PA0 - 世界诞生的前夜: 开发环境配置

世界诞生的故事 - 序章

PA讲述的是一个"先驱创造计算机"的故事.

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

! 提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 10小时

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

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

信息框说明

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

实验进度相关的提示

扩展阅读

选做思考题

选做编程题

实验必做内容

实验进度相关的必读信息

重要性超越实验的原则与方法

对, 你没有看错, 除了一些重要的信息之外, PA0的实验讲义都是英文!

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

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

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

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

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

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

We are going to install the Ubuntu 21.04 distribution directly over your physical machine. If you already have one copy of GNU/Linux distribution, and you want to use your copy as the development environment, just use it! But if you encounter some troubles because of platform disparity, please search the Internet for trouble-shooting.

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

必须使用64位的GNU/Linux

如果你打算使用已有的GNU/Linux平台, 请确保它是64位版本. PA的某些特性会依赖于64位平台.

使用GNU/Linux真机的一些原因

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

建议体验真机的安装

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

Mac用户的不幸

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

如果你不必修读南京大学的课程, 可以忽略这个问题.

Preparation

首先当然是备份重要数据!

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

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

Please reserve at least one partitions (20GB ~ 50GB) on the disk for Ubuntu to install. If you want to install `Vivado/Quartus` in Ubuntu, please reserve about 100GB of disk space. For how to do this, please search the Internet. For example, if you use Windows, you can search for `压缩卷 Windows` .

Installing Ubuntu

Please search the Internet for "Ubuntu 21.04 安装教程" and follow the tutorial.

我可以选择其它Ubuntu的版本吗?

可以. 不过不同版本的Ubuntu中的工具版本会稍有差异, 尤其是编译器GCC. 如果你选择了其它版本, 可能会遇到因此带来的少量问题, 不过这不会对实验内容产生大幅的影响.

选择语言时选择English, 不要选择中文

使用中文的系统环境会给一些命令行操作带来不便, 甚至会增加排查错误的难度. 每年都有部分同学因为选择了中文环境, 在工具报错时无法在互联网上搜索到有效的解决方案, 从而影响了实验进度.

事实上, 即使在英文的系统环境中, 也可以通过安装中文输出法来键入中文.

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

First Exploration with GNU/Linux

Open `terminal`, you will see the following prompt:

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

This prompt shows your username, host name, and the current working directory. The username should be the same as you set during the installation. The current working directory is `~` now. As you switching to another directory, the prompt will change as well. You are going to code under this environment, so try to make friends with terminal!

Where is GUI?

Many of you always use operating system with GUI, such as Windows. But the terminal is completely with CLI (Command Line Interface). Have you wondered if there is something that you can do it in CLI, but can not in GUI? Have no idea? If you are asked to count how many lines of code you have coded during the 程序设计基础 course, what will you do?

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

GUI wins when you do something requires high definition displaying, such as watching movies. Here is an article discussing the comparision between GUI and CLI.

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

```
df -h
```

To shut down the system, issue the following command:

```
poweroff
```

However, you will receive an error message:

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

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

Why executing the "poweroff" command requires superuser privilege?

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

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

```
su -
```

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

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

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

不要强制关闭虚拟机!!!

如果你使用虚拟机, 你务必通过 `poweroff` 命令关闭虚拟机. 如果你通过点击窗口右上角的 `X` 按钮强制关闭虚拟机, 可能会造成虚拟机中文件损坏的现象. 往届有若干学长因此而影响了实验进度, 甚至由于损坏了实验相关的文件而影响了分数. 请大家引以为鉴, 不要贪图方便, 否则后果自负!

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

导航

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

Installing Tools

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

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

Checking network state

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

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

You should receive reply packets successfully:

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

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

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

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

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

Setting APT source file

Run the following commands to update the APT source file:

```
bash -c 'echo "deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ hirsute main restricted uni
```

! 如果你的系统不是Ubuntu 21.04, 请更换合适的源

使用与系统不相符的源会导致工具包版本冲突, 强行安装将会损坏系统. 如果你使用的系统不是Ubuntu 21.04, 请勿使用上述的源, 你需要自行搜索合适的源并更换.

However, you will receive an error message:

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

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

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

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

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

```
whoami
sudo whoami
```

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

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

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

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

Updating APT package information

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

```
apt-get update
```

However, you will receive an error message:

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

This is because `apt-get` requires superuser privilege to run. Run the command with superuser privilege. Since it requires Internet accessing, it may cost some time to finish.

Installing tools for PAs

The following tools are necessary for PAs:

```
apt-get install build-essential    # build-essential packages, include binary utilities, g
apt-get install man                # on-line reference manual
apt-get install gcc-doc            # on-line reference manual for gcc
apt-get install gdb                # GNU debugger
apt-get install git                # revision control system
apt-get install libreadline-dev    # a library used later
apt-get install libsdl2-dev        # a library used later
apt-get install llvm               # llvm project, which contains libraries used later
```

The usage of these tools is explained later.

Installing Chinese input method

Search the Internet for how to install a Chinese input method in Ubuntu.

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

≡ 导航
Learning vim
Enabling syntax highlight
Enabling more vim features

Configuring vim

```
apt-get install vim
```

`vim` is called 编辑器之神. You will use `vim` for coding in all PAs and Labs, as well as editing other files. If you prefer to `emacs`, you can download and install `emacs` from network mirrors.

Learning vim

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

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

```
vim test
```

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

🤔 为什么上课不讲GNU/Linux的使用？

你可能会想：这是我第一次接触GNU/Linux，为什么上课不讲讲怎么用？

因为说明书不是用来讲的，是用来一边看一边操作的，你也不能光靠课听讲说来掌握这些工具，而是要自己去动手尝试，你在大学课堂上应该学习到的是那些一般规律的知识，然后去思考这些知识背后的原因和意想，将来有着力将这些原则和思想应用到新的领域。

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

💬 Some games operated with vim

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

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

💬 The power of vim

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

The first example is to generate the following file:

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

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

```
1i<ESC>ii<ctrl-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:

```
1i<ESC>ii<ctrl-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:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee
gggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
```

You want to modify it into:

```
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh
jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
```

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>2414jdSp
```

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>24199jdSp
```

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

Enabling syntax highlight

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

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

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

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

```
cd ~
```

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

```
ls -a
```

Then open `.vimrc` using `vim`:

```
vim .vimrc
```

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

```
--- before modification
+++ after modification
@@ -37,3 +37,3 @@
 " VimS and later versions support syntax highlighting. Uncommenting the next
 " line enables syntax highlighting by default.
-"syntax on
+syntax on
```

We present the modification with GNU diff format. If you do not understand the diff format, please search the Internet for more information.

🤔 为什么要STFW？

你或许会想：我问别人最为了节省我的时间。

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

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

正确的STFW方式能够增加找到解决方案的概率，包括

- 使用Google搜索引擎要一般性问题
- 使用英文维基百科查阅概念
- 使用stack overflow问答网站搜索编程语言相关问题

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

🤔 为什么不要用百度？

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

- 找不到相关结果，你感到挫败
- 搜到看似相关的结果，但无法解决问题，你在感到挫败之余，也发现自己浪费了不少时间
- 你搜到了解决问题的方案，但没有发现原因分析，结果你不知道这个问题背后的细节

你可能会觉得“可以解决问题就行，不需要了解问题背后的细节”，但对于一些问题例如编程问题，你了解这些细节就能日子使用了的时间，所以你应该去了解这些细节，让自己懂得更多。

如果你歌唱记录周的概率提供可以解决问题的方案，并且带有原因分析，你在这没有理由使用百度来搜索技术问题，如果你仍然坚持使用百度，原因就只有一个：你不想主动去成长。

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

🤔 为什么要这么麻烦？

搞了半天，你发现其实也就是改动一个字母而已，为什么不直接说清楚呢？

这是为了“人多操心”：我们希望你了解怎么用计算机思维精简准确地表达我们想要的事情. diff格式是一种描述文件改动的常用方式. 实际上，计算机的世界里面有很多的定位语的“规矩”，当你慢慢去接触去了解这些规矩的时候，你就会在不知不觉中明白计算机世界是怎么运转的.

Enabling more vim features

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

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

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

```
setlocal noswapfile " 不要生成swap文件
set bufhidden=hide " 当buffer被丢弃的时候隐藏它
colorscheme evening " 设定配色方案
set number " 显示行号
set cursorline " 突出显示当前行
set ruler " 打开状态栏标尺
set shiftwidth=4 " 设定 << 和 >> 命令移动的时宽度为 4
set softtabstop=4 " 使得按退格键时可以一次删除 4 个空格
set tabstop=4 " 设定 tab 长度为 4
set nobackup " 覆盖文件时不备份
set autochdir " 自动切换当前目录为当前文件所在的目录
set backupcopy=yes " 设置备份时的行为为覆盖
set hlsearch " 搜索时高亮显示被找到的文本
set noerrorbells " 关闭错误信息响铃
set novisualbell " 关闭使用可视响铃代替呼叫
set t_vb= " 置空错误铃声的终端代码
set matchtime=2 " 短暂跳转到匹配括号的时间
set magic " 设置魔术
set smartindent " 开启新行时使用智能自动缩进
set backspace=indent,eol,start " 不设定在插入状态无法用退格键和 Delete 键删除回车符
set cmdheight=1 " 设定命令行的行数为 1
set laststatus=2 " 显示状态栏 (默认值为 1, 无法显示状态栏)
set statusline\ %<%f\%h%m%r%=%b\%B\ %l,%v\ %6\*%L%{if(&fileformat!\|\|&encoding!\|\|exists("+bomb"),"[".(&ff!\|\|&fileformat!\|\|exists("+bomb")?&ff:"")."]",)}
set foldenable " 开始折叠
set foldmethod=syntax " 设置语法折叠
set foldcolumn=0 " 设置折叠区域的宽度
setlocal foldlevel=1 " 设置折叠层数为 1
nnoremap <space> @=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')<CR> " 用空格键来开关折叠
```

🤔 提高开发效率的编程技巧

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

在GNU/Linux中, 与记事本的操作方式相近, 学习 `vim` 的基本操作就已经可以大大提高开发效率. 还有各种插件来增强 `vim` 的功能, 比如可以在代码中空格键执行 `rsync` 等等. 你可以花点时间去配置一下 `vim`，具体配置方式请STFW. 总之, “编辑器之神”可不是浪费得虚名的.

Getting Source Code for PAs

Getting Source Code

Go back to the home directory by

```
cd ~
```

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

不要使用root用户来做实验

使用root用户来做实验会带来很多不方便的麻烦。可能会导致文件属权或者无法显示等。所以，在开始之前，你的操作环境应该是普通用户的文件夹。特别是对无法显示的问题或者不清楚出现原因的情况下，更应该使用不同的操作系统用户创建一个新用户来尝试。这有助于定位问题。

如果你的用户名解析出现问题，也可以尝试设置下变量：Why is it bad to login as root? 之类的解答。这一方面可以帮助你排除与操作系统用户有关的麻烦问题。如果需要进一步调试gdb或strace等工具的使用，请参阅 `sudo`。

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

```
git clone -b 2021 https://github.com/AScX-Projects/ics-pa-gzb ics2021
```

A directory called `ics2021` will be created. This is the project directory for PAs. Details will be explained in PA1.

Then issue the following commands to perform `git configuration`

```
git config --global user.name "2022XXXXX-Zhang San" # your student ID and name
git config --global user.email "zhangsan@qq.com"    # your email
git config --global core.editor vim                 # your favorite editor
git config --global color.ui true                   # enable color output
```

You should configure `git` with your student ID, name, and email. Before continuing, please read this `git` tutorial to learn some basics of `git`. Another material recommended by us is Visualizing Git Concepts with D3. You can learn some `git` commands with the help of visualization.

Enter the project directory `ics2021`, then run:

```
git branch -a master
back ics2021.mmap
back ics2021.abstract-machine
```

to initialize some subprojects. The script will pull some subprojects from gitlab. We will explain them later.

Besides, the script will also add some environment variables into the bash configuration file `~/.bashrc`. These variables are defined by sourcing `~/.bashrc` during the compilation of the subprojects. Therefore, DO NOT move your project to another directory once finishing the initialization, else these variables will become invalid. Particularly, if you see shell other than `bash`, please set these variables in the corresponding configuration file manually.

To let the environment variables take effect, run

```
source ~/.bashrc
```

Then try

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

to initialize some subprojects. The script will pull some subprojects from gitlab. We will explain them later.

Besides, the script will also add some environment variables into the bash configuration file `~/.bashrc`. These variables are defined by sourcing `~/.bashrc` during the compilation of the subprojects. Therefore, DO NOT move your project to another directory once finishing the initialization, else these variables will become invalid. Particularly, if you see shell other than `bash`, please set these variables in the corresponding configuration file manually.

To let the environment variables take effect, run

```
source ~/.bashrc
```

Then try

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

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

Git usage

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

You can list all branches by

```
git branch
```

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

```
* master
```

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

```
git checkout -b pa0
```

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

```
master
* pa0
```

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

Now have a try! Modify the `STUID` and `STUNAME` variables in `nemu/scripts/pa.mk`:

```
STUID = 2022100000 # your student ID
STUNAME = Bill     # your Chinese name
```

Run

```
git status
```

to see those files modified from the last commit:

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

    modified:   nemu/scripts/pa.mk

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

Run

```
git diff
```

to list modifications from the last commit:

```
diff --git a/nemu/scripts/pa.mk b/nemu/scripts/pa.mk
index c9e1f8b..a7b2a42 100644
--- a/nemu/scripts/pa.mk
+++ b/nemu/scripts/pa.mk
@@ -1,4 +1,4 @@
-STUID = 2022100000
-STUNAME = Bill
+STUID = 2022100001
+STUNAME = Jack
# DO NOT modify the following code!!!
```

You should see `STUID` and `STUNAME` are modified. Now add the changes to commit by `git add .` and issue `git commit`:

```
git add .
git commit
```

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

```
git log
```

Now switch back to the `master` branch by

```
git checkout master
```

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

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

```
git checkout pa0
```

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

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

- before starting a new PA, new a branch `pa0` and check out to it
- coding in the branch `pa0`. (this will introduce lot of modifications)
- after finish the PA, merge the branch `pa0` into `master`, and check out back to `master`

Compiling and Running NEMU

Now enter `nemu/` directory. Before the first time to compile NEMU, a configuration file should be generated by

```
make menuconfig
```

一定要做这步！

但有可能会报出一个错误信息，对的概念要讲清楚，解决方法也一个有10吧：你需要把缺少的工具装上。

至于怎么装，自然是百度了。

如果报错了

你有可能会报出一个错误信息，对的概念要讲清楚，解决方法也一个有10吧：你需要把缺少的工具装上。

至于怎么装，自然是百度了。

在Ubuntu下

根据提示的报错信息，比如说

```
scripts/kconfig/conf.o: In function `read_init_dependency':
scripts/kconfig/conf.o:(.text+0x1a0): warning: Unknown `fd' in `/proc/self/fd/NN'
/usr/bin/ld: cannot open output file conf: Permission denied
collect2: error: ld returned 1 exit status
```

通常是 `libncurses-devel` 依赖的问题。解决方法可以是自己搜索，明明白白地安装依赖。

但为了Ubuntu 20.04，可不妨直接 `sudo apt-get install libncurses` 就搞定啦！只需要这样下面的依赖就安装 libncurses-dev 即可搞定。

```
diff --git a/nemu/scripts/build/fixdep.mk b/nemu/scripts/build/fixdep.mk
index c9e1f8b..a7b2a42 100644
--- a/nemu/scripts/build/fixdep.mk
+++ b/nemu/scripts/build/fixdep.mk
@@ -3,3 +3,3 @@
-FIXDEP_DEPS_CHECK
-CHECKC = scripts/kconfig.mk
-CHECKFLAGS := $(shell find-config -o -name "$(CHECKC)" -PPSX
+CHECK := $(shell find-config -o -name
+CHECKFLAGS := $(shell find-config -i -name "$(CHECK)" -PPSX
-mkdir -p $(dir $(fixdep.d))
+mkdir -p $(dir $(fixdep.d))
endif
```

看你要怎样改变工程文件，最自行你发挥大师

Which Implement?

You should know how a program is generated in the 理论课计算机系统 course. But do you have any idea about what happened when a bunch of information is output to the screen during `make -j -s execution`?

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:

```
lib/mkconfig/checker.sh warning: Exercise: Please remove mk in the source code and conf
/local/home/stevenshiqx/ics2021/nemu/tools/kvm-qemu/Makefile.git: warning: Nonsense $'\t'error
```

This message tells you that the program has triggered an assertion fail at line 21 of the file `lib/mkconfig/checker.sh`. If you did not know what is assertion, blame the 理论课计算机系统. But 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 4873f3b7b4c3b6f3f7f7f3d07b1d8d3d1e8e7b
Author: tracer-ics2021-fitness<tracer@ics.nju.edu>
Date:   Sun Jul 18 14:06:31 2021 +0800

    * compile
    INCLUDEMAK
    - Static binrtmem: mytest
      - Size name: simpleform-hartigen
      - Checsize: testrep
      - Machine ID: 4f9f6e67b09a8ae8b7e7427e5d66ba
      - Boot ID: a0324d9d3d1d47b4a9d3d4b7d2d6d79
    - Operating System: Ubuntu 21.04
      - Kernel: Linux 5.11.0-25-generic
    - Architecture: x86-64
    24.04.04 at 20 Mem, 23.6Gb, 16 cores, load average: 3.45, 3.62, 3.87
```

this means the change is traced successfully.

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

```
fatal: fetch-ts create '/home/user/ics2021/.git/index.lock': File exists,

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

Try to clean the compilation result and type again:

```
make clean
make
```

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

如果遇到

我们提供 `gzb` 可帮你实现记录每次代码改动，不会有麻烦的工单名称和代码改动。但是有个前提：你需要先安装 `gzb` 命令。

`gzb` 是一个用户自定义的命令，通常需要安装在用户的目录下。`gzb` 的安装方法可参考 `gzb` 的安装说明文档。

- 安装 `gzb` 命令
- 通常不需要重复执行 `gzb`，每次重复执行会提示已经存在的文件或目录。
- 重复执行的情况通常是出现报错。
- 不要把你的代码工作目录放在你的用户目录的root个人文件夹下的目录下。
- 出现上面提示的情况，不要直接放弃这个目录，而应一定先检查你的工作目录下，这 `rm -rf` 命令是最后最后的手段。
- 不要直接 `rm -rf` 你的工作目录，你最终会后悔的。
- 不要删除你打算保存的内容，否则会导致你的工作目录丢失。
- 不要删除gzb的目录，否则会导致你的工作目录无法正常工作。

如果你的工作目录不小心被删掉了，别慌，别担心，别着急！

可能出现的情况有两种：

- 你的工作目录被删掉了，但是你的工作目录下的内容还在，这种情况下，你可以通过 `gzb -c -d -f -t -r -s -v -w -x -y -z -a -b -c -d -e -f -g -h -i -j -k -l -m -n -o -p -q -r -s -t -u

other elements in a computer system (data structure, computer architecture, assembly language, linking, operating system, network...). It covers most parts of this course. You are strongly recommended to read this tutorial.

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

- STFW
- RTFM
- RTFSC

必答题

独立解决问题是作为码农的一项十分重要的生存技能. 往届有同学在群里提出如下问题:

- su认证失败是怎么回事?
- grep提示no such file or directory是什么意思?
- 请问怎么卸载Ubuntu?
- C语言的xxx语法是什么意思?
- ignoring return vaule of 'scanf'是什么意思?
- 出现curl: not found该怎么办?
- 为什么strtok返回NULL?
- 为什么会有Segmentation fault这个错误?
- 什么是busybox?

请仔细阅读提问的智慧和别像弱智一样提问 (这篇文章很短, 1分钟就能看完)这两篇文章, 结合自己在大一时提问和被提问, 以及完成PA0的经历, 写一篇不少于800字的读后感, 谈谈你对"好的提问"以及"通过STFW和RTFM独立解决问题"的看法.

Hint: 我们设置这道题并不是为了故意浪费大家的时间, 也不是为了禁止大家提出任何问题, 而是为了让大家知道"什么是正确的". 当你愿意为这些"正确的做法"去努力, 并且尝试用专业的方式提出问题的时候, 你就已经迈出了成为"成为专业人士"的第一步.

Reminder

This ends PA0. Please submit your project and report.

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

More Exploration

Learning to use basic tools

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

🚩 RTFM

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

🚩 为什么要RTFM？

RTFM是STFW的长辈，在互联网还不是很流行的年代，RTFM是解决问题的一种有效方法。这是因为手册包含了复杂对象的所有信息，关于查找对象的一切问题都可以在手册中找到答案。

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

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

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

📝 Write a "Hello World" program under GNU/Linux

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

📝 Write a Makefile to compile the "Hello World" program

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

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

📝 Learn to use GDB

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

！提！前瞻瞻瞻！

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

Installing tmux

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

```
apt-get install tmux
```

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

```
cd ~
```

New a file called `.tmux.conf`:

```
vim .tmux.conf
```

Append the following content to the file:

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

These three lines of settings make `tmux` "remember" the current working directory of the current pane while creating new window/pane.

Maximize the terminal windows size, then use `tmux` to create multiple normal-size terminals within single screen. For example, you may edit different files in different directories simultaneously. You can edit them in different terminals, compile them or execute other commands in another terminal, without opening and closing source files back and forth. You can scroll the content in a `tmux` terminal up and down. For how to use `tmux`, please STFW.

🚩 又要没完没了地STFW？？

对.

PA是了让大家巩固ICS理论课的知识之外，还承担着一个重要的任务：把大家培养成一个素质合格的CSer. 事实上，一个素质合格的CSer, 需要具备独立搜索解决方案的能力，这是计业全科和研机构对程序员的一个基本要求. 在将来的老板或项目组看起一个任务直接丢给你，如果你一遇到困难就找人帮忙，老板就会认为你没法创造价值.

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

🚩 提问的智慧/阐像弱智一样提问

一个素质合格的CSer需要具备的另一个标准是, 懂得如何提问.

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

事实上, 如果希望能提高得到回答的概率, 提问者应该学会如何更好地提问. 换句话说, 提问者应该去积极思考“我可以主动做些什么来让对方更方便地帮助我排除问题”. 文科小伙伴确实不是学习计算机专业的, 你可以选择原谅他; 但你是CSer, 至少你得在问题中增进其体的程度以及你做过的尝试, 而不是直接丢一句“你的程序挂了”. 就等着别人来救你. 在你将来的职业生涯中也将有可能需要向别人求助, 比如在论坛/等开源社区中求issue, 或者是在stackoverflow等论坛上发帖, 或者给技术工程师发邮件等. 如果你的提问方式非常不专业, 很可能没有人愿意关注你的问题. 因为这不仅让人觉得你随便提的问题没那么重要, 而且大家也不愿意花费大量的时间向你来回地沟通.

一种推荐的提问方式如下:

```
我在xxx的时候遇到了xxx的错误，这个错误可以通过以下步骤复现：(描述具体的现象)
1. 我的系统版本是xxx，相关的工具版本是xxx
2. 我做了xxx（必要的时候贴个图）
3. 然后xxx（必要的时候贴个图）
...
为了排查这个错误，我进行了以下尝试：(说明我曾希望可以解决问题，真的动动脑子提问的)
1. 我做了xxx，出现了xxx的结果（必要的时候贴个图）
2. 我还做了xxx，出现了xxx的结果（必要的时候贴个图）
...
最后问题还没有解决，请问我还需要做哪些事情？
```

另外请大家务必阅读提问的智慧和阐像弱智一样提问这两篇文章, 里面有不少例子供大家参考.

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

🚩 为什么要使用tmux？

这其实是一个“使用工具的工具做事情”的例子.

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

当然, 学习不是零或本的, 往往有学长说是一种学学习成本的分算方式: 打开4个终端, 并将它们分别指动到屏幕的4个角落, 发现用Alt+Tab快捷键不方便选择窗口因为4个窗口的外观都差不多), 就使用鼠标点击的方式来切换. 然后形容安装学习 `tmux` 是“脱裤子放屁 - 多此一举”.

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

🌍 Things behind scrolling

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

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

Why GNU/Linux and How to

🚩 为什么要使用Linux？

我们先来看两个例子.

```
1. 如何比较两个文件是否完全相同？
```

这个例子看上去非常简单, 在Linux下使用 `diff` 命令就可以实现. 如果文件很大, 那不妨用 `md5sum` 来计算并比较它们的MD5. 对一个Linux用户来说, 键入这些命令只需要花费大约3秒的时间. 但在Windows下, 这件事要麻烦来很不那么容易了: 也许你下载了一个MD5计算工具, 但你需要点击多少次鼠标才能完成这短短1.5秒钟的简单一次好像也谁不了多少时间. 然而鼠标是, 你的开发效率就是这样一点点降低的.

```
2. 如何列出一个C语言项目中所有被包含过的头文件？
```

这个例子比刚才的稍微复杂一但, 但在Windows下你几乎无法高效地做到它. 在Linux中, 我们只需要通过一行命令就可以做到了:

```
find . -name "*.[ch]" | xargs grep "#include" | sort | uniq
```

通过查阅 `man`, 你应该不难理解上述命令是如何实现所需功能的. 这个例子再次体现了Unix哲学:

- 每个工具只做一件事情, 但做到极致
- 工具采用文本方式进行输入输出, 从而易于使用
- 通过工具之间的组合来解决复杂问题

Unix哲学的最后一点最能体现Linux和Windows的区别: 编程经验. 如果把工具比作代码中的函数, 工具之间的组合就是一种编程. 而Windows的工具之间几乎无法组合. 因为面向普通用户的Windows需要强调易用性.

所以, 你应该使用Linux的原因非常简单: 作为一个码农, Windows一直在阻碍你思想, 能力和效率的提升.

🚩 如何用好Linux？

1. 抛弃Windows, 解放思想. 摆脱Windows对你的束缚. 与其默认“没办法, 也只能这样了”, 你应该去尝试“看看能不能把这件事做好”.
   - Linux下也有相应的常用软件, 如Chrome, WPS, 中文输入法, mplayer...
   - 没有Windows你也可以活下去
   - 实在不行可以做个Windows虚拟机备用
2. 熟悉一些常用的命令行工具, 并强迫自己在日常操作中使用它们
   - 文件管理：`cd` | `pwd` | `mkdir` / `rmdir` | `ls` | `cp` | `rm` | `mv` | `tar`
   - 文件检索：`cat` | `more` | `less` | `head` | `tail` | `file` | `find`
   - 输入输出控制 - 重定向, 管道, `tee` | `xargs`
   - 文本处理：`vim` | `grep` | `awk` | `sed` | `sort` | `wc` | `uniq` | `cut` | `tr`
   - 正则表达式
   - 系统监控：`ps` | `top` | `kill` | `free` | `dmesg` | `lsof`
   - 上述工具最大的特点是水多必胖瘦
     - 可以先从简单的尝试开始, 用得多就记住了, 记不住就 `man`
3. RTFM + STFW
4. 坚持.
   - 心态上, 相信总有对的工具能帮助我做得更好
   - 行动上, 愿意付出时间去找到它, 学它, 用它

🚩 信息技术: The Missing Semester of Your CS Education

The Missing Semester of Your CS Education是jyy强烈推荐的Linux工具系列教程, 教你如何使用各种工具来帮助你在计算机上高效地完成各种任务, 让你终身收益.

这套教程有中文版, 去看看吧.

⚠ 克服恐惧, 要积极的信心

事实上, 学习使用Linux是一个抵成本, 高成功率的锻炼机会. 只要你愿意STFW和RTFM, 就能解决绝大部分的问题. 相比较而言, 你之后出现PA中/后续课程中工作中遇到的问题只会更加困难. 因此, 独立解决这些简单的小问题, 你就会开始积累最初的信心, 从而也慢慢相信自己有能力解决更难的问题.

PA1 - 开天辟地的篇章: 最简单的计算机

世界诞生的故事 - 第一章

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

! 代码管理

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

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

! 提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 30小时

阶段性安排:

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

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

在开始编译的PA之旅吧

❓ PA是一种安装的游戏

我们以以下图片来阐述你会实现哪些计组：

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

即PA的目标是阐述清楚计组原理，并实现一个能运行程序的计算机系统，来深入理解程序如何在计算机上运行，和即在“阐述时实现实现者”的程序设计作业之不同，计算机系统出现错误要复杂得多，这原因是，通过程序设计作业的实践方式是不会出现错误的，只会实现错误的程序，而计算机系统的实现者，则会出现错误的程序，才能一步步实现PA。

同时，不要高估用程序设计作业的风格来做即PA的定写思考方案，之所以说之的理由在于，即是：程序设计作业是可能会出现错误的，但是，在编程每一个程序，也需要每一个程序设计作业，也是为了以大家思考方案的思维的了解来提对本质PA的Bug。

这对应来说也是一种即应用来有的游戏方式，即以你也需要意业全新的思考来理受全新的挑战。

❓ 即PA的正确性，从今天开始，不要被骗了！这是一个挑战，当你将来觉得通过的测试程序，但来这里看看

我们向你一一给你解答：

- 通向问题，不要被表面的现象迷惑，切勿不要被骗了！
- 明确你的目标，并且在开始编程前，考虑将来的
  - 你不要在开始编程前，不考虑将来的情况，这会导致一种错误的方式来编程
- 设计最佳的代码结构，不要为了一时的方便，而不顾长远的
- 不要在没有考虑清楚的情况下，就开始编程
- 要选择最佳的一个方案，而不是随便选择一个方案
- 在这个阶段中，不要为了一时的方便，而不顾长远的

如果你遵守以上的原则，你将会发现，你的程序编写的方式，即以你也需要意业全新的思考来理受全新的挑战。

正确的原则是：

- 多思考为妙
  - 从问题的本质来思考理解系统的是个不错的方法
- 独立解决问题
  - 即使是一个很简单的问题，“即即”使用别人的方法也是很多多多
  - 换句话说，如果你选择别人的，你失去的机会也应该是很多多多
- 要成功可能要解决一点困难
  - 所有的bug的时候，这是你可能是你多多多的方法之后
  - 换句话说，你也应该bug的时候要别人不等，不是也要
- 要成功可能要解决一点困难
- 要相信不要心急，不要心急浮躁
- 永远不要相信一个你的对手，或是一个同伴的话，你清楚的话

如果你遵守以上的原则，你将会发现，你的程序编写的方式，即以你也需要意业全新的思考来理受全新的挑战。

我们向你一一给你解答：

- 通向问题，不要被表面的现象迷惑，切勿不要被骗了！
- 明确你的目标，并且在开始编程前，考虑将来的
  - 你不要在开始编程前，不考虑将来的情况，这会导致一种错误的方式来编程
- 设计最佳的代码结构，不要为了一时的方便，而不顾长远的
- 不要在没有考虑清楚的情况下，就开始编程
- 要选择最佳的一个方案，而不是随便选择一个方案
- 在这个阶段中，不要为了一时的方便，而不顾长远的

NEMU是什么？

PA的目的是要实现NEMU，一款经过简化的全系统模拟器。但什么是模拟器？

你小时候玩过任天堂的红白机吗，超级马里奥，坦克等等。它们的画面是怎么设计记忆的呢？像素阵列之间会有几个，当操作的时候，你已经将操作传上表的记忆在内存条了，当你正在为此感到惊喜的时候，模拟器的解决方案就了：你心中记忆了这个画面，这个画面，它的内存条记忆了这个画面，它的内存条记忆了这个画面，它的内存条记忆了一个真正的记忆，即这个画面，它的内存条的记忆的内存条，我们称呼这个画面为模拟器显示器的CGA。你在PA中已经知道了吧，即可以在写出这个红白机的基本程序的时候，再次回顾你之时的体会时候，这实在是太神奇了！

❓ 不要着一下吗？

我们在这里需要在电脑网内部记问这些下一个要求的内存MEM下测试，说说并理解 `fixes-and-NEMU` 中的内容的进行的内容，即在完善这些要记问这些的问题。

你也可以将自己的TFTP协议程序的其它的MCS文件放进来，这将即可以进行其它测试了。

❓ 性能表现，但模拟严谨

在运行游戏的过程中，你需要感受的是一种意义上的：即你的，实现的只有程序性质模拟出来的程序，能将你的在软件的程序中在模拟程序，但你在在记忆进入至今的程序，如果实现不好，你在即PA的部分在你的时候，但不你在即将成熟，但如果真是不能成熟，即在即PA的部分的实现的时候，这实在是太神奇了！

为了程序性质，你需要理解一下即的子项目 `am-kernels` ，重要的在于二部分的内容：

```
cd ics2021
bash init.sh am-kernels

然后进行其中的按键测试程序：

cd am-kernels/tests/am-tests
make ARCH=native mainargs=k run
```

运行后会弹出一个窗口，在即即口中不同程序，即在即程序的程序性质模拟出来的即的程序性质，但即即键盘，键盘的，以及即键式的，如果你发现键出的即键盘与按下的键不等，请你打报复解决方案时实用键盘“SDL keyboard”的，即即键盘的即即键盘与不实现入口程序

导航
最简单的计算机
重新认识程序: 程序是个状态机

开天辟地的篇章

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

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

最简单的计算机

为了执行程序，因此，我们需要一个足够大容量的部件，来放下各种各样的程序，这个部件就是存储器。于是，先聪创造了存储器，并把程序放在存储器中，等待着CPU去执行。

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

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

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

> **计算机可以没有寄存器吗？(建议二周后思考)**
>
> 如果没有寄存器，计算机还可以工作吗？如果可以，这会对硬件提供的编程模型有什么影响呢？
>
> 就算你是二周目来思考这个问题，你也有可能是第一次听到“编程模型”这个概念，不过如果一周目的时候你已经仔细地阅读过ISA手册，你会记得确实有这么个概念，所以，如果想知道什么是编程模型，RTFM吧。

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

有了指令以后，先聪提出了一个划时代的设想：能否让程序来自动控制计算机的执行？为了实现这个设想，先聪和CPU终于一个简单的约定：当执行完一条指令之后，就继续执行下一条指令，但CPU怎么知道现在执行到哪一条指令呢？为此，先聪为CPU创造了一个特殊的计数器，叫“程序计数器”(Program Counter, PC). 在x86中，它有一个特殊的名字，叫 `EIP` (Extended Instruction Pointer).

从此以后，计算机就只需要做一件事情：

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

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

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

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

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

> **尝试理解计算机如何计算**
>
> 在看到上述例子之前，你可能会觉得指令是一个既神秘又难以理解的概念，不过当你看到对应的C代码时，你就会发现指令做的事情竟然这么简单！而且看上去还有点蠢，你随手写一个for循环都要比这段C代码看上去更高级。
>
> 不过你也不妨站在计算机的角度来理解一下，计算机究竟是怎么通过这种既简单又笨拙的方式来计算 `1+2+...+100` 的，这种理解会使你建立“程序如何在计算机上运行”的最初原的认识。

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

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

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

> **计算机是个状态机**
>
> 既然计算机是一个数组逻辑电路, 那么我们可以把计算机划分成两部分, 一部分由所有时序逻辑部件(存储器, 计数器, 寄存器)构成, 另一部分则是剩余的组合逻辑部件(如加法器等). 这样以后, 我们就可以从状态机模型的视角来理解计算机的工作过程了: 在每个时钟周期到来的时候, 计算机根据当前时序逻辑部件的状态, 在组合逻辑部件的作用下, 计算出并转移到下一时钟周期的新状态.
>
> 计算机的这个视角有什么用呢？好像除了让你明白计算机硬件不再那么神秘之外, 也没什么特别的用处. 毕竟ICS课不要求大家用硬件描述语言来实现计算机硬件, 大家只要相信这件事能做成就可以了.
>
> 不过对于程序来说, 这个视角的作用会超乎你的想象.

重新认识程序: 程序是个状态机

如果把计算机看成一个状态机, 那么运行在计算机上面的程序又是什么呢？

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

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

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

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

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

这个程序可能是:

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

> **从状态机视角理解程序运行**
>
> 以上一小节中 `1+2+...+100` 的指令序列为例, 尝试画出这个程序的状态机.
>
> 这个程序比较简单, 需要更新的状态只包括 `PC` 和 `r1` , `r2` 这两个寄存器, 因此我们用一个三元组 `(PC, r1, r2)` 就可以表示程序的所有状态. 而天真画出内存的具体状态, 初始状态是 `(0, x, x)` , 此处的 `x` 表示未初始化. 程序 `PC=0` 处的指令是 `mov r1, 0` , 执行完之后 `PC` 会指向下一条指令, 因此下一个状态是 `(1, 0, x)` , 如此类推, 我们可以画出执行前3条指令的状态转移过程:
>
> ```
> (0, x, x) -> (1, 0, x) -> (2, 0, 0) -> (3, 0, 1)
> ```
>
> 请你尝试继续画出这个状态机, 其中程序中的循环只需要画出前两次循环和最后两次循环即可.

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

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

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

> **“程序在计算机上运行”的微观视角: 程序是个状态机**
>
> “程序是个状态机”这一说法是对ICS和PA来说都是非常重要的, 因为“理解程序如何在计算机上运行”就是ICS和PA的根本目标. 至于这个问题的宏观视角, 我们将会在PA的中期来介绍.
>
> 这一小节的文字对你来说应该不难理解, 但如果你将没有养成从状态机视角理解程序行为的意识, 你可能会感到PA非常困难, 因为在PA中你需要不断地和代码打交道. 如果你不能从微观视角理解某些关键代码的行为, 你也无法从宏观视角完全弄清整程序究竟是如何运行的.

- 专题
- 现代汇编语言
- 第第第第第第第第
- 汇编语言Linux/Unix
- 第第第第第第第第
- 第第第第一个第一个程序
- 运行第一个第一个程序
- 用第三方工具编

RTFSC

第第之二 TMON的汇编语言编程，以及下面的几个程序的它们的用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它们的用途，用途，以及下面的几个程序的它

程序代码目录

程序代码目录是一个，其中包含了本书全部的程序代码目录，程序代码目录，程序代码目录，

当前PC指向的一条指令, 然后更新PC.

❓ 究竟要执行多久?

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

❓ 潜在的威胁 (建议二周目思考)

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

不同的ISA有着不同的指令格式和含义, 因此执行指令的代码自然是ISA相关的. 这部分代码位于 `nemu/src/isa/$ISA/instr/` 目录下. 关于指令执行的详细说明需要涉及很多细节, 目前你无需关心, 我们将会在PA2中进行说明.

由于刚才我们运行NEMU的时候并未给出客户程序的镜像文件, 此时NEMU将会运行上文提到的内置客户程序. NEMU将不断执行指令, 直到遇到以下情况之一, 才会退出指令执行的循环:

- 达到要求的循环次数.
- 客户程序执行了 `nemu_trap` 指令. 这是一条特殊的指令, 在ISA手册中并不存在, 它是为了在NEMU中让客户程序指示执行的结束而加入的. 为了表示客户程序是否成功结束, `nemu_trap` 指令还会接收一个表示结束状态的参数. 当客户程序执行了这条指令之后, NEMU将会根据这个结束状态参数来设置NEMU的结束状态, 并根据不同的状态输出不同的结束信息, 主要包括
  - `HIT GOOD TRAP` - 客户程序正确地结束执行
  - `HIT BAD TRAP` - 客户程序错误地结束执行
  - `ABORT` - 客户程序意外终止, 并未结束执行

当你看到NEMU输出类似以下的内容时(不同ISA的pc输出值会有所不同):

```
nemu: HIT GOOD TRAP at pc = 0x8000000c
```

说明客户程序已经成功地结束运行. NEMU会在 `cpu_exec()` 函数的最后打印执行的指令数目和花费的时间, 并计算出指令执行的频率. 但由于内置客户程序太小, 执行很快就结束了, 目前无法计算出有意义的频率, 将来运行一些复杂的程序时, 此处输出的频率可以用于粗略地衡量NEMU的性能.

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

❓ 谁来指示程序的结束?

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

❓ 有始有终 (建议二周目思考)

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

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

最后我们聊聊代码中一些值得注意的地方.

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

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

理解框架代码

你需要结合上述文字理解NEMU的框架代码.

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.

📢 RTFSC != 盯着代码看

你很可能是第一次接触到这么多源文件的项目, 看代码的时候可能会感到很迷茫: 不知道函数的定义在哪个文件, 不理解函数的功能是什么, 不清楚某段代码的行为具体如何... 同时你也很可能沿用以前看代码最原始的方式: 用眼睛看. 你坚持看了一段时间, 发现还是没什么收获, 于是你开始感到沮丧...

对于只有一两个源文件, 代码只有几百行的项目, 直接RTFSC还是有效果的. 但如果源文件更多, 代码量更大, 你很快就会发现这种做法效率很低, 这是因为人脑的短期记忆能力是很有限的, 即使是静态的代码也无法完全记得住, 更何况程序的动态行为是一个巨大的状态机, 你的大脑在一段时间内只能模拟出这个状态机很小的一部分.

有没有工具能够帮你模拟这个巨大的状态机呢? 这时我们在PA0里面提到的一个工具就派上用场了, 它就是GDB. 在GDB中, 我们可以通过单步执行的方式让程序一次执行一条指令, 相当于让状态机一次只前进一步, 这样我们就可以观察程序任意时刻的状态了! 而且状态机前进的轨迹就是程序执行的真实顺序, 于是你就可以一边运行程序一边理解程序的行为了. 这对于一些指针相关的代码有着不错的效果, 尤其是函数指针, 因为你从静态代码上很可能看不出来程序运行的时候这个指针会指向哪个函数.

GDB还自带一个叫TUI的简单界面. 在一个高度较高的窗口中运行GDB后, 输入 `layout split` 就可以切换到TUI, 这样你就可以同时从源代码和指令的角度来观察程序的行为了. 如果你想了解TUI的更多内容, STFW.

为了帮助你更高效地RTFSC, 你最好通过RTFM和STFW多认识GDB的一些命令和操作, 比如:

- 单步执行进入你感兴趣的函数
- 单步执行跳过你不感兴趣的函数(例如库函数)
- 运行到函数末尾
- 打印变量或寄存器的值
- 扫描内存
- 查看调用栈
- 设置断点
- 设置监视点

如果你之前没有使用过GDB, 然后在PA0中又跳过了GDB相关的内容, 现在你就要吃偷懒的亏了.

为NEMU编译时添加GDB调试信息

menuconfig已经为大家准备好相应选项了, 你只需要打开它:

```
Build Options
  [*] Enable debug information
```

然后清除编译结果并重新编译即可. 尝试阅读相关代码, 理解开启上述menuconfig选项后会导致编译NEMU时的选项产生什么变化.

优美地退出

为了测试大家是否已经理解框架代码, 我们给大家设置一个练习: 如果在运行NEMU之后直接键入 `q` 退出, 你会发现终端输出了一些错误信息. 请分析这个错误信息是什么原因造成的, 然后尝试在NEMU中修复它.

就是这么简单

事实上, TRM的实现已经都蕴含在上述的介绍中了.

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

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

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

导航
解析命令
单步执行
打印寄存器
扫描内存

基础设施: 简易调试器

基础设施 - 提高项目开发的效率

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

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

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

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

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

真实故事

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

这一科研工作需要运行各种不同的测试来验证效果, 在两台4核8线程的PC机上运行完所有的测试需要花费约24小时. 每次对设计进行改动之后, 都需要重新运行所有测试, 可以说是, 改一行代码, 要过24小时后才能得到结果. 事实上, 在投稿截止3个月之前, 我们已经得知有一台112核的服务器可以使用. 如果在这台服务器上部署测试环境, 预计可以使测试总时间减少到原来的1/5 (多核服务器的主频比PC机低1倍, 架构也落后两代, 1/5是一个综合考虑之后的数字).

但是带领这一项目的同学并没有意识到基础设施的重要性: 他一直在PC机上进行测试. 事实上, 测试总时间减少到原来的1/5, 其实意味着改进设计的机会是原来的5倍, 结果到投稿截止之前, 设计还在修改, 测试还在反复运行, 最终只能无奈地采用一个质量有待提高的设计版本提交论文.

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

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

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

备注:

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

总有一天会找上门来的bug

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

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

解析命令

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

```
man readline
```

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

```
man strtok
```

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

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

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

如何测试字符串处理函数?

你可能会抑制不住编码的冲动: 与其RTFM, 还不如自己写. 如果真是这样, 你可以考虑一下, 你会如何测试自己编写的字符串处理函数?

如果你愿意RTFM, 也不妨思考一下这个问题, 因为你会在PA2中遇到类似的问题.

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

单步执行

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

打印寄存器

打印寄存器就更简单了. 不过既然寄存器的结构是ISA相关的, 我们希望能为简易调试器屏蔽ISA的差异. 框架代码已经为大家准备了如下的API:

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

执行 `info r` 之后, 就调用 `isa_reg_display()`, 在里面直接通过 `printf()` 输出所有寄存器的值即可. 如果你从来没有使用过 `printf()`, 请RTFM或者STFW. 如果你不知道要输出什么, 你可以参考GDB中的输出.

扫描内存

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

```
x 10 0x80000000
```

这样的简化可以让你暂时不必纠缠于表达式求值的细节, 解析出待扫描内存的起始地址之后, 就可以使用循环将指定长度的内存数据通过十六进制打印出来. 如果你不知道要怎么输出, 同样的, 你可以参考GDB中的输出. 问题是, 我们要如何访问客户计算机的内存数据呢? (答案早就说了哦)

实现了扫描内存的功能之后, 你可以打印 `0x80000000` 或者 `0x100000` 附近的内存, 你应该会看到程序的代码, 和内置客户程序的内容进行对比, 检查你的实现是否正确.

实现单步执行, 打印寄存器, 扫描内存

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

NEMU默认会把单步执行的指令打印出来(这里面埋了一些坑, 你需要RTFSC看看指令是在哪里被打印的), 这样你就可以验证单步执行的效果了.

不知道如何下手? 嗯, 看来你需要再阅读一遍RTFSC小节的内容了. 如果你已经忘记了某些注意事项, 重新去阅读一遍也是应该的.

我怕代码写错了啊, 怎么办?

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

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

总之, 只有掌握正确的方法和工具, 才能真正驱散心中对bug的恐惧.

温馨提示

PA1阶段1到此结束.

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

表达式求值

数学表达式求值

给定一个表达式字符串

```
"1+2*(4-3)+5"
```

你如何求出这个表达式的值呢？一种方法是将表达式转换成一个抽象语法树（Abstract Syntax Tree）的数据结构，然后遍历该树来计算出表达式的值。

1. 数字直接作为叶子节点
2. 根据表达式的运算符创建子节点

词法分析

词法分析（Lexer）是表达式求值的第一步，它的工作是将一个字符串分解成一系列的令牌（Token）。令牌是一个具有特殊含义的字符序列，它代表了表达式中的一个基本单位。例如，在上面的表达式中，令牌可能包括：

- 数字：`1`、`2`、`4`、`3`、`5`
- 运算符：`+`、`*`、`-`
- 括号：`(`、`)`

这个过程可以看作是一个状态机，根据当前字符和当前状态来决定下一个令牌的类型和内容。例如，`1` 是一个数字，`+` 是一个运算符，`(` 是一个括号。

```
"(abc123+456)"
```

在这个表达式中，我们可以先识别出 `(`，然后识别出 `abc123`，接着是 `+`，最后是 `456`，最后是 `)`。这个过程中，我们需要根据当前字符的类型来决定如何处理它。例如，当遇到 `(` 时，我们知道接下来可能会出现一个新的表达式，因此会创建一个新的子节点；当遇到 `)` 时，我们知道当前表达式已经结束，因此会将当前表达式的结果返回给父节点。

```
"-123+456"
```

在这个表达式中，`-` 可能是一个减号，也可能是一个负号。为了区分这两种情况，我们需要根据上下文来判断。例如，如果 `-` 前面是一个数字，那么它很可能是一个减号；否则，它很可能是一个负号。

```
"1.23+4.56"
```

在这个表达式中，`.` 是一个小数点，它表示接下来的数字是一个小数。为了处理这种情况，我们需要在识别数字的过程中，检查是否遇到了 `.`，并将其作为数字的一部分来处理。

```
"a+b+c"
```

在这个表达式中，`a`、`b`、`c` 是变量名，它们可以是任意长度的字母或数字组成的序列。为了处理这种情况，我们需要在识别数字的过程中，检查是否遇到了字母，并将其作为变量名的一部分来处理。

```
"1+2*(4-3)+5"
```

```
(expression)
```

```
"1.23+4.56"
```

```
"a+b+c"
```

```
"1+2*(4-3)+5"
```

```
"1+2*(4-3)+5"
```

```
"1+2*(4-3)+5
```

目 导航
监视点
扩展表达式求值的功能
实现监视点
调试工具与查看
断点

监视点

监视点的功能是监视一个表达式的值时对发生变化，如果你以来没有使用过监视点，请在GDB中体验一下它的作用。

扩展表达式求值的功能

你之前已经实现了算术表达式的求值，但这些表达式都是非常简单的。它们的语不会发生变化，这样的表达式在监视点中没有任何意义，为了能真实反应的功能，我们需要扩展表达式求值的功能。

我们需给RDP表达式需要扩展解析功能：

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

它们的功能和C语言中运算符的功能是一致的，包括优先级和结合性，如有疑问，请查阅相关资料。

关于获取寄存器的值，这显然是一个64位长的功能，相关代码已经准备了如下的API：

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

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

还需要注意的是指针解引用(dereference)的问题，在进行词法分析的时候，我们其实没有办法把乘法和指针解引用区别开来，因为它们都是 `*` ，在进行语法分析之后，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，将会过程中会引入语法错误。

为了区别这两种情况，我们可以在 `token` 的定义中加入一个 `subtype` 的成员，来表示一个token的类型，我们就可以决定这个 `*` 是乘法还是指针解引用了，可惜的是这样的话，我们在这里定义的 `expr()` 函数的模型：

```
bool expr(char *, int *, bool *);
```

其中的 `certain_type` 就是你自己来思考的！其实上述模型也可以处理负数问题，如果你之前实现了负数，，的你应该对你来说应该什么困难了。

另外RDP的中的测试模块，我们都了解是，随着测试器中的功能的不断增加，测试的复杂程度会随着增加，这时候，我们需要新的测试模块：

- 所有结果都是 `uint32_t` 类型。
- 指针也没有类型，进行指针解引用的时候，我们总是从客户计算机的内存中读出一个 `uint32_t` 类型的整数。

实现监视点

```
$ make run-nemu-test
$ ./build/nemu -b -l ./nemu.log -e ./build/test-expr
[src/monitor/monitor.c:10,init_monitor] Welcome to NEMU-RISCV32!
For help, type "help"
[src/utils/log.c:94,init_log] Log is written to ./nemu.log
[src/utils/sdb.c:63,init_sdb] Initializing simple debugger...
[src/utils/sdb.c:73,init_sdb] Testing expression evaluator: PASS!
[src/utils/sdb.c:75,init_sdb] Testing watchpoint pool: PASS!
[src/utils/sdb.c:77,init_sdb] Testing memory scanner: PASS!
[src/utils/sdb.c:79,init_sdb] Testing watchpoint: PASS!
[src/utils/sdb.c:81,init_sdb] Testing watchpoint with expr: PASS!
[src/utils/sdb.c:83,init_sdb] Testing watchpoint with expr in parallel: PASS!
[src/utils/sdb.c:85,init_sdb] Testing watchpoint with expr in parallel 2: PASS!
[src/utils/sdb.c:87,init_sdb] Testing watchpoint with expr in parallel 3: PASS!
[src/utils/sdb.c:89,init_sdb] Testing watchpoint with expr in parallel 4: PASS!
[src/utils/sdb.c:91,init_sdb] Testing watchpoint with expr in parallel 5: PASS!
```

如何阅读手册

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

学会使用目录

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

尝试通过目录定位关注的问题

假设你现在需要了解一个叫 `selector` 的概念, 请通过i386手册的目录确定你需要阅读手册中的哪些地方. 即使你选择的ISA并不是x86, 也可以尝试去查阅这个概念.

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

逐步细化搜索范围

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

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

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

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

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

必答题

你需要在实验报告中回答下列问题:

- 程序是个状态机 画出计算 `1+2+...+100` 的程序的状态机, 具体请参考这里.
- 理解基础设施 我们通过一些简单的计算来体会简易调试器的作用. 首先作以下假设:
  - 假设你需要编译500次NEMU才能完成PA.
  - 假设这500次编译当中, 有90%的次数是用于调试.
  - 假设你没有实现简易调试器, 只能通过GDB对运行在NEMU上的客户程序进行调试. 在每一次调试中, 由于GDB不能直接观测客户程序, 你需要花费30秒的时间来从GDB中获取并分析一个信息.
  - 假设你需要获取并分析20个信息才能排除一个bug.

  那么这个学期下来, 你将会在调试上花费多少时间?

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

  事实上, 这些数字也许还是有点乐观, 例如就算使用GDB来直接调试客户程序, 这些数字假设你能通过10分钟的时间排除一个bug. 如果实际上你需要在调试过程中获取并分析更多的信息, 简易调试器这一基础设施能带来的好处就更大.
- RTFM 理解了科学查阅手册的方法之后, 请你尝试在你选择的ISA手册中查阅以下问题所在的位置, 把需要阅读的范围写到你的实验报告里面:
  - x86
    - EFLAGS寄存器中的CF位是什么意思?
    - ModR/M字节是什么?
    - mov指令的具体格式是怎么样的?
  - mips32
    - mips32有哪几种指令格式?
    - CP0寄存器是什么?
    - 若除法指令的除数为0, 结果会怎样?
  - riscv32
    - riscv32有哪几种指令格式?
    - LUI指令的行为是什么?
    - mstatus寄存器的结构是怎么样的?
- shell命令 完成PA1的内容之后, `nemu/` 目录下的所有.c和.h和文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在PA1中编写了多少行代码? (Hint: 目前 `pa0` 分支中记录的正好是做PA1之前的状态, 思考一下应该如何回到"过去"?) 你可以把这条命令写入 `Makefile` 中, 随着实验进度的推进, 你可以很方便地统计工程的代码行数, 例如敲入 `make count` 就会自动运行统计代码行数的命令. 再来个难一点的, 除去空行之外, `nemu/` 目录下的所有 `.c` 和 `.h` 文件总共有多少行代码?
- RTFM 打开 `nemu/scripters/build.mk` 文件, 你会在 `CFLAGS` 变量中看到gcc的一些编译选项. 请解释gcc中的 `-Wall` 和 `-Werror` 有什么作用? 为什么要使用 `-Wall` 和 `-Werror` ?

温馨提示

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

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

PA2 - 简单复杂的机器: 冯诺依曼计算机系统

世界诞生的故事 - 第二章

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

! 代码管理

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

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

! 提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 40小时

阶段性安排:

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

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

导航

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

YEMU: 一个简单的CPU模拟器

热身结束, 进入真正的PA

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

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

所以, 不要再偷懒了.

不停计算的机器

在PA1中, 我们已经见识到最简单的计算机TRM的工作方式:

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

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

取指(instruction fetch, IF)

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

译码(instruction decode, ID)

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

```
10111001 00110100 00010010 00000000 00000000
```

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

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

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

当然, 译码逻辑实际上也并非只有一张查找表那么简单, 还需要根据不同的指令通过多路选择器选择不同的操作数. 回想一下, 计算机现在已经有存储器和寄存器了, 它们都可以存放操作数, 指令中也可以存放立即数. 也可能还有二次译码的处理... 不过无论再怎么复杂, 我们只需要知道, 这个过程终究也只是一些数字电路的事情, 毕竟所有需要的信息都在指令里面了, 没什么神秘的操作.

执行(execute, EX)

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

更新PC

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

于是, 计算机不断地重复上述四个步骤, 不断地执行指令, 直到永远.

YEMU: 一个简单的CPU模拟器

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

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

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

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

#define NREG 4
#define NMEM 16

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

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

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

int halt = 0; // 结束标志

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

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

理解YEMU如何执行程序

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

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

思考一下, 以上两者有什么联系?

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

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

一种新的几乎自定义的、RISC-V 指令集扩展（称为 “Xmem”）。中定义，它们的目标是不需要修改现有指令集，而是通过新指令来实现高效的内存操作。不同的内存操作可以使用不同的几乎自定义的、RISC-V 指令集扩展（称为 “Xmem”）。中定义，它们的目标是不需要修改现有指令集，而是通过新指令来实现高效的内存操作。

- 加载指令集：将数据从内存加载到寄存器。示例指令包括 `lbr1` （加载字节）、`lhr1` （加载半字）、`lwr1` （加载字）、`ldr1` （加载双字）。
- 存储指令集：将数据从寄存器存储到内存。示例指令包括 `sbr1` （存储字节）、`shr1` （存储半字）、`swr1` （存储字）、`sdr1` （存储双字）。
- 原子指令集：执行原子内存操作，例如 `fetch_add` （原子加）、`fetch_and` （原子与）、`fetch_or` （原子或）、`fetch_xor` （原子异或）、`fetch_min` （原子最小值）、`fetch_max` （原子最大值）、`fetch_swap` （原子交换）、`fetch_cas` （原子比较并交换）。
- 屏障指令集：执行内存屏障操作，例如 `fence` （全屏障）、`fence.r` （读屏障）、`fence.w` （写屏障）。
- 内存管理指令集：管理内存分配和释放，例如 `malloc` （分配内存）、`free` （释放内存）、`realloc` （重新分配内存）。
- 缓存指令集：管理缓存操作，例如 `cache.flush` （刷新缓存）、`cache.invalidate` （使缓存无效）。
- 内存屏障指令集：执行内存屏障操作，例如 `mfence` （全屏障）、`lfence` （读屏障）、`sfence` （写屏障）。
- 内存保护指令集：管理内存保护，例如 `mprotect` （修改内存保护）、`mlock` （锁定内存）、`munlock` （解锁内存）。
- 内存调试指令集：调试内存操作，例如 `memcheck` （内存检查）、`memprofile` （内存分析）、`memtrace` （内存跟踪）。
- 内存压缩指令集：压缩内存数据，例如 `compress` （压缩）、`decompress` （解压）。
- 内存加密指令集：加密内存数据，例如 `encrypt` （加密）、`decrypt` （解压）。
- 内存虚拟化指令集：虚拟化内存，例如 `vmalloc` （虚拟分配）、`vfree` （虚拟释放）。
- 内存模拟指令集：模拟内存操作，例如 `memset` （内存设置）、`memcpy` （内存复制）、`memmove` （内存移动）。
- 内存分析指令集：分析内存使用情况，例如 `memstat` （内存统计）、`memprofile` （内存分析）。
- 内存安全指令集：确保内存安全性，例如 `memcheck` （内存检查）、`memguard` （内存保护）。
- 内存优化指令集：优化内存操作，例如 `memopt` （内存优化）、`memalign` （内存对齐）。
- 内存调试指令集：调试内存操作，例如 `memdump` （内存转储）、`memtrace` （内存跟踪）。
- 内存管理指令集：管理内存分配和释放，例如 `malloc` （分配内存）、`free` （释放内存）、`realloc` （重新分配内存）。
- 内存压缩指令集：压缩内存数据，例如 `compress` （压缩）、`decompress` （解压）。
- 内存加密指令集：加密内存数据，例如 `encrypt` （加密）、`decrypt` （解压）。
- 内存虚拟化指令集：虚拟化内存，例如 `vmalloc` （虚拟分配）、`vfree` （虚拟释放）。
- 内存模拟指令集：模拟内存操作，例如 `memset` （内存设置）、`memcpy` （内存复制）、`memmove` （内存移动）。
- 内存分析指令集：分析内存使用情况，例如 `memstat` （内存统计）、`memprofile` （内存分析）。
- 内存安全指令集：确保内存安全性，例如 `memcheck` （内存检查）、`memguard` （内存保护）。
- 内存优化指令集：优化内存操作，例如 `memopt` （内存优化）、`memalign` （内存对齐）。
- 内存调试指令集：调试内存操作，例如 `memdump` （内存转储）、`memtrace` （内存跟踪）。

为什么不使用 `mcl_mem_ldr`

我们没有定义用于获取寄存器数据或加载数据的指令。例如，`mcl_mem_ldr` 。

为什么不使用 `mcl_mem_ldr`

如果您选择了 riscv64，上述寄存器操作指令将使用 `mcl_mem_ldr` 。

为什么不使用 `mcl_mem_ldr`

如果您选择了 riscv64，上述寄存器操作指令将使用 `mcl_mem_ldr` 。

- 定义寄存器操作数据类型的基本算术/逻辑运算，例如 `rt_add(dst,lt,rt)` 。
- 定义寄存器操作数据类型的基本算术/逻辑运算，例如 `rt_and(dst,lt,rt)` 。
- 定义寄存器操作数据类型的基本算术/逻辑运算，例如 `rt_or(dst,lt,rt)` 。

这些代码生成寄存器操作数据类型的基本算术/逻辑运算，例如 `rt_add(dst,lt,rt)` 。

为什么不使用 `mcl_mem_ldr`

我们没有定义用于获取寄存器数据或加载数据的指令。例如，`mcl_mem_ldr

程序，运行时环境与AM

运行时环境

现代起源于60年代，当时一个程序（一个进程）可能是一个程序的全部或是多个程序的一部分。一个程序（一个进程）通常是一个或多个可执行文件（.exe, .dll, .so, .jar）和一个或多个数据文件。这些文件通常位于同一个目录下。

一个程序（一个进程）通常是一个或多个可执行文件（.exe, .dll, .so, .jar）和一个或多个数据文件。这些文件通常位于同一个目录下。

在这之后，一个程序（一个进程）可以使用多个数据文件，可能是不同目录下的多个数据文件。这些文件通常位于：

- 当前目录（通常是一个或多个数据文件）。
- 用户目录（通常是一个或多个数据文件）。
- 系统目录（通常是一个或多个数据文件）。

在这之后，一个程序（一个进程）可以使用多个可执行文件。这些可执行文件通常位于：

- 当前目录（通常是一个或多个可执行文件）。
- 用户目录（通常是一个或多个可执行文件）。
- 系统目录（通常是一个或多个可执行文件）。

将运行时环境抽象成模型

现代操作系统的运行时环境通常是可计算机操作之上的，因此运行时环境的抽象是指，在虚拟机器内的，虚拟机器内的操作系统，虚拟机器内的运行时环境。例如：Java的JVM，Microsoft的.Net Framework，Mono，Linux的Mono，Python的Python，Perl的Perl，Ruby的Ruby，Lua的Lua，等等。

现代的虚拟机器（JVM，.Net Framework，Mono，Python，Perl，Ruby，Lua，等等）通常使用字节码（Bytecode）或中间语言（.Net Framework的CIL，Mono的Mono Intermediate Language，Python的.pyc，Perl的.pmc，Ruby的.rbc，Lua的.luac，等等）。

字节码（Bytecode）或中间语言（.Net Framework的CIL，Mono的Mono Intermediate Language，Python的.pyc，Perl的.pmc，Ruby的.rbc，Lua的.luac，等等）通常使用JIT（Just In Time）编译器，在运行时，编译成机器码。

程序，运行时环境与AM

- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。

运行时环境的数据通常是结构化的

现代操作系统的运行时环境通常是可计算机操作之上的，因此运行时环境的数据通常是结构化的。

例如：Microsoft的.Net Framework，Mono，Linux的Mono，Python，Perl，Ruby，Lua，等等，通常使用XML，MSIL，CIL，Bytecode，JSON，.net.cfg，Pipe，等等。

例如：Microsoft的.Net Framework，Mono，Linux的Mono，Python，Perl，Ruby，Lua，等等，通常使用XML，MSIL，CIL，Bytecode，JSON，.net.cfg，Pipe，等等。

例如：Microsoft的.Net Framework，Mono，Linux的Mono，Python，Perl，Ruby，Lua，等等，通常使用XML，MSIL，CIL，Bytecode，JSON，.net.cfg，Pipe，等等。

程序，运行时环境与AM

- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。

程序，运行时环境与AM

- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。
- 程序（一个文件）的数据通常是结构化的，因此，程序的数据通常是结构化的。

AM - 模式Model-view-meta运行时环境

一个程序，又称：一个文件，一个进程，一个数据文件，一个可执行文件，一个目录，一个目录，一个目录

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

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

"程序在计算机上运行"的宏观视角: 计算机是个抽象层

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

每一层抽象都有它存在的理由:

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

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

PA究竟在做什么?

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

- 微观视角: 程序是个状态机
- 宏观视角: 计算机是个抽象层

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

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

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

bug类型判断 - 原因

bug原因判断 - 修复方案

AMD汇编语法

数据结构

Differential Testing

一键回归测试

基础设施的

bug原因判断 - 原因

我们在试图实现新指令下的优化，结果每次测试都失败，这时候我们就需要知道是哪个环节出问题了。

bug的原因判断，就是在优化流程下，工具会判断，是哪个环节出问题了。

我们在实现新指令下的优化，结果每次测试都失败，这时候我们就需要知道是哪个环节出问题了。

bug的原因判断，就是在优化流程下，工具会判断，是哪个环节出问题了。

1 Python的Disassembler原理

我们在Disassembler的测试流程下一直失败，这时候在Disassembler的实现模块，结果报告说的1+10行 2010-06-22 星期四 Python 版本：2.6.5 Disassembler的测试，结果报告说的这样子：

```
gcc opt -output t.out
gcc t.out -m -o -lm2t1 https://psyche.pku.edu.cn/ /tmp/mem
./tmp/mem -d
gcc "mrtest-mem.o" "RTMSU.o" -DTEMP
./tmp/RTMSU -d
# 下面的命令是为了生成一个文件，用来比较两个结果的差异。
# 这个文件应该不同，可以通过以下命令来检查。diff -u
diff -u /tmp/mem.out /tmp/mem-fine-disassembler.out -pack
```

当我们遇到diff -u 的时候，我们就要知道结果是啥，我发现结果出现两个字节不同：

```
1 ret of 1 hash FAILED -- saving objects to file srcinfo/pin-exec.o/.o
```

的结果时，我会发现每次的测试结果不一样了，有时候，结果是说哪条指令的字节不同，有时候，结果是说哪个字节不同。

```
1. mmx/sse/xsave/xsave.o/.o 中结果的不同，我可以把结果说成的这样的结果：
0x 0000e0e0: 0f 7e 0000e0e0 movd %mm0, (%eax)
0x 0000e0e1: 0f 7e 0000e0e1 movd %mm0, (%eax)
这两条指令，一个是movd的结果，一个是movq的结果。
2. mmx/sse/pxor.o/.o 中的结果：
0x 0000e0e0: 0f ef 0000e0e0 pxor %mm0, %mm0
0x 0000e0e1: 0f ef 0000e0e1 pxor %mm0, %mm0
```

我们在实现新指令下的优化，结果每次测试都失败，这时候我们就需要知道是哪个字节不同。

2 Python的Disassembler - vimgdiff

一般来说，我们在这么试图实现的时候，我会打一些大范围的测试，这时候我们vimgdiff大多时候有很重要的作用。一般来说，我们在这么试图实现的时候，我们通常是在第一个测试的结果，然后通过vimgdiff来比较结果的差异，这样的话，我们就可以很快的找出问题。

我们在这么试图实现的时候，我们通常是在第一个测试的结果，然后通过vimgdiff来比较结果的差异，这样的话，我们就可以很快的找出问题。

我们在这么试图实现的时候，我们通常是在第一个测试的结果，然后通过vimgdiff来比结果的差异，这样的话，我们就可以很快的找出问题。

```
0x 0000e0e0: ed 6c 0000e0e0 insd (%dx), %es:(%edi)
0x 0000e0e1: e1 6c 0000e0e1 loope 0x 0000e0a3
0x 0000e0e2: 6c 0000e0e2 insb (%dx), %es:(%edi)
0x 0000e0e3: ab 0000e0e3 stosl %eax, %es:(%edi)
0x 0000e0e4: 6c 0000e0e4 insb (%dx), %es:(%edi)
0x 0000e0e5: 6c 0000e0e5 insb (%dx), %es:(%edi)
0x 0000e0e6: 6c 0000e0e6 insb (%dx), %es:(%edi)
0x 0000e0e7: 6c 0000e0e7 insb (%dx), %es:(%edi)
0x 0000e0e8: 6c 0000e0e8 insb (%dx), %es:(%edi)
0x 0000e0e9: 6c 0000e0e9 insb (%dx), %es:(%edi)
0x 0000e0ea: 6c 0000e0ea insb (%dx), %es:(%edi)
0x 0000e0eb: 6c 0000e0eb insb (%dx), %es:(%edi)
0x 0000e0ec: 6c 0000e0ec insb (%dx), %es:(%edi)
0x 0000e0ed: 6c 0000e0ed insb (%dx), %es:(%edi)
0x 0000e0ee: 6c 0000e0ee insb (%dx), %es:(%edi)
0x 0000e0ef: 6c 0000e0ef insb (%dx), %es:(%edi)
0x 0000e0f0: 6c 0000e0f0 insb (%dx), %es:(%edi)
0x 0000e0f1: 6c 0000e0f1 insb (%dx), %es:(%edi)
0x 0000e0f2: 6c 0000e0f2 insb (%dx), %es:(%edi)
0x 0000e0f3: 6c 0000e0f3 insb (%dx), %es:(%edi)
0x 0000e0f4: 6c 0000e0f4 insb (%dx), %es:(%edi)
0x 0000e0f5: 6c 0000e0f5 insb (%dx), %es:(%edi)
0x 0000e0f6: 6c 0000e0f6 insb (%dx), %es:(%edi)
0x 0000e0f7: 6c 0000e0f7 insb (%dx), %es:(%edi)
0x 0000e0f8: 6c 0000e0f8 insb (%dx), %es:(%edi)
0x 0000e0f9: 6c 0000e0f9 insb (%dx), %es:(%edi)
0x 0000e0fa: 6c 0000e0fa insb (%dx), %es:(%edi)
0x 0000e0fb: 6c 0000e0fb insb (%dx), %es:(%edi)
0x 0000e0fc: 6c 0000e0fc insb (%dx), %es:(%edi)
0x 0000e0fd: 6c 0000e0fd insb (%dx), %es:(%edi)
0x 0000e0fe: 6c 0000e0fe insb (%dx), %es:(%edi)
0x 0000e0ff: 6c 0000e0ff insb (%dx), %es:(%edi)
```

3 Python的Disassembler - 内存访问的原理

我们在Disassembler的测试流程下，一直会发现，结果报告说的这样子：

```
./tiscc64-linux-gcc-rmmdbf -m add-rlscc/lib-rmm.o
```

我们在Disassembler的测试流程下，一直会发现，结果报告说的这样子：

```
Symbol table '.symtab' contains 29 entries:
Num:    Value  Size Type    Bind  Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 FILE    LOCAL  DEFAULT  ABS add.c
     8: 00000000     1 OBJECT  LOCAL  DEFAULT    2 mainargc
     9: 00000000     4 NOTYPE  LOCAL  DEFAULT    2 _IO_stdin_used
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _gcc_qsort
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _gcc_qsort_r
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _fini_start
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _end
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _fini_end
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _start
    16: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _bss_start
    17: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _init_end
    18: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _data_start
    19: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    20: 00000000     0 FUNC    GLOBAL DEFAULT    2 _init_begin
    21: 00000000     0 FUNC    GLOBAL DEFAULT    2 main
    22: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort_setup
    23: 00000000     108 FUNC  GLOBAL DEFAULT    2 main
    24: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort
    25: 00000000     256 OBJECT  GLOBAL DEFAULT    2 _gcc_qsort_offset
    26: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort_setup_end
    27: 00000000     32 FUNC   GLOBAL DEFAULT    2 _start_data
```

其中每一行代表一个变量，每一个字段的含义不清楚，结果报告说的这样子：

```
Symbol table '.symtab' contains 29 entries:
Num:    Value  Size Type    Bind  Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 00000000     0 FILE    LOCAL  DEFAULT  ABS add.c
     8: 00000000     1 OBJECT  LOCAL  DEFAULT    2 mainargc
     9: 00000000     4 NOTYPE  LOCAL  DEFAULT    2 _IO_stdin_used
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _gcc_qsort
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _gcc_qsort_r
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _fini_start
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _end
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _fini_end
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _start
    16: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _bss_start
    17: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _init_end
    18: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _data_start
    19: 00000000     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    20: 00000000     0 FUNC    GLOBAL DEFAULT    2 _init_begin
    21: 00000000     0 FUNC    GLOBAL DEFAULT    2 main
    22: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort_setup
    23: 00000000     108 FUNC  GLOBAL DEFAULT    2 main
    24: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort
    25: 00000000     256 OBJECT  GLOBAL DEFAULT    2 _gcc_qsort_offset
    26: 00000000     0 FUNC    GLOBAL DEFAULT    2 _gcc_qsort_setup_end
    27: 00000000     32 FUNC   GLOBAL DEFAULT    2 _start_data
```

Disassembler的测试流程下，一直会发现，结果报告说的这样子：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flag Lk Inf Al
  [ 1] .text             PROGBITS        00000000 000034 000000 00  AX  0   0  1
```

Disassembler的测试流程下，一直会发现，结果报告说的这样子：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flag Lk Inf Al
  [ 1] .text             PROGBITS        00000000 000034 000000 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 0001d0 000008 08      8   1  4
  [ 3] .data             PROGBITS        00000000 000034 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000034 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 000034 000004 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 000038 000011 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 000049 000000 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 0001d8 000101 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000050 000190 10     10  11  4
  [10] .strtab           STRTAB          00000000 0001e0 000000 00      0   0  1
```

Disassembler的测试流程下，一直会发现，结果报告说的这样子：

```
00000000: <add>:
00000000:  55 89 e5 83 ec 04 83 e4 f0 [illegal]
00000010:  56 bf 01 00 00 00 [illegal]
00000016:  [illegal]
0000001a:  [illegal]
0000001c:  [illegal]
0000001e:  [illegal]
00000020:  [illegal]
00000022:  [illegal]
00000024:  [illegal]
00000026:  [illegal]
00000028:  [illegal]
0000002a:  [illegal]
0000002c:  [illegal]
0000002e:  [illegal]
00000030:  [illegal]
00000032:  [illegal]
00000034:  [illegal]
00000036:  [illegal]
00000038:  [illegal]
0000003a:  [illegal]
0000003c:  [illegal]
0000003e:  [illegal]
00000040:  [illegal]
00000042:  [illegal]
00000044:  [illegal]
00000046:  [illegal]
00000048:  [illegal]
0000004a:  [illegal]
0000004c:  [illegal]
0000004e:  [illegal]
00000050:  [illegal]
00000052:  [illegal]
00000054:  [illegal]
00000056:  [illegal]
00000058:  [illegal]
0000005a:  [illegal]
0000005c:  [illegal]
0000005e:  [illegal]
00000060:  [illegal]
00000062:  [illegal]
00000064:  [illegal]
00000066:  [illegal]
00000068:  [illegal]
0000006a:  [illegal]
0000006c:  [illegal]
0000006e:  [illegal]
00000070:  [illegal]
00000072:  [illegal]
00000074:  [illegal]
00000076:  [illegal]
00000078:  [illegal]
0000007a:  [illegal]
0000007c:  [illegal]
0000007e:  [illegal]
00000080:  [illegal]
00000082:  [illegal]
00000084:  [illegal]
00000086:  [illegal]
00000088:  [illegal]
0000008a:  [illegal]
0000008c:  [illegal]
0000008e:  [illegal]
00000090:  [illegal]
00000092:  [illegal]
00000094:  [illegal]
00000096:  [illegal]
00000098:  [illegal]
0000009a:  [illegal]
0000009c:  [illegal]
0000009e:  [illegal]
000000a0:  [illegal]
000000a2:  [illegal]
000000a4:  [illegal]
000000a6:  [illegal]
000000a8:  [illegal]
000000aa:  [illegal]
000000ac:  [illegal]
000000ae:  [illegal]
000000b0:  [illegal]
000000b2:  [illegal]
000000b4:  [illegal]
000000b6:  [illegal]
000000b8:  [illegal]
000000ba:  [illegal]
000000bc:  [illegal]
000000be:  [illegal]
000000c0:  [illegal]
000000c2:  [illegal]
000000c4:  [illegal]
000000c6:  [illegal]
000000c8:  [illegal]
000000ca:  [illegal]
000000cc:  [illegal]
000000ce:  [illegal]
000000d0:  [illegal]
000000d2:  [illegal]
000000d4:  [illegal]
000000d6:  [illegal]
000000d8:  [illegal]
000000da:  [illegal]
000000dc:  [illegal]
000000de:  [illegal]
000000e0:  [illegal]
000000e2:  [illegal]
000000e4:  [illegal]
000000e6:  [illegal]
000000e8:  [illegal]
000000ea:  [illegal]
000000ec:  [illegal]
000000ee:  [illegal]
000000f0:  [illegal]
000000f2:  [illegal]
000000f4:  [illegal]
000000f6:  [illegal]
000000f8:  [illegal]
000000fa:  [illegal]
000000fc:  [illegal]
000000fe:  [illegal]
00000100:  [illegal]
00000102:  [illegal]
00000104:  [illegal]
00000106:  [illegal]
00000108:  [illegal]
0000010a:  [illegal]
0000010c:  [illegal]
0000010e:  [illegal]
00000110:  [illegal]
00000112:  [illegal]
00000114:  [illegal]
00000116:  [illegal]
00000118:  [illegal]
0000011a:  [illegal]
0000011c:  [illegal]
0000011e:  [illegal]
00000120:  [illegal]
00000122:  [illegal]
00000124:  [illegal]
00000126:  [illegal]
00000128:  [illegal]
0000012a:  [illegal]
0000012c:  [illegal]
0000012e:  [illegal]
00000130:  [illegal]
00000132:  [illegal]
00000134:  [illegal]
00000136:  [illegal]
00000138:  [illegal]
0000013a:  [illegal]
0000013c:  [illegal]
0000013e:  [illegal]
00000140:  [illegal]
00000142:  [illegal]
00000144:  [illegal]
00000146:  [illegal]
00000148:  [illegal]
0000014a:  [illegal]
0000014c:  [illegal]
0000014e:  [illegal]
00000150:  [illegal]
00000152:  [illegal]
00000154:  [illegal]
00000156:  [illegal]
00000158:  [illegal]
0000015a:  [illegal]
0000015c:  [illegal]
0000015e:  [illegal]
00000160:  [illegal]
00000162:  [illegal]
00000164:  [illegal]
00000166:  [illegal]
00000168:  [illegal]
0000016a:  [illegal]
0000016c:  [illegal]
0000016e:  [illegal]
00000170:  [illegal]
00000172:  [illegal]
00000174:  [illegal]
00000176:  [illegal]
00000178:  [illegal]
0000017a:  [illegal]
0000017c:  [illegal]
0000017e:  [illegal]
00000180:  [illegal]
00000182:  [illegal]
00000184:  [illegal]
00000186:  [illegal]
00000188:  [illegal]
0000018a:  [illegal]
0000018c:  [illegal]
0000018e:  [illegal]
00000190:  [illegal]
00000192:  [illegal]
00000194:  [illegal]
00000196:  [illegal]
00000198:  [illegal]
0000019a:  [illegal]
0000019c:  [illegal]
0000019e:  [illegal]
000001a0:  [illegal]
000001a2:  [illegal]
000001a4:  [illegal]
000001a6:  [illegal]
000001a8:  [illegal]
000001aa:  [illegal]
000001ac:  [illegal]
000001ae:  [illegal]
000001b0:  [illegal]
000001b2:  [illegal]
000001b4:  [illegal]
000001b6:  [illegal]
000001b8:  [illegal]
000001ba:  [illegal]
000001bc:  [illegal]
000001be:  [illegal]
000001c0:  [illegal]
000001c2:  [illegal]
000001c4:  [illegal]
000001c6:  [illegal]
000001c8:  [illegal]
000001ca:  [illegal]
000001cc:  [illegal]
000001ce:  [illegal]
000001d0:  [illegal]
000001d2:  [illegal]
000001d4:  [illegal]
000001d6:  [illegal]
000001d8:  [illegal]
000001da:  [illegal]
000001dc:  [illegal]
000001de:  [illegal]
000001e0:  [illegal]
000001e2:  [illegal]
000001e4:  [illegal]
000001e6:  [illegal]
000001e8:  [illegal]
000001ea:  [illegal]
000001ec:  [illegal]
000001ee:  [illegal]
000001f0:  [illegal]
000001f2:  [illegal]
000001f4:  [illegal]
000001f6:  [illegal]
000001f8:  [illegal]
000001fa:  [illegal]
000001fc:  [illegal]
000001fe:  [illegal]
00000200:  [illegal]
00000202:  [illegal]
00000204:  [illegal]
00000206:  [illegal]
00000208:  [illegal]
0000020a:  [illegal]
0000020c:  [illegal]
0000020e:  [illegal]
00000210:  [illegal]
00000212:  [illegal]
00000214:  [illegal]
00000216:  [illegal]
00000218:  [illegal]
0000021a:  [illegal]
0000021c:  [illegal]
0000021e:  [illegal]
00000220:  [illegal]
00000222:  [illegal]
00000224:  [illegal]
00000226:  [illegal]
00000228:  [illegal]
0000022a:  [illegal]
0000022c:  [illegal]
0000022e:  [illegal]
00000230:  [illegal]
00000232:  [illegal]
00000234:  [illegal]
00000236:  [illegal]
00000238:  [illegal]
0000023a:  [illegal]
0000023c:  [illegal]
0000023e:  [illegal]
00000240:  [illegal]
00000242:  [illegal]
00000244:  [illegal]
00000246:  [illegal]
00000248:  [illegal]
0000024a:  [illegal]
0000024c:  [illegal]
0000024e:  [illegal]
00000250:  [illegal]
00000252:  [illegal]
00000254:  [illegal]
00000256:  [illegal]
00000258:  [illegal]
0000025a:  [illegal]
0000025c:  [illegal]
0000025e:  [illegal]
00000260:  [illegal]
00000262:  [illegal]
00000264:  [illegal]
00000266:  [illegal]
00000268:  [illegal]
0000026a:  [illegal]
0000026c:  [illegal]
0000026e:  [illegal]
00000270:  [illegal]
00000272:  [illegal]
00000274:  [illegal]
00000276:  [illegal]
00000278:  [illegal]
0000027a:  [illegal]
0000027c:  [illegal]
0000027e:  [illegal]
00000280:  [illegal]
00000282:  [illegal]
00000284:  [illegal]
00000286:  [illegal]
00000288:  [illegal]
0000028a:  [illegal]
0000028c:  [illegal]
0000028e:  [illegal]
00000290:  [illegal]
00000292:  [illegal]
00000294:  [illegal]
00000296:  [illegal]
00000298:  [illegal]
0000029a:  [illegal]
0000029c:  [illegal]
0000029e:  [illegal]
000002a0:  [illegal]
000002a2:  [illegal]
000002a4:  [illegal]
000002a6:  [illegal]
000002a8:  [illegal]
000002aa:  [illegal]
000002ac:  [illegal]
000002ae:  [illegal]
000002b0:  [illegal]
000002b2:  [illegal]
000002b4:  [illegal]
000002b6:  [illegal]
000002b8:  [illegal]
000002ba:  [illegal]
000002bc:  [illegal]
000002be:  [illegal]
000002c0:  [illegal]
000002c2:  [illegal]
000002c4:  [illegal]
000002c6:  [illegal]
000002c8:  [illegal]
000002ca:  [illegal]
000002cc:  [illegal]
000002ce:  [illegal]
000002d0:  [illegal]
000002d2:  [illegal]
000002d4:  [illegal]
000002d6:  [illegal]
000002d8:  [illegal]
000002da:  [illegal]
000002dc:  [illegal]
000002de:  [illegal]
000002e0:  [illegal]
000002e2:  [illegal]
000002e4:  [illegal]
000002e6:  [illegal]
000002e8:  [illegal]
000002ea:  [illegal]
000002ec:  [illegal]
000002ee:  [illegal]
000002f0:  [illegal]
000002f2:  [illegal]
000002f4:  [illegal]
000002f6:  [illegal]
000002f8:  [illegal]
000002fa:  [illegal]
000002fc:  [illegal]
000002fe:  [illegal]
00000300:  [illegal]
00000302:  [illegal]
00000304:  [illegal]
00000306:  [illegal]
00000308:  [illegal]
0000030a:  [illegal]
0000030c:  [illegal]
0000030e:  [illegal]
00000310:  [illegal]
00000312:  [illegal]
00000314:  [illegal]
00000316:  [illegal]
00000318:  [illegal]
0000031a:  [illegal]
0000031c:  [illegal]
0000031e:  [illegal]
00000320:  [illegal]
00000322:  [illegal]
00000324:  [illegal]
00000326:  [illegal]
00000328:  [illegal]
0000032a:  [illegal]
0000032c:  [illegal]
0000032e:  [illegal]
00000330:  [illegal]
00000332:  [illegal]
00000334:  [illegal]
00000336:  [illegal]
00000338:  [illegal]
0000033a:  [illegal]
0000033c:  [illegal]
0000033e:  [illegal]
00000340:  [illegal]
00000342:  [illegal]
00000344:  [illegal]
00000346:  [illegal]
00000348:  [illegal]
0000034a:  [illegal]
0000034c:  [illegal]
0000034e:  [illegal]
00000350:  [illegal]
00000352:  [illegal]
00000354:  [illegal]
00000356:  [illegal]
00000358:  [illegal]
0000035a:  [illegal]
0000035c:  [illegal]
0000035e:  [illegal]
00000360:  [illegal]
00000362:  [illegal]
00000364:  [illegal]
00000366:  [illegal]
00000368:  [illegal]
0000036a:  [illegal]
0000036c:  [illegal]
0000036e:  [illegal]
00000370:  [illegal]
00000372:  [illegal]
00000374:  [illegal]
00000376:  [illegal]
00000378:  [illegal]
0000037a:  [illegal]
0000037c:  [illegal]
0000037e:  [illegal]
00000380:  [illegal]
00000382:  [illegal]
00000384:  [illegal]
00000386:  [illegal]
00000388:  [illegal]
0000038a:  [illegal]
0000038c:  [illegal]
0000038e:  [illegal]
00000390:  [illegal]
00000392:  [illegal]
00000394:  [illegal]
00000396:  [illegal]
00000398:  [illegal]
0000039a:  [illegal]
0000039c:  [illegal]
0000039e:  [illegal]
000003a0:  [illegal]
000003a2:  [illegal]
000003a4:  [illegal]
000003a6:  [illegal]
000003a8:  [illegal]
000003aa:  [illegal]
000003ac:  [illegal]
000003ae:  [illegal]
000003b0:  [illegal]
000003b2:  [illegal]
000003b4:  [illegal]
000003b6:  [illegal]
000003b8:  [illegal]
000003ba:  [illegal]
000003bc:  [illegal]
000003be:  [illegal]
000003c0:  [illegal]
000003c2:  [illegal]
000003c4:  [illegal]
000003c6:  [illegal]
000003c8:  [illegal]
000003ca:  [illegal]
000003cc:  [illegal]
000003ce:  [illegal]
000003d0:  [illegal]
000003d2:  [illegal]
000003d4:  [illegal]
000003d6:  [illegal]
000003d8:  [illegal]
000003da:  [illegal]
000003dc:  [illegal]
000003de:  [illegal]
000003e0:  [illegal]
000003e2:  [illegal]
000003e4:  [illegal]
000003e6:  [illegal]
000003e8:  [illegal]
000003ea:  [illegal]
000003ec:  [illegal]
000003ee:  [illegal]
000003f0:  [illegal]
000003f2:  [illegal]
000003f4:  [illegal]
000003f6:  [illegal]
000003f8:  [illegal]
000003fa:  [illegal]
000003fc:  [illegal]
000003fe:  [illegal]
00000400:  [illegal]
00000402:  [illegal]
00000404:  [illegal]
00000406:  [illegal]
00000408:  [illegal]
0000040a:  [illegal]
0000040c:  [illegal]
0000040e:  [illegal]
00000410:  [illegal]
00000412:  [illegal]
00000414:  [illegal]
00000416:  [illegal]
00000418:  [illegal]
0000041a:  [illegal]
0000041c:  [illegal]
0000041e:  [illegal]
00000420:  [illegal]
00000422:  [illegal]
00000424:  [illegal]
00000426:  [illegal]
00000428:  [illegal]
0000042a:  [illegal]
0000042c:  [illegal]
0000042e:  [illegal]
00000430:  [illegal]
00000432:  [illegal]
00000434:  [illegal]
00000436:  [illegal]
00000438:  [illegal]
0000043a:  [illegal]
0000043c:  [illegal]
0000043e:  [illegal]
00000440:  [illegal]
00000442:  [illegal]
00000444:  [illegal]
00000446:  [illegal]
00000448:  [illegal]
0000044a:  [illegal]
0000044c:  [illegal]
0000044e:  [illegal]
00000450:  [illegal]
00000452:  [illegal]
00000454:  [illegal]
00000456:  [illegal]
00000458:  [illegal]
0000045a:  [illegal]
0000045c:  [illegal]
0000045e:  [illegal]
00000460:  [illegal]
00000462:  [illegal]
00000464:  [illegal]
00000466:  [illegal]
00000468:  [illegal]
0000046a:  [illegal]
0000046c:  [illegal]
0000046e:  [illegal]
00000470:  [illegal]
00000472:  [illegal]
00000474:  [illegal]
00000476:  [illegal]
00000478:  [illegal]
0000047a:  [illegal]
0000047c:  [illegal]
0000047e:  [illegal]
00000480:  [illegal]
00000482:  [illegal]
00000484:  [illegal]
00000486:  [illegal]
00000488:  [illegal]
0000048a:  [illegal]
0000048c:  [illegal]
0000048e:  [illegal]
00000490:  [illegal]
00000492:  [illegal]
00000494:  [illegal]
00000496:  [illegal]
00000498:  [illegal]
0000049a:  [illegal]
0000049c:  [illegal]
0000049e:  [illegal]
000004a0:  [illegal]
000004a2:  [illegal]
000004a4:  [illegal]
000004a6:  [illegal]
000004a8:  [illegal]
000004aa:  [illegal]
000004ac:  [illegal]
000004ae:  [illegal]
000004b0:  [illegal]
000004b2:  [illegal]
000004b4:  [illegal]
000004b6:  [illegal]
000004b8:  [illegal]
000004ba:  [illegal]
000004bc:  [illegal]
000004be:  [illegal]
000004c0:  [illegal]
000004c2:  [illegal]
000004c4:  [illegal]
000004c6:  [illegal]
000004c8:  [illegal]
000004ca:  [illegal]
000004cc:  [illegal]
000004ce:  [illegal]
000004d0:  [illegal]
000004d2:  [illegal]
000004d4:  [illegal]
000004d6:  [illegal]
000004d8:  [illegal]
000004da:  [illegal]
000004dc:  [illegal]
000004de:  [illegal]
000004e0:  [illegal]
000004e2:  [illegal]
000004e4:  [illegal]
000004e6:  [illegal]
000004e8:  [illegal]
000004ea:  [illegal]
000004ec:  [illegal]
000004ee:  [illegal]
000004f0:  [illegal]
000004f2:  [illegal]
000004f4:  [illegal]
000004f6:  [illegal]
000004f8:  [illegal]
000004fa:  [illegal]
000004fc:  [illegal]
000004fe:  [illegal]
00000500:  [illegal]
00000502:  [illegal]
00000504:  [illegal]
00000506:  [illegal]
00000508:  [illegal]
0000050a:  [illegal]
0000050c:  [illegal]
0000050e:  [illegal]
00000510:  [illegal]
00000512:  [illegal]
00000514:  [illegal]
00000516:  [illegal]
00000518:  [illegal]
0000051a:  [illegal]
0000051c:  [illegal]
0000051e:  [illegal]
00000520:  [illegal]
00000522:  [illegal]
00000524:  [illegal]
00000526:  [illegal]
00000528:  [illegal]
0000052a:  [illegal]
0000052c:  [illegal]
0000052e:  [illegal]
00000530:  [illegal]
00000532:  [illegal]
00000534:  [illegal]
00000536:  [illegal]
00000538:  [illegal]
0000053a:  [illegal]
0000053c:  [illegal]
0000053e:  [illegal]
00000540:  [illegal]
00000542:  [illegal]
00000544:  [illegal]
00000546:  [illegal]
00000548:  [illegal]
0000054a:  [illegal]
0000054c:  [illegal]
0000054e:  [illegal]
00000550:  [illegal]
00000552:  [illegal]
00000554:  [illegal]
00000556:  [illegal]
00000558:  [illegal]
0000055a:  [illegal]
0000055c:  [illegal]
0000055e:  [illegal]
00000560:  [illegal]
00000562:  [illegal]
00000564:  [illegal]
00000566:  [illegal]
00000568:  [illegal]
0000056a:  [illegal]
0000056c:  [illegal]
0000056e:  [illegal]
00000570:  [illegal]
00000572:  [illegal]
00000574:  [illegal]
00000576:  [illegal]
00000578:  [illegal]
0000057a:  [illegal]
0000057c:  [illegal]
0000057e:  [illegal]
00000580:  [illegal]
00000582:  [illegal]
00000584:  [illegal]
00000586:  [illegal]
00000588:  [illegal]
0000058a:  [illegal]
0000058c:  [illegal]
0000058e:  [illegal]
00000590:  [illegal]
00000592:  [illegal]
00000594:  [illegal]
00000596:  [illegal]
00000598:  [illegal]
0000059a:  [illegal]
0000059c:  [illegal]
0000059e:  [illegal]
000005a0:  [illegal]
000005a2:  [illegal]
000005a4:  [illegal]
000005a6:  [illegal]
000005a8:  [illegal]
000005aa:  [illegal]
000005ac:  [illegal]
000005ae:  [illegal]
000005b0:  [illegal]
000005b2:  [illegal]
000005b4:  [illegal]
000005b6:  [illegal]
000005b8:  [illegal]
000005ba:  [illegal]
000005bc:  [illegal]
000005be:  [illegal]
000005c0:  [illegal]
000005c2:  [illegal]
000005c4:  [illegal]
000005c6:  [illegal]
000005c8:  [illegal]
000005ca:  [illegal]
000005cc:  [illegal]
000005ce:  [illegal]
000005d0:  [illegal]
000005d2:  [illegal]
000005d4:  [illegal]
000005d6:  [illegal]
000005d8:  [illegal]
000005da:  [illegal]
000005dc:  [illegal]
000005de:  [illegal]
000005e0:  [illegal]
000005e2:  [illegal]
000005e4:  [illegal]
000005e6:  [illegal]
000005e8:  [illegal]
000005ea:  [illegal]
000005ec:  [illegal]
000005ee:  [illegal]
000005f0:  [illegal]
000005f2:  [illegal]
000005f4:  [illegal]
000005f6:  [illegal]
000005f8:  [illegal]
000005fa:  [illegal]
000005fc:  [illegal]
000005fe:  [illegal]
00000600:  [illegal]
00000602:  [illegal]
00000604:  [illegal]
00000606:  [illegal]
00000608:  [illegal]
0000060a:  [illegal]
0000060c:  [illegal]
0000060e:  [illegal]
00000610:  [illegal]
00000612:  [illegal]
00000614:  [illegal]
00000616:  [illegal]
00000618:  [illegal]
0000061a:  [illegal]
0000061c:  [illegal]
0000061e:  [illegal]
00000620:  [illegal]
00000622:  [illegal]
00000624:  [illegal]
00000626:  [illegal]
00000628:  [illegal]
0000062a:  [illegal]
0000062c:  [illegal]
0000062e:  [illegal]
00000630:  [illegal]
00000632:  [illegal]
00000634:  [illegal]
00000636:  [illegal]
00000638:  [illegal]
0000063a:  [illegal]
0000063c:  [illegal]
0000063e:  [illegal]
00000640:  [illegal]
00000642:  [illegal]
00000644:  [illegal]
00000646:  [illegal]
00000648:  [illegal]
0000064a:  [illegal]
0000064c:  [illegal]
0000064e:  [illegal]
00000650:  [illegal]
00000652:  [illegal]
00000654:  [illegal]
00000656:  [illegal]
00000658:  [illegal]
0000065a:  [illegal]
0000065c:  [illegal]
0000065e:  [illegal]
00000660:  [illegal]
00000662:  [illegal]
00000664:  [illegal]
00000666:  [illegal]
00000668:  [illegal]
0000066a:  [illegal]
0000066c:  [illegal]
0000066e:  [illegal]
00000670:  [illegal]
00000672:  [illegal]
00000674:  [illegal]
00000676:  [illegal]
00000678:  [illegal]
0000067a:  [illegal]
0000067c:  [illegal]
0000067e:  [illegal]
00000680:  [illegal]
00000682:  [illegal]
00000684:  [illegal]
00000686:  [illegal]
00000688:  [illegal]
0000068a:  [illegal]
0000068c:  [illegal]
0000068e:  [illegal]
00000690:  [illegal]
00000692:  [illegal]
00000694:  [illegal]
00000696:  [illegal]
00000698:  [illegal]
0000069a:  [illegal]
0000069c:  [illegal]
0000069e:  [illegal]
000006a0:  [illegal]
000006a2:  [illegal]
000006a4:  [illegal]
000006a6:  [illegal]
000006a8:  [illegal]
000006aa:  [illegal]
000006ac:  [illegal]
000006ae:  [illegal]
000006b0:  [illegal]
000006b2:  [illegal]
000006b4:  [illegal]
000006b6:  [illegal]
000006b8:  [illegal]
000006ba:  [illegal]
000006bc:  [illegal]
000006be:  [illegal]
000006c0:  [illegal]
000006c2:  [illegal]
000006c4:  [illegal]
000006c6:  [illegal]
000006c8:  [illegal]
000006ca:  [illegal]
000006cc:  [illegal]
000006ce:  [illegal]
000006d0:  [illegal]
000006d2:  [illegal]
000006d4:  [illegal]
000006d6:  [illegal]
000006d8:  [illegal]
000006da:  [illegal]
000006dc:  [illegal]
000006de:  [illegal]
000006e0:  [illegal]
000006e2:  [illegal]
000006e4:  [illegal]
000006e6:  [illegal]
000006e8:  [illegal]
000006ea:  [illegal]
000006ec:  [illegal]
000006ee:  [illegal]
000006f0:  [illegal]
000006f2:  [illegal]
000006f4:  [illegal]
000006f6:  [illegal]
000006f8:  [illegal]
000006fa:  [illegal]
000006fc:  [illegal]
000006fe:  [illegal]
00000700:  [illegal]
00000702:  [illegal]
00000704:  [illegal]
00000706:  [illegal]
00000708:  [illegal]
0000070a:  [illegal]
0000070c:  [illegal]
0000070e:  [illegal]
00000710:  [illegal]
00000712:  [illegal]
00000714:  [illegal]
00000716:  [illegal]
00000718:  [illegal]
0000071a:  [illegal]
0000071c:  [illegal]
0000071e:  [illegal]
00000720:  [illegal]
00000722:  [illegal]
00000724:  [illegal]
00000726:  [illegal]
00000728:  [illegal]
0000072a:  [illegal]
0000072c:  [illegal]
0000072e:  [illegal]
00000730:  [illegal]
00000732:  [illegal]
00000734:  [illegal]
00000736:  [illegal]
00000738:  [illegal]
0000073a:  [illegal]
0000073c:  [illegal]
0000073e:  [illegal]
00000740:  [illegal]
00000742:  [illegal]
00000744:  [illegal]
```

这样的一个规则，每个元素都由十字形，而且它们的邻居都不同。如果我们在这个图上进行测试，只要发现输出有十字形不正确，都可以大概率检查出来。为了得到图的输出，我们需要用考测试点的图的行为，这么说最难的部分就是如何生成这样的图。这个问题可以用下面的算法来解决：

- 第一段是直接写入正确的元素，这一段应该用到哪些测试点呢？测试点的输出是 `assert(cell == i + 3)`
- 第二段是直接写入的区域长度，这一段应该用到哪些测试点的输出行为有关
- 第三段是直接写入正确的元素，这一段应该用到哪些测试点呢？测试点的输出是 `assert(cell == i + 3)`

于是我们可以编写出下面的测试点：

```
// 检查行,列不同的情况是否正确,cell = i, val = i,
void check_row(int i, int r, int col){
    int cl;
    for (i = 1; i < r; i ++) {
        assert(cell[i] == val + i - 1);
    }
}

// 检查行,列不同的情况
void check_col(int i, int r, int col){
    int cl;
    for (i = 1; i < r; i ++) {
        assert(cell[i] == val);
    }
}

每个正确产生的，都会有一个测试点来检验输入，并且使用最短编写出测试点的数的明确输出。于是我们可 memset() ，我们可以编写写下的测试代码：

void test_memset() {
    int i, r;
    for (i = 1; i < N; i ++) {
        for (r = 1; r < N; r ++) {
            reset();
            cinit(i, val + (i * r) / 2);
            memset(data + i, val, r - 1);
            check_sep(i, 1, 1);
            check_sep(i, r, val);
            check_sep(r, N, r + 1);
        }
    }
}
```

✓ 编写更多测试

现试解决上述测试结果的程序的测试的，并把 `an-kernel/tests/` 目录下新增一个对应的测试集 `kls-tests` ，测试集的文件结构可以参考 `an-kernel/tests/an-tests` 和 `an-kernel/tests/kls-tests` 。

然后针对上文所说的问题，以及编写能检测测试的代码，我们可以编写新的测试集要求第一部分如下：

- memcpy() 的行为应对覆盖的时候输出，可以在覆盖的时候检查区间是否覆盖，覆盖，到期区一定会发现，能覆盖覆盖的区间的覆盖率为 `100%` ，这样就不会出现覆盖的情况
- 字符串的覆盖覆盖要求注意 `\0` 和最终覆盖的区间

编写后，你可以先在 `outline.txt` 用gdb的调试命令来测试你编写的测试代码，然后在 `outline.txt` 用这些测试代码来测试你的内容，最后用 `make kls-test` 命令来测试上述提交的测试集测试你的NEMU实现。

✓ 设置覆盖率目标

设定覆盖率目标的结果是明确的测试的，并在 `an-kernel/scripts/` 目录下新增一个对应的测试集 `kls-tests` ，测试集的文件结构可以参考 `an-kernel/tests/an-tests` 和 `an-kernel/tests/kls-tests` 。

然后针对上文所说的问题，以及编写能检测测试的代码，我们可以编写新的测试集要求第二部分如下：

- memcpy() 的行为应对覆盖的时候输出，可以在覆盖的时候检查区间是否覆盖，覆盖，到期区一定会发现，能覆盖覆盖的区间的覆盖率为 `100%` ，这样就不会出现覆盖的情况
- 字符串的覆盖覆盖要求注意 `\0` 和最终覆盖的区间

编写后，你可以先在 `outline.txt` 用gdb的调试命令来测试你编写的测试代码，然后在 `outline.txt` 用这些测试代码来测试你的内容，最后用 `make kls-test` 命令来测试上述提交的测试集测试你的NEMU实现。

✓ 设置覆盖率目标

设定覆盖率目标的结果是明确的测试的，并在 `an-kernel/scripts/` 目录下新增一个对应的测试集 `kls-tests` ，测试集的文件结构可以参考 `an-kernel/tests/an-tests` 和 `an-kernel/tests/kls-tests` 。

然后针对上文所说的问题，以及编写能检测测试的代码，我们可以编写新的测试集要求第三部分如下：

- memset() 的行为应对覆盖的时候输出，可以在覆盖的时候检查区间是覆盖，覆盖，到期区一定会发现，能覆盖覆盖的区间的覆盖率为 `100%` ，这样就不会出现覆盖的情况
- 字符串的覆盖覆盖要求注意 `\0` 和最终覆盖的区间

编写后，你可以先在 `outline.txt` 用gdb的调试命令来测试你编写的测试代码，然后在 `outline.txt` 用这些测试代码来测试你的内容，最后用 `make kls-test` 命令来测试上述提交的测试集测试你的NEMU实现。

✓ 设置覆盖率目标

设定覆盖率目标的结果是明确的测试的，并在 `an-kernel/scripts/` 目录下新增一个对应的测试集 `kls-tests` ，测试集的文件结构可以参考 `an-kernel/tests/an-tests` 和 `an-kernel/tests/kls-tests` 。

然后针对上文所说的问题，以及编写能检测测试的代码，我们可以编写新的测试集要求第四部分如下：

- memset() 的行为应对覆盖的时候输出，可以在覆盖的时候检查区间是覆盖，覆盖，到期区一定会发现，能覆盖覆盖的区间的覆盖率为 `100%` ，这样就不会出现覆盖的情况
- 字符串的覆盖覆盖要求注意 `\0` 和最终覆盖的区间

编写后，你可以先在 `outline.txt` 用gdb的调试命令来测试你编写的测试代码，然后在 `outline.txt` 用这些测试代码来测试你的内容，最后用 `make kls-test` 命令来测试上述提交的测试集测试你的NEMU实现。

✓ 设置覆盖率目标

设定覆盖率目标的结果是明确的测试的，并在 `an-kernel/scripts/` 目录下新增一个对应的测试集 `kls-tests` ，测试集的文件结构可以参考 `an-kernel/tests/an-tests` 和 `an-kernel/tests/kls-tests` 。

Differential Testing

理解每个的执行过程之后，应当参考每个要求的意义和实现，工程实现推荐使用Debug，实现不正确的时候经常进行测试，实现不正确的时候，可以参考下面的提示。思考一下，详细的时候有那些东西与NEMU有关？有哪些东西是NEMU有关的？思考一下，哪些东西是NEMU有关的？哪些东西是NEMU有关的？哪些东西是NEMU有关的？

输入模型
设备IO
设备IO示例
输入输出
NEMU中的输入输出
IOE
IOE示例
RTC模拟
IDE模拟
IDE示例
设备驱动模型
IDE
IDE驱动
IDE驱动示例
总结

输入输出

我们已经实现了运行 cpu-tests 所需的一切硬件，但这些测试用例只能在裸机环境下运行。要运行更复杂的程序，我们还需要实现操作系统，而操作系统需要通过硬件来实现进程的调度。本章将介绍如何通过实现一个简单的设备来实现 Hello World 程序，并在此基础上实现一个简单的文本编辑器。在实现这些设备的过程中，我们会介绍一些常见的设备的工作原理。

设备IO

设备的工作原理非常复杂，但从软件的角度来看，设备的工作原理可以简单地理解为通过某些特殊的指令和内存区域来与设备进行交互。例如，在第 2 章实现的第一个程序中，我们通过向某个内存区域写入字符来实现输出字符的功能，这是最简单的设备IO的实现方式之一。这一简单的设备IO实现方式存在一个问题：程序需要知道内存区域的一个固定地址，这使得程序的可移植性受到了一定的限制。为了解决这一问题，我们可以通过在 nemu/include/common.h 中定义一个宏来实现对内存区域的抽象：

```
#define DEVICE_BASE 0xa0000000

#define MMIO(x) (DEVICE_BASE + (x))
```

这样一来，程序只需要知道设备的偏移量，而不必关心设备的具体地址。例如，在 Hello World 程序中，我们可以通过 `MMIO(1)` 来访问第 1 号设备的内存区域。这样一来，程序的可移植性就得到了一定的保证。

上述内存映射方式的实现方式非常简单，但在实际情况下，设备的内存区域可能会比较大，这会给内存管理带来一定的困难。为了解决这一问题，我们可以通过在 nemu/include/common.h 中定义一个宏来实现对设备内存区域的抽象：

```
#define PMEM_SIZE (128 * 1024 * 1024)

#define MMIO(x) (PMEM_SIZE + (x))
```

上述代码表明，设备内存区域紧接在物理内存区域的后面。这样一来，设备内存区域的大小就不会对物理内存区域的大小产生影响，这样就解决了上述问题。

除了内存映射IO，还有一种常见的设备IO实现方式：通过特殊的指令来访问设备内存区域。这种方式的优点是，设备内存区域可以被映射到物理内存区域之外，这样就不会对物理内存区域的大小产生影响。例如，在 x86 架构中，我们可以通过 `in` 和 `out` 指令来访问设备。例如，`in 0x92, %al` 指令表示从 0x92 号设备的内存区域中读取一个字节到寄存器 `%al` 中，而 `out %al, 0x92` 则表示将 `%al` 寄存器的内容写入 0x92 号设备的内存区域。

输入输出模型

输入输出（Input/Output，简称 IO）是计算机系统中一种重要的功能，它允许计算机与外部世界进行交互。输入输出模型通常包括以下几个方面：

- 输入设备：用户通过输入设备（如键盘、鼠标、触摸屏等）向计算机输入数据。
- 输出设备：计算机通过输出设备（如显示器、打印机、扬声器等）将数据输出给用户。
- 数据传输：输入设备和输出设备通过数据传输线与计算机连接，数据通过这些线路在设备和计算机之间传输。
- 控制信号：计算机通过控制信号来控制输入设备和输出设备的工作，例如开始输入、结束输入、开始输出、结束输出等。
- 缓冲区：为了提高效率，计算机通常会使用缓冲区来暂存输入数据和输出数据，以减少数据传输的次数。
- 中断：当输入设备有数据输入或输出设备准备好接收数据时，它们会向计算机发送中断信号，以通知计算机处理这些数据。

设备IO示例

在开始实现设备IO之前，我们先来实现一个简单的设备IO示例：在 nemu/include/common.h 中定义一个宏来实现对设备内存区域的抽象：

```
#define MMIO(x) (0x10000000 + (x))
```

在 nemu/test/hello.s 中，我们可以通过以下方式来实现输出字符的功能：

```
.text
.global _start
_start:
    movl $0x48, %eax
    movl $0x10000000, %edx
    movb %al, (%edx)
    movl $0x65, %eax
    movl $0x10000001, %edx
    movb %al, (%edx)
    movl $0x6c, %eax
    movl $0x10000002, %edx
    movb %al, (%edx)
    movl $0x6c, %eax
    movl $0x10000003, %edx
    movb %al, (%edx)
    movl $0x6f, %eax
    movl $0x10000004, %edx
    movb %al, (%edx)
    movl $0x0a, %eax
    movl $0x10000005, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000006, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000007, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000008, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000009, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000000f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000010, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000011, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000012, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000013, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000014, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000015, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000016, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000017, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000018, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000019, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000001f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000020, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000021, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000022, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000023, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000024, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000025, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000026, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000027, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000028, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000029, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000002f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000030, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000031, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000032, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000033, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000034, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000035, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000036, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000037, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000038, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000039, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000003f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000040, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000041, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000042, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000043, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000044, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000045, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000046, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000047, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000048, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000049, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000004f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000050, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000051, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000052, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000053, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000054, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000055, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000056, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000057, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000058, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000059, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000005f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000060, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000061, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000062, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000063, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000064, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000065, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000066, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000067, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000068, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000069, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000006f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000070, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000071, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000072, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000073, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000074, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000075, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000076, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000077, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000078, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000079, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000007f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000080, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000081, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000082, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000083, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000084, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000085, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000086, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000087, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000088, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000089, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000008f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000090, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000091, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000092, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000093, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000094, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000095, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000096, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000097, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000098, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000099, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000009f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000a9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000aa, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ab, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ac, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ad, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ae, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000af, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000b9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ba, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000bb, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000bc, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000bd, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000be, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000bf, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000c9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ca, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000cb, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000cc, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000cd, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ce, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000cf, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000d9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000da, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000db, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000dc, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000dd, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000de, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000df, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000e9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ea, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000eb, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ec, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ed, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ee, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ef, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f0, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f1, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f2, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f3, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f4, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f5, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f6, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f7, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f8, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000f9, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000fa, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000fb, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000fc, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000fd, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000fe, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x100000ff, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000100, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000101, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000102, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000103, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000104, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000105, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000106, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000107, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000108, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000109, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000010f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000110, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000111, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000112, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000113, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000114, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000115, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000116, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000117, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000118, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000119, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000011f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000120, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000121, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000122, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000123, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000124, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000125, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000126, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000127, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000128, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000129, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000012f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000130, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000131, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000132, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000133, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000134, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000135, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000136, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000137, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000138, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000139, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000013f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000140, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000141, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000142, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000143, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000144, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000145, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000146, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000147, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000148, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000149, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000014f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000150, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000151, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000152, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000153, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000154, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000155, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000156, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000157, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000158, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000159, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000015f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000160, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000161, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000162, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000163, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000164, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000165, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000166, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000167, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000168, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000169, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016a, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016b, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016c, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016d, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016e, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x1000016f, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000170, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000171, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000172, %edx
    movb %al, (%edx)
    movl $0x00, %eax
    movl $0x10000173, %ed
```

1 BIOS中断调用表查询

中断调用表查询，是指通过查阅特定资料，以获取与硬件交互所需的中断号、功能号（AH）、参数（AX/BX/CX/DX）、返回值等信息，是编写底层程序（如引导扇区、内核、驱动）的核心工具。

1.1 通用查询流程（适用于任何中断）

1. 确定目标硬件：明确要操作的设备（如磁盘、键盘、显卡）。
2. 查阅资料：优先顺序：① 官方文档（如 Ralf Brown 的 Interrupt List）；② 硬件厂商手册（如 Intel 芯片组手册）；③ 社区资料（如 OSDev.org、Stackoverflow）。
3. 筛选关键信息：从资料中提取：中断号（如 13h 磁盘中断）、功能号（AH 寄存器值）、参数（AX/BX/CX/DX 寄存器含义）、返回值（如 AX 成功/失败标志）、错误码（如 AH=01h 表示参数错误）。
4. 验证兼容性：确认该中断在目标硬件上可用（部分中断仅在特定主板/显卡上支持）。

1.2 磁盘操作中断（13h）

磁盘操作中断（13h）是 BIOS 提供的通用磁盘操作接口，支持软盘、硬盘、U 盘（模拟为硬盘）、光盘（模拟为硬盘）等设备。

1.2.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
00h	重置磁盘系统	DL=驱动器号（00h=软盘 A，80h=硬盘 C）	CF=0 成功，CF=1 失败
02h	读扇区	AL=扇区数；CH=柱面；CL=扇区；DH=磁头；DL=驱动器号；ES:BX=缓冲区地址	CF=0 成功，AL=实际读取扇区数；CF=1 失败
03h	写扇区	AL=扇区数；CH=柱面；CL=扇区；DH=磁头；DL=驱动器号；ES:BX=缓冲区地址	CF=0 成功，AL=实际写入扇区数；CF=1 失败
08h	获取驱动器参数	DL=驱动器号	CF=0 成功，CH=最大柱面；CL=最大扇区；DH=最大磁头；DL=驱动器数量；ES:DI=磁盘参数表地址；CF=1 失败
15h	检测磁盘类型	DL=驱动器号	CF=0 成功，AH=磁盘类型（00h=无盘，01h=软盘，02h=硬盘，03h=光盘）；CF=1 失败

1.2.2 关键参数说明

- 柱面（CH）：最大值 1023（10 位）。
- 扇区（CL）：从 1 开始（不是 0）。
- 磁头（DH）：从 0 开始。
- 缓冲区（ES:BX）：必须对齐（如 512 字节扇区需对齐 512 字节）。

1.2.3 错误码（AH 寄存器值）

AH	含义
00h	成功
01h	参数错误
02h	地址标记未找到
04h	扇区未找到
05h	重置失败
07h	驱动器参数活动失败
08h	缓冲区溢出
09h	尝试跨 64KB 边界
0Ch	不支持的磁道/扇区/磁头
10h	数据校验错误（CRC）
20h	控制器失败
40h	寻道失败
80h	超时
AAh	驱动器未就绪
BBh	未定义错误
CCh	写保护
E0h	状态寄存器错误

1.2.4 实用技巧

- 检测磁盘是否存在：调用 08h 检查 CF 标志，若 CF=0 则磁盘存在。
- 读取 MBR：调用 02h，CH=0，CL=1，DH=0，DL=80h，ES:BX=0000:7C00。
- 写入 MBR：调用 03h，CH=0，CL=1，DH=0，DL=80h，ES:BX=0000:7C00（需谨慎，易破坏分区表）。
- 错误重试：若失败，重置磁盘（00h）后重试 3 次。

1.3 键盘操作中断（16h）

键盘操作中断（16h）是 BIOS 提供的键盘输入接口，支持按键读取、状态检测、缓冲区管理。

1.3.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
00h	读取按键（阻塞）		AL=ASCII 码；AH=扫描码
01h	检测按键（非阻塞）		ZF=0 有按键；ZF=1 无按键；AL=ASCII 码；AH=扫描码
02h	获取键盘状态		AL=状态标志（bit0=右 Shift，bit1=左 Shift，bit2=Ctrl，bit3=Alt，bit4=Scroll Lock，bit5=Num Lock，bit6=Caps Lock，bit7=Insert）
05h	写入按键到缓冲区	CL=ASCII 码；CH=扫描码	AL=0 成功，AL=1 缓冲区满
10h	读扩展按键（增强键盘）		AL=ASCII 码；AH=扫描码
11h	检测扩展按键（增强键盘）		ZF=0 有按键；ZF=1 无按键；AL=ASCII 码；AH=扫描码
12h	获取扩展键盘状态（增强键盘）		AL=状态标志（bit0=右 Shift，bit1=左 Shift，bit2=Ctrl，bit3=Alt，bit4=Scroll Lock，bit5=Num Lock，bit6=Caps Lock，bit7=Insert，bit8=左 Ctrl，bit9=左 Alt，bit10=右 Ctrl，bit11=右 Alt）

1.3.2 关键参数说明

- 扫描码（AH）：硬件相关，用于区分按键（如 1 和 Shift+1 扫描码不同）。
- ASCII 码（AL）：0 表示无可打印字符（如 F1、方向键）。
- 缓冲区：最大 15 个按键，满后丢弃新按键。

1.3.3 实用技巧

- 检测按键：调用 01h，若 ZF=0 则有按键，调用 00h 读取。
- 模拟按键：调用 05h，将 ASCII 码和扫描码写入缓冲区。
- 检测 Ctrl+Alt+Del：调用 02h，检查 AL 的 bit2（Ctrl）和 bit3（Alt）。
- 兼容性：优先用 00h/01h，避免用 10h/11h/12h（旧键盘不支持）。

1.4 显示输出中断（10h）

显示输出中断（10h）是 BIOS 提供的屏幕显示接口，支持文本模式、图形模式、光标控制、滚动、调色板等功能。

1.4.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
00h	设置显示模式	AL=模式（03h=80×25 彩色文本，13h=320×200 256 色图形）	
01h	设置光标形状	CH=开始扫描线；CL=结束扫描线	
02h	设置光标位置	DH=行；DL=列；BH=页	
03h	获取光标位置	BH=页	DH=行；DL=列；CH=开始扫描线；CL=结束扫描线
05h	设置显示页	AL=页	
06h	向上滚动窗口	AL=行数（0=全屏）；BH=属性；CH=左上角行；CL=左上角列；DH=右下角行；DL=右下角列	
07h	向下滚动窗口	AL=行数（0=全屏）；BH=属性；CH=左上角行；CL=左上角列；DH=右下角行；DL=右下角列	
08h	读字符+属性	BH=页	AL=字符；AH=属性
09h	写字符+属性	AL=字符；BH=页；BL=属性；CX=重复次数	
0Ah	写字符（无属性）	AL=字符；BH=页；CX=重复次数	
0Eh	写字符（TTY 模式）	AL=字符；BH=页；BL=前景色	
0Fh	获取显示模式		AL=模式；AH=列数；BH=页
10h	设置调色板	AL=子功能；BX=参数	
12h	获取显示信息	BL=子功能	视子功能而定
13h	写字符+属性（扩展）	AL=字符；BH=页；BL=属性；CX=重复次数；ES:BP=字符串地址	

1.4.2 关键参数说明

- 模式（AL）：03h（80×25 彩色文本）、13h（320×200 256 色图形）。
- 属性（BL）：前景色（低 4 位）+ 背景色（高 4 位）。
- 窗口坐标：左上角（CH=0，CL=0）到右下角（DH=24，DL=79）。

1.4.3 实用技巧

- 清屏：调用 06h，AL=0，BH=07h，CH=0，CL=0，DH=24，DL=79。
- 隐藏光标：调用 01h，CH=20h。
- 显示字符串：调用 13h，AL=01h（更新光标）。
- 兼容性：优先用 00h/01h/02h/03h/06h/07h/08h/09h/0Ah/0Eh/0Fh，避免用 10h/12h/13h（旧显卡不支持）。

1.5 磁盘扩展中断（13h 扩展）

磁盘扩展中断（13h 扩展）是 BIOS 提供的现代磁盘接口，支持大容量磁盘（>8GB）、LBA 寻址、异步操作、设备信息查询。

1.5.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
41h	检测扩展支持	DL=驱动器号	CF=0 支持，BX=AA55h，AH=版本
42h	扩展读扇区	DL=驱动器号；DS:SI=DAP 地址	CF=0 成功，AL=实际读取扇区数；CF=1 失败
43h	扩展写扇区	DL=驱动器号；DS:SI=DAP 地址	CF=0 成功，AL=实际写入扇区数；CF=1 失败
48h	扩展获取驱动器参数	DL=驱动器号；DS:SI=结果缓冲区	CF=0 成功，结果写入 DS:SI；CF=1 失败
49h	扩展检测磁盘类型	DL=驱动器号	CF=0 成功，AH=磁盘类型（00h=无盘，01h=软盘，02h=硬盘，03h=光盘）；CF=1 失败

1.5.2 关键参数说明

- DAP（Disk Address Packet）：
  - 大小：16 字节。
  - 内容：扇区数、缓冲区地址、起始扇区（LBA）。
- 柱面/磁头/扇区（CHS）：
  - 最大：1024 柱面、256 磁头、63 扇区。
- LBA（逻辑块寻址）：
  - 最大：2^48 扇区（128PB）。
  - 优势：简单、高效、支持大容量。

1.5.3 实用技巧

- 检测扩展支持：调用 41h，检查 BX=AA55h。
- 读取 MBR：调用 42h，DS:SI 指向 DAP，扇区=0，缓冲区=0000:7C00。
- 写入 MBR：调用 43h，DS:SI 指向 DAP，扇区=0，缓冲区=0000:7C00（需谨慎，易破坏分区表）。
- 错误重试：若失败，重置磁盘（00h）后重试 3 次。
- 兼容性：优先用 42h/43h/48h/49h，避免用 41h（旧磁盘不支持）。

1.6 内存管理中断（15h）

内存管理中断（15h）是 BIOS 提供的内存管理接口，支持内存检测、扩展内存（>1MB）、虚拟内存（286 保护模式）、设备等待（多任务）。

1.6.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
88h	获取扩展内存大小（286 及以前）		AX=扩展内存大小（KB，最大 65536KB）
87h	移动扩展内存块（286 及以前）	ES:SI=GDT 地址；CX=字节数；BX=源句柄；DX=目标句柄	CF=0 成功，CF=1 失败
89h	进入保护模式（286 及以前）	ES:SI=GDT 地址；BX=IDT 地址	CF=0 成功，CF=1 失败
86h	设置等待时间（多任务）	CX:DX=微秒数	CF=0 成功，CF=1 失败
83h	设置/清除等待标志（多任务）	AL=00h 设置，01h 清除；ES:BX=标志地址	CF=0 成功，CF=1 失败
C0h	获取系统配置		ES:BX=配置表地址；CF=0 成功，CF=1 失败
C1h	获取扩展 BIOS 数据区地址		ES:BX=EBDA 地址；CF=0 成功，CF=1 失败

1.6.2 关键参数说明

- 扩展内存：
  - 大小：AX（最大 65536KB）。
  - 限制：286 及以前最大 16MB，386 及以后无限制。
- 等待时间：
  - 单位：微秒（1/1000000 秒）。
  - 精度：取决于硬件（通常 1ms）。
- 标志：
  - 0：设置等待。
  - 1：清除等待。

1.6.3 实用技巧

- 检测内存：调用 88h，若 AX=0 则无扩展内存。
- 移动内存：调用 87h，ES:SI 指向 GDT，CX=字节数，BX=源句柄，DX=目标句柄。
- 等待：调用 86h，CX:DX=微秒数（如 1000000=1 秒）。
- 兼容性：优先用 88h/86h/C0h/C1h，避免用 87h/89h（旧系统不支持）。

1.7 键盘扩展中断（16h 扩展）

键盘扩展中断（16h 扩展）是 BIOS 提供的增强键盘接口，支持扩展按键、状态查询、缓冲区管理。

1.7.1 核心功能（AH 寄存器值）

AH	功能	参数	返回值
10h	读扩展按键（阻塞）		AL=ASCII 码；AH=扫描码
11h	检测扩展按键（非阻塞）		ZF=0 有按键；ZF=1 无按键；AL=ASCII 码；AH=扫描码
12h	获取扩展键盘状态		AL=状态标志（bit0=右 Shift，bit1=左 Shift，bit2=Ctrl，bit3=Alt，bit4=Scroll Lock，bit5=Num Lock，bit6=Caps Lock，bit7=Insert，bit8=左 Ctrl，bit9=左 Alt，bit10=右 Ctrl，bit11=右 Alt）

1.7.2 关键参数说明

- 扩展按键：
  - 包括：F11、F12、右 Ctrl、右 Alt、方向键、小键盘。
  - 兼容性：旧键盘不支持。
- 状态标志：
  - bit8：左 Ctrl。
  - bit9：左 Alt。
  - bit10：右 Ctrl。
  - bit11：右 Alt。

1.7.3 实用技巧

- 检测扩展键：调用 11h，若 ZF=0 则有扩展键。
- 读取扩展键：调用 10h，AL=ASCII 码，AH=扫描码。
- 检测 Ctrl+Alt+Del：调用 12h，检查 AL 的 bit2（Ctrl）和 bit3（Alt）。
- 兼容性：优先用 10h/11h/12h，避免用旧版 16h。

1.8 其他常用中断

1.8.1 时钟中断（1Ah）

- 功能：读取/设置实时时钟（RTC）。
- 关键：AH=00h 读系统时钟（CX:DX=自午夜以来的时钟滴答数）；AH=01h 设置系统时钟。

1.8.2 打印机中断（17h）

- 功能：控制并行打印机。
- 关键：AH=00h 打印字符（AL=字符，DX=打印机端口）；AH=01h 初始化打印机；AH=02h 检查打印机状态。

1.8.3 串口中断（14h）

- 功能：控制串行通信（COM 口）。
- 关键：AH=00h 初始化串口；AH=01h 发送字符；AH=02h 接收字符；AH=03h 检查状态。

1.8.4 系统服务中断（15h）

- 功能：提供扩展系统服务（如 APM、VESA）。
- 关键：AH=53h APM 安装检查；AH=4Fh VESA 检测；AH=86h 等待（微秒）。

2 实模式下的内存管理

实模式（Real Mode）是 x86 架构上电后的默认运行模式，特点是：

- 直接访问物理内存（无虚拟内存）
- 段寄存器（CS、DS、ES、SS）与偏移量（IP、SP、BP）共同生成 20 位物理地址
- 无内存保护（任何程序可修改任何内存）

2.1 内存分段

2.1.1 段寄存器

寄存器	用途	默认偏移量
CS	代码段	IP
DS	数据段	SI、DI、BX
ES	附加段	DI（字符串操作）
SS	堆栈段	SP、BP

2.1.2 地址计算

物理地址 = 段寄存器 × 16 + 偏移量

例：CS=0000h，IP=7C00h → 物理地址 = 0000h × 16 + 7C00h = 07C00h

2.1.3 段的特性

- 大小：最大 64KB（偏移量 0000h–FFFFh）
- 重叠：多个段可指向同一内存区域
- 对齐：无强制要求，但建议 16 字节对齐（性能优化）

2.1.4 常用段布局

```
; 典型引导扇区布局
CS = 0000h
DS = 0000h
ES = 0000h
SS = 0000h
SP = 7C00h（栈向下生长，不覆盖代码）
```

2.2 内存布局

2.2.1 典型布局

```
0000:0000 ┌─────────────────┐
          │ 中断向量表 (IVT) │ 1024 字节
          │ 256 × 4 字节     │
0000:0400 ├─────────────────┤
          │ BIOS 数据区 (BDA)│ 256 字节
          │ 设备状态、配置   │
0000:0500 ├─────────────────┤
          │ 可用内存         │
          │ 栈 (向下生长)    │
          │ 堆 (向上生长)    │
0000:7C00 ├─────────────────┤
          │ 引导扇区 (512B)  │
0000:7E00 ├─────────────────┤
          │ 可用内存         │
0000:9FFF └─────────────────┘
```

2.2.2 关键区域

- 中断向量表（IVT）：0000:0000–0000:0400，256 个中断向量，每个 4 字节（段:偏移）。
- BIOS 数据区（BDA）：0000:0400–0000:0500，设备状态、配置、计时器。
- 引导扇区：0000:7C00–0000:7E00，512 字节，BIOS 加载的启动代码。
- 栈：SS:SP 向下生长，不覆盖代码。
- 堆：向上生长，动态分配。

2.2.3 安全区域

- 可覆盖：IVT、BDA、栈、堆、引导扇区（加载后）。
- 不可覆盖：BIOS 代码、显存、ROM。

2.3 内存分配

2.3.1 静态分配

```
; 固定地址
buffer: dw 0x7E00
stack_top: dw 0x9000
```

2.3.2 动态分配

```
; 简单堆管理
heap_start: dw 0x0500
heap_ptr: dw 0x0500
malloc:
    ax = [heap_ptr]
    [heap_ptr] += bx
    ret
```

2.3.3 内存池

```
; 固定大小内存池
pool_start: db 512 dup (0)
pool_bitmap: db 64 dup (0)
alloc_block:
    ; 查找空闲位
    ; 设置位图
    ; 返回地址
free_block:
    ; 清除位图
```

2.4 内存保护

实模式无内存保护，任何程序可修改任何内存。

2.4.1 风险

- 栈溢出：栈覆盖代码区。
- 缓冲区溢出：写入超出数组边界。
- 指针错误：指向错误内存。
- 段错误：段寄存器设置错误。

2.4.2 检查

```
; 栈溢出检测
check_stack:
    ax = sp
    cmp ax, 0x0500
    jb stack_overflow
    ret

; 缓冲区溢出检测
check_buffer:
    ax = [buffer_end]
    cmp ax, [buffer_ptr]
    jb buffer_overflow
    ret

; 指针有效性检测
check_pointer:
    ax = [pointer]
    cmp ax, 0x0500
    jb invalid_pointer
    cmp ax, 0x9000
    ja invalid_pointer
    ret
```

2.4.3 防护

- 栈溢出：栈与代码分开，栈顶留安全区。
- 缓冲区溢出：检查长度，使用固定长度。
- 指针错误：检查指针有效性。
- 段错误：检查段寄存器。

2.5 内存管理

2.5.1 简单管理

```
; 固定分配
buffer: dw 0x7E00
stack_top: dw 0x9000

; 简单堆
heap_start: dw 0x0500
heap_ptr: dw 0x0500
malloc:
    ax = [heap_ptr]
    [heap_ptr] += bx
    ret
```

2.5.2 复杂管理

```
; 内存池
pool_start: db 512 dup (0)
pool_bitmap: db 64 dup (0)
alloc_block:
    ; 查找空闲位
    ; 设置位图
    ; 返回地址
free_block:
    ; 清除位图
```

2.5.3 最佳实践

- 简单：固定分配、栈顶留安全区。
- 复杂：内存池、位图管理。
- 安全：检查指针、检查段寄存器。

3 声卡

3.1 声卡基本概念

声卡是计算机的音频处理单元，负责采样、合成、混音、输出。

3.1.1 核心功能

- 采样（Sampling）：将模拟信号转换为数字信号（如 44.1kHz 表示每秒采样 44100 次）。
- 合成（Synthesis）：生成音频信号（如 FM 合成、波表合成）。
- 混音（Mixing）：将多个音频流合并为一个输出。
- 输出（Output）：将数字信号转换为模拟信号（通过 DAC）。

3.1.2 关键参数

- 采样率（Sample Rate）：每秒采样次数（如 44.1kHz、48kHz、96kHz）。
- 位深（Bit Depth）：每个采样的精度（如 16 位、24 位、32 位）。
- 通道（Channels）：单声道（1）、立体声（2）、环绕声（5.1、7.1）。
- 延迟（Latency）：从输入到输出的时间（通常 10–100ms）。
- 缓冲区（Buffer）：存储音频数据的内存区域（通常 512–4096 字节）。

3.1.3 常见声卡

- 集成声卡：主板上的声卡（如 Realtek ALC 系列）。
- 独立声卡：PCI/PCIe 接口的声卡（如 Creative Sound Blaster 系列）。
- USB 声卡：通过 USB 接口的声卡（如 Focusrite 系列）。
- 专业声卡：高端声卡（如 RME、Apogee）。

3.2 声卡工作原理

3.2.1 采样（Sampling）

- 模拟信号：连续的电压变化。
- 采样：每隔一段时间测量电压值。
- 量化：将电压值转换为数字值（如 16 位整数）。
- 编码：将数字值存储为二进制。

3.2.2 合成（Synthesis）

- 波表合成（Wavetable）：预录制的音色（如 GM 音色）。
- FM 合成（Frequency Modulation）：通过调频生成音色（如 Yamaha OPL 系列）。
- 物理建模（Physical Modeling）：模拟乐器物理特性（如弦、管、膜）。

3.2.3 混音（Mixing）

- 多通道输入：将多个音频流合并为一个输出。
- 音量控制：调整每个通道的音量（0–255）。
- 平衡控制：调整左右声道的音量（左 255，右 0 → 左 0，右 255）。
- 效果处理：添加回声、混响、均衡器等效果。

3.2.4 输出（Output）

- 数字信号：通过 I²S、SPDIF 等接口输出。
- 模拟信号：通过 DAC（数模转换器）转换为模拟信号。
- 放大：通过放大器驱动扬声器或耳机。

3.3 声卡编程

3.3.1 检测声卡

```
; 检测 PCI 设备
check_pci:
    ax = 0x0000
    bx = 0x0000
    cx = 0x0000
    dx = 0x0000
    int 0x1A
    ; 查找 Class = 0x04（多媒体设备）
    ; 查找 Subclass = 0x01（音频设备）
    ret

; 检测 ISA 设备
check_isa:
    ; 尝试读取 DSP 版本
    ; 若成功，则存在 Sound Blaster
    ret
```

3.3.2 初始化

```
; 初始化 Sound Blaster
init_sb:
    ; 重置 DSP
    ; 设置采样率
    ; 设置模式
    ; 启用输出
    ret

; 初始化 HDA
init_hda:
    ; 重置控制器
    ; 设置 CORB/RIRB
    ; 设置流
    ; 启用输出
    ret
```

3.3.3 播放音频

```
; 播放 PCM 数据
play_pcm:
    ; 设置缓冲区
    ; 设置采样率
    ; 设置模式
    ; 启用输出
    ; 等待完成
    ret

; 播放 MIDI 数据
play_midi:
    ; 发送 MIDI 消息
    ; 等待完成
    ret
```

3.3.4 混音

```
; 设置音量
set_volume:
    ; 设置主音量
    ; 设置 PCM 音量
    ; 设置 FM 音量
    ret

; 设置平衡
set_balance:
    ; 设置左声道音量
    ; 设置右声道音量
    ret
```

3.4 声卡性能

3.4.1 延迟

- 硬件延迟：声卡处理时间（通常 1–10ms）。
- 软件延迟：驱动程序处理时间（通常 10–100ms）。
- 总延迟：硬件延迟 + 软件延迟（通常 10–100ms）。
- 优化：使用专业声卡、减小缓冲区、优化驱动程序。

3.4.2 吞吐量

- 采样率：44.1kHz（CD 质量）、48kHz（DVD 质量）、96kHz（专业）。
- 位深：16 位（CD 质量）、24 位（专业）、32 位（浮点）。
- 通道：2（立体声）、5.1（环绕声）、7.1（环绕声）。
- 计算：采样率 × 位深 × 通道 = 数据率（如 44.1kHz × 16 位 × 2 = 1.411Mbit/s）。

3.4.3 优化

- 减小缓冲区：减小缓冲区大小（如 512 字节）。
- 优化驱动：使用专业驱动（如 ASIO）。
- 硬件加速：使用支持硬件加速的声卡。
- 实时内核：使用实时内核（如 Linux RT）。

3.5 声卡兼容性

3.5.1 Sound Blaster 系列

- Sound Blaster 16：ISA 接口，16 位，44.1kHz，FM 合成。
- Sound Blaster Pro：ISA 接口，8 位，22kHz，FM 合成。
- Sound Blaster 1.0：ISA 接口，8 位，11kHz，FM 合成。
- 检测：读取 DSP 版本（命令 0xE1）。

3.5.2 HDA（High Definition Audio）

- 现代标准：PCI 接口，32 位，192kHz，多通道。
- 兼容性：大多数现代主板。
- 检测：查找 PCI Class = 0x04，Subclass = 0x03。

3.5.3 USB 音频

- 即插即用：无需驱动。
- 兼容性：大多数现代设备。
- 限制：延迟较高（10–100ms）。

3.6 声卡调试

3.6.1 检测问题

- 无声音：检查连接、检查静音、检查驱动。
- 杂音：检查接地、检查屏蔽、检查电源。
- 延迟：减小缓冲区、优化驱动。
- 断续：增大缓冲区、优化驱动。

3.6.2 工具

- 测试：播放测试音频、录制测试音频。
- 监控：监控 CPU 占用、监控内存占用。
- 日志：记录驱动日志、记录错误。

4 显卡

4.1 显卡基本概念

显卡是计算机的图形处理单元，负责渲染、显示、加速。

4.1.1 核心功能

- 渲染（Rendering）：生成图像（如 3D 场景、UI 界面）。
- 显示（Display）：将图像输出到显示器（如 VGA、HDMI、DP）。
- 加速（Acceleration）：硬件加速图形计算（如 3D 变换、光照、纹理）。
- 视频（Video）：视频编码/解码（如 H.264、H.265）。

4.1.2 关键参数

- 分辨率（Resolution）：屏幕像素数（如 1920×1080、3840×2160）。
- 色深（Color Depth）：每个像素的颜色数（如 8 位、16 位、24 位、32 位）。
- 刷新率（Refresh Rate）：每秒刷新次数（如 60Hz、144Hz、240Hz）。
- 显存（VRAM）：显卡内存（如 1GB、2GB、4GB）。
- 带宽（Bandwidth）：数据传输速度（如 100GB/s、500GB/s）。

4.1.3 常见显卡

- 集成显卡：主板上的显卡（如 Intel HD Graphics、AMD Radeon Vega）。
- 独立显卡：PCI/PCIe 接口的显卡（如 NVIDIA GeForce、AMD Radeon）。
- 专业显卡：专业图形卡（如 NVIDIA Quadro、AMD Radeon Pro）。
- 服务器显卡：服务器图形卡（如 NVIDIA Tesla、AMD Instinct）。

4.2 显卡工作原理

4.2.1 渲染流程

- 顶点处理：处理顶点数据（如变换、光照）。
- 光栅化：将顶点转换为像素（如三角形光栅化）。
- 片元处理：处理像素（如纹理、光照、阴影）。
- 输出合并：合并多个渲染结果（如 Z 缓冲、模板缓冲）。

4.2.2 显存（VRAM）

- 帧缓冲（Frame Buffer）：存储渲染结果（如 1920×1080×4 字节）。
- 纹理（Texture）：存储纹理数据（如 2048×2048×4 字节）。
- 顶点缓冲（Vertex Buffer）：存储顶点数据（如 100MB）。
- 索引缓冲（Index Buffer）：存储索引数据（如 10MB）。

4.2.3 显示输出

- 数字信号：通过 HDMI、DP、DVI 输出。
- 模拟信号：通过 VGA、DVI-I 输出。
- 刷新率：决定显示更新频率（如 60Hz 表示每秒更新 60 次）。
- 垂直同步（VSync）：同步刷新与渲染，防止撕裂。

4.3 显卡编程

4.3.1 检测显卡

```
; 检测 PCI 设备
check_pci:
    ax = 0x0000
    bx = 0x0000
    cx = 0x0000
    dx = 0x0000
    int 0x1A
    ; 查找 Class = 0x03（显示设备）
    ; 查找 Subclass = 0x00（VGA 兼容）
    ; 查找 Subclass = 0x02（3D 控制器）
    ret

; 检测 VBE（VESA BIOS Extensions）
check_vbe:
    ax = 0x4F00
    int 0x10
    ; 若 ax = 0x004F，则支持 VBE
    ret
```

4.3.2 初始化

```
; 初始化 VGA
init_vga:
    ; 设置模式 13h（320×200 256 色）
    ; 设置调色板
    ; 启用输出
    ret

; 初始化 VBE
init_vbe:
    ; 设置模式（如 1920×1080×32）
    ; 设置调色板
    ; 启用输出
    ret

; 初始化 GPU
init_gpu:
    ; 重置控制器
    ; 设置显存
    ; 设置流
    ; 启用输出
    ret
```

4.3.3 绘制

```
; 绘制像素
draw_pixel:
    ; 计算偏移
    ; 写入显存
    ret

; 绘制线
draw_line:
    ; 计算坐标
    ; 绘制像素
    ret

; 绘制矩形
draw_rect:
    ; 计算坐标
    ; 绘制线
    ret

; 绘制文本
draw_text:
    ; 查找字符
    ; 绘制字形
    ret
```

4.3.4 显存管理

```
; 显存分配
alloc_vram:
    ; 计算大小
    ; 分配内存
    ret

; 显存释放
free_vram:
    ; 释放内存
    ret

; 显存拷贝
copy_vram:
    ; 计算源
    ; 计算目标
    ; 拷贝数据
    ret
```

4.4 显卡性能

4.4.1 渲染

- 顶点处理：每个顶点 10–100 指令。
- 光栅化：每个三角形 100–1000 指令。
- 片元处理：每个像素 10–100 指令。
- 输出合并：每个像素 1–10 指令。

4.4.2 显存

- 帧缓冲：1920×1080×4 = 8MB。
- 纹理：2048×2048×4 = 16MB。
- 顶点缓冲：100MB。
- 索引缓冲：10MB。

4.4.3 带宽

- 显存带宽：100GB/s（典型）。
- 计算：显存频率 × 位宽 / 8（如 1GHz × 256 位 / 8 = 32GB/s）。
- 瓶颈：带宽不足会限制性能。

4.4.4 优化

- 减少绘制：减少绘制调用。
- 减少纹理：减少纹理大小。
- 减少顶点：减少顶点数。
- 减少片元：减少片元数。
- 使用缓存：使用顶点缓冲、索引缓冲。

4.5 显卡兼容性

4.5.1 VGA

- 模式 13h：320×200 256 色。
- 模式 12h：640×480 16 色。
- 模式 03h：80×25 文本。
- 检测：查询 VBE 信息。

4.5.2 VBE（VESA BIOS Extensions）

- 现代标准：PCI 接口，32 位，192kHz，多通道。
- 兼容性：大多数现代主板。
- 检测：查找 PCI Class = 0x03，Subclass = 0x00。

4.5.3 GPU

- 即插即用：无需驱动。
- 兼容性：大多数现代设备。
- 限制：延迟较高（10–100ms）。

4.6 显卡调试

4.6.1 检测问题

- 无显示：检查连接、检查驱动、检查显存。
- 花屏：检查显存、检查驱动、检查时序。
- 卡顿：减少绘制、减少纹理、减少顶点、减少片元、优化驱动。
- 崩溃：检查驱动、检查内存、检查温度。

4.6.2 工具

- 测试：绘制测试图形、测试性能。
- 监控：监控 CPU 占用、监控内存占用、监控温度。
- 日志：记录驱动日志、记录错误。

关定义的各种RTL指令函数. 选择其中一个函数, 分别去掉 `static`, 去掉 `inline` 或去掉两者, 然后重新进行编译, 你可能会看到发生错误. 请分别解释为什么这些错误会发生/不发生? 你有办法证明你的想法吗?

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

温馨提示

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

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

PA3 - 穿越时空的旅程: 批处理系统

世界诞生的故事 - 第三章

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

! 代码管理

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

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

! 提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 40小时

阶段性安排:

- task PA3.1: 实现自陷操作 `yield()`
- task PA3.2: 实现用户程序的加载和系统调用, 支撑TRM程序的运行
- task PA3.3: 运行仙剑奇侠传并展示批处理系统, 提交完整的实验报告

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

↑2 没有调不出的bug，只有不理解的系统

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

这意味着，bug的传播链条会变得更复杂，调试也越来越考验你对系统行为的理解。如果你之前调bug都是拍大腿，并没有在调试过程中去理解系统的细节，你会发现在PA3中自己和大腿的差距正在迅速扩大：大腿对整个系统的掌握越来越熟练，而你则是越往前就越觉得寸步难行；例如讲义和代码越来越看不懂，bug越来越神秘。

亡羊补牢的方法只有一个：真的不能再偷懒了。

批处理系统

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

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

后来人们就想，每次都要管理员来手动加载新的程序，这太麻烦了，回想起冯诺依曼计算机的工作方式，它的其中一个特点就是：当计算机执行完一条指令的时候，它就会自动地去执行下一条指令。类似的，我们能不能让管理员事先准备好一组程序，让计算机执行完一个程序之后，就自动执行下一个程序呢？

这就是批处理系统的意思，有了批处理系统之后，就可以解放管理员的双手了，而批处理系统的关键，就是要有一个后台程序，当一个前台程序执行结束的时候，后台程序就会自动加载一个新的前台程序来执行。

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

❓ 什么是操作系统？(建议二周回思考)

这可是个大问题，我们也鼓励你学习完操作系统课程之后再回来重新审视它。

最简单的操作系统

眼见为实，我们先来展示一下操作系统究竟可以多简单。在PA中使用的操作系统叫Nanos-lite，它是南京大学操作系统Nanos的裁剪版，是一个为PA量身订造的操作系统，通过编写Nanos-lite的代码，你将会认识到操作系统是如何使用硬件提供的机制(也就是ISA和AM)，来实现程序的运行，这也符合PA的终极目标：至于完整版的Nanos，你将会在下学期的操作系统课程上见识到它的庐山真面目。

我们为大家准备了Nanos-lite的程序代码，通过执行以下命令获取：

```
cd ics2021
bash init.sh nanos-lite
```

Nanos-lite包含了后续PA用到的所有模块，但大部分模块的具体功能都没有实现，由于硬件(NEMU)的功能是逐渐添加的，Nanos-lite也要配合这个过程，你会通过 `nanos-lite/include/common.h` 中一些与实验进度相关的宏来控制Nanos-lite的功能，随着实验进度的推进，我们会逐渐讲解所有的模块，Nanos-lite做的工作也会越来越多，因此在阅读Nanos-lite的代码时，你只需要关心和当前进度相关的模块就可以了，不要纠缠于和当前进度无关的代码。

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

需要提醒的是，Nanos-lite是运行在AM之上，AM的API在Nanos-lite中都是可用的，虽然操作系统对我们来说是一个特殊的概念，但在AM看来，它只是一个调用AM API的普通C程序而已，和超级玛丽没什么区别。同时，你会再次体会到AM的好处：Nanos-lite的实现可以是架构无关的，这意味着，无论你之前选择的是哪一款ISA，都可以很容易地运行Nanos-lite，甚至你可以像开发klib那样，在 `native` 上调试你编写的Nanos-lite。

另外，虽然不会引起明显的误解，但在引入Nanos-lite之后，我们还是会在某些地方使用“用户进程”的概念，而不是“用户程序”，如果你现在不能理解什么是进程，你只需要把进程作为“正在运行的程序”来理解就可以了，还感觉不出这两者的区别？单一个简单的例子吧，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，但是盘中的记事本程序只有一个，进程是操作系统中一个重要的概念，有关进程的详细知识会在操作系统课上进行介绍。

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

1. 打印Project-N的logo，并通过 `Log()` 输出hello信息和编译时间，需要说明的是，Nanos-lite中定义的 `Log()` 宏并不是NEMU中定义的 `Log()` 宏，Nanos-lite和NEMU是两个独立的项目，它们的代码不会相互影响，你在阅读代码的时候需要注意这一点，在Nanos-lite中，`Log()` 宏通过你在 `klib` 中编写的 `printf()` 输出，最终会调用TRM的 `putch()`。
2. 调用 `init_device()` 对设备进行一些初始化操作，目前 `init_device()` 会直接调用 `ioe_init()`。
3. 初始化ramdisk，一般来说，程序应该存放在永久存储的介质中(比如磁盘)，但要在NEMU中对磁盘进行模拟是一个略显复杂工作，因此先让Nanos-lite把其中的一段内存作为磁盘来使用，这样的磁盘有一个专门的名字，叫ramdisk。
4. `init_fs()` 和 `init_proc()`，分别用于初始化文件系统和创建进程，目前它们均未进行有意义的操作，可以忽略它们。
5. 调用 `panic()` 结束Nanos-lite的运行。

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

```
make ARCH=$ISA-nemu run
```

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

💬 操作系统是个C程序

你也许有点不敢相信，但是现代码的.c和.h文件无疑蕴含着这铁一般的事实，甚至连编译方式也看不出什么特殊的地方. GNU/Linux也是这样: 如果你阅读它的源代码, 就会发现GNU/Linux只不过是个巨大的C程序而已.

那么和一般的C程序相比, 操作系统究竟有何特殊之处呢? 完成PA3之后, 相信你就会略知一二了.

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

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

来自操作系统的新需求

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

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

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

等级森严的制度

为了阻止程序将执行流切换到操作系统的任意位置, 硬件中逐渐出现保护机制相关的功能, 比如386中引入了保护模式(protected mode)和特权级(privilege level)的概念, 而mips32处理器可以运行在内核模式和用户模式, riscv32则有机器模式(M-mode), 监控者模式(S-mode)和用户模式(U-mode). 这些概念的思想都是类似的: 简单地说, 只有高特权级的程序才能去执行一些系统级别的操作, 如果一个特权级低的程序尝试执行它没有权限执行的操作, CPU将会抛出一个异常信号, 来阻止这一非法行为的发生. 一般来说, 最适合担任系统管理员的角色就是操作系统了, 它拥有最高的特权级, 可以执行所有操作; 而除非经过允许, 运行在操作系统上的用户程序一般都处于最低的特权级, 如果它试图破坏社会的和谐, 它将会被判“死刑”.

以支持现代操作系统的RISC-V处理器为例, 它们存在M, S, U三个特权模式, 分别代表机器模式, 监管者模式和用户模式. M模式特权级最高, U模式特权级最低, 低特权级能访问的资源, 高特权级也能访问. 即CPU是怎么判断一个进程是否执行了无权限操作呢? 答案很简单, 只要在硬件上维护一个用于标识当前特权模式的寄存器(属于计算机状态的一部分), 然后在访问那些高特权级才能访问的资源时, 对当前特权模式进行检查, 例如RISC-V中有一条特权指令 `sfence.vma`，手册要求只有当处理器当前的特权模式不低于S模式才能执行, 因此我们可以在硬件上添加一些简单的逻辑来实现特权模式的检查:

```
is_sfence_vma_ok = (priv_mode == M_MODE) || (priv_mode == S_MODE);
```

可以看到, 特权模式的检查只不过时一些门电路而已. 如果检查不通过, 此次操作将会被判定为非法操作, CPU将会抛出异常信号, 并跳转到一个和操作系统约定好的内存位置, 交由操作系统进行后续处理.

通常来说, 操作系统运行在S模式, 因此有权限访问所有的代码和数据; 而一般的程序运行在U模式, 这就决定了它只能访问U模式的代码和数据. 这样, 只要操作系统将其私有代码和数据放S模式中, 恶意程序就永远没有办法访问到它们.

类似地, x86的操作系统运行在ring 0, 用户进程运行在ring 3; mips32的操作系统运行在内核模式, 用户进程运行在用户模式. 这些保护相关的概念和检查过程都是通过硬件来实现的, 只要软件运行在硬件上面, 都无法逃出这一天网. 硬件保护机制使得恶意程序永远无法全身而退, 为构建计算机和谐社会作出了巨大的贡献.

这是多美的功能呢？这意味着，上面提到的很多概念其实只是一种约定，真正的保护机制也还需要考虑更多的细节. ISA手册中一般都会专门有一章来描述保护机制, 这就已经看出来保护机制并不是简单说说而已. 根据KISS法则, 我们并不打算在NEMU中加入保护机制. 我们运行所有用户进程都运行在最高特权级, 虽然所有用户进程都有权限执行所有指令, 不过由于PA中的用户程序都是我们自己编写的, 一切还是在我们的控制范围之内. 毕竟, 我们也已经从上面的故事中体会到保护机制的本质了: 在硬件中加入一些与特权级检查相关的门电路(例如比较器电路), 如果发现了非法操作, 就会抛出一个异常信号, 让CPU跳转到一个约定好的目标位置, 并进行后续处理.

💬 分量离析的秩序

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

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

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

穿越时空的旅程
mips32
riscv32
ARM64
x86_64
TRM
CTE

穿越时空的旅程

每个操作系统中都会用到，用户程序所依赖的底层硬件抽象层的设计是操作系统设计的一个关键决策。本章将通过对比 mips32、riscv32、ARM64 和 x86_64 这四种主流架构的实现，以两个经典指令集模拟器 NEMU 和 Spike 为参考，来探究这一抽象层的设计原则。这个抽象层的目标是为上层提供一个统一、简洁、可移植的接口，屏蔽掉不同硬件的细节。

这一过程最终会抽象出一个接口，我们将称之为 **TRM**（Transfer Register Machine）。这个接口的核心思想是：将整个计算机系统视为一个简单的寄存器机，通过对寄存器的读写来完成计算。

ARM64

ARM64 的 .bin 文件中包含的是一个纯粹的、没有任何元数据的机器代码（Machine Code）。这个文件内容，用户程序和内核都是以完全相同的方式被加载到内存中的。这个过程是一个由操作系统（或引导程序）来执行的简单文件拷贝。

这个过程发生在内存地址 0x40000000 处。这个地址是一个由操作系统（或引导程序）分配给用户程序的虚拟内存地址。

内存布局

一旦文件被加载到内存中，操作系统会将控制权交给程序的入口点。对于 ARM64，用户程序期望内存布局是一个连续的、从 0x40000000 开始的、可执行、可读、可写的区域。为了满足这种期望，操作系统会为一个进程分配一个连续的、足够大的内存块，并将文件的内容原封不动地放到这个内存块的开头。

入口点

这个内存区域的开头，通常是一个指令。这个指令的地址，就是程序的入口点（Entry Point）。对于 ARM64，用户程序期望的第一条指令是 _start 函数的第一条指令。

栈

程序在运行时，需要一个栈来存储函数调用的信息、局部变量、以及保存寄存器的内容。这个栈的内容，是由程序自己在运行时，通过对 sp 寄存器的操作，来动态地分配和释放的。

堆

程序在运行时，通过 malloc 这样的函数来动态地分配内存。这个内容，是由程序自己在运行时，通过对 brk 或 mmap 这样的系统调用，来动态地分配和释放的。

寄存器

程序在运行时，会使用一些通用寄存器来存储数据。这个内容，是由程序自己在运行时，通过对寄存器的读写，来动态地分配和释放的。

指令

程序的指令，是以一个连续的、没有任何间隔的方式，紧密地排列在内存中的。这个内容，是由编译器在编译时，通过对指令的生成，来静态地分配和释放的。

数据

程序的数据，是以一个连续的、没有任何间隔的方式，紧密地排列在内存中的。这个内容，是由编译器在编译时，通过对数据的生成，来静态地分配和释放的。

堆栈

堆栈，是指向堆和栈的指针。这个内容，是由操作系统在程序启动时，通过对 sp 寄存器的初始化，来动态地分配和释放的。

内核

内核，是操作系统的核心。这个内容，是由操作系统在启动时，通过对内核代码的加载，来动态地分配和释放的。

设置栈顶

在将控制权交给程序之前，操作系统会将一个指针，指向栈的顶部，保存在 sp 寄存器中。这个指针的值，通常是栈的起始地址减去一个足够大的偏移量。这个偏移量，是由编译器在编译时，通过对栈帧的大小的计算，来静态地决定的。

跳转到入口点

最后，操作系统会将控制权，通过一个跳转指令，交给程序的入口点。这个跳转指令的目标地址，就是程序的入口点。

程序开始执行

程序的第一条指令，通常是一个对栈的操作，比如 sub sp, sp, #16。这个指令的作用，是为程序在栈上分配一个大小为 16 字节的内存空间。这个空间，通常是用来存放一些局部变量的。

函数调用

当程序调用一个函数时，比如 bl _main，这个指令的作用，是将当前的程序计数器（PC）的值，保存到一个特殊的寄存器 lr（Link Register）中，然后将 PC 的值，设置为 _main 函数的入口地址。

参数传递

函数的参数，是通过寄存器来传递的。这个过程，是由编译器在编译时，通过对参数的分析，来静态地决定的。

返回值

函数的返回值，是通过寄存器来传递的。这个过程，是由编译器在编译时，通过对返回值的分析，来静态地决定的。

函数返回

当一个函数执行完毕时，通常是一个 ret 指令。这个指令的作用，是将 lr 寄存器的值，重新赋值给 PC，从而实现程序的跳转。

程序结束

当程序的 _start 函数执行完毕时，它会调用 exit 系统调用，来通知操作系统，程序已经结束。这个系统调用的参数，通常是一个表示程序退出状态的整数。

内存释放

操作系统会回收程序所占用的内存，并将其标记为可用，以便后续的程序使用。

程序消亡

程序的生命，就此结束。

指令集

指令集，是一个硬件设计中，最核心的抽象。它定义了一个处理器能够理解和执行的，所有基本操作的集合。

寄存器

寄存器，是指令集的一个重要组成部分。它们是一个处理器，能够直接访问的，最快的存储单元。

内存模型

内存模型，是指一个处理器，如何看待和组织它的内存的一个抽象。

指令编码

指令编码，是指一个指令，在内存中，是如何被表示成一个二进制数据的一个规则。

寻址模式

寻址模式，是指一个指令，如何指定它要操作的数据的一个方式。

特权级

特权级，是指一个处理器，为了实现操作系统的安全性，而设计的一个安全机制。

中断

中断，是一个硬件设计中，最重要的一个概念。它允许一个硬件设备，通知处理器，有一个需要处理的事件发生了。

异常

异常，是一个软件设计中，与中断类似的一个概念。它通常是由一个正在执行的程序，自己触发的，用来通知操作系统，发生了一个错误或一个特殊情况。

系统调用

系统调用，是一个操作系统，为其上的应用程序，提供的一个接口。应用程序，通过这个接口，来请求操作系统，为它执行一个特权操作。

内存管理单元（MMU）

内存管理单元（MMU），是一个硬件设计中，最重要的一个组件。它负责将虚拟内存地址，翻译成物理内存地址。

页表

页表，是内存管理单元（MMU），为了实现虚拟内存，而使用的一个数据结构。它存储了虚拟内存页，到物理内存页的映射关系。

缺页异常

缺页异常，是一个硬件设计中，与页表紧密相关的一个异常。当一个程序，试图访问一个，在页表中，没有映射的虚拟内存地址时，就会发生一个缺页异常。

上下文切换

上下文切换，是一个操作系统，为了实现多任务，而设计的一个核心机制。它允许操作系统，暂停一个正在运行的进程，保存它的状态，然后恢复另一个，之前被暂停的进程的状态，让它继续运行。

进程

进程，是一个操作系统，为了实现多任务，而设计的一个核心抽象。它代表了一个，正在运行的程序的实例。

线程

线程，是一个操作系统，为了实现并发，而设计的一个抽象。它是一个，比进程更轻量级的，执行单元。

文件系统

文件系统，是一个操作系统，为了实现，对持久性数据的，管理而设计的一个抽象。它将，硬盘上的，原始数据块，组织成，我们，熟悉的，文件和目录的，结构。

设备驱动

设备驱动，是一个操作系统，为了实现，与硬件设备的，交互，而设计的一个，软件模块。它，是，操作系统，与，硬件，之间的，桥梁。

总线

总线，是一个计算机系统，为了实现，各个，硬件，模块，之间，通信，而，设计，的，一，组，共享，的，通信，通道。

中断控制器

中断控制器，是一个，硬件，设计，它，负责，管理，和，调度，来，自，于，各，个，硬，件，设，备，的，中，断，请，求。

定时器

定时器，是一个，硬，件，设，计，它，能，够，在，一，个，预，设，的，时，间，后，向，处，理，器，发，出，一，个，中，断，信，号。

直接内存访问（DMA）

直接内存访问（DMA），是一个，硬，件，设，计，它，允，许，一，些，硬，件，设，备，在，不，需，要，处，理，器，的，情，况，下，直，接，与，内，存，进，行，数，据，交，换。

指令流水线

指令流水线，是一个，硬，件，设，计，的，一，个，核，心，概，念，它，将，一，个，指，令，的，执，行，过，程，分，解，为，多，个，独，立，的，阶，段，来，实，现，指，令，级，并，行。

超标量

超标量，是一个，硬，件，设，计，的，一，个，高，级，概，念，它，通，过，在，一，个，时，钟，周，期，内，发，射，多，条，指，令，来，进，一，步，提，高，指，令，级，并，行，的，潜，力。

乱序执行

乱序执行，是一个，硬，件，设，计，的，一，个，极，为，复，杂，的，概，念，它，允，许，处，理，器，为，了，提，高，性，能，而，改，变，指，令，的，执，行，顺，序。

分支预测

分支预测，是一个，硬，件，设，计，的，一，个，关，键，技，术，它，试，图，在，一，条，指，令，的，结，果，还，未，确，定，之，前，就，猜，测，其，结，果，并，提，前，执，行，后，续，指，令。

缓存

缓存，是一个，硬，件，设，计，的，一，个，核，心，概，念，它，是，一，个，小，而，快，的，存，储，单，元，用，于，存，放，处，理，器，最，近，或，最，常，用，的，数，据，和，指，令，以，减，少，对，主，内，存，的，访，问，延，迟。

多级缓存

多级缓存，是一个，硬，件，设，计，的，一，个，高，级，概，念，它，通，过，在，处，理，器，和，主，内，存，之，间，设，置，多，个，容，量，和，速，度，不，同，的，缓，存，来，进，一，步，减，少，对，主，内，存，的，访，问，延，迟。

缓存一致性

缓存一致性，是一个，硬，件，设，计，的，一，个，关，键，挑，战，它，确，保，在，多，个，处，理，器，拥，有，各，自，的，缓，存，的，情，况，下，对，同，一，个，内，存，地，址，的，数，据，保，持，一，致，性。

内存一致性

内存一致性，是一个，硬，件，设，计，的，一，个，极，为，复，杂，的，概，念，它，定，义，了，多，个，处，理，器，对，内，存，的，访，问，顺，序，应，该，是，怎，么，样，的，一，个，规，则。

原子操作

原子操作，是一个，硬，件，设，计，的，一，个，关，键，概，念，它，保，证，一，个，操，作，在，执，行，过，程，中，不，会，被，其，他，操，作，所，打，断，或，干，扰。

内存屏障

内存屏障，是一个，硬，件，设，计，的，一，个，关，键，概，念，它，用，于，强，制，处，理，器，按，照，一，个，特，定，的，顺，序，来，执，行，内，存，访，问，指，令，以，保，证，数，据，的，一，致，性，和，可，预，测，性。

指令集架构（ISA）

指令集架构（ISA），是一个，硬，件，设，计，的，一，个，抽，象，层，它，定，义，了，一，个，处，理，器，能，够，理，解，和，执，行，的，所，有，指，令，的，集，合，以，及，这，些，指，令，的，格，式，和，语，义。

微架构

微架构，是一个，硬，件，设，计，的，一，个，具，体，实，现，它，是，指，在，一，个，给，定，的，指，令，集，架，构，（ISA，）下，，如，何，设，计，一，个，处，理，器，的，内，部，结，构，来，执，行，指，令，集，中，的，每，一，条，指，令，。

流水线冒险

流水线冒险，是一个，硬，件，设，计，的，一，个，重，要，问，题，它，指，的，是，在，指，令，流，水，线，中，，由，于，某，些，原，因，，导，致，后，续，指，令，无，法，在，其，预，定，的，时，钟，周，期，内，执，行，的，情，况，。

数据冒险

数据冒险，是一个，流，水，线，冒，险，的，一，种，，它，是，由，于，一，条，指，令，需，要，等，待，另，一，条，指，令，的，执，行，结，果，，才，能，够，继，续，执，行，的，情，况，。

控制冒险

控制冒险，是一个，流，水，线，冒，险，的，一，种，，它，是，由，于，一，条，指，令，的，执，行，结，果，，会，改，变，后，续，指，令，的，执，行，顺，序，，从，而，导，致，流，水，线，需，要，清，空，或，暂，停，的，情，况，。

结构冒险

结构冒险，是一个，流，水，线，冒，险，的，一，种，，它，是，由，于，硬，件，资，源，不，足，，导，致，多，条，指，令，需，要，同，时，使，用，同，一，个，硬，件，资，源，，从，而，导，致，流，水，线，需，要，清，空，或，暂，停，的，情，况，。

数据旁路

数据旁路，是一个，硬，件，设，计，的，一，个，优，化，技，术，它，通，过，在，硬，件，层，面，，将，一，条，指，令，的，结，果，，直，接，从，其，产，生，的，地，方，，传，送，给，后，续，需，要，该，结，果，的，指，令，，，而，无，需，等，待，该，结，果，写，入，到，寄，存，器，，从，而，减，少，了，流，水，线，的，停，顿，。

寄存器重命名

寄存器重命名，是一个，硬，件，设，计，的，一，个，高，级，技，术，它，通，过，为，寄，存，器，分，配，一，个，虚，拟，的，名，称，，来，消，除，指，令，之，间，的，假，性，冒，险，，，从，而，实，现，指，令，的，乱，序，执，行，

代码最后会返回到Nanos-lite触发自陷的代码位置, 然后继续执行. 在它看来, 这次时空之旅就好像没有发生过一样.

恢复上下文

你需要实现这一过程中的新指令. 重新运行Nanos-lite, 如果你的实现正确, 你会看到在 `do_event()` 中输出的信息, 并且最后仍然触发了 `main()` 函数末尾设置的 `panic()` .

必答题(需要在实验报告中回答) - 理解穿越时空的旅程

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

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

mips32延迟槽和异常

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

异常处理的踪迹 - etrace

处理器抛出异常也可以反映程序执行的行为, 因此我们也可以记录异常处理的踪迹(exception trace). 你也许认为在CTE中通过 `printf()` 输出信息也可以达到类似的效果, 但这一方案和在NEMU中实现的etrace还是有如下区别:

- 打开etrace不改变程序的行为(对程序来说是非侵入式的): 你将来可能会遇到一些bug, 当你尝试插入一些 `printf()` 之后, bug的行为就会发生变化. 对于这样的bug, etrace还是可以帮助你进行诊断, 因为它是在NEMU中输出的, 不会改变程序的行为.
- etrace也不受程序行为的影响: 如果程序包含一些致命的bug导致无法进入异常处理函数, 那就无法在CTE中调用 `printf()` 来输出; 在这种情况下, etrace仍然可以正常工作

事实上, QEMU和Spike也实现了类似etrace的功能, 如果在上面运行的系统软件发生错误, 开发者也可以通过这些功能快速地进行bug的定位和诊断.

实现etrace

你已经在NEMU中实现了很多trace工具了, 要实现etrace自然也难不倒你啦.

温馨提示

PA3阶段1到此结束.

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

用户程序和系统调用

每个应用程序，用户程序都可以认为时间和资源独享系统提供的入口了，现在我们来解决实现这些要求的第二个问题：如何加载用户程序。

加载第一个用户程序

在操作系统中，加载用户程序是由操作系统内核来做的，我们在加载程序中应该代码和数据，以及初始化寄存器，设置栈指针，最后跳转到程序入口。这个程序入口通常是程序的开始，但是也有例外情况，例如动态链接的程序，这一部分我们会在以下，更具体的，为了简化 hello-world 程序，我们要解决的问题：

- 代码和文件在哪里？
- 代码和数据应该加载在内存的哪个位置？
- 代码和数据有多大？
- 这个程序的入口在哪？应该跳转到哪？

这个问题，在以前，我们以要求为 hello-world 下载一个程序，现在我们要求自己加载并跳转。这个，我们要求 hello-world 的程序，现在我们要求自己加载并跳转。

```
$ ls -la hello
-rwxrwxr-x 1 ubuntu ubuntu 16816 Jan 12 10:11 hello
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          16816 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         12
  Section header string table index: 11
$ ./hello
Hello, world!
$ strace ./hello
execve("./hello", ["./hello"], 0x7ffc9a7d2a20 /* 25 vars */) = 0
write(1, "Hello, world!\n", 14)         = 14
exit_group(0)                           = ?
+++ exited with 0 +++
$ ls -la /usr/bin/ls
-rwxrwxr-x 1 root root 142144 Jan 12 10:11 /usr/bin/ls
$ file /usr/bin/ls
/usr/bin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, stripped
$ readelf -h /usr/bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x6a00
  Start of program headers:          64 (bytes into file)
  Start of section headers:          142144 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
$ strace /usr/bin/ls
execve("/usr/bin/ls", ["/usr/bin/ls"], 0x7ffc9a7d2a20 /* 25 vars */) = 0
brk(NULL)                               = 0x55a8c4e2a000
arch_prime_mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3a9b2c8000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=89234, st_ino=1048576, st_dev=fd00:2a, st_nlink=1, st_uid=0, st_gid=0, st_blksize=4096, st_blocks=176, st_atime=1704067200, st_atime_nsec=0, st_mtime=1704067200, st_mtime_nsec=0, st_ctime=1704067200, st_ctime_nsec=0}) = 0
mmap(NULL, 89234, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f3a9b2b2000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0", 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, st_ino=1048577, st_dev=fd00:2a, st_nlink=1, st_uid=0, st_gid=0, st_blksize=4096, st_blocks=3968, st_atime=1704067200, st_atime_nsec=0, st_mtime=1704067200, st_mtime_nsec=0, st_ctime=1704067200, st_ctime_nsec=0}) = 0
mmap(NULL, 2029592, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f3a9b0c7000
mmap(0x7f3a9b0ed000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0x26000) = 0x7f3a9b0ed000
mmap(0x7f3a9b265000, 393216, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0x1ae000) = 0x7f3a9b265000
mmap(0x7f3a9b2bd000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x205000) = 0x7f3a9b2bd000
mmap(0x7f3a9b2c3000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f3a9b2c3000
close(3)                                = 0
arch_prime_mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3a9b0c5000
arch_prime_mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3a9b0c3000
arch_prime_set_fs(0x7f3a9b0c5000)       = 0
mprotect(0x7f3a9b2bd000, 16384, PROT_READ)  = 0
mprotect(0x55a8c4e2a000, 4096, PROT_READ)   = 0
mprotect(0x7f3a9b2f8000, 4096, PROT_READ)   = 0
munmap(0x7f3a9b2b2000, 89234)           = 0
brk(NULL)                               = 0x55a8c4e2a000
brk(0x55a8c4e4b000)                     = 0x55a8c4e4b000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3004928, st_ino=1048579, st_dev=fd00:2a, st_nlink=1, st_uid=0, st_gid=0, st_blksize=4096, st_blocks=5872, st_atime=1704067200, st_atime_nsec=0, st_mtime=1704067200, st_mtime_nsec=0, st_ctime=1704067200, st_ctime_nsec=0}) = 0
close(3)                                = 0
write(1, "total 0\n", 8)                = 8
write(1, "drwxrwxr-x 2 ubuntu ubuntu 4096 Jan 12 10:11 .\n", 47) = 47
write(1, "drwxrwxr-x 3 ubuntu ubuntu 4096 Jan 12 10:11 ..\n", 47) = 47
exit_group(0)                           = ?
+++ exited with 0 +++
```

在Nanos-lite上运行Hello world

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

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

堆区管理

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

调整堆区大小是通过 `sbrk()` 库函数来实现的, 它的原型是

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

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

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

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

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

上述代码是在用户层的库函数中实现的, 我们还需要在Nanos-lite中实现 `SYS_brk` 的功能. 由于目前Nanos-lite还是一个单任务操作系统, 空闲的内存都可以让用户程序自由使用, 因此我们只需要让 `SYS_brk` 系统调用总是返回 `0` 即可, 表示堆区大小的调整总是成功. 在PA4中, 我们会对这一系统调用进行修改, 实现真正的内存分配.

实现堆区管理

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

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

如果你的实现正确, 你可以借助strace看到 `printf()` 不再是逐个字符地通过 `write()` 进行输出, 而是将格式化完毕的字符串通过一次性进行输出.

缓冲区与系统调用开销

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

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

printf和换行

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

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

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

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

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

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

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

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

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

支持多个ELF的ftrace

事实上, 如果我们想了解C库中 `printf()` 到 `write()` 的过程, ftrace将是一个很好的工具. 但我们知道, Nanos-lite和它加载的用户程序是两个独立的ELF文件, 这意味着, 如果我们给NEMU的ftrace指定其中一方的ELF文件, 那么ftrace就无法正确将另一方的地址翻译成正确的函数名. 事实上, 我们可以让NEMU的ftrace支持多个ELF: 如果一个地址不属于某个ELF中的任何一个函数, 那就尝试下一个ELF. 通过这种方式, ftrace就可以同时追踪Nanos-lite和用户程序的函数调用了.

温馨提示

PA3阶段2到此结束.

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

简易文件系统

最近在一个项目里需要实现一个简易文件系统，我们是要把多个小文件打包成一个大文件，但还要能快速从大文件中读取任何一个小文件。经过调研和实践，我们最终选择了用 SQLite 来做这个事情。这篇文章，我会从我们遇到的问题开始说起，分享一下我们是如何设计这个简易文件系统，以及 SQLite 在其中扮演了怎样关键的角色。

问题：海量小文件的存储与检索

我们的应用需要处理大量的小文件，这些文件大小通常在几百字节到几千字节之间。一开始，我们采用的是最简单的方案：每个文件对应一个独立的文件。这在开发初期看起来很简单，但随着文件数量的增长，问题开始出现了：

- 文件系统开销：每个文件都需要文件系统为其创建一个条目（inode）来管理。当文件数量达到几万甚至几十万时，这会显著增加文件系统的管理负担，导致性能下降。
- 检索效率：要查找一个特定的小文件，文件系统需要遍历目录结构来定位文件的位置。虽然现代文件系统有优化，但对于大量文件来说，这仍然是一个耗时的操作。
- 备份与迁移：当需要备份或迁移这些文件时，成千上万个小文件的复制和传输，效率通常会非常低下，并且容易出错。

我们需要一个能够将大量小文件高效地组织、存储和检索的解决方案。

为什么选择 SQLite？

在考虑了多种方案后，我们最终选择了 SQLite。这并不是因为它是一个数据库，而是因为它的一个特性：SQLite 可以作为一个单文件的应用程序文件格式来使用。这个特性，正是我们构建简易文件系统的核心。

核心设计：BLOB 存储

我们设计了一个非常简单的表来存储文件内容：

```
CREATE TABLE files (
    id INTEGER PRIMARY KEY,
    name TEXT UNIQUE,
    content BLOB
);
```

这个表结构非常简单，只有三个字段：

- `id`：文件的唯一标识符。
- `name`：文件的名称，通常是相对路径。
- `content`：文件的内容，以 BLOB（二进制大对象）的形式存储。

当一个文件需要被存储时，我们会将它的名称和内容插入到这个表中。当需要读取一个文件时，我们会通过其名称来查询，并从 `content` 字段中获取其内容。

性能考量：事务与批处理

虽然 SQLite 是一个单文件数据库，但它的性能表现依然可以非常出色，前提是我们合理地使用事务。默认情况下，每一条 `INSERT` 或 `UPDATE` 语句都会被自动包装成一个事务，这会带来一定的开销。当我们需要批量插入大量文件时，手动开启一个事务，将所有的 `INSERT` 语句放在这个事务中执行，通常会带来数倍的性能提升。

局限性与未来展望

这个简易文件系统，并不是一个万能的解决方案。它的设计是为特定场景下的特定问题而生的。它的主要局限在于：

- 并发写入：SQLite 默认情况下，对数据库的写入操作是加锁的，这意味着同一时间只能有一个写入操作。对于高并发写入的场景，这可能会成为�bottleneck。
- 文件大小：虽然 SQLite 理论上支持很大的数据，但将非常大的文件（例如几个 GB）存入 BLOB 字段，可能会对数据库的性能产生负面影响。
- 功能：它没有文件系统的权限、目录、软链接等概念。它就是一个纯粹的数据存储。

这个方案，更适合作为一个应用程序的内部数据格式，来管理一组相关的、不需要频繁修改的文件。

一切皆文件

MMORPG的核心玩法通常是一个庞大的世界，玩家可以自由地探索、战斗、与其他玩家互动、交易、组队。这一切都发生在一个持续运行的服务器上。而这一切的基石，就是一个能够实时、高效地管理成千上万个玩家、怪物、NPC、物品、技能、任务状态的数据库系统。

我们的目标是：

- 性能：支持高并发读写，以及低延迟的数据查询，确保玩家的操作能够实时响应，不能有明显的卡顿。
- 灵活性：能够快速地添加新的游戏内容（例如新的怪物、新的物品、新的技能），而不需要对数据库结构进行大规模的修改。
- 持久性：数据必须能够可靠地保存，即使服务器发生崩溃或重启，玩家的进度和状态也不能丢失。

这篇文章，将会分享我们是如何利用 SQLite 这个轻量级、嵌入式数据库，来实现一个高性能的游戏内数据管理系统的。

核心设计：内存优先，异步持久化

为了满足高性能和低延迟的要求，我们采用了一种经典的内存优先（In-Memory First）的设计模式。

- 内存数据库：在服务器启动时，我们会将所有需要持久化的数据，从 SQLite 数据库中加载到内存中，建立一个功能完整的、支持复杂查询的内存数据库。
- 实时操作：所有玩家的实时操作（移动、战斗、交易），都会直接在内存数据库中进行操作，以保证最快的响应速度。
- 异步持久化：为了保证数据不丢失，我们会在后台启动一个独立的线程，定期（例如每隔 5 分钟）将内存数据库中发生了变化的数据，批量地、异步地写入到 SQLite 数据库中。

数据模型：关系型 vs. 键值对

在内存数据库中，我们可以采用多种数据模型。对于这个项目，我们选择了一种混合模式：

- 核心实体（玩家、怪物、NPC）：采用关系型表格。每个实体一个表，字段清晰定义。这保证了数据的结构性和完整性，便于进行复杂的关联查询（例如，查找某个公会的所有成员）。
- 动态属性：对于一些经常变化、结构不固定的属性（例如，一件装备可能会有一个特殊的、随机生成的附加属性），我们会将这些属性序列化（例如，使用 JSON 格式）后，存入一个 `TEXT` 或 `BLOB` 字段中。这样，我们就可以在不修改数据库表结构的情况下，为游戏添加新的内容。

性能优化：索引、事务与批处理

- 索引：我们为经常用于查询（`WHERE` 子句）和连接（`JOIN`）的字段，创建了索引。例如，`player_id` 字段。这能够将查询从全表扫描，变为高效的索引查找。
- 事务：所有对数据库的写入操作，都必须在一个事务（`BEGIN TRANSACTION` ... `COMMIT`）中执行。这能保证数据的一致性，并且能显著提高写入性能。
- 批处理：在内存中，我们会将多个玩家的操作，合并成一个批处理（Batch）。例如，在一个 tick 内，可能有 100 个玩家移动了位置，我们会生成一个包含 100 条 `UPDATE` 语句的事务，来一次性地更新数据库。

数据库迁移（Schema Migration）

随着游戏的开发，我们可能需要为数据库添加新的表、新的字段、新的索引。为了能够平滑地、安全地管理这些数据库结构的变更，我们会编写一个简单的数据库迁移（Schema Migration）脚本。这个脚本，会在服务器启动时，检查数据库的版本，并自动执行必要的 `ALTER TABLE` 和 `CREATE INDEX` 语句，来将数据库升级到最新版本。

总结

这个基于 SQLite 的内存数据库方案，为我们的 MMORPG 项目，提供了一个在开发效率、性能表现、和稳定性之间，极为平衡的选择。它让我们能够专注于游戏逻辑的开发，而不用担心大规模的数据管理问题。

虚拟文件系统

在一个大型的、分布式的应用程序中，我们经常需要对文件系统的操作进行拦截、重定向、或者模拟。例如，在一个测试环境中，我们可能需要模拟磁盘满了的情况；在一个安全沙箱中，我们需要限制一个进程可以访问的文件；或者，在一个跨平台的应用中，我们需要为不同的操作系统，提供一个统一的文件接口。

这篇文章，将会分享我们是如何设计和实现一个高性能、可配置、且功能强大的虚拟文件系统（VFS）的。这个 VFS，是我们内部一个核心的基础设施，它为上层的数据库、缓存、和业务逻辑，提供了一个统一、可编程、且可测试的文件操作接口。

核心设计：接口与实现的分离

我们设计了一个非常简单的接口，来定义一个文件系统应该具备的基本操作：

```
interface IFileSystem {
  read(path): Promise<Buffer>;
  write(path, content): Promise<void>;
  stat(path): Promise<FileStat>;
  mkdir(path): Promise<void>;
  rmdir(path): Promise<void>;
  readdir(path): Promise<string[]>;
  unlink(path): Promise<void>;
}
```

这个接口，定义了文件系统的核心操作：读、写、查看状态、创建目录、删除目录、列出目录内容、删除文件。

我们的核心设计是：

- 接口与实现分离：上层的业务逻辑，只会依赖这个接口，而不会直接调用具体的文件系统实现。
- 可组合性：我们可以创建多个实现，并通过配置来动态地切换它们。

实现一：本地文件系统（Local FS）

这是最简单的实现，它直接将接口的调用，转发给 Node.js 的 `fs` 模块。

```
class LocalFS implements IFileSystem {
  read(path) { return fs.promises.readFile(path); }
  write(path, content) { return fs.promises.writeFile(path, content); }
  stat(path) { return fs.promises.stat(path); }
  mkdir(path) { return fs.promises.mkdir(path); }
  rmdir(path) { return fs.promises.rmdir(path); }
  readdir(path) { return fs.promises.readdir(path); }
  unlink(path) { return fs.promises.unlink(path); }
}
```

这个实现，是生产环境中，默认使用的实现。它保证了，文件操作的原子性、持久性、和性能，与直接使用 Node.js 的 `fs` 模块，是完全一致的。

实现二：内存文件系统（Memory FS）

这个实现，是为了测试、开发、和沙盒环境，而设计的。它将所有的文件和目录，都存在一个内存的对象（Map）中。

```
class MemoryFS implements IFileSystem {
  private root = new Map();
  read(path) { ... }
  write(path, content) { ... }
  stat(path) { ... }
  mkdir(path) { ... }
  rmdir(path) { ... }
  readdir(path) { ... }
  unlink(path) { ... }
}
```

这个实现，的优点是：

- 速度：所有操作，都是纯粹的内存操作，速度极快。
- 隔离性：每个测试用例，可以创建一个全新的、干净的 MemoryFS 实例，来保证测试的独立性，不受外部文件系统的影响。
- 可清理性：测试结束后，只需将 MemoryFS 实例，从内存中释放，即可自动清理所有的文件和目录，无需担心留下垃圾文件。

实现三：沙盒文件系统（Sandbox FS）

这个实现，是一个装饰器（Decorator），它会包装一个已有的 IFileSystem 实例，并在其之上，强制执行一个安全策略。

```
class SandboxFS implements IFileSystem {
  private allowedPaths: string[];
  private fs: IFileSystem;
  constructor(allowedPaths, fs) { ... }
  read(path) {
    if (!this.isAllowed(path)) throw new PermissionDenied();
    return this.fs.read(path);
  }
  // ...
}
```

这个实现，的工作原理是：

- 白名单：在创建时，传入一个允许访问的路径前缀列表（例如，`['/tmp/', '/var/app/data/']`）。
- 拦截：对于所有的文件操作，它会首先检查目标路径，是否在白名单内。
- 委托：如果路径合法，则将操作，委托给内部的、被包装的 IFileSystem 实例，来执行。
- 拒绝：如果路径非法，则会抛出一个 PermissionDenied 错误，来拒绝该操作。

这个实现，是实现安全性、隔离性、和可审计性，的关键。

实现四：远程文件系统（Remote FS）

这个实现，是一个客户端，它会将文件操作，通过网络，发送到一个远程的文件服务器，来执行。

```
class RemoteFS implements IFileSystem {
  private client: HttpClient;
  read(path) { return this.client.post('/read', { path }); }
  write(path, content) { return this.client.post('/write', { path, content }); }
  // ...
}
```

这个实现，的优点是：

- 透明：上层的业务逻辑，完全不知道，它是在与一个远程的文件系统，在交互。
- 集中管理：所有的文件，都存在一个中心服务器上，便于进行备份、监控、和管理。

总结

这个虚拟文件系统，是一个，非常灵活、且可扩展的，设计。它通过，接口与实现的分离，以及，装饰器模式的应用，成功地，将，文件操作的，复杂性，与，业务逻辑，解耦了。

这，为，我们的，项目，带来了，巨大的，好处：

- 可测试性：我们，可以，为，每个，测试，用例，创建，一个，干净，的，隔离，的，文件，系统，来，进行，测试，
- 可配置性：我们，可以，通过，配置，来，为，不，同，的，环，境，（，开，发，、，测，试，、，生，产，），选，择，不，同，的，文，件，系，统，实，现，
- 可扩展性：我们，可以，轻，松，地，添，加，一，个，新，的，文，件，系，统，实，现，（，例，如，，，一，个，基，于，内，存，的，缓，存，文，件，系，统，），来，满，足，新，的，需，求，
- 可维护性：，，，

← 专栏
相关资讯和热榜
更多 Ctrl+Alt+R
Ctrl+Alt+R
全部
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R
Nova中的应用程序
Windows Media Player
MEMU (代表者)
Microsoft Edge
Photos 图片
PNG (图像格式)
Photos 图片
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R
Windows Media Player
PNG (图像格式)
Photos 图片
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R

在GPU中，即使在单核上运行，我们也可以将要处理的数据分成多个部分，并在多个线程中同时处理。这种方式通常被称为“数据并行”。

定义Hello程序

定义Hello程序

在Hello.cpp文件中，我们定义了一个名为Hello的类。这个类将用于管理GPU上的数据和计算任务。

```
class Hello
{
    int *d_a = nullptr;
    int *d_b = nullptr;
    int *d_c = nullptr;
    int *h_c = nullptr;
    int n;
    int blocks, threads;
    int *kernel_1(int, int, int);
    int *kernel_2(int, int, int);
    int *kernel_3(int, int, int);
    int *kernel_4(int, int, int);
    int *kernel_5(int, int, int);
    int *kernel_6(int, int, int);
    int *kernel_7(int, int, int);
    int *kernel_8(int, int, int);
    int *kernel_9(int, int, int);
    int *kernel_10(int, int, int);
    int
```

PA4 - 虚实交错的魔法: 分时多任务

世界诞生的故事 - 第四章

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

! 代码管理

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

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

! 提交要求(请认真阅读以下内容, 若有违反, 后果自负)

预计平均耗时: 40小时

阶段性安排:

- task PA4.1: 实现基本的多道程序系统, 支持带参数的仙剑奇侠传与Hello内核线程的分时运行
- task PA4.2: 实现支持虚存管理的多道程序系统
- task PA4.3: 实现抢占式分时多任务系统, 并提交完整的实验报告

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

多进程程序

多进程程序

以Nginx为例，当大部分工作都是单线程不能时的情况下，进行大规模的并行处理可能要部署以这些高性能进程的多个实例。这种情况下，Nginx的工作进程通常会使用共享内存区域来共享信息，并且使用信号量来协调这些进程。然后使用共享内存区域来存储共享数据。

一个实例说明，共享数据的大小取决于配置，这不会随工作负载变化。共享内存区域通常用于：为工作进程存放会话信息；缓存会话重要的公共数据；记录，如果要记录重要的公共数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据。

多线程程序

多线程程序

以Nginx为例，当大部分工作都是单线程不能时的情况下，进行大规模的并行处理可能要部署以这些高性能进程的多个实例。这种情况下，Nginx的工作进程通常会使用共享内存区域来共享信息，并且使用信号量来协调这些进程。然后使用共享内存区域来存储共享数据。

一个实例说明，共享数据的大小取决于配置，这不会随工作负载变化。共享内存区域通常用于：为工作进程存放会话信息；缓存会话重要的公共数据；记录，如果要记录重要的公共数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据。

内存管理

内存管理

- 管理共享内存区域
- 共享内存区域的内存分配
- 共享内存区域的内存释放

内存管理使用

- 共享内存区域的内存分配，可以分配内存区域的内存分配

内存管理实例

内存管理实例

- 管理共享内存区域
- 共享内存区域的内存分配
- 共享内存区域的内存释放

内存管理常见问题

内存管理常见问题

- 共享内存区域的内存分配，可以分配内存区域的内存分配

上下文功能

上下文功能

以Nginx为例，当大部分工作都是单线程不能时的情况下，进行大规模的并行处理可能要部署以这些高性能进程的多个实例。这种情况下，Nginx的工作进程通常会使用共享内存区域来共享信息，并且使用信号量来协调这些进程。然后使用共享内存区域来存储共享数据。

一个实例说明，共享数据的大小取决于配置，这不会随工作负载变化。共享内存区域通常用于：为工作进程存放会话信息；缓存会话重要的公共数据；记录，如果要记录重要的公共数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据；记录，如果要记录重要的公共数据或者记录所有会话重要的数据。

要使用函数

要使用函数，请在配置文件中使用指令。

使用函数

- 共享内存区域的内存分配
- 共享内存区域的内存释放

共享内存区域的内存分配

共享内存区域的内存分配，可以分配内存区域的内存分配

共享内存区域的内存释放

共享内存区域的内存释放，可以分配内存区域的内存分配

共享内存区域的内存分配

共享内存区域的内存分配

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

上图把这些参数分成两部分，一部分是字符串区域(string area)，另一部分是 `argv/envp` 这两个字符串指针数组，数组中的每一个元素是一个字符串指针，而这些字符串指针都会指向字符串区域中的某个字符串。此外，上图中的 `Unspecified` 表示一段任意长度(也可为0)的间隔，字符串区域中各个字符串的顺序也不作要求，只要用户进程可以通过 `argv/envp` 访问到正确的字符串即可。这些参数的放置格式与ABI手册中的描述非常类似，你也可以参考ICS课本第七章的第六图。

> **阅读ABI手册，理解计算机系统**
>
> 事实上，ABI手册是ISA, OS, 编译器, 运行时环境, C语言和用户进程的桥梁，非常值得大家去阅读. ICS课本上那些让你摸不着头脑的约定，大部分也是出自ABI手册. Linux上遵守的ABI是System V ABI, 它又分为两部分，一部分是和处理器无关的generic ABI(gABI), 例如ELF格式, 动态连接, 文件系统结构等; 另一部分是和处理器相关的processor specific ABI(psABI), 例如调用约定, 操作系统接口, 程序加载等. 你至少应该去看看ABI手册的目录, 翻一下正文部分的图, 这样你就会对ABI手册有一个大致的了解. 如果你愿意深入挖掘一下"为什么这样约定"，那就是真正的"深入理解计算机系统了"。

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

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

这样以后, 用户进程就可以接收到属于它的参数了.

> **给用户进程传递参数**
>
> 这个任务的本质是一个指针相关的编程练习, 不过你需要注意编写可移植的代码. 因为 `call_main()` 是被各种ISA所共享的, 然后修改位置你就传错了地方的话, 如果它接收到一个 `--skip` 参数, 就跳过片头曲标动画的播放, 否则不跳过. 实现这个功能你有利于加速你前奇快传的测试. 商标动画的播放从代码逻辑上距离 `main()` 函数并不远, 于是就交给你来RTFSC吧.
>
> 不过为了给用户进程传递参数, 你还需要修改 `context_uload()` 的原型:
>
> ```
> void context_uload(PCB *pcb, const char *filename, char *const argv[], char *const env
> ```
>
> 这样你就可以在 `init_proc()` 中直接给出用户进程的参数来测试了: 在创建仙剑奇侠传用户进程的时候给出 `--skip` 参数, 你需要观察到仙剑奇侠传确实跳过了商标动画. 目前我们的测试程序中不会用到环境变量, 所以不必传递真实的环境变量字符串. 至于实参应该写什么, 这又是一个指针相关的问题, 就交给你来解决吧.
>
> 让操作系统为每一个用户进程手动设定参数是一件不现实的事情, 因为用户进程的参数还是应该由用户来指定的. 于是最好能有一个方法能把用户指定的参数告诉操作系统, 让操作系统来把指定的参数放到新进程的用户栈里面. 这个方法当然就是系统调用 `SYS_execve` 啦, 如果你去看 `man` , 你会发现它的原型是

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

> **为什么少了一个const?**
>
> 在 `main()` 函数中, `argv` 和 `envp` 的类型是 `char * []` ，而在 `execve()` 函数中, 它们的类型则是 `char *const []` ，从这一差异来看, `main()` 函数中 `argv` 和 `envp` 所指向的字符串是可写的, 你知道为什么会这样吗?

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

- 如何在A的执行流中创建用户进程B?
- 如何结束A的执行流?

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

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

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

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

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

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

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

> **实现带参数的execve()**
>
> 根据上述讲义内容, 实现带参数的 `execve()` ，有一些细节我们并没有完全给出, 例如调用 `context_uload()` 的 `pcb` 参数应该传入什么内容, 这个问题就交给你来思考吧!
>
> 实现后, 运行以下程序:
>
> - 测试程序 `navy-apps/tests/exec-test` ，它会以参数递增的方式不断地执行自身. 不过由于我们没有实现堆区内存的回收, `exec-test` 在运行一段时间之后, `new_page()` 就会把 `0x300000` / `0x3080000` 附近的内存分配出去, 导致用户进程的代码段被覆盖. 目前我们无法修复这一问题, 你只需要看到 `exec-test` 可以正确运行一段时间即可.
> - MENU开机菜单.
> - 完善NTerm的内建Shell, 使得它可以解析输入的参数, 并传递给启动的程序. 例如可以在NTerm中键入 `pal --skip` 来运行仙剑奇侠传并跳过商标动画.

运行Busybox

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

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

```
make menuconfig
```

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

Busybox会把其中的Shell工具链接成一个ELF可执行文件, 而不是像Unix/Linux那样为每个命令行程序的Shell工具都将, 每个工具都是独立的ELF可执行文件. Busybox的 `main()` 函数会根据传入的参数来调用相应工具的功能:

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

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

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

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

> **运行Busybox**
>
> 尝试通过NTerm运行Busybox中的一些简单命令, 比如 `cat` 和 `printenv` 等. 如果你不清楚这些命令的用法, 可以通过 `man` 来查阅它们. 注意不要让这些命令的输出淹没在 `hello_fun()` 打印的信息中, 为此你可能需要调整 `hello_fun()` 打印信息的频率.
>
> 有一些工具并不是放在 `/bin` 目录下, 而是放在 `/usr/bin` 目录下, 例如 `wc` . 为了不必输入完整的路径, 我们可以把 `/usr/bin` 也加入到 `PATH` 环境变量中. 不同的路径通过 ; 进行分隔, 具体格式可以参考在Linux上运行 `echo $PATH` 命令的结果. 这样以后, 我们就可以通过一个库函数 `execvp()` 来尝试遍历 `PATH` 中的所有路径, 直到找到一个存在的可执行文件为止. 找到之后就会调用 `SYS_execve` . 你可以通过阅读 `navy-apps/libs/libc/src/posix/execvp.c` 来了解这一功能是如何实现的.
>
> 不过为了遍历 `PATH` 中的路径, `execvp()` 可能会尝试执行一个不存在的用户程序, 例如 `/bin/wc` . 因此Nanos-lite在处理 `SYS_execve` 系统调用的时候就需要检查将要执行的程序是否存在, 如果不存在, 就需要返回一个错误码. 我们可以通过 `fs_open()` 来进行检查, 如果需要打开的文件不存在, 就返回一个错误码, 此时 `SYS_execve` 返回 `-2` . 另一方面, libos中的 `execve()` 还需要检查系统调用的返回值: 如果系统调用的返回值小于0, 则通常表示系统调用失败, 此时需要将系统调用返回值取负, 作为失败原因设置到一个全局的外部变量 `errno` 中, 然后返回 `-1` .
>
> > **2和errno**
> >
> > `errno` 是C标准定义的, 运行时环境中的一个全局变量, 用于存放最近一次失败的系统调用或库函数调用的错误码. 你可以通过运行 `errno -l` 命令(需要通过 `apt-get` 安装 `moreutils` 包) 来查看所有的错误码及其含义, 你应该能看到错误码 `2` 是你比较熟悉的一种错误. 关于 `errno` 全局变量的更多信息, 可以参考 `man 3 errno` .
>
> > **运行Busybox(2)**
> >
> > 实现上述内容, 让 `execvp()` 支持 `PATH` 的遍历, 然后尝试通过NTerm运行 `wc` 等位于 `/usr/bin` 目录下的命令, 例如 `wc /share/games/bird/atlas.txt` . 你可以通过在Linux上运行相应命令来查看结果是否正确. 此外, 你可以通过阅读 `execvp()` 的代码来了解操作系统是如何判断返回的设置是否正确.

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

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

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

> **温馨提示**
>
> PA4阶段1到此结束.

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

导航

程序和内存位置
绝对代码
可重定位代码
位置无关代码

虚实交错的魔法
分段

程序和内存位置

我们已经实现了一个多道程序的多任务系统, 但却在尝试运行第二个用户进程的时候遇到了问题. 如果你还记得编译Navy-apps中的程序时, 我们都把它们链接到 `0x3000000` 或 `0x83000000` 的内存位置, 你就会知道问题的原因了: 我们在加载第二个用户进程的时候, 就会覆盖第一个进程的内容!

出现上述问题的根本原因是, "在内存中可以同时存在多个进程"的条件被打破了. 显然我们需要对内存进行管理, 而不是让多个进程随意使用. 我们在PA3中提到操作系统有管理系统资源的义务, 在多任务操作系统中, 内存作为一种资源自然也要被管理起来. 操作系统需要记录内存的分配情况, 需要运行一个新进程的时候, 就给它分配一片空闲的内存位置, 把它加载到这一内存位置上即可.

不过, 这片空闲的内存位置是操作系统的加载器在加载时刻指定的, 但进程代码真的可以在这一内存位置上正确运行吗?

绝对代码

一般来说, 程序的内存位置是在链接时刻(link time)确定的(Navy-apps中的程序就是这样), 以前的程序员甚至在程序中使用绝对地址来进行内存访问, 这两种代码称为绝对代码(absolute code). 绝对代码会假设程序中的对象(函数和数据)位于某个固定的位置, 比如链接时重定位之后, 程序就会认为某个变量 `a` 位于内存位置 `0x83001234` , 如果我们把程序加载到别的位置, 加载后变量 `a` 的实际位置就改变了, 但程序中的绝对代码还是认为变量 `a` 仍然位于内存位置 `0x83001234` , 访问 `0x83001234` 也就无法访问真正的变量 `a` . 显然, 绝对代码只能在固定的内存位置才能正确运行.

但操作系统在加载时刻分配的空闲内存位置, 并不总是能让这种程序正确运行. 因此, 这个问题的一个解决方案, 就是让操作系统记录程序的加载位置, 当一个程序试图加载到一个已经被使用的内存位置时, 加载将会失败, 操作系统将返回一个错误. 为了避免加载失败, 一个方法是为每个程序维护多个不同加载地址的版本, 期望其中有一个版本可以被成功加载.

可重定位代码

但这样太麻烦了. 为什么一定要提前确定一个程序的加载位置呢? 如果我们把链接时的重定位阶段往后推迟, 不就可以打破绝对代码的限制了吗?

于是有程序员开发了一类"自重定位(self-relocation)"的特殊程序, 这种程序可以在开始运行的时候, 先把自己重定位到其它内存位置, 然后再开始真正的运行. 这种重定位类型称为"运行时(run time)重定位".

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

但对多任务操作系统来说, 这并没有真正解决问题, 因为程序在运行时刻并不知道重定位的目标内存位置是否空闲.

既然只有操作系统才知道内存是否空闲, 那就干脆让加载器来进行重定位吧, 于是有了"加载时(load time)重定位"的说法. 具体地, 加载器会申请一个空闲的内存位置, 然后将程序加载到这个内存位置, 并把程序重定位到这个内存位置, 之后才会执行这个程序. 今天的GNU/Linux就是通过这种方式来插入内核模块的.

位置无关代码

从某种程度上来说, 加载时重定位会带来额外的开销: 如果一个程序要被重复执行多次, 那么就要进行多次的加载时重定位. 早期的计算机速度较慢, 大家觉得加载时重定位的开销还是不能忍略的, 而且加载时重定位需要对整个程序进行处理, 如果程序比较大的话, 开销就更明显了.

有没有方法可以节省重定位的开销, 甚至不进行重定位呢? 但链接时的重定位又可能会产生绝对代码, 这并不是我们所希望的. 如果程序中的所有寻址, 都是针对程序位置来进行相对寻址操作, 这样的程序就可以被加载到任意位置执行, 而不会出现绝对代码的问题了.

这就是PIC(position-independent code, 位置无关代码)的基本思想. 今天的动态库都是PIC, 这样它们就可以被加载到任意的内存位置了. 此外, 如果一个可执行文件全部由PIC组成, 那么它有一个新名字, 叫PIE(position-independent executable, 位置无关可执行文件). 编译器可以通过特定的选项编译出PIE. 和一般的程序不同, PIE还能在一定程度上对恶意的攻击程序造成了干扰: 恶意程序也无法提前假设PIE运行的地址. 也正是因为这一安全相关的特性, 最近的不少GNU/Linux的发行版上配套的gcc都默认生成PIE. 不过, 使用相对寻址会使得程序的代码量增大, 性能也会受到一些影响, 但对于早期的计算机来说, 内存是一种非常珍贵的资源, 降低性能也是大家不愿意看到的, 因此对于PIC和PIE, 大家也会慎重考虑.

❓ 实现基于PIE的loader (建议二周目思考)

天下并没有免费的午餐, PIE之所以能做到位置无关, 其实是要依赖于程序中一个叫GOT(global offset table, 全局偏移量表)的数据结构. 要正确运行PIE, 加载器需要在加载程序的时候往GOT中填写正确的内容.

有兴趣的同学可以让Nanos-lite的loader支持PIE, 当然这需要了解一些ELF相关的细节, 具体细节可以参考ABI手册.

如果世界上的所有程序都是可重定位的, 或者是PIC, 内存覆盖的问题就不攻自破了. 但总有一些包含绝对代码的程序, 考虑到兼容问题, 还需要想办法运行它们. 有没有更好的, 一劳永逸的方案呢?

虚实交错的魔法

我们刚才是从程序本身来考量, 自然无法绕开绝对代码的问题. 为了解决这个问题, 我们需要从另一个方面 - 内存 - 来思考. 我们知道程序会经历编译, 链接, 加载, 运行这四个阶段, 绝对代码经过编译链接之后, 程序看到的内存地址就会确定下来了, 加载运行的时候就会让程序使用这一内存地址, 来保证程序可以正确运行. 一种尝试是把程序看到的内存和它运行时候真正使用的内存解耦开来, 这就是虚拟内存的思想.

所谓虚拟内存, 就是在真正的内存(也叫物理内存)之上的一层专门给进程使用的抽象. 有了虚拟内存之后, 进程只需要认为自己运行在虚拟地址上就可以了, 真正运行的时候, 才把虚拟地址映射到物理地址. 这样, 我们只要把程序链接到一个固定的虚拟地址, 加载的时候把它们加载到不同的物理地址, 并维护好虚拟地址到物理地址的映射关系, 就可以一劳永逸地解决上述问题了!

那么, 在进程运行的时候, 谁来把虚拟地址映射成物理地址呢? 我们在PA1中已经了解到指令的生命周期:

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

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

但是, 只有操作系统才知道具体要把虚拟地址映射到哪些物理地址上. 所以, 虚拟内存机制是一个软硬协同才能工作的机制: 操作系统负责进行物理内存的管理, 加载进程的时候决定要把进程的虚拟地址映射到哪些物理地址; 等到进程真正运行之前, 还需要配置MMU, 把之前决定好的映射落实到硬件上, 进程运行的时候, MMU就会进行地址转换, 把进程的虚拟地址映射到操作系统希望的物理地址. 注意到这个映射是进程相关的: 不同的进程有不同的映射, 这意味着对不同的进程来说, 同一个虚拟地址可能会被映射到不同的物理地址. 这恰好一劳永逸地解决了内存覆盖的问题. 绝大部分多任务操作系统就是这样做的.

分段

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

分段机制在硬件上的实现可以非常简单, 只需要在MMU中实现一个段基址寄存器就可以了. 操作系统在运行不同进程的时候, 就在段基址寄存器中设置不同的值, MMU会把进程使用的虚拟地址加上段基址, 来生成真正用于访问内存的物理地址, 这样就实现了"让不同的进程使用不同的段"的目的. 作为教学操作系统的Minix就是这样工作的, 一些简单的嵌入式系统和实时系统, 也是通过分段机制来进行虚存管理.

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

```
        15                0    31                                    0
LOGICAL +-----------------+    +-------------------------------------+
ADDRESS |    SELECTOR     |    |              OFFSET                 |
        +----+----------+---+    +-------------------+-----------------+
        +------+          V                          |
        | DESCRIPTOR TABLE                           |
        |  +-------------+                           |
        |  |             |                           |
        |  |             |                           |
        |  |             |                           |
        |  |             |                           |
        |  |             |                           |
        |  |-------------|                           |
        |  |  SEGMENT    | BASE          +----+    |
        +->| DESCRIPTOR  |---------------->| + |<------+
           |-------------| ADDRESS       +-+--+
           |             |                 |
           +-------------+                 |
                                           V
              LINEAR  +-------------+-------------+---------------+
              ADDRESS |    DIR      |   PAGE      |    OFFSET     |
                      +-------------+-------------+---------------+
```

咋看之下真是眼花缭乱, 让人一头雾水.

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

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

此页面修订于: 2021-10-02 22:50:51

超级容器的界限

超级容器的界限是指超级容器在一定时间内，能够同时存在的、Google容器中运行的Linux内核实例的最大数量。这个界限通常是一个动态变化的数值，其具体数值取决于当时资源Google容器所支持的硬件和软件配置。它通常不会固定在某个具体数值，而是会根据实时资源状况进行动态调整。这个界限的存在是为了确保Google容器能够稳定运行，并防止因资源过度分配而导致性能下降或崩溃。

这个界限主要受以下几个因素的影响：

- 内存资源：这是最主要的限制因素。每个Linux内核实例都需要一定的内存来运行。Google容器会根据可用内存来决定能够启动多少个内核实例，以确保每个实例都有足够的内存来运行。
- 计算资源：CPU的处理能力也是一个重要的限制因素。每个Linux内核实例都需要一定的CPU资源来执行任务。Google容器会根据可用CPU资源来决定能够启动多少个内核实例，以确保每个实例都有足够的CPU资源来运行。
- 网络资源：网络带宽也是一个重要的限制因素。每个Linux内核实例都需要一定的网络带宽来进行通信。Google容器会根据可用网络带宽来决定能够启动多少个内核实例，以确保每个实例都有足够的网络带宽来进行通信。
- 存储资源：存储空间也是一个重要的限制因素。每个Linux内核实例都需要一定的存储空间来存储数据。Google容器会根据可用存储空间来决定能够启动多少个内核实例，以确保每个实例都有足够的存储空间来存储数据。

分页

在计算机系统，分页是一种内存管理技术，它将内存分成固定大小的页（Page），并将虚拟内存地址映射到物理内存地址。分页的目的是为了提高内存的利用率，并简化内存管理。

kernel code 和 kernel data 是内核代码和数据，它们通常会被映射到每个进程的虚拟内存空间中，以便内核可以访问这些数据。这样做的目的是为了提高内核的性能，因为内核需要频繁地访问这些数据，如果每次都需要从磁盘加载，会导致性能下降。

user data 和 user code 是用户进程的数据和代码，它们通常会被映射到物理内存的不同位置，以确保每个进程都有自己的内存空间，并防止进程之间的内存冲突。

页表（Page Table）

简单来说，页表是一张用于将虚拟内存地址转换为物理内存地址的表。它通常由一个或多个页表项（Page Table Entry，PTE）组成，每个页表项包含了虚拟页面的信息，例如物理内存中对应的页面的地址、该页面的权限（读、写、执行）、该页面是否被加载到内存中、该页面是否被修改过、该页面是否被引用过等。页表的作用是为了实现虚拟内存到物理内存的映射，并提供内存保护和管理的功能。

页表的结构通常是多级的，以提高查找效率。例如，一个三级页表可能包含一个顶级页表，指向多个中级页表，每个中级页表指向多个底级页表，底级页表包含实际的页表项。这样做的目的是为了减少内存占用，并提高查找效率。

页表项（Page Table Entry，PTE）是页表中的一个条目，它包含了关于一个虚拟页面的信息，以便将虚拟地址转换为物理地址。一个典型的页表项通常包含以下信息：

- 物理页帧号（Physical Frame Number，PFN）：指向该虚拟页面在物理内存中对应的物理页帧的编号。
- 权限位（Permission Bits）：指示该页面的权限，例如读、写、执行。这些权限通常以位的形式表示，每个位代表一种权限。
- 存在位（Present Bit）：指示该页面是否被加载到内存中。如果该位为1，则表示该页面在内存中；如果该位为0，则表示该页面不在内存中，需要从磁盘加载。
- 脏位（Dirty Bit）：指示该页面是否被修改过。如果该位为1，则表示该页面被修改过；如果该位为0，则表示该页面未被修改过。
- 引用位（Reference Bit）：指示该页面是否被引用过。这个位通常用于实现页面置换算法，以决定哪些页面应该被换出内存。

页表项（Page Table Entry，PTE）

PAGE FRAME
PGD
PUD
PMD
OFFSET
PAGE DIRECTORY
PAGE TABLE
PHYSICAL ADDRESS
PT_TO_DIRTY
PT_TO_WRITE

页表项（Page Table Entry，PTE）是页表中的一个条目，它包含了关于一个虚拟页面的信息，以便将虚拟地址转换为物理地址。一个典型的页表项通常包含以下信息：

- 物理页帧号（Physical Frame Number，PFN）：指向该虚拟页面在物理内存中对应的物理页帧的编号。
- 权限位（Permission Bits）：指示该页面的权限，例如读、写、执行。这些权限通常以位的形式表示，每个位代表一种权限。
- 存在位（Present Bit）：指示该页面是否被加载到内存中。如果该位为1，则表示该页面在内存中；如果该位为0，则表示该页面不在内存中，需要从磁盘加载。
- 脏位（Dirty Bit）：指示该页面是否被修改过。如果该位为1，则表示该页面被修改过；如果该位为0，则表示该页面未被修改过。
- 引用位（Reference Bit）：指示该页面是否被引用过。这个位通常用于实现页面置换算法，以决定哪些页面应该被换出内存。

页表（Page Table）

页表（Page Table）是一张用于将虚拟内存地址转换为物理内存地址的表。它通常由一个或多个页表项（Page Table Entry，PTE）组成，每个页表项包含了虚拟页面的信息，例如物理内存中对应的页面的地址、该页面的权限（读、写、执行）、该页面是否被加载到内存中、该页面是否被修改过、该页面是否被引用过等。页表的作用是为了实现虚拟内存到物理内存的映射，并提供内存保护和管理的功能。

页表的结构通常是多级的，以提高查找效率。例如，一个三级页表可能包含一个顶级页表，指向多个中级页表，每个中级页表指向多个底级页表，底级页表包含实际的页表项。这样做的目的是为了减少内存占用，并提高查找效率。

超级管理模式

目录

- 超级管理模式
- 一、内核管理
- 二、内存管理
- 三、进程管理
- 四、文件系统
- 五、设备管理
- 六、网络子系统
- 七、安全子系统
- 八、虚拟化子系统
- 九、性能与调试子系统
- 十、内核模块与扩展

内核管理模块的内存管理

一、内核管理

- 内核管理：内核管理模块负责管理内核的运行，包括内存管理、进程管理、文件系统、设备管理、网络子系统、安全子系统、虚拟化子系统、性能与调试子系统、内核模块与扩展。
- 内存管理：内核管理模块负责管理内存的分配和释放，以确保内核和用户进程能够高效地使用内存。
- 进程管理：内核管理模块负责管理进程的创建、调度、终止和通信，以确保多个进程能够高效地运行。
- 文件系统：内核管理模块负责管理文件系统的挂载、卸载、文件的创建、删除、读取和写入，以确保文件系统能够高效地运行。
- 设备管理：内核管理模块负责管理设备的驱动程序，以确保设备能够高效地运行。
- 网络子系统：内核管理模块负责管理网络协议的实现，以确保网络能够高效地运行。
- 安全子系统：内核管理模块负责管理安全策略的实现，以确保系统能够安全地运行。
- 虚拟化子系统：内核管理模块负责管理虚拟机的创建、调度、终止和通信，以确保虚拟机能够高效地运行。
- 性能与调试子系统：内核管理模块负责管理性能监控和调试工具的实现，以确保系统能够高效地运行。
- 内核模块与扩展：内核管理模块负责管理内核模块的加载、卸载、编译和安装，以确保内核能够灵活地扩展。

二、内存管理

- 内存分配：内核管理模块负责为内核和用户进程分配内存，以确保内存能够高效地使用。
- 内存释放：内核管理模块负责释放不再使用的内存，以确保内存能够高效地回收。
- 内存保护：内核管理模块负责保护内存，以确保内核和用户进程的内存不会被对方访问或修改。
- 内存映射：内核管理模块负责将虚拟内存映射到物理内存，以确保内存能够高效地使用。
- 内存交换：内核管理模块负责将内存中的数据交换到磁盘上，以确保内存能够高效地使用。
- 内存压缩：内核管理模块负责压缩内存中的数据，以确保内存能够高效地使用。
- 内存回收：内核管理模块负责回收不再使用的内存，以确保内存能够高效地回收。
- 内存碎片整理：内核管理模块负责整理内存碎片，以确保内存能够高效地使用。
- 内存统计：内核管理模块负责统计内存的使用情况，以确保内存能够高效地管理。
- 内存调试：内核管理模块负责调试内存的问题，以确保内存能够高效地调试。

三、进程管理

- 进程创建：内核管理模块负责创建进程，以确保进程能够高效地创建。
- 进程调度：内核管理模块负责调度进程，以确保进程能够高效地运行。
- 进程终止：内核管理模块负责终止进程，以确保进程能够高效地终止。
- 进程通信：内核管理模块负责进程通信，以确保进程能够高效地通信。
- 进程同步：内核管理模块负责进程同步，以确保进程能

超级容器的界限

超级容器的界限是指超级容器在一定时间内，能够同时存在的、Google容器中运行的Linux内核实例的最大数量。这个界限通常是一个动态变化的数值，其具体数值取决于当时资源Google容器所支持的硬件和软件配置。它通常不会固定在某个具体数值，而是会根据实时资源状况进行动态调整。这个界限的存在是为了确保Google容器能够稳定运行，并防止因资源过度分配而导致性能下降或崩溃。

这个界限主要受以下几个因素的影响：

- 内存资源：这是最主要的限制因素。每个Linux内核实例都需要一定的内存来运行。Google容器会根据可用内存来决定能够启动多少个内核实例，以确保每个实例都有足够的内存来运行。
- 计算资源：CPU的处理能力也是一个重要的限制因素。每个Linux内核实例都需要一定的CPU资源来执行任务。Google容器会根据可用CPU资源来决定能够启动多少个内核实例，以确保每个实例都有足够的CPU资源来运行。
- 网络资源：网络带宽也是一个重要的限制因素。每个Linux内核实例都需要一定的网络带宽来进行通信。Google容器会根据可用网络带宽来决定能够启动多少个内核实例，以确保每个实例都有足够的网络带宽来进行通信。
- 存储资源：存储空间也是一个重要的限制因素。每个Linux内核实例都需要一定的存储空间来存储数据。Google容器会根据可用存储空间来决定能

超级容器的界限

超级容器的界限是指超级容器在一定时间内，能够同时存在的、Google容器中运行的Linux内核实例的最大数量。这个界限通常是一个动态变化的数值，其具体数值取决于当时资源Google容器所支持的硬件和软件配置。它通常不会固定在某个具体数值，而是会根据实时资源状况进行动态调整。这个界限的存在是为了确保Google容器

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

对于x86和riscv32，你无需实现TLB.

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

让DiffTest支持分页机制

为了让DiffTest机制正确工作，你需要

- 对于x86:
  - 在 `restart()` 函数中我们需要对CR0寄存器初始化为 `0x60000011` ，但我们不必关心其含义.
  - 实现分页机制中accessed位和dirty位的功能
- 处理 `attach` 命令时，需要将分页机制相关的寄存器同步到REF中
- 对快照功能进行更新

这毕竟是个选做题而已，实现细节就不提示了，遇到困难就自己思考一下解决方案吧.

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

RISC-V的分页机制

如果你仔细RTFM，你会发现标准RISC-V的分页机制需要在S模式及U模式下才能开启，而在M模式下的话存并不会进行MMU的地址转换，但我们在NEMU中进行了简化，允许M模式的访存也进行地址转换，这样可以避免以S模式相关的知识，让大家把注意力集中在分页机制本身.

在分页机制上运行用户进程

成功实现分页机制之后，你会发现创剑奇侠传也同样成功运行了，但仔细想想就会发现这其实不太对劲: 我们在 `vme_init()` 中创建了内核的虚拟地址空间，之后就再也没有切换过这一虚拟地址空间. 也就是说，我们让创剑奇侠传也运行在内核的虚拟地址空间之上! 这不太合理，毕竟用户进程还是应该有自己的一套虚拟地址空间. 更可况，Navy-apps之前以用户程序链接到 `0x300000` / `0x83000000` 的位置，是因为之前Nanos-lite并没有对空间的物理内存进行管理: 现在引入了分页机制，由MMU来负责所有物理页的分配. 这意味着, 如果将来MMIE `0x300000` / `0x83000000` 所在的物理页分配出去, 创剑奇侠传的内容将会被覆盖!你之前在运行 `exec-test` 的时候已经遇到过这个问题了! 因此, 目前仍旧奇侠传看似运行成功, 其实重蹈覆辙杀机.

正确的做法是, 我们应该让用户进程运行在操作系统为其分配的虚拟地址空间之上. 为此, 我们需要对工程作一些变动. 首先, 编译Navy应用程序的时候需要为 `make` 添加 `VME=1` 的参数, 这样就可以将应用程序的链接地址改为 `0x40000000` , 这是为了避免用户进程的虚拟地址空间与内核相互重叠, 从而产生非预期的错误. 这时, "虚拟地址"也为物理地址的抽象"这一好处已经体现出来了: 原则上用户程序可以运行在任意的虚拟地址, 不受物理内存容量的限制. 我们让用户进程的代码从 `0x40000000` 附近开始, 这个地址已经不在物理内存的地址空间中(NEMU提供的物理内存是128MB), 但分页机制保证了进程能够正确运行. 这样, 链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址, 它们只要使用虚拟地址就可以了, 而虚拟地址和物理地址之间的映射则全部交给操作系统的MM来管理.

此外, 我们还需要对创建用户进程的过程进行较多的改动. 我们首先需要在加载用户进程之前为其创建地址空间. 由于地址空间是进程相关的, 我们将 `AddrSpace` 结构体作为PCB的一部分. 这样以后, 我们只需要在 `context_uload()` 的开头调用 `protect()` , 就可以实现地址空间的创建. 目前这个地址空间除了内核映射之外就没有其它内容了, 具体可以参考 `abstract-machine/am/src/$ISA/nemu/vme.c` .

不过, 此时 `loader()` 不能直接把用户进程加载到内存位置 `0x40000000` 附近了, 因为这个地址并不在内核的虚拟地址空间中, 内核不能直接访问它. `loader()` 要做的事情是, 获取程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 通过 `map()` 把这一物理页映射到用户进程的虚拟地址空间中. 由于AM native实现了权限检查, 为了让程序可以在AM native上正确运行, 你调用 `map()` 的时候需要将 `prot` 设置成可读可写可执行
- 从文件中读入一页的内容到这一物理页中

这一切都是为了让用户进程在将来可以正确地运行: 用户进程在将来使用虚拟地址访问内存, 在loader为用户进程建护的映射下, 虚拟地址被转换成物理地址, 通过这一物理地址访问到的物理内存, 恰好就是用户进程想要访问的数据.

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

最后, 为了让这一地址空间生效, 我们还需要将它落实到MMU中, 具体地, 我们希望在CTE恢复进程上下文的时候来切换地址空间. 为此, 我们需要将进程的地址空间相关信息保存在 `as->ptr` 加入到上下文中, 相关代码已经实现了这一功能(见 `abstract-machine/am/include/arch/$ISA-nemu.h`), 在x86中这一成员为 `cr3` , 而在mips32/riscv32中则为 `pdir` . 你还需要

- 修改 `ucontext()` 的实现, 在创建的用户进程上下文中设置地址空间描述符指针
- 在 `__am_irq_handle()` 的开头调用 `__am_get_cur_as()` (在 `abstract-machine/am/src/$ISA/nemu/vme.c` 中定义), 来将当前的地址空间描述符指针保存到上下文中
- 在 `__am_irq_handle()` 返回前调用 `__am_switch()` (在 `abstract-machine/am/src/$ISA/nemu/vme.c` 中定义)来切换地址空间, 将被调度进程的地址空间落实到MMU中

为mips32实现真正的分页机制

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

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

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

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

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

在分页机制上运行用户进程

根据上述的讲义内容, 对创建用户进程的过程进行相应改动, 让用户进程在分页机制上成功运行. 如果你选择了mips32或riscv32, 请注意地址空间描述符在上下文结构体中的位置, 如果你不确定这一位置, 请根据PA3中的讲义内容检查你的代码实现.

为了测试实现的正确性, 我们先单独运行dummy(则忘记修改调度代码), 并先在 `exit` 的实现中调用 `halt()` 结束系统的运行, 这是因为让其它程序成功运行还需要进行一些额外的改动. 如果你的实现正确, 你会看到dummy程序最后输出GOOD TRAP的信息, 说明它确实在分页机制上成功运行了.

让DiffTest支持分页机制(2)

如果你选择的是riscv32, 为了让DiffTest机制正确地支持用户进程的运行, 你还需要:

- 实现 `M` 和 `U` 两种特权级模式, 具体地
  - 在NEMU中实现 `mstatus.MPP` 位的功能
  - 执行 `ecall` 指令时, 根据当前特权级做出不同号码的异常, 但在CTE中对它们进行统一的处理
  - 创建内核线程上下文时, 额外将 `mstatus.MPP` 设置为 `M` 模式
  - 创建用户进程上下文时, 额外将 `mstatus.MPP` 设置为 `U` 模式
- 填写页表时, 需要额外设置 `R` , `W` , `X` , `U` , `A` , `D` 位
- 创建用户进程上下文时, 为 `mstatus` 额外设置 `MXR` 和 `SUM` 位

内核映射的作用

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

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

尝试注释这处代码, 重新编译并运行, 你会看到发生了错误. 请解释为什么会发生这个错误.

为了在分页机制上运行仙剑奇侠传, 我们还需要考虑堆区的问题. 之前我们让 `mm_brk()` 函数直接返回 `0` , 表示用户进程的堆区大小修改总是成功, 这是因为在实现分页机制之前, `0x300000` / `0x83000000` 之上的内存都可以让用户进程自由使用. 现在用户进程运行在分页机制之上, 我们还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中, 这样才能保证运行在分页机制上的用户进程可以正确地访问新申请的堆区.

为了识别堆区中的哪些空间是新申请的, 我们还需要记录堆区的位置. 由于每个进程的堆区使用情况是独立的, 我们需要为它们分别维护堆区的位置. 因此我们在PCB中添加成员 `max_brk` , 来记录program break曾经达到的最大位置. 引入 `max_brk` 是为了简化实现: 我们可以不实现堆区的回收功能, 而是只为当前新program break超过 `max_brk` 部分的虚拟地址空间分配物理页.

在分页机制上运行仙剑奇侠传

根据上述内容, 实现 `nanos-lite/src/mm.c` 中的 `mm_brk()` 函数. 你需要注意 `map()` 参数是否需要投页对齐的问题(这取决于你的 `map()` 实现).

实现正确后, 仙剑奇侠传就可以正确在分页机制上运行了.

一致性问题 - mips32的噩梦来了

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

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

native的VME实现

尝试阅读 `native` 的VME实现, 你发现 `native` 是如何实现VME的? 为什么可以这样做?

可以在用户栈里面创建用户进程上下文吗?

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

支持虚存管理的多道程序

终于一大圈引入了虚存管理, 现在我们终于回来了: 我们可以支持多个用户进程的并发执行了. 不过我们还是先让运行在分页机制上的用户进程和内核线程并发执行, 来对分页机制进行进一步的测试.

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

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

支持虚存管理的多道程序

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

不过我们会发现, 和之前相比, 在分页机制上运行的仙剑奇侠传的性能有了明显的下降. 尽管NEMU在串行模拟MMU的功能, 并不能完全代表硬件MMU的真实运行情况, 但这也说明了虚存机制确实会带来额外的运行时开销. 由于这个原因, 60年代的工程师普遍对虚存机制有所顾虑, 不敢轻易在系统中实现虚存机制. 但"不必停这程序即可让多个程序并发运行"的好处越来越明显, 以至于虚存机制成为了现代计算机系统的标配.

并发执行多个用户进程

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

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

Hint: 程序是个状态机.

这一阶段的内容算是整个PA中最难的了, 连选做题的难度和之前比也不是一个量级的. 这也展示了构建系统的挑战: 随着一个系统趋于完善, 模块之间的交互会越来越复杂, 代码看似避繁就简却可谓字字珠玑, 牵一发而动全身. 不过这是项目复杂度上升的必然现象, 等到代码量到了一定程度, 即使是开发一个应用程序, 也会面临类似的困难. 那我们要如何理解和管理规模日趋复杂的项目代码呢?

答案是抽象. 所以你在PA中看到各种各样的API, 我们并不是随随便便定义它们的, 它们确实蕴含了模块行为的本质, 从而帮助我们更容易地从宏观的角度理解整个系统的行为. 就算是调试, 这些API对我们梳理代码的行为也有巨大的帮助. 当你在理解, 实现, 调试这些API的过程中, 你对整个系统的认识也会越来越深刻. 如果你确实独立完成到这里, 就算你以后面对更复杂的项目, 相信你也不会畏惧了.

温馨提示

PA4阶段2到此结束.

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

目 · 概览
分析器优势
分析器工作原理
惯用分析器
惯用分析器
字符过滤器
标记化器
内容感知模式下的
令牌过滤器
上下文切换
令牌
令牌
令牌

我们可以对上述代码进行一些简单的优化，一处可以优化的地方是把 `pp` 的功能合并到 `ksp` 中，让 `ksp` 既可以表示内核栈的地址，也可以表示进入CTE之前的特权级，注意到 `ksp` 总是在 `pp == USER` 的时候被使用，而且 `ksp` 的值肯定不为 `0`，因此我们可以通过 `ksp == 0` 来表示 `pp == KERNEL`，我们对 `ksp` 的值进行如下的约定：

- 若当前位于用户态，则 `ksp` 的值为内核栈的栈底
- 若当前位于内核态，则 `ksp` 的值为 `0`

这样以后，上述代码可以优化为：

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

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

  // save context

  __am_irq_handle(c);

  // restore context

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

  return_from_trap();
}
```

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

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

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

  // save context

  __am_irq_handle(c);

  // restore context

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

  return_from_trap();
}
```

栈切换的具体实现

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

mips32

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

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

你只需要根据上述约定在CTE中添加少量代码，就可以实现内核栈切换的功能了。

riscv32

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

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

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

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

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

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

❓ 临时寄存器的方案

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

x86

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

因此x86必须通过硬件机制来提供上述问题的解决方案：首先我们必须让硬件识别出当前的特权级，这是通过 `CS` 寄存器中的 `RPL` 域来识别的(也即386手册中 `CPL` 的定义)，若 `CS.RPL` 为0，表示处理器处于内核态；若 `CS.RPL` 为3，表示处理器处于用户态，然后我们根据根据386手册的Figure 9-5的异常处理机制和 `iret` 指令的行为进行扩展：

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

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

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

然后我们可以来完比TSS机制的处理过程了：

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

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

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

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

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

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

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

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

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

  return true;
}
```

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

📝 实现内核栈和用户栈之间的切换

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

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

一些提示：

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

需要注意的是，我们目前只允许最多一个需要更新画面的进程参与调度，这是因为多个这样的进程并发运行会导致画面被相互覆盖，影响画面输出的效果，在真正的操作系统中，通常由一个窗口管理进程统一管理画面的显示，需要显示画面的进程与这一管理进程进行通信，来实现更新画面的目的，但这需要操作系统支持进程间通信的机制，这已经超出了ICS的范围，而且Nanos-lite作为一个裁剪版的操作系统，也不提供进程间通信的服务，因此我们进行了简化，最多只允许一个需要更新画面的进程参与调度即可。

💬 真实系统的复杂性

通过剧析"并发执行多个用户进程"需要解决的问题和方案，我们看了到当一些学过的知识点组合起来的时候，问题的复杂度可能会指数增长：中断随时会到来，自陷操作可能会发生，这些都可能会引发上下文切换；一方面这可能会导致虚拟地址空间的切换，另一方面在上下文返回的时候，有可能需要切换到用户栈，也有可能切换到内核栈，而这又取决于一些变量的维护；这些变量虽然都是用过C语言来定义和使用，但在中断等常规上下文切换的共同作用下，全局变量以及Context结构体中的成员变量有着不同的生存期，我们需要根据不同的场景将一些关键的变量定义到不同的位置...

现代计算机系统一般都配备多核处理器，运行着多核操作系统，于是多了一类和处理器数量相关的变量：除了中断异常的处理带来并发的场景之外，多个处理器是会真正地并行执行代码，如何在多核系统中正确地解决各种并发问题一直以来都是计算机系统领域的一个挑战。

❓ Nanos-lite与并发bug (建议二周目/学完操作系统课思考)

上文讨论并发的时候提到：更一般地，用户进程都会并发地执行系统调用，操作系统还需要保证它们都能按照系统调用的语义正确地执行。

我们在PA3中知道 `printf()` 会通过 `malloc()` 申请堆冲区，而 `malloc()` 又可能会执行 `_sbrk()`，通过 `SYS_brk` 陷入内核；在上一个阶段中，我们实现了支持分页机制的 `mm_brk()`，在必要的时候它会通过 `new_page()` 申请一页，而仙剑和hello用户进程都会调用 `printf()`，使得它们可能会并发执行 `SYS_brk`，思考一下，目前Nanos-lite的设计会导致并发bug吗？为什么？
```

目 录 概览
编写文件的内容
通过文件进行交互式
运行ONDemandRpcClient
运行ONDemandRpcServer
实现服务
实现服务
序列化数据，编写自定义数据类型
序列化数据，通过数据类型ID
序列化数据，使用Google Protobuf
实现客户端
实现服务器
小结
习题

编写不同的传输

在Pipe的基础，我们可以编写不同的传输。

提示你的计算机系统

Pipe的数据流量不可以进行打开，因此，我们可以进行使用管道来传输数据。管道本质，我们可以理解为管道文件，Pipe的数据，实际保存在管道文件中。

Windows系统，管道文件，保存在 \\.\pipe\ 目录下。可以用命令行管理管道文件查看它的信息，管道文件，我们可以理解为管道文件，Pipe的数据，实际保存在管道文件中。 \\.\pipe\ 目录下的文件，

此为了Google NCM官方文档翻译，仅供爱好者Linux/unix内核开发 ncm + sm 的朋友们。

[x] Devices ····
[x] Network ····
Screen size changed ····

实现工具箱，但现代化的NMU入口在QMobileUpr了。

工具箱概述

NCMNUMUPR的安装方式，就是直接解压缩一个压缩包，然后运行一个脚本，就可以开始使用了。

和TOMA、和其他的一个一般都是Linux的，但是现在是要更古代的桌面系统的Linux桌面，OMAD目前对应的目标已经不是“桌面系统的Linux”了，而是“Linux桌面”。

下李解释各 - 操作系统课程辅助

KDE的内容为各位大学生或者研究生的毕业设计，或者是的课程之大学辅助用的工具。

- 程序设计方法
- 计算机原理与接口

操作系统是现代计算机的一个非常重要的基础设施，现在的现代的操作系统的内容，一般都是一个一个的一个一个的一个一个的

PA是什么意思

PA是不是组成原理/体系结构实验

PA是不是硬件

为什么要用这么复杂的PA来折磨我

我将来不从事系统相关的工作，我不会那么痛苦了

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

为什么不能把代码/心得/攻略上传到公开的地方让大家学习

就不能公布那些最初的参考答案，这样即便bug调不出来，也能往后做

我总觉得这个问题不清楚

为什么要用这么复杂的PA来折磨我

我将来不从事系统相关的工作，我不会那么痛苦了

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

为什么不能把代码/心得/攻略上传到公开的地方让大家学习

就不能公布那些最初的参考答案，这样即便bug调不出来，也能往后做

我总觉得这个问题不清楚

我将来不从事系统相关的工作，我不会那么痛苦了

我是

≡ 导航

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

站得高，看得远
实践是检验真理的唯一标准

为什么要学习计算机系统基础

一知半解

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

数组求和

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

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

整数的平方

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

多重定义符号

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

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

在上述两个模块链接生成的可执行文件被执行时，`main` 函数的 `printf` 语句打印出来的值是：`d=0,x=107269248` 。为什么不是 `d=100,x=200` ？

奇怪的函数返回值

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

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

- `fun(0)` 和 `fun(1)` 为 `3.14`
- `fun(2)` 为 `3.1399998664856`
- `fun(3)` 为 `2.00000006103516`
- `fun(4)` 为 `3.14` 并会发生 `访问违例` 这是为什么？

时间复杂度和功能都相同的程序

```
void copy1(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
void copy2(int src[2048][2048], int dst[2048][2048]) {
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

上述两个功能完全相同的函数，时间复杂度也完全一样，但在Pentium 4处理器上执行时，所测时间相差大约21倍，这是为什么？猜猜看是 `copy1` 更快还是 `copy2` 更快？

网友贴出的一道百度招聘题

请给出以下C语言程序的执行结果，并解释为什么。

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

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

整数除法

以下两个代码段的运行结果是否一样呢？

- 代码段一：`c int a = 0x80000000; int b = a / -1; printf("%d\n", b); c`
- 代码段二：

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

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

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

站得高，看得远

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

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

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

随着大规模数据中心(WSC)的建立和个人移动设备(PMD)的大量普及使用，计算机发展进入了后PC时代，呈现出"人与信息世界及物理世界融合"的趋势和网络化、服务化、普适化和智能化的鲜明特征. 后PC时代WSC、PMD和PC等并存，使得原先基于PC机提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"设计向变为更强调"系统"设计.

后PC时代，并行成为重要主题. 培养具有系统观的, 能够进行软, 硬件协同设计的软硬件贯通人才是关键. 而且，后PC时代对于大量从事应用开发的应用程序员的要求也变得更高. 首先，后PC时代的应用程序员面临更复杂, 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特徵.

从图中可以看出，计算机系统由不同的抽象层构成，"计算"的过程就是不同抽象层转换的过程，上层是下层的抽象，而下层则是上层的具体实现. 计算机学科主要研究的是计算机系统各个不同抽象层的实现及其相互转换的机制, 计算机学科培养的应该主要是在计算机系统或在系统某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, 名比率和时延, 并发和并行, 这程过程调用(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, 程序移植和建过性等方面成为高手; 建立扎实的计算机系统概念, 为后续的OS, 编译, 体系结构等课程打下坚实基础.

实践是检验真理的唯一标准

旷日持久的计算机教学只为解答三个问题：

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

除了纯理论工作之外, 计算机相关的工作无不强调动手实践的能力. 很多时候, 你会觉得理解某一个知识点是一件简单容易, 但当你真正动手实践的时候, 你才发现你的之前的理解只是停留在表面, 例如你知道链表的基本结构, 但你能写出一个正确的链表程序吗？你知道程序加载的基本原理, 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:

> **| 思考题**
>
> 当你运行一个Hello World程序的时候, 计算机究竟做了些什么？

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

目前也有以下观点:

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

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

而对这些原理的透彻理解, 离不开动手实践.

> **| 来公读桌**
>
> 学汽车制造专业是要学发动机怎么设计, 学开车怎么开得过司机吧?

Linux入门教程

以下内容引用自公开的技术资料及教程资料，并有少量修改和补充，如果你是第一次使用Linux，请你一定仔细阅读教程，一定要记住所有操作细节的内容。

探索命令行

Linux是专门设计的命令使用模式都是相同的：

```
命令名称 参数1 参数2 参数3 ...
```

参数之间用空格隔开，参数之间不会有空格，对于命令行，可以用选项 -选项名称 来表示，然后跟在行结束等参数的一些：

- ls 用于列出当前目录的文件下的内容，如果不使用任何参数，则会列出当前目录的文件，ls -l 可以显示详细信息，使用 man 命令查看帮助手册。
- mkdir 用来创建目录，使用方法 mkdir 目录名称
- cd 目录 可以切换到 目录 目录，在Linux中，每个目录中都至少包含两个目录：.，指向目录目录自身，以及 ..，指向该目录的上级目录，使用 cd .. 可以切换到上级目录。
- touch NEWFILE 可以创建一个内容为空的新文件 NEWFILE，和 NEWFILE 已存在，其内容不会改变。
- cp SRCFILE DEST 可以复制文件 SRCFILE 文件复制为 DEST 文件，如果 DEST 是一个目录，则将 SRCFILE 文件复制到该目录下。
- mv SRCFILE DEST 可以移动文件 SRCFILE 文件重命名为 DEST 文件，如果 DEST 是一个目录，则将 SRCFILE 文件移动到该目录下。
- rm -f -r -i -v -d -p -m -s -u -a -b -c -e -g -h -k -l -n -o -p -q -t -u -v -w -x -y -z -A -B -C -D -E -F -G -H -I -J -K -L -M -N -O -P -Q -R -S -T -U -V -W -X -Y -Z -1 -2 -3 -4 -5 -6 -7 -8 -9 -0 -- FILE 删除指定文件 FILE，如果目录不为空则会报错，如果目录为空，则会删除目录，使用选项 -r 递归删除目录下的内容，使用 -f 强制删除，使用 -i 交互模式，使用 -v 显示删除过程，使用 -d 删除空目录，使用 -p 删除目录及其父目录，使用 -m 删除目录下的文件名中含有 -m 参数的文件，使用 -s 删除目录下的文件名中含有 -s 参数的文件，使用 -u 删除目录下的文件名中含有 -u 参数的文件，使用 -a 删除目录下的文件名中含有 -a 参数的文件，使用 -b 删除目录下的文件名中含有 -b 参数的文件，使用 -c 删除目录下的文件名中含有 -c 参数的文件，使用 -e 删除目录下的文件名中含有 -e 参数的文件，使用 -g 删除目录下的文件名中含有 -g 参数的文件，使用 -h 删除目录下的文件名中含有 -h 参数的文件，使用 -k 删除目录下的文件名中含有 -k 参数的文件，使用 -l 删除目录下的文件名中含有 -l 参数的文件，使用 -n 删除目录下的文件名中含有 -n 参数的文件，使用 -o 删除目录下的文件名中含有 -o 参数的文件，使用 -p 删除目录下的文件名中含有 -p 参数的文件，使用 -q 删除目录下的文件名中含有 -q 参数的文件，使用 -t 删除目录下的文件名中含有 -t 参数的文件，使用 -u 删除目录下的文件名中含有 -u 参数的文件，使用 -v 删除目录下的文件名中含有 -v 参数的文件，使用 -w 删除目录下的文件名中含有 -w 参数的文件，使用 -x 删除目录下的文件名中含有 -x 参数的文件，使用 -y 删除目录下的文件名中含有 -y 参数的文件，使用 -z 删除目录下的文件名中含有 -z 参数的文件，使用 -A 删除目录下的文件名中含有 -A 参数的文件，使用 -B 删除目录下的文件名中含有 -B 参数的文件，使用 -C 删除目录下的文件名中含有 -C 参数的文件，使用 -D 删除目录下的文件名中含有 -D 参数的文件，使用 -E 删除目录下的文件名中含有 -E 参数的文件，使用 -F 删除目录下的文件名中含有 -F 参数的文件，使用 -G 删除目录下的文件名中含有 -G 参数的文件，使用 -H 删除目录下的文件名中含有 -H 参数的文件，使用 -I 删除目录下的文件名中含有 -I 参数的文件，使用 -J 删除目录下的文件名中含有 -J 参数的文件，使用 -K 删除目录下的文件名中含有 -K 参数的文件，使用 -L 删除目录下的文件名中含有 -L 参数的文件，使用 -M 删除目录下的文件名中含有 -M 参数的文件，使用 -N 删除目录下的文件名中含有 -N 参数的文件，使用 -O 删除目录下的文件名中含有 -O 参数的文件，使用 -P 删除目录下的文件名中含有 -P 参数的文件，使用 -Q 删除目录下的文件名中含有 -Q 参数的文件，使用 -R 删除目录下的文件名中含有 -R 参数的文件，使用 -S 删除目录下的文件名中含有 -S 参数的文件，使用 -T 删除目录下的文件名中含有 -T 参数的文件，使用 -U 删除目录下的文件名中含有 -U 参数的文件，使用 -V 删除目录下的文件名中含有 -V 参数的文件，使用 -W 删除目录下的文件名中含有 -W 参数的文件，使用 -X 删除目录下的文件名中含有 -X 参数的文件，使用 -Y 删除目录下的文件名中含有 -Y 参数的文件，使用 -Z 删除目录下的文件名中含有 -Z 参数的文件，使用 -1 删除目录下的文件名中含有 -1 参数的文件，使用 -2 删除目录下的文件名中含有 -2 参数的文件，使用 -3 删除目录下的文件名中含有 -3 参数的文件，使用 -4 删除目录下的文件名中含有 -4 参数的文件，使用 -5 删除目录下的文件名中含有 -5 参数的文件，使用 -6 删除目录下的文件名中含有 -6 参数的文件，使用 -7 删除目录下的文件名中含有 -7 参数的文件，使用 -8 删除目录下的文件名中含有 -8 参数的文件，使用 -9 删除目录下的文件名中含有 -9 参数的文件，使用 -0 删除目录下的文件名中含有 -0 参数的文件，使用 -- 删除目录下的文件名中含有 -- 参数的文件，使用 -- 删除目录下的文件名中含有 -- 参数的文件，使用 -- 删除目录下的文件名中含有 --

≡ 导航

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

git快速入门

光玉

想象一下你正在玩Flappy Bird, 你今晚的目标是拿到100分, 不然就不睡觉. 经过千辛万苦, 你拿到了99分, 就要看到成功的曙光的时候, 你竟然失手了! 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我? 为什么不让我存档?"

想象一下你正在写代码, 你今晚的目标是实现某一个新功能, 不然就不睡觉. 经过千辛万苦, 你终于把代码写好了, 保存并编译运行, 你看到调试信息一行一行地在终端上输出. 就要看到成功的曙光的时候, 竟然发生了段错误! 你仔细思考, 发现你之前的构思有着致命的错误, 但之前正确运行的代码已经永远离你而去了. 你悲痛欲绝, 滴血的心在呼喊着, "为什么上天要这样折磨我?" 你绝望地倒在屏幕前... 这时, 你发现身边渐渐出现无数的光玉, 把你包围起来, 耀眼的光芒令你无法睁开眼睛... 等到你回过神来, 你发现屏幕上正是那份之前正确运行的代码! 但在你的记忆中, 你确实经历过那悲痛欲绝的时刻... 这一切真是不可思议啊...

人生如戏, 戏如人生

人生就像不能重玩的Flappy Bird, 但软件工程领域却并非如此, 而那不可思议的光玉就是"版本控制系统". 版本控制系统给你的开发流程提供了比朋也收集的更强大的光玉, 能够让你在过去和未来中随意穿梭, 避免上文中的悲剧降临你的身上.

没听说过版本控制系统就完成实验, 艰辛地排除万难, 就像游戏通关之后才知道原来游戏可以存档一样, 其实玩游戏的时候进行存档并不是什么丢人的事情.

在实验中, 我们使用 `git` 进行版本控制. 下面简单介绍如何使用 `git` .

游戏设置

首先你得安装 `git` :

```
apt-get install git
```

安装好之后, 你需要先进行一些配置工作. 在终端里输入以下命令

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

经过这些配置, 你就可以开始使用 `git` 了.

在实验中, 你会通过 `git clone` 命令下载我们提供的框架代码, 里面已经包含一些 `git` 记录, 因此不需要额外进行初始化. 如果你想在别的实验/项目中使用 `git` , 你首先需要切换到实验/项目的目录中, 然后输入

```
git init
```

进行初始化.

查看存档信息

使用

```
git log
```

查看目前为止所有的存档.

使用

```
git status
```

可以得知, 与当前存档相比, 哪些文件发生了变化.

存档

你可以像以前一样编写代码. 等到你的开发取得了一些阶段性成果, 你应该马上进行"存档".

首先你需要使用 `git status` 查看是否有新的文件或已修改的文件未被跟踪, 若有, 则使用 `git add` 将文件加入跟踪列表, 例如

```
git add file.c
```

会将 `file.c` 加入跟踪列表. 如果需要一次添加所有未被跟踪的文件, 你可以使用

```
git add -A
```

但这样可能会跟踪了一些不必要的文件, 例如编译产生的 `.o` 文件, 和最后产生的可执行文件. 事实上, 我们只需要跟踪代码源文件即可. 为了让 `git` 在添加跟踪文件之前作筛选, 你可以编辑 `.gitignore` 文件 (你可以使用 `ls -a` 命令看到它), 在里面给出需要被 `git` 忽略的文件和文件类型.

把新文件加入跟踪列表后, 使用 `git status` 再次确认. 确认无误后就可以存档了, 使用

```
git commit
```

提交工程当前的状态. 执行这条命令后, 将会弹出文本编辑器, 你需要在第一行中添加本次存档的注释, 例如"fix bug for xxx". 你应该尽可能添加详细的注释, 将来你需要根据这些注释来区别不同的存档. 编写好注释之后, 保存并退出文本编辑器, 存档成功. 你可以使用 `git log` 查看存档记录, 你应该能看到刚才编辑的注释.

读档

如果你遇到了上文提到的让你悲痛欲绝的情况, 现在你可以使用光玉来救你一命了. 首先使用 `git log` 来查看已有的存档, 并决定你需要回到哪个过去. 每一份存档都有一个hash code, 例如 `b87c512d10348fd8f1e32ddea8ec95f87215aaa5` , 你需要通过hash code来告诉 `git` 你希望读哪一个档. 使用以下命令进行读档:

```
git reset --hard b87c
```

其中 `b87c` 是上文hash code的前缀: 你不需要输入整个hash code. 这时你再看看你的代码, 你已经成功地回到了过去!

但事实上, 在使用 `git reset` 的hard模式之前, 你需要再三确认选择的存档是不是你的真正目标. 如果你读入了一个较早的存档, 那么比这个存档新的所有记录都将被删除! 这意为着你不能随便回到"将来"了.

第三视点

当然还是有办法来避免上文提到的副作用的, 这就是 `git` 的分支功能. 使用命令

```
git branch
```

查看所有分支. 其中 `master` 是主分支, 使用 `git init` 初始化之后会自动建立主分支.

读档的时候使用以下命令

```
git checkout b87c
```

而不是 `git reset` . 这时你将处于一个虚构的分支中, 你可以

- 查看 `b87c` 存档的内容
- 使用以下命令切换到其它分支

  ```
  git checkout 分支名
  ```

- 对代码的内容进行修改, 但你不能使用 `git commit` 进行存档, 你需要使用

  ```
  git checkout -B 分支名
  ```

  把修改结果保存到一个新的分支中, 如果分支已存在, 其内容将会被覆盖

不同的分支之间不会相互干扰, 这也给项目的分布式开发带来了便利. 有了分支功能, 你就可以像第三视点那样在一个世界的不同时间(一个分支的多个存档), 或者是多个平行世界(多个分支)之间来回穿梭了.

更多功能

以上介绍的是 `git` 的一些基本功能, `git` 还提供很多强大的功能, 例如使用 `git diff` 比较同一个文件在不同版本中的区别, 使用 `git bisect` 进行二分搜索来寻找一个bug在哪次提交中被引入...

其它功能的使用请参考 `git help` , `man git` , 或者在网上搜索相关资料.

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

初识man
寻找帮助
探索man
开始旅程

man快速入门

这是一个man的使用教程, 同时给出了一个如何寻找帮助的例子.

初识man

你是一只Linux菜鸟, 因为课程实验所迫, 你不得不使用Linux, 不得不使用十分落后的命令行. 实验内容大多数都要在命令行里进行, 面对着一大堆陌生的命令和参数, 这个链接中的饼图完美地表达了你的心情.

不行! 还是得认真做实验, 不然以后连码农都当不上了! 这样的想法鞭策着你, 因为你知道, 就算是码农, 也要有适应新环境和掌握新工具的能力. "还是先去找man吧." 于是你在终端里输入 `man` , 敲了回车. 只见屏幕上输出了一行信息:

> What manual page do you want?

噢, 原来命令行也会说人话! 你明白这句话的意思, `man` 在询问你要查询什么内容. 你能查询什么内容呢? 既然 `man` 会说人话, 还是先多了解 `man` 吧. 为了告诉 `man` 你想要了解ta, 你输入

```
man man
```

敲了回车之后, `man` 把你带到了一个全新的世界. 这时候, 你又看到了一句人话了, 那是 `man` 的独白, ta告诉你, ta的真实身份其实是

> an interface to the on-line reference manuals

接下来, ta忽然说了一大堆你听不懂的话, 似乎是想告诉你ta的使用方法. 可是你还没做好心理准备啊, 于是你无视了这些话.

寻找帮助

很快, 你已经看到"最后一行"了. 难道man的世界就这么狭小? 你仔细一看, "最后一行"里面含有一些信息:

> Manual page man(1) line 1 (press h for help or q to quit)

原来可以通过按 `q` 来离开这个世界啊, 不过你现在并不想这么做, 因为你想多了解 `man` , 以后可能会经常需要 `man` 的帮助. 为了更了解ta, 你按了 `h` .

这时你又被带到了新的世界, 世界的起点是"SUMMARY OF LESS COMMANDS", 你马上知道, 这个世界要告诉你如何使用 `man` , 你十分激动. 于是你往下看, 这句话说"带有'''标记的命令可以在前面跟一个数, 这时命令的行为在括号里给出". 这是什么意思? 你没看懂, 还是找个带'''的命令试试吧. 你继续往下看, 看到了两个功能和相应的命令:

- 第一个是展示帮助, 原来除了 `h` 之外, `H` 也可以看到帮助, 而且这里把帮助的命令放在第一个, 也许 `man` 想暗示你, 找到帮助是十分重要的.
- 第二个命令是退出. "哈哈, 知道怎么退出之后, 就不用通过重启来退出一个命令行程序啦", 你心想. 但你现在还是不想退出, 还是再看看其它的吧.

继续往下看, 你看到了用于移动的命令. 果然, 你还是可以在这个世界里面移动的. 第一个用于移动的功能是往下移动一行, 你看到有5种方法可以实现:

```
e  ^E  j  ^N  CR
```

`e` 和 `j` 你看懂了, 就是按 `e` 或者 `j` . 但 `^E` 是什么意思呢? 你尝试找到 `^` 的含义, 但是你没找到, 还是让我告诉你吧. 在上下文和按键有关的时候, `^` 是Linux中的一个传统记号, 它表示 `ctrl+` . 还记得Windows下 `ctrl+c` 代表复制的例子吗? 这里的 `^E` 表示 `ctrl+E` . `CR` 代表回车键, 其实 `CR` 是控制字符(ASCII码小于32的字符)的一个, 这里有一段关于控制字符的问答.

你决定使用 `j` , 因为它像一个向下的箭头, 而且它是右手食指所按下的键. 其实这点和 `vim` 的使用是类似的, 如果你不能理解为什么 `vim` 中使用 `h` , `j` , `k` , `l` 作为方向键, 这里有一个初学者的提问. 事实上, 这是一种touch typing.

你按下了 `j` , 发现画面上的信息向下滚动了一行. 你看到了 `*` , 想起了 `*` 标记的命令可以在前面跟一个数. 于是你试着输入 `10j` , 发现画面向下滚动了10行, 你第一次感觉到在这个"丑陋"的世界中也有比GUI方便的地方. 你继续阅读帮助, 并且尝试每一个命令. 于是你掌握了如何通过移动来探索 `man` 所在的世界.

继续往下翻, 你看到了用于搜索的命令. 你十分感动, 因为使用关键字可以快速定位到你关心的内容. 帮助的内容告诉你, 通过按 `/` 激活前向搜索模式, 然后输入关键字(可以使用正则表达式), 按下回车就可以看到匹配的内容了. 帮助中还列出了后向搜索, 跳到下一匹配处等功能. 于是你掌握了如何使用搜索.

探索man

你一边阅读帮助, 一边尝试新的命令, 就这样探索着这个陌生的世界. 你虽然记不住这么多命令, 但你知道你可以随时来查看帮助. 掌握了一些基本的命令之后, 你按 `q` 离开了帮助, 回到了 `man` 的世界. 现在你可以自由探索 `man` 的世界了. 你向下翻, 跳过了看不懂的 `SYNOPSIS` 小节, 在 `DESCRIPTION` 小节看到了人话, 于是你阅读这些人话. 在这里, 你看到整个manual分成9大类, 每个manual page都属于其中的某一类; 你看到了一个manual page主要包含以下的小节:

- NAME - 命令名
- SYNOPSIS - 使用方法大纲
- CONFIGURATION - 配置
- DESCRIPTION - 功能说明
- OPTIONS - 可选参数说明
- EXIT STATUS - 退出状态, 这是一个返回给父进程的值
- RETURN VALUE - 返回值
- ERRORS - 可能出现的错误类型
- ENVIRONMENT - 环境变量
- FILES - 相关配置文件
- VERSIONS - 版本
- CONFORMING TO - 符合的规范
- NOTES - 使用注意事项
- BUGS - 已经发现的bug
- EXAMPLE - 一些例子
- AUTHORS - 作者
- SEE ALSO - 功能或操作对象相近的其它命令

你还看到了对 `SYNOPSIS` 小节中记号的解释, 现在你可以回过头来看 `SYNOPSIS` 的内容了. 但为了弄明白每个参数的含义, 你需要查看 `OPTIONS` 小节中的内容.

你想起了搜索的功能, 为了弄清楚参数 `-k` 的含义, 你输入 `/-k` , 按下回车, 并通过 `n` 跳过了那些 `OPTIONS` 小节之外的 `-k` , 最后大约在第254行找到了 `-k` 的解释: 通过关键字来搜索相关功能的manual page. 在 `EXAMPLES` 小节中有一个使用 `-k` 的例子:

```
man -k printf
```

你阅读这个例子的解释: 搜索和 `printf` 相关的manual page. 你还是不太明白这是什么意思, 于是你退出 `man` , 在命令行中输入

```
man -k printf
```

并运行, 发现输出了很多和 `printf` 相关的命令或库函数, 括号里面的数字代表相应的条目属于manual的哪一个大类. 例如 `printf (1)` 是一个shell命令, 而 `printf (3)` 是一个库函数. 要访问库函数 `printf` 的manual page, 你需要在命令行中输入

```
man 3 printf
```

当你想做一件事的而不知道用什么命令的时候, `man` 的 `-k` 参数可以用来列出候选的命令, 然后再通过查看这些命令的manual page来学习怎么使用它们.

接下来, 你又开始学习 `man` 的其它功能...

开始旅程

到这里, 你应该掌握 `man` 的用法了. 你应该经常来拜访ta, 因为在很多时候, ta总能给你提供可靠的帮助.

在这个励志的故事中, 你学会了:

- 阅读程序输出的提示和错误信息
- 通过搜索来定位你关心的内容
- 动手实践是认识新事物的最好方法
- 独立寻找帮助, 而不是一有问题就问班上的大神

于是, 你就这样带着 `man` 踏上了Linux之旅...

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

导航

全局类型
Monitor相关
寄存器相关
指令执行相关
虚拟内存相关
中断异常相关
DiffTest相关
NEMU作为DUT时使用
NEMU作为REF时使用

NEMU ISA相关的API说明文档

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

全局类型

```
word_t;
```

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

```
sword_t;
```

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

```
char *FMT_WORD;
```

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

Monitor相关

```
char isa_logo[];
```

用于在未实现指令的报错信息中提示开发者阅读相关的手册.

```
word_t RESET_VECTOR;
```

表示PC的初始值.

```
void init_isa();
```

在monitor初始化时调用, 进行至少如下ISA相关的初始化工作:

- 设置必要的寄存器初值, 如 `PC` 等
- 加载内置客户程序

寄存器相关

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

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

```
CPU_state cpu;
```

寄存器结构的全局定义.

```
void isa_reg_display();
```

用于打印寄存器当前的值.

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

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

指令执行相关

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

用于存放ISA相关的译码信息, 会嵌入在译码信息结构体 `Decode` 的定义中.

```
vaddr_t isa_fetch_decode(Decode *s);
```

取出 `s->pc` 指向的指令并译码, 同时更新 `s->snpc` , 最后返回一个用于标识指令执行辅助函数的ID.

虚拟内存相关

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

检查当前系统状态下对内存区间为 `[vaddr, vaddr + len)` , 类型为 `type` 的访问是否需要经过地址转换, 其中 `type` 可能为:

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

函数返回值可能为:

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

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

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

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

中断异常相关

```
vaddr_t isa_raise_intr(word_t NO, vaddr_t epc);
```

抛出一个号码为 `NO` 的异常, 其中 `epc` 为触发异常的指令PC, 返回异常处理的出口地址.

```
word_t isa_query_intr();
```

查询当前是否有未处理的中断, 若有则返回中断号码, 否则返回 `INTR_EMPTY` .

DiffTest相关

NEMU作为DUT时使用

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

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

```
void isa_difftest_attach();
```

将当前的所有状态同步到REF, 并在之后的执行中开启DiffTest.

NEMU作为REF时使用

```
void isa_difftest_regcpy(void *dut, bool direction);
```

`direction` 为 `DIFFTEST_TO_DUT` 时, 获取REF的寄存器状态到 `dut` ;
`direction` 为 `DIFFTEST_TO_REF` 时, 设置REF的寄存器状态为 `dut` .

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

触发一个编号为 `NO` 的异常.

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

x86指令系统简介

i386手册有一套专门用怎了所有指令的细节，附录中的Opcode map也很有用。在这里，我们先对x86指令系统作一些简单的梳理，当你对x86指令系统有有任何疑惑时，请查阅i386手册，关于指令系统的一切细节都在里面。

i386手册推荐

由于PC厂商本的i386手册的印刷错误较多，一定程度上影响理解，我们在gitHub上开放了一个repo，用于提供修复版本的手册，同时我们也为修复错误后的手册提供在线的HTML版本。

如果你在做实验的过程中出现了奇的错误，欢迎帮助我们修复这些错误。

指令格式

x86指令的一般格式如下：

```
+----------------------------------------------------------------------------------------+
|[instruction] address- | | operand- |segment |opcode|ModR/M| SIB |displacement| immediate |
| prefix  |size prefix|size prefix|override|      |      |     |            |           |
|----------------------------------------------------------------------------------------|
|  0 OR 1 |  0 OR 1   |  0 OR 1   | 0 OR 1 | 1 OR 2|0 OR 1|0 OR 1| 0,1,2 OR 4 |0,1,2 OR 4 |
|----------------------------------------------------------------------------------------|
|                                                number of bytes                         |
+----------------------------------------------------------------------------------------+
```

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

```
100017:    66 c7 84 99 00 e0 ff ff 01 00       movw   $0x1,-0x2000(%ecx,%ebx,4)
```

其组成部分的划分如下：

```
+----------------------------------------------------------------------------------------+
|[instruction] address- | | operand- |segment |opcode|ModR/M| SIB |displacement| immediate |
| prefix  |size prefix|size prefix|override|      |      |     |            |           |
|----------------------------------------------------------------------------------------|
|                                   66                   c7    84   99    00 e0 ff ff    01 00 |
+----------------------------------------------------------------------------------------+
```

关什么 0x84 要被解释成 ModR/M 字节呢？这是由 opcode 决定的，opcode 决定了这是什么指令的什么形式，同时也决定了 opcode 之后的比特串的解释。如果你熟悉汇编语言，opcode 那部很清晰可见。

PA中的x86经过简化，address-size prefix 和 segment override prefix 都不会用到，因此NEMU也不需要实现这两者的功能。

❗ 编码的艺术

对于以下5个复杂：

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

它们是两不相交的，这是必须的！这背后反映了设计思想：

对于以下5个复杂：

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

它们是两不相交的，这是必须的！这背后反映了设计思想：

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

```
100017:    66 c7 84 99 00 e0 ff ff 01 00       movw   $0x1,-0x2000(%ecx,%ebx,4)
```

其组成部分的划分如下：

```
+----------------------------------------------------------------------------------------+
|[instruction] address- | | operand- |segment |opcode|ModR/M| SIB |displacement| immediate |
| prefix  |size prefix|size prefix|override|      |      |     |            |           |
|----------------------------------------------------------------------------------------|
|                                   66                   c7    84   99    00 e0 ff ff    01 00 |
+----------------------------------------------------------------------------------------+
```

PA中的x86经过简化，address-size prefix 和 segment override prefix 都不会用到，因此NEMU也不需要实现这两者的功能。

❗ 编码的艺术

对于以下5个复杂：

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

它们是两不相交的，这是必须的！这背后反映了设计思想：

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

```
100017:    66 c7 84 99 00 e0 ff ff 01 00       movw   $0x1,-0x2000(%ecx,%ebx,4)
```

其组成部分的划分如下：

```
+----------------------------------------------------------------------------------------+
|[instruction] address- | | operand- |segment |opcode|ModR/M| SIB |displacement| immediate |
| prefix  |size prefix|size prefix|override|      |      |     |            |           |
|----------------------------------------------------------------------------------------|
|                                   66                   c7    84   99    00 e0 ff ff    01 00 |
+----------------------------------------------------------------------------------------+
```

关什么 0x84 要被解释成 ModR/M 字节呢？这是由 opcode 决定的，opcode 决定了这是什么指令的什么形式，同时也决定了 opcode 之后的比特串的解释。如果你熟悉汇编语言，opcode 那部很清晰可见。

导航

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

x86的mov指令执行例子剖析

我们举两个 `mov` 指令的例子, 它们是NEMU自带的客户程序 `mov` 中的两条指令:

```
100000:    b8 34 12 00 00        mov    $0x1234,%eax
......
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

简单mov指令的执行

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

取指(instruction fetch, IF)

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

译码(instruction decode, ID)

在 `fetch_decode_exec()` 函数中用这条指令的第一个字节 `0xb8` 来查找 `switch-case` 的分支, 发现这一指令的操作数宽度是 `4` 字节的 `mov` 指令, 形式是将立即数移入寄存器(move immediate to register).

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

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

接下来还需要识别指令的操作数. 对于 `mov $0x1234, %eax` 指令来说, 识别操作数其实就是识别寄存器 `%eax` 和立即数 `$0x1234` . 在x86中, 通用寄存器都有自己的编号, `I2r` 形式的指令把寄存器编号也放在指令的第一个字节里面, 我们可以通过位运算将寄存器编号抽取出来; 立即数存放在指令的第二个字节, 可以很容易得到它. 需要说明的是, 由于立即数是指令的一部分, 我们还需要通过 `instr_fetch()` 函数来获得它.

此外, x86的译码辅助函数的命名很直观, 直接反映了操作数的类型和数据流向.
以 `nemu/src/isa/x86/instr/decode.c` 为例, 其中的译码辅助函数主要以i386手册附录A中的操作数表示记号来命名. 例如 `I2r` 表示将立即数移入寄存器, 其中 `I` 表示立即数, `2` 表示英文 `to` , `r` 表示通用寄存器, 更多的记号请参考i386手册.

执行(execute, EX)

对于 `mov $0x1234, %eax` 指令来说, 执行阶段的工作就是把立即数 `$0x1234` 送到寄存器 `%eax` 中. 由于 `mov` 指令的功能可以统一成"把源操作数的值传送到目标操作数中", 而译码阶段已经把操作数都准备好了, 所以只需要针对 `mov` 指令编写一个执行辅助函数即可. 这个函数就是 `exec_mov()` , 它是通过 `def_EHelper` 宏来定义的:

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

其中 `write_operand()` 函数会根据第二个参数中记录的类型的不同进行相应的写操作, 包括写寄存器和写内存. `print_asm_template2()` 是个宏, 用于输出带有两个操作数的指令的汇编形式.

更新PC

调用 `update_pc()` 即可.

复杂mov指令的执行

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

首先是取指. 这条mov指令比较特殊, 它的第一个字节是 `0x66` , 如果你查阅i386手册, 你会发现 `0x66` 是一个 `operand-size prefix` . 因为这个前缀的存在, 本例中的 `mov` 指令才能被CPU识别成 `movw` . NEMU通过一个ISA相关的译码信息成员 `s->isa.is_operand_size_16` 来记录操作数宽度前缀是否出现, `switch-case` 语句中的 `case 0x66` 分支实现了这个功能. `case 0x66` 分支对 `s->isa.is_operand_size_16` 成员变量做了标识之后, 通过 `goto` 语句回到函数的开头重新取出操作码, 此时取得了真正的操作码 `0xc7` . 由于 `s->isa.is_operand_size_16` 成员变量进行过标识, 在 `set_width()` 函数中将会确定操作数长度为 `2` 字节.

x86的操作数宽度处理

大部分x86指令都有不同操作数宽度的版本, 因此x86的操作数宽度信息记录会更复杂: 首先通过 `set_width()` 函数记录译码查找表中记录的操作数宽度; 若这一操作数宽度结果为 `0` , 表示仅仅根据操作码来判断, 操作数宽度还不能确定, 可能是16位或者32位, 需要通过 `s->isa.is_operand_size_16` 成员变量来决定.

接下来是识别操作数. 根据操作码 `0xc7` 查找相应的 `case` 分支, 调用译码辅助函数 `decode_mov_I2E()` , 这个译码辅助函数又分别调用 `decode_op_I()` 和 `operand_rm()` 这两个操作数译码辅助函数来取出操作数. 阅读代码, 你会发现 `operand_rm()` 最终会调用 `read_ModR_M()` 函数. 由于本例中的 `mov` 指令需要访问内存, 因此除了要识别出立即数之外, 还需要确定好要访问的内存地址. x86通过 `ModR/M` 字节来指示内存操作数, 支持各种灵活的寻址方式. 其中最一般的寻址格式是

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

相应内存地址的计算方式为

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

其它寻址格式都可以看作这种一般格式的特例, 例如

> displacement(R[base_reg])

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

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

```
100017:    66 c7 84 99 00 e0 ff    movw   $0x1,-0x2000(%ecx,%ebx,4)
10001e:    ff 01 00
```

根据 `mov_I2E` 的指令形式, `0xc7` 是 `opcode` , `0x84` 是 `ModR/M` 字节. 在i386手册中查阅表格17-3得知, `0x84` 的编码表示在 `ModR/M` 字节后面还跟着一个 `SIB` 字节, 然后跟着一个32位的 `displacement` . 于是读出 `SIB` 字节, 发现是 `0x99` . 在i386手册中查阅表格17-4得知, `0x99` 的编码表示 `base_reg = ECX, index_reg = EBX, scale_factor = 4` . 在 `SIB` 字节后面读出一个32位的 `displacement` , 发现是 `00 e0 ff ff` , 在小端存储方式下, 它被解释成 `-0x2000` . 于是内存地址的计算方式为

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

框架代码已经实现了 `load_addr()` 函数和 `read_ModR_M()` 函数(在 `nemu/src/isa/x86/decode.c` 中定义), 它们的函数原型为

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

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

本例中的执行阶段就是要把立即数写入到相应的内存位置. 译码阶段已经把操作数准备好了, 执行辅助函数 `exec_mov()` 会完成数据移动的操作, 最终在 `update_pc()` 函数中更新PC.

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