

# The Missing Course of Your Computer Science Education

王慧妍

[why@nju.edu.cn](mailto:why@nju.edu.cn)

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

---

- 关于《计算机系统基础》习题课
- 关于传说中的PA实验
- 学术诚信
- 选讲PA0

# 关于《计算机系统基础》习题课

课程网站：[http://www.why.ink:8080/ICS/2023/Main\\_Page](http://www.why.ink:8080/ICS/2023/Main_Page)

实验网站：<https://nju-projectn.github.io/ics-pa-gitbook/ics2023/>

# 习题课的内容

---

- 《计算机系统基础》理论课中遗失的实践细节
- 对你们从Lab/PA中生存下来是至关重要的
- 对你们未来XX年作为“码农”的生涯都是至关重要的
- 本讲标题来自于 [The Missing Semester of Your CS Education](#)
  - jyy墙裂推荐！
  - 我们miss的比这门课多
- 习题课（和课程网站）会布置所有作业
  - PA（编程大实验）
  - Lab（编程小实验）
  - Homework（书后习题）——以理论课为准

# 老师/助教的使用

- 联系/求助：使用邮件

- 能使你更好地整理问题。也许在整理问题的过程中你就发现答案了
- 老师：
  - 王慧妍：[why@nju.edu.cn](mailto:why@nju.edu.cn);
- 查重助教：
  - 刘瀚之 [jm23333@outlook.com](mailto:jm23333@outlook.com)
  - 张灵毓 [zly@smail.nju.edu.cn](mailto:zly@smail.nju.edu.cn)
- 答疑助教：
  - 林朗 [211850008@smail.nju.edu.cn](mailto:211850008@smail.nju.edu.cn)
  - 胡皓明 [211250182@smail.nju.edu.cn](mailto:211250182@smail.nju.edu.cn)
  - 潘昕田 [211240001@smail.nju.edu.cn](mailto:211240001@smail.nju.edu.cn)



- Ask

- 对上课内容没有理解的地方、对课程/学习的疑惑、.....

- Do not ask

- 安装XXX错了怎么办？Segmentation Fault怎么办？

# 关于传说中地狱难度PA的一些真相

- 它很难，的确很难
  - 确切地说，对最优异的同学来说依然有一些挑战性
    - 如果你感到异常困难，你更需要的其实是C/C++编程的训练
    - (并且我们知道这一点！)
- 往年一些同学都因为不诚信的举动获得了成绩
  - 他们得到了相应的报应（例如在《操作系统》中惨挂）
  - (并且我们知道这一点！)
- 你总是可以耍一些小聪明，从别人那里得到帮助
  - 但越是独立完成，受到的训练就越好
  - (并且我们知道这一点！)

# 你们可能比较关心的

- 分数占比 (期末只剩下30%)
  - PA: 40%、Lab: 15%、Homework: 15%

- 评分标准

- DDL
  - 10% bonus
  - 100%封顶
- Hard DDL
  - 80%分数

## 3.2.1 PA: 几乎完全客观评分

PA0~4

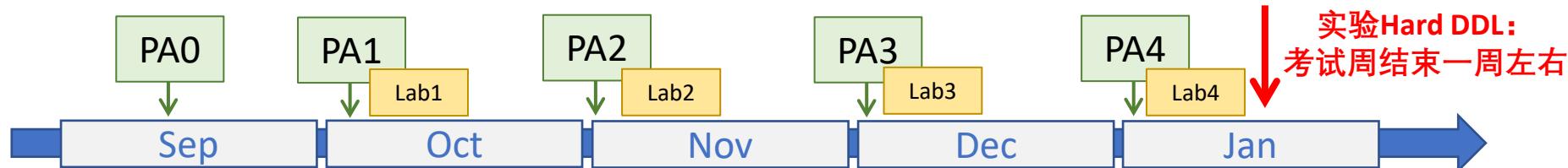
- Rejected, 编译错误或没有通过任何测试用例: 10% (诚信分)
- Partial Accepted, 部分 easy 测试通过 (此时不运行 hard 用例): 10%~60%, 根据比例加权换算
- Partial Accepted, 全部 easy 测试通过, 部分hard测试通过: 60%~80%, 根据比例加权换算
- Accepted, 通过全部 easy/hard 测试:  $\geq 80\%$ , 剩余部分由人工评价给出
- 没有通过全部 easy 测试用例的作业将没有人工评分的机会 (即意味着实验报告不得分。但我们会阅读你的反馈)。

## 3.2.2 Labs: 完全客观评分

Lab1~4

- Rejected, 编译错误或没有通过任何测试用例: 10% (诚信分)
- Partial Accepted, 部分 easy 测试通过 (此时不运行 hard 用例): 50%
- Partial Accepted, 全部 easy 测试通过 (hard 用例没有全部通过): 75%
- Accepted, 通过全部 easy/hard 测试: 100%

## 以课程网站为准



# 关于传说中的PA实验

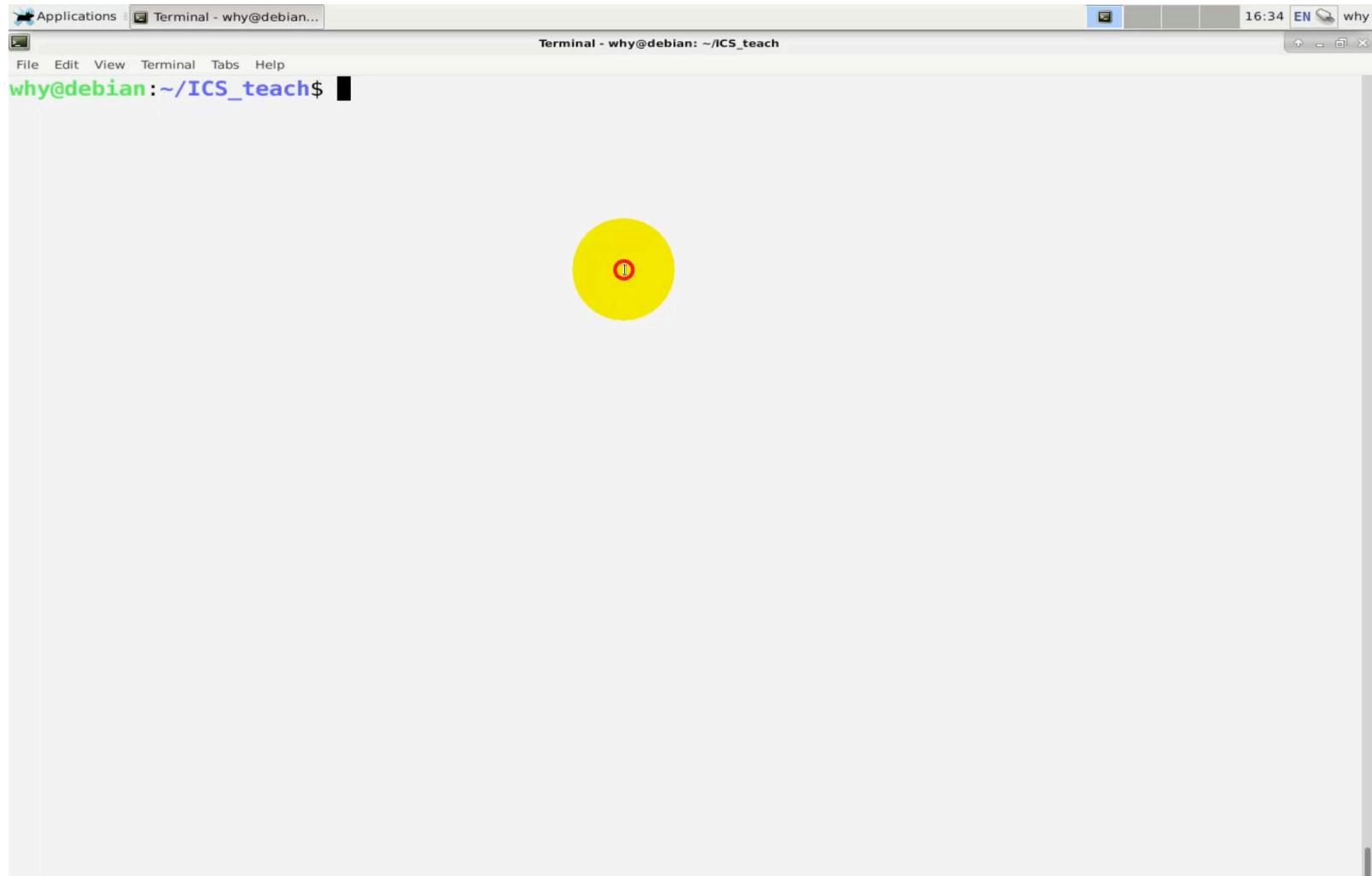
PA是什么东西?

登记TOKEN需求:

<https://table.nju.edu.cn/dtable/links/7331055c81d54c369813>

# 回答终极拷问

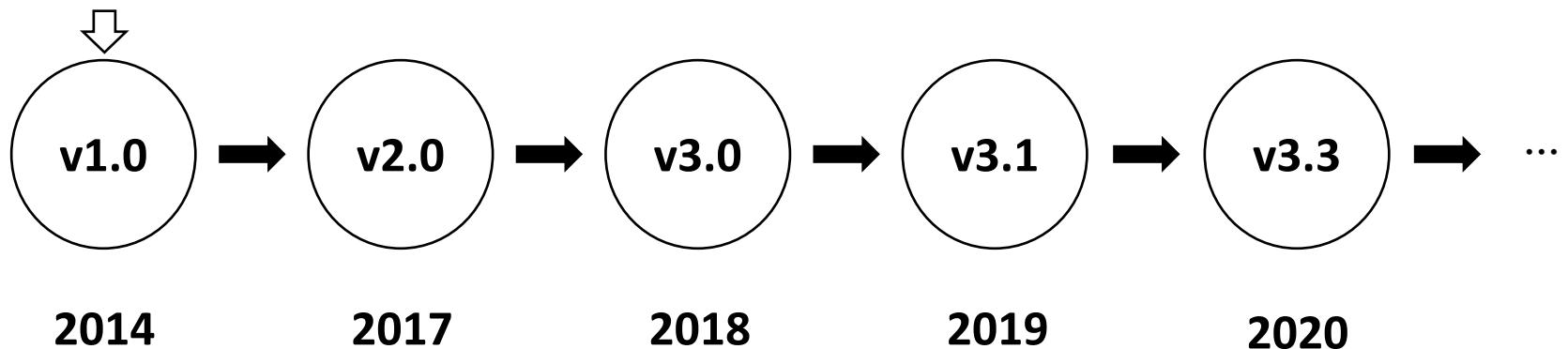
当你运行一个Hello World程序的时候，计算机究竟做了什么？



# PA实验

- 功能完备但简化的模拟器NEMU(NJU EMULATOR)的实现
  - 实验环境配置 (PA0)
  - 四个连贯的实验内容 (PA1~PA4) , 即: 简易调试器、程序执行、cache与存储管理、异常与I/O

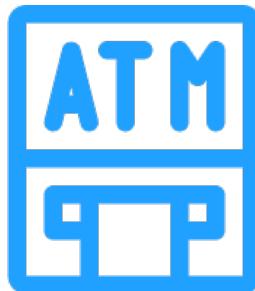
余子濠 开发完成



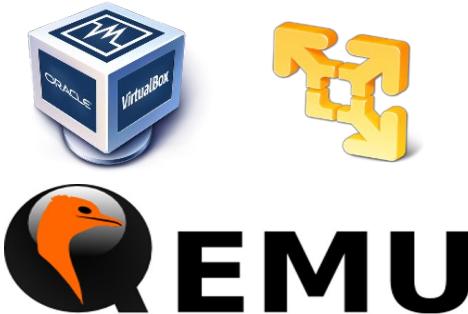
PA的简要历史

# NEMU

- NEMU是一个用来执行其它程序的程序（怎么理解？）
  - 支付宝：用软件模拟硬件ATM
    - 取款、存款、转账、汇款等等



- NEMU：用软件模拟出“计算机”

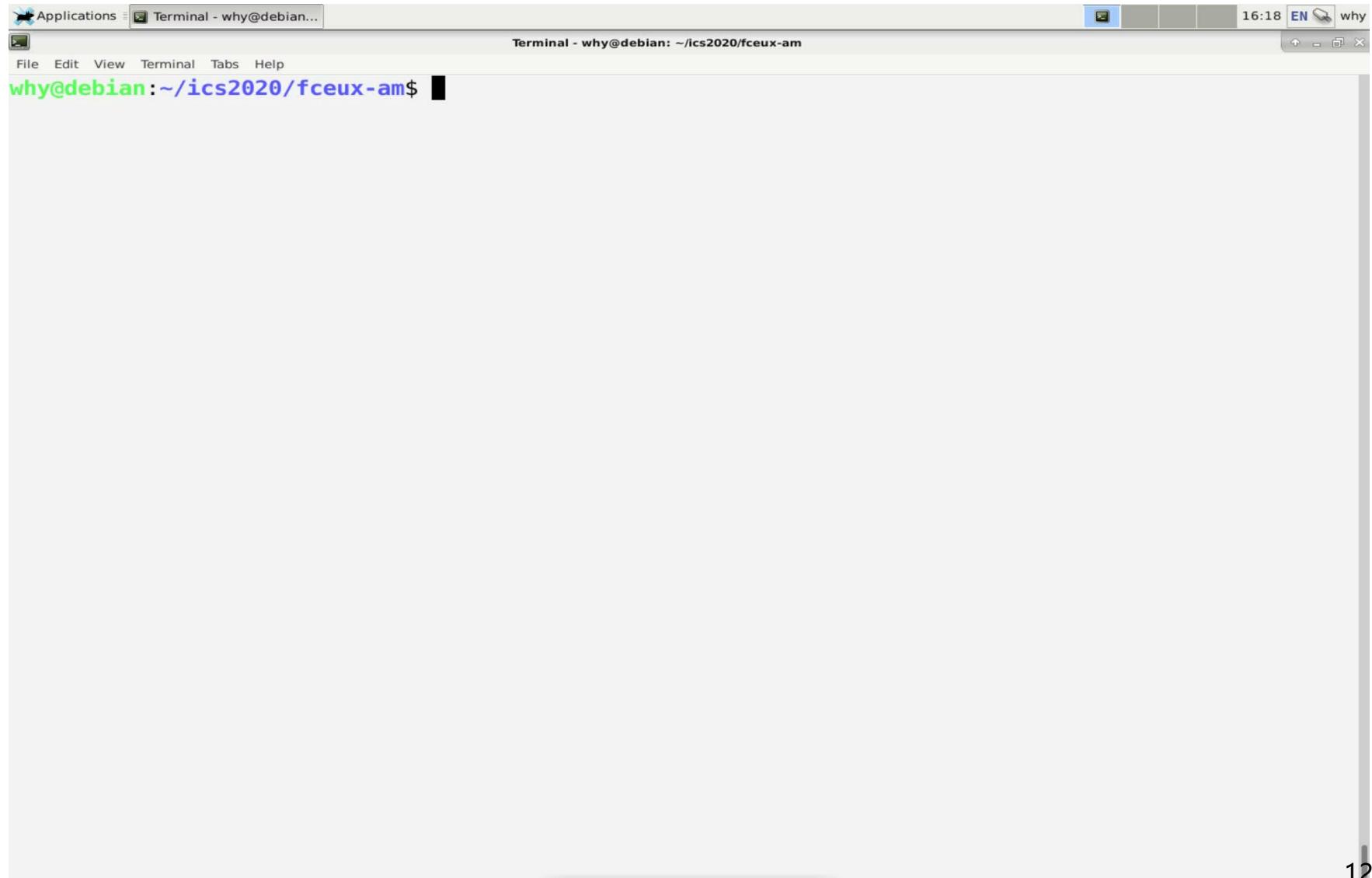


✓ 模拟PC机

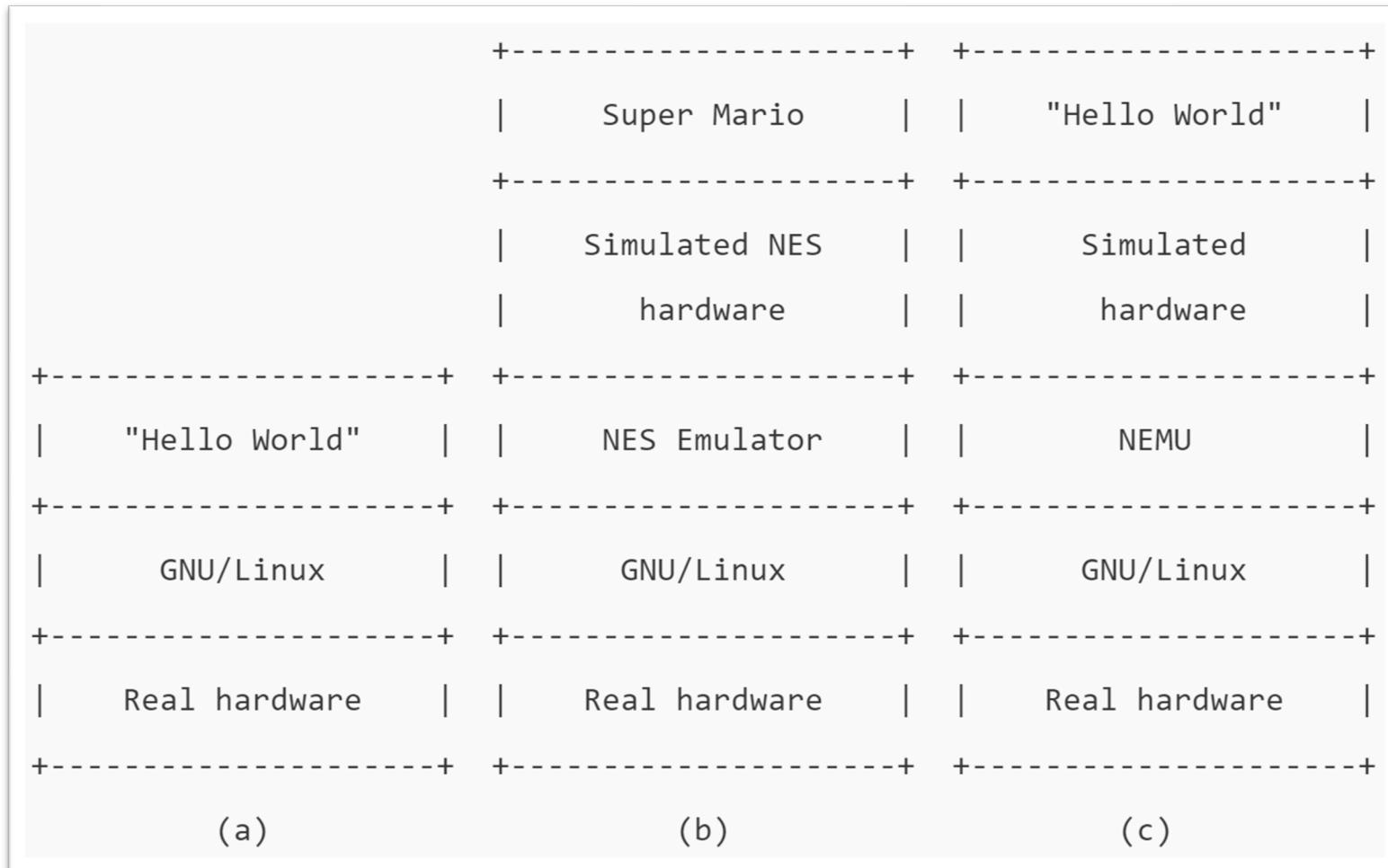
✓ 模拟手机

✓ 模拟游戏机

# 用红白机模拟器NES Emulator玩Mario



# NEMU模拟器



# 学术诚信 (Academic Integrity)

# 学生手册：不能抄作业

- What is academic integrity?
  - [What is Academic Integrity? | Academic Integrity at MIT](#)

Plagiarism	Cheating	
<p><b>Do:</b></p> <p>Trust the instructor.</p> <p>Undertake your own work.</p>	<p><b>Do:</b></p> <p>Demonstrate your own achievement.</p> <p>Accept corrections from the instructor as part of the learning process.</p> <p>Do original work for each class.</p>	<p><b>Don't:</b></p> <p>Don't copy answers from another student; don't ask another student to do your work for you. Don't fabricate results. Don't use electronic or other devices during exams.</p> <p>Don't alter graded exams and submit them for re-grading.</p> <p>Don't submit projects or papers that have been done for a previous class.</p>
<p><b>Unauthorized Collaboration</b></p> <p><b>Do:</b></p> <p>Trust the instructor.</p>	<p><b>Facilitating Academic Dishonesty</b></p> <p><b>Do:</b></p> <p>Showcase your own abilities.</p>	<p><b>Don't:</b></p> <p>Don't allow another student to copy your answers on assignments or exams. Don't take an exam or complete an assignment for another student.</p>

# 学生手册：不能抄作业

---

- What is academic integrity?
  - 简单概况：**独立完成**
- 针对作业的独立完成
  - 自己完成作业
  - 对使用的已有资料作出明确的标示
  - [ACM Policy on Plagiarism, Misrepresentation, and Falsification](#)
- 针对代码的独立完成
  - 自己完成代码的编写
  - 自己完成测试用例
  - 在允许的范围内使用他人的成果

# 具体案例：MIT 6.005 Elements of Software Construction

- Individual work
  - Problem sets in this class are intended to be primarily individual efforts. You are encouraged to discuss approaches with other students but *your code and your write-up must be your own.*
  - You *may not use materials produced as course work by other students*, whether in this term or previous terms, nor may you provide work for other students to use.
  - It's good to help other students. But as a general rule, during the time that you are helping another student, *your own solution should not be visible*, either to you or to them. Make a habit of closing your laptop while you're helping.

# 具体案例：MIT 6.005 Elements of Software Construction (cont'd)

---

- Using external resources
  - It's fine to use material from external sources like [StackOverflow](#), but only with proper attribution, and only if the assignment allows it. In particular, if the assignment says “implement X,” then you must create your own X, not reuse one from an external source.
  - It's also fine to use any code provided by this semester's 6.031 staff (in class, readings, or problem sets), without need for attribution. Staff-provided code may not be publicly shared without permission, however, as discussed later in this document.

# 具体的案例 (1)

- Alyssa and Ben sit next to each other with their laptops while working on a problem set. They talk in general terms about different approaches to doing the problem set. They draw diagrams on the whiteboard. When Alyssa **discovers a useful class in the Java library**, she mentions it to Ben. When Ben finds a StackOverflow answer that helps, he sends the URL to Alyssa. **OK.**
- As they type lines of code, they speak the code aloud to the other person, to make sure both people have the right code. **INAPPROPRIATE.**
- In a tricky part of the problem set, Alyssa and Ben look at each other's screens and compare them so that they can get their code right. **INAPPROPRIATE.**

# 具体的案例 (2)

---

- Jerry already finished the problem set, but his friend Ben is now struggling with a nasty bug. Jerry sits next to Ben, looks at his code, and helps him **debug**. **OK**.
- Jerry opens his own laptop, finds his solution to the problem set, and refers to it while he's helping Ben correct his code. **INAPPROPRIATE**.

# 具体的案例 (3)

- Louis had three problem sets and two quizzes this week, was away from campus for several days for a track meet, and then got sick. Ben feels sorry for Louis and wants to help, so he sits down with Louis and talks with him about how to do the problem set while Louis is working on it. Ben already handed in his own solution, but he doesn't open his own laptop to look at it while he's helping Louis. **OK**.
- Ben opens his laptop and reads his own code while he's helping Louis. **INAPPROPRIATE**.
- Ben has by now spent a couple hours with Louis, and Louis still needs help, but Ben really needs to get back to his own work. He puts his code in a Dropbox and shares it with Louis, after Louis promises only to look at it when he really has to. **INAPPROPRIATE**.

# 具体的案例 (4)

---

- John and Ellen both worked on their problem sets separately. They exchange their test cases with each other to check their work. **INAPPROPRIATE**. Test cases are part of the material for the problem set, and part of the learning experience of the course. You *are copying if you use somebody else's test cases, even if temporarily.*

# 具体的案例：PA0

PA0: 安装 Linux 系统，并提交空的文件。允许向互联网/同学求助。

- 遇到问题（如安装错误）找同学询问/解决 **OK**
  - 但你可能就失去了这门课原本的训练
    - 尽可能先自己解决
    - 帮其他同学解决问题的人
      - 一起还原解决问题的过程
- 请别人安装系统，或使用他人的虚拟机镜像 **INAPPROPRIATE**
  - Arm架构？

# Academic integrity

---

- 感到三观尽毁?
  - 原来拿个测试用例也违反academic integrity?
  - 拿个大腿的作业来改改不香吗? 我还读懂了呢!
  - ~~老师压根就没精力管, 对他来说吃力不讨好~~
    - 有些事情是“天然”**被禁止的**
    - 但是我们的教育里缺失了“这是不对的”
- 那些痛苦是对你的训练 (training)
  - “看懂” 和 “自己设计测试用例、自己做出来” 天差地别
    - PA难度无形剧增←过去没有academic integrity欠的债
  - 不要看不起美国人
    - (大部分) 学生真的明白并自发地执行这个标准

# 知乎贴：作业抄袭中的人生百态



农夫山泉  
NONGFU SPRING

我们不生产水,我们只是大自然的搬运工

代码

万能网



代码抄袭：那些让985学

```
1 void P1() {  
2 ...  
3     puts("Game_Over");  
4 ...  
5 }  
6 void P2() {  
7 ...  
8     puts("G"); puts("a")  
9     puts("e"); puts("_")  
10    puts("v"); puts("e")  
11 ...  
12 }
```

```
1 cur->lineno = temp->lineno;  
2 strcpy(cur->type, type);  
3 cur->isLexical = 0;  
4 cur->children = temp;
```

```
1 $$->is_root=1;  
2 $$->no_leaves=1;  
3 $$->leaves[0]=(Node*)$1;  
4 if(exit_error==0)  
5     {print_tree($$,0);}
```

```
1 temp->line = a->line;  
2 temp->lChild = a;  
3 while(num > 1){  
4     a->rChilds = va_arg(list,  
5                           node*);  
6     a = a->rChilds;  
7     num--;  
8 }
```

```
1 head->number_signal = 0;  
2 head->line = temp->line;  
3 strcpy(head->type,type);  
4 head->child_left = temp;
```

```
1 $$->final=0;  
2 $$->num_children=1;  
3 $$->children=(Node**)malloc  
4     (sizeof(Node*)*$$->  
5      num_children);  
6 $$->children[0]=(Node*)$1;  
7 if(!wrong)  
8     printNode($$,0);
```

```
1 p_node->left_child = temp;  
2 p_node->line = temp->line;  
3 for(int i=0;i < num-1;++){  
4     temp->right_child =  
5         va_arg(valist,struct  
6                   Node*);  
7     temp = temp->right_child;  
8 }
```

知乎 @蒋炎岩



Aims and Objectives of RPg  
Education

Strategic Framework for TPg  
Education

HKUST Fok Ying Tung Graduate  
School

News

抄袭

学术

论文

ICCV

科研热点

关注者

3,449

被浏览

6,642,782

# HKUST attaches great importance to academic integrity

The Hong Kong University of Science and Technology (HKUST) has initiated an investigation in accordance with procedures into an alleged plagiarism incident involving HKUST members.

HKUST attaches great importance to academic integrity. The University has established guidelines on academic integrity and expects all members to abide by the relevant regulations. In case of violation, the

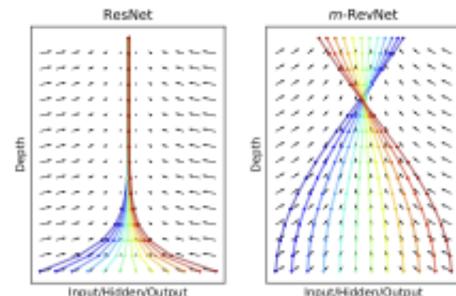
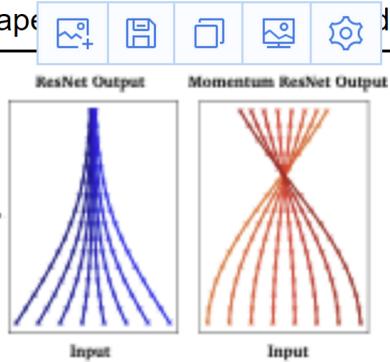
## 如何看待 ICCV21 接收的某港科大学生为一作的论文被指抄袭 ICML21 发表的论文？

ICCV21接收论文m-RevNet: Deep Reversible Neural Networks with Momentum被指出与ICML21接收论文Momentum residual neural networks在核心思路、实验和图表上有多处雷同，疑似抄袭

ICML21论文作者的声明和详细的抄袭证据：

<https://michaelsdr.github.io/momentumnet/plagiarism/>  
[🔗 michaelsdr.github.io/momentumnet/plagiarism/](https://michaelsdr.github.io/momentumnet/plagiarism/)

被指抄袭的论文的一作个人主页网页快照：

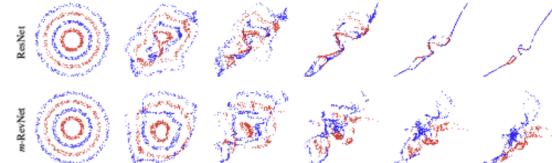
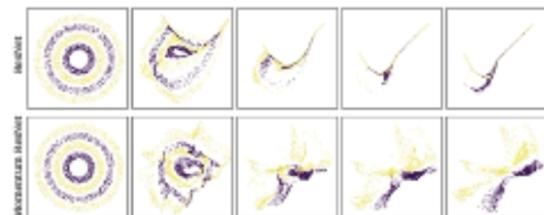


Same figure, added vector field and changed colors.

Table 1. Comparison of reversible residual architectures

	Neur.ODE	<i>i</i> -ResNet	<i>i</i> -RevNet	RevNet	Mom.Net
Closed-form inversion	✓	✗	✓	✓	✓
Same parameters	✗	✓	✗	✗	✓
Unconstrained training	✓	✗	✓	✓	✓

Method	ResNet	RevNet/ <i>i</i> -RevNet	<i>i</i> -ResNet	NODE (and variants)	<i>m</i> -RevNet (ours)
Analytical Reversal	N/A	✓	✗	✗	✓
Architectural Preservation	N/A	✗	✓	✓	✓
End-to-End Optimization	✓	✓	✗	✓	✓

 Table 1. Comparing *m*-RevNet and ResNet to RevNet, *i*-RevNet, *i*-ResNet, NODE (and its variants). Architectural preservation refers to

 Figure 2. Comparison of learning dynamics between ResNet (top) and *m*-RevNet (bottom) in a 2D example. The leftmost distribution of

The three rows have the same meaning: closed form inversion = analytical reversal, same parameters = architecture preservation and unconstrained training = end-to-end optimization. The columns are shuffled, and Paper B added the ResNet baseline.

Choice of the exact same initial dataset, four nested rings. The usual experiments as found e.g. in <https://arxiv.org/abs/1904.01681> as cited by both papers rather use 2 nested rings. To the best of our knowledge, using four nested rings as an illustration had never been done before.

Installing GNU/Linux

First Exploration with GNU/Linux

Installing Tools

Configuring vim

[More Exploration](#)

Getting Source Code for PAs

# PA0 选讲：进入Linux世界

学会用Linux工作吧

# 让时间回到1980s

- 就算回到1980s，该干的事情还得干啊
  - 管理文件；编代码；写作业；排版杂志；上网.....

- 例子

- vi a.txt
- ip addr / ping baidu.com
- df -h /
- find . -name "\*.cpp"
- fdisk /dev/sdb
- shutdown -h 0
- apt install qemu-system
- pdfjoin a.pdf b.pdf
- iconv -f gbk -t utf-8 file.txt



# 现在都2023年了，还整这些玩意干嘛？

上面这些事情不是点点鼠标就能搞定的吗？

- 被迫接受
  - ~~不学你就挂了~~
- 主动接受
  - 对系统更强的控制力
    - 应用程序通常无法满足 power user 的全部需求
  - 生产系统编程的事实标准
    - Linux, macOS, Windows, ...运维基本都靠命令行工具
  - 来自开源社区的一份礼物
    - 非常丰富、可定制、看得见源码的软件栈
- 流传自远古时代的OS实验课程网站中的Linux入门教程

# Linux命令行概述

# 第一课

- 这是个啥？让我用这个度过余生？

\$ █

Unix is user friendly. It's just selective about who its friends are.

Read the manual.

Search the web.

# 为什么大家在一开始都感到困难？

- 1980s: 以 MHz 为单位的主频；80 x 24 的字符终端
  - 无法提供丰富的交互界面
  - 但依然要完成各类任务
  - 你会如何设计？

不可避免，需要用户查阅 **手册** 记住一些系统里的约定  
(下面是最重要的一些)。

- 目录结构、文件命名规律、访问权限
- 命令执行的约定
- 常用命令行工具
- Shell 编程语言

# 常见的命令行工具

```
missing:~$ date  
Fri 10 Jan 2020 11:49:31 AM EST  
missing:~$
```

```
missing:~$ ls  
missing:~$ cd ..  
missing:/home$ ls  
missing  
missing:/home$ cd ..  
missing:/$ ls  
bin  
boot
```

```
missing:~$ ls -l /home  
drwxr-xr-x 1 missing users 4096 Jun 15 2019 missing
```

• • •

# 常见的命令行工具

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
missing:/$ cd ./home
missing:/home$ pwd
/home
missing:/home$ cd missing
missing:~$ pwd
/home/missing
missing:~$ ../../bin/echo hello
hello
```

# 常见的命令行工具

```
missing:~$ echo hello  
hello
```

```
missing:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
missing:~$ which echo  
/bin/echo  
missing:~$ /bin/echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
missing:~$ echo hello > hello.txt  
missing:~$ cat hello.txt  
hello  
missing:~$ cat < hello.txt  
hello  
missing:~$ cat < hello.txt > hello2.txt  
missing:~$ cat hello2.txt  
hello
```

# 常见的命令行工具

- 查看文件
  - ls (list), cd (change directory), find, tree, ...
- 阅读手册
  - man (man man); apropos; info
- 文本处理
  - cat (concatenate), wc (word count), grep (g/re/p)
  - tr (translate), cut, awk, sed (stream editor)

命令行工具多到什么程度呢.....

- unzip, strip, touch, finger, grep, mount, fsck, more, yes, umount, sleep.....

# UNIX哲学

- *Keep it simple, stupid.* (KISS)
  - *Everything* is a file and *pipeline* programs to work together
- 一个命令只做 “一件事”
  - 从stdin输入 (printf)
  - 向stdout输出 (scanf)
  - 使用参数控制行为 (int main (int argc, char \* argv[]);)
- 命令的输入和输出都是人类 + 机器均可读的文本
  - find .
  - wc -l a.txt b.txt
- 把命令的输入/输出连接起来 (管道) 协作完成任务
  - find . | grep '\.cpp\$' | xargs cat | wc -l

# The Shell Scripting Language

祝贺！刚才其实是Shell执行了一段Shell语言编写的程序。

- Shell 是一门基于**字符串**和**命令**的编程语言
  - a=hello - 赋值 (注意 = 左右没有空格)
  - \$a - 将变量的值 “粘贴”
  - \$(cmd) - 将 cmd 运行的 stdout “粘贴”
  - if cmd; then; ... ; fi - 根据 cmd 运行结果执行分支
  - cmd > file - 把 cmd 的 stdout 重定向到 file
  - cmd1 | cmd2 - 把 cmd1 的 stdout 作为 cmd2 的 stdin
- 有趣的小问题
  - 如何用 if 比较存储了整数字符串的大小？
  - (if 1 > 2 会发生什么？ )

# 开始编程吧！

---

- 输出当前用户是不是root

- [ \$UID -eq 0 ] && echo “is root!”

- 查看磁盘引导扇区 (Master Boot Record)

- cat /dev/sdb | head -c 512 | ndisasm -b 16 -o 0x7c00 -

- 统计所有cpp文件的行数

- find . | grep '\.cpp\$' | xargs cat | wc -l

- 统计命令行命令的频率

- history | tr -s ' ' | cut -d ' ' -f3 | sort | uniq -c | sort -nr

- 以上都是命令行里的命令，但同时也是 bash script 的程序
  - (这就是程序员的宿命吧)

# 自动化工具：程序员福利系列

- 重构福利
  - (ex) 对一个目录里的所有.cpp文件执行同样的vim动作
- LaTeX用户福利
  - (pdfcrop) 把LibreOffice的演示文稿导出为PDF，然后将每一页的白边部分裁去，分别命名为fig-1.pdf, fig-2.pdf, ...
- 视频制作福利
  - (ffmpeg) 为视频添加水印和字幕

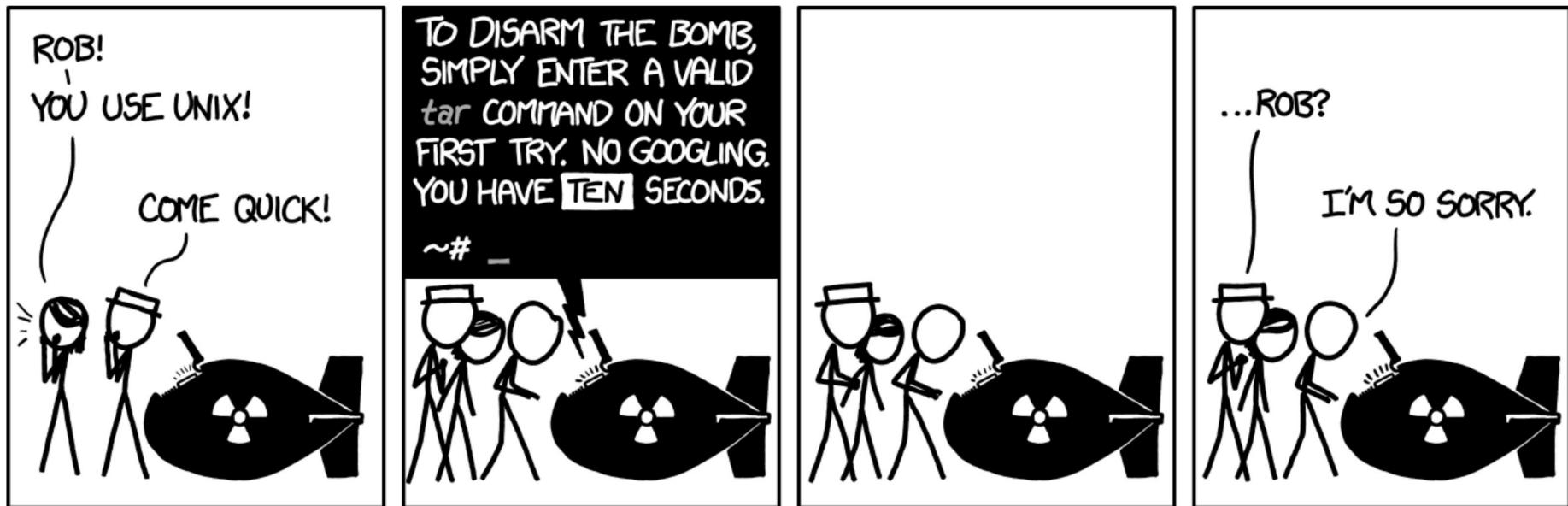
# 生存指南

# 拥抱变化

---

- 很烦躁：一下就碰壁？
  - 胡乱尝试一通？对了就对了，不对就抓瞎？
  - 有时候不知道该到底怎么STFW, RTFW?
- 静下心来，从头开始
  - [The Missing Semester of Your CS Education](#)
  - [流传自远古时代的OS实验课程网站中的Linux入门教程](#)
  - RTFM (slides), STFW, RTFSC
    - 不要觉得Makefile、提交脚本都是和你无关的
- 试图理解一切事情是如何发生的
  - 从读得懂开始，逐渐过渡到会写

# 现代化你的命令行工具



- 例子：

- tldr (替代man)
- zsh-z (替代pushd/popd)
- fzf
- vim/vscode

# 拥抱开源社区

---

- 用好 Github 的 “[Awesome](#)” 系列
  - 例如[The art of command line](#)
- 用好 Stack Overflow / Stack Exchange
- 禁用百度和中文关键字（强烈不推荐大家使用中文系统）
  - Linux/macOS
    - /etc/hosts
  - Windows
    - C:\Windows\System32\drivers\etc\hosts
  - 增加一行127.0.0.1 www.baidu.com

选讲PA0

# 总结

# 本次课程最重要的内容

---

- 静下心来，从头开始
  - [The Missing Semester of Your CS Education](#)
  - RTFM (slides), STFW, RTFSC
- 用好 Github的 “[Awesome](#)” 系列
  - 例如[The art of command line](#)
- 用好 Stack Overflow / Stack Exchange
- 禁用百度和中文关键字

# 总结

- 你不会感到学习这门课很舒适
  - 不要用 “我们学得比较理论……” 来骗自己
    - 就是不扎实
  - Academic integrity 可能让你感到三观尽毁
    - 你会理解到南大还不是 “世界一流大学”
- 但请不要放弃/躺倒
- 你们未来是要承担大事业的

[PAO](#)已发布，暂时未上线OJ，请关注群通知

# 一些福利建议

---

- 用好Git
- 关注DDL

机器永远是对的 (and RTFM)

RTFM: Read The Friendly Manual

STFW: Search The Friendly Web

# 自动化工具：程序员福利系列 (cont'd)

```
#自动登陆p.nju.edu.cn不香吗?  
curl -d "username=学号&password=密码" \  
http://p.nju.edu.cn/portal_io/login
```

- 这有什么用？

- 校园网内某台机器的长久连接 (brasd)
- 作为你的校内代理服务器
- 实现内网穿透.....

- 一个有趣的问题

- 密码是明文，被舍友偷窥了怎么办？？？
- 文件系统有权管理：chmod -r
  - 但是shell script必须读权限？

# C 语言拾遗(1): 机制

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 讲前提醒

PA0即将截止：

- \* 2023年9月24日23:59:59 (以此 deadline 计按时提交bonus)

PA1已悄悄发布： (OJ暂未开启PA1评测)

- \* PA 1.1: 2023.9.30 (此为建议的不计分 deadline)
- \* PA 1.2: 2023.10.5 (此为建议的不计分 deadline)
- \* PA 1.3: 2023.10.15 23:59:59 (以此 deadline 计按时提交bonus)

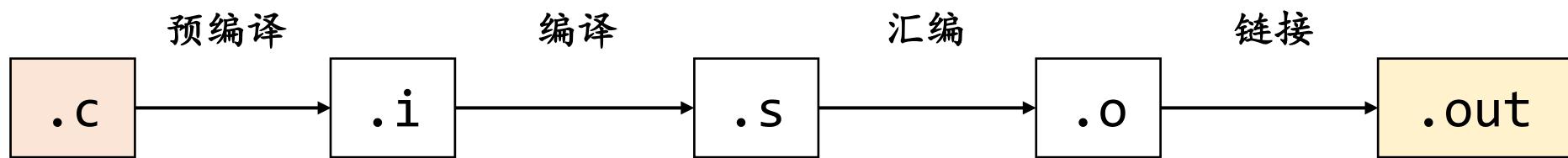
# 本讲概述

- 在IDE里，为什么按一个键，就能够编译运行？
  - 编译、链接
    - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
  - 加载执行
    - ./a.out
- 背后是通过调用命令行工具完成的
  - RTFM: man gcc; gcc -help; tldr gcc
    - 控制行为的三个选项: -E, -S, -c
- 本次课程
  - 预热：编译、链接、加载到底做了什么？
  - RTFSC时需要关注的C语言特性

# IDE的一个键到底发生了什么？



# IDE的一个键到底发生了什么？





Applications Terminal - why@debian...

18:22 EN why



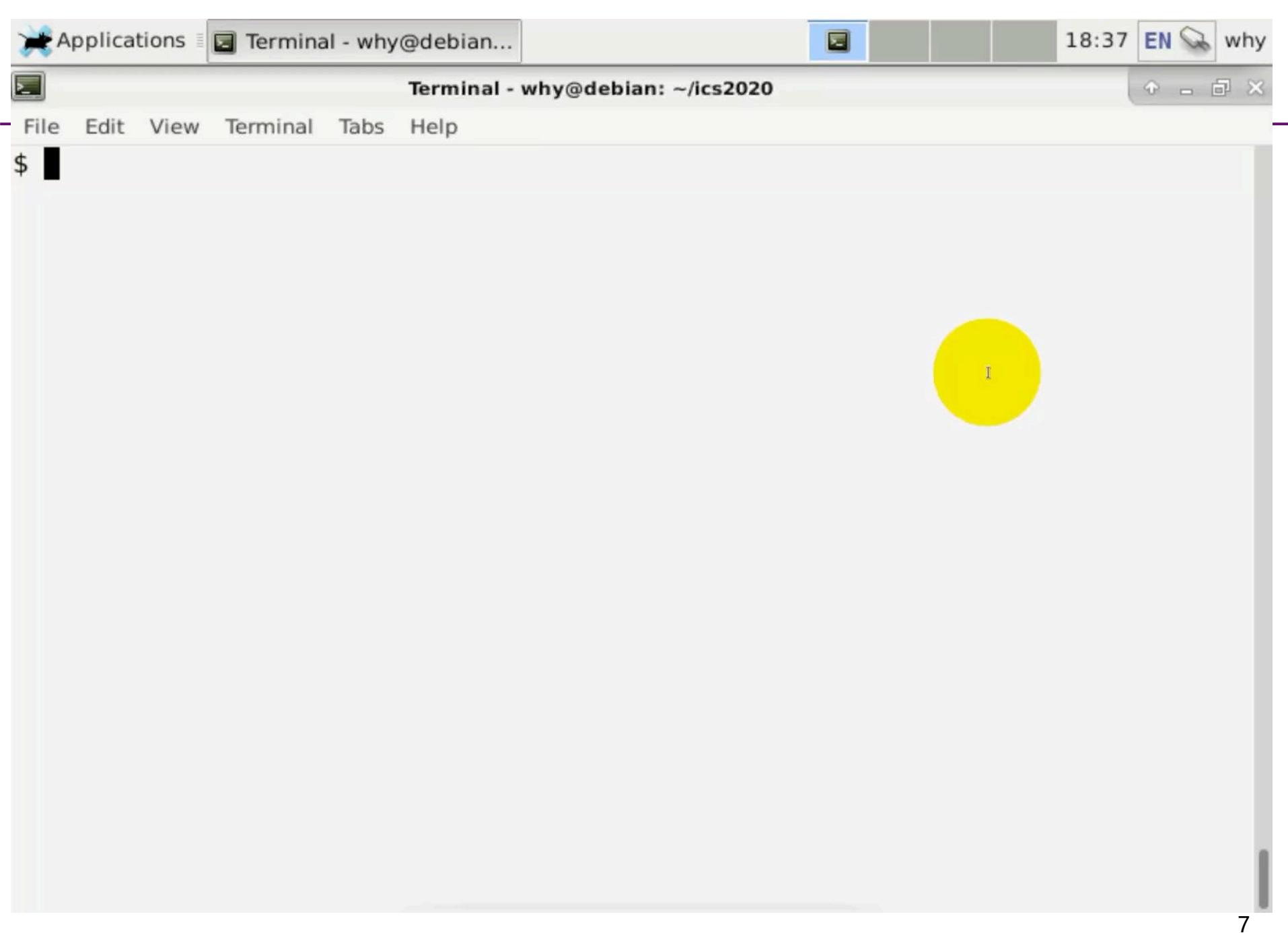
Terminal - why@debian: ~/ics2020



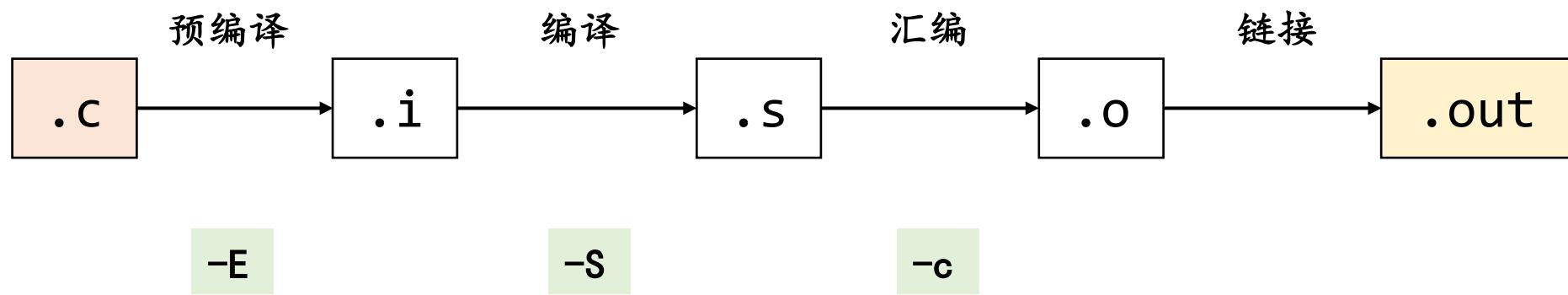
File Edit View Terminal Tabs Help

\$





# 从源代码到可执行文件



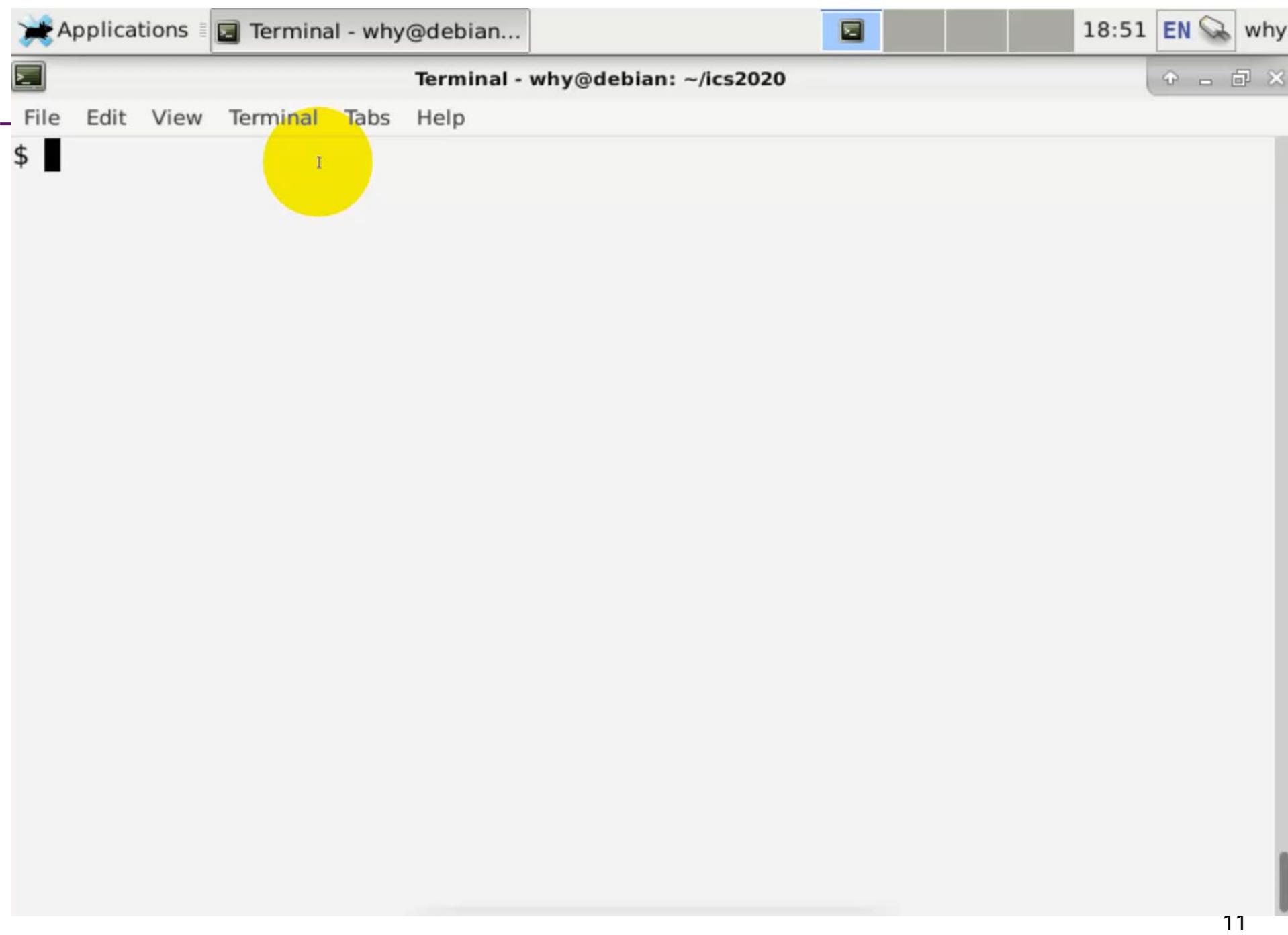
# 进入C语言之前：预编译

```
#include  
#define  
#  
##  
#ifdef
```

# #include指令

---

- 什么是#include?
- 怎么理解?



# #include<>指令

- 以下代码有什么区别？

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

- 为什么在没有安装库时会发生错误？

```
#include <SDL/SDL2.h>
```

- 你可能在书/阅读材料上了解过一些相关的知识
  - 但更好的办法是**阅读命令的日志**
  - gcc --verbose a.c

Applications Terminal - why@debian... 18:55 EN why

Terminal - why@debian: ~/ics2020

File Edit View Terminal Tabs Help

```
$ ls
a.c a.inc a.out a.s b
$ █
```

# #include<>指令

- 以下代码有什么区别？

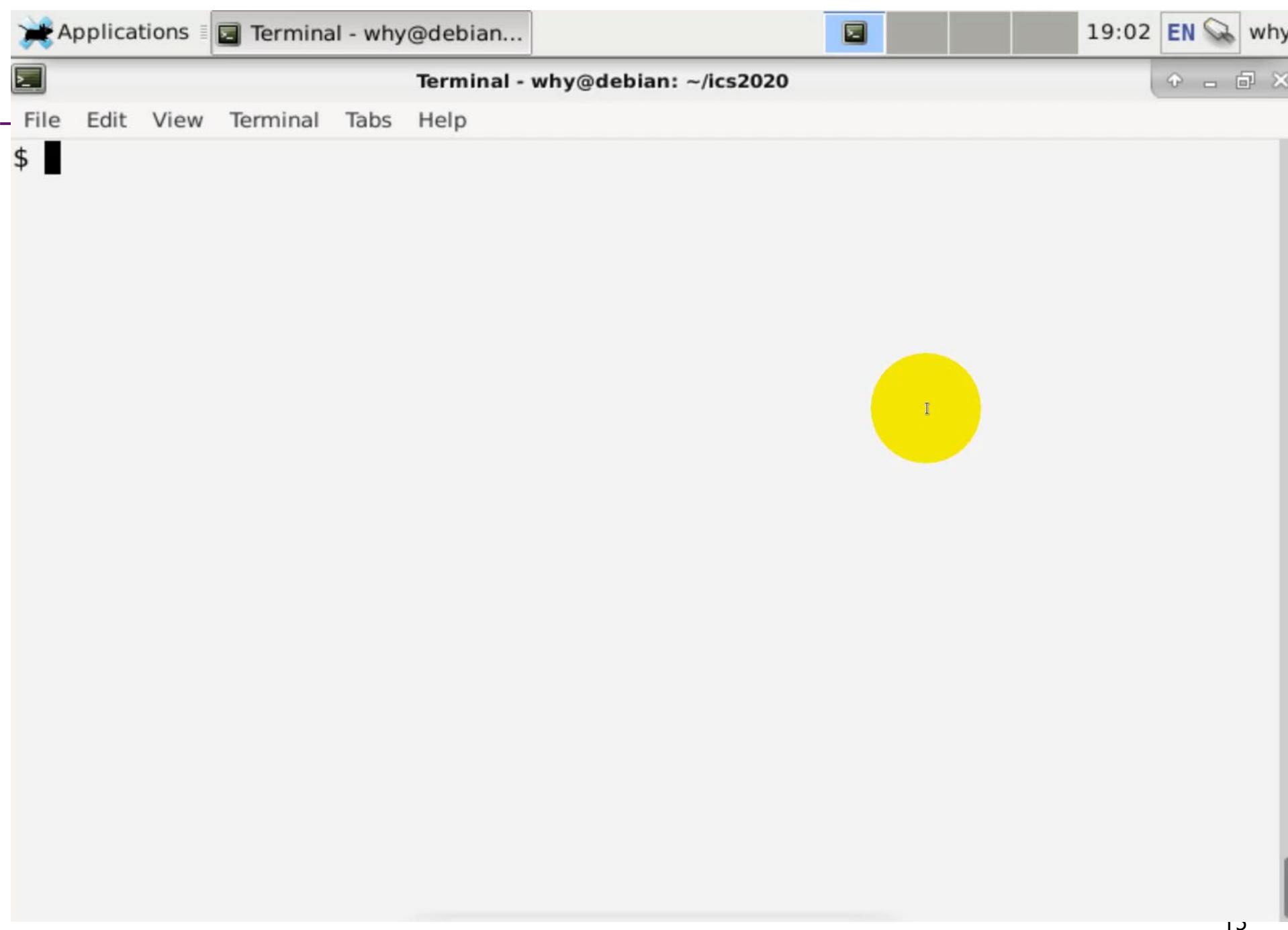
```
#include <stdio.h>
#include "stdio.h"
```

- 为什么在没有安装库时会发生错误？

```
#include <SDL/SDL2.h>
```

- 你可能在书/阅读材料上了解过一些相关的知识
  - 但更好的办法是**阅读命令的日志**
  - gcc --verbose a.c

```
#include "..." search starts here:
#include <...> search starts here:
/usr/lib/gcc/x86_64-linux-gnu/8/include
/usr/local/include
/usr/lib/gcc/x86_64-linux-gnu/8/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
```

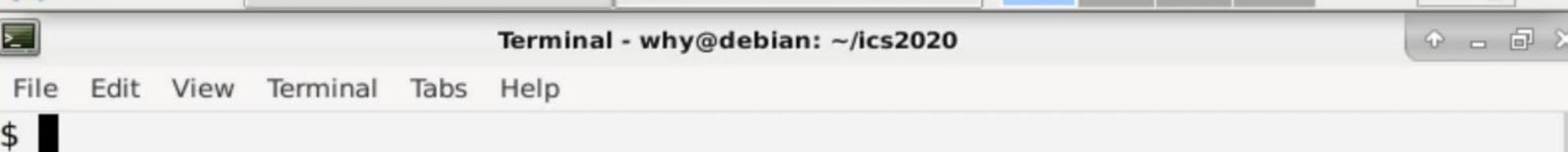
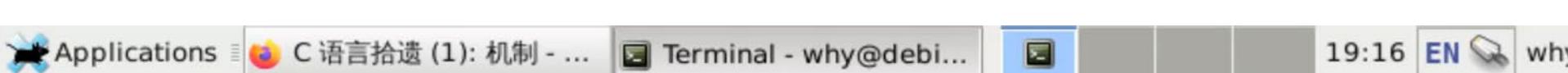


# 有趣的预编译

- 以下代码会输出什么?
  - 为什么?

```
#include <stdio.h>

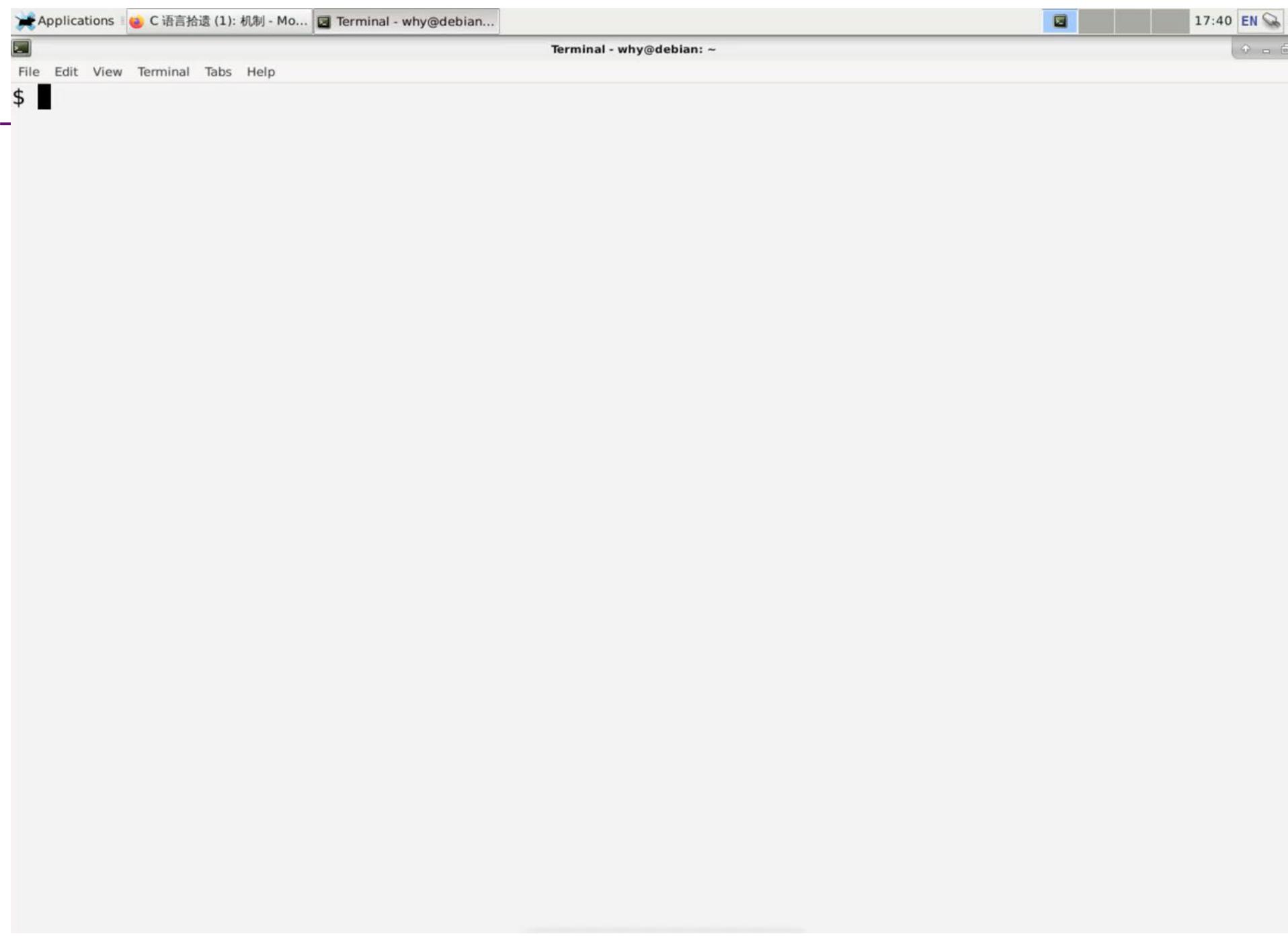
int main() {
#if aa == bb
    printf("Yes\n");
#else
    printf("No\n");
#endif
}
```



# 宏定义与展开

- 宏展开：通过**复制/粘贴**改变代码的形态
  - #include →粘贴文件
  - aa, bb →粘贴符号
- 知乎问题：如何搞垮一个OJ？

```
#define A "aaaaaaaaaa"  
#define TEN(A) A A A A A A A A A A  
#define B TEN(A)  
#define C TEN(B)  
#define D TEN(C)  
#define E TEN(D)  
#define F TEN(E)  
#define G TEN(F)  
int main() { puts(G); }
```



# 宏定义与展开

- 如何躲过Online Judge的关键字过滤?

```
#define SYSTEM sys ## tem
```

```
File Edit View Terminal Tabs Help
1 #define A sys ## tem
2
3 int main(){
4     A("echo Hello\n");
5 }
```

```
$ ./a.out
Hello
```

# 宏定义与展开

- 如何毁掉一个身边的同学?

```
#define true (_LINE_ % 2 != 0)
```

```
File Edit View Terminal Tabs Help
1 #define true (_LINE_ % 2 != 0)
2
3 #include <cstdio>
4
5 int main(){
6     if (true) printf("yes %d\n", _LINE_);
7     if (true) printf("yes %d\n", _LINE_);
8     if (true) printf("yes %d\n", _LINE_);
9     if (true) printf("yes %d\n", _LINE_);
10    if (true) printf("yes %d\n", _LINE_);
11    if (true) printf("yes %d\n", _LINE_);
12    if (true) printf("yes %d\n", _LINE_);
13    if (true) printf("yes %d\n", _LINE_);
14    if (true) printf("yes %d\n", _LINE_);
15    if (true) printf("yes %d\n", _LINE_);
16    if (true) printf("yes %d\n", _LINE_);
17    if (true) printf("yes %d\n", _LINE_);
18    if (true) printf("yes %d\n", _LINE_);
19    if (true) printf("yes %d\n", _LINE_);
20    if (true) printf("yes %d\n", _LINE_);
21 }
```

```
$ g++ a.c
$ ./a.out
yes 7
yes 9
yes 11
yes 13
yes 15
yes 17
yes 19
```

# 宏定义与展开

```
#define s ((((((((((((( 0
#define _ * 2)
#define X * 2 + 1)
static unsigned short stopwatch[ ] = {
    s _ _ _ _ _ X X X X X _ _ _ X X _ ,
    s _ _ _ X X X X X X X X _ X X X ,
    s _ _ X X X _ _ _ _ _ X X X _ X X ,
    s _ X X _ _ _ _ _ _ _ _ X X _ _ ,
    s X X _ _ _ _ _ _ _ _ _ X X _ ,
    s X X _ X X X X X _ _ _ _ X X _ ,
    s X X _ _ _ _ _ X _ _ _ _ X X _ ,
    s X X _ _ _ _ _ X _ _ _ _ X X _ ,
    s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
    s _ _ X X X _ _ _ _ X X X _ _ _ ,
    s _ _ _ X X X X X X X X _ _ _ ,
    s _ _ _ _ X X X X X X X X _ _ _ ,};
```

# 宏定义与展开

gcc -E a.c

```
#define s (((((((((((( 0
#define _ * 2)
#define X * 2 + 1)
static unsigned short stopwatch[] = {
    S _ _ _ _ _ X X X X X _ _ _ X X _ ,
    S _ _ _ X X X X X X X X X _ X X X ,
    S _ _ X X X _ _ _ _ _ X X X _ X X ,
    S _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
    S X X _ _ _ _ _ _ _ _ _ _ _ X X _ ,
    S X X _ X X X X X _ _ _ _ _ X X _ ,
    S X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
    S X X _ _ _ _ _ X _ _ _ _ _ X X _ ,
    S _ X X _ _ _ _ X _ _ _ _ _ X X _ _ ,
    S _ _ X X X _ _ _ _ _ X X X _ _ _ ,
    S _ _ _ X X X X X X X X X _ _ _ _ ,
    S _ _ _ _ X X X X X X X X X _ _ _ _ ,
    S _ _ _ _ _ X X X X X X X X X _ _ _ _ }
```

```
$ ./a.out  
1990  
8183  
14395  
24588  
49158  
57094  
49414  
49414  
24844  
14392  
8176  
1984
```

# X-Macros

- 宏展开：通过**复制/粘贴**改变代码的形态
  - 反复粘贴，直到没有宏可以展开为止

- 例子：X-macro

```
#define NAMES(X) \
    X(Tom) X(Jerry) X(Tyke) X(Spike)
```

```
int main() {
    #define PRINT(x) puts("Hello, " #x "!");
    NAMES(PRINT)
}
```

```
$ ./a.out
Hello, Tom!
Hello, Jerry!
Hello, Tyke!
Hello, Spike!
```

PRINT(TOM) PRINT(Jerry) PRINT(Tyke) PRINT(Spike)

# X-Macros

```
#define NAMES(X) \
    X(Tom) X(Jerry) X(Tyke) X(Spike)
```

```
int main() {
    #define PRINT(x) puts("Hello, " #x "!");
    NAMES(PRINT)
    #define PRINT2(x) puts("Goodbye, " #x "!");
    NAMES(PRINT2)
}
```

PRINT(TOM) PRINT(Jerry) PRINT(Tyke) PRINT(Spike)

PRINT2(TOM) PRINT2(Jerry) PRINT2(Tyke) PRINT2(Spike)

```
$ ./a.out
Hello, Tom!
Hello, Jerry!
Hello, Tyke!
Hello, Spike!
Goodbye, Tom!
Goodbye, Jerry!
Goodbye, Tyke!
Goodbye, Spike!
```

# 有趣的预编译

- 发生在实际编译之前
  - 也称为元编程 (meta-programming)
    - Gcc的预处理器同样可以处理汇编代码
    - C++中的模板元编程；Rust中的macros； ...
- Pros
  - 提供灵活的用法 (X-macros)
  - 接近自然语言的写法
- Cons
  - 破坏可读性IOCCC、程序分析（补全）……

```
#define L (  
int main L ) { puts L "Hello, World" ); }
```

# 编译与链接

(先行剧透本学期的主要内容)

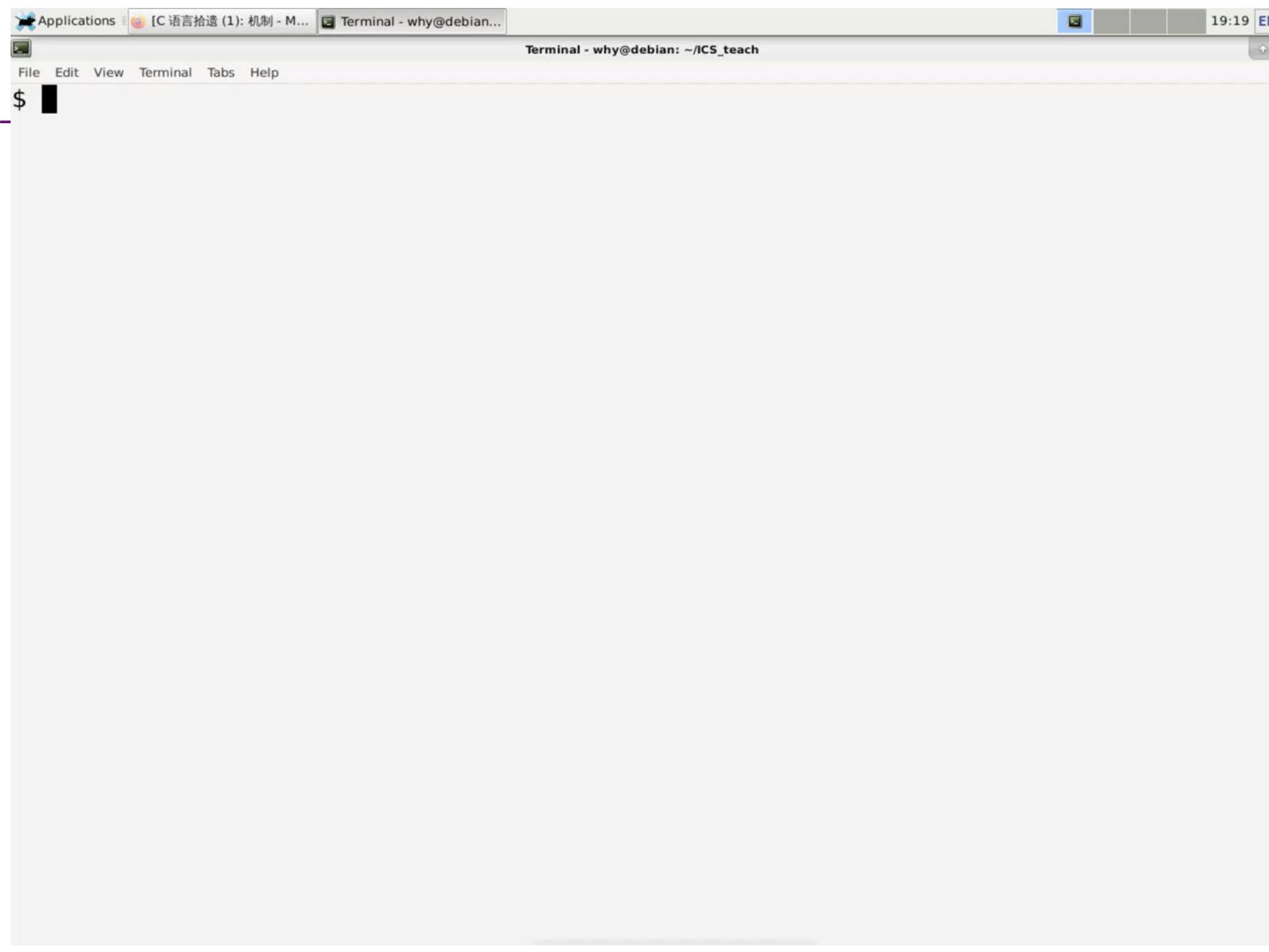
编译、链接

.c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → .out

# 编译

- 一个不带优化的简易（理想）的编译器
  - C代码中的连续一段总能找到对应的一段连续的机器指令
    - 这就是为什么大家会觉得C是高级的汇编语言！

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```



# 编译

- 一个不带优化的简易（理想）的编译器
  - C代码中的连续一段总能找到对应的一段连续的机器指令
    - 这就是为什么大家会觉得C是高级的汇编语言！

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

# a.c 到 a.s 到 a.o

a.s

```
1 foo:  
2     n = ARG-1  
3     sum = 0  
4     i = 1  
5     goto    .L2  
6 .L3:  
7     tmp = i  
8     sum += tmp  
9     i += 1  
10 .L2:  
11    tmp = i  
12    compare (n, tmp)  
13    if(<=) goto .L3  
14    RETURN-VAL = sum  
15  
16    ret
```

a.c

```
1 int foo(int n) {  
2     int sum = 0;  
3     for (int i = 1; i <= n; i++) {  
4         sum += i;  
5     }  
6 }
```

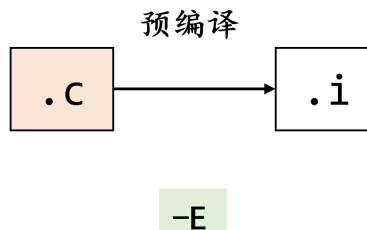
```
$ gcc -c a.c  
$ objdump -d a.o
```

```
a.o:      file format elf64-x86-64
```

Disassembly of section .text:

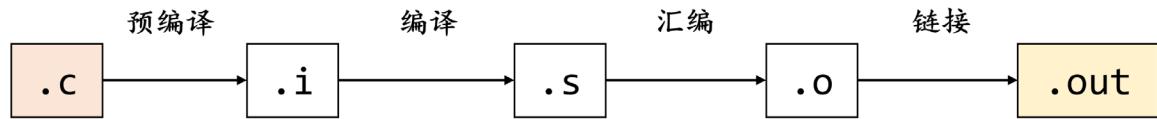
```
0000000000000000 <foo>:  
    0: 55                      push  %rbp  
    1: 48 89 e5                mov   %rsp,%rbp  
    4: 89 7d ec                mov   %edi,-0x14(%rbp)  
    7: c7 45 fc 00 00 00 00    movl  $0x0,-0x4(%rbp)  
    e: c7 45 f8 01 00 00 00    movl  $0x1,-0x8(%rbp)  
   15: eb 0a                  jmp   21 <foo+0x21>  
   17: 8b 45 f8                mov   -0x8(%rbp),%eax  
   1a: 01 45 fc                add   %eax,-0x4(%rbp)  
   1d: 83 45 f8 01             addl  $0x1,-0x8(%rbp)  
   21: 8b 45 f8                mov   -0x8(%rbp),%eax  
   24: 3b 45 ec                cmp   -0x14(%rbp),%eax  
   27: 7e ee                  jle   17 <foo+0x17>  
   29: 8b 45 fc                mov   -0x4(%rbp),%eax  
   2c: 5d                      pop   %rbp  
   2d: c3                      retq
```

a.o



没有main还不能运行

# 链接



- 将多个二进制目标代码拼接在一起
  - C中称为编译单元 (compilation unit)

a.c

```
int foo(int n) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

b.c

```
#include <stdio.h>  
int foo(int n);  
int main(){  
    printf("%d\n"),foo(100));  
}
```

gcc -c a.c

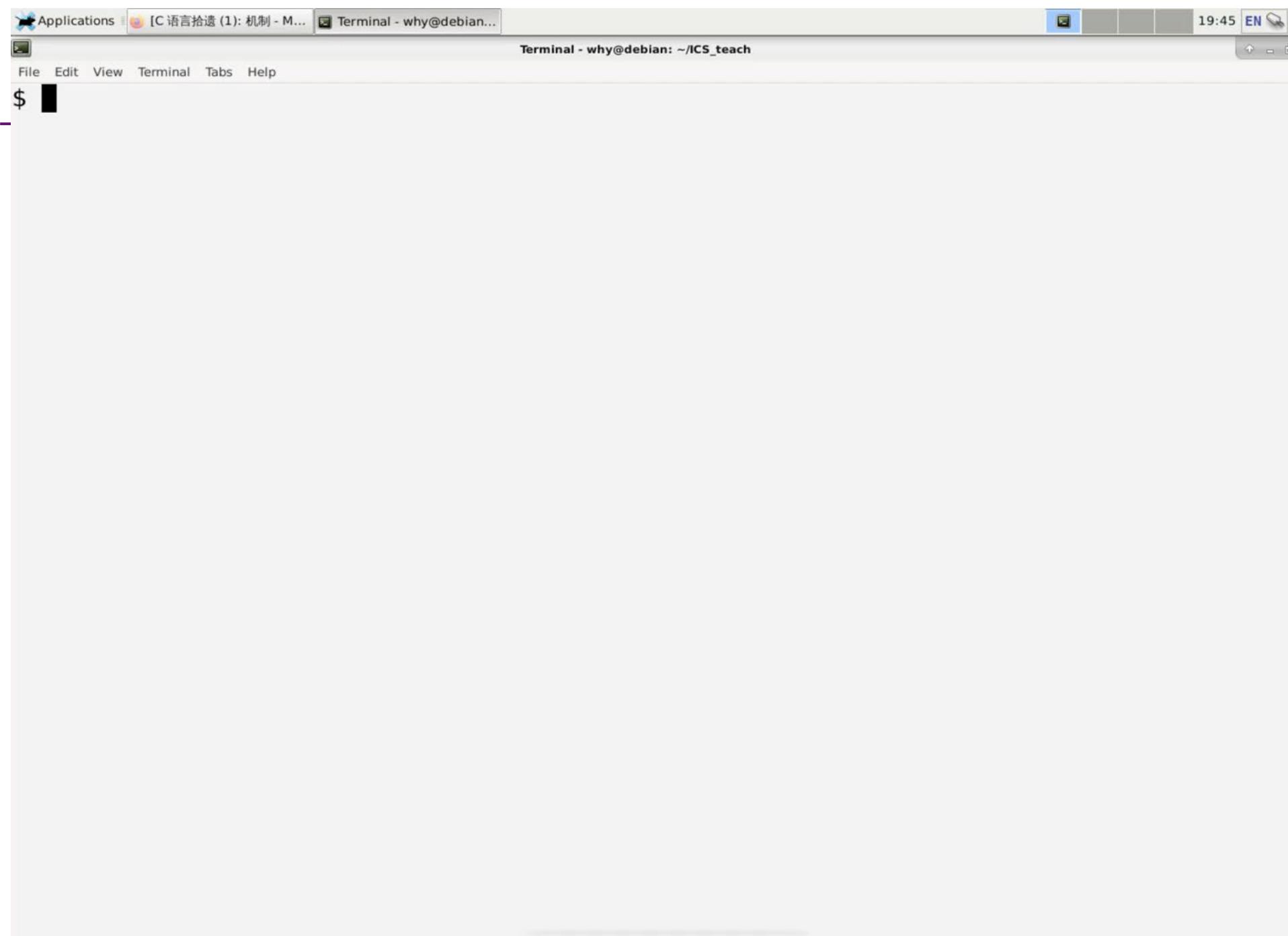
gcc -c b.c

链接

a.o

b.o

?



# 链接

- 将多个二进制目标代码拼接在一起
  - C中称为编译单元 (compilation unit)
  - 甚至可以链接C++, rust, ...代码

```
extern "C" {  
    int foo() { return 0; }  
}
```

```
int bar() { return 0; }
```

```
$ g++ -c a.cc  
$ objdump -d a.o
```

```
a.o:      file format elf64-x86-64
```

```
Disassembly of section .text:
```

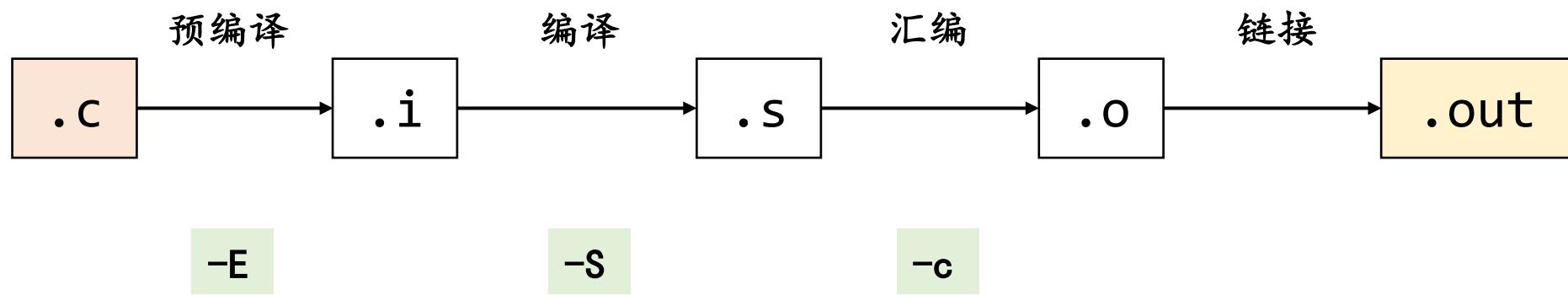
```
0000000000000000 <foo>:
```

0:	55	push %rbp
1:	48 89 e5	mov %rsp,%rbp
4:	b8 00 00 00 00	mov \$0x0,%eax
9:	5d	pop %rbp
a:	c3	retq

```
000000000000000b <_Z3barv>:
```

b:	55	push %rbp
c:	48 89 e5	mov %rsp,%rbp
f:	b8 00 00 00 00	mov \$0x0,%eax
14:	5d	pop %rbp
15:	c3	retq

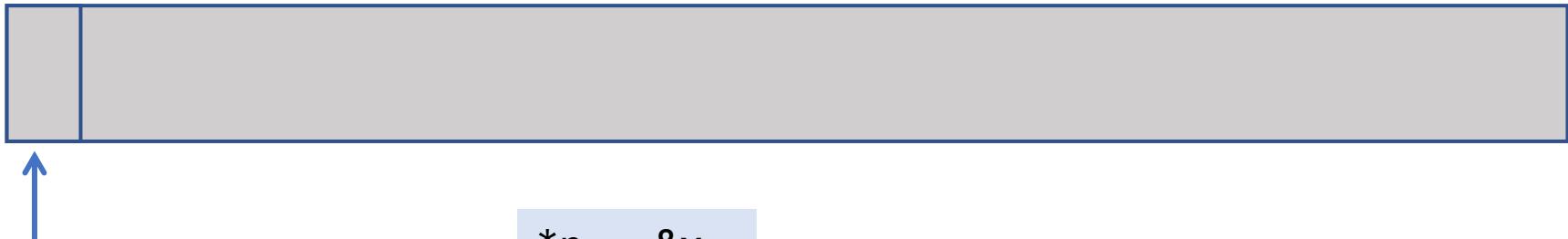
# 从源代码到可执行文件



# 加载：进入C语言的世界

# C程序执行的两个视角

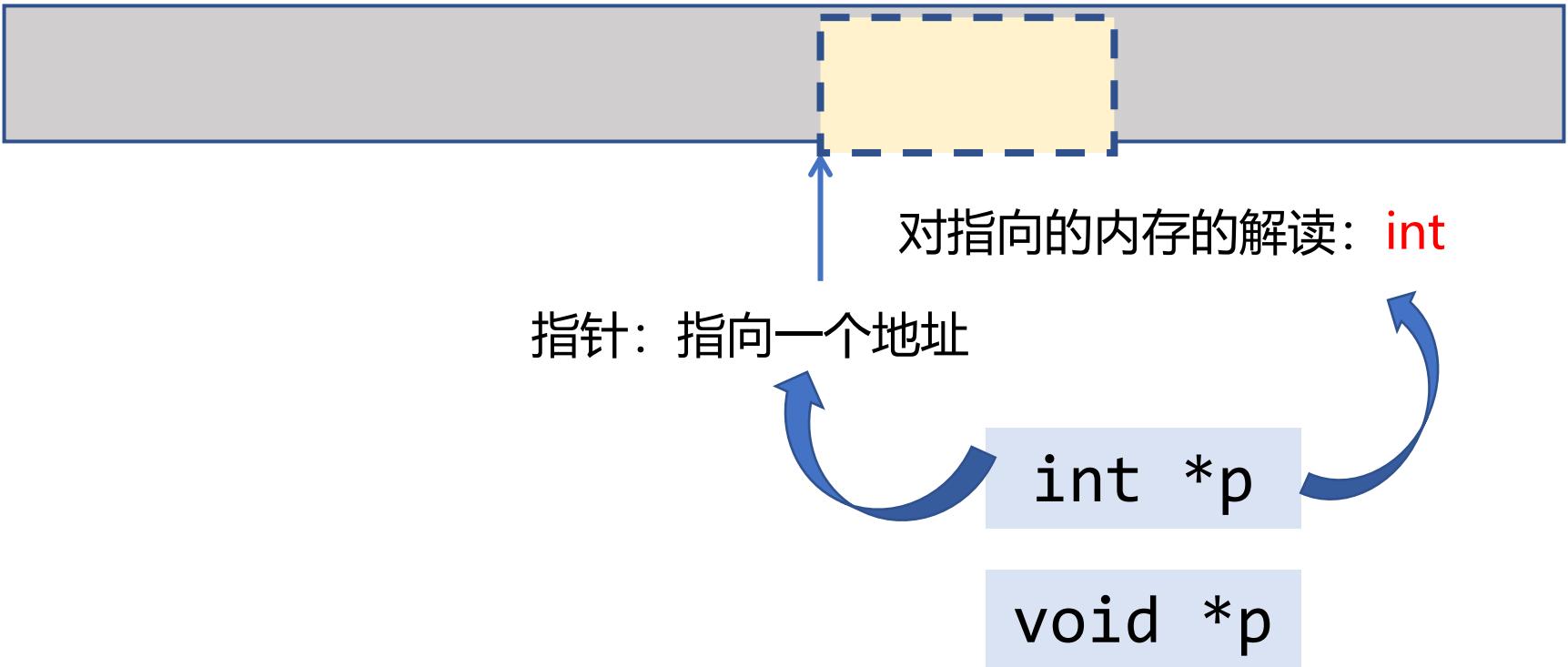
- 静态：C 代码的连续一段总能对应到一段连续的机器指令
- 动态：C 代码执行的状态总能对应到机器的状态
  - 源代码视角
    - 函数、变量、指针……
  - 机器指令视角
    - 寄存器、内存、地址……
- 两个视角的共同之处：内存
  - 代码、变量 (源代码视角) = 地址 + 长度 (机器指令视角)
  - (不太严谨地) 内存 = 代码 + 数据 + 堆栈
  - 因此理解 C 程序执行最重要的就是内存模型



$x=1 \rightarrow$    
 \*p = &x  
 \*p = 1

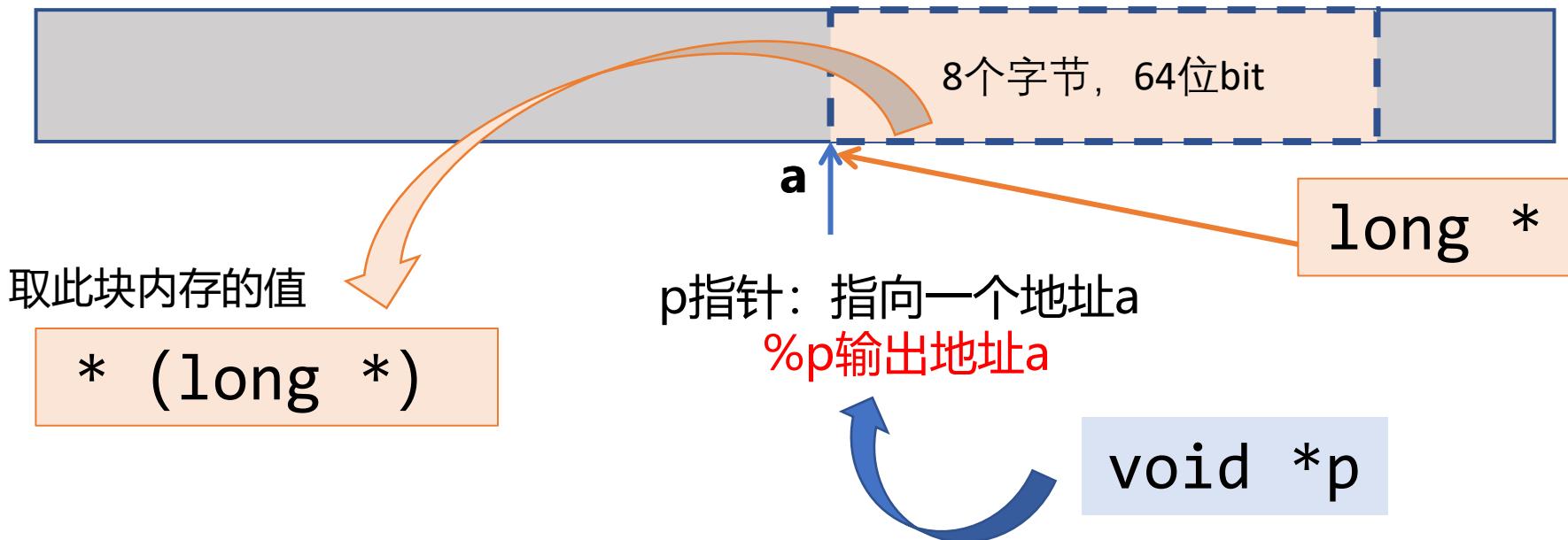
```
int main(int argc, char *argv[]) {  
    int *p = (void *) 1;      //OK  
    *p = 1;      //Segmentation fault  
}
```

# 内存



# 内存

对a地址开始的内存的解读: long



```
void printptr(void *p) {  
    printf("p = %p; *p = %016lx\n", p, *(long *)p);  
}  
.
```

指向的内存解读为long输出  
输出指针指向的地址

以16位16进制数格式输出:  $16 \times 4 = 64\text{bit}$

# main, argc和argv

- 一切皆可取地址！

```
void printptr(void *p) {      指向的地址处内存解读为long输出16位16进制
    printf("p = %p; *p = %016lx\n", p, *(long *)p);
}      输出指针指向的地址
int x;
int main(int argc, char *argv[]) {
    printptr(main); // 代码
    printptr(&main);
    printptr(&x); // 数据
    printptr(&argc); // 堆栈
    printptr(argv);
    printptr(&argv);
    printptr(argv[0]);
}
```

File Edit View Terminal Tabs Help

\$

Terminal - why@debian: ~

```
0000000000001163 <main>:  
1163: 55                  push %rbp  
1164: 48 89 e5             mov %rsp,%rbp  
1167: 48 83 ec 10          sub $0,1(%rbp)  
116b: 89 7d fc             mov %edi,-0x4(%rbp)  
116e: 48 89 75 f0          mov %rsi,-0x10(%rbp)  
1172: 48 8d 3d ea ff ff ff lea -0x10(%rip),%rd  
1179: e8 b7 ff ff ff      callq 1135 <printptr>  
117e: 48 8d 3d de ff ff ff lea -0x22(%rip),%rd
```

```
void printptr(void *p) {  
    printf("p = %p;  *p = %016lx\n",  
}  
int x;  
int main(int argc, char *argv[]) {  
    printptr(main); // 代码  
    printptr(&main);  
    printptr(&x); // 数据  
    printptr(&argc); // 堆栈  
    printptr(argv);  
    printptr(&argv);  
    printptr(argv[0]);  
}
```

代码

数据

堆栈

```
p = 0x563afd3cf163;  *p = 10ec8348e5894855  
p = 0x563afd3cf163;  *p = 10ec8348e5894855  
p = 0x563afd3d2034;  *p = 0000000000000000  
p = 0x7ffcada0761c;  *p = fd3cf1d000000001  
p = 0x7ffcada07708;  *p = 00007ffcada084fe  
p = 0x7ffcada07610;  *p = 00007ffcada07708  
p = 0x7ffcada084fe;  *p = 0074756f2e612f2e
```

# C Type System

- **类型**: 对一段内存的**解读方式**
  - 非常汇编: 没有class, polymorphism, type traits, .....
  - C里的所有的数据都可以理解成**地址 (指针) +类型 (对地址的解读)**
- 例子 (是不是感到学到了假了C语言)

```
int main(int argc, char *argv[]) {
    int (*f)(int, char *[ ]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}
```

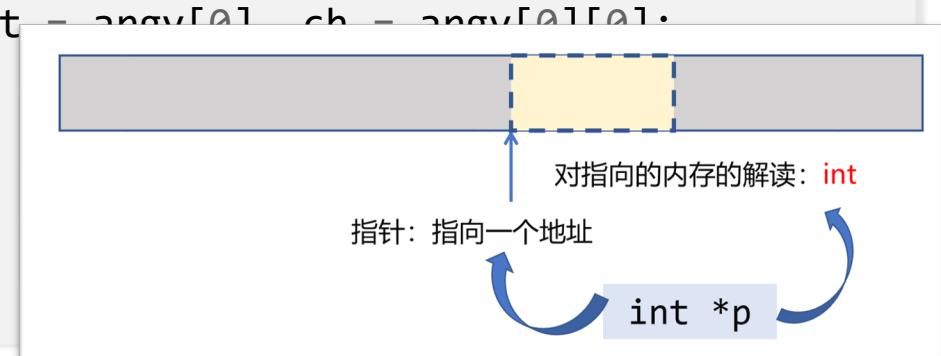
```
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'
```

```

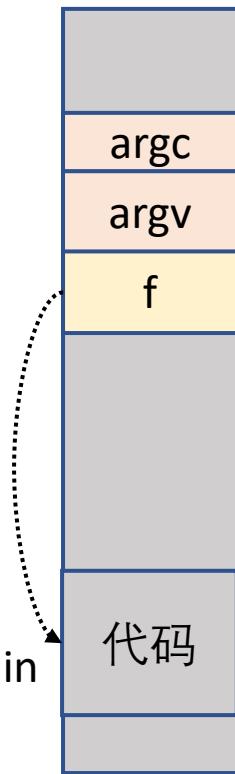
→ int main(int argc, char *argv[]) {
    int (*f)(int, char *[ ]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"\n"; ch
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

## 函数指针



内存



**char \*argv[] → char \*\*argv → (char \* )\*argv**

4字节

? 指针, 存储地址, 64位机器, 8字节

? 指针, 存储地址, 64位机器, 8字节

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first
        printf("arg = \"%s\"; ch\n", *a);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```

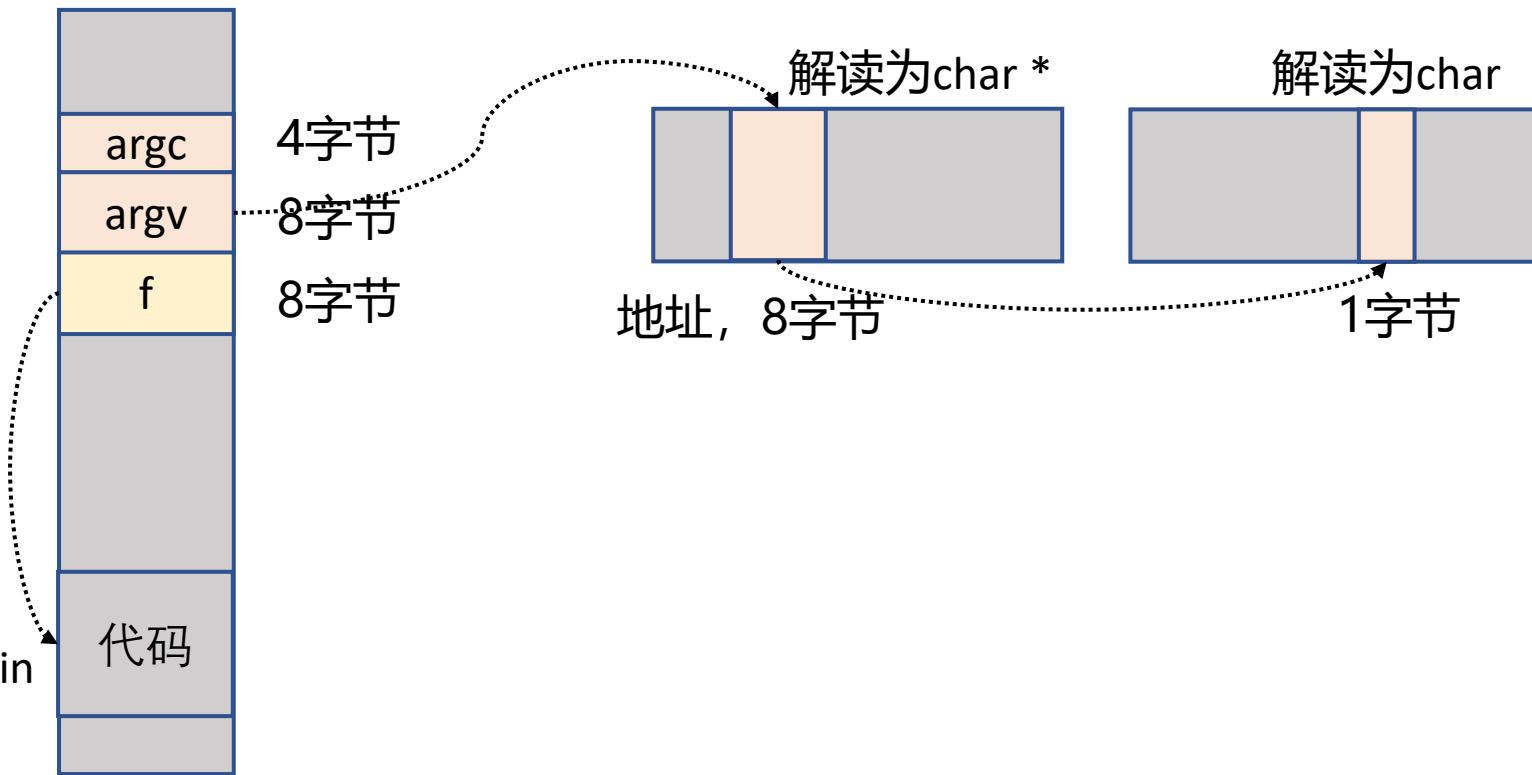


**char \*argv[] → char \*\*argv → (char \*) \*argv**

内存

堆栈

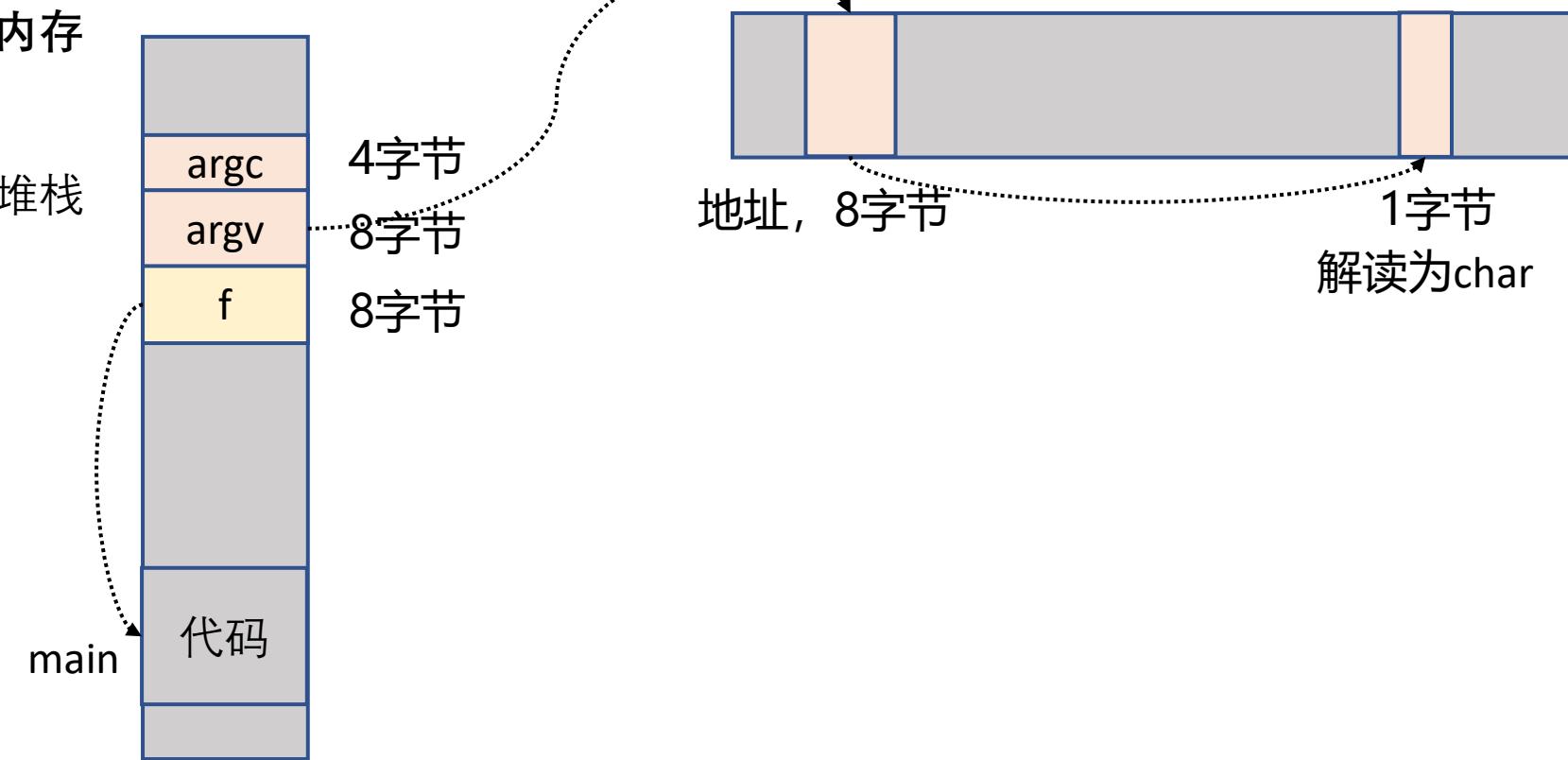
main



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[ ]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

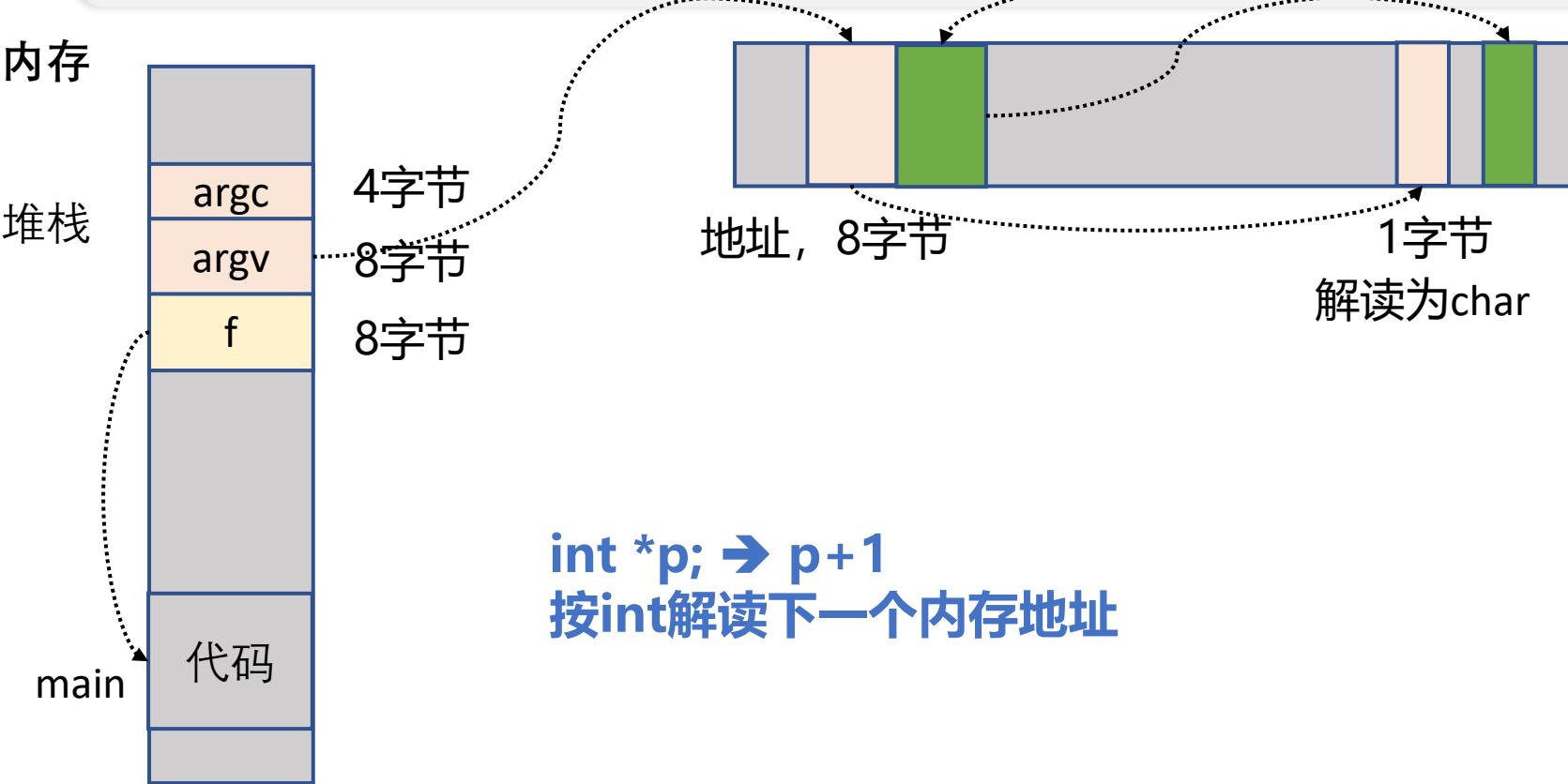
```



```
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}
```

什么是`argv+1`?

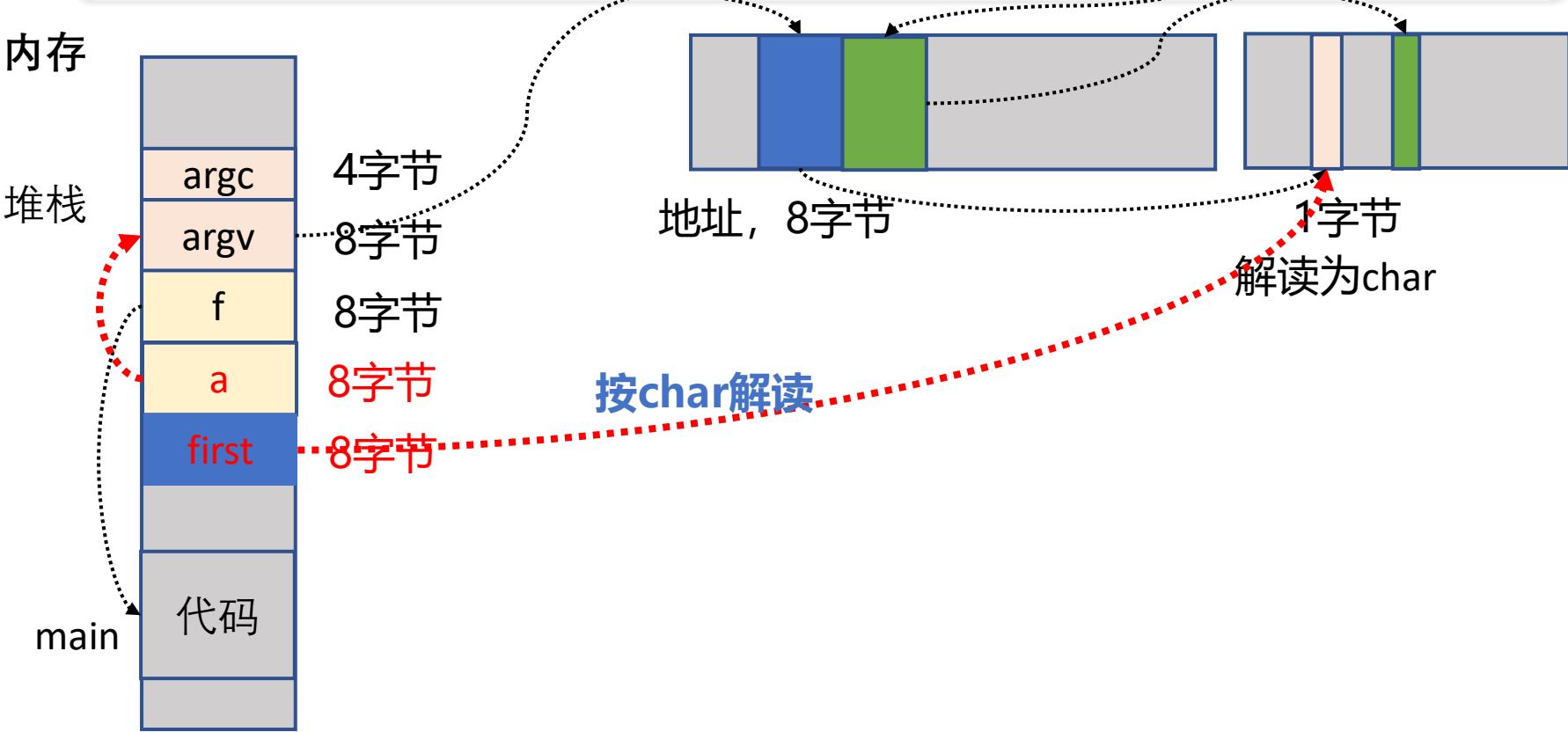
## 什么是argv+1?



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

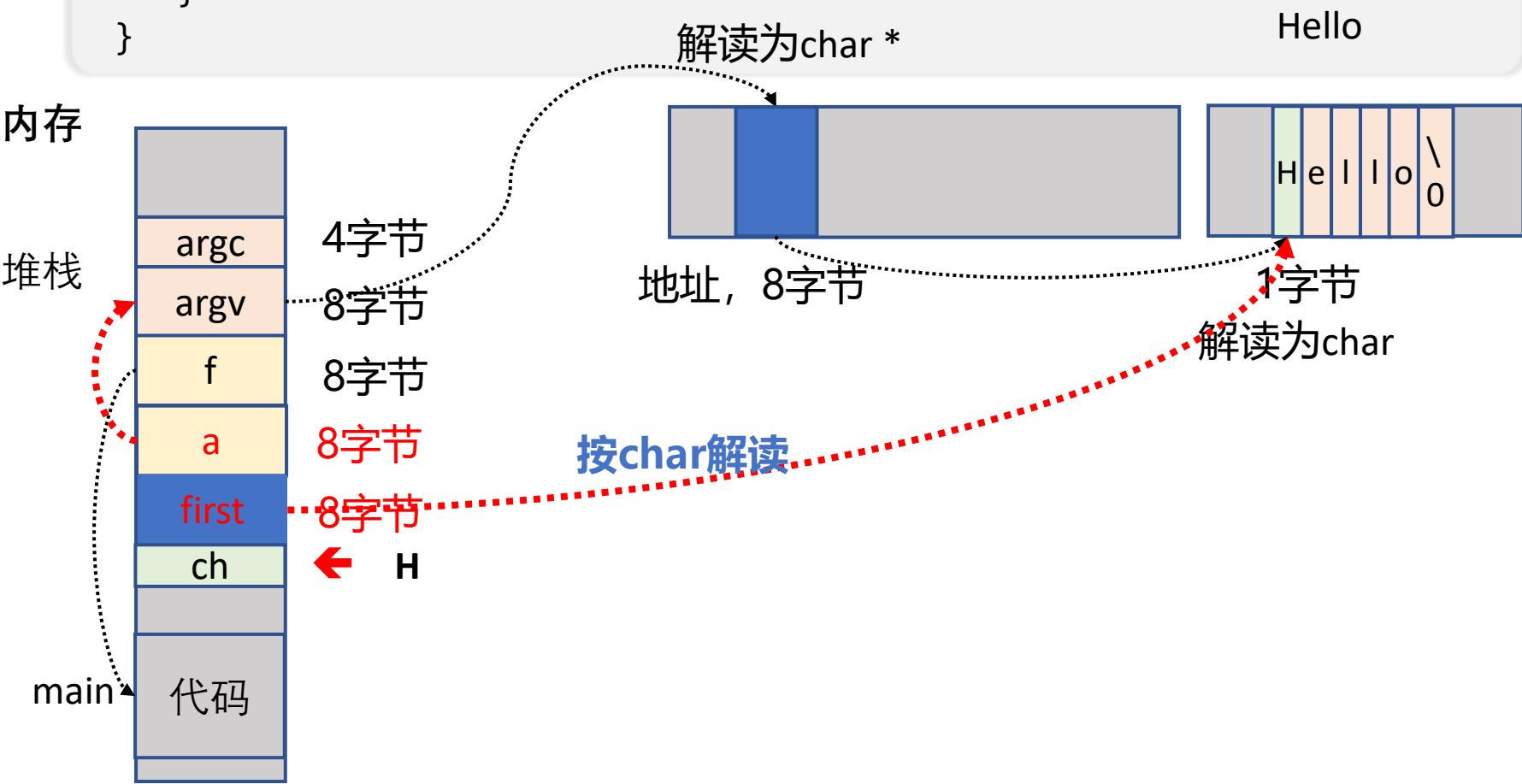
```



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

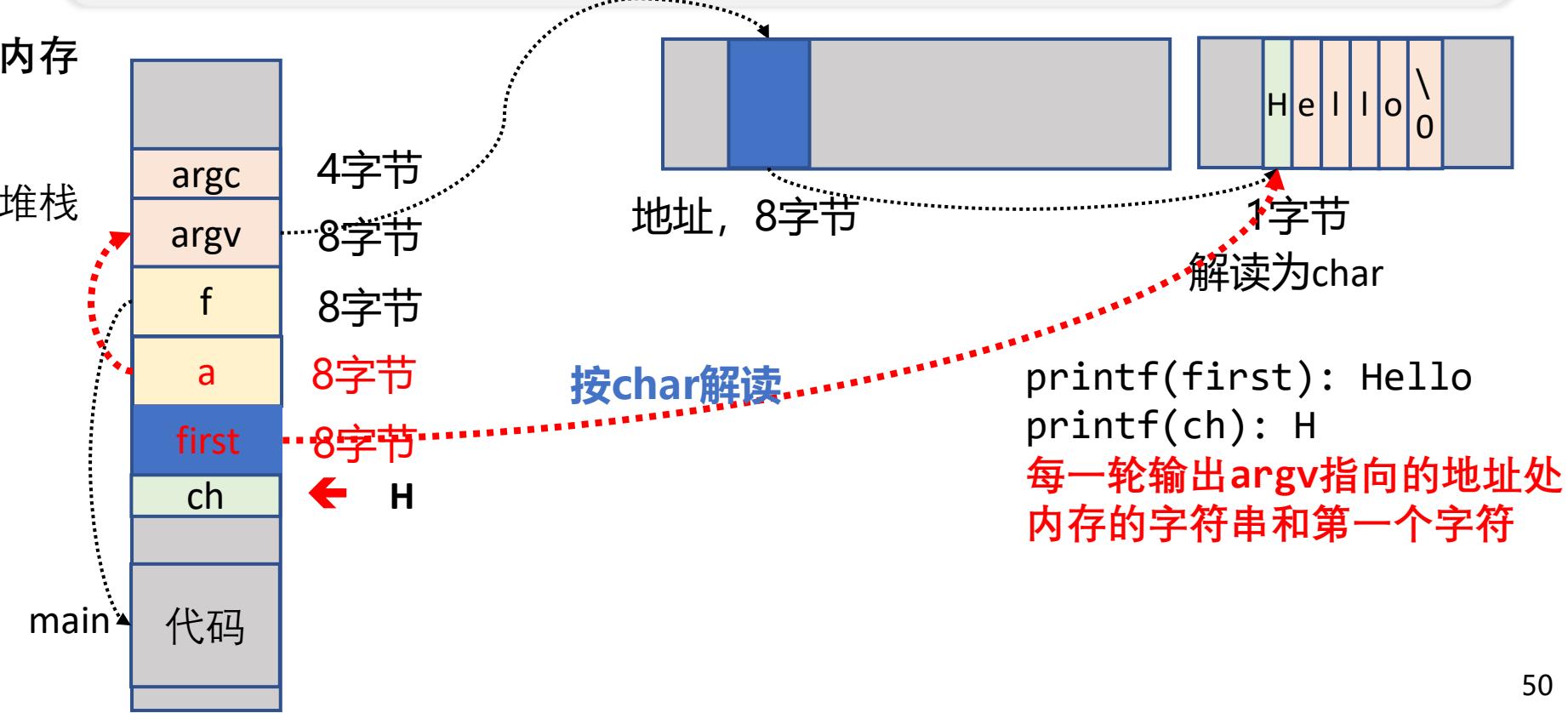
```



```

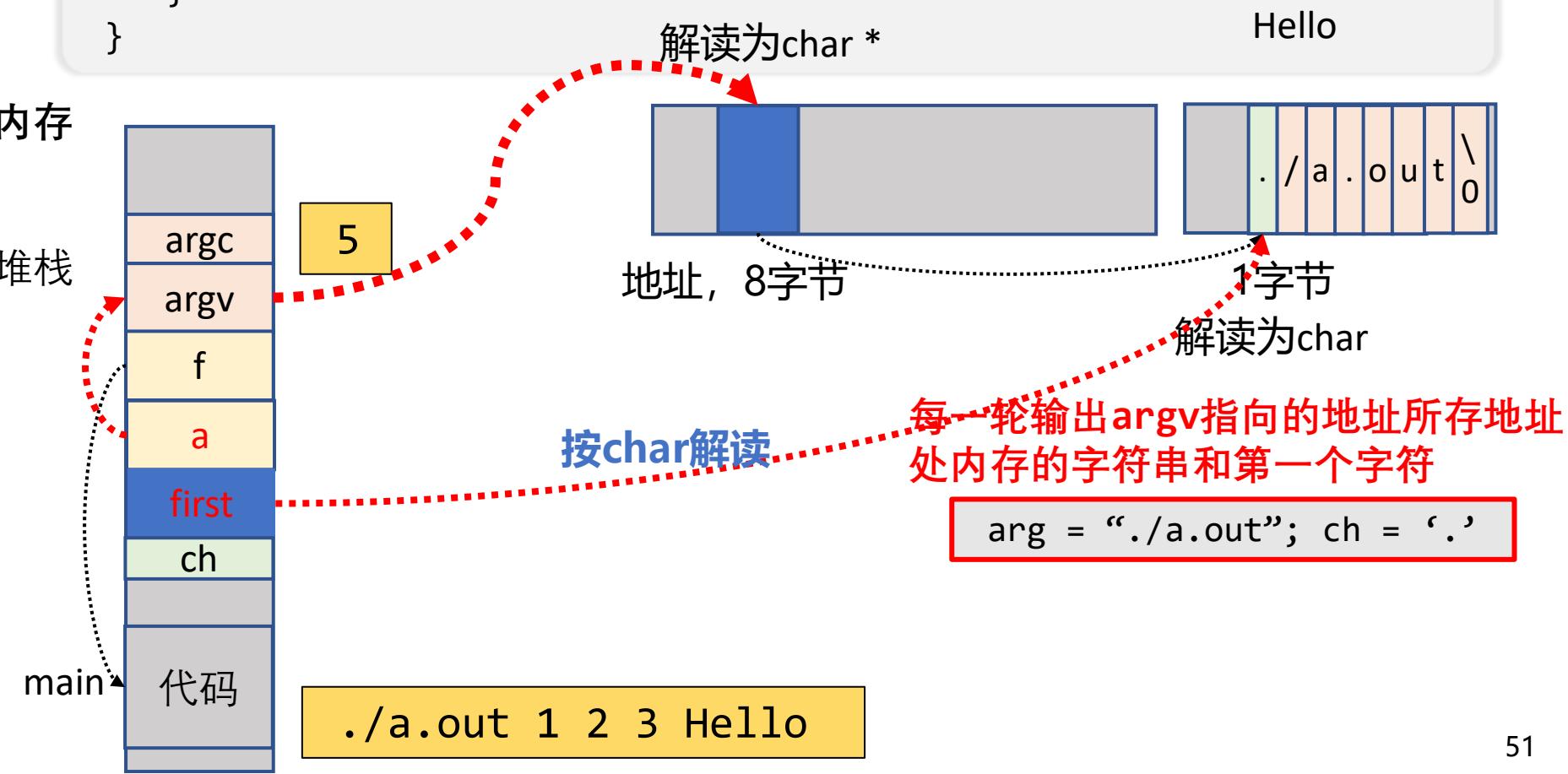
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```



```
int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first =
printf("arg = \"%s\"; ch =
assert(**a == ch);
f(argc - 1, argv + 1);
}
}
```

```
$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'
```



```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\n", *a, **a);
        assert(**a == *first);
        f(argc - 1, argv + 1);
    }
}

```

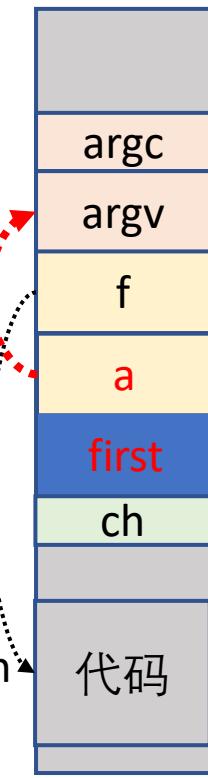
\$ ./a.out 1 2 3 hello  
 arg = "./a.out"; ch = '.'  
 arg = "1"; ch = '1'  
 arg = "2"; ch = '2'  
 arg = "3"; ch = '3'  
 arg = "hello"; ch = 'h'

内存

堆栈

main

代码

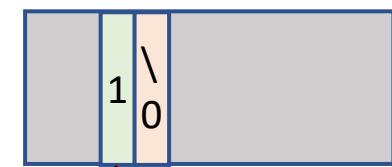


./a.out 1 2 3 Hello

按char解读

解读为char \*

地址, 8字节



Hello

1字节  
解读为char

每一轮输出 argv 指向的地址所存地址  
 处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\n", *a, **a);
        assert(**a == *first);
        f(argc - 1, argv + 1);
    }
}

```

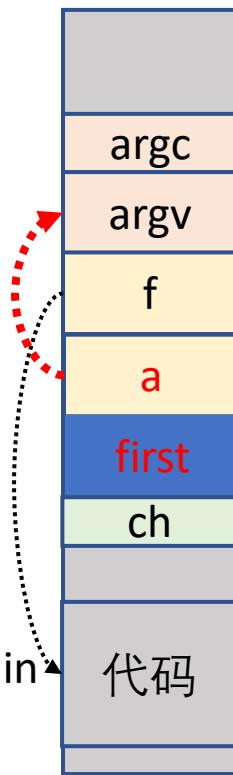


```

$ ./a.out 1 2 3 hello
arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'

```

内存



3

按char解读

地址, 8字节



Hello

1字节  
解读为char

每一轮输出 argv 指向的地址所存地址  
处内存的字符串和第一个字符

arg = “./a.out”; ch = ‘.’
arg = “1”; ch = ‘1’
arg = “2”; ch = ‘2’

./a.out 1 2 3 Hello

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\n", *a, **a);
        assert(**a == *first);
        f(argc - 1, argv + 1);
    }
}

```

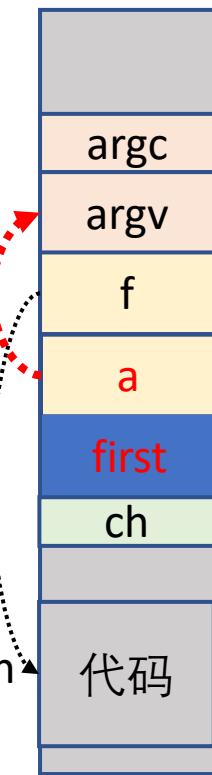
\$ ./a.out 1 2 3 hello  
 arg = "./a.out"; ch = '.'  
 arg = "1"; ch = '1'  
 arg = "2"; ch = '2'  
 arg = "3"; ch = '3'  
 arg = "hello"; ch = 'h'

内存

堆栈

main

./a.out 1 2 3 Hello

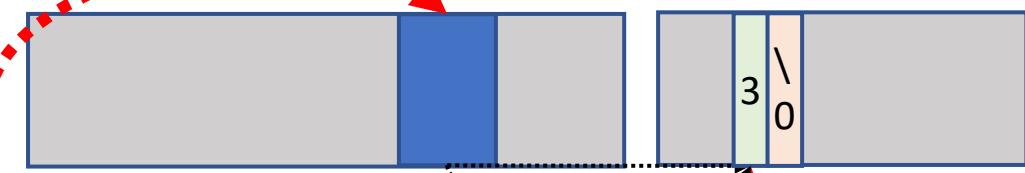


2

按char解读

地址, 8字节

解读为char \*



Hello

1字节  
解读为char

每一轮输出 argv 指向的地址所存地址  
 处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'

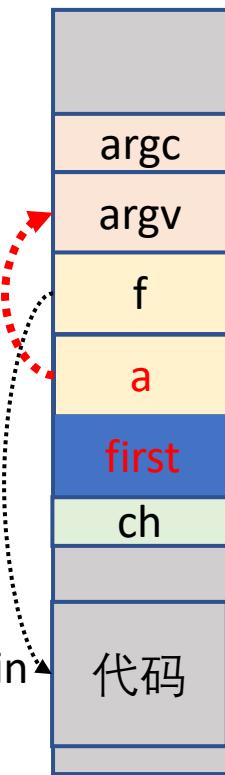
```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\n", *a, **a);
        assert(**a == *first);
        f(argc - 1, argv + 1);
    }
}

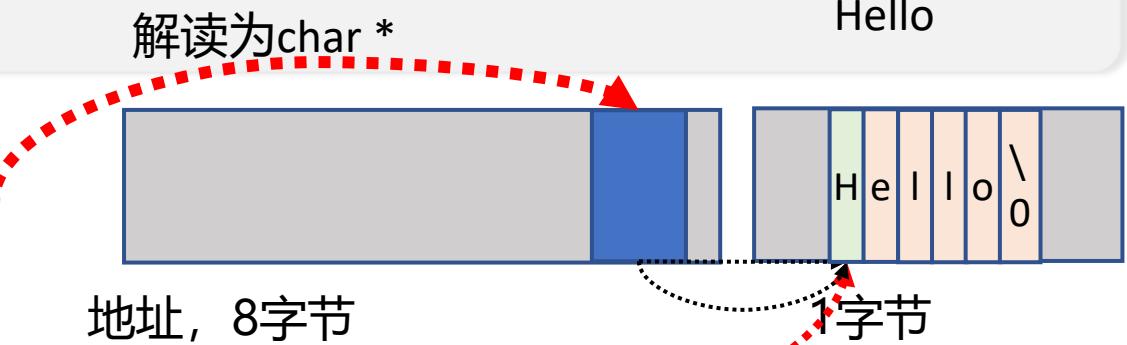
```

\$ ./a.out 1 2 3 hello  
 arg = "./a.out"; ch = '.'  
 arg = "1"; ch = '1'  
 arg = "2"; ch = '2'  
 arg = "3"; ch = '3'  
 arg = "hello"; ch = 'h'

内存  
堆栈  
main  
代码

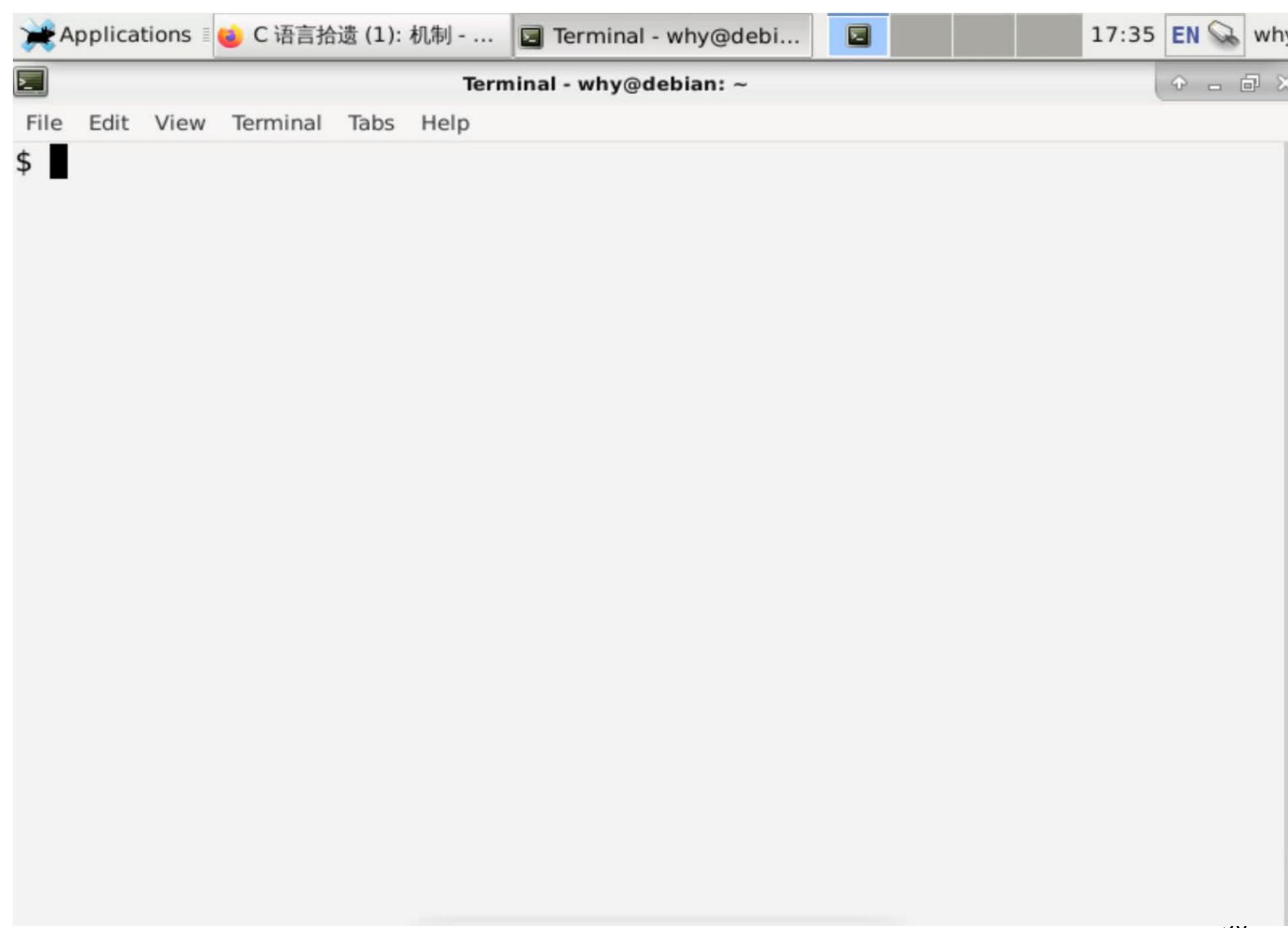


./a.out 1 2 3 Hello



每一轮输出 argv 指向的地址所存地址处内存的字符串和第一个字符

arg = "./a.out"; ch = '.'
arg = "1"; ch = '1'
arg = "2"; ch = '2'
arg = "3"; ch = '3'
arg = "hello"; ch = 'h'



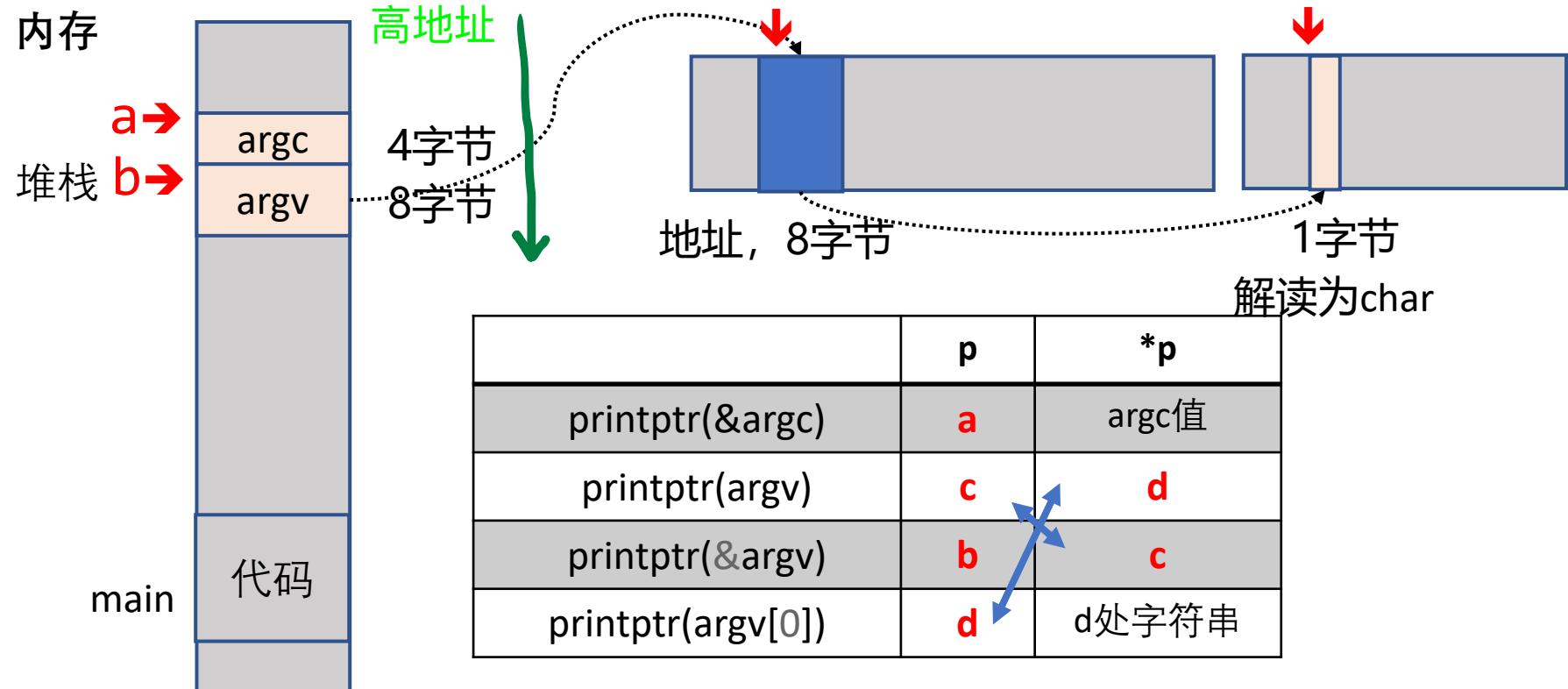
```

void printptr(void *p) {    指向的地址处内存解读为long输出16位16进制
    printf("p = %p; *p = %016lx\n", p, *(long *)p);
}    输出指针指向的地址
int x;
int main(int argc, char *argv[]) {
    printptr(main); // 代码
    printptr(&main);
    printptr(&x); // 数据
    printptr(&argc); // 堆栈
    printptr(argv);
    printptr(&argv);
    printptr(argv[0]);
}

```

```

$ ./a.out
p = 0x563af3cf163; *p = 10ec8348e5894855
p = 0x563af3cf163; *p = 10ec8348e5894855
p = 0x563af3d2034; *p = 10ec8348e5894855
p = 0x7ffcada0761c; *p = fd3cf1d000000001
p = 0x7ffcada07708; *p = 00007ffcada084fe
p = 0x7ffcada07610; *p = 00007ffcada07708
p = 0x7ffcada084fe; *p = 0074756f2e612f2e
.
```



# 从main函数开始执行

- 标准规定C程序从main开始执行
  - (思考题：谁调用的main？进程执行的第一条指令是什么？)

```
int main(int argc, char *argv[]);
```

- argc (argument count): 参数个数
- argv (argument vector): 参数列表 (NULL结束)
- ls -al
  - argc = 2, argv = ["ls", "-al", NULL]

End.

- C语言**简单** (在可控时间成本里可以精通)
- C语言**通用** (大量系统是C语言编写的)
- C语言**实现对底层机器的精准控制** (鸿蒙)
- 推荐阅读: [The Art of Readable Code](#)

# C 语言拾遗(2): 编程实践

王慧妍

[why@nju.edu.cn](mailto:why@nju.edu.cn)

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

PA1已悄悄发布：（OJ暂未开启PA1评测）

- \* PA 1.1: 2023.9.30 (此为建议的不计分 deadline)
- \* PA 1.2: 2023.10.5 (此为建议的不计分 deadline)
- \* PA 1.3: 2023.10.15 23:59:59 (以此 deadline 计按时提交bonus)

- 假设你已经熟练使用C语言的各种机制（并没有）
  - 原则上给需求就能搞定任何代码（并不是）
- 本次课程
  - 怎样写代码才能从一个大型项目中存活下来？
    - 核心准则：编写可读代码
    - 两个例子：计数器，YEMU

# 核心准则：编写可读代码

# 一个极端不可读的例子

- IOCCC'11 best self documenting program
  - 不可读 = 不可维护

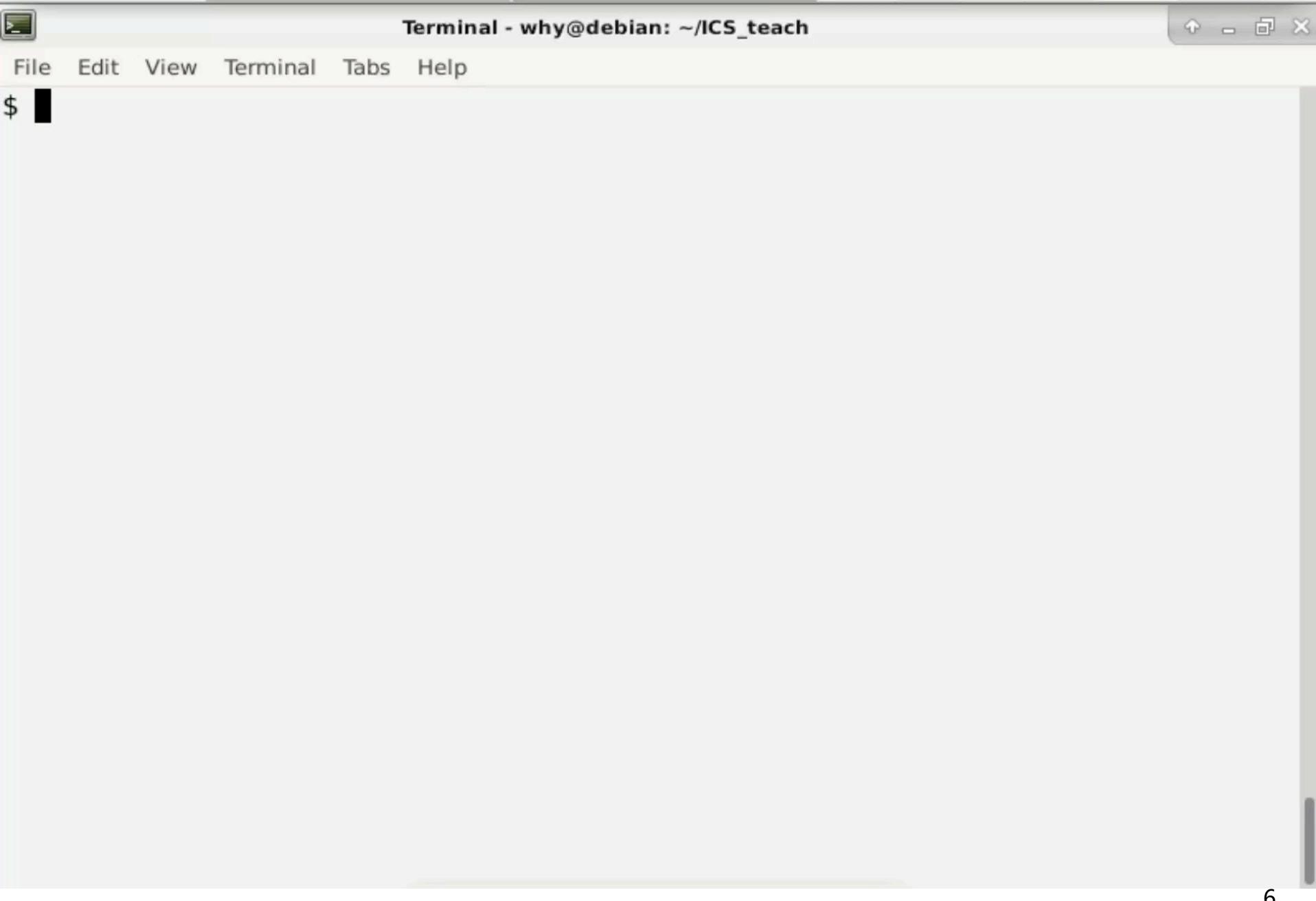
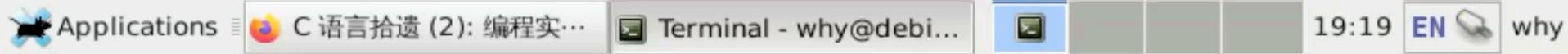
```
puts(usage: calculator 11/26+222/31
+~~~~~calculator-\
!
7.584,367 )
+~~~~~+
! clear ! 0 ||1 -x 1 tan I (/) |
+~~~~~+
! 1 | 2 | 3 ||1 1/x 1 cos I (*) |
+~~~~~+
! 4 | 5 | 6 ||1 exp 1 sqrt I (+) |
+~~~~~+
! 7 | 8 | 9 ||1 sin 1 log I (-) |
+~~~~~+(0 )
```

# 一个极端不可读的例子

- IOCCC'11 best self documenting program

- 不可读 = 不可维护

```
#define clear 1;
    if(c>=11){c=0;sscanf(_, "%lf%c",&r,&c);while(*++_-
c);}\
    else if(argc>=4&&!main(4-
(*_++=='('),argv))_++;g:c+=
#define puts(d,e) return 0;}{double a;int b;char
c=(argc<4?d)&15; \
b=(*%__LINE__+7)%9*(3*e>>c&1);c+=
#define I(d)
(r);if(argc<4&&*#d==*_){a=r;r=usage?r*a:r+a;goto
g;}c=c
#define return if(argc==2)printf("%f\n",r);return
argc>=4+ #define usage main(4-__LINE__/26,argv)
#define calculator *_*(int)
#define l (r);r=--b?r:
#define _ argv[1]
```

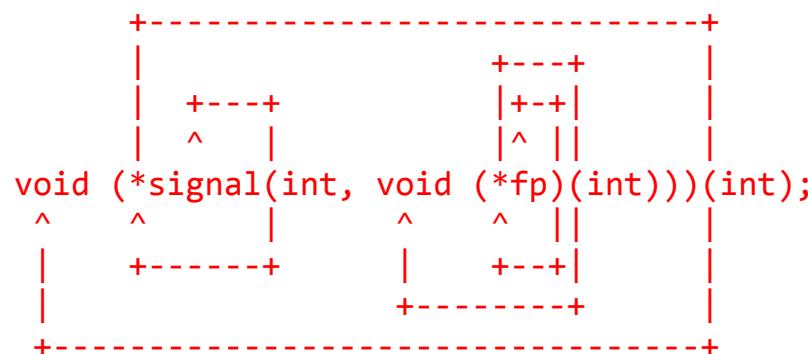


# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

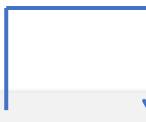
```
void (*signal (int sig, void (*func)(int)))(int);
```



# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

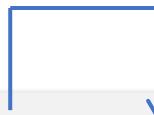


- signal是一个函数
  - 参数为.....
  - 返回值为.....

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

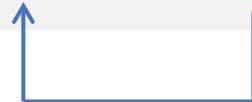


- signal是一个函数
  - 参数为int和一个函数指针 (参数为int, 返回值为void)
  - 返回值为.....

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```



- signal是一个函数
  - 参数为int和一个函数指针 (参数为int, 返回值为void)
  - 返回值为一个.....指针

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针 (函数参数为int, 返回值为void)
  - 返回值为一个指向函数的指针 (参数为..., 返回值为...)

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针 (函数参数为int, 返回值为void)
  - 返回值为一个指向函数的指针 (参数为int, 返回值为...)

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则的案例

```
void (*signal (int sig, void (*func)(int)))(int);
```



- signal是一个函数
  - 参数为int和一个指向函数的指针 (函数参数为int, 返回值为void)
  - 返回值为一个指向函数的指针 (参数为int, 返回值为void)
- 太复杂了>\_<, 有没有更简单的办法

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则

```
void (*signal (int sig, void (*func)(int)))(int);
```

- signal是一个函数
  - 参数为int和一个指向函数的指针 (函数参数为int, 返回值为void)
  - 返回值为一个指向函数的指针 (参数为int, 返回值为void)

```
void (*signal (int sig, void (*func)(int)))(int);
```

```
void (*signal (int sig, func))(int);
```

```
void (*)(int);
```

# 一个现实中可能遇到的例子

- 人类不可读版本 (STFW: clockwise/spiral rule)
  - 终极使用顺时针螺旋法则

```
void (*signal (int sig, void (*func)(int)))(int);
```

- 人类可读版本

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int, sighandler_t);
```

# man 2 signal

Applications Terminal - why@debian... 09:42 EN why

File Edit View Terminal Tabs Help

SIGNAL(2) Linux Programmer's Manual SIGNAL(2)

**NAME**

signal - ANSI C signal handling

**SYNOPSIS**

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

**DESCRIPTION**

The behavior of **signal()** varies across UNIX versions, and has also varied historically across different versions of Linux. **Avoid its use:** use **sigaction(2)** instead. See [Portability](#) below.

**signal()** sets the disposition of the signal signum to handler, which is either **SIG\_IGN**, **SIG\_DFL**, or the address of a programmer-defined function (a "signal handler").

If the signal signum is delivered to the process, then one of the following happens:

- \* If the disposition is set to **SIG\_IGN**, then the signal is ignored.
- \* If the disposition is set to **SIG\_DFL**, then the default action associated with the signal (see **signal(7)**) occurs.
- \* If the disposition is set to a function, then first either the disposition is reset to **SIG\_DFL**, or the signal is blocked (see [Portability](#) below), and then

Manual page signal(2) line 1 (press h for help or q to quit)

# 编写代码的准则：降低维护成本

Programs are meant to be read by humans and only incidentally for computers to execute. — D. E. Knuth

(程序首先是拿给人读的，其次才是被机器执行。)

- 宏观

- 做好分解和解耦（现实世界也是这样管理复杂系统）

- 微观

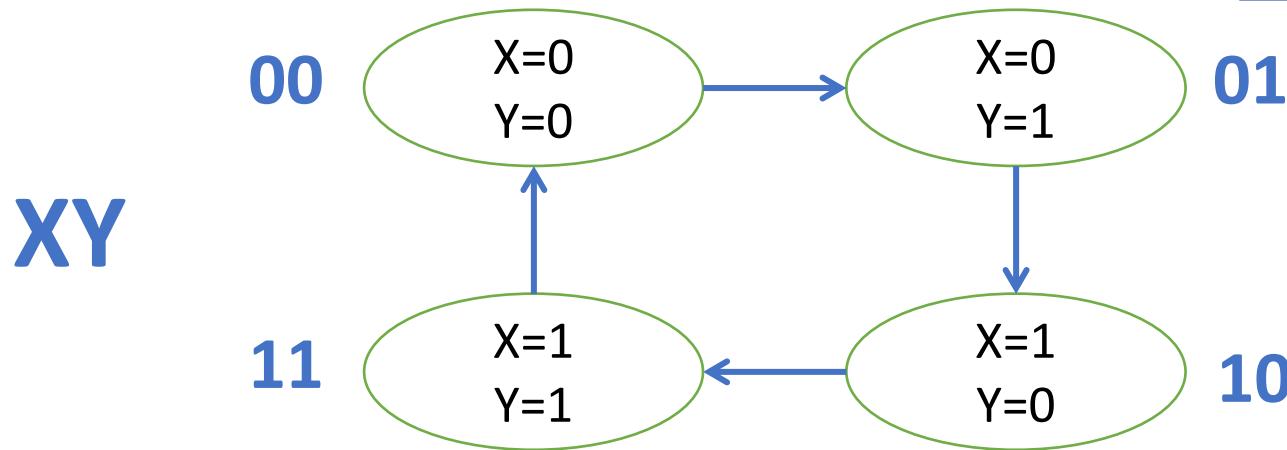
- “不言自明”
    - 通过阅读代码能够理解一段程序的行为 (**implementation**)
  - “不言自证”
    - 通过阅读代码能够验证一段程序**implementation**与**specification**的一致性

例子：实现数字逻辑电路模拟器

# 数字逻辑电路模拟器

- 假想的数字逻辑电路

- 若干个1-bit边沿触发寄存器 ( $X, Y, \dots$ )
- 若干个逻辑门



- 基本思路：状态（存储模拟）+计算模拟

- 状态 = 变量
  - int  $X = 0, Y = 0;$
- 计算

```
X1 = (!X && Y) || (X && !Y);  
Y1 = !Y;  
X = X1; Y = Y1;
```

# 数字逻辑电路模拟器

- 需求

- 加一位边沿寄存器
- 自己独立的逻辑

```
int X=0, Y=0;  
int X1=0, Y1=0;  
while(1){  
    X1 = (!X&&Y) || (X&&!Y);  
    Y1 = !Y;  
    X = X1; Y = Y1;  
}
```

```
int X=0, Y=0, Z=0;  
int X1=0, Y1=0, Z1=0;  
while(1){  
    X1 = (!X&&Y) || (X&&!Y);  
    Y1 = !Y;  
    Z1 = .....;  
    X = X1; Y = Y1; Z = Z1;  
}
```



# 通用数字逻辑电路模拟器

```
#define FORALL_REGS(_)    _(_X) _(_Y)
#define LOGIC                X1 = (!X&&Y) || (X&&!Y); \
                           Y1 = !Y;
#define DEFINE(X)             static int X, X##1;
#define UPDATE(X)              X = X##1;
#define PRINT(X)               printf(#X " = %d; ", X);

int main() {
    FORALL_REGS(DEFINE);

    while (1) { // clock
        FORALL_REGS(PRINT);
        putchar('\n');
        sleep(1);
        LOGIC;
        FORALL_REGS(UPDATE);
    }
}
```

Terminal - a.c (~/ICS\_teach) - VIM

File Edit View Terminal Tabs Help

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #define FORALL_REGS(_X) _X(X) _X(Y)
4 #define LOGIC X1 = (!X && Y) || (X&&!Y); \
5 Y1 = !Y;
6 #define DEFINE(X) static int X, X##1;
7 #define UPDATE(X) X = X##1;
8 #define PRINT(X) printf(#X " = %d; ", X);
9
10 int main() {
11     FORALL_REGS(DEFINE);
12     while (1) { // clock
13         FORALL_REGS(PRINT); putchar('\n'); sleep(1);
14         LOGIC;
15         FORALL_REGS(UPDATE);
16     }
17 }
18
```

a.c 3,1 All  
"a.c" 18L, 437C

Applications C 语言拾遗 (2): 编程实... Terminal - a.c (~/ICS\_... 11:54 EN why

Terminal - a.c (~/ICS\_teach) - VIM

File Edit View Terminal Tabs Help

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #define FORALL_REGS(_X) _X(X) _X(Y) _X(Z)
4 #define LOGIC           X1 = (!X&&Y&&Z) || (X&&(!Y&&Z)); \
5                                Y1 = (!Y&&Z) || (Y&&!Z); \
6                                Z1 = !Z;
7 #define DEFINE(X)        static int X, X##1;
8 #define UPDATE(X)         X = X##1;
9 #define PRINT(X)          printf(#X " = %d; ", X);
10
11 int main() {
12     FORALL_REGS(DEFINE);
13     while (1) { // clock
14         FORALL_REGS(PRINT); putchar('\n'); sleep(1);
15         LOGIC;
16         FORALL_REGS(UPDATE);
17     }
18 }
19
```

a.c 4,1 All  
"a.c" 19L, 498C

# 通用数字逻辑电路模拟器

```
#define FORALL_REGS(_)(X)(Y)
#define LOGIC X1 = (!X&&Y)|| (X&&!Y); \
Y1 = !Y;
#define DEFINE(X) static int X, X##1;
#define UPDATE(X) X = X##1;
#define PRINT(X) printf(#X " = %d; ", X);

int main() {
    FORALL_REGS(DEFINE);

    while (1) { // clock
        FORALL_REGS(PRINT);
        putchar('\n');
        sleep(1);
        LOGIC;
        FORALL_REGS(UPDATE);
    }
}
```

# 使用预编译：Pros and Cons

---

- Pros

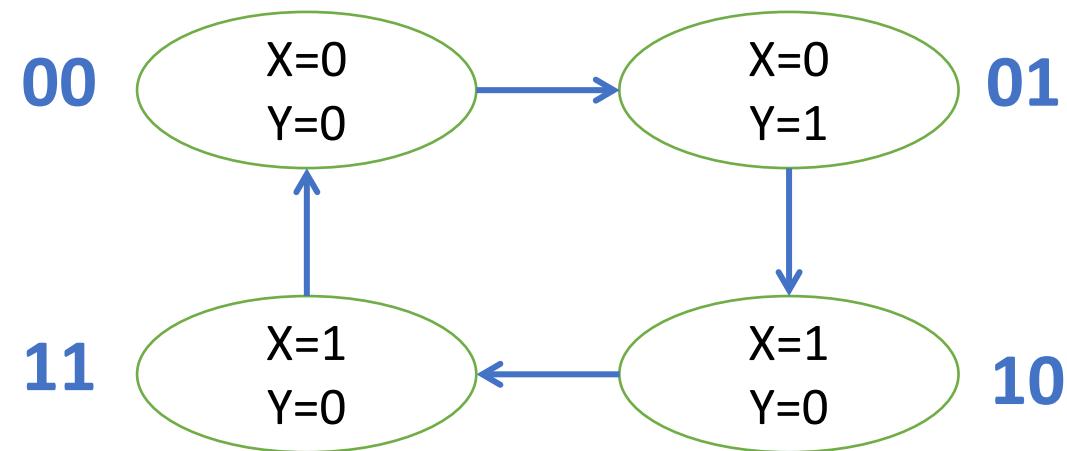
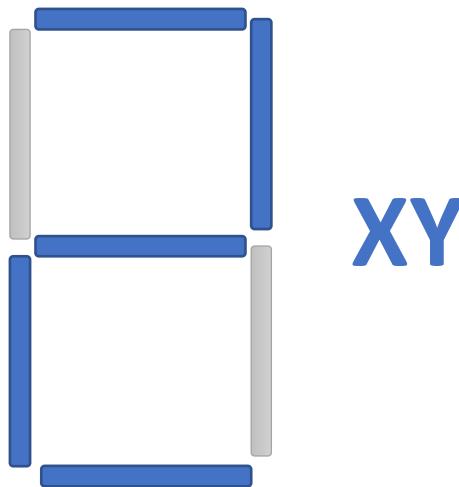
- 增加/删除寄存器只要改很少的地方
- 阻止一些编程错误
  - 忘记更新寄存器
  - 忘记打印寄存器
- “不言自明”

- Cons

- 可读性变差（不太像C代码）
  - “不言自证”差一些
- 给IDE解析带来一些困难

# 更完整的实现：数码管显示

- logisim.c 和 display.py





## Terminal - why@debian: ~/ICS\_teach

File Edit View Terminal Tabs Help

\$

```

#include <stdio.h>
#include <unistd.h>
#define FORALL_REGS(_)      _(_X) _(_Y)
#define OUTPUTS(_)          _(_A) _(_B) _(_C) _(_D) _(_E) _(_F) _(_G)
#define LOGIC               X1 = (!X && Y) || (X && !Y); \
                           Y1 = !Y; \
                           A = (!X && !Y) || (X && !Y) || (X && Y); \
                           B = 1; \
                           C = (!X && !Y) || (!X && Y) || (X && Y); \
                           D = (!X && !Y) || (X && !Y) || (X && Y); \
                           E = (!X && !Y) || (X && !Y); \
                           F = (!X && !Y); \
                           G = (X && !Y) || (X && Y);
#define DEFINE(X)           static int X, X##1;
#define UPDATE(X)            X = X##1;
#define PRINT(X)             printf(#X " = %d; ", X);

int main() {
    FORALL_REGS(DEFINE);
    OUTPUTS(DEFINE);
    while (1) { // clock
        LOGIC;
        OUTPUTS(PRINT);
        putchar('\n');
        fflush(stdout);
        sleep(1);
        FORALL_REGS(UPDATE);
    }
}

```

# 更完整的实现：数码管显示

- logisim.c 和 display.py
  - 也可以考虑增加更多外设：开关、UART等
  - 原理无限接近大家数字电路课玩过的FPGA
- 等等.....FPGA?
  - 这玩意不是万能的吗？？？
  - 我们能模拟它，是不是就能模拟计算机系统？
    - Yes!
    - 我们实现了一个超级超级低配版 NEMU!



例子：实现YEMU全系统模拟器

# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



PC

0000  
0001

R0

0001  
0010

R1



R2



R3



0

1

15



内存

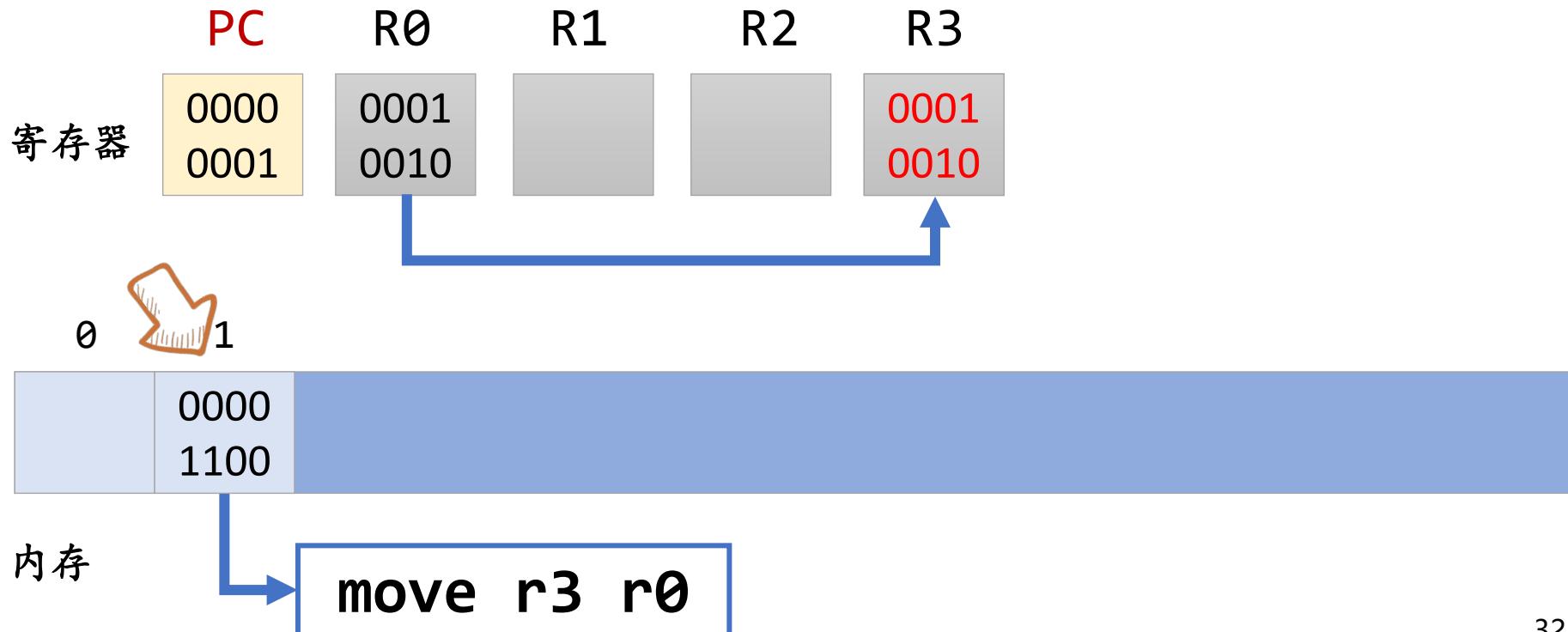
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



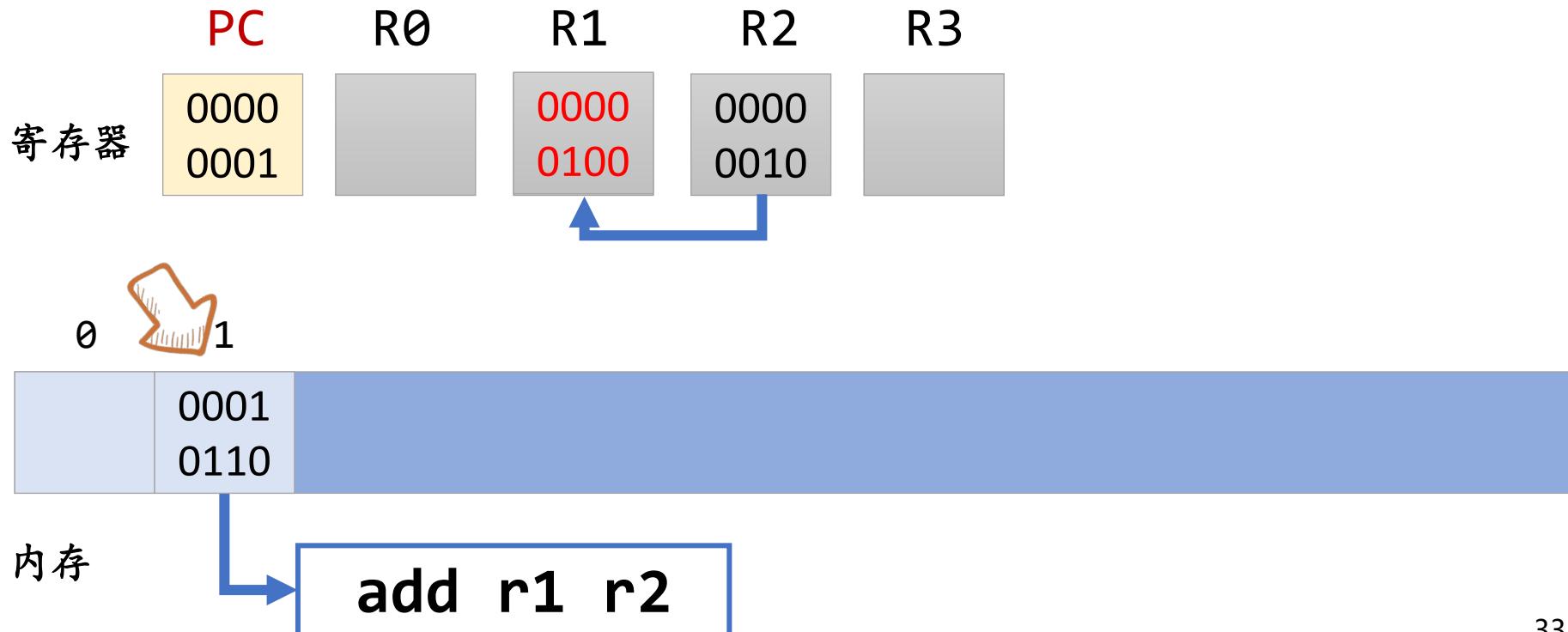
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



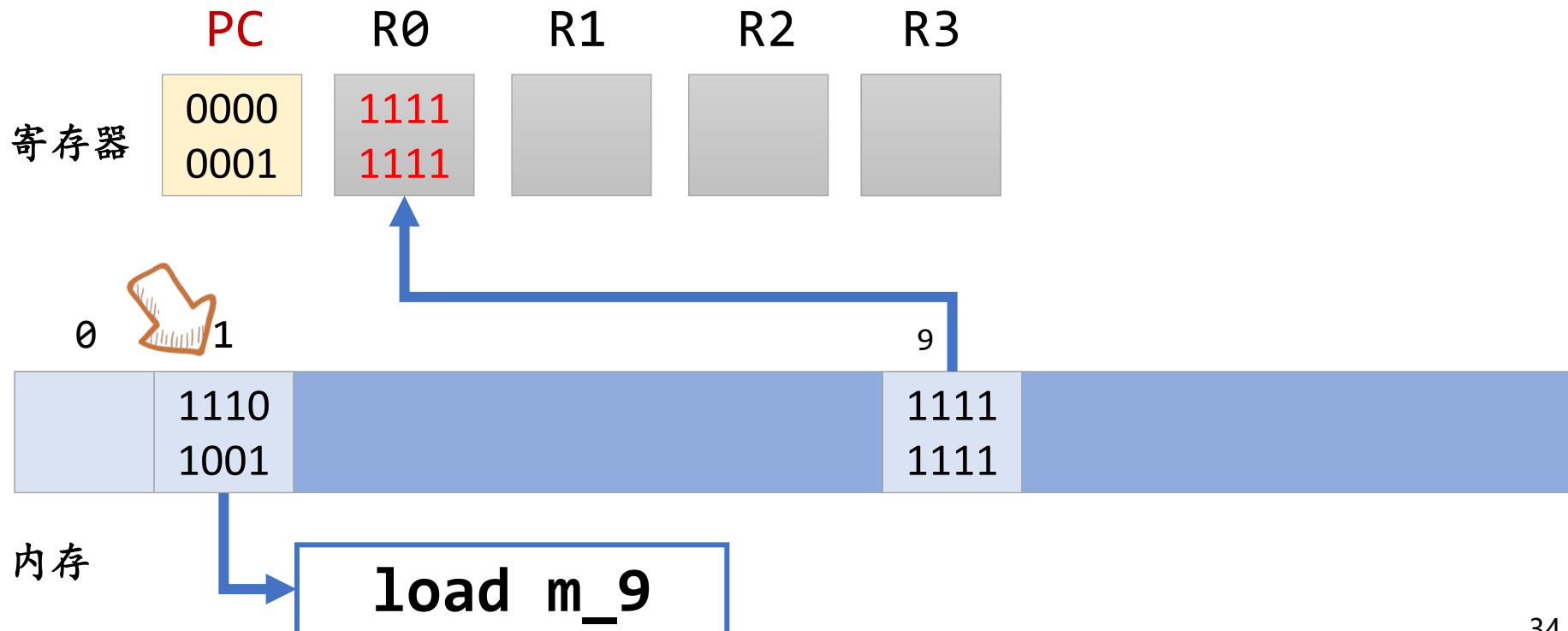
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



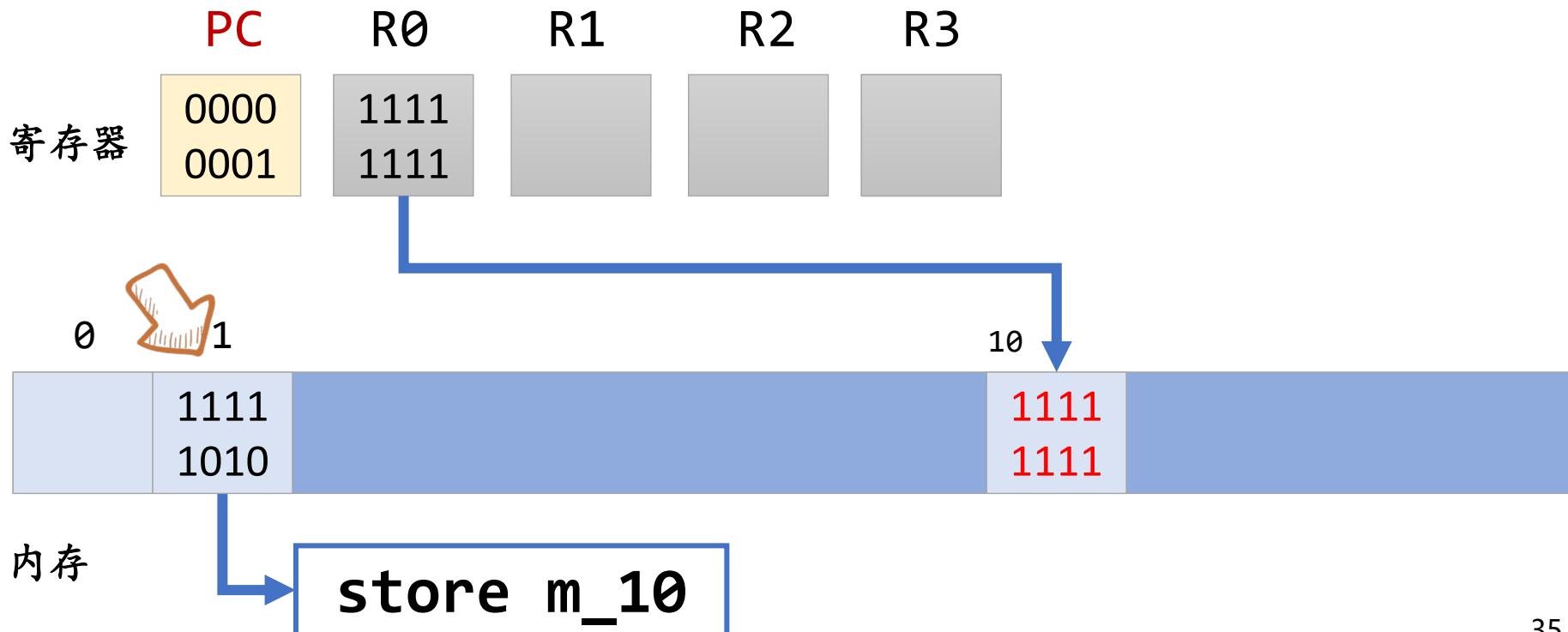
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



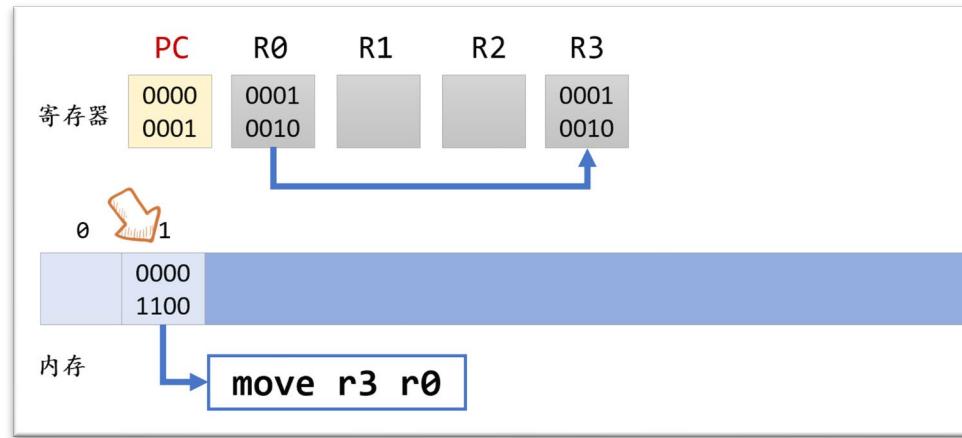
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0			
mov	[0	0	0	0	[	r	t	]	[ r	s	]
add	[0	0	0	1	[	r	t	]	[ r	s	]
load	[1	1	1	0	[		addr		]		
store	[1	1	1	1	[		addr		]		



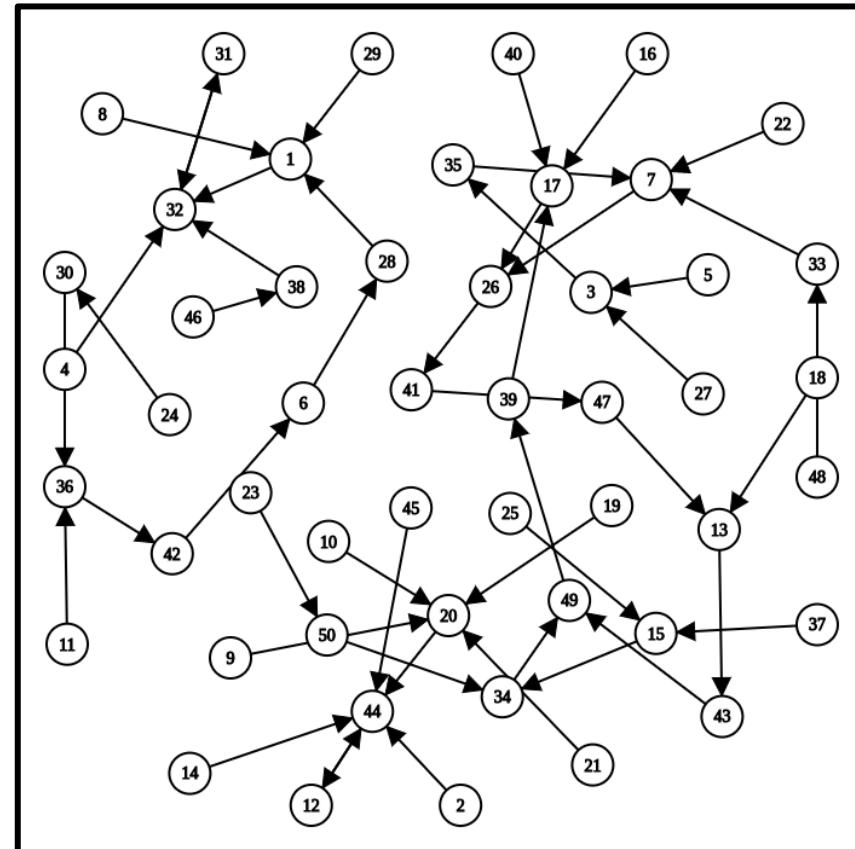
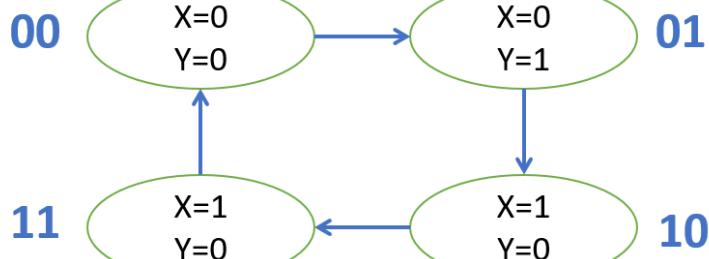
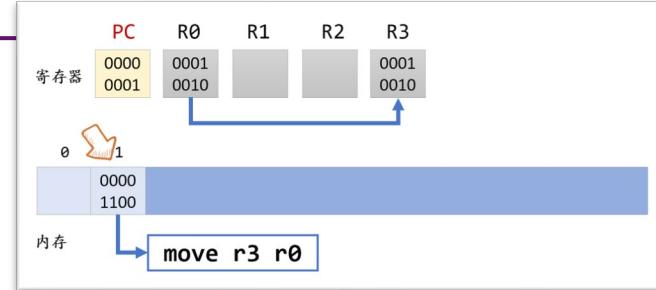
- 有“计算机系统”的感觉了?

- 它显然可以用数字逻辑电路实现
- 不过我们不需要在门层面实现它
  - 我们接下来实现一个超级低配版 NEMU.....

# Y-Emulator (YEMU) 设计与实现

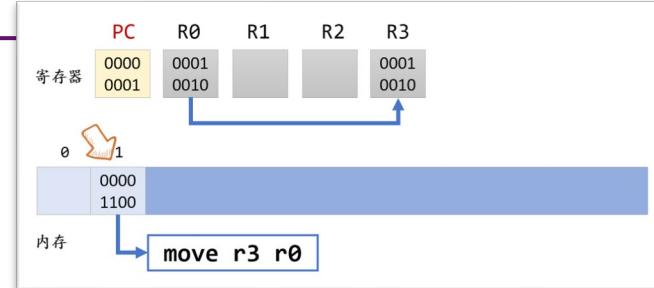
- 存储模型：内存 + 寄存器（包含PC）

- $16 + 5 = 21 \text{ bytes} = 168 \text{ bits}$
- 总共有 $2^{168}$ 种不同的取值状态
  - 任给一个状态，我们都能计算出PC处的指令，从而计算出下一个状态



# Y-Emulator (YEMU) 设计与实现

- 存储模型：内存 + 寄存器（包含PC）
  - $16 + 5 = 21 \text{ bytes} = 168 \text{ bits}$
  - 总共有 $2^{168}$ 种不同的取值状态
    - 任给一个状态，我们都能计算出PC处的指令，从而计算出下一个状态
- 理论上，任何计算机系统都是这样的状态机
  - $(M, R)$  构成计算机的状态
  - 32GiB内存有 $2^{274877906944}$ 种不同的状态 ( $= =$ )
    - $32 \times 1024 \times 1024 \times 1024 \times 8 \text{ bits}$
  - 每一个时钟周期，取出 $M[R[PC]]$ 的指令；执行；写回
    - 受制于物理实现(和功耗)的限制，通常每个时钟周期只能改变少量寄存器和内存的状态
    - 未来：(量子计算机颠覆了这个模型：同一时刻可以处于多个状态)



# YEMU: 模拟存储

- 存储是计算机能实现“计算”的重要基础
  - 寄存器 (PC)、内存
  - 这简单，用全局变量就好了！

```
#include <stdint.h>
#define NREG 4
#define NMEM 16
typedef uint8_t u8; // 没用过 uint8_t?
u8 pc = 0, R[NREG], M[NMEM] = { ... };
```

- 建议 STFW (C 标准库) → bool 有没有？
- 现代计算机系统: `uint8_t == unsigned char`
  - C Tips: 使用 `unsigned int` 避免潜在的 UB
    - `-fwrapv` 可以强制有符号整数溢出为 wraparound
  - C Quiz: 把指针转换成整数，应该用什么类型? (`int`) `&p`? `intptr_t`?

# 提升代码质量

- 还有更好地写法么？

```
#include <stdint.h>
#define NREG 4
#define NMEM 16
typedef uint8_t u8; // 没用过 uint8_t?
u8 pc = 0, R[NREG], M[NMEM] = { ... };
```

# 提升代码质量

- 给寄存器名字？

```
#define NREG 4
u8 R[NREG], pc; // 有些指令是用寄存器名描述的
#define RA 1 // BUG: 数组下标从0开始
...
...
```

```
enum { RA, R1, ..., PC };
u8 R[] = {
    [RA] = 0, // 这是什么语法? ?
    [R1] = 0,
    ...
    [PC] = init_pc,
};
#define pc (R[PC]) // 把 PC 也作为寄存器的一部分
#define NREG (sizeof(R) / sizeof(u8))
```

```
enum { RA, R1, ..., PC };
u8 R[] = { 0, 0, 0, 0, 0, ..., init_pc};
```

# 从一小段代码看软件设计

- 软件里有很多隐藏的 dependencies (一些额外的、代码中没有体现和约束的“规则”)
  - 一处改了，另一处忘了 (例如加了一个寄存器忘记更新 NREG...)
  - 减少 dependencies → 降低代码耦合程度

```
// breaks when adding a register
#define NREG 5 // 隐藏假设max{RA, RB, ... PC} == (NREG - 1)

// breaks when changing register size
#define NREG (sizeof(R) / sizeof(u8)) // 隐藏假设寄存器是8-bit

// never breaks
#define NREG (sizeof(R) / sizeof(R[0])) // 但需要R的定义

// even better (why?)
enum { RA, ... , PC, NREG }
```

# PA框架代码中的CPU\_state

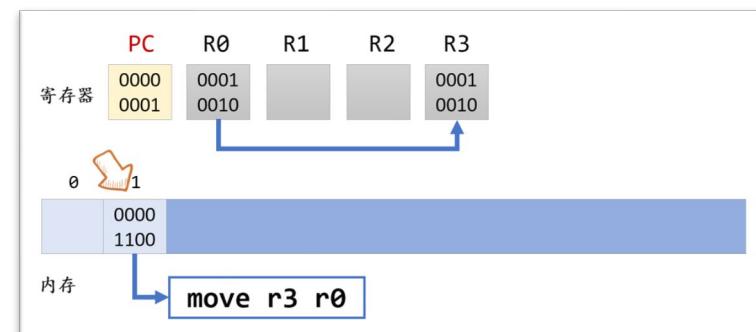
```
" Press ? for help
.. (up a dir)
~/ICS2021/ics2021/nemu/
> build/
> configs/
> include/
> resource/
> scripts/
> src/
>   > cpu/
>   > device/
>   > engine/
>   > isa/
>     > riscv32/
>       > difftest/
>     > include/
>       isa-all-instr.h
>       isa-def.h
  1 #ifndef __ISA_RISCV32_H__
  2 #define __ISA_RISCV32_H__
  3
  4 #include <common.h>
  5
  6 typedef struct {
  7   struct {
  8     rtlreg_t _32;
  9   } gpr[32];
 10
 11   vaddr_t pc;
 12 } riscv32_CPU_state;
 13
 14 // decode
 15 typedef struct {
 16   union {
 17     struct {
 18       uint32_t opcode1_0 : 2;
 19       uint32_t opcode6_2 : 5;
 20     }
 21     ...
 22   }
 23 } riscv32_OPCODE;
 24
 25 #endif
 26
 27 #ifndef __RISCV32_REG_H__
 28 #define __RISCV32_REG_H__
 29
 30 #include <common.h>
 31
 32 static inline int check_reg_idx(int idx) {
 33   IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && idx < 32));
 34   return idx;
 35 }
 36
 37 #define gpr(idx) (cpu.gpr[check_reg_idx(idx)]._32)
 38
 39 static inline const char* reg_name(int idx, int width) {
 40   extern const char* regs[];
 41   return regs[check_reg_idx(idx)];
 42 }
 43
 44 #endif
```

- 对于复杂的情况，struct/union 是更好的设计
  - 担心性能 (check\_reg\_idx)?
    - 在超强的编译器优化面前，不存在的

# YEMU

- 在时钟信号驱动下，根据( $M, R$ )更新系统状态
- RISC 处理器 (以及实际的 CISC 处理器实现):
  - 取指令 (fetch): 读出  $M[R[PC]]$  的一条指令
  - 译码 (decode): 根据指令集规范解析指令的语义 (顺便取出操作数)
  - 执行 (execute): 执行指令、运算后写回寄存器或内存
- 最重要的就是实现 `idx()`
  - 这就是 PA 里你们最挣扎的地方 (囊括了整个手册)

```
int main() {
    while (!is_halt(M[pc])) {
        idx();
    }
}
```

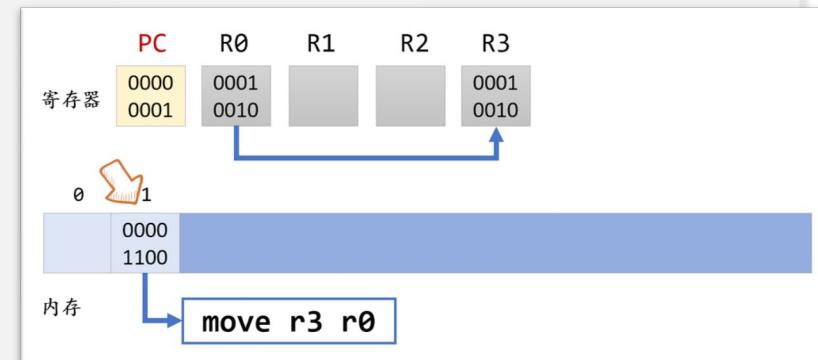
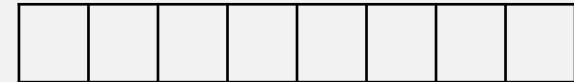


# 代码例子1

寄存器: PC, R0 (RA), R1, R2, R3 (8-bit)  
内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	][	rs
add	[0	0	0	1	[	rt	][	rs
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	

```
void idex() {
    if ((M[pc] >> 4) == 0) {
        R[(M[pc] >> 2) & 3] = R[M[pc] & 3];
        pc++;
    } else if ((M[pc] >> 4) == 1) {
        R[(M[pc] >> 2) & 3] += R[M[pc] & 3];
        pc++;
    } else if ((M[pc] >> 4) == 14) {
        R[0] = M[M[pc] & 0xf];
        pc++;
    } else if ((M[pc] >> 4) == 15) {
        M[M[pc] & 0xf] = R[0];
        pc++;
    }
}
```

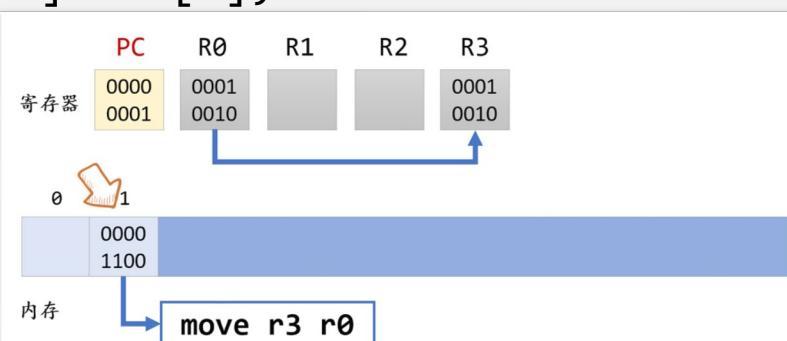


# 代码例子2

寄存器: PC, R0 (RA), R1, R2, R3 (8-bit)  
内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	][	rs
add	[0	0	0	1	[	rt	][	rs
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	

```
void index() {
    u8 inst = M[pc++];
    u8 op = inst >> 4;
    if (op == 0x0 || op == 0x1) {
        int rt = (inst >> 2) & 3, rs = (inst & 3);
        if (op == 0x0)          R[rt] = R[rs];
        else if (op == 0x1)     R[rt] += R[rs];
    }
    if (op == 0xe || op == 0xf) {
        int addr = inst & 0xf;
        if (op == 0xe) R[0] = M[addr];
        else if (op == 0xf) M[addr] = R[0];
    }
}
```



# 代码例子3 (YEMU代码)

```
typedef union inst {
```

```
    struct { u8 rs : 2, rt: 2, op: 4; } rtype;
```

```
    struct { u8 addr: 4,          op: 4; } mtype;
```

```
} inst_t;
```

```
#define RTYPE(i) u8 rt = (i)->rtype.rt, rs = (i)->rtype.rs;
```

```
#define MTYPE(i) u8 addr = (i)->mtype.addr;
```

```
void idex() {
```

```
    inst_t *cur = (inst_t *)&M[pc];
```

```
    switch (cur->rtype.op) {
```

```
        case 0b0000: { RTYPE(cur); R[rt] = R[rs]; pc++; break; }
```

```
        case 0b0001: { RTYPE(cur); R[rt] += R[rs]; pc++; break; }
```

```
        case 0b1110: { MTYPE(cur); R[RA] = M[addr]; pc++; break; }
```

```
        case 0b1111: { MTYPE(cur); M[addr] = R[RA]; pc++; break; }
```

```
        default: panic("invalid instruction at PC = %x", pc);
```

```
}
```

```
}
```

	7 6 5 4 3 2 1 0
mov	[0 0 0 0] [ rt ][ rs ]
add	[0 0 0 1] [ rt ][ rs ]
load	[1 1 1 0] [   addr   ]
store	[1 1 1 1] [   addr   ]

op	rt	rs
op	addr	

op	addr
----	------

# 有用的C语言特性

- Union / bit fields

```
typedef union inst {
    struct { u8 rs : 2, rt: 2, op: 4; } rtype;
    struct { u8 addr: 4,          op: 4; } mtype;
} inst_t;
```

- 指针

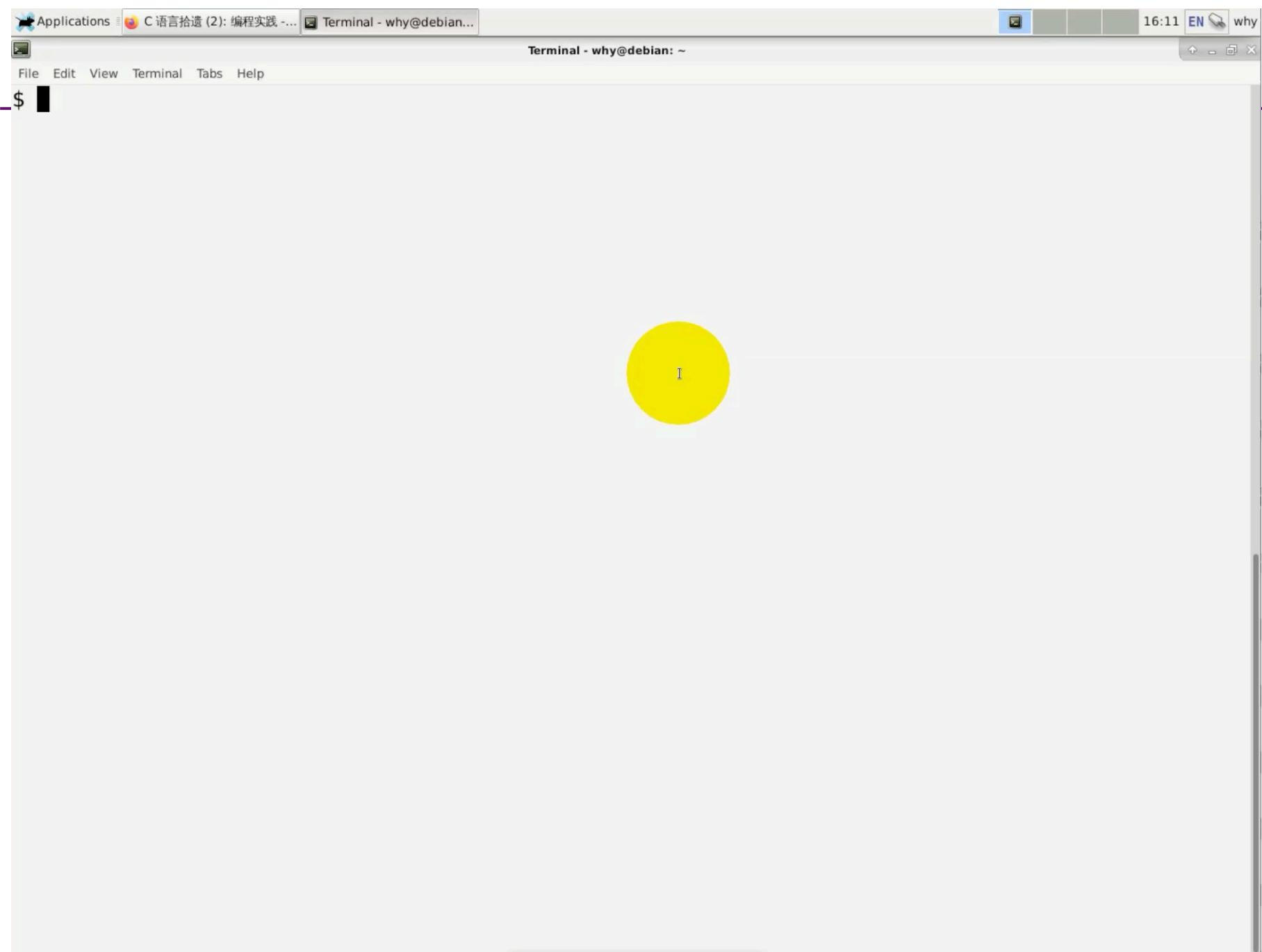
- 内存只是个字节序列
- 无论何种类型的指针都只是地址 + 对指向内存的解读

```
inst_t *cur = (inst_t *)&M[pc];
// cur -> rtype.op
// cur -> mtype.addr
```

```
14 // decode
15 typedef struct {
16     union {
17         struct {
18             uint32_t opcode1_0 : 2;
19             uint32_t opcode6_2 : 5;
20             uint32_t rd      : 5;
21             uint32_t funct3  : 3;
22             uint32_t rs1     : 5;
23             int32_t  imm11_0 : 12;
24         } i;
25         struct {
26             uint32_t opcode1_0 : 2;
27             uint32_t opcode6_2 : 5;
28             uint32_t imm4_0   : 5;
29             uint32_t funct3  : 3;
30             uint32_t rs1     : 5;
31             uint32_t rs2     : 5;
32             int32_t  imm11_5 : 7;
33         } s;
34         struct {
35             uint32_t opcode1_0 : 2;
36             uint32_t opcode6_2 : 5;
37             uint32_t rd       : 5;
38             uint32_t imm31_12 : 20;
39         } u;
40         uint32_t val;
41     } instr;
42 } riscv32_ISADecodeInfo;
43
```

```
// --- pattern matching wrappers for decode ---
#define INSTPAT(pattern, ...) do { \
    uint64_t key, mask, shift; \
    pattern_decode(pattern, STRLEN(pattern), &key, &mask, &shift); \
    if (((INSTPAT_INST(s) >> shift) & mask) == key) { \
        INSTPAT_MATCH(s, ##__VA_ARGS__); \
        goto *(__instpat_end); \
    } \
} while (0)
```

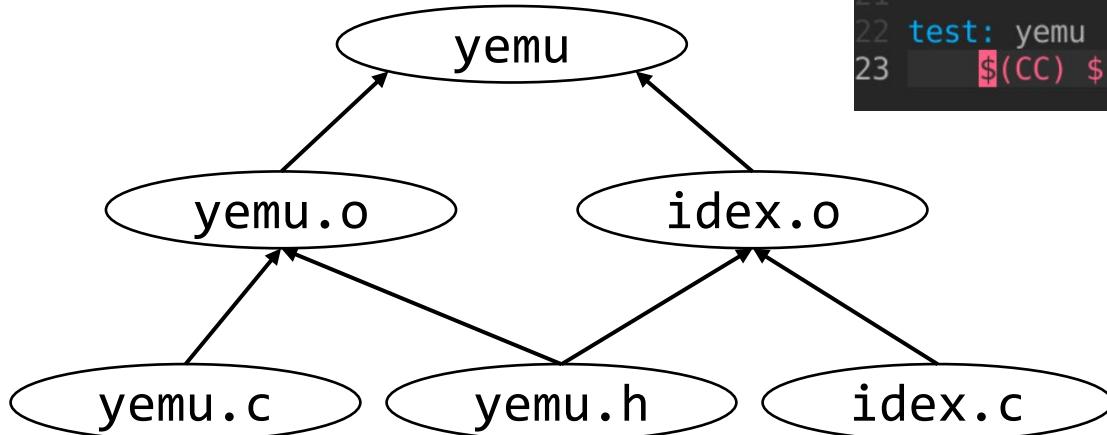
```
65  INSTPAT("??????? ???? ????? ?? ?????? 00101 11", auipc , U, R(dest) = src1 + s->pc);
66  INSTPAT("??????? ???? ????? 011 ????? 00000 11", ld     , I, R(dest) = Mr(src1 + src2, 8));
67  INSTPAT("??????? ???? ????? 011 ????? 01000 11", sd     , S, Mw(src1 + dest, 8, src2));
68
```



# Makefile

- make

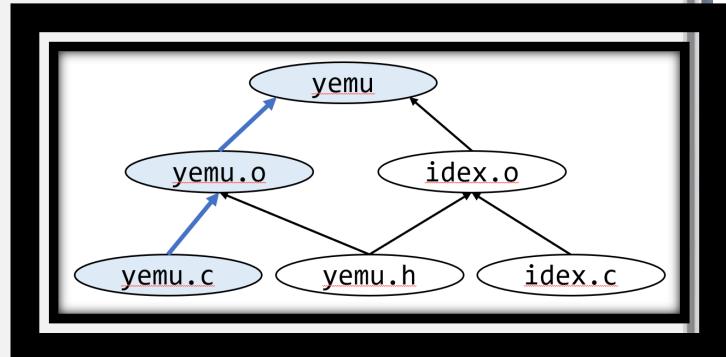
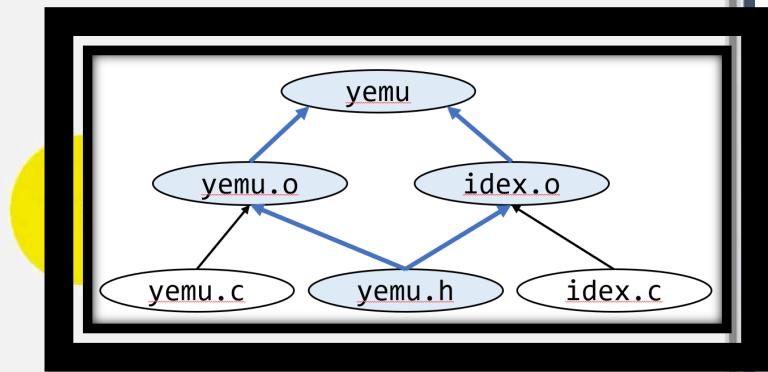
```
1 .PHONY: run clean test
2
3 CFLAGS = -Wall -Werror -std=c11 -O2
4 CC = gcc
5 LD = gcc
6
7 yemu: yemu.o idex.o
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o
9
10 yemu.o: yemu.c yemu.h
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c
12
13 idex.o: idex.c yemu.h
14     $(CC) $(CFLAGS) -c -o idex.o idex.c
15
16 run: yemu
17     ./yemu
18
19 clean:
20     rm -f test yemu *.o
21
22 test: yemu
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```



Applications Terminal - why@debian... 16:30 EN why

File Edit View Terminal Tabs Help

```
$ cd ICS_teach/
$ ls
yemu yemu.tar.gz
$ cd yemu/
$ ls
index.c index.o Makefile test.c yemu yemu.c yemu.h yemu.o
$ make clean
rm -f test yemu *.o
$
```



```
$ make run
```

```
Hit GOOD trap @ pc = 5.  
M[00] = 0xe7 (231)  
M[01] = 0x04 (4)  
M[02] = 0xe6 (230)  
M[03] = 0x11 (17)  
M[04] = 0xf8 (248)  
M[05] = 0x00 (0)  
M[06] = 0x10 (16)  
M[07] = 0x21 (33)  
M[08] = 0x31 (49)  
M[09] = 0x00 (0)  
M[10] = 0x00 (0)  
M[11] = 0x00 (0)  
M[12] = 0x00 (0)  
M[13] = 0x00 (0)  
M[14] = 0x00 (0)  
M[15] = 0x00 (0)  
$ vim Makefile
```

```
$ make test
```

```
gcc -Wall -Werror -std=c11 -O2 -o test idex.o test.c && ./test  
Test #1 (0b11100111): PASS  
Test #2 (0b00000100): PASS  
Test #3 (0b11100101): PASS  
Test #4 (0b00010001): PASS  
Test #5 (0b11110111): PASS
```

```
1 .PHONY: run clean test  
2  
3 CFLAGS = -Wall -Werror -std=c11 -O2  
4 CC = gcc  
5 LD = gcc  
6  
7 yemu: yemu.o idex.o  
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o  
9  
10 yemu.o: yemu.c yemu.h  
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c  
12  
13 idex.o: idex.c yemu.h  
14     $(CC) $(CFLAGS) -c -o idex.o idex.c  
15  
16 run: yemu  
17     ./yemu  
18  
19 clean:  
20     rm -f test yemu *.o  
21  
22 test: yemu  
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```

# make test

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include "yemu.h"
5
6 u8 R[NREG], M[NMEM], oldR[NREG], oldM[NMEM], *newM = M, *newR = R;
7
8 void random_rm() {
9     for (int i = 0; i < NREG; i++)
10        R[i] = rand() & 0xff;
11     for (int i = 0; i < NMEM; i++)
12        M[i] = rand() & 0xff;
13 }
14
15 #define ASSERT_EQ(a, b) ((u8)(a) == (u8)(b))
16
17 #define TESTCASES(_)
18     _ (1, 0b11100111, random_rm, ASSERT_EQ(newR[0], oldM[7])) \
19     _ (2, 0b00000100, random_rm, ASSERT_EQ(newR[1], oldR[0])) \
20     _ (3, 0b11100101, random_rm, ASSERT_EQ(newR[0], oldM[5])) \
21     _ (4, 0b00010001, random_rm, ASSERT_EQ(newR[0], oldR[0] + oldR[1])) \
22     _ (5, 0b11110111, random_rm, ASSERT_EQ(newM[7], oldR[0])) \
23
24 #define MAKETEST(id, inst, precond, postcond) { \
25     precond(); \
26     memcpy(oldM, M, NMEM); \
27     memcpy(oldR, R, NREG); \
28     pc = 0; M[pc] = inst; \
29     idex(); \
30     printf("Test #%d (%s): %s\n", \
31           id, #inst, (postcond) ? "PASS" : "FAIL"); \
32 }
33
34 int main() {
35     TESTCASES(MAKETEST)
36     return 0;
37 }
```

# 预编译的解析

- gcc -E test.c | indent - | vim -

```
#34 "test.c"
int
main ()
{
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b11100111;
    idex ();
    printf ("Test #%d (%s): %s\n", 1, "0b11100111",
            (((u8) (newR[0])) == (u8) (oldM[7]))) ? "PASS" : "FAIL");
}
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b00000100;
    idex ();
    printf ("Test #%d (%s): %s\n", 2, "0b00000100",
            (((u8) (newR[1])) == (u8) (oldR[0]))) ? "PASS" : "FAIL");
}
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b11100101;
    idex ();
    printf ("Test #%d (%s): %s\n", 3, "0b11100101",
            (((u8) (newR[0])) == (u8) (oldM[5]))) ? "PASS" : "FAIL");
}
```

# 小结

- 如何管理“更大”的项目(YEMU)?
  - 我们分多个文件管理它
    - yemu.h - 寄存器名; 必要的声明
    - yemu.c - 数据定义、主函数
    - idex.c - 译码执行
    - Makefile - 编译脚本(能实现增量编译)
- 使用合理的编程模式
  - 减少模块之间的依赖
    - enum { RA, ... , NREG }
  - 合理使用语言特性, 编写可读、可证明的代码
    - `inst_t *cur = (inst_t *)&M[pc]`
- NEMU 就是加强版的 YEMU

# 更多的计算机系统模拟器

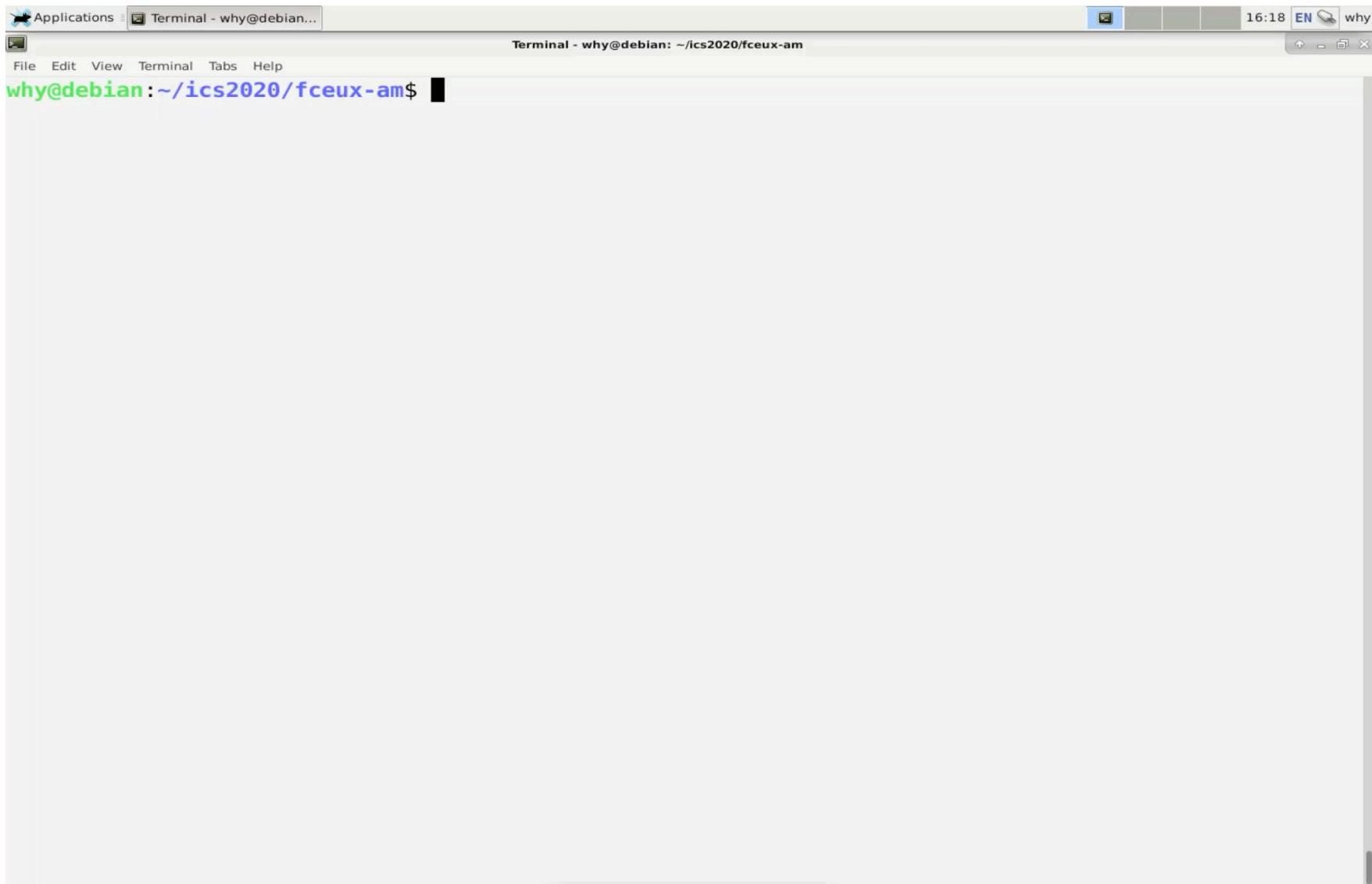
---

- `am-kernels/litenes`
  - 一个“最小”的 NES 模拟器
  - 自带硬编码的 ROM 文件
- `fceux-am`
  - 一个非常完整的高性能 NES 模拟器
  - 包含对卡带定制芯片的模拟 (`src/boards`)
- QEMU
  - 工业级的全系统模拟器
    - 2011 年发布 1.0 版本
    - 有兴趣的同学可以 RTFSC
  - 作者：传奇黑客 Fabrice Bellard

End.

永远不要停止对好代码的追求

# 用红白机模拟器NES Emulator玩Mario



# Git, GitHub与代码仓库

# 本讲概述

---

- Git, GitHub与代码仓库
  - Git选讲
  - Git在实验中的应用
- 项目的构建Make

# The Community Way

开源社区 (百度百科): 根据相应的开源软件许可证协议公布软件源代码的网络平台，同时也为网络成员提供一个自由学习交流的空间。

- 从GitHub获取代码
  - 传统工具链 + “现代” 编程体验
  - 有的时候网络不太靠谱



```
git clone -b 2022 git@github.com:NJU-ProjectN/ics-pa.git ics2022  
git clone https://github.com/NJU-ProjectN/ics-workbench
```



GitLab



<https://git.nju.edu.cn/>

# The Community Way (cont'd)

GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 50 million developers.

- **无所不能的代码聚集地**
  - 有整个计算机系统世界的代码
    - 硬件、操作系统、分布式系统、库函数、应用程序……
- **学习各种技术的最佳平台**
  - 海量的文档、学习资料、博客（新世界的大门）
    - 提供友好的搜索（例子：`awesome C`）

# 学习Git?

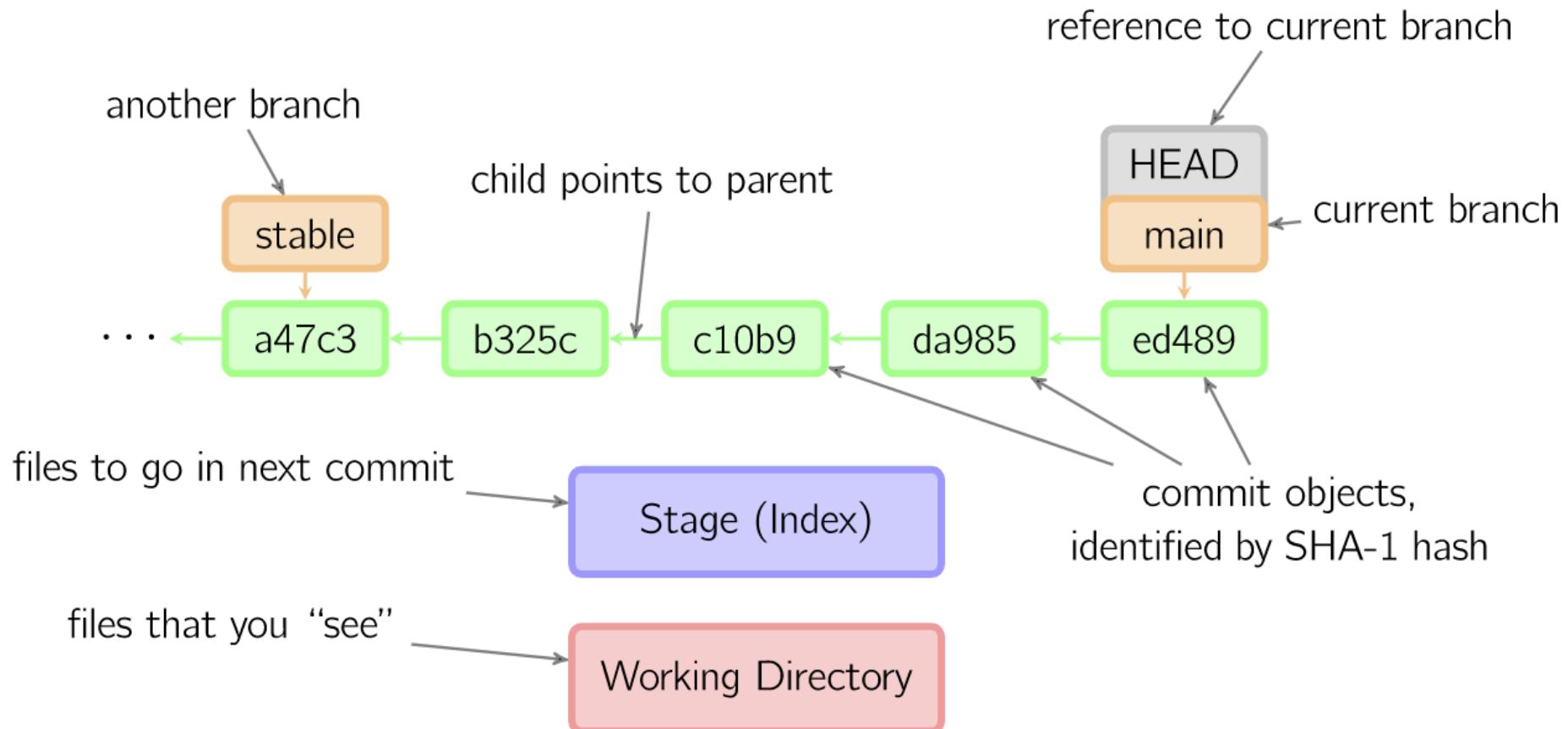
```
git clone -b 2021 https://github.com/NJU-ProjectN/ics-pa ics2021
```

内心独白：又要学新东西，我拒绝==

- RTFM?STFW!

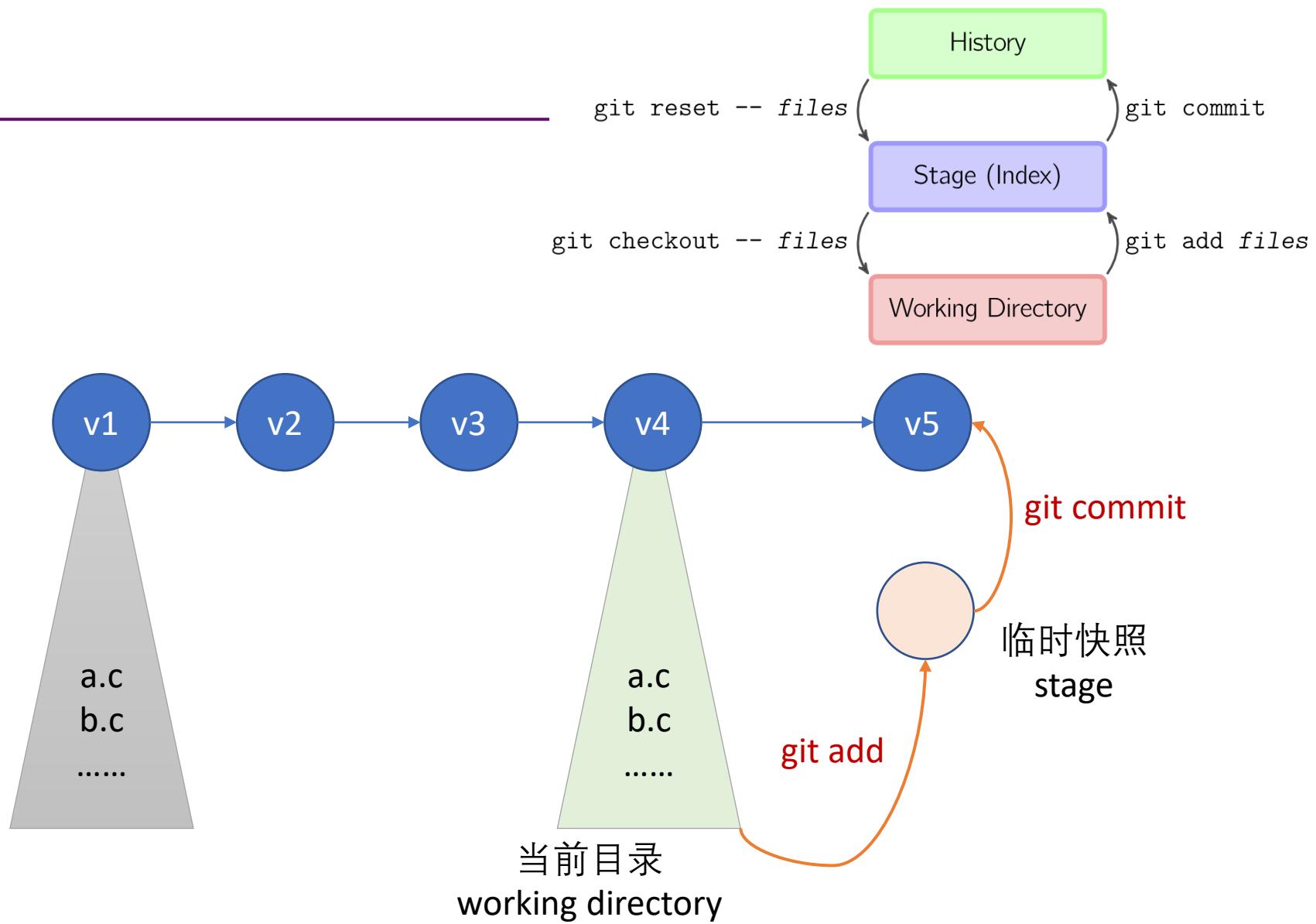
- 百度：得到一堆不太靠谱的教程
- 大家已经见识过**开源社区**的力量了
  - A Visual Git Reference
    - 英文、中文、日文.....
  - 好的文档是存在的
    - 还记得 tldr 吗？

# A Visual Git Reference

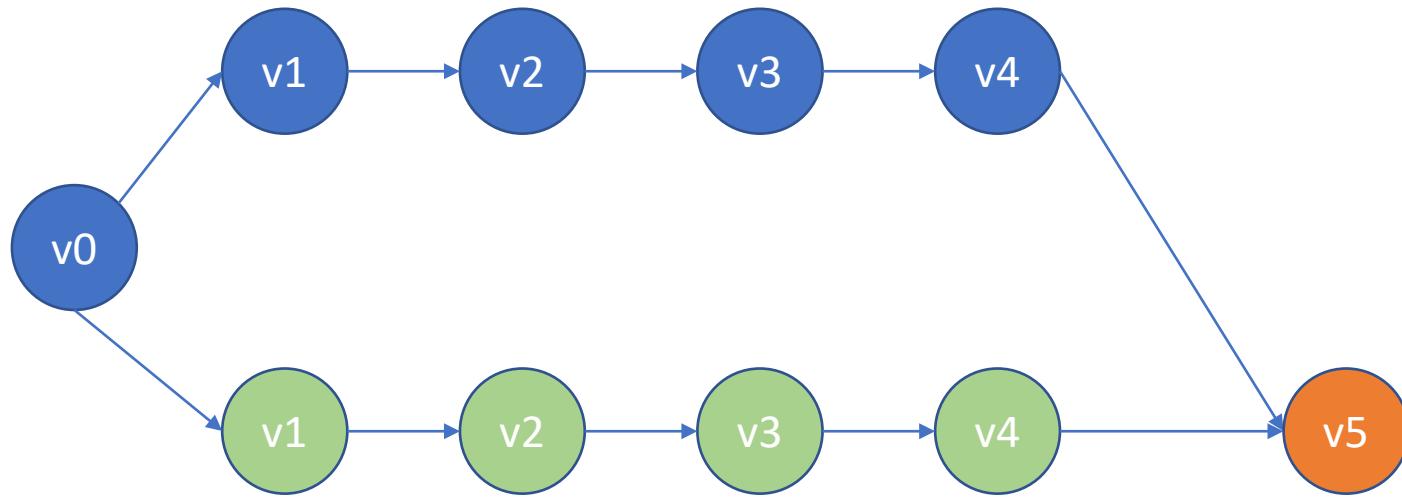


```
commit 8738b7b32e71a78f5b73376dc664780dd16800eb (HEAD -> pa1)
Author: tracer-ics2020 <tracer@njuics.org>
Date:   Thu Aug 19 18:27:15 2021 +0800
```

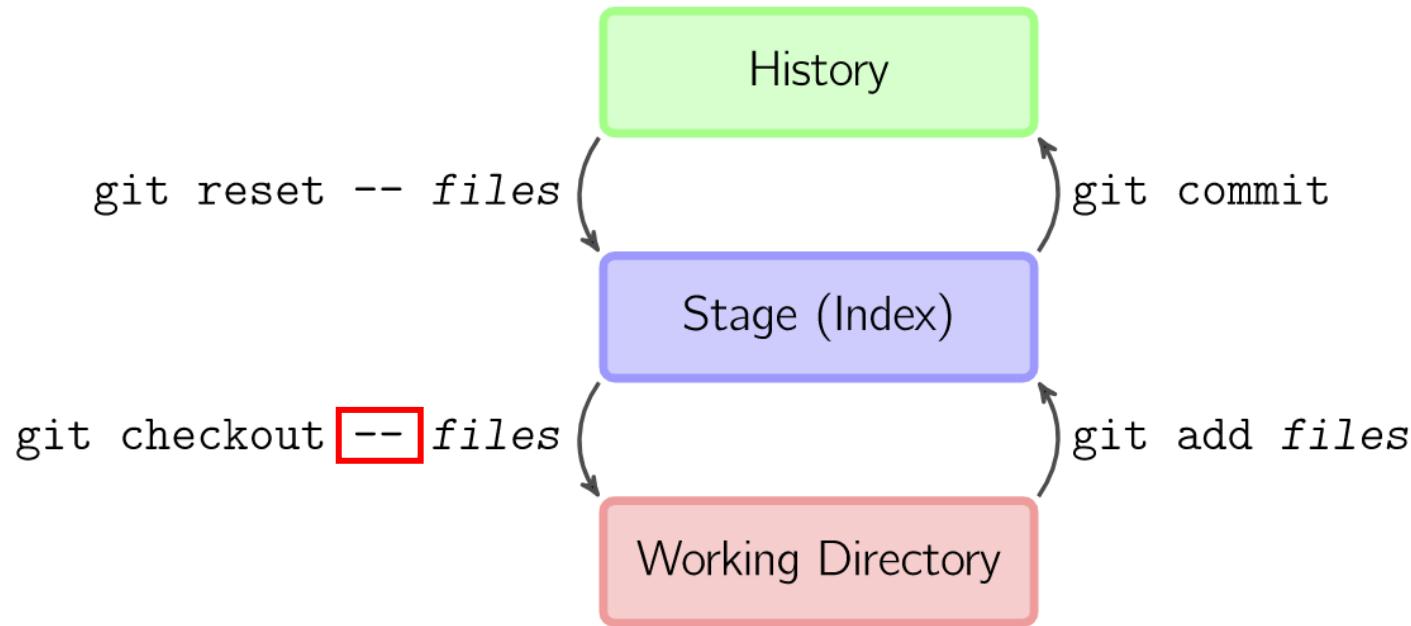
# Git



# Git：分布式版本控制系统



# A Visual Git Reference (cont'd)



```
$ ls -l
total 0
-rw-r--r-- 1 why why 0 Aug 25 18:10 -rf
$ rm -- -rf
$ ls -l
total 0
```

# 一些Comments

---

- 体验 Git
  - 创建一个新的 repo, 自由探索
    - 为什么 “预计完成时间 XX 小时” 是骗人的?
      - 预计完成时间是假设你在大一开始就用 Git 的
  - Visualizing Git Concepts with D3

# Visualizing Git Concepts with D3

← → ⌂ ⚠ 不安全 | onlywei.github.io/explain-git-with-d3/#commit

Visualizing Git Concepts with D3

This website is designed to help you understand some basic git concepts visually. This is my first attempt at using both SVG and D3. I hope it is helpful to you.

Adding/staging your files for commit will not be covered by this site. In all sandbox playgrounds on this site, just pretend that you always have files staged and ready to commit at all times. If you need a refresher on how to add or stage files for commit, please read [Git Basics](#).

Sandboxes are split by specific git commands, listed below.

Basic Commands	Undo Commits	Combine Branches	Remote Server
<a href="#">git commit</a>	<a href="#">git reset</a>	<a href="#">git merge</a>	<a href="#">git fetch</a>
<a href="#">git branch</a>	<a href="#">git revert</a>	<a href="#">git rebase</a>	<a href="#">git push</a>
			<a href="#">git pull</a>
			<a href="#">git tag</a>

Fork me on GitHub

We are going to skip instructing you on how to add your files for commit in this explanation. Let's assume you already know how to do that. If you don't, go read some other tutorials.

Pretend that you already have your files staged for commit and enter `git commit` as many times as you like in the terminal box.

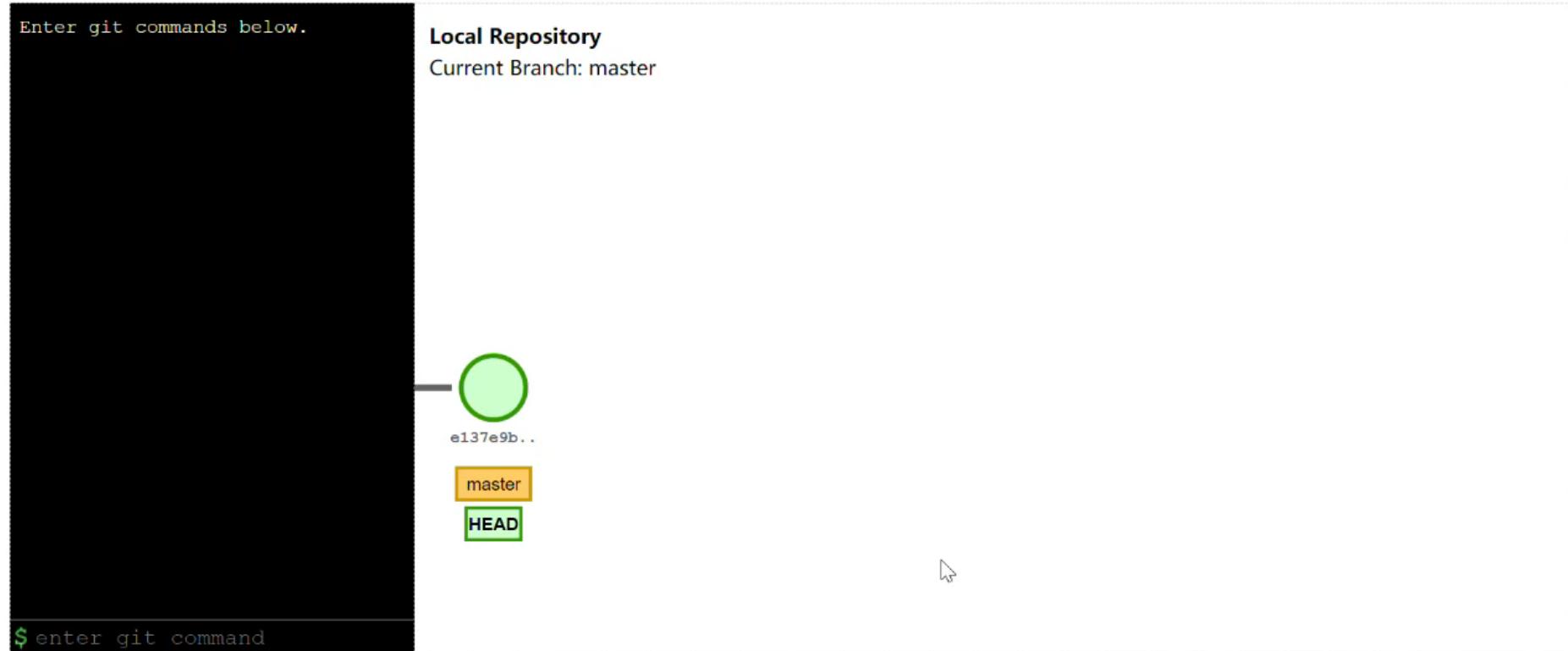
## Specific Examples

Below I have created some specific real-world scenarios that I feel are quite common and useful.

[Restore Local Branch to State on Origin Server](#)  
[Update Private Local Branch with Latest from Origin](#)  
[Deleting Local Branches](#)

[Free Playground](#)  
[Zen Mode](#)

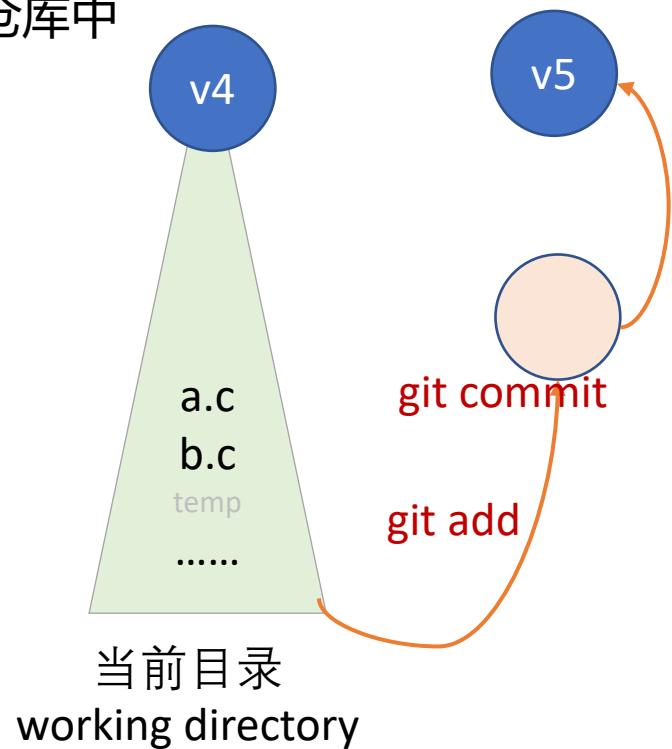
# Visualizing Git Concepts with D3



# 一些Comments (cont'd)

- 我们使用了“白名单” .gitignore文件
  - 只在Git repo里管理.c, .h和Makefile
    - 基本原则：一切生成的文件都不放在Git仓库中

```
*          # 忽略一切文件  
!*/        # 除了目录  
!* .c      # .c  
!* .h      # ...  
!Makefile*  
.gitignore
```



- 为什么ls看不到这个文件?
  - 怎么还有一个.git

```
git clone -b 2021 https://github.com/NJU-ProjectN/ics-pa ics2021
```

# 获取PA代码

```
git clone -b 2022 git@github.com:NJU-ProjectN/ics-pa.git ics2022
```

- 获取来自github的一整个历史

# 提交脚本

- make submit会下载执行<http://jyywiki.cn/static/submit.sh>
  - Makefile

```
submit:
```

```
git gc  
STUID=$(STUID) STUNAME=$(STUNAME) bash -c “$(curl -s  
http://why.ink:8080/static/submit.sh)”
```



```
bash -c “$(curl -s http://why.ink:8080/static/submit.sh)”
```

- 我们做了什么?

```
why@why-VirtualBox:~/Documents/ICS2022/ics2022$ curl -s http://why.ink:8080/static/submit.sh  
COURSE=ICS2022  
MODULE=$(git rev-parse --abbrev-ref HEAD | tr '[a-z]' '[A-Z]')  
FILE=/tmp/upload.tar.bz2  
tar caf "$FILE" ".git" $(find . -maxdepth 2 -name "*.pdf") && \  
curl -F "token=$TOKEN" \  
    -F "course=$COURSE" \  
    -F "module=$MODULE" \  
    -F "file=@$FILE" \  
http://why.ink:8080/upload
```

# 提交脚本

- make submit会下载执行<http://why.ink:8080/static/submit.sh>

```
bash -c "$(curl -s http://why.ink:8080/static/submit.sh)"
```



```
# submit.sh (服务器)
```

```
COURSE=ICS2023
```

```
MODULE=$(git rev-parse --abbrev-ref HEAD | tr '[a-z]' '[A-Z]')
FILE=/tmp/upload.tar.bz2
```

**PA1?PA2?**

```
$ git branch
  master
* pa0
  pa1
$ git rev-parse --abbrev-ref HEAD
pa0
$ git checkout pa1
Switched to branch 'pa1'
$ git rev-parse --abbrev-ref HEAD
pa1
```

```
tar caf "$FILE" "$NAME/.git" $(find $NAME -maxdepth 2 -name "*.pdf") && \
curl -F "token=$TOKEN"
      -F "course=$COURSE"
      -F "module=$MODULE" \
      -F "file=@$FILE" http://why.ink:8080/upload
```

# OJ得到的是.git

- .git包含了OJ想知道的一切

```
1 ./git  
2 ./git/objects  
3 ./git/objects/info  
4 ./git/objects/pack  
5 ./git/hooks  
6 ./git/info  
7 ./git/info/exclude  
8 ./git/HEAD  
9 ./git/branches  
10 ./git/description  
11 ./git/refs  
12 ./git/refs/heads  
13 ./git/refs/tags  
14  
15 ./git/config # 配置
```

# Git追踪

- 另一段神秘代码 (来自 Makefile.lab)

```
git:  
@git add $(shell find . -name "*.c") \  
$(shell find . -name "*.h") -A --ignore-errors  
@while (test -e .git/index.lock); do sleep 0.1; done  
@(hostnamectl && uptime) | \  
git commit -F - -q --author=tracer-nju <tracer@nju.edu.cn>  
--no-verify --allow-empty @sync
```

- 在每次 make 执行时
  - git 目标都会被执行
  - 将 .c 和 .h 添加、强制提交到 Git repo

```
$ hostnamectl  
Static hostname: why-VirtualBox  
Icon name: computer-vm  
Chassis: vm  
Machine ID: 22e7c7921a0a437bb1817612c8ab2cce  
Boot ID: 51bcab304005479b8dcec150413ee8bf  
Virtualization: oracle  
Operating System: Ubuntu 21.04  
Kernel: Linux 5.11.0-34-generic  
Architecture: x86-64  
$  
$ uptime  
17:25:29 up 1:45, 1 user, load average: 0.11, 0.09, 0.02
```

# Git追

```
6 GITFLAGS = -q --author='tracer-ics2021 <tracer@njuics.org>' --no-verify --allow-empty
7
8 # prototype: git commit(msg)
9 define git_commit
10   -@git add $(NEMU_HOME)/.. -A --ignore-errors
11   -@while (test -e .git/index.lock); do sleep 0.1; done
12   -@(echo "> $(1)" && echo $(STUID) && hostnamectl && uptime) | git commit -F - $(GITFLAGS)
13   -@sync
14 endef
```

```
commit 2662595b8712e2247fd1a92b5cb8b9103c9fc01b
Author: tracer-ics2021 <tracer@njuics.org>
Date: Thu Sep 16 12:07:21 2021 +0800

    > compile
ky2107911
        Static hostname: why-VirtualBox
        Icon name: computer-vm
        Chassis: vm
        Machine ID: 22e7c7921a0a437bb1817612c8ab2cce
        Boot ID: 6cbe073460244ae0ad90966cdbbe5961
        Virtualization: oracle
        Operating System: Ubuntu 21.04
            Kernel: Linux 5.11.0-34-generic
            Architecture: x86-64
12:07:21 up 2:40, 2 users, load average: 0.23, 0.23, 0.19

commit 713801b46b12063c27db811bd61a9b935395cd35
Author: tracer-ics2021 <tracer@njuics.org>
Date: Thu Sep 16 12:06:39 2021 +0800

    > run
ky2107911
        Static hostname: why-VirtualBox
        Icon name: computer-vm
        Chassis: vm
        Machine ID: 22e7c7921a0a437bb1817612c8ab2cce
        Boot ID: 6cbe073460244ae0ad90966cdbbe5961
        Virtualization: oracle
        Operating System: Ubuntu 21.04
            Kernel: Linux 5.11.0-34-generic
            Architecture: x86-64
12:06:39 up 2:39, 2 users, load average: 0.46, 0.26, 0.20
```

# 这是什么？

- 对抄袭代码的一种威慑
  - 如何抓抄袭
    - 旧方法；但至今仍在使用
  - Git 追踪是 “一劳永逸的新方法”
    - 透过它，我们看到南大学子的人生百态



我们不生产~~水~~，我们只是~~大自然的搬运工~~  
代码 互联网



农夫山泉®  
NONGFU SPRING

代码抄袭：那些让985学生沉默，211学生流泪的真相



蒋炎岩 ✨  
计算机科学与技术博士

2,221 人赞同了该文章

这是我们在TURC' 18 (SIGCSE China)的论文Needle: Detecting Code Plagiarism on Student Submissions [1]的科普版本。

“我们不生产代码，我们只是互联网的搬运工” ——佚名

在平均学历是985的知乎，考不上个好学校都没脸出来冒泡。然而，到底是我们上了大学，还是大学上了我们？这篇文章来谈谈一个985学生知道了会沉默，211学生知道了会流泪的真相：广泛存在的抄作业问题。我想大部分读者朋友们看到这里的感受都是：中枪了。



心态崩了

你还记得上大学的时候，把“大腿”的作业拿来抄抄改改，然后去打游戏的日子吗？或者如果你就是同学们心目中的“大腿”，有没有慷慨相助同学们的经历？也许这类事情的确没有发生在你身上，但在计算机学科里，我们严肃的研究发现：

- 某985大学的一次实验大作业中，若不对代码抄袭控制，有超过82%的同学涉嫌拷贝代码(抄袭或被抄袭)，并且抄袭者会对代码做出不同程度的修改。大家也都知道，计算机专业课程光说不练课可就白上了，也就是说，在一届同学中有超过4/5这门课都白上了。考虑到还有同学选择不交作业，这个比例还会再高一些.....
- 在一所985大学的一门课程中，即便使用了抄袭检测工具、制定严格的惩罚措施、给有困难的同学提供“诚信分”，一门课程整个学期下来，依然有约20%的同学涉嫌拷贝代码。甚至有同学被逮到以后还会再抄、再被逮到以后再抄代码，铤而走险，屡教不改。

(以上结论仅针对我们调研的编程实验有效，而且导致同学们抄代码的原因是多种多样的，整体的代码抄袭情况我们尚未完全统计。)

# 这是什么？

---

- 对抄袭代码的一种威慑
  - 如何抓抄袭
    - 旧方法；但至今仍在使用
    - Git 追踪是“一劳永逸的新方法”
      - 透过它，我们看到南大学子的人生百态
- 记得 academic integrity
  - 我们留了足够多的分数给努力尝试过的同学通过
- 未来可能有终极加强版

# 小结：现代化的项目管理方式

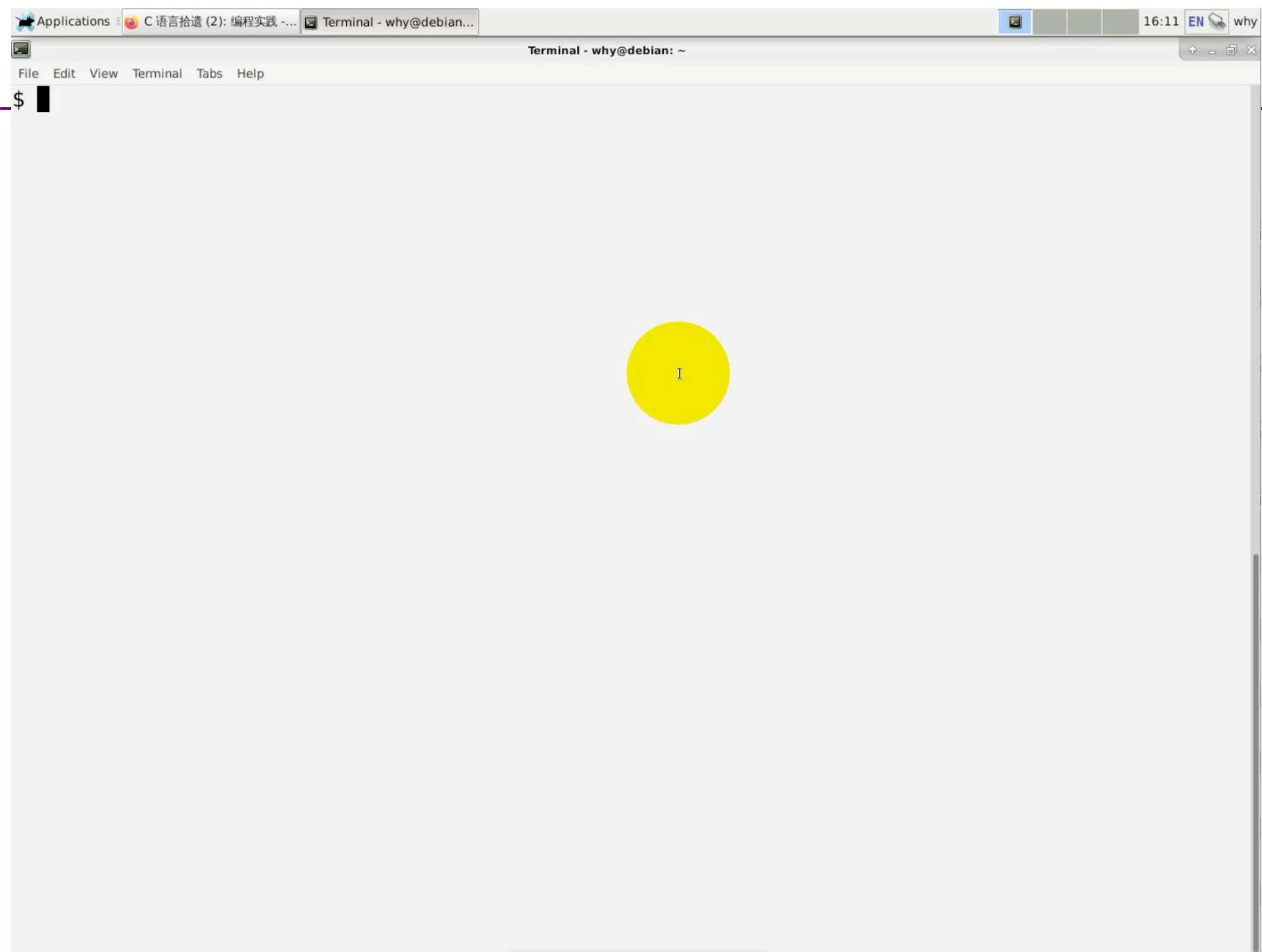
- Git: 代码快照管理工具
  - 是一种 “可持久化数据结构”
  - 拓展阅读: Pro Git
- 框架代码中的两处非常规 Git 使用
  - 提交脚本
    - 仅上传 .git; 在服务器执行 git reset
    - 减少提交大小 (仅源文件)
  - Git 追踪
    - 编译时强制提交, 获取同学编码的过程
- 思考题: 如何管理自己的代码快照?
  - 提示: 分支/HEAD/... 只是指向快照的指针 (references)

# 项目构建

# Make工具

- 回顾：YEMU 模拟器
- Makefile 是一段 “declarative” 的代码
  - 描述了构建目标之间的依赖关系和更新方法

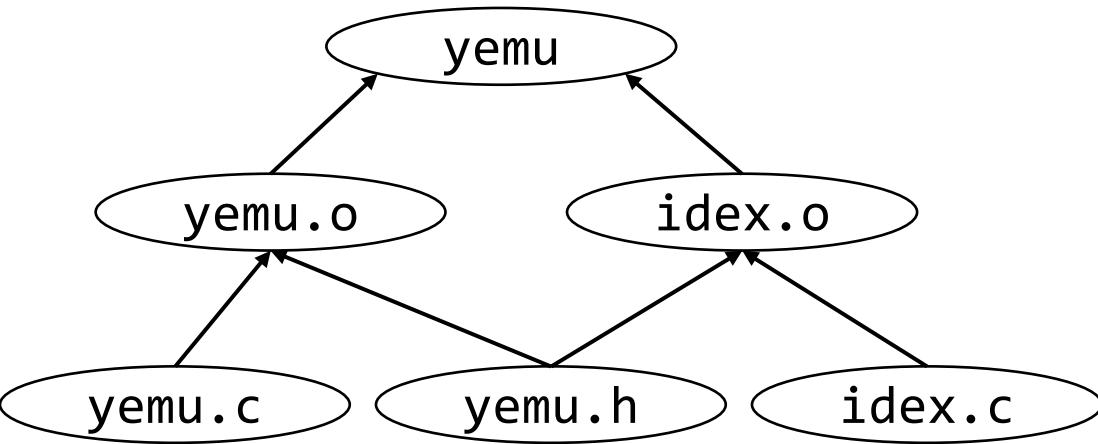
```
1 .PHONY: run clean test
2
3 CFLAGS = -Wall -Werror -std=c11 -O2
4 CC = gcc
5 LD = gcc
6
7 yemu: yemu.o idex.o
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o
9
10 yemu.o: yemu.c yemu.h
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c
12
13 idex.o: idex.c yemu.h
14     $(CC) $(CFLAGS) -c -o idex.o idex.c
15
16 run: yemu
17     ./yemu
18
19 clean:
20     rm -f test yemu *.o
21
22 test: yemu
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```



# Makefile

- make

- gcc -Wall -Werror -std=c11 -O2 -c -o yemu.o yemu.c
- gcc -Wall -Werror -std=c11 -O2 -c -o idex.o idex.c
- gcc -o yemu yemu.o idex.o

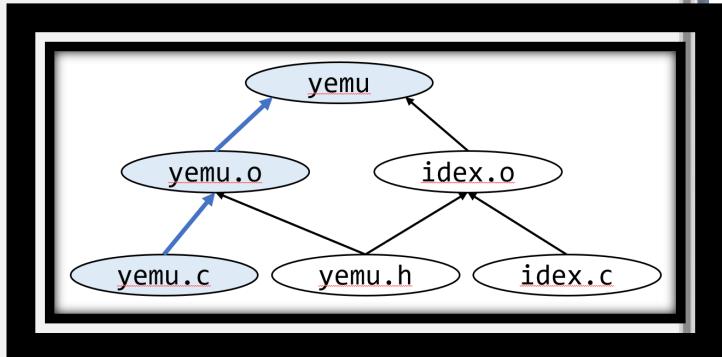
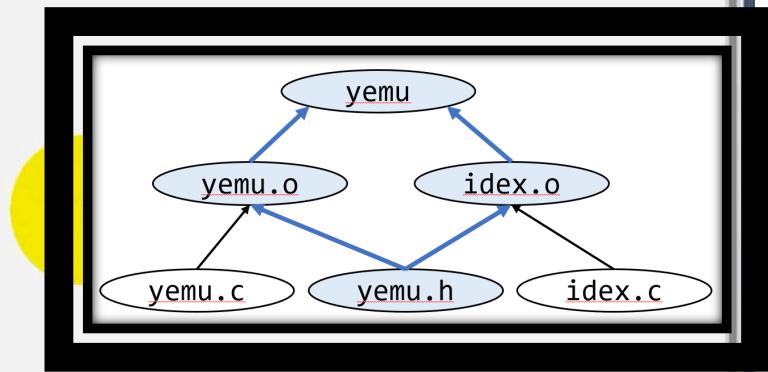


```
1 .PHONY: run clean test
2
3 CFLAGS = -Wall -Werror -std=c11 -O2
4 CC = gcc
5 LD = gcc
6
7 yemu: yemu.o idex.o
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o
9
10 yemu.o: yemu.c yemu.h
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c
12
13 idex.o: idex.c yemu.h
14     $(CC) $(CFLAGS) -c -o idex.o idex.c
15
16 run: yemu
17     ./yemu
18
19 clean:
20     rm -f test yemu *.o
21
22 test: yemu
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```

Applications Terminal - why@debian... 16:30 EN why

File Edit View Terminal Tabs Help

```
$ cd ICS_teach/
$ ls
yemu yemu.tar.gz
$ cd yemu/
$ ls
index.c index.o Makefile test.c yemu yemu.c yemu.h yemu.o
$ make clean
rm -f test yemu *.o
$
```



```
$ make run
```

```
Hit GOOD trap @ pc = 5.  
M[00] = 0xe7 (231)  
M[01] = 0x04 (4)  
M[02] = 0xe6 (230)  
M[03] = 0x11 (17)  
M[04] = 0xf8 (248)  
M[05] = 0x00 (0)  
M[06] = 0x10 (16)  
M[07] = 0x21 (33)  
M[08] = 0x31 (49)  
M[09] = 0x00 (0)  
M[10] = 0x00 (0)  
M[11] = 0x00 (0)  
M[12] = 0x00 (0)  
M[13] = 0x00 (0)  
M[14] = 0x00 (0)  
M[15] = 0x00 (0)  
$ vim Makefile
```

```
$ make test
```

```
gcc -Wall -Werror -std=c11 -O2 -o test idex.o test.c && ./test  
Test #1 (0b11100111): PASS  
Test #2 (0b00000100): PASS  
Test #3 (0b11100101): PASS  
Test #4 (0b00010001): PASS  
Test #5 (0b11110111): PASS
```

```
1 .PHONY: run clean test  
2  
3 CFLAGS = -Wall -Werror -std=c11 -O2  
4 CC = gcc  
5 LD = gcc  
6  
7 yemu: yemu.o idex.o  
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o  
9  
10 yemu.o: yemu.c yemu.h  
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c  
12  
13 idex.o: idex.c yemu.h  
14     $(CC) $(CFLAGS) -c -o idex.o idex.c  
15  
16 run: yemu  
17     ./yemu  
18  
19 clean:  
20     rm -f test yemu *.o  
21  
22 test: yemu  
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```

# make test

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include "yemu.h"
5
6 u8 R[NREG], M[NMEM], oldR[NREG], oldM[NMEM], *newM = M, *newR = R;
7
8 void random_rm() {
9     for (int i = 0; i < NREG; i++)
10        R[i] = rand() & 0xff;
11     for (int i = 0; i < NMEM; i++)
12        M[i] = rand() & 0xff;
13 }
14
15 #define ASSERT_EQ(a, b) ((u8)(a) == (u8)(b))
16
17 #define TESTCASES(_)
18     _ (1, 0b11100111, random_rm, ASSERT_EQ(newR[0], oldM[7])) \
19     _ (2, 0b00000100, random_rm, ASSERT_EQ(newR[1], oldR[0])) \
20     _ (3, 0b11100101, random_rm, ASSERT_EQ(newR[0], oldM[5])) \
21     _ (4, 0b00010001, random_rm, ASSERT_EQ(newR[0], oldR[0] + oldR[1])) \
22     _ (5, 0b11110111, random_rm, ASSERT_EQ(newM[7], oldR[0])) \
23
24 #define MAKETEST(id, inst, precond, postcond) { \
25     precond(); \
26     memcpy(oldM, M, NMEM); \
27     memcpy(oldR, R, NREG); \
28     pc = 0; M[pc] = inst; \
29     idex(); \
30     printf("Test #%d (%s): %s\n", \
31           id, #inst, (postcond) ? "PASS" : "FAIL"); \
32 }
33
34 int main() {
35     TESTCASES(MAKETEST)
36     return 0;
37 }
```

# 预编译的解析

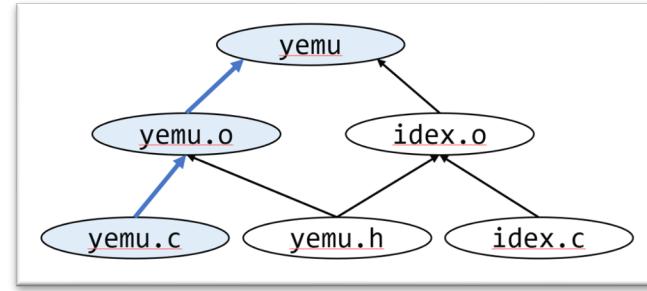
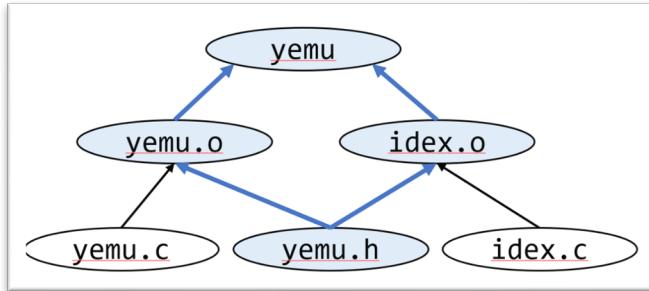
- gcc -E test.c | indent - | vim -

```
#34 "test.c"
int
main ()
{
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b11100111;
    idex ();
    printf ("Test #%d (%s): %s\n", 1, "0b11100111",
            (((u8) (newR[0])) == (u8) (oldM[7]))) ? "PASS" : "FAIL");
}
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b00000100;
    idex ();
    printf ("Test #%d (%s): %s\n", 2, "0b00000100",
            (((u8) (newR[1])) == (u8) (oldR[0]))) ? "PASS" : "FAIL");
}
{
    random_rm ();
    memcpy (oldM, M, 16);
    memcpy (oldR, R, NREG);
    (R[PC]) = 0;
    M[(R[PC])] = 0 b11100101;
    idex ();
    printf ("Test #%d (%s): %s\n", 3, "0b11100101",
            (((u8) (newR[0])) == (u8) (oldM[5]))) ? "PASS" : "FAIL");
}
{
```

# Make工具

- 回顾：YEMU 模拟器
- Makefile 是一段 “declarative”的代码
  - 描述了构建目标之间的依赖关系和更新方法

```
$ make  
gcc -Wall -Werror -std=c11 -O2 -c -o yemu.o yemu.c  
gcc -Wall -Werror -std=c11 -O2 -c -o idex.o idex.c  
gcc -o yemu yemu.o idex.o  
$ make  
make: 'yemu' is up to date.
```



- make -j8

# Make工具

---

- 回顾： YEMU 模拟器
- Makefile 是一段 “declarative” 的代码
  - 描述了构建目标之间的依赖关系和更新方法
  - 有用的编译选项
    - -nB、 -j
  - 同时也是和 Shell 结合紧密的编程语言
    - 能够生成各种字符串
    - 支持 “元编程” (#include, #define, ...)

# Lab代码的构建

- 顶层 (top-level) Makefile:

```
# := -> C #define
NAME := $(shell basename $(PWD))
export MODULE := Lab1
```

```
# 变量 -> 字面替换
all: $(NAME)-64 $(NAME)-32
```

```
# include -> C #include
include ../Makefile
```

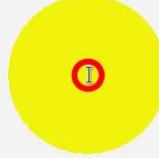
```
1 # **DO NOT MODIFY**
2
3 ifeq ($(NAME),)
4 $(error Should make in each lab's directory)
5 endif
6
7 SRCS := $(shell find . -maxdepth 1 -name "*.c")
8 DEPS := $(shell find . -maxdepth 1 -name "*.h") $(SRCS)
9 CFLAGS += -O1 -std=gnu11 -ggdb -Wall -Werror -Wno-unused-result -Wno-
unused-value -Wno-unused-variable
10
11 .PHONY: all git test clean commit-and-make
12
13 .DEFAULT_GOAL := commit-and-make
14 commit-and-make: git all
15
16 $(NAME)-64: $(DEPS) # 64bit binary
17     gcc -m64 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
18
19 $(NAME)-32: $(DEPS) # 32bit binary
20     gcc -m32 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
21
22 $(NAME)-64.so: $(DEPS) # 64bit shared library
23     gcc -fPIC -shared -m64 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
24
25 $(NAME)-32.so: $(DEPS) # 32bit shared library
26     gcc -fPIC -shared -m32 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
27
28 clean:
29     rm -f $(NAME)-64 $(NAME)-32 $(NAME)-64.so $(NAME)-32.so
30
31 include ../Makefile.lab
```

Applications Lab1: 大整数运算 - Mozilla... Terminal - why@debian... 18:11 拼写助手

File Edit View Terminal Tabs Help

Terminal - why@debian: ~/ICS\_teach/ics-workbench/multimod

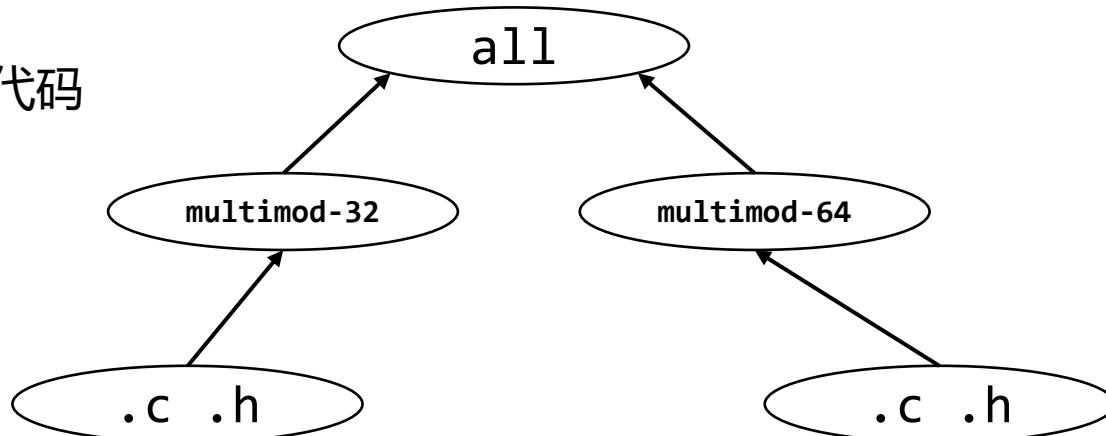
```
$ ls
main.c Makefile multimod-32 multimod-64 multimod.c
$ vim Makefile
$ vim Makefile
```



A yellow circle with a red dot in the center. The red dot contains the letter 'I'.

# Lab代码的构建 (cont'd)

- 构建目标
  - 总目标
    - .DEFAULT\_GOAL := commit-and-make
    - commit-and-make: git all (all 在顶层 Makefile 中定义)
  - 可执行文件
    - multimod-64: gcc -m64
    - multimod-32: gcc -m32
  - 共享库 (之后的 lab 使用)
    - multimod-64.so: gcc -fPIC -shared -m64
    - multimod-32.so: gcc -fPIC -shared -m32
  - clean
    - 删除构建的代码



# NEMU代码构建

- Makefile 真复杂
  - 放弃?

```
$ make
+ CC src/nemu-main.c
+ CC src/isa/riscv32/difftest/dut.c
+ CC src/isa/riscv32/system/intr.c
+ CC src/isa/riscv32/system/mmio.c
+ CC src/isa/riscv32/init.c
+ CC src/isa/riscv32/logo.c
+ CC src/isa/riscv32/reg.c
+ CC src/isa/riscv32/instr/decode.c
+ CC src/device/io/map.c
+ CC src/device/io/mmio.c
+ CC src/device/io/port-io.c
+ CC src/engine/interpreter/hostcall.c
+ CC src/engine/interpreter/init.c
+ CC src/cpu/difftest/dut.c
+ CC src/cpu/cpu-exec.c
+ CC src/monitor/sdb/watchpoint.c
+ CC src/monitor/sdb/sdb.c
+ CC src/monitor/sdb/expr.c
+ CC src/monitor/monitor.c
+ CC src/utils/log.c
+ CC src/utils/rand.c
+ CC src/utils/state.c
+ CC src/utils/timer.c
+ CC src/memory/paddr.c
+ CC src/memory/vaddr.c
+ LD /home/why/Documents/ICS2021/ics2021/nemu/build/riscv32-nemu-interpret
```

```
1 # Sanity check
2 ifeq ($(wildcard $(NEMU_HOME)/src/nemu-main.c),)
3   $(error NEMU_HOME=$(NEMU_HOME) is not a NEMU repo)
4 endif
5
6 # Include variables and rules generated by menuconfig
7 -include $(NEMU_HOME)/include/config/auto.conf
8 -include $(NEMU_HOME)/include/config/auto.conf.cmd
9
10 remove_quote = $(patsubst "%",%,$(1))
11
12 # Extract variables from menuconfig
13 GUEST_ISA ?= $(call remove_quote,$(CONFIG_ISA))
14 ENGINE ?= $(call remove_quote,$(CONFIG_ENGINE))
15 NAME    = $(GUEST_ISA)-nemu-$(ENGINE)
16
17 # Include all filelist.mk to merge file lists
18 FILELIST_MK = $(shell find ./src -name "filelist.mk")
19 include $(FILELIST_MK)
20
21 # Filter out directories and files in blacklist to obtain the final set of source files
22 DIRS_BLACKLIST-y += $(DIRS_BLACKLIST)
23 SRCS_BLACKLIST-y += $(SRCS_BLACKLIST) $(shell find $(DIRS_BLACKLIST-y) -name "*.c")
24 SRCS-y += $(shell find $(DIRS-y) -name "*.c")
25 SRCS = $(filter-out $(SRCS_BLACKLIST-y),$(SRCS-y))
26
27 # Extract compiler and options from menuconfig
28 CC = $(call remove_quote,$(CONFIG_CC))
29 CFLAGS_BUILD += $(call remove_quote,$(CONFIG_CC_OPT))
30 CFLAGS_BUILD += $(if $(CONFIG_CC_LTO),-flto,)
31 CFLAGS_BUILD += $(if $(CONFIG_CC_DEBUG),-ggdb3,)
32 CFLAGS_BUILD += $(if $(CONFIG_CC_ASAN),-fsanitize=address,)
33 CFLAGS += $(CFLAGS_BUILD) -D_GUEST_ISA_=$(GUEST_ISA)
34 LDFLAGS += $(CFLAGS_BUILD)
35
36 # Include rules for menuconfig
37 include $(NEMU_HOME)/scripts/config.mk
38
39 ifdef CONFIG_TARGET_AM
40 include $(AM_HOME)/Makefile
41 LINKAGE += $(ARCHIVES)
42 else
43 # Include rules to build NEMU
44 include $(NEMU_HOME)/scripts/native.mk
45 endif
```

~/Documents/ICS2021/ics2021/nemu/Makefile[1]

# 总结

# 关于《计算机系统基础》习题课

- 教会大家 “计算机的正确打开方式”
  - 编程 ≠ 闷头写代码
  - 使用工具也是编程的一部分
    - version-control systems: git, svn, ...
    - build systems: make, cmake (C++), maven (Java), ...
    - shell: bash, zsh, ...
- 基本原则：任何感到不爽的事情都一定有工具能帮你
  - 如果真的没有，自己造一个的就会就来了
    - (不太可能是真的)
    - 但这将会是一份非常好的研究工作

End.

(RTFM & RTFSC)

# NEMU 框架选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 提醒

PA1:

Deadline: 2023年10月15日 23:59:59

Lab1:

Deadline: 2023年10月22日 23:59:59

框架代码已上线

# 本讲概述

---

- NEMU编译运行

- 浏览构建文件
- 模拟构建过程

- NEMU代码导读

- 浏览源代码
- 启动代码选讲
- 编辑器配置

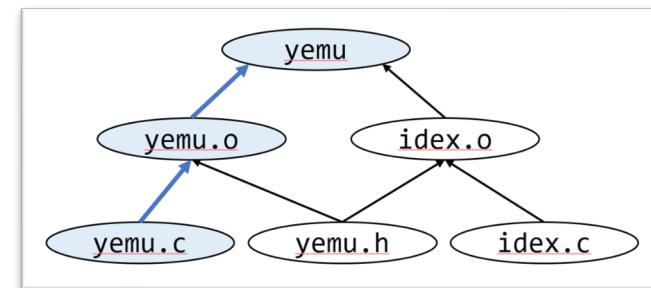
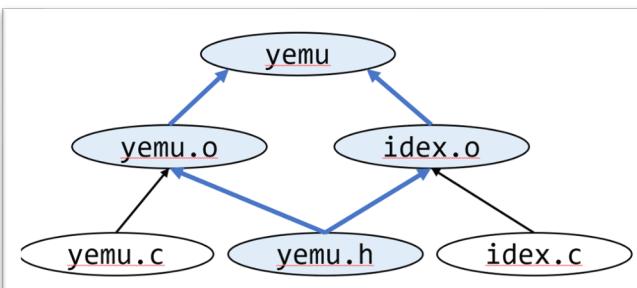
# Make工具

- 回顾：YEMU 模拟器

- Makefile 是一段 “declarative”的代码
  - 描述了构建目标之间的依赖关系和更新方法

```
1 .PHONY: run clean test
2
3 CFLAGS = -Wall -Werror -std=c11 -O2
4 CC = gcc
5 LD = gcc
6
7 yemu: yemu.o idex.o
8     $(LD) $(LDFLAGS) -o yemu yemu.o idex.o
9
10 yemu.o: yemu.c yemu.h
11     $(CC) $(CFLAGS) -c -o yemu.o yemu.c
12
13 idex.o: idex.c yemu.h
14     $(CC) $(CFLAGS) -c -o idex.o idex.c
15
16 run: yemu
17     ./yemu
18
19 clean:
20     rm -f test yemu *.o
21
22 test: yemu
23     $(CC) $(CFLAGS) -o test idex.o test.c && ./test
```

```
$ make
gcc -Wall -Werror -std=c11 -O2 -c -o yemu.o yemu.c
gcc -Wall -Werror -std=c11 -O2 -c -o idex.o idex.c
gcc -o yemu yemu.o idex.o
$ make
make: 'yemu' is up to date.
```



# NEMU代码构建

- Makefile 真复杂
  - 放弃?

```
$ make
+ CC src/nemu-main.c
+ CC src/isa/riscv32/difftest/dut.c
+ CC src/isa/riscv32/system/intr.c
+ CC src/isa/riscv32/system/mmio.c
+ CC src/isa/riscv32/init.c
+ CC src/isa/riscv32/logo.c
+ CC src/isa/riscv32/reg.c
+ CC src/isa/riscv32/instr/decode.c
+ CC src/device/io/map.c
+ CC src/device/io/mmio.c
+ CC src/device/io/port-io.c
+ CC src/engine/interpreter/hostcall.c
+ CC src/engine/interpreter/init.c
+ CC src/cpu/difftest/dut.c
+ CC src/cpu/cpu-exec.c
+ CC src/monitor/sdb/watchpoint.c
+ CC src/monitor/sdb/sdb.c
+ CC src/monitor/sdb/expr.c
+ CC src/monitor/monitor.c
+ CC src/utils/log.c
+ CC src/utils/rand.c
+ CC src/utils/state.c
+ CC src/utils/timer.c
+ CC src/memory/paddr.c
+ CC src/memory/vaddr.c
+ LD /home/why/Documents/ICS2021/ics2021/nemu/build/riscv32-nemu-interpret
```

```
1 # Sanity check
2 ifeq ($(wildcard $(NEMU_HOME)/src/nemu-main.c),)
3   $(error NEMU_HOME=$(NEMU_HOME) is not a NEMU repo)
4 endif
5
6 # Include variables and rules generated by menuconfig
7 -include $(NEMU_HOME)/include/config/auto.conf
8 -include $(NEMU_HOME)/include/config/auto.conf.cmd
9
10 remove_quote = $(patsubst "%",%,$(1))
11
12 # Extract variables from menuconfig
13 GUEST_ISA ?= $(call remove_quote,$(CONFIG_ISA))
14 ENGINE ?= $(call remove_quote,$(CONFIG_ENGINE))
15 NAME    = $(GUEST_ISA)-nemu-$(ENGINE)
16
17 # Include all filelist.mk to merge file lists
18 FILELIST_MK = $(shell find ./src -name "filelist.mk")
19 include $(FILELIST_MK)
20
21 # Filter out directories and files in blacklist to obtain the final set of source files
22 DIRS_BLACKLIST-y += $(DIRS_BLACKLIST)
23 SRCS_BLACKLIST-y += $(SRCS_BLACKLIST) $(shell find $(DIRS_BLACKLIST-y) -name "*.c")
24 SRCS-y += $(shell find $(DIRS-y) -name "*.c")
25 SRCS = $(filter-out $(SRCS_BLACKLIST-y),$(SRCS-y))
26
27 # Extract compiler and options from menuconfig
28 CC = $(call remove_quote,$(CONFIG_CC))
29 CFLAGS_BUILD += $(call remove_quote,$(CONFIG_CC_OPT))
30 CFLAGS_BUILD += $(if $(CONFIG_CC_LTO),-flto,)
31 CFLAGS_BUILD += $(if $(CONFIG_CC_DEBUG),-ggdb3,)
32 CFLAGS_BUILD += $(if $(CONFIG_CC_ASAN),-fsanitize=address,)
33 CFLAGS += $(CFLAGS_BUILD) -D_GUEST_ISA_=$(GUEST_ISA)
34 LDFLAGS += $(CFLAGS_BUILD)
35
36 # Include rules for menuconfig
37 include $(NEMU_HOME)/scripts/config.mk
38
39 ifdef CONFIG_TARGET_AM
40 include $(AM_HOME)/Makefile
41 LINKAGE += $(ARCHIVES)
42 else
43 # Include rules to build NEMU
44 include $(NEMU_HOME)/scripts/native.mk
45 endif
```

~/Documents/ICS2021/ics2021/nemu/Makefile[1]

# Make工具

---

- 回顾： YEMU 模拟器
- Makefile 是一段 “declarative” 的代码
  - 描述了构建目标之间的依赖关系和更新方法
  - 有用的编译选项
    - -nB、 -j
  - 同时也是和 Shell 结合紧密的编程语言
    - 能够生成各种字符串
    - 支持 “元编程” (#include, #define, ...)

# 还是尝试去读一下

管理 控制 视图 热键 设备 帮助

```
why@why-VirtualBox:~/Documents/ICS2021/ics2021/nemu$ ls
build configs include Kconfig Makefile README.md resource scripts src tools
why@why-VirtualBox:~/Documents/ICS2021/ics2021/nemu$ vim Kconfig
why@why-VirtualBox:~/Documents/ICS2021/ics2021/nemu$
```

```
1 # Automatically generated file; DO NOT EDIT.
2 # NEMU Configuration Menu
3 #
4 #
5 CONFIG_DIFFTEST_REF_NAME="none"
6 CONFIG_ENGINE="interpreter"
7 CONFIG_PC_RESET_OFFSET=0
8 CONFIG_TARGET_NATIVE_ELF=y
9 CONFIG_MSIZE=0x8000000
10 CONFIG_CC_O2=y
11 CONFIG_MODE_SYSTEM=y
12 CONFIG_MEM_RANDOM=y
13 CONFIG_ISA_riscv32=y
14 CONFIG_LOG_START=0
15 CONFIG_MBASE=0x80000000
16 CONFIG_TIMER_GETTIMEofday=y
17 CONFIG_ENGINE_INTERPRETER=y
18 CONFIG_CC_OPT="-O2"
19 CONFIG_LOG_END=10000
20 CONFIG_RT_CHECK=y
21 CONFIG_CC="gcc"
22 CONFIG_DIFFTEST_REF_PATH="none"
23 CONFIG_CC_DEBUG=y
24 CONFIG_CC_GCC=y
25 CONFIG_DEBUG=y
26 CONFIG_ISA="riscv32"
```

```
~/Documents/ICS2021/ics2021/nemu/include/config/auto.conf[+1]
```

```
[conf] unix utf-8 Ln 1, Col 0/26
```

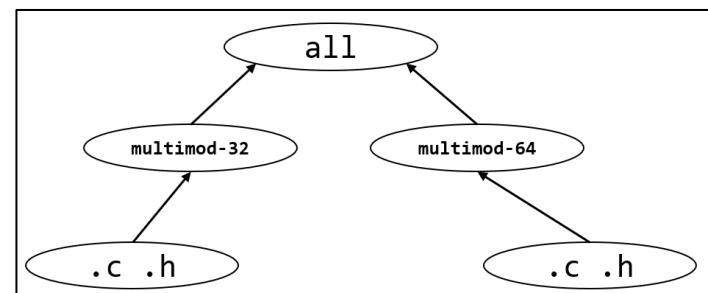
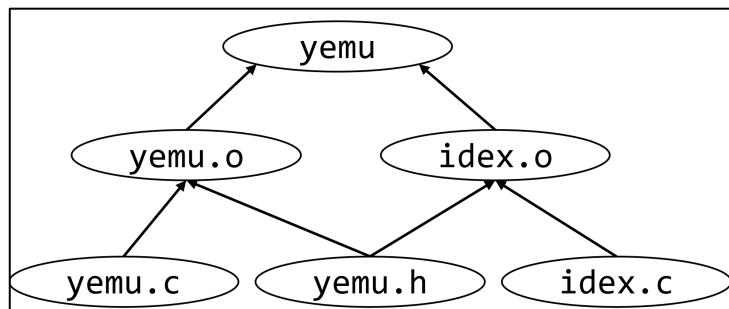
```
:q
```

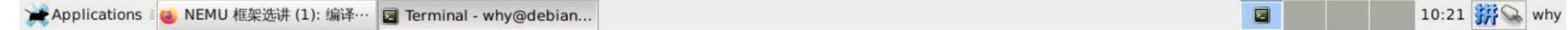
```
"why - Vi 英 韩 拼 22-9月 -21
```



# NEMU代码构建

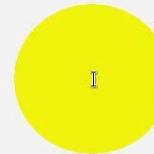
- Makefile 真复杂
  - 放弃?
- 一个小诀窍
  - 先观察 make 命令实际执行了什么 (trace)





Terminal - why@debian: ~/ics2021/nemu

```
$ ls
configs  include  Kconfig  Makefile  README.md  resource  scripts  src  tools
$ █
```



```

1 .DEFAULT_GOAL = app
2
3 # Add necessary options if the target is a shared library
4 ifdef SHARE
5 SO = -SO
6 CFLAGS += -fPIC
7 LDFLAGS += -rdynamic -shared -fPIC
8 endif
9
10 WORK_DIR = $(shell pwd)
11 BUILD_DIR = $(WORK_DIR)/build
12
13 INC_PATH := $(WORK_DIR)/include $(INC_PATH)
14 OBJ_DIR = $(BUILD_DIR)/obj-$(NAME)$(SO)
15 BINARY = $(BUILD_DIR)/$(NAME)$(SO)
16
17 CC ?= gcc
18
19 # Compilation flags
20 CC := $(CC)
21 LD := $(CC)
22 INCLUDES = $(addprefix -I, $(INC_PATH))
23 CFLAGS := -O2 -MMD -Wall -Werror $(INCLUDES) $(CFLAGS)
24 LDFLAGS := -O2 $(LDFLAGS)
25
26 OBJS = $(SRCS:.c=$(OBJ_DIR)/%.o)
27
28 # Compilation patterns
29 $(OBJ_DIR)/%.o: %.c
30   @echo + CC $<
31   @mkdir -p $(dir $@)
32   @$(CC) $(CFLAGS) -c -o $@ $<
33   $(call call_fixdep, $(@:o=d), $@)
34
35 # Dependencies
36 -include $(OBJS:.o=.d)
37
38 # Some convenient rules
39
40 .PHONY: app clean
41
42 app: $(BINARY)
43
44 $(BINARY): $(OBJS) $(ARCHIVES)
45   @echo + LD $@
46   @$(LD) -o $@ $(OBJS) $(LDFLAGS) $(ARCHIVES) $LIBS
47
48 clean:
49   -rm -rf $(BUILD_DIR)
~/Documents/ICS2021/ics2021/nemu/scripts/build.mk[2] [make] unix utf-8 Ln 46, Col 53/49
-- INSERT --
[0] 0:vim*

```

Applications NEMU 框架选讲 (1): 编译... Terminal - why@debian...

10:22 100% why

File Edit View Terminal Tabs Help

uthor='tracer-ics2021 <tracer@njuics.org>' --no-verify --allow-empty sync

echo + LD /home/why/ics2021/nemu/build/riscv32-nemu-interpreter

gcc -o /home/why/ics2021/nemu/build/riscv32-nemu-interpreter /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/nemu-main.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/engine/interpreter/init.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/engine/interpreter/hostcall.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/init.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/system/intr.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/system/mmu.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/instr/decode.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/logo.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/isa/riscv32/reg.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/cpu/cpu-exec.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/cpu/difftest/dut.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/monitor/monitor.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/monitor/sdb/watchpoint.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/monitor/sdb/sdb.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/monitor/sdb/expr.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/utils/log.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/utils/state.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/utils/timer.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/utils/rand.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/memory/paddr.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/memory/vaddr.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/device/io/map.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/device/io/mmio.o /home/why/ics2021/nemu/build/obj-riscv32-nemu-interpreter/src/device/io/port-io.o -O2 -fPIE -lreadline -ldl -pie

\$

\$ █

# NEMU代码构建

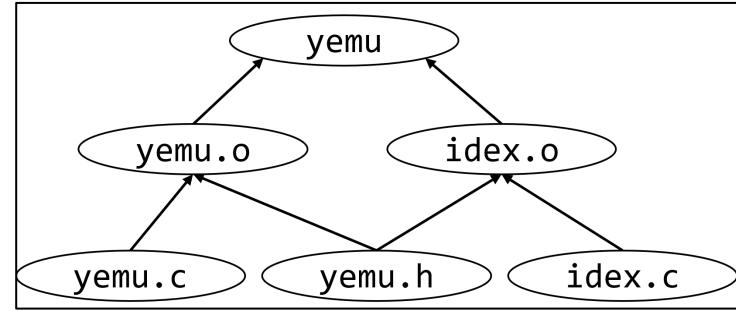
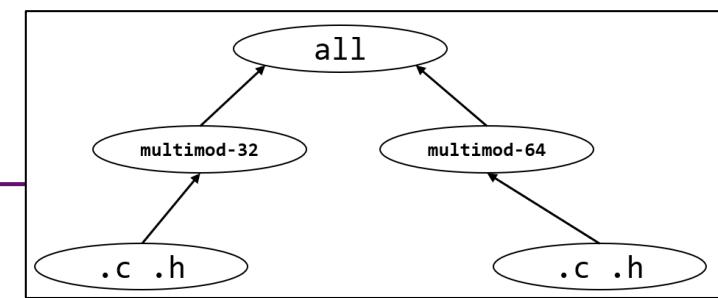
- Makefile 真复杂

- 放弃?

- 一个小诀窍

- 先观察 make 命令实际执行了什么 (trace)
  - RTFM/STFW: make 提供的两个有用的选项
    - -n 只打印命令不运行
    - -B 强制 make 所有目标

```
make -nB \
| grep -ve '^(\#\|echo\|mkdir\)' \
| vim -
```



```

1 .DEFAULT_GOAL = app
2
3 # Add necessary options if the target is a shared library
4 ifndef SHARE
5 SO = -SO
6 CFLAGS += -fPIC
7 LDFLAGS += -rdynamic -shared -fPIC
8 endif
9
10 WORK_DIR = $(shell pwd)
11 BUILD_DIR = $(WORK_DIR)/build
12
13 INC_PATH := $(WORK_DIR)/include $(INC_PATH)
14 OBJ_DIR = $(BUILD_DIR)/obj-$(NAME)$(SO)
15 BINARY = $(BUILD_DIR)/$(NAME)$(SO)
16
17 CC ?= gcc
18
19 # Compilation flags
20 CC := $(CC)
21 LD := $(CC)
22 INCLUDES = $(addprefix -I, $(INC_PATH))
23 CFLAGS := -O2 -MMD -Wall -Werror $(INCLUDES) $(CFLAGS)
24 LDFLAGS := -O2 $(LDFLAGS)
25
26 OBJS = $(SRCS:.c=$(OBJ_DIR)/%.o)
27
28 # Compilation patterns
29 $(OBJ_DIR)/%.o: %.c
30   @echo + CC $<
31   @mkdir -p $(dir $@)
32   @$(CC) $(CFLAGS) -c -o $@ $<
33   $(call call_fixdep, $(@:o=d), $@)
34
35 # Dependencies
36 -include $(OBJS:.o=.d)
37
38 # Some convenient rules
39
40 .PHONY: app clean
41
42 app: $(BINARY)
43
44 $(BINARY): $(OBJS) $(ARCHIVES)
45   @echo + LD $@
46   @$(LD) -o $@ $(OBJS) $(LDFLAGS) $(ARCHIVES) $LIBS
47
48 clean:
49   -rm -rf $(BUILD_DIR)
~/Documents/ICS2021/ics2021/nemu/scripts/build.mk[2] [make] unix utf-8 Ln 46, Col 53/49
-- INSERT --
[0] 0:vim*

```

Terminal - Makefile (~/ics2021/nemu) - VIM

```

19 include $(FILELIST_MK)
20
21 # Filter out directories and files in blacklist to obtain the final
22 DIRS-BLACKLIST-y += $(DIRS-BLACKLIST)
23 SRCS-BLACKLIST-y += $(SRCS-BLACKLIST) $(shell find $(DIRS-BLACKLIST-y) -name "*.c")
24 SRCS-y += $(shell find $(DIRS-y) -name "*.c")
25 SRCS = $(filter-out $(SRCS-BLACKLIST-y),$(SRCS-y))
26
27 # Extract compiler and options from menuconfig
28 CC = $(call remove_quote,$(CONFIG_CC))
29 CFLAGS_BUILD += $(call remove_quote,$(CONFIG_CC_OPT))
30 CFLAGS_BUILD += $(if $(CONFIG_CC廖0),-fllto,)
31 CFLAGS_BUILD += $(if $(CONFIG_CC_DEBUG),-ggdb3,)
32 CFLAGS_BUILD += $(if $(CONFIG_CC_ASAN),-fsanitize=address,)
33 CFLAGS += $(CFLAGS_BUILD) -D__GUEST_ISA__=$(GUEST_ISA)
34 LDFLAGS += $(CFLAGS_BUILD)
35
36 # Include rules for menuconfig

```

Terminal - native.mk (~/ics2021/nemu/scripts) - VIM

```

1 include $(NEMU_HOME)/scripts/git.mk
2 include $(NEMU_HOME)/scripts/build.mk
3
4 include $(NEMU_HOME)/tools/difftest.mk
5
6 compile_git:
7     $(call git_commit, "compile")
8 $(BINARY): compile_git
9
10 # Some convenient rules
11
12 override ARGS ?= --log=$(BUILD_DIR)/nemu-log.txt
13 override ARGS += $(ARGS_DIFF)
14
15 # Command to execute NEMU
16 IMG ?=
17 NEMU_EXEC := $(BINARY) $(ARGS) $(IMG)
18
19 run-env: $(BINARY) $(DIFF_REF_S0)
20
21 run: run-env
22     $(call git_commit, "run")
23     $(NEMU_EXEC)
24
25 gdb: run-env
26     $(call git_commit, "gdb")
27     gdb -s $(BINARY) --args $(NEMU_EXEC)

```

scripts/native.mk  
scripts/native.mk" 35L, 760C

File Edit View Terminal Tabs Help

Terminal - config.mk (~/ics2021/nemu/scripts) - VIM

```

26 $(FIXDEP):
27     $(Q)$($MAKE) $(silent) -C $(FIXDEP_PATH)
28
29 menuconfig: $(MCONF) $(CONF) $(FIXDEP)
30     $(Q)$($MCONF) $(Kconfig)
31     $(Q)$($CONF) $(silent) --syncconfig $(Kconfig)
32
33 savedefconfig: $(CONF)
34     $(Q)$< $(silent) --@=configs/defconfig $(Kconfig)
35
36 %defconfig: $(CONF) $(FIXDEP)
37     $(Q)$< $(silent) --defconfig=configs/$@ $(Kconfig)
38     $(Q)$< $(silent) --syncconfig $(Kconfig)
39
40 .PHONY: menuconfig savedefconfig defconfig
41
42 # Help text used by make help
43 help:
44     @echo ' menuconfig
45             - Update current config utilising a

```

File Edit View Terminal Tabs Help

Terminal - build.mk (~/ics2021/nemu/scripts) - VIM

```

17 CC ?= gcc
18
19 # Compilation flags
20 CC := $(CC)
21 LD := $(CC)
22 INCLUDES = $(addprefix -I, $(INC_PATH))
23 CFLAGS := -O2 -MMD -Wall -Werror $(INCLUDES) $(CFLAGS)
24 LDFLAGS := -O2 $(LDFLAGS)
25
26 OJJS = $(SRCS:%.c=$(OBJ_DIR)/%.o)
27
28 # Compilation patterns
29 $(OBJ_DIR)/%.o: %.c
30     @echo + CC $<
31     @mkdir -p $(dir $@)
32     @$(CC) $(CFLAGS) -c -o $@ $<
33     $(call call_fixdep, $($@:o=d), $@)
34
35 # Dependencies
36 -include $(OJJS:.o=.d)
37
38 # Some convenient rules
39
40 .PHONY: app clean
41
42 app: $(BINARY)
43

```

File Edit View Terminal Tabs Help

scripts/build.mk

10,1

# NEMU代码构建

- 一个小诀窍
  - 先观察 make 命令实际执行了什么 (trace)
  - RTFM/STFW: make 提供的两个有用的选项
    - -n 只打印命令不运行
    - -B 强制 make 所有目标

```
make -nB \
| grep -ve '^(\#\|echo\|mkdir\)' \
| vim -
```

- 其实没有那么复杂
  - 就是一堆gcc -c (编译) 和一个gcc (链接)
    - 大部分Makefile都是编译选项
- Read the friendly source code!

# Lab代码的构建

- 顶层 (top-level) Makefile:

```
# := -> C #define
NAME := $(shell basename $(PWD))
export MODULE := Lab1
```

```
# 变量 -> 字面替换
all: $(NAME)-64 $(NAME)-32
```

```
# include -> C #include
include ../Makefile
```

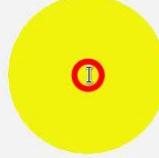
```
1 # **DO NOT MODIFY**
2
3 ifeq ($(NAME),)
4 $(error Should make in each lab's directory)
5 endif
6
7 SRCS := $(shell find . -maxdepth 1 -name "*.c")
8 DEPS := $(shell find . -maxdepth 1 -name "*.h") $(SRCS)
9 CFLAGS += -O1 -std=gnu11 -ggdb -Wall -Werror -Wno-unused-result -Wno-
unused-value -Wno-unused-variable
10
11 .PHONY: all git test clean commit-and-make
12
13 .DEFAULT_GOAL := commit-and-make
14 commit-and-make: git all
15
16 $(NAME)-64: $(DEPS) # 64bit binary
17     gcc -m64 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
18
19 $(NAME)-32: $(DEPS) # 32bit binary
20     gcc -m32 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
21
22 $(NAME)-64.so: $(DEPS) # 64bit shared library
23     gcc -fPIC -shared -m64 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
24
25 $(NAME)-32.so: $(DEPS) # 32bit shared library
26     gcc -fPIC -shared -m32 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
27
28 clean:
29     rm -f $(NAME)-64 $(NAME)-32 $(NAME)-64.so $(NAME)-32.so
30
31 include ../Makefile.lab
```

Applications Lab1: 大整数运算 - Mozilla... Terminal - why@debian... 18:11 拼写助手

File Edit View Terminal Tabs Help

Terminal - why@debian: ~/ICS\_teach/ics-workbench/multimod

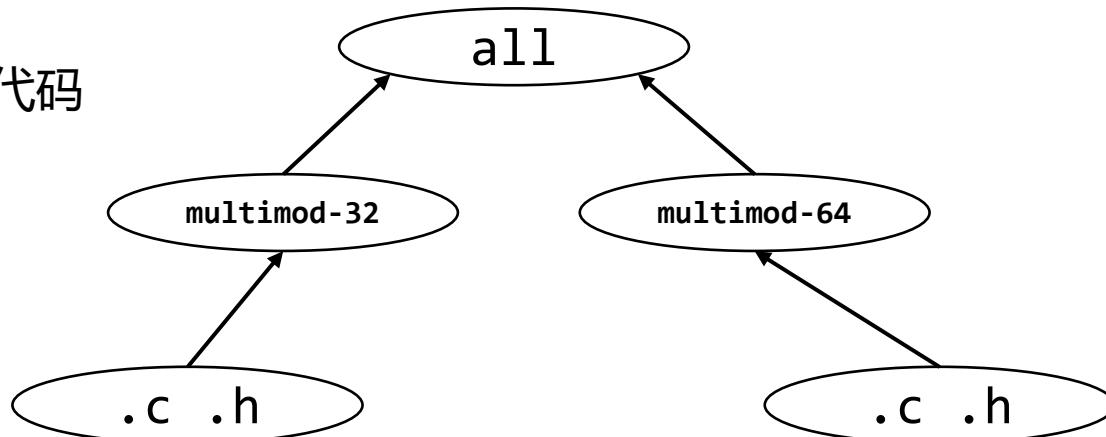
```
$ ls
main.c Makefile multimod-32 multimod-64 multimod.c
$ vim Makefile
$ vim Makefile
```



A large yellow circle is centered on the page. Inside the circle is a smaller red circle containing the white letter 'I'.

# Lab代码的构建 (cont'd)

- 构建目标
  - 总目标
    - .DEFAULT\_GOAL := commit-and-make
    - commit-and-make: git all (all 在顶层 Makefile 中定义)
  - 可执行文件
    - multimod-64: gcc -m64
    - multimod-32: gcc -m32
  - 共享库 (之后的 lab 使用)
    - multimod-64.so: gcc -fPIC -shared -m64
    - multimod-32.so: gcc -fPIC -shared -m32
  - clean
    - 删除构建的代码



# AbstractMachine 代码构建

- 更长，更难读
  - 但我们在给大家留了一个小彩蛋

```
1 Makefile for AbstractMachine Kernels and Libraries
2
3 ### *Get a more readable version of this Makefile* by `make html` (requires python-markdown)
4 html:
5   cat Makefile | sed 's/^\\([^\#]\\)/ \\1/g' | markdown_py > Makefile.html
6 .PHONY: html
7
8 ## 1. Basic Setup and Checks
9
10 ### Default to create a bare-metal kernel image
11 ifeq ($(MAKECMDGOALS),)
12   MAKECMDGOALS = image
13   .DEFAULT_GOAL = image
14 endif
15
16 ### Override checks when `make clean/clean-all/html`
17 ifeq ($(findstring $(MAKECMDGOALS),clean|clean-all|html),)
18
19 ### Print build info message
20 $(info # Building $(NAME)-$(MAKECMDGOALS) [$(ARCH)])
21
```

###\*Get a more readable version of this Makefile\* by  
`make html` (requires python-markdown)

html:

```
cat Makefile | sed 's/^\\([^\#]\\)/ \\1/g' | \
  markdown_py > Makefile.html
```

.PHONY: html

```
45
46 ## 2. General Compilation Targets
47
48 ### Create the destination directory (`build/$ARCH`)
49 WORK_DIR = $(shell pwd)
50 DST_DIR = $(WORK_DIR)/build/$(ARCH)
51 $(shell mkdir -p $(DST_DIR))
52
```

管理 控制 视图 热键 设备 帮助

```
$ ls
am build klib LICENSE Makefile README scripts
$ vim Makefile
$ |
```

# AbstractMachine 代码构建

- “现代”的文档编写方式
  - “docs as code”，例子： LLVM 使用 Doxygen [自动生成文档](#)

LLVM 14.0.0git

## Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

► <a href="#">N adjust</a>	
► <a href="#">N AllocaSlices</a>	
► <a href="#">N false</a>	
► <a href="#">N INITIALIZE_PASS</a>	TargetPassConfig
► <a href="#">N llvm</a>	----- PC ---
► <a href="#">N std</a>	
► <a href="#">C __itt_api_info</a>	
► <a href="#">C __itt_api_info_20101001</a>	
► <a href="#">C __itt_global</a>	
► <a href="#">C __itt_group_list</a>	
► <a href="#">C __itt_thread_info</a>	Detailed Description
► <a href="#">C _JIT_Method_Id</a>	Abstract Attribute helper functions.
► <a href="#">C _JIT_Method_Load</a>	Function Documentation
► <a href="#">C _JIT_Method_NIDS</a>	
► <a href="#">C _LineNumberInfo</a>	
► <a href="#">C AAAAlignArgument</a>	• <a href="#">combineOptionalValuesInAAValueLatice()</a>
► <a href="#">C AAAAlignCallSiteArgument</a>	Optional< Value * > llvm::AA::combineOptionalValuesInAAValueLatice ( const Optional< Value * > & A, const Optional< Value * > & B, Type * Ty )
► <a href="#">C AAAAlignCallSiteReturned</a>	Return the combination of A and B such that the result is a possible value of both. B is potentially casted to match the type Ty or the type of A if Ty is null.

LLVM 14.0.0git

Main Page Related Pages Modules Namespaces Classes Files Examples

Ilvm AA

Namespaces Functions

### Ilvm::AA Namespace Reference

Abstract Attribute helper functions. More...

#### Namespaces

PointerInfo

#### Functions

bool [isDynamicallyUnique](#)(Attributor &A, const AbstractAttribute &QueryingAA, const Value &V)  
Return true if V is dynamically unique, that is, there are no two "instances" of V at runtime with different values. More...

bool [isValidInScope](#)(const Value &V, const Function \*Scope)  
Return true if V is a valid value in Scope, that is a constant or an instruction/argument of Scope. More...

bool [isValidAtPosition](#)(const Value &V, const Instruction &CtxI, InformationCache &InfoCache)  
Return true if V is a valid value at position CtxI, that is a constant, an argument of the same function as CtxI, or an instruction in that function that dominates CtxI. More...

Value \* [getWithType](#)(Value &V, Type &Ty)  
Try to convert V to type Ty without introducing new instructions. More...

Optional< Value \* > [combineOptionalValuesInAAValueLatice](#)(const Optional< Value \* > &A, const Optional< Value \* > &B, Type \*Ty)  
Return the combination of A and B such that the result is a possible value of both. More...

Constant \* [getInitialValueForObj](#)(Value &Obj, Type &Ty)  
Return the initial value of Obj with type Ty if that is a constant. More...

bool [getAssumedUnderlyingObjects](#)(Attributor &A, const Value &Ptr, SmallVectorImpl< Value \* > &Objects, const AbstractAttribute &QueryingAA, const Instruction \*CtxI)  
Collect all potential underlying objects of Ptr at position CtxI in Objects. More...

bool [getPotentialCopiesOfStoredValue](#)(Attributor &A, StoreInst &SI, SmallSetVector< Value \* , 4 > &PotentialCopies, const AbstractAttribute &QueryingAA, bool &UsedAssumedInformation)  
Collect all potential values of the one stored by SI into PotentialCopies. More...

#### Detailed Description

Abstract Attribute helper functions.

#### Function Documentation

• [combineOptionalValuesInAAValueLatice\(\)](#)

Optional< Value \* > llvm::AA::combineOptionalValuesInAAValueLatice ( const Optional< Value \* > & A,  
const Optional< Value \* > & B,  
Type \* Ty  
)

Return the combination of A and B such that the result is a possible value of both.

B is potentially casted to match the type Ty or the type of A if Ty is null.

# pydoc

- pydoc3 list

```
Help on class list in module builtins:

class list(object)
| list(iterable=(), /)

    Built-in mutable sequence.

    If no argument is given, the constructor creates a new empty list.
    The argument must be an iterable if specified.

Methods defined here:

__add__(self, value, /)
    Return self+value.

__contains__(self, key, /)
    Return key in self.

__delitem__(self, key, /)
    Delete self[key].

__eq__(self, value, /)
    Return self==value.

__ge__(self, value, /)
    Return self>=value.

__getattribute__(self, name, /)
    Return getattr(self, name).

__getitem__(...)
    x.__getitem__(y) <=> x[y]

__gt__(self, value, /)
    Return self>value.

__iadd__(self, value, /)
    Implement self+=value.

__imul__(self, value, /)
    Implement self*=value.

__init__(self, /, *args, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.

__iter__(self, /)
    Implement iter(self).

__le__(self, value, /)
    Return self<=value.

:
```

就是理解了构建的过程，NEMU代码依然很难读？

RTFSC

Read the friendly source code!

# 拿到源代码，先做什么？

NEMU对大部分同学来说是一个“前所未有的大”的项目。

- 先大致了解一下

- 项目总体组织

- tree 要翻好几个屏幕

- find . -name "\*.c" -o -name "\*.h" (100+ 个文件)

- 项目规模

- find ... | xargs cat | wc -l

- 5000 行左右 (其实很小了)

# tree

```
why@why-VirtualBox:~/Documents/ICS20:
```

```
.
```

```
  configs
    riscv32-am_defconfig
    riscv64-am_defconfig
  include
    common.h
  config
    auto.conf
    auto.conf.cmd
    cc
      gcc.h
      o2.h
      opt.h
    cc.h
    difftest
      ref
        name.h
        path.h
    engine
      interpreter.h
      engine.h
    isa
      riscv32.h
    isa.h
    itrace
      cond.h
    itrace.h
    mbase.h
    mem
      random.h
    mode
      system.h
    msiz.e.h
    pc
      reset
        offset.h
    pmem
      garray.h
    rt
      check.h
    target
      native
        elf.h
    timer
      gettimeofday.h
    trace
      end.h
      start.h
    trace.h
```

```
.
```

```
  cpu
    cpu-exec.c
    difftest
      dut.c
      ref.c
  device
    alarm.c
    audio.c
    device.c
    disk.c
    filelist.mk
    intr.c
    io
      map.c
      mmio.c
      port-io.c
    Kconfig
    keyboard.c
    mmc.h
    sdcard.c
    serial.c
    timer.c
    vga.c
  engine
    filelist.mk
    interpreter
      hostcall.c
      init.c
    filelist.mk
    riscv32
      difftest
        dut.c
      include
        isa-def.h
      init.c
      inst.c
      local-include
        reg.h
      logo.c
      reg.c
      system
        intr.c
        mmu.c
  riscv64
    difftest
      dut.c
    include
      isa-def.h
    init.c
    inst.c
    local-include
      reg.h
    logo.c
    reg.c
    system
```

```
.
```

```
  memory
    Kconfig
    paddr.c
    vaddr.c
  monitor
    monitor.c
  sdb
    expr.c
    sdb.c
    sdb.h
    watchpoint.c
  nemu-main.c
  utils
    disasm.cc
    filelist.mk
    log.c
    rand.c
    state.c
    timer.c
  tags
  tools
    difftest.mk
    fixdep
      build
        fixdep
          fixdep.d
          fixdep.o
        fixdep.c
        Makefile
      gen-expr
        gen-expr.c
        Makefile
      kconfig
        build
          conf
            lexer.lex.c
          mconf
            obj-conf
              build
                conf.d
                confdata.d
                confdata.o
                conf.o
                expr.d
                expr.o
                preprocess.d
                preprocess.o
  yesno.o
  mconf.d
  mconf.o
  preprocess.d
  preprocess.o
  symbol.d
  symbol.o
  util.d
  util.o
  parser.output
  parser.tab.c
  parser.tab.h
  conf.c
  confdata.c
  expr.c
  expr.h
  lexer.l
  list.h
  lkc.h
  lkc_proto.h
  lxdialog
    checklist.c
    dialog.h
    inputbox.c
    menubox.c
    textbox.c
    util.c
    yesno.c
    Makefile
    mconf.c
    menu.c
    parser.y
    preprocess.c
    symbol.c
    util.c
  kvm-diff
    include
      paddr.h
    Makefile
    src
      kvm.c
  qemu-diff
    include
      common.h
      isa.h
      protocol.h
    Makefile
    src
      diff-test.c
      gdb-host.c
      isa.c
      protocol.c
  spike-diff
    difftest.cc
    Makefile
```

70 directories, 208 files

# 拿到源代码，先做什么？

NEMU对大部分同学来说是一个“前所未有的大”的项目。

- 先大致了解一下
  - 项目总体组织
    - tree要翻好几个屏幕
    - find . -name "\*.c" -o -name "\*.h" (100+ 个文件)

```
find . -name "*.c" -o -name "*.h"
```

```
$ find . -name "*.c" -o -name "*.h"
./tools/gen-expr/gen-expr.c
./tools/fixdep/fixdep.c
./tools/kvm-diff/src/kvm.c
./tools/kvm-diff/include/paddr.h
./tools/kconfig/confdata.c
./tools/kconfig/mconf.c
./tools/kconfig/lkc.h
./tools/kconfig/symbol.c
./tools/kconfig/menu.c
./tools/kconfig/lxdialog/menubox.c
./tools/kconfig/lxdialog/dialog.h
./tools/kconfig/lxdialog/checklist.c
./tools/kconfig/lxdialog/textbox.c
./tools/kconfig/lxdialog/inputbox.c
./tools/kconfig/lxdialog/util.c
./tools/kconfig/lxdialog/yesno.c
./tools/kconfig/lkc_proto.h
./tools/kconfig/list.h
./tools/kconfig/build/lexer.lex.c
./tools/kconfig/build/parser.tab.h
./tools/kconfig/build/parser.tab.c
```

# 拿到源代码，先做什么？

NEMU对大部分同学来说是一个“前所未有的大”的项目。

- 先大致了解一下

- 项目总体组织

- tree要翻好几个屏幕

- find . -name "\*.c" -o -name "\*.h" (100+个文件)

- 项目规模

- find ... | xargs cat | wc -l

```
$ find . -name "*.c" -o -name "*.h" | xargs cat | wc -l  
23069
```

```
$ find tools -name "*.c" -o -name ".h" | xargs cat | wc -l  
17728
```

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用**IDE (vscode: Edit→Find in files)**

```
$ ls  
abstract-machine  am-kernels  fceux-am  init.sh  Makefile  nemu  README.md  tags  
$
```

I

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用IDE (**vscode**: **Edit**→**Find in files**)
- The UNIX Way (无需启动任何程序，即可直接查看)

```
grep -n main $(find . -name "*.c") # RTFM: -n
```

# grep -n main \$(find . -name "\*.c")

```
$ grep -n main $(find . -name "*.c")
./tools/gen-expr/gen-expr.c:13:int main() { "
./tools/gen-expr/gen-expr.c:23:int main(int argc, char *argv[]) {
./tools/fixdep/fixdep.c:385:int main(int argc, char *argv[])
./tools/kconfig/mconf.c:1004:int main(int ac, char **av)
./tools/kconfig/symbol.c:1265:           /* for choice groups start the check with main
choice symbol */
./tools/kconfig/menu.c:331:                      /* set the type of the remaining choic
e values */
./tools/kconfig/build/lexer.lex.c:144:/* The state buf must be large enough to hold on
e state per character in the main buffer.
./tools/kconfig/build/lexer.lex.c:2684:/** The main scanner function which does all th
e work.
./tools/kconfig/build/lexer.lex.c:4119:/* Defined in main.c */
./tools/kconfig/build/parser.tab.c:541: "T_NOT", "$accept", "input", "mainmenu_stmt",
"stmt_list",
./tools/kconfig/conf.c:500:int main(int ac, char **av)
./resource/sdcard/nemu.c:91:     unsigned int          blocks;          /* remaining P
IO blocks */
./src/engine/interpreter/init.c:3:void sdb_mainloop();
./src/engine/interpreter/init.c:10:    sdb_mainloop();
./src/isa/riscv64/instr/decode.c:55:def_THelper(main) {
./src/isa/riscv64/instr/decode.c:65:    int idx = table_main(s);
./src/isa/riscv32/instr/decode.c:55:def_THelper(main) {
./src/isa/riscv32/instr/decode.c:65:    int idx = table_main(s);
./src/nemu-main.c:8:int main(int argc, char *argv[]) {
./src/monitor/sdb/sdb.c:84:void sdb_mainloop() {
./src/monitor/sdb/sdb.c:97:    /* treat the remaining string as the arguments,
```

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用IDE (**vscode**: **Edit**→**Find in files**)
- The UNIX Way (无需启动任何程序，即可直接查看)

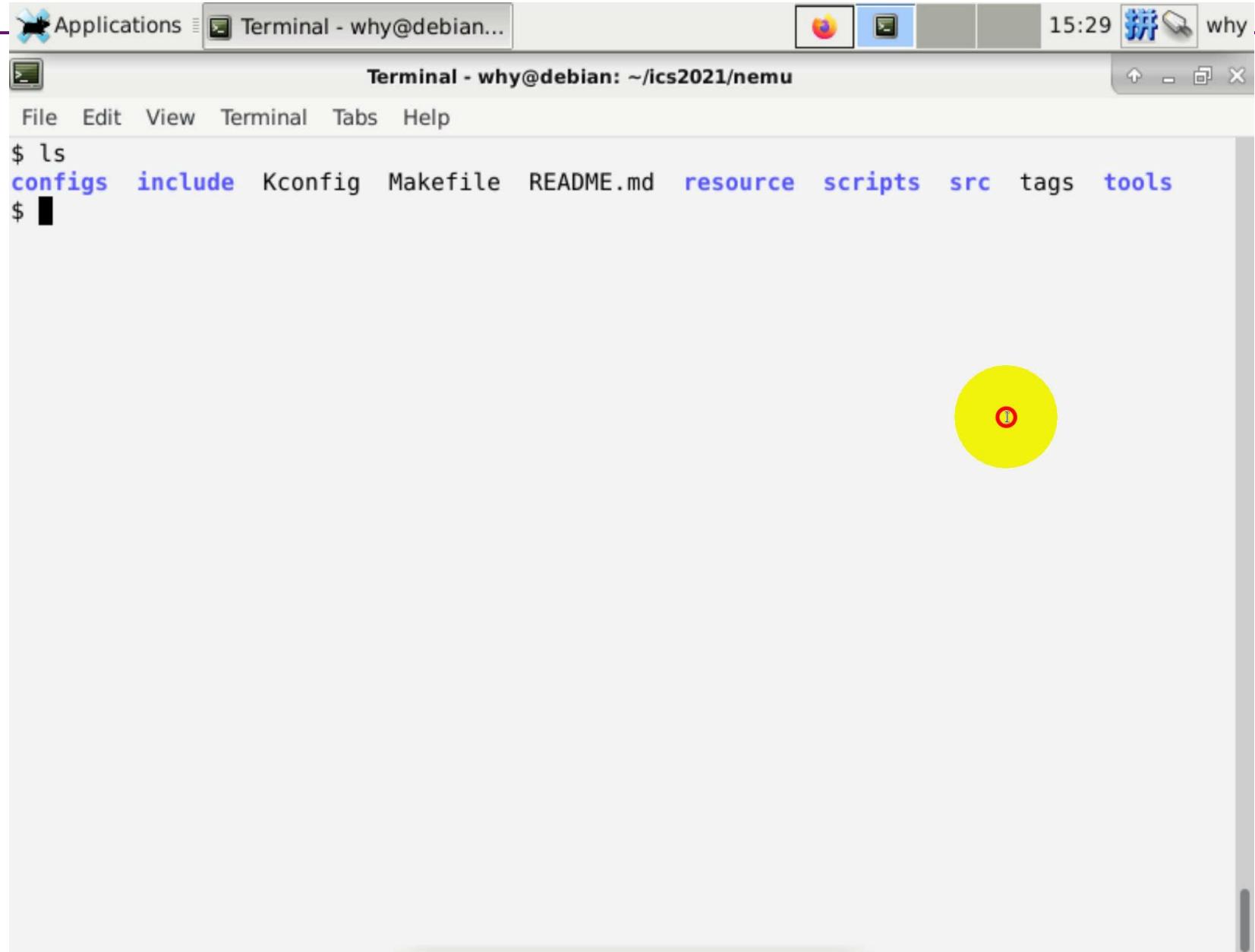
```
grep -n main $(find . -name "*.c") # RTFM: -n
```

```
find . | xargs grep --color -nse '\<main\>'
```

```
find . | xargs grep --color -nse '\<main\>'
```

```
$ find . | xargs grep --color -nse '\<main\>'  
./README.md:7:The main features of NEMU include  
.tools/gen-expr/gen-expr.c:13:int main() {  
.tools/gen-expr/gen-expr.c:23:int main(int argc, char *argv[]) {  
.tools/fixdep/fixdep.c:385:int main(int argc, char *argv[])  
Binary file ./tools/fixdep/build/fixdep matches  
Binary file ./tools/fixdep/build/obj-fixdep/fixdep.o matches  
.tools/kconfig/mconf.c:1004:int main(int ac, char **av)  
.tools/kconfig/symbol.c:1265:           /* for choice groups start the check with main  
choice symbol */  
Binary file ./tools/kconfig/build/conf matches  
Binary file ./tools/kconfig/build/obj-conf/conf.o matches  
.tools/kconfig/build/lexer.lex.c:144:/* The state buf must be large enough to hold on  
e state per character in the main buffer.  
.tools/kconfig/build/lexer.lex.c:2684:/* The main scanner function which does all th  
e work.  
.tools/kconfig/build/lexer.lex.c:4119:/* Defined in main.c */  
Binary file ./tools/kconfig/build/obj-mconf/mconf.o matches  
Binary file ./tools/kconfig/build/mconf matches  
.tools/kconfig/conf.c:500:int main(int ac, char **av)  
.tools/kconfig/expr.h:70:           S_DEF_USER,           /* main user value */  
.src/isa/riscv64/instr/decode.c:55:def_THelper(main) {  
.src/isa/riscv32/instr/decode.c:55:def_THelper(main) {  
.src/filelist.mk:1:SRCS-y += src/nemu-main.c  
.src/nemu-main.c:8:int main(int argc, char *argv[]) {  
Binary file ./build/riscv32-nemu-interpreter matches  
.build/obj-riscv32-nemu-interpreter/src/nemu-main.d:1:cmd_/home/why/ics2021/nemu/buil  
d/obj-riscv32-nemu-interpreter/src/nemu-main.o := unused
```

# fzf



A screenshot of a terminal window titled "Terminal - why@debian: ~/ics2021/nemu". The window shows the output of the command \$ ls, which lists several files: configs, include, Kconfig, Makefile, README.md, resource, scripts, src, and tags. The word "resource" is highlighted in blue, indicating it was selected by fzf. The terminal has a standard window title bar with icons for Applications, Terminal, and a user icon. The status bar at the top right shows the time as 15:29 and the user as why. A yellow circle with a red "I" is overlaid on the right side of the terminal window.

```
$ ls
configs  include  Kconfig  Makefile  README.md  resource  scripts  src  tags  tools
$ █
```

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用IDE (vscode: Edit→Find in files)
- The UNIX Way (无需启动任何程序，即可直接查看)

```
grep -n main $(find . -name "*.c") # RTFM: -n  
find . | xargs grep --color -nse '\<main\>'
```

- Fuzzy Finder
- Vim当然也支持

```
:vimgrep /\<main\>/ **/*.c
```

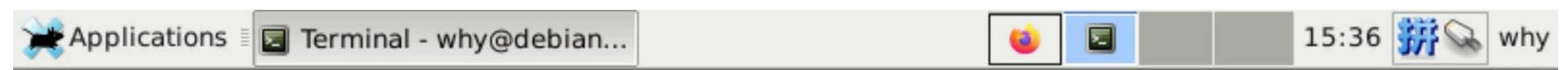
- 浏览：:cn, :cp, .....

# main()

- 比想象中短很多.....

```
int main(int argc, char *argv[]) {  
    init_monitor(argc, argv);  
    engine_start();  
    return is_exit_status_bad();  
}
```

```
1 #include <common.h>  
2  
3 void init_monitor(int, char *[]);  
4 void am_init_monitor();  
5 void engine_start();  
6  
7  
8 int is_exit_status_bad();  
9  
10 int main(int argc, char *argv[]) {  
11     /* Initialize the monitor. */  
12 #ifdef CONFIG_TARGET_AM  
13     am_init_monitor();  
14 #else  
15     init_monitor(argc, argv);  
16 #endif  
17  
18     /* Start engine. */  
19     engine_start();  
20  
21     return is_exit_status_bad();  
22 }
```



# main()

- 比想象中短很多.....

```
int main(int argc, char *argv[]) {
    init_monitor(argc, argv);
    engine_start();
    return is_exit_status_bad();
}
```

```
1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6
7
8 int is_exit_status_bad();
9
10 int main(int argc, char *argv[]) {
11     /* Initialize the monitor. */
12 #ifdef CONFIG_TARGET_AM
13     am_init_monitor();
14 #else
15     init_monitor(argc, argv);
16 #endif
17
18     /* Start engine. */
19     engine_start();
20
21     return is_exit_status_bad();
22 }
```

## • Comments

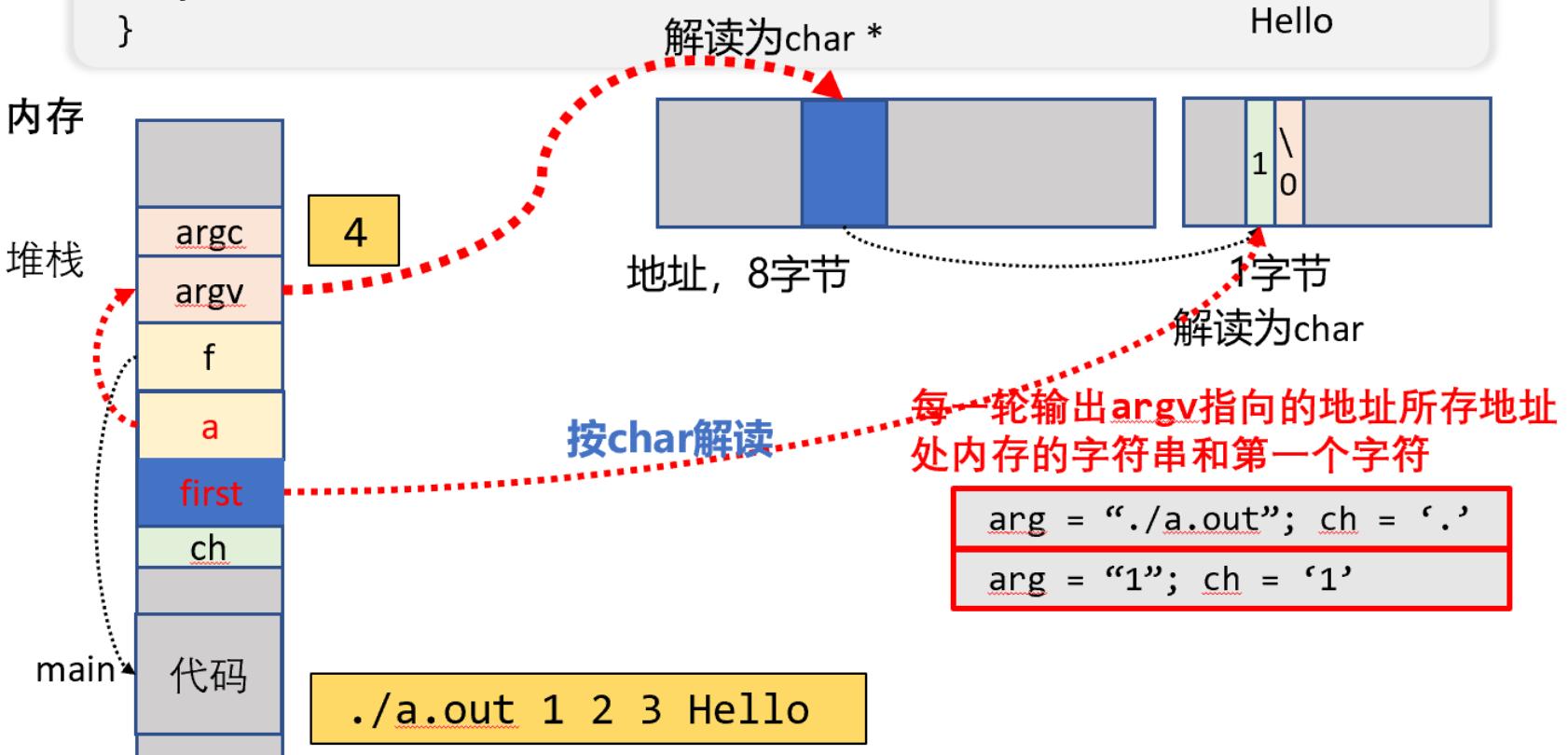
- 把 argc, argv 传递给另一个函数是 C 的 idiomatic use
- init\_monitor 代码在哪里?
  - 每次都 grep 效率太低
  - 需要更先进的工具 (稍候介绍)

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0];
        printf("arg = \"%s\"; ch = '%c'\n", **a, *first);
        assert(**a == first);
        f(argc - 1, argv + 1);
    }
}

```

\$ ./a.out 1 2 3 hello  
 arg = "./a.out"; ch = '.'  
 arg = "1"; ch = '1'  
 arg = "2"; ch = '2'  
 arg = "3"; ch = '3'  
 arg = "hello"; ch = 'h'



Applications Terminal - nemu-main.c...

15:36 拼 why

Terminal - nemu-main.c (~/ics2021/nemu/src) - VIM

File Edit View Terminal Tabs Help

```
1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6
7
8 int is_exit_status_bad();
9
10 int main(int argc, char *argv[]) {
11     /* Initialize the monitor. */
12 #ifdef CONFIG_TARGET_AM
13     am_init_monitor();
14 #else
15     init_monitor(argc, argv);
16 #endif
17
18     /* Start engine. */
19     engine_start();
20
21     return is_exit_status_bad();
22 }
```

src/nemu-main.c 14,1 All  
"src/nemu-main.c" 22L, 357C

Applications Terminal - why@debian...

15:52 why

File Edit View Terminal Tabs Help

```
$ ls
configs include Kconfig Makefile README.md resource scripts src tags tools
$ vim $(fzf)
$ vim $(fzf)
$
```



41

```
56 static int parse_args(int argc, char *argv[]) {
57     const struct option table[] = {
58         {"batch" , no_argument , NULL, 'b'},
59         {"log"   , required_argument, NULL, 'l'},
60         {"diff"  , required_argument, NULL, 'd'},
61         {"port"  , required_argument, NULL, 'p'},
62         {"help"  , no_argument    , NULL, 'h'},
63         {0       , 0             , NULL, 0 },
64     };
65     int o;
66     while ((o = getopt_long(argc, argv, "-bhl:d:p:", table, NULL)) != -1) {
67         switch (o) {
68             case 'b': sdb_set_batch_mode(); break;
69             case 'p': sscanf(optarg, "%d", &difftest_port); break;
70             case 'l': log_file = optarg; break;
71             case 'd': diff_so_file = optarg; break;
72             case 1: img_file = optarg; return optind - 1;
73             default:
74                 printf("Usage: %s [OPTION...] IMAGE [args]\n\n", argv[0]);
75                 printf("\t-b,--batch                  run with batch mode\n");
76                 printf("\t-l,--log=FILE                output log to FILE\n");
77                 printf("\t-d,--diff=REF_SO             run DiffTest with reference REF_SO\n");
78                 printf("\t-p,--port=PORT               run DiffTest with port PORT\n");
79                 printf("\n");
80         exit(0);
81     }
82 }
```

src/monitor/monitor.c

80,1

49%

21

42

- make run

```
$ make run
+ LD /home/why/ics2021/nemu/build/riscv32-nemu-interpreter
/home/why/ics2021/nemu/build/riscv32-nemu-interpreter --log=/home/why/ics2021/nemu/build/nemu-log.txt
[src/memory/paddr.c:35 init_mem] physical memory area [0x80000000, 0x88000000]
[src/monitor/monitor.c:36 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:12 welcome] Debug: ON
[src/monitor/monitor.c:16 welcome] If debug mode is on, a log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:17 welcome] Build time: 15:57:12, Aug 30 2021
Welcome to riscv32-NEMU!
For help, type "help"
[src/monitor/monitor.c:20 welcome] Exercise: Please remove me in the source code and compile NEMU again.
(nemu) help
help - Display informations about all supported commands
c - Continue the execution of the program
q - Exit NEMU
(nemu) █
```

- ./build/riscv32-nemu-interpreter

```
$ ./build/riscv32-nemu-interpreter
[src/memory/paddr.c:35 init_mem] physical memory area [0x80000000, 0x88000000]
[src/monitor/monitor.c:36 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:12 welcome] Debug: ON
[src/monitor/monitor.c:16 welcome] If debug mode is on, a log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:17 welcome] Build time: 15:57:12, Aug 30 2021
Welcome to riscv32-NEMU!
For help, type "help"
[src/monitor/monitor.c:20 welcome] Exercise: Please remove me in the source code and compile NEMU again.
(nemu) █
```

# parse\_args()

---

- 这个函数的名字起的很好，看了就知道要做什么
  - 满足好代码不言自明的特性
  - 的确是用来解析命令行参数的， -b, -l, ...
  - 使用了 getopt → RTFM!
- 失败的尝试： man getopt → getopt (1)
- 成功的尝试
  - 捷径版： STFW “C getopt” → 网页/博客/...
  - 专业版： man -k getopt → man 3 getopt
- 意外之喜： man 还送了个例子！跟 parse\_args 的用法一样耶

```
$ ./build/riscv32-nemu-interpreter --help
Usage: ./build/riscv32-nemu-interpreter [OPTION...] IMAGE [args]

-b,--batch           run with batch mode
-l,--log=FILE        output log to FILE
-d,--diff=REF_S0     run DiffTest with reference REF_S0
-p,--port=PORT       run DiffTest with port PORT

$ ./build/riscv32-nemu-interpreter -b
[src/memory/paddr.c:34 init_mem] physical memory area [0x80000000, 0x88000000]
[src/monitor/monitor.c:36 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:12 welcome] Debug: ON
[src/monitor/monitor.c:13 welcome] If debug mode is on, a log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:17 welcome] Build time: 10:17:25, Sep 23 2021
Welcome to riscv32-NEMU!
For help, type "help"
[src/cpu/cpu-exec.c:104 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x8000000c
[src/cpu/cpu-exec.c:69 monitor_statistic] host time spent = 6 us
[src/cpu/cpu-exec.c:70 monitor_statistic] total guest instructions = 4
[src/cpu/cpu-exec.c:71 monitor_statistic] simulation frequency = 666,666 instr/s
c
```

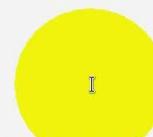
# 怎么给nemu传指令？

---



File Edit View Terminal Tabs Help

```
$ ls
build configs include Kconfig Makefile README.md resource scripts src tags tools
$ make run
+ LD /home/why/ics2021/nemu/build/riscv32-nemu-interpreter
/home/why/ics2021/nemu/build/riscv32-nemu-interpreter --log=/home/why/ics2021/nemu/build/nemu-log.
txt
[src/memory/paddr.c:35 init_mem] physical memory area [0x80000000, 0x88000000]
[src/monitor/monitor.c:36 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:12 welcome] Debug: ON
[src/monitor/monitor.c:16 welcome] If debug mode is on, a log file will be generated to record eve
ry instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can t
urn it off in include/common.h.
[src/monitor/monitor.c:17 welcome] Build time: 15:57:12, Aug 30 2021
Welcome to riscv32-NEMU!
For help, type "help"
[src/monitor/monitor.c:20 welcome] Exercise: Please remove me in the source code and compile NEMU
again.
(nemu) q
make: *** [/home/why/ics2021/nemu/scripts/native.mk:23: run] Error 1
$
```



## Terminal - native.mk (~/ics2021/nemu/scripts) - VIM



File Edit View Terminal Tabs Help

```
2 include $(NEMU_HOME)/scripts/build.mk
3
4 include $(NEMU_HOME)/tools/difftest.mk
5
6 compile_git:
7     $(call git_commit, "compile")
8 $(BINARY): compile_git
9
10 # Some convenient rules
11
12 override ARGS ?= --log=$(BUILD_DIR)/nemu-log.txt
13 override ARGS += $(ARGS_DIFF)
14
15 # Command to execute NEMU
16 IMG ?=
17 NEMU_EXEC := $(BINARY) $(ARGS) $(IMG)
18
19 run-env: $(BINARY) $(DIFF_REF_S0)
20
21 run: run-env
22     $(call git_commit, "run")
23     $(NEMU_EXEC)
24
25 gdb: run-env
26     $(call git_commit, "gdb")
27     |gdb -s $(BINARY) --args $(NEMU_EXEC)
28
```

# NEMU：一个命令行工具

- The friendly source code
  - 命令行可以控制 NEMU 的行为
  - 我们甚至看到了 --help 帮助信息
- 如何让我们的 NEMU 打印它?
  - 问题等同于: make run 到底做了什么
  - 方法 1: 阅读 Makefile
  - 方法 2: 借助 GNU Make 的 -n 选项

开始痛苦的代码阅读之旅：坚持！

# 代码选讲

```

56 static int parse_args(int argc, char *argv[]) {
57     const struct option table[] = {
58         {"batch"      , no_argument      , NULL, 'b'},
59         {"log"        , required_argument, NULL, 'l'},
60         {"diff"       , required_argument, NULL, 'd'},
61         {"port"       , required_argument, NULL, 'p'},
62         {"help"       , no_argument      , NULL, 'h'},
63         {0            , 0              , NULL, 0 },
64     };
65     int o;
66     while ((o = getopt_long(argc, argv, "-bhl:d:p:", table, NULL)) != -1) {
67         switch (o) {
68             case 'b': sdb_set_batch_mode(); break;
69             case 'p': sscanf(optarg, "%d", &difftest_port); break;
70             case 'l': log_file = optarg; break;
71             case 'd': diff_so_file = optarg; break;
72             case 'l': img_file = optarg; return optind - 1;
73             default:
74                 printf("Usage: %s [OPTION...] IMAGE [args]\n\n", argv[0]);
75                 printf("\t-b,--batch           run with batch mode\n");
76                 printf("\t-l,--log=FILE         output log to FILE\n");
77                 printf("\t-d,--diff=REF_SO      run DiffTest with reference REF_SO\n");
78                 printf("\t-p,--port=PORT        run DiffTest with port PORT\n");
79                 printf("\n");
80                 exit(0);
81         }
82     }
83     return 0;
84 }
85 
```

src/monitor/monitor.c

85,0-1

# static

```
static int parse_args(int argc, char *argv[]) { ... }
```

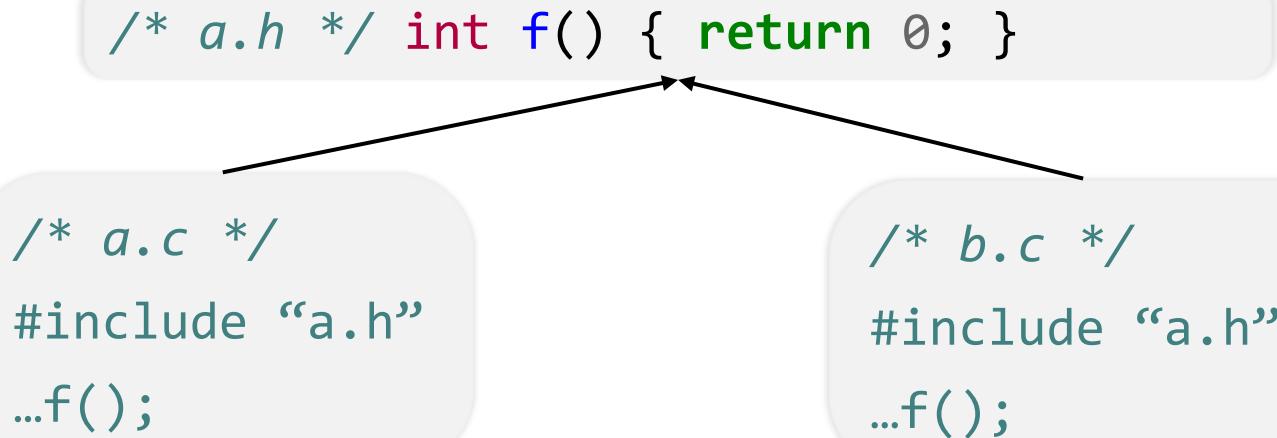
- `parse_args` 函数是 `static` 的，这是什么意思?
  - `static` (C99 6.2.2 #3): If the declaration of a file scope identifier for an object or a function contains the storage-class specifier `static`, the identifier has internal linkage.
- 可以是 `static`，建议是 `static`
- 告诉编译器符号不要泄露到文件 (*translation unit*) 之外

# static

- 我们都知道，如果在两个文件里定义了重名的函数，能够分别编译，但链接会出错：

```
/* a.c */ int f() { return 0; }  
/* b.c */ int f() { return 1; }
```

- b.c:(.text+0x0): multiple definition of f; a.c:(.text+0xb): first defined here



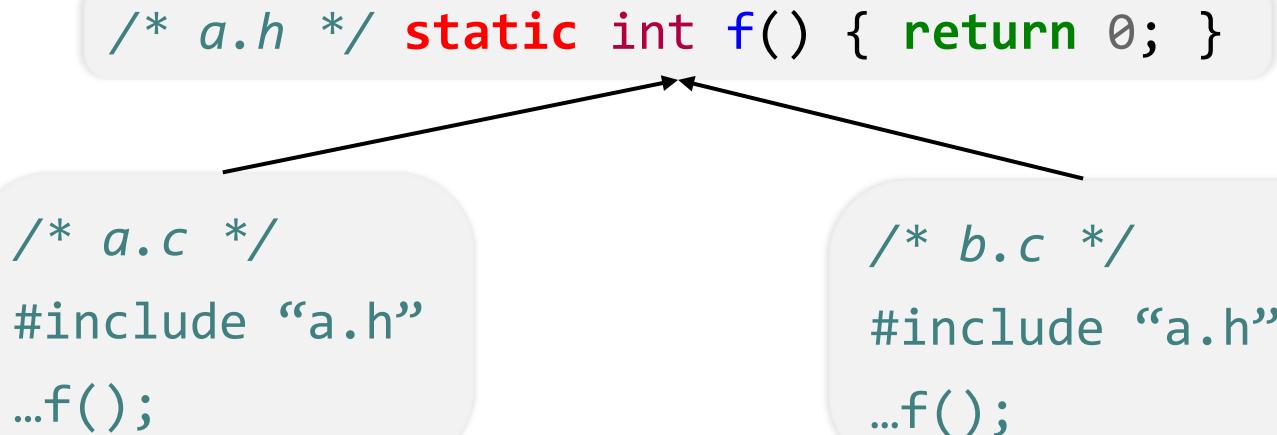
ERROR

# static

- 我们都知道，如果在两个文件里定义了重名的函数，能够分别编译，但链接会出错：

```
/* a.c */ int f() { return 0; }  
/* b.c */ int f() { return 1; }
```

- b.c:(.text+0x0): multiple definition of f; a.c:(.text+0xb): first defined here



Pass

# static

- 我们都知道，如果在两个文件里定义了重名的函数，能够分别编译，但链接会出错：

```
/* a.c */ int f() { return 0; }  
/* b.c */ int f() { return 1; }
```

- b.c:(.text+0x0): multiple definition of f; a.c:(.text+0xb): first defined here

```
/* a.h */ static int f() { return 0; }
```

```
// static int a;
```

```
/* a.c */  
#include "a.h"  
...f();
```

```
/* b.c */  
#include "a.h"  
...f();
```

```
/* x.c */  
#include "a.h"
```



# static

- 我们都知道，如果在两个文件里定义了重名的函数，能够分别编译，但链接会出错：

```
/* a.c */ int f() { return 0; }  
/* b.c */ int f() { return 1; }
```

- b.c:(.text+0x0): multiple definition of f; a.c:(.text+0xb): first defined here

```
/* a.h */ static int f() { return 0; }
```

```
// static int a;
```

```
/* a.c */  
#include "a.h"  
...f();
```

```
/* b.c */  
#include "a.h"  
...f();
```

```
/* x.c */  
#include "a.h"
```

**WARNING**  
**ERROR (-Wall -Werror)**

## \*\*/riscv32/\*\*/reg.h

```
1 #ifndef __RISCV32_REG_H__
2 #define __RISCV32_REG_H__
3
4 #include <common.h>
5
6 static inline int check_reg_idx(int idx) {
7     IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && idx < 32));
8     return idx;
9 }
10
11 #define gpr(idx) (cpu.gpr[check_reg_idx(idx)]._32)
12
13 static inline const char* reg_name(int idx, int width) {
14     extern const char* regs[];
15     return regs[check_reg_idx(idx)];
16 }
17
18 #endif
```

# static inline

- 如果你的程序较短且性能攸关，则可以使用 static inline 函数定义在头文件中。例子 (\*\*/riscv32/\*\*/reg.h):

```
static inline int check_reg_index(int idx) {  
    IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && index < 32));  
    return idx;  
}
```

```
/* a.h */ static inline int f() { return 0; }  
// static int a;
```

```
/* a.c */  
#include "a.h"  
...f();
```

```
/* b.c */  
#include "a.h"  
...f();
```

```
/* x.c */  
#include "a.h"
```



# static inline

- 如果你的程序较短且性能攸关，则可以使用 static inline 函数定义在头文件中。例子 (\*\*/riscv32/\*\*/reg.h):

```
static inline int check_reg_index(int idx) {  
    IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && index < 32));  
    return idx;  
}
```

- check\_reg\_index完全可以单独放在一个 C 文件里，头文件中只保留声明：

```
int check_reg_index(int index);
```

- 但这样会导致在编译时，编译出一条额外的 call 指令 (假设没有 LTO)
- 使用 inline 可以在调用 check\_reg\_index(0) 编译优化成零开销

# assert

```
#define assert(cond) if (!(cond)) panic(...);
```

- 注意特殊情况

```
if (...) assert(0); // 上面的assert对么?  
else ...
```

```
#define assert(cond) \ // nemu/**/debug.h  
do { \  
    if (!(cond)) { \  
        fprintf(stderr, "Fail @ %s:%d", __FILE__, __LINE__);  
    \  
        exit(1); \  
    } \  
} while (0)
```

```
#define assert(cond) ({ ... }) // GCC
```

# \*\*/debug.h

```
Terminal - debug.h (~/ics2021/nemu/include) - VIM
File Edit View Terminal Tabs Help
1 #ifndef __DEBUG_H__
2 #define __DEBUG_H__
3
4 #include <common.h>
5 #include <stdio.h>
6 #include <utils.h>
7
8 #define Log(format, ...) \
9     _Log(ASNI_FMT("[%-s:%d %s] " format, ASNI_FG_BLUE) "\n", \
10          __FILE__, __LINE__, __func__, ##__VA_ARGS__)
11
12 #define Assert(cond, format, ...) \
13     do { \
14         if (!(cond)) { \
15             MUXDEF(CONFIG_TARGET_AM, printf(ASNI_FMT(format, ASNI_FG_RED) "\n", ##__VA_ARGS__), \
16                 (fflush(stdout), fprintf(stderr, ASNI_FMT(format, ASNI_FG_RED) "\n", ##__VA_ARGS__))) \
17             extern void isa_reg_display(); \
18             extern void monitor_statistic(); \
19             isa_reg_display(); \
20             monitor_statistic(); \
21             assert(cond); \
22         } \
23     } while (0)
24
25 #define panic(format, ...) Assert(0, format, ##__VA_ARGS__)
26
27 #define TODO() panic("please implement me")
28
29 #endif
-
include/debug.h
"include/debug.h" 291 797C
27,6 All 61
```

Applications Terminal - why@debian... Terminal - nemu-main.c...

Terminal - nemu-main.c (~/ics2021/nemu/src) - VIM

File Edit View Terminal Tabs Help

```
1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6
7
8 int is_exit_status_bad();
9
10 int main(int argc, char *argv[]) {
11     /* Initialize the monitor. */
12 #ifdef CONFIG_TARGET_AM
13     am_init_monitor();
14 #else
15     init_monitor(argc, argv);
16 #endif
17
18     /* Start engine. */
19     engine_start();
20
21     return is_exit_status_bad();
22 }
```

src/nemu-main.c 15,3 All  
"src/nemu-main.c" 22L, 357C

# 千辛万苦.....

- 之后的历程似乎就比较轻松了。有些东西不太明白(比如 `init_device()`), 但好像也不是很要紧, 到了 `welcome()`

```
$ make run
+ LD /home/why/ics2021/nemu/build/riscv32-nemu-interpreter
/home/why/ics2021/nemu/build/riscv32-nemu-interpreter --log=/home/why/ics2021/nemu/build/nemu-log.txt
[src/memory/paddr.c:35 init_mem] physical memory area [0x80000000, 0x88000000]
[src/monitor/monitor.c:36 load_img] No image is given. Use the default build-in image.
[src/monitor/monitor.c:12 welcome] Debug: ON
[src/monitor/monitor.c:16 welcome] If debug mode is on, a log file will be generated to record every instruction NEMU executes. This may lead to a large log file. If it is not necessary, you can turn it off in include/common.h.
[src/monitor/monitor.c:17 welcome] Build time: 17:23:31, Aug 30 2021
Welcome to riscv32-NEMU!
For help, type "help"
[src/monitor/monitor.c:20 welcome] Exercise: Please remove me in the source code and compile NEMU again.
```

```
$ echo -e "\033[34m HAHA \033[0m"
    HAHA
$ echo -e "\033[43;34m HAHA \033[0m"
    HAHA
$ echo -e "\033[44;37m HAHA \033[0m"
    HAHA
```

Terminal - why@debian: ~

File Edit View Terminal Tabs Help

\$ ls

**build configs include in.txt Kconfig Makefile README.md resource scripts src tags tools**

\$

# 千辛万苦.....

---

- 初始化终于完成
- 根本没碰到核心代码

# 理解代码：更进一步

# 来自以往同学的反馈

“我决定挑战自己，坚持在命令行中工作、使用 Vim 编辑代码。但在巨多的文件之间切换真是一言难尽。”

- 上手以后还在用 grep 找代码?
  - 你刚拿到项目的时候, grep 的确不错
  - 但如果要在一学期都在这个项目上, 效率就太低了
    - 曾经有无数的同学选择容忍这种低效率

# Vim：这都搞不定还引发什么编辑器圣战

- Marks (文件内标记)
  - ma, 'a, mA, 'A, ...
- Tags (在位置之间跳转)
  - :jumps, C-], C-i, C-o, :tjump, ...
- Tabs/Windows (管理多文件)
  - :tabnew, gt, gT, ...
- Folding (浏览大代码)
  - zc, zo, zR, ...
- 更多的功能/插件: (RTFM, STFW)
  - vimtutor

# vi / vim graphical cheat sheet

**ESC**

normal mode

~ toggle case	! external filter	@ play macro	# prev ident	\$ eol	% goto match	^ "soft" bol	& repeat :s	* next ident	( begin sentence	) end sentence	"soft" bol down	+ next line
\` goto mark	1	2	3	4	5	6	7	8	9	0 "hard" bol	- prev line	= auto <sup>3</sup> format
Q ex mode	W next WORD	E end WORD	R replace mode	T back 'till	Y yank line	U undo line	I insert at bol	O open above	P paste before	{ begin parag.	}	end parag.
q record macro	W next word	e end word	r replace char	t 'till	y yank	u undo	i insert mode	o open below	p paste <sup>1</sup> after	[ misc	]	• misc
A append at eol	S subst line	D delete to eol	F "back" find ch	G eof/ goto ln	H screen top	J join lines	K help	L screen bottom	• ex cmd line	". reg. spec	bol/ goto col	
a append	S subst char	d delete	f find char	g extra <sup>6</sup> cmd	h ←	j ↓	k ↑	l →	; repeat t/T/f/F	' goto mk. bol	\ not used!	
Z quit <sup>4</sup>	X back-space	C change to eol	V visual lines	B prev WORD	N prev (find)	M screen mid'l	< un- <sup>3</sup> indent	> indent <sup>3</sup>	? find (rev.)			
Z extra <sup>5</sup>	X delete char	C change	V visual mode	b prev word	n next (find)	m set mark	, reverse	. repeat cmd	/ find			

**motion**

moves the cursor, or defines the range for an operator

**command**

direct action command, if red, it enters insert mode

**operator**

requires a motion afterwards, operates between cursor & destination

**extra**

special functions, requires extra input

**Q·**

commands with a dot need a char argument afterwards

bol = beginning of line, eol = end of line, mk = mark, yank = copy

words: guux([foo], bar, baz);

WORDs: guux(foo, bar, baz);

## Main command line commands ('ex'):

:w (save), :q (quit), :q! (quit w/o saving)  
 :e f (open file f),  
 :%s/x/y/g (replace 'x' by 'y' filewide),  
 :h (help in vim), :new (new file in vim),

## Other important commands:

CTRL-R: redo (vim),  
 CTRL-F/-B: page up/down,  
 CTRL-E/-Y: scroll line up/down,  
 CTRL-V: block-visual mode (vim only)

## Visual mode:

Move around and type operator to act on selected region (vim only)

## Notes:

(1) use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,\*)  
 (e.g.: "ay\$ to copy rest of line to reg 'a')

(2) type in a number before any action to repeat it that number of times  
 (e.g.: 2p, d2w, 5i, d4j)

(3) duplicate operator to act on current line (dd = delete line, >> = indent line)

(4) ZZ to save & quit, ZQ to quit w/o saving

(5) zt: scroll cursor to top,  
 zb: bottom, zz: center

(6) gg: top of file (vim only),  
 gf: open file under cursor (vim only)

# 如果你有块更大的屏幕

```
" Press ? for help
.. (up a dir)
</nemu/src/monitor/sdb/
expr.c
sdb.c
sdb.h
watchpoint.c

1 +... 6 lines: -----
7 #define CONFIG_DIFFTEST_REF_NAME "none"
8 #define CONFIG_ENGINE "interpreter"
9 #define CONFIG_PC_RESET_OFFSET 0x0
10 #define CONFIG_TARGET_NATIVE_ELF 1
11 #define CONFIG_MSIZE 0x8000000
12 #define CONFIG_CC_02 1
13 #define CONFIG_MODE_SYSTEM 1
14 #define CONFIG_MEM_RANDOM 1
15 #define CONFIG_ISA_riscv32 1
16 #define CONFIG_LOG_START 0
17 #define CONFIG_MBASE 0x80000000
18 #define CONFIG_TIMER_GETTIMEOFDAY 1
19 #define CONFIG_ENGINE_INTERPRETER 1
20 #define CONFIG_CC_OPT "-02"
21 #define CONFIG_LOG_END 10000
22 #define CONFIG_RT_CHECK 1
23 #define CONFIG_CC "gcc"
24 #define CONFIG_DIFFTEST_REF_PATH "none"
25 #define CONFIG_CC_DEBUG 1
26 #define CONFIG_CC_GCC 1
27 #define CONFIG_DEBUG 1
28 #define CONFIG_ISA "riscv32"

1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6 int is_exit_status_bad();
7
8 int main(int argc, char *argv[]) {
9     /* Initialize the monitor. */
10    #ifdef CONFIG_TARGET_AM
11        am_init_monitor();
12    #else
13        init_monitor(argc, argv);
14    #endif
15
16    /* Start engine. */
17    engine_start();
18
19    return is_exit_status_bad();
20 }

$ ls build include Makefile resource src
configs Kconfig README.md scripts tools
$ make
+ LD /home/why/Documents/ICS2021/ics2021/nemu/build/riscv
32-nemu-interpreter
$



<1/ics2021/nemu/src/monitor/sdb  <oconf.h[3] [cpp] unix utf-8 Ln 1, Col 1/28  <CS2021/ics2021/nemu/src/nemu-main.c[1] [c] unix utf-8 Ln 13, Col 5/20
-- INSERT --
[0] 0:vim*                                     S 英 韩 拼 8
"why-VirtualBox" 10:51 22-9月 -21
```

# VSCODE: 现代工具来一套?

- 刚拿到手, VSCODE 的体验并不是非常好
  - 满屏的红线/蓝线
  - 因为 Code 并知道 NEMU 是怎么编译的
  - IDE “编译运行” 背后没有魔法
- 另一方面, 这些东西一定是可以配置的
  - 配置解析选项: `c_cpp_properties.json`
    - 解锁正确的代码解析
  - 配置构建选项: `tasks.json`
    - 解锁 make (可跟命令行参数)
  - 配置运行选项: `launch.json`
    - 解锁单步调试 (我们并不非常推荐单步调试)

# 总结

# 怎样读代码？

---

- 读代码 ≠ “读” 代码
  - 用正确的工具，使自己感到舒适
  - 但这个过程本身可能是不太舒适的 (走出你的舒适区)
- 我们看到太多的同学，到最后都没有学会使用编辑器/IDE
  - 要相信一切不爽都有办法解决
- 信息来源
  - 在 /etc/hosts 中屏蔽百度
  - 去开源社区找 tutorials
    - 例子： vim-galore, awesome-c

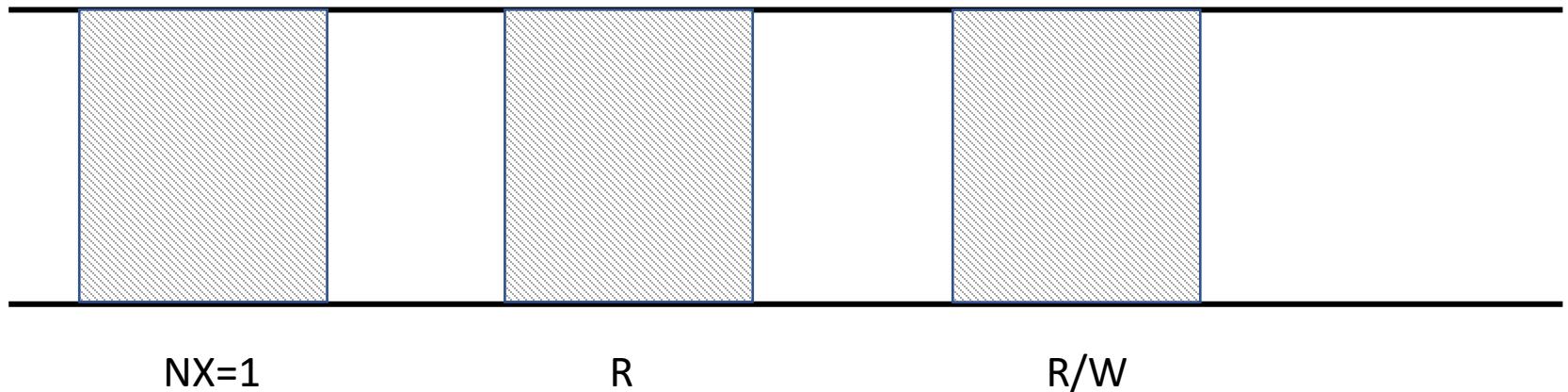
End.

# 插入福利：调试Segmentation Fault

- 听说你的程序又 Segmentation Fault 了?
  - 百度 Segmentation Fault 得到的回答的解释可能是完全错误的
- 正确的解释
  - 指令越权访问内存 (r/w/x)
    - 原因很多，数组越界、memory corruption, ...
  - 指令未被执行，进程收到 SIGSEGV 信号
    - 默认的信号处理程序会 core dump 退出
    - Gdb 调试：backtrace

# Segmentation Fault

- `*((int *) 0) = 0;`



- 指令越权访问内存 (r/w/x)

- SIGSEGV

- Core dumped
  - Gdb调试: backtrace

管理 控制 视图 热键 设备 帮助

```
1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6 int is_exit_status_bad();
7
8 int main(int argc, char *argv[]) {
9     /* Initialize the monitor. */
10    #ifdef CONFIG_TARGET_AM
11        am_init_monitor();
12    #else
13        init_monitor(argc, argv);
14    #endif
15
16    /* Start engine. */
17    engine_start();
18
19    return is_exit_status_bad();
20 }
```

```
$ ls
build configs include Kconfig Makefile README.md resource scripts src tools
$
```

~/Documents/ICS2021/ics2021/nemu/src/nemu-main.c[+1] [c] unix utf-8 Ln 13, Col 4/20

[0] 0:vim\*

"why - Vi 英 韩 拼 22-9月 -21



# 好的编辑器：有时候也许不是万能的

- 又爱又恨的元编程
- 办法 1: RTFM + RTFSC + 写小程序尝试
- 办法 2: 预编译以后的代码应该好理解！
  - 还记得我们对 Makefile 的导读吗？
    - (说的容易做得难。直接 gcc -E 不是编译错误吗.....)

```
#define def_INSTR_IDTABW(pattern, id, tab, width) \
    def_INSTR_raw(pattern_decode, pattern, \
        { concat(decode_, id)(s, width); return concat(table_, tab)(s); }) \
#define def_INSTR_IDTAB(pattern, id, tab)    def_INSTR_IDTABW(pattern, id, tab, 0) \
#define def_INSTR_TABW(pattern, tab, width)  def_INSTR_IDTABW(pattern, empty, tab, width) \
#define def_INSTR_TAB(pattern, tab)         def_INSTR_IDTABW(pattern, empty, tab, 0)

#define def_hex_INSTR_IDTABW(pattern, id, tab, width) \
    def_INSTR_raw(pattern_decode_hex, pattern, \
        { concat(decode_, id)(s, width); return concat(table_, tab)(s); }) \
#define def_hex_INSTR_IDTAB(pattern, id, tab)    def_hex_INSTR_IDTABW(pattern, id, tab, 0) \
#define def_hex_INSTR_TABW(pattern, tab, width)  def_hex_INSTR_IDTABW(pattern, empty, tab, width) \
#define def_hex_INSTR_TAB(pattern, tab)         def_hex_INSTR_IDTABW(pattern, empty, tab, 0)
```

```

// --- pattern matching wrappers for decode ---
#define INSTPAT(pattern, ...) do { \
    uint64_t key, mask, shift; \
    pattern_decode(pattern, STRLEN(pattern), &key, &mask, &shift); \
    if (((INSTPAT_INST(s) >> shift) & mask) == key) { \
        INSTPAT_MATCH(s, ##_VA_ARGS_); \
        goto *(_instpat_end); \
    } \
} while (0)

#define INSTPAT_START(name) { const void ** _instpat_end = &&concat( _instpat_end, name); \
#define INSTPAT_END(name) concat(_instpat_end, name);

#define INSTPAT_INST(s) ((s)->isa.inst)
#define INSTPAT_MATCH(s, name, type, ... \
    decode_operand(s, &dest, &src1, &src2, \
    __VA_ARGS__); \
}

INSTPAT_START();
INSTPAT("?????? ?? ???? ?? ???? 01101 11", lui      , U, R(dest) = src1);
INSTPAT("?????? ?? ???? 010 ????? 00000 11", lw       , I, R(dest) = Mr(src1 + src2, 4));
);
INSTPAT("?????? ?? ???? 010 ????? 01000 11", sw      , S, Mw(src1 + dest, 4, src2));

INSTPAT("0000000 00001 00000 000 00000 11100 11", ebreak , N, NEMUTRAP(s->pc, R(10))); // \
R(10) is $a0
INSTPAT("?????? ?? ???? ?? ???? ????? ??", inv     , N, INV(s->pc));
INSTPAT_END();

R(0) = 0; // reset $zero to 0

return 0;
}

```

```

#define src1R(n) do { *src1 = R(n); } while (0)
#define src2R(n) do { *src2 = R(n); } while (0)
#define destR(n) do { *dest = n; } while (0)
#define src1I(i) do { *src1 = i; } while (0)
#define src2I(i) do { *src2 = i; } while (0)
#define destI(i) do { *dest = i; } while (0)

```

```

#define R(i) gpr(i)
#define Mr vaddr_read
#define Mw vaddr_write

```

# Don't Give Up Easy

- 我们既然知道 Makefile 里哪一行是 .c → .o 的转换
  - 我们添一个一模一样的 gcc -E 是不是就行了？

```
$(OBJ_DIR)/%.o: src/%.c
@$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
@$(CC) $(CFLAGS) $(SO_CFLAGS) -E -MF /dev/null $< | \
grep -ve '^#' | \
clang-format - > $(basename $@).i
```

敲黑板：征服你畏惧的东西，就会有意想不到的收获。

# 关于OJ

Ubuntu 20.04

Gcc 9.4.0

主要方式：观察Nemu控制台输出检查

# recap: 自我管理git

重装系统了？

虚拟机崩溃了？

环境配置出问题了？

# PA2导读

Lab1和PA2接踵而至，不要慌，慢慢来。

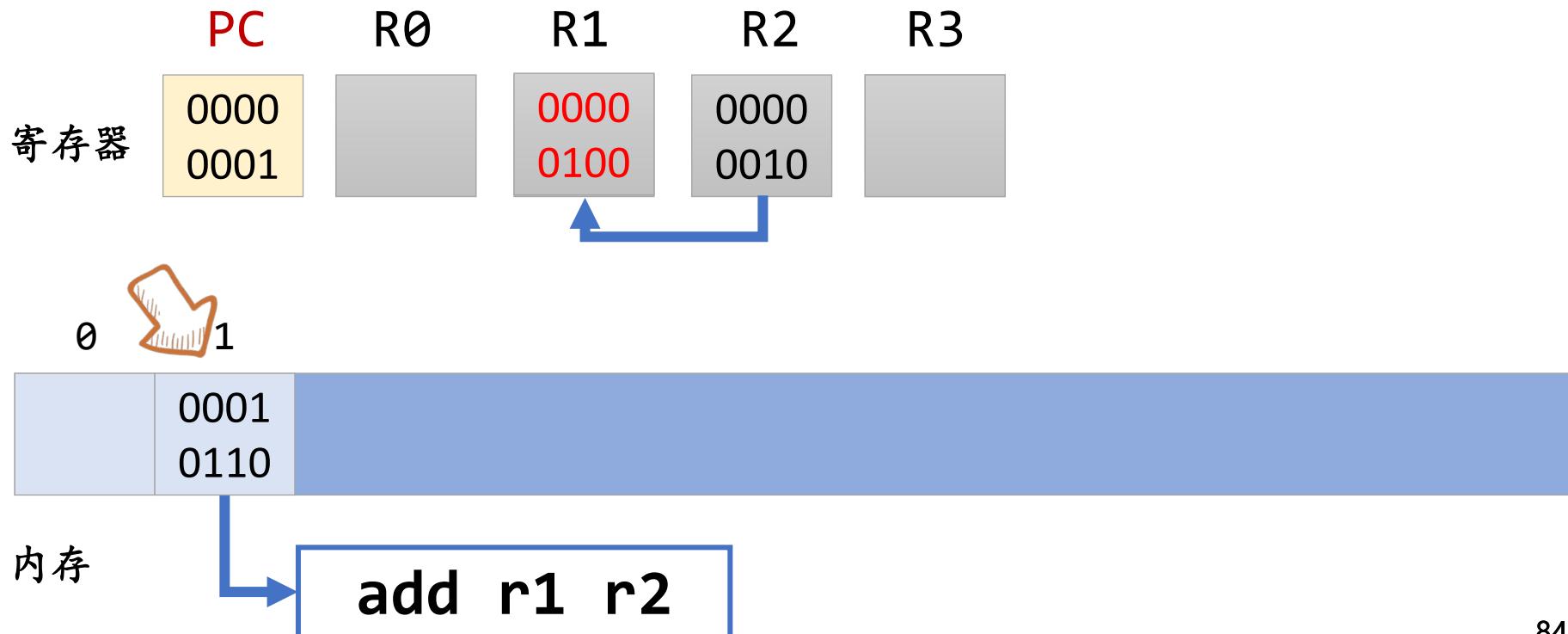
# 教科书第一章上的“计算机系统”

## 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



```

#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;
}

```

# Don't Give Up Easy

- 我们既然知道 Makefile 里哪一行是 .c → .o 的转换
  - 我们添一个一模一样的 gcc -E 是不是就行了？

```
$(OBJ_DIR)/%.o: src/%.c
@$(CC) $(CFLAGS) $(SO_CFLAGS) -c -o $@ $<
@$(CC) $(CFLAGS) $(SO_CFLAGS) -E -MF /dev/null $< | \
grep -ve '^#' | \
clang-format - > $(basename $@).i
```

```

static int decode_exec(Decode *s) {
    int dest = 0;
    word_t src1 = 0, src2 = 0, imm = 0;
    s->dhpc = s->snpc;

{
    const void **__instpat_end = &&__instpat_end_;
    ;
}

static int decode_exec(Decode *s) {
    int dest = 0;
    word_t src1 = 0, src2 = 0, imm = 0;
    s->dhpc = s->snpc;

#define INSTPAT_INST(s) ((s)->isa.inst.val)
#define INSTPAT_MATCH(s, name, type, ...) /* execute body */ ) { \
    decode_operand(s, &dest, &src1, &src2, &imm, concat(TYPE_, type)); \
    __VA_ARGS__ ; \
}

INSTPAT_START();
INSTPAT("?????? ?? ?? ?? ?? ?? ?? 01101 11", lui      , U, R(dest) = imm);
INSTPAT("?????? ?? ?? ?? ?? 010 ????? 00000 11", lw       , I, R(dest) = Mr(src1 + imm, 4));
INSTPAT("?????? ?? ?? ?? 010 ????? 01000 11", sw       , S, Mw(src1 + imm, 4, src2));

INSTPAT("00000000 00001 00000 000 00000 11100 11", ebreak   , N, NEMUTRAP(s->pc, R(10))); // R(10) is $a0
INSTPAT("?????? ?? ?? ?? ?? ????? ????? ??", inv     , N, INV(s->pc));
INSTPAT_END();

R(0) = 0; // reset $zero to 0

return 0;
}

```

```

        goto *(__instpat_end);
    }
} while (0);
do {
    uint64_t key, mask, shift;
    pattern_decode("?????? ?? ?? ?? ?? 010 ????? 01000 11",
                  (sizeof("?????? ?? ?? ?? ?? 010 ????? 01000 11") - 1),
                  &key, &mask, &shift);
    if (((uint64_t)((s)->isa.inst.val) >> shift) & mask) == key) {
    {
        decode_operand(s, &dest, &src1, &src2, &imm, TYPE_S);
        vaddr_write(src1 + imm, 4, src2);
    };
    goto *(__instpat_end);
    }
} while (0);

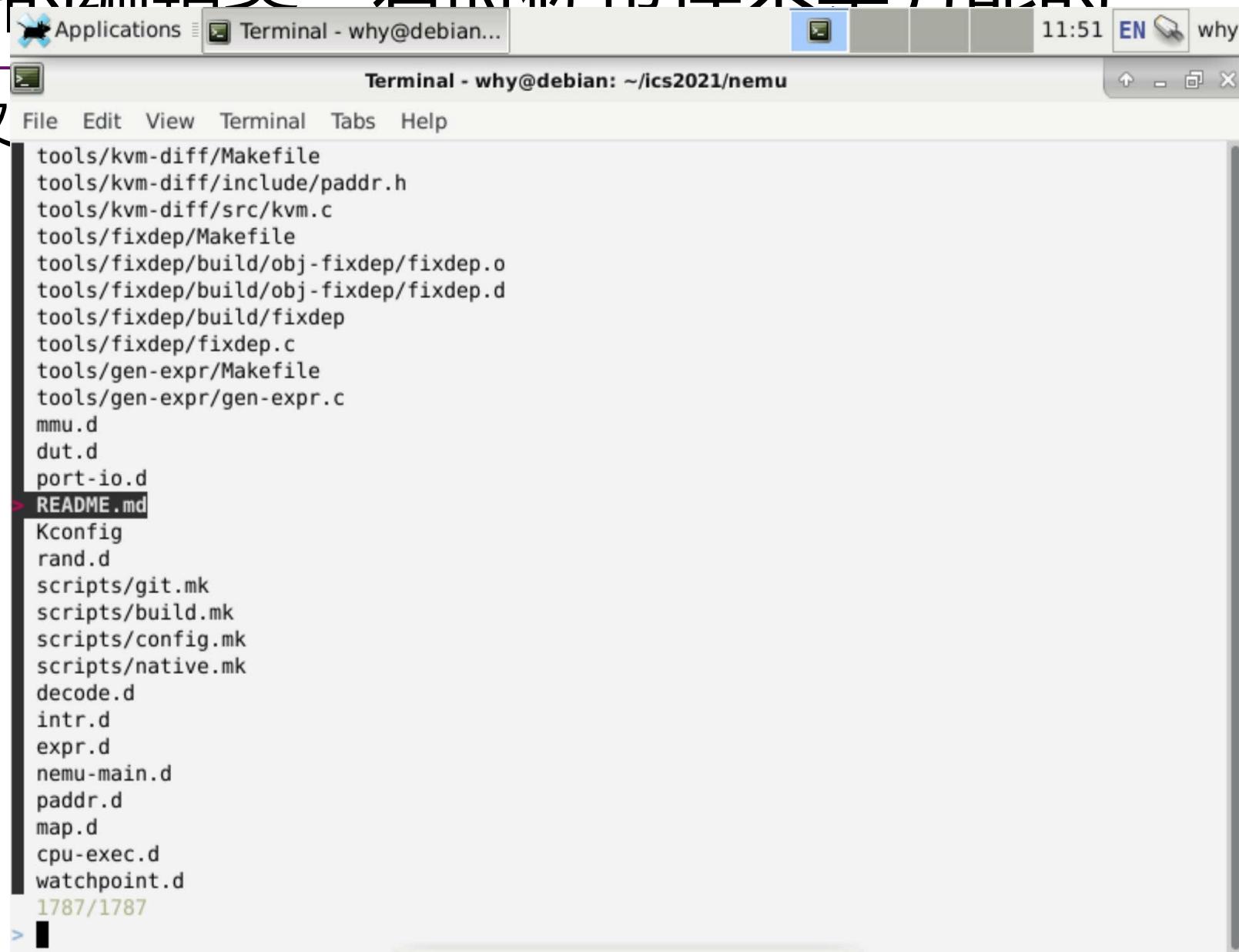
```

# 好的编辑器：有时候也许不是万能的

---

- 又爱又恨的元编程（视频？）

# 好的编辑器·有时候也许不是万能的



Applications Terminal - why@debian... 11:51 EN why

Terminal - why@debian: ~/ics2021/nemu

File Edit View Terminal Tabs Help

```
tools/kvm-diff/Makefile
tools/kvm-diff/include/paddr.h
tools/kvm-diff/src/kvm.c
tools/fixdep/Makefile
tools/fixdep/build/obj-fixdep/fixdep.o
tools/fixdep/build/obj-fixdep/fixdep.d
tools/fixdep/build/fixdep
tools/fixdep/fixdep.c
tools/gen-expr/Makefile
tools/gen-expr/gen-expr.c
mmu.d
dut.d
port-io.d
> README.md
Kconfig
rand.d
scripts/git.mk
scripts/build.mk
scripts/config.mk
scripts/native.mk
decode.d
intr.d
expr.d
nemu-main.d
paddr.d
map.d
cpu-exec.d
watchpoint.d
1787/1787
>
```

• 又

# 数据的机器级表述

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 提醒

- PA

- PA1 本周日晚23:59:59截止
- PA2:
  - PA 2.1: 2023年10月22日 (此为建议的不计分 deadline)
  - PA 2.2: 2023年11月05日 (此为建议的不计分 deadline)
  - PA 2.3: 2023年11月19日 23:59:59 (以此 deadline 计按时提交奖励分)

- Lab

- Lab1即将截止  
Deadline: 2023年10月22日

# 本讲概述

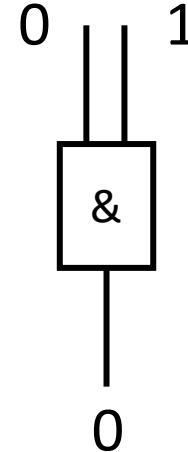
我们已经知道数据是如何在计算机中表示的。但为什么要这样表示？这样的表示有什么好处和用法？

- 位运算与单指令多数据
- 整数溢出与 Undefined Behavior
- IEEE754 浮点数

# 位运算与单指令多数据

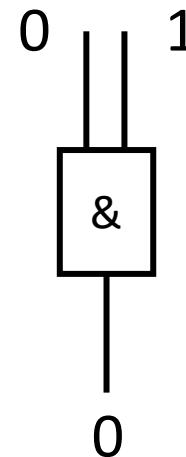
# 为什么会有位运算

- 逻辑门和导线是构成计算机(组合逻辑电路)的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)



# 位运算与逻辑电路密不可分

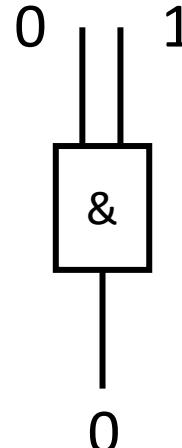
a ->	0	1	0	1	5
b ->	0	1	1	0	10
	↓				
a & b ->	0	1	0	0	8



加法呢？

# 为什么会有位运算

- 逻辑门和导线是构成计算机(组合逻辑电路)的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)
  - 例子：一代传奇处理器 8-bit [MOS 6502](#)
    - 3510 晶体管；56 条指令，算数指令仅有加减法和位运算



## Instructions by Name

[ADC](#) .... add with carry  
[AND](#) .... and (with accumulator)  
[ASL](#) .... arithmetic shift left  
[BCC](#) .... branch on carry clear  
[BCS](#) .... branch on carry set  
[BEQ](#) .... branch on equal (zero set)  
[BIT](#) .... bit test  
[BMI](#) .... branch on minus (negative set)  
[BNE](#) .... branch on not equal (zero clear)  
[BPL](#) .... branch on plus (negative clear)  
[BRK](#) .... break / interrupt

[BVC](#) .... branch on overflow clear  
[BVS](#) .... branch on overflow set  
[CLC](#) .... clear carry  
[CLD](#) .... clear decimal  
[CLI](#) .... clear interrupt disable  
[CLV](#) .... clear overflow  
[CMP](#) .... compare (with accumulator)  
[CPX](#) .... compare with X  
[CPY](#) .... compare with Y  
[DEC](#) .... decrement  
[DEX](#) .... decrement X  
[DEY](#) .... decrement Y

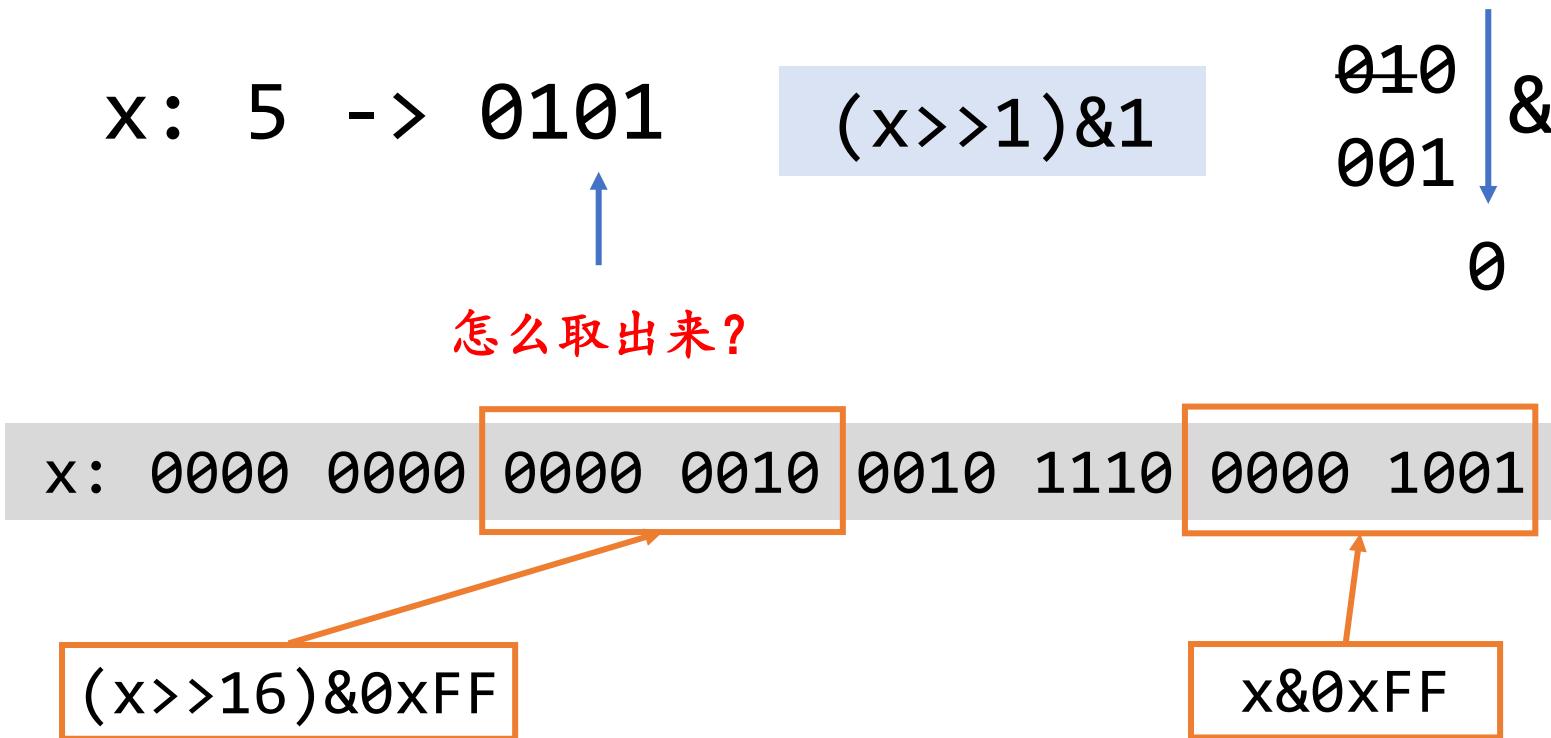
# 为什么会有位运算

- 逻辑门和导线是构成计算机 (组合逻辑电路) 的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)
  - 例子：一代传奇处理器 8-bit [MOS 6502](#)
    - 3510 晶体管；56 条指令，算数指令仅有加减法和位运算
  - 数学上自然的整数需要实现成固定长度的 01 字符串
- 习题：用上述位运算和常数实现 4 位整数的加法运算/Lab1
  - 加法比上述运算在电路上实现 fundamentally 更困难 (为什么？)
    - “Circuit Complexity”

# 整数：固定长度的 Bit String

- 142857 -> 0000 0000 0000 0010 0010 1110 0000 1001
  - 假设 32-bit 整数；约定 MSB 在左，LSB 在右

- 热身问题：字符串操作
  - 分别取出 4 个字节



# 整数：固定长度的 Bit String

- 142857 -> 0000 0000 0000 0010 0010 1110 0000 1001
  - 假设 32-bit 整数；约定 MSB 在左，LSB 在右

- 热身问题：字符串操作

- 交换高/低 16 位

x: 0000 0000 0000 0010 0010 1110 0000 1001

a

b

c

d

c d 0 0



$((x \& 0xFFFF) \ll 16)$

0 0 a b



$((x >> 16) \& 0xFFFF)$

# 单指令多数据

`&`, `|`, `~`, ... 对于整数里的每一个 bit 来说是独立（并行）的

- 如果我们操作的对象刚好每一个 bit 是独立的
  - 我们在一条指令里就实现了多个操作
  - SIMD (Single Instruction, Multiple Data)
- 例子： Bit Set
  - 32-bit 整数  $x \rightarrow S \subseteq \{0, 1, 2, 3, \dots, 31\}$
  - 位运算是对所有 bit 同时完成的
    - C++ 中有 `bitset`, 性能非常可观

5 -> 0101

$S: \{0, 2\}$

# Bit Set: 基本操作

- 测试  $x \in S$ 
  - $(S \gg x) \& 1$

5 -> 0101 ->  $S: \{0, 2\}$

**S**

- 求  $S' = S \cup \{x\}$ 
  - $S \mid (1 \ll x)$

0101 | (0001 << 1)  
0101 | 0010 -> **0111**

- 更多习题
  - 求  $|S|$
  - 求  $S_1 \cup S_2, S_1 \cap S_2$
  - 求  $S_1 \setminus S_2$
  - 遍历  $S$  中的所有元素 (foreach)

$S_1: \{0, 2\}, S_2: \{1, 2\}$

0101      0110

| : **0111**

& : **0100**

$S_1 \setminus S_2: 0001$

0	0	0
1	1	0
0	1	0
1	0	1

**$S_1 \& (\sim S_2)$**

# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)

- 测试  $x \in S$ 
  - $(S \gg x) \& 1$

5 -> 0101 ->  $S: \{0, 2\}$

```
int bitset_size(uint32_t S) {  
    int n = 0;  
    for (int i = 0; i < 32; i++) {  
        n += bitset_contains(S, i);  
    }  
    return n;  
}
```

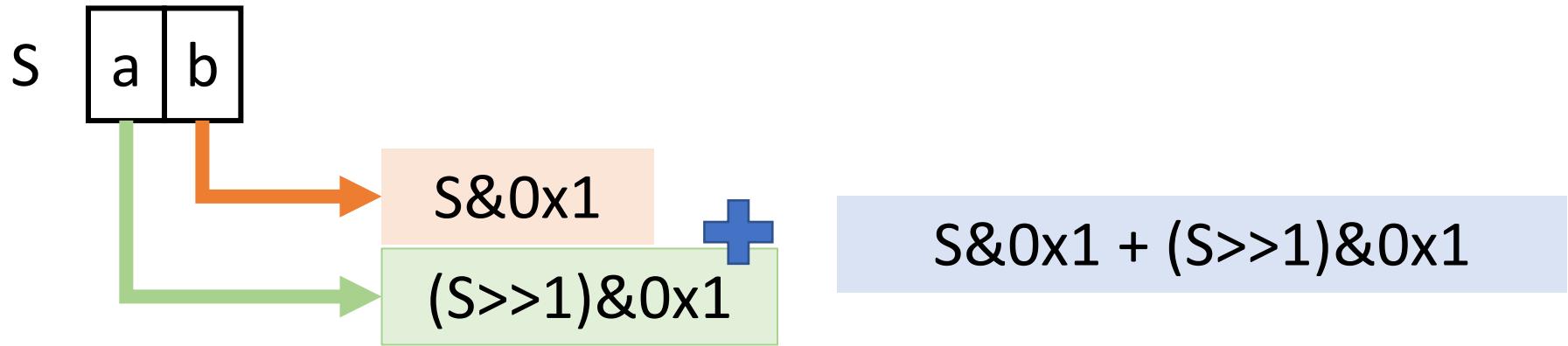
# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)

```
int bitset_size(uint32_t S) {
    int n;
    for (int i = 0; i < 32; i++) {
        n += bitset_contains(S, i);
    }
    return n;
}
```

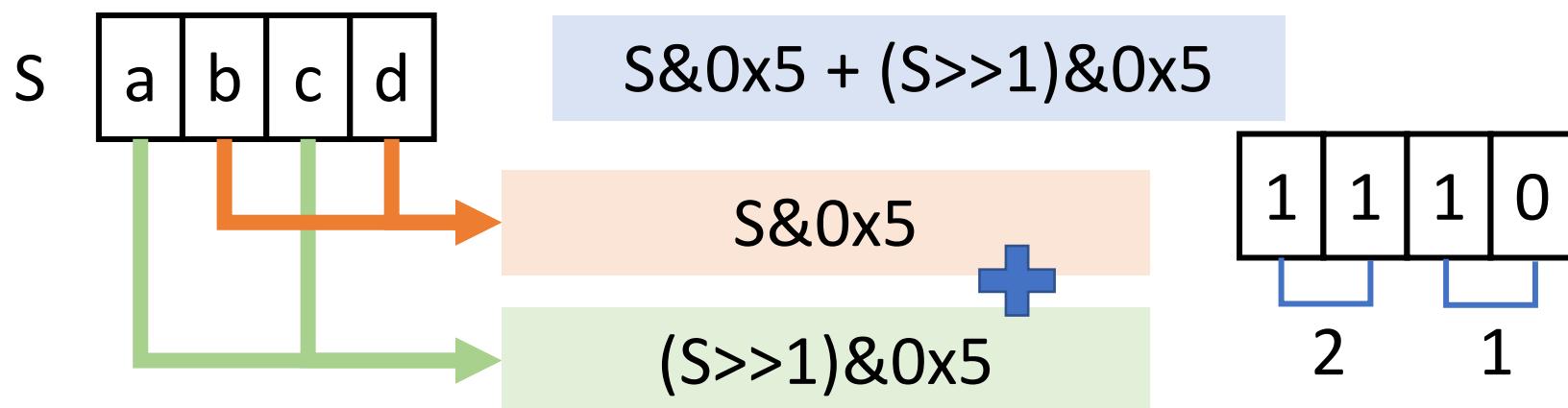
```
int bitset_size1(uint32_t S) { // SIMD
    S = (S & 0x55555555) + ((S >> 1) & 0x55555555);
    S = (S & 0x33333333) + ((S >> 2) & 0x33333333);
    S = (S & 0x0F0F0F0F) + ((S >> 4) & 0x0F0F0F0F);
    S = (S & 0x00FF00FF) + ((S >> 8) & 0x00FF00FF);
    S = (S & 0x0000FFFF) + ((S >> 16) & 0x0000FFFF);
    return S;
}
```

# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)

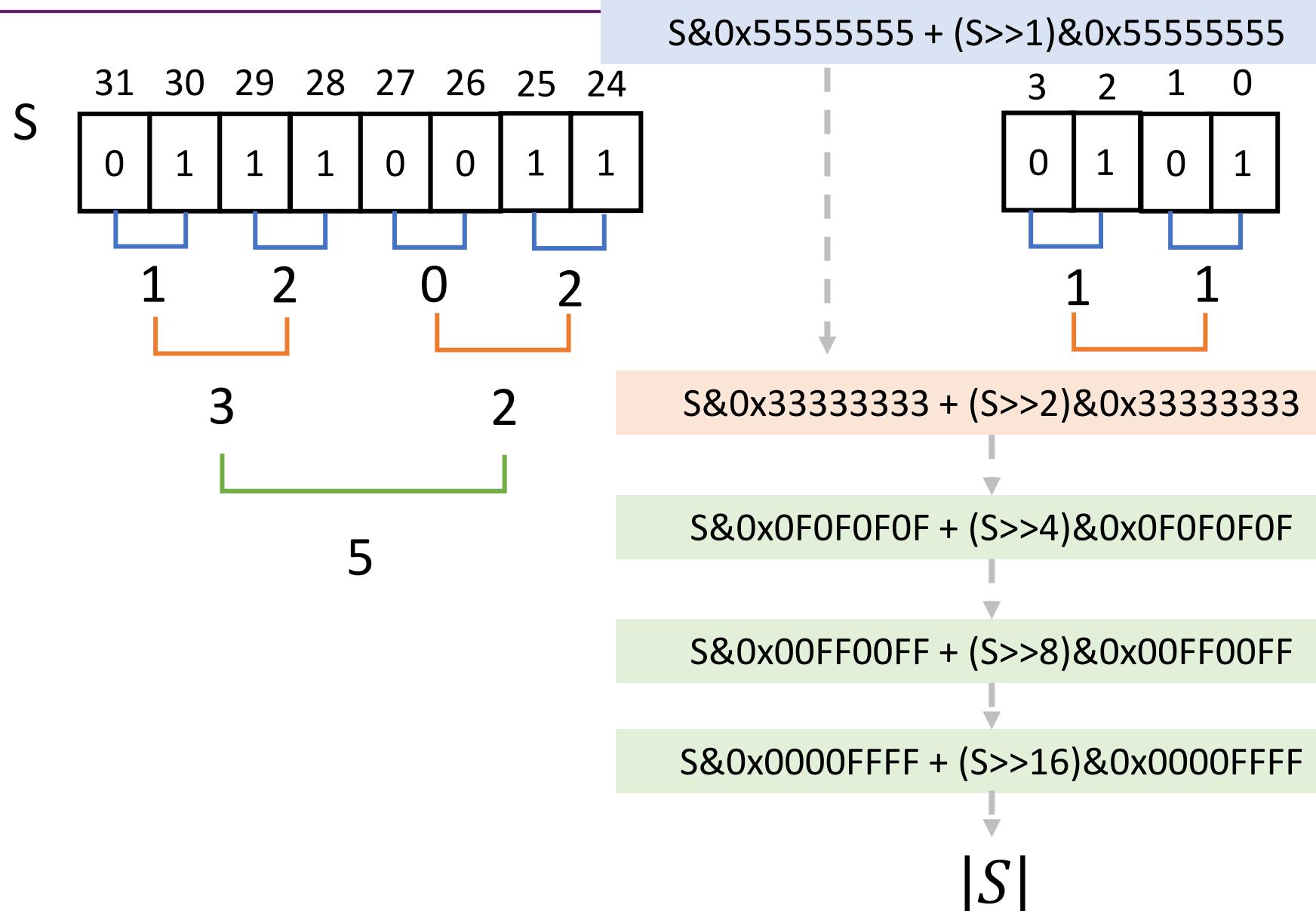
- $S: 0b10 \rightarrow S = \{1\}$



- $S: 0b1110 \rightarrow S = \{1, 3\}$



# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)



# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)

```
int bitset_size(uint32_t S) {
    int n;
    for (int i = 0; i < 32; i++) {
        n += bitset_contains(S, i);
    }
    return n;
}
```

```
int bitset_size1(uint32_t S) { // SIMD
    S = (S & 0x55555555) + ((S >> 1) & 0x55555555);
    S = (S & 0x33333333) + ((S >> 2) & 0x33333333);
    S = (S & 0x0F0F0F0F) + ((S >> 4) & 0x0F0F0F0F);
    S = (S & 0x00FF00FF) + ((S >> 8) & 0x00FF00FF);
    S = (S & 0x0000FFFF) + ((S >> 16) & 0x0000FFFF);
    return S;
}
```

# 遍历 S 中的元素

```
x=5 -> 0101
```

0101:  $x^{\wedge}(1 << 0)$

0100:  $x^{\wedge}(1 << 2)$

- Lowbit: 找到最右边的1

- $x^{\wedge}\text{lowbit}(x)$

- $x \& -x$

- 这是怎么来的?

# Bit Set: 返回 $S \neq \emptyset$ 中的某个元素

- 有二进制数  $x = 0b+++++100$ , 我们希望得到最后那个100
  - 想法: 使用基本操作构造一些结果, 能把++++的部分给抵消掉

表达式	结果
$x$	$0b+++++100$
$x-1$	$0b+++++011$
$\sim x$	$0b-----011$
$\sim x+1$	$0b-----100$

- 一些有趣的式子:
  - $x \& (x-1) \rightarrow 0b+++++000$ ;  $x ^ (x-1) \rightarrow 0b00000111$ ;
  - $x \& (\sim x+1) \rightarrow 0b00000100$  (*lowbit*)
    - $x \& -x$ ,  $(\sim x \& (x-1))+1$  都可以实现*lowbit*
    - 只遍历存在的元素可以加速求 $|S|$

# Lowbit: x&-x

- 无符号整数
  - 32位整数: 0x0 ~ 0xFFFFFFFF (32个1)
- 带符号整数
  - 原码表示法: 最高位为符号位
  - 补码表示法 (普遍采用) : 各位取反末位加一
    - 用加法来实现减法

$x = 0b+++++100$

$-x = 0b-----011+0b1 = 0b-----100$

$x \& -x: 0b00000100 \rightarrow \text{lowbit}$

# Bit Set: 求 $\lfloor \log_2(x) \rfloor$

- 等同于 $31 - \text{clz}(x)$

```
int __builtin_clz(uint32_t x) {  
    int n = 0;  
    if (x <= 0x0000ffff) n += 16, x <= 16;  
    if (x <= 0x00ffffff) n += 8, x <= 8;  
    if (x <= 0xfffffff) n += 4, x <= 4;  
    if (x <= 0x3fffffff) n += 2, x <= 2;  
    if (x <= 0x7fffffff) n ++;  
    return n;  
}
```

x: 0x00000001	16
x: 0x00010000	24
x: 0x01000000	28
x: 0x03000000	30

$S = \{0\}$       31

- (奇怪的代码) 假设 $x$ 是lowbit的结果?

```
#define LOG2(x) \  
    ("`01J2GK-3@HNL;-=47A-IFO?M:<6-E>95D8CB"[(x) % 37] - '0')
```

# Bit Set: 求 $\lfloor \log_2(x) \rfloor$ (cont'd)

- 用一点点元编程 (meta-programming) ; 试一试[log2.c](#)

```
import json

n, base = 64, '0'
for m in range(n, 10000):
    if len({(2**i) % m for i in range(n)}) == n:
        M = {j: chr(ord(base) + i)
              for j in range(0, m)
              for i in range(0, n)
              if (2**i) % m == j}
        break

magic = json.dumps(''.join(
    [M.get(j, '-') for j in range(0, m)])
).strip('"')

print(f'#define LOG2(x) ("{magic}")[({x} % {m}) - \'{base}\']')
```

\$

S 英卽简拼

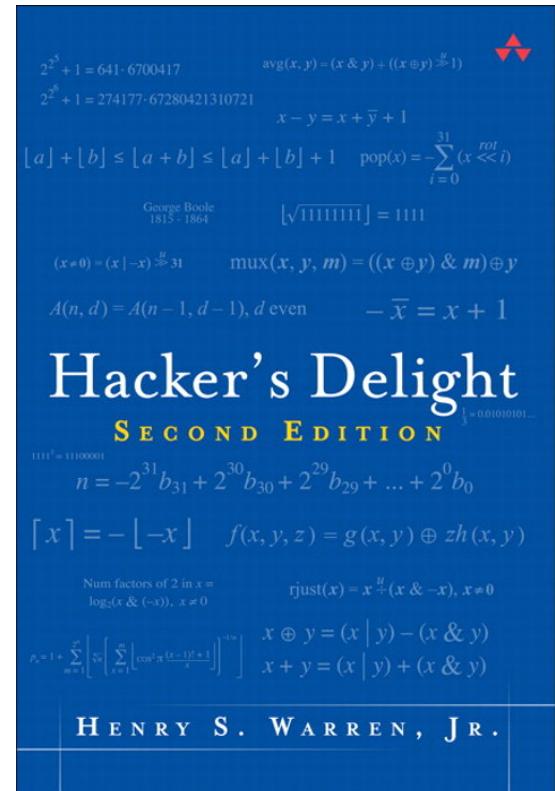


Right Shift + Right Alt

# 一本有趣的参考书

Henry S. Warren, Jr. *Hacker's Delight* (2ed), Addison-Wesley, 2012.

- 让你理解写的更快的代码并不是“瞎猜”
  - 主要内容是各种数学（带来的代码优化）
  - 官方网站：[hackersdelight.org](http://hackersdelight.org)
  - 见识一下真正的“奇技淫巧”



# 整数溢出与 Undefined Behavior

# Undefined Behavior (UB)

*Undefined behavior* (UB) is the result of executing computer code whose behavior is not prescribed by the language specification to which the code adheres, for the current state of the program. This happens when the translator of the source code makes certain assumptions, but these assumptions are not satisfied during execution. -- Wikipedia

- C对UB的行为是不做任何约束的，把电脑炸了都行
  - 常见的 UB：非法内存访问（空指针解引用、数组越界、写只读内存等）、被零除、**有符号整数溢出**、函数没有返回值.....
    - 通常的后果比较轻微，比如 wrong answer, crash

# 为什么 C/C++ 会有 UB?

---

- 为了尽可能高效 (zero-overhead)
  - 不合法的事情的后果只好 undefined 了
  - Java, js, python, ... 选择所有操作都进行合法性检查
- 为了兼容多种硬件体系结构
  - 有些硬件 /0 会产生处理器异常
  - 有些硬件啥也不发生
  - 只好 undefined 了

# Undefined Behavior: 一个历史性的包袱

- 埋下了灾难的种子
  - CVE: *Common Vulnerabilities and Exposures*, 公开发布软件中的漏洞
    - buffer/integer overflow 常年占据 CVE 的一席之地
    - 高危漏洞让没有修补的机器立马宕机/变成肉鸡
- 例子: [CVE-2018-7445](#) (RouterOS), 仅仅是忘记检查缓冲区大小.....

```
while (len) {  
    for (i = offset; (i - offset) < len; ++i) {  
        dst[i] = src[i+1];  
    }  
    len = src[i+1]; ...  
    offset = i + 1;  
}
```

# Undefined Behavior: 警惕整数溢出

表达式	值
<code>UINT_MAX+1</code>	0
<code>INT_MAX+1; LONG_MAX+1</code>	undefined
<code>char c = CHAR_MAX; c++;</code>	varies (???)
<code>1 &lt;&lt; -1</code>	undefined
<code>1 &lt;&lt; 0</code>	1
<code>1 &lt;&lt; 31</code>	undefined
<code>1 &lt;&lt; 32</code>	undefined
<code>1 / 0</code>	undefined
<code>INT_MAX % -1</code>	undefined

- W. Dietz, et al. Understanding integer overflow in C/C++.  
In *Proceedings of ICSE*, 2012.

# 整数溢出和编译优化

```
int f() { return 1 << -1; }
```

- 根据手册，这是个UB，于是clang这样处置

```
0000000000000000 <f>:  
0: c3      retq
```

- 编译器把这个计算直接删除了

W. Xi, et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of SOSP*, 2013.

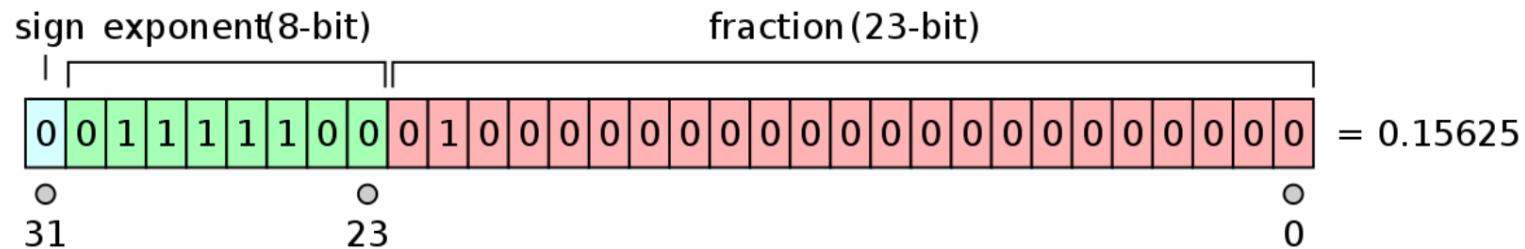
- `if(UB==0) yes; else if(UB!=0) no; //???`

浮点数： IEEE 754

# 实数的计算机表示

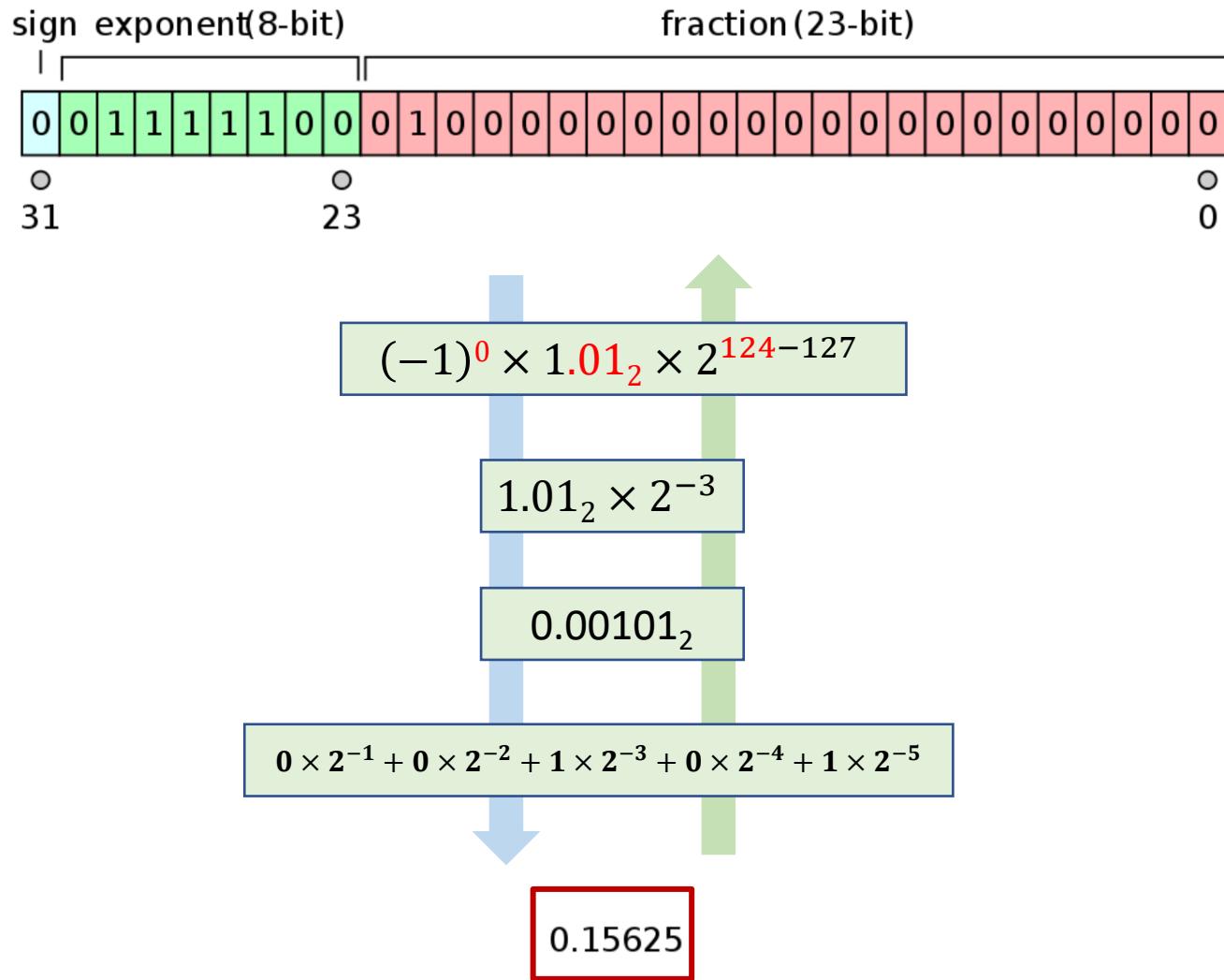
- 实数非常非常多
  - 只能用32/64-bit 01串来表述一小部分实数
    - 确定一种映射方法，把一个01串映射到一个实数
      - 运算起来不太麻烦
      - 计算误差不太可怕
- 于是有了IEEE754
  - 1bit S, 23/53bits Fraction (尾数), 8/11bits Exponent (阶码),

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$



# 实数的计算机表示

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$



# IEEE754: 你有可能不知道的事实

---

- 一个有关浮点数大小/密度的实验([float.c](#))
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化

\$

S 英 拼

Right Shift + Right Alt

# IEEE754: 你有可能不知道的事实

---

- 一个有关浮点数大小/密度的实验([float.c](#))
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化
- $1.00000000000000000000000_2 \times 2^{24}$ 
  - 16,777,216
- $1.00000000000000000000001_2 \times 2^{24}$ 
  - 16,777,218

# IEEE754: 你有可能不知道的事实

- 一个有关浮点数大小/密度的实验([float.c](#))
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化
- 例子：计算 $1 + \frac{1}{2} + \frac{1}{3} \dots + \frac{1}{n}$

```
#define SUM(T, st, ed, d) ({ \
    T s = 0; \
    for (int i = st; i != ed + d; i += d) \
        s += (T)1 / i; \
    s; \
})
```

```
1 #define SUM(T, st, ed, d) ([ \n
2     T s = 0; \
3     for (int i = st; i != ed + d; i += d) \
4         s += (T)1 / i; \
5     s; \
6 ])\n7\n8 #define n 1000000\n9\n10 int main(){\n11     printf("%.16f\n", SUM(float, 1, n, 1));\n12     printf("%.16f\n", SUM(float, n, 1, -1));\n13     printf("%.16f\n", SUM(double, 1, n, 1));\n14     printf("%.16f\n", SUM(double, n, 1, -1));\n15 }
```

16

~/Documents/ICS2021/teach/sum.c[1]

[c] unix utf-8 Ln 1, Col 1/16

S 英 拼



# IEEE754: 你可能不知道的事实 (cont'd)

- 比较
  - $a == b$  需求谨慎判断 (要假设自带 $\varepsilon$ )
  - 浮点数:  $(a + b) + c \neq a + (b + c)$
- 非规格化数 (Exponent == 0)
  - $x = (-1)^S \times (0.F) \times 2^{-126}$
- 零
  - $+0.0, -0.0$  的 Sbit 是不一样的, 但  $+0.0 == -0.0$
- Inf/NaN (Not a Number)
  - Inf: 浮点数溢出很常见, 不应该作为 undefined behavior
  - NaN( $0.0/0.0$ ): 能够满足  $x != x$  表达式的值

# IEEE754：异常复杂

- 除了 $x = (-1)^S \times (1.F) \times 2^{E-B}$ , 还要考虑
  - 非规格化数, +0.0/-0.0, Inf, NaN
    - 一度引起了硬件厂商的众怒 (碰到非规格数干脆软件模拟吧)
    - 很多“对浮点数精度要求不高”硬件厂商选择不兼容 IEEE 754 (比如各种 GPU 制造商)
    - Nvidia 从 Fermi 才开始完整支持 IEEE754 (2010)

An interview with the old man of floating-point.

Reminiscences elicited from William Kahan by Charles Severance.

例子：计算  $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$

---

- 如果考虑比较极端的数值条件？
  - 消去误差： $-b$  v.s.  $\sqrt{b^2 - 4ac}$  (catastrophic cancellation)
    - 还记得  $x + 1.0 == x$  的例子吗
  - 溢出： $b^2$
- 一个更好的一元二次方程求根公式
  - $\frac{(-b)^2 - (\sqrt{b^2 - 4ac})^2}{(-b) + \sqrt{b^2 - 4ac}} / 2a = \frac{4ac}{(-b) + \sqrt{b^2 - 4ac}} / 2a \ (b < 0)$
  - $(-b - \sqrt{b^2 - 4ac}) \cdot \frac{1}{2a} \ (0 \leq b \leq 10^{127})$
  - $-\frac{b}{a} + \frac{c}{b} \ (b > 10^{127})$ 
    - P. Panekha, et al. Automatically improving accuracy for floating point expressions. In *Proc. of PLDI*, 2015.

# 凭什么？

It looked pretty complicated. On the other hand, we had a rationale for everything. -- [William Kahan](#), 1989 ACM Turing Award Winner for his *fundamental contributions to numerical analysis.*

浮点数可以直接当成整数比较大小

+0.0/-0.0和Inf保证 $(^1/x)/x$ 不会发生sign shift

.....

- IEEE754天才的设计保证了数值计算的稳定
  - 如果想了解IEEE754, 请阅读D. Goldberg. [What every computer scientist should know about floating-point arithmetic](#). *ACM Computing Surveys*, 23(1), 1991.

# 数据的机器级表示：综合案例

# 案例：计算 $1/\sqrt{x}$

---

- 应用：Surface Norm

- 平面的法向量
  - 用于计算光照/反射
  - ~~第一次感觉数学派上了用场~~
- 计算方法
  - 幂级数展开
  - 二分法
  - 牛顿法.....

- 如何不借助硬件指令，快速（近似）计算 $f(x)$ ？

- 人眼对像素级的误差并不敏感，因此算错一点也没事
- 快1.5倍： $640 \times 480 \rightarrow 800 \times 600$ （这个推导并不严格）

# 神奇的 $O(1)$ 代码

```
float Q_rsqrt( float number ) {
    union { float f; uint32_t i; } conv;
    float x2 = number * 0.5F;
    conv.f = number;
    conv.i = 0x5f3759df - ( conv.i >> 1 ); // ???
    conv.f = conv.f * ( 1.5F - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

- 看看别人的毕业设计

- Matthew Robertson. *A Brief History of InvSqrt, Bachelor Thesis, The University of New Brunswick, 2012.*

# 案例：计算 $(a \times b) \bmod m$

```
int64_t multimod_fast(int64_t a, int64_t b, int64_t m) {  
    int64_t x = (int64_t)((double)a * b / m) * m;  
    int64_t t = (a * b - x) % m;  
    return t < 0 ? t + m : t;  
}
```

- 令 $a \times b = p \cdot m + q$ 
  - $a \star b \rightarrow (p \cdot m + q) \bmod 2^{64}$
  - $x \rightarrow \left(\left\lfloor \frac{p \cdot m + q}{m} \right\rfloor \cdot m\right) \bmod 2^{64}$ 
    - 如果浮点数的精度是无限的， $x = (p \cdot m) \bmod 2^{64}$
    - 同余就得到了 $q$

# 总结

# 数据的机器级表示

---

- *Everything is a bit-string!*
  - 但是却能玩出很多花样来
    - bit-set/SIMD
    - 浮点数的表示
- PA：禁止写出不可维护的代码
  - 如果你想玩出花来，请做合适的封装和充分的测试

End.

# recap: 自我管理git

重装系统了？

虚拟机崩溃了？

环境配置出问题了？

# PA2导读

Lab1和PA2接踵而至

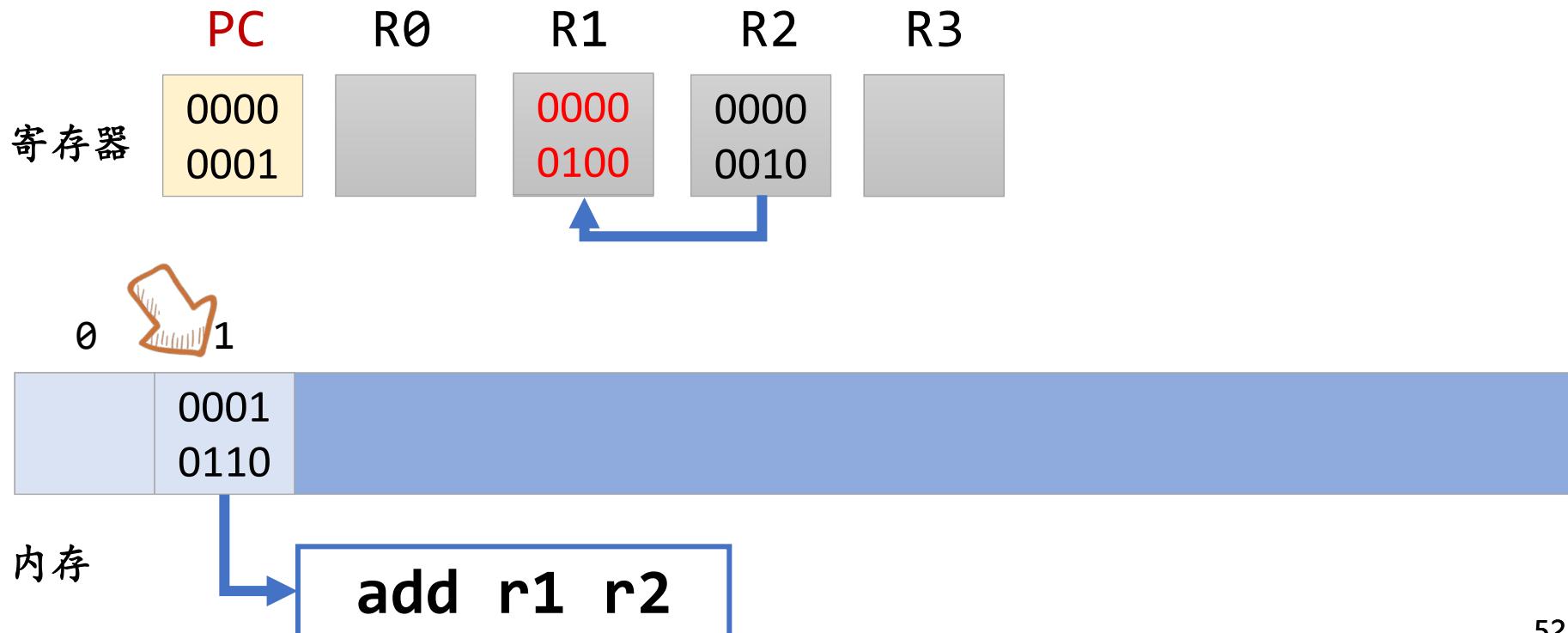
# 教科书第一章上的“计算机系统”

## 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	



```

#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;
}

```

# x86-64与内联汇编

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

课堂上/PA 以 x86 (IA32) 为主授课?

今天连手机都是 64-bit 了……

- 本讲内容

- 一些背景
- x86-64 体系结构与汇编语言
- inline assembly

# 机器字长的发展

# 字长 (Word size)

In computing, a word is the natural unit of data used by a particular processor design. The *number of bits in a word* (the word size, word width, or word length)...

- 能直接进行整数/位运算的大小
- 指针的大小 (索引内存的范围)

# 8 位机 (6502)

- 16 bit PC 寄存器 (64 KiB 寻址能力, KiB 级内存, 无乘除法/浮点数)
  - Apple II; Atari 2600; NES; ...



# 16位机 (8086/8088)

- 我们需要更大的内存！更大的数据宽度！
  - 20 bit 地址线 (两个寄存器)



# 32 位机 (Intel x86)

- 8086 处理器 4,096 倍的地址空间
  - 4 GiB 内存在 1980s 看起来是非常不可思议的



# 64 位机 (x86-64; AArch64; RV64; ...)

- 64 位地址空间能索引 17,179,869,184 GiB 内存
  - 我们的服务器有 128 GiB 内存
  - 目前看起来是非常够用的 (PML4)
    - 现在的处理器一般实现 48 bit 物理地址 (256 TiB)



# Fun Facts

---

- int类型的长度
  - 8 bit computer: 8 bit
  - 16 bit computer: 16 bit
  - 32 bit computer: 32 bit
  - 64 bit computer: 32 bit
  - JVM (32/64 bit): 32 bit
- 在逻辑世界里描述日常世界, 2,147,483,647 已经足够大了

# 概念复习：ABI

# 程序的机器级表示

---

- 程序经历：.c → .o (编译)和 .o → a.out (链接)
  - 不同版本、不同编译器、不同语言的二进制文件都可以链接
    - 他们需要一个“共同语言”
- 例如：我们熟悉的x86 calling convention
  - cdecl (Linux)
  - stdcall (Win32)
  - 只要遵循标准的函数就可以互相调用

# Application Binary Interface (ABI)

- 区别于API (Application Programming Interface)
  - 程序源代码中的规范
- ABI: 约定binary的行为
  - 二进制文件的格式
  - 函数调用、系统调用.....
    - C语言规范只定义了运行时内存和内存上的计算
    - printf都无法实现，必须借助外部的库函数
  - 链接、加载的规范



The screenshot shows a terminal window with the following details:

- Title bar: stdio.h
- Path: D: > gcc > mingw64 > x86\_64-w64-mingw32 > include > stdio.h
- Content:

```
424 #pragma GCC diagnostic push
425 #pragma GCC diagnostic ignored "-Wshadow"
426 #endif
427 __attribute__((__format__(__MINGW_PRINTF_FORMAT, 2, 3))) __MINGW_ATTRIB_NONNULL(2)
428 int __cdecl fprintf(FILE * __restrict__ _File, const char * __restrict__ _Format, ...);
429 __attribute__((__format__(__MINGW_PRINTF_FORMAT, 1, 2))) __MINGW_ATTRIB_NONNULL(1)
430 int __cdecl printf(const char * __restrict__ _Format, ...);
431 __attribute__((__format__(__MINGW_PRINTF_FORMAT, 2, 3))) __MINGW_ATTRIB_NONNULL(2)
432 int __cdecl sprintf(char * __restrict__ _Dest, const char * __restrict__ _Format, ...) __M
```

# 例子：cdecl函数调用

- caller stack frame

- 所有参数以数组的形式保存在堆栈上（所以才有“反序压栈”）
- 然后是返回地址
- 跳转到callee

- callee

- EAX作为返回值
- 其他寄存器都是callee save

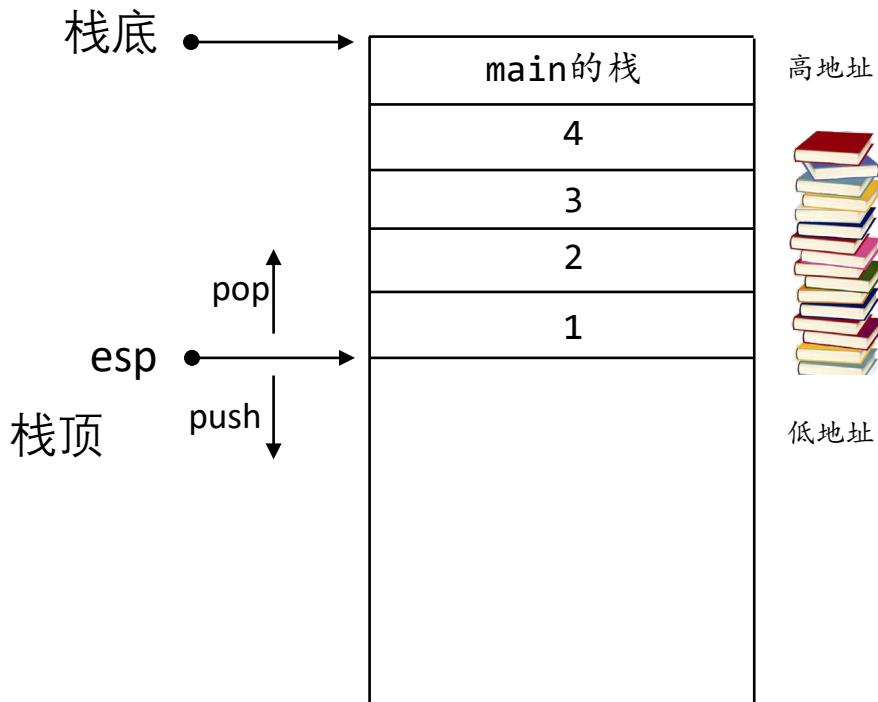
```
void bar(int *);  
int foo(int x) {  
    bar(&x);  
    return x;  
}
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```

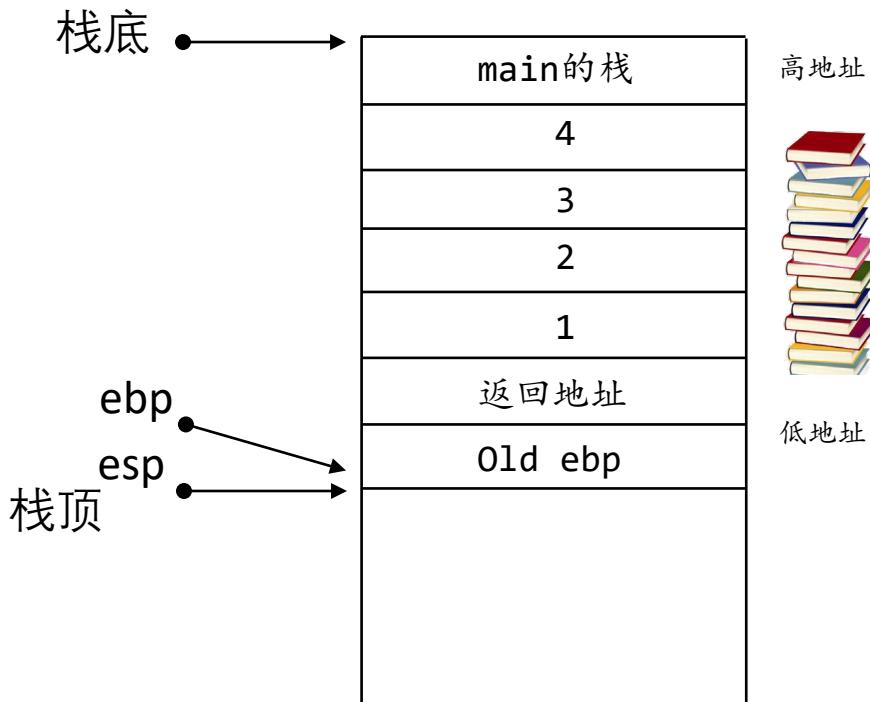
```
int main() { foo(1,2,3,4); }
```



# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



```
int main() { foo(1,2,3,4); }
```

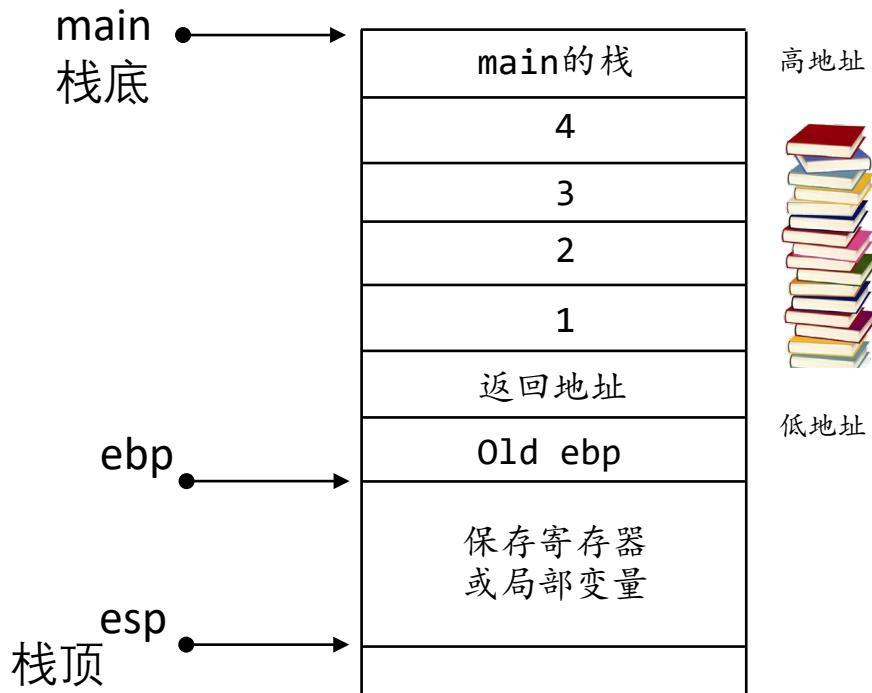
Disassembly of section .text:

```
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55               push %ebp  
 5: 89 e5             mov %esp,%ebp  
 7: e8 fc ff ff ff   call 8 <foo+0x8>  
 c: 05 01 00 00 00    add $0x1,%eax  
11: 8b 55 08          mov 0x8(%ebp),%edx  
14: 8b 45 0c          mov 0xc(%ebp),%eax  
17: 01 c2             add %eax,%edx  
19: 8b 45 10          mov 0x10(%ebp),%eax  
1c: 01 c2             add %eax,%edx  
1e: 8b 45 14          mov 0x14(%ebp),%eax  
21: 01 d0             add %edx,%eax  
23: 5d               pop %ebp  
24: c3               ret  
  
00000025 <main>:  
25: f3 0f 1e fb      endbr32  
29: 55               push %ebp  
2a: 89 e5             mov %esp,%ebp  
2c: e8 fc ff ff ff   call 2d <main+0x8>  
31: 05 01 00 00 00    add $0x1,%eax  
36: 6a 04             push $0x4  
38: 6a 03             push $0x3  
3a: 6a 02             push $0x2  
3c: 6a 01             push $0x1  
3e: e8 fc ff ff ff   call 3f <main+0x1a>  
43: 83 c4 10          add $0x10,%esp  
46: b8 00 00 00 00    mov $0x0,%eax  
4b: c9               leave  
4c: c3               ret
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



```
int main() { foo(1,2,3,4); }
```

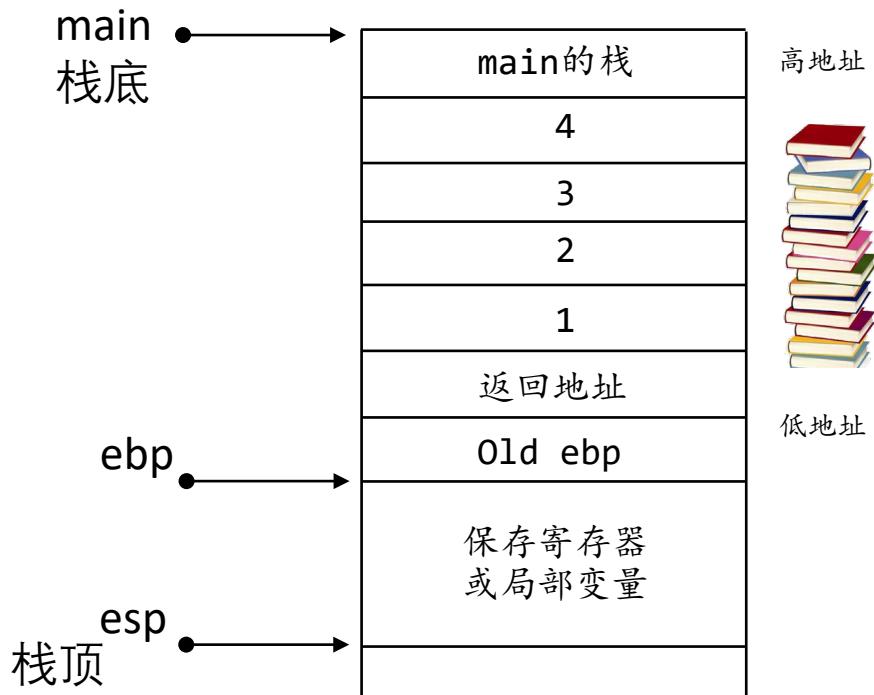
Disassembly of section .text:

```
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55               push %ebp  
 5: 89 e5             mov %esp,%ebp  
 7: e8 fc ff ff ff   call 8 <foo+0x8>  
 c: 05 01 00 00 00    add $0x1,%eax  
11: 8b 55 08          mov 0x8(%ebp),%edx  
14: 8b 45 0c          mov 0xc(%ebp),%eax  
17: 01 c2             add %eax,%edx  
19: 8b 45 10          mov 0x10(%ebp),%eax  
1c: 01 c2             add %eax,%edx  
1e: 8b 45 14          mov 0x14(%ebp),%eax  
21: 01 d0             add %edx,%eax  
23: 5d               pop %ebp  
24: c3               ret  
  
00000025 <main>:  
25: f3 0f 1e fb      endbr32  
29: 55               push %ebp  
2a: 89 e5             mov %esp,%ebp  
2c: e8 fc ff ff ff   call 2d <main+0x8>  
31: 05 01 00 00 00    add $0x1,%eax  
36: 6a 04             push $0x4  
38: 6a 03             push $0x3  
3a: 6a 02             push $0x2  
3c: 6a 01             push $0x1  
3e: e8 fc ff ff ff   call 3f <main+0x1a>  
43: 83 c4 10          add $0x10,%esp  
46: b8 00 00 00 00    mov $0x0,%eax  
4b: c9               leave  
4c: c3               ret
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



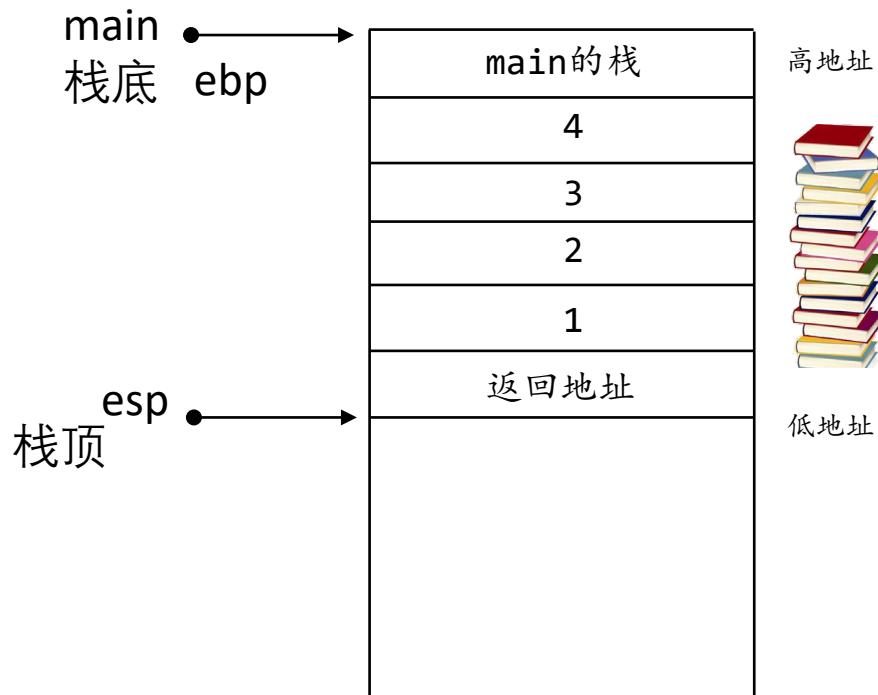
```
int main() { foo(1,2,3,4); }
```

```
Disassembly of section .text:  
  
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55                push %ebp  
 5: 89 e5              mov %esp,%ebp  
 7: e8 fc ff ff ff    call 8 <foo+0x8>  
 c: 05 01 00 00 00     add $0x1,%eax  
11: 8b 55 08          mov 0x8(%ebp),%edx  
14: 8b 45 0c          mov 0xc(%ebp),%eax  
17: 01 c2              add %eax,%edx  
19: 8b 45 10          mov 0x10(%ebp),%eax  
1c: 01 c2              add %eax,%edx  
1e: 8b 45 14          mov 0x14(%ebp),%eax  
21: 01 d0              add %edx,%eax  
23: 5d                pop %ebp  
24: c3                ret  
  
00000025 <main>:  
25: f3 0f 1e fb      endbr32  
29: 55                push %ebp  
2a: 89 e5              mov %esp,%ebp  
2c: e8 fc ff ff ff    call 2d <main+0x8>  
31: 05 01 00 00 00     add $0x1,%eax  
36: 6a 04              push $0x4  
38: 6a 03              push $0x3  
3a: 6a 02              push $0x2  
3c: 6a 01              push $0x1  
3e: e8 fc ff ff ff    call 3f <main+0x1a>  
43: 83 c4 10          add $0x10,%esp  
46: b8 00 00 00 00     mov $0x0,%eax  
4b: c9                leave  
4c: c3                ret
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



```
int main() { foo(1,2,3,4); }
```

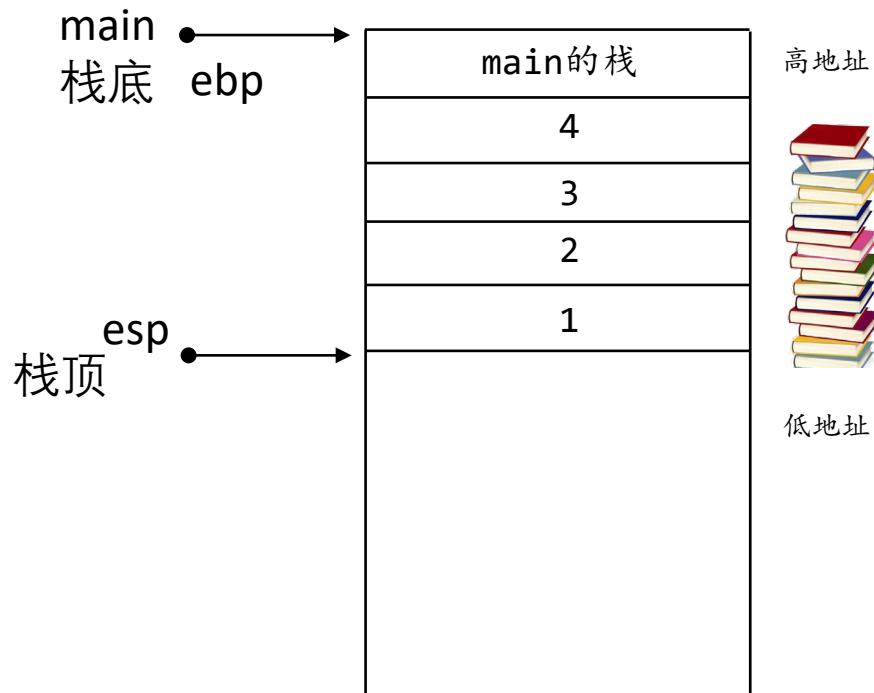
Disassembly of section .text:

```
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55                push %ebp  
 5: 89 e5              mov %esp,%ebp  
 7: e8 fc ff ff ff    call 8 <foo+0x8>  
 c: 05 01 00 00 00     add $0x1,%eax  
 11: 8b 55 08          mov 0x8(%ebp),%edx  
 14: 8b 45 0c          mov 0xc(%ebp),%eax  
 17: 01 c2              add %eax,%edx  
 19: 8b 45 10          mov 0x10(%ebp),%eax  
 1c: 01 c2              add %eax,%edx  
 1e: 8b 45 14          mov 0x14(%ebp),%eax  
 21: 01 d0              add %edx,%eax  
 23: 5d                pop %ebp  
 24: c3                ret  
  
00000025 <main>:  
 25: f3 0f 1e fb      endbr32  
 29: 55                push %ebp  
 2a: 89 e5              mov %esp,%ebp  
 2c: e8 fc ff ff ff    call 2d <main+0x8>  
 31: 05 01 00 00 00     add $0x1,%eax  
 36: 6a 04              push $0x4  
 38: 6a 03              push $0x3  
 3a: 6a 02              push $0x2  
 3c: 6a 01              push $0x1  
 3e: e8 fc ff ff ff    call 3f <main+0x1a>  
 43: 83 c4 10          add $0x10,%esp  
 46: b8 00 00 00 00     mov $0x0,%eax  
 4b: c9                leave  
 4c: c3                ret
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



```
int main() { foo(1,2,3,4); }
```

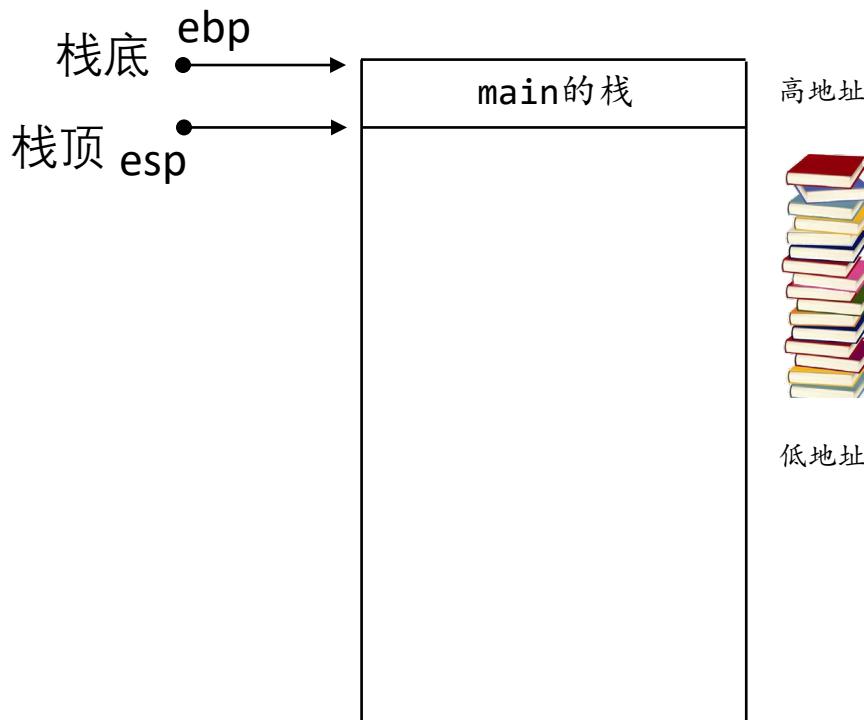
Disassembly of section .text:

```
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55                push %ebp  
 5: 89 e5              mov %esp,%ebp  
 7: e8 fc ff ff ff    call 8 <foo+0x8>  
 c: 05 01 00 00 00    add $0x1,%eax  
11: 8b 55 08          mov 0x8(%ebp),%edx  
14: 8b 45 0c          mov 0xc(%ebp),%eax  
17: 01 c2              add %eax,%edx  
19: 8b 45 10          mov 0x10(%ebp),%eax  
1c: 01 c2              add %eax,%edx  
1e: 8b 45 14          mov 0x14(%ebp),%eax  
21: 01 d0              add %edx,%eax  
23: 5d                pop %ebp  
24: c3                ret  
  
00000025 <main>:  
25: f3 0f 1e fb      endbr32  
29: 55                push %ebp  
2a: 89 e5              mov %esp,%ebp  
2c: e8 fc ff ff ff    call 2d <main+0x8>  
31: 05 01 00 00 00    add $0x1,%eax  
36: 6a 04              push $0x4  
38: 6a 03              push $0x3  
3a: 6a 02              push $0x2  
3c: 6a 01              push $0x1  
3e: e8 fc ff ff ff    call 3f <main+0x1a>  
43: 83 c4 10          add $0x10,%esp  
46: b8 00 00 00 00    mov $0x0,%eax  
4b: c9                leave  
4c: c3                ret
```

# cdecl函数调用惯例

```
int __cdecl foo(int m1, int m2, int m3, int m4);
```

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



```
int main() { foo(1,2,3,4); }
```

```
Disassembly of section .text:  
  
00000000 <foo>:  
 0: f3 0f 1e fb      endbr32  
 4: 55               push %ebp  
 5: 89 e5             mov %esp,%ebp  
 7: e8 fc ff ff ff   call 8 <foo+0x8>  
 c: 05 01 00 00 00    add $0x1,%eax  
11: 8b 55 08          mov 0x8(%ebp),%edx  
14: 8b 45 0c          mov 0xc(%ebp),%eax  
17: 01 c2             add %eax,%edx  
19: 8b 45 10          mov 0x10(%ebp),%eax  
1c: 01 c2             add %eax,%edx  
1e: 8b 45 14          mov 0x14(%ebp),%eax  
21: 01 d0             add %edx,%eax  
23: 5d               pop %ebp  
24: c3               ret  
  
00000025 <main>:  
25: f3 0f 1e fb      endbr32  
29: 55               push %ebp  
2a: 89 e5             mov %esp,%ebp  
2c: e8 fc ff ff ff   call 2d <main+0x8>  
31: 05 01 00 00 00    add $0x1,%eax  
36: 6a 04             push $0x4  
38: 6a 03             push $0x3  
3a: 6a 02             push $0x2  
3c: 6a 01             push $0x1  
3e: e8 fc ff ff ff   call 3f <main+0x1a>  
43: 83 c4 10          add $0x10,%esp  
46: b8 00 00 00 00    mov $0x0,%eax  
4b: c9               leave  
4c: c3               ret
```

参数传递	出栈方	名字修饰
从右往左顺序压参数入栈	函数调用方	函数名前加下划线

# 调用惯例

```
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```

	参数传递	出栈方	名字修饰
cdecl	从右往左顺序压参数入栈	函数调用方	下划线+函数名
stdcall	从右往左顺序压参数入栈	函数本身	下划线+函数名+@+参数字节数 <code>_foo@16</code>
fastcall	头两个 <b>DWORD</b> 类型或更少字节参数放入寄存器，其他参数从右往左顺序入栈	函数本身	@+函数名+@+参数字节数
pascal	从左往右顺序压参数入栈	函数本身	较复杂，见pascal文档

# 阅读汇编代码：“符号执行”

- 试着把内存/寄存器用数学符号表示出来
  - 然后假想地“单步执行”程序，用符号公式
  - James C. King. Symbolic execution and program verification. ACM, 19(7), 1976.

```
$ gcc -c -O2 -m32 foo.c
$ objdump -d foo.o

foo.o:      file format elf32-i386

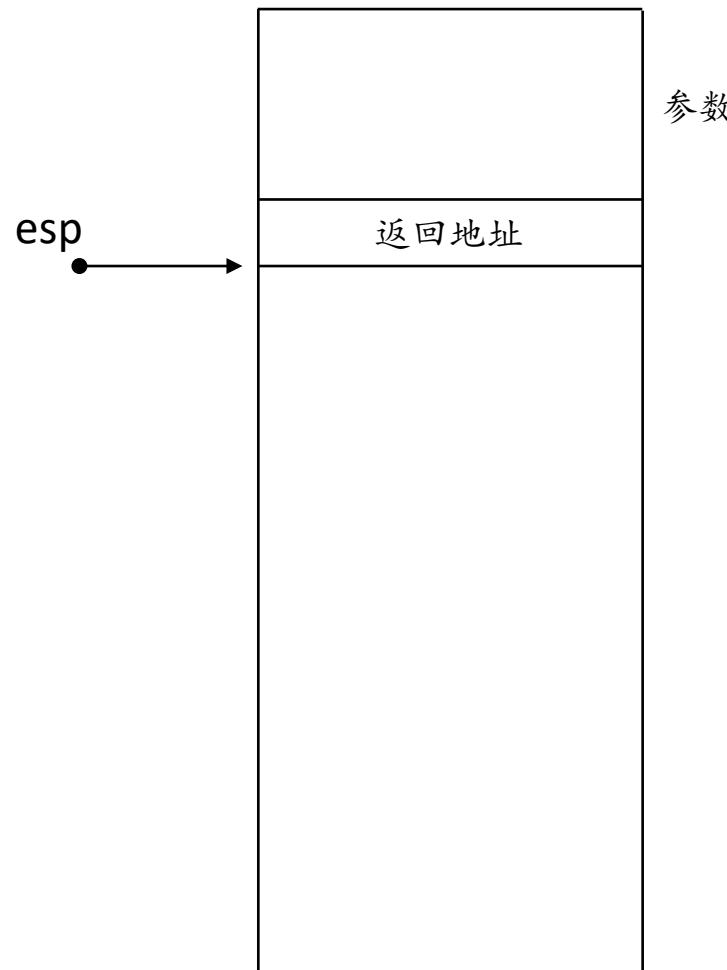
Disassembly of section .text:
00000000 <foo>:
 0:  f3 0f 1e fb          endbr32
 4:  8b 44 24 08          mov    0x8(%esp),%eax
 8:  03 44 24 04          add    0x4(%esp),%eax
 c:  03 44 24 0c          add    0xc(%esp),%eax
10:  03 44 24 10          add    0x10(%esp),%eax
14:  c3                   ret

Disassembly of section .text.startup:
00000000 <main>:
 0:  f3 0f 1e fb          endbr32
 4:  31 c0                xor    %eax,%eax
 6:  c3                   ret
```

- 编译选项：-m32 -O2 -fno-pic(便于大家理解)

```
000004f0 <foo>:
4f0: 83 ec 18          sub   $0x18,%esp
4f3: 8d 44 24 1c        lea    0x1c(%esp),%eax
4f7: 50                 push   %eax
4f8: e8 13 00 00 00      call   510 <bar>
4fd: 8b 44 24 20        mov    0x20(%esp),%eax
501: 83 c4 1c          add    $0x1c,%esp
504: c3                 ret
```

# 假裝單步調試



参数

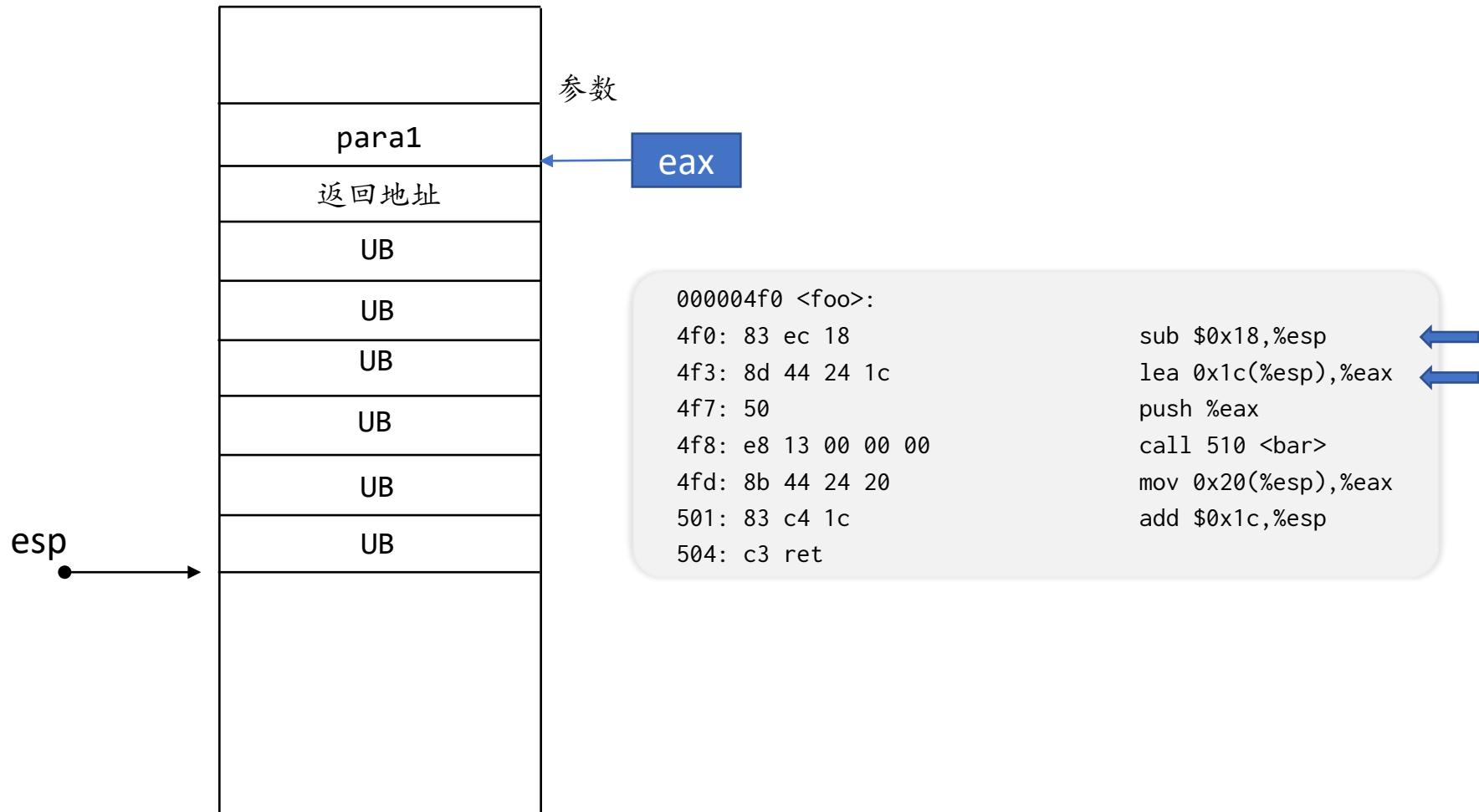
返回地址

```
000004f0 <foo>:  
4f0: 83 ec 18  
4f3: 8d 44 24 1c  
4f7: 50  
4f8: e8 13 00 00 00  
4fd: 8b 44 24 20  
501: 83 c4 1c  
504: c3 ret
```

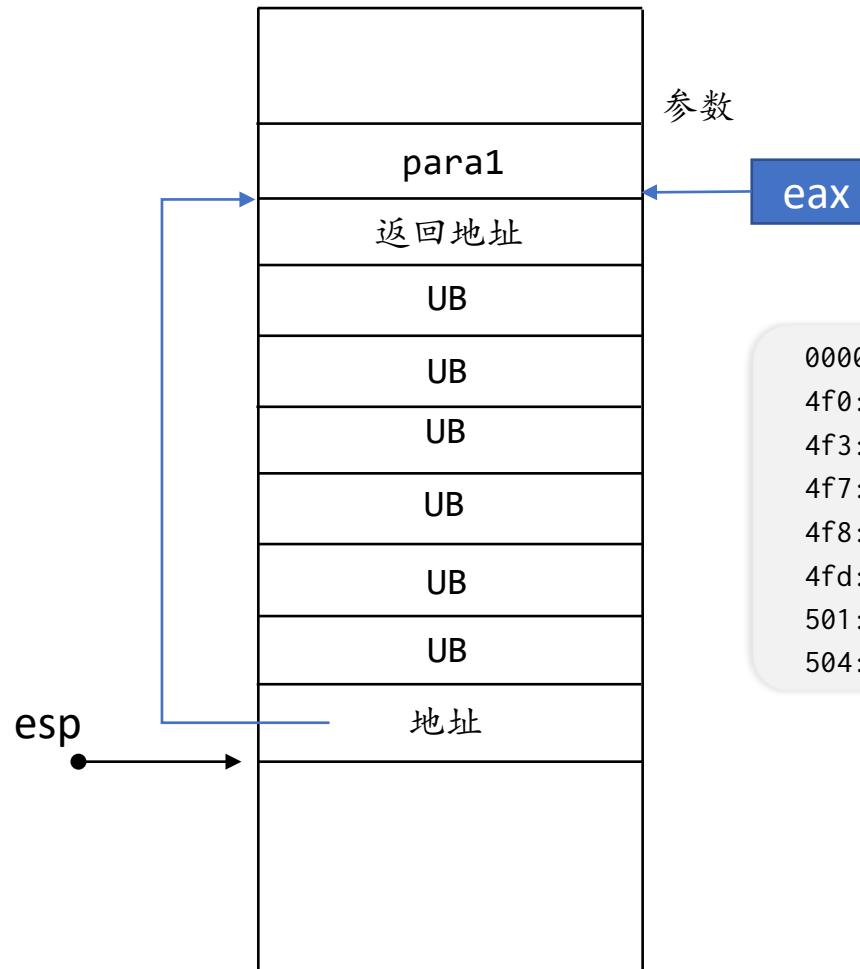
```
sub $0x18,%esp  
lea 0x1c(%esp),%eax  
push %eax  
call 510 <bar>  
mov 0x20(%esp),%eax  
add $0x1c,%esp
```



# 假裝單步調試



# 假裝單步調試

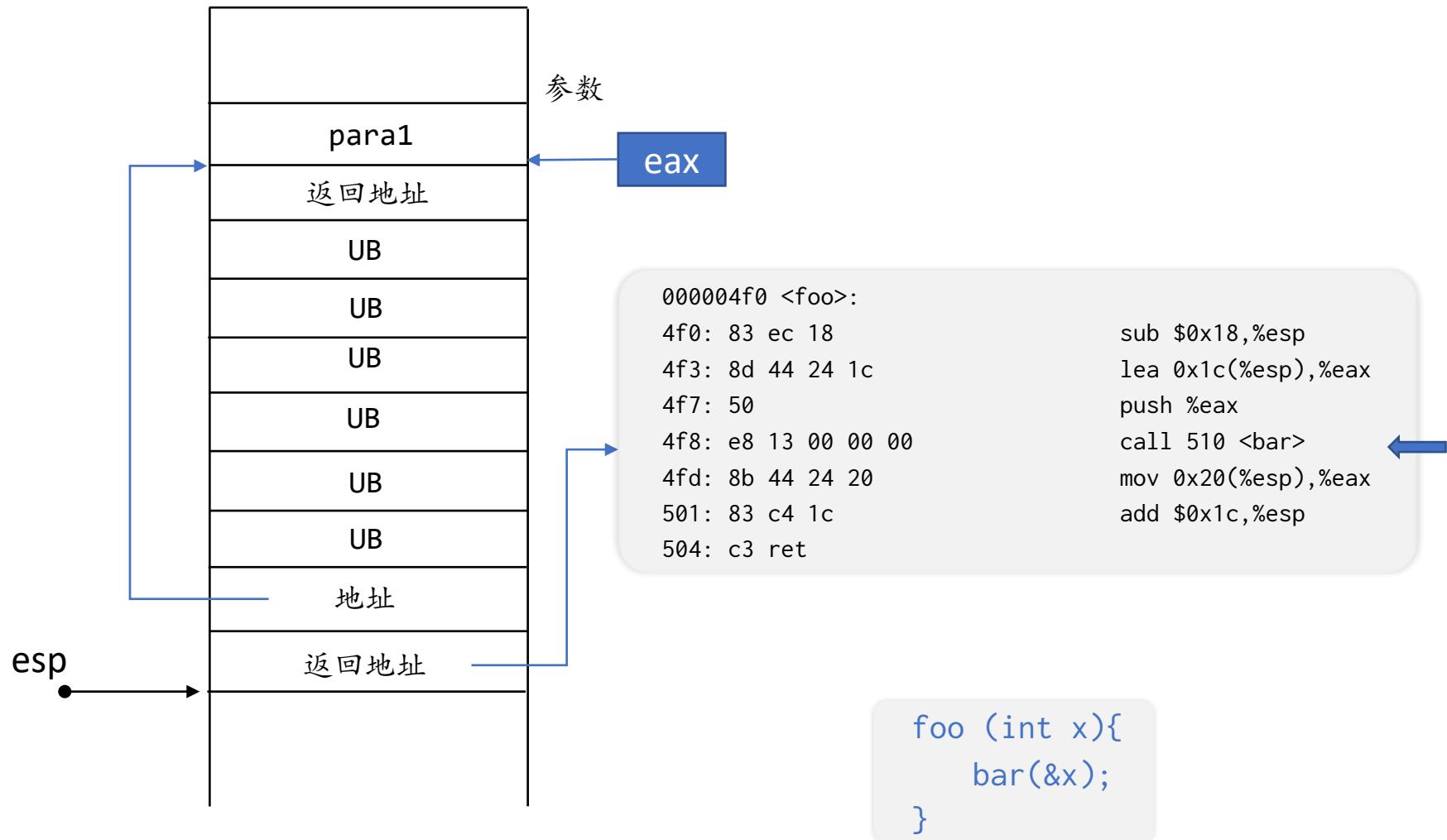


```
000004f0 <foo>:  
4f0: 83 ec 18  
4f3: 8d 44 24 1c  
4f7: 50  
4f8: e8 13 00 00 00  
4fd: 8b 44 24 20  
501: 83 c4 1c  
504: c3 ret
```

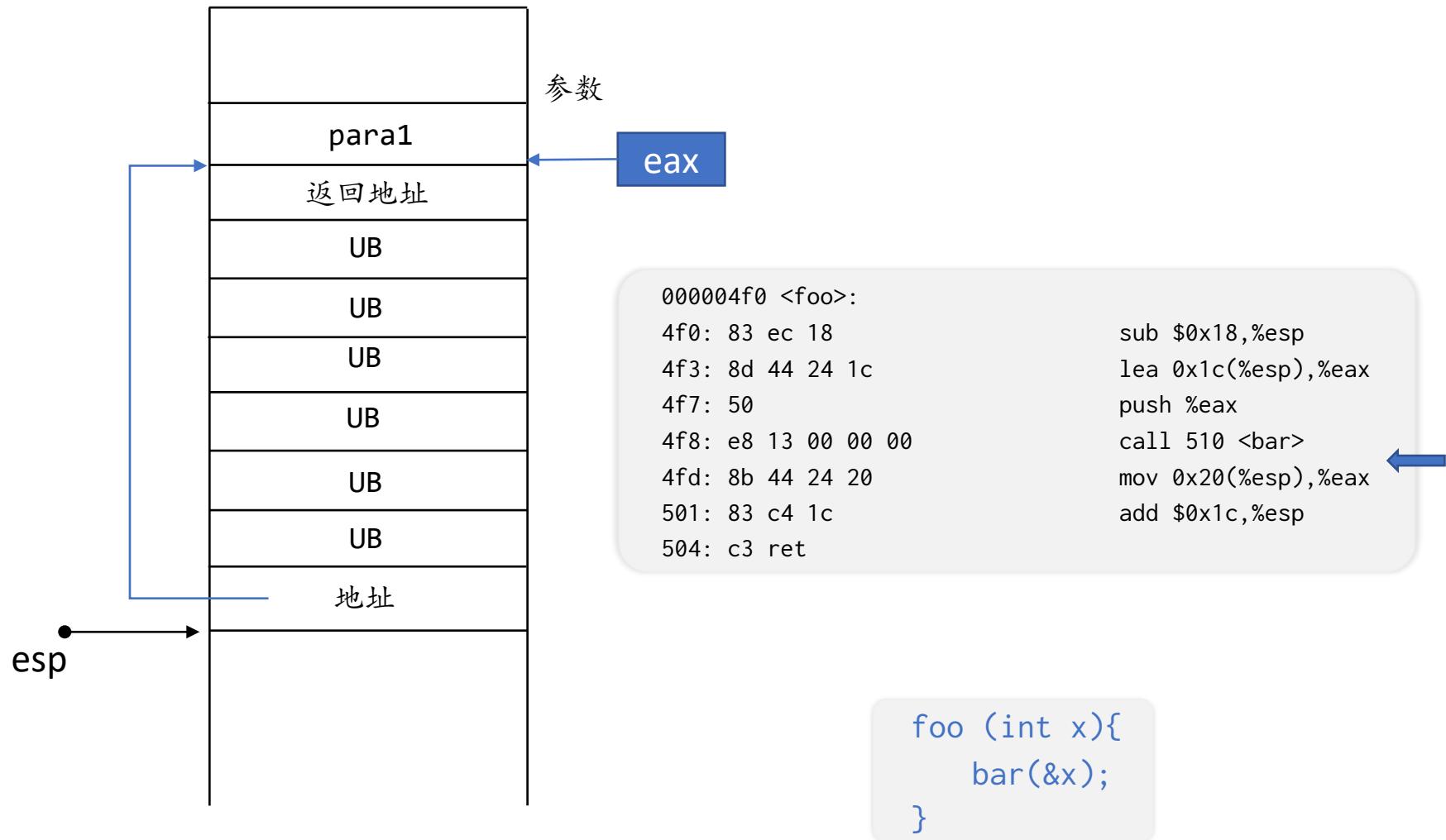
```
sub $0x18,%esp  
lea 0x1c(%esp),%eax  
push %eax  
call 510 <bar>  
mov 0x20(%esp),%eax  
add $0x1c,%esp
```



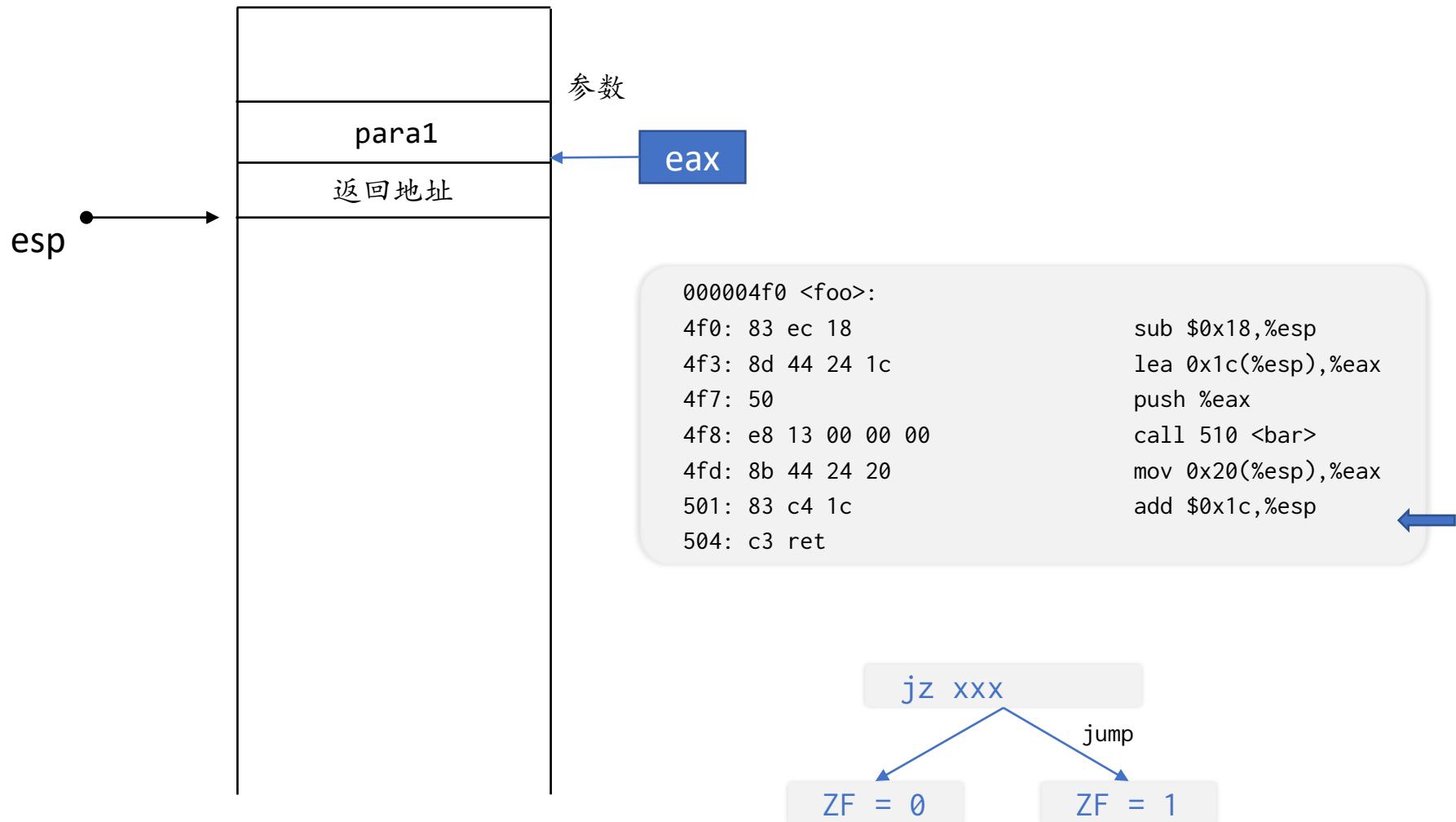
# 假裝單步調試



# 假裝單步調試



# 假裝單步調試



# 阅读汇编代码：“符号执行”

- 试着把内存/寄存器用数学符号表示出来
  - 然后假想地“单步执行”程序，用符号公式
  - James C. King. Symbolic execution and program verification. ACM, 19(7), 1976.

```
$ gcc -c -O2 -m32 foo.c
$ objdump -d foo.o

foo.o:      file format elf32-i386

Disassembly of section .text:
00000000 <foo>:
 0:  f3 0f 1e fb          endbr32
 4:  8b 44 24 08          mov    0x8(%esp),%eax
 8:  03 44 24 04          add    0x4(%esp),%eax
 c:  03 44 24 0c          add    0xc(%esp),%eax
10:  03 44 24 10          add    0x10(%esp),%eax
14:  c3                   ret

Disassembly of section .text.startup:
00000000 <main>:
 0:  f3 0f 1e fb          endbr32
 4:  31 c0                xor    %eax,%eax
 6:  c3                   ret
```

- 编译选项：-m32 -O2 -fno-pic(便于大家理解)

```
000004f0 <foo>:
4f0: 83 ec 18             sub   $0x18,%esp
4f3: 8d 44 24 1c          lea    0x1c(%esp),%eax
4f7: 50                   push   %eax
4f8: e8 13 00 00 00        call   510 <bar>
4fd: 8b 44 24 20          mov    0x20(%esp),%eax
501: 83 c4 1c             add    $0x1c,%esp
504: c3                   ret
```

# x86-64：寄存器与函数调用

# 寄存器 (1): 继承自 IA32

用途	64b	低32b	低16b	低8b	8-15b
返回值	%rax	%eax	%ax	%al	%ah
调用者保存	%rbx	%ebx	%bx	%bl	%bh
参数4	%rcx	%ecx	%cx	%cl	%ch
参数3	%rdx	%edx	%dx	%dl	%dh
参数2	%rsi	%esi	%si	%sil	
参数1	%rdi	%edi	%di	%dil	
调用者保存	%rbp	%ebp	%bp	%bpl	
栈顶	%rsp	%esp	%sp	%spl	

# 寄存器(2): 新增加的寄存器

- x86-64 扩充了很多寄存器!
  - 于是再也不用像 IA32 那样, 用堆栈传递参数了! !

用途	64b	低32b	低16b	低8b	8-15b
参数5	%r8	%r8d	%r8w	%r8b	
参数6	%r9	%r9d	%r9w	%r9b	
调用者保存	%r10	%r10d	%r10w	%r10b	
链接	%r11	%r11d	%r11w	%r11b	
C unsued	%r12	%r12d	%r12w	%r12b	
调用者保存	%r13	%r13d	%r13w	%r13b	
调用者保存	%r14	%r14d	%r14w	%r14b	
调用者保存	%r15	%r15d	%r15w	%r15b	(没有)

# A + B in x86-64

```
int f(int a, int b) {  
    return a + b;  
}
```

```
00000510 <add_32>:  
510: 8b 44 24 08      mov 0x8(%esp),%eax  
514: 03 44 24 04      add 0x4(%esp),%eax  
518: c3                ret
```

```
000000000000630 <add_64>:  
630: 8d 04 37          lea (%rdi,%rsi,1),%eax  
633: c3                retq
```

# max in x86-64

```
int max(int a, int b) {  
    if (a > b) return a;  
    else return b;  
}
```

00000514 <max\_32>:

514: 8b 44 24 04	mov 0x4(%esp),%eax
518: 3b 44 24 08	cmp 0x8(%esp),%eax
51c: 7d 04	jg 522 <max+0xe>
51e: 8b 44 24 08	mov 0x8(%esp),%eax
522: c3	ret

movl 4(%esp), %eax
movl 8(%esp), %edx
cmpl %edx, %eax
cmove %edx, %eax
ret

000000000000640 <max\_64>:

640: 39 f7	cmp %esi,%edi
642: 89 f0	mov %esi,%eax
644: 0f 4d c7	cmove %edi,%eax
647: c3	retq

# 使用寄存器传递函数参数：优势

- 支持 6 个参数的传递： rdi, rsi, rdx, rcx, r8, r9
  - callee 可以随意修改这 6 个寄存器的值
  - 编译器有了更大的调度空间 (大幅减少堆栈内存访问)
- 例子：

```
void plus(int a, int b) {  
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);  
}
```

```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- `calling convention: rdi, rsi, rdx, rcx, r8, r9`

000000000000700 <plus>:

700: 44 8d 0c 37	lea (%rdi,%rsi,1),%r9d		
704: 89 f9	mov %edi,%ecx	201010	←
706: 48 8b 3d 03 09 20 00	mov 0x200903(%rip),%rdi # XXXXXX <stdout@@GLIBC_2.2.5>		
70d: 48 8d 15 b0 00 00 00	lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>		
714: 41 89 f0	mov %esi,%r8d		
717: 31 c0	xor %eax,%eax		
719: be 01 00 00 00	mov \$0x1,%esi		
71e: e9 5d fe ff ff	jmpq 580 <_fprintf_chk@plt>		

rdi	rsi	r9	rcx
a	b	a+b	a

```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

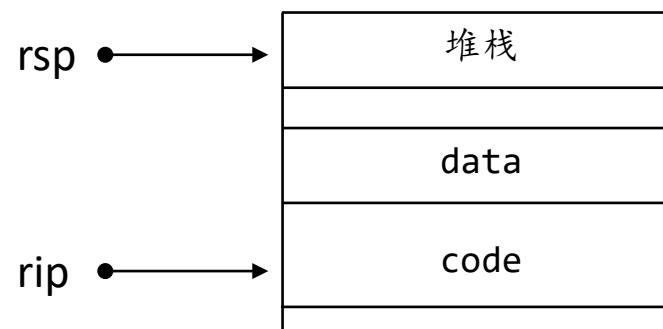
```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>	←
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>	
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>	
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>	
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>	
<code>717: 31 c0</code>	<code>xor %eax,%eax</code>	
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code>	
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>	



```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

```

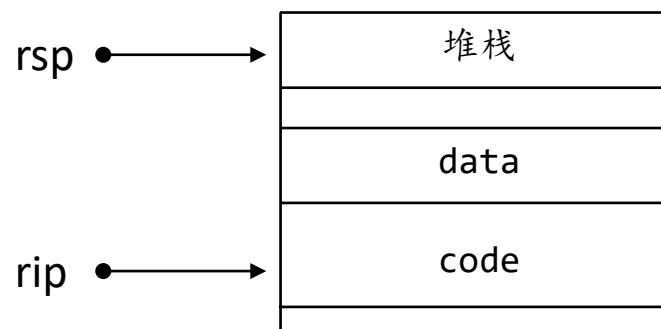
- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>
<code>717: 31 c0</code>	<code>xor %eax,%eax</code>
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code>
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>

<code>rdi</code>	<code>rsi</code>	<code>r9</code>	<code>rcx</code>
<code>stdout</code>	<code>b</code>	<code>a+b</code>	<code>a</code>



```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

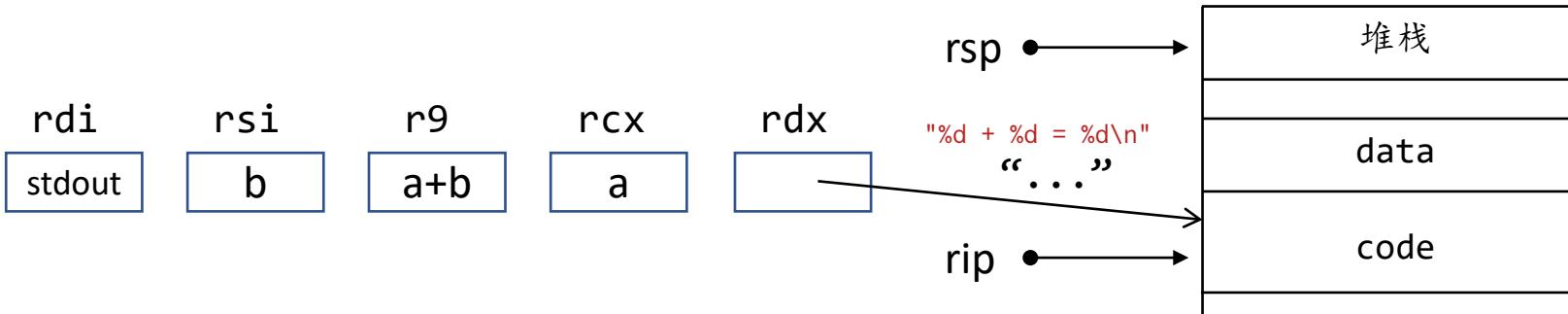
```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>
<code>717: 31 c0</code>	<code>xor %eax,%eax</code>
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code>
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>



```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

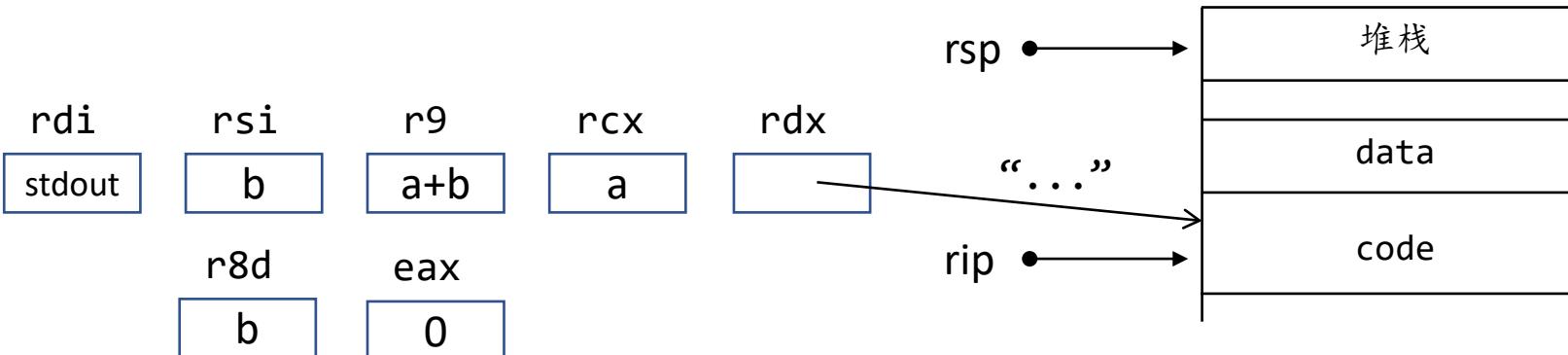
```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>
<code>717: 31 c0</code>	<code>xor %eax,%eax</code> ←
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code>
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>



```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

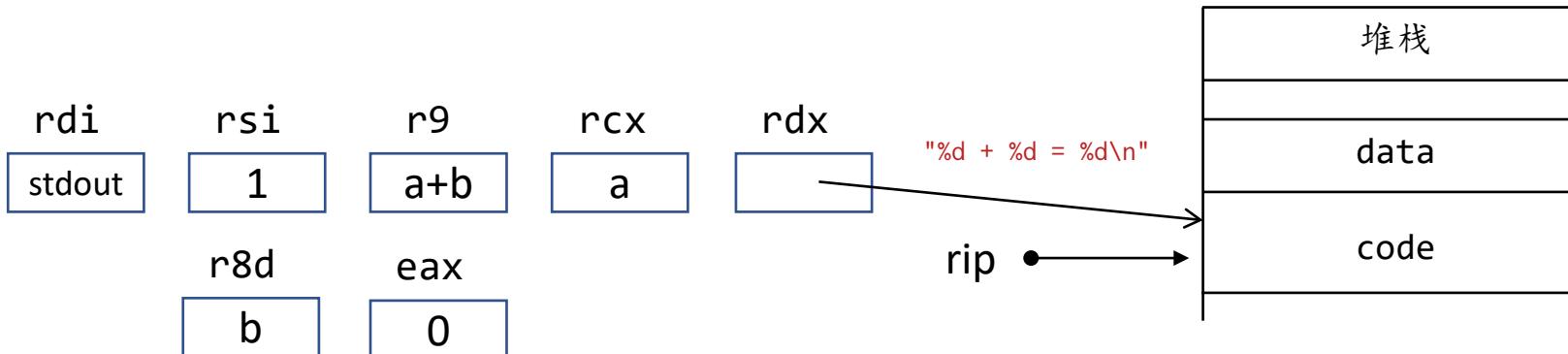
```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>
<code>717: 31 c0</code>	<code>xor %eax,%eax</code>
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code> ←
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>



# 一些更多的分析

- plus的最后一条指令

```
71e: e9 5d fe ff ff      jmpq 580 <__fprintf_chk@plt>
```

- 并不是调用的printf，而是调用的[有堆栈检查的版本](#)
  - 准备参数时有 mov \$0x1,%esi
- 直接jmp是因为函数末尾有一条ret指令
  - 借用了\_\_fprintf\_chk@plt的ret指令返回到plus的调用者
  - 如果有返回值，就会生成call指令；如果plus返回printf的结果，依然是jmp
  - 省的不止是一条指令
    - 连续的ret对分支预测是很大的挑战

# 对比32位printf

- 好读，但指令执行起来会稍慢一些

```
void plus(int a, int b) {  
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);  
}
```

```
000005b4 <plus>:  
5b4: 83 ec 14          sub    $0x14,%esp  
5b7: 8b 44 24 18       mov    0x18(%esp),%eax  
5bb: 8b 54 24 1c       mov    0x1c(%esp),%edx  
5bf: 8d 0c 10          lea    (%eax,%edx,1),%ecx  
5c2: 51                push   %ecx  
5c3: 52                push   %edx  
5c4: 50                push   %eax  
5c5: 68 60 06 00 00     push   $0x660  
5ca: 6a 01              push   $0x1  
5cc: ff 35 00 00 00 00  pushl  0x0  
5d2: e8 fc ff ff ff     call   5d3 <plus+0x1f>  
5d7: 83 c4 2c          add    $0x2c,%esp  
5da: c3                ret  
5db: 90                nop
```

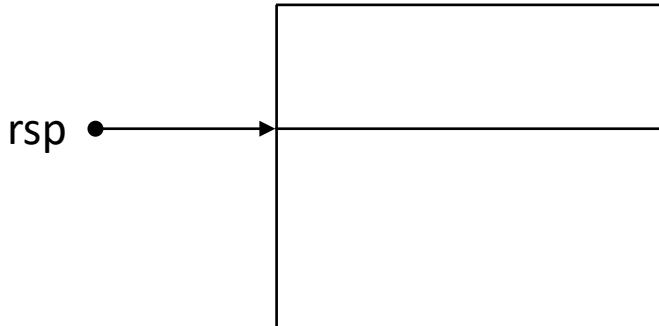
# 一个有趣的小问题

- x86-64 按寄存器传递参数
  - `void f(int x) {... &x ...}` 会发生什么？
- 编译器会给参数分配内存，保证后续访问合法
  - 给编译器带来了轻微的负担
  - 但编译器并不觉得这是负担

```
void bar(int *);  
int foo(int x) {  
    bar(&x);  
    return x;  
}
```

# 假裝單步調試

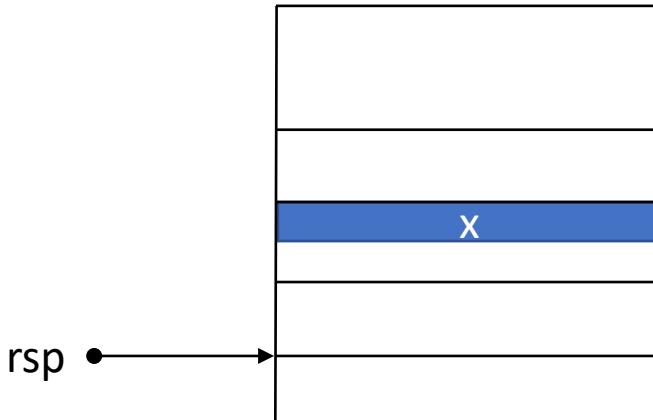
```
void bar(int *);  
int foo(int x) {  
    bar(&x);  
    return x;  
}
```



```
$ gcc -c -O2 foo.c  
$ objdump -d foo.o  
  
foo.o:      file format elf64-x86-64  
  
Disassembly of section .text:  
  
0000000000000000 <foo>:  
 0:   f3 0f 1e fa        endbr64  
 4:   48 83 ec 18        sub    $0x18,%rsp  
 8:   89 7c 24 0c        mov    %edi,0xc(%rsp)  
 c:   48 8d 7c 24 0c     lea    0xc(%rsp),%rdi  
11:   e8 00 00 00 00     call   16 <foo+0x16>  
16:   8b 44 24 0c        mov    0xc(%rsp),%eax  
1a:   48 83 c4 18        add    $0x18,%rsp  
1e:   c3                 ret
```

# 假裝單步調試

```
void bar(int *);  
int foo(int x) {  
    bar(&x);  
    return x;  
}
```



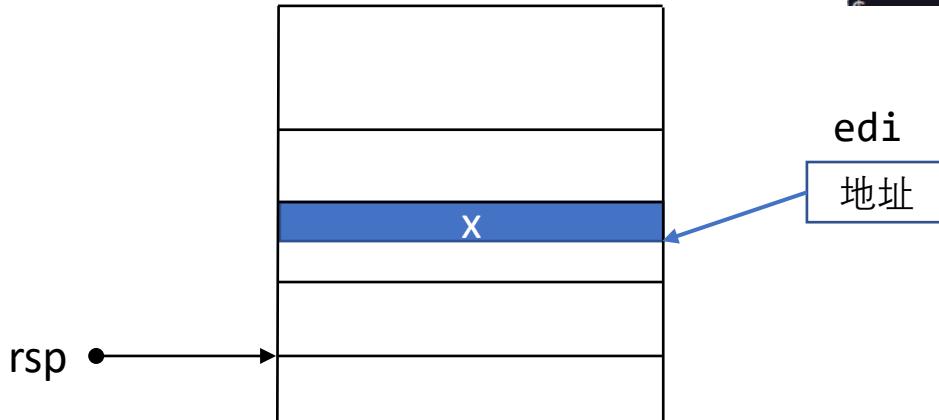
```
$ gcc -c -O2 foo.c  
$ objdump -d foo.o  
  
foo.o:      file format elf64-x86-64  
  
Disassembly of section .text:  
  
0000000000000000 <foo>:  
 0:   f3 0f 1e fa        endbr64  
 4:   48 83 ec 18        sub    $0x18,%rsp  
 8:   89 7c 24 0c        mov    %edi,0xc(%rsp)  
 c:   48 8d 7c 24 0c    lea    0xc(%rsp),%rdi  
11:   e8 00 00 00 00    call   16 <foo+0x16>  
16:   8b 44 24 0c        mov    0xc(%rsp),%eax  
1a:   48 83 c4 18        add    $0x18,%rsp  
1e:   c3                  ret
```

edi

x

# 假裝單步調試

```
void bar(int *);  
int foo(int x) {  
    bar(&x);  
    return x;  
}
```



```
$ gcc -c -O2 foo.c  
$ objdump -d foo.o  
  
foo.o:      file format elf64-x86-64  
  
Disassembly of section .text:  
  
0000000000000000 <foo>:  
 0:   f3 0f 1e fa        endbr64  
 4:   48 83 ec 18        sub    $0x18,%rsp  
 8:   89 7c 24 0c        mov    %edi,0xc(%rsp)  
 c:   48 8d 7c 24 0c     lea    0xc(%rsp),%rdi  
11:   e8 00 00 00 00     call   16 <foo+0x16>  
16:   8b 44 24 0c        mov    0xc(%rsp),%eax  
1a:   48 83 c4 18        add    $0x18,%rsp  
1e:   c3                 ret
```

# x86-64程序：更多的例子

# swap in x86-64

- 总的来说，x86-64是更现代的体系结构，更精简的指令序列
  - void swap(int \*x, int \*y); 交换两个指针指向的数字

```
mov    0x8(%esp),%edx  
mov    0xc(%esp),%eax  
mov    (%edx),%ecx  
mov    (%eax),%ebx  
mov    %ebx,(%edx)  
mov    %ecx,(%eax)  
pop    %ebx
```

```
mov    (%rdi),%eax  
mov    (%rsi),%edx  
mov    %edx,(%rdi)  
mov    %eax,(%rsi)
```

# 例子：循环

```
int fact(int n) {  
    int res = 1;  
    do { res *= n; n--; } while (n > 0);  
    return res;  
}
```

```
        mov    $0x1,%eax  
        nopl    (%rax)  
.L1:   imul   %edi,%eax  
        sub    $0x1,%edi  
        test   %edi,%edi  
        jg     .L1  
        repz  retq
```

- 两个诡异代码：
  - `nopl (%rax)` : 内存对齐 (padding)
  - `repz retq`: 防止连续分支指令

# 例子：递归

000000000000704 <fib>:

704: 55	push	%rbp
705: 53	push	%rbx
706: 89 fd	mov	%edi,%ebp
708: 31 db	xor	%ebx,%ebx
70a: 48 83 ec 08	sub	\$0x8,%rsp
70e: 83 fd 01	cmp	\$0x1,%ebp
711: 7e 0f	jle	722 <fib+0x1e>
713: 8d 7d ff	lea	-0x1(%rbp),%edi
716: 83 ed 02	sub	\$0x2,%ebp
719: e8 e6 ff ff ff	callq	704 <fib>
71e: 01 c3	add	%eax,%ebx
720: eb ec	jmp	70e <fib+0xa>
722: 8d 43 01	lea	0x1(%rbx),%eax
725: 5a	pop	%rdx
726: 5b	pop	%rbx
727: 5d	pop	%rbp
728: c3	retq	

# Inline Assembly

# 在 C 代码中嵌入汇编

- 编译器：把 C 代码 “原样翻译” 成汇编代码
  - 那我们是不是可以在语句之间插入汇编呢？
    - 可以！编译器就做个“复制粘贴”

```
int foo(int x, int y) {  
    x++; y++;  
    asm ("endbr64;"  
        ".byte 0xf3, 0x0f, 0x1e, 0xfa"  
    );  
    return x + y;  
}
```

```
$ objdump -d foo.o  
  
foo.o:      file format elf64-x86-64  
  
Disassembly of section .text:  
  
0000000000000000 <foo>:  
 0:  f3 0f 1e fa          endbr64  
 4:  55                  push   %rbp  
 5:  48 89 e5             mov    %rsp,%rbp  
 8:  89 7d fc             mov    %edi,-0x4(%rbp)  
 b:  89 75 f8             mov    %esi,-0x8(%rbp)  
 e:  83 45 fc 01          addl   $0x1,-0x4(%rbp)  
12: 83 45 f8 01          addl   $0x1,-0x8(%rbp)  
16:  f3 0f 1e fa          endbr64  
1a:  f3 0f 1e fa          endbr64  
1e:  8b 55 fc             mov    -0x4(%rbp),%edx  
21:  8b 45 f8             mov    -0x8(%rbp),%eax  
24:  01 d0                add    %edx,%eax  
26:  5d                  pop    %rbp  
27:  c3                  ret
```

# 在汇编中访问 C 世界的表达式

```
int foo(int x, int y) {  
    int z;  
    x++; y++;  
    asm (  
        "addl %1, %2; "  
        "movl %2, %0; "  
        : "=r"(z) // output  
        : "r"(x), "r"(y) // input  
    );  
    return z;  
}
```

gcc -O0

```
0000000000000000 <foo>:  
 0: f3 0f 1e fa      endbr64  
 4: 55                push   %rbp  
 5: 48 89 e5          mov    %rsp,%rbp  
 8: 89 7d ec          mov    %edi,-0x14(%rbp)  
 b: 89 75 e8          mov    %esi,-0x18(%rbp)  
 e: 83 45 ec 01       addl   $0x1,-0x14(%rbp)  
 12: 83 45 e8 01     addl   $0x1,-0x18(%rbp)  
 16: 8b 45 ec          mov    -0x14(%rbp),%eax  
 19: 8b 55 e8          mov    -0x18(%rbp),%edx  
 1c: 01 c2              add    %eax,%edx  
 1e: 89 d0              mov    %edx,%eax  
 20: 89 45 fc          mov    %eax,-0x4(%rbp)  
 23: 8b 45 fc          mov    -0x4(%rbp),%eax  
 26: 5d                pop    %rbp  
 27: c3                ret
```

# 在汇编中访问 C 世界的表达式

```
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return z;
}
```

gcc -O2

```
0000000000000000 <foo>:
0:   f3 0f 1e fa          endbr64
4:   83 c7 01          add    $0x1,%edi
7:   8d 46 01          lea    0x1(%rsi),%eax
a:   01 f8          add    %edi,%eax
c:   89 c0          mov    %eax,%eax
e:   c3          ret
```

# 在汇编中访问 C 世界的表达式

```
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return z;
}
```

gcc -O2

```
0000000000000000 <foo>:
0: f3 0f 1e fa          endbr64
4: 83 c7 01             add    $0x1,%edi
7: 83 c6 01             add    $0x1.%esi
a: 01 fe                add    %edi,%esi
c: 89 f0                mov    %esi.%eax
e: c3                   ret
```

# 在汇编中访问 C 世界的表达式

```
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return 0;
}
```

gcc -O2

```
0000000000000000 <foo>:
 0:   f3 0f 1e fa          endbr64
 4:   31 c0                xor    %eax,%eax
 6:   c3                  ret
```

# 与编译器交互

---

- 实际的编译器可能会
  - 将某个变量分配给某个寄存器
    - inline assembly 改写寄存器就会导致错误
    - → clobber list
  - 代码优化
    - 例如 assembly 的返回值没有用到，就可以删除
      - → volatile

# 在汇编中访问 C 世界的表达式

```
int foo(int x, int y) {
    int z;
    x++; y++;
    asm volatile(
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return 0;
}
```

gcc -O2

```
0000000000000000 <foo>:
 0: f3 0f 1e fa          endbr64
 4: 83 c7 01            add    $0x1,%edi
 7: 83 c6 01            add    $0x1,%esi
 a: 01 fe               add    %edi,%esi
 c: 89 f7               mov    %esi,%edi
 e: 31 c0               xor    %eax,%eax
 10: c3                 ret
```

# 总结

# 对汇编感到很痛苦？

---

- 两个建议
  - 原理：想一想 YEMU 和 NEMU
  - 实践：“模拟调试”
    - 指令不过是 CPU 执行的基本单元
- 同时，付出也是必要的
  - 但你一旦掌握了分析汇编代码的方法
    - x86-64 也不可怕
    - AArch64 也不可怕
    - 《计算机系统基础也不可怕》

End. (你找对手册了吗? )

x86-64 Machine-Level Programming

How to Use Inline Assembly Language in C Code

# 调试：理论与实践

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

代码出 bug 了？又浪费了一天？太真实了？

- 本讲内容
  - 调试理论
  - 调试理论的应用
    - 调试 “不是代码”
    - 调试 PA/Lab/任何代码
- 为什么 PA 都开始那么久了现在才讲?
  - 因为你们不吃苦头是理解不了方法学的

# 调试理论

# 开始调试之前

- 摆正心态 (编程哲♂学)

机器永远是对的

不管是 crash 了，图形显示不正常了，还是 HIT BAD TRAP 了，最后都是你自己背锅

未测代码永远是错的

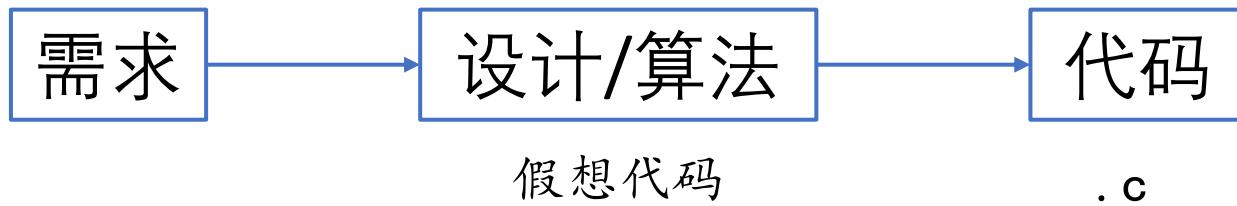
你以为最不可能出 bug 的地方，往往 bug 就在那躺着

# 调试理论

- 程序的两个功能
  - 人类世界需求的载体
    - 理解错需求 → bug
  - 计算过程的精确描述
    - 实现错误 → bug
- 调试 (debugging)
  - 已知程序有 bug, 如何找到?

Photo # NH 96566-KN (Color) First Computer "Bug", 1947



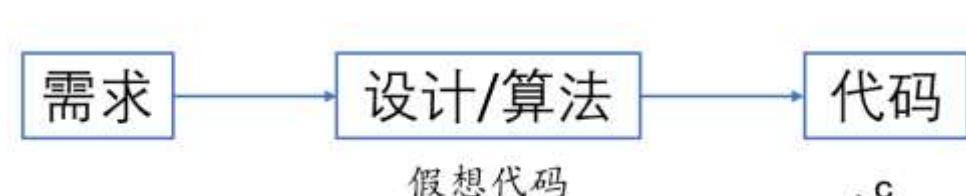


NEMU PA

# 为什么debug那么困难

- 因为 bug 的触发经历了漫长的过程
  - 需求 → 设计 → 代码 → Fault (bug) → Error (程序状态错) → Failure
    - 我们只能观测到 failure (可观测的结果错)
    - 我们可以检查状态的正确性 (但非常费时)
    - 无法预知 bug 在哪里 (每一行 “看起来” 都挺对的)

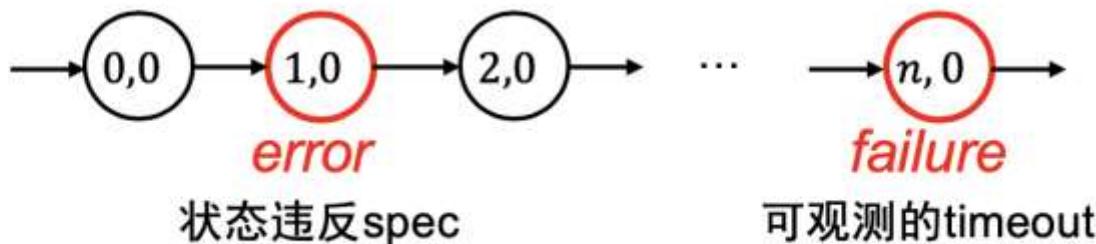
```
1 #include <stdio.h>
2
3 #define n 3
4
5 int main(){
6     int sum = 0;
7     for(int i = 0; i<n; i++){
8         for(int j = 0; j<n; i++){
9             sum += i * j;
10        }
11    printf("sum = %d\n", sum);
12 }
```



# 为什么debug那么困难

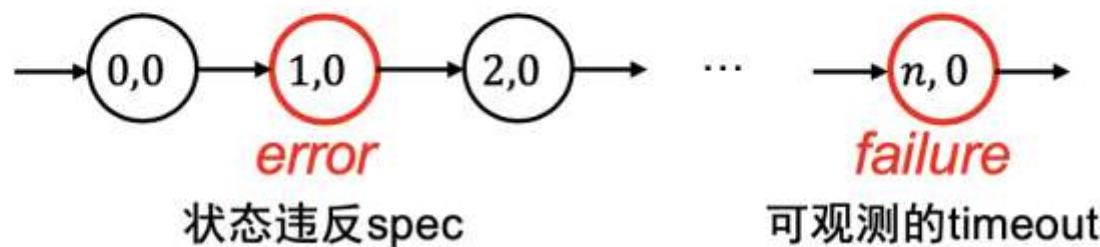
- 因为 bug 的触发经历了漫长的过程
  - 需求 → 设计 → 代码 → Fault (bug) → Error (程序状态错) → Failure
    - 我们只能观测到 failure (可观测的结果错)
    - 我们可以检查状态的正确性 (但非常费时)
    - 无法预知 bug 在哪里 (每一行 “看起来” 都挺对的)

```
for (int i = 0; i < n; i++) fault 程序bug
  for (int j = 0; j < n; i++)
    ...
    ...
```

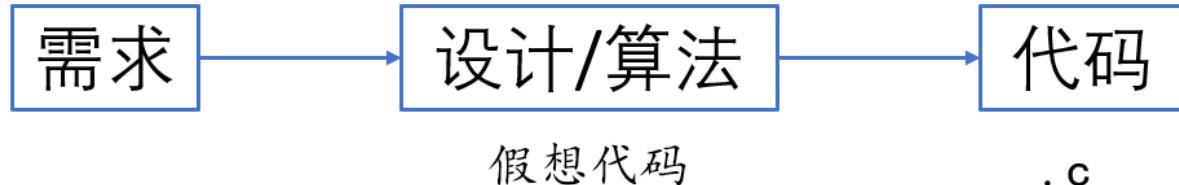
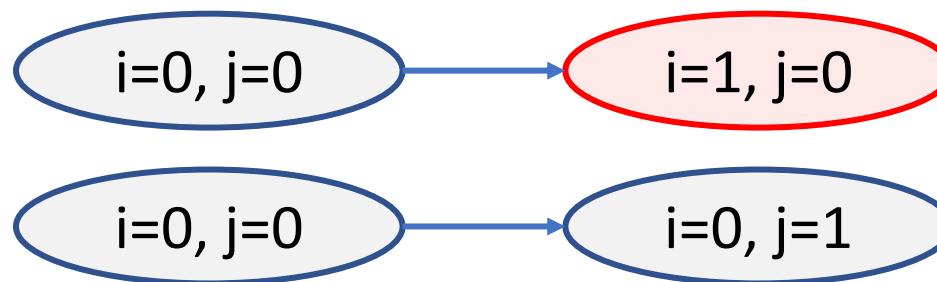


# 假想执行的困难

```
for (int i = 0; i < n; i++) fault 程序bug  
  for (int j = 0; j < n; i++)  
    ...
```

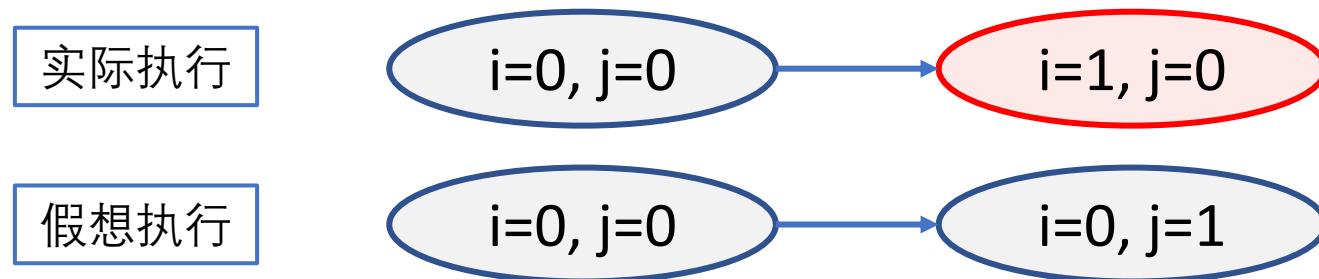


实际执行  
假想执行



# 调试理论

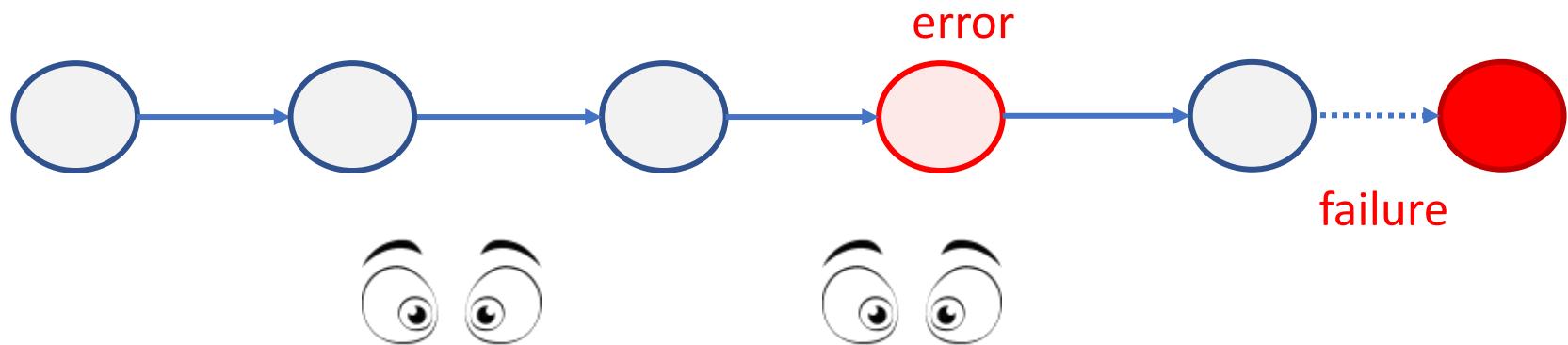
调试理论：如果我们能判定任意程序状态的正确性，那么给定一个 failure，我们可以通过二分查找定位到第一个 error 的状态，此时的代码就是 fault (bug)。



```
for (int i = 0; i < n; i++) fault 程序bug  
for (int j = 0; j < n; i++)
```

# Common practice

```
int main() {  
    sort();  
    //printf();  
    find_something();  
    //printf();  
    delete_something();  
    //printf();  
}
```

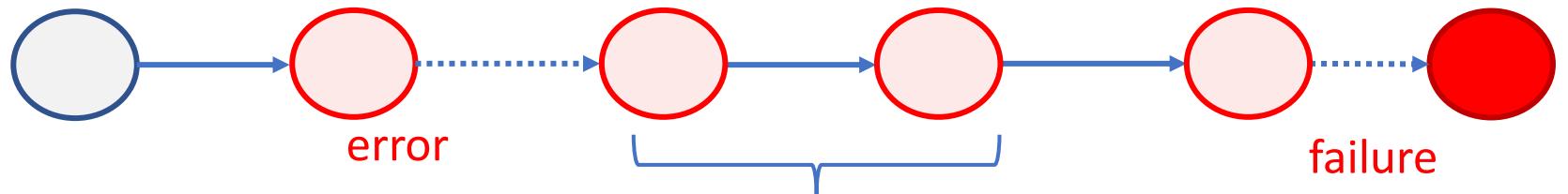
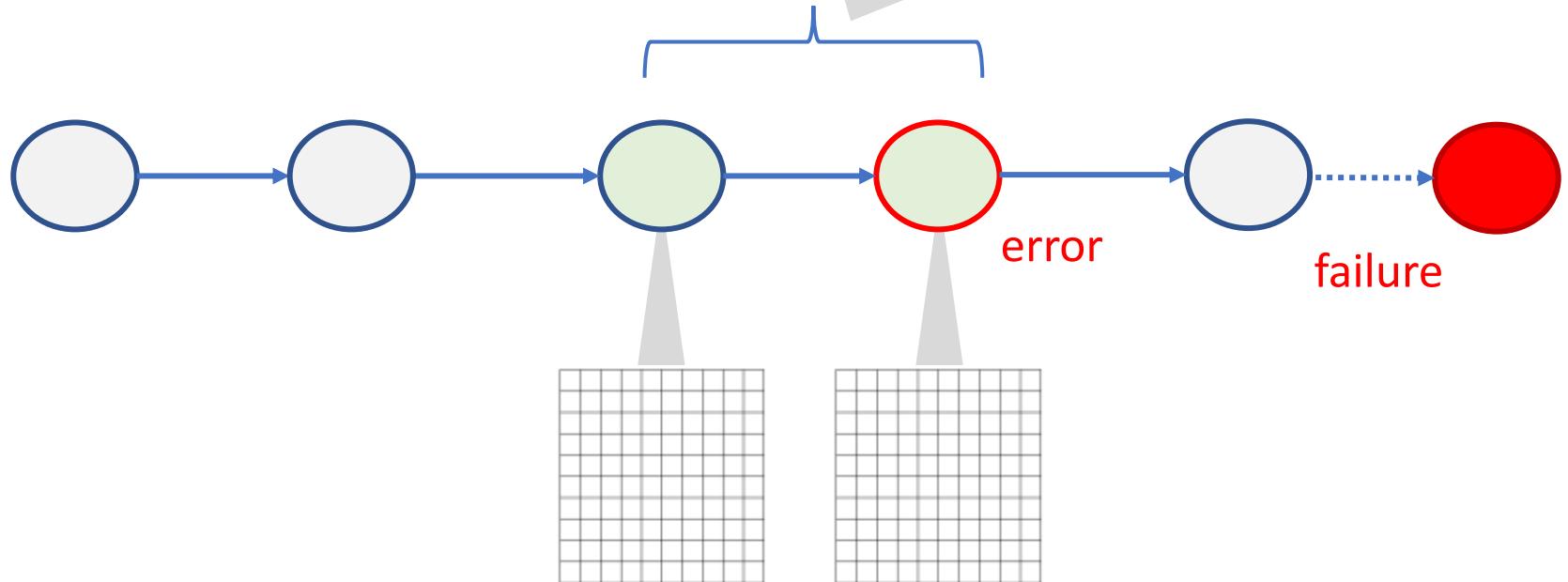


# 调试理论

调试理论：如果我们能判定任意程序状态的正确性，那么给定一个 failure，我们可以通过二分查找定位到第一个 error 的状态，此时的代码就是 fault (bug)。

- 调试理论：推论
  - 为什么我们喜欢“单步调试”？
    - 从一个假定正确的状态出发
    - 每个语句的行为有限，容易判定是否是 error
  - 为什么调试理论看起来没用？
    - 因为判定程序状态的正确性非常困难
      - (是否在调试 DP 题/图论算法时陷入时间黑洞？)

# Common practice: DP

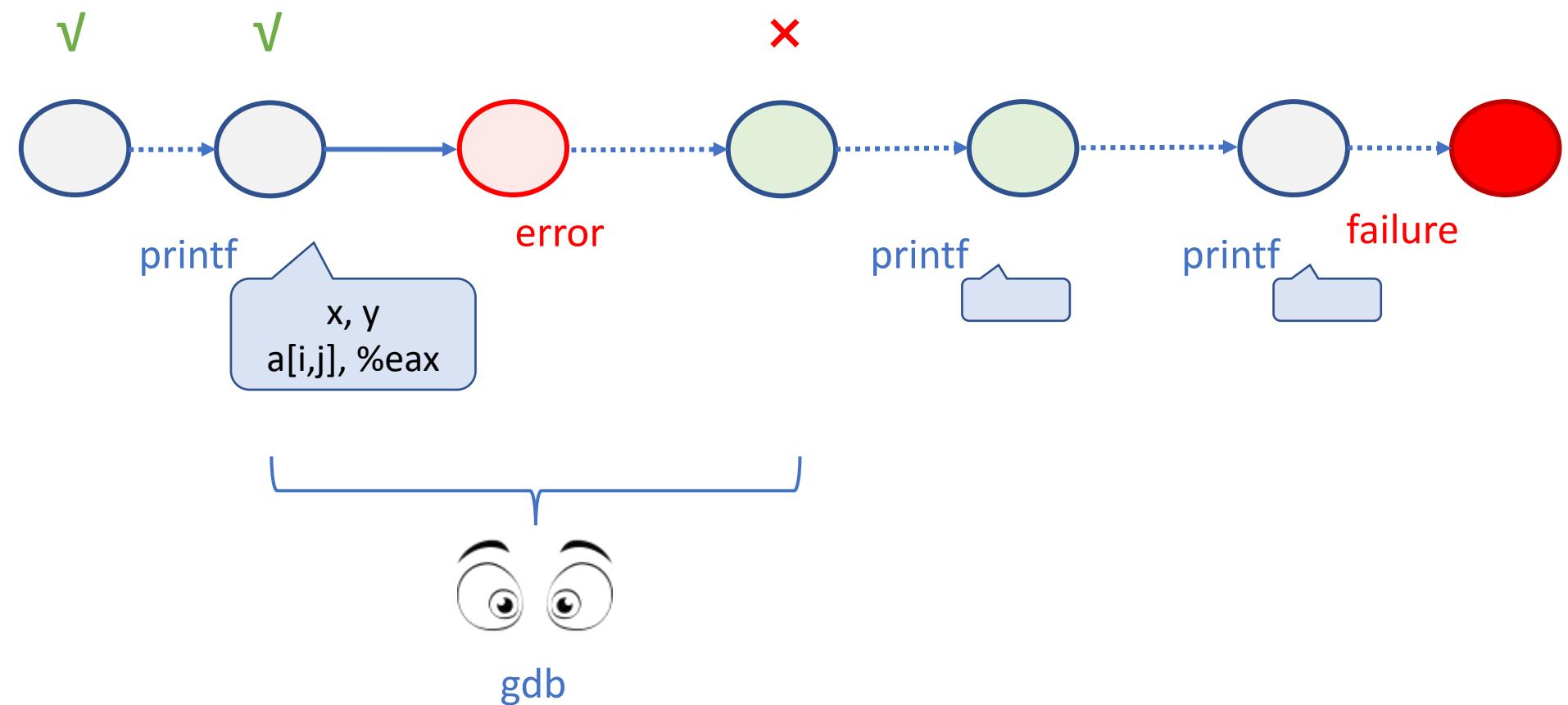


# 调试理论 (cont'd)

- 实际中的调试：通过**观察程序执行的轨迹** (trace)
  - 缩小错误状态 (error) 可能产生的位置
  - 作出适当的假设
  - 再进行细粒度的定位和诊断
- 最重要的两个工具
  - printf → 自定义log的trace
    - + 灵活可控、能快速定位问题大概位置、适用于大型软件
    - - 无法精确定位、大量的logs管理起来比较
  - gdb → 指令/语句级trace
    - + 精确、指令级定位、任意查看程序内部状态
    - - 耗费大量时间

# 调试理论 (cont'd)

- 尝试接近状态的判定



需求

设计/算法

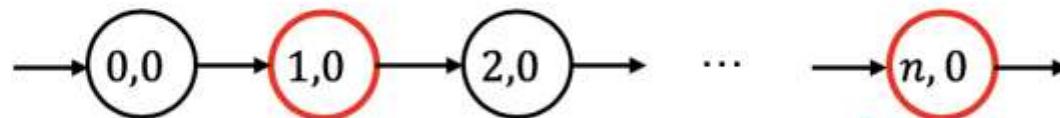
代码

假想代码

. c

```
for (int i = 0; i < n; i++) fault 程序bug  
  for (int j = 0; j < n; i++)
```

...



状态违反spec

可观测的timeout

printf → 自定义log的trace

+ 灵活可控、能快速定位问题大概位置、适用于大型软件

- 无法精确定位、大量的logs管理起来比较



gdb → 指令/语句级trace

+ 精确、指令级定位、任意查看程序内部状态

- 耗费大量时间

# 调试理论：应用（1）

调试（不是一般意义上“程序”的）bug

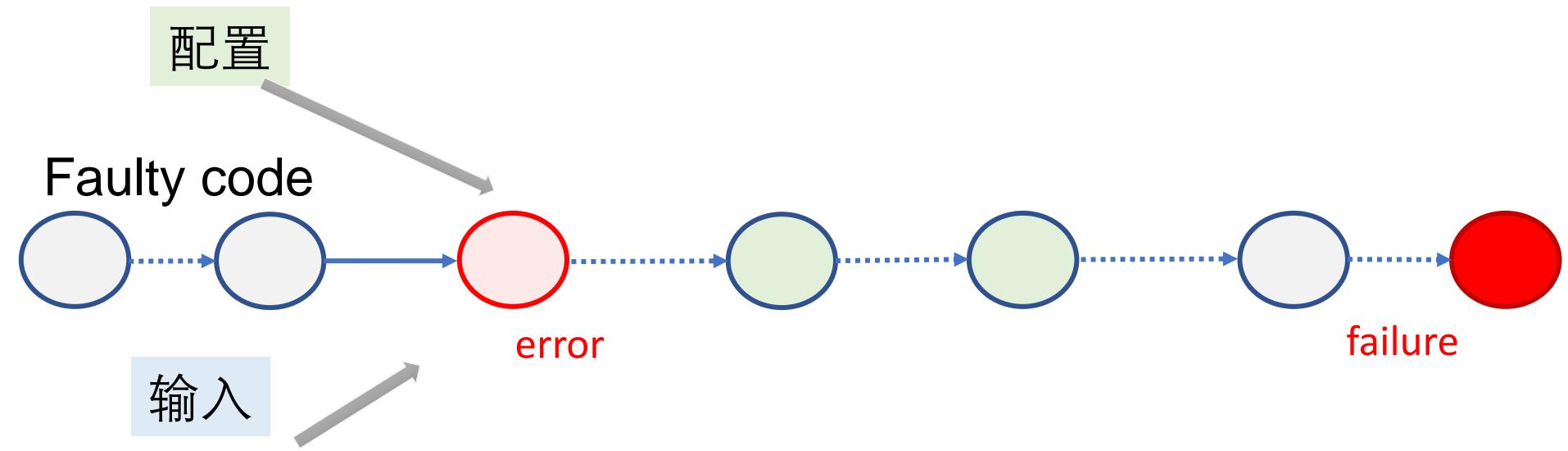
# PA体验极差

- 做实验会遇到大量与编程无关的问题
  - 也许没有想过：问题诊断其实也是调试
  - 我们应该利用调试理论去解决问题！

```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include
<sys/cdefs.h>
```

```
make[2]: *** run: No such file or directory. Stop.
Makefile:31: recipe for target 'run' failed
make[1]: *** [run] Error 2
...
```



管理 控制 视图 热键 设备 帮助

\$

S 英 拼 · 简 拼 \*



Right Shift + Right Alt

```
$ bash a.sh
<html>
<meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
</html>
$ bash a.sh
```

S 英 拼



# PA体验极差

- 做实验会遇到大量与编程无关的问题
  - 也许没有想过：问题诊断其实也是调试
  - 我们应该利用调试理论去解决问题！

```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include
<sys/cdefs.h>
```

```
make[2]: *** run: No such file or directory. Stop.
Makefile:31: recipe for target 'run' failed
make[1]: *** [run] Error 2
...
```

管理 控制 视图 热键 设备 帮助

```
$ ls  
a.c  a.out  a.sh  
$ █
```



Right Shift + Right Alt

# 例子：找不到sys/cdefs.h

- 'sys/cdefs.h': No such file or directory, 找不到文件
  - (这看起来是用 perror() 打印出来的哦! )
  - #include = 复制粘贴，自然会经过路径解析
  - (折腾20分钟) 明明/usr/include/x86\_64-linux-gnu/sys/cdefs.h 是存在的 (man 1 locate)
- 推理：#include<>一定有一些搜索路径
  - 为什么两个编译选项，一个通过，一个不通过？
  - gcc -m32 -v **V.S.** gcc -v
- 这是标准的解决问题办法：自己动手排查
  - 在面对复杂/小众问题时比 STFW 有效

# PA体验极差

- 做实验会遇到大量与编程无关的问题
  - 也许没有想过：问题诊断其实也是调试
  - 我们应该利用调试理论去解决问题！

```
bash: curl: command not found
```

```
fatal error: 'sys/cdefs.h': No such file or directory #include
<sys/cdefs.h>
```

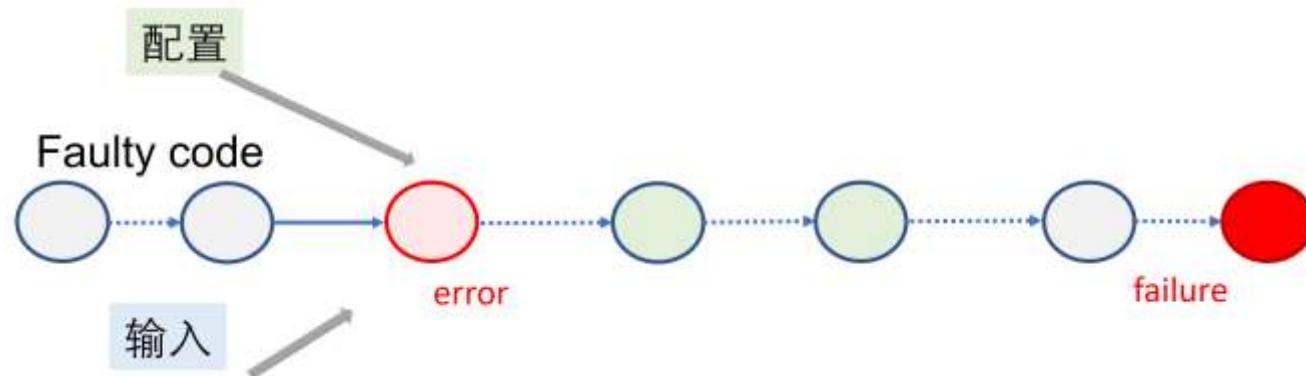
```
make[2]: *** run: No such file or directory. Stop.
Makefile:31: recipe for target 'run' failed
make[1]: *** [run] Error 2
...
```

# 使用调试理论 (cont'd)

- 正确的方法：理解程序的**执行过程**，弄清楚到底为何导致了bug
  - ssh：使用 -v 选项检查日志
  - gcc：使用 -v 选项打印各种过程
  - make：使用 -n 选项查看完整命令
    - make -nB | grep -ve '^\\(echo\\|mkdir\\)' 可以查看完整编译 nemu 的编译过程
  - 各个工具普遍提供调试功能，帮助用户/开发者了解程序的**行为**

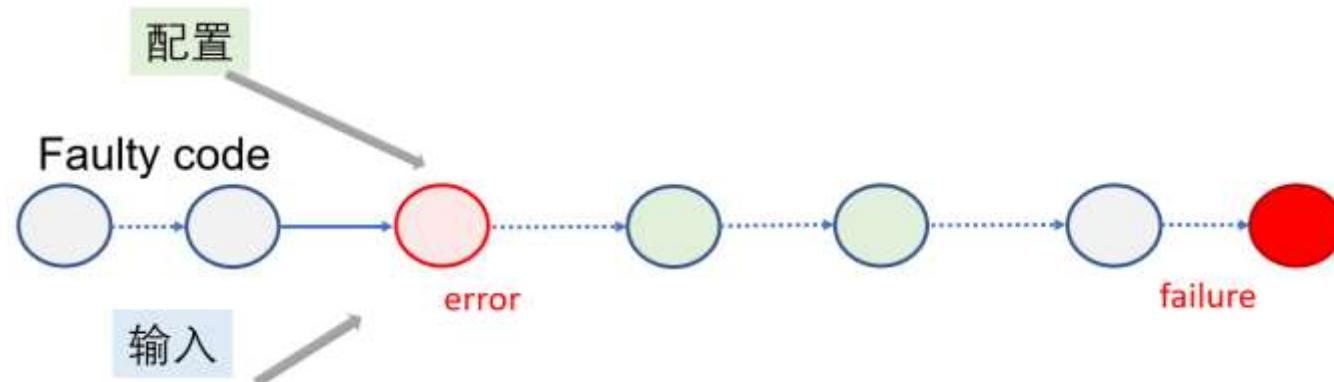
# 调试（不是程序的）Bug

- UNIX世界里你做任何事情都是在编程
  - 因此配置错、make 错等，都是程序或输入/配置有 bug
    - (输入/配置可以看成是程序的一部分)
- 正确的态度：把所有问题当程序来调试
  - 你写了一个程序，现在这个程序出 bug 了（例如 Segmentation Fault），你是怎样排查这个问题的？
  - curl: command not found
  - ‘sys/cdef.h’: No such file or directory
  - make run: No such file or directory



# 使用调试理论

- Debug (fault localization) 的基本理论回顾：
  - Fault (程序/输入/配置错) → Error → Failure (可观测)
    - 绝大部分工具的 Failure 都有 “原因报告”
    - 因此能帮助你快速定位 fault
  - man perror: 标准库有打印error message的函数
- 为什么会抓瞎？
  - 出错原因报告不准确或不够详细
  - 程序执行的过程不详
    - 既然我们有需求，那别人肯定也会有这个需求
      - 一定有信息能帮助我们！



# 调试理论：应用（2）

调试PA/Lab/任何代码

# 调试 NEMU 的难处

## 1. NEMU 没有这些 debug 信息

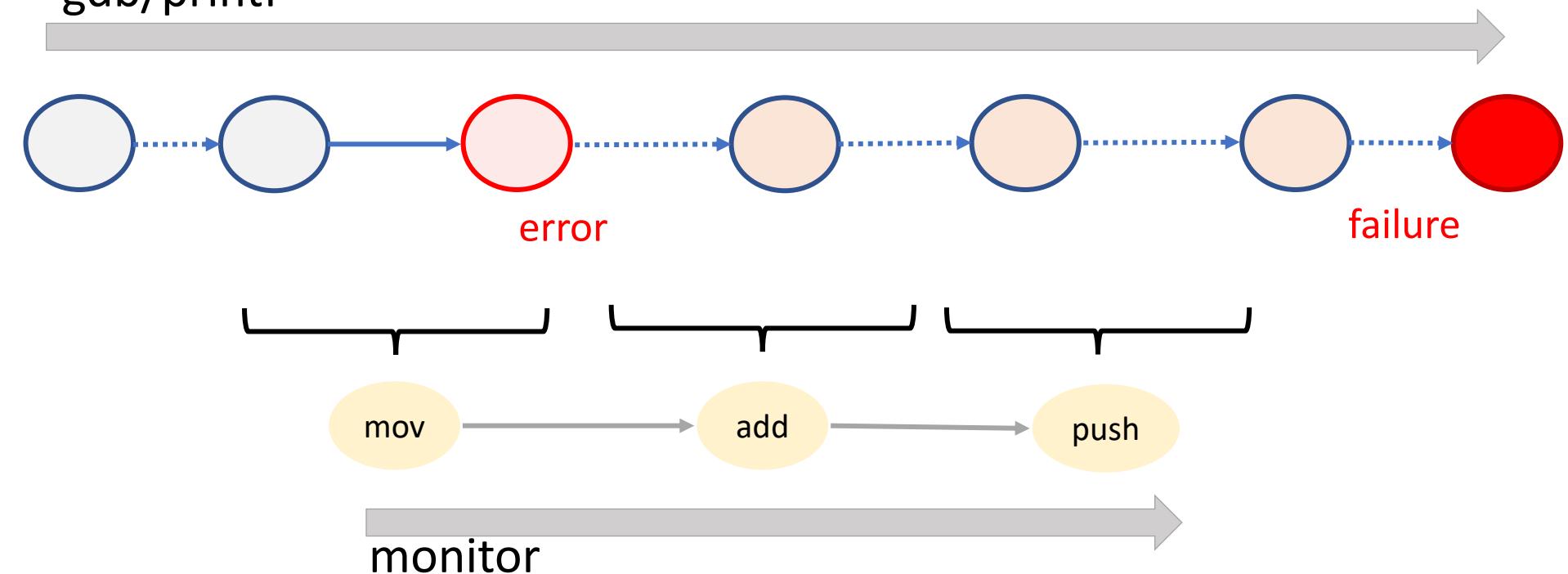
- 指令执行着执行着，悄悄就错了，直到在某个地方死循环
- 到底哪里错了呢？
  - 对于 cpu-test，还能一条一条看指令日志
  - 打字游戏的时候 bug 了，那么多指令……
  - LiteNES bug 了，这上哪玩啊……

## 2. NEMU 有两个层次的状态机

- NEMU binary 本身是个状态机
  - 它模拟了另一个状态机 (指令序列)
    - 这就是 monitor 的用处！



gdb.printf



# 调试理论回顾

---

需求 → 设计 → 代码 → Fault → Error → Failure

# 调试理论：应用 (Again)

需求 → 设计 → 代码 → Fault → Error → Failure

- “Technical Debt”

每当你写出不好维护的代码，你都在给你未来的调试挖坑。

- 中枪了？

- 为了快点跑程序，随便写的klib
- 为了赶紧实现指令，随手写的代码
- .....

# 编程基本准则：回顾

Programs are meant to be read by humans and only incidentally for computers to execute. — D. E. Knuth

(程序首先是拿给人读的，其次才是被机器执行。)

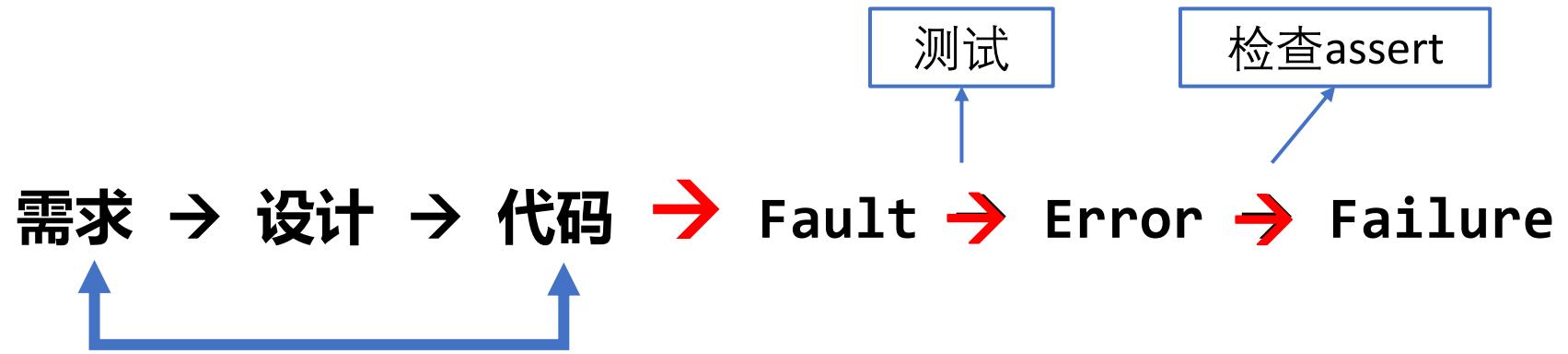
- 好的程序

- 不言自明：能知道是做什么的
  - 因此代码风格很重要
- 不言自证：能确认代码和 specification 一致
  - 因此代码中的逻辑流很重要

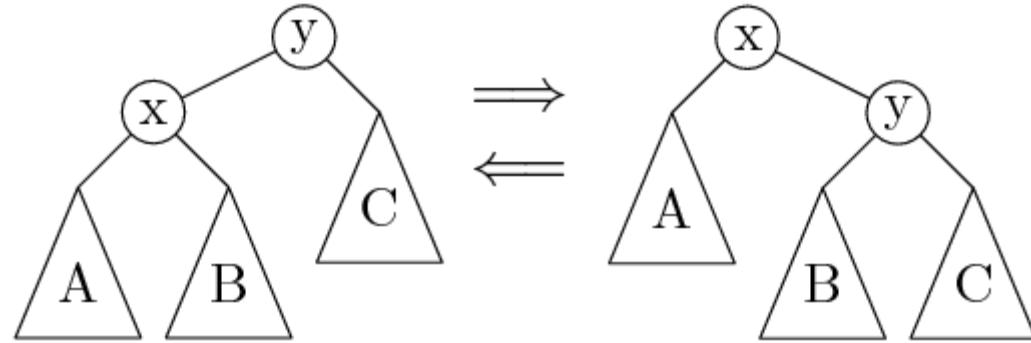
需求 → 设计 → 代码 → Fault → Error → Failure



# 调试理论：

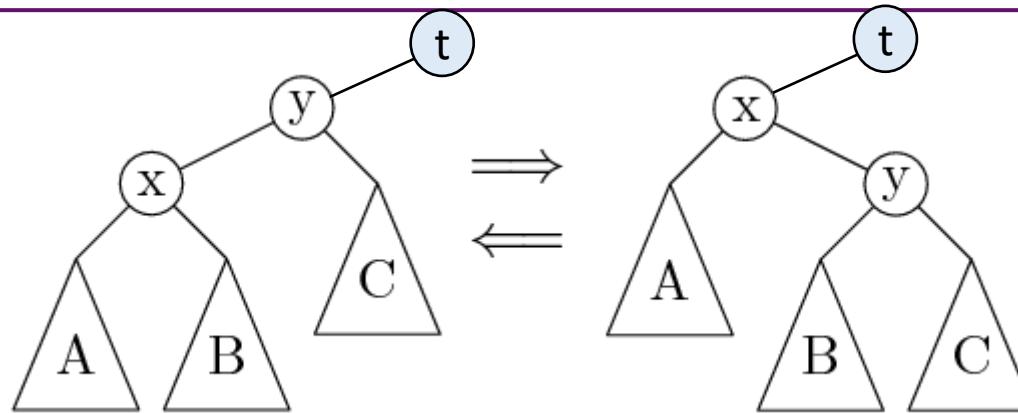


# 例子：平衡树的测试



# 例子：维护父亲节点的平衡树

parent  
left  
right



$t=y->\text{parent}$   
X  
y  
A  
B  
C

parent	left	right

需求 → 设计 → 代码 → Fault → Error → Failure

测试

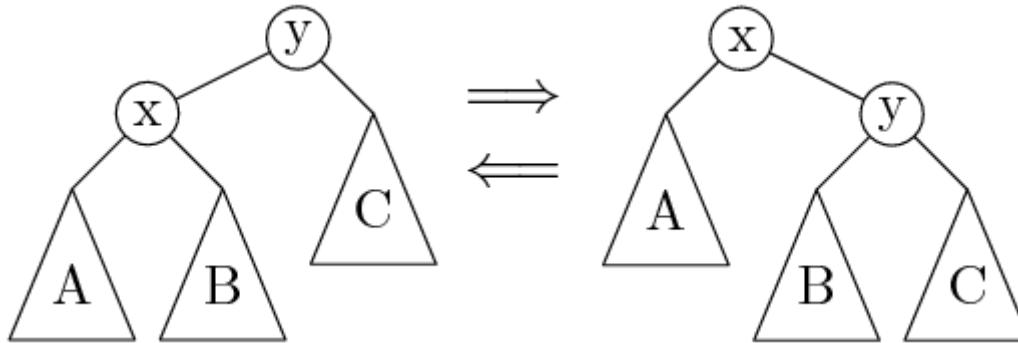
检查 assert

# 例子：维护父亲节

测试

检查assert

需求 → 设计 → 代码 → Fault → Error → Failure



// 结构约束

```
assert(u->parent == u ||  
       u->parent->left == u ||  
       u->parent->right == u);  
assert(!u->left || u->left->parent == u);  
assert(!u->right || u->right->parent == u);
```

// 数值约束

```
assert(!u->left || u->left->val < u->val);  
assert(!u->right || u->right->val > u->val);
```

# 调试理论的最重要应用

写好读、易验证的代码  
在代码中添加更多的断言 (assertions)

- 断言的意义

- 把代码中隐藏的 specification 写出来 (回到需求的另一种理解)
  - Fault → Error (靠测试)
  - Error → Failure (靠断言)
    - Error 暴露的越晚，越难调试
    - 追溯导致 assert failure 的变量值 (slice) 通常可以快速定位到 bug

# 例子：NEMU 中的断言

- 看起来很没必要，但可以提前拦截一些未知的 bug
  - 例如 memory error

```
static inline int check_reg_index(int index) {
    IFDEF(CONFIG_RT_CHECK, assert(idx >= 0 && idx < 32));
    return index;
}

#define gpr(idx) (cpu.gpr[check_reg_idx(idx)]._64)
```

```
Assert(map != NULL && addr <= map->high && addr >= map->low,
       "address (0x%08x) is out of bound {%s} [0x%08x, 0x%08x] at pc = "
       FMT_WORD, addr, (map ? map->name : "???"), (map ? map->low : 0), (map ?
       map->high : 0), cpu.pc);
```

# 福利：更多的断言

- 你是否希望在每一次指针访问时，都增加一个断言
  - `assert(obj->low <= ptr && ptr < obj->high);`

```
int *ref(int *a, int i) {  
    return &a[i];  
}  
  
void foo() {  
    int arr[64];  
    *ref(arr, 64) = 1; // bug  
}
```

- 一个神奇的编译选项
  - `-fsanitize=address`
  - **Address Sanitizer**; `asan` “动态程序分析”

\$

S 英 拼 · 简 拼 \*



# 调试理论：应用

---

**Fault → Error → Failure**

Fault → Error

- 充分的测试
  - 例子：Lab1 测试

Error → Failure

- 充足的日志
- 及时的检查

用好工具

- monitor, gdb, ...

# 例子：使用GDB

```
$ ./a.out  
Segmentation fault (core dumped)
```

- GDB: 最常用的命令在 [gdb cheat sheet \(unavailable? \)](#)
- Segmentation fault 是一个非常好的 failure
  - 某条指令访问了非法内存
  - 无非就是空指针、野指针、栈溢出、堆溢出.....
- 怎样定位 segmentation fault 发生时的语句/指令?
  - (core dumped)
  - gdb - 自带 backtrace, 95% 都能帮你定位到问题

# 想要更好的体验？ RTFM！

- 每次 gdb 都要输入一堆命令，都烦了
  - 当你感到不爽的时候，一定有办法解决
- gdb可以支持命令 (-x, -ex, ...)

```
set pagination off
set confirm off
layout asm
file a.out
b main
r
```

# 总结

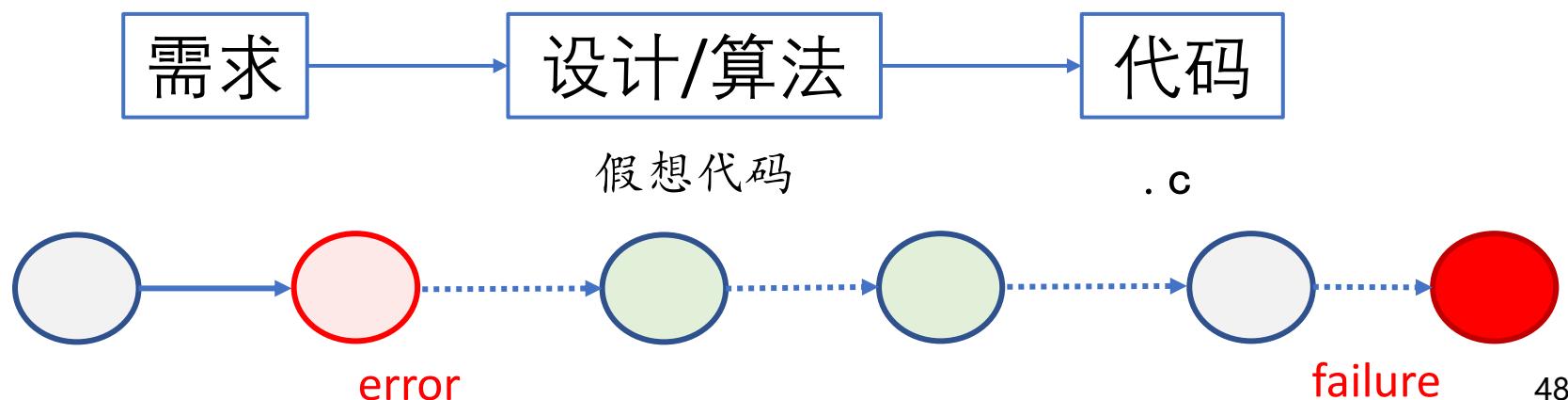
# 残酷的现实和难听的本质

---

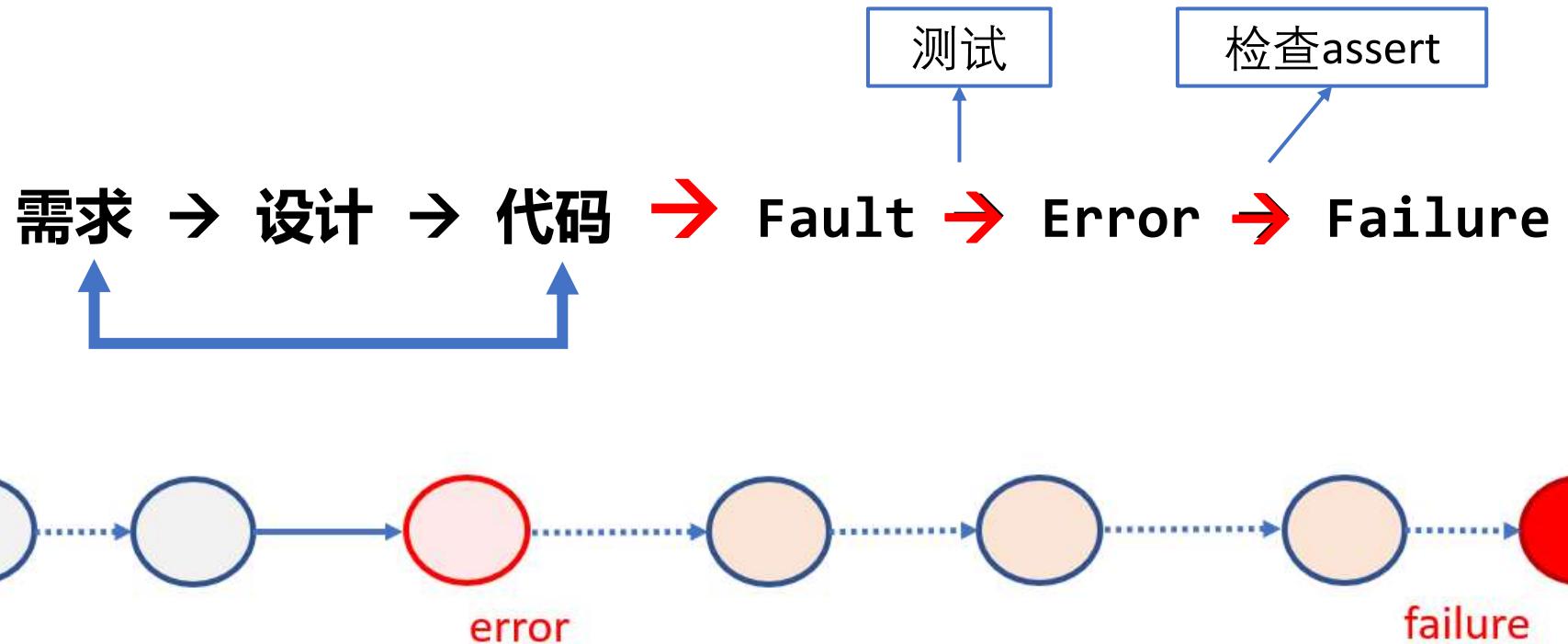
- 道理都懂，但出 bug 了，还是不知道怎么办
  - 一句难听的话
    - 你对代码的关键部分还不熟悉
    - 不知道如何判定程序状态是否正确
      - 对项目缺乏了解
      - 缺乏基础知识
      - 抱有“我不理解这个也行”的侥幸信息
- 《计算机系统基础》PA 给大家最重要的训练
  - 消除畏惧，但有些同学还没能理解
  - 努力做到知晓所有细节
  - 为你自己的代码负责

# 用好调试理论

- 调试理论
  - Fault → (测试) → Error → (断言) → Failure
    - 二分查找、观察trace、.....
- 调试实践
  - 理解程序如何完成对现实世界任务的抽象
  - 我们很容易用一小句话来概括一大段程序执行过程
  - 并且排除/确认其中的问题



# 调试理论：



# 史上著名的bug们

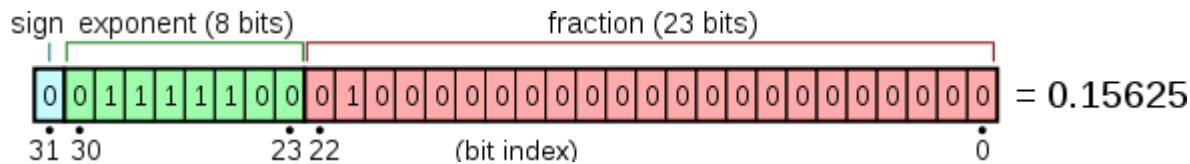
- First bug



- 千年虫



- Intel浮点数bug



假想代码

. c

End.

不要放弃PA。

# I/O设备选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 提醒

- PA

PA2: Deadline: 2023年11月19日 23:59:59

- Lab

Lab2: Deadline: 2023年11月19日 23:59:59

# 本讲概述

对所有学计算机的同学来说，“设备”才是真正触摸到的，但设备到底是什么？

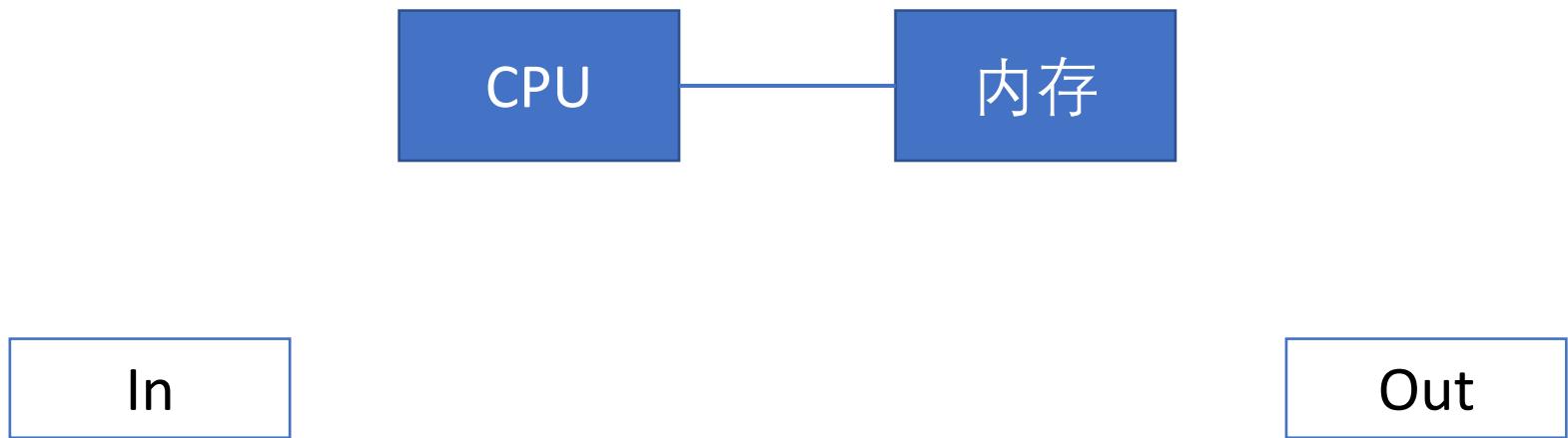
按下键盘之后，计算机硬件/软件系统到底发生了什么？

- 本讲内容

- 处理器-设备接口
- I/O设备选讲

# 处理器-设备接口

## Turing Machine

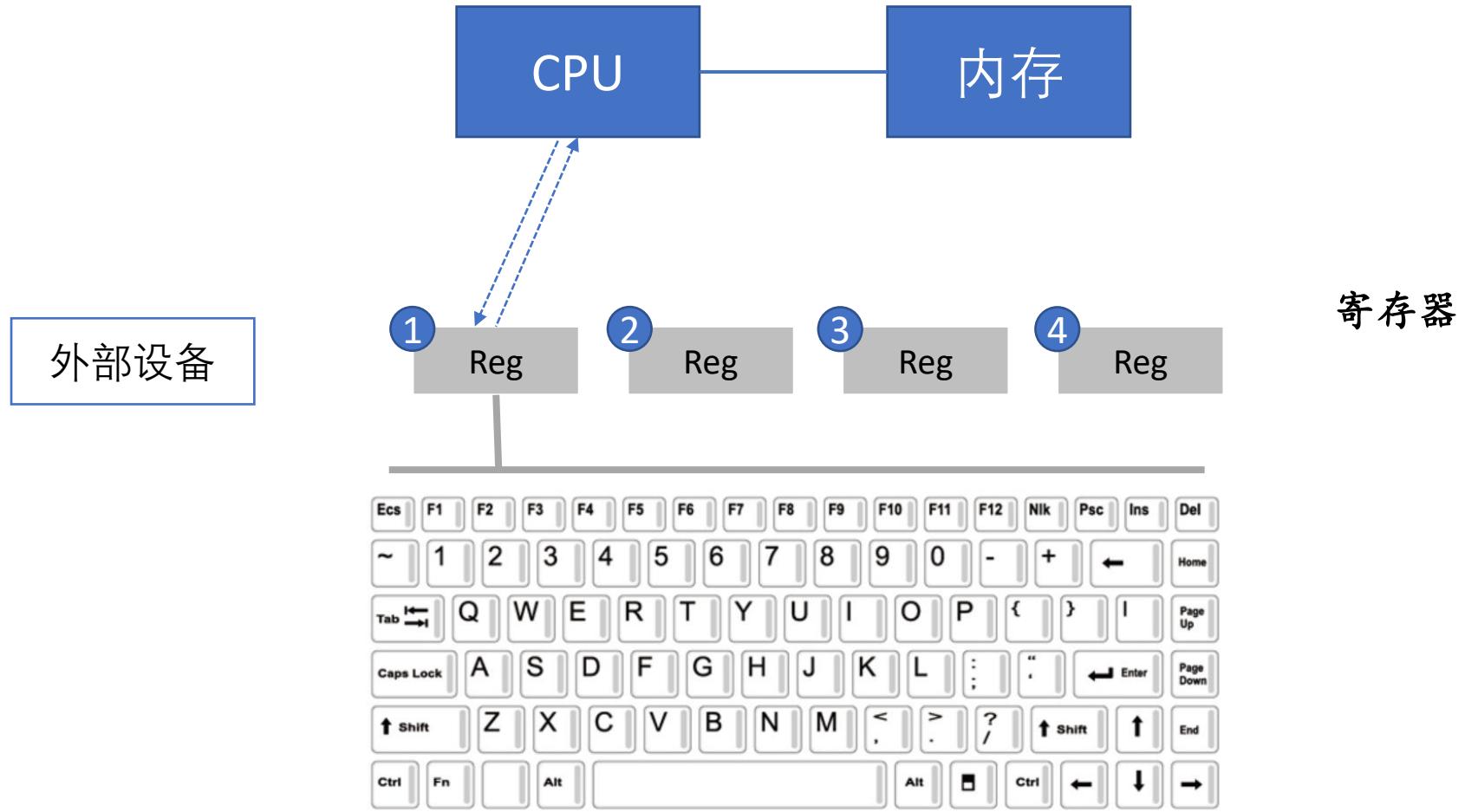


# 回顾：为什么会有I/O设备

1950s：随着计算机运算速度和存储器的发展，计算机已经快到可以“计算”人类日常任务了。

- 计算机 “Turing Machine” 中的一切都是数据
  - 因此计算机一定会提供一个机制
    - 从设备中读取数据 (input data)
      - 键盘按键的代码、鼠标移动的偏移量、.....
    - 向设备写入数据 (output data)
      - 输出到打印机的字符串、屏幕上显示的像素.....
    - 设备可以向处理器发送中断 (后话)

## Turing Machine



# 设备-处理器接口

设备 = 一组寄存器，每次可以交换一定数量的数据。

- 每个寄存器有特定的含义和格式
  - 主要功能：读取数据/写入数据/配置设备

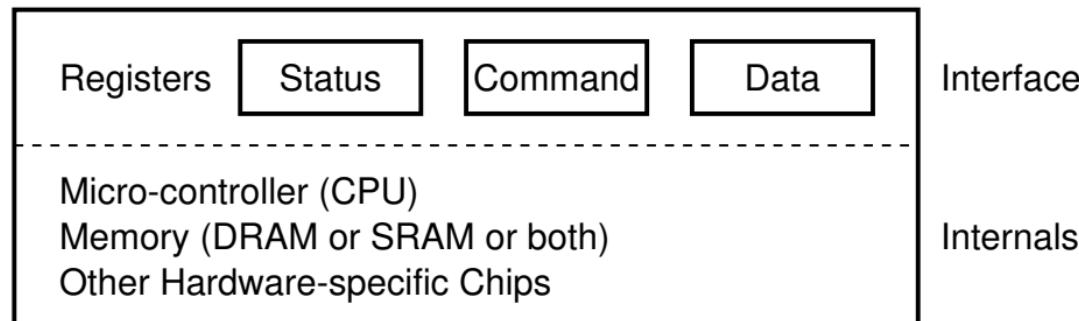
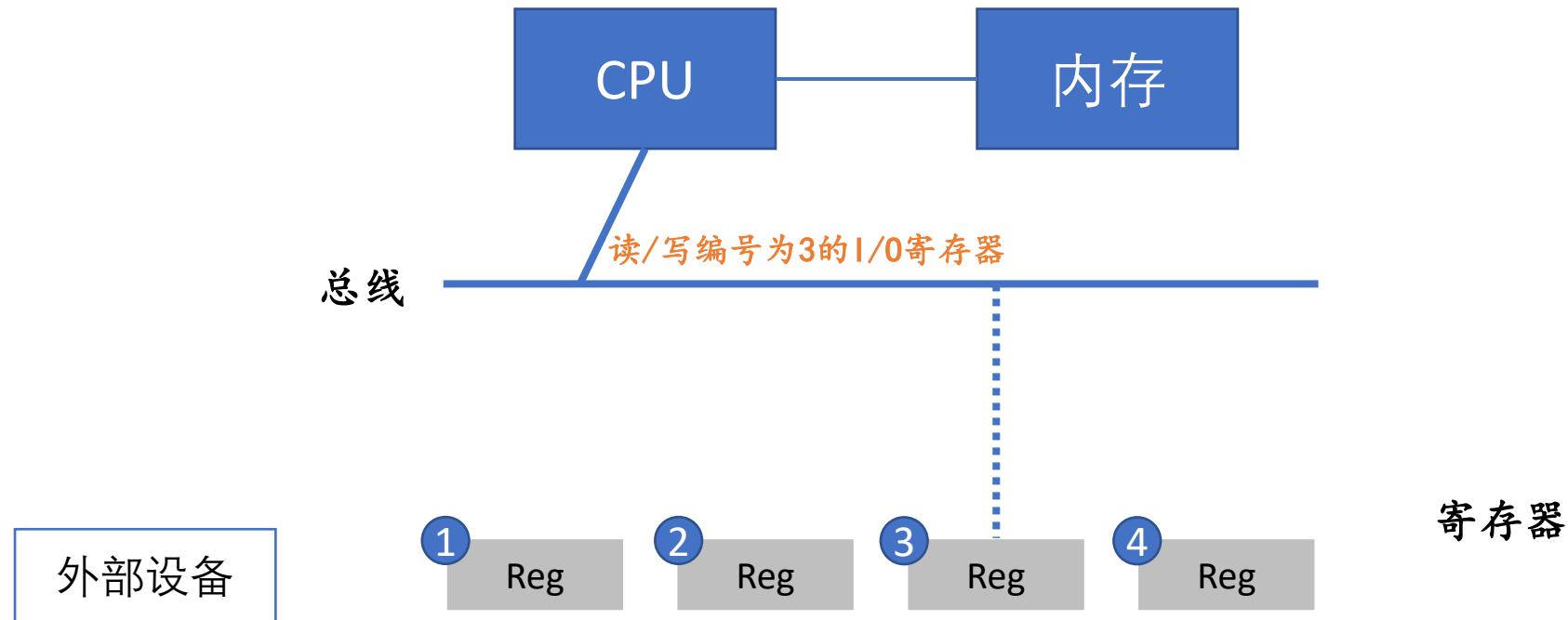
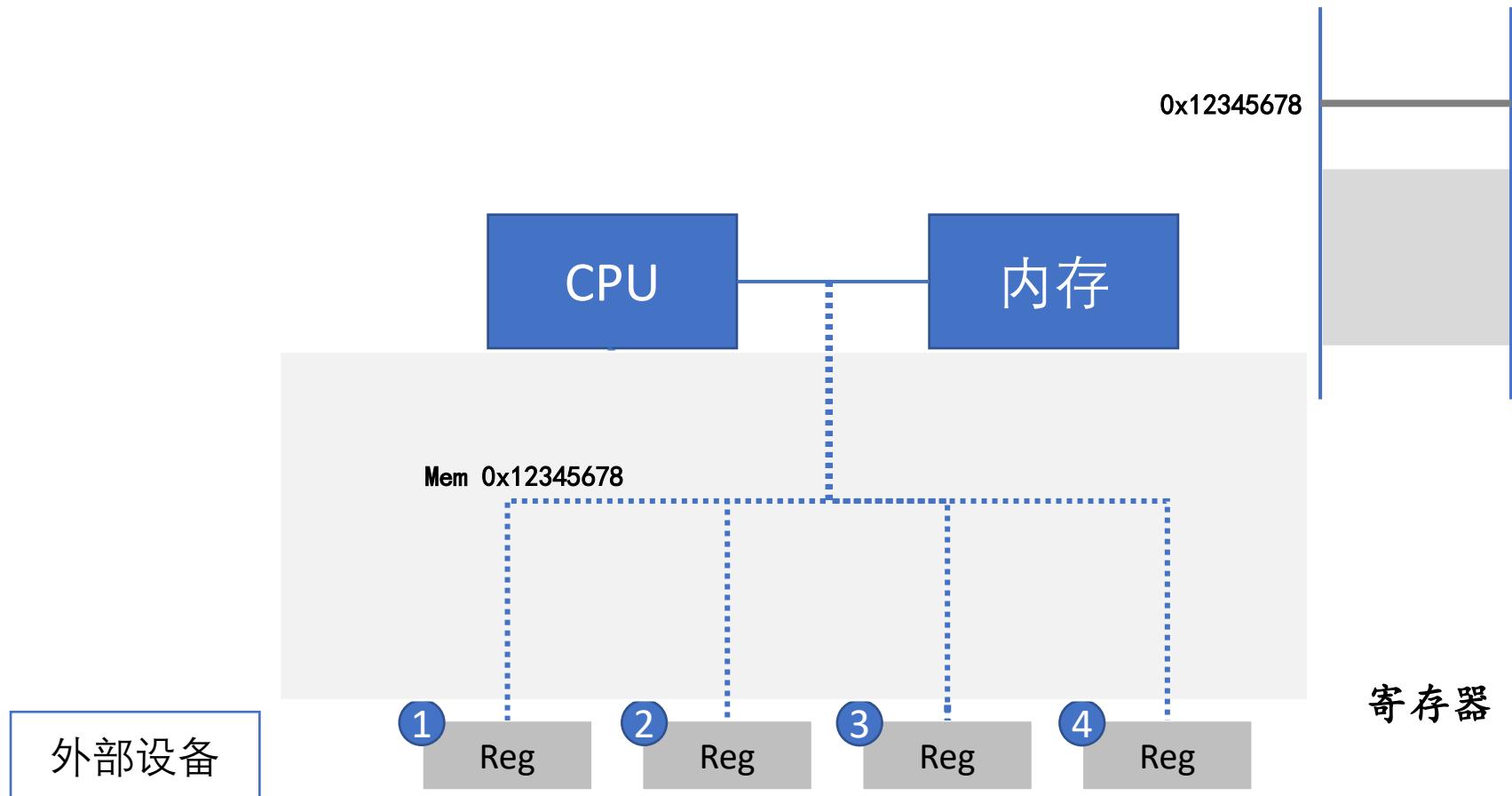


Image source: [OSTEP](#)

# I/O交换方式1



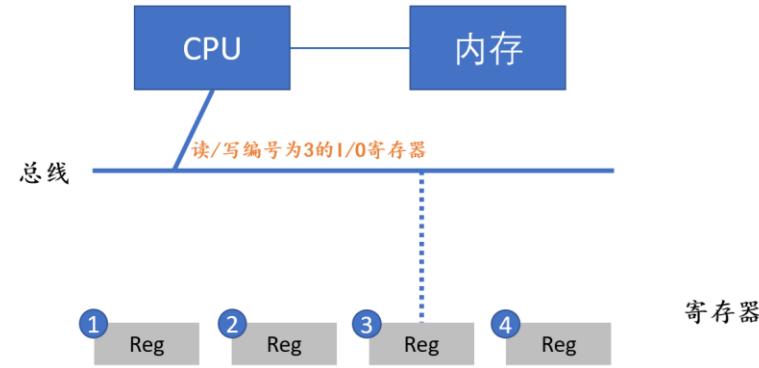
# I/O交换方式2



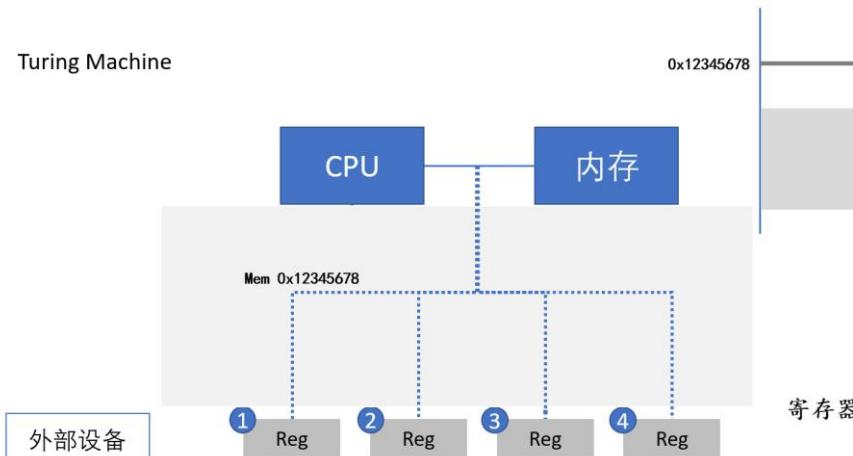
# 设备-处理器接口 (cont'd)

- CPU可以直接通过指令读写这些寄存器

- Port-mapped I/O (PMIO)
  - I/O地址空间 (port)
  - CPU直连I/O总线



- Memory-mapped I/O (MMIO)
  - 直观：使用普通内存读写指令就能访问
  - 带来了一些设计和实现的麻烦：编译器优化、缓存、乱序执行



管理 控制 视图 热键 设备 帮助

```
1 #define ADDR 0X12345678
2
3 void foo(){
4     for (int i = 0; i<1024; i++){
5         //out(ADDR, 0);
6         (*[(char *)]ADDR) = 0;
7     }
8
9 }
```

~/Documents/ICS2021/teach/I0/a.c[+1] [c] unix utf-8 Ln 6 英 拼



Right Shift + Right Alt

# 处理器眼中的I/O设备：x86-qemu UART

- “COM1”; putch()的实现

```
#define COM1 0x3f8

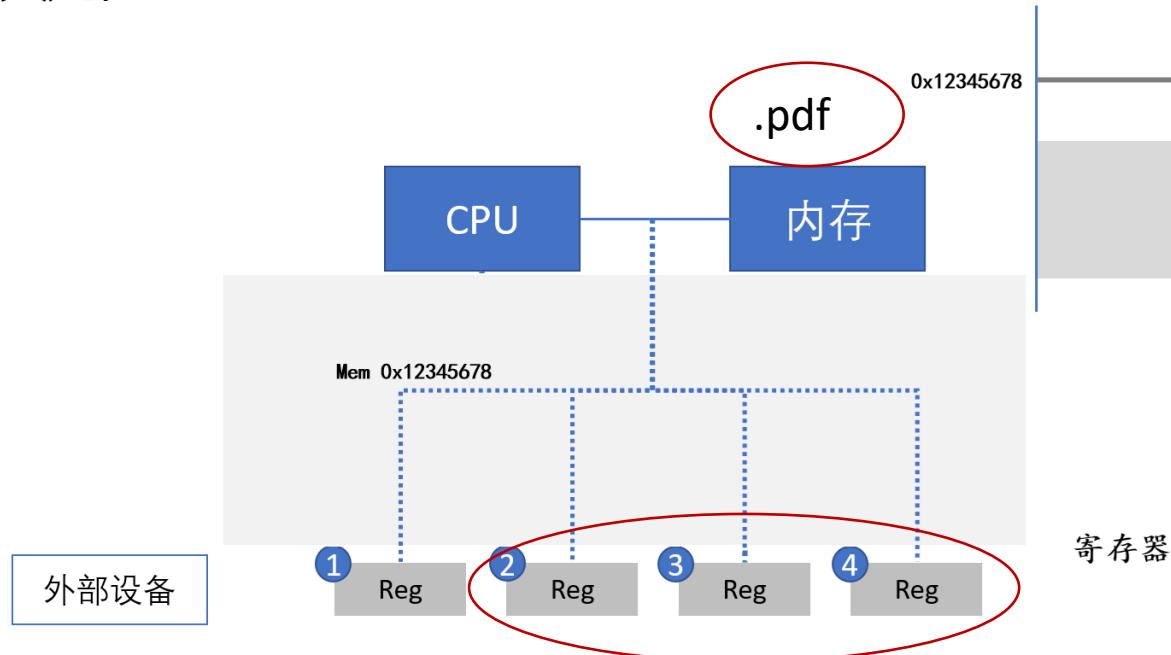
static int uart_init() {
    outb(COM1 + 2, 0); // 控制器相关细节
    outb(COM1 + 3, 0x80);
    outb(COM1 + 0, 115200 / 9600);
    ...
}

static void uart_tx(AM_UART_TX_T *send) {
    outb(COM1, send->data);
}

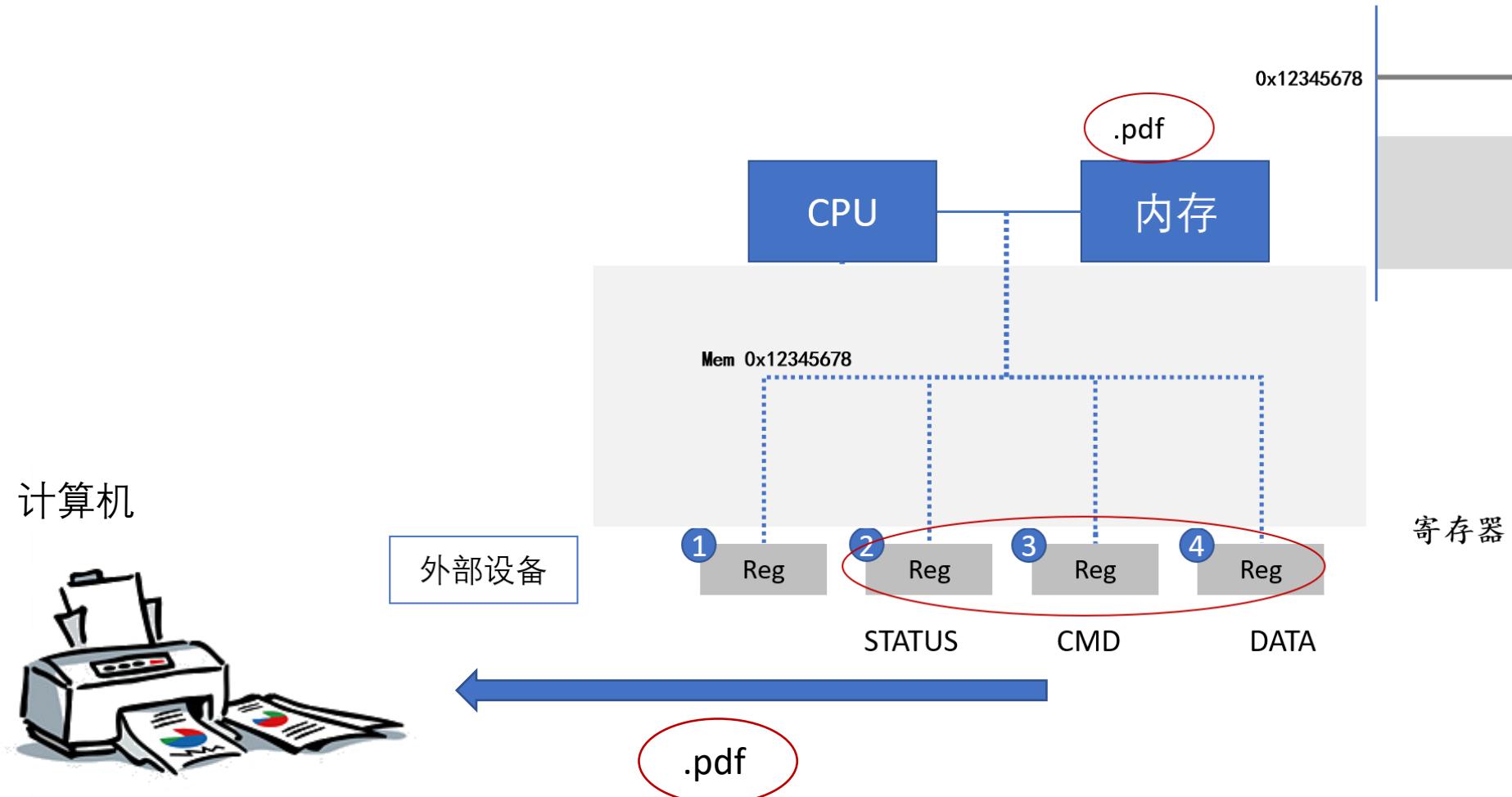
static void uart_rx(AM_UART_RX_T *recv) {
    recv->data = (inb(COM1 + 5) & 0x1) ? inb(COM1) : -1;
}
```

# 更复杂的设备：打印机与PostScript

- 打印机：将字节流描述的文字/图形打印到纸张上
  - 可简单 (ASCII 文本序列，像打字机一样打印)
  - 可复杂 (编程语言描述的图形)
    - 高清全页图片的传输是很大的挑战
    - 尤其是在 1980s



# 更复杂的设备：打印机与PostScript



# 更复杂的设备：打印机与PostScript

---

- 打印机：将字节流描述的文字/图形打印到纸张上
  - 可简单 (ASCII 文本序列，像打字机一样打印)
  - 可复杂 (编程语言描述的图形)
    - 高清全页图片的传输是很大的挑战
    - 尤其是在 1980s
- 例子：PostScript (1984)
  - 一种描述页面布局的 domain-specific language
    - 类似于汇编语言
    - 可以在命令行中创建高质量的文稿
  - PDF 是它的 superset
    - 例子：[page.ps](#)

管理 控制 视图 热键 设备 帮助

\$

S 英 拼 简 拼 中

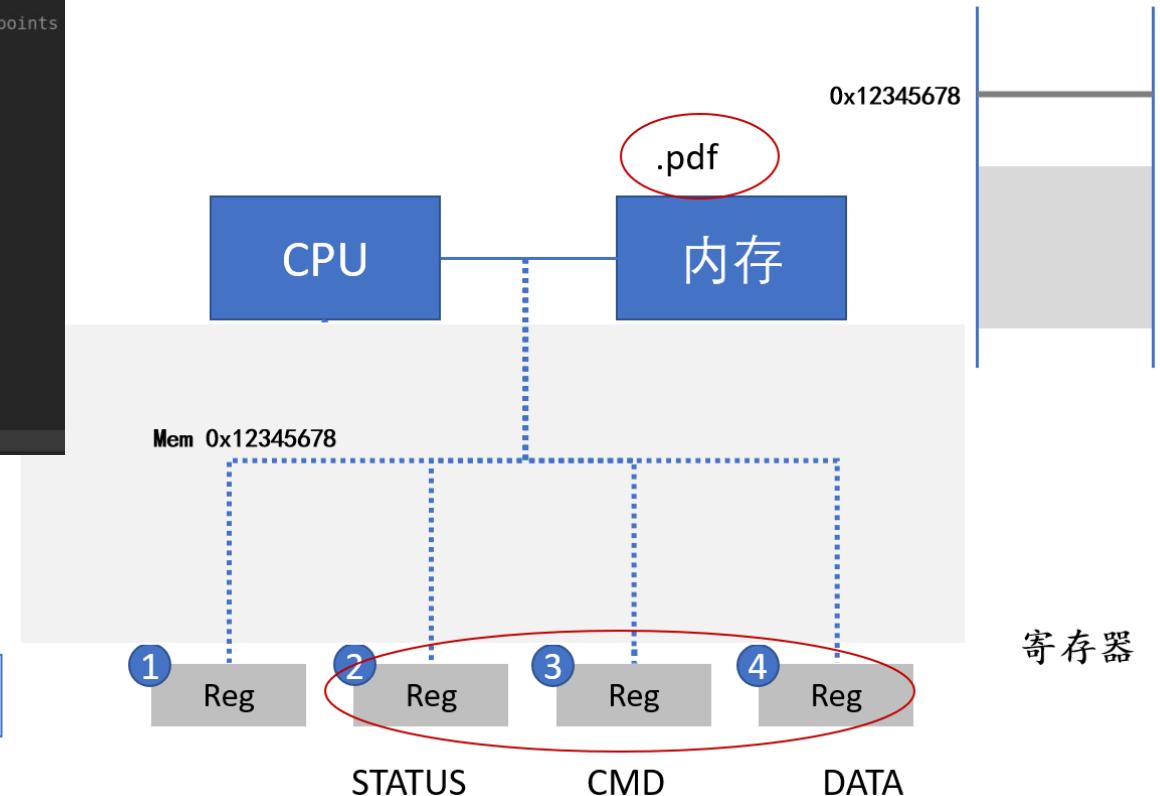
Right Shift + Right Alt

# 更复杂的设备：打印机与PostScript

```
3  
4 72 72 scale % scale coordinate system so units are inches, not points  
5 2 2 translate % put origin 2 inches from lower left of page  
6  
7 /Symbol findfont 4 scalefont setfont  
8 % current font is now Symbol about 4 inches high  
9 gsave % save graphics state (coordinate system & stuff)  
10 .5 setgray  
11 60 rotate  
12 0 0 moveto (abcde) show  
13 grestore % restore previous graphics state  
14  
15 /Helvetica-Bold findfont .2 scalefont setfont  
16 % current font is now slanted Helvetica about .2 inches high  
17 0 4 moveto (Postscript is good at this stuff) show  
18  
19 /Times-Italic findfont 2 scalefont setfont  
20 % current font is now italic Times about 2 inches high  
21 1 0 0 setrgbcolor  
22 0 6 moveto (yowza!) show  
23  
24 showpage
```



外部设备



# 更复杂的设备：打印机与PostScript

---

- 打印机：将字节流描述的文字/图形打印到纸张上
  - 可简单 (ASCII 文本序列，像打字机一样打印)
  - 可复杂 (编程语言描述的图形)
    - 高清全页图片的传输是很大的挑战
    - 尤其是在 1980s
- 例子：PostScript (1984)
  - 一种描述页面布局的 domain-specific language
    - 类似于汇编语言
    - 可以在命令行中创建高质量的文稿
  - PDF 是它的 superset
    - 例子：[page.ps](#)

# 两个特殊的I/O设备

- 总线

- 系统里可能有很多(甚至是可变的)I/O设备
  - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU可以只直接连接到总线
  - 总线可以连接其他总线
  - 例子: `lspci -t`, `lsusb -t`

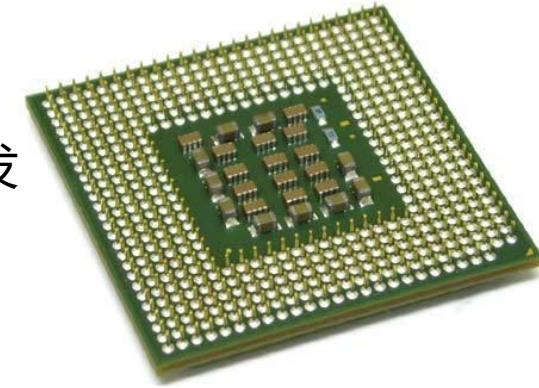
- 中断控制器

- 管理多个产生中断的设备
  - 汇总成一个中断信号给CPU
  - 支持中断的屏蔽、优先级管理等

# 两个特殊的I/O设备

- 总线

- 系统里可能有很多(甚至是可变的)I/O设备
  - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU可以只直接连接到总线
  - 总线可以连接其他总线
  - 例子: `lspci -t, lsusb -t`



```
#define COM1 0x3f8

static int uart_init() {
    outb(COM1 + 2, 0); // 控制器相关细节
    outb(COM1 + 3, 0x80);
    outb(COM1 + 0, 115200 / 9600);
    ...
}

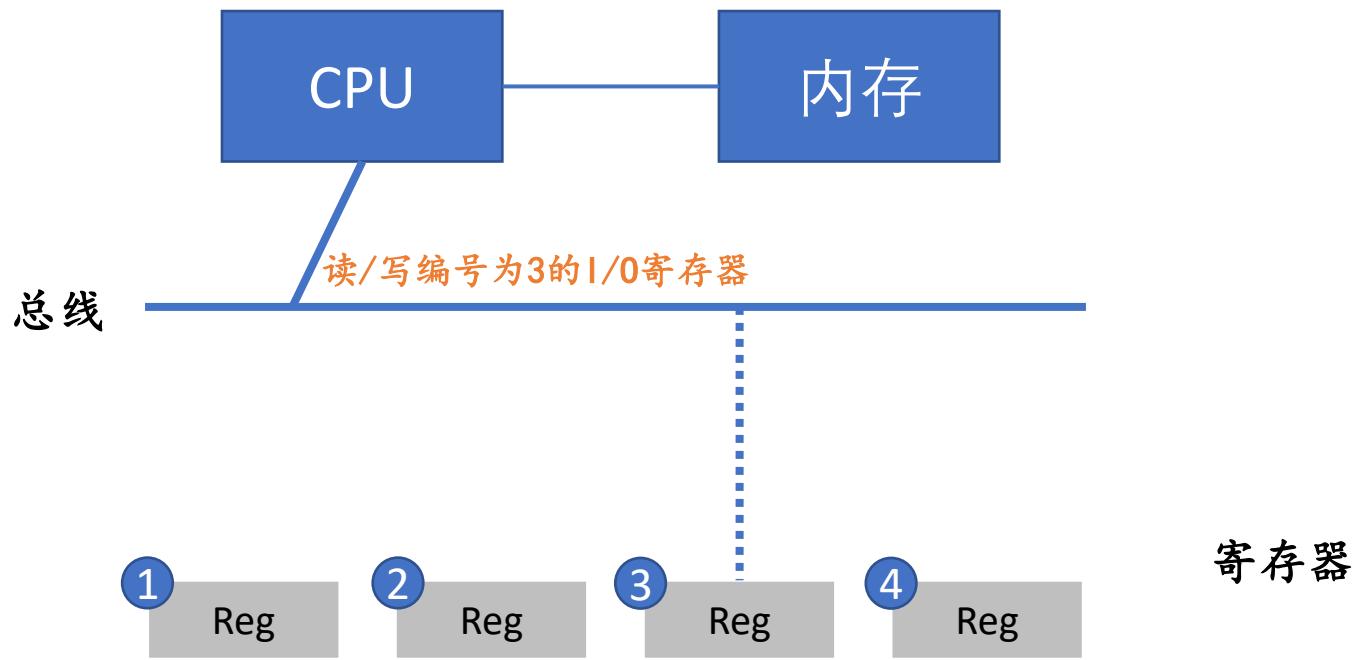
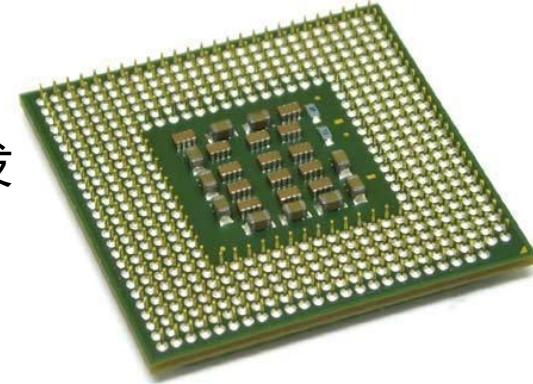
static void uart_tx(AM_UART_TX_T *send) {
    outb(COM1, send->data);
}

static void uart_rx(AM_UART_RX_T *recv) {
    recv->data = (inb(COM1 + 5) & 0x1) ? inb(COM1) : -1;
}
```

# 两个特殊的I/O设备

- 总线

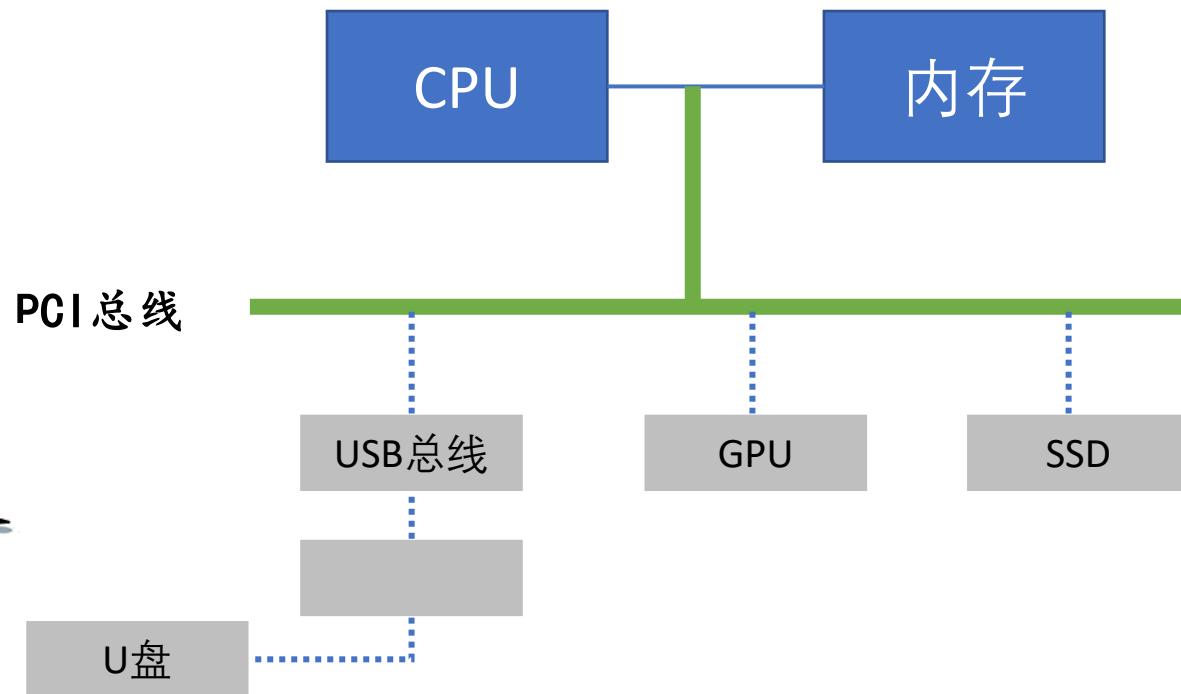
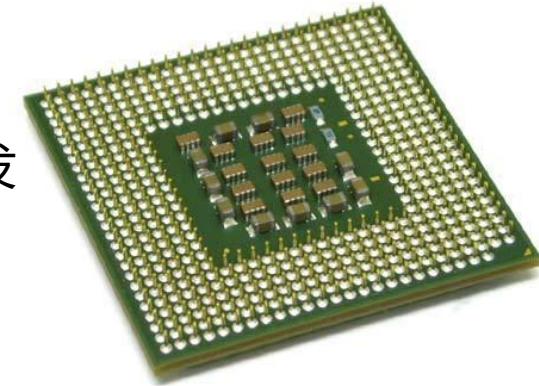
- 系统里可能有很多(甚至是可变的)I/O设备
  - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU可以只直接连接到总线
  - 总线可以连接其他总线
  - 例子: `lspci -t`, `lsusb -t`



# 两个特殊的I/O设备

- 总线

- 系统里可能有很多(甚至是可变的)I/O设备
  - 总线实现了设备的查找、映射、和命令/数据的转发
- CPU可以只直接连接到总线
  - 总线可以连接其他总线
  - 例子: `lspci -t`, `lsusb -t`

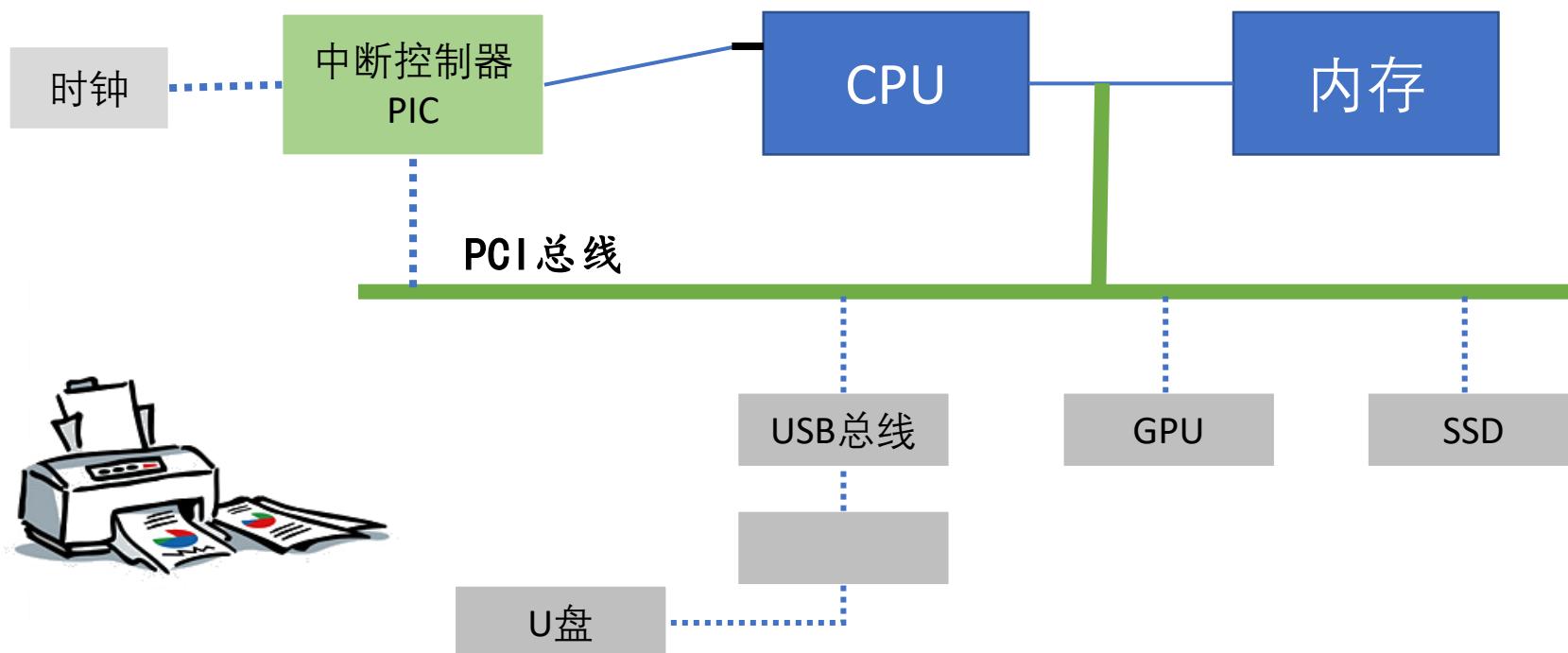


```
$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: VMware SVGA II Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SAT A Controller [AHCI mode] (rev 02)
```

```
$ lsusb
Bus 001 Device 002: ID 80ee:0021 VirtualBox USB Tablet
Bus 001 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

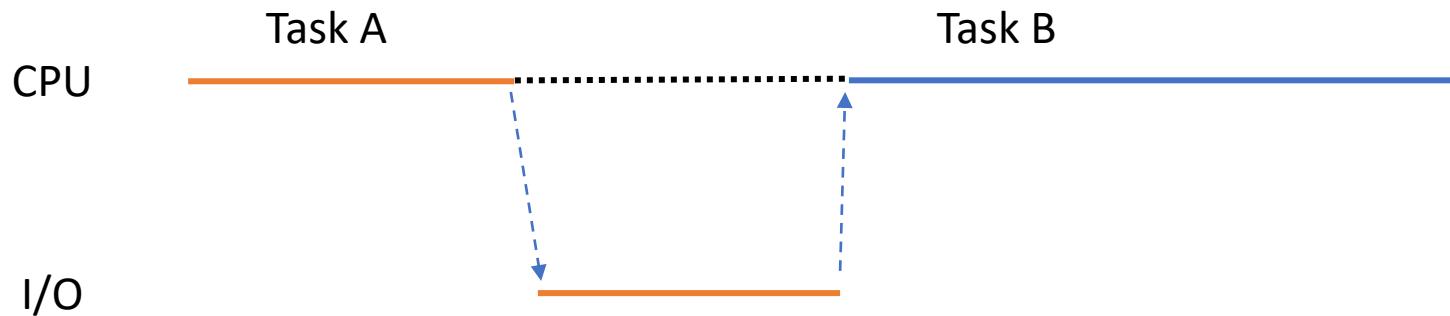
# 两个特殊的I/O设备

- 中断控制器
  - 管理多个产生中断的设备
    - 汇总成一个中断信号给 CPU
    - 支持中断的屏蔽、优先级管理等



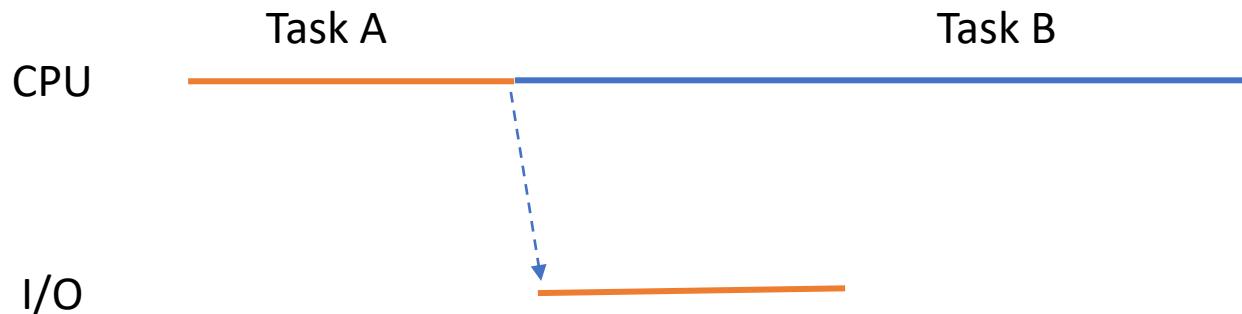
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



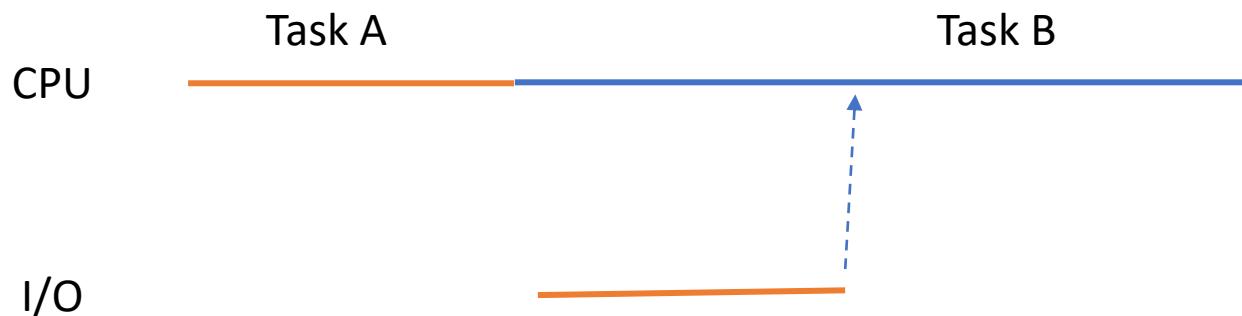
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动的函数调用
  - 相当于在每条语句后都插入

```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
  - 于是就有了最早的操作系统：管理I/O设备的库代码

# 处理器-设备接口：NES实现（1983）

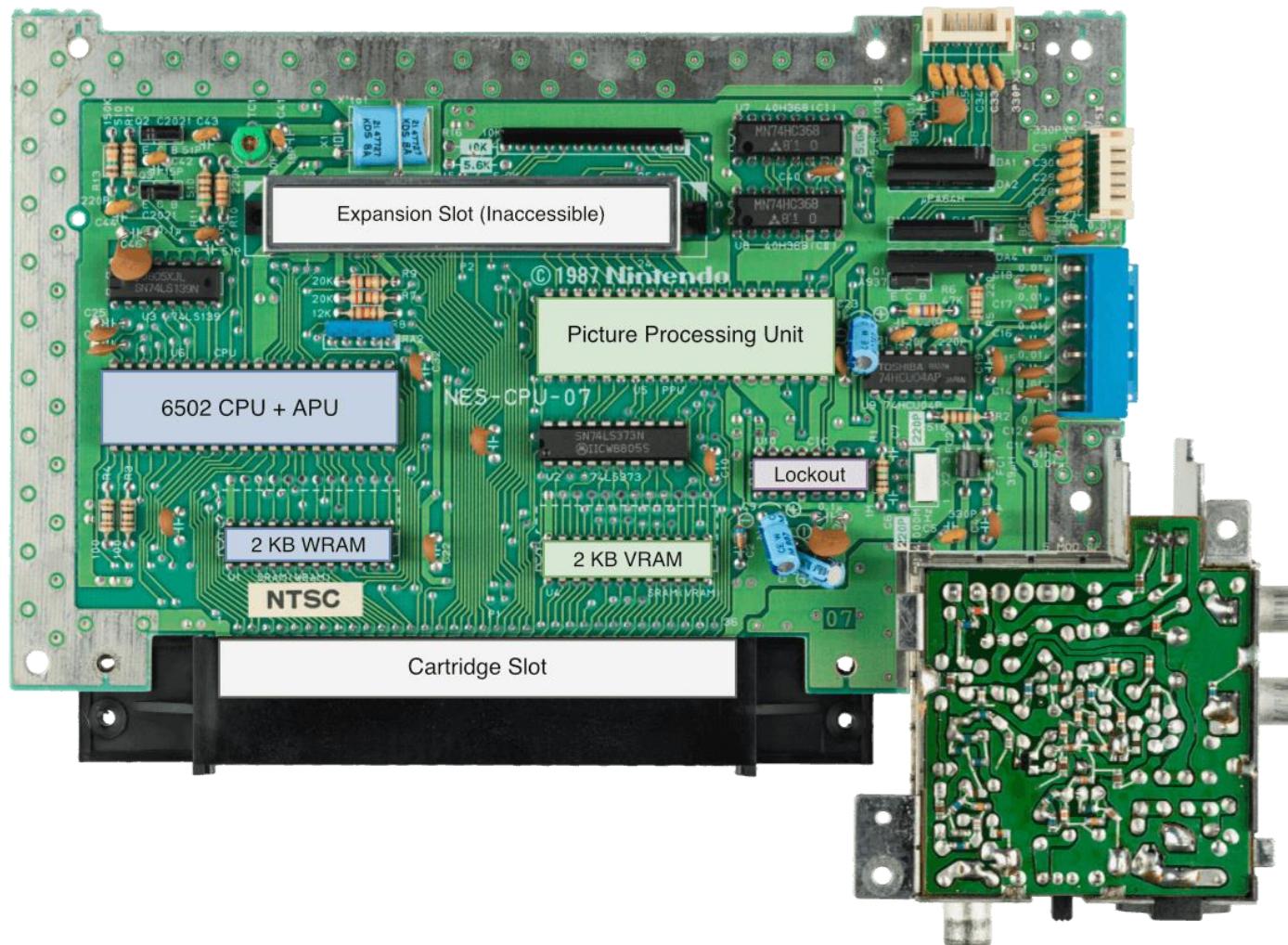
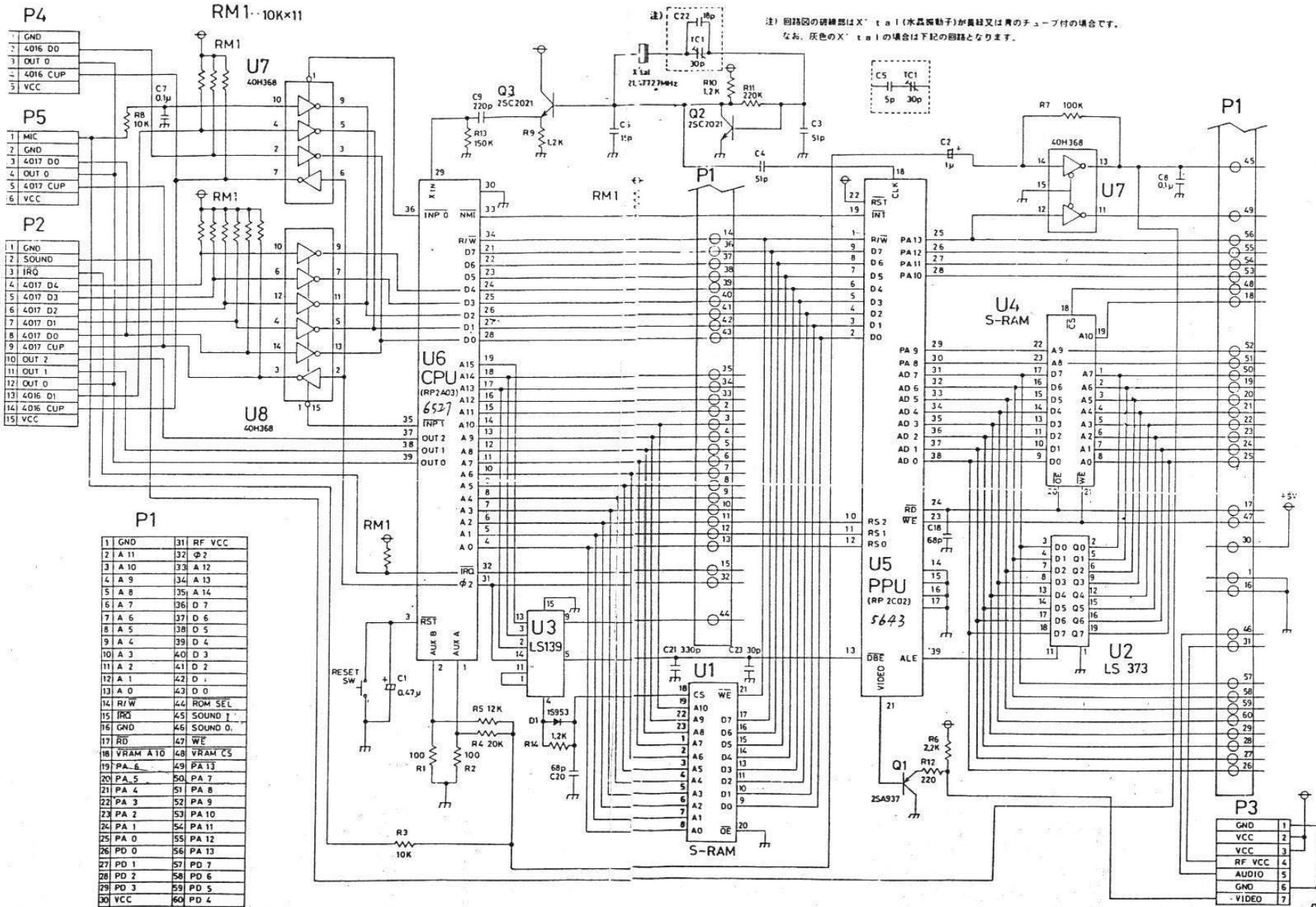


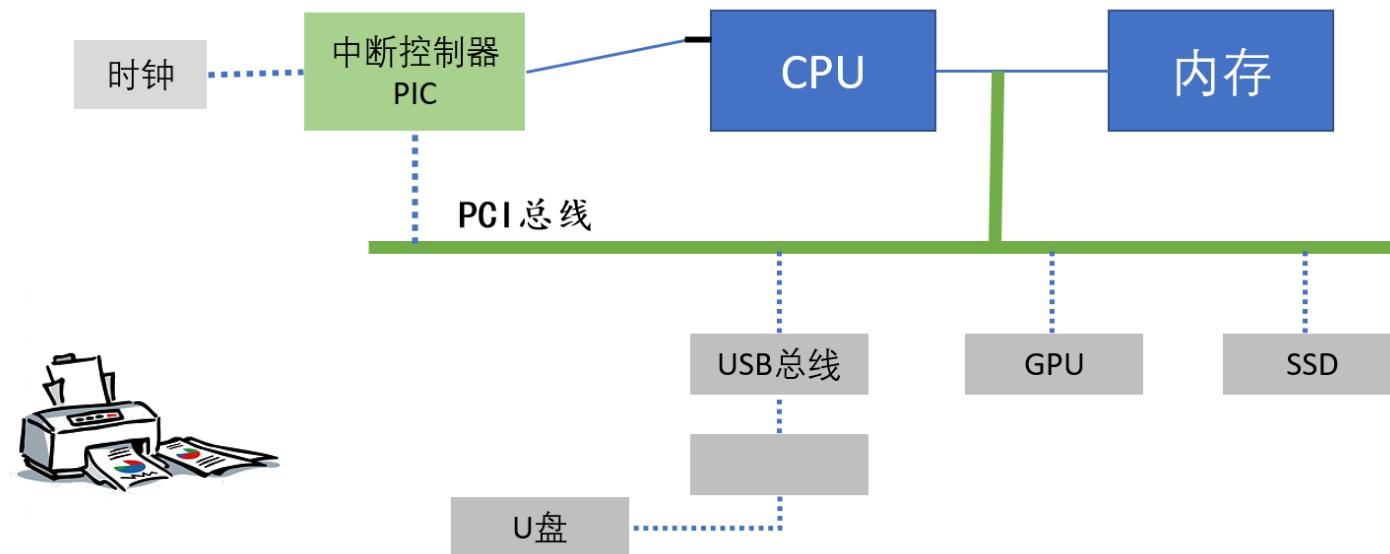
Image by [Rodrigo Copetti](#); 以及 [电路图](#)

# CPU基板回路図

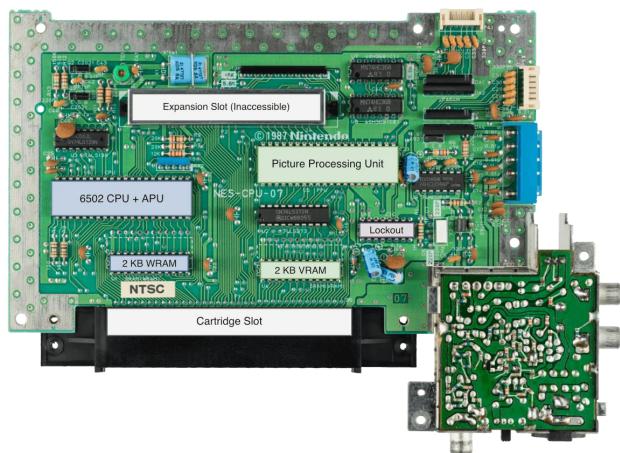


# 小结

- I/O 设备：“小计算机”
  - 完成和物理世界的交互功能
  - 连接到总线/中断控制器
  - CPU 通过 PIO/MMIO 访问
- 今天很多 I/O 设备都带有或简单或复杂的 CPU
  - 带跑马灯/编程功能的键盘和鼠标
  - 显示加速器和“显卡计算”



# 2D图形绘制硬件



# 理论：一切皆可 “计算”

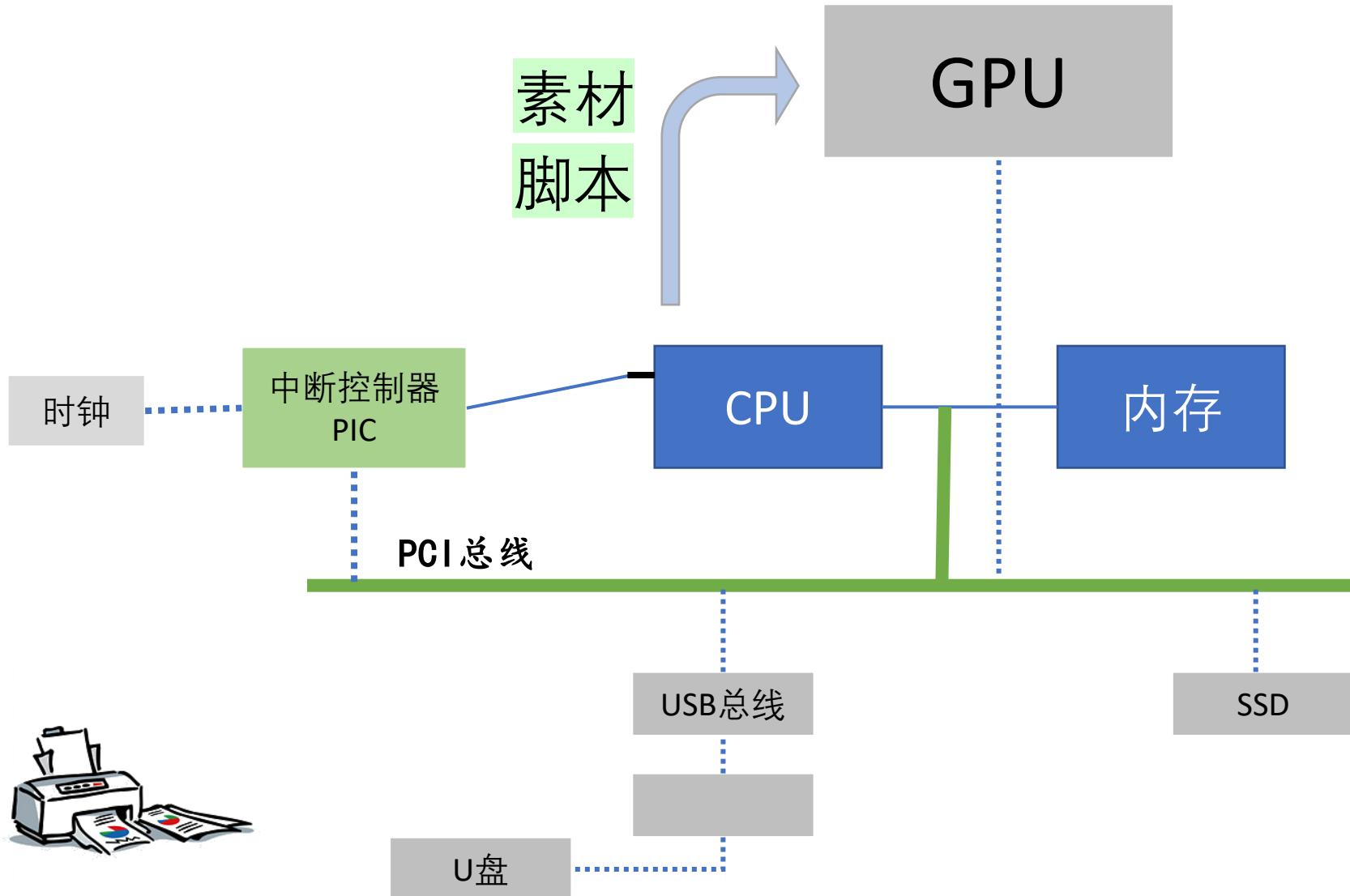
```
for (int i = 1; i <= H; i++) {  
    for (int j = 1; j <= W; j++)  
        putchar(j <= i ? '*' : ' ');  
    putchar('\n');  
}
```

\*  
\* \*  
\* \* \*

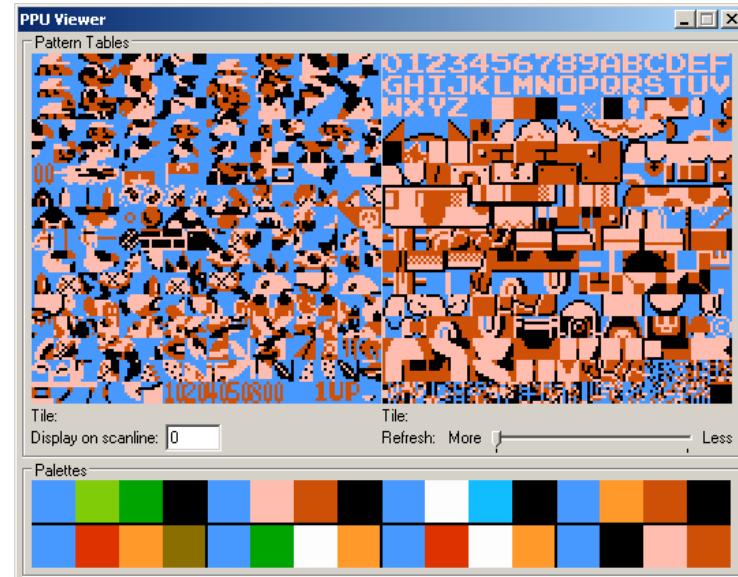
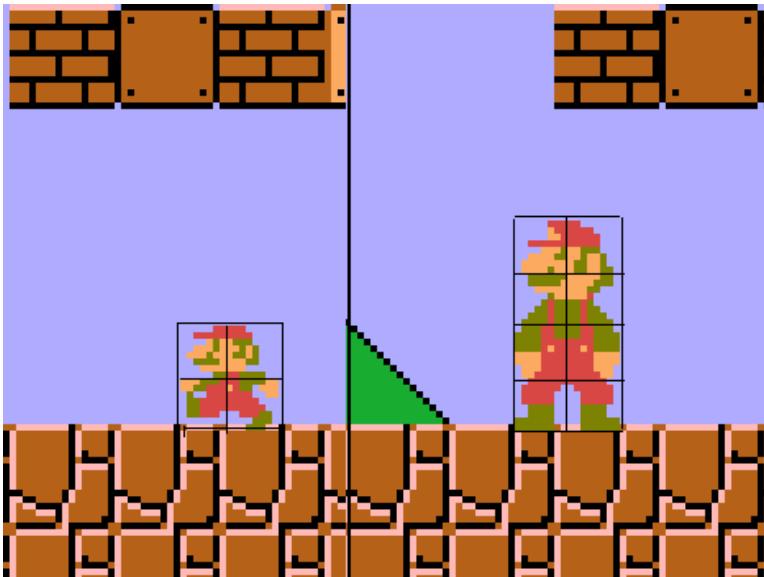
f

- 难办的是性能
  - NES: 6502 @ 1.79Mhz; IPC = 0.43
    - 屏幕共有  $256 \times 240 = 61K$  像素 (256 色)
    - 60FPS → 每一帧必须在 ~10K 条指令内完成
      - 如何在有限的 CPU 运算力下实现 60Hz?

# 画什么？怎么画？



# PR2C02 Picture Processing Unit (PPU)



- CPU 只描述 8x8 “贴块” 的摆放方法
- 类似于 PostScript 脚本
  - 背景是 “大图” 的一部分
  - 每行的前景块不超过 8 个
- PPU 完成图形的绘制

76543210

|||||||

|||||||++- Palette

|||++++- Unimplemented

||+----- Priority

|+----- Flip horizontally

+----- Flip vertically

# PPU Viewer



## Pattern Tables



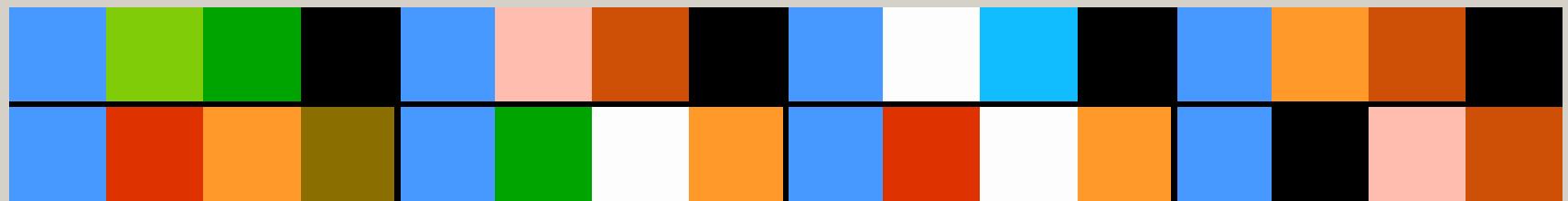
Tile:

Display on scanline:

Tile:

Refresh: More  Less

## Palettes

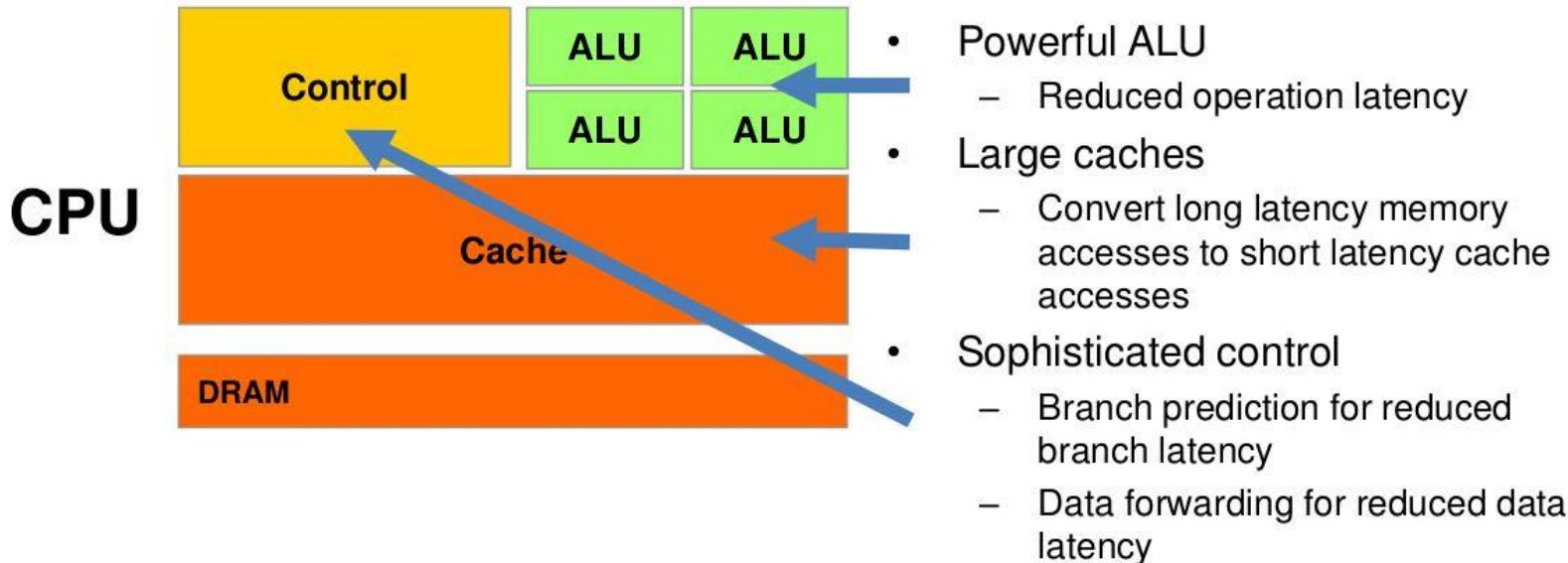


# PPU: 只执行一个程序的CPU

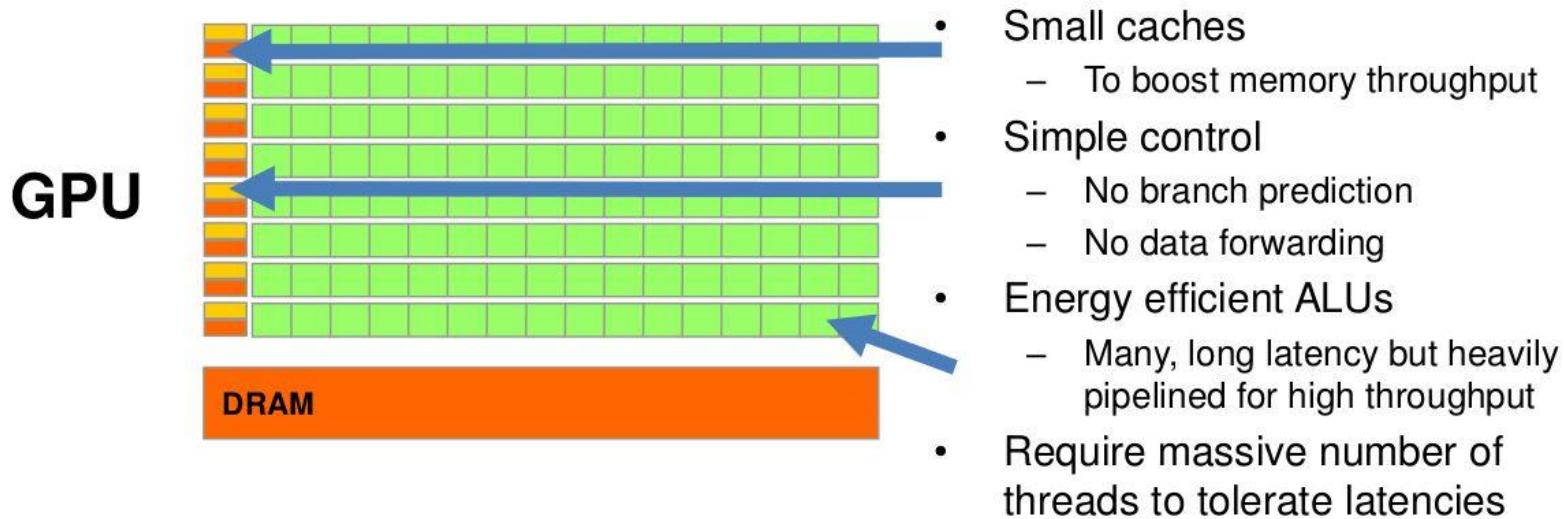
```
for (int x = 0, pos = 0; x < HEIGHT; x++) { // 行扫描
    for (int y = 0; y < WIDTH; y++, pos++) {
        vbuf[pos] = draw(x, y); // 算出 (x,y) 的贴块 (和颜色)
    }
}
```

地址空间	大小	功能
\$0000-\$1FFF	8 KB	<a href="#">Pattern tables</a>
\$2000-\$2FFF	4 KB	<a href="#">Nametables</a>
\$3000-\$3EFF	3.75KB	Mirrors of \$2000-\$2EFF
\$3F00-\$3F1F	32B	<a href="#">Palette RAM</a> indexes
\$3F20-\$3FFF	224B	Mirrors of \$3F00-\$3F1F
OAM (DMA 访问)	256B	Sprite Y, #, attribute, X

## CPUs: Latency Oriented Design



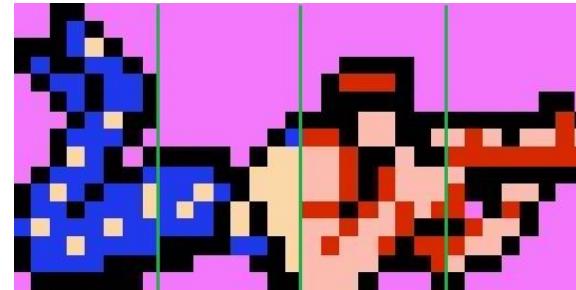
## GPUs: Throughput Oriented Design



# 在受限的机能下提供丰富的图片

- 前景：

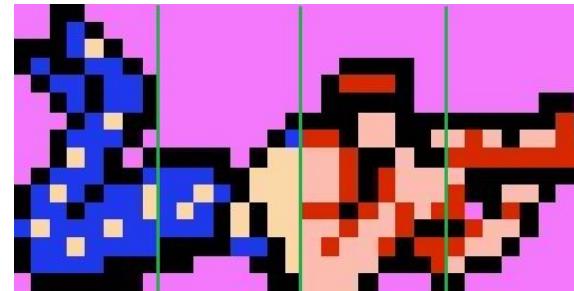
- 《魂斗罗》(Contra) 中角色为什么要「萝莉式屈腿俯卧」？



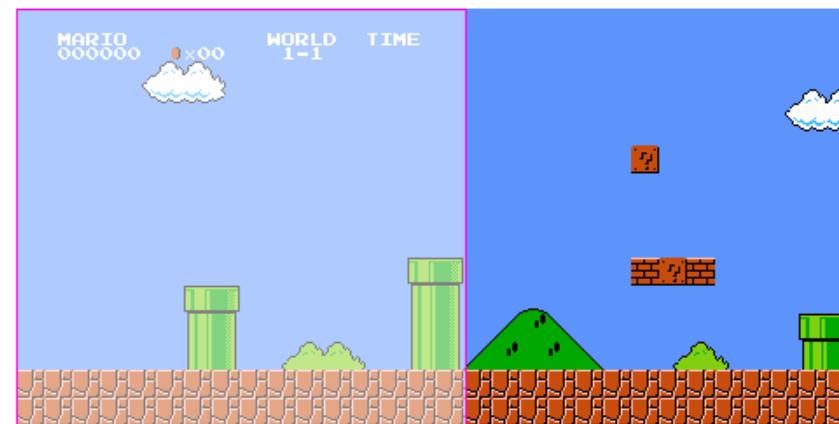
# 在受限的机能下提供丰富的图片

- 前景：

- 《魂斗罗》(Contra) 中角色为什么要「萝莉式屈腿俯卧」？

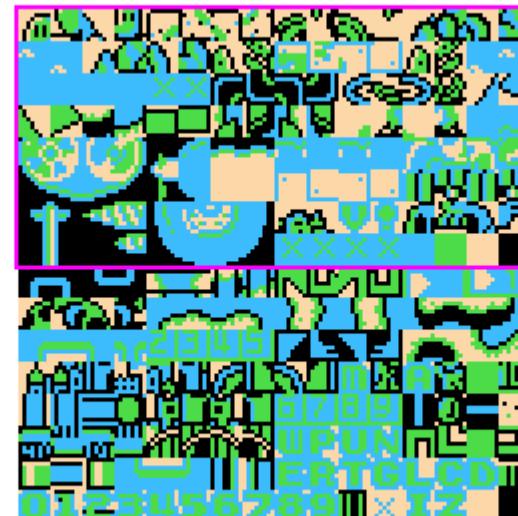


- 背景：“卷轴”操作



# 在受限的硬件下做游戏：背景动画

- 通过切换 tile pattern table 实现背景动画
  - 同时更新屏幕上的所有 tiles
  - 难怪为什么有些 “次世代” 的游戏画面那么精良
    - 更大的存储
    - 专有的硬件 (图形甚至整个计算系统)
      - 例子：[GUN-NAC](#) (1990)



**NEXOFT**

~GunNac~

**PLAYED BY Valls77**

**WWW.LONGPLAYS.ORG**

# 处理器-设备接口：NES实现（1983）

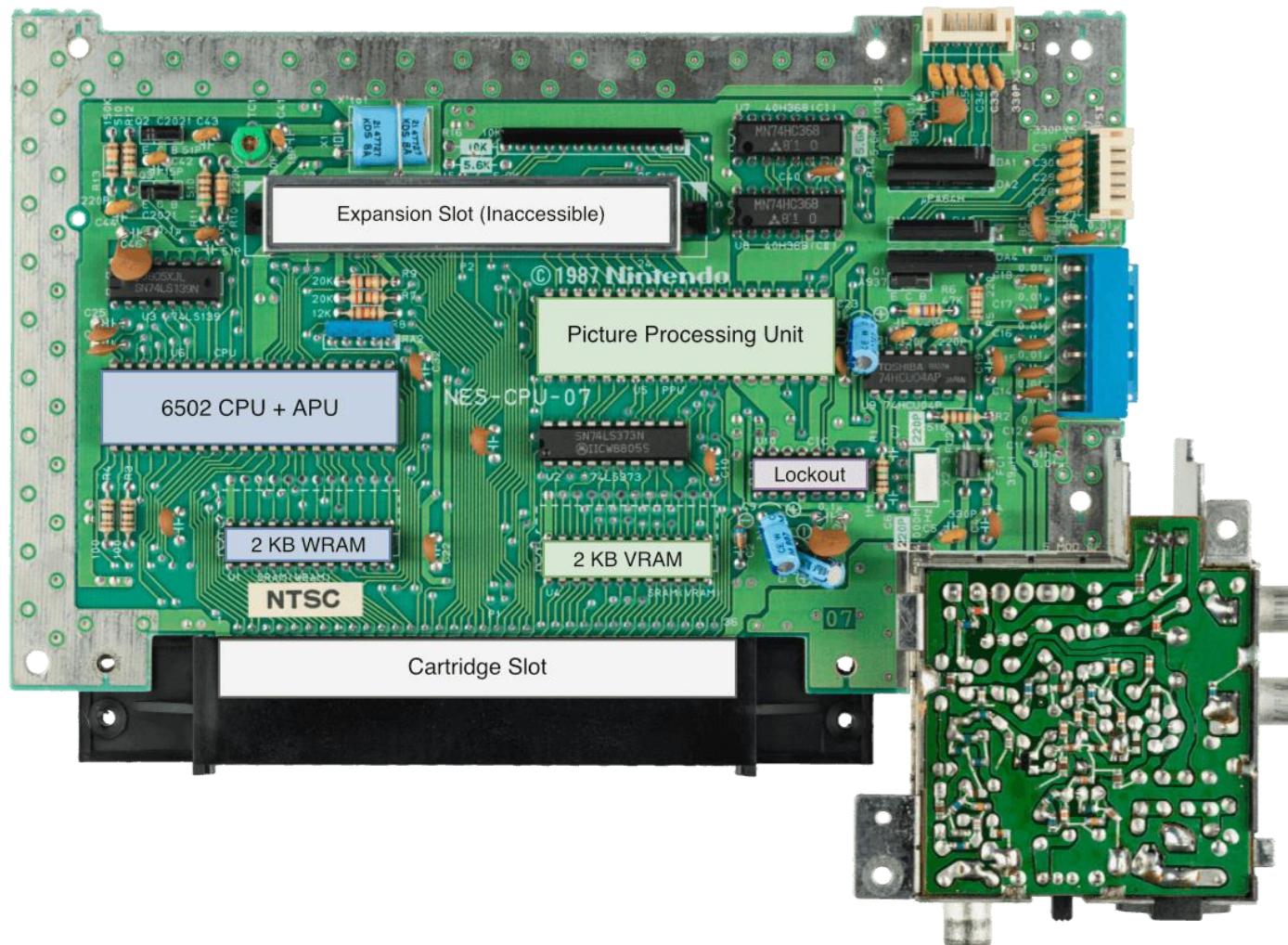
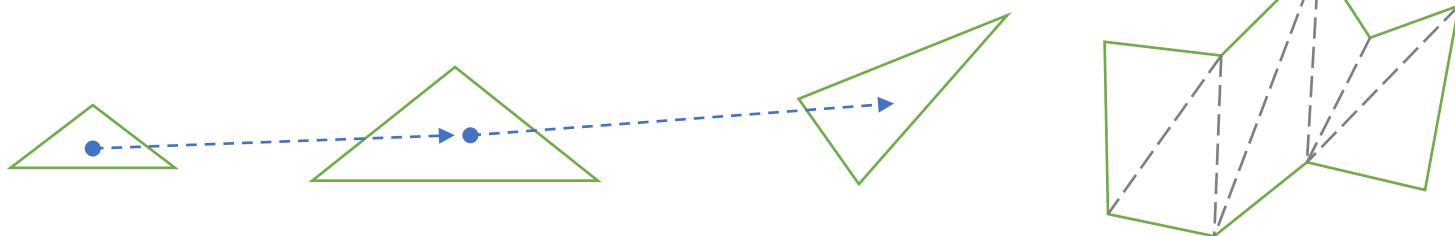


Image by [Rodrigo Copetti](#); 以及 [电路图](#)

# 3D图形绘制硬件

# 更好的 2D 游戏引擎

- 如果我们有更多的晶体管?
  - NES PPU 的本质是和坐标轴平行的 “贴块块”
    - 实现上只需要加法和位运算
    - 更强大的计算能力 = 更复杂的图形绘制
- 2D 图形加速硬件: 图片的 “裁剪” + “拼贴”
  - 支持旋转、材质映射(缩放)、后处理、.....



- 实现 3D
  - 三维空间中的多边形，在视平面上也是多边形
    - 任何  $n$  边形都可以分解成  $n-2$  个三角形

# 以假乱真的剪贴3D

- GameBoy Advance
  - 4 层背景; 128 个剪贴 objects; 32 个 affine objects
    - CPU 给出描述; GPU 绘制 (执行 “一个程序” 的 CPU)



([V-Rally](#); Game Boy Advance, 2002)



Licensed by  
NINTENDO



00:05 / 06:09



自动

倍速



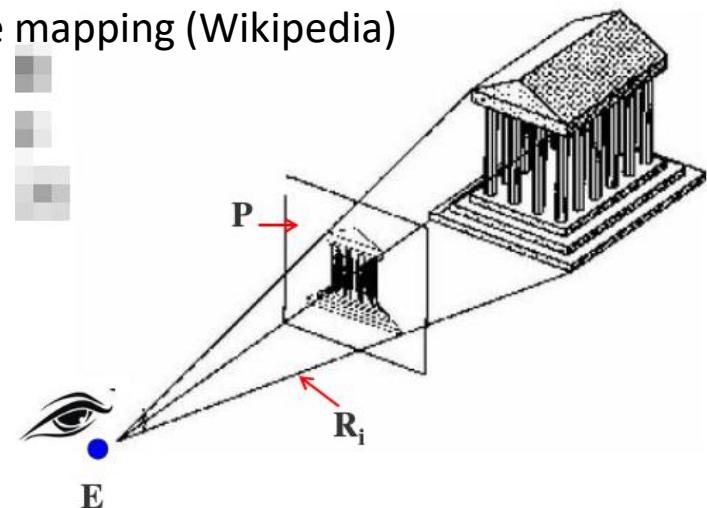
# 但我们还是需要真正的3D

- 三维空间中的三角形需要正确渲染



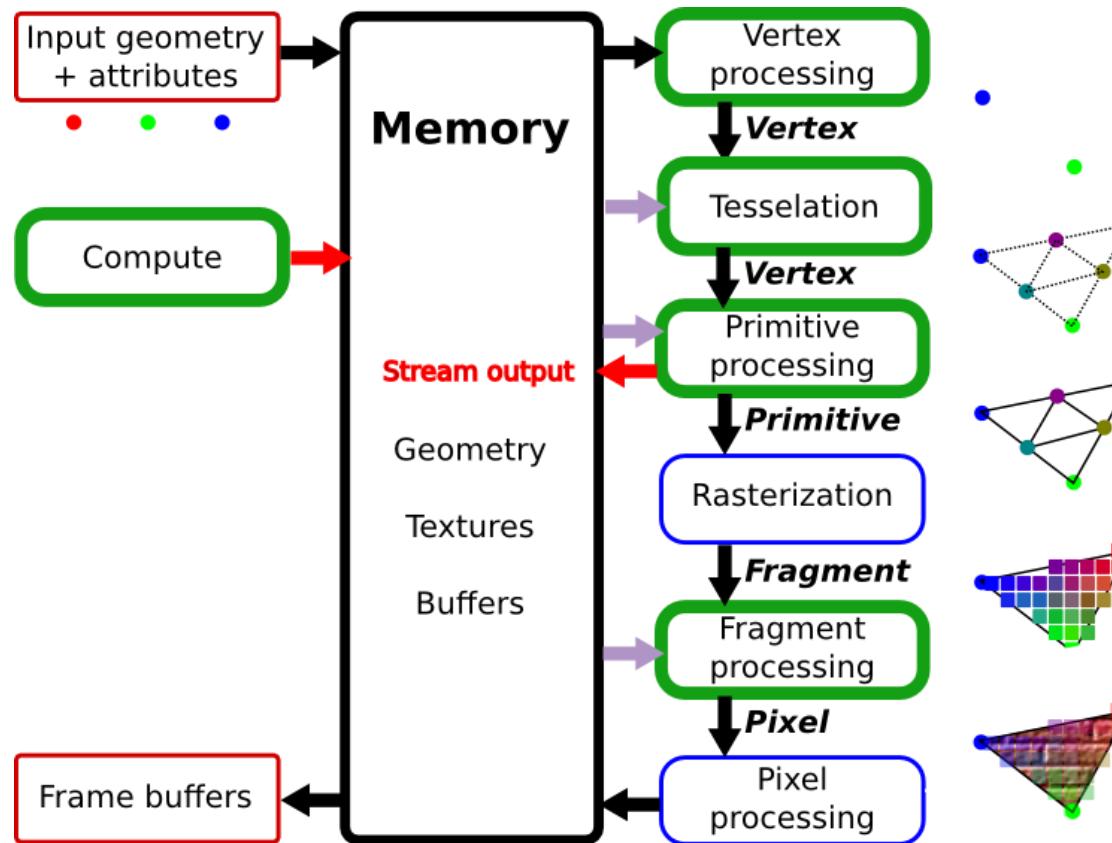
“Perspective correct” texture mapping (Wikipedia)

- 我们不要纸片人
  - 我们要真正的 3D
  - 要数百万个多边形构造的精细场景



# 现代 3D 图形

- CPU 负责描述，GPU 负责渲染
  - GPU 收到代码 + 数据 → 运算 → 写回结果到内存 → 发送中断
    - GPU 中有数千个运算单元实现并行的计算



# 理论：一切皆可 “计算”

```
for (int i = 1; i <= H; i++) {  
    for (int j = 1; j <= W; j++)  
        putchar(j <= i ? '*' : ' ');  
    putchar('\n');  
}
```

\* \* \*

f

- 难办的是性能

- NES: 6502 @ 1.79Mhz; IPC = 0.43
  - 屏幕共有  $256 \times 240 = 61K$  像素 (256 色)
  - 60FPS → 每一帧必须在 ~10K 条指令内完成
    - 如何在有限的 CPU 运算力下实现 60Hz?

# 现代 3D 图形：硬件上的实现

```
--global__  
void hello(char *a, char *b) {  
    a[threadIdx.x] += b[threadIdx.x];  
}  
  
char a[N] = "Hello ";  
char b[N] = {15, 10, 6, 0, -11, 1};  
cudaMalloc( (void**)&ad, N );  
cudaMalloc( (void**)&bd, N );  
cudaMemcpy( ad, a, N, cudaMemcpyHostToDevice );  
cudaMemcpy( bd, b, N, cudaMemcpyHostToDevice );  
printf("%s", a); // Hello  
dim3 dimBlock( blocksize, 1 );  
dim3 dimGrid( 1, 1 );  
hello<<<dimGrid, dimBlock>>>(ad, bd); // run on GPU  
cudaMemcpy( a, ad, N, cudaMemcpyDeviceToHost );  
printf("%s\n", a); // World!  
...
```

# 题外话：如此丰富的图形是怎么来的？



# 答案：全靠 PS (后处理)

- 例子：GLSL (Shading Language)
  - 使 “shader program” 可以在 GPU 上执行
    - 可以作用在各个渲染级别上：vertex, fragment, pixel shader
    - 相当于一个 “PS” 程序，算出每个部分的光照变化
      - 全局光照、反射、阴影、环境光遮罩……

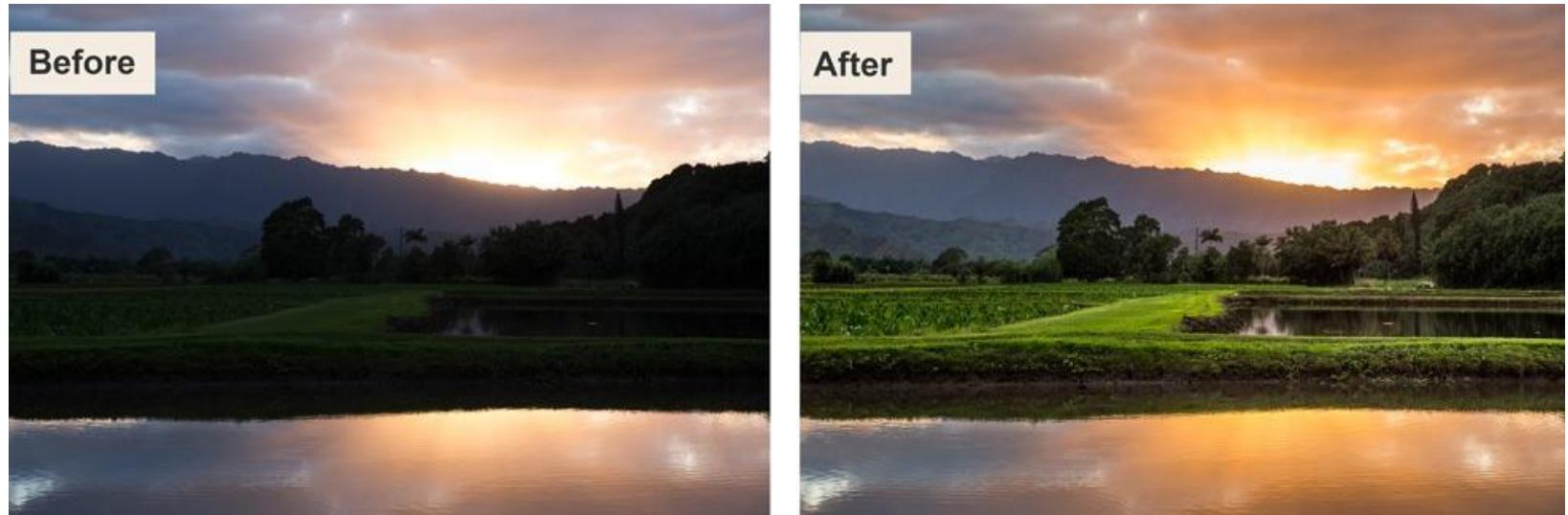


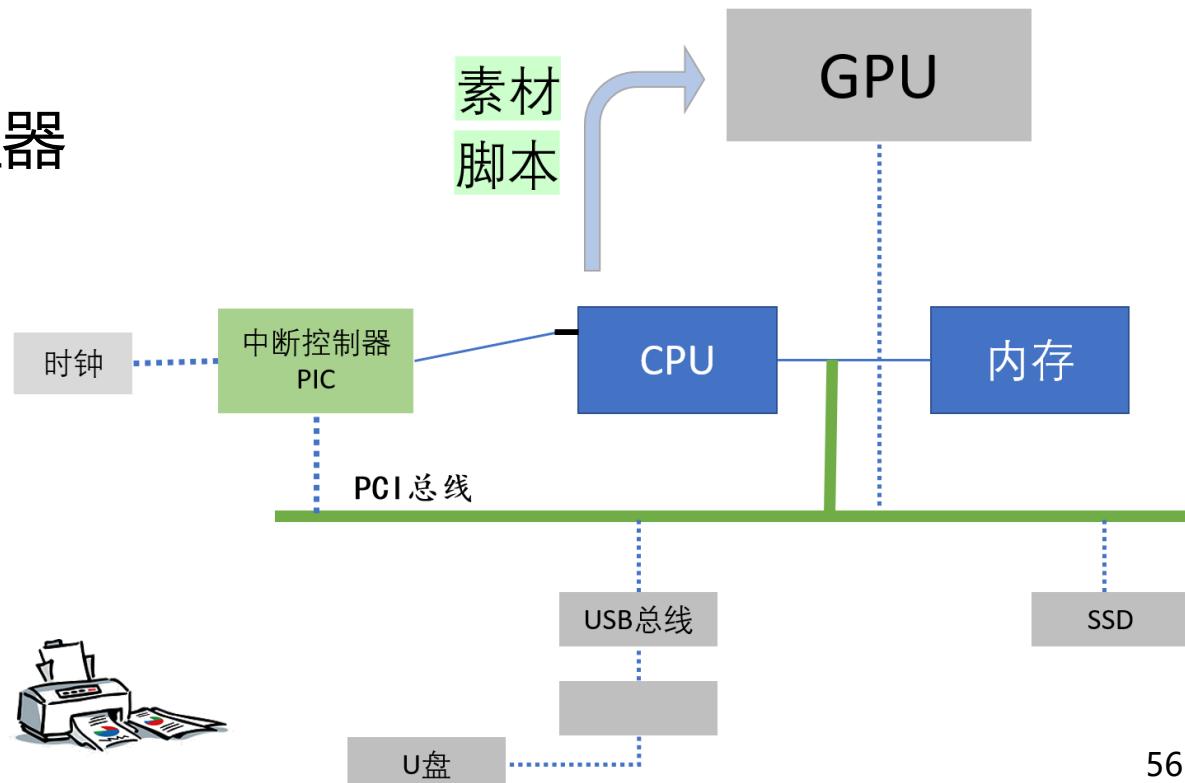
Photo by [Rob Lim](#)

# 总结

# I/O 设备

任何能与 CPU 交换数据的“东西”。

- 完成与物理世界的交互
  - 键盘、鼠标、打印机.....
- 效率更高的专用处理器
  - GPU, NPU, FPGA, ...



# 复杂系统的构造和解释

# 《计算机系统基础》到底学什么？

一句话的 take-away message: “计算机系统是一个状态机”。

- 更具体一点?

- 状态机的状态是什么?
  - 内存 + 寄存器 + (外部设备状态)
- 状态机的行为是什么?
  - 取指令 + 译码 + 执行
  - 输入/输出设备访问
  - 中断 + 异常控制流
  - 地址转换
- (是否感到有点驾驭不了? )

# 复杂中隐藏的秩序

- 理解/构造一个复杂系统 (操作系统/处理器/航母)?
  - USS Midway (CV-41); 1945 年首航, “沙漠风暴” 行动后退役
    - 舰船配置; 资源管理/调度; 容错.....



(picture: [www.seaforces.org](http://www.seaforces.org); [familyvacationhub.com](http://familyvacationhub.com))

# 复杂中隐藏的秩序 (cont'd)

航母不是一天造成的。



(picture: history.navy.mil)

# 复杂中隐藏的秩序 (cont'd)

采矿船继承了航海时代的设计。



(picture: history.navy.mil)

# 复杂中隐藏的秩序 (cont'd)

复杂系统的演化通常是*evolutionary*的而不是*revolutionary*的。

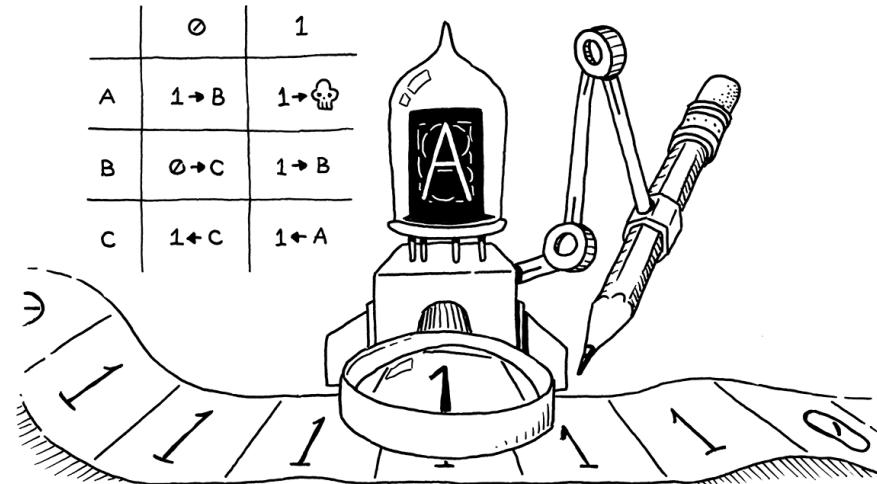
- 无法第一次就设计出“绝对完美”的复杂系统
  - (因为环境一直在变)
- 实际情况：从 minimal, simple, and usable 的系统不断经过 local modifications (trial and errors)
  - 计算机硬件
  - 操作系统
  - 编译器/程序设计语言
    - 需求和系统设计/实现螺旋式迭代

# 理解计算机系统

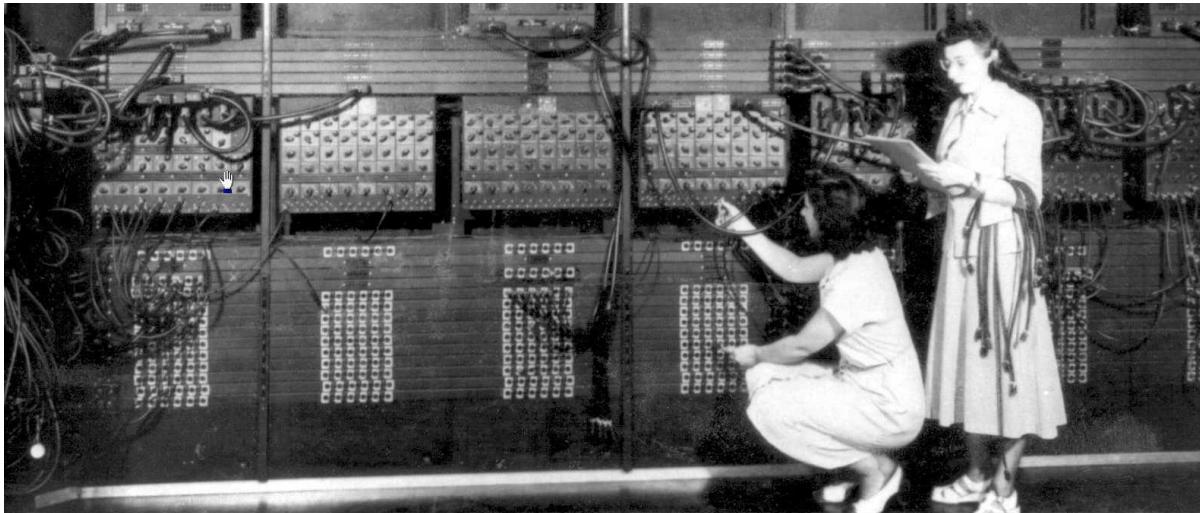
# 在计算机诞生之前

- Alan Turing's “machine” (1936)

- 并没有真正“造出来”
- 纸带+自动机
  - 纸带  $\text{map} < \text{int}, \text{int} \rangle \text{ mem}$
  - 读写头 pos
  - 自动机（程序）
    - （移动读写头）  $\text{pos}++, \text{pos}--$
    - （写内存）  $\text{mem[pos]} = 0, \text{mem[pos]} = 1$
    - （读内存）  $\text{if}(\text{mem[pos]}) \{ \} \text{ else} \{ \}$
    - （跳转）  $\text{goto label}$
    - （终止）  $\text{halt()}$



# ENIAC: 人类可用的Turing Machine



- [ENIAC Simulator by Brian L. Stuart](#)

- 20 word (not bit) memory
- 自带“寄存”的状态(寄存器)
- 支持算数运算
- 支持纸带(串行)输入输出



# von Neumann Machine: 存储程序控制

可以把状态机的状态保存在存储器里，而不要每次重新设置。

- 计算机的设计受到数字电路实现的制约
  - 执行一个动作(指令)只能访问有限数量的存储器
    - 常用的临时存储(寄存器；包括PC)
    - 更大的、编址的内存
  - (是不是想起了YEMU?)

# von Neumann Machine: 更多的I/O设备

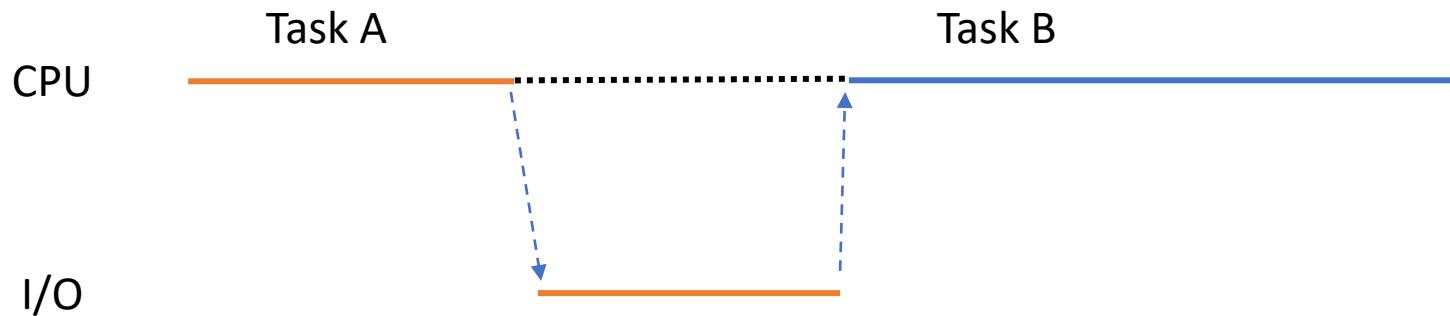
存储程序的通用性真正掀起了计算机走向全领域的革命。



- 只要增加 in 和 out 指令，就可以和物理世界建立无限的联系
  - 持久存储 (磁带、磁盘.....)
  - 读卡器、打印机.....

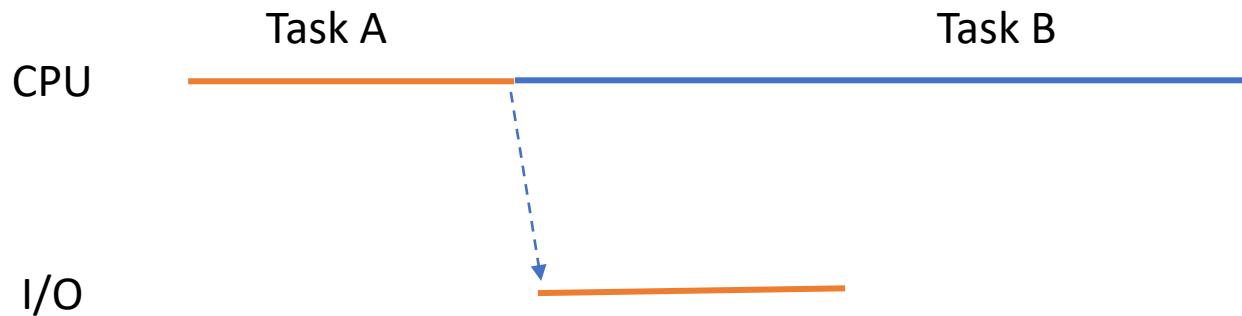
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



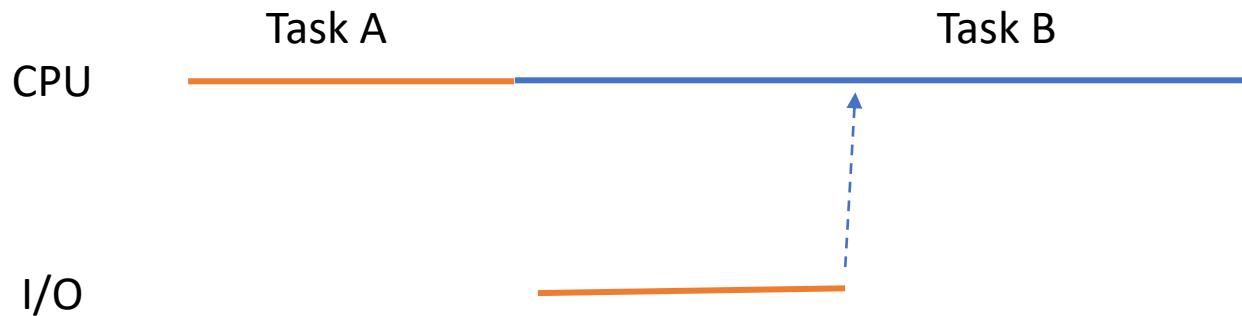
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动的函数调用
  - 相当于在每条语句后都插入

```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
  - 于是就有了最早的操作系统：管理I/O设备的库代码

# 中断 + 更大的内存 = 分时多线程

```
void foo() { while (1) printf("a"); }
void bar() { while (1) printf("b"); }
```

- 能够让foo()和bar()“同时”在处理器上执行?
  - 借助每条语句后被动插入的interrupt\_handler()调用

```
void interrupt_handler() {
    dump_regs(current->regs);
    current = (current->func == foo) ? bar : foo;
    restore_regs(current->regs);
}
```

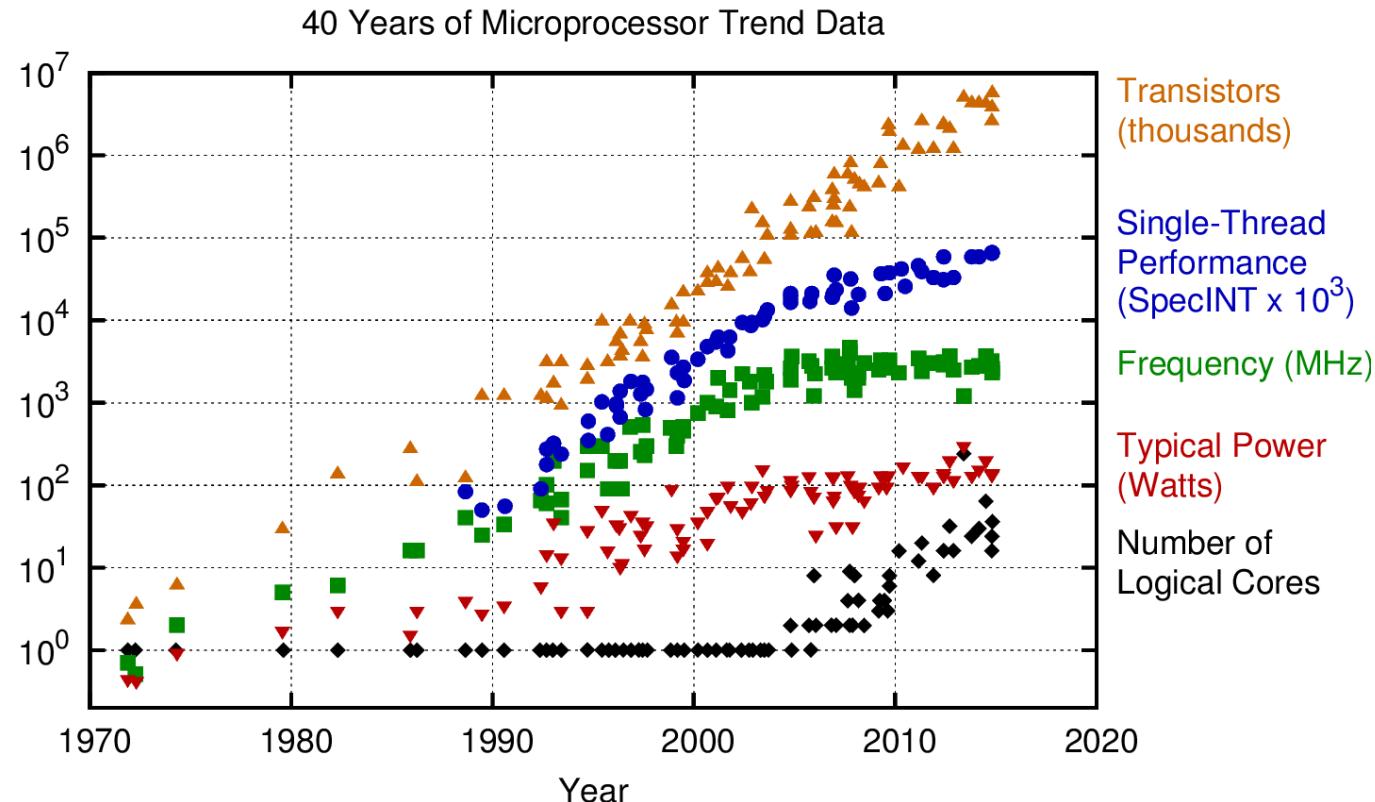
- 操作系统背负了“调度”的职责

# 分时多线程 + 虚拟存储 = 进程

- 让`foo()`和`bar()`的执行互相不受影响
  - 在计算机系统里增加映射函数 $f_{\text{foo}}, f_{\text{bar}}$
  - `foo`访问内存地址 $m$ 时，将被重定位到 $f_{\text{foo}}(m)$
  - `bar`访问内存地址 $m$ 时，将被重定位到 $f_{\text{bar}}(m)$
- `foo`和`bar`本身无权管理 $f$
- 操作系统需要完成进程、存储、文件的管理
- UNIX诞生（就是我们今天的进程；Android app；...）
  - `gcc a.c`
  - `readelf -a a.out` → 二进制文件的全部信息

# 故事其实没有停止.

- 面对有限的功耗、难以改进的制程、无法提升的频率、应用需求
  - 多处理器、big.LITTLE、异构处理器 (GPU、NPU、...)
  - 单指令多数据(MMX, SSE, AVX, ...), 虚拟化(VT; EL0/1/2/3), ..., 安全执行环节 (TrustZone; SGX), ...



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

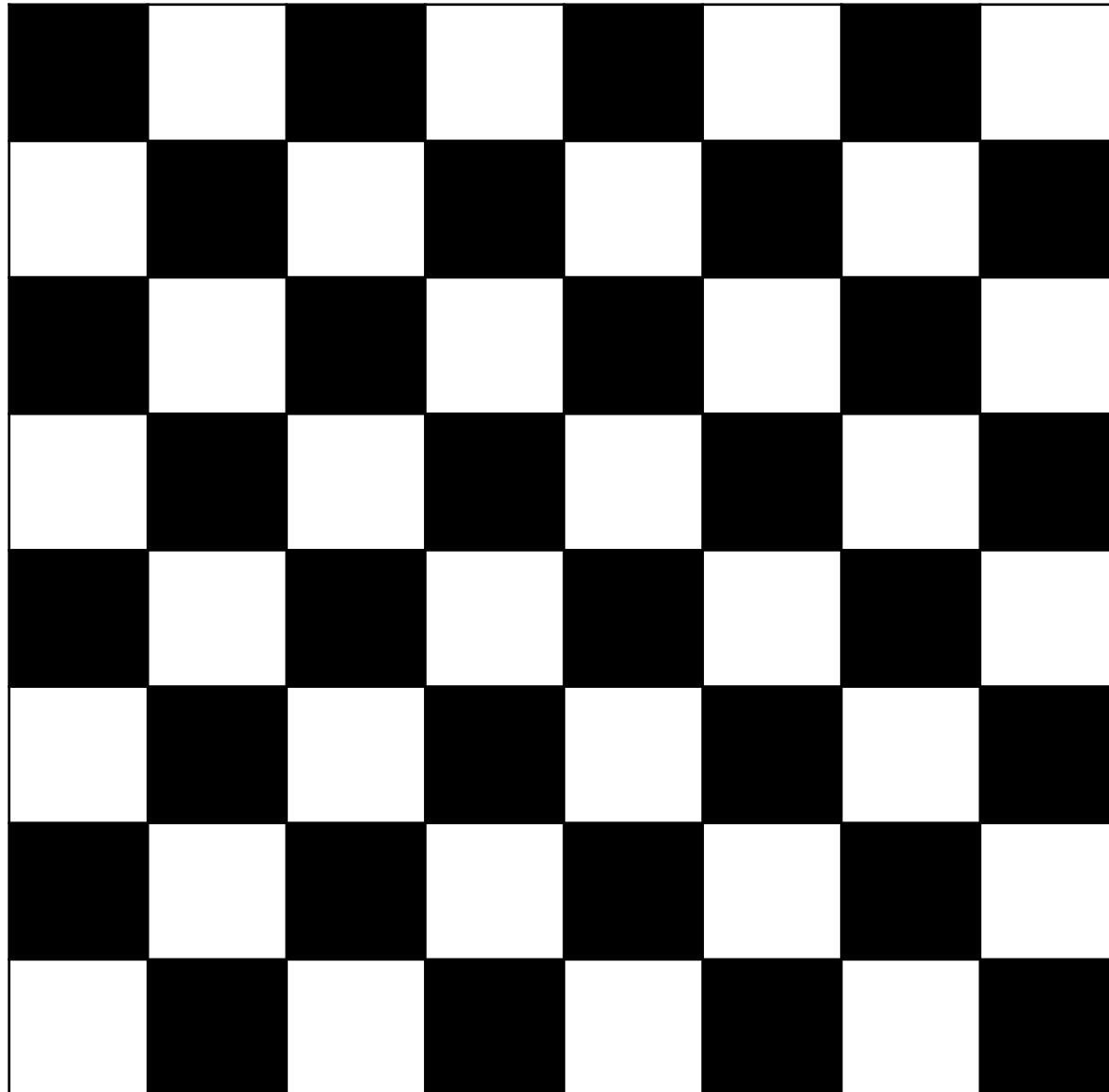
End.

(今天应该解开了很多同学对计算机专业的疑惑? )

GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 50 million developers.

```
git clone -b 2021 https://github.com/NJU-ProjectN/ics-pa ics2021  
git clone https://github.com/NJU-ProjectN/ics-workbench
```





# 链接与加载选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 提醒

PA2 Deadline 即将截止:

2023. 11. 19 23:59:59

推迟一周: 2023. 11. 26 23:59:59

Lab2 Deadline 即将截止:

2023. 11. 19 23:59:59

推迟一周: 2023. 11. 26 23:59:59

# 本讲概述

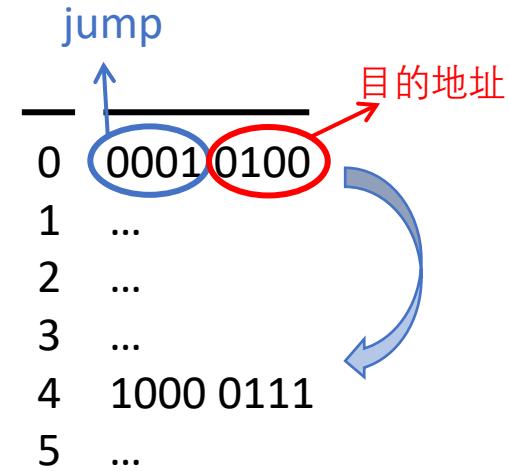
动态链接，大家明白了吗？  
(根据我们的经验，大家上课没听懂)

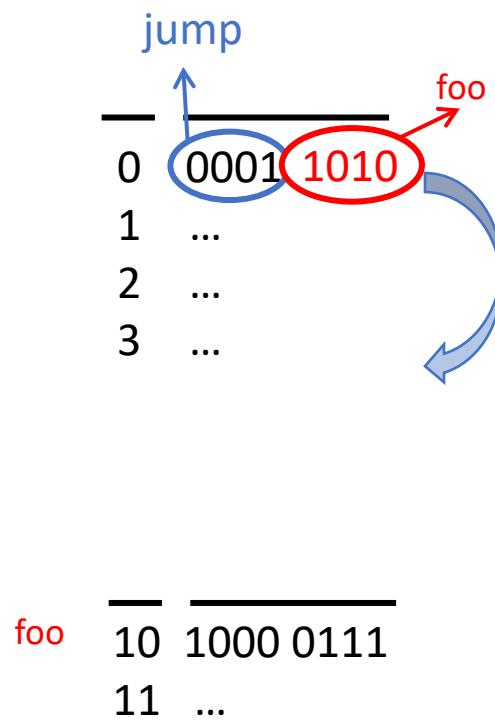
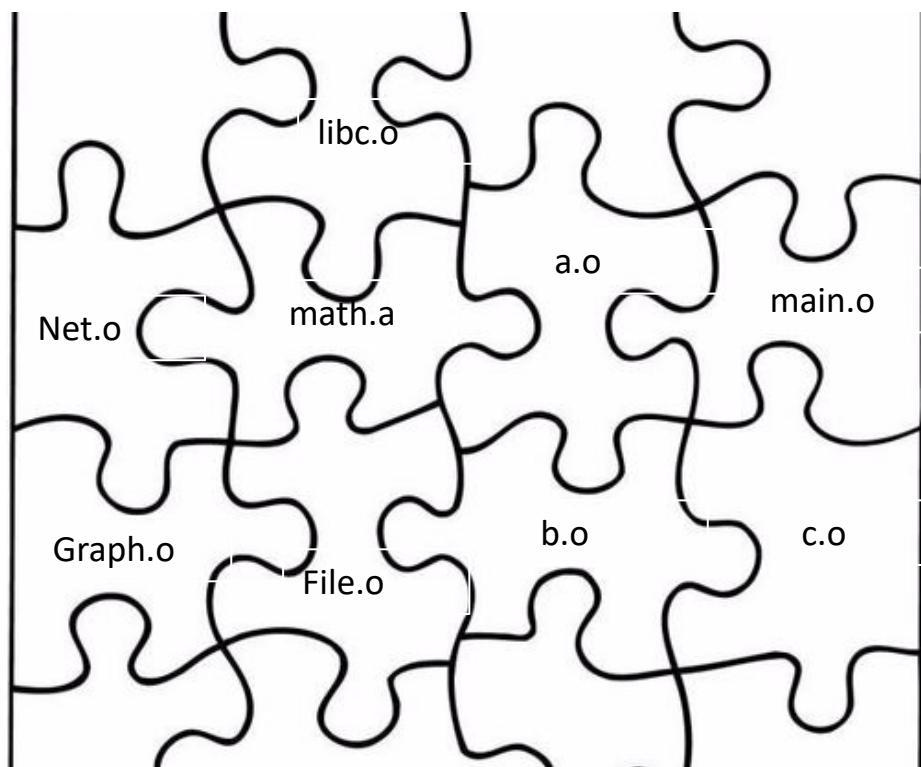
- 本讲内容

- 静态链接与加载
  - Hello 程序的链接与加载
- 动态链接与加载
  - 自己动手实现动态加载

# 静态链接与加载

# 什么是链接?





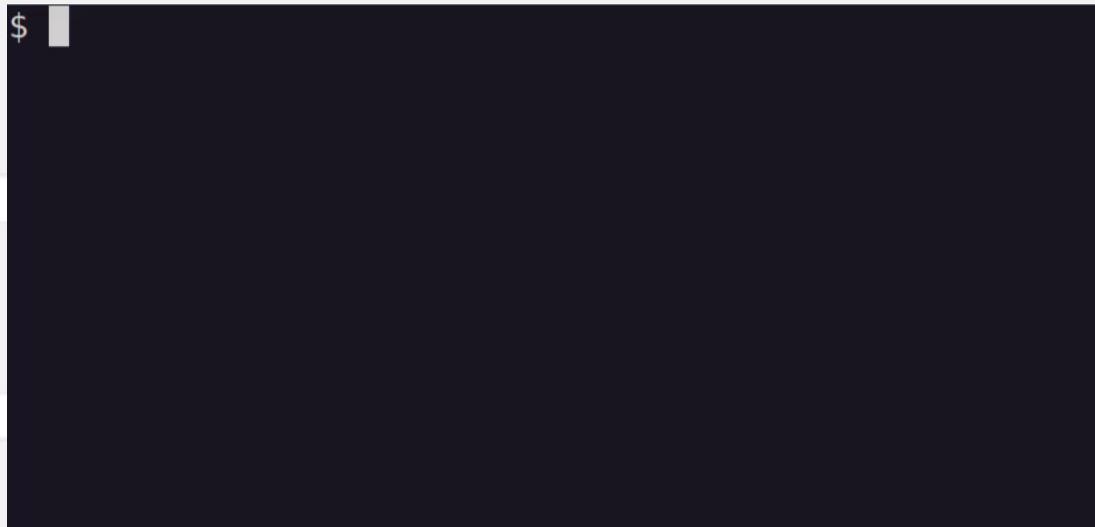
# 一个实验： -fno-pic编译； -static链接

- 代码

```
// a.c
int foo(int a, int b) {
    return a + b;
}
```

```
// b.c
int x = 100, y = 200;
```

```
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```



```
$ cat a.c
int foo(int a, int b) {
    return a + b;
}
```

```
$ cat b.c
int x = 100, y = 200;
```

```
$ cat main.c
#include <stdio.h>

extern int x, y;
int foo(int a, int b);

int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

```
1 CFLAGS := -O2 -fno-pic
2 LDFLAGS := -static
3
4 a.out: a.o b.o main.o
5     gcc $(LDFLAGS) a.o b.o main.o
6
7 a.o: a.c
8     gcc $(CFLAGS) -c a.c
9
10 b.o: b.c
11     gcc $(CFLAGS) -c b.c
12
13 main.o: main.c
14     gcc $(CFLAGS) -c main.c
15
16 clean:
17     rm -f *.o a.out
```

# 链接的a.o, b.o, main.o是什么？

- ELF relocatable

```
$ file a.o
a.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$ file b.o
b.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$ file main.o
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

# ELF文件

ELF文件类型	说明	实例
可重定位文件 (Relocatable File)	这类文件包括代码和数据，可以被用来链接成可执行文件或共享目标文件，静态链接库也可归为这一类	Linux的.o Windows的.obj
可执行文件 (Executable File)	这类文件包含了可以直接执行的程序，它的代表就是ELF可执行文件	/bin/bash, a.out Windows的.exe
共享目标文件 (Shared Object File)	这种文件包含了代码和数据，可以在下面两种情况下使用： <ol style="list-style-type: none"><li>1. 链接器使用此文件和其他可重定位文件和共享目标文件链接，产生新的目标文件；</li><li>2. 动态链接器将几种共享目标文件与可执行文件结合，作为进程映像的一部分运行</li></ol>	Linux的.so (e.g., /lib/glibc-2.5.so) Windows的DLL
核心转储文件 (Core Dump File)	当进程意外终止时，系统可以将该进程地址空间的内容及终止时的一些信息转储到此	Linux下的core dump

# 可执行文件格式之EL

```
int global_int_var = 84;
```

```
int global_int_var2;
```

```
void func1(int i){  
    printf("%d\n", i);  
}  
int main(void){
```

```
    static int static_var = 85;
```

```
    static int static_var2;
```

```
    int a = 1;  
    int b;  
    func1( static_var + static_var2  
    + a + b);  
    return 0;  
}
```

-fno-zero-initialized-in-bss

If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section—e.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

The default is ‘-fzero-initialized-in-bss’.

File Header

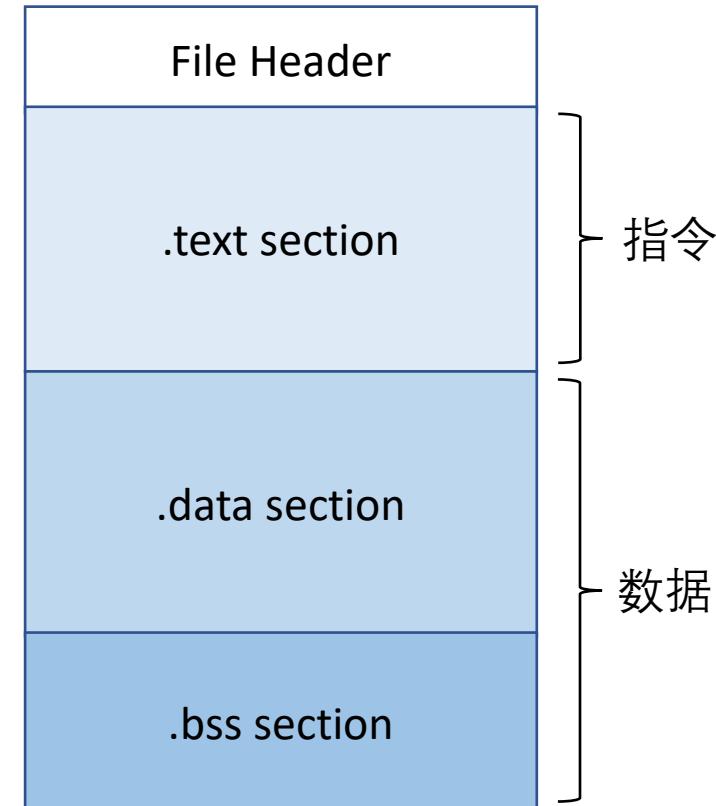
.text section

.data section

.bss section

# 小问题？

- 为什么要把程序指令和数据分开放？而不放在一个段中？
  - 权限隔离
  - 缓存命中
  - 共享内存



# GNU Binutils

- [Binary utilities](#): 分析二进制文件的工具
  - RTFM: 原来有那么多工具!
    - 有 addr2line, 自己也可以实现类 gdb 调试器了
- 使用binutils
  - objdump查看.data节的x, y (-D)
  - objdump查看main对应的汇编代码
  - readelf查看relocation信息
    - 思考: 为什么要-4?

## GNU Binutils

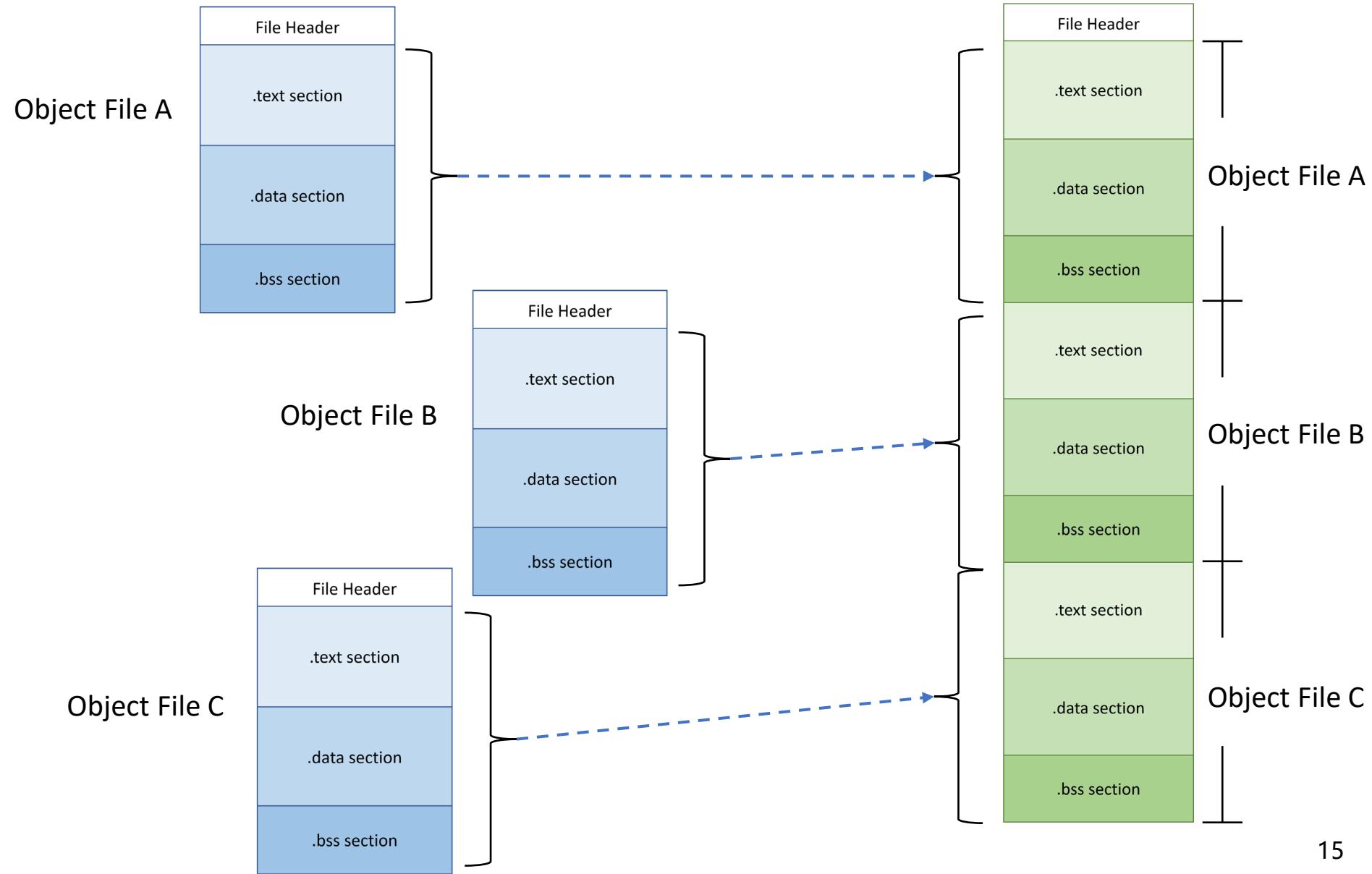
The GNU Binutils are a collection of binary tools. The main ones are:

- **ld** - the GNU linker.
- **as** - the GNU assembler.

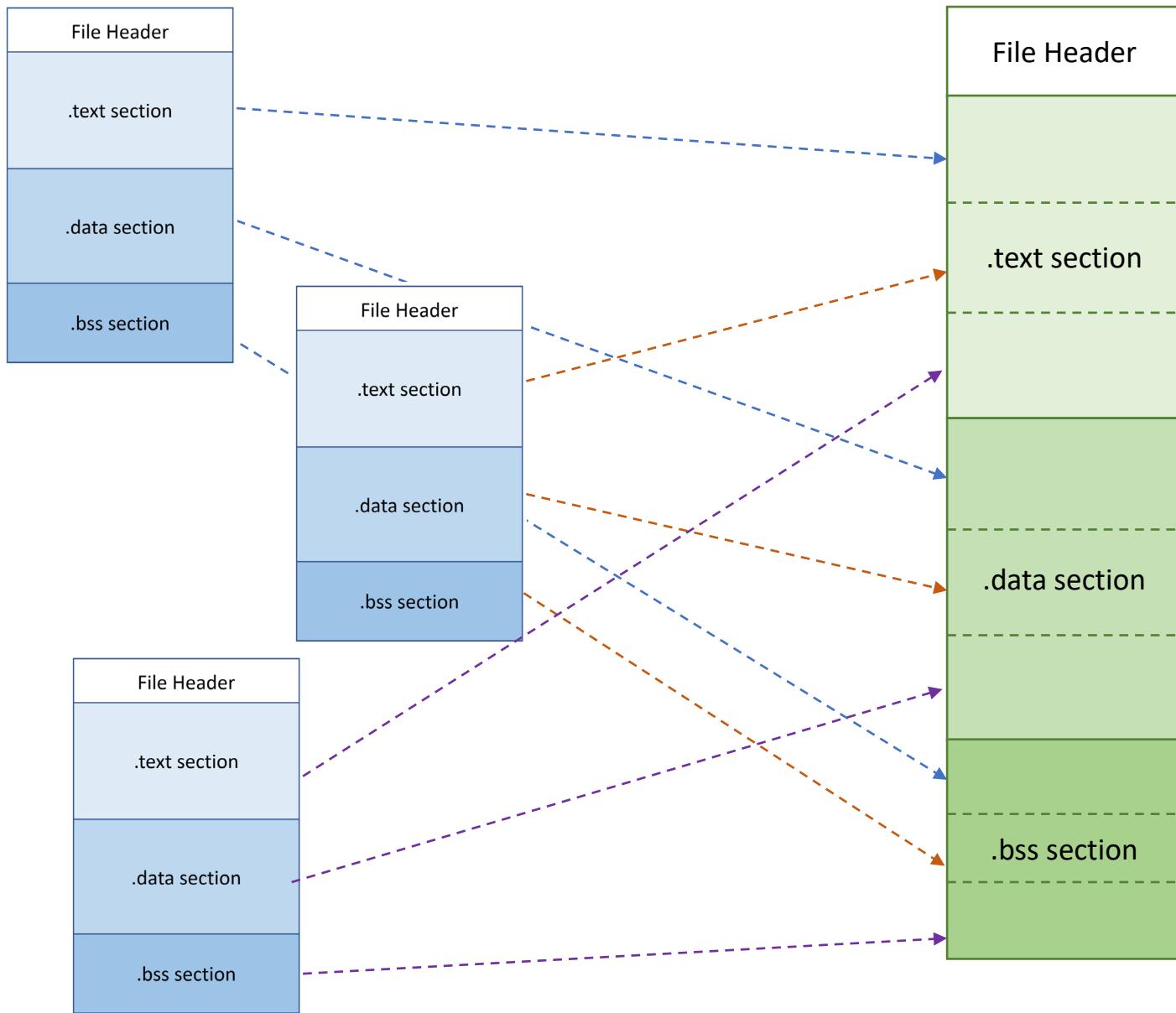
But they also include:

- **addr2line** - Converts addresses into filenames and line numbers.
- **ar** - A utility for creating, modifying and extracting from archives.
- **c++filt** - Filter to demangle encoded C++ symbols.
- **dlltool** - Creates files for building and using DLLs.
- **gold** - A new, faster, ELF only linker, still in beta test.
- **gprof** - Displays profiling information.
- **nlmconv** - Converts object code into an NLM.
- **nm** - Lists symbols from object files.
- **objcopy** - Copies and translates object files.
- **objdump** - Displays information from object files.
- **ranlib** - Generates an index to the contents of an archive.
- **readelf** - Displays information from any ELF format object file.
- **size** - Lists the section sizes of an object or archive file.
- **strings** - Lists printable strings from files.
- **strip** - Discards symbols.
- **windmc** - A Windows compatible message compiler.
- **windres** - A compiler for Windows resource files.

# 链接多个.O



# 链接多个.O



# 一个实验： -c编译； ld链接

- 简化一点 (不要printf)

```
// a.c
int foo(int a, int b) {
    return a + b;
}
```

```
// b.c
int x = 100, y = 200;
```

```
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    foo(x, y);
    //printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

# Trv一下

管理 控制 视图 热键 设备 帮助

\$

File Header

0x3c

.text section

0x18

0x00

0x24

0x08

.data section

0x00

0x08

0x00

0x00

.bss section

0x00

0x00

0x00

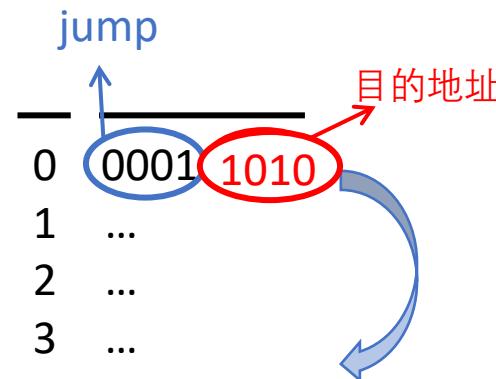
0x00

18



# 链接 Two-pass linking

- 空间和地址的分配
  - 重新建立符号表，合并段，并计算段长度建立映射关系
- 符号解析和重定位
  - 如何填空？



— —  
10 1000 0111  
11 ...

# 一个实验： -fno-pic编译； -static链接

- 代码

```
// a.c
int foo(int a, int b) {
    return a + b;
}
```

```
// b.c
int x = 100, y = 200;
```

```
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

```
$ ls  
a.c  a.o  a.out  b.c  b.o  main.c  main.o  Makefile  
$ █
```

S 英 ∙ · 简 拼 \*

2 0 1 2 3 4 5 6 7 8 9 Right Shift + Right Alt

# a.c / a.o

```
// a.c
int foo(int a, int b) {
    return a + b;
}
```

```
Disassembly of section .text:
0000000000000000 <foo>:
 0:   f3 0f 1e fa          endbr64
 4:   8d 04 37          lea    (%rdi,%rsi,1),%eax
 7:   c3                ret
```

# b.c / b.o

```
// b.c  
int x = 100, y = 200;
```

```
Disassembly of section .data:  
  
0000000000000000 <y>:  
 0: c8 00 00 00          enter $0x0,$0x0  
  
0000000000000004 <x>:  
 4: 64 00 00          add    %al,%fs:(%rax)  
 ...
```

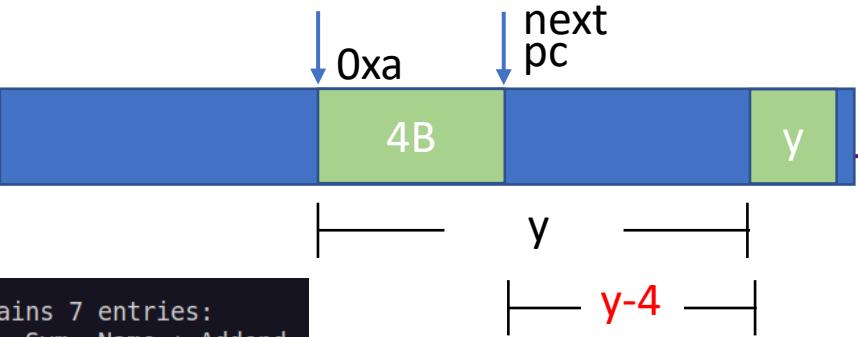
# main.c / main.o

```
// main.c
extern int x, y;
int foo(int a, int b); // 可以试试 extern int foo;
int main() {
    printf("%d + %d = %d\n", x, y, foo(x, y));
}
```

Disassembly of section .text.startup:

```
0000000000000000 <main>:
 0:   f3 0f 1e fa          endbr64
 4:   48 83 ec 08          sub    $0x8,%rsp
 8:   8b 35 00 00 00 00    mov    0x0(%rip),%esi      # e <main+0xe>
 e:   8b 3d 00 00 00 00    mov    0x0(%rip),%edi      # 14 <main+0x14>
14:  e8 00 00 00 00        call   19 <main+0x19>
19:  8b 0d 00 00 00 00    mov    0x0(%rip),%ecx      # 1f <main+0x1f>
1f:  8b 15 00 00 00 00    mov    0x0(%rip),%edx      # 25 <main+0x25>
25:  be 00 00 00 00        mov    $0x0,%esi
2a:  41 89 c0              mov    %eax,%r8d
2d:  bf 01 00 00 00        mov    $0x1,%edi
32:  31 c0                xor    %eax,%eax
34:  e8 00 00 00 00        call   39 <main+0x39>
39:  31 c0                xor    %eax,%eax
3b:  48 83 c4 08          add    $0x8,%rsp
3f:  c3                   ret
```

# 填什么？为什么y - 4？

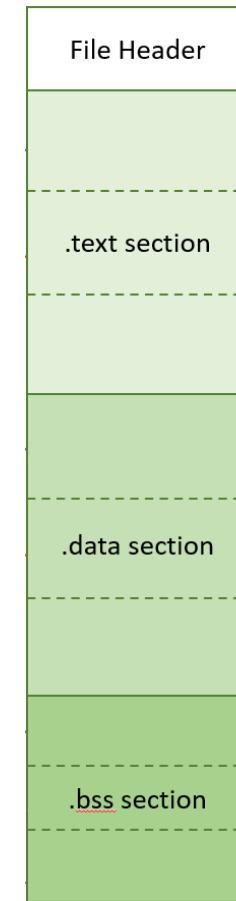


- readelf –a main.o

```
Relocation section '.rela.text.startup' at offset 0x208 contains 7 entries:
  Offset          Info           Type            Sym. Value   Sym. Name + Addend
000000000000000a  000500000002 R_X86_64_PC32    0000000000000000 y - 4
00000000000010  000600000002 R_X86_64_PC32    0000000000000000 x - 4
00000000000015  000700000004 R_X86_64_PLT32   0000000000000000 foo - 4
0000000000001b  000500000002 R_X86_64_PC32    0000000000000000 y - 4
00000000000021  000600000002 R_X86_64_PC32    0000000000000000 x - 4
00000000000026  00020000000a R_X86_64_32      0000000000000000 .rodata.str1.1 + 0
00000000000035  000800000004 R_X86_64_PLT32   0000000000000000 __printf_chk - 4
```

- objdump –d main.o

```
Disassembly of section .text.startup:
0000000000000000 <main>:
 0: f3 0f 1e fa        endbr64
 4: 48 83 ec 08        sub    $0x8,%rsp
 8: 8b 35 00 00 00 00  mov    0x0(%rip),%esi      # e <main+0xe>
 e: 8b 3d 00 00 00 00  mov    0x0(%rip),%edi      # 14 <main+0x14>
14: e8 00 00 00 00      call   19 <main+0x19>
19: 8b 0d 00 00 00 00  mov    0x0(%rip),%ecx      # 1f <main+0x1f>
1f: 8b 15 00 00 00 00  mov    0x0(%rip),%edx      # 25 <main+0x25>
25: be 00 00 00 00      mov    $0x0,%esi
2a: 41 89 c0          mov    %eax,%r8d
2d: bf 01 00 00 00      mov    $0x1,%edi
32: 31 c0             xor    %eax,%eax
34: e8 00 00 00 00      call   39 <main+0x39>
39: 31 c0             xor    %eax,%eax
3b: 48 83 c4 08        add    $0x8,%rsp
3f: c3                  ret
```



# 填什么？

- objdump -d a.out

```
00000000000401790 <main>:  
 401790: f3 0f 1e fa          endbr64  
 401794: 48 83 ec 08          sub    $0x8,%rsp  
 401798: 8b 35 52 89 0b 00      mov    0xb8952(%rip),%esi      # 4ba0f0 <y>  
 40179e: 8b 3d 50 89 0b 00      mov    0xb8950(%rip),%edi      # 4ba0f4 <x>  
 4017a4: e8 57 01 00 00          call   401900 <foo>  
 4017a9: 8b 0d 41 89 0b 00      mov    0xb8941(%rip),%ecx      # 4ba0f0 <y>  
 4017af: 8b 15 3f 89 0b 00      mov    0xb893f(%rip),%edx      # 4ba0f4 <x>  
 4017b5: be 04 e0 48 00          mov    $0x48e004,%esi  
 4017ba: 41 89 c0              mov    %eax,%r8d  
 4017bd: bf 01 00 00 00          mov    $0x1,%edi  
 4017c2: 31 c0              xor    %eax,%eax  
 4017c4: e8 47 3e 04 00          call   445610 <__printf_chk>  
 4017c9: 31 c0              xor    %eax,%eax  
 4017cb: 48 83 c4 08          add    $0x8,%rsp  
 4017cf: c3                  ret
```

# 填什么？

```
00000000000401790 <main>:  
401790: f3 0f 1e fa          endbr64  
401794: 48 83 ec 08          sub    $0x8,%rsp  
401798: 8b 35 52 89 0b 00      mov    0xb8952(%rip),%esi      # 4ba0f0 <y>  
40179e: 8b 3d 50 89 0b 00      mov    0xb8950(%rip),%edi      # 4ba0f4 <x>  
4017a4: e8 57 01 00 00          call   401900 <foo>  
4017a9: 8b 0d 41 89 0b 00      mov    0xb8941(%rip),%ecx      # 4ba0f0 <y>  
4017at: 8b 15 3f 89 0b 00      mov    0xb893f(%rip),%edx      # 4ba0f4 <x>  
4017b5: be 04 e0 48 00          mov    $0x48e004,%esi  
4017ba: 41 89 c0              mov    %eax,%r8d  
4017bd: bf 01 00 00 00          mov    $0x1,%edi  
4017c2: 31 c0              xor    %eax,%eax  
4017c4: e8 47 3e 04 00          call   445610 <__printf_chk>  
4017c9: 31 c0              xor    %eax,%eax  
4017cb: 48 83 c4 08          add    $0x8,%rsp  
4017cf: c3                  ret
```

```
00000000000401900 <foo>:  
401900: f3 0f 1e fa          endbr64  
401904: 8d 04 37              lea    (%rdi,%rsi,1),%eax  
401907: c3                  ret  
401908: 0f 1f 84 00 00 00 00      nopl   0x0(%rax,%rax,1)  
40190f: 00
```

管理 控制 视图 热键 设备 帮助

\$

S 英 拼 简 拼 章

Right Shift + Right Alt  
20

# 谁来加载a.out?

管理 控制 视图 热键 设备 帮助

```
0000000000401810 <main>:
401810: f3 0f 1e fa        endbr64
401814: 48 83 ec 08        sub    $0x8,%rsp
401818: bf a0 17 40 00      mov    $0x4017a0,%edi
40181d: e8 de 96 00 00      call   40af00 <atexit>
401822: 8b 35 c8 88 0b 00    mov    0xb88c8(%rip),%esi      # 4ba0
f0 <y>
401828: 8b 3d c6 88 0b 00    mov    0xb88c6(%rip),%edi      # 4ba0
f4 <x>
40182e: e8 5d 01 00 00      call   401990 <foo>
401833: 8b 0d b7 88 0b 00    mov    0xb88b7(%rip),%ecx      # 4ba0
f0 <y>
401839: 8b 15 b5 88 0b 00    mov    0xb88b5(%rip),%edx      # 4ba0
f4 <x>
40183f: be 10 e0 48 00      mov    $0x48e010,%esi
401844: 41 89 c0            mov    %eax,%r8d
401847: bf 01 00 00 00      mov    $0x1,%edi
40184c: 31 c0                xor    %eax,%eax
40184e: e8 4d 40 04 00      call   4458a0 <__printf_chk>
401853: 31 c0                xor    %eax,%eax
401855: 48 83 c4 08          add    $0x8,%rsp
401859: c3                  ret
40185a: 66 0f 1f 44 00 00    nopw   0x0(%rax,%rax,1)
:
```

S 英 · · 简 拼 \*

# 静态程序的加载

由操作系统加载（下学期内容）

- 你可以认为是“一步到位”的
  - 使用gdb (starti)调试
  - 使用strace查看系统调用序列

```
$ objdump -d a.out | less
$ readelf -h a.out
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - GNU
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x401860
  Start of program headers: 64 (bytes into file)
  Start of section headers: 844280 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 10
  Size of section headers: 64 (bytes)
  Number of section headers: 32
  Section header string table index: 21
```

World

```
$ objdump -d a.out | less
$ readelf -h a.out
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - GNU
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
Entry point address: 0x401860
Start of program headers: 64 (bytes into file)
Start of section headers: 844280 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 10
Size of section headers: 64 (bytes)
Number of section headers: 32
Section header string table index: 31
```

\$

S 英 单 简 拼 \*

# strace

```
$ make clean
rm -f *.o a.out
$ make
gcc -O2 -fno-pic -c a.c
gcc -O2 -fno-pic -c b.c
gcc -O2 -fno-pic -c main.c
gcc -static a.o b.o main.o
$ ./a.out
100 + 200 = 300
$ strace ./a.out
execve("./a.out", ["../a.out"], 0x7ffcc0649630 /* 60 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe0d8c6370) = -1 EINVAL (Invalid argument)
brk(NULL)                      = 0x1dad000
brk(0x1dadd80)                 = 0x1dadd80
arch_prctl(ARCH_SET_FS, 0x1dad380) = 0
uname({sysname="Linux", nodename="why-VirtualBox", ...}) = 0
readlink("/proc/self/exe", "/home/why/Documents/ICS2021/teac"..., 4096) = 49
brk(0x1dc000)                   = 0x1dc000
brk(0x1dcf000)                 = 0x1dcf000
mprotect(0x4b6000, 16384, PROT_READ) = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "100 + 200 = 300\n", 16100 + 200 = 300
)      = 16
exit_group(0)                  = ?
+++ exited with 0 +++
```

I

# 一个有趣的问题：“最小”的可执行代码

不能用ld链接么？

- (试一试)
  - 我们能否写一个最小的汇编代码，能正确返回？
  - 仅执行一个操作系统调用
    - `rax = 231; rdi = 返回值`
    - `syscall`指令执行系统调用

\$ vim as.S



# gcc和ld链接

不能用ld链接么？

- man gcc

```
to write -Xlinker "-assert definitions", because this passes the entire string as a single argument, which is not what the linker expects.
```

When using the GNU linker, it is usually more convenient to pass arguments to linker options using the option=value syntax than as separate arguments. For example, you can specify **-Xlinker -Map=output.map** rather than **-Xlinker -Map -Xlinker output.map**. Other linkers may not support this syntax for command-line options.

#### **-Wl,option**

Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, **-Wl,-Map,output.map** passes **-Map output.map** to the linker. When using the GNU linker, you can also get the same effect with **-Wl,-Map=output.map**.

NOTE: In Ubuntu 8.10 and later versions, for LDFLAGS, the option **-Wl,-z,relro** is used. To disable, use **-Wl,-z,norelro**.

#### **-u symbol**

Pretend the symbol symbol is undefined, to force linking of library modules to define it. You can use **-u** multiple times with different symbols to force loading of additional library modules.

#### **-z keyword**

**-z** is passed directly on to the linker along with the keyword keyword. See the section in the documentation of your linker for permitted values and their meanings.

#### **Options for Directory Search**

These options specify directories to search for header files, for libraries and for parts of the compiler:

```
-I dir  
-isystem dir
```



\$ |

|

S 英 单 简 拼

# 一些常见的链接库

```
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)unordtf2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)letf2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a)sfp-exceptions.o
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-dw2.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-dw2-fde-dip.o
(/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a)unwind-c.o
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)dl-iterateph
dr.o
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc.a
/usr/lib/gcc/x86_64-linux-gnu/10/libgcc_eh.a
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a
attempt to open /usr/lib/gcc/x86_64-linux-gnu/10/crtend.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/10/crtend.o
attempt to open /usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/crtn
.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/crtn.o
```

# Linux/C世界的宝藏

---

- gcc -Wl,--verbose
- C世界里还有很多大家不知道的东西
  - 一系列crt对象
    - \_\_attribute\_\_((constructor))
    - \_\_attribute\_\_((destructor)) (atexit)
- RTFM; RTFSC!

管理 控制 视图 热键 设备 帮助

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 extern int x, y;
5 int foo(int a, int b);
6
7 __attribute__((constructor)) void a(){
8     printf("Hello\n");
9 }
10 __attribute__((destructor)) void b(){
11     printf("World\n");
12 }
13
14 int main() {
15     printf("%d + %d = %d\n", x, y, foo(x, y));
16 }
17 }
```

~/Documents/ICS2021/teach/link-test/main.c[+1]  
-- INSERT --

[c] unix utf-8 Ln 14, Col 13/17

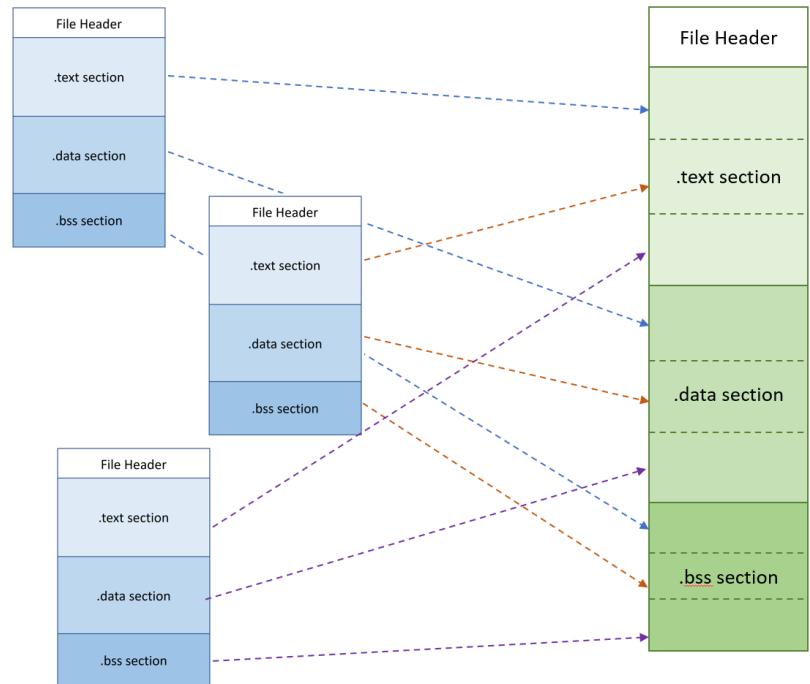
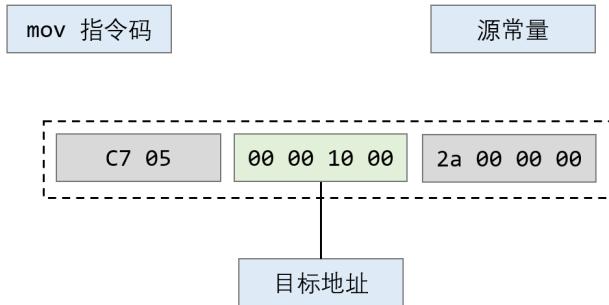


Right Shift + Right Alt

# Libc.a

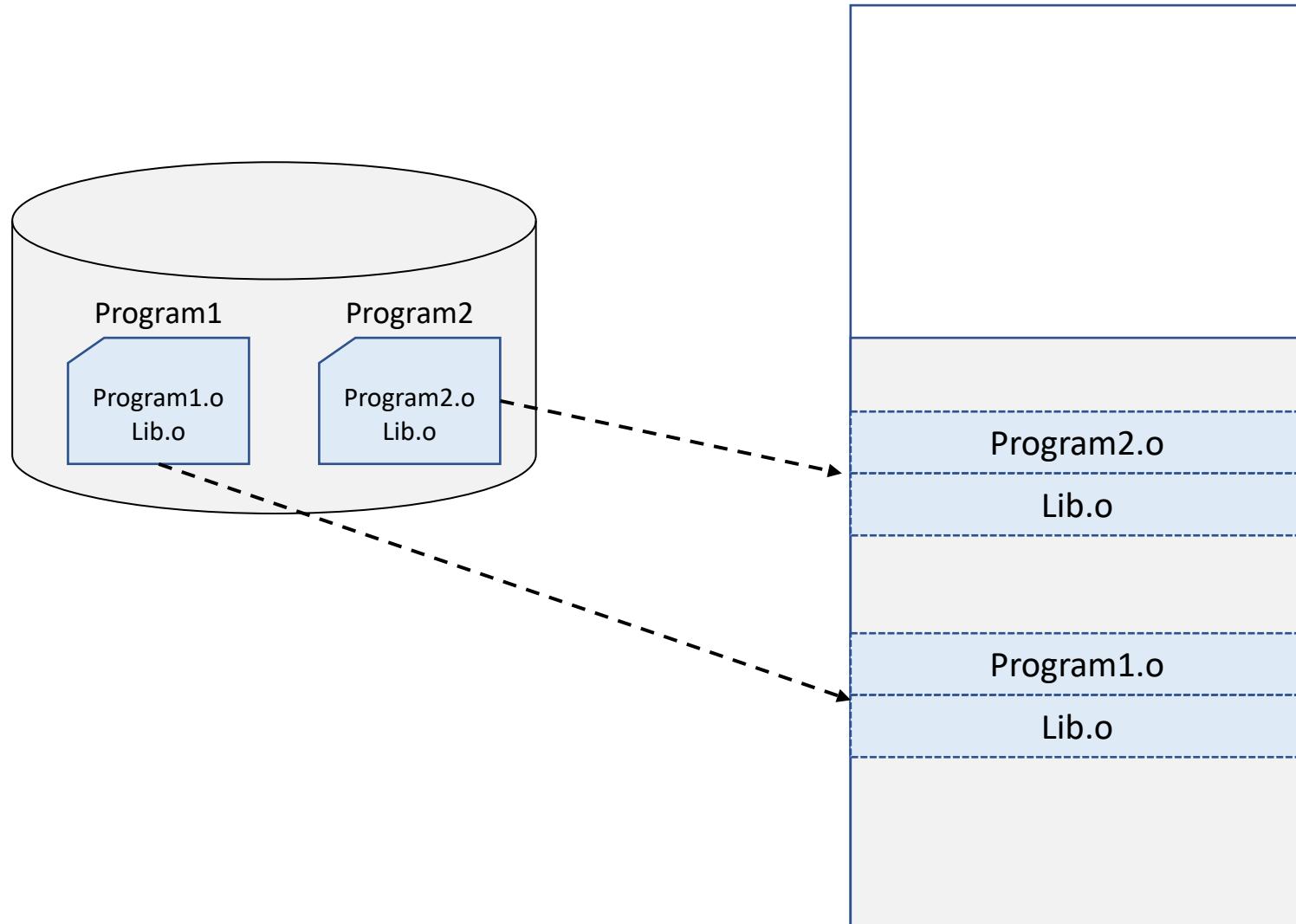
```
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strops.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)alloca_cutoff.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)libc-lowlevellock.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)malloc.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)morecore.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strchr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strcmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strcspn.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strdup.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strlen.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strncmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strrstr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)memcmp.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)memmove.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)memset.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)mempcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)stpcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)strcasecmp_l.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)memcpy.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)rawmemchr.o
(/usr/lib/gcc/x86_64-linux-gnu/10/../../../../x86_64-linux-gnu/libc.a)st
```

- A: 定义var (0x1000)
- B: movl \$0x2a, var

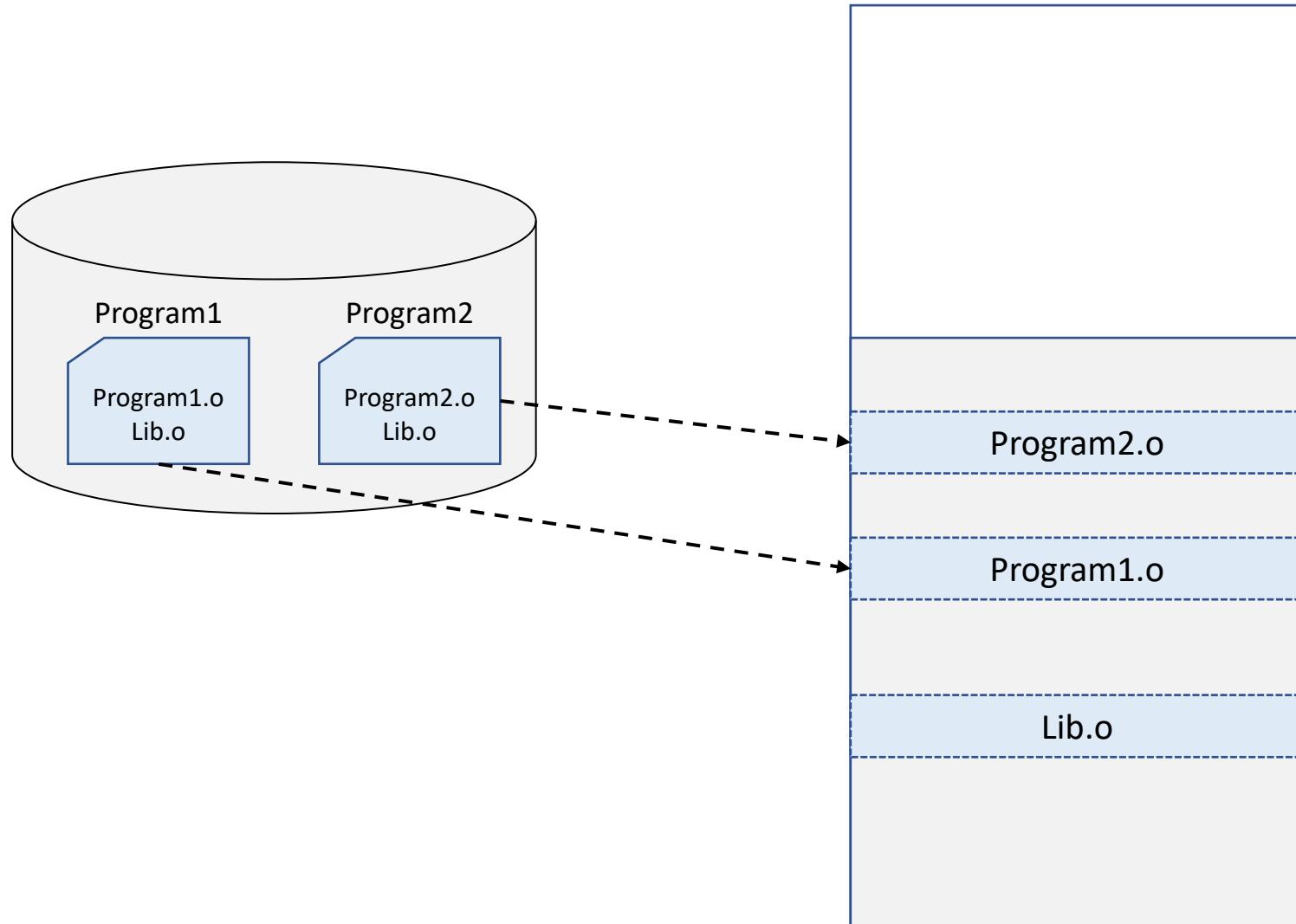


# 动态链接与加载

# 为什么要动态链接?



# 为什么要动态链接?



# 为什么要动态链接?

```
$ ls  
a.c  a.o  a.out  b.c  b.o  main.c  main.o  Makefile  
$ ls -l  
total 856  
-rw-rw-r-- 1 why why      42 11月   3 2020 a.c  
-rw-rw-r-- 1 why why    1200 9月  28 10:29 a.o  
-rwxrwxr-x 1 why why 846328 9月  28 10:37 a.out  
-rw-rw-r-- 1 why why      22 11月   3 2020 b.c  
-rw-rw-r-- 1 why why     984 9月  28 10:29 b.o  
-rw-rw-r-- 1 why why     278 9月  28 10:37 main.c  
-rw-rw-r-- 1 why why   2752 9月  28 10:37 main.o  
-rw-rw-r-- 1 why why     241 9月  28 10:25 Makefile
```

# 为什么要动态链接

- 去掉-fno-pic和-static

```
$ make
gcc -c a.c
gcc -c b.c
gcc -c main.c
gcc a.o b.o main.o
$ ls -l
total 48
-rw-rw-r-- 1 why why    42 11月  3  2020 a.c
-rw-rw-r-- 1 why why 1224 9月 28 10:47 a.o
-rwxrwxr-x 1 why why 16416 9月 28 10:47 a.out
-rw-rw-r-- 1 why why    22 11月  3  2020 b.c
-rw-rw-r-- 1 why why    984 9月 28 10:47 b.o
-rw-rw-r-- 1 why why   278 9月 28 10:37 main.c
-rw-rw-r-- 1 why why  2576 9月 28 10:47 main.o
-rw-rw-r-- 1 why why   245 9月 28 10:47 Makefile
```

10月 11 17:11



Terminal

why@why-VirtualBox: ~/Documents/ICS2021/teach/link-test



\$

S 英 拼

Right Shift + Right Alt

# 去掉-fno-pic和-static

这是默认的gcc编译/链接选项

- 文件相比静态链接大幅瘦身
  - a.o, b.o没有变化
  - main.o里面依然有00 00 00 00
    - 但是相对于rip的offset
    - relocation依然是x - 4, y - 4, foo - 4
  - a.out里面库的代码都不见了

# 位置无关代码

- 使用位置无关代码 (PIC) 的原因
  - 共享库不必事先决定加载的位置
  - 应用程序自己也是
    - 新版本gcc无论32/64 bit均默认PIC
    - 重现课本行为可以使用`-no-pie`选项编译
- PIC的实现
  - i386并不支持以下代码

```
movl $1, 1234(%eip)
```

- 于是有了你们经常看到的`__i686.get_pc_thunk.bx`
  - “获取 next PC 的地址” (如何实现?)

```
mov (%esp), %ebx
```

# 动态链接

\$



# ./a.out的执行

```
$ vim $(which ldd) █
```

```
$ vim $(which ldd)
$ /lib64/ld-linux-x86-64.so.2 --list a.out
a.out: error while loading shared libraries: a.out: cannot open shared object file
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
    linux-vdso.so.1 (0x00007ffe23928000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9727540000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f9727743000)
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
    linux-vdso.so.1 (0x00007fff7ffd6000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff6f3e66000)
    /lib64/ld-linux-x86-64.so.2 (0x00007ff6f4069000)
$ /lib64/ld-linux-x86-64.so.2 --list ./a.out
    linux-vdso.so.1 (0x00007ffc715df000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2042827000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2042a2a000)
$ /lib64/ld-linux-x86-64.so.2 ./a.out
Hello
100 + 200 = 300
World
World
$ █
```

管理 控制 视图 热键 设备 帮助

\$ |



# 共享对象 (a.out) 的加载

- 命令: ldd
  - “print shared object dependencies”
    - linux-vdso.so.1
      - 暂时忽略它
    - libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6
    - /lib64/ld-linux-x86-64.so.2
      - 多次打印, 地址会发生变化 (ld会执行加载过程)
- Ldd竟然是一个脚本 (vim \$(which ldd))
  - 挨个尝试调用若干ld-linux.so候选
    - 加上一系列环境变量
    - 我们可以用--list选项来达到类似效果

# 共享对象 (a.out) 的加载 (cont'd)

- 终于揭开谜题

- readelf -a a.out

- program header中有一个INTERP
    - /lib64/ld-linux-x86-64.so.2
    - 这个字符串可以直接在二进制文件中看到

```
$ readelf -l a.out | grep interpreter
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

- 神奇的ld.so

- /lib64/ld-linux-x86-64.so.2 ./a.out
    - 和执行./a.out的行为完全一致
      - (这才是./a.out真正在操作系统里发生的事情)
      - 与sha-bang(#!)实现机制完全相同
    - ld.so到底做了什么? (下学期分解)

# 动态链接器/lib/ld-linux.so.2

- 动态链接器本身是动态还是静态链接的，为什么？

```
$ ldd /lib64/ld-linux-x86-64.so.2  
      statically linked
```

- 动态链接器是否是PIC的？

```
$ readelf -d /lib64/ld-linux-x86-64.so.2  
  
Dynamic section at offset 0x32e70 contains 19 entries:  
  Tag          Type           Name/Value  
 0x000000000000000e (SONAME)   Library soname: [ld-linux-x86-64.so.2]  
 0x0000000000000004 (HASH)     0x2f0  
 0x0000000006ffffef5 (GNU_HASH) 0x3b8  
 0x0000000000000005 (STRTAB)   0x790  
 0x0000000000000006 (SYMTAB)   0x4a8  
 0x000000000000000a (STRSZ)    549 (bytes)  
 0x000000000000000b (SYMENT)   24 (bytes)  
 0x0000000000000003 (PLTGOT)   0x34000  
 0x0000000000000002 (PLTRELSZ) 96 (bytes)  
 0x0000000000000014 (PLTREL)   RELA  
 0x0000000000000017 (JMPREL)   0xb90  
 0x0000000000000007 (RELA)     0xaa0  
 0x0000000000000008 (RELASZ)   240 (bytes)  
 0x0000000000000009 (RELAENT)  24 (bytes)  
 0x0000000006fffffc (VERDEF)   0x9f8  
 0x0000000006fffffd (VERDEFNUM) 5  
 0x0000000006fffff0 (VERSYM)   0x9b6  
 0x0000000006fffff9 (RELACOUNT) 8  
 0x0000000000000000 (NULL)     0x0
```

# 动态链接：实现

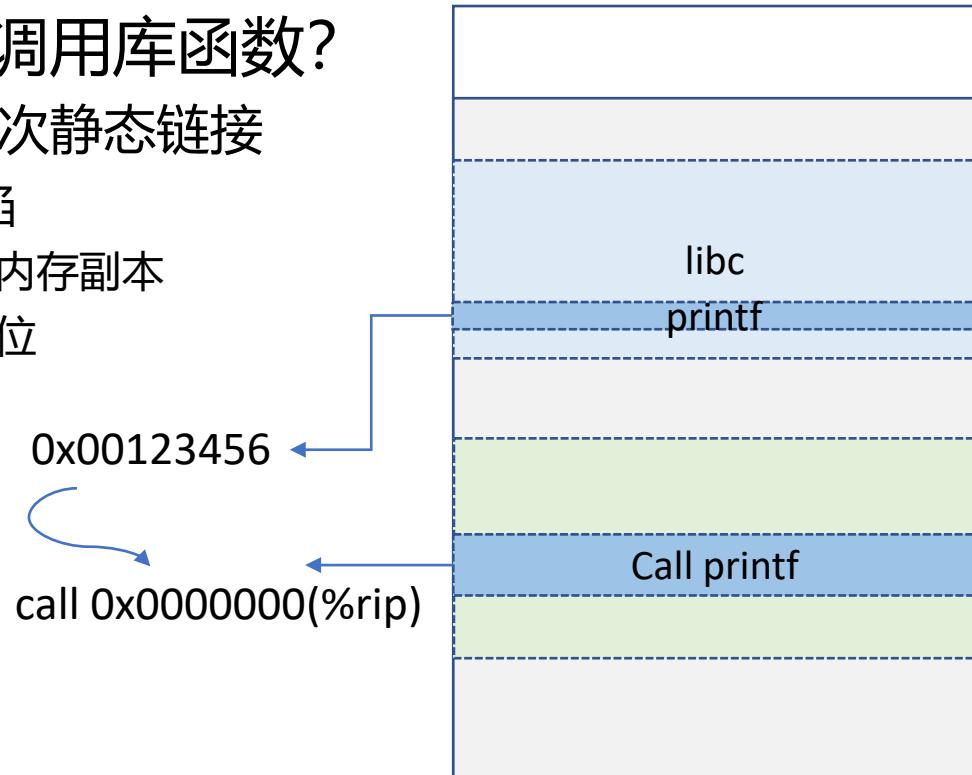
# 我们刚才忽略了一个巨大的问题

main.o在链接时，对printf的调用地址就确定了  
但是libc每次加载的位置都不一样啊！

- 应用程序使用怎样的指令序列调用库函数？

- 可以在库加载的时候重新进行一次静态链接
  - 但是这个方案有一些明显的缺陷
    - 各个进程的代码不能共享一个内存副本
    - 没有使用过的符号也需要重定位

```
000000000000002e <main>:  
2e: f3 0f 1e fa        endbr64  
32: 55                 push %rbp  
33: 48 89 e5           mov %rsp,%rbp  
36: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi      # 3d <main+0xf>  
3d: e8 00 00 00 00     call 42 <main+0x14>  
42: b8 15 00 00 00 00 00 mov 0x0(%rip),%edx      # 48 <main+0x1a>  
48: b8 05 00 00 00 00 00 mov 0x0(%rip),%eax      # 4e <main+0x20>  
4e: 89 d6               mov %edx,%esi  
50: 89 c7               mov %eax,%edi  
52: e8 00 00 00 00     call 57 <main+0x29>  
57: 89 c1               mov %eax,%ecx  
59: b8 15 00 00 00 00 00 mov 0x0(%rip),%edx      # 5f <main+0x31>  
5f: b8 05 00 00 00 00 00 mov 0x0(%rip),%eax      # 65 <main+0x37>  
65: 89 c6               mov %eax,%esi  
67: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi      # 6e <main+0x40>  
6e: b8 00 00 00 00     mov $0x0,%eax  
73: e8 00 00 00 00     call 78 <main+0x4a>  
78: b8 00 00 00 00     mov $0x0,%eax  
7d: 5d                 pop %rbp  
7e: c3                 ret
```



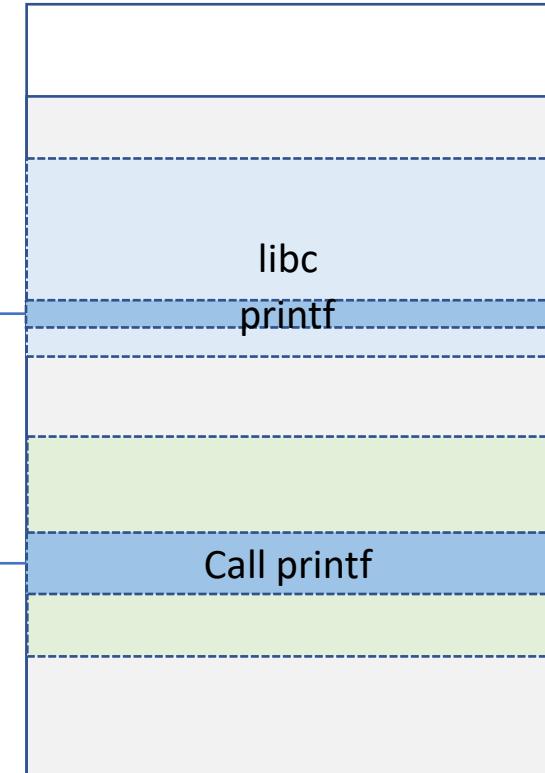
# ELF：查表

- 基本款

```
call *table[PRINTF]
```

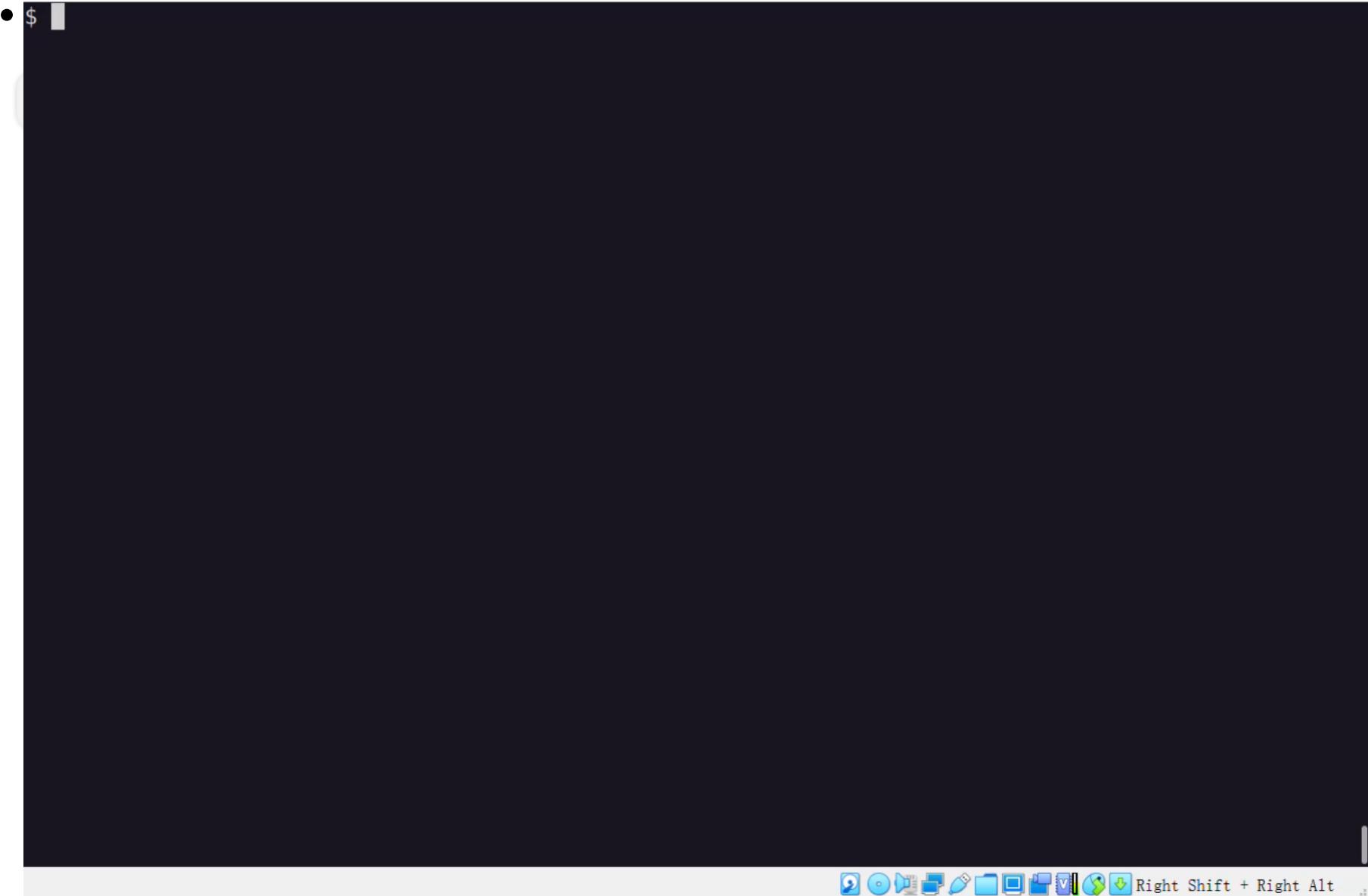
- 在链接时，填入运行时的table
  - 使用-fno-plt选项可以开启此款

0x00123456  
call \*table[printf]



# ELF：查表

管理 控制 视图 热键 设备 帮助



# ELF：查表

- 基本款

```
call *table[PRINTF]
```

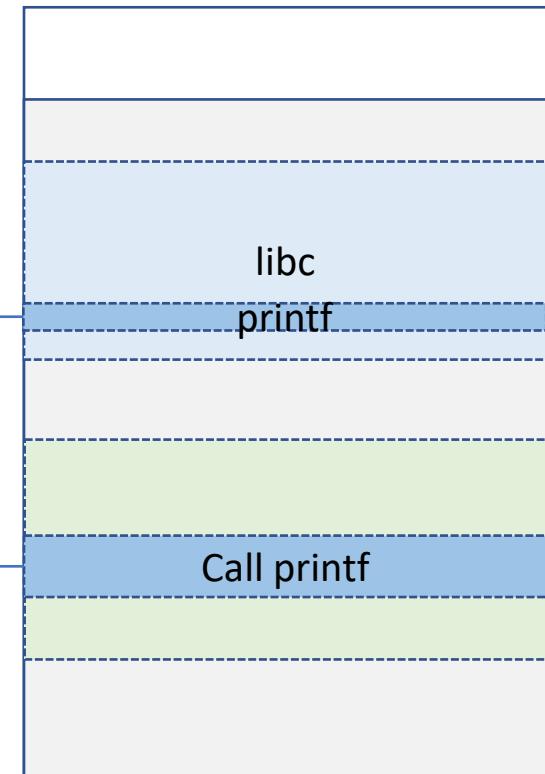
- 在链接时，填入运行时的table
  - 使用-fno-plt选项可以开启此款

- 豪华款（默认选项，使用PLT）

```
printf@plt:
```

```
jmp *table[PRINTF]  
push $PRINTF  
call resolve
```

```
0x00123456  
call *table[printf]
```



# 总结和反思

# 为什么我从来没听说过这些知识？？？

因为百度搜不到啊？

- 中文社区几乎不存在这些知识的详细解释
  - the friendly manual是英文写的
  - mailing list/StackOverflow都是英文
  - 国内的专家本来就很少
    - 仅有的那些也没空写文档
- 你们才是未来的希望
  - 不要动不动就说内卷
  - 不要为了无意义的GPA沾沾自喜
  - 不要停止自我救赎

End.  
(RTFM; STFW; RTFSC)

还有一个福利

# 理论课例子

管理 控制 视图 热键 设备 帮助

```
$ cat Lib.h
/*Lib.h*/
#ifndef LIB_H
#define LIB_H
void foobar(int i);
#endif
$ cat Lib.c
/*Lib.c*/
#include <stdio.h>
void foobar(int i){
    printf("Printing from Lib.so %d\n", i);
    sleep(-1);
}
$ cat Program1.c
/*Program1.c*/
#include "Lib.h"
int main(){
    foobar(1);
    return 0;
}
$ cat Program2.c
/*Program2.c*/
#include "Lib.h"
int main(){
    foobar(2);
    return 0;
}
$
```

# 动态链接.got.plt

管理 控制 视图 热键 设备 帮助

```
C/Lib.so
7f7fd02fe000-7f7fd02ff000 r-xp 00001000 08:03 9444077
C/Lib.so
7f7fd02ff000-7f7fd0300000 r--p 00002000 08:03 9444077
C/Lib.so
7f7fd0300000-7f7fd0301000 r--p 00002000 08:03 9444077
C/Lib.so
7f7fd0301000-7f7fd0302000 rw-p 00003000 08:03 9444077
C/Lib.so
7f7fd0302000-7f7fd0304000 rw-p 00000000 00:00 0
7f7fd0304000-7f7fd0305000 r--p 00000000 08:03 4726417
7f7fd0305000-7f7fd032c000 r-xp 00001000 08:03 4726417
7f7fd032c000-7f7fd0336000 r--p 00028000 08:03 4726417
7f7fd0336000-7f7fd0338000 r--p 00031000 08:03 4726417
7f7fd0338000-7f7fd033a000 rw-p 00033000 08:03 4726417
7fff71e8a000-7fff71eab000 rw-p 00000000 00:00 0
7fff71fdf000-7fff71fe3000 r--p 00000000 00:00 0
7fff71fe3000-7fff71fe5000 r-xp 00000000 00:00 0
ffffffff600000-ffffffff601000 --xp 00000000 00:00 0
$ readelf -sD Lib.so
```

Symbol table for image contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterT[...]
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	[...}@GLIBC_2.2.5 (2)
3:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_gmon_start
4:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_registerTMC[...]
5:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	sleep@GLIBC_2.2.5 (2)
6:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	[...}@GLIBC_2.2.5 (2)
7:	0000000000001139	55	FUNC	GLOBAL	DEFAULT	14	foobar



# 系统编程与基础设施

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

感受到 PA 的恶意了吗？

- 不就是写几行代码么，怎么.....怎么写不对啊.....
- 这就是为什么 PA 要搞那么麻烦：又是 Makefile，又是各种项目/工具
- 没有适当的基础设施，PA 的完成率会大幅降低

- 开发/调试的基础设施
- 系统编程：基础设施
- Differential testing 代码导读

# 开发/调试的基础设施

# 你们是怎么写程序的？

- 虽然很多同学已经配置好了良好的编程环境，但依然有很多同学在 **“面向浪费时间编程”**
  - yzh 推荐 vim/tmux 的原因是大家用 IDE 容易迷失自我 (关注表象，而不知道内在是如何工作的)
  - 在你搞清楚的前提下，也可以解放自我

```
$ gcc a.c
a.c: In function ‘main’: a.c:5:1:
error: ‘a’ undeclared (first use in this function)
$ vi a.c
$ gcc a.c
$ ./a.out 1 2 // 你输入的
zsh: segmentation fault (core dumped) ./a.out
...
```

\$

S 英 汉 简 拼 \*



Right Shift + Right Alt

```
$ ./a.out  
3 3  
3 + 3 = 6  
$ █
```



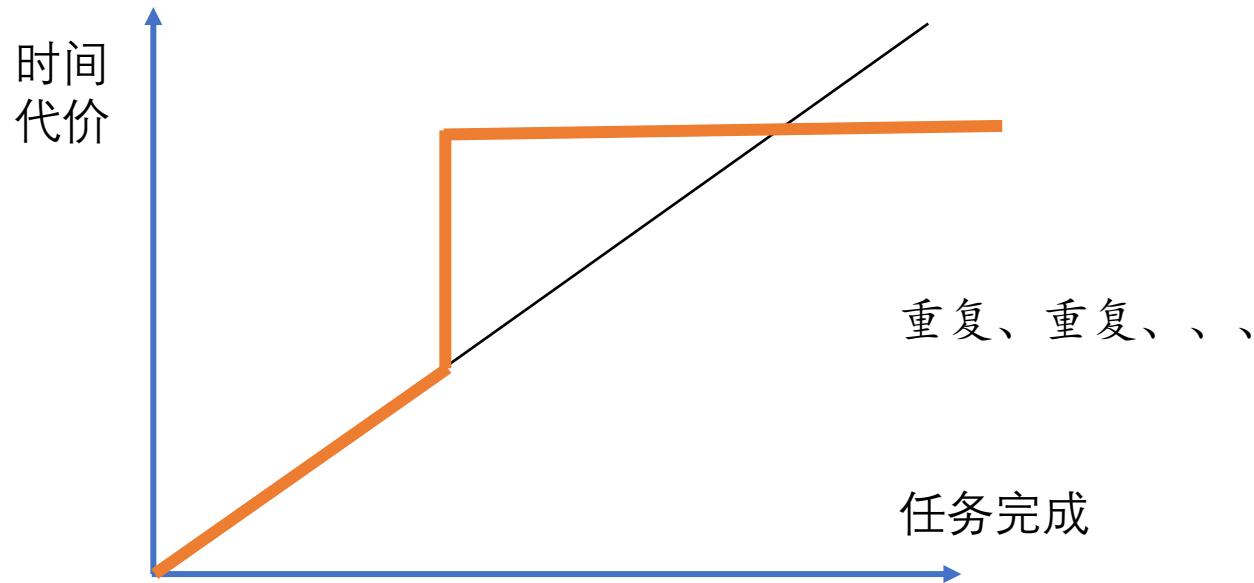
Right Shift + Right Alt 6

# 你们是怎么写程序的？

- 虽然很多同学已经配置好了良好的编程环境，但依然有很多同学在 “**面向浪费时间编程**”
  - yzh 推荐 vim/tmux 的原因是大家用 IDE 容易迷失自我 (关注表象，而不知道内在是如何工作的)
  - 在你搞清楚的前提下，也可以解放自我

```
$ gcc a.c
a.c: In function ‘main’: a.c:5:1:
error: ‘a’ undeclared (first use in this function)
$ vi a.c
$ gcc a.c
$ ./a.out 1 2 // 你输入的
zsh: segmentation fault (core dumped) ./a.out
...
```

# 时间分析



# 基础设施的本质

---

- 通过适当的配置、脚本减少思维中断的时间，提高连贯性，保持短时记忆活跃
  - make (fresh build) → 4s, 已被打断
  - make (parallel) → 0.5s
- 基本原则：
  - 如果你认为有提高效率的可能性，一定有人已经做了

```
$ ls
abstract-machine      fceux-am    Makefile    README.md
am-kernels             init.sh     nemu        tags
$ █
```

S 英 单 简 拼 \*

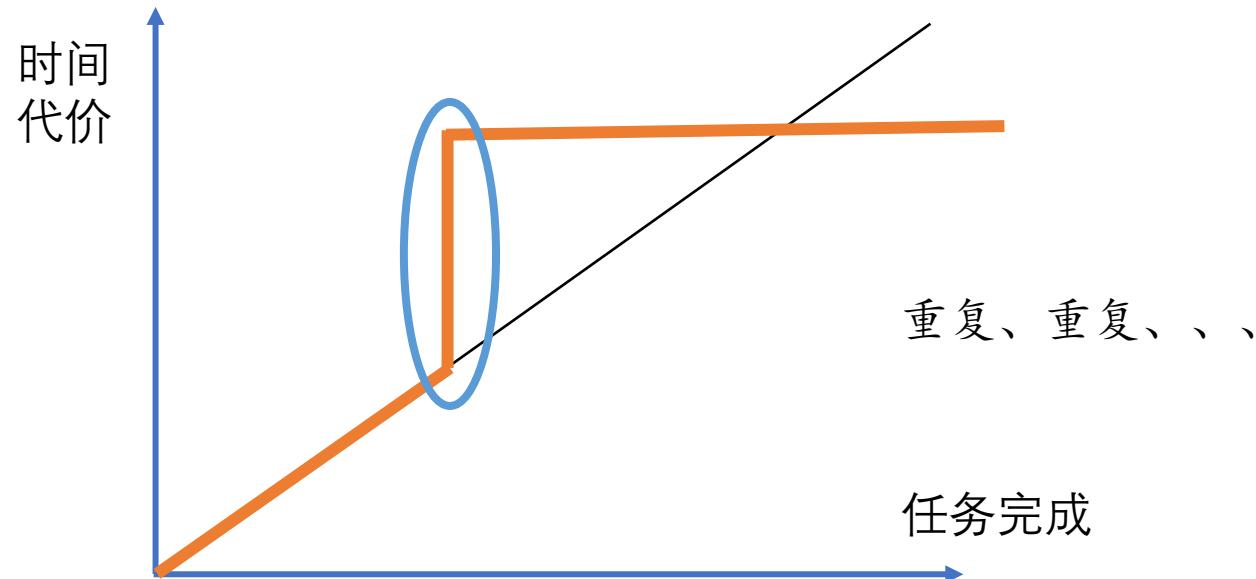
# 基础设施的本质

- 通过适当的配置、脚本减少思维中断的时间，提高连贯性，保持短时记忆活跃
  - make (fresh build) → 4s, 已被打断
  - make (parallel) → 0.5s
- 基本原则：
  - 如果你认为有提高效率的可能性，一定有人已经做了
  - 每次都 make -j8?
    - 或者 alias make='make -j8'
    - 在 Makefile 里加一行 MAKEFLAGS += -j 8 (better)
      - 配置一键编译运行 → 1s 内完成

```
# Building fceux-image [x86-nemu]
# Building am-archive [x86-nemu]
# Building klib-archive [x86-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-x86-nemu.a
+ LD -> build/fceux-x86-nemu.elf
# Creating image [x86-nemu]
+ OBJCOPY -> build/fceux-x86-nemu.bin
$ export ARCH=x86-nemu
$ make
# Building fceux-image [x86-nemu]
+ CXX src/emufile.cpp
# Building am-archive [x86-nemu]
+ CC src/platform/nemu/trm.c
+ AR -> build/am-x86-nemu.a
# Building klib-archive [x86-nemu]
+ LD -> build/fceux-x86-nemu.elf
# Creating image [x86-nemu]
+ OBJCOPY -> build/fceux-x86-nemu.bin
$ export ARCH=native
$ make
```

# 心态分析

- 本质上，这都不是难事，STFW 随手即来，但大家通常做不好
  - 尚未 GET STFW 的技能
    - 在 hosts 中屏蔽百度 (或修改默认搜索引擎)
  - 惰性
    - 键入 make -j8: 增加 1s 时间
    - STFW: 至少需要几分钟，而且有不少失败尝试 (短期收益是负的，尤其是上课 workloads 已经很重的前提下)



管理 控制 视图 热键 设备 帮助

\$ vim Makefile |

S 英 ◙ 简 拼 \*

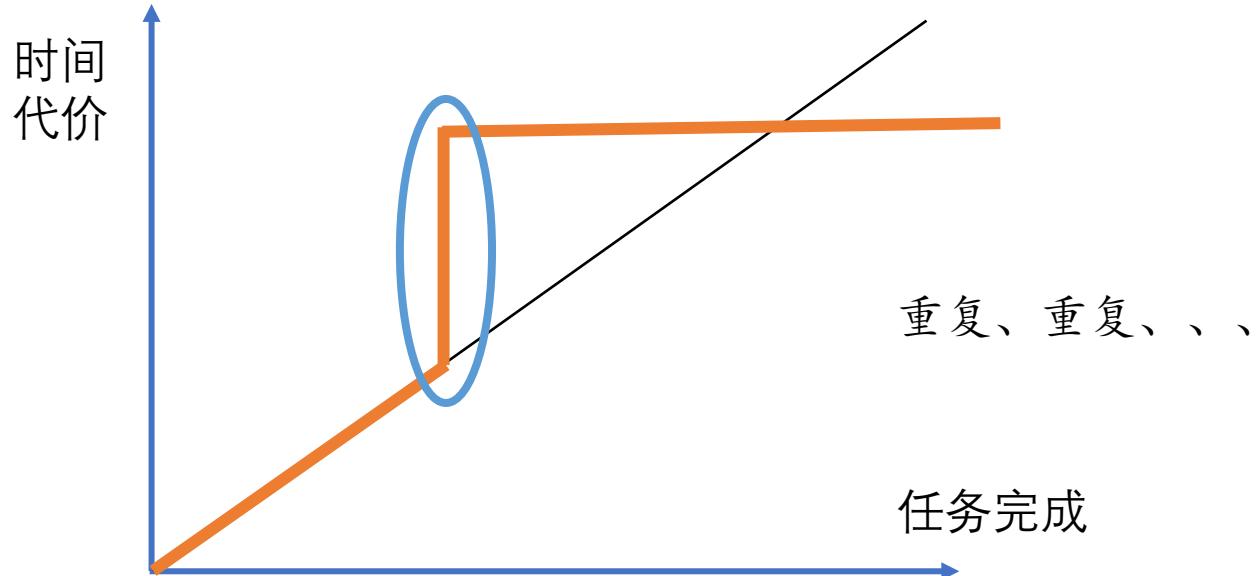


14

# 心态分析 (cont'd)

克服惰性可以使你快速成长。

- 在很多小事上，可能并不带来显著的收益
  - 每次键入 make -j8 可能并不显著缩短 PA 完成的时间
  - 但久而久之成为习惯，你就总是想着改进基础设施

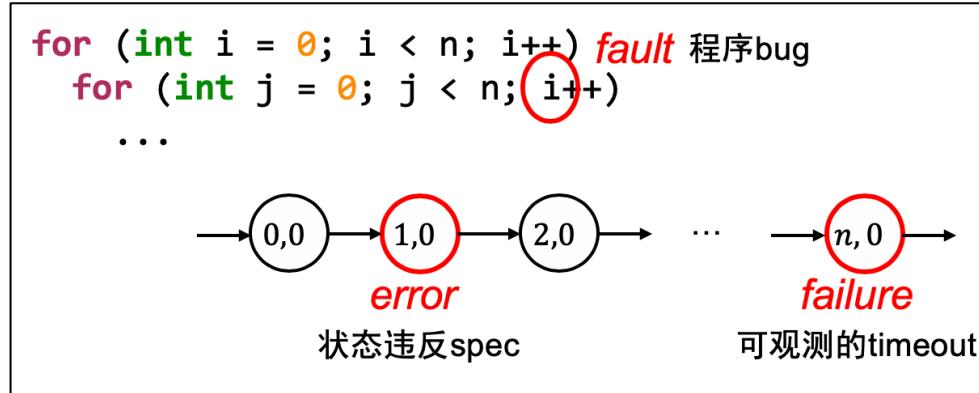


# 系统编程：建立基础设施

# 复习：测试/调试的理论

Fault → Error → Failure

- 对于 PA 来说，failure 是显而易见的：
  - Segmentation Fault 了， fail
  - HIT BAD TRAP 了， fail
  - 马里奥/仙剑不能跑嘛， fail
- 我已经调了很久了，但就是找不到那个导致 error 的指令啊
  - 怎样找到 error 发生的位置？
  - 二分法似乎还是太麻烦了？



# 抱大腿：一种想法

- 如果学长/同学已经有一份正确的代码，能不能借助这个代码快速诊断出自己代码的问题？
  - 同学们是非常智慧的，在 ICS-PA 创立的初期发明了替换调试法
    - PA 是分模块实现的 (若干个 .c)
    - 从自己的代码开始，逐个替换进大腿同学的 .c
    - 第一个替换后通过测试的文件里有 bug
- Cool!
  - 可能的改进：以函数级进行替换

大腿同学



小腿同学



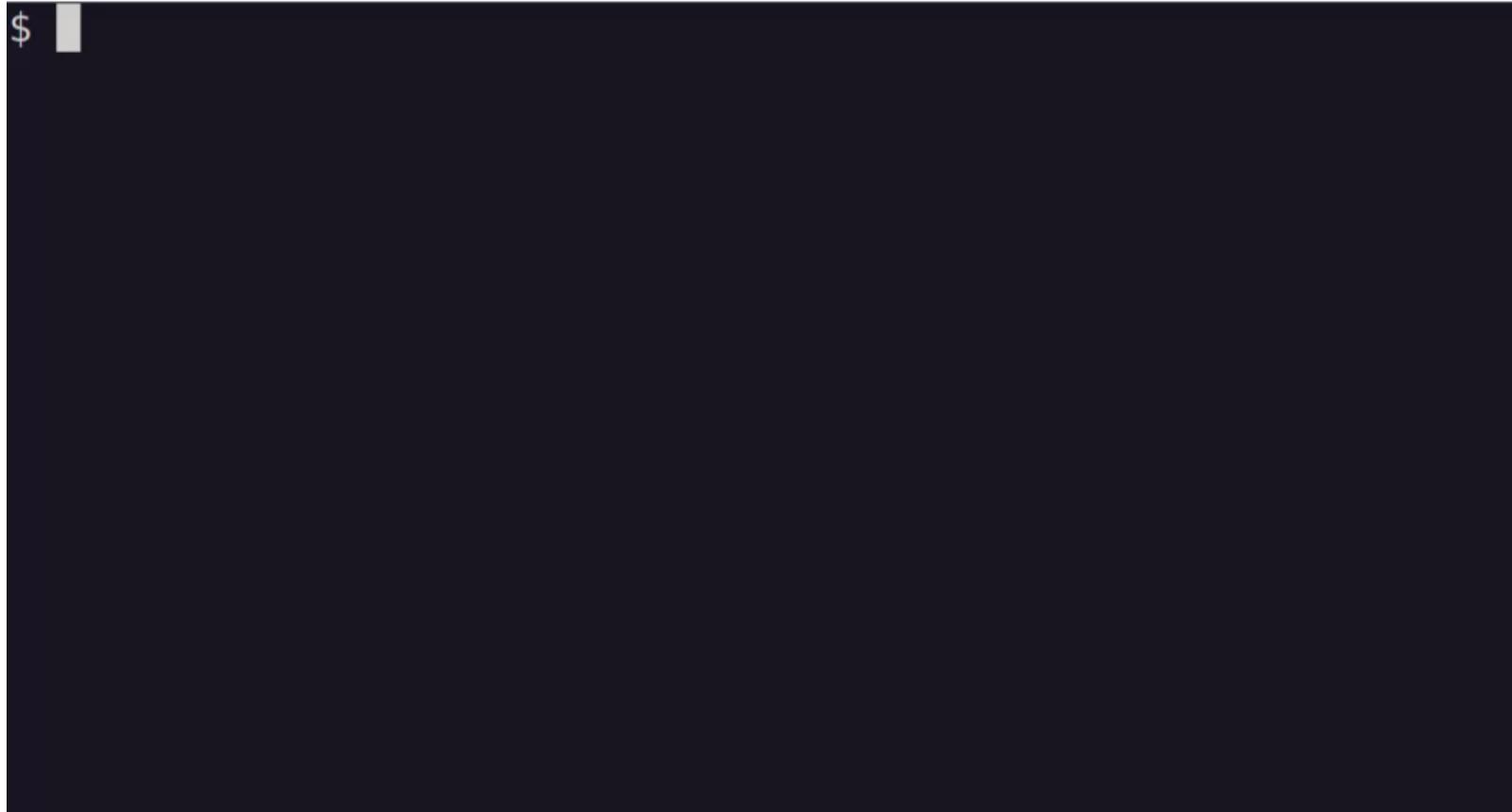
# Delta Debugging

- 假设程序  $P$  会fail,  $P_{\text{大腿}}$  会pass
  - 并且假设两份程序所有函数行为都相同
    - 将  $P_{\text{大腿}}$  中的函数  $f$  替换成  $P$  中的  $f$
    - 依然通过  $\rightarrow f$  实现正确
    - 否则  $\rightarrow f$  实现有bug
- 把类似的想法用在输入上
  - 不断把输入的一部分去掉, 直到不能触发 bug 为止
    - Anders Zeller and Ralf Hildebrandt. [Simplifying and isolating failure-inducing input.](#) IEEE Transactions on Software Engineering, 28(2), 2002.
  - 不断把输入的一部分去掉, 直到行为不再保持
    - Md Rafiqul Islam Rabin et al. [Understanding Neural Code Intelligence through Program Simplification.](#) ICSE'21

# 大腿同学代码的另一种用法

- 大腿同学的代码还可以帮助我们直接定位到错误的指令！
  - 只需要大腿的 compiled binary，就能指令级定位出错的位置

```
for i in range(int(sys.argv[1])):
    print('\n'.join(['si' + f'p ${r}' for r in ['eax', ...]]))
```



A terminal window with a black background and a white cursor. The prompt '\$' is visible at the top left. The rest of the window is empty, showing only the dark background.

# 大腿同学代码的另一种用法

- 大腿同学的代码还可以帮助我们直接定位到错误的指令！
  - 只需要大腿的 compiled binary
  - 就能指令级定位出错的位置

```
for i in range(int(sys.argv[1])):
    print('\n'.join(['si' + f'p ${r}' for r in ['eax', ...]]))
```

- 然后找到 log 第一个不一致的地方！

```
N=10000
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
<(python3 cmdgen.py $N | ./x86-nemu img)
```

# 基础设施：其实没那么困难

每做的一点自动化都是在给大项目节约维护成本。

- 程序员们就是喜欢造轮子
  - (日常管理) 效率低
    - → 熟练使用命令行工具/Python
  - (项目管理) 你已经会 gcc a.c 了，但没法管理几十个文件
    - → make, 一键编译/运行/测试
  - (代码编辑) 在代码里跳来跳去很麻烦
    - → IDE/配置 Vim/装插件
  - (错误检查) 很容易犯低级错误
    - → -Wall, -Werror, fsanitize=address
  - (代码调试) Segmentation Fault了
    - → gdb

# Differential Testing

N=10000

```
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
<(python3 cmdgen.py $N | ./x86-nemu img)
```

# Differential Testing

“同一套接口 (API) 的两个实现应当行为完全一致”

- 大腿同学 & 小腿同学：指令集的两套实现
  - 还有什么现实中软件系统的例子？
- 你能找到两份独立实现的东西，都可以测试
  - 浏览器 [Mesbah and Prasad, ICSE'11](#)
  - GCC (vs clang), [Yang et al., PLDI'11](#)
  - 文件系统, [Min et al., SOSP'15](#)
  - 数据库 [Rigger and Su, OSDI'20](#)
  - Gcov (vs llvm-cov)真的能在 gcc/llvm 里发现很多 bugs
  - DL library [Pham et al., ICSE'19](#)

# NEMU: 实现 Differential Testing

---

- 刚才我们已经给大腿的代码实现了一个简单版本的 diff-testing
- 真正的大腿: QEMU
- ICS PA = 缩水版 QEMU
  - 实际上 PA 就是简化(教学)版的 QEMU
- diff-testing 实验
  - 以前我们都只是自己写自己的程序, 调用库函数
  - diff-testing 是和其他程序 (QEMU, gdb) 协作/交互的例子

# NEMU Differential Testing: 原理

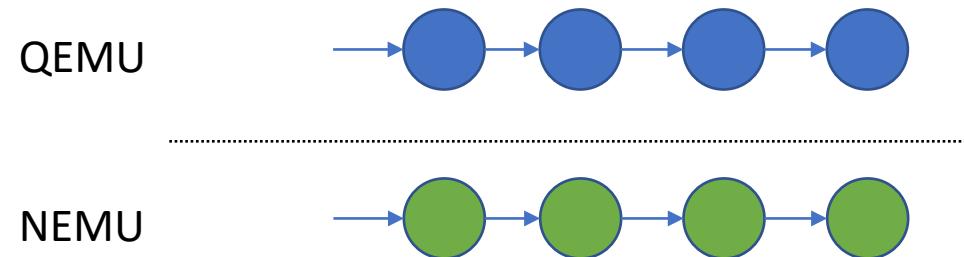
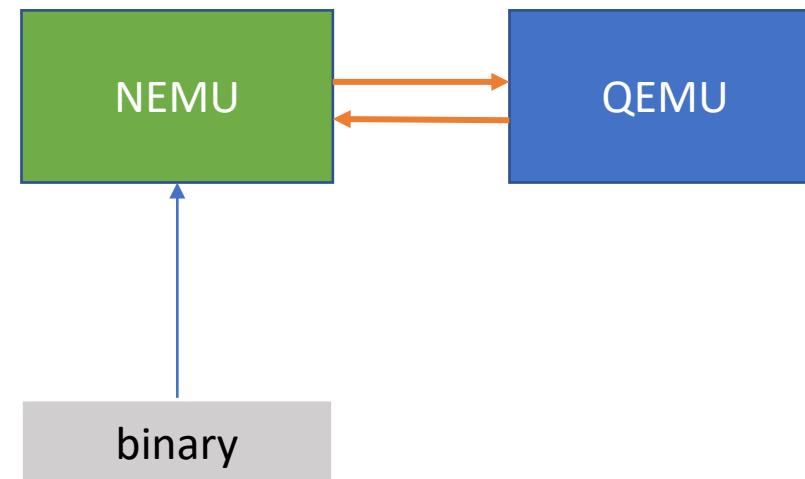
```
# gen-cmds: 不断生成 si; p $eax; p $ebx; ...
diff-stream <(gen-cmds | nemu) <(gen-cmds | qemu-system-i386)
```

- 改进版的“抱大腿”代码，不需二分查找
  - 同时启动两个 NEMU (QEMU)
  - 在第一个输出不同时停止
- (QEMU Monitor 展示)
  - -serial mon:stdio 启动 monitor!

# NEMU Differential Testing: 原理 (cont'd)

- 问题分析：我们需要使 QEMU 像 NEMU 一样执行指令！
  - QEMU 实现了 gdb 的协议
  - 协议格式同 monitor
- qemu-diff 实现：
  1. 启用 gdb 连接 QEMU
  2. 用 `gdb_si()` 在 QEMU 中执行一条指令
  3. 比较指令执行后寄存器是否有区别
  4. 动 QEMU 并配置它进入与 PA 类似的模式

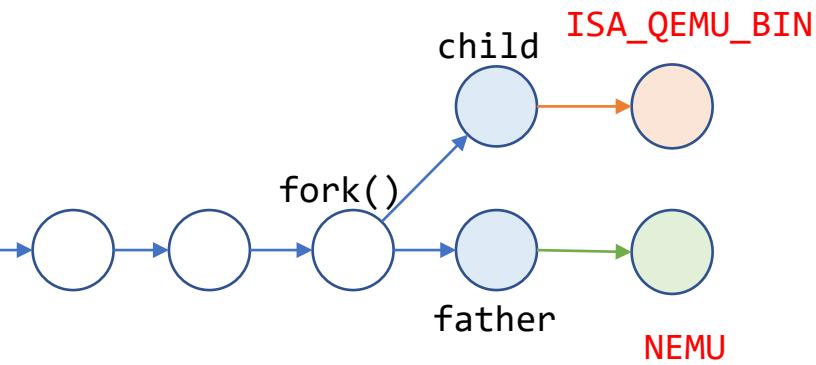
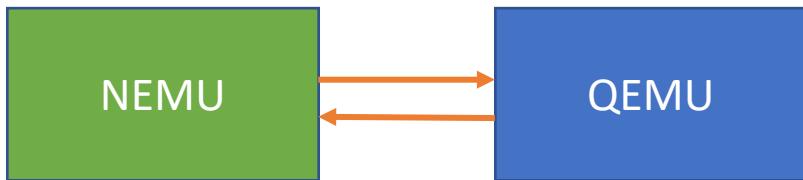
# NEMU和QEMU协作



\$ vim nemu/

\$ █

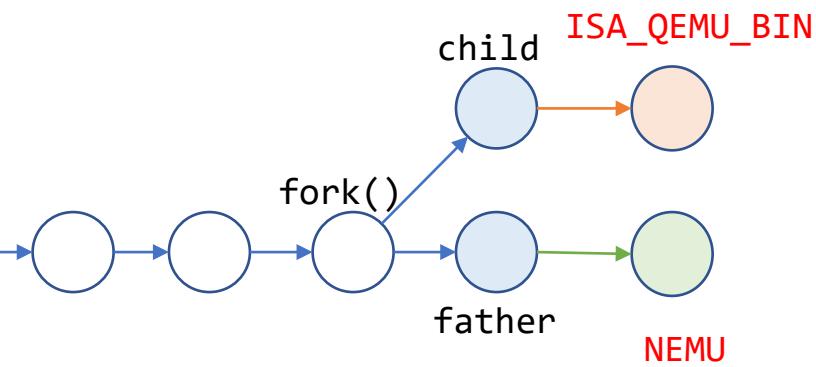
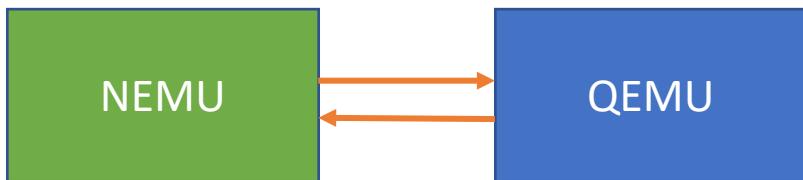
# NEMU和QEMU协作



```
9 #define ISA_QEMU_BIN "qemu-system-riscv32"
10#define ISA_QEMU_ARGS "-bios", "none",
11#elif defined(CONFIG_ISA_riscv64)
12#define ISA_QEMU_BIN "qemu-system-riscv64"
```

```
38 void difftest_init(int port) {
39     char buf[32];
40     sprintf(buf, "tcp::%d", port);
41
42     int ppid_before_fork = getpid();
43     int pid = fork();
44     if (pid == -1) {
45         perror("fork");
46         assert(0);
47     }
48     else if (pid == 0) {
49         // child
50
51         // install a parent death signal in the chlid
52         int r = prctl(PR_SET_PDEATHSIG, SIGTERM);
53         if (r == -1) {
54             perror("prctl error");
55             assert(0);
56         }
57
58         if (getppid() != ppid_before_fork) {
59             printf("parent has died!\n");
60             assert(0);
61         }
62
63         close(STDIN_FILENO);
64         execvp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-gdb", buf,
65                "-serial", "none", "-monitor", "none", NULL);
66         perror("exec");
67         assert(0);
68     }
69     else {
70         // father
71
72         gdb_connect_qemu(port);
73         printf("Connect to QEMU with %s successfully\n", buf);
74
75         atexit(gdb_exit);
76
77         init_isa();
78     }
79 }
```

# NEMU和QEMU协作



```
bool gdb_connect_qemu(int port) {
    // connect to gdbserver on localhost port 1234
    while ((conn = gdb_begin_inet("127.0.0.1", port)) == NULL) {
        msleep(1);
    }
    return true;
}
```

```
38 void difftest_init(int port) {
39     char buf[32];
40     sprintf(buf, "tcp::%d", port);
41
42     int ppid_before_fork = getpid();
43     int pid = fork();
44     if (pid == -1) {
45         perror("fork");
46         assert(0);
47     }
48     else if (pid == 0) {
49         // child
50
51         // install a parent death signal in the chlid
52         int r = prctl(PR_SET_PDEATHSIG, SIGTERM);
53         if (r == -1) {
54             perror("prctl error");
55             assert(0);
56         }
57
58         if (getppid() != ppid_before_fork) {
59             printf("parent has died!\n");
60             assert(0);
61         }
62
63         close(STDIN_FILENO);
64         execl(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-gdb", buf,
65               "-serial", "none", "-monitor", "none", NULL);
66         perror("exec");
67         assert(0);
68     }
69     else {
70         // father
71
72         gdb_connect_qemu(port);
73         printf("Connect to QEMU with %s successfully\n", buf);
74
75         atexit(gdb_exit);
76
77         init_isa();
78     }
79 }
```

- 大腿同学的代码还可以帮助我们直接定位到错误的指令！

- 只需要大腿的 compiled binary
- 就能指令级定位出错的位置

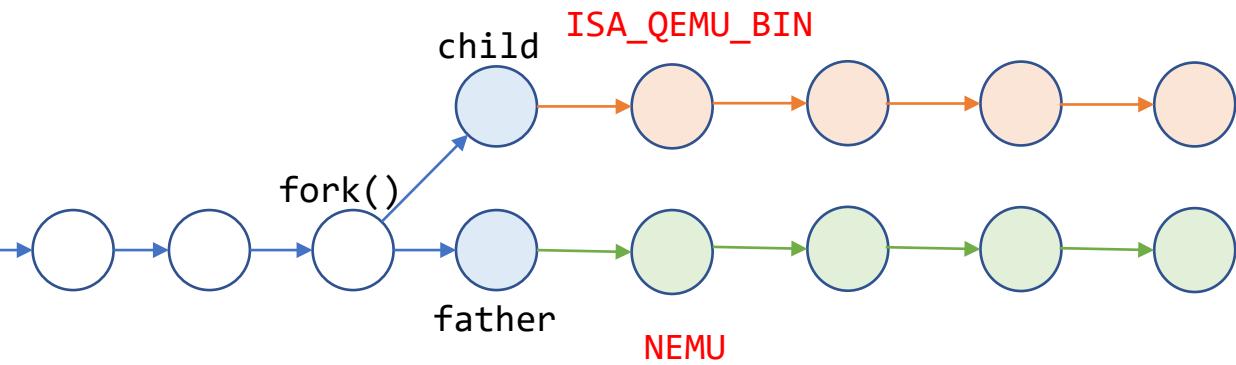
NEMU

- 然后找到 log 第一个不一致的地方！

```
N=10000
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
<(python3 cmdgen.py $N | ./x86-nemu img)
```

```
ssfully\n", buf);
```

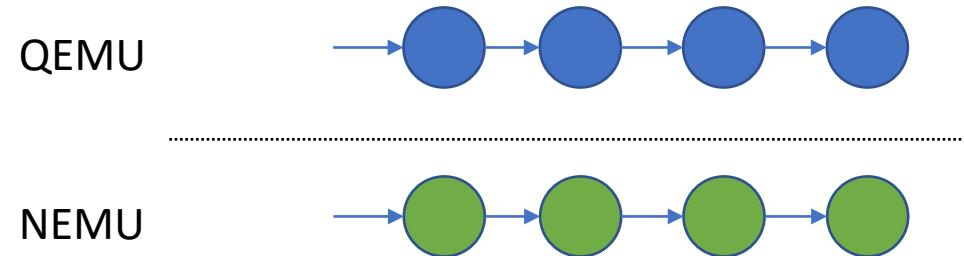
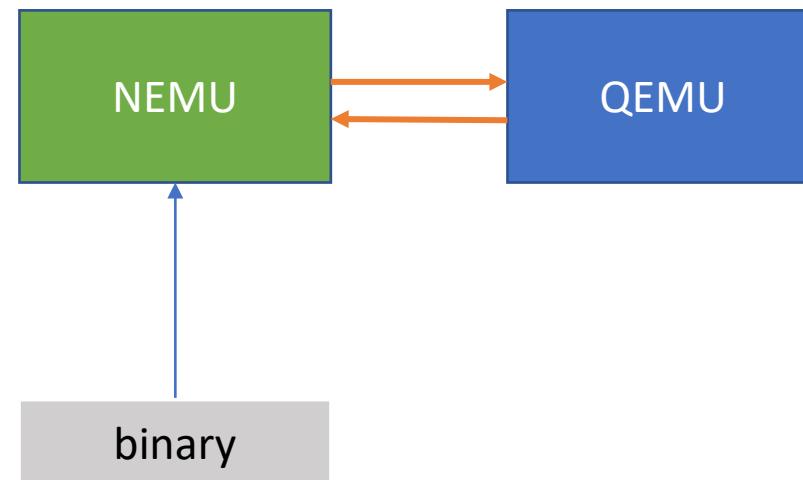
```
int64_t n) {
    i();
```



```
47 bool gdb_getregs(union isa_gdb_regs *r) {
48     gdb_send(conn, (const uint8_t *)"g", 1);
49     size_t size;
50     uint8_t *reply = gdb_recv(conn, &size);
51
52     int i;
53     uint8_t *p = reply;
54     uint8_t c;
55     --- 7 lines: for (i = 0; i < sizeof(union isa_gdb_
```

```
bool gdb_si() {
    char buf[] = "vCont;s:1";
    gdb_send(conn, (const uint8_t *)buf, strlen(buf));
    size_t size;
    uint8_t *reply = gdb_recv(conn, &size);
    free(reply);
    return true;
}
```

# NEMU和QEMU协作



管理 控制 视图 热键 设备 帮助

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course14

why@why-VirtualBox: ~/Documents/jyy-wiki-git/wiki

\$

S 英 单 简 拼 \*

Right Shift + Right Alt

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course14

why@why-VirtualBox: ~/Documents/jyy-wiki-git/wiki

X V

\$

S 英 汉 简 拼 \*

Right Shift + Right Alt

# 代码导读

- Differential testing 的初始化

- 我们可以用 QEMU + gdb 调试这段代码！
- -s -S 启动 QEMU; gdb target remote localhost:1234

```
uint8_t mbr[] = {  
    0xfa,                                // cli  
    0x31, 0xc0,                            // xorw %ax,%ax  
    0x8e, 0xd8,                            // movw %ax,%ds  
    0x8e, 0xc0,                            // movw %ax,%es  
    0x8e, 0xd0,                            // movw %ax,%ss  
    0x0f, 0x01, 0x16, 0x44, 0x7c,        // lgdt gdtdesc  
    0x0f, 0x20, 0xc0,                      // movl %cr0,%eax  
    0x66, 0x83, 0xc8, 0x01,                // orl $CR0_PE,%eax  
    0x0f, 0x22, 0xc0,                      // movl %eax,%cr0  
    0xea, 0x1d, 0x7c, 0x08, 0x00,        // ljmp $GDT_ENTRY(1),$start32  
    ...  
};
```

管理 控制 视图 热键 设备 帮助

```
$ ls  
a.cpp  a.out  in.txt  mbr  
$ █
```

管理 控制 视图 热键 设备 帮助

00000090:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000a0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000b0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000c0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000d0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000e0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000000f0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000100:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000110:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000120:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000130:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000140:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000150:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000160:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000170:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000180:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
00000190:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001a0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001b0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001c0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001d0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001e0:	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	.....
000001f0:	0000	0000	0000	0000	0000	0000	0000	0000	55aa	.....	U.

\$ qemu-system-i386

^Cqemu-system-i386: terminating on signal 2

\$ qemu-system-i386 mbr

WARNING: Image format was not specified for 'mbr' and probing guessed raw.

Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.

Specify the 'raw' format explicitly to remove the restrictions.

\$



Right Shift + Right Alt

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course12

why@why-VirtualBox: ~/Documents/ICS2021/teach

00000120: 0000 0000 0000 0000 0000 0000 0000 0000 .....

QEMU [Paused] - Press Ctrl+Alt+G to release grab

Machine View

0000

0000

0000

0000

0000

0000

0000

0000

0000

0000

0000

0000

0000

0000

[2]-

Guest has not initialized the display (yet).

\$ qe

^Cqe

\$ qe

WARN

blo

^Cqe

\$ qe

WARN

blo

Specify the 'raw' format explicitly to remove the restrictions.

管理 控制 视图 热键 设备 帮助

```
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000 . . . . .  
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 . . . . .  
- 00000110: 0000 0000 0000 0000 0000 0000 0000 0000
```

**QEMU [Paused]**

Machine View

9999

0000

0000

0000

0000

0000

0006

0000

0000

0000

888

6

86

6

WAPN

1

bloo

1

\$ ge

[1]

\$ WARNING: Image format was not specified for .mst and probing guessed raw.

Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.

Specify the 'raw' format explicitly to remove the restrictions.



```
B+>0x7c00 cli
0x7c01 xor %eax,%eax
0x7c03 mov %eax,%ds
0x7c05 mov %eax,%es
0x7c07 mov %eax,%ss
0x7c09 lgdtl (%esi)
0x7c0c inc %esp
0x7c0d jl 0x7c1e
0x7c0f and %al,%al
0x7c11 or $0x1,%ax
0x7c15 mov %eax,%cr0
0x7c18 ljmp $0xb866,$0x87c1d
0x7c1f adc %al,(%eax)
0x7c21 mov %eax,%ds
0x7c23 mov %eax,%es
0x7c25 mov %eax,%ss
0x7c27 jmp 0x7c27
0x7c29 lea 0x0(%esi),%esi
```

remote Thread 1.1 In: L?? PC: 0x7c00

Breakpoint 1 at 0x7c00

(gdb) c

Continuing.

Breakpoint 1, 0x00007c00 in ?? ()

(gdb) x/10x 0x7c00

0x7c00:	0x8ec031fa	0x8ec08ed8	0x16010fd0	0x200f7c44
---------	------------	------------	------------	------------

0x7c10:	0xc88366c0	0xc0220f01	0x087c1dea	0x10b86600
---------	------------	------------	------------	------------

0x7c20:	0x8ed88e00	0xebd08ec0		
---------	------------	------------	--	--

(gdb)

- 大腿同学的代码还可以帮助我们直接定位到错误的指令！

- 只需要大腿的 compiled binary
- 就能指令级定位出错的位置

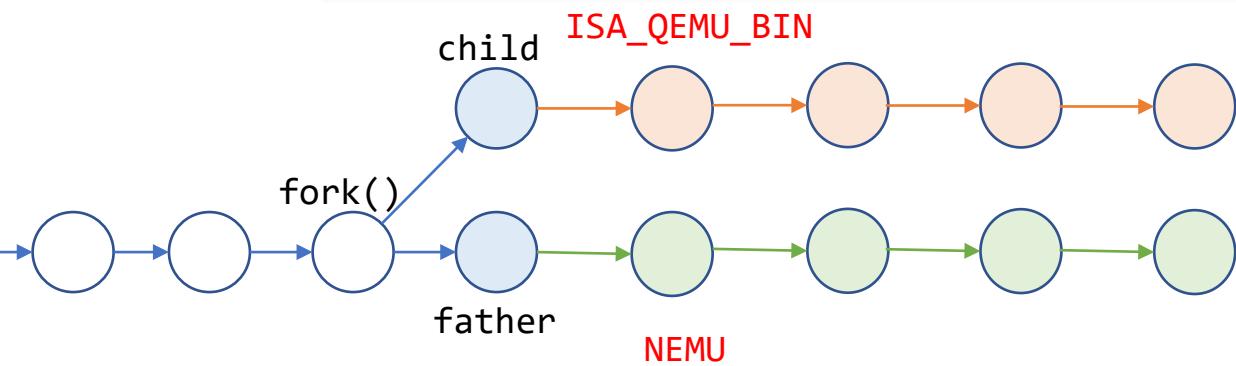
```
for i in range(int(sys.argv[1])):
    print('\n'.join(['si'] + [f'p ${r}' for r in ['eax', ...]]))
```

- 然后找到 log 第一个不一致的地方！

```
N=10000
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
<(python3 cmdgen.py $N | ./x86-nemu img)
```

```
ssfully\n", buf);
```

```
int64_t n) {
i();
```



```
47 bool gdb_getregs(union isa_gdb_regs *r) {
48     gdb_send(conn, (const uint8_t *)"g", 1);
49     size_t size;
50     uint8_t *reply = gdb_recv(conn, &size);
51
52     int i;
53     uint8_t *p = reply;
54     uint8_t c;
55     --- 7 lines: for (i = 0; i < sizeof(union isa_gdb_
```

```
bool gdb_si() {
    char buf[] = "vCont;s:1";
    gdb_send(conn, (const uint8_t *)buf, strlen(buf));
    size_t size;
    uint8_t *reply = gdb_recv(conn, &size);
    free(reply);
    return true;
}
```

# 代码导读 (cont'd)

- 经过 RTFM/RTFSC：
  - `nemu/tools/qemu-diff` 是 differential testing 实际实现的目录
- 然后 RTFSC，看到了若干有用的函数：
  - `gdb_connect_qemu`, 看起来就是用来连接到 QEMU 的，创建一个到 127.0.0.1、端口是 1234 的 gdb 连接
  - `gdb_si`, 和 `monitor` 一样，单步执行指令
  - `gdb_setregs`, `gdb_getregs`, 好像复杂一点，不过就是用 `gdb_send()` 和 `gdb_recv()` 发送/接收消息

# 代码导读

- 首先 PA 代码 (dut.c) 里没有 diff-test 的实际代码，只有一堆函数指针：

```
void (*ref_difftest_memcpy)(paddr_t addr, void *buf, size_t n, bool direction) = NULL;
void (*ref_difftest_regcpy)(void *dut, bool direction) = NULL;
void (*ref_difftest_exec)(uint64_t n) = NULL;
void (*ref_difftest_raise_intr)(uint64_t NO) = NULL;
```

- 这些函数封装了参考实现的功能
  - 既可以和 QEMU diff-test，也可以和 NEMU diff-test
  - exec\_wrapper() 中执行一条指令之后直接对比结果就行

```
#if defined(DIFF_TEST)
    difftest_step(ori_pc, cpu.pc);
#endif
```

# 总结

# 系统编程的困难

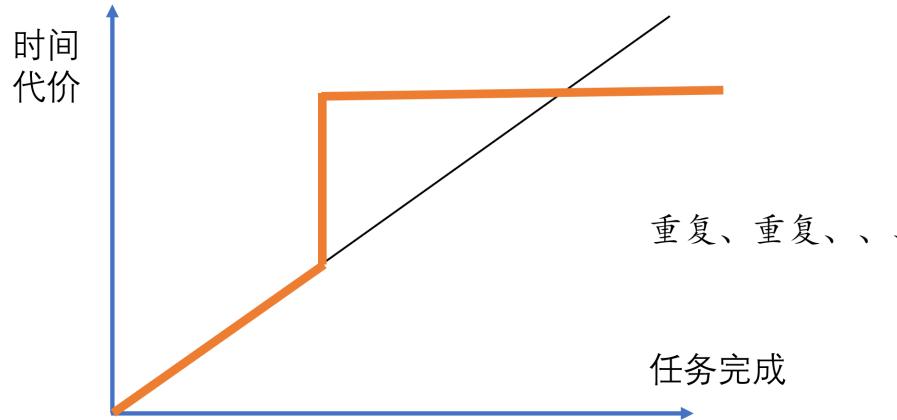
- 在程序规模到达一定程度的时候，代码既难管理，也难写对
  - 即便在项目文件之间浏览就已经非常耗时
    - 面对不熟悉的模块/API
    - 需要好的 IDE、代码折叠、第二块屏幕……
  - 编程经验可以减少 bug，但很难完全消灭它们
    - 各类肉眼难以发现的低级错误 (`i` vs. `j`, `0x` vs `0b`, `int` with `unsigned`, ...)
    - 读错了手册 (忘记更新某个 `flags`, 记错 0/符号扩展, ...)
    - 逻辑上的错误 (使用一块已经释放的内存, 虚拟/物理内存地址访问错, ...)
- 做过 PA 的人就知道，“机器永远是对的” 不是开玩笑的
  - 你折腾一天，两天，可能就是多打了一个空格

# 基础设施：帮助你更快更好地生产代码

- 终极梦想：让计算机自动帮我们写程序
  - 告诉计算机需求 → 计算机输出正确的代码
  - 计算机科学的 holy grail 之一
- 现在我们还在软件自动化的初级阶段
  - 计算机只能提供有限的自动化 (基础设施)
    - 集成开发环境 (IDE)
    - 静态分析
    - 动态分析
  - 但这些基础设施已经从本质上改变了我们的开发效率
    - 你绝对不会愿意用记事本写程序的

# 基础设施：小结

- 当轮子都不够用的时候，我们就去造轮子
  - 调试困难，有参考实现 → diff-test
  - 难以贯通多门实验课 → Abstract Machine



- 轮子还不够用呢？
  - diff-test 每秒只能检查 ~5000 条指令
  - 中断和 I/O 具有不确定性
  - 没有参考实现呢 → 一个新的研究问题在等着你

End.

# 优化程序性能选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 概述

---

- 程序的性能优化
  - 算法的优化
  - 代码的优化
  - 指令的优化
- 一般practice
  - 选择适合的算法和数据结构
    - 算法课
  - 编写能够被编译器有效优化并转化为高效的可执行代码的源码
    - 理解编译器优化的能力和局限

# 理解编译器优化的能力和局限性

---

- 理解程序如何编译+执行
- 理解现代处理器和存储系统
- 如何诊断性能瓶颈和提高性能
- 编译器优化可以做什么?
  - 建立程序到机器的有效映射
    - 分配寄存器
    - 代码筛选和调度
    - 死代码消除
    - 简单低效消除
  - Optimization blockers

# 常见的有效优化：减少计算操作的次数频率

- 不考虑处理器或编译器之外的优化方式
- 减少计算操作的次数频率
  - 保持程序行为总是一致的前提
  - 尤其对循环中的操作

```
void set_row(double *a,  
double *b, long i, long n){  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```



```
void set_row(double *a,  
double *b, long i, long n){  
    long j;  
    int ni = n * i;  
    for (j = 0; j < n; j++)  
        a[ni+j] = b[j];  
}
```

# GCC –O1

```
void set_row(double *a, double *b,  
long i, long n){  
    long j;  
    for (j = 0; j < n; j++)  
        a[n*i+j] = b[j];  
}
```

```
long j;  
long ni = n * i;  
double *rowp = a + ni;  
for (j = 0; j < n; j++)  
    *rowp++ = b[j];
```

```
5 set_row:  
6 .LFB0:  
7     .cfi_startproc  
8     endbr64  
9     testq %rcx, %rcx  
10    jle .L1  
11    imulq %rcx, %rdx  
12    leaq  (%rdi,%rdx,8), %rdx  
13    movl $0, %eax  
14 .L3:  
15    movsd (%rsi,%rax,8), %xmm0  
16    movsd %xmm0, (%rdx,%rax,8)  
17    addq $1, %rax  
18    cmpq %rax, %rcx  
19    jne .L3  
20 .L1:  
21    ret
```

```
# Test n  
# If 0, goto done  
# ni = n * I  
# rowp = A + ni * 8  
# j = 0  
# loop:  
# t = b[j]  
# M[A+ni*8+j*8] = t  
# j ++  
# if != , goto loop  
# done:
```

# 常见的有效优化：强度降低

- 用简单操作代替复杂操作
- Shift, add代替乘除
  - $16*x \rightarrow x << 4$

```
for (i = 0; i < n; i++){  
    int ni = n * i;  
    for (j = 0; j < n; j++)  
        a[ni+j] = b[j];  
}
```



```
int ni = 0;  
for (i = 0; i < n; i++){  
    for (j = 0; j < n; j++)  
        a[ni+j] = b[j];  
    ni += n;  
}
```

# 常见的有效优化：子表达式的复用

```
/*sum neighbors of i,j*/
up = val[ (i-1)*n + j ];
down = val[ (i+1)*n + j ];
left = val[ i*n + j-1 ];
right = val[ i*n + j+1 ];
sum = up + down + left + right;
```

```
6 sum_func:
7 .LFB0:
8 .cfi_startproc
9 endbr64
10 subl $1, %edi
11 movq val(%rip), %rcx
12 movl %esi, %eax
13 imull %edx, %edi
14 addl %edi, %esi
15 movslq %esi, %rsi
16 movsd (%rcx,%rsi,8), %xmm0
17 leal (%rdi,%rdx,2), %esi
18 leal (%rsi,%rax), %edi
19 subl %edx, %esi
20 movslq %edi, %rdi
21 movsd %xmm0, up(%rip)
22 addl %eax, %esi
23 movsd (%rcx,%rdi,8), %xmm3
24 movslq %esi, %rsi
25 addsd %xmm3, %xmm0
26 movsd %xmm3, down(%rip)
27 movsd -8(%rcx,%rsi,8), %xmm2
28 movsd %xmm2, left(%rip)
29 movsd 8(%rcx,%rsi,8), %xmm1
30 addsd %xmm2, %xmm0
31 movsd %xmm1, right(%rip)
32 addsd %xmm1, %xmm0
33 movsd %xmm0, sum(%rip)
34 ret
```

```
/*sum neighbors of i,j*/
long inj = i * n + j;
up = val[ inj - n ];
down = val[ inj + n ];
left = val[ inj - 1 ];
right = val[ inj + 1 ];
sum = up+down+left+right;
```

```
5 sum_func:
6 .LFB0:
7 .cfi_startproc
8 endbr64
9 imull %edx, %edi
10 addl %esi, %edi
11 movslq %edi, %rsi
12 movq val(%rip), %rax
13 movslq %edx, %rdx
14 movq %rsi, %rcx
15 subq %rdx, %rcx
16 movsd (%rax,%rcx,8), %xmm1
17 movsd %xmm1, up(%rip)
18 addq %rsi, %rdx
19 movsd (%rax,%rdx,8), %xmm3
20 movsd %xmm3, down(%rip)
21 movsd -8(%rax,%rsi,8), %xmm2
22 movsd %xmm2, left(%rip)
23 movsd 8(%rax,%rsi,8), %xmm0
24 movsd %xmm0, right(%rip)
25 addsd %xmm3, %xmm1
26 addsd %xmm2, %xmm1
27 addsd %xmm1, %xmm0
28 movsd %xmm0, sum(%rip)
29 ret
```

# 我们的目标

---

- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 编写适合编译优化的源码
  - [GCC online documentation - GNU Project](#)
    - -O0
    - -O1
    - -O2
    - -O3
    - -Os
    - -Ofast
    - -Og

# GCC的优化选项

管理 控制 视图 热键 设备 帮助

```
$ gcc -Q -O1 --help=optimizers
```

英 拼

Right Shift + Right Alt

## GCC online documentation - GNU Project

-01

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With ‘-O’, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

‘-O’ turns on the following optimization flags:

- fauto-inc-dec
- fbranch-count-reg
- fcombine-stack-adjustments
- fcompare-elim
- fcprop-registers
- fdce
- fdefer-pop
- fdelayed-branch
- fdse

- fforward-propagate
- fguess-branch-probability
- fif-conversion
- fif-conversion2
- finline-functions-called-once
- fipa-profile
- fipa-pure-const
- fipa-reference
- fipa-reference-addressable
- fmerge-constants
- fmove-loop-invariants
- fomit-frame-pointer
- freorder-blocks
- fshrink-wrap
- fshrink-wrap-separate
- fsplit-wide-types
- fssa-backprop
- fssa-phiopt
- ftree-bit-ccp
- ftree-ccp
- ftree-ch
- ftree-coalesce-vars
- ftree-copy-prop
- ftree-dce
- ftree-dominator-opts
- ftree-dse
- ftree-forwprop
- ftree-fre
- ftree-phiprop
- ftree-pta
- ftree-scev-cprop
- ftree-sink
- ftree-slsr
- ftree-sra
- ftree-ter
- funit-at-a-time

## GCC online documentation - GNU Project

-02

Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to ‘-O’, this option increases both compilation time and the performance of the generated code.

‘-O2’ turns on all optimization flags specified by ‘-O’. It also turns on the following optimization flags:

- falign-functions -falign-jumps
- falign-labels -falign-loops
- fcaller-saves
- fcode-hoisting
- fcrossjumping
- fcse-follow-jumps -fcse-skip-blocks
- fdelete-null-pointer-checks
- fdevirtualize -fdevirtualize-speculatively
- fexpensive-optimizations
- ffinite-loops
- fgcse -fgcse-lm
- fhoist-adjacent-loads
- finline-functions
- finline-small-functions
- findirect-inlining

- fipa-bit-cp -fipa-cp -fipa-icf
- fipa-ra -fipa-sra -fipa-vrp
- fisolate-erroneous-paths-dereference
- flra-remat
- foptimize-sibling-calls
- foptimize-strlen
- fpartial-inlining
- fpeephole2
- freorder-blocks-algorithm=stc
- freorder-blocks-and-partition -freorder-functions
- frerun-cse-after-loop
- fschedule-insns -fschedule-insns2
- fsched-interblock -fsched-spec
- fstore-merging
- fstrict-aliasing
- fthread-jumps
- ftree-builtin-call-dce
- ftree-pre
- ftree-switch-conversion -ftree-tail-merge
- ftree-vrp

# 理解编译器优化的能力和局限性

- 理解程序如何编译+执行
- 理解现代处理器和存储系统
- 如何诊断性能瓶颈和提高性能
- 编译器的一些局限，理解编译器的优化痛点
  - 编译器优化前提是safe
    - Within procedure analyses
    - Static information analyses
    - “Conservative” to be “safe”

# 理解编译器优化的痛点

---

- Two optimization blocker
  - Memory aliasing
  - Function calls

# 示例程序1.1

```
void sum_row1(double *a, double *b, long n){  
    long i, j;  
    for (i = 0; i<n; i++){  
        b[i] = 0;  
        for (j = 0; j <n; j++)  
            b[i] += a[i*n+j];  
    }  
}
```

```
21 .L3:  
22    movsd (%rdx), %xmm0  
23    addsd (%rax), %xmm0  
24    movsd %xmm0, (%rdx)  
25    addq $8, %rax  
26    cmpq %rcx, %rax  
27    jne .L3
```

## B数组

```
double A[9] =  
{ 0, 1, 2,  
 4, 8, 16,  
 32, 64, 128};  
  
double *B = A+3;  
  
sum_row1(A, B, 3)
```

init: [4, 8, 16]

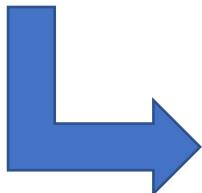
i=0: [3, 8, 16]

i=1: [3, 22, 16]

i=2: [3, 22, 224]

# 示例程序1.1

```
void sum_row1(double *a, double *b, long n){  
    long i, j;  
    for (i = 0; i<n; i++){  
        b[i] = 0;  
        for (j = 0; j <n; j++)  
            b[i] += a[i*n+j];  
    }  
}
```



```
void sum_row1(double *a, double *b, long n){  
    long i, j;  
    for (i = 0; i<n; i++){  
        double val = 0;  
        for (j = 0; j <n; j++)  
            val += a[i*n+j];  
        b [i] = val;  
    }  
}
```

# 示例程序1.2

```
//a.c
#include <stdio.h>
void f1 (int *xp, int *yp){
    *xp += *yp;
    *xp += *yp;
}
```

gcc -O1 a.c

```
$ objdump -d a.o

a.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f1>:
 0:  f3 0f 1e fa          endbr64
 4:  8b 06                mov    (%rsi),%eax
 6:  03 07                add    (%rdi),%eax
 8:  89 07                mov    %eax,(%rdi)
 a:  03 06                add    (%rsi),%eax
 c:  89 07                mov    %eax,(%rdi)
 e:  c3                  ret
```

```
//b.c
#include <stdio.h>
void f1 (int *xp, int *yp){
    *xp += 2* *yp;
}
```

gcc -O1 b.c

```
$ objdump -d b.o

b.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <f1>:
 0:  f3 0f 1e fa          endbr64
 4:  8b 06                mov    (%rsi),%eax
 6:  01 c0                add    %eax,%eax
 8:  01 07                add    %eax,(%rdi)
 a:  c3                  ret
```

如果xp和yp指向同一块内存地址?

# 示例程序1.2

```
//a.c  
#include <stdio.h>  
void f1 (int *xp, int *yp){  
    *xp += *yp;  
    *xp += *yp;  
}
```

```
//b.c  
#include <stdio.h>  
void f1 (int *xp, int *yp){  
    *xp += 2* *yp;  
}
```

如果xp和yp指向同一块内存地址?

```
*xp += *xp;  
*xp += *xp;
```

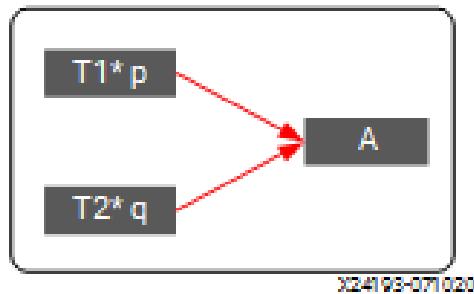
```
*xp = 4 * *xp
```

```
*xp += 2* *xp;
```

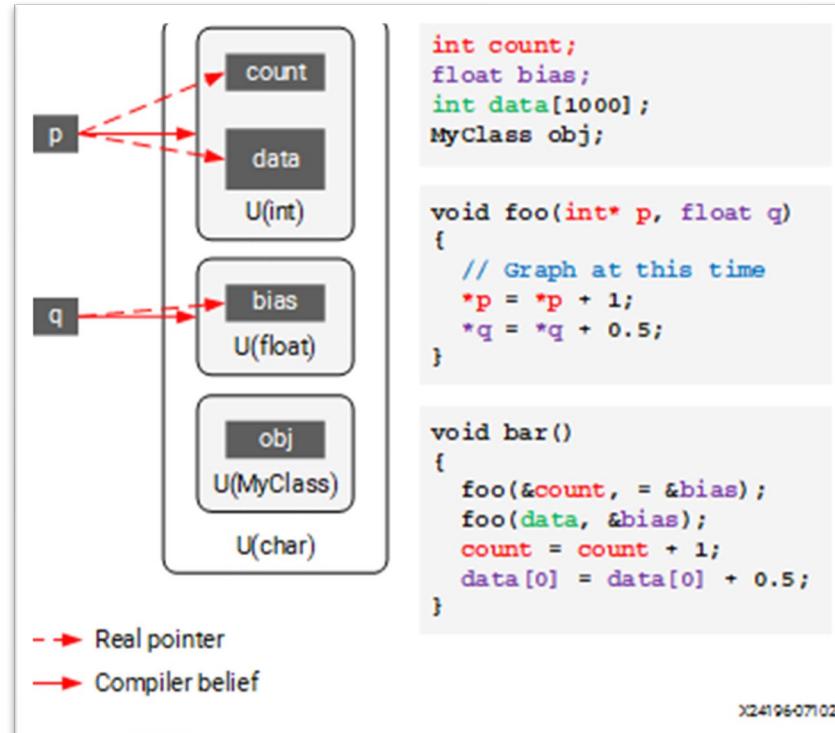
```
*xp = 3 * *xp
```

Memory aliasing

# Memory aliasing & strict aliasing rules



```
//a.c
#include <stdio.h>
void f1 (int *xp, int *yp){
    *xp += *yp;
    *xp += *yp;
}
```



```

// C program to illustrate aliasing
#include <stdio.h>

// Function to change the value of
// ptr1 and ptr2
int foo(int* ptr1, int* ptr2)
{
    *ptr1 = 10;
    *ptr2 = 11;
    return *ptr1;
}

// Driver Code
int main()
{
    int data1 = 10, data2 = 20;

    // Function Call
    int result = foo(&data1, &data2);

    // Print result
    printf("%d ", result);
    return 0;
}

```

Disassembly of section .text:

0000000000000000 <foo>:	
0: f3 0f 1e fa	endbr64
4: c7 07 0a 00 00 00	movl \$0xa, (%rdi)
a: c7 06 0b 00 00 00	movl \$0xb, (%rsi)
10: 8b 07	mov (%rdi), %eax
12: c3	ret

gcc -O1 a.c

gcc -O2 a.c

# Memory aliasing引发不同的优化结果

- What to produce by GCC-O1, GCC-O2 optimizers?

```
3 #include <stdio.h>
4
5 // Function to change the value of
6 // ptr1 and ptr2
7 int foo(int* ptr1, long* ptr2)
8 {
9     *ptr1 = 10;
10    *ptr2 = 11.0;
11    return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
24     return 0;
25 }
```

```
1 // C program to illustrate aliasing
2 // Function to change the value of
3 // ptr1 and ptr2
4 int foo(int* ptr1, long* ptr2)
5 {
6     *ptr1 = 10;
7     *ptr2 = 11.0;
8     return *ptr1;
9 }
10 // Driver Code
11 int main()
12 {
13     long data = 100.0;
14     // Function Call
15     int result = foo((int *)&data, &data);
16
17     // Print result
18     printf("%d \n", result);
19
<ts/ICS2021/teach/Course17/demo.c[1] [c] unix utf-8 Ln 1, Col 0/25>
```

```

3 #include <stdio.h>
4
5 // Function to change the value of
6 // ptr1 and ptr2
7 int foo(int* ptr1, long* ptr2)
8 {
9     *ptr1 = 10;
10    *ptr2 = 11.0;
11    return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
24     return 0;
25 }

```

0000000000001149 <foo>:

1149:	f3 0f 1e fa	endbr64
114d:	c7 07 0a 00 00 00	movl \$0xa,(%rdi)
1153:	48 c7 06 0b 00 00 00	movq \$0xb,(%rsi)
115a:	8b 07	mov (%rdi),%eax
115c:	c3	ret

gcc -O1 a.c

0000000000001180 <foo>:

1180:	f3 0f 1e fa	endbr64
1184:	c7 07 0a 00 00 00	movl \$0xa,(%rdi)
118a:	b8 0a 00 00 00	mov \$0xa,%eax
118f:	48 c7 06 0b 00 00 00	movq \$0xb,(%rsi)
1196:	c3	ret

gcc -O2 a.c

-fstack-reuse=[all named_vars none]	all
-fstdarg-opt	[enabled]
-fstore-merging	[enabled]
-fstrict-aliasing	[enabled]
-fstrict-enums	[available in C++, ObjC++]
-fstrict-volatile-bitfields	[enabled]
-fthread-jumps	[enabled]

1200:	41 5e	pop %r14
1202:	41 5f	pop %r15
1204:	c3	ret
1205:	66 66 2e 0f 1f 84 00	data16 cs nopw 0x0(%rax,%ra
x,1)		
120c:	00 00 00 00	
0000000000001210 <_libc_csu_fini>:		
1210:	f3 0f 1e fa	endbr64
1214:	c3	ret

Disassembly of section .fini:

0000000000001218 <_fini>:		
1218:	f3 0f 1e fa	endbr64
121c:	48 83 ec 08	sub \$0x8,%rsp
1220:	48 83 c4 08	add \$0x8,%rsp
1224:	c3	ret

\$ gcc -O1 demo.c

\$ ./a.out

11

\$ gcc -O2 demo.c

\$ ./a.out

10

\$ gcc -O2 demo.c █

```

3 #include <stdio.h>
4
5 // Function to change the value of
6 // ptr1 and ptr2
7 int foo(int* ptr1, long* ptr2)
8 {
9     *ptr1 = 10;
10    *ptr2 = 11.0;
11    return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
24     return 0;
25 }

```

gcc -O2 a.c -fno-strict-aliasing

```

0000000000001149 <foo>:
1149:    f3 0f 1e fa          endbr64
114d:    c7 07 0a 00 00 00      movl   $0xa,(%rdi)
1153:    48 c7 06 0b 00 00 00  movq   $0xb,(%rsi)
115a:    8b 07
115c:    c3                   mov    (%rdi),%eax
                                ret

```

gcc -O1 a.c

```

0000000000001180 <foo>:
1180:    f3 0f 1e fa          endbr64
1184:    c7 07 0a 00 00 00      movl   $0xa,(%rdi)
118a:    b8 0a 00 00 00        mov    $0xa,%eax
118f:    48 c7 06 0b 00 00 00  movq   $0xb,(%rsi)
1196:    c3                   ret

```

gcc -O2 a.c

-fstack-reuse=[all named_vars none]	all
-fstdarg-opt	[enabled]
-fstore-merging	[enabled]
-fstrict-aliasing	[enabled]
-fstrict-enums	[available in C++, ObjC++]
-fstrict-volatile-bitfields	[enabled]
-fthread-jumps	[enabled]

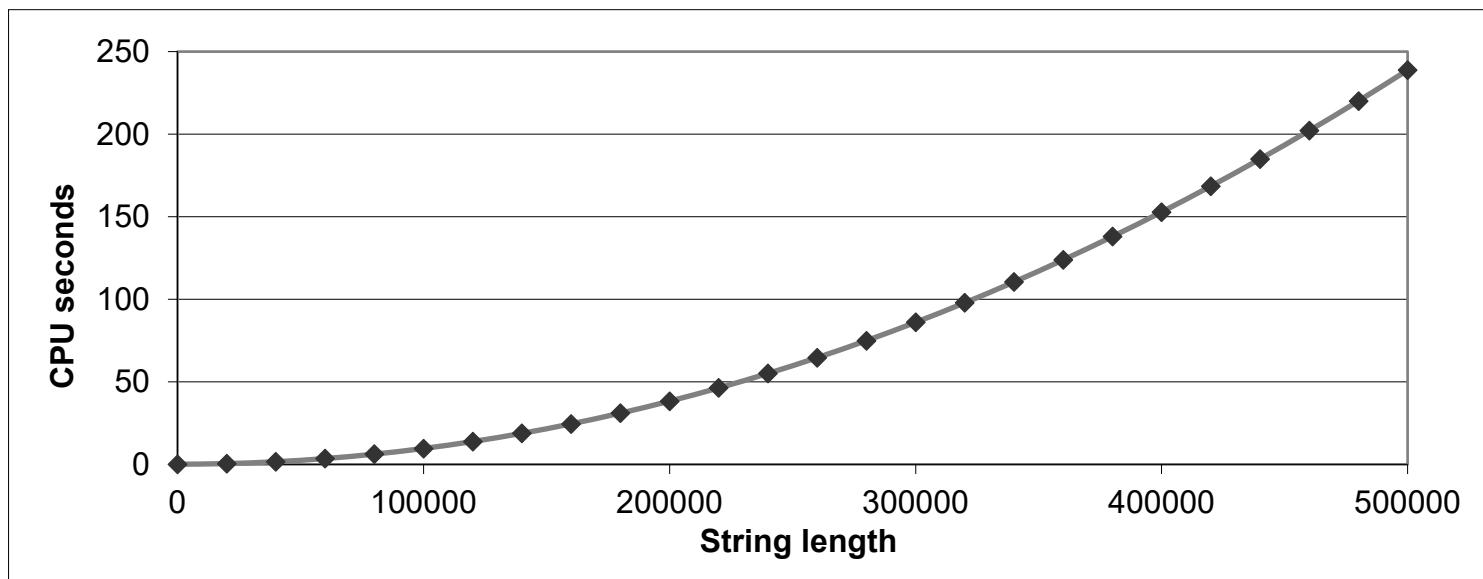
# Optimization blocker: memory aliasing

---

- Aliasing
  - 两个不同的内存reference可能指向同一块内存
  - C语言中常见
    - C允许地址操作
  - 合适地引入局部变量
    - 函数内部
    - 开发时消除aliasing

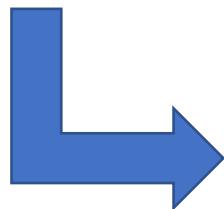
# 示例程序2.1

```
void lower(char *s){  
    size_t i;  
    for (i = 0; i<strlen(s); i++)  
        if(s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```



# 示例程序2.1

```
void lower(char *s){  
    size_t i;  
    for (i = 0; i<strlen(s); i++)  
        if(s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

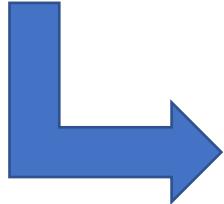


- `strlen()`: 找结束符
  - 线性复杂度
  - N次进入循环, 二次复杂度

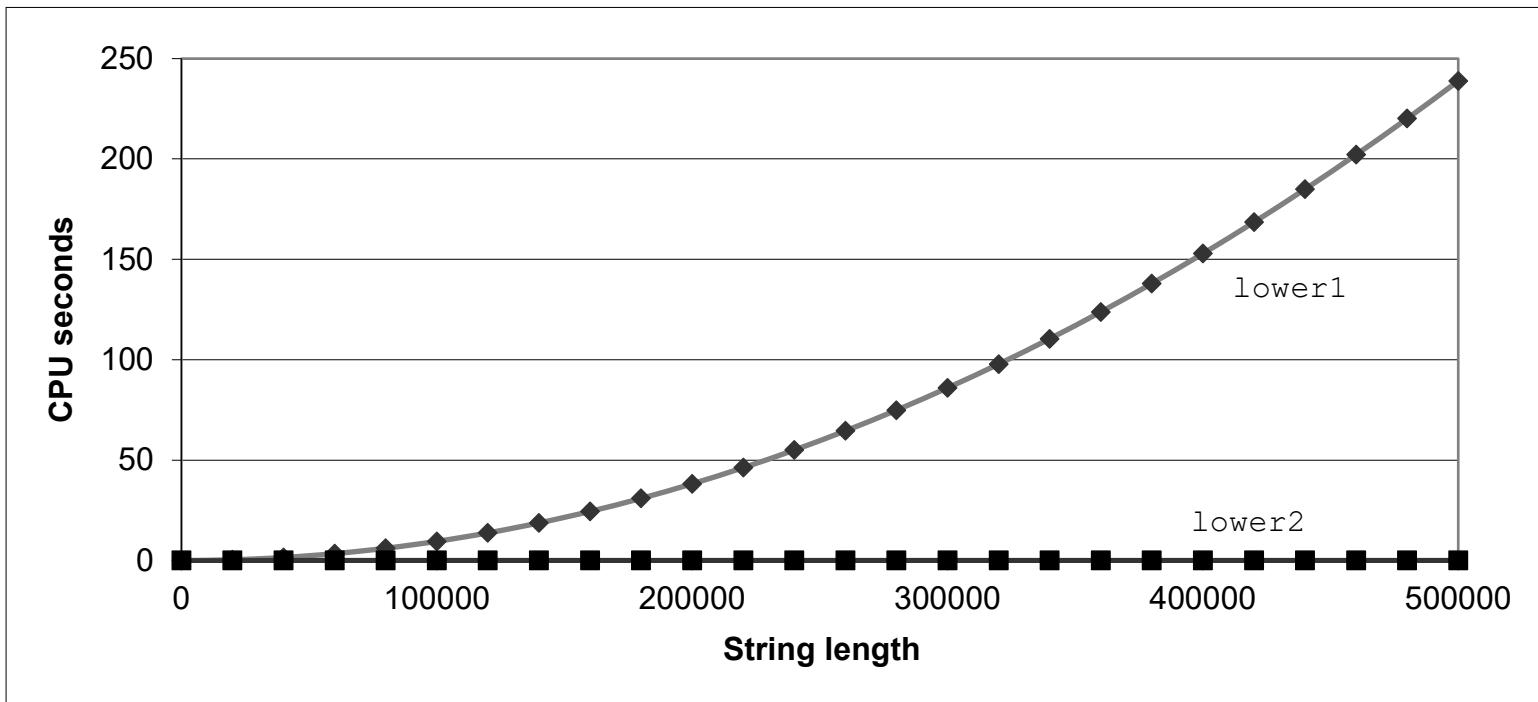
```
void lower(char *s){  
    size_t i = 0;  
    if (i >= strlen(s))  
        goto done;  
loop:  
    if(s[i] >= 'A' && s[i] <= 'Z')  
        s[i] -= ('A' - 'a');  
    i++;  
    if(i< strlen(s))  
        goto loop;  
done:  
}
```

# 示例程序2.1

```
void lower1(char *s){  
    size_t i;  
    for (i = 0; i<strlen(s); i++)  
        if(s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```



```
void lower2(char *s){  
    size_t i;  
    size_t len = strlen(s);  
    for (i = 0; i<len; i++)  
        if(s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```



# 思考

- 为什么编译器不能将`strlen`自动优化出循环?
  - 过程调用的副作用
    - 全局变量
    - 每次调用返回值不一定一样
- 编译器一般会将其作为黑盒处理
  - Linking决定具体函数调用的实现

## • 其他措施

- Inline functions
- 更好的coding

```
void lower1(char *s){  
    size_t i;  
    for (i = 0; i<strlen(s); i++)  
        if(s[i] >= 'A' && s[i] <= 'Z')  
            s[i] -= ('A' - 'a');  
}
```

# 示例程序2.2

```
long f();
```

```
long func1(){
    return f() + f() + f() + f();
}
```

```
long func2(){
    return 4 * f();
}
```

```
func2:
endbr64
subq $8, %rsp
xorl %eax, %eax
call f@PLT
addq $8, %rsp
salq $2, %rax
ret
```

```
func1:
endbr64
pushq %rbx
xorl %eax, %eax
call f@PLT
movq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rbx, %rax
popq %rbx
ret
```

为了safe地优化，编译器考虑函数调用  
可能存在副作用（side effect）

# 示例程序2.2

```
long f();
```

```
long func1(){
    return f() + f() + f() + f();
}
```

counter = 6

```
long func2(){
    return 4 * f();
}
```

counter = 0

func2:

```
endbr64
subq $8, %rsp
xorl %eax, %eax
call f@PLT
addq $8, %rsp
salq $2, %rax
ret
```

```
func1:
endbr64
pushq %rbx
xorl %eax, %eax
call f@PLT
movq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rax, %rbx
xorl %eax, %eax
call f@PLT
addq %rbx, %rax
popq %rbx
ret
```

```
long counter = 0;
long f(){
    return counter++;
}
```

# Inlin

-findirect-inlining

Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining. This option has any effect only when inlining itself is turned on by the ‘-finline-functions’ or ‘-finline-small-functions’ options.

Enabled at levels ‘-O2’, ‘-O3’, ‘-Os’.

long

```
long func1(){
    return f() + f() + f() + f();
}
```

```
long func2(){
    return 4 * f();
}
```

```
long func1in(){
    long t = counter++;
    t += counter++;
    t += counter++;
    t += counter++;
    return t;
}
```

```
long func1opt(){
    long t = 4 * counter + 6;
    return t;
}
```

\$

S 英 汉 简 拼 \*

Right Shift + Right 34

# Inline substitution

```
long f();  
  
long func1(){  
    return f() + f() + f() + f();  
}  
  
long func2(){  
    return 4 * f();  
}
```

gcc -O0 a.c

gcc -O1 a.c

gcc -O2 a.c

0000000000000000 <func1>:  
0: f3 0f 1e fa endbr64  
4: 55 push %rbp  
5: 48 89 e5 mov %rsp,%rbp  
8: 53 push %rbx  
9: 48 83 ec 08 sub \$0x8,%rsp  
d: b8 00 00 00 00 mov \$0x0,%eax  
12: e8 00 00 00 00 call 17 <func1+0x17>  
17: 48 89 c3 mov %rax,%rbx  
1a: b8 00 00 00 00 mov \$0x0,%eax  
1f: e8 00 00 00 00 call 24 <func1+0x24>  
24: 48 01 c3 add %rax,%rbx  
27: b8 00 00 00 00 mov \$0x0,%eax  
2c: e8 00 00 00 00 call 31 <func1+0x31>  
31: 48 01 c3 add %rax,%rbx  
34: b8 00 00 00 00 mov \$0x0,%eax  
39: e8 00 00 00 00 call 3e <func1+0x3e>  
3e: 48 01 d8 add %rbx,%rax  
41: 48 8b 5d f8 mov -0x8(%rbp),%rbx  
45: c9 leave  
46: c3 ret

0000000000000000 <func1>:  
0: f3 0f 1e fa endbr64  
4: 48 8b 05 00 00 00 00 mov 0x0(%rip),%rax # b <func1+0xb>  
b: 48 8d 50 04 lea 0x4(%rax),%rdx  
f: 48 8d 04 85 06 00 00 lea 0x6(%rax,4),%rax  
16: 00  
17: 48 89 15 00 00 00 00 mov %rdx,0x0(%rip) # 1e <func1+0x1e>  
1e: c3 ret  
1f: 90 nop

管理 控制 视图 热键 设备 帮助

\$ gcc -O2 -c demo2.c

S 英 汉 简 拼 \*

Right Shift + Right Alt

# Inline substitution

```
long f();  
  
long func1(){  
    return f() + f() + f() + f();  
}  
  
long func2(){  
    return 4 * f();  
}
```

gcc -O0 a.c

gcc -O1 a.c

gcc -O2 a.c

gcc -O2 a.c -fno-inline

```
0000000000000000 <func1>:  
0: 48 89 e5          mov    %rsp,%rbp  
4: 53                push   %rbx  
8: 48 83 ec 08       sub    $0x8,%rsp  
d: b8 00 00 00 00     mov    $0x0,%eax  
12: e8 00 00 00 00    call   17 <func1+0x17>  
17: 48 89 c3          mov    %rax,%rbx  
1a: b8 00 00 00 00    mov    $0x0,%eax  
1f: e8 00 00 00 00    call   24 <func1+0x24>  
24: 48 01 c3          add    %rax,%rbx  
27: b8 00 00 00 00    mov    $0x0,%eax  
2c: e8 00 00 00 00    call   31 <func1+0x31>  
31: 48 01 c3          add    %rax,%rbx  
34: b8 00 00 00 00    mov    $0x0,%eax  
39: e8 00 00 00 00    call   3e <func1+0x3e>  
3e: 48 01 d8          add    %rbx,%rax  
41: 48 8b 5d f8       mov    -0x8(%rbp),%rbx  
45: c9                leave  
46: c3                ret
```

```
0000000000000000 <func1>:  
0: f3 0f 1e fa        endbr64  
4: 48 8b 05 00 00 00 00  mov    0x0(%rip),%rax      # b <func1+0xb>  
b: 48 8d 50 04        lea    0x4(%rax),%rdx  
f: 48 8d 04 85 06 00 00  lea    0x6(%rax,4),%rax  
16: 00  
17: 48 89 15 00 00 00 00  mov    %rdx,0x0(%rip)      # 1e <func1+0x1e>  
1e: c3                ret  
1f: 90                nop
```

```
1 #include <stdio.h>
2 long counter = 0;
3 static inline long f(){ return counter ++;
4 }
5
6 long func1(){
7     return f() + f() + f() + f();
8 }
9
10 long func2(){
11     return 4 * f();
12 }
13 int main(){
14     printf("%ld \n", func1());
15 }
16 }
```

```
1 #include <stdio.h>
2 long counter = 0;
3 static inline long f(){ return counter++;
4 }
5
6 long func1(){
7     return f() + f() + f() + f();
8 }
9
10 long func2(){
11     return 4 * f();
12 }
13 int main(){
14     printf("%ld\n", func1());
15
16 }
```

~/Documents/ICS2021/teach/Course17/demo2.c[1]  
1 change; before #1 3 seconds ago

[c] unix utf-8 Ln 1, Col 1/16

# Optimization blocker: function call

---

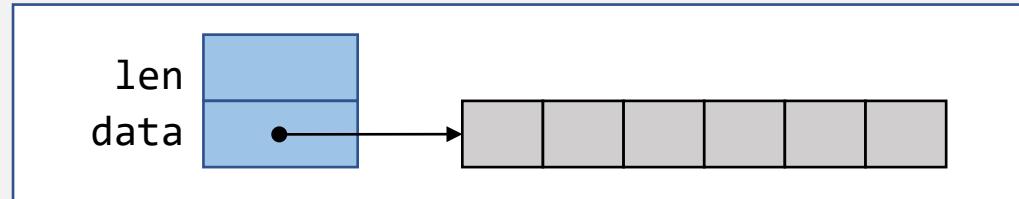
- Function call
  - 过程调用的副作用
    - 全局变量
  - 每次调用返回值不一定一样
- 编译器一般会将其作为黑盒处理
  - Linking决定具体函数调用的实现
- 其他措施
  - Inline functions
  - 更好的coding

# 示例程序3

```
typedef struct{  
    long len;  
    data_t *data;  
} vec_rec, *vec_ptr;
```

```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    *dest = 0;  
    for (i =0; i < vec_length(v); i++){  
        data_t val;  
        get_val_element(v, i, &val);  
        *dest = *dest + val;  
    }  
}
```

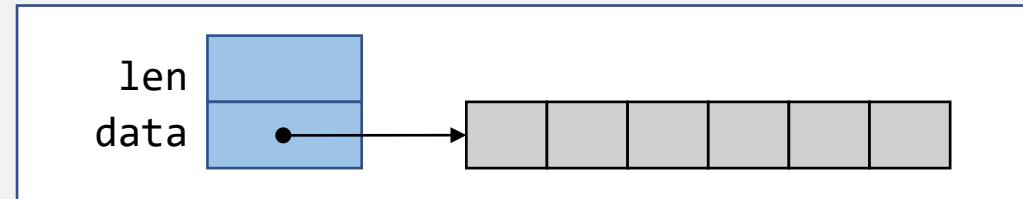


1

- 消除循环低效

# 示例程序3

```
typedef struct{  
    long len;  
    data_t *data;  
} vec_rec, *vec_ptr;
```



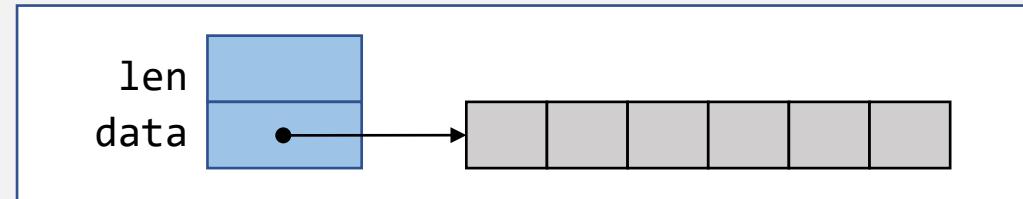
```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    *dest = 0; 1  
    for (i =0; i < length; i++){  
        data_t val;  
        get_val_element(v, i, &val);  
        *dest = *dest + val;  
    }  
}
```

- 消除循环低效

# 示例程序3

```
typedef struct{  
    long len;  
    data_t *data;  
} vec_rec, *vec_ptr;
```



```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    *dest = 0;  
    for (i =0; i < length; i++){  
        data_t val;  
        get_val_element(v, i, &val);  
        *dest = *dest + val;  
    }  
}
```

- 消除循环低效
- 减少函数调用

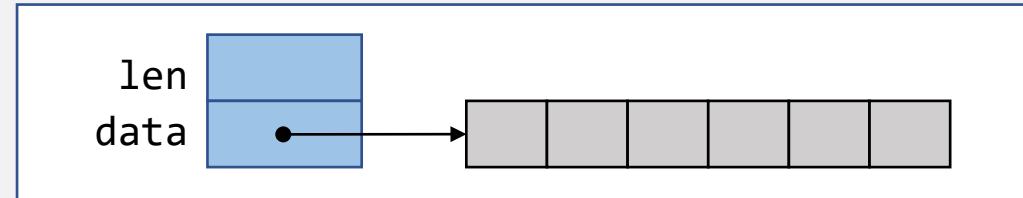
2

# 示例程序3

```
typedef struct{  
    long len;  
    data_t *data;  
} vec_rec, *vec_ptr;
```

```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v); 2  
    data_t *data = get_vec_start(v);  
    *dest = 0;  
    for (i = 0; i < length; i++){  
        *dest = *dest + data[i];  
    }  
}
```



- 消除循环低效
- 减少函数调用

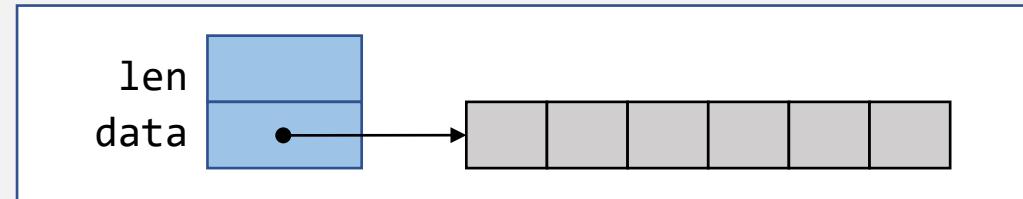
# 示例程序3

```
typedef struct{
    long len;
    data_t *data;
} vec_rec, *vec_ptr;
```

```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++){
        *dest = *dest + data[i];
    }
}
```

3



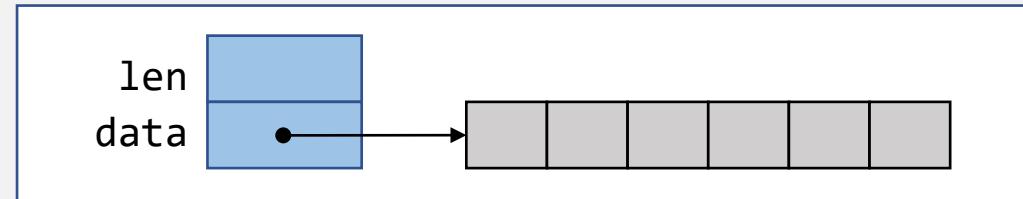
.L3:

```
movq (%rax), %rdx
addq %rdx, (%rbx)
addq $8, %rax
cmpq %rcx, %rax
jne .L3
```

- 消除循环低效
- 减少函数调用
- 减少无需的内存访问

# 示例程序3

```
typedef struct{  
    long len;  
    data_t *data;  
} vec_rec, *vec_ptr;
```



```
typedef long data_t;
```

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    data_t *data = get_vec_start(v);  
    data_t acc = 0;  
    for (i = 0; i < length; i++){  
        acc = acc + data[i];  
    }  
    *dest = acc;  
}
```

```
.L3:  
    addq (%rax), %rdx  
    addq $8, %rax  
    cmpq %rcx, %rax  
    jne .L3
```

- 消除循环低效
- 减少函数调用
- 减少无需的内存访问

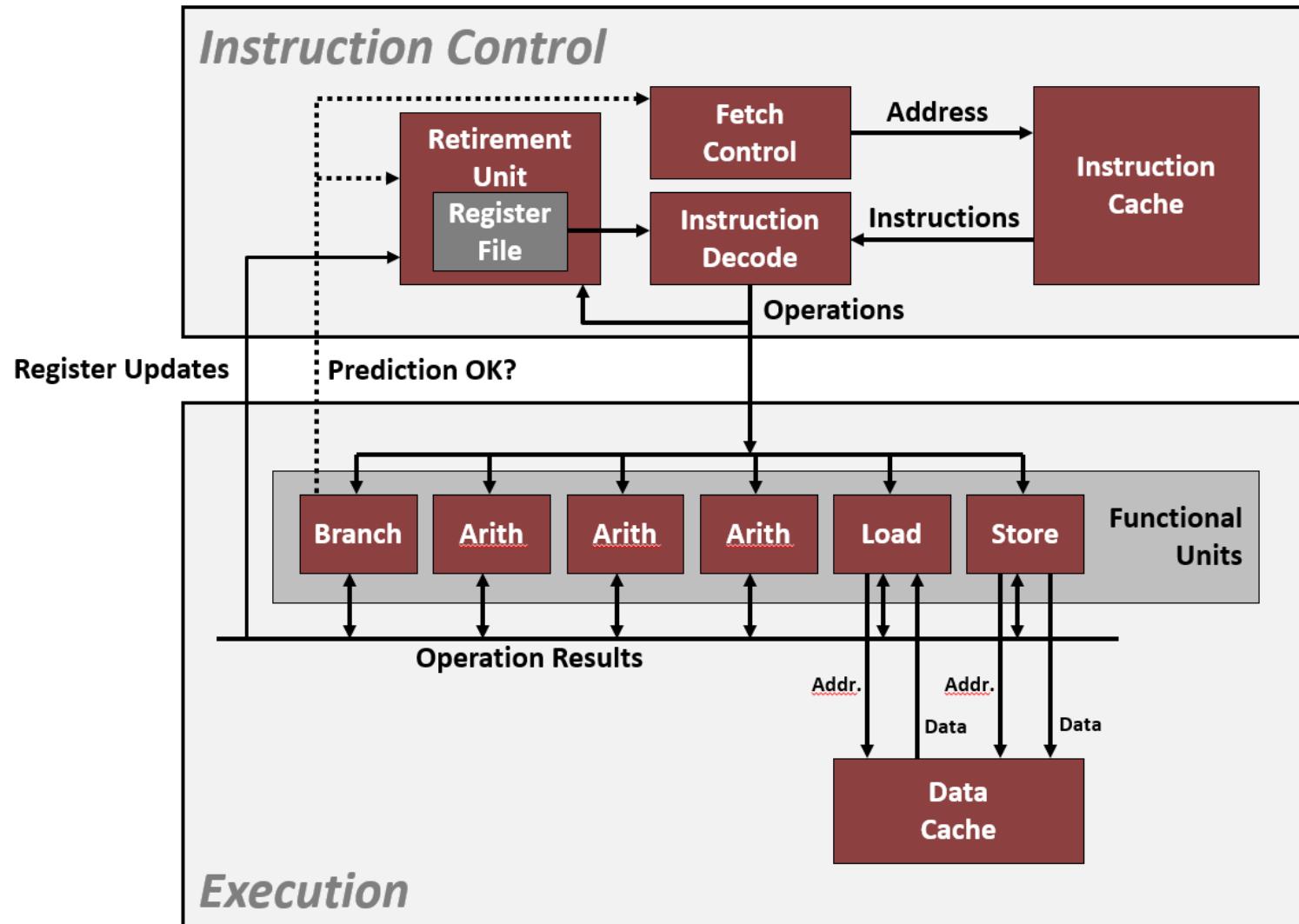
3

# 我们能做的事情

---

- 简单的优化treatment
  - 养成习惯是重要的
- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 理解编译器的优化痛点
    - 编译器优化前提是safe
  - 编写适合编译优化的源码
    - Memory aliasing
    - Function call

# 现代处理器

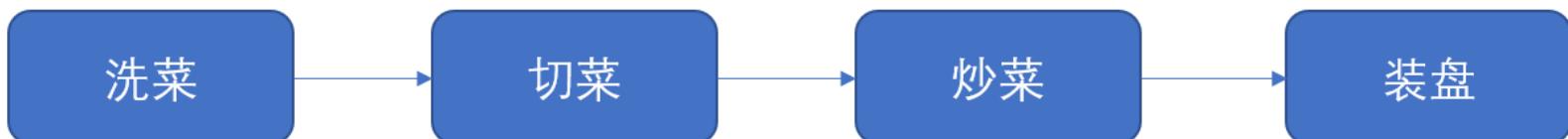


From CMU CSAPP Course

# Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

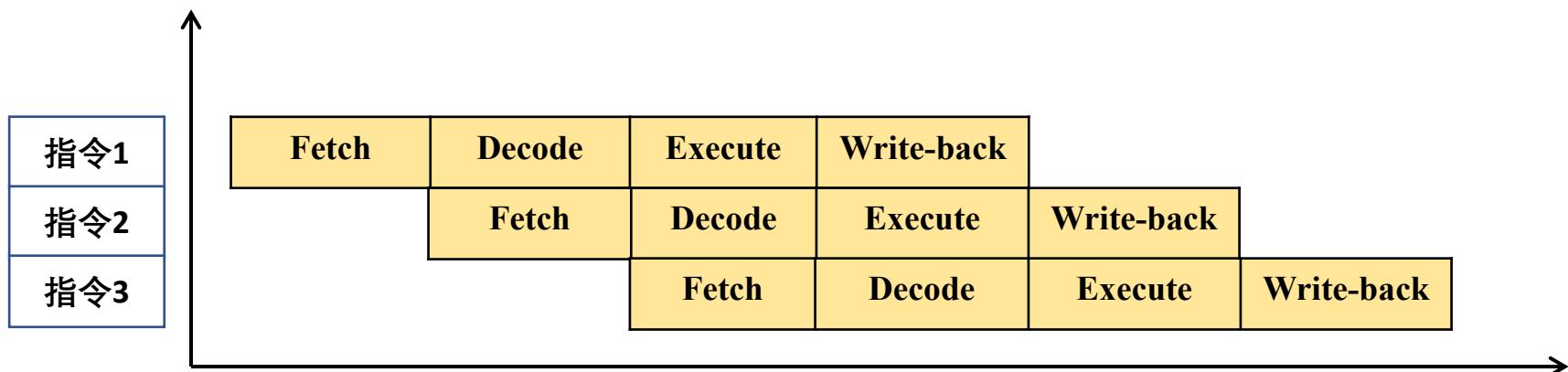
```
long mult_eq(long a, long b, long c){  
    long p1 = a * b;  
    long p2 = a * c;  
    long p3 = p1 * p2;  
    return p3;  
}
```



# Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

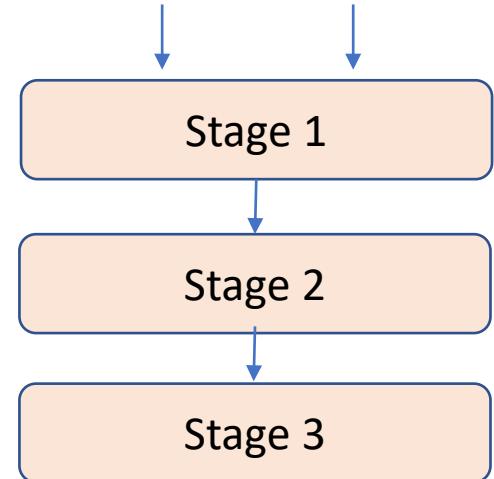
```
long mult_eq(long a, long b, long c){  
    long p1 = a * b;  
    long p2 = a * c;  
    long p3 = p1 * p2;  
    return p3;  
}
```



# Pipeline

- CPU中有一条以上的流水线
  - 每时钟周期内可以完成一条以上的指令

```
long mult_eq(long a, long b, long c){  
    long p1 = a * b;  
    long p2 = a * c;  
    long p3 = p1 * p2;  
    return p3;  
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

# 超标量处理器

## Superscalar Issue (Pentium)

Cycle	1	2	3	4	5	6	7	8	9
Instr <sub>1</sub>	Fetch	Decode	Execute			Write			
Instr <sub>2</sub>	Fetch	Decode	Wait			Execute	Write		
Instr <sub>3</sub>		Fetch	Decode	Execute	Write				
Instr <sub>4</sub>		Fetch	Decode	Wait			Execute	Write	
Instr <sub>5</sub>			Fetch	Decode	Execute	Write			
Instr <sub>6</sub>			Fetch	Decode	Execute	Write			
Instr <sub>7</sub>				Fetch	Decode	Execute	Write		
Instr <sub>8</sub>				Fetch	Decode	Execute	Write		

- Superscalar issue allows multiple instructions to be issued at the same time

# 分支预测

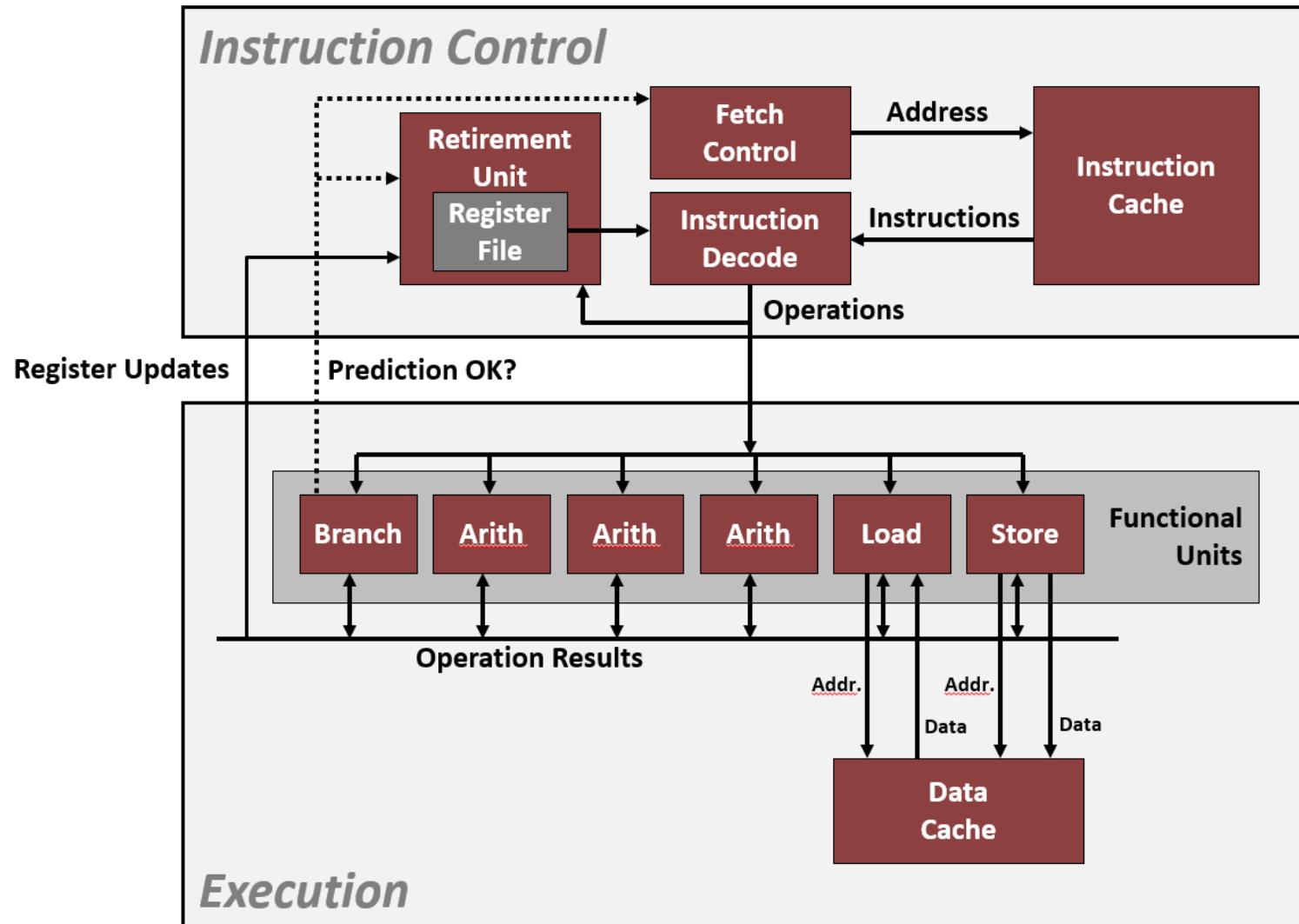
- 预测分支的跳转

```
404663: mov $0x0, %eax  
404668: cmp (%rdi), %rsi  
40466b: jge 404685 ← How to continue?  
40466d: mov 0x8(%rdi), %rax  
.....  
404685: repz retq
```

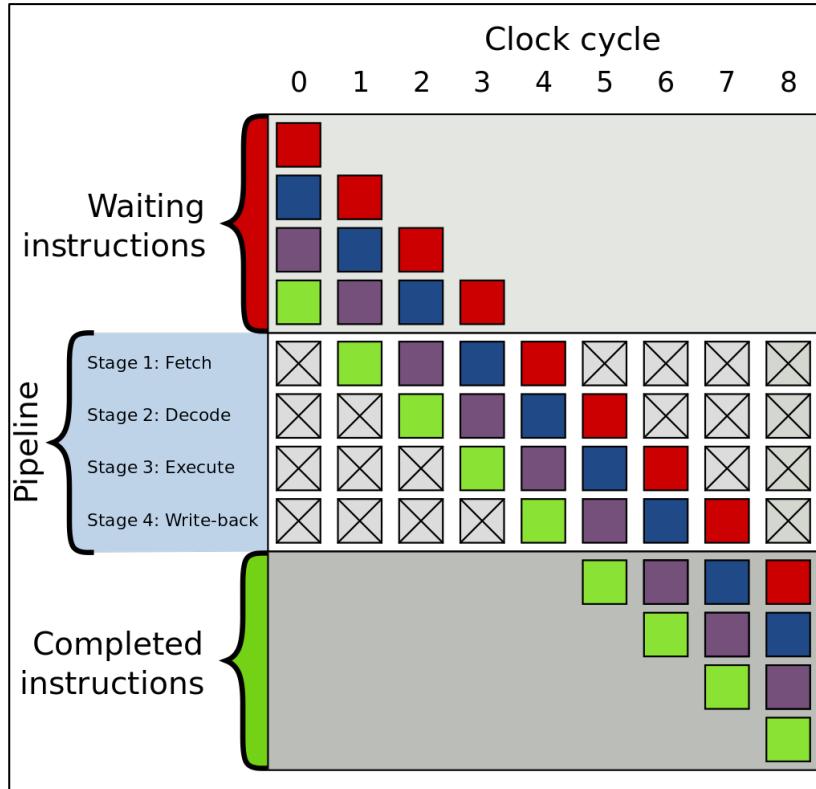
}] Executing

- 本质：猜测并预测分支的走向
- [java - Why is processing a sorted array faster than processing an unsorted array? - Stack Overflow](#)

# 现代处理器



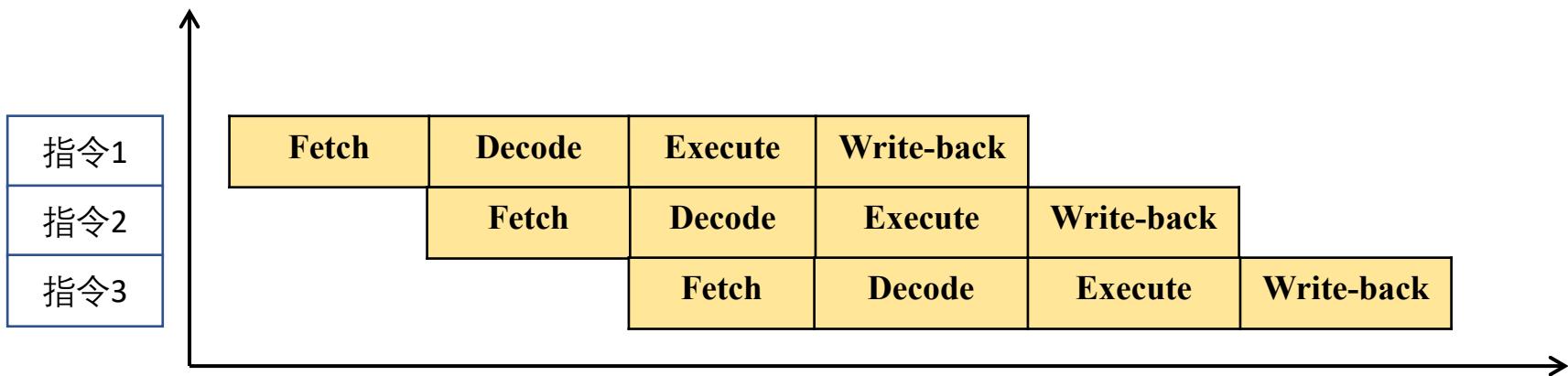
From CMU CSAPP Course



```

int data[N];
int sum = 0;
for (int i = 0; i<N; i++){
    if(data[i] < 128)
        sum += data[i];
}

```



```
int t = (data[i]-128)>>31;
sum+=~t & data[c];
```

```
int data[N];
int sum = 0;
for (int i = 0; i<N; i++){
    if(data[i] < 128)
        sum += data[i];
}
```

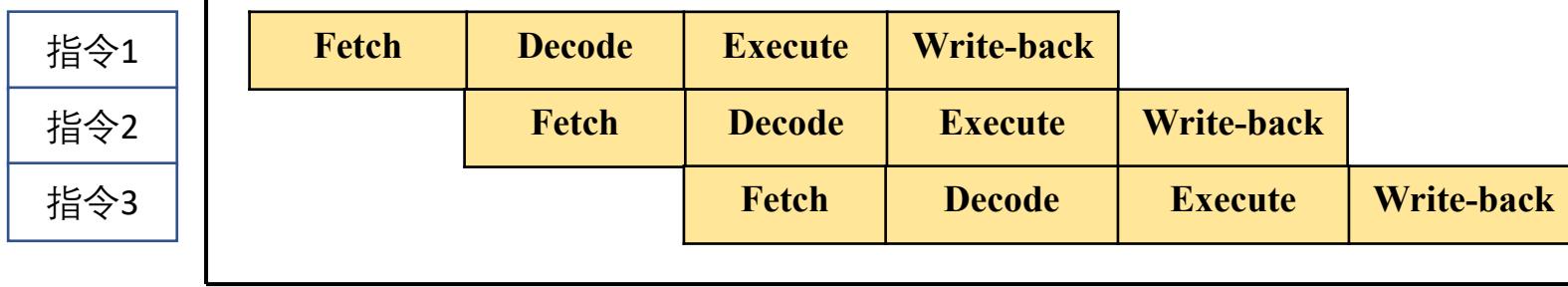
Sorted:

```
data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118, 14, 150, 177, 182, ...
branch = T, T, N, T, T, T, N, T, N, N, T, T, T ...
= TTNTTTNTNNTT ... (completely random - impossible to predict)
```

Unsorted:

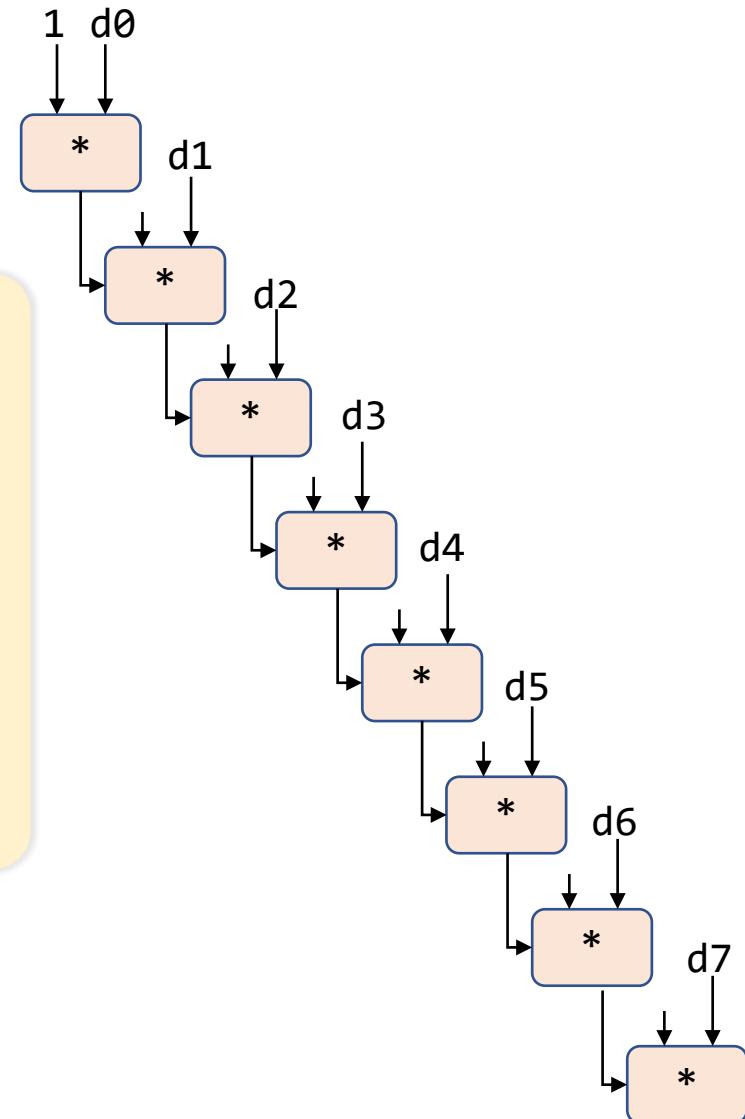
T = branch taken  
N = branch not taken

```
data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, 251, 252, ...
branch = N N N N N ... N N T T T ... T T T ...
= NNNNNNNNNN ... NNNNNNNTTTTTT ... TTTTTTTT (easy to predict)
```



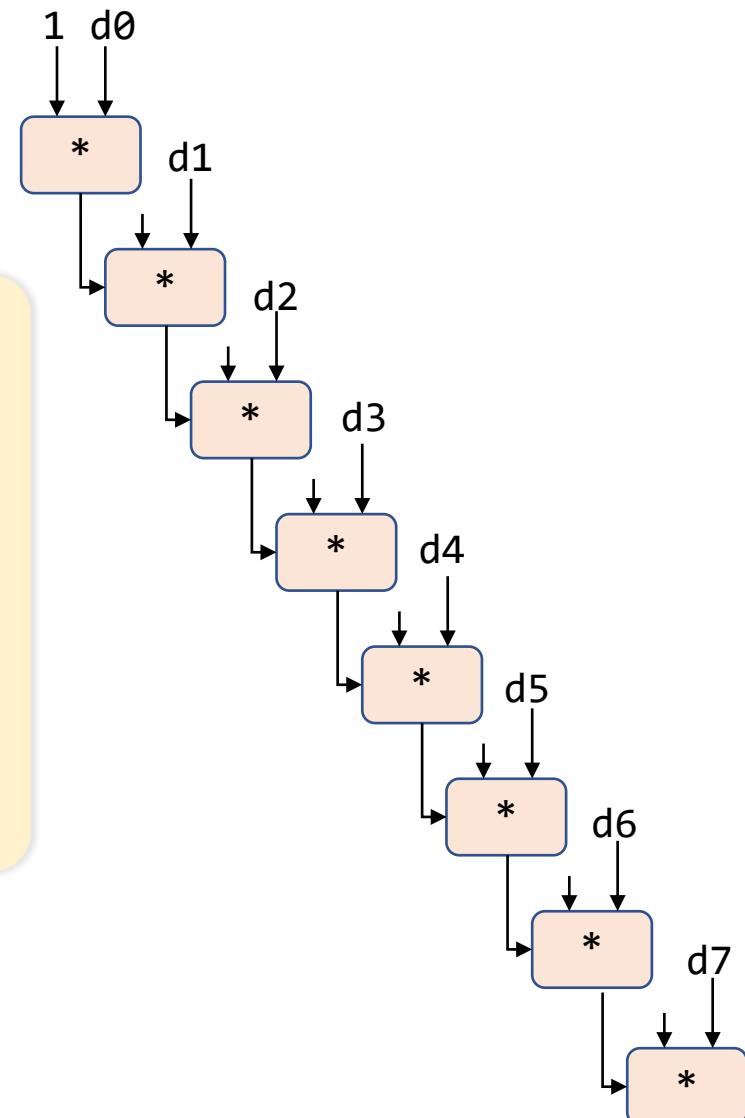
# 考慮指令並行的優化

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    data_t *data = get_vec_start(v);  
    data_t acc = IDENT; //0 for +, 1 for *  
    for (i =0; i < length; i++){  
        acc = acc OP data[i];  
    }  
    *dest = acc;  
}
```



# 循环展开1

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    data_t *data = get_vec_start(v);  
    data_t acc = IDENT; //0 for +, 1 for *  
    for (i =0; i < length; i+=2){  
        acc = (acc OP data[i]) OP data[i+1];  
    }  
    *dest = acc;  
}
```

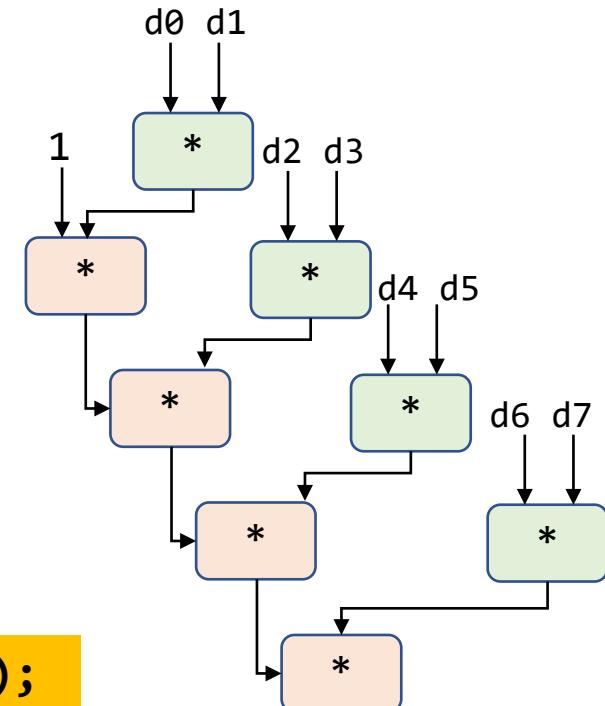


# 循环展开2

- 打破指令内在的顺序依赖关系

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    data_t *data = get_vec_start(v);  
    data_t acc = IDENT; //0 for +, 1 for *  
    for (i =0; i < length; i+=2){  
        acc = (acc OP data[i]) OP data[i+1];  
    }  
    *dest = acc;  
}
```

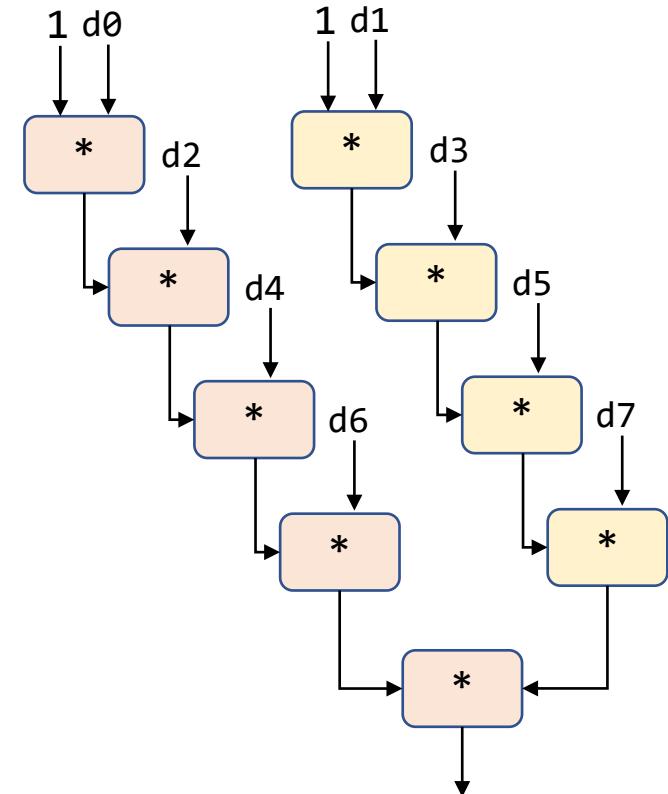
acc = acc OP (data[i] OP data[i+1]);



# 循环展开3-multiple accumulator

- 打破指令内在的顺序依赖关系

```
void combine1 (vec_ptr v, data_t *dest){  
    long i;  
    long length = vec_length(v);  
    long limit = length - 1;  
    data_t *data = get_vec_start(v);  
    data_t acc = IDENT; //0 for +, 1 for *  
    //combine 2 elements at a time  
    for (i =0; i < limit; i+=2){  
        x0 = x0 OP data[i];  
        x1 = x1 OP data[i+1];  
    }  
    //finish any remaining element  
    for(; i < length; i++)  
        x0 = x0 OP data[i];  
    *dest = x0 OP x1;  
}
```



# 循环展开的思想

- 尝试通过控制参数，接近吞吐量的最优化
  - Unrolling
  - Accumulating

```
for (i =0; i < limit; i+=2){  
    x0 = x0 OP data[i];  
    x1 = x1 OP data[i+1];  
}
```

# SIMD

- ICS Lecture 06

## 单指令多数据

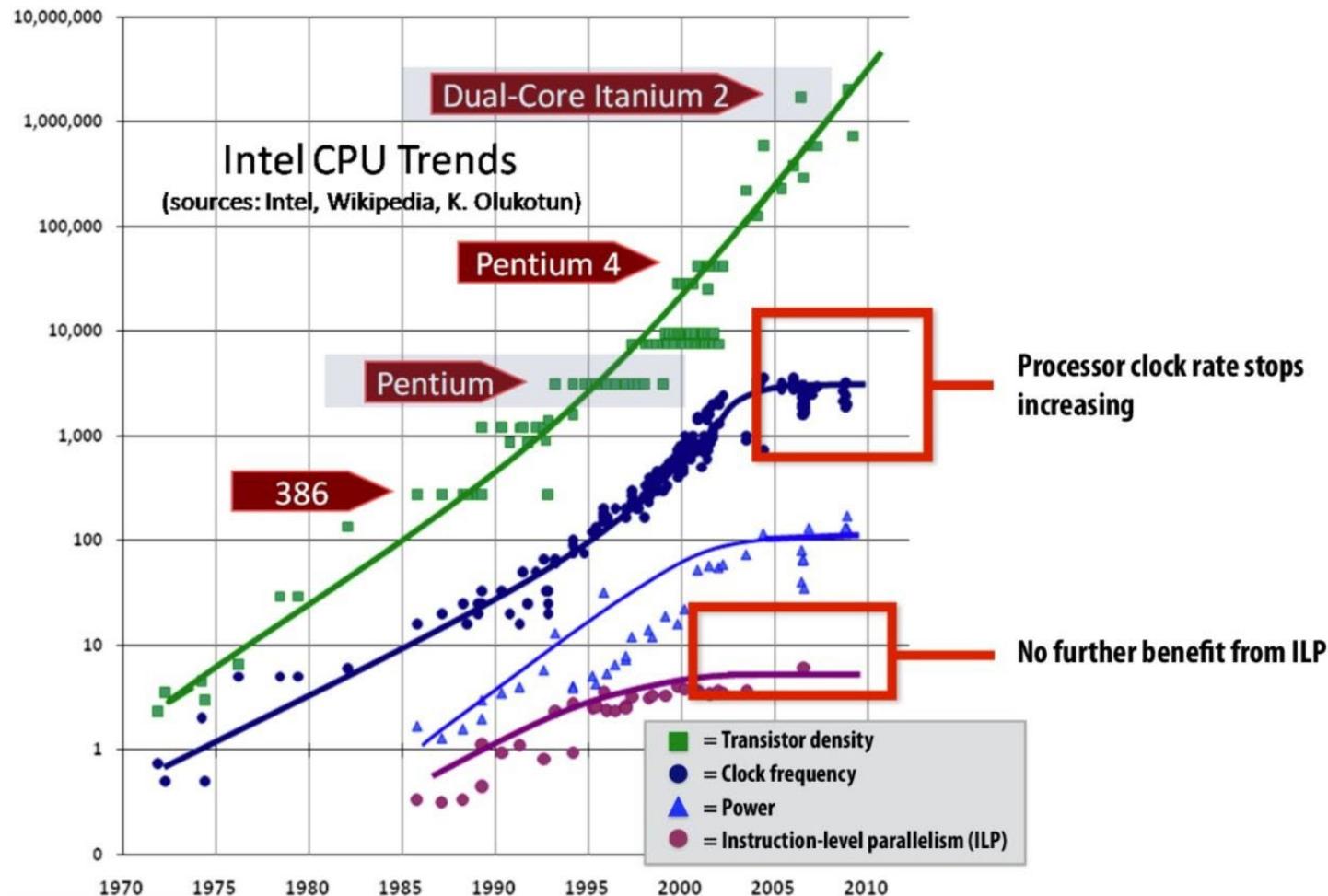
&, |, ~, ... 对于整数里的每一个 bit 来说是独立（并行）的

- 如果我们操作的对象刚好每一个 bit 是独立的
  - 我们在一条指令里就实现了多个操作
  - SIMD (Single Instruction, Multiple Data)
- 例子： Bit Set
  - 32-bit 整数  $x \rightarrow S \subseteq \{0, 1, 2, 3, \dots, 31\}$
  - 位运算是对所有 bit 同时完成的
    - C++ 中有 `bitset`, 性能非常可观

5 -> 0101

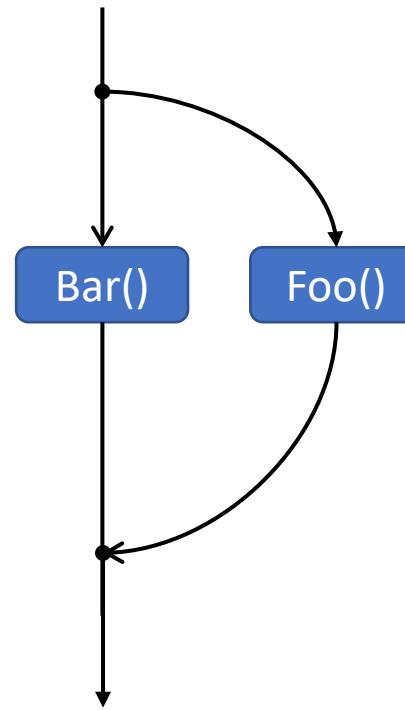
$S: \{0, 2\}$

# 多核时代: parallel thinking



# parallel thinking

```
//part A  
  
Click_spawn Foo();  
Bar();  
Click_sync;  
  
//part B
```



- E.g., QuickSort
  - Parallel
    - $\text{QuickSort}(A_1 \dots s)$
    - $\text{QuickSort}(A_{s+1} \dots |A|)$ ,

# 总结

---

- 程序性能的优化需求明显
  - Optimization重要
  - Readable code更重要
- 推荐的资料
  - CMU: 15-853: Algorithms in the “Real World”
    - A graduate-level course that provides many useful links to parallel algorithms and I/O-efficient algorithm
  - MIT 6.172: Performance Engineering of Software Systems
    - A more thorough explanation on performance analysis on parallel (multi-core) and distributed setting
  - CMU 15-210: Parallel and Sequential Data Structures and Algorithms
    - An overview of basic algorithms and data structures that makes no distinction between sequential and parallel

# 总结

- 选择适合的算法和数据结构
  - 算法课
  - Parallel thinking
- 编写能够被编译器有效优化并转化为高效的可执行代码的源码
  - 理解编译器优化的能力和局限
  - 基本编码原则和低级优化
- 现实的性能诊断任务，用好trace和profiler工具

## Lab3: 性能调优

### 小实验说明

小实验 (Labs) 是 ICS 这门课程里的一些综合编程题，旨在结合课堂知识解决一些实际中的问题。因为问题来自实际，所以有时候未必能立即在课本上找到相关知识的答案，而是需要“活学活用”。因此，大家需要利用互联网上的知识解决这些问题，但不要试图直接搜索这些问题的答案，即便有也不要点进去(也请自觉不要公开发布答案)。

Deadline: 2022 年 12 月 4 日 23:59:59。

# Life in real world

## Lab3: 性能调优

### 小实验说明

小实验 (Labs) 是 ICS 这门课程里的一些综合编程题，旨在结合课堂知识解决一些实际中的问题。因为问题来自实际，所以有时候未必能立即在课本上找到相关知识的答案，而是需要“活学活用”。因此，大家需要利用互联网上的知识解决这些问题，但**不要试图直接搜索这些问题的答案，即便有也不要点进去**(也请自觉不要公开发布答案)。

Deadline: 2022 年 12 月 4 日 23:59:59。

- 用好trace和profiler工具

# PA + Lab

## PA大作业

每次实验前，请仔细阅读[实验须知/提交方法](#)和[PA实验指南](#)。

- PA0: [环境安装与配置](#) (已截止, DDL: 2022年9月20日23:59:59 (extended))
- PA1: [监视器](#) (已发布, DDL: 2022年10月13日23:59:59 (extended))
- PA2: [模拟指令运行](#) (已发布, DDL: 2022年11月20日23:59:59)
- PA3: [中断与异常](#) (已发布, DDL: 2022年12月18日23:59:59)
- PA4: [分时多任务](#) (已发布, DDL: 2023年1月15日23:59:59)

## Lab小作业

每次实验前，请仔细阅读[实验须知/提交方法](#)。

- Lab1: [大整数运算](#) (已截止, DDL: 2022年10月16日23:59:59)
- Lab2: [x86-64内联汇编](#) (已截止, DDL: 2022年11月13日23:59:59)
- Lab3: [性能调优](#) (已发布, DDL: 2022年12月4日)
- Lab4: [缓存模拟器](#) (已发布, DDL: 2022年12月25日)

# 虚拟存储选讲

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



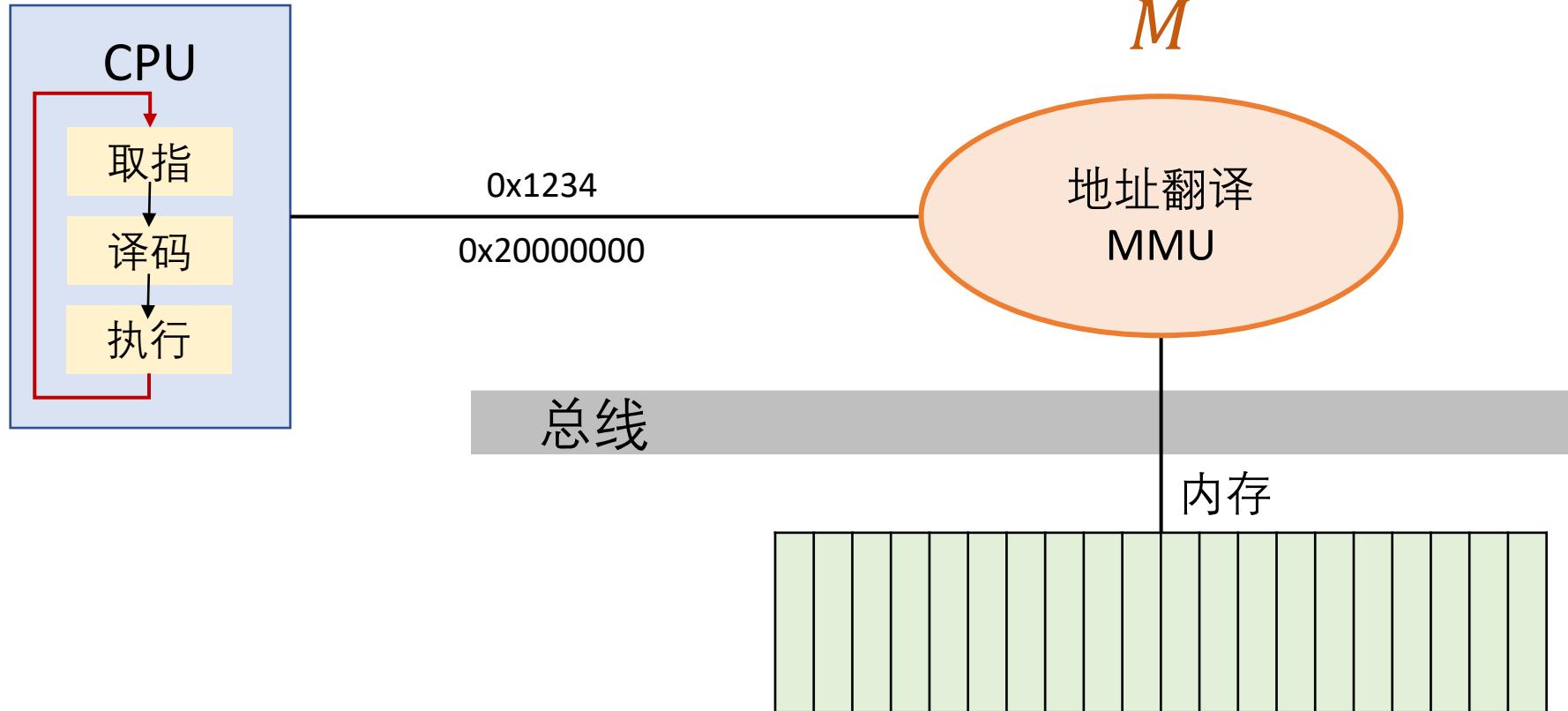
# 本讲概述

为什么同一个程序、同一个地址可以启动很多份？

- 本讲内容

- 虚拟存储：机制
- 存储器体系结构
- Meltdown

# 虚拟存储：机制





```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/mman.h>
4 #include <assert.h>
5
6 int main() {
7     for (int i = 0; i < 3; i++) {
8         fork(); // create some processes
9     }
10
11     volatile int *ptr = mmap(
12         (void *)0x20000000,           // mapping address (hint)
13         1 << 20,                  // mapping size
14         PROT_WRITE | PROT_READ,    // read/write mapping
15         MAP_PRIVATE | MAP_ANONYMOUS, // mapping flags
16         -1,                      // file descriptor
17         0                         // offset
18     );
19     assert((intptr_t) ptr != -1);
20
21     *ptr = getpid();
22     printf("#%d sets %p = %d\n", getpid(), ptr, getpid());
23
24     sleep(1);
25     printf("#%d *%p = %d\n", getpid(), ptr, *ptr);
<Documents/ICS2021/teach/Course14/vm-demo.c[1] [c] unix utf-8 Ln 7, Col 3/26
=((foldclosed(line('.')) < 0) ? 'zc' : 'zo')
```

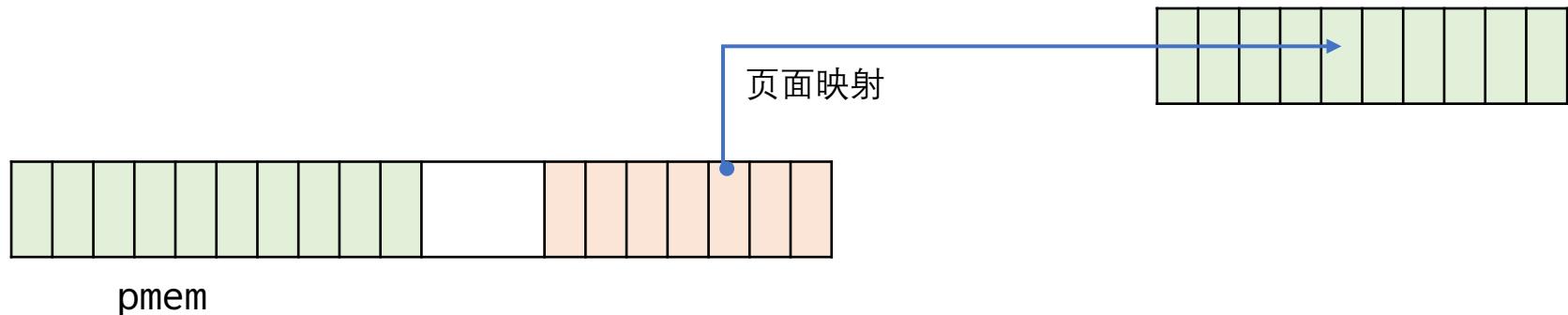
# 虚拟地址空间

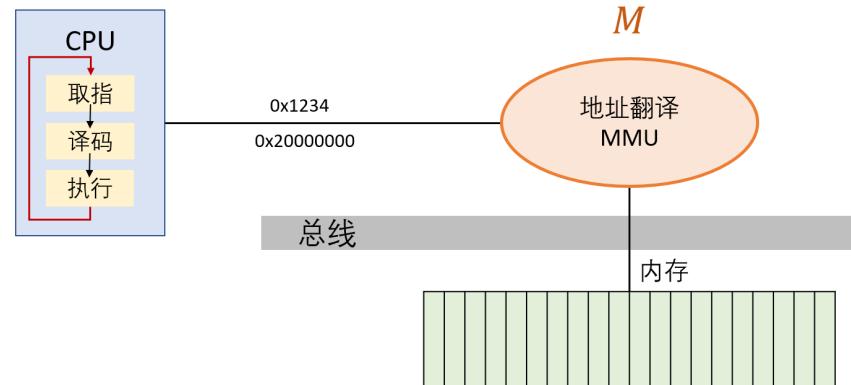
- 操作系统中的每个进程都有独立的地址空间
  - [vm-demo.c](#) (同一个指针、不同的物理内存)
  - MMU: 硬件维护数据结构  $M$ 
    - 在运行时把地址  $x$  映射到  $M(x)$
    - 所有内存访问都将执行这个转换 (取指令、访存……)
- 虚拟地址空间设计
  - 低特权程序无权修改  $M(x)$
  - 访问未映射的内存产生异常 (Segmentation Fault)
  - 能够控制 read/write/execute 权限
    - 思考题：代码、数据、堆栈分别设置为什么权限？

# 理解虚拟地址空间：VME API

- 内存是分为“页面”的
- 支持在内存中创建“地址空间”对象( $M$ )
  - 支持建立/取消页面到页面的映射
  - 可以将 $M$ “加载”到系统上

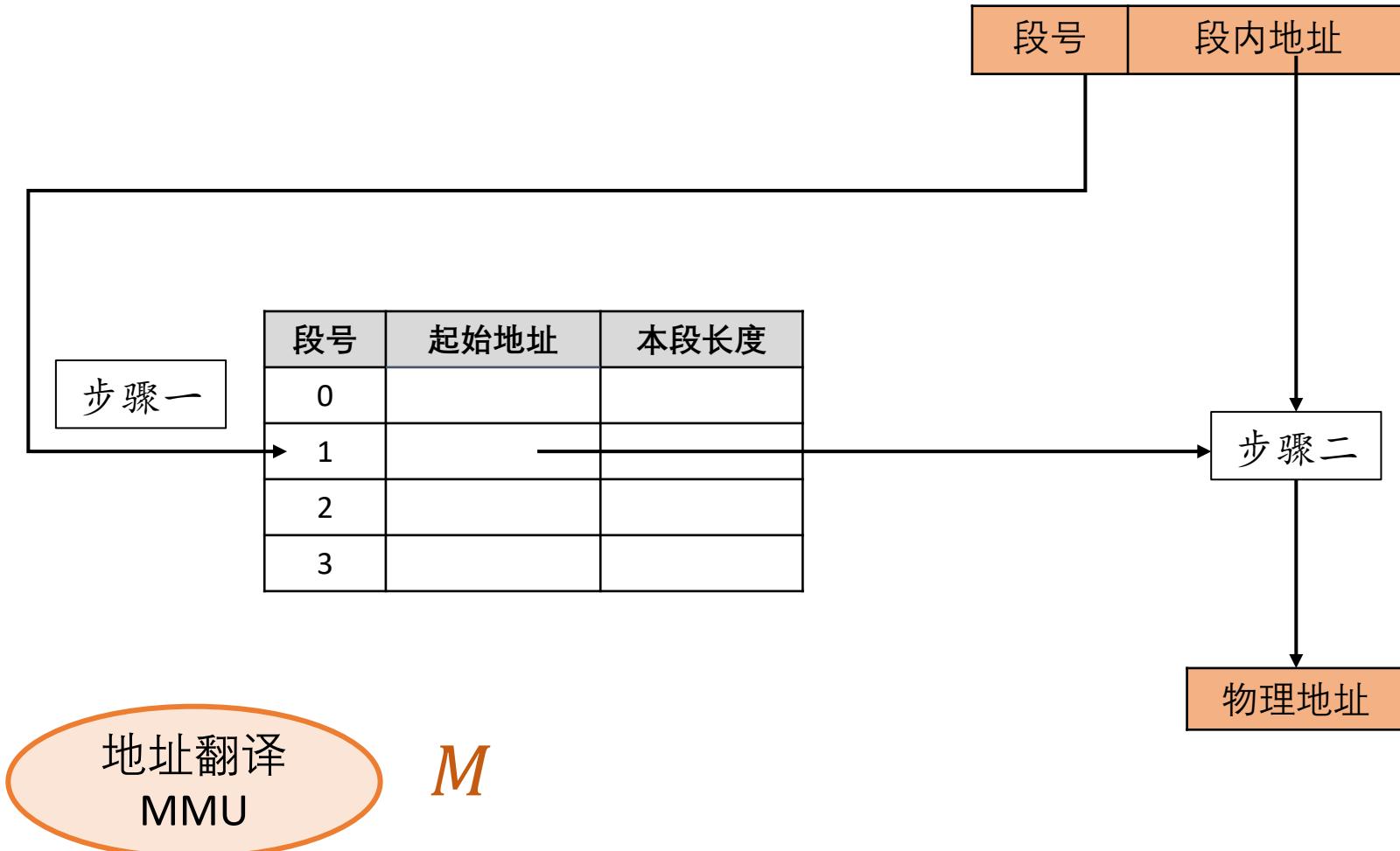
```
bool      vme_init    (void *(*alloc)(int), void (*free)(void *));
void     protect     (AddrSpace *a);
void     unprotect   (AddrSpace *a);
void     map         (AddrSpace *a, void *va, void *pa, int prot);
Context  *ucontext   (AddrSpace *a, Area kstack, void *entry);
```



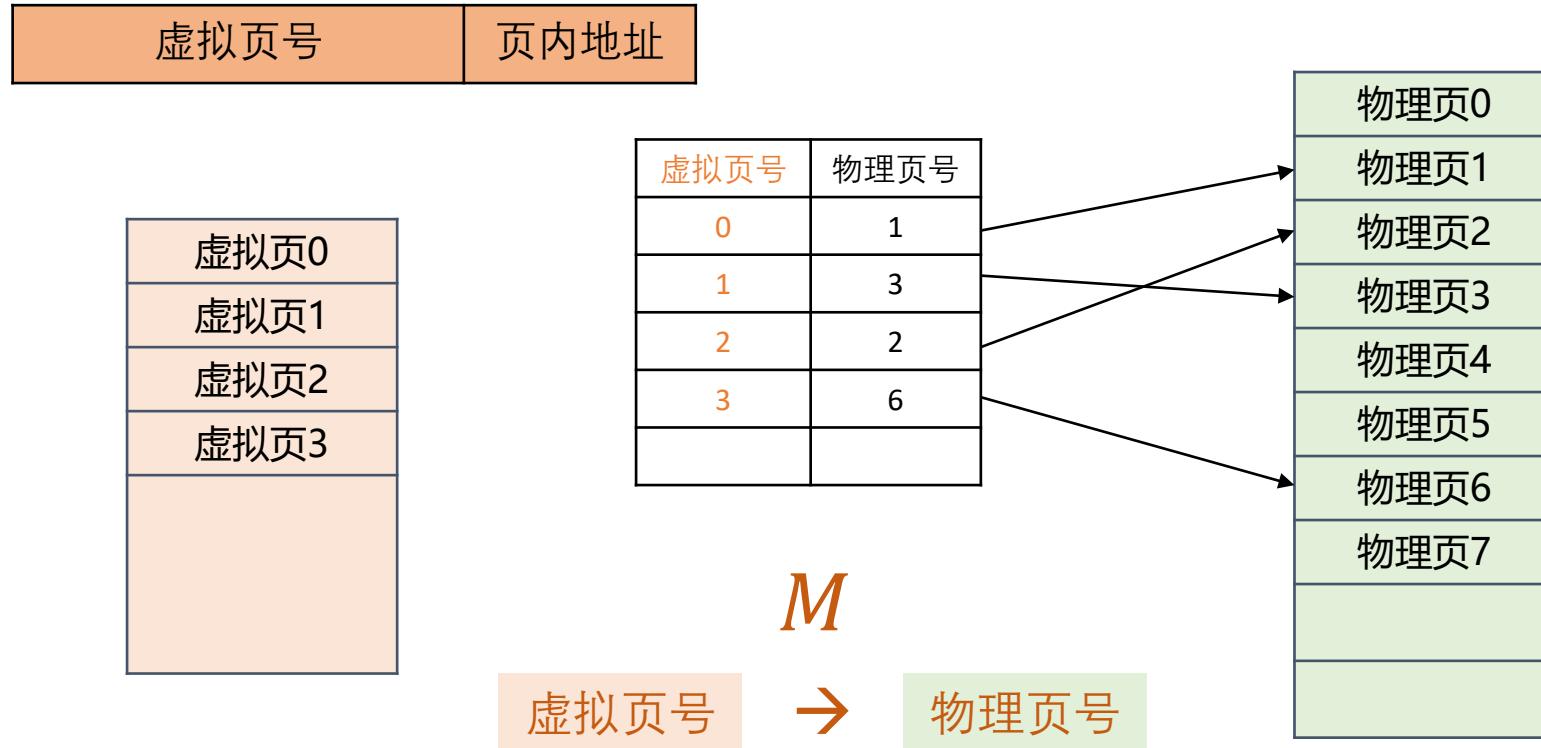


# 虚拟存储：分段与分页机制

# 分段机制



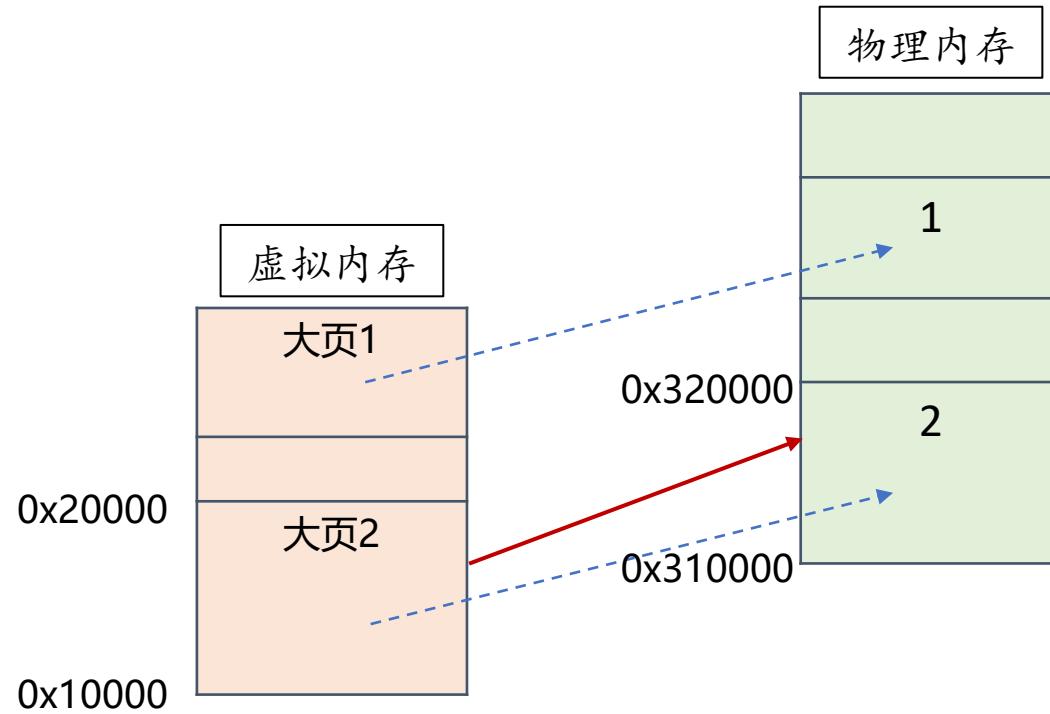
# 分页机制



地址翻译  
MMU

# 虚拟存储：硬件实现

x86:  $2^{20} \rightarrow 2^{20}$



$$(x, y) + base$$

# 地址翻译的实现

- 现数据结构好办，就是个 `map<uintptr_t, uintptr_t>`
  - 查询/修改速度要快，最好  $O(1)$
  - 空间消耗要少 (`map` 消耗  $c \cdot n$  的内存,  $c > 1$  )
- 我们的老朋友：局部性
  - 内存是连续排布的：代码、数据、堆栈.....
  - 而且访问也是局部的
  - 因此可以按页面 (4KB, 4MB, ...) 来分配和管理内存

虚拟页号	物理页号
0	1
1	3
2	2
3	6

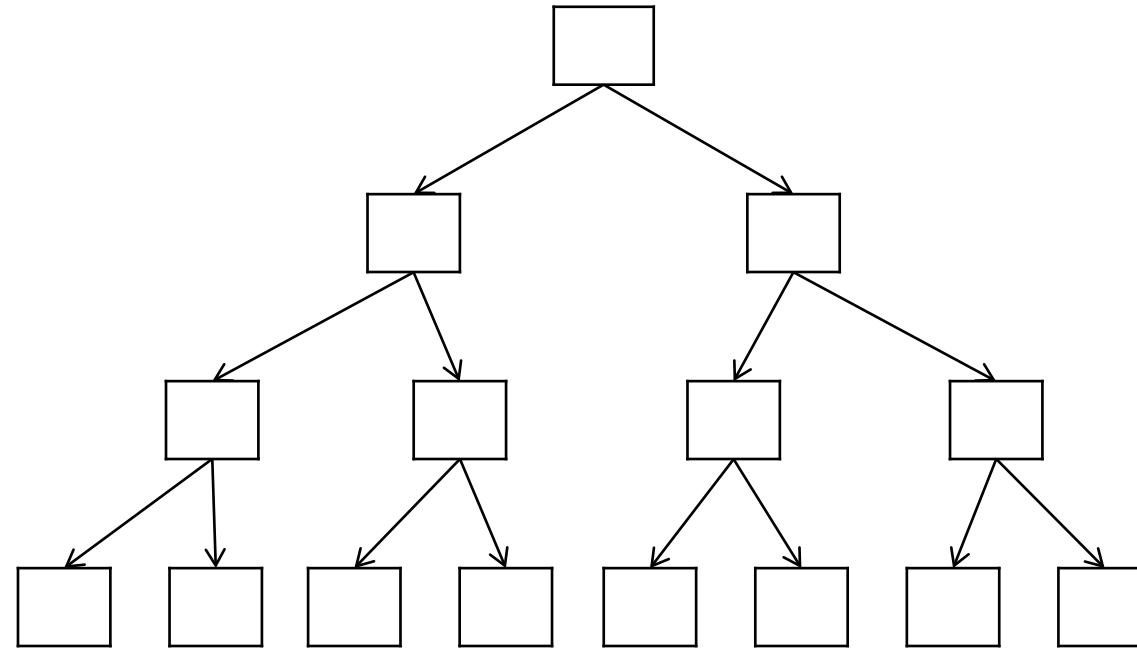


x86:  $2^{20} \rightarrow 2^{20}$

# 地址翻译的实现

- X86:  $2^{20} \rightarrow 2^{20}$ 
  - map<uintptr\_t, uintptr\_t>
  - E.g.,  $0, 1, \dots, 15 \rightarrow 0, 1, \dots, 15$

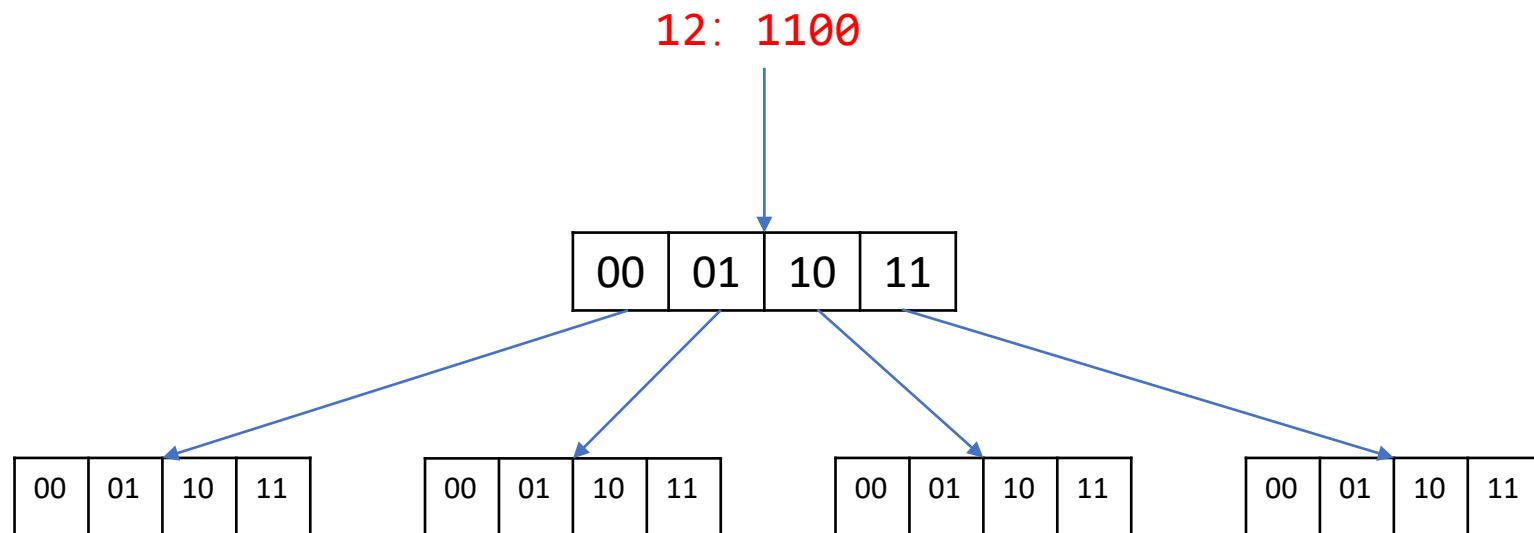
虚拟页号	物理页号
0	1
1	3
2	2
3	6



# 地址翻译的实现

- X86:  $2^{20} \rightarrow 2^{20}$ 
  - map<uintptr\_t, uintptr\_t>
  - E.g.,  $0, 1, \dots, 15 \rightarrow 0, 1, \dots, 15$

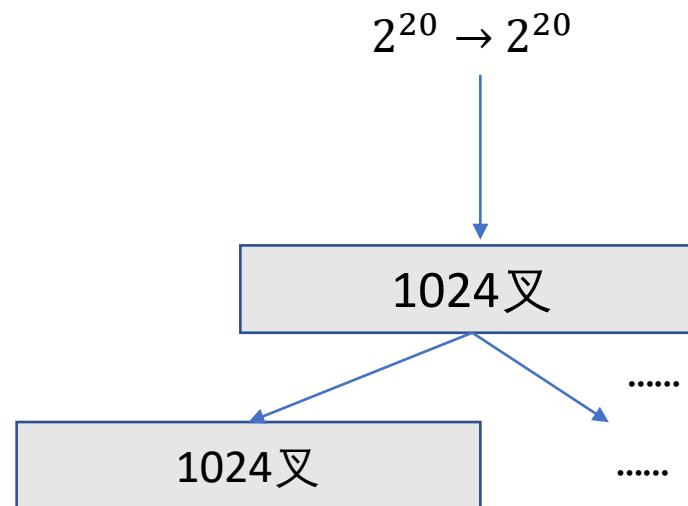
虚拟页号	物理页号
0	1
1	3
2	2
3	6



# 地址翻译的实现

- X86:  $2^{20} \rightarrow 2^{20}$ 
  - map<uintptr\_t, uintptr\_t>
  - E.g.,  $0, 1, \dots, 15 \rightarrow 0, 1, \dots, 15$

虚拟页号	物理页号
0	1
1	3
2	2
3	6

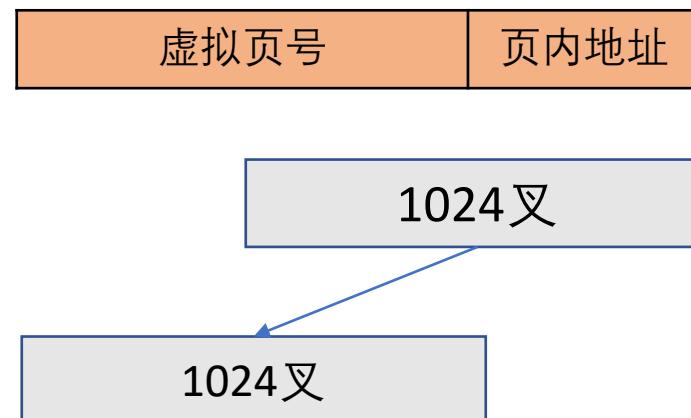


# 地址翻译：普通实现

$4GB/4KB = 1024^2$ , 那就实现成一个 2 层的 1024 叉树。

- 数据结构实现的技巧

- 未映射的地址空间不需要分配
- 根节点可以控制子节点的属性
- 32位地址翻译（4KB）
  - $10 + 10 + 12$
  - 8KB内？
- 64位呢？（4KB）
  - 48位有效地址



# 地址翻译：文艺实现

维护一张查找表。

- $(x_i, y_i, b_i) \Rightarrow \forall m \in [x_i, y_i), M(x_i + m) = b_i + m$
- 在每次内存访问时由硬件负责查表
  - 查找失败  $\rightarrow$  异常 (操作系统重填)
- 这是可编程MMU (MIPS)
  - 非常灵活，可以实现任意页表
  - 但 TLB miss 重填相对较慢
  - MIPS 是学生最容易实现完整计算机系统的指令集

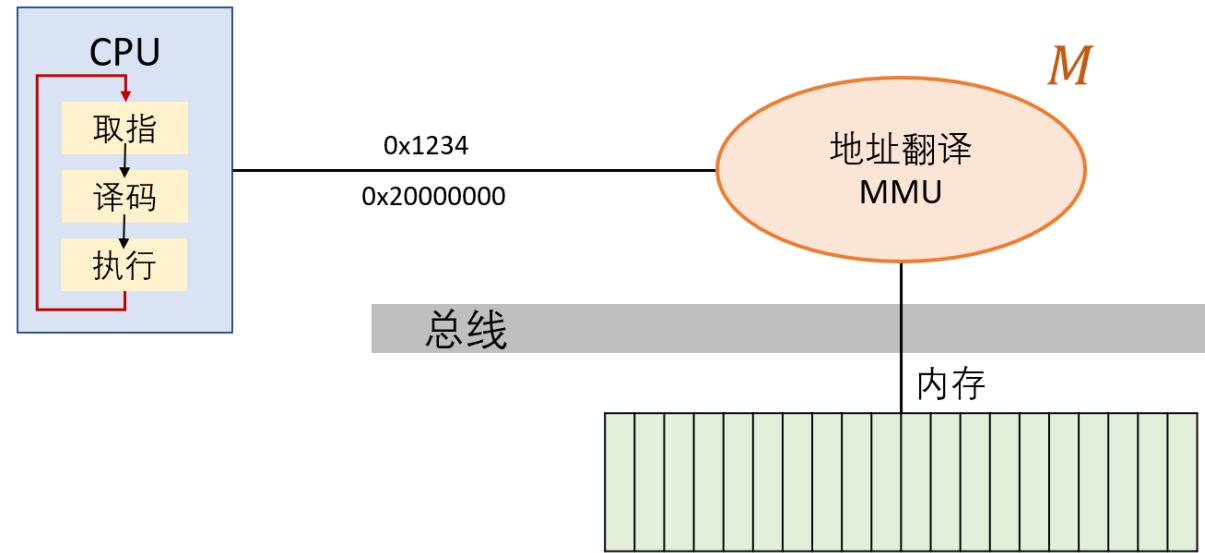
x	y	base
2000	3000	+10000

# 地址翻译：二逼实现

用一张hash table维护 $(x, asid) \rightarrow M(x)$ 的映射。

- 为什么这是可以的?
  - 只要系统里所有进程都 ASLR，hash 冲突就会很少
  - J Huck, J Hays. Architectural support for translation table management in large address space machines. In *Proc. of ISCA*, 1993.

x	y	base



# 存储器体系结构

# 第1层：寄存器

---

- 寄存器其实是地址空间非常小的内存 ([mov-reg.S](#))
  - 5bit → 32 个寄存器
  - 6bit → 64 个寄存器
    - 因此寻址通常是非常快的

管理 控制 视图 热键 设备 帮助

\$ vim mov-reg.S

\$ vim █



```
1  
2 mov-reg.o:      file format elf64-x86-64  
3  
4  
5 Disassembly of section .text:  
6 █  
7 0000000000000000 <.text>:  
8 0: 48 c7 c0 00 00 00 00    mov    $0x0,%rax  
9 7: 48 c7 c1 00 00 00 00    mov    $0x0,%rcx  
10 e: 48 c7 c2 00 00 00 00   mov    $0x0,%rdx  
11 15: 48 c7 c3 00 00 00 00  mov    $0x0,%rbx  
12 1c: 48 c7 c4 00 00 00 00  mov    $0x0,%rsp  
13 23: 48 c7 c5 00 00 00 00  mov    $0x0,%rbp  
14 2a: 48 c7 c6 00 00 00 00  mov    $0x0,%rsi  
15 31: 48 c7 c7 00 00 00 00  mov    $0x0,%rdi  
16 38: 49 c7 c0 00 00 00 00  mov    $0x0,%r8  
17 3f: 49 c7 c1 00 00 00 00  mov    $0x0,%r9  
18 46: 49 c7 c2 00 00 00 00  mov    $0x0,%r10  
19 4d: 49 c7 c3 00 00 00 00  mov    $0x0,%r11  
20 54: 49 c7 c4 00 00 00 00  mov    $0x0,%r12  
21 5b: 49 c7 c5 00 00 00 00  mov    $0x0,%r13  
22 62: 49 c7 c6 00 00 00 00  mov    $0x0,%r14  
23 69: 49 c7 c7 00 00 00 00  mov    $0x0,%r15
```

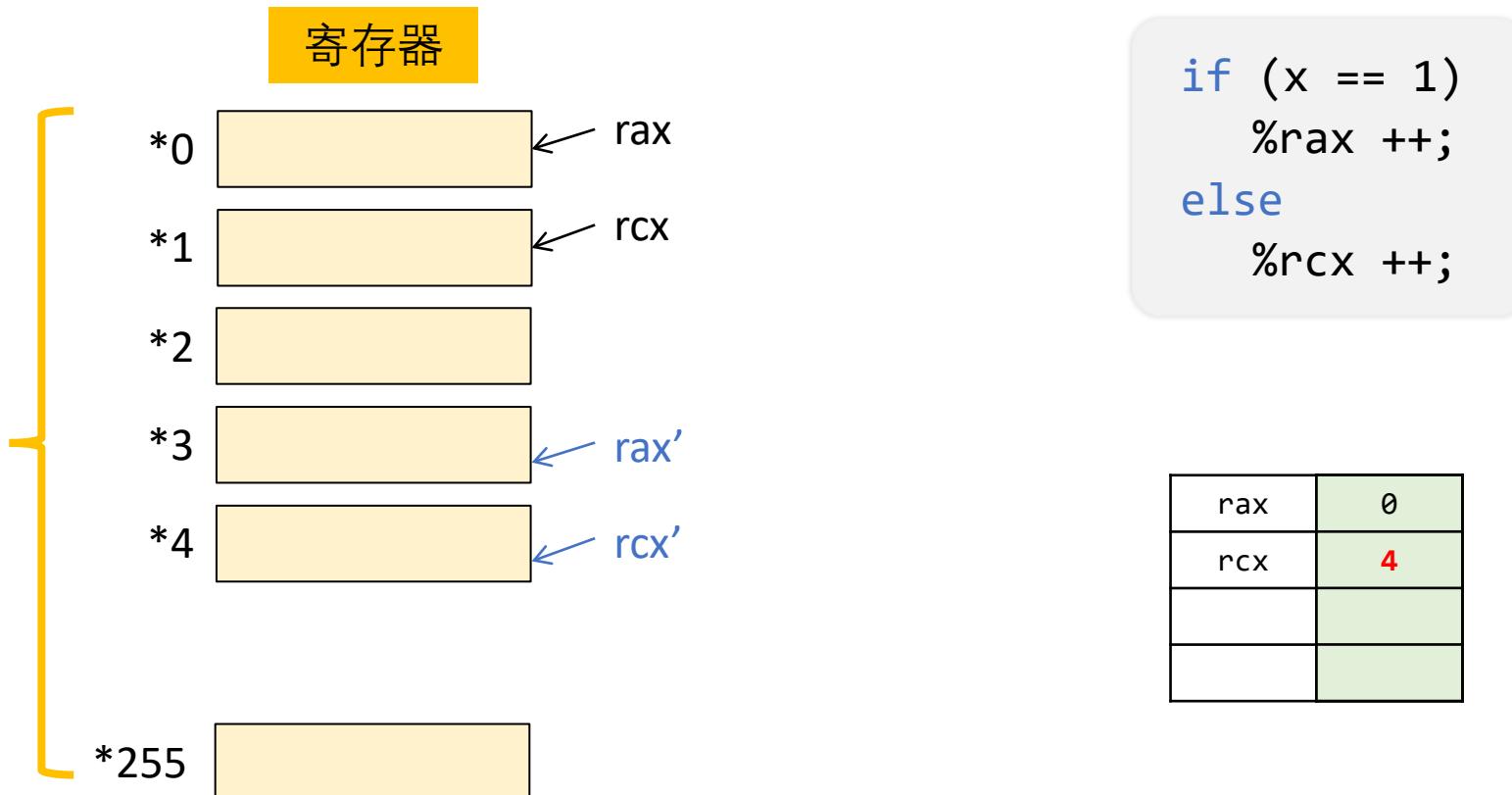
[No Name] [+1]

unix utf-8 Ln 6, Col 0/

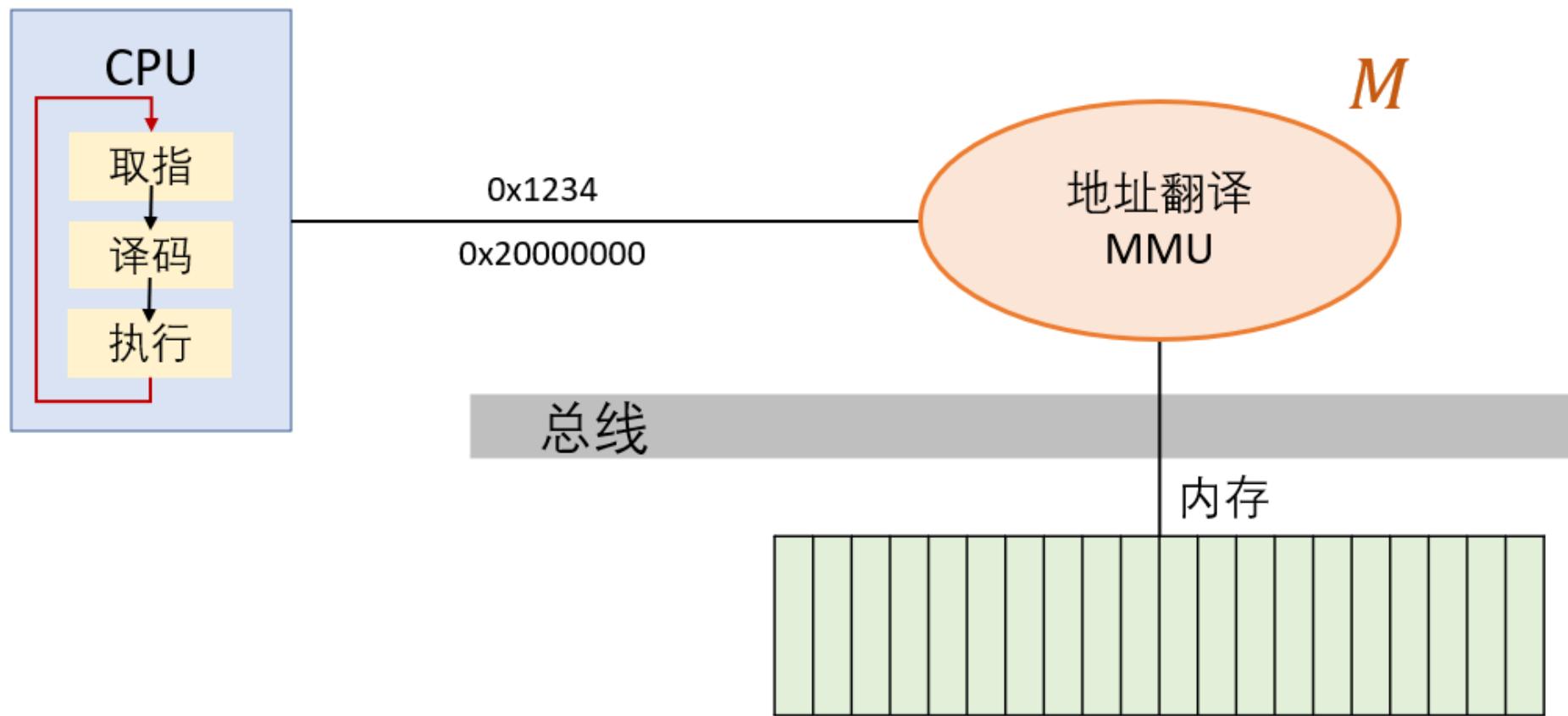
"-stdin-" 23L, 819B

# 第1层：寄存器

- 实际上，会有更大的 Physical Register File (PRF)
  - 我们执行的指令 (如 `mov $1, %rax`) 会被“重命名”
    - 即分配 PRF 的地址
    - 乱序执行、投机执行、异常处理……

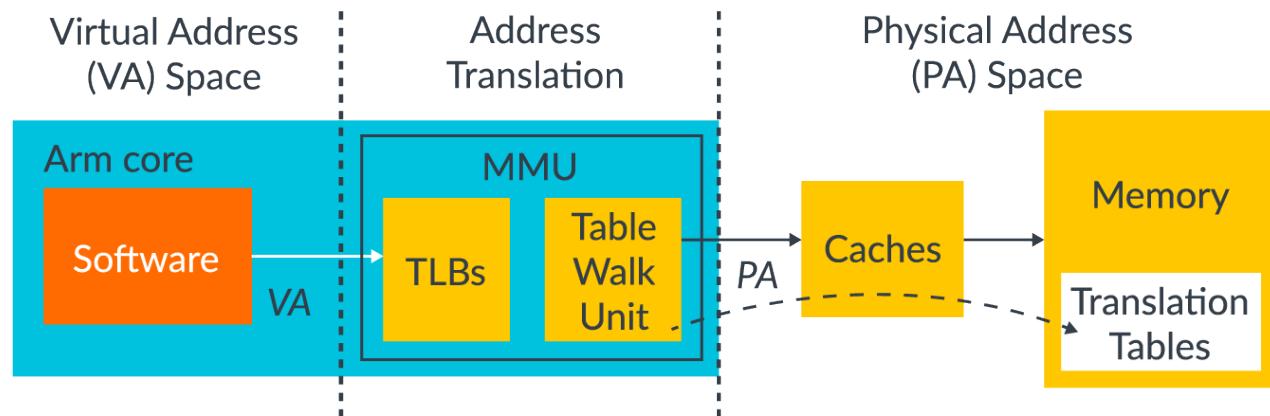


# 第1.5层：MMU



# 第1.5层：MMU

- [Armv8-A 手册](#)
- Radix tree
  - 每个节点可以是 4KB, 16KB, 64KB



## 第2层：缓存

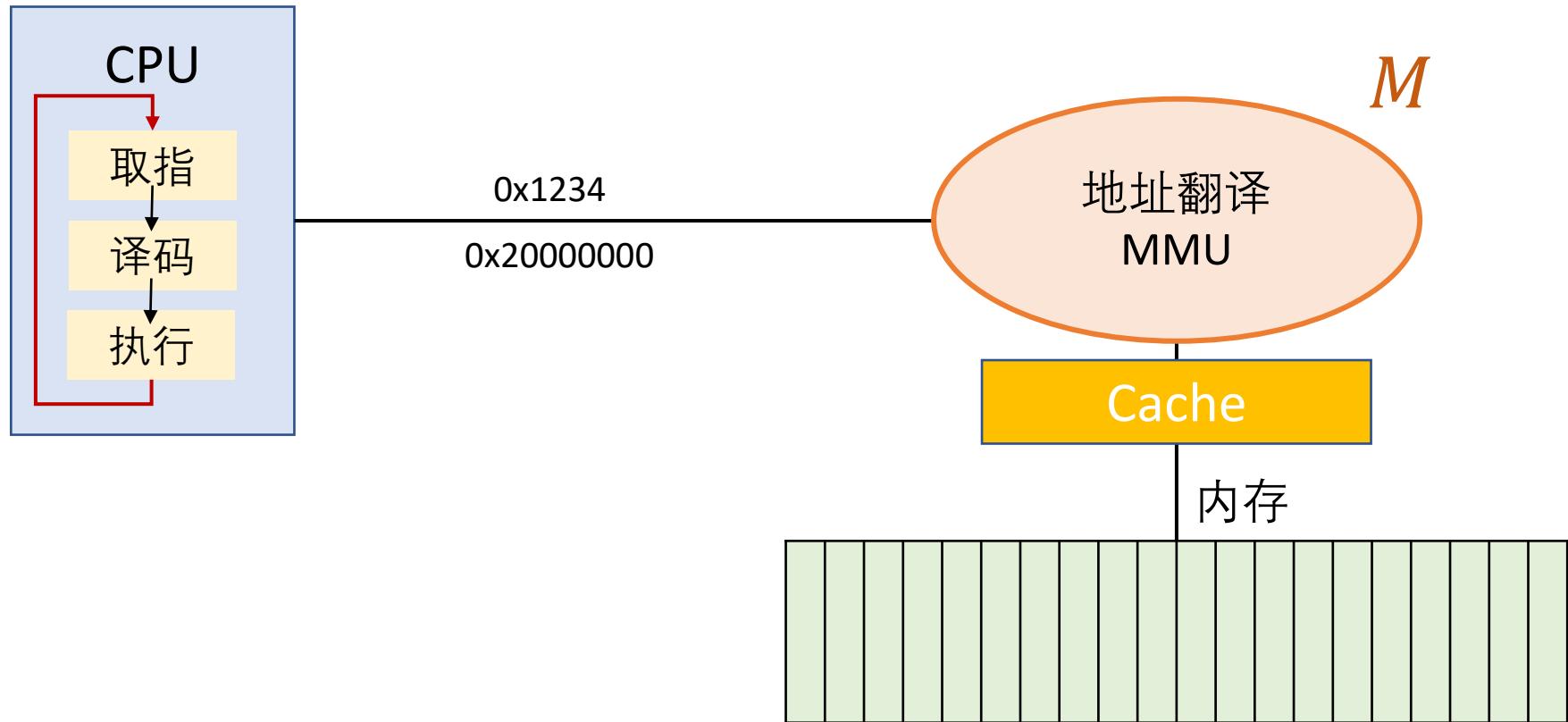
- 内存访问符合 Locality of References，“局部性原理”

A phenomenon in which the same values, or related storage locations, are frequently accessed, depending on the memory access pattern.

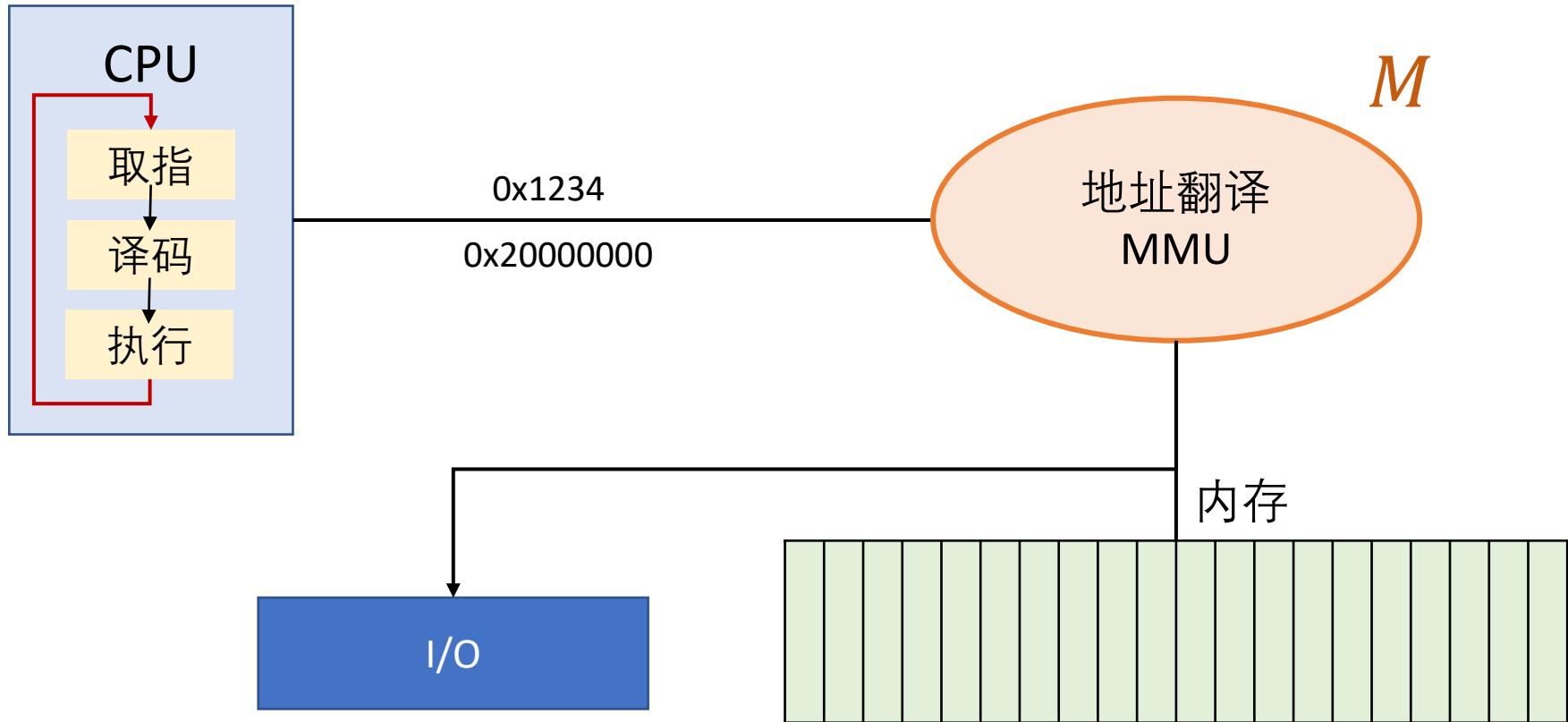
- 从另一个角度，根据内存访问的历史，通常能较为准确地预测未来可能访问的内存，并且访问临近内存居多
  - 缓存对寄存器/CPU 是完全透明的(吗？)

```
int *p = .....;
for (int i = 1; i < 1000; i++)
    *p = 1;
```

# 缓存给编程带来的麻烦

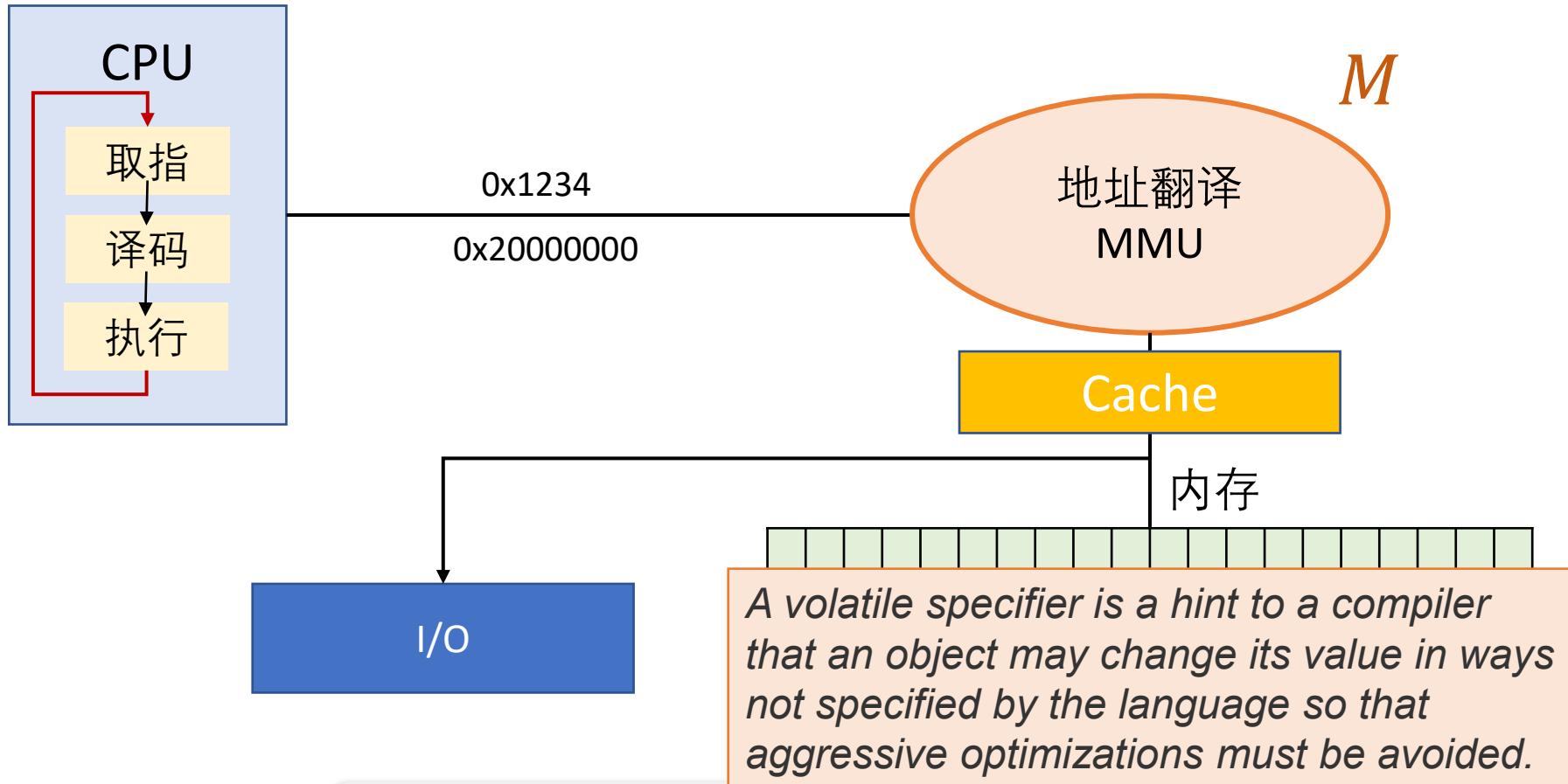


# 缓存给编程带来的麻烦



```
int *p = .....;
for (int i = 1; i < 1000; i++)
    *p = 1;
```

# 缓存给编程带来的麻烦



```
volatile int *p = .....;  
for (int i = 1; i < 1000; i++)  
    *p = 1;
```

# 第3层：内存(DRAM)

- 内存 DRAM 即一个大数组 (三维数据结构)

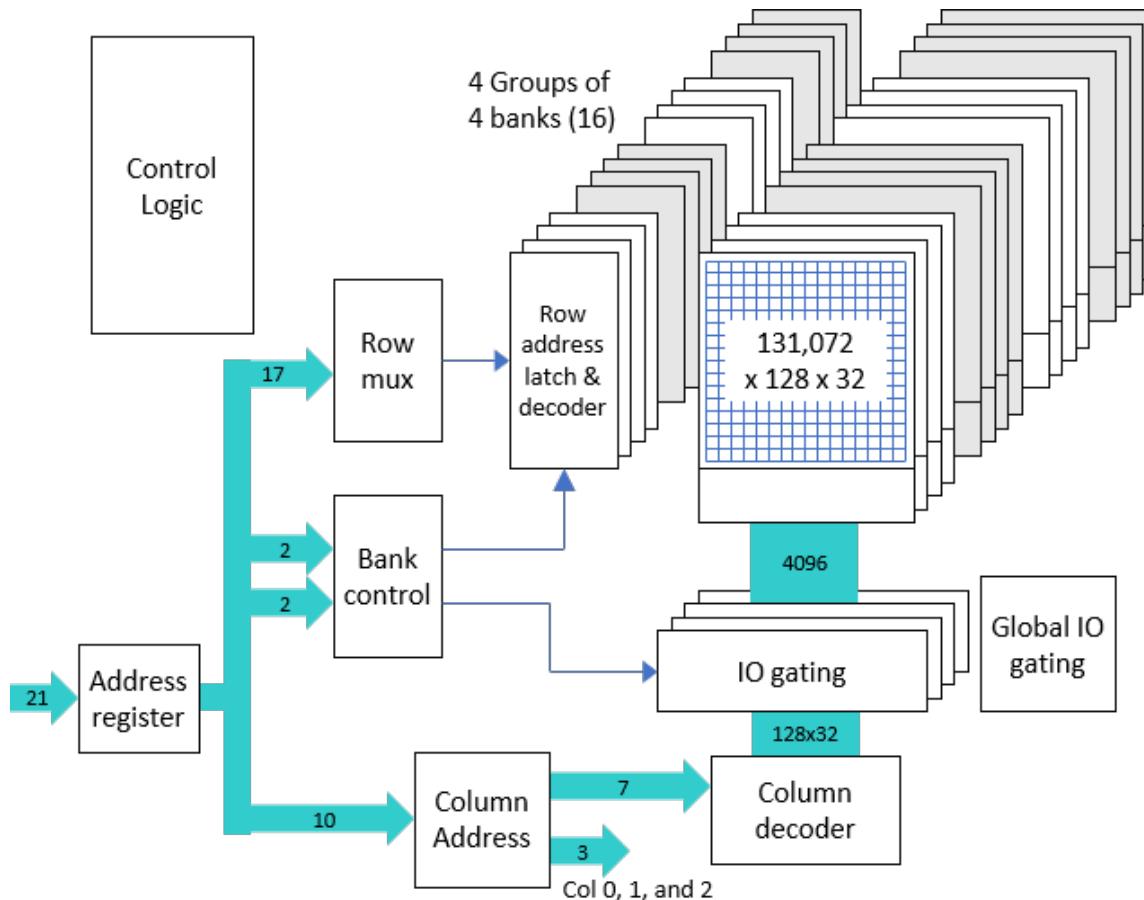


Image source: qdpma.com

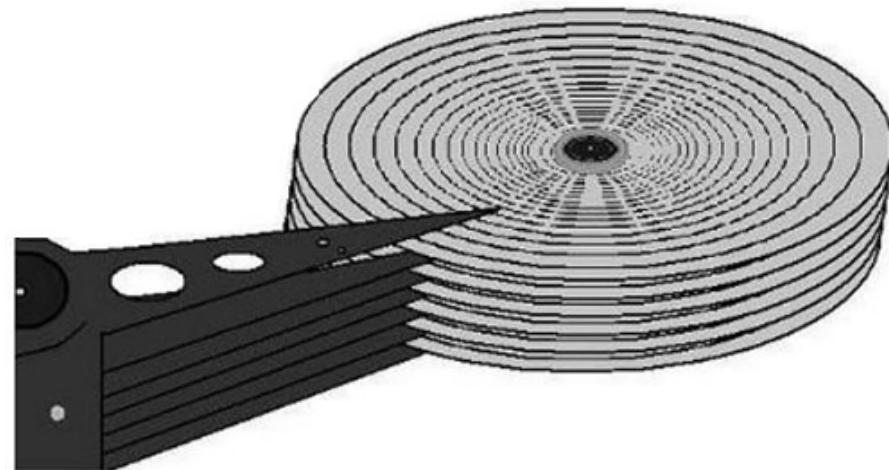
# 第3.5层：NVRAM

---

- 当内存/cache 越来越大、prefetch 越来越好.....
- DRAM 的快慢已经不是那么敏感了
  - 巨大的功耗 (定时刷新)
  - 内存错误率无法忽视
- 新的存储技术
  - PCM (phase change memory)
  - 产品: Intel Optane DC

# 第4层：外部存储器

- SSD Flash; 磁盘
  - 更大的延迟
  - 更大的容量
  - 更低的成本



中断 + 虚存 + 缓存 + 投机执行 = ?

# 问题 (1): 实现上的一个大麻烦

缓存虚拟地址还是物理地址?

- 缓存虚拟地址

- 每次切换  $M$ (进程) 都要清空缓存
- $M$ 的别名 (aliasing) 问题

- 缓存物理地址

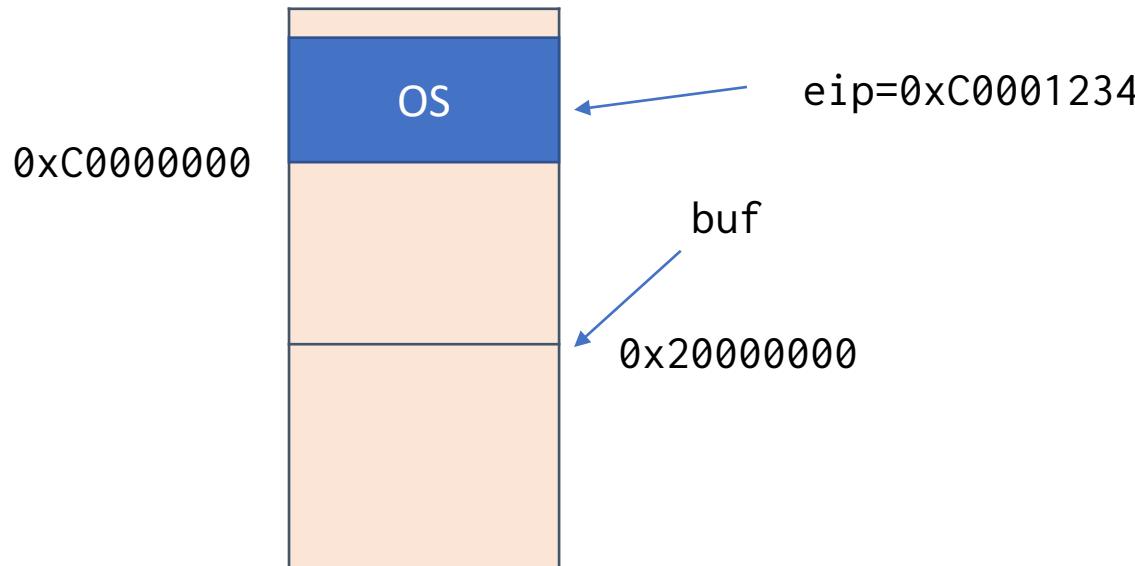
- 实现简单
- 每次 cache 访问都要等地址翻译

# 问题 (2): 把操作系统代码/数据的映射

```
char buf[SIZE];  
...  
ssize_t nwrite = write(fd, buf, SIZE); // in printf  
...
```

- 操作系统

- 希望直接向 buf 里写入
- 内核/用户进程映射同一地址空间 (但访问权限不同)
  - \*(KERNEL\_ADDRESS) 将引发 page fault (SIGSEGV)



# 操作系统：实现

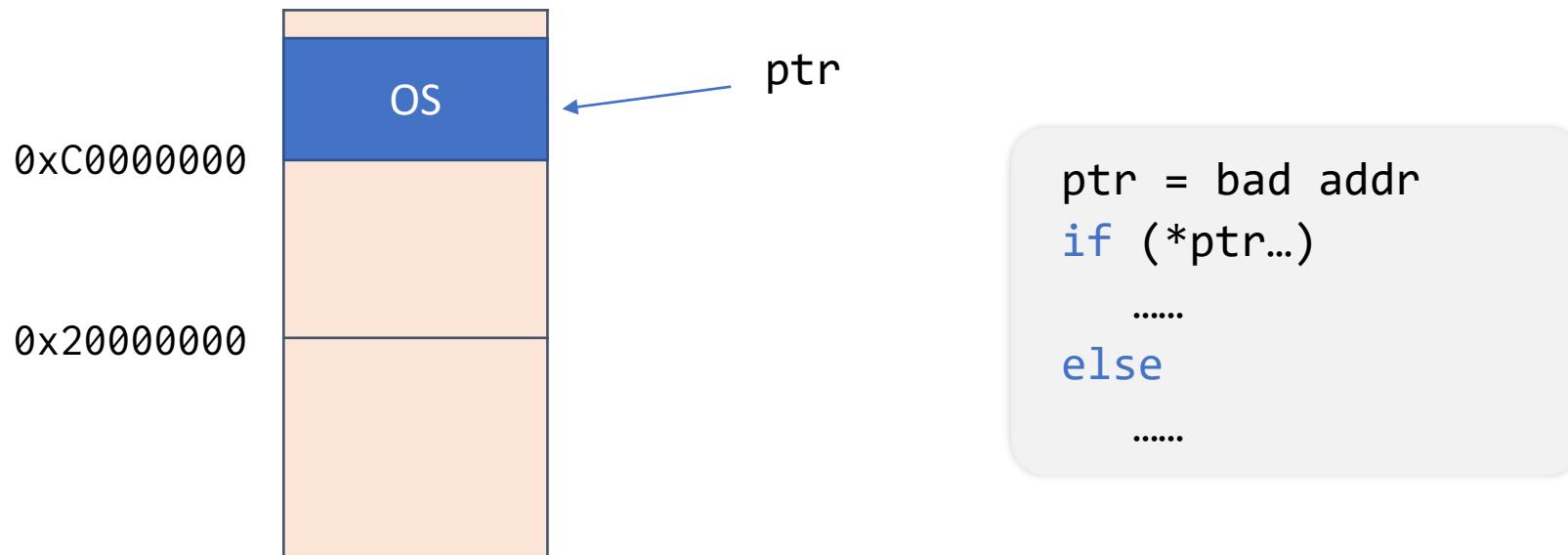
在地址空间中同时维护操作系统内核与用户进程的内存映射。

- 例子

- $0xc0000000-0xffffffff$  内存用户进程不可访问 (U-bit = 1)
  - 操作系统代码/数据映射到此
- 中断处理程序入口是  $0xc0001234$ 
  - 中断发生后，处理器自动切换运行级别
  - 合法跳转到操作系统代码执行 (%eip =  $0xc0001234$ )
    - 但 Ring 3 访问将导致 segmentation fault

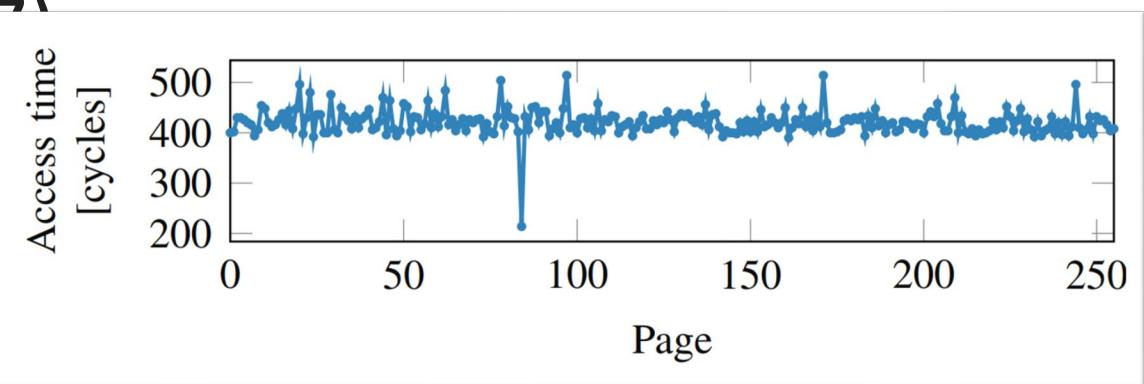
# Meltdown (2017)

- 复杂系统里有惊喜！
  - 中断 + 虚存 + 缓存 + 投机执行 = Meltdown
  - [Meltdown: Reading kernel memory from user space](#)



# Meltdown (2017)

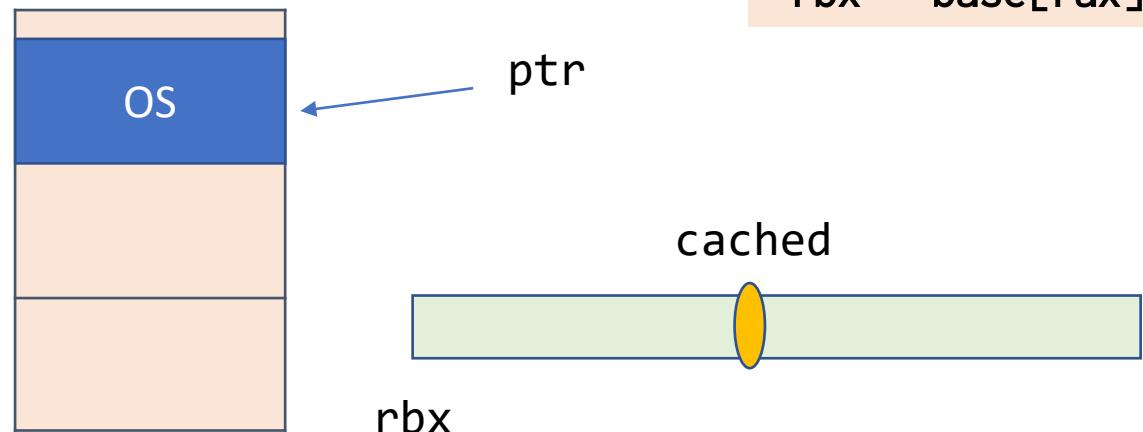
- 复杂系统里有惊喜！
  - 中断 + 虚存 + 缓存 +
  - [Meltdown: Reading kernel memory](#)



```
// %rcx: 无权限访问的地址; %rbx: 未载入缓存的数组
    xorq %rax, %rax
retry: movzbq (%rcx), %rax // 非法访问; Page Fault      rax = *ptr (1B)
        shrq $0xc, %rax
        jz retry
        movq (%rbx, %rax), %rbx // (%rbx + (%rcx) * 4096)
```

$$\text{rax} = \text{rax} * 4096$$

$$\text{rbx} = \text{base}[\text{rax}]$$



End.



# 中断与分时多任务

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

为什么 `while(1);` 死循环不会把电脑卡死？

- 本讲内容
  - 中断机制
  - 实现分时多任务

# 中断机制

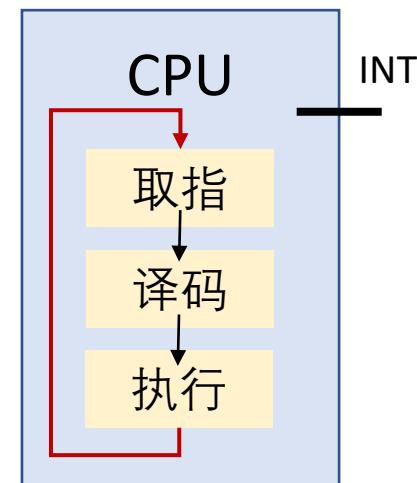
# 回到一个经典问题

为什么 `while (1);` 死循环不会把电脑卡死？

- 因为有中断
  - 相当于在指令之后插入一段代码

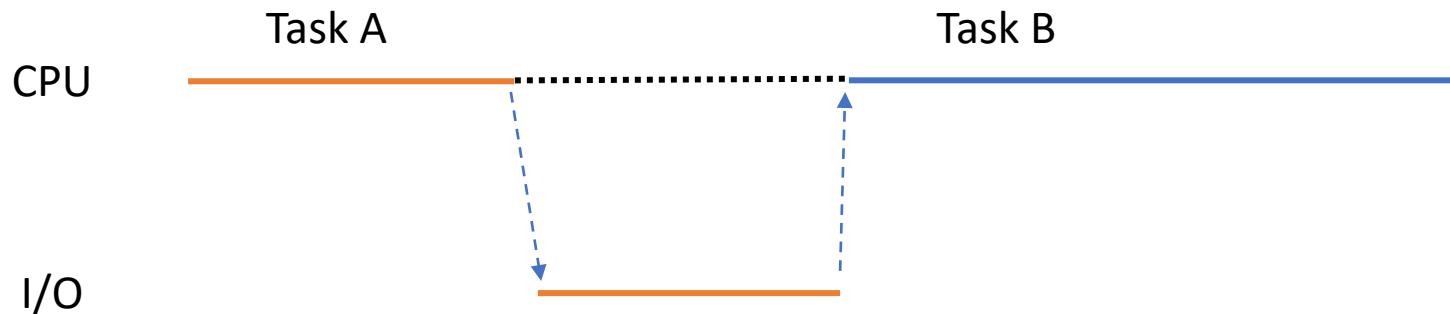
```
if (has_interrupt && int_enabled) {  
    interrupt_handler();  
}
```

- 类似的是异常机制
  - 有时候称为同步中断



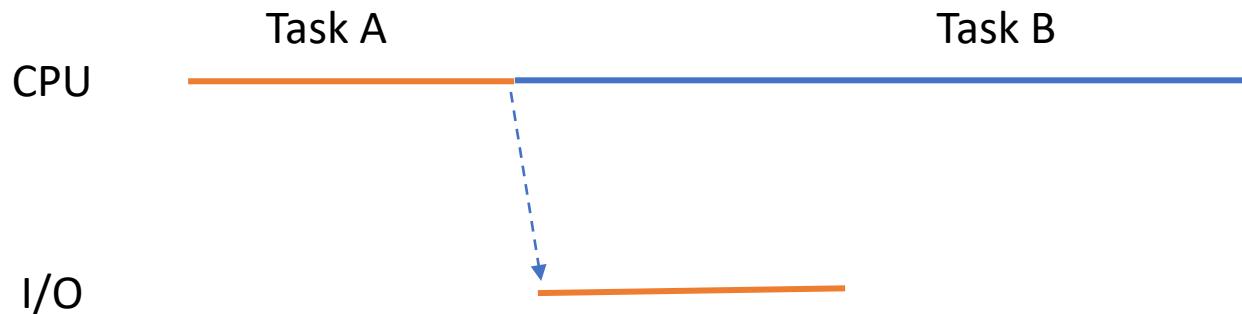
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



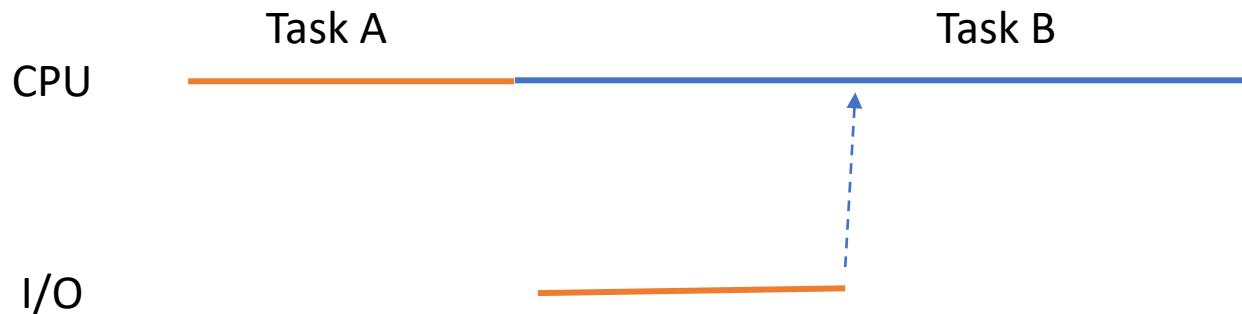
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了



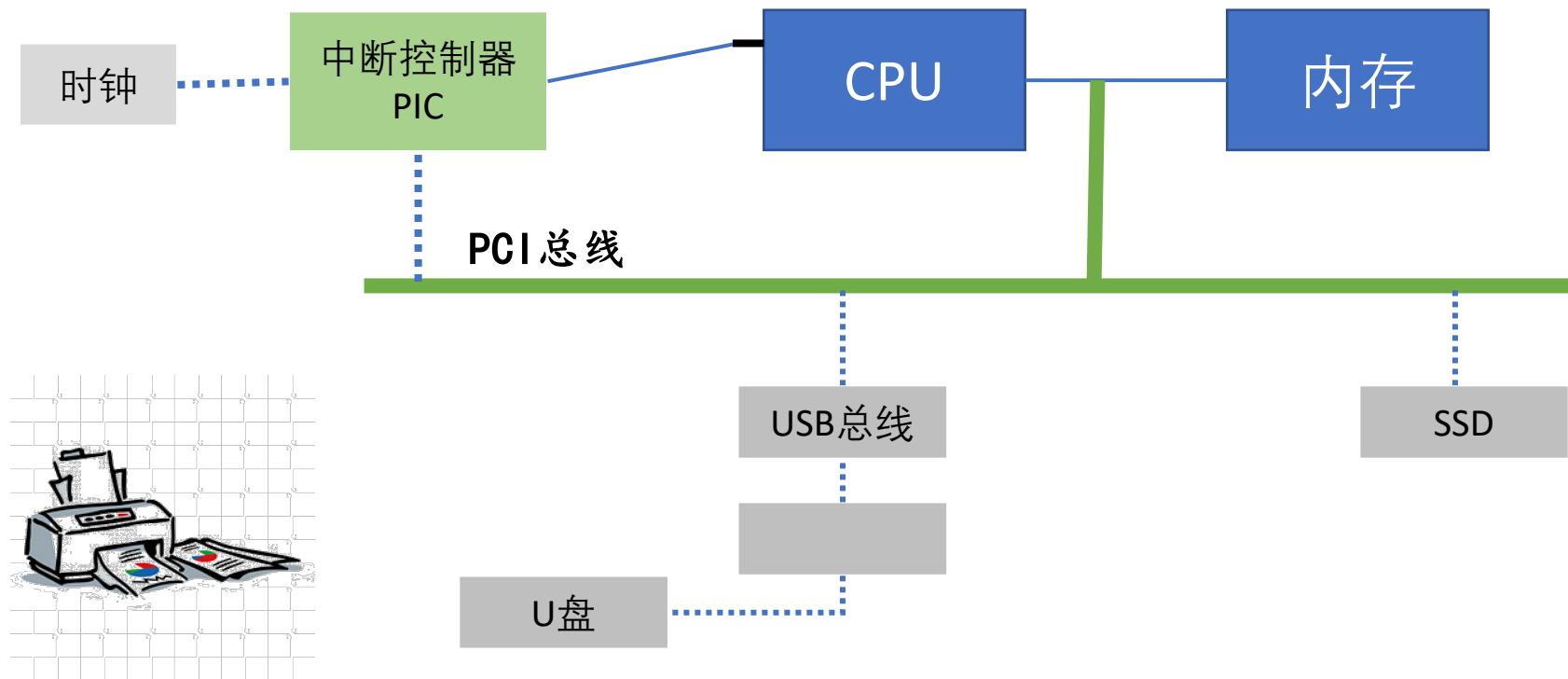
# 中断：弥补I/O设备的速度缺陷

- CPU cycles 实在太珍贵了
  - 不能用来浪费在等 I/O 设备完成上
    - 拥有机械部件的 I/O 设备相比于 CPU 来说实在太慢了
- 中断 = 硬件驱动的函数调用
  - 相当于在每条语句后都插入

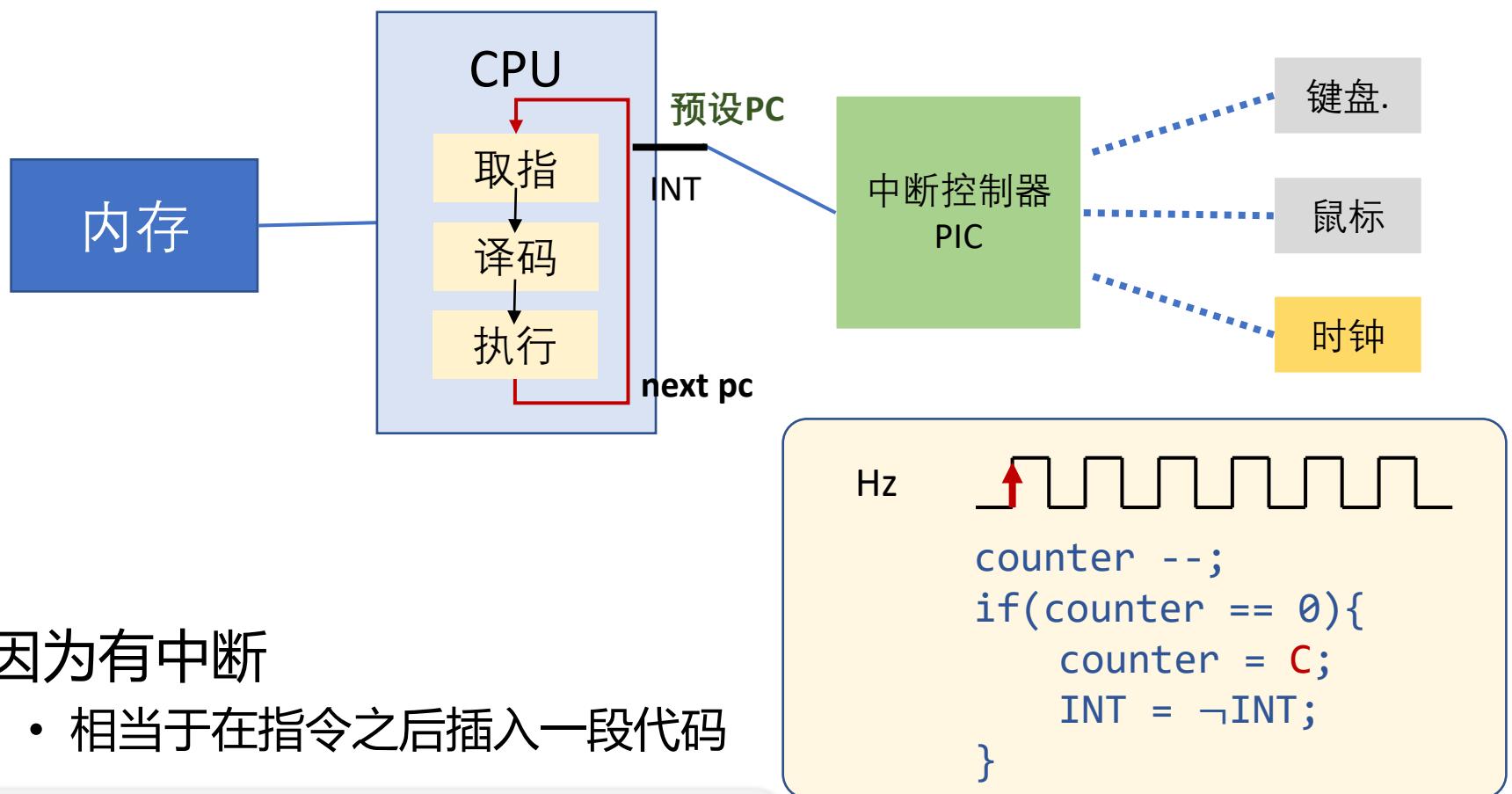
```
if (pending_io && int_enabled) {  
    interrupt_handler();  
    pending_io = 0;  
}
```

- (硬件上好像不太难实现)
  - 于是就有了最早的操作系统：管理I/O设备的库代码

# I/O设备回顾



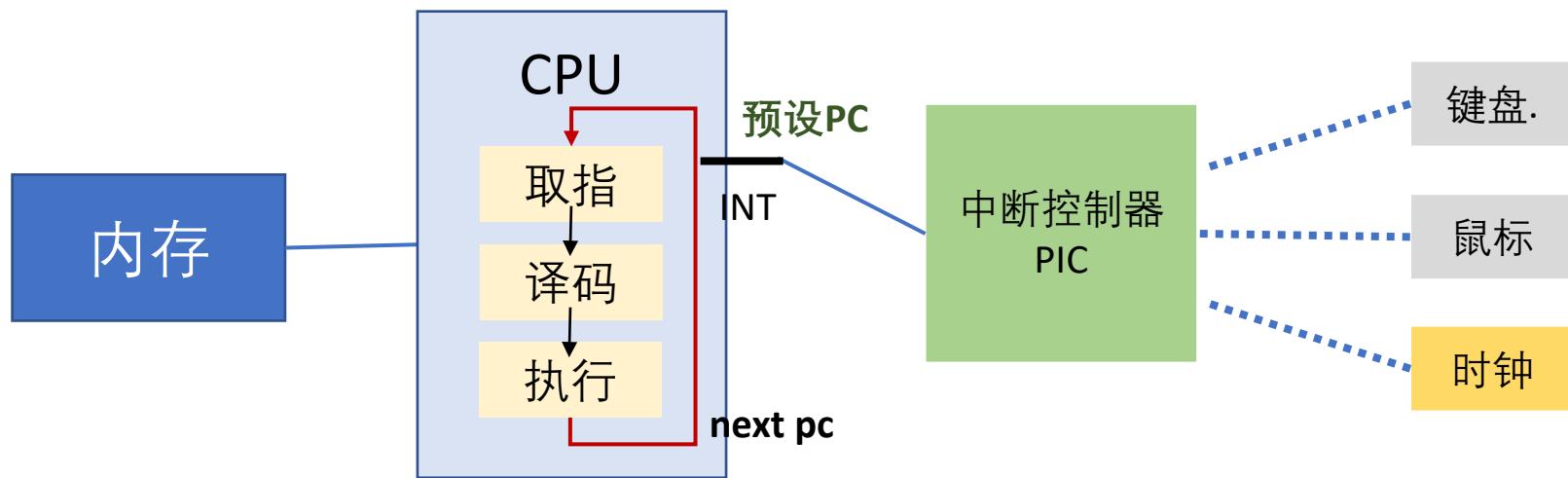
# 中断机制总览



- 因为有中断
  - 相当于在指令之后插入一段代码

```
if (has_interrupt && int_enabled) {  
    interrupt_handler();  
}
```

# INT指令和Segmentation fault

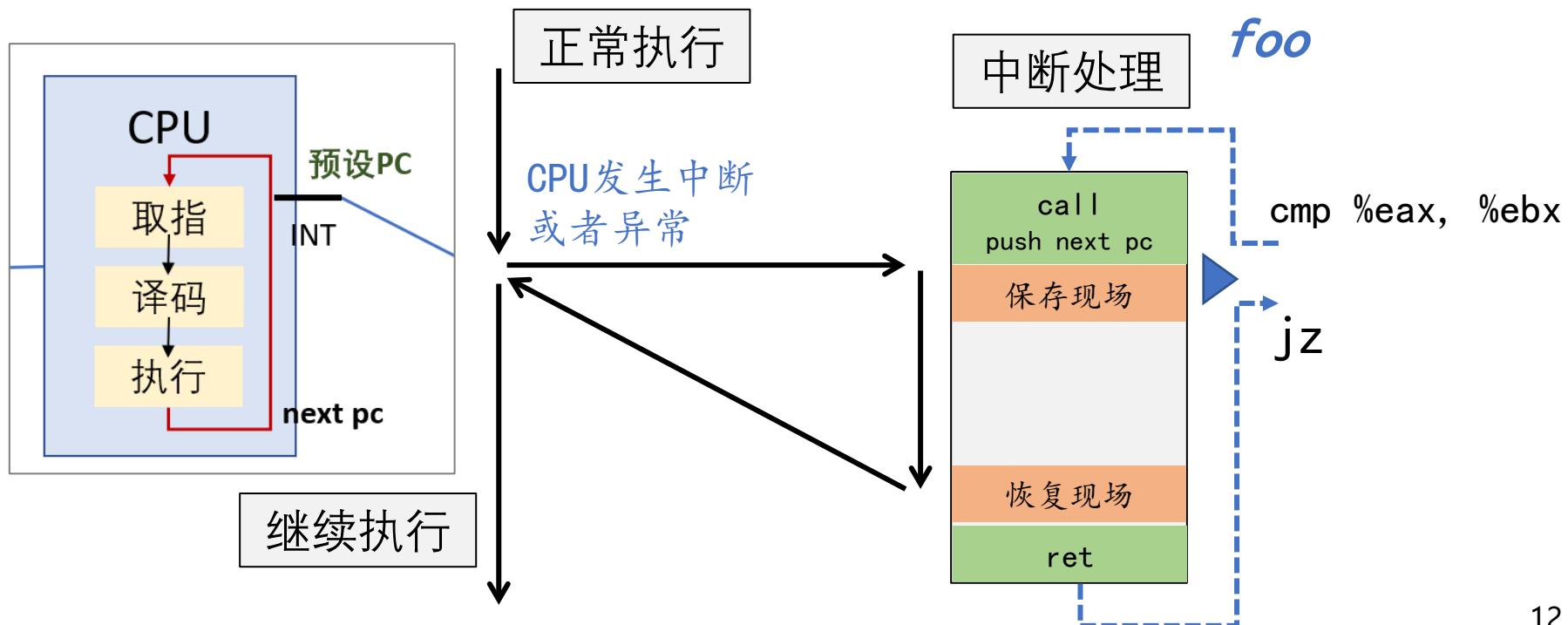


- 类似的是异常机制
  - 有时候称为同步中断

int \$0x80  
mov \$1, (%eax) → #14 中断  
(异常)

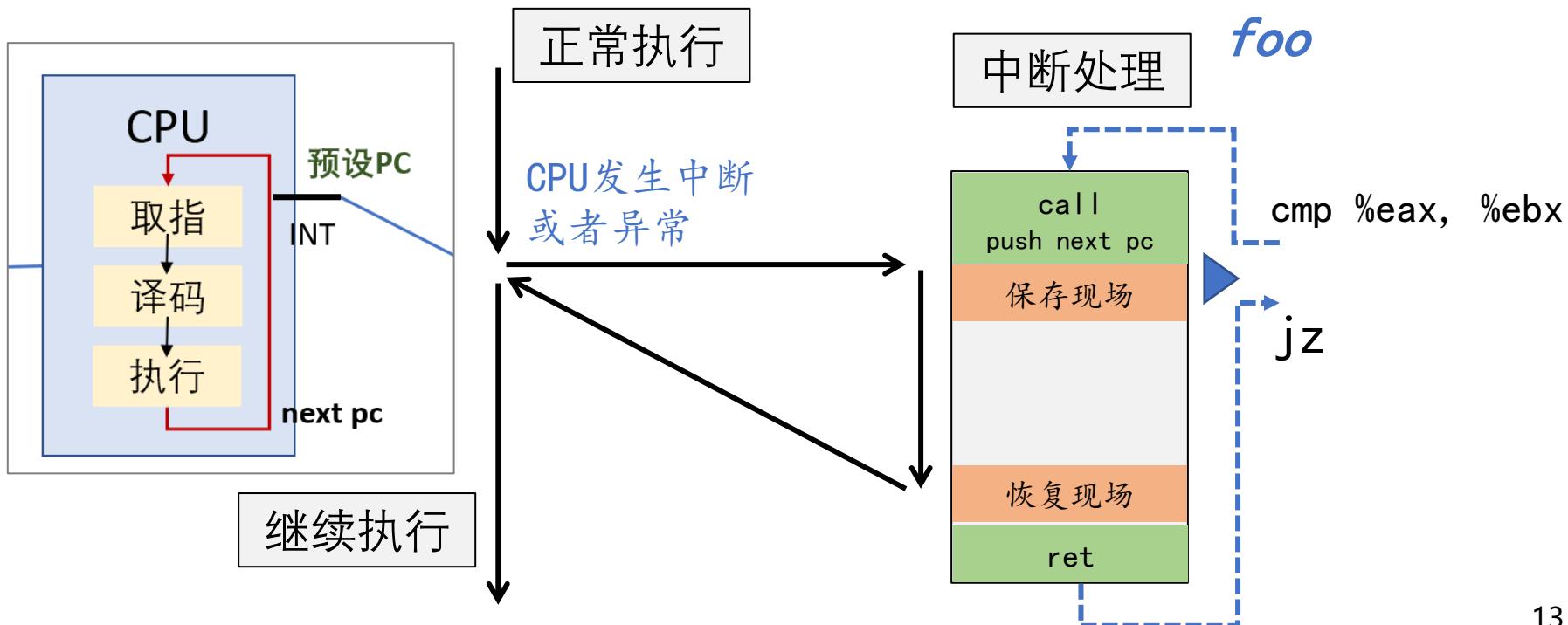
# 中断的实现

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)

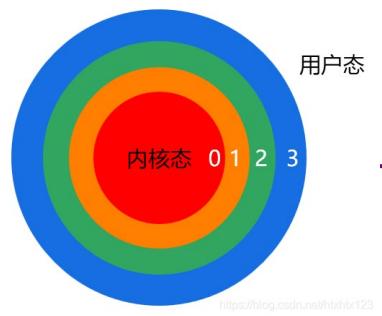


# 中断的实现

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)
  - 但中断不仅要保存 PC (尤其是在有特权级切换的时候)
- 中断处理
  - 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)



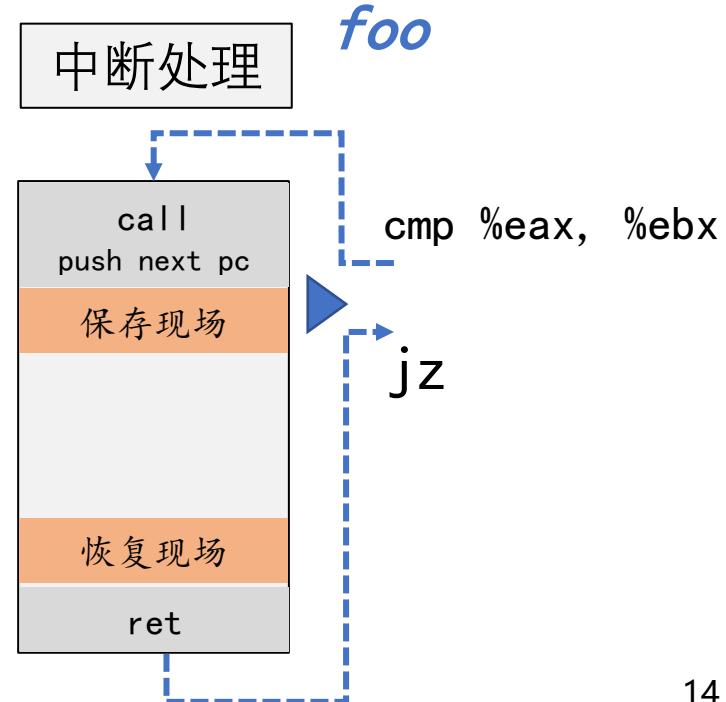
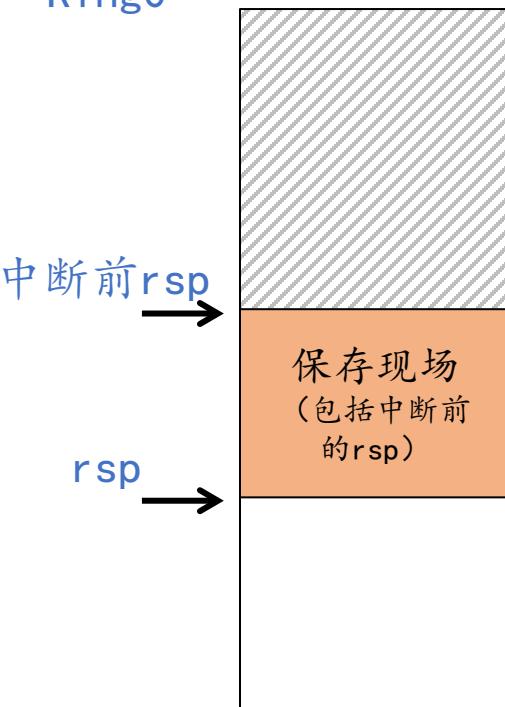
# 中断的实现



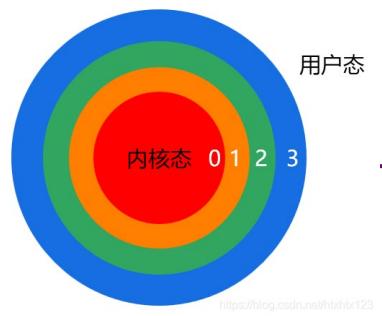
## • 中断处理

- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - int \$0x80 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断

Ring0

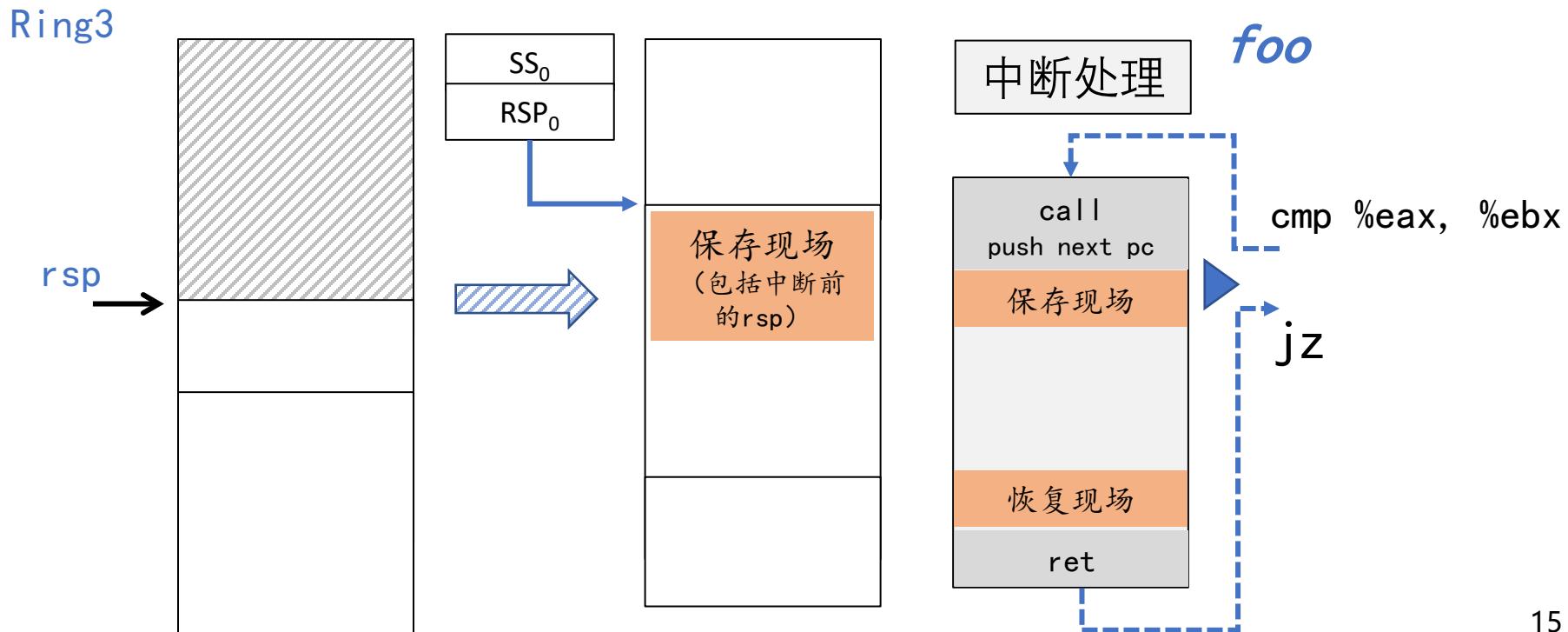


# 中断的实现



## • 中断处理

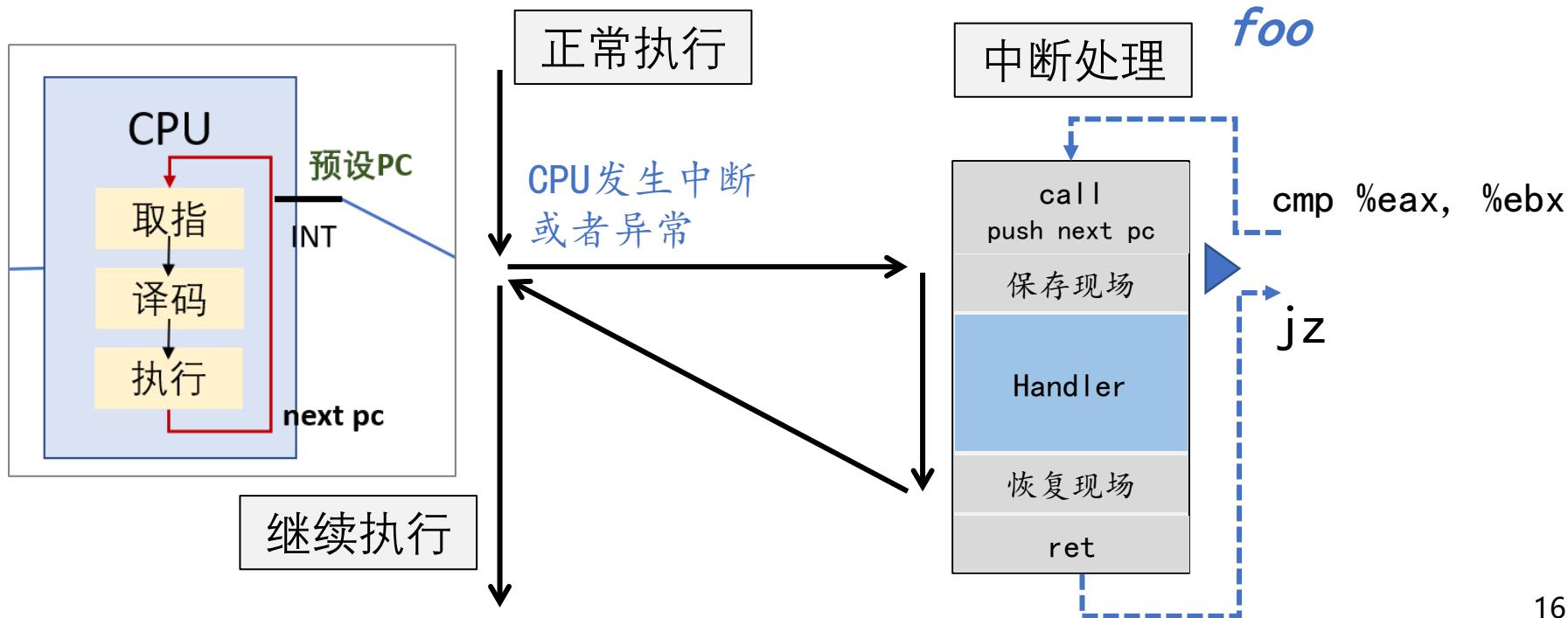
- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - int \$0x80 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断



# 中断的实现

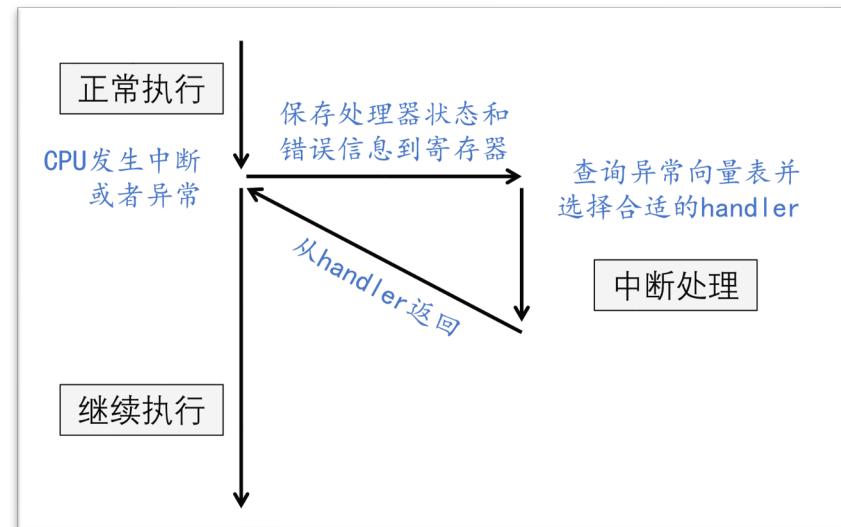
- 中断处理

- 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
- 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
  - int \$0x80 指令可以产生 128 号异常
  - 时钟会产生 32 号中断
  - 键盘会产生 33 号中断



# x86-64中断过程

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)
  - 但中断不仅要保存 PC (尤其是在有特权级切换的时候)
- 中断处理
  - 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)
  - 根据中断/异常号跳转到处理程序 (特权级切换会触发堆栈切换)
    - int \$0x80 指令可以产生 128 号异常
    - 时钟会产生 32 号中断
    - 键盘会产生 33 号中断
    - .....



# 为什么死循环不会把电脑卡死?

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))

## x86 Instruction Set Reference

### CLI

#### Clear Interrupt Flag

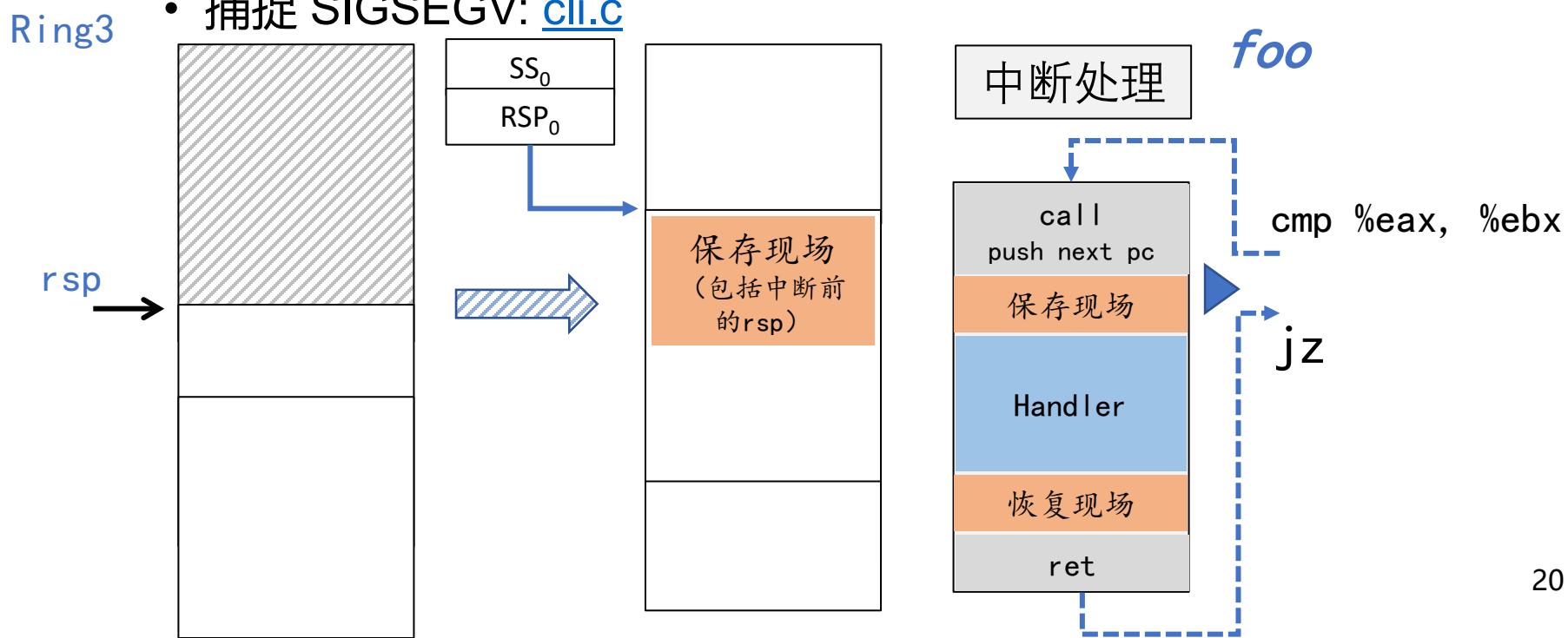
Opcode	Mnemonic	Description
FA	CLI	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Operation
<pre>if(PE == 0) IF = 0; //Reset Interrupt Flag else {     if(VM == 0) {         if(IOPL != CPL) IF = 0; //Reset Interrupt Flag         else {             if(IOPL &lt; CPL &amp;&amp; CPL &lt; 3 &amp;&amp; PVI == 1) VIF = 0; //Reset Virtual Interrupt Flag             else Exception(GP(0));         }     }     else {         if(IOPL == 3) IF = 0; //Reset Interrupt Flag         else {             if(IOPL &lt; 3 &amp;&amp; VME == 1) VIF = 0; //Reset Virtual Interrupt Flag             else Exception(GP(0));         }     } }</pre>

```
1 int main(){
2     asm volatile ("cli");
3     while(1);
4 }
```

# 为什么死循环不会把电脑卡死?

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))
    - 发生 13 号异常
    - Error Code 0
      - 异常机制进入操作系统代码执行
      - 操作系统发送 SIGSEGV 信号
  - 捕捉 SIGSEGV: [cli.c](#)





```
1 #define _GNU_SOURCE
2
3 #include <signal.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <ucontext.h>
7 #include <stdint.h>
8
9 void on_sigsegv(int sig, siginfo_t *info, void *ucontext) {
10    ucontext_t *ctx = (ucontext_t *)ucontext;
11    uint8_t *pc = (void *)ctx->uc_mcontext.gregs[REG_RIP];
12    printf("PC = %p ", pc);
13    for (int i = -8; i < 8; i++) {
14        printf(i == 0 ? "[%02x] " : "%02x ", pc[i]);
15    }
16    printf("\n");
17    exit(1);
18 }
19
20 int main() {
21    struct sigaction act;
22    act.sa_sigaction = on_sigsegv;
23    sigemptyset(&act.sa_mask);
24    act.sa_flags = SA_SIGINFO;
25    sigaction(SIGSEGV, &act, NULL);
26    asm volatile("cli");
27 }
```

~/Documents/ICS2021/teach/Course13/cli.c[1]

[c] unix utf-8 Ln 13, Col 2/27

=((foldclosed(line('.')) &lt; 0) ? 'zc' : 'zo')

```
$ ls  
a.c a.out cli.c thread-os.c  
$ vim cli.c  
$ gcc cli.c  
$ ./a.out  
PC = 0x5634316132ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b  
$ █
```

12e3:	bf 0b 00 00 00	mov \$0xb,%edi
12e8:	e8 c3 fd ff ff	call 10b0 <sigaction@plt>
12ed:	fa	cli
12ee:	b8 00 00 00 00	mov \$0x0,%eax
12f3:	48 8b 4d f8	mov -0x8(%rbp),%rcx

```
$ ls
a.c a.out cli.c thread-os.c
$ vim cli.c
$ gcc cli.c
$ ./a.out
PC = 0x5634316132ed  00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b
$
```

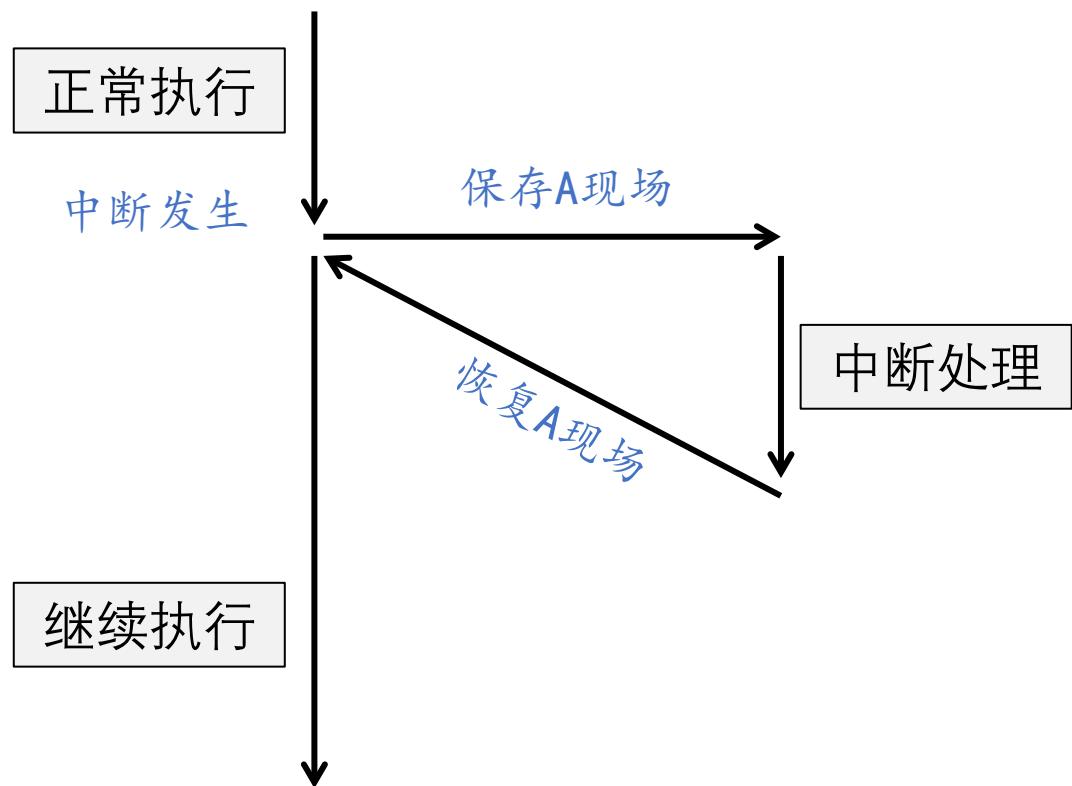
```
$ ./a.out  
PC = 0x5620206812ed 00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b  
$ ./a.out  
PC = 0x555cd8d502ed 00 00 00 e8 c3 fd ff ff [fa] b8 00 00 00 00 48 8b  
$ █
```

# 为什么死循环不会把电脑卡死?

- 中断是强制的 (普通的进程不能关闭)
  - Ring 3 关闭中断将导致 Exception(GP(0)) ([手册](#))
    - 发生 13 号异常
    - Error Code 0
      - 异常机制进入操作系统代码执行
      - 操作系统发送 SIGSEGV 信号
    - 捕捉 SIGSEGV: [cli.c](#)
- 中断发生后, 操作系统代码将会切换到另一个进程执行
  - “调度” : 让进程公平地共享 CPU

# 实现分时多任务

A



A

正常执行

中断发生

保存A现场

中断处理

继续执行

恢复A现场

恢复B现场

正常执行

继续执行

# 代码选讲

---

- Abstract Machine
  - trap64.S
  - cte.c
- 调试 “迷你” 操作系统
  - [thread-os.c](#)
    - -s -S 启动 QEMU
    - gdb target remote localhost:1234
      - [gdbnotes for CSAPP](#)

```
1 #include <am.h>
2 #include <klib.h>
3 #include <klib-macros.h>
4
5 typedef union task {
6     struct {
7         const char *name;
8         union task *next;
9         void (*entry)(void *);
10        Context *context;
11    };
12    uint8_t stack[8192];
13 } Task;
14
15 Task *current;
16
17 void func(void *arg) {
18     while (1) {
19         putch(*((char *)arg));
20         for (int volatile i = 0; i < 100000; i++) ;
21     }
22 }
23
24 Task tasks[] = {
25     { .name = "a", .entry = func },
26     { .name = "b", .entry = func },
27     { .name = "c", .entry = func },

```

```
24 Task tasks[] = {  
25     { .name = "a", .entry = func },  
26     { .name = "b", .entry = func },  
27     { .name = "c", .entry = func },  
28 };  
29  
30 Context *on_interrupt(Event ev, Context *ctx) {  
31     if (!current) current = &tasks[0];  
32     else           current->context = ctx;  
33     current = current->next;  
34     return current->context;  
35 }  
36  
37 int main() {  
38     ioe_init();  
39     cte_init(on_interrupt);  
40  
41     for (int i = 0; i < LENGTH(tasks); i++) {  
42         Task *task      = &tasks[i];  
43         Area stack    = (Area) { &task->context + 1, task + 1 };  
44         task->context = kcontext(stack, task->entry, (void *)task->name);  
45         task->next     = &tasks[(i + 1) % LENGTH(tasks)];  
46     }  
47     iset(true);  
48     while(1);  
49 }
```

<s2021/am-kernels/kernels/thread-os/thread-os.c[1] [c] unix utf-8 Ln 49, Col 1/49

管理 控制 视图 热键 设备 帮助

\$ █



```
$ ls  
build Makefile thread-os.c  
$ █
```

管理 控制 视图 热键 设备 帮助

```
$ ls  
thread-os-x86_64-qemu  thread-os-x86_64-qemu.elf  x86_64-qemu  
$ █
```

```
/home/why/Documents/ICS2021/ics2021/am-kernels/kernels/thread-os/thread-os.c
30     Context *on_interrupt(Event ev, Context *ctx) {
31         if (!current) current = &tasks[0];
32         else           current->context = ctx;
33         current = current->next;
34         return current->context;
35     }
36
B+ 37     int main() {
38         ioe_init();
>39         cte_init(on_interrupt);
40
41         for (int i = 0; i < LENGTH(tasks); i++) {
42             Task *task    = &tasks[i];
43             Area stack   = (Area) { &task->context + 1, task + 1 };
44             task->context = kcontext(stack, task->entry, (void *)task->na
45             task->next    = &tasks[(i + 1) % LENGTH(tasks)];
```

remote Thread 1.1 In: main

L39 PC: 0x1000d3

(gdb) n

(gdb) █

```

0x1016e0 <iset>                                endbr64
0x1016e4 <iset+4>                               test    %dil,%dil
0x1016e7 <iset+7>                               je     0x1016f0 <iset+16>
0x1016e9 <iset+9>                               sti
>0x1016ea <iset+10>                            ret
0x1016eb <iset+11>                               nopl   0x0(%rax,%rax,1)
0x1016f0 <iset+16>                               cli
0x1016f1 <iset+17>                               ret
0x1016f2                                     data16 nopw %cs:0x0(%rax,%rax,1)
0x1016fd                                     nopl   (%rax)
0x101700 <__am_panic_on_return>                endbr64
0x101704 <__am_panic_on_return+4>              push   %rbx
0x101705 <__am_panic_on_return+5>              mov    $0x41,%edi
0x10170a <__am_panic_on_return+10>             lea    0x1239(%rip),%rbx      # 0x
0x101711 <__am_panic_on_return+17>             nopl   0x0(%rax)
0x101718 <__am_panic_on_return+24>             call   0x1002b0 <putch>

```

remote Thread 1.1 In: iset L?? PC: 0x1016ea

rip	0x1016ea	0x1016ea <iset+10>
eflags	0x202	[ IOPL=0 IF ]
cs	0x8	8
ss	0x0	0
ds	0x0	0
es	0x0	0
fs	0x0	0
gs	0x0	0

--Type <RET> for more, q to quit, c to continue without paging--

0x100132 <main+130>	sub	%rdx,%rax
0x100135 <main+133>	shl	\$0xd,%rax
0x100139 <main+137>	add	%rbp,%rax
0x10013c <main+140>	mov	%rax,-0x3ff8(%rbx)
0x100143 <main+147>	cmp	\$0x4,%r13
0x100147 <main+151>	jne	0x1000ed <main+61>
0x100149 <main+153>	mov	\$0x1,%edi
0x10014e <main+158>	call	0x1016e0 <iset>
>0x100153 <main+163>	jmp	0x100153 <main+163>
0x100155	nopw	%cs:0x0(%rax,%rax,1)
0x10015f	nop	
0x100160 <on_interrupt>	endbr64	
0x100164 <on_interrupt+4>	mov	\$0x109700,%rdx
0x10016b <on_interrupt+11>	mov	(%rdx),%rax
0x10016e <on_interrupt+14>	test	%rax,%rax
0x100171 <on_interrupt+17>	je	0x100188 <on_interrupt+40>

remote Thread 1.1 In: main

L48 PC: 0x100153

cr2	0x0	0
cr3	0x1000	[ PDBR=0 PCID=0 ]
cr4	0x20	[ PAE ]
cr8	0x0	0

--Type &lt;RET&gt; for more, q to quit, c to continue without paging--q

Quit

(gdb) si

(gdb) si

(gdb)

## Useful information

backtrace	Print the current address and stack backtrace
where	Print the current address and stack backtrace
info program	Print current status of the program)
info functions	Print functions in program
info stack	Print backtrace of the stack)
info frame	Print information about the current stack frame
info registers	Print registers and their contents
info breakpoints	Print status of user-settable breakpoints
display /FMT EXPR	Print expression EXPR using format FMT every time GDB stops
undisplay	Turn off display mode
help	Get information about gdb

x/zw \$rsp	Examine two (4-byte) words starting at address in \$rsp
x/2wd \$rsp	Examine two (4-byte) words starting at address in \$rsp. Print in decimal
x/g \$rsp	Examine (8-byte) word starting at address in \$rsp.
x/gd \$rsp	Examine (8-byte) word starting at address in \$rsp. Print in decimal
x/a \$rsp	Examine address in \$rsp. Print as offset from previous global symbol.
x/s 0xbffff890	Examine a string stored at 0xbffff890
x/20b sum	Examine first 20 opcode bytes of function sum
x/10i sum	Examine first 10 instructions of function sum

0x100118	<main+104>	mov	%r13,%rax
0x10011b	<main+107>	mul	%r12
0x10011e	<main+110>	mov	%rdx,%rax
0x100121	<main+113>	and	\$0xfffffffffffffe,%rdx
0x100125	<main+117>	shr	%rax
0x100128	<main+120>	add	%rax,%rdx
0x10012b	<main+123>	mov	%r13,%rax
0x10012e	<main+126>	add	\$0x1,%r13
0x100132	<main+130>	sub	%rdx,%rax
0x100135	<main+133>	shl	\$0xd,%rax
0x100139	<main+137>	add	%rbp,%rax
0x10013c	<main+140>	mov	%rax,-0x3ff8(%rbx)
0x100143	<main+147>	cmp	\$0x4,%r13
0x100147	<main+151>	jne	0x1000ed <main+61>
0x100149	<main+153>	mov	\$0x1,%edi
0x10014e	<main+158>	call	0x1016e0 <iset>

remote Thread 1.1 In: \_\_am\_irq32 L?? PC: 0x102855

rip	0x102855	0x102855 <__am_irq32>
eflags	0x202	[ IOPL=0 IF ]
cs	0x8	8
ss	0x0	0
ds	0x0	0
es	0x0	0
fs	0x0	0
gs	0x0	0

--Type <RET> for more, q to quit, c to continue without paging-- █

0x100118	<main+104>	mov	%r13,%rax
0x10011b	<main+107>	mul	%r12
0x10011e	<main+110>	mov	%rdx,%rax
0x100121	<main+113>	and	\$0xfffffffffffffe,%rdx
0x100125	<main+117>	shr	%rax
0x100128	<main+120>	add	%rax,%rdx
0x10012b	<main+123>	mov	%r13,%rax
0x10012e	<main+126>	add	\$0x1,%r13
0x100132	<main+130>	sub	%rdx,%rax
0x100135	<main+133>	shl	\$0xd,%rax
0x100139	<main+137>	add	%rbp,%rax
0x10013c	<main+140>	mov	%rax,-0x3ff8(%rbx)
0x100143	<main+147>	cmp	\$0x4,%r13
0x100147	<main+151>	jne	0x1000ed <main+61>
0x100149	<main+153>	mov	\$0x1,%edi
0x10014e	<main+158>	call	0x1016e0 <iset>

remote Thread 1.1 In: \_\_am\_irq32 L?? PC: 0x102855

fs 0x0 0  
gs 0x0 0

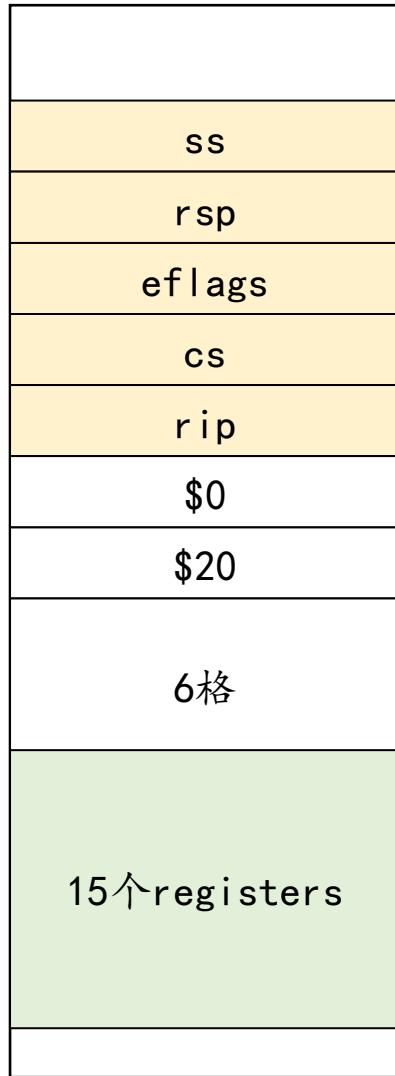
--Type <RET> for more, q to quit, c to continue without paging--q

Quit

(gdb) x/5gx \$rsp

0x20c848	<__am_cpuinfo+8264>:	0x00000000000100153	0x000000000000000000000008
0x20c858	<__am_cpuinfo+8280>:	0x000000000000000202	0x00000000000020c870
0x20c868	<__am_cpuinfo+8296>:	0x000000000000000000000000	

(gdb) █



```

void __am_irq_handle(struct trap_frame *tf) {
    Context *saved_ctx = &tf->saved_context;
    Event ev = {
        .event = EVENT_NULL,
        .cause = 0, .ref = 0,
        .msg = "(no message)",
    };

#if __x86_64
    saved_ctx->rip      = tf->rip;
    saved_ctx->cs       = tf->cs;
    saved_ctx->rflags   = tf->rflags;
    saved_ctx->rsp      = tf->rsp;
    saved_ctx->rsp0     = CPU->tss.rsp0;
    saved_ctx->ss       = tf->ss;
#else
    saved_ctx->eip      = tf->eip;
    saved_ctx->cs       = tf->cs;
    saved_ctx->eflags   = tf->eflags;
    saved_ctx->esp0     = CPU->tss.esp0;
    saved_ctx->ss3      = USEL(SEG_UDATA);
    // no ss/esp saved for DPL_KERNEL
    saved_ctx->esp = (tf->cs & DPL_USER ? tf->esp : (uint32_t)(tf + 1) - 8);
#endif
    saved_ctx->cr3      = (void *)get_cr3();
}

```

0x102770	<trap+13>	push	%r11
0x102772	<trap+15>	push	%r10
0x102774	<trap+17>	push	%r9
0x102776	<trap+19>	push	%r8
0x102778	<trap+21>	push	%rdi
0x102779	<trap+22>	push	%rsi
0x10277a	<trap+23>	push	%rbp
0x10277b	<trap+24>	push	%rdx
0x10277c	<trap+25>	push	%rcx
0x10277d	<trap+26>	push	%rbx
0x10277e	<trap+27>	push	%rax
0x10277f	<trap+28>	push	\$0x0
0x102781	<trap+30>	mov	%rsp,%rdi
>0x102784	<trap+33>	call	0x100750 < am_irq_handle>
0x102789	< am_iret>	mov	%rdi,%rsp
0x10278c	< am_iret+3>	mov	0xa0(%rsp),%rax

remote Thread 1.1 In: trap

L?? PC: 0x102784

```

0x0000000000010277a in trap ()
0x0000000000010277b in trap ()
0x0000000000010277c in trap ()
0x0000000000010277d in trap ()
0x0000000000010277e in trap ()
0x0000000000010277f in trap ()
0x00000000000102781 in trap ()
0x00000000000102784 in trap ()

```

(gdb) █

```

>0x102789 < am_iret>    mov    %rdi,%rsp
0x10278c < _am_iret+3>   mov    0xa0(%rsp),%rax
0x102794 < _am_iret+11>  mov    %eax,%ds
0x102796 < _am_iret+13>  mov    %eax,%es
0x102798 < _am_iret+15>  add    $0x8,%rsp
0x10279c < _am_iret+19>  pop    %rax
0x10279d < _am_iret+20>  pop    %rbx
0x10279e < _am_iret+21>  pop    %rcx
0x10279f < _am_iret+22>  pop    %rdx
0x1027a0 < _am_iret+23>  pop    %rbp
0x1027a1 < _am_iret+24>  pop    %rsi
0x1027a2 < _am_iret+25>  pop    %rdi
0x1027a3 < _am_iret+26>  pop    %r8
0x1027a5 < _am_iret+28>  pop    %r9
0x1027a7 < _am_iret+30>  pop    %r10
0x1027a9 < _am_iret+32>  pop    %r11

```

remote Thread 1.1 In: \_am\_iret

L?? PC: 0x102789

```

0x000000000001008a0 in _am_irq_handle ()
0x000000000001008a4 in _am_irq_handle ()
0x000000000001008a7 in _am_irq_handle ()
0x000000000001008a8 in _am_irq_handle ()
0x000000000001008a9 in _am_irq_handle ()
0x00000000000102789 in _am_iret ()

```

(gdb) ■



```

0x1001a8 <func+8>      sub    $0x10,%rsp
0x1001ac <func+12>     nopl   0x0(%rax)
0x1001b0 <func+16>     movsbl (%rbx),%edi
0x1001b3 <func+19>     call   0x1002b0 <putch>
0x1001b8 <func+24>     movl   $0x0,0xc(%rsp)
0x1001c0 <func+32>     mov    0xc(%rsp),%eax
0x1001c4 <func+36>     cmp    $0x1869f,%eax
0x1001c9 <func+41>     jg    0x1001b0 <func+16>
>0x1001cb <func+43>    mov    0xc(%rsp),%eax
0x1001cf <func+47>     add    $0x1,%eax
0x1001d2 <func+50>     mov    %eax,0xc(%rsp)
0x1001d6 <func+54>     mov    0xc(%rsp),%eax
0x1001da <func+58>     cmp    $0x1869f,%eax
0x1001df <func+63>     jle   0x1001cb <func+43>
0x1001e1 <func+65>     jmp   0x1001b0 <func+16>
0x1001e3                  nopw   %cs:0x0(%rax,%rax,1)

```

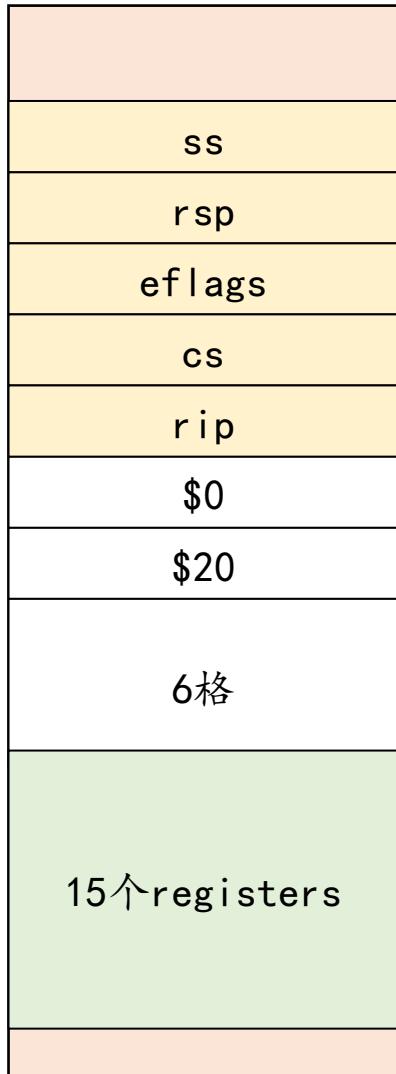
remote Thread 1.1 In: func L20 PC: 0x1001cb

```

(gdb) x/5gx $rsp
0x107258 <tasks+16248>: 0x000000000001001cb      0x000000000000000000000008
0x107268 <tasks+16264>: 0x0000000000000000297     0x000000000001072c0
0x107278 <tasks+16280>: 0x000000000000000000000000
(gdb) si
0x000000000001001cb in func (arg=0x102922)
  at /home/why/Documents/ICS2021/ics2021/am-kernels/kernels/thread-os/thread-os.
c:20
(gdb) 

```

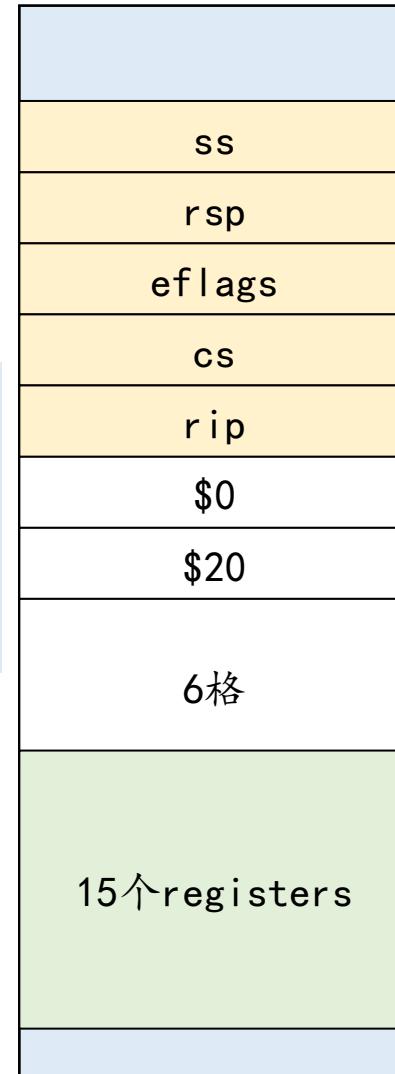
A现场  
(包括中断前的rsp)



保存A现场

中  
断  
处  
理

B现场  
(包括中断前的rsp)



恢复B现场

# 中断 + 更大的内存 = 分时多线程

```
void foo() { while (1) printf("a"); }
void bar() { while (1) printf("b"); }
```

- 能够让foo()和bar()“同时”在处理器上执行?
  - 借助每条语句后被动插入的interrupt\_handler()调用

```
void interrupt_handler() {
    dump_regs(current->regs);
    current = (current->func == foo) ? bar : foo;
    restore_regs(current->regs);
}
```

- 操作系统背负了“调度”的职责

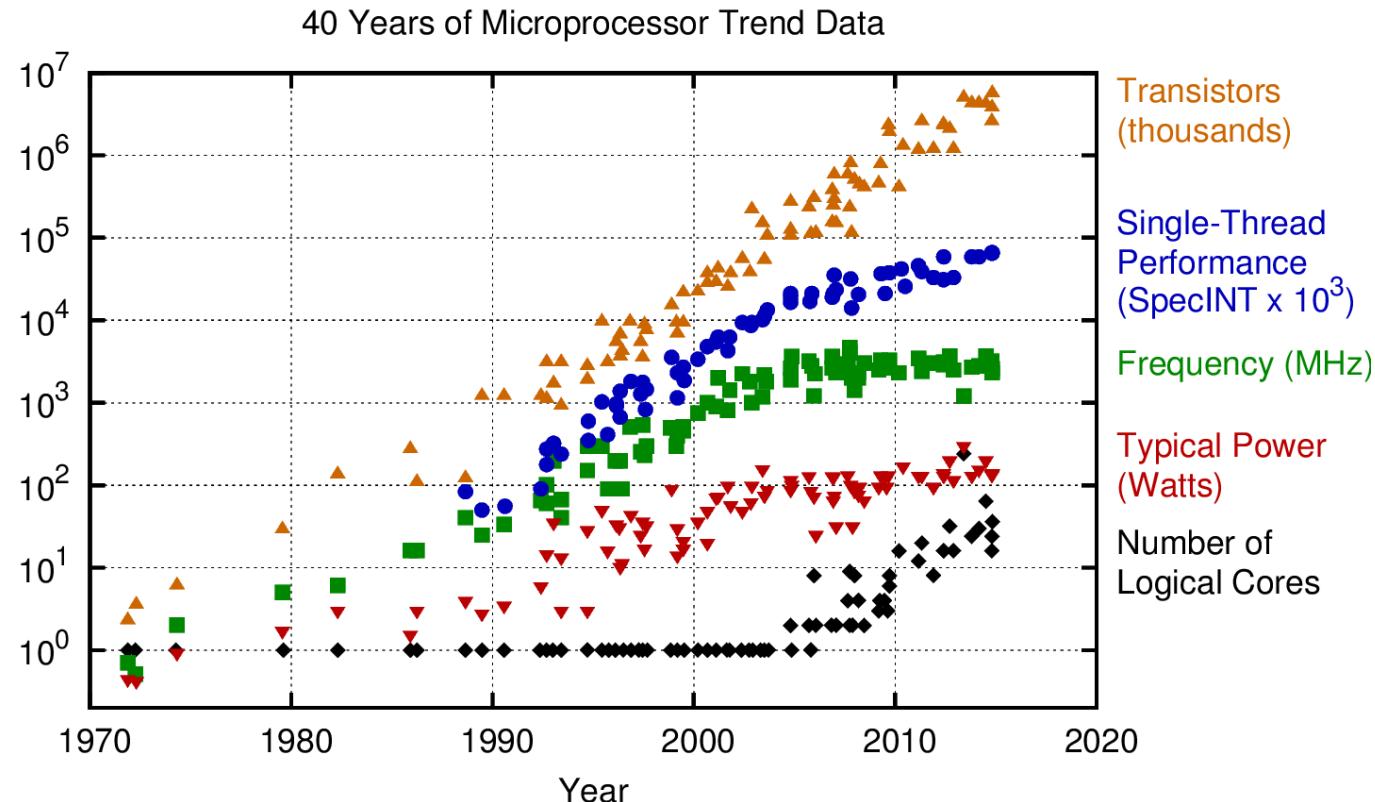
# 分时多线程 + 虚拟存储 = 进程

---

- 让`foo()`和`bar()`的执行互相不受影响
  - 在计算机系统里增加映射函数 $f_{\text{foo}}, f_{\text{bar}}$
  - `foo`访问内存地址 $m$ 时，将被重定位到 $f_{\text{foo}}(m)$
  - `bar`访问内存地址 $m$ 时，将被重定位到 $f_{\text{bar}}(m)$
- `foo`和`bar`本身无权管理 $f$
- 操作系统需要完成进程、存储、文件的管理
- UNIX诞生（就是我们今天的进程；Android app；...）
  - `gcc a.c`
  - `readelf -a a.out` → 二进制文件的全部信息

# 故事其实没有停止.

- 面对有限的功耗、难以改进的制程、无法提升的频率、应用需求
  - 多处理器、big.LITTLE、异构处理器 (GPU、NPU、...)
  - 单指令多数据(MMX, SSE, AVX, ...), 虚拟化(VT; EL0/1/2/3), ..., 安全执行环节 (TrustZone; SGX), ...



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

# 总结

# 操作系统：中断驱动的上下文切换

- 应用程序视角

- 一组系统调用的 API
    - 进程管理、存储管理、文件管理.....

- 硬件视角

- 操作系统是一个中断处理程序
    - 设备中断：上下文切换；调用设备驱动程序； .....
    - 系统调用：操作系统代码执行 (API 实现)
    - 异常：发送信号 (类似系统调用)

End.

# “造轮子”的方法和乐趣

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

《计算机系统基础》课到底学了啥？

- 本讲内容

- 一个关于编译运行的问题
- 那些年我们造过的轮子

# 一个关于编译运行的问题

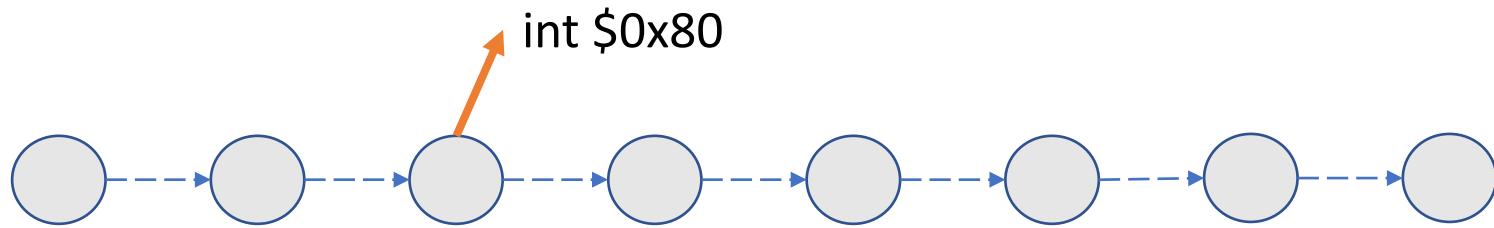
# 回到第一次C语言课

- 在IDE里，  
    • 编译、链接 → a.out  
    • 加载执行  
现在，一位同学对这个过程提出了质疑  
    • 我不信！我就觉得是 gcc 一个程序直接搞定的  
        • 道理上完全可以这么实现  
    • 如何说服这位同学？

```
$ ls -l
total 4
-rw-rw-r-- 1 why why 14 12月 16 08:19 a.c
$ gcc a.c
$ ls -l
total 20
-rw-rw-r-- 1 why why      14 12月 16 08:19 a.c
-rwxrwxr-x 1 why why 15912 12月 16 08:19 a.out
```

→ a.out

# 所有程序的执行都是状态机



```
$ ls -l
total 4
-rw-rw-r-- 1 why why 14 12月 16 08:19 a.c
$ gcc a.c
$ ls -l
total 20
-rw-rw-r-- 1 why why      14 12月 16 08:19 a.c
-rwxrwxr-x 1 why why 15912 12月 16 08:19 a.out
```

# 答案：用工具！

- 观察 trace: 查看 gcc 是否调用了其他命令

- strace -f -qq gcc a.c 2>&1 | vim -
  - 可以使用 grep (shell) 或 :g! (vim) 筛选感兴趣的系统调用

STRACE(1)

General Commands Manual

STRACE(1)

## NAME

strace - trace system calls and signals

## SYNOPSIS

```
strace [-ACdffhikqqrrttTvvVwxxyyzz] [-I n] [-b execve] [-e expr]...
[-O overhead] [-S sortby] [-U columns] [-a column] [-o file]
[-s strsize] [-X format] [-P path]... [-p pid]...
[--seccomp-bpf] { -p pid | [-DDD] [-E var[=val]]...
[-u username] command [args] }
```

```
strace -c [-dfwzZ] [-I n] [-b execve] [-e expr]... [-O overhead]
[-S sortby] [-U columns] [-P path]... [-p pid]...
[--seccomp-bpf] { -p pid | [-DDD] [-E var[=val]]...
[-u username] command [args] }
```

## DESCRIPTION

In the simplest case **strace** runs the specified command until it exits. It intercepts and records the system calls which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed on standard error or to the file specified with the **-o** option.

**strace** is a useful diagnostic, instructional, and debugging tool. System administrators, diagnosticians and trouble-shooters will find

Manual page strace(1) line 1 (press h for help or q to quit)

管理 控制 视图 热键 设备 帮助

```
$ ls  
a.c  a.out  gdb-internals.txt  
$ █
```



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```
$ strace gcc a.c 2>&1 | vim -
```



```

201 readlink("/usr/bin/gcc-10", "x86_64-linux-gnu-gcc-10", 1023) = 23
202 readlink("/usr/bin/x86_64-linux-gnu-gcc-10", 0x7ffe8a430820, 1023) = -1 EINVAL (I
203 openat(AT_FDCWD, "/tmp/ccIx8ZMR.s", O_RDWR|O_CREAT|O_EXCL, 0600) = 4
204 close(4)
205 [No Name] [+1]

```

STRACE(1)

General Commands Manual

STRACE(1)

**NAME**

strace - trace system calls and signals

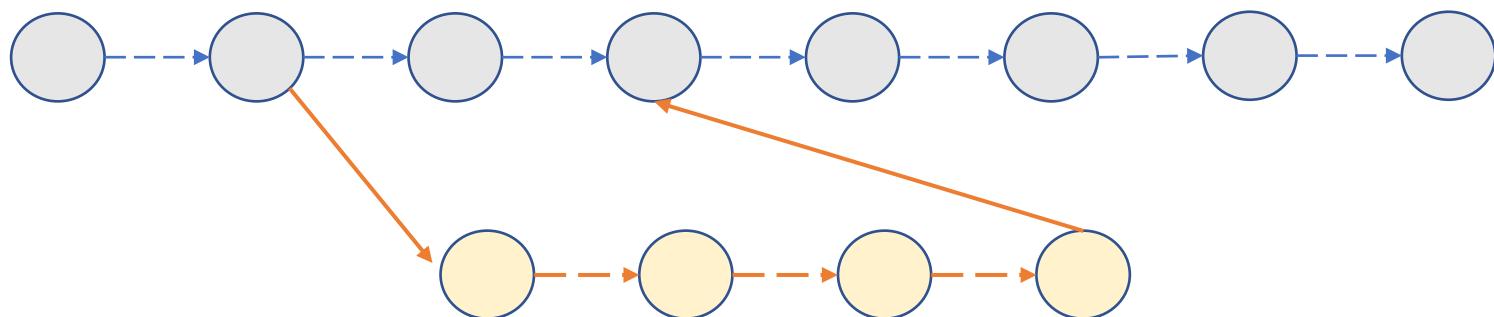
**-f****--follow-forks**

Trace child processes as they are created by currently traced processes as a result of the **fork(2)**, **vfork(2)** and **clone(2)** system calls. Note that **-p PID -f** will attach all threads of process **PID** if it is multi-threaded, not only thread with **thread\_id = PID**.

```

224 vfork()                      = 17244
225 close(5)                     = 0
[No Name] [+1]                   [strace] unix utf-8 Ln 201, Col 1/234

```



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

\$ man strace



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```

249 newfstatat(AT_FDCWD, "/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper", {st_mode=S_IFREG|0755, st_size=655432, ...}, 0) = 0
250 access("/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper", X_OK) = 0
251 access("/tmp", R_OK|W_OK|X_OK) = 0
252 openat(AT_FDCWD, "/tmp/ccFnrvKs.s", O_RDWR|O_CREAT|O_EXCL, 0600) = 4
253 close(4) = 0
254 newfstatat(AT_FDCWD, "/usr/lib/gcc/x86_64-linux-gnu/10/cc1", {st_mode=S_IFREG|0755, st_size=25697696, ...}, 0) = 0
255 access("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", X_OK) = 0
256 pipe2([4, 5], O_CLOEXEC) = 0
257 vfork(<unfinished ...>
258 [pid 17480] close(4) = 0
259 [pid 17480] execve("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", [/usr/lib/gcc/x86_64-linux-gnu/10"..., "-quiet", "-imultiarch", "x86_64-linux-gnu", "a.c", "-quiet", "-dumpbase", "a.c", "-mtune=generic", "-march=x86-64", "-auxbase", "a", "-fasynchronous-unwind-tables", "-fstack-protector-strong", "-Wformat", "-Wformat-security", "-fstack-clash-protection", "-fcf-protection", "-o", "/tmp/ccFnrvKs.s"], 0x18ec000 /* 66 vars */ <unfinished ...>
260 [pid 17479] <... vfork resumed> = 17480
261 [pid 17479] close(5) = 0
262 [pid 17479] read(4, "", 16) = 0
263 [pid 17480] <... execve resumed> = 0
264 [pid 17479] close(4) = 0
265 [pid 17480] brk(NULL <unfinished ...>
266 [pid 17479] wait4(17480, <unfinished ...>

[No Name] [+1] [strace] unix utf-8 Ln 266, Col 1/2923

```

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```

345 [pid 17480] mmap(0x7fc30ecdf000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FILE
  XED|MAP_DENYWRITE, 4, 0x1dc000) = 0x7fc30ecdf000
346 [pid 17480] mmap(0x7fc30ece5000, 33688, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FILE
  XED|MAP_ANONYMOUS, -1, 0) = 0x7fc30ece5000
347 [pid 17480] close(4) = 0
348 [pid 17480] mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
  1, 0) = 0x7fc30eaff000
349 [pid 17480] arch_prctl(ARCH_SET_FS, 0x7fc30eaffac0) = 0
350 [pid 17480] mprotect(0x7fc30ecdf000, 12288, PROT_READ) = 0
351 [pid 17480] mprotect(0x7fc30edb9000, 4096, PROT_READ) = 0
352 [pid 17480] mprotect(0x7fc30edd7000, 4096, PROT_READ) = 0
353 [pid 17480] mprotect(0x7fc30edde000, 4096, PROT_READ) = 0
354 [pid 17480] mprotect(0x7fc30ee61000, 4096, PROT_READ) = 0
355 [pid 17480] mprotect(0x7fc30f108000, 8192, PROT_READ) = 0
356 [pid 17480] mprotect(0x7fc30f12e000, 4096, PROT_READ) = 0
357 [pid 17480] mprotect(0x7fc30f311000, 4096, PROT_READ) = 0
358 [pid 17480] mprotect(0x1c75000, 8192, PROT_READ) = 0
359 [pid 17480] mprotect(0x7fc30f359000, 8192, PROT_READ) = 0
360 [pid 17480] munmap(0x7fc30f316000, 69006) = 0
361 [pid 17480] brk(NULL) = 0x3313000
362 [pid 17480] brk(0x3334000) = 0x3334000
363 [pid 17480] prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=65536*1024, rlim_max=RLIM
  64_INFINITY}) = 0
364 [pid 17480] openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC
  C) = 4

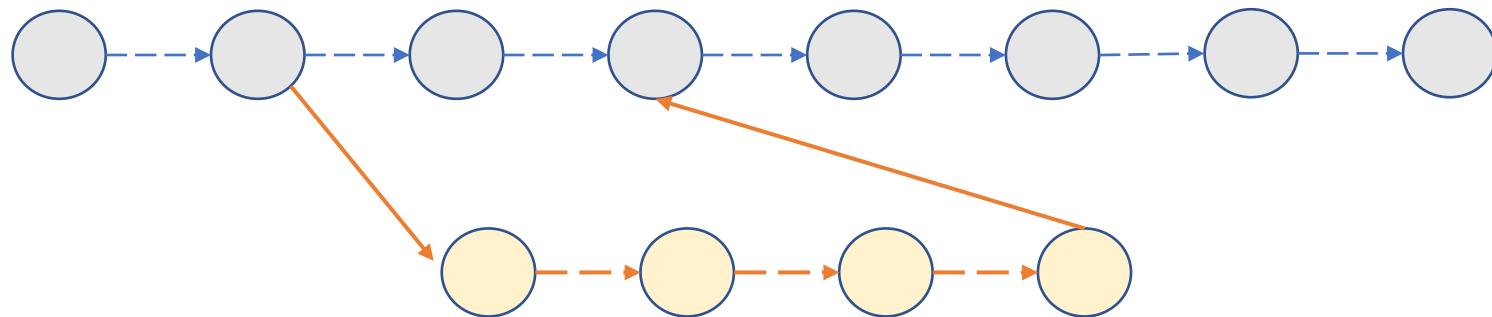
```

[No Name] [+1]

[strace] unix utf-8 Ln 345, Col 1/2923



# .c到.s



执行cc1  
.c → .s

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```

248 access("/usr/lib/gcc/x86_64-linux-gnu/10/", X_OK) = 0
249 newfstatat(AT_FDCWD, "/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper", {st_mode=S_IFREG|0755, st_size=655432, ...}, 0) = 0
250 access("/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper", X_OK) = 0
251 access("/tmp", R_OK|W_OK|X_OK) = 0
252 openat(AT_FDCWD, "/tmp/ccFnrVKs.s", O_RDWR|O_CREAT|O_EXCL, 0600) = 4
253 close(4) = 0
254 newfstatat(AT_FDCWD, "/usr/lib/gcc/x86_64-linux-gnu/10/cc1", {st_mode=S_IFREG|0755, st_size=25697696, ...}, 0) = 0
255 access("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", X_OK) = 0
256 pipe2([4, 5], O_CLOEXEC) = 0
257 vfork(<unfinished ...>
258 [pid 17480] close(4) = 0
259 [pid 17480] execve("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", ["/usr/lib/gcc/x86_64-linux-gnu/10"..., "-quiet", "-imultiarch", "x86_64-linux-gnu", "a.c", "-quiet", "-dumpbase", "a.c", "-mtune=generic", "-march=x86-64", "-auxbase", "a", "-fasynchronous-unwind-tables", "-fstack-protector-strong", "-Wformat", "-Wformat-security", "-fstack-clash-protection", "-fcf-protection", "-o", "/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */ <unfinished ...>
260 [pid 17479] <... vfork resumed> = 17480
261 [pid 17479] close(5) = 0
262 [pid 17479] read(4, "", 16) = 0
263 [pid 17480] <... execve resumed> = 0
264 [pid 17479] close(4) = 0
265 [pid 17480] brk(NULL <unfinished ...>

```

[No Name] [+1] [strace] unix utf-8 Ln 264, Col 1/2923



Right Shift + Right Alt

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```
1 execve("/usr/lib/ccache/gcc", ["gcc", "a.c"], 0x7ffd5a558738 /* 60 vars */) = 0
2 execve("/usr/bin/gcc", ["usr/bin/gcc", "a.c"], 0x56492d0e5320 /* 61 vars */) = 0
3 [pid 17480] execve("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", ["usr/lib/gcc/x86_64-
linux-gnu/10"..., "-quiet", "-imultiarch", "x86_64-linux-gnu", "a.c", "-quiet",
-dumpbase", "a.c", "-mtune=generic", "-march=x86-64", "-auxbase", "a", "-fasynchr
ond$ echo $PATH
, /usr/lib/ccache:/home/why/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin
000:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
4 [pid 17480] <... execve resumed> = 0
5 [pid 17481] execve("/usr/lib/ccache/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or directo
ry)
6 [pid 17481] execve("/home/why/.local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT
.o", "/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or di
rectory)
7 [pid 17481] execve("/usr/local/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or directo
ry)
8 [pid 17481] execve("/usr/local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or director
y)
9 [pid 17481] execve("/usr/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/
ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or directory)
10 [pid 17481] execve("/usr/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/c
cFnrVKs.s"], 0x18ec000 /* 66 vars */ <unfinished ...>
[No Name][+1] [strace] unix utf-8 Ln 1, Col 1/15
```



Right Shift + Right Alt

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```

1 execve("/usr/lib/ccache/gcc", ["gcc", "a.c"], 0x7ffd5a558738 /* 60 vars */) = 0
2 execve("/usr/bin/gcc", ["usr/bin/gcc", "a.c"], 0x56492d0e5320 /* 61 vars */) = 0
3 [pid 17480] execve("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", ["usr/lib/gcc/x86_64-
linux-gnu/10"..., "-quiet", "-imultiarch", "x86_64-linux-gnu", "a.c", "-quiet",
-dumpbase", "a.c", "-mtune=generic", "-march=x86-64", "-auxbase", "a", "-fasynchr-
onous-unwind-tables", "-fstack-protector-strong", "-Wformat", "-Wformat-security",
-fstack-clash-protection", "-fcf-protection", "-o", "/tmp/ccFnrVKs.s"], 0x18ec
000 /* 66 vars */ <unfinished ...>
4 [pid 17480] <... execve resumed> = 0
5 [pid 17481] execve("/usr/lib/ccache/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or direc-
tory)
6 [pid 17481] execve("/home/why/.local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.
o", "/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or di-
rectory)
7 [pid 17481] execve("/usr/local/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or direc-
tory)
8 [pid 17481] execve("/usr/local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
"/tmp/ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or director-
y)
9 [pid 17481] execve("/usr/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/
ccFnrVKs.s"], 0x18ec000 /* 66 vars */) = -1 ENOENT (No such file or directory)
10 [pid 17481] execve("/usr/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/c-
cFnrVKs.s"], 0x18ec000 /* 66 vars */ <unfinished ...>

```

[No Name] [+1]

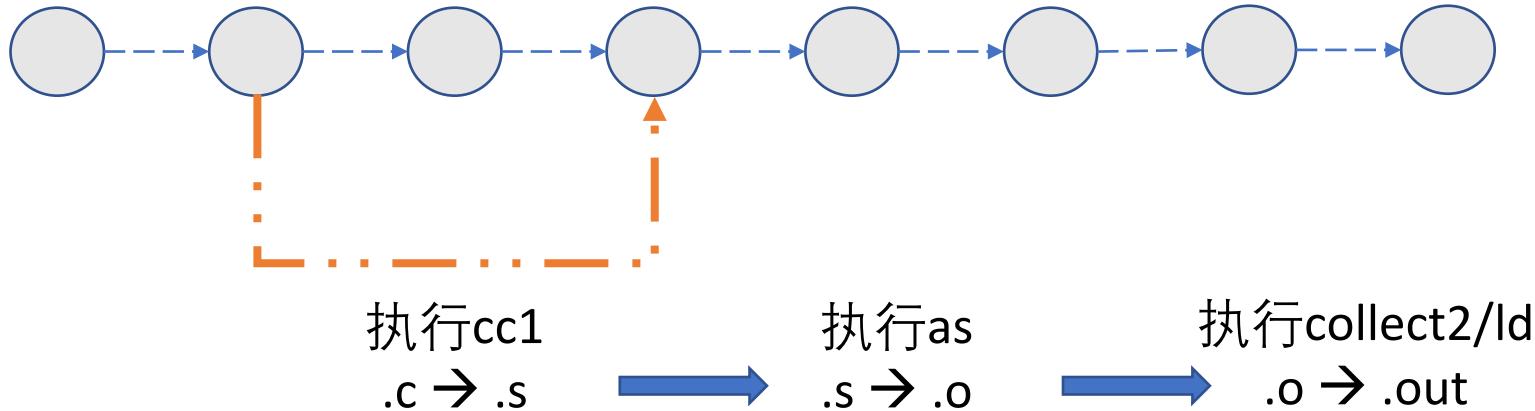
[strace] unix utf-8 Ln 1, Col 1/15



Right Shift + Right Alt

# 回到最初的问题

- 在IDE里，为什么按一个键，就能够编译运行？
  - 编译、链接
    - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
  - 加载执行
    - ./a.out
- 现在，一位同学对这个过程提出了质疑
  - 如何说服这位同学？



# 答案：用工具！

---

- 观察 trace: 查看 gcc 是否调用了其他命令
  - strace -f -qq gcc a.c 2>&1 | vim -
    - 可以使用 grep (shell) 或 :g! (vim) 筛选感兴趣的系统调用
- 调试 gcc: 查看每一个阶段的中间结果
  - 在哪里打断点?
    - [gdb-internals.txt](#)

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

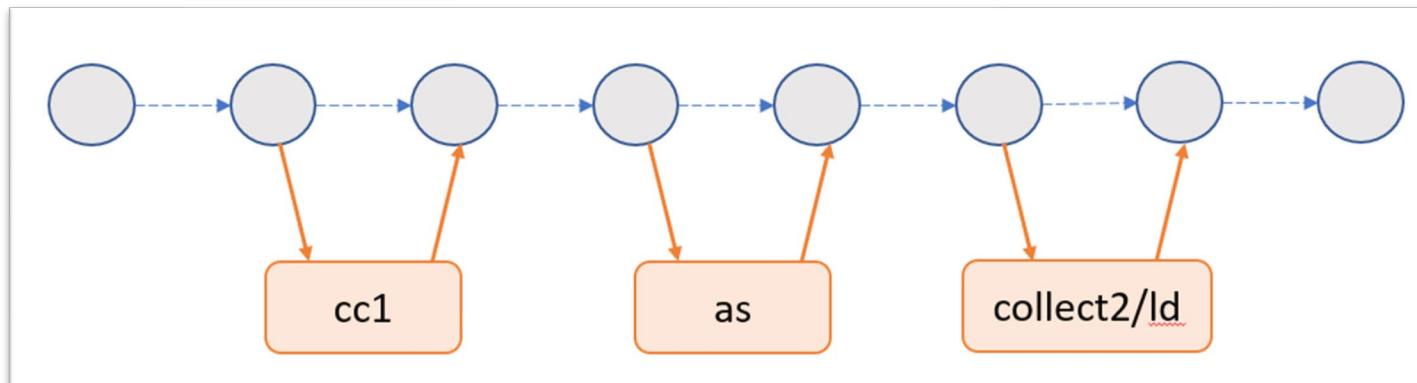
why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

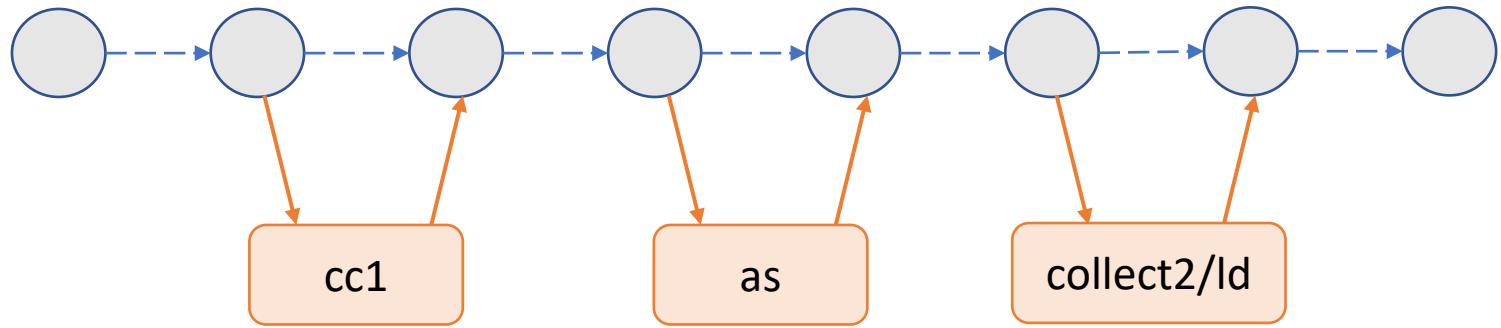
```
$ ls  
a.c  a.out  gdb-internals.txt  
$ █
```



# 回到第一次C语言课

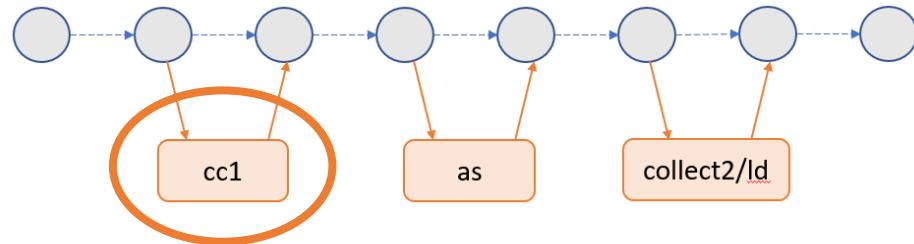
- 在IDE里，为什么按一个键，就能够编译运行？
  - 编译、链接
    - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
  - 加载执行
    - ./a.out
- 现在，一位同学对这个过程提出了质疑
  - 我不信！我就觉得是 gcc 一个程序直接搞定的
    - 道理上完全可以这么实现
  - 如何说服这位同学？



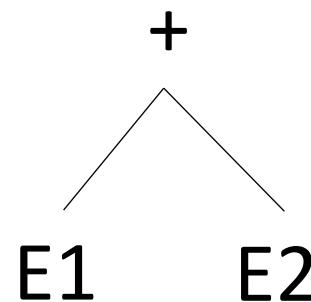


课堂上见过的轮子们

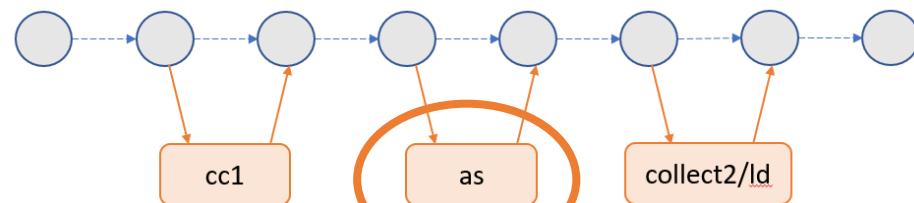
# 编译器 (.c → .s)



- 同 “表达式求值问题” (PA1)
  - 编译器 (OJ 题): 输入一个字符串, 输出计算它的汇编代码
    - 表达式  $(a + b) * (c + d)$
    - → 指令序列 (stack machine)
      - push \$a; push \$b; add; push \$c; push \$d; add; mul
- 现代编译器
  - 预编译 → 词法分析 → 语法树 → IR 中间代码 → 多趟优化 (内联、传播/折叠、删除冗余) → 指令生成/寄存器分配



# 汇编器 (.s → .o)



- 除去预编译，汇编代码和指令几乎一一对应

- 根据指令集手册规定翻译
- 生成符合 ELF 规范的二进制目标文件
  - printf("\x7f\x45\x4c\x46");...

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  
00000020: 0000 0000 0000 0000 b81e 0000 0000 0000  
00000030: 0000 0000 4000 0000 0000 4000 0f00 0e00  
00000040: 5548 89e5 4883 ec10 4889 7df8 4889 75f0  
00000050: 488b 55f0 488b 45f8 4801 d048 8905 0000  
00000060: 0000 488b 0500 0000 00ba 0d00 0000 4889  
00000070: c648 8d3d 0000 0000 b800 0000 00e8 0000  
00000080: 0000 488b 0500 0000 0048 8b10 4889 1500  
00000090: 0000 008b 5008 8915 0000 0000 0fb6 400c
```

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

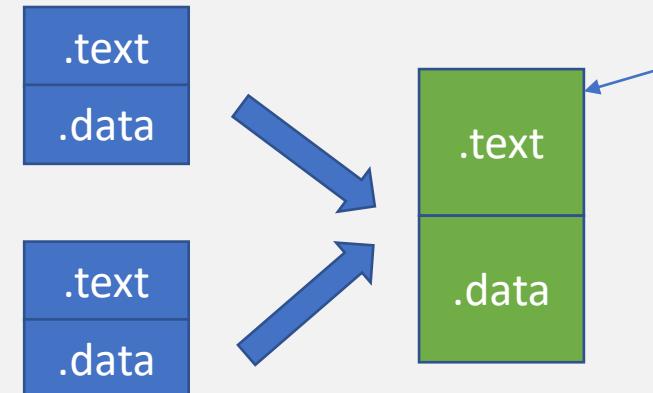
why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```
$ ls  
a.c  a.out  gdb-internals.txt  
$
```

# 连接器 (.o → a.out)

- 按照 ld script 指示完成二进制文件构造、符号解析和重定位
  - gcc a.c -Wl,--verbose
    - “.” 代表当前位置；基本功能：粘贴；定义符号

```
SECTIONS {  
    . = 0x100000;  
    .text : { *(entry) *(.text*) }  
    etext = .; _etext = .;  
    .rodata : { *(.rodata*) } .data : { *(.data*) }  
    edata = .; _edata = .;  
    .bss : { *(.bss*) }  
    _start_start = ALIGN(4096);  
    . = _start_start + 0x8000;  
    _stack_pointer = .;  
    end = .; _end = .;  
    _heap_start = ALIGN(4096);  
    _heap_end = 0x8000000;  
}
```



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

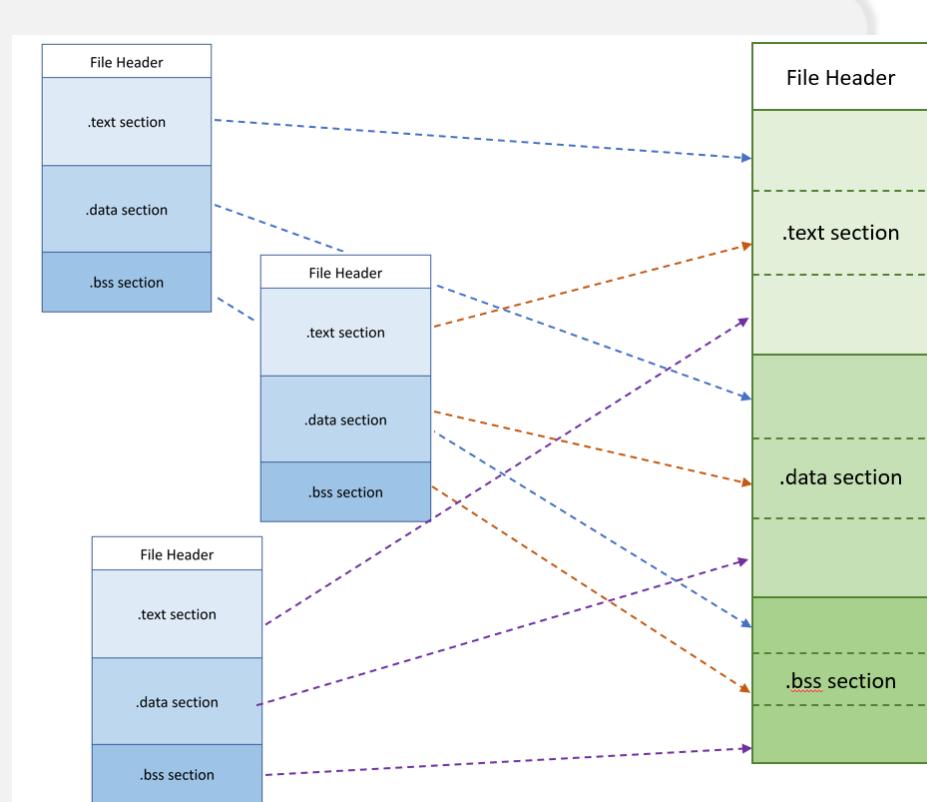
\$

I

# 链接器 (.o → a.out)

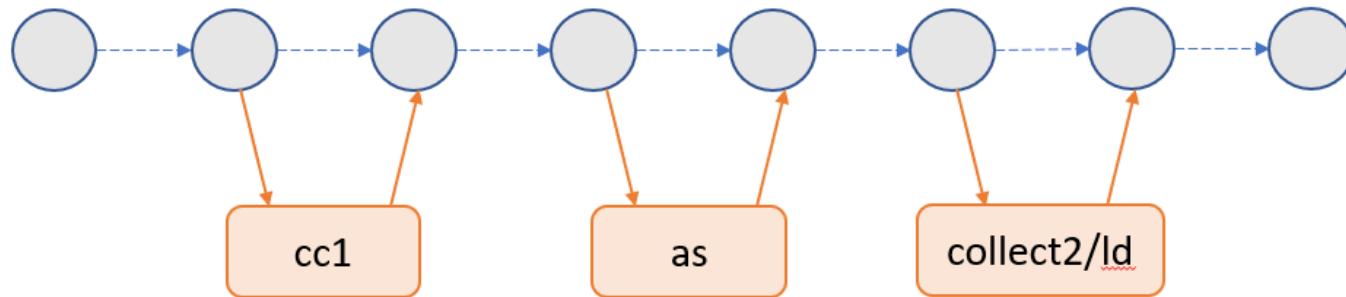
- 按照 ld script 指示完成二进制文件构造、符号解析和重定位
  - gcc a.c -Wl,--verbose
    - “.” 代表当前位置；基本功能：粘贴；定义符号

```
SECTIONS {  
    . = 0x100000;  
    .text : { *(entry) *(.text*) }  
    etext = .; _etext = .;  
    .rodata : { *(.rodata*) }  
    .data : { *(.data*) }  
    edata = .; _edata = .;  
    .bss : { *(.bss*) }  
    _start_start = ALIGN(4096);  
    . = _start_start + 0x8000;  
    _stack_pointer = .;  
    end = .; _end = .;  
    _heap_start = ALIGN(4096);  
    _heap_end = 0x8000000;  
}
```



# 加载器

- PA里做过了
- 编译、链接、加载
  - 好像没那么难啊（都是表达式求值和翻译）



# 单元测试框架

- YEMU 中的神奇代码

```
#define TESTCASES(_)
  \_(1, 0b11100111, random_rm, ASSERT_EQ(newR[0], oldM[7]) )
  \_(2, 0b00000100, random_rm, ASSERT_EQ(newR[1], oldR[0]) )
  \_(3, 0b11100101, random_rm, ASSERT_EQ(newR[0], oldM[5]) )
  \_(4, 0b00010001, random_rm, ASSERT_EQ(newR[0], oldR[0] + oldR[1]) )
  \_(5, 0b11110111, random_rm, ASSERT_EQ(newM[7], oldR[0]) )
```

- 但是我相信大家都没有做好单元测试

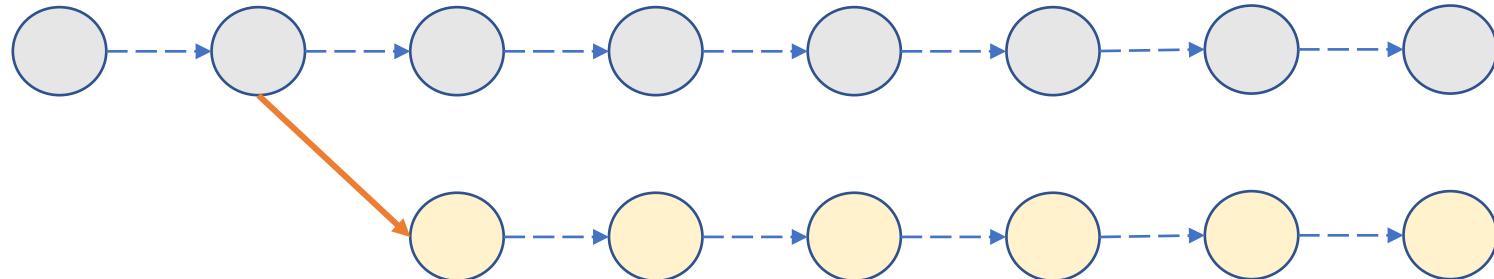
# Differential Testing

- 设置好状态；各走一步

```
for i in range(int(sys.argv[1])):
    print('\n'.join(['si'] + [f'p ${r}' for r in ['eax', ...]]))
```

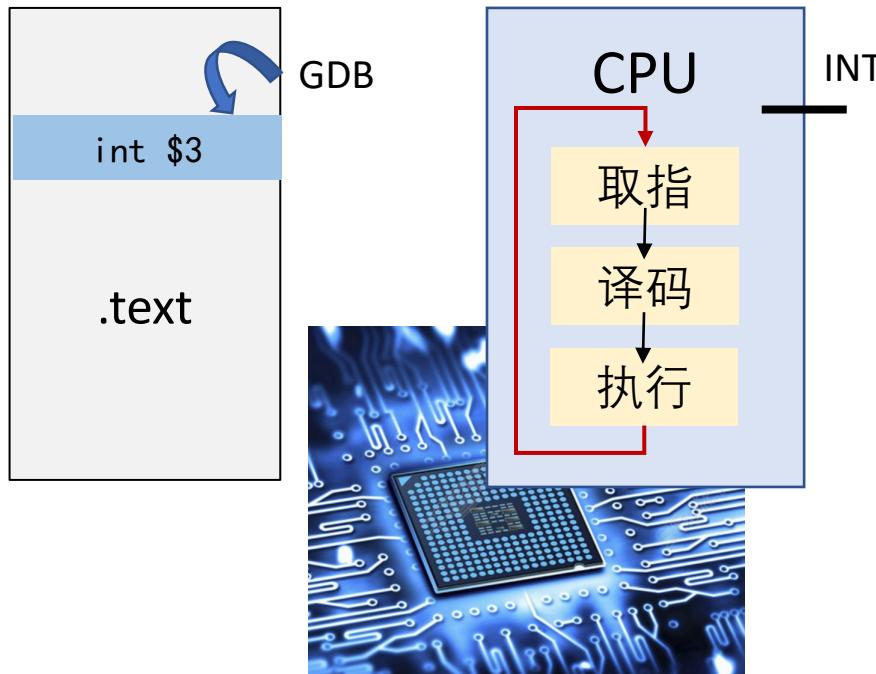
N=10000

```
diff <(python3 cmdgen.py $N | ./x86-nemu-datui img) \
<(python3 cmdgen.py $N | ./x86-nemu img)
```



# 调试器GDB

- 好奇它是怎么实现的？
  - 考虑核心功能：在任意 PC 的断点
  - 其他功能都可以基于断点实现
    - 单步调试：在下一条指令打断点
    - watch point：单步调试 + 检查条件



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

\$

MPROTECT(2)

Linux Programmer's Manual

MPROTECT(2)

**NAME**

`mprotect, pkey_mprotect - set protection on a region of memory`

**SYNOPSIS**

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

int mprotect(void *addr, size_t len, int prot);

#define _GNU_SOURCE           /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_mprotect(void *addr, size_t len, int prot, int pkey);
```

**DESCRIPTION**

`mprotect()` changes the access protections for the calling process's memory pages containing any part of the address range in the interval `[addr, addr+len-1]`. `addr` must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a `SIGSEGV` signal for the process.

`prot` is a combination of the following access flags: `PROT_NONE` or a bit-wise-or of the other values in the following list:

# 调试器GDB

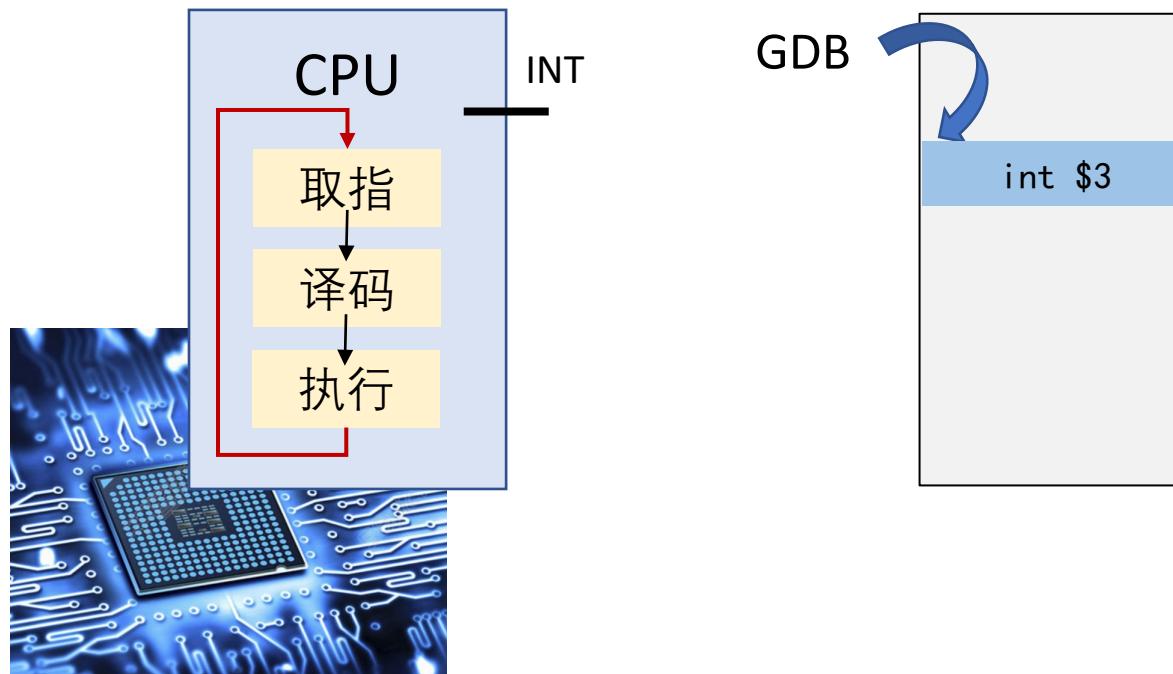
- 好奇它是怎么实现的?

- 考虑核心功能

- 其他功能都可

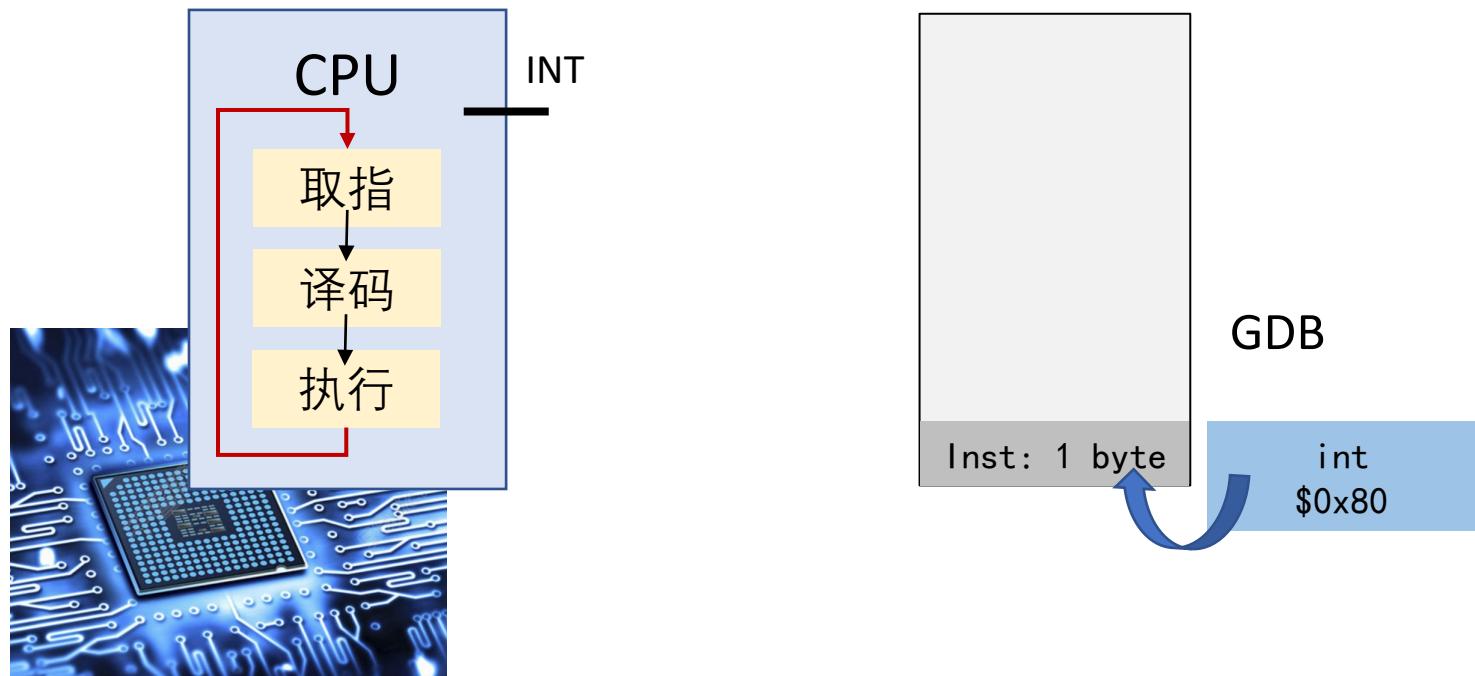
- 单步调试:
- watch point.

The `ptrace()` system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.



# 调试器GDB

- 好奇它是怎么实现的?
- 考虑核心功能: 在任意 PC 的断点
- 其他功能都可以基于断点实现
  - 单步调试: 在下一条指令打断点
  - watch point: 单步调试 + 检查条件



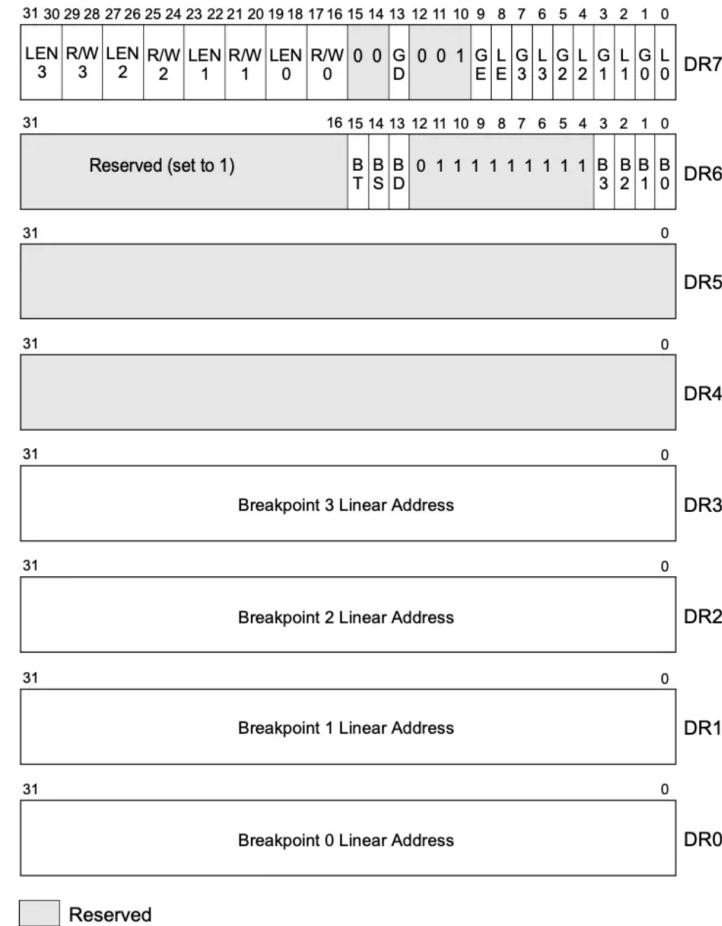
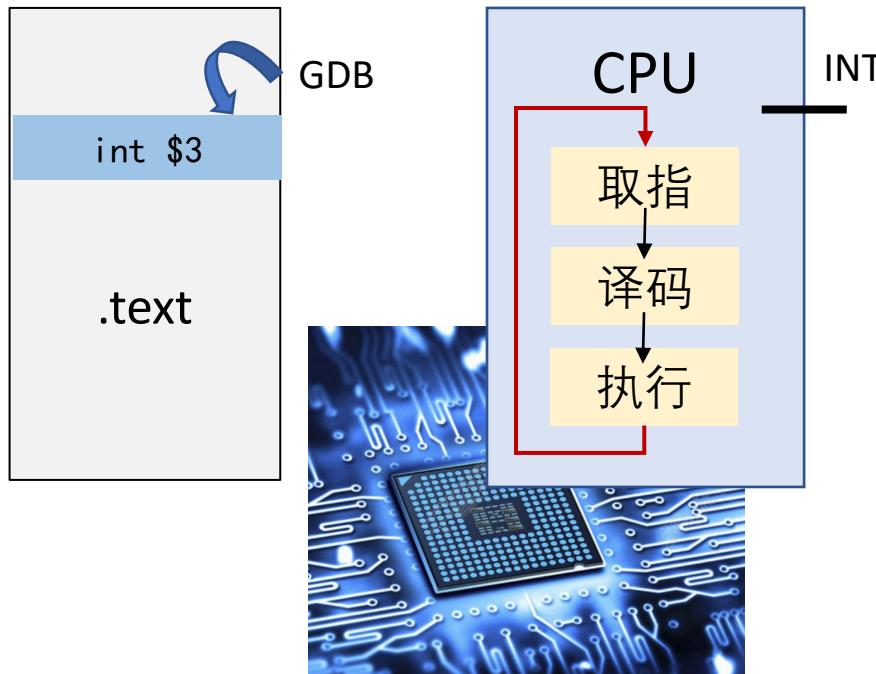
why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

\$ vim a.S

# 调试器GDB

- 好奇它是怎么实现的?
- 考虑核心功能: 在任意 PC 的断点
- 其他功能都可以基于断点实现
  - 单步调试: 在下一条指令打断点
  - watch point: 单步调试 + 检查条件



# Trace/Profiler

- strace: 刚才已经展示过威力了
    - 如何实现?

PTRACE(2)

## Linux Programmer's Manual

PTRACE(2)

**NAME****ptrace** - process trace**SYNOPSIS**

#include &lt;sys/ptrace.h&gt;

```
long ptrace(enum __ptrace_request request, pid_t pid,
           void *addr, void *data);
```

**DESCRIPTION**

The **ptrace()** system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

```
ptrace(PTRACE_foo, pid, ...)
```

Manual page **ptrace(2)** line 1 (press h for help or q to quit)

# 静态分析工具 Lint

- 大家见到的第一个 lint: `gcc -Wall -Werror`
  - `cppcheck`
  - [Cert C Coding Standard](#)
    - 自带 [checker](#)

## Introduction

由 Robert Seacord 创建, 最终由 Robert Schiela 修改于 一月 03, 2017

- Scope
- Audience
- How this Coding Standard is Organized
- History
- ISO/IEC TS 17961 C Secure Coding Rules
- Tool Selection and Validation
- Taint Analysis
- Rules versus Recommendations
- Conformance Testing
- Development Process
- Usage
- System Qualities
- Automatically Generated Code
- Government Regulations
- Acknowledgments

This standard provides rules for secure coding in the C programming language. The goal of these rules and recommendations is to develop safe, reliable, and secure systems, for example by eliminating undefined behaviors that can lead to undefined program behaviors and exploitable vulnerabilities. Conformance to the coding rules defined in this standard are necessary (but not sufficient) to ensure the safety, reliability, and security of software systems developed in the C programming language. It is also necessary, for example, to have a safe and secure design. Safety-critical systems typically have stricter requirements than are imposed by this coding standard, for example requiring that all memory be statically allocated. However, the application of this coding standard will result in high-quality systems that are reliable, robust, and resistant to attack.

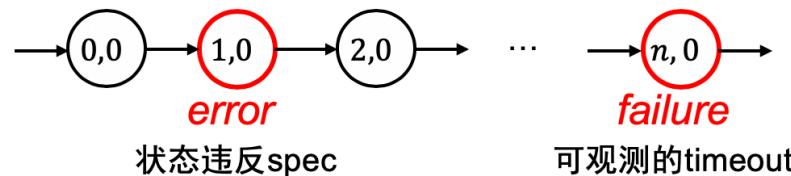
# 动态分析工具 Sanitizer

- 本质：运行时额外增加的 assert()

- 回顾调试理论

```
for (int i = 0; i < n; i++) fault 程序bug  
for (int j = 0; j < n; i++)
```

...



- 一些非常实用的 assertions

- `assert(IS_GUESTPTR(ptr));`
  - `(PMEM_MAP_START <= (x) && (x) < PMEM_MAP_END)`
- `assert(IS_SMALLINT(x));`
  - `(0 <= (x) && (x) <= 100)`
- `assert(IS_BOOL(ptr));`
  - `((x) == 0 || (x) == 1)`

# 总结

# 我们学到了什么？

“程序是个状态机。”

“我们可以观测状态机的设计、实现和执行。”

- 能实现几乎任何工具 (的玩具版)

- 并且能在需要的使用善用它们

- 编译器 (gcc)
    - 汇编器 (as)
    - 链接器 (ld)
    - 调试器 (gdb)
    - 追踪器 (strace/ltrace)
    - profiler (perf)

Cheers!

(你们完成了了不起的事情！ )

# 期末复习

王慧妍

why@nju.edu.cn

南京大学



计算机科学与技术系



计算机软件研究所



# 本讲概述

考试：占比不高，不必惊慌

- 本讲内容
  - 学期总结
  - 如何阅读汇编代码

# 学期总结

这学期讲了些什么？

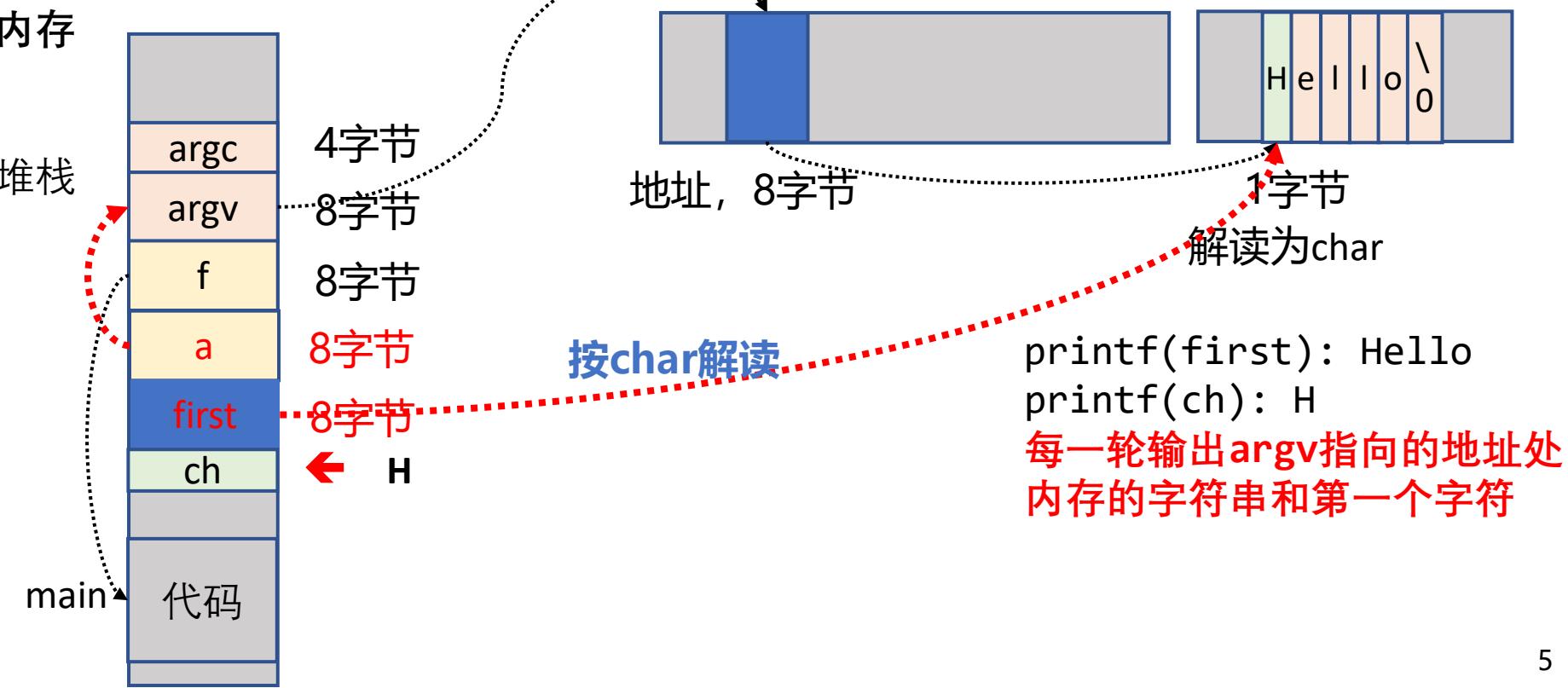
# Week2: C语言拾遗1之机制

- 在IDE里，为什么按一个键，就能够编译运行？
  - 编译、链接
    - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
  - 加载执行
    - ./a.out
- 背后是通过调用命令行工具完成的
  - RTFM: man gcc; gcc -help; tldr gcc
    - 控制行为的三个选项: -E, -S, -c
- 本次课程
  - 预热：编译、链接、加载到底做了什么？
  - RTFSC时需要关注的C语言特性

```

int main(int argc, char *argv[]) {
    int (*f)(int, char *[]) = main;
    if (argc != 0) {
        char ***a = &argv, *first = argv[0], ch = argv[0][0];
        printf("arg = \"%s\"; ch = '%c'\n", first, ch);
        assert(**a == ch);
        f(argc - 1, argv + 1);
    }
}

```



# Week3: C语言拾遗2之编写可读的代码

- IOCCC'11 best self documenting program

- 不可读 = 不可维护

```
#define clear 1;
    if(c>=11){c=0;sscanf(_, "%lf%c",&r,&c);while(*++_-=c);}\
    else if(argc>=4&&!main(4-*_-+=('),argv))_++;g:c+=
#define puts(d,e) return 0;}{double a;int b;char
    c=(argc<4?d)&15;\ b=(*%__LINE__+7)%9*(3*e>>c&1);c+=
#define I(d) (r);if(argc<4&&*#d==*_){a=r;r=usage?r*a:r+a;goto
    g;}c=c
#define return if(argc==2)printf("%f\n",r);return argc>=4+
#define usage main(4-__LINE__/26,argv)
#define calculator *_*(int)
#define l (r);r=--b?r:
#define _ argv[1]
```

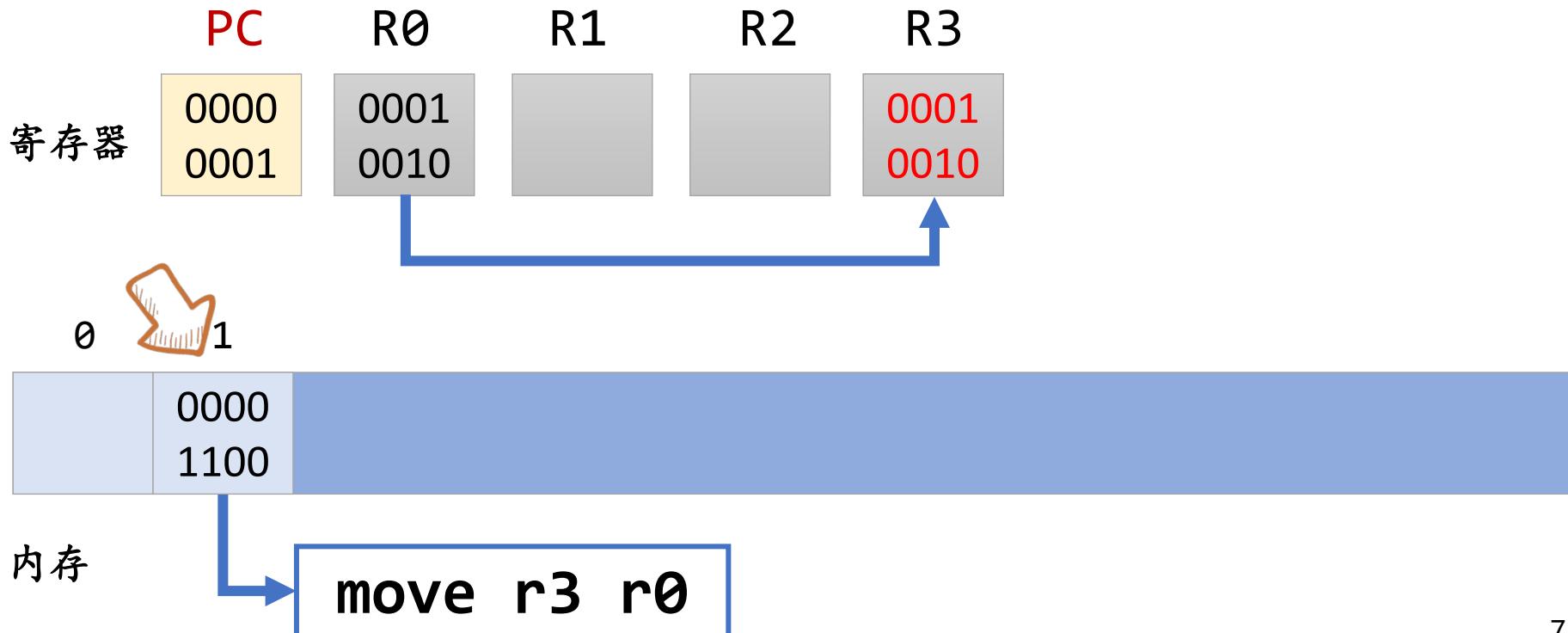
# 教科书第一章上的“计算机系统”

- 存储系统and指令集

寄存器: PC, R0 (RA), R1, R2, R3  
(8-bit)

内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	]	] [ rs ]
add	[0	0	0	1	[	rt	]	] [ rs ]
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	

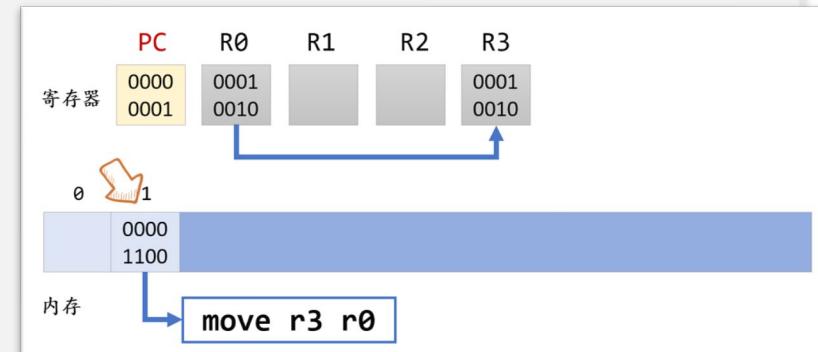
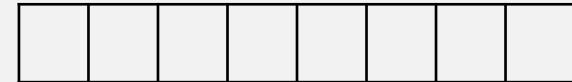


# 代码例子1

寄存器: PC, R0 (RA), R1, R2, R3 (8-bit)  
内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	][	rs
add	[0	0	0	1	[	rt	][	rs
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	

```
void idex() {  
    if ((M[pc] >> 4) == 0) {  
        R[(M[pc] >> 2) & 3] = R[M[pc] & 3];  
        pc++;  
    } else if ((M[pc] >> 4) == 1) {  
        R[(M[pc] >> 2) & 3] += R[M[pc] & 3];  
        pc++;  
    } else if ((M[pc] >> 4) == 14) {  
        R[0] = M[M[pc] & 0xf];  
        pc++;  
    } else if ((M[pc] >> 4) == 15) {  
        M[M[pc] & 0xf] = R[0];  
        pc++;  
    }  
}
```

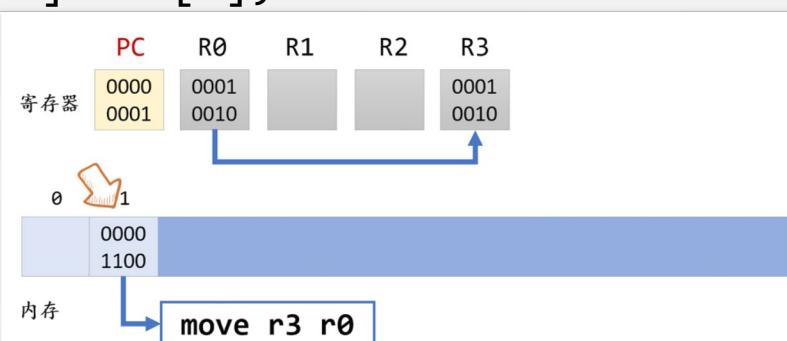


# 代码例子2

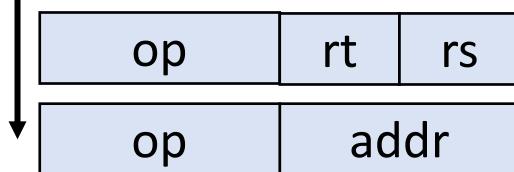
寄存器: PC, R0 (RA), R1, R2, R3 (8-bit)  
内存: 16字节 (按字节访问)

	7	6	5	4	3	2	1	0
mov	[0	0	0	0	[	rt	][	rs
add	[0	0	0	1	[	rt	][	rs
load	[1	1	1	0	[	addr	]	
store	[1	1	1	1	[	addr	]	

```
void index() {
    u8 inst = M[pc++];
    u8 op = inst >> 4;
    if (op == 0x0 || op == 0x1) {
        int rt = (inst >> 2) & 3, rs = (inst & 3);
        if (op == 0x0) R[rt] = R[rs];
        else if (op == 0x1) R[rt] += R[rs];
    }
    if (op == 0xe || op == 0xf) {
        int addr = inst & 0xf;
        if (op == 0xe) R[0] = M[addr];
        else if (op == 0xf) M[addr] = R[0];
    }
}
```



# 代码例子3 (YEMU代码)



```
typedef union inst {
    struct { u8 rs : 2, rt: 2,  op: 4; } rtype;
    struct { u8 addr: 4,           op: 4; } mtype;
} inst_t;
#define RTYPE(i) u8 rt = (i)->rtype.rt, rs = (i)->rtype.rs;
#define MTYPE(i) u8 addr = (i)->mtype.addr;

void idex() {
    inst_t *cur = (inst_t *)&M[pc];
    switch (cur->rtype.op) {
        case 0b0000: { RTYPE(cur); R[rt] = R[rs]; pc++; break; }
        case 0b0001: { RTYPE(cur); R[rt] += R[rs]; pc++; break; }
        case 0b1110: { MTYPE(cur); R[RA] = M[addr]; pc++; break; }
        case 0b1111: { MTYPE(cur); M[addr] = R[RA]; pc++; break; }
        default: panic("invalid instruction at PC = %x", pc);
    }
}
```

# 有用的C语言特性

- Union / bit fields

```
typedef union inst {
    struct { u8 rs : 2, rt: 2, op: 4; } rtype;
    struct { u8 addr: 4,          op: 4; } mtype;
} inst_t;
```

- 指针

- 内存只是个字节序列
- 无论何种类型的指针都只是地址 + 对指向内存的解读

```
inst_t *cur = (inst_t *)&M[pc];
// cur -> rtype.op
// cur -> mtype.addr
```

```
14 // decode
15 typedef struct {
16     union {
17         struct {
18             uint32_t opcode1_0 : 2;
19             uint32_t opcode6_2 : 5;
20             uint32_t rd      : 5;
21             uint32_t funct3  : 3;
22             uint32_t rs1     : 5;
23             int32_t  imm11_0 : 12;
24         } i;
25         struct {
26             uint32_t opcode1_0 : 2;
27             uint32_t opcode6_2 : 5;
28             uint32_t imm4_0   : 5;
29             uint32_t funct3  : 3;
30             uint32_t rs1     : 5;
31             uint32_t rs2     : 5;
32             int32_t  imm11_5 : 7;
33         } s;
34         struct {
35             uint32_t opcode1_0 : 2;
36             uint32_t opcode6_2 : 5;
37             uint32_t rd      : 5;
38             uint32_t imm31_12 : 20;
39         } u;
40         uint32_t val;
41     } instr;
42 } riscv32_ISADecodeInfo;
43
```

# Week4: 框架代码选讲1

GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside 50 million developers.

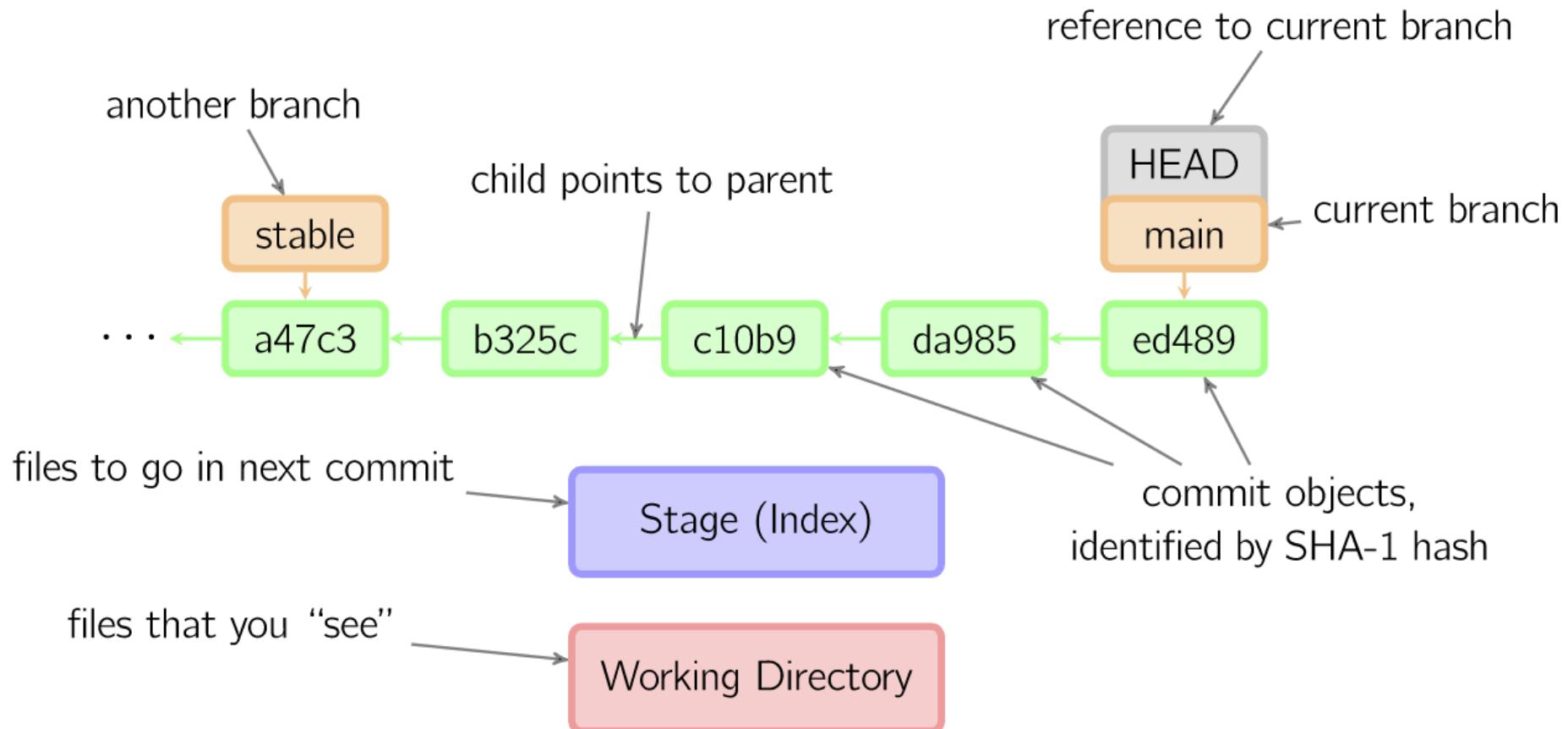
- **无所不能的代码聚集地**

- 有整个计算机系统世界的代码
  - 硬件、操作系统、分布式系统、库函数、应用程序……

- **学习各种技术的最佳平台**

- 海量的文档、学习资料、博客（新世界的大门）
  - 提供友好的搜索（例子：`awesome C`）

# A Visual Git Reference

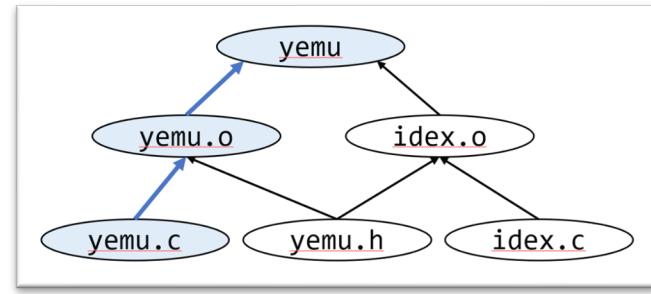
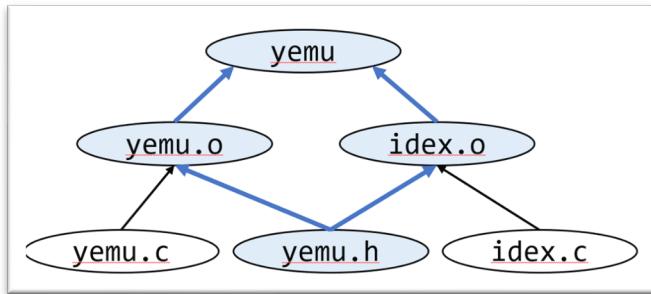


```
commit 8738b7b32e71a78f5b73376dc664780dd16800eb (HEAD -> pa1)
Author: tracer-ics2020 <tracer@njuics.org>
Date:   Thu Aug 19 18:27:15 2021 +0800
```

# Make工具

- 回顾：YEMU 模拟器
- Makefile 是一段 “declarative”的代码
  - 描述了构建目标之间的依赖关系和更新方法

```
$ make  
gcc -Wall -Werror -std=c11 -O2 -c -o yemu.o yemu.c  
gcc -Wall -Werror -std=c11 -O2 -c -o idex.o idex.c  
gcc -o yemu yemu.o idex.o  
$ make  
make: 'yemu' is up to date.
```



- make -j8

# NEMU代码构建

- Makefile 真复杂

- 放弃?

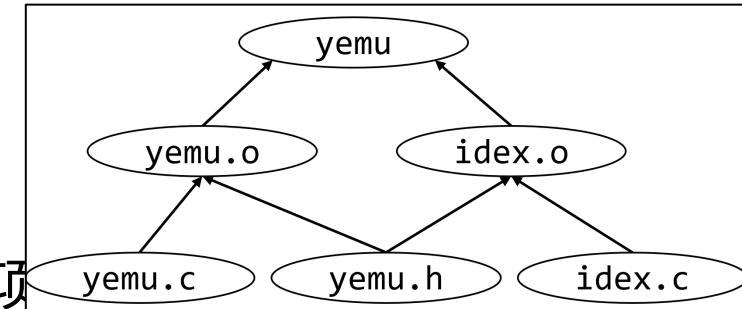
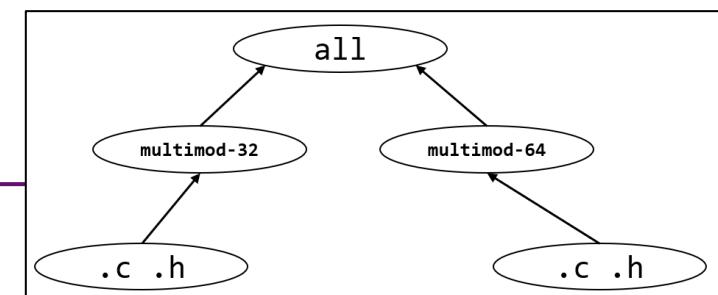
- 一个小诀窍

- 先观察 make 命令实际执行了什么 (trace)
  - RTFM/STFW: make 提供的两个有用的选择
    - -n 只打印命令不运行
    - -B 强制 make 所有目标

```
make -nB \
    | grep -ve '^(\#|echo|mkdir)' \
    | vim -
```

- 其实没有那么复杂

- 就是一堆gcc -c (编译) 和一个gcc (链接)
    - 大部分Makefile都是编译选项



# Week5: 框架代码选讲2

NEMU对大部分同学来说是一个“前所未有的大”的项目。

- 先大致了解一下

- 项目总体组织

- tree要翻好几个屏幕

- find . -name "\*.c" -o -name "\*.h" (100+个文件)

- 项目规模

- find ... | xargs cat | wc -l

```
$ find . -name "*.c" -o -name "*.h" | xargs cat | wc -l
```

```
23069
```

```
$ find tools -name "*.c" -o -name ".h" | xargs cat | wc -l
```

```
17728
```

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用**IDE (vscode: Edit→Find in files)**

```
$ ls  
abstract-machine  am-kernels  fceux-am  init.sh  Makefile  nemu  README.md  tags  
$
```

I

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用IDE (**vscode**: **Edit**→**Find in files**)
- The UNIX Way (无需启动任何程序，即可直接查看)

```
grep -n main $(find . -name "*.c") # RTFM: -n
```

# grep -n main \$(find . -name "\*.c")

```
$ grep -n main $(find . -name "*.c")
./tools/gen-expr/gen-expr.c:13:int main() { "
./tools/gen-expr/gen-expr.c:23:int main(int argc, char *argv[]) {
./tools/fixdep/fixdep.c:385:int main(int argc, char *argv[])
./tools/kconfig/mconf.c:1004:int main(int ac, char **av)
./tools/kconfig/symbol.c:1265:           /* for choice groups start the check with main
choice symbol */
./tools/kconfig/menu.c:331:                      /* set the type of the remaining choic
e values */
./tools/kconfig/build/lexer.lex.c:144:/* The state buf must be large enough to hold on
e state per character in the main buffer.
./tools/kconfig/build/lexer.lex.c:2684:/** The main scanner function which does all th
e work.
./tools/kconfig/build/lexer.lex.c:4119:/* Defined in main.c */
./tools/kconfig/build/parser.tab.c:541: "T_NOT", "$accept", "input", "mainmenu_stmt",
"stmt_list",
./tools/kconfig/conf.c:500:int main(int ac, char **av)
./resource/sdcard/nemu.c:91:     unsigned int          blocks;          /* remaining P
IO blocks */
./src/engine/interpreter/init.c:3:void sdb_mainloop();
./src/engine/interpreter/init.c:10:    sdb_mainloop();
./src/isa/riscv64/instr/decode.c:55:def_THelper(main) {
./src/isa/riscv64/instr/decode.c:65:    int idx = table_main(s);
./src/isa/riscv32/instr/decode.c:55:def_THelper(main) {
./src/isa/riscv32/instr/decode.c:65:    int idx = table_main(s);
./src/nemu-main.c:8:int main(int argc, char *argv[]) {
./src/monitor/sdb/sdb.c:84:void sdb_mainloop() {
./src/monitor/sdb/sdb.c:97:    /* treat the remaining string as the arguments,
```

# 尝试阅读代码：从main开始

- C语言代码，都是从main()开始运行的。那么哪里才有main呢？
  - 浏览代码：发现main.c，估计在里面
  - 使用IDE (**vscode**: **Edit**→**Find in files**)
- The UNIX Way (无需启动任何程序，即可直接查看)

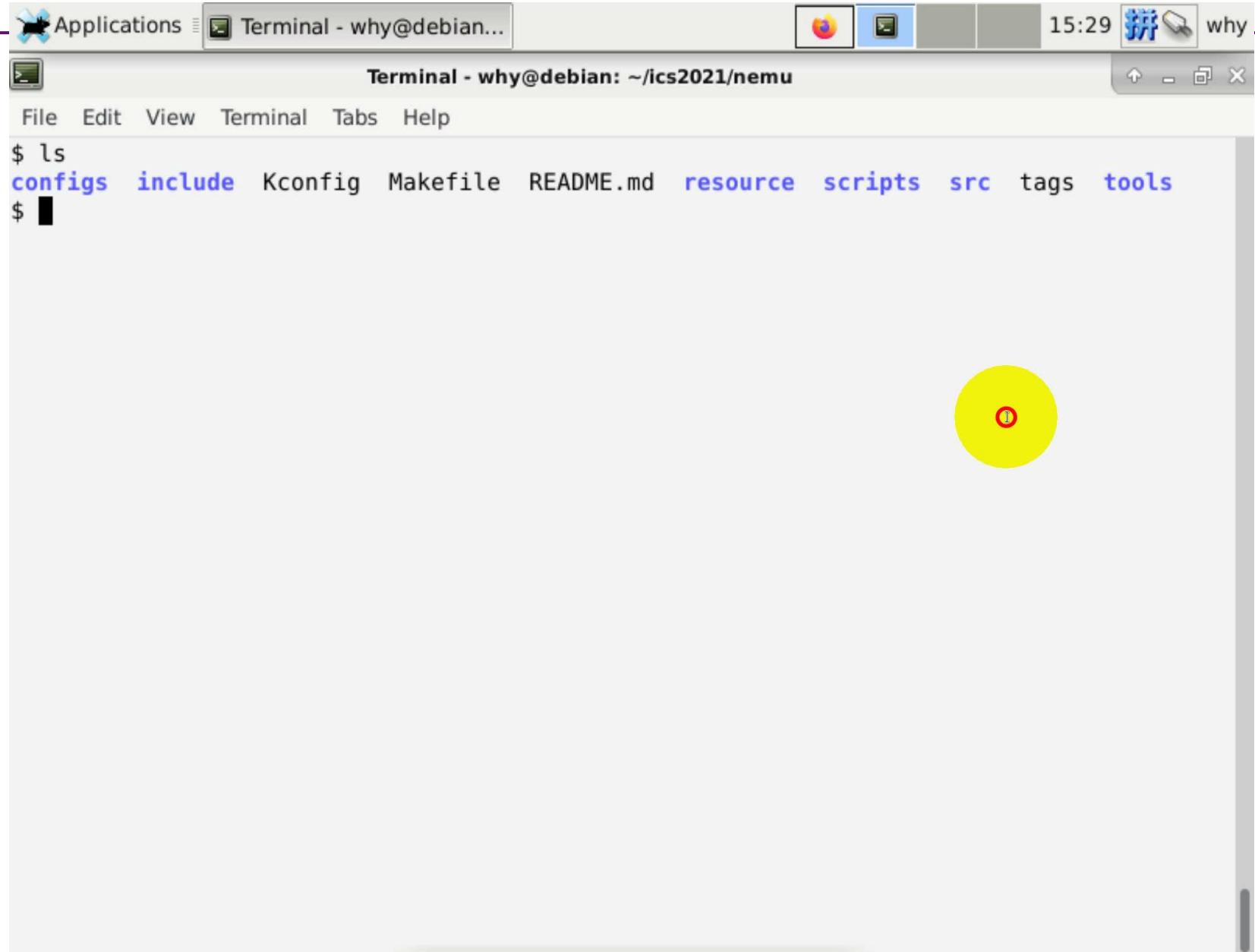
```
grep -n main $(find . -name "*.c") # RTFM: -n
```

```
find . | xargs grep --color -nse '\<main\>'
```

```
find . | xargs grep --color -nse '\<main\>'
```

```
$ find . | xargs grep --color -nse '\<main\>'  
./README.md:7:The main features of NEMU include  
.tools/gen-expr/gen-expr.c:13:int main() {  
.tools/gen-expr/gen-expr.c:23:int main(int argc, char *argv[]) {  
.tools/fixdep/fixdep.c:385:int main(int argc, char *argv[])  
Binary file ./tools/fixdep/build/fixdep matches  
Binary file ./tools/fixdep/build/obj-fixdep/fixdep.o matches  
.tools/kconfig/mconf.c:1004:int main(int ac, char **av)  
.tools/kconfig/symbol.c:1265:           /* for choice groups start the check with main  
choice symbol */  
Binary file ./tools/kconfig/build/conf matches  
Binary file ./tools/kconfig/build/obj-conf/conf.o matches  
.tools/kconfig/build/lexer.lex.c:144:/* The state buf must be large enough to hold on  
e state per character in the main buffer.  
.tools/kconfig/build/lexer.lex.c:2684:/* The main scanner function which does all th  
e work.  
.tools/kconfig/build/lexer.lex.c:4119:/* Defined in main.c */  
Binary file ./tools/kconfig/build/obj-mconf/mconf.o matches  
Binary file ./tools/kconfig/build/mconf matches  
.tools/kconfig/conf.c:500:int main(int ac, char **av)  
.tools/kconfig/expr.h:70:           S_DEF_USER,           /* main user value */  
.src/isa/riscv64/instr/decode.c:55:def_THelper(main) {  
.src/isa/riscv32/instr/decode.c:55:def_THelper(main) {  
.src/filelist.mk:1:SRCS-y += src/nemu-main.c  
.src/nemu-main.c:8:int main(int argc, char *argv[]) {  
Binary file ./build/riscv32-nemu-interpreter matches  
.build/obj-riscv32-nemu-interpreter/src/nemu-main.d:1:cmd_/home/why/ics2021/nemu/buil  
d/obj-riscv32-nemu-interpreter/src/nemu-main.o := unused
```

# fzf



A screenshot of a terminal window titled "Terminal - why@debian: ~/ics2021/nemu". The window shows the command \$ ls being run, followed by a list of files: configs, include, Kconfig, Makefile, README.md, resource, scripts, src, tags, and tools. The word "resource" is highlighted in blue, indicating it was selected by fzf. A yellow circle with a red "I" is overlaid on the right side of the terminal window.

```
$ ls
configs  include  Kconfig  Makefile  README.md  resource  scripts  src  tags  tools
$ █
```

# Vim: 这都搞不定还引发什么编辑器圣战

---

- Marks (文件内标记)
  - ma, 'a, mA, 'A, ...
- Tags (在位置之间跳转)
  - :jumps, C-], C-i, C-o, :tjump, ...
- Tabs/Windows (管理多文件)
  - :tabnew, gt, gT, ...
- Folding (浏览大代码)
  - zc, zo, zR, ...
- 更多的功能/插件: (RTFM, STFW)
  - vimtutor

## ESC normal mode

# vi / vim graphical cheat sheet

~ toggle case	! external filter	@ play macro	# prev ident	\$ eol	% O match	^ "soft" bol	& repeat :s	* next ident	( begin sentence	) end sentence	"soft" bol down	+ next line
\. goto mark	1 <sup>2</sup>	2	3	4	5	6	7	8	9	0 "hard" bol	- prev line	= auto <sup>3</sup> format
Q ex mode	W next WORD	E end WORD	R replace mode	T back 'till	Y yank line	U undo line	I insert at bol	O open above	P paste before	{ begin parag.	} end parag.	
Q record macro	W next word	E end word	R replace char	t 'till	y yank <sup>1,3</sup>	U undo	i insert mode	O open below	P paste <sup>4</sup> after	. misc	. misc	
A append at eol	S subst line	D delete to eol	F "back" find ch	G eof/ goto ln	H screen top	J join lines	K help	L screen bottom	. ex cmd line	!" reg. <sup>1</sup> spec	bol/ goto col	
a append	S subst char	d delete <sup>1,3</sup>	f find char	g extra <sup>6</sup> cmds	h ←	j ↓	k ↑	l →	; repeat t/T/f/F	'. goto mk. bol	\ not used!	
Z quit <sup>4</sup>	X back-space	C change to eol	V visual lines	B prev WORD	N prev (find)	M screen mid'l	< un- <sup>3</sup> indent	> indent <sup>3</sup>	? find (rev.)			
Z extra <sup>5</sup> cmd	X delete char	C change <sup>1,3</sup>	V visual mode	b prev word	n next (find)	m set mark	, reverse , t/T/f/F	. repeat cmd	/ find			

**motion** moves the cursor, or defines the range for an operator

**command** direct action command, if red, it enters insert mode

**operator** requires a motion afterwards, operates between cursor & destination

**extra** special functions, requires extra input

**q·** commands with a dot need a char argument afterwards

bol = beginning of line, eol = end of line, mk = mark, yank = copy

words: `:quux(foo, bar, baz);`  
WORDS: `:quux(foo, bar, baz);`

**Main command line commands ('ex'):**

:w (save), :q (quit), :q! (quit w/o saving)  
:e f (open file f),  
:%s/x/y/g (replace 'x' by 'y' filewide),  
:h (help in vim), :new (new file in vim),

**Other important commands:**

CTRL-R: redo (vim),  
CTRL-L/-B: page up/down,  
CTRL-E/-Y: scroll line up/down,  
CTRL-V: block-visual mode (vim only)

**Visual mode:**

Move around and type operator to act on selected region (vim only)

**Notes:**

(1) use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,\*)  
(e.g.: "ay\$ to copy rest of line to reg 'a')

(2) type in a number before any action to repeat it that number of times  
(e.g.: 2p, d2w, 5i, d4j)

(3) duplicate operator to act on current line  
(dd = delete line, >> = indent line)

(4) ZZ to save & quit, ZQ to quit w/o saving

(5) zt: scroll cursor to top,  
zb: bottom, zz: center

(6) gg: top of file (vim only),  
gf: open file under cursor (vim only)

For a graphical vi/vim tutorial & more tips, go to [www.viemu.com](http://www.viemu.com) - home of ViEmu, vi/vim emulation for Microsoft Visual Studio



File Edit View Terminal Tabs Help

why@debian:~/ics2021/nemu\$

why@debian:~/ics2021/nemu\$

---

why@debian:~/ics2021/nemu\$

I

# 如果你有块更大的屏幕

```
" Press ? for help
.. (up a dir)
</nemu/src/monitor/sdb/
expr.c
sdb.c
sdb.h
watchpoint.c

1 +... 6 lines: -----
7 #define CONFIG_DIFFTEST_REF_NAME "none"
8 #define CONFIG_ENGINE "interpreter"
9 #define CONFIG_PC_RESET_OFFSET 0x0
10 #define CONFIG_TARGET_NATIVE_ELF 1
11 #define CONFIG_MSIZE 0x8000000
12 #define CONFIG_CC_02 1
13 #define CONFIG_MODE_SYSTEM 1
14 #define CONFIG_MEM_RANDOM 1
15 #define CONFIG_ISA_riscv32 1
16 #define CONFIG_LOG_START 0
17 #define CONFIG_MBASE 0x80000000
18 #define CONFIG_TIMER_GETTIMEOFDAY 1
19 #define CONFIG_ENGINE_INTERPRETER 1
20 #define CONFIG_CC_OPT "-02"
21 #define CONFIG_LOG_END 10000
22 #define CONFIG_RT_CHECK 1
23 #define CONFIG_CC "gcc"
24 #define CONFIG_DIFFTEST_REF_PATH "none"
25 #define CONFIG_CC_DEBUG 1
26 #define CONFIG_CC_GCC 1
27 #define CONFIG_DEBUG 1
28 #define CONFIG_ISA "riscv32"

1 #include <common.h>
2
3 void init_monitor(int, char *[]);
4 void am_init_monitor();
5 void engine_start();
6 int is_exit_status_bad();
7
8 int main(int argc, char *argv[]) {
9     /* Initialize the monitor. */
10    #ifdef CONFIG_TARGET_AM
11        am_init_monitor();
12    #else
13        init_monitor(argc, argv);
14    #endif
15
16    /* Start engine. */
17    engine_start();
18
19    return is_exit_status_bad();
20 }

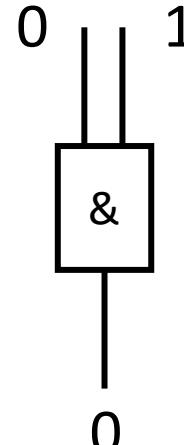
$ ls build include Makefile resource src
configs Kconfig README.md scripts tools
$ make
+ LD /home/why/Documents/ICS2021/ics2021/nemu/build/riscv
32-nemu-interpreter
$



<1/ics2021/nemu/src/monitor/sdb  <oconf.h[3] [cpp] unix utf-8 Ln 1, Col 1/28  <CS2021/ics2021/nemu/src/nemu-main.c[1] [c] unix utf-8 Ln 13, Col 5/20
-- INSERT --
[0] 0:vim*
```

# Week6: 数据的机器级表示

- 逻辑门和导线是构成计算机(组合逻辑电路)的基本单元
  - 位运算是用电路最容易实现的运算
    - & (与), | (或), ~ (非)
    - ^ (异或)
    - << (左移位), >> (右移位)
  - 例子：一代传奇处理器 8-bit [MOS 6502](#)
    - 3510 晶体管；56 条指令，算数指令仅有加减法和位运算



## Instructions by Name

[ADC](#) .... add with carry  
[AND](#) .... and (with accumulator)  
[ASL](#) .... arithmetic shift left  
[BCC](#) .... branch on carry clear  
[BCS](#) .... branch on carry set  
[BEQ](#) .... branch on equal (zero set)  
[BIT](#) .... bit test  
[BMI](#) .... branch on minus (negative set)  
[BNE](#) .... branch on not equal (zero clear)  
[BPL](#) .... branch on plus (negative clear)  
[BRK](#) .... break / interrupt

[BVC](#) .... branch on overflow clear  
[BVS](#) .... branch on overflow set  
[CLC](#) .... clear carry  
[CLD](#) .... clear decimal  
[CLI](#) .... clear interrupt disable  
[CLV](#) .... clear overflow  
[CMP](#) .... compare (with accumulator)  
[CPX](#) .... compare with X  
[CPY](#) .... compare with Y  
[DEC](#) .... decrement  
[DEX](#) .... decrement X  
[DEY](#) .... decrement Y

# Bit Set: 求 $|S|$ ( $S$ 二进制表示有多少个1)

```
int bitset_size(uint32_t S) {
    int n;
    for (int i = 0; i < 32; i++) {
        n += bitset_contains(S, i);
    }
    return n;
}
```

```
int bitset_size1(uint32_t S) { // SIMD
    S = (S & 0x55555555) + ((S >> 1) & 0x55555555);
    S = (S & 0x33333333) + ((S >> 2) & 0x33333333);
    S = (S & 0x0F0F0F0F) + ((S >> 4) & 0x0F0F0F0F);
    S = (S & 0x00FF00FF) + ((S >> 8) & 0x00FF00FF);
    S = (S & 0x0000FFFF) + ((S >> 16) & 0x0000FFFF);
    return S;
}
```

# Bit Set: 返回 $S \neq \emptyset$ 中的某个元素

- 有二进制数  $x = 0b+++++100$ , 我们希望得到最后那个100
  - 想法: 使用基本操作构造一些结果, 能把++++的部分给抵消掉

表达式	结果
$x$	$0b+++++100$
$x-1$	$0b+++++011$
$\sim x$	$0b-----011$
$\sim x+1$	$0b-----100$

- 一些有趣的式子:
  - $x \& (x-1) \rightarrow 0b+++++000$ ;  $x ^ (x-1) \rightarrow 0b00000111$ ;
  - $x \& (\sim x+1) \rightarrow 0b00000100$  (*lowbit*)
    - $x \& -x$ ,  $(\sim x \& (x-1))+1$  都可以实现*lowbit*
    - 只遍历存在的元素可以加速求 $|S|$

# Bit Set: 求 $\lfloor \log_2(x) \rfloor$

- 等同于 $31 - \text{clz}(x)$

```
int __builtin_clz(uint32_t x) {
    int n = 0;
    if (x <= 0x0000ffff) n += 16, x <<= 16;
    if (x <= 0x00ffffff) n += 8, x <<= 8;
    if (x <= 0xfffffff) n += 4, x <<= 4;
    if (x <= 0x3fffffff) n += 2, x <<= 2;
    if (x <= 0x7fffffff) n++;
    return n;
}
```

x: 0x00000001	16
x: 0x00010000	24
x: 0x01000000	28
x: 0x03000000	30

$S = \{0\}$       31

- (奇怪的代码) 假设 $x$ 是lowbit的结果?

```
#define LOG2(x) \
    ("`-01J2GK-3@HNL ; -=47A-IFO?M:<6-E>95D8CB"[(x) % 37] - '0')
```

# Undefined Behavior: 警惕整数溢出

表达式	值
<code>UINT_MAX+1</code>	0
<code>INT_MAX+1; LONG_MAX+1</code>	undefined
<code>char c = CHAR_MAX; c++;</code>	varies (???)
<code>1 &lt;&lt; -1</code>	undefined
<code>1 &lt;&lt; 0</code>	1
<code>1 &lt;&lt; 31</code>	undefined
<code>1 &lt;&lt; 32</code>	undefined
<code>1 / 0</code>	undefined
<code>INT_MAX % -1</code>	undefined

- W. Dietz, et al. Understanding integer overflow in C/C++.  
In *Proceedings of ICSE*, 2012.

# 整数溢出和编译优化

```
int f() { return 1 << -1; }
```

- 根据手册，这是个UB，于是clang这样处置

```
0000000000000000 <f>:  
0: c3      retq
```

- 编译器把这个计算直接删除了

W. Xi, et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of SOSP*, 2013.

# Bit Set: 求 $\lfloor \log_2(x) \rfloor$ (cont'd)

- 用一点点元编程 (meta-programming) ; 试一试[log2.c](#)

```
import json

n, base = 64, '0'
for m in range(n, 10000):
    if len({(2**i) % m for i in range(n)}) == n:
        M = {j: chr(ord(base) + i)
              for j in range(0, m)
              for i in range(0, n)
              if (2**i) % m == j}
        break

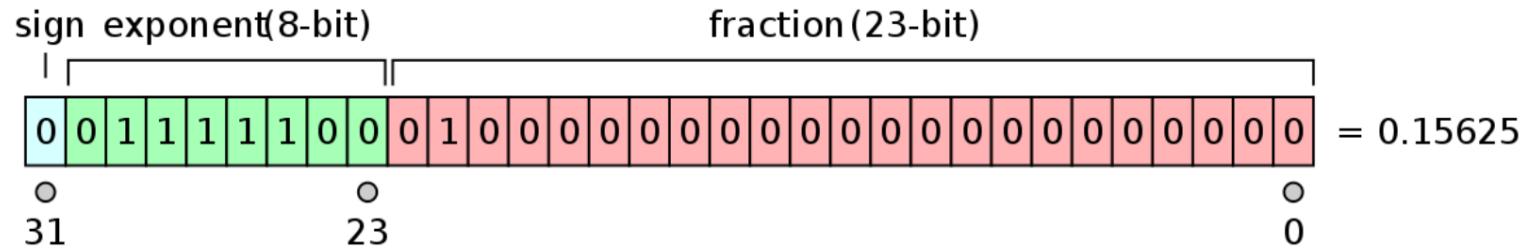
magic = json.dumps(''.join(
    [M.get(j, '-') for j in range(0, m)])
).strip('"')

print(f'#define LOG2(x) ("{magic}")[({x} % {m}) - \'{base}\']')
```

# 实数的计算机表示

- 实数非常非常多
  - 只能用32/64-bit 01串来表述一小部分实数
    - 确定一种映射方法，把一个01串映射到一个实数
      - 运算起来不太麻烦
      - 计算误差不太可怕
- 于是有了IEEE754
  - 1bit S, 23/53bits Fraction (尾数), 8/11bits Exponent (阶码),

$$x = (-1)^S \times (1.F) \times 2^{E-B}$$



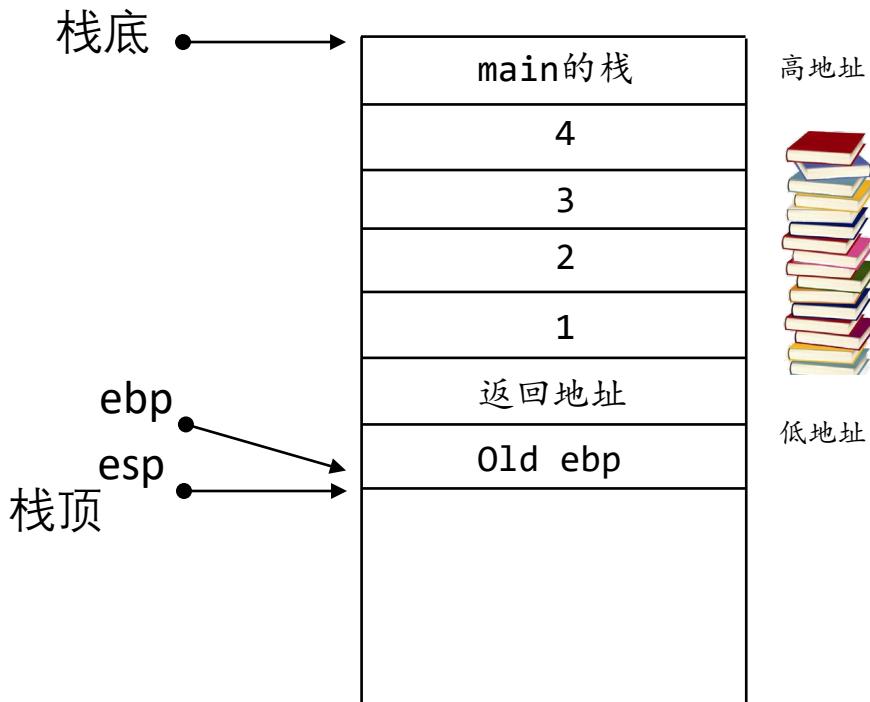
# IEEE754: 你有可能不知道的事实

- 一个有关浮点数大小/密度的实验([float.c](#))
- 越大的数字，距离下一个实数的距离就越大
  - 可能会带来相当的绝对误差
  - 因此很多数学库都会频繁做归一化
- 例子：计算 $1 + \frac{1}{2} + \frac{1}{3} \dots + \frac{1}{n}$

```
#define SUM(T, st, ed, d) ({ \
    T s = 0; \
    for (int i = st; i != ed + d; i += d) \
        s += (T)1 / i; \
    s; \
})
```

# Week7: x86-64与内联汇编

```
int __cdecl foo(int m1, int m2, int m3, int m4);  
int foo(int m1, int m2, int m3, int m4) {  
    return m1 + m2 + m3 + m4;  
}
```



cdecl函数调用惯例

```
int main() { foo(1,2,3,4); }
```

```
Disassembly of section .text:  
00000000 <foo>:  
 0: f3 0f 1e fb        endbr32  
 4: 55                 push %ebp  
 5: 89 e5               mov %esp,%ebp  
 7: e8 fc ff ff ff    call 8 <foo+0x8>  
 c: 05 01 00 00 00      add $0x1,%eax  
11: 8b 55 08            mov 0x8(%ebp),%edx  
14: 8b 45 0c            mov 0xc(%ebp),%eax  
17: 01 c2               add %eax,%edx  
19: 8b 45 10            mov 0x10(%ebp),%eax  
1c: 01 c2               add %eax,%edx  
1e: 8b 45 14            mov 0x14(%ebp),%eax  
21: 01 d0               add %edx,%eax  
23: 5d                 pop %ebp  
24: c3                 ret  
  
00000025 <main>:  
25: f3 0f 1e fb        endbr32  
29: 55                 push %ebp  
2a: 89 e5               mov %esp,%ebp  
2c: e8 fc ff ff ff    call 2d <main+0x8>  
31: 05 01 00 00 00      add $0x1,%eax  
36: 6a 04               push $0x4  
38: 6a 03               push $0x3  
3a: 6a 02               push $0x2  
3c: 6a 01               push $0x1  
3e: e8 fc ff ff ff    call 3f <main+0x1a>  
43: 83 c4 10            add $0x10,%esp  
46: b8 00 00 00 00      mov $0x0,%eax  
4b: c9                 leave  
4c: c3                 ret
```

```

void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}

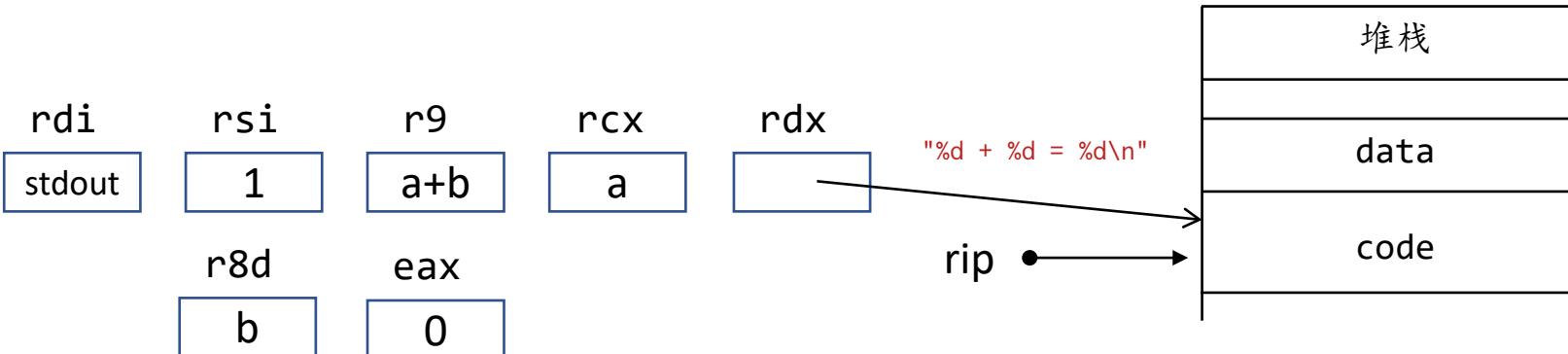
```

- 实际调用的是 `_fprintf_chk@plt`

- 需要传递的参数: `stdout, %d + %d = %d\n, a, b, a + b`
- calling convention: `rdi, rsi, rdx, rcx, r8, r9`

`0000000000000700 <plus>:`

<code>700: 44 8d 0c 37</code>	<code>lea (%rdi,%rsi,1),%r9d</code>
<code>704: 89 f9</code>	<code>mov %edi,%ecx</code>
<code>706: 48 8b 3d 03 09 20 00</code>	<code>mov 0x200903(%rip),%rdi # 201010 &lt;stdout@@GLIBC_2.2.5&gt;</code>
<code>70d: 48 8d 15 b0 00 00 00</code>	<code>lea 0xb0(%rip),%rdx # 7c4 &lt;_IO_stdin_used+0x4&gt;</code>
<code>714: 41 89 f0</code>	<code>mov %esi,%r8d</code>
<code>717: 31 c0</code>	<code>xor %eax,%eax</code>
<code>719: be 01 00 00 00</code>	<code>mov \$0x1,%esi</code> ←
<code>71e: e9 5d fe ff ff</code>	<code>jmpq 580 &lt;_fprintf_chk@plt&gt;</code>



# 在汇编中访问 C 世界的表达式

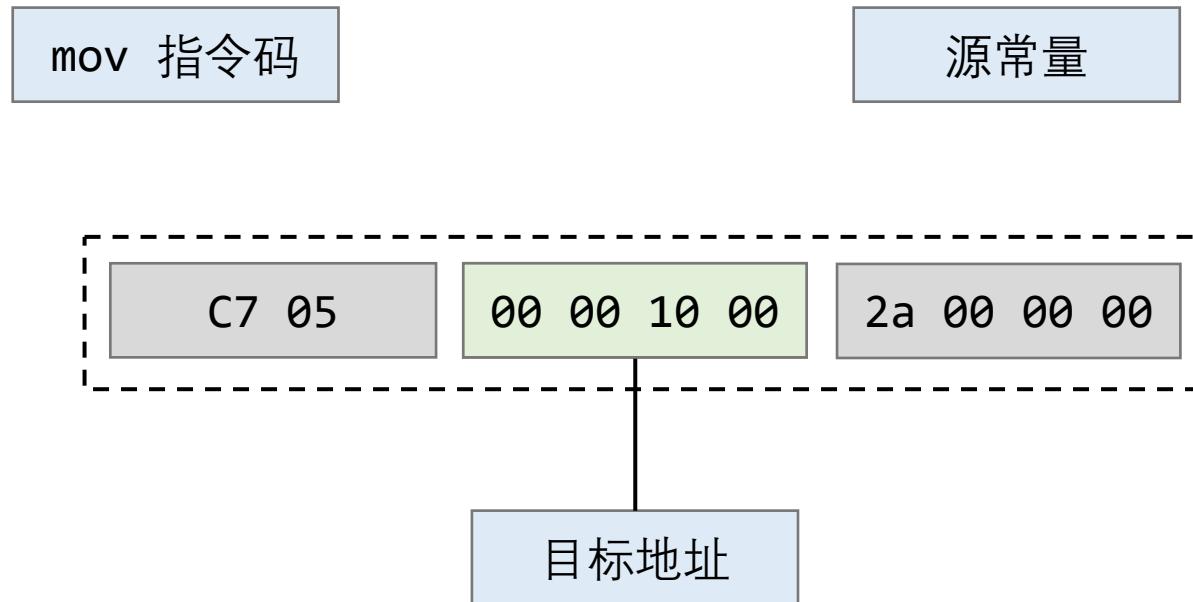
```
int foo(int x, int y) {  
    int z;  
    x++; y++;  
    asm (  
        "addl %1, %2; "  
        "movl %2, %0; "  
        : "=r"(z) // output  
        : "r"(x), "r"(y) // input  
    );  
    return z;  
}
```

gcc -O0

```
0000000000000000 <foo>:  
 0: f3 0f 1e fa      endbr64  
 4: 55                push   %rbp  
 5: 48 89 e5          mov    %rsp,%rbp  
 8: 89 7d ec          mov    %edi,-0x14(%rbp)  
 b: 89 75 e8          mov    %esi,-0x18(%rbp)  
 e: 83 45 ec 01       addl   $0x1,-0x14(%rbp)  
 12: 83 45 e8 01     addl   $0x1,-0x18(%rbp)  
 16: 8b 45 ec          mov    -0x14(%rbp),%eax  
 19: 8b 55 e8          mov    -0x18(%rbp),%edx  
 1c: 01 c2              add    %eax,%edx  
 1e: 89 d0              mov    %edx,%eax  
 20: 89 45 fc          mov    %eax,-0x4(%rbp)  
 23: 8b 45 fc          mov    -0x4(%rbp),%eax  
 26: 5d                pop    %rbp  
 27: c3                ret
```

# Week8: 链接与加载选讲

- A: 定义var (0x1000) → var = 42
- B: movl \$0x2a, var



# 可执行文件格式之ELF

```
int global_int_var = 84;
```

```
int global_int_var2;
```

```
void func1(int i){  
    printf("%d\n", i);  
}  
int main(void){
```

```
    static int static_var = 85;
```

```
    static int static_var2;
```

```
    int a = 1;  
    int b;  
    func1( static_var + static_var2  
    + a + b);  
    return 0;  
}
```

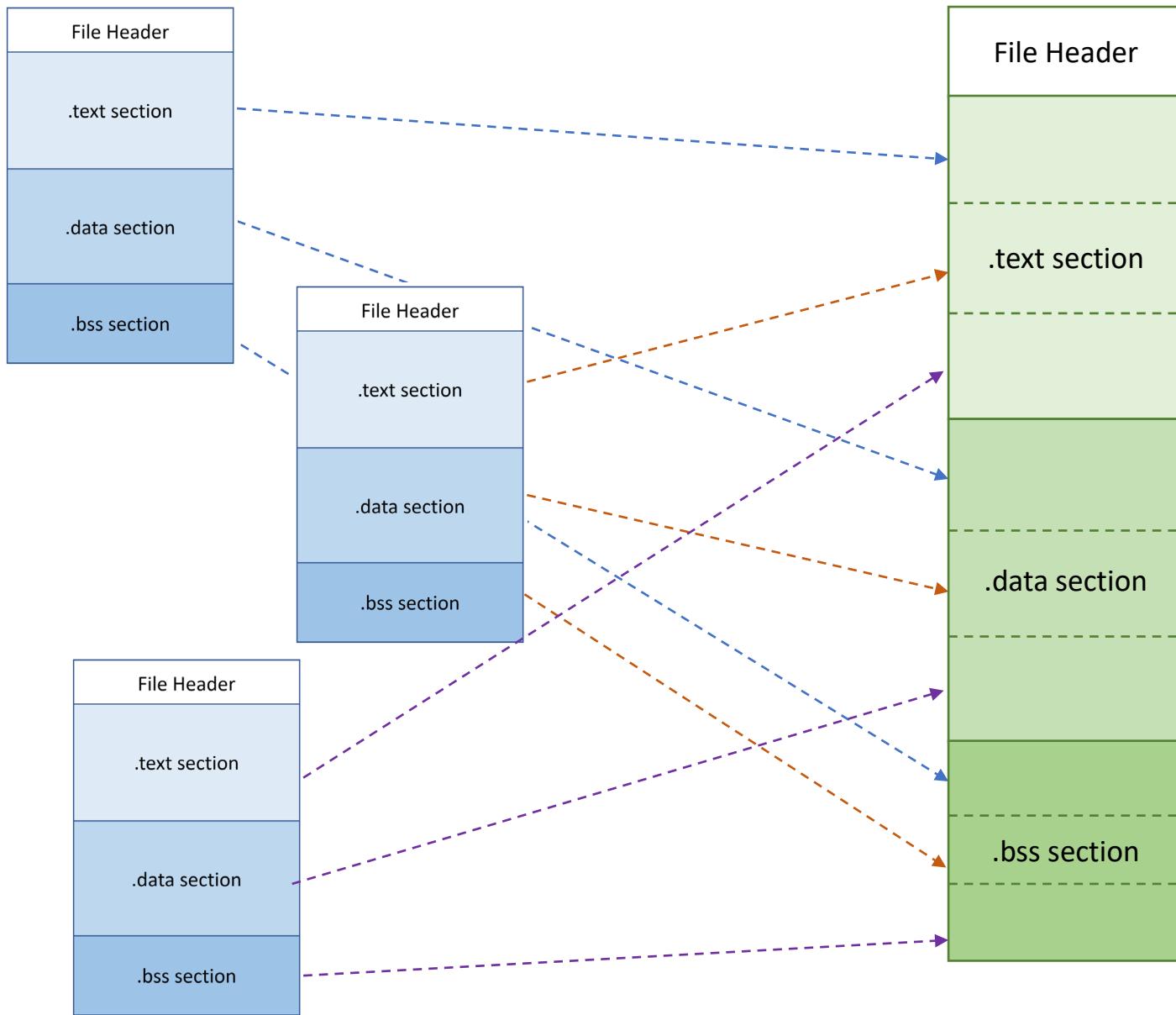
File Header

.text section

.data section

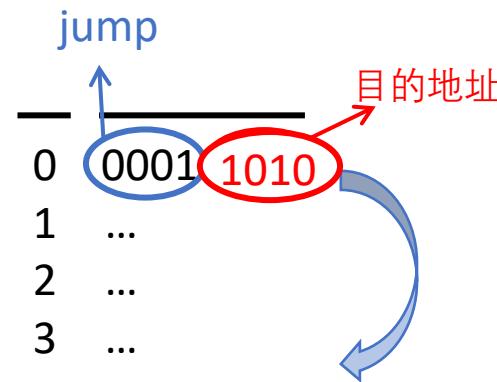
.bss section

# 链接多个.O



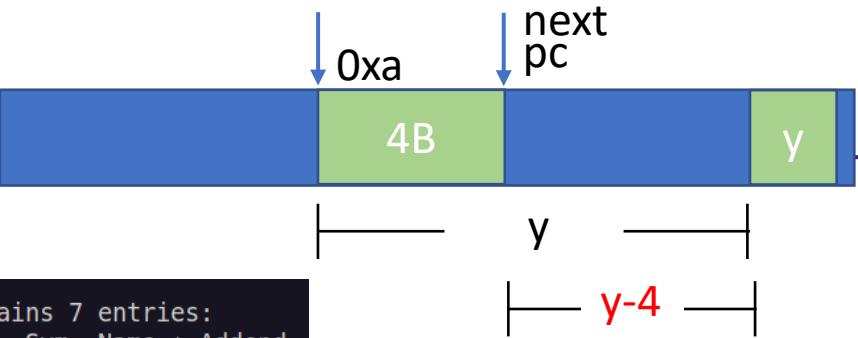
# 链接 Two-pass linking

- 空间和地址的分配
  - 重新建立符号表，合并段，并计算段长度建立映射关系
- 符号解析和重定位
  - 如何填空？



— —  
10 1000 0111  
11 ...

# 填什么？为什么 $y - 4$ ？



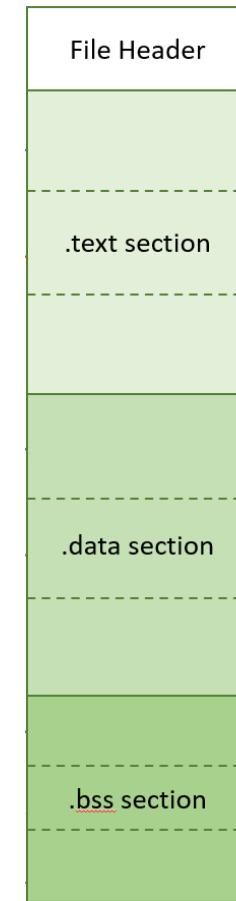
- `readelf -a main.o`

```
Relocation section '.rela.text.startup' at offset 0x208 contains 7 entries:
  Offset          Info      Type            Sym. Value   Sym. Name + Addend
000000000000000a  000500000002 R_X86_64_PC32    0000000000000000 y - 4
00000000000010  000600000002 R_X86_64_PC32    0000000000000000 x - 4
00000000000015  000700000004 R_X86_64_PLT32   0000000000000000 foo - 4
0000000000001b  000500000002 R_X86_64_PC32    0000000000000000 y - 4
00000000000021  000600000002 R_X86_64_PC32    0000000000000000 x - 4
00000000000026  00020000000a R_X86_64_32     0000000000000000 .rodata.str1.1 + 0
00000000000035  000800000004 R_X86_64_PLT32   0000000000000000 __printf_chk - 4
```

- `objdump -d main.o`

```
Disassembly of section .text.startup:
```

```
0000000000000000 <main>:
 0: f3 0f 1e fa        endbr64
 4: 48 83 ec 08        sub    $0x8,%rsp
 8: 8b 35 00 00 00 00    mov    0x0(%rip),%esi      # e <main+0xe>
 e: 8b 3d 00 00 00 00    mov    0x0(%rip),%edi      # 14 <main+0x14>
14: e8 00 00 00 00        call   19 <main+0x19>
19: 8b 0d 00 00 00 00    mov    0x0(%rip),%ecx      # 1f <main+0x1f>
1f: 8b 15 00 00 00 00    mov    0x0(%rip),%edx      # 25 <main+0x25>
25: be 00 00 00 00        mov    $0x0,%esi
2a: 41 89 c0        mov    %eax,%r8d
2d: bf 01 00 00 00        mov    $0x1,%edi
32: 31 c0        xor    %eax,%eax
34: e8 00 00 00 00        call   39 <main+0x39>
39: 31 c0        xor    %eax,%eax
3b: 48 83 c4 08        add    $0x8,%rsp
3f: c3             ret
```



# gcc和ld链接

不能用ld链接么？

- man gcc

```
to write -Xlinker "-assert definitions", because this passes the entire string as a single argument, which is not what the linker expects.
```

When using the GNU linker, it is usually more convenient to pass arguments to linker options using the option=value syntax than as separate arguments. For example, you can specify **-Xlinker -Map=output.map** rather than **-Xlinker -Map -Xlinker output.map**. Other linkers may not support this syntax for command-line options.

#### **-Wl,option**

Pass option as an option to the linker. If option contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, **-Wl,-Map,output.map** passes **-Map output.map** to the linker. When using the GNU linker, you can also get the same effect with **-Wl,-Map=output.map**.

NOTE: In Ubuntu 8.10 and later versions, for LDFLAGS, the option **-Wl,-z,relro** is used. To disable, use **-Wl,-z,norelro**.

#### **-u symbol**

Pretend the symbol symbol is undefined, to force linking of library modules to define it. You can use **-u** multiple times with different symbols to force loading of additional library modules.

#### **-z keyword**

**-z** is passed directly on to the linker along with the keyword keyword. See the section in the documentation of your linker for permitted values and their meanings.

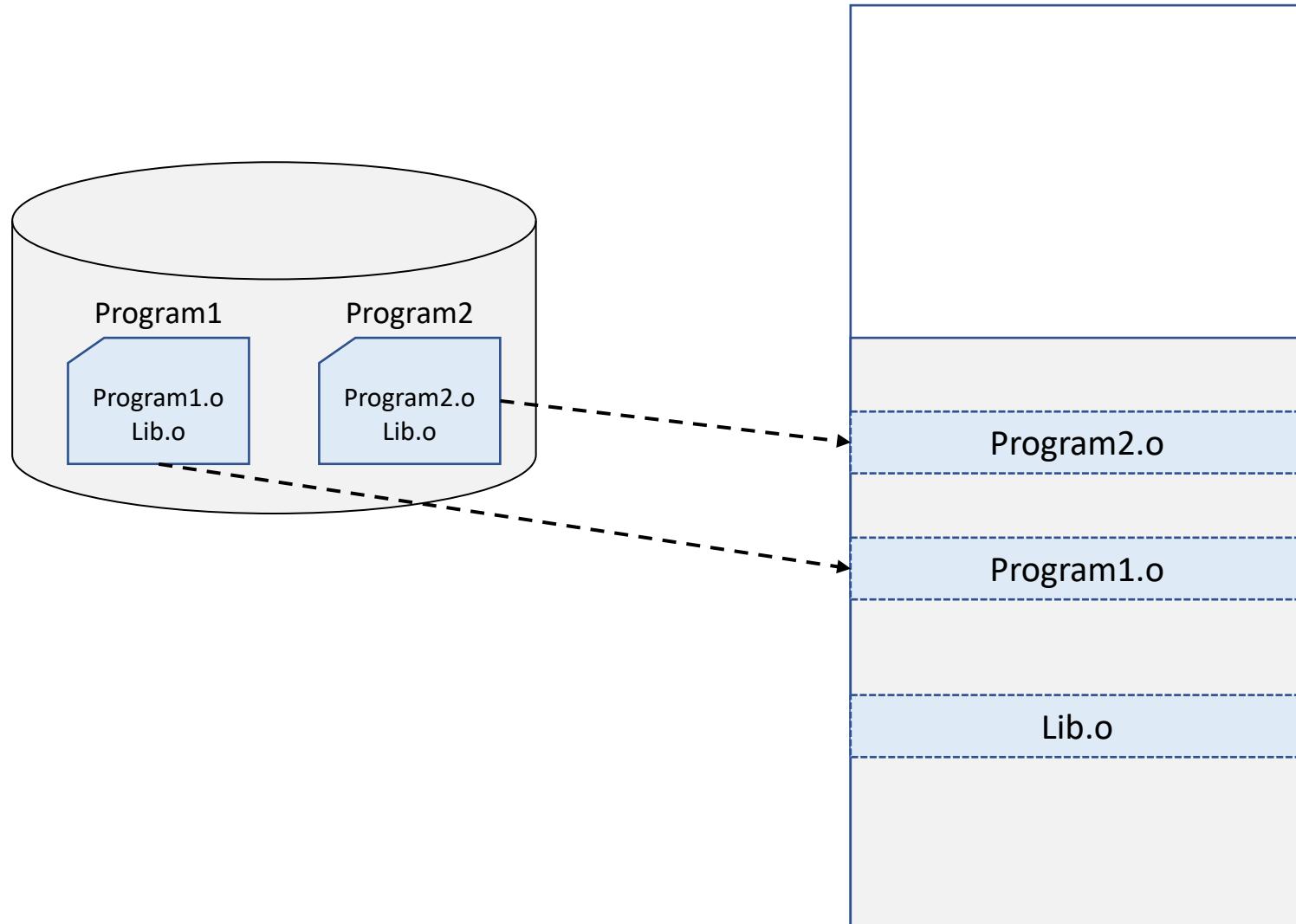
#### **Options for Directory Search**

These options specify directories to search for header files, for libraries and for parts of the compiler:

```
-I dir  
-isystem dir
```



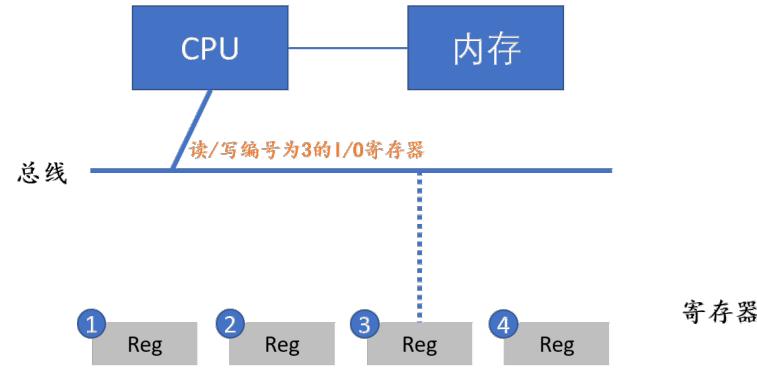
# 为什么要动态链接?



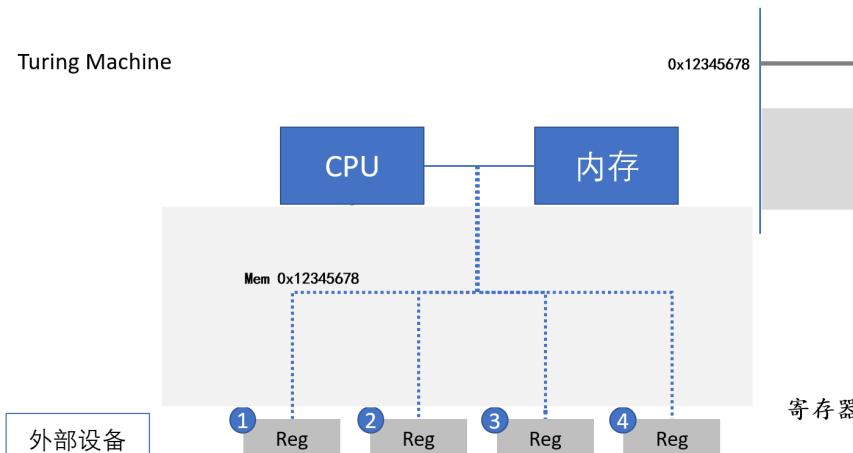
# Week10: I/O设备选讲

- CPU可以直接通过指令读写这些寄存器

- Port-mapped I/O (PMIO)
  - I/O地址空间 (port)
  - CPU直连I/O总线



- Memory-mapped I/O (MMIO)
  - 直观：使用普通内存读写指令就能访问
  - 带来了一些设计和实现的麻烦：编译器优化、缓存、乱序执行



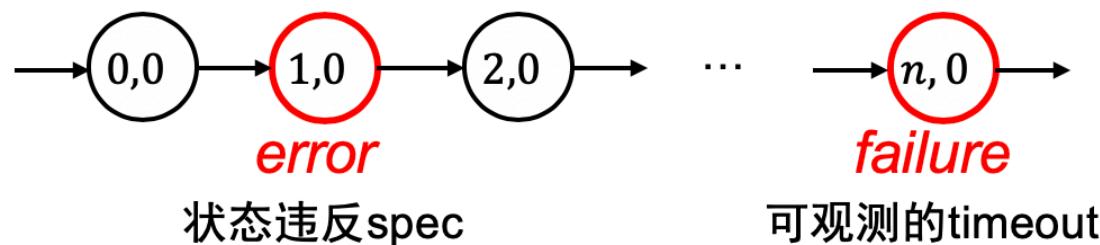
# Week11: 调试理论

- 因为 bug 的触发经历了漫长的过程
    - 需求 → 设计 → 代码 → Fault (bug) → Error (程序状态错) → Failure
      - 我们只能观测到 failure (可观测的结果错)
      - 我们可以检查状态的正确性 (但非常费时)
      - 无法预知 bug 在哪里 (每一行 “看起来” 都挺对的)

**for (int i = 0; i < n; i++)** *fault* 程序bug

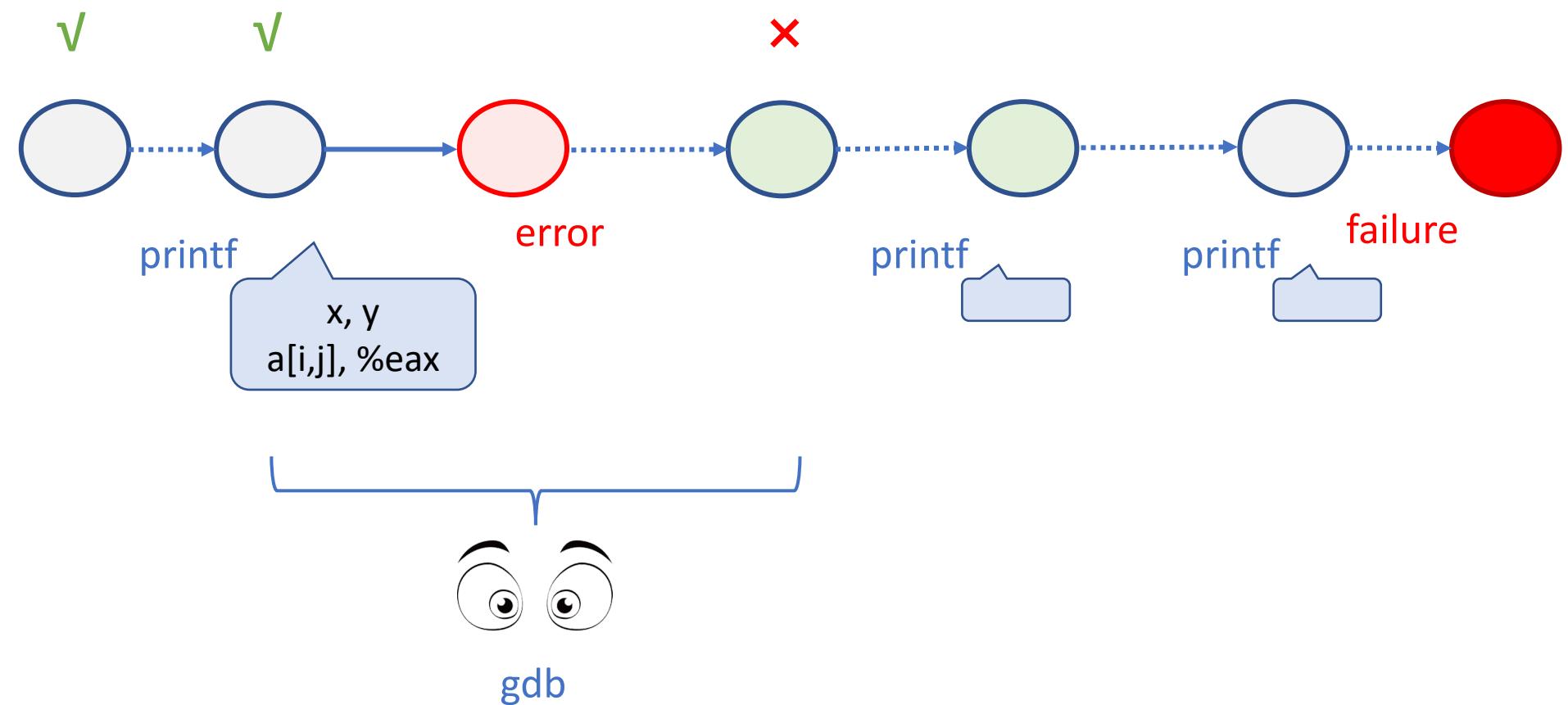
```
for (int j = 0; j < n; i++)
```

1

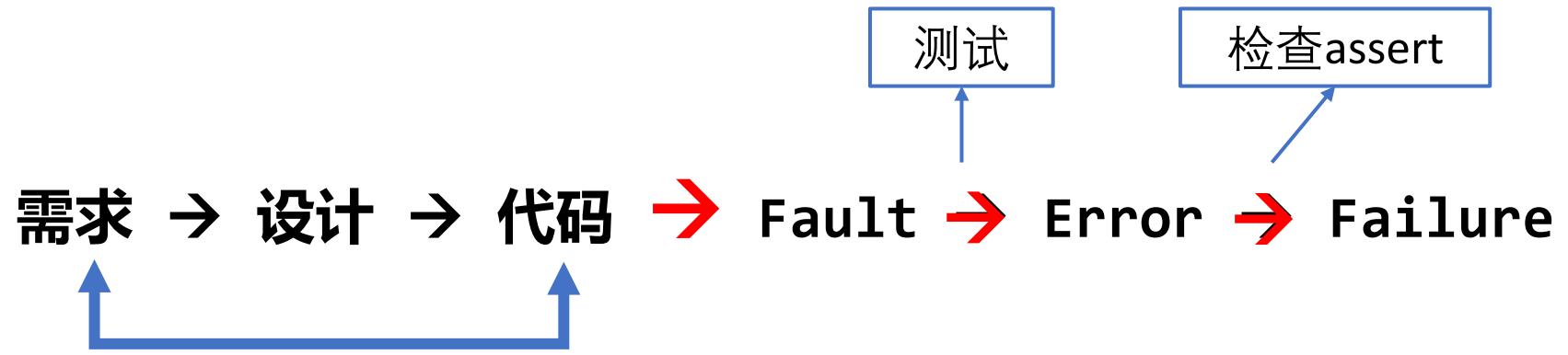


# 调试理论 (cont'd)

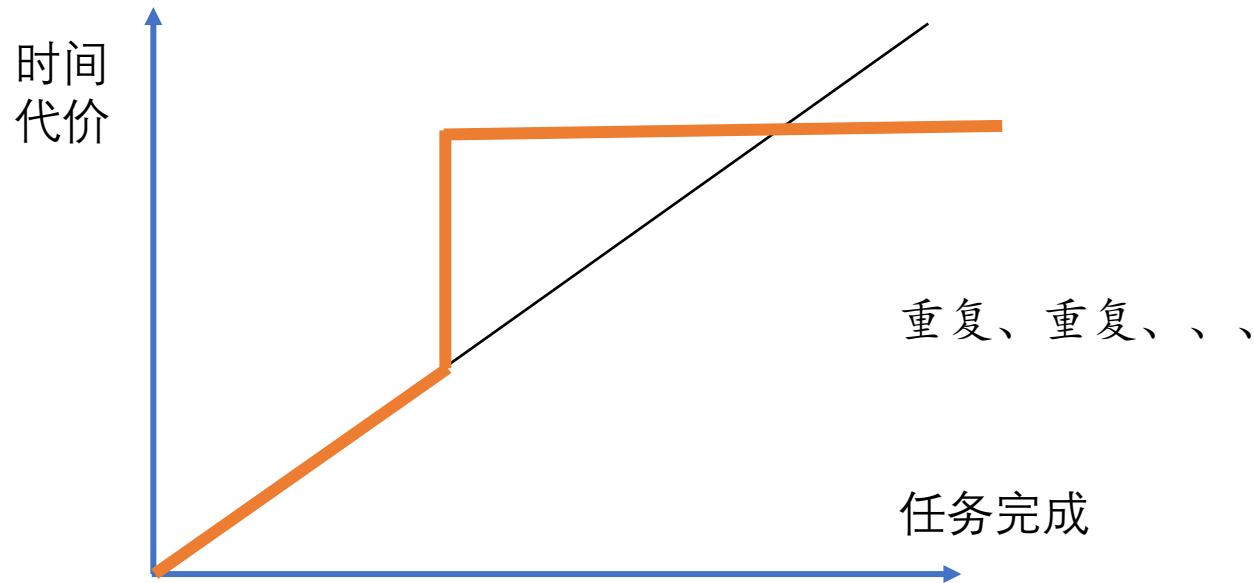
- 尝试接近状态的判定



# 调试理论：



# Week12: 系统编程与基础设施

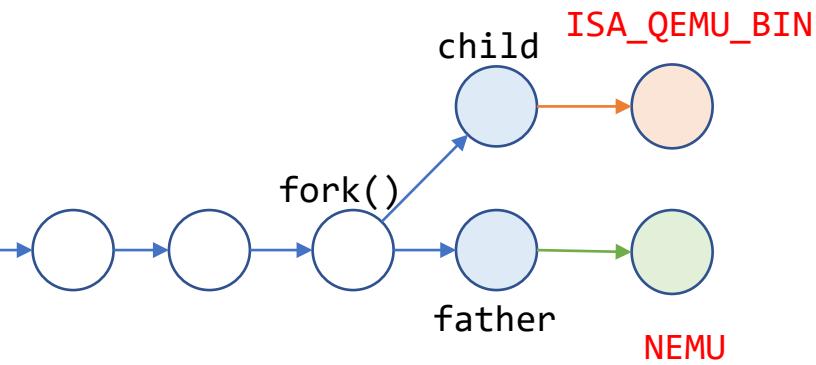
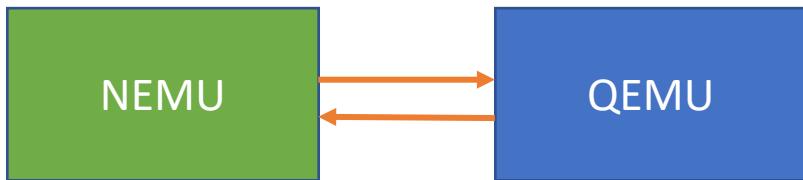


# Differential Testing

“同一套接口 (API) 的两个实现应当行为完全一致”

- 大腿同学 & 小腿同学：指令集的两套实现
  - 还有什么现实中软件系统的例子？
- 你能找到两份独立实现的东西，都可以测试
  - 浏览器 [Mesbah and Prasad, ICSE'11](#)
  - GCC (vs clang), [Yang et al., PLDI'11](#)
  - 文件系统, [Min et al., SOSP'15](#)
  - 数据库 [Rigger and Su, OSDI'20](#)
  - Gcov (vs llvm-cov)真的能在 gcc/llvm 里发现很多 bugs
  - DL library [Pham et al., ICSE'19](#)

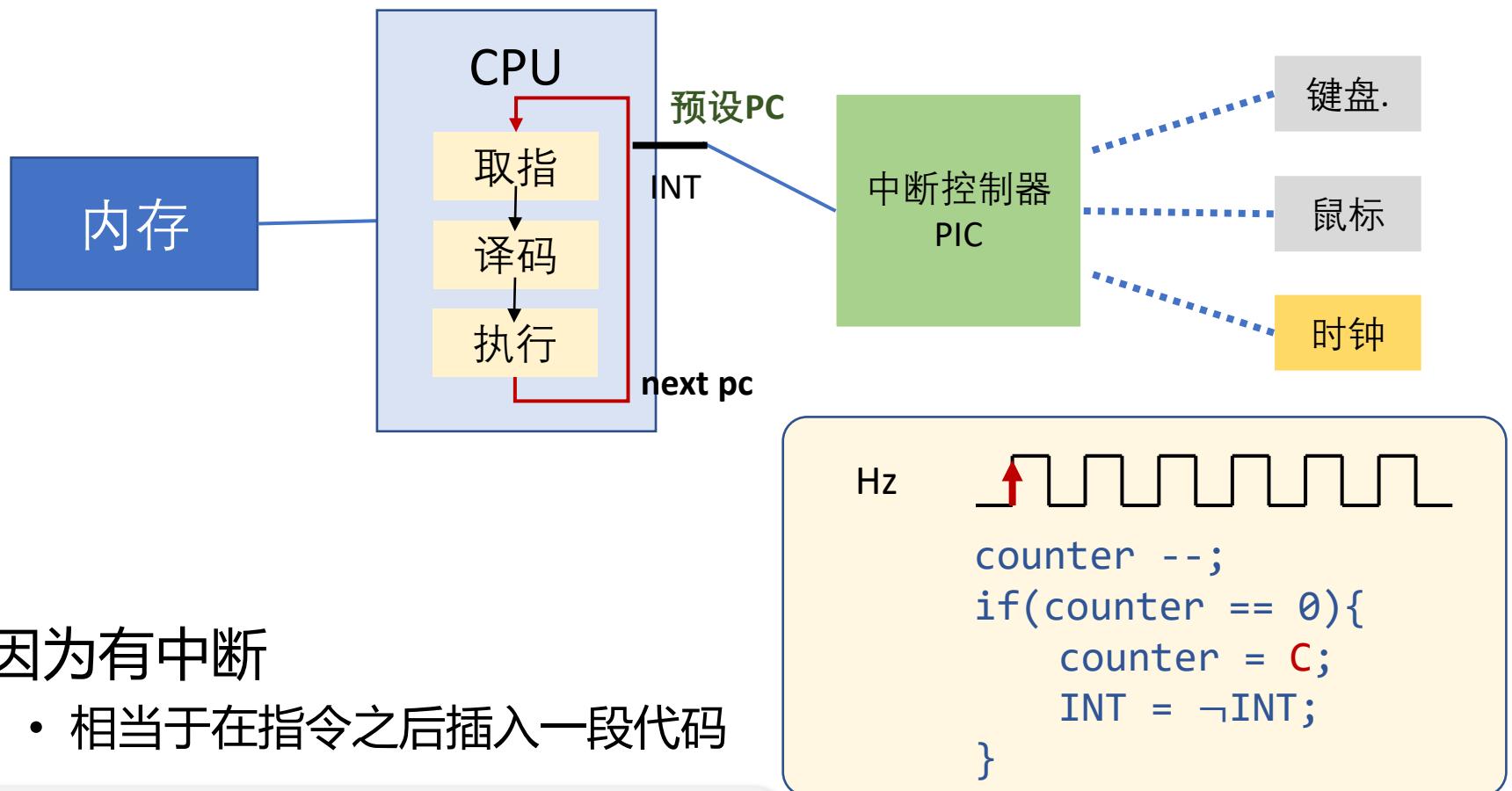
# NEMU和QEMU协作



```
9 #define ISA_QEMU_BIN "qemu-system-riscv32"
10 #define ISA_QEMU_ARGS "-bios", "none",
11 #elif defined(CONFIG_ISA_riscv64)
12 #define ISA_QEMU_BIN "qemu-system-riscv64"
```

```
38 void difftest_init(int port) {
39     char buf[32];
40     sprintf(buf, "tcp::%d", port);
41
42     int ppid_before_fork = getpid();
43     int pid = fork();
44     if (pid == -1) {
45         perror("fork");
46         assert(0);
47     }
48     else if (pid == 0) {
49         // child
50
51         // install a parent death signal in the chlid
52         int r = prctl(PR_SET_PDEATHSIG, SIGTERM);
53         if (r == -1) {
54             perror("prctl error");
55             assert(0);
56         }
57
58         if (getppid() != ppid_before_fork) {
59             printf("parent has died!\n");
60             assert(0);
61         }
62
63         close(STDIN_FILENO);
64         execvp(ISA_QEMU_BIN, ISA_QEMU_BIN, ISA_QEMU_ARGS "-S", "-gdb", buf,
65                "-serial", "none", "-monitor", "none", NULL);
66         perror("exec");
67         assert(0);
68     }
69     else {
70         // father
71
72         gdb_connect_qemu(port);
73         printf("Connect to QEMU with %s successfully\n", buf);
74
75         atexit(gdb_exit);
76
77         init_isa();
78     }
79 }
```

# Week13: 中断与分时多任务

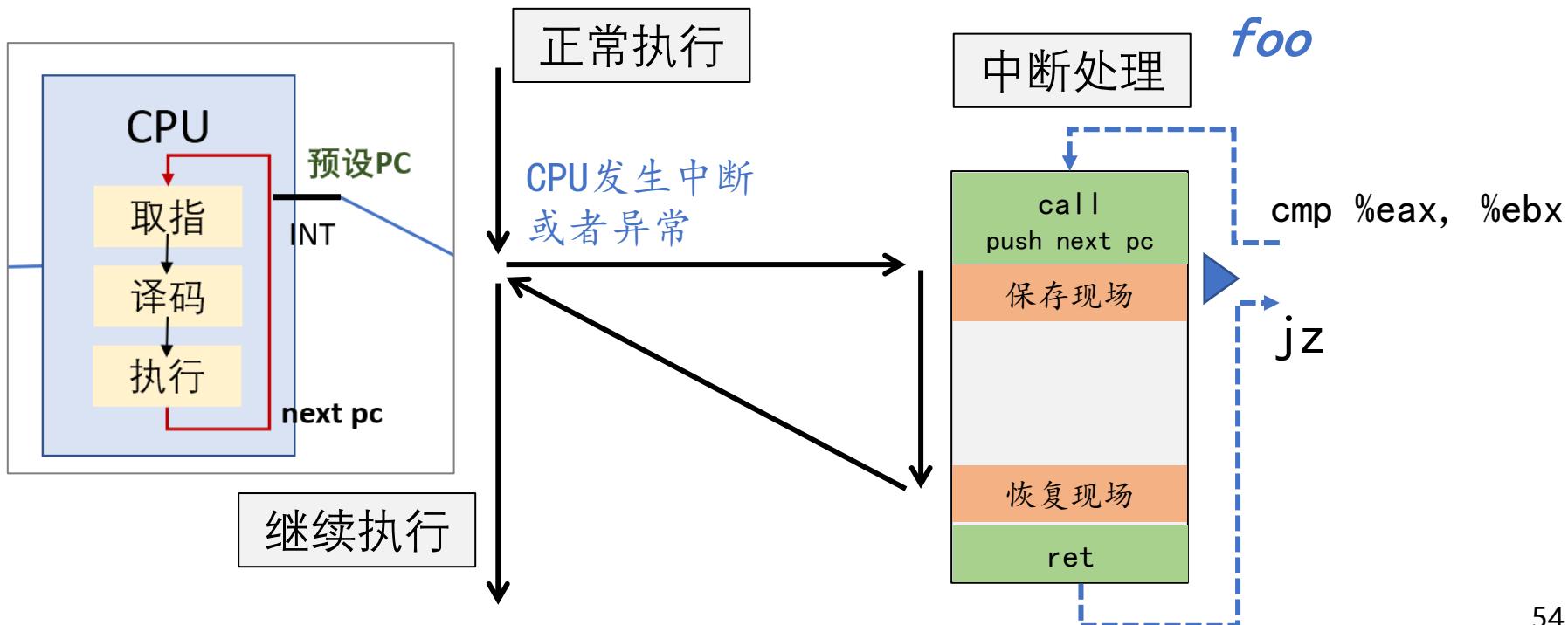


- 因为有中断
  - 相当于在指令之后插入一段代码

```
if (has_interrupt && int_enabled) {  
    interrupt_handler();  
}
```

# 中断的实现

- 比“函数调用”复杂一些
  - 函数调用需要保存 PC 到堆栈 (%rsp)
  - 但中断不仅要保存 PC (尤其是在有特权级切换的时候)
- 中断处理
  - 自动保存 RIP, CS, RFLAGS, RSP, SS, (Error Code)





正常执行

中断发生

保存A现场

中断处理

继续执行

恢复A现场

恢复B现场

正常执行

继续执行

A现场  
(包括中断前的rsp)

ss
rsp
eflags
cs
rip
\$0
\$20
6格
15个registers

保存A现场

中  
断  
处  
理

B现场  
(包括中断前的rsp)

ss
rsp
eflags
cs
rip
\$0
\$20
6格
15个registers

恢复B现场

# 中断 + 更大的内存 = 分时多线程

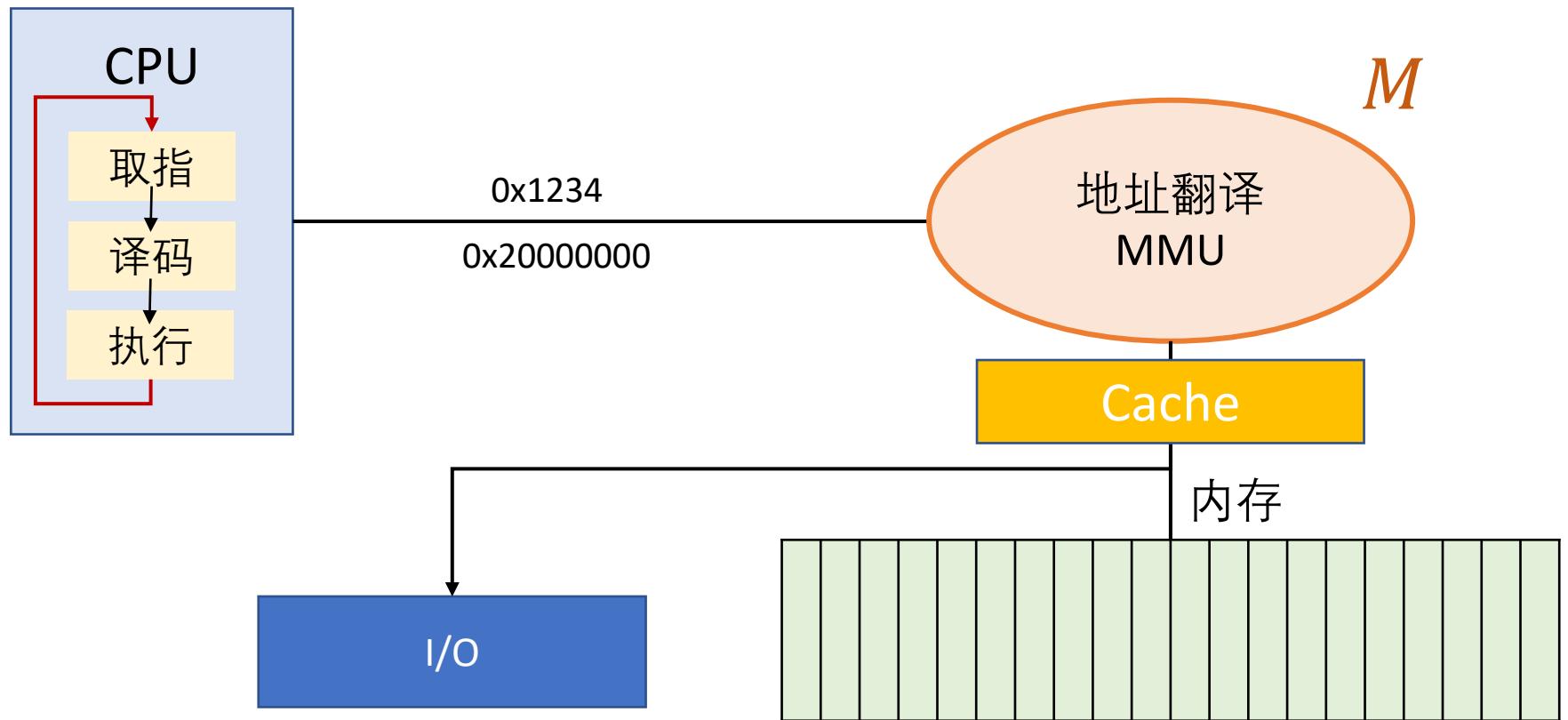
```
void foo() { while (1) printf("a"); }
void bar() { while (1) printf("b"); }
```

- 能够让foo()和bar()“同时”在处理器上执行?
  - 借助每条语句后被动插入的interrupt\_handler()调用

```
void interrupt_handler() {
    dump_regs(current->regs);
    current = (current->func == foo) ? bar : foo;
    restore_regs(current->regs);
}
```

- 操作系统背负了“调度”的职责

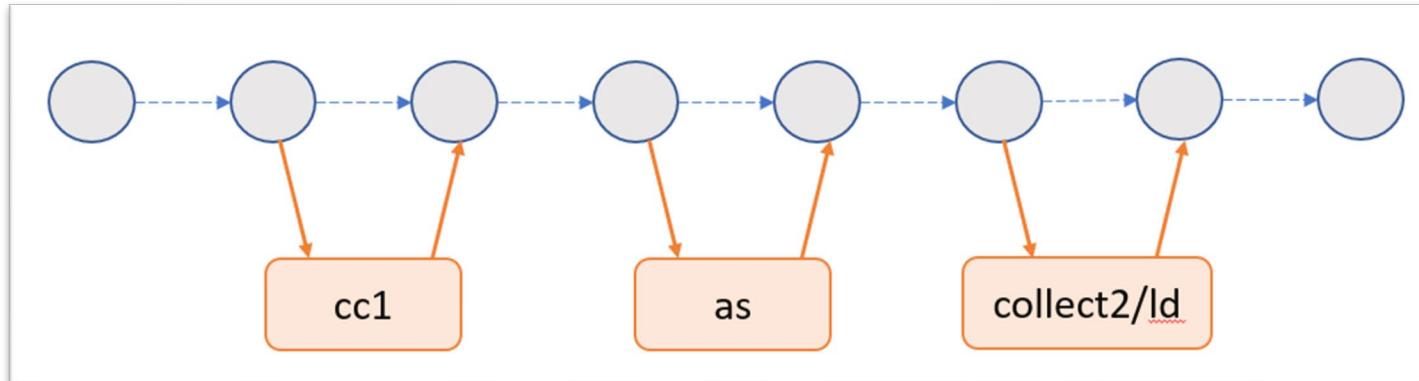
# Week14: 虚拟存储选讲



```
volatile int *p = .....;  
for (int i = 1; i < 1000; i++)  
    *p = 1;
```

# W15：造轮子的方法和乐趣

- 在IDE里，为什么按一个键，就能够编译运行？
  - 编译、链接
    - .c → 预编译 → .i → 编译 → .s → 汇编 → .o → 链接 → a.out
  - 加载执行
    - ./a.out
- 现在，一位同学对这个过程提出了质疑
  - 我不信！我就觉得是 gcc 一个程序直接搞定的
    - 道理上完全可以这么实现
  - 如何说服这位同学？



why@why-VirtualBox: ~/Documents/ICS2021/teach/Course17

why@why-VirtualBox: ~/Documents/ICS2021/teach/Course15

```
1 execve("/usr/lib/ccache/gcc", ["gcc", "a.c"], 0x7ffd5a558738 /* 60 vars */) = 0
2 execve("/usr/bin/gcc", ["/usr/bin/gcc", "a.c"], 0x56492d0e5320 /* 61 vars */) = 0
3 [pid 17480] execve("/usr/lib/gcc/x86_64-linux-gnu/10/cc1", ["/usr/lib/gcc/x86_64-
4 [pid 17480] <... execve resumed>          = 0
5 [pid 17481] execve("/usr/lib/ccache/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
6 [pid 17481] execve("/home/why/.local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT
7 [pid 17481] execve("/usr/local/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o",
8 [pid 17481] execve("/usr/local/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "
9 [pid 17481] execve("/usr/sbin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/
10 [pid 17481] execve("/usr/bin/as", ["as", "--64", "-o", "/tmp/cc6160LT.o", "/tmp/c
11 [pid 17481] <... execve resumed>          = 0
12 [pid 17482] execve("/usr/lib/gcc/x86_64-linux-gnu/10/collect2", ["/usr/lib/gcc/x8
13 [pid 17482] <... execve resumed>          = 0
14 [pid 17483] execve("/usr/bin/ld", ["/usr/bin/ld", "-plugin", "/usr/lib/gcc/x86_64
15 [pid 17483] <... execve resumed>          = 0
```

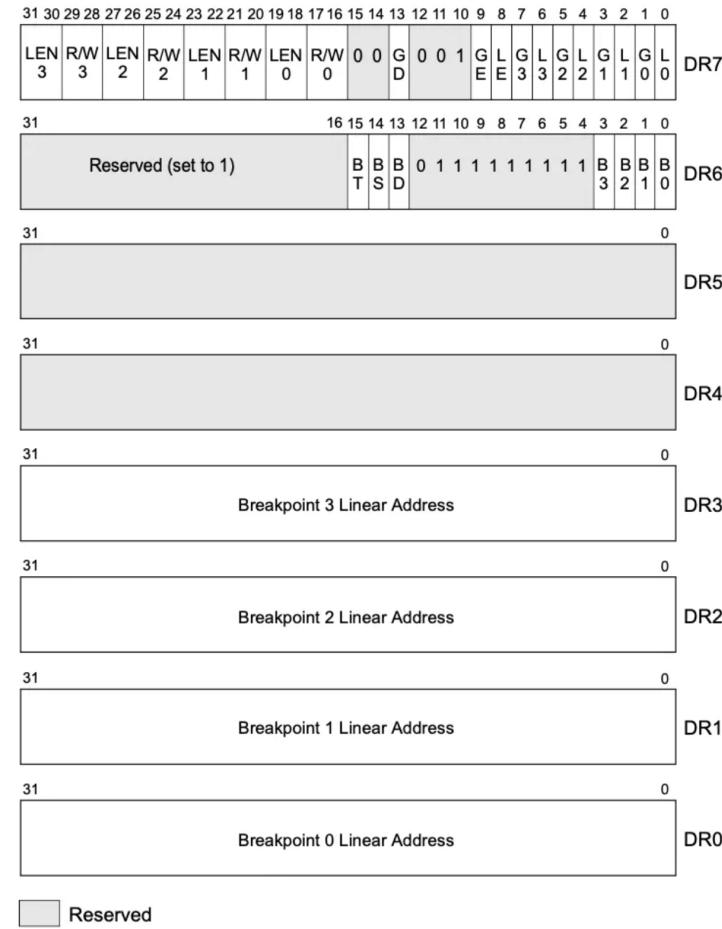
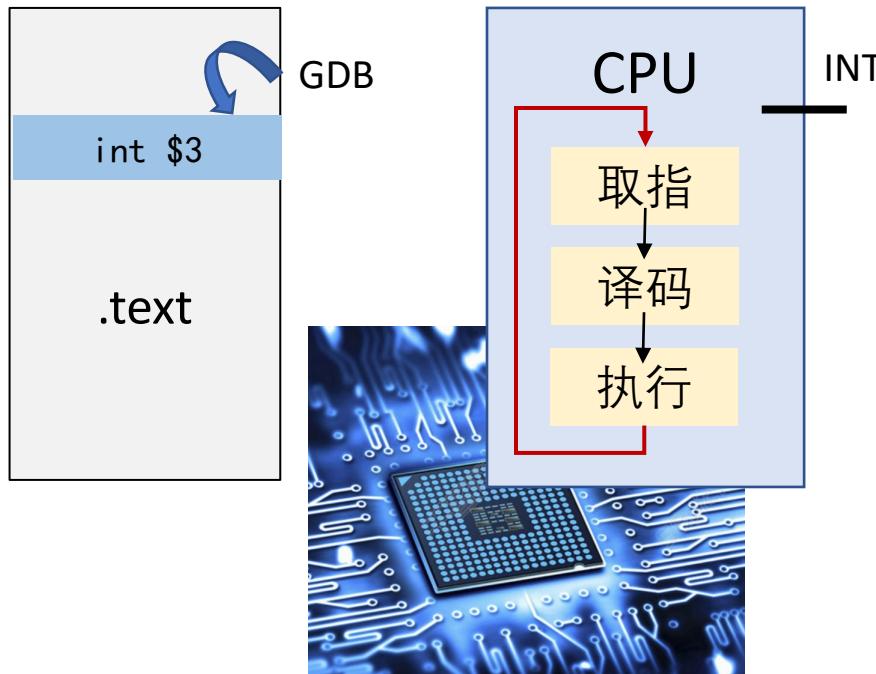
~  
~  
~  
~  
~  
~  
~  
~  
~

[No Name][+1]

[strace] unix utf-8 Ln 1, Col 1/15

# 调试器GDB

- 好奇它是怎么实现的?
- 考虑核心功能: 在任意 PC 的断点
- 其他功能都可以基于断点实现
  - 单步调试: 在下一条指令打断点
  - watch point: 单步调试 + 检查条件



# Week16：性能优化

---

- 理解程序如何编译+执行
- 理解现代处理器和存储系统
- 如何诊断性能瓶颈和提高性能
- 编译器的一些局限，理解编译器的优化痛点
  - 编译器优化前提是safe
    - Within procedure analyses
    - Static information analyses
    - “Conservative” to be “safe”
  - Two optimization blocker
    - Memory aliasing
    - Function calls

# 示例程序1.2

```
//a.c  
#include <stdio.h>  
void f1 (int *xp, int *yp){  
    *xp += *yp;  
    *xp += *yp;  
}
```

```
//b.c  
#include <stdio.h>  
void f1 (int *xp, int *yp){  
    *xp += 2* *yp;  
}
```

如果xp和yp指向同一块内存地址?

```
*xp += *xp;  
*xp += *xp;
```

```
*xp = 4 * *xp
```

```
*xp += 2* *xp;
```

```
*xp = 3 * *xp
```

Memory aliasing

```

3 #include <stdio.h>
4
5 // Function to change the value of
6 // ptr1 and ptr2
7 int foo(int* ptr1, long* ptr2)
8 {
9     *ptr1 = 10;
10    *ptr2 = 11.0;
11    return *ptr1;
12 }
13
14 // Driver Code
15 int main()
16 {
17     long data = 100.0;
18
19     // Function Call
20     int result = foo((int *)&data, &data);
21
22     // Print result
23     printf("%d \n", result);
24     return 0;
25 }

```

gcc -O2 a.c -fno-strict-aliasing

```

0000000000001149 <foo>:
1149:    f3 0f 1e fa          endbr64
114d:    c7 07 0a 00 00 00      movl   $0xa,(%rdi)
1153:    48 c7 06 0b 00 00 00  movq   $0xb,(%rsi)
115a:    8b 07
115c:    c3                   mov    (%rdi),%eax
                                ret

```

gcc -O1 a.c

```

0000000000001180 <foo>:
1180:    f3 0f 1e fa          endbr64
1184:    c7 07 0a 00 00 00      movl   $0xa,(%rdi)
118a:    b8 0a 00 00 00        mov    $0xa,%eax
118f:    48 c7 06 0b 00 00 00  movq   $0xb,(%rsi)
1196:    c3                   ret

```

gcc -O2 a.c

-fstack-reuse=[all named_vars none]	all
-fstdarg-opt	[enabled]
-fstore-merging	[enabled]
-fstrict-aliasing	[enabled]
-fstrict-enums	[available in C++, ObjC++]
-fstrict-volatile-bitfields	[enabled]
-fthread-jumps	[enabled]

# Inline substitution

```
long f();  
  
long func1(){  
    return f() + f() + f() + f();  
}  
  
long func2(){  
    return 4 * f();  
}
```

gcc -O0 a.c

gcc -O1 a.c

gcc -O2 a.c

0000000000000000 <func1>:  
0: 48 89 e5 mov %rsp,%rbp  
4: 53 push %rbx  
8: 48 83 ec 08 sub \$0x8,%rsp  
d: b8 00 00 00 00 mov \$0x0,%eax  
12: e8 00 00 00 00 call 17 <func1+0x17>  
17: 48 89 c3 mov %rax,%rbx  
1a: b8 00 00 00 00 mov \$0x0,%eax  
1f: e8 00 00 00 00 call 24 <func1+0x24>  
24: 48 01 c3 add %rax,%rbx  
27: b8 00 00 00 00 mov \$0x0,%eax  
2c: e8 00 00 00 00 call 31 <func1+0x31>  
31: 48 01 c3 add %rax,%rbx  
34: b8 00 00 00 00 mov \$0x0,%eax  
39: e8 00 00 00 00 call 3e <func1+0x3e>  
3e: 48 01 d8 add %rbx,%rax  
41: 48 8b 5d f8 mov -0x8(%rbp),%rbx  
45: c9 leave  
46: c3 ret

0000000000000000 <func1>:  
0: f3 0f 1e fa endbr64  
4: 48 8b 05 00 00 00 00 mov 0x0(%rip),%rax # b <func1+0xb>  
b: 48 8d 50 04 lea 0x4(%rax),%rdx  
f: 48 8d 04 85 06 00 00 lea 0x6(%rax,4),%rax  
16: 00  
17: 48 89 15 00 00 00 00 mov %rdx,0x0(%rip) # 1e <func1+0x1e>  
1e: c3 ret  
1f: 90 nop

# 超标量处理器

## Superscalar Issue (Pentium)

Cycle	1	2	3	4	5	6	7	8	9
Instr <sub>1</sub>	Fetch	Decode	Execute			Write			
Instr <sub>2</sub>	Fetch	Decode	Wait			Execute	Write		
Instr <sub>3</sub>		Fetch	Decode	Execute	Write				
Instr <sub>4</sub>		Fetch	Decode	Wait			Execute	Write	
Instr <sub>5</sub>			Fetch	Decode	Execute	Write			
Instr <sub>6</sub>			Fetch	Decode	Execute	Write			
Instr <sub>7</sub>				Fetch	Decode	Execute	Write		
Instr <sub>8</sub>				Fetch	Decode	Execute	Write		

- Superscalar issue allows multiple instructions to be issued at the same time

# 分支预测

- 预测分支的跳转

```
404663: mov $0x0, %eax
404668: cmp (%rdi), %rsi
40466b: jge 404685 ← How to continue?
40466d: mov 0x8(%rdi), %rax
.....
404685: repz retq
```

} Executing

- 本质：猜测并预测分支的走向
- [java - Why is processing a sorted array faster than processing an unsorted array? - Stack Overflow](#)

# 这学期学了什么？

`printf()` 在计算机软件/硬件系统上，各自发生了什么？

- CPU (NEMU) 始终在执行指令
  - (用户代码执行)
  - 库函数 (Navy, ...)
  - 系统调用 wrapper
  - `syscall / int $0x80`
  - (操作系统代码)
  - 中断/异常处理程序 (AbstractMachine)
  - 系统调用实现

# 福利：如何阅读汇编代码

# 期末试题 (2018)

---

- 期末真题 [dump.txt](#)
- 一些观察
  - 试卷同时提供 C 和对应的汇编代码
    - 并不严格需要 (可以用来检查你的理解是否准确)
  - 编译不带优化 (默认 -O0)
    - 大量冗余的视觉干扰
  - “7 ± 2 法则”
    - 除非刻意训练，否则人的短时记忆能力非常有限
      - George A. Miller, 1956
  - 抱怨做不完？先花 20 分钟把代码改写一下

```
0 08048530 <sort>:  
1 8048530: 55 push %ebp  
2 8048531: e5 mov %esp,%ebp  
3 8048533: ec 10 sub $0x10,%esp  
4 8048536: 45 f8 00 00 00 00 00 movl $0x0,-0x8(%ebp)  
5 804853d: 45 fc 00 00 00 00 00 movl $0x0,-0x4(%ebp)  
6 8048544: 93 00 00 00 jmp 80485dc <sort+0xac>  
7 8048549: 45 fc mov -0x4(%ebp),%eax  
8 804854c: b6 14 c5 60 a0 04 08 movzbl 0x804a060(,%eax,8),%edx  
9 8048554: 45 fc mov -0x4(%ebp),%eax  
10 8048557: c0 01 add $0x1,%eax  
11 804855a: b6 04 c5 60 a0 04 08 movzbl 0x804a060(,%eax,8),%eax  
12 8048562: 38 c2 cmp %al,%dl  
13 8048564: 76 72 jbe 80485d8 <sort+0xa8>  
14 8048566: 45 fc mov -0x4(%ebp),%eax  
15 8048569: b6 04 c5 60 a0 04 08 movzbl 0x804a060(,%eax,8),%eax  
16 8048571: 45 f0 mov %al,-0x10(%ebp)  
17 8048574: 45 fc mov -0x4(%ebp),%eax  
18 8048577: 04 c5 64 a0 04 08 mov 0x804a064(,%eax,8),%eax  
19 804857e: 45 f4 mov %eax,-0xc(%ebp)  
... ... ...  
38 80485d1: 45 f8 01 00 00 00 movl $0x1,-0x8(%ebp)  
39 80485d8: 45 fc 01 addl $0x1,-0x4(%ebp)  
40 80485dc: 80 86 04 08 mov 0x8048680,%eax  
41 80485e1: e8 01 sub $0x1,%eax  
42 80485e4: 45 fc cmp -0x4(%ebp),%eax  
43 80485e7: 8f 5c ff ff ff jg 8048549 <sort+0x19>  
44 80485ed: 7d f8 00 cmpl $0x0,-0x8(%ebp)  
45 80485f1: 85 3f ff ff ff jne 8048536 <sort+0x6>  
46 80485f7: 90 nop  
47 80485f8: c9 leave  
48 80485f9: c3 ret
```

```

0 08048530 <sort>:
1 8048530: 55          push  %ebp
2 8048531: 89 e5        mov   %esp,%ebp
3 8048533: 83 ec 10    sub   $0x10,%esp
4 8048536: c7 45 f8 00 00 00 00 00  movl  $0x0,-0x8(%ebp) //j = 0
5 804853d: c7 45 fc 00 00 00 00 00  movl  $0x0,-0x4(%ebp) //i = 0
6 8048544: e9 93 00 00 00  jmp   80485dc <sort+0xac>
7 8048549: 8b 45 fc    mov   -0x4(%ebp),%eax //ra = i
8 804854c: 0f b6 14 c5 60 a0 04 08  movzbl 0x804a060(,%eax,8),%edx //rd=*(u8*)(arr+ra*8)
9 8048554: 8b 45 fc    mov   -0x4(%ebp),%eax //ra = i
10 8048557: 83 c0 01   add   $0x1,%eax      //ra = ra + 1
11 804855a: 0f b6 04 c5 60 a0 04 08 08  movzbl 0x804a060(,%eax,8),%eax //ra=*(u8*)(arr+ra*8)
12 8048562: 38 c2        cmp   %al,%dl       //compare((u8)ra, (u8)rd)
13 8048564: 76 72        jbe   80485d8 <sort+0xa8> //if (<=) goto 39
14 8048566: 8b 45 fc    mov   -0x4(%ebp),%eax //ra = i
15 8048569: 0f b6 04 c5 60 a0 04 08 08  movzbl 0x804a060(,%eax,8),%eax //ra=*(u8*)(arr+ra*8)
16 8048571: 88 45 f0   mov   %al,-0x10(%ebp) // stack1 = (u8)ra
17 8048574: 8b 45 fc    mov   -0x4(%ebp),%eax // ra = i
18 8048577: 8b 04 c5 64 a0 04 08 08  mov   0x804a064(,%eax,8),%eax //ra=*(arr-data+ra*8)
19 804857e: 89 45 f4   mov   %eax,-0xc(%ebp) // stack2 = ra
...
38 80485d1: c7 45 f8 01 00 00 00
39 80485d8: 83 45 fc 01
40 80485dc: a1 80 86 04 08
41 80485e1: 83 e8 01
42 80485e4: 3b 45 fc
43 80485e7: 0f 8f 5c ff ff ff
44 80485ed: 83 7d f8 00
45 80485f1: 0f 85 3f ff ff ff
46 80485f7: 90
47 80485f8: c9
48 80485f9: c3

push  %ebp
mov   %esp,%ebp
sub   $0x10,%esp
movl  $0x0,-0x8(%ebp) //j = 0
movl  $0x0,-0x4(%ebp) //i = 0
jmp   80485dc <sort+0xac>
mov   -0x4(%ebp),%eax //ra = i
movzbl 0x804a060(,%eax,8),%edx //rd=*(u8*)(arr+ra*8)
mov   -0x4(%ebp),%eax //ra = i
add   $0x1,%eax      //ra = ra + 1
movzbl 0x804a060(,%eax,8),%eax //ra=*(u8*)(arr+ra*8)
cmp   %al,%dl       //compare((u8)ra, (u8)rd)
jbe   80485d8 <sort+0xa8> //if (<=) goto 39
mov   -0x4(%ebp),%eax //ra = i
movzbl 0x804a060(,%eax,8),%eax //ra=*(u8*)(arr+ra*8)
mov   %al,-0x10(%ebp) // stack1 = (u8)ra
mov   -0x4(%ebp),%eax // ra = i
mov   0x804a064(,%eax,8),%eax //ra=*(arr-data+ra*8)
mov   %eax,-0xc(%ebp) // stack2 = ra
...
movl  $0x1,-0x8(%ebp)
addl  $0x1,-0x4(%ebp) //i = i + 1
mov   0x8048680,%eax //ra = *(mem) (ra = n)
sub   $0x1,%eax      //ra = ra - 1
cmp   -0x4(%ebp),%eax //compare (i, ra)
jg    8048549 <sort+0x19> //if (>), goto 7
cmpl  $0x0,-0x8(%ebp)
jne   8048536 <sort+0x6>
nop
leave
ret

```

```

/* main.c */
#include <stdio.h>
#include <stdlib.h>
typedef struct record {
    .....
} RECORD;
typedef struct index {
    unsigned char key;
    RECORD *pdata;
} INDEX;
extern void sort();
INDEX rec_idx[256];
const int rec_num = 256;
void main(int argc, char *argv[])
{
    FILE *fp = fopen(argv[1], "rb");
    if (fp) {
        fread(rec_idx, sizeof(INDEX), rec_num, fp);
        fclose(fp);
    } else exit(1);
    sort();
}

```

```

/* sort.c: bubble sort */
extern INDEX rec_idx[];
extern const int rec_num;

void sort()
{
    int i, swapped;
    INDEX temp;
    do { swapped = 0;
        for(i=0; i<rec_num-1; i++)
            if (rec_idx[i].key > rec_idx[i+1].key) {
                temp.key = rec_idx[i].key;
                temp_pdata = rec_idx[i].pdata;
                rec_idx[i].key = rec_idx[i+1].key;
                rec_idx[i].pdata = rec_idx[i+1].pdata;
                rec_idx[i+1].key = temp.key;
                rec_idx[i+1].pdata = temp_pdata;
                swapped = 1;
            }
        } while (swapped);
}

```

高地址

```
0 080491b6 <func>:  
1 80491b6: f3 0f 1e fb endbr32  
2 80491ba: 55 push %ebp  
3 80491bb: 89 e5 mov %esp,%ebp  
4 80491bd: 53 push %ebx  
5 80491be: 83 ec 04 sub $0x4,%esp  
6 80491c1: 8b 45 0c mov 0xc(%ebp),%eax  
7 80491c4: 3b 45 10 cmp 0x10(%ebp),%eax  
8 80491c7: 75 13 jne 80491dc <func+0x26>  
9 80491c9: 8b 45 0c mov 0xc(%ebp),%eax  
10 80491cc: 8d 14 85 00 00 00 00 lea 0x0(,%eax,4),%edx  
11 80491d3: 8b 45 08 mov 0x8(%ebp),%eax  
12 80491d6: 01 d0 add %edx,%eax  
13 80491d8: 8b 00 mov (%eax),%eax  
14 80491da: eb 2b jmp 8049207 <func+0x51>  
15 80491dc: 8b 45 0c mov 0xc(%ebp),%eax  
16 80491df: 8d 14 85 00 00 00 00 lea 0x0(,%eax,4),%edx  
17 80491e6: 8b 45 08 mov 0x8(%ebp),%eax  
18 80491e9: 01 d0 add %edx,%eax  
19 80491eb: 8b 18 mov (%eax),%ebx  
20 80491ed: 8b 45 0c mov 0xc(%ebp),%eax  
21 80491f0: 83 c0 01 add $0x1,%eax  
22 80491f3: 83 ec 04 sub $0x4,%esp  
23 80491f6: ff 75 10 push 0x10(%ebp)  
24 80491f9: 50 push %eax  
25 80491fa: ff 75 08 push 0x8(%ebp)  
26 80491fd: e8 b4 ff ff ff call 80491b6 <func>  
27 8049202: 83 c4 10 add $0x10,%esp  
28 8049205: 01 d8 add %ebx,%eax  
29 8049207: 8b 5d fc mov -0x4(%ebp),%ebx  
30 804920a: c9 leave  
31 804920b: c3 ret
```

低地址

高地址

%ebp  
%esp

%ebp

%ebx

低地址

0 80491b6 <func>:	
1 80491b6: f3 0f 1e fb	endbr32
2 80491ba: 55	push %ebp
3 80491bb: 89 e5	mov %esp,%ebp
4 80491bd: 53	push %ebx
5 80491be: 83 ec 04	sub \$0x4,%esp
6 80491c1: 8b 45 0c	mov 0xc(%ebp),%eax
7 80491c4: 3b 45 10	cmp 0x10(%ebp),%eax
8 80491c7: 75 13	jne 80491dc <func+0x26>
9 80491c9: 8b 45 0c	mov 0xc(%ebp),%eax
10 80491cc: 8d 14 85 00 00 00 00	lea 0x0(,%eax,4),%edx
11 80491d3: 8b 45 08	mov 0x8(%ebp),%eax
12 80491d6: 01 d0	add %edx,%eax
13 80491d8: 8b 00	mov (%eax),%eax
14 80491da: eb 2b	jmp 8049207 <func+0x51>
15 80491dc: 8b 45 0c	mov 0xc(%ebp),%eax
16 80491df: 8d 14 85 00 00 00 00	lea 0x0(,%eax,4),%edx
17 80491e6: 8b 45 08	mov 0x8(%ebp),%eax
18 80491e9: 01 d0	add %edx,%eax
19 80491eb: 8b 18	mov (%eax),%ebx
20 80491ed: 8b 45 0c	mov 0xc(%ebp),%eax
21 80491f0: 83 c0 01	add \$0x1,%eax
22 80491f3: 83 ec 04	sub \$0x4,%esp
23 80491f6: ff 75 10	push 0x10(%ebp)
24 80491f9: 50	push %eax
25 80491fa: ff 75 08	push 0x8(%ebp)
26 80491fd: e8 b4 ff ff ff	call 80491b6 <func>
27 8049202: 83 c4 10	add \$0x10,%esp
28 8049205: 01 d8	add %ebx,%eax
29 8049207: 8b 5d fc	mov -0x4(%ebp),%ebx
30 804920a: c9	leave
31 804920b: c3	ret

高地址	
p3:0x10(%ebp)	0 80491b6 <func>:
p2:0xc(%ebp)	1 80491b6: f3 0f 1e fb
p1:0x8(%ebp)	2 80491ba: 55
返回地址	3 80491bb: 89 e5
	4 80491bd: 53
%ebp	5 80491be: 83 ec 04
	6 80491c1: 8b 45 0c
%esp	7 80491c4: 3b 45 10
	8 80491c7: 75 13
%esp	9 80491c9: 8b 45 0c
	10 80491cc: 8d 14 85 00 00 00 00 00
	11 80491d3: 8b 45 08
	12 80491d6: 01 d0
	13 80491d8: 8b 00
	14 80491da: eb 2b
	15 80491dc: 8b 45 0c
	16 80491df: 8d 14 85 00 00 00 00 00
	17 80491e6: 8b 45 08
	18 80491e9: 01 d0
	19 80491eb: 8b 18
	20 80491ed: 8b 45 0c
	21 80491f0: 83 c0 01
	22 80491f3: 83 ec 04
	23 80491f6: ff 75 10
	24 80491f9: 50
	25 80491fa: ff 75 08
	26 80491fd: e8 b4 ff ff ff
	27 8049202: 83 c4 10
	28 8049205: 01 d8
	29 8049207: 8b 5d fc
	30 804920a: c9
	31 804920b: c3

低地址

```

0 80491b6 <func>:
1 80491b6: f3 0f 1e fb endbr32
2 80491ba: 55 push %ebp
3 80491bb: 89 e5 mov %esp,%ebp
4 80491bd: 53 push %ebx
5 80491be: 83 ec 04 sub $0x4,%esp
6 80491c1: 8b 45 0c mov 0xc(%ebp),%eax
7 80491c4: 3b 45 10 cmp 0x10(%ebp),%eax
8 80491c7: 75 13 jne 80491dc <func+0x26>
9 80491c9: 8b 45 0c mov 0xc(%ebp),%eax
10 80491cc: 8d 14 85 00 00 00 00 00 lea 0x0(,%eax,4),%edx
11 80491d3: 8b 45 08 mov 0x8(%ebp),%eax
12 80491d6: 01 d0 add %edx,%eax
13 80491d8: 8b 00 mov (%eax),%eax
14 80491da: eb 2b jmp 8049207 <func+0x51>
15 80491dc: 8b 45 0c mov 0xc(%ebp),%eax
16 80491df: 8d 14 85 00 00 00 00 00 lea 0x0(,%eax,4),%edx
17 80491e6: 8b 45 08 mov 0x8(%ebp),%eax
18 80491e9: 01 d0 add %edx,%eax
19 80491eb: 8b 18 mov (%eax),%ebx
20 80491ed: 8b 45 0c mov 0xc(%ebp),%eax
21 80491f0: 83 c0 01 add $0x1,%eax
22 80491f3: 83 ec 04 sub $0x4,%esp
23 80491f6: ff 75 10 push 0x10(%ebp)
24 80491f9: 50 push %eax
25 80491fa: ff 75 08 push 0x8(%ebp)
26 80491fd: e8 b4 ff ff ff call 80491b6 <func>
27 8049202: 83 c4 10 add $0x10,%esp
28 8049205: 01 d8 add %ebx,%eax
29 8049207: 8b 5d fc mov -0x4(%ebp),%ebx
30 804920a: c9 leave
31 804920b: c3 ret

```

高地址	
p3:0x10(%ebp)	0 80491b6 <func>:
p2:0xc(%ebp)	1 80491b6: f3 0f 1e fb
p1:0x8(%ebp)	2 80491ba: 55
返回地址	3 80491bb: 89 e5
	4 80491bd: 53
%ebp	5 80491be: 83 ec 04
	6 80491c1: 8b 45 0c
%ebp	7 80491c4: 3b 45 10
	8 80491c7: 75 13
%esp	9 80491c9: 8b 45 0c
	10 80491cc: 8d 14 85 00 00 00 00
	11 80491d3: 8b 45 08
	12 80491d6: 01 d0
	13 80491d8: 8b 00
	14 80491da: eb 2b
	15 80491dc: 8b 45 0c
	16 80491df: 8d 14 85 00 00 00 00
	17 80491e6: 8b 45 08
	18 80491e9: 01 d0
	19 80491eb: 8b 18
	20 80491ed: 8b 45 0c
	21 80491f0: 83 c0 01
	22 80491f3: 83 ec 04
	23 80491f6: ff 75 10
	24 80491f9: 50
	25 80491fa: ff 75 08
	26 80491fd: e8 b4 ff ff ff
	27 8049202: 83 c4 10
	28 8049205: 01 d8
	29 8049207: 8b 5d fc
	30 804920a: c9
	31 804920b: c3

低地址

```

endbr32
push %ebp
mov %esp,%ebp
push %ebx
sub $0x4,%esp
mov 0xc(%ebp),%eax //ra = p2
cmp 0x10(%ebp),%eax //compare(p3,p2)
jne 80491dc <func+0x26> //if(≠0)goto 15
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra*4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%eax //ra = *(p1+p2*4)
jmp 8049207 <func+0x51>
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra * 4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%ebx //rb = *(p1+p2*4)
mov 0xc(%ebp),%eax //ra = p2
add $0x1,%eax //ra = p2+1
sub $0x4,%esp
push 0x10(%ebp)
push %eax
push 0x8(%ebp)
call 80491b6 <func>
add $0x10,%esp
add %ebx,%eax
mov -0x4(%ebp),%ebx
leave
ret

```

高地址	
p3:0x10(%ebp)	0 80491b6 <func>:
p2:0xc(%ebp)	1 80491b6: f3 0f 1e fb
p1:0x8(%ebp)	2 80491ba: 55
返回地址	3 80491bb: 89 e5
	4 80491bd: 53
%ebp	5 80491be: 83 ec 04
	6 80491c1: 8b 45 0c
%ebp	7 80491c4: 3b 45 10
	8 80491c7: 75 13
	9 80491c9: 8b 45 0c
%esp	10 80491cc: 8d 14 85 00 00 00 00
	11 80491d3: 8b 45 08
	12 80491d6: 01 d0
	13 80491d8: 8b 00
	14 80491da: eb 2b
	15 80491dc: 8b 45 0c
	16 80491df: 8d 14 85 00 00 00 00
	17 80491e6: 8b 45 08
	18 80491e9: 01 d0
	19 80491eb: 8b 18
	20 80491ed: 8b 45 0c
	21 80491f0: 83 c0 01
	22 80491f3: 83 ec 04
	23 80491f6: ff 75 10
	24 80491f9: 50
	25 80491fa: ff 75 08
	26 80491fd: e8 b4 ff ff ff
低地址	27 8049202: 83 c4 10
	28 8049205: 01 d8
	29 8049207: 8b 5d fc
	30 804920a: c9
	31 804920b: c3

```

endbr32
push %ebp
mov %esp,%ebp
push %ebx
sub $0x4,%esp
mov 0xc(%ebp),%eax //ra = p2
cmp 0x10(%ebp),%eax //compare(p3,p2)
jne 80491dc <func+0x26> //if(≠0)goto 15
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra*4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%eax //ra = *(p1+p2*4)
jmp 8049207 <func+0x51>
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra * 4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%ebx //rb = *(p1+p2*4)
mov 0xc(%ebp),%eax //ra = p2
add $0x1,%eax //ra = p2+1
sub $0x4,%esp
push 0x10(%ebp)
push %eax
push 0x8(%ebp)
call 80491b6 <func>
add $0x10,%esp
add %ebx,%eax
mov -0x4(%ebp),%ebx
leave
ret

```

高地址	
p3:0x10(%ebp)	0 80491b6 <func>:
p2:0xc(%ebp)	1 80491b6: f3 0f 1e fb
p1:0x8(%ebp)	2 80491ba: 55
返回地址	3 80491bb: 89 e5
	4 80491bd: 53
%ebp	5 80491be: 83 ec 04
	6 80491c1: 8b 45 0c
%ebx	7 80491c4: 3b 45 10
	8 80491c7: 75 13
	9 80491c9: 8b 45 0c
	10 80491cc: 8d 14 85 00 00 00 00
	11 80491d3: 8b 45 08
	12 80491d6: 01 d0
	13 80491d8: 8b 00
	14 80491da: eb 2b
	15 80491dc: 8b 45 0c
	16 80491df: 8d 14 85 00 00 00 00
	17 80491e6: 8b 45 08
	18 80491e9: 01 d0
	19 80491eb: 8b 18
	20 80491ed: 8b 45 0c
	21 80491f0: 83 c0 01
	22 80491f3: 83 ec 04
	23 80491f6: ff 75 10
	24 80491f9: 50
	25 80491fa: ff 75 08
	26 80491fd: e8 b4 ff ff ff
	27 8049202: 83 c4 10
	28 8049205: 01 d8
	29 8049207: 8b 5d fc
	30 804920a: c9
	31 804920b: c3

低地址

%ebp →
 %ebp
  
%esp →
 p3

```

endbr32
push %ebp
mov %esp,%ebp
push %ebx
sub $0x4,%esp
mov 0xc(%ebp),%eax //ra = p2
cmp 0x10(%ebp),%eax //compare(p3,p2)
jne 80491dc <func+0x26> //if(≠0)goto 15
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra*4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%eax //ra = *(p1+p2*4)
jmp 8049207 <func+0x51>
mov 0xc(%ebp),%eax //ra = p2
lea 0x0(%eax,4),%edx //rd = ra * 4
mov 0x8(%ebp),%eax //ra = p1
add %edx,%eax //ra = p1+p2*4
mov (%eax),%ebx //rb = *(p1+p2*4)
mov 0xc(%ebp),%eax //ra = p2
add $0x1,%eax //ra = p2+1
sub $0x4,%esp
push 0x10(%ebp)
push %eax
push 0x8(%ebp)
call 80491b6 <func>
add $0x10,%esp
add %ebx,%eax
mov -0x4(%ebp),%ebx
leave
ret

```

高地址	0 80491b6 <func>:	
p3:0x10(%ebp)	1 80491b6: f3 0f 1e fb	endbr32
p2:0xc(%ebp)	2 80491ba: 55	push %ebp
p1:0x8(%ebp)	3 80491bb: 89 e5	mov %esp,%ebp
返回地址	4 80491bd: 53	push %ebx
	5 80491be: 83 ec 04	sub \$0x4,%esp
%ebp	6 80491c1: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
%ebx	7 80491c4: 3b 45 10	cmp 0x10(%ebp),%eax //compare(p3,p2)
	8 80491c7: 75 13	jne 80491dc <func+0x26> //if(≠0)goto 15
	9 80491c9: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
	10 80491cc: 8d 14 85 00 00 00 00	lea 0x0(%eax,4),%edx //rd = ra*4
p3	11 80491d3: 8b 45 08	mov 0x8(%ebp),%eax //ra = p1
	12 80491d6: 01 d0	add %edx,%eax //ra = p1+p2*4
	13 80491d8: 8b 00	mov (%eax),%eax //ra = *(p1+p2*4)
p2+1	14 80491da: eb 2b	jmp 8049207 <func+0x51>
	15 80491dc: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
	16 80491df: 8d 14 85 00 00 00 00	lea 0x0(%eax,4),%edx //rd = ra * 4
	17 80491e6: 8b 45 08	mov 0x8(%ebp),%eax //ra = p1
	18 80491e9: 01 d0	add %edx,%eax //ra = p1+p2*4
	19 80491eb: 8b 18	mov (%eax),%ebx //rb = *(p1+p2*4)
	20 80491ed: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
	21 80491f0: 83 c0 01	add \$0x1,%eax //ra = p2+1
	22 80491f3: 83 ec 04	sub \$0x4,%esp
	23 80491f6: ff 75 10	push 0x10(%ebp)
	24 80491f9: 50	push %eax
	25 80491fa: ff 75 08	push 0x8(%ebp)
	26 80491fd: e8 b4 ff ff ff	call 80491b6 <func>
低地址	27 8049202: 83 c4 10	add \$0x10,%esp
	28 8049205: 01 d8	add %ebx,%eax
	29 8049207: 8b 5d fc	mov -0x4(%ebp),%ebx
	30 804920a: c9	leave
	31 804920b: c3	ret

高地址

p3:0x10(%ebp)	0 80491b6 <func>:	
p2:0xc(%ebp)	1 80491b6: f3 0f 1e fb	endbr32
p1:0x8(%ebp)	2 80491ba: 55	push %ebp
返回地址	3 80491bb: 89 e5	mov %esp,%ebp
	4 80491bd: 53	push %ebx
%ebp	5 80491be: 83 ec 04	sub \$0x4,%esp
%ebp	6 80491c1: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
%ebx	7 80491c4: 3b 45 10	cmp 0x10(%ebp),%eax //compare(p3,p2)
	8 80491c7: 75 13	jne 80491dc <func+0x26> //if(≠0)goto 15
	9 80491c9: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
	10 80491cc: 8d 14 85 00 00 00 00	lea 0x0(%eax,4),%edx //rd = ra*4
	11 80491d3: 8b 45 08	mov 0x8(%ebp),%eax //ra = p1
p3	12 80491d6: 01 d0	add %edx,%eax //ra = p1+p2*4
p2+1	13 80491d8: 8b 00	mov (%eax),%eax //ra = *(p1+p2*4)
p1	14 80491da: eb 2b	jmp 8049207 <func+0x51>
	15 80491dc: 8b 45 0c	mov 0xc(%ebp),%eax //ra = p2
	16 80491df: 8d 14 85 00 00 00 00	lea 0x0(%eax,4),%edx //rd = ra * 4
	17 80491e6: 8b 45 08	mov 0x8(%ebp),%eax //ra = p1

func(array, a, b)

func(array, a+1, b)

func(array, b, b)

低地址


22 80491f3: 83 ec 04	sub \$0x4,%esp
23 80491f6: ff 75 10	push 0x10(%ebp)
24 80491f9: 50	push %eax
25 80491fa: ff 75 08	push 0x8(%ebp)
26 80491fd: e8 b4 ff ff ff	call 80491b6 <func>
27 8049202: 83 c4 10	add \$0x10,%esp //回收栈
28 8049205: 01 d8	add %ebx,%eax //ra = rb + func()
29 8049207: 8b 5d fc	mov -0x4(%ebp),%ebx
30 804920a: c9	leave
31 804920b: c3	ret

# 阅读代码的建议

---

- “逆向工程”
  - 从汇编代码推导出程序的行为
- 几点要素
  1. 将不容易理解的符号用容易理解的方式代替
  2. 将一段汇编代码用容易理解的方式表达
  3. 假设内存/寄存器中的数值是变量，人肉模拟调单步调试
    - 每当出现分支时，世界都会复制两份

# Happy New Year & 明年见 (?)

## Take-aways

机器永远是对的

没什么是 RTFM/RTFSC 解决不了的

知道了计算机系统这个 “状态机” 是如何工作的