

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
......
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

简单mov指令的执行

对于大部分指令来说, 执行它们都可以抽象成取指-译码-执行的指令周期. 为了使描述更加清晰, 我们借助指令周期中的一些概念来说明指令执行的过程. 我们先来剖析第一条 `mov $0x1234, %eax` 指令的执行过程.

取指(instruction fetch, IF)

要执行一条指令, 首先要拿到这条指令. 指令究竟在哪里呢? 还记得冯诺依曼体系结构的核心思想吗? 那就是"存储程序, 程序控制". 你以前听说这两句话的时候可能没有什么概念, 现在是实践的时候了. 这两句话告诉你, 指令在存储器中, 由PC(program counter, 在x86中就是 `%eip`)指出当前指令的位置. 事实上, `%eip` 就是一个指针! 在计算机世界中, 指针的概念无处不在, 如果你觉得对指针的概念还不是很熟悉, 就要赶紧复习指针这门必修课啦. 取指令要做的事情自然就是将 `%eip` 指向的指令从内存读入到CPU中. 在NEMU中, 有一个函数 `instr_fetch()` (在 `nemu/include/cpu/exec.h` 中定义)专门负责取指令的工作.

译码(instruction decode, ID)

在取指阶段, CPU拿到的是指令的比特串. 如果想知道这串比特串究竟代表什么意思, 就要进行译码的工作了. 我们可以把译码的工作作进一步的细化: 首先要决定具体是哪一条指令的哪一种形式, 这主要是通过查看指令的 `opcode` 来决定的. 对于大多数指令来说, CPU只要看指令的第一个字节就可以知道具体指令的形式了. 在NEMU中, `exec_real()` 函数首先通过 `instr_fetch()` 取出指令的第一个字节, 将其解释成 `opcode` 并记录在全局译码信息 `decoding` 中. 然后通过 `set_width()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)记录默认的

操作数宽度. 若操作数宽度结果为 `0`, 表示光看操作码的首字节, 操作数宽度还不能确定, 可能是16位或者32位, 需要通过 `decoding.is_operand_size_16` 成员变量来决定. 这其实实现了"操作数宽度前缀"的相关功能, 关于 `is_operand_size_16` 成员的更多内容会在下文进行说明.

返回后, `exec_real()` 接下来会根据取到的 `opcode` 查看 `opcode_table`, 得到指令的译码helper函数和执行helper函数, 并将其作为参数调用 `idex()` 函数来继续模拟这条指令的执行. `idex()` 函数的原型为

```
void idex(vaddr_t *eip, opcode_entry *e);
```

它的作用是通过 `e->decode` 函数(若不为 `NULL`)对参数 `eip` 指向的指令进行译码, 然后通过 `e->execute` 函数执行这条指令.

以 `mov $0x1234, %eax` 指令为例, 首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb8`, 然后将这个字节作为opcode来索引 `opcode_table`, 发现这一指令的操作数宽度是 `4` 字节, 并通过 `set_width()` 函数记录. 接着按照同样的方式来索引 `opcode_table`, 确定取到的是一条 `mov` 指令, 它的形式是将立即数移入寄存器(move immediate to register).

事实上, 一个字节最多只能区分256种不同的指令形式. 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们. x86中有主要有两种方法来解决这个问题(在PA2中你都会遇到这两种情况):

- 一种方法是使用转义码(escape code), x86中有一个2字节转义码 `0x0f`, 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况). 后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时, 表示需要再读入一个字节才能决定具体的指令形式.
- 另一种方法是使用 `ModR/M` 字节中的扩展opcode域来对 `opcode` 的长度进行扩充. 有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式. x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展opcode域来区分.

决定了具体的指令形式之后, 译码工作还需要决定指令的操作数. 事实上, 在确定了指令的 `opcode` 之后, 指令形式就能确定下来了, CPU可以根据指令形式来确定具体的操作数. 对于 `mov $0x1234, %eax` 指令来说, 确定操作数其实就是确定寄存器 `%eax` 和立即数 `$0x1234`. 在x86中, 通用寄存器都有自己的编号, `I2r` 形式的指令把寄存器编号也放在指令的第一个字节里面, 我们可以通过位运算将寄存器编号抽取出来; 立即数存放在指令的第二个字节, 可以很容易得到它. 需要说明的是, 由于立即数是指令的一部分, 我们还是通过 `instr_fetch()` 函数来获得

它. 总的来说, 由于指令变长的特性, 指令长度和指令形式需要一边取指一边译码来确定, 而不像RISC指令集那样可以泾渭分明地处理取指和译码阶段, 因此你会在NEMU的实现中看到译码函数里面也会有 `instr_fetch()` 的操作.

执行(execute, EX)

译码阶段的工作完成之后, CPU就知道当前指令具体要做什么了, 执行阶段就是眞正完成指令的工作. 对于 `mov $0x1234, %eax` 指令来说, 执行阶段的工作就是把立即数 `$0x1234` 送到寄存器 `%eax` 中. 由于 `mov` 指令的功能可以统一成"把源操作数的值传送到目标操作数中", 而译码阶段已经把操作数都准备好了, 所以只需要针对 `mov` 指令编写一个模拟执行过程的函数即可. 这个函数就是 `exec_mov()` , 它是通过 `make_EHelper` 宏来定义的:

```
make_EHelper(mov) {
    write_operand((id_dest, &id_src->val);
    print_asm_template2(mov);
}
```

其中 `write_operand()` 函数会根据第一个参数中记录的类型的不同进行相应的写操作, 包括写寄存器和写内存. `print_asm_template2()` 是个宏, 用于输出带有两个操作数的指令的汇编形式.

更新 `%eip`

执行完一条指令之后, CPU就要执行下一条指令. 在这之前, CPU需要更新 `%eip` 的值, 让 `%eip` 指向下一条指令的位置. 为此, 我们需要确定刚刚执行完的指令的长度. 事实上, 在 `instr_fetch()` 中, 每次取指都会更新它的 `eip` 参数, 而这个参数就是在 `exec_wrapper()` 调用 `exec_real()` 时传入的 `decoding.seq_eip` . 因此当 `exec_wrapper()` 执行完一条指令调用 `update_eip()` 时, `decoding.seq_eip` 已经正确指向下一条指令了, 这时候直接更新 `%eip` 即可.

复杂mov指令的执行

对于第二个例子 `movw $0x1, -0x2000(%ecx,%ebx,4)` , 执行这条执行还是分取指, 译码, 执行三个阶段.

首先是取指. 这条mov指令比较特殊, 它的第一个字节是 `0x66` , 如果你查阅i386手册, 你会发现 `0x66` 是一个 `operand-size prefix` . 因为这个前缀的存在, 本例中的 `mov` 指令才能被CPU识别成 `movw` . NEMU使用 `decoding.is_operand_size_16` 成员变量来记录操作数宽度前缀是否出现, `0x66` 的helper函数 `operand_size()` 实现了这个功能. `operand_size()` 函数对 `decoding.is_operand_size_16` 成员变量做了标识之后, 越过前缀重新调用 `exec_real()` 函数, 此时取得了眞正的操作码 `0xc7` . 由于 `decoding.is_operand_size_16` 成员变量进行过标识, 在 `set_width()` 函数中将会确定操作数长度为 `2` 字节.

接下来是识别操作数. 根据操作码 `0xc7` 查看 `opcode_table` , 调用译码函数 `decode_mov_I2E()` , 这个译码函数又分别调用 `decode_op_I()` 和 `decode_op_rm()` 来取出操作数. 阅读代码, 你会发现 `decode_op_rm()` 最终会调用 `read_ModR_M()` 函数. 由于本例中的 `mov` 指令需要访问内存, 因此除了要识别出立即数之外, 还需要确定好要访问的内存地址. x86通过 `ModR/M` 字节来指示内存操作数, 支持各种灵活的寻址方式. 其中最一般的寻址格式是

> displacement(R[base_reg], R[index_reg], scale_factor)

相应内存地址的计算方式为

> addr = R[base_reg] + R[index_reg] * scale_factor + displacement

其它寻址格式都可以看作这种一般格式的特例, 例如

> displacement(R[base_reg])

可以认为是在一般格式中取 `R[index_reg] = 0, scale_factor = 1` 的情况. 这样, 确定内存地址就是要确定 `base_reg` , `index_reg` , `scale_factor` 和 `displacement` 这4个值, 而它们的信息已经全部编码在 `ModR/M` 字节里面了.

我们以本例中的 `movw $0x1, -0x2000(%ecx,%ebx,4)` 说明如何识别出内存地址:

```
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

根据 `mov_I2E` 的指令形式, `0xc7` 是 `opcode` , `0x84` 是 `ModR/M` 字节. 在i386手册中查阅表格17-3得知, `0x84` 的编码表示在 `ModR/M` 字节后面还跟着一个 `SIB` 字节, 然后跟着一个32位的 `displacement` . 于是读出 `SIB` 字节, 发现是 `0x99` . 在i386手册中查阅表格17-4得知, `0x99` 的编码表示 `base_reg = ECX, index_reg = EBX, scale_factor = 4` . 在 `SIB` 字节后面读出一个32位的 `displacement` , 发现是 `00 e0 ff ff` , 在小端存储方式下, 它被解释成 `-0x2000` . 于是内存地址的计算方式为

> addr = R[ECX] + R[EBX] * 4 - 0x2000

框架代码已经实现了 `load_addr()` 函数和 `read_ModR_M()` 函数 (在 `nemu/src/cpu/decode/modrm.c` 中定义), 它们的函数原型为

```
void load_addr(swaddr_t *eip, ModR_M *m, Operand *rm);
void read_ModR_M(swaddr_t *eip, Operand *rm, bool load_rm_val, Operand *reg, bool load
_reg_val);
```

它们将变量 `eip` 所指向的内存位置解释成 `ModR/M` 字节, 根据上述方法对 `ModR/M` 字节和 `SIB` 字节进行译码, 把译码结果存放到参数 `rm` 和 `reg` 指向的变量中. 虽然i386手册中的表格17-3和表格17-4内容比较多, 仔细看会发现, `ModR/M` 字节和 `SIB` 字节的编码都是有规律可

循的, 所以 `load_addr()` 函数可以很简单地识别出计算内存地址所需要的4个要素(当然也处理了一些特殊情况). 不过你现在可以不必关心其中的细节, 框架代码已经为你封装好这些细节, 并且提供了各种用于译码的接口函数.

本例中的执行阶段就是要把立即数写入到相应的内存位置. 译码阶段已经把操作数准备好了, 执行函数 `exec_mov()` 会完成数据移动的操作, 最终在 `update_eip()` 函数中更新 `%eip` .