第四章 程序的机器级表示

C语言程序的机器级表示
复杂数据类型的分配和访问
越界访问和缓冲区溢出

程序的转换与机器级表示

- 主要教学目标
  - 了解高级语言源程序中的语句与机器级代码之间的对应关系
  - 了解复杂数据类型（数组、结构等）的机器级实现
- 主要教学内容
  - 介绍C语言程序与机器级指令之间的对应关系。
  - 主要包括：C语言中控制语句和过程调用等机器级实现、复杂数据类型（数组、结构等）的机器级实现等。
  - 本章所用的机器级表示主要以汇编语言形式表示为主。

简言之：理解编译器、汇编器各自的输入和输出

程序的机器级表示

- 分以下四个部分介绍
  - 第一讲：　过程调用的机器级表示
    - 过程调用约定、变量的作用域和生存期
    - 按值传参和按地址传参、递归过程调用
    - 非静态局部变量的存储分配
    - 入口参数的传递和分配
  - 第二讲：　流程控制语句的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

过程调用的机器级表示

以下过程（函数）调用对应的机器级代码是什么？

如何将t1(125)、t2(80)分别传递给add中的形式参数x、y

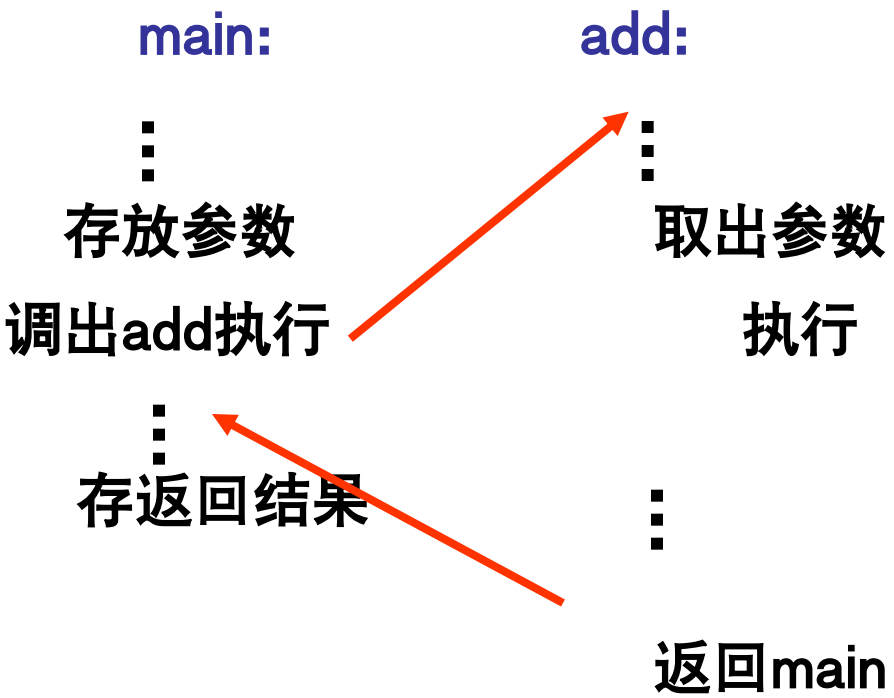
add函数执行的结果如何返回给caller?

```
int add ( int x, int y ) {
    return x+y;
}
int main ( ) {
    int t1 = 125;
        int t2 = 80;
    int sum = add (t1, t2);
    return sum;
}
```

add

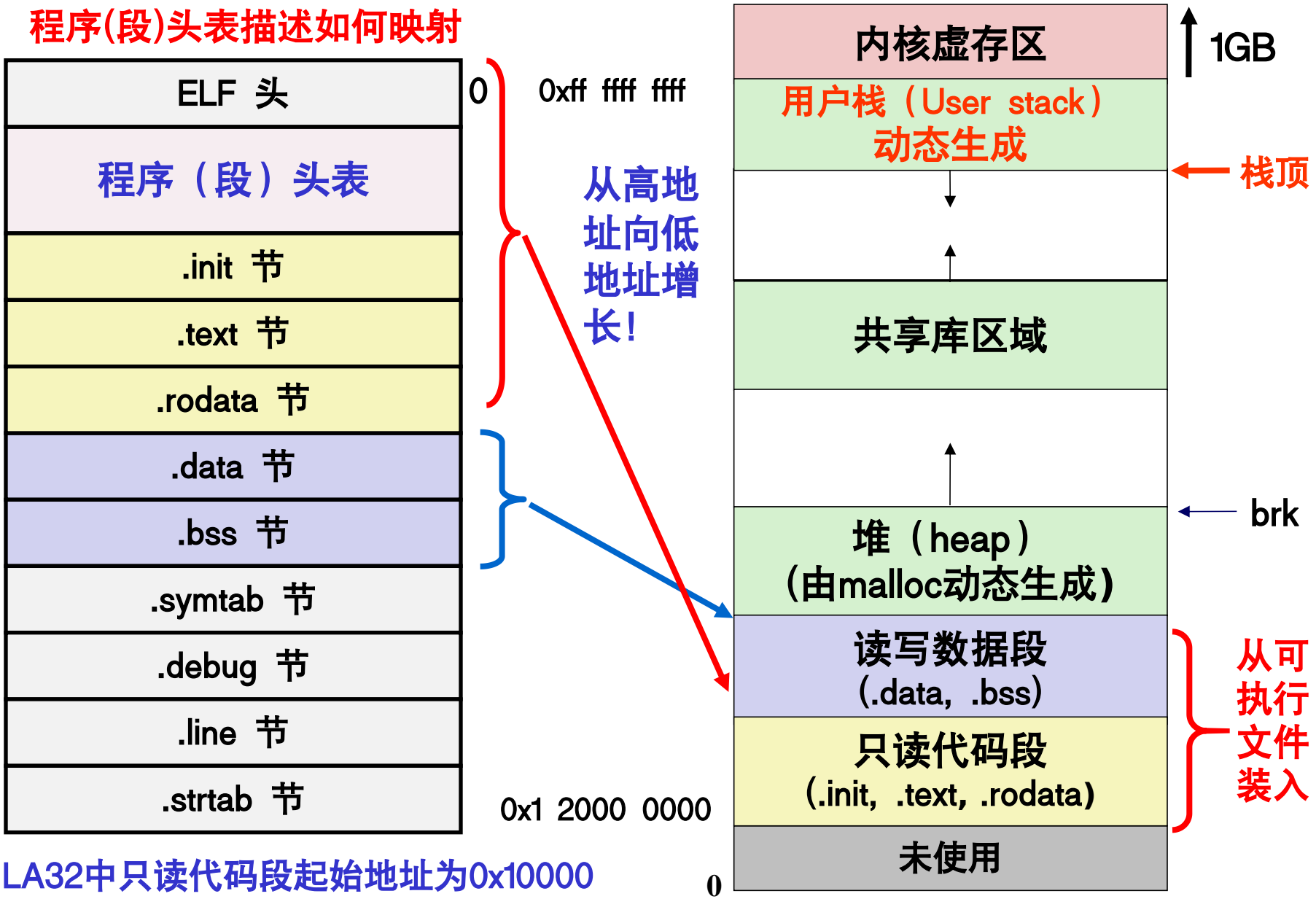
↑

main

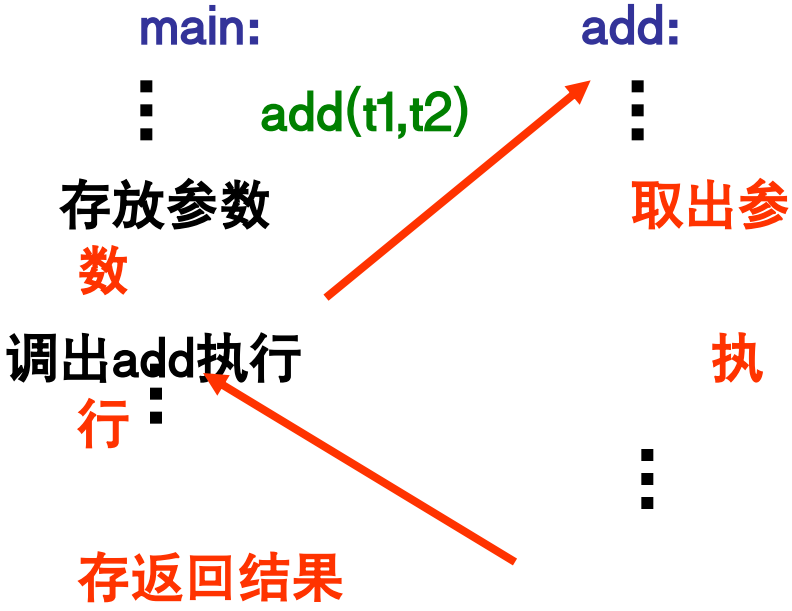


为了统一，模块代码之间必须遵循调用接口约定，称为调用约定(calling convention)，具体由ABI规范定义，编译器强制执行，汇编语言程序员也必须强制按照这些约定执行，包括寄存器的使用、栈帧的建立和参数传递等。

可执行文件的存储器映像



过程调用的机器级表示



何为现场?

通用寄存器的内容!

为何要保存现场?

因为所有过程共享一套通用寄存器

想象：妈妈做菜过程中，让你来完成其中一个工序时共用一套盘子的情况

过程调用的执行步骤(假设P为调用者，Q为被调用者)

（1）P将入口参数（实参）放到Q能访问到的地方；

（2）P将返回地址放到Q能取到的地方，然后将控制转移到Q；

P过程

调用指令

（3）Q保存P的现场，并为非静态局部变量分配空间；

准备阶段

（4）执行Q的过程体（函数体）；处理阶段

（5）Q恢复P的现场，释放局部变量空间；

（6）Q取出返回地址，将控制转移到P。返回指令

结束阶段

Q过程

过程调用所使用的栈

- 在过程P和Q中，需要为入口参数、返回地址、过程执行时用到的临时数据、过程中的auto变量、过程返回结果等找到存放空间。
- 若有足够的寄存器，最好都保存在寄存器中，可快速取得这些数据。
- 用户可见寄存器有限且被所有过程共享；此外，过程中使用的复杂类型auto型变量（如数组和结构等）不可能保存在寄存器中。因此，除通用寄存器外，还需要有一个专门的存储区保存这些数据，这个存储区就是栈（stack）。
- 实现过程的嵌套或递归调用，也需使用栈保存信息。

那么，上述数据中哪些存放在寄存器、哪些存放在栈中呢?

寄存器是所有过程共享的资源，若在调用过程中存放值x，在被调用过程中又被写入了新值y，那么，返回到调用过程执行时，执行结果就会发生错误。因而，使用寄存器需遵循一定的惯例，即使用约定。

过程调用时寄存器使用约定

- LoongArch中**通用寄存器**使用约定

名称	别名	用途	被调用者保存
$r0	$zero	常数 0	（常数）
$r1	$ra	返回地址	否
$r2	$tp	线程指针	（不可分配）
$r3	$sp	栈指针	是
$r4~$r5	$a0~$a1	参数寄存器、返回值寄存器	否
$r6~$r11	$a2~$a7	参数寄存器	否
$r12~$r20	$t0~$t8	临时寄存器	否
$r21		保留	（不可分配）
$r22	$fp/$s9	栈帧指针/保存寄存器	是
$r23~$r31	$s0~$s8	保存寄存器	是

过程调用时寄存器使用约定

- LoongArch中浮点寄存器使用约定

名称	别名	用途	被调用者保存
$f0~$f1	$fa0~$fa1	参数寄存器、返回值寄存器	否
$f2~$f7	$fa2~$fa7	参数寄存器	否
$f8~$f23	$ft0~$ft15	临时寄存器	否
$f24~$f31	$fs0~$fs7	保存寄存器	是

假设过程P调用过程Q

fa0~fa7用于传送前8个浮点数入口参数

fa0~fa1用于传送从Q返回的浮点结果

ft0~ft15属于临时寄存器，与fa0~fa7一样，在Q中无需保存，可按需使用，其内容可能会被Q破坏。若从Q返回后P需要继续使用它们，则P在调用Q前，应将它们先入栈保存，从Q返回后，P再恢复之

fs0~fs7属于被调用者保存寄存器，也称为静态寄存器，若Q需使用它们，则应先入栈保存，在返回P前恢复它们

ABI规定的过程调用约定

- LoongArch ABI规范约定

1）ra为专门的返回地址寄存器，BL指令执行时将返回地址写入ra（即r1）

2）a0~a7用于传送前8个非浮点数入口参数。若参数超过8个，则其余参数保存在栈中。 a0~a7属于调用者保存寄存器，无需在被调用过程Q中保存

3）a0~a1用于传送非浮点返回值，在Q中先将返回值写入a0~a1再返回P

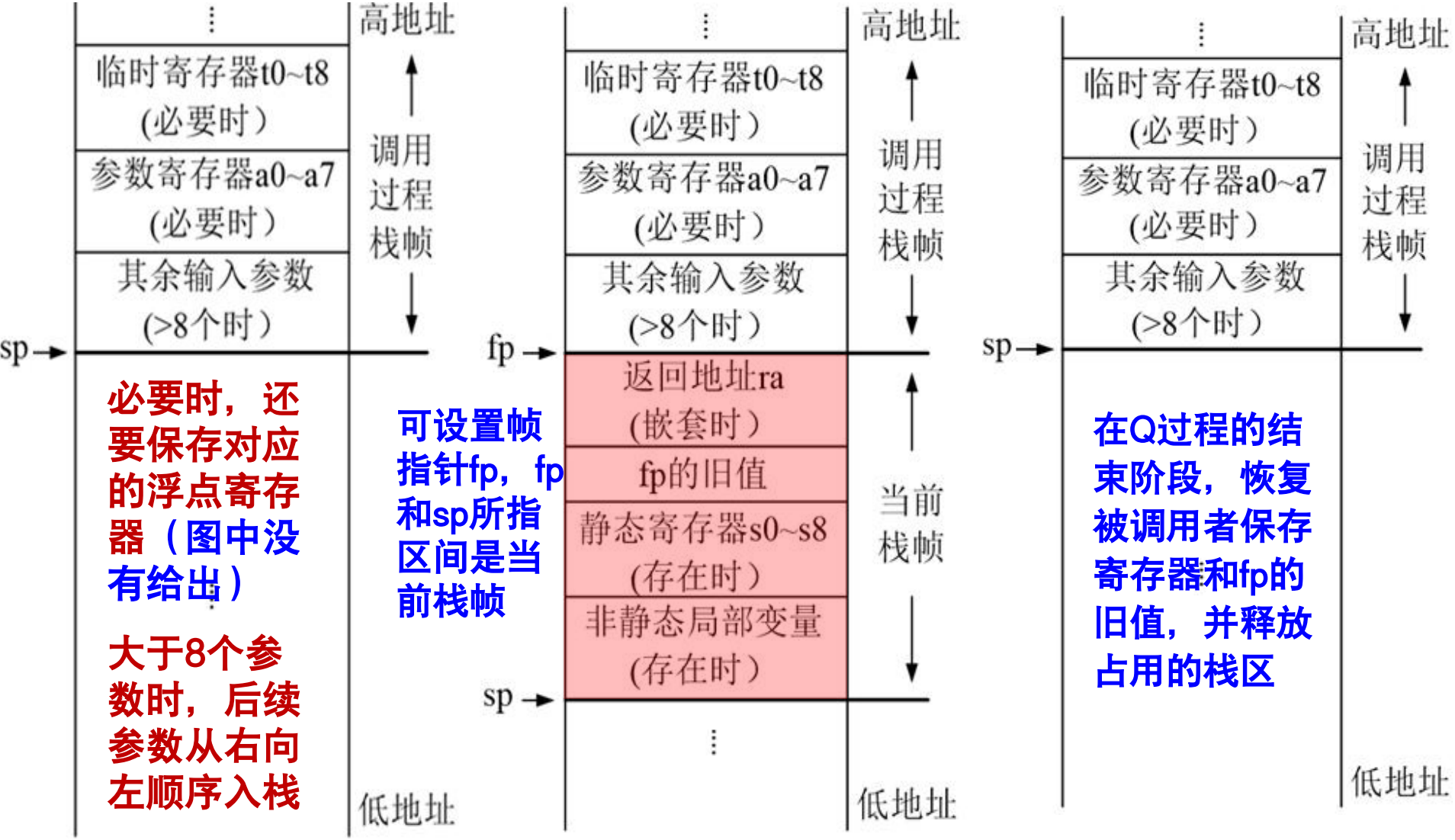
4）t0~t8是调用者保存寄存器（caller saved register）。当P调用Q时，Q可直接使用它们，若P在从Q返回后还要用它们，则P应在转到Q前先保存之，并在从Q返回后先恢复再使用，t0~t8也被称为临时寄存器

5）s0~s8是被调用者保存寄存器（callee saved register）。Q须先保存之，并在返回P前先恢复。在LoongArch中s0~s8被称为静态寄存器。

6）fa0~fa7用于传送前8个浮点入口参数；ft0~ft15属于临时寄存器；fs0~fs7属于被调用者保存寄存器，也称为静态寄存器

7）fp和sp分别是帧指针和栈指针寄存器，分别指向当前栈帧的底部和顶部。通过将fp或sp作为基址寄存器来访问非静态局部变量和入口参数

过程调用栈和栈帧



(a) 过程 Q 被调用前

(b) 过程 Q 执行中

(c) 返回过程 P 前

一个简单的过程调用例子

```
int   caller ( ) {
      int   t1 = 125;
        int          t2 = 80;
      int   sum = add (t1, t2);
      return sum;
}
```

```
准备阶段:
addi.w   $r3, $r3, -32(0xfe0)     #R[sp]←R[sp]-32，生成一个新的栈帧
st.w     $r1, $r3, 28(0x1c)       #M[R[sp]+28]←R[ra]，保存返回地址
st.w     $r22, $r3, 24(0x18)      #M[R[sp]+24]←R[fp]，保存fp的旧值
addi.w   $r22, $r3, 32(0x20)      #R[fp]←R[sp]+32，生成新的fp帧指针

分配局部变量:
addi.w   $r12, $r0, 125(0x7d)     #R[r12]←125，t1=125
st.w     $r12, $r22, -20(0xfec)   #M[R[fp]-20]←R[r12]，t1入栈
addi.w   $r12, $r0, 80(0x50)      #R[r12]←80，t2=80
st.w     $r12, $r22, -24(0xfe8)   #M[R[fp]-24]←R[r12]，t2入栈

传参并调用:
ld.w     $r5, $r22, -24(0xfe8)    #R[r5]←M[R[fp]-24]，参数t2传给a1
ld.w     $r4, $r22, -20(0xfec)    #R[r4]←M[R[fp]-20]，参数t1传给a0
bl       -88(0xfffffa8)           #调用add，将返回地址保存在ra中

传返回值:
st.w     $r4, $r22, -28(0xfe4)    #M[R[fp]-28]←R[r4]，返回值存sum
ld.w     $r12, $r22, -28(0xfe4)   #R[r12]←M[R[fp]-28]，读取sum
move     $r4, $r12          #R[r4]←R[r12]，sum作为返回值送a0

结束阶段:
ld.w     $r1, $r3, 28(0x1c)       #R[ra]←M[R[sp]+28]，读取返回地址
ld.w     $r22, $r3, 24(0x18)      #恢复帧指针fp的旧值
addi.w   $r3, $r3, 32(0x20)       #释放栈帧空间
jirl     $r0, $r1, 0               #返回调用过程
```

的过程调用例子

```
int add ( int x, int y ) {
      return x+y;
}
int   caller ( ) {
      int   t1 = 125;
        int             t2 = 80;
      int   sum = add (t1, t2);
      return sum;
}
```

add → caller → P

```
addi.w    $r3, $r3, -32(0xfe0)
st.w      $r1, $r3, 28(0x1c)
st.w      $r22, $r3, 24(0x18)
addi.w    $r22, $r3, 32(0x20)
addi.w    $r12, $r0, 125(0x7d)
st.w      $r12, $r22, -20(0xfec)
addi.w    $r12, $r0, 80(0x50)
st.w      $r12, $r22, -24(0xfe8)
ld.w      $r5, $r22, -24(0xfe8)
ld.w      $r4, $r22, -20(0xfec)
bl        -88(0xfffffa8)
st.w      $r4, $r22, -28(0xfe4)
ld.w      $r12, $r22, -28(0xfe4)
move      $r4, $r12
ld.w      $r1, $r3, 28(0x1c)
ld.w      $r22, $r3, 24(0x18)
addi.w    $r3, $r3, 32(0x20)
jirl      $r0, $r1, 0
```

准备阶段

分配局部变量

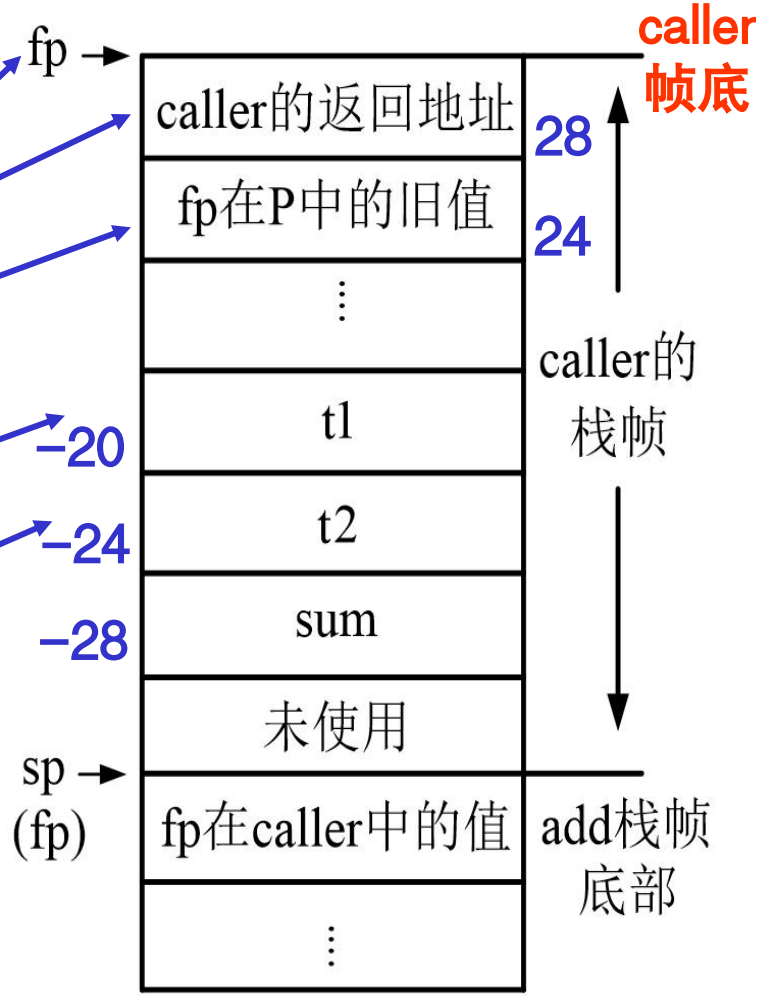
传参并调用

传返回值

结束阶段

释放栈帧

返回指令



r1: 返回地址ra; r22: fp; r3: sp

r4: a0; r5: a1; r12: t0; r0: 0

一个简单的过程调用例子

```
int add ( int x, int y ) {
    return x+y;
}
int  caller ( ) {
    int  t1 = 125;
      int          t2 = 80;
    int  sum = add (t1, t2);
    return sum;
}
```

add过程中未用到任何保存寄存器，也没有非静态局部变量，且add不再调用其他过程，即是叶子过程，因而无须保存返回地址，因此可以没有栈帧，此时，对应反汇编代码如下：

```
add.w     $r4, $r4, $r5
jirl        $r0, $r1, 0
```

add过程比较简单，经GCC编译（-O1）并链接而生成的可执行文件被反汇编后的对应代码如下：

```
准备阶段
addi.w    $r3, $r3, -16(0xff0)   #R[sp]←R[sp]-16，生成新栈帧
st.w      $r22, $r3, 12(0xc)     #M[R[sp]+12]←R[fp]，保存fp的旧值
addi.w    $r22, $r3, 16(0x10)    #R[fp]←R[sp]+16，生成新的fp帧指针
传返回值
add.w     $r4, $r4, $r5          #R[r4]←x+y，返回结果存入a0
结束阶段
ld.w      $r22, $r3, 12(0xc)     #恢复帧指针fp的旧值
addi.w    $r3, $r3, 16(0x10)     #释放栈帧空间
jirl      $r0, $r1, 0            #返回caller调用过程
```

过程（函数）的结构

- 一个C过程的大致结构如下：
  - 准备阶段
    - 生成栈帧：addi 指令(sp加负数）
    - 形成帧底：st 指令（返回地址、fp等）存入栈帧底部
    - 保存现场（如果有被调用者保存寄存器）：st 指令
  - 过程（函数）体
    - 分配局部变量空间，并赋值
    - 具体处理逻辑，如果遇到函数调用时
      - 准备参数：将实参送参数寄存器
      - BL指令：保存返回地址并转被调用函数
    - 在a0（r4）中准备返回参数
  - 结束阶段
    - 退栈：addi指令(sp加正数）
    - 返回：jirl指令（根据r1中的返回地址）

过程调用参数传递举例

程序一

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf ("a=%d\tb=%d\n", a, b);
   swap (&a, &b);
   printf ("a=%d\tb=%d\n", a, b);
}
swap (int *x, int *y )
{
     int t=*x;
     *x=*y;
     *y=t;
}
```

按地址传递参数

执行结果？为什么？

程序一的输出：

a=15　　b=22

a=22　　b=15

程序二

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf ("a=%d\tb=%d\n", a, b);
   swap (a, b);
   printf ("a=%d\tb=%d\n", a, b);
}
swap (int x, int y )
{
     int t=x;
     x=y;
     y=t;
}
```

按值传递参数

程序二的输出：

a=15　　b=22

a=15　　b=22

过程调用参数传递举例

程序一汇编代码片段：

```
main:
    ……
  addi.w   $r5, $r22, -24(0xfe8)
  addi.w   $r4, $r22, -20(0xfec)
  bl       -152(0xffffff68) # 10674 <swap>
    ……
  jirl     $r0, $r1, 0
```

程序二汇编代码片段：

```
main:
    ……
  ld.w     $r5, $r22, -24(0xfe8)
  ld.w     $r4, $r22, -20(0xfec)
  bl       -120(0xffffff88) # 10674 <swap>
    ……
  jirl     $r0, $r1, 0
```

a和b分别存放在main栈帧的R[r22]-20和R[r22]-24处

传参时，程序一用addi.w指令，程序二用的是ld.w指令

故程序一传递的是a和b的地址，程序二传递的是a和b的内容

过程调用参数传递举例

程序一汇编代码片段：

```
main:
        ......
swap:
# 以下是准备阶段
addi.w      $r3, $r3, -48(0xfd0)
st.w        $r22, $r3, 44(0x2c)
addi.w      $r22, $r3, 48(0x30)
st.w        $r4, $r22, -36(0xfdc)
st.w        $r5, $r22, -40(0xfd8)
# 以下是过程体
ld.w        $r12, $r22, -36(0xfdc)
ld.w        $r12, $r12, 0
st.w        $r12, $r22, -20(0xfec)
ld.w        $r12, $r22, -40(0xfd8)
ld.w        $r13, $r12, 0
ld.w        $r12, $r22, -36(0xfdc)
st.w        $r13, $r12, 0
ld.w        $r12, $r22, -40(0xfd8)
ld.w        $r13, $r22, -20(0xfec)
st.w        $r13, $r12, 0
# 以下是结束阶段
ld.w        $r22, $r3, 44(0x2c)
addi.w      $r3, $r3, 48(0x30)
jirl        $r0, $r1, 0
```

程序二汇编代码片段：

```
main:
        ......
swap:
# 以下是准备阶段
addi.w      $r3, $r3, -48(0xfd0)
st.w        $r22, $r3, 44(0x2c)
addi.w      $r22, $r3, 48(0x30)
st.w        $r4, $r22, -36(0xfdc)
st.w        $r5,$r22,-40(0xfd8)
# 以下是过程体
ld.w        $r12, $r22, -36(0xfdc)
st.w        $r12, $r22, -20(0xfec)
ld.w        $r12, $r22, -40(0xfd8)
st.w        $r12, $r22, -36(0xfdc)
ld.w        $r12, $r22, -20(0xfec)
st.w        $r12, $r22, -40(0xfd8)
# 以下是结束阶段
ld.w        $r22, $r3, 44(0x2c)
addi.w      $r3, $r3, 48(0x30)
jirl        $r0, $r1, 0
```

准备阶段都将r4和r5中的参数存入了栈帧

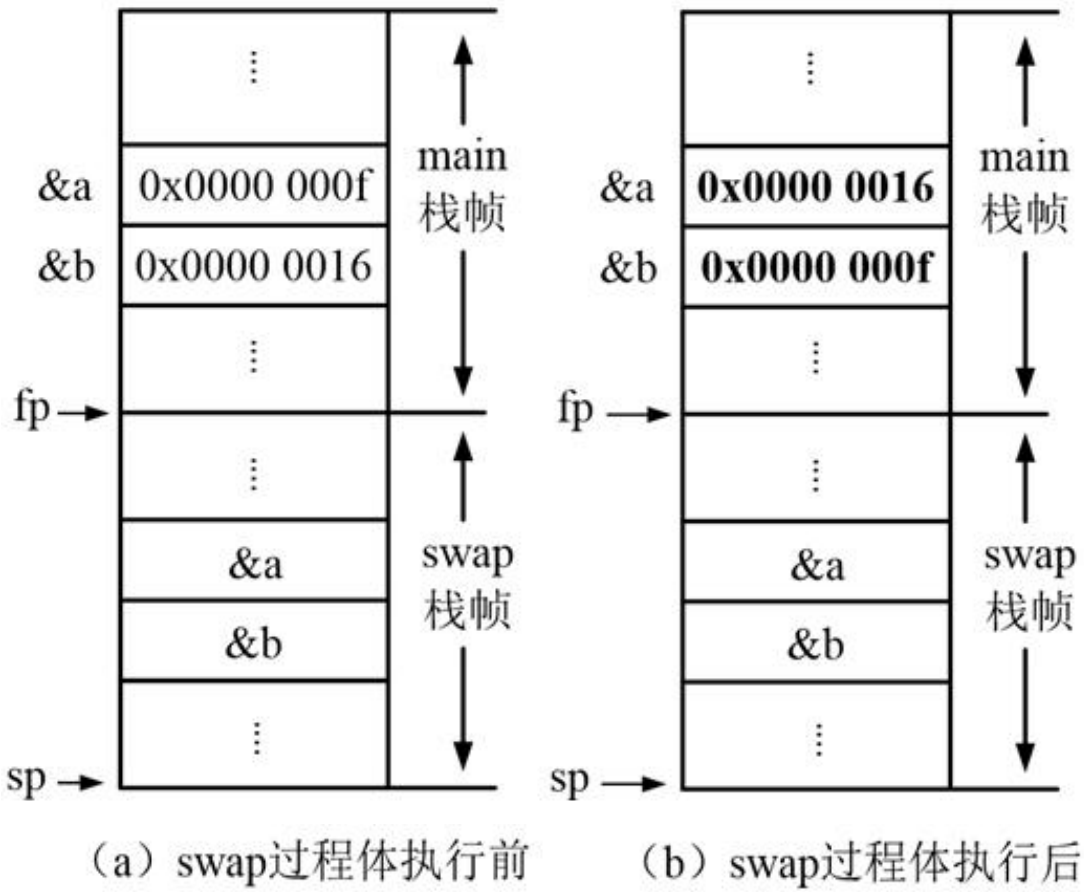
程序一的参数x和y是指针型变量，相当于间接寻址，先取出地址，再根据地址存取x和y，因而改变了main栈帧中局部变量a和b所在位置的内容

程序二的形参x和y是基本类型，直接存取x和y的内容，改变的是swap()入口参数x和y所在位置的值。

过程调用参数传递举例

程序一

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf ("a=%d\tb=%d\n", a, b);
   swap (&a, &b);
   printf ("a=%d\tb=%d\n", a, b);
}
swap (int *x, int *y )
{
     int t=*x;
     *x=*y;
     *y=t;
}
```

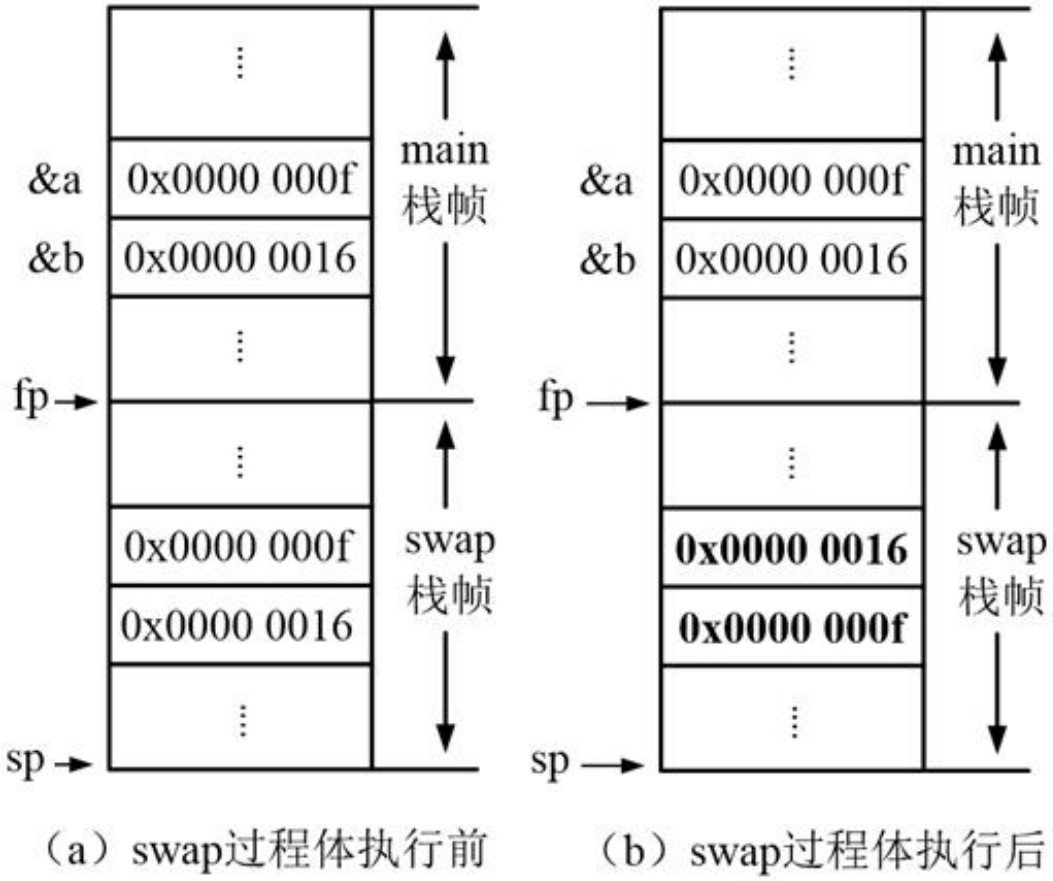


程序一的swap入口参数x和y是指针型变量，相当于间接寻址，先取出地址，再根据地址存取x和y，因而改变了main栈帧中局部变量a和b所在位置的内容

过程调用参数传递举例

程序二

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf ("a=%d\tb=%d\n", a, b);
   swap (a, b);
   printf ("a=%d\tb=%d\n", a, b);
}
swap (int x, int y )
{
      int t=x;
      x=y;
      y=t;
}
```



（a）swap过程体执行前　　（b）swap过程体执行后

程序二的swap入口参数x和y是基本类型，直接存取x和y的内容，改变的是swap()入口参数x和y所在位置的值。

过程调用举例

```
1   void test ( int x, int *ptr )
2   {
3         if  ( x>0 && *ptr>0 )
4         *ptr+=x;
5   }
6                  100    200
7   void caller (int a, int y )
8   {
9           int x = a>0 ? a : a+100;
10          test (x, &y);
11  }
```

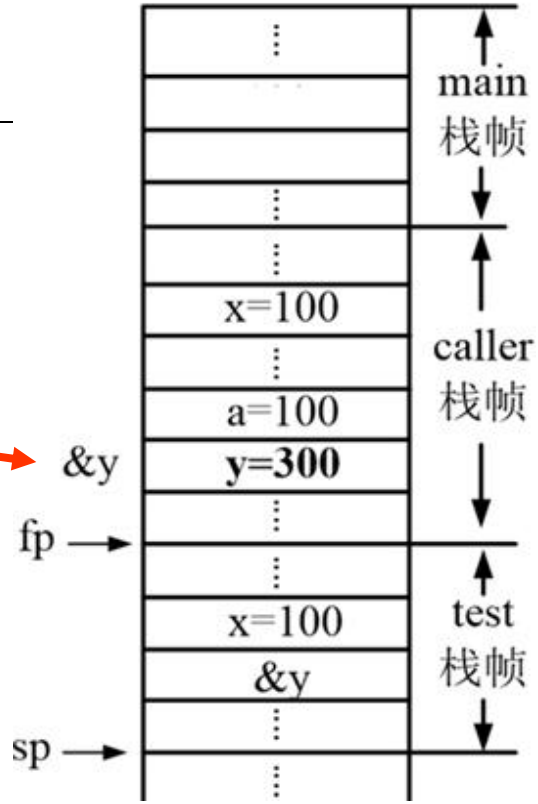
test

caller

main

则函数返回400

若return x+y;



调用caller的过程为P，P中给出形参a和y的实参分别是100和200，画出相应栈帧中的状态，并回答下列问题。

（1）test的形参是按值传递还是按地址传递？test的形参ptr对应的实参是一个什么类型的值？ 前者按值、后者按地址。一定是一个地址

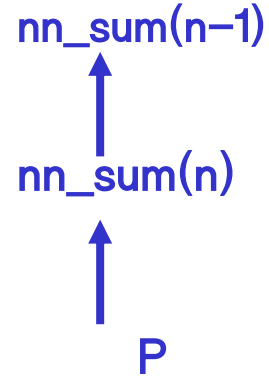
（2）test中被改变的*ptr的结果如何返回给它的调用过程caller?

第10行执行后，caller栈帧中y处200变成300，test退帧后，caller中通过y引用300

（3）caller中被改变的y的结果能否返回给过程P？为什么？

第11行执行后caller退帧并返回到main，因main中无变量与之对应，故无法引用300

```
int  nn_sum ( int n) {
    int result;
    if  (n<=0 )
        result=0;
    else
        result=n+nn_sum(n-1);
    return  result;
}
```




```
nn_sum:
    blt     $r0, $r4, 12(0xc)  #.L2
    move    $r4, $r0
    jirl    $r0, $r1, 0
.L2
    addi.w  $r3, $r3, -16(0xff0)
    st.w    $r1, $r3, 12(0xc)
    st.w    $r22, $r3, 8(0x8)
    st.w    $r23, $r3, 4(0x4)
    addi.w  $r22, $r3, 16(0x10)
    move    $r23, $r4
    addi.w  $r4, $r4, -1(0xfff)
    bl      -40(0xffffd8)  #nn_sum
    add.w   $r4, $r4, $r23
    ld.w    $r1, $r3, 12(0xc)
    ld.w    $r22, $r3, 8(0x8)
    ld.w    $r23, $r3, 4(0x4)
    addi.w  $r3, $r3, 16(0x10)
    jirl    $r0, $r1, 0
```

if（n>0）转L2

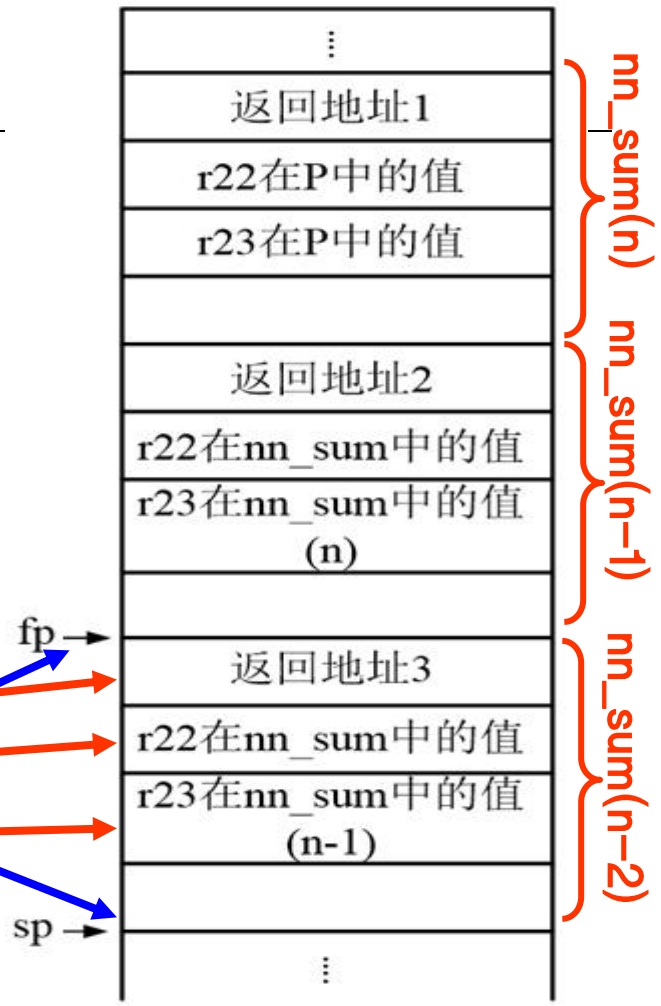
R[a0]←0

$r1中返回地址不改变

R[s0]←n

R[a0]←n-1

R[a0]←0+1+2+…+(n-1)+n

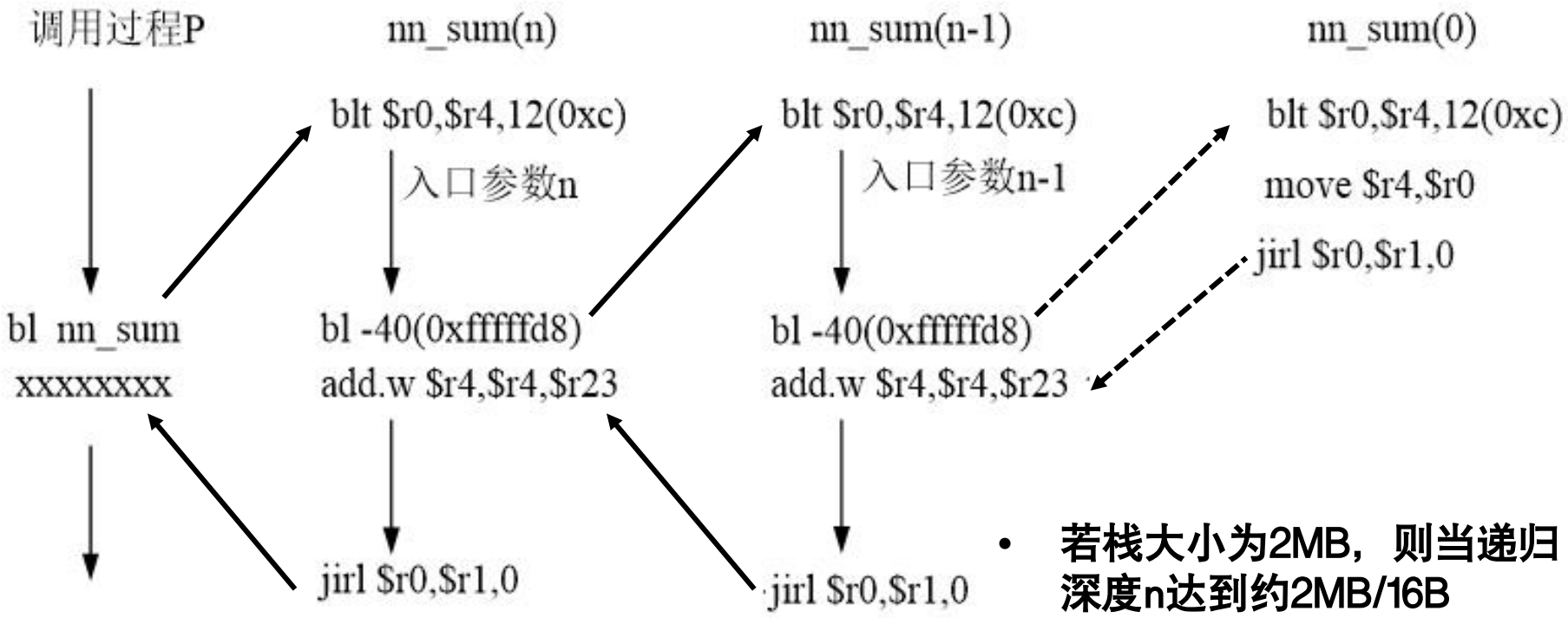


操作系统为程序分配的栈会有默认的大小限制，如2MB、8MB

每次递归调用都会增加一个栈帧（该例为16B），所以空间开销很大。当n很大时会发生**栈溢出**！

过程调用的机器级表示

- 递归函数nn_sum的执行流程



- 若栈大小为2MB，则当递归深度n达到约2MB/16B $=2^{17}=131\ 072$时，发生栈溢出！

为支持过程调用，每个过程包含准备阶段和结束阶段。因而每增加一次过程调用，就要增加许多条包含在准备阶段和结束阶段的额外指令，它们对程序性能影响很大，应尽量避免不必要的过程调用，特别是递归调用。

x86+Windows中的存储映像

```
#include   ….
int  g1=0,  g2=0,  g3=0;
int  main()
{
    static  int  s1=0,  s2=0,  s3=0;
    int  v1=0,  v2=0,  v3=0;
        printf("0x%08x\n",&v1);
    printf("0x%08x\n",&v2);
    printf("0x%08x\n\n",&v3);
    printf("0x%08x\n",&g1);
    printf("0x%08x\n",&g2);
    printf("0x%08x\n\n",&g3);
    printf("0x%08x\n",&s1);
        printf("0x%08x\n",&s2);
    printf("0x%08x\n\n",&s3);
    return  0;
}
```

说明了什么？

注意：每个存储区地址的特征！

执行结果如下：

0x0012ff78
0x0012ff7c
0x0012ff80

局部变量存放在另一个存储区：栈区

0x004068d0
0x004068d4
0x004068d8
0x004068dc
0x004068e0
0x004068e4

全局变量和静态变量连续存放在同一个存储区：可读写数据区

x86+Windows/Linux中的存储映像

说明了什么？注意：每个存储区地址的特征！

非静态局部变量
不一定按顺序分配

```
1  #include <stdio.h>
2  int g1 = 0, g2 = 0, g3 = 0;
3  int main()
4  {
5      static int s1 = 0, s2 = 0, s3 = 0;
6      int v1 = 0, v2 = 0, v3 = 0;
7      printf("0x%p\n", &v1);
8      printf("0x%p\n", &v2);
9      printf("0x%p\n\n", &v3);
10     printf("0x%p\n", &g1);
11     printf("0x%p\n", &g2);
12     printf("0x%p\n\n", &g3);
13     printf("0x%p\n", &s1);
14     printf("0x%p\n", &s2);
15     printf("0x%p\n\n", &s3);
16     return 0;
17 }
```

Windows	Linux
0x0013FD28	0x0xff8a5864
0x0013FD24	0x0xff8a5868
0x0013FD20	0x0xff8a586c
0x00C43384	0x0x804a024
0x00C43380	0x0x804a028
0x00C4337C	0x0x804a02c
0x00C43374	0x0x804a030
0x00C43378	0x0x804a034
0x00C43388	0x0x804a038

局部变量存放在另一个存储区：**栈区**

全局变量和静态变量连续存放在同一个存储区：**可读写数据区**

全局、静态变量
不一定按顺序分配

x86+Windows中的分配顺序

```
#include   ……
int  main()
{
    int  a;
    char  b;
    int  c;
    printf( “a:  0x%08x\n",&a);
    printf( “b:  0x%08x\n",&b);
    printf( “c:  0x%08x\n",&c);
    return  0;
}
```

局部变量（如a、b、c）
不一定按顺序分配！

用VC编译后的执行结果：

a: 0x0012ff7c

b: 0x0012ff7b

c: 0x0012ff80

顺序：b(1B)–a(4B)–c(4B)

用Dev–C++编译后的执行结果：

a: 0x0022ff7c

b: 0x0022ff7b

c: 0x0022ff74

顺序：c(4B)–隔3B–b(1B)–a(4B)

用lcc编译后的执行结果：

a: 0x0012ff6c

b: 0x0012ff6b

c: 0x0012ff64

顺序：同上（大地址–>小地址）

x86+Windows/Linux中的存储映像

说明了什么？注意：每个存储区地址的特征！

```
1  #include<stdio.h>
2  int func(int param1, int param2, int param3)
3  {
4      int var1 = param1;
5      int var2 = param2;
6      int var3 = param3;
7      printf("0x%p\n", &param1);
8      printf("0x%p\n", &param2);
9      printf("0x%p\n\n", &param3);
10     printf("0x%p\n", &var1);
11     printf("0x%p\n", &var2);
12     printf("0x%p\n\n", &var3);
13     return 0;
14 }
15 int main()
16 {
17     func(1, 2, 3);
18     return 0;
19 }
```

局部变量和参数都存放在：栈区

Linux

```
0x0xffff2b50
0x0xffff2b54
0x0xffff2b58

0x0xffff2b34
0x0xffff2b38
0x0xffff2b3c
```

参数的地址总比局部变量的地址大！

因为栈的生长方向：高地址→ 低地址

Linux总是最右边参数的地址最大，因为参数入栈顺序为：右→左

x86+Windows中的存储映像

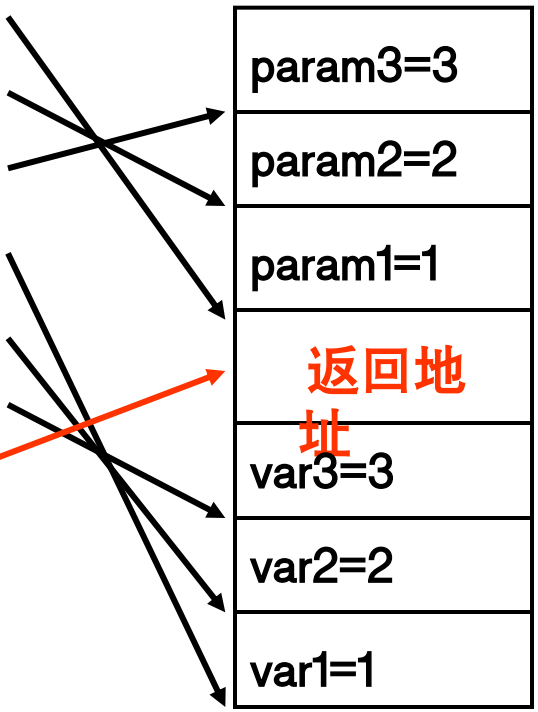
```
#include  ……
void __stdcall func(int param1,int param2,int param3)
{
    int var1=param1;
    int var2=param2;
    int var3=param3;
    printf( “%08x\n” ,&param1);
    printf("%08x\n", &param2);
    printf("%08x\n\n", &param3);
    printf("%08x\n",&var1);
    printf("%08x\n",&var2);
    printf("%08x\n\n",&var3);
    return;
}
int main()
{
    func(1,2,3);
    return 0;
}
```

说明了什么？

Windows中栈区也是高地址向低地址生长！

执行结果如下：

0x0012ff78
0x0012ff7c
0x0012ff80
0x0012ff68
0x0012ff6c
0x0012ff70



猜猜这里是什么？

这里与Linux的差别是什么？ EBP未压栈！

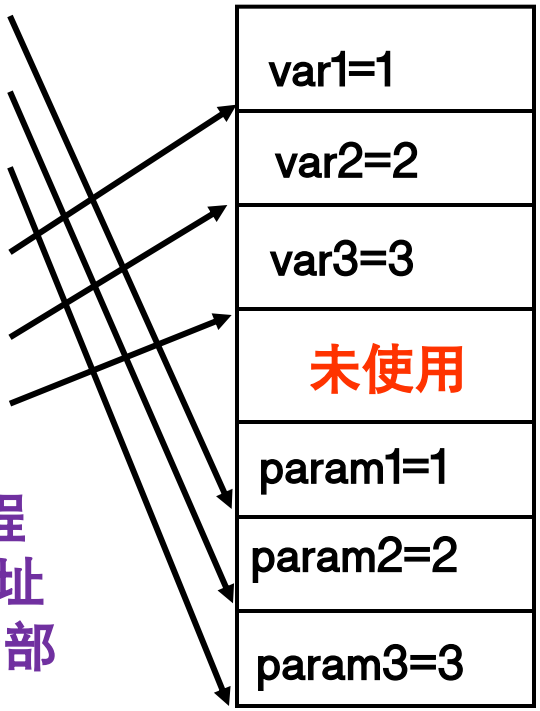
LA32+Linux中的存储映像

```
#include   <stdio.h>
void  func(int  param1,int  param2,int  param3)   {
          int  var1=param1;
    int  var2=param2;
    int  var3=param3;
    printf("%p\n",&param1);
    printf("%p\n",  &param2);
    printf("%p\n\n",  &param3);
    printf("%p\n",&var1);
    printf("%p\n",&var2);
    printf("%p\n\n",&var3);
    return;
}
int  main()  {
      func(1,2,3);
    return  0;
}
```

说明了什么？

执行结果如下：

0x4080031c
0x40800318
0x40800314
0x4080032c
0x40800328
0x40800324



入口参数在参数寄存器中，若过程体中调用其他过程，则需将返回地址和入口参数入栈保存，LA中通常局部变量在高地址，入口参数在低地址

变量的存储分配

非静态局部变量占用的空间分配在本过程的栈帧中

全局、静态变量在可读可写数据区分配

- C标准中，没有规定必须按顺序分配，不同的编译器有不同的处理方式。
- C标准明确指出，对不同变量的地址进行除==和!=之外的关系运算，都属未定义行为（undefined behavior）

  如，语句“if（&var1 < &var2）{...};”属于未定义行为
- 编译优化的情况下，会把属于简单数据类型变量分配在通用寄存器中
- 对于复杂数据类型变量，如数组、结构和联合等数据类型变量，一定会分配在存储器中

LA入口参数传递和分配

```
long  caller(long  x){
        long  a=1000;
        long  b=test(&a,2000);
        return  x*32+b;
}
```

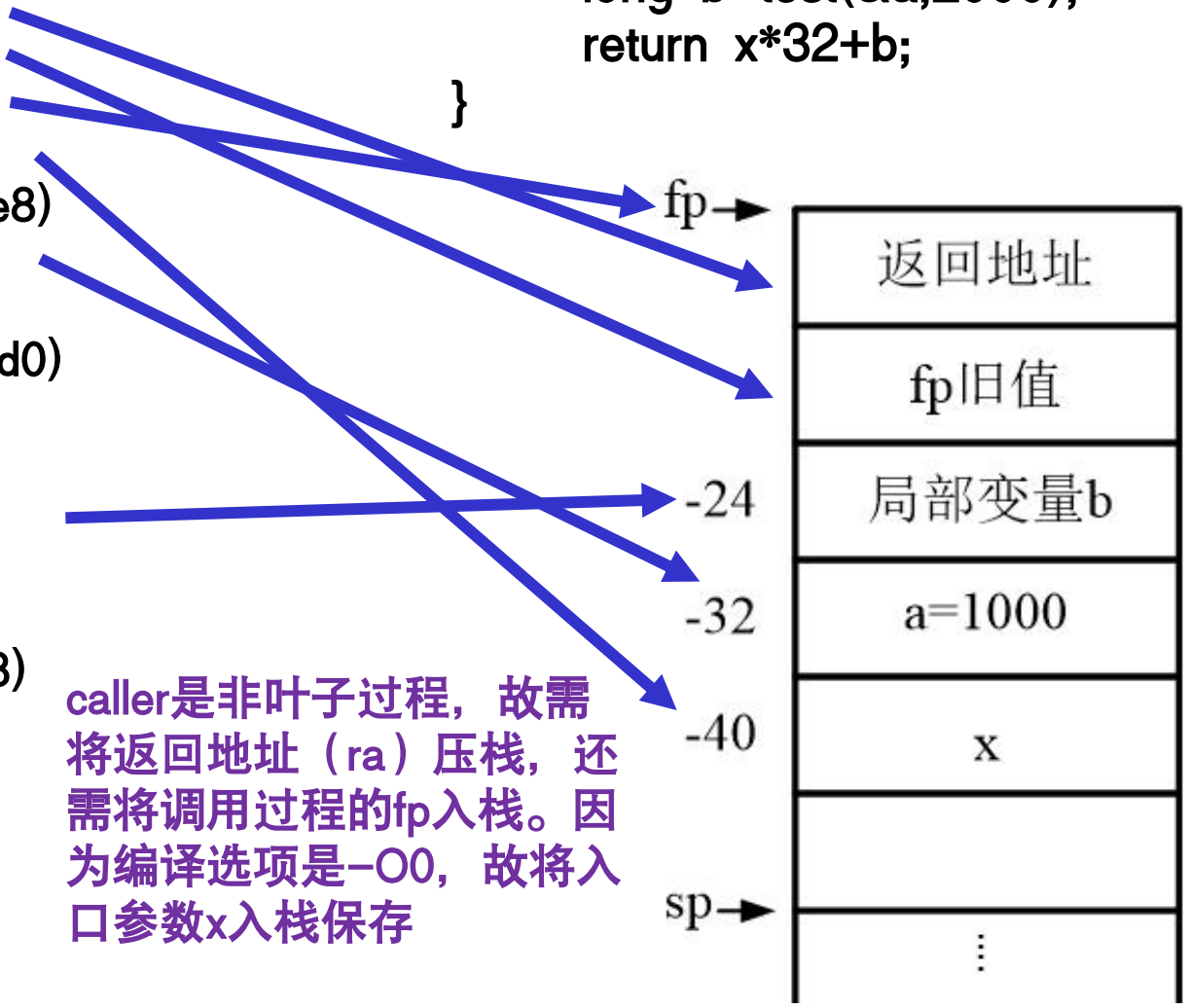
```
addi.d    $sp,$sp,-48(0xfd0)    #R[sp]←R[sp]-48,生成栈帧
st.d      $ra,$sp,40(0x28)        #M[R[sp]+40]←R[ra]，返回地址入栈
st.d      $fp,$sp,32(0x20)        #M[R[sp]+32]←R[fp]，fp的旧值入栈
addi.d    $fp,$sp,48(0x30)        #R[fp]←R[sp]+48,生成栈帧指针
st.d      $a0,$fp,-40(0xfd8)     #入口参数x入栈
addi.w    $t0,$zero,1000(0x3e8)       #R[t0]←1000
st.d      $t0,$fp,-32(0xfe0)      #M[R[fp]-32]←1000，对变量a赋值
addi.d    $t0,$fp,-32(0xfe0)      #R[t0]←R[fp]-32，获取变量a的地址
addi.w    $a1,$zero,2000(0x7d0)       #R[a1]←2000，第二个参数送a1寄存器
move       $a0,$t0                    #R[a0]←R[t0]，第一个参数送a0寄存器
bl        0  #  28                    #调用test（R[ra]←返回地址，PC←test）
st.d      $a0,$fp,-24(0xfe8)      #M[R[fp]-24]←R[a0]，test返回结果存入b处
ld.d      $t0,$fp,-40(0xfd8)      #R[t0]←M[R[fp]-40]，取出x
slli.d    $t1,$t0,0x5                 #R[t1]←R[t0]<<,5计算x*32
ld.d        $t0,$fp,-24(0xfe8)     #R[t0]←M[R[fp]-24],取出b
add.d     $t0,$t1,$t0                 #R[t0]←R[t0]+R[t1]，计算x*32+b
move       $a0,$t0                    #R[a0]←R[t0]，返回结果送a0
ld.d        $ra,$sp,40(0x28)          #R[ra]←M[R[sp]+40]，返回地址出栈
ld.d      $fp,$sp,32(0x20)        #R[fp]←M[R[sp]+32]，fp旧值出栈
addi.d    $sp,$sp,48(0x30)        #R[sp]←R[sp]+48,释放栈帧
jirl      $zero,$ra,0
```

LA中入口参数的传递和分配

ABI规定栈帧按16B对齐，故大小为48B

```
addi.d    $sp,$sp,-48(0xfd0)
st.d      $ra,$sp,40(0x28)
st.d      $fp,$sp,32(0x20)
addi.d    $fp,$sp,48(0x30)
st.d      $a0,$fp,-40(0xfd8)
addi.w    $t0,$zero,1000(0x3e8)
st.d      $t0,$fp,-32(0xfe0)
addi.d    $t0,$fp,-32(0xfe0)
addi.w    $a1,$zero,2000(0x7d0)
move       $a0,$t0
bl        0  #  28
st.d      $a0,$fp,-24(0xfe8)
ld.d      $t0,$fp,-40(0xfd8)
slli.d    $t1,$t0,0x5
ld.d        $t0,$fp,-24(0xfe8)
add.d     $t0,$t1,$t0
move       $a0,$t0
ld.d        $ra,$sp,40(0x28)
ld.d      $fp,$sp,32(0x20)
addi.d    $sp,$sp,48(0x30)
jirl      $zero,$ra,0
```

```
long  caller(long  x){
        long  a=1000;
        long  b=test(&a,2000);
        return  x*32+b;
}
```



caller是非叶子过程，故需将返回地址（ra）压栈，还需将调用过程的fp入栈。因为编译选项是-O0，故将入口参数x入栈保存

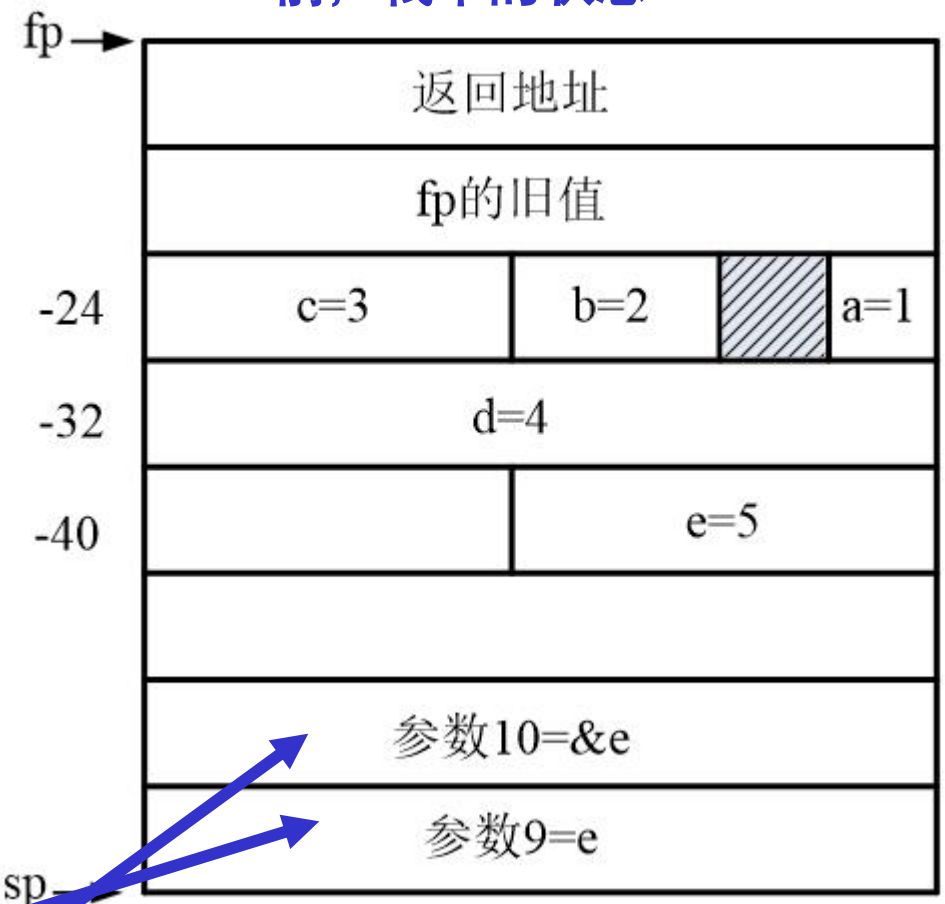
LA64过程调用举例

```
void test(char a, char *ap,
                  short b, short *bp,
                  int c, int *cp,
                  long d, long *dp
                  int e, int *ep) {
    *ap+=a;    *bp+=b;
    *cp+=c;    *dp+=d;    *ep+=e;
}

long caller ( ) {
      char a=1;
      short b=2;
      int c=3;
      long d=4;
      int e=5;
      test(a,&a,b,&b,c,&c,d,&d,e,&e);
      return   a*b+c*d+e;
}
```

前8个参数在a0~a7中，后两参数在栈中

执行到caller的bl指令前，栈中的状态



局部变量在栈帧中按自然边界对齐

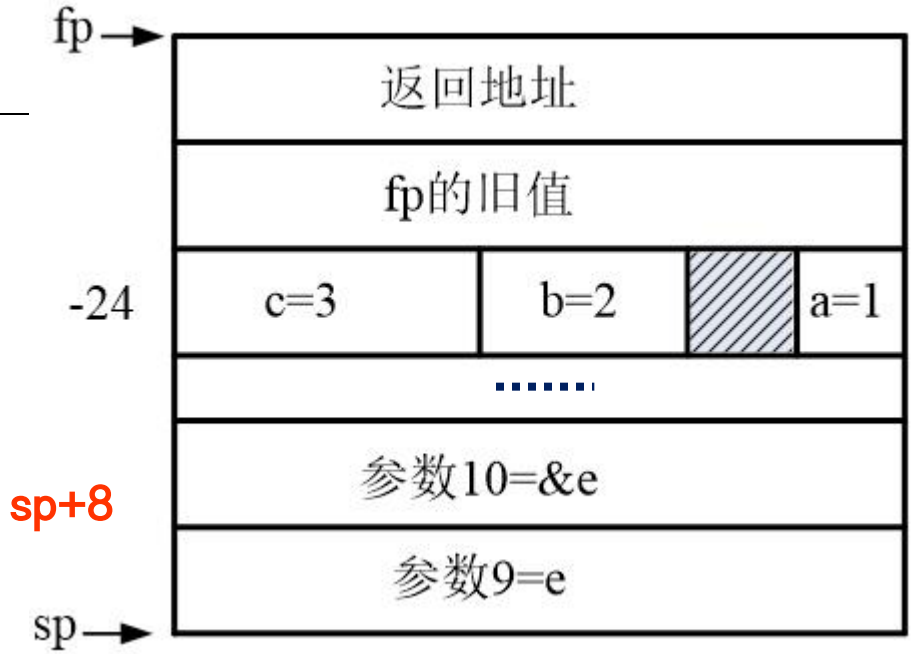
LA32、LA64中参数各按4B和8B对齐

在LA64中，若在栈中传递的参数不是long型或指针型，也都应分配8B。例如，如参数e为int型，在栈中也占8字节。

LA64过程调用举例

函数test()对应的LA64汇编代码

```
ld.bu    $t0, $a1, 0
add.w    $a0, $t0, $a0      *ap+=a;
st.b     $a0, $a1, 0
ld.hu    $t0, $a3, 0
add.w    $a2, $t0, $a2      *bp+=b;
st.h     $a2, $a3, 0
ld.w     $t0, $a5, 0
add.w    $a4, $t0, $a4      *cp+=c;
st.w     $a4, $a5, 0
ld.d     $t0, $a7, 0
add.d    $t0, $t0, $a6      *dp+=d;
st.d     $t0, $a7, 0
ld       $t1, $sp, 8(0x8)   R[t1]←&e
ld.w     $t0, $t1, 0
ld.w     $t2, $sp, 0        R[t2]←e
add.w    $t0, $t0, $t2      *ep+=e;
st.w     $t0, $t1, 0
jirl     $zero, $ra, 0
```



$a0、$a1、$a2、$a3、$a4、$a5

```
void test(char a, char *ap,
                  short b, short
    *bp,
                  int c, int *cp,
                  long d, long
    *dp
                  int e, int *ep)
{
*ap+=a; *bp+=b;
*cp+=c;  *dp+=d;
  *ep+=e;
}
```

LA32、LA64过程调用举例

未定义行为（undefined behavior）语句

变量a的机器数为4024 0000 0000 0000H

LA32中寄存器位宽为32，double型数据保存偶-奇对寄存器（如a0-a1）中，故参数a在r6-r7(a2-a3)寄存器中。即a2中为4024 0000H，a3中为0000 0000H

例：以下是一段C语言代码：

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

r4(a0)、r5(a1)

LA32的反汇编代码：

```
1  ld.w          $r6, $r22, -24(0xfe8)
2  ld.w          $r7, $r22, -20(0xfec)
3  pcaddu12i     $r4, 101(0x65)
4  addi.w        $r4, $r4, -1980(0x844)
5  bl            47776(0xbaa0) # <_IO_printf>
```

在LA32上运行时，打印结果为a=1082131396

在LA64上运行时，打印出来的a=0

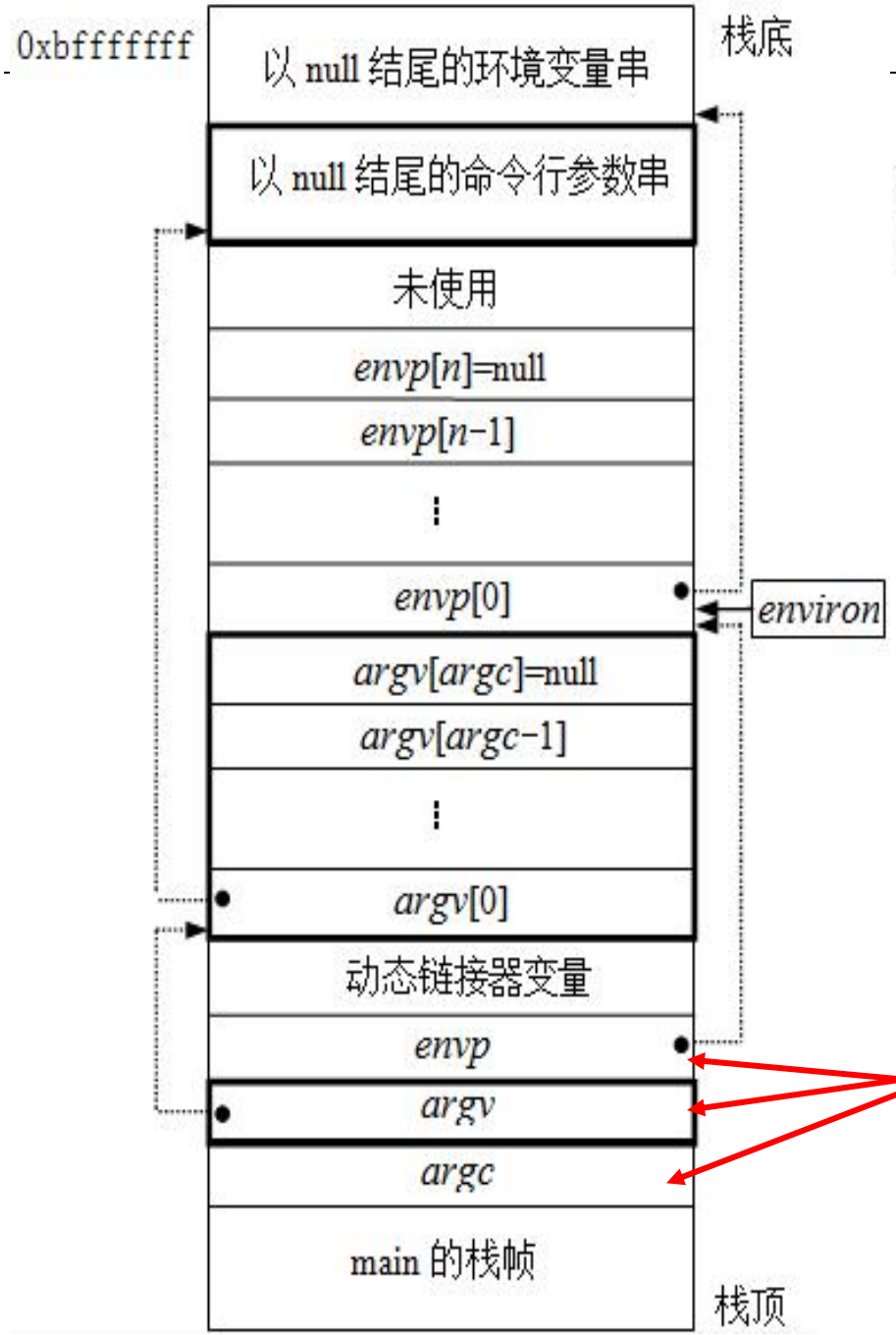
按 int 型解释，按double型传参

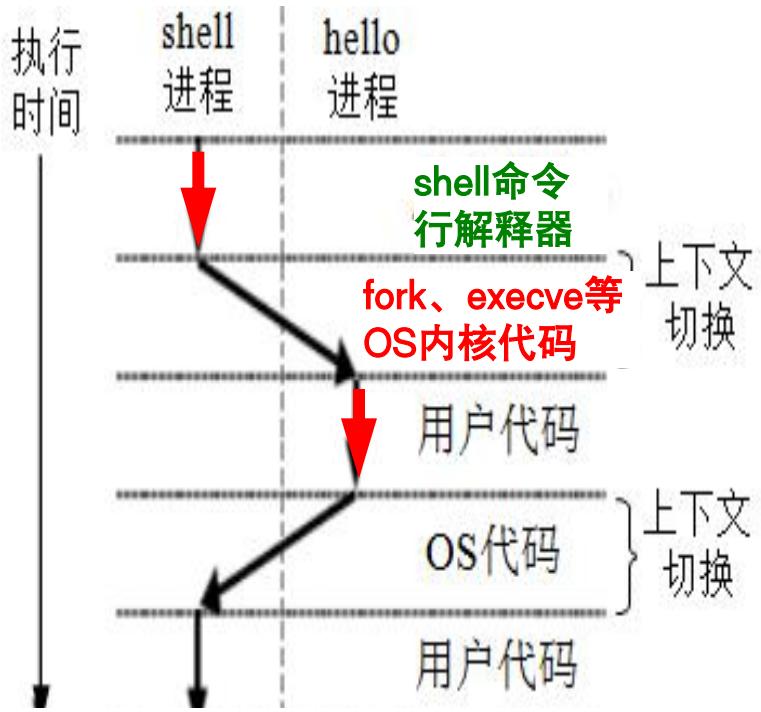
为什么？

未对r5(a1)赋值，为何输出1082131396?

实际输出的是栈区某地址（0x408003c4）对应的十进制数值

程序加载和运行





当Linux系统开始执行main()函数时，在虚拟地址空间的用户栈中的结构如左图所示

```
int main(int    argc,          r4 (a0)
         char *argv[],         r5 (a1)
         char *envp[]);        r6 (a2)
```

r5中内容或固定，或不确定（栈随机化机制时），但内容应是栈区某地址

LA32、LA64过程调用举例

未定义行为（undefined behavior）语句

例：以下是一段C语言代码：

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

r4(a0)、r5(a1)中低32位

在LA32上运行时，打印结果为a=1082131396

在LA64上运行时，打印出来的a=0

为什么？

变量a的机器数为4024 0000 0000 0000H

LA64中寄存器位宽为64，故参数a在r5(a1)寄存器中，即a1中为4024 0000 0000 0000H

LA64的反汇编代码：

```
1 ld.d          $a1, $fp, -24(0xfe8)
2 pcaddu12i $a0, 87(0x57)
3 addi.d        $a0, $a0, -1136(0xb90)
4 bl    24716(0x608c) # <_IO_printf>
```

按 int 型解释，按double型传参

r5(a1)中低32位为0，故打印结果为a=0

补充说明：LA ABI规范规定函数中浮点入口参数用浮点寄存器传参，但printf()是可变参数函数，多用通用寄存器传参，是特殊的传参规定

程序的机器级表示

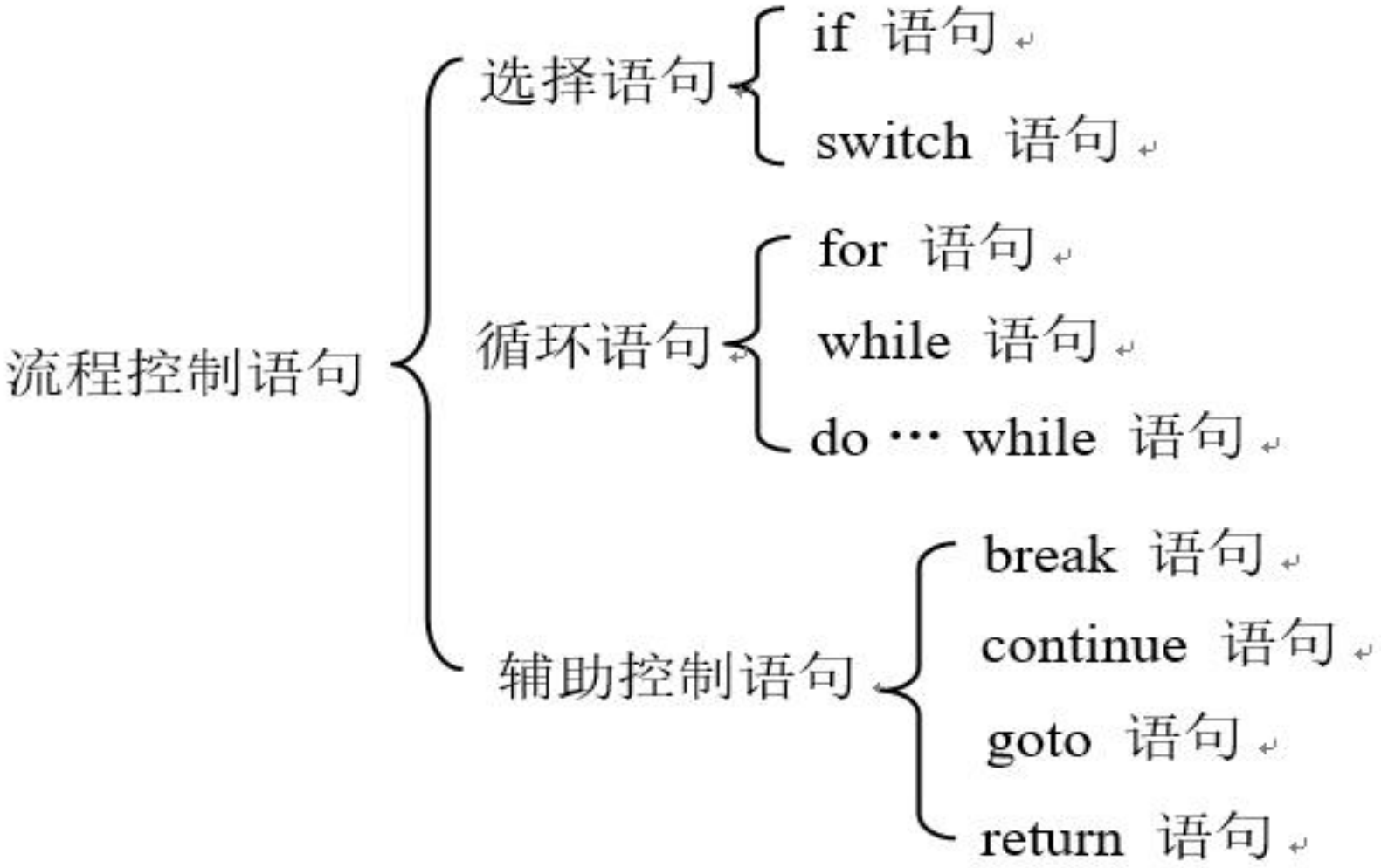
- 分以下四个部分介绍
  - 第一讲：　过程调用的机器级表示
    - 过程调用约定、变量的作用域和生存期
    - 按值传参和按地址传参、递归过程调用
    - 非静态局部变量的存储分配
    - 入口参数的传递和分配
  - 第二讲：　流程控制语句的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

程序的机器级表示

- C语言中的流程控制语句



选择结构的机器级表示

- if ~ else语句的机器级表示

```
if (cond_expr)
        then_statement
else
        else_statement
```

```
        c=cond_expr;
        if (!c)
                goto false_label;
        then_statement
        goto done;
false_label:
        else_statement
done:
```

Bxx 指令

B 指令

```
        c=cond_expr;
        if (c)
                goto true_label;
        else_statement
        goto done;
true_label:
        then_statement
done:
```

If-else语句举例

```
int   get_cont( int *p1, int *p2 )
{
        if  ( p1 > p2 )
                return *p2;
        else
                return *p1;
}
```

p1和p2对应实参分别在寄存器$a0和$a1中，返回参数在$a0中

为何这里是” bge”指令?

```
    bge         $a1,$a0,12(0xc)#  .L1  #若p2>=p1，则转.L1处执行
    ldptr.w   $a0,$a1,0                    #R[a0]←M[R[a1]]，即R[a0]=*p2
    jirl        $zero,$ra,0                #返回调用过程
.L1:
    ldptr.w    $a0,$a0,0                   #R[a0]←M[R[a0]]，即R[a0]=*p1
    jirl        $zero,$ra,0                #返回调用过程
```

If-else语句举例

```
void test(int x,int *ptr){
    if (x>0 && *ptr>0)
        *ptr+=x;
}

int caller(int a, int y) {
    int x = a>0 ? a :
a+100;
    test(a, &y);
    return x+y;
}
```

caller的入口参数a和y分别在r4和r5中，C语句“x = a>0 ? a:a+100;”对应的汇编代码如下。

```
    blt   $r0,$r4,12(0xc)  #.L1      #若a>0,则转.L1
    addi.w  $r12,$r4,100(0x64)  #x=a+100
    b       8(0x8)  #.L2             #转.L2处执行
.L1
     move $r12,  $r4            #x=a
.L2
```

test()的入口参数x和ptr分别在r4和r5，LA64汇编代码：

```
    bge     $r0,$r4,20(0x14)  #  .L1    #若x<=0，则转.L1处执行
    ld.w    $r12,$r5,0                   #R[r12]←*ptr
    bge     $r0,$r12,12(0xc)  #  .L1    #若*ptr  <=0，则转.L1处执行
    add.w   $r4,$r12,$r4                 #实现*ptr+=x的功能
    stptr.w  $r4,$r5,0                   #保存*ptr+=x的结果
.L1
     jirl          $r0,$r1,0             #返回调用过程
```

LA32中switch-case语句举例

```
int sw_test(int a, int b, int c)
{
    int result;
    switch(a) {
    case 15:
        c=b&0x0f;
    case 10:
        result=c+50;
        break;
    case 12:
    case 17:
        result=b+50;
        break;
    case 14:
        result=b
        break;
    default:
        result=a;
    }
    return result;
}
```

a在10和17之间

```
  addi.w    $r12, r4, -10(0xff6)
  addi.w    $r13, r0, 7(0x7)
  bltu      $r13, r12, 52(0x34)
  slli.w    $r12, $r12, 0x2
  pcaddu12i     $r13, 101(0x65)
  addi.w    $r13, $r13, -1936(0x870)
  add.w     $r12, $r13, $r12
  ld.w      $r12, $r12, 0
  jirl      $r0, $r12, 0
.L1
  andi      $r6, $r5, 0xf
.L2
  addi.w    $r4, $r6, 50(0x32)
  jirl      $r0, $r1, 0
.L3
  addi.w    $r4, $r5, 50(0x32)
  jirl      $r0, $r1, 0
.L4
  move          $r4, $r5
.L5
  jirl      $r0, $r1, 0
```

R[r12]=a−10=i

if (a−10)>7 转 L5

R[r12]←i*4

转.L6+4*i 处的地址

r12中为跳转表表项

跳转表在目标文件的只读节中，按4字节边界对齐。

```
  .section      .rodata
  .align        2
.L6
  .word     .L2     10
  .word     .L5     11
  .word     .L3     12
  .word     .L5     13
  .word     .L4     14
  .word     .L1     15
  .word     .L5     16
  .word     .L3     17
```

a=

LA64中switch-case语句举例

C语言函数switch_test()的部分代码及其LA64部分汇编代码和跳转表如下

```
int sw_test(int x, int *ptr)
{
    switch(x) {
        ……
    default:
        ……
    }
    *ptr+=x;
}
```

```
addi.w      $t0,$a0,3(0x3)
addi.w      $t1,$zero,6(0x6)
bltu        $t1,$t0,52(0x34) # .L3
pcaddu12i   $t1,87(0x57)
addi.d      $t1,$t1,-1116(0xba4)
alsl.d      $t0,$t0,$t1,0x3
ldptr.d     $t0,$t0,0
jirl        $zero,$t0,0
```

```
    .section    .rodata
    .align 3
.L7:
    .dword      .L2
    .dword      .L3
    .dword      .L4
    .dword      .L5
    .dword      .L3
    .dword      .L5
    .dword      .L6
```

switch_test()函数的switch语句中共有几个case分支？case取值各是什么？各对应跳转表中哪个标号？

当x+3>6时为default情况，对应标号.L3。bltu指令按无符号整数比较，故仅在0<=x+3<=6（x在−3~3之间）才不满足bltu条件，从而需执行jirl指令通过跳转表进行指令跳转。

alsl.d指令实现R[t0]←R[t1]+(x+3)*8，即t0中为表项地址

ldptr.d指令用于将表项内容送t0.

switch语句中共有6个分支，对应的x取值分别−3（.L2）、−1（.L4）、0（.L5）、2（.L5）、3（.L6）和default（.L3）

循环结构的机器级表示

- do~while循环的机器级表示

```
do  loop_body_statement
      while (cond_expr);
```

```
loop:
        loop_body_statement
        c=cond_expr;
        if (c) goto loop;
```

红色处为条件转移指令！

不一定有无条件跳转指令

- for循环的机器级表示

```
for (begin_expr; cond_expr; update_expr)
        loop_body_statement
```

- while循环的机器级表示

```
while (cond_expr)
      loop_body_statement
```

```
        c=cond_expr;
        if (!c) goto done;
loop:
        loop_body_statement
        c=cond_expr;
        if (c) goto loop;
done:
```

```
        begin_expr;
        c=cond_expr;
        if (!c) goto done;
loop:
        loop_body_statement
        update_expr;
        c=cond_expr;
        if (c) goto loop;
done:
```

循环结构与递归的比较

递归函数nn_sum仅为说明原理，实际上可直接用公式，为说明循环的机器级表示，这里用循环实现。

```
int   nn_sum ( int n)
{
        int i;
                int result=0;
        for (i=1; i <=n; i++)
                result+=i;
        return result;
}
```

```
  addi.w $r13, $r0, 0(0x0)
  addi.w $r12, $r0, 1(0x1)
    b       .L1
.Loop
  add.w  $r13, $r12, $r13
  addi.w $r12, $r12, 1(0x1)
.L1
    bge     $r4, $r12, -8(0x3fff8)
    move   $r4, $r13
    jirl      $r0, $r1, 0
```

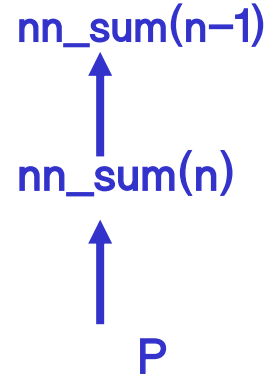
局部变量 i 和 result 各分配在r12和r13中。

通常复杂局部变量分配在栈中，而这里都是简单变量

SKIP

过程体中没用到被调用过程保存寄存器，其栈帧大小为0字节，而递归方式则占用了16n字节栈空间；前n次递归调用每次都要执行15条指令，递归情况下共多n次过程调用，而非递归情况下每次循环只要执行3条指令，因而递归方式比非递归方式大约多执行(15−3)n=12n条指令。由此可以看出，为了提高程序的性能，若能用非递归方式执行则最好用非递归方式。

```
int  nn_sum ( int n) {
    int result;
    if  (n<=0 )
        result=0;
    else
        result=n+nn_sum(n-1);
    return  result;
}
```




```
nn_sum:
    blt     $r0, $r4, 12(0xc)  #.L2
    move    $r4, $r0
    jirl    $r0, $r1, 0
.L2
    addi.w  $r3, $r3, -16(0xff0)
    st.w    $r1, $r3, 12(0xc)
    st.w    $r22, $r3, 8(0x8)
    st.w    $r23, $r3, 4(0x4)
    addi.w  $r22, $r3, 16(0x10)
    move    $r23, $r4
    addi.w  $r4, $r4, -1(0xfff)
    bl      -40(0xffffd8)  #nn_sum
    add.w   $r4, $r4, $r23
    ld.w    $r1, $r3, 12(0xc)
    ld.w    $r22, $r3, 8(0x8)
    ld.w    $r23, $r3, 4(0x4)
    addi.w  $r3, $r3, 16(0x10)
    jirl    $r0, $r1, 0
```

if（n>0）转L2

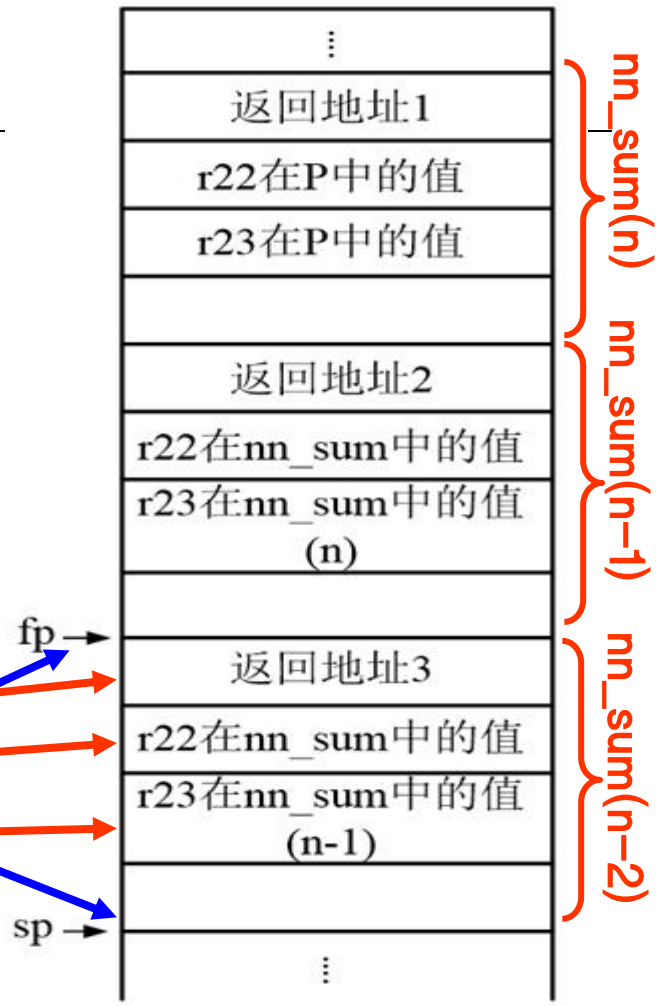
R[a0]←0

$r1中返回地址不改变

R[s0]←n

R[a0]←n-1

R[a0]←0+1+2+…+(n-1)+n



操作系统为程序分配的栈会有默认的大小限制，如2MB、8MB

每次递归调用都会增加一个栈帧（该例为16B），所以空间开销很大。当n很大时会发生**栈溢出**！

LA64逆向工程举例

```
int function_test( unsigned x)
{
    int result=0;
    int i;
    for ( ____①____ ; ____②____ ;
   ③_____ ) {
                ____④____
   ;
    }
    return result;
}
```

```
move    $t1, $a0
  move  $a0, $zero
  addi.w $t2, $zero, 11(0xb)
.L1
  slli.d    $a0, $a0, 0x1
  andi      $t0, $t1, 0xf
   or                $a0, $t0, $a0
   srai.w    $t1, $t1, 0x1
   addi.w $t2, $t2, –1(0xfff)
   bnez              $t2, –
20(0x7fffec) #.L1
   jirl      $zero, $ra, 0
```

该函数有几个参数？ 1个

处理结构是怎样的？ 循环结构！

① 处为i=11，② 处为i!=0，③ 处为i--。

入口参数 x 在a0中，返回参数 result 在a0中。slli实现将result左移1位，即“result<<1”；andi实现“x&0x0f”；or实现“result=(result<<1) | (x&0x0f)”，srai实现“x>>=1”。综上所述，④ 处的C语言语句是“result=(result<<1) | (x & 0x0f); x>>=1;”。

翻转课堂

翻转题目一:

```
int add(int *xp, int *yp) {
        return *xp+*yp;
}
int sub(int *zp, int *xp, int *yp) {
        *zp=*xp-*yp;
        return 0;
}
int main() {
        static int t1=100, t2=200;
        int sum, diff;
        sum=add(&t1, &t2);
    sub(&diff, &t1, &t2);
    printf( "sum=%d, diff=%d", sum, diff);
}
```

反汇编并分析说明以下内容:

（1）t1、t2的地址特征

（2）sum、diff的地址特征

（3）main和add、sub的栈帧变化过程，说明add和sub的栈帧底部是否在同一个位置？为什么？

翻转课堂

翻转题目二： 将可执行文件反汇编(基于LA32、LA64)，在Linux下进行分析，给出打印的精确结果（不用%f，可用%e）。

该贴给出的结果是在x86+Windows上得到的

C/C++ code

```
#include "stdafx.h"
int main(int argc, char* argv[])
{
    int a=10;
    double *p=(double*)&a;
    printf("%f\n",*p);              //结果为0.000000
    printf("%f\n",(double(a)));     //结果为10.000000

    return 0;
}
```

为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？

不都是强制类型转换吗？怎么会不一样

翻转课堂

翻转题目三:

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

在LA32和LA64上运行时结果各是什么?

上台报告:

要求做PPT并现场演示、讲解，根据报告情况加分。

程序的机器级表示

- 分以下四个部分介绍
  - 第一讲：　过程调用的机器级表示
    - 过程调用约定、变量的作用域和生存期
    - 按值传参和按地址传参、递归过程调用
    - 非静态局部变量的存储分配
    - 入口参数的传递和分配
  - 第二讲：　流程控制语句的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

数组的分配和访问

- 数组元素在内存的存放和访问
  - 例如，定义一个具有4个元素的静态存储型 short 数据类型数组A，可以写成“static short A[4];”
  - 第 i（0≤i≤3）个元素的地址计算公式为&A[0]+2*i。
  - 假定数组A的首地址存放在r12中，i 存放在r13中，现要将A[i]取到r14中，则汇编指令“alsl.w $r14,$r13,$r12,0x1”可计算出A[i]的地址，汇编指令“ld.h $r14,$r14,0”可将A[i]取到r14中。

LA32中若干数组的定义以及它们在内存中的存放情况如下所示：

数组定义	数组名	数组元素类型	数组元素大小（B）	数组大小（B）	起始地址	元素 i 的地址
char S[10]	S	char	1	10	&S[0]	&S[0]+i
char * SA[10]	SA	char *	4	40	&SA[0]	&SA[0]+4*i
double D[10]	D	double	8	80	&D[0]	&D[0]+8*i
double * DA[10]	DA	double *	4	40	&DA[0]	&DA[0]+4*i

数组元素在内存的存放和访问

SKIP

- LA32中分配在静态区的数组的初始化和访问

```
int buf[2] = {10, 20};
int main ( )
{
        int i, sum=0;
        for (i=0; i<2;
    i++)

    sum+=buf[i];
        return sum;
}
```

buf是在静态区分配的数组，链接后，buf在可执行文件的可读写数据段中分配了空间

```
a8000 <buf>:
a8000:    0a 00 00 00 14 00 00 00
```

语句“sum+=buf[i];”对应汇编指令序列

```
pcaddu12i  $r13, 152(0x98)
addi.w     $r13, $r13, -1680(0x970)
ld.w       $r12, $r22, -20(0xfec)
slli.w     $r12, $r12, 0x2
add.w      $r12, $r13, $r12
ld.w       $r12, $r12, 0
ld.w       $r13, $r22, -24(0xfe8)
add.w      $r12, $r13, $r12
st.w       $r12, $r22, -24(0xfe8)
```

根据PC+0x98000+0x970=0xa8000，可推出上述pcaddu12i指令的地址为PC=0x10690.

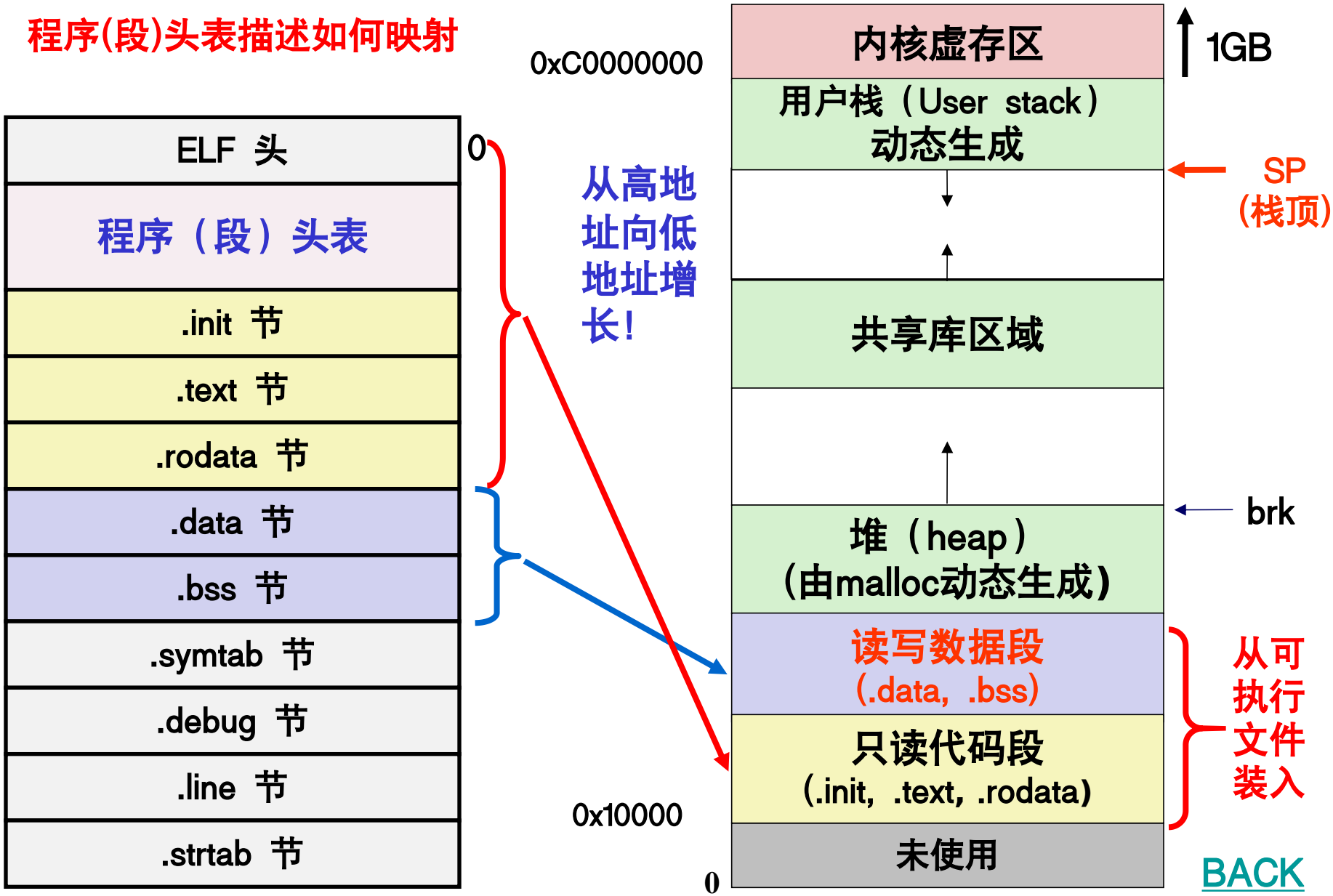
红色背景4条指令将buf[i]取到r12;

蓝色背景指令将sum取到r13

绿色背景指令将r12和r13内容相加，并存入sum

可执行文件的存储器映像

程序(段)头表描述如何映射



BACK

数组元素在内存的存放和访问

SKIP

- LA32中分配在静态区的数组的初始化和访问

```
int buf[2] = {10, 20};
int main ( )
{
        int i, sum=0;
        for (i=0; i<2;
    i++)

    sum+=buf[i];
        return sum;
}
```

buf是在静态区分配的数组，链接后，buf在可执行文件的可读写数据段中分配了空间

```
a8000 <buf>:
a8000:    0a 00 00 00 14 00 00 00
```

语句“sum+=buf[i];”对应汇编指令序列

```
pcaddu12i  $r13, 152(0x98)
addi.w     $r13, $r13, -1680(0x970)
ld.w       $r12, $r22, -20(0xfec)
slli.w     $r12, $r12, 0x2
add.w      $r12, $r13, $r12
ld.w       $r12, $r12, 0
ld.w       $r13, $r22, -24(0xfe8)
add.w      $r12, $r13, $r12
st.w       $r12, $r22, -24(0xfe8)
```

根据PC+0x98000+0x970=0xa8000，可推出上述pcaddu12i指令的地址为PC=0x10690.

红色背景4条指令将buf[i]取到r12;

蓝色背景指令将sum取到r13

绿色背景指令将r12和r13内容相加，并存入sum

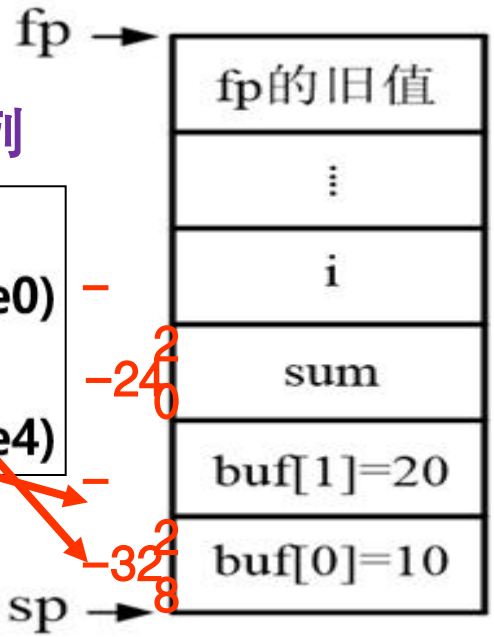
数组元素在内存的存放和访问

- LA32中auto型数组的初始化和访问

```
int adder ( )
{
        int buf[2] = {10, 20};
    int i, sum=0;
    for (i=0; i<2; i++)
                sum+=buf[i];
    return sum;
}
```

buf数组初始化汇编指令序列

```
addi.w   $r12, $r0, 10(0xa)
st.w     $r12, $r22, -32(0xfe0)
addi.w   $r12, $r0, 20(0x14)
st.w     $r12, $r22, -28(0xfe4)
```



语句“sum+=buf[i];”对应汇编指令序列

```
ld.w     $r12, $r22, -20(0xfec)    #R[r12]←M[R[r22]-20]=i
addi.w   $r13, $r22, -32(0xfe0)    #R[r13]←R[r22]-32，buf 首地址
alsl.w   $r12, $r12, $r13, 0x2     #R[r12]←R[r13]+i*4，buf[i]地址
ld.w     $r12, $r12, 0(0x0)        #R[r12]←M[R[r13]+i*4]，buf[i]
ld.w     $r13, $r22, -24(0xfe8)    #R[r13]←M[R[r22]-24]=sum
add.w    $r12, $r13, $r12          #R[r12]←sum+buf[i]
st.w     $r12, $r22, -24(0xfe8)    #sum←R[r12]
```

数组元素在内存的存放和访问

- 数组与指针

ü 在指针变量目标数据类型与数组类型相同的前提下，指针变量可以指向数组或数组中任意元素

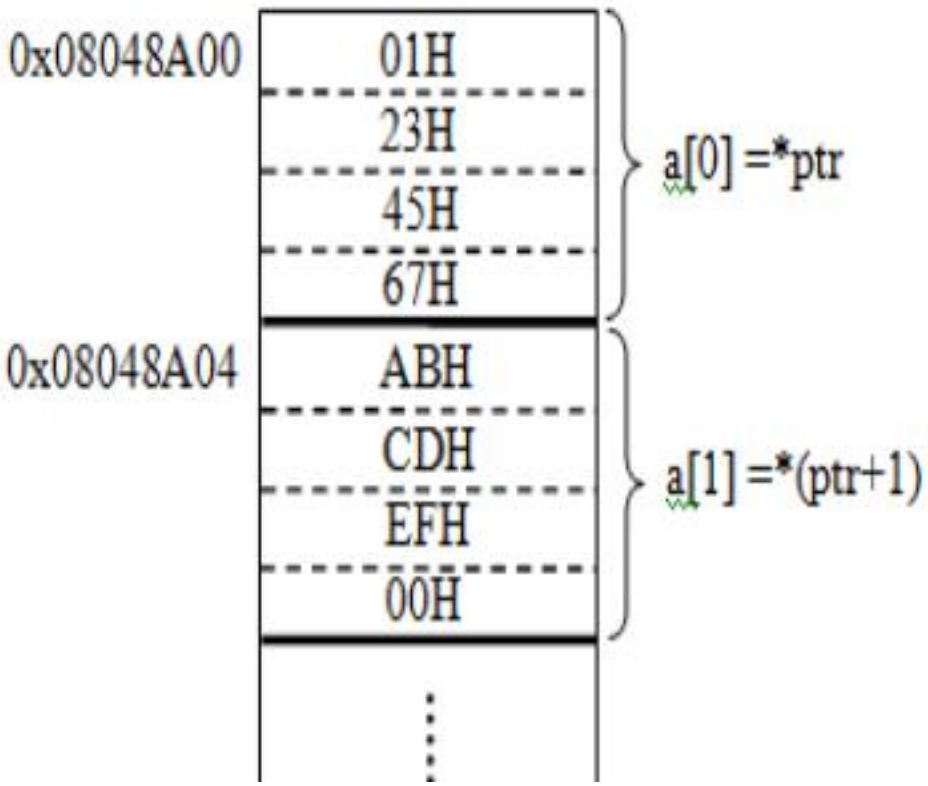
ü 以下两个程序段功能完全相同，a的值就是其首地址，即a=&a[0]，因而 a=ptr，从而有&a[i]=ptr+i=a+i 以及a[i]=ptr[i]=*(ptr+i)=*(a+i)。

（1）int a[10];

int *ptr=&a[0];

（2） int a[10], *ptr;

ptr=&a[0];



小端方式下a[0]=?,a[1]=?

a[0]=0x67452301, a[1]=0x0efcdab

数组首址0x8048A00在ptr中，ptr+i 并不是用0x8048A00加 i 得到，而是等于0x8048A00+4*i

数组元素在内存的存放和访问

- 数组与指针

序号	表达式	类型	值的计算方式	汇编代码
1	A	int *		
2	A[0]	int		
3	A[i]	int		
4	&A[3]	int *		
5	&A[i]-A	int		
6	*(A+i)	int		
7	*(&A[0]+i-1)	int		
8	A+i	int *		

问题:

假定数组A的首址SA在r12中，i在r13中，表达式结果在r14中，在LA32架构中，各表达式的计算方式以及汇编代码各是什么?

2、3、6和7对应汇编指令都需访存，指令中源操作数的寻址方式分别是“基址”、“基址加比例变址”、“基址加比例变址”和“基址加比例变址加位移”的方式，因为数组元素的类型为int型，故比例因子为4。

数组元素在内存的存放和访问

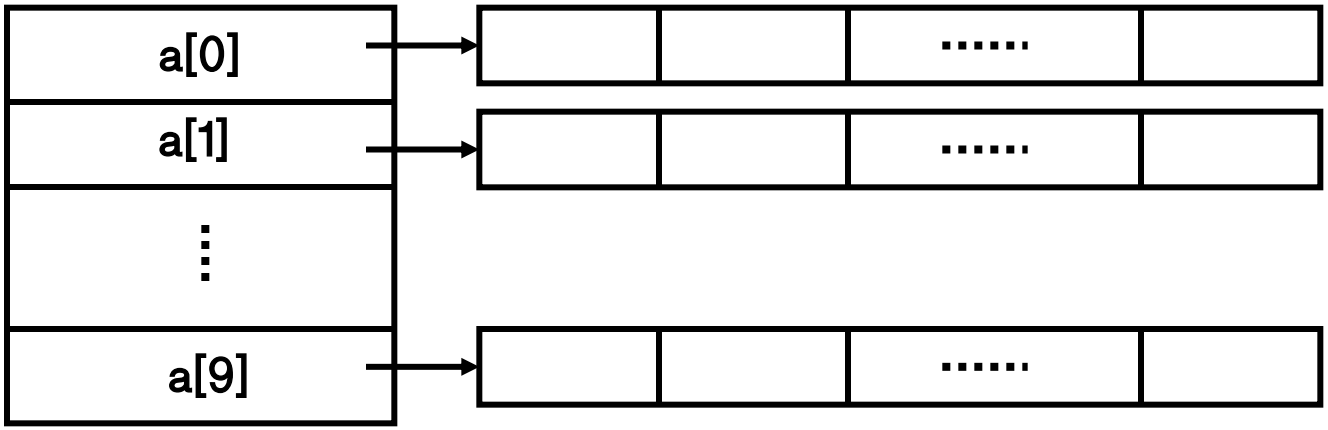
- **数组与指针** 假设A首址在r12，i在r13，表达式结果在r14中

表达式	类型	值的计算方式	汇编代码
A	int *	SA	addi.w $r14, $r12, 0(0x0)
A[0]	int	M[SA]	ld.w $r14, $r12, 0(0x0)
A[i]	int	M[SA+4*i]	alsl.w $r12, $r13, $r12, 0x2 ld.w $r14, $r12, 0(0x0)
&A[3]	int *	SA+12	addi.w $r14, $r12, 12(0xc)
&A[i]-A	int	(SA+4*i-SA)/4=i	or $r14, $r13, $r0
*(A+i)	int	M[SA+4*i]	alsl.w $r12, $r13, $r12, 0x2 ld.w $r14, $r12, 0(0x0)
*(&A[0]+i-1)	int	M[SA+4*i-4]	alsl.w $r12, $r13, $r12, 0x2 ld.w $r14, $r12, -4(0xffc)
A+i	int *	SA+4*i	alsl.w $r12, $r13, $r12, 0x2

数组元素在内存的存放和访问

- 指针数组和多维数组
  - 由若干指向同类目标的指针变量组成的数组称为指针数组。
  - 其定义的一般形式如下:

    存储类型 数据类型 *指针数组名[元素个数];
  - 例如，“int *a[10];”定义了一个指针数组a，它有10个元素，每个元素都是一个指向int型数据的指针。
    - 一个指针数组可以实现一个二维数组。



数组元素在内存的存放和访问

- 指针数组和多维数组

  – 计算一个两行四列整数矩阵中每一行数据的和。

按行优先方式存放数组元素

SKIP

```
main ( ) {
    static short num[ ][4]={ {2, 9, –1, 5},
                             {3, 8, 2, –6}};
    static short *pn[ ]={num[0], num[1]};
    static short s[2]={0, 0};
    int i, j;
    for (i=0; i<2; i++) {
        for (j=0; j<4; j++)
            s[i]+=*pn[i]++;
        printf (sum of line %d: %d\n” , i+1, s[i]);
    }
}
```

若处理“s[i]+=*pn[i]++;”时假设pn、i和s分别保存在r12、r13和r14寄存器中，则对应指令序列是什么?

若num=0xa8000,则num、pn和s在存储区中如何存放?

num=num[0]=&num[0][0]=0xa8000

```
a8000 <num>:
a8000:    02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
a8010 <pn>:
a8010:    00 80 0a 00 08 80 0a 00
a8018<s>:
a8018:    00 00 00 00
```

pn=&pn[0]=0x000a8010

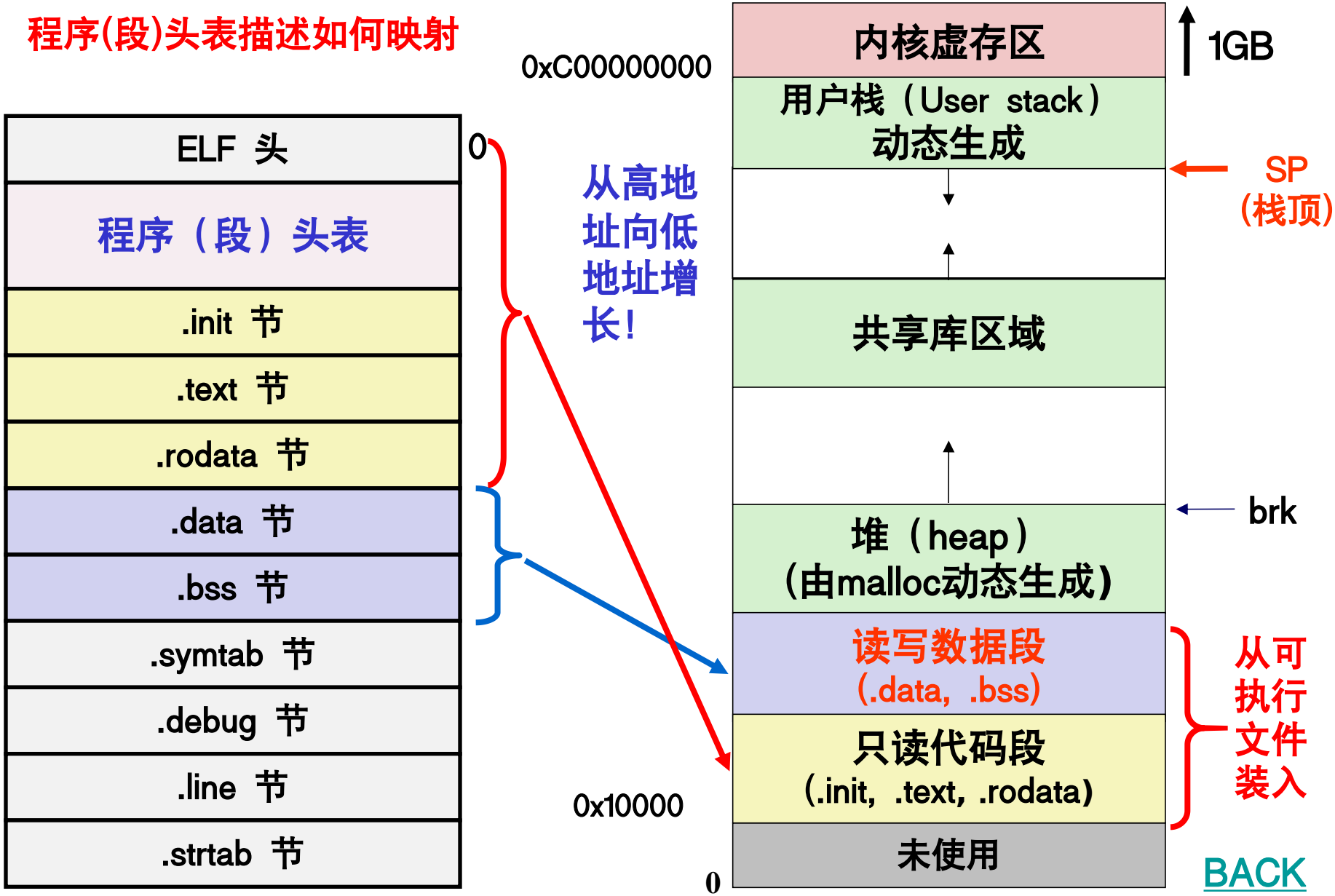
pn[0]=num[0]=0x000a8000

pn[1]=num[1]=0x000a8008

当i=1时，pn[i]=*(pn+i)=M[pn+4*i]=0x000a8008

可执行文件的存储器映像

程序(段)头表描述如何映射



BACK

数组元素在内存的存放和访问

- 指针数组和多维数组

按行优先方式存放数组元素

若处理“s[i]+=*pn[i]++;”时假设pn、i和s分别保存在r12、r13和r14寄存器中，则对应指令序列是什么?

```
alsl.w   $r15, $r13, $r12, 0x2    #R[r15]←&pn[i]=pn+i*4
ld.w     $r16, $r15, 0            #R[r16]←pn[i]，M[pn+i*4]
addi.w   $r17, $r16, 2(0x2)       #R[r17]←pn[i]+2，实现 pn[i]++
st.w     $r17, $r15, 0            #保存 pn[i]的新值
alsl.w   $r18, $r13, $r14, 0x1    #R[r18]←&s[i]=s+i*2
ld.h     $r19, $r18, 0            #R[r19]←s[i]，M[s+i*2]
ld.h     $r20, $r16, 0            #读取*pn[i]
add.w    $r19, $r19, $r20         #R[r19]←s[i]+*pn[i]
st.h     $r19, $r18, 0            #保存 s[i]的新值
```

第3~4行指令实现“pn[i]+1→pn[i]”，因pn[i]是指针，故“pn[i]+1→pn[i]”是指针运算，操作数长度为4B，即长度后缀为w，指针变量增量时应加目标数据长度。因为目标数据为short型，长度为2，所以增量时每次加2。

short型数据访存指令的长度后缀是h(表示半字，即16位)

结构体数据的分配和访问

- 结构体成员在内存的存放和访问
  - 分配在栈中的auto结构型变量的首地址由EBP或ESP来定位
  - 分配在静态区的结构型变量首地址是一个确定的静态区地址
  - 结构型变量 x 各成员首址可用“基址加偏移量”的寻址方式

```
struct cont_info {
        char id[8];
        char name [12];
        unsigned post;
        char
   address[100];
        char phone[20];
};
```

若变量x分配在地址0xa8000开始的区域，那么x=&(x.id)= 0xa8000 （若x在r12中）
&(x.name)= 0xa8000+8= 0xa8008
&(x.post)= 0xa8000+8+12= 0xa8014
&(x.address)= 0xa8000+8+12+4= 0xa8018
&(x.phone)= 0xa8000+8+12+4+100= 0xa807c

```
struct cont_info x={ “0000000”, “ZhangS”, 210022, “273 long street,
High Building #3015”, “12345678” };
```

x初始化后，在地址0xa8008到0xa800D处是字符串“ZhangS”， 0xa800E处是字符‘\0’，从0xa800F到0xa8013处都是空字符。

若x分配在r12中，xpost在r13中，“unsigned xpost=x.post;”对应汇编指令为“ld.wu $r13, $r12, 20(0x14)”

结构体数据的分配和访问

- 结构体数据作为入口参数

按地址调用

```
void stu_phone1 ( struct cont_info *s_info_ptr)          stu_phone1(&x)
{
    printf ( "%s phone number: %s" , (*s_info_ptr).name, (*s_info_ptr).phone);
}
void stu_phone2 ( struct cont_info s_info)      按值调用   stu_phone2(x)
{
    printf ( "%s phone number: %s" , s_info.name, s_info.phone);
}
```

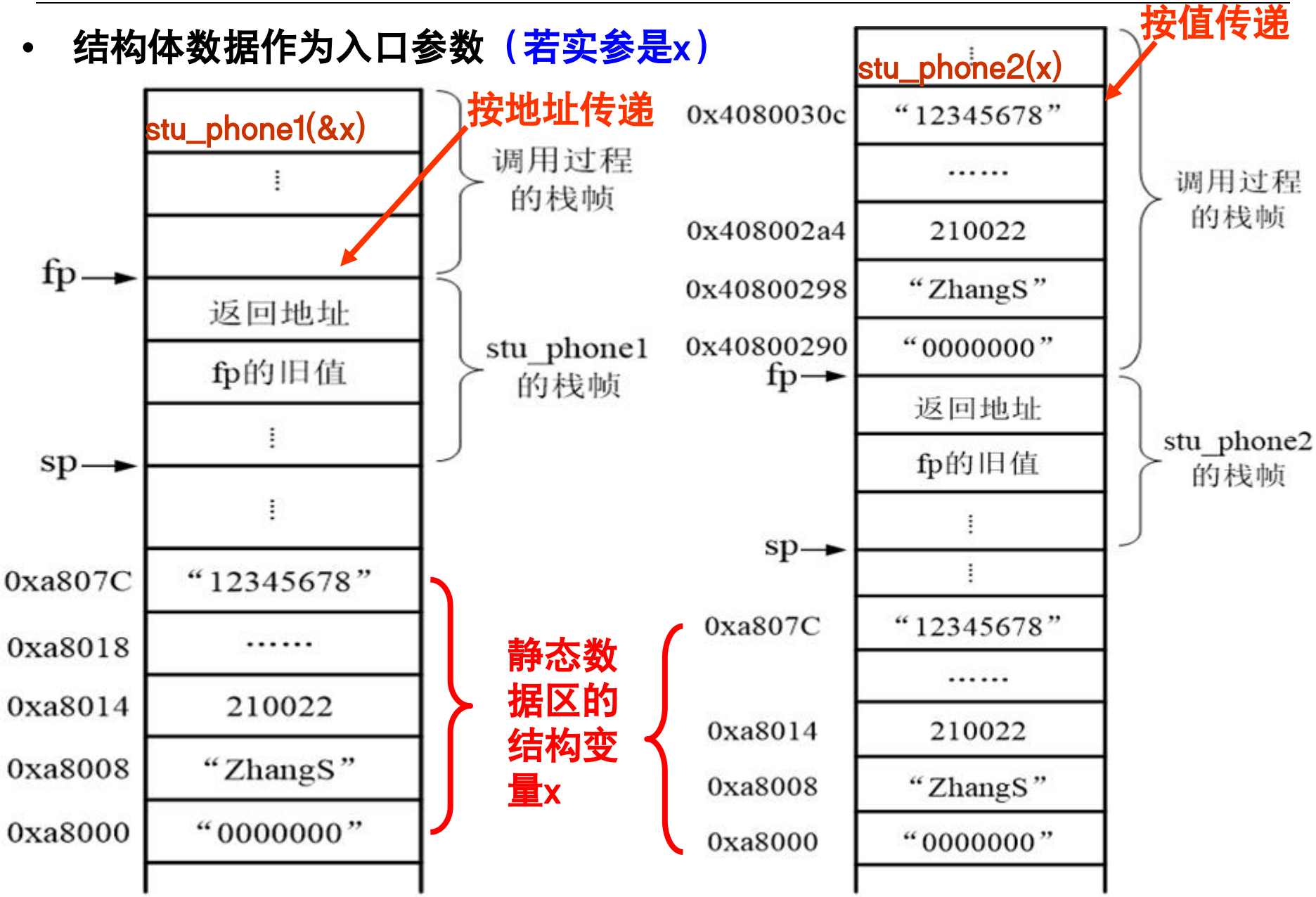
  - 当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构

    struct cont_info x={ "0000000" , "ZhangS" , 210022, "273 long street, High Building #3015" , "12345678" };

  - 若采用按值传递，则结构成员都要复制到栈中参数区，这既增加时间开销又增加空间开销，且更新后的数据无法在调用过程使用(如前面的swap(a,b)例子)
  - 通常应按地址传递，即：在执行BL指令前，仅需传递指向结构体的指针而不需复制每个成员到栈中

结构体数据的分配和访问

- 结构体数据作为入口参数（若实参是x）



结构体数据的分配和访问

- 按地址传递参数　　stu_phone1(&x)

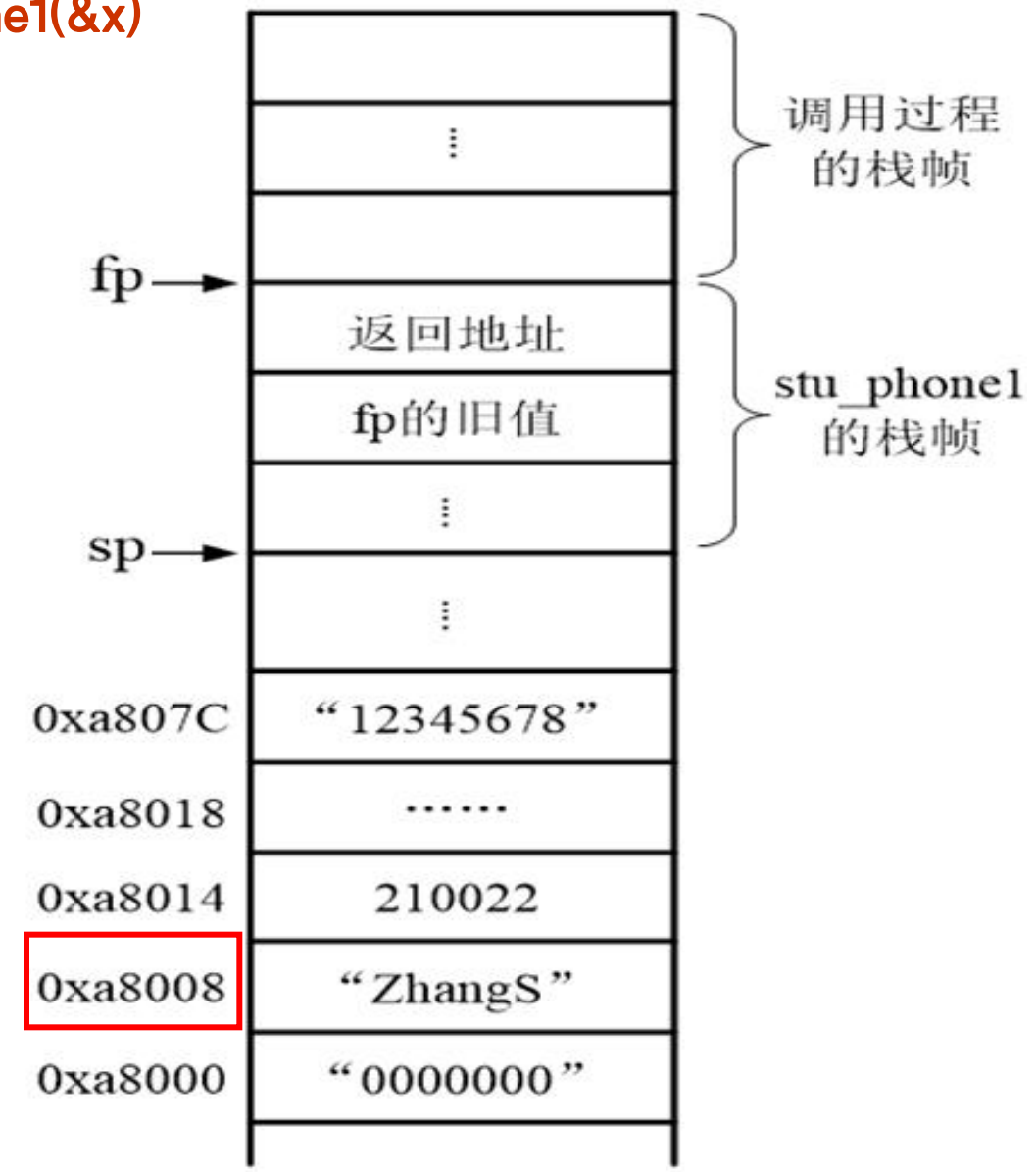
(*s_info_ptr).name 可写成 s_info_ptr->name,

x的地址0xa8000作为实参在r4（参数寄存器a0）中

将(*stu_info_ptr).name的地址送r12的指令如下:

addi.w $r12, $r4, 8(0x8)

执行完上述指令，r12中存放的是字符串"ZhangS"在静态存储区内的首地址0xa8008。



结构体数据的分配和访问

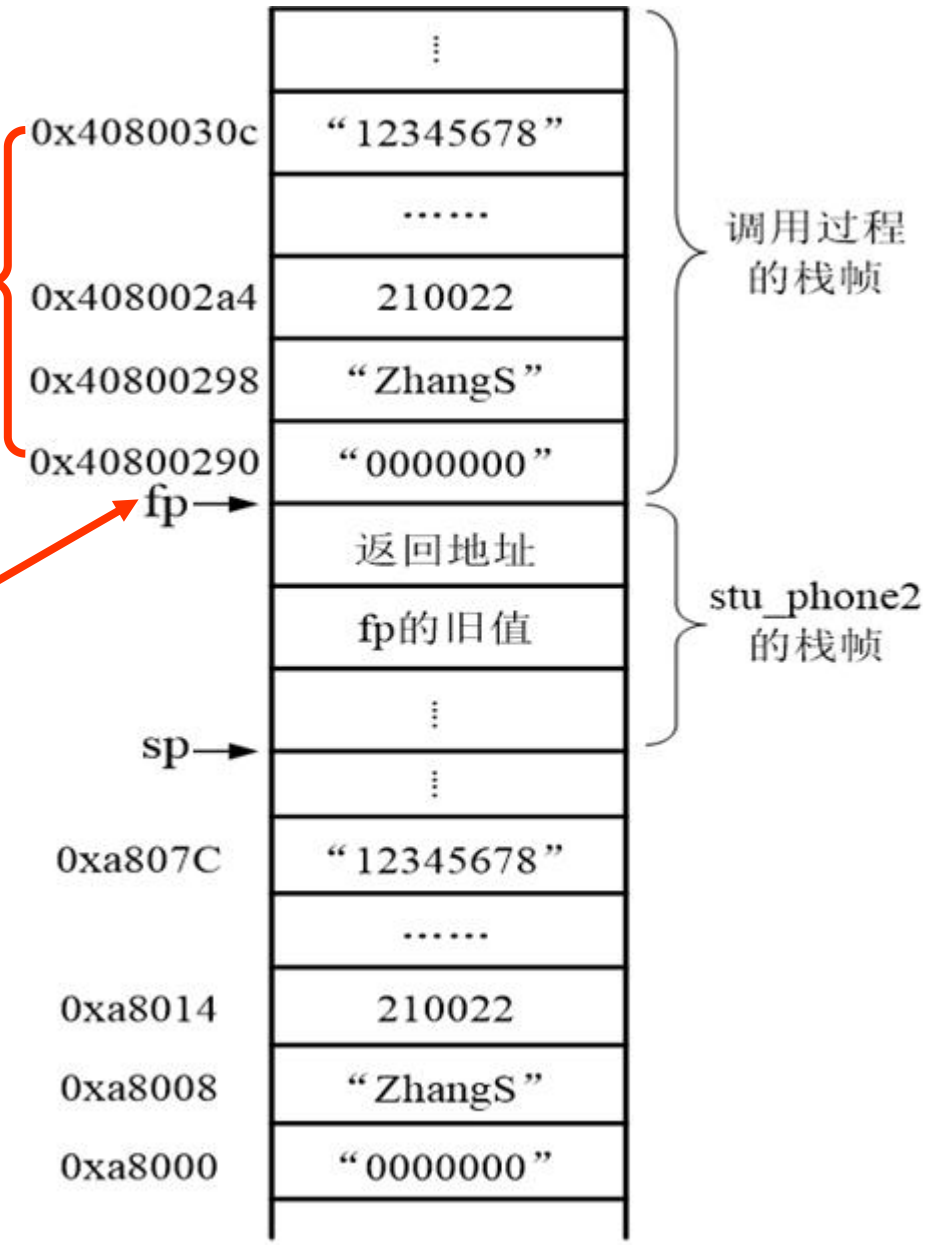
- 按值传递参数　　stu_phone2(x)

调用函数把x保存到参数区，其首地址0x40800290作为参数传入r4。可使用表达式stu_info.name来引用结构体成员name。实现将stu_info.name所在地址送r12的指令如下：

addi.w $r12, $r4, 8(0x8)

该指令将R[fp]+8的值送入r12，因此，r12中存放的是字符串“ZhangS”在栈中参数区内的首地址0x40800298。

stu_phone1和stu_phone2功能相同，但后者开销更大，它需对结构体成员整体从静态区复制到栈中，因而需执行很多条传送指令，从而使执行时间更长，并占更多栈空间和代码空间，特别是，按值传递时，无法获得更新后的结果



联合体数据的分配和访问

```
union  uarea  {
        char
c_data;
        short
s_data;
        int
i_data;
        long
l_data;
};
```

联合体各成员共享存储空间，按最大长度成员所需空间大小为目标

在LA32中编译时，long和int长度一样，故uarea所占空间为4字节。而对于与uarea有相同成员的结构型变量来说，其占用空间大小至少有11字节，对齐的话则占用更多。

- 通常用于特殊场合，如，当事先知道某种数据结构中的不同字段的使用时间是互斥的，就可将这些字段声明为联合，以减少空间。
- 但有时会得不偿失，可能只会减少少量空间却大大增加处理复杂性。

联合体数据的分配和访问

- 可实现对相同位序列进行不同数据类型的解释

```
unsigned float2unsign( float f)
{
    union {
            float f;
            unsigned u;
      } tmp_union;
      tmp_union.f=f;
      return tmp_union.u;
}
```

函数形参是float型，按值传递参数，因而传递过来的实参是float型数据，赋值给非静态局部变量（联合体变量成员）

过程体中主要指令序列如下:

```
fst.s   $fa0, $fp, -24(0xfe8)
ld.w    $a0, $fp, -24(0xfe8)
```

fst.s将fa0中的f送到tmp_union.f的存储空间，ld.w读取同样的数据送入a0。可以看到tmp_union.f和tmp_union.u存储空间具有相同的栈地址R[fp]-24

问题：float2unsign(10.0)=?

$2^{30}+2^{24}+2^{21}=1092616192$

从该例可看出：机器级代码并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

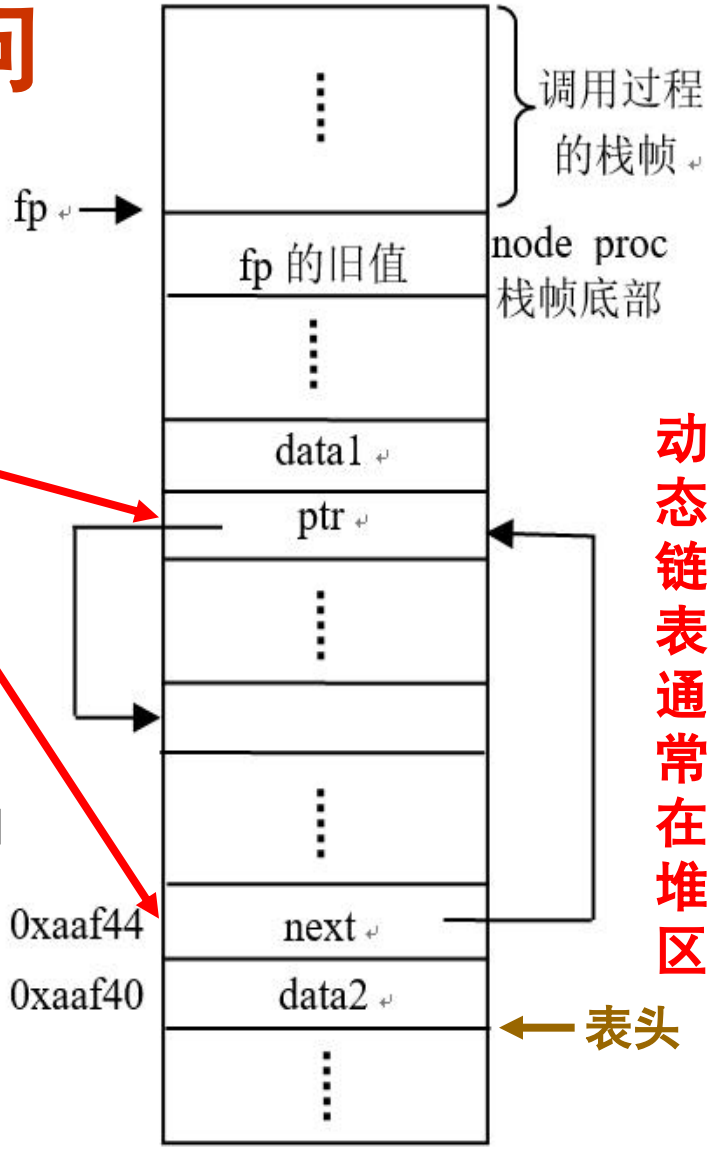
联合体数据的分配和访问

- 利用嵌套可定义链表结构

```
union node {
    struct {
        int *ptr;
        int data1
    } node1;
    struct {
        int data2;
        union node *next;
    } node2;
};
```

```
ld.w  $r13, $r4, 4(0x4)
ld.w  $r12, $r13, 0
ld.w  $r12, $r12, 0
ld.w  $r14, $r4, 0
add.w $r12, $r12, $r14
st.w  $r12, $r13, 4(0x4)
```

链表首址0xaaf40作为第1个入口参数在r4中，第1行指令执行后，r13中是指针next。第2~3行指令执行后，r12中是*(np->node2.next->node1.ptr)。第4行指令执行后，r14中是np->node2.data2




```
void node_proc ( union node *np) {
  np->node2.next->node1.data1=*(np->node2.next->node1.ptr)+np->node2.data2;
}
```

数据的对齐

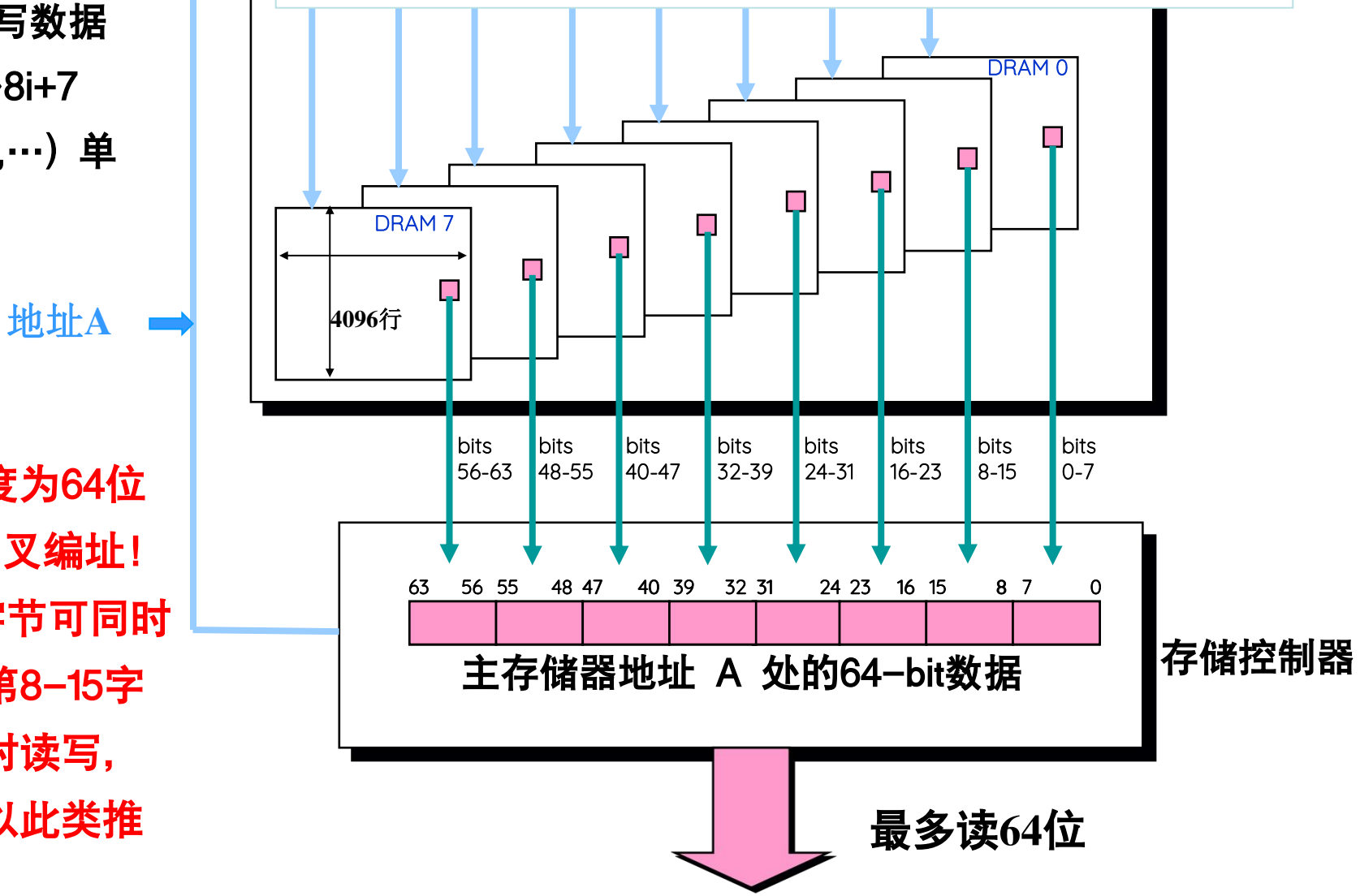
SKIP

- CPU访问主存时只能一次读取或写入若干特定位
  - 例如，若每次最多读写64位，则第0–7字节可同时读写，第8–15字节可同时读写，……，以此类推
- 按边界对齐可使读写数据位于8i~8i+7(i=0,1,2,…）单元内
- 最简单的对齐策略是，**按其数据长度对齐**。如，int型地址是4的倍数，short型地址是2的倍数，double和long long型则8的倍数，float型是4的倍数，char不对齐。Windows遵循的ABI规范采用该简单对齐策略
- LA32 ABI中定义的对齐策略：short型数据地址是2的倍数，其他如int、long、float和指针等类型数据的地址都是4的倍数，long long和double类型数据的地址都是8的倍数。LoongArch定义long double型数据为128位，即16B，GCC遵循该定义，其地址按16字节对齐。
- LA64 ABI定义中long和指针类型数据都是8字节大小，因此其地址都是8的倍数，其余的数据类型与LA32 ABI中定义一致。

从该存储器结构可理解为什么规定数据对齐存放。

例如，一个32位int型数据若存放在第8、9、10、11这4个单元，则需要访问几次内存？若存放在6、7、8、9这4个单元，则需要访问几次内存？

按边界对齐，可使读写数据位于8i~8i+7 (i=0,1,2,…) 单元内



存储宽度为64位=8B，交叉编址！第0-7字节可同时读写，第8-15字节可同时读写，……，以此类推

Alignment(对齐)

BACK

如： int i, short k, double x, char c, short j,……

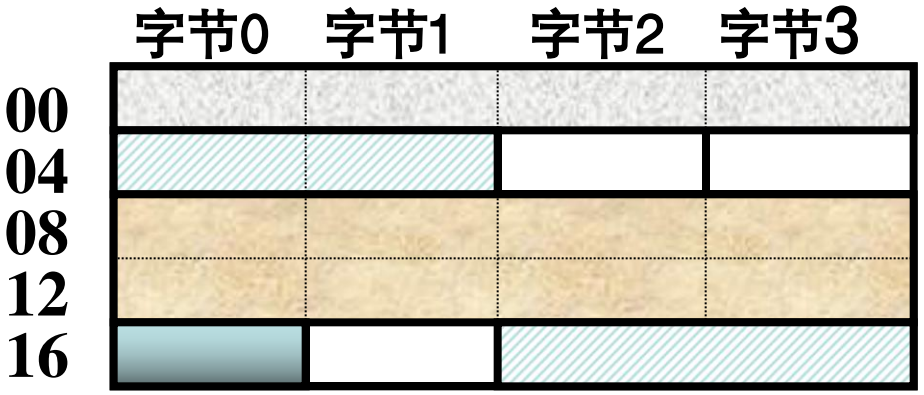
按字节编址

每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节

按边界对齐

x： 2个周期

j： 1个周期



则： &i=0; &k=4; &x=8; &c=16; &j=18;……

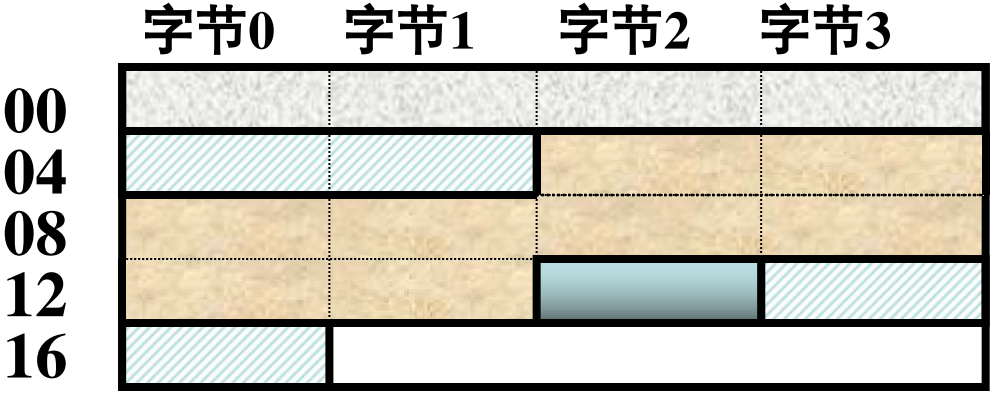
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

边界不对齐

x： 3个周期

j： 2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

Windows中的对齐和分配顺序

```
#include   ……
int  main()
{
    int  a;
    char  b;
    int  c;
    printf( “a:  0x%08x\n",&a);
    printf( “b:  0x%08x\n",&b);
    printf( “c:  0x%08x\n",&c);
    return  0;
}
```

a、b、c不一定按顺序分配，但a和c的地址总是4的倍数，b的地址则不一定是4的倍数。

用VC编译后的执行结果：

a: 0x0012ff7c

b: 0x0012ff7b

c: 0x0012ff80

顺序：b(1B)–a(4B)–c(4B)

用Dev–C++编译后的执行结果：

a: 0x0022ff7c

b: 0x0022ff7b

c: 0x0022ff74

顺序：c(4B)–隔3B–b(1B)–a(4B)

用lcc编译后的执行结果：

a: 0x0012ff6c

b: 0x0012ff6b

c: 0x0012ff64

顺序：同上（大地址–>小地址）

数据的对齐

LoongArch ABI对struct结构体数据的对齐方式有如下几条规则:

① 整个结构体变量的对齐方式与其中对齐方式最严格的成员相同;

② 每个成员在满足其对齐方式的前提下，取最小的可用位置作为成员在结构体中的偏移量，这可能导致内部插空;

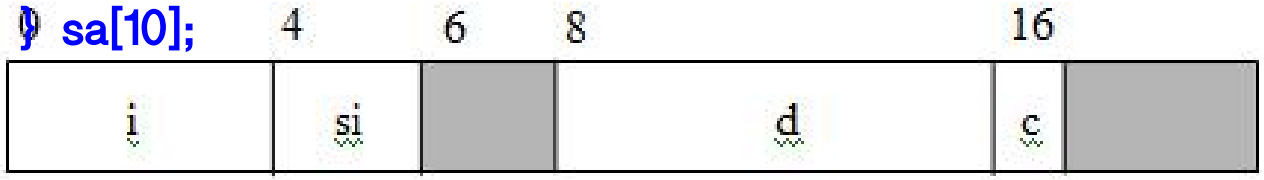
③ 结构体大小应为对齐边界长度的整数倍，这可能会导致尾部插空。

前两条规则是为了保证结构体中的任意成员都能以对齐的方式访问。

第③条规则是为了保证使结构体数组中的每个元素都能满足对齐要求

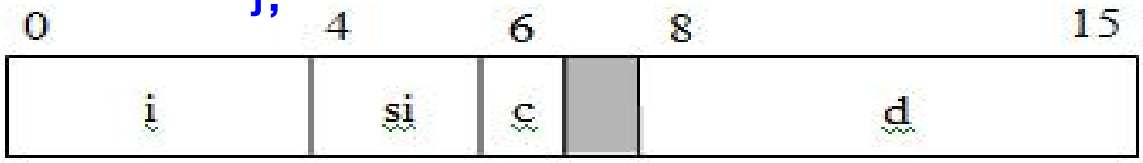
```
struct SDT {
    int i;
    short si;
    double d;
    char c;
} sa[10];
```

结构数组变量的最末可能需要插空，以使每个数组元素都按8字节边界对齐




```
struct SD {
    int i;
    short si;
    char c;
    double d;
};
```

结构变量首地址按8字节边界对齐



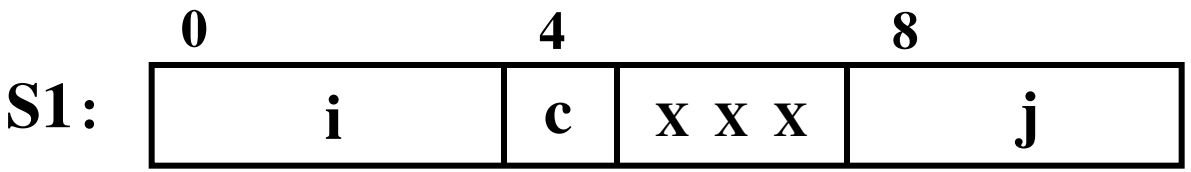
只要SD首址按8B边界对齐，所有字段都能按要求对齐

Alignment(对齐) 举例

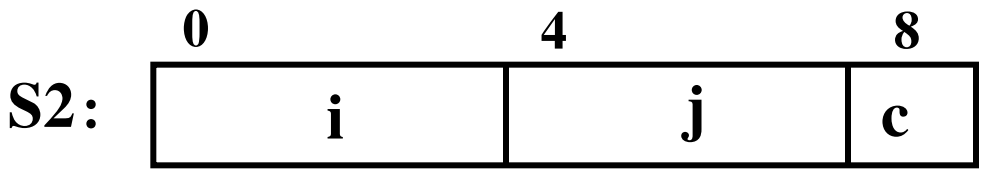
例如，考虑下列两个结构声明：

```
struct S1 {
        int     i;
        char    c;
        int     j;
};
```

```
struct S2 {
        int     i;
        int     j;
        char    c;
};
```

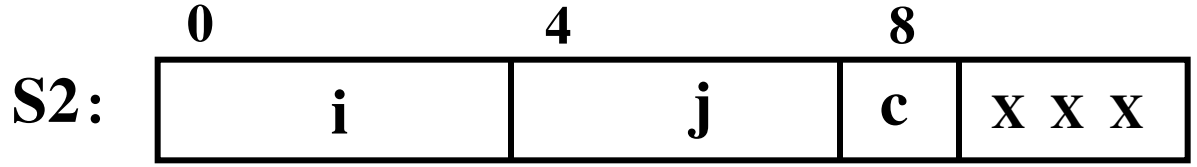


需要12个字节



至少需要9个字节

对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？ 不能！



需要12个字节

Alignment(对齐) 举例

Alignment(对齐)

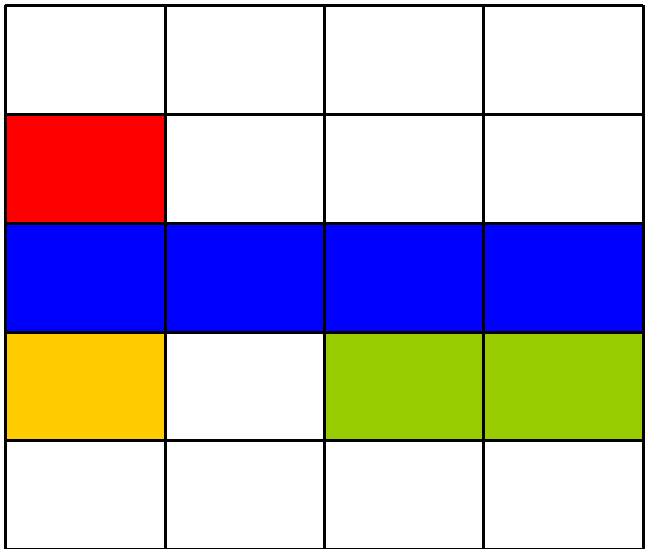
```
struct record {
        char     a;
        int      b;
        char   c;
        short  d;
 } ;
struct  record R[10];
```

在上述定义中，数组R占用多少字节？代码是否可以优化？

```
11:     R[0].a=1;
00401028 C6 45 E8 01          mov       byte ptr [ebp-18h],1
12:     R[0].b=2;
0040102C C7 45 EC 02 00 00 00 mov       dword ptr [ebp-14h],2
13:     R[0].c=3;
00401033 C6 45 F0 03          mov       byte ptr [ebp-10h],3
14:     R[0].d=4;
00401037 66 C7 45 F2 04 00    mov       word ptr [ebp-0Eh],offset main+2Bh (0040103b)
15:     R[1].a=5;
0040103D C6 45 F4 05          mov       byte ptr [ebp-0Ch],5
16:     R[1].b=6;
00401041 C7 45 F8 06 00 00 00 mov       dword ptr [ebp-8],6
17:     R[1].c=7;
00401048 C6 45 FC 07          mov       byte ptr [ebp-4],7
18:     R[1].d=8;
0040104C 66 C7 45 FE 08 00    mov       word ptr [ebp-2],offset main+40h (00401050)
```

地址: 0x0012FF48

```
0012FF20  CC CC CC CC CC CC CC CC
0012FF28  CC CC CC CC CC CC CC CC
0012FF30  01 CC CC CC 02 00 00 00
0012FF38  03 CC 04 00 05 CC CC CC
0012FF40  06 00 00 00 07 CC 08 00
0012FF48  88 FF 12 00 39 12 40 00
0012FF50  01 00 00 00 A0 0E 30 00
0012FF58  C0 0D 30 00 00 00 00 00
```



Alignment(对齐) 举例

Alignment(对齐)

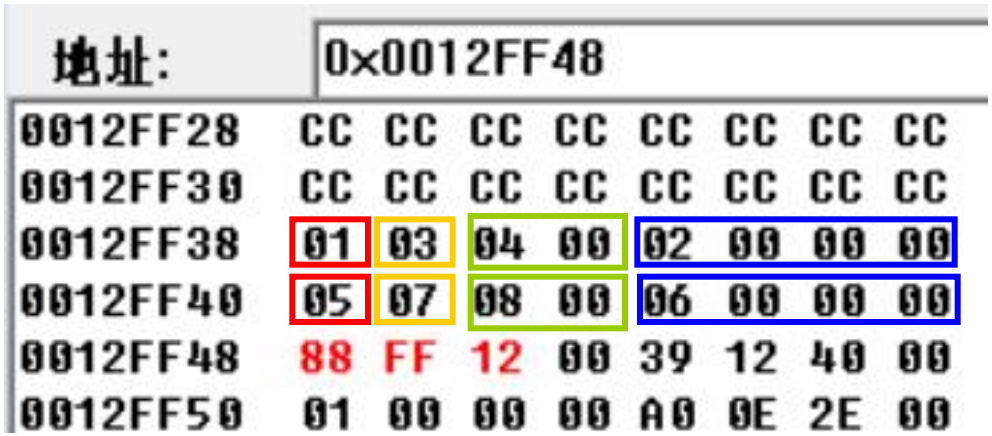
```
struct record {
  char        a;
  int   b;
  char        c;
  short       d;
 } ;
struct  record R[10];
```

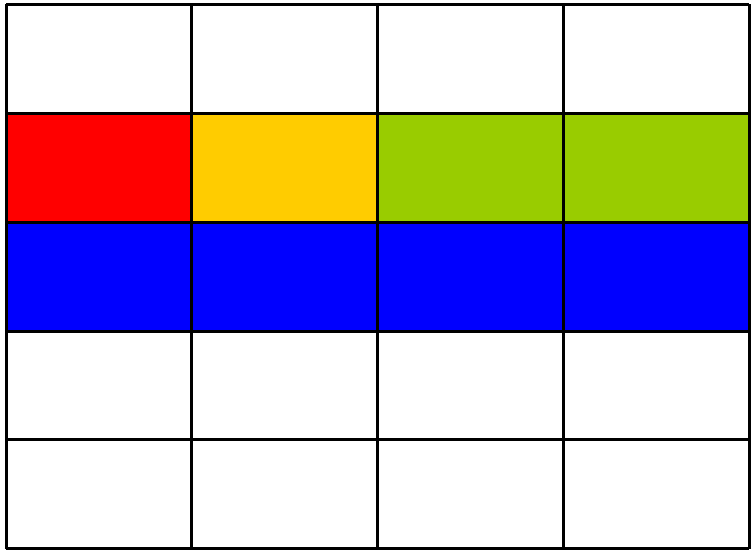
```
struct record {
  char        a;
  char        c;
  short       d;
  int         b;
 } ;
struct  record R[10];
```

上述定义中，数组R占用多少字节？代码是否可以优化？

数组R从120字节减少到80字节

```
11:     R[0].a=1;
00401028 C6 45 F0 01          mov     byte ptr [ebp-10h],1
12:     R[0].b=2;
0040102C C7 45 F4 02 00 00 00 mov     dword ptr [ebp-0Ch],2
13:     R[0].c=3;
00401033 C6 45 F1 03          mov     byte ptr [ebp-0Fh],3
14:     R[0].d=4;
00401037 66 C7 45 F2 04 00    mov     word ptr [ebp-0Eh],offset main+2Bh (0040103b)
15:     R[1].a=5;
0040103D C6 45 F8 05          mov     byte ptr [ebp-8],5
16:     R[1].b=6;
00401041 C7 45 FC 06 00 00 00 mov     dword ptr [ebp-4],6
17:     R[1].c=7;
00401048 C6 45 F9 07          mov     byte ptr [ebp-7],7
18:     R[1].d=8;
0040104C 66 C7 45 FA 08 00    mov     word ptr [ebp-6],offset main+40h (00401050)
```





对齐方式的设定

#pragma pack(n)

- 为编译器指定结构体或类内部的成员变量的对齐方式。
- 当自然边界（如int型按4字节、short型按2字节、float按4字节）比n大时，按n字节对齐。
- 缺省或#pragma pack()，按自然边界对齐。

__attribute__((aligned(m)))

- 为编译器指定一个结构体或类或联合体或一个单独的变量(对象)的对齐方式。
- 按m字节对齐(m必须是2的幂次方)，且其占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐。

__attribute__((packed))

- 不按边界对齐，称为紧凑方式。

对齐方式的设定

```
#include<stdio.h>
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出:

```
Struct size is: 1024, aligned on 1024
Allocate f1 on address: 0x0
Allocate f2 on address: 0x4
Allocate f3 on address: 0x5
Allocate f4 on address: 0x8
Allocate f5 on address: 0xc
```

```c
#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

输出结果是什么?

```
size=15
size=20
size=24
```

```c
#include <stdio.h>
#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(1)，结果又是什么？

```
size=15
size=15
size=16
```

```c
#include <stdio.h>
#pragma pack(2)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(2)，结果又是什么?

size=15
size=16
size=16

x86-64架构中数据的对齐

- x86-64中各类型数据遵循一定的对齐规则，而且更严格
- x86-64中存储器访问接口被设计成按8字节或16字节为单位进行存取，其对齐规则是，任何K字节宽的基本数据类型和指针类型数据的起始地址一定是K的倍数。
  - short型数据必须按2字节边界对齐
  - int、float等类型数据必须按4字节边界对齐
  - long型、double型、指针型变量必须按8字节边界对齐
  - long double型数据必须按16字节边界对齐

具体的对齐规则可以参考AMD64 System V ABI手册。

程序的机器级表示

- 分以下四个部分介绍
  - 第一讲：　过程调用的机器级表示
    - 过程调用约定、变量的作用域和生存期
    - 按值传参和按地址传参、递归过程调用
    - 非静态局部变量的存储分配
    - 入口参数的传递和分配
  - 第二讲：　流程控制语句的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

过程调用举例

例：应始终返回d[0]中的3.14，但并非如此。Why?

```
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];
}
```

IA-32+Linux平台上的执行结果:

fun(0)　　3.14
fun(1)　　3.14
fun(2)　　3.1399998664856
fun(3)　　2.0000061035156
fun(4)　　3.14，然后存储保护错

为何每次返回不一样

？为什么会引起保护错

？栈帧中的状态如何?

不同系统上执行结果可能不同

例如，编译器对局部变量分配方式可能不同

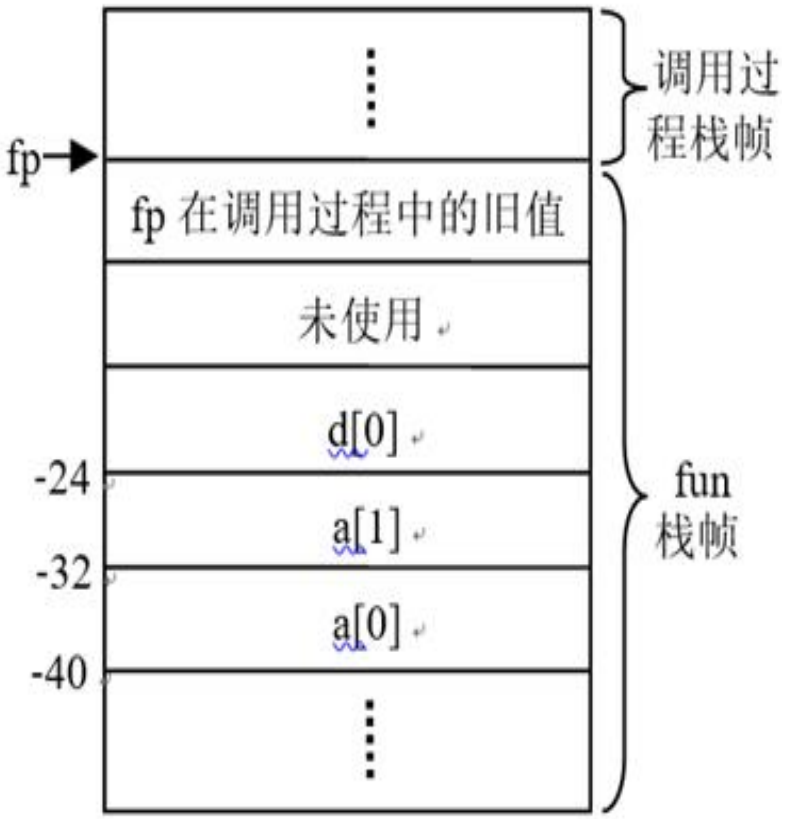
过程调用举例

fun(2) = 0.00
fun(3) = 3.14
fun(4) = 3.14，然后存储保护错

```
double fun(int i) {
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824;
    return d[0];
}
```

在LA64机器中：
当i=0或1，OK
当i=2，d[0]=0x0000000040000000
d[0]被改变为很小的非规格化数
当i=3，0x0000000040000000在未使用处
当i=4，fp的旧值被改变

```
addi.d   $sp, $sp, -64(0xfc0)      栈帧大小为64
std      $fp, $sp, 56(0x38)        fp存入栈帧底部
addi.d   $fp, $sp, 64(0x40)        fp指向栈帧底部
pcaddu12i $t0, 87(0x57)
addi.d   $t0, $t0, -1044(0xbec)    } d[0]=3.14;
fld.d    $fa0, $t0, 0
fst.d    $fa0, $fp, -24(0xfe8)
addi.d   $t0, $fp, -40(0xfd8)
alsl.d   $t0, $a0, $t0, 0x3        } a[i]=1073741824;
lu12i.w  $t1, 262144(0x40000)
st.d     $t1, $t0, 0(0x0)
fld.d    $fa0, $fp, -24(0xfe8)     return d[0]
ld.d     $fp, $sp, 56(0x38)        恢复fp
addi.d   $sp, $sp, 64(0x40)        释放栈帧
Jirl     $zero, $ra, 0
```



越界访问和缓冲区溢出

```
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];
}
```

在LA64机器中：

当i=0或1，OK

当i=2，d[0]=0x0000000040000000

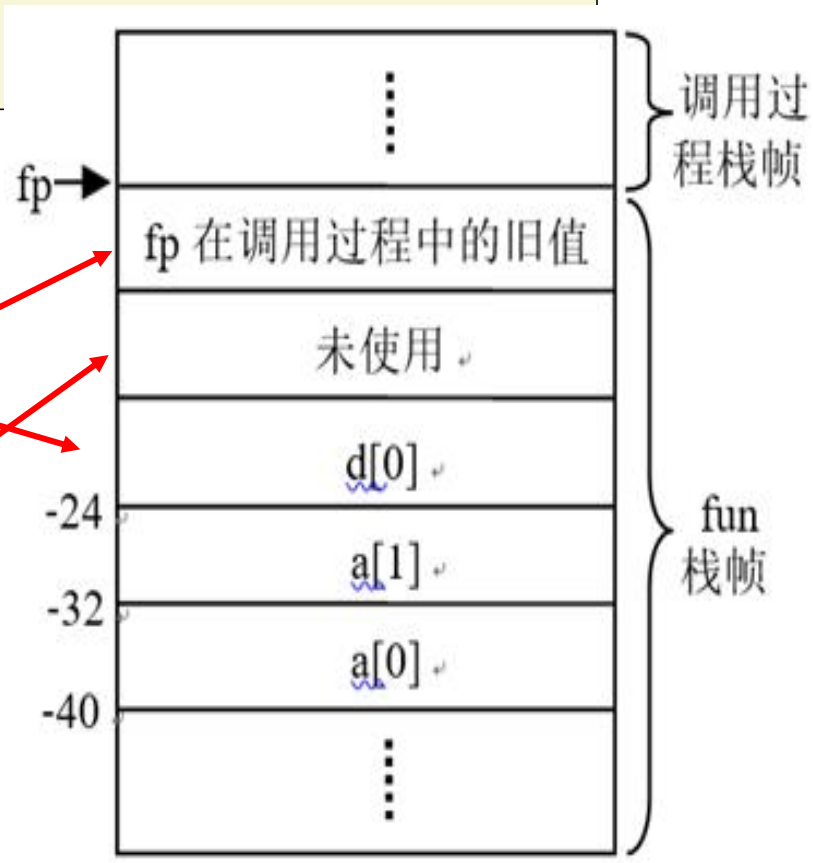
d[0]被改变为很小的非规格化数

当i=3，0x0000000040000000在未使用处

当i=4，fp的旧值被改变

为什么当 i>1 就有问题？

因为数组访问越界！



越界访问和缓冲区溢出

- C语言中的数组元素可使用指针来访问，因而对数组的引用没有边界约束，也即程序中对数组的访问可能会有意或无意地超越数组存储区范围而无法发现
- C标准规定，数组越界访问属于未定义行为，访问结果是不可预知的
- 数组存储区可看成是一个缓冲区，超越数组存储区范围的写入操作称为缓冲区溢出
- 例如，对于一个有10个元素的char型数组，其定义的缓冲区有10个字节。若写一个字符串到这个缓冲区，那么只要写入的字符串多于9个字符（结束符‘\0’占一个字节），就会发生“写溢出”
- 缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在
- 缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。利用缓冲区溢出攻击，可导致程序运行失败、系统关机、重新启动等后果

main()函数的原型

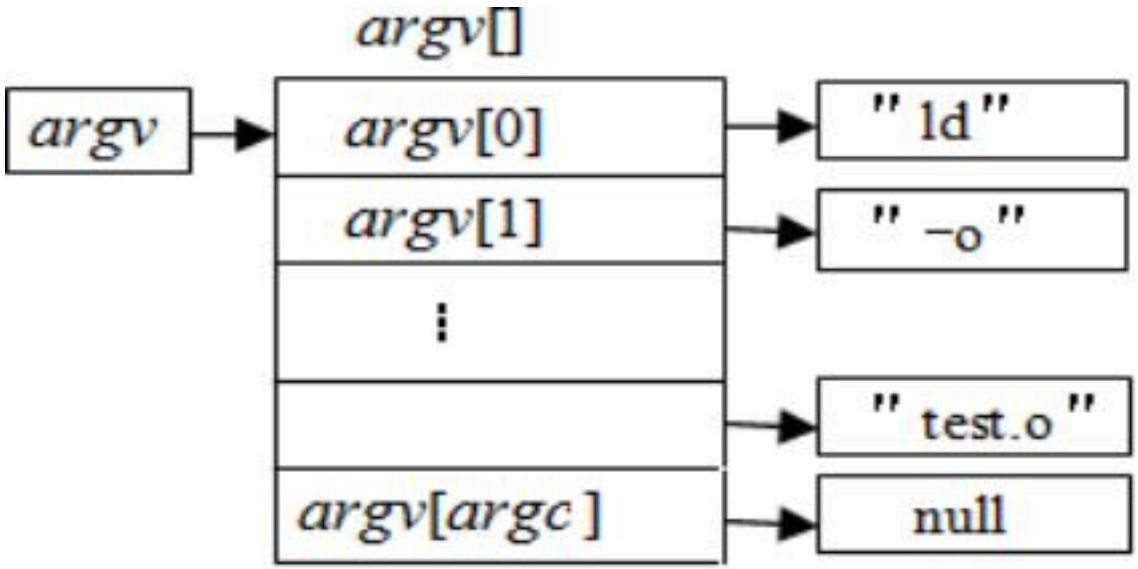
- 主函数main()的原型形式如下：

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

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

  argc：参数列表长度，参数列表中开始是命令名（可执行文件名），最后以NULL结尾。例：当“.\hello” 时，argc=1

  例：当“ld –o test main.o test.o” 时，argc=5



越界访问和缓冲区溢出

- 造成缓冲区溢出的原因是**没有对栈中作为缓冲区的数组的访问进行越界检查**。

```
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];
}
```

```
double fun(int i)
{
   volatile double d[1] = {3.14};
   volatile long int a[2];
   if (i<2)   a[i] = 1073741824;
   return d[0];
}
```

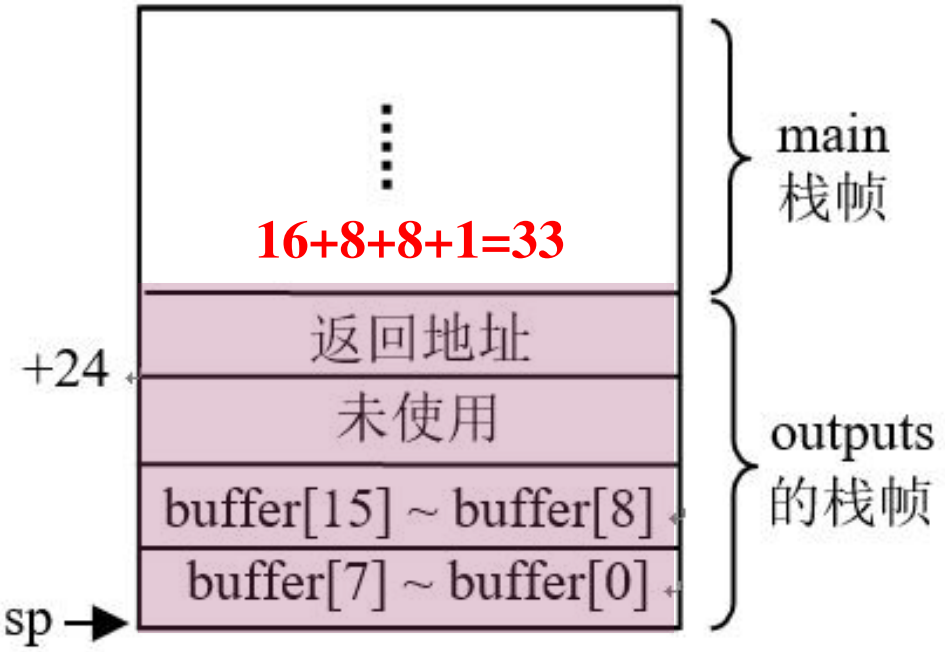
越界访问和缓冲区溢出

举例：利用缓冲区溢出转到自设的程序hacker去执行

```
#include "stdio.h"
#include "string.h"
void outputs(char *str) {
      char buffer[16];
      strcpy(buffer,str);
      puts(buffer);
}
void hacker(void) {
      printf("being hacked\n");
}
int main(int argc, char *argv[])
{
      outputs(argv[1]);
      return 0;
}
```

假定可执行文件名为test

outputs漏洞：当命令行中构造的33个字符的字符串通过strcpy函数复制到buffer起始处，并将攻击代码hacker()首地址置于字符串结束符\0前8字节，则在执行完strcpy()后，hacker代码首地址将置于main栈帧下的返回地址处，从而转hacker执行



程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来加载并执行程序。
- execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回–1，并将控制权交给调用程序；若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。
- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp);　　或者：

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

前述例子：　“.\test 0123456789ABCDEFXXXXXXXX8　　”　,argc=2

argv[0]　　argv[1]

缓冲区溢出攻击

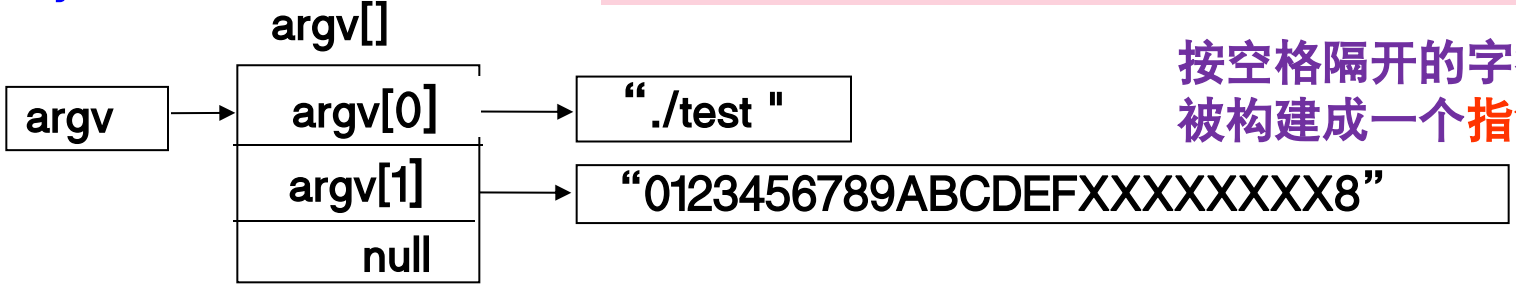
假定hacker首地址为0x1 2000 0738，outputs()的返回地址为0x1 2000 0768，可编写如下的攻击代码实施攻击。

```
#include "stdio.h"
char code[]=
"0123456789ABCDEFXXXXXXXX"
"\x38\x07\x00\x20\x01\x00\x00\x00"
"\x00";
int main(void)
{
        char *argv[3];
        argv[0]="./test";
        argv[1]=code;
        argv[2]=NULL;
        execve(argv[0],argv,NULL);
        return 0;
}
```

```
#include "stdio.h"
#include "string.h"
void outputs(char *str) {
        char buffer[16];
       strcpy(buffer,str);
       printf("%s \n", buffer);
}
void hacker(void) {
       printf("being hacked\n");
}
int main(int argc, char *argv[])
{
       outputs(argv[1]);
       return 0;
}
```

可执行文件名为test

命令行: .\test 0123456789ABCDEFXXXXXXXX8



按空格隔开的字符串被构建成一个指针数组

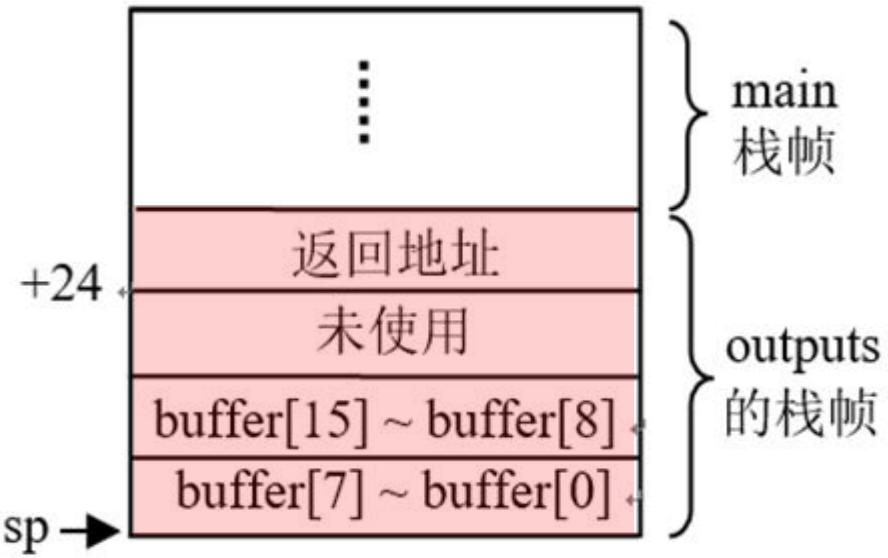
越界访问和缓冲区溢出

hacker首地址为0x1 2000 0738，outputs()的返回地址为0x1 2000 0768

```
void hacker(void) {
    printf("being hacked\n");
}
#include "stdio.h"
char code[]=
"0123456789ABCDEFXXXXXXXX
"\x38\x07\x00\x20\x01\x00\x00\x00 “
"\x00";
int main(void) {
        char *argv[3];
        argv[0]="./test";
        argv[1]=code;
        argv[2]=NULL;
        execve(argv[0],argv,NULL);
        return 0;
}
```

执行上述攻击程序后的输出结果为：

```
"0123456789ABCDEFXXXXXXXX8
being hacked
being hacked
……
```



字符串中前16字符被复制到buffer中，随后8字符X覆盖掉未使用空白区，0x38覆盖掉原返回地址中的0x68，使得本应返回到main执行的返回地址0x1 2000 0768被修改为返回到hacker执行的返回地址0x1 2000 0738。

hacker过程中又把ra保存在其栈帧中，在结束阶段恢复ra，即返回地址依旧为0x1 2000 0738，程序进入死循环，反复调用hacker()函数输出“being hacked”

缓冲区溢出攻击的防范

- 两个方面的防范
  - 从程序员角度去防范
    - 用辅助工具帮助程序员查漏，例如，用grep来搜索源代码中容易产生漏洞的库函数（如strcpy和sprintf等）的调用；用fault injection查错
  - 从编译器和操作系统方面去防范

    （1）地址空间随机化ASLR

    是一种比较有效的防御缓冲区溢出攻击的技术

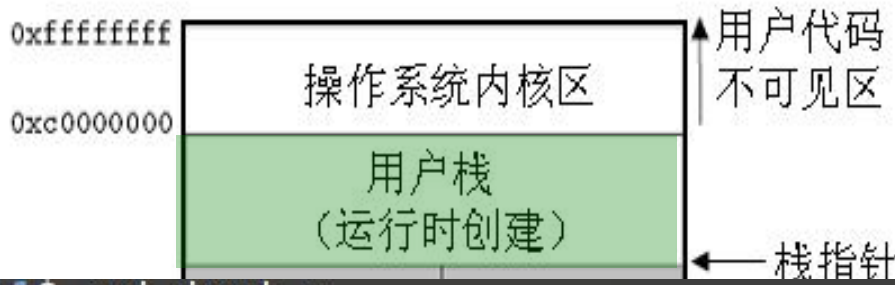
    目前在Linux、FreeBSD和Windows Vista等OS使用

    （2）栈破坏检测

    （3）可执行代码区域限制

    等等

缓冲溢出攻击防范



（1）地址空间随机化

- 只要操作系统相同，则
就一样，若攻击者知道
序使用的栈地址空间，
计一个针对性攻击，在
程序机器上实施攻击
- 地址空间随机化（栈随
的基本思路是，将加载
生成的代码段、静态数
堆区、动态库和栈区各
首地址进行随机化处理，
次启动时，程序各段被
不同地址起始处
- 对于随机生成的栈起始
攻击者不太容易确定栈
位置

```
环境1$ cat test.c
#include <stdio.h>
int main()
{
    int a=10;
    double *p=(double*)&a;
    printf("Scientific: %e\n", *p);
    printf("Machine: %08x %08x\n", *(&a+1), a);
    printf("Address: %p\n\n", &a);
    return 0;
}
环境1$ gcc test.c -o test
环境1$ for i in `seq 3`; do ./test; done
Scientific: -4.083169e-02
Machine: bfa4e7e4 0000000a
Address: 0xbfa4e7e4

Scientific: -1.102164e-02
Machine: bf869284 0000000a
Address: 0xbf869284

Scientific: -3.986657e-02
Machine: bfa46964 0000000a
Address: 0xbfa46964
```

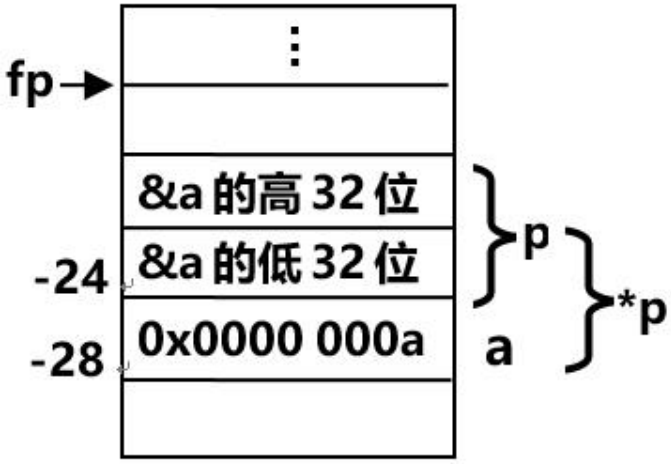
栈地址空间随机化举例

在LA64+Linux系统平台中，以下程序的3次运行结果为：−3.343455e+288、−6.745968e+287、−9.124737e+286，为什么每次不同？

```
#include <stdio.h>
void main() {
    int a=10;
    double *p=(double*)&a;
    printf("%e\n", *p);
}
```

局部变量a和p在栈帧中分别分配在R[fp]−28、R[fp]−24的位置，显然，p在高地址上，a在低地址上，且存储位置相邻。因而*p对应的double型数据就是&a开始的64位数据，*P的高32位就是p值的低32位（即&a的低32位），*P的低32位就是a的值（即0x0000 000a）。

该程序使用double型访问一个int型变量，其行为是未定义的



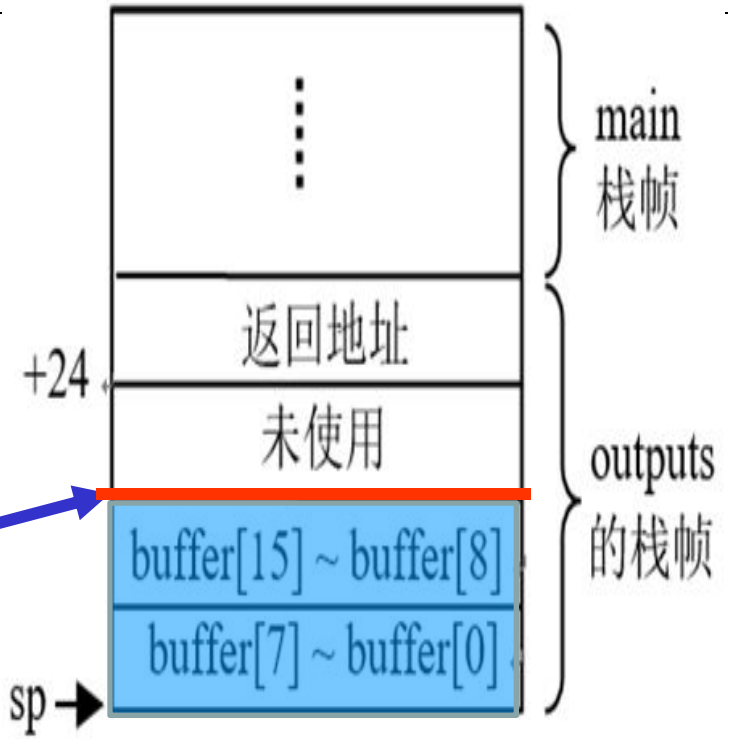
栈随机化策略使a和p所分配地址随机变化，&a的变化使得*p的高32位每次都不同，因而打印结果每次不同。随机变化的地址限定在一定的范围内，因而每次打印的*p的值仅在一定范围内变化。

验证结果如下：机器数0xfbd5 f514 0000 000a的真值为−3.343455e+288；机器数0xfbb1 b894 0000 000a的真值为−6.745968e+287；机器数0xfb83 2d04 0000 000a的真值为−9.124737e+286。因此，对应的&a的低32位分别为0xfbd5 f514、0xfbb1 b894、0xfb83 2d04。

缓冲区溢出攻击的防范

（2）栈破坏检测

- 若在程序跳转到攻击代码前能检测出程序栈已被破坏，就可避免受到严重攻击
- 新GCC版本在代码中加入了一种栈保护者（stack protector）机制，用于检测缓冲区是否越界
- 主要思想：在函数准备阶段，在其栈帧中缓冲区底部（如buffer[15]与未使用空间之间）加入一个随机生成的特定值；在函数恢复阶段，在恢复寄存器并返回到调用函数前，先检查该值是否被改变。若改变则程序异常中止。因为插入在栈帧中的特定值是随机生成的，所以攻击者很难猜测出它是什么



加入一个随机生成的特定值，称为金丝雀值（哨兵）

若不想让GCC插入栈破坏检测代码，则可用命令行选项“–fno–stack–protector”进行编译

栈破坏检测举例

```
#include <stdlib.h>
int main(){
    void*a=alloca(10);
    return 0;
}
```

main()函数准备阶段的机器级代码

```
addi.d      $sp, $sp, -32(0xfe0)
st.d        $ra, $sp, 24(0x18)
st.d        $fp, $sp, 16(0x10)
addi.d      $fp, $sp, 32(0x20)
pcaddu12i   $t0, 8(0x8)
ld.d        $t0, $t0, -1808(0x8f0)
ldptr.d     $t0, $t0, 0
st.d        $t0, $fp, -24(0xfe8)
```

从M[0x120008040]所指存储单元中取出金丝雀值，写入栈中R[fp]−24处，其下方R[fp]−32处是局部变量a的空间，随后就是alloca()所分配的一块缓冲区，若发生缓冲区写溢出，则很有可能会改变写入的金丝雀值。

main()函数结束阶段的机器级代码

```
1  pcaddu12i   $t0, 8(0x8)
2  ld.d        $t0, $t0, -1856(0x8c0)
3  ld.d        $t1, $fp, -24(0xfe8)
4  ldptr.d     $t0, $t0, 0
5  beq         $t1, $t0, 8(0x8)
6  bl          -484(0xffffe1c)
7  addi.d      $sp, $fp, -32(0xfe0)
8  ld.d        $ra, $sp, 24(0x18)
9  ld.d        $fp, $sp, 16(0x10)
10 addi.d      $sp, $sp, 32(0x20)
11 jirl        $zero, $ra, 0
```

第1、2、4行指令从M[0x120008040]所指存储单元中读取原始金丝雀值；第3行指令从R[fp]−24处读取金丝雀值，第5行指令比较两个值是否相等，若不等则执行第6行指令，转检测失败异常处理

缓冲区溢出攻击的防范

（3）可执行代码区域限制

- 通过将程序栈区和堆区设置为不可执行，从而使得攻击者不可能执行被植入在输入缓冲区的代码，这种技术也被称为非执行的缓冲区技术。
- 早期Unix系统只有代码段的访问属性是可执行，其他区域的访问属性是可读或可读可写。但是，近来Unix和Windows系统由于要实现更好的性能和功能，允许在栈段中动态地加入可执行代码，这是缓冲区溢出的根源。
- 为保持程序兼容性，虽然有些非代码段可设置成可执行区域。但是通过将动态的栈段设置为不可执行，既可保证程序的兼容性，又可以有效防止把代码植入栈（自动变量缓冲区）的溢出攻击。