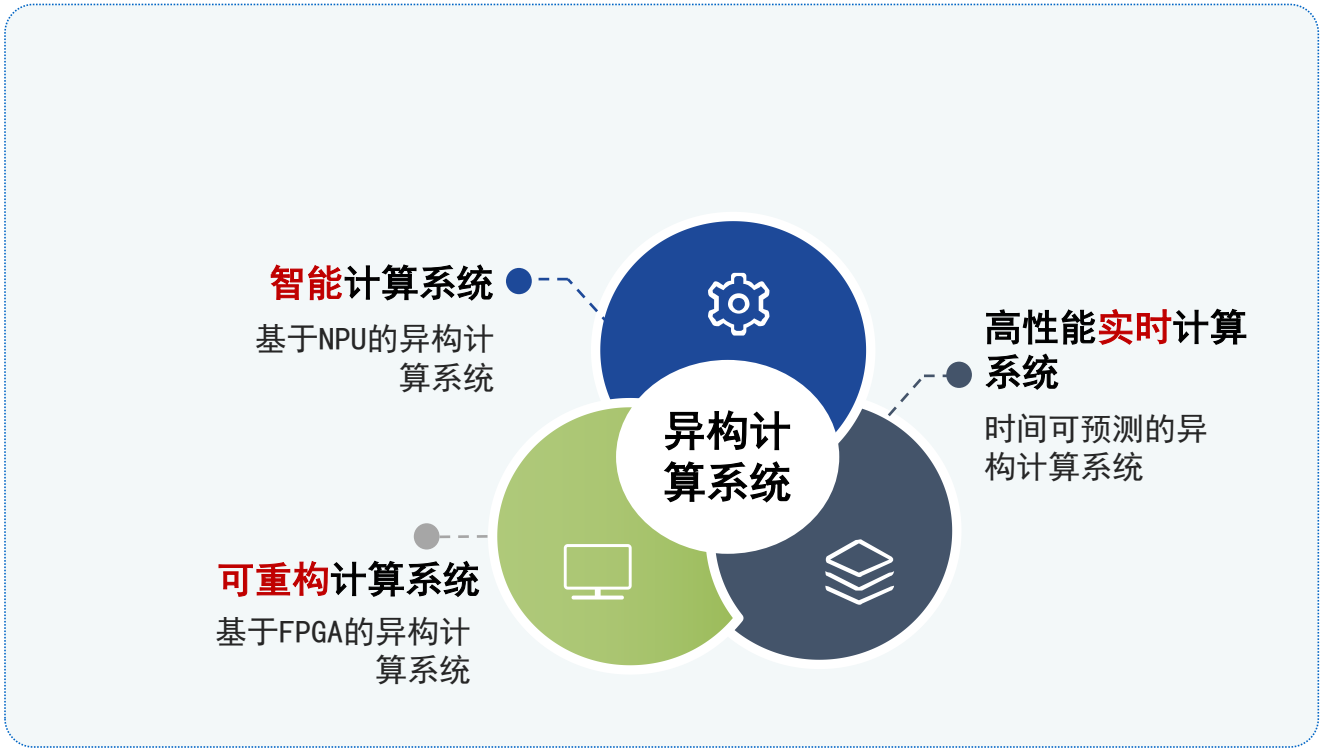



Computer Systems: A Programmer's Perspective
计算机系统

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

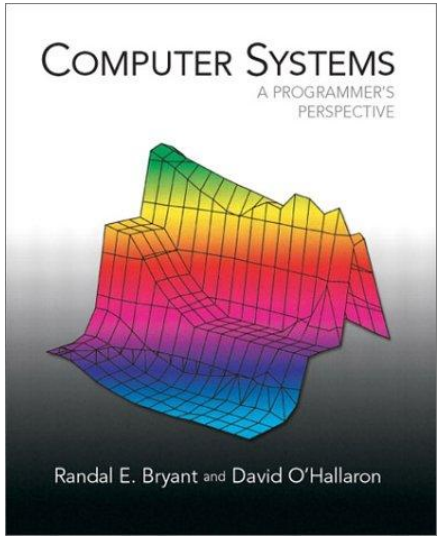
Welcome to ……

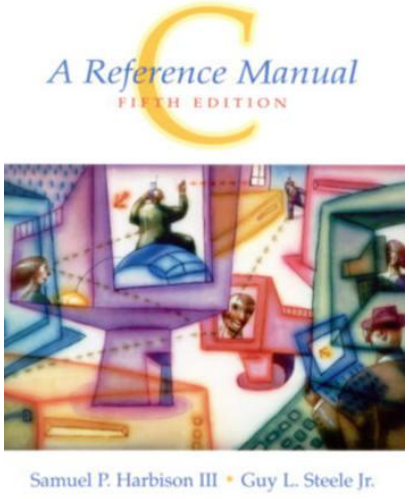
- **主讲：周学海(xhzhou@ustc.edu.cn)**
- **办公地点: 高新区1号学科楼A430A（周五）**



关于这门课……

- **上课地点：G3-113**
- **上课时间：2~19周 5(3,4,5)**
- **参考书：**
  - Randal E. Bryant and David R. O'Hallaron, "Computer Systems: A Programmer's Perspective", 2015第3版 (http://csapp.cs.cmu.edu/
  - 中译本，深入理解计算机系统，龚亦利等译，机械工业出版社(第3版,2016)
  - Samuel P. Harbison III and Guy L. Steele Jr., "C A Reference Manual 5th Edition",Prentice Hall, 2002





Welcome to ……

- **助教：**

姓名	电子邮件
郑振东	zzd411@mail.ustc.edu.cn
唐承	tangcheng@mail.ustc.edu.cn
张学辰	zhangxuechen@mail.ustc.edu.cn

- **课程主页:**
  - bb.ustc.edu.cn
- **QQ群号：703874881**

Grading

- **实验： 50%**
- **平时作业: 10%**
- **期末考试: 40%**
- **……**

第1章 Introduction

- **计算机系统?**
  - **系统、模型、物理世界**
  - **模型vs.现实，抽象vs.具体**
  - **深入理解计算机系统的重要性:**
    - **举例：5方面具体问题**
- **本课程主要内容**
  - Builder-Centric vs. Programming-Centric

Part1：计算机系统?

1.1 系统的定义

1.2 计算机系统

1.3 从程序员的视角：计算机系统

1.4 从程序员的视角：面向计算机系统的优化

System

- **系统（System）：指由若干相互关联、相互作用的元素结合而成，具有特定功能的有机整体。（多元性、相关性、整体性）**
- **系统分类：**

类型	特点	示例
自然系统	按自然法则演变，无人工干预	生态系统、天体系统
人工系统	人为设计规则，实现特定功能	计算机系统、交通系统
复合系统	自然与人工系统的结合	导航系统、人机系统
生理系统	生物体内多个器官协同完成生理机能	消化系统、神经系统

- **（人工）系统**
  - **ISO/IEC/IEEE15288：**为了达到一个或多个所描述的目标由相互作用的要素构成的集合。是**人为创造的**、用于在所定义的环境中提供产品或服务，以造福于用户和其他利益相关者
  - **The International Council on Systems Engineering (INCOSE)：**完成所定义目标的要素、子系统或组件的集合，这些要素包括产品、过程、人员、信息、技术、设施、服务和其他支持要素
- **In Summary: 系统并不仅仅是一些元素的简单集合，是一个由一组相互关联的元素构成的、能够实现某个（些）目标的整体。**

Which of the following would be considered a system ?

- ☐ A box full of headphones
- ☐ The houses in a neighborhood
- ☐ The list of text messages exchanged between two people
- ☐ A jet airplane with components to assist and control the plane

Definition of System

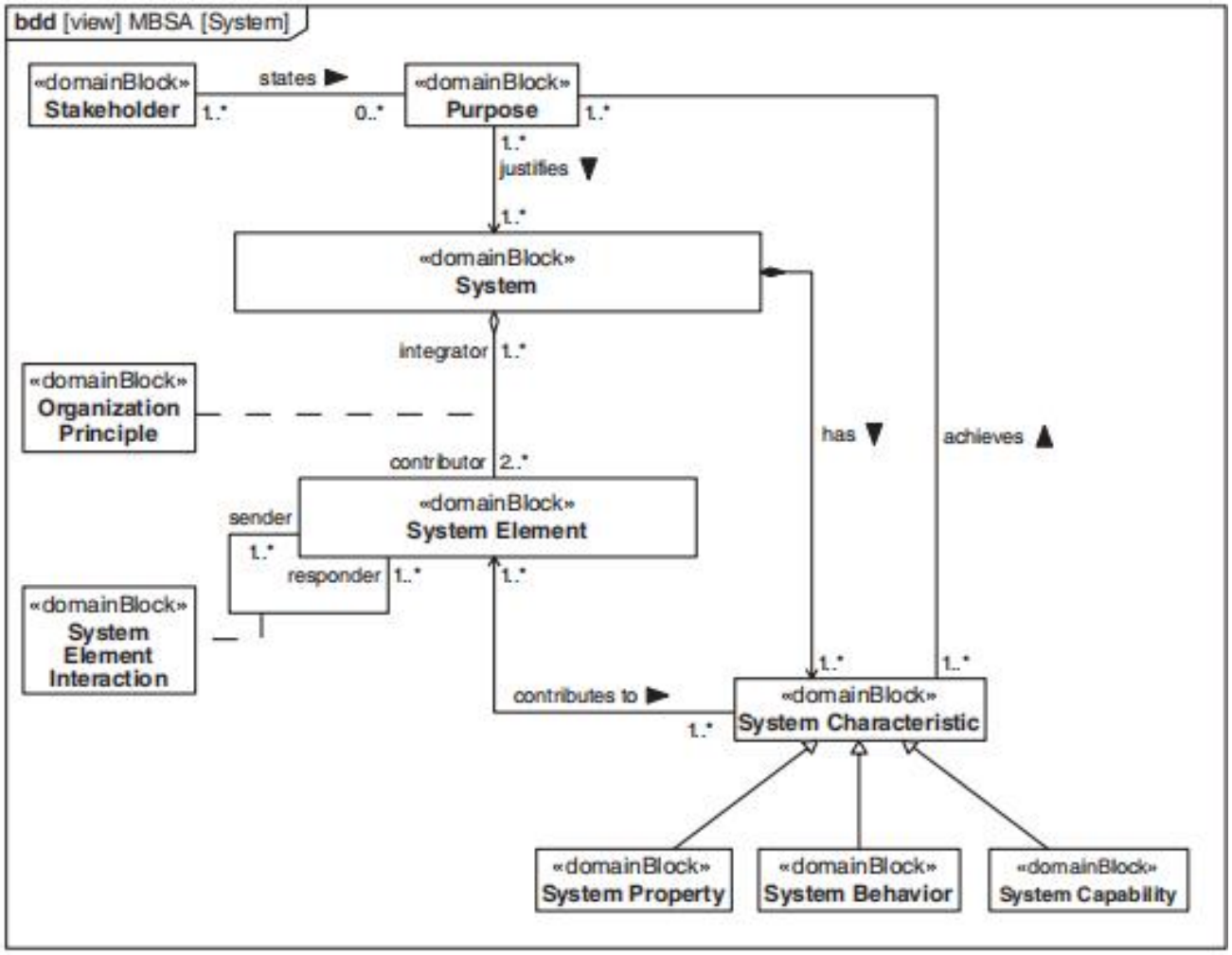


Figure 4.1. Definition of "System".

系统的核心特性?

- **多元性:**
  - 系统由至少两个可区分的要素组成，单一或完全相同的元素无法构成系统
- **相关性:**
  - 元素间通过特定规则相互作用，形成结构化网络
- **层次性:**
  - 元素本身可以由**更简单的功能实体或部件**组成(层次结构)
- **整体性:**
  - 系统是一个整体。除了构成系统的单一元素的性质，还有元素间相互叠加作用的结果，称为emergency
- **目的性:**
  - 系统为实现特定目标而设计

- **动态性：**
  - 系统通过物质/能量/信息的动态交换实现功能，且自身经历从产生到消亡的生命周期演化。
- **环境适应性：**系统通常在特定的环境/场景（**上下文**）中运行
  - operating environment or context（系统的外部视图）：包含不属于系统但确实与系统交互的元素。称为”操作环境或上下文“（包括系统的使用者）
  - system boundary (系统边界）：系统内部和外部视图产生了系统边界的概念。

- **与系统使用相关的质量属性：**
  - 例如可维护、可靠、互操作性等
- **系统结构（System Architecture）：**
  - 系统在其环境中的基本概念或性质。体现在其构成的元素、元素间关系以及其设计和进化的原则中

如何理解系统？ Model

抽象是我们理解复杂系统的重要手段

- **人所认识的系统：物理世界的系统的“心里表征”**
  - 我们所认识的现实世界实体都只是模型
  - 模型通常是与现实世界的实体高度一致的
  - 模型仍远远达不到能完整地描绘实体的程度

- **Model：现实世界系统的一种抽象**
  - 模型是在物理世界中可以实现的**实体的表示**
  - 模型是为某一目的而开发的感兴趣系统的**抽象**
  - 一个模型应该解决**特定的利益相关者**的需求，并对系统的工程具有明确的有用性
- **Model的特性：**
  - Mapping：是实体的一种**映射**。
  - Reduction：反映了实体的**部分**特性。
  - Pragmatic：模型（模拟）要能完成某些特定的功能，以反应实体的特性。

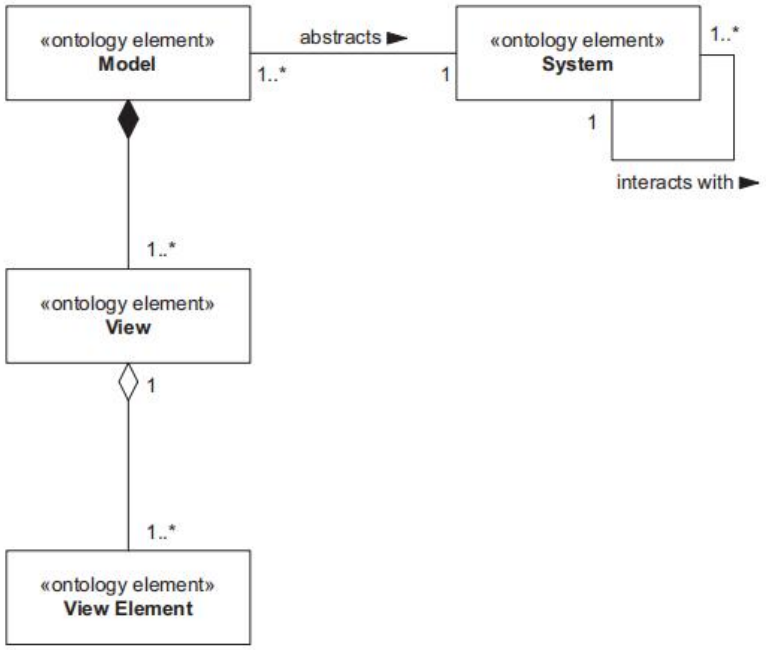


Figure 2.1 The Model concept

计算机系统

- **计算机系统：自动进行信息存储、处理和交换的系统**
  - 计算、存储、信息交换
- **关键元素：**
  - **硬件:** 物理设备（如CPU、内存、存储设备、输入/输出设备等），是系统运行的基础
  - **软件**：程序与数据（如操作系统、应用软件），负责控制硬件并实现具体功能
  - **固件**：嵌入硬件的低级软件（如BIOS），介于硬件与软件之间，提供基础控制逻辑
- **系统特性体现：**
  - **整体性：**各组件协同工作，缺一不可（硬件、固件、软件）
  - **层次性**：从底层硬件到高层应用形成分层结构（如硬件→操作系统→应用软件）
  - **目的性**：最终服务于计算任务（如数据处理、网络通信等）

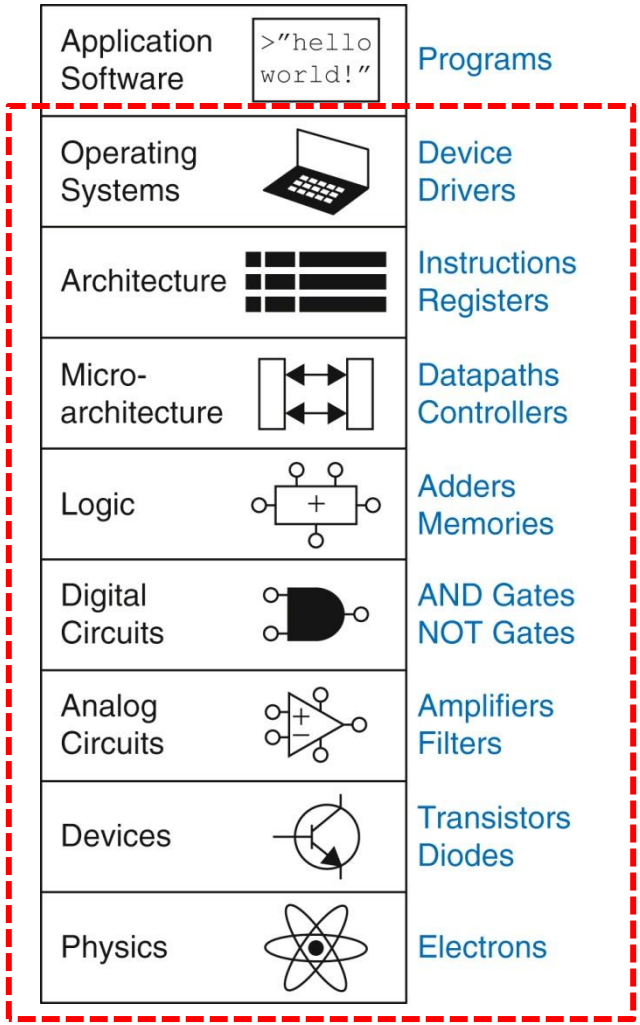
- **计算机系统与其他人工系统（如机械系统、社会组织系统等）的核心区别: 在于其基于信息处理的自动化能力和高度抽象的符号化操作。**
  - 计算机系统的独特性在于其将现实问题转化为可计算的符号模型，并依赖硬件系统实现超高速、低误差的信息处理。

计算机系统的层次结构

- **设计和理解复杂的计算机系统：抽象化、模块化、层次化、分级化**
- **MBSE：Model-based System Engineering**

- **计算机系统的基本抽象（模型）：存储器、解释器、通信链路**

- **计算机系统可以有不同层次（视角）的抽象，**
  - 不同层级的抽象
  - **不同视角的抽象**



现代计算机系统的抽象层次

从程序员的视角：计算机系统

- **抽象是处理复杂系统的重要手段**
  - 我们所认识的计算机系统仍然是现实物理世界的近似表示
  - 现实世界的系统与我们所认识的系统是有差异的

- **大多数计算机课程都强调抽象**
  - Abstract data types
  - Asymptotic analysis
- **抽象是有局限性的**
  - Especially in the presence of bugs
  - Need to understand underlying implementations
  - Sometimes the abstract interfaces don’t provide the level of control or performance you need

计算机系统中计算&存储与现实世界的差异

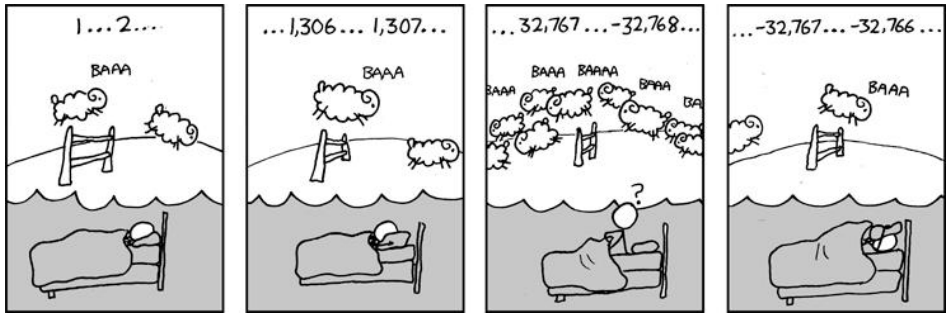
- **计算：计算机系统中的计算是对数学上计算的模拟**
  - 现实世界（数学）
    - 抽象性：计算基于理想化的符合和规则，不考虑物理限制
    - 连续性：数学运算可处理连续值，理论上可支持无限精度
    - 确定性：结果精确
  - 计算机系统
    - 离散性：基于二进制（0/1）和有限位宽（如32位浮点数），存在精度限制（如浮点误差）
    - 物理约束：受硬件限制（如CPU时钟周期、寄存器位数），计算速度和并行能力有限
    - 近似性：浮点运算可能引入舍入误差（如0.1+0.2≠0.3）
- **存储：计算机系统中的存储是在物理约束条件下的存储**
  - 现实世界（理论）
    - 无限容量："存储"是抽象的，可假设无限内存（如无限长的数列）
    - 无损性：数据可完美保存，无丢失或损坏风险
  - 计算机系统
    - 有限容量：受物理介质的限制，需管理存储空间
    - 易失性与可靠性：内存断电丢失数据，磁盘可能因位翻转或物理损害导致数据错误
    - 编码开销：数据需按特定格式编码（ASCII，UTF-8）编码，而占用额外空间

计算机系统在物理世界中实现计算和存储时
需妥协于离散性、有限资源、误差容忍等现实约束

Great Reality #1

#1：Int，Float并不等价于数学中的整数和实数

- **Example1： Is $x^2 \geq 0$?**
  - Float's: Yes!
  - Int's:
    - 40000 * 40000 --> 1600000000
    - 50000 * 50000 --> **-1794967296**



- **Example2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> **0**

计算机系统中的计算是对现实世界计算的模拟≠现实世界

Computer Arithmetic

- **计算机算术运算是离散量间的（有模）运算**
  - 现实世界的算术运算具有重要的数学特性
- **不能假定计算机的算术运算都满足“通常”的数学属性**
  - Due to finiteness of representations
  - Integer operations
    - satisfy “ring” properties：Commutativity, associativity, distributivity
    - **but，two’s comp. arithmetic do not obey ordering properties (保序性)**
  - Floating point operations
    - satisfy “ordering” properties: Monotonicity, values of signs
    - **but, Violates associativity/distributivity**
- **给我们带来的启示:**
  - 需要理解抽象概念适用的场景
  - 这对编译器开发者以及关键应用的程序员尤为重要

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- **类似于在FreeBSD的getpeername实现中的代码**
- **有许多人都在试图找出程序中的漏洞**

从系统的角度看：存储是有界的，是需要管理的

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

```
typedef long unsigned int  size_t
```

```
#include <string.h>
void *memcpy(void *dest, const void
&src, size_t n);
```

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

```
typedef long unsigned int  size_t
```

```
#include <string.h>
void *memcpy(void *dest, const void
&src, size_t n);
```

Part1：计算机系统?

1.1 系统的定义

1.2 计算机系统

1.3 从程序员的视角：计算机系统

1.4 从程序员的视角：面向计算机系统的优化

面向计算机系统的程序优化-软件层面的优化

- **算法与数据结构优化**
  - 时间复杂度优化：选择更优的算法
  - 空间换时间：使用缓存、预计算
  - 避免冗余计算：循环展开、查表
- **并发与并行优化**
  - 多线程优化
    - 减少锁竞争：使用无锁数据结构、线程局部存储
    - 任务并行化：使用线程池
  - 多进程优化：进程间通信（IPC）优化、消息队列（POSIX MQ）
  - 分布式计算：使用MPI、MapReduce等
- **编译器与代码生成优化**
  - 编译器优化选项：
  - 内联函数：减少函数调用开销
  - Profile-Guided Optimization（PGO): 基于运行时数据优化热点代码

面向计算机系统的程序优化-系统层面的优化

- **操作系统交互优化**
  - 系统调用优化
  - 文件系统优化
  - 网络协议优化
- **虚拟化与容器化优化**

面向计算机系统的程序优化-硬件层面的优化

- **利用CPU的特性**
  - SIMD指令集：加速并行计算
  - 分支预测优化：减少if-else分支，使用位运算或查表替代条件判断
- **利用存储层次特性**
  - 数据局部性：优化数组访问顺序
  - 减少缓存失效：避免随机内存访问，使用结构体数组或数组结构体
- **内存管理优化**
  - 减少内存分配/释放：预分配内存池（如对象池、内存池），避免频繁malloc/free
  - 避免内存碎片：使用连续内存
  - 对齐访问
- **存储与I/O优化**
  - 批量读写：减少小文件/小数据I/O操作，使用缓存区
  - 异步I/O： 使用epoll(linux), IOCP(windows) 或 libuv提高吞吐量
  - 压缩与序列化：选择高效格式

程序优化方法论

- **测量**
  使用工具定位瓶颈（如perf、valgrind）
- **分析**
  理解系统行为（如缓存命中率、锁竞争）
- **优化**
  针对性改进（算法替换、并行化）
- **验证**
  确保优化效果及副作用分析

程序员应结合计算机系统知识（硬件、操作系统、编译器）和工程实践（profiling、调试），实现高效、可靠的代码优化

Great Reality #2

#2: 需要理解熟悉汇编语言

- **虽然可能您永远不会用汇编语言编写程序**
  - Compilers are much better & more patient than you are
- **但是熟悉汇编代码是理解机器底层执行模型(machine-level execution model）的关键**
  - Behavior of programs in presence of bugs
    - High-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating/fighting malware
    - x86 assembly is the language of choice!

Assembly Code Example

- **Time Stamp Counter**
  - Special 64-bit register in Intel-compatible machines
  - Incremented every clock cycle
  - Read with rdtsc instruction (Read Time-Stamp Counter)
- **Application**
  - Measure time required by procedure
  - In units of clock cycles

```
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```

Code to Read Counter

- 使用GCC的内联asm编写少量的汇编代码
- 编译器将汇编代码插入生成的机器代码中

```
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits
   of the cycle counter.
*/
void access_counter(unsigned *hi, unsigned *lo)
{
    asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
    : "=r" (*hi), "=r" (*lo)
    :
    : "%edx", "%eax");
}
```

Code to Read Counter

```
/* Record the current value of the cycle counter. */
void start_counter()
{
    access_counter(&cyc_hi, &cyc_lo);
}

/* Number of cycles since the last call to start_counter. */
double get_counter()
{
    unsigned ncyc_hi, ncyc_lo;
    unsigned hi, lo, borrow;
    /* Get cycle counter */
    access_counter(&ncyc_hi, &ncyc_lo);
    /* Do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;  //检查低32位是否借位）
    hi = ncyc_hi - cyc_hi - borrow;
    return (double) hi * (1 << 30) * 4 + lo;
}
```

Measuring Time

- **比看起来要复杂：差异的来源有很多因素造成的**
- **例如：Sum integers from 1 to n**

n	cycle	Cycles/n
100	961	9.61
1000	8,407	8.41
1,000	8,426	8.43
10,000	82,861	8.29
10,000	82,876	8.29
1,000,000	8,419,907	8.42
1,000,000	8,425,181	8.43
1,000,000,000	8,371,2305,591	8.37

Great Reality #3 *Memory Matters*

#3: 计算机中随机读写存储器是一种非物理的抽象概念 (与现实世界有差距)

- **内存不是非受限的（unbounded)**
  - It must be allocated and managed
  - Many applications are memory dominated
- **内存性能不是统一一致的**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
- **尤其内存引用错误非常有害**
  - Effects are distant in both time and space

Memory Referencing Bug Example

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

```
fun(0)   → 3.14
fun(1)   → 3.14
fun(2)   → 3.1399998664856
fun(3)   → 2.0000061035156
fun(4)   → 3.14, then segmentation fault
```

- **其结果与特定的体系结构相关**

Memory Referencing Bug Example

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

```
fun(0)   → 3.14
fun(1)   → 3.14
fun(2)   → 3.1399998664856
fun(3)   → 2.0000000061035156
fun(4)   → 3.14, then segmentation fault
```

Explanation:

Saved State	
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

4

Location accessed by `fun(i)`

Memory Referencing Errors

- **C和c++不提供任何内存保护**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- **内存不当引用会导致bug（nasty bugs）**
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- **我们该如何应对?**
  - Program in Java,Ruby, or ML
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)
  - .....

Great Reality #4:

#4：算法的不同实现，其性能也会有较大差别

- **渐进复杂度分析中的常数因子也很重要**
- **甚至精确的操作数量也不能精确预测性能**
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- **必须了解系统才能更好地优化性能**
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Memory System Performance Example

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

4.3ms

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

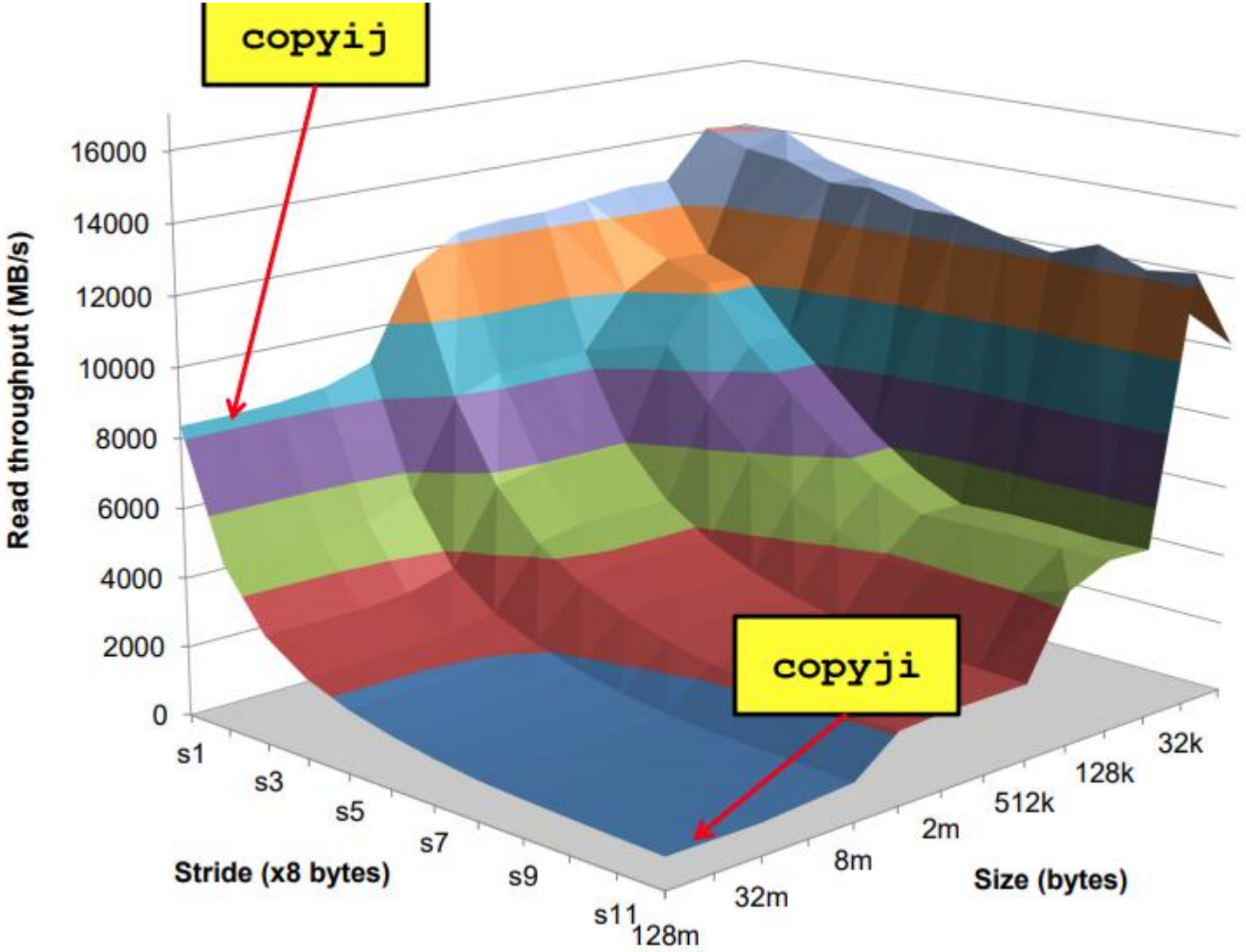
81.8ms

(2.0 GHz Intel Core i7 Haswell)

19+ times slower!

- **存储按照分层结构组织**
- **性能取决于访问模式**
  - Including how step through multi-dimensional array

Why The Performance Differs



Memory Performance Example

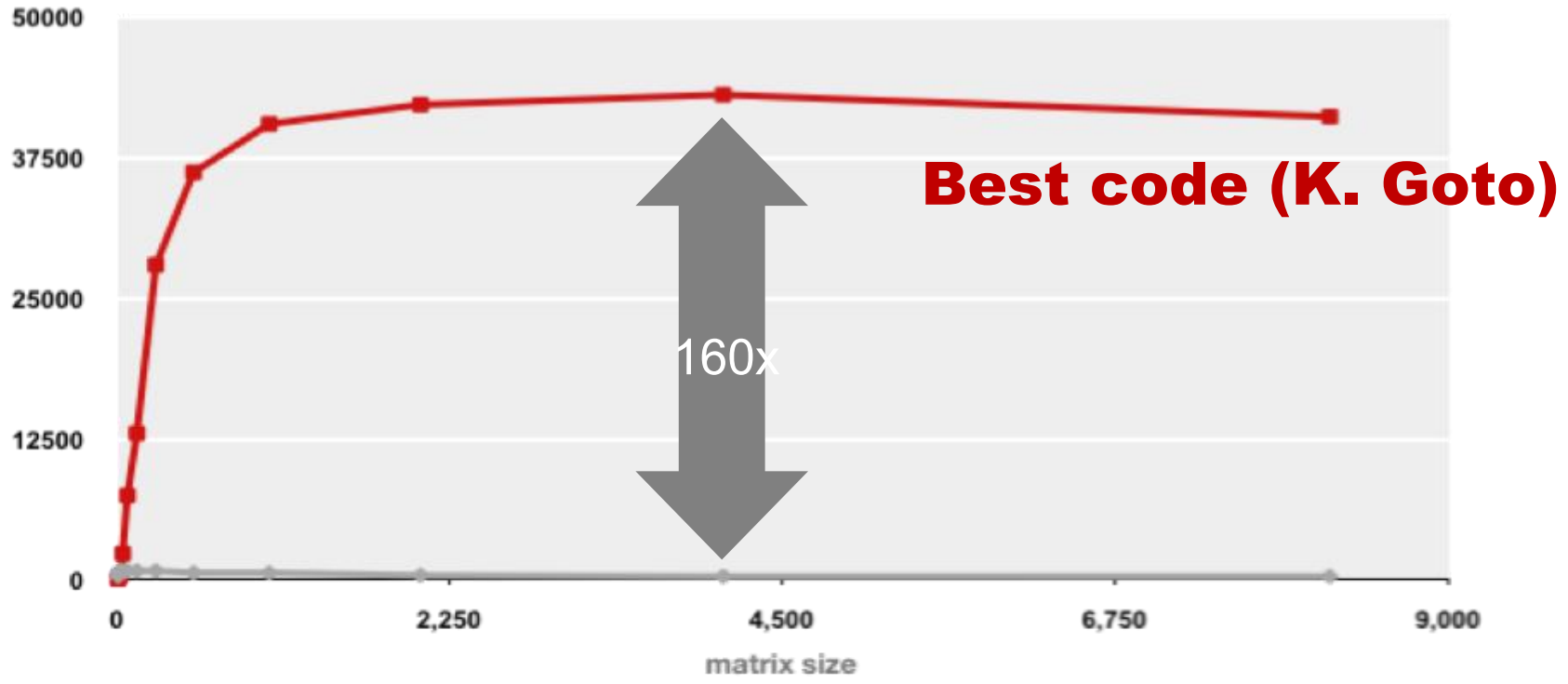
- **矩阵-矩阵乘法的实现**
  - Multiple ways to nest loops

```
/* 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;
  }
}
```

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

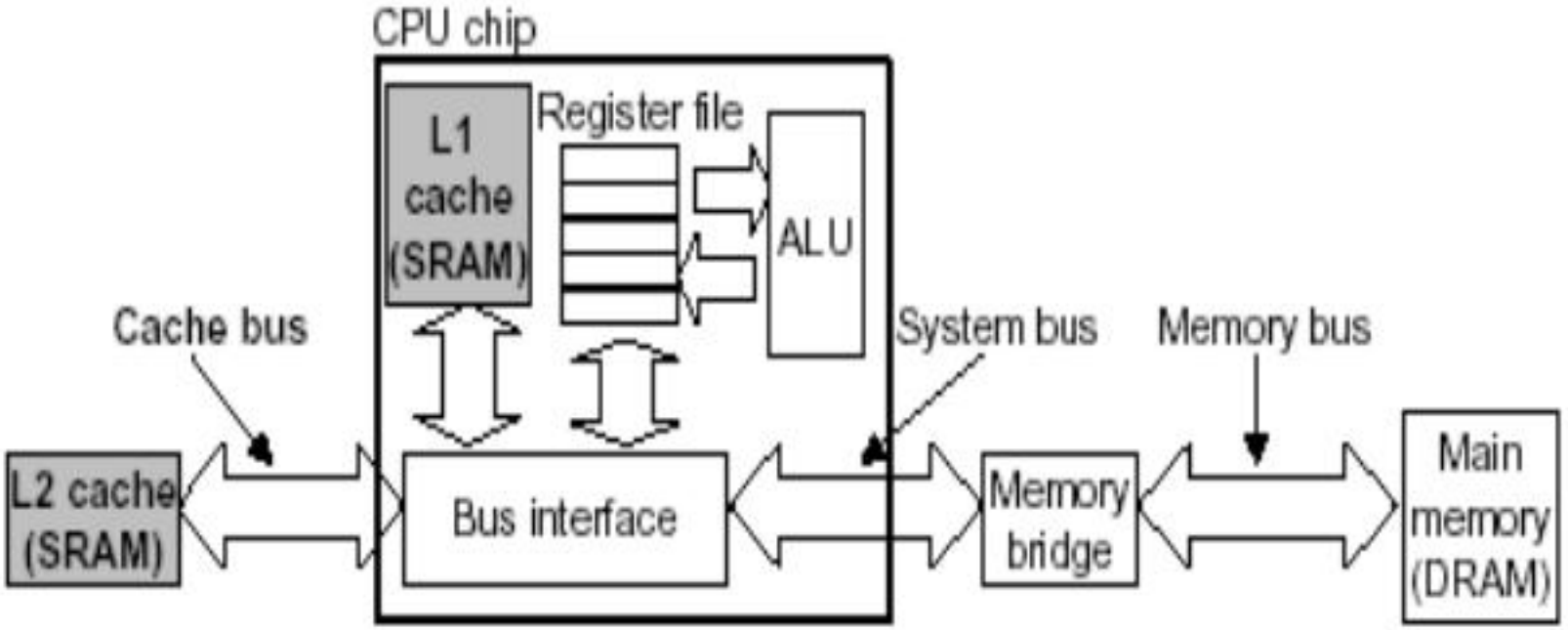
Example Matrix Multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision)



- **Standard desktop computer, vendor compiler, using optimization flags**
- **Both implementations have exactly the same operations count ($2n^3$)**
- **What is going on?**

Memory System

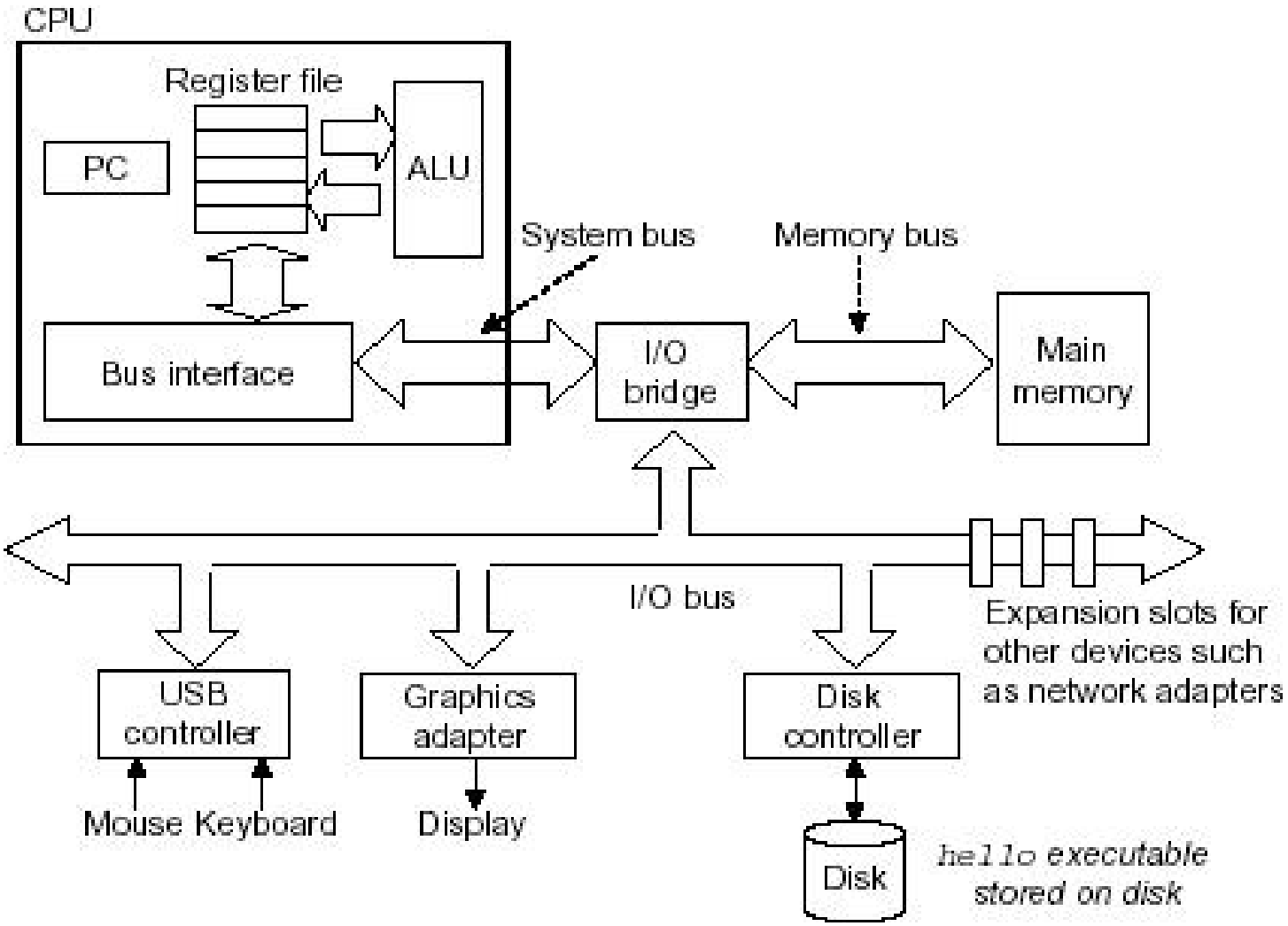


Great Reality #5

#5: 计算机不仅仅是执行计算任务

- **计算机系统需要输入和输出数据**
  - I/O系统对程序的可靠性和性能至关重要
- **计算机系统通过网络互联并相互通信**
  - 许多系统级问题出现在网络环境中
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Hardware Organization (Naïve)



Part2：本课程内容

- **计算机系统?**
  - 系统、模型、物理世界
  - 模型vs.现实，抽象vs.具体
  - 深入理解计算机系统的重要性:
    - 举例：5方面具体问题
- **本课程主要内容**
  - Builder-Centric vs. Programming-Centric

Course Perspective

- **大多数系统课程是Builder-Centric**
- **Computer Architecture**
  - Design pipelined processor in Verilog
- **Operating Systems**
  - Implement large portions of operating system
- **Compilers**
  - Write compiler for simple language
- **Networking**
  - Implement and simulate network protocols

Course Perspective

- **课程定位： 从编程者的角度理解计算机系统**
- **目标：为编写更高效、更可靠的程序提供支撑**
  - 程序设计语言的要素在系统中是如何表示的?
  - 程序在计算机系统上是如何运行的?
  - 如何实现面向计算机系统特性的性能优化?
    - 从计算、存储、信息交互视角
  - 程序并发执行的正确性问题
  - ……

程序员需掌握的计算机系统知识 -编写高性能代码

- **硬件与程序性能优化 （计算）**
  - 处理器体系结构：理解流水线、超标量、分支预测等机制。
- **存储层次与局部性原理 （访存）**
  - 存储层次 （金字塔结构）
  - 缓存优化实践：利用局部性，优化数据布局，避免随机访问模式，使用连续内存分配，利用编译器指令预取数据
- **并发与并行编程**
  - 多核并行：通过线程/进程模型实现任务并行，结合同步机制避免竞态条件。
  - 异步与向量化：使用SIMD指令加速数据并行计算，或通过异步I/O重叠计算与数据传输

编写安全代码

- **内存安全与漏洞防护**
  - 缓冲区溢出防护：边界检查函数，栈随机化、金丝雀值（Stack Canaries）
  - 输入验证：对所有外部输入进行白名单过滤，防止SQL注入、XSS等攻击
- **安全编程原则与机制**
  - 最小权限原则：限制进程权限，避免root权限运行服务
  - 加密与敏感数据保护
  - 安全错误处理：返回通用错误信息，避免泄露系统细节
- **代码安全性与可维护性**
  - 静态分析与测试
  - 防御性编程：断言检查、代码审查及遵循安全编码标准

系统思维在编程中的综合应用

- **程序生命周期的理解**
  - 编译链接过程：掌握从源码到可执行文件的步骤（预处理→编译→汇编→链接），理解符号解析、重定位对库依赖的影响（如静态链接 vs 动态链接）
  - 加载与运行：了解ELF文件格式、虚拟内存布局（如Linux的0x08048000地址），以及动态链接库（如dlopen）的运行时加载机制。
- **操作系统与硬件协同**
  - 异常控制流：理解中断、陷阱（如系统调用int 0x80）、故障（如缺页异常）的处理流程，优化信号处理逻辑
  - 虚拟内存管理：通过mmap管理大内存映射，减少物理内存占用；结合页表机制优化TLB命中率
- **性能与安全的权衡**
  - 加密开销
  - 并发安全：无锁数据结构替代锁机制，减少同步开销但需谨慎处理内存序（Memory Ordering)问题

About the Textbook

- **作者:**
  - Randal E. Bryant: CMU, Professor, ACM Fellow & IEEE Fellow
  - O'Hallaron: CMU, Professor
- **教材结构:**
  1. **Introduction**
  2. **Representing and Manipulating Information**
  3. **Machine-Level Representing of Programs**
  4. **Processor Architecture**
  5. **Optimizing Program Performance**
  6. **The Memory Hierarchy**
  7. **Linking**
  8. **Exceptional Control Flow**
  9. **Virtual Memory**
  10. **System-Level I/O**
  11. **Network Programming**
  12. **Concurrent Programming**

程序结构和运行

在系统上运行

程序间的交互和通信

Course Structure

- **Lectures:**
  - Introduction (chap. 1)
  - Information Representing (chap. 2)
  - Machine Language (chap. 3)
  - Processor Architecture (chap. 4)
  - Code Optimization (chap. 5)
  - Memory Hierarchy (chap. 6)
  - Linking (chap.7)
  - Exception Control(chap.8)
  - Virtual Memory(Chap.9)
  - I/O(Chap.10)
  - Network Programming(Chap.11)
  - Concurrent Programming(Chap.12)

Lecture topics

- **程序和数据**
- **主要内容**
  - Bits operations, arithmetic, assembly language programs
  - Representation of C control and data structures
  - **Includes aspects of architecture and compilers**

The Memory Hierarchy

- **存储层次**
- **主要内容**
  - Memory technology, memory hierarchy, caches, disks, locality
  - **Includes aspects of architecture and OS.**
- **实验**
  - L4 (Perflab): Optimize the performance of an application kernel function.
    - Learn how to exploit locality in your programs.

Linking and Exceptional Control Flow

- **链接和异常控制流**
- **主要内容**
  - Object files, static and dynamic linking, libraries, loading
  - Hardware exceptions, processes, process control, Unix signals, nonlocal jumps
  - **Includes aspects of compilers, OS, and architecture**

Virtual memory

- **虚拟存储**
- **主要内容**
  - Virtual memory, address translation, dynamic storage allocation
  - **Includes aspects of architecture and OS**
- **实验**
  - L5: Writing your own malloc package
    - Get a real feel for systems programming

I/O, Networking, and Concurrency

- **系统I/O、网络、并发**
- **主要内容**
  - **High level and low-level I/O**, network programming
  - Internet services, Web servers
  - concurrency, concurrent server design, threads
  - I/O multiplexing with select.
  - **Includes aspects of networking, OS, and architecture.**

L1:Data Lab

- **实验要求：**
  - Students implement simple logical and arithmetic functions, but using a highly restricted subset of C.
  - For example, they must compute the absolute value of a number using only bit-level operations.
- **实验目的：**
  - This lab helps students understand the bit-level representations of C data types and the bit-level behavior of the operations on data.

L2:Bomb Lab

- **实验要求：**
  - A "binary bomb" is a program provided to students as an object code file.
  - When run, it prompts the user to type in 6 different strings. If any of these is incorrect, the bomb ``explodes,'' printing an error message and logging the event on a grading server.
  - Students must ``defuse'' their own unique bomb by disassembling and reverse engineering the program to determine what the 6 strings should be.
- **实验目的：**
  - The lab teaches students to understand assembly language, and also forces them to learn how to use a debugger. It's also great fun. A legendary lab among the CMU undergrads.

L3: Buffer Lab

- **实验要求：**
  - Students modify the run-time behavior of a binary executable by exploiting a buffer overflow bug.
- **实验目的：**
  - This lab teaches the students about the stack discipline and teaches them about the danger of writing code that is vulnerable to buffer overflow attacks.

L4:Performance Lab

- **实验要求：**
  - Students must optimize the performance of an application kernel function such as convolution or matrix transposition.
- **实验目的：**
  - This lab provides a very clear demonstration of the properties of cache memories, and gives students experience with low-level program optimization.

L5: Malloc Lab

- **实验要求：**
  - Students implement their own versions of malloc, free, and realloc.
- **实验目的：**
  - This lab gives students a clear understanding of data layout and organization, and requires them to evaluate different trade-offs between space and time efficiency.
  - One of our favorite labs. When students finish this one, they really understand pointers!

Acknowledgements

- **This course was developed and fine-tuned by Randal E. Bryant and David O'Hallaron. They wrote *The Book*!**
- **http://www.cs.cmu.edu/~./213/schedule.html**