

# Introduction to Computer System

By ICS Team, based on lectures by Danfeng Shan, Hao Li, and others.

These are the course notes for [COMPSCI 400727: Introduction to Computer System](#) at Xi'an Jiaotong University.

Here is the official course description:

## Info

This course is inspired by the [CMU-15-213 curriculum](#) and delves into the intricacies of computer hardware, guiding students through the step-by-step process of how C code is translated into X86\_64 assembly and executed on the CPU. The course is structured with increasing levels of complexity:

It begins with **data representation**, introducing the fundamental concepts of **assembly language**, followed by an exploration of memory structure and the significance of **cache design**. The course then covers **CPU pipelining** and strategies for program optimization to enhance performance. Additionally, we will examine how a C program transitions into machine-readable machine code, including the process of **linking**. Finally, the concept of **virtual memory** will be introduced.

As an introductory course to systems, it offers both depth and breadth, serving as a prerequisite for future research in computer architecture and network systems.

## Disclaimer: Beta

These notes have not been proofread. They likely contain errors.

If you're an ICS student at XJTU, in any case of dispute, the official course lectures are the correct source of truth.

# Corrections

As of the Spring 2025 semester, this textbook is still being actively maintained and updated.

If you see any parts that needs to be corrected, please open a Github issue [here](#).

# Source and Changelog

The source for the textbook and a log of all changes are [available on Github](#).

# License



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

# Acknowledgements

The design of this textbook is inspired by many outstanding courses. We would like to express our gratitude to the following courses that have provided us with valuable inspiration:

- [CSAPP @CMU](#)
- [CS168 @UC Berkeley](#)
- [CS161 @UC Berkeley](#)
- [Operating System @PKU](#)
- [Operating System @IPADS-SJTU](#)
- [ICS @NJU](#)

# Chapter 0 Overview

---

Hello World

Hello ICS

---

---

© 2025. ICS Team. All rights reserved.

# Why ICS

---

“在当今AI火遍全球的当下，了解计算机系统的内部工作原理比以往任何时候都更加重要”

-- ICS Team

---

乍一看，你可能会以为这是课程组在吹牛，但是等你了解到 MLSys / Parallel Computing 等领域的时候，一定会明白斯言不谬 😊

《计算机系统导论》（ICS）课程将为你提供一个全面的基础，帮助你成为一个对系统高效运行有深刻理解的程序员。

这门课会“跳脱”高层次的编程语言，侧重点在讲解程序如何执行、信息如何存储以及系统各个组件如何交互。

通过这门课程，你将获得解决性能、可移植性和鲁棒性等关键问题的技能，这对于编写高效且可靠的代码至关重要。你还将深入探讨内存管理、机器级代码生成和网络协议等核心主题，从而对系统架构有更深的理解。

ICS不仅适用于对 **操作系统** 或 **网络系统** 感兴趣的同学，它还是学习高级课程（如 **高效编译系统**、**并行与分布式系统**）的基础。

无论你是开发高性能应用程序，还是仅仅想了解“幕后”发生了什么，ICS都能让你掌握系统级编程所需的知识和工具。

---

© 2025. ICS Team. All rights reserved.

# How to Start

学习方法之类的过于老生常谈，在此不赘述。我们会在这里罗列ICS给出的课程资源，欢迎自行DIY适合的学习路径

## 资源汇总

### 1. 课程主页

- 课堂政策、给分细则（“学霸组”狂喜）
- 调课通知、课程安排（“罢学组”狂喜）
- 课程简介、课程组成员
- 实验指南、配置方法
- 课程资源汇总

### 2. Textbook

- 课程参考书，与课程内容互为补充（用于课前预习与课后复习）
- 计算机基础运维指南，适合自学

### 3. CLI ToolKit

- 环境配置讲座合集

### 4. 书籍推荐

- C++基础
- 系统设计与开发
- 网络系统

如果你有任何问题，欢迎随时与任课老师和助教团队联系 

## 提问的智慧

在正式开课前，我们极力推荐你先看看著名的《提问的智慧》

### 传送门（中文版）

如果你的提问并没有得到广泛的关注，可以想想其是否遵循了“提问的智慧”

# What's More

1. [Github](#)
  2. [Stack Overflow](#)
- 

© 2025. ICS Team. All rights reserved.

# Coding is All You Need

我们课程的要求本质上只有一个，那就是拥有一颗想“搞懂并深入理解计算机系统”的心

- 如果你缺乏系统背景，没关系，Lecture is great 
- 如果你缺乏代码基础，没关系，TA is here 
- 如果你缺乏运维背景，没关系，Textbook is free 

因此，我们并不需要你有多么强悍的代码背景、ACM竞赛经历，抑或是系统研究开发的经历，feel free 

当然，有肯定是最好的，但是没有也并非寸步难行

我们为你提供了非常丰富的线上、线下资源，希望你能充分利用，尽最大能力去coding，并在其中感受到系统设计之美 

---

© 2025. ICS Team. All rights reserved.

# Chapter 2 Representing and Manipulating Information

---

© 2025. ICS Team. All rights reserved.

# Chapter 2.1 Information Storage

---

Everything is bit.

---

## Introduction

欢迎来到计算机系统底层的世界，如果你只是一个初学者，这里一定~~(maybe)~~ 会颠覆你对计算机的认知，理解系统精巧的设计 (以及被各种繁琐的 historical stuff 气晕) 😎.

### What is bit?

区别于高级语言所提供的纷繁复杂的数组类型，计算机系统中所有数据都是由一个个二进制位组成，一个二进制位就是所谓的 **bit**。所有数据类型不过是对不同数量的位人为做出的不同的解释。

一般的整数可以简单的通过**位权(Weight)** 用二进制表示，而整数又可以通过对应**ASCII**码表示字符...

$114514_{10}$  二进制下表示为  $1101111101010010_2$ .

$1.1_{10}$  二进制下表示为  $1.00011[0011]\dots_2$  ( $1.919810$  二进制小数循环节太长了 😭).

### Hexadecimal

冗长的二进制位书写与阅读都极为不方便，相信你也不想在写代码时为了写一个  $1024_{10}$  而写 10 位。于是人们用**十六进制(Hexadecimal)** 来表示二进制位。

为什么用十六进制呢？用更为熟悉的十进制不是更好懂吗？~~早晚有一天你会觉得十六进制比十进制更好懂。~~

$16 = 2^4$ ，每一个十六进制位恰好对应着四个二进制位，在两者相互转化时按位直接一一对应更为方便。

十六进制、二进制与十进制三者间的相互转化相信你已经听过无数次了(高中数学、程序设计、数电、汇编...), 这里就不再赘述了。

## Byte

冗长的二进制位不仅仅不方便人类阅读, 单独的位也不方便计算机在存储系统管理。

**Bytes** : 大多数计算机采用8位的块, 又称**字节(Byte)**, 作为最小的可寻址内存单位。 **1 Byte = 8 bits.**

高级语言中常见的数据类型也以字节作为基本单位。下面给出 C 语言中的常见数据类型对应字节数的表。

| C Data Type    | Typical 32-bit | Typical 64-bit |
|----------------|----------------|----------------|
| <b>char</b>    | 1              | 1              |
| <b>short</b>   | 2              | 2              |
| <b>int</b>     | 4              | 4              |
| <b>long</b>    | 4              | 8              |
| <b>float</b>   | 4              | 4              |
| <b>double</b>  | 8              | 8              |
| <b>pointer</b> | 4              | 8              |

可以看见对大多数数据类型, 32位与64位机器所用字节数一致, 除开两个例外 long and **pointer**.

Why? 为了说明这个问题我们需要一点**虚拟内存(virtual memory)** 与**字长(word size)** 的概念。

## Virtual Memory and Word

在前文中我们提到 字节是一个最小可寻址内存单位， 但什么是内存呢？什么是寻址呢？

我们这里提到的内存是指**虚拟内存**， 是操作系统与硬件隐藏了物理内存管理的细节为程序提供的一个较低层次的**抽象**， 为程序提供一个连续、 统一的地址空间， 使得程序员无需关心底层的物理内存如何分配。

关于什么是抽象， 作为计算机科学的核心概念之一， 课程内由于课时原因没有讲解实在可惜， 为什么不尝试问问 GPT or Deepseek 计算机科学中的抽象是指什么？ 呢？

用人话来说就是我们暂时可以不用管计算机物理上是怎么复杂的存储， 可以简单的认为操作系统 (OS)为内存中的每个字节建立了像数组一样的的索引， 直接用索引号就可以找到对应的内存单位， 就像一个**字节数组**一样。这个索引的过程就是内存寻址， 而每个索引号也就对应了内存中一个字节， 一个**地址(Address)**。

这不就是C语言中的**指针(pointer)** 吗！没错， C语言中的指针就对应了字节数组的一个索引。那么指针为什么在不同机器中有不同的字节数呢？这就引出**机器字长**这一概念。

---

**机器字长(word size)**：CPU处理数据的基本单位， 即处理器一次性能处理、 存储、 传输的二进制位数。

---

我们所谓的32位、 64位机器中的32、 64指的就是机器的**机器字长**。而机器字长也是CPU支持的**最大寻址空间**， 所以在32位机器中最大寻址空间的索引就只能是32位， 那么指针变量长度为4字节， 同理64位机器则是8字节， 刚刚的疑问也就迎刃而解了。

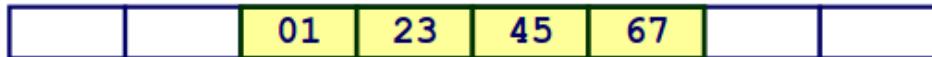
## Big-endian and Little-endian

一个整型需要4个字节， 那么字节在内存中是按什么顺序排布的呢？常见的惯例是**大端法和小端法**

下图中展示一个例子简单了解一下大端与小端。`x=0x01234567` 存储在起始地址为 `&x=0x100` 的位置上。

**Big Endian**

0x100 0x101 0x102 0x103

**Little Endian**

0x100 0x101 0x102 0x103



两者的选择几乎没有什么技术上的理由，更多是出于惯例和历史遗留。和直觉相反的是多数的新式处理器两者均可以，反倒是操作系统仅能适用一种。

## Bit-level Manipulations

### Boolean Algebra

我们假定你作为一个大二CS学生应当熟悉四种基本位运算 & | ~ ^ .

如果你对离散数学的简要介绍的代数系统还有记忆的话，那么接下来的内容想必你会非常亲切(厌恶)。

考虑四种基本位运算运算以及定长的位向量集合，可以建立布尔环，有逆元、分配律之类的性质。想学习进一步的知识，(透彻)理解布尔代数以及代数系统，*Abstract Algebra is all you need*.

### Bitwise Operations in C

注意区别位运算 & ~ 和逻辑运算即可 && !，这应该不需要我们课程科普。

一个swap小游戏：

```
void swap(int *x,int *y)
{
    (*y) = (*x) ^ (*y);
    (*x) = (*x) ^ (*y);
    (*y) = (*x) ^ (*y);
    return;
}
```

如果运行一下，你可以惊奇的发现这段代码没有定义中间变量就实现了变量交换诶 😊，实际上用加减也能达到同样的效果，这里不再列出，感兴趣的同學可以尝试一下。

## Shift Operations

1. **左移(Left Shift)**：舍弃掉左溢出的位，在空出的右侧填0。
2. **右移(Right Shift)**：舍弃掉右溢出的位，那么在空出的左侧填什么呢？和左移一样填0吗？  
NO！
  - **逻辑右移(Logical shift)**：在左侧填0，简单的规则。
  - **算术右移(Arithmetic shift)**：在左侧填**原本的最高位**。

How strange! 为什么要有特殊的算数右移呢？

正如其名，为**算数**而生，这个疑问将在有符号整数运算那一节将得到解答。

事实上 C 语言大部分实现中所有有符号整数右移均采用算数右移而无符号整数采用算数右移~~(然而 C 的标准中却没有规定这一点)~~。在 Java 中，你甚至能显式的指定右移的类型！

---

关于 Information Storage 这个话题就谈论这么多吧，我想我们已经对位有了基本的了解与熟悉。后面两节我们将讨论一个个二进制位会怎么构成我们熟悉的整数，并在运算上表现出奇妙~~(?)~~的性质。

---

© 2025. ICS Team. All rights reserved.

# Chapter 2.2 Integer Representations

大量数学表达式预警 ! ! ! ~~(但其实都是小学二年级就学过的难度)~~。

课堂上时间的限制仅仅提到位权表达式，没有运用这个式子去推导性质。而很多性质又是直接给出，没能加以证明，毕竟计算机是实践的科学，一些简单好懂一眼丁真的性质也没有必要去大费周章的证明。

但笔者认为如果学有余力，从位权、同余的角度，用数学的语言去合理表述与证明这些性质，对于理解计算机中整型表示与了解计算的底层数学原理也是颇有裨益 

本节以及下一节中式子部分来源于 CS:APP 教材，部分来源于笔者自己推导，难免会有疏漏，如果发现什么错误或者更简洁严谨的推导，欢迎在仓库的issue中提出，或者提个PR 

---

计算机中的数据是对现实世界数据的有限近似

---

## Unsigned

编码方式相对简单，直接按位权编码，可以按下面的方式形式化定义函数  $B_2 U_\omega$  (Bits to Unsigned)。

$$B_2 U_\omega(x) = \sum_{i=0}^{\omega-1} x_i \cdot 2^i$$

其中  $x$  是编码的位向量，位向量长度为  $\omega$ ， $x_i$  代表位向量第  $i$  位。

## Signed: two's-complement encode

相信经过一年半的计算机学习，你已经对编码有过多次了解，那么本文档默认你清楚原码，反码，补码的基础概念，如果在以前的课摆了如今悔过从头再来，又或是有点记不清了，不妨试试 GPT or Deepseek，相信你会得到满意的答案 

绝大多数计算机编码有符号数均采用**补码**编码，后文的重点也将落在补码上。部分同学或许觉得反码或者原码这种更为直观简单的编码或许更好理解，但从位运算以及电路的角度上看，补码更为简洁高效，一个最直观的例子就是加法。

本课程和文档不会进一步深入讨论为什么补码优于反码和原码，如果十分感兴趣，或许你在数电课上会找到答案 😊

## 补码的基本定义

在听过无数个老师讲过无数次补码以后，个人认为，用位权的方式定义**补码**最为清晰，对各种性质的证明也最为严谨。课上由于时间原因没能讲到补码自身以及其运算很多性质的证明，本文均会采用位权的方式补充证明，包括下一节中的位移、加法溢出等等。同样形式定义函数  $B_2 T_\omega$  (Bits to Two's):

$$B_2 T_\omega(x) = -x_{\omega-1} \cdot 2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i \cdot 2^i$$

其中数学符号定义与无符号数中相同，编码的主要差异在于无符号数最高位表示权重为  $+2^{\omega-1}$ ，而补码编码下最高位位权为  $-2^{\omega-1}$ 。

## 反码与原码

- 反码(One's Complement):**  $B_2 O_\omega(x) = -(x_{\omega-1} \cdot 2^{\omega-1} - 1) + \sum_{i=0}^{\omega-2} x_i \cdot 2^i$
- 原码(Sign-Magnitude):**  $B_2 S_\omega(x) = (-1)^{x_{\omega-1}} \sum_{i=0}^{\omega-2} x_i \cdot 2^i$

原码的定义比较好理解，应该和其他课接触的定义保持一致，其中反码就相对抽象了，我们第一次接触反码时的定义往往是**正数原码按位取反**，非常直观，我们这里简单证明一下两种定义等价。

记任意正数原码真值为  $Tval$ ，那么按位取反后除符号位外位权和为  $Dval = (2^{\omega-1} - 1) - Tval$ ，而考虑符号位权值，则  $Tval' = Dval - (2^{\omega-1} - 1) = -Tval$ 。两种定义形式等价。

观察反码与补码的位权定义，可以很容易的发现  $\sim x + 1 = -x$  这一性质，这也是大多数其他课程定义的补码。

进一步思考原码补码，对于一个非负数  $x$  其位向量表示为  $x$ ，我们试图定义  $-x$  的位形式。

1. 补码定义  $[00\dots0]_\omega - x$  为  $-x$  的位向量。

2. 反码定义  $[11\dots1]_\omega - x$  为  $-x$  的位向量。

有了这个理解，在整型运算时你对所谓的**模数系统**的理解也会更深入。

## Mapping Between Signed & Unsigned

### 转换方法

要将有符号数与无符号数相互转化，一种很自然的想法就是：我们不改变位向量，仅仅改变位向量的含义。那既然位向量保持不变，那么对比无符号数以及补码定义的有符号数的位权式，转化就十分显然了。

$$B_2 U_\omega(x) = x_{w-1} \cdot 2^\omega + B_2 T_\omega(x)$$

### C语言中的转换

```
int foo = -1;
unsigned bar = 1;
foo < bar == true ?
```

相信上过课的同学对这个小谜题不会太陌生，既然都问你了，答案肯定是反直觉的那个啦~~(某种意义上来说符合程序员直觉)~~。

C语言中若表达式既包含有符号数又包含无符号数，C编译器会隐式的全部转换为无符号数再执行运算。 $-1$  根据刚刚的公式转换为无符号数后应为  $2^{32} - 1$ ，那么结果自然是  $\text{foo} > \text{bar}$ 。

## Expanding and Truncating

了解了有无符号数在计算机中的相互转化，计算机中还存在不同位数的整型相互转化。比如，C 语言中的 `short`、`int`、`long` 等类型之间相互转化。

## 位表示的截断

截断的法则相对简单，先讨论无符号数的截断。

将一个  $\omega$  位的无符号数  $x$  截断为  $k$  位的  $x'$ ，直接将  $[k.. \omega - 1]$  位舍弃即可。从位权数值的角度，实际上  $x' = x \bmod 2^k$ ，因为显然有：

$$x' = \sum_{i=0}^{\omega-1} x_i \cdot 2^i \bmod 2^k = \sum_{i=0}^{k-1} x_i \cdot 2^i$$

而对于有符号数，截断方式在位向量法则是完全一致。可以转化为无符号数，按照无符号数的方法截断，再转化为有符号数，但考虑到同余关系：

$$x_{\omega-1} \cdot 2^\omega + x \equiv x(\bmod x^k)$$

所以从数值的角度考虑，直接视作无符号数取模(如果你对模运算不熟悉，或许你应该注意模运算结果恒为正数)得到新的值后再考虑最高位的权值。

这里值得注意的是如果最高位是 1 需要减去  $2^k$  而非  $2^{k-1}$ 。

## 位表示的拓展

我们希望较小类型转换为较大类型时保持其值的不变。对于无符号数法则简单，直接拓展位上填 0 就可以了。

但对于有符号数，直接简单在延伸的位上填 0 显然无法保持数值不变。应当**填充符号位的值！**

1. 当符号位为 0 时，显然与无符号数相同。
2. 当符号位为 1 时，如果从反码类似的观点来看，我们将补码意义下当符号位为 1 时的 0 才视为绝对值权值的贡献者，那么在高位填充 1 显然可以保持值不变。但由于前文一直没有提到这种观点，这里还是延续前文的位权给出一个证明。

若从  $\omega$  位长拓展到  $\omega'$  位长，显然位拓展没有影响到  $[0..\omega - 2]$  位的权值，只需考虑从  $[\omega - 1..\omega' - 1]$  位引入的权值变化，这些位在补码意义下权值和为：

$$-2^{\omega'-1} + \sum_{i=\omega-1}^{\omega'-2} 2^i = -2^{\omega-1}$$

恰好就等于原本最高位  $\omega - 1$  的权值，所以值保持不变。

---

© 2025. ICS Team. All rights reserved.

# Chapter 2.3 Integer Arithmetic

上一节我们讨论了整型的表示，这一节我们将延用上一节的风格，运用位权式去解释整型在运算中表现出的性质。

虽然仍然有语言支持无精度限制的运算，但更多的时候，运算时是限制了位数的。

## Addition

### Unsigned Addition

我们记无符号加法为  $+_{\omega}^u$ ，即  $\omega$  位限制下无符号数加法。

虽然加数仅有  $\omega$  位，但结果却可能有  $\omega + 1$  位，额外溢出的位只能被舍弃，联想上一节截断的表示，形式化的定义加法为：

$$s = x_1 +_{\omega}^u y = (x + y) \bmod 2^{\omega}$$

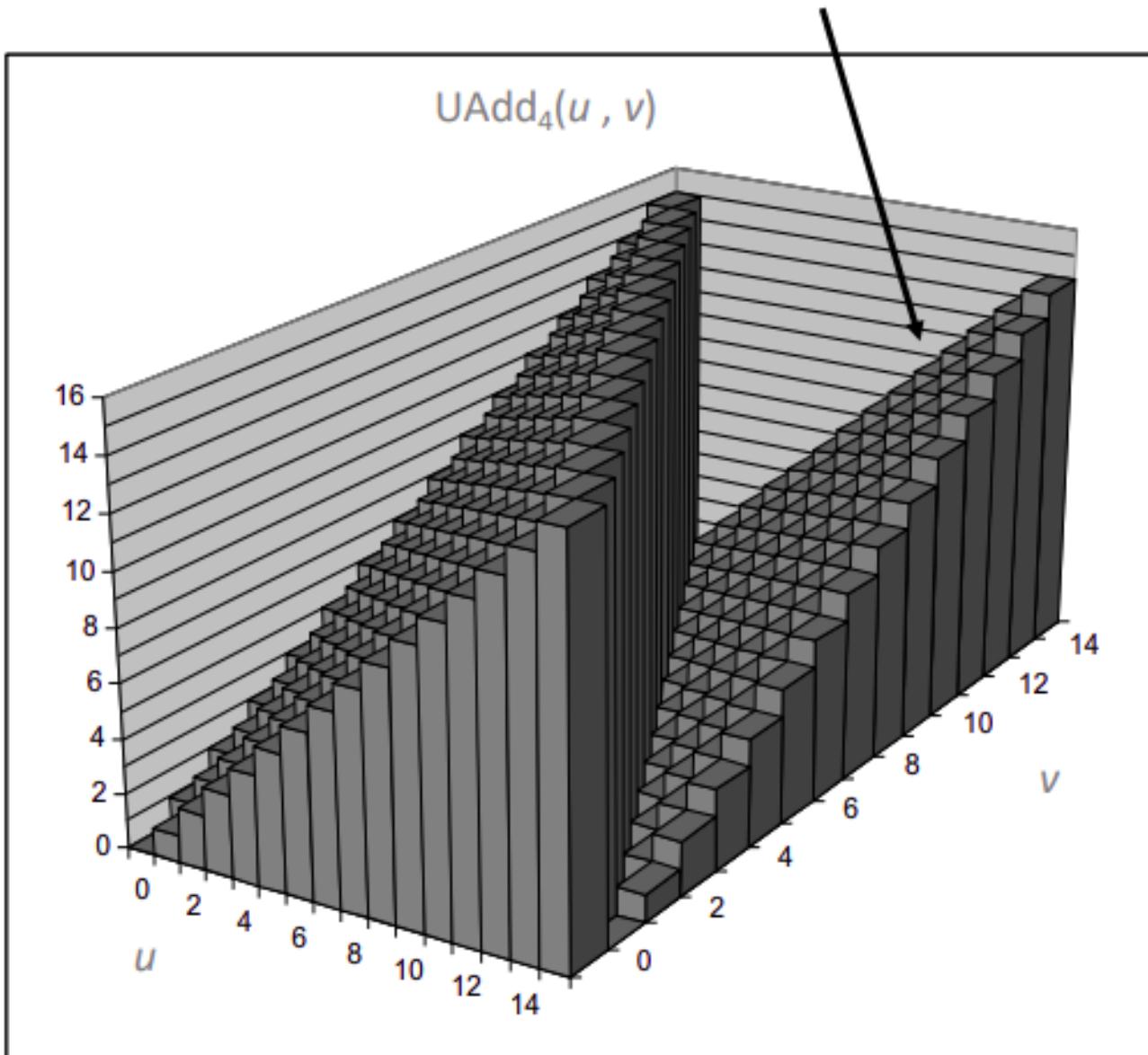
不难发现，一旦发生了溢出，必然  $2^{\omega} > x$  且  $2^{\omega} > y$  所以  $s < x$  且  $s < y$ ，可以用这个方法检验溢出。

---

模数加法实际上形成了一个阿贝尔群。

---

## Overflow



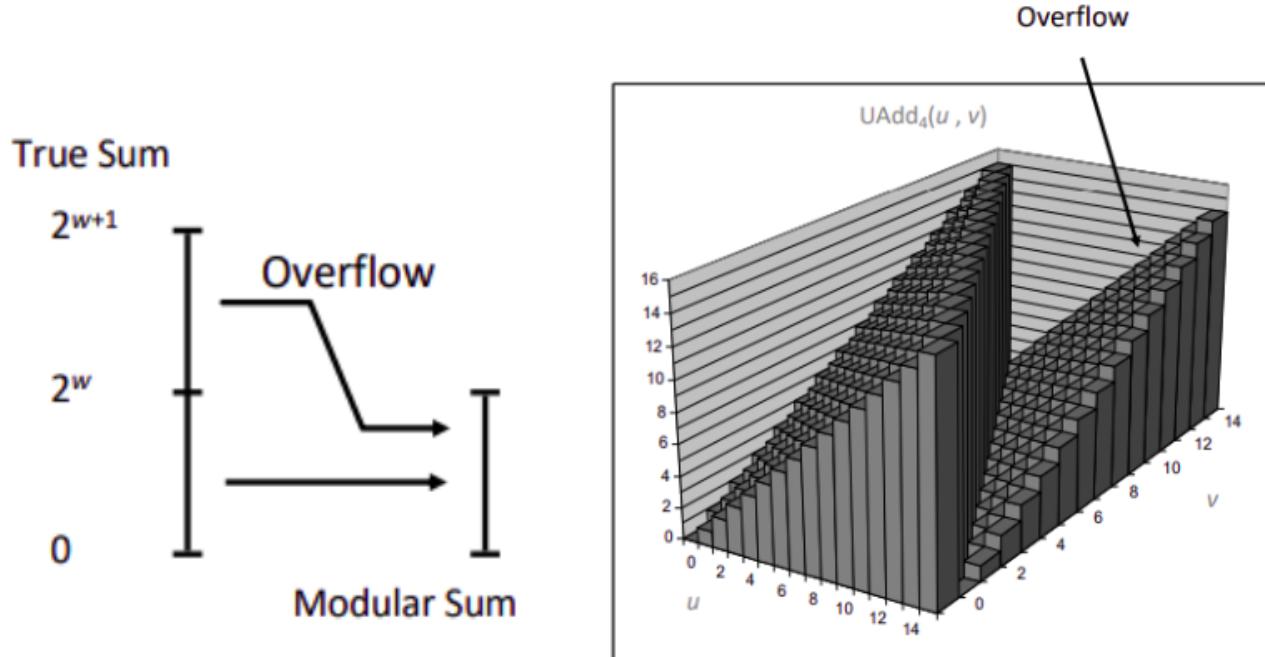
## Tow's Complement Addition

补码表示整数  $2^{\omega-1} \leq x < 2^{\omega-1}$ 。补码的加法和无符号加法在位表现上完全一致，一个自然的想法就是转化为无符号数的加法之后再对位权进行重新解释，但在数值上还不够直观，这里我们给出一组数值的形式：

$$x +_{\omega}^u y = \begin{cases} x + y - 2^{\omega}, & x + y \geq 2^{\omega-1} \\ x + y, & -2^{\omega-1} \leq x + y \leq 2^{\omega-1} \\ x + y + 2^{\omega}, & x + y \leq -2^{\omega-1} \end{cases}$$

证明也相对容易，从位权的角度：

- 当发生正溢出时，进位实际上位权从  $2 \cdot 2^{\omega-2}$  变化为  $-2^{\omega-1}$ ，产生  $-2^{\omega}$  的差值。
- 当发生负溢出时，从位权的角度也是和正溢出类似的变化，此处不再赘述。



关于补码的加法运算还有一个点值得说，显然我们只需要关注加法而不需要关注减法，当遇到减法时我们转化为加上减数的**减法逆元**就可以了。这个减法逆元我们记做  $-_{\omega}^t x$ ，区别于一般的整数中的减法逆元就是  $-x$ ，在计算机中**大多数情况下**就是  $-x$ ，但计算机取负数时实质上执行的是取补码的过程，考虑补码表示范围的不对称性，实际上有：

$$-_{\omega}^t x = \begin{cases} TMin_{\omega}, & x = TMin_{\omega} \\ -x, & x \geq TMin_{\omega} \end{cases}$$

你对  $TMin_{\omega} = 1[00..00]_{\omega-1}$  按位取反加一就是本身！即  $TMin_{\omega}$  的逆元是其本身。

## Multiplication

对于无符号数，也和加法截断想法一致，这里不再赘述，只给出表达式

$$x *_{\omega}^u y = x * y \bmod 2^{\omega}.$$

而对于补码乘法也是完全一致，由于在位表现上一致，同样第一想法是转化为补码乘在转化回去，我们还是来看看数值上的表现。现要对有符号数  $x$  和  $y$  转化为无符号数

$$x_u = x + x_{\omega-1} \cdot 2^{\omega}, y_u = y + y_{\omega-1} \cdot 2^{\omega}, \text{ 那么有:}$$

$$\begin{aligned} x_u \cdot y_u &= (x + x_{\omega-1} \cdot 2^{\omega}) \cdot (y + y_{\omega-1} \cdot 2^{\omega}) \\ &= x \cdot y + x \cdot y_{\omega-1} \cdot 2^{\omega} + x_{\omega-1} \cdot y \cdot 2^{\omega} + x_{\omega-1} \cdot y_{\omega-1} \cdot 2^{2\omega} \equiv x \cdot y (\bmod 2^{\omega}) \end{aligned}$$

所以在数值上截断意义下，转化为无符号数取模等价于直接相乘取模。再重新解释为有符号数就可以了。

## Shifting

### Left Shift

在本章第一节我们提到左移的概念，左移相对比较简单，左移  $k$  位完全等价于乘以  $2^k$ ，注意这里的乘不是现实中的乘而是无符号数或者有符号数乘~~(应该不需要提醒吧)~~。

大部分同学应该会有一个朴素的认知就是**位移和加法快于普通乘法**，所以我们可以用加法和位移得组合去代替乘法，这里也蕴含了快速乘与倍增得思想，这里就不在展开了。

总有些兄弟写代码时爱用复杂的位运算代替一般的算数运算，展现自己程序中的超高性能，曾经我也是其中一员~~(然后去不懂位运算的哥们那里装X)~~，但当我们查看汇编代码，一看傻眼了，你做的优化，聪明的编译器早就帮你做了 😎 所以，最好别这样做，在现在的编译器优化下，这种方法不会加快你的程序，但却对代码可读性有实打实的破坏。

### Right Shift

在第一节中，我们提到过右移分为逻辑与算术，对无符号数的逻辑右移我们就不谈了，相信大家都了解。

我们直接来看算术右移，PPT上有一句话非常有意思，但课上没有展开讲。

# Right Shift: $x \gg y$

Shift bit-vector  $x$  right  $y$  positions

Throw away extra bits on right

Two kinds:

“Logical”: Fill with 0’s on left

“Arithmetic”: Replicate most significant bit on left

**Almost** equivalent to dividing by  $2^y$

“**Almost**”, 可以回去看看左移时我们用的什么词:“完全等价”。为什么左移是完全等价而右移又是几乎等价呢。

显然当符号位为 0 时和逻辑右移表现是一致的, 我们只需要关注符号位为 1 时的情况就好了, 我们仅仅考虑右移 1 位时的表现, 右移  $k$  位是类似的~~(但用Markdown写公式难度却完全不一样)~~。对于最高位为 1 的有符号数  $x$  其右移一位后  $x'$  位权表达式如下:

$$x' = -2^{\omega-1} + 2^{\omega-2} + \sum_{i=1}^{\omega-2} x_i \cdot 2^{i-1} = -2^{\omega-2} + \sum_{i=1}^{\omega-2} x_i \cdot 2^{i-1}$$

对比  $x$  的位权式:

$$x = -2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i \cdot 2^i$$

不难发现  $x' = \lfloor x/2 \rfloor$ , 那么 `almost` 在哪儿我请问了? 诶这熟悉 C 语言的同学可能就发现不同

了，C语言中的除法应当是**向零取整的**，也就是说正数应当向下取整符合右移，但负数应当**向上取整**，但右移对于负数而言是向下取整。至于应当怎样才能让负数右移也向零取整呢？

这里提一个名词**舍入**，感兴趣的同學可以去了解一下，本文就不再赘述了，这里只是为了提醒一下大家这除法与右移这两种行为的不同，并解释一下PPT上的 almost，对这个问题的讨论就在这里终止了~~(因为笔者已经写公式写吐了)~~。

---

那么这一节的主要內容就到此为止了，也是本章的结尾。这一章从第两节起，开始主要运用位权的方式，从数学上证明了很多同学们~~耳熟能详~~的性质，提供一个用数学更为严谨的方法研究计算机整数表示的视角。应该是涵盖课筆者认为有趣或值得一证的性质，希望能对你理解整型的表示有所帮助。

最后提供一个C语言小謎題供大家思考：

```
int is_overflow_add(int x,int y)
{
    int sum = x + y;
    return sum - x == y;
}
// 这个小函数能完成检验是否溢出的目的吗？

int is_overflow_mul(int x,int y)
{
    int mul = x * y;
    return abs(mul / x - y) > 1e-5;
}
// 这个小函数能完成检验是否溢出的目的吗？
```

---

© 2025. ICS Team. All rights reserved.

# Chapter 3 Machine-Level Representation of Programs

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.1 A Historical Perspective

## Introduction

本章将带领你进入计算机底层的世界，从机器层面了解我们编写的程序，近距离地观察具有可读性的机器代码——汇编代码。😍

什么？开始头晕眼花了？相信我，学完本章，你就能成为一名优秀的拆弹专家！Bomb lab is waiting for you~

## Intel x86 Processor

首先，让我们先从历史的角度，了解一下我们手中计算机的处理器的发展沿革，通过历史激起你学习汇编语言的动力。😎

Intel处理器系列俗称x86，它经历了一个长期的、不断进化的发展过程。

1978年，Intel推出具有划时代意义的8086微处理器，它是第一代单芯片、16位微处理器之一。一年后简化版的8088推出，它在8086的基础上支持8位数据总线，IBM公司1981年生产的第一台电脑使用的就是这种芯片。这也标志着x86架构和IBM PC 兼容电脑的产生。此后，x86不断地成长，利用进步的技术满足更高性能和支持更高级操作系统的需求。

下表展示了Intel处理器体系发展过程中的几个里程碑：

| Name       | Evolution                                | Date | Transistors | MHz       |
|------------|--|------|-------------|-----------|
| 8086       | First 16-bit Intel processor             | 1978 | 29K         | 5-10      |
| 386        | First 32-bit Intel processor(IA32)       | 1985 | 275K        | 16-33     |
| Pentium 4E | First 64-bit Intel x86 processor(x86-64) | 2004 | 125M        | 2800-3800 |
| Core 2     | First multicore Intel processor          | 2006 | 291M        | 1060-3333 |
| Core i7    | Four cores                               | 2008 | 731M        | 1600-     |

可以看到，Intel处理器在近五十年的发展中，晶体管数量大约以每年37%的速率增加。虽然不像Intel创始人Gordon Moore想象的“晶体管数量每年翻一番”那样迅猛，不过在超过五十年中，半导体工业一直能够使晶体管数目每18个月翻一倍。

同时，在发展中，Intel处理器还增加了更多更强大的特性，体系结构从16位扩展到32位乃至如今的64位，核数也由单核转向多核，并且支持多媒体操作和高效的条件指令。

一个非常人性化的设计是，为了便于用户（苦命的程序员）使用，每个后继处理器的设计都是**后向兼容(Backwards compatibility)**的，即较早版本上编译的代码可以在较新的处理器上运行。这使得Intel使用的x86架构指令集中有多种格式的许多不同指令，它也因此被反对者诟病为**CISC(Complex Instruction Set Computer)**。

与之相对的是**RISC(Reduced Instruction Set Computer)**，主张"very few instructions, with very few modes for each"，优点是高效率和低功耗。但由于兼容性的问题，很长时间以来仍然是Intel的CISC在市场中占据优势。

不过近年来，由于移动时代用户对于低能耗的需求，RISC又流行起来，在移动设备和嵌入式系统中得到了广泛应用。

## x86 Clones: Advanced Micro Devices (AMD)

既然讲了Intel的发展，那么我们就不得不提到它的死对头——AMD。😂

数年来，AMD一直充当Intel的小弟，在技术上紧随Intel，执行的市场策略是：生产性能稍低但价格更便宜的处理器。

1996年，AMD收购了芯片设计公司NexGen，之后推出K6处理器以及迭代产品K6-2、K6-3，以产品廉价和高性价比抢占了极大市场份额，打破了Intel在处理器市场的垄断局面，同时进入笔记本市场对Intel进行挑战。

2002年，AMD率先突破了可商用微处理器的1GHz的时钟速度屏障，并引用了IA32的64位扩展x86-64。2003年推出皓龙（Opteron）服务器处理器，又为AMD打开了部分服务器市场份额。此后AMD微处理器市场份额持续上升，成为Intel的强劲对手。

不过故事并没有结束，在AMD收购了合作伙伴NVIDIA在GPU领域的死对头ATI之后，NVIDIA一怒

之下转向Intel，而Intel也在2006年推出新一代处理器Core 2以及4核CPU，重新占据主导地位。

近年来，TSMC成为世界领先的半导体晶圆厂，Intel再次落后。

2017年3月，AMD以Zen架构为核心的锐龙（Ryzen）系列处理器正式发行，迅速抢占CPU市场份额。此后，AMD在CPU和GPU市场双线作战，Zen架构的持续迭代和性能进步显著，在纸面参数上超越英特尔十代酷睿处理器，动摇了英特尔在CPU市场的长期霸权。

AMD和Intel的大型商战还在继续.....

## Our Coverage

本课程仅采用x86-64架构进行描述。参考教材CS:APP3e的Web Asides有对IA-32编程的介绍。

### Info

IA-32(Intel Architecture 32-bit): 是一个32位架构，使用传统的 x86 指令集。

x86-64: 在 x86 指令集的基础上进行了扩展，增加了新的指令以支持 64 位操作。x86-64 架构仍然可以执行 IA-32 的指令，这保证了向后兼容性。

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.2 Program Encodings

## Machine-level Code

讲到机器级代码，我们就必须了解抽象的概念。抽象（Abstraction）是简化复杂的现实问题的途径，是计算机科学中最为重要的概念之一。比如我们为一组函数规定一个简单的应用程序接口（API）就是一个很好的编程习惯，程序员无需了解它内部的工作便可以使用这些代码。这就是抽象的作用。

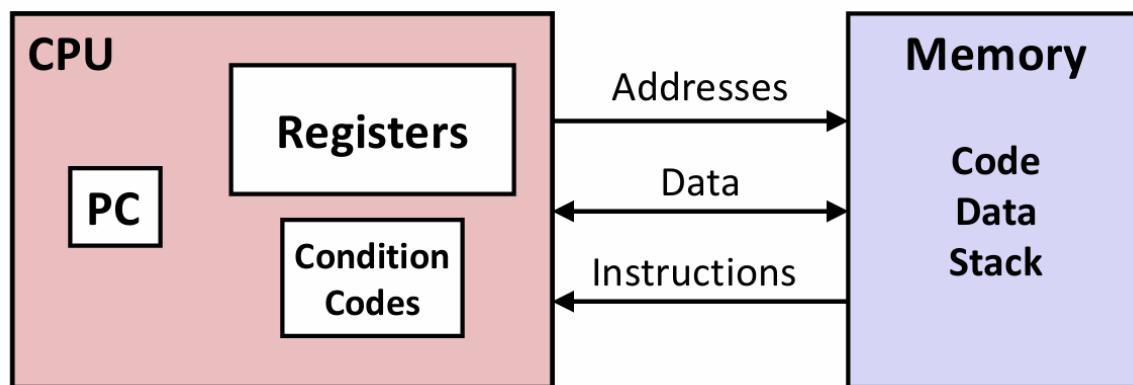
计算机系统使用了多种不同形式的抽象，利用更简单的抽象模型来隐藏实现的细节。包括指令集架构抽象，微体系结构抽象，逻辑计算抽象（想必大家在数电的学习中对此已非常熟悉了~）等。

对于机器级编程来说，以下两种抽象尤为重要：

1. **指令集体系架构或指令集架构**（Instruction Set Architecture, ISA）：用来定义机器级程序的格式和行为。它定义了处理器状态、指令的格式，以及每条指令对状态的影响。
2. **虚拟内存**：机器级程序使用的内存地址是虚拟的，提供的内存模型看上去是一个非常大的字节数组，使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间）。

## Programmer-Visible State of Processor

x86-64的机器代码和原始的C代码差别非常大，程序员通常在C代码中看不到的处理器状态在机器代码中都是可见的：



## Assembly/Machine Code View

- **程序计数器 (Program Counter, PC)** ,在x86-64中用%rip(~~rest in peace~~ 😊)表示, 给出将要执行的下一条指令在内存中的地址。
- **整数寄存器文件 (Register file)** : 包含16个命名的位置, 分别存储64位的值。这些寄存器可以存储地址或整数数据。作用: 记录程序状态; 保存临时数据。
- **条件码 (Condition Codes) 寄存器** : 保存最近执行的算术或逻辑指令的状态信息, 用于实现控制或数据流中的条件变化, 如if和while语句。
- 一组向量寄存器可以存放一个或多个整数或浮点数值。

## Code Example

### 1. Compile Into Assembly

假设我们写了一个C语言代码文件mstore.c, 包含如下函数定义:

```
long plus(long x, long y);  
  
void sumstore(long x, long y, long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

在命令行使用“-S”选项:

```
linux> gcc -Og -S mstore.c
```

这会使GCC运行**编译器**, 产生一个汇编文件mstore.s:

```
sumstore:  
    pushq  %rbx  
    movq  %rdx, %rbx  
    call  plus  
    movq  %rax, (%rbx)  
    popq  %rbx  
    ret
```

上面代码中每一个缩进都对应一段机器指令。比如，pushq指令表示将寄存器%rbx的内容压入程序栈中。

事实上，这段代码是简化版的，剔除了很多我们不太关心也看不懂的信息。

Actually, this is what it really looks like...

```
.globl sumstore  
.type sumstore, @function  
sumstore:  
.LFB35:  
.cfi_startproc  
pushq %rbx  
.cfi_def_cfa_offset 16  
.cfi_offset 3, -16  
movq %rdx, %rbx  
call plus  
movq %rax, (%rbx)  
popq %rbx  
.cfi_def_cfa_offset 8  
ret  
.cfi_endproc  
.LFE35:  
.size sumstore, .-sumstore
```

是不是看花眼了？别担心，事实上所有以'.'开头的行但是指导汇编器和链接器工作的伪指令，看的时候可以忽略~

## 2. Generate Object Code

使用"-c"命令行选项：

```
linux> gcc -Og -c mstore.c
```

此时GCC会**编译并汇编**该代码，产生目标代码文件mstore.o，它是二进制格式的，上面汇编代码对应的目标代码序列如下：

```
0x0400595:  
0x53  
0x48  
0x89  
0xd3  
0xe8  
0xf2  
0xff  
0xff  
0xff  
0x48  
0x89  
0x03  
0x5b  
0xc3
```

从中我们再次印证了机器执行的程序只是一个字节序列，它对于产生这些指令的源代码几乎一无所知。

要查看机器代码文件的内容，有一类称为**反汇编器**（Disassembler）的程序非常有用。这些程序根据机器代码产生一种类似于汇编代码的格式。在Linux系统中，带-d命令行标志的程序OBJDUMP可以充当这个角色：

```
Linux> objdump -d mstore.o
```

结果如下：

```
Disassembly of function sum in binary file mstore.o
1 0000000000000000 <multstore>:
Offset   Bytes           Equivalent assembly language
2     0: 53               push    %rbx
3     1: 48 89 d3         mov     %rdx,%rbx
4     4: e8 00 00 00 00    callq   9 <multstore+0x9>
5     9: 48 89 03         mov     %rax,(%rbx)
6     c: 5b               pop    %rbx
7     d: c3               retq
```

### Info

反汇编器只是基于机器代码文件中的字节序列来确定汇编代码，它不需要访问该程序的源代码或汇编代码。

反汇编器使用的指令命名规则与GCC生成的汇编代码有细微差别。在上面的示例中，它省略了很多指令结尾的'q'（我们都知道'q'代表"quad"，指的是4个字即8个字节），因为是大小指示符，在大多数情况下可以省略。而反汇编器又给call和ret指令添加了'q'后缀，同样，省略这些后缀也没问题。

## 3. Generate the Executable File

生成可执行的代码需要对一组目标代码文件运行链接器，而这一组目标代码文件中必须有一个main函数。假设在文件main.c中有如下函数：

```
#include<stdio.h>

void multstore(long, long, long*);

int main(){
    long d;
    multstore(2, 3, &d);
    printf("2*3-->%ld\n", d);
    return 0;
}

long mult2(long a, long b){
    long s=a*b;
    return s;
}
```

然后我们用如下方法生成可执行文件prog：

```
linux>gcc -Og -o progmain cmstore.c
```

链接是在程序开始执行时发生的。其实链接器的工作主要就是解析文件之间的引用，与静态运行时库结合（例如，malloc、printf的代码）；也有一些库是动态链接的。

下面概括了一般的C程序编码过程：

假设一个C程序有两个文件p1.c和p2.c。用Unix命令行编译这些代码。

```
linux> gcc -Og -o p p1.c p2.c
```

gcc调用了一整套的程序，将源代码转化成可执行代码：

1. 首先，**C预处理器**（Preprocessor）扩展源代码，插入所有用 `#include` 命令指定的文件，并扩展所有用 `#define` 声明指定的宏。
2. 其次，**编译器**（Compiler）产生两个源文件的汇编代码p1.s和p2.s。
3. 接下来，**汇编器**（Assembler）将汇编代码转化成二进制目标代码文件p1.o和p2.o。
4. 最后，**链接器**（Linker）将两个目标代码文件与实现库函数的代码合并，产生最终的可执行代码文件p。

以上就是C文件被转化为可执行文件的全过程，相信大家对于程序编码已经有了一个较为透彻的理

解。

btw, 还要提醒大家一个关于汇编代码格式的问题。

本课程的表述是ATT（根据“AT&T”命名的）格式的汇编代码，这是GCC、OBJDUMP和其他一些我们使用的工具的默认格式。

而Microsoft的工具和来自Intel的文档，其汇编代码的格式是Intel格式的。

以下是它们二者的对比：

|                        | <b>Direction of Operands</b> | <b>Memory Operands</b> | <b>Prefixes</b> | <b>Suffixes</b>  | <b>Register Name</b> |
|------------------------|------------------------------|------------------------|-----------------|------------------|----------------------|
| <b>Intel Syntax</b>    | instr dest, src              | [rbx]                  | 1               | mov              | rbx                  |
| <b>AT&amp;T Syntax</b> | instr src, dest              | (%rbx)                 | \$1             | movq,<br>movl... | %rbx                 |

区别：

1. Intel是先destination后source,而ATT是先source后destination。
2. Intel省略寄存器名字前的‘%’符号。
3. Intel立即数无前缀‘\$’，指令的大小指示后缀也被省略。
4. Intel描述内存位置用的是[]，而ATT用的是()。

# Chapter 3.3 Data Formats

在上一节中，我们已经接触到了一些机器指令中的数据格式，本节将具体介绍C语言数据类型对应的x86-64表示。

首先我们要明确一个观点，机器只是简单地将内存看成一个按字节寻址的巨大数组，C语言中的聚合数据类型，如数组和结构体，在机器代码中也是用一组连续的字节表示。即使是对标量数据类型，汇编代码也不区分有符号数或无符号数，不区分各种类型的指针，甚至不区分指针和整数。

所以在汇编代码中，所有的数据类型都是“Integer”，也就是没有类型之分，而只有大小之分，如1个字节，2个字节，4个字节和8个字节。

下面是具体的C语言数据类型在汇编中的对应表：

| C declaration | Intel data type  | Assembly-code suffix | Size (bytes) |
|---------------|------------------|----------------------|--------------|
| char          | Byte             | b                    | 1            |
| short         | Word             | w                    | 2            |
| int           | Double word      | l                    | 4            |
| long          | Quad word        | q                    | 8            |
| char *        | Quad word        | q                    | 8            |
| float         | Single precision | s                    | 4            |
| double        | Double precision | l                    | 8            |

在上一节中，我们已经向大家介绍了‘q’后缀的含义，此处又出现了如b, w, l的后缀，这也许让你有些迷惑。

实际上，产生这样的后缀名是有历史根源的。由于是从16位体系结构扩展而来的（还记得那个具有划时代意义的8086吗？它就是第一代16位处理器），Intel用术语“字（word）”表示16位数据类型。因此，称32位数为“双字（double words）”，称64位数为“四字（quad words）”。

而不同位数对应的数据传送指令也有四种：movb（传送字节，byte）、movw（传送字，word）、movl（传送长字，long word）和movq（传送四字）。

本课程不涉及浮点数，故不予讨论。有兴趣的同学可以参考[CS:APP](#)进行学习。

# Chapter 3.4 Accessing Information

## Register

前文中我们提到过寄存器这个对大部分同学十分陌生的名词，因为同学们经常接触的抽象程度较高的高级语言中，寄存器已经被隐藏不再可见，但在机器级代码中，寄存器却是表示数据的重要一环，我们有必要深入了解一下它。

一个 x86-64 的 CPU 包含一组 16 个存储 64 位值的通用目的寄存器，用于存储整数数据以及指针，他们和 CPU 靠的很近，访问速度极快，当然数量也十分有限。下图中展示了这 16 个寄存器。

|      |       |
|------|-------|
| %rax | %eax  |
| %rbx | %ebx  |
| %rcx | %ecx  |
| %rdx | %edx  |
| %rsi | %esi  |
| %rdi | %edi  |
| %rsp | %esp  |
| %rbp | %ebp  |
| %r8  | %r8d  |
| %r9  | %r9d  |
| %r10 | %r10d |
| %r11 | %r11d |
| %r12 | %r12d |
| %r13 | %r13d |
| %r14 | %r14d |
| %r15 | %r15d |

可以看到这 16 个寄存器他们的名字都是以 **%r** 开头，不过后面还跟着一些不同命名规则的名字。在历史上每个寄存器都有特殊的用途，他们的名字反应了这些特殊的用途，但后来不再有这个约束。随着时代发展，寄存器从最初的 8 位一步步变为 16 位、32 位，再到今天的 64 位，图中以 **%e** 命名开头的寄存器就是 32 位寄存器。

还应当注意到的是其实 CPU 中并不是只有 16 个寄存器，还有许多寄存器但从机器级代码的角度不可见，被保留给 CPU 进行一些硬件的实现。

# Instruction

## 操作数

在介绍一些基本的指令之前，我们需要先知道数据在机器语言中的表示，我们称之为**操作数 (operand)**。大体上来说总共有三类：

1. **立即数**：用来表示常数值，立即数的书写规范是一个 '\$' 符号后面跟一个用 C 标准表示的常数值。如果不是十进制表示，需要指出进制，比如 \$0x17 就表示十六进制下的 (17)<sub>16</sub>。
2. **寄存器**：前文介绍的寄存器，直接用它的名字表示就可以了 %rsp 就表示前文表中的 %rsp 寄存器。
3. **内存引用**：它会根据计算出来的地址，访问某个内存的位置。一般而言我们用 () 来表明这是一个内存地址的表示，比如 (%rsp) 就表示将寄存器 %rsp 中的值视为内存地址，访问那个内存地址。

## 数据传送指令

我们将主要介绍 movq 这个指令，这个指令需要两个操作数，分别表示数据源和数据目的，这个指令也正如其名，将源的数据复制一份到目的当中，在前几节中我们介绍过后缀 q，表示移动的是 64 位数据。

指令格式: **movq Source, Dest**，注意源在第一个操作数位，目的在第二操作数位。这一点和汇编课上讲的的 80x86 格式的汇编相反，同时上两节课的同学注意区分，别混淆了。

这里的 Source 与 Dest 的对应上文中讲的操作数，比如我们可以把一个立即数移入寄存器中，或者内存当中。但显然我们不能把寄存器的值移入立即数中。看到这你应该可以意识到，目的与源应当有些限制，具体限制如下图所示。

|             | Source     | Dest       | Src,Dest                  | C Analog              |
|-------------|------------|------------|---------------------------|-----------------------|
| <b>movq</b> | <b>Imm</b> | <b>Reg</b> | <b>movq \$0x4,%rax</b>    | <b>temp = 0x4;</b>    |
|             | <b>Imm</b> | <b>Mem</b> | <b>movq \$-147,(%rax)</b> | <b>*p = -147;</b>     |
|             | <b>Reg</b> | <b>Reg</b> | <b>movq %rax,%rdx</b>     | <b>temp2 = temp1;</b> |
|             | <b>Reg</b> | <b>Mem</b> | <b>movq %rax,(%rdx)</b>   | <b>*p = temp;</b>     |
|             | <b>Mem</b> | <b>Reg</b> | <b>movq (%rax),%rdx</b>   | <b>temp = *p;</b>     |

可以看到除了不能移入立即数的限制以外，只有一种组合受到限制，即源和目的都是内存也是不可行的，这主要是为了执行效率考虑而设计的，大部分同学经过那么多节课的熏陶，应该都有一个基本的认知：从内存中读取、存储数据都十分慢。那么一条指令既要从内存中读，又要写入内存，过于慢了，干脆拆成多条指令来执行。

图中还展示了汇编指令对应的 C 中的操作，将寄存器理解为局部变量，内存引用理解为指针确实是不错的理解方式。

## 寻址

我们先前的例子中，内存地址的表示都相对简单，但往往我们会需要更为复杂的内存表示模式，方便我们更好的定位，下面由简单到复杂介绍地址的表示：

1. **Normal: (R) -> Mem[Reg[R]]**，最为简单的表示方式，前文也已经解释过了，不再赘述。
2. **Displacement: D(R) -> Mem[Reg[R]+D]**，即在寄存器的值的基础上加上一个偏移量 D，这个 D 用立即数表述。这个设计主要方便我们访问结构体中的成员。例如我们想要一个结构体中的第二个成员，已知这个结构体的起始地址存在 %rbp 中，而结构体的第一个成员占了八字节（比如是一个 long），那么我们需要的目的成员的地址就可以表示为 8(%rbp)，可以用 movq 8(%rbp),%rdx 将它取出。
3. **Most General Form D(Rb,Ri,S) -> Mem[Reg[Rb]+S\*Reg[Ri]+D]**，其中 D 和第二条中一样是偏移量，而其中的另外几个符号我们还是通过 C 中的概念来类比解释：
  - Rb, **base register**, 一个结构体数组的起始地址。
  - Ri, **index register**, 结构体数组的索引，我们要找第 Reg[Ri] 个元素。

- S, **scale**, 比例因子, 这个结构体数组每个成员占的字节数。注意 S 的取值只能是 1, 2, 4, or 8.

有了最通用的形式, 可以发现前两种形式不过是省掉了一些元素的形式罢了。这个一般形式中除了 Rb 不能省去以外其他部分都可以合乎逻辑的组合。

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.5 Arithmetic and Logical Operations

在一小节我们讲了机器级程序中数据的转移，这一节我们进一步深入，了解基本的算术逻辑操作。

## Address Computation Instruction

我们先延续上一节的内容介绍一个特殊的地址计算指令 **leaq**，和 **movq** 极其相同。

格式为 **leaq Src, Dst**，其中的 **Src** 是一个地址表达式，它的形式应当符合上一节结尾介绍的地址表达式的要求，而 **leaq** 指令会将源中计算出的地址的**地址值**赋给目的。区别于 **movq**，它会将地址指向的内存值取出赋给目的。

这条指令的设计最初的目的就是快速计算地址的值，然而后来的编译器发现这条指令计算值是不是地址又有什么关系呢，反正计算出了一个值，就可以用这个指令来进行一些运算。比如：

```
long m12(long x)
{
    return x*12;
}
```

会被编译器翻译为：

```
leaq (%rdi,%rdi,2), %rax # t=x+2*x
salq $2, %rax           # return t<<2
```

这种诡异的设计令人摸不着头脑，可能来自于设计师下午茶时间的灵光一动，或者是早上没睡醒时的胡言乱语，但已经无从考证，anyway，总之习惯就好。这条指令是算数运算中用的比较多的指令，要能理解是什么意思。

## Some Arithmetic Operations

我们在此列举出常见的运算，这张表也不用背，大部分看名字一眼就知道什么意思，考试也会给出

一张表告诉你每条指令是什么。

先是常见的二元运算：

| Format | Computation                      |
|--------|----------------------------------|
| addq   | Src,Dest      Dest = Dest + Src  |
| subq   | Src,Dest      Dest = Dest – Src  |
| imulq  | Src,Dest      Dest = Dest * Src  |
| salq   | Src,Dest      Dest = Dest << Src |
| sarq   | Src,Dest      Dest = Dest >> Src |
| shrq   | Src,Dest      Dest = Dest >> Src |
| xorq   | Src,Dest      Dest = Dest ^ Src  |
| andq   | Src,Dest      Dest = Dest & Src  |
| orq    | Src,Dest      Dest = Dest   Src  |

要强调的是一定要注意对于二元运算两个操作数之间的顺序，尤其是减法比较反人类的直觉，一定是目的减去源的值存储在目的中。

还需要注意的是我们的加或乘运算没有区别有无符号，可以回想一下我们在第二章运算中强调的，无符号数与补码在运算上的一致性。

接下来是常见的一元运算：

|      |      |                 |
|------|------|-----------------|
| incq | Dest | Dest = Dest + 1 |
| decq | Dest | Dest = Dest - 1 |
| negq | Dest | Dest = - Dest   |
| notq | Dest | Dest = ~Dest    |

看到了 C 中的 `++i`，无内鬼，来点谭浩强教材笑话。所以 `(++i++) + ((++i)++) + (i++)++` 到底等于多少呢？

## Example

本课程要求的对算术逻辑运算的知识并不复杂，我们举个简单的例子来熟悉一下一些简单的指令。

看如下一个程序：

```
long arith(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

对应每条一一翻译为汇编代码如下：

```
arith:  
    leaq    (%rdi,%rsi), %rax      # t1  
    addq    %rdx, %rax            # t2  
    leaq    (%rsi,%rsi,2), %rdx  
    salq    $4, %rdx              # t4  
    leaq    4(%rdi,%rdx), %rcx  
    imulq   %rcx, %rax           # rval  
    ret
```

我们也在汇编码中简单标注了计算的值对应C中的变量，对照着看相信比较容易可以看懂。

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.6 Control

到目前为止，我们只考虑了直线代码的行为，也就是指令一条接着一条顺序地执行。然而C语言中有很多结构要求有条件的执行，即根据数据测试的结果来决定操作执行的顺序，比如条件语句、循环语句和分支语句。

本节将涉及实现条件操作的两种方式，并描述表达循环和switch语句的方法。

## Condition Code

除了整数寄存器，CPU还维护着一组单个位的**条件码**（Condition Code）寄存器，它们描述了最近的算术或逻辑操作的属性。最常用的条件码有：

- CF (Carry Flag) (for unsigned)：进位标志。最近的操作使最高位产生了进位。可用来检查无符号操作的溢出。
- ZF (Zero Flag)：零标志。最近的操作得出的结果为0。
- SF (Sign Flag) (for signed)：符号标志。最近的操作得到的结果为负数。
- OF (Overflow Flag) (for signed)：溢出标志。最近的操作导致一个补码溢出（正溢出或负溢出）。

For example :

加法运算 addq Src,Dest  $\leftrightarrow t = a+b$

| 条件码 | C 表达式                       | 说明    |
|-----|-----------------------------|-------|
| CF  | (unsigned) t < (unsigned) a | 无符号溢出 |
| ZF  | (t == 0)                    | 零     |
| SF  | (t < 0)                     | 负数    |
| OF  | (a<0==b<0) && (t<0 != a<0)  | 有符号溢出 |

除了leaq指令（用于进行地址计算，不改变任何条件码），我们前面学习的所有的整数算术操作的指令都会设置条件码。

另外，还有两类指令CMP和TEST，它们只设置条件码而不改变其他寄存器。

- CMP指令与SUB指令的行为一样，根据两个操作数之差来设置条件码，只是不更新目的寄存器。通常用于 `if(a<b){...}`。
- TEST指令与AND指令的行为一样，同样是只设置条件码（只有ZF和SF，ZF Set when  $a \& b == 0$ , SF Set when  $a \& b < 0$ ）而不改变目的寄存器的值。典型的用法是 `testq %rax, %rax` 用来检查%rax和零的大小关系。

## Access Condition Code

条件码通常不会直接读取，常用的使用方法有三种：

- 可以根据条件码的某种组合，将一个字节设置为0或1。
- 可以条件跳转到程序的某个其他的部分。
- 可以有条件地传送数据。

对于第一种情况，我们用SET指令来实现：

| <b>SetX</b>  | <b>Condition</b>   | <b>Description</b>               |
|--------------|--|----------------------------------|
| <b>sete</b>  | <b>ZF</b>  | <b>Equal / Zero</b>              |
| <b>setne</b> | <b><math>\sim ZF</math></b>                                | <b>Not Equal / Not Zero</b>      |
| <b>sets</b>  | <b>SF</b>  | <b>Negative</b>                  |
| <b>setns</b> | <b><math>\sim SF</math></b>                                | <b>Nonnegative</b>               |
| <b>setg</b>  | <b><math>\sim (SF \wedge OF) \ \&amp; \ \sim ZF</math></b> | <b>Greater (Signed)</b>          |
| <b>setge</b> | <b><math>\sim (SF \wedge OF)</math></b>                    | <b>Greater or Equal (Signed)</b> |
| <b>setl</b>  | <b><math>(SF \wedge OF)</math></b>                         | <b>Less (Signed)</b>             |
| <b>setle</b> | <b><math>(SF \wedge OF) \mid ZF</math></b>                 | <b>Less or Equal (Signed)</b>    |
| <b>seta</b>  | <b><math>\sim CF \ \&amp; \ \sim ZF</math></b>             | <b>Above (unsigned)</b>          |
| <b>setb</b>  | <b>CF</b>  | <b>Below (unsigned)</b>          |

这些指令的后缀不再是操作数的大小，而表示不同的条件。例如setl和setb分别表示“小于时设置 (set less, signed) ”和“低于时设置 (set below, unsigned) ”，切勿混淆。同时要注意有符号数和无符号数在相同指令下的不同后缀。

一条SET指令的目的操作数是低位单字节寄存器元素之一(还记得那个巨大的寄存器表格吗😎, 没错, 这里指的是它的最右边那一列, 如%al, %r8b, etc.), 指令会把这个字节设置成0或1。为了得到一个32位或64位结果, 必须对高位清零。

我们用一个简单的C语言表达式 $x < y$ 来说明。

```
int gt (long x, long y)
{
    return x > y;
}
```

对应汇编指令：

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al            # Set when >
movzbl %al, %eax      # Zero rest of %rax
ret
```

其中%rdi存放x, %rsi存放y, %rax存放返回值。

注意比较的次序 (cmp src, dest) ! (再次强调啊🤣)

还有一个比较奇怪的地方需要注意, 这里movbl指令不仅把%eax的高3个字节清零, 还会把整个寄存器%rax的高4个字节都清零。这是因为x86-64的惯例是任何为寄存器生成32位值的指令都会把该寄存器的高位部分置为0。

## Jump Instructions

正常执行的情况下, 指令按照它们出现的顺序一条一条地执行。跳转 (jump) 指令会导致执行切换到程序中一个全新的位置。在汇编代码中, 这些跳转的目的地用一个标号 (label) 指明。

示例 (人为编造的, 只是为了展示其用法) :

```
movq $0,%rax          #Set %rax to 0
jmp .L1                #Goto .L1
movq (%rax),%rdx      #Null pointer dereference (skipped)
.L1:
    popq %rdx          #Jump target
```

指令jmp .L1会导致程序跳过movq指令，而从popq指令开始继续执行。在产生目标代码文件时，汇编器会确定所有带标号指令的地址，并将**跳转目标**（目的指令的地址）编码为跳转指令的一部分。

下表列出了所有jump指令。这些指令的名字和跳转条件与SET指令相匹配。

| jX  | Condition      | Description               |
|-----|----------------|---------------------------|
| jmp | 1              | Unconditional             |
| je  | ZF             | Equal / Zero              |
| jne | ~ZF            | Not Equal / Not Zero      |
| js  | SF             | Negative                  |
| jns | ~SF            | Nonnegative               |
| jg  | ~(SF^OF) & ~ZF | Greater (Signed)          |
| jge | ~(SF^OF)       | Greater or Equal (Signed) |
| jl  | (SF^OF)        | Less (Signed)             |
| jle | (SF^OF)   ZF   | Less or Equal (Signed)    |
| ja  | ~CF & ~ZF      | Above (unsigned)          |
| jb  | CF             | Below (unsigned)          |

## Conditional Moves

实现条件操作的传统方法是通过使用**控制**的条件转移。当条件满足时，程序沿着一条执行路径执行；而当条件不满足时，就走另一条路径。这种机制虽然简单，但在现代处理器上可能会非常低效。

一种替代的策略是使用**数据**的条件转移。这种方法计算一个条件操作的两种结果，然后再根据条件是否满足从中选取一个。这样就可以用一条简单的条件传送指令来实现它，更符合现代处理器的性能特性。

我们以一个例子说明：

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

产生的汇编代码：

```
absdiff:
movq    %rdi, %rax    # x
subq    %rsi, %rax    # result = x-y
movq    %rsi, %rdx
subq    %rdi, %rdx    # eval = y-x
cmpq    %rsi, %rdi    # x:y
cmovle %rdx, %rax    # if <=, result = eval
ret
```

我们可以看到在汇编中，既计算了 $x-y$ 的值，也计算了 $y-x$ 的值。然后再测试 $x$ 是否小于等于 $y$ ，如果是，就在函数返回result前，将eval复制到result中。

条件数据传送提供了一种用条件控制转移来实现条件操作的替代策略，只能用于非常受限制的情况（当计算量非常大时，性能大大降低）。不过这种情况还是相当常见的，而且与现代处理器的运行方式更契合。

## Loops

C语言提供了多种循环结构，即do-while、while和for。汇编中没有相应的指令，可以用条件测试和跳转组合起来实现循环的效果。

### Do-While Loop

do-while语句的通用形式如下：

```
do
  Body-statement
  while (Test);
```

goto版本：

```
loop:
  Body-statement
  if (Test)
    goto loop
```

效果：重复执行Body-statement，对Test求值，如果求值结果非零，就继续循环。可以看到，在do-while语句中，**Body-statement至少会执行一次。**

下面来看一个例子（goto version），它计算了一个无符号长整型变量x中有多少个二进制位是1：

```
long pcount_goto(unsigned long x) {
  long result = 0;
  loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

对应的汇编代码：

```
movl $0, %eax          # result = 0
.L2:                   # loop:
  movq %rdi, %rdx
  andl $1, %edx          # t = x & 0x1
  addq %rdx, %rax        # result += t
  shrq %rdi              # x >>= 1
  jne .L2                # if (x) goto
  rep; ret
```

条件跳转指令jne是实现循环的关键指令，它决定了是否需要继续重复还是退出循环。

## While Loop

while语句的通用形式如下：

```
while (Test)
Body-statement
```

效果：在第一次执行Body-statement之前，它会先对Test求值，循环有可能直接终止，这与do-while不同。

while语句有两种翻译为机器代码的方法。

## 1. Jump to Middle

goto version:

```
goto test;
loop:
    Body-statement
test:
    if (Test)
        goto loop;
done:
```

这种翻译方法执行一个无条件跳转跳到循环结尾处的测试，以此来执行初始的测试。这也表现出while与do-while的区别，即先执行Test，然后再根据测试结果执行Loop。

## 2. Guarded-do

goto version:

```
if (!Test)
    goto done;
loop:
    Body-statement
    if (Test)
        goto loop;
done:
```

第二种翻译方法实际上把转化为了do-while循环，首先用条件分支，如果初始条件不成立就跳过循环，把代码转换为do-while循环。当使用较高优化等级编译时（例如使用命令行选项-O1），GCC会采用此策略。

## For Loop

for循环的通用形式如下：

```
for (Init; Test; Update )  
    Body-statement
```

它实际上与下面这段while循环代码的行为一样：

```
Init;  
while (Test ) {  
    Body-statement  
    Update;  
}
```

程序首先对初始表达式Init求值，然后进入循环；在循环中它先对测试条件Test求值，如果测试结果为假就退出，否则执行循环体Body-statement；最后对更新表达式Update求值。

GCC为for循环产生的代码是while循环的两种翻译之一，这取决于优化的等级。

综上，C语言中三种形式的所有的循环——do-while、while和for——都可以用一种简单的策略来翻译，产生包含一个或多个条件分支的代码。控制的条件转移提供了将循环翻译为机器代码的基本机制。

## Switch Statements

switch（开关）语句可以根据一个整数索引值进行**多重分支**（multiway branching）。通过使用**跳转表**（jump table）这种数据结构使得实现更加高效。

跳转表是一个数组，表项i是一个代码段的地址，这个代码段实现当开关索引值等于i时程序应该采取的动作。程序代码用开关索引值来执行一个跳转表内的数组引用，确定跳转指令的目标。与使用一组很长的if-else语句相比，使用跳转表的优点是执行开关语句的时间与开关情况的数量无关。

我们通过一个例子来理解机器执行switch语句的工作原理：

switch语句：

```
void switch_eg(long x, long n, long *dest)
{
    long val=x;
    switch(n){
        case 100:
            val *=13;
            break;
        case 102:
            val +=10;
            /*Fallthrough*/
        case 103:
            val +=11;
            break;
        case 104:
        case 106:
            val *=val;
            break;
        default:
            val=0;
    }
    *dest = val;
}
```

下面是对应的汇编代码：

```

void switch_eg(long x, long n, long *dest)
x in %rdi, n in %rsi, dest in %rdx
1    switch_eg:
2        subq    $100, %rsi           Compute index = n-100
3        cmpq    $6, %rsi            Compare index:6
4        ja      .L8               If >, goto loc_def
5        jmp     *.L4(%rsi,8)       Goto *jg[index]
6        .L3:
7        leaq    (%rdi,%rdi,2), %rax
8        leaq    (%rdi,%rax,4), %rdi
9        jmp     .L2
10       .L5:
11       addq    $10, %rdi
12       .L6:
13       addq    $11, %rdi
14       jmp     .L2
15       .L7:
16       imulq   %rdi, %rdi
17       jmp     .L2
18       .L8:
19       movl    $0, %edi
20       .L2:
21       movq    %rdi, (%rdx)
22       ret

```

*Compute index = n-100*  
*Compare index:6*  
*If >, goto loc\_def*  
*Goto \*jg[index]*  
**loc\_A:**  
*3\*x*  
*val = 13\*x*  
*Goto done*  
**loc\_B:**  
*x = x + 10*  
**loc\_C:**  
*val = x + 11*  
*Goto done*  
**loc\_D:**  
*val = x \* x*  
*Goto done*  
**loc\_def:**  
*val = 0*  
**done:**  
*\*dest = val*  
*Return*

执行switch语句的关键步骤是通过跳转表来访问代码位置。jmp指令的操作数有前缀'\*'，表明这是一个间接跳转（Indirect jump），操作数指定一个内存位置，索引由寄存器%rsi给出，这个寄存器保存着index的值。

C代码将跳转表声明为一个数组，数组中的每一个元素都是一个指向代码位置的指针。这些元素跨越index的值0~6，对应于n的值100~106，这样就可以间接地访问并处理所有分支。这就是间接（Indirection）的作用。

在上面这个例子中，程序可以只用一次跳转表引用就分支到5个不同的位置。甚至当switch语句有上百种情况的时候，也可以只用一次跳转表访问来处理。

所以说，

---

"All problems in computer science can be solved by another level of indirection."

----David Wheeler

---

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.7 Procedures

在上一节中我们讨论了条件控制以及循环语句在机器级代码中的翻译，在这一节中我们将会进一步讨论 C 语言中另一个重要的组成部分：**函数**。

无论是**函数**还是说面向对象中的**方法**又或是汇编中的**过程**其实就是一段代码，一个代码块。研究机器级代码中函数调用机制，帮助我们更好的理解**运行栈**，可以显著提高我们代码运行效率

## Mechanisms in Procedures

我们先宏观的来思考一下，函数调用过程应当做些什么。总的来说应该有三个方面：

1. **Passing control:** 我们有先前的积累，已经能够理解代码翻译后在机器中只是一条一条指令。那么函数调用时，实际上就是要从一条待执行的指令的位置，跳转到去执行另一条指令的位置，并从那里开始执行一系列指令。函数结束时，我们需要返回到调用指令的位置，接着执行调用指令之后的指令。**PC(程序计数器)** 指令变更的过程我们需要研究。
2. **Passing data:** 函数调用过程还需要传递一些参数，这些参数该以怎样的形式传递？函数调用结束时需要向调用者返回一些返回值，返回值应当怎样传递？这些数据传递的问题需要讨论。
3. **Memory management:** C 语言中函数中可以定义局部变量，这些局部变量应当怎样分配存储空间？在函数结束时，为了避免空间浪费，我们需要释放这些空间，如何保证空间释放的合理？

接下来几个小节我们将分别深入去研究这些部分。但在开始研究这些问题以前，我们还要提一个重要的概念：**Application Binary Interface(ABI)**。

从计算机系统的角度，如果不考虑效率便捷性之类的问题，要实现函数调用时参数传递可以有很多方式，我可以存在内存里、存在运行栈里或者保存在特殊的寄存器中，但为了一个统一标准，人们提出了**ABI**来约束，机器指令的实现都会遵循这个标准，这个标准就是**ABI**。

甚至对于不同的操作系统会有不同的**ABI**，我们接下来讨论的内容都是基于 Linux 系统的 **ABI**。如果你在学完这节课后觉得你有更好的想法，欢迎你提出你的 **ABI**，并说服全世界计算机行业的人放弃以前的标准采取你的标准~~（先不考虑兼容性问题）~~

# Stack Structure

要研究函数调用绕不开**运行栈**这个概念，相信大家在初学递归的时候都有过一个经历：因为忘记写递归的退出条件，导致无限递归下去，最后程序被操作系统杀死并告诉你**栈溢出了**。那个时候你一定有疑问：我根本没有用到栈这个数据结构，溢出在哪？🤔

如果没有学过 ics 课，你只会觉得操作系统又犯病了 😅，事已至此先爬一把塔吧 🤔。唉，别急着打开 steam，我们这门课会给你一个答案。

这里因为大家应该都学过数据结构与算法这门课，栈这个基础的数据结构就不再过多介绍了，如果你有点记不清了，可以去问问 GPT or Deepseek：“什么是栈？”

这里所谓的栈溢出实际上就是指运行栈溢出了。所谓的**运行栈是内存中一段特殊的位置**，位于虚拟内存中地址较大的位置。x86-64 中的栈是一个**倒置的栈**，栈底位于内存很高的一个地址，这个栈的起始地址会是一个随机数~~(你问为什么要随机，这你可能要问一下黑客们了，问问他们干了什么好事)~~ 然后向下“生长”，每当我向其中存元素时栈就向下延伸。

在前文我们介绍寄存器的时候我们经常提到 **%rsp** 这个寄存器的特殊性，这一节我们终于能讲清楚特殊在哪里。**%rsp** 始终保存的是**当前栈顶**的位置，也就是一根指针便于我们维护栈。

提到栈离不开两个操作：**入栈(push)**和**出栈(pop)**。由于这两个操作用的太多了，x86 指令集将这两个操作单独抽出来做成了两条指令~~（尽管他们都能被其他指令代替）~~。

1. **Push:** `pushq Src`，将 `Src` 写入当前 `%rsp` 指向的内存位置，并将 `%rsp` 下移。
2. **Pop:** `popq Dest`，将 `%rsp` 指向内存的位置写入 `Dest` 中。

有了栈的知识，我们可以开始研究在第一小节中提到的几个问题了。

# Passing Control

我们通过运行栈来帮助我们完成过程的调用与返回。我们约定返回位置位于**调用指令的后一条**（返回在前面或者返回在调用处不就一直反复调用了吗）

函数调用指令 **Call**, `call label`，将返回的位置存入栈中，并跳转到 `label` 标记的代码块。

返回指令 **Ret**, `ret`，将返回指令的地址从栈中 `pop` 出来，并跳转到那条指令去。

控制的转移实际上就这么简单，调用的时候存一下返回地址然后调用，返回的时候返回到预先存好的目的地就可以了。

## Passing Data

API约定函数调用前六个参数保存在指定寄存器中，多于六个的参数保存在栈中~~（真会用到吗？）~~。返回值始终保存在 `%rax` 中。下面一张图片指明了寄存器的传参的顺序。

### Registers

#### First 6 arguments

|                   |
|-------------------|
| <code>%rdi</code> |
| <code>%rsi</code> |
| <code>%rdx</code> |
| <code>%rcx</code> |
| <code>%r8</code>  |
| <code>%r9</code>  |

### Stack



### Return value

`%rax`

**Only allocate stack space when needed**

举一个简单的例子熟悉一下数据的转移：

# Data Flow Examples

```
void multstore
    (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x in %rdi, y in %rsi, dest in %rdx
...
400541: mov    %rdx,%rbx      # Save dest
400544: call   400550 <mult2>  # mult2(x,y)
# t in %rax
400549: mov    %rax,(%rbx)    # Save at dest
...
```

```
long mult2
    (long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
# a in %rdi, b in %rsi
400550: mov    %rdi,%rax      # a
400553: imul   %rsi,%rax      # a * b
# s in %rax
400557: ret                 # Return
```

可以看到在这个例子中 `multstore` 调用了 `mult2`，`mult2` 直接将 `%rdi` 和 `%rsi` 中的值作为参数使用，并且计算返回的结果在 `%rax` 中。

## Managing local data

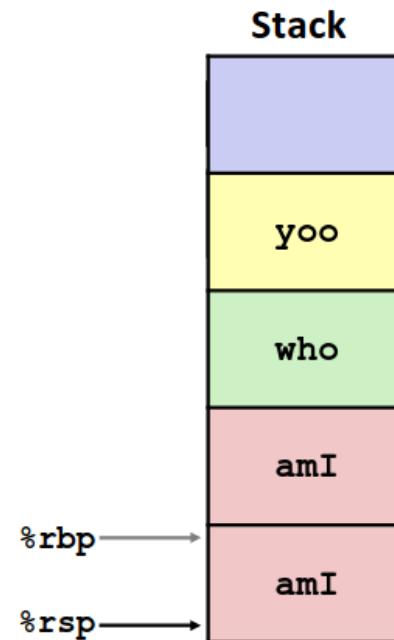
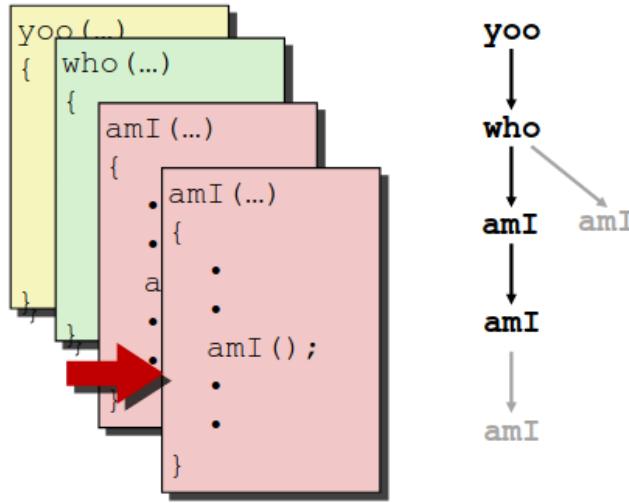
在函数调用的过程，我们不止需要传递参数，还需要管理保存数据。当 A 调用 B 的时候 A 应当妥善保管自己的数据，告诉 B 哪些数据是可以动的，哪些是 A 还需要的。对于 B 来说，他需要知道他可以使用哪些数据，他新定义的变量数据存在哪里。

机器级代码通过栈帧(Stack Frames)这个概念来实现数据的管理，简单来讲就是对于每个调用的过程，会在栈中开辟一段属于自己的空间，栈帧中保存应当返回的位置，函数的局部变量需要时也存在自己的栈帧中。一个函数不应当去篡改其他函数栈帧中的内容，那里面有其他函数需要的重要数据。

在函数结束时，函数所属的栈帧中的数据内容已经不再有用，当返回回去以后，这个栈帧就可以被释放了，这一段栈空间供其他函数使用以节省内存空间。

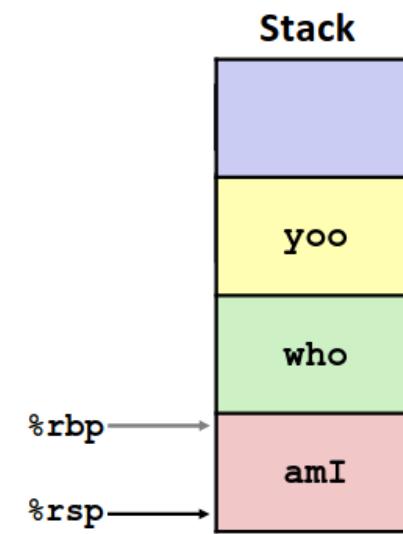
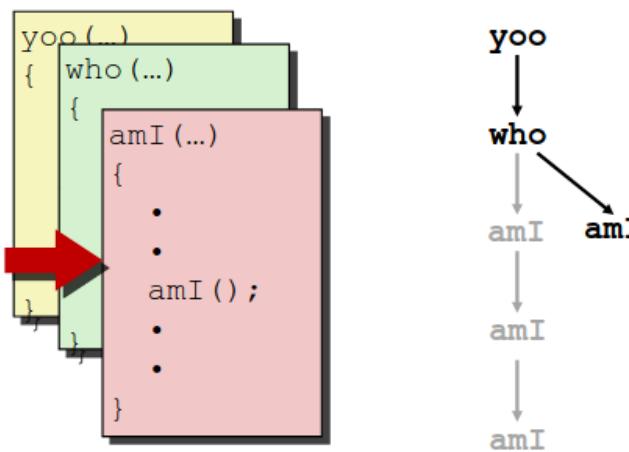
我们举一个简单的例子来理解一下这个过程。如图是一个函数调用的路径。

## Example



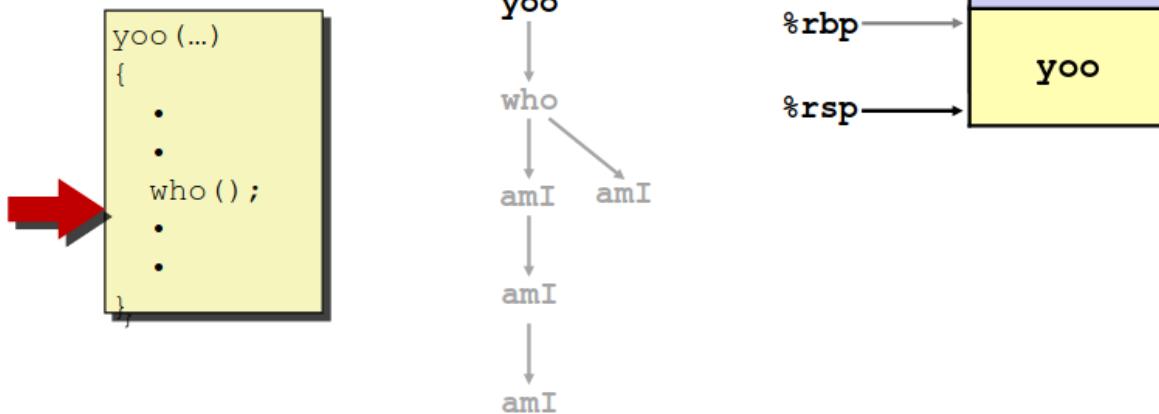
不难看到从 `yoo -> who -> amI -> amI` 的调用路径中，越早调用的函数栈帧在越上面（别忘了栈是倒序排列的）。而栈先进后出的性质，退栈时显然是相反的顺序。

## Example



可以看到退栈的时候，后调用的栈按顺序退出，只留下先调用的函数栈帧。这时候如果有函数调用，位置位于先前退掉的位置。

# Example



然后所有调用都结束了，退会最初的调用栈帧 `yoo` 中，这时候如果 `yoo` 中还有数据处理要做，就接着在 `yoo` 的栈帧中处理，直到退回主函数。

这时候就可以回答这一小节开始时的问题了，无限递归时我都没有用到栈这种数据结构这么会栈溢出呢？

无限递归是不断创建栈帧却没有返回，就算函数中什么都没做，也需要栈帧来存返回地址，不断创建栈帧，直到把操作系统分配给你这个进程的栈空间用完了，栈就溢出了，操作系统无情的杀死你的程序，并冷血的清空了剩余宝贵~~（完全没用）~~的数据 😢

那么函数调用时存在栈中的数据得到了妥善的管理，那么寄存器呢？一个寄存器既需要被调用者使用，也需要被被调用者使用。

唉，我们怎么给过程分类的，分为**调用者(caller)**和**被调用者(callee)**。同样我们也将保存寄存器的职责划分给两者。寄存器分为 **caller-saved** 与 **callee-saved** 两类。

对于调用者，如果一个寄存器的值你认为很重要并且是 **caller-saved**，并且你在调用结束以后还需要用到这个寄存器，你就应当保存这个寄存器的值到你的栈帧中。在调用结束的时候调用者会将这些值从栈中取出还原回寄存器中。具体来讲 **caller-saved** 寄存器有如图这些：

# x86-64 Linux Register Usage #1

## %rax

Return value

Also caller-saved

Can be modified by procedure

## %rdi, ..., %r9

Arguments

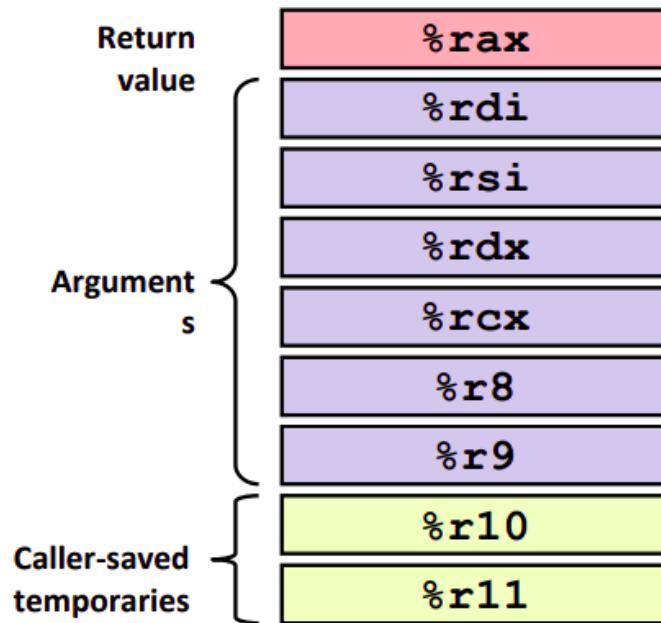
Also caller-saved

Can be modified by procedure

## %r10, %r11

Caller-saved

Can be modified by procedure



对被调用者有同样的职责，如果有些寄存器你需要使用并且是 callee-saved，不管调用者是否需要，那么你就应当将这些寄存器的值保存在你的栈帧中，并且在将控制还给 caller 时，你应当现将寄存器的值取出并还原。具体来讲 callee-saved 寄存器有如图这些：

# x86-64 Linux Register Usage #2

**%rbx, %r12, %r13, %r14**

Callee-saved

Callee must save & restore

**%rbp**

Callee-saved

Callee must save & restore

May be used as frame pointer

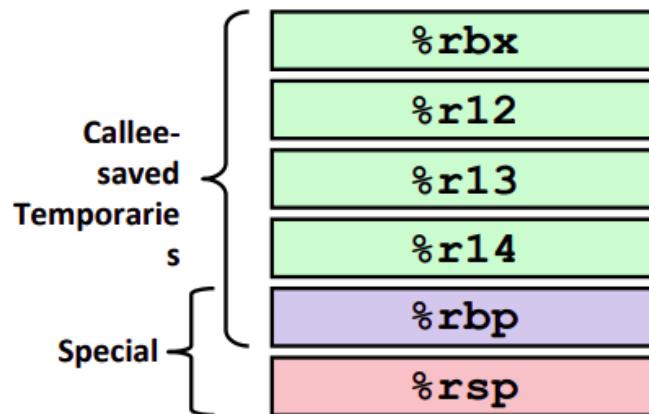
Can mix & match

**%rsp**

Special form of callee save

Restored to original value upon

exit from procedure



那么关于函数过程调用这个话题我们就讨论就到此为止了，关于递归之类的话题，我们也在本节中间穿插的提到了一些，甚至中间举的一个例子就是递归调用，不难意识到所谓的递归调用和一般的函数调用没有任何区别，本文就不再额外去说明这一点了。希望对你有帮助。

# Chapter 3.8 Array Allocation and Access

C语言中的数组是一种将标量数据聚集成更大数据类型的方式。C语言可以产生指向数组中元素的指针，并对这些指针进行运算。在机器代码中，这些指针会被翻译成地址计算。（别忘了，机器代码在使用内存时是不区分数据类型的。）

## Basic Principles

对于数据类型Type和整型常数Length，声明如下：

```
Type name[Length];
```

起始位置表示为 $x_{name}$ 。这个声明有两个效果：

- 首先，它在内存中分配了一个大小为  $Length * sizeof(Type)$  字节的连续区域。
- 其次，它引入了标识符name，作为指向数组首地址的指针，该指针的值为 $x_{name}$ 。

因此，我们可以用0~Length-1的整数索引来访问数组元素。

数组元素i的地址为 $x_{name} + sizeof(Type) * i$ 。

x86-64的内存引用指令可以简化数组访问。例如，假设E是一个int型的数组，我们现在希望计算E[i]。其中，E的地址存放在%rdx寄存器中，i存放在%rcx寄存器中。那么，指令 movl (%rdx,%rcx,4),%eax 就会执行地址计算 $x_E + 4i$ ，读取这个内存位置的值，并将结果存放到%eax寄存器中。

## Pointer Arithmetic

C语言允许对指针进行运算，而计算出来的值会根据该指针引用的数据类型的大小进行伸缩。也就是说：若p是一个指向类型为T的数据的指针，p的值为 $x_p$ ，那么表达式 $p + i$ 的值为 $x_p + L * i$ 。（L为数据类型T的大小）

通过指针运算，我们可以扩展一下前面的例子，对整型数组E[i]做一些指针运算：

| Expression | Type  | Value              | Assembly code              |
|------------|-------|--------------------|----------------------------|
| E          | int * | $x_E$              | movl %rdx,%rax             |
| E[0]       | int   | $M[x_E]$           | movl (%rdx),%eax           |
| E[i]       | int   | $M[x_E + 4i]$      | movl (%rdx,%rcx,4),%eax    |
| &E[2]      | int * | $x_E + 8$          | leaq 8(%rdx),%rax          |
| E+i-1      | int * | $x_E + 4i - 4$     | leaq -4(%rdx,%rcx,4),%rax  |
| *(E+i-3)   | int   | $M[x_E + 4i - 12]$ | movl -12(%rdx,%rcx,4),%eax |
| &E[i]-E    | long  | $i$                | movq %rcx,%rax             |

其中，整型数组E的起始地址和整数索引i分别存放在寄存器%rdx和%rcx中。可以看出，返回数组值的操作类型为int，因而使用4字节操作 movl，并将结果存放在%eax中。而返回指针的操作类型为int \*，因此使用8字节操作 leaq等，结果存放在%rax中。

## Nested Arrays

嵌套数组可以理解为数组的数组。如果我们声明：

```
int A[5][3];
```

它等价于下面的声明：

```
typedef int row3_t[3];
row3_t A[5];
```

这里的操作实际上把数据类型 row3\_t 定义为一个有三个整数的数组，数组A包含5个这样的元素，即每个元素都是有三个整数的数组。因此每个元素的大小是12个字节，那么整个数组A的大小就是60个字节。

我们也可以把数组A看成一个5行3列的二维数组，用 A[0][0] 到 A[4][2] 来引用。数组元素在内存中按照行优先的顺序排列，这正是我们上面描述的嵌套声明的结果，所以第0行的所有元素可以写作 A[0][i] ( $i < 3$ )，以此类推。

访问多维数组的元素和一维数组类似，编译器会以数组起始为基地址，偏移量为索引（可能需要经过伸缩）来访问期望元素。通常来说，对于一个声明如下的数组：

```
T A[R][C];
```

它的数组元素 $A[i][j]$ 的内存地址为 $x_A + \text{sizeof}(T)(C * i + j)$ 。

仍然考虑前面定义的数组 $A[5][3]$ , 其寻址方式在机器代码中表示如下:

|   | <i>A in %rdi, i in %rsi, and j in %rdx</i> |  |
|---|--|--|
| 1 | <code>leaq (%rsi,%rsi,2), %rax</code>      | <i>Compute 3i</i>                              |
| 2 | <code>leaq (%rdi,%rax,4), %rax</code>      | <i>Compute <math>x_A + 12i</math></i>          |
| 3 | <code>movl (%rax,%rdx,4), %eax</code>      | <i>Read from M[<math>x_A + 12i + 4</math>]</i> |

可以看到, 访问二维数组中的元素, 必须执行两次内存读取: 首先获取指向行数组的指针, 然后访问行数组内的元素。计算所得的元素地址为 $x_A + 12i + 4j = x_A + 4(3i + j)$ , 使用了x86-64地址运算的伸缩和加法特性。

## N X N Matrix Code

对于n维矩阵, 在C语言中有三种类型:

### 1. 固定维度

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j)
{
    return A[i][j];
}
```

### 2. 可变维度, 显式索引(Traditional way to implement dynamic arrays)

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A, size_t i, size_t j)
{
    return A[IDX(n,i,j)];
}
```

### 3. 可变维度，隐式索引( Added to language in 1999)

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
    return A[i][j];
}
```

---

btw, `#define N` 声明是一个很好的编程习惯, 当我们要在程序中使用常数时 (如作为数组的维度), 最好通过 `#define` 声明将该常数与一个名字联系起来, 然后在使用时以这个名字替换常数的值。这样一来, 如果想要修改这个值, 只需简单地修改 `#define` 声明即可, 同时也能提升代码的可读性。

---

访问可变维度的n维矩阵的元素的机器代码如下：

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq  %rdx, %rdi          # n*i
leaq    (%rsi,%rdi,4), %rax # A + 4*n*i
movl    (%rax,%rcx,4), %eax # Mem[A + 4*n*i + 4*j]
ret
```

可以看到, 在访问可变维度矩阵的元素时, 编译器使用了乘法操作 `imulq` 来获取行数组指针。在前面的章节中, 我们知道编译器为了提升程序运行的速度会尽量避免使用乘法, 但在可变维度矩阵中, 由于n是未知的, 因此乘法运算是无法避免的。这是机器代码实现变长数组访问时与定长数组的区别。

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.9 Heterogeneous Data Structures

C语言提供了两种将不同类型的对象组合到一起创建数据类型的机制：

- **结构 (structure)**：用关键字**struct**来声明，将多个对象集合到一个单位中。
- **联合 (union)**：用关键字**union**来声明，允许用几种不同的类型来引用一个对象。

## Structures

### Allocation

结构体的声明示例如下：

```
struct rec {  
    int i;  
    int j;  
    int a[4];  
};
```

结构体在内存中表现为一块足够大以容纳所有字段的内存块，其中的字段按声明的顺序排序，并由编译器确定字段的大小和位置。

### Access

仍然考虑上面声明的结构体，若要访问结构体里的字段，编译器产生的代码要将结构体的地址加上适当的偏移。例如，要访问y，由于字段x的偏移量是0，这个字段的地址就是r的值（r是**struct rec\***类型的变量）。为了存储到字段y，代码要将r的地址加上偏移量4。

若要访问结构体内部的数组a[4]，需要产生一个指向该数组的指针。在本例中，我们只需将结构的地址加上偏移量 $8 + 4 * i$ 就可以得到指针&(r->a[i])。

我们通过一个具体的指令来看看机器代码中获取结构体内元素地址的操作：

```
r->p=&r->a[r->i+r->j];
```

对应汇编指令：

|   | Registers: r in %rdi      |                            |
|---|---------------------------|----------------------------|
| 1 | movl 4(%rdi), %eax        | Get r->j                   |
| 2 | addl (%rdi), %eax         | Add r->i                   |
| 3 | cltq                      | Extend to 8 bytes          |
| 4 | leaq 8(%rdi,%rax,4), %rax | Compute &r->a[r->i + r->j] |
| 5 | movq %rax, 16(%rdi)       | Store in r->p              |

从中我们可以看到，结构的每个字段的选取完全是在编译时处理的。机器代码不包含关于字段声明或字段名称的信息。所以我们在汇编代码中只能看到偏移量，而看不到字段名称。

## Alignment

许多计算机系统对基本数据类型的合法地址做出了一些限制，要求某种类型对象的地址必须是某个值（通常是2、4或8）的倍数。这种**对齐限制**简化了形成处理器和内存系统之间接口的硬件设计。如果没有字节对齐，我们就可能需要多次内存访问来获取一个对象的地址，这无疑降低了运行效率。

**对齐原则是：任何K字节的基本对象的地址必须是K的倍数。（指原始数据类型，如int a[3]应以int的大小为基准）**

具体如下：

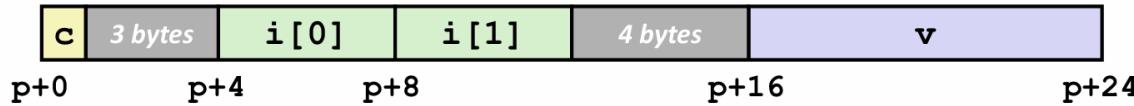
| K | Types               |
|---|---------------------|
| 1 | char                |
| 2 | short               |
| 4 | int, float          |
| 8 | long, double, char* |

在结构体内部，为了保证每个结构体元素都满足它的对齐要求，编译器可能需要在字段的分配中插入间隙。

例如，考虑下面的结构体声明：

```
struct S1 {
    char c;
    int i[2];
    long v;
} *p;
```

它的偏移内容如下：

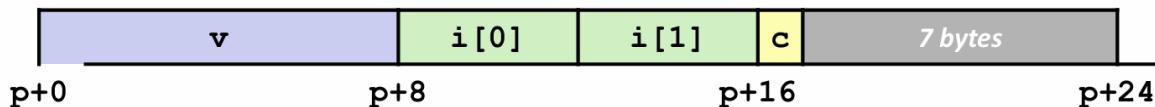


可以看到，为了满足字段i[2]的4字节对齐要求和字段v的8字节对齐要求，编译器在字段c后面插入了3字节的间隙，在字段i[2]后面插入了4字节的间隙。

同时，为了使所有结构体都满足对齐要求，编译器还可能在结构体的末尾加入多余字节。

例如，考虑下面的结构体声明：

```
struct S2 {
    long v;
    int i[2];
    char c;
} *p;
```



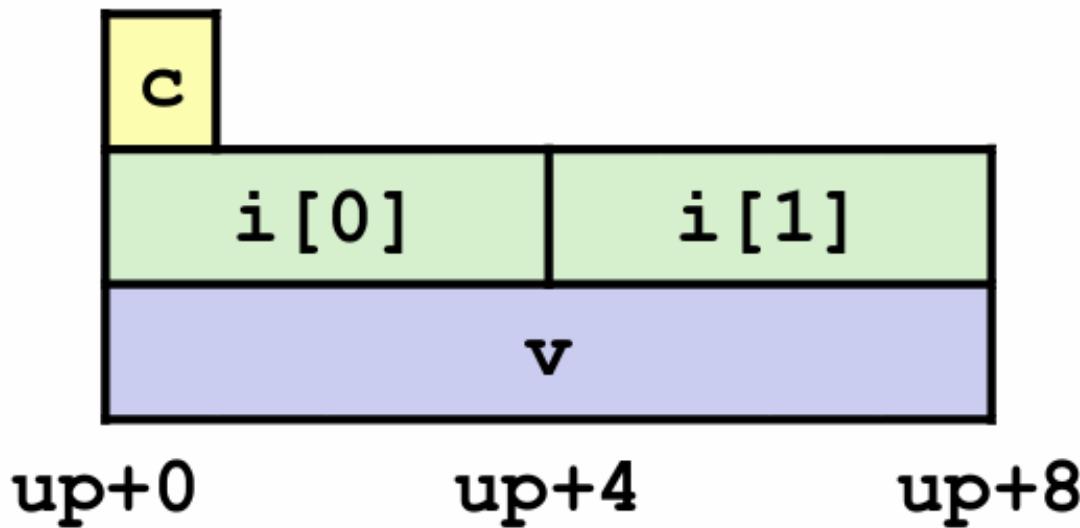
可以看到，编译器为结构体S2分配了24个字节，这是为了保证整体的字节对齐要求。

## Unions

联合提供了一种方式，能够规避C语言的类型系统，允许以多种类型来引用一个对象。联合声明的语法和结构体一样，但是语义相差很大。它们是用不同的字段来引用相同的内存块。

我们一个例子来说明：

```
union U1 {  
    char c;  
    int i[2];  
    long v;  
} *up;
```



可以观察到，一个联合的总大小等于它最大子段的大小，这是因为它内部的不同字段引用的是相同的内存。这就是联合与结构的最大区别。

---

© 2025. ICS Team. All rights reserved.

# Chapter 3.10 Advanced Topic

经过前面的学习，我们已经掌握了基础机器级代码的知识，在本章的最后一节我们将讨论一个有趣的话题：内存以及内存溢出。

相信大家都是对内存溢出深恶痛绝，C 语言作为一个不检查数组越界的语言，由于程序员的疏忽导致越界而造成的 bug 不计其数。

然而内存溢出不仅仅只会造成 bug，还会带来一些有趣的黑客技术，我们将简单介绍两种攻击技术以及保护手段。并且你会在 attack lab 中运用这个知识去攻击我们为你提供的简单程序。

如果你看了本节的内容并完成了 attack lab 后跃跃欲试，去学习了一些更为先进的技术，准备干一些不为人知的勾当，请在被捕并判处有期徒刑（~~这取决于你干了票多大的~~）后不要供出 ics 课程组 😎 (你日后若惹出祸来.jpg)

## Memory Layout

要介绍内存溢出以前我们先要了解内存是什么样的。在之前的内容中我们已经提及过一些，这里再总结一下。下图展示了 Linux 系统中内存地址的分配。

## ■ Stack

- Runtime stack (8MB limit)
- e.g., local variables

## ■ Heap

- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

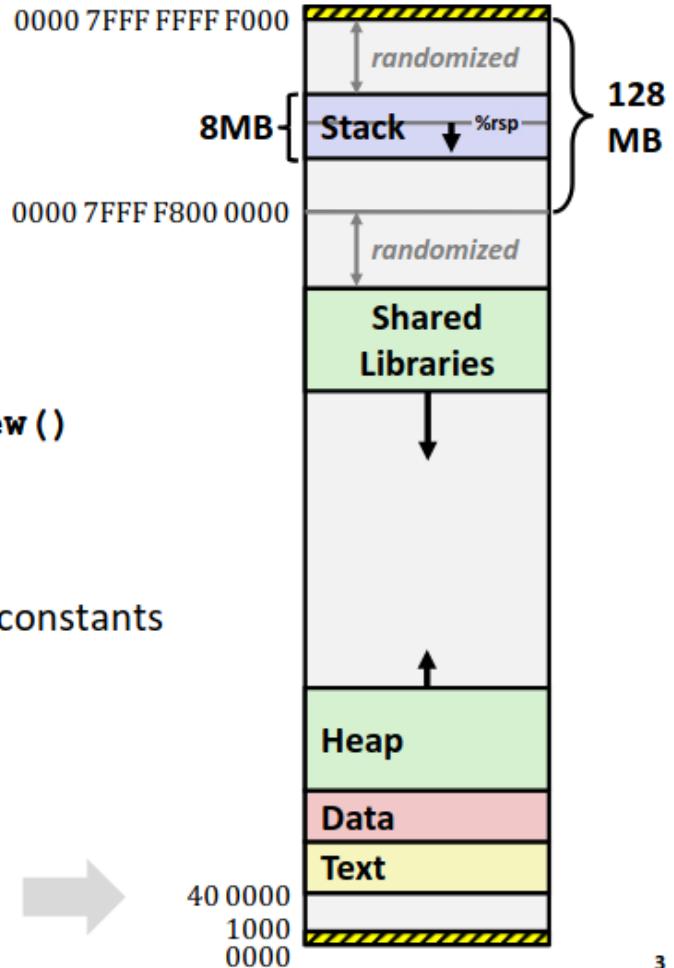
## ■ Data

- Statically allocated data
- e.g., global vars, `static` vars, string constants

## ■ Text / Shared Libraries

- Executable machine instructions
- Read-only

Hex Address →



3

从图中不难看到，虽然我们常说64位机器地址总共有 64 位，然而我们内存空间实际上只用了 47 位，然而这个数字实际上已经足够大了。

图中也简要介绍了一下不同的内存区域是用于什么功能的。其中 Stack, Heap, Data, Text 这些区域之前我们已经多次接触过了比较熟悉了。至于 Shared Libraries 是做什么的，我们将在 Link 那一章节中介绍。

## Buffer Overflow

在简要的回顾了一下内存布局之后，我们开始今天的主题：**内存溢出**。

## Vulnerability

我们举一个简单的内存溢出的例子。下图是一个简单的代码以及一些输入时的结果。

```
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

|               |    |                           |
|---------------|----|---------------------------|
| <b>fun(0)</b> | -> | <b>3.1400000000</b>       |
| <b>fun(1)</b> | -> | <b>3.1400000000</b>       |
| <b>fun(2)</b> | -> | <b>3.1399998665</b>       |
| <b>fun(3)</b> | -> | <b>2.0000006104</b>       |
| <b>fun(6)</b> | -> | <b>Segmentation fault</b> |
| <b>fun(8)</b> | -> | <b>3.1400000000</b>       |

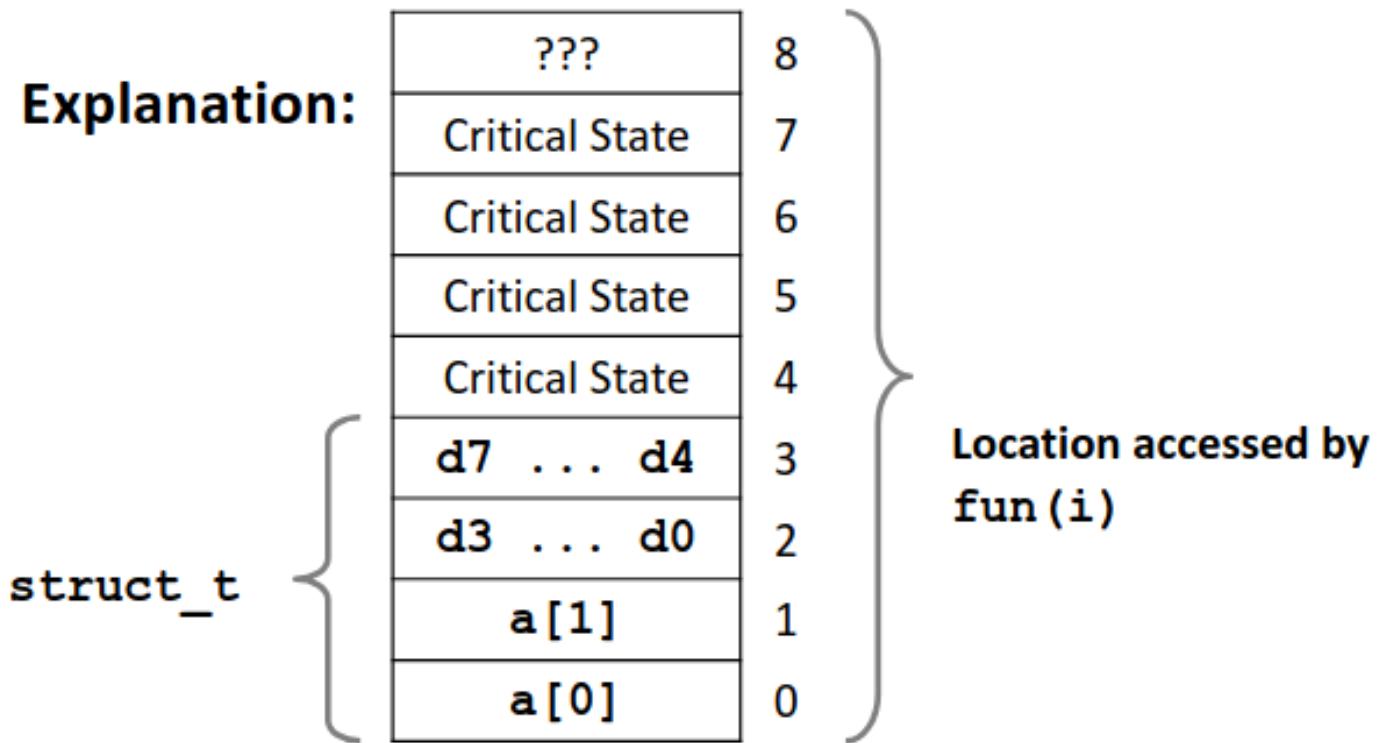
可以看到由于 C 语言不做越界检查，导致明明 a 数组仅有两个元素，却可以访问 a[2] 乃至 a[8]，这时候编译器会顺延访问 a 后面的内存，并将那些内存字节解释为 int 并进行运算。可以看见当 i=2,3 时，根据结构体排列顺序，a 后面的 double 的值被篡改了。

但奇怪的是为什么 i=6 时会发生 segmentation fault 但 i=8 时却没什么~~（看似）~~影响了呢？这就需要研究一下栈中元素的排列了。下图是这个函数栈排布的示意图。

```
typedef struct {
    int a[2];
    double d;
} struct_t;
```

|        |    |                    |
|--------|----|--------------------|
| fun(0) | -> | 3.1400000000       |
| fun(1) | -> | 3.1400000000       |
| fun(2) | -> | 3.1399998665       |
| fun(3) | -> | 2.0000006104       |
| fun(6) | -> | Segmentation fault |
| fun(8) | -> | 3.1400000000       |

## Explanation:



显然在这个函数的栈帧中，`a[6]` 会访问到一些不应当访问的部分，导致报错，而进一步访问`a[8]`时越过了这个重要的部分。根据之前的知识可能是访问到了栈帧中的**返回地址**，也有可能会是我们在之后会介绍的**金丝雀**。

我们再举一个例子，我们来拷打一下 C 语言的一些库函数的设计。

在初学 C 语言时，相信有老师告诉过我们，不要用 `gets` 不安全，今天我们终于可以解释清楚为什么不安全了。下图是一个 `gets` 的典型实现。

## ■ Implementation of Unix function gets ()

```

/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}

```

- No way to specify limit on number of characters to read

可以看到这个函数不会对输入进行任何的限制，只是一味读入直到遇见换行或者文件结尾(EOF)。那么我们大可输入一长串字符来刻意制造一个内存溢出，以达到一些不可告人的秘密。这就引出了今天介绍的第一个黑客技术：**Code Injection Attacks**.

### Code Injection Attacks

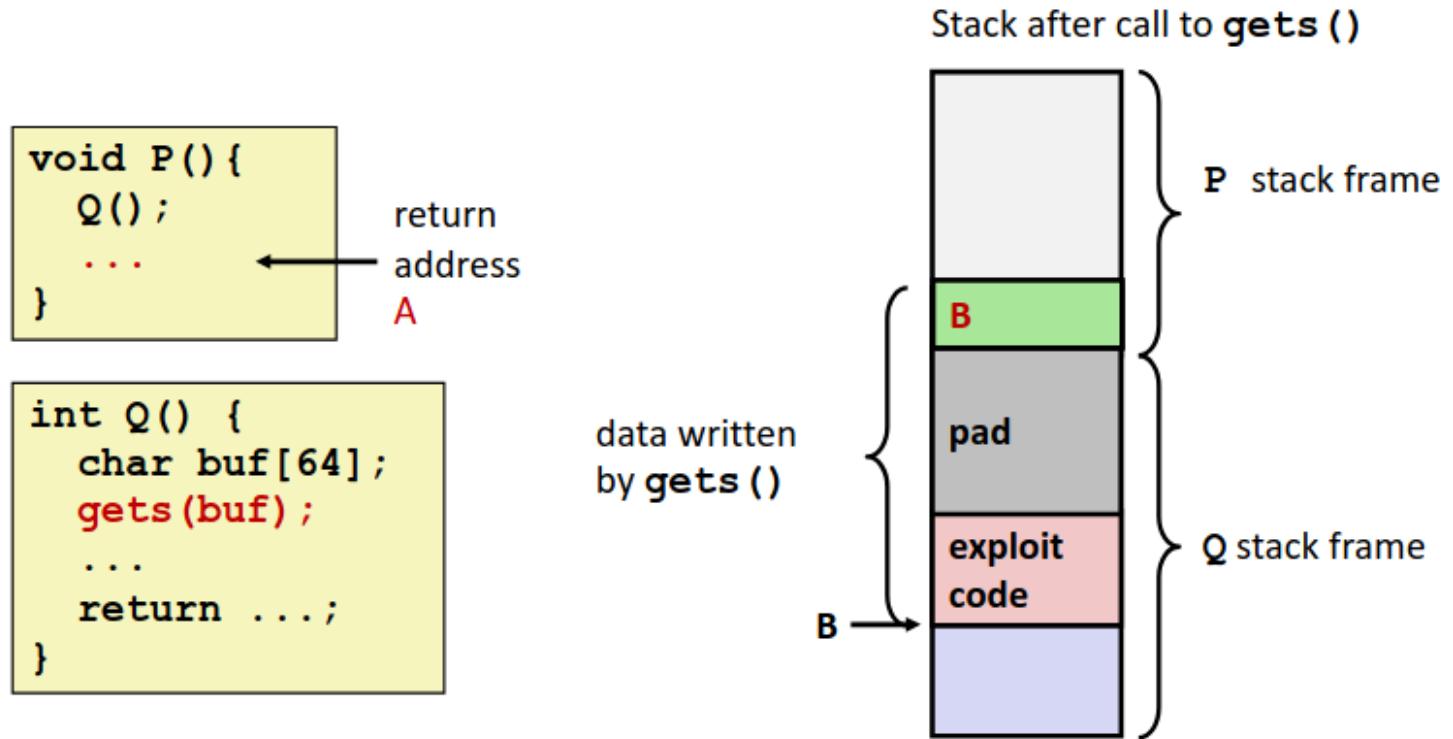
如果我们通过反汇编，碰巧发现～（精心搜寻）～～了一个像 gets 一样的漏洞。

我们可以干什么？我们可以输入一个‘碰巧’的非常长的字符串，制造一个内存溢出，如果‘碰巧’这个字符串溢出更改后的栈帧中，函数返回时‘碰巧’返回到了你插入字符串的部分的地址，然后顺着执行了一串代码，‘碰巧’这段看似毫无意义的字符串解释为机器代码的话是帮你‘碰巧’得到了服务器的 root 权限，你‘碰巧’得到了你所有想要的，实在运气太好了！

当然不是碰巧，一切都是预先设计好的，我们通过预先分析栈帧，就可以清晰的得知返回地址存在栈中的位置，通过溢出篡改那个地址的值，让他返回到我们设计好的注入代码的位置，就可以执行

我们注入的代码了。

下图是一个注入攻击的简单示意图。



## Protection

如果我们预先知道了这种攻击手段，我们该怎样保护我们的程序免收攻击呢？

- Avoid Overflow Vulnerabilities in Code:** 最好的方法，如果每一个程序员都接受过 ics 的教育，那么他会自觉的用 fgets 代替不安全的 gets，无时无刻不谨记着检查溢出。那么黑客也就无机可乘了。
- System-Level Protections Can Help:** 然而不是每个程序员都上过 ics 课~~(有可能只是他没选上课吧)~~。那么我们这些上过 ics 课的‘高手’们自然要设计出一些机制去让他们写出不安全的代码也免受攻击。
  - Randomized stack offsets:** 如果栈起始的地址是一个随机数，那么在每次执行时栈帧的位置都不同，那么黑客就不知道篡改返回地址应该返回到哪个地方才能执行他们的注入代码了。
  - Non-executable memory:** 栈是拿来存数据的地方，不是用来存代码的地方！如果操

作系统好好监管，将栈设做不可执行区，那么黑客试图在栈上执行代码时会被操作系统无情的杀死。

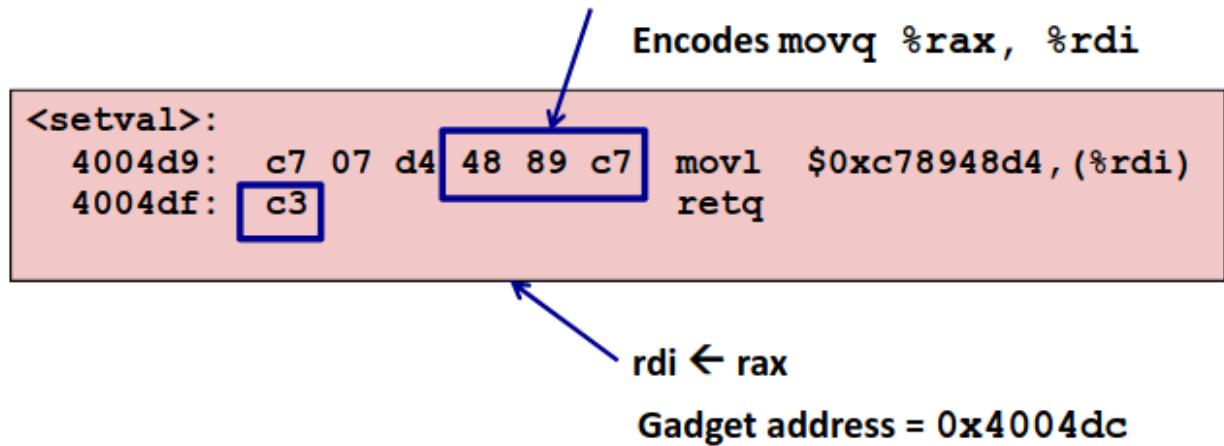
## Bypassing Protection

道高一尺魔高一丈，你栈随机了，我不知道返回到栈的哪个地方了，我就不返回到栈上嘛，你写好的原有的代码的地址总不是随机的吧？栈上不让执行，你原有的代码总让执行吧？聪明的黑客发明了一种新技术Return-Oriented Programming Attacks。

如果我将字符串设置为一串程序原有的代码的地址，这些代码有一个共有的特征：进行了一些简单的操作之后调用了指令 `0x3c: ret`，那么会返回到我在栈中准备好的下一个位置，依次进行下去，我会通过一个个简单的指令，拼出一段程序，达成我不可告人的目的。

这些以 `0x3c` 结尾的代码，被黑客们称为 **gadget**，下图是一个 gadget 的例子。

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



你说这能干啥？但是，一个足够大的工程上商业上的程序，虽然不会有像我们实验中那样可以营造的 gadget，但足够大的样本也会产生足够的 gadget 去让黑客发掘了。拼拼凑凑总能凑出些东西来实施一些破坏了。

那么魔高一尺道高一丈，我们总得有些办法来阻止黑客吧，毕竟不是所有聪明的人都去挖空心思攻

击别人了，总还有些高手费尽心思来防范攻击吧。

我们回想一切漏洞的根源：**内存溢出**。前面的技术总是通过溢出的数据篡改了栈帧。那么我们如果能够检查到栈帧被篡改，发现这个现象时直接退出，不给黑客留下操作空间不就可以了吗？

**金丝雀(Canaries)** 应运而生。

金丝雀这个名字有一个由来，传说金丝雀这种动物很怕死，并且对有毒气体非常敏感。由于矿井中可能出现有毒气体，而人的嗅觉不一定对那些气体敏感，那么那些煤矿工人在下矿时就会带上一只金丝雀，这样如果金丝雀出现了异常的反应证明存在有毒气体。

那么我们在创建栈帧的时候也带上一只“金丝雀”，下图是一个简单的例子。

```
1169: push    %rbx
116a: sub     $0x10,%rsp
116e: mov     %fs:0x28,%rax
1177: mov     %rax,0x8(%rsp)
117c: xor     %eax,%eax
117e: lea     0x4(%rsp),%rbx
1183: mov     %rbx,%rdi
1186: callq   1050 <gets@plt>
118b: mov     %rbx,%rdi
118e: call    1030 <puts@plt>
1193: mov     0x8(%rsp),%rax
1198: sub     %fs:0x28,%rax
11a1: jne    11a9 <echo+0x40>
11a3: add    $0x10,%rsp
11a7: pop    %rbx
11a8: ret
11a9: call    1040 <__stack_chk_fail@plt>
```

可以看到图中红色部分的代码，可以理解为，我们在创建栈帧时就取一个随机值，把这个随机值保存在一个寄存器中，并存在栈中，在退出函数时，我们再检查这个寄存器中的值和保存在栈中的值是否一致，如果不一样，要么是无意为之的溢出，要么有别有用心的人刻意篡改，直接报错退出。

这个技术就叫金丝雀技术，它杀死了通过一般的栈溢出攻击的可能性。现代编译器在编译的时候都会为栈中加上一只“金丝雀”，像我们的 attack lab 的程序都是通过特定的指令去让编译器关掉金丝雀的~~(无视风险继续访问)~~，否则你的技术是无法攻克金丝雀的。

这时候你可能想说了，唉，道高一尺魔高一丈，我想学...打住打住，我们对这些技术的介绍就到此为止了，可以看到我们介绍的黑客技术所利用的漏洞都已经被修复了，如果你想听点没被修复的漏洞，可能你今天上的课，明天课程组就在橘子里了。更多的技术等待着你自己探索了，你日后若是惹出事来...

---

那么这一节的知识就到此为止了，我们整个第三章程序的机器级表示也就介绍完了。相信大家在学习完这一章的内容后对程序有了更深入的理解，希望这对你有帮助 

---

© 2025. ICS Team. All rights reserved.

# Chapter 4 Processor Architecture

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.1 The Y86-64 Instruction Set Architecture

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.2 Logic Design and the Hardware Control Language HCL

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.3 Sequential Y86-64 Implementations

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.4 General Principles of Pipelining

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.5 Pipelined Y86-64 Implementations

---

© 2025. ICS Team. All rights reserved.

# Chapter 4.6 Summary

---

© 2025. ICS Team. All rights reserved.

# Chapter 5 Optimizing Program Performance

---

© 2025. ICS Team. All rights reserved.

# Chapter 5.1 Capabilities and Limitations of Optimizing Compilers

## Introduction

一个好的程序不仅要保证正确和具有良好的可读性，在很多情况下，让程序运行得快也是一个重要的考虑因素。本章我们将探讨如何使用几种不同类型的程序优化技术，使程序运行得更快。这意味着我们要理解优化编译器的能力和局限性，从而编写出编译器能够有效优化以转换成高效可执行代码的源代码。

## Goals and Capabilities of compiler optimization

一般来说，编译器优化的目的有：使指令数达到最小，避免等待内存和避免产生分支。

大多数编译器向用户提供了一些对它们所使用的优化的控制，如指定优化级别。在GCC中，以命令行选项“-Og”调用GCC是让其使用一组基本的优化，以选项“-O1”或更高的“-O2”和“-O3”会让它使用更大量的优化。这样做可以进一步提高程序的性能，但也可能增加程序的规模。

现代编译器会运用复杂精细的算法来确定一个程序中计算的是什么值，以及它们是如何被使用的。然后会利用一些机会来简化表达式，如在几个不同的地方使用同一个计算，以及降低一个给定的计算必须被执行的次数。接下来我们举例说明编译器都能为我们做出哪些优化。

## Constant Folding

任何常数表达式都可以被编译器直接计算出来，有时甚至不需要调用库函数。

例如：

```
int f(int num)
{
    return num + 4337651 * 2;
}

int mul_twice(int num)
{
    return f(f(num));
}
```

上面这段代码在“-O1”优化下产生的汇编代码如下：

```
f:
    lea      eax, [rdi+8675302]
    ret
mul_twice:
    lea      eax, [rdi+17350604]
    ret
```

可见有关常数的计算，编译器会直接将其优化为具体值。

甚至对于一些调用了库函数的常数计算指令，编译器仍然可以优化，例如：

```
int Length(int namelen)
{
    namelen = strlen("Harry Bovik");
    return namelen;
}
```

“-O1”优化结果如下：

```
length:
    mov      eax, 11
    ret
```

本质上，常量字符串的长度就是一个常数，因此仍然可以采用常量折叠的方法优化。

## Share Common Subexpressions

当函数中存在一些相同的计算时，编译器会将其优化为只计算一次，然后将所得结果在各表达式中

共享，以减少计算次数。下面是一个简单的例子：

```
int CSE (int num)
{
    return num * num + num * num;
}
```

“-O1”优化结果如下：

```
CSE:
imul    edi, edi
lea     eax, [rdi+rdi]
ret
```

很明显，在上面的函数体中有两次完全相同的乘法运算num\*num，而编译器将其优化为只计算一次，然后把计算结果共享。

## Optimization Example: Bubblesort

了解了编译器优化的主要方式，下面我们以冒泡排序程序为例来看看编译器的优化能力。

```
void bubbleSort(int *A,int n,int i,int j)
{
    for (i = n-1; i >= 1; i--)
    {
        for (j = 1; j <= i; j++)
        if (A[j] > A[j+1])
        {
            int temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

用伪代码表述的原始的汇编代码如下：

```

        i := n-1
L5: if i<1 goto L1
        j := 1
L4: if j>i goto L2
        t1 := j-1
        t2 := 4*t1
        t3 := A[t2] // A[j]
        t4 := j+1
        t5 := t4-1
        t6 := 4*t5
        t7 := A[t6] // A[j+1]
        if t3<=t7 goto L3

for (i = n-1; i >= 1; i--) {
    for (j = 1; j <= i; j++) {
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}

        t8 := j-1
        t9 := 4*t8
        temp := A[t9] // temp:=A[j]
        t10 := j+1
        t11 := t10-1
        t12 := 4*t11
        t13 := A[t12] // A[j+1]
        t14 := j-1
        t15 := 4*t14
        A[t15] := t13 // A[j]:=A[j+1]
        t16 := j+1
        t17 := t16-1
        t18 := 4*t17
        A[t18]:=temp // A[j+1]:=temp
L3: j := j+1
    goto L4
L2: i := i-1
    goto L5
L1:

```

**Instructions**  
**29 in outer loop**  
**25 in inner loop**

下面就是编译器大展身手的时刻，仅仅通过我们上面讲的减少冗余的方法，此程序最终优化的结果如下：

```

        i := n-1
L5: if i<1 goto L1
        t2 := 0
        t6 := 4
        t19 := i << 2
L4: if t6>t19 goto L2
        t3 := A[t2]
        t7 := A[t6]
        if t3<=t7 goto L3
        A[t2] := t7
        A[t6] := t3
L3: t2 := t2+4
        t6 := t6+4
        goto L4
L2: i := i-1
        goto L5
L1:

```

可以看到，程序优化后，外层循环指令数从29减少到15，内层循环指令数由25减少到9，效果显著。

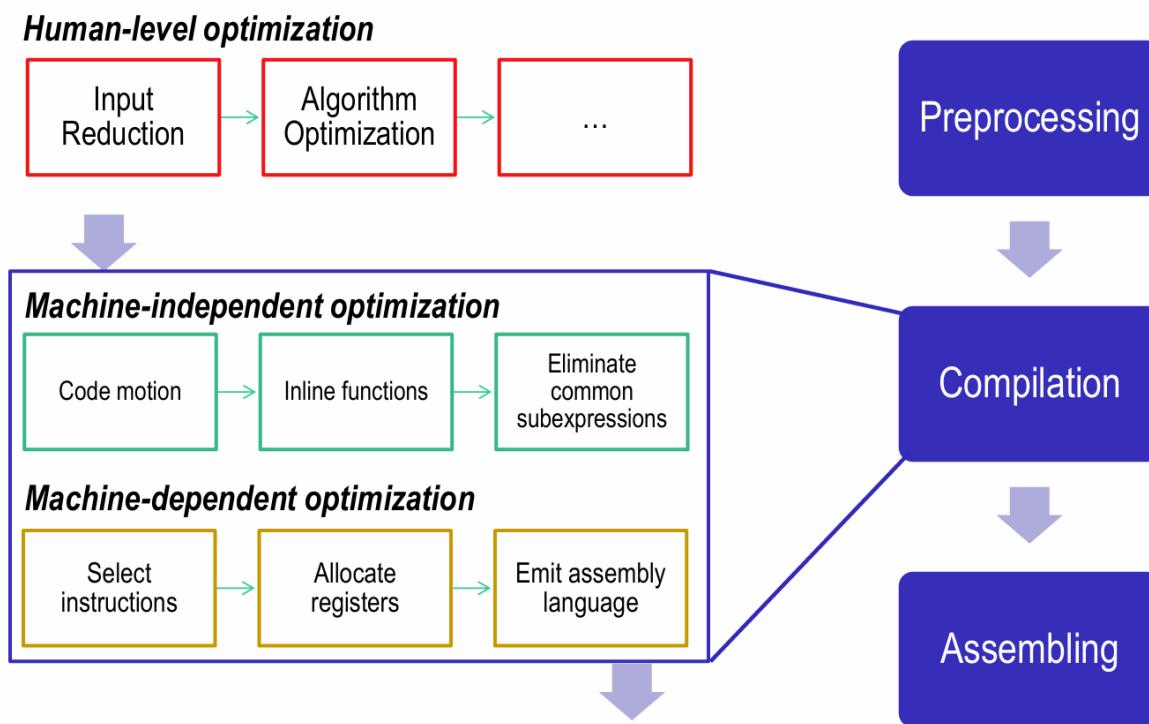
# Limits to compiler optimization

编译器优化也有很多局限性。

- 编译器优化无法改变算法的渐进复杂度。（只能优化常数，不过常数也可以产生很大的影响！）
- 编译器必须很小心地对程序只使用安全的优化，禁止对程序行为做出改变。所以在不确定程序员意图的情况下，编译器只能**保守**地优化。
- 每次只能分析一个函数（除非使用内联）。
- 无法预料运行时的输入情况。

## Multiple Levels of Optimizations

下图展示了不同层面的程序优化：



编译器对程序进行的优化分为机器无关优化（Machine-independent optimizations）和机器相关优化（Machine-dependent optimizations）。我们前面介绍的都属于机器无关优化，主要优化在

中间代码。由于编译器优化的局限性，程序员在编写程序的过程中也要进行代码优化。我们将在下一节详细介绍编译器优化受限情况下的代码优化方法。

---

© 2025. ICS Team. All rights reserved.

# Chapter 5.2 Optimization Blocker

前面我们提到，当编译器不确定程序员的意图时，它会十分保守地进行优化。所以为了使程序性能更优，程序员自己也需要显式地对程序进行优化。

## Procedure Calls

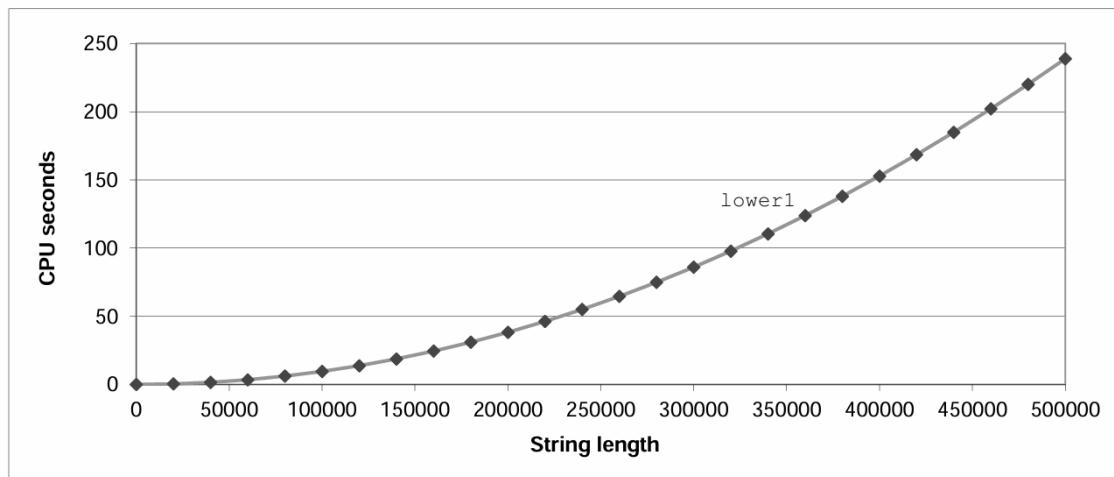
程序中的过程调用会带来意想不到的巨大开销，而且由于编译器的“保守性”，过程调用往往会妨碍大多数形式的程序优化。因此在编写程序的过程中，要减少不必要的过程调用。

下面以一个具体例子来说明，该代码实现了把字符串中所有大写字母变为小写字母：

```
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');
}
```

下图展示了随着字符串长度的增加，该函数运行时间的变化。

- Time quadruples when double string length
- Quadratic performance



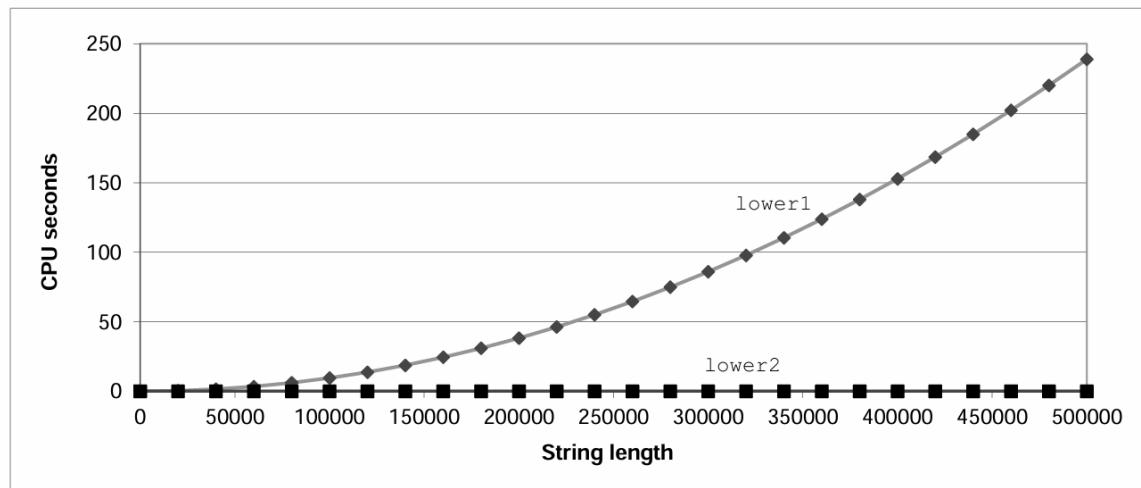
很明显，lower函数曲线图随字符串长度的增加上升地很陡峭，它的算法复杂度是 $O(n^2)$ 。这显然是不可接受的。

问题在于，我们实际上每一次循环都调用了strlen这个函数，所以产生了二次的运行时间。但事实上，字符串的长度是固定的，我们只需要在进入循环前算一次就可以了。因此改进代码如下：

```
void lower(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');
}
```

对比一下改进前后的算法性能：

- Time doubles when double string length
- Linear performance of lower2



这样一来，我们仅仅只是做了这么一个小小的改变，算法的运行时间就变成了线性的，性能得到了显著改进。

## Memory Aliasing

**内存别名使用** (memory aliasing) 是指两个指针可能指向同一个内存位置的情况，尽管程序员的本来意图并非如此，但是在保证安全优化的前提下，编译器必须假设不同的指针可能会指向内存中同一个位置。这使得编译器无法做出一些我们期望的优化。此时便又需要程序员显式地优化代码，以提高程序性能。

考虑下面这段代码，它实现了将矩阵a的每一行元素求和存储到向量b中：

```
void sum_rows1(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];
    }
}
```

对应汇编代码如下：

```
# sum_rows1 inner loop
.L4:
    movsd  (%rsi,%rax,8), %xmm0    # FP load
    addsd  (%rdi), %xmm0          # FP add
    movsd  %xmm0, (%rsi,%rax,8)   # FP store
    addq   $8, %rdi
    cmpq   %rcx, %rdi
    jne    .L4
```

从汇编中我们可以看出，编译器对 $b[i]$ 的操作是每次循环都要访问内存进行更新，这是我们非常不想看到的，因为内存操作很慢。产生这个结果的原因就是内存别名使用，编译器必须考虑对 $b[i]$ 的修改可能对整个程序的内存产生的影响。

因此，当我们去掉内存别名使用，用一个临时变量来替代直接赋值后：

```
void sum_rows2(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;
    }
}
```

汇编代码变成了下面的样子：

```
# sum_rows2 inner loop
.L10:
    addsd    (%rdi), %xmm0 # FP load + add
    addq    $8, %rdi
    cmpq    %rax, %rdi
    jne     .L10
```

是不是清爽多了！现在就不用再存储每一次的值到内存了。因此，在程序中避免内存别名使用也是一个提升程序性能的好方法。

---

© 2025. ICS Team. All rights reserved.

# Chapter 5.3 Understanding Modern Processors

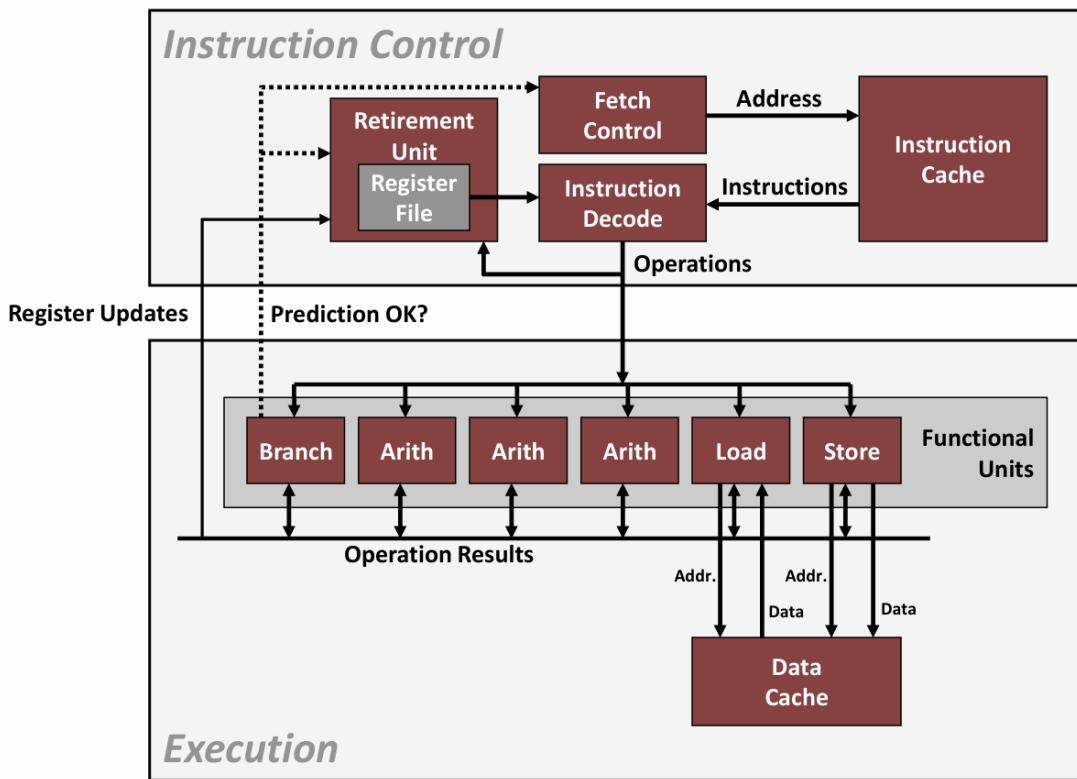
我们前面两小节介绍的都是不依赖于目标机器特性的优化（Machine-independent Optimization），这些优化只是简单地降低了过程调用的开销，以及消除了一些重大的妨碍优化的因素。要想进一步提高性能，我们必须考虑利用处理器微体系结构的优化。我们接下来介绍的基于处理器底层设计的代码优化方法（Machine-dependent Optimization）在很大一类处理器上都能实现整体性能提高，虽然具体性能结果不一定完全相同，但操作和优化的通用原则对各种各样的机器都适用。

现代处理器复杂而精妙的微体系结构使得在实际操作中，处理器是同时对多条指令求值的，这种现象称为**指令级并行**（Instruction-level Parallelism）。相对应的，我们发现存在两种下界描述了程序的最大性能。当一系列操作必须按照严格顺序执行时，就会遇到**延迟界限**（Latency Bound），当代码中的数据相关限制了处理器利用指令级并行的能力时，延迟界限就成为程序性能的限制。而**吞吐量界限**（Throughput Bound）则刻画了处理器功能单元的原始计算能力，它是程序性能的终极限制。

本课程不涉及现代微处理器的详细设计，但大致了解这些微处理器运行的原则就可以理解它们如何实现指令级并行，以便我们更好的利用指令级并行优化程序性能。

## Modern CPU Design

下图是现代微处理器的一个简化示意图：



这种类型的处理器可以在每个时钟周期执行多个操作，称为**超标量**（superscalar），且执行是乱序的，即指令执行的顺序不一定与它们在机器级程序中的顺序一致。

从图中我们可以看到，整个设计有两个主要部分：

- **指令控制单元** (Instruction Control Unit, ICU)：负责从内存中读取指令序列，并根据这些指令序列生成一组针对程序数据的基本操作。
- **执行单元** (Execution Unit, EU)：负责执行这些操作。

通常情况下，ICU会在当前正在执行的指令很早之前从指令高速缓存 (instruction cache) 中读取指令，这样它才有足够的时间对指令译码，并把操作发送到EU。而EU通常每个时钟周期会接收多个操作，它们会被分派到一组**功能单元**中去执行。这些功能单元专门用来处理不同类型的操作。而通过利用功能单元的一些特性就可以实现指令级并行。

## Functional Unit

以Intel Core i7 Haswell参考机为例，它有8个功能单元（其中整数运算指的是加法、位级操作和移

位等基本操作) :

0. 整数运算、浮点乘、整数和浮点数除法、分支
1. 整数运算、浮点加、整数乘、浮点乘
2. 加载、地址计算
3. 加载、地址计算
4. 存储
5. 整数运算
6. 整数运算、分支
7. 存储、地址计算

从中我们可以发现, 功能单元的各种组合具有同时执行多个同类型操作的潜力: 它有4个单元能执行整数操作, 2个单元能执行加载操作, 2个单元能执行浮点乘法。这种特性意味着我们可以实现多条同类型指令并行处理。

同时, 各种类型的操作有不同的性能, 下图提供了参考机各操作的延迟、发射时间和容量特性。

| Operation      | Integer |       |          | Floating point |       |          |
|----------------|---------|-------|----------|----------------|-------|----------|
|                | Latency | Issue | Capacity | Latency        | Issue | Capacity |
| Addition       | 1       | 1     | 4        | 3              | 1     | 1        |
| Multiplication | 3       | 1     | 1        | 5              | 1     | 2        |
| Division       | 3–30    | 3–30  | 1        | 3–15           | 3–15  | 1        |

---

注:

**延迟** (latency) 表示完成运算所需要的总时间。**发射时间** (issue time) 表示两个连续的同类型的运算之间需要的最小时钟周期数。**容量** (capacity) 表示能够执行该运算的功能单元的数量。

---

我们看到, 从整数运算到浮点运算, 延迟增加。同时, 加法和乘法运算的发射时间为1, 即在每个时钟周期, 处理器都可以开始一条新的这样的运算。这种很短的发射时间是通过使用**流水线**实现的。流水线化的功能单元实现为一系列的阶段, 每个阶段完成一部分的运算。但只有当要执行的运算是连续的、逻辑上独立的时候才能利用这种功能。

利用功能单元的流水线化和同类型操作指令并行的特性, 我们就能够突破延迟界限, 实现更进一步的程序性能优化。

---

© 2025. ICS Team. All rights reserved.

# Chapter 5.4 Instruction-Level Parallelism

本节我们会基于一个合并运算的代码，使用不同的方法对其进行一系列的优化，并使用**每元素的周期数**（Cycles Per Element, CPE）作为表示程序性能的度量标准。

该代码使用某种运算，将一个向量中所有的元素合并为一个值，其初始实现如下：

```
void combine1(vec_ptr v, data_t *dest)
{
    long i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++)
    {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

其中常数IDENT和操作OP由编译时的不同定义来决定：

- `#define IDENT 0` 和 `#define OP +` 表示对向量元素求和。
- `#define IDENT 1` 和 `#define OP *` 表示计算向量元素的乘积。

通过进行我们前面介绍的最基础的机器无关优化，包括编译器自行优化，消除循环的低效率（代码移动）和减少内存引用，我们可以很轻松地将代码优化为下面这个版本：

```
void combine4(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++)
        acc = acc OP d[i];
    *dest = acc;
}
```

程序性能的变化如下：

| Method       | Integer |       | Double FP |       |
|--------------|---------|-------|-----------|-------|
| Operation    | Add     | Mult  | Add       | Mult  |
| Combine1 –O1 | 10.12   | 10.12 | 10.17     | 11.14 |
| Combine4     | 1.27    | 3.01  | 3.01      | 5.01  |

事实上，经过基础优化的代码已经接近延迟界限，参考上一节我们介绍的参考机各操作的延迟：

| Integer<br>Add | Integer<br>Multiply | Single/Double FP<br>Add | Single/Double FP<br>Multiply |
|----------------|---------------------|-------------------------|------------------------------|
| 1              | 3                   | 3                       | 5                            |

对比之下，细心的你可能会产生一个疑问：为什么整数乘法（3.01 vs 3）、浮点数加法（3.01 vs 3）和乘法（5.01 vs 5）都逼近延迟界限，但整数加法（1.27 vs 1）似乎还留有余地呢？这和循环开销有关，一般来说，循环次数越多，开销越大。由于整数乘法和浮点数运算循环本身开销较大，循环带来的额外开销对它们影响并不明显；但由于整数加法本身开销小，所以循环的额外开销就会显著地影响到整数加法的性能。因此我们采用循环展开的方法进行优化。

## Loop Unrolling

循环展开是一种程序变换，通过增加每次迭代计算的元素的数量，减少循环的迭代次数。

针对前例，我们可以使用2x1循环展开，即每次循环处理两个元素。

```

void combine5(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2)
    {
        acc = (acc OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++)
    {
        acc = acc OP d[i];
    }
    *dest = acc;
}

```

让我们再看一看程序性能：

| Method        | Integer |      | Double FP |      |
|---------------|---------|------|-----------|------|
| Operation     | Add     | Mult | Add       | Mult |
| Combine4      | 1.27    | 3.01 | 3.01      | 5.01 |
| Unroll 2x1    | 1.01    | 3.01 | 3.01      | 5.01 |
| Latency Bound | 1.00    | 3.00 | 3.00      | 5.00 |

现在所有的操作都达到了延迟界限！

但是这还不够，我们的目标是突破延迟界限，达到吞吐量界限。循环展开无法突破延迟界限的原因是，在每次迭代中存在必须顺序执行的两个乘法。因此要想突破此界限，必须要突破顺序相关，提高并行性。

## Reassociation Transformation

**重新结合变换**是一种打破顺序相关从而使性能提高到延迟界限之外的方法。

在本例中，我们仅将combine5中的语句  $acc = (acc \text{ OP } d[i]) \text{ OP } d[i+1]$ ；修改为  $acc = acc \text{ OP } (d[i] \text{ OP } d[i+1])$ 。即改变了括号的位置，如下：

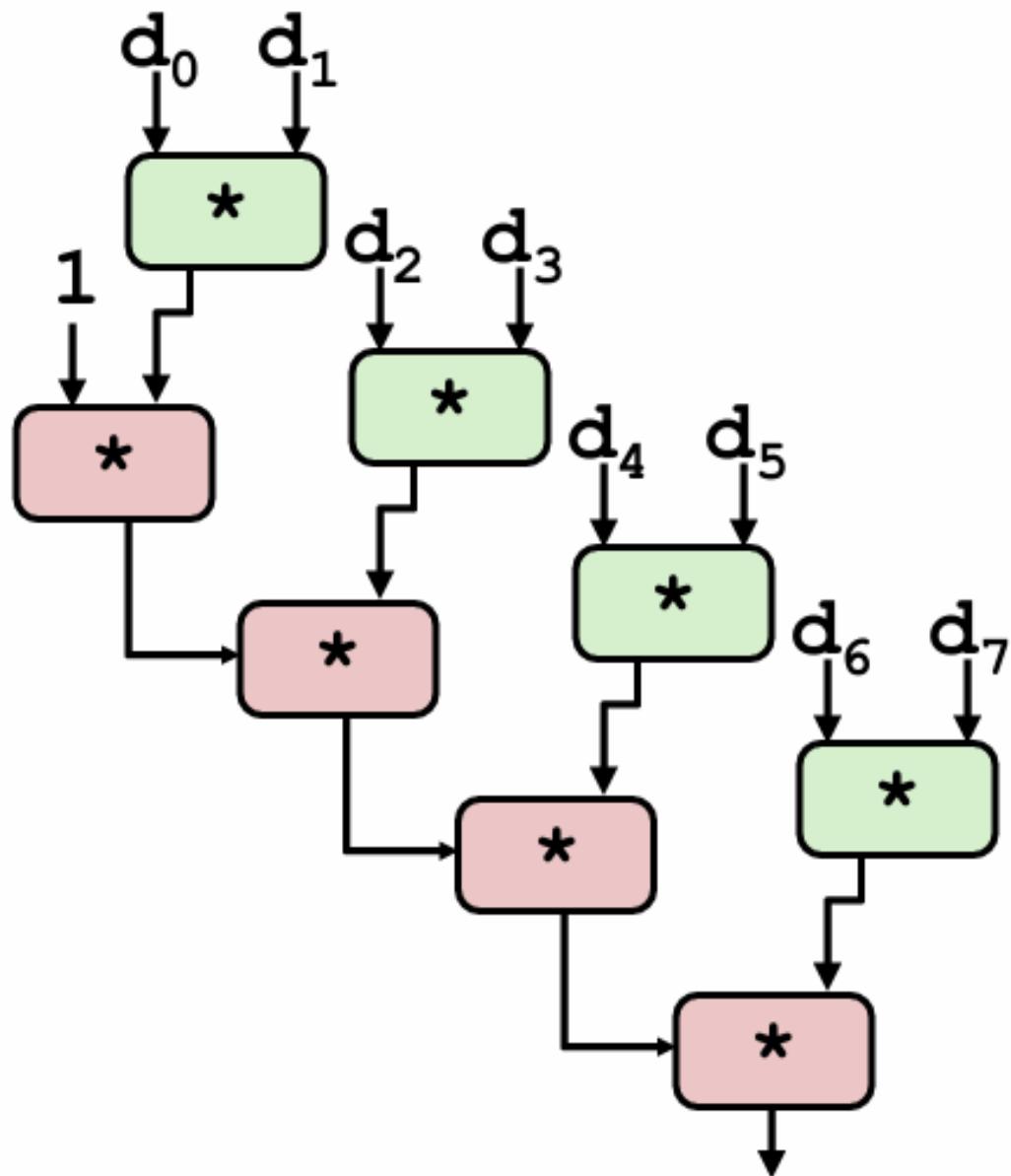
```
void combine6(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t acc = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2)
        acc = acc OP (d[i] OP d[i+1]);
    /* Finish any remaining elements */
    for (; i < length; i++)
        acc = acc OP d[i];
    *dest = acc;
}
```

你可能想问，只是括号的位置变了一下，这能有什么影响？但是当我们测量CPE时，得到令人吃惊的结果：

| Method           | Integer |      | Double FP |      |
|------------------|---------|------|-----------|------|
| Operation        | Add     | Mult | Add       | Mult |
| Combine4         | 1.27    | 3.01 | 3.01      | 5.01 |
| Unroll 2x1       | 1.01    | 3.01 | 3.01      | 5.01 |
| Unroll 2x1a      | 1.01    | 1.51 | 1.51      | 2.51 |
| Latency Bound    | 1.00    | 3.00 | 3.00      | 5.00 |
| Throughput Bound | 0.50    | 1.00 | 1.00      | 0.50 |

采用2x1a展开的combine6中，整数加的性能和采用2x1展开的combine5性能相同，而其他三种情况达到了2x1展开的两倍，突破了延迟界限！

下图说明了产生该巨大影响的原因：



括号位置变化后，下一次迭代中 ( $d[i] \text{ OP } d[i+1]$ ) 可以提前进行，不需要等待前一次迭代的累积值，即突破了顺序相关性。因此，最小可能的CPE减少为原来的一半。

可以看到，重新结合变换能够减少计算中关键路径上操作的数量，通过更好地利用功能单元的流水线能力得到更好的性能。

## Multiple Accumulators

另一种方式是引入多个累积变量。由于执行加法和乘法的功能单元是完全流水线化的，理论上它们可以每个时钟周期开始一个新操作。但是在combine5中，由于我们将累积值放在一个单独的变量acc中，在前面的计算完成之前，都无法计算acc的新值。虽然计算acc新值的功能单元能够每个时钟周期开始一个新的操作，但是由于这种顺序依赖，它只能每L个周期开始一条新操作（L是合并操作的延迟）。

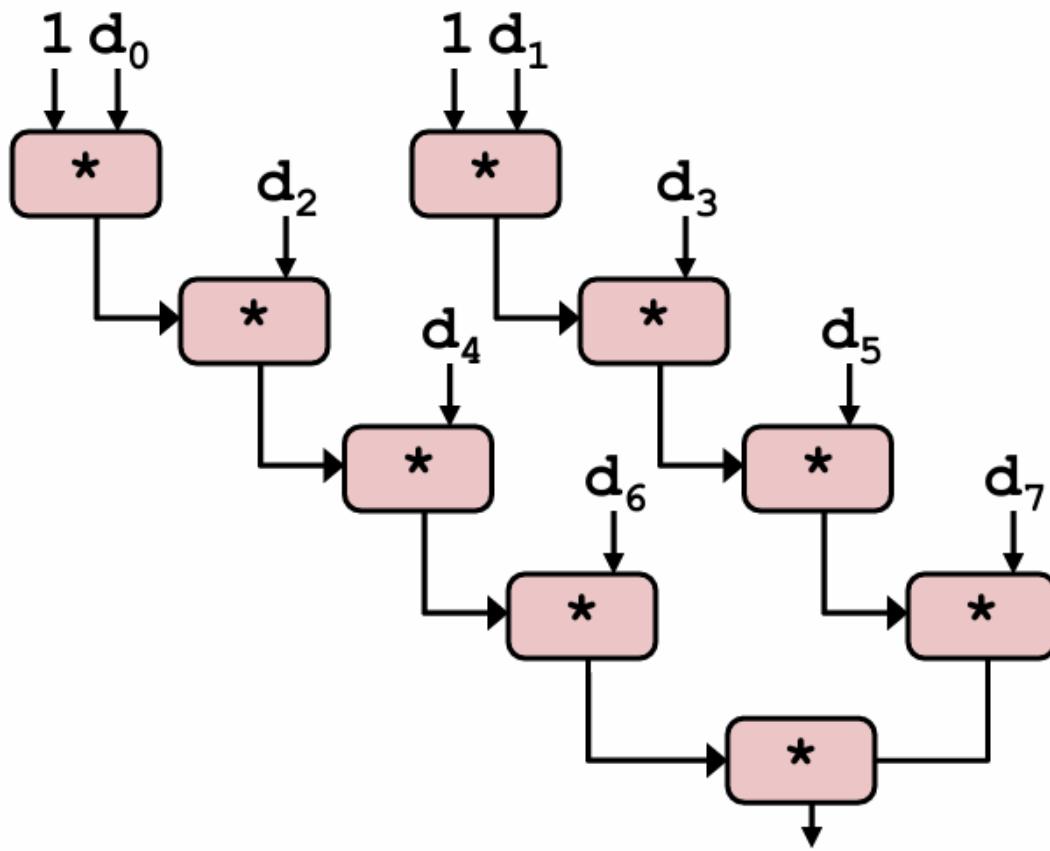
在本例中，我们引入两个累积变量来打破这种顺序相关，将一组合并运算分割成两个部分，并在最后合并结果，并称之为2x2展开。

```
void combine7(vec_ptrv, data_t*dest)
{
    long i;
    long length=vec_length(v);
    long limit=length-1;
    data_t* data=get_vec_start(v);
    data_t acc0=IDENT;
    data_t acc1=IDENT;

    /*Combine 2 elements at a time*/
    for(i=0;i<limit;i+=2)
    {
        acc0=acc0 OP data[i];
        acc1=acc1 OP data[i+1];
    }

    /*Finish any remaining elements*/
    for(;i<length;i++)
        acc0=acc0 OP data[i];
    *dest=acc0 OP acc1;
}
```

采用2x2展开的combine7版本的实际运行过程如下：



我们引入的两个累积变量使它能够同时利用两个加载功能单元，处理器不再需要延迟一个加法或乘法操作以等待前一个操作完成。这实际上形成了两条独立的操作流，一条处理偶数索引的计算，一条处理奇数索引的计算。

我们的程序性能也因此得到了更进一步的提升：

| Method           | Integer |      | Double FP |      |
|------------------|---------|------|-----------|------|
| Operation        | Add     | Mult | Add       | Mult |
| Combine4         | 1.27    | 3.01 | 3.01      | 5.01 |
| Unroll 2x1       | 1.01    | 3.01 | 3.01      | 5.01 |
| Unroll 2x1a      | 1.01    | 1.51 | 1.51      | 2.51 |
| Unroll 2x2       | 0.81    | 1.51 | 1.51      | 2.51 |
| Latency Bound    | 1.00    | 3.00 | 3.00      | 5.00 |
| Throughput Bound | 0.50    | 1.00 | 1.00      | 0.50 |

这一次，除了整数乘、浮点数加乘和2x1a展开的combine6版本性能相同外，整数加法性能又得到

了提升。

重复实验结果表明，采用上述两种方法，我们的合并函数最终能达到下面的性能：

| Method           | Integer |      | Double FP |      |
|------------------|---------|------|-----------|------|
|                  | Add     | Mult | Add       | Mult |
| Best             | 0.54    | 1.01 | 1.01      | 0.52 |
| Latency Bound    | 1.00    | 3.00 | 3.00      | 5.00 |
| Throughput Bound | 0.50    | 1.00 | 1.00      | 0.50 |

即使用以上优化技术，程序的CPE已经接近于吞吐量界限。

---

© 2025. ICS Team. All rights reserved.

# Chapter 5.5 Branch Predictions

现代处理器的工作远超前于当前正在执行的工作，从内存读指令，译码指令，以确定在什么操作数上执行什么操作。只要指令遵循的是一种简单的顺序，那么这种指令流水线化就能很好地工作。当遇到分支时，处理器必须猜测分支该往哪个方向走。我们这里主要讨论条件分支，即预测是否会选择分支。

分支预测错误处罚的代价很高，因此要求预测的准确率要尽可能高。事实上，现代处理器采用一种简单的预测方式，它的准确度可以达到95%：

1. 向后 (backward) 跳转的指令通常是循环，因此预测采取。
2. 向前 (forward) 跳转的指令通常是if条件，因此预测不采取。

当然，仅仅依靠机器的分支预测并不能保证程序性能良好，程序员本身也要尽量写分支较少或有利于分支预测准确性的代码。

## Transform Branches

有时分支可以通过一些巧妙的运算变换为顺序执行的代码，例如：

```
for (int c=0; c < size; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}
```

通过位运算等技巧，分支完全可以消除：

```
int t = (data[c] -128) >> 31;
sum += ~t & data[c];
```

## Make Branch More Predictable

同时，针对上例，如果不采用分支转化的方法，我们也要尽量遵循处理器的分支预测方法，设置更容易预测的读取数据。例如：

我们可以把输入数据从这样的

data[] = 226, 185, 125, 158, 198, 144, 217, 79, 202, 118, 14, ... (完全乱序，毫无规律可循😢)

改成这样的

data[] = 0, 1, 2, 3, 4, ... 126, 127, 128, 129, 130, ... 250, ... (增序序列，预测轻松多了~)

## Conditional Moves

编译器也会对分支进行一些优化，典型的操作是用cmov条件传送指令替换分支。例如：

```
int absdiff(int x, int y)
{
    int result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

其优化后的汇编代码如下：

```
absdiff:
    mov     edx, edi
    sub     edx, esi
    mov     eax, esi
    sub     eax, edi
    cmp     edi, esi
    cmovg  eax, edx
```

但是，这种方式并不一定都有效，因为条件传送指令需要一开始就将两种情况的结果计算出来，当不需要被执行的分支计算量很大时，这样做显然是不划算的。

---

这一章的内容到这里就结束了，相信大家现在对于程序性能优化有了更深刻的认识！🎉

---

© 2025. ICS Team. All rights reserved.

# Chapter 6 The Memory Hierarchy

---

© 2025. ICS Team. All rights reserved.

# Chapter 6.1 Storage Technologies

## Introduction

到目前为止，在对系统的研究中，我们都是依赖一个简单的计算机系统模型，即存储器系统是一个线性的字节数组，而CPU能够在常数时间内访问每个存储器位置。虽然它迄今为止似乎仍然有效，但它并不能反映现代系统实际工作的方式。

实际上，存储器系统是一个具有不同容量、成本和访问时间的存储器设备的层次结构。它的整体效果是一个大的存储器池，其成本与层次结构底层最便宜的存储设备相当，但是却以接近于层次结构顶部存储设备的高速率向程序提供数据。

本章将带领大家进入到存储器层次结构的世界，深入了解相应的存储技术、分析程序的局部性，以及学习如何改进你的程序性能。随着学习的深入，你将体会到存储器层次结构的设计是多么的优美与精妙，它突破了物理结构的限制，从而导致物理学家失业。

## The Memory Abstraction

数据流通过称为**总线**（bus，是一组并行的导线，能携带地址、数据和控制信号）的共享电子电路在处理器和DRAM主存之间来来回回。每次CPU和主存之间的数据传送都是通过一系列步骤来完成的，这些步骤称为**总线事务**（bus transaction）。读事务（read transaction）从主存传送数据到CPU，写事务（write transaction）从CPU传送数据到主存。

### Read Transaction

当CPU执行一个如下加载操作时：

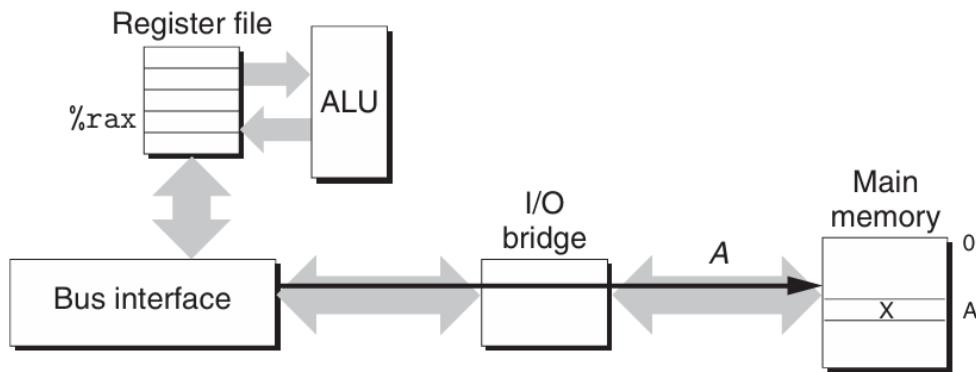
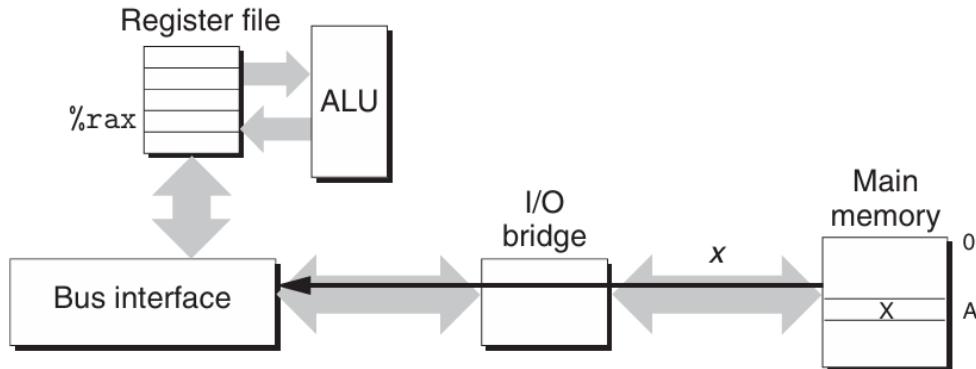
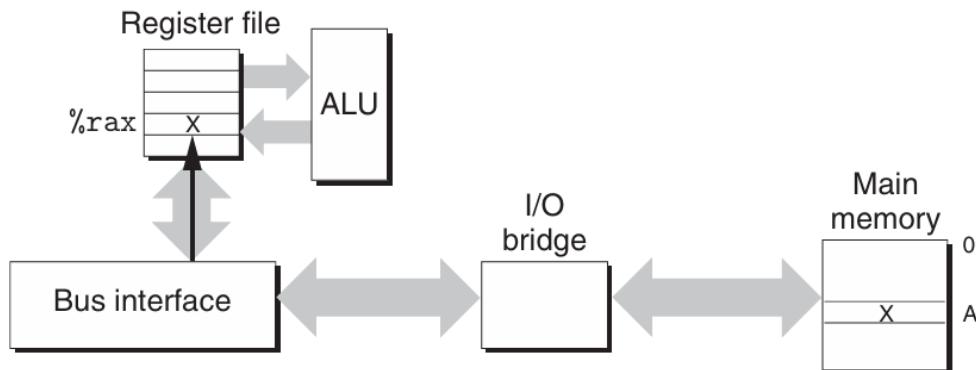
```
movq A,%rax
```

这里，地址A的内容被加载到寄存器%rax中。CPU芯片上的**总线接口**（bus interface）的电路在总线上发起读事务。

读事务由三个步骤组成：

1. CPU将地址A放在系统总线上, I/O桥将信号传递到内存总线。
2. 主存从内存总线读出地址, 从DRAM取出数据字, 并将数据写到内存总线。I/O桥将内存总线信号翻译成系统总线信号, 然后沿着系统总线传递。
3. CPU从总线上读数据, 并将数据复制到寄存器%rax。

该过程如下图所示：

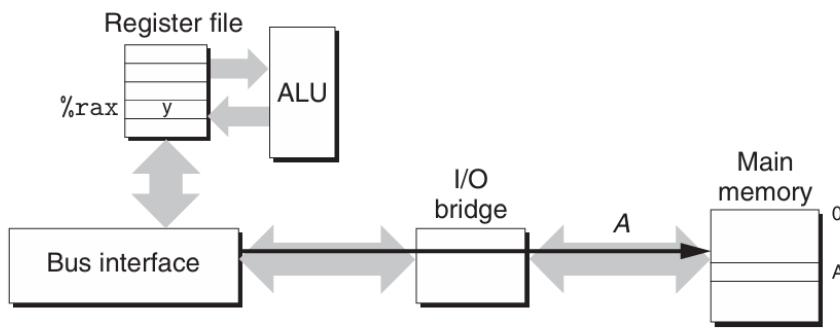
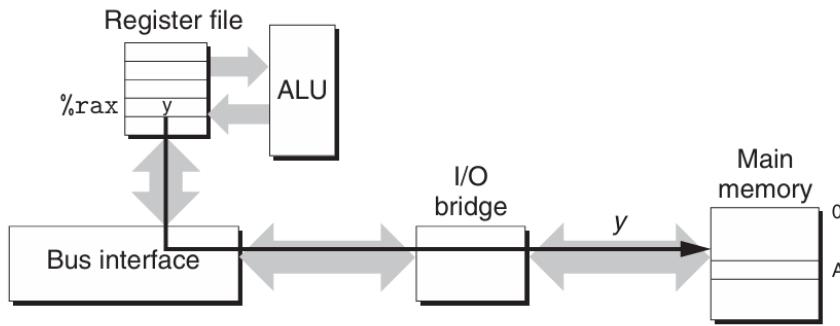
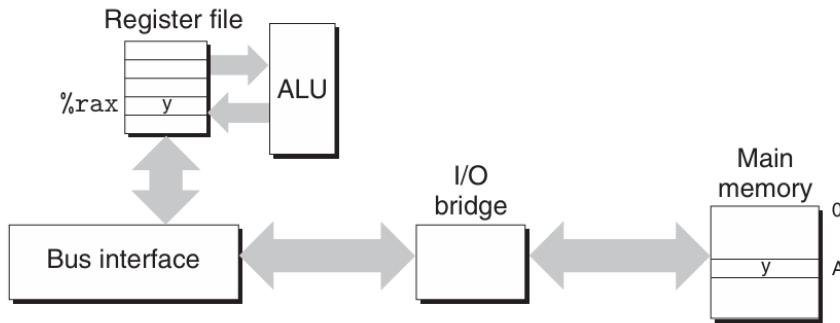
(a) CPU places address  $A$  on the memory bus.(b) Main memory reads  $A$  from the bus, retrieves word  $x$ , and places it on the bus.(c) CPU reads word  $x$  from the bus, and copies it into register  $\%rax$ .

## Write Transaction

反过来，当CPU执行如下的存储操作时：

```
movq %rax, A
```

这里，寄存器 $\%rax$ 的内容被写到地址 $A$ ，CPU发起写事务。同样由三个基本步骤，如下图所示：

(a) CPU places address  $A$  on the memory bus. Main memory reads it and waits for the data word.(b) CPU places data word  $y$  on the bus.(c) Main memory reads data word  $y$  from the bus and stores it at address  $A$ .

## Random-Access Memory

随机访问存储器（Random-Access Memory, RAM）分为两类，静态RAM（SRAM）和动态RAM（DRAM）。

- SRAM将每个位存储在一个双稳态的（bistable）存储器单元里，每个单元用一个六晶体管电路来实现。由于其双稳态特性，只要有电，它就会永远保持它的值。即使有干扰来扰乱电压，当干扰消除时，电路就会恢复到稳定值。
- DRAM将每个位存储为对一个电容的充电。它的每个单元由一个电容和一个访问晶体管组

成，所以可以制造得很密集，但是DRAM存储器单元对干扰非常敏感，当电容的电压被扰乱后，它就永远不能恢复了。

|      | Trans. Per bit | Access Time | Needs Fresh? | Needs EDC? | Cost[2023] | Applications |
|------|----------------|-------------|--------------|------------|------------|--------------|
| SRAM | 6 or 8         | 1x          | No           | Maybe      | 100x       | Cache memory |
| DRAM | 1              | 10x         | Yes          | Yes        | 1x         | Main memory  |

总的来说，SRAM比DRAM更快，但也更贵。SRAM用来作为高速缓存存储器，既可以在CPU芯片上，也可以在片外。DRAM用来作为主存以及图形系统的帧缓冲区。

### Info

DRAM有许多种增强版，它们都是基于传统的DRAM单元，进行一些接口逻辑和I/O的优化，提高访问DRAM基本单元的速度。如同步DRAM（SDRAM），双倍数据速率同步DRAM（DDR SDRAM）等。

## Disk Storage

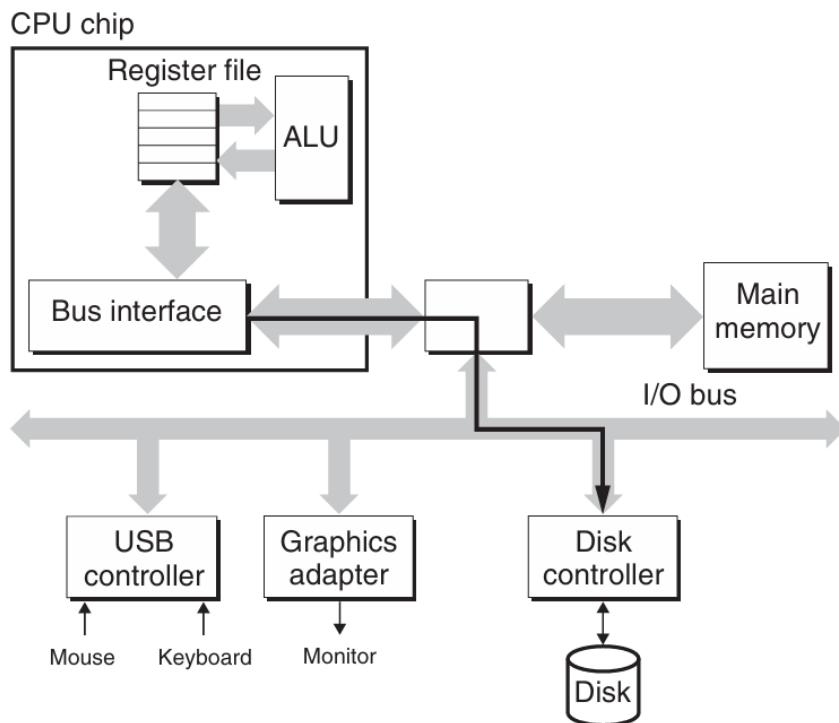
磁盘是广为应用的保存大量数据的存储设备，它存储数据的数量级可达几百到几千GB (gigabyte)，不过，从磁盘上读信息的时间比DRAM慢了10万倍，比SRAM慢了100万倍。

- **磁盘构造**：磁盘是由**盘片** (platter) 构成的。每个盘片有两面，称为**表面** (surface)，表面覆盖着磁性记录材料。盘片中央有一个可以旋转的**主轴** (spindle)，它使得盘片以固定的旋转速率旋转。磁盘包含一个或多个这样的盘片，并封装在一个密封容器内。典型的磁盘表面是由一组称为**磁道** (track) 的同心圆组成的，每个磁道被划分为一组**扇区** (sector)。每个扇区包含相等数量的数据位。
- **磁盘容量**：磁盘容量由以下技术因素决定：记录密度 (recording density)，磁道密度 (track density) 和面密度 (areal density)。
- **磁盘操作**：磁盘用**读/写头** (read/write head) 来读写存储在磁性表面的位，而读写头连接到一个**传动臂** (actuator arm) 一端。通过沿着半径轴前后移动这个传动臂，驱动器可以将

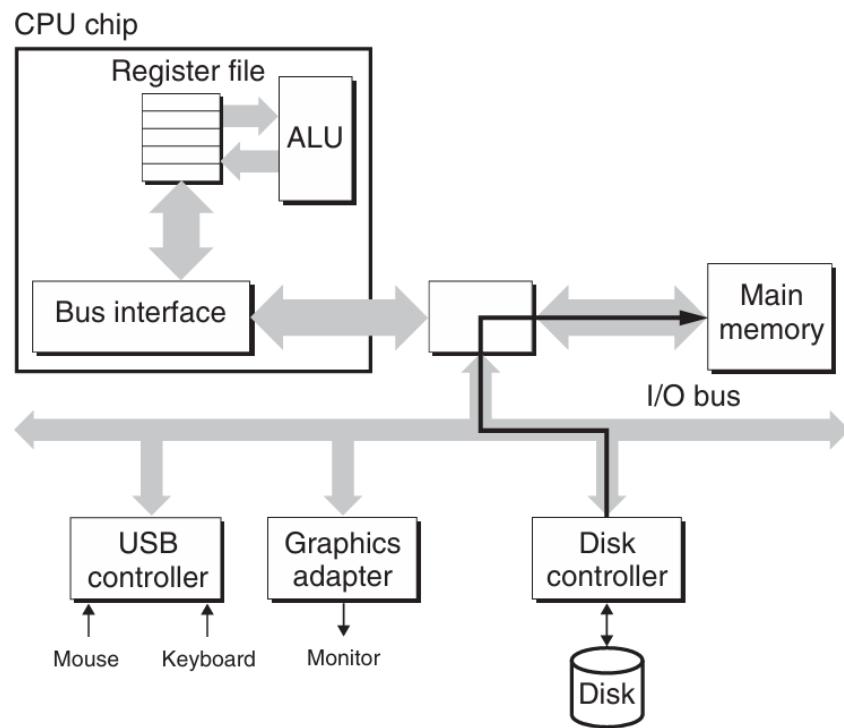
读/写头定位在盘面的任何磁道上。这样的机械运动称为 **寻道** (seek)。一旦读/写头定位到了期望的磁道上，那么当磁道上的每个位通过它的下面时，读/写头可以感知到这个位的值（读该位），也可以修改这个位的值（写该位）。磁盘以扇区大小的块来读写数据。对扇区的访问时间 (access time) 有三个主要的部分，分别是寻道时间 (seek time)，旋转时间 (rotational latency) 和传送时间 (transfer time)。

- 访问磁盘：CPU使用一种称为**内存映射I/O** (memory-mapped I/O) 的技术来向I/O设备发射命令。在使用内存映射I/O的系统中，地址空间中有一块地址是为与I/O设备通信保留的。每个这样的地址称为一个I/O端口 (I/O port)。当一个设备连接到总线时，它与一个或多个端口相关联。当磁盘控制器收到来自CPU的读指令后，它将逻辑块号翻译为一个扇区地址，读该扇区的内容，然后将这些内容直接传送到主存，不需要CPU的干涉，即**直接内存访问下的数据传送** (DMA transfer)。在DMA传送完成，磁盘扇区的内容被安全地储存在主存中以后，磁盘控制器通过给CPU发送一个中断信号来通知CPU。这使得CPU暂停它当前正在做的工作，跳转到一个操作系统例程。这个程序会记录下I/O已经完成，然后将控制返回的CPU被中断的地方。

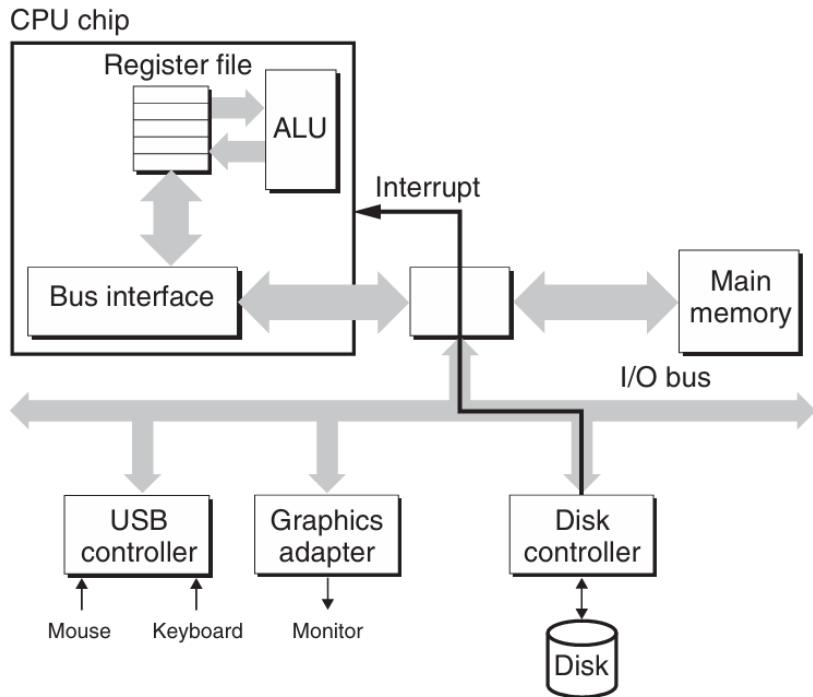
磁盘访问过程如下图所示：



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

## Solid State Disks

**固态硬盘 (SSD)** 是一种基于**闪存** (flash memory, 一类非易失性存储器) 的存储技术，在某些情况下成为传统旋转磁盘的替代产品。

一个SSD封装由一个或多个闪存芯片和**闪存翻译层** (flash translation layer) 组成，闪存芯片替代传统旋转磁盘中的机械驱动器，而闪存翻译层是一个硬件设备，扮演与磁盘控制器相同的角色，将对逻辑块的请求翻译成对底层物理设备的访问。

比起旋转磁盘，SSD有很多优点。它的随机访问时间比旋转磁盘更快，能耗更低，同时也更结实。

(现在的电脑可再也不会晃着晃着就晃坏了 )但是，SSD的缺点是在反复写之后容易磨损（但其实要很多年才会磨损坏.....），并且它的价格也较贵（不过现在SSD和旋转磁盘的价格差越来越小了.....）。

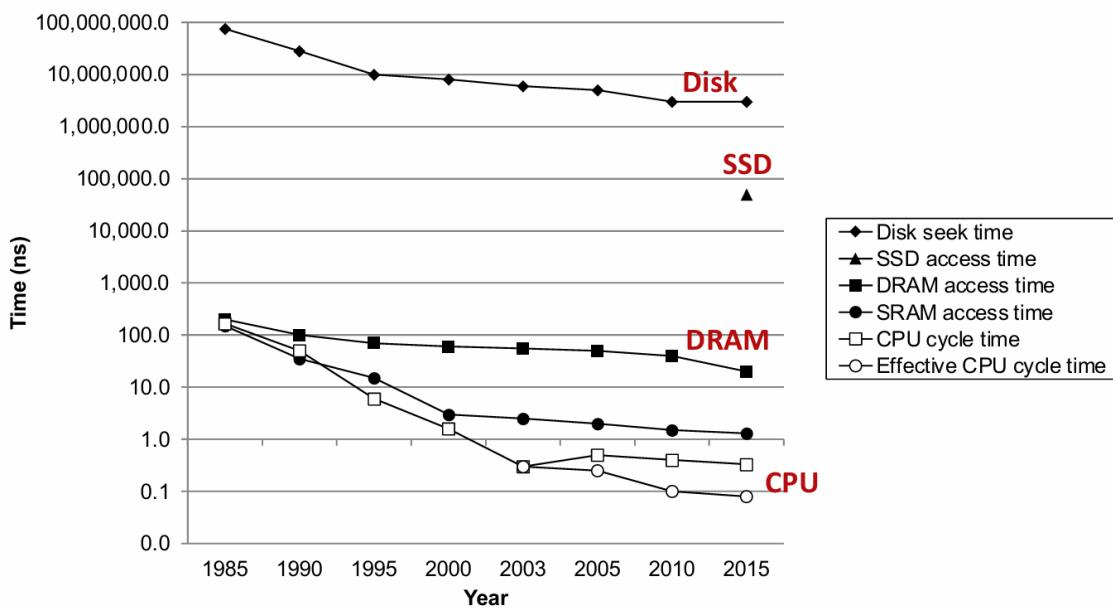
这么看来，SSD优点还是远大于缺点的。如今，SSD在便携音乐设备中已经完全取代了旋转磁盘，在笔记本电脑中也越来越多地作为硬盘的替代品，甚至在台式机和服务器中也开始出现了。

## Storage Technology Trends

从以上我们对存储技术的讨论中，可以总结出几个重要的思想：

- 不同的存储技术有不同的价格和性能折中。
- 不同存储技术的价格和性能属性以截然不同的速率变化着。
- DRAM和磁盘的性能滞后于CPU的性能。

从下图我们可以清楚地看出磁盘、DRAM和CPU速度之间逐渐增大的差距：



不过，办法总比困难多！现代计算机频繁地使用基于SRAM的高速缓存来弥补处理器-内存之间的差距，这种方法之所以可行是因为应用程序的一个称为**局部性** (locality) 的基本属性，我们将在下一节讨论这个问题。

---

© 2025. ICS Team. All rights reserved.

# Chapter 6.2 Locality

一个编写良好的计算机程序常常具有良好的**局部性** (locality)。也就是说，它们倾向于引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身。这种倾向性称为**局部性原理** (principle of locality)。

局部性通常有两种不同的形式：**时间局部性** (temporal locality) 和**空间局部性** (spatial locality)。

- 时间局部性：被引用过一次的内存位置很可能在不久的将来再被多次访问。
- 空间局部性：如果一个内存位置被引用了一次，那么程序很可能在不久的将来引用附近的一个内存位置。

## Locality of References to Program Data

考虑下面的一个简单函数：

```
int sumvec(int v[N])
{
    int i,sum=0;
    for(i=0;i<N;i++)
    {
        sum+=v[i];
    }
    return sum;
}
```

这是向量v的引用模式：

| Address      | 0     | 4     | 8     | 12    | 16    | 20    | 24    | 28    |
|--------------|-------|-------|-------|-------|-------|-------|-------|-------|
| Contents     | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ |
| Access order | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     |

可以看到，向量v的元素是被顺序读取的。因此，对于变量v，函数有很好的空间局部性，但是时间

局部性很差，因为每个向量元素只被访问一次。（因为对于循环体中的每个变量，这个函数要么有很好的时间局部性，要么有很好的空间局部性）所以我们可以说明，sumvec函数有良好的局部性。

像sumvec这样顺序访问一个向量每个元素的函数，称其具有**步长为1的引用模式**（stride-1 reference pattern）（相对于元素的大小）。在一个连续向量中，每隔k个元素进行访问，就称为**步长为k的引用模式**。一般而言，步长越大，空间局部性越差。

下面是一个例子：

```
int sumarrayrows(int a[M][N])
{
    int i,j,sum=0;
    for(i=0;i<M;i++)
    {
        for(j=0;j<N;j++)
        {
            sum+=a[i][j];
        }
    }
    return sum;
}
```

它的数组a的引用模式：

| Address      | 0        | 4        | 8        | 12       | 16       | 20       |
|--------------|----------|----------|----------|----------|----------|----------|
| Contents     | $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
| Access order | 1        | 2        | 3        | 4        | 5        | 6        |

该函数具有步长为1的引用模式，显然具有良好的空间局部性。

如果我们稍加改动，例如：

```

int sumarrayrows(int a[M][N])
{
    int i,j,sum=0;
    for(j=0;j<N;j++)
    {
        for(i=0;i<M;i++)
        {
            sum+=a[i][j];
        }
    }
    return sum;
}

```

它的数组a的引用模式变成了：

| Address      | 0        | 4        | 8        | 12       | 16       | 20       |
|--------------|----------|----------|----------|----------|----------|----------|
| Contents     | $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
| Access order | 1        | 3        | 5        | 2        | 4        | 6        |

显然函数的空间局部性变得很差。虽然我们只是交换了i和j的循环，但是这却导致函数按照**列顺序**来扫描数组，由于C数组在内存中是按照行顺序来存放的，结果就得到了步长为N的引用模式。

## Locality of Instruction Fetches

因为程序指令是存放在内存中的，CPU必须取出这些指令。所以我们也能够评价一个程序关于取指令的局部性。例如，上例sumvec函数for循环体里的指令是按照连续的内存顺序执行的，因此循环具有良好的空间局部性。同时，因为循环体会被执行多次，所以它也具有良好的时间局部性。

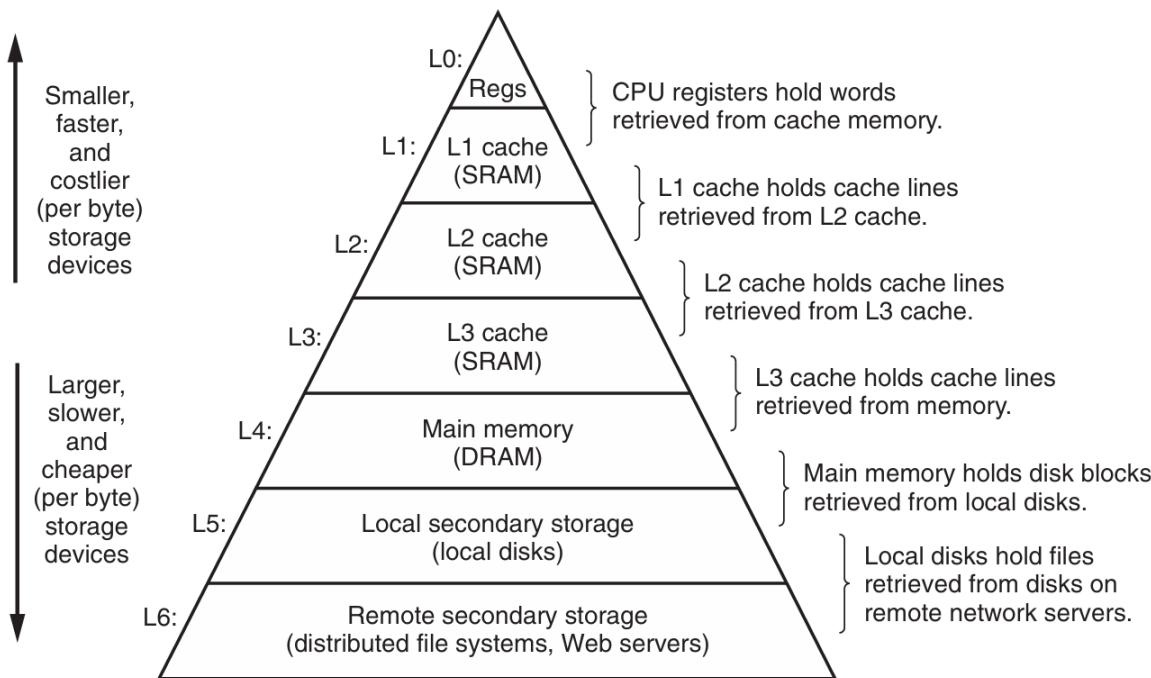
# Chapter 6.3 The Memory Hierarchy

我们在前面的两节中分别讨论了存储技术和计算机软件的一些基本属性：

- 存储技术：不同存储技术的访问时间差异很大。速度较快的技术每字节的成本要比速度较慢的技术高，而且容量较小。CPU和主存之间的速度差距在增大。
- 计算机软件：一个编写良好的程序倾向于展示出良好的局部性。

在计算机中，硬件和软件的这些基本属性互相补充地很完美，这得益于一种组织存储器系统的方法，称为**存储器层次结构**（memory hierarchy），所有的现代计算机系统中都使用了这种方法。

下图展示了一个典型的存储器层次结构：



一般而言，从高层往底层走，存储设备变得更慢、更大、更便宜。

## Caching in the Memory Hierarchy

**高速缓存**（cache）是一个小而快速的存储设备，它作为存储在更大、也更慢的设备中的数据对象的缓冲区域。使用高速缓存的过程称为**缓存**（caching）。

存储器层次结构的中心思想是：对于每个k，位于k层的更快更小的存储设备作为位于k+1层的更大更慢的存储设备的缓存。每一层的存储器都被划分成连续的数据对象组块，称为**块**（block）。数据总是以块大小为**传送单元**（transfer unit）在第k层和第k+1层来回复制的。存储器层次结构能够成功就是因为程序具有局部性。

下面介绍几个关于缓存的概念：

## Cache Hits

当程序需要第k+1层的某个数据对象d时，它首先到当前存储在第k层的一个块中查找d。如果d刚好缓存在第k层中，这就是**缓存命中**（cache hit）。该程序直接从第k层读取d即可。

## Cache Misses

如果第k层没有缓存数据对象d，那就是**缓存不命中**（cache miss）。当发生缓存不命中时，第k层的缓存从第k+1层缓存中取出包含d的那个块，如果第k层的缓存已经满了，可能就会覆盖现存的一个块，称为**替换**（replacing）或**驱逐**（evicting）这个块，被驱逐的块被称为**牺牲块**（victim block）。决定替换哪个块是由缓存的**替换策略**（replacement policy）来控制的。

## Types of Cache Misses

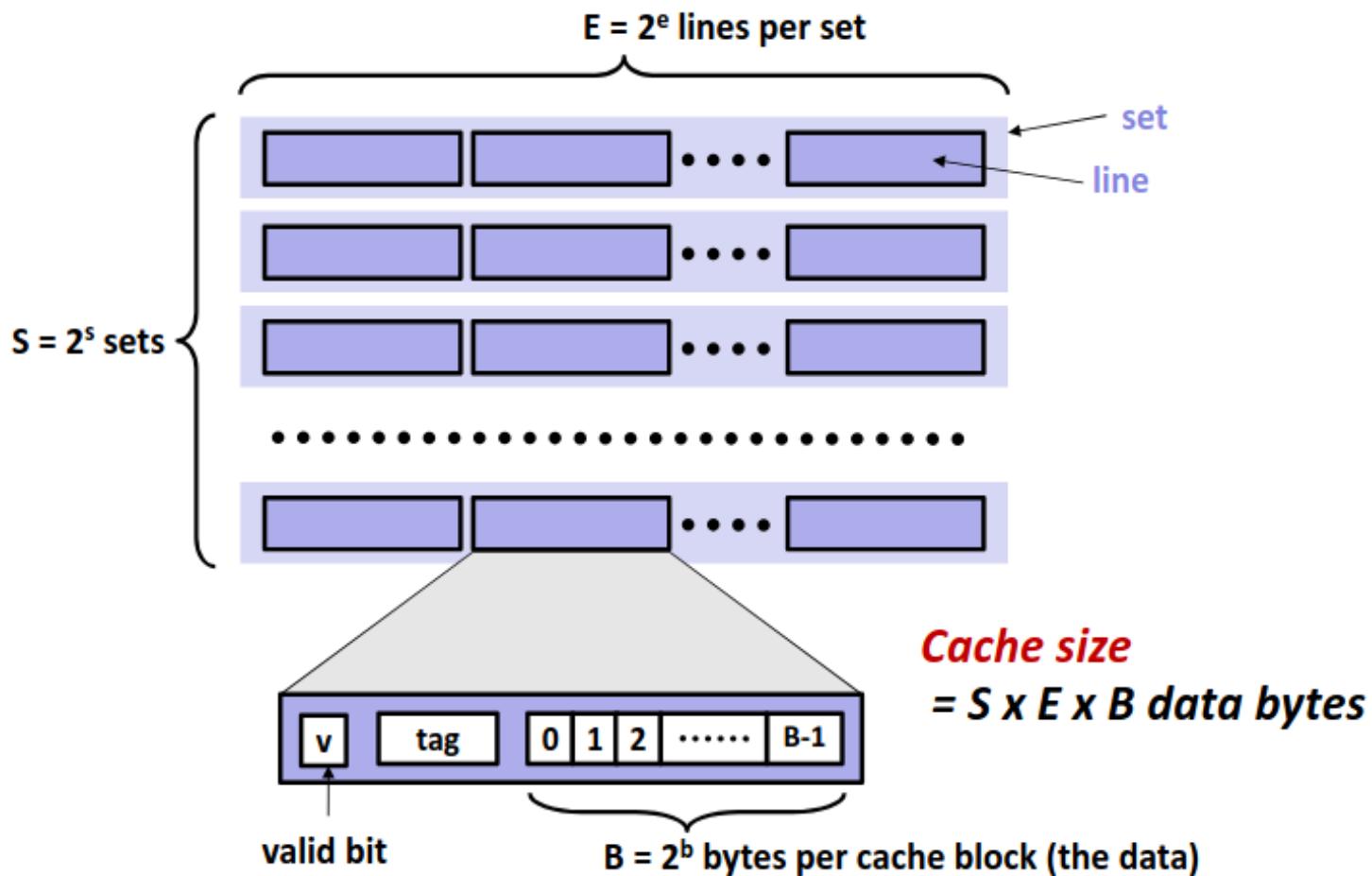
- 强制性不命中（compulsory miss）/冷不命中（cold miss）：如果第k层的缓存是空的，那么对任何数据对象的访问都不会命中。它们通常是短暂的事件，不会在反复访问存储器使得**缓存暖身**（warmed up）之后的稳定状态中出现。
- 冲突不命中（conflict miss）：它是由于限制性的放置策略引起的，在这种情况下，缓存足够大，能够保存被引用的数据对象，但是因为这些对象会映射到同一个缓存块，缓存会一直不命中。
- 容量不命中（capacity miss）：这是由于**工作集**（working set，一个阶段访问缓存块的集合）的大小超过了缓存的大小而产生的。也就是说，缓存太小了，无法处理该工作集。

# Chapter 6.4 Cache Memories

在上一节中我们为大家介绍了存储器的层次结构，其中提及了 Cache 与一些关于高速缓存的基本概念。广义上来讲，任何更为高级的存储器都作为更第一级的存储器的高速缓存。狭义上的高速缓存特指计算机体系结构中 CPU 中的 SRAM 存储器：L1, L2, L3 Cache。本一节将更为详细的介绍狭义上的高速缓存。

## Generic Cache Memory Organization (S,E,B)

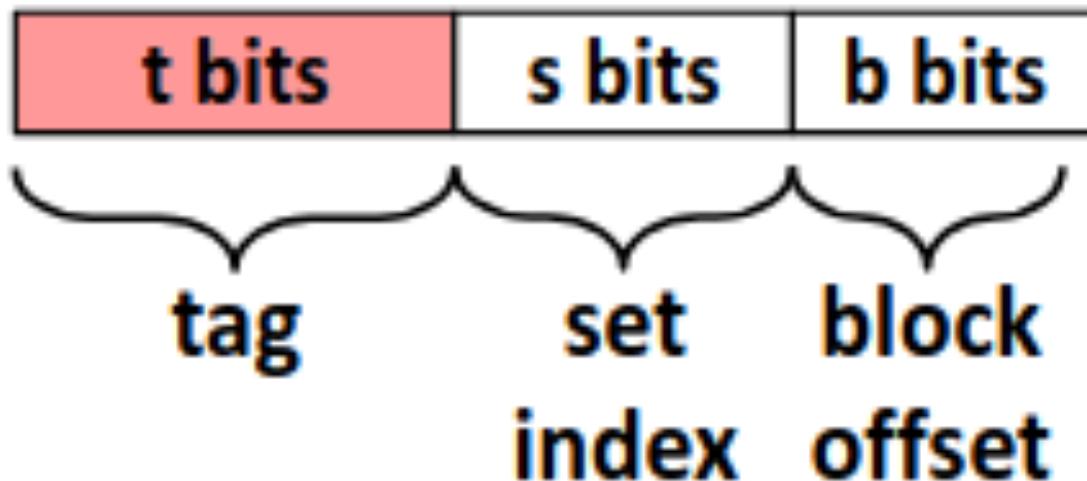
在现代处理器中，高速缓存通常以 **高速缓存组** 的形式组织在一起，一个高速缓存器有  $S = 2^s$  个高速缓存组。其中每个组包含  $E$  个 **高速缓存行**。每个行是由一个  $B = 2^b$  字节的数据块组成。每个行中还包含一个 **有效位** 指明这个行是否包含有意义的信息。



那么对于一个已知的高速缓存和地址长度，我们可以根据高速缓存的特性划分地址，来确定从内存

地址到高速缓存中的映射。

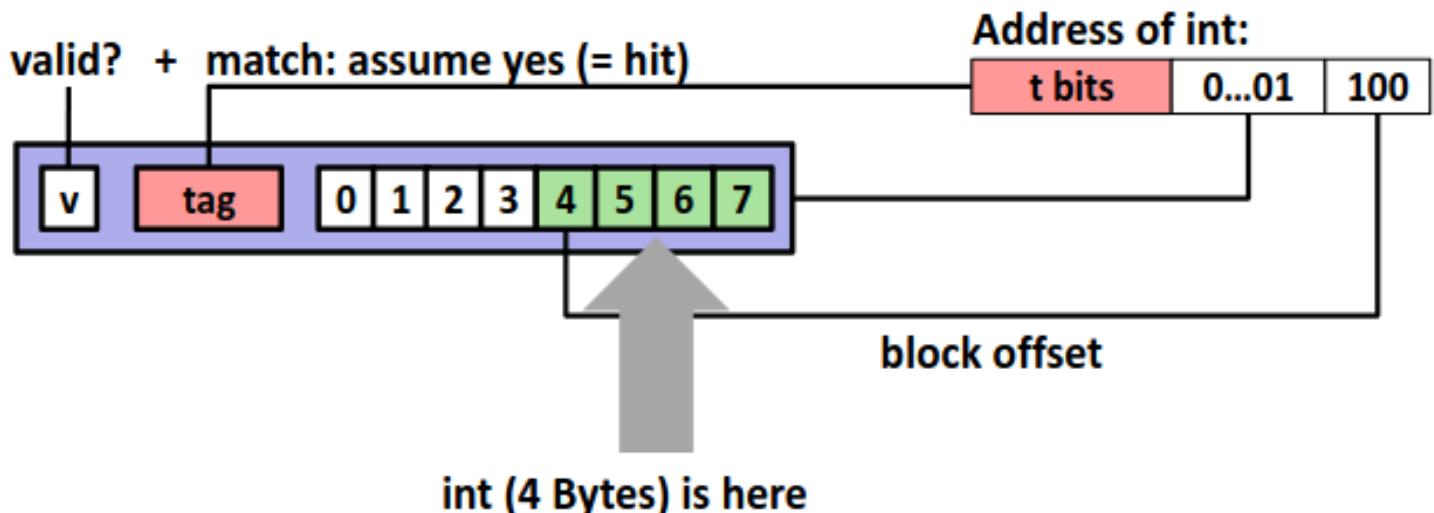
## Address of word:



其中 set index 指明地址对应的组编号，而 block offset 指明数据位于块中的偏移量。而剩下的位数作为 tag 标识与组中每个行比较来指明一个组中是否有目标内存块。

## Direct-Mapped Caches (E=1)

我们先来看一个比较简单的组织方式，**直接映射**，即每个组中只有一个行。对于每次访问，我们只需要根据地址计算出组编号，比较对应高速缓存组的标识位与目标地址，如果相等那么恭喜我们缓存命中不再需要去访问内存，但如果不想等，那么我们只能去内存中并将目标地址的块取出替换掉高速缓存中的块。

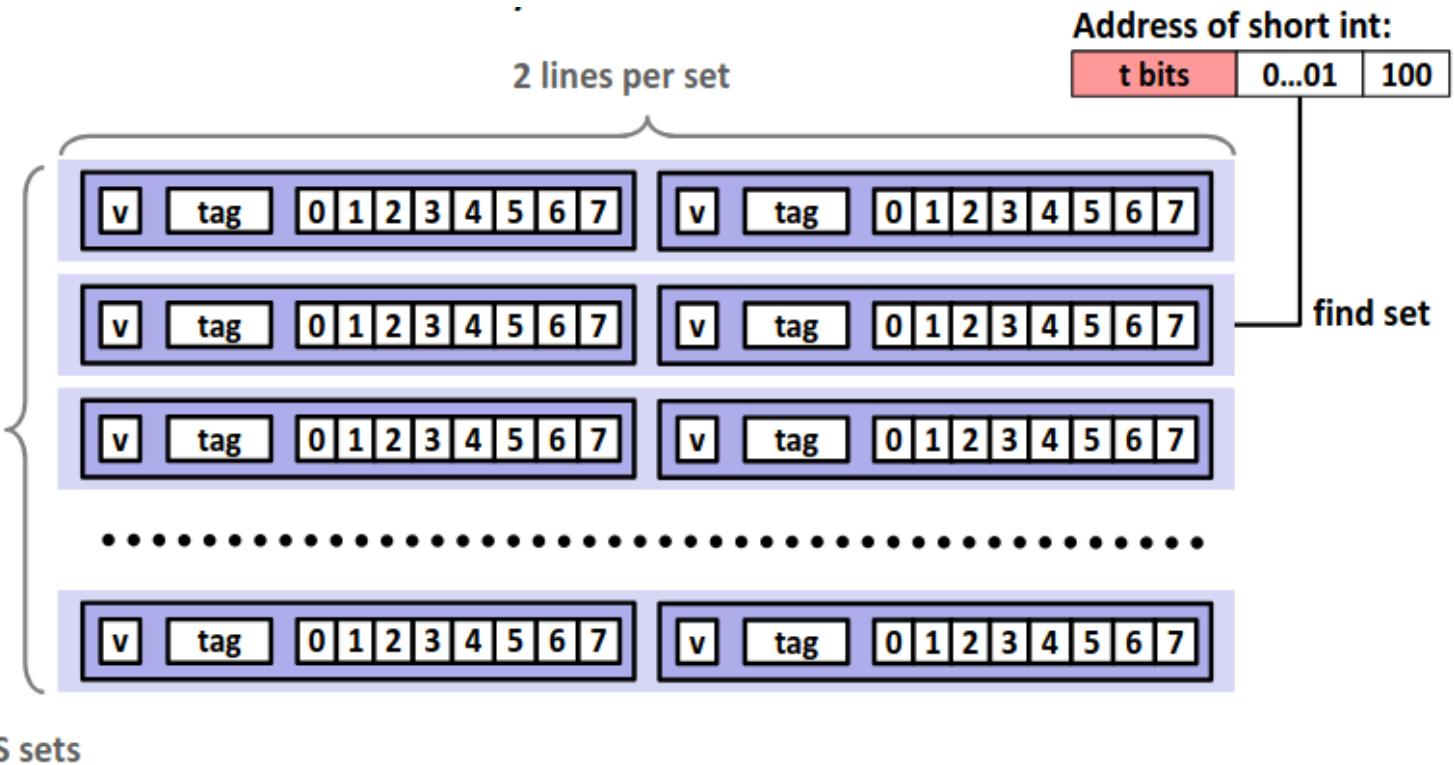


显然这种方式极其容易发生冲突不命中 (conflict miss)。

## Set Associative Caches

相较于直接映射，每一组不再仅有一个行，而是有多个行。这样可以有效减少 conflict miss 的次数，但是对于每次访问，我需要将目标地址的 tag 值顺序比较各个行的 tag 值，增加了电路实现的成本。

下图是一个 E=2 的高速缓存的示意图。



## Issues with Writes

每当我们向内存写入的时候，由于高速缓存的存在，我们有不同的写入策略。

如果缓存命中，我们有两种策略：

1. **Write-through:** 直接写入内存当中。
2. **Write-back:** 我们只在缓存中进行更改，当我们需要替换掉缓存中这个块时，再将这更改写入内存。这种方式需要额外的 dirty bit 去指示块是否受到更改。

如果缓存不命中，我们同样有两种策略：

1. **Write-allocate:** 我们将修改内存对应的块加载到缓存中，再在缓存中修改。
2. **No-write-allocate:** 直接对内存修改，不将块加载到缓存中。

现代处理器通常采用 **Write-back + Write-allocate** 的策略组合。

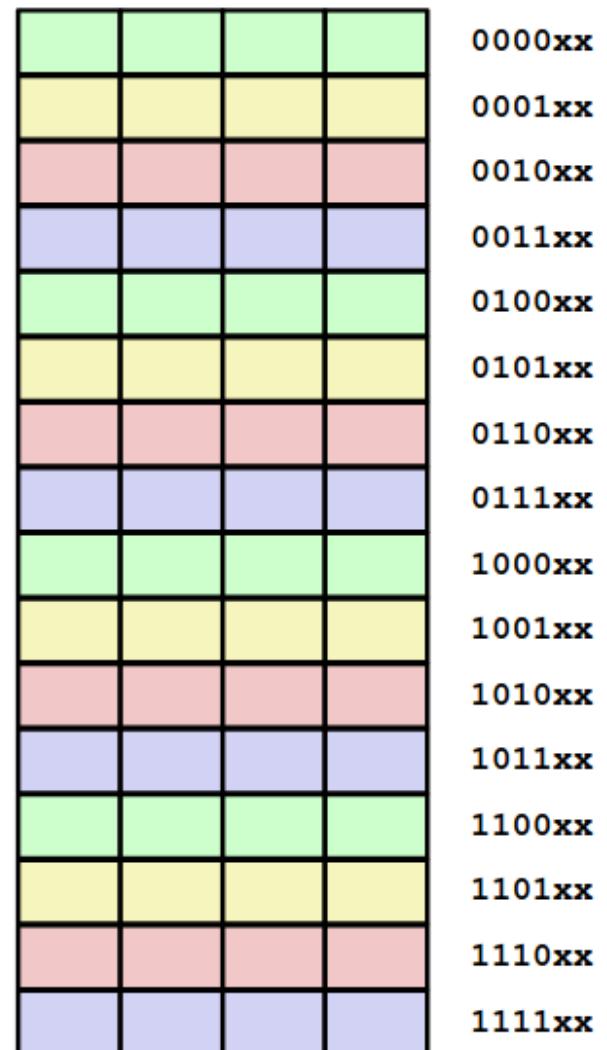
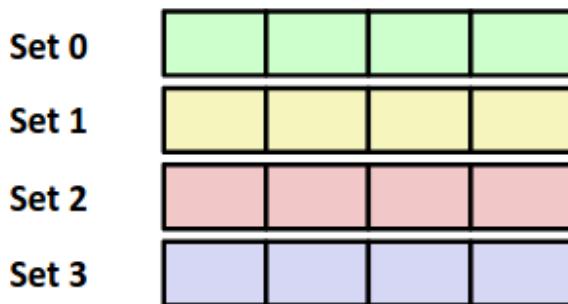
# Why Index Using Middle Bits

对于一个地址的划分，为什么我们采用中间的位作为组索引呢？由于空间局部性的存在，如果我们将中间的位作为组索引，有利于我们将一个更大的内存整体放在缓存中。

如图是我们组索引放在中间位与放在高位的对比，显然前一种方式更优。

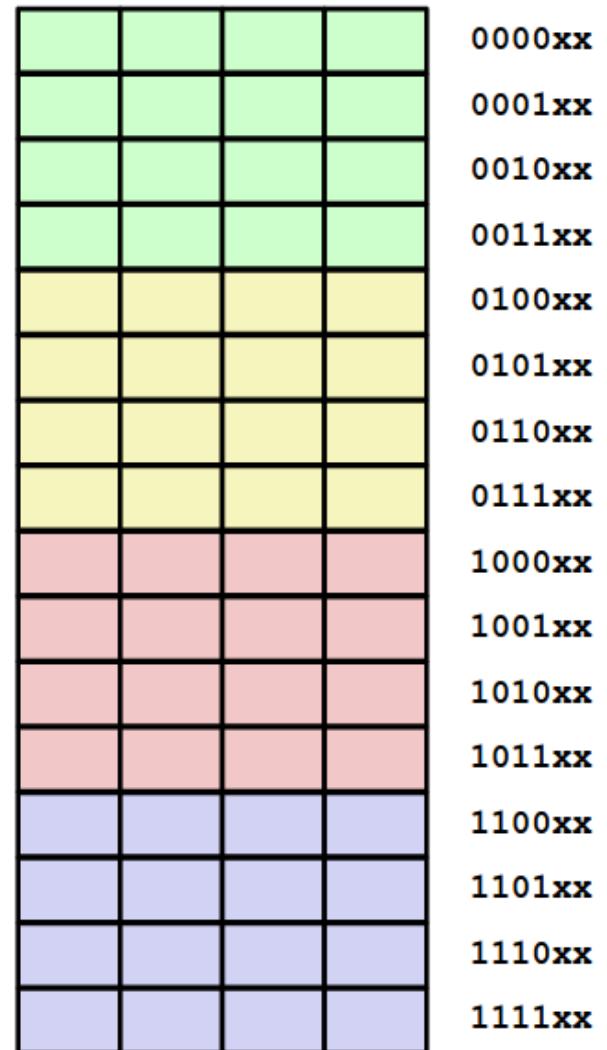
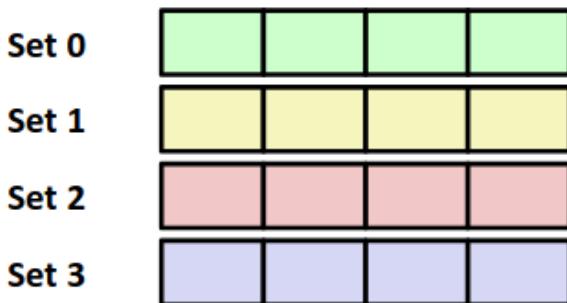
## Middle Bit Indexing

- Addresses of form **TTSSBB**
  - **TT** Tag bits
  - **SS** Set index bits
  - **BB** Offset bits
- Makes good use of spatial locality



# High Bit Indexing

- Addresses of form **SSTTBB**
  - **SS** Set index bits
  - **TT** Tag bits
  - **BB** Offset bits
- Program with high spatial locality would generate lots of conflicts

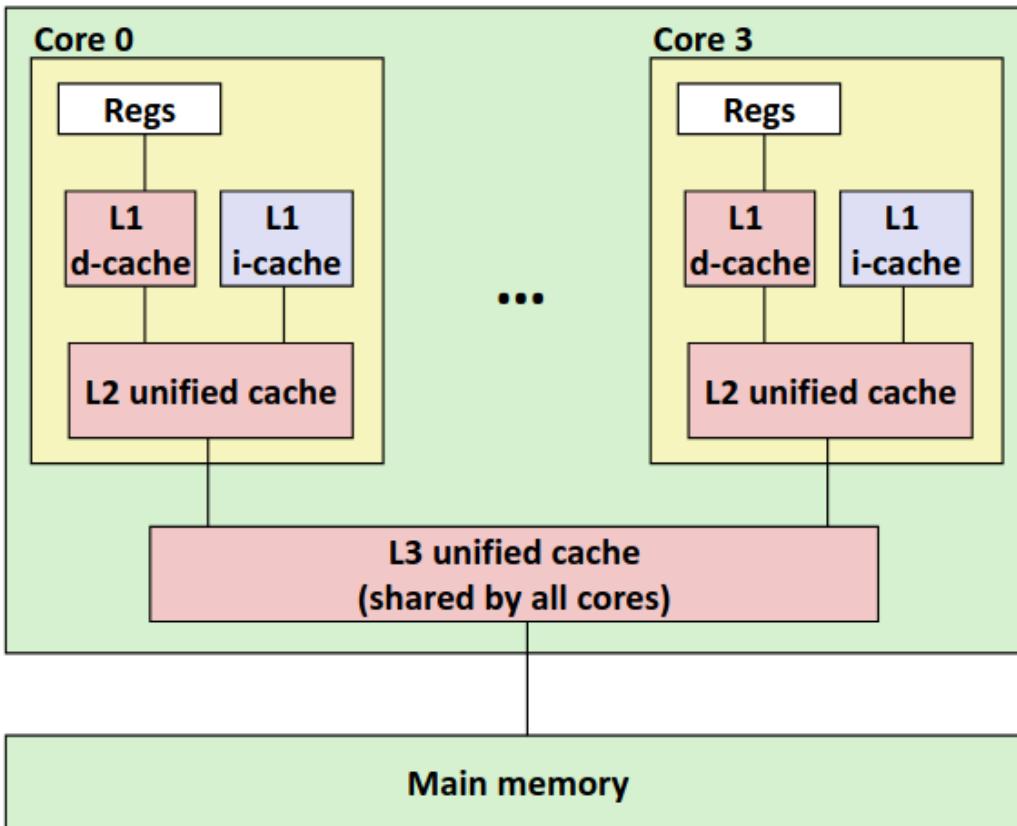


## Anatomy of a Real Cache Hierarchy

我们展示一个典型的现代处理器中高速缓存的组织。

# Intel Core i7 Cache Hierarchy

Processor package



**L1 i-cache and d-cache:**

32 KB, 8-way,  
Access: 4 cycles

**L2 unified cache:**

256 KB, 8-way,  
Access: 10 cycles

**L3 unified cache:**

8 MB, 16-way,  
Access: 40-75 cycles

**Block size:** 64 bytes for all caches.

22

可以看到数据和指令在第一级缓存中是分别存储的，而第二级缓存为每个核独有并且不再区分数据与指令。第三级缓存则是所有核共有了。

---

© 2025. ICS Team. All rights reserved.

# Chapter 6.5 The Impact of Caches on Program Performance

我们前文不断强调高速缓存以及程序局部性的重要性，在这一节中，我们会先通过一个简单的程序验证局部性以及缓存对程序性能的影响，然后举一个简单的例子来说明我们应当怎样提高程序局部性，更好的利用缓存。

## The Memory Mountain

我们通过一个简单的程序，改变程序的一些参数，测试程序内存吞吐量来衡量程序读写性能。

## Memory Mountain Test Function

```

long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *         array "data" with stride of "stride",
 *         using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}

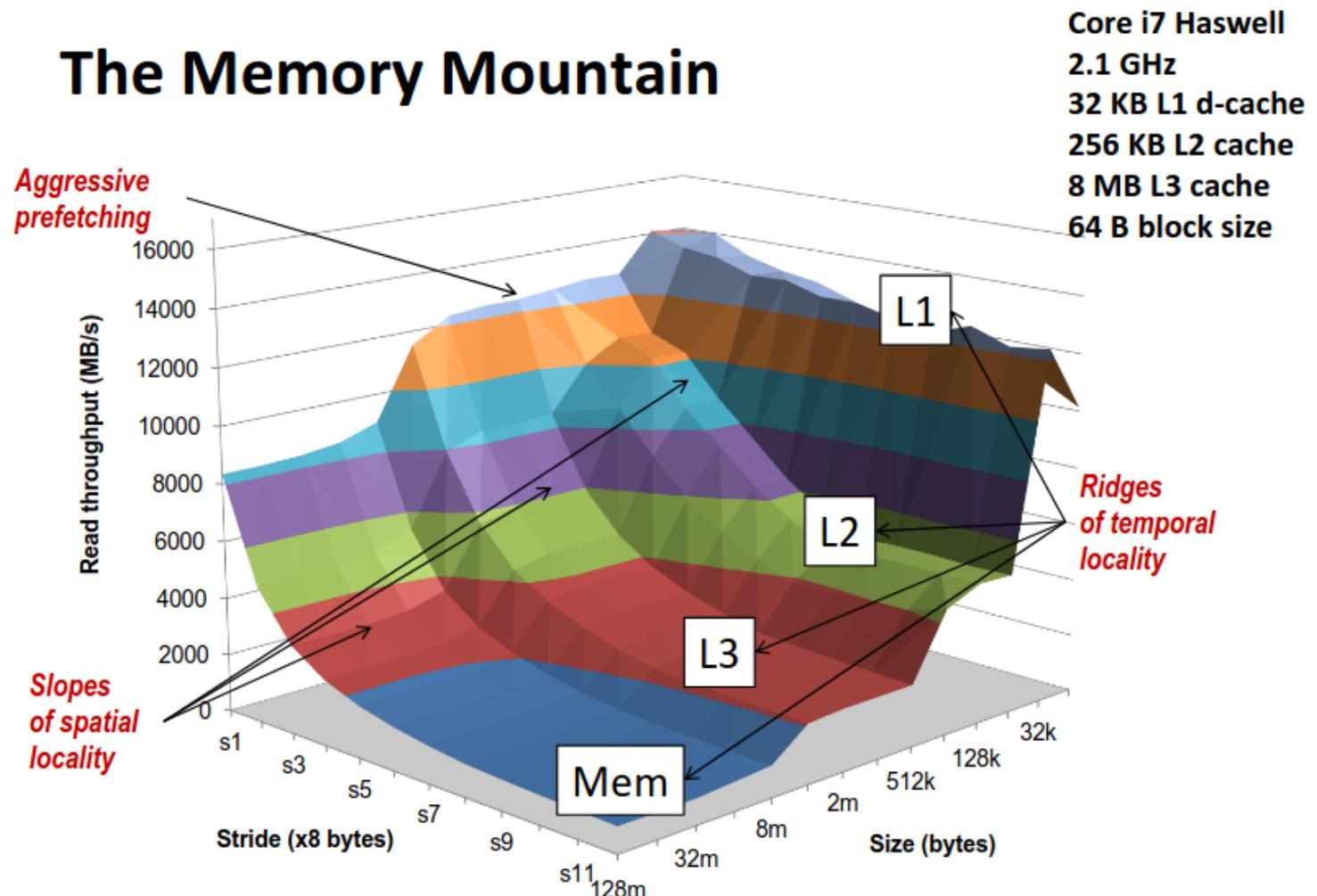
```

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.
2. Call `test()` again and measure the read throughput(MB/s)

我们通过改变数组访问的步长以及数组的大小，然后测试程序内存吞吐量，得到了下图。



整个测试数据输出集排布在坐标系下，就像一座山一样，所以我们称呼它为**内存山(The Memory Mountain)**。

越往山的上方走去，有更小的步长以及数组大小，对应程序更好的空间局部性、更高的缓存命中率以及对缓存更好的利用，我们得到了更优良的程序性能。而越往山的下方走也对应了更差的程序性能。

通过 ics 课程的学习，我们在写程序时也应当有意的提高程序的局部性，让自己的程序在内存山的较高点，而不是山底。

## Rearranging Loops to Increase Spatical Locality

矩阵乘法在计算机程序中极为常见，广泛应用在图像渲染处理与机器学习等领域。我们今天不讨论从数学与算法的角度怎样乘会更优，我们就应用  $O(n^3)$  的算法，在计算机体系结构的角度下我们讨论怎样的乘法次序是最优的。

我们先来看一种最典型的写法: ijk

```
/* ijk */
for (i=0; i<n; i++) t
    for (j=0; j<n; j++) {
        sum = 0.0; ←
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

*Variable sum  
held in register*

*matmult/mm.c*

那么对于内层循环的访问，我们通过为命中率计算公式:  $miss\ rate = sizeof(a_{ij})/B$  。我们假定 Block size = 32B (four double)。

我们可以计算得到未命中率如下图:

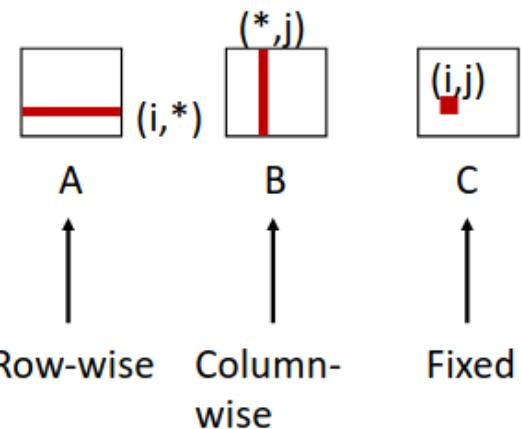
```

/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

```

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

| <u>A</u> | <u>B</u> | <u>C</u> |
|----------|----------|----------|
| 0.25     | 1.0      | 0.0      |

**Block size = 32B (four doubles)**

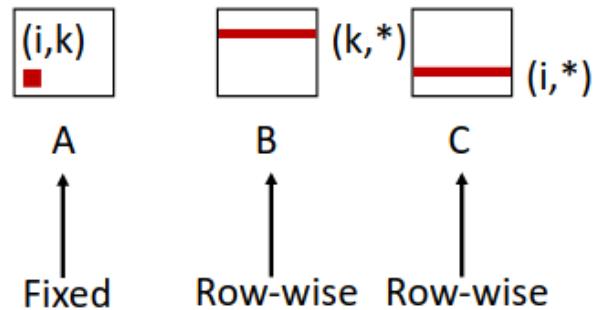
同理我们看另两种写法:  $kij$  与  $jki$ , 并计算 miss rate。

# Matrix Multiplication ( $kij$ )

```
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

*matmult/mm.c*

Inner loop:



Miss rate for inner loop iterations:

|          |          |          |
|----------|----------|----------|
| <u>A</u> | <u>B</u> | <u>C</u> |
| 0.0      | 0.25     | 0.25     |

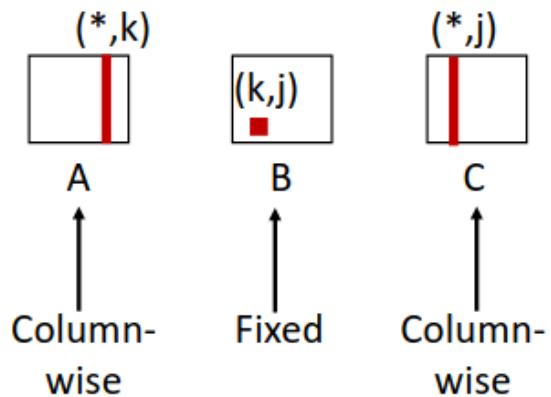
**Block size = 32B (four doubles)**

# Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

*matmult/mm.c*

Inner loop:

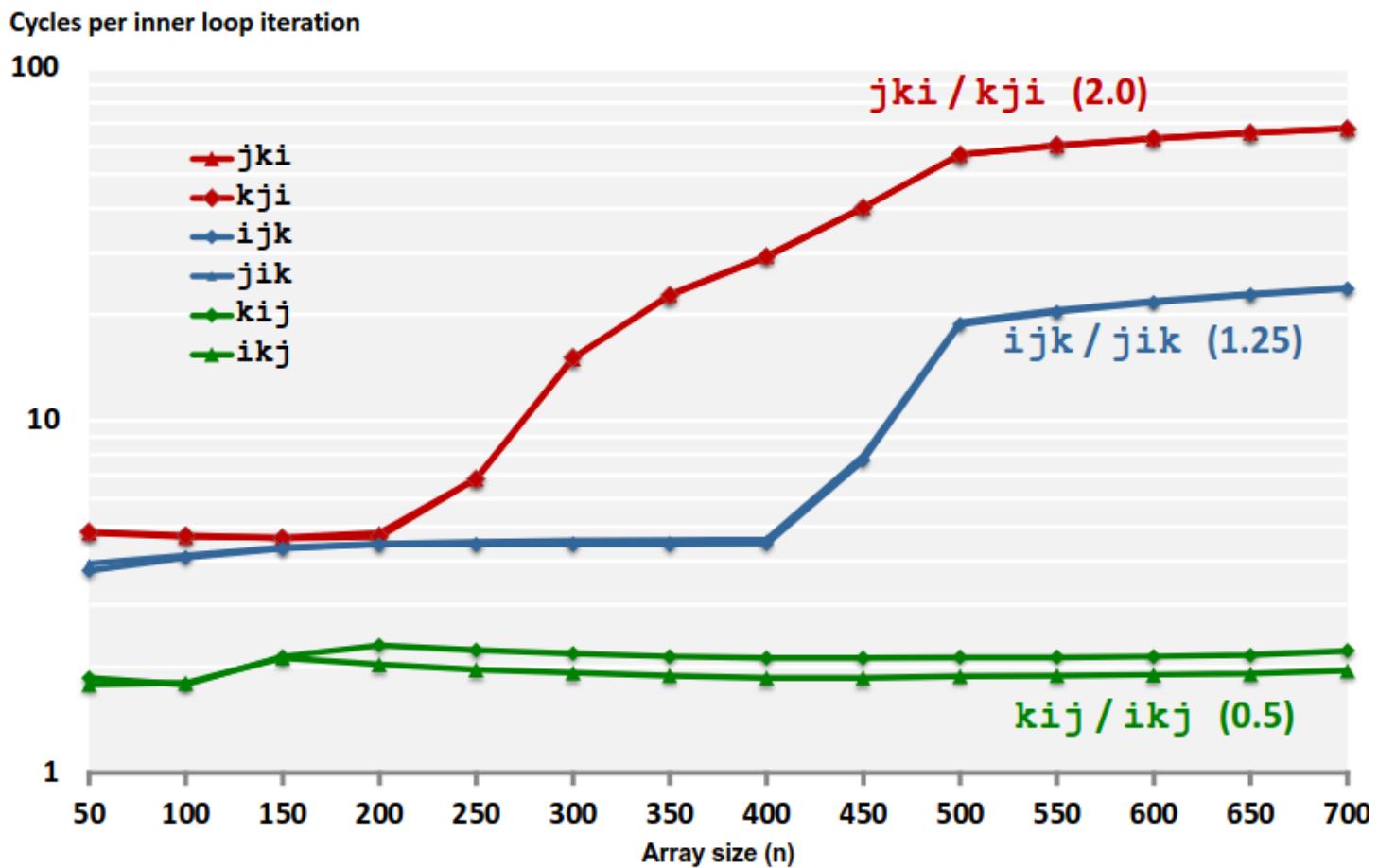


Miss rate for inner loop iterations:

|                 |                 |                 |
|-----------------|-----------------|-----------------|
| <u>A</u><br>1.0 | <u>B</u><br>0.0 | <u>C</u><br>1.0 |
|-----------------|-----------------|-----------------|

**Block size = 32B (four doubles)**

显然根据理论计算  $kij$  是最优的访问次序。我们在处理器上试验一下



最优的访问方式远远优于最差的访问方式！这就是计算机体系结构的力量。

我们可以通过 blocking 的技术进一步优化矩阵乘法，在此处也就不再进一步介绍了，感兴趣的同學可以查询一些资料以了解。

那么到此为止本课程介绍的 The Memory Hierarchy 内容就结束了，希望对你有帮助 🌸

© 2025. ICS Team. All rights reserved.

# Chapter 7 Linking

---

© 2025. ICS Team. All rights reserved.

# Chapter 7.1 Base Concepts

前六章我们一直在讨论程序的结构与执行的话题，即程序本身的表示、结构与运行过程。从本章起，我们将关注一个全新的话题：**在系统上运行程序**，即程序是如何在一个**操作系统**上运行的，不仅仅关注程序本身，更关注它与操作系统的交互。

这一章我们将聚焦一个极其容易被忽视而又十分关键的过程：**链接(Linking)**。我们将讨论：

1. 为什么需要链接，链接带来了什么好处？
2. 现有的计算机系统中是怎样链接的？
3. 链接带来的一些有趣的技术。

## Why Linking?

想象这样一个过程，这是个没有链接的世界，你写的每个代码都直接**整体**的编译成一个程序。

在这样一个世界，你是牛逼的开发 Windows 系统项目的管理人，吃着火锅唱着歌，老板来了个电话：程序出 bug 了！

你回到电脑前，经过漫长的调试，你发现是新来的小伙把 `i==0` 写成了 `i=0`，你一怒之下开除了他，改了这个小小的错误。然后面对几百万行代码，怎么办？重新编译！

你说编译就编译呗，不就点一下的事吗？几秒钟就搞定了。那是你平常写的玩具程序很短，几秒就完成了编译。But! 在前面优化第五章我们学过，编译过程编译器会做很多复杂的优化，需要消耗大量的资源去复杂的分析，就算降低优化强度极长的代码也需要漫长的时间翻译。几百万行代码往往需要以天、周甚至月为单位的时间进行编译。

为了重新编译这个有一丁点改动的代码，你们重新编译了这个庞大的代码，在一个月之后，终于完成了。你把修改了 bug 版本的程序打包发给客户，绝望的发现：用户早就开始用友商的程序了。这时你接到老板愤怒的电话：带着你的项目组滚蛋！

失业的你极其愤怒并且无事可干，你想到我就改了那么小一个部分的代码，却要把整个代码重新编译一遍，如果我可以只把那一小部分抽出来重新编译，再和其他部分**链接**在一起就好了！你在悲愤之中写出了人类历史上第一个链接器！你将被历史铭记！

好了，白日梦结束了，你既不是 Windows 的开发这并且人们早就发明了链接没机会给你永垂不朽

了。

从上面的例子我们可以看出链接的重要性。在学校中大家往往写的都是写简单的玩具代码，最长也不过百来行，全部揉进一个文件中也不是不行，链接往往被忽略。

然而在实际的工程实践中，代码往往十几万行起步，百万行的程序更是比比皆是，并且涉及到多人合作、频繁修改等问题。这种时候如果在一个文件中这个工程就别维护了，项目组该滚了。那么在不同的文件中必然涉及到如何**整合成一个程序**。

接下来我们将系统的阐述链接的一些思想方法与好处。

## Modularity and Efficiency

链接是程序**模块化思想**的一部分，程序可以划分成一个一个模块，有不同的人分别编写调试完成再整合为一个完整的程序。并且可以调用整合一个个外部的库去简化编程提高代码复用率。

这样**模块化+链接**的模式带来开发效率上巨大的好处。

1. 时间上，对于每个文件分别编译，当我改变其中某个文件是无需将整个文件编译，只需要重新编译那个更改过的较小模块，并重新链接，这个的时间代价远远小于完整编译。而且对于不同的模块我可以并行编译。
2. 空间上，通过**库**的思想，我们实现将常用的功能封装进库中，在需要的时候调用库而非再重新编写。通过**动态库**的技术我们甚至能够节省内存！

## Procedures

在介绍完了链接的基本好处和思想后，我们来看该如何链接。链接有两个基本的步骤**符号解析**以及**重定位**。

## Symbol Resolution

```
// swap.c
void swap(){...} /* define symbol swap */

// main.c
int main()
{
    ...
    swap();           /* reference symbol swap */
    int *xp = &x;    /* define symbol xp, reference x */
    ...
}
```

在代码中我们会产生大量的**符号(Symbol)**，包括函数、变量等等被命名的单位。符号会经过定义和引用的过程。定义时向编译器声明了这个符号是什么，引用时则是使用这个符号。

但是这两步骤往往可能**并不在同一个文件中**！在 swap.c 中定义了 swap() 这个函数，但我却需要在 main.c 中的 main() 函数中调用。那么分别编译时编译器在编译 main.c 时就无法得知 swap() 具体的定义，在链接的过程就需要将**每个符号的引用匹配上它的定义**。这个过程被称为**符号解析(Symbol Resolution)**。

## Relocation

这不同的编译好的文件合并为一个文件的过程中，势必要排布不同文件的位置。经过前几个章节的熏陶学习我们知道，程序在执行时被加载经内存中，每条指令对应了一个地址，所谓调用函数的过程不过是跳转执行某个地址开始的指令。

然而由于分别编译的缘故，在单个文件编译时**不知道自己会被分配到哪个地址处**，每个符号有着不确定的地址，对于函数符号无法调用执行，对于变量无法确定分配内存位置读写。

那么链接时，在完成了符号解析后，显然需要对每个符号分配一个合理的地址，并且对于每个引用了这个符号的代码处填上合理的值。这个过程就是**重定位(Relocation)**。

# Object Files and ELF

## Three Kinds of Objects Files

通过了编译器以及汇编器翻译后的代码是**目标文件(Object Files)**，在第三章我们学过里面的二进制数对应了一个个汇编指令，常见的二进制文件有以下三种格式。

1. **Relocatable object file(.o file)**: 包含代码和数据，可以和其他可重定位二进制文件经过链接器处理形成可执行文件。
2. **Executable object file(.out file)**: 包含代码和数据，正如其名可以直接被加载到内存中执行。
3. **Shared object file(.so file)**: 一类特定的可重定位文件，也被称为**动态库(Dynamic Link Libraries DLLS)**。我们将在库一节中详细介绍这种文件。

## Excutable and Linkable Format(ELF)

Linux 中目标文件的标准二进制格式称为: ELF。正如其英文缩写展开后对应所示，包含可执行的程序以及通过链接后可执行的程序。本课程不会过于详细的介绍这种格式，只是简单介绍其中的一些内容为后面详细介绍链接过程提供必要的知识。对这种格式感兴趣的同學可以进一步查询资料了解这种格式~~(反正你做 Linkerlab 也得查)~~。

ELF 文件被被划分为一个个 **section**，不同的 section 对应了数据以及代码。基本的有 .text 对应了代码部分、.data 对应了数据部分。其中为了链接有几个特殊的 section 。

**.syntab**， 符号表(symbol table)对应了本文件中符号信息，无论是定义的符号还是引用的符号，在符号解析时链接器会到符号表中查询符号的定义以及处理待解析的符号。

**.rel.text** 存储了 .text 中待重定位的信息，**.rel.data** 存储了 .data 中待重定位的信息。在重定位时链接器就会到这些段中查找待处理的重定位需求

---

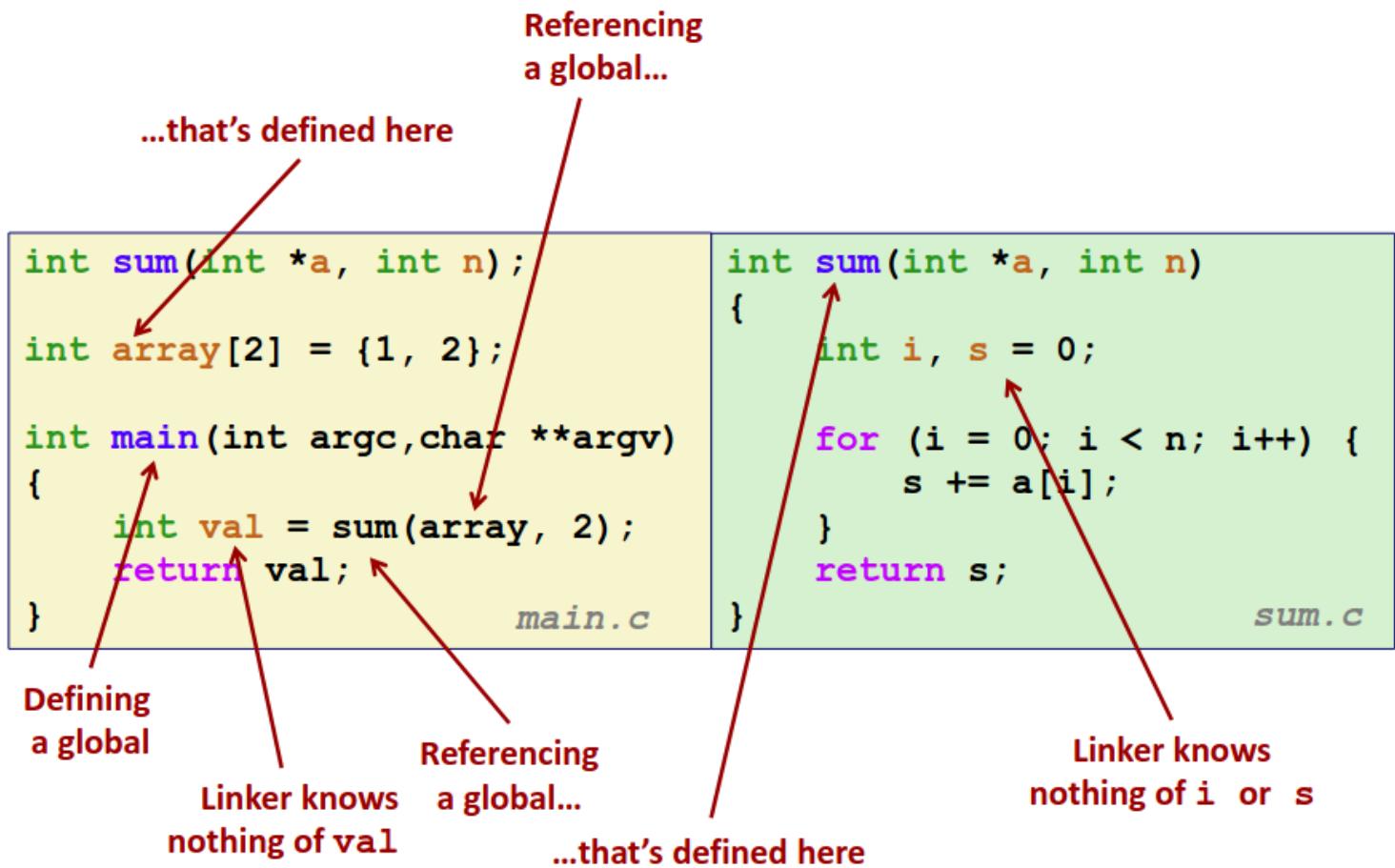
© 2025. ICS Team. All rights reserved.

# Chapter 7.2 Procedures

在上一节中我们讨论了关于链接的基本知识，并简要介绍了链接的步骤，在这一节中我们将更为详细的介绍链接的步骤。

## Symbol Resolution

### Step 1: Symbol Resolution



上图是一个简单的程序的实例，可以看到即便在如此简单的程序当中也有大量的符号以及引用关系。为了解析程序中的所有符号，为符号分类是必须的。

因为我们做符号解析的最终目的是将符号引用绑定到符号的定义上。对于函数而言，定义和引用的区别十分明显：有函数体的是定义，反之为引用。而对于变量由于 `extern` 等语法的存在，定义与

引用不太容易区别，我们引入下面的分类标准。

我们大体上将符号分为两种：**强符号与弱符号**。

强符号包含**函数的定义以及初始化过的全局变量**。弱符号包含**未初始化的全局变量或者被 `extern` 修饰的变量**。可以看到我们只关注了全局变量，对于局部变量在其作用域内解析是十分轻松的，无需额外关注。

有了这个分类标准，我们便可以以此进行符号绑定，有以下三条规则。

1. 同名强符号只允许有一个。显然多个同名强符号相互冲突。
2. 当同时有同名强符号以及弱符号时，符号绑定到强符号上。
3. 如果有多个同名弱符号并且没有强符号时，随机选择一个作为符号的定义。

这三条规则形成了我们符号解析的基本逻辑，有了三条规则符号解析变得相对清晰。值得注意的是规则三，虽然编译器不会报错，但随机选择一个无疑是一个糟糕的结果，可能引发错误。现有的部分编译器链接器在触发规则三时会有显示的警告或报错。

三条完备的规则足以让我们避免链接过程中所有错误吗？其实不然。看下面这样一个例子。

```
// a.c
int x=7;
int y=5;
p1(){}

// b.c
double x;
p2()
```

在这个例子中 b.c 中的 x 为弱符号绑定到 a.c 中的强符号 x 上。然而 b.c 在编译时并不知道 x 最终的绑定结果是一个 int，符号绑定时又不会做任何类型检查。故在 b.c 中对 x 的修改仍是对 double 类型 8 个字节进行操作，不仅会让 a.c 中 x 无法得到预期的值，甚至会覆盖掉 y 的值。可以说是非常糟糕了。

这种情况下，编译时编译器无法得知 x 的绑定结果，没有类型检查连一个警告都没有，而链接器在链接时根本不做类型检查，导致出错也难以修改，可以说是必须有程序员把关了。

可以说全局变量是一切灾难的根源，它使得程序模块之间耦合度增加，不仅使得开发过程困难，更使得维护调试困难。所以，`avoid if you can！` 如果实在无法避免，尽可能的用 `static` or `extern` 去修饰限制它，并为它初始化。

# Relocation

完成符号解析之后，我们就可以进行重定位了，在上一节中我们介绍了由于分别编译的缘故，为什么需要重定位？这一节中我们将不再赘述而是直接讲解如何进行重定位。

对于每一个符号引用，我们都需要为它进行一次重定位。而对于每个符号引用的是什么以及在 ELF 文件中的位置是什么，编译器在单个文件编译时已经整理好储存在 ELF 中，也就是我们上一节介绍 ELF 时提到的 .rel.text 以及 .rel.data 区中。

其中的每一个需要重定位的引用我们称为**重定位实体(Relocation entries)**。链接器只需要遍历重定位实体，根据符号绑定关系，在对应地址上填上恰当的值即可。

然而对于每个重定位实体在表中所对应的地址值是编译时单个文件产生的地址，需要根据链接器对文件排布作出相应的修改。而填入值，即对应绑定符号的位置也是同理。

由于计算值涉及到相对地址等本课程没有介绍的内容，此处只是提供一个大致介绍，详细计算不再过多赘述。

在完成对每一个重定位实体完成重定位后重定位就结束了，那么链接的工作也就大致完成了。

---

© 2025. ICS Team. All rights reserved.

# Chapter 7.3 Library

在这一节我们会讨论通过链接实现的重要技术：库。

## Libraries and Static Linking

**库(Libraries)**，就是打包好的函数集合。任何实用的 C 程序都离不开库，无论是 I/O `printf` and `scanf...`、还是 `math sqrt` and `log...`。

通过先前对链接的知识，库这种技术的实现离不开链接。调用库的过程就是将库函数文件与你的 C 代码文件链接在一起的过程。但怎样链接在一起值得探讨。

一种暴力的做法是，将库文件整个链接进程序当中。当我需要用到 `sqrt` 时，不管三七二十一直接将指数对数三角什么的全部链接进程序。这种方法毫无疑问可行，但过于低效。

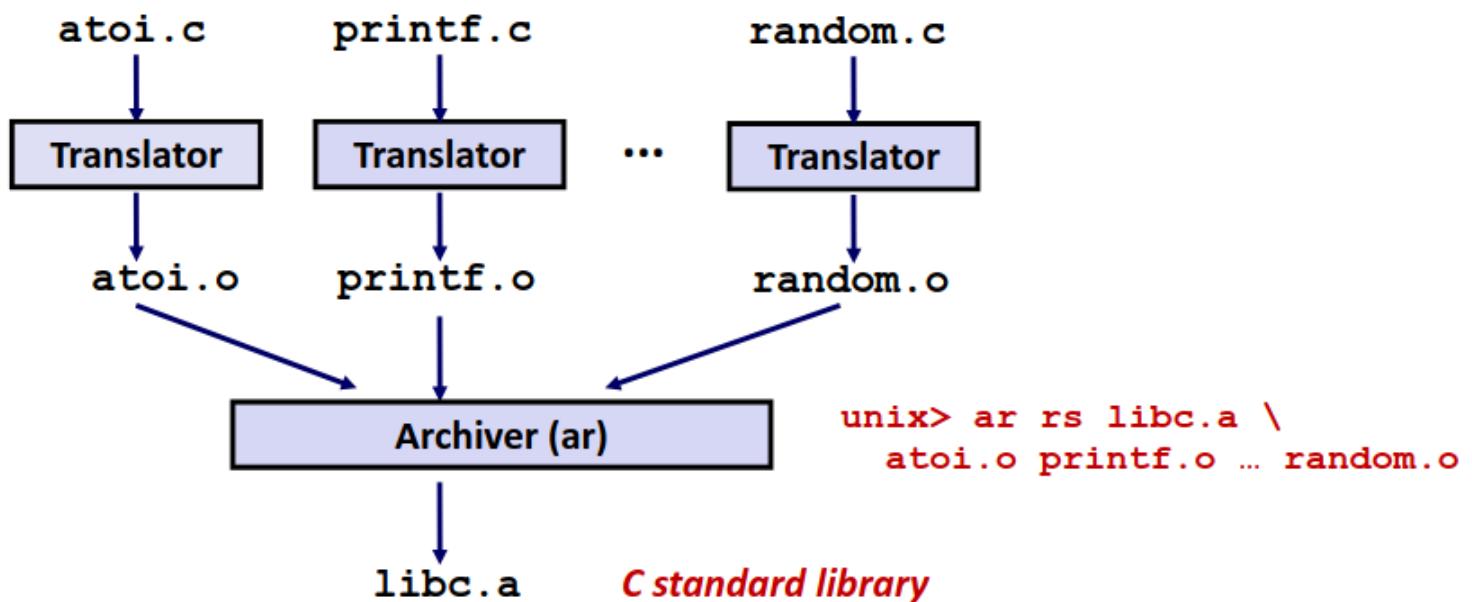
另一种做法是对于每一个函数单独列成一个文件，当我需要 `sqrt` 时，我将其对应的文件加入到链接当中。这种做法显然会让库函数的引用变得相当繁琐，而且考虑到很多函数的实现有依赖关系，单个文件实现一个函数也会十分低效。

在过去十分流行的解决方案是**静态库(Static Libraries)**。静态库在 linux 下通常以 `.a(archive files)` 结尾。

将许多函数编译成可重定位目标文件，打包进入一个静态库文件中，并为他们添加索引。通过改进链接器当其在做符号解析时会到静态库中去搜寻特定的符号。当一个待解析的符号匹配了库中的符号时，仅仅将符号对应的函数以及其依赖的函数链接进程序中。

这种方法不仅提高了效率，仅仅链接了需要的部分，同时也简化了编程以及库的编写。

下图展示了一个静态库打包的过程。



举一个简单的例子来帮助大家理解静态库链接的过程。

```
// vector.h

void addvec(int *x,int *y,int *z,int n);
void multvec(int *x,int *y,int *z,int n);

// main2.c

#include<stdio.h>
#include<vector.h>

int x[2] = {1,2};
int y[2] = {3,4};
int z[2];

int main()
{
    addvec(x,y,z,2);
    printf("z=[%d %d]\n",z[0],z[1]);
    return 0;
}

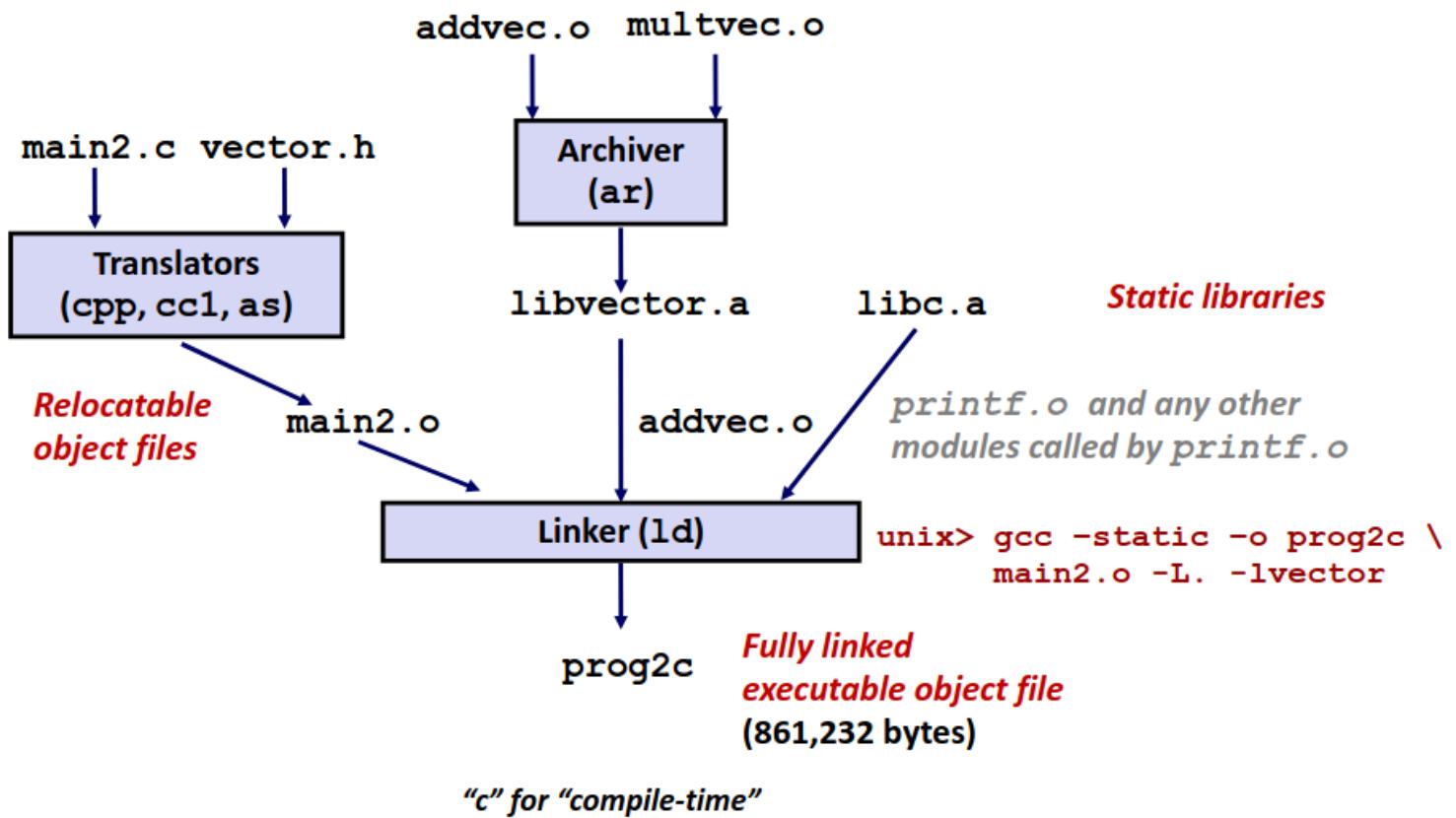
// libvector.a

void addvec(int *x,int *y,int *z,int n){...}
void multvec(int *x,int *y,int *z,int n){...}
```

可以看到在这个简单的程序中，我们自己定义了一个向量静态库，其中包含向量加和向量乘函数的实现。而 vector.h 文件中实际上只包含了函数的声明，然后我们在编译时刻意去包含libvector.a 才提供了函数的定义。

这是极为容易误解的一点，很容易误解为当我引用 stdio.h 这个文件时我就引用了这个库，其中包含了函数的实现，其实并不是。.h 文件中只会提供函数的声明，让你在编译时不会报错，而实际上对库的引用是在链接过程中。

那你就会问了：我在编译时也没有可以去链接 lib.c 库啊？对于这种标准库，编译器会默认去链接它们。



## Dynamic Linking

在介绍静态库时我们提到，这是老式十分流行的实现方法，而现在更为流行的方法是**共享库(Shared Libraries)+动态链接(Dynamic Linkng)**。

为什么要用新的技术呢，静态库有什么不好吗？静态库有几个缺点：

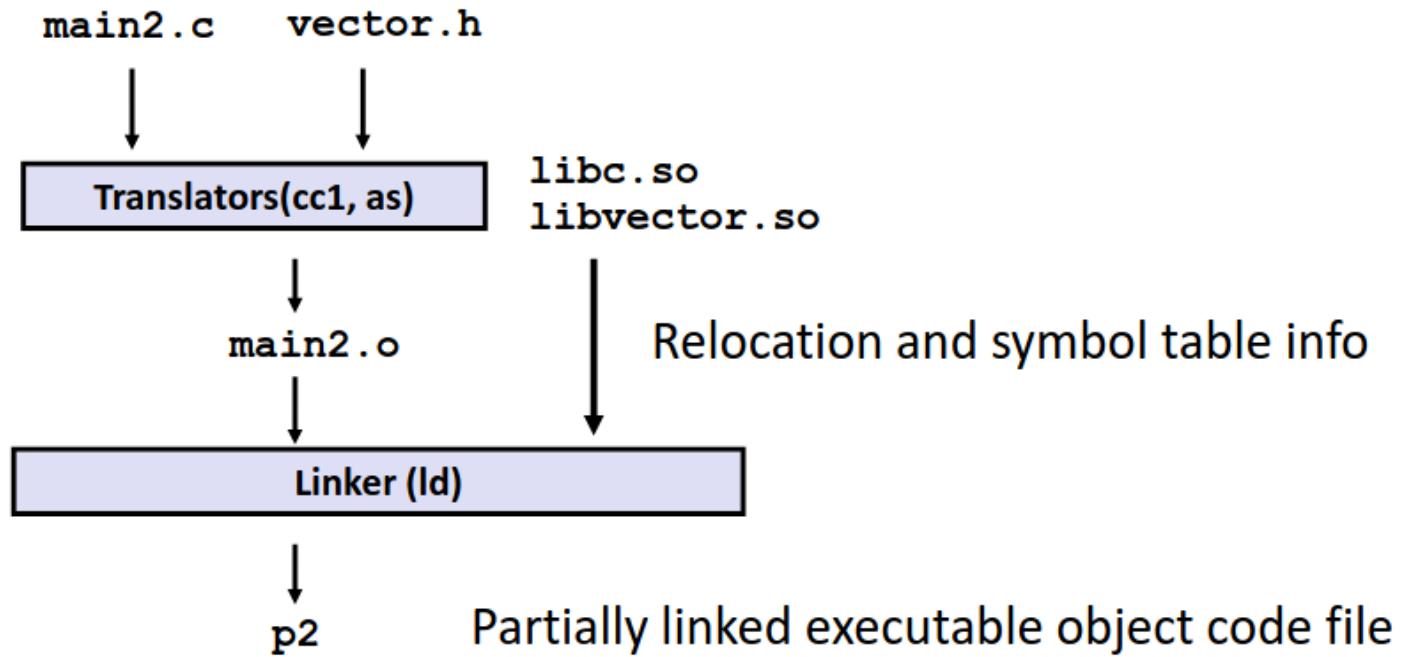
1. **重复**：几乎每个文件都会用到 libc，每个文件中都有 printf or scanf，那么对于静态链接，每个程序中都包含了一份库的代码，重复存储。并且现在计算机同时执行上百个进程，而每个进程的代码中同样也包含重复的库，重复消耗主存。
2. **灵活性差**：如果我库函数发现了 bug，或者有一些改进，那么如果想要让已经链接好的程序用上新的库函数，唯一的方法是重新链接。但是对于那些广为流传的库函数，包含了它的程序成千上万，让这些程序都重新链接显然是一个不可能的事。

那么现在的解决方案**共享库(Shared Libraries)**，linux 下通常以 .so 结尾，windows下常见后缀是 .dll。对于库我在程序加载甚至运行时在链接进程序当中。

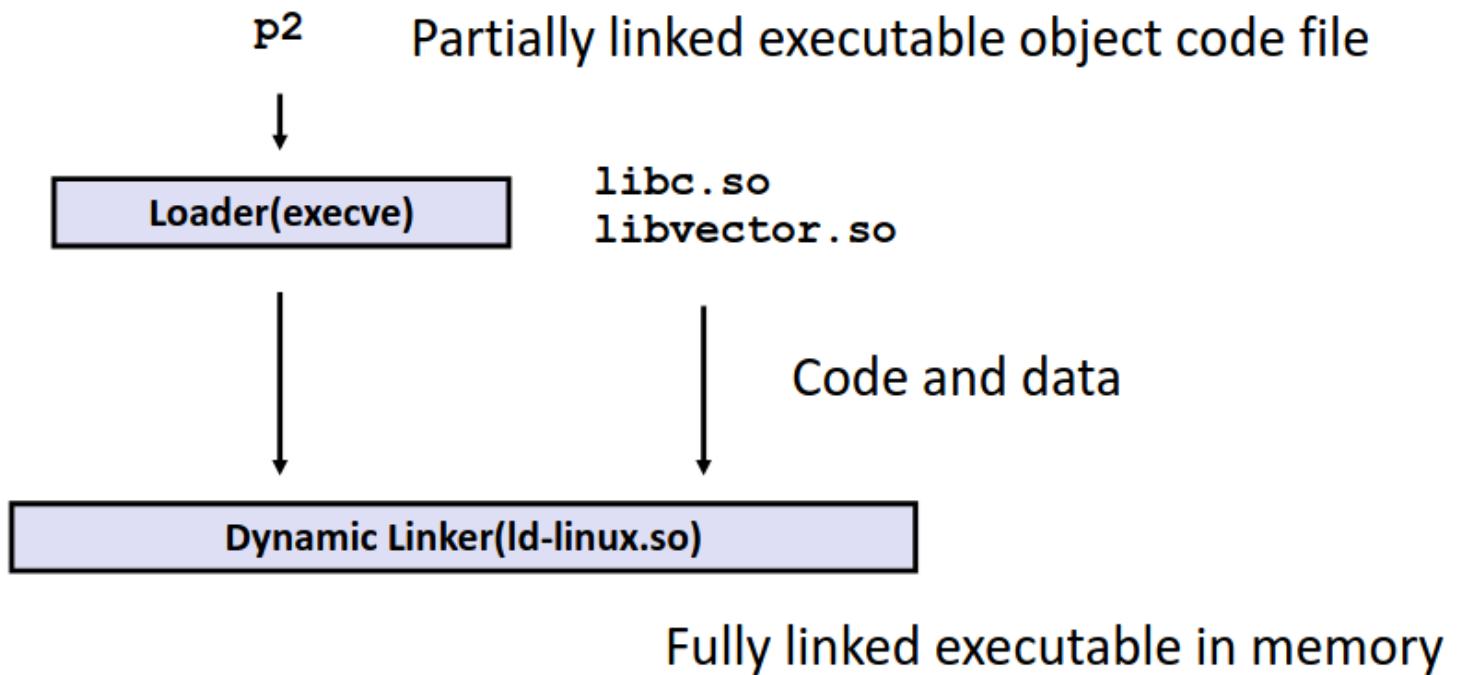
正如其名，共享库，我的 libc 在计算机硬盘上仅有一份，对于每个包含了这个库的程序我并不会复制这个库，而是当我在运行时再将库链接进程序中。甚至对于同时执行的进程我也同样共享一个库，即在主存中也同样只包含一份库的代码。

在对程序打包时我只会部分的链接。

## Partially Linking with Shared Libraries



当开始加载入内存时我再完全链接。



动态库更为强大的点在于其甚至可以再运行时再链接，本文不再过多介绍。

动态库完美解决了静态库的所有缺点，节省了资源，对于库函数有改动时我重新分发库就好了，反正运行时再链接。

但极高的灵活性也必然伴随着代价，一个显而易见的代价就是由于运行时需要额外链接，无疑降低了程序启动的速度。同时在一些对稳定性与安全性要求较高的场合，动态库往往也不是一个合适的选择。

---

© 2025. ICS Team. All rights reserved.

# Chapter 8 Exceptional Control Flow

---

© 2025. ICS Team. All rights reserved.

# Chapter 8.1 Exceptions

## Introduction

从给处理器加电开始，程序计数器假设一个值的序列  $a_0, a_1, \dots, a_{n-1}$ 。其中每个  $a_k$  是某个相应的指令  $I_k$  的地址。每次从  $a_k$  到  $a_{k+1}$  的过渡称为**控制转移**（control transfer）。这样的控制转移序列叫做处理器的**控制流**（control flow）。

最简单的一种控制流是一个“平滑的”序列，这种平滑流的突变通常是由诸如跳转、调用和返回这样一些指令造成的，这些都是使程序能够对由程序变量表示的内部程序状态变化作出反应的必要机制。

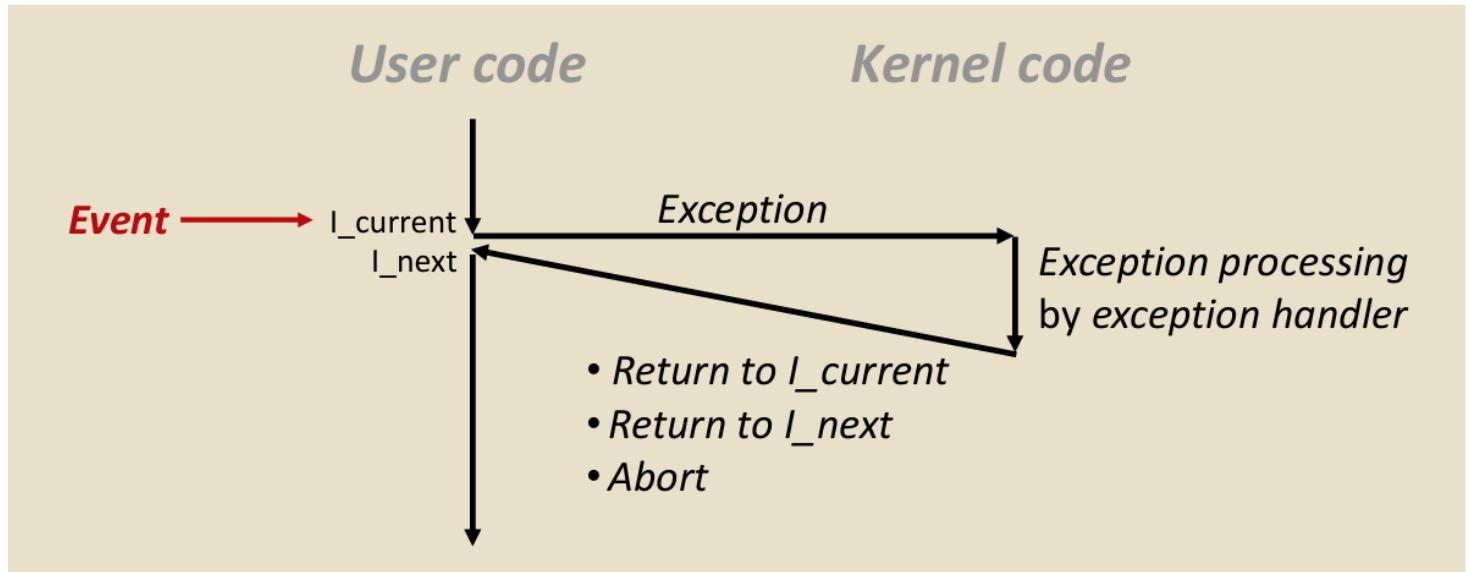
但同时系统也要对系统状态变化做出反应，而这些系统状态不是被内部程序变量捕获的，也不一定与程序的执行相关。那么现代系统就通过使控制流发生突变来对这些情况做出反应，即**异常控制流**（Exceptional Control Flow, ECF）。

异常控制流发生在计算机系统的各个层次。本章将介绍存在于一个计算机系统中所有层次上的各种形式的ECF。我们将由低层到高层进行讲解，涵盖异常、系统调用、进程和信号，以及非本地跳转等内容。

## Exceptions

异常是异常控制流的的一种形式，简单来说就是控制流中的突变，用来响应处理器状态中的某些变化。它位于硬件和操作系统交界的部分，因而一部分由硬件实现，一部分由操作系统实现。

下图展示了异常的基本思想：

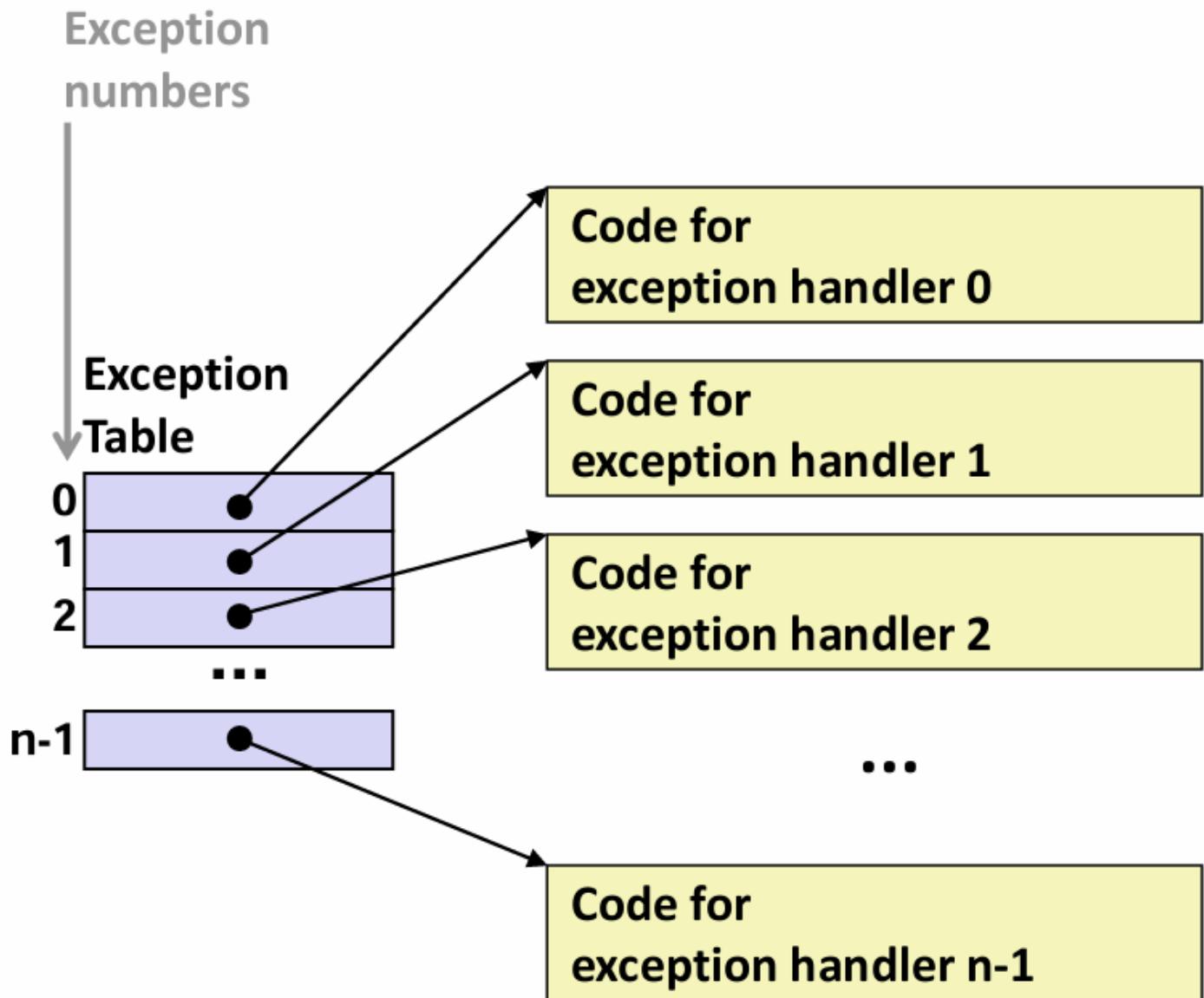


在图中，处理器正在执行某个当前指令`I_current`，此时处理器状态中发生一个重要变化，这种状态变化称为**事件**（event）。当处理器检测到有事件发生时，它会通过一张叫做**异常表**（exception table）的跳转表，进行一个间接过程调用（即异常）到一个专门设计用来处理这类事件的操作系统子程序——**异常处理程序**（exception handler，它运行在内核模式下）。完成处理后，根据引起异常的事件类型，会发生以下三种情况中的一种：

1. 处理程序将控制返回给当前指令`I_current`；
2. 处理程序将控制返回给`I_next`（即没有发生异常时将会执行的下一条指令）；
3. 处理程序终止被中断的程序。

## Exception Handling

系统中可能的每种类型的异常都分配了一个唯一的非负整数的**异常号**（exception number）。在系统启动时，操作系统分配和初始化一张异常表，使得表目k包含异常号为k的处理程序的地址。异常表格式如下图所示：



当系统在执行某个程序时，处理器检测到发生了一个事件，并确定了相应的异常号k。随后处理器触发异常（执行间接过程调用），通过异常表的表目k，转到相应的处理程序。

一旦硬件触发了异常，剩下的工作就是由异常处理程序在软件中完成。事件处理完之后，处理程序通过执行一条“从中断返回”指令可选地返回到被中断的程序，并将适当的状态弹回到处理器的控制和数据寄存器中。

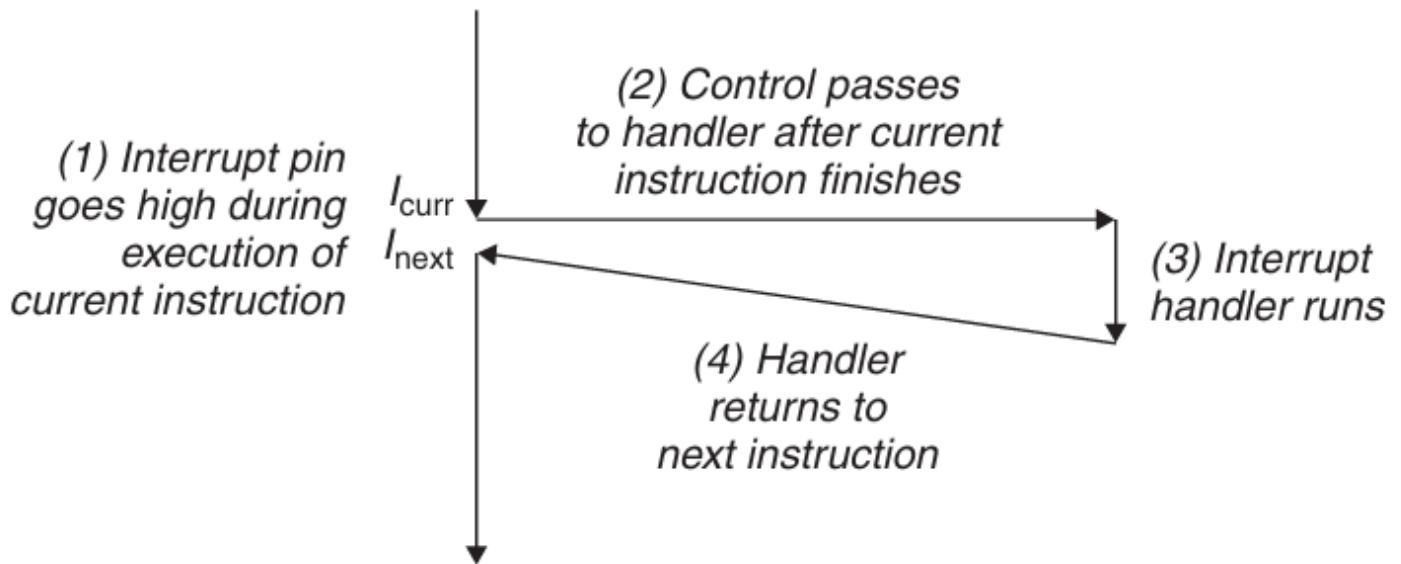
## Classes of Exceptions

异常可以分为四类：中断（interrupt）、陷阱（trap）、故障（fault）和终止（abort）。下表是对这些类别属性的小结：

| Class     | Cause                         | Async/sync | Return behavior                     |
|-----------|-------------------------------|------------|-------------------------------------|
| Interrupt | Signal from I/O device        | Async      | Always returns to next instruction  |
| Trap      | Intentional exception         | Sync       | Always returns to next instruction  |
| Fault     | Potentially recoverable error | Sync       | Might return to current instruction |
| Abort     | Nonrecoverable error          | Sync       | Never returns                       |

## Interrupts

中断是异步发生的，它不是由任何一条专门的指令造成的，是来自处理器外部I/O设备的信号的结果。I/O设备（如网络适配器、磁盘控制器等）通过向处理器芯片上的一个引脚发信号，并将异常号放到系统总线上来触发中断。下图概述了一个中断的处理：



从图中可以看到，中断处理程序返回时，是将控制返回给下一条指令。结果是程序继续执行，就好像没有发生过中断一样。

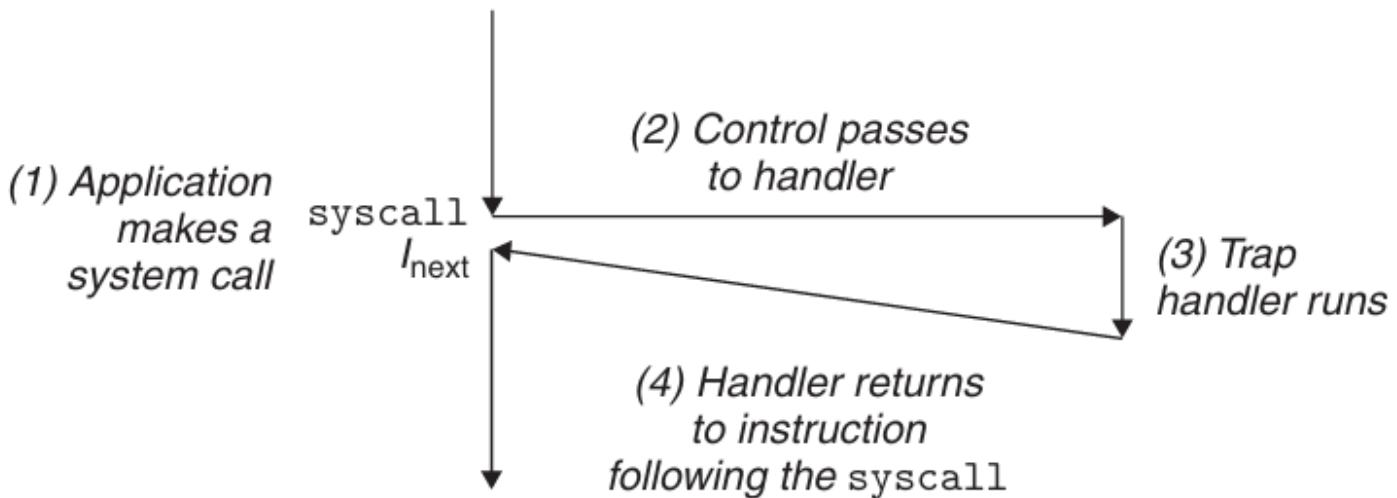
剩下的异常类型都是同步发生的，是执行当前指令的结果，我们把这类指令称为**故障指令**

(faulting instruction)。

## Traps and System Calls

陷阱是有意的异常，是执行一条指令的结果，陷阱处理程序也将控制返回到下一条指令。它最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，即**系统调用**。

用户程序经常需要向内核请求服务，如读文件（read）、创建新进程（fork）、加载新程序（execve）、终止当前进程（exit）等。为了允许对这些内核服务的受控的访问，处理器提供了一条特殊的“`syscall n`”指令，当用户程序想要请求服务n时，可以执行这条指令，结果就会导致一个到异常处理程序的陷阱。陷阱处理流程如下图所示：



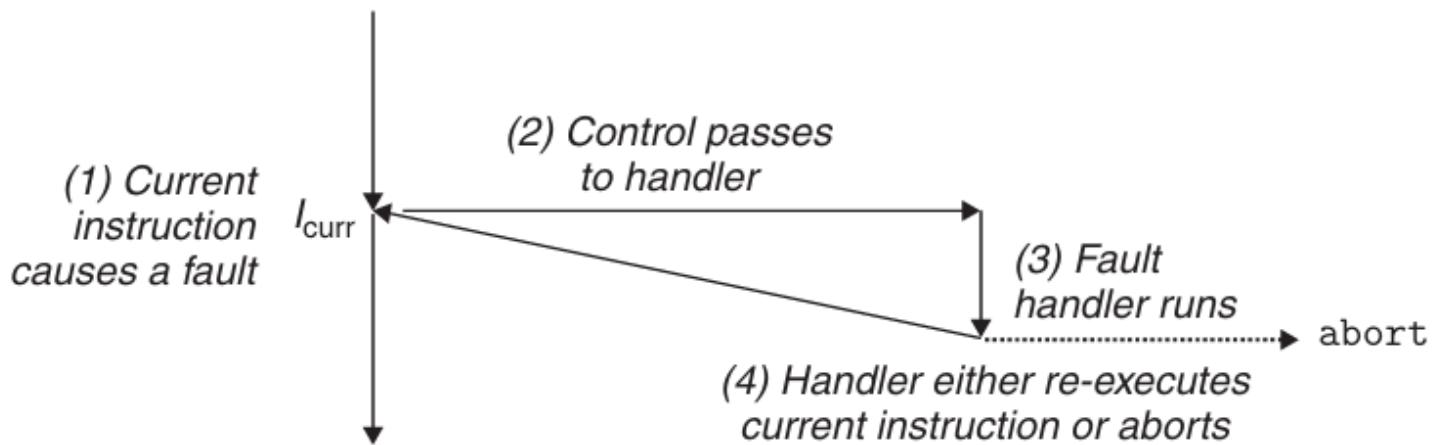

---

注：在程序员眼中，系统调用和普通的函数调用是一样的。然而它们的实现非常不同。普通的函数运行在**用户模式**中，函数可执行的指令的类型是受限的，且只能访问与调用函数相同的栈。而系统调用运行在**内核模式**中，允许执行特权指令，并访问定义在内核中的栈。

---

## Faults

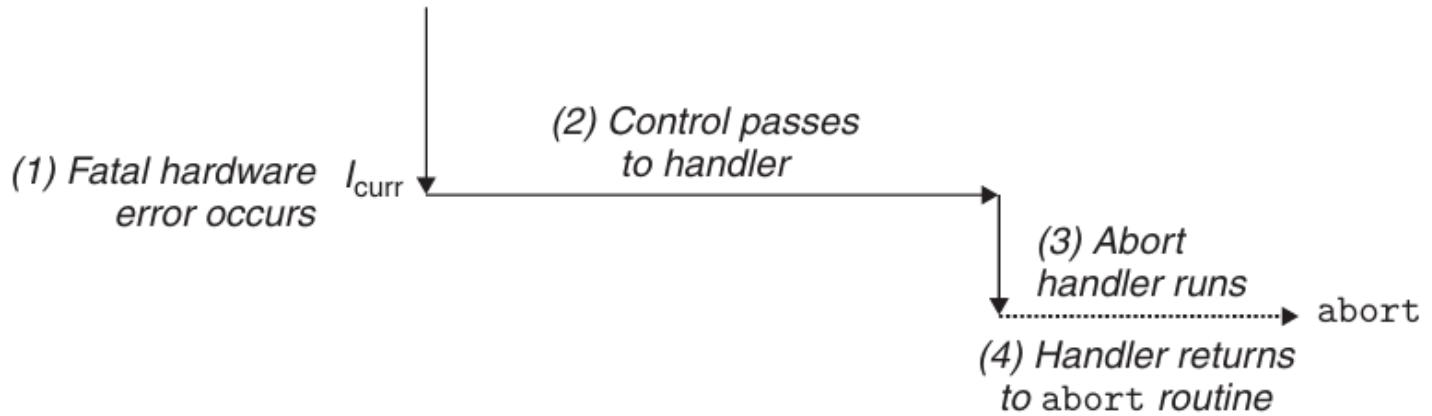
故障由错误情况引起，它可能能够被故障处理程序修正。故障发生时，处理器将控制转移给故障处理程序。如果故障处理程序能够修正该错误，它就将控制返回到引起故障的指令，从而重新执行它。否则返回到内核中的abort例程，abort例程会终止引起故障的应用程序。如下图所示：



一个典型的例子是缺页异常。当指令引用一个虚拟地址，而与该地址相对应的物理页面不在内存中，必须从磁盘中取出时，就会发生故障。缺页处理程序从磁盘加载适当的页面，然后将控制返回给引起故障的指令。当指令再次执行时，因为相应的物理页面已经驻留在内存中了，此时指令就可以没有故障地运行完成了。

## Abort

终止是不可恢复的致命错误造成的结果，通常是一些硬件错误。终止程序从不将控制返回给应用程序，如下图所示，处理程序将控制返回给一个abort例程，该例程会终止这个应用程序。



# Chapter 8.2 Processes

异常是允许操作系统内核提供**进程** (process) 概念的基本构造块，而进程是计算机科学中最深刻、最成功的概念之一。

在现代系统上运行一个程序时，我们会得到一系列的假象：我们的程序好像是系统中当前运行的唯一程序；它好像是独占地使用处理器和内存；处理器好像无间断地一条条执行我们程序中的指令；我们程序中的代码和数据好像是系统内存中唯一的对象。其实，这些假象都是通过进程的概念提供给我们的。

**进程是一个执行中程序的实例。**系统中的每个程序都运行在某个进程的**上下文** (context) 中。上下文是由程序正确运行所需的状态组成的，包括存放在内存中的代码和数据、它的栈、通用目的寄存器的内容、程序计数器等。

每次用户通过向shell输入一个可执行目标文件的名字，运行程序时，shell就会创建一个新进程，然后在这个新进程的上下文中运行该可执行文件。应用程序也能创建新进程，并在新进程的上下文中运行它们自己的代码或其它应用程序。

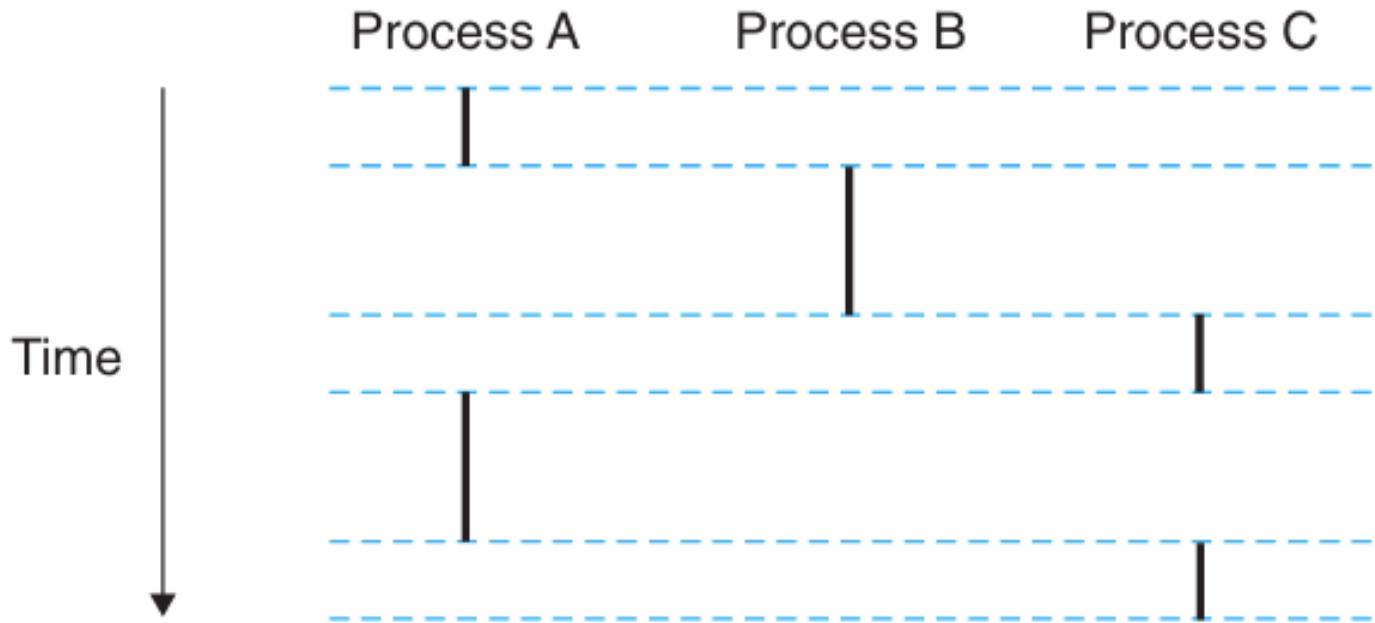
本节我们将关注进程提供给应用程序的关键抽象：

- 一个独立的逻辑控制流：提供程序独占使用处理器的假象。
- 一个私有的地址空间：提供程序独占使用内存系统的假象。

## Logical Control Flow

如果我们用调试器单步执行程序，就会看到一系列程序计数器 (PC) 的值，这些值唯一地对应于包含在程序的可执行目标文件中的指令，或是包含在运行时动态链接到程序的共享对象中的指令。这个PC值的序列叫做**逻辑控制流**，简称逻辑流。

举个例子，假如一个系统运行着三个进程，如下图所示，处理器的一个物理控制流被分成了三个逻辑流，每个进程一个。每条竖线代表了一个进程的逻辑流的一部分。这三个逻辑流的执行是交错的，过程是A->B->C->A->C，三个进程轮流使用处理器，直到三个进程都执行完毕。而对于一个运行在这些进程之一的上下文中的程序，它看上去就像是在独占地使用处理器。



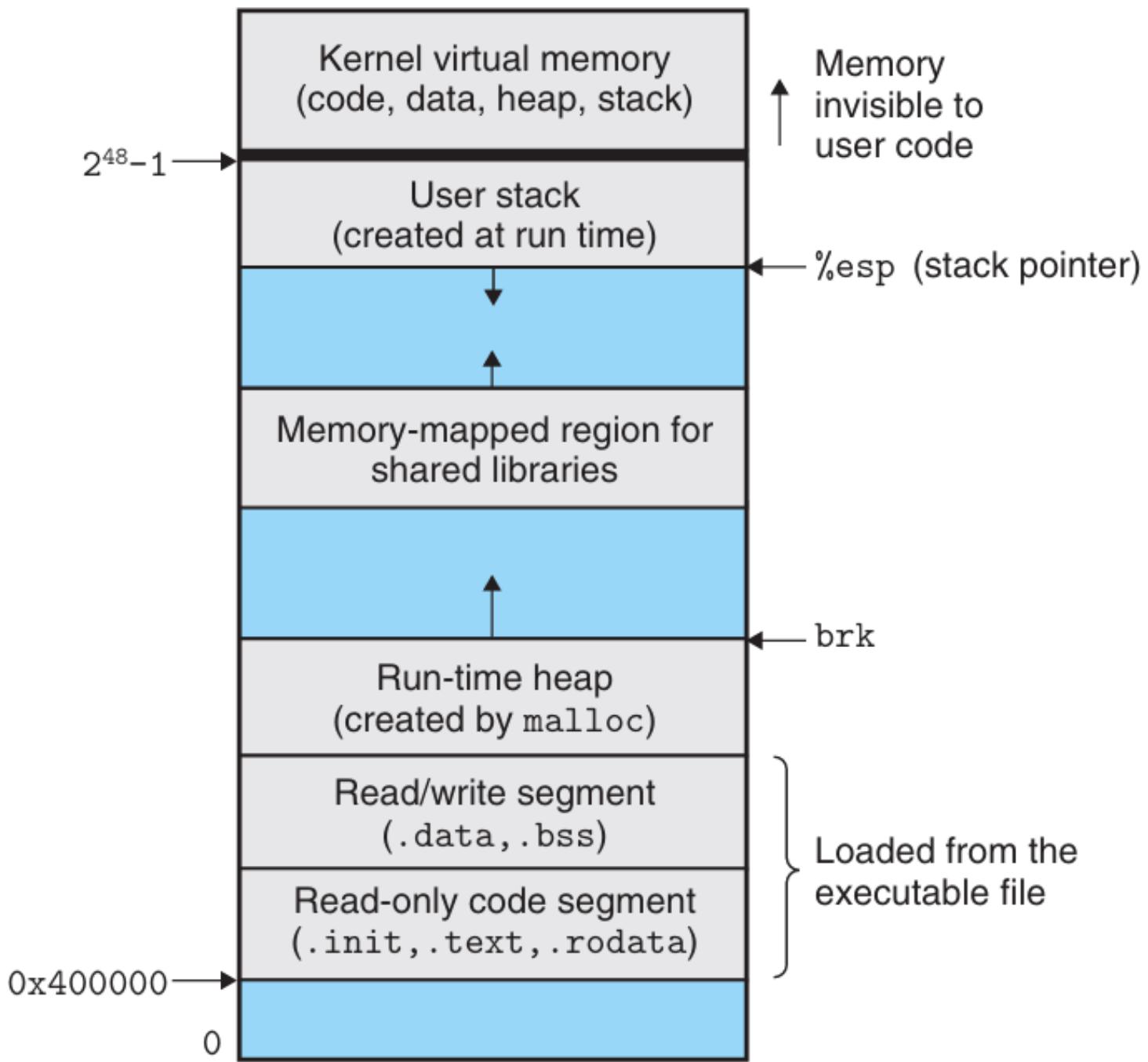
## Concurrent Flows

一个逻辑流的执行在时间上与另一个流重叠，称为**并发流**（concurrent flow），这两个流被称为并发地运行。例如在上例中，A和B并发运行，A和C并发运行，但B和C没有并发运行。

应当注意的是，并发流的思想与流运行的处理器核数或者计算机数无关。如果两个流在时间上有重叠，那它们就是并发的，即使它们是在同一个处理器上。而针对运行在不同处理器核或计算机上的两个并发运行的流，我们称之为**并行流**（parallel flow）。

## Private Address Space

前面提到，进程为每个程序提供一种假象，好像它独占地使用系统地址空间。下图展示了一个x86-64 Linux进程的地址空间的组织结构。



地址空间底部是保留给用户程序的，包括通常的代码、数据、堆和栈段。代码段总是从 $0x400000$ 开始，地址空间顶部保留给内核，该部分包含内核在代表进程执行指令时使用的代码、数据和栈。

## User and Kernel Modes

用户模式和内核模式限制了一个应用可以执行的指令和可以访问的地址空间范围。

处理器通常用某个控制寄存器的一个模式位（mode bit）来提供这种功能，该寄存器描述了进程当前享有的特权。当设置了模式位时，进程就运行在内核模式中，可以执行指令集的任何命令，访问系统中的任何内存位置。而没有设置模式位时，进程运行在用户模式中，不允许执行特权指令，比如停止处理器，发起一个I/O操作等，也不允许直接引用地址空间中内核区内的代码和数据，必须通过系统调用接口间接访问内核代码和数据。

运行应用程序代码的进程初始时在用户模式中。进程从用户模式变为内核模式的唯一方法是通过中断、故障或陷入系统调用等异常。异常发生时，控制传到异常处理程序，处理器将模式从用户模式变为内核模式。处理程序运行在内核模式中，当他返回到应用程序代码时，处理器就把模式从内核模式改回到用户模式。

## Context Switches

操作系统内核使用一种称为**上下文切换**（context switch）的较高层形式的异常控制流来实现多任务，这种机制是建立在我们前面介绍的较低层异常机制之上的。

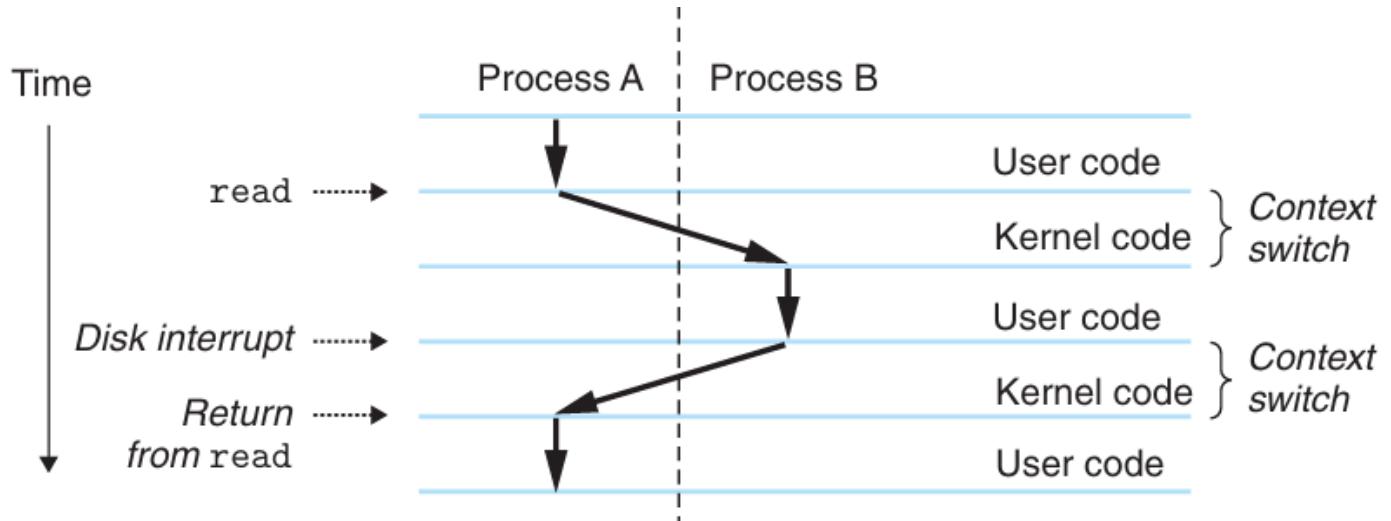
内核为每个进程维护一个上下文（context），它是内核重新启动一个被抢占的进程所需的状态，由一些对象的值组成，包括通用目的寄存器、程序计数器、用户栈和各种数据结构等。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个先前被抢占了的进程，这种决策叫做**调度**（scheduling）。当内核选择一个新的进程时，我们说内核调度了这个进程。在内核调度了一个新的进程运行后，它就抢占当前进程，并使用上下文切换的机制来将控制转移到新进程。上下文切换的功能如下：

1. 保存当前进程的上下文。
2. 恢复某个先前被抢占的进程被保存的上下文。
3. 将控制传递给这个新恢复的进程。

下图展示了一对进程A和B之间上下文切换的示例。进程A初始在用户模式，直到它通过执行系统调用read陷入到内核，此时内核中的陷阱处理程序请求来自磁盘控制器的DMA传输。由于磁盘取数据要用一段相对较长的时间，所以内核执行从进程A到进程B的上下文切换。随后，进程B在用户模式下运行直到磁盘发出一个中断信号，表示数据已经从磁盘传到了内存。内核因此执行一个从进程B到进程A的上下文切换，将控制返回给进程A中紧随在系统调用read之后的那条指令，进程A继续

运行。



---

© 2025. ICS Team. All rights reserved.

# Chapter 8.3 Process Control

本节将描述一些从C程序中操作进程的系统调用函数，并举例说明如何使用它们。

## Obtaining Process IDs

每个进程都有一个唯一的进程ID（PID），它总是大于零的。`getpid` 函数返回调用进程的PID，而 `getppid` 函数返回它的父进程的PID。

这两个函数的声明如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

## Creating and Terminating Processes

在任何时候，进程总是处于以下四种状态之一：

- Running：进程要么在CPU上执行，要么在等待被执行且最终会被内核调度。
- Blocked/Sleeping：进程在发生某些外部事件（通常是 I/O）之前无法执行更多指令。
- Stopped：进程的执行被挂起（suspended），且不会调度。（用户行为引起）
- Terminated/Zombie：进程永远地停止了。（zombie状态是指一个终止了还未被回收的进程）

调用 `exit` 函数可以终止进程，它以 `status` 退出状态来终止进程：

```
#include <stdlib.h>
void exit(int status);
```

父进程通过调用 `fork` 函数创建一个新的运行的子进程：

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

新创建的子进程得到与父进程用户级虚拟地址空间相同但独立的一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这意味着父进程调用 `fork` 时，子进程可以读写父进程中打开的任何文件。父进程和子进程最大的区别在于它们有不同的PID。

`fork` 函数的特点是：**只调用一次，但返回两次**。一次在调用进程中，一次在新创建的子进程中。在父进程中，`fork` 返回子进程的PID（非零）；在子进程中，`fork` 返回0。因此可以通过返回值分辨程序是在父进程中还是子进程中执行。

下面展示了一个使用 `fork` 创建子进程的示例。当 `fork` 调用返回时，父进程和子进程中的x的值都为1.子进程执行 $x+1$ 并输出，父进程执行 $x-1$ 并输出。（注：`Fork` 是 `fork` 的错误处理包装函数）

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;
    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }
    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

而当我们运行这个程序时，每次运行结果都不一定一样：

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

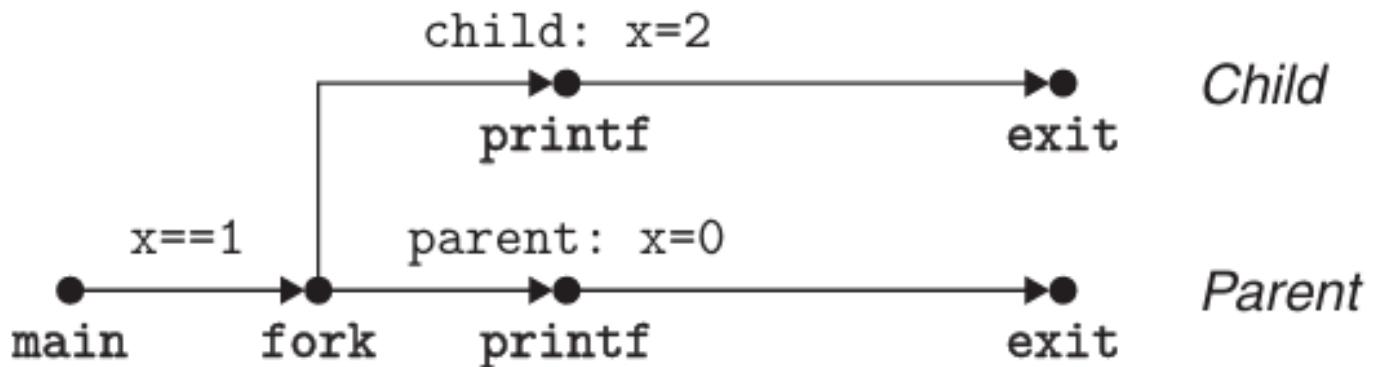
```
linux> ./fork
parent: x=0
child : x=2
```

这个例子能够展现 `fork` 的一些微妙之处：

- 调用一次，返回两次。
- 子进程和父进程是并发运行的独立进程。内核能够以任意方式交替执行它们的逻辑控制流中的指令。如上图所示，父进程和子进程完成 `printf` 语句的先后顺序并不固定，我们无法对不同进程中指令的交替执行做任何假设。
- 相同但独立的地址空间。本例中，当 `fork` 函数返回时，本地变量 `x` 在父进程和子进程中都为 1。而父进程和子进程对 `x` 所做的任何改变都是独立的，不会反映到另一个进程的内存中。因此它们各自的 `printf` 语句输出结果不同。
- 共享文件。子进程继承了父进程所有的打开文件。当父进程调用 `fork` 时，`stdout` 文件是打开的，且指向屏幕。因此子进程的输出也指向屏幕。

画进程图可以很好的帮助我们理解 `fork` 函数。进程图是刻画程序语句的偏序的一种简单的前趋图。每个顶点 `a` 对应于一条程序语句的执行，有向边 `a->b` 表示语句 `a` 发生在语句 `b` 之前。边上可以标注一些信息，如变量的当前值和输出内容。

下图为上例的进程图：



在进程图中，由有向边相连的两个语句在执行时有严格的先后顺序，比如 `fork` 一定在 `printf` 前执行，而没有有向边相连的语句在执行时的先后顺序是不定的，比如父进程和子进程的 `printf` 语句。从中我们也可以很容易地观察出子进程与父进程的并行性。

## Reaping Child Processes

当一个进程终止时，由它的父进程负责回收。当父进程回收已经终止的子进程时，内核将子进程的退

出状态传递给父进程，然后抛弃已终止的进程，从此时开始，该进程就不存在了。一个终止了但还未被回收的进程称为**僵死进程**（zombie）。

如果一个父进程终止了，内核会安排init进程成为它孤儿进程的养父。init进程的PID为1，是在系统启动时由内核创建的，它不会终止，是所有进程的祖先。

一个进程可以通过调用 `waitpid` 函数来等待它的子进程终止或停止。

```
#include<sys/types.h>
#include<sys/wait.h>
pid_t waitpid(pid_t pid, int *statusp, int options);
```

默认情况下（当`options=0`时），`waitpid` 挂起调用进程的执行，直到它的等待集合（wait set）中的一个子进程终止。如果等待集合中的一个进程在刚调用的时候就已经终止了，那么 `waitpid` 就立即返回。在这两种情况中，`waitpid` 返回导致它返回的已终止子进程的PID。`waitpid` 函数可以选择等待某个指定的子进程或所有子进程。

`wait` 函数是 `waitpid` 函数的简单版本：

```
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int *statusp);
```

如果成功，`wait` 函数返回子进程的PID；如果出错，则返回-1。

---

注：如果等待的是多个子进程，程序不会按照特定的顺序回收子进程。

---

## Putting Processes to Sleep

`sleep` 函数将一个进程挂起一段指定的时间。

```
#include<unistd.h>
unsigned int sleep(unsigned int secs);
```

如果请求的时间到了，`sleep` 会返回0，否则返回剩下的休眠秒数。

另一个很有用的函数是 `pause`，该函数让调用函数休眠，直到该进程收到一个信号。

```
#include <unistd.h>
int pause(void);
```

## Loading and Running Programs

`execve` 函数在当前进程的上下文加载并运行一个新程序。

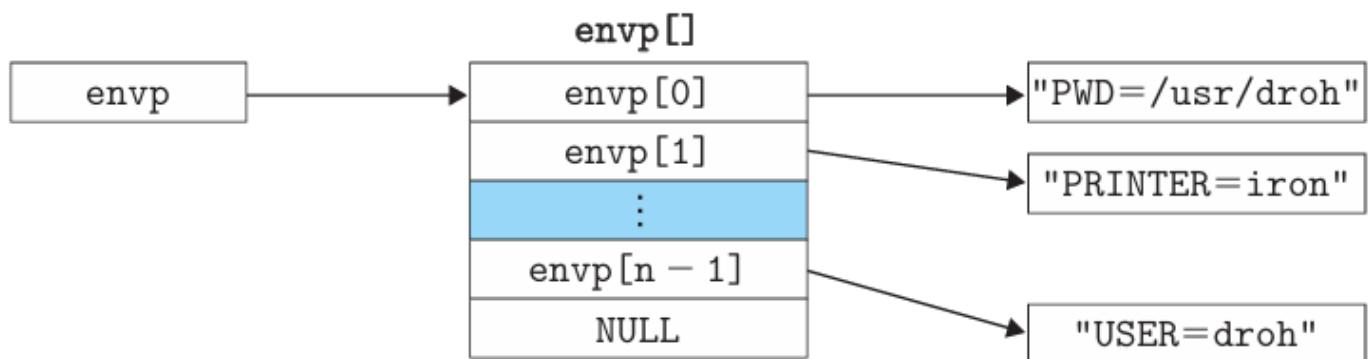
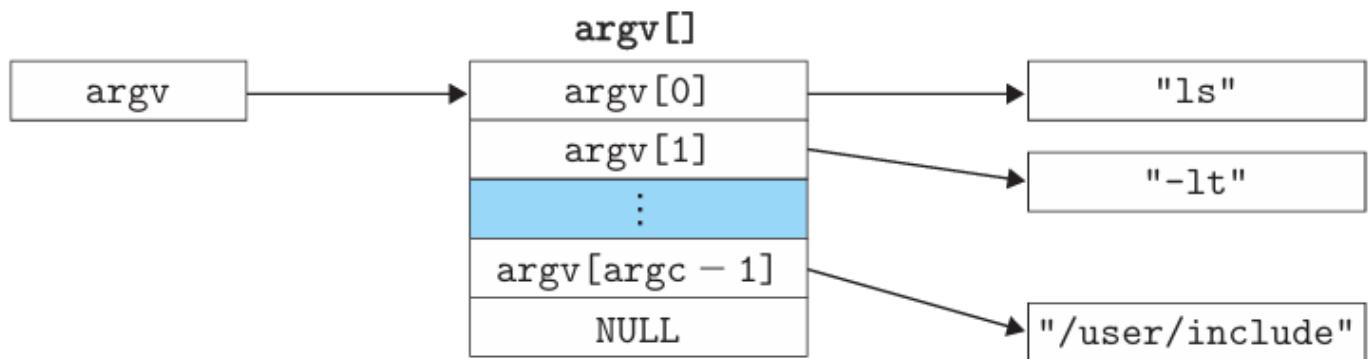
```
#include <unistd.h>
int execve(const char *filename, const char *argv[], const char *envp[]);
```

`execve` 函数加载并运行可执行文件 `filename`，且带参数列表 `argv` 和环境变量 `envp`。只有当出现错误时（例如找不到 `filename`），`execve` 才会返回到调用程序。所以，`execve` 的特点是：调用一次并从不返回。

`execve` 函数的参数列表和环境变量列表是由结构类似的数据结构表示的，如下图所示。

参数列表中，`argv` 变量指向一个以 `null` 结尾的指针数组，其中每个指针都指向一个参数字符串。按照惯例，`argv[0]` 是可执行目标文件的名字。

类似地，在环境变量列表中，`envp` 变量指向一个以 `null` 结尾的指针数组，其中每个指针指向一个环境变量字符串，每个串都是形如 `name=value` 的名字—值对。



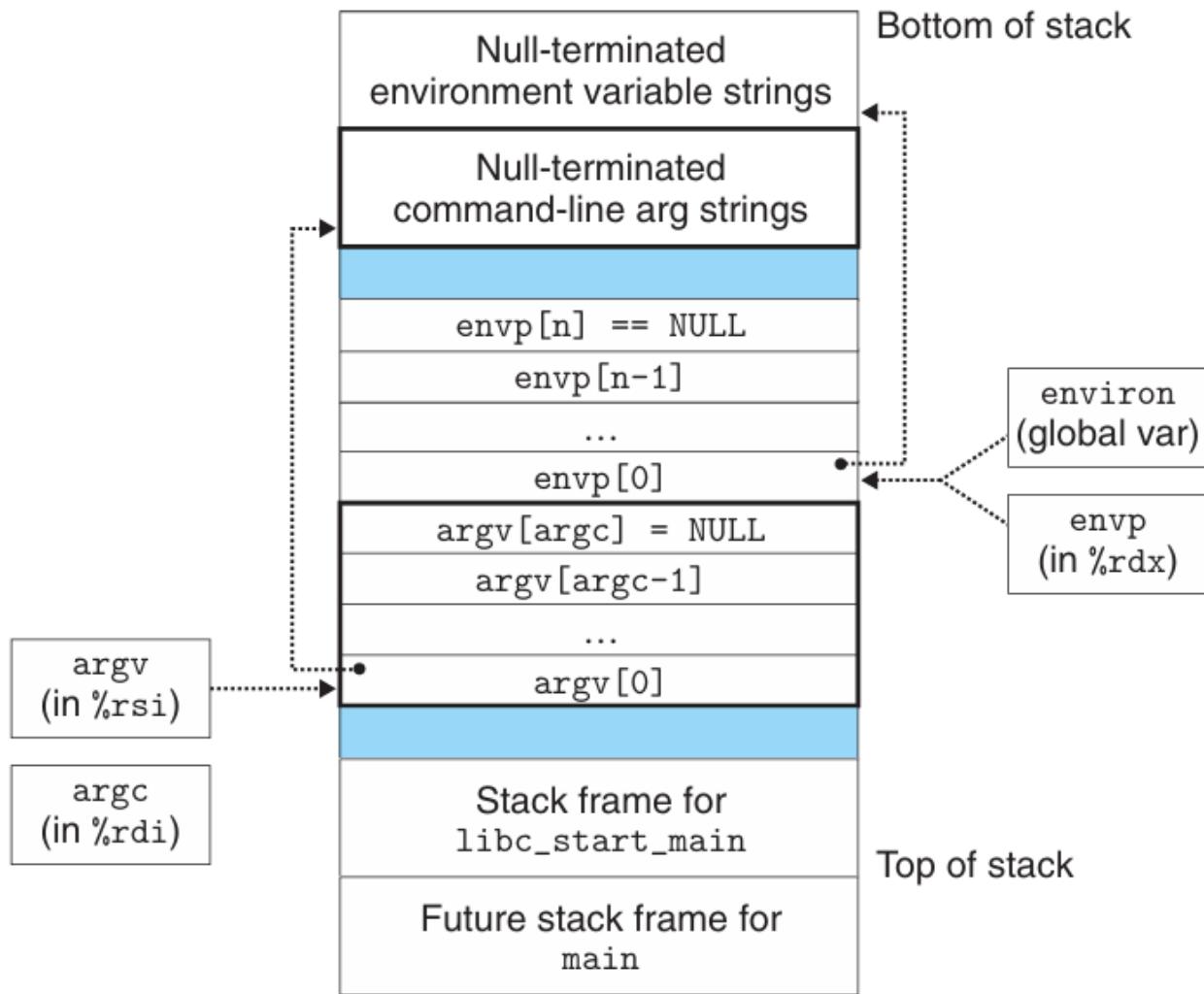
在 `execve` 加载了 `filename` 之后，它会调用启动代码，启动代码设置栈，并将控制传递给新程序的 `main` 函数，`main` 函数原型为

```
int main(int argc, char **argv, char **envp);
```

或等价的

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

当 `main` 开始执行时，用户栈的组织结构如下图所示。从栈底（高地址）到栈顶（低地址），首先是环境变量字符串和命令行字符串。之后是以 `null` 结尾的指针数组，其中每个指针都指向栈中的一个环境变量字符串，全局变量 `environ` 指向这些指针中的第一个 `envp[0]`。环境变量数组之后是以 `null` 结尾的 `argv[]` 指针数组，其中每个指针都指向栈中的一个参数字符串。栈顶则是系统启动函数 `libc_start_main` 的栈帧。




---

注：在 main 函数中，参数 argc 表示 argv[] 数组中非空指针的数量。

---

## Simple Shell Example

shell是一个交互型的应用级程序，它代表用户运行其他程序。shell的基本操作就是执行一系列的**读/求值** (read/evaluate) 步骤，然后终止。读步骤读取来自用户的一个命令行；求值步骤解析命令行，并代表用户运行程序。

Unix shell中大量使用了 fork 和 execve 函数，下面我们以一个简单的shell程序为例，学习shell工作的基本原理以及如何利用 fork 和 execve 运行程序。

首先是一个简单shell的 main 函数架构：

```
int main(int argc, char** argv)
{
    char cmdline[MAXLINE]; /* command line */
    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

我们可以看到shell的基本结构就是一个简单的循环。在每次循环中，它会先打印出一个命令行提示符，等待用户在 `stdin` 上输入命令行，然后调用 `eval` 对该命令行求值。

下面是对命令行求值的 `eval` 函数的核心部分：

```

void eval(char * cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }
    }

    /* Parent waits for foreground job to terminate */
    if (!bg) {
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitfg: waitpid error");
    }
    else
        printf("%d%s", pid, cmdline);
}
return;
}

```

`eval` 函数的结构大致可分为以下几个部分：

- 首先是调用 `parseline` 函数，其作用是解析以空格分隔的命令行参数，并构造最终传递给 `execve` 的 `argv` 向量。如果最后一个参数是 `&`，那么 `parseline` 返回1，表示在后台执行该程序，即shell不会等待它完成；否则返回0，表示应在前台执行完此程序。
- 完成命令行解析后，`eval` 接着调用 `builtin_command` 函数，检查第一个命令行参数是否为一个内置的shell命令（在我们这个简单shell中，只有一个内置命令 `quit`，该命令会终止 shell）。如果是，它就会立即解释该命令，并返回1；否则返回0。
- 如果 `builtin_command` 返回0，shell就会创建一个子进程，并在子进程中执行所请求的程序。如果用户要求在后台运行该程序，shell就会打印该进程的PID，然后返回到循环的顶

部，等待下一个命令行。否则，使用 `waitpid` 函数等待作业终止。当作业终止时，shell 就开始下一轮迭代。

---

注：Unix shell 用作业（job）这个抽象概念来表示为对一条命令行求值而创建的进程。

---

上面的 Simple Shell Example 看起来已经实现了必要的功能，但是它却存在一个巨大的缺陷：它没有回收后台运行的子进程。这就会导致这些子进程终止时，它们会成为僵死进程（zombies），而它们永远也不会被回收，因为 shell 一般情况下是不会终止的。

要解决这样一个问题，就需要使用信号。我们将在下一节详细讲述它。

---

© 2025. ICS Team. All rights reserved.

# Chapter 8.4 Signals

到目前为止，我们已经看到了硬件和软件是如何合作以提供基本的低层异常机制的，以及操作系统如何利用异常来支持进程上下文切换的异常控制流形式。在本节中，我们将研究一种更高层的软件形式的异常，称为Linux信号，它允许进程和内核中断其他进程。

一个信号就是一条小信息，用于通知进程系统中发生了一个某种类型的事件。下面列举了几种常用的Linux信号。

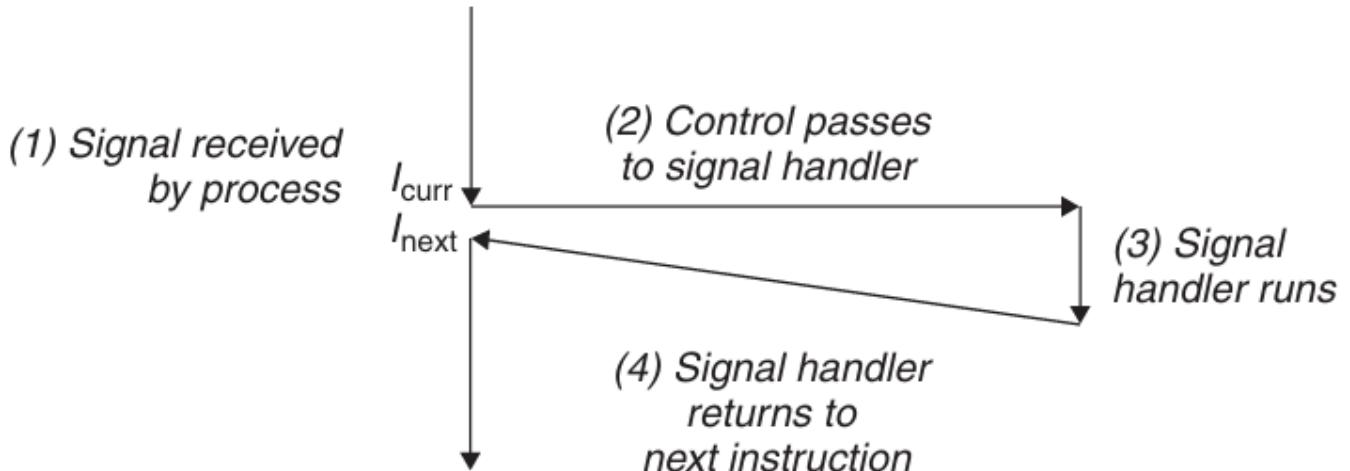
| <b>ID</b> | <b>Name</b> | <b>Default Action</b> | <b>Corresponding Event</b>               |
|-----------|-------------|-----------------------|--|
| 2         | SIGINT      | Terminate             | User typed ctrl-c                        |
| 9         | SIGKILL     | Terminate             | Kill program (cannot override or ignore) |
| 11        | SIGSEGV     | Terminate             | Segmentation violation                   |
| 14        | SIGALRM     | Terminate             | Timer signal                             |
| 17        | SIGCHLD     | Ignore                | Child stopped or terminated              |

每个信号类型都对应于某种系统事件。低层的硬件异常是由内核异常处理程序处理的，所以在正常情况下对用户进程不可见。而信号就提供了这样一种机制，通知用户进程发生了这些异常。比如我们熟悉的SIGSEGV信号，就是内核在进程进行非法内存引用时发送给进程的。

## Signal Concepts

传送一个信号到目的进程是由两个不同步骤组成的：

1. **发送信号** (Sending a Signal)。内核通过更新目的进程上下文中的某个状态，发送一个信号给目的进程。发送信号可以有如下两种原因：(1) 内核检测到一个系统事件，比如除零错误或子进程终止。(2) 一个进程调用了 `kill` 函数，显式地要求内核发送一个信号给目的进程。一个进程也可以发送信号给它自己。
2. **接收信号** (Receiving a Signal)。当目的进程被内核强迫以某种方式对信号的发送做出反应时，它就接收了信号。进程对该信号的反应可以是**忽略** (Ignore)，**终止** (Terminate) 或者通过执行信号处理程序 (signal handler) 的用户层函数来**捕获** (Catch) 该信号。下图展示了捕获信号的基本思想：



基本流程是：接收到信号会触发控制转移到信号处理程序，在信号处理程序完成处理后，它将控制返回给被中断的程序。

一个发出而没有被接收的信号叫做**待处理信号**（pending signal）。在任何时刻，一种类型至多只有一个待处理信号。如果一个进程有一个类型为k的待处理信号，那么任何接下来发送到这个进程的类型为k的信号都不会排队等待，而是被简单地丢弃。

一个进程可以选择地**阻塞**（block）接收某种信号。当一种信号被阻塞时，它仍然可以发送，但是产生的待处理信号不会被接收。

一个待处理信号最多只能被接收一次。

## Sending a Signal

每个进程都只属于一个**进程组**（process group），进程组由一个进程组ID（正整数）来标识。默认情况下，一个子进程和它的父进程同属于一个进程组。

## Sending Signals with /bin/kill Program

/bin/kill程序可以向另外的进程发送任意信号。比如，命令

```
/bin/kill -9 24818
```

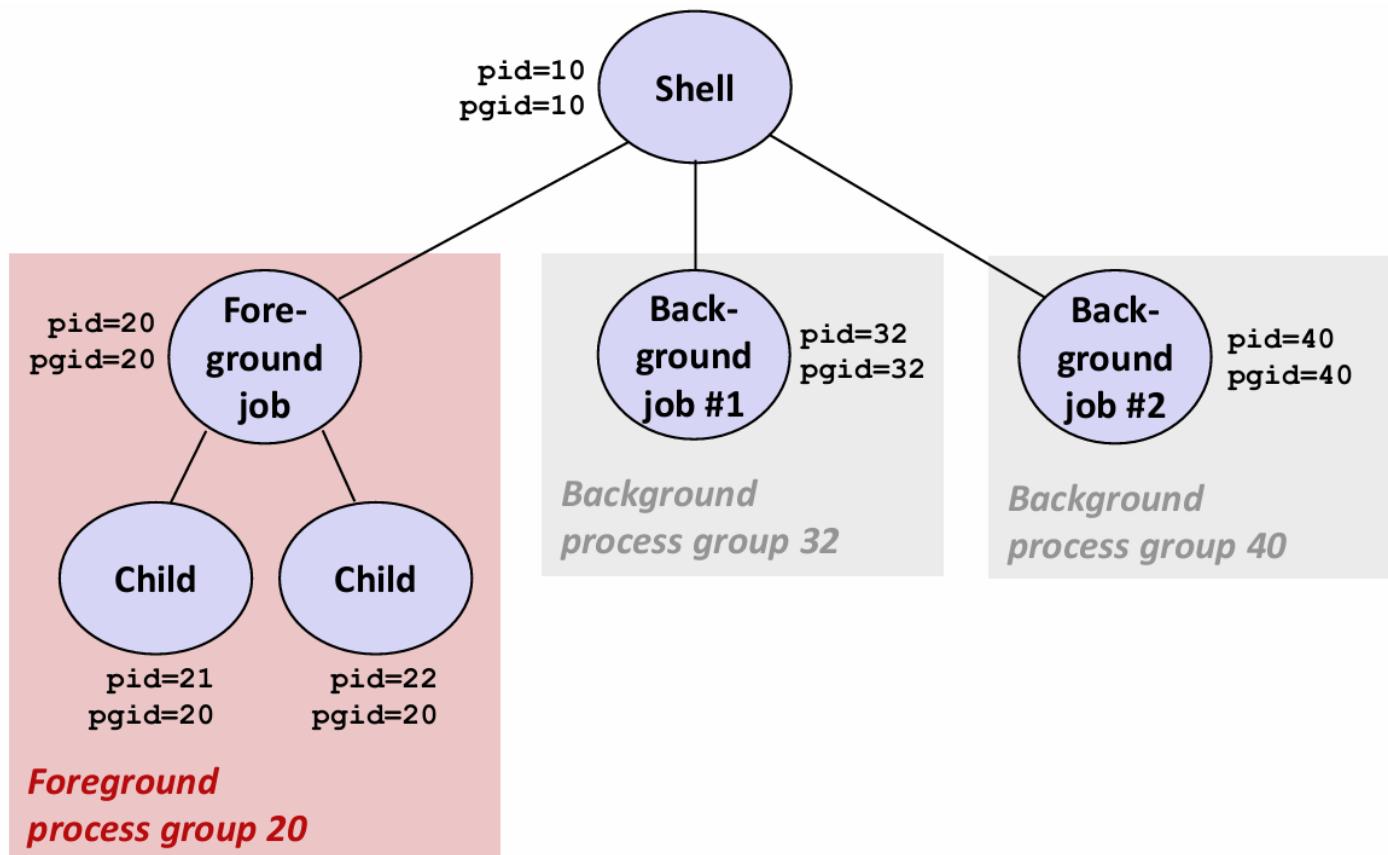
会发送信号9 (SIGKILL) 给进程24818。而一个负的PID会导致信号被发送到进程组PID中的每个进程，所以如果将上面的命令改为

```
/bin/kill -9 -24818
```

就会发送一个SIGKILL信号给进程组24818中的每个进程。

## Sending Signals from the Keyboard

在Unix shell中，任何时刻至多有一个前台作业和0个或多个后台作业。shell会为每个作业创建一个独立的进程组，进程组ID通常取自作业中父进程中的一个。下图展示了有一个前台作业和两个后台作业的shell。



在键盘上输入Ctrl+C会导致内核发送一个SIGINT信号到前台进程组中的每个进程。默认情况下，结果是终止前台作业。类似地，输入Ctrl+Z会发送一个SIGTSTP信号到前台进程组中的每个进程。默

认情况下，结果是停止（挂起）前台作业。

## Sending Signals with kill Function

进程通过调用 `kill` 函数发送信号给其他程序（包括它们自己）。

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid, int sig);
```

- 如果pid大于零，那么kill发送信号sig给进程pid。
- 如果pid等于零，那么kill发送信号sig给调用进程所在进程组中的每个进程，包括调用进程自己。
- 如果pid小于零，kill发送信号sig给进组中的每个进程。

## Receiving a Signal

当内核把进程p从内核模式切换到用户模式时（比如我们前面讲的context switch），它会检查进程p的未被阻塞的待处理信号的集合（ $pnb = pending \& \sim blocked$ ）。如果集合为空，那么内核将控制传递到p的逻辑控制流中的下一条指令；如果集合非空，那么内核选择集合中的最小的非零信号k，并强制p接收信号k。收到这个信号会触发进程采取某种行为。一旦进程完成了这个行为，那么控制就传递回p的逻辑控制流中的下一条指令。

每个信号类型都有一个预定义的默认行为：

- 进程终止。
- 进程终止并转储内存。
- 进程停止（挂起）直到被SIGCONT信号重启。
- 进程忽略该信号。

进程可以通过使用 `signal` 函数修改和信号相关联的默认行为（除了SIGSTOP和SIGKILL）：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

`signal` 函数可以通过下列三种方法之一来改变和信号`signum`相关联的行为：

- 如果`handler`是`SIG_IGN`, 那么忽略类型为`signum`的信号。
- 如果`handler`是`SIG_DFL`, 那么类型为`signum`的信号行为恢复为默认行为。
- 否则, `handler`就是用户定义的函数的地址, 这个函数被称为信号处理程序, 只要进程接收到一个类型为`signum`的信号, 就会调用这个程序。通过把处理程序的地址传递到 `signal` 函数从而改变默认行为, 这叫做**设置信号处理程序** (installing the handler)。调用信号处理程序被称为**捕获信号**。执行信号处理程序被称为**处理信号**。

## Blocking and Unblocking Signals

Linux提供阻塞信号的隐式和显式机制：

**隐式阻塞机制。**内核默认阻塞任何当前处理程序正在处理信号类型的待处理信号。例如, 一个`SIGINT`处理程序不能被另一个`SIGINT`信号中断。

**显式阻塞机制。**应用程序可以使用 `sigprocmask` 函数和它的辅助函数, 明确地阻塞和解除阻塞选定的信号。

`sigprocmask` 函数原型如下：

```
#include<signal.h>
int sigprocmask(int how,const sigset_t *set,sigset_t *oldset);
```

下述辅助函数用于对`set`集合进行操作：

```
#include<signal.h>
int sigemptyset(sigset_t *set); // 初始化set为空集合
int sigfillset(sigset_t *set); // 把每个信号都添加到set中
int sigaddset(sigset_t *set,int signum); // 把signum添加到set
int sigdelset(sigset_t *set,int signum); // 从set中删除signum
int sigismember(const sigset_t *set,int signum); // 如果signum是set的成员, 返回1, 否则返回0
```

# Writing Signal Handlers

信号处理程序的一些属性使得程序员很难对它们进行推理分析：

1. 处理程序与主程序并发运行，共享同样的全局变量，因此可能与主程序和其他的处理程序互相干扰。
2. 如何以及何时接收信号的规则常常有违人的直觉。
3. 不同的系统有不同的信号处理语义。

接下来我们将讲述这些问题，介绍如何编写安全、正确、可移植的信号处理程序。

## Safe Signal Handling

由于信号处理程序和主程序以及其他信号处理程序并发运行，所以我们应该遵守一些较为保守的编写原则，使得这些处理程序能安全地并发运行。

G0. **处理程序要尽可能简单。** e.g. 简单地设置全局变量并立即返回。

G1. **在处理程序中只调用异步信号安全的函数。** 原因是：异步信号安全的函数要么是可重入的（例如只访问局部变量），要么是不能被信号处理程序中断的。需要注意的是，许多我们常用的函数，如 `printf`、`malloc`、`exit` 都不是异步信号安全的！

G2. **保存和恢复errno。**

G3. **阻塞所有的信号，保护对共享全局数据结构的访问。** 原因是从主程序访问一个数据结构d通常需要一系列指令，如果指令序列被访问d的处理程序中断，那么处理程序可能会发现d的状态不一致，得到不可预知的结果。而在访问d时暂时阻塞信号就保证了处理程序不会中断该指令序列。

G4. **用volatile声明全局变量。** 用volatile类型限定符来定义一个变量，作用是告诉编译器不要缓存该变量。例如：`volatile int g;` 所以每次在代码中引用g时，都要从内存中读取g的值。

G5. **用sig\_atomic\_t声明标志。** 在常见的处理程序设计中，处理程序会写全局标志（flag）来记录收到了信号。主程序周期性地读这个标志，响应信号，再清除该标志。用sig\_atomic\_t数据类型声明变量，保证了对该变量的读和写都是原子的（不可中断的），所以无需暂时阻塞信号。

## Correct Signal Handling

信号的一个与直觉不符的方面是未处理的信号是不排队的。因为pending位向量中每种类型的信号只对应一位，所以每种类型最多只能有一个未处理的信号。这里面的关键思想是：如果存在一个未处理的信号，就表明至少有一个信号到达了。

基于此我们可以得到：不可以用信号来对其他进程中发生的事件计数。因为如果来自其他进程的两个类型k的信号发送给一个目的进程，而目的进程当前正在执行信号k的处理程序，那么信号k就会被阻塞，第二个信号k就简单地被丢弃了。

所以要解决信号不会排队等待这一问题，我们应该在处理程序的每次循环中加入wait/waitpid函数，回收僵死程序。

## Portable Signal Handling

Unix信号处理的另一个缺陷在于不同的系统有不同的信号处理语义。比如signal函数的语义各有不同，系统调用可以被中断等。

我们引入包装函数Signal来解决这些问题，它的信号处理语义如下：

- 只有这个处理程序当前正在处理的那种类型的信号被阻塞。
- 信号不会排队等待。
- 只要可能，被中断的系统调用会自动重启。
- 一旦设置了信号处理程序，它就会一直保持，直到Signal带着handler参数为SIG\_IGN或SIG\_DFL被调用。

## Synchronizing Flows to Avoid Races

如何编写读写相同存储位置的并发流程序是一个十分棘手的问题。一般而言，流可能交错的数量与指令的数量呈指数关系。这些交错中有些会产生正确结果，有些则不会。

下面展示了Unix shell的典型结构，作为竞争（race）的同步错误的示例：

```

void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;
    sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (pid != 0 && errno != ECHILD)
        sio_error("waitpid error");
    errno = olderrno;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    int n = N; /* N = 5 */
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */
    while (n--) {
        if ((pid = fork()) == 0) { /* Child */
            execve("/bin/date", argv, NULL);
        }
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid); /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}

```

在这个程序中，由于它认为父进程比子进程先运行，导致main函数中调用addjob和处理程序中调用deletejob之间存在竞争。如果addjob赢得竞争，那么结果就是正确的；否则就是错误的。为了解决竞争问题，我们可以在调用fork之前，阻塞SIGCHLD信号，然后在调用addjob之后取消阻塞这些信号。这样就能保证子进程被添加到作业列表中之后回收该子进程。修改后的main函数如下：

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;
    int n = N; /* N = 5 */
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);
    signal(SIGCHLD, handler);
    initjobs(); /* Initialize the joblist */
    while (n--) {
        if ((pid = fork()) == 0) { /* Child process */
            sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
            if (execve("/bin/date", argv, NULL) != -1)
                _exit(0);
        }
        sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid); /* Add the child to the job list */
        sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    _exit(0);
}
```

## Explicitly Waiting for Signals

有时候主程序需要显式地等待某个信号处理程序运行。例如，当Linux shell创建一个前台作业时，在接收下一条用户指令之前，它必须等待作业终止，被SIGCHLD处理程序回收。

下面的代码展示了一个基本思路：父进程设置SIGINT和SIGCHLD的处理程序，然后进入一个无限循环，它阻塞SIGCHLD信号，避免产生竞争。创建子进程后，pid置为0，取消阻塞SIGCHLD，然后以循环的方式等待pid变为非零。子进程终止后，处理程序回收它，把它非零的PID赋给全局pid变量。这会终止循环，父进程继续其他工作，然后开始下一次迭代。

```

volatile sig_atomic_t pid;
void sigchld_handler(int s)
{
    int olderrno = errno;
    pid= waitpid(-1, NULL, 0); /* Main is waiting for nonzeropid*/
    errno= olderrno;
}
void sigint_handler(int s)
{
}

int main(int argc, char **argv) {
    sigset(SIGCHLD, sigchld_handler);
    signal(SIGINT, sigint_handler);
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    while (n--) {
        if (fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid= 0;
        sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */
        /* Wait for SIGCHLD to be received (wasteful!) */
        while(!pid)
            ;
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    printf("\n");
    exit(0);
}

```

上面这段代码虽然正确无误，但是运行起来会非常慢。我们可能想到的解决办法是在循环体内插入 pause：

```

while (!pid) /* Race! */
    pause();

```

但是它会带来严重的竞争：如果在while测试后和pause之前收到SIGCHLD信号，pause会永远睡眠。

另一个选择是用sleep替换pause：

```
while (!pid) /* Too slow! */
    sleep(1);
```

但是这样又太慢了，如果在while之后sleep之前收到信号，程序必须等相当长的一段时间才会再次检查循环的终止条件。

合适的解决方法是使用sigsuspend：

```
#include<signal.h>
int sigsuspend(const sigset_t *mask);
```

sigsuspend函数暂时用mask替换当前的阻塞集合，然后挂起该进程，直到收到一个信号。如果该信号的行为是终止，那么该进程不从sigsuspend返回就直接终止；如果信号行为是运行一个处理程序，那么sigsuspend从处理程序返回，恢复调用sigsuspend时原有的阻塞集合。

使用sigsuspend修改减少了浪费，同时还避免了引入pause带来的竞争以及sleep的低效率。

---

© 2025. ICS Team. All rights reserved.

# Chapter 9 Virtual Memory

---

© 2025. ICS Team. All rights reserved.

# Chapter 9.1 Concepts

虚拟内存，一个在前面章节中无数次提到的神秘概念，出没于程序的机器级表示、链接与异常控制流等等章节，这一章我们将正式介绍虚拟内存。

## Introduction

### Physical Address

现在给你一个机会，你将穿越回 50 年前由你来设计一套计算机内存管理的模式，很遗憾这时候的你还没有学过虚拟内存，所以你只能靠自己得小创意了。

内存就是一个个的位拼凑成一个个字节组成。显然每个内存的位置应当有个索引，不然无法访问到某块内存，你觉得给每个字节编一个编号是个好主意，毕竟给每个位都编一个号太多了。你决定把这个编号叫做内存的地址。

那么有了地址以后，我们可以拿着地址直接去访问内存了，这个地址直接对应了物理内存的地址，即**物理地址(Physical address, PA)**。

一个问题出现在你面前，对于每个物理地址，显然只能放一个数据，也就意味着程序的数据必须放在空的地址处，但是对于一个程序显然在编译时它并不知道运行时哪一块会是空的。这时候你想我们不是学过一种寻址方式叫**相对寻址(PC-relative)** 吗？这样在编译的时候就可以不用知道数据需要放在哪里了，只用在程序运行时，操作系统找块地方放着就好了！于是你大笔一挥，以后编译器都只准编译成相对寻址的模式！

这时候你意识到，计算机存储不只是主存，还有磁盘这个巨慢无比的设备，那么我的物理地址应该是表示磁盘上的数据还是主存上的呢？你想起学过的 cache，你觉得我们应该把**主存视为磁盘的缓存**，所以地址表示的是磁盘上的存储位置。然后每次内存访问到主存就像访问 cache 一样去找。

完成了这一套设计后你心满意足，你顺手编写了一个简单的操作系统，然后写了个小程序，一试系统直接崩了，你经过无数天的排查发现：小程序向一个未初始化的指针指向的地址写了一个奇怪的值，结果那个地址刚好是内核核心代码的位置，然后系统就挂了~~（为什么你都会写操作系统了还会犯这种错误）~~。

你意识到内存访问应当设置权限，不是每一块内存都可以让随便什么人访问。简单的解决办法是由

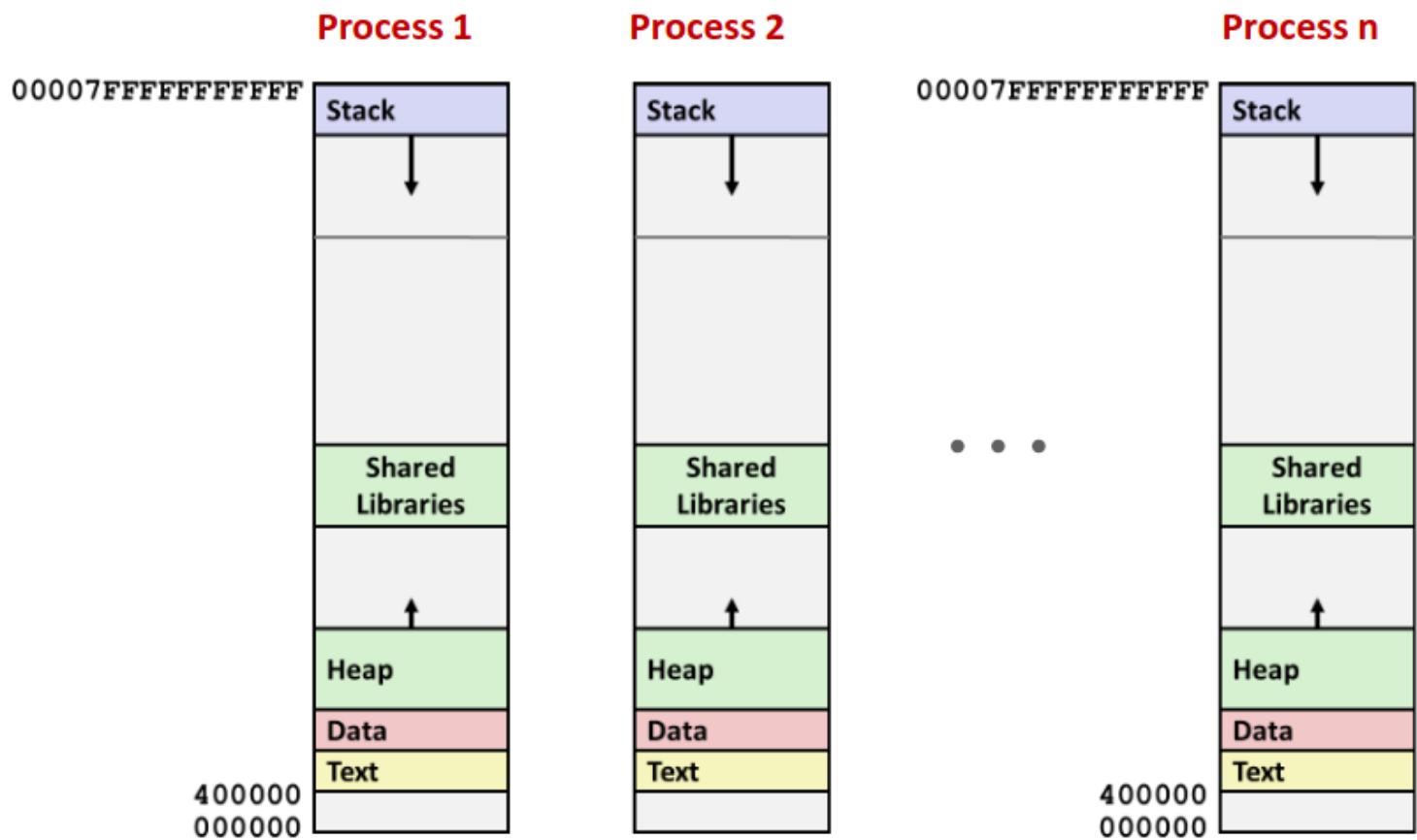
操作系统管理一张表，里边记录分配的内存，并记录权限，这样程序访问每块内存时都由操作系统检测权限。

终于你完成了你的设计，但是由于在这个设计之中**内存管理混乱**，每个地址的分配位置完全取决于运行时的哪有空位置；不同进程间共享一个可见的物理地址空间，**隔离性极差**；并且由于相对寻址的影响，因为不同进程加载相对寻址很难映射到同一个库上，想要**共享一段库也成为了奢谈**，你的设计最终被市场抛弃了。你悲愤之中，穿越回了 ICS 的课堂，准备看一看是什么样的设计打败了你的物理地址。

但这种物理内存的简单设计并不是完全被抛弃了，由于其简单的特性，在一些嵌入式的微处理器中还有所应用，比如老式油车中的车载电脑。但是其他稍微复杂点的计算机系统都已经采用虚拟内存的设计了。

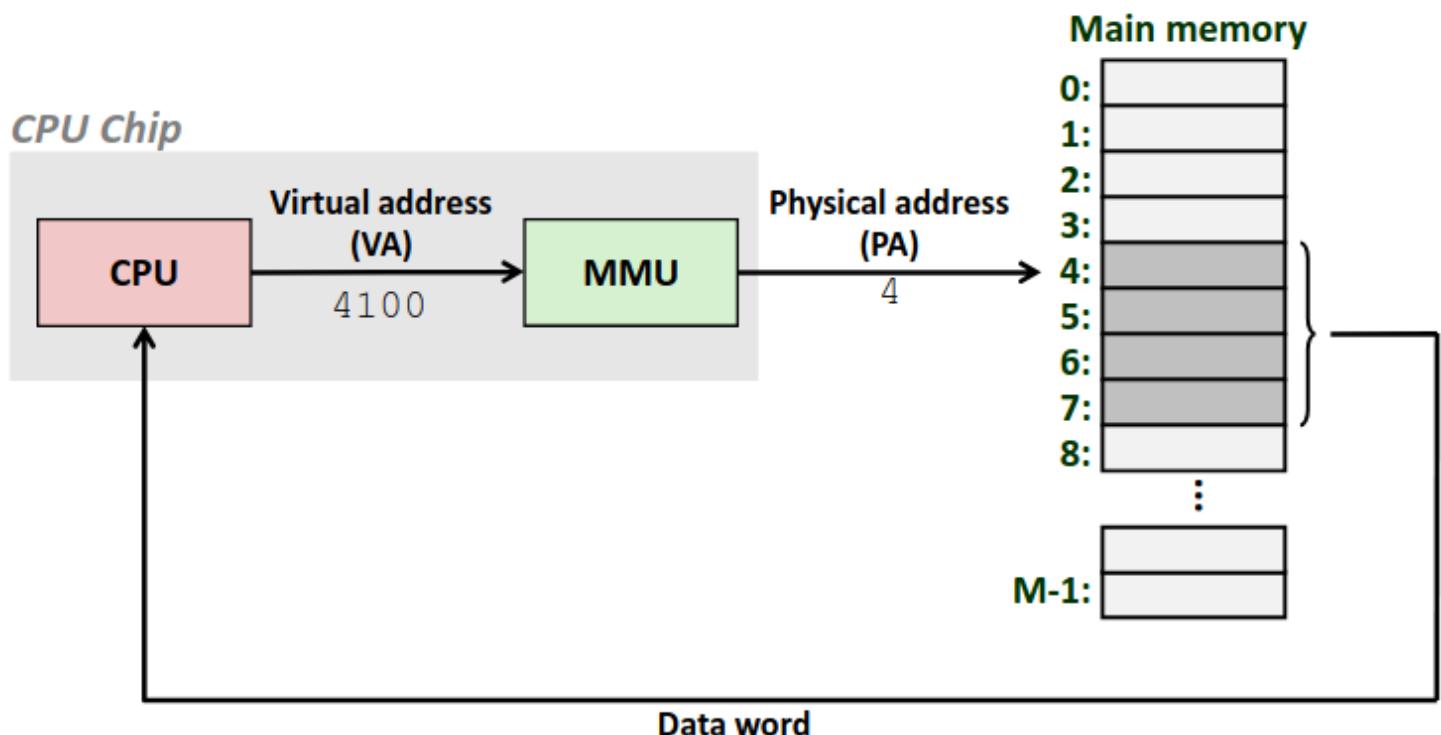
## Virtual Memory

虚拟内存，顾名思义**进程可见的内存空间是虚拟的**，当进程访问一块内存时，由操作系统配合硬件**将虚拟的内存地址翻译为实际的内存地址**。在这个模式下就形成了我们之前一直看到的内存假象，仿佛每一个进程都独有一个完整的地址空间。



其中执行内存翻译的在硬件上有专门的实现，称为**内存管理单元(Memory Management Unit, MMU)**，由于内存访问在程序中再常见不过了，故而 MMU 十分重要，其直接位于 CPU 的那块芯片上。

每当 CPU 执行一条内存访问指令时，其访问的都是**虚拟地址(Virtual Address, VA)**，将对应的地址发送给 MMU，然后 MMU 翻译出**物理地址(Physical Address, PA)**以后，再去访问内存，然后内存将数据发送给 CPU。可以发现在这种模式中，PA 已经对 CPU 不可见了。



## Address Space

为了便于后文讨论，我们引入地址空间的概念，分别为虚拟地址空间与物理地址空间。

虚拟地址空间(Virtual Address Space)用  $n$  位长的二进制数表示，那么地址空间总长度为  $N = 2^n$  对应编号集合 $\{0, 1, 2, 3, \dots, N - 1\}$ 。虚拟地址空间为每个进程独立享有，这也是采用虚拟内存的一大优势。

物理地址空间(Physical Address Space)用  $m$  位长的二进制数表示，那么地址空间总长度为  $M = 2^m$  对应编号集合 $\{0, 1, 2, 3, \dots, M - 1\}$ 。物理内存空间为所有进程共同享有，其对应硬件内存地址。

而所谓的内存翻译就是从虚拟地址空间向物理地址空间做映射。然而根据已有的知识我们知道现在 64 位机下  $N = 2^{64}$ ，显然不可能有那么长的物理内存，所以  $N$  不等于  $M$ ，两个集合的势都不相等，必然不可能建立双射。然而如果单个进程的虚拟地址空间映射到物理地址空间要求单射(每个物理地址显然只能分配给一个虚拟地址)，故在物理地址空间中引入  $\emptyset$ ，对于没有分配的虚拟地址就是映射到了空集上。

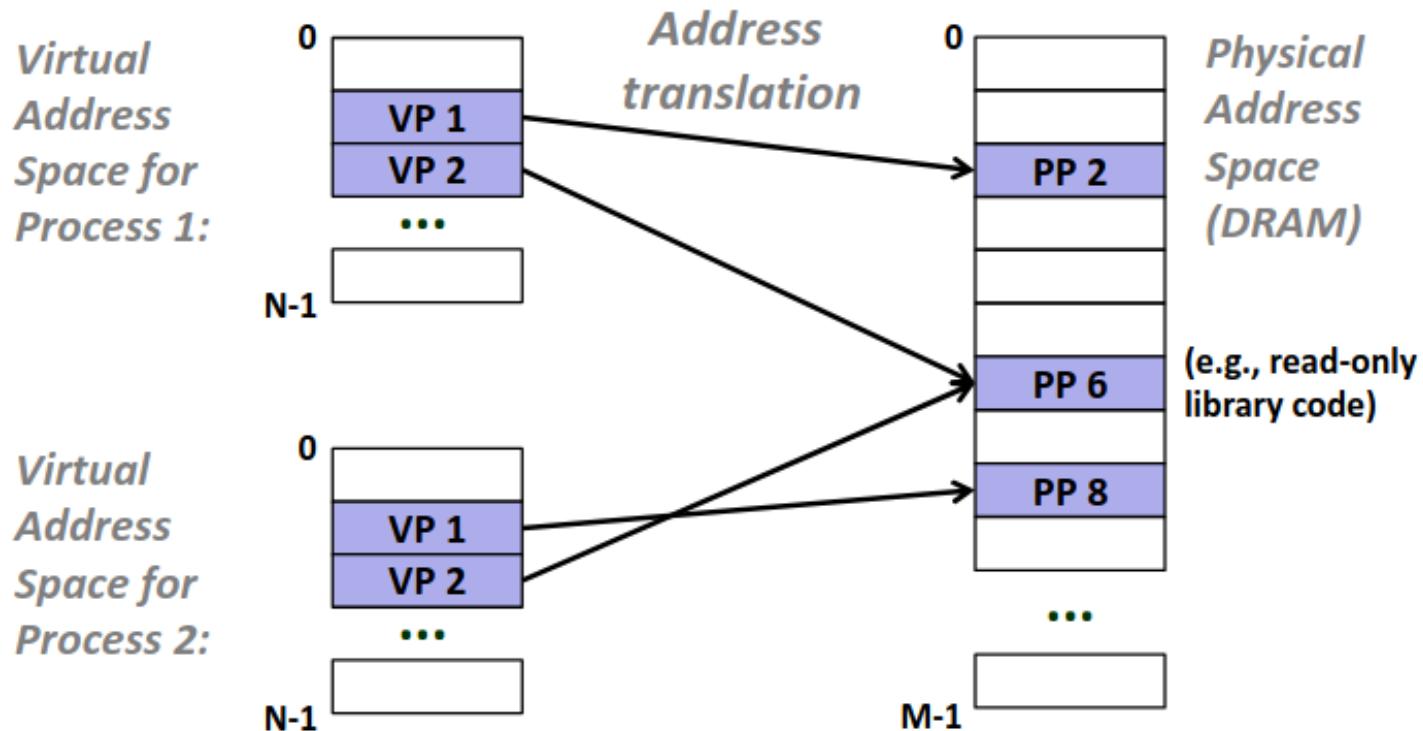
# Why Virtual Memory

前文简要介绍了一下虚拟内存相关的知识，批判了直接用物理地址的方案，接下来会更为详细的介绍 VM 的知识，并引出 VM 的好处。

## VM as a tool for memory management

VM 一个最为直观重要的好处是每个进程有了自己独有的虚拟地址空间。这样程序编写编译时可以直接将内存视为一块线性的数组，而对于 VA 映射为 PA 的映射方式有操作系统与 MMU 把关，十分灵活。

不同进程的同一 VA 既可以映射为同一 PA，也可以不同；不同 VA 也可以映射位同一 PA。



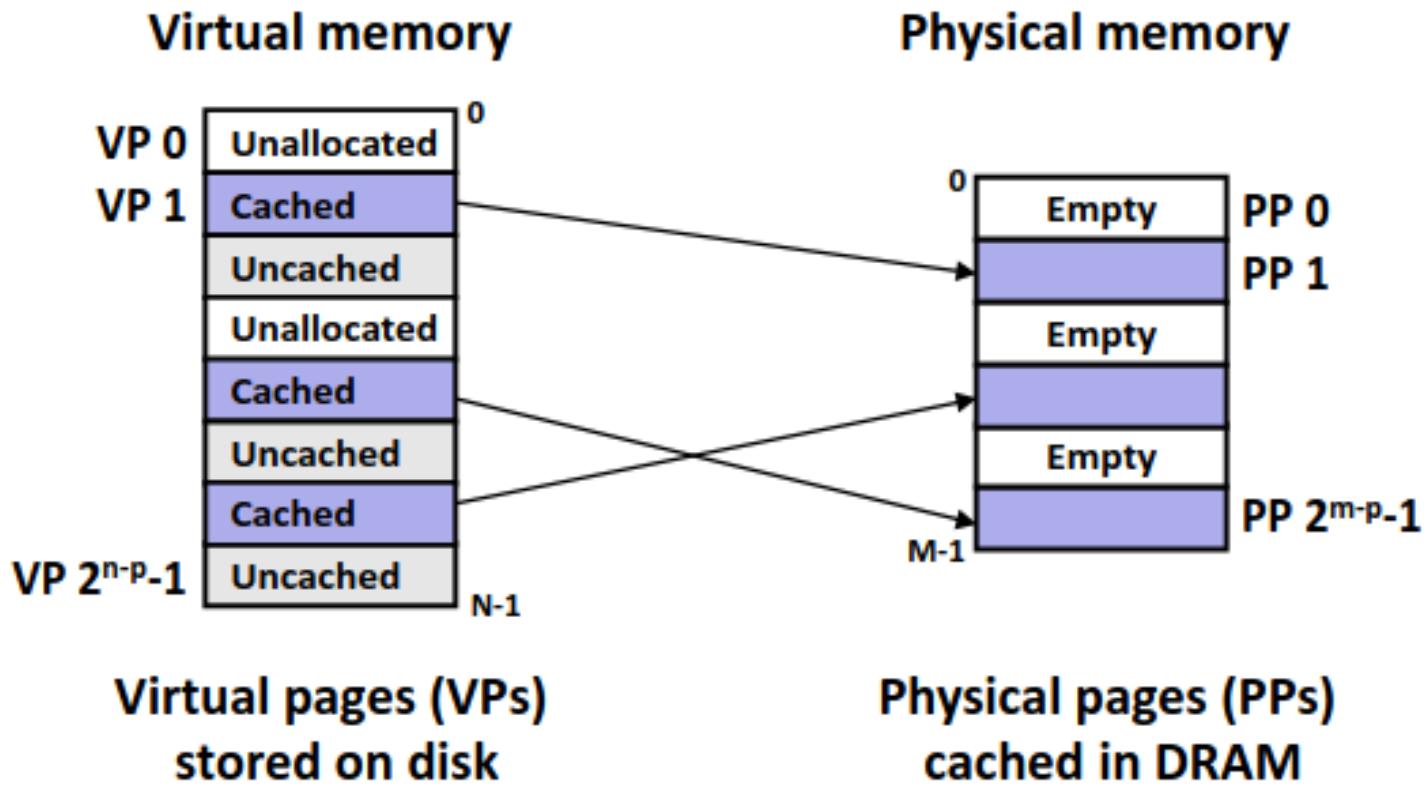
## VM as a tool for caching

我们上文中提到主存是磁盘的一个缓存，那么我们可以认为 VM 就是一个存在在磁盘上的存储空间。虽然没有任何一个磁盘有虚拟内存那么大。

我们以页(Page)，作为管理磁盘的单位，如同我们将内存中数据放入 cache 中一样，我不会每次

用哪个字节放哪个字节，而是把一块内存数据放入缓存。对于磁盘主存我们同样处理，每次我们都将一整个页放入主存当中。

那么 VM 有着跟磁盘的对应关系，将 VM 也按照页的大小划分，每次需要为某个 VA 分配 PA 时，就为对应 VM 的那个页分配一个磁盘上的对应的页。然后需要用某个 VA 时，就会将对应的页拿到主存中。由此我们可以将主存视为 VM 的一个缓存。



既然讨论了缓存就必然需要讨论映射模式，即怎么从磁盘映射到主存呢？对缓存熟悉的同学可能还记的几种映射模式：全相连、组相连...我们在第六章第四节中详细的讨论过不同映射模式的含义以及其优劣，在这里不再重复。

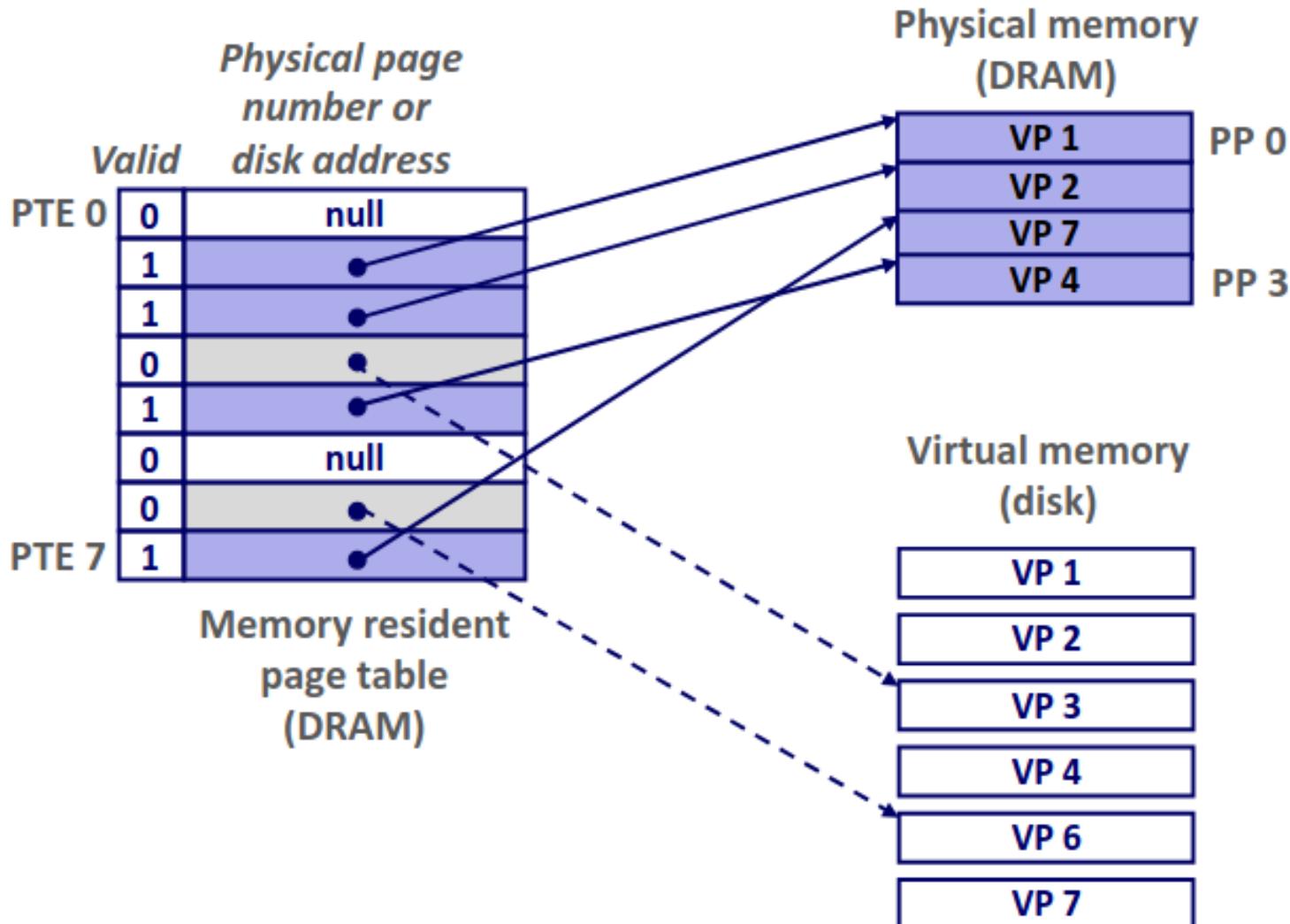
直接说结论的话内存采用的是全相连的模式，全相连的好处是不会有冲突不命中，但会增加硬件实现的成本，降低映射效率。但是考虑到磁盘慢的可怕，miss 的代价实在太高，与之相比在主存的额外花费变得可接受了。

有了这个认知我们即可建立起对虚拟内存的一套相对完整的认知，即对于分配每个 VA 是按页映射到磁盘上，然后当调用到这个地址时，将磁盘中对应的数据载入到内存中。

由于磁盘到内存的映射直接按照 cache 的方式即可，那么只用在关注 VP 向 PP，即虚拟页向磁盘页映射。

我们只需要建立维护一个简单的数据结构：**页表(Page Table)**。页表存储在内存中，由每个进程的内核代码维护，在页表中我们只需要记录每个分配的虚拟页对应的物理页即可。考虑到页通常是比较大的，所以说页表中条目不会过多，也就无需占用过大的内存。值得注意的是页表为**每个进程独有**。

页表可以类比为一个结构体数组，每个元素称为单个**页表实体(Page Table Entries, PTEs)**，对应记录一条 VP 向 PP 的映射。



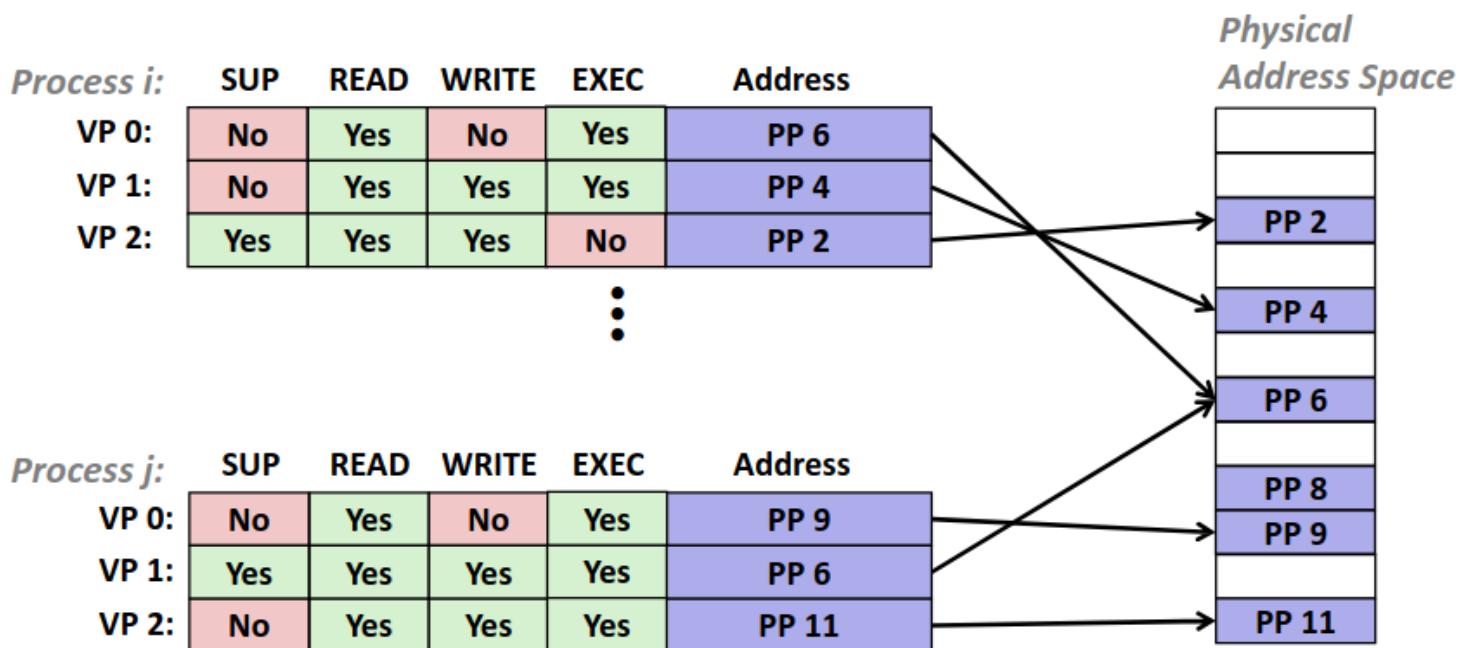
可以看到在图片中，VP 被分配到了磁盘上，而内存则是充当了缓存。那么缓存就必然有命中与未命中，即 hit 与 miss，在内存中我们称为**页命中(Page hit)**与**页缺失(Page Fault)**。

Page Fault 这个概念我们在异常控制流中提到过，其实就是当访问某个你事先要求分配的内存，因为种种原因，可能是因为你分配的太大，可能是太久没用了，操作系统只在磁盘上给你分了一个页，而在内存中没用，当你后续用到这块内存时，操作系统就会去处理这个问题，把对应的页从磁盘拉到内存中，然后再把数据给你。

## VM as a tool for protection

有了页表进行记录 VP 向 PP 的映射关系，也便于我们更好的执行内存保护。

1. 可以更好的知道哪些内存时分配可用的，每次内存访问都需要通过 MMU 把关，而查询对应页表的过程也可以知道哪些内存是分配过可用，哪些内存是程序乱报的数，根本没分配过。
2. 在页表中增加额外的位表示权限，可以更好的管理内存读写权限，简单的权限包括可读可写可执行等等，只需要访问页表时根据权限位判断当前操作是否合法。



本节的内容就到此为止了，在这一节中我们简要介绍了一些关于 VM 的概念与 VM 的基本运作逻辑，下一节我们将更为详细的介绍 VM 的机制，主要是内存翻译以及页表的数据组织方式。

# Chapter 9.2 Address Translation

在上一节中我们讨论了关于虚拟内存的一些基本的内容，这一节我们将进一步深入研究内存翻译这个过程。

## Address Translation

和 cache 中的思想方法一致，我们先对地址进行划分。

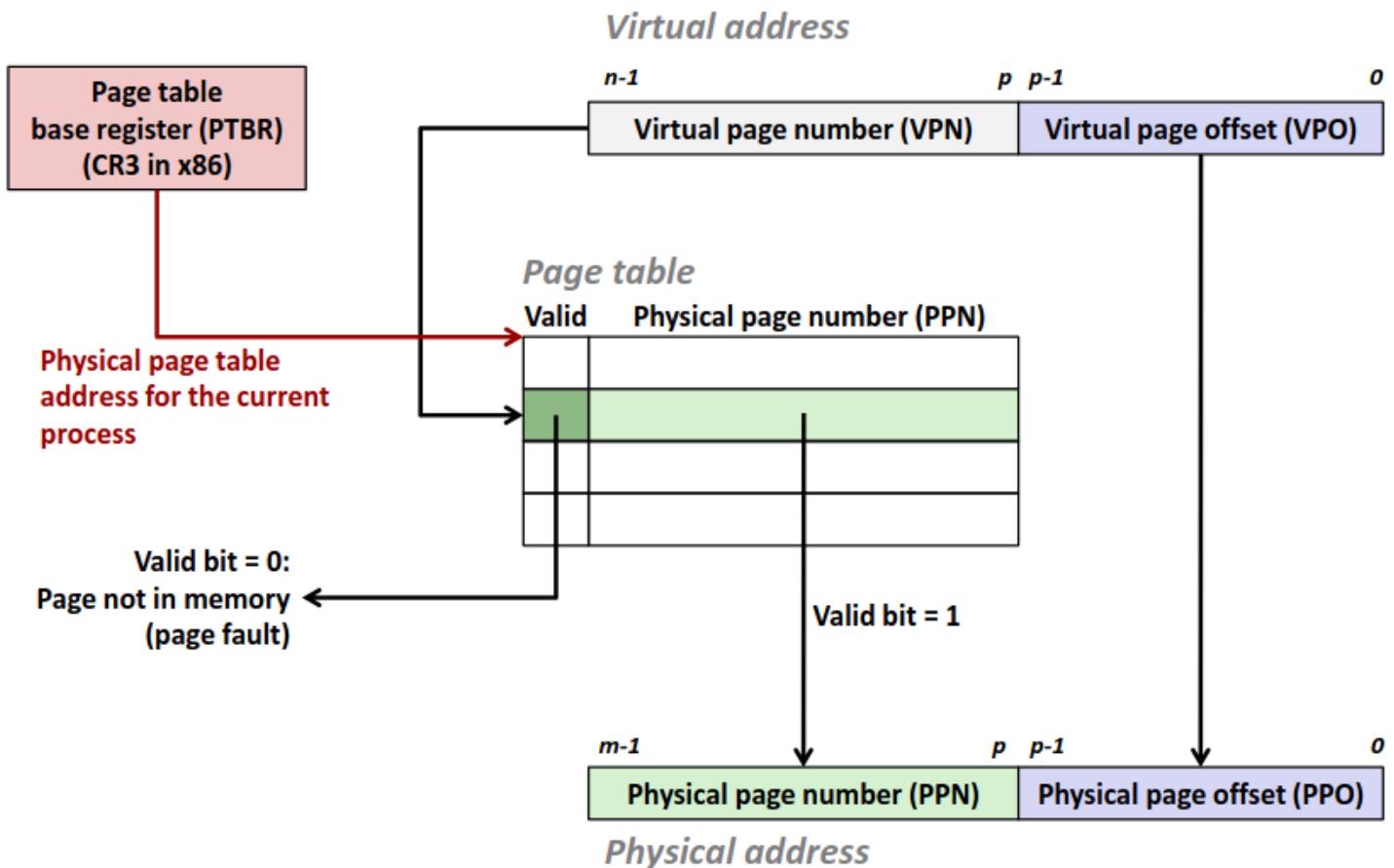
**Virtual address = Virtual page offset + Virtual page number (VA = VPO + VPN).** VPN 不是梯子，是对应的页的编号，而 VPO 则是代表页内的偏移量。同理可以对物理内存进行划分。

**Physical address = Physical page offset + Physical page number (PA = PPO + PPN).** 不难意识到，PPO 的大小应当等于 VPO 的大小，否者会出现无位置分配或者物理内存浪费的情况。

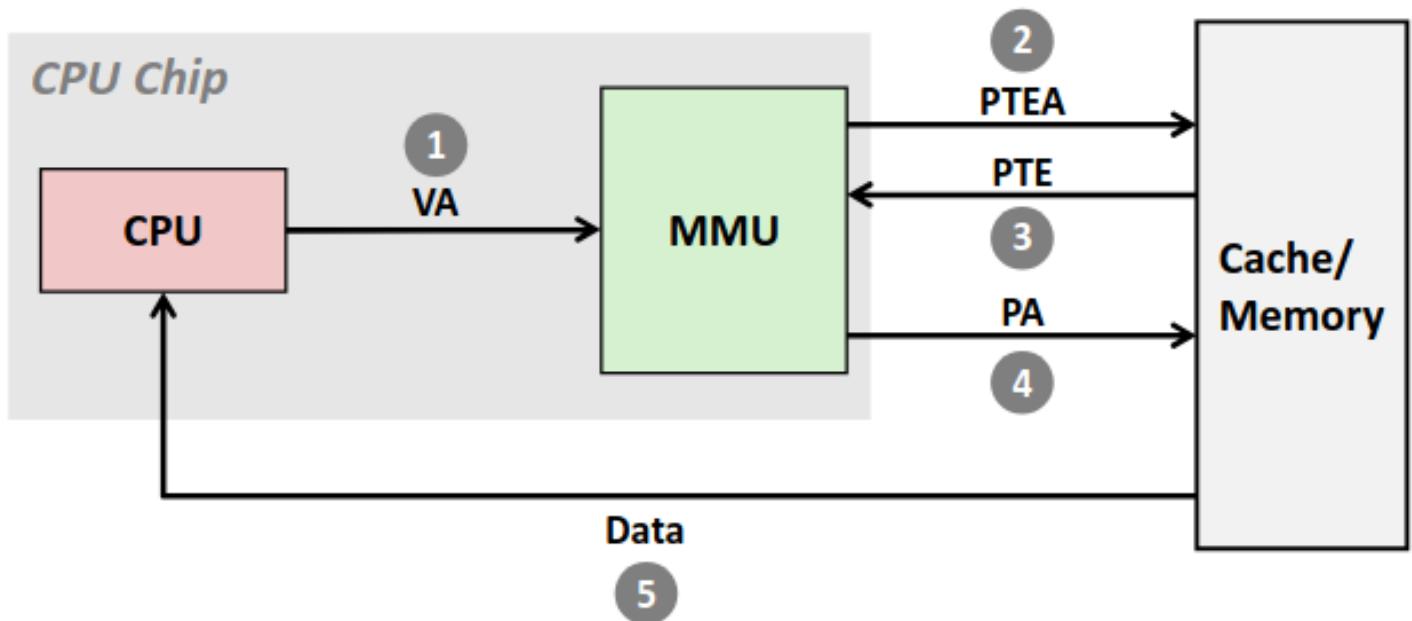
由于我们访问一个地址前需要先访问页表，而页表也是存在内存中的，那么这就矛盾了，我要访问内存，需要通过页表翻译为物理地址，然而页表又再内存中，我要先访问内存。

解决方案很简单，我们在 CPU 上引入一块新的寄存器，称为**页表基址寄存器 (Page Table Base Register, PTBR)**，里面存放当前进程的页表的**物理地址**(如果是虚拟地址又陷入循环了)。

那么每次内存访问，我们先将 VP 拆为 VPO 与 VPN，然后把 VPN 拿到页表中找到对应的 PPN 再把 PA = PPN + VPO，拿到物理地址到内存中找对应的地址即可。



如果对应的页在内存中，就会形成如下图的访问顺序。

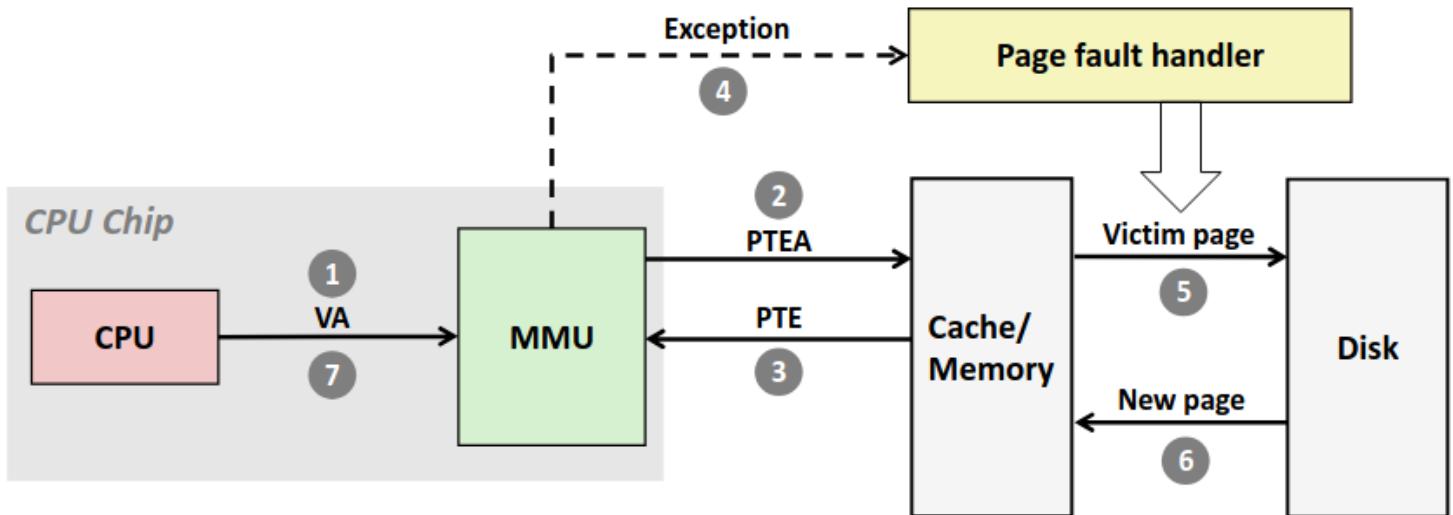


CPU 现将需要访问的 VA 发送给 MMU, MMU 再根据页表的位置向内存中去找页表中 VA 对应的条目, 然后很幸运的对应的 VP 分配在了内存中, 内存返回条目, 然后 MMU 根据这些信息拼凑出

对应实际的 PA，再向内存访问 PA，然后数据返还给 CPU。

不难意识到虽然只有一个内存访问请求但实际上需要两次的内存访问。

而如果对应的页不在内存中，那就灾难了，大致流程如下图。



可以看到我们需要去磁盘中寻找数据，并且还要将被替换页的数据写入磁盘中，代价十分惨烈。

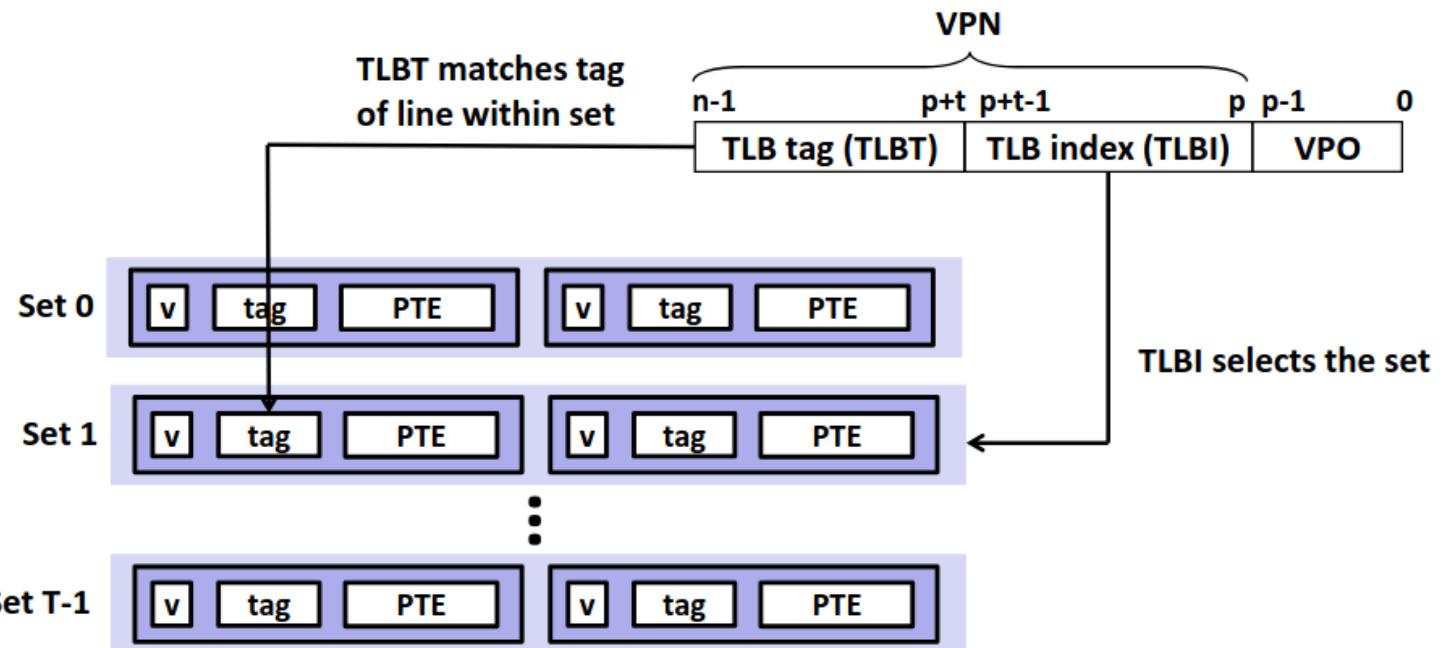
## Translation Lookaside Buffer

可以看到目前为止虚拟内存虽然带来了一些好处，但在效率上却不尽如人意。对于 Page Fault 的情况无法避免，总要访问到在磁盘上而不在内存中的数据的，就算直接采用物理地址也无法避免。糟糕的在与 Page Hit 时我们竟然也需要访问两次内存，而如果采用物理地址只需要一次，而且页表也是在内存中的数据，增加了数据总量，考虑到 cache，代价更大了。

核心在于 PTE 作为重要的数据，混杂在了所有数据当中，占内存抢 cache，如果我们对它区别对待，增加一块缓存专门用于存储页表的项，会极大提高效率！

**转译后备缓冲器 (Translation Lookaside Buffer, TLB)**，也称为页表缓存，一块新的 cache 在 CPU 芯片上，非常小，仅能存储极少的条目(8~16条)。

那么我们在从 VPN 得到 PPN 的过程中，就像访问 cache 一样，将 VPN 划分为索引与标记，先去 TLB 中找对应的条目。



这种方法极大的加快了访问速度，我们一直提到 cache 的访问极快，更别说 TLB 直接在 CPU 芯片上，速度几乎和 L1 cache 等同，相较于后面的内存访问，这个代价几乎可以忽略不计了。

但是 TLB 的条目不是很少吗，miss rate 不会很高吗？如果我们条目过多显然会增加硬件实现的成本，在 CPU 这个寸土寸金的地方容不得我们挥霍。那么为什么很少的条目就足够了呢？核心在于 **页很大**，对于一个进程，往往用到的页的数量是相当有限的，所以极少的条目就可以保证一个极高的 hit rate 了。

## Multi-Level Page Tables

在前文里我们说页表像一个数组一样，里面存储所有虚拟页与实际页的对应，就像一个魔法箱，你给一个 VA，它出一个 PA。但是如果你实际算一笔账的话就会意识到不对。

如果我们考虑需要 O(1) 访取元素，显然页表中的元素应当顺序组织，而不能用链表，必须直接索引，如果接触过桶排序的同学就能意识到，通过桶的想法能实现我们的需求，即开辟一个很大的数组，数组索引对应了 VA 的标号，数组存的值对应 PA。有经验的同学就会意识到对于桶最大的限制来源于空间复杂度。

如果考虑 4K 的页大小，以及 48 位的地址空间的话，如果每个条目 8 字节大。那么对于一个页表我就需要  $2^{48} * 2^{-12} * 2^3 = 2^{39}$  bytes，约 512 GB 的空间！完全不可接受。

然而实际上大部分虚拟页都是不可能分配的，这种稀疏的桶优化为哈希表是常见的思路。利用

Hashing 的思想，我们提出了多级页表(Multi-Level Page Tables)。

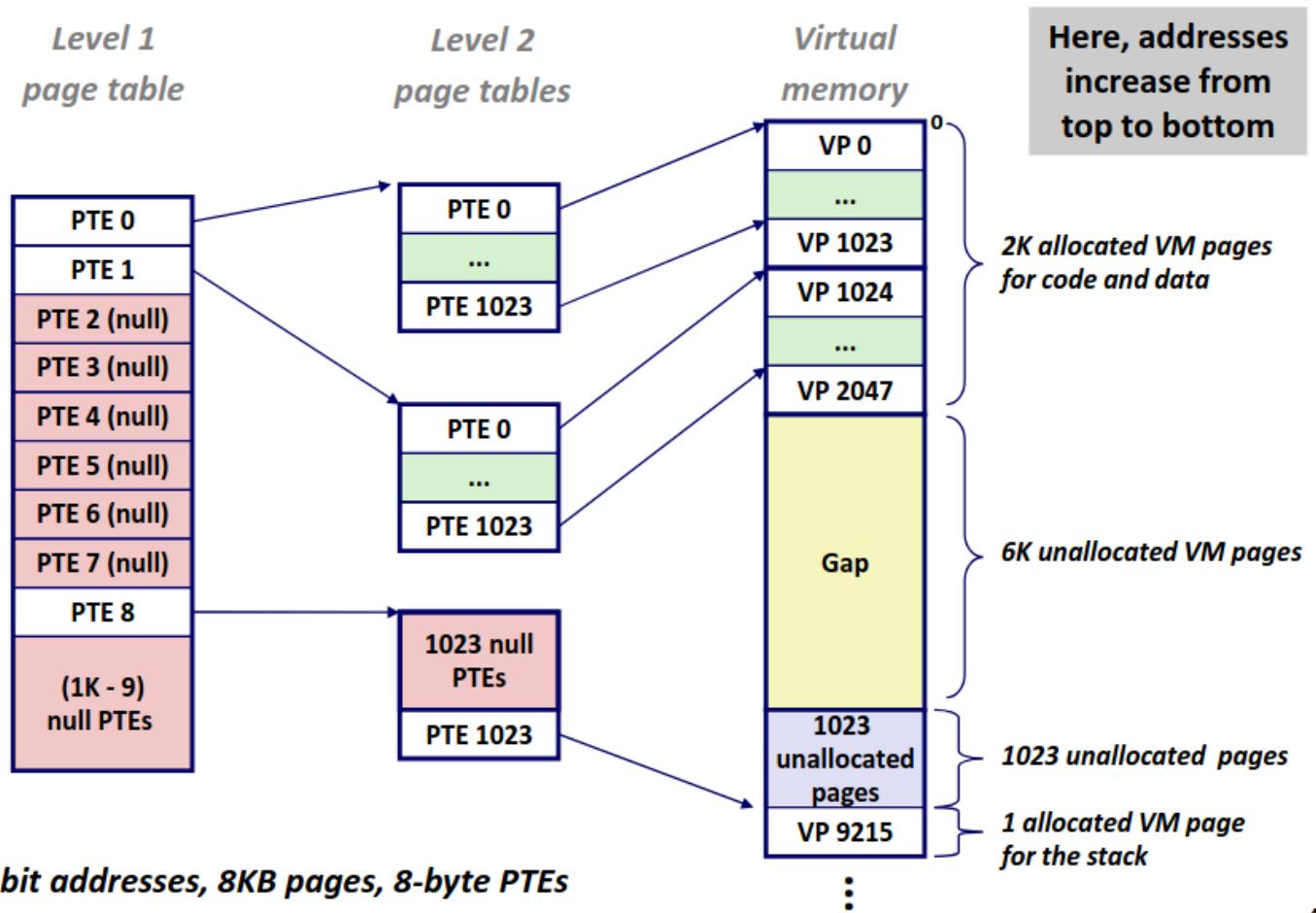
多级页表即采用间接映射，第一级的页表每个条目实际上为一个指针指向第二级页表，第二级页表在指向更低一级的页表。直到最低一级的页表中的条目存放的是 PPN。

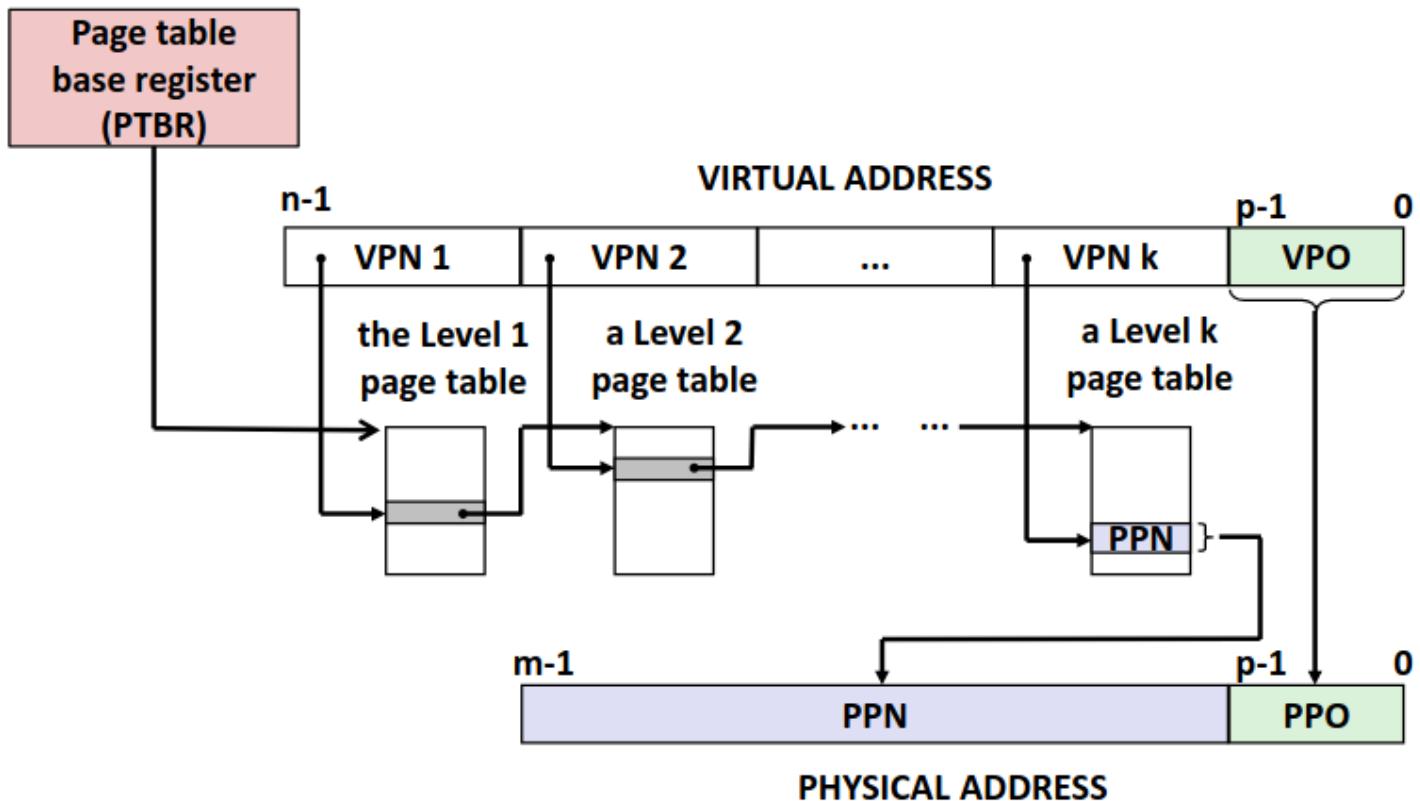
而这种方案节省内存在于我只需要激活高级页表中用到的节点所对应的低级页表，对于没用到的部分空间便被省去了。

前文中提到哈希或许对部分同学有误导性，对于哈希，低级的存储常用链表结构，那我们页表的低级也采用链表结构的话在低级需要常数级的访问。有人说常数级就常数级呗，很快啊。别忘了每次对页表的访问都是一次赤裸裸的内存访问，在这种系统级问题下，完全是灾难级的负优化。

但是哈希我们低层次采用链表的原因是递属哈希节点下的实体数目是不确定的，然而对于最低层次的页表下所拥有的实体数仍是确定的，所以我们仍可在最低层次的页表下直接建立桶，实现 O(1) 的访问。

## A Two-Level Page Table Hierarchy





那么一整套的内存访问逻辑就以及构筑完成了，主要到多级页表的访问仍然需要多次访问内存，显然不是级数越多越好，并且这种访问是缓存不友好的，但是考虑到 TLB 的存在，页表的访问本身极少了，所以这种效率也可以接受了。

那么对于虚拟内存部分的主体内容就大致结束了，下一节我们将会在讲一讲关于内存映射即共享内存相关的内容，再举一个简单的例子。

# Chapter 9.3 Memory Mapping

最后一节我们来讨论关于内存映射部分的内容并且举一个简单的内存系统模型加深理解。

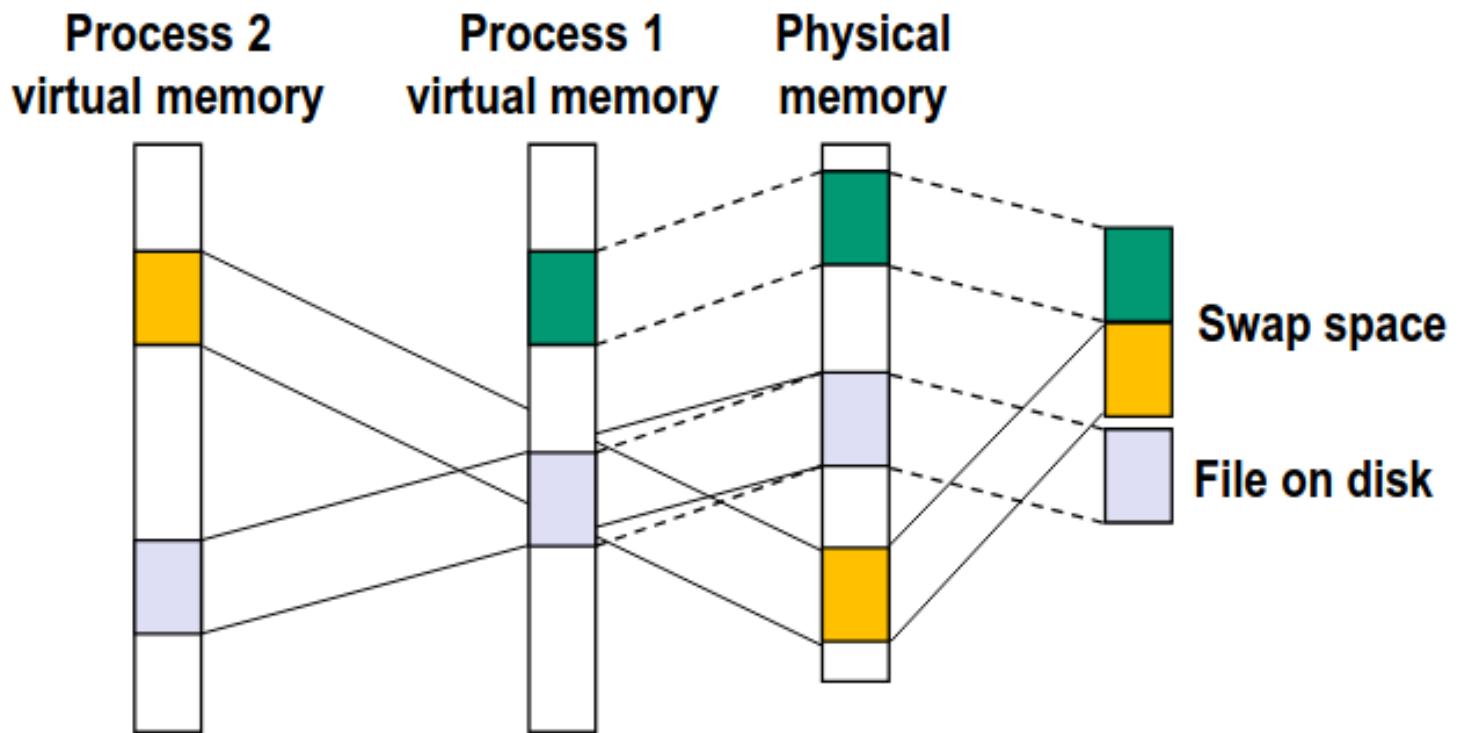
## Memory Mapping

我们先前讨论了虚拟内存的基本知识以及地址翻译的设计。在虚拟内存的优势部分我们曾说过，通过虚拟内存的机制，我们可以很简单的实现共享库这个强大的工具。在介绍如何实现以前我们还需要介绍关于 linux 系统常见关于内存管理的设计。

在先前我们提到对于虚拟内存映射到物理内存上时我们提到应当在磁盘上也为这个虚拟内存分配一块位置，而将主存视为对磁盘的缓存。而在硬盘上分配的空间，或者说主存中放不下的工作集，就可以存放在硬盘上的 **swap 分区** 中。

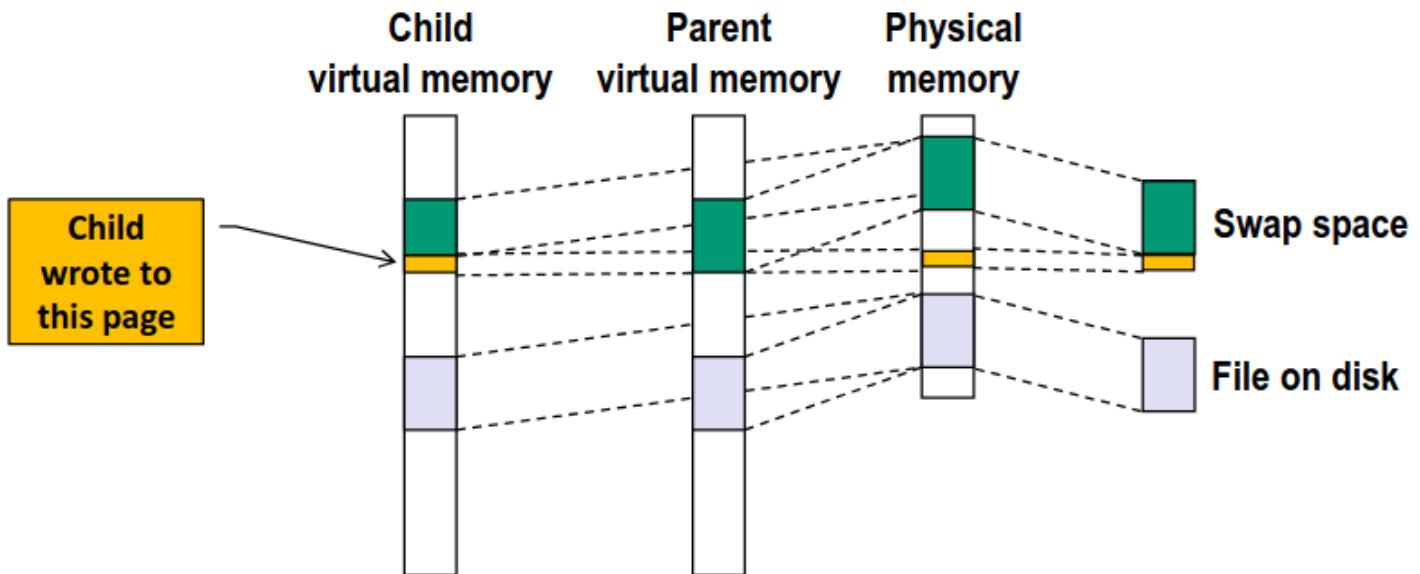
而对于实现共享库的核心在于不同的进程之间共享**文件**，而文件不过是硬盘上一块数据，对于共享库的实现方法就很显然了，我只需要磁盘上的文件放在主存中，然后不同的进程的不同或者同一 VA 一同映射到同一 PA 上即实现了文件共享。

而我们如果把 swap 分区也视为文件，那么这套机制设计统一、逻辑自洽，也完美符合了 Linux 的设计理念：一切皆文件。



同理在先前在讲进程的时候我们提到对于 fork，产生一个与父进程完全相同的子进程时，我们会采用 "**Copy-on-write**" 的策略，这种策略可以有效提高我们 fork 的效率，那么在虚拟内存的下 Copy-on-write 的实现也非常简单了，子进程产生的时候直接复制父进程的页表，这样子进程的所有 VA 就映射到父进程对应的 PA 上，两者享有了完全一样的内存。

这时候我就从 OS 层面将对应的内存权限缩紧，父进程或者子进程试图修改对应位置的值时，再触发复制机制即可。

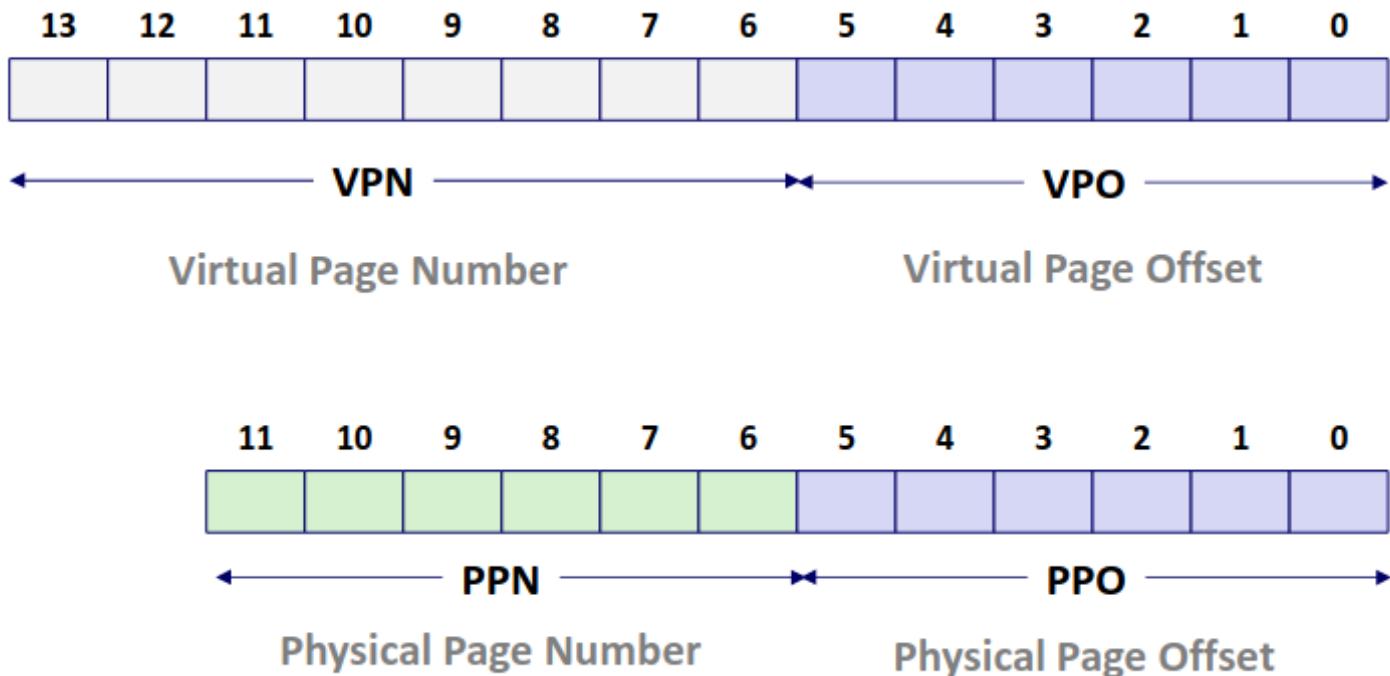


当然这个设计只是一个现有的设计而不是一个普适性的设计，比如在 Windows 系统中就限制了 swap 分区的使用甚至抛弃这种设计。利弊讨论超越了本课程的范围，感兴趣的同學可以查阅资料了解。

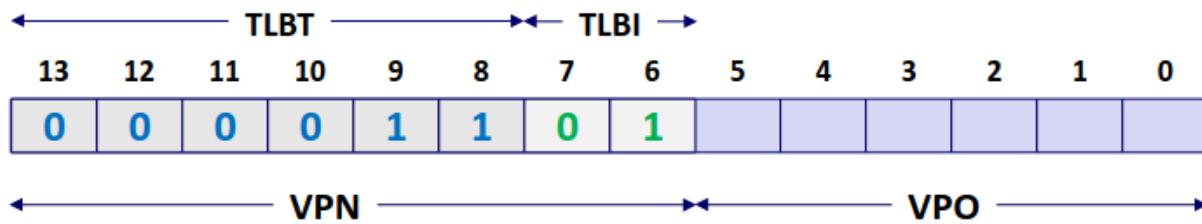
## Simple Memory System

我们通过一个例子算一算参数来加深对内存系统的理解~~(从考试角度这部分可能反而更重要因为方便出计算题)~~

对于一组简单数据物理内存大小为 4K，即需要 12 位来表示，而虚拟内存大小为 16K，即需要 14 位来表示，而页的大小为 64 bytes，那么自然需要 6 位  $2^6 = 64$  表示在页内偏移。那么 VA 与 PA 可以如下图划分。



现在来看我们的 TLB，TLB 可以存 16 个实体，采用四路组相连。那么显然  $2^2 = 4$  TLBI 需要两位，剩下的留作 TLBT。



$$\text{VPN} = \text{0b1101} = \text{0xD}$$

### Translation Lookaside Buffer (TLB)

| <i>Set</i> | <i>Tag</i> | <i>PPN</i> | <i>Valid</i> |
|------------|------------|------------|--------------|------------|------------|--------------|------------|------------|--------------|------------|------------|--------------|
| 0          | 03         | -          | 0            | 09         | 0D         | 1            | 00         | -          | 0            | 07         | 02         | 1            |
| 1          | 03         | 2D         | 1            | 02         | -          | 0            | 04         | -          | 0            | 0A         | -          | 0            |
| 2          | 02         | -          | 0            | 08         | -          | 0            | 06         | -          | 0            | 03         | -          | 0            |
| 3          | 07         | -          | 0            | 03         | 0D         | 1            | 0A         | 34         | 1            | 02         | -          | 0            |

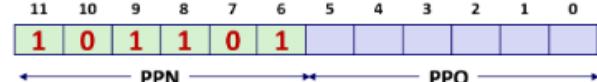
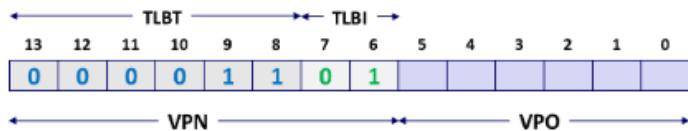
可以看到图中对于一个 VPN 我们将其进行划分，至于为什么 Set index 采用中间的位在 cache 中已经讨论过了。

来到页表这一层次，相对就简单很多了直接根据 VPN 的索引找到 PPN，和 VPO 拼在一起再访问物理内存就可以了。

| <i>VPN</i> | <i>PPN</i> | <i>Valid</i> |
|------------|------------|--------------|
| 00         | 28         | 1            |
| 01         | -          | 0            |
| 02         | 33         | 1            |
| 03         | 02         | 1            |
| 04         | -          | 0            |
| 05         | 16         | 1            |
| 06         | -          | 0            |
| 07         | -          | 0            |

| <i>VPN</i> | <i>PPN</i> | <i>Valid</i> |
|------------|------------|--------------|
| 08         | 13         | 1            |
| 09         | 17         | 1            |
| 0A         | 09         | 1            |
| 0B         | -          | 0            |
| 0C         | -          | 0            |
| 0D         | 2D         | 1            |
| 0E         | 11         | 1            |
| 0F         | 0D         | 1            |

$$0x0D \rightarrow 0x2D$$

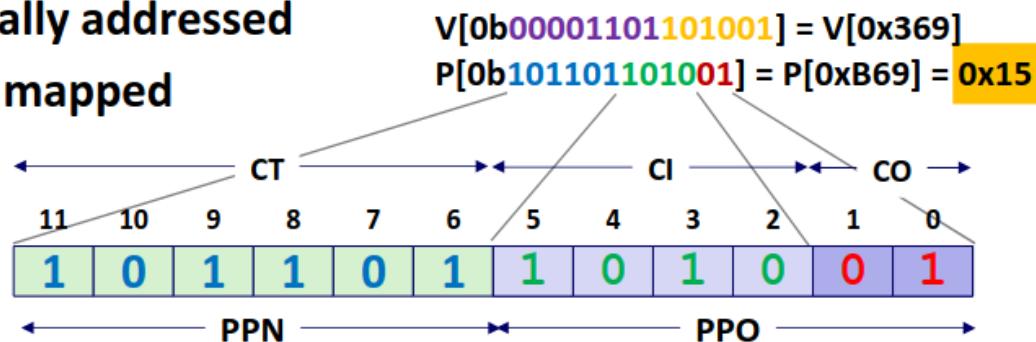


在得到 PA 后，再访问内存，如果考虑的更为复杂，这中间还有 cache 的访问，我们这里就直接考虑访问内存了。内存采用全相连的模式，顺序比较 tag 即可，我们就得到了需要的数据。

## ■ 16 lines, 4-byte cache line size

## ■ Physically addressed

## ■ Direct mapped



| <i>Idx</i> | <i>Tag</i> | <i>Valid</i> | <i>B0</i> | <i>B1</i> | <i>B2</i> | <i>B3</i> |
|------------|------------|--------------|-----------|-----------|-----------|-----------|
| 0          | 19         | 1            | 99        | 11        | 23        | 11        |
| 1          | 15         | 0            | -         | -         | -         | -         |
| 2          | 1B         | 1            | 00        | 02        | 04        | 08        |
| 3          | 36         | 0            | -         | -         | -         | -         |
| 4          | 32         | 1            | 43        | 6D        | 8F        | 09        |
| 5          | 0D         | 1            | 36        | 72        | F0        | 1D        |
| 6          | 31         | 0            | -         | -         | -         | -         |
| 7          | 16         | 1            | 11        | C2        | DF        | 03        |

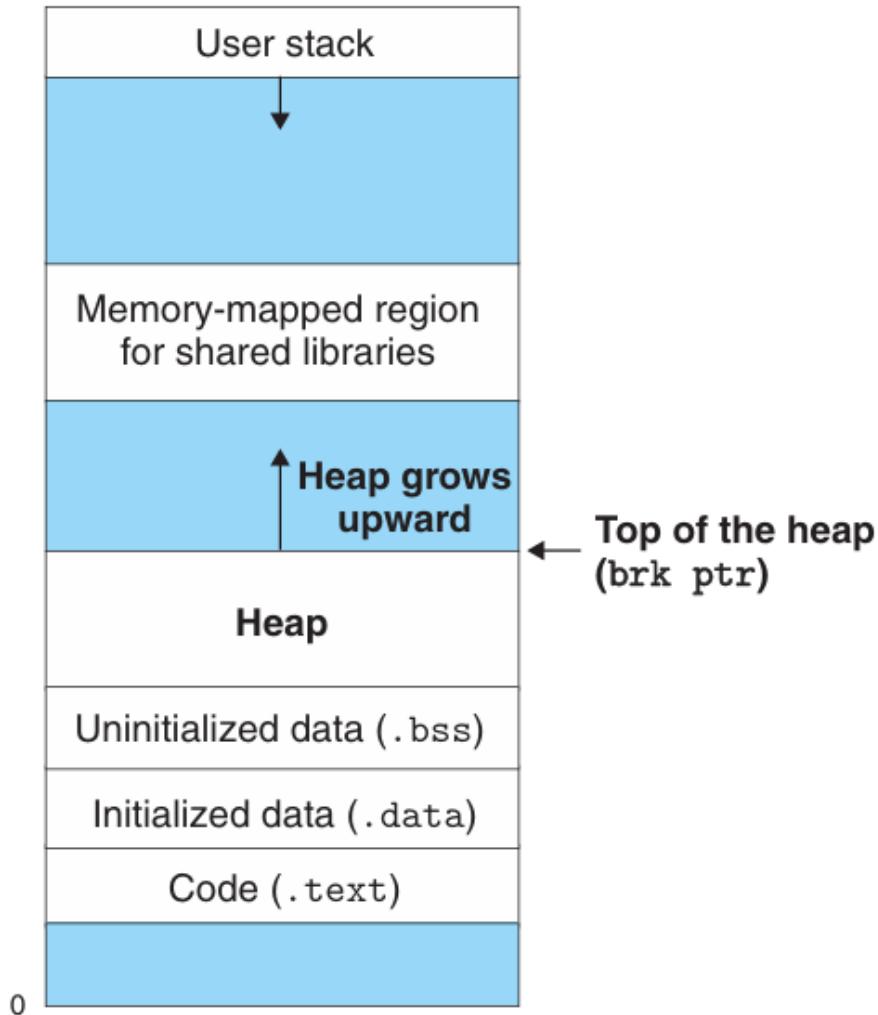
| <i>Idx</i> | <i>Tag</i> | <i>Valid</i> | <i>B0</i> | <i>B1</i> | <i>B2</i> | <i>B3</i> |
|------------|------------|--------------|-----------|-----------|-----------|-----------|
| 8          | 24         | 1            | 3A        | 00        | 51        | 89        |
| 9          | 2D         | 0            | -         | -         | -         | -         |
| A          | 2D         | 1            | 93        | 15        | DA        | 3B        |
| B          | 0B         | 0            | -         | -         | -         | -         |
| C          | 12         | 0            | -         | -         | -         | -         |
| D          | 16         | 1            | 04        | 96        | 34        | 15        |
| E          | 13         | 1            | 83        | 77        | 1B        | D3        |
| F          | 14         | 0            | -         | -         | -         | -         |

我们关于虚拟内存基本知识的介绍就到此为止了，接下来几节我们将会在虚拟内存的基础上介绍动态内存分配的基本知识，希望对你有帮助。

# Chapter 9.4 Dynamic Memory Allocation: Basic Concepts

当运行时需要额外的虚拟内存时，动态内存分配器就要派上用场了。较之低级的mmap和munmap，动态内存分配器更方便，也有更好的移植性。

动态内存分配器（dynamic memory allocator）维护着一个进程的虚拟内存区域，称为**堆**（heap）。下图展示了它的具体结构。假设堆是一个请求二进制零的区域，它会紧接在未初始化的数据（Uninitialized Data）区域后面开始，并向上生长（即向高地址生长）。对于每个进程，内核维护着一个变量brk，它指向堆的顶部。



分配器将堆视为一组不同大小的块（block）的集合来维护。每个块就是一个连续的虚拟内存片（chunk），其状态要么是已分配（allocated）的，要么是空闲（free）的。

分配器有以下两种基本风格，这两种风格都要求应用显式地分配块，它们的不同之处在于由哪个实体来负责释放已分配的块。

- 显式分配器 (explicit allocator) : 要求应用显式地释放任何已分配的块。e.g. C标准库中的 malloc和free函数。
- 隐式分配器 (implicit allocator) : 要求分配器当检测到一个已分配块不再被程序使用时，就释放这个块。e.g. 高级语言 (如Lisp, Java) 中的垃圾收集 (garbage collection) 。

本课程将围绕显式分配器进行讨论，对隐式分配器感兴趣的同学可参阅CSAPP第9.10节的内容或自行查阅相关资料。

## The malloc and free Functions

C标准库提供了一个称为malloc程序包的显示分配器，程序通过调用malloc函数来从堆中分配块。

```
#include <stdlib.h>
void *malloc(size_t size);
```

malloc函数分配成功会返回一个指针，指向大小至少为size字节的内存块。这个块会为可能包含在块内的任何数据对象类型做对齐。在32位模式中，malloc返回的块地址总是8的倍数，而在64位模式中总是16的倍数。

如果malloc分配失败，它就会返回NULL，并设置errno。

像malloc这样的动态内存分配器可以通过使用mmap和munmap函数显式地分配和释放堆内存，也可以使用sbrk函数：

```
#include <unistd.h>
void *sbrk(intptr_t incr);
```

sbrk函数通过将内核的brk指针增加incr来扩展和收缩堆。如果成功，它就会返回brk的旧值，否则返回-1，并将errno设置为ENOMEM。

其他动态分配函数：

- calloc函数：是一个基于malloc函数的瘦包装函数，可以将分配的内存初始化为零。

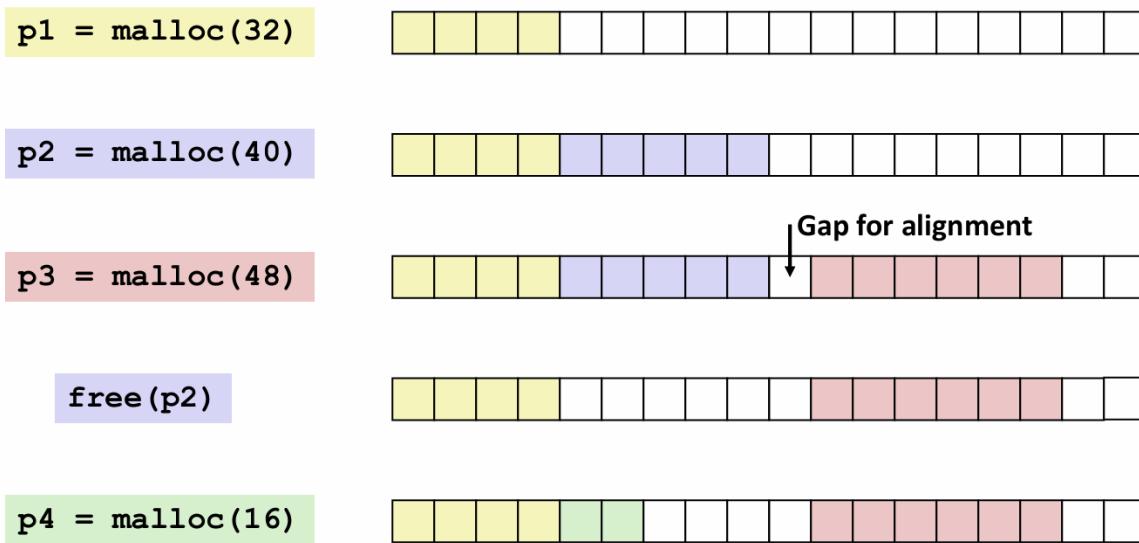
- realloc函数：可以改变一个以前已分配的块的大小。

程序通过调用free函数来释放已分配的堆块：

```
#include <stdlib.h>
void free(void *ptr);
```

free函数的ptr参数必须指向一个从malloc、calloc或者realloc获得的已分配块的起始位置。如果不是，那么free的行为就是未定义的，而且由于它不产生返回，所以就不会告诉应用出现了错误，这会导致出现一些令人迷惑的运行时错误。

下图概念性地模拟了malloc和free的实现，图中每个方框对应一个字（8字节），每个粗线标出的矩形对应一个块。阴影部分是已分配的块，无阴影部分是空闲块。堆地址是从左往右增加的。



几个需要注意的地方：

- 在申请 `p2=malloc(40)` 时，程序请求一个5字的块，malloc实际分配了6字的块，是为了保持双字边界对齐。
- 在 `free(p2)` 时，调用free返回后，指针p2仍然指向被释放了的块。所以在它被一个新的malloc调用重新初始化之前，不能再使用p2。

## Allocator Requirements and Goals

显示分配器必须在一些非常严格的约束条件下工作：

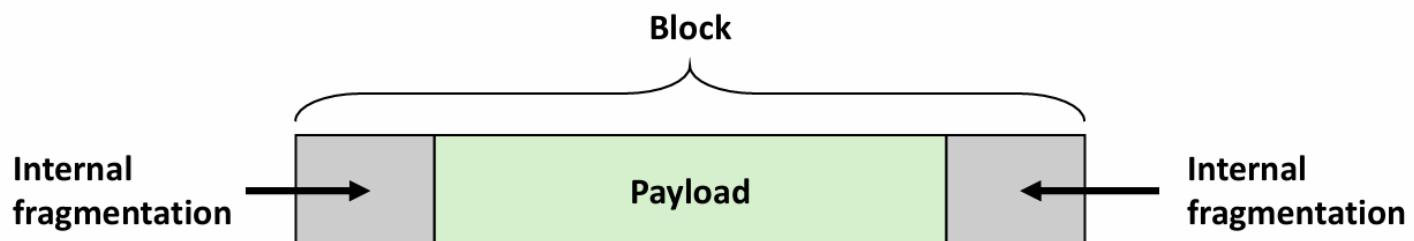
- 处理任意请求序列。一个应用可以有任意的满足约束条件的分配请求和释放请求序列。（即每个释放请求对应一个当前已分配块，这个块是由一个以前的分配请求获得的。）因此，分配器不可以假设分配和释放请求的顺序。
- 立即响应请求。不允许分配器为了提高性能重新排列或缓冲请求。
- 只使用堆。
- 对齐块（满足对齐要求）。
- 不修改已分配的块。一旦块被分配，就不允许修改或移动它了。

在这些限制条件下，分配器的编写者的目标仍然是实现吞吐率（throughput）最大化和内存使用率（memory utilization）最大化，然而性能和开销永远势不两立，这也意味着这两者是相互矛盾的。所以，分配器设计所面临的一个挑战就是在两个目标之间找到一个适当的平衡点。

## Fragmentation

造成堆利用率很低的主要原因是一种称为碎片（fragmentation）的现象。它发生在虽然有未使用的内存但不能满足分配请求时。有两种形式的碎片：内部碎片（internal fragmentation）和外部碎片（external fragmentation）。

内部碎片是在一个已分配块比有效载荷大时发生的，如下图所示：



产生内部碎片的原因有很多，比如维护堆数据结构产生的额外开销，或者为了满足对齐约束条件而增加块的大小。

内部碎片的量化非常简单，它就是所有已分配块的大小和它们的有效载荷大小之差的总和。所以内部碎片的数量只取决于以前请求的模式和分配器的实现方式。

外部碎片是当空闲内存合并起来能够满足一个分配请求，但是没有一个单独的空闲块足够大到能处理这个请求时发生的。例如在前面malloc和free示例图中，如果在 `free(p2)` 之后再申请8字的块，就无法满足要求。问题的产生在于，虽然堆中仍有8个空闲的字，但是它们分在两个空闲块

中。

外部碎片的量化比内部碎片难得多，因为它不仅取决于以前请求的模式和分配器的实现方式，还取决于将来请求的模式，然而我们不可能知道将来请求的模式。所以分配器通常使用启发式策略来试图维持少量的大空闲块，而不是大量的小空闲块。

## Implementation Issues

一个实际的分配器要在吞吐率和利用率之间把握好平衡，就必须考虑以下几个问题：

- 空闲块组织 (free block organization)：如何记录空闲块？
- 放置 (placement)：如何选择一个合适的空闲块来放置一个新分配的块？
- 分割 (splitting)：在将一个新分配的块放置到某个空闲块之后，如何处理这个空闲块中的剩余部分？
- 合并 (coalescing)：如何处理一个刚刚被释放的块？

接下来我们将详细地讨论这些问题，并介绍几种不同的空闲块组织结构。

## Implicit Free Lists

任何实际的分配器都需要一些数据结构，允许它来区分块边界，以及区别已分配块和空闲块。大多数分配器将这些信息嵌入块本身。一个简单的方法如下图所示：



**a = 1: Allocated block**  
**a = 0: Free block**

**Size: total block size**

**Payload: application data  
(allocated blocks only)**

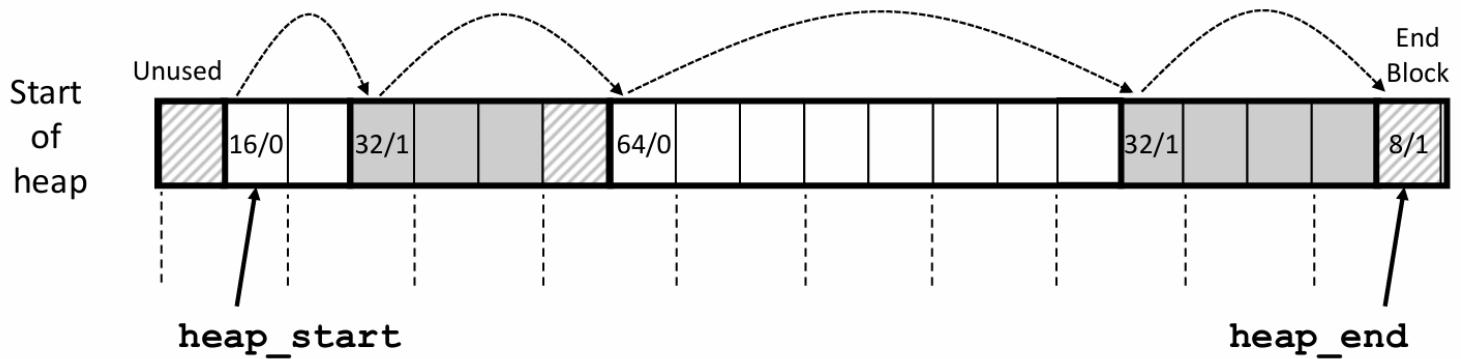
在这种情况下，一个块由头部（Header，大小为一个字）、有效载荷（Payload），以及可能的一些额外填充（Padding）组成。头部编码了这个块的完整大小和分配状态（即标志这个块是已分配的还是空闲的）。

---

注：因为有双字的对齐约束条件存在，块的大小总是16的倍数，即块大小的最低四位总是0。因此我们只需前面的高位来存储块大小，用最低四位来指明这个块的分配状态。这样就使得原来需要2字空间存储的头部节省为只需1字。

---

假设块的结构如上图所示，我们可以将堆组织为一个连续的已分配块和空闲块序列，如下图所示：



Double-word aligned

**Allocated blocks:** shaded  
**Free blocks:** unshaded  
**Headers:** labeled with “size in bytes/allocated bit”  
 Headers are at non-aligned positions  
 → Payloads are aligned

我们称这种结构为**隐式空闲链表** (implicit free lists)，是因为空闲块通过头部中的大小字段隐含地连接着。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。

在上图中不难发现，所有块的头部都处在不满足双字对齐约束条件的位置，这恰恰是为了使每个块的有效载荷满足双字对齐。

同时，在链表结尾还设置了一个已分配的终止头部 (terminating header)，我们将在实现一个动态内存分配器时看到它实际上简化了空闲块的合并。

隐式空闲链表的优点是简单，相对应地，其显著的缺点就是开销过大，对空闲链表进行搜索所需要的时间与堆中已分配块和空闲块的总数呈线性关系。

## Placing Allocated Blocks

当一个应用请求一个k字节的块时，分配器搜索空闲链表，查找一个足够大可以放置所请求块的空闲块。分配器执行这种搜索的方式是由**放置策略** (placement policy) 确定的。一些常见的策略是**首次适配** (first fit)、**下一次适配** (next fit) 和**最佳适配** (best fit)。

- 首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块。其优点是它趋向于将大的空闲块保留在链表的后面。缺点是它趋向于在靠近链表起始处留下小空闲块的“碎片”，这就增

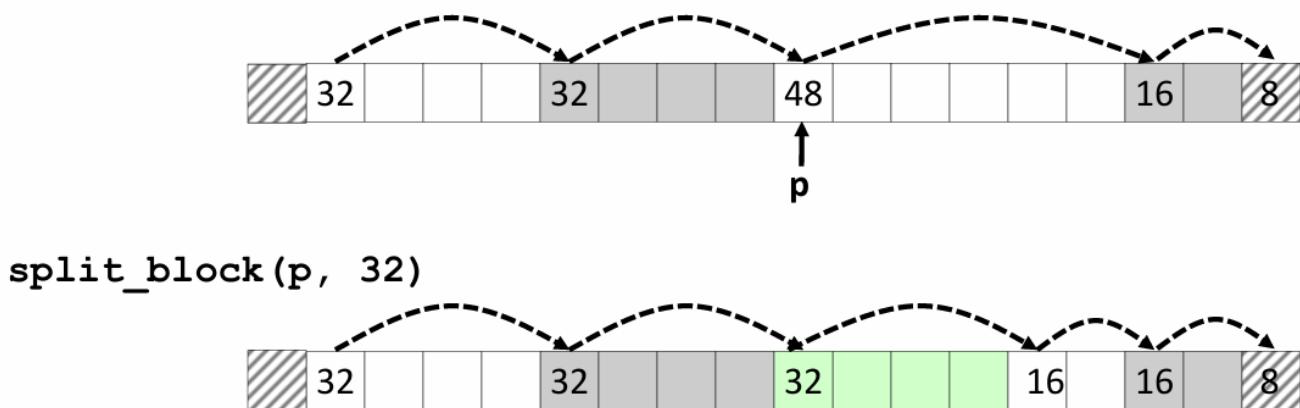
加了对较大块的搜索时间。

- 下一次适配：由大名鼎鼎的Donald Knuth提出。（源于这样一个想法：如果上一次在某个空闲块中发现了一个匹配，那么很可能下一次也能在这个剩余块中发现匹配。）它和首次适配很相似，区别在于它是从上一次查询结束的地方开始搜索。下一次适配比首次适配运行起来明显要快一些，尤其是当链表的前面布满了许多小碎片时。但是下一次适配的内存利用率比首次适配低得多。
- 最佳适配：检查每个空闲块，选择适合所需请求大小的最小空闲块。最佳适配比首次适配和下一次适配的内存利用率都要高一些。然而，在简单空闲链表组织结构中，比如隐式空闲链表，使用最佳适配的缺点是它要求对堆进行彻底的搜索。

## Splitting Free Blocks

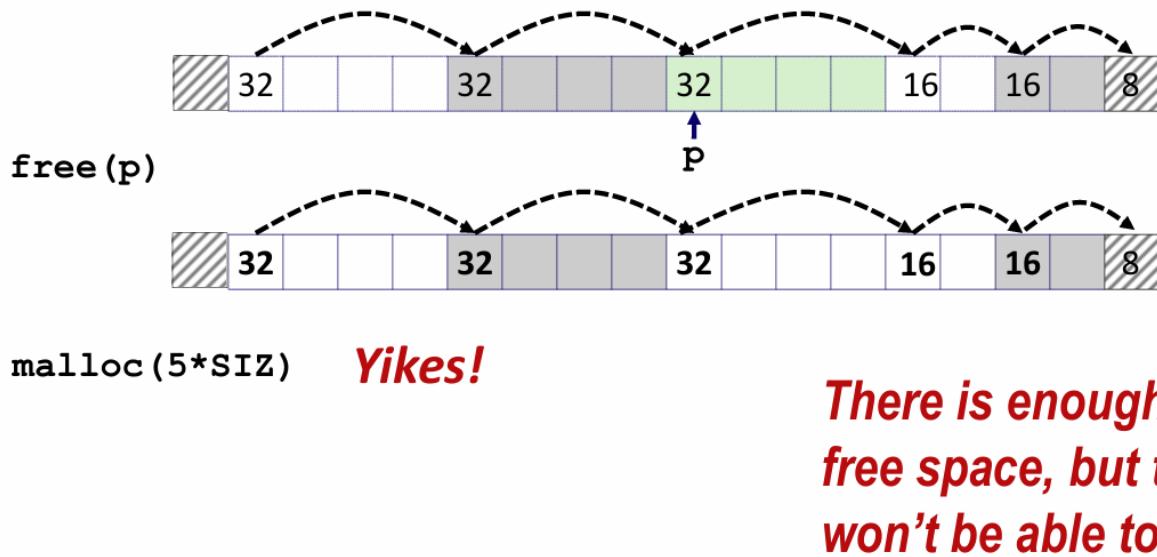
一旦分配器找到一个匹配的空闲块，它就必须做另一个策略决定，那就是要分配这个空闲块中的多少空间。一个选择是用整个空闲块，虽然这种方法简单快捷，但是主要的缺点就是它会产生内部碎片。如果放置策略趋向于产生好的匹配，那么额外的内部碎片还可以接受。然而，如果匹配不太好，那么分配器通常会选择将这个空闲块**分割**（split）为两部分。第一部分变成分配块，剩下的变成一个新的空闲块。

下图展示了分配器如何分割一个48字节的空闲块，来满足一个应用对堆内存32字节的请求。



## Coalescing Free Blocks

当分配器释放应该已分配块时，可能有其他空闲块与这个新释放的空闲块相邻。这些邻接的空闲块可能引起一种现象，叫做假碎片（fault fragmentation），就是有许多可用的空闲块被切割成小的、无法使用的空闲块。比如，下图展示了释放一个分配块后产生的问题，结果是两个相邻的空闲块，大小分别为32字节和16字节，那么接下来一个对5字（需分配6字即48字节）有效载荷的请求就会失败，尽管两个空闲块的合计大小其实是足够大的。



为了解决假碎片问题，任何实际的分配器都必须合并相邻的空闲块，该过程称为**合并**（coalescing）。这就出现了一个重要的策略决定，即何时执行合并。分配器有两种选择：

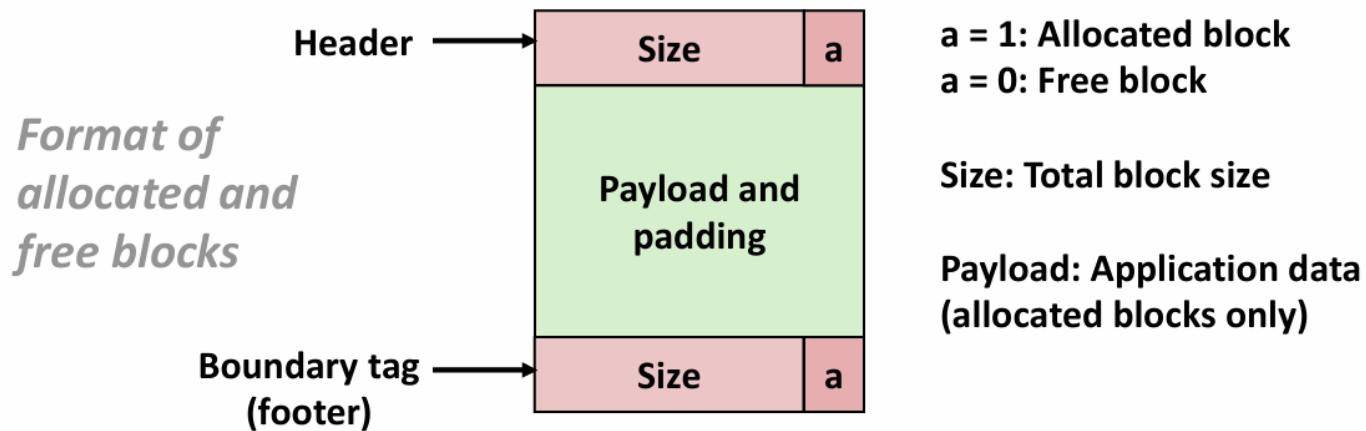
- 立即合并（immediate coalescing）：在每次一个块被释放时，就合并所有的相邻块。
- 推迟合并（deferred coalescing）：等到某个稍晚的时候再合并空闲块。比如等到某个分配请求失败时再扫描整个堆，合并所有空闲块。

## Coalescing with Boundary Tags

我们称想要释放的块为**当前块**。合并下一个空闲块显然是简单且高效的：当前块的头部指向下一个块的头部，可以检查这个指针来判断下一个块是不是空闲的。如果是，就将它的大小简单地加到当前块头部的大小上，这两个块就在常数时间内被合并了。

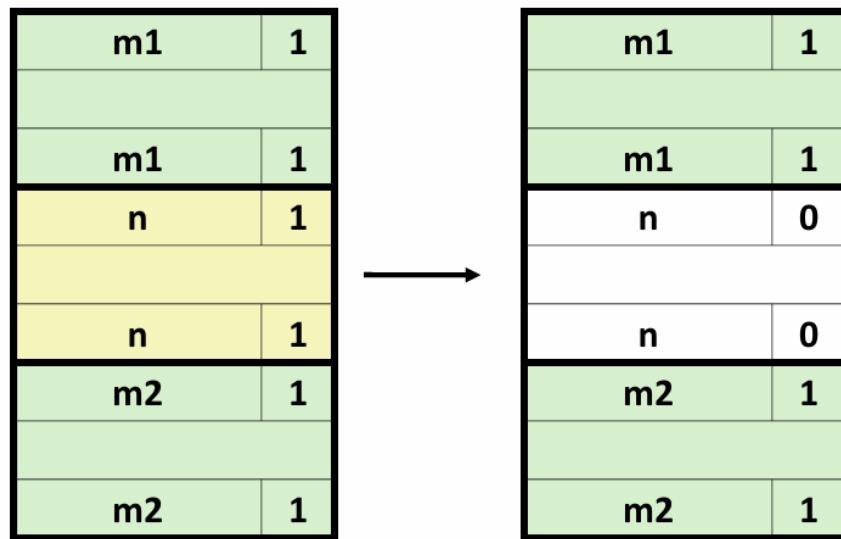
但是如何合并前面的块呢？如果是在一个带头部的隐式空闲链表中，唯一的选择就是搜索整个链表，记住前面块的位置，直到到达当前块。这无疑会浪费大量的时间。

Donald Knuth (没错，又是这位传奇人物) 提出了一种聪明且通用的技术，叫做**边界标记** (boundary tag)，允许在常数时间内进行对前面块的合并。这种思想是在每个块的结尾处添加一个**脚部** (footer, 边界标记)，如下图所示。脚部就是头部的一个副本 (记录块大小和分配状态)，现在分配器就可以通过检查每个块的脚部，来判断前面一个块的起始位置和状态。

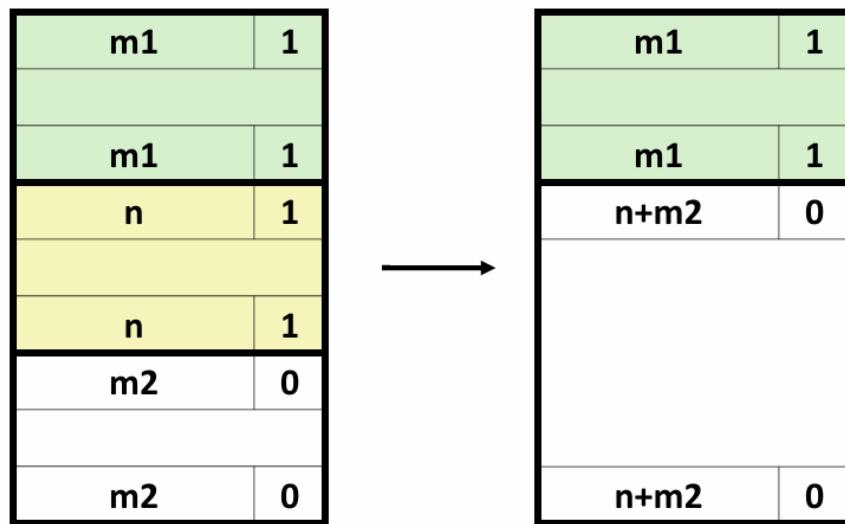


下面列举了分配器释放当前块时的不同情况中，使用边界标记合并的结果（在每种情况下中，合并都是在常数时间内完成的）：

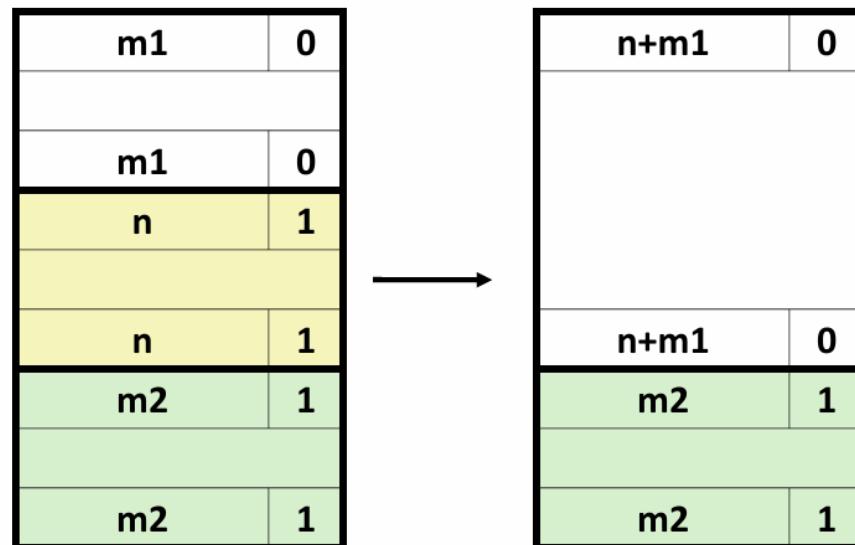
1. 前面的块和后面的块都是已分配的。



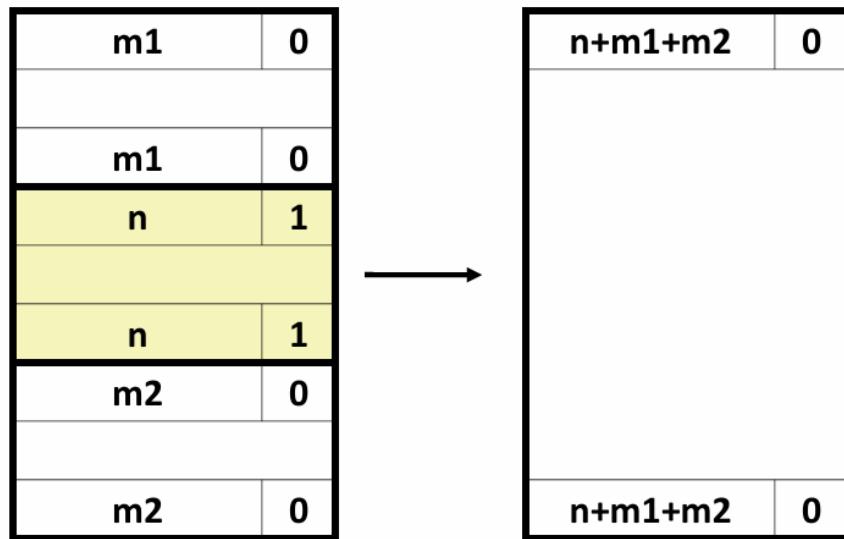
2. 前面的块是已分配的，后面的块是空闲的。



3. 前面的块是空闲的，后面的块是已分配的。

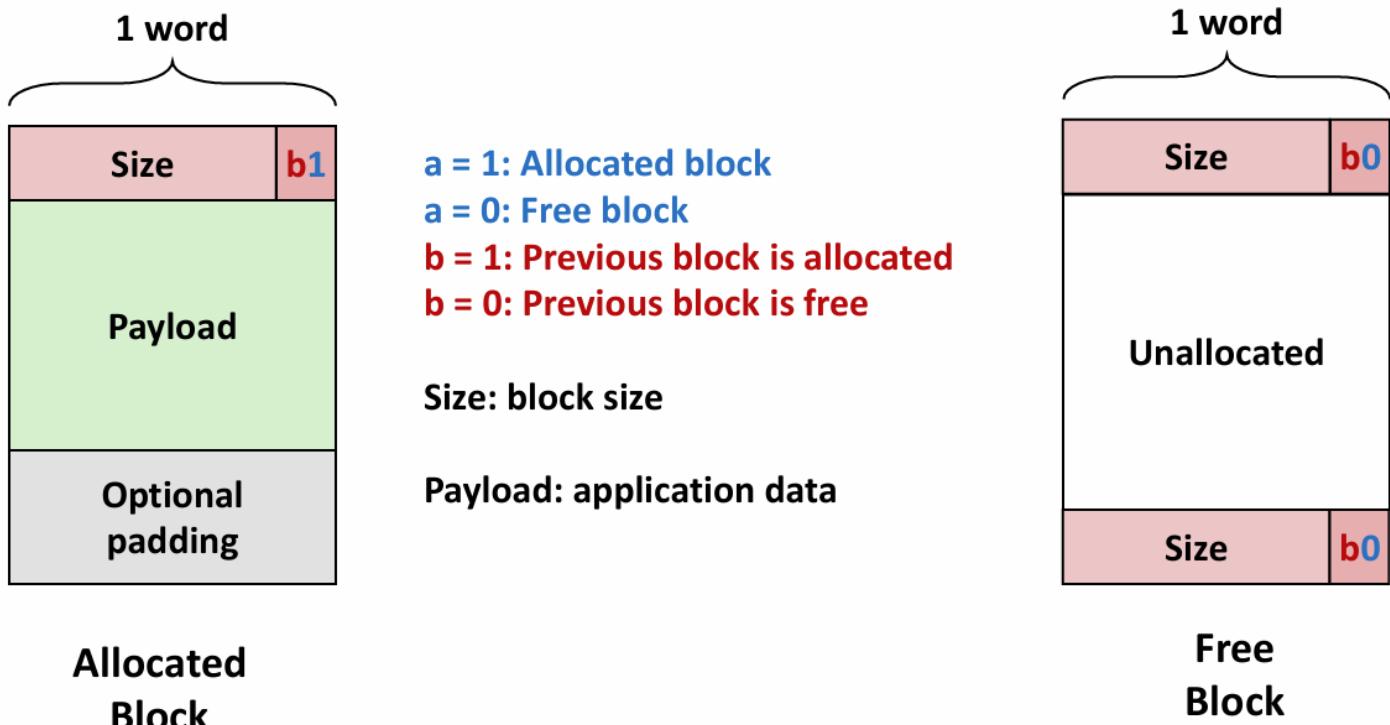


4. 前面的块和后面的块都是空闲的。



边界标记的概念是简单优雅的，它对许多不同类型的分配器和空闲链表组织都是通用的。然而，它也存在一个潜在的缺陷：它要求每个块都保持一个头部和一个脚部，在应用程序操作许多个小块时，会产生显著的内存开销。

幸运的是，有一种非常聪明的边界标记优化方法。想一想究竟是哪些块需要一个脚部？没错，只有前面的块是空闲块时，当前块才需要用到脚部。也就是说，已分配块中不需要脚部。因此，我们只需将前面块的分配位存放在当前块中多出来的低位中即可（双字对齐时，size是16的整数倍，因此有4个位可以空出，用于存储前面块的分配状态和当前块的分配状态）。如下图所示：



---

到此为止，我们介绍了动态内存分配的基础知识。在下一节中，我们将学习一些进阶的知识，包括更复杂的空闲链表组织结构，并尝试实现一个简单的分配器。

---

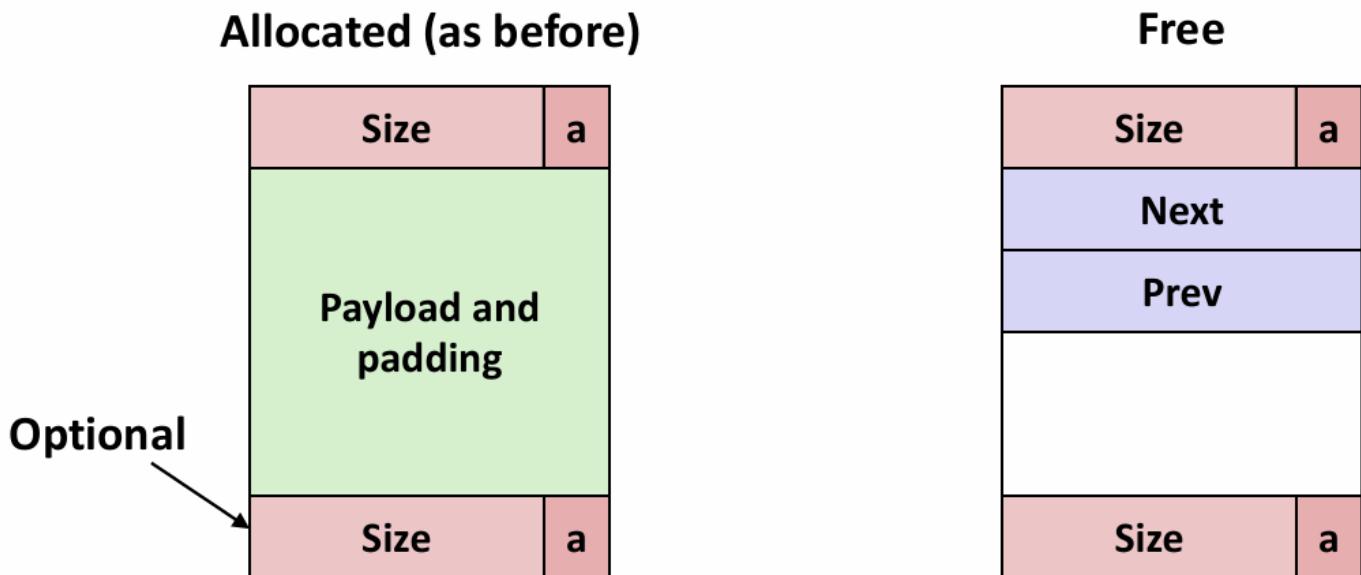
© 2025. ICS Team. All rights reserved.

# Chapter 9.5 Dynamic Memory Allocation: Advanced Concepts

## Explicit Free Lists

隐式空闲链表是一种较为简单的空闲链表组织结构。然而，由于块分配与堆块的总数呈线性关系，所以对于通用的分配器，隐式空闲链表是不适合的。一种更好的方法是将空闲块组织为某种形式的显式数据结构。因为程序实际上不需要一个空闲块的主体，因此实现这个数据结构的指针就可以存放在这些空闲块的主体中。

例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个prev（前驱）和next（后继）指针，如下图所示：



使用双向链表代替隐式空闲链表，我们实际上只需要维护空闲块的链表，这使得首次适配的分配时间从块总数的线性时间减少到空闲块数量的线性时间。

不过对于释放一个块来说，它的时间可能是线性的，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用LIFO

的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

另一种方法是按照**地址顺序**来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。不过，按照地址排序的首次适配比LIFO排序的首次适配有更高的内存利用率，接近于最佳适配的内存利用率。

一般而言，显示链表的缺点是空闲块必须足够大，这样才能够包含所有需要的指针，以及头部和可能需要的脚部。这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

## Segregated Free Lists

一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块。一种流行的减少分配时间的方法，通常称为**分离存储** (segregated storage)，就是维护多个空闲链表，其中每个链表中的块有大致相等的大小。一般的思路是将所有可能的块大小分成一些等价类，也叫做**大小类** (size class)。

分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序排列。当分配器需要一个大小为n的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，它就搜索下一个链表，以此类推。

分离的空闲链表的优势是有更高的吞吐率和内存利用率。例如对于以2的幂次方划分的的大小类，分配时间从线性缩短到对数时间。同时，它的首次适配搜索近似于整个堆的最佳适配搜索。（极端情况：每个块大小都有一个对应的大小类，此时等价于最佳适配。）

# Chapter 9.6 Implementing a Simple Allocator

本节将基于前面所讲知识，从头到尾实现一个简单的分配器。这也直接对应我们课程的malloc lab。当然，本节内容也只是为构建动态内存分配器提供一种简单的思路和实现框架，具体不同层面上的实现还需要同学们自行完成。

我们所实现的分配器基于隐式空闲链表，使用立即边界标记合并方式，最大块大小为 $2^{32} = 4GB$ 。代码可以不加修改地运行在32位或64位的进程中。

## General Allocator Design

我们的分配器使用 `memlib.c` 包所提供的一个内存系统模型，它允许我们在不干涉已存在的系统层 `mmalloc` 包的情况下，运行分配器。

`memlib.c` 实现如下：

```

/*Private global variables*/
static char *mem_heap; /*Points to first byte of heap*/
static char *mem_brk; /*Points to last byte of heap plus 1*/
static char *mem_max_addr; /*Max legal heap addr plus 1*/

/*
*mem_init-Initialize the memory system model
*/
void mem_init(void)
{
    mem_heap=(char*)Malloc(MAX_HEAP);
    mem_brk=(char*)mem_heap;
    mem_max_addr=(char*)(mem_heap+MAX_HEAP);
}

/*
*mem_sbrk-Simple model of the sbrk function. Extends the heap
* by incr bytes and returns the start address of the new area. In
* this model, the heap cannot be shrunk.
*/
void *mem_sbrk(int incr)
{
    char *old_brk=mem_brk;

    if((incr<0) || ((mem_brk+incr)>mem_max_addr)){
        errno=ENOMEM;
        fprintf(stderr,"ERROR: mem_sbrk failed. Ran out of memory...\n");
        return (void*)-1;
    }
    mem_brk+=incr;
    return (void*)old_brk;
}

```

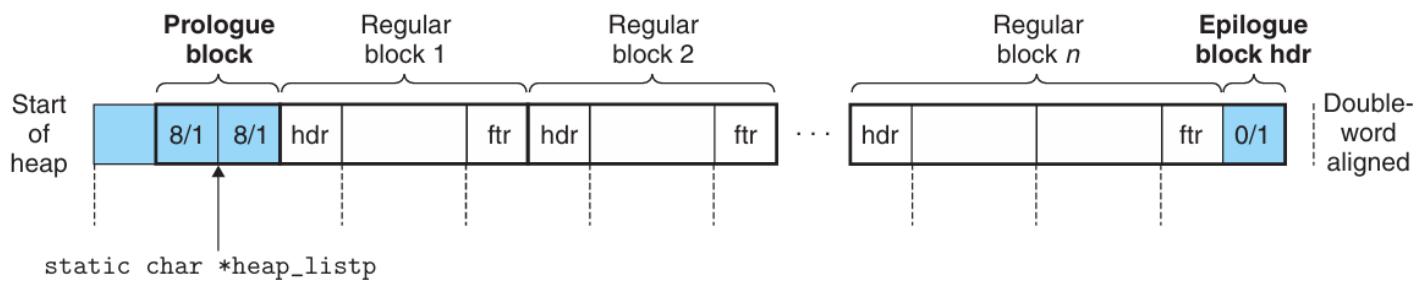
分配器包含在一个源文件 mm.c 中，用户可以编译和链接这个源文件到他们的应用中。分配器输出三个函数到应用中：

```

extern int mm_init(void);
extern void *mm_malloc(size_t size);
extern void mm_free(void *ptr);

```

mm\_init函数初始化分配器，如果成功返回0，否则返回-1。mm\_malloc和mm\_free函数与它们对应的系统函数有相同的接口和语义。分配器使用我们在9.4小节提到的带边界标记的块格式（块结尾加一个footer）。最小块大小为16字节，空闲链表组织成如下形式的隐式空闲链表：



第一个字是一个双字边界对齐的不使用的填充字（padding），填充后面紧跟着一个特殊的**序言块**（prologue block），这是一个8字节的已分配块，只有一个头部和脚部组成。序言块是在初始化时创建的，并且永不释放。在序言块后面紧跟的是零个或多个由malloc或者free调用创建的普通块。堆的末尾是一个特殊的**结尾块**（epilogue block），这个块是一个大小为零的已分配块，只有一个头部组成。这种设置的好处我们在前面的章节已经提到过，它是一种消除合并时边界条件的技巧。分配器使用一个单独的私有全局变量`heap_listp`，它总是指向序言块。这里我们可以做一个优化，即让它指向下一个块，而不是这个序言块。

## Basic Constants and Macros for Manipulating the Free List

下面展示了一些我们在分配器编码中将要使用的基本常数和宏。第2~4行定义了一些基本的大小常数。第9~25行定义了一小组宏来访问和遍历空闲链表。

```
1 /*Basic constants and macros*/
2 #define WSIZE 4 /*Word and header/footer size (bytes)*/
3 #define DSIZE 8 /*Double word size (bytes)*/
4 #define CHUNKSIZE (1<<12) /*Extend heap by this amount (bytes)*/
5
6 #define MAX(x,y) ((x)>(y)?(x):(y))
7
8 /*Pack a size and allocated bit into a word*/
9 #define PACK(size,alloc) ((size)|(alloc))
10
11 /*Read and write a word at address p*/
12 #define GET(p) (*(unsignedint*)(p)) //读取和返回参数p引用的字
13 #define PUT(p,val) (*(unsignedint*)(p)=(val)) //将val存放在参数p指向的字中
14
15 /*Read the size and allocated fields from address p*/
16 #define GET_SIZE(p) (GET(p)&~0x7)//从地址p的头部或脚部返回大小
17 #define GET_ALLOC(p) (GET(p)&0x1)//从地址p的头部或脚部返回已分配位
18
19 /*Given block ptr bp, compute address of its header and footer*/
20 #define HDRP(bp) ((char*)(bp)-WSIZE)//返回指向这个块的头部的指针
21 #define FTRP(bp) ((char*)(bp)+GET_SIZE(HDRP(bp))-DSIZE)//返回指向这个块的脚部的指针
22
23 /*Given block ptr bp, compute address of next and previous blocks*/
24 #define NEXT_BLKP(bp) ((char*)((bp)+GET_SIZE(((char*)(bp)-WSIZE)))//返回指向后面的块的块指针
25 #define PREV_BLKP(bp) ((char*)((bp)-GET_SIZE(((char*)(bp)-DSIZE)))//返回指向前面的块的块指针
```

## Creating the Initial Free List

在调用mm\_malloc或者mm\_free之前，应用必须通过调用mm\_init函数来初始化堆。

mm\_init函数从内存系统中得到4个字，并将它们初始化，创建一个空的空闲链表。然后调用extend\_heap函数，该函数将堆扩展CHUNKSIZE字节，并且创建初始的空闲块。

```
1 int mm_init(void)
2 {
3     /*Create the initial empty heap*/
4     if((heap_listp=mem_sbrk(4*WSIZE))==(void*)-1)
5         return -1;
6     PUT(heap_listp,0); /*Alignment padding*/
7     PUT(heap_listp+(1*WSIZE),PACK(DSIZE,1));/*Prologue header*/
8     PUT(heap_listp+(2*WSIZE),PACK(DSIZE,1));/*Prologue footer*/
9     PUT(heap_listp+(3*WSIZE),PACK(0,1)); /*Epilogue header*/
10    heap_listp+=(2*WSIZE);
11
12    /*Extend the empty heap with a free block of CHUNKSIZE bytes*/
13    if(extend_heap(CHUNKSIZE/WSIZE)==NULL)
14        return -1;
15    return 0;
16 }
```

## Freeing and Coalescing Blocks

应用通过调用mm\_free函数来释放一个以前分配的块，这个函数释放所请求的块（bp），然后使用边界标记合并技术将之与邻接的空闲块合并起来。

```
1 void mm_free(void *bp)
2 {
3     size_t size=GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp),PACK(size,0));
6     PUT(FTRP(bp),PACK(size,0));
7     coalesce(bp);
8 }
9
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc=GET_ALLOC(FTRP(PREV_BLKP(bp)));
13     size_t next_alloc=GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14     size_t size=GET_SIZE(HDRP(bp));
15
16     if(prev_alloc&&next_alloc){ /*Case1*/
17         return bp;
18     }
19
20     else if(prev_alloc&&!next_alloc){ /*Case2*/
21         size+=GET_SIZE(HDRP(NEXT_BLKP(bp)));
22         PUT(HDRP(bp),PACK(size,0));
23         PUT(FTRP(bp),PACK(size,0));
24     }
25
26     else if(!prev_alloc&&next_alloc){ /*Case3*/
27         size+=GET_SIZE(HDRP(PREV_BLKP(bp)));
28         PUT(FTRP(bp),PACK(size,0));
29         PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
30         bp=PREV_BLKP(bp);
31     }
32
33     else{ /*Case4*/
34         size+=GET_SIZE(HDRP(PREV_BLKP(bp)))+
35             GET_SIZE(FTRP(NEXT_BLKP(bp)));
36         PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
37         PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));
38         bp=PREV_BLKP(bp);
39     }
40     return bp;
41 }
```

## Allocating Blocks

一个应用通过调用mm\_malloc函数来向内存请求大小为size字节的块。在检查完请求的真假后，分配器必须调整请求块的大小，从而为头部和脚部留有空间，并满足双字对齐的要求。

一旦分配器调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块，如果有合适的，那么分配器就放置这个请求块，并可选地分割出多余的部分，然后返回新分配块的地址。

如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，把请求放置在这个新的空闲块里，可选地分割这个块，然后返回一个指针，指向这个新分配的块。

```
1 void *mm_malloc(size_t size)
2 {
3     size_t asize; /* Adjusted block size*/
4     size_t extendsize; /* Amount to extend heap if no fit*/
5     char *bp;
6
7     /*Ignore spurious requests*/
8     if(size==0)
9         return NULL;
10
11    /*Adjust block size to include over head and alignment reqs.*/
12    if(size<=DSIZE)
13        asize=2*DSIZE;
14    else
15        asize=DSIZE*((size+(DSIZE)+(DSIZE-1))/DSIZE);
16
17    /*Search the free list for a fit*/
18    if((bp=find_fit(asize))!=NULL){
19        place(bp,asize);
20        return bp;
21    }
22
23    /*No fit found. Get more memory and place the block*/
24    extendsize=MAX(asize,CHUNKSIZE);
25    if((bp=extend_heap(extendsize/WSIZE))==NULL)
26        return NULL;
27    place(bp,asize);
28    return bp;
29 }
```

以上就是采用隐式空闲链表实现的一个简单的动态内存分配器。本章内容到这里也就结束了。

# Developer Ops

---

Talk is cheap, show show way 

-- ICS Team

---

这一部分我们会介绍一些基础运维的知识， 相当于简单版的 “A Missing Semester”

为什么要学习运维？

很简单，因为我们学的是计算机科学与技术，而不是计算机科学与艺术...

计算机科学是一门强调动手实践的学科。我们不能仅仅停留在对理论的了解，更要具备将理论知识应用于实际场景的能力。运维便是连接理论与实践的桥梁：

它涉及到计算机系统的配置、部署、监控、维护和优化，是确保软件系统稳定、高效运行的关键环节。掌握基本的运维技能，能够帮助我们：

1. 更好地理解系统底层原理
2. 提升问题解决能力

ICS-Team 计划从 Spring 2025 开始，加入基础运维的知识讲解，包含 git / github / ssh / vim / vscode 等工具的使用，希望通过这部分内容的学习，能够帮助大家更好地掌握计算机技术，为未来的 科学研究 / 团队开发 打下坚实的基础。

我们的假设是：

1. 面向读者：对计算机运维完全没有了解的学生，
2. 讲解方式：最简单最形象的“人话”
3. 难度系数：入门级，不需要任何前置知识

希望你能有所收获 

---

© 2025. ICS Team. All rights reserved.

# How to Use VSCode

这篇笔记会手把手教你如何在不同系统的电脑上安装VScode；如何写配置文件，让你的C++代码跑起来；跑起来以后如何Debug...

此外，我们还会从一个“老鸟”程序员的视角，给大家推荐一些辅助后期开发的插件

## 为什么要用 VSCode

都2025年了，不会真有人还在用 Dev C++ 等小学生工具写C++代码吧

选择VSCode的理由包括但不限于：

1. 开源：版本迭代快，功能齐全
2. 几乎可以用来编写任何语言、甚至能写文档、渲染PDF：只要是关于文本编辑/代码编辑，VS Code都有相应的插件去支持
3. 插件市场庞大，定制化程度高：VS Code通过内置的设置和插件市场基本都可以满足你的需求

这就是为什么VSCode被誉为“新编辑器之神” 

### Note

一般意义上，我们认为：“编辑器之神”是Vim；“神之编辑器”是Emacs，详见 [《编辑器之战》](#)

但是在2025年的视角来看，笔者更倾向于认为：

“编辑器之神”是VSCode；“神之编辑器”是Vim

## 如何安装 VSCode

## For MacOS

为什么总是把MacOS优先介绍，主要原因有两点：

第一：

笔者是Mac战神，基本没用过Windows（刚上大一用过，太难用了，直接换电脑！）

第二：

我相信在座的绝大多数人都使用windows系统！我猜测是高考结束后被各种论坛（知乎/B站/...）上的“Mac不适合工科学生”等言论吓到了。没办法，只能入手windows 😅

在这种背景下，Mac用户作为“长期被忽略”的“少数群体”，很难得到相关的指导与教程 😭

另外，以往的任课老师大多会以Windows系统为例说明如何写代码，这对使用苹果电脑也就是Mac的同学很不友好。而且有些老师在编程环境的配置、编辑器的选取上也会有一些忽略，这部分的内容往往让同学们自学；而网上的博客或教程良莠不齐，很多时候会出现版本错误/系统不匹配/配置过时等现象

因此，这份教程提供对于 MacOS (Apple Silicon) 的手把手教学！如果你有什么困惑与建议，欢迎课下跟TA交流 🍹

## 基础编程环境配置

提前说明，下列步骤需要严格遵循给出的顺序，并且要确保全程可以使用“墙外的网”⚠️

### (1) 安装 XCode

xcode 是苹果提供的一个开发工具集，类似于微软的 visual studio

1. 从 App store 或苹果开发者网站安装 Xcode。
2. 待XCode安装完毕，安装 Xcode command line tools，只需要在CLI中运行：

```
xcode-select --install
```

运行命令后，按照指引，你将完成 Xcode command line tools 安装 🌸

### (2) 安装 Homebrew

包管理工具让软件的安装（升级）和卸载都变得简便了许多。Homebrew 就是 Mac 下面的这样一个包管理工具（类似于 `apt-get`, `yum`）。

在你电脑的Terminal中，按照[官网给出的安装方式](#)做：

```
# 一键安装
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

### (3) 安装编译器g++

省流：当你安装好XCode时，它就已经自带g++了

你可以在CLI中看看长啥样：

```
> g++ --version
Apple clang version 15.0.0 (clang-1500.1.0.2.5)
Target: arm64-apple-darwin24.0.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

这里有个很有意思的现象：

#### Note

明明我装的是g++，为什么上面显示 `clang version 15.0.0`？

这是因为在 macOS 上，`g++` 实际上是 `clang++` 的一个符号链接。macOS 系统自带的编译器工具链并不是 GNU GCC，而是基于 LLVM 的 Clang 工具链。

所以，当你运行 `g++` 时，它会调用 Clang 编译器，而 Clang 的版本显示出来了。即使你安装了 GNU 的 GCC 编译器，`g++` 可能依然会指向 Clang。

### (4) 安装编辑器VSCode

VSCode，全名是Visual Studio Code.app，一般缩写成 vsc

在微软[VSCode官网](#)直接下载macOS平台的最新版本即可

下载安装后，双击打开应用即可，所有的安装流程到此结束，下面开始书写配置文件

## 书写配置文件

VSCode对c++的配置是以文件夹（及其内部的文件系统）为覆盖单位，这点跟python不同。python有全局，也有基于文件夹的虚拟环境(pyenv)，也有池化的(conda)。

C++ for VSC 是基于文件夹的，可以给不同的项目文件夹建立不同的配置。

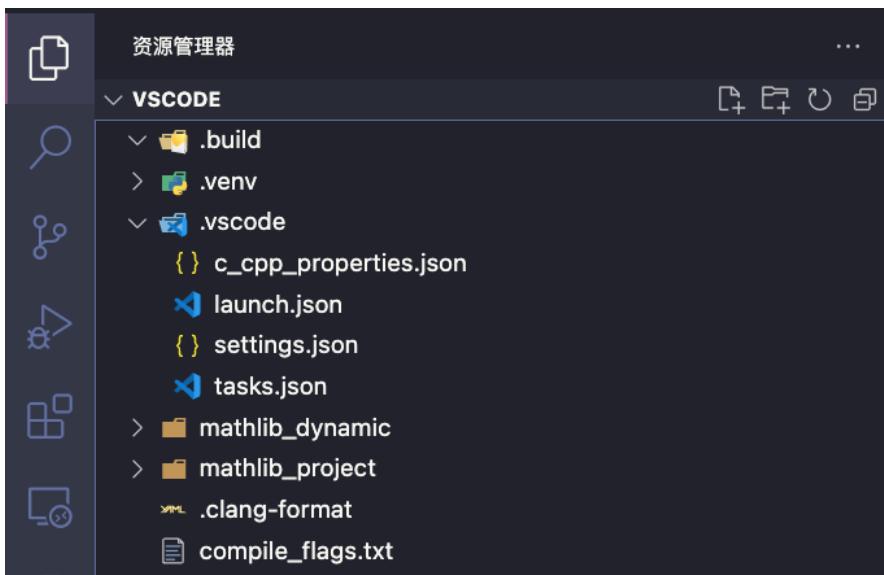
这里笔者介绍的主要应用场景是 单文件cpp 的编译（比如你要写洛谷的算法题），并不一定适用于 CMake 等工具。

我会在每一个部分附上参考脚本，它们都是开箱即用的，你可以直接复制粘贴到你的项目中

### (1) 建立项目文件夹

我们需要先选中/新建一个合适的项目文件夹，然后使用VSCode打开它，在这里我们将书写“专用于这个文件夹”的配置文件。

形如这份项目构建示意图 (你可以自动忽略 .venv / mathlib\_dynamic / mathlib\_project / .DS\_Store 及其子文件系统)：



```
> tree -L 2 -a
.
├── .DS_Store
├── .build
├── .clang-format
└── .venv
    ├── bin
    ├── include
    ├── lib
    ├── pyvenv.cfg
    └── share
└── .vscode
    ├── c_cpp_properties.json
    ├── launch.json
    ├── settings.json
    └── tasks.json
├── compile_flags.txt
└── mathlib_dynamic
    ├── lib
    ├── main
    ├── main.cpp
    ├── mathlib.cpp
    └── mathlib.h
└── mathlib_project
    ├── lib
    ├── main
    ├── main.cpp
    ├── mathlib.cpp
    └── mathlib.h
```

12 directories, 16 files

所有对c++的配置都在 .vscode 文件夹下，其余我们先暂时不用管

## (2) 配置文件

1. 确保 clang++ 已经正确安装（通过 clang++ -v 可以验证）
  - 对于 macOS, 运行 xcode-select --install 可以安装好本文用到的所有包
2. 确保 vscode 已启用 **CodeLLDB** 插件（报错无法下载可以先按报错给的 url 用浏览器下载，然后手动安装）
3. tasks.json, 放入 .vscode 文件夹中
4. launch.json, 放入 .vscode 文件夹中
5. 在文件夹里新建一个 .build 文件夹（macOS / Linux 必做）

- 把所有的可执行文件放到 `./build/` 文件夹下，这样可以避免污染项目根目录，也可以方便地清理编译产生的中间文件。

6. 按 F5 (FN + F5)，就可以编译调试了

参考的 `task.json`：

```
{  
  "version": "2.0.0",  
  "tasks": [  
    {  
      "type": "shell",  
      "label": "C/C++: clang++ build active file",  
      "command": "/usr/bin/clang++", // `which clang++` may help you find  
      this path  
      "args": [  
        "--std=c++17",  
        "--fcolor-diagnostics",  
        "--fansi-escape-codes",  
        "-g",  
        "${file}",  
        "-o",  
        "${workspaceFolder}/.build/${fileBasenameNoExtension}"  
        "-fstandalone-debug", // to enable viewing std::string etc. when  
        using lldb on Windows or Linux  
      ],  
      "options": {  
        "cwd": "${fileDirname}"  
      },  
      "group": {  
        "kind": "build",  
        "isDefault": true  
      },  
      "detail": "Task generated by Debugger."  
    }  
  ]  
}
```

参考的 `launch.json`：

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "C/C++: clang++ build and debug active file customize",
      "type": "lldb",
      "request": "launch",
      "program": "${workspaceFolder}/.build/${fileBasenameNoExtension}",
      "args": [],
      "cwd": "${workspaceFolder}",
      "preLaunchTask": "C/C++: clang++ build active file"
    },
    {
      "name": "C/C++ Runner: Debug Session",
      "type": "lldb",
      "request": "launch",
      "args": [],
      "cwd": "/Users/huluobo/code_projects/vscode",
      "program": "/Users/huluobo/code_projects/vscode/build/Debug/outDebug"
    }
  ]
}
```

参考的 c\_cpp\_properties.json :

```
{
  "configurations": [
    {
      "name": "macos-clang-arm64",
      "includePath": [
        "${workspaceFolder}/**"
      ],
      "compilerPath": "/usr/bin/clang",
      "cStandard": "${default}",
      "cppStandard": "${default}",
      "intelliSenseMode": "macos-clang-arm64",
      "compilerArgs": [
        ""
      ]
    }
  ],
  "version": 4
}
```

参考的 settings.json :

```
{  
  "files.associations": {  
    "iostream": "cpp",  
    "cstring": "cpp",  
    "algorithm": "cpp",  
    "queue": "cpp",  
    "iomanip": "cpp",  
    "__config": "cpp"  
  },  
  "C_Cpp_Runner.cCompilerPath": "clang",  
  "C_Cpp_Runner.cppCompilerPath": "clang++",  
  "C_Cpp_Runner.debuggerPath": "lldb",  
  "C_Cpp_Runner.cStandard": "",  
  "C_Cpp_Runner.cppStandard": "",  
  "C_Cpp_Runner.msvcBatchPath": "",  
  "C_Cpp_Runner.useMsvc": false,  
  "C_Cpp_Runner.warnings": [  
    "-Wall",  
    "-Wextra",  
    "-Wpedantic",  
    "-Wshadow",  
    "-Wformat=2",  
    "-Wcast-align",  
    "-Wconversion",  
    "-Wsign-conversion",  
    "-Wnull-dereference"  
  ],  
  "C_Cpp_Runner.msvcWarnings": [  
    "/W4",  
    "/permissive-",  
    "/w14242",  
    "/w14287",  
    "/w14296",  
    "/w14311",  
    "/w14826",  
    "/w44062",  
    "/w44242",  
    "/w14905",  
    "/w14906",  
    "/w14263",  
    "/w44265",  
    "/w14928"  
  ],  
  "C_Cpp_Runner.enableWarnings": true,  
  "C_Cpp_Runner.warningsAsError": false,  
  "C_Cpp_Runner.compilerArgs": [],  
  "C_Cpp_Runner.linkerArgs": []  
}
```

```
"C_Cpp_Runner.includePaths": [],
"C_Cpp_Runner.includeSearch": [
    "/*",
    "**/*"
],
"C_Cpp_Runner.excludeSearch": [
    "**/build",
    "**/build/**",
    "**/.**",
    "**/.*/**",
    "**/.vscode",
    "**/.vscode/**"
],
"C_Cpp_Runner.useAddressSanitizer": false,
"C_Cpp_Runner.useUndefinedSanitizer": false,
"C_Cpp_Runner.useLeakSanitizer": false,
"C_Cpp_Runner.showCompilationTime": false,
"C_Cpp_Runner.useLinkTimeOptimization": false,
"C_Cpp_Runner.msvcSecureNoWarnings": false
}
```

现在你的C++代码应该可以跑起来了, Congratulations!

如果你好奇这些json是干什么用的, 它们对VSC做了什么, 欢迎移步至这篇[Blog](#)。这篇笔记的配置部分到此结束 !

## For Windows

前往[VSCode官网](#)下载安装包, 下载结束后双击打开安装包, 按照提示一步步安装即可.

在纯 Windows 环境下搭建开发环境, 尤其是 C/C++ 开发环境是一种精神酷刑. 所以我推荐你用 WSL2, 在 Windows 继承的 Linux 环境中进行开发. 具体教程请参考[这里](#). 后续的配置同 Linux 部分.

## For Linux

首先祝贺你愿意使用 Linux 作为你的主力环境. 恭喜你向极客迈进了一步! 

如果你正在使用 Arch Linux, 那太棒了, 我太喜欢你了! 可以加个 QQ 吗? 

## Arch Users

对 Arch Linux 用户, 想要安装 VSCode, 有以下三种方式:

1. 从 [arch4edu](#) 源安装 (最推荐)
2. 从 AUR 安装 (需要你能科学上网)
3. 从 VSCode 官网安装包安装 (不推荐)

### 从 arch4edu 源安装

首先添加 arch4edu 源, 如果你没添加过的话:

- 编辑 `/etc/pacman.conf` 文件. 你用 vim, neovim, nano 什么都好, 只要你会用就行. 然后在文件末尾添加以下内容:

```
[arch4edu]
SigLevel = Optional TrustAll
Server = https://mirrors.cernet.edu.cn/arch4edu/$arch
```

- 刷新 pacman 缓存: `sudo pacman -Syyu`
- 安装 VSCode: `sudo pacman -S visual-studio-code-bin`

### 从 AUR 安装

- 首先确保你有 yay 包管理器, 如果没有, 请安装先添加 [archlinux-cn](#) 源, 然后运行 `sudo pacman -S yay` 安装.
- 然后运行 `yay -S visual-studio-code-bin` 安装 VSCode.

## Debian/Ubuntu Users

对于 Debian/Ubuntu 用户, 你可以直接从[VSCode官网](#)下载 .deb 安装包, 然后使用 apt 安装:

```
sudo apt install <your-downloaded-vscode>.deb
```

但有一说一, Ubuntu 全是私货, 系统臃肿容易崩溃, 非常不推荐使用.

## 为什么你应该使用 Arch Linux

## 1. 极简主义与高度定制化

- **纯净起点**：Arch 不预装冗余软件，安装后仅包含基础系统(base-devel)，用户可完全按需构建系统环境。这种“空白画布”特性吸引了追求系统精简和掌控力的用户。
- **DIY 哲学**：从内核模块到桌面环境(如 KDE/GNOME 或轻量级 WM)，所有组件均由用户主动选择，避免了其他发行版因预装软件导致的资源占用或风格冲突。

## 2. 滚动更新与软件时效性

- **前沿软件生态**：作为滚动更新(Rolling Release)发行版，Arch 用户可直接获取最新稳定版软件(如 Linux 内核、开发工具链)，无需等待大版本升级。对开发者、硬件兼容性(如新显卡支持)或追求新功能的用户至关重要。
- **更新可控性**：通过定期维护(如查看 [Arch News](#))和谨慎处理关键包(如 `pacman -Syu` 前检查更新日志)，可有效避免更新冲突。

## 3. Arch User Repository (AUR)

- **海量软件覆盖**：AUR 社区仓库提供超过 8 万个第三方软件包(如小众工具、闭源程序)，通过 `yay` / `paru` 等工具可一键编译安装，极大扩展了软件可用性。
- **用户贡献驱动**：AUR 的开放性允许用户快速发布新软件包或补丁，解决了其他发行版仓库更新滞后的问题(例如第一时间体验 beta 版应用)。

## 4. 卓越的文档与社区

- **Arch Wiki**：被公认为 Linux 领域最全面、细致的文档库，涵盖系统配置、故障排查、软件优化等，即使非 Arch 用户也常参考。
- **问题解决效率**：社区倾向于提供“授人以渔”的解决方案(如解释配置原理而非直接给命令)，有助于用户深入理解系统运作。

## 5. 现代工具链与安装简化

- **安装体验进化**：官方提供的 `archinstall` 脚本简化了安装流程，30 分钟内即可完成基础系统部署，降低了传统 CLI 安装的学习曲线。
- **Pacman 包管理**：`pacman` 命令简洁高效(如 `pacman -Syu` 更新全系统，`pacman -Qs` 搜索包)，配合 AUR 助手工具形成强大的软件管理生态。

## 配置 C/C++ 开发环境

就我个人而言，我几乎不使用 VSCode 原生的配置文件。因为我也经常使用 NeoVim，所以需要跨编

辑器通用才行。我选用的方案是 CMake。这里就不展开讲解 CMake 的语法和教程了，可以通过这个仓库来学习 CMake 的用法。

想要让 CMake 与 VSCode 集成，可以安装以下插件，这也是我自己在使用的：

- CMake Tools
- CMake Language Support
- CMake Integration

如果你在寻找跨平台通用的格式化方案，我推荐 clang-format。VSCode 插件 Clang-Format 提供了集成方案。

## 如何使用VSCode进行调试

这一部分强烈建议跟着[官方文档](#)走一通

如果你觉得不够直观，可以看看这个：[一份CMU强烈推荐的Debugging Tutorial](#)

这些调试的技巧将会非常有利于你构建大规模代码库

## VSCode中的插件推荐

这一部分笔者会给一些自用的VSC插件推荐，亲测好用，不仅能提升日常代码开发效率，也可以美化界面，提升写代码的幸福感

当然这只是很小的一部分，如果你有什么比较好的建议，欢迎在仓库的issue中提出，或者提个PR



### 0) Chinese (Simplified)

适用于 VS Code 的中文（简体）语言包

### 1) Dracula Theme Official

一个非常适合程序员的主题，通过美观度提升写代码的幸福感

## 2) C/C++

写大型项目必备的道具，支持“一键跳转”等功能，神中神 💰

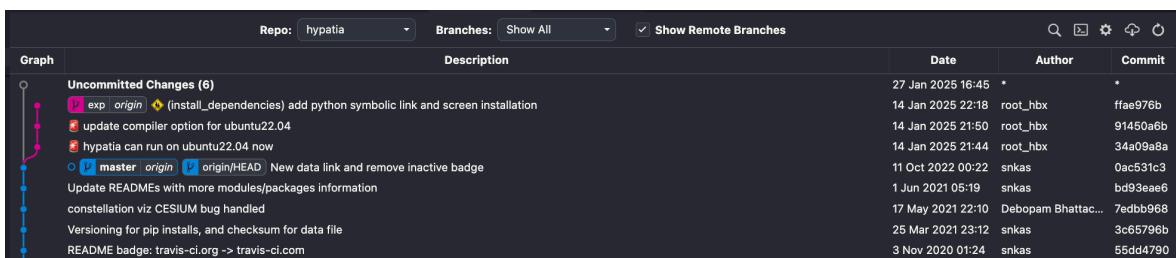
## 3) Github Copilot

这个不用说了吧 🎉



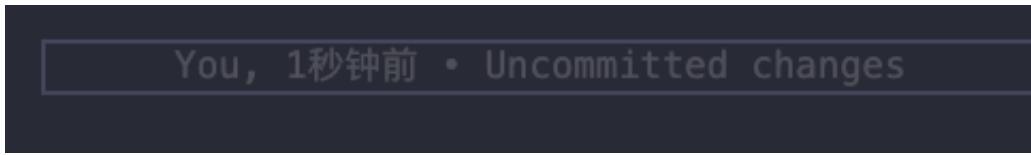
## 4) Git Graph

能够清晰地显示不同分支，并且进行快捷操作，神中神 😍



## 5) GitLens

清晰地显示每行代码的提交者，方便“找锅” 😅



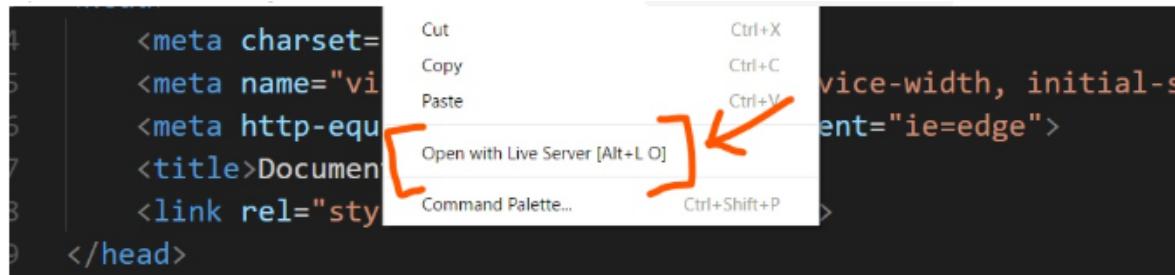
## 6) Gitmoji

让你的每个commit信息变得有趣👍

```
# my personal format  
🎉 (branch_name) commit message
```

## 7) Live Server

在VSC中快捷使用默认浏览器渲染HTML网页🌐



## 8) vscode-pdf 和 PDF Preview

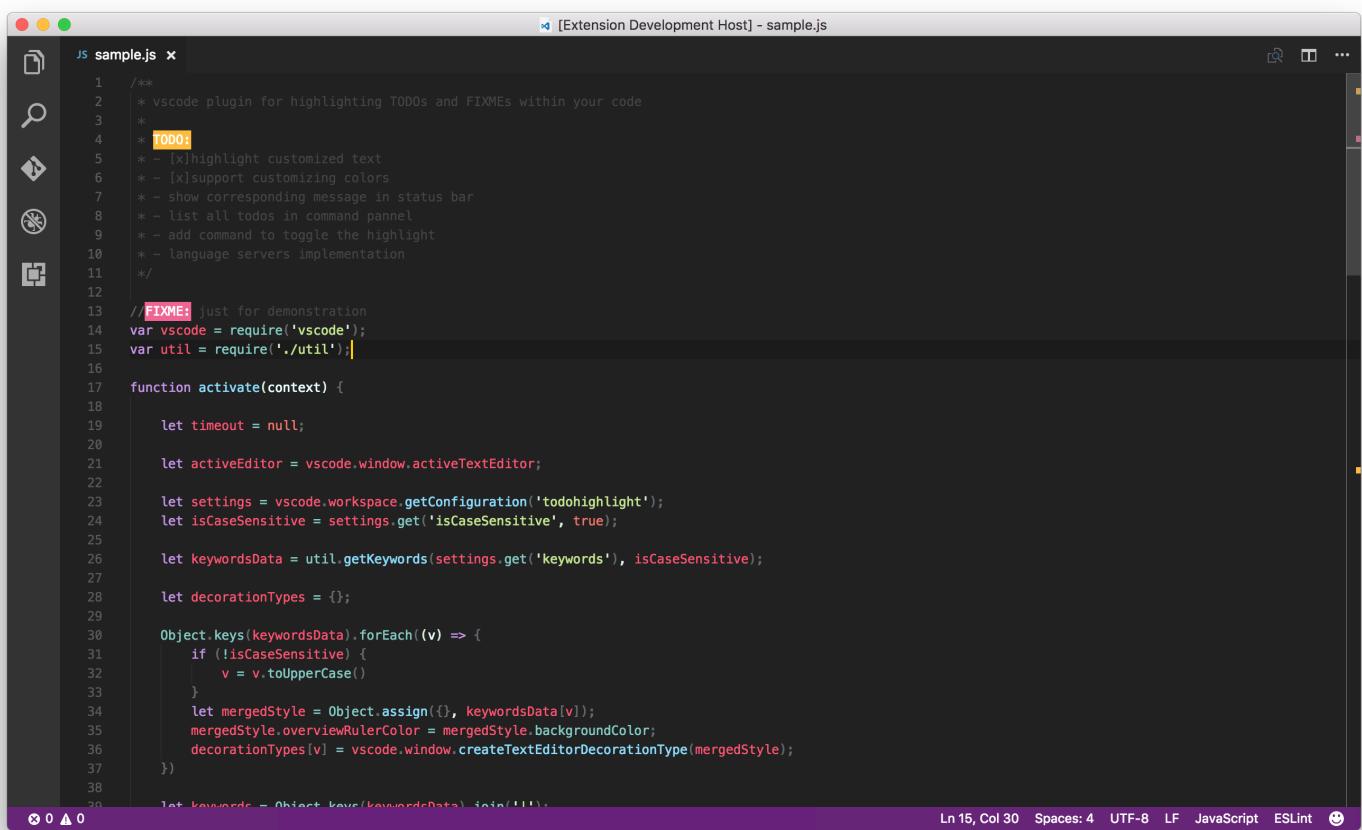
PDF预览👉

## 9) Remote - SSH

SSH远程连接🎉

## 10) TODO Highlight

高亮显示TODO等信息，醒目🌟



A screenshot of the Visual Studio Code (VS Code) interface. The title bar reads "[Extension Development Host] - sample.js". The left sidebar contains icons for file operations like Open, Save, Find, and Refresh. The main editor area shows a JavaScript file named "sample.js". The code includes several TODO and FIXME comments, which are highlighted with specific colors and styles. The status bar at the bottom right shows "Ln 15, Col 30" and other system information.

```
JS sample.js x [Extension Development Host] - sample.js
1  /**
2   * vscode plugin for highlighting TODOs and FIXMEs within your code
3   *
4   * [TODO]
5   * - [x]highlight customized text
6   * - [x]support customizing colors
7   * - show corresponding message in status bar
8   * - list all todos in command pannel
9   * - add command to toggle the highlight
10  * - language servers implementation
11  */
12
13 //FIXME: just for demonstration
14 var vscode = require('vscode');
15 var util = require('./util');
16
17 function activate(context) {
18
19     let timeout = null;
20
21     let activeEditor = vscode.window.activeTextEditor;
22
23     let settings = vscode.workspace.getConfiguration('todohighlight');
24     let isCaseSensitive = settings.get('isCaseSensitive', true);
25
26     let keywordsData = util.getKeywords(settings.get('keywords'), isCaseSensitive);
27
28     let decorationTypes = {};
29
30     Object.keys(keywordsData).forEach((v) => {
31         if (!isCaseSensitive) {
32             v = v.toUpperCase()
33         }
34         let mergedStyle = Object.assign({}, keywordsData[v]);
35         mergedStyle.overviewRulerColor = mergedStyle.backgroundColor;
36         decorationTypes[v] = vscode.window.createTextEditorDecorationType(mergedStyle);
37     })
38
39     let keywords = Object.keys(keywordsData).join(' ');
40
41     context.subscriptions.push(vscode.commands.registerCommand('todohighlight.toggle', () => {
42         if (timeout)
43             clearTimeout(timeout);
44         else
45             timeout = setTimeout(() => {
46                 vscode.window.showInformationMessage(`Todos: ${keywords}`);
```

## 11) indent-rainbow

高亮缩进，便于观察，尤其是大型项目开发

The screenshot shows a terminal window with the title 'cli.py 5, ↓M'. The code is a Python script named 'cli.py' with line numbers 4396 to 4459. The script defines a function 'serve\_down' which takes 'service\_names' (List[str]), 'all' (bool), 'purge' (bool), and 'yes' (bool). It handles cases for serving down specific replicas or all replicas. It also checks for conflicts between 'all' and 'replica\_id'. The code uses Click library for command-line argument handling. A commit message from 'Zhanghao Wu' is visible at the bottom of the code block.

```
sky > cli.py > serve_down
4396 def serve_down(service_names: List[str], all: bool, purge: bool, yes: bool,
4424     \b
4425     # Tear down a specific replica
4426     sky serve down my-service --replica-id 1
4427     \b
4428     # Forcefully tear down a specific replica, even in failed status.
4429     sky serve down my-service --replica-id 1 --purge
4430 """
4431 if sum([len(service_names) > 0, all]) != 1:
4432     argument_str = f'SERVICE_NAMES=("{".join(service_names)})' if len(
4433         service_names) > 0 else ''
4434     argument_str += ' --all' if all else ''
4435     raise click.UsageError(
4436         'Can only specify one of SERVICE_NAMES or --all. '
4437         f'Provided {argument_str!r}.')
4438
4439 replica_id_is_defined = replica_id is not None
4440 if replica_id_is_defined:
4441     if len(service_names) != 1:
4442         service_names_str = ', '.join(service_names)
4443         raise click.UsageError(f'The --replica-id option can only be used '
4444             f'with a single service name. Got: '
4445             f'{service_names_str}.')
4446 if all:
4447     raise click.UsageError('The --replica-id option cannot be used '
4448             'with the --all option.')
4449
4450 backend_utils.is_controller_accessible()  Zhanghao Wu, 11个月前 via PR #3288 + [Serve/Spot] Allow spot queue/cancel/logs durin...
4451 controller=controller_utils.Controllers.SKY_SERVE_CONTROLLER,
4452 stopped_message='All services should have been terminated.',
4453 exit_if_not_accessible=True]
4454
4455 if not yes:
4456     if replica_id_is_defined:
4457         click.confirm(
4458             f'Terminating replica ID {replica_id} in '
4459             f'{service_names[0]!r}. Proceed?',
```

## 12) CodeSnap

适合给代码截屏并展示，很美观👍

```
1  export const sortBy = ( ...cbs) => (a, b) => {
2      for (let i = 0; i < cbs.length; i++) {
3          const cb = cbs[i].desc ? cbs[i].cb : cbs[i];
4          const aa = cb(a);
5          const bb = cb(b);
6          const diff = cbs[i].desc
7              ? isString(aa)
8                  ? bb.localeCompare(aa)
9                      : bb - aa
10             : isString(aa)
11                 ? aa.localeCompare(bb)
12                     : aa - bb;
13             if (diff !== 0) return diff;
14     }
15     return 0;
16 };
17 export const desc = cb => ({ desc: true, cb });
```

# A Bite of Linux

## 为什么你应该用 Linux

---

冷知识0: 在 Linux 下一样可以玩 Windows 上的游戏, 诸如 Steam 上的各种游戏, 且很多时候性能比 Windows 更好. 你可以在 [ProtonDB](#) 上检查你 Steam 上喜欢的游戏的兼容性和性能

冷知识1: Arch Linux 的官方源里包含 Steam, 你可以直接从包管理器安装 Steam, 免去了上网和诈骗网站/Steam助手等斗智斗勇的烦恼

冷知识2: Mac(Apple Silicon) 对 Steam 的兼容性稀烂, 尤其是大型游戏

---

### 1. 贴近生产环境与开发者工具链

- 服务器环境主导: 绝大多数服务器, 云计算平台(如 AWS, Azure), 容器技术(如 Docker/Kubernetes)均基于 Linux. 熟悉 Linux 的操作和运维是开发者必备技能.
- 原生开发工具链: Linux 对编程语言(Python, C/C++, Java, Go 等), 编译器(GCC/Clang), 调试工具(GDB), 构建工具(Make/CMake)以及脚本环境(Bash)的支持更原生, 无需依赖第三方模拟器或兼容层.
- 包管理高效: 通过 `apt` (Debian/Ubuntu), `dnf` (Fedora) 或 `pacman` (Arch) 等包管理器, 可快速安装开发库, 工具和依赖项, 避免手动下载安装的繁琐.

### 2. 命令行与自动化能力

- 强大的 Shell 生态: Linux 的命令行工具(如 `grep`, `sed`, `awk`, `ssh`, `tmux`)和脚本能力(Bash/Python)是开发者的生产力工具, 适合处理文本, 自动化任务, 远程服务器管理.
- 开发流程无缝衔接: 从本地代码编写, 编译, 测试到部署到服务器, Linux 提供一致的环境, 避免跨平台兼容性问题(如 Windows 换行符, 路径分隔符导致的错误).

### 3. 开源与可定制性

- 系统透明可控: Linux 允许开发者深入操作系统内核, 网络协议栈, 文件系统等底层机制, 适合学习计算机原理(如操作系统, 编译原理课程).
- 高度可定制: 开发者可以自由配置开发环境(如窗口管理器, 终端工具链), 优化性能, 甚至修改内核参数, 满足特定需求.

### 4. 社区与学习资源

- 开发者导向的社区: Linux 生态的文档(如 `man` 手册, Arch Wiki), 论坛(Stack Overflow,

GitHub)和开源项目资源更贴近开发者需求, 问题解决效率高.

- 前沿技术支持: 多数开源项目(如 Kubernetes, TensorFlow, Node.js)优先适配 Linux, 新工具和框架在 Linux 上的支持更完善.

## 5. 轻量化与资源效率

- 低资源占用: Linux 对硬件资源(CPU, 内存)的需求更低, 适合在虚拟机, 老旧设备或笔记本上流畅运行, 提升开发效率.
- 稳定性与安全性: Linux 系统崩溃概率低, 病毒攻击风险小, 适合长期运行的开发任务.

---

Windows 和 Linux 占用资源对比:

我同一台电脑, 桌面壁纸采用 Wallpaper engine 加载同一张动态壁纸

Windows 11 进入系统, 任务管理器显示显存占用 5.9 GB

Arch Linux 进入 KDE Plasma 桌面, 任务管理器显示显存占用 0.8 GB

---

## Linux 小知识

按理来说, 我应该好好写写 Linux 的背景知识, 发展历程, GNU 和自由软件, 开源软件什么的. 但我懒得写了, 我打算教你一些黑话, 让你可以到处装X. 😎

## Linux, GNU/Linux, Systemd/Linux

如果有人向你提起 Linux, 你可以这么跟他说: 应该是 GNU/Linux 才对. 不过按现在各大 Distro 的情况来看, 说 Systemd/Linux 也不为过.

解释:

- Linux: 一个操作系统内核, 并不等于操作系统本身
- GNU/Linux: 当初 GNU 一直缺一个可用的内核, 于是把 Linux 拉了过来. GNU 的各个组件都能运行在 Linux 上, 而 Linux 作为承载 GNU 组件的内核. 这样的组合被成为 GNU/Linux, 并沿用至今
- Systemd/Linux: Systemd 是一套服务管理工具, 在现代 Linux Distro 中, 它大包大揽了非常多的事务. Linux 如果少了 GNU 组件, 一样能找到一些好用的替代品, 但要是少了 Systemd,

可能连启动都没法启动. 考虑到 GNU/Linux 名称的来历, 我们可以按同样的方法叫 Systemd/Linux.

---

Systemd 真的能引导启动, 我在用的就是 Systemd-boot 引导启动.

---

## Tragedy of Systemd

---

[视频链接](#)

---

简单来说, Systemd 是 Linux 中的一个初始化系统(init system)和服务管理器, 它提供了一系列用于初始化系统, 管理系统进程, 服务, 日志, 设备, 网络和其它系统资源的工具套件.

---

回忆一下操作系统的知识: init system 是操作系统启动时运行的第一个进程(PID=1)

---

这么看来 Systemd 似乎很厉害, 一个软件能干这么多事情. 但这也违背了 Unix 哲学: 一个工具只做一件事. Unix 原教旨主义者普遍讨厌 Systemd All-in-One 的设计. 但我觉得吧, 在系统编程的层面, 死扣着独立设计/模块化设计的哲学也没必要, 有时候需要一些更灵活的手段. 包括 Linux 内核自己都不是所谓的微内核.

事实上, Systemd 的效率和规范化远胜于它的前任 SysVinit, 也比常见的 cron 配置文件好写得多, 反正我觉得用起来挺顺手的.

但对于系统开发者来说, Systemd 的设计是有问题的. 它在 kernel 和 user 之间插入了一层 system, 但它也没有明确地定义 system 的边界, 即 system 应该做什么, 不应该做什么. 这就使得, 如果你对 Systemd 的某个功能模块不满意, 想自己写一个更好的, 那么你要么自己实现一个完整的 Systemd 来代替掉它创建的 system 层, 要么你就只能用 Systemd, 并在它的源码上进行有限的修改. 而且大而全的 Systemd 没有保留 kernel 和 user 的直通接口, 这可能会影响到用户对系统开发的可定制性.

---

而且 Systemd 的开发者曾经开发过 pulseaudio, 是一个 bug 一堆的音频控制工具, 不如 pipewire 一根. 所以 Systemd 的代码质量是不如 Linux 内核的.

---

但是管他呢, 我又不是系统程序员, 用 Systemd 就用吧.

## Tragedy of GNU

与其说是 Tragedy of GNU, 不如说是 Tragedy of Free Software.

自由软件的兴起, 发展, 衰落我懒得写了, 网上讲自由软件运动的视频有很多. 曾经轰轰烈烈的自由软件运动已经被各大商业公司用 偷梁换柱了. 现在的热词是 软件, 但实际上, 开源软件和自由软件是两个概念. 具体参考[开源错失了自由软件的重点](#).

如果你也想成为一名**自由软件**支持者, 一个最简单的步骤就是, 在你发布你的软件和源代码时, 采用最新的 GPL 系列 License. 比如 GPL-v3 和 AGPL-v3.

## Tragedy of Linux

Linux 社区采用的开发和管理方式是仁慈君主独裁制, 即所有的 code review 都由 Linus 本人最终负责. 然而可惜的是, Linus 本人并不是一个自由软件者, 尽管 Linux 前被冠上了 GNU 的名号. 当时 GNU 和 Linux 的合作, 只是迫于 UNIX 的巨大压力下而达成的. Linus 本人并不是多么在意自由软件的理念, 比起理念, 他更像是一个实用主义者. 尽管 GNU 推出了最新版的 GPL-v3 协议和 AGPL-v3 协议, 进一步确保了软件和代码的**自由**, 并且自由软件创始人理查德·斯托曼(Richard Stallman)也曾多次建议 Linus 将 Linux 的协议从 GPL-v2 升级到 GPL-v3, 但都被 Linus 无视了.

在 2024 年 10 月 18 日, Linux 社区发生了一件足以被钉在历史耻辱柱上的事: Linus 未经社区审议和正常流程, 直接将若干个来自俄罗斯的内核模块维护者移除维护名单. 并且面对社区的质疑, Linus 本人发邮件回复, 表达了自己对俄罗斯国籍的敌意.

可悲可叹, 开源社区的精神支柱 Linus, 亲手打碎了大家树立起来的神像. 这一刻, Linus 以前爆过的所有的典, 都化成了巨大的回旋镖打在了自己脑门上.

---

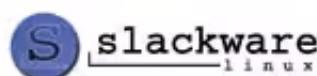
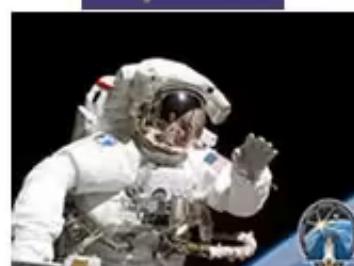
Linus: code is cheap, show me your nationality.

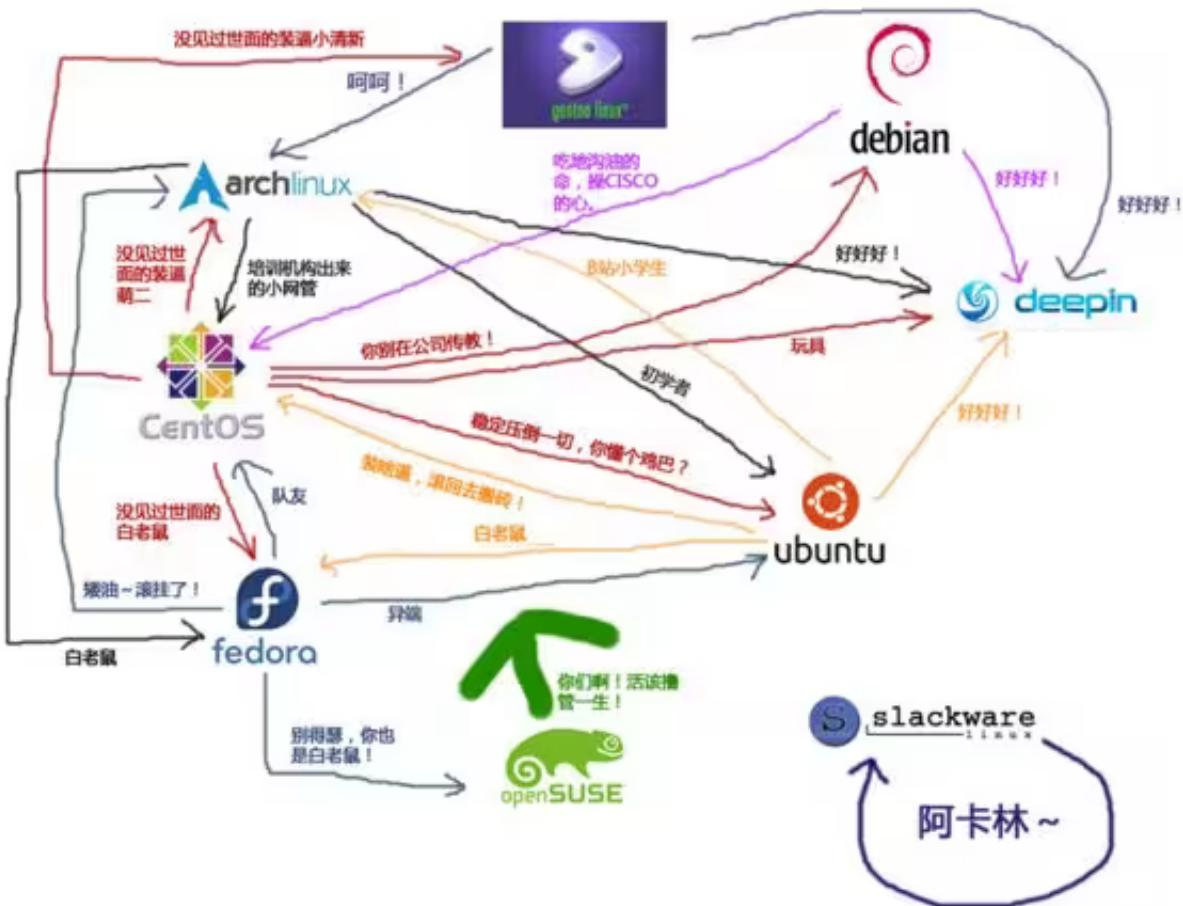
---

## Linux 发行版及其刻板印象

Linux 发行版(Distro)是基于 Linux 内核开发出的完整操作系统. 基于不同的设计思想和理念, 以及不同的应用场景, Linux 社区衍生出了众多的发行版. 这里我首先列举出**自由的 GNU/Linux 发行版**. 一个令人悲伤的事实是, 考虑到各大商业公司的驱动, 固件等都不是自由的, 为了能满足日常流畅使用, 主流的 Linux 发行版不得不包含这些非自由的固件, 因而丧失了自己自由的性质.

# 我眼中的各Linux发行版用户





刻板印象就图一乐，认真你就输了. 😂

## Debian 系列

- Debian: 正统发行版, 曾经一度是坚定的 GNU/Linux, 可惜还是在现实的重压下低头了. 因为其标志特别像雌二醇包装盒上的标志, 因而被认为是男娘系统. 如果你在用 Debian, 那记得加我 QQ, 我喜欢香香软软的小男娘. 😊







- Ubuntu: 声量最大的 Linux 发行版, 曾经一度让小白以为 Ubuntu == Linux, 可能也是很多小白的第一款 Linux. 由商业公司 Canonical 开发并维护, 塞满了公司的私货(比如 snap 包管理器)以及一些神奇的政治倾向. Ubuntu 系统饱受诟病的一点就在于它十分不稳定, 动不动就给你弹一个报错. 如果你在用 Ubuntu, 那就别用了, 换个发行版吧.
- Deepin: 国产的操作系统. 很抽象, , , 这两个词竟然能凑在一起. 我只简单地尝试过, 鉴定为比 Ubuntu 还不如的东西. 如果你在用 Deepin, 那我相信你也一定在用鸿蒙.
- NixOS: 一种很新的东西, 自己重新搞了一套独立的包管理系统, 采用函数式的声明来配置整个系统. 我只是简单地用过, 懒得学习 Nix 语言, 就没用了.

## Red Hat 系列

- Red Hat Enterprise Linux(RHEL): 红帽企业版 Linux, 红帽公司推出的商业 Linux 发行版, 专注于企业商用.
- Fedora: 红帽公司推出的社区版 Linux, 专注于个人开发者. Red Hat 会将 Fedora 作为新特性的试验田, 当特性成熟稳定后会进入到 RHEL 中. 所以 Fedora 算是 RHEL 的上游.
- CentOS: 已经死掉了的 Linux 发行版, 本来是作为社区版的 RHEL 在运行的, 结果被 Red Hat 收购之后就成了 RHEL 的上游去了, 换言之, 企业商用所追求的稳定性和安全性就没有了. 如果你在用 CentOS, 那你应该是买了国内老掉牙的 Linux 入门书籍. 国内企业也是用 CentOS 居多. 很符合我对国内的刻板印象.
- OpenEULER: 华为推出的 RHEL 衍生版, 仅在做数据库实验时用过, 臭不可闻. 如果你在用 OpenEULER... 🤦‍♂️ 😅

## Arch 系列

- Arch Linux: 我的日常操作系统, 只有你用了才知道它的好. 如果你也在用 Arch Linux, 那太棒了, 你一定是和我一样的小男娘, 快来加个 QQ 吧! 😊



- Manjaro: 你是? 都用这个了, 为什么不直接一步到位用 Arch Linux 呢?
- SteamOS: 惊不惊喜, 意不意外? SteamOS 其实是 Arch Linux 的衍生版. 如果你玩 Steam Deck, 那你已经在不知不觉间用上了 Linux 了!

## Gentoo 系列

- Gentoo: 我没用过, 我不知道.

## 安装 Linux

前面我已经简单介绍了几个知名的 Linux 发行版, 相信你也选择出了你想要安装的 Linux 发行版. 我正在使用的是 Arch Linux, 我强烈推荐 Arch Linux, 后文的所有内容我也会基于 Arch Linux 撰写.

想要安装 Arch Linux, 有若干方法可供选择:

-  做好文件备份, 然后把你的 Windows 丢到垃圾桶里去, 直接在物理机上安装 Arch Linux. (推荐)
-  购买/组装一台新电脑, 然后安装 Arch Linux. (推荐)
- 考虑到你正在使用 Windows, 可以在 WSL2 中安装 WSL2-Arch. (推荐)
- 在 Windows 上安装虚拟机, 然后安装 Arch Linux.
- 使用双系统, 在已有 Windows 的基础上安装 Arch Linux.

## WSL2-Arch

参考[这篇文章](#)

## 完整地安装 Arch Linux

### 准备工作

首先从[ISO镜像源](#)找到最新的 ISO 镜像文件, 下载到本地.

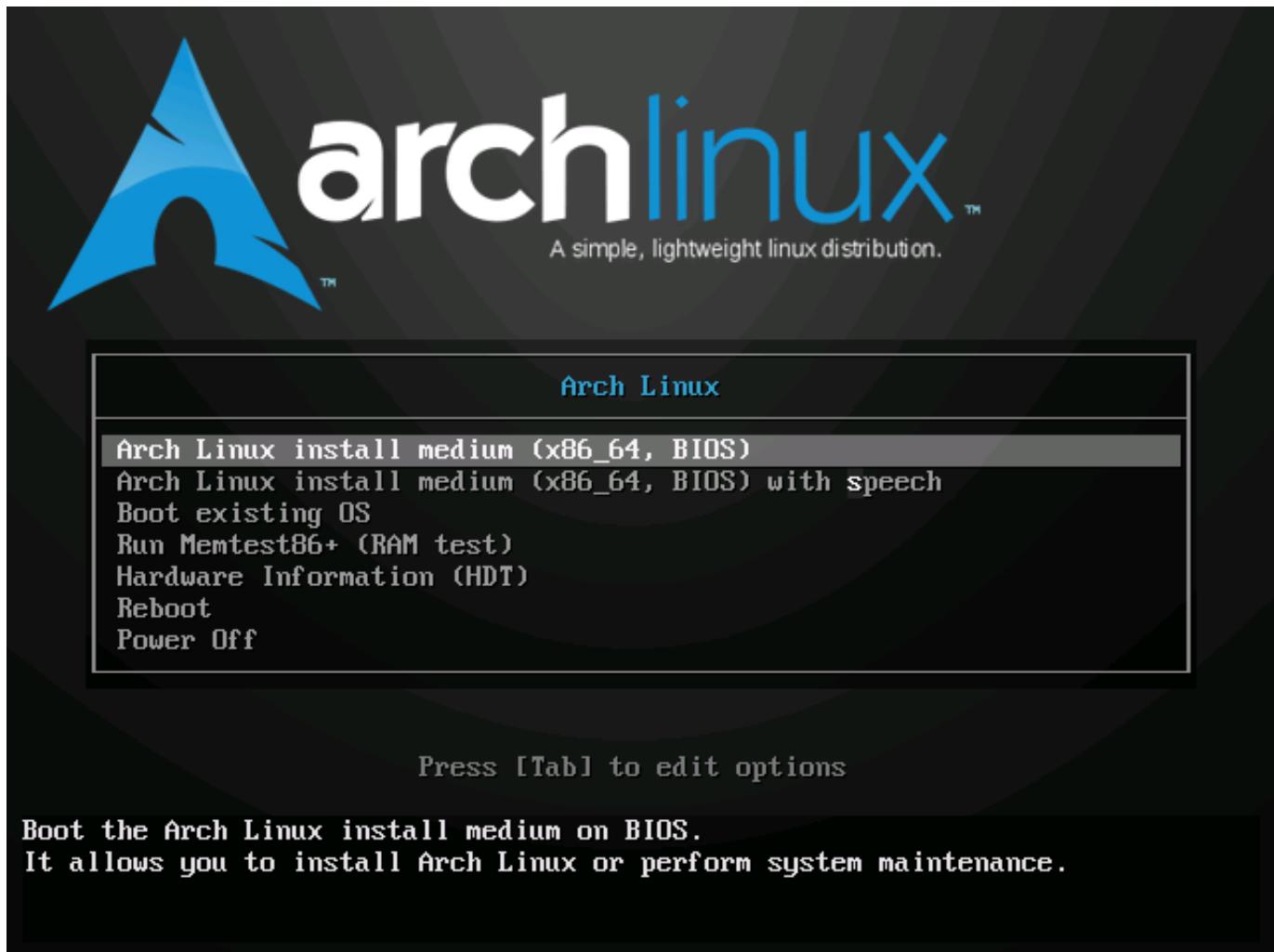
---

Arch Linux 采用滚动更新的策略, 所以不会像 Ubuntu 一样有特定的版本号. Arch Linux 会每隔一段时间创建一个最新的系统快照, 作为 ISO 镜像文件使用.

---

然后准备一个将这个镜像烧入到你准备好的启动U盘中. Windows 下可选 [Rufus](#), Linux 下可选 [balenaEtcher](#).

烧录完成后, 将电脑从启动U盘启动, 即可进入 Arch Linux 安装界面.



注意, 如果是笔记本的话, 记得先在 BIOS 中关闭安全启动, 不然会无法安装.

选择第一项 **Arch Linux install medium**, 按回车键即可进入安装.

在一阵炫酷的文字滚动后, 你会进入如下界面:

```
Arch Linux 6.12.7-arch1-1 (tty1)
archiso login: root (automatic login)

To install Arch Linux follow the installation guide:
https://wiki.archlinux.org/title/Installation_guide

For Wi-Fi, authenticate to the wireless network using the iwctl utility.
For mobile broadband (WWAN) modems, connect with the mmcli utility.
Ethernet, WLAN and WWAN interfaces using DHCP should work automatically.

After connecting to the internet, the installation guide can be accessed
via the convenience script Installation_guide.

root@archiso ~ # _
```

现在我们已经成功进入了 Arch Linux 的安装界面。这里我们将使用 `archinstall` 来快速且方便地安装 Arch Linux。

## 配置网络

如果你已经通过有线网络连接到互联网，那就不用额外配置网络连接。如果你的笔记本没有无线网口，那么就要使用 `iwctl` 命令来连接无线网络。

首先输入 `iwctl`，然后按回车键，进入 `iwctl` 命令行界面。

然后输入 `device list`，列出所有的无线网卡。这里假设你的无线网卡是 `<card-name>`。

然后用以下命令搜索可用的无线网络：

```
station <card-name> scan  
station <card-name> get-networks
```

搜索完成后, 找到你想要连接的网络, 假设为 `<wifi-name>`.

输入以下命令连接无线网络:

```
station <card-name> connect <wifi-name>
```

之后会提示你输入 WiFi 密码, 输入密码后即可连接到网络.

使用 `exit` 命令退出 `iwctl` 命令行界面.

现在你已经成功连接到网络了.

---

参考 [wiki](#)

---

## 换源

Arch Linux 默认采用的是国外的源, 会很慢. 所以推荐使用 `reflector` 进行换源:

```
reflector -c China --sort rate --latest 20 --verbose --save  
/etc/pacman.d/mirrorlist
```

命令解释:

- `reflector`: 是一个用来更新 pacman 源的工具.
- `-c`: 国家参数, 这里选择 China 内的镜像源
- `--sort`: 排序手段, 这里按照镜像源的下载速率从高到低排序
- `--latest`: 显示前 20 个镜像源
- `--verbose`: 显示详细信息
- `--save`: 将更新后的源列表保存到 `/etc/pacman.d/mirrorlist` 文件中

当命令完成后, 即可使用 `pacman -Syu` 命令更新系统软件.

## 安装系统

输入:

archinstall

然后按回车键, 进入安装程序:

```
Press ? for help

> Archinstall language      English (100%)
Locales                      +
Mirrors
Disk configuration
Swap                         +
Bootloader                   +
Unified kernel images
Hostname                     +
Root password
User account
Profile
Audio
Kernels                      +
Network configuration
Additional packages
Optional repositories
Timezone                     +
Automatic time sync (NTP)   +
Save configuration
Install
Abort
```

由于版本不同, 你看到的界面可能和我有所不同, 但需要配置的项目都是一样的. 让我们来逐一配置:

- Archinstall language: 不要动
- Locales: 不要动, 等装完了再改
- Mirrors: 不要动, 已经用 reflector 配置过了
- Disk configuration: 配置磁盘
  - Partitioning: 磁盘分区
    - Use a best-effort default partition layout: 使用默认分区方案, 建议选这个

- 选择你要安装的硬盘, 然后进入 Filesystem 选择:
  - btrfs: 推荐, 支持很多高效的特性
    - use BTRFS subvolumes with a default structure: 使用 BTRFS 子卷, 并使用默认结构, 建议选择 Yes
    - use compression or disable CoW: 使用压缩或禁用 CoW, 建议选择 Use compression
    - separate partition for /home: 分离 /home 目录, 建议选择 No
  - ext4: 老牌选择, 兼容性好
  - xfs/f2fs: 没用过, 不知道
- Manual partitioning: 手动分区
- Disk encryption: 不要动
- Swap: 不要动
- Bootloader:
  - systemd-boot: 推荐, 支持 UEFI.
  - grub: 兼容性和可配置性强. 随你.
- Hostname: 主机名, 取一个你喜欢的名字
- Root password: Root 用户密码
- User account: 配置普通用户
  - Add a user: 添加一个普通用户
    - Username: 用户名, 只能用小写字符
    - Password: 密码, 可以设置成和 Root password 一样
    - should <username> be a superuser (sudo)?: 是否授予 sudo 权限, 建议选择 Yes
  - Confirm and exit: 确认并退出用户配置
- Profile: 安装方案
  - Type: 安装类型
    - Desktop: 安装桌面环境, 有 GUI, 你可以选择若干桌面环境. 以下是我的选择(使用空格键勾选, 使用回车确认并退出)
      - KDE Plasma
      - Hyprland
      - Seat access: 权限认证工具, 我选择 polkit, 因为这是 KDE 内置的工具
    - Minimal: 最小化安装, 除了系统本体, 啥都没有
    - Server: 服务器安装, 会安装用于网络服务器的组件, 没有 GUI
    - Xorg: 不知道, 我没用过
    - Graphics driver: 当你选择 Desktop 选项后出现, 如果你是 N 卡用户, 则选择 Nvidia(proprietary); 否则不要动

- Greeter: 登录界面, 不要动

- Audio: 选 pipewire
- Kernels: 不要动
- Network configuration: 安装完成后新系统配置网络的方式, 如果你选择安装的 Desktop 中有 KDE Plasma 或者 GNOME, 则可以选择 Use NetworkManager
- Additional packages: 安装一些额外的软件包, 不要动, 可以等安装完了在新系统里自己装
- Optional repositories: 额外的软件仓库, 建议勾选 multilib, 因为里面有 Steam (笑)
- Timezone: 时区, 键入 /shanghai 即可跳转到 Asia/Shanghai 时区, 回车选择确认
- Automatic time sync: 自动校时, 不要动

当一切配置完成, 你可以选择下方的 Install 选项, 回车确认后, 此时系统会自动开始安装.

## 本地化

当 archinstall 安装完成后, 会提示是否 chroot 进入新系统. 选择 Yes 进入新系统终端, 输入命令:

```
vim /etc/locale.gen
```

找到并取消注释如下内容所在的行:

```
zh_CN.UTF-8
```

保存退出后用如下命令更新本地化设置:

```
locale-gen
```

## Arch Install: Odyssey

虽然使用 archinstall 工具安装 Arch Linux 非常方便快捷, 但我还是建议你至少按照 arch wiki 完整地手动走一遍安装流程, 这对你理解 Linux 系统的运行原理和结构有很大帮助.

# 配置 Arch Linux

在 KDE Plasma 桌面环境中, 快捷键 Crtl+Alt+T 可以快捷打开终端.

GUI 的美化千千万, 你可以自己探索你喜欢的美化设置. 这里我将主要讲解终端的美化和配置.

## 更换系统字体和语言

你不能直接将系统语言更换为中文, 因为此时系统缺少中文字体. 使用如下命令安装常用字体:

```
sudo pacman -S ttf-hack-nerd noto-fonts-cjk
```

然后在系统设置中, 先更改字体为 Noto Sans CJK SC, 再更改语言为简体中文.

## 添加第三方源

Arch Linux 有许多有用第三方源, 可以让你不用翻墙就能安装一些好用的软件. 使用 vim 打开 /etc/pacman.conf, 在末尾加入以下内容:

```
[archlinuxcn]
SigLevel = Optional TrustAll
Server = https://mirrors.cernet.edu.cn/archlinuxcn/$arch
```

```
[arch4edu]
SigLevel = Optional TrustAll
Server = https://mirrors.cernet.edu.cn/arch4edu/$arch
```

保存退出后, 使用 sudo pacman -Syu 更新软件源.

## 安装 AUR 助手 yay

yay<sup>archlinux-cn</sup> 是 Arch Linux 的 AUR 助手, 它可以帮助你管理 AUR 软件包, 并自动编译安装. 从 archlinuxcn 仓库安装 yay:

```
sudo pacman -S yay
```

需要导入密钥时, 选择同意导入.

## 安装常用软件

可以参考[安装脚本](#)中的内容, 自助选择需要安装的软件.

## 输入法设置

使用Fcitx5作为输入法:

```
sudo pacman -S fcitx5-im fcitx5-chinese-addons
```

然后在系统设置 > 虚拟键盘 中选择Fcitx5作为输入法.

## 终端美化

我的美化方案需要用到以下软件:

```
sudo pacman -S ttf-hack-nerd zsh tmux fzf fd bat eza tldr thefuck trash-cli  
atuin autojump starship
```

请先将你的终端字体切换到Hack Nerd Font, 否则无法显示某些符号.

## 切换默认终端

将默认终端切换为zsh:

```
sudo chsh -s /bin/zsh
```

## 安装oh-my-zsh

```
sh -c "$(curl -fsSL  
https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)"
```

## 安装主题和扩展

```
git clone --depth=1 https://github.com/romkatv/powerlevel10k.git ${ZSH_CUSTOM:-$HOME/.oh-my-zsh/custom}/themes/powerlevel10k
git clone https://github.com/zsh-users/zsh-autosuggestions ${ZSH_CUSTOM:-$HOME/.oh-my-zsh/custom}/plugins/zsh-autosuggestions
git clone https://github.com/zsh-users/zsh-syntax-highlighting.git ${ZSH_CUSTOM:-$HOME/.oh-my-zsh/custom}/plugins/zsh-syntax-highlighting
git clone https://github.com/wfxr/forgit.git ${ZSH_CUSTOM:-~/oh-my-zsh/custom}/plugins/forgit
mkdir ${ZSH_CUSTOM:-$HOME/.oh-my-zsh/custom}/plugins/incr
curl -fsSL https://raw.githubusercontent.com/Orion-zhen/incr-zsh/main/incr.zsh -o ${ZSH_CUSTOM:-$HOME/.oh-my-zsh/custom}/plugins/incr/incr.zsh
```

## 配置 .zshrc

.zshrc 文件位于你的用户目录下, 是控制终端行为的配置文件.

```
curl -fsSL https://raw.githubusercontent.com/Orion-zhen/dotfiles/main/.zshrc -o ~/.zshrc
```

现在重启电脑, 你应该可以看到一个漂亮的终端了.

# Linux 入门

## 包管理器

Linux 下, 安装软件都应该从包管理器安装. 这和 Windows 下需要上网找安装包然后自行安装的方式有很大的不同. 正是这个包管理器机制, 让 Linux 下的软件安装和更新变得如此简单.

一个内容重组的包管理器软件源就显得尤为重要. 这也正是 Arch Linux 的优势区间所在. Arch 官方源中本身就有非常丰富的软件, Arch User Repository (AUR) 则是由广大 Arch 用户提供的软件仓库. 你总能找到你想要的软件包.

## pacman

Pacman软件包管理器是 Arch Linux 的一大亮点. 它将一个简单的二进制包格式和易用的构建系统结合了起来. Pacman的目标是简化对软件包的管理, 无论软件包是来自官方软件仓库还是用户自己创建的软件包.

Pacman 通过和主服务器同步软件包列表来保持系统是最新的. 这种服务器/客户端模式可使得用户使用简单的命令, 就能下载或安装软件包, 并包含其所有必需的依赖包.

Pacman 用 C 语言编写, 并使用 bsdtar(1) tar 作为打包格式.

具体参考 [archwiki -> pacman](#)

## 软件源

可以向 `/etc/pacman.conf` 的末尾追加软件源. 例如前文已经向你展示了两个比较常用的软件源: `archlinuxcn` 和 `arch4edu`. 这里再补充几个好用的软件源:

- `chaotic-aur`: 构建了很多 AUR 软件包
- `alerque`: 提供了很多字体
- `our`: 我自己构建的软件源, 包含了诸如 QQ, 微信, 腾讯会议, 钉钉等未被 `archlinuxcn` 和 `arch4edu` 收录的软件, 以及一些妙妙工具, 如果喜欢的话记得给个 star 哟 😊

以上提到的软件源可以如下导入:

```
# /etc/pacman.conf
[chaotic-aur]
SigLevel = Optional TrustAll
Server = https://geo-mirror.chaotic.cx/$repo/$arch

[alerque]
SigLevel = Optional TrustAll
Server = https://arch.alerque.com/$arch

[our]
SigLevel = Optional TrustAll
Server = https://orion-zhen.github.io/our/$arch
```

## AUR 助手

安装 AUR 软件总是很麻烦: 你要将 AUR 仓库克隆下来, 然后手动编译安装. 幸运的是, 有一个叫做 yay 的 AUR 助手可以自动化这一过程. 具体参考 [archwiki -> yay](#)

---

当然你也可以学我, 自己打包 AUR 软件并发布软件源 😊

---

## Linux 文件结构

和 Windows 不同, Linux 文件系统是树状结构, 并不会区分 C 盘, D 盘 云云. 使用 `cd` 命令切换到根目录下:

```
cd /
```

然后使用 `ls` 命令查看根目录下的文件夹:

```
ls
```

让我们来逐个检查这些文件夹:

- `/bin`: binary 的缩写. 存放着对操作系统至关重要的二进制文件/可执行文件. 在现代 Linux 系统中, 这个文件夹常被作为符号链接指向 `/usr/bin` 目录
- `/sbin`: system binary 的缩写. 存放着系统管理相关的二进制文件/可执行文件, 仅应该被 root 用户或 sudo 权限使用. 在现代 Linux 系统中, 这个文件夹常被作为符号链接指向 `/usr/bin` 目录
- `/lib`: library 的缩写. 存放着系统的共享库文件. 这些文件在系统启动时被加载到内存中, 并由各个程序共享. 在现代 Linux 系统中, 这个文件夹常被作为符号链接指向 `/usr/lib` 目录
- `/lib64`, `lib32`: `/lib` 的变体, 和 `lib` 指向同样的目录
- `/usr`: Unix System Resources 的缩写. 顾名思义, 即 Unix 系统资源目录, 包含了运行系统所需要的各种重要资源, 比如命令, 库, 字体, 文档等
- `/etc`: Editable Text Configuration 的缩写. 存放着系统的配置文件, 包括各种服务的配置文件, 登录脚本, 环境变量等. 所有的系统级配置文件都应该存放在这个目录下. 里面有许多以 `.conf` 结尾的文件, 其实都是文本文件. 应用程序们读取各自对应的配置文件, 并进行相应的配置
- `/home`: 用户目录所在. 不同的用户对应不同的文件夹, 例如用户名 `sample` 的用户目录就在 `/home/sample` 目录下. 在终端中, 用户目录常可以用 ~ 符号来访问, 也可以从环境变量

## \$HOME 中获取

- `/var` : Variable 的缩写. 存放着系统运行时产生的各种数据文件, 比如日志文件, 缓存文件, 临时文件等, 这些文件在系统运行时会被频繁地修改. 另外, 系统日志文件也会被存放在这个目录下
- `/boot` : 存放着启动相关的文件, 包括内核, 引导程序, 启动脚本等. 系统启动时, 内核会被加载到内存中, 然后启动脚本就会被执行, 引导系统进入操作系统
- `/dev` : Device 的缩写. Linux 的设计思想是"一切皆文件", 因此 Linux 系统中的设备也被看作文件. 这个目录存放着 Linux 系统中所有的设备文件, 包括块设备文件和字符设备文件. 块设备文件通常用来存放硬盘, 字符设备文件通常用来存放打印机, 网络设备等. 在这个目录中有一些有意思的设备文件:
  - `/dev/null` : 黑洞设备, 所有写入到这个设备的数据都会被丢弃
  - `/dev/zero` : 零设备, 所有写入到这个设备的数据都会被填充为 0
  - `/dev/urandom` : 伪随机设备, 产生伪随机数据
- `/opt` : Optional 的缩写. 存放着第三方软件的安装目录. 例如, 你使用官方包管理器安装了 cuda 驱动, 那么 cuda 的安装目录就会被存放在 `/opt/cuda` 目录下
- `/tmp` : Temporary 的缩写. 存放着临时文件, 一般是应用程序运行时产生的临时文件. 系统重启后, 这个目录下的文件都会被删除
- `/proc` : Process 的缩写. 存放着系统的运行信息, 例如系统的内核信息, 进程列表, 网络连接信息等. 这个目录的内容是动态储存在内存中的, 并不占用磁盘空间
- `/mnt` : Mount 的缩写. 存放着临时挂载点, 即将外部设备挂载到系统的文件系统. 你可以使用 `mount` 命令将不同的设备挂载到系统中的任何位置. 不过一般都是临时挂载在这个目录下:  
`sudo mount /dev/sda1 /mnt`

### Note

有时候你可能会想, 既然已经有 `/usr/bin` 和 `/usr/lib` 目录了, 为什么还要创建两个符号链接呢? 这其实也是历史遗留问题, 早期的 Linux 文件结构还没有这么清楚, 当时光是 `bin` 目录就区分了 `/bin`, `/sbin`, `/usr/bin`, `/usr/local/bin` 等等. 后来 Linux 社区意识到这个问题, 于是就把这些目录都合并到 `/usr` 目录下, 并创建了符号链接. 这样做的好处是, 系统中只需要维护一个目录, 而不需要维护多个目录.

### Note

关于 `/usr` 目录, 有人误认为它是 User 的缩写, 其实是错误的. 在 Mac 中, 用户目录都存放在 `/User` 下, 而 `/usr` 仍然存在.

### Note

小心 `/` 和 `./` 的区别, 前者是根目录, 后者是当前目录. 不要一不小心少打一个 `.`, 导致错误地对根目录进行了一些不妙的操作.

## Linux 文件

### 文件权限

使用 `ls -l` 命令查看文件的权限:

```
ls -l
```

- 第一个字符表示文件类型:
- 后面三位表示文件所属用户的权限
- 再后面三位表示文件所在的用户组的权限
- 最后三位表示其他用户的权限

对于权限位, 各位的含义依次如下:

- `r` : 可读, 数字为 4
- `w` : 可写, 数字为 2
- `x` : 可执行, 数字为 1
- `-` : 没有权限, 数字为 0

将这些权限求和, 即可得到不同的数字, 每个数字唯一地表示一种权限组合. 三个数字在一起, 即是完整的文件权限. 例如, `rw-r--r--` 就是 644, `rwxr-xr-x` 就是 755.

想要改变文件的权限, 可以使用 `chmod` 命令:

```
chmod <access-mode> [-R] file
```

其中, `<access-mode>` 是你想要设置的权限位, 例如 755 就是 `rwxr-xr-x`。`-R` 选项可以递归地设置目录下所有文件的权限。

如果仅想授予某个文件可执行的权限, 可以:

```
chmod +x file
```

---

我曾经的个性签名是: `sudo chmod 777 -R /world`, 你知道这是什么意思吗?

---

## 文件操作

### 创建和删除

- `touch`: 创建空文件
- `mkdir`: 创建目录
- `rm`: 删除文件, 选项 `-r` 可以递归删除目录, 选项 `-f` 可以强制删除
- `cp`: 复制文件或目录, 选项 `-r` 可以递归复制目录
- `mv`: 移动文件或目录

命令示例:

```
mkdir test
mkdir test2
touch test.txt
cp test.txt another-test.txt
mv test.txt test1.txt
mv test1.txt test/
cp -r test test2
rm another-test.txt
rm -r test
rm -rf test2
```

### 切换目录

- `pwd` : 显示当前目录
- `cd` : 切换目录

命令示例:

```
cd ~ # 切换到用户目录  
pwd
```

## 列出目录项

- `ls` : 列出目录内容
- `tree` : 递归列出目录内容

命令示例:

```
ls  
ls -a # 显示隐藏文件  
ls -l # 显示详细信息  
tree .
```

## 查看文件内容

- `cat` : 打印文件内容
- `less` : 逐页查看文件内容

命令示例:

```
cat test.txt  
less test.txt
```

## 查找文件和目录

- `find` : 查找文件或目录

命令格式:

```
find [options] <path> [expression]
```

| expression | 含义       | 示例   |
|------------|----------|--|
| -name      | 根据文件名查找  | -name "*.txt" 即寻找所有以 .txt 结尾的文件, * 是通配符                  |
| -type      | 根据文件类型查找 | -type f 即寻找普通文件, -type d 即寻找目录                           |
| -size      | 根据文件大小查找 | -size +10k 即寻找大于 10KB 的文件, -size -10k 即寻找小于 10KB 的文件     |
| -or        | 或        | -name "*.txt" -or -name "*.pdf" 即寻找所有以 .txt 或 .pdf 结尾的文件 |

## Linux 用户和用户组

### root 用户

你应该也注意到了, 之前运行 `pacman` 命令时, 总是要在前面加上 `sudo`, 这其实就是在以 `root` 用户的身份安装软件. 一般地, 根用户 (`root` 用户) 在 Linux 操作系统中拥有最高权限.

---

这里的最高权限是真的最高, 能完全地掌控整个电脑, 哪怕你要做危害系统的命令也行, 不像 Windows 和 MacOS, 它们的 权限根本不是系统最高权限. 最高权限被它们的公司牢牢抓在自己手里, 不肯分给用户半点.

---

既然 `root` 用户拥有最高权限, 那么它就可以对系统做任何动作, 包括你们可能早有耳闻的 `rm -rf /*` 命令. 因此, 你应该小心地使用 `root` 用户, 尤其是在重要的系统目录和文件上. `root` 用户的用户文件夹在 `/root` 下.

---

有人为了图方便, 就不创建普通用户, 平常就用 `root` 用户进行操作, 这样是非常危险的, 因为你不能保证你任何时候都不会失误. 而且有的命令反而要求不能以 `root` 身份运行.

### 系统用户

除了你, root 用户, 还有很多系统用户. 它们一般是由系统或者相关程序创建, 用于执行服务等系统任务. 例如当你安装了 ollama 后, 它将自动创建一个名为 ollama 的用户和用户组, 用来管理 ollama 程序的运行. 不要随意删除这些用户, 以免系统运行出现问题.

## 普通用户

就是你, 平时我们使用的普通用户.

## 用户组

可以将用户分组, 以便管理. 使用 `groups` 命令, 即可查看自己所属的用户组. 一个典型的例子是 `docker` 用户组. 当你安装好 `docker` 后, 它会自动创建一个名为 `docker` 的用户组, 而一般情况下你不在这个组里, 所以你必须使用 `sudo` 才能运行 `docker` 命令. 而你可以把自己加入 `docker` 用户组:

```
sudo usermod -aG docker $USER
```

这样你就可以在不使用 `sudo` 的情况下运行 `docker` 命令了.

## 在 Linux 上编程

或许你们已经习惯在 IDE 上写代码, 然后按一个按钮, 代码就能自动地跑起来的体验了. Visual Studio, Clion, PyCharm, IntelliJ IDEA 等 IDE 给你们在 Windows 下提供了非常舒服的开发体验, 因为 Windows 系统并不能在系统级为你们提供编译器和解释器, 所以需要 IDE 中集成开发环境来帮你编译和运行代码.

---

让我看看还有谁在用 DevC++

---

但你们在运行代码的时候是否心中有一些隐忧? 这个代码是怎么跑起来的? 我的开发是不是已经被局限在 IDE 中了? 离开了 IDE, 我还有什么办法能让我的代码跑起来呢?

有的兄弟, 有的.

Linux 下提供了系统级的编译器和解释器, 我们得以脱离 IDE 的温柔乡, 将 IDE 运行代码的环节拆开, 暴露在你的面前.

## 编辑器+编译器/解释器

我们可以粗略地将开发环节分成两部分: 在编辑器里编辑代码, 然后将编辑好的代码交由编译器或者解释器运行. 编辑器就是我们敲代码的地方, 而编译器/解释器则是将代码翻译成机器语言的工具.

在很久以前, 人们使用 IDE 而不是编辑器+编译器的一大理由是, 编辑器没有代码补全和语法高亮等实用功能, 而 IDE 则提供了这些功能. 但随着技术的发展, LSP (Language Server Protocol) 横空出世, 使得编辑器可以和编译器/解释器沟通, 获得更好的代码补全和语法高亮等功能.

一个好消息是, VSCode 自带了 LSP, 你可以非常方便地在 VSCode 上体验到强大的语法高亮和代码补全功能.

接下来我将以 C/C++ 和 Python 为例, 讲解如何在 Linux 上搭建开发环境.

## C/C++

所有的 Linux 发行版都自带了 GCC (GNU Compiler Collection) 编译器. 你可以通过如下命令来检查你的 gcc 编译器:

```
gcc --version
```

接下来我们创建一个文件夹, 在里面新建一个 `hello.cpp` 文件:

```
mkdir cpp  
cd cpp  
touch hello.cpp
```

编辑 `hello.cpp` 文件, 输入以下代码:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

然后, 在命令行中运行如下命令:

```
gcc hello.cpp -o hello
./hello
```

你应该会看到屏幕上输出 Hello, world! .

## 构建工具

上面是一个非常简单的示例, 只涉及到单文件编译和运行. 实际开发中, 我们面对的情况比这个复杂得多. 我自己常用的构建工具是 CMake, 它可以自动地生成 Makefile, 并根据你的代码生成对应的可执行文件.

至于 CMake 和 Makefile 的教程, 这里就不展开了, 你们可以自行搜索入门.

---

我在讲 Linux 下使用 VSCode 的时候提到过 CMake 的入门教程

---

## Python

Linux 下也自带 Python 解释器, 你可以通过如下命令来检查你的 Python 版本:

```
python --version
```

运行 Python 代码就更简单了:

```
# hello.py
print("Hello, world!")
```

```
python hello.py
```

## pip

pip 是 Python 的包管理工具, 你可以用它来安装第三方库:

```
pip install numpy  
pip install --upgrade pip # 升级 pip
```

## 依赖管理

如果你的项目有很多的依赖包, 手动一个一个一个地敲 `pip install` 命令显然不现实. 这时, 你可以使用 `requirements.txt` 文件来管理依赖:

```
numpy  
matplotlib  
torch
```

在这个依赖文件中, 每个依赖项都单独占一行. 然后, 你可以运行如下命令来安装所有依赖:

```
pip install -r requirements.txt
```

## 虚拟环境

一般地, 你应该会发现你无法在全局环境下使用 `pip` 安装依赖包, 这是因为 PEP 668 规定了, 全局环境不能用 `pip` 来安装依赖包, 以免全局环境被各种不同的依赖项弄得乱七八糟. 想要往全局 Python 中安装依赖包, 需要从系统的包管理器中找到对应的包. 例如 `numpy` 对应的在包管理器中的名称是 `python-numpy`, 而 `pip` 则是 `python-pip`.

想要用 `pip` 安装依赖包, 并且保持你的 Python 环境干净整洁, 你可以使用虚拟环境来隔离不同的项目依赖. 要创建一个虚拟环境, 可以通过 Python 内置的模块:

```
python -m venv .venv
```

这会在当前目录中创建一个名为 `.venv` 的目录, 这个目录中存放着你刚刚创建的虚拟环境. 想要进

入这个虚拟环境, 只需要:

```
source .venv/bin/activate
```

这时, 你应该会看到命令行提示符变成了 (`venv`) 这样的形式, 表明你已经进入了虚拟环境. 退出虚拟环境, 你只需要运行:

```
deactivate
```

在虚拟环境下, 你可以任意地安装依赖, 而不用担心把全局的默认环境搞乱. 要是一不小心你把虚拟环境弄脏了, 没关系, 直接删掉重建一个就好了!

### Note

一般建议对每个项目, 都在项目根目录下创建一个虚拟环境, 这样可以避免不同项目之间依赖的冲突.

## Python 版本

一般地, 系统中都会默认提供一个 Python, 比如 Arch 总是会给你提供最新版的 Python. 可是如果我有一个项目, 必须要老版本的 Python 怎么办?

这时候就体现出 Arch 的优越来了, 你可以在 AUR 中直接搜索老版本的 Python, 并直接安装:

```
yay python 3.10
```

这样, 你的系统中就有了老版本的 Python 了. 要使用老版本创建虚拟环境, 和正常创建差不多:

```
python3.10 -m venv .venv310
```

---

当然你要是不嫌麻烦的话可以装其他的虚拟环境管理工具, 里面通常会提供 Python 版本切换的功能. 但我非常不喜欢这样, 因为通过系统包管理器和 Python 自带的功能就可以实现的要求, 为什么还要引入额外的软件去做呢? 更何况诸如 anaconda 之类的东西体积太大了.

---

# 美化 Linux

我并不打算在这里教你如何美化你的 Linux, 因为这是一个非常复杂的主题, 而且每个人都有自己的审美观. 如果你不知道从何下手, 可以先在网上搜一搜别人的配置.

我加上这一节的目的仅在于告诉你, Linux 的界面绝对不是像你刚刚接触的那样简陋 -- 那只是个毛坯房 -- 你要用自己的双手将你的 Linux ~~调教~~ 美化成你的形状. 相信我, 配置美化的过程和结果一样令人着迷.

---

以上就是 A Byte of Linux 的全部内容, 感谢你能读到这里. 你已经成功地迈出了走向 Linux 的第一步, 证明你是一个有勇气的探索者. Linux 就像是一个巨大的瑞士军刀, 不仅功能强大, 还能让你在使用过程中不断地发现惊喜.

我知道, 刚开始接触 Linux 的时候, 你会感到陌生, 感到不适应, 感到它高冷且难以亲近. 我当然知道, 因为我也是这样过来的. 但请你相信, 你已经跨过了最艰难的一步: 开始使用 Linux. 随着你使用 Linux 的时间越来越长, 你会逐渐地熟悉上 Linux, 熟悉命令行的操作逻辑, 熟悉各种开发工具, 熟悉那些看似复杂却强大无比的 shell 语言.

所以不要放弃, 慢慢来.

我希望你能继续保持这份对技术的好奇和热爱. Linux 不仅仅是一个操作系统, 它更是一扇通往无限可能的大门. 跨国巨企的服务器运行在 Linux 上, 技术前沿的科研团队使用 Linux 开发, 数以亿计的开发者们在 Linux 社区里分享自己的经验. 无论是编程, 数据科学, 还是系统管理, Linux 都是你最可靠的伙伴. 期待有一天, 你能用 Linux 完成一个你自己都为之骄傲的项目.

最后, 送给你一句 Unix 世界里的经典问候:

Hello, world!

Hello, Linux!

---

© 2025. ICS Team. All rights reserved.

# VSCODE? Vim!

## 编辑器圣战

重生于5202年，你踏足这颗蔚蓝的星球

在这片浩渺的宇宙中，依然流传着两大神器的传说：Vim，传闻它是神之编辑器；而VSCode，被誉为编辑器之神。

追求独步天下的英豪，或是尚显稚嫩的新手，纷纷朝圣前往，想一探Vim的真容。可当他们看到那简朴而素净的界面时，心中难免生出疑问：这真的是神器吗？甚至有些人，心头泛起了轻视之情。

有的人大声讥讽：“什么年代了，居然还抱着这陈旧的工具不放？”

有的则无端评论：“Vim不过是时代的尘埃，新时代必然要革新而废弃！”

更有甚者，举起了写满“打倒Vim”的大字报。鲁镇街头，烟花绽放，锣鼓喧天，声势浩大…

就在这时，一位白须长者从众人中走出，他缓缓开口，声音低沉却有力：“所有人都冷静片刻，听我一言！它们虽然古老，已历百年风霜，但你们可曾思考过，为什么这古老的编辑器，依然有如此众多的人为之归依？”

鲁镇的百姓开始了沉思。

经过长久的思索，有人勇敢地尝试了Vim，甚至Emacs，却因那陡峭的学习曲线，几乎难以承受。面对界面的生疏，退出命令的繁琐，许多人选择放弃，感叹：“这也太难用了，连界面都不知如何退出，何谈其他！”

然而，依旧有一部分人没有放弃，他们坚信着这一切的背后必定有着不为人知的力量…

这场编辑器的圣战，或许还远未结束…



## 一份适合纯新手入门的Vim教程

### Note

我们默认这份指南的面向群体是：连 :wq 保存退出都不知道的纯新手用户

在这份教程里，我们一反网上各类博客/教程的顺序，即：Vim工作原理优先，按需增量加配置在后。

我承认这是本源上的学习步骤，并没有问题，这才是对的 ✓

但是它不适合新手入门！ ! ! !

对于第一次接触Vim的群体而言，我认为他们最需要的是在短时间内，尽可能多的感受到Vim作为“文本编辑器”的使用方式，而不是设计原理/配置方式等太过底层的东西。

因此，这份教程的顺序是：

1. 给出一份“标准”的vim配置模板，开箱即用，无需思考
2. 学生自行安装（安装方式等见后文）
3. 按教程尝试如何使用Vim写代码，快速入门

- 快捷键
- 不同模式切换
- ...

4. 待学生熟悉Vim基本使用方式后，讲解部分原理和模板配置

## 配置文件

~/.vimrc 文件内容，你可以直接在我的仓库复制：

## 如何配置

在你的命令行中，敲入：

```
vim ~/.vimrc
```

复制上述文件内容， :wq 保存退出。

然后再进入vim， 输入 :PluginInstall， 等待插件安装完成即可（如有提醒直接按 Enter 即可）。

此时你打开一份正常的 .cpp 代码，应该可以看见界面长这样：

```

1 #include<iostream>
2 using namespace std;
3
4 const int maxn = 1e6+10;
5 int n, q, k;
6 int a[maxn];
7
8 int ope_lfs(int l, int r, int ans)
9 {
10    // 找到左边界
11    // 第一个 >= ans的数的index
12    // 课上的绿模型
13    while(l < r)
14    {
15        int mid = (l+r) >> 1; // TBD
16
17        if (a[mid] < ans){
18            l = mid + 1;
19        }
20        else {
21            r = mid;
22        }
23    }
24
25    if(a[l] != ans) {
26        return -1;
27    }
28    else {
29        return l;
30    }
31 }
32
33 int ope_rfs(int l, int r, int ans)
34 {
35    // 找到右边界
36    // 最后一个 <= ans的数 的index
37    // 课上的红模型
38    while(l < r)
39    {
40        int mid = (l+r+1) >> 1; // TBD
41
42        if (a[mid] > ans){
43            r = mid - 1;
44        }
45        else {
46            l = mid;
47        }
48    }
49
50    if(a[r] != ans) {
51        return -1;
52    }
53    else {
54        return r;
55    }
56 }
57
58 int main()
59 {
60     int n, q, k;
61     cin >> n >> q >> k;
62
63     a[0] = 0;
64     for(int i = 1; i <= n; i++)
65         a[i] = a[i-1] + rand() % k;
66
67     for(int i = 0; i < q; i++)
68     {
69         int l, r, ans;
70         cin >> l >> r >> ans;
71
72         cout << ope_lfs(l, r, ans) << endl;
73         cout << ope_rfs(l, r, ans) << endl;
74     }
75
76     return 0;
77 }

```

现在就可以直接使用了

## 常用指法

---

免责申明：下面的部分指法只适用于笔者的vimrc配置，也就是说，如果你严格遵循上述步骤，那就ok；但是你要有自己额外的个性化配置，那就不保证了

---

## Note

现在使用我的vimrc, 你会发现有几个最方便的点 :

1. 可以用鼠标/触控板进行光标移动
2. 复制/粘贴可以直接用鼠标, 且对全文进行了快捷键定义 (空格 + a)
3. 默认 tab 缩进是 4

其实已经失去了vim的灵魂了

像语法高亮/状态栏/搜索高亮之类的就不用说了, 肯定都是有的

# 基础操作

## Normal Mode

### 插入

```
i -- enter insert mode and begin inserting or deleting text  
a -- enter insert mode, one space after cursor position  
<escape> -- enter normal mode
```

### 保存退出

```
<:q!> -- quit without writing  
<:wq> -- write and quit
```

### 移动

```
<Up/Down/Left/Right> -- 方向键
```

```
number + <Up/Down/Left/Right> -- 向上/下/左/右移动几number格
```

## 词单元

```
<w> -- next word  
<b> -- beginning of word  
<e> -- end of word  
<0> (zero) -- move to beginning of line  
<$> -- move to end of line  
<^> -- first non-null part of the line
```

我已经将 `<0>` 和 `<$>` 重定向成 `shift -` 和 `shift +` 了，很明显我的开箱即用教程已经最大程度地减轻需要记忆的负担 :)

```
<number w,b> -- eg: <4w> - moves forward 4 words
```

## 跳转

```
<G> -- go to end of file  
<gg> -- go to beginning of file  
  
<ctrl u> -- scroll up (half a page)  
<ctrl d> -- scroll down (half a page)
```

## 查找

```
< /search_item > -- searches for all occurrences in the file  
<n> -- jumps to the next occurrence  
<N> -- jumps to the previous occurrence
```

## 删除

```
dd -- delete this line  
cc -- delete this line and into `Insert` mode
```

## 撤销和回退

```
<u> -- undo edit  
<ctrl r> -- redo edit
```

## 复制和粘贴

<yy> -- yanks(or copies) current line  
<p> -- pastes copied item  
y5<Right> -- 复制右边的5个字符

<space>a -- 复制全文进入系统粘贴板

## 注释/解注释

我是用 [NerdCommenter](#) 做的

并且将快捷键全部改成了 <shift> /

- 行注释/解注释: 来到对应行, 使用 <shift> / 即可
- 段注释: v 进入visual模式, 选中所需区域, 使用 <shift> / 即可

## 分屏

---

s + 方向键

---

s <Right> -- 向右分屏  
s <Left> -- 向左分屏  
s <Up> -- 向上分屏  
s <Down> -- 向下分屏

## 分屏时切换光标的区域

---

q + 方向键

---

q <Right> <C-w>l  
q <Left> <C-w>h  
q <Up> <C-w>k  
q <Down> <C-w>j

## 在vim打开的file中执行终端命令

`:!<command>` -- eg: `:!ls` 就会在终端中执行命令

## Visual Mode

`<V>` -- enter multi-lined visual line mode  
`<v>` -- enter single-lined visual mode

Actions:

`<y>` -- copy current item  
`<d>` -- delete current item -> automatically goes to clipboard, so you can `<p>` to paste it  
`<escape>` -- go back to normal mode

## 彩蛋

### 显示文件系统结构

`ff` -- 按下`ff`就会自动显示文件系统结构, 鼠标点击即可打开细节

### 代码分层显示

`T` -- 按下 `shift + t` 会显示代码中的Tag, 分好层级

显示效果应该如下：

The screenshot shows a terminal window with a dark background. On the left, there's a file tree (NERDTree) listing various C files and directories under the path /home/cc/cs61c/fa24/class/cs61c-akp/. The files include ex1\_hello.c, ex2\_pointers\_and\_function.c, ex3\_arrays.c, ex4\_pointer\_arithmetic.c, ex5\_strings.c, ex6\_strcpy.c, ex6\_strcpy\_fixed.c, and tools/1.c. The right side of the screen displays the content of the file 1.c, which contains a simple C program with functions add, main, and print\_message.

```
" Press ? for help
.. (up a dir)
</cs61c/fa24/class/cs61c-akp/
`- labs/
  `-- lab01/
    * ex1_hello.c
    * ex2_pointers_and_function.c
    * ex3_arrays.c
    * ex4_pointer_arithmetic.c
    * ex5_strings.c
    * ex6_strcpy.c
    * ex6_strcpy_fixed.c
    * tools/1.c

1 #include <stdio.h>
2
3 int add(int a, int b) {
4     return a + b;
5 }
6
7 void print_message() {
8     printf("Hello, World!\n");
9 }
10
11 int main() {
12     int result = add(3, 4);
13     print_message();
14     printf("Result: %d\n", result);
15     return 0;
16 }
```

" Press <F1>, ? for help

functions

- add(int a, int b)
- main
- print\_message

## 一些别的指法

重新开一个 iterm2 窗口 (左右分屏)

<cmd + d> -- 水平分屏

重新开一个 iterm2 窗口 (上下分屏)

<cmd + shift + d> -- 垂直分屏

下面是一些mac常用的指法：

<control> + <d> 退出 (logout) , 常用于服务器/虚拟机退出

<control> + <l> 等价于清屏命令clear

<control> + <c> 暂停, programmer都知道

## 后续

事实上新时代Vim Family Tree里还有很多，比如 [neovim](#)。

这里提供一份开箱即用的版本，[lazynvim](#)，用过的都说好

不过笔者自己还是更喜欢“自配的vim”，毕竟这个更有geek风

这份简明教程就结束了，如果你有一些比较好的建议 or Vim插件推荐，plz feel free to report an issue [here](#) 

如果你想contribute more，尤其是想另写一篇《vim? neovim!》之类的，欢迎PR 

---

© 2025. ICS Team. All rights reserved.

# G\_hub? Github!

这篇文章会带你走近全球最大“同性交友网站”🌐🌐🌐

众所周知，GitHub是一个全球最大的代码托管平台，在这里我们借助Git来管理项目代码。

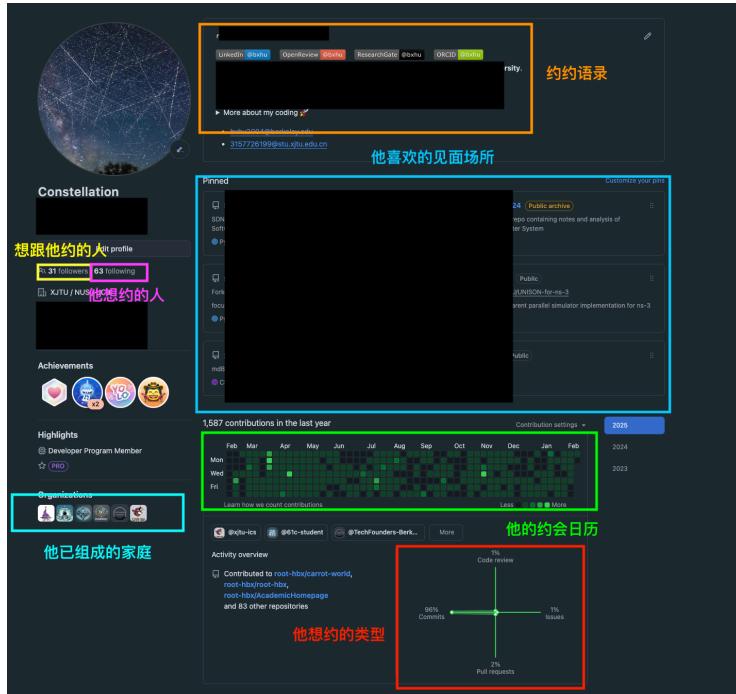
在GitHub上，我们可以很轻松的找到一些开源代码，因此，GitHub也有“程序员的维基百科”之称👀

经常上网冲浪的同学肯定会听说很多github的戏称，比如 Ga\_hub :))

那是因为GitHub上有95%的用户都是 😱

因此GitHub也被称为全球最大同性交友网站

这里拿一份某不知名人类高质量男性约会简历举例，你心动了吗❤️



心动不如行动 🐛

# 如何从零开始使用github

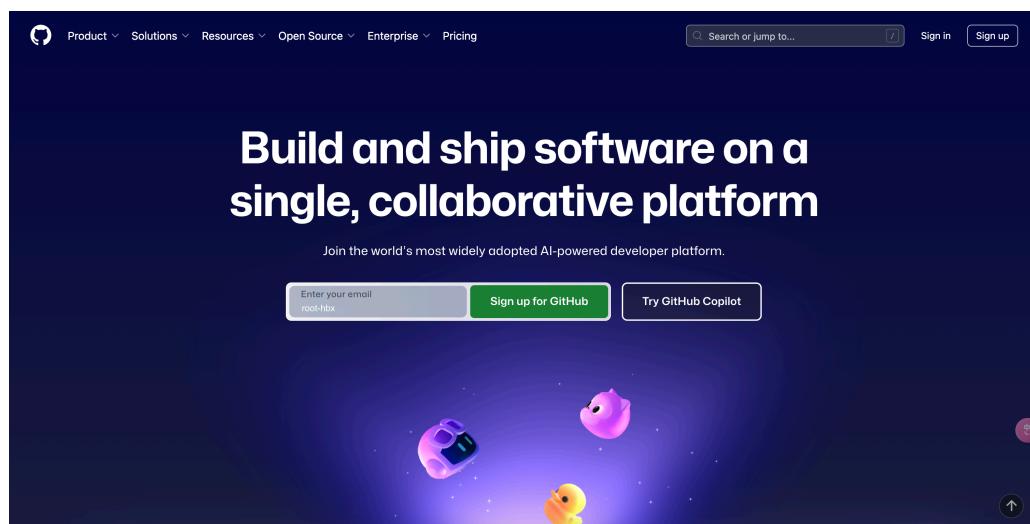
这篇文章会以一个刚刚开始学习的 Github 小白视角写的，希望能帮助到想开始学习 GitHub 又不知如何上手的学习者 😊

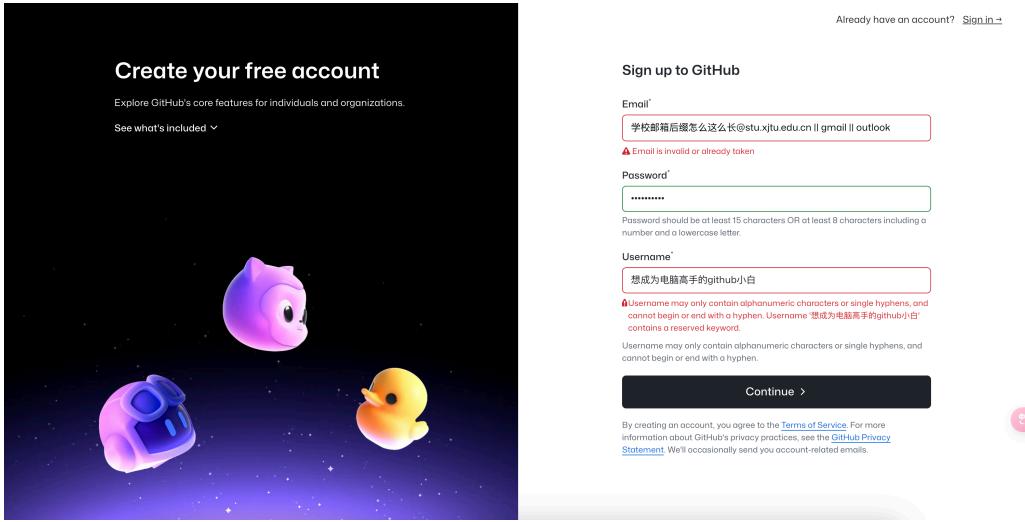
大家可能会对有以下困惑：

1. GitHub 是干什么用的？现在我只知道它跟代码有关 T\_T
2. 联想英文构词法，不难理解Github应该是跟Git相挂钩？但是这两个具体有什么关联吗？
3. 大家都说 GitHub 是个开源社区，可以在上面与其他人一起切磋技术、完善代码，那具体是怎么与别人交流呢？

## 注册账号并浏览初始页面

打开 GitHub 网站，首页如下图所示，点击右上角 **Sign up** 按钮，进入注册页面。使用电子邮件注册，设置好用户名和密码，即可生成账号。





完成注册后，点击旁边的 `Sign in` 登录即可

这时你会看见初始页面如下（肯定没有笔者这里展示的丰富，因为你刚注册，是新号）：

整个界面分成左中右三个部分，简单概括就是：

- 左：自己的仓库 / 自己近期的活动 / 自己的团队
- 中：关注的人(我想约的人)的动态
- 右：官方推荐，兴趣投喂，发现新的热门/有趣仓库

这个界面被叫做“公告栏” (Dashboard)

## 工具栏

点击右上角的个人头像，你会发现有很多栏目：

- Your Profile
- Your Repository
- Your Copilots
- Your Projects
- Your Stars
- Your Organizations
- ...

这个就是工具栏，是集很多功能显示于一体的“目录”，了解一下即可。我们会在后面逐个介绍主要功能的使用

## 个人主页 (Your Profile)

在之前的Dashboard中，点击右上角的个人头像，在下拉列表中选择 `Your profile` 项，到达个人主页。

主页面可以对比本文开头给出的“人类高质量男性约会简历”，了解一下 Github Profile 的使用方式：

他喜欢的见面场所

想跟他约的人 profile

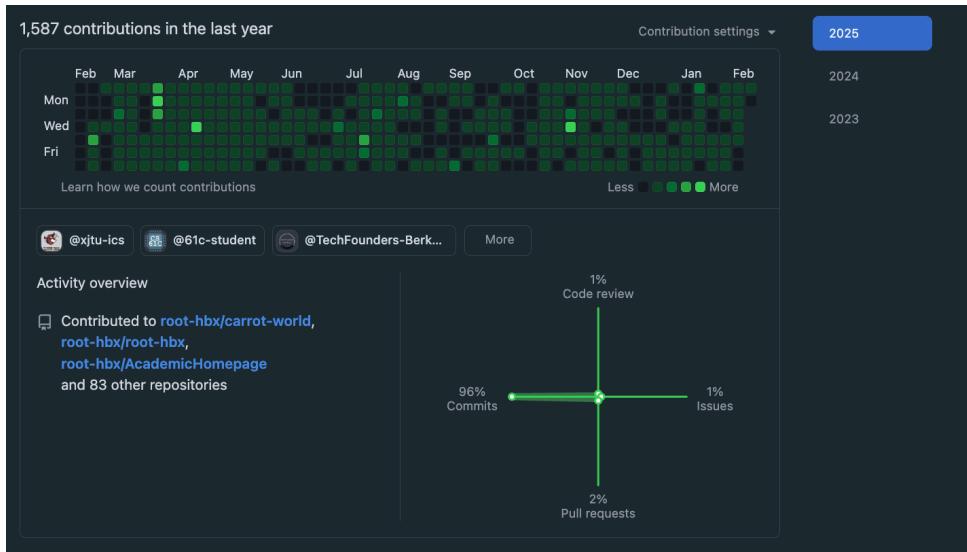
他想约的人

他已组成家庭

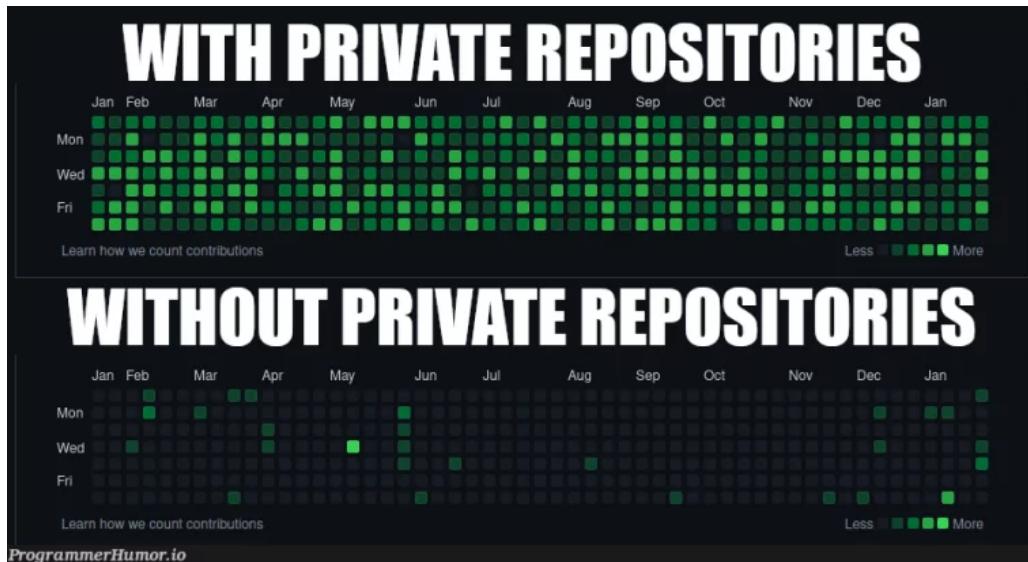
他想约的类型

他的约会日历

1. 点击头像下方的 `Edit profile` 按钮，可以修改自己的个人信息，比如这里的简介，公司、地址、联系方式等；而如果是别人的个人主页，这里看到的就是 `Follow`（关注）按钮和别人的展示信息 😊
2. "Pinned" 区域是自己的个人展示区，在自己的仓库中精选几个放到这里展示，让别人能很快发现你的闪光点 🌟
3. 活跃度表格：绿色格子越多，颜色越深，说明该用户在 GitHub 上提交次数越多，是重度网瘾患者 🚀
4. 雷达图：显示你/你看的用户的个人工作重心，是commit多，还是pr多...



每个提交对应于仓库的改动( public / private ), 因此很多人会看见这张图 :



还需要注意一个东西, 叫 follow

### Note

- Follower: 表示该用户的“粉丝”
- Following: 表示该用户的“偶像”

这里展示一下关于follow的梗图, followers的数目可以大致反映一个用户的“大佬程度”



1000  
instagram  
followers



100  
twitter  
followers



5 reddit  
followers



1 follower on github

简单解释一下雷达图旁批这四个名词的含义，只是直观理解：

- Commit：你在自己仓库里做了更改并提交，比如你写了一篇博客并传到github仓库，这就算一次“commit”
- Pull Request：你对别人的仓库做了更改，要跟仓库主人说一声并经过他的同意，这个“待检查的更改”就叫做一次“Pull Request”，简称PR
- Issue：你看了别人的教程仓库，但是还是有困惑/发现错误，要跟人家提问题，这就叫“issue”
- Code Review：你是大佬，别人交的代码要你过目，你“过目”的过程就叫“code review”

上述的四个操作非常非常重要，这里我只是形象的打个比方，不完全正确，后面会做详细讲解✓



这时我们注意到主页面上方有菜单栏：



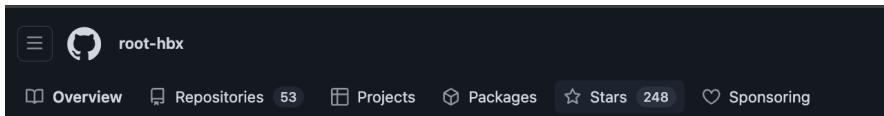
- **Overview**：就是你现在看到了个人页面，用于向别人展示你的个人形象
- **Repositories**：存放个人代码的仓库，我们将在后面做详细解释
- **Projects**：相当于是个计划表，你可以在这里列上对应的计划，可以进行链接
- **Packages**：你所发布的代码包，暂且不用管
- **Stars**：这就是传说中的 Star🌟🌟🌟 我们看技术文章可能常会遇到如“GitHub 上 Star 数过万的项目”等，就是指这个Star！比如，我们认为一个仓库很棒，那就在它的右上角点个Star，就相当于是“一键三连”了，之后还可以在自己的主页中点击星星图标查看
- **Sponsoring**：你可能还会注意到上面的图片显示了这个专栏，不过大概率你现在是没有的，先不用管

## Star

接触计算机科学来已有近两年半(警觉⚠️)，你肯定听说过“高star项目”之类的表达

这里的star表示你的“喜欢/赞赏”

具体可以看你的菜单栏：



点击 Stars，相当于现在来到你的“打赏列表”：

Lists (6)

- System: 18 repositories
- Networks: 36 repositories
- file: 27 repositories
- just4fun: 45 repositories
- xCCL: 5 repositories
- UCB-Course: 8 repositories

Starred repositories

- rcore-os / rCore**  
Rust version of THU uCore OS. Linux compatible.  
Rust ⭐ 3,518 📈 379 Updated on Aug 24, 2023
- Golden-Slumber / Decentralized-Satellite-FL-dev**  
Python ⭐ 4 📈 1 Updated on Jan 20

Starred topics

- gRPC
- jekyll-theme
- website-template

当你在开源社区冲浪时，看见自己觉得还不错的项目，可以考虑给一个star

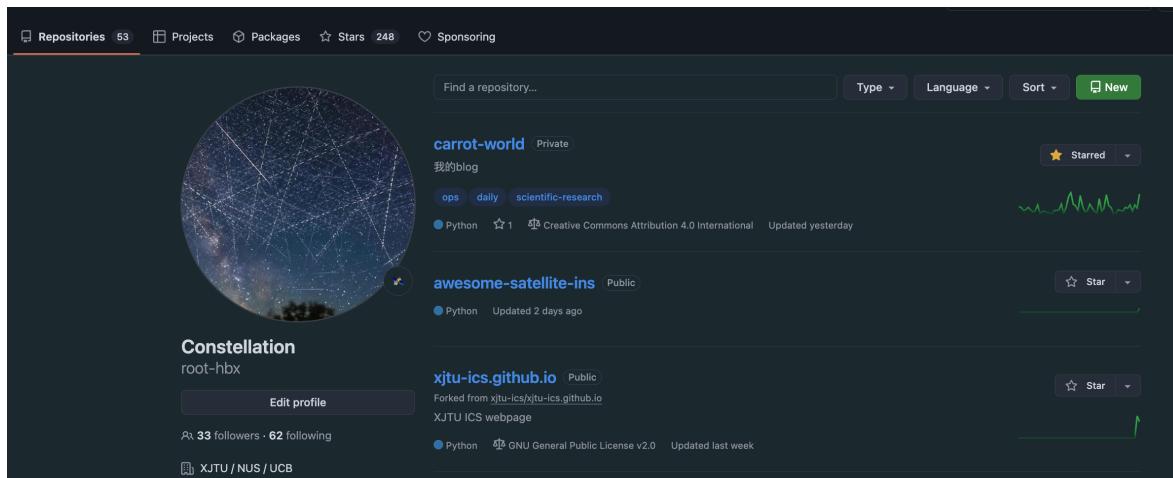
给star后，就会在这里呈现

同时，你也可以对star列表进行合理的分类，点击右上方的 create list 即可

比如，看上面的图片，这是笔者的“打赏列表”，分成 System / Networks 等类别

## 如何创建你的第一个仓库

我们经常会听到“在github上创建一个仓库”这类说法，这里我们会手把手教你如何来做



在菜单栏中，选择 `Repositories` 进入你的仓库列表：

点击右上角， `New`

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Required fields are marked with an asterisk (\*).

**Repository template**

No template ▾

Start your repository with a template repository's contents.

**Owner \***  root-hbx ▾ / **Repository name \*** test-for-ics ✓ test-for-ics is available.

Great repository names are short and memorable. Need inspiration? How about [cautious-palm-tree](#)?

**Description (optional)**

test-for-ics is a repo for demo (ics spring 2025)

 Public  
Anyone on the internet can see this repository. You choose who can commit.

 Private  
You choose who can see and commit to this repository.

**Initialize this repository with:**

Add a README file  
This is where you can write a long description for your project. [Learn more about READMEs](#).

**Add .gitignore**

.gitignore template: **None** ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

**Choose a license**

License: **None** ▾

A license tells others what they can and can't do with your code. [Learn more about licenses](#).

ⓘ You are creating a public repository in your personal account.

**Create repository**

在创建的界面里：

1. Repository name: 你想给这个仓库取什么名？
2. Description (optional): 对这个仓库的一句话介绍
3. Public / Private : 公有还是私有

- 公有: 所有用户都能看见这个仓库的全部内容和操作
  - 私有: 只有你自己可以看 (比如ics的实验, 就需要你们自行创建private仓库)
4. Add a README file (optional): 给这个仓库默认加一个“说明界面”
  5. Add .gitignore (optional): 给这个仓库默认加一个“自忽略白名单”
  6. Choose a license (optional): 选择一个默认的开源协议
  7. 点击 Create Repository 即可

## 如何创建/加入一个组织

组织: organization

这里以 XJTU-ICS 课程对应的开源社区为例 🙌

The screenshot shows the GitHub organization page for XJTU-ICS. At the top, there's a header with the organization's logo (a stylized dragon), name, description ("Introduction to Computer System of Xi'an Jiaotong University"), and stats (7 followers, China, URL). A "Unfollow" button is on the right. Below the header, there's a "README.md" file preview showing the course logo and title. To the right, there's a sidebar with options like "View as: Public", "Top discussions this past month", "People" (listing several user profiles), "Top languages" (Python, C, Shell, JavaScript, CSS), and "Most used topics" (xjtu, ics, computer-architecture, computer-systems, computersystems). The main content area below the README preview contains sections for the course schedule, textbook, DevOps, news, and support.

This is the official repository for the Introduction to Computer Systems (ICS) Course in Xi'an Jiaotong University.

- ICS Course Website: schedule / dashboard / course slides / labs ...
- ICS Textbook: orthogonal to certain aspects of the classroom material, serving as a reference for previewing / reviewing.
- ICS DevOps: contains basic command line essentials.

News

- [Feb 2025] First session of ICS Course is around the corner: [Time Table](#)
- [Feb 2025] Piazza is ready to go: [Enrollment](#)

Support and Questions

We are excited to hear your feedback!

- For issues and feature requests, please [open a GitHub issue](#).
- For questions, please use [GitHub Discussions](#).

For general discussions, join us on [XJTU-ICS Piazza](#).

Contributing

We welcome all contributions to the project! See [CONTRIBUTING](#) for how to get involved.

The screenshot shows the GitHub organization page for 'xjtu-ics.github.io'. It lists several repositories:

- xjtu-ics.github.io** (Public)  
XJTU ICS webpage  
Python, 11 stars, GPL-2.0 license, 3 issues, 0 pull requests, 0 forks, Updated 1 minute ago.
- cli-toolkit** (Public)  
cli-toolkit contains materials about git / ssh / vim / shell, etc.  
0 stars, AGPL-3.0 license, 0 issues, 0 pull requests, 0 forks, Updated 2 days ago.
- vm-scripts** (Private)  
some bash scripts for vm-ics  
Shell, 0 stars, 0 issues, 0 pull requests, 0 forks, Updated 5 days ago.
- datalab-sp25** (Public)  
XJTU-ICS spring 2025 datalab  
C, 0 stars, 1 issue, 0 pull requests, 0 forks, Updated 5 days ago.
- datalab** (Private)  
XJTU-ICS DataLab for instructors  
C, 0 stars, 2 issues, 0 pull requests, 0 forks, Updated 5 days ago.
- textbook** (Public template)  
mdBook for ICS course in Xi'an Jiaotong University  
CSS, 0 stars, GPL-3.0 license, 4 issues, 3 pull requests, 1 fork, Updated last week.
- .github** (Public)  
config for XJTU-ICS GitHub profile  
0 stars, GPL-3.0 license, 0 issues, 0 pull requests, 0 forks, Updated last week.
- cache-std-impl** (Private)  
0 stars, AGPL-3.0 license, 0 issues, 1 pull request, 1 fork, Updated on Jan 11.

organization 也有 profile、也有repo，你可以简单的把它看成一个“更大”的用户

具体如何创建、管理一个organization，此处省略，感兴趣的同學可以自行上网搜索

## TL;DR

至此，github的“最基础入门之旅”就结束 🎉🎉🎉

很显然这只是github的冰山一角，更多的內容还得自行去探索，我们这里只是给大家一个最基础的

宏观介绍，相当于是“埋下一颗种子”

至于是否能成长为“参天大树”，还得看你自己的求知欲和实践情况 😊

这里给出一些“探索路线”的推荐 🤙

Basic Operation (ignored for now):

- How to create a repo and make a commit
- Local machine and remote repo
- Connect github with SSH (server / local machine)

Advanced (ignored as well):

- Github Action and Workflows
- Github Pages
- Make a PR
- Report an Issue
- Github Organization Management
- GPG Keys
- ...

希望大家可以在开源这条路上越走越远 💪

---

© 2025. ICS Team. All rights reserved.

# SSH

现在是5202年，你在无数学长学姐的安利下选择了《计算机系统导论》，并被告知这门课有可能是你在沙坡村学院能够上到的最有用的课。

在被沙坡村学院折磨了许久之后，你像抓住了救命稻草一般，准备再度踏上计算机学习的正轨...



突然，你的手机震动一下，打开一看是助教在ICS课程群里发送了消息：

“...我们为大家准备了服务器，并创建了账号密码，需要使用服务器的同学请SSH到服务器上...”

然而，你没有听说过SSH，也不知道它如何使用，只是盯着自己电脑上的Dev C++发呆，

“算了，先开一把瓦吧”，你对自己说到...

这篇文章，我们将手把手教你如何使用SSH，常见的SSH使用场景，如何编写SSH配置文件...

此外，我们还会介绍SSH的底层工作原理，供对底层感兴趣的同学进行学习

当然，笔者不过是比你们多上了两年学的学长/学姐，对于SSH的理解难免出现偏差 或错误 ，欢迎各位同学批评指正

## 什么是SSH？

SSH(Secure Shell)是一个**应用层协议**，旨在在不安全的网络上提供安全的远程访问和网络服务。

在SSH出现之前，一般是使用Telnet和FTP等进行远程登陆和文件传输。

然而这些传输方式均采用**明文**进行传输，相当于直接公开自己传输的信息，十分不安全。

SSH使用**加密技术**在两台通信主机之间建立了一个安全的通信信道，因此更加适合现代网络环境。



### Note

**明文**和**密文**是密码学中的两种术语。

假设两台主机之间需要传递的内容称为信息，则明文指没有任何修饰的（例如加密）原始信息，密文则指使用加密技术进行加密之后的信息。

### Note

SSH一般指协议标准，而我们日常用的SSH工具一般是OpenSSH，这是SSH协议的开源实现。

## SSH的常见使用场景

一般而言，SSH主要有下面几种应用场景：

- 远程登陆
- 安全文件传输
- 端口转发

## SSH登陆远程服务器

SSH远程登陆类似于Windows系统上的远程桌面，不过不同的是，没有桌面环境，只有一个命令行

终端。

使用SSH登陆服务器之后，你可以像在本地使用终端一样来操作服务器，这在网络管理和远程开发中十分常见。

由于笔者使用的是物理机Arch Linux，因此下面主要以Linux系统为例介绍命令。

### Note

本文后续统一将自己本地的主机称为客户端，将远程主机称作服务器。无特殊说明时，客户端和服务器均为Linux系统。

## For Linux

SSH的登陆验证方式主要有**用户密码验证**和**密钥验证**两种

### 用户密码登陆

用户密码登陆是常见的身份验证方式，一般使用以下的命令格式：

```
ssh user@host [-p port]
```

详细解释一下上述命令：

- `ssh`：表示使用 `ssh` 命令
- `user`：表示需要远程登陆的主机上的用户名，如果用户名和本地用户名相同，则这一部分可以省略
- `host`：表示要登陆的主机名，可以是主机的域名，也可以是主机的IP地址（公网IP或者局域网内私有IP）
- `-p port`：可选，用于指定端口，默认端口为22，如果SSH服务器监听默认端口，则可以省略这一部分

输入上述命令之后，如果没有问题，终端会提示让你输入密码（首次登陆密码一般会告知你，或者按照自己重新设置的密码），

 Note

输入密码的时候，终端不会有任何显示，不会显示密码原文或是\*，防止别人窃取密码内容或密码长度。

正确输入密码之后，终端上如果输出一堆系统信息，然后打印出远程主机的终端提示符，则说明我们成功登陆了

 Tip

首次登陆时终端会显示类似这样的一段话（例如ssh首次登陆github）：

```
The authenticity of host '[ssh.github.com]:443 (<no hostip for proxy command>)' can't be established.  
ED25519 key fingerprint is  
SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvC0qU.  
This key is not known by any other names.  
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

这段话的目的就是告诉你，客户端知道了这台服务器的公钥的fingerprint，但是无法验证服务器的身份，主要是用于防止中间人攻击。

你需要输入 yes， no 或者 fingerprint 来跳过验证直接连接，拒绝连接或者使用 fingerprint 来验证。

大部分安全（相信我，没人会攻击你每月几块钱的云服务器的 :blush:) 的情况下，可以直接输入 yes。

对于公开的服务器，比如Github，官方一般会提供 fingerprint 供用户验证，例如[Github SSH Key fingerprints](#)。

输入fingerprint并按下回车，这时终端会提示：

```
Please type 'yes', 'no' or the fingerprint:
```

此时输入官方提供的fingerprint，如果验证通过，则终端会提示：

```
Warning: Permanently added '[ssh.github.com]:443' (ED25519) to the list of known hosts.
```

表示这台服务器通过了认证，并且会永久添加到 known\_hosts，这是 ~/.ssh/ 目录下的一个文件。

这个文件中记录了所有通过认证的服务器，当下次连接服务器时，若服务器在 known\_hosts 中出现，则会将服务器公钥的fingerprint和这个文件中记录的fingerprint比较，若匹配成功则连接继续，否则会发出警告，提醒你服务器fingerprint改变，需要用户自己确认服务器身份。

## Tip

fingerprint指公钥的**数字指纹**, 由服务器公钥通过特定哈希算法（如SHA256）生成, 这样长度比较短小, 方便比较。

对于公开的大型服务器, 如Github等, 一般会给出自己服务器公钥的fingerprint, 用于用户验证服务器身份。

如果是自己的公网服务器, 可以登陆上服务器查看公钥fingerprint, 服务器公钥一般位于 /etc/ssh/ 目录下, 命名为 ssh\_host\_xxx\_key.pub , xxx是算法类型, 比如使用rsa算法生成的公钥文件就是 ssh\_host\_rsa\_key.pub 。

服务器在第一次安装ssh服务器时会生成这些文件, 使用命令 ssh-keygen -lf /etc/ssh/ssh\_host\_rsa\_key.pub 可以查看公钥的fingerprint。

## Danger

用户密码登陆方式存在安全性问题, 尽管用户密码会通过加密传输, 如果用户设置的密码比较简单（尤其是root用户的密码）, 很容易被黑客通过暴力破解出来, 因此十分建议配置ssh文件, 使用密钥进行登陆, 这样不仅提高了安全性, 每次登陆时也不用输入密码, 实现免密登陆。

如果需要将服务器部署在公网, 通过建议在第一次登陆之后就**禁用密码登陆或者限制尝试次数**, 以及**修改ssh默认端口**等一系列复杂的操作来确保安全。

## Tip

登陆之前确保在远程服务器上已经安装了ssh服务器, 通常是openssh-server, 如果没有安装的话根据自己的linux发行版google如何安装吧, 这里就不展开讲了。

 Tip

如果出现了 `Permission denied (publickey)`., 通常是远程服务器禁用了密码登陆服务器, 这时候必须使用密钥验证进行登陆。

如果你可以操作远程主机打开密码登陆, 则修改ssh服务器的配置文件 `/etc/ssh/sshd_config`, 找到 `PasswordAuthentication no` 这一行, 将其注释掉, 然后使用 `sudo systemctl restart ssh` 重启ssh服务。

## Example

这里给出一个登陆的例子，假设我的本地主机和远程主机在同一个局域网内，远程主机ip为192.168.110.242，假如不知道远程主机ip的话，可以在远程主机上使用 ifconfig 命令查看。

现在远程主机创建了一个用户叫做ttang，接着输入ssh命令进行连接，出现以下内容：

```
ttang in ~ ^ ssh ttang@192.168.110.242
ttang@192.168.110.242's password:
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-40-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

Expanded Security Maintenance for Applications is not enabled.

209 updates can be applied immediately.
119 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

14 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

The list of available updates is more than a week old.
To check for new updates run: sudo apt update
New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Tue Feb 11 14:56:50 2025 from 192.168.110.56
```

上述欢迎界面说明登陆成功。

如果是公网服务器等，将远程ip换成对应的公网ip或者域名，如不知道，请联系相应的服务器管理员。

## 密钥登陆

前面提到，用户密码登陆的方式并不安全，因此更加建议使用密钥进行身份验证。同时，这种方式也无需每次都输入密码，使用起来更加方便。

首先我们需要在本地创建一对公钥和私钥，可以使用下面这条命令进行创建：

## ssh-keygen

这条命令默认使用ED25519算法生成长度为256位的密钥对

### Note

OpenSSH较新的版本已经将默认的rsa密钥生成算法改成了ed25519，这时更加现代的做法，兼顾了安全性和效率。

一些常用可选项：

- `[-t rsa | dsa | ecdsa | ecdsa-sk | ed25519 | ed25519-sk | rsa]`：指定加密算法，不同加密算法复杂性和安全性也不同
- `[-b bits]` 指定密钥长度，有些加密算法，如rsa，密钥长度是可以变化的

### Note

对于 RSA 算法，常见的长度是 2048, 3072, 4096。一般推荐 3072 长度，是安全性和性能的最佳平衡，4096 长度的密钥能提供最高的安全性，但是加解密的时间会显著增加。不过在现代计算机上，椭圆加密的 ed25519 是更好的选择，因为它兼顾了安全性和性能。

输入以下命令了解更多：

```
ssh-keygen --help
```

输入上述命令之后，不出意外的话，会提示指定密钥存放路径，默认为 `~/.ssh/id_rsa`（如果使用rsa算法），若你不想指定其他路径的话，直接按回车即可。

接着会提示你输入密码和确认密码，这个密码是每次使用密钥需要输入的，如果不设置密码，直接按两次回车即可。

于是我们的ssh密钥对就生成好了 

通过 `ls` 查看 `~/.ssh` 目录或者你自己指定的存放目录

```
ls /your/path/to/ssh/keypair
```

目录下会存在两个文件 `keypair.pub` 和 `keypair`, 其中 **keypair** 是你自己指定的文件名, 或者使用特定算法生成的默认名, 例如 `id_rsa`, 带有 `.pub` 后缀的是公钥, 不带后缀的是私钥。

### Danger

注意私钥一定不要透露给任何人, 否则加密就失效了

然后我们需要将公钥复制到服务器上, 这可以使用以下命令 :

```
ssh-copy-id [-i /your/path/to/ssh/keypair.pub] user@host
```

上述命令 `-i` 选项指定公钥文件目录, 如果你没有自己指定密钥名, 则可以省略这个选项

拷贝成功之后, 就可以免密登陆啦 

```
ssh user@host [-i /your/path/to/ssh/keypair]
```

如果你不是默认路径, 使用 `-i` 选项指定路径即可

### Example

接着上面的例子, 首先在本地生成ssh密钥对 :

```
ttang in ~ λ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ttang/.ssh/id_rsa):
Enter passphrase for "/home/ttang/.ssh/id_rsa" (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ttang/.ssh/id_rsa
Your public key has been saved in /home/ttang/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:WEu57cJoz9G0hRy9J1YFRzizpfUbUNDp1pEhYjkT4j4 ttang@ttang-desktop
The key's randomart image is:
+---[RSA 4096]---+
| . +**Bo+ |
| ..o==++= |
| +.. =Bo.o |
| +.= +o. +o |
| . SEB + . o |
| o =.= . |
| o + + |
| . o o |
| o |
+---[SHA256]---+
```

查看一下生成的公钥：

```
ttang in ~ λ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAQD8cNz84bogv43Rga0M1LjjL9uHuseaquryR2TMNRVx0yyf5W6xE8Tp04ipJgo5u7LWjNFXLHmE05BgYS3XYHmkn0F0JNAQTA4bBM2IVF9MEvM0rbYdy2UR11n3B
XcTX3jGUjB82ye1822oJAPvZx07Mz02XIG6onw0ZbVKjvJz4g/49rmmi1EA1nW3IpHTbna6Yuqls6sdU2d19wTTg19A/NNOno8+5yfLE1ekkzYZZYJhAu6REFwU/Vf4o1ht3De28rTwYQhf7L26/wpA9ffAgx90PB
Wx7e2xZhjg/QY51-TvgkyQNIIrdJ+4710HowEr0NIMc21vIauuHyF/g67iSUb800KXXCyglvNp2hynv17fFefh1sCrU6g2YWeS9KbLSiSuVNU3/KctdxG6bCbdk75cTCJngvVdl.17ZmEisdynnJQ01ukZRBf2LPjsNM
j26x+F1wHM9Ec36nulzU0i0hBq1kaV3c+okPEWcEIJ9sz70/www== ttang@ttang-desktop
```

将公钥复制到服务器上：

```
ttang in ~ λ ssh-copy-id ttang@192.168.110.242
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/ttang/.ssh/
id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompt
ed now it is to install the new keys
ttang@192.168.110.242's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'ttang@192.168.110.242'"
and check to make sure that only the key(s) you wanted were added.
```

使用ssh进行免密登陆：

```
ttang in ~ λ ssh ttang@192.168.110.242
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-40-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

Expanded Security Maintenance for Applications is not enabled.

209 updates can be applied immediately.
119 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

14 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

The list of available updates is more than a week old.
To check for new updates run: sudo apt update
New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Tue Feb 11 16:13:15 2025 from 192.168.110.56
```

## 配置config文件快速登陆

尽管使用公钥登陆的方式免去了密码，但是每次登陆还是需要输入很长一串东西，太繁琐

作为计算机专业的同学，当然是要学会编写各种配置简化使用，下面我们介绍如何配置ssh以更高效的使用

ssh的配置文件位于`~/.ssh/config`，如果没有就创建一个，使用编辑器（推荐`vim/neovim`，毕竟是神之编辑器）打开这个文件，然后输入以下内容（具体内容替换成自己的）

```
Host hostname
  HostName host
  User user
  Port port
  IdentityFile /your/path/to/ssh/keypair
```

详细解释一下上述的每个字段：

- **Host**：给你自己要连接的服务器随便起个名
- **HostName**：这里host填入实际服务器的域名，或者公网IP，或者局域网内的私有IP

- User : 填入自己服务器上的用户名
- Port : SSH服务器监听的端口, 默认为22, 如果SSH服务器监听22, 可以去掉这一行
- IdentityFile 填入自己的私钥文件的地址

全部配置完成之后, 输入以下命令:

```
ssh hostname
```

即可成功登陆, 是不是简单了很多呢

### Example

编写ssh配置文件如下:

```
Host myhost
  HostName 192.168.110.242
  User ttang
  #IdentityFile "~/.ssh/id_rsa"
```

直接使用myhost进行登陆:

```
ttang in ~ λ ssh myhost
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-40-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

Expanded Security Maintenance for Applications is not enabled.

209 updates can be applied immediately.
119 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

14 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

The list of available updates is more than a week old.
To check for new updates run: sudo apt update
New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Tue Feb 11 16:28:36 2025 from 192.168.110.56
```

## For MacOS

MacOS系统的步骤和Linux完全相同，这里就不再赘述了。

## For Windows

### SCP安全传输文件

在两台主机之间**安全**的传输文件一般有scp和sftp两种方式，不过笔者没太用过sftp，对其不太熟悉，因此主要介绍一下scp。

scp类似cp，不同的是scp可以在两台联网主机之间进行文件传输，并且**基于ssh进行加密传输**。

scp的用法包括：

- **在本地和远程主机之间文件传输**：文件可以从远程主机下载到本地，或者从本地上传到远程主机
- **传输文件夹**：scp支持使用 -r 参数复制整个目录
- **两台远程主机之间进行传输**：scp可以直接指定两台远程主机之间进行文件传输，而无需先下载到本地。

scp针对上述用法有不同的命令，不过结构上都很相似。

将本地文件上传至远程主机时：

```
scp [-P port] [-i /path/to/your/key] /local/file/path [-r]
user@host:/remote/file/path
```

其中：

- **[-P]指定端口，默认端口22，注意P要大写**
- **\*\*[-i]\*\*指定私钥文件路径，如果需要使用密钥验证的话，否则将使用默认私钥路径或者密码验证**
- **\*\*[-r]\*\*如果目标文件是目录则需要指定**

注意远程路径格式为**主机名:远程主机的路径名**。

先出现的路径是**源路径**, 后出现的是**目的路径**。

因此, 从远程服务器下载时 :

```
scp [-P port] [-i /path/to/your/key] user@host:/remote/file/path  
/local/file/path
```

在两台远程主机之间传输时 :

```
scp [-P port] [-i /path/to/your/key] user1@host1:/remote/file/path  
user2@host2:/remote/file/path
```

### Note

如果你使用了ssh配置文件, 那么上述的主机名均可直接替换成你在配置文件中指定的**Host**名, 也不再需要输入密码或者是指定私钥路径等, 这也是推荐的做法。

## SSH的工作流程和原理

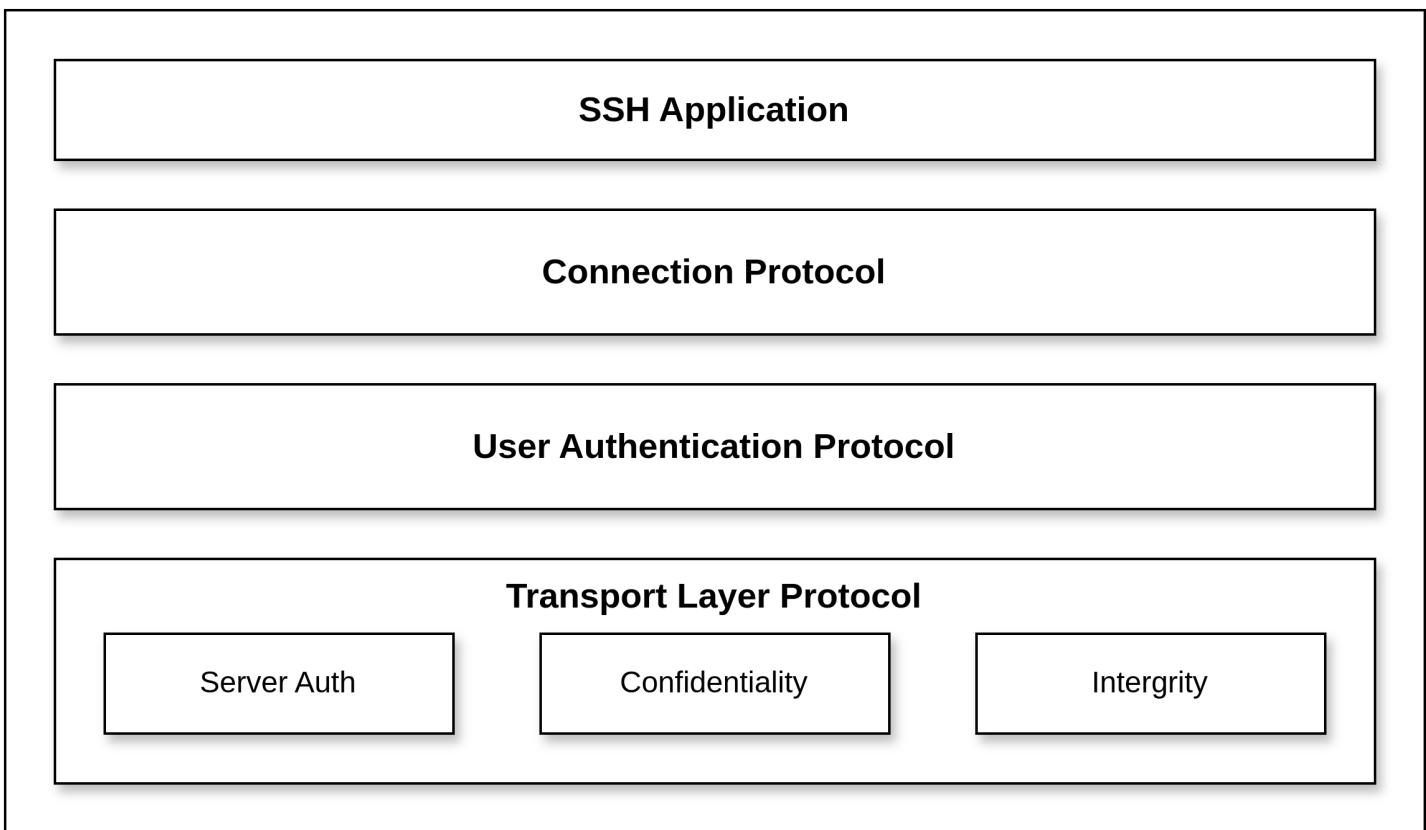
这一部分详细讲解有关SSH协议的工作流程。

### Warning

这部分涉及到有关**操作系统**, **计算机网络**, **密码学**等相关知识, 如果不熟悉这部分知识理解起来会有难度。

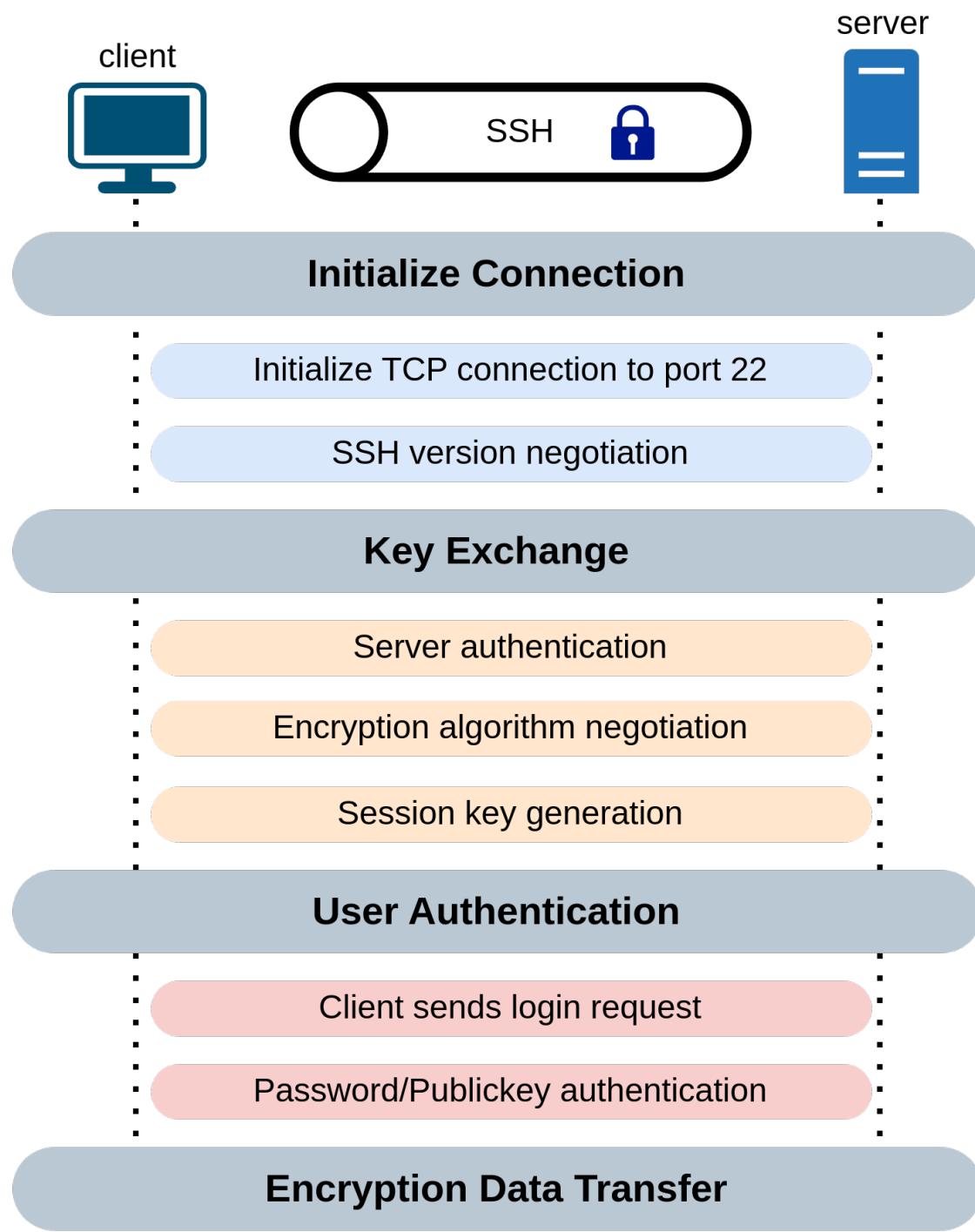
如果你只是希望了解ssh的有关应用, 直接跳过这一节即可。

## SSH Overview



SSH协议结构如上所示，主要分为以下几个部分：

- **传输层协议**：SSH传输层协议主要确保**服务器身份验证**、**数据加密**和**数据完整性**。注意这个传输层协议应该和OSI标准的7层网络协议栈中定义的传输层协议区分开，SSH传输层协议直接运行在TCP/IP协议栈之上，通常基于TCP进行可靠数据传输，默认端口22。
- **用户身份认证协议**：这个协议用于验证用户身份，最常见的两种方式为**用户密码验证**和**公钥认证**，或是二者结合的方式。
- **连接协议**：连接协议**多路复用**了单个ssh客户端-服务器组成的ssh通道。换句话说，这个协议创建不同的数据流和多个逻辑通道。



SSH协议基于客户端-服务器（C/S）模型，其通常使用TCP协议作为传输层协议，使用TCP时，默认监听22端口。SSH最终使用密钥对用户传输的数据进行加密传输，因此实现了在不安全的网络上进行安全传输，SSH从建立连接到数据加密传输的流程大致如上图，主要有以下几个阶段：

- **连接建立**：客户端首先发起TCP连接，经过三次握手之后，TCP连接成功建立。之后客户端与服务器互相协商可用的SSH版本。

- **密钥交换**：这一部分主要任务是**协商加密算法**（对称/非对称加密，哈希），然后通过**密钥交换算法**生成用于加密信息的会话密钥，这一阶段同时还包含了**服务器端的身份认证**，以防止中间人攻击。
- **用户身份认证**：这一阶段用于验证**客户端身份信息**，也就是前面提到的登陆阶段。
- **信息加密传输**：这一阶段连接双方的身份信息均已验证，后续的数据传输均采用之前生成的会话密钥进行加密解密。

### Note

**协议**是计算机网络中的一个重要概念，其规定了通信双方进行数据交换时应该遵守的一组规则。

**协议栈**则是计算机网络中一系列协议的集合，不同协议位于不同层次，旨在解决不同层次上的问题，低层次的协议为高层次提供服务。

两大经典协议栈是**OSI模型**和**TCP/IP模型**：

OSI模型分为了7层，分别是应用层，表示层，会话层，传输层，网络层，数据链路层和物理层。

TCP/IP模型分为了4层，分别是应用层，传输层，网际层和网络接口层。

在一般的计算机网络教学中，为了方便，一般分为应用层，传输层，网络层，数据链路层和物理层。

### Note

TCP协议是**传输层可靠数据传输服务**，即应用层可以确保TCP会将数据**按序，不丢包**的发送给对端，常用于需要数据可靠传输的应用层协议（如电子邮件，文件传输）中，但时延较高。

与之相对的是UDP协议，UDP不保证数据可靠传输，因此性能好，常用于需要保证低时延等高性能场景，如流媒体，音视频流等。

 Note

**进程**是操作系统中的一个重要抽象，简单来说，进程可以看作一个**运行中的程序**。以C语言为例，编译系统会将一个或多个C源程序编译链接成一个可执行程序，到目前为止，这个可执行程序仍然驻留在磁盘上，本质上是**静态的一堆机器指令的集合**。当运行这个可执行程序时，加载器在内核中会创建一个进程，将代码正确映射到进程的地址空间中，设置好各个寄存器，最后CPU开始运行，此时**程序从磁盘被加载到了内存**。

一个进程可以看作一个容器，管理了很多资源，包括寄存器，地址空间以及各种信息，通常叫做**进程上下文**。进程中执行代码的部分可以看作一个执行流，通常叫做**线程**，一个进程可以包括一个或多个线程。

简单概括，进程更加注重**资源管理**，线程则聚焦于**命令执行**，可以直接被CPU进行调度，因此线程更加轻量。

 Note

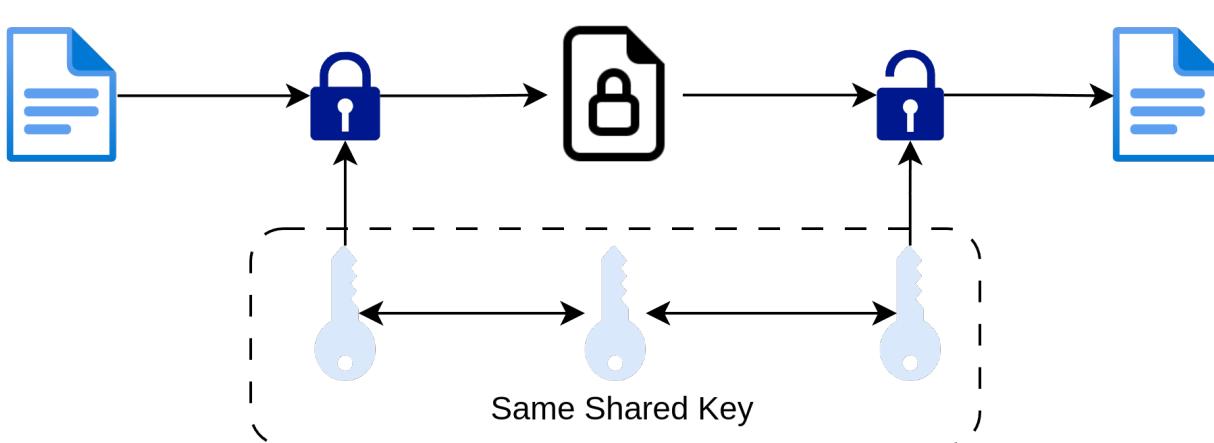
**客户端-服务器模型**是主流的通信模型，其余通信模型包括P2P（Peer to Peer）等。客户端和服务器都可以视为运行在主机上的一个进程。其通过网络进行通信，这是一种常见的**进程间通信方式**。在客户端-服务器模型中，客户端作为主动发起连接的一方，而服务器被动接受来自客户端的连接。

 Note

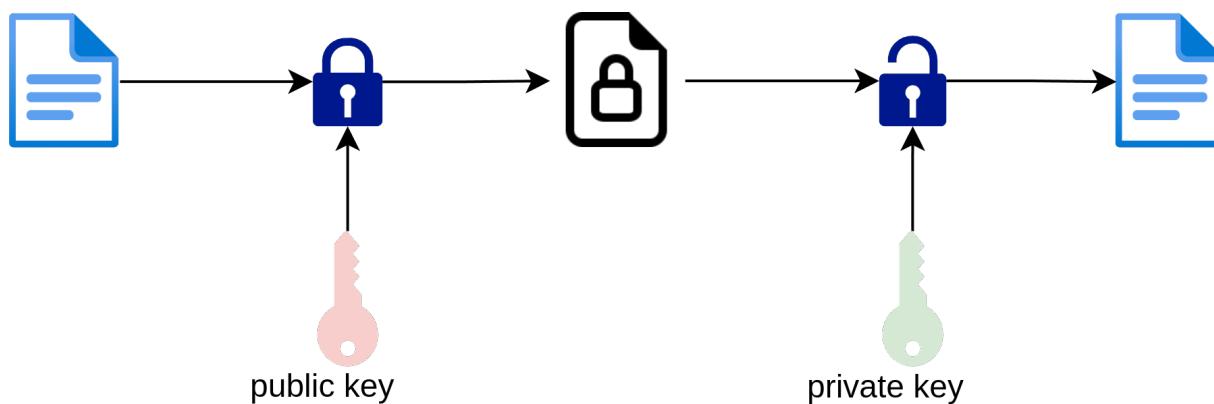
计算机网络中，用于标记一个主机的身份通常需要**IP地址**和**端口**，更准确的说，IP地址用于识别主机，是网络层中的概念。端口号用于识别主机上运行的不同进程，是传输层的概念。例如，同一台主机上可能既运行了HTTP服务器，监听80端口，也可能同时运行了ssh服务器，监听22端口，他们共享内核中实现的TCP模块，TCP模块接受一个数据包时，根据端口号将数据包分发给不同的进程。

 Note

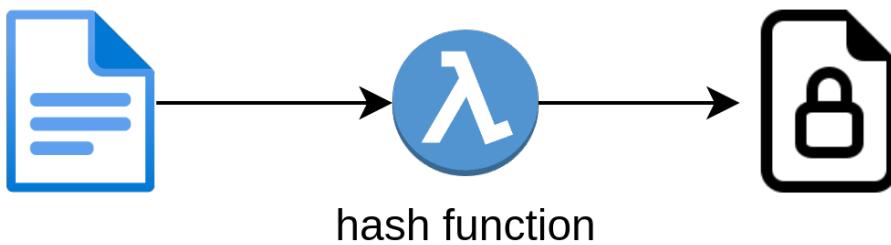
密码学中加密方式主要有**对称加密**，**非对称加密**和**哈希**三种，三种加密方式分别应用于不同的情况。



对称加密又叫做**共享密钥加密**，其基本特征是，发送方和接收方均采用同一个密钥对数据进行加密和解密，就像用同一把钥匙上锁和开锁一样。这种加密方式简单且高效，但是共享密钥的保管是个问题，因为一旦存在第三者得到了共享密钥，其就能对加密内容进行解密，安全性就荡然无存。因此，若需要在网络中传递共享密钥，需要**对共享密钥进行加密**。



非对称加密使用**一对密钥**进行加密和解密，而不是单个密钥。发送方生成一对密钥，分别叫做**公钥**和**私钥**，其中公钥可以安全的公开，私钥则只能自己知道，并且任何人**不能从公钥轻易的推出私钥**。非对称加密拥有的一个重要性质是：**公钥加密的内容只能由私钥进行解密，反之，私钥加密的内容只能由公钥进行解密**。二者分别应用于不同场景，对于信息加密传输的场景，需要采用公钥加密私钥解密的方式，因为公钥是公开的，若使用私钥加密则任何人都可以使用公钥解密，加密就失效了。而私钥加密公钥解密通常应用于身份验证，比如**数字签名**。



**哈希**是另外一种加密方式，哈希加密通常以消息作为输入，并且对于不同的输入可以生成唯一且定长的哈希内容，又叫做消息的**摘要**，且哈希算法必须保证无法从哈希值推出原始信息内容。哈希加密与上述两种加密不同的是其**不需要解密**，因此是单向的，通常用于验证**数据完整性**，确保数据没有被篡改，比如MAC等。通信的双方只需要知道原始信息和使用的哈希算法就能够验证哈希值是否相同。

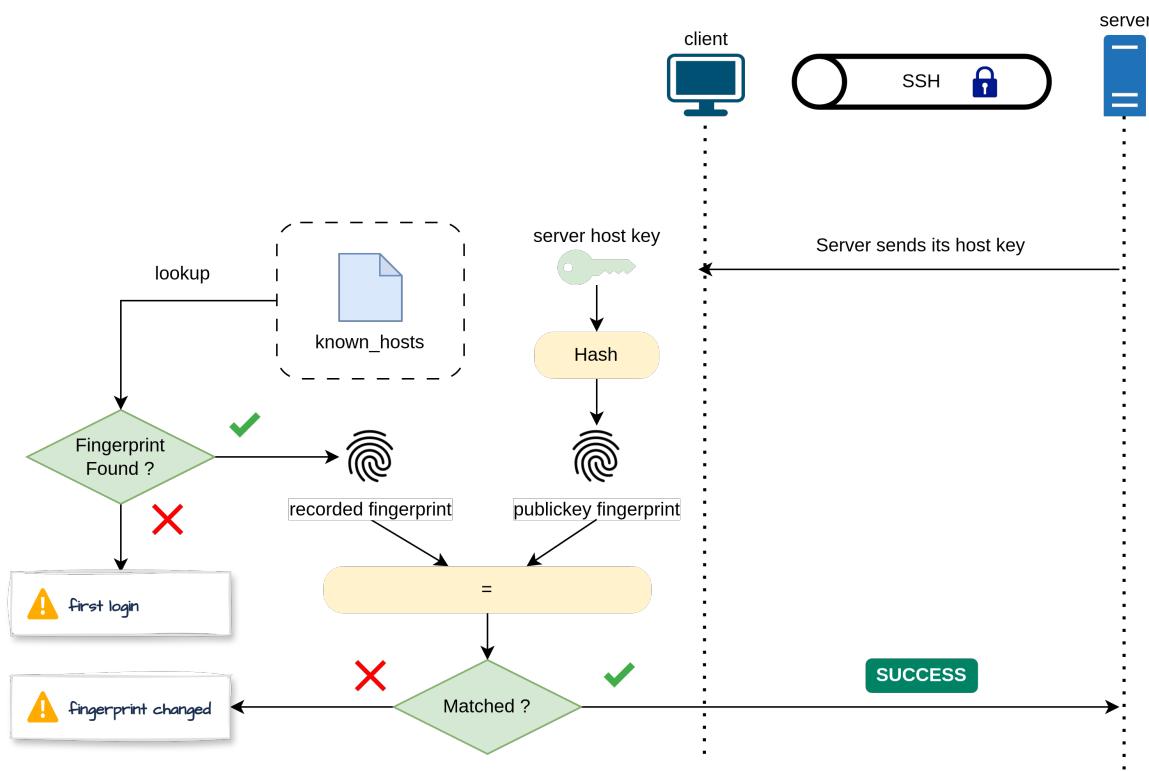
SSH加密技术经常混合使用上述三种方式。比如，非对称加密用于生成会话密钥，而会话密钥是对称加密，负责后续所有数据传输的加密。哈希加密用于确保数据完整性，在会话密钥生成之后，数据传输还需要附上由<会话密钥，数据包ID，实际信息内容>由MAC算法生成的摘要。

### Note

**数字签名**在密码学中用于验证身份信息，发送方可以将需要发送的信息以及附加信息进行哈希生成，然后使用自己的私钥对此摘要进行加密，然后将签名附在原有信息后面进行传输，这样任何拥有发送方公钥的用户可以解密这个签名，由于能够进行加密和解密的公钥和私钥是配对的，而发送方私钥不公开，因此接收方可以证明数据真实来源与发送方。这样做的效果就像你在合同上签名，然后对方通过签名可以判断合同是你签的。

尽管数字签名可以验证身份，但是其会受到中间人攻击的影响，假如黑客窃取了原有的数据包，然后用发送方的公钥进行了解密得到原始信息，之后使用自己的私钥将信息加密，再发送自己的公钥。这样接收方同样可以解密，但是发送方的身份就无法得到验证。为了应对这种攻击，**需要对发送方的公钥也进行验证**，这通常通过CA (Certificate Authority) 完成，即发送方的公钥必须得到CA的验证，具体方式为，CA使用自己的私钥将发送方的公钥以及附加信息进行签名，接收方使用CA的公钥进行解密，得到发送方公钥，在使用这个公钥对发送方签名进行解密，从而进行验证。由于CA的公钥一般不可能作假，因此使用这种方式只要信息被中间人窃听并篡改，接受方就能立刻验证失败。这就好比有人仿照你的字迹进行签名，而你在签名边上再附上一位权威人士的签名，证明你的签名是真的。

## 服务器身份认证



在SSH建立连接之后，客户端需要验证服务器的身份，以防止**中间人攻击**。在这个阶段，服务器会将自己的公钥（通常位于`/etc/ssh/`）目录下，发送给客户端，客户端接收公钥并计算其指纹。如果客户端保存了服务器的公钥指纹（在`~/.ssh/known_hosts`文件中存在），则会进行比对，若比对成功则连接继续，否则警告用户指纹不匹配，需要用户自己确认身份。这也就是为什么第一次连接时会发出警告，告诉用户需要自己验证服务器的指纹，因为SSH服务器使用的公钥没办法通过CA进行认证，无法自动确认服务器的真假。

### Note

**\*\*中间人攻击(Man In the Middle Attack)\*\***是网络安全中的一中常见攻击方式，方式为存在第三者隐藏并且窃听通信双方的内容，充当前代理转发的作用，从而窃取通信内容或者恶意篡改通信内容，而通信双方察觉不到这个中间人的存在。

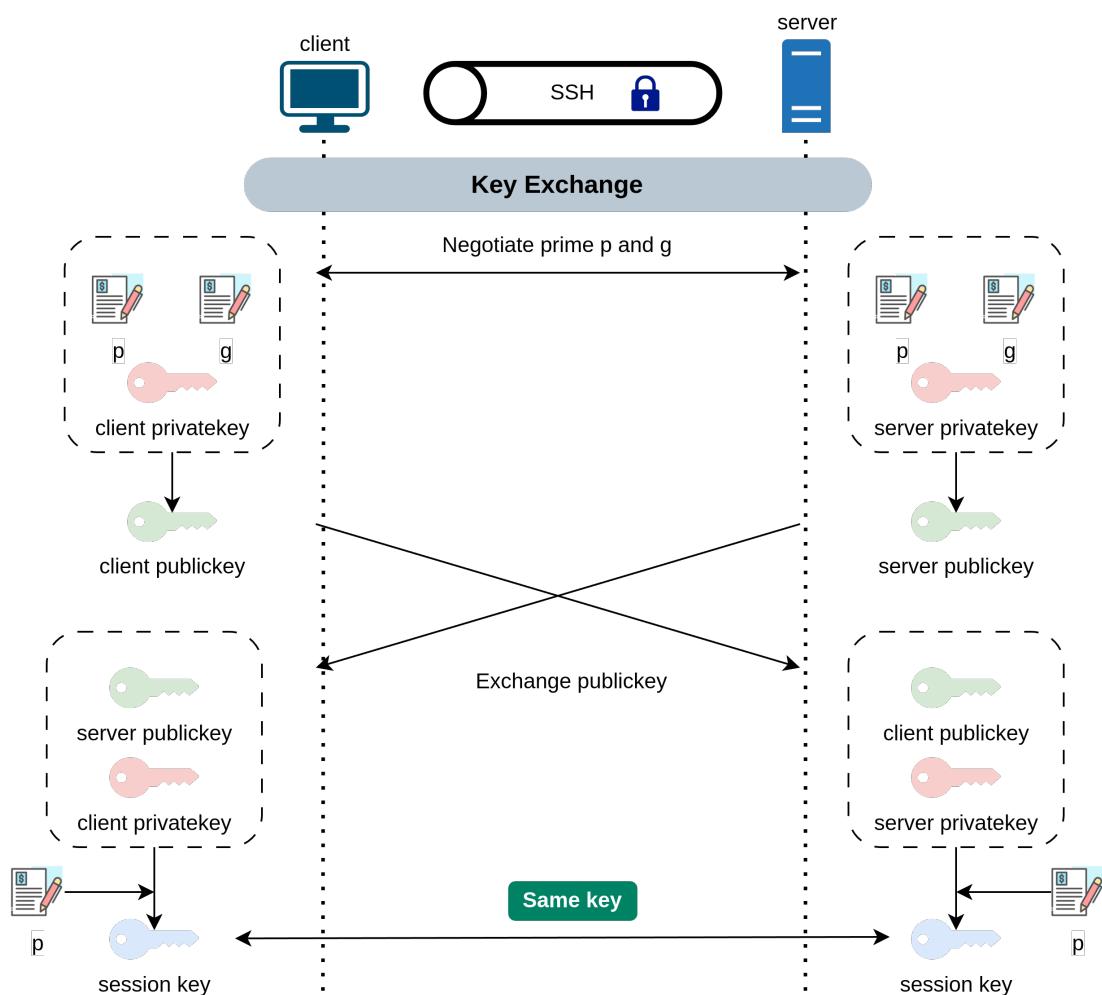
## 算法协商

算法协商阶段，客户端和服务器各自生成自己支持的算法列表，然后双方根据列表选择大家都支持的算法，需要协商的算法包括**对称加密算法**、**非对称加密算法**和**哈希算法**等。SSH协议混合使用了各种加密算法，以享受不同加密算法在不同场景下的优势，比如对称加密使用简单快速，因此用于**会话密钥**，而非对称加密使用比较复杂，但安全性高，因此常用于生成会话密钥以及身份验证，而哈希算法通常用于生成短小的指纹或者消息的摘要等。当所有类型的算法全部协商完成之后，协商阶段结束，若任意一个阶段双方无法找到共同的算法，则协商阶段失败。

## 密钥交换

密钥交换指一类算法，其目的是**生成会话密钥**。在SSH中，会话密钥通常使用的是对称加密算法，因此需要保证客户端和服务器双方的会话密钥是一致的，常见的算法有DH、ECDH等。

### Example



经典的DH (Diffie-Hellman) 密钥交换算法流程如下所示：

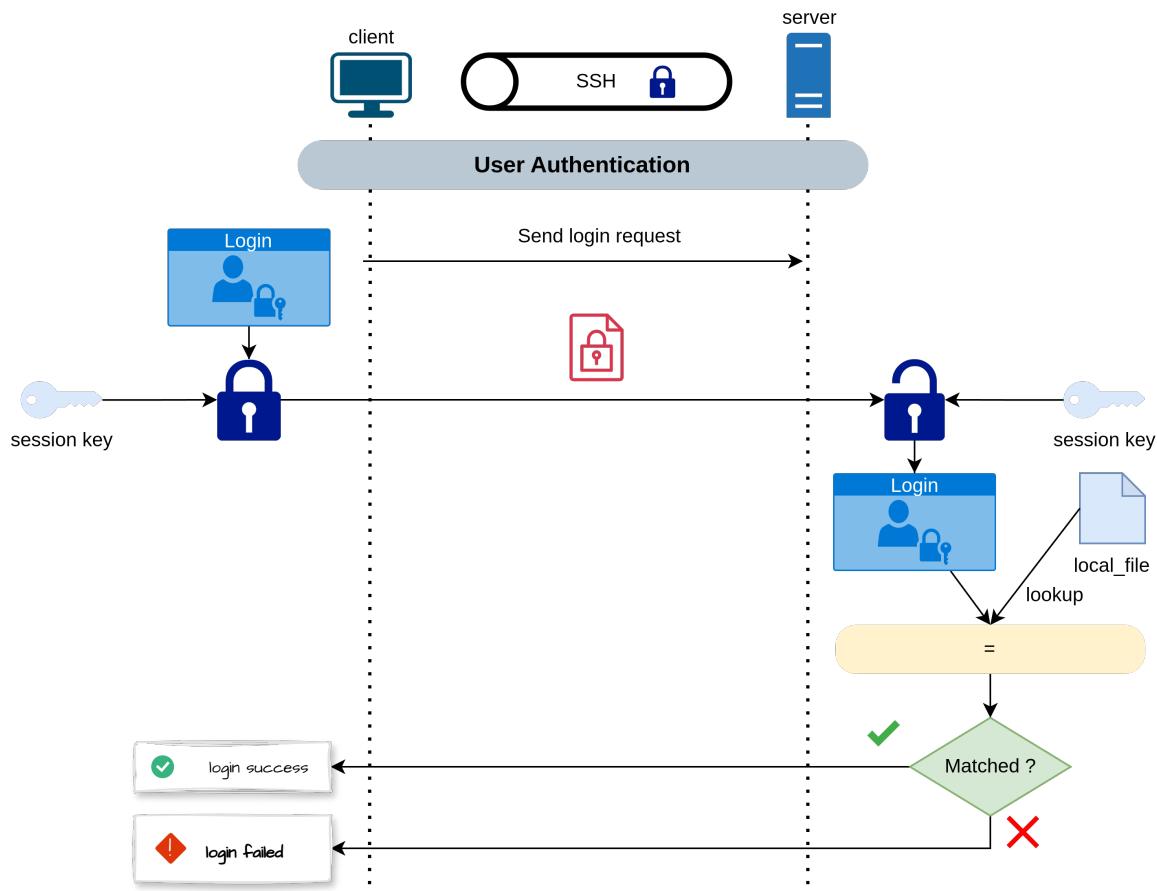
1. 客户端和服务器约定好一个大素数 $p$  (几百位) 和素数 $g$  ( $g$ 通常比较小, 且满足 $g < p$ ) ,  $p$ 和 $g$ 可以安全公开
2. 客户端选择一个很大的自然数 $a$ 作为私钥 (满足 $a < p$ ) , 并计算出客户端的公钥 $A = (g^a) \bmod p$
3. 服务端选择一个很大的自然数 $b$ 作为私钥 (满足 $b < p$ ) , 并计算出服务器的公钥 $B = (g^b) \bmod p$
4. 双方各自公开并交换自己的公钥 $A$ 和 $B$
5. 客户端根据自己的私钥 $a$ , 服务器公钥 $B$ 和 $p$ 计算会话密钥 $S = (B^a) \bmod p$
6. 服务器根据自己的私钥 $b$ , 客户端公钥 $A$ 和 $p$ 计算会话密钥 $S = (A^b) \bmod p$

DH算法核心基于**离散对数**, 数学上可以证明客户端和服务器计算得到的会话密钥 $S$ 是相同的。DH算法从根本上移除了会话密钥在网络中传输的步骤, 而保证通信双方得到相同的会话密钥, 因此大大提高了安全性。

## 用户身份验证

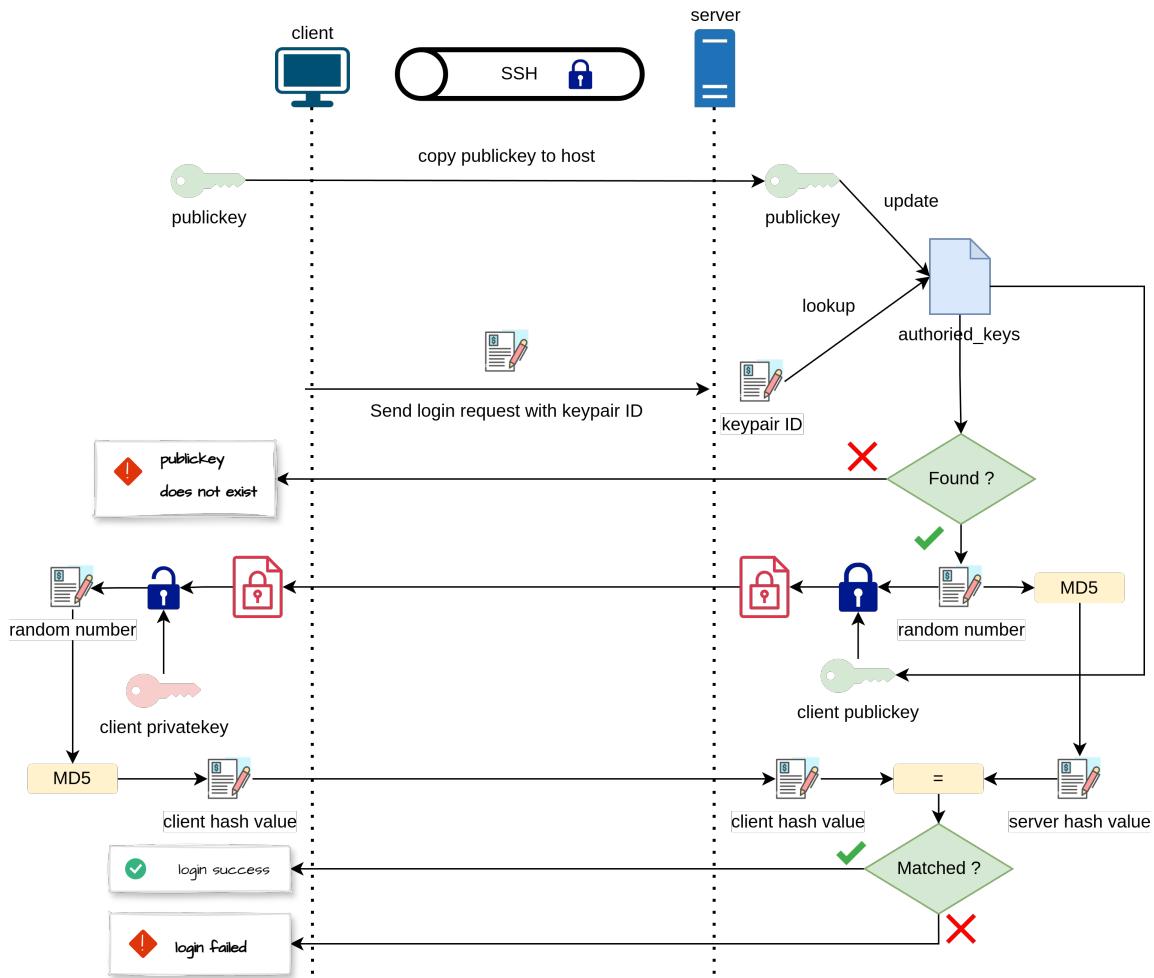
前面提到, 用户身份验证阶段通常分为**用户密码登陆**和**密钥登陆**两种, 这两种方式的验证机制并不相同, 接下来分别解释两种验证方式。

### 用户密码登陆



使用用户密码方式时，客户端发送登陆请求给服务器，告知服务器使用密码登陆的方式。接着用**会话密钥**将用户名，密码等信息进行加密，服务器端收到加密后的信息之后，用会话密钥进行解密得到用户名和密码，并且与本地存储的用户名密码信息比对，若匹配成功则向客户端报告登陆成功，否则客户端继续尝试登陆。

## 密钥验证



使用密钥验证时，客户端发送登陆请求给服务器，告知服务器使用密钥验证的方式，并且会附上密钥对的ID。服务器受到信息后，根据ID查找本地保存客户端密钥的文件（通常是`~/.ssh/authorized_keys`），如果发现了对应的客户端公钥，则服务器生成一个随机数，使用**客户端公钥**进行加密，并发送给客户端，客户端收到信息后，使用自己的私钥进行解密，解密之后得到相应的随机数，接着使用哈希算法（一般是MD5算法）通过这个字符串生成摘要，并发送给服务器。服务器收到客户端的摘要之后，自己也使用相同的哈希算法应用于生成的随机数，和客户端的摘要进行比较，若比较通过，则验证成功。否则客户端登陆失败并重试。

## 其他SSH使用场景

### vscode使用SSH进行远程开发

如果说vscode相比于其他一体化的IDE的优势，那么vscode强大的**远程开发**功能必然榜上有名。

远程开发可以**将本地环境和开发环境隔离**，避免了本地的开发环境配置问题（比如windows配置c++开发环境非常困难），同时解除了本地环境和开发环境的地理限制。

如果你手边没有Linux环境，那么我强烈建议你搭建一个Linux开发环境，尤其是使用windows的同学（macOS同学看个人喜好）。

一方面Linux相比windows，更加适用于搭建开发环境，另一方面，使用Linux环境可以不知不觉中强化自己对Linux命令的使用（你一点不会Linux？那么请移步这一篇）。

不过，在使用Linux的同时，我个人还是建议你手边准备一台windows**以备不时之需**（比如办公，游戏等，更重要的是**预防学校某些实验课要安装的臭不可闻的软件**）

Linux的环境一般有以下几种选择：

- 租一个云服务器（个人不推荐，但是如果你有强烈的公网环境需求或者算力需求或是学校或机构有丰富服务器资源，那么当我说）
- Linux物理机（推荐，如果你觉得自己无法驾驭，那还是算了）
- Linux/windows双系统（都双系统了，还是一步到位物理机了吧）
- windows+linux虚拟机（推荐，但是更建议wsl）
- windows+wsl（推荐，毕竟曾经被称为最好的linux发行版）

如果你比较激进，而且喜欢折腾，或者手边不止一台电脑，那么直接安装linux物理机再好不过。

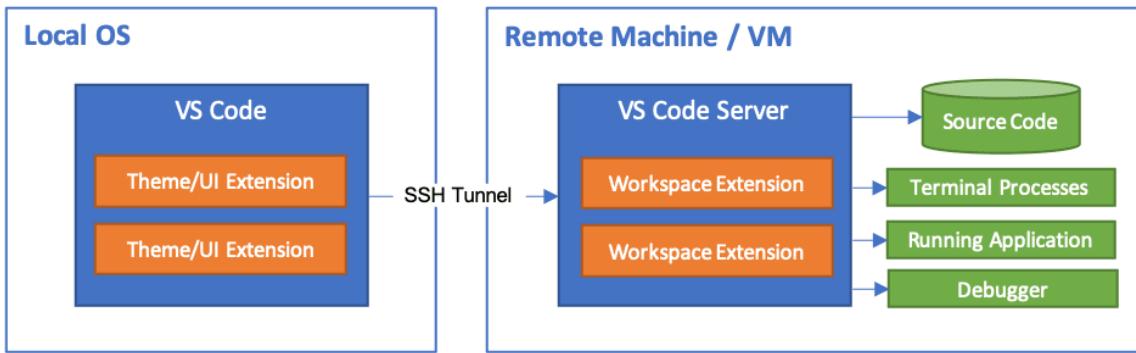
如果你有强烈的windows需求，而且只有一台电脑，那么我建议你使用windows+linux，linux可以使用windows下的虚拟机环境（vmware等），或者wsl2。

如果你决定选择使用vmware等虚拟机软件，那么我建议**不要安装linux的图形界面**，将linux作为一个命令行终端的接口来使用。

常用的方式是使用vscode远程开发功能连接自己的服务器或者虚拟机或者wsl。这样既保留了windows环境，又支持使用linux开发环境。

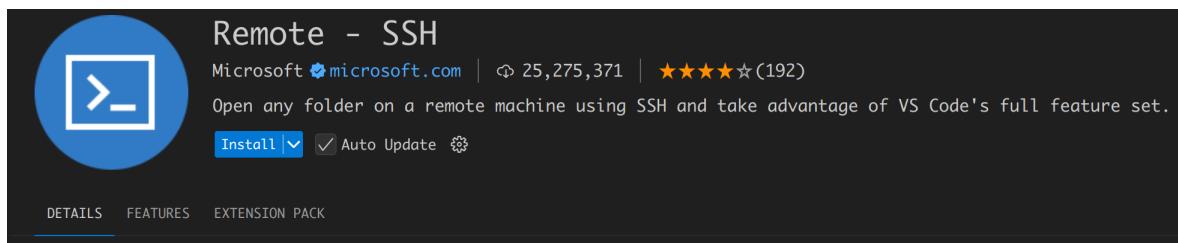
好了，让我们回归正题，如何使用vscode进行远程开发：

vscode远程开发基于ssh连接，ssh在本地vscode客户端和远程vscode服务器上建立了一条连接通道，这里借用[vscode docs](#)的图片来解释：

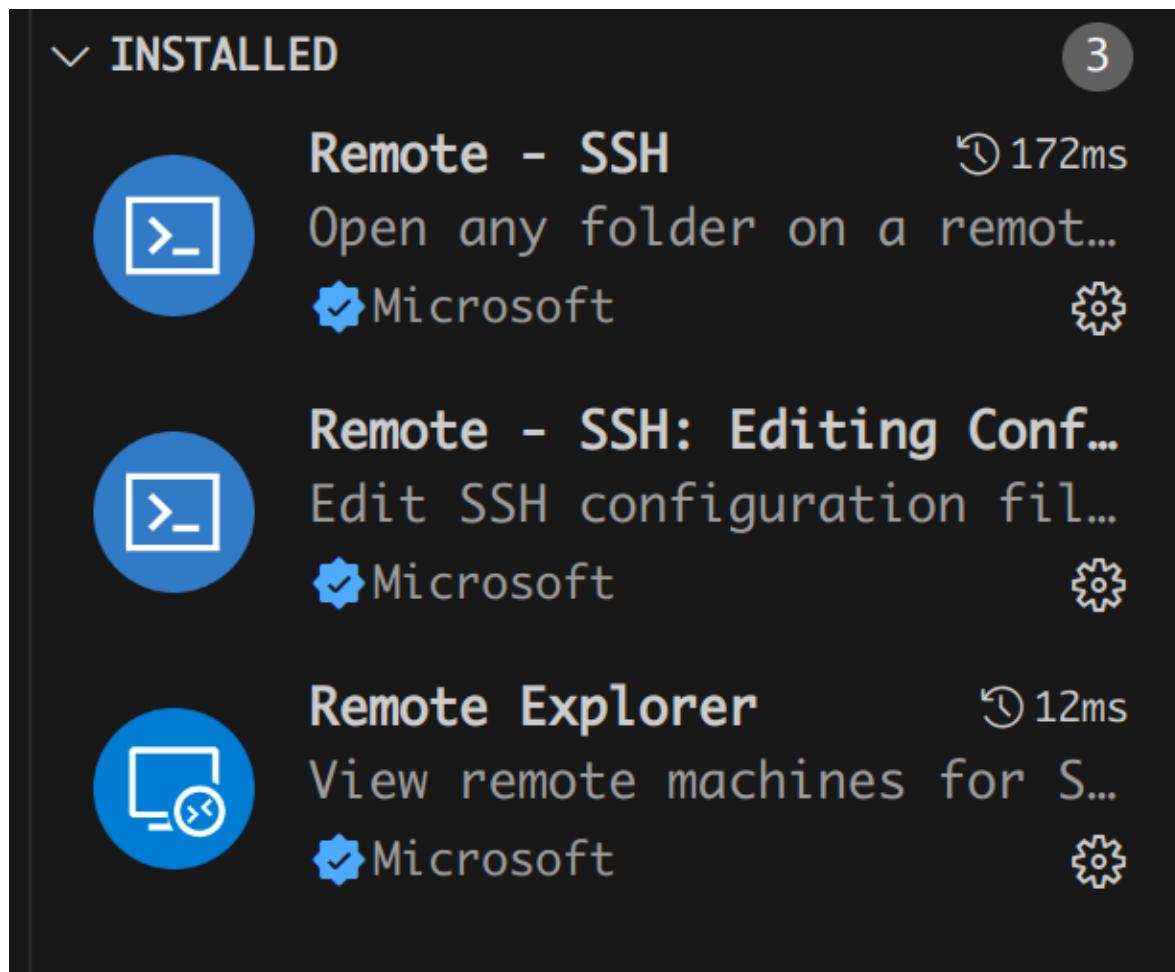


vscode客户端就是本地下载安装的vscode，而vscode server是远程主机上的一个进程，当进行远程连接时，客户端会复用自己的一些主题插件，而服务器会自动下载安装一些开发环境相关的插件。

首先，你需要安装一个名为**Remote-SSH**的插件：



安装完成之后，你的插件列表应该会出现了3个已安装的插件：



第一个插件， Remote - SSH， 帮助你连接到远程主机。

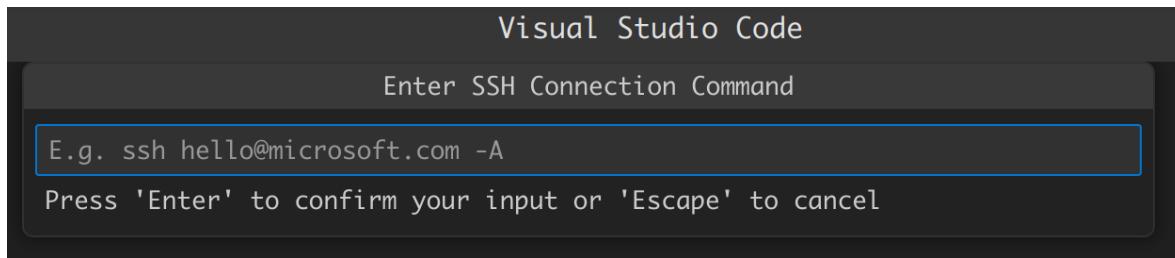
第二个插件， Remote - SSH: Editing Configuration Files， 用于修改ssh配置文件

第三个插件， Remote Explorer， 这个就很牛逼了， 提供图形化界面用于访问远程主机的目录等，让用户感觉自就好像在本地开发一样。

现在你的左侧栏应该会出现一个远程主机的图标， 点开他，在右侧会出现REMOTES的菜单栏， 点开外层下拉框， 里面会出现名为SSH的下拉框， 继续点开SSH， 即可呈现所有的远程主机列表， 这是根据你的SSH配置文件决定的。

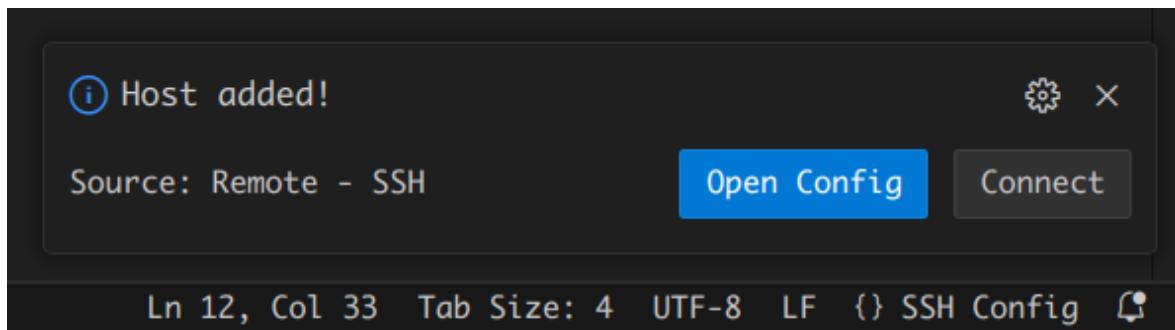
如果你发现你要连接的主机不再列表中， 这时候需要添加远程主机，

将光标移到SSH那一栏， 点击右边的+， 这个是添加远程主机的选项， 然后你的vscode界面上方应该会出现一个下拉框：



这个输入框让你输入ssh的登陆命令，就像在命令行内进行登陆一样。

输入相关的ssh命令，这时候vscode会提示选择需要更新的ssh配置文件，选择自己用户目录下的那个文件（/home/username/.ssh/config），按下enter，然后vscode会提示host added!



这时候打开ssh配置界面，发现vscode已经自动添加了主机的配置文件：

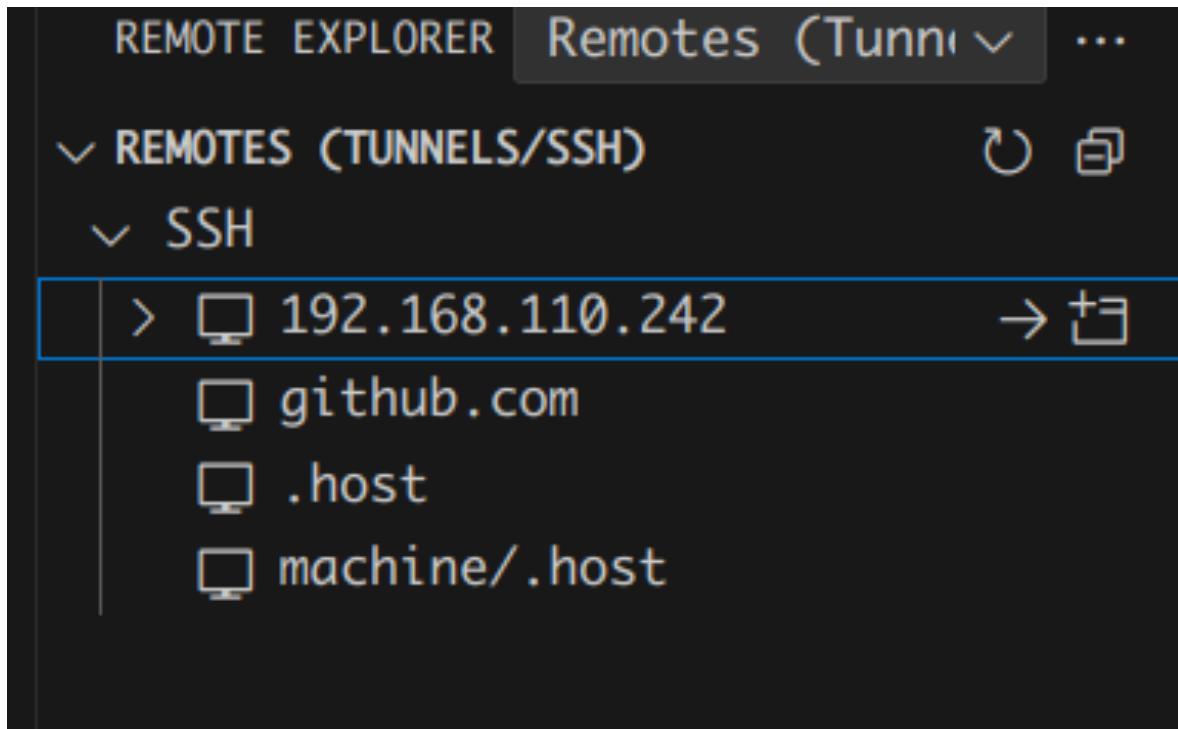
A screenshot of the Visual Studio Code terminal. The text shows the contents of a ".ssh/config" file:

```
.ssh > config
1 Host 192.168.110.242
2   HostName 192.168.110.242
3   User ttang
4
```

The numbers 1, 2, 3, and 4 are likely line numbers from a code editor.

当然，你可以将Host那一栏的ip地址换成自己喜欢的名字，比如myhost。

然后，在左侧的remote exploer里面会出现新的主机：

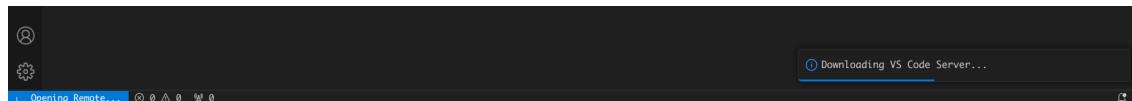


将光标移到目标主机那一行上，右侧会出现 -> (在当前窗口内连接)和 + (新开一个窗口进行连接)，根据你的需求选择一个，

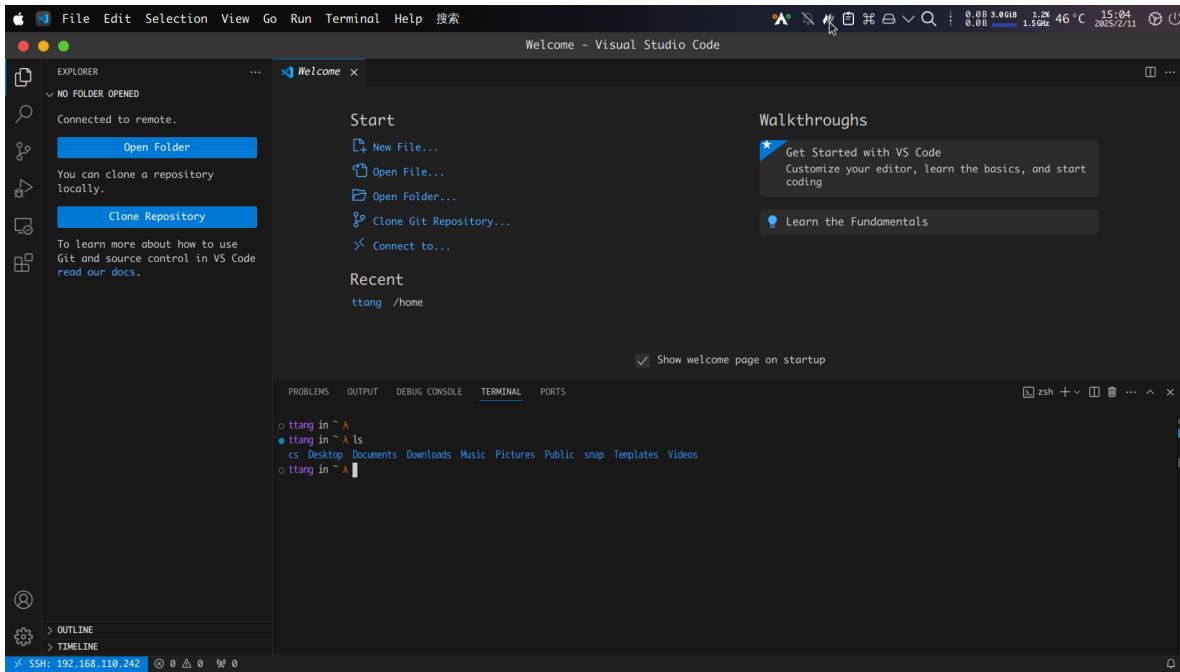
然后vscode会打开窗口，并且弹出一个框提示你输入密码，输入密码之后，vscode则开始连接。

### Note

如果你自己的远程主机上没有安装vscode-server，那么远程连接时会自动安装，并且vscode会弹出如下提示：



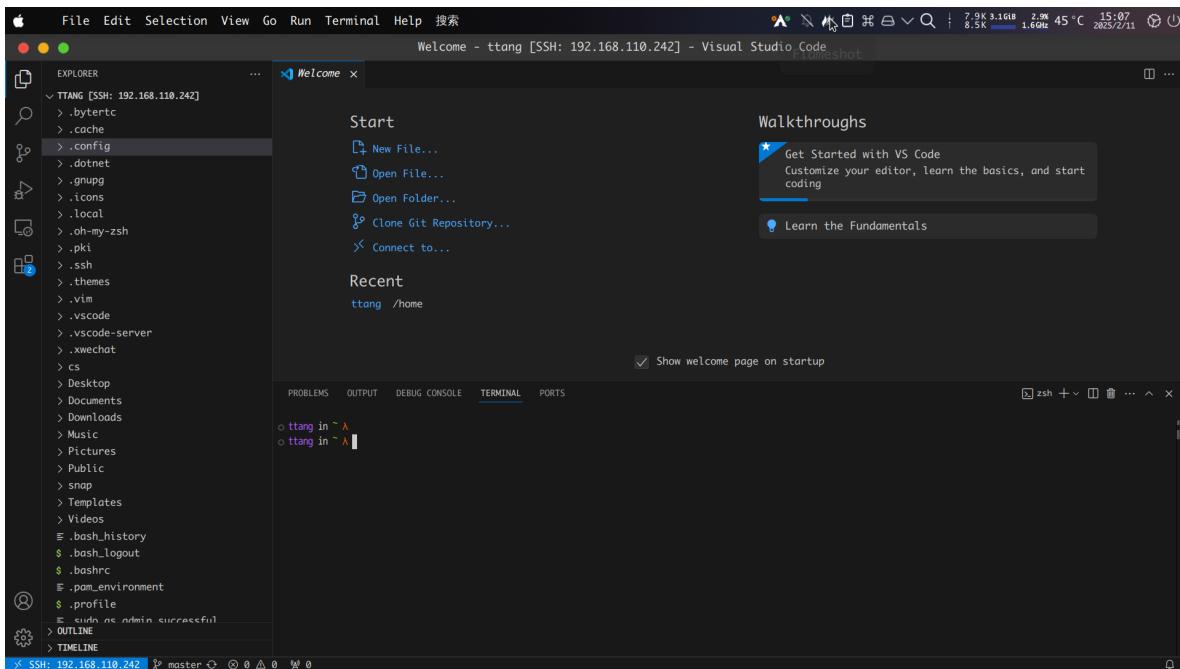
连接成功后在新的窗口内会显示：



观察左下角会显示**SSH: HOSTNAME**, 说明当前已经处在远程环境中, 在左上角工具栏中打开终端, 会显示远程系统中的终端 (即使你的本地环境是windows, 远程主机是linux, 那么就会使用你linux系统下的终端)。

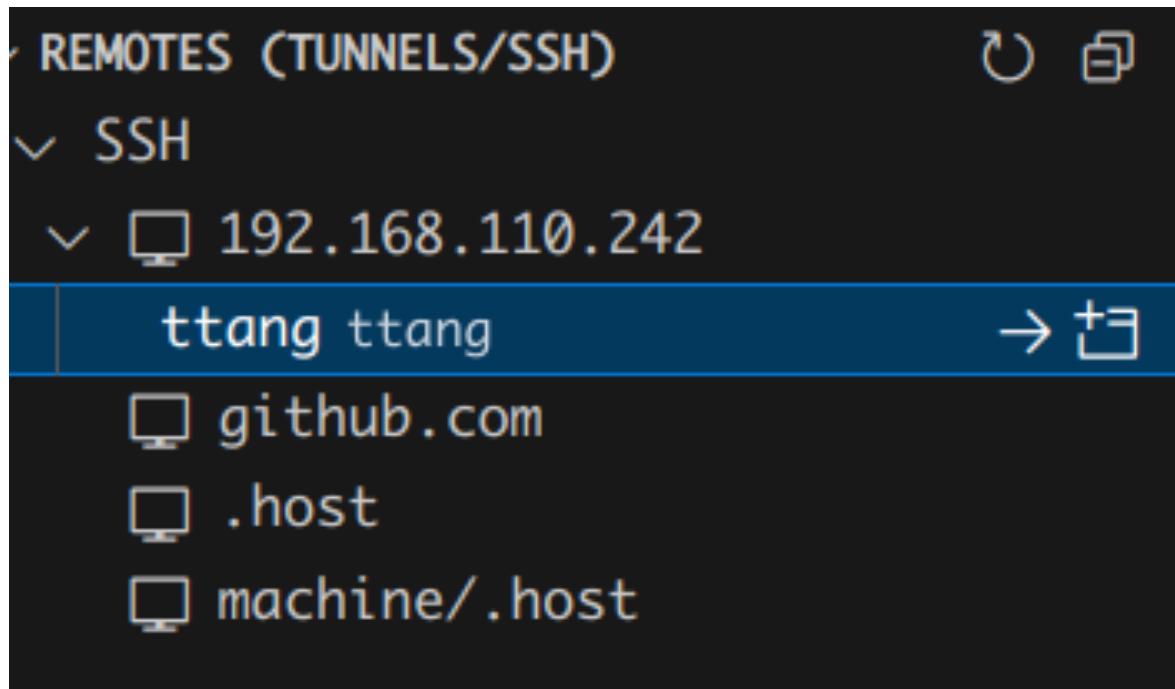
点击左侧的文件图标 (Explorer) 将打开你的远程主机的工作目录, 由于现在没有打开任何一个目录, 因此会提示我们open a folder,

随便选择一个目录, 比如用户的家目录, 然后打开, 远程窗口会变为 :



这样就和本地开发几乎一模一样了。

当然，vscode会帮我们记住上次打开的目录，这样下次连接就可以直接打开对于的远程目录了。



上述方式连接每次都要输入密码，如果想改成使用密钥登陆，也非常简单，只需要将ssh关于远程主机的配置修改为：

```
Host myhost
  HostName 192.168.110.242
  User ttang
  IdentityFile "~/.ssh/id_rsa"
```

注意将私钥路径改成自己的，这样就可以使用密钥进行登陆了。

### ⚠ Warning

使用免密登陆之前一定确保将公钥正确的上传至远程主机，上传的方法见SSH密钥登陆那一节，如果没上传公钥，vscode还是会提示输入密码进行登陆。

## Github使用SSH

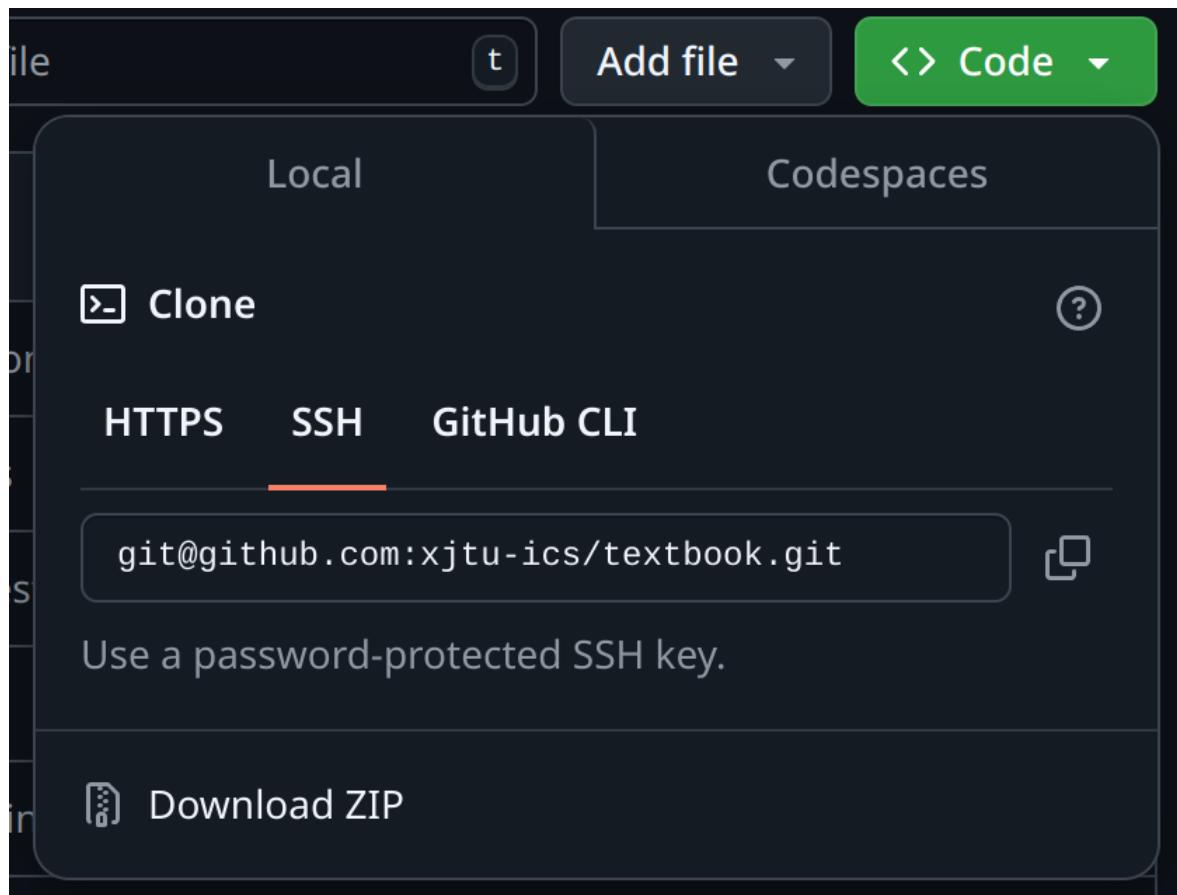
到了这个阶段，相信你已经接触过了github（什么？你没用过github，那么请移步[这一篇](#)），且已经学会了科学上网（不知道什么是科学上网？那我没法教你了，自行搜索或者问同学吧，或者带杯奶茶来面基助教也是可以的）。

出于众所周知的原因，大量的境外网站我们是无法访问的，github作为一个例外（可以不挂梯子直接访问，但是由于DNS污染，以及最近网速肉眼可见的变慢，建议还是使用梯子访问），给我们提供了天然的中转地，因此这一部分教大家如何使用代理和ssh方式访问github。

Github作为全世界有名的～同性交流网站～代码托管平台，可以和git搭配使用从而进行十分方便的代码托管和版本控制。

这其中最重要的两个操作莫过于 `git pull` 和 `git clone` 以及 `git` 等。前者从github拉取代码或者下载完整副本，后者将本地仓库代码上传至github远程仓库。

github对于上述操作提供了两种访问方式，分别是[https](https://)和[ssh](ssh://)。比如，当我们需要clone一个仓库到本地时，点击仓库右上角的code，会分别提供https和ssh的链接



将上述链接复制，然后输入：

```
git clone git@github.com:xjtu-ics/textbook.git
```

即可复制远程仓库到本地，并且在本地自动关联远程仓库。

在不使用代理的情况下，实测发现使用https的成功率可以忽略不计，使用ssh的成功率则高很多（最近不知道什么原因，ssh的成功率也可以忽略不计）。当然，当你无法使用代理时，你也可以点击Download zip手动下载压缩包并解压。不过，既然能使用代理解决，何必多此一举呢。

下面我们教大家如何设置https的代理和ssh的代理，并且更推荐使用ssh。

假设你已经准备好了代理，不管用什么方式，首先记录下代理的端口，比如这是我笔记本上的：



使用http协议的端口为20171，socks5协议端口为20170。

如果要使用https与github进行交互，那么代理几乎是必挂的，使用以下两条命令为git添加github的代理（更推荐使用socks5代理，如果使用http代理，则将代理url的协议名改成http）

```
git config --global http.https://github.com.proxy socks5://127.0.0.1:20170  
git config --global https.https://github.com.proxy socks5://127.0.0.1:20170
```

上述方式指定了为域名https://github.com设置代理，其他域名的流量不走代理，因此是推荐的做法。

如果你嫌麻烦，那么设置全局代理也是可以的：

```
git config --global http.proxy socks5://127.0.0.1:20170  
git config --global https.proxy socks5://127.0.0.1:20170
```

不过这种方式将git的全部流量都从代理进行转发，当你用git访问一些不需要挂代理的网站（比如

gitee等），就需要关闭代理，特别麻烦，因此并不推荐这么做。

如果要取消设置，使用：

```
git config --global --unset http.https://github.com.proxy  
git config --global --unset https.https://github.com.proxy
```

如果你设置的是全局代理，那么这样取消设置：

```
git config --global --unset http.proxy  
git config --global --unset https.proxy
```

查看一下git的设置：

```
git config --list
```

如果包含类似下面两行，说明代理设置成功了：

```
http.https://github.com.proxy=socks5://127.0.0.1:20170  
https.https://github.com.proxy=socks5://127.0.0.1:20170
```

只要代理工作正常，基本上就可以使用https进行clone了，且速度还挺快的。

不过，使用https的方式，**每次和github交互都需要输入密码**，十分不方便且安全性也不高，因此更加建议使用ssh进行连接。

ssh的使用和ssh远程登陆的流程差不多，首先在本地生成ssh密钥对，然后将公钥上传至github。

进入github官网 -> 右上角自己头像 -> 下拉菜单选择**设置** -> 左边栏选择**SSH and GPG keys** 进入SSH key管理页面：

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

**hbstack**  
SHA256: EX86eYf1SBhEPBeU6TfkhGGmK6nXNjurj/vQddKo60  
Added on Dec 4, 2024  
Last used within the last 3 months — Read/write

**ttang host**  
SHA256: zkqyjVYdf0cWQtkhHe1kF1eVbU9SF8Nzng2SML51BQ  
Added on Jan 18, 2025  
Last used within the last 4 weeks — Read/write

**ttang-desktop**  
SHA256: WEu57cJoz9G0hRy9J1YFRzizpfUbUNDp1pEhYjkT4j4  
Added on Feb 5, 2025  
Last used within the last week — Read/write

Check out our guide to [connecting to GitHub using SSH keys](#) or troubleshoot [common SSH problems](#).

点击右上角**New SSH key**, 随便输入一个title, 将自己本地的公钥内容复制到key里面（注意key的格式要求, 不要复制少了！一般将 `id_rsa.pub` 文件的内容全部复制就行）：

Add new SSH Key

Title  
ttang-desktop

Key type  
Authentication Key

Key  
ssh-rsa  
AAAAAB3NzaC1yc2EAAAQABAAQACQDBcNzR84bDagy43Rga0MiLjjLv9uHUseoqouryR2JMNRVx0yyf5W6xfX8jpO4ipjgo5u7LWjNFXLH  
mEM05BgYS3XYHmKn0FOUNAQIA4b8M2MVF9MEvMOrbYdY2UR1iaN3BXcTX3jGUjB82yep1822oUAPv2Xa7MZo2XIgoanw0ZbVKjVjz4g/4  
g9rmmniTEA1nW3IpHTbNA6YuqLsGsdvUjdI9wTlgl9A/NNOnaB+5yfLE1ekkzYZZhjhAuAREFwU/Vf4o1Qj3Dte2BrTmYQHfYTl2G/wpa9WA  
egx9QPBWx7c2xZHgYQY5lrTvgkyzQNIIdRoNIMic2IVIauuHyf/GGZISUsb8OQkCYCyg1vNp2hynvlj6Fefh1scRUGx2YWeS9Kb  
LSiSuVnu3/KctdxG6Cbdk75cJCNsgVvdLi7ZmFisdnruUQOlukZR8F2LPjSMWNlg8FUzd/vFHhOSUW2urg6iMWzfNfw19LjcDDqy2Fm9IEb8  
B4m1P098D963yqRwrZY3sMsTzlbNkiX9te0WFfIwGHtvRj9BjAIa9ZW3JS53UBkFPoxzb0UWVfGqXv9uMjtMievXBvpZowgdT9cM/peogN71tQ  
Nb1P0AlMwLoH9j26X+FiwHM9EC3GnulzU0iOhBqTksV3c+okPEWcEIj9szj7O/w== ttang@ttang-desktop

Add SSH key

最后点击**Add SSH key**, 退回到ssh key管理界面, 就可以看到自己最新添加的公钥了。

接着修改SSH配置文件, 加入下面的内容：

```
Host github.com
  HostName github.com
  User git
  # using socks5 proxy
  ProxyCommand ncat --proxy 127.0.0.1:20170 --proxy-type socks5 %h %p
  # private key path(change it to your own path)
  IdentityFile "~/.ssh/id_rsa"
```

具体解释一下代理设置的那一行：

- `ProxyCommand` 选项表示使用代理，SSH连接时会执行这个选项后面的命令
- `ncat` 是一个命令行工具，具体根据自己的linux发行版进行安装，可以指定使用代理连接特定的服务器
- `--proxy` 指定本地代理服务器，实际使用时替换成自己的服务器地址，尤其是端口号
- `--proxy-type` 指定代理类型，可以使用http或者socks5
- `%h %p` 是占位符，表示主机名和端口，实际连接时会自动替换成ssh的目标主机名和端口号

修改完成后，使用 `ssh -T git@github.com` 进行测试，如果出现类似以下结果，则表示ssh连接成功：

```
Hi Scorpicate! You've successfully authenticated, but GitHub does not provide
shell access.
```

有些环境中，比如某些企业的内网会设置防火墙阻止ssh流量通过22端口进行转发，万幸的是，github提供了一种方式可以使用https端口，即443端口建立ssh连接。

首先测试443端口是否可行：

```
ssh -T -p 443 git@ssh.github.com
```

注意，443端口的github域名为 `ssh.github.com`，使用时注意修改。

如果出现类似以下内容，说明端口有效：

```
Hi Scorpicate! You've successfully authenticated, but GitHub does not provide
shell access.
```

否则的话，查阅[Github官方文档](#)进行错误排查。

于是，当需要使用ssh的时候，将github的地址 `github.com` 改为 `ssh.github.com` 即可，比如：

```
git clone ssh://git@ssh.github.com:443/xjtu-ics/textbook.git
```

不过每次clone时修改url显然是一件很烦的事，可以修改ssh配置文件来避免：

将配置文件改成以下内容：

```
Host github.com
  HostName ssh.github.com
  Port 443
  User git
  # using socks5 proxy
  ProxyCommand ncat --proxy 127.0.0.1:20170 --proxy-type socks5 %h %p
  # private key path
  IdentityFile "~/.ssh/id_rsa"
```

这样设置强制ssh每次连接通过443端口来进行，因此可以绕过防火墙。

上述不过是ssh连接github的冰山一角，更多信息请咨询[GitHub Docs](#)

---

© 2025. ICS Team. All rights reserved.

# Git

---

© 2025. ICS Team. All rights reserved.

# Google Style Guide

---

© 2025. ICS Team. All rights reserved.

# Contributors

## Core Developers

We would like to express our heartfelt thanks to all the contributors who have helped improve XJTU-ICS! Your efforts and collaboration make this project better every day .

Special thanks to **XJTU-ICS Team Members**. Big shout-out to them :

- [Danfeng Shan](#) (Professor)
- [Hao Li](#) (Professor)
- [Yunguang Li](#) (HEAD TA, Master)
- [Orion](#) (TA, Senior)
- [Tang Tang](#) (TA, Senior)
- [Jinnuo Du](#) (TA, Senior)
- [Jiajun Luan](#) (TA, Senior)
- [Jinyu Fu](#) (TA, Senior)
- [Boxuan Hu](#) (TA, Junior)
- [Yike Liu](#) (TA, Junior)
- [Yuxuan Li](#) (TA, Junior)

You can view the full list of contributors and their contributions [here](#).

If you feel your contributions are not listed, please feel free to open a pull request to add yourself. Your work is greatly appreciated !

## How to Contribute

We welcome contributions from everyone! If you're interested in contributing to XJTU-ICS, please follow these steps :

1. Fork the repository to your own GitHub account.
2. Clone the forked repository to your local machine.

3. Create a new branch for your changes.
4. Make your changes and commit them with clear and concise commit messages.
5. Push your changes to your forked repository.
6. Open a pull request to the main repository.
  - Since *Rule Set* is in use, you must contribute with a PR!
7. Delete your branch after merging it. This keeps the repo clean and faster to sync.

It's always a good habit to check [CONTRIBUTING.md](#) before contributing !

If you are curious about how to set up a textbook with mdBook, we have also provided a [detailed tutorial](#) for you to start.

---

© 2025. ICS Team. All rights reserved.

