



南京大學  
NANJING UNIVERSITY



# C语言程序举例

南京大学

计算机科学与技术系

袁春风

email: [cfyuan@nju.edu.cn](mailto:cfyuan@nju.edu.cn)

2015.6

# 用“系统思维”分析问题

---

ISO C90标准下，在32位系统上

以下C表达式的结果是什么？

`-2147483648 < 2147483647`

`false`（与事实不符）！Why？

以下关系表达式结果呢？

`int i = -2147483648;`

`i < 2147483647`

`true`！Why？

`-2147483647-1 < 2147483647`，结果怎样？

理解该问题需要知道：

编译器如何处理字面量

高级语言中运算规则

高级语言与指令之间的对应

机器指令的执行过程

机器级数据的表示和运算

.....

# 用“系统思维”分析问题

```
sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生访存异常。但当len为int型时则正常。Why?

访问违例地址为何是0xC0000005?

理解该问题需要知道：

高级语言中运算规则

机器指令的含义和执行

计算机内部的运算电路

异常的检测和处理

虚拟地址空间

.....

Microsoft Visual C++

X

Test.exe: 0xC0000005: Access Violation.

确定

# 用“系统思维”分析问题

---

若x和y为int型，当 $x=65535$ 时， $y=x*x$ ；y的值为多少？

$y=-131071$ 。Why？

现实世界中， $x^2 \geq 0$ ，但在计算机世界并不一定成立。

对于任何int型变量x和y， $(x > y) == (-x < -y)$  总成立吗？

当 $x=-2147483648$ ，y任意（除-2147483648外）时不成立

Why？

在现实世界中成立，

但在计算机世界中并不一定成立。

理解该问题需要知道：  
机器级数据的表示  
机器指令的执行  
计算机内部的运算电路

# 用“系统思维”分析问题

main.c

```
int d=100;
int x=200;
int main()
{
    p1();
    printf ( "d=%d, x=%d\n" , d, x );
    return 0;
}
```

p1.c

```
double d;

void p1( )
{
    d=1.0;
}
```

**打印结果是什么？**

**d=0 , x=1 072 693 248**

**Why ?**

理解该问题需要知道：

机器级数据的表示

变量的存储空间分配

数据的大端/小端存储方式

链接器的符号解析规则

.....

# 用“系统思维”分析问题

/\* 复制数组到堆中，count为数组元素个数 \*/

```
int copy_array(int *array, int count) {
```

```
    int i;
```

```
    /* 在堆区申请一块内存 */
```

```
    int *myarray = (int *) malloc(count*sizeof(int));
```

```
    if (myarray == NULL)
```

```
        return -1;
```

```
    for (i = 0; i < count; i++)
```

```
        myarray[i] = array[i];
```

```
    return count;
```

```
}
```

当 $\text{count}=2^{30}+1$ 时，  
程序会发生什么情况？

理解该问题需要知道：

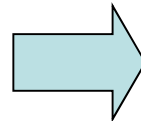
乘法运算及溢出

虚拟地址空间

存储空间映射

.....

当参数count很大时，则  
 $\text{count} \times \text{sizeof}(\text{int})$ 会溢出。  
如 $\text{count}=2^{30}+1$ 时，  
 $\text{count} \times \text{sizeof}(\text{int})=4$ 。



堆 ( heap ) 中大量  
数据被破坏！

# 用“系统思维”分析问题

代码段一：

```
int a = 0x80000000;
```

```
int b = a / -1;
```

```
printf("%d\n", b);
```

运行结果为-2147483648

代码段二：

```
int a = 0x80000000;
```

```
int b = -1;
```

```
int c = a / b;
```

```
printf("%d\n", c);
```

运行结果为“Floating point exception”，显然CPU检测到了溢出异常

为什么两者结果不同！

objdump  
反汇编代码,  
得知除以 -1  
被优化成取  
负指令neg,  
故未发生除  
法溢出

a/b用除法指令IDIV实现，但它不生成OF  
标志，那么如何判断溢出异常的呢？

实际上是“除法错”异常#DE（类型0）  
Linux中，对#DE类型发SIGFPE信号

理解该问题需要知道：

编译器如何优化

机器级数据的表示

机器指令的含义和执行

计算机内部的运算电路

除法错异常的处理

.....

# 用“系统思维”分析问题

---

以下是一段C语言代码：

```
#include <stdio.h>
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

理解该问题需要知道：

IEEE 754 的表示

X87 FPU的体系结构

IA-32和x86-64中过程  
调用的参数传递

计算机内部的运算电路

.....

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印出来的a是一个不确定值

为什么？



# 用“系统思维”分析问题

```
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];
}
```

对于上述C语言函数， $i=0\sim 4$ 时， $\text{fun}(i)$ 分别返回什么值？

$\text{fun}(0) \rightarrow 3.14$   
 $\text{fun}(1) \rightarrow 3.14$   
 $\text{fun}(2) \rightarrow 3.1399998664856$   
 $\text{fun}(3) \rightarrow 2.00000061035156$   
 $\text{fun}(4) \rightarrow 3.14$ , 然后存储保护错

Why ?

理解该问题需要知道：

机器级数据的表示

过程调用机制

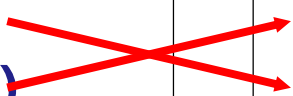
栈帧中数据的布局

.....

# 用“系统思维”分析问题

```
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];
}
```

```
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];
}
```



以上两个程序功能完全一样，算法完全一样，因此，时间和空间复杂度完全一样，执行时间一样吗？

21 times slower  
(Pentium 4)  
**Why ?**

理解该问题需要知道：

数组的存放方式

Cache机制

访问局部性

.....

# 用“系统思维”分析问题

使用老版本gcc -O2编译时，程序一输出0，程序二输出却是1

Why ?

程序一：

```
#include <stdio.h>
double f(int x) {
    return 1.0 / x ;
}
void main() {
    double a, b;
    int i;
    a = f(10) ;
    b = f(10) ;
    i = a == b ;
    printf( "%d\n" , i ) ;
}
```

程序二：

```
#include <stdio.h>
double f(int x) {
    return 1.0 / x ;
}
void main() {
    double a, b, c;
    int i;
    a = f(10) ;
    b = f(10) ;
    c = f(10) ;
    i = a == b ;
    printf( "%d\n" , i ) ;
}
```

# 用“系统思维”分析问题

C/C++ code

```
1  #include "stdafx.h"
2  int main(int argc, char* argv[])
3  {
4      int a=10;
5      double *p=(double*)&a;
6      printf("%f\n", *p);           //结果为0.000000
7      printf("%f\n", (double)a);   //结果为10.000000
8
9      return 0;
10 }
11 为什么printf("%f", *p)和printf("%f", (double)a)结果不一样呢?
```

理解该问题需要知道：

**数据的表示**

**编译（程序的转换）**

**局部变量在栈中的位置**

.....

关键差别在于一条指令：

**fldl 和 fildl**

不都是强制类型转换吗？怎么会不一样

# 你在想什么？

---

- 看了前面的举例，你的感觉是什么呢？
  - 计算机好像不可靠                      从机器角度来说，它永远是对的！
  - 程序执行结果不仅依赖于高级语言语法和语义，还与其他好多方面有关
    - 一点不错！理解程序的执行结果要从系统层面考虑！
  - 本来以为学学编程和计算机基本原理就能当程序员，没想到还挺复杂的，计算机专业不好学
    - 学完“计算机系统基础”就会对计算机系统有清晰的认识，以后再学其他相关课程就容易多了。
  - 感觉要把很多概念和知识联系起来才能理解程序的执行结果
    - 你说对了！把许多概念和知识联系起来就是李国杰院士所提出的“系统思维”。      即：站在“计算机系统”的角度考虑问题！

# 系统能力基于“系统思维”

---

- 系统思维

- 从计算机系统角度出发分析问题和解决问题
- 首先取决于对计算机系统有多了解，“知其然并知其所以然”
  - 高级语言语句都要转换为机器指令才能在计算机上执行
  - 机器指令是一串0/1序列，能被机器直接理解并执行
  - 计算机系统是模运算系统，字长有限，高位被丢弃
  - 运算器不知道参加运算的是带符号数还是无符号数
  - 在计算机世界， $x*x$ 可能小于0， $(x+y)+z$ 不一定等于 $x+(y+z)$
  - 访问内存需几十到几百个时钟，而访问磁盘要几百万个时钟
  - 进程具有独立的逻辑控制流和独立的地址空间
  - 过程调用使用栈存放参数和局部变量等，递归过程有大量额外指令，增加时间开销，并可能发生栈溢出
  - .....

只有先理解系统，才能改革系统，并应用好系统!

# 什么是计算机系统？

## 计算机系统抽象层的转换

程序执行结果

不仅取决于  
算法、程序编写

而且取决于  
语言处理系统  
操作系统

ISA

微体系结构

不同计算机课程  
处于不同层次

必须将各层次关  
联起来解决问题



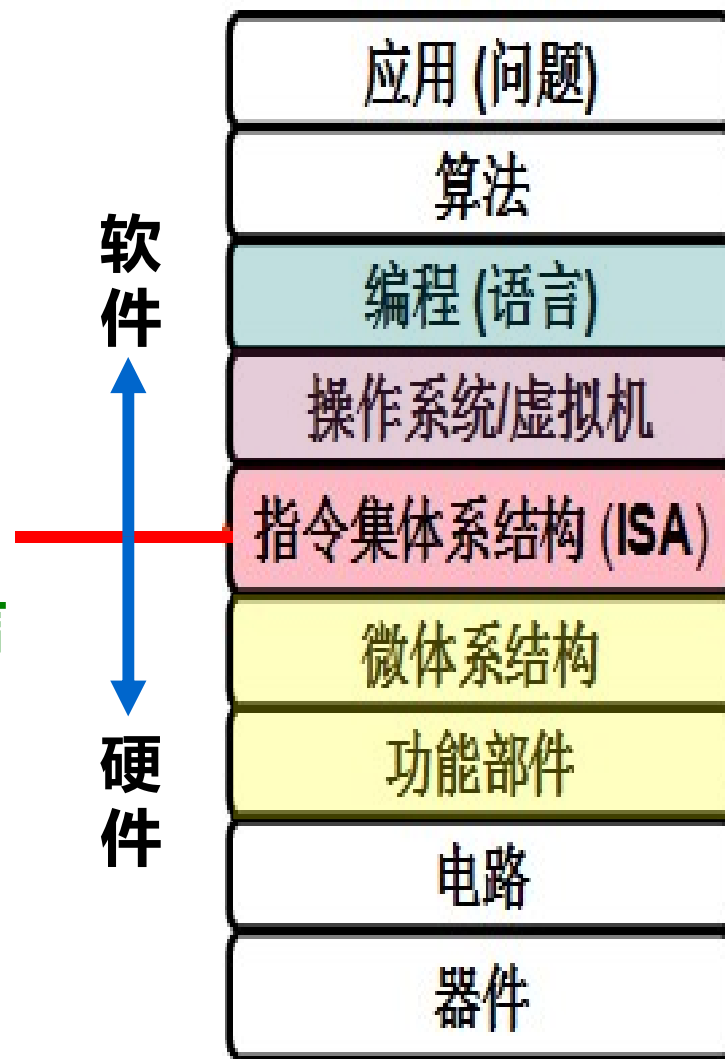
# “计算机系统基础” 内容提要

**课程目标：**使学生清楚理解计算机是如何生成和运行可执行文件的！

**重点在高级语言以下各抽象层**

- **C语言程序设计层**
  - 数据的机器级表示、运算
  - 语句和过程调用的机器级表示
- **操作系统、编译和链接的部分内容**
- **指令集体系结构 (ISA) 和汇编层**
  - 指令系统、机器代码、汇编语言
- **微体系结构及硬件层**
  - CPU的通用结构
  - 层次结构存储系统

**计算机系统抽象层**





# 为什么要学习“计算机系统基础”？

---

- 为什么要学习“计算机系统基础”呢？
  - 为了编程序时少出错
  - 为了在程序出错时很快找到出错的地方
  - 为了明白程序是怎样在计算机上执行的
  - 为了强化“系统思维”
  - 为了更好地理解计算机系统，从而编写出更好的程序
  - 为后续课程的学习打下良好基础
  - 为了编写出更快的程序
  - 为了更好地认识计算机系统
  - .....

**本课程属于计算机系统基础（一）**

**下面就开始本课程的学习吧！**