# 深入理解计算机系统

Computer Systems - A programmer’s perspective

# 背景

豆瓣9.7

这本书是用C语言作为example的，所以还是非常不错的。我们可以借助这个机会，提升一下C语言的熟练度。后续无论是开发网页版控件，还是参与算法，还是参与开源项目，大概率会用到C语言。

## Why

我们自然而然会问这个问题，在AI、AR/VR、元宇宙、微服务这些热门的应用大行其道的时候，为啥还要"深入理解计算机系统"？作者在第一章开篇就解释了原因：

### 1.学习使用技能

通过学习本书，可以学习一些实用的技能，比如涉及底层的软件异常

学习各种编程技巧，包括shell脚本、动态内存分配技术、webserver

### 2.优化C程序。

基于现代CPU、内存模型，对C程序进行有针对性的性能优化

### 3.了解编译原理

为我们学习编译原理打基础

### 4.帮助我们学习Java

我们在生产环境会碰到各种涉及底层的Java问题，比如的托管accps-convert JVM崩溃的案例，针对这个案例，有一些相关的网文参考文章，比如《如何通过⽇志分析解密JVM Crash？》。这些生产案例、网文，都会涉及一些底层的内容。为了解决这些Java相关的问题，我们可以借助本书。如果没有系统学习过这些底层的知识，这篇网文就像天书一样无法理解。

### 5.有助于我们理解涉及底层的网文内容

我们调研了很多公众号网文，虽然大部分网文都是停留在应用层的，但也有一部分涉及了底层。我们随便举一个例子：来自字节跳动技术的《深⼊理解 Linux 内核--jemalloc 引起的 TLB shootdown 及优化》这篇网文涉及到了大量内存的知识，包括页表page table、内存管理单元MMU、进程间中断IPI等等知识，这些内容在《深入计算机系统》中深入、详细的讨论。要理解这类深入内核的问题，是不是必须要掌握书中的体系知识？

类似的例子还有很多很多：  
《Linux 系统中编译、链接的基石 - ELF 文件：扒开它的层层外衣，从字节码的粒度来探索》

《GCC 链接过程中的【重定位】过程分析》

《高级语言的编译：链接及装载过程介绍》

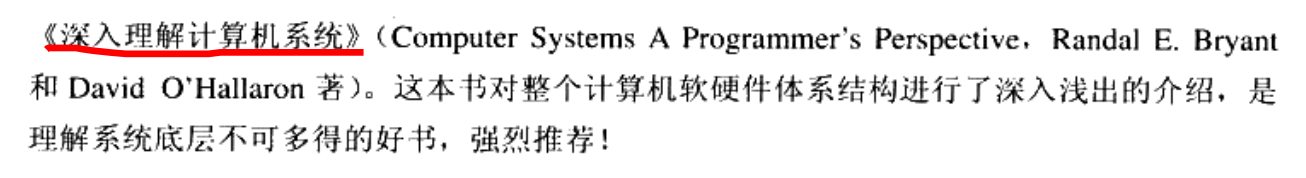
......

我们不能一一列出。这些文章，单独拿出来，讲了一个知识点，比如编译、链接、重定位、ELF...如果读了本书，就知道了这些知识点在整个计算机体系下处于哪个位置。

### 其他

我们还可以加入更多的理由，比如：

1.《程序员的自我修养/ 链接、装载与库》作者强烈推荐这本书



2.在鸿蒙系统快速发展的时代，我们也慢慢意识到了基于软硬件的平台的重要性。安卓、iOS的成功，移动互联网带来的颠覆性的变革，给我们带来了深刻的启发。要参与到鸿蒙系统发展的伟大时代，学习这种底层的知识是非常必要的。

3.其他原因我们在《读书笔记-Linkers and loaders》中已经做了总结，在这里就不再赘述了。

4.防止各种网络攻击

5.理解底层原理

参考《极速优化：⼗倍提升JS代码运⾏效率的技巧》

这篇网文介绍了如何从JS引擎(V8引擎)提升js代码运行小的技巧，涉及了编译原理、字节码、机器码、运行时、对象模型等内容，这些内容在这篇网文的背景下可能非常复杂，有些难以理解。但是在本书中，这些概念都是基础概念。所以理解本文有助于我们理解这类涉及底层的网文。那么了解chrome V8 js引擎有什么意义呢？这类内容非常底层、非常难，但是一旦形成了成果，就会有很深的护城河，比如chrome之于谷歌，连微软都要退避三舍。

## 各种感想

1.C语言的重要性

现在知道为啥一上来就要学习C语言了吧？即便在工作中、项目中用到C很少(因为我们一般都是在应用层游荡的CRUD boy)，但是，我们说C/C++还是很重要。这是为啥呢？看了本书就明白了，其中一个原因就是，要介绍本书这种偏底层的内容，用C语言这种天然和底层联系比较亲密的语言，是比较合适的。

2.浏览目录

光是把本书的三层目录浏览一遍、罗列一遍，就需要花费很长的时间。足见本书整体架构之宏大、体系之完整、细节之深入。什么是体系？这就是体系！这本书涵盖了计算机这个学科大部分的内容。

我们要借此书好好体会。

3.看这本书，有一种魔术解密的爽快感。

随便举一个例子，我们知道linux下的管道操作，很神奇对不对：

ls > a.txt

这个是怎么实现的呢？

我们看"10.9 I/O Redirection"就明白原理了：

实现原理就是依托Linux提供的 dup2() function

这个function的实现原理是，在descriptor table中作一下调整：把目标文件的PID也指向源文件，这样目标文件的输出看起来就和源文件的输出保持一致了(本来就已经是同一个文件了)。这个小节中有一幅形象的图来说明这个原理。是不是很神奇？如果脱离这个体系，我无法想象要怎么去清晰、形象地解释管道的原理。

## Reference

本书还提供了很多参考书目，我们罗列在这里

### 1.Intel® 64 and IA-32 Architectures Software Developer's Manual

豆瓣9.5

# 如何读这本书？

## 1.第一遍 罗列目录

罗列本书所有的目录，了解各个章节的大致内容，无需看详细内容（已完成）

## 2.第二遍 快速浏览

快速浏览各个章节，总结一下各个章节的内容，了解重要概念/感兴趣的主题即可。不用深入分析细节，也不用代码实践，但是建议把书中的代码列出来，比如mem.c这种，搞懂代码大致意思，有助于理解内容。

（正在做）

## 3.第三遍 精读 + 代码实践

精读本书的内容，在精读的同时，要做好代码实践，把书中的代码实践一遍。书中的示例代码主要是C语言、汇编语言。

这个步骤我觉得是最重要的，毕竟代码实践最重要。

## 4.第四遍 习题

做习题。把每个小节后面的作业写一下。

## 5.实践

尝试利用本书的知识，实践一下，开发一个小工具软件

## 6.扩展阅读

以本书作为整体框架、脉络，阅读一些扩展读物，比如《编译原理Compilers》 《Linkers and loaders》、《程序员的自我修养/ 链接、装载与库》。扩展一些网络文章，比如一些涉及底层的网文。

# 代码实践

## 代码目录

书中有很多C语言形式的demo，我们也都实践一下，放在这个目录：

/Users/zhoushuo/Documents/Read/深入理解计算机系统/code

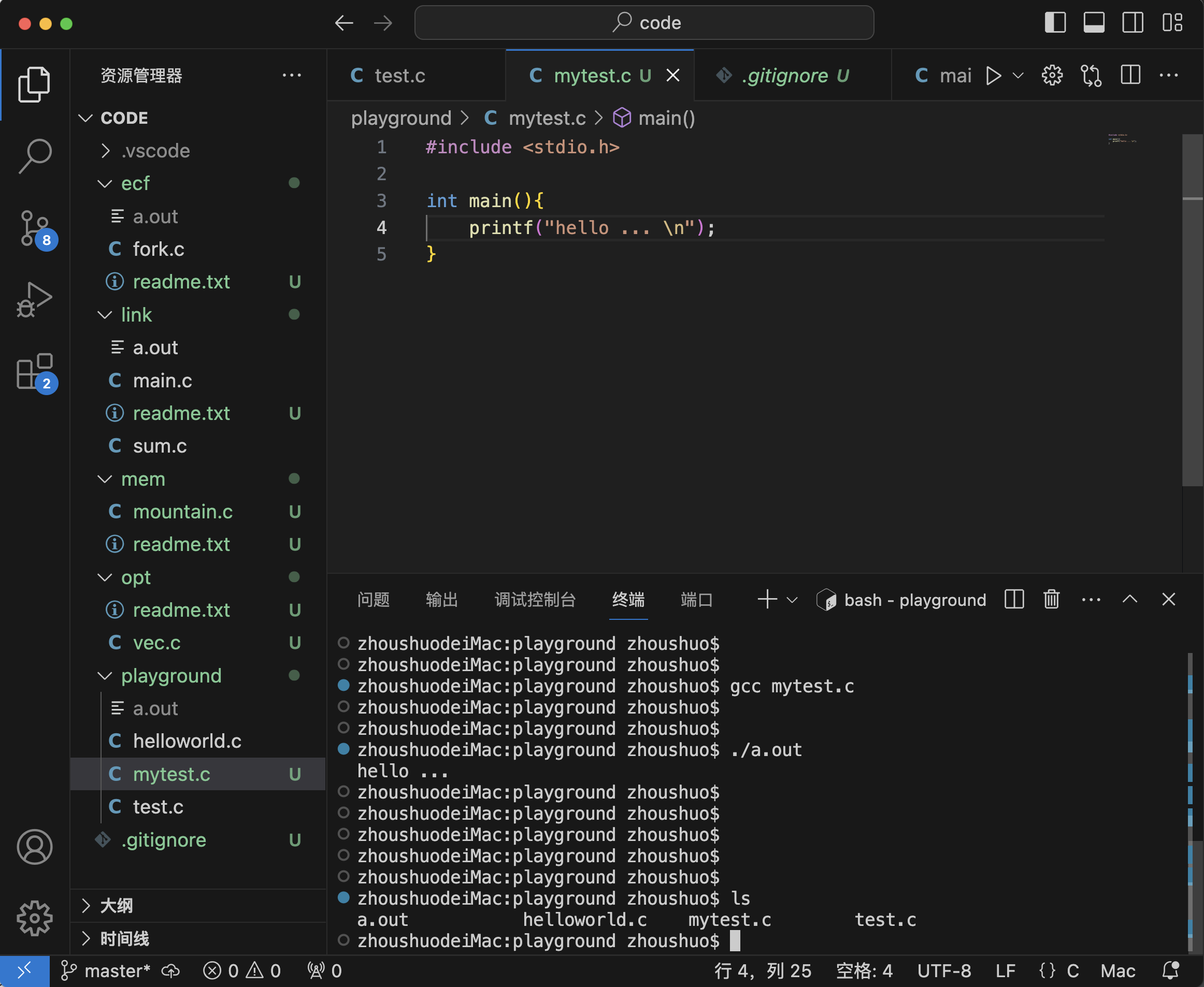
备注：后续建议放到gitee上

代码实践是非常非常重要的。我们学习本书，一个重要的原因就是优化我们的程序代码。

## 代码操作说明

通过vs code编辑器打开code目录

然后在集成终端中编译、执行代码就行了：



# Q & A

## 1.虚拟机的实现原理

看完本书，是否能够了解虚拟机的原理？

## 2.了解JVM的实现原理

我们知道，JVM也是那种提供了类似汇编语言的指令，那么看完本书，是否也能够了解JVM的实现原理？

## 3.了解容器的实现原理

通过对计算机体系的底层，分析，是否有助于我们了解容器的实现原理？

## 4.程序执行的整体流程

一段程序，从文本，到被编译为可执行文件，到执行，整体的流程到底是怎么样的？

我们要整理一下。我相信看完本书，应该能够梳理出来。

具体到实践层面，给你一个CPU、一个内存、一个磁盘，然后你能否将这段代码执行成功？

## 5.从底层硬件角度解释一下，为什么在大模型AI时代，GPU比CPU更加适用？

# 后续扩展

## 1.扩展到其他知识

比如我们一直很关注的《The Linux Programming Interface》

之前看起来比较吃力，现在有本书作为铺垫，应该会轻松一下；

## 2.尝试技术输出

这个是我们一直在设想，基于本书的铺垫，能否将计算机的整个体系，以一种比较轻松愉快的方式，输出一下。

3.

# Chap1 A Tour of Computer Systems

对计算机系统做一个概览。

## 1.1 Information is Bits + context

1.1说明了程序的本质：就是由bytes组成的一段内容。所谓的context，就是程序可以是保存在磁盘中，可以加载到内存中，也可以在网络中传输，这些环境就是所谓的context。程序的本质其实就是在某个context中的bytes。这真可谓是一针见血。

## 1.2 Programs are translated by other programs into different forms

程序的本质2：程序A由程序B转化为其他形式。这个也非常形象，比如C程序由GCC程序(经过编译、链接等步骤)转化为各种形式(比如object file/machine code等等)。

这个小节详细解释了hello.c程序如何转化为可执行程序的步骤

## 1.3 It pays to understand how compilation systems work

这个小节特地解释了为啥要关注程序编译的过程。

包括三个原因：  
1.优化程序性能

2.解决编译、链接时的异常

3.避免各种安全漏洞

## 1.4 Processors read and interpret instructions stored in memoty

CPU从内存读取并解释那些指令。

这个小节简单介绍了我们执行./hello的实现原理

## 1.5 Cache Matter

为了提升CPU执行效率，加入了cache，就是我们之前了解的CPU L1cache L2cache...这些。所谓的cache，就是把内存中的内容缓存到这些L1/L2Cache中。

## 1.6 Storage Device form a hierachy

## 1.7 The Operating System Manages the hardware

这个小节介绍了操作系统，介绍了操作系统在执行一个程序时扮演的角色。

### 1.7.1 process

进程

### 1.7.2 thread

线程

### 1.7.3 virtual memory

virtual memory给某个程序提供了一段看起来专属的内存空间，供程序使用。virtual memory包括heap/stack/shared lib等等。

### 1.7.4 files

保存程序文件、各种数据、IO

## 1.8 Systems communicate with other systems using networks

这个小节介绍了网络

## 1.9 important themes

这个小节介绍了一些重要主题

### 1.9.1 Amdahl's law

这个规则的意思是这样的，比如我们优化了系统中某个组件的性能，那么这个优化对于整个系统的性能提升有多少帮助呢？这取决于这个组件在整个系统中的作用。

我们随便举一个例子，比如我们优化某个后台服务，把服务耗时从100ms优化到了50ms。但是因为这个后台服务发布在公网上，受限于公网的网络条件，这个服务的网络耗时就需要2秒，那么我们对于服务的优化(50ms)，在整个服务性能提升方面作用非常有限(2.5%左右的提升： 0.05/2)。如果这个服务发布在内网，网络耗时只要100ms，那么这个提升就非常明显(25%左右的提升: 1 - 150/200).

这个规则的意思也非常明确，就是要优先去优化那些重要的模块，不要纠结在较小的部分。

### 1.9.2 concurrency and parallelism

并发相关

### 1.9.3 the importance of abstraction in computer system

API对于系统的重要性，其实就是类似java中的接口。在实现一些具体的功能之前，先指定好接口。

## 1.10 Summary

总结

# Chap2 Representing and Manipulating Infomation

计算机系统中，信息是如何表示、如何操作的。比如int/float是如何表示、如何操作、如何保存的

## 2.1 Information storage

各种字段类型(数字、String、Boolean)是如何保存的

计算机把1 byte(8 bit)作为内存最小单元

### 2.1.1 Hexadecimal Notation

先引入了16进制

机器码就是在二进制、十进制、16进制之间转换

16进制占据了4个bit: 0000

用十进制表示就是:0 1 2 3 4 5 6 7 8 9 A B C D E F

我们经常看到内存地址使用16进制表示的，比如：

0xD5E4C

备注：这里可以写一个小程序作为练习：在十进制、二进制和16进制之间转换。

### 2.1.2 Data size

数据类型的长度

(1)VM的地址长度

一上来先介绍的是VM的地址长度，这个长度就是我们经常说的32位系统、64位系统。

这里清楚告诉我们这是什么意思，比如32位，意思就是VM中的地址长度32 bit，每个地址对应一个内存byte的位置。那么在32位系统下，内存最多包含2的32次方个byte,大约位4 \* 10的9次方，大概是4GB。

64位，意思就是VM中的地址长度64 bit，最多包含的字节数就远远大于32位了(2的64次方)。

(2)介绍了C语言中常见数据类型的长度

比如int 是4个字节(32 bit)

### 2.1.3 Addressing and Byte ordering

地址和字符顺序

作为的byte ordering，就是little-endian小头和big-endian大头。

小头就是从右往左，大头就是从左往右。

程序实例：

show\_bytes.c

就是展示各种数据类型如何用byte展示。

支持展示int/float/pointer

### 2.1.4 Representing Strings

如何表示string

通过上面的show\_bytes.c代码，用16进制的形式展示了一段string，比如“hello”，用16进制表示就是：

68 65 6c 6c 6f

### 2.1.5 Representing code

代码在计算机中是如何表示的。

这里有一段示例程序：

就是一个sum()函数

int sum(int x, int y){

return x + y;

}

1.源码转汇编

gcc -Og -c sum.c

2.通过objdump命令再展示汇编内容

objdump -S sum.o

备注：

(1)这里的-S参数是能够展示汇编指令对应的十六进制

(2)如果直接通过vi打开汇编代码程序文件的话，就只有单纯的汇编内容

具体的汇编指令和对应的十六进制代码如下：

0: 55 pushq %rbp

1: 48 89 e5 movq %rsp, %rbp

4: 8d 04 37 leal (%rdi,%rsi), %eax

7: 5d popq %rbp

8: c3 retq

所以，这段代码在计算机中标识如下：

55 48 89 e5 8d 04 37 5d c3

### 2.1.6 Introduction to Boolean Algebra

布尔运算

### 2.1.7 Bit level operations in C

C语言中的位运算

### 2.1.8 Logical Operations in C

C语言中的逻辑运算

### 2.1.9 Shift operations in C

C语言中的位移运算

## 2.2 Integer representations

Integer类型是如何表示的

### 2.2.1 Ingeger data types

### 2.2.2 Unsigned Encodings

### 2.2.3 Two's-complementation encodings

### 2.2.4 Conversions between signed and unsigned

### 2.2.5 signed versus unsigned in C

### 2.2.6 Expanding the bit representation of a Number

### 2.2.7 Truncating numbers

### 2.2.8 Advice on signed verus unsigned

## 2.3 Integer Arithmetic

Integer类型的各种计算

### 2.3.1 Unsigned Addition

### 2.3.2 Two's-complementation addition

### 2.3.3 Two's-complementation negation

### 2.3.4 Unsigned multiplication

### 2.3.5 Two's-complementation multiplication

### 2.3.6 multiplying by constants

### 2.3.7 dividing by power of 2

## 2.4 Floating point

浮点数相关

### 2.4.1 Fractional binary numbers

### 2.4.2 IEEE Floating-point representations

### 2.4.3 Example numbers

### 2.4.4 Rounding

浮点数的四舍五入

### 2.4.5 Floating point operations

### 2.4.6 Floating point in C

## 2.5 Summary

这章看起来很枯燥，但是基础字段类型，比如integer/floating point ，以及对应的运算，是我们在编程中没法避免的，也是计算机的基本功能。

# Chap3 Machine-Level Representation of Programs

介绍机器码，计算机程序如何用机器码表示，如何通过机器码执行计算、循环、数组等操作

## 3.1 A historical perspective

回顾历史上各个版本的CPU，因为机器码是和CPU相关的

## 3.2 program encodings

程序是如何编码成机器码的。

这里介绍了一个例子，如何通过gcc命令把C程序转化为机器码(汇编程序)

### 3.2.1 machine-level code

### 3.2.2 code examples

引入一个代码示例，说明如何通过gcc命令把C语言转化为机器码(汇编程序)，具体执行方式：

gcc -Og -S msstore.c

生成文件为： msstore.s

里面是汇编代码： movq这类

更近一步，可以把通过gcc命令把C语言转化为目标程序，具体执行方式：

gcc -Og -c msstore.c

生成文件为： msstore.o

里面是二进制代码

可以通过objdump命令，再把目标代码再次转化为汇编代码：

objdump -d msstore.o

### 3.2.3 notes on formatting

## 3.3 Data formats

数据格式

介绍了各种基本数据类型，比如int/char/short/char \*/float在汇编语言中是如何表示的。

## 3.4 accessing information

访问信息

意思就是在汇编语言中如何访问信息，包括操作、移动、堆栈操作，等等。

### 3.4.1 operand specifiers

### 3.4.2 data movement instructions

### 3.4.3 data movement example

### 3.4.4 pushing and popping stack data

## 3.5 Arithmetic and logical operation

机器码的算数操作、逻辑运算，就是各种加减乘除运算

3.5.1 load effective address

3.5.2 Unary and Binary operations

3.5.3 shift operations

3.5.4 discussion

3.5.5 special Arithmetic operation

## 3.6 Control

各种控制流，if/else,循环，switch啥的，在机器码中是如何实现的

### 3.6.1 conditional codes

### 3.6.2 acessing the conditional codes

### 3.6.3 Jump instructions

### 3.6.4 jump instructions encodings

### 3.6.5 Implementing conditional branches with conditional control

### 3.6.6 Implementing conditional branches with conditional moves

### 3.6.7 loops

### 3.6.8 switch statement

## 3.7 Procedures

涉及到各种过程的机器码，是如何运行的，比如堆栈操作、函数之间的控制转移、函数之间的数据传递、堆栈数据存储、寄存器数据存储、递归操作等等。

### 3.7.1 The run-time stack

运行时的堆栈操作

### 3.7.2 control transfer

控制转移(代码从一个function转移到另一个function)

### 3.7.3 data transfer

数据转移。比如把数据通过参数的形式，传递给一个function

### 3.7.4 local storage on the stack

保存在stack上的数据存储

### 3.7.5 local storage in register

寄存器上的数据存储

### 3.7.6 recursive procedures

嵌套

## 3.8 Array allocation and access

机器码是是如何实现数组以及数组操作的

### 3.8.1 basic principals

数组基本操作

### 3.8.2 pointer arithmetic

指针运算

### 3.8.3 nested arrays

多维数组

### 3.8.4 fiexed-size arrays

固定长度的数组

### 3.8.5 variable-size arrays

非固定长度的数组

## 3.9 Heterogeneous Data Structures

这个小节介绍C语言中的两个基本数据结构是如何通过机器码实现的：struct/union

### 3.9.1 Structures

### 3.9.2 Unions

### 3.9.3 Data Alignment

## 3.10 Combining Control and Data in Machine-Level programs

主要介绍了机器码如何支持指针、如何避免内存溢出、如何支持变长的堆栈，等等。

### 3.10.1 Understanding Pointers

理解指针的实现原理

### 3.10.2 life in the real world: using the GDB debugger

引入GDB，通过GDB调试程序、分析程序对应的机器码

### 3.10.3 out-of-bound memory reference and buffer overflow

访问合法范围之外的内存、buffer溢出

### 3.10.4 Thwarting buffer overflow attacks

如何避免、阻挠buffer溢出的攻击

### 3.10.5 supporting varialble-size stack frames

机器码如何支持变长的堆栈

## 3.11 floating-point code

机器码如何支持浮点数

### 3.11.1 floating-point movement and conversion operations

机器码如何支持浮点数的move操作和转换操作

### 3.11.2 floating-point code in procedures

### 3.11.3 floating-point arithmetic operations

机器码如何支持浮点数的加减乘除这些运算

### 3.11.4 defining and using floating-point constants

机器码如何定义浮点数常量

### 3.11.5 Using bitwise operations in floating-point code

机器码如何支持浮点数的位运算

### 3.11.6 floating-point comparasion operations

机器码如何支持浮点数的比较

### 3.11.7 observations about floating-point code

## 3.12 summary

这章花费了大量的篇幅，告诉你机器码的运行原理。这章汇总了编程的各种元素，比如各种控制运算、数组、结构体、浮点数等，告诉我们这些元素在机器码层面是如何实现的。

具体到实操层面，可以综合采用gcc/gdb/objdump等工具，将C程序源码转化为汇编程序，即机器码。也就是说，只要花时间，是能够搞清楚汇编码运行原理的。但是话说回来，我们只要看一下汇编码形式的程序就知道了，直靠自己的肉眼是无法理解的，所以，要依靠本书的指导，才能搞明白汇编码的实现原理。

不要觉得本章的机器码枯燥，机器码可是连接了底层硬件(CPU)和高层的高级程序语言(C语言)之间的桥梁。后续很多章的内容，会大量用到机器码。比如：

1.第五章的程序优化，就通过机器码，说明如何优化应用程序；

2.第六章的内存，会通过机器码，说明程序中的数据、地址，如何传递给内存；

3.

# Chap4 Processor Architecture

这章介绍CPU的整体架构

## 4.1 The Y86-64 Instruction set architecture

介绍了Y86-64处理器指令集的整体架构，注意，这里的Y86指令集，不是我们熟悉的X86指令集

### 4.1.1 programmer-visible state

Y86指令集提供给程序员可见的状态，其实就是提供给程序员可以访问的register

### 4.1.2 Y86-64 Instructions

Y86指令集提供的各个指令

### 4.1.3 instruction encoding

指令集编码，就是把各个具体的指令集编码成一些数字，比如把addq编码为60

### 4.1.4 Y86-64 Exception

Y86指令集如何处理异常

### 4.1.5 Y86-64 Programs

在Y86指令集如何把程序源码转化为机器码

这里有和X86指令集对应的机器码的对比

### 4.1.6 Some Y86-64 Instruction details

一些Y86指令集的细节

## 4.2 Logical design and the hardware control language HCL

这个小节涉及到了硬件层面的电路，就是逻辑电路那些，与门非门啥的。

### 4.2.1 logical gates

逻辑门，就是基于半导体的与门、或门、非门。

逻辑门是数字电路的基础

### 4.2.2 Combinational circuits and HCL Boolean Expressions

将各个逻辑门组装起来，通过HCL(硬件控制语言)的形式，组装成电路

### 4.2.3 word level combinational circuits and HCL Integer Expressions

以word为单位的组合电路，所谓的word，是由字节bit组成的，表示int/指令码/address啥的，

### 4.2.4 Set membership

### 4.2.5 Memory and clocking

引入了存储和时钟

## 4.3 Sequential Y86-64 Implementations

顺序执行的相关内容

### 4.3.1 Oganizing Processing into stages

把处理过程分为各个步骤

### 4.3.2 SEQ Hardware Structure

支持顺序处理的硬件架构

### 4.3.3 SEQ timing

### 4.3.4 SEQ stage implementations

各个顺序执行步骤的实现逻辑

## 4.4 General principles of pipelines

这里引入了流水线的概念

### 4.4.1 computational pipelines

### 4.4.2 A detailed look at pipelines operations

### 4.4.3 limitations of pipelines

### 4.4.4 pipelining a system with feedback

## 4.5 pipelined Y86-64 implementations

这个小节介绍了在Y86-64 架构下，流水线是怎么实现的

### 4.5.1 SEQ+ : Rearranging the computation stages

### 4.5.2 inserting pipeline registers

### 4.5.3 Rearranging and relabeling signals

### 4.5.4 next PC prediction

### 4.5.5 pipeline hazards

### 4.5.6 Exception handling

### 4.5.7 PIPE stage implementations

### 4.5.8 pipeline control logic

### 4.5.9 performance analysis

### 4.5.10 unfinished business

## 4.6 summary

这章介绍了CPU处理器指令集架构是怎么样的，内部实现原理是啥样的。这章承接了第三章的内容，回答了第三章中机器码是怎么来的：其实就是CPU处理的指令集提供的。

这章的难度曲线陡然上升，因为离我们的应用程序越来越远了。并且本章应该是本书唯一没法实践的内容了。像是逻辑门(logic gate)，流水线，等等，涉及数字电路。虽然是数字的，但因为和硬件强相关，所以基本上没法实践。当然，如果硬要实践，可以参考我们之前买的关于电路实践的书，试一下。

但是本章内容还是要了解一下的，一方面，这是实现汇编码原理的基础，另一方面，在中美贸易战的背景下，半导体的制裁是核心，为了突破美国的半导体封锁，了解一些硬件的基础知识，还是非常必要的。

# Chap5 Optimizing Program Performance

程序性能优化

这个章通过各种优化方案，优化一个demo程序：

vec.c

## 5.1 capabilities and limitations of optimizing compilers

优化编译器的局限性

这里举了几个C程序的例子，说明了编译器在某些场景下的局限性

## 5.2 Expressing program performance

引入了一个评估程序效率的标准： cycles per elements

因为本章的目标是优化程序，那么如何评估程序的优化效果呢？就靠这个CPE参数。

CPE中的cycle意思就是CPU中的计算次数。

这个小节还举了一个程序实例，通过程序优化，降低了程序的CPE，实现了程序性能优化的目标。

## 5.3 Program example

引出demo程序

这里有一个核心方法：

void combine(vec\_ptr v, data\_t \*dest)

这是demo程序的最初版本，后续会通过各种方式优化这个方法

## 5.4 Elimimating loop inefficiencies

优化循环

通过减少循环提升性能，优化今后的版本为：

void combine2(vec\_ptr v, data\_t \*dest)

## 5.5 Reducing procedure calls

通过减少函数调用，提升性能。意思就是尽量不要在循环中频繁调用函数。

void combine3(vec\_ptr v, data\_t \*dest)

## 5.6 Elimimating Unneeded memory reference

减少无效的内存引用

意思就是不要把频繁的指针操作放在循环中

优化后的版本为：

void combine4(vec\_ptr v, data\_t \*dest)

5.6还通过汇编代码来说明指针操作对于内存的引用方式。

## 5.7 Understanding modern processors

从CPU角度进行优化

到目前为止，之前的优化都是平台无关的。从这里开始，需要研究在特定CPU平台下的优化方案。这个小节介绍一些现代的CPU处理器

5.7.1 overall operation

5.7.2 functional unit performance

5.7.3 an abstract model of processor operation

## 5.8 Loop unrolling

所谓的loop unrolling(展开)，其实就是拆分一个for循环，拆分成两个for循环。具体原理没怎么看清楚，后续我们结合for循环的汇编形式，进行理解。

优化后的程序版本如下：

void combine5(vec\_ptr v, data\_t \*dest)

这个版本中，把版本4的一个for循环拆成了两个

## 5.9 Enhancing Parallelism

增强并行处理能力

### 5.9.1 multiple accumulators

在for循环中，原来是每次累加1，现在每次累加2。

然后在for处理中，每次循环处理两个元素。

这个方案的本质是，一个累加器改为两个累加器，提升并行处理能力

参考：

void combine6(vec\_ptr v, data\_t \*dest)

### 5.9.2 Reassociation transformation

在combine6()的基础上，for循环还是每次累加2。

但是在for循环处理中，把两个处理元素合并为一个。

void combine7(vec\_ptr v, data\_t \*dest)

## 5.10 Summary of Results for optimizing combining code

总结一下5.4-5.8各个版本升级之后，性能有多少提升

从combine() -> combine6()

## 5.11 Some limiting factors

一些限制因素

### 5.11.1 register spilling

### 5.11.2 branch prediction and misprediction penalties

## 5.12 understanding memory performance

理解内存的性能表现

### 5.12.1 load performance

load performance，意思就是从内存加载数据，比如遍历一个struct list，并从中读取数据。

### 5.12.2 store performance

所谓的store performance，意思就是把数据保存到内存的数据结构，比如遍历一个struct list，给list中每个元素赋值。

## 5.13 Life in the real world : performance improvement techniques

在真实项目开发过程中，提升性能的一些手段，包括：

1.在设计阶段就考虑性能问题；

2.一些基础的编程原则；

3.减少非必要的内存引用

4.

## 5.14 Identifying and Eliminating performance bottlenecks

这个小节 的意思是，通过gprof命令，分析可执行文件，并且提出性能优化意见

### 5.14.1 program profiling

执行gprof命令的步骤如下：

gcc -Og -pg hello.c -o hello

gprof hello.exe

### 5.14.2 Using a profiler to guide optimizationg

## 5.15 Summary

这个章的主题是程序性能优化。放在这里，是因为前面已经有了机器语言、汇编语言的铺垫。很多优化场景可以从机器码的层面进行说明。否则，如果是了解某个编程语言，比如C语言/Java语言，从这些编程语言、从应用层角度来说明的话，可能比较困难。

我们看到这些优化方案还是非常底层的，并且是针对C语言这种底层语言。对于我们日常工作看上去没啥帮助。在web后台开发中，这个层面的优化对于整体性能的提升可以说是微乎其微。但是如果我们有远大的理想，要涉及redis/nginx/db这类高性能、基础软件的开发，就非掌握这类技巧不可。

# Chap6 The Memory Hierarchy

内存

## 6.1 Storage technologies

存储技术

### 6.1.1 Random access memory

以前我们把内存分为ROM和RAM，其中ROM是只读内存，应该是之前我们卡带游戏中，把游戏焊死在固定的卡带中的那种。现在不怎么提ROM了，都是RAM，随机访问内存。

其中RAM又分为SRAM、DRAM

我们现在给电脑配的内存，一般就是DRAM，SRAM一般作为CPU的cache，就是所谓的CPU L1 cache/L2 cache/L3 cache这些。

现在的DDR，是一种内存技术： Dual Data Rate，就是所谓的双通道内存

这里通过一条机器码，来说明如何从内存load数据(到CPU register进行计算)、如何把(register中的)数据store到内存。这里就通过机器码，把程序数据和内存存储、加载的过程联系起来了。

### 6.1.2 dis storage

磁盘存储。

磁盘存储的要点：

1.磁盘的形状：圆形；

2.磁盘的存储空间；

3.磁盘的操作：寻址、旋转延迟、转换时间

4.磁盘逻辑块：block，对磁盘硬件的逻辑抽象

5.各种IO设备，通过IO bus进行连接，磁盘也是IO设备的一种

5.访问磁盘

磁盘可不仅仅是简单的存储设备，在DB这个基础软件中有着举足轻重的影响。

### 6.1.3 solid state disks

SSD

### 6.1.4 storage technology trends

存储技术的趋势

## 6.2 Locality

所谓的locality，就是局部效应，意思就是和缓存差不多。

主要分为temporal locality /spatial locality

temporal locality意思就是最近访问过的数据，在短期内再次被访问的概率比较大，这样程序效率较高，CPU可以把这些频繁访问的数据放在L1/L2这些高速内存缓存中；

spatial locality意思就是相邻位置的数据，后续被访问到的概率比较大。

基于这个locality原理，我们可以有针对性地优化程序性能。比如各种缓存啥的。

### 6.2.1 Locality of references to program data

通过一个数组累加的C代码实例，说明了locality对于程序性能的影响。

这里有一个非常重要的概念，叫做spatial locality

意思就是如果访问连续的内存地址，那么效率更高。

这种性能差异，在高性能场景、海量数据场景，有非常重要的作用。

那么，如何量化spatial locality呢？

如果逐个element访问，叫做stride-1

如果每隔N个元素访问，叫做stride-N

后续6.6总结的时候，会用到stride-N的概念，当然是stride-N的数量越小，内存利用效率越高，程序性能却越。

### 6.2.2 Locality of instruction fetches

内存locality对于(CPU)指令获取的影响

### 6.2.3 summary of locality

## 6.3 The memoy hierarchy

所谓的memory hierarchy，图上已经展示得很清晰了：

cpu register

SRAM L1

SRAM L2

SRAM L3

DRAM

local disk

disk

### 6.3.1 caching in the memoy hierarchy

memory hierarchy中的缓存，重要的概念如下：

1.缓存命中；

2.缓存未命中；

3.缓存未命中的几种场景；

4.缓存管理；

### 6.3.2 summary of memoy hierarchy concepts

## 6.4 Cache memories

把内存缓存起来。

典型的架构，就是通过SRAM(L1/L2/L3 cache)，将内存的内容缓存起来，供CPU快速访问。

那么在SRAM中如何组织缓存的形式呢？6.4这个小节就是说这个事情的。

### 6.4.1 generic cache memory organization

内存缓存的组织形式，有一个概念叫set，一个set包含若干line

### 6.4.2 direct-mapped caches

直接映射的缓存，意思就是一个set只包含一条line，这样CPU处理缓存就比较方便。

这里有一段小程序，说明direct-mapped cache冲突失效的场景。

### 6.4.3 set assosiative caches

通过调整set，规避direct-mapped cache冲突失效的场景

### 6.4.4 fully associative caches

### 6.4.5 issue with writes

### 6.4.6 Anatomy of a real cache hierarchy

### 6.4.7 Performance impact of cache parameters

用于标识缓存参数是否合适的几个标志：

缓存失效率

缓存命中率

缓存命中时间

## 6.5 Writing cache-friendly code

如何写出内存友好的代码。

前几个小节我们讨论了内存的Locality原理、内存的整体架构、内存缓存的架构，这个小节再次回到代码，例子还是之前的sumvect，就是累加array中各个元素，如何实现内存效率最高化。这里用到了内存缓存命中的概念，命中率越高，内存使用效率越高。

## 6.6 Puting it together: the impact of caches on program performance

### 6.6.1 the memory mountain

一上来就提出了memory mountain的概念。

什么是memory mountain呢？

就是一个三维的模型，包括三个坐标：

1.吞吐量

所谓的吞吐量，就是MB/s，标志着程序处理数据的效率，也就是程序运行的效率

2.temporal locality

包含了L1 /L2/ L3 的CPU三级缓存，越接近 CPU，吞吐率越高。比如一段程序temporal locality做得非常好，需要的数据都在L1缓存，那么这段程序的效率越高

3.spatial locality

这个就是我们之前中6.2.1中讨论的stride-N概念，N越小，程序效率越高

这三个坐标模型，能够衡量程序使用内存的效率，这里有一段程序，能够检测吞吐量。

### 6.6.2 rearranging loops to increase spacial locality

介绍了如何通过调整循环中的代码，提升

spacial locality，进而提升程序效率。

spacial locality我们之前已经讨论了是什么意思，

### 6.6.3 Exploiting Locality in your programs

审视一下我们自己的程序，是否能够通过调整Locality，提升程序性能。

## 6.7 Summary

我们总结一下这章，这章不是生硬地介绍内存的种类/功能啥的概念，而是重点介绍了和开发密切相关的主题：如何通过调整程序，提升内存使用效率，进而提升程序整体性能。要实现这个目录，前提是要了解内存。

# Chap7 Linking

关于编译、链接啥的

## 7.1 Compiler Drivers

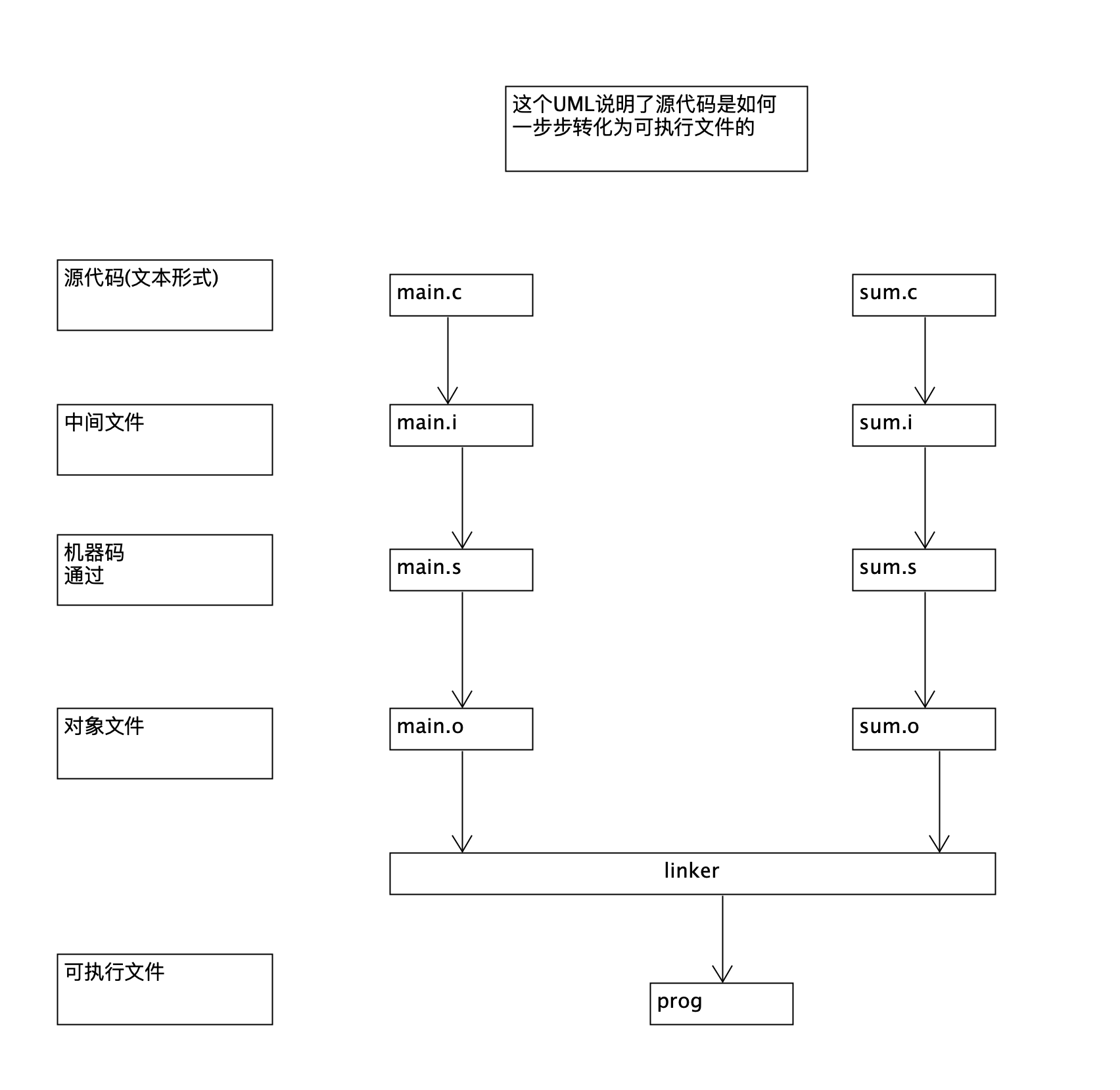
整体介绍了C代码是如何通过编译、链接，转化为可执行代码的。

这里有个代码量非常少的例子，说明了main file/header file是如何通过编译、链接，转化为可执行代码的。

详细步骤如下：

1. 源码转化为中间代码 .c -> .i
2. 中间代码转为机器码 .i -> .s
3. 机器码转化为对象文件 .s -> .o
4. 对象文件链接为可执行文件 .o -> 可执行文件

每个步骤对应gcc的相应工具，整体步骤如下图UML：



## 7.2 Static Linking

静态链接

静态链接的意思就是7.1中说的，把object file转化为可执行文件

静态链接包含两个步骤：

1.符号处理 symbol resolution

所谓的符号，就是代码中的函数名称、各种变量名称啊什么的

2.重定向 relocation

重定向的意思，我猜大概是，编译器生成的object file中，各种代码、数据，都是从地址0开始的(也就是相对地址)，重定向做的事情，就是把这些代码、数据、符号，分配到具体的内存地址。

## 7.3 Object Files

对象文件分为三种：

1.relocatable object file

2.executuable object file

3.shared object file

1.relocatable object file

就是我们在7.1/7.2中介绍的object file，需要linker经过reloacate处理的object file

2.executuable object file

就是那种无需relocate处理，可以直接拷贝到内存中执行的object file

3.shared object file

一种特殊的relocatable object file，可以被动态加载到内存，进行链接。也就是说，可以在加载阶段、运行阶段执行链接步骤。

## 7.4 Relocatable Ojbect Files

可重定向的对象文件

然后重点介绍7.3中所说的Relocatable Ojbect Files。

介绍Relocatable Ojbect Files中包含的各个内容

## 7.5 Symbols and Symbol Tables

符号和符号表

所谓的symbol，包括：

1.global symbol

定义在本模块的全局符号，包括function/ global variables全局变量

2.定义在其它模块，但是被本模块引用的global symbol

比如一个定义在其他模块的function，但是在本模块声明，并且在本模块使用

3.local symbol

## 7.6 Symbol Resolution

符号解析

### 7.6.1 How Linkers resolve duplicated symbol names

介绍链接器处理重复的符号名

### 7.6.2 Linking with static libraries

链接器处理链接静态库，就是那些.o文件

### 7.6.3 How Linkers use static libraries to resolve references

这个小节啥意思，不太懂

## 7.7 Relocation

重定向

在7.6中处理好symbol之后，链接器开始处理重定向

### 7.7.1 Relocation entries

重定向那些function、变量、数据的入口

因为编译器生成object file之后，并不知道代码、数据具体存放在内存哪个位置；也不知道那些引用了其他模块的全局变量、function，具体位置在哪里，所以编译器一旦碰到这些这些未知的引用，就留下一个relocation entries，由linker去完成。

### 7.7.2 Relocating symbol references

重定向那些符号的引用，比如处理指针啥的

这个小节介绍了linker如何实现relocation，这里有一段伪代码，介绍了linker遍历relocation entries，实现relocation功能。

## 7.8 Executable Ojbect Files

可执行的对象文件

经过之前的编译器、Linker处理(relocation、链接)之后，就生成了可执行的对象文件，这个小节介绍了可执行的对象文件( Executable Ojbect Files)，也称为ELF( Executable and linking Files)，具体包含了哪些内容。

在ELF文件中，程序代码(指令)和数据，都有明确的内存地址，方便后续程序的执行。

## 7.9 Loading Executable Ojbect Files

加载可执行的对象文件

以Linux加载可执行文件为例，Linux会为可执行文件开辟一个run-time library image，专门用来执行这个程序。这个run-time library image包含几个区域，包括：

user stack/heap/可读写的segment/只读segment，等等。

## 7.10 Dynamic Linking with shared libraries

用共享库实现动态链接

这里引入了shared libraries的概念，之前我们在7.6.2中讨论了static library，static library需要Linker在编译时期，链接到可执行文件中。而shared libraries可以在运行是加载进来。类似printf()这种需要频繁使用的function，就放在shared libraries中。

在Linux系统中，shared libraries就是我们熟悉的.so文件， 在Window平台下，是.dll文件。

这个小节介绍了如何通过gcc打包shared libraries文件(.so文件)，如何通过gcc命令将.so文件链接到可执行文件中，使得可执行文件能够在运行时调用shared libraries。

## 7.11 Loading and linking shared libraries from applications

从应用加载、连接共享库

在7.10中我们讨论了shared libraries。在7.10中，应用程序要使用shared libraries，还是需要通过gcc预先引用shared libraries。在7.11中更近一步，可以在应用程序运行时，加载shared libraries(.so文件)。

这个小节还有一个C程序代码，演示如何在运行时加载shared libraries。

## 7.12 Positon-Independent Code (PIC)

和位置无关的代码

我们在7.10/7.11讨论了shared libraries，知道应用程序可以在运行时调用shared libraries，那么如果不同应用程序调用同一个shared libraries，应该如何处理呢？这个小节就是解释这个问题的。

首先引入PIC(Positon-Independent Code)，PIC是啥意思呢？之前我们在7.7中讨论过了，Linker要把不同的object file链接在一起，需要有一个relocation的步骤，就是重新调整程序、数据在内存中的位置。那么，shared libraries要在不同应用程序之间共享，当然不能relocation，要不然shared libraries程序位置调来调去，各个应用程序就很难找到。所以这里引入了PIC的概念，就是shared libraries中的程序和数据是不需要relocation，就能被应用程序引用。

解决方案包括两块：

1.PIC 数据，就是shared libraries包含的数据，放在专门的一个data segment，通过GOT技术解决shared libraries数据存放的问题

2.PIC 函数调用，就是shared libraries中包含的function，放在专门的code segment，通过lazy binding / PLT技术解决shared libraries函数调用的问题

## 7.13 Library interpositioning

所谓的interpositioning，意思是介入，介入什么呢？也许换一个单词我们就熟悉了： intercept，就是拦截。这我们在AOP中使用了很多了，非常好用。Java中的AOP，也是拦截，拦截的是java方法。这里的Library interpositioning，拦截的是shared libraries中的方法，能够基于shared libraries，提供一些自定义的逻辑。

这个小节分为三个部分，分别是编译时拦截、链接时拦截和运行时拦截。

针对这几种拦截的场景，有一段实例代码进行说明，这个代码非常清晰，就是拦截了系统自带的内存分配malloc()函数，利用自定义的mymalloc()函数，对原有的malloc()函数进行了封装，封装逻辑非常简单，就是在malloc()分配完内存之后，打印一句话。当然这里重点不是拦截逻辑，而是说明如何实现在编译时拦截、链接时拦截和运行时拦截。

### 7.13.1 compile-time interpositioning

编译时拦截的思路非常清楚，就是通过#define的形式，用自定义的mymalloc()函数覆盖系统自带的内存分配malloc()函数。因为是编译时的拦截，难度最小，通过代码就能实现。

### 7.13.2 link-time interpositioning

链接时拦截就稍微有一些tricky了，需要通过指定symbol reference的方式，在链接时，将malloc()函数指向我们的自定义mymalloc()函数。

### 7.13.1 run-time interpositioning

运行时拦截最为复杂，代码最复杂，操作也比较复杂，需要把自定义的mymalloc()函数编译成.so文件，然后在执行主程序的时候，动态引用.so

## 7.14 Tools for manipulating object files

操作对象文件的工具

这些工具各有特点，能够实现诸如：打印对象文件中的string、对象文件中symbol table 中的symbol、展示对象文件的结构和信息、列出对象文件引用的所有shared libraries。

这些工具能够实现一些反编译、软件破解等高级功能。

## 7.15 Summary

总结一下Linkling(链接)

本章介绍了Linker在程序编译中的作用，重点是将object file转化为可执行文件。涉及的内容包括：静态链接、relocate、符号(symbol)处理、加载可执行文件、动态链接(shared libraries)。

这里一块tricy的地方，就是静态链接和动态链接。所谓的静态链接，就是将静态库.o文件，通过linker，链接到可执行文件中，每个可执行程序保留一份静态库；动态链接库，就是将动态库文件.so，通过linker引用到可执行文件，或者可行性文件在运行时调用，多个可执行文件共用一份动态链接库。

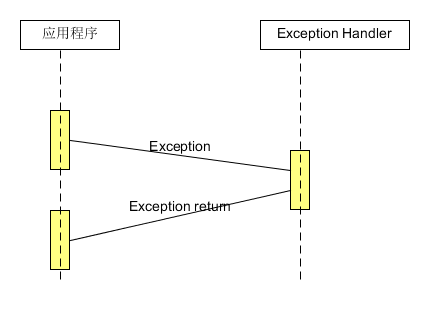
当然，本章着重的内容是object file -> executable file的过程，至于代码文件到object file的过程，可以参考另一本书：《编译原理Compilers》

# Chap8 Exceptional Control Flow

就是异常、信号量、进程这些非常规的系统调用。

## 8.1 Exceptions

所谓的exception，我理解是这样的，我们在前几章也已经了解了，执行一段program，其实就是执行一连串的机器码，执行一些加减乘除啥的计算。但是有一些特殊场景，比如从网络接收到了数据包(socket 相关system call)、程序需要从磁盘读取数据(read() system call)，父进程创建子进程，等等。这些动作可能常规的CPU Processor没法直接处理，需要硬件 + OS参与。比如从程序需要从硬盘读数据，那当然需要disk、OS。怎么实现呢？就是通过exception(异常)，大致情况如下：



### 8.1.1 Exception Handling

异常处理，引入了Exception table的概念

### 8.1.2 classes of Exceptions

介绍了异常的类型，包括：

中断interrupts、traps/system call系统调用、fault异常、Aborts退出

### 8.1.3 Exceptions in Linux /x86-64 systems

Linux /x86-64 系统下的异常，包括：

Linux /x86-64 fault and aborts

Linux /x86-64 systems calls

这里有一段C程序，说明了Linux下如何通过system call实现printf()功能、exit()的功能。

这里有C语言，有机器码，

机器码就解释得非常清楚，机器码提供了不同的system call供程序调用，这些system call是由编码的，比如编号为1就是调用write() system call，编号为60就是调用exit() system call。

后续C语言通过调用system call的实现原理，都可以通过这种方式了解。

## 8.2 Processes

这个小节介绍进程

### 8.2.1 Logical control flow

这个小节为了说明各个进程看起来好像都是独占CPU，其实是分时段使用CPU的，这里有一副图清楚说明了进程分时复用的原理。

### 8.2.2 Concurrent flows

进一步解释了进程的并发，多个进程可以跑在一个CPU上，叫做多任务模式，通过分时复用的方式，某个进程获取到时间片，就能够执行CPU；如果多个进程分别跑在各自不同的CPU上，就叫做parallel flow，也就是真正的并发。

### 8.2.3 Private address space

所谓的Private address space，意思就是各个进程看起来可以拥有自己专属的内存空间

### 8.2.4 User and kernel modes

内存开辟了一段内存区域，只有操作系统可以访问，被成为内核态；普通进程可以访问的内存区域成为用户态。

### 8.2.5 Contex switches

每个进程要运行，需要维护一个context。内核通过context switch机制，实现多任务，就是多个进程看起来好像同时在执行。

## 8.3 System Call error Handling

这个小节的意思是如何处理system call的异常情况，比如典型的创建子进程的system call:

fork()，就有可能失败。

这个小节通过演示Fork()方法，说明了处理system call的方式。

备注：Fork()方法封装了fork()，并处理了异常情况。

这个小节是为后续8.4 Process control做铺垫。

## 8.4 Process control

进程控制

### 8.4.1 Obtaining process IDs

进程号

相关的系统调用方法有：

getpid()

getppid()

### 8.4.2 Creating and terminating processes

创建进程、终结进程

创建进程的方法： fork()

终结进程的方法： exit()

这里有一段C程序，说明如何通过fork()方法创建子进程

fork.c

### 8.4.3 Reaping child processes

获取子进程

通过waitpid()/wait() system call，获取子进程的信息

这里有一段代码，用来说明如何通过waitpid()获取子进程：

waitpid1.c

### 8.4.4 putting processes to sleep

让进程休眠

让进程休眠的system call包括：

sleep()/ pause()

### 8.4.5 Loading and running programs

这个小节介绍了如何在一个进程中，执行一个程序文件，相关的system call为：

execve() // 执行一个程序文件

getenv() // 获取程序执行结果

### 8.4.6 Using fork and execve to Run programs

8.4最后一个小节内容，是介绍如何通过fork()/execve()执行一段程序，经典的场景是unix/linux下的shell执行程序。我们都知道shell的用法：

sh a.out

这个小节有示例程序，从源码层级介绍shell的实现原理，大致意思就是通过fork()创建一个子进程，然后在这个子进程中，通过execve()执行shell程序。

另外，这个小节一开头就介绍了program(程序)和进程(process)的区别，非常精辟，值得细细品味。

## 8.5 Signals

所谓的signal，就是进程或者OS内核发送消息给别的进程，通知别的进程做一些事情。

典型的Linux signal 场景有：

SIGKILL ,就能够kill 进程。(我猜就是kill -9的原理，后续证实一下)。

SIGINT：来自键盘的signal 中断，通知进程：用户通过键盘输入了Ctrl +C 终止进程

### 8.5.1 signal terminology

signal 发送者：内核

signal接收者：进程 进程通过signal handler处理signal

### 8.5.2 Sending signals

包含了如下内容：  
1.process group进程组的概念

2.通过shell脚本：kill 命令发送signal 给进程，kill该进程；

3.通过键盘输入，发送signal 给进程，比如通过键盘输入各种shell命令，ls/grep等，创建各种进程、子进程。

4.通过kill程序，发送signal给进程，我猜这是Linux操作系统提供的kill()方法；

5.通过alerm程序，

本质上，上面的shell kill / kill()/alarm()，都是Linux 提供的Signals

### 8.5.3 Receiving signals

这个小节介绍了进程是如何接收signal的，这里有一段C程序，演示了进程如何接收来自键盘的signal(这个signal 对应SIGINT，也就是在键盘上输入Ctrl +C 终止程序)

### 8.5.4 Blocking and unblocking signals

阻塞、释放signal

意思就是比如signal handler正在处理一个signalA，这时signalB过来，那么signalB可能就需要阻塞(block)了。

### 8.5.5 Writing signal handlers

如何写一个signal handler，处理signal。写一个好用的signal handler有如下困难：

1.signal handler和进程同时执行，和进程共享变量，这里就有个并发的问题；

2.signal接收的方式往往是反直觉的；

3.不同的系统，处理signal的方式也不同，也就是说signal handler是依赖具体系统的

为了说明如何写signal handler，这里有好几段代码进行说明。

### 8.5.6 Synchronizing flows to avoid nasty concurrency bugs

规避并发BUG

这里有两段C代码，一个是有并发问题的；另一个是通过技术手段规避并发问题的。

### 8.5.7 Explicitly waiting for signals

介绍了进程显式地等待接收signal

比如shell启动了一个进程，这个进程作为父进程，会设置SIGINT/SIGCHLD。

## 8.6 Nonlocal Jumps

所谓的nonlocal jumps是一种在function之间可以互相调用的signal

这里有两个C语言程序的例子，来说明nonlocal jumps的用法

## 8.7 Tools for manipulating Processes

介绍了各种操作进程的工具，包括我们熟悉的ps

trace

ps

top

pmap

/proc

## 8.8 Summary

我们总结一下这章，这章的题目是"Exceptional control flow"，但是我们知道，这章的重头戏是进程process，那process和异常处理有什么关系呢？这个逻辑链条是这样的：

异常处理有一种是system call

进程的创建、终结、控制，都需要通过调用system call实现

所以，本章虽然题目是exception，但是实际上是在讲exception最大的一种场景：进程。

# Chap9 Virtual Memory

虚拟内存相关

## 9.1 Physical and Virtual Addressing

介绍了内存的物理地址(PA)和虚拟地址(VA)

## 9.2 Address space

所谓的地址空间，就是一段连续的地址：

{ 0, 1,2 ...}

## 9.3 VM as a Tool for Caching

然后开始引入本章的重点：VM(virtual memory虚拟内存)

所谓的VM，就是保存在磁盘上的一段连续的单元。VM按照一段段的virtual page(vp)进行组织。备注:物理内存也是按照page进行组织的:physical page(pg)。

VM本质上就是物理内存（PM）保存在磁盘上的一段缓存

啥意思呢？就是物理内存有限的情况下，可以部分物理内存的内容，放到磁盘中，放在磁盘中的这些就是VM。VM的引入，主要就是解决物理内存不足的问题。

### 9.3.1 DRAM Cache Organization

DRAM我们之前在内存中已经讨论过了，就是main memory，VM部分内容就缓存在DRAM中。

### 9.3.2 page table

所谓的Page table，就是保存了页的信息：哪些页在DRAM，哪些页在VM

### 9.3.3 Page hits

如果CPU需要的page在DRAM，那就是page hit，否则如果不在DRAM，那就在VM，那就是page fault

### 9.3.4 page fauts

### 9.3.5 Allocating pages

介绍了如何基于VM分配页空间，比如通过malloc()申请一段内存，有可能是在disk分配一段VM空间。

### 9.3.6 Locality to the Rescue again

VM是否会拖慢程序运行效率呢？毕竟VM是在disk中的。

解决方案是locality。设计良好的程序，会通过locality，把要访问的内存限定在某个范围内，保证page hit。如果程序设计得不好，使用的内存超过了physical memery，那就可能会用到VM，导致性能问题。

## 9.4 VM as a Tool for Management

VM有什么好处呢？

1.方便linking

2.方便loading

3.方便共享 进程之间、进程和OS之间的共享

4.方便内存分配

## 9.5 VM as a Tool for Memory Protection

VM用于安全防护方面的，比如禁止用户进程之间互访内存、禁止用户进程访问内核的内存；

## 9.6 Address Translation

所谓的Address Translation，就是把VM中的内容映射到物理内存。

### 9.6.1 Integrating caches and VM

### 9.6.2 Speeding up address Translation with a TLB

引入了TLB(translation lookaside buffer)，加入地址转换

### 9.6.3 multi-level page tables

多级页表

页表的概念我们之前已经了解了：

所谓的Page table，就是保存了页的信息：哪些页在DRAM，哪些页在VM

### 9.6.4 putting it together:end-to-end address translation

把之前的内容整合起来：cache、TLB、多级页表

## 9.7 Case study: The Intel Core i7/Linux Memory System

以intel-i7 CPU 架构为例，说明内存系统的结构，

一上来就介绍了CPU封装的内容，包括：

(1)CPU相关： 四核CPU、寄存器、指令获取、L1/L2/L3高速缓存；

(2)内存相关：内存控制器(和main memory/disk交互)、MMU(内存控制单元)、L1/L2 TLB；

### 9.7.1 Core i7 address translation

这个小节介绍了intel core i7如何实现VM和物理内存的映射

### 9.7.2 Linux virtual memory system

这个小节介绍Linux如何管理VM，包括如何处理page fault

## 9.8 Memory Mapping

内存映射

什么是memory mapping呢？

Linux把VM 区域对应的数据，关联到disk上的一个对象，这个对象可以是Linux文件系统上的一个文件、也可以是Linux 内核直接管理的一个匿名文件。

说白了就是把文件系统的文件和内存映射起来。

### 9.8.1 shared objects revisited

既然在Linux中，VM可以映射到文件系统上的一个对象，那么进程之间需要共享的内容，比如那些只读的运行时代码库(iostd.h中的printf()函数)，就可以通过memory mapping来实现。

### 9.8.2 the fork function revisited

了解了VM之后，我们知道fork()是如何实现的了，其实就是在VM中复制一份当前父进程的数据结构、数据啥的，供新创建的子进程使用。

### 9.8.3 the execve function revisited

然后我们再来回顾一下execve()，这个之前在"chap8 Exception control flow/8.4 process control/8.4.6 Using fork and execve to Run programs"中已经讨论过了，其实就是程序A通过execve()执行程序B。

基于VM的背景，这个小节解释了execve()具体是如何把程序B加载进来的，其实，就是把程序B、程序B引用的库加载到VM。

### 9.8.4 user -level memory mapping with the mmap function

引入了memory mapping相关的API： mmap()/munmap()。我理解是基于VM原理，将某个文件中定义的对象直接映射到VM中。

这么做的目的有两个：

1.极限的文件读取效率；

2.进程间的通信；

关于mmap，我们之前也看到过一些网文，在java中也有应用，参考：

|  |
| --- |
| ./Kirito的技术分享/文件IO/#已阅#重新认识 Java 中的内存映射（mmap）.pdf |
| ./xjjdog/#已阅#OS近距离：mmap给你想要的快！.pdf |

## 9.9 Dynamic Memory Allocation

动态内存分配

VM到这里，终于可以引入动态内存分配了。所谓的动态内存分配，就是在程序运行的时候，申请一段内存。比如我们熟悉的malloc()。这和静态内存申请相对，比如我们在代码里声明一个10个byte长度的数组，就是在程序编译、加载的时候就申请好内存了。

VM的特性，能够用于动态内存分配：进程中有一段VM虚拟内存区域，专门用于动态内存分配，就是我们熟悉的heap。这样就把VM和heap串起来了。

### 9.9.1 The malloc() and free() function

介绍了C语言中和动态内存分配相关的库函数：

malloc() 和free()

介绍 了malloc() 和free()申请内存、释放内存的原理。

### 9.9.2 why Dynamic Memory Allocation

为啥要引入动态内存分配？

这个很直接，就是很多场景下，要申请的内存长度是在程序运行时才确定的。比如一个常见的场景就是创建一个String，String长度基本上都是在程序运行是才能确定的。

### 9.9.3 Allocator requirements and goals

动态内存分配器的各种限制条件，包括：

1.申请一段明确长度的内存；

2.对于动态内存分配请求，要有明确的响应；

3.只通过heap分配动态内存；

4.以block为单位申请内存；

5.对于已经分配的区域，不能修改；

### 9.9.4 fragmentation

碎片化，

就是Heap中的可用内存过于碎片化，导致动态内存分配的申请失败

### 9.9.5 implementation issues

动态内存申请实现中需要考虑的几个现实问题：

比如：

1.空闲的块如何组织、

2.如果在空闲的块中占用内存；

3.空闲块的分割：在空闲的块中占用一段内存之后，剩下的空闲块怎么处理？

4.空闲块的聚合：我们释放一段内存空间之后，释放出来的这段块，如何和原来的空闲块聚合？

这里只是起了一个头，后续几个小节会详细讨论这些实现细节。

### 9.9.6 implicit free lists

管理空闲块的实现方案1： implicit free lists

这个方案的意思是每个空闲块block保存了一段长度的空闲内存，

### 9.9.7 placing allocated issues

当一个alloc请求过来的时候，如何在空闲块中分配一段合适的可用内存，供alloc使用

### 9.9.8 splitting free blocks

在空闲块中找到一段合适的可用内存之后，把这段空闲块分割出来，供alloc使用。

### 9.9.9 getting additional heap memory

如果进程现有的Heap中没有足够的空闲内存，需要申请更多的Heap空间

### 9.9.10 coalescing free blocks

引入coalescing的概念，所谓的coalescing，意思就是聚合。聚合什么呢？我们通过free函数释放一段内存之后，这段空闲的内存需要和heap中的内存聚合起来，解决空闲内存过于分散的问题(内存碎片)。

### 9.9.11 coalescing with boundary tags

这个小节详细解释了coalescing(聚合)的实现方案。

### 9.9.12 putting it together: implementing a simple allocator

综合之前的动态内存分配相关的技术点，提出了一个malloc()的C语言实现。

### 9.9.13 Explicit free list

这里介绍了另一种空闲内存管理方案，我理解是用链表的方式，把空闲块串联起来。

之前介绍的"implicit free lists"方案，感觉是数组形式的block。

### 9.9.14 segregated free list

segregated(隔离的) free list，意思就是把长度类似的Explicit free list，放到一组中。这样后续申请内存的时候，只要找对应长度的list就行了。这个方案的目标是快速找到合适的空闲内存。

## 9.10 Garbage Collection

垃圾回收

### 9.10.1 Garbage Collector basics

### 9.10.2 Mark&Sweep Garbage Collectors

介绍了Mark&Sweep的内存回收方案

这里有一段Mark&Sweep垃圾回收方案的伪代码

### 9.10.3 Conservative Mark&Sweep for C Programs

对Mark&Sweep内存回收方案的改进。

## 9.11 Common Memory-Related Bugs in C Programs

C语言中内存相关的BUG

总共有10种场景。作者对C语言也是相当相当熟悉了。

### 9.11.1 Dereferencing Bad Pointers

引用了错误的指针。

我们知道，VM提供的内存空间，不是所有地方都能引用的。部分VM对应的内存不存在；部分VM对应的内存是只读的。如果我们在程序中引用了错误的指针地址，可能就会报错。

### 9.11.2 Reading Uninitialized memory

读取了一段还未初始化完毕的heap内存

### 9.11.3 Allowing Stack Buffer Overflows

比如通过gets()读取一段超大的字节，导致stack buffer overflow

### 9.11.4 Assuming That pointers and the objects they points to are the same size

把指针的size和指针要指向的对象size混为一谈。

比如我们要申请一段内存以保存n个指针，正常应该是 n \* sizeof(int \*)，却不小心申请了一段内存，大小是n \* sizeof(int)，这两段申请的内存长度是不同的

### 9.11.5 Making off-by-one errors

申请了一段内存，用于保存数组，数组长度是n，那么我们只能访问0- n-1，如果访问位置n，就是所谓的"off-by-one"异常。

### 9.11.6 Referencing a pointer instead of the object it points to

明明应该操作指针指向的元素，却不小心操作了指针。比如：

\*size++ 这是指针前移一格

正确应该是(\*size)++，意思就是size指针指向的元素加一

### 9.11.7 misunderstanding pointer arithmetic

操作指针的方式不对。

比如我们通过移动指针，遍历数据，正常应该是：

int \*p

p++;

我们可能会写成：

int \*p

p += sizeof(int)

这是不对的，指针每次前移4个元素。

### 9.11.8 Referencing nonexistent varialbles

指针指向了一个不存在的变量。

一个常见的例子，就是指针指向了一个function local variable，这个variable在function结束后会被垃圾回收，那么这个指针指向的内容可能就会异常。

### 9.11.9 Referencing data in free heap blocks

引用了heap中某段内存，但是这段内存已经被释放了。

### 9.11.10 introducing memory leaks

内存泄漏，这个是最常见的，比如alloc()一段内存之后，没有free()进行释放，导致内存缓慢泄漏。

## 9.12 Summary

VM是对物理内存的一种抽象。避免了直接使用物理内存的各种问题，提升了内存使用的效率，降低了内存使用的各种风险。所以，在现代操作系统中，我们所说的使用内存，往往是说使用VM。

VM如此重要，篇幅和chap6旗鼓相当。因为我们开发者接触到的所谓的申请内存空间，malloc()就是通过操作VM实现的。

# Chap10 System-Level I/O

系统层面的IO接口，比如linux发布的read()/write()接口

进入最后三章了，这三章统一归为"interaction and communication between programs"，意思就是程序之间交互和通讯。这三章的特点，就是离应用层越来越近了，离底层硬件越来越远了。对于这三个主题，我们在日常工作中接触得更多了。

## 10.1 UNIX I/O

一上来就揭示了UNIX/Linux体系下，文件的本质：一串byte

然后告诉我们，Linux下，网络、磁盘、终端，都会按照文件进行处理

然后大致介绍了针对文件的一些操作的实现逻辑，包括：

打开文件、改变文件的位置、读写文件、关闭文件

## 10.2 Files

介绍了Linux下几种文件类型：

普通文件、目录、socket

尤其是socket，我们之前还以为是网络对象，其实socket本质上也是文件，用于网络通讯。

## 10.3 Opening and Closing Files

涉及文件打开、关闭操作的linux function，包括：open()/close()

## 10.4 Reading and Writing Files

涉及文件读写操作的funciton，包括：read()/write()

## 10.5 Robust Reading and Writing with RIO packages

引入了RIO的概念，RIO意思是robust IO，意思就是可靠的IO读写操作。这是一个基于Linux IO体系的一套文件读写API。

### 10.5.1 RIO unbuffered input and output functions

相关API包括：

rio\_readn()

rio\_wirten()

### 10.5.2 RIO buffered input functions

相关API包括：

rio\_readinitb()

rio\_readlineb()

rio\_readnb()

## 10.6 Reading File Metadata

读取文件的元信息

相关function 包括：

stat()

fstat()

## 10.7 Reading Directory Contents

读取目录内容

相关function 包括：

opendir()

readdir()

closedir()

## 10.8 Sharing Files

文件共享，有三个层级：

descriptor table :保存了pid

file table: 保存了所有打开的文件

v-node:保存了inode信息

## 10.9 I/O Redirection

IO重定向

就是linux下的管道

ls > a.txt 这种

实现原理就是依托dup2() function

这个function的实现原理是，在descriptor table中作一下调整：把目标文件的PID也指向源文件，这样目标文件的输出看起来就和源文件的输出保持一致了(本来就已经是同一个文件了)。这个小节中有一副形象的图。

## 10.10 Standard I/O

C语言提供了一些封装好的IO相关的API：

## 10.11 Putting it together: Whick I/O functions should I use?

介绍了UNIX IO、 RIO、 Standard IO之间的关系

## 10.12 Summary

# Chap11 Network Programming

网络编程

## 11.1 The Client-Server Programming Model

我们熟悉的CS模型

## 11.2 Networks

介绍了网络在整个计算机系统中的位置、网络相关的组件：hub/bridge/route

## 11.3 The Global IP Internet

11.3.1 IP Address

IP 地址在UNIX系统中是如何表示的：32位int

11.3.2 Internet Domain Names

域名

11.3.3 Internet Connections

因特网连接，就是我们熟悉的四要素：源IP、源端口、目标IP、目标端口

## 11.4 The Sockets Interface

这个小节介绍socket在网络通讯中的作用。

比如socket在CS架构下，实现client和server之间的交互过程。这里有一副图，非常形象

后续几个子小节，都是围绕这幅图展开的。

备注：

1.这个我们之前用Java实现过这个socket通讯的过程，C语言虽然没有实践过，但是原理、过程差不多。

2.关于socket的本质，上一章已经介绍了，一个socket连接本质上是UNIX下的一个file，在操作系统中有一个文件描述符(fd)。这个文件主要是用于网络通讯。

### 11.4.1 socket address structure

socket通讯过程中，地址是用什么结构体表示的

### 11.4.2 The socket function

客户端和服务端要实现通讯，都需要创建socket对象，包括client socket/server socket

### 11.4.3 The connect function

就是client 发起connect 请求的function

### 11.4.4 the bind function

服务端发起监听前，先要bind某个IP地址

### 11.4.5 the listen function

服务器发起监听

### 11.4.6 the accept function

服务器接收到来自客户端的socket请求，就调用accept接收这个请求

### 11.4.7 Host and service conversion

host(hostname)和IP地址之间的转换

### 11.4.8 Helper function for th esockets interface

socket通讯的辅助类

### 11.4.9 Example Echo client and server

通过socket通讯实现client和server之间的echo功能，这里有一段代码

## 11.5 Web Servers

### 11.5.1 web basics

介绍了HTTP协议、HTML语言，这些是web server需要实现的协议、标准

### 11.5.2 web content

所谓的web content，就是client/server请求中的请求体。比如一次HTTP请求，服务端返回HTML文件，那么HTML文件内容就放在response content中。

### 11.5.3 http transactions

介绍了HTTP请求的过程，包括http request /http response

### 11.5.4 serving dynamic content

介绍了server端如何(根据client请求带上来的参数)提供动态内容给client

## 11.6 Putting it together : The Tiny Web Server

这个小节整合了前面的内容，包括process/IO//HTTP/等内容，提供了一个C语言的程序实例。这个程序实现了一个简单的web server ，能够提供静态、动态服务。

## 11.7 Summary

这章内容主要是和网络交互相关，介绍了CS模型、IP、socket、HTTP、web server。我们对这块内容已经非常熟悉了，因为工作中经常碰到网络相关的问题、我们也自己实现了Java版本的简易tomcat。在现在这个时代，从事计算机相关的工作，不接触网络，几乎是不可能的。而网络以其清晰、开放的分层架构实现了高效的传输功能，当时发明这个架构的人、通过socket予以实现的人，真的是天才啊。

# Chap12 Concurrent Programming

并发编程

## 12.1 Concurrent Programming with Processes

并发编程的方案一，就是通过fork()/execc()等function 启动多个进程process，处理并发任务。

### 12.1.1 a concurrent server based on processes

这里通过一段C语言程序，说明如何通过fork()启动一个新的进程，处理并发场景。这个场景是socket server 启动新进程，处理来自客户端的socket请求，然后echo back what client says

### 12.1.2 prons and cons of processes

介绍了进程模型的在并发场景下的优点和缺点。

优点是不同的进程有不同的地址空间，进程之间不会串；缺点是进程间的交互比较麻烦，共享信息比较麻烦。

## 12.2 Concurrent Programming with I/O multiplexing

并发编程的方案二，I/O multiplexing。所谓的IO multiplexing，实现的function 是select()，select() function 维护了一组fd，这些fd就是和进程交互的IO文件。一旦有IO事件，select() funciton 就会要求内核把控制权从进程传递给应用。

这里有一段C程序，这段C程序演示了如何通过select()监听来自键盘输入的IO和来自客户端网络的IO，实现的功能是：一边处理来自客户端的echo请求，一边处理来自用户的键盘输入，把键盘输入信息echo back to client。

### 12.2.1 a concurrent sevent-driven erver based on I/O multiplexing

这里有一段C语言程序，基于IO multiplexing 原理，通过select()实现了类似线程池的功能：server维护了一个Pool，这个Poll维护了来自客户端的socket请求对应的fd，一旦有客户端socket请求过来，就把这个socket对一个的fd放到Pool中。最后，批量echo back一段text给这些client。

### 12.2.2 prons and cons of I/O multiplexing

介绍了IO multiplexing方案的优缺点。

## 12.3 Concurrent Programming with Threads

然后就来到了我们在Java中用得最多的并发方案： thread

### 12.3.1 Thread execution model

线程执行模型，上下文不断在不同的线程之间切换

### 12.3.2 posix threads

如何通过Pthread接口创建线程，这里有一段demo程序：

thread.c

### 12.3.3 creating threads

创建线程的接口：

pthread\_create()

### 12.3.4 terminating threads

结束线程的接口：

pthread\_exti()

### 12.3.5 reaping terminated threads

捕获结束的线程：

pthread\_join()

### 12.3.6 detaching threads

线程分离接口

pthread\_detach()

### 12.3.7 initializing threads

初始化线程接口

pthread\_once()

### 12.3.8 a concurrent server based on threads

掌握了以上线程创建的各种API，这个小节引入了一个demo程序：echoserver.c

通过创建线程，开发一个支持并发的程序，实现echo server 的功能。

功能实现很直接：socket server接收到来自客户端的socket请求之后，就启动一个thread处理这个socket请求。

### 12.3.9 thread总结

12.3总结了thread各种API的用法，最后引入了一个实例。

## 12.4 Shared variables with Threaded Programs

介绍如何在线程之间共享变量

这里有一段示例程序，sharing.c，来说明如何共享变量

这段程序的原理是，定义一个全局的指针，然后各个子线程通过这个全局的指针，共享变量。

整个程序思路非常清晰。

### 12.4.1 Threads memory model

介绍 了线程的内存模型

### 12.4.2 mapping variables to memory

线程中不同类型的变量，是如何映射到VM中的

比如全局变量、本地变量(定义在function中的变量)、本地static变量

### 12.4.3 shared variables

被不同线程共享的变量，就成为共享变量

## 12.5 Synchronizing Threads with Semaphores

### 12.5.1 progress graphs

### 12.5.2 Semaphores

### 12.5.3 Using Semaphores for mutual exclusion

### 12.5.4 Using Semaphores to schedule shared resources

### 12.5.5 putting it together: a concurrent server based on prethreading

基于线程的支持并发的服务器

## 12.6 Using Threads for Parallelism

这个小节引入了并行的概念：不同线程跑在不同的CPU上，就是并行

## 12.7 Others Concurrency Issues

### 12.7.1 Thread safety

### 12.7.2 reentrancy

### 12.7.3 Using existing library functions in threaded programs

### 12.7.4 Races

### 12.7.5 Deadlocks

死锁相关

## 12.8 Summary

这章介绍线程。正如《think in java》中，最有意思的一章，莫过于concurrency。thread放在这最后一章，也是代表了计算机应用层，thread是非常重要、非常有用、非常有意思的一块内容。C语言里的线程，也有同步、有死锁、也有著名的生产者、消费这模型。感觉在某种程度上，Java就好像是对C语言做了一层封装，是一层套壳，对开发者更友好的一层套壳。

C语言中的并发，不仅有线程，还有进程、IO多路复用(著名的select /poll/ epoll 估计和这个相关)；C语言中的线程，能够映射到内存VM，能够映射到不同的CPU ，更加贴近底层硬件。