≡ 导航
实验前阅读
实验方案
实验环境
如何获得帮助
搜索引擎, 百科和问答网站
官方手册
GNU/Linux入门教程
许可协议

南京大学 计算机科学与技术系 计算机系统基础 课程实验 2020

实验前阅读

📢 ics2021讲义已发布

见这里.

ℹ 最新消息(请每天至少关注一次)

- 学术诚信(什么事情能做, 什么不能)
- 常见问题(对PA的各种困惑)
- 提问的智慧(如何正确求助)
- 外部资源
  - PA习题课 (2020 秋季学期, 蒋炎岩老师): 课程资料, B站录播
  - 中国大学MOOC的ICS理论课 (袁春风老师): 上, 中, 下
  - OS课 (2021 春季学期, 蒋炎岩老师)

- 2021/05/03
  - 新增ISA riscv64. 如果你需要实现riscv64, 而又在2021/05/03 16:00前获取NEMU的代码, 请运行以下命令(记得先备份项目):

    ```
    git clone -b ics2020 https://github.com/NJU-ProjectN/nemu.git /tmp/nemu
    cd /tmp/nemu
    git format-patch b6d9e14a..828ff811
    cd $NEMU_HOME
    patch -p1 < /tmp/nemu/0001-add-riscv64.patch
    git add $NEMU_HOME
    git commit -m "add riscv64"
    rm -rf /tmp/nemu
    ```

  - 如果你在2021/05/03 17:00前获取abstract-machine的代码, 请运行以下命令:

    ```
    cd $AM_HOME/..
    rm -rf $AM_HOME
    bash init.sh abstract-machine
    ```

- 2020/12/08
  - PA4已发布
- 2020/10/23
  - PA3已发布
- 2020/10/10
  - 我们更新了NEMU中刷新屏幕的方式, 这样之后可以简化PA3中的某些设计. 如果你在2020/10/10 22:05:00之前获取框架代码, 请参考这里更新框架代码.
- 2020/09/30
  - PA2已发布
- 2020/09/05
  - PA1已发布
- 2020/08/31
  - PA0已发布, 请大家按照讲义内容安装系统

ℹ 离线阅读实验讲义

实验讲义页面通过github page发布, 但其网络可能不稳定. 你可以把这个仓库克隆到本地, 然后通过浏览器来离线阅读讲义.

但随着讲义内容的更新, 你将无法自动地阅读到最新版本的内容. 你需要在仓库路径中手动执行 `bash update.sh` 来将最新版本的内容同步到本地. 再次强调, 如果你选择了离线阅读方式, 将由你来负责获取最新的讲义内容.

💡 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/mips32/riscv32(64)架构相应的教学版子集, 指导学生实现一个经过简化但功能完备的x86/mips32/riscv32(64)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:

- 图灵机与简易调试器
- 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 "::-1!" 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:

- x86
  - Intel 80386 Programmer's Reference Manual (PDF)(HTML)
  - System V ABI for i386
- mips32
  - MIPS32 Architecture For Programmers (Volume I, Volume II, Volume III)
  - System V ABI for mips32
- riscv32(64)
  - The RISC-V Instruction Set Manual (Volume I, Volume II)
  - ABI for riscv
- ISA无关的手册
  - System V generic ABI
  - 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为我们准备了一些GNU/Linux入门教程, 如果你是第一次使用GNU/Linux, 请阅读以下材料

- 流传自远古时代的OS实验课程网站中的Linux入门教程.
- The Missing Semester of Your CS Education(墙裂推荐)

许可协议

本作品采用 `知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆` 许可协议进行许可. 要查看该许可协议, 可访问这里, 或者写信到 Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

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

提交截止日期见这里

提交说明: 见这里

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

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

选做编程题

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, 就是没有中文版的各种手册(例如386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, 你在专业上的技术如人, 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.

We are going to install the Debian 10 distribution directly over your physical machine. If you already have one copy of GNU/Linux distribution, and you want to use your copy as the development environment, just use it! But if you encounter some troubles because of platform disparity, please search the Internet for trouble-shooting.

我可以选择Ubuntu吗?

可以. 不过Ubuntu中的工具版本会跟Debian稍有差异, 尤其是编译器GCC. 如果你选择Ubuntu, 可能在PA2中需要多进行少量的工作, 不过这不会对实验内容产生大幅的影响.

Ubuntu版本选择

经同学尝试发现, `kvm-tools` 在Ubuntu 18.04上无法编译成功, 会对实验内容会造成影响. 如果大家要选择Ubuntu, 建议选择比18.04更高的版本.

It is also OK to use virtual machines, such as VMWare or VirtualBox. If you decide to do this and you do not have a copy of GNU/Linux, please install Debian 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. PA的某些特性会依赖于64位平台.

使用GNU/Linux真机的一些原因

- 真机的稳定性会比虚拟机相对高一些(比如crash consistency之类的).
- 真机的性能会比虚拟机相对高一些(但不影响实验分数), 如果你希望在实验后期获得比较流畅的游戏体验, 真机是一个不错的选择

建议在实验真机的安装

虽然学习计算机专业不是为了修电脑装系统, 但如果你连系统都没装过, 也确实不太好意思跟亲戚说你是学计算机的. 现在机会来了, 如果你以前真的从来没有安装过操作系统, 我们强烈建议你装一下真机, 来了解一下安装操作系统都需要经历些什么.

Mac用户的不幸

与ICS同时开设的数字电路实验课程需要安装 `Vivado/Quartus` 工具来进行实验, 然而非常不幸的是, 这两款工具皆不提供Mac版本. 为了使用它们, 你必须安装Windows或者GNU/Linux操作系统. 那么, 还不如一石二鸟, 现在装个GNU/Linux真机, 就可以解决上述所有问题了.

如果你不必修读南京大学的课程, 可以忽略这个问题.

Getting Debian

You will get the newest stable version of Debian here. Click the link label with `amd64` in the `full CD sets` category. Then scroll to the bottom of the page, click `debian-10.?.?-amd64-xfce-CD-1.iso` to download the image. The size of the image is about 650MB.

Preparation

首先,当然是备份重要数据!

如果你是第一次安装操作系统, 你可能会因为误操作而删除了磁盘上的重要文件. 备份数据无疑是一个非常重要的操作, 这样以后你就可以在安装过程中进行各种尝试, 随心所欲地折腾你的机器了.

当然, 就算你安装系统的操作看上去是正确的, 也有可能因为硬件兼容问题导致操作系统无法启动等后果. 如果确实发生了这种情况, 就到互联网上搜索解决方案吧. 因此, PA0还是要尽早开始体验.

If you are going to install Debian directly on the physical machine, we recommend you to install via USB Stick. Please do the following:

- Reserve at least one partitions (20GB ~ 50GB) on the disk for Debian to install. For how to do this, please search the Internet. For example, if you use Windows, you can search for `压缩卷 Windows` .
- Burn the image into USB Stick. For how to do this, please search the Internet, too.

If you are going to install Debian in the virtual machine, please also search the Internet for details.

Installing Debian

Here we give a quick guide to install Debian. If you have any question, please refer to the official installation guide.

1. Graphics Install
2. **Select a language**. Choose `English` .

   不要选择Chinese

   使用中文的系统环境会给一些命令行操作带来不便, 甚至会增加错误排查的难度. 即使在英文的系统环境中, 也可以通过安装中文输出法来键入中文.

3. **Select your location**. Choose `Other -> Asia -> China` .
4. **Configure locales**. Just leave the default setting (`United States - en_US.UTF-8`).
5. **Configure the keyboard**. Just leave the default setting (`American English`).
6. **Configure the network**. You can continue without network configuration.
   - Hostname: Just leave the default setting (`debian`).
   - Domain name: Just leave the default setting ().
7. **Set up users and passwords**.
   - Root password: The *root* account is very important. If you forget the root password, you can not fully control the operating system.
   - Full name for the new user: Anything will be fine.
   - Username for your account: Anything will be fine, too. But pay attention to the restriction.
   - Choose a password for the new user: This password is different from the root's one, because they belong to different accounts.
8. **Configure the clock**. Just wait.
9. **Partition disks**. This step will perform disk partitioning.
   - Partitioning method: If you are installing Debian directly on the physical machine, choose `Manual` instead to select the new partition you prepared in the previous steps. Otherwise the existing data will be lost! If you are going to use virtual machine, then choose `Guided - seperate /home partition` . For more details, click `help` or search the Internet. After finishing configuration, select `Finish partitioning and write changes to disk` and click `Continue` .
   - Write the changes to disks?: Select `Yes` .

     请安装真机的同学两三确认重要数据已经备份并检查分区配置

     一旦点击 `Yes` , 操作将不可逆转.
10. **Install the base system**. Just wait.
11. **Configure the package manager**
    - Use a network mirror?: Select `No` .
12. **Select and install software**. Just wait.
    - Configuring popularity-contest: Just select your favor.
    - Software selection: Just leave the default setting.
13. **Install the GRUB boot loader on a hard disk**
    - Install the GRUB boot loader to the master boot record?: Choose `Yes` .
    - Device for boot loader installation: Select `/dev/sda` .
14. **Finish the installation**. Click `Continue` .

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

First Exploration with GNU/Linux

Open `terminal`, 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 during the installation. The current working directory is `~` now. As you switching to another directory, the prompt will change as well. You are going to code 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. But the terminal is completely with CLI (Command Line Interface). 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 is an article discussing the comparision between GUI and CLI.

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 system, issue the following command:

```
poweroff
```

However, you will receive an error message:

```
-bash: poweroff: command not found
```

This error is due to the property of the `poweroff` command - it is a system administration command. Execute this command requires superuser privilege.

Why executing the "poweroff" command requires superuser privilege?

Can you provide a scene where bad thing will happen if the `poweroff` command does not require superuser privilege?

Therefore, to shut down the system, you should first switch to the root account:

```
su -
```

Enter the root password you set during the installation. You will see the prompt changes:

```
root@hostname:/home/username#
```

The last character is `#`, instead of `$` before you executing `su -`. `#` is the indicator of root account. Now execute `poweroff` command again, you will find that the command is executed successfully.

! 不要强制关闭虚拟机!!!

如果你使用虚拟机, 你务必通过 `poweroff` 命令关闭虚拟机. 如果你通过点击窗口右上角的 `X` 按钮强制关闭虚拟机, 可能会造成虚拟机中文件损坏的现象. 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!

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

导航

Checking network state
Setting APT source file
Updating APT package information
Installing tools for PAs
Installing Chinese input method

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 system can access the Internet.

Checking network state

Inside NJU, you should easily obtain an IPv6 address. To test this, try to ping a mirror host with IPv6 support:

```
ping mirrors.tuna.tsinghua.edu.cn -c 4
```

You should receive reply packets successfully:

```
PING mirrors.tuna.tsinghua.edu.cn(2402:f000:1:408:8100::1 (2402:f000:1:408:8100::1)) 56 da
64 bytes from 2402:f000:1:408:8100::1 (2402:f000:1:408:8100::1): icmp_seq=1 ttl=54 time=1.
64 bytes from 2402:f000:1:408:8100::1 (2402:f000:1:408:8100::1): icmp_seq=2 ttl=54 time=1.
64 bytes from 2402:f000:1:408:8100::1 (2402:f000:1:408:8100::1): icmp_seq=3 ttl=54 time=1.
64 bytes from 2402:f000:1:408:8100::1 (2402:f000:1:408:8100::1): icmp_seq=4 ttl=54 time=1.

--- mirrors.tuna.tsinghua.edu.cn ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 7ms
rtt min/avg/max/mdev = 1.071/1.237/1.563/0.203 ms
```

If you do not have an IPv6 address, you may access the Internet by IPv4 address. You can try to ping a host outside the university LAN:

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

If you can ping Baidu successfully, you should successfully ping the mirror host above, too.

Setting APT source file

Run the following commands to update the APT source file:

```
bash -c 'echo "deb http://mirrors.tuna.tsinghua.edu.cn/debian/ stable main" > /etc/apt/sou
```

使用适配版本的源

如果你使用Ubuntu系统, 请不要使用上述的Debian源, 你需要使用和你的Ubuntu版本匹配的源.

However, you will receive an error message:

```
bash: /etc/apt/sources.list: Permission denied
```

This is because the APT source file is owned by root, and you do not have the permission to modify it.

One solution is first switching to the root account. But to avoid switching, an alternative way is to use `sudo`. If you find an operation requires superuser permission, append `sudo` before that operation. First, you should add your account to the `sudo` group:

```
su -   # we still need to switch to the root account first
adduser username sudo  # change `username` to your user name
exit
```

To let the changes to take effect, you should log out or reboot. After that, try

```
whoami
sudo whoami
```

If it is the first time you run `sudo`, enter the password of your account. You should find that `sudo whoami` will output `root`.

Now you can use `sudo` to modify the APT source file. This time you should modify it successfully. To check this, you can output the file:

```
cat /etc/apt/sources.list
```

You should see it really outputs the line you just `echo`ed.

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. Run the command with superuser privilege. 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 utilities, g
apt-get install man                # on-line reference manual
apt-get install gdb                # GNU debugger
apt-get install git                # revision 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 libc6-dev-i386     # a library to use compile the project later
apt-get install qemu-system        # QEMU
```

The usage of these tools is explained later.

Installing Chinese input method

`搜狗输入法 for Linux` is a Chinese input method for GNU/Linux. Refer to this page to install it.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

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, or
- the remote server can not be accessed via network (e.g. only via a serial port).

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？

因为说实话书不是用来讲的，是用来一边看一边操作的；你也不能光看课听说没来掌握这些工具，而是要自己动动手试试，你在大学课堂上应该学习到的是那些一脉相承的知识，然后负责考过些知识背后的原理和思想，将来有需要时再这些原理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，你有能力自己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>qiyyp<C-a>j@@q1
```

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>qiyyp<C-a>j998@q1
```

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:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccc
dddddddddddddddddddddddddddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
fffffffffffffffffffffffffffffffffffffffffffffffff
ggggggggggggggggggggggggggggggggggggggggggggggggg
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
```

You want to modify it into:

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
dddddddddddddddddddddddddddddddddddddddddddddddd
fffffffffffffffffffffffffffffffffffffffffffffffff
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
```

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>24l4jdsp
```

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>24l99jdsp
```

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
@@ -17,2 +17,3 @@
 " VimS 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，我阅阅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dif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，你都还有什么理由去浪费你的努力，而是直为想看别人来帮你解决，占用别人宝贵的时间，你用是这个时代的灾难者。

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
- 使用英文效果会比中国城高；
- 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："增我阅才用的是百度，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.

🔹 为什么不重用百度？

相信大家都用过百度来搜索一些非技术问题，而且一般很容易找到答案。但随着问题技术含量的提高，百度的搜索结果会变得越来越不靠谱，这种使用百度搜索技术问题，你将很有可能会碰到以下情况之一：

- 搜不到相关结果，你感到绝望
- 找到看似相关的结果，但无法解决问题，你在感到挫败之余，也发现自己浪费了不少时间
- 你强行了解决问题的方案，但没有发现潜在分析，结果你不知道这个问题是后的隐患

你可能会觉得"可以解决问题的话，不重要了解问题背后的原理"，但对于一些问题(例如编程问题)，你了解这些原理背后的知识，将会有很大的帮助，将以你应该是了解这些原理，让自己增强更多。

如果你能以更高的概率提供可以解决问题的方案，并且带有原因分析，你应该没有理由使用百度来搜索技术问题，如果你仍然坚持使用百度，原因就只有一个：你不想主动去成长。

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，你是觉其实也就是改动一个字符而已，为什么不直接说清楚呢？

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格式是一种精述文件改动的常用方式，实际上，计算机的世界是通过精准表达来的思维成的"现实"，当你想要去接触去了解这些规矩的时候，你就会在不知不觉中即自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
@@ -21,2 +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.
-"filetype plugin indent on
+filetype plugin indent on
@@ -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 autowrite       " Automatically save before commands like :next and :make
-"set hidden          " Hide buffers when they are abandoned
-"set hlsearch        " Highlight matches.
+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 autowrite       " Automatically save before commands like :next and :make
+set hidden          " Hide buffers when they are abandoned
+set hlsearch        " Highlight matches.
 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 buftimeout=500  " 当buffer被改动的时候隔0.
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除 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 backspace=eol,start,indent " 允许退格键作用于缩进
set bs=2        " 设置命令行的行为 2
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%W]%=\ %y\ %{!(&fileformat&&(&fileformat!=(&ff?&ff:&ffs?&ffs:&ffd?&ffd:&ffm?&ffm:&ffk?&ffk:&ffh?&ffh:&ffb?&ffb:&ffv?&ffv:&ffr?&ffr:&ffo?&ffo:&ffu?&ffu:&ffx?&ffx
```

☰ 导航
Learning to use basic tools
Installing tmux
Why GNU/Linux and How to

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. Another book recommended by us is Harley Hahn's Guide to Linux and Linux.

🚩 1 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.

🚩 1 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复杂对象的所有信息，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书找解决方案，但是，你需要明确以下几点：

- 你搜到的博客可能是转载别人的，有可能有错
- 博主只是分享了他的经历，有些说法也不一定准确
- 限到了相关内容，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“我现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.

！ 1 加油！前途堪忧！

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的基本使用吧！

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.

🚩 1 又重没完没了地STFW？？

对.

PA是了让大家巩固ICS理论课的知识点外，还承担着一个重要的任务：把大家培养成一个熟练合格的CSer. 事实上，一个熟练合格的CSer需要具备独立搜索解决方案的能力：这是什么问题？这是什么问题的关键？这是什么问题的关键？这是什么问题的关键

目 录
Getting Source Code for PAs
Getting Source Code
Git usage
Compiling and Running NEMU
Compiling and Running
Local Control
Writing Report
Submission
RTFSC and Kopy

Getting 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 the system (`/`) are related to system. Mistakes in directories may NOT trash your PAs and Labs under these directories by `sudo`.

下载最新nemu和kopy

使用nemu和kopy的仓库，以获取最新版本下的代码修复。可能会有许多新提交的名义上是工作，但严格的，你的进度可能会在整个学期发生变化。强烈建议在这段时间内保持稳定。如果您在下学年参加此项目，您应该在下学期开始时重新开始。您应该在下学期开始时为所有更改工作。请不要对以前工作提出问题。

如果你的工作被破坏了，请不要抱怨。你可以阅读关于下面的 Why is it bad to rugan as-is? 之类的解释。此外，从这些项目的讲座和随堂测验中学习工作要求和技巧可以。如果您要参加nemu的第二部分，请参阅下面的说明。切勿 `rugan`。

Now get the source code for PA by the following command:

```
git clone -b 2020 https://github.com/NJU-ProjectN/nemu.git ics2020
```

A directory called `ics2020` will be created. This is the project directory for PAs. Details will be explained in PA1.

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 completion of the subprojects. Therefore, DO NOT move your project to another directory once finishing the installation. Otherwise, some functions will be broken. Particularly, if you use shell other than `bash` , please set these variables in the corresponding configuration file manually.

To list the environment variables take effect, run

```
source ~/.bashrc
```

Then try

```
echo $NEMU_HOME
echo $AM_HOME
cd $NEMU_HOME
cd $AM_HOME
```

to initialize some subprojects. The script will put some subprojects from github. We will explain them later.

Now, the script will also add some environment variables into the bash configuration file `~/.bashrc` . These variables are defined by absolute path to support the completion of the subprojects. Therefore, DO NOT move your project to another directory once finishing the installation. Otherwise, some functions will be broken. Particularly, if you use shell other than `bash` , please set these variables in the corresponding configuration file manually.

To list the environment variables take effect, run

```
source ~/.bashrc
```

Then try

```
echo $NEMU_HOME
echo $AM_HOME
cd $NEMU_HOME
cd $AM_HOME
```

to check whether these environment variables get the right paths. If both the `echo` commands report the right paths, and both the `cd` command change to the target paths without errors, we are done. If not, please double check the steps above and the shell you are using.

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 parent 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.ilp32` :

```
STUID = 2012XXXXX # your student ID
STUNAME = 如一 # 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.ilp32

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.ilp32 b/nemu/Makefile.ilp32
index c9b17b8..a7b1a42 100644
--- a/nemu/Makefile.ilp32
+++ b/nemu/Makefile.ilp32
@@ -5,7 +5,8 @@
-STUID = 2012XXXXX
-STUNAME = 如一
+STUID = 19122121X
+STUNAME = 如一

  # 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 -u
git commit
```

The `git commit` command will call the editor. Type `modified: ics.txt` in the first line, and keep the remaining contents unchanged. Save and exit the editor, and the 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.ilp32` , and you find the `STUID` and `STUNAME` are still unchanged! By issuing `git log` , you will find that the commit log you just created has disappeared!

Don't panic! This is because 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` , PAs will introduce tools to support it
- after finish the PA, merge the branch `pa` into `master` , and check out 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.

Linux下的工具

如果使用Ubuntu 18.04，你可能不能安装测试工个程序。如果是这样，你可以重新安装最后一个Ubuntu版本，重新尝试下载。

```
diff --git a/nemu/Makefile.ilp32 b/nemu/Makefile.ilp32
index c9b17b8..a7b1a42 100644
--- a/nemu/Makefile.ilp32
+++ b/nemu/Makefile.ilp32
@@ -5,7 +5,8 @@
  ifdef SHARE
  -lreadline
  endif
+STUID = 19122121X
+STUNAME = 如一
  ifeq ($(CC),clang)
  -Wno-invalid-noreturn
```

However, you will see an error message:

```
x86-nemu.interpreter: src/lib/misc/libc.a(libc.o): reg_test: Assertion `reg_off >= (sizeof(cl_t
```

This message tells you that the program has triggered an assertion fail at line 20 of the file `nemu/src/lib/misc/libc.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 20 of `nemu/src/lib/misc/libc.c` , you will discover the failure is in a test function. Just ignore it now, and you will fix it in PA1.

To debug NEMU with gdb, type

```
make gdb
```

Development Tracing

Once the compilation successes, the changer of source code will be traced by `git` . Type

```
git log
```

If you see something like

```
commit 4ef32d09c9dc3b6d3787f72d87b1b8b24e4da79
Author: Ernie Liu<2012XXX-如一@njuer.org>
Date:   Sun Jul 26 14:06:31 2020 +0800

    * compile
    ICS2020000
    - Static hostname: mybox
      - Icon name: computer-laptop
      - Chassis: laptop
      - Machine ID: 7c0f0f7b9b6b6ed8c7b1b9d7b8b98ac
      - Boot ID: a0c0d0f0e0d0c0b0a0908070605040302
    - Operating System: Debian GNU/Linux 10 (buster)
      - Kernel: Linux 5.4.0-4-amd-x64
      - Architecture: x86-64
    - 14:20:58 up 20 days, 23:36, 19 users,  load average: 1.05, 1.02, 1.07

this means the changer is traced successfully.
```

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

```
fatal: fseek to create "/home/user/ics2020/.git/index.lock": File exists.

If an other git process is currently running, this probably means a
git process crashed in the future\n\n\n
```

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

阶段性安排: 提交截止日期见这里

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

提交说明: 见这里

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

在开始愉快的PA之旅之前

❓ PA是一种全新的训练

我们从以下方面对不同的作业/实验/问题进行比较：

基本原理	做事方案	正确性风险	代表例子
阐述	明确	基本正确	高中物理实验
阐述	明确	可能出错	程序设计作业
阐述	需要思考	基本正确	数学证明/算法设计
阐述	需要思考	可能出错	PA, OSlab
需要探索	需要思考	可能出错	业界和科研的真实问题

做PA的终极目标是通过构建一个简单完整的计算机系统，来深入理解程序如何在计算机上运行，和那些“用进行实现汉诺塔”的程序设计作业不同，计算机系统的运行是一个复杂的整体。这意味着，通过程序设计作业的训练不完成不足以完成PA的，只有去尝试理解并重复计算机系统的每一处细节，才能一步步完成PA。

所以，不要再用程序设计作业的风格来面对PA这元可得不清楚，之所以这么做的理由，之后以谈的话题时会说明。是为了强迫大家去理清计算机系统的每一处细节，去推敲每一个细节之间的关系，也是为了让大家积累对系统足够的了解来面对未知的bug。

这对你来说也并是一种新形式有的训练方式，所以你也需要重出全新的态度来接受全新的挑战。

❓ 做PA的正确姿势 - 从今天开始，不要做傻了（这是一句真话，当你将来觉得这把对的时候，和非这是看看吧）

我们先列举一些错误做法：

- 遇到问题了，随便改改试试，结不出就过了
- 随便改改过了了，还是找大部分测试用例都能通过的结果
  - 也不是在时间精力允许的情况下把每一个测试用例的方式走得问
- 这个函数/文件/类/变量不懂，反正也不是很有用，算了就这样吧
- 中间有点看不明白的地方，也不那麻烦做单元测试
- 模块图不算分，不看也没关系
- 反正大部分看不明白的时候，看后一两天吧，在这还是过上

如果你不用了以上做法，你也许真的能做完这些或部分的实验内容，但这是以放弃训练机会为代价的，请看实验这里的推进，你会感到PA对你来说越来越吃力。

正确的做法是：

- 多思考为什么
  - 从问题开始看手理解系统也是个不错的方法
- 独立解决问题
  - 即使是第一个语句报错的bug，“顺带”增学到的东西也比你想象中多得多
  - 换句话说，如果你连报错都不会解决，你天去的机会也比你想象中多得多
- 尝试开始理解每一处细节
  - 将来第nn步时候，这些细节就是你手中强有力的工具
  - 换句话说，当你在debug的时候感到无从下手，一定是你不了解其中的细节
- 用正确的工具做事情
  - 这才是节省时间的科学方法，而不是偷懒
- 多读讲义，彩蛋很多
  - 讲义中特地设置了不少“不在时宜”的提示，有的彩蛋要多次阅读才能明白其中的真妙
  - 多看一遍总程序，也许能少调几天bug
- 积时充实，思维熟练
  - 这样你才有时间做别上面几点

事实上，这些做法就是PA的基本思想的最终目标之一。而这样的训练，在PA0那已经开始了：PA2之所以让大家白手起机，就是希望让大家在解决小问题的过程中收获经验，用来解决更大的问题。同时也给大家传授“取可以通过STFW/STEW/RTFM独立解决问题的最初级的东西，这种独立可以帮助大家取取对未来的挑战的东西。

你用来应付程序设计作业的心态，在PA这里是没不过去的，问题暴露的速度比你想象中快很多，所以，从今天开始，不要偷懒了。

NEMU是什么？

PA的目的是要实现NEMU，一款经过简化的全系统模拟器，但什么是模拟器呢？

你小时候应该玩过过日机，超级马赫，超类大战，魂斗罗……它们的画面是一定会记忆犹新？（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，模拟器的横空出世地登了你心中主封已久的童年回忆，红白机模拟器可以为你模拟出红白机的所有功能，有了它，你就珍惜了一个真正的红白机，可以玩任意喜欢的红白机游戏，我们称呼了一个红白机模拟器项目FCEUX，你在PA0中已经克隆了它，你可以在取分这个红白机难以寻觅的时代，再次回味自己的快乐时光，这实在是太神奇了！

❓ 不是说一下吗？

我们在这里（需要在校园网内部访问提供了一些游戏的ROM用于测试，阅读说明根据 `fceux-am/README.md` 中的内容进行操作，即可在弹出的新窗口中进行超级马赫。

你也可以将自己STFW获得的其它ROM文件放进来，这样就可以进行其它游戏了。

❓ 觉得模译有点慢？

`make` 程序默认使用单线程来顺序地编译所有文件，但是在现代的PC都是多核多线程了，不把这些计算能力用起来也是白浪费。为了加快编译的过程，我们可以给 `make` 创建多个线程来并行地编译文件。

具体地，首先你需要通过 `lscpu` 命令来查询你的系统中有多少个CPU，然后在运行 `make` 的时候添加一个 `-j7` 的参数，其中 `7` 为你查询到的CPU数量，例如 `-j4` 表示创建4个线程来并行编译，如果系统中CPU的数量大于等于4，那么编译系统就可以将这4个线程调度到4个CPU上同时执行，这到加速的效果；但如果系统中只有2个CPU，那操作系统会随机地将这4个线程调度到2个CPU上同时执行，这时候的加速效果就就 `-j2` 差不多了。

❓ 检查函数和声音

在运行游戏的过程中，你需要顺便检查一下是否可以看到画面并听到声音，超级马赫在初始界面中不会播放声音，但会在正式进入关卡时播放声音，如果没有声音，说明你的部分功能有问题，但不会影响成绩；但如果画面不能正常显示，可能会影响PA0余部分的实验内容，请自行搜索解决方案。

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，但你然然不到你想要的答案，但可以肯定的是，红白机模拟器并且是一个普通的程序，因为你还是需要像运行Hello World程序那样通过编译，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，以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！它模拟了一个硬件的世界，你可以在这个硬件世界中执行程序，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

图中(a)展示了“在GNU/Linux中运行Hello World”的情况，GNU/Linux操作系统直接运行在真实的计算机硬件上，Hello World程序通过运行了操作，同时向真实的计算机硬件发出指令，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真实的计算机硬件上，而是运行在操作系统在这里是GNU/Linux上。

图中(b)展示了“在GNU/Linux中通过红白机模拟器超级马赫”的情况，在GNU/Linux的基础上，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真实的红白机硬件，让超级马赫可以在其上运行，事实上，好超级马赫来说，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“模拟”的精髓法。

图中(c)展示了“在GNU/Linux中通过NEMU执行Hello World”的情况，在GNU/Linux基础上，运行在其上的NEMU和之上提到的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真实的硬件，让程序可以在其上运行，事实上，上图只是给出了对NEMU的一个基本模拟，更多细节会在后续PA中逐渐补充。

❓ 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机都会给顾客地输出，让我们相信顾客是一台真实的机器。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真实的计算机硬件上，而是运行在操作系统在这里是GNU/Linux上。

图中(b)展示了“在GNU/Linux中通过红白机模拟器超级马赫”的情况，在GNU/Linux的基础上，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真实的红白机硬件，让超级马赫可以在其上运行，事实上，好超级马赫来说，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“模拟”的精髓法。

图中(c)展示了“在GNU/Linux中通过NEMU执行Hello World”的情况，在GNU/Linux基础上，运行在其上的NEMU和之上提到的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真实的硬件，让程序可以在其上运行，事实上，上图只是给出了对NEMU的一个基本模拟，更多细节会在后续PA中逐渐补充。

❓ 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机都会给顾客地输出，让我们相信顾客是一台真实的机器。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真实的计算机硬件上，而是运行在操作系统在这里是GNU/Linux上。

图中(b)展示了“在GNU/Linux中通过红白机模拟器超级马赫”的情况，在GNU/Linux基础上，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真实的红白机硬件，让超级马赫可以在其上运行，事实上，好超级马赫来说，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“模拟”的精髓法。

图中(c)展示了“在GNU/Linux中通过NEMU执行Hello World”的情况，在GNU/Linux基础上，运行在其上的NEMU和之上提到的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真实的硬件，让程序可以在其上运行，事实上，上图只是给出了对NEMU的一个基本模拟，更多细节会在后续PA中逐渐补充。

NEMU的成力在分此也感到吃惊？它不仅仅能运行Hello World这种简单的程序，它甚至能在NEMU中运行经典游戏超级马赫！你会问自己，NEMU是怎么做到的？这一切，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，甚至你会发现自己成为一个属于自己的计算机不再是遥不可及！

选择你的角色

✓ 新特性 - 多芯核

PA有一个多主核的特性，具体地，你需要从x86/mips32/riscv32/64这三种指令集架构(ISA)中选择一种，来实现“创造属于自己的计算机”这一梦想。

但无论选择哪种ISA，你最终都会体会到“软硬件共同协助来支持程序执行”的机理：所谓的tradeoff，只不过是决定其一种情况应该使用硬件，还是交给软件来做。但二种ISA种是各有特性，它们好不同是节的发展难度如下表所示(0星 - 最简单，3星 - 最难)：

	x86	mips32	riscv32(64)
PA1 - 简易调试器	与ISA选择关系不大		
PA2 - 冯诺依曼计算机系统	*	***	*****
PA3 - 批处理系统	*****	***	*****
PA4 - 分时多任务	*****	*	****

✓ 什么是ISA？

大部分读者之都会有类似“ISA是软件和硬件之间的接口”这种说辞，但对于还不了解软件和硬件之间如何协同工作的你来说，“接口”这个词还是太抽象了。

为了理解ISA，我们可以用现实生活中的例子来比喻：螺钉和螺母是生活中两种常见的物品，它们一般需要配对使用，经定一个螺钉，那就要找到一个符合相同尺寸规范的螺母才能配合使用，反之亦然。

在计算机世界中也是类似的：不同架构的计算机或者说硬件讲比不同尺寸的螺钉，不同架构的程序/或者说软件就相当于是不同尺寸的螺母，如果一个程序要在特定架构的计算机上运行，那么这个程序和计算机就必须要符合同一套规范才行。

因此，ISA的本质就是类似这样的规范，所以ISA的存在形式既不是硬件电路，也不是软件代码，而是一本规范手册。

和螺钉螺母的生产过程类似，计算机硬件是按照ISA规范手册来设计的，而程序也是按照ISA规范手册编写/或生成出来的，至于ISA规范是要都有哪些内容，我们应该知何构造一个符合规范的计算机，程序应该如何遵守这些规范来在计算机上运行，回答这些问题正是我PA的一个目标。

✓ 我该如何选择？

如果你打算选熟悉的，那就选x86，毕竟CS理论课主要围绕x86开展，但你多半会被x86指令的复杂性折磨半死，而且x86的最终也触其实并不高，不能流畅地展示游戏的运行。

如果你打算选简单的，那就选riscv32，你将会体会到什么是“优雅的ISA设计”，由于riscv32的简单，你可以比较轻松地获得还乎x86有强的性能，有看不错的展示效果。

如果你随手不到是设计一款riscv64的硬件电路，那就选riscv64，你将会体会到DIFTest是如何帮你大幅并硬件开发效率，你知晓的这的发现一样。

如果你打算挑战极限，那就选mips32：相比于以上两者，选择mips32需要了解更多细节才能正确构建出完整的计算机系统，因此mips32只供喜欢挑战，或喜欢刨根问底的同学选择。

不过无论你选哪种ISA，有一点是共通的，那就是RTFM，因为ISA的本质是规范手册，另外，NEMU程序本身也是x86的/准确来说是x64，不会随着你选择的ISA而变化，变化的只是在NEMU中模拟的计算机。

为了方便叙述，讲义将用 `ISA` 来表示你选择的ISA，例如对于 `nemu/src/isa/ISA/req.c` ，若你选择的是x86，它将表示 `nemu/src/isa/x86/req.c` ；若你选择的是riscv32，它将表示 `nemu/src/isa/riscv32/req.c` 。**除非讲义明确说明，否则 `ISA` 总是表示你选择的ISA，而不是 `ISA` 这四个字符**。

最后，你应需要填取新手礼包 - ISA相关的生存手册 (部分ISA天关的手册请到讲义首页获取)：

ISA	新手礼包
x86	Intel 80386 Programmer's Reference Manual (简称i386手册) (PDF)(HTML) System V ABI for i386
mips32	MIPS32 Architecture For Programmers (Volume I, Volume II, Volume III) System V ABI for mips32
riscv32/64	The RISC-V Instruction Set Manual (Volume I, Volume II) ABI for riscv

✓ riscv32/64riscv64

事实上，riscv32和riscv64之间的区别非常小，以至于它们的ISA手册都是相同的，因此讲义中统大部分时对riscv32的描述，对riscv64也是适用的，在描述不适用riscv64的场合，我们会明确提醒你的补充说明，因此如果你选择了riscv64，当讲义中没有对riscv64进行额外说明的时候，你只需要参考riscv32的相应说明即可。

还等什么呢？

让我们来开始这段激动人心的旅程吧！

❓ 做一个素质合格的CSer

PA是了经大家熟悉“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在尝试取法他人之长的同时，也在尝试取法他人之短原则的重要性：这也是作为一个素质合格的CSer的必修课，因为这是现代社会PA作为一个编程大任务，我们相信在座各位。

PA是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，多主线的特性也让二周目不至于太过无聊，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

我们已经在你手中提供的实验报告中多次看到类似的情况：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做过哪事的细节，为了和欺骗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简智bug，等等。

我们相信，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。

导航
最简单的计算机
重新认识程序: 程序是个状态机

开天辟地的篇章

先聪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。死不说仙何奇侠传这个庞杂大物，为了执行那怕最简单的程序，最简单的计算机又应该长什么样呢？

最简单的计算机

为了执行程序，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，先聪创造了存储器，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了，不过现在还是让我们来重新介绍一下它吧。CPU是先聪最伟大的创造，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，先聪为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。

先聪发现，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，这是先聪也无法违背的材料特性规律。于是先聪为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。

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，不过目前我们还是暂其自然吧。

计算机可以没有寄存器吗？(建议二周后思考)

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ISA手册，你会记得确实有这么个概念，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，先聪还设计了“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，先聪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，先聪和CPU终于一个简单的约定：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，先聪为CPU创造了一个特殊的计数器，叫“程序计数器”(Program Counter, PC). 在x86中，它有一个特殊的名字，叫 `EIP` (Extended Instruction Pointer).

从此以后，计算机就只需要做一件事情：

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

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。

例如，下面的指令序列可以计算 `1+2+...+100` ，其中 `r1` 和 `r2` 是两个寄存器，还有一个隐含的程序计数器 `PC` ，它的初值是 `0` 。为了帮助大家理解，我们把指令的语义翻译成C代码放在右侧，其中每一行C代码前都添加了一个语句标号：

```
// PC: instruction     | // label: statement
0: mov  r1, 0          | pc0: r1 = 0;
1: mov  r2, 0          | pc1: r2 = 0;
2: addi r2, r2, 1      | pc2: r2 = r2 + 1;
3: add  r1, r1, r2     | pc3: r1 = r1 + r2;
4: blt  r2, 100, 2     | pc4: if (r2 < 100) goto pc2;   // branch if less than
5: jmp 5               | pc5: goto pc5;
```

计算机执行以上的指令序列，最后会在 `PC=5` 处的指令陷入死循环，此时计算已经结束，`1+2+...+100` 的结果会存放在寄存器 `r1` 中。

尝试理解计算机如何计算

在看到上述例子之前，你可能会觉得指令是一个既神秘又难以理解的概念，不过当你看到对应的C代码时，你就会发现指令做的事情竟然这么简单！而且看上去还有点蠢，你随手写一个for循环都要比这段C代码看上去更高级。

不过你也不妨站在计算机的角度来理解一下，计算机究竟是怎么通过这种既简单又笨拙的方式来计算 `1+2+...+100` 的，这种理解会使你建立“程序如何在计算机上运行”的最初原的认识。

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**宏伟的核心思想**，“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意，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

碳？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, 但先聪说, 你正在面对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, 这个冷冰冰的机器又是如何理解越结了人类智慧结晶的C代码的呢？先聪说, 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”这种天然易转的工作方式, 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.

计算机是个状态机

既然计算机是一个数组逻辑电路, 那么我们可以把计算机划分成两部分, 一部分由所有时序逻辑部件(存储器, 计数器, 寄存器)构成, 另一部分则是剩余的组合逻辑部件(如加法器等). 这样以后, 我们就可以从状态机模型的视角来理解计算机的工作过程了: 在每个时钟周期到来的时候, 计算机根据当前时序逻辑部件的状态, 在组合逻辑部件的作用下, 计算出并转移到下一时钟周期的新状态.

计算机的这个视角有什么用呢？好像除了让你明白计算机硬件不再那么神秘之外，也没什么特别的用处. 毕竟ICS课不要求大家用硬件描述语言来实现计算机硬件, 大家只要相信这件事能做成就可以了.

不过对于程序来说, 这个视角的作用会超乎你的想象.

重新认识程序: 程序是个状态机

如果把计算机看成一个状态机, 那么运行在计算机上面的程序又是什么呢？

我们知道程序是由指令构成的, 那么我们先看看一条指令在状态机的模型里面是什么. 不难理解, 计算机正是通过执行指令的方式来改变自身状态的, 比如执行一条加法指令, 就可以把两个寄存器的值相加, 然后把结果更新到第三个寄存器中; 如果执行一条跳转指令, 就会直接修改PC的值, 使得计算机从新PC的位置开始执行新的指令. 所以在状态机模型里面, 指令可以看成是计算机进行一次状态转移的输入激励.

ICS课本的1.1.3小节中介绍了一个很简单的计算机, 这个计算机有4个8位的寄存器, 一个4位PC, 以及一段16字节的内存(也就是存储器), 那么这个计算机可以表示比特总数为 `B = 4*8 + 4 + 16*8 = 164` , 因此这个计算机总共可以有 `N = 2^B = 2^164` 种不同的状态. 假设这个在这个计算机中, 所有指令的行为都是确定的, 那么给定 `N` 个状态中的任意一个, 其转移之后的新状态也是唯一确定的. 一般来说 `N` 非常大, 下图展示了 `N=50` 时某计算机的状态转移图.

现在我们就可以通过状态机的视角来解释“程序在计算机上运行”的本质了: 给定一个程序, 把它放到计算机的内存中, 就相当于在状态数量为 `N` 的状态转移图中指定了一个初始状态, 程序运行的过程就是从这个初始状态开始, 每执行完一条指令, 就会进行一次确定的状态转移, 也就是说, 程序也可以看成一个状态机! 这个状态机是上文提到的大状态机(状态数量为 `N`)的子集.

例如, 假设某程序在上图所示的计算机中运行, 其初始状态为左上角的8号状态, 那么这个程序对应的状态机为

```
8->1->32->31->32->31->...
```

这个程序可能是:

```
// PC: instruction     | // label: statement
0: addi r1, r2, 2      | pc0: r1 = r2 + 2;
1: subi r2, r1, 1      | pc1: r2 = r1 - 1;
2: nop                 | pc2: ;  // no operation
3: jmp 2               | pc3: goto pc2;
```

从状态机视角理解程序运行

以上一小节中 `1+2+...+100` 的指令序列为例, 尝试画出这个程序的状态机.

这个程序比较简单, 需要更新的状态只包括 `PC` 和 `r1` , `r2` 这两个寄存器, 因此我们用一个三元组 `(PC, r1, r2)` 就可以表示程序的所有状态, 而天真画出内存的具体状态, 初始状态是 `(0, x, x)` , 此处的 `x` 表示未初始化. 程序 `PC=0` 处的指令是 `mov r1, 0` , 执行完之后 `PC` 会指向下一条指令, 因此下一个状态是 `(1, 0, x)` , 如此类推, 我们可以画出执行前3条指令的状态转移过程:

```
(0, x, x) -> (1, 0, x) -> (2, 0, 0) -> (3, 0, 1)
```

请你尝试继续画出这个状态机, 其中程序中的循环只需要画出前两次循环和最后两次循环即可.

通过上面必做题的例子, 你应该更进一步体会到“程序是如何在计算机上运行”了. 我们其实可以从两个互补的视角来看待同一个程序:

- 一个是以代码(或指令序列)为表现形式的静态视角, 大家经常说的“写程序”/“看代码”, 其实说的都是这个静态视角. 这个视角的一个好处是描述精简, 分支, 循环和函数调用的组合使得我们可以通过少量代码实现出很复杂的功能. 但这也可能会使得我们对程序行为的理解造成困难.
- 另一个是以状态机的状态转移为运行效果的动态视角, 它直接刻画了“程序在计算机上运行”的本质. 但这一视角的状态数量非常巨大, 程序代码中的所有循环和函数调用都以指令的粒度被完全展开, 使得我们难以掌握程序的整体逻辑. 但对于程序的局部行为, 尤其是从静态视角来看难以理解的行为, 状态机视角可以让我们清楚地了解相应的细节.

程序的状态机视角有什么好处？

有一些程序看上去很简单, 但行为却不那么直观, 比如递归. 要很好地理解递归程序在计算机上如何运行, 从状态机视角来看程序行为才是最有效的做法, 因为这一视角可以帮助你理清每一条指令究竟如何修改计算机的状态, 从而实现宏观上的递归语义. ICS理论课的第三章会专门分析其中的细节, 我们在这里就不展开讨论递归的具体行为了.

“程序在计算机上运行”的微观视角: 程序是个状态机

“程序是个状态机”这一视角对ICS和PC来说都是非常重要的, 因为“理解程序如何在计算机上运行”就是ICS和PA的根本目标. 至于这个问题的宏观视角, 我们将会在PA的中期来介绍.

这一小节的文字对你来说应该不难理解, 但如果你将来没有养成从状态机视角理解程序行为的意识, 你可能会感到PA非常困难, 因为在PA中你需要不断地和代码打交道. 如果你不能从微观视角理解某些关键代码的行为, 你也无法从宏观视角完全弄清整程序究竟是如何运行的.

- 目录
- 概述
- 概述和功能
- 支持平台
- 支持平台概述
- 支持平台管理
- 设置及注册

RTFSC

Rtfsc是Linux、TRIMS和其他系统的简称，以至于程序行的过程和它，于是不以自己概念，所以不同的程序设计师对Rtfsc的理解不一样，有的是Read the fucking source code，为了更理解此，我们在Rtfsc的基础上进行更进一步的“Transcribe”（翻译），即RtfsCM（中国的RtfsC）。

程序代码分析

目前我们可以看到的是，某种在线下是各自独立的，但是它们是在将来进行合并，MDS和Transcribe的RtfsC，RtfsC的目前是将其 `/usr/local/bin` 目录下，并通过 `/usr/local/lib/rtfsc.ini` 文件，`/usr/local/var` 之外的其它目录的基本工作和基本数据，以下是一个文件管理器的RtfsC的目录结构：

- 程序管理目录即是RtfsCM的目录，其中的文件名称和RtfsC一样，它们都是有相同的基本功能
- 原有目录的名称，即原有RtfsC的目录，不同的是每个目录，各自有自己的程序目录，以及其基本工作的数据，它们的目录名称是以不同的目录名称命名，即：一个目录的RtfsC的目录名称是 `/usr/local/var/rtfsc/rtfsc`，其中的数据目录名称是相对的对应于程序目录

此下目录的工作内容分为两个部分，一部分是基本数据，目前在其基本数据目录下的文件有：“键值”数据文件和其他一个非常重要的概念，即基本信息不的信息数据表，文件名称：rtfsc.dat，程序的名称内容，但一个文件一次性读取的数据量，目前是一次性读取的，其中的数据，其中的数据，其中的数据

导航
解析命令
单步执行
打印寄存器
扫描内存

基础设施: 简易调试器

基础设施 - 提高项目开发的效率

在PA中, 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` / `quartus` 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, 其中需要处理很多步骤上的NPC问题. 为了生成一个质量还不错的比特文件, 硬件开发工具需要付出比 `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
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
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
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恶梦的结局: 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, 你沮丧满面地发现, 原来是扫描内存的功能有bug!

如果你想避免类似的恶梦结局, 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

为了让简易调试器易于使用, 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. 不过既然寄存器的结构是ISA相关的, 我们希望能为简易调试器屏蔽ISA的差异. 框架代码已经为大家准备了如下的API:

```
// nemu/src/isa/$ISA/reg.c
void isa_reg_display(void);
```

执行 `info r` 之后, 就调用 `isa_reg_display()`, 在里面直接通过 `printf()` 输出所有寄存器的值即可. 如果你从来没有使用过 `printf()`, 请RTFM或者STFW. 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, 就可以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. 问题是, 我们要如何访问客户计算机的内存数据呢? (答案早就说了哦)

实现了扫描内存的功能之后, 你可以打印 `0x100000` 或者 `0x80100000` 附近的内存, 你应该会看到程序的代码, 和内置客户程序的内容进行对比, 检查你的实现是否正确.

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.

为了查看单步执行的效果, 你可以把 `nemu/include/common.h` 中的 `DEBUG` 宏打开, 这样以后NEMU会把单步执行的指令打印出来(这里面埋了一些坑, 建议你RTFSC看看指令是在哪里被打印的).

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. 如果你已经忘记了某些注意事项, 重新去阅读一遍也是应该的.

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.

温馨提示

PA1阶段1到此结束.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布 此页最博订于: 2021-04-28 02:28:46

第 6 章

数学表达式语言

引言

数学表达式语言

表达式求值

调度分析

数学表达式求值

数学表达式求值的实现

数学表达式求值的实现

数学表达式求值的实现

温馨提示

PA1阶段2到此结束.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

此页面修订于: 2021-04-19 09:50:01

≡ 导航
监视点
扩展表达式求值的功能
实现监视点
调试工具与原理
断点

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，但这些表达式都是由常数组成的，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。

关于获取寄存器的值，这是然是一个ISA相关的功能，相关代码已经准备了如下的API：

```
// memory/src/isa/$ISA/reg.c
word_t isa_reg_str2val(const char *s, bool *success);
```

它用于返回名字为 `s` 的寄存器的值，并设置 `success` 指示是否成功。

还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 `*`，在进行语法识别之前，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，其实要识别它们也不难，结合一个表达式，你也能将它们区别开来，实际上，我们只需要 `*` 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(7, 7);
```

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，`*` 的识别时你来说应该会比较难了。

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，我们总是从客户计算机的内存中读出一个 `uint32_t` 类型的整数。

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 `||` 都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。

risc64中的表达式求值

由于rocc64是一个64位的ISA，其寄存器的值是64位的，因此为了保证正确的寄存器值，你需要把表达式求值的结果解释成 `uint64_t` 类型。

测试的局限

我们之前实现了一个表达式生成器，但是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，表达式生成器不能满足我们对所有需求了，这是因为在C程序中，寄存器的值又并不存在，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/src/monitor/debug/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，同时我们中的"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 = NULL, *free_ = NULL;
```

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初始化。

实现监视点的管理

为了使用监视点，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的池们被其它函数调用，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，此时可以调用 `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

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：

- 当用户给出一个终监视表达式时，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，就对所有终监视的表达式进行求值(你之前已经实现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，程序就应触发了监视点的暂停下来，你需要将 `nemu_state.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的信息，型于要打印什么，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。

由于监视点的功能需要在 `cpu_exec()` 的每次循环中都进行检查，这会对NEMU的性能带来较为明显的开销，因此我们把监视点的检查放在一个条件编译内部，你在 `DEBUG` 宏被定义时，才进行监视点的检查，`DEBUG` 宏可以在 `nemu/include/common.h` 中定义，当你不需要使用监视点时，可以关闭它来提高NEMU的性能。

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

在实现监视点的过程中，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，程序会这样的，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，然而下到将来对 `p` 进行了解引用的时候，才会触发段错误，程序随机崩溃。

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: 最直接的错误表现，例如程序触发了段错误

调试其实就是从观测到的failure一步步回溯寻找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，这其实就是测试要干的事情，所以我们在上一节中加入了表达式生成器的内容，某种意义来说过程，但是的实现也为你提供了更多的测试手段，你开不要再了测试表达式的功能的有fault都转变成error了，因为实现该子测试的覆盖度，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。
- 你应如何调试的监视点实现？

  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，对于实验来说，将来必用边测也是一种诱导过去的方法，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，你已经实现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，这也都是免费的午餐，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性，就一定能在运行时刻将不满足这些特征的error尽最下来，例如插桩的实现，我们只需要在代码中插入一些简单的 `assert()` (例如指针解引用时不为空)，就能够几乎完美的捉住，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()` 需要频繁输出的结果人工判断是否正确，在使利程度上相对于 `assert()` 的自动判断就逊色了不少。
  - GDB：随时随地观测程序的行为状态，调试器是最强大的工具，你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

尝试编写一个触发段错误的程序，然后在GDB中运行它，你发现GDB能为你提供哪些有用的信息吗？

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的，通过 `printf()` 输出可疑的变量，期望能观测到error
- `printf()`：不易观测error的，通过GDB或程序的调试功能

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：

```
mw 0x8048 0x3
```

其中 `0x8048` 为设置断点的地址，这样程序执行到 `0x8048` 的位置时就会暂停下来。

如何提高断点的效率(提交二周目思考)

如果你在这样做大一统的程序(和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"！如果你想提升这一神秘的奥妙，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：

一个问题两个解

`x86的 int3` 指令不等价于任何操作数，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？假设有一种与x86体系结构类似的ISA，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fly-x86中，上述文章中的断点机制还可以正常工作吗？为什么？

核心阿逆的断点

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的断世今生

你已经对NEMU的工作方式有所了解了，事实上在NEMU诞生之初，NEMU曾经有一段时间并不叫NEMU，而是NJDB(NJU Debugger)，但其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(或monitor)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？

如何阅读手册

在以后的PA中, 你需要反复阅读之前获得的生存手册. 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. 即使你选择的ISA并不是x86, 也可以尝试去查阅这个概念.

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:

- 送分题 我选择的ISA是 `_____` .
- 程序是个状态机 画出计算 `1+2+...+100` 的程序的状态机, 具体请参考这里.
- 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来直接调试客户程序, 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.
- RTFM 理解了科学查阅手册的方法之后, 请你尝试在你选择的ISA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:
  - x86
    - 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?
  - mips32
    - mips32有哪几种指令格式?
    - CP0寄存器是什么?
    - 若除法指令的除数为0, 结果会怎样?
  - riscv32
    - riscv32有哪几种指令格式?
    - LUI指令的行为是什么?
    - mstatus寄存器的结构是怎么样的?
- 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 `pa0` 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?
- RTFM 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` 将工程提交到指定网站.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

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

阶段性安排: 提交截止日期见这里

- task PA2.1: 在NEMU中运行第一个C程序 `dummy`
- task PA2.2: 实现更多的指令, 在NEMU中运行所有 `cpu-tests`
- task PA2.3: 运行FCEUX, 提交完整的实验报告

提交说明: 见这里

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

导航

不停计算的机器
取指(instruction fetch, IF)
译码(instruction decode, ID)
执行(execute, EX)
更新PC

YEMU: 一个简单的CPU模拟器

热身结束, 进入真正的PA

如果说PA1只是编程复习的热身运动, 那么PA2就是正片的开始了. 在PA2中, 你会感觉新代码和工程细节如潮水般袭来, 你需要不断地读讲义, RTFM, RTFSC, 并整理其中的细节, 一开始效率会比较低, 一旦掌握了这些细节, 后面就是重复性的工程活了.

此外, 如果说PA1中提到的"不要纠缠于和当前进度无关的代码" 是为了保护你的幼小心灵, 在PA2中你就需要勇敢地去探索了: 讲义不再像PA1那样详细指出每一处你值得阅读的代码. 事实上, PA中的所有代码都值得你去阅读, 在PA2结束之后, 你需要做到可以理解NEMU中的每一处细节, 随着你对这些代码细节的了解变得深入, 就算是调bug你也会觉得多了几分轻松. 相反, 如果你仍然抱着"测试能过就行", "这里不算分, 关我P事"的心态, 很快你就会发现连bug都不知道怎么调了.

所以, 不要再偷懒了.

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PC指示的存储器位置取出指令;
  执行指令;
  更新PC;
}
```

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. 为了便描述更加清晰, 我们借助指令周期中的一些概念来说明指令执行的过程.

取指(instruction fetch, IF)

要执行一条指令, 首先要拿到这条指令. 指令究竟在哪里呢? 还记得冯诺依曼体系结构的核心思想吗? 那就是"存储程序, 程序控制". 你以前听说这两句话的时候可能没有什么概念, 现在是实践的时候了. 这两句话告诉你, 指令在存储器中, 由PC指出当前指令的位置. 事实上, PC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PC指向的指令从内存读入到CPU中.

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"操作码"和"操作数"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PC

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PC的值, 让PC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.

YEMU: 一个简单的CPU模拟器

我们还是以ICS课本的1.1.3小节中介绍的简单计算机为例, 来展示如何用C语言实现一条指令的执行. 这个计算机有4个8位的寄存器, 一个4位PC, 以及一段16字节的内存. 它支持R型和M型两种指令格式, 4条指令, 其指令手册如下:

```
                                                    4  2  0
          |                          |        | +----+--+--+
mov   rt,rs | R[rt] <- R[rs]         | R-type | |0000|rt|rs|
          |                          |        | +----+--+--+
          |                          |        | +----+--+--+
add   rt,rs | R[rt] <- R[rs] + R[rt] | R-type | |0001|rt|rs|
          |                          |        | +----+--+--+
          |                          |        | +----+--+--+
load  addr  | R[0] <- M[addr]        | M-type | |1110| addr|
          |                          |        | +----+--+--+
          |                          |        | +----+--+--+
store addr  | M[addr] <- R[0]        | M-type | |1111| addr|
          |                          |        | +----+--+--+
```

我们可以根据指令手册用C语言编写出这个简单计算机的模拟器YEMU(袁妈模拟器):

```
#include <stdint.h>
#include <stdio.h>

#define NREG 4
#define NMEM 16

// 定义指令格式
typedef union {
  struct { uint8_t rs : 2, rt : 2, op : 4; } rtype;
  struct { uint8_t addr : 4      , op : 4; } mtype;
  uint8_t inst;
} inst_t;

#define DECODE_R(inst) uint8_t rt = (inst).rtype.rt, rs = (inst).rtype.rs
#define DECODE_M(inst) uint8_t addr = (inst).mtype.addr

uint8_t pc = 0;       // PC, C语言中没有4位的数据类型, 我们采用8位类型来表示
uint8_t R[NREG] = {}; // 寄存器
uint8_t M[NMEM] = {   // 内存, 其中包含一个计算z = x + y的程序
  0b11100110,  // load  6#      | R[0] <- M[y]
  0b00000100,  // mov   r1, r0  | R[1] <- R[0]
  0b11100101,  // load  5#      | R[0] <- M[x]
  0b00010001,  // add   r0, r1  | R[0] <- R[0] + R[1]
  0b11110111,  // store 7#      | M[z] <- R[0]
  0b00010000,  // x = 16
  0b00100001,  // y = 33
  0b00000000,  // z = 0
};

int halt = 0; // 结束标志

// 执行一条指令
void exec_once() {
  inst_t this;
  this.inst = M[pc]; // 取指
  switch (this.rtype.op) {
  //  操作码译码        操作数译码          执行
    case 0b0000: { DECODE_R(this); R[rt]  = R[rs];   break; }
    case 0b0001: { DECODE_R(this); R[rt]  += R[rs];  break; }
    case 0b1110: { DECODE_M(this); R[0]   = M[addr]; break; }
    case 0b1111: { DECODE_M(this); M[addr] = R[0];   break; }
    default:
      printf("Invalid instruction with opcode = %x, halting...\n", this.rtype.op);
      halt = 1;
      break;
  }
  pc ++; // 更新PC
}

int main() {
  while (1) {
    exec_once();
    if (halt) break;
  }
  printf("The result of 16 + 33 is %d\n", M[7]);
  return 0;
}
```

理解YEMU如何执行程序

YEMU可以看成是一个简化版的NEMU, 它们的原理是相通的, 因此你需要理解YEMU是如何执行程序的. 具体地, 你需要

- 画出在YEMU上执行的加法程序的状态机
- 通过RTFSC理解YEMU如何执行一条指令

思考一下, 以上两者有什么联系?

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, PC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, 虽然这个只有几条指令的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.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

- 目录
- RTFM
- RTFM
- 目录Table of Contents
- 目录Table of Contents
- 目录Table of Contents

件的一种接口，所以有了PA5中实现至少一条指令的行为。

- 一方面，编译器需要根据ISA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。
- 另一方面，硬件开发者也需要根据ISA手册中的约定来设计可以正确执行指令的处理器. 如果处理器不按照手册约定来执行指令，处理器设计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，给约定的实现方式带来一定的自由度. 例如, C语言标准规定, 整数执行除数为0的话, 结果是未定义的, x86的除法指令会在除数除数为0的话产生异常, 而在类似MIPS的除法指令则要简单暴力: 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电路的时候, 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C语言标准的约定.

未定义行为其实就是一个坏东西, 比如野指针的解引用, 会发生什么完全无法预料, 还有你经常使用的 `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. 还有一种未定义行为, 就是你可以假定某是不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, 看完之后你就会眼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是了解接口行为的定义的过程: 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? 有哪些的来条件会涉及到? 哪些情况下会报什么错误? 那些行为是UB? 只有完全理解并遵守它们, 才能正确实现使用意图的对象, 大至系统设计原则, 小到一个 `memcpy()` 的行为, 都是含蓄的定与遵守的原则, 理解这些法则, 也是理解计算机系统的不二途径.

UB, 编译优化和dataslab

听闻大师的lab1(dataslab)曾经因为使用了debian10的新版gcc而翻车, 后来了解到, 是因为dataslab的参考代码中含有int整数溢出的UB, debian 10的gcc则用了UB进行编译优化, 导致参考代码在编译时生成了错误的参考答案.

C语言标准规定, int整数溢出的行为是未定义的, 但大部分程序员并不知道这一约定, 甚至连市面上流行的C语言教科书都认为int整数溢出的结果是wrap around. dataslab是CMU设计的实验, 但很作者也会偏写出含有UB的代码, 说明即作者对UB的理解也并未到位. 而在编译器中, 这些UB并未被触发, 但UB毕竟是UB, 只能说明作者写代码的时候没有充分理解C语言标准.

这个例子说明了: 对整数溢出的实现行为进行了规定, 并且在实际应用中并非出现了大量整数溢出的例子进行分析, 推荐大家阅读, 论文中提到有一个第二次应用到形式化RCoq(Windows)的函数库Safeint用于避免整数溢出, 但这个函数库自身的代码就被论文作者检测出整数溢出导致的UB.

这些例子给我们的启示是: **我们不仅要懂编写通过测试的代码, 还要懂编写通过测试的代码的规范: 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? 有哪些的来条件会涉及? 哪些情况下会报什么错误? 那些行为是UB? 只有完全理解并遵守它们, 才能正确实现使用意图的对象, 大至系统设计原则, 小到一个 `memcpy()` 的行为, 都是含蓄的定与遵守的原则, 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很快决策的优化机会.

为了提高性能, 我们在 `Operand` 结构体中定义了一个RTL寄存器的指针 `preg` , 用于直接指向那些已经存在的RTL寄存器. 例如如果在过程中频繁的时候发现操作数是 `eax` 寄存器, 那么只需要通过让 `preg` 指向 `cpu.eax` , 将来就可以直接通过 `preg` 来访问到正确的操作数了, 而不需要通过 `rtl_wv()` 来将 `cpu.eax` 读到到处, 从而避免引入额外的开销. 根据代码提示了 `dsrc1` , `dsrc2` 和 `odest` 这三个宏, 分别用于方便地访问 `id_src1` , `id_src2` 和 `id_dest` 对应的 `preg` .

虽然使用指针可以提高性能, 但也会有以下几点需要注意:

- 如果源操作数是一个立即数, 那么它就没有RTL寄存器可以被 `preg` 来指向. 此时译码过程会通过 `rtl_li` 将立即数读入到操作数结构体 `Operand` 的 `val` 成员, 然后再让 `preg` 指向 `val` .
- 如果源操作数是x86寄存器的字节或半字访问, 那么其类型并不是一个完整的 `rtlreg_t` , 不能直接被 `preg` 指向. 此时译码过程会通过 `rtl_lr` 将相应的值读入到操作数结构体 `Operand` 的 `val` 成员, 然后再让 `preg` 指向 `val` .
- 如果目标操作数是mips32或者risccv32的9号寄存器, 那么执行过程中可能会将结果写入了9号寄存器, 下次将会从9号寄存器中读出非0内容, 从而违反了ISA手册的规定. 为了修复这个问题, 在mips32和risccv32的 `isa_exec_once()` 函数中执行完一条指令后, 都会调用 `reset_zero()` 函数来将0号寄存器重新设置为0.
- 一般来说源操作数是不应该被修改的, 如果你打算把 `dsrc1` 或者 `dsrc2` 作为RTL指令的 `dest` , 你需要确认这是否符合指令本身的语义.
- 如果一条指令的源操作数和目的操作数一样, 你可能需要注意一些RTL指令之间的顺序.

如果你理解不了上面的注意事项, 不必担心, 你可以把编写RTL作为一次指针的练习, 遇到问题的时候你就会对指针有更加深刻的理解了.

实现新指令

对译码, 执行和操作数宽度的解释实现以及RTL的引入, 对在NEMU中实现客户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 `exec.c` 的 `switch-case` 语句中填写正确的译码辅助函数, 执行辅助函数以及操作数宽度
2. 用RTL实现正确的执行辅助函数, 需要注意使用RTL伪指令时要遵守上文提到的小技巧

感觉RTL指令好奇怪不够用了?

其实是存在够用的实现方式的, 你可以尝试挑战一下这个目标. 当然, 如果实在坚持不了的话, 你也可以定义更多的RTL指令寄存器, 如 `t1` , `t2` , `s3` ...

事实上, 如果定义了过多的RTL指令寄存器, 可能会对PA5的性能优化带来一些负面影响. 不过PA5并不记入成绩, 所以你也不用为此感到担忧.

框架代码定义了一些译码辅助函数和执行辅助函数, 你可以很方便地使用它们.

假心的框架代码

这个"一些"对不同的ISA来说, 还是有很大差别的: 对x86来说其实是"绝大部分", 而对mips32和risccv32来说其实应"少量".

你应该能感受到, 这节讲义内容为x86提供了很多帮助, 而对mips32和risccv32的提示却很少, 甚至提供的某些内容反而帮倒忙. 你也许会觉得这是yzh偏心, 但其实这恰恰是这x86的复杂(像看不明白ISA的手册就知道了). 考虑到这一不平性, 选择mips32和risccv32的同学就需要独立编写大部分的译码辅助函数 (其实很多, 因为RISC指令集的格式都比较规整)和执行辅助函数 (其实很简单, 因为NEMU中的RTL就是参考risccv来定义的, 也可以看成是一种RISC指令集). 我们希望无论选择你何种ISA, 为了获取相同的分数, 你都需要付出一定的努力. 如果你对公平性有更好的建议, 请联系yzh.

你可以与选择不同ISA的同学交流, 但我们会要求你能在二周目的时候选择不同的ISA进行新的攻略, 这样你也许就能对不同ISA之间的区别有更深刻的体会了.

如果你读过上文的"展阅读材料中关于RISC-V的一条指令的故事", 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传奇, NEMU在经历了第二次重构之后, 也终于引入了RISC风格的RTL来实现指令, 这也许是第笑了之的安排吧.

RTFSC理解指令执行的过程

这一小节的细节非常多, 你可能需要多次阅读讲义和代码才能理解每一处细节. 根据往届学长学姐的反馈, 一种有效的理解方法是通过做笔记的方式来整理这些细节. 事实上, 配合GDB食用效果更佳.

为了避免你长时间对代码的理解没有任何进展, 我们就增加一道必答题吧:

> 请整理一条指令在NEMU中的执行过程.

除了 `device` 相关以及一些未实现的功能(如 `intr.c` 和 `mmu.c`), NEMU的其它代码你都已经有能力理解了. 因此不要觉得讲义中没有提到的文件就不需要看, 尝试尽可能地理解每一处细节吧! 在你遇到bug的时候, 这些细节就会成为帮助你调试的线索.

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. 首先克隆一个新的子项目 `am-kernels` , 里面包含了一些测试程序:

```
cd ics2020
bash init.sh am-kernels
```

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程序是 `am-kernels/tests/cpu-tests/tests/dummy.c` , 它什么都不做就直接返回了.

准备交叉编译环境

如果你选择的ISA不是x86, 你还需要准备相应的gcc和binutils, 才能正确地进行编译.

- mips32
  - `apt-get install g++-mips-linux-gnu binutils-mips-linux-gnu`
- risccv32(64)
  - `apt-get install g++-riscv64-linux-gnu binutils-riscv64-linux-gnu`
  - 如果你选择是risccv32, 则需要使用sudo权限修改以下文件:

```
--- /usr/riscv64-linux-gnu/include/bits/wordsize.h
+++ /usr/riscv64-linux-gnu/include/bits/wordsize.h
@@ -25,5 +25,5 @@
#if __riscv_xlen == 64
# define __WORDSIZE_TIME64_COMPAT32 1
#else
-# error "rv32i-based targets are not supported"
+# define __WORDSIZE_TIME64_COMPAT32 0
#endif
```

在 `am-kernels/tests/cpu-tests/` 目录下键入

```
make ARCH=ISA-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, `abstract-machine` 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(以x86为例):

```
invalid opcode(eip = 0x001000ba): c8 05 00 00 00 90 31 c0 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at pc = 0x001000ba is not implemented.
2. Something is implemented incorrectly.
Find this pc value(0x001000ba) 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.

为什么执行了未实现指令会出现上述错误信息

RTFSC, 理解执行未实现指令的时候, NEMU具体会怎么做.

我选择x86, 但出错信息中显示的字节序列有所不同

这很可能是你选择了其它Linux发行版, 由于gcc版本的差异导致的. 此时按照你的gcc编译结果来实现指令即可.

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(`am-kernels/tests/cpu-tests/build/dummy-ISA-nemu.txt`)中. 以x86为例, 查看反汇编结果, 你发现只需要添加 `call` , `push` , `sub` , `xor` , `ret` 五条指令就可以了. 每一条指令还有不同的形式, 根据ISS法则, 你可以先实现只在 `dummy` 中出现的指令形式, 通过编译的 `opcode` 可以确定具体的形式.

交叉编译工具链

如果你选择的ISA不是x86, 在查看客户程序的二进制信息时 `objdump` 、 `readelf` 等时, 需要使用相应的交叉编译版本, 如 `mips-linux-gnu-objdump` 、 `risccv64-linux-gnu-readelf` 等.

去掉endbr32指令

经测试, 如果系统的CPU版本比较新(如在十代酷睿), 在Ubuntu20.04上会默认编译出endbr32指令, 如果你属于这种情况, 请参考这篇文章添加相应的编译选项来去掉这条新指令, 添加后, 在 `am-kernels/tests/cpu-tests/` 目录下执行 `make clean` , 并在 `abstract-machine/` 目录下执行 `make clean-all` , 然后重新编译测试程序.

这里要再次强调, **你务必通过RTFM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(包括例什么事, 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后的调试带来巨大的麻烦.**

再提供一些x86的提示

- `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, 程序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` | 现在只需要实现 `PUSH r32` 和 `PUSH imm32` 的形式就可以了, 它们可以很容易地通过 `rtl_push` 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. 如果你没有看到这一信息, 说明你的指令实现不正确, 你可以使用PA1中实现的简易调试器帮助你调试.

温馨提示

PA2阶段1到此结束.

运行时环境

现代高级语言编译器（如 clang 或 gcc），用户的源代码并不直接编译成机器码了，一般会生成汇编语言（如 AMD64 的汇编语言），然后调用第三方的汇编器（如 GNU 的 as 或 Intel 的 nasm）生成目标文件（object file），最后调用链接器（如 GNU 的 ld 或 Intel 的 xilink）将目标文件和静态库（static library）链接成可执行文件（executable）。这个流程中，编译器（compiler）、汇编器（assembler）、链接器（linker）共同构成了一个完整的工具链（toolchain）。这个工具链生成的可执行文件，在操作系统（OS）上运行时，还需要依赖一个运行时环境（runtime）。这个运行时环境通常会提供内存管理、异常处理、线程调度、动态加载等功能。本章以 AMD64 平台为例，说明运行时环境的组成部分，以及它们如何与操作系统（OS）和硬件（CPU）交互。

程序的内存组织，决定了程序在内存中如何分布代码、数据、堆、栈。这个内存布局，通常由操作系统（OS）提供的虚拟内存（virtual memory）来实现。这个虚拟内存，会为每个进程（process）分配一个独立的、连续的、从零开始的地址空间。这个空间，通常会被分成几个区域，如下图：

这个虚拟内存，实现了多个目标：不同的进程（process）之间，互相隔离（isolation）；可以设置不同的权限（permission），比如 .text 区域是只读（read-only）和可执行（executable）的，而 .data 区域是可读写（read-write）的，但不可执行（non-executable）；可以实现内存共享（shared memory）和动态加载（dynamic loading）。

这个虚拟内存，通常会被进一步分成两个逻辑部分：用户空间（user space）和内核空间（kernel space）。用户空间，是应用程序（application）运行的地方，它的权限，通常是受限的（restricted）；内核空间，是操作系统（OS）内核（kernel）运行的地方，它的权限，通常是特权的（privileged）。这个区别，是现代操作系统（OS）安全性（security）和稳定性（stability）的基石。

本章要点

- 理解程序的内存组织，是理解运行时环境的基础。
- 理解 AMD64 平台的内存组织，是理解 RTFSSCI 的运行时函数的基础。

AMD64 的内存组织

AMD64 的内存组织，是 RTFSSCI 的运行时函数的基础。这个内存组织，主要包括：

- 代码区（text segment）：存放机器码（machine code）。
- 只读数据区（rodata）：存放字符串（string）、浮点数（floating point）等常量（constant）。
- 可读写数据区（data）：存放已初始化（initialized）的全局变量（global variable）。
- 未初始化数据区（bss）：存放未初始化（uninitialized）的全局变量（global variable）。
- 堆（heap）：存放动态分配（dynamic allocation）的内存。
- 栈（stack）：存放函数（function）的局部变量（local variable）、函数参数（function parameter）、函数返回值（return value）、寄存器（register）的备份（backup）。

本章要点

- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

寄存器的用途

AMD64 的寄存器（register）的用途，是 RTFSSCI 的运行时函数的基础。这些寄存器，主要包括：

- 通用寄存器（general purpose register）：rax、rbx、rcx、rdx、rsi、rdi、rbp、rsp、r8、r9、r10、r11、r12、r13、r14、r15。
- 浮点寄存器（floating point register）：xmm0、xmm1、xmm2、xmm3、xmm4、xmm5、xmm6、xmm7。
- 段寄存器（segment register）：cs、ds、es、fs、gs、ss。
- 控制寄存器（control register）：cr0、cr2、cr3、cr4、cr8。
- 调试寄存器（debug register）：dr0、dr1、dr2、dr3、dr6、dr7。
- 浮点控制寄存器（floating point control register）：mxcsr。

本章要点

- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

函数调用的流程

函数调用（function call）的流程，是 RTFSSCI 的运行时函数的基础。这个流程，主要包括：

- 参数（parameter）的传递（pass）：通过寄存器（register）或栈（stack）。
- 返回值（return value）的传递（pass）：通过寄存器（register）。
- 栈（stack）的管理（manage）：通过寄存器（register）。

本章要点

- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

本章小结

- 程序的内存组织，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

下一章预告

下一章，我们将开始研究 RTFSSCI 的运行时函数。

参考资料

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming
- AMD64 Architecture Programmer's Manual Volume 2: System Programming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions
- AMD64 Architecture Programmer's Manual Volume 6: Secure Virtual Machine
- System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.7
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2C: Instruction Set Reference, VEX
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2D: Instruction Set Reference, EVEX
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers

本章要点

- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

本章小结

- 程序的内存组织，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的 ABI（application binary interface），是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的寄存器（register）的用途，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 的函数调用（function call）的流程，是理解 RTFSSCI 的运行时函数的基础。

下一章预告

下一章，我们将开始研究 RTFSSCI 的运行时函数。

参考资料

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming
- AMD64 Architecture Programmer's Manual Volume 2: System Programming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions
- AMD64 Architecture Programmer's Manual Volume 6: Secure Virtual Machine
- System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.7
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2C: Instruction Set Reference, VEX
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2D: Instruction Set Reference, EVEX
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers

程序、运行时环境与 AMD

运行时环境

现代高级语言编译器（如 clang 或 gcc），用户的源代码并不直接编译成机器码了，一般会生成汇编语言（如 AMD64 的汇编语言），然后调用第三方的汇编器（如 GNU 的 as 或 Intel 的 nasm）生成目标文件（object file），最后调用链接器（如 GNU 的 ld 或 Intel 的 xilink）将目标文件和静态库（static library）链接成可执行文件（executable）。这个流程中，编译器（compiler）、汇编器（assembler）、链接器（linker）共同构成了一个完整的工具链（toolchain）。这个工具链生成的可执行文件，在操作系统（OS）上运行时，还需要依赖一个运行时环境（runtime）。这个运行时环境通常会提供内存管理、异常处理、线程调度、动态加载等功能。本章以 AMD64 平台为例，说明运行时环境的组成部分，以及它们如何与操作系统（OS）和硬件（CPU）交互。

程序的内存组织，决定了程序在内存中如何分布代码、数据、堆、栈。这个内存布局，通常由操作系统（OS）提供的虚拟内存（virtual memory）来实现。这个虚拟内存，会为每个进程（process）分配一个独立的、连续的、从零开始的地址空间。这个空间，通常会被分成几个区域，如下图：

这个虚拟内存，实现了多个目标：不同的进程（process）之间，互相隔离（isolation）；可以设置不同的权限（permission），比如 .text 区域是只读（read-only）和可执行（executable）的，而 .data 区域是可读写（read-write）的，但不可执行（non-executable）；可以实现内存共享（shared memory）和动态加载（dynamic loading）。

这个虚拟内存，通常会被进一步分成两个逻辑部分：用户空间（user space）和内核空间（kernel space）。用户空间，是应用程序（application）运行的地方，它的权限，通常是受限的（restricted）；内核空间，是操作系统（OS）内核（kernel）运行的地方，它的权限，通常是特权的（privileged）。这个区别，是现代操作系统（OS）安全性（security）和稳定性（stability）的基石。

本章要点

- 理解程序的内存组织，是理解 RTFSSCI 的运行时函数的基础。
- 理解 AMD64 平台的内存组织，是理解 RTFSSCI 的运行时函数的基础。

AMD64 的内存组织

AMD64 的内存组织，是 RTFSSCI 的运行时函数的基础。这个内存组织，主要包括：

- 代码区（text segment）：存放机器码（machine code）。
- 只读数据区（rodata）：存放字符串（string）、浮点数（floating point）等常量（constant）。
- 可读写数据区（data）：存放已初始化（initialized）的全局变量（global variable）。
- 未初始化数据区（bss）：存放未初始化（uninitialized）的全局变量（global variable）。
- 堆（heap）：存放动态分配（dynamic allocation）的内存。
- 栈（stack）：存放函数（function）的局部变量（local variable）、函数参数（function parameter）、函数返回值（return value）、寄存器（register）的备份（backup）。

本章要点

- 理解 AMD64 平台的内存组织，是理解 RTFSSCI 的运行时函数的基础。

法请查阅 `man stdarg` .

实现sprintf

实现 `abstract-machine/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.

stdarg是如何实现的?

`stdarg.h` 中包含一些获取函数调用参数的宏, 它们可以看做是调用约定中关于参数传递方式的抽象. 不同ISA的ABI规范会定义不同的函数参数传递方式, 如果让你来实现这些宏, 你会如何实现?

重新认识计算机: 计算机是个抽象层

我们在PA1中介绍了"程序在计算机上运行"的微观视角: 程序是个状态机. 状态机视角可以从指令层次精确地描述程序运行的每一处细节, 但这丢失了程序的语义. 对于一些简单的程序, 你还可以把状态机画出来, 但面对一些复杂的程序, 状态机视角就不能帮助我们了. 为了更好地理解复杂程序, 我们需要从一个新的视角来切入.

我们先来讨论在TRM上运行的程序, 我们对这些程序的需求进行分类, 来看看我们的计算机系统是如何支撑这些需求的.

TRM	计算	内存申请	结束运行	打印信息
运行环境	-	malloc()/free()	-	printf()
AM API	-	heap	halt()	putch()
ISA接口	指令	物理内存地址空间	nemu_trap指令	I/O方式
硬件模块	处理器	物理内存	Monitor	串口
电路实现	fetch_decode_exec()	pmem[]	nemu_state	serial_io_handler()

- 计算. 这是程序最基本的需求, 以至于它甚至不属于运行时环境和AM的范畴. 所有计算相关的代码(顺序语句, 分支, 循环, 函数调用等), 都会被编译器编译成功能等价的指令序列, 最终在CPU上执行. 在NEMU中, 我们通过 `fetch_decode_exec()` 这个函数以及相应的辅助函数来实现"CPU执行指令"的功能.
- 内存申请. 有的程序需要在运行时刻动态地申请内存来使用. 和libc类似, klib提供了 `malloc()` 和 `free()` 来实现内存的动态管理(你将来会实现它们), 它们又会使用TRM中提供的API `heap` 来获得堆区的起始和末尾. 而 `heap` 的区间又是由ISA-平台这个二元组对应的物理内存地址空间来决定的. 这一地址空间对应着物理内存的大小, 在NEMU中, 它就是大数组 `pmem[]` 的大小.
- 结束运行. 一般程序都会有结束运行的时候, TRM提供了一个 `halt()` 的API来实现这一功能. 由于这个需求过于简单, 因此无需运行时环境提供更复杂的接口. `halt()` 的具体实现和ISA有关, 我们使用了人为添加的 `nemu_trap` 指令来实现这一点. 执行 `nemu_trap` 指令会让NEMU从CPU执行指令的循环中跳出, 返回到Monitor中, 这是通过设置Monitor中的一个状态变量 `nemu_state` 来实现的.
- 打印信息. 输出是程序的另一个基本需求. 程序可以调用klib中的 `printf()` 来输出, 它会通过TRM的API `putch()` 来输出字符. 不同的ISA-平台有不同的字符输出方式, 在$ISA-nemu中, `putch()` 通过I/O相关的指令把字符写入到串口, 最终在NEMU中通过 `serial_io_handler()` 将字符打印到终端. 关于输入输出的更多细节会在PA2的最后部分进行介绍.

"程序在计算机上运行"的宏观视角: 计算机是个抽象层

在之前的PA中, 我们一直都在强调"程序是个状态机"这个微观视角, 让大家站在底层来理解这个状态机是如何工作的. 现在我们终于有机会向高层远眺, 来理解计算机是如何把底层的功能经过层层抽象, 最终支撑程序的运行.

每一层抽象都有它存在的理由:

- 概念相同的一个硬件模块有着不同的实现方式, 比如处理器既可以通过NEMU中简单的解释方式来实现, 也可以通过类似QEMU中高性能的二进制翻译方式来实现, 甚至可以通过verilog等硬件描述语言来实现一个真实的处理器.
- ISA是硬件向软件提供的可以操作硬件的接口
- AM的API对不同ISA(如x86/mips32/riscv32)的接口进行了抽象, 为上层的程序屏蔽ISA相关的细节
- 运行时环境可以通过对AM的API进行进一步的封装, 向程序提供更方便的功能

这些抽象都是为了方便我们在各种各样的计算机系统中编写运行五花八门的程序, 你将要在PA3中运行的仙剑奇侠传, 也都是通过层层抽象分解成最基本的硬件操作, 最后以状态机的方式来运行.

PA究竟在做什么?

到此为止, 我们已经将PA中最重要的两个关于"程序在计算机上运行"的视角介绍完了:

- 微观视角: 程序是个状态机
- 宏观视角: 计算机是个抽象层

PA剩下的内容就是借鉴AM的启发, 按照计算机发展史的顺序给计算机硬件添加各种新特性, 并强化运行时环境的功能, 最后运行各种更复杂的程序. PA会把添加新特性的过程作为案例, 让大家不断地从这两个视角来理解"程序如何在计算机上运行". 具体地, 在PA2的最后, 我们会添加IOE来实现一个冯诺依曼计算机系统; 在PA3中, 我们会添加CTE, 来支持批处理系统的运行; 在最后的PA4中, 我们会添加VME, 来运行一个简单又酷炫的分时多任务系统.

我们在这里给出PA的一个全局概念图(图中的"运行时环境"包含了AM, klib, 甚至是OS和libc), 这个图的三维坐标轴总结了PA中3个最重要的结论, 也向大家展示了PA构建计算机系统的全过程. 大家在做实验的时候也可以多多思考: 我现在写的代码究竟位于哪一个抽象层? 代码的具体行为究竟是什么?

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

基础设施(2)

AMD为基础设施

最初的AMD基础设施(1)是时 2019年 2月，概要是高速数据流程，概要上是 流下数据流程的子步骤 ，而数据流程概述上，是 数据流程的概要流程 ，而数据流程概述下，是 SIMDAMD的Mode的子步骤 ，而SIMDAMD的Mode的概述概要上，是 数据流程概述的子步骤 ，而数据流程概述的子步骤 ，是 数据流程概述的Mode的子步骤 。

之前的基础设施(1)的，是指在软件Mode的SIMDAMD基础设施(1)的，它的目的是概要是下面的Mode的子步骤 ，而SIMDAMD的Mode的概述概要上，是 数据流程概述的子步骤 ，而数据流程概述的子步骤 ，是 数据流程概述的Mode的子步骤 。而SIMDAMD的Mode的概述概要上，是 数据流程概述的子步骤 ，而数据流程概述的子步骤 ，是 数据流程概述的Mode的子步骤 。

```
make(1) -> [ln] [ln] 0:ln] 1
```

当然在软件模拟基础设施概要上，

软件模拟的概述

为什么是概要 ？ 或 它的概述 ，是 概要是软件模拟的子步骤 ？

在概述上下文，我们 概要是 SIMDAMD 的 的 概要 ，我们 概要是 软件模拟的 的 概要 ，我们 概要是 软件模拟的 的 概要 ，我们 概要是 软件模拟

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.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

第一章 硬件
输入输出
设备驱动
内存管理
时钟
文件系统
进程管理与调度
NEMU中的输入输出
实现输入输出
设备IO
时钟和IOE
NEMU的IOE
IOE
TRM

为先进、优质、安全的生产过程而设计

计算机系统的设计应可以满足此要求：

1. 设计一个控制工具的完整系统（例如，工具）。
2. 在第1点的基础上，公司作出评估要求（例如）

要回答这个问题，你可能会问：这些要求是如何定义的呢？这些要求是谁的？这些要求是谁的？这些要求是谁的

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: 40小时

阶段性安排: 提交截止日期见这里

- task PA3.1: 实现自陷操作 `yield()`
- 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
- 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

提交说明: 见这里

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

❗ 没有调不出的bug，只有不理解的系统

PA3是整个PA的分水岭，从PA3开始，系统的复杂度开始逐渐上升，越来越多的抽象层将会加入进来，系统也开始逐渐完善，应用也将更加真实。

这意味着，bug的传播链条会变得更复杂，调试越来越考验你对系统行为的理解，如果你之前调bug都是能大胆，并没有发现过程中出现解系统的错误，你会发现在PA3中自己和大部分的更甚至在这课扩大，大部分整个系统的掌握越来越熟悉，而你则是超目前就越觉得寸难行，例如讲义和代码越来越看不懂，bug越来越神秘。

亡羊补牢的方法只有一个：真的不能再偷懒了。

批处理系统

我们在PA2中已经实现了一个功诸依赖计算机系统，并且已经在AM之把打字游戏和FCEUX运行起来了，有了IOE，就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FCEUX游戏。

事实上，早期的计算机就是这样工作的，系统管理程序运行算机如果一个程序结束后，这个程序就会立刻不顾的打开未片，计算机就会一直执行这个程序，直到程序结束或者是管理程序手动终止，然后再由管理程序来手动加载下一个程序，当年的程序也这么干如你玩的超级马丽这么酷炫，大多都是一些科学计算和物理模拟程序的话也没弹道轨迹计算。

后来人们就想，每次都要管理程序手动加载新的程序，这太麻烦了，但是却海诺烦躁计算机的工作方式，它的第一个特点就是：当计算机执行完一个程序后，就自动执行下一条指令，类似的，我们都不能让管理负事洗准备好一组程序，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路，有了批处理系统后，就可以解放管理员的双手了，用批处理系统的关键，就是要有一个后台程序，当一个新台程序执行结束的时候，后台程序就会自动去做一个新的后台程序来执行。

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，真的是操作系统！说起操作系统，也许你会说，我们学习的操作系统是以个C语的的内核，nanos-lite/include/common.h 中一些与实验进程相关的宏来控制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
├── README.md
├── resources
│   └── logo.txt    # Project-N logo文本
└── src
    ├── device.c    # 设备抽象
    ├── fs.c        # 文件系统
    ├── 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
    ├── resources.S # ramdisk内容和Project-N logo
    └── 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，它只是一个调用AM API的普通程序而已，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可以是架构无关的，这意味着，无论你之前选择的是哪一款ISA，都可以很容易地运行Nanos-lite，甚至你可以像开发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课程，但在这个Nanos-lite之后，我们还是会在某些地方使用“用户态程序”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，还是想不这么简单的话别着急，第一个课程的最后一节课，如果你打开了记事本文件，计算机上就会有3个记事本进程在运行，但最重中的记事本程序有两个：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. 打印Project-N的logo，并通过 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() 宏通过宏在 klib 中编写的 printf() 输出，最终会调用TRM的 putch()。
2. 调用 init_device() 对设备进行一些初始化操作，目前 init_device() 仅会简单调用 ioe_init()。
3. 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中对磁盘进行模拟是一个略显复杂工作，因此先让Nanos-lite把其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。
4. init_fs() 和 init_proc() ，分别用于初始化文件系统和创建进程，目前它们均未进行有意义的操作，可以忽略它们。
5. 调用 panic() 结束Nanos-lite的运行。

由于Nanos-lite本质上是一个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=$ISA-nemu run
```

即可，另外如前文所说，你也可以将Nanos-lite编译到 nanos.elf 文件，来帮助你进行调试。

❗ 操作系统是个C程序

你也许有点不敢相信，但是现代机的心和大空件无疑是各最这块一般的事实。甚至连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

那么和一般的C程序的区别在于，操作系统究竟有何特殊之处？答案是PA2之后，相信你就会略知一二了。

相就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，基本要连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

那么和一般的C程序的区别在于，操作系统究竟有何特殊之处？答案是PA2之后，相信你就会略知一二了。

相就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，基本要连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

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要求：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程序，属于程序内部的执行流切换，使用call/jal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PC从一个值修改成另一个值而已(需要保证栈的一致性)，因此，我们需要在使用call/jal指令来实现程序之间的执行流切换吧？

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：操作系统就是一个死函数，用户程序通过的时候，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，但是，操作系统本身出错了，整个计算机系统都将无法工作，所以，人们还是希望能把操作系统保护起来，尽量保证它不以正确工作。

在这个需求面前，用call/jal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，但无论用户程序是怎样写的，我们都不希望它可以随意地进行操作系统中的程序函数，我们将希望的，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。

等级保护的制度

为了阻止程序将执行流切换到操作系统的任意位置，硬件中通常会提供保护机制相关的功能，比如386中引入了保护模式(protected mode)和特权级(privilege level)的概念，而mips32处理器可以运行在内核模式和用户模式，riscv32则有机器模式(M-mode)、监控模式(S-mode)和用户模式(U-mode)，这些概念的思想都是类似的：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”。

以386为例，存在0, 1, 2, 3四个特权级，3特权级最高，3特权级最低，特权级0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取形成了一个环：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”的说法来描述一个特权级在的特权级。

```
+-----------------------------------------------+
|  +-------------------------------------------+ |
|  |                APPLICATIONS               | |
|  |  +-----------------------------------+    | |
|  |  |          CUSTOM EXTENSIONS        |    | |
|  |  |  +-------------------------------+ |    | |
|  |  |  |          SYSTEM SERVICES      | |    | |
|  |  |  |  +---------------------------+ | |    | |
|  |  |  |  |            OS             | | |    | |
|  |  |  |  +---------------------------+ | |    | |
|  |  |  +-------------------------------+ |    | |
|  |  |                                   |    | |
|  |  |                                   |    | |
|  |  |                                   |    | |
|  |  |                                   |    | |
|  +-------------------------------------------+ |
|                                               |
+-----------------------------------------------+
```

虽然386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386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入员办理取款业务，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，那么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
- 如果你打算自己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直接进行操作(data.DPL < current_process.CPL)，因此你那有可能会被抓起来

在计算机中存在着类似的情况：用户进程通常会把它自己拥有的数据(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代码和数据；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的代码和数据，这样，只要操作系统将其私有代码和数据放在ring 0中，那普通用户就永远没有办法访问它们了。

类似地，mips32的操作系统运行在内核模式，<2>式表示当前进程有权限访问目标数据，而另子riscv32，操作系统运行在监控者模式，用户进程运行在用户模式，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，都无法违反这一天规，硬件保护机制使得恶意程序永远无法全身而退，为构建计算机环境社会作出了巨大的贡献。

这些多发的功能：通常的，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很多的细节，ISA手册中一般都会专门有一章来描述保护机制，这就已经超出本保护机制并不是课单设说范围，但是ISS这里，我们并不打算在NEMU中加入保护机制，我们记所有用户进程都运行在最高特权级，虽然所有用户进程都有权限执行所有指令，不过由于NEMU是个模拟器，它并不会真的有自己编写的，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(例如比较数据)，如果发现了非法操作，就会抛出一个异常信号，CPU要做这些数据是不是真的是用户进程有权使用，并进行后续处理。

❗ 分离原则的用户

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保护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，最近两年计算机领域的，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，这两个全球级别的漏洞是现代计算领域里程碑，是因为它们打破了特权级的边界，恶意程序在完成特定的条件下可以公然高兴地读取位于内核空间的原理都是一脸懵逼的，在一个小西瓜的数字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。

Meltdown和Spectre利用这种漏洞，通过一系列的技巧，成功地将内核空间的内容读出来，甚至连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

那么和一般的C程序的区别在于，操作系统究竟有何特殊之处？答案是PA2之后，相信你就会略知一二了。

相就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，基本要连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

那么和一般的C程序的区别在于，操作系统究竟有何特殊之处？答案是PA2之后，相信你就会略知一二了。

相就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，基本要连编译方式也看不出什么特殊的地方，GNU/Linux也是这样：如果你阅读它的源代码，就会发现GNU/Linux只不过是个巨大的C程序而已。

那么和一般的C程序的区别在于，操作系统究竟有何特殊之处？答案是PA2之后，相信你就会略知一二了。

By Zhao Yu 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布 此页最终修订：2021-05-03 23:51:38

- 目 录 概
- 第一部分 早期内存管理
- 第1章 内存管理概述
- 1.1 内存管理的目标
- 1.2 地址空间
- 1.3 虚拟内存
- 1.4 页表
- 1.5 内存分配与回收
- 1.6 小结
- 第2章 下一代内存管理框架：MGLRU
- 2.1 设计哲学与动机
- 2.2 核心数据结构
- 2.3 页面年龄（Generation）的管理
- 2.4 页面回收（Eviction）流程
- 2.5 与传统 LRU 的对比
- 2.6 小结
- 第3章 内存压缩：zswap 与 zram
- 3.1 为什么需要内存压缩？
- 3.2 zswap：透明交换缓存
- 3.3 zram：内存盘交换设备
- 3.4 zswap 与 zram 的协同
- 3.5 小结
- 第4章 内存回收：kswapd 与 direct reclaim
- 4.1 为什么需要内存回收？
- 4.2 kswapd：后台异步回收
- 4.3 direct reclaim：前台同步回收
- 4.4 水位线（watermark）与回收触发
- 4.5 小结
- 第5章 内存分配：slab/slub/slub 分配器
- 5.1 为什么需要专门的内核对象分配器？
- 5.2 slab 的设计哲学
- 5.3 slub 的改进
- 5.4 kmem_cache 的生命周期
- 5.5 小结
- 第6章 内存碎片与 compaction
- 6.1 碎片的成因
- 6.2 compaction 的工作原理
- 6.3 与 kswapd 的协作
- 6.4 小结
- 第7章 内存管理的可观测性
- 7.1 /proc/meminfo 与 /proc/vmstat
- 7.2 /proc/zoneinfo
- 7.3 /proc/pagetypeinfo
- 7.4 /proc/slabinfo
- 7.5 小结
- 第二部分 现代内存管理
- 第8章 现代内存管理概览
- 8.1 从 2.6 到 6.x 的演进
- 8.2 现代内存管理的核心问题
- 8.3 小结
- 第9章 NUMA 内存管理
- 9.1 NUMA 的硬件基础
- 9.2 内核的 NUMA 抽象
- 9.3 NUMA 内存分配策略
- 9.4 NUMA 内存回收
- 9.5 NUMA 内存迁移
- 9.6 小结
- 第10章 透明大页（THP）
- 10.1 为什么需要大页？
- 10.2 THP 的设计
- 10.3 THP 的分配
- 10.4 THP 的回收
- 10.5 THP 的拆分
- 10.6 小结
- 第11章 内存热插拔
- 11.1 内存热插拔的应用场景
- 11.2 内核的内存热插拔子系统
- 11.3 memory_hotplug 的工作流程
- 11.4 memory_hotplug 的数据结构
- 11.5 小结
- 第12章 内存管理的调试与诊断
- 12.1 /proc/meminfo 的逐项解读
- 12.2 /proc/vmstat 的逐项解读
- 12.3 /proc/zoneinfo 的逐项解读
- 12.4 /proc/pagetypeinfo 的逐项解读
- 12.5 /proc/slabinfo 的逐项解读
- 12.6 小结
- 第三部分 内存管理子系统
- 第13章 mm/page_alloc.c：伙伴系统
- 13.1 文件概览
- 13.2 核心数据结构
- 13.3 分配路径
- 13.4 释放路径
- 13.5 水位线与 watermark
- 13.6 小结
- 第14章 mm/slab.c：slab 分配器
- 14.1 文件概览
- 14.2 核心数据结构
- 14.3 分配路径
- 14.4 释放路径
- 14.5 小结
- 第15章 mm/vmscan.c：页面回收
- 15.1 文件概览
- 15.2 核心数据结构
- 15.3 kswapd 的主循环
- 15.4 shrink_zone 与 shrink_slab
- 15.5 小结
- 第16章 mm/compaction.c：内存整理
- 16.1 文件概览
- 16.2 核心数据结构
- 16.3 compaction 的主循环
- 16.4 小结
- 第17章 mm/memory.c：页表与缺页处理
- 17.1 文件概览
- 17.2 核心数据结构
- 17.3 缺页处理
- 17.4 小结
- 第18章 mm/mmap.c：虚拟内存区域（VMA）
- 18.1 文件概览
- 18.2 核心数据结构
- 18.3 mmap 的实现
- 18.4 munmap 的实现
- 18.5 小结
- 第19章 mm/swap.c：页面换入换出
- 19.1 文件概览
- 19.2 核心数据结构
- 19.3 换出路径
- 19.4 换入路径
- 19.5 小结
- 第四部分 内存管理的扩展与未来
- 第20章 内存管理的扩展与未来
- 20.1 内存分层（memory tiering）
- 20.2 CXL 内存
- 20.3 内存安全
- 20.4 小结
- 第21章 内存管理的调试与诊断
- 21.1 /proc/meminfo 的逐项解读
- 21.2 /proc/vmstat 的逐项解读
- 21.3 /proc/zoneinfo 的逐项解读
- 21.4 /proc/pagetypeinfo 的逐项解读
- 21.5 /proc/slabinfo 的逐项解读
- 21.6 小结
- 附录A 内核源码阅读指南
- 附录B 内核编译与调试
- 附录C 术语表
- 附录D 参考资料

穿越时空的旅程

每个人都曾经幸运地活过，那个能带来生活的东西还能保持年轻的秘诀吗？在为了这个目标做准备的路上，我们发现最重要的事情，是在那条已经过去的路上，我们曾经遇见的那些人。这些人可能是朋友，陌生的打扫房间的工人，甚至是路上那条老狗的主人。我们曾经遇见的那些人，这个旅程中的每一个人都值得一个感谢。

这一切都要感谢我的妻子，我的孩子，我的父母，我的朋友，以及这个旅程中的所有人。我要感谢，我要感谢所有曾经在我的旅程中，给我帮助，关心我的人。

x86

x86架构，Intel 推出的处理器系列，世界第第一款拥有32位寄存器的CPU的处理器芯片，型号为80386，之前的系列是16位的处理器。它是Intel公司的第三代处理器，也是Intel公司最成功的一个设计产品。

x86

x86架构，Intel 推出的处理器系列，世界第第一款拥有32位寄存器的CPU的处理器芯片，型号为80386，之前的系列是16位的处理器。它是Intel公司的第三代处理器，也是Intel公司最成功的一个设计产品。

为了方便理解和了解此部分，我们将x86架构的一系列处理器，简称为：Intel 80386 Intel Pentium Intel Celeron Intel Xeon，中间省略的部分，我们将统一用 ...。

Intel 80386 Intel Pentium ... 的区别，主要是指一个CPU能进行多少个指令的并行处理，那意味着是多个指令的并行处理，那对CPU的内部结构，以及内部的一个指令的并行处理，有着巨大的区别。

31	23	15	8
EAX	EBX	ECX	EDX
UTCL DI SSL_DS	P	Dev's Cam	14
Dev's Cam		OPTOT IDL_D	18

对这有兴趣的读者，可以通过查阅英文版本，OPTOT和本章后面的一个英文单词，来了解这些，用户能用来记录数据的部分，因为这部分OPTOT的英文的记录，用户能用来记录的部分，因为这部分的英文的记录，用户能用来记录的部分，因为这部分的英文的记录，用户能用来记录的部分，因为这部分

事实上, 自陷只是其中一种异常类型. 有一种故障类异常, 它们返回的PC和触发异常的PC是同一个, 例如缺页异常, 在系统将故障排除后, 将会重新执行相同的指令进行重试, 因此异常返回的PC无需加4. 所以根据异常类型的不同, 有时候需要加4, 有时候则不需要加.

这时候, 我们就可以考虑这样的一个问题了: 决定要不要加4的, 是硬件还是软件呢? CISC和RISC的做法正好相反, CISC都交给硬件来做, 而RISC则交给软件来做. 思考一下, 这两种方案各有什么取舍? 你认为哪种更合理呢? 为什么?

代码最后会返回到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

你需要实现这一过程中的新指令. 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()` .

必答题(需要在实验报告中回答) - 理解穿越时空的旅程

从Nanos-lite调用 `yield()` 开始, 到从 `yield()` 返回的期间, 这一趟旅程具体经历了什么? 软(AM, Nanos-lite)硬(NEMU)件是如何相互协助来完成这趟旅程的? 你需要解释这一过程中的每一处细节, 包括涉及的每一行汇编代码/C代码的行为, 尤其是一些比较关键的指令/变量. 事实上, 上文的必答题"理解上下文结构体的前世今生"已经涵盖了这趟旅程中的一部分, 你可以把它的回答包含进来.

别被"每一行代码"吓到了, 这个过程也就大约50行代码, 要完全理解透彻并不是不可能的. 我们之所以设置这道必答题, 是为了强迫你理解清楚这个过程中的每一处细节. 这一理解是如此重要, 以至于如果你缺少它, 接下来你面对bug几乎是束手无策.

mips32延迟槽和异常

我们在PA2中提到, 标准的mips32处理器采用了分支延迟槽技术. 思考一下, 如果标准的mips32处理器在执行延迟槽指令的时候触发了异常, 从异常返回之后可能会造成什么问题? 该如何解决? 尝试RTFM对比你的解决方案.

温馨提示

PA3阶段1到此结束.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

- 专题
- 加载用户界面内容
- 操作系统的启动
- 系统调用
- 进程管理
- 内存管理
- 文件系统
- 设备管理

用户程序和系统调用

每个函数都有一个用户程序要可以调用的对应的函数和系统调用的入口了，现在我们来解决实现系统调用的第二个问题：如何实现用户程序

加载第一个用户程序

现在有了函数，也有了用户程序要加载到内存的函数，我们可以加载第一个程序了。

现在我们来看一下，当我们在命令行输入命令后，内核是怎样加载第一个程序的呢？

- 在命令行输入命令
- 命令行解析程序会调用一个函数
- 这个函数会调用一个系统调用
- 这个系统调用会调用一个函数
- 这个函数会调用一个程序

- 在命令行输入命令
- 命令行解析程序会调用一个函数
- 这个函数会调用一个系统调用
- 这个系统调用会调用一个函数
- 这个函数会调用一个程序

为了加载第一个函数，我们需要先加载一个用户程序名称或者命令的，用户程序应该放在哪里呢？这个程序应该放在哪里呢？我们可以在 /bin 目录下，放一个叫做 hello 的程序，这样当用户输入 /bin/hello 的时候，内核应该会加载 /bin/hello 这个程序。

```
$ ls/ls/bin
bash ls.elf cat.elf hello.elf
```

hello 程序的内容是什么呢？我们来看一下 /usr/src/kernel/init.s 这个文件：

```
hello:
    .ascii "Hello, world!\n"
    .ascii "Hello, world!\n"
    .ascii "Hello, world!\n"
```

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. 在PA4中，我们会对这一系统调用进行修改，实现真正的内存分配.

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. 于是有了批处理(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. 缓冲区是批处理技术的核心，libc中的 `fread()` 和 `fwrite()` 正是通过缓冲区来将数据累积起来，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.

printf和换行

我们在PA1的时候提示过大家，使用 `printf()` 调试时需要添加 `\n`，现在终于可以解释为什么了: `fwrite()` 的实现中有缓冲区，`printf()` 打印的字符不一定会马上通过 `write()` 系统调用输出，但遇到 `\n` 时可以强行将缓冲区中的内容进行输出. 有兴趣的同学可以阅读 `navy-apps/libs/libc/src/stdio/wbuf.c`，这个文件实现了缓冲区的功能.

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.

必答题(需要在实验报告中回答) - hello程序是什么，它从而何来，要到哪里去

到此为止，PA中的所有组件已经全部亮相，整个计算机系统也开始趋于完整. 你也已经在这个自己创造的计算机系统上跑起了hello这个第一个还说得过去的用户程序 (dummy是给大家热身用的，不算)，好消息是，我们已经距离运行仙剑奇侠传不远了(下一个阶段就是啦).

不过按照PA的传统，光是跑起来还是不够的，你还要明白它究竟怎么跑起来才行. 于是来回答这道必答题吧:

> 我们知道 `navy-apps/tests/hello/hello.c` 只是一个C源文件，它会被编译链接成一个ELF文件. 那么，hello程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? hello程序在不断地打印字符串，每一个字符又是经历了什么才会最终出现在终端上?

上面一口气问了很多问题，我们想说的是，这其中蕴含着非常多需要你理解的细节. 我们希望你能够认真整理其中涉及的每一行代码，然后用自己的语言融会贯通地把这个过程的理解描述清楚，而不是机械地分点回答这几个问题.

同样地，上一阶段的必答题"理解穿越时空的旅程"也已经涵盖了一部分内容，你可以把它的回答包含进来，但需要描述清楚有差异的地方. 另外，C库中 `printf()` 到 `write()` 的过程比较繁琐，而且也不属于PA的主线内容，这一部分不必展开回答. 而且你也已经在PA2中实现了自己的 `printf()` 了，相信你也不难理解字符串格式化的过程. 如果你对Newlib的实现感兴趣，你也可以RTFSC.

总之，扣除C库中 `printf()` 到 `write()` 转换的部分，剩下的代码就是你应该理解透彻的了. 于是，努力去理解每一行代码吧!

温馨提示

PA3阶段2到此结束.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

简易文件系统

最后一个实验的内容是实现一个简易文件系统。实现文件系统的内容主要包括实现文件系统底层的数据结构，实现文件系统提供的基础功能，实现文件系统提供给用户程序的syscall接口。这里，我们实现的文件系统是一个单级文件系统，也就是说，文件系统中只有一个根目录，所有的文件都放在根目录下。这样，用户程序在调用文件相关的syscall时，用户程序传递给内核的参数，它只需要指定文件的文件名，而不需要考虑目录结构。这大大简化了文件系统的设计。这里，我们将实现文件系统的底层数据结构，实现文件系统提供给内核的基础功能，以及实现文件系统提供给用户程序的syscall接口。

文件系统的底层数据结构，主要包括：磁盘数据结构布局、磁盘数据结构的内存表示、文件描述符的内存表示。

文件系统提供给内核的基础功能，主要包括：

- 文件的创建、删除、读取、写入、关闭
- 写入文件时，文件长度的扩展
- 文件的权限管理
- 目录管理

文件系统提供给用户程序的syscall接口，主要包括：

- 文件的创建、删除、打开、关闭
- 文件的读取、写入、定位
- 目录的创建、删除、遍历
- 文件的权限管理

我们可以把文件系统的内容看成是：

```
/
├── file1
├── file2
└── file3
```

为了方便调试和学习文件系统的实现，我们还需要一个用户工具来对文件系统进行操作。

```
$ ls
$ echo hello > file1
$ ls
file1
$ cat file1
hello
$ echo world > file2
$ ls
file1
file2
$ cat file2
world
$ rm file1
$ ls
file2
$ cat file1
cannot open file1
$
```

这个工具需要提供文件系统的读、写、删除、列出目录内容、查看文件内容等功能。这样，我们可以在文件系统实现的过程中，通过这个工具来对文件系统进行测试。这个工具，一般称为文件系统镜像工具。这个工具，一般称为mkfs。这个工具，一般称为fsck。

我们可以把文件系统的内容看成一个数组：

```
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
[ 文件系统 ]
```

为了方便磁盘和内存之间数据的交换，我们可以把文件系统的内容看成一个数组：

```
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
[ 引导扇区 ]
```

为了方便磁盘和内存之间数据的交换，我们可以把文件系统的内容看成一个数组：

```
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
[ 超级块 ]
```

为了方便磁盘和内存之间数据的交换，我们可以把文件系统的内容看成一个数组：

```
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
[ 文件描述符 ]
```

为了方便磁盘和内存之间数据的交换，我们可以把文件系统的内容看成一个数组：

```
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
[ 目录项 ]
```

为了方便磁盘和内存之间数据的交换，我们可以把文件系统的内容看成一个数组：

```
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
[ 数据块 ]
```

一切皆文件

AMPHROS的文件系统设计思路是：一切皆文件。

- 内存是文件管理器的文件，内存管理器提供的基础功能，实现文件系统的内存表示，以及提供给用户程序的syscall接口。
- 磁盘是文件管理器的文件，磁盘管理器提供的基础功能，实现文件系统的磁盘数据结构布局，以及提供给内存管理器的基础功能。
- 控制台是文件管理器的文件，控制台管理器提供的基础功能，实现文件系统的内存表示，以及提供给内存管理器的基础功能。
- 管道是文件管理器的文件，管道管理器提供的基础功能，实现文件系统的内存表示，以及提供给内存管理器的基础功能。

这样，文件管理器的基础功能，主要包括：

- 文件的创建、删除、打开、关闭、读取、写入、定位
- 目录的创建、删除、遍历
- 文件的权限管理

这样，文件管理器提供给用户程序的syscall接口，主要包括：

- 文件的创建、删除、打开、关闭、读取、写入、定位
- 目录的创建、删除、遍历
- 文件的权限管理

这样，文件管理器提供给内存管理器的基础功能，主要包括：

- 内存管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 磁盘管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 控制台管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 管道管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位

这样，文件管理器提供给内存管理器的syscall接口，主要包括：

- 内存管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 磁盘管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 控制台管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位
- 管道管理器的文件，文件的创建、删除、打开、关闭、读取、写入、定位

这样，文件管理器的基础功能，主要包括：

- 文件的创建、删除、打开、关闭、读取、写入、定位
- 目录的创建、删除、遍历
- 文件的权限管理

这样，文件管理器的syscall接口，主要包括：

- 文件的创建、删除、打开、关闭、读取、写入、定位
- 目录的创建、删除、遍历
- 文件的权限管理

这样，文件管理器的内部，主要包括：

- 磁盘数据结构布局
- 磁盘数据结构的内存表示
- 文件描述符的内存表示

这样，文件管理器的外部，主要包括：

- 内存管理器
- 磁盘管理器
- 控制台管理器
- 管道管理器

磁盘文件系统

为了实现一切皆文件的设计思路，我们需要实现磁盘文件系统。磁盘文件系统，主要包括：

- 磁盘数据结构布局
- 磁盘数据结构的内存表示

一切皆文件，内容是要存储在磁盘上的。磁盘数据结构布局，主要包括：

- 引导扇区
- 超级块
- 文件描述符
- 目录项
- 数据块

这样，磁盘文件系统的内容，主要包括：

- 引导扇区
- 超级块
- 文件描述符
- 目录项
- 数据块

这样，磁盘文件系统的内部，主要包括：

- 磁盘数据结构布局
- 磁盘数据结构的内存表示

这样，磁盘文件系统的外部，主要包括：

- 内存管理器
- 磁盘管理器
- 控制台管理器
- 管道管理器

这样，磁盘文件系统的内部，主要包括：

- 磁盘数据结构布局
- 磁盘数据结构的内存表示

这样，磁盘文件系统的外部，主要包括：

- 内存管理器
- 磁盘管理器
- 控制台管理器
- 管道管理器

磁盘文件系统的内部，主要包括：

- 磁盘数据结构布局
- 磁盘数据结构的内存表示

目录
概览
概览
Linux 内核概览
Linux 内核概览
Linux 内核

APU中，即在这个模式下，我们可以分配更多的内存给图形处理单元（GPU）。事实上，一个可能只需要256MB内存的应用程序，现在可以分配512MB甚至1GB内存给GPU。这意味着更大的纹理、更复杂的模型和更高的帧率。

正在加载 | 即将

在Steam中安装了游戏后，然后在Steam中点击一些从应用程序（2.2Alternative可以在Steamapps目录中找到）。这会启动一个新的Steam应用程序，并开始下载游戏。这个过程通常涉及从Steam的内容服务器下载大文件

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: 40小时

阶段性安排: 提交截止日期见这里

- task PA4.1: 实现基本的多道程序系统, 支持带参数的仙剑奇侠传与Hello内核线程的分时运行
- task PA4.2: 实现支持虚存管理的多道程序系统
- 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

提交说明: 见这里

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

- 每周
- 关键词
- 文章
- 基本概念
- 内存管理
- 内存结构
- 内存分配与回收
- 垃圾回收算法
- 内存泄漏
- 内存溢出
- 性能优化
- 实战案例
- 面试题
- 工具
- 学习资源
- 个人思考

第3篇 Nginx 模块开发

大家好，我是 Nginx 模块开发系列的第3篇！

这个系列的目标是：从零开始，带你掌握 Nginx 模块开发的核心技能，最终能够独立编写生产级别的模块。

前两篇我们搭建了开发环境，并成功编译了第一个模块 `ngx_hello_world`，并学习了 `ngx_http_request_t` 结构体。

这个模块虽然能运行，但它只是一个简单的 Hello World 程序，没有任何实用价值。

今天，我们将从一个真实的生产问题出发，一步步构建一个功能完整的流量统计模块，并在过程中，深入剖析 Nginx 模块开发的核心概念与设计哲学。

1. 从生产问题出发：为什么需要流量统计？

在我们动手编写代码之前，我们先来聊聊，为什么需要为 Nginx 编写一个流量统计模块？

假设您是一名运维工程师，您的公司有一个重要的 API 服务，部署在 Nginx 之后，为多个客户端提供服务。

现在，您的老板要求您，为每个客户端（通过 IP 地址区分）统计出每天的请求数和流量，以便进行计费和分析。

这个需求看似简单，但在生产环境中，会面临一系列挑战：

- 性能开销：统计逻辑会不会拖慢 Nginx 的处理速度？
- 内存管理：如何高效地存储和管理成千上万个客户端的统计数据？
- 数据持久化：Nginx 重启后，统计数据会丢失吗？
- 多进程/多线程：在 Nginx 的多进程模型下，如何保证数据的一致性？

这些问题，正是生产级模块开发的核心挑战。

2. 核心概念：Nginx 的内存与数据结构

在开始编写代码之前，我们需要先了解 Nginx 内部，一些最基本的内存管理和数据结构。

2.1. 内存池 (Memory Pool)

Nginx 采用了一种高效的内存管理机制，叫做内存池。

您可以把它想象成一个大的、预先分配好的内存块。当 Nginx 需要为某个请求（Request）创建一个对象（比如 `ngx_http_request_t`）时，它会从这个内存池中，快速地分配一小块内存。

当这个请求处理完毕，Nginx 会一次性地，将整个内存池中，所有分配出去的内存，全部回收。

这种机制，带来了巨大的性能优势：

- 分配速度：内存分配，只是简单的指针移动，速度极快。
- 内存碎片：大幅减少内存碎片的产生。
- 生命周期：内存的生命周期，与请求的生命周期，完全一致，易于管理。

2.2. 核心数据结构：`ngx_http_request_t`

我们在上一篇，已经接触过这个结构体了。

`ngx_http_request_t` 是 Nginx 为每一个 HTTP 请求，所创建的，最核心的数据结构。

它包含了，关于这个请求的，所有信息，比如：

- 客户端信息：客户端的 IP 地址、端口。
- 请求信息：请求的方法（GET, POST）、URI、Header、Body。
- 响应信息：将要发送给客户端的，响应内容。
- 状态：请求的当前处理阶段、是否出错、是否完成。

这个结构体，是我们，在模块中，与 Nginx 核心，进行交互，的桥梁。

3. 设计与实现：流量统计模块

3.1. 目标与设计

我们的目标，是为每个客户端（IP 地址），统计其，每天的，请求数和流量。

为了，保证性能，我们将，把统计数据，存在，内存中，并，定期，将，数据，持久化，到，磁盘。

3.2. 数据结构设计

我们需要，为，每个客户端，设计一个，数据结构，来，存储，统计数据。

这个，数据结构，本身，也，需要，在，内存，中，被，管理，和，分配。

我们，可以，在，模块，的，配置，文件，中，定义，一个，新，的，指令，来，允许，用户，开启，这个，统计，功能，并，指定，数据，文件，的，路径。

```
# 在 nginx.conf 中
traffic_stat on;
traffic_stat_file /var/log/nginx/traffic.stat;
```

3.3. 模块配置文件

`traffic_stat.conf`

```
# 是否开启流量统计
traffic_stat on;

# 统计数据文件的路径
traffic_stat_file /var/log/nginx/traffic.stat;
```

3.4. 核心代码实现

`src/ngx_http_traffic_stat_module.c`

```
/*
 * 定义模块的配置结构体
 */
typedef struct {
    ngx_flag_t  enable;
    ngx_str_t   file;
} ngx_http_traffic_stat_conf_t;

/*
 * 定义模块的上下文结构体
 */
static ngx_http_module_t  ngx_http_traffic_stat_module_ctx = {
    NULL,                           /* preconfiguration */
    ngx_http_traffic_stat_init,     /* postconfiguration */

    NULL,                           /* create main configuration */
    NULL,                           /* init main configuration */

    NULL,                           /* create server configuration */
    NULL,                           /* merge server configuration */

    ngx_http_traffic_stat_create_loc_conf,  /* create location configuration */
    ngx_http_traffic_stat_merge_loc_conf    /* merge location configuration */
};

/*
 * 定义模块的配置指令
 */
static ngx_command_t  ngx_http_traffic_stat_commands[] = {
    { ngx_string("traffic_stat"),
      NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_traffic_stat_conf_t, enable),
      NULL },

    { ngx_string("traffic_stat_file"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_str_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_traffic_stat_conf_t, file),
      NULL },

    ngx_null_command
};

/*
 * 定义模块的主体
 */
ngx_module_t  ngx_http_traffic_stat_module = {
    NGX_MODULE_V1,
    &ngx_http_traffic_stat_module_ctx,
    ngx_http_traffic_stat_commands,
    NGX_HTTP_MODULE,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NGX_MODULE_V1_PADDING
};
```

3.5. 编译与测试

现在，我们，可以，来，编译，这个，新，的，模块，了，

```
./configure --add-module=/path/to/ngx_http_traffic_stat_module
make
```

编译，成功，后，我们，可以，在，配置，文件，中，使用，这个，新，的，指令，了，

```
# 在 nginx.conf 中
traffic_stat on;
traffic_stat_file /var/log/nginx/traffic.stat;
```

4. 总结与展望

今天，我们，从，一个，真实，的，生产，问题，出发，设计，并，实现，了，一个，简单，的，Nginx，流量，统计，模块，

这个，模块，虽然，还，很，简单，

但，它，已经，具备，了，一个，生产，级，模块，的，雏形，

我们，学习，了，内存，池，的，重要，性，

以，及，如，何，为，每，个，请，求，创，建，数，据，结，构，

在，这，个，基，础，上，我，们，将，会，在，后，续，的，文，章，中，

为，这，个，模，块，添，加，更，多，的，功，能，

例，如，

- 支持，IP，白，名，单，
- 支持，更，多，的，统，计，维，度，
- 将，数，据，定，期，地，持，久，化，到，磁，盘，
- 提，供，一，个，简，单，的，管，理，接，口，

敬，请，期，待，

5. 进阶：内存管理与性能优化

5.1. 内存池 (Memory Pool) 的使用

我们，在，上，一，篇，的，基，础，上，

我们，将，会，在，模，块，的，配，置，文，件，中，

为，这，个，新，的，指，令，

进，行，

```
# 在 nginx.conf 中
traffic_stat_mem_pool_size 10m;
```

这，个，指，令，

允，许，我，们，为，这，个，模，块，

单，独，地，

分，配，一，个，大，小，为，10MB，的，内，存，池，

这，样，

即，使，

我，们，的，模，块，

发，生，了，内，存，泄，漏，

也，不，会，影，响，到，

Nginx，的，其，他，部，分，

5.2. 共享内存 (Shared Memory) 与多进程

现在，

我，们，的，模，块，

工，作，

是，

完，全，

独，立，的，

这，

意，味，着，

在，

多，进，程，

模，式，

下，

每，个，

工，作，进，程，

都，会，

有，一，份，

自，己，的，

统，计，数，据，

这，

会，导，致，

数，据，

不，一，致，

为，了，解，决，这，个，问，题，

我，们，需，要，

引，入，

共，享，内，存，

这，

是，

一，种，

可，以，让，

多，个，

进，程，

访，问，

同，一，块，

内，存，区，域，

的，

机，制，

5.3. 实现共享内存

首先，

我，们，需，要，

在，

`ngx_http_traffic_stat_module.h`

中，

定，义，

一，个，

共，享，

内，存，

区，域，

```
/*
 * 定义共享内存区域的结构体
 */
typedef struct {
    ngx_shm_zone_t  *shm_zone;
} ngx_http_traffic_stat_main_conf_t;
```

然，后，

我，们，需，要，

在，

`ngx_http_traffic_stat_init_main_conf`

函，数，

中，

来，

初，始，化，

这，

个，

共，享，

内，存，

区，域，

```
static ngx_int_t
ngx_http_traffic_stat_init_main_conf(ngx_conf_t *cf, void *conf)
{
    ngx_http_traffic_stat_main_conf_t  *mcf = conf;

    ngx_int_t  rc;

    rc = ngx_shm_zone_init(cf, &mcf->shm_zone, ngx_shm_zone_size, ngx_shm_zone_name);

    if (rc != NGX_OK) {
        return NGX_ERROR;
    }

    return NGX_OK;
}
```

5.4. 数据结构与内存分配

现在，

我，们，需，要，

为，

`ngx_http_request_t`

创，建，

一，个，

新，的，

数，据，结，构，

```
/*
 * 定义每个客户端的统计数据结构
 */
typ
```

第4篇 Nginx 模块开发

大家好，我是 Nginx 模块开发系列的第4篇！

这个系列的目标是：从零开始，带你掌握 Nginx 模块开发的核心技能，最终能够独立编写生产级别的模块。

前三篇，我们搭建了开发环境，成功编译了第一个模块，并学习了 `ngx_http_request_t` 结构体。

我们，还，为，这个，模，块，添，加，了，一，个，简，单，的，流，量，统，计，功，能，

今天，我们，将，从，一，个，真，实，的，生，产，问，题，出，发，

为，这，个，模，块，

添，加，

一，个，

简，单，的，

IP，白，名，单，

功，能，

1. 从生产问题出发：为什么需要 IP 白名单？

在，我，们，动，手，编，写，代，码，之，前，

我，们，先，来，聊，聊，

为，什，么，需，要，

IP，白，名，单，

假，设，您，是，一，名，运，维，工，程，师，

您，的，公，司，有，一，个，重，要，的，内，部，管，理，后，台，

现，在，您，的，老，板，要，求，您，

为，

IP，地，址，在，

192.168.1.100

至，

192.168.1.200

这，

一，段，

IP，地，址，

提，供，

访，问，

这，个，

后，台，

这，个，

需，求，

看，似，

简，单，

但，

在，

生，产，

环，境，

中，

会，

面，临，

一，系，列，

挑，战，

- 安全性：如何，确，保，只，有，这，些，IP，地，址，能，够，访，问，这，个，后，台，
- 灵活性：如何，方，便，地，修，改，这，个，IP，白，名，单，
- 性能：如何，高，效，地，检，查，一，个，请，求，的，IP，地，址，是，否，在，这，个，白，名，单，

这，些，问，题，

2. 核心概念：Nginx 的配置文件与指令

在，开，始，编，写，代，码，之，前，

我，们，需，要，

了，解，一，些，

Nginx，的，配，置，文，件，

和，指，令，

2.1. 配置文件

Nginx，的，配，置，文，件，

通，常，是，

`nginx.conf`

文，件，

2.2. 指令

Nginx，的，指，令，

通，常，是，

`nginx.conf`

文，件，

2.3. 模块配置指令

我，们，

`ngx_http_traffic_stat_module.h`

文，件，

```
/*
 * 定义模块的配置指令
 */
static ngx_command_t  ngx_http_traffic_stat_commands[] = {
    { ngx_string("traffic_stat_ip_whitelist"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_http_traffic_stat_ip_whitelist,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_traffic_stat_loc_conf_t, ip_whitelist),
      NULL },

    ngx_null_command
};
```

3. 设计与实现：IP 白名单模块

3.1. 目标与设计

我，们，的，目，标，

是，

为，

`ngx_http_request_t`

这，个，

请，求，

创，建，

一，个，

新，的，

数，据，结，构，

3.2. 数据结构设计

我，们，需，要，

为，

`ngx_http_request_t`

创，建，

一，个，

新，的，

数，据，结，构，

```
/*
 * 定义每个请求的 IP 白名单数据结构
 */
typ
```

第5篇 Nginx 模块开发

大家好，我是 Nginx 模块开发系列的第5篇！

这个系列的目标是：从零开始，带你掌握 Nginx 模块开发的核心技能，最终能够独立编写生产级别的模块。

前四篇，我们搭建了开发环境，成功编译了第一个模块，并学习了 `ngx_http_request_t` 结构体。

我们，还，为，这个，模，块，添，加，了，一，个，简，单，的，流，量，统，计，功，能，

今天，我们，将，从，一，个，真，实，的，生，产，问，题，出，发，

为，这，个，模，块，

添，加，

一，个，

简，单，的，

IP，白，名，单，

功，能，

1. 从生产问题出发：为什么需要 IP 白名单？

在，我，们，动，手，编，写，代，码，之，前，

我，们，先，来，聊，聊，

为，什，么，需，要，

IP，白，名，单，

假，设，您，是，一，名，运，维，工，程，师，

您，的，公，司，有，一，个，重，要，的，内，部，管，理，后，台，

现，在，您，的，老，板，要，求，您，

为，

IP，地，址，在，

192.168.1.100

至，

192.168.1.200

这，

一，段，

IP，地，址，

提，供，

访，问，

这，个，

后，台，

这，个，

需，求，

看，似，

简，单，

但，

在，

生，产，

环，境，

中，

会，

面，临，

一，系，列，

挑，战，

- 安全性：如何，确，保，只，有，这，些，IP，地，址，能，够，访，问，这，个，后，台，
- 灵活性：如何，方，便，地，修，改，这，个，IP，白，名，单，
- 性能：如何，高，效，地，检，查，一，个，请，求，的，IP，地，址，是，否，在，这，个，白，名，单，

这，些，问，题，

2. 核心概念：Nginx 的配置文件与指令

在，开，始，编，写，代，码，之，前，

我，们，需，要，

了，解，一，些，

Nginx，的，配，置，文，件，

和，指，令，

2.1. 配置文件

Nginx，的，配，置，文，件，

通，常，是，

`nginx.conf`

文，件，

2.2. 指令

Nginx，的，指，令，

通，常，是，

`nginx.conf`

文，件，

2.3. 模块配置指令

我，们，

`ngx_http_traffic_stat_module.h`

文，件，

```
/*
 * 定义模块的配置指令
 */
static ngx_command_t  ngx_http_traffic_stat_commands[] = {
    { ngx_string("traffic_stat_ip_whitelist"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_http_traffic_stat_ip_whitelist,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_traffic_stat_loc_conf_t, ip_whitelist),
      NULL },

    ngx_null_command
};
```

3. 设计与实现：IP 白名单模块

3.1. 目标与设计

我，们，的，目，标，

是，

为，

`ngx_http_request_t`

这，个，

请，求，

创，建，

一，个，

新，的，

数，据，结，构，

3.2. 数据结构设计

我，们，需，要，

为，

`ngx_http_request_t`

创，建，

一，个，

新，的，

数，据，结，构，

```
/*
 * 定义每个请求的 IP 白名单数据结构
 */
typ
```

第6篇 Nginx 模块开发

大家好，我是 Nginx 模块开发系列的第6篇！

这个系列的目标是：从零开始，带你掌握 Nginx 模块开发的核心技能，最终能够独立编写生产级别的模块。

前五篇，我们搭建了开发环境，成功编译了第一个模块，并学习了 `ngx_http_request_t` 结构体。

我们，还，为，这个，模，块，添，加，了，一，个，简，单，的，流，量，统，计，功，能，

今天，我们，将，从，一，个，真，实，的，生，产，问，题，出，发，

为，这，个，模，块，

添，加，

一，个，

简，单，的，

IP，白，名，单，

功，能，

1. 从生产问题出发：为什么需要 IP 白名单？

在，我，们，动，手，编，写，代，码，之，前，

我，们，先，来，聊，聊，

为，什，么，需，要，

IP，白，名，单，

假，设，您，是，一，名，运，维，工，程，师，

您，的，公，司，有，一，个，重，要，的，内，部，管，理，后，台，

现，在，您，的，老，板，要，求，您，

为，

IP，地，址，在，

192.168.1.100

至，

192.168.1.200

这，

一，段，

IP，地，址，

提，供，

访，问，

这，个，

后，台，

这，个，

需，求，

看，似，

简，单，

但，

在，

生，产，

环，境，

中，

会，

面，临，

一，系，列，

挑，战，

- 安全性：如何，确，保，只，有，这，些，IP，地，址，能，够，访，问，这，个，后，台，
- 灵活性：如何，方，便，地，修，改，这，个，IP，白，名，单，
- 性能：如何，高，效，地，检，查，一，个，请，求，的，IP，地，址，是，否，在，这，个，白，名，单，

这，些，问，题，

2. 核心概念：Nginx 的配置文件与指令

在，开，始，编，写，代，码，之，前，

我，们，需，要，

了，解，一，些，

Nginx，的，配，置，文，件，

和，指，令，

2.1. 配置文件

Nginx，的，配，置，文，件，

通，常，是，

`nginx.conf`

文，件，

2.2. 指令

Nginx，的，指，令，

通，常，是，

`nginx.conf`

文，件，

2.3. 模块配置指令

我，们，

`ngx_http_traffic_stat_module.h`

文，件，

```
/*
 * 定义模块的配置指令
 */
static ngx_command_t  ngx_http_traffic_stat_commands[] = {
    { ngx_string("traffic_stat_ip_whitelist"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_http_traffic_stat_ip_whitelist,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_traffic_stat_loc_conf_t, ip_whitelist),
      NULL },

    ngx_null_command
};
```

3. 设计与实现：IP 白名单模块

3.1. 目标与设计

我，们，的，目，标，

是，

为，

`ngx_http_request_t`

这，个，

请，求，

创，建，

一，个，

新，的，

数，据，结，构，

3.2. 数据结构设计

我，们，需，要，

为，

`ngx_http_request_t`

创，建，

一，个，

新，的，

数，据，结，构，

```
/*
 * 定义每个请求的 IP 白名单数据结构
 */
typ
```

```
+-----------------+ <---- argv
|                 |
+-----------------+
|                 |
|     ......      |
|                 |
+-----------------+
|    argv[1]      | ---------+ |
+-----------------+          |
|                 |
+-----------------+
|    argv[0]      | ----------+
+-----------------+
|                 |
|      argc       |
+-----------------+ <---- cp->GPRx
|                 |
```

上图把这些参数分成两部分，一部分是字符串区域(string area)，另一部分是 `argv/envp` 这两个字符串指针数组，数组中的每一个元素是一个字符串指针，而这些字符串指针都会指向字符串区域中的某个字符串。此外，上图中的 `Unspecified` 表示一段任意长度(也可为0)的间隔，字符串区域中各个字符串的顺序也不作要求，只要用户进程可以通过 `argv/envp` 访问到正确的字符串即可，这些参数的放置格式与ABI手册中的描述非常类似，你也可以参考ICS课本第七章的某个图。

> 阅读ABI手册，理解计算机系统

事实上，ABI手册是ISA、OS、编译器，运行时环境，C语言和用户进程的桥梁，非常值得大家去阅读. ICS课本上那些让你摸不着头脑的约定，大部分也是出自ABI手册. Linux上遵守的ABI是System V ABI，它又分为两部分，一部分是和处理器无关的generic ABI(gABI)，例如ELF格式，动态连接，文件系统的顺序也不作要求，只要用户进程可以通过 `argv/envp` 访问到正确的字符串即可. 这些参数的放置格式与ABI手册中的大致的了解. 如果你愿意深入推敲一下"为什么这样的定"，那就是真正的"深入理解计算机系统了".

根据这一约定，你需要修改Navy中 `_start` 的代码，把 `argc` 的地址作为参数传递给 `call_main()` . 然后修改 `call_main()` 的代码，让它解析出真正的 `argc/argv/envp` ，并调用 `main()` :

```
void call_main(uintptr_t *args) {
  argc = ???
  argv = ???
  envp = ???
  environ = envp;
  exit(main(argc, argv, envp));
  assert(0);
}
```

这样以后，用户进程就可以接收到属于它的参数了.

> 给用户进程传递参数

这个任务的本质是一个指针相关的编程练习，不过你需要注意编写可移植的代码，因为 `call_main()` 是被各种ISA所共享的，然后修改仙剑奇侠传的少量代码，如果它接收到一个 `--skip` 参数，就跳过片头商标动画的播放，否则不跳过. 实现这个功能将有利于加速仙剑奇侠传的测试. 商标动画的播放从代码逻辑上距离 `main()` 函数并不远，于是就交给你来RTFSC吧.

不过为了给用户进程传递参数，你还需要修改 `context_uload()` 的原型:

```
void context_uload(PCB *pcb, const char *filename, char *const argv[], char *const env
```

这样你就可以在 `init_proc()` 中直接给出用户进程的参数来测试了: 在创建仙剑奇侠传用户进程的时候给出 `--skip` 参数，你需要观察到仙剑奇侠传确实跳过了商标动画. 目前我们的测试程序中不会用到环境变量，所以不必传递真实的环境变量字符串. 至于实参应该写什么，这又是一个指针相关的问题，就交给你来解决吧.

让操作系统为每一个用户进程手动设定参数是一件不现实的事情，因为用户进程的参数还是应该由用户来指定的. 于是最好解决方案是每一个方法能把用户指定的参数直传操作系统，让操作系统来把指定的参数放到新进程的用户栈里面. 这个方法当然就是系统调用 `SYS_execve` 啦. 如果你去看 `man` ，你会发现它的原型是

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

> 为什么少了一个const?

在 `main()` 函数中， `argv` 和 `envp` 的类型是 `char * []` ，而在 `execve()` 函数中, 它们的类型则是 `char *const []` . 你知道为什么吗?

为了实现带参数的 `SYS_execve` ，我们可以在 `sys_execve()` 中直接调用 `context_uload()` . 但我们还需要考虑如下的一些细节, 为了方便描述, 我们假设用户进程A将要通过 `SYS_execve` 来执行另一个新程序B.

- 如何在A的执行流中创建用户进程B?
- 如何结束A的执行流?

为了回答第一个问题, 我们需要回顾创建用户进程B需要进行哪些操作. 首先是在PCB的内核栈中创建B的上下文结构, 这个过程是安全的, 因为当前进程的内核栈是空的. 接下来就是要在用户栈中放置用户进程B的参数. 但这会涉及到一个新的问题: 我们是否还能复用位于 `heap.end` 附近的同一个用户栈?

为了探究这个问题, 我们需要了解当Nanos-lite尝试通过 `SYS_execve` 加载B时, A的用户栈里面已经有什么内容. 我们可以从栈底(`heap.end`)到栈顶(栈指针 `sp` 当前的位置)列出用户栈中的内容:

- Nanos-lite之前为A传递的用户进程参数 (`argc/argv/envp`)
- A从 `_start` 开始进行函数调用的栈帧, 这个栈帧会一直生长, 直到调用了libos中的 `execve()`
- CTE保存的上下文结构, 这是由于A在 `execve()` 中执行了系统调用自陷指令导致的
- Nanos-lite从 `__am_irq_handle()` 开始进行函数调用的栈帧, 这个栈帧会一直生长, 直到调用了 `SYS_execve` 的系统调用处理函数

通过上述分析, 我们得出一个重要的结论: 在加载B时, Nanos-lite使用的是A的用户栈! 这意味着在A的执行流结束之前, A的用户栈是不能被破坏的. 因此 `heap.end` 附近的用户栈是不能被B复用的, 我们应该申请一段新的内存作为B的用户栈, 来让Nanos-lite把B的参数放置到这个新分配的用户栈里面.

为了实现这一点, 我们可以让 `context_uload()` 统一通过调用 `new_page()` 函数来获得用户栈的内存空间. `new_page()` 函数在 `nanos-lite/src/mm.c` 中定义, 它会通过一个 `pf` 指针来管理堆区, 用于分配一段大小为 `nr_page * 4KB` 的连续内存区域, 并返回这段区域的首地址. 我们让 `context_uload()` 通过 `new_page()` 来分配32KB的内存作为用户栈, 这对PA中的用户程序来说已经足够使用了. 此外为了简化, 我们在PA中无需实现 `free_page()` .

> 操作系统的内存管理

我们知道kilib中的 `malloc()` 函数也可以进行堆区的管理, 使得AM应用可以方便地进行动态内存申请. 但操作系统作为一个特殊的AM应用, 很多时候对动态内存申请都有更严格的要求, 例如申请一段起始地址是4KB整数倍的内存区域. `malloc()` 通常不能满足这样的要求. 因此操作系统一般都会自己来管理堆区, 而不会调用kilib中的 `malloc()` . 在操作系统中管理堆区是MM(Memory Manager)模块的工作, 我们会在后续内容中进一步介绍它.

最后, 为了结束A的执行流, 我们可以在创建B的上下文之后, 通过 `switch_boot_pcb()` 修改当前的 `current` 指针, 然后调用 `yield()` 来强制触发进程调度. 这样以后, A的执行流就不会再被调度, 等到下一次调度的时候, 就可以恢复并执行B了.

> 实现带参数的execve()

根据上述讲义内容, 实现带参数的 `execve()` , 有一些细节我们并没有完全给出, 例如调用 `context_uload()` 的 `pcb` 参数应该传入什么内容, 这个问题就交给你来思考吧!

实现后, 运行以下程序:

- 测试程序 `navy-apps/tests/exec-test` , 它会以参数迷墙的方式不断地执行自身. 不过由于我们没有实现堆区内存的回收, `exec-test` 在运行一段时间之后, `new_page()` 就会把 `0x300000` / `0x830000` 附近的内存分配出去, 导致用户进程的代码段被覆盖. 目前我们无法修复这一问题, 你只需要看到 `exec-test` 可以正确运行一段时间即可.
- MENU开机菜单
- 完善NTerm的内建Shell, 使得它可以解析输入的参数, 并传递给启动的程序. 例如可以在NTerm中键入 `pal --skip` 来运行仙剑奇侠传并跳过商标动画.

运行Busybox

我们已经成功运行了NTerm, 但却没多少Shell工具可以运行. Busybox正是用来解决这个问题的, 它是一个精简版Shell工具的集合, 包含了大部分常用命令的常用功能. 哦, 你平时在Linux中使用命令行的经历, 很快就可以在你自己构建的计算机系统里面呈现了!

Navy的根架代码已经准备了Busybox的编译脚本, 首次编译Busybox时脚本会自动克隆项目, 并使用根架代码提供的配置文件. Busybox中包含很多小工具, 你可以通过

```
make menuconfig
```

来打开一个配置菜单来查看它们(但不要保存对配置的修改). 根架代码提供的配置文件默认只选中了很少的工具, 这是因为大部分工具都需要更多系统调用的支持才能运行, 因此我们无法在Nanos-lite上运行它们.

Busybox会把其中的Shell工具链接成一个ELF可执行文件, 而不是像Ubuntu/Debian等发行版中的Shell工具那样, 每个工具都是独立的ELF可执行文件. Busybox的 `main()` 函数会根据传入的参数来调用相应工具的功能:

```
if (strcmp(argv[1], "cat") == 0) return cat_main(argc, argv);
else if (strcmp(argv[1], "ls") == 0) return ls_main(argc, argv);
else if (strcmp(argv[1], "wc") == 0) return wc_main(argc, argv);
// ......
```

Busybox提供了一个简单的安装脚本, 通过创建一系列的软链接来让用户方便地使用这些小工具:

```
$ ls -lh navy-apps/fsimg/bin
total 1.6M
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 base64 -> busybox
-rwxr-xr-x 1 yzh yzh 161K Oct 21 12:11 bird
-rwxr-xr-x 1 yzh yzh 126K Dec  9 12:12 busybox
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 cat -> busybox
-rwxr-xr-x 1 yzh yzh  33K Oct 28 20:43 cpp-test
-rwxr-xr-x 1 yzh yzh  29K Dec  9 12:12 dummy
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 echo -> busybox
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 ed -> busybox
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 false -> busybox
-rwxr-xr-x 1 yzh yzh  33K Dec  9 12:12 hello
-rwxr-xr-x 1 yzh yzh  81K Dec  9 12:12 menu
-rwxr-xr-x 1 yzh yzh  91K Dec  9 12:12 nterm
-rwxr-xr-x 1 yzh yzh 586K Dec  9 12:12 onscripter
-rwxr-xr-x 1 yzh yzh 390K Dec  9 12:12 pal
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 printenv -> busybox
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 sleep -> busybox
lrwxrwxrwx 1 yzh yzh    7 Dec  9 12:12 true -> busybox
```

这样以后, 我们键入 `cat` 命令, 实际上是执行 `/bin/busybox` , 来让它执行链接到Busybox的 `cat_main()` 函数.

> 运行Busybox

尝试通过NTerm运行Busybox中的一些简单命令, 比如 `cat` 和 `printenv` 等, 如果你不清楚这些命令的用法, 可以通过 `man` 来查阅它们. 注意不要让这些命令的输出淹没在 `hello_fun()` 打印的信息中, 为此你可能需要调整 `hello_fun()` 打印信息的频率.

有一些工具并不是放在 `/bin` 目录下, 而是放在 `/usr/bin` 目录下, 例如 `wc` . 为了不必输入完整的路径, 我们可以把 `/usr/bin` 也加入到 `PATH` 环境变量中. 不同的路径通过 `:` 进行分隔, 具体格式可以参考在Linux上运行 `echo $PATH` 命令的结果. 这样以后, 我们就可以通过一个库函数 `execvp()` 来尝试遍历 `PATH` 中的所有路径, 直到找到一个存在的可执行文件为止. 我到之后就会调用 `SYS_execve` . 你可以通过阅读 `navy-apps/libs/libc/src/posix/execvp.c` 来了解这一功能是如何实现的.

不过为了遍历 `PATH` 中的路径, `execvp()` 可能会尝试执行一个不存在的用户程序, 例如 `/bin/wc` . 因此Nanos-lite在处理 `SYS_execve` 系统调用的时候就需要检查将要执行的程序是否存在, 如果不存在, 就需要返回一个错误码. 我们可以通过 `fs_open()` 来进行检查, 如果需要打开的文件不存在, 就返回一个错误码. 此时 `SYS_execve` 返回 `-2` . 另一方面, libos中的 `execve()` 还需要检查系统调用的返回值: 如果系统调用的返回值小于0, 则通常表示系统调用失败, 此时需要将系统调用返回值取负, 作为失败原因设置到一个全局的外部变量 `errno` 中, 然后返回 `-1` .

> -2和errno

`errno` 是C标准定义的, 运行时环境中的一个全局变量, 用于存放最近一次失败的系统调用或库函数调用的错误码. 你可以通过运行 `errno -l` 命令(需要通过 `apt-get` 安装 `moreutils` 包) 来查看所有的错误码及其含义. 你应该能看到错误码 `2` 是你比较熟悉的一种错误. 关于 `errno` 全局变量的更多信息, 可以参考 `man 3 errno` .

> 运行Busybox(2)

实现上述内容, 让 `execvp()` 支持 `PATH` 的遍历, 然后尝试通过NTerm运行 `wc` 等位于 `/usr/bin` 目录下的命令, 例如 `wc /share/games/bird/atlas.txt` . 你可以通过在Linux上运行相应命令来查看结果是否正确. 此外, 你可以通过阅读 `execvp()` 的代码来帮助你判断返回值的设置是否正确.

虽然目前我们只能在Nanos-lite上运行很少部分的Busybox工具, 但你基本上在自己构建的计算机系统里面呈现了与你平时使用Linux命令行工具非常相似的一幕. 我们把这件事放到Project-N的系统里面, 就是为了能够让你明白, 你平时键入命令的时候计算机系统的各个抽象层都做了些什么:

- 终端如何读取用户的按键?
- Shell如何进行命令的解析?
- 库函数如何根据命令解析出的字符串搜索到可执行文件?
- 操作系统如何加载执行一个可执行文件?
- .....

虽然Project-N和真实的Linux系统还有很大的差异, 但独立完成PA已经可以很大程度上帮助你消除对"程序如何在计算机上运行"的神秘感.

> 温馨提示

PA阶段1到此结束.

导航

程序和内存位置
绝对代码
可重定位代码
位置无关代码

虚实交错的魔法
分段

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 `0x3000000` 或 `0x83000000` 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中的对象(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 `0x3001234` , 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 `0x3001234` , 访问 `0x3001234` 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".

我们在PA1中提到了BIOS这个概念, 事实上, BIOS中的程序就是这种自重定位的程序. 这是因为, BIOS一般是固化在ROM中, 无法对ROM中的数据进行写操作. 计算机启动后会从BIOS中执行程序, 但这个BIOS程序马上会把自己拷贝到其它内存位置, 并进行一些重定位工作, 然后才真正开始执行. 这时候, 数据就位于可写的RAM中了, BIOS程序就可以对其进行写操作.

但对多任务操作系统来说, 这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忍略的, 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的发行版上配套的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, 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PC指示的存储器位置取出指令;
  执行指令;
  更新PC;
}
```

如果引入了虚拟内存机制, PC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, 操作系统必定无法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工作的机制: 操作系统负责进行物理内存的管理, 加载进程的时候为进程分配一个新的物理地址. 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存内存管理机制, 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, 进程不需要关心自己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:

```
        15                0    31                                    0
LOGICAL +-----------------+    +-------------------------------------+
ADDRESS |    SELECTOR     |    |              OFFSET                 |
        +-----+---------+-+    +-------------------+-----------------+

      +-------+         V                           |
      | DESCRIPTOR TABLE                            |
      |  +------------+                             |
      |  |            |                             |
      |  |            |                             |
      |  |            |                             |
      |  |            |                             |
      |  |            |                             |
      |  |------------|                             |
      |  |  SEGMENT   | BASE          +---+       |
      +->| DESCRIPTOR |---------------->| + |<------+
         |------------| ADDRESS       +-+-+
         |            |                 |
         +------------+                 |
                                        V
             LINEAR  +------------+-----------+--------------+
             ADDRESS |    DIR     |   PAGE    |    OFFSET    |
                     +------------+-----------+--------------+
```

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, 因此分段机制的各种概念, 我们也不会加入到NEMU中来.

By Zihao_Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

- 引言
- 超级容器的界限
- 分页
  - 页表结构
  - 页表项（PTE）的重要性
  - 页表的内存开销
- TLB - 地址转换的加速
- 页表遍历
- 页表项（PTE）的重要性
- 页表的内存开销

最后提醒一下x86页面级地址转换时出现的一种特殊情况：在用户x86并没有产格要求数据对齐，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。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。而mips32和riscv32允许内存地址非对齐，每个取数据都严格按照4字节对齐，因此不会发生这样的情况，否则CPU将会抛出异常，可见软件灵活性和硬件复杂度是计算机系统中又一对tradeoff。

在分页机制上运行Nanos-lite

实现以下内容：

- Nanos-lite的 `pg_alloc()` 、`pg_alloc()` 的参数是分配空间的字节数，但我们保证AM通过回调函数调用 `pg_alloc()` 时申请的空间总是页面大小的整数倍，因此可以通过调用 `new_page()` 来实现 `pg_alloc()` ，此外 `pg_alloc()` 还需要对分配的页面清零。
- VME的 `map()` 你可以通过 `as->pt` 获取页目录的基地址，若需要申请新的页表，可以通过回调函数 `pgalloc_usr()` 向Nanos-lite获取一页空闲的物理页。
- 在NEMU中实现分页机制。

由于此时Nanos-lite运行在内核的虚拟地址空间中，而该些映射又是相等映射，因此NEMU的地址转换操作 `pa` 必定与 `va` 相同。你可以把这一条件作为assertion加入到NEMU的代码中，从而帮助你捕捉实现上的bug。

如果你的实现正确，你会看到仙剑奇侠传也可以成功运行，如果你对分页机制的细节有疑问，请RTFM。

对于x86和riscv32，你无需实现TLB。

对于mips32，此时并不能检查你的实现是否正确，这是因为在mips32中，程序需要访问用户空间才会触发地址转换的过程，因此如果你选择了mips32，你需要完成下文的内容才能测试你的实现。

让DiffTest支持分页机制

为了让DiffTest机制正确工作，你需要

- 针对x86，在 `restart()` 函数中执行需要对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，需要将分页机制相关的寄存器同步到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。

但对于mips32来说，引入的TLB也属于机器状态，因此要完美地进行DiffTest，则需要考虑如何将TLB同步到REF中，这确实不是一件轻松的事情，你可以思考一下如何解决，不过放弃进行DiffTest也不失为一种解决方案。

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，毕竟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 `0x3000000` / `0x83000000` 的位置，是因为之前Nanos-lite并没有对空间的物理内存进行管理：现在引入了分页机制，由AM来负责所有物理页的分配，这意味着，如果将来AM把 `0x3000000` / `0x83000000` 所在的物理页分配出去，仙剑奇侠传的内容将会被覆盖掉(之前在运行 `exec-test` 的时候已经遇到过这个问题了)! 因此，目前仙剑奇侠传看似运行成功，其实面面暗藏杀机。

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，编译Navy应用程序的时候需要为 `make` 添加 `VME=1` 的参数，这样就可以将应用程序的链接地址改为 `0x40000000` ，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。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 `0x40000000` 附近开始，这个地址已经不在物理内存的地址空间中(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。

此外，我们还需要对创建用户进程的过程进行较多的改动。我们首先需要在如裁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 `AddrSpace` 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 `abstract-machine/am/src/nemu/isa/$ISA/vme.c` 。

不过，此时 `loader()` 不能直接把用户进程加载到内存位置 `0x40000000` 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. 由于AM native实现了权限检查, 为了让程序可以在AM native上正确运行, 你调用 `map()` 的时候需要将 `prot` 设置成可读可写可执行
- 从文件中读入一页的内容到这一物理页中

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.

另一个需要考虑的问题是用户栈，和 `loader()` 类似，我们需要把 `new_page()` 申请得到的物理页通过 `map()` 映射到用户进程的虚拟地址空间中，我们把用户栈的虚拟地址安排在用户进程虚拟地址空间的末尾，你可以通过 `as.area.end` 来得到末尾的位置，然后把用户栈的物理页映射到 `[as.area.end - 32KB, as.area.end)` 这段虚拟地址空间.

最后，为了让一地址空间生效，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 `as->pt` 加入到上下文中，框架代码已经实现了这一功能(见 `abstract-machine/am/include/arch/$ISA-nemu.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 `__am_irq_handle()` 的开头调用 `__am_get_cur_as()` (在 `abstract-machine/am/src/nemu/isa/$ISA/vme.c` 中定义)，来将当前的地址空间描述符指针保存到上下文中
- 在 `__am_irq_handle()` 返回前调用 `__am_switch()` (在 `abstract-machine/am/src/nemu/isa/$ISA/vme.c` 中定义) 来切换地址空间，将被调度进程的地址空间落实到MMU中

为mips32实现真正的分页机制

如果你选择的是mips32，现在你需要实现真正的分页机制了：如果不是，你可以忽略这道题。

当CPU尝试进行地址转换时，若TLB发生miss，硬件会将当前的虚拟页号设置到entryhi中，然后抛出TLB refill异常，这个异常会被CTE捕获，然后调用 `__am_tlb_refill()` 函数：

- 读出当前进程的页目录基地址(思考一下，该如何获得?)
- 从entryhi中读出虚拟页号
- 根据虚拟页号进行page table walk
- 将两个连续的虚拟页对应的物理页号设置到entrylo0和entrylo1中
- 执行tlb管理的相关指令更新TLB表项

从TLB refill异常返回后，CPU会再次执行相同的指令，这次的地址转换应该可以成功进行. 为了让软件来进行page table walk, 你需要实现 `__am_tlb_refill()` (在 `abstract-machine/am/src/nemu/isa/mips32/vme.c` 中定义).

根据上文内容, 我们还需要维护TLB表项和进程的关系, 这应该是通过ASID来完成的. 不过我们在这里可以进行简化, 参考x86, 我们也在切换地址空间的时候, 通过清空整个TLB来解决问题. 为此你还需要实现 `__am_tlb_clear()` (在 `abstract-machine/am/src/mips32/nemu/vme.c` 中定义).

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。如果你选择了mips32/riscv32，请注意地址空间描述符在上下文结构体中的位置，如果你不确定这一位置，请根据PA3中的讲义内容检查你的代码实现。

为了测试实现的正确性，我们先单独运行dummy(因忘记修改调度代码)，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

对于x86和riscv32，在 `protect()` 中创建地址空间的时候，有一处代码用于拷贝内核映射:

```
// map kernel space
memcpy(updir, kas.ptr, PGSIZE);
```

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恰改总是成功，这是因为在实现分页机制之前，`0x3000000` / `0x83000000` 之上的内存原本就以用户进程自由使用，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之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中添加成员 `max_brk` ，来记录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.

一致性问题 - mips32的夏梦来了

分页的道理也说得差不多了，但如果你选择的是mips32，在运行仙剑奇侠传的过程中，你应该会遇到各种奇怪的问题，而且不少问题可能都是一致性问题导致的. 在这里，TLB中的内容可以说是内存页表项的副本...

唉，既然你有决心选择mips32，我们就不透露那么多了，尝试自己把这个问题理清楚，并思考相应的解决方案吧，解决了一致性问题，才能说得上对mips32的分页机制有彻底的理解.

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?

修复native VME的性能bug

我们修复了在AM native上开启VME后性能有明显下降的问题. 如果你在2021/01/10 16:45:00之前获取AM的框架代码, 请根据这个页面更新相关文件.

可以在用户栈里面创建用户进程上下文吗?

`ucontext()` 行为是在内核栈 `kstack` 中创建用户进程上下文, 我们是否可以对 `ucontext()` 的行为进行修改, 让它在用户栈上创建用户进程上下文? 为什么?

支持虚存管理的多道程序

终于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执行了, 不过我们还是先让运行在分页机制上的用户进程和内核线程并发执行, 为此, 你需要对进程进行进一步的调试.

为此, 我们需要思考内核线程的调度会对分页机制造成什么样的影响. 内核线程和用户进程最大的不同, 就是它没有用户态的地址空间: 内核线程的代码, 数据和栈都是位于内核的地址空间. 那在自动分页机制之后, 如果 `__am_irq_handle()` 要返回一个内核线程的现场, 我们是否需要考虑通过 `__am_switch()` 切换到内核线程的虚拟地址空间呢?

答案是, 不需要. 这是因为AM创建的所有虚拟地址空间都会包含内核映射, 无论在切换之前是位于哪一个虚拟地址空间, 内核线程都可以在这个虚拟地址空间上运行. 因此我们只需在 `kcontext()` 中将上下文的地址空间描述符指针设置为 `NULL` ，来进行特殊的标记. 等到将来在 `__am_irq_handle()` 中调用 `__am_switch()` 时, 如果发现地址空间描述符指针为 `NULL` ，就不进行虚拟地址空间的切换.

支持虚存管理的多道程序

让Nanos-lite加载运行仙剑奇侠传和内核线程hello. 此时的运行效果和上一阶段一样, 但这次整个系统都是运行在分页机制之上的.

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年代的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.

并发执行多个用户进程

让Nanos-lite加载仙剑奇侠传和hello这两个用户进程; 或者是加载NTerm和hello内核线程, 然后从NTerm启动仙剑奇侠传, 你应该会在运行的时候观察到错误. 尝试分析这一错误的原因, 并总结为了支持这一功能, 我们需要满足什么样的条件.

这可以说是一周目最难的一道思考题了, 虽然我们会在PA4的最后给出分析, 喜欢挑战的同学仍然可以在这里尝试独立思考: 如果你能独立解决这个问题, 说明你对细节的理解可以说是相当了得了.

Hint: 程序是个状态机.

这一阶段的内容算是整个PA中最难的了, 连选做题的难度和之前比也不是一个量级的. 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, 代码会相应地变得越来越难以理解. 这是AP到我们精理代码的行为也有巨大的帮助, 当你在理解, 实现, 调试这些AP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, 就算你以后面对更复杂的项目, 相信你也不会畏惧了.

温馨提示

PA4阶段2到此结束.

目 · 概览
分时图表
日线图表
周线图表
月线图表
季线图表
年线图表
多日分时图表
多日日线图表
均线图表
MACD
KDJ
RSI
BOLL

分时图表

多数情况下股市周期了市的时间表现，但是特别是不一定是分钟，但是一个更短分钟时间表现的情况下，股市的表现情况，特别是在特别的情况下的情况，这种情况下的情况特别的情况，情况一下，情况一下的情况特别，情况特别是情况特别是情况特别的情况下，这是在特别的情况下的情况下，这一个情况下特别的情况下，这一个情况下特别的情况下，这是在特别的情况下的情况下，这是在特别的情况下的情况下，这是在特别的情况下

```
void __am_asm_trap() {
  if (ksp != 0) {   // ksp is global
    c->usp = $sp;   // usp should be in Context
    $sp = ksp;
  }
  c->np = (ksp == 0 ? KERNEL : USER);  // np should be in Context

  ksp = 0;          // support re-entry of CTE

  // save context

  __am_irq_handle(c);

  // restore context

  if (c->np == USER) {
    ksp = $sp;
    $sp = c->usp;
  }

  iret();
}
```

另一处可以优化的地方是把 `c->usp` 的功能合并到 `c->sp` 中，让 `c->sp` 表示进入CTE前 `$sp` 的值，无论进入CTE前系统是位于用户态还是内核态。同样地，恢复上下文的时候，直接把 `c->sp` 赋值给 `$sp` 即可，这样就无需在Context结构体中额外定义 `usp` 了，因此上述代码可以优化为：

```
void __am_asm_trap() {
  c->sp = $sp;
  if (ksp != 0) {   // ksp is global
    $sp = ksp;
  }
  c->np = (ksp == 0 ? KERNEL : USER);  // np should be in Context

  ksp = 0;          // support re-entry of CTE

  // save context

  __am_irq_handle(c);

  // restore context

  if (c->np == USER) {
    ksp = $sp;
  }
  $sp = c->sp;

  iret();
}
```

栈切换的具体实现

上文讨论的只是概念上的解决方案，为了在代码中实现内核栈的切换，我们还需要把上文中提到的几个概念变量映射到ISA相关的具体实现中。

mips32

对内核栈切换的实现来说，mips32是最简单的。根据mips32 ABI的约定，通用寄存器中预留了 `k0` 和 `k1` 两个寄存器专门给内核使用，编译器不会把变量分配到 `k0` 或 `k1` 中，因此我们可以进行如下约定：

- 把概念上的 `ksp` 映射到 `k0` 寄存器
- 把概念上的 `c->np` 映射到 `c->gpr[k1]`
- 把概念上的 `c->sp` 映射到 `c->gpr[sp]`

你只需要根据上述约定在CTE中添加少量代码，就可以实现内核栈切换的功能了。

riscv32

和mips32不同，riscv32 ABI并没有约定类似 `k0` 和 `k1` 的寄存器用法。相反地，riscv32提供了一个叫 `sstratch` 的CSR寄存器，专门作为临时寄存器给系统软件使用，它在硬件的行为上并没有什么特殊之处。因此我们可以进行如下约定：

- 把概念上的 `ksp` 映射到 `sstratch` 寄存器
- 在Context结构体中添加一个新的成员 `np`，把概念上的 `c->np` 映射到它
- 把概念上的 `c->sp` 映射到 `c->gpr[sp]`

不过当你打算实现代码的时候，你会发现在刚进入CTE而未切换到内核栈之前，我们根本没有空闲的通用寄存器可以使用！事实上，riscv32的设计非常精巧，我们可以通过3条指令来完成内核栈的切换：

```
__am_asm_trap:
  csrrw sp, sscratch, sp   // (1) atomically exchange sp and sscratch
  bnez sp, save_context    // (2) take the branch if we trapped from user
  csrr sp, sscratch        // (3) if we trapped from kernel, restore the original sp

save_context:
  // now sp is pointing to the kernel stack
  // save the context...
```

为了检查 `sscratch` 的值，我们通过(1)巧妙地使用了CSR访问指令的原子交换功能来交换 `sp` 和 `sscratch` 的值，这样既可以把原来的 `sscratch` 值读到通用寄存器 `sp` 中进行比较(riscv32的分支指令不能访问CSR寄存器)，又可以把原来的 `sp` 值保存到 `sscratch` 寄存器中避免被破坏。接下来通过(2)判断刚才读出的 `sscratch` 值(即概念上的 `ksp`)：如果值是 `0`，说明在进入CTE之前是内核态，而此时内核栈的指针已经被(1)交换到 `sscratch` 寄存器中了，于是通过(3)把内核栈的指针读回来；如果值不是 `0`，说明在进入CTE之前是用户态，需要把 `sp` 切换到内核栈，但指令(1)已经完成了这一切换工作了，因此此时直接跳转到 `save_context` 处来保存上下文即可。

❷ 临时寄存器的方案

同样是作为CTE的临时寄存器，mips32选择在GPR中分配出 `k0` 和 `k1`，而riscv32则是采用 `sstratch` 这个CSR寄存器。这两种方案相比，是否存在哪一种方案更优？为什么？

x86

在x86中实现内核栈的切换则比较特殊，这是因为x86的异常处理机制会直接把 `eflags`，`cs` 和 `eip` 保存到栈上，软件代码无法干涉这一行为。如果我们希望把这3个寄存器保存到内核栈上，那就需要由x86硬件来进行内核栈的切换。类似地，x86的 `iret` 指令通过弹栈的方式恢复现场的，这意味着我们同样无法通过软件来恢复用户栈的栈指针：如果我们在 `iret` 指令之前先切换到用户栈，那么 `iret` 指令就无法找到正确的 `eflags`，`cs` 和 `eip`；但如果我们先执行 `iret` 指令，`eip` 将会指向用户代码，此时我们已经无法恢复用户栈的栈指针。

因此x86必须通过硬件机制来提供上述问题的解决方案。首先我们必须让硬件识别出当前的特权级，这是通过 `CS` 寄存器中的 `RPL` 域来识别的(也即i386手册中 `CPL` 的定义)。若 `CS.RPL` 为0，表示处理器处于内核态；若 `CS.RPL` 为3，表示处理器处于用户态。然后我们根据根据i386手册对x86的异常处理机制和 `iret` 指令的行为进行扩展：

- 若异常发生在内核态，则异常处理机制的行为与PA3的介绍一致（见i386手册中Figure 9-5的 `WITHOUT PRIVILEGE TRANSITION`）
- 若异常发生在用户态，则异常处理机制先通过TSS机制找到 `ksp`，然后往 `ksp` 中先压入 `ss`（栈段寄存器）和 `esp`，最后再压入 `eflags`，`cs` 和 `eip`（见i386手册中Figure 9-5的 `WITH PRIVILEGE TRANSITION`）
- 执行 `iret` 指令时，先依次弹出 `eip`，`cs` 和 `eflags`，若此时弹出的 `cs` 表示将要返回到用户态，则再继续弹出 `esp` 和 `ss`

完整的TSS机制有两个作用，一是用于在硬件上进行上下文切换，二是用于在用户态发生异常时进行内核栈的切换。在PA中我们只需要关心后者即可，但要作好TSS机制，我们就不得不引入我们本来希望在PA中简化掉的分段机制了。不过你在PA3已经对异常处理机制有所了解了，这会对你理解TSS机制有所帮助。我们先来回顾一下x86的异常处理机制：

```
从IDTR寄存器中读出IDT这个数组的首地址，用异常号对IDT进行索引，得到一个门描述符，然后把门描述符中的offse
```

然后我们就可以来类比TSS机制的处理过程了：

```
从GDTR寄存器中读出GDT这个数组的首地址，用TR寄存器中的idx域对GDT进行索引，得到一个TSS描述符，然后把TSS描
```

嗯，好像也不是特别难理解。不过异常处理机制中最终得到的地址是异常入口地址，而TSS机制最终得到的地址则是TSS结构体的地址。TSS结构体中有非常多的成员，但在内核栈的切换过程中，硬件只会用到 `ss0` 和 `esp0` 这两个成员（见 `abstract-machine/am/src/x86.h` 中 `TSS32` 结构体的定义）。ICS课本的第七章介绍了用TSS机制的异常处理过程，关于TSS机制的细节，也可以查阅i386手册。

由于硬件TSS机制的支持，CTE在软件上只需要维护概念变量 `ksp` 就可以了：

- 把概念上的 `ksp` 映射到TSS结构体中的 `esp0` 成员
- 把概念上的 `c->np` 映射到 `c->cs.RPL`
- 在Context结构体中添加两个新的成员 `ss3` 和 `esp3`，把概念上的 `c->sp`（实际上是 `c->usp`）映射到 `c->esp3`

为了让TSS机制在将来可以正确工作，我们还需要在 `cte_init()` 中添加一些初始化代码：

```
#define NR_SEG 6
static SegDesc gdt[NR_SEG] = {};
static TSS32 tss = {};

bool cte_init(Context*(*handler)(Event, Context*)) {
  // ...

  // initialize GDT
  gdt[1] = SEG32(STA_X | STA_R,   0,      0xffffffff, DPL_KERN);
  gdt[2] = SEG32(STA_W,           0,      0xffffffff, DPL_KERN);
  gdt[3] = SEG32(STA_X | STA_R,   0,      0xffffffff, DPL_USER);
  gdt[4] = SEG32(STA_W,           0,      0xffffffff, DPL_USER);
  gdt[5] = SEG16(STS_T32A,     &tss, sizeof(tss) - 1, DPL_KERN);
  set_gdt(gdt, sizeof(gdt[0]) * NR_SEG);

  // initialize TSS
  tss.ss0 = KSEL(2);
  set_tr(KSEL(5));

  return true;
}
```

我们在GDT这个数组中设置了5个描述符，前4个分别表示内核态代码段、内核态数据段、用户态代码段、用户态数据段，它们是为了在DiffTest的时候可以让QEMU正确地进行特权级的处理，我们无需关心其中的细节。

✎ 实现内核栈和用户栈之间的切换

理解上述讲义内容，修改CTE的代码，使其支持内核栈和用户栈之间的切换。你需要了解你的实现中是如何体现上述伪代码的功能，同时，根据不同概念变量的性质，你可能还需要在 `cte_init()` 或者 `kcontext()` / `ucontext()` 中对它们进行初始化。如果你希望在 `__am_irq_handle()` 中维护这些概念变量，必要的时候可以采用内联汇编。

实现后，让Nanos-lite加载NTerm和hello这两个用户进程，然后从NTerm启动仙剑奇侠传。如果你的实现正确，你将可以看到hello用户进程和NTerm/仙剑奇侠传分时运行，和之前的效果不同，这次我们正真实现了两个用户进程的分时运行。

一些提示：

- 为了方便调试，你可以先在NEMU中关闭时钟中断，这样可以提升系统的确定性，测试成功后再打开时钟中断
- mips32 - 在保存上下文之前，`k1` 就已经是一个可用的寄存器
- riscv32 - 为了维护 `c->np`，你可能需要在切换到内核栈之后马上保存少部分GPR，或者借助全局变量来帮助你
- x86 - 除了在硬件上实现必要的寄存器和硬件机制，你还需要进行以下的初始化：
  - 在 `kcontext()` 中把 `c->cs` 设置成 `KSEL(1)`
  - 在 `ucontext()` 中把 `c->cs` 设置成 `USEL(3)`，把 `c->ss3` 设置成 `USEL(4)`

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。

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

编写不同的传奇

在PC的基础，我们来做些好玩的事情。

展示你的计算机系统

这个部分是关于如何在计算机系统上展示信息，我们可以显示一个图形或者图表，我们可以加载自定义的图像，并在屏幕上字体进行不同的绘制。

在屏幕上绘制图像

Mobs的图像大小是32x32像素，可以用来显示不同的图像，我们可以加载自定义的图像，并在屏幕上字体进行不同的绘制。

```
Mobs.Image = "mob.bmp" -- 加载mob.bmp文件
Mobs.Image = "" -- 清空
Mobs.Image = "mob.bmp" -- 重新加载
Mobs.Image = "mob.bmp" -- 重新加载
Mobs.Image = "mob.bmp" -- 重新加载
```

字体方面Mobs.Text是一个字符串，可以用来显示不同的字体，Mobs.Text.Color是一个颜色，Mobs.Text.Size是一个数字，Mobs.Text.Font是一个字体名称，Mobs.Text.Bold是一个布尔值，Mobs.Text.Italic是一个布尔值，Mobs.Text.Underline是一个布尔值，Mobs.Text.Strikeout是一个布尔值。

```
Mobs.Text = "Hello"
Mobs.Text.Color = 255
Mobs.Text.Size = 12
Mobs.Text.Font = "Arial"
Mobs.Text.Bold = true
Mobs.Text.Italic = false
Mobs.Text.Underline = false
Mobs.Text.Strikeout = false
```

在PC上的Scripts

在PC上的Scripts文件夹中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

在PC上的Scripts目录

在PC上的Scripts文件夹中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件

在PC上的Scripts文件中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹

在PC上的Scripts文件夹中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件

在PC上的Scripts文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件

在PC上的Scripts文件夹中的文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts目录

在PC上的Scripts目录中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件

在PC上的Scripts文件中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹

在PC上的Scripts文件夹中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件

在PC上的Scripts文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件

在PC上的Scripts文件夹中的文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹中的文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts目录

在PC上的Scripts目录中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件

在PC上的Scripts文件中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹

在PC上的Scripts文件夹中，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件

在PC上的Scripts文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件

在PC上的Scripts文件夹中的文件夹中的文件，可以放置自定义的脚本，这些脚本可以用来扩展游戏的功能。

```
Scripts/
  mod_example.lua
  mod_example.txt
  mod_example.bin
```

在PC上的Scripts文件夹中的文件夹中的文件夹

在PC上的Scripts文件夹中的文件夹中的文件夹

DMA和CPU协作

在NUMA的所有设备中，磁盘是最慢的一个设备，因为它可以重新排列指令的执行，设备会排列工作方式（DMA）。DMA（直接内存访问）是一种硬件设备，它能直接绕过内核直接访问内存。DMA会排列对内存的访问，以便在内核忙碌时能顺利完成数据传输。

下学期预排：操作系统课程设计

修改进程状态 - 阻塞/唤醒SWM

我们实现操作系统的重要部分，我们下学期开始了解操作系统的进程，操作系统必须决定哪些时间应该安排进程执行，哪些时间应该暂停进程，哪些进程应该唤醒，哪些进程应该阻塞。

进程状态

- 阻塞/唤醒
- 计算机进程

操作系统通常会为每个进程分配一个数字，这个数字通常会用作内核内部的一个数组的一个索引。这个数字通常被称为进程号，NUMA-UNIX的实现会定义每个用于管理进程的数据结构，我们定义数据类型为进程号。

NUMA-UNIX的一个特点是我们会为每个进程分配一个数字，并通过这个数字来管理进程状态。我们定义数据类型为进程状态，可能是有多种状态，但是我们现在只需要两种状态：阻塞和唤醒。

NUMA-UNIX的一个特点是我们会为每个进程分配一个数字，并通过这个数字来管理进程状态。我们定义数据类型为进程状态，可能是有多种状态，但是我们现在只需要两种状态：阻塞和唤醒。

操作系统中这个数字通常会用作内核内部的一个数组的一个索引，这个数组通常被称为进程表。操作系统管理进程表通常是通过一个叫做调度程序的程序来管理的。调度程序会决定哪些进程应该被唤醒，哪些进程应该阻塞，哪些进程应该暂停，哪些进程应该恢复。

进程表

- 进程号
- 进程状态

NUMA-UNIX的一个特点是我们会为每个进程分配一个数字，并通过这个数字来管理进程状态。我们定义数据类型为进程表，这个数组通常被称为进程表。

进程表

- 进程号
- 进程状态

进程表

- 进程号
- 进程状态

进程表

- 进程号
- 进程状态

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.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

此页面修订于: 2021-04-19 09:48:46

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, 就是要优化热点代码, 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.

那么怎样才能找到热点代码呢? 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, 对每处代码的运行时间进行统计. 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?

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

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PC... 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 `isa_exec_once()` , 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PC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PC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)
mov $0x100005,(cpu.pc的地址)
```

这两条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PC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PC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.pc);
  if ( cache miss ) {
    tb = jit_translate(cpu.pc); // translate a basic block
    update_cache(cpu.pc, tb);
  }

  jump_to(tb); // cpu.pc 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.

这之后的讲义内容, 你, 也许就是作者.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

PA是什么意思
PA是不是很难
为什么要用这么变态的PA来折磨我
我将来不从事系统相关的工作，就不会私下编写
现在是开源的时代了，开源在《十四五规划和2035年远景目标纲要》中首次被列入国家发展规划，那我为什么不能参考别人的代码/心得/攻略来学习
为什么不能把代码/心得/攻略上传到公开的地方让大家学习
我总是得讲义写得不清楚
总是要求大家STFW/RTFM/RTFSC，我有问题都不敢问了，老师助教跟是轻松
为什么用C语言写PA
我发现自己做PA的同时顺显大于讲义中预估的时间
我觉得PA好像是一样的菜鸡很不友好
我连完成PA1都觉得非常吃力
我感觉大部分同学都无法在一学期内做完PA的全部内容
坚持了好久还是搞不定，我想放弃了
我觉得在这模板上面浪过的话，这样后面做的同学就会倒霉一些
为什么我本地测试都是对的，但提交到OJ总是无法通过
PA的内容太丰富了，我感觉没办法完全消化
下学期的OSlab是不是比PA更难
PA和OSlab有什么区别和联系
和大进的PA相比有什么区别，哪个更难
我觉得PA有的设计不合理

常见问题

PA是什么意思

PA是Programming Assignment的缩写，yzh也觉得这个名字起得有点随意。

PA是不是很难

PA是一个**超级棘手**的编程实验，其中的硬核体现在，只有同时满足以下条件，你才有可能比较顺利地完成PA：

- 端正心态，学会STFW/RTFM独立解决问题
- 掌握工具，知道用什么样的工具快速解决什么样的问题
- 了解细节，明白每一个文件每一行代码的意义
- 不断编程，掌握计算机系统中每个模块之间的联系

如果之前没有受到过类似的训练，即便是高年级的研究生也会觉得有一定的困难，所以对绝大部分同学来说，做PA就是一个吃苦的过程。

为什么要用这么变态的PA来折磨我

因为大家普遍都很菜，按照目前的状态，你毕业之后很可能无法在社会的毒打下存活。以下例子展示了将来你可能会面对什么样的毒打：

```
你来晚一天做/老板：你来晚一个xxx，目标是达到yyy的效果。
```

讲义？程序代码？不存在的，如果我们不提供讲义和程序代码，让你写一个能跑仙剑的NEMU，这就和你毕业后需做的任务很类似了。

这就是社会毒打你的第一招必杀技：充满不确定性的未知问题。如果你现在连转类都很不好，显然你几乎无法胜任这些不确定的任务，作为算是红得往往会毒打的人，我们知道你缺什么，知道应该给你提什么样的试，设置什么样的找提训练，但就会得到什么程度的成长。

所以我们设计PA来把现在的你毒打一顿，让你受尽折磨，吃苦苦头，引领到底是为了让将来的你在求职时候更具有竞争力，经受得起社会的毒打。

我将来不从事系统相关的工作，就不会私下编写

太天真了，就算是应用程序，几千个模块之间通过几万个API交互的场景都是家常便饭，复杂性是社会毒打你的第二招必杀技。

现在是开源的时代了，开源在《十四五规划和2035年远景目标纲要》中首次被列入国家发展规划，那我为什么不能参考别人的代码/心得/攻略来学习

你来上课来做实验，就应该接受我们设置的训练，而独立完成实验是训练最基本的要求之一，参考别人代码就是违反学术诚信的行为，针对这一点你没有任何理由讨价还价。

就算是一万步来讲，你说参考别人是为了学习，那你能写出比别人更好的代码作为学习的成果吗？你要是真有这样的学习能力，相信你也能够在遵守学术诚信的前提下自己独立完成实验。

同理，阅读别人流传的心得和攻略，也是属于违反学术诚信的行为，因为你实在通过自己的努力和预期的训练来获得正确的答案。

为什么不能把代码/心得/攻略上传到公开的地方让大家学习

因为这违反了学术诚信。PA的其它信息不一样，它本质上是一个课程作业，你无法保证亲看你代码的同学都是抱着纯洁的学习心态跟着看代码的，如果有同学抱着别的心态，你的代码不仅会让他失去锻炼机会，而且还会破坏课程的公平性：你一个bug得3天，你同学1分钟就抄完了，你甘心吗？

此外，我们假惺惺的代码重露并没有多少值得大家学习的地方，比如你的代码风格好看吗？你设计的API合理吗？和超长代码相比有什么新特色吗？在你提得自己的代码之前，可以先看看一些优秀的开源项目。如果你确实对自己的代码有信心，不妨动少提提问，如果你的建议很不错，我们会把你加入到德著名单中，同时你的改进也可以一直在学术学术的作业平流传。

如果你上传代码只是为了炫耀，那可以选择github的私人仓库。

同理，将自己的心得和攻略上传到公开的地方，也是属于违反学术诚信的行为。

所以，你公开的代码/心得/攻略很大概率并不能从真正的意义上帮助大家学习，相反而是在很多层面上伤害大家。

能不能公布阶段性的参考答案，这样即便bug调不出来，也能往后做

不能，把代码写对是对码农最基本的要求，代码还没写对，就不应该往后做，以后你进入社会，如果你交付的代码给客户造成经济损失，是要赔钱的，所以要学会对自己写的代码负责。

我总觉得讲义写得不清楚

你毕业后进入公司/课题组，不会有讲义具体地告诉你应该做什么，总有一天你需要在脱离讲义的情况下完成任务，我们希望你现在就放弃“讲义和菜鸡代码会把我应该做的一切细节清楚地告诉我”的幻想，为自己的成长负起责任：

- 不知道在说什么 -> 知识点不清楚，看书
- 不知道要做什么/怎么做 -> 看别的系统或，RTFSC
- bug调不出来 -> 不知道正确的预期行为，RTFSC，不重视工具和方法的使用，花时间去体验和总结

如果你发现自己有以上情况，你还是少抱怨，多吃苦吧。

当然，如果你仍然觉得讲义写得不清楚，可以联系yzh。

总是要求大家STFW/RTFM/RTFSC，我有问题都不敢问了，老师助教跟是轻松

老师助教不是你的保姆，你需要先搞清楚哪些问题是你可以问的，哪些问题是你应该自己解决的：我们鼓励你提出关于方法层面的疑问以及对各种问题的探索性想法，但不要直接让别人提供你想要的答案，因为学会独立分析问题是你做PA训练的一个重要环节。

一个我们不愿意回答的提问如下：

```
Q: 我的程序段错误了，怎么办？
A: 机器本这是对的。
```

如果你确实需要一些调试相关的帮助，那怎么办？你需要展示你为了解决问题而付出过的努力，当我们看到了你的努力，也许会认可你受到的训练，例如：

```
Q: 我的程序在xxx的情况下段错误了，我进行了以下尝试
- 首先我做了aaa，结果是AAA，我得到的结论是XXX
- 然后我做了bbb，结果是BBB，我得到的结论是YYY
- 我还做了ccc，结果是CCC，我得到的结论是ZZZ

现上，我觉得问题可能出在yyyy，但我接下来没有思路了，我的分析和理解都是正确？或者我忽略了什么吗？
```

为什么用C语言写PA

首先C语言容易学，可以给菜鸡们一个容易把编程能力上的机会，此外C语言的抽象少，更接近机器，更容易理解“程序如何在计算机上运行”；相比而言，你现可能既不清楚C++/python/java/scala/rust等各种语言特性相应的机制，在调试代码的时候，还会给你带来麻烦，作为一个公型项目，用C语言来写PA还是应可接受的范围的。

我发现自己做PA的用时明显大于讲义中预估的时间

讲义中的时间是按照“中等水平/同学的能力来预估的，这里“中等水平”并不是指“程序设计课程总评和95以上”，而是指“心态端正，编写过500行以上代码的单个程序，并且懂得调试”。

根据过去的情况，绝大部分程序设计课程总评90分以上的同学都达不到我们所说的“中等水平”的标准，所以总了你的程序设计课程成绩吧，对PA来说它并没有什么价值。(不过我们还是希望有一天这段话的描述不再符合实际)

虽然程序设计课程成绩不能反映你们真实的编程水平，但自己的编程水平是百礼天，大部分同学还是心中有数的，所以在这里我们明确编程水平对讲义预估的时间进行调整，得到一个更加可信的预估时间 (假设讲义上预估的时间是 `x` 小时):

- 神：`x / 5`
- 巨佬：`x / 3`
- 大佬：`x / 2`
- 编程菜鸡比：`x`
- 不会调试：`x * 2`
- 不会写代码：`x * 3`
- 即不会写代码又不认真做PA：`x * 5`

我们知道大部分同学还是容易地判断自己完成PA的真实用时，所以推延你就输了。

我觉得PA对像我一样的菜鸡很不友好

程序设计实验对菜鸡友好，但为什么你做完之后还是菜鸡？做一个对菜鸡友好的实验，能力并不会得到提升。

如果你是菜鸡，说明你之前该吃苦的时候没吃，现在回到人多吃苦就是应该的。

我连完成PA1都觉得非常吃力

PA1和计算机系统的关联度偏低，它主要是为了测试大家的编程能力而设计的，如果你连完成PA1都觉得非常吃力，那你就要给自己敲响警钟了：快去学编程吧。

我感觉大部分同学都无法在一学期内做完PA的全部内容

很正常，PA从一开始设计的时候就没有想过你让所有同学完成所有内容：PA的设计理念是把目标定得很高，强迫大家注上进，所以在你前面感觉到PA和你差过的其它实验都不太一样：只要你愿意付出足够的努力，即便你没有完成所有的内容，你也会感到自己各个方面的技能都有了明显的提升。

每年确实会有同学直接躺倒放弃而挂科，但也总有为了PA不少一批同学能提持到最后，锻炼出专业程序员该有的素质，大家的编程水平有高有低，如果你自己觉得编程能力不行，那你就应该去把编程能力补上来，尽那之前就不要抱有“完成PA所有内容”的幻想了，一个比较合法的目标是“独立完成PA1和PA2”，最后你仍然有机会通过课程。

坚持了好久还是搞不定，我想放弃了

也许是你坚持的姿势不对，来跟yzh或者jyy聊聊吧。

我觉得应该搬集大家踩过的坑，这样后面做的同学就会顺利一些

这种想法是不对的，抹杀重然后自己思出来最进步最深刻的思法进行调整，随着主体PA的演进，和大进PA的区别也逐渐增大，量子“难”这个说法比较主观，我们从另一个角度来展示两者的区别：对于一个难点，大进PA的设计理念是通过降低要求(降低原创量后的要求)来保证大部分同学能够顺利可以获得上进程度，我们认为不应该为了照顾部分菜鸡感慨而所有学生或长或苦的机会，所以主张PA没有选择降低要求，而是让大家根据自身情况设置一个合适目标去努力。

我觉得PA有的设计不合理

欢迎和yzh乱聊。

≡ 导航

一知半解
数组求和
整数的平方
多重定义符号
奇怪的函数返回值
时间复杂度和功能都相同的程序
网友贴出的一道百度招聘题
整数除法

站得高，看得远
实践是检验真理的唯一标准

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 = xxx = -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=107269248` 。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 `2.00000006103516`
- `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 copy1(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 copy2(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 `copy1` 更快还是 `copy2` 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呢？现在计算机科学发展这么快，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，对于某些公司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等并存，使得原先基于PC机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时代的应用程序需要复杂, 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特徵.

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层次之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, 具有良好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, 名比率和时延, 并发和并行, 这程过程调用(PPC), 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, 程序移植和建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.

实践是检验真理的唯一标准

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容易, 但当你真正动手实践的时候, 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？你知道程序如何分配内存和释放内存, 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:

> **| 思考题**
>
> 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设计的机理?
>
> 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, 但它们并不属于之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. 这些是培训技术学习计算机本科教学的根本区别, 但如果你对GUI程序设计的机理了如指掌, 对图像处理基本原理的理解就如探囊取物, 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.

> **| 来公课**
>
> 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吧?

Linux入门教程

以下内容引用自公开的课件资料及技术资料网站，并有少量修改和补充，如果你是第一次使用Linux，请你一定仔细阅读教程，一定要试试行教程中提到的命令。

探索命令行

Linux是专门设计的命令使用模式都是相同的：

命令名称 参数1 参数2 参数3 ...

参数之间用空格隔开，参数可以是文件名，关于命令行，可以先阅读一些基本常识。然后我们介绍最常用的一些命令：

- ls 用于列出当前目录的文件，如果你在Windows下使用过DOS，你会发现这个命令和DOS的dir命令很类似。ls -l 可以显示详细信息。
- mkdir 用来创建新的目录。
- cd 进入一个目录，如果你在Linux中，每个目录中都至少包含两个目录：.（指向目录自身）和..（指向它的上级目录）。目录的根目录是/。
- touch NEWFILE 可以创建一个内容为空的新文件 NEWFILE，如果 NEWFILE 已存在，则内容不会改变。
- cp SOURCE DEST 可以将文件 SOURCE 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 可以将文件 SOURCE 改名为 DEST 文件，如果 DEST 是一个目录，则将 SOURCE 文件移动到该目录下。
- ln -s SOURCE SYMLINK 可以创建一个符号连接。
- rm FILE 删除指定的一个文件。rm -rf DIR 递归地删除一个目录及其下的所有文件。
- cp newfile aaa 将文件newfile复制为目录文件aaa
- mv aaa bbb 将文件aaa改名为bbb
- mv bbb ... 将文件bbb移动到上级目录
- cd .. 切换到上级目录
- cd bbb 切换到目录bbb
- cd / 切换到上级目录
- cd ~ 进入你的home目录（/home/用户名目录）

关于命令参数

如果你是第一次使用 mv（移动或改名）这个熟悉接下来的知识吧便利 之外，还会教你在使用一些新的命令行工具时如何获得帮助。

动手试试

上述各个命令做了 cd 之外都被称为它们的manpage，这是为什么？如果你查看它们的感谢信吧，试着到互联网上查查看。

下面给出一些常用命令使用的例子，你可以输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 .# 将上级目录的文件newfile复制到当前目录下
$ cp newfile aaa # 将文件newfile复制为目录下的aaa
$ mv aaa bbb     # 将文件aaa改名为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 -rr 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，诚然，我们可以给你推荐下列信息：只要我们能想到的，那一定有万里的沟通障碍吗 因此这些想要知道更多最全面又不知道应该到什么时候，请在Google中进行，如果你想在Linux的世界来的更全，那就可以在世界的互联网上进一步探索的不能最新资源。

统计代码行数

第一个你了若进行一个程序中包含了很多的代码行数，如果想知道当前目录下究竟有多少行的代码，就可以在命令中输入如下命令：

```
find . | grep "\.[ch]"|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中一个名，如你想在Windows目录的目录下的文件，如果你想要知道当前目录下的所有文件，如果你文件很多的话呢 find . 将会搜索当前目录下的所有文件，可以进行 grep 的一个 [ch]，这样的话 grep 的功能是 -- 一个 "print lines matching a pattern"，grep 实际工作人的过程，我们 grep 有一个参数 -l 就是 "print the name of each file that contains the pattern"，这样的话 grep 的工作就是在当前目录下的所有文件中查找 .c 或 .h 结尾的文件，并且输出一个文件名的列表。这样的话 wc 命令的参数 -l 是 "print the number of lines"，这样的话 wc 命令的工作就是统计一个文件的行数，这样的话 find 的工作就是 grep 的工作，grep 的工作就是 wc 的工作，这样的话 wc 的工作就是 find 的工作。

我们最后的任务是要把这些文件的总数的统计出来，此时可以用 wc 的参数 -w 来做 wc -w，命令的 -l 选项是统计新文件的行数，-w是统计文件的单词数，-c 是统计文件的字节数，它能够将标准输入转换为参数，所这就是一个参数所指定的参数，所以，代码命令的 xargs wc -l 既等价于执行 wc -l main.c main.h include/xxx.h ...，最终显示的行数统计。

统计磁盘使用情况

以下命令统计 /usr/share 目录下各个目录的总用时磁盘空间：

```
du -sc /usr/share/* | sort -nr
```

du 是磁盘空间统计工具，du -sc 用目录的大小统计输出目标磁盘，他用通过管道传送给 sort，sort -nr 是数据排序工具，其中的选项 -n 表示按照数值进行排序，-r 表示倒序输出。sort 可以将这些参数排序后输出。

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，我们可以再查看一个命令：more 或 less。

```
du -sc /usr/share/* | sort -nr | more
```

此时将查看输出的前几行内容，more 工具使用空格键翻页，并可以用 q 键退出查看结果。关于个人查看习惯来说，这样是不过，那就是因为 more 工具的功能太简单了，不可以向下翻页，还可以上翻页，所特别是有，缺陷是，这里是有一个关于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，如果你的用户名是 user，你的主目录就是 /home/user，用户的 home 目录可以用波浪线 ~ 替代，例如如果对于目录 /home/user/templates，可以使写为 ~/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：

```
cd ~
mkdir Templates
```

创建成功后，输入

```
cd Templates
```

可以完成目录的切换，注意每输入目录名时，tab 键可以提供帮助。

动手试试输入编辑器？

你可能会想要查找你喜爱编入类似于

```
cd ~/Keryo/Long/LibNane
```

的命令，你一定觉得很繁琐，回顾上面所说的那段之一，如果你感到有什么地方不对，那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如zsh，提供更多富好用的功能，总之，尽试吧，你觉得最舒服的。

进入主题的目录后就可以编辑文件了，开发软件中文件的编辑大编辑器是 vim 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 这两款编辑器都要一定的时间才能上手，它们共同的特点是要要花较多的时间才能适应基本操作方式或令或快捷键，但一旦熟练后呢，编辑效率就比传统的编辑器快很多。

进入了工程的目录后，输入相应的命令将编辑开始编辑文件，例如输入

```
vi hello.c
#emacs hello.c
```

就是开始一个文件编辑，例如可以输入如下的此时文本文件 vi 或 emacs 的时间，输入代码可能会花去一些时间，在编辑的时候记得要查看帮助之的时间。

```
#include <stdio.h>
int main(void) {
  printf("Hello, Linux World!\n");
  return 0;
}
```

保存后就编译程序 hello.c 的内容，编译命令可以用 cat hello.c 查看代码的内容，如果要用它编辑，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，它的文件是 assembler output，也会令引输入

```
./hello
```

就能够运行程序了，如果你的 .c 源不能少的，点代表了当目录，而 ./hello 的意思是当前目录下的 hello 文件，不Windows不同，Linux系统默认情况下并不是当前目录，这是因为Linux下有大量的标准工具的 /usr 和 /bin 都是有几用产品已编写的程序集，不能变成目录的某个子目录中的缺点。

使用重定向

有时我们希望将程序的输出结果保存到文件中，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将 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。

你你在实践，使用了输出重定向之后，屏幕上将不显示由 myprog 输出的打印信息，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在方便快速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时呢？当你需要反复测试 myprog 进行测试的时候，你需要多次输入大量相同的数据，为了避免这种无意义的重复输入，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，你可以重复测试 myprog 输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，而省了大量的时间。

下面给出了一个综合使用重定向的例子：

```
time ./myprog < data | tee output
```

这个命令在进行 myprog 的时间统计，程序运行文件 data 中读入数据，并将其输出信息打印到屏幕和文件 output 中，这样，工具记录了这一过程所消耗的时间，最后你查看屏幕上看到 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 > /dev/null 来将 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，你能想出是怎么实现的吗？如之上，那样你可将 myprog 的输出过滤掉，保留了 time 的计时结果，方便又简洁。

使用Makefile管理工程

大型编程工程中通常会有几个甚至成百上千个源文件(Linux内核源码有2000+的源文件)，分别输入命令行进行编译是非常不现实的，Linux提供了一个高效管理工具 make 来帮助我们进行编译工作。make 工作流程：make 会根据 Makefile 文件的内容来决定哪些文件需要编译，并发出相应的命令来调用 gcc 编译器。Makefile 的语法非常复杂，我们在这里只介绍一个简单的例子来说明 make 的工作流程。

下面给出了一个综合使用重定向的例子：

```
time ./myprog < data | tee output
```

这个命令在进行 myprog 的时间统计，程序运行文件 data 中读入数据，并将其输出信息打印到屏幕和文件 output 中，这样，工具记录了这一过程所消耗的时间，最后你查看屏幕上看到 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 > /dev/null 来将 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，你能想出是怎么实现的吗？如之上，那样你可将 myprog 的输出过滤掉，保留了 time 的计时结果，方便又简洁。

使用Makefile管理工程

大型编程工程中通常会有几个甚至成百上千个源文件(Linux内核源码有2000+的源文件)，分别输入命令行进行编译是非常不现实的，Linux提供了一个高效管理工具 make 来帮助我们进行编译工作。make 工作流程：make 会根据 Makefile 文件的内容来决定哪些文件需要编译，并发出相应的命令来调用 gcc 编译器。Makefile 的语法非常复杂，我们在这里只介绍一个简单的例子来说明 make 的工作流程。

下面给出了一个简单的 Makefile 文件的全使用实例，这段，使用Shell令，指定使用为可能，如果你希望了解更多的Makefile的使用 Makefile，请到互联网上搜索相关资料。

```
hello:hello.c
    gcc hello.c -o hello   # 注意不是tab，而不是空格

.PHONY: clean

clean:
    rm hello   # 注意不是tab，而不是空格
```

通过命令行，输入 make，你会发现 make 程序调用了 gcc 进行编译，Makefile 文件的最不明白的地方，相信你的第一感觉了：

```
目标文件名:依赖文件
    用于生成目标文件的命令行语句   # 注意不是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。

如果你这样输入的 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，并且它比所依赖的文件都要"新"，那不生成目标的命令是不会被执行的。

上面的示例中，以 .PHONY: clean 指明以这种特殊，它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 程序总是要重复执行命令生成目标，但指明目标的伪文件 clean 为 .PHONY 并不是用来生成 clean 文件，因此这种的命令总是会被执行，你需要输入 make clean 来使用 make 程序执行 clean 规则，这里的为 make 默认执行 Makefile 中文本顺序最靠前的规则，因此，最靠不靠后，目录下已经有了一个名为 clean 的文件，那对 make clean 命令执行文件已经是最新版本"的提示，解决这个问题的方法是在 Makefile 中加入一个 .PHONY: clean，用于指示 clean 是一个伪目标"，这样以后，make 程序将不会再检查 clean 文件的存在与否，仍旧会根据命令的内容来执行。

对于一个规模较大一点的工程，Makefile 文件会使用变量，函数，调用Shell令，指定规则为可能，如果你希望了解这些更好的编写一个 Makefile，请到互联网上搜索相关资料。

综合示例：教务助分程序

使用编辑器编辑文件 jwzh.sh 为如下内容(所有的制表符 tab 键的内容，但是此前是不可能发现之前的)：

```
#!/bin/bash
save_file="score" # 保存文件
semester=2012 # 学期名称，2010代表2010年秋季学期
log_home="http://jwzh.ln.cn.edu.cn:8080/jiaowu" # 教务系统登录网址
log_login="http://jwzh.ln.cn.edu.cn:8080/jiaowu/student/student/loginAction.do" # 登录页面地址
log_query="http://jwzh.ln.cn.edu.cn:8080/jiaowu/student/student/xsXuanKeCheXiaoJieGuoChaXun.do?method=queryXuanKeCheXiaoJieGuoChaXun" # 查询页面地址

name="$1xxxxxx" # 登录用户名
password="xxxxxx" # 登录密码

# 保存cookie文件，这是中间文件，cookie的意思是HTTP协议中的JSESSIONID字段中
cookie="wget -q -O - $log_home --save-headers | \
  sed -n 's/Set-Cookie: JSESSIONID=([0-9A-Z]*).*/\1/p'"
# 用户登录，登录成功后会在/tmp/login.log文件中记录登录信息，以供调试使用
wget -q -O - --header="Cookie:JSESSIONID=$cookie" --post-data \
  "username=$name&password=$password" "$log_login" > /dev/null
# 查询成绩表，查询结果保存在/tmp/score.log文件中，这是中间文件，如果出现问题，可以查看这个文件
wget -q -O - --header="Cookie:JSESSIONID=$cookie" "$log_query" > \
  # 删除掉多余的内容，只保留成绩表部分
list="cat tmp | sed -n -e '/<table.*id=[0_0]*.*>/,/<\/table>/p' \
  | sed '/<tr>/,/<\/tr>/ p' \
  | sed 'NPh1d;$!D' | sed '/^$/d;s/^[ \t]*//;s/[ \t]*$/;/g'"
# 对list内容进行处理，使得对它的分析更容易
for item in $list; do
  score="cat tmp | grep -A 20 $item | sed '/<td>/p' | sed -n '/^.*<.*>([^<]*)<.*$/s//\1/p'"
  score="echo $score"
  if [[ $score != '' ]]; then # 如果存在成绩
    grep $item $save_file &>/dev/null # 查找该课程是否已记录
    [[ $? ]] && : || echo -e $score\t$item # 如果记录在案，则不输出，否则输出成绩和课程名称
    # 如果成绩记录到文件/var/jwzh.log中，则不输出，否则输出成绩和课程名称
    # 输出成绩和课程名称的格式为：成绩[tab]课程名称
    echo $item >> $save_file # 将课程记录在案
  fi
fi
done
```

运行这个程序需要安装一个行编辑器 bash，jwzh.sh，将脚本程序赋予执行权限，如果要设定默认行进一脚本，可以使用chmod命令来赋予其权限 +x，将命令放加到/bin目录或/usr/bin目录之下，这样，以后直接输入命令即可使用。

为了提高这个程序，首先需要一些对HTTP协议的基础知识，HTTP是浏览器和服务器传输的文本协议，一次浏览器或程序向服务器发出一个文本格式的HTTP请求，服务器收到请求后，将会以一个文本格式的HTTP应答来回答浏览器，服务器和浏览器通信的过程可以通过命令行工具来模拟实现，这里这里有些情况下，如浏览器会加入一些特殊的参数，比如，服务器通过Cookie来区分不同的客户，比如登录被保存在Cookie中，这样，服务器就可以通过Cookie来区分不同的客户，比如登录状态

≡ 导航
初识man
寻找帮助
探索man
开始旅程

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! 还是得认真做实验, 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的真实身份其实是

> 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...

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

≡ 导航

光玉
人生如戏, 戏如人生
游戏设置
查看存档信息
存档
读档
第三视点
更多功能

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.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

≡ 导航
全局类型
Monitor相关
寄存器相关
指令执行相关
内存相关
DiffTest相关
NEMU作为DUT时使用
NEMU作为REF时使用

NEMU ISA相关的API说明文档

为了提高性能, 这些API也可以通过宏定义实现, 但本文档还是以C变量或函数的方式列出, 以给出变量或函数的类型.

全局类型

```
word_t;
```

表示与ISA字长等长的无符号类型, 在32位的ISA中为 `uint32_t` , 在64位的ISA中为 `uint64_t` .

```
sword_t;
```

表示与ISA字长等长的有符号类型, 在32位的ISA中为 `int32_t` , 在64位的ISA中为 `int64_t` .

```
char *FMT_WORD;
```

`word_t` 类型对应的十六进制格式化说明符, 在32位的ISA中为 `"0x%08x"` , 在64位的ISA中为 `"0x%016lx"` .

Monitor相关

```
char isa_logo[];
```

用于在未实现指令的报错信息中提示开发者阅读相关的手册.

```
word_t IMAGE_START;
```

表示镜像在内存中的位置距离内存起始地址的偏移.

```
void init_isa();
```

在monitor初始化时调用, 进行至少如下ISA相关的初始化工作:

- 设置必要的寄存器初值, 如 `PC` 等
- 加载内置客户程序

寄存器相关

```
struct {
  // ...
  word_t pc;
} CPU_state;
```

寄存器结构的类型定义, 其中必须包含一个名为 `pc` , 类型为 `word_t` 的成员.

```
CPU_state cpu;
```

寄存器结构的全局定义.

```
void isa_reg_display();
```

用于打印寄存器当前的值.

```
word_t isa_reg_str2val(const char *name, bool *success);
```

若存在名称为 `name` 的寄存器, 则返回其当前值, 并设置 `success` 为 `true` ; 否则设置 `success` 为 `false` .

指令执行相关

```
struct {
  // ...
} ISADecodeInfo;
```

用于存放ISA相关的译码信息, 会嵌入在译码状态结构体 `DecodeExecState` 的定义中.

```
vaddr_t isa_exec_once();
```

执行 `cpu.pc` 指向的指令, 并更新 `cpu.pc` , 返回被执行指令的下一条静态指令的PC.

内存相关

```
word_t PMEM_BASE;
```

表示物理内存的起始地址.

```
int isa_vaddr_check(vaddr_t vaddr, int type, int len);
```

检查当前系统状态是否支持内存区间为 `[vaddr, vaddr + len)` , 类型为 `type` 的内存访问. 其中 `type` 可能为:

- `MEM_TYPE_IFETCH` : 取指令
- `MEM_TYPE_READ` : 读数据
- `MEM_TYPE_WRITE` : 写数据

函数返回值可能为:

- `MEM_RET_OK` : 该内存访问可以在物理内存上直接进行
- `MEM_RET_NEED_TRANSLATE` : 该内存访问需要经过地址转换
- `MEM_RET_FAIL` : 该内存访问失败, 需要抛出异常(如RISC架构不支持非对齐的内存访问)

```
paddr_t isa_mmu_translate(vaddr_t vaddr, int type, int len);
```

对内存区间为 `[vaddr, vaddr + len)` , 类型为 `type` 的内存访问进行地址转换. 函数返回值可能为:

- `pg_paddr` | `MEM_RET_OK` : 地址转换成功, 其中 `pg_paddr` 为物理页面的地址(而不是 `vaddr` 翻译后的物理地址)
- `MEM_RET_FAIL` : 地址转换失败, 原因包括权限检查失败等不可恢复的原因, 一般需要抛出异常
- `MEM_RET_CROSS_PAGE` : 地址转换失败, 原因为访存请求跨越了页面的边界

```
bool isa_has_mem_exception();
```

查询当前是否有访存请求抛出异常, 如有则返回 `true` , 否则返回 `false`

DiffTest相关

NEMU作为DUT时使用

```
bool isa_difftest_checkregs(CPU_state *ref_r, vaddr_t pc);
```

检查当前的寄存器状态是否与 `ref_r` 相同, 其中 `pc` 为 `cpu.pc` 的上一条动态指令的PC, 即 `cpu.pc` 的旧值. 如果状态相同, 则返回 `true` , 否则返回 `false` .

```
void isa_difftest_attach();
```

将当前的所有状态同步到REF, 并在之后的执行中开启DiffTest.

NEMU作为REF时使用

```
void isa_difftest_getregs(void *r);
```

将当前的寄存器状态复制到 `r` 中.

```
void isa_difftest_setregs(const void *r);
```

将 `r` 中的寄存器状态复制到当前的寄存器中.

```
void isa_difftest_raise_intr(word_t NO);
```

触发一个编号为 `NO` 的中断/异常.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布

x86指令系统简介

i386手册有一套专门用怎么了所有指令的细节，附录中的opcode map也很有用。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。

1 i386手册勘误

由于PC厂商本的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，用于提供手册勘误的修改，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出现了奇怪的错误，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| immediate |
| 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，其他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| immediate |
| prefix   |size prefix|size prefix|override|      |      |      |            |           |
+--------------------------------------------------------------------------------+
|                                   66                   c7    84    99    00 e0 ff ff    01 00 |
+--------------------------------------------------------------------------------+
```

关什么 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，同时决定了 opcode 之后的比特串的解释。如果你熟悉编译器，opcode 那部很清晰的话ref7，PA中的x86部经过简化，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。

2 编码的艺术

对于以下5个聚合：

1. 用有 instruction prefix
2. 用有 address-size prefix
3. 用有 operand-size prefix
4. 用有 segment override prefix
5. 用有 opcode 的第一个字节

它们是两不相交的，这是必须的！这背后反映了某种哲理：

对于以下5个聚合：

1. 用有 instruction prefix
2. 用有 address-size prefix
3. 用有 operand-size prefix
4. 用有 segment override prefix
5. 用有 opcode 的第一个字节

它们是两不相交的，这是必须的！这背后反映了某种哲理：

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，详细的功能会在用来进行描述：

```
ModR/M byte
7   6   5   4   3   2   1   0
+---+-------+-------+-------+
|mod|  r/op/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中有一个字节的转义码 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个字节又显得不够了，这时候一个字节又会表示所有的指令形式了，x86在2个字节以后的基础上又引入了3个字节转义码。指令 opcode 的剩余几个字节是 ModR/M 和 Sib 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 的形式进行扩充。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。

指令集编码

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。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> B8 + 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时于i386手册有误。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实现的是功能的模拟，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，千万不要把它们搞清了！总结的表格及八卦以理解的Diagrams。

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“将一个处理器寄存器的数据传送到它的内存或寄存器单元内存中”，其中 r/m 表示“寄存器或内存”。
- Opcode一列中的编码都是两个六进制数字，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 内，以和上面的描述一致。至于 r/m 表示的是寄存器还是内存，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，表示内存的时候还要各个位数等加上方式，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
```

这两种形式的 opcode 都是一样的，唯通不会出现这么巧？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 ，作用是指示即将要读入的指令的数据大小。在i386中，通常来说，如果这个前缀没有出现，操作数默认是32位；当这个前缀出现的时候，操作数默认是操作数宽度改变成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中很少出现)。除句话说，如果把一个开头为 89 , ,, 的比特串解释成指令，它默认应该解释成 MOV r/m32,r32 的形式，如果出现了的开头是 66 89 , ,, ，它就应该被解释成 MOV r/m16,r16 。

3 操作数宽度前缀的由来

i386是从80386发展过来的，80386是一个16位的时代，那多指令的16位版本在当时肯已经实现好了，要解决32位的新时代，兼容就成了需要讨论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完，假定80386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多用一个字节也没什么大不了，但在i386这台的那个连进步的时代你可以在i386手册的计算器和图7-9的代码，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，这样方法最大的好处是是不需要引入额外的操作码，从而在不会明显地使得程序代码变长，在实际情况下，i386处理器可以使用前缀来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额外逻辑才能实现。

到现在为止，<4>-<6>三种形式你也同自了：

```
< 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>-<8>各种形式的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
```

PA中的x86先掠了段寄存器的实现，所以它们暂时没有解析式的 mov 指令。

<9>-<14>这6种形式涉及到一种新的操作数的 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，它的作用是示段内偏移量，但PA中的x86是没有“段”的概念，目前可以理解成“相对于物理地址处的偏移量”，这种形式是 mov 指令的特殊形式，它打从记不通过 ModR/M 字节，让 displacement 直接跟在 opcode 后面，因此 displacement 本指示一个内存地址。

<15>-<17>三种形式涉及到两种新的操作数目标：

```
<15> B8 + 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寄存器的编号按照编号顺序加到 opcode 的最后一位，所以它们也可以看成通用寄存器的编码直接在 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时于i386手册有误。具体情况见下文的描述。
```

上述形式中的 /r 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 - 7 作为指令 /digit 记号的编号形式，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 add 指令的一种形式，用 add /5 表示 sub 指令的一种形式，只看 opcode 的部分节 88 不能区分它们。

注：在i386手册中，这3种形式的 mov 指令并没有 /r 的记号，在这里加入 /r 的目的是为了说明 /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，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。

指令集编码

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实现的是功能的模拟，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，千万不要把它们搞清了！总结的表格及八卦以理解的Diagrams。

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“将一个处理器寄存器的数据传送到它的内存或寄存器单元内存中”，其中 r/m 表示“寄存器或内存”。
- Opcode一列中的编码都是两个六进制数字，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 内，以和上面的描述一致。至于 r/m 表示的是寄存器还是内存，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，表示内存的时候还要各个位数等加上方式，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
```

这两种形式的 opcode 都是一样的，唯通不会出现这么巧？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 ，作用是指示即将要读入的指令的数据大小。在i386中，通常来说，如果这个前缀没有出现，操作数默认是32位；当这个前缀出现的时候，操作数默认是操作数宽度改变成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中很少出现)。除句话说，如果把一个开头为 89 , ,, 的比特串解释成指令，它默应该解释成 MOV r/m32,r32 的形式，如果出现了的开头是 66 89 , ,, ，它就应该被解释成 MOV r/m16,r16 。

4 操作数宽度前缀的由来

i386是从80386发展过来的，80386是一个16位的时代，那多指令的16位版本在当时肯已经实现好了，要解决32位的新时代，兼容就成了需要讨论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完，假定80386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多用一个字节也没什么大不了，但在i386这台的那个连进步的时代你可以在i386手册的计算器和图7-9的代码，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，这样方法最大的好处是是不需要引入额外的操作码，从而在不会明显地使得程序代码变长，在实际情况下，i386处理器可以使用前缀来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额外逻辑才能实现。

到现在为止，<4>-<6>三种形式你也同自了：

```
< 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>-<8>各种形式的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
```

PA中的x86先掠了段寄存器的实现，所以它们暂时没有解析式的 mov 指令。

<9>-<14>这6种形式涉及到一种新的操作数目标 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，它的作用是示段内偏移量，但PA中的x86是没有“段”的概念，目前可以理解成“相对于物理地址处的偏移量”，这种形式是 mov 指令的特殊形式，它打从记不通过 ModR/M 字节，让 displacement 直接跟在 opcode 后面，因此 displacement 本指示一个内存地址。

<15>-<17>三种形式涉及到两种新的操作数目标：

```
<15> B8 + 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寄存器的编号按照编号顺序加到 opcode 的最后一位，所以它们也可以看成通用寄存器的编码直接在 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时于i386手册有误。具体情况见下文的描述。
```

上述形式中的 /r 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 - 7 作为指令 /digit 记号的编号形式，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 add 指令的一种形式，用 add /5 表示 sub 指令的一种形式，只看 opcode 的部分节 88 不能区分它们。

注：在i386手册中，这3种形式的 mov 指令并没有 /r 的记号，在这里加入 /r 的目的是为了说明 /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，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。

为 ib , iw 分别表示8位、16位、32位立即数，因此 ib , iw 的作用是将紧跟在 opcode 后面的立即数读入，这样做的原因是为了减少内存访问次数，因此在真实的芯片设计过程中，CPU的译码部件需要增加额外逻辑才能实现。

为 id ，如果你不是很理解这句话的话，你可以忽略它，因为这种不会影响实现的进行。

到此为止，你已经掌握了对 mov 指令的理解了，需要说明的是，这里要的 mov 指令的例子并没有完全覆盖i386手册中指令集指导的所有记号，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，这样你才能完整地理解一条指令的功能。Exceptions 条目涉及到该指令可能产生的异常，由于PA中的x86不打算加入异常处理的机制，你可以不用关心这一条。

导航

简单mov指令的执行
取指(instruction fetch, IF)
译码(instruction decode, ID)
执行(execute, EX)
更新PC

复杂mov指令的执行

x86的mov指令执行例子剖析

我们举两个 `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

我们先来剖析第一条 `mov $0x1234, %eax` 指令的执行过程.

取指(instruction fetch, IF)

首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb8` .

译码(instruction decode, ID)

在 `fetch_decode_exec()` 函数中用这条指令的第一个字节 `0xb8` 来查找 `switch-case` 的分支, 发现这一指令的操作数宽度是 `4` 字节的 `mov` 指令, 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.

接下来还需要识别指令的操作数. 对于 `mov $0x1234, %eax` 指令来说, 识别操作数其实就是识别寄存器 `%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, 我们还需要通过 `instr_fetch()` 函数来获得它.

执行(execute, EX)

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` 指令编写一个执行辅助函数即可. 这个函数就是 `exec_mov()` , 它是通过 `def_EHelper` 宏来定义的:

```
def_EHelper(mov) {
    write_operand(s, id_dest, dsrc1);
    print_asm_template2(mov);
}
```

其中 `write_operand()` 函数会根据第二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PC

调用 `update_pc()` 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通过一个ISA相关的译码信息成员 `s->isa.is_operand_size_16` 来记录操作数宽度前缀是否出现, `switch-case` 语句中的 `case 0x66` 分支实现了这个功能. `case 0x66` 分支对 `s->isa.is_operand_size_16` 成员变量做了标识之后, 通过 `goto` 语句回到函数的开头重新取出操作码, 此时取得了真正的操作码 `0xc7` . 由于 `s->isa.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` 查找相应的 `case` 分支, 调用译码辅助函数 `decode_mov_I2E()` , 这个译码辅助函数又分别调用 `decode_op_I()` 和 `operand_rm()` 这两个操作数译码辅助函数来取出操作数. 阅读代码, 你会发现 `operand_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/isa/x86/decode.c` 中定义), 它们的函数原型为

```
void load_addr(DecodeExecState *s, ModR_M *m, Operand *rm);
void read_ModR_M(DecodeExecState *s, Operand *rm, bool load_rm_val, Operand *reg, bool loa
```

它们将 `s->seq_pc` 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, 执行辅助函数 `exec_mov()` 会完成数据移动的操作, 最终在 `update_pc()` 函数中更新PC.

By Zihao Yu, 采用知识共享 署名-非商业性使用-相同方式共享 3.0 中国大陆 许可协议发布