Table of Contents

Introduction

计算机科学与技术系 计算机系统基础 课程实验

实验前阅读

最新消息

- 实验前请先阅读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我们

调试公理

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

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

成长是一个痛苦的过程

PA是充满挑战性的, 在实验过程中, 你会看到自己软弱的一面: 没到deadline就不想动手的拖延症, 打算最后抱大腿的侥幸, 面对英文资料的恐惧, 对不熟悉工具的抵触, 遇到问题就请教大神的懒惰, 多次失败而想放弃的念头, 对过去一年自己得过且过的悔恨, 对完成实验的绝望, 对将来的迷茫... 承认自己的软弱, 是成长的第一步; 对这样的自己的不甘, 是改变自己的动力. 做PA不仅仅是做实验, 更重要的是认识并改变那个软弱的自己. 即使不能完成所有的实验内容, 只要你坚持下来, 你就是非常了不起的! 你会看到成长的轨迹, 看到你正在告别过去的自己.

小百合系版"有像我一样不会写代码的cser么?"回复节选

- 我们都是活生生的人, 从小就被不由自主地教导用最小的付出获得最大的得到, 经常会忘记我们究竟要的是什么. 我承认我完美主义, 但我想每个人心中

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

实验方案

理解"程序如何在计算机上运行"的根本途径是实现一个真正的计算机系统, 但这需要很长的开发周期, 不适宜作为教学实验. 南京大学计算机科学与技术系 计算机系统基础 课程的小型项目(Programming Assignment, PA)将指导学生实现一个功能完备(但经过简化)的x86模拟器NEMU(NJU EMUlator), 通过实现NEMU来探究 程序在计算机上运行 的机理. NEMU受到了QEMU的启发, 结合了GDB调试器的特性, 并去除了大量与课程内容差异较大的部分. PA包括一个准备实验(配置实验环境)以及4部分连贯的实验内容:

- 简易调试器
- 指令系统
- 存储管理
- 中断与I/O

实验环境

- CPU架构: x86-64 or x86-32
- 操作系统: GNU/Linux(推荐ubuntu)
- 编译器: GCC（4.*版本:推荐4.7 or 4.8）
- 编程语言: C语言

如何获得帮助

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

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

如何适应查阅英文资料? 方法是尝试并坚持查阅英文资料.

搜索引擎, 百科和问答网站

为了查找英文资料, 你应该使用下表中推荐的网站:

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

一些说明:

- 一般来说, 百度对英文关键词的处理能力比不上Google.
- 通常来说, 英文维基百科比中文维基百科和百度百科包含更丰富的内容. 为了说明为什么要使用英文维基百科, 请你对比词条 前束范式 分别在百度百科, 中文维基百科和英文维基百科中的内容.
- stackoverflow是一个程序设计领域的问答网站, 里面除了技术性的问题(What is ":-!!" in C code?)之外, 也有一些学术性(Is there a regular expression to detect a valid regular expression?)和历史相关的问题(Why is the linux kernel not implemented in c++?).

官方手册

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

这里列出一些本课程中可能会用到的手册:

- Intel 80386 Programmer's Reference Manual (人手一本的i386手册)
- GCC 4.7.4 Manual
- GDB User Manual
- GNU Make Manual
- System V ABI for i386
- On-line Manual Pager (即man, 这里有一个入门教程)

GNU/Linux入门教程

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

PA0 - 世界诞生的前夜: 开发环境配置

PA0 - 世界诞生的前夜: 开发环境配置

世界诞生的故事 - 序章

PA讲述的是一个"上帝创造计算机"的故事.

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

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 15小时

截止时间: 见elearning

提交说明:

- 为了尽可能避免拖延症影响实验进度, 我们采用分阶段方式进行提交, 强迫大家每周都将实验进度往前推进. 在阶段性提交截止前, 你只需要提交你的工程, 并且实现的正确性不影响你的分数, 即我们允许你暂时提交有bug的实现. 在最后阶段中, 你需要提交你的工程和完整的实验报告, 同时我们也会检查实现的正确性.
- 如无特殊原因, 迟交的作业将损失30%的成绩(即使迟了1秒), 请大家合理分配时间.
- 但是, 如果你完全没有开始进行某阶段的实验内容, 请你不要进行相应的提交, 因为这会影响我们的工作. 一旦发现这种情况, 我们将会额外扣除你 `发现次数*10%` 的PA总成绩.

学术诚信: 如果你确实无法独立完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数.

下表说明了你可能采取的各种策略的收益:

	并非完全没有完成相应内容	完全没有完成相应内容	抄袭
按时提交	100%(获得完成部分的全部分数)	-发现次数*10%	0%, 并通知辅导员
未按时提交	70%(迟交惩罚)	-发现次数*10%	0%, 并通知辅导员
不提交	10%(学术诚信奖励)	10%(学术诚信奖励)	10%(学术诚信奖励)

总的来说, 最好的策略是: 做了就交, 没做就不要交.

提交地址: eLearning

提交格式: 把实验报告放到工程目录下之后, 使用 `make submit` 命令直接将整个工程打包即可. 请注意:

- 我们会清除中间结果, 使用原来的编译选项重新编译(包括 `-Wall` 和 `-Werror`), 若编译不通过, 本次实验你将得0分(编译错误是最容易排除的错误, 我们有理由认为你没有认真对待实验).
- 我们会使用脚本进行批量解压缩. `make submit` 命令会用你的学号来命名压缩包, 不要修改压缩包的命名. 另外为了防止出现编码问题, 压缩包中的所有文件名都不要包含中文.
- 我们只接受pdf格式, 命名只含学号的实验报告, 不符合格式的实验报告将视为没有提交报告. 例如 `141220000.pdf` 是符合格式要求的实验报告, 但 `141220000.docx` 和 `141220000张三实验报告.pdf` 不符合要求, 它们将不能被脚本识别出来.
- 如果你需要多次提交, 请先手动删除旧的提交记录(提交网站允许下载, 删除自己的提交记录)

git版本控制: 我们鼓励你使用git管理你的项目, 如果你提交的实验中包含均匀合理的, 你手动提交的git记录(不是开发跟踪系统自动提交的), 你将会获得本次实验代码分数20%的奖励(总得分不超过本次实验的上限). 这里有一个十分简单的git教程, 更多的git命令请查阅相关资料. 另外, 请你不定期查看自己的git log, 检查是否与自己的开发过程相符. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据.

实验报告内容: 你必须在实验报告中描述以下内容:

- 实验进度. 简单描述即可, 例如"我完成了所有内容", "我只完成了xxx". 缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验.
- 必答题.

你可以自由选择报告的其它内容. 你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- 你遇到的问题和对这些问题的思考
- 对讲义中蓝框思考题的看法
- 或者你的其它想法, 例如实验心得, 对提供帮助的同学的感谢等(如果你希望匿名吐槽, 请移步提交地址中的课程吐槽讨论区, 使用账号stu_ics登陆后进行吐槽)

认真描述实验心得和想法的报告将会获得分数的奖励; 蓝框题为选做, 完成了也不会得到分数的奖励, 但它们是经过精心准备的, 可以加深你对某些知识的理解和认识. 因此当你发现编写实验报告的时间所剩无几时, 你应该选择描述实验心得和想法. 如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求. 但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出来是凑字数的)

来让你的报告看起来十分丰富, 编写和阅读这样的报告毫无任何意义, 你也不会因此获得更多的分数, 同时还可能带来扣分的可能.

对, 你没有看错, 除了一些重要的信息之外, PA0的实验讲义都是英文!

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

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

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

作为过渡, 我们为大家准备了全英文的PA0. PA0的目的是配置实验环境, 同时熟悉GNU/Linux下的工作方式, 其中涉及的都是一些操作性的步骤, 你不必为了完成PA0而思考深奥的问题. 你需要独立完成PA0, 请你认真阅读讲义中的每一个字符, 并按照讲义中的内容进行操作: 当讲义提到要在互联网上搜索某个内容时, 你就去互联网上搜索这个内容. 如果遇到了错误, 请认真反复阅读讲义内容, 机器永远是对的. 如果你是第一次使用GNU/Linux, 你还需要查阅大量资料或教程来学习一些新工具的使用方法, 这需要花费大量的时间 (例如你可能需要花费一个下午的时间, 仅仅是为了使用 `vim` 在文件中键入两行内容). 这就像阅读英文材料一样, 一开始你会觉得效率很低, 但随着时间的推移, 你对这些工具的使用会越来越熟练. 相反, 如果你通过"投机取巧"的方式来完成PA0, 你将会马上在PA1中遇到麻烦. 正如etone所说, 你在专业上的技不如人, 迟早有一天会找上来. 至于你信不信, 我反正信了.

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

Installing a GNU/Linux VM

PA0 is a guide to GNU/Linux development environment configuration. You are guided to install a GNU/Linux VM (virtual machine). 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 the lecture note carefully. Remember, the machine is always right!

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

Installing VirtualBox

Download VirtualBox from this website according to your host operating system, then install VirtualBox. Note that if your host is GNU/Linux, you can install VirtualBox by

```
apt-get install virtualbox
```

in Ubuntu or Debian. Different distribution uses different package tools. Please search the Internet for more information.

You can use other virtualization softwares (such as VMware) instead of VirtualBox. Also, if you have troubles about VMware or other virtualization softwares, please search the Internet.

Installing a GNU/Linux VM

We choose the Debian distribution for the VM(virtual machine), since it can be quite small.

Getting Debian

You can get the newest stable version of Debian here. For our experiment, we use the netinst CD image with i386 architecture. Click the link label with `i386` in the `netinst CD image` category to download the image.

Creating a VM

1. Launch the VirutalBox software.
2. To create a new VM, click the `New` button in the tool bar. This will invoke the wizard for newing a VM.
3. **Name and operating system**. You can name anything with the new VM, such as `ics`. For OS type, choose `Linux` operating system and `Debian (32 bit)` version. Then click `Next`.
4. **Memory size**. Just leave the default setting for base memory size (`512MB`). Click `Next`.
5. **Hard drive**. Leave the default setting (`Create a virtual hard drive now`). Click `Next`. This will invoke another wizard for virtual hard drive creation.
    1. **Hard drive file type**. Choose the VDI file type (default setting). Then click `Next`.
    2. **Storage on physical hard drive**. Choose `Dynamically allocated` (default setting). Click `Next`.
    3. **File location and size**. Just leave the default settings (with size of 8GB). If you want to

modify the location of the virtual disk file, choose another path as you wish. But do NOT choose a path with Chinese character (such as `D:\我的虚拟机`), else VirtualBox may not recognize the path correctly. Click `Create`.

This will create a new VM as configured.

Loading Debian installation image

1. Choose the new VM in the VirtualBox Manager, then click the `Start` button in the tool bar to launch the VM.
2. If you launch the VM for the first time, a wizard will be invoked. Select the Debian image file you have downloaded as the media source. If you miss the first run wizard by mistake (such as clicking the `Cancel` button), don't worry. Click `Devices` in the menu bar in the VM window, navigate to `CD/DVD Devices`, click `Choose a virtual CD/DVD disk file...`, then select the Debian image file, and relaunch the VM.
3. After setting the Debian image file correctly, you will see the `Debian GNU/Linux installer boot menu`. Select `Graphical install` by keyboard. This will start the installation wizard. If your mouse is captured by the VM, press the `right Ctrl` key to release the mouse.

Installing Debian in the VM

在select and install software一步出现错误

据同学反应, 安装debian时会在select and install software一步出现如下错误:

> An installation step failed. You can try to run the failing item again from the menu, or skip it and choose something else. The failing step is : Select and install software

建议大家在安装debian前先断网, 例如拔网线, 不登陆p.nju等. 这样VM就无法访问Internet了, 从而不会因为尝试从镜像网站获取信息而失败. 但断网之后, 安装过程中会遇到类似"无法连接网络"的提示, 你可以忽略这些提示, 在断网状态下继续安装. debian安装结束后, 就可以重新连网了.

1. **Select a language**. Choose `English`. NOTE: Do NOT choose `Chinese`, because it may lead to some unnecessary issues, such as switching input method back and forth, and some encoding troubles. Remember, in all experiments, Chinese environment is unnecessary.
2. **Select your location**. Choose `Other` -> `Asia` -> `China`.
3. **Configure locales**. Just leave the default setting (`United States - en_US.UTF-8`).
4. **Configure the keyboard**. Just leave the default setting (`American English`).
5. **Configure the network**. Just wait.
   - Hostname: Just leave the default setting (`debian`).
   - Domain name: Just leave the default setting ().
6. **Set up users and passwords**. Since the VM is only for experimental usage, you do not need to set up complex 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. Again, a simple password will be fine.

7. **Configure the clock**. Just wait.
8. **Partition disks**. This step will perform disk partitioning.
    - Partitioning method: Choose `Guided - use entire disk`. If you are installing GNU/Linux in the host machine, and there are data in some partitions, choose `Manual` instead to perform partition configuration manually. Otherwise the existing data will be lost! For more details, click `help` or search the Internet.
    - Select disk to partition: You only have one disk, the virtual disk you created before, to select.
    - Partition scheme: Choose `All files in one partition`.
    - Overview: The guided partitioning method will configure the virtual disk into two partitions, one for the ext4 file system, whose mount point is set to the root of file system (labeled with `/`), the other for the swap area. Select `Finish partitioning and write changes to disk` and click `Continue`.
    - Write the changes to disks?: Select `Yes`.
9. **Install the base system**. Just wait.
10. **Configure the package manager**
    - Debian archive mirror country: The installation guide is going to download and install packages from the Internet, which is unnecessary for the experiment. Under a poor network environment, this may cost a long time. Click `Go Back`.
    - Continue without a network mirror: Select `Yes`.
11. **Select and install software**. Just wait.
    - Configuring popularity-contest: Just select your favor.
    - Software selection: Just leave the default setting (select `Standard system utilities`).
12. **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`.
13. **Finish the installation**. Click `Continue`.

After finishing the installation, the system will restart.

First Exploration with GNU/Linux

First Exploration with GNU/Linux

After booting and finishing system initialization, the follow message is displayed with a character interface:

```
Debian GNU/Linux 8 debian tty1

debian login: _
```

Enter your username and password set during the installation. Note that when you enter the password, your input will not be displayed on the screen.

After logging in, you will see the following prompt:

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

This working environment is call terminal. This prompt shows your username, host name, and the current working directory. The current working directory is ~ now. As you switching to another directory, the prompt will change as well. You are going to finish all the experiments under the environment, so try to make friends with terminal!

Where is GUI?

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

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

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

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

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

Now you can see how much disk space Debian occupies. Type the following command (every command is issued by pressing the `Enter` key):

```
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 you installed can be so "slim"?

To shut down the VM, it is recommended to issue command instead of closing the VM window rudely (just like you shut down Windows by the start menu, instead of unplugging the power):

```
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 VM, 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!!!

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!

Installing More Tools

Installing More Tools

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

Installing tools from the Debian image

Now you are going to install some tools from the Debian image for convinence. First "insert" the Debian image: Click `Devices` in the menu bar in the VM window, navigate to `CD/DVD Devices`, click `Choose a virtual CD/DVD disk file...`, then select the Debian image file. Then add the sources in the image to the APT's list by the following command:

```
apt-cdrom add
```

You will see the following prompt:

```
Please insert a Disc in the drive and press enter
```

Since you have already "inserted the Disc" just now, press the `Enter` key, and you will see some message is output. Read them, you will find an error labeled with "Permission denied". Switch to the root account and execute the above command again, you will find that the command is executed without errors. Now you can install the following tools.

sudo

```
apt-get install sudo
```

`sudo` allows you to execute a command as another user (usually root). This means you do not need to switch to the root account to execute a system administration command or modify a file owned by root. But before you can use `sudo`, you should add your user account to the sudo group:

```
addgroup jack sudo
```

Replace "jack" above with your username. To let the above command go into effect, you should login the system again. Type

```
exit
```

to go back to your user account. And `exit` again to logout, then login again. Now you can use `sudo`. If you find an operation requires root permission, append `sudo` before that operation. For example,

```
username@hostname:~$ sudo poweroff
```

Note that running `sudo` may require password. This password is your user account password, not the root one.

Why use "sudo" instead of "su"?

You may consider `sudo` unnecessary, because you can always perform all operations with the root account. But this may take your system at risk. Can you figure out why?

In fact, all operations related to system changing require root permission. If a malicious program obtains root permission, it can do very bad things, such as deleting system files, to destory your system! Therefore, if an operation can be performed without root permission, perform it without root permission.

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.

In these situations, `vim` is still a good choice. If you prefer to `emacs`, you can download and install `emacs` from network mirrors after the APT sources file is configured.

ssh

```
apt-get install openssh-server
```

`ssh` is a tool for remote accessing. Using `ssh` in the experiment can take some advantage of the host system. For `ssh` configuration, see the Logging in via SSH section.

Installing tools from the network mirrors

Since some tools needed for the PAs can not be found in the Debian image, you will download and install them from the network mirrors. The Debian image will not be used any longer, so "eject" the Debian image: Click `Devices` in the menu bar in the VM window, navigate to `CD/DVD Devices`, click `Remove disk from virtual drive`.

Before using the network mirrors, you should check whether the VM can access the Internet.

Checking network state

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

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

You should receive reply packets successfully:

```
PING www.a.shifen.com (220.181.111.188) 56(84) bytes of data.
```

```
64 bytes from 220.181.111.188: icmp_seq=1 ttl=51 time=5.81 ms
64 bytes from 220.181.111.188: icmp_seq=2 ttl=51 time=6.11 ms
64 bytes from 220.181.111.188: icmp_seq=3 ttl=51 time=6.88 ms
64 bytes from 220.181.111.188: icmp_seq=4 ttl=51 time=4.92 ms

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

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

Learning vim

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

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

  ```
  vim test
  ```

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

Some games operated with vim

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

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

The power of vim

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

The first example is to generate the following file:

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

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

normal state), then press the following keys sequentially:

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

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

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

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

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

```
aaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccddddddddddddddddddddddd
eeeeeeeeeeeeeeeeeeeeeeeeffffffffffffffffffffffff
ggggggggggggggggggggggggghhhhhhhhhhhhhhhhhhhhhhh
iiiiiiiiiiiiiiiiiiiiiiijjjjjjjjjjjjjjjjjjjjjjj
```

You want to modify it into:

```
bbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaa
ddddddddddddddddddddddddddcccccccccccccccccccccc
ffffffffffffffffffffffffeeeeeeeeeeeeeeeeeeeeeee
hhhhhhhhhhhhhhhhhhhhhhhhggggggggggggggggggggggg
jjjjjjjjjjjjjjjjjjjjjjjjiiiiiiiiiiiiiiiiiiiiiii
```

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

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

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

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

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

Adding APT sources

To use network mirrors, you should configure the source list file. This file lists the sources for `apt` to obtain software information. The source list file is called `sources.list`, and it is located under `/etc/apt` directory. Switch to this directory by `cd` command:

```
cd /etc/apt
```

Then open `sources.list` using `vim`:

```
vim sources.list
```

you can see the file content:

```
#

# deb cdrom:[Debian GNU/Linux 8.0.0 _Jessie_ - Official i386 NETI

#deb cdrom:[Debian GNU/Linux 8.0.0 _Jessie_ - Official i386 NETIN

deb cdrom:[Debian GNU/Linux 8.0.0 _Jessie_ - Official i386 NETINS
deb http://security.debian.org/ jessie/updates main
deb-src http://security.debian.org/ jessie/updates main
```

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

```
--- before modification
+++ after modification
@@ -7,4 +7,5 @@
-deb cdrom:[Debian GNU/Linux 8.0.0 _Jessie_ - Official i386 NETIN
 deb http://security.debian.org/ jessie/updates main
 deb-src http://security.debian.org/ jessie/updates main

+deb http://mirrors.163.com/debian/ jessie main contrib non-free
+deb http://ftp.cn.debian.org/debian/ jessie main contrib non-fre
```

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

After you are done, you should save your modification. Type

```
:w
```

to save the file. However, you receive an error message:

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

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

```
:w!
```

But you receive another error message this time:

```
"sources.list" E212: Can't open file for writing
```

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

```
:q!
```

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

```
ls -l
```

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

```
total 20
drwxr-xr-x 2 root root 4096 May  6 22:30 apt.conf.d
drwxr-xr-x 2 root root 4096 Apr 14 01:26 preferences.d
-rw-r--r-- 1 root root  432 May  6 22:30 sources.list
drwxr-xr-x 2 root root 4096 Apr 14 01:26 sources.list.d
drwxr-xr-x 2 root root 4096 May  6 22:30 trusted.gpg.d
```

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

You can see that the `sources.list` file is owned by root, and you do not have permission to modify it. Therefore, use `sudo` to open the file:

```
sudo vim sources.list
```

Then perform the modification. This time you should save the file successfully.

After saving the modification, you can tell `apt` to retrieve software information from the sources specified in `sources.list`:

```
apt-get update
```

This command requires root permission, too. And it requires Internet accessing. It costs some time for this command to finish. If some errors are reported, please check

- whether there are any typos in `sources.list`, and
- whether your host is able to access the Internet.

Installing tools for PAs

The following tools are necessary for PAs:

```
apt-get install build-essential    # build-essential packages, in
apt-get install gcc-doc        # GCC document
apt-get install gdb            # GNU debugger
apt-get install git            # reversion control system
apt-get install time        # we use the GNU time program instead
```

The usage of these tools is explained later.

More Exploration

More Exploration

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.

Configure vim

`vim` provides more improvements comparing with `vi`. But these improvements are disabled by default. Therefore, you should enable them first. You can append the following content at the end of `/etc/vim/vimrc`. To enable this settings, exit and run `vim` again. Note that contents after a double quotation mark `"` are comments, and you do not need to include them in `/etc/vim/vimrc`.

```
setlocal noswapfile " 不要生成swap文件
set bufhidden=hide " 当buffer被丢弃的时候隐藏它
set nocompatible " 关闭 vi 兼容模式
syntax on " 自动语法高亮
colorscheme evening " 设定配色方案
set number " 显示行号
set cursorline " 突出显示当前行
set ruler " 打开状态栏标尺
set shiftwidth=4 " 设定 << 和 >> 命令移动时的宽度为 4
set softtabstop=4 " 使得按退格键时可以一次删掉 4 个空格
set tabstop=4 " 设定 tab 长度为 4
set nobackup " 覆盖文件时不备份
set autochdir " 自动切换当前目录为当前文件所在的目录
filetype plugin indent on " 开启插件
set backupcopy=yes " 设置备份时的行为为覆盖
set ignorecase smartcase " 搜索时忽略大小写，但在有一个或以上大写字母时仍保
set incsearch " 输入搜索内容时就显示搜索结果
set hlsearch " 搜索时高亮显示被找到的文本
set noerrorbells " 关闭错误信息响铃
set novisualbell " 关闭使用可视响铃代替呼叫
set t_vb= " 置空错误铃声的终端代码
set showmatch " 插入括号时，短暂地跳转到匹配的对应括号
set matchtime=2 " 短暂跳转到匹配括号的时间
set magic " 设置魔术
set hidden " 允许在有未保存的修改时切换缓冲区，此时的修改由 vim 负责保存
set smartindent " 开启新行时使用智能自动缩进
set backspace=indent,eol,start " 不设定在插入状态无法用退格键和 Delete 键
set cmdheight=1 " 设定命令行的行数为 1
set laststatus=2 " 显示状态栏 (默认值为 1, 无法显示状态栏)
set statusline=\ %<%F[%1*%M%*%n%R%H]%=\ %y\ %0(%{&fileformat}\ %{&e
set foldenable " 开始折叠
set foldmethod=syntax " 设置语法折叠
```

```
set foldcolumn=0 " 设置折叠区域的宽度
setlocal foldlevel=1 " 设置折叠层数为 1
nnoremap <space> @=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')<CR>
```

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

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.

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

Learn to use GDB

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

RTFM

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

Logging in via SSH

Logging in via SSH

Since there is one terminal with small size in the VM, it is not very convenient to use. Therefore, we recommend you to log in the VM via SSH.

Creating a Host-only Network

First, you should create a host-only network for the VM. Do the followings:

1. Shut down the VM.
2. In the VirtualBox Manager, click `File` in the menu bar, then select `Preferences...`. This will invoke a window for preference configuration.
3. Select the `Network` category on the left.
4. Click a plus button on the right of the window. This will add a host-only network named "VirtualBox Host-Only Ethernet Adapter".
5. Click the button labeled with "Edit host-only network". This will invoke a dialog.
    - In the `Adapter` tab, do the following settings:
        - **IPv4 Address**: 192.168.56.1
        - **IPv4 Network Mask**: 255.255.255.0
    - In the `DHCP Server` tab, do the following settings:
    - Check `Enable Server`
        - **Server Address**: 192.168.56.100
        - **Server Mask**: 255.255.255.0
        - **Lower Address Bound**: 192.168.56.101
        - **Upper Address Bound**: 192.168.56.245
6. Click `OK` to go back to the VirtualBox Manager, choose the VM, then click the `Settings` button in the tool bar. This will invoke a window for VM configuration.
7. Select the `Network` category on the left.
8. Select the `Adapter 2` tab on the right.
9. Enable "Adapter 2", and modify the `Attached to` attribute of Adapter 2 to `Host-only Adapter`. The `Name` attribute should become `VirtualBox Host-Only Ethernet Adapter` by default.

Now launch the VM. Run

```
sudo ifconfig eth1
```

You should see some information about the network interface "eth1". If you receive an error message like

```
eth1: error fetching interface information: Device not found
```

This is probably that you did not configure the host-only network correctly. Check yourself.

However, you can not see the IP address of eth1, this is because eth1 is still down. Again, you should modify the configuration file. Append the following line at the end of the file `/etc/network/interfaces`:

```
iface eth1 inet dhcp
```

After modifying the "interfaces" file, restart the network:

```
sudo /etc/init.d/networking restart
```

Run

```
sudo ifconfig eth1
```

again. You should see the IP address of eth1, such as 192.168.56.xxx. Remember this IP address, and you will use it later.

Host Configuration

Now keep the VM on and go back to the host. You should perform some configurations to access the VM via SSH. According to the type of your host operating system, you will perform different configuration.

For GNU/Linux and Mac users

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

```
ssh username@ip_addr
```

where `username` is your VM user name, `ip_addr` is the IP address of eth1 mentioned above. For example:

```
ssh ics@192.168.56.101
```

If you are prompted with

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

enter "yes". Then enter your VM user password. If everything is fine, you will access the VM via SSH successfully. To exit SSH, just type

```
exit
```

in terminal.

For Windows users

Windows has no build-in `ssh` tool, and you have to download one manually. Download the **latest release version** of `putty.exe` here. Run `putty.exe`, and you will see a dialog is invoked. In the input box labeled with `Host Name (or IP address)`, enter the IP address of eth1 mentioned above. Leave other settings default, then click `Open` button. Enter your VM user name and password. If everything is fine, you will access the VM via SSH successfully. To exit SSH, just type

```
exit
```

in terminal.

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 (that is why we recommend you to log in via SSH). To install `tmux`, just issue the following command:

```
apt-get install tmux
```

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

```
cd ~
```

New a file called `.tmux.conf`:

```
vim .tmux.conf
```

Append the following content to the file:

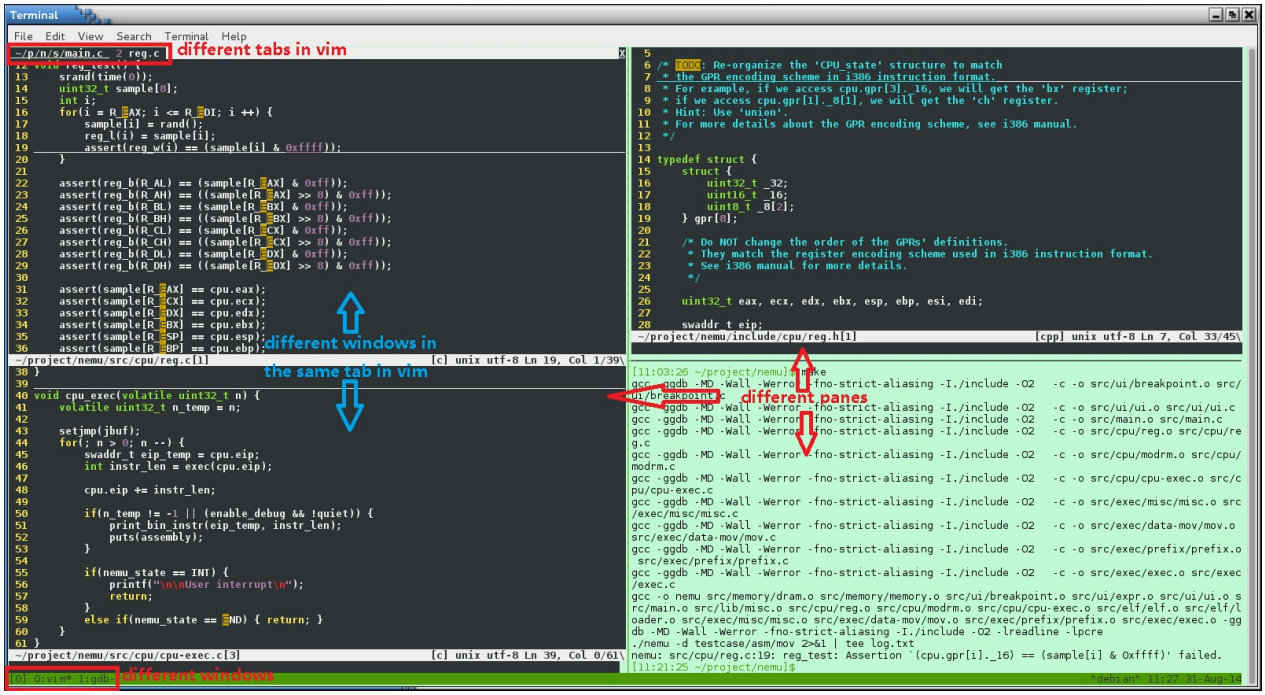
```
setw -g c0-change-trigger 100
setw -g c0-change-interval 250

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

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

If you use `ls` to list files, you will not see the `.tmux.conf` you just created. 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
```

You can scroll the content in a `tmux` terminal up and down. Also, using `tmux` with SSH, you can create multiple normal-size terminals within single screen. For how to use `tmux`, please search the Internet. The following picture shows a scene working with multiple terminals within single screen. Is it COOL?



Things behind scrolling

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

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

Transferring Files between host and VM

Transferring Files Between host and VM

Although VM is running on the host, they are isolated logically. This means the host can not directly access files in the VM, and vice versa. Therefore, a way is needed to transfer files from/to the VM.

Host Configuration

Now keep the VM on and go back to the host. According to the type of your host operating system, you will perform different configuration.

For GNU/Linux and Mac users

Thanks to Unix Philosophy, you can use one command to achieve the transferring.

To transfer files from host to VM, issue the following command in the host:

```
tar cj file | ssh username@ip_addr 'tar xvjf -'
```

where `file` is the file to transfer in the host, `username` and `ip_addr` are the same as mentioned in the Logging in via SSH section. You will be asked to enter your VM user password. If everything is fine, the `file` will be transferred to the home directory of the VM.

To transfer files from VM to host, issue the following command in the host:

```
ssh username@ip_addr 'tar cj file' | tar xvjf -
```

where `file` is the file to transfer in the VM, `username` and `ip_addr` are the same as mentioned in the Logging in via SSH section. You will be asked to enter your VM user password. If everything is fine, the `file` will be transferred to the home directory of the host.

For Windows users

Download the **latest release version** of `psftp.exe` here. Run `psftp.exe`, and connect to the VM by the following command:

```
open ip_addr
```

where `ip_addr` is the same as mentioned in the Logging in via SSH section. Then enter your VM user name as well as the password, as if logging in via `ssh`. We list some useful commands in `psftp.exe`:

```
cd     change your remote working directory
exit   finish your SFTP session
get    download a file from the server to your local machine
help   give help
lcd    change local working directory
lpwd   print local working directory
ls     list remote files
put    upload a file from your local machine to the server
```

`pwd` `print your remote working directory`

Under our situation, "local machine" stands for the host, "server" and "remote" stand for the VM. For more details about the command, refer to `help COMMAND`.

Have a try!

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

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

Acquiring Source Code for PAs

Acquiring Source Code for PAs

Getting Source Code

Go back to the home directory by

```
cd ~
```

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

不要使用root账户做实验!!!

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

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

Now acquire source code for PA by the following command:

```
git clone https://github.com/nju-ics/ics2015
```

A directory called `ics2015` will be created. This is the project directory for PAs. Details will be explained in PA1.

Compiling and Running NEMU

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

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

Another important thing to do is `git` configuration. Issue the following commands:

```
git config --global user.name "141220000-Zhang San"    # your stude
git config --global user.email "zhangsan@foo.com"    # your email
git config --global core.editor vim            # your favorite edit
git config --global color.ui true
```

You should configure `git` with your student ID, name, and email. This finishes `git` configuration.

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

```
make
```

If nothing goes wrong, NEMU will be compiled successfully.

What happened?

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

To perform a fresh compilation, type

```
make clean
```

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

To run NEMU, type

```
make run
```

However, you will see an error message:

```
nemu: nemu/src/cpu/reg.c:21: reg_test: Assertion `(cpu.gpr[check_r
```

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

To debug NEMU with gdb, type

```
make gdb
```

Development Tracing

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

```
git log
```

If you see something like

```
commit 4072d39e5b6c6b6837077f2d673cb0b5014e6ef9
Author: tracer-ics2015 <tracer@njuics.org>
Date:   Sun Jul 26 14:30:31 2015 +0800

    > compile NEMU
    141220000
    user
    Linux debian 3.16.0-4-686-pae #1 SMP Debian 3.16.7-3 i686 GNU/
     14:30:31 up  3:44,  2 users,  load average: 0.28, 0.09, 0.07
```

```
    3860572d5cc66412bf85332837c381c5c8c1009f
```

this means the change is traced successfully.

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

```
fatal: Unable to create '/home/user/ics2015/.git/index.lock': File

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

Try to clean the compilation result and compile again:

```
make clean
make
```

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

开发跟踪

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

- 请你不定期查看自己的git log, 检查是否与自己的开发过程相符.
- 提交往届代码将被视为没有提交.
- 不要把你的代码上传到公开的地方.
- 总是在工程目录下进行开发, 不要在其它地方进行开发, 然后一次性将代码复制到工程目录下, 这样`git`将不能正确记录你的开发过程.
- 不要修改`Makefile`中与开发跟踪相关的内容.

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

Local Commit

Although the development tracing system will trace the change of your code after every successful compilation, the trace record is not suitable for your development. This is because the code is still buggy at most of the time. Also, it is not easy for you to identify those bug-free traces. Therefore, you should trace your bug-free code manually. But before continuing, please read this `git` tutorial to learn some basics of `git`.

When you want to commit the change, type

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

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

by

```
git log
```

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

Submission

Finally, you should submit your project to the submission website. First, you should modify the `STU_ID` variable in `config/Makefile.git`:

```
STU_ID=141220000            # your student ID
```

To submit PA0, put your report file (ONLY `.pdf` file is accepted) under the project directory. Then issue

```
make submit
```

This command does 2 things:

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

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

RTFSC and Enjoy

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

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

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

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

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

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

Reminder

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

PA1 - 洞察世界的视点: 简易调试器

PA1 - 洞察世界的视点: 简易调试器

世界诞生的故事 - 第一章

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

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 30小时

截止时间: 本次实验的阶段性安排如下:

- 阶段1: 实现单步执行, 打印寄存器状态, 扫描内存
- 阶段2: 实现调试功能的表达式求值
- 最后阶段: 实现所有要求, 提交完整的实验报告
- 具体时间见Elearning

提交说明:

- 为了尽可能避免拖延症影响实验进度, 我们采用分阶段方式进行提交, 强迫大家每周都将实验进度往前推进. 在阶段性提交截止前, 你只需要提交你的工程, 并且实现的正确性不影响你的分数, 即我们允许你暂时提交有bug的实现. 在最后阶段中, 你需要提交你的工程和完整的实验报告, 同时我们也会检查实现的正确性.
- 如无特殊原因, 迟交的作业将损失30%的成绩(即使迟了1秒), 请大家合理分配时间.
- 但是, 如果你完全没有开始进行某阶段的实验内容, 请你不要进行相应的提交, 因为这会影响我们的工作. 一旦发现这种情况, 我们将会额外扣除你 `发现次数*10%` 的PA总成绩.

学术诚信: 如果你确实无法独立完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数.

下表说明了你可能采取的各种策略的收益:

	并非完全没有完成相应内容	完全没有完成相应内容	抄袭
按时提交	100%(获得完成部分的全部分数)	-发现次数*10%	0%, 并通知辅导员
未按时提交	70%(迟交惩罚)	-发现次数*10%	0%, 并通知辅导员
不提交	10%(学术诚信奖励)	10%(学术诚信奖励)	10%(学术诚信奖励)

总的来说, 最好的策略是: 做了就交, 没做就不要交.

提交地址: eLearning

提交格式: 把实验报告放到工程目录下之后, 使用 `make submit` 命令直接将整个工程打包即可. 请注意:

- 我们会清除中间结果, 使用原来的编译选项重新编译(包括 `-Wall` 和 `-Werror`), 若编译不通过, 本次实验你将得0分(编译错误是最容易排除的错误, 我们有理由认为你没有认真对待实验).
- 我们会使用脚本进行批量解压缩. `make submit` 命令会用你的学号来命名压缩包, 不要修改压缩包的命名. 另外为了防止出现编码问题, 压缩包中的所有文件名都不要包含中文.
- 我们只接受pdf格式, 命名只含学号的实验报告, 不符合格式的实验报告将视为没有提交报告. 例如 `141220000.pdf` 是符合格式要求的实验报告, 但 `141220000.docx` 和 `141220000张三实验报告.pdf` 不符合要求, 它们将不能被脚本识别出来.
- 如果你需要多次提交, 请先手动删除旧的提交记录(提交网站允许下载, 删除自己的提交记录)

git版本控制: 我们鼓励你使用git管理你的项目, 如果你提交的实验中包含均匀合理的, 你手动提交的git记录(不是开发跟踪系统自动提交的), 你将会获得本次实验代码分数20%的奖励(总得分不超过本次实验的上限). 这里有一个十分简单的git教程, 更多的git命令请查阅相关资料. 另外, 请你不定期查看自己的git log, 检查是否与自己的开发过程相符. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据.

实验报告内容: 你必须在实验报告中描述以下内容:

- 实验进度. 简单描述即可, 例如"我完成了所有内容", "我只完成了xxx". 缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验.
- 必答题.

你可以自由选择报告的其它内容. 你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- 你遇到的问题和对这些问题的思考
- 对讲义中蓝框思考题的看法
- 或者你的其它想法, 例如实验心得, 对提供帮助的同学的感谢等(如果你希望匿名吐槽, 请移步提交地址中的课程吐槽讨论区, 使用账号stu_ics登陆后进行吐槽)

认真描述实验心得和想法的报告将会获得分数的奖励; 蓝框题为选做, 完成了也不会得到分数的奖励, 但它们是经过精心准备的, 可以加深你对某些知识的理解和认识. 因此当你发现编写实验报告的时间所剩无几时, 你应该选择描述实验心得和想法. 如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求. 但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出来是凑字数的)

来让你的报告看起来十分丰富, 编写和阅读这样的报告毫无任何意义, 你也不会因此获得更多的分数, 同时还可能带来扣分的可能.

在开始愉快的PA之旅之前

在开始愉快的PA之旅之前

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

你小时候应该玩过红白机, 超级玛丽, 坦克大战, 魂斗罗... 它们的画面是否让你记忆犹新? (希望我们之间没有代沟...) 随着时代的发展, 你已经很难在市场上看到红白机的身影了. 当你正在为此感到苦恼的时候, 模拟器的横空出世唤醒了你心中尘封已久的童年回忆. 红白机模拟器可以为你模拟出红白机的所有功能, 有了它, 你就好像有了一个真正的红白机, 可以玩你最喜欢的红白机游戏(这里是jyy移植的一个小型项目LiteNES, 但由于debian虚拟机中缺少GUI, 因此要运行LiteNES是一件比较困难的事情). 你可以在如今这个红白机难以寻觅的时代, 再次回味你儿时的快乐时光, 这实在是太神奇了!

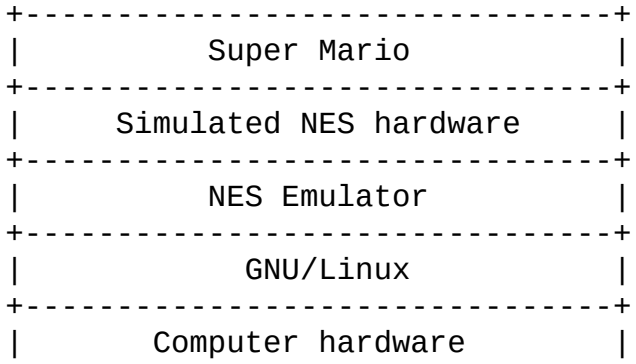
你被计算机强大的能力征服了, 你不禁思考, 这到底是怎么做到的? 你学习完程序设计基础课程, 但仍然找不到你想要的答案. 但你可以肯定的是, 红白机模拟器只是一个普通的程序, 因为你还是需要像运行Hello World程序那样运行它. 但同时你又觉得, 红白机模拟器又不像一个普通的程序, 它究竟是怎么模拟出一个红白机的世界, 让红白机游戏在这个世界中运行的呢?

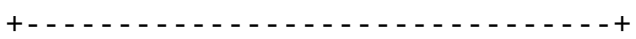
事实上, NEMU就是在做类似的事情! 它模拟了一个x86的世界(准确地说, 是x86的一个子集), 你可以在这个x86世界中执行程序. 换句话说, 你将要编写一个用来执行其它程序的程序! 为了更好地理解NEMU的功能, 下面将

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

这三种情况进行比较.

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





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

上图展示了"在GNU/Linux中通过NEMU执行Hello World程序"的情况. 在GNU/Linux看来, 运行在其上的NEMU和上面提到的Hello World程序一样, 都只不过是一个用户程序而已. 但NEMU的功能是负责模拟出一套x86硬件, 让程序可以在其上运行. 不过, 我们还需要先在模拟出的x86硬件之上运行一个微型操作系统, 之后才让Hello World程序在这个微型操作系统上面运行. 为了方便叙述, 我们将在NEMU中运行的程序称为"用户程序".

初识虚拟化

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

要虚拟出一个计算机系统并没有你想象中的那么困难. 我们可以把计算机看成由若干个硬件部件组成, 这些部件之间相互协助, 完成"运行程序"这件事情. 在NEMU中, 每一个硬件部件都由一个C语言的数据对象来模拟, 例如变量, 数组, 结构体等; 而对这些部件的操作则通过对相应数据对象的操作来模拟. 例如NEMU中使用结构体来模拟通用寄存器, 那么对这个结构体进行读写则相当于对通用寄存器进行读写.

这些数据对象之间相互协助的威力会让你感到吃惊! NEMU不仅仅能运行Hello World这样的小程序, 在PA的最后, 你将会在NEMU中运行仙剑奇侠传(很酷! %>_<%). 完成PA之后, 你在程序设计课上对程序的认识会被彻底颠覆, 你会觉得红白机模拟器不再是一件神奇的玩意儿, 甚至你会发现编写一个属于自己的红白机模拟器不再是遥不可及!

让我们来开始这段激动人心的旅程吧! 但请不要忘记:

- 机器永远是对的
- 未测试代码永远是错的
- RTFM

在开始愉快的PA之旅之前

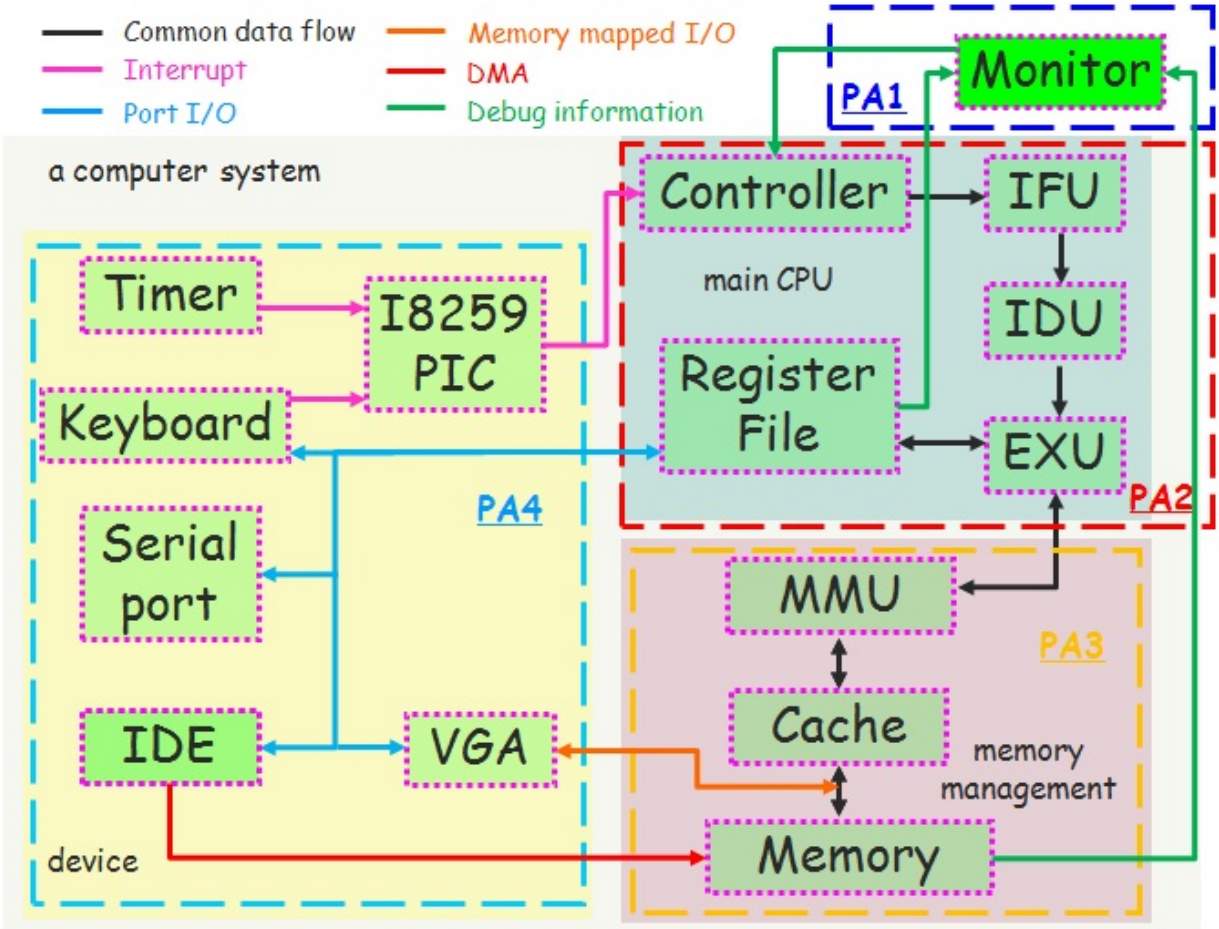
RTFSC

RTFSC

拿到框架代码之后, 第一件事就是RTFSC. 不过框架代码内容众多, 其中包含了很多在后续阶段中才使用的代码, 随着实验进度的推进, 我们会逐渐解释所有的代码. 因此在阅读代码的时候, 你只需要关心和当前进度相关的模块就可以了, 不要纠缠于和当前进度无关的代码, 否则将会给你的心灵带来不必要的恐惧.

```
ics2015
├── config              # 包含Makefile的一些配置
├── game              # 包含打字小游戏和仙剑奇侠传两个游戏
├── kernel              # 微型操作系统内核
├── lib-common          # 公用的库
├── Makefile          # 提供工程的构建, 运行, 测试, 打包等功能
├── nemu              # NEMU
├── testcase          # 测试用例
└── test.sh             # 测试脚本
```

目前我们只需要关心NEMU的内容, 其它内容会在将来进行介绍. 下图给出了NEMU的结构.



NEMU主要由4个模块构成: monitor, CPU, 存储管理, 设备, 它们依次作为4个PA关注的主题. 其中, CPU, 存储管理, 设备这3个模块共同组成一个虚拟的计算机系统, 程序可以在其上运行; monitor位于这个虚拟计算机系统之外, 主要用于监视这个虚拟计算机系统是否正确运行.

monitor虽然不属于虚拟计算机系统, 但对PA来说, 它是必要的. 它除了负责与GNU/Linux进行交互(例如读写文件)之外, 还带有调试器的功能, 为NEMU的调试提供了方便的途径. 缺少monitor模块, 对NEMU的调试将会变得十分困难.

代码中 nemu 目录下的源文件组织如下(部分目录下的文件并未列出):

```
nemu
├── include                          # 存放全局使用的头文件
│   ├── common.h                     # 公用的头文件
│   ├── cpu
│   │   ├── decode                   # 译码相关
│   │   ├── exec                     # 执行相关
│   │   └── reg.h                    # 寄存器结构体的定义
│   ├── debug.h                      # 一些方便调试用的宏
│   ├── device                       # 设备相关
│   ├── macro.h                      # 一些方便的宏定义
│   ├── memory
│   │   └── memory.h                 # 访问内存相关
│   ├── misc.h                       # 杂项
│   ├── monitor
│   │   ├── monitor.h
│   │   └── watchpoint.h             # 监视点相关
│   └── nemu.h
├── Makefile.part                    # 指示NEMU的编译和链接
└── src                              # 源文件
    ├── cpu
    │   ├── decode                   # 译码相关
    │   ├── exec                     # 执行相关
    │   └── reg.c                    # 寄存器相关
    ├── device                       # 设备相关
    ├── lib
    │   └── logo.c                   # "i386"的logo
    ├── main.c                       # 你知道的...
    ├── memory
    │   ├── burst.h
    │   ├── dram.c                   # DRAM工作方式的模拟
    │   └── memory.c                 # 访问内存的接口函数
    └── monitor
        ├── cpu-exec.c               # 指令执行的主循环
        ├── debug                    # 简易调试器相关
        │   ├── elf.c                # ELF文件格式的解析
        │   ├── expr.c               # 表达式求值的实现
        │   ├── ui.c                 # 用户界面相关
        │   └── watchpoint.c         # 监视点的实现
        └── monitor.c
```

为了给出一份可以运行的框架代码, 代码中完整实现了 mov 指令的功能(部分特殊的 mov 指令并未实现, 例如 mov %eax, %cr3), 并附带一个 mov 指令的用户程序(testcase/src/mov.S). 另外, 部分代码中会涉及一些硬件细节(例如 nemu/src/cpu/decode/modrm.c)和文件格式(例如 nemu/src/monitor/debug/elf.c). 在你第一次阅读代码的时候, 你需要尽快掌握NEMU的框架, 而不要纠缠于这些细节. 随着PA的进行, 你会反复回过头来探究这些细节.

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

需要多费口舌吗?

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

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

对vim的使用感到困难?

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

NEMU开始执行的时候, 会进行一些和monitor相关的初始化工作, 包括打开日志文件, 读入ELF文件的符号表和字符串表, 编译正则表达式, 初始化监视点结构池. 这些初始化工作你几乎一个也看不懂, 但不要紧, 因为你现在根本不必关心它们的细节, 因此可以继续阅读代码. 之后代码会对寄存器结构的实现进行测试, 测试通过后会调用 `restart()` 函数 (在 `nemu/src/monitor/monitor.c` 中定义), 它模拟了"计算机启动"的功能, 主要是进行一些和"计算机启动"相关的初始化工作, 包括

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

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

入口代码entry的引入其实是一种简化. 我们知道内存是一种RAM, 是一种易失性的存储介质, 这意味着计算机刚启动的时候, 内存中的数据都是无意义的; 而BIOS是固化在ROM中的, 它是一种非易失性的存储介质, BIOS中的内容不会因为断电而丢失. 因此在真实的计算机系统中, 计算机启动后首先会把控制权交给BIOS, BIOS经过一系列初始化工作之后, 再从磁盘中将有意义的程序读入内存中执行. 对这个过程的模拟需要了解很多超出本课程范围的细节, 我们在这里做了简化, 让monitor直接把一个有意义的程序entry读入到一个固定的内存位置 `0x100000`, 并把这个内存位置作为 `%eip` 的初值. 这时内存的布局如下:

```
0                                  0x100000
-------------------------------------------------------------------
|                  |                        |                    |
```

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

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

`restart()` 函数执行完毕后, NEMU会进入用户界面主循环 `ui_mainloop()` (在 `nemu/src/monitor/debug/ui.c` 中定义), 代码已经实现了几个简单的命令, 它们的功能和GDB是很类似的. 键入 `c` 之后, NEMU开始进入指令执行的主循环 `cpu_exec()` (在 `nemu/src/monitor/cpu-exec.c` 中定义).

`cpu_exec()` 模拟了CPU的工作方式: 不断执行指令. `exec()` 函数(在 `nemu/src/cpu/exec/exec.c` 中定义)的功能是让CPU执行一条指令. 已经执行的指令会输出到日志文件 `log.txt` 中, 你可以打开 `log.txt` 来查看它们.

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

温故而知新

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

NEMU将不断执行指令, 直到遇到以下情况之一, 才会退出指令执行的循环:

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

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

究竟要执行多久?

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

谁来指示程序的结束?

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

最后我们聊聊代码中一些值得注意的地方.

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

  代码中已经给出了使用这三个宏的例子, 如果你不知道如何使用它们, RTFSC.

- 访问模拟的内存
  - 在程序运行的过程中, 总是使用 `swaddr_read()` 和 `swaddr_write()` 访问模拟的内存. swaddr, lnaddr, hwaddr分别代表虚拟地址, 线性地址, 物理地址, 这些概念将在PA3中用到, 但从现在开始保持接口的一致性可以在将来避免一些不必要的麻烦.

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

理解框架代码

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

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

简易调试器

简易调试器

简易调试器是monitor的一项重要功能, 我们需要在monitor中实现一个具有如下功能的简易调试器(相关部分的代码在 `nemu/src/monitor/debug` 目录下), 如果你不清楚命令的格式和功能, 请参考如下表格:

命令	格式	使用举例	说明
帮助(1)	`help`	`help`	打印命令的帮助信息
继续运行(1)	`c`	`c`	继续运行被暂停的程序
退出(1)	`q`	`q`	退出NEMU
单步执行	`si [N]`	`si 10`	让程序单步执行 `N` 条指令后暂停执行, 当 `N` 没有给出时, 缺省为 `1`
打印程序状态	`info SUBCMD`	`info r` `info w`	打印寄存器状态 打印监视点信息
表达式求值	`p EXPR`	`p $eax + 1`	求出表达式 `EXPR` 的值, `EXPR` 支持的运算请见调试中的表达式求值小节
扫描内存(2)	`x N EXPR`	`x 10 $esp`	求出表达式 `EXPR` 的值, 将结果作为起始内存地址, 以十六进制形式输出连续的 `N` 个4字节
设置监视点	`w EXPR`	`w *0x2000`	当表达式 `EXPR` 的值发生变化时, 暂停程序执行
删除监视点	`d N`	`d 2`	删除序号为 `N` 的监视点
打印栈帧链(3)	`bt`	`bt`	打印栈帧链

备注:

- (1) 命令已实现
- (2) 与GDB相比, 我们在这里做了简化, 更改了命令的格式
- (3) 在PA2中实现

总有一天会找上门来的bug

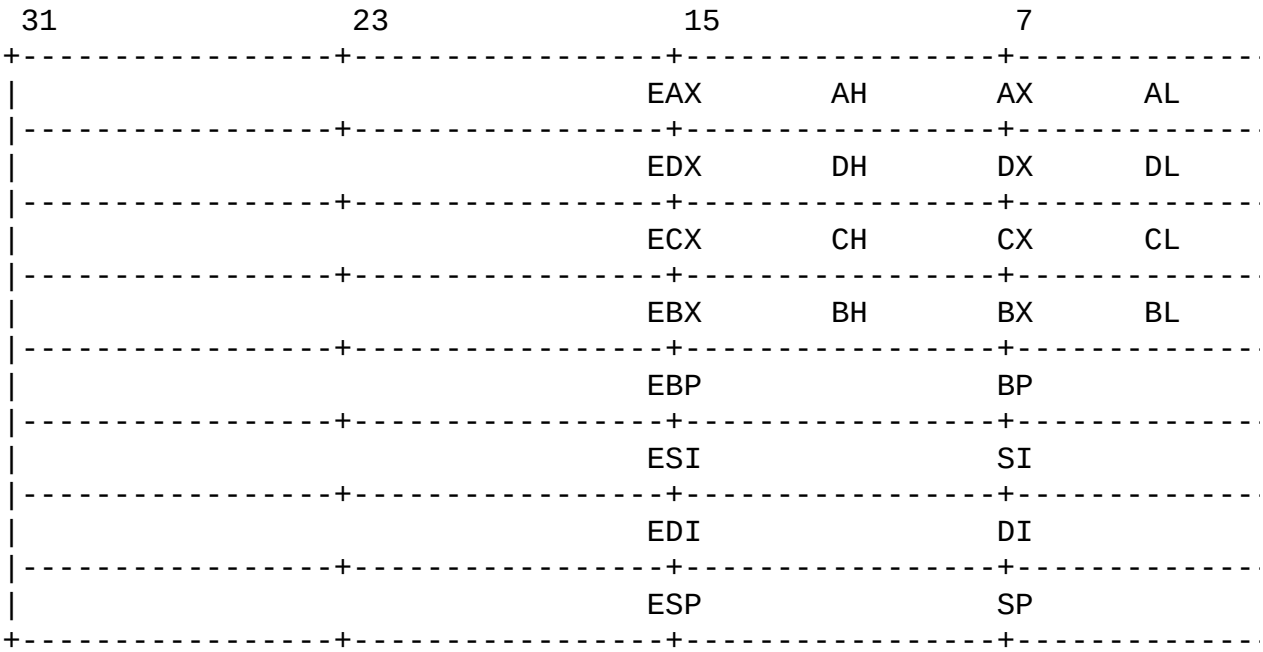
你需要在将来的PA中使用这些功能来帮助你进行NEMU的调试, 如果你的实现是有问题的, 将来你有可能会面临以下悲惨的结局: 你实现了某个新功能之后, 打算对它进行测试, 通过扫描内存的功能来查看一段内存, 发现输出并非预期结果. 你认为是刚才实现的新功能有问题, 于是对它进行调试. 经过了几天几夜的调试之后, 你泪流满面地发现, 原来是扫描内存的功能有bug!

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

基本功能

寄存器结构体

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

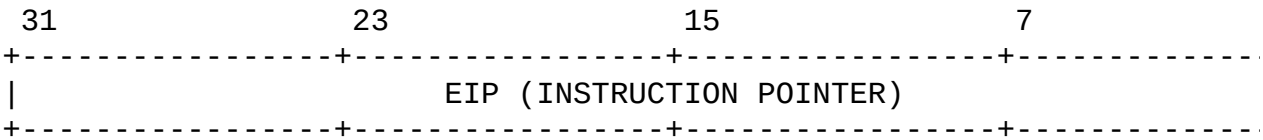


其中

- `EAX`, `EDX`, `ECX`, `EBX`, `EBP`, `ESI`, `EDI`, `ESP` 是32位寄存器;
- `AX`, `DX`, `CX`, `BX`, `BP`, `SI`, `DI`, `SP` 是16位寄存器;
- `AL`, `DL`, `CL`, `BL`, `AH`, `DH`, `CH`, `BH` 是8位寄存器.

但它们在物理上并不是相互独立的, 例如 `EAX` 的低16位是 `AX`, 而 `AX` 又分成 `AH` 和 `AL`. 这样的结构有时候在处理数据时能提供一些便利. 至于如何实现这样的结构, 当然是难不倒聪明的你啦!

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



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

实现正确的寄存器结构体

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

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

```
(nemu)
```

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

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

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

解析命令

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

```
man readline
```

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

```
man strtok
```

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

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

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

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

单步执行

单步执行的功能十分简单, 而且框架代码中已经给出了模拟CPU执行方式的函数, 你只要使

用相应的参数去调用它就可以了. 如果你仍然不知道要怎么做, RTFSC.

打印寄存器

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

扫描内存

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

```
x 10 0x100000
```

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

实现了扫描内存的功能之后, 你可以打印 `0x100000` 附近的内存, 你应该会看到程序的代码, 和用户程序的objdump结果进行对比(此时用户程序是 `mov`, 其dump结果在 `obj/testcase/mov.txt` 中), 看看你的实现是否正确.

实现单步执行, 打印寄存器, 扫描内存

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

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

温馨提示

PA1阶段1到此结束.

表达式求值

数学表达式求值

给你一个表达式的字符串

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

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

1. 首先识别出表达式中的单元
2. 根据表达式的归纳定义进行递归求值

词法分析

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

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

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

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

- 十进制整数
- `+` , `-` , `*` , `/`
- `(` , `)`
- 空格串(一个或多个空格)

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

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

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

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

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

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

系统设计的黄金法则 -- KISS法则

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

你已经学习过程序设计基础, 这意味着你已经学会写程序了, 但这并不意味着你可以顺利地完成PA, 因为在现实世界中, 我们需要的是可以运行的system, 而不是求阶乘的小程序. NEMU作为一个麻雀虽小, 五脏俱全的小型系统, 其代码量达到6000多行(不包括空行). 随着PA的进行, 代码量会越来越多, 各个模块之间的交互也越来越复杂, 工程的维护变得越来越困难, 一个很弱智的bug可能需要调好几天. 在这种情况下, 系统能跑起来才是王道, 跑不起来什么都是浮云, 追求面面俱到只会增加代码维护的难度.

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

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

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

实现算术表达式的词法分析

你需要完成以下的内容:

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

递归求值

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

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

的token表达式为

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

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

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

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

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

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

```
eval(p, q) {
    if(p > q) {
        /* Bad expression */
    }
    else if(p == q) {
        /* Single token.
         * For now this token should be a number.
         * Return the value of the number.
```

```
         */
    }
    else if(check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of paren
         * If that is the case, just throw away the parentheses.
         */
        return eval(p + 1, q - 1);
    }
    else {
        /* We should do more things here. */
    }
}
```

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

```
"(2 - 1)"                 // true
"(4 + 3 * (2 - 1))"         // true
"4 + 3 * (2 - 1)"         // false, the whole expression is not sur
"(4 + 3)) * ((2 - 1)"     // false, bad expression
"(4 + 3) * (2 - 1)"         // false, the leftmost '(' and the righ
```

至于怎么检查左右括号是否匹配, 就留给聪明的你来思考吧!

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

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


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


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

上面列出了3种可能的分裂, 注意到我们不可能在非运算符的token处进行分裂, 否则分裂得到的结果均不是合法的表达式. 根据dominant operator的定义, 我们很容易发现, 只有第

一种分裂才是正确的, 这其实也符合我们人工求值的过程: 先算 4 和 3 * (2 - 1), 最后把它们的结果相加. 第二种分裂违反了算术运算的优先级, 它会导致加法比乘法更早进行. 第三种分裂破坏了括号的平衡, 分裂得到的结果均不是合法的表达式.

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

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

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

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

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

        switch(op_type) {
            case '+': return val1 + val2;
            case '-': /* ... */
            case '*': /* ... */
            case '/': /* ... */
            default: assert(0);
        }
    }
}
```

实现算术表达式的递归求值

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

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

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

实现带有负数的算术表达式的求值(选做)

在上述实现中, 我们并没有考虑负数的问题, 例如

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

它们会被判定为不合法的表达式. 为了实现负数的功能, 你需要考虑两个问题:

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

你可以选择不实现负数的功能, 但你很快就要面临类似的问题了.

调试中的表达式求值

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

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

```
    | "*" <expr>                # 指针解引用
```

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

```
if(!make_token(e)) {
    *success = false;
    return 0;
}

/* TODO: Implement code to evaluate the expression. */

for(i = 0; i < nr_token; i ++) {
    if(tokens[i].type == '*' && (i == 0 || tokens[i - 1].type ==
        tokens[i].type = DEREF;
    }
}

return eval(?, ?);
```

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

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

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

实现更复杂的表达式求值

你需要实现上文BNF中列出的功能. 一个要注意的地方是词法分析中编写规则的顺序, 不正确的顺序会导致一个运算符被识别成两部分, 例如 != 被识别成 ! 和 = . 关于变量的功能, 它需要涉及符号表和字符串表的查找, 因此你会在PA2中实现它.

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

从表达式求值窥探编译器

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

程, 你是否有什么新的体会?

事实上, 词法分析也是编译器编译源代码的第一个步骤, 编译器也需要从你的源代码中识别出token, 这个功能也可以通过正则表达式来完成, 只不过token的类型更多, 更复杂而已. 这也解释了你为什么可以在源代码中插入任意数量的空白字符(包括空格, tab, 换行), 而不会影响程序的语义; 你也可以将所有源代码写到一行里面, 编译仍然能够通过.

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

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

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

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

温馨提示

PA1阶段2到此结束.

监视点

监视点

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

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

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

    /* TODO: Add more members if necessary */


} WP;
```

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

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

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

实现监视点池的管理

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

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

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

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

温故而知新(2)

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

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

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

实现监视点

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

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

断点

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

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

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

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

一点也不能长?

我们知道 `int3` 指令不带任何操作数, 操作码为1个字节, 因此指令的长度是1个字节. 这是必须的吗? 假设有一种IA-32体系结构的变种my-IA-32, 除了 `int3` 指令的长度变成了2个字节之外, 其余指令和IA-32相同. 在my-IA-32中, 文章中的断点机制还可以正常工作吗? 为什么?

"随心所欲"的断点

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

NEMU的前世今生

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

熟悉i386手册

熟悉i386手册

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

学会使用目录

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

尝试通过目录定位关注的问题

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

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

逐步细化搜索范围

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

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

不过有时候, 你会发现一个概念在手册中的多个地方都有提到. 这时你需要明确你要关心概念的哪个方面, 通常一个概念的某个方面只会在手册中的一个地方进行详细的介绍. 你需要在这多个地方中进行进一步的筛选, 但至少你已经过滤掉很多与这个概念无关的章节了. 筛选也是有策略的, 你不需要把多个地方的所有内容全部阅读一遍才能进行筛选, 小标题, 每段的第一句话, 图表的注解, 这些都可以帮助你很快地了解这一部分的内容大概在讲什么. 这不就是高中英语考试中的快速阅读吗? 对的, 就是这样. 如果你觉得目前还缺乏这方面的能力, 现在锻炼的好机会来了.

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

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

必答题

你需要在实验报告中回答下列问题:

- 查阅i386手册 理解了科学查阅手册的方法之后, 请你尝试在i386手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - EFLAGS寄存器中的CF位是什么意思?
  - ModR/M字节是什么?
  - mov指令的具体格式是怎么样的?
- shell命令 完成PA1的内容之后, nemu目录下的所有.c和.h文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 使用 `git checkout` 可以回到"过去", 具体使用方法请查阅 `man git-checkout`) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, nemu目录下的所有.c和.h文件总共有多少行代码?
- 使用man 打开工程目录下的 `Makefile` 文件, 你会在 `CFLAGS` 变量中看到gcc的一些编译选项. 请解释gcc中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror` ?

温馨提示

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

PA2 - 不停计算的机器: 指令系统

PA2 - 不停计算的机器: 指令系统

世界诞生的故事 - 第二章

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

提交要求(请认真阅读以下内容, 若有违反, 后果自负)

截止时间: 为了尽可能避免拖延症影响实验进度, 我们采用分阶段方式进行提交, 强迫大家每周都将实验进度往前推进. 在阶段性提交截止前, 你只需要提交你的工程, 并且实现的正确性不影响你的分数, 即我们允许你暂时提交有**bug**的实现. 在最后阶段中, 你需要提交你的工程和完整的实验报告, 同时我们也会检查实现的正确性. 本次实验的阶段性安排如下:

- 阶段1: 实现5个helper函数, 在NEMU中运行第一个C程序
- 阶段2: 实现更多的指令, 完善简易调试器
- 前半阶段最终提交： 实现前阶段要求，提交实验报告
- 阶段3: 实现更多的指令
- 阶段4: 实现loader
- 最后阶段: 实现所有要求, 提交完整的实验报告
- 提交时间见Elearning

提交说明:

- 为了尽可能避免拖延症影响实验进度, 我们采用分阶段方式进行提交, 强迫大家每周都将实验进度往前推进. 在阶段性提交截止前, 你只需要提交你的工程, 并且实现的正确性不影响你的分数, 即我们允许你暂时提交有bug的实现. 在最后阶段中, 你需要提交你的工程和完整的实验报告, 同时我们也会检查实现的正确性.
- 如无特殊原因, 迟交的作业将损失30%的成绩(即使迟了1秒), 请大家合理分配时间.
- 但是, 如果你完全没有开始进行某阶段的实验内容, 请你不要进行相应的提交, 因为这会影响我们的工作. 一旦发现这种情况, 我们将会额外扣除你 `发现次数*10%` 的PA总成绩.

学术诚信: 如果你确实无法独立完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数.

下表说明了你可能采取的各种策略的收益:

	并非完全没有完成相应内容	完全没有完成相应内容	抄袭
按时提交	100%(获得完成部分的全部分数)	-发现次数*10%	0%, 并通知辅导员
未按时提交	70%(迟交惩罚)	-发现次数*10%	0%, 并通知辅导员

不提交	10%(学术诚信奖励)	10%(学术诚信奖励)	10%(学术诚信奖励)

总的来说, 最好的策略是: 做了就交, 没做就不要交.

提交地址: eLearning

提交格式: 把实验报告放到工程目录下之后, 使用 `make submit` 命令直接将整个工程打包即可. 请注意:

- 我们会清除中间结果, 使用原来的编译选项重新编译(包括 `-Wall` 和 `-Werror`), 若编译不通过, 本次实验你将得0分(编译错误是最容易排除的错误, 我们有理由认为你没有认真对待实验).
- 我们会使用脚本进行批量解压缩. `make submit` 命令会用你的学号来命名压缩包, 不要修改压缩包的命名. 另外为了防止出现编码问题, 压缩包中的所有文件名都不要包含中文.
- 我们只接受pdf格式, 命名只含学号的实验报告, 不符合格式的实验报告将视为没有提交报告. 例如 `141220000.pdf` 是符合格式要求的实验报告, 但 `141220000.docx` 和 `141220000张三实验报告.pdf` 不符合要求, 它们将不能被脚本识别出来.
- 如果你需要多次提交, 请先手动删除旧的提交记录(提交网站允许下载, 删除自己的提交记录)

git版本控制: 我们鼓励你使用git管理你的项目, 如果你提交的实验中包含均匀合理的, 你手动提交的git记录(不是开发跟踪系统自动提交的), 你将会获得本次实验代码分数20%的奖励(总得分不超过本次实验的上限). 这里有一个十分简单的git教程, 更多的git命令请查阅相关资料. 另外, 请你不定期查看自己的git log, 检查是否与自己的开发过程相符. git log是独立完成实验的最有力证据, 完成了实验内容却缺少合理的git log, 不仅会损失大量分数, 还会给抄袭判定提供最有力的证据.

实验报告内容: 你必须在实验报告中描述以下内容:

- 实验进度. 简单描述即可, 例如"我完成了所有内容", "我只完成了xxx". 缺少实验进度的描述, 或者描述与实际情况不符, 将被视为没有完成本次实验.
- 必答题.

你可以自由选择报告的其它内容. 你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- 你遇到的问题和对这些问题的思考
- 对讲义中蓝框思考题的看法
- 或者你的其它想法, 例如实验心得, 对提供帮助的同学的感谢等(如果你希望匿名吐槽, 请移步提交地址中的课程吐槽讨论区, 使用账号stu_ics登陆后进行吐槽)

认真描述实验心得和想法的报告将会获得分数的奖励; 蓝框题为选做, 完成了也不会得到分数的奖励, 但它们是经过精心准备的, 可以加深你对某些知识的理解和认识. 因此当你发现编写实验报告的时间所剩无几时, 你应该选择描述实验心得和想法. 如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求. 但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出来是凑字数的)

来让你的报告看起来十分丰富, 编写和阅读这样的报告毫无任何意义, 你也不会因此获得更多的分数, 同时还可能带来扣分的可能.

x86指令系统简介

x86指令系统简介

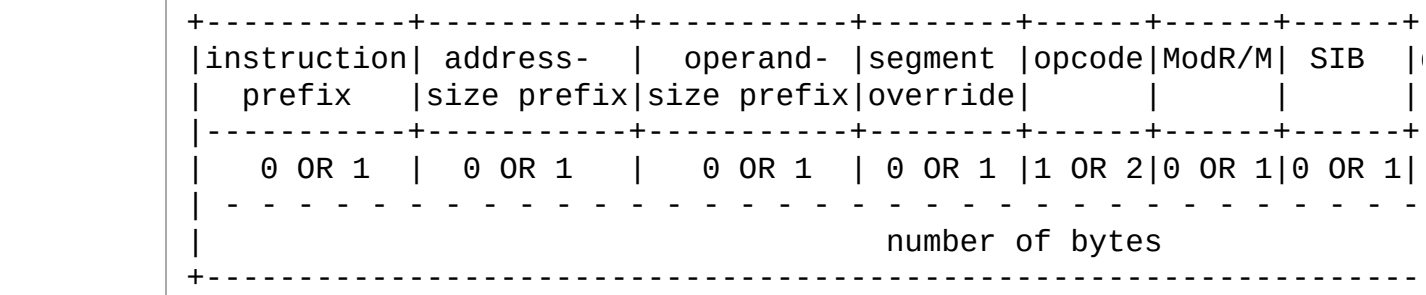
PA2的任务是在NEMU中实现x86指令系统(的子集), 你不可避免地需要了解x86指令系统的细节. i386手册有一章专门列出了所有指令的细节, 你需要在完成PA2的过程中反复阅读这一章的内容, 附录中的opcode map也很有用. 在这一小节中, 我们对x86指令系统作一些简单的梳理. 当你对x86指令系统有任何疑惑时, 请查阅i386手册, 关于指令系统的一切细节都在里面

i386手册勘误

我们在这个页面列出目前找到的错误, 如果你在做实验的过程中也发现了新的错误, 请帮助我们更新勘误信息.

指令格式

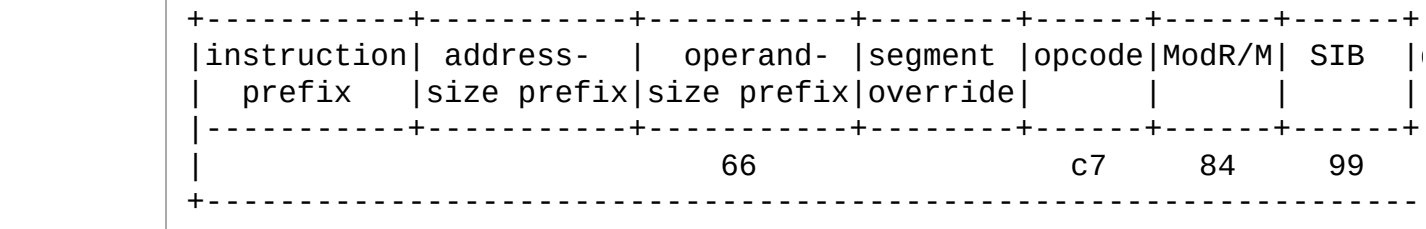
x86指令的一般格式如下:



除了opcode(操作码)必定出现之外, 其余组成部分可能不出现, 而对于某些组成部分, 其长度并不是固定的. 但给定一条具体指令的二进制形式, 其组成部分的划分是有办法确定的, 不会产生歧义(即把一串比特串看成指令的时候, 不会出现两种不同的解释). 例如对于以下指令:

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

其组成部分的划分如下:



凭什么 `0x84` 要被解释成 `ModR/M` 字节呢? 这是由 opcode 决定的, opcode 决定了这是什么指令的什么形式, 同时也决定了 opcode 之后的比特串如何解释. 如果你要问是谁来决定 opcode , 那你就得去问Intel了.

在我们的PA中, `address-size prefix` 和 `segment override prefix` 都不会用到, 因此NEMU也不需要实现这两者的功能.

编码的艺术

对于以下5个集合:

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

它们是两两不相交的. 这是必须的吗? 这背后反映了怎样的隐情?

另外我们在这里先给出 ModR/M 字节和 SIB 字节的格式, 它们是用来确定指令的操作数的, 详细的功能会在将来进行描述:

```
ModR/M byte
7     6     5     4     3     2     1     0
+--------+-------------+-------------+
|  mod   | reg/opcode  |     r/m     |
+--------+-------------+-------------+


SIB (scale index base) byte
7     6     5     4     3     2     1     0
+--------+-------------+-------------+
|   ss   |    index    |    base     |
+--------+-------------+-------------+
```

RISC - 与CISC平行的另一个世界

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

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

指令集细节

要实现一条指令, 首先你需要知道这条指令的格式和功能, 格式决定如何解释, 功能决定如何执行. 而这些信息都在instruction set page中, 因此你务必知道如何阅读它们. 我们以 mov 指令的opcode表为例来说明如何阅读:

```
      Opcode        Instruction       Clocks        Description

< 1> 88 /r         MOV r/m8,r8       2/2           Move byte regis
< 2> 89 /r         MOV r/m16,r16     2/2           Move word regis
```

```
< 3> 89 /r          MOV r/m32,r32     2/2             Move dword regi
< 4> 8A /r          MOV r8,r/m8       2/4             Move r/m byte t
< 5> 8B /r          MOV r16,r/m16     2/4             Move r/m word t
< 6> 8B /r          MOV r32,r/m32     2/4             Move r/m dword
< 7> 8C /r          MOV r/m16,Sreg    2/2             Move segment re
< 8> 8D /r          MOV Sreg,r/m16    2/5,pm=18/19    Move r/m word t
< 9> A0             MOV AL,moffs8     4               Move byte at (s
<10> A1             MOV AX,moffs16    4               Move word at (s
<11> A1             MOV EAX,moffs32   4               Move dword at (
<12> A2             MOV moffs8,AL     2               Move AL to (seg
<13> A3             MOV moffs16,AX    2               Move AX to (seg
<14> A3             MOV moffs32,EAX   2               Move EAX to (se
<15> B0 + rb ib     MOV r8,imm8       2               Move immediate
<16> B8 + rw iw     MOV r16,imm16     2               Move immediate
<17> B8 + rd id     MOV r32,imm32     2               Move immediate
<18> C6 /0 ib (*)   MOV r/m8,imm8     2/2             Move immediate
<19> C7 /0 iw (*)   MOV r/m16,imm16   2/2             Move immediate
<20> C7 /0 id (*)   MOV r/m32,imm32   2/2             Move immediate

-----------------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset relat
to the segment base. The 8, 16, and 32 refer to the size of the d
address-size attribute of the instruction determines the size of
offset, either 16 or 32 bits.
-----------------------------------------------------------------------
```

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.

上表中的每一行给出了 mov 指令的不同形式, 每一列分别表示这种形式的opcode, 汇编语言格式, 执行所需周期, 以及功能描述. 由于NEMU关注的是功能的模拟, 因此 Clocks 一列不必关心. 另外需要注意的是, i386手册中的汇编语言格式都是Intel格式, 而objdump的默认格式是AT&T格式, 两者的源操作数和目的操作数位置不一样, 千万不要把它们混淆了! 否则你将会陷入难以理解的bug中.

首先我们来看 mov 指令的第一种形式:

```
     Opcode       Instruction       Clocks        Description
< 1> 88 /r        MOV r/m8,r8       2/2           Move byte regis
```

- 从功能描述可以看出, 它的作用是"将一个8位寄存器中的数据传送到8位的寄存器或者内存中", 其中 r/m 表示"寄存器或内存".
- Opcode一列中的编码都是用十六进制表示, 88 表示这条指令的opcode的首字节是 0x88 , /r 表示后面跟一个 ModR/M 字节, 并且 ModR/M 字节中的 reg/opcode 域解释成通用寄存器的编码, 用来表示其中一个操作数. 通用寄存器的编码如下:

二进制编码	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
8位寄存器	AL	CL	DL	BL	AH	CH	DH	BH
16位寄存器	AX	CX	DX	BX	SP	BP	SI	DI
32位寄存器	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

- Instruction一列中, r/m8 表示操作数是8位的寄存器或内存, r8 表示操作数是8位寄存器, 按照Intel格式的汇编语法来解释, 表示将8位寄存器(r8)中的数据传送到8位寄存器或内存(r/m8)中, 这和功能描述是一致的. 至于 r/m 表示的究竟是寄存器还是内存, 这是由 ModR/M 字节的 mod 域决定的: 当 mod 域取值为 3 的时候, r/m 表示的是寄存器; 否则 r/m 表示的是内存. 表示内存的时候又有多种寻址方式, 具体信息参考i386手册中的表格17-3.

看明白了上面的第一种形式之后, 接下来的两种形式也就不难看懂了:

```
< 2> 89 /r          MOV r/m16,r16      2/2             Move word regis
< 3> 89 /r          MOV r/m32,r32      2/2             Move dword regi
```

但你会发现, 这两种形式的 Opcode 都是一样的, 难道不会出现歧义吗? 不用着急, 还记得指令一般格式中的 operand-size prefix 吗? x86正是通过它来区分上面这两种形式的. operand-size prefix 的编码是 0x66 , 作用是指示当前指令需要改变操作数的长度. 在IA-32中, 通常来说, 如果这个前缀没有出现, 操作数长度默认是32位; 当这个前缀出现的时候, 操作数长度就要改变成16位(也有相反的情况, 这个前缀的出现使得操作数长度从16位变成32位, 但这种情况在IA-32中极少出现). 换句话说, 如果把一个开头为 89 ... 的比特串解释成指令, 它就应该被解释成 MOV r/m32,r32 的形式; 如果比特串的开头是 66 89... , 它就应该被解释成 MOV r/m16,r16 .

操作数长度前缀的由来

i386是从8086发展过来的. 8086是一个16位的时代, 很多指令的16位版本在当时就已经实现好了. 要踏进32位的新时代, 兼容就成了需要仔细考量的一个重要因素.

一种最直接的方法是让32位的指令使用新的操作码, 但这样1字节的操作码很快就会用光. 假设8086已经实现了200条16位版本的指令形式, 为了加入这些指令形式的32位版本, 这种做法需要使用另外200个新的操作码, 使得大部分指令形式的操作码需要使用两个字节来表示, 这样直接导致了32位的程序代码会变长. 现在你可能会觉得每条指令的长度增加一个字节也没什么大不了, 但在i386诞生的那个遥远的时代(你可以在i386手册的封面看到那个时代), 内存是一种十分珍贵的资源, 因此这种使用新操作码的方法并不是一种明智的选择.

Intel想到的解决办法就是引入操作数长度前缀, 来达到操作码复用的效果. 当处理器工作在16位模式(实模式)下的时候, 默认执行16位版本的指令; 当处理器工作在32位模式(保护模式)下的时候, 默认执行32位版本的指令. 当某些需要的时候, 才通过操作数长度前缀来指示操作数的长度. 这种方法最大的好处就是不需要引入额外的操作码, 从而也不会明显地使得程序代码变长. 虽然我们可以使用很简单的方法来模拟这个功能, 但在真实的芯片设计过程中, CPU的译码部件需要增加很多逻辑才能实现.

到现在为止, <4>-<6>三种形式你也明白了:

```
< 4> 8A /r          MOV r8,r/m8        2/4             Move r/m byte
< 5> 8B /r          MOV r16,r/m16      2/4             Move r/m word
< 6> 8B /r          MOV r32,r/m32      2/4             Move r/m dword
```

<7>和<8>两种形式的mov指令涉及到段寄存器:

```
< 7> 8C /r          MOV r/m16,Sreg     2/2             Move segment r
< 8> 8D /r          MOV Sreg,r/m16     2/5,pm=18/19    Move r/m word
```

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

<9>-<14>这6种形式涉及到一种新的操作数记号 moffs :

```
< 9> A0           MOV AL,moffs8     4             Move byte at (
<10> A1           MOV AX,moffs16    4             Move word at (
<11> A1           MOV EAX,moffs32   4             Move dword at
<12> A2           MOV moffs8,AL     2             Move AL to (se
<13> A3           MOV moffs16,AX    2             Move AX to (se
<14> A3           MOV moffs32,EAX   2             Move EAX to (s
-------------------------------------------------------------
NOTES:
moffs8, moffs16, and moffs32 all consist of a simple offset rela
to the segment base. The 8, 16, and 32 refer to the size of the
address-size attribute of the instruction determines the size of
offset, either 16 or 32 bits.
-------------------------------------------------------------
```

NOTES中给出了 moffs 的含义, 它用来表示段内偏移量, 但NEMU现在还没有"段"的概念, 目前可以理解成"相对于物理地址0处的偏移量". 这6种形式是 mov 指令的特殊形式, 它们可以不通过 ModR/M 字节, 让 displacement 直接跟在 opcode 后面, 同时让 displacement 来指示一个内存地址.

<15>-<17>三种形式涉及到两种新的操作数记号:

```
<15> B0 + rb ib   MOV r8,imm8       2             Move immediate
<16> B8 + rw iw   MOV r16,imm16     2             Move immediate
<17> B8 + rd id   MOV r32,imm32     2             Move immediate
```

其中:

- +rb , +rw , +rd 分别表示8位, 16位, 32位通用寄存器的编码. 和 ModR/M 中的 reg 域不一样的是, 这三种记号表示直接将通用寄存器的编号按数值加到 opcode 中(也可以看成通用寄存器的编码嵌在 opcode 的低三位), 因此识别指令的时候可以通过 opcode 的低三位确定一个寄存器操作数.
- ib , iw , id 分别表示8位, 16位, 32位立即数

最后3种形式涉及到一种新的操作码记号 /digit , 其中 digit 为 0 ~ 7 中的一个数字:

```
<18> C6 /0 ib (*) MOV r/m8,imm8     2/2           Move immediate
<19> C7 /0 iw (*) MOV r/m16,imm16   2/2           Move immediate
<20> C7 /0 id (*) MOV r/m32,imm32   2/2           Move immediate
```

注:
标记了(*)的指令形式的Opcode相对于i386手册有改动, 具体情况见下文的描述.

上述形式中的 /0 表示一个 ModR/M 字节, 并且 ModR/M 字节中的 reg/opcode 域解释成扩展opcode, 其值取 0 . 对于含有 /digit 记号的指令形式, 需要通过指令本身的 opcode 和 ModR/M 中的扩展opcode共同决定指令的形式, 例如 80 /0 表示 add 指令的一种形式, 而 80 /5 则表示 sub 指令的一种形式, 只看 opcode 的首字节 80 不能区分它们.

注: 在i386手册中, 这3种形式的 mov 指令并没有 /0 的记号, 在这里加入 /0 纯粹是为了说明 /digit 记号的意思. 但同时这条指令在i386中也比较特殊, 它需要使用 ModR/M 字节来表示一个寄存器或内存的操作数, 但 ModR/M 字节中的 reg/opcode 域却没有用到(一般情况下, ModR/M 字节中的 reg/opcode 域要么表示一个寄存器操作数, 要么作为扩展opcode), i386手册也没有对此进行特别的说明, 直觉上的解释就是"无论 ModR/M 字节中的 reg/opcode 域是什么值, 都可以被CPU识别成这种形式的 mov 指令". x86是商业CPU, 我们无法从电路级实现来考证这一解释, 但对编译器生成代码来说, 这条指令中的 reg/opcode 域总得有个确定的值, 因此编译器一般会把这个值设成 0 . 在NEMU的框架代码中, 对这3种形式的 mov 指令的实现和i386手册中给出 Opcode 保持一致, 忽略 ModR/M 字节中的 reg/opcode 域, 没有判断其值是否为 0 . 如果你不能理解这段话在说什么, 你可以忽略它, 因为这并不会影响实验的进行.

到此为止, 你已经学会了如何阅读大部分的指令集细节了. 需要说明的是, 这里举的 mov 指令的例子并没有完全覆盖i386手册中指令集细节的所有记号, 若有疑问, 请参考i386手册.

除了opcode表之外, Operation , Description 和 Flags Affected 这三个条目都要仔细阅读, 这样你才能完整地掌握一条指令的功能. Exceptions 条目涉及到执行这条指令可能产生的异常, 由于NEMU不打算实现异常处理的机制, 你可以不用关心这一条目.

RTFSC(2)

RTFSC(2)

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

在PA1中, 你已经阅读了monitor部分的框架代码, 了解了NEMU执行的粗略框架. 但现在, 你需要进一步弄明白, 一条指令是怎么在NEMU中执行的, 即我们需要进一步探究 `exec()` 函数中的细节. 为了说明这个过程, 我们举了两个 `mov` 指令的例子, 它们是框架代码自带的用户程序mov(`testcase/src/mov.S`)中的两条指令(mov的反汇编结果在 `obj/testcase/mov.txt` 中):

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

helper函数命名约定

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

```
指令_形式_操作数后缀
```

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

一个特殊的操作数后缀是 `v`, 表示variant, 意味着光看操作码的首字节, 操作数长度还不能确定, 可能是16位或者32位, 需要通过 `ops_decoded.is_data_size_16` 成员变量来决定. 其实这种helper函数做的事情, 就是在根据指令是否出现 `operand-size prefix` 来确定操作数长度, 从而决定最终的指令形式, 调用最终的helper函数来模拟指令的执行.

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

简单**mov**指令的执行

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

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

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

对于大部分指令来说, 执行它们都可以抽象成取指-译码-执行的指令周期. 为了使描述更加清晰,

我们借助指令周期中的一些概念来说明指令执行的过程.

取指**(instruction fetch, IF)**

要执行一条指令, 首先要拿到这条指令. 指令究竟在哪里呢? 还记得冯诺依曼体系结构的核心思想吗? 那就是"存储程序, 程序控制". 你以前听说这两句话的时候可能没有什么概念, 现在是实践的时候了. 这两句话告诉你, 指令在存储器中, 由PC(program counter, 在x86中就是 `%eip`)指出当前指令的位置. 事实上, `%eip` 就是一个指针! 在计算机世界中, 指针的概念无处不在, 如果你觉得对指针的概念还不是很熟悉, 就要赶紧复习指针这门必修课啦. 取指令要做的事情自然就是将 `%eip` 指向的指令从内存读入到CPU中. 在NEMU中, 有一个函数 `instr_fetch()` (在 `nemu/include/cpu/helper.h` 中定义)专门负责取指令的工作.

译码**(instruction decode, ID)**

在取指阶段, CPU拿到的是指令的比特串. 如果想知道这串比特串究竟代表什么意思, 就要进行译码的工作了. 我们可以把译码的工作作进一步的细化: 首先要决定具体是哪一条指令的哪一种形式, 这主要是通过查看指令的 `opcode` 来决定的. 对于大多数指令来说, CPU只要看指令的第一个字节就可以知道具体指令的形式了. 在NEMU中, `exec()` 函数首先通过 `instr_fetch()` 取出指令的第一个字节, 然后根据取到的这个字节查看 `opcode_table` , 得到指令的helper函数, 从而调用这个helper函数来继续模拟这条指令的执行. 以 `mov $0x8000, %ecx` 指令为例, 首先通过 `instr_fetch()` 取得这条指令的第一个字节 `0xb9` , 然后根据这个字节来索引 `opcode_table` , 找到了一个名为 `mov_i2r_v` 的helper函数, 这样就可以确定取到的是一条 `mov` 指令, 它的形式是将立即数移入寄存器(move immediate to register).

事实上, 一个字节最多只能区分256种不同的指令形式, 当指令形式的数目大于256时, 我们需要使用另外的方法来识别它们. x86中有主要有两种方法来解决这个问题(在PA2中你都会遇到这两种情况):

- 一种方法是使用转义码(escape code), x86中有一个2字节转义码 `0x0f` , 当指令 `opcode` 的第一个字节是 `0x0f` 时, 表示需要再读入一个字节才能决定具体的指令形式(部分条件跳转指令就属于这种情况). 后来随着各种SSE指令集的加入, 使用2字节转义码也不足以表示所有的指令形式了, x86在2字节转义码的基础上又引入了3字节转义码, 当指令 `opcode` 的前两个字节是 `0x0f` 和 `0x38` 时, 表示需要再读入一个字节才能决定具体的指令形式.
- 另一种方法是使用 `ModR/M` 字节中的扩展opcode域来对 `opcode` 的长度进行扩充. 有些时候, 读入一个字节也还不能完全确定具体的指令形式, 这时候需要读入紧跟在 `opcode` 后面的 `ModR/M` 字节, 把其中的 `reg/opcode` 域当做 `opcode` 的一部分来解释, 才能决定具体的指令形式. x86把这些指令划分成不同的指令组(instruction group), 在同一个指令组中的指令需要通过 `ModR/M` 字节中的扩展opcode域来区分.

决定了具体的指令形式之后, 译码工作还需要决定指令的操作数. 事实上, 在确定了指令的 `opcode` 之后, 指令形式就能确定下来了, CPU可以根据指令形式来确定具体的操作数. 我们还是以 `mov $0x8000, %ecx` 来说明这个过程, 但在这之前, 我们需要作一些额外的说明. 在上文的描述中, 我们通过这条指令的第一个字节 `0xb9` 找到了 `mov_i2r_v()` 的helper函数, 这个helper函数的定义在 `nemu/src/cpu/exec/data-mov/mov.c` 中:

```
make_helper_v(mov_i2r)
```

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

```
#define make_helper_v(name) \
    make_helper(concat(name, _v)) { \
```

```
        return (ops_decoded.is_data_size_16 ? concat(name, _w) : con
    }
```

查看预处理结果

框架代码中使用了很多宏定义, 如果你对宏定义不太熟悉, 可能会对阅读框架代码带来困难. 我们准备了生成 nemu/src/cpu/decode 目录和 nemu/src/cpu/exec 目录下预处理结果的命令, 键入

```
make cpp
```

可以生成这两个目录下 .c 文件的预处理结果, 它们可以帮助你阅读框架代码, 调试与宏相关的错误. 键入

```
make clean-cpp
```

可以移除这些预处理结果.

进行宏展开之后, mov_i2r_v() 的函数体如下:

```
int mov_i2r_v(swaddr_t eip) {
    return (ops_decoded.is_data_size_16 ? mov_i2r_w : mov_i2r_l) (e
}
```

它的作用是根据全局变量 ops_decoded (在 nemu/src/cpu/decode/decode.c 中定义)中的 is_data_size_16 成员变量来决定操作数的长度, 然后从复用 opcode 的两个 helper函数中选择一个进行调用. 全局变量 ops_decoded 用于存放一些译码的结果, 其中的 is_data_size_16 成员和指令中的 operand-size prefix 有关, 而且会经常用到, 框架代码把类似于 mov_i2r_v() 这样的功能抽象成一个宏 make_helper_v() , 方便代码的编写. 关于 is_data_size_16 成员的更多内容会在下文进行说明. 根据指令 mov $0x8000, %ecx 的功能, 它的操作数长度为4字节, 因此这里会调用 mov_i2r_l() 的 helper函数. mov_i2r_l() 的helper函数在 nemu/src/cpu/exec/data-mov/mov-template.h 中定义, 它的函数体是通过宏展开得到的, 在这里我们直接给出宏展开的结果, 关于宏的使用请阅读相应的框架代码:

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

其中 idex() 函数的原型为

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

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

对于 mov $0x8000, %ecx 指令来说, 确定操作数其实就是确定寄存器 %ecx 和立即数 $0x8000 . 在x86中, 通用寄存器都有自己的编号, mov_i2r 形式的指令把寄存器编号也放在指令的第一个字节里面, 我们可以通过位运算将寄存器编号抽取出来. 对于 mov_i2r 形式的指令来说, 立即数存放在指令的第二个字节, 可以很容易得到它. 然而很多指令都具有i2r的形式, 框架代码提供了几个函数(decode_i2r_l() 等), 专门用于进行对i2r形式的指令的译码工作. decode_i2r_l() 函数会把指令中的立即数信息和寄存器信息分别记录在全局

变量 ops_decoded 中的 src 成员和 dest 成员中, nemu/include/cpu/helper.h 中定义了两个宏 op_src 和 op_dest , 用于方便地访问这两个成员.

立即数背后的故事

在 decode_i_l() 函数中通过 instr_fetch() 函数获得指令中的立即数, 别看这里就这么一行代码, 其实背后隐藏着针对字节序的慎重考虑. 我们知道x86是小端机, 当你使用高级语言或者汇编语言写了一个32位常数 0x8000 的时候, 在生成的二进制代码中, 这个常数对应的字节序列如下(假设这个常数在内存中的起始地址是x):

```
x    x+1  x+2  x+3
+----+----+----+----+
| 00 | 80 | 00 | 00 |
+----+----+----+----+
```

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

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

这行代码会将 00 80 00 00 这个字节序列原封不动地从内存读入 imm 变量中, 主机的CPU会按照小端方式来解释这一字节序列, 于是会得到 0x8000 , 符合我们的预期结果.

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

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

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

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

执行**(execute, EX)**

译码阶段的工作完成之后, CPU就知道当前指令具体要做什么了, 执行阶段就是真正完成指令的工作. 对于 mov $0x8000, %ecx 指令来说, 执行阶段的工作就是把立即数 $0x8000 送到寄存器 %ecx 中. 由于 mov 指令的功能可以统一成"把源操作数的值传送到目标操作数中", 而译码阶段已经把操作数都准备好了, 所以只需要针对 mov 指令编写一个模拟执行过程的函数即可. 这个函数就是 do_mov_l() , 它是通过在 nemu/src/cpu/exec/data-mov/mov-template.h 中定义的 do_execute() 函数进行宏展开后得到的:

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

其中 write_operand_l() 函数会根据第一个参数中记录的类型的不同进行相应的写

操作, 包括写寄存器和写内存.

更新 **%eip**

执行完一条指令之后, CPU就要执行下一条指令. 在这之前, CPU需要更新 %eip 的值, 让 %eip 指向下一条指令的位置. 为此, 我们需要确定刚刚执行完的指令的长度. 在NEMU中, 指令的长度是通过helper函数的返回值进行传递的, 最终会传回到 cpu_exec() 函数中, 完成对 %eip 的更新.

复杂**mov**指令的执行

对于第二个例子 movw $0x1, -0x2000(%ecx,%ebx,4) , 执行这条执行还是分取指, 译码, 执行三个阶段.

首先是取指. 这条mov指令比较特殊, 它的第一个字节是 0x66 , 如果你查阅i386手册, 你会发现 0x66 是一个 operand-size prefix . 因为这个前缀的存在, 本例中的 mov 指令才能被CPU识别成 movw . NEMU使用 ops_decoded.is_data_size_16 成员变量来记录操作数长度前缀是否出现, 0x66 的helper函数 data_size() 实现了这个功能.

data_size() 函数对 ops_decoded.is_data_size_16 成员变量做了标识之后, 越过前缀重新调用 exec() 函数, 此时取得了真正的操作码 0xc7 , 通过查看 opcode_table 调用了helper函数 mov_i2rm_v() . 由于 ops_decoded.is_data_size_16 成员变量进行过标识, 在 mov_i2rm_v() 中将会调用 mov_i2rm_w() 的helper函数. 到此为止才识别出本例中的指令是一条 movw 指令.

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

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

这里使用 decode_i2rm_w() 函数来进行译码的工作, 阅读代码, 你会发现它最终会调用 read_ModR_M() 函数. 由于本例中的 mov 指令需要访问内存, 因此除了要识别出立即数之外, 还需要确定好要访问的内存地址. x86通过 ModR/M 字节来指示内存操作数, 支持各种灵活的寻址方式. 其中最一般的寻址格式是

displacement(R[base_reg], R[index_reg], scale_factor)

相应内存地址的计算方式为

addr = R[base_reg] + R[index_reg] * scale_factor + displacement

其它寻址格式都可以看作这种一般格式的特例, 例如

displacement(R[base_reg])

可以认为是在一般格式中取 R[index_reg] = 0, scale_factor = 1 的情况. 这样, 确定内存地址就是要确定 base_reg , index_reg , scale_factor 和 displacement 这4个值, 而它们的信息已经全部编码在 ModR/M 字节里面了.

我们以本例中的 movw $0x1, -0x2000(%ecx,%ebx,4) 说明如何识别出内存地址:

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

根据 mov_i2rm 的指令形式, 0xc7 是 opcode , 0x84 是 ModR/M 字节. 在i386手册中查阅表格17-3得知, 0x84 的编码表示在 ModR/M 字节后面还跟着一个 SIB 字节, 然后跟着一个32位的 displacement . 于是读出 SIB 字节, 发现是 0x99 . 在i386手册中查阅表格17-4得知, 0x99 的编码表示 base_reg = ECX, index_reg = EBX, scale_factor = 4 . 在 SIB 字节后面读出一个32位的 displacement , 发现是 00 e0 ff ff , 在小端存储方式下, 它被解释成 -0x2000 . 于是内存地址的计算方式为

addr = R[ECX] + R[EBX] * 4 - 0x2000

框架代码已经实现了 load_addr() 函数和 read_ModR_M() 函数(在 nemu/src/cpu/decode/modrm.c 中定义), 它们的函数原型为

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

它们将变量 eip 所指向的内存位置解释成 ModR/M 字节, 根据上述方法对 ModR/M 字节和 SIB 字节进行译码, 把译码结果存放到参数 rm 和 reg 指向的变量中, 同时返回这一译码过程所需的字节数. 在上面的例子中, 为了计算出内存地址, 用到了 ModR/M 字节, SIB 字节和32位的 displacement , 总共6个字节, 所以 read_ModR_M() 返回6. 虽然i386手册中的表格17-3和表格17-4内容比较多, 仔细看会发现, ModR/M 字节和 SIB 字节的编码都是有规律可循的, 所以 load_addr() 函数可以很简单地识别出计算内存地址所需要的4个要素(当然也处理了一些特殊情况). 不过你现在可以不必关心其中的细节, 框架代码已经为你封装好这些细节, 并且提供了各种用于译码的接口函数.

本例中的执行阶段就是要把立即数写入到相应的内存位置, 这是通过 do_mov_w() 函数实现的. 执行结束后返回指令的长度, 最终在 cpu_exec() 函数中更新 %eip .

用宏实现的模板功能

另外, 直接使用模板的做法并不是最佳的, 你会发现不同指令的模板之间还是有很多重复的代码(例如mov-template.h和lea-template.h), 甚至同一个指令的模板中也有重复的代码(例如mov-template.h中的mov_rm2r和movr2rm), 你可以尝试进一步减少代码之间的耦合(减少重复代码), 体会真正的结构化程序设计, 同时也可以减少由于疏忽而出错的可能性(例如复制了一大片代码之后忘了修改其中某处的代码).

强大的宏

用宏实现的模板功能极大地方便了helper函数的编写, 如果不使用模板功能, mov_i2r 形式的helper函数就要写三次: mov_i2r_b , mov_i2r_w , mov_i2r_l , 而且需要修改的时候, 三个helper函数都要分别修改(漏了一个就会造成bug)... 这不仅使得代码量大大增加, 维护的难度也急速上升.

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

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

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

在编译的时候, 编译器会提示代码的第2行有语法错误, 但如果你光看第2行代码, 你很难发现错误, 甚至会怀疑编译器有bug.

那宏到底要不要用呢? 一种客观的观点是, 在你可以控制的范围中使用. 这就像goto语句一样, 当你希望在多重循环中从最内层循环直接跳出所有循环, goto是最方便的做法. 但如果代码中到处都是goto, 已经严重影响到代码段的可读性了, 这种情况当然是不可取的. 至于ctags不支持宏的识别, 要从另一个角度来看这问题: 不是"因为ctags不支持宏, 所以不应该使用宏", 而是"因为ctags不支持宏, 所以ctags有可以改善的地方". 正是因为不被束缚, 世界才能不断进步.

源文件组织

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

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

- `exec.c` 中定义了操作码表 `opcode_table` 和helper函数 `exec()`, `exec()` 根据指令的 `opcode` 首字节查阅 `opcode_table`, 并调用相应的helper函数来模拟相应指令的执行. 除此之外, 和2字节转义码相关的2字节操作码表 `_2byte_opcode_table`, 以及各种指令组表也在 `exec.c` 中定义.
- `all-instr.h` 中列出了所有用于模拟指令执行的helper函数的声明, 这个头文件被 `exec.c` 包含, 这样就可以在 `exec.c` 中的 `opcode_table` 直接使用各种helper函数了.

- 除了 `exec.c` 和 `all-instr.h` 两个源文件之外, 目录下还有若干子目录, 这些子目录分别存放用于模拟不同功能的指令的源文件. i386手册根据功能对所有指令都进行了分类, 框架代码中对相关文件的管理参考了手册中的分类方法(其中 `special` 子目录下模拟了和NEMU相关的功能, 与i386手册无关). 以 `nemu/src/cpu/exec/data-mov` 目录下与 `mov` 指令相关的文件为例, 我们对其文件组织进行进一步的说明:
  - `mov.h` 中列出了用于模拟 `mov` 指令所有形式的helper函数的声明, 这个头文件被 `all-instr.h` 包含.
  - `mov-template.h` 是 `mov` 指令helper函数定义的模板, `mov` 指令helper函数的函数体都在这个文件中定义. 模板的功能是通过宏来实现的: 对于一条指令, 不同操作数长度的相近形式都有相似的行为, 可以将它们的公共行为用宏抽象出来. `mov-template.h` 的开头包含了头文件 `nemu/include/cpu/exec/template-start.h`, 结尾包含了头文件 `nemu/include/cpu/exec/template-end.h`, 它们包含了一些在模板头文件中使用的宏定义, 例如 `DATA_TYPE, REG()` 等, 使用它们可以编写出简洁的代码.
  - `mov.c` 中定义了 `mov` 指令的所有helper函数, 其中分三次对 `mov-template.h` 中定义的模板进行实例化, 进行宏展开之后就可以得到helper函数的完整定义了; 另外操作数后缀为 `v` 的helper函数也在 `mov.c` 中定义.

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

运行第一个C程序

运行第一个C程序

说了这么多, 现在到了动手实践的时候了, 你在PA2的第一个任务, 就是编写几条指令的helper函数, 使得第一个简单的C程序可以在NEMU中运行起来. 这个简单的C程序的代码是`testcase/src/mov-c.c`, 它做的事情十分简单, 对数组的某些元素进行赋值, 然后马上读出这些元素的值, 检查它们是否被正确赋值.

使用**assertion**进行验证

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

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

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

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

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

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

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

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

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

的方式来给出指令, 汇编器将会报错, 因为这条特殊的指令是我们人为添加的, 标准的汇编器并

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

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

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

另外唯一一点要注意的是, 目前我们不能让用户程序从 `main` 函数返回, 否则将会产生错误, 因此我们在用户程序从 `main` 函数返回之前, 使用 `HIT_GOOD_TRAP` 强行结束用户程序的运行, 同时也提示我们用户程序通过了所有的assertion.

不能返回的main函数

为什么目前让用户程序从 `main` 函数返回就会发生错误? 这个错误具体是怎么发生的?

运行时环境与交叉编译

在让NEMU运行用户程序之前, 我们先来讨论NEMU需要为用户程序的运行提供什么. 在你运行hello world程序时, 你敲入一条命令(或者点击一下鼠标), 程序就成功运行了, 但这背后其实隐藏着操作系统开发者和库函数开发者的无数汗水. 一个事实是, 应用程序的运行都需要运行时环境的支持, 包括加载, 销毁程序, 以及提供程序运行时的各种动态链接库(你经常使用的库函数就是运行时环境提供的)等. 现在轮到你来为用户程序提供运行时环境的支持了, 不用担心, 由于NEMU目前的功能并不完善, 我们必定无法向用户程序提供GNU/Linux般的运行时环境. 目前, 我们约定NEMU提供的运行时环境有:

1. 物理内存有128MB(当然, 这是我们模拟出来的物理内存), 所有内存地址都是物理地址
2. 程序入口位于地址 `0x100000`, 程序总是从这里开始执行
3. `%ebp` 的初值为 `0`, `%esp` 的初值为 `0x8000000`. 需要注意的是, 这个地址是物理内存的最大值, 是一个非法的物理地址, 不能直接访问
4. 程序通过 `nemu_trap` 结束运行
5. 不提供库函数的动态链接, 但提供静态链接, 故实际上对用户程序来说, 库函数的使用与运行时环境无关. 库函数的静态链接是通过框架代码中提供的函数库newlib实现的, 相应的文件有 `lib-common/newlib/libc.a` 和 `lib-common/newlib/include` 目录下的头文件, `Makefile` 中已经有相应的设置了. newlib是专门为嵌入式系统提供的, 库中的函数对运行时环境的要求极低, 其中一些函数甚至不需要任何运行时环境的支持(例如 `memcpy` 等), 这正好符合NEMU的情况. 这样, 你就可以在用户程序中使用一些不需要运行时环境支持的库函数了. 但类似于 `printf()` 这种需要运行时环境支持的库函数目前还是无法使用, 否则将会发生链接错误.

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

误, 因此我们不能使用gcc的默认选项来编译用户程序.

解决这个问题的方法是交叉编译, 我们需要在GNU/Linux下根据NEMU提供的运行时环境编译出能够在NEMU中运行的可执行文件. 框架代码已经把相应的配置准备好了.

修改工程目录下的 `Makefile` 文件, 更换NEMU的用户程序:

```
--- Makefile
+++ Makefile
@@ -55,2 +55,2 @@
-USERPROG = obj/testcase/mov
+USERPROG = obj/testcase/mov-c
 ENTRY = $(USERPROG)
```

修改后, 键入

```
make run
```

使用新的用户程序运行NEMU, 你会发现NEMU输出以下信息:

```
invalid opcode(eip = 0x0010000a): 83 ec 10 e8 00 00 00 00 ...

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

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

for more details.

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

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

这是因为你还没有实现以 `0x83` 为首字节的指令, 因此, 你需要开始在NEMU中添加指令了.

实现最少的指令

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

这里要再次强调, 你务必通过i386手册来查阅指令的功能, 不能想当然. 手册中给出了指令功能的完整描述(包括做什么事, 怎么做的, 有什么影响), 一定要仔细阅读其中的每一个单词, 对

指令功能理解错误和遗漏都会给以后的调试带来巨大的麻烦.

- `sub`, `cmp`: 要注意被减数和减数的位置
- `call`: `call` 指令有很多形式, 不过在PA中只会用到其中的几种, 现在只需要实现 `CALL rel32` 的形式就可以了
- `push`: 现在只需要实现 `PUSH r32` 的形式就可以了
- `test`: 在实现 `test` 指令之前, 你首先需实现EFLAGS寄存器, 你只需要在寄存器结构体中添加EFLAGS寄存器即可. EFLAGS是一个32位寄存器, 它的结构如下:

```
31                  23                  15                  7
+-------------------+---------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                     |V|R| |N|I O|O|D|I|T|S|Z| |
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | | |0| |   | | | | | | |0|
|                                     |M|F| |T|P L|F|F|F|F|F|F| |
+-------------------+---------------+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  关于EFLAGS中每一位的含义, 请查阅i386手册. 在NEMU中, 我们只会用到EFLAGS中以下的7个位: `CF`, `PF`, `ZF`, `SF`, `IF`, `DF`, `OF`. 其余位的功能可暂不实现. 添加EFLAGS寄存器需要用到结构体的位域(bit field)功能, 如果你从未听说过位域, 请查阅相关资料. 关于EFLGAS的初值, 我们遵循i386手册中提到的约定, 你需要在i386手册的第10章中找到这一初值, 然后在 `restart()` 函数中对EFLAGS寄存器进行初始化. 实现了EFLAGS寄存器之后, 你就可以实现 `test` 指令了.
- `je`: `je` 指令是 `jcc` 的一种形式

运行用户程序mov-c

编写相应的helper函数实现上文提到的六条指令, 具体细节请务必参考i386手册. 实现成功后, 在NEMU中运行用户程序mov-c, 你将会看到 `HIT GOOD TRAP` 的信息.

温馨提示

PA2阶段1到此结束.

简易调试器(2)

简易调试器(2)

你将要实现更多的指令, 然后在NEMU中运行更多更复杂的程序. 在这之前, 我们建议你先对简易调试器的功能进行扩展, 为调试提供更多的手段.

运行用户程序add

查看 `obj/testcase/add.txt` , 实现其中那些还没有实现的指令, 使得用户程序add可以在NEMU中成功运行. 你将使用add程序来测试简易调试器的新功能.

断点(2)

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

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

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

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

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

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

添加变量支持

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

```
(nemu) p test_data
```

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

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

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

```
readelf -a add
```

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

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

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

消失的符号

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

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

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES
  [ 0]                   NULL            00000000 000000 000000 00
  [ 1] .text             PROGBITS        00100000 001000 00009a 00
  [ 2] .eh_frame         PROGBITS        0010009c 00109c 000058 00
  [ 3] .data             PROGBITS        00100100 001100 000120 00
```

```
  [ 4] .comment          PROGBITS        00000000 001220 00001c 01
  [ 5] .shstrtab         STRTAB          00000000 00123c 00003a 00
  [ 6] .symtab           SYMTAB          00000000 0013b8 0000a0 10
  [ 7] .strtab           STRTAB          00000000 001458 00001e 00
```

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

```
hd add
```

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

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

寻找"Hello World!"

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

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

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

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

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

而对于一个整型数组A, 如果想输出A[1]的值, 可以通过以下方式:

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

为表达式求值添加变量的支持

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

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

```
man 5 elf
```

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

丢失的信息

在用户程序中定义以下字符数组:

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

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

冗余的符号表

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

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

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

目标文件中也有符号表, 我们同样可以丢弃它:

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

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

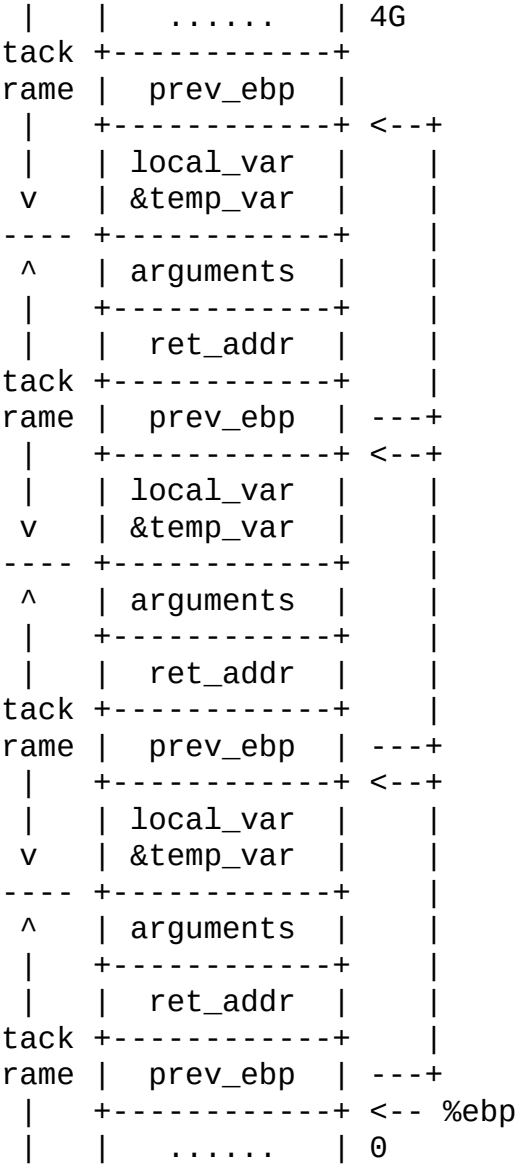
```
gcc -o hello hello.o
```

你发现了什么问题? 尝试对比上述两种情况, 并分析其中的原因.

打印栈帧链

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

在堆栈中形成的栈帧链结构如下:



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

```
typedef struct {
```

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

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

由于缺乏形参和局部变量的具体信息, 我们只需要打印地址, 函数名, 以及前5个参数就可以了, 打印格式可以参考GDB中 bt 命令的输出. 如何确定某个地址落在哪一个函数中呢? 这就需要符号表的帮助了:

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

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

打印栈帧链

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

%ebp是必须的吗?

使用优化选项编译代码的时候, gcc会对代码进行优化, 会将 %ebp 当作普通的寄存器来使用, 不再让其作为指示当前的栈帧, 更多的信息可以查阅 man gcc 中的 -fomit-frame-pointer 选项. 我们使用 -O2

来编译NEMU, 你可以对NEMU进行反汇编, 查看一些函数的代码. 在这种情况下, 代码要怎么找到函数调用的参数和局部变量?

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

温馨提示

PA2阶段2到此结束.

实现更多的指令

实现更多的指令

为了让NEMU支持大部分程序的运行, 目前你需要实现以下指令(一些例外会在备注中说明):

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

你只需要实现上述红色字体的指令, 它们不多不少都和以下的某些内容相关: EFLAGS, 堆栈, 整数扩展, 加减溢出判断. 框架代码已经把黑色字体的指令实现好了, 但并没有填写`opcode_table`. 此外, 某些需要更新EFLAGS的指令, 以及有符号立即数的译码函数(在`nemu/src/cpu/decode/decode-template.h`中定义)并没有完全实现好(框架代码中已经插入了`panic()`作为提示), 你还需要编写相应的功能.

测试用例

未测试代码永远是错的, 你需要足够多的测试用例来测试你的NEMU. 我们在`testcase`目录下准备了一些测试用例, 需要更换测试用例时, 修改工程目录下`Makefile`中的`USERPROG`变量, 改成测试用例的可执行文件(`obj/testcase/xxx`, 不是C源文件)即可.

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

- hello-inline-asm
- hello
- hello-str
- integral
- quadratic-eq

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

实现**binary scaling**

要在NEMU中实现浮点指令也不是不可能的事情. 但实现浮点指令需要涉及x87架构的很多细节, 而且我们并不打算在用户程序中直接使用浮点指令. 为了在保持程序逻辑的同时不引入浮点指令, 我们通过整数来模拟实数的运算, 这样的方法叫binary scaling.

我们先来说明如何用一个32位整数来表示一个实数. 为了方便叙述, 我们称用binary scaling方法表示的实数的类型为`FLOAT`. 我们约定最高位为符号位, 接下来的15位表示整数部分, 低16位表示小数部分, 即约定小数点在第15和第16位之间(从第0位开始). 从这个约定可以看到, `FLOAT`类型其实是实数的一种定点表示.

```
31  30                  16                   0
+----+-------------------+--------------------+
```

```
|sign|      integer      |      fraction      |
+----+-------------------+--------------------+
```

这样, 对于一个实数 a , 它的 FLOAT 类型表示 A = a * 2^16 (截断结果的小数部分). 例如实数 1.2 和 5.6 用 FLOAT 类型来近似表示, 就是

```
1.2 * 2^16 = 78643 = 0x13333
+----+-------------------+--------------------+
| 0  |         1         |        3333        |
+----+-------------------+--------------------+


5.6 * 2^16 = 367001 = 0x59999
+----+-------------------+--------------------+
| 0  |         5         |        9999        |
+----+-------------------+--------------------+
```

而实际上, 这两个 FLOAT 类型数据表示的数是:

```
0x13333 / 2^16 = 1.19999695
0x59999 / 2^16 = 5.59999084
```

对于负实数, 我们用相应正数的相反数来表示, 例如 -1.2 的 FLOAT 类型表示为:

```
-(1.2 * 2^16) = -0x13333 = 0xfffecccd
```

比较FLOAT和float

FLOAT 和 float 类型的数据都是32位, 它们都可以表示2^32个不同的数, 但由于表示方法不一样, FLOAT 和 float 能表示的数集是不一样的. 思考一下, 我们用 FLOAT 来模拟表示 float , 这其中隐含着哪些取舍?

接下来我们来考虑 FLOAT 类型的常见运算, 假设实数 a , b 的 FLOAT 类型表示分别为 A , B .

- 由于我们使用整数来表示 FLOAT 类型, FLOAT 类型的加法可以直接用整数加法来进行:

  ```
  A + B = a * 2^16 + b * 2^16 = (a + b) * 2^16
  ```

- 由于我们使用补码的方式来表示FLOAT类型数据, 因此FLOAT类型的减法用整数减法来进行.

  ```
  A - B = a * 2^16 - b * 2^16 = (a - b) * 2^16
  ```

- FLOAT 类型的乘除法和加减法就不一样了:

  ```
  A * B = a * 2^16 * b * 2^16 = (a * b) * 2^32 != (a * b) * 2^16
  ```

  也就是说, 直接把两个 FLOAT 数据相乘得到的结果并不等于相应的两个浮点数乘积的 FLOAT 表示. 为了得到正确的结果, 我们需要对相乘的结果进行调整: 只要将结果除以 2^16 , 就能得出正确的结果了. 除法也需要对结果进行调整, 至于如何调整, 当然难不倒聪明的你啦.

- 如果把 A = a * 2^16 看成一个映射, 那么在这个映射的作用下, 关系运算是保序的, 即 a <= b 当且仅当 A <= B, 故 FLOAT 类型的关系运算可以用整数的关系运算来进行.

有了这些结论, 要用 FLOAT 类型来模拟实数运算就很方便了. 除了乘除法需要额外实现之外, 其余运算都可以直接使用相应的整数运算来进行. 例如

```
float a = 1.2;
float b = 10;
int c = 0;
if(b > 7.9) {
    c = (a + 1) * b / 2.3;
}
```

用 FLOAT 类型来模拟就是

```
FLOAT a = f2F(1.2);
FLOAT b = int2F(10);
int c = 0;
if(b > f2F(7.9)) {
    c = F2int(F_div_F(F_mul_F((a + int2F(1)), b), f2F(2.3)));
}
```

其中还引入了一些类型转换函数来实现和 FLOAT 相关的类型转换.

实现binary scaling

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

```
/* lib-common/FLOAT.h */
int32_t F2int(FLOAT a);
FLOAT int2F(int a);
FLOAT F_mul_int(FLOAT a, int b);
FLOAT F_div_int(FLOAT a, int b);
/* lib-common/FLOAT.c */
FLOAT f2F(float a);
FLOAT F_mul_F(FLOAT a, FLOAT b);
FLOAT F_div_F(FLOAT a, FLOAT b);
FLOAT Fabs(FLOAT a);
```

其中 F_mul_int() 和 F_div_int() 用于计算一个 FLOAT 类型数据和一个整型数据的积/商, 这两种特殊情况可以快速计算出结果, 不需要将整型数据先转化成 FLOAT 类型再进行运算. lib-common/FLOAT.c 中的 pow() 函数目前不会用到, 我们会在PA4再提到它. 实现成功后, 你还需要在 lib-common/Makefile.part 中编写用于生成 FLOAT.o 的规则, 要求如下:

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

`-p` 目录创建

编写规则后, 修改工程目录下的 `Makefile` 文件:

```
--- Makefile
+++ Makefile
@@ -11,2 +11,2 @@
 NEWLIBC = $(NEWLIBC_DIR)/libc.a
-#FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT.a
+FLOAT = obj/$(LIB_COMMON_DIR)/FLOAT.a
```

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

事实上, 我们并没有考虑计算结果溢出的情况, 不过我们的测试用例中的浮点数结果都可以在 `FLOAT` 类型中表示, 所以你可以不关心溢出的问题. 如果你不放心, 你可以在上述函数的实现中插入assertion来捕捉溢出错误.

编写自己的测试用例

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

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

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

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

实现更多的指令

你需要实现上表提到的更多指令, 以支持 `testcase` 目录下更多程序的运行. 实现的时候尽可能使用框架代码中的宏(参考 `include/cpu/exec/helper.h` 和 `include/cpu/exec/template-start.h`), 它们可以帮助你编写出简洁的代码. 你可以自由选择按照什么顺序来实现指令.

你需要使用 `testcase` 目录下的测试用例来测试你的实现. 你不需要实现所有指令的所有形式, 只需要通过 `testcase` 目录下的所有测试就可以了(hello-inline-asm和hello除外).

NEMU的本质

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

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

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

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

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

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

早在Martin Davis教授提出L语言之前, 科学家们就已经在探索什么问题是可以计算的了. 回溯到19世纪30年代, 为了试图回答这个问题, 不同的科学家提出并研究了不同的计算模型, 包括Gödel, Herbrand和Kleen研究的递归函数, Church提出的λ-演算, Turing提出的图灵机, 后来发现这些模型在计算能力上都是等价的; 到了40年代, 计算机就被制造出来了. 后来甚至还有人证明了, 如果使用无穷多个算盘拼接起来进行计算, 其计算能力和图灵机等价! 我们可以从中得出一个推论, 通用程序在不同的

计算模型中有不同的表现形式. NEMU作为一个通用程序, 在19世纪30年代有着非凡的意义, 如果你能在80年前设计出NEMU, 说不定"图灵奖"就要用你的名字来命名了. 计算的极限这一篇科普文章叙述了可计算理论的发展过程, 我们强烈建议你阅读它, 体会人类的文明(当然一些数学功底还是需要的). 如果你对可计算理论感兴趣, 可以选修宋方敏老师的计算理论导引课程.

把思绪回归到PA中, 通用程序的性质告诉我们, NEMU的潜力是无穷的. 但NEMU现在连输出一句话的功能都无法向用户程序提供, 为了创造出一个缤纷多彩的世界, 你觉得NEMU还缺少些什么呢?

捕捉死循环(有点难度)

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

当用户程序陷入死循环时, 让用户程序暂停下来, 并输出相应的提示信息

你觉得应该如何实现? 如果你感到疑惑, 在互联网上搜索相关信息.

温馨提示

PA2阶段3到此结束. 此阶段需要实现较多指令, 你不必在一周内完成此阶段的所有内容, 但注意合理分配时间, 不要影响到后续阶段的实验.

实现加载程序的loader

实现加载程序的loader

loader是一个用于加载程序的模块, 实现了足够多的指令之后, 你也可以实现一个很简单的loader, 帮助你理解加载程序的过程.

可执行文件的组织

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

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

我们在PA1中已经提到了以下内容:

> 在一个完整的模拟器中, 程序应该存放在磁盘中, 但目前我们并没有实现磁盘的模拟, 因此NEMU先把内存开始的位置作为ramdisk来使用.

现在的ramdisk十分简单, 它只有一个文件, 也就是我们将要加载的用户程序, 访问内存位置0就可以得到用户程序的第一个字节. 这其实已经回答了上述第一个问题: 可执行文件位于内存位置0. 为了回答剩下的问题, 我们首先需要了解可执行文件是如何组织的.

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

堆栈在哪里?

我们提到了代码和数据都在可执行文件里面, 但却没有提到堆栈. 为什么堆栈的内容没有放入可执行文件里面? 那程序运行时刻用到的堆栈又是怎么来的?

如何识别不同格式的可执行文件?

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

你应该已经学会使用 `readelf` 命令来查看ELF文件的信息了. ELF文件提供了两个视角来组织一个可执行文件, 一个是面向链接过程的section视角, 这个视角提供了用于链接与

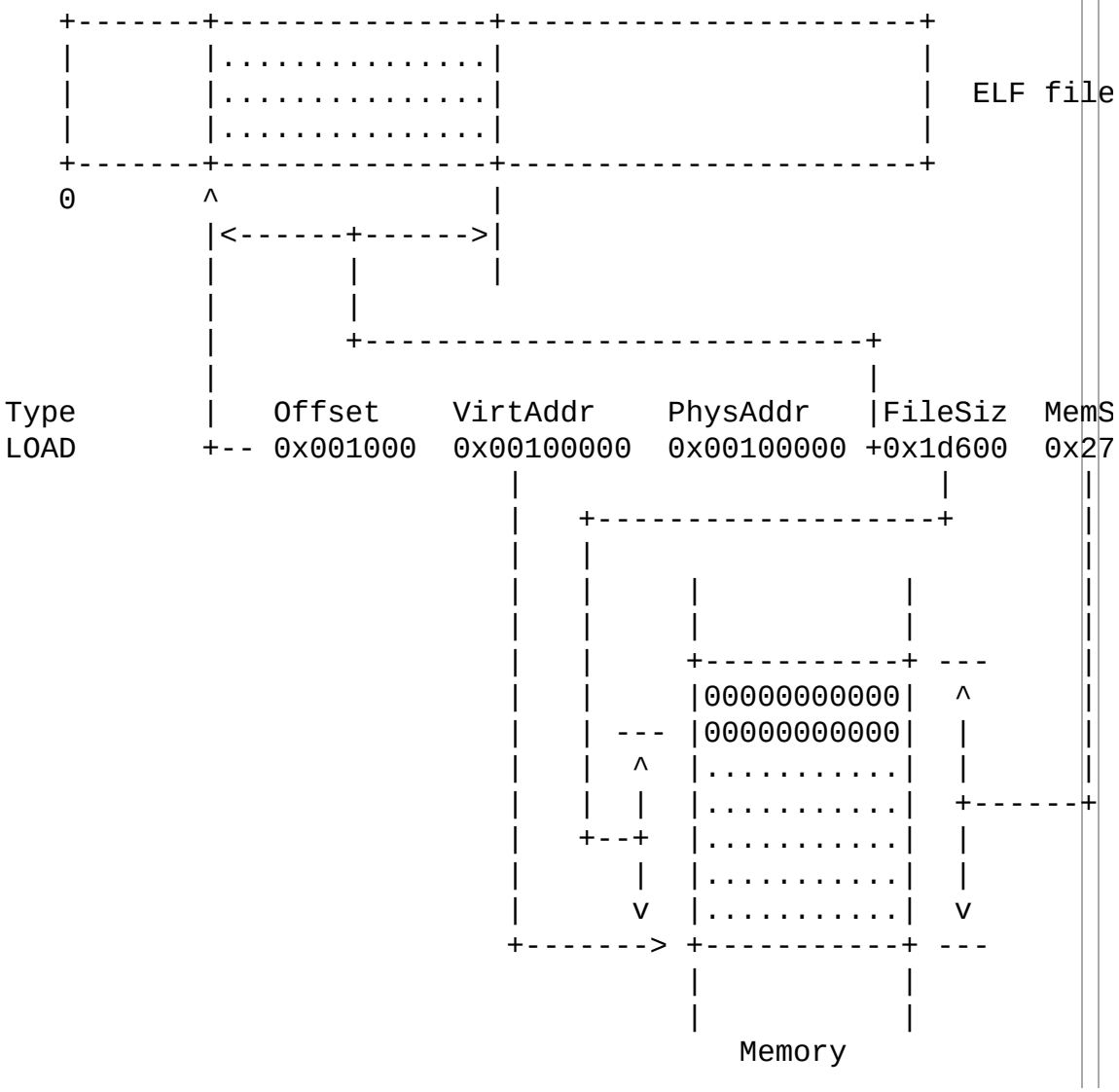
重定位的信息(例如符号表); 另一个是面向执行的segment视角, 这个视角提供了用于加载可执行文件的信息. 通过readelf命令, 我们还可以看到section和segment之间的映射关系: 一个segment可能由0个或多个section组成, 但一个section可能不被包含于任何segment中.

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

冗余的属性?

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

我们通过下面的图来说明如何根据segment的属性来加载它:



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

为什么要清零?

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

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

```
man 5 elf
```

在**kernel**中实现**loader**

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

在PA中, kernel是一个单任务微型操作系统的内核, 其代码在工程目录的 `kernel` 目录下. kernel已经包含了后续PA用到的所有模块, 换句话说, 我们现在就要在NEMU上运行一个操作系统了(尽管这是一个十分简陋的操作系统), 同时也将带领你根据课堂上的知识剖析一个简单操作系统内核的组成. 这不仅是作为对这些抽象知识的很好的复(预)习, 同时也是为以后的操作系统实验打下坚实的基础. 由于NEMU的功能是逐渐添加的, kernel也要配合这个过程, 你会通过 `kernel/include/common.h` 中的一些与实验进度相关的宏来控制kernel的功能. 随着实验进度的推进, 我们会逐渐讲解所有的模块, kernel做的工作也会越来越多. 因此在阅读kernel的代码时, 你只需要关心和当前进度相关的模块就可以了, 不要纠缠于和当前进度无关的代码.

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

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

```
kernel
├── include
│   ├── common.h
│   ├── debug.h
│   ├── memory.h
│   ├── x86
│   │   ├── cpu.h
```

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

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

1. 第一条指令从 `kernel/start/start.S` 开始, 设置好堆栈之后就跳转到 `kernel/src/main.c` 的 `init()` 函数处执行.
2. 由于NEMU还没有实现分段分页的功能, 此时kernel会跳过很多初始化工作, 直接跳转到 `init_cond()` 函数处继续进行初始化.
3. 继续跳过一些初始化工作之后, 会通过 `Log()` 宏输出一句话. 需要说明的是, kernel中定义的 `Log()` 宏并不是NEMU中定义的 `Log()` 宏, kernel和NEMU的代码是相互独立的, 因此编译某一方时都不会受到对方代码的影响, 你在阅读代码的时候需要注意这一点. 在kernel中, `Log()` 宏通过 `printk()` 输出. 阅读 `printk()` 的代码, 发现此时 `printk()` 什么都不做就直接返回了, 这是由于现在NEMU还不能提供输出的功能, 因此现在kernel中的 `Log()` 宏并不能成功输出.
4. 调用 `loader()` 函数加载用户程序, `loader()` 函数会返回用户程序的入口地址.
5. 跳转到用户程序的入口执行.

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

原型如下:

```
int ramdisk_read(uint8_t *buf, uint32_t offset, uint32_t le
```

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

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

```
--- testcase/Makefile.part
+++ testcase/Makefile.part
@@ -8,2 +8,2 @@
 testcase_START_OBJ := $(testcase_OBJ_DIR)/start.o
-testcase_LDFLAGS := -m elf_i386 -e start -Ttext=0x00100000
+testcase_LDFLAGS := -m elf_i386 -e main -Ttext-segment=0x0
```

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

```
--- Makefile
+++ Makefile
@@ -55,2 +55,2 @@
 USERPROG = obj/testcase/mov-c
-ENTRY = $(USERPROG)
+ENTRY = $(kernel_BIN)
```

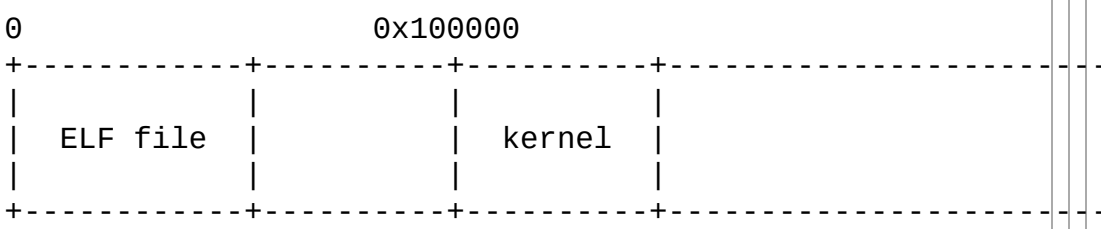
我们从运行时刻的角度来描述NEMU中物理内存的变化过程:

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

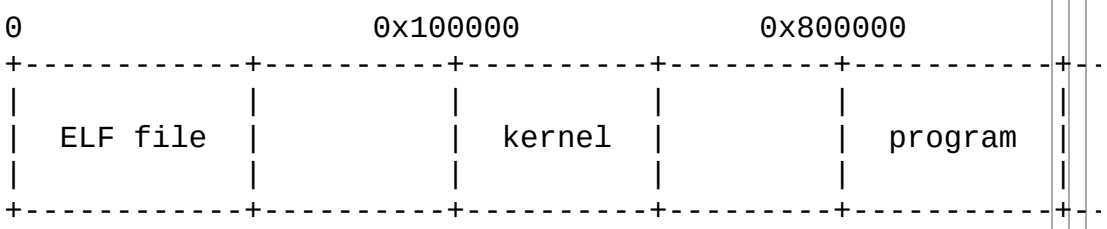
  

- 第二项初始化操作是将entry加载到内存位置 `0x100000` (由 `nemu/src/monitor/monitor.c` 中的 `load_entry()` 函数完成). 之前entry就是用户程序本身, 引入kernel之后, entry就变成kernel了. 换句话说, 在我们的PA中, kernel是由NEMU的monitor模块直接加载到内存中的. 这一点与真实的计算机有所不同, 在真实的计算机中, 计算机启动之后会首先运行BIOS程序, BIOS程序会将MBR加载到内存, MBR再加载别的程序... 最后加载kernel. 要模拟这个过程需要一个完善的模拟磁盘, 而且需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

在NEMU开始执行第一条指令之前, kernel就已经在内存中了:



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



实现loader

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

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

```
make test
```

脚本会将 `testcase` 中的测试用例逐个进行测试, 并报告每个测试用例的运行结果, 这样你就不需要手动切换测试用例了. 如果一个测试用例运行失败, 脚本将会保留相应的日志文件; 当使用脚本通过这个测试用例的时候, 日志文件将会被移除.

温馨提示

PA2阶段4到此结束.

运行hello-str程序

运行hello-str程序

hello-str程序的功能是通过 `sprintf()` 函数将字符串格式化打印到一个缓冲区中, 然后使用 `strcmp()` 函数来检查字符串是否正确. hello-str程序编译通过之后, 你就可以在NEMU中运行它了. 不过在运行过程中, 你可能会发现还有一些指令没有在NEMU中实现, 你需要实现它们. 另外 `libc.a` 中用到的 `cmovcc` 指令是在i386之后添加, 你在i386手册上找不到它们, 这个页面提供了它们的信息.

捣蛋鬼的恶作剧

你还会发现在 `sprintf()` 格式化的执行过程中遇到了两条浮点指令(下面的例子中涉及到的地址可能与你看到的反汇编结果不一样):

```
80482c8:    89 c7             mov    %eax,%edi
80482ca:    89 85 74 ff ff ff     mov    %eax,-0x8c(%ebp)
80482d0:    d9 ee             fldz
80482d2:    29 d0             sub    %edx,%eax
80482d4:    c7 85 7c ff ff ff 00    movl   $0x0,-0x84(%ebp)
80482db:    00 00 00
80482de:    dd 9d 10 ff ff ff     fstpl  -0xf0(%ebp)
80482e4:    c7 85 78 ff ff ff 00    movl   $0x0,-0x88(%ebp)
80482eb:    00 00 00
```

我们在代码中并没有使用浮点数, 这两条浮点指令是怎么来的呢? Intel有一套x87指令集专门处理浮点运算, 查阅相关资料后发现, `fldz` 是把浮点数0压入浮点堆栈ST(0), 而 `fstpl` 则是把浮点堆栈ST(0)的值弹出到内存中. 我们可以推测这两条执行是在对一个浮点变量进行初始化, 如果格式化字符串中含有浮点说明符 `%lf`, 代码将会使用这个浮点变量进行处理:

```
double a = 0.0;
// ...
if( the conversion specifier is "%lf" ) {
    a = get_value();
    /* format the double value */
}
```

虽然我们没有在代码中使用浮点数, 但执行格式化的代码还是需要对 `a` 变量进行初始化, 因此在执行过程中就会遇到因此而产生的两条浮点指令.

如果这两条指令仅仅是为了初始化一个我们不会用到的浮点变量, 有没有办法能够绕过它们呢? 你可能会想修改newlib的源代码, 然后重新编译. 这是一个能行的方法, 但代价比较大, 有兴趣的同学可以到newlib的官网上下载源代码并尝试修改.

如果能把这两条浮点指令去掉就好了... 没错, 我们要想办法把这两条浮点指令去掉, 既然不能从源代码着手, 我们就直接修改编译后的ELF文件吧!

体验黑客的乐趣

这好像黑客做的事情一样, 听上去真有意思! 不过, 黑客能做一些我们认为很厉害的事情, 是因

为他们对计算机系统的构成了如指掌. 事实上, 你在学习完ICS之后, 就已经具备一些黑客的入门知识了. 不信? 那就动手试试吧!

我们的破解目标是在ELF文件中找到上述两条浮点指令, 然后去掉它们. 在NEMU中运行报出的"invalid opcode"错误已经告诉我们这两条浮点指令在哪一个地址了, 你也可以根据 objdump 的结果找到浮点指令的前后指令, 方便进行对照.

回想一下loader加载程序的过程, ELF文件中已经存储了每一个segment的静态内容, 加载程序的过程就是把这些静态内容从ELF文件中读入到正确的内存位置. 现在我们已经找到浮点指令的地址, 我们就可以反过来推导它们在ELF文件中的位置了. 以我们编译出来的hello-str程序为例, 通过readelf命令查看hello-str的program header信息:

```
Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Al
LOAD           0x000000 0x08048000 0x08048000 0x09bdc 0x09bdc R E 0x
LOAD           0x00a000 0x08052000 0x08052000 0x0098c 0x009f8 RW  0x
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x
```

上述的第一条浮点指令 fldz 的地址是 0x80482d0 , 由于

```
0x8048000 = segment1.VirtAddr <= instr.VirtAddr < segment1.VirtAddr
```

所以第一条浮点指令落在第一个segment中, 因此我们可以算出 fldz 指令在ELF文件中的偏移量:

```
instr.offset_in_ELF = segment1.Offset + (instr.VirtAddr - segment1.V
```

找到了浮点指令在ELF文件中的偏移量之后, 我们就可以动手去掉它了. 使用 vim 打开hello-str可执行文件:

```
vim -b hello-str
```

使用 -b 参数代表以二进制方式打开, 可以避免一些和中文字符相关的编码问题. 哎呀, 打开之后竟然是一片乱码! 要怎么改呢? 别着急, 我们需要 xxd 工具的帮助, 在 vim 中键入以下内容:

```
:%!xxd
```

xxd 工具已经把hello-str可执行文件转化成十六进制表示了, 这下ELF文件中的所有内容就一览无余了, 就连ELF文件首部的魔数也无所遁形. 根据刚才计算的结果, 我们直捣黄龙, 直接找到ELF文件中偏移量为 0x2d0 的位置, 和 objdump 结果进行对照, 发现正好是 fldz 指令的编码.

找到这个捣蛋鬼之后, 接下来就是要收拾它. 不过可别一时冲动, 把 fldz 指令对应的两个字节直接删掉, 因为这样会影响segment的长度, 需要修改program header中的相应信息, 同时还要修改相应section header的信息... 总之冲动是魔鬼, 差点就功亏一篑了. 能不能在不影响segment属性的情况下把 fldz 指令去掉呢? 或者有没有什么指令可以把 fldz 替换掉, 同时不影响程序其它部分的功能? 这当然难不倒聪(yin)明(xian)的你, 你默默地把i386手册翻到 nop 的页面... 这下你知道 nop 的指令长度为什么是1个字节了吧. 于是 fldz 这个捣蛋鬼就被悄悄地抹杀了.

把 fldz 干掉之后, 在 vim 中键入以下内容:

```
:%!xxd -r
```

xxd 工具会把十六进制表示转换回二进制表示的ELF. 保存后退出 vim , 用 objdump 查看修改后的hello-str可执行文件:

```
80482c8:    89 c7             mov    %eax,%edi
80482ca:    89 85 74 ff ff ff    mov    %eax,-0x8c(%ebp)
80482d0:    90                nop
80482d1:    90                nop
80482d2:    29 d0             sub    %edx,%eax
80482d4:    c7 85 7c ff ff ff 00    movl   $0x0,-0x84(%ebp)
80482db:    00 00 00
80482de:    dd 9d 10 ff ff ff    fstpl  -0xf0(%ebp)
80482e4:    c7 85 78 ff ff ff 00    movl   $0x0,-0x88(%ebp)
80482eb:    00 00 00
```

抹杀成功! 接下来用同样的方式把 `fstpl` 指令也干掉, hello-str程序就可以在NEMU中成功运行了. 不过如果你不打算使用 `vim`, 你需要在shell中使用 `xxd` 工具把十六进制表示的结果输出到一个临时文件中, 修改后再使用 `xxd` 转换成二进制文件. 关于如何使用 `xxd`, 请查阅

```
man xxd
```

不过你会发现, 如果重新编译hello-str程序, 讨厌的捣蛋鬼又起死回生了. 为了斩草除根, 我们需要修改 `libc.a` 文件.

`libc.a` 是一个归档文件, 其中包含了很多用于链接的目标文件. 直接在 `libc.a` 中定位到需要修改的位置会比较麻烦, 我们可以将需要修改的那个目标文件抽取出来, 然后单独修改它. 我们知道需要修改的代码在 `_svfprintf_r()` 函数中, 可以使用如下指令找到这个函数在哪一个目标文件中:

```
nm -o libc.a | grep '_svfprintf_r'
```

命令的执行结果显示, `_svfprintf_r()` 函数在归档文件中的 `lib_a-svfprintf.o` 中定义. 接下来将这个目标文件从归档文件中抽取出来:

```
ar x libc.a lib_a-svfprintf.o
```

抽取出目标文件之后, 就可以通过上文提到的方法来去掉浮点指令了. 不过这次需要修改的文件是没有经过链接的目标文件, 目标文件中没有segment的信息, 不过仅仅利用section的信息也可以实现我们的目的, `readelf` 和 `objdump` 的结果已经包含了我们需要的所有信息, 至于如何发现和利用它们, 就交给你来思考吧!

修改成功后, 将修改后的目标文件重新加入归档文件中:

```
ar r libc.a lib_a-svfprintf.o
```

现在的 `libc.a` 中的 `_svfprintf_r()` 函数已经经过我们的处理了, 使用它进行链接产生的hello-str程序已经不会再含有我们之前去掉的那两条浮点指令.

和代码玩游戏

在程序设计基础课上, 你学会了如何对数据做一些基本操作, 学会了play with the data, 例如算阶乘, 用链表组织学生信息, 但你从来都不知道代码是什么东西. 而在NEMU中实现断点其实就是在play with the code, 你发现你可以很简单地控制程序的执行流. 和代码玩游戏的最高境界就要数self-modifying code了, 它们能够在运行时刻修改自己, 但这种代码极难读懂, 维护更是难上加难, 因此它们成为了hacker们炫耀自己的一种工具.

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

在NEMU中运行hello-str程序

根据上述讲义内容, 在NEMU中运行hello-str程序.

必答题

你需要在实验报告中用自己的语言, 尽可能详细地回答下列问题.

- 编译与链接 在 `nemu/include/cpu/helper.h` 中, 你会看到由 `static inline` 开头定义的 `instr_fetch()` 函数和 `idex()` 函数. 选择其中一个函数, 分别尝试去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你会看到发生错误. 请分别解释为什么会发生这些错误? 你有办法证明你的想法吗?
- 编译与链接
  1. 在 `nemu/include/common.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问重新编译后的NEMU含有多少个 `dummy` 变量的实体? 你是如何得到这个结果的?
  2. 添加上题中的代码后, 再在 `nemu/include/debug.h` 中添加一行 `volatile static int dummy;` 然后重新编译NEMU. 请问此时的NEMU含有多少个 `dummy` 变量的实体? 与上题中 `dummy` 变量实体数目进行比较, 并解释本题的结果.
  3. 修改添加的代码, 为两处 `dummy` 变量进行初始化: `volatile static int dummy = 0;` 然后重新编译NEMU. 你发现了什么问题? 为什么之前没有出现这样的问题? (回答完本题后可以删除添加的代码.)
- 了解Makefile 请描述你在工程目录下敲入 `make` 后, `make` 程序如何组织.c和.h文件, 最终生成可执行文件 `obj/nemu/nemu`. (这个问题包括两个方面: `Makefile` 的工作方式和编译链接的过程.) 关于 `Makefile` 工作方式的提示:
  - `Makefile` 中使用了变量, 函数, 包含文件等特性
  - `Makefile` 运用并重写了一些implicit rules
  - 在 `man make` 中搜索 `-n` 选项, 也许会对你有帮助
  - RTFM

温馨提示

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

杂项

为什么要学习计算机系统基础

为什么要学习计算机系统基础

一知半解

你已经学过 程序设计基础 课程了, 对于C和C++程序设计已有一定的基础, 但你会发现, 你可能还是不能理解以下程序的运行结果:

数组求和

```
int sum(int a[ ], unsigned len) {
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

当 `len = 0` 时, 执行 `sum` 函数的for循环时会发生 `Access Violation`, 即"访问违例"异常. 但是, 当参数 `len` 说明为 `int` 型时, `sum` 函数能正确执行, 为什么?

整数的平方

若 `x` 和 `y` 为 `int` 型, 当 `x = 65535` 时, 则 `y = x*x = -131071`. 为什么?

多重定义符号

```
/*---main.c---*/
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main() {
    p1();
    printf("d=%d,x=%d\n",d,x);
    return 0;
}

/*---p1.c---*/
double d;
void p1() {
    d=1.0;
}
```

在上述两个模块链接生成的可执行文件被执行时, `main` 函数的 `printf` 语句打印出来的值是: `d=0,x=1072693248`. 为什么不是 `d=100,x=200`?

奇怪的函数返回值

```
double fun(int i) {
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

从 fun 函数的源码来看, 每次返回的值应该都是 3.14 , 可是执行 fun 函数后发现其结果是:

- fun(0) 和 fun(1) 为 3.14
- fun(2) 为 3.1399998664856
- fun(3) 为 2.00000061035156
- fun(4) 为 3.14 并会发生 访问违例 这是为什么?

时间复杂度和功能都相同的程序

```
void copyij(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
void copyji(int src[2048][2048], int dst[2048][2048]) {
      int i,j;
      for (j = 0; j < 2048; j++)
          for (i = 0; i < 2048; i++)
              dst[i][j] = src[i][j];
}
```

上述两个功能完全相同的函数, 时间复杂度也完全一样, 但在Pentium 4处理器上执行时, 所测时间相差大约21倍. 这是为什么? 猜猜看是 copyij 更快还是 copyji 更快?

网友贴出的一道百度招聘题

请给出以下C语言程序的执行结果, 并解释为什么.

```
#include <stdio.h>
int main() {
    double a = 10;
    printf("a = %d\n", a);
    return 0;
}
```

该程序在IA-32上运行时, 打印结果为 a=0 ; 在x86-64上运行时, 打印出来的 a 是一个不确定值. 为什么?

整数除法

以下两个代码段的运行结果是否一样呢?

- 代码段一:

```
    int a = 0x80000000;
    int b = a / -1;
```

```
printf("%d\n", b);
```

- 代码段二:

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

代码段一的运行结果为 `-2147483648`; 而代码段二的运行结果为 `Floating point exception`, 显然, 代码段二运行时被检测到了"溢出"异常. 看似同样功能的程序为什么结果完全不同?

类似上面这些例子还可以举出很多. 从这些例子可以看出, 仅仅明白高级语言的语法和语义, 很多情况下是无法理解程序执行结果的.

站得高, 看得远

国内很多学校老师反映, 学完高级语言程序设计后会有一些学生不喜欢计算机专业了, 这是为什么? 从上述给出的例子应该可以找到部分答案, 如果一个学生经常对程序的执行结果百思不得其解, 那么他对应用程序开发必然产生恐惧心理, 也就对计算机专业逐渐失去兴趣. 其实, 程序的执行结果除了受编程语言的语法和语义影响外, 还与程序的执行机制息息相关. 计算机系统基础 课程主要描述程序的底层执行机制, 因此, 学完本课程后同学们就能很容易地理解各种程序的执行结果, 也就不会对程序设计失去信心了.

我们还经常听到学生问以下问题: 像地质系这些非计算机专业的学生自学JAVA语言等课程后也能找到软件开发的工作, 而我们计算机专业学生多学那么多课程不也只能干同样的事情吗? 我们计算机专业学生比其他专业自学计算机课程的学生强在哪里啊? 现在计算机学科发展这么快, 什么领域都和计算机相关, 为什么我们计算机学科毕业的学生真正能干的事也不多呢? ...

确实, 对于大部分计算机本科专业学生来说, 硬件设计能力不如电子工程专业学生, 行业软件开发和应用能力不如其他相关专业学生, 算法设计和分析基础又不如数学系学生. 那么, 计算机专业学生的特长在哪里? 我们认为计算机专业学生的优势之一在于计算机系统能力, 即具备计算机系统层面的认知与设计能力、能从计算机系统的高度考虑和解决问题.

随着大规模数据中心(WSC)的建立和个人移动设备(PMD)的大量普及使用, 计算机发展进入了后PC时代, 呈现出"人与信息世界及物理世界融合"的趋势和网络化, 服务化, 普适化和智能化的鲜明特征. 后PC时代WSC, PMD和PC等共存, 使得原先基于PC而建立起来的专业教学内容已经远远不能反映现代社会对计算机专业人才的培养要求, 原先计算机专业人才培养强调"程序"设计也变为更强调"系统"设计.

后PC时代, 并行成为重要主题, 培养具有系统观的, 能够进行软, 硬件协同设计的软硬件贯通人才是关键. 而且, 后PC时代对于大量从事应用开发的应用程序员的要求也变得更高. 首先, 后PC时代的应用问题更复杂, 应用领域更广泛. 其次, 要能够编写出各类不同平台所适合的高效程序, 应用开发人员必需对计算机系统具有全面的认识, 必需了解不同系统平台的底层结构, 并掌握并行程序设计技术和工具.

下图描述了计算机系统抽象层的转换.

从图中可以看出, 计算机系统由不同的抽象层构成, "计算"的过程就是不同抽象层转换的过程, 上层是下层的抽象, 而下层则是上层的具体实现. 计算机学科主要研究的是计算机系统各个不同抽象层的实现及其相互转换的机制, 计算机学科培养的应该主要是在计算机系统或在系统某些层次上从事相关工作的人才.

相比于其他专业, 计算机专业学生的优势在于对系统深刻的理解, 能够站在系统的高度考虑和解决应用问题, 具有系统层面的认知和设计能力, 包括:

- 能够对软, 硬件功能进行合理划分
- 能够对系统不同层次进行抽象和封装
- 能够对系统的整体性能进行分析和调优
- 能够对系统各层面的错误进行调试和修正
- 能够根据系统实现机理对用户程序进行准确的性能评估和优化
- 能够根据不同的应用要求合理构建系统框架等

要达到上述这些在系统层面上的分析, 设计, 检错和调优等系统能力, 显然需要提高学生对整个计算机系统实现机理的认识, 包括:

- 对计算机系统整机概念的认识
- 对计算机系统层次结构的深刻理解
- 对高级语言程序, ISA, OS, 编译器, 链接器等之间关系的深入掌握
- 对指令在硬件上执行过程的理解和认识

- 对构成计算机硬件的基本电路特性和设计方法等的基本了解等 从而能够更深刻地理解时空开销和权衡, 抽象和建模, 分而治之, 缓存和局部性, 吞吐率和时延, 并发和并行, 远程过程调用(RPC), 权限和保护等重要的核心概念, 掌握现代计算机系统中最核心的技术和实现方法.

  计算机系统基础 课程的主要教学目标是培养学生的系统能力, 使其成为一个"高效"程序员, 在程序调试, 性能提升, 程序移植和健壮性等方面成为高手; 建立扎实的计算机系统概念, 为后续的OS, 编译, 体系结构等课程打下坚实基础.

实践是检验眞理的唯一标准

旷日持久的计算机教学只为解答三个问题：

- (theory, 理论计算机科学) 什么是计算?
- (system, 计算机系统) 什么是计算机?
- (application, 计算机应用) 我们能用计算机做什么?

除了纯理论工作之外, 计算机相关的工作无不强调动手实践的能力. 很多时候, 你会觉得理解某一个知识点是一件简单是事情, 但当你眞正动手实践的时候, 你才发现你的之前的理解只是停留在表面. 例如你知道链表的基本结构, 但你能写出一个正确的链表程序吗? 你知道程序加载的基本原理, 但你能写一个加载器来加载程序吗? 你知道编译器, 操作系统, CPU的基本功能, 但你能写一个编译器, 操作系统, CPU吗? 你甚至会发现, 虽然你在程序设计课上写过很多程序, 但你可能连下面这个看似很简单的问题都无法回答:

终极拷问

当你运行一个Hello World程序的时候, 计算机究竟做了些什么?

很多东西说起来简单, 但做起来却不容易, 动手实践会让你意识到你对某些知识点的一知半解, 同时也给了你深入挖掘其中的机会, 你会在实践过程中发现很多之前根本没有想到过的问题(其实科研也是如此), 解决这些问题反过来又会加深你对这些知识点的理解. 理论知识和动手实践相互促进, 最终达到对知识点透彻的理解.

目前也有以下观点:

> 目前像VS, Eclipse这样的IDE功能都十分强大, 点个按钮就能编译, 拖动几个控件就能设计一个GUI程序, 为什么还需要学习程序运行的机理?
>
> PhotoShop里面的滤镜功能繁多, 随便点点就能美化图片, 为什么还需要学习图像处理的基本原理?

像"GUI程序开发", "PhotoShop图片美化"这样的工作也确实需要动手实践, 但它们并不属于上文提到的计算机应用的范畴, 也不是计算机本科教育的根本目的, 因为它们强调的更多是技能的培训, 而不是对"计算机能做什么"这个问题的探索, 这也是培训班教学和计算机本科教学的根本区别. 但如果你对GUI程序运行的机理了如指掌, 对图像处理基本原理的理解犹如探囊取物, 上述工作对你来说根本就不在话下, 甚至你还有能力参与Eclipse和PhotoShop的开发.

而对这些原理的透彻理解, 离不开动手实践.

宋公语录

学汽车制造专业是要学发动机怎么设计, 学开车怎么开得过司机呢?

Linux入门教程

Linux入门教程

以下内容引用自jyy的操作系统实验课程网站(http://cslab.nju.edu.cn/opsystem), 并有少量修改和补充. 如果你是第一次使用Linux, 请你一边仔细阅读教程, 一边尝试运行教程中提到的命令.

探索命令行

Linux命令行中的命令使用格式都是相同的:

```
命令名称 参数1 参数2 参数3 ...
```

参数之间用任意数量的空白字符分开. 关于命令行, 可以先阅读一些基本常识. 然后我们介绍最常用的一些命令:

- `ls` 用于列出当前目录(即"文件夹")下的所有文件(或目录). 目录会用蓝色显示. `ls -l` 可以显示详细信息.
- `pwd` 能够列出当前所在的目录.
- `cd DIR` 可以切换到 `DIR` 目录. 在Linux中, 每个目录中都至少包含两个目录: `.` 指向该目录自身, `..` 指向它的上级目录. 文件系统的根是 `/`.
- `touch NEWFILE` 可以创建一个内容为空的新文件 `NEWFILE`, 若 `NEWFILE` 已存在, 其内容不会丢失.
- `cp SOURCE DEST` 可以将 `SOURCE` 文件复制为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件复制到该目录下.
- `mv SOURCE DEST` 可以将 `SOURCE` 文件重命名为 `DEST` 文件; 如果 `DEST` 是一个目录, 则将 `SOURCE` 文件移动到该目录下.
- `mkdir DIR` 能够创建一个 `DIR` 目录.
- `rm FILE` 能够删除 `FILE` 文件; 如果使用 `-r` 选项则可以递归删除一个目录. 删除后的文件无法恢复, 使用时请谨慎!
- `man` 可以查看命令的帮助. 例如 `man ls` 可以查看 `ls` 命令的使用方法. 灵活应用 `man` 和互联网搜索, 可以快速学习新的命令.

下面给出一些常用命令使用的例子, 你可以键入每条命令之后使用 `ls` 查看命令执行的结果:

```
$ mkdir temp          # 创建一个目录temp
$ cd temp              # 切换到目录temp
$ touch newfile          # 创建一个空文件newfile
$ mkdir newdir         # 创建一个目录newdir
$ cd newdir              # 切换到目录newdir
$ cp ../newfile .     # 将上级目录中的文件newfile复制到当前目录下
$ cp newfile aaa     # 将文件newfile复制为新文件aaa
$ mv aaa bbb         # 将文件aaa重命名为bbb
$ mv bbb ..              # 将文件bbb移动到上级目录
$ cd ..                  # 切换到上级目录
$ rm bbb             # 删除文件bbb
$ cd ..                  # 切换到上级目录
$ rm -r temp         # 递归删除目录temp
```

消失的cd

上述各个命令除了 cd 之外都能找到它们的manpage. 这是为什么? 如果你思考后仍然感到困惑, 试着到互联网上寻找答案.

man 的功能不仅限于此. man 后可以跟两个参数, 可以查看不同类型的帮助(请在互联网上搜索). 例如当你不知道C标准库函数 freopen 如何使用时, 可以键入命令

```
man 3 freopen
```

学会使用man

如果你是第一次使用 man , 请阅读这里. 这个教程除了说明如何使用 man 之外, 还会教你在使用一款新的命令行工具时如何获得帮助.

更多的命令行知识

仅仅了解这些最基础的命令行知识是不够的. 通常, 我们可以抱着如下的信条: 只要我们能想到的, 就一定有方便的办法能够办到. 因此当你想要完成某件事却又不知道应该做什么的时候, 请向Google求助.

如果你想以Linux作为未来的事业, 那就可以去图书馆或互联网上找一些相关的书籍来阅读.

统计代码行数

第一个例子是统计一个目录中(包含子目录)中的代码行数. 如果想知道当前目录下究竟有多少行的代码, 就可以在命令行中键入如下命令:

```
find . | grep '\.c$\|\.h$' | xargs wc -l
```

如果用 man find 查看 find 操作的功能, 可以看到 find 是搜索目录中的文件. Linux中一个点 . 始终表示Shell当前所在的目录, 因此 find . 实际能够列出当前目录下的所有文件. 如果在文件很多的地方键入 find . ,将会看到过多的文件, 此时可以按 CTRL + c 退出.

同样, 用 man 查看 grep 的功能——"print lines matching a pattern". grep 实现了输入的过滤, 我们的 grep 有一个参数, 它能够匹配以 .c 或 .h 结束的文件. 正则表达式是处理字符串非常强大的工具之一, 每一个程序员都应该掌握其相关的知识. 有兴趣的同学可以首先阅读一个基础的教程, 然后看一个有趣的小例子: 如何用正则表达式判定素数. 正则表达式还可以用来编写一个30行的java表达式求值程序(传统方法几乎不可能), 聪明的你能想到是怎么完成的吗? 上述的 grep 命令能够提取所有 .c 和 .h 结尾的文件.

刚才的 find 和 grep 命令, 都从标准输入中读取数据, 并输出到标准输出. 关于什么是标准输入输出, 请参考这里. 连接起这两个命令的关键就是管道符号 | . 这一符号的

左右都是Shell命令, A | B 的含义是创建两个进程 A 和 B, 并将 A 进程的标准输出连接到 B 进程的标准输入. 这样, 将 find 和 grep 连接起来就能够筛选出当前目录(.)下所有以 .c 或 .h 结尾的文件.

我们最后的任务是统计这些文件所占用的总行数, 此时可以用 man 查看 wc 命令. wc 命令的 -l 选项能够计算代码的行数. xargs 命令十分特殊, 它能够将标准输入转换为参数, 传送给第一个参数所指定的程序. 所以, 代码中的 xargs wc -l 就等价于执行 wc -l aaa.c bbb.c include/ccc.h ..., 最终完成代码行数统计.

统计磁盘使用情况

以下命令统计 /usr/share 目录下各个目录所占用的磁盘空间:

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

du 是磁盘空间分析工具, du -sc 将目录的大小顺次输出到标准输出, 继而通过管道传送给 sort . sort 是数据排序工具. 其中的选项 -n 表示按照数值进行排序, 而 -r 则表示从大到小输出. sort 可以将这些参数连写在一起.

然而我们发现, /usr/share 中的目录过多, 无法在一个屏幕内显示. 此时, 我们可以再使用一个命令: more 或 less .

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

此时将会看到输出的前几行结果. more 工具使用空格翻页, 并可以用 q 键在中途退出. less 工具则更为强大, 不仅可以向下翻页, 还可以向上翻页, 同样使用 q 键退出. 这里还有一个关于less的小故事.

在Linux下编写Hello World程序

Linux中用户的主目录是 /home/用户名称 , 如果你的用户名是 user , 你的主目录就是 /home/user . 用户的 home 目录可以用波浪符号 ~ 替代, 例如临时文件目录 /home/user/Templates 可以简写为 ~/Templates . 现在我们就可以进入主目录并编辑文件了. 如果 Templates 目录不存在, 可以通过 mkdir 命令创建它:

```
cd ~
mkdir Templates
```

创建成功后, 键入

```
cd Templates
```

可以完成目录的切换. 注意在输入目录名时, tab 键可以提供联想.

你感到键入困难吗

你可能会经常要在终端里输入类似于

```
cd AVeryVeryLongFileName
```

的命令, 你一定觉得非常烦躁. 回顾上面所说的原则之一: 如果你感到有什么地方不对, 就一定有什么好办法来解决. 试试 tab 键吧.

Shell中有很多这样的小技巧, 你也可以使用其他的Shell例如zsh, 提供更丰富好用的功能. 总之, 尝试和改变是最重要的.

进入正确的目录后就可以编辑文件了, 开源世界中主流的两大编辑器是 vi(m) 和 emacs , 你可以使用其中的任何一种. 如果你打算使用 emacs , 你还需要安装它

```
apt-get install emacs
```

vi 和 emacs 这两款编辑器都需要一定的时间才能上手, 它们共同的特点是需要花较多的时间才能适应基本操作方式(命令或快捷键), 但一旦熟练运用, 编辑效率就比传统的编辑器快很多.

进入了正确的目录后, 输入相应的命令就能够开始编辑文件. 例如输入

```
vi hello.c
或emacs hello.c
```

就能开启一个文件编辑. 例如可以键入如下代码(对于首次使用 vi 或 emacs 的同学, 键入代码可能会花去一些时间, 在编辑的同时要大量查看网络上的资料):

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

保存后就能够看到 hello.c 的内容了. 终端中可以用 cat hello.c 查看代码的内容. 如果要将它编译, 可以使用 gcc 命令:

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

gcc 的 -o 选项指定了输出文件的名称, 如果将 -o hello 改为 -o hi , 将会生成名为 hi 的可执行文件. 如果不使用 -o 选项, 则会默认生成名为 a.out 的文件, 它的含义是assembler output. 在命令行输入

```
./hello
```

就能够运行改程序. 命令中的 ./ 是不能少的, 点代表了当前目录, 而 ./hello 则表示当前目录下的 hello 文件. 与Windows不同, Linux系统默认情况下并不查找当前目录, 这是因为Linux下有大量的标准工具(如 test 等), 很容易与用户自己编写的程序重名, 不搜索当前目录消除了命令访问的歧义.

使用重定向

有时我们希望将程序的输出信息保存到文件中, 方便以后查看. 例如你编译了一个程序 myprog , 你可以使用以下命令对 myprog 进行反汇编, 并将反汇编的结果保存到 output 文件中:

```
objdump -d myprog > output
```

> 是标准输出重定向符号, 可以将前一命令的输出重定向到文件 output 中. 这样, 你就可以使用文本编辑工具查看 output 了.

但你会发现, 使用了输出重定向之后, 屏幕上就不会显示 myprog 输出的任何信息. 如果你希望输出到文件的同时也输出到屏幕上, 你可以使用 tee 命令:

```
objdump -d myprog | tee output
```

使用输出重定向还能很方便地实现一些常用的功能, 例如

```
> empty               # 创建一个名为empty的空文件
cat old_file > new_file     # 将文件old_file复制一份, 新文件名为new_f
```

如果 myprog 需要从键盘上读入大量数据(例如一个图的拓扑结构), 当你需要反复对 myprog 进行测试的时候, 你需要多次键入大量相同的数据. 为了避免这种无意义的重复键入, 你可以使用以下命令:

```
./myprog < data
```

< 是标准输入重定向符号, 可以将前一命令的输入重定向到文件 data 中. 这样, 你只需要将 myprog 读入的数据一次性输入到文件 data 中, myprog 就会从文件 data 中读入数据, 节省了大量的时间.

下面给出了一个综合使用重定向的例子:

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

这个命令在运行 myprog 的同时, 指定其从文件 data 中读入数据, 并将其输出信息打印到屏幕和文件 output 中. time 工具记录了这一过程所消耗的时间, 最后你会在屏幕上看到 myprog 运行所需要的时间. 如果你只关心 myprog 的运行时间, 你可以使用以下命令将 myprog 的输出过滤掉:

```
time ./myprog < data > /dev/null
```

/dev/null 是一个特殊的文件, 任何试图输出到它的信息都会被丢弃, 你能想到这是怎么实现的吗? 总之, 上面的命令将 myprog 的输出过滤掉, 保留了 time 的计时结果, 方便又整洁.

使用Makefile管理工程

大规模的工程中通常含有几十甚至成百上千个源文件(Linux内核源码有25000+的源文件), 分别键入命令对它们进行编译是十分低效的. Linux提供了一个高效管理工程文件的工具: GNU Make. 我们首先从一个简单的例子开始, 考虑上文提到的Hello World的例子, 在 hello.c 所在目录下新建一个文件 Makefile , 输入以下内容并保存:

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

.PHONY: clean

clean:
    rm hello     # 注意开头的tab, 而不是空格
```

返回命令行, 键入 make , 你会发现 make 程序调用了 gcc 进行编译. Makefile 文件由若干规则组成, 规则的格式一般如下:

```
目标文件名:依赖文件列表
    用于生成目标文件的命令序列      # 注意开头的tab, 而不是空格
```

我们来解释一下上文中的 hello 规则. 这条规则告诉 make 程序, 需要生成的目标文件是 hello , 它依赖于文件 hello.c , 通过执行命令 gcc hello.c -o hello 来生成 hello 文件.

如果你连续多次执行 make , 你会得到"文件已经是最新版本"的提示信息, 这是 make 程序智能管理的功能. 如果目标文件已经存在, 并且它比所有依赖文件都要"新", 用于生成目标的命令就不会被执行. 你能想到 make 程序是如何进行"新"和"旧"的判断的吗?

上面例子中的 clean 规则比较特殊, 它并不是用来生成一个名为 clean 的文件, 而是用于清除编译结果, 并且它不依赖于其它任何文件. make 程序总是希望通过执行命令来生成目标, 但我们给出的命令 rm hello 并不是用来生成 clean 文件, 因此这样的命令总是会被执行. 你需要键入 make clean 命令来告诉 make 程序执行 clean 规则, 这是因为 make 默认执行在 Makefile 中文本序排在最前面的规则. 但如果很不幸地, 目录下已经存在了一个名为 clean 的文件, 执行 make clean 会得到"文件已经是最新版本"的提示. 解决这个问题的方法是在 Makefile 中加入一行 PHONY: clean , 用于指示" clean 是一个伪目标", 这样以后, make 程序就不会判断目标文件的新旧, 伪目标相应的命令序列总是会被执行.

对于一个规模稍大一点的工程, Makefile 文件还会使用变量, 函数, 调用Shell命令, 隐含规则等功能. 如果你希望学习如何更好地编写一个 Makefile , 请到互联网上搜索相关资料.

综合示例: 教务刷分脚本

使用编辑器编辑文件 jw.sh 为如下内容(如果显示不正常, 可以复制到文本编辑器中查看. 另外由于教务网站的升级改版, 目前此脚本可能不能实现正确的功能):

```
#!/bin/bash
save_file="score" # 临时文件
semester=20102 # 刷分的学期, 20102代表2010年第二学期
jw_home="http://jwas3.nju.edu.cn:8080/jiaowu" # 教务网站首页地址
jw_login="http://jwas3.nju.edu.cn:8080/jiaowu/login.do" # 登录页面
jw_query="http://jwas3.nju.edu.cn:8080/jiaowu/student/studentinf

name="09xxxxxxx" # 你的学号
passwd="xxxxxxxx" # 你的密码

# 请求jw_home地址, 并从中找到返回的cookie. cookie信息在http头中的JSESS
cookie=`wget -q -O - $jw_home --save-headers | \
    sed -n 's/Set-Cookie: JSESSIONID=\([0-9A-Z]\+\);.*$/\1/p'`
# 用户登录, 使用POST方法请求jw_login地址, 并在POST请求中加入userName和p
wget -q -O - --header="Cookie:JSESSIONID=$cookie" --post-data \
    "userName=${name}&password=${passwd}" "$jw_login" &> /dev/nu
# 登录完毕后, 请求分数查询页面. 此时会返回html页面并输出到标准输出. 我们将
wget -q -O - --header="Cookie:JSESSIONID=$cookie" "$jw_query" >
# 获取分数列表. 因为教务网站的代码实在是实现得不太规整, 我们又想保留shell的
list=`cat tmp | sed -n '/<table.*TABLE_BODY.*>/,/<\/table>/p' \
        | sed '/<--/,/-->/d' | grep td \
```

```
        | awk 'NR%11==3' | sed 's/^.*>\(.*\)<.*$/\1/g'`
# 对list中的每一门课程, 都得到它的分数
for item in $list; do
    score= `cat tmp | grep -A 20 $item | awk "NR==18" | sed -n '
    score= `echo $score`
    if [[ ${#score} != 0 ]]; then # 如果存在成绩
        grep $item $save_file &>/dev/null # 查找分数是否显示过
        if [[ $? != 0 ]]; then # 如果没有显示过
        # 考虑到尝试的同学可能没有安装notify-send工具, 这里改成echo  --
            # notify-send "新成绩：$item $score" # 弹出窗口显示新成绩
            echo "新成绩：$item $score" # 在终端里输出新成绩
            echo $item >> $save_file # 将课程标记为已显示
        fi
    fi
done
```

运行这个例子需要在命令行中输入 `bash jw.sh`, 用bash解释器执行这一脚本. 如果希望定期运行这一脚本, 可以使用Linux的标准工具之一: `cron`. 将命令添加到crontab就能实现定期自动刷新.

为了理解这个例子, 首先需要一些HTTP协议的基础知识. HTTP请求实际就是来回传送的文本流——浏览器(或我们例子中的爬虫)生成一个文本格式的HTTP请求, 包括header和content, 以文本的形式通过网络传送给服务器. 服务器根据请求内容(header中包含请求的URL以及浏览器等其他信息), 生成页面并返回.

用户登录的实现, 就是通过HTTP头中header中的cookie实现的. 当浏览器第一次请求页面时, 服务器会返回一串字符, 用来标识浏览器的这次访问. 从此以后, 所有与该网站交互时, 浏览器都会在HTTP请求的header中加入这个字符串, 这样服务器就"记住"了浏览器的访问. 当完成登录操作(将用户名和密码发送到服务器)后, 服务器就知道这个cookie隐含了一个合法登录的帐号, 从而能够根据帐号信息发送成绩.

得到包含了成绩信息的html文档之后, 剩下的事情就是解析它了. 我们用了大量的 `sed` 和 `awk` 完成这件事情, 同学们不用去深究其中的细节, 只需知道我们从文本中提取出了课程名和成绩, 并且将没有显示过的成绩显示出来.

我们讲解这个例子主要是为了说明新环境下的工作方式, 以及实践Unix哲学:

- 每个程序只做一件事, 但做到极致
- 用程序之间的相互协作来解决复杂问题
- 每个程序都采用文本作为输入和输出, 这会使程序更易于使用

一个Linux老手可以用脚本完成各式各样的任务: 在日志中筛选想要的内容, 搭建一个临时HTTP服务器(核心是使用 `nc` 工具)等等. 功能齐全的标准工具使Linux成为工程师, 研究员和科学家的最佳搭档.

man入门教程

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! 还是得认眞做实验, 不然以后连码农都当不上了! 这样的想法鞭策着你, 因为你知道, 就算是码农, 也要有适应新环境和掌握新工具的能力. "还是先去找man吧." 于是你在终端里输入 `man` , 敲了回车. 只见屏幕上输出了一行信息:

> What manual page do you want?

噢, 原来命令行也会说人话! 你明白这句话的意思, `man` 在询问你要查询什么内容. 你能查询什么内容呢? 既然 `man` 会说人话, 还是先多了解 `man` 吧. 为了告诉 `man` 你想更了解ta, 你输入

```
man man
```

敲了回车之后, `man` 把你带到了一个全新的世界. 这时候, 你又看到了一句人话了, 那是 `man` 的独白, ta告诉你, ta的眞实身份其实是

> an interface to the on-line reference manuals

接下来, ta忽然说了一大堆你听不懂的话, 似乎是想告诉你ta的使用方法. 可是你还没做好心理准备啊, 于是你无视了这些话.

寻找帮助

很快, 你已经看到"最后一行"了. 难道man的世界就这么狭小? 你仔细一看, "最后一行"里面含有一些信息:

> Manual page man(1) line 1 (press h for help or q to quit)

原来可以通过按 `q` 来离开这个世界啊, 不过你现在并不想这么做, 因为你想多了解 `man` , 以后可能会经常需要 `man` 的帮助. 为了更了解ta, 你按了 `h` .

这时你又被带到了新的世界, 世界的起点是"SUMMARY OF LESS COMMANDS", 你马上知道, 这个世界要告诉你如何使用 `man` , 你十分激动. 于是你往下看, 这句话说"带有'*'标记的命令可以在前面跟一个数, 这时命令的行为在括号里给出". 这是什么意思? 你没看懂, 还是找个带'*'的命令试试吧. 你继续往下看, 看到了两个功能和相应的命令:

- 第一个是展示帮助, 原来除了 `h` 之外, `H` 也可以看到帮助, 而且这里把帮助的命令放在第一个, 也许 `man` 想暗示你, 找到帮助是十分重要的.
- 第二个命令是退出. "哈哈, 知道怎么退出之后, 就不用通过重启来退出一个命令行程序啦", 你心想. 但你现在还是不想退出, 还是再看看其它的吧.

继续往下看, 你看到了用于移动的命令. 果然, 你还是可以在这个世界里面移动的. 第一个用于移动的功能是往下移动一行, 你看到有5种方法可以实现:

```
e  ^E  j  ^N  CR
```

e 和 j 你看懂了, 就是按 e 或者 j . 但 ^E 是什么意思呢? 你尝试找到 ^ 的含义, 但是你没找到, 还是让我告诉你吧. 在上下文和按键有关的时候, ^ 是Linux中的一个传统记号, 它表示 ctrl+ . 还记得Windows下 ctrl+c 代表复制的例子吗? 这里的 ^E 表示 ctrl+E . CR 代表回车键, 其实 CR 是控制字符(ASCII码小于32的字符)的一个, 这里有一段关于控制字符的问答.

你决定使用 j , 因为它像一个向下的箭头, 而且它是右手食指所按下的键. 其实这点和 vim 的使用是类似的, 如果你不能理解为什么 vim 中使用 h , j , k , l 作为方向键, 这里有一个初学者的提问, 事实上, 这是一种touch typing.

你按下了 j , 发现画面上的信息向下滚动了一行. 你看到了 * , 想起了 * 标记的命令可以在前面跟一个数. 于是你试着输入 10j , 发现画面向下滚动了10行, 你第一次感觉到在这个"丑陋"的世界中也有比GUI方便的地方. 你继续阅读帮助, 并且尝试每一个命令. 于是你掌握了如何通过移动来探索 man 所在的世界.

继续往下翻, 你看到了用于搜索的命令. 你十分感动, 因为使用关键字可以快速定位到你关心的内容. 帮助的内容告诉你, 通过按 / 激活前向搜索模式, 然后输入关键字(可以使用正则表达式), 按下回车就可以看到匹配的内容了. 帮助中还列出了后向搜索, 跳到下一匹配处等功能. 于是你掌握了如何使用搜索.

探索man

你一边阅读帮助, 一边尝试新的命令, 就这样探索着这个陌生的世界. 你虽然记不住这么多命令, 但你知道你可以随时来查看帮助. 掌握了一些基本的命令之后, 你按 q 离开了帮助, 回到了 man 的世界. 现在你可以自由探索 man 的世界了. 你向下翻, 跳过了看不懂的 SYNOPSIS 小节, 在 DESCRIPTION 小节看到了人话, 于是你阅读这些人话. 在这里, 你看到整个manual分成9大类, 每个manual page都属于其中的某一类; 你看到了一个manual page主要包含以下的小节:

- NAME - 命令名
- SYNOPSIS - 使用方法大纲
- CONFIGURATION - 配置
- DESCRIPTION - 功能说明
- OPTIONS - 可选参数说明
- EXIT STATUS - 退出状态, 这是一个返回给父进程的值
- RETURN VALUE - 返回值
- ERRORS - 可能出现的错误类型
- ENVIRONMENT - 环境变量
- FILES - 相关配置文件
- VERSIONS - 版本
- CONFORMING TO - 符合的规范
- NOTES - 使用注意事项
- BUGS - 已经发现的bug
- EXAMPLE - 一些例子
- AUTHORS - 作者
- SEE ALSO - 功能或操作对象相近的其它命令 你还看到了对 SYNOPSIS 小节中记号的解释, 现在你可以回过头来看 SYNOPSIS 的内容了. 但为了弄明白每个参数的含义, 你需要查看 OPTIONS 小节中的内容.

你想起了搜索的功能, 为了弄清楚参数 -k 的含义, 你输入 /-k , 按下回车, 并通过 n 跳过了那

些 OPTIONS 小节之外的 -k , 最后大约在第254行找到了 -k 的解释: 通过关键字来搜索相关功能的manual page. 在 EXAMPLES 小节中有一个使用 -k 的例子:

```
man -k printf
```

你阅读这个例子的解释: 搜索和 printf 相关的manual page. 你还是不太明白这是什么意思, 于是你退出 man , 在命令行中输入

```
man -k printf
```

并运行, 发现输出了很多和 printf 相关的命令或库函数, 括号里面的数字代表相应的条目属于manual的哪一个大类. 例如 printf (1) 是一个shell命令, 而 printf (3) 是一个库函数. 要访问库函数 printf 的manual page, 你需要在命令行中输入

```
man 3 printf
```

当你想做一件事的而不知道用什么命令的时候, man 的 -k 参数可以用来列出候选的命令, 然后再通过查看这些命令的manual page来学习怎么使用它们.

接下来, 你又开始学习 man 的其它功能...

开始旅程

到这里, 你应该掌握 man 的用法了. 你应该经常来拜访ta, 因为在很多时候, ta总能给你提供可靠的帮助.

在这个励志的故事中, 你学会了:

- 阅读程序输出的提示和错误信息
- 通过搜索来定位你关心的内容
- 动手实践是认识新事物的最好方法
- 独立寻找帮助, 而不是一有问题就问班上的大神

于是, 你就这样带着 man 踏上了Linux之旅...

git入门教程

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 edito
git config --global color.ui true
```

经过这些配置, 你就可以开始使用 `git` 了.

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

```
git init
```

进行初始化.

查看存档信息

使用

```
git log
```

查看目前为止所有的存档.

使用

```
git status
```

可以得知, 与当前存档相比, 哪些文件发生了变化.

存档

你可以像以前一样编写代码. 等到你的开发取得了一些阶段性成果, 你应该马上进行"存档".

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

```
git add file.c
```

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

```
git add -A
```

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

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

```
git commit
```

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

读档

如果你遇到了上文提到的让你悲痛欲绝的情况, 现在你可以使用光玉来救你一命了. 首先使用 `git log` 来查看已有的存档, 并决定你需要回到哪个过去. 每一份存档都有一个hash code, 例如 `b87c512d10348fd8f1e32ddea8ec95f87215aaa5` , 你需要通过hash code来告诉 `git` 你希望读哪一个档. 使用以下命令进行读档:

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

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

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

第三视点

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

```
git branch
```

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

读档的时候使用以下命令

```
git checkout b87c
```

而不是 `git reset` . 这时你将处于一个虚构的分支中, 你可以

- 查看 `b87c` 存档的内容
- 使用以下命令切换到其它分支

  ```
  git checkout 分支名
  ```

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

  ```
  git checkout -B 分支名
  ```

  把修改结果保存到一个新的分支中, 如果分支已存在, 其内容将会被覆盖

不同的分支之间不会相互干扰, 这也给项目的分布式开发带来了便利. 有了分支功能, 你就可以像第三视点那样在一个世界的不同时间(一个分支的多个存档), 或者是多个平行世界(多个分支)之间来回穿梭了.

更多功能

以上介绍的是 `git` 的一些基本功能, `git` 还提供很多强大的功能, 例如使用 `git diff` 比较同一个文件在不同版本中的区别, 使用 `git bisect` 进行二分搜索来寻找一个bug在哪次提交中被引入...

其它功能的使用请参考 `git help` , `man git` , 或者在网上搜索相关资料.

i386手册勘误

i386手册勘误

- 17.2.1 ModR/M and SIB Bytes中的Table 17-3:

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

- 17.2.1 ModR/M and SIB Bytes中的Table 17-4:

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

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

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

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

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

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

```
@@ -?,2 +?,2 @@
 72  cb         JB rel8           7+m,3    Jump short if below (CF=1
-76  cb         JBE rel8          7+m,3    Jump short if below or (CF
+76  cb         JBE rel8          7+m,3    Jump short if below or equ
@@ -?,2 +?,2 @@
```

```
 7C  cb         JL rel8           7+m,3    Jump short if less (SF!=OI
-7E  cb         JLE rel8          7+m,3    Jump short if less or equa
+7E  cb         JLE rel8          7+m,3    Jump short if less or equa
@@ -?,2 +?,2 @@
 0F  8C cw/cd   JL rel16/32       7+m,3    Jump near if less (SF!=OF
-0F  8E cw/cd   JLE rel16/32      7+m,3    Jump near if less or equa
+0F  8E cw/cd   JLE rel16/32      7+m,3    Jump near if less or equa
```

- 17.2.2.11 Instruction Set Detail中的MOV -- Move Data

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

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

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

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

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

- 17.2.2.11 Instruction Set Detail中的SETcc - Byte Set on Condition

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

```
 0F  9D   SETNL r/m8   4/5     Set byte if not less (SF=OF)
-0F  9F   SETNLE r/m8  4/5     Set byte if not less or equal (ZF=1 a
+0F  9F   SETNLE r/m8  4/5     Set byte if not less or equal (ZF=1 a
```

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

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

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

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