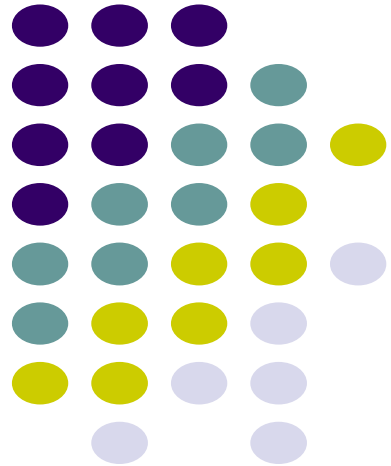
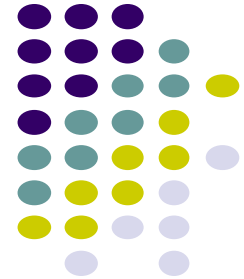
《计算机系统基础（四）：编程与调试实践》

C语言编程实践



概述

- 从高层语言角度解释计算机系统对程序性能的影响，加深对IA-32架构下程序各组成部分的理解
- 以c语言为例，介绍若干编程实践：
  - 位操作
  - 浮点精度
  - Cache
  - 异常处理等；
- 实验环境：Linux 32-bit i386，C/汇编语言



位运算

逻辑运算

操作	C语言操作符	汇编指令
按位取反	~	notb、notw、notl
按位与	&	andb、andw、andl
按位或	|	orb、orw、orl
按位异或	^	xorb、xorw、xorl

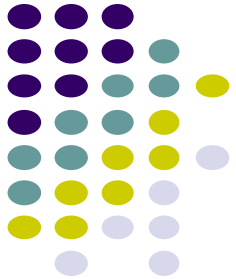
- 注意与C语言的逻辑与（&&）、逻辑或（||）、逻辑非（！）操作之间的区别

移位运算

- 移位运算（左/右移时，最高/最低位送CF）

操作	C语言操作符	汇编指令
逻辑左移	<<	shlb、shlw、shll
算术左移	<<	salb、salw、sall
逻辑右移	>>	shrb、shrwshrl
算术右移	>>	sarb、sarw、sarl

Why位运算？



- 特定功能
  - 取特定位、保留特定位、……
- 速度快
  - 左移、右移可用于实现快速的整数乘、除法
- 其他
  - 原位交换、……

交换变量a与b的值

- 普通方法

  c = a; a = b; b = c;

- 位操作交换法

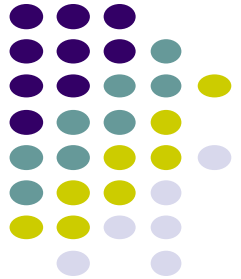
  a = a^b; b = b^a; a = a^b;

- 原理

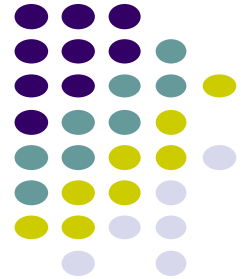
  b = b^(a^b) = b^a^b = b^b^a = a

  a = (a^b)^(b^(a^b)) = a^b^b^a^b = b

小结



- 本节课介绍了C语言中位操作的主要内容，包括逻辑操作和移位操作。

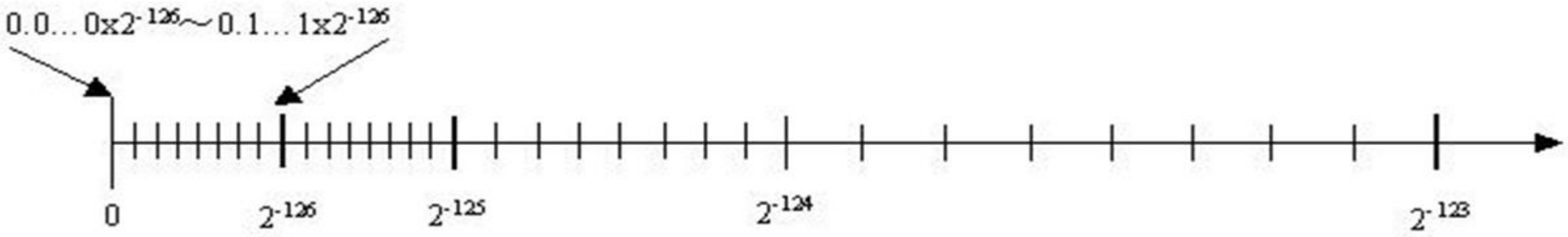


浮点数精度

IEEE754单精度浮点数

Single Precision：

S	Exponent	Significand
1 bit	8 bits	23 bits



浮点数精度

jie@debian: ~/class/ch2/precision

```
jie@debian:~/class/ch2/precision$ ./main
1234567936.000000
61.419998
61.419998
61.419998
61.419998
61.420002
61.420002
1.19999981
jie@debian:~/class/ch2/precision$
```

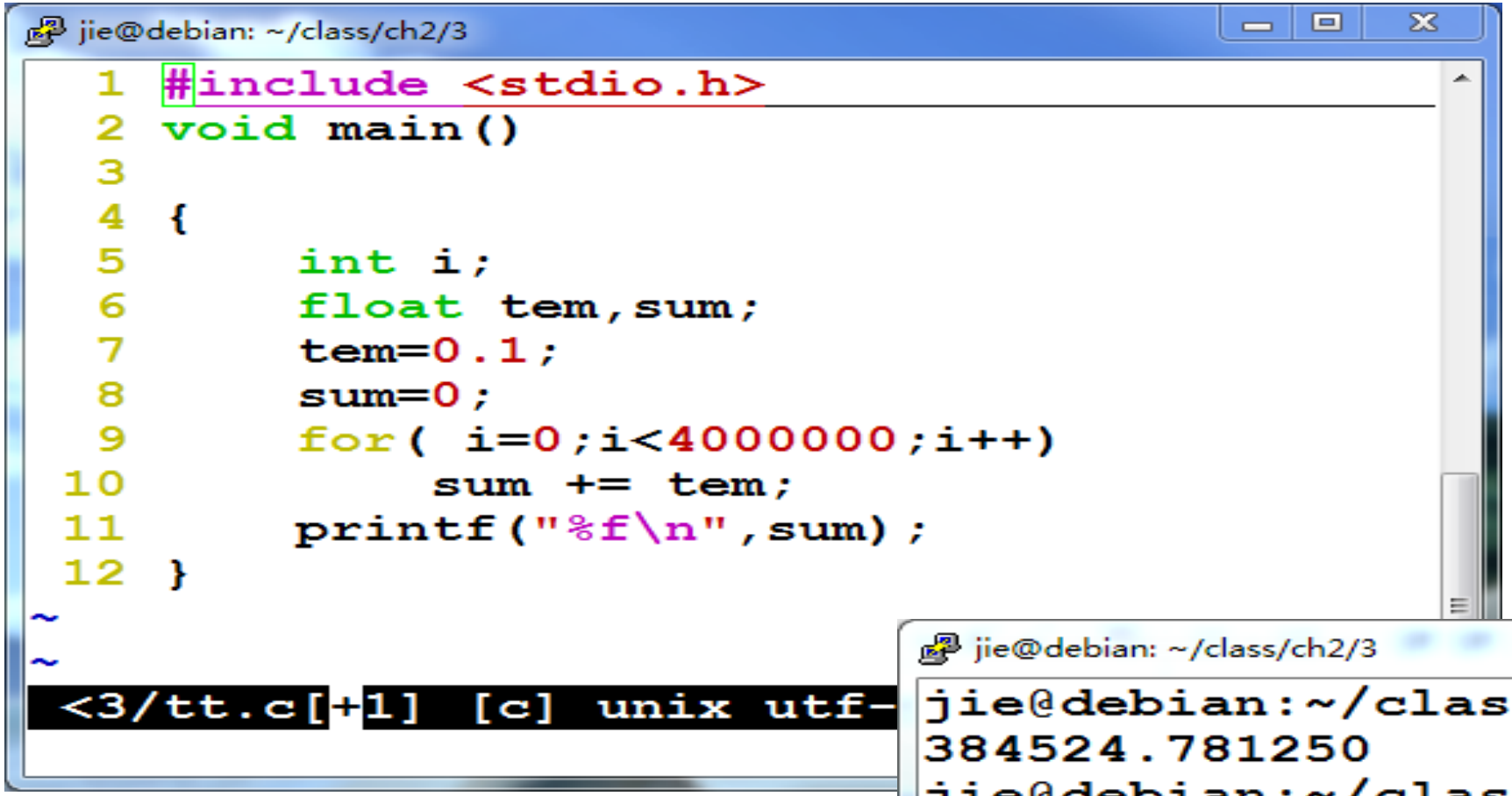
jie@debian: ~/class/ch2/precision

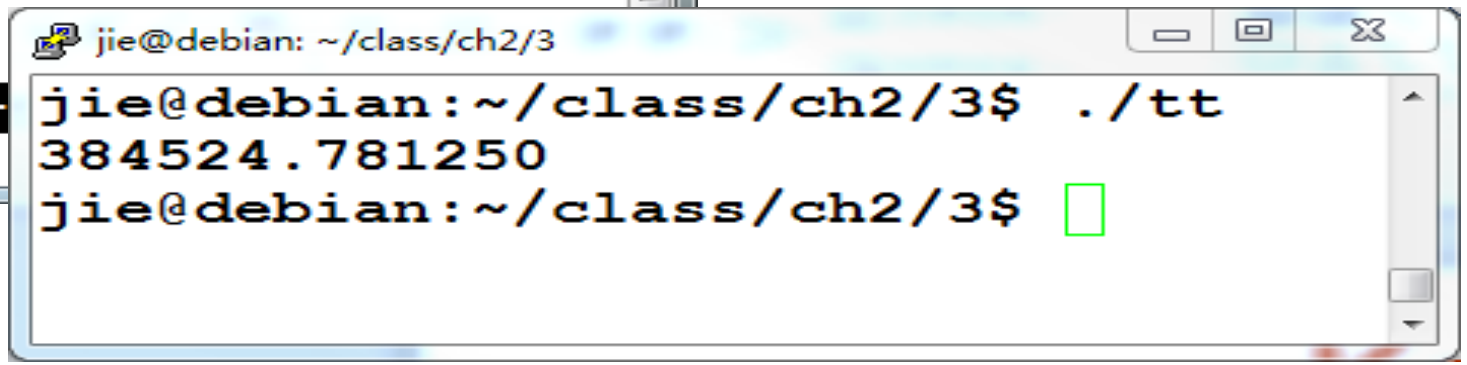
```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5       float tem[10];
 6       float a = 10.2;
 7       float b = 9;
 8       float c;
 9       int i;
10       tem[0] = 1234567890;
11       tem[1]= 61.419997;
12       tem[2]=61.419998;
13       tem[3]=61.419999;
14       tem[4] = 61.42;
15       tem[5]=61.420001;
16       tem[6]=61.420002;
17       for(i=0;i<7;i++)
18           printf("%.6f\n", tem[i]);
19       c = a-b;
20       printf ("%.8f\n",c);
21
22       return 0;
23 }
~
<n/main.c[1] [c] unix utf-8 Ln 1, Col 1/2
```

61.419998和61.420002是两个可表示数，两者之间相差0.000004。当输入数据是一个不可表示数时，机器将其转换为最邻近的可表示数。

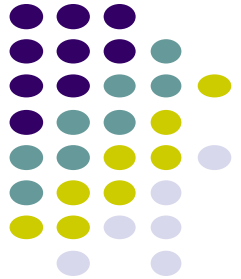
单精度浮点数的有效位数为7

大数吃小数问题


jie@debian: ~/class/ch2/3
1 #include <stdio.h>
2 void main()
3
4 {
5 int i;
6 float tem,sum;
7 tem=0.1;
8 sum=0;
9 for(i=0;i<4000000;i++)
10 sum += tem;
11 printf("%f\n",sum);
12 }
~
~
<3/tt.c[+1] [c] unix utf-


jie@debian: ~/class/ch2/3
jie@debian:~/class/ch2/3$ ./tt
384524.781250
jie@debian:~/class/ch2/3$

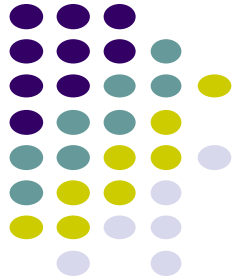
大数吃小数问题



- “对阶”操作：目的是使两数阶码相等
  - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
  - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

进行尾数加减运算前，必须“对阶”！最后还要考虑舍入

Kahan累加算法



sum = 10000.0 + 3.14159 + 2.71828
= 10003.14159 + 2.71828
= 10005.85987
= 10005.9

sum = 10000.0 + 3.14159
= 10003.14159
= 10003.1
sum = 10003.1+2.71828
= 10005.81828
= 10005.8

Kahan累加算法

sum = 10000.0;
c = 0;
y = 3.14159 – c
  = 3.14159 – 0
  = 3.14159
t = sum + y
  = 10000.0 + 3.14159
  = 10003.1
c = (t – sum) - y
  = (10003.1 - 10000.0) - 3.14159
  = 3.10000 - 3.14159
  = -.0415900
sum = t
  = 10003.1

y = 2.71828 - c
  = 2.71828 - -.0415900
  = 2.75987
t = sum + y
  = 10003.1 + 2.75987
  = 10005.9
c = (t – sum) - y
  = (10005.9 - 10003.1) - 2.75987
  = 2.80000 - 2.75987
  = .040130
sum = t
  = 10005.9

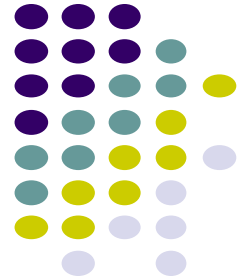
Kahan累加算法

1989年度的图灵奖获得者，
浮点数之父William Kahan

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float sum = 0;
6      float c = 0;
7      float y, t;
8      int i;
9      for( i=0;i<4000000;i++)
10     {
11         y = 0.1 -c;
12         t = sum +y;
13         c = (t-sum)-y;
14         sum = t;
15     }
16     printf("sum=%f\n",sum);
17
18     return 0;
19 }
```

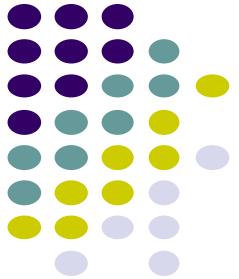
小结

- 本节课针对浮点数的精度问题进行了介绍，并通过Kahan算法进行了详细的解释，希望能够对大家进一步理解浮点数精度问题有所帮助。



Cache友好代码

Cache友好的代码



- 尽可能地重复使用一个数据（时间局部性）
- 尽可能跨距为1地方问数据（空间局部性）

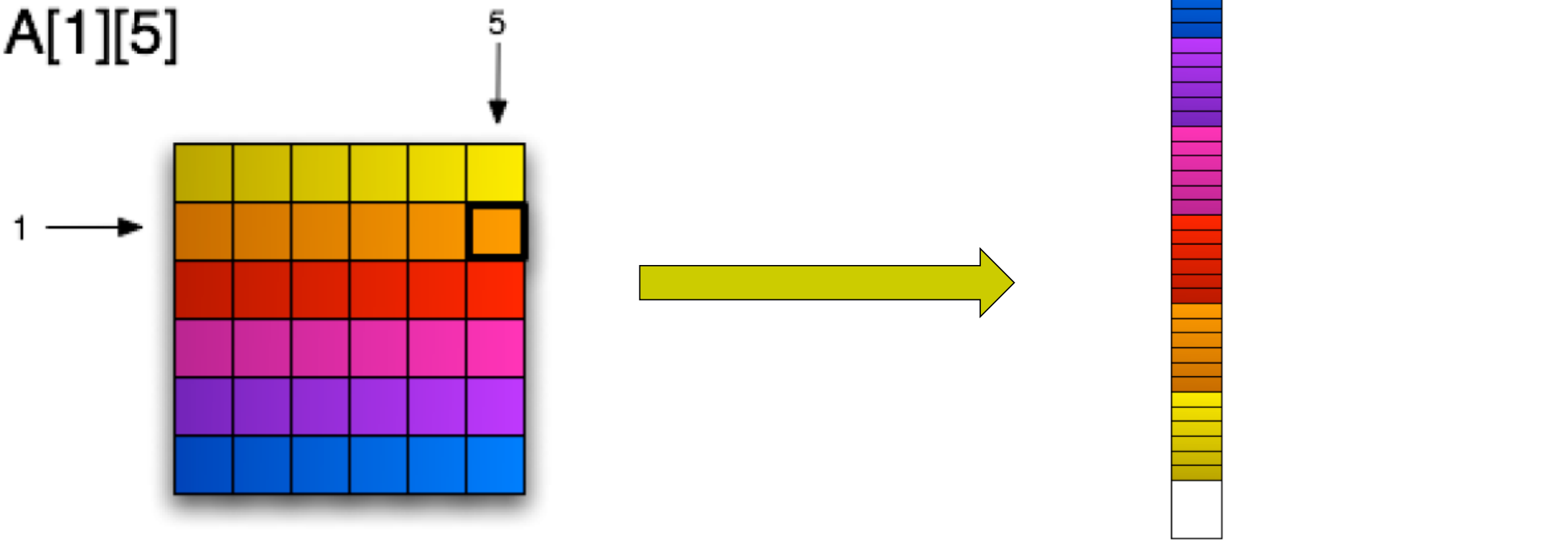
矩阵相乘

```
for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                for(k=0;k<n;k++)
                        C[i+j*n]+=A[i+k*n]*B[k+j*n];
```

6种循环顺序

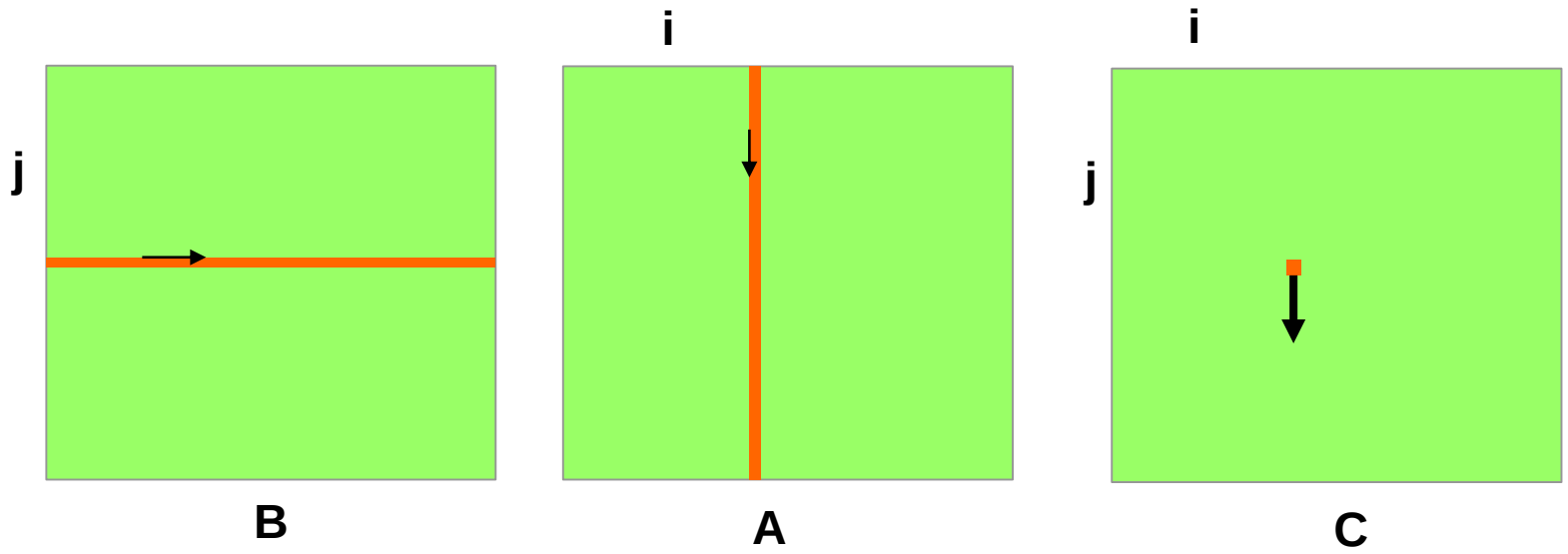
ijk、ikj、jik、jki、kij、kji

矩阵的存储



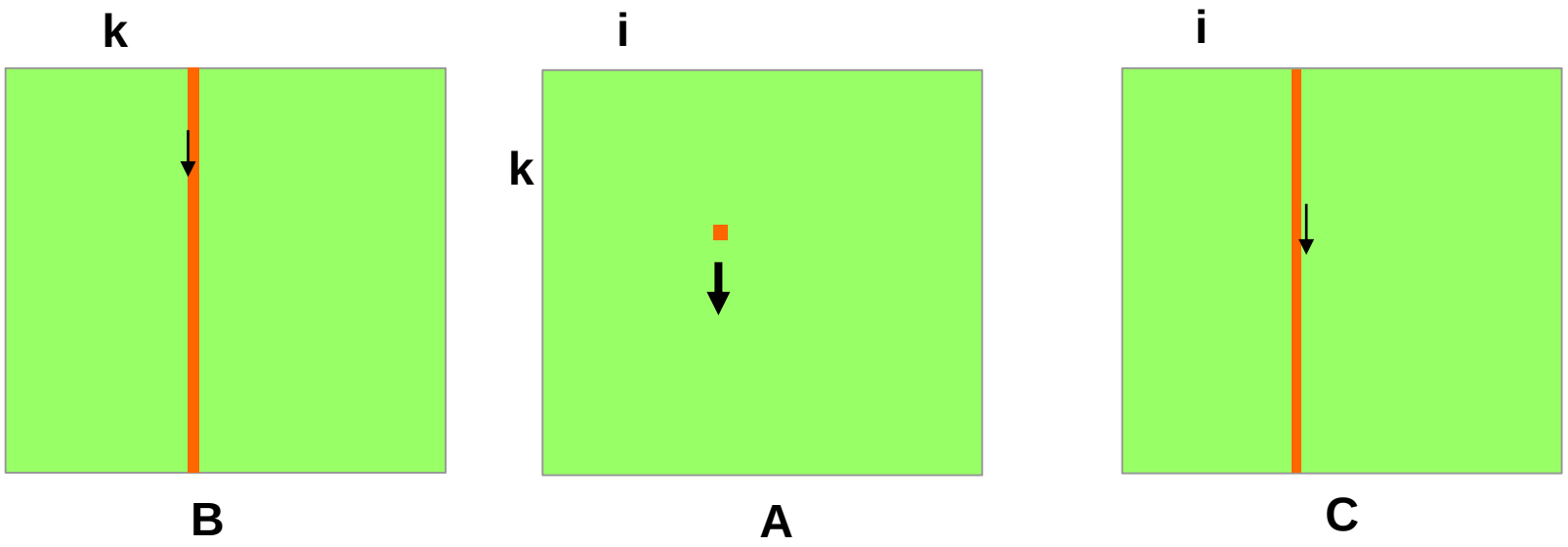
矩阵相乘——ijk

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        for(k=0;k<n;k++)
            C[i+j*n]+=A[i+k*n]*B[k+j*n];
```



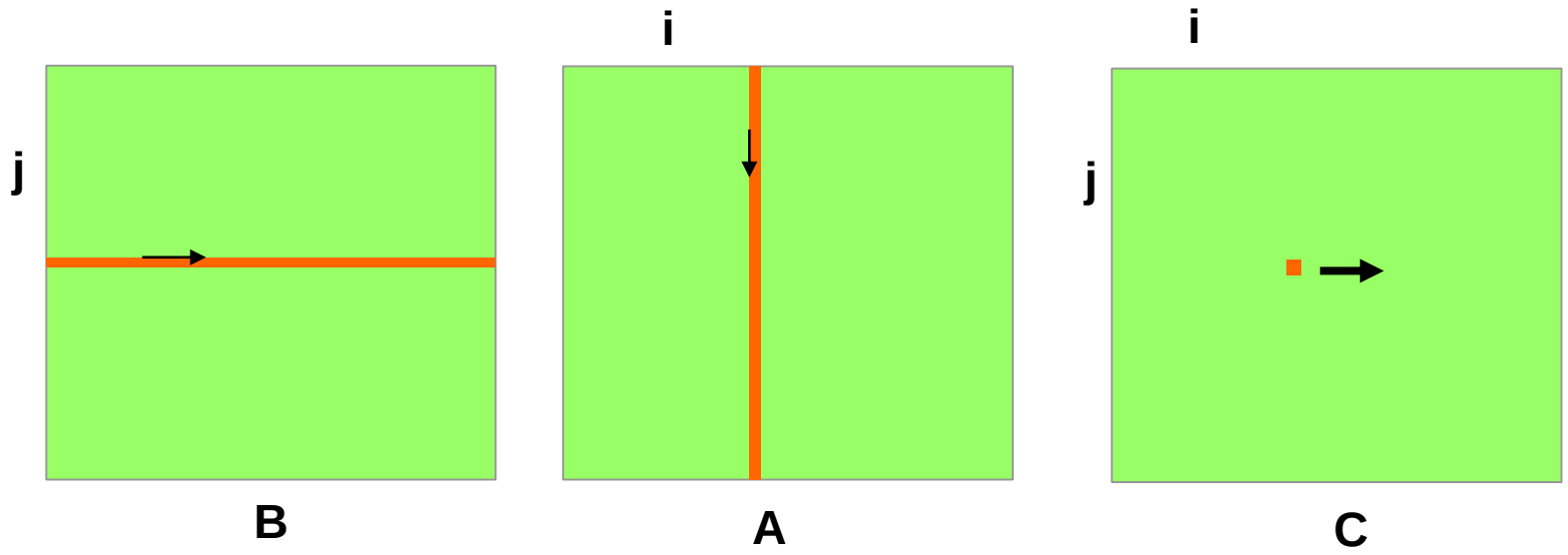
矩阵相乘——ikj

```
for(i=0;i<n;i++)
    for(k=0;k<n;k++)
        for(j=0;j<n;j++)
            C[i+j*n]+=A[i+k*n]*B[k+j*n];
```



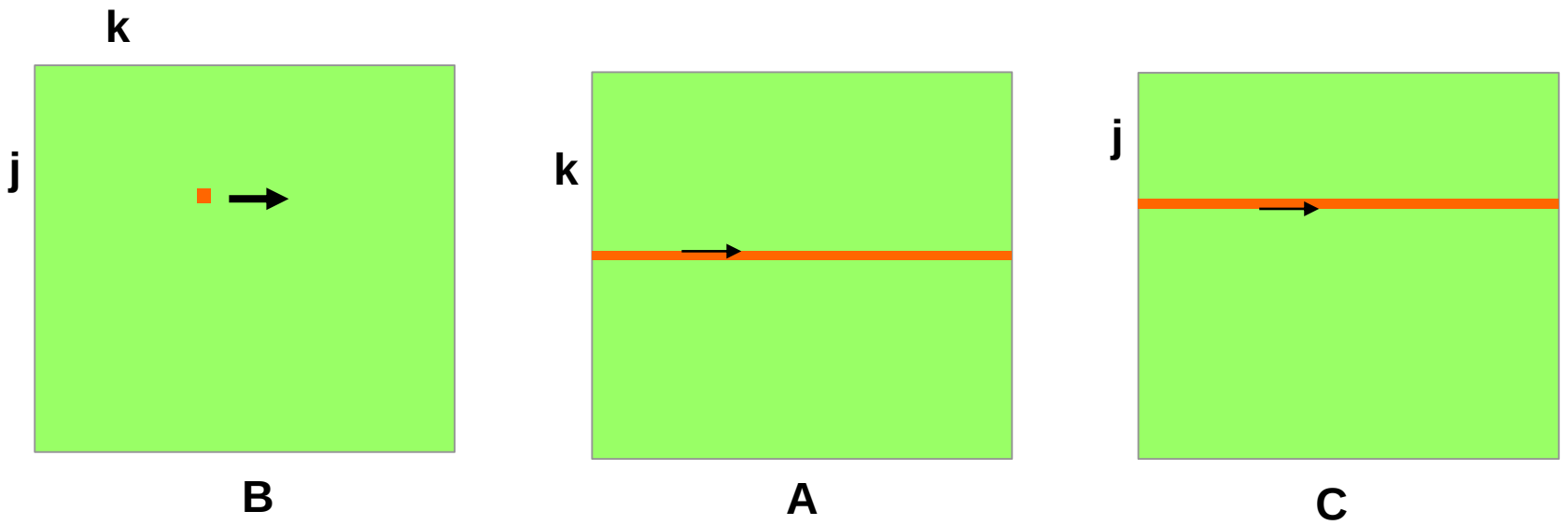
矩阵相乘——jik

```
for(j=0;j<n;j++)
    for(i=0;i<n;i++)
        for(k=0;k<n;k++)
            C[i+j*n]+=A[i+k*n]*B[k+j*n];
```

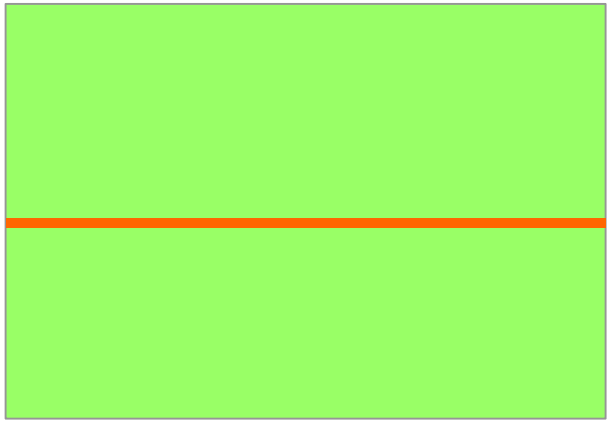


矩阵相乘——jki

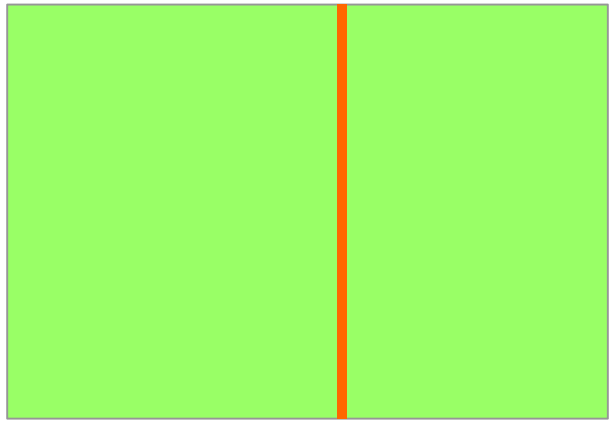
```
for(j=0;j<n;j++)
    for(k=0;k<n;k++)
        for(i=0;i<n;i++)
            C[i+j*n]+=A[i+k*n]*B[k+j*n];
```



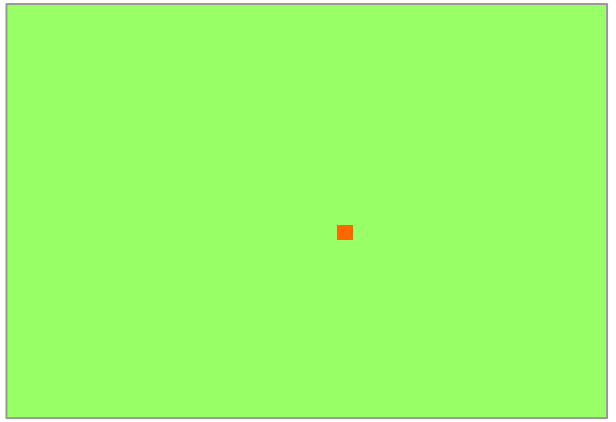
分块矩阵相乘



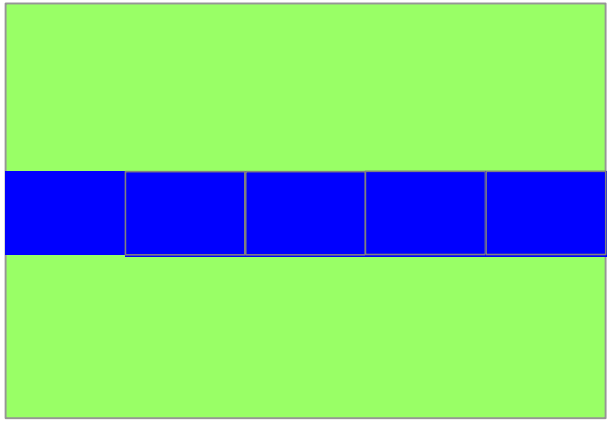
B



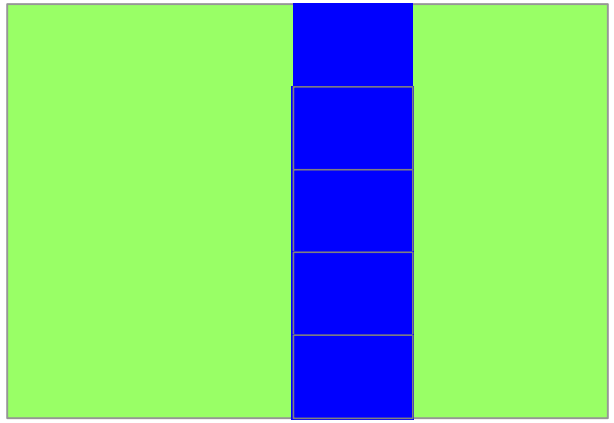
A



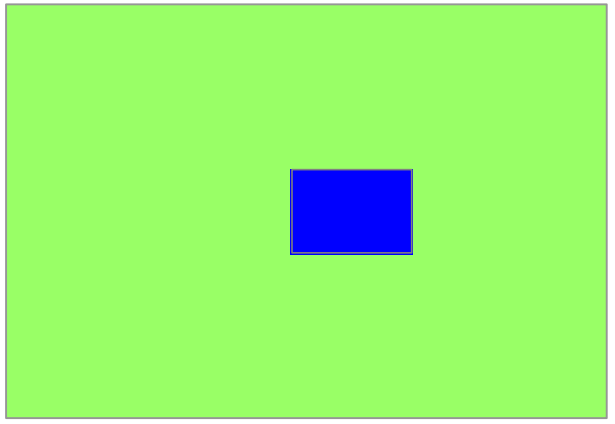
C



B

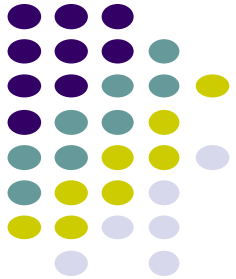


A

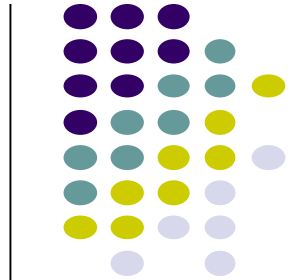


C

小结



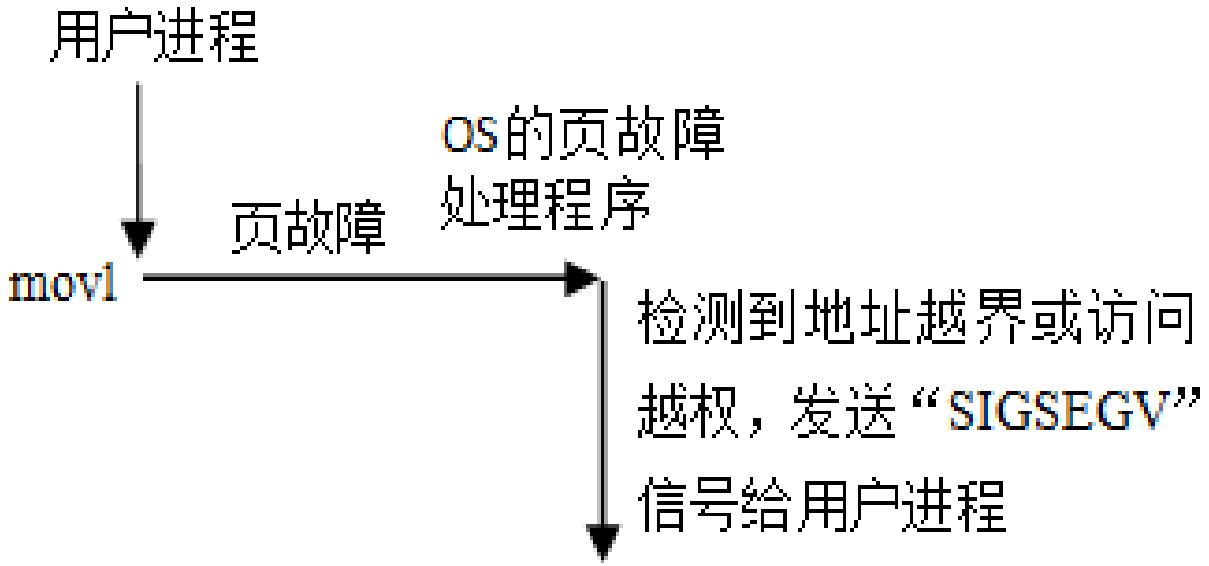
- 可以通过编程来充分利用cache性能
  - 数据的组织形式
  - 数据的访问形式
    - 嵌套的循环调用
    - 分块
- 所有的系统都希望“cache友好代码”
  - 获得最佳cache性能与平台相关
    - Cache大小、cache行大小、映射策略等
  - 通用的法则
    - 工作集越小越好
    - 访问跨距越小越好



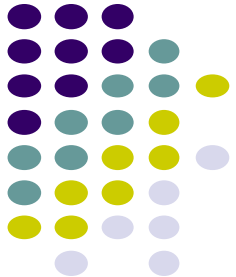
异常处理

IA32的异常的处理

- 硬件
  - 异常检测
  - 保护现场
  - 跳转到中断服务程序
  - 恢复现场
- 软件
  - 中断服务程序
  - 异常处理程序
  - 信号处理程序



异常的处理

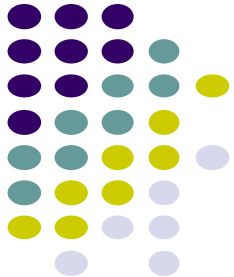


- 异常处理程序发送相应的信号给发生异常的当前进程，或者进行故障恢复，然后返回到断点处执行。
- 采用向发生异常的进程发送信号的机制实现异常处理，可尽快完成在内核态的异常处理过程，因为异常处理过程越长，嵌套执行异常的可能性越大，而异常嵌套执行会付出较大的代价。
- 并不是所有异常处理都只是发送一个信号到发生异常的进程。

Linux中异常对应的信号名和处理程序名

类型号	助记符	含义描述	处理程序名	信号名
0	#DE	除法出错	divide_error( )	SIGFPE
1	#DB	单步跟踪	debug( )	SIGTRAP
2		NMI 中断	nmi( )	无
3	#BP	断点	int3( )	SIGTRAP
4	#OF	溢出	overflow( )	SIGSEGV
5	#BR	边界检测（BOUND）	bounds( )	SIGSEGV
6	#UD	无效操作码	invalid( )	SIGILL
7	#NM	协处理器不存在	device_not_available( )	无
8	#DF	双重故障	doublefault( )	无
9	#MF	协处理器段越界	coprocessor_segment_overrun( )	SIGFPE
10	#TS	无效 TSS	invalid_tss( )	SIGSEGV
11	#NP	段不存在	segment_not_present( )	SIGBUS
12	#SS	栈段错	stack_segment( )	SIGBUS
13	#GP	一般性保护错（GPF）	general_protecton( )	SIGSEGV
14	#PF	页故障	page_fault( )	SIGSEGV
15		保留	无	无
16	#MF	浮点错误	coprocessor_error( )	SIGFPE
17	#AC	对齐检测	alignment_check( )	SIGSEGV
18	#MC	机器检测异常	machine_check( )	无
19	#XM	SIMD 浮点异常	simd_coprocessor_error( )	SIGFPE

setjump和longjump



- setjump(env) 将程序中的上下文存储在env中，包括程序计数器、栈帧、通用寄存器等；
- longjmp(env, status)；env为指代 setjmp 所保存的函数执行状态的变量，status用于让setjump函数返回的值；

小结

- 本次课介绍了信号处理函数，并以实例演示了如何自定义一个信号处理函数，希望同学们课后自行练习相应的代码！