



第4季

# 函数调用规范与栈

# 本节课主要内容

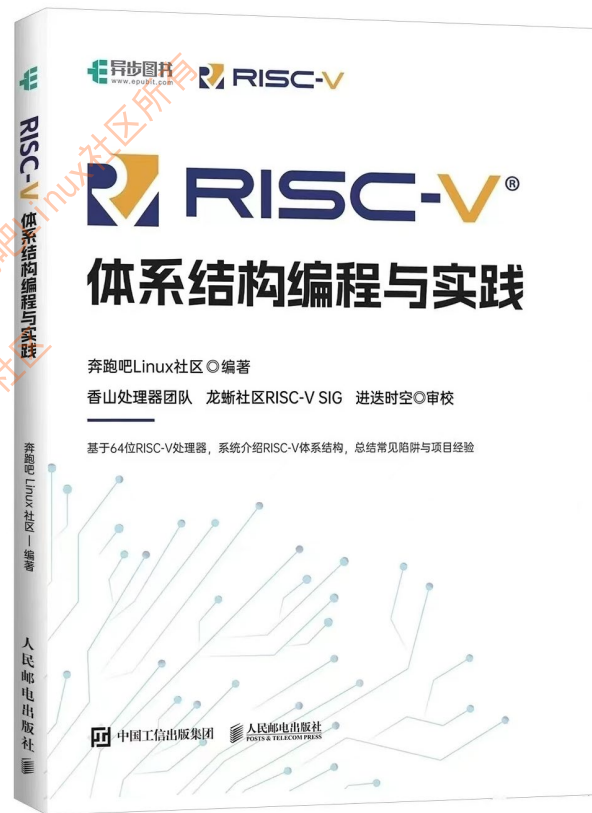
- 本章主要内容
  - 函数调用规范
  - 入栈与出栈
  - RISC-V栈的布局

技术手册：

1. RISC-V ABIs Specification, v0.01



扫码订阅RISC-V视频课程



本节课主要讲解书上第4章内容

# 函数调用规范 (Calling Conventions)

表 4.1 整型通用寄存器函数调用规范

名称	ABI 别名	描述	调用过程中是否需要保存
x0	zero	内容一直为0的寄存器	否
x1	ra	返回地址	否
x2	sp	栈指针	是
x3	gp	全局指针	否
x4	tp	线程指针	否
x5-x7	t0-t2	临时寄存器	否
x8-x9	s0-s1	被调用者需要保存的寄存器	是
x10-x17	a0-a7	用于传递子程序的参数和结果	否
x18-x27	s2-s11	被调用者需要保存的寄存器	是
x28-x31	t3-t6	临时寄存器	否

函数调用规范可以总结如下规则。

- 函数的前8个参数使用a0~a7寄存器来传递。
- 如果函数参数大于8个，后面的参数使用栈来传递。
- 如果传递的参数小于寄存器宽度（64位），那么符号扩展到64位。
- 如果传递的参数为2倍的寄存器宽度（128位），那么将使用一对寄存器来传递该参数。
- 函数的返回参数保存到a0-a1寄存器中。
- 函数的返回地址保存在ra寄存器中。
- 如果子函数里使用s0~s11寄存器，那么子函数在使用前需要把这些寄存器的内容保存到栈中，使用完成之后再从栈中恢复内容到这些寄存器里。
- 栈向下增长(向较低的地址)，栈指针寄存器SP在程序进入时要对齐到16字节边界上。
- 传递给栈的第一个参数位于栈指针寄存器的偏移量0处，后续的参数存储在相应的较高地址处。
- 如果GCC使用了“-fno-omit-frame-pointer”编译选项，那么编译器使用s0作为栈帧指针FP。
- 位域（bitfield）按照小端来排布。它会 填充到下一个 整型类型对齐的地方

```
struct {  
    int x : 10;  
    int y : 12;  
}
```

那么x是bit[9:0]，y是bit[21:10]，剩余  
Bit[31:22]的填充0

```
struct {  
    short x : 10;  
    short y : 12;  
}
```

那么x是bit[9:0]，y是bit[27:16]，剩余  
Bit[15:10]和bit[31:28]的填充0

# 浮点数函数调用规范

整型通用寄存器函数调用规范

名称	ABI 别名	描述	调用过程中是否需要保存
f0-f7	ft0-ft7	临时寄存器	否
f8-f9	fs0-fs1	被调用者需要保存的寄存器	是
f10-f17	fa0-fa7	用于传递子程序的参数和结果	否
f18-f27	fs2-fs11	被调用者需要保存的寄存器	是
f28-f31	ft8-ft11	临时寄存器	否

21

# 例子

使用汇编语言来实现下面的C语言程序。  
见《RISC-V体系结构编程与实践》例4-1。

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 2, c = 3, d = 4, e = 5, f = 6, g = 7, h = 8, i = 9, j = -1;

    printf("data: %d %d %d %d %d %d %d %d %d %d\n",
           a, b, c, d, e, f, g, h, i, j);

    return 0;
}
```

使用printf()函数来打印10个参数的值

```

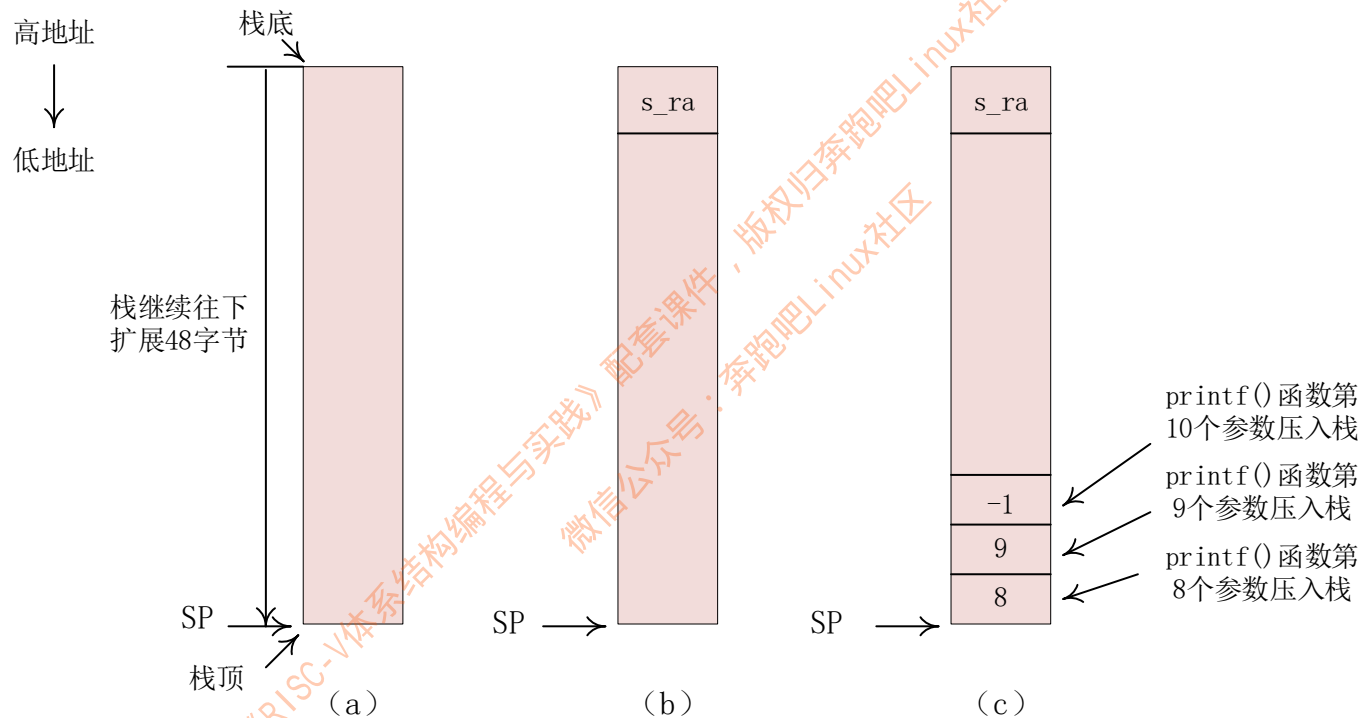
<test.S>
1      .section    .rodata
2      .align 3
3      .string:
4      .string "data: %d %d %d %d %d %d %d %d %d %d\n"
5
6      data:
7      .word 1, 2, 3, 4, 5, 6, 7, 8, 9, -1
8
9      .text
10     .align 2
11
12     .global main
13     main:
14         /*栈往下扩展48字节*/
15         addi    sp,sp,-48
16
17         /*保存main函数的返回地址ra到栈里*/
18         sd      ra,40(sp)
19
20         /* a0传递第一个参数: .string字符串 */
21         la      a0,.string
22
23         /* a1 - a7 传递printf()前7个参数 */
24         li      a1,1
25         li      a2,2
26         li      a3,3
27         li      a4,4
28         li      a5,5
29         li      a6,6
30         li      a7,7

```

```

31
32     /* printf()第8-10个参数通过栈来传递 */
33     li      t0,8
34     sd      t0,0(sp)
35     li      t0,9
36     sd      t0,8(sp)
37     li      t0,-1
38     sd      t0,16(sp)
39
40     /* 调printf()函数 */
41     call    printf
42
43     /* 从栈中恢复ra返回地址 */
44     ld      ra,40(sp)
45     /* 设置main函数返回值为0 */
46     li      a0,0
47     /* SP回到原点 */
48     addi    sp,sp,48
49     ret

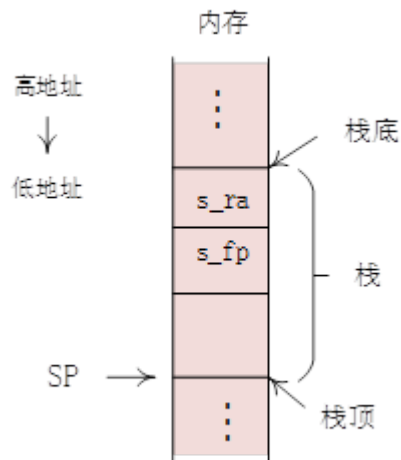
```





# 栈 (stack)

- 栈 (stack) 是一种后进先出的数据存储结构。
  - 临时存储的数据，例如局部变量等。
  - 当参数大于8个时，则需要使用栈来传递参数。
- 一种从高地址往低地址扩展（生长）的数据存储结构
- 栈底：栈的起始地址
- 栈顶：栈从高地址往低地址延伸到某个点
- SP：通常指向栈顶
- 压栈 (push)：把数据往栈里存储
- 出栈 (pop)：从栈中移出数据
- 栈帧

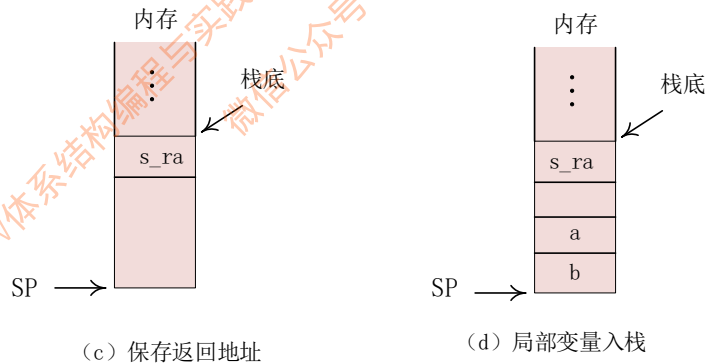
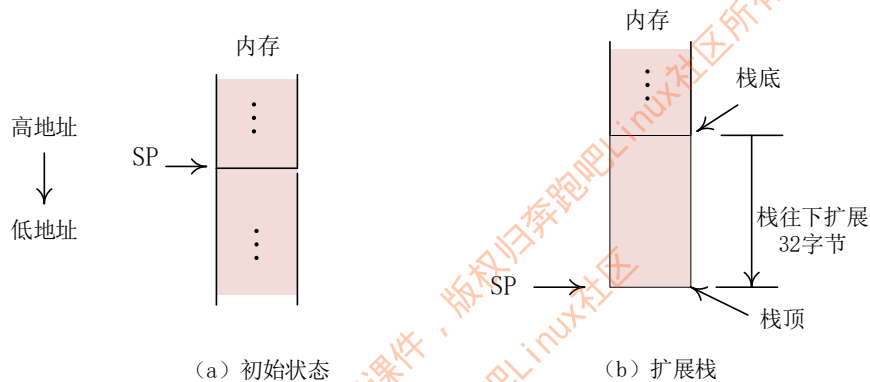


# 例子

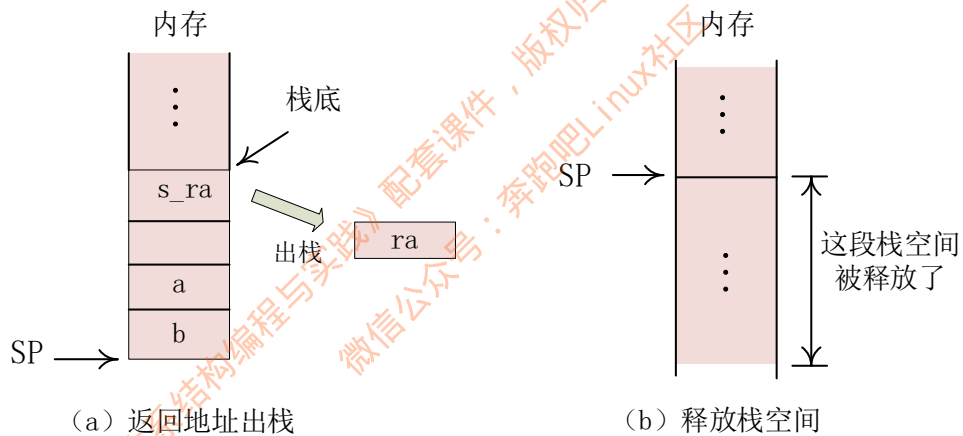
```
1  func1:↵
2      /* 栈往下扩展32字节*/↵
3      addi    sp, sp, -32↵
4      /* 把返回地址ra存储到SP+24*/↵
5      sd     ra, 24(sp)↵
6      ↵
7      /* 把局部变量存储到栈里*/↵
8      li     a5, 1↵
9      sd     a5, 8(sp)↵
10     li     a5, 2↵
11     sd     a5, 0(sp)↵
12     ↵
13     /* 从栈里取出局部变量*/↵
14     ld     a1, 0(sp)↵
15     ld     a0, 8(sp)↵
16     /* 调子函数*/↵
17     call   add_c↵
18     ↵
19     /* 从栈里恢复返回地址ra*/↵
20     ld     ra, 24(sp)↵
21     /* 释放栈, SP回到原点*/↵
22     addi    sp, sp, 32↵
23     /* 返回*/↵
24     ret↵
```

见《RISC-V体系结构编程与实践》例4-2.

# 入栈



# 出栈



# 栈的布局 – 不使用FP的情况

GCC使用了“-fomit-frame-pointer”编译选项

在BenOS中, kernel\_main()调用子函数func1(), 然后在func1()函数中继续调用add\_c()函数。

首先在boot.S文件中分配栈空间, 假设栈指针寄存器SP为0x80203000, 然后跳转到C语言的kernel\_main()函数中。

```
<boot.S>
1  .section ".text.boot"
2  .globl _start
3  _start:
4      /* 分配栈空间, 设置SP */
5      la sp, stacks_start
6      li t0, 4096
7      add sp, sp, t0
8
9      tail kernel_main
10
11 .section .data
12 .align 12
13 .global stacks_start
14 stacks_start:
15     .skip 4096
```

```
<kernel.c>
1  int add_c(int a, int b)
2  {
3      return a + b;
4  }
5
6  int func1(void)
7  {
8      int a = 1;
9      int b = 2;
10
11     return add_c(a, b);
12 }
13
14 void kernel_main(void)
15 {
16     func1();
17 }
```

# 调用kernel\_main函数

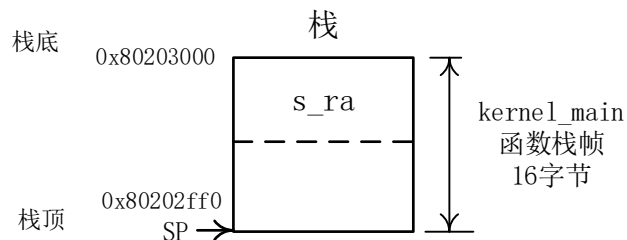
<kernel\_main()函数的反汇编>

←

(gdb) disassemble ←

Dump of assembler code for function kernel\_main:←

```
0x00000000080200160 <+0>:   addi    sp,sp,-16←  
0x00000000080200162 <+2>:   sd      ra,8(sp)←  
=>0x00000000080200164 <+4>:   jal     ra,0x8020013e <func1>←  
0x00000000080200168 <+8>:   nop←  
0x0000000008020016a <+10>:  ld      ra,8(sp)←  
0x0000000008020016c <+12>:  addi    sp,sp,16←  
0x0000000008020016e <+14>:  ret←
```



# 调用func1函数

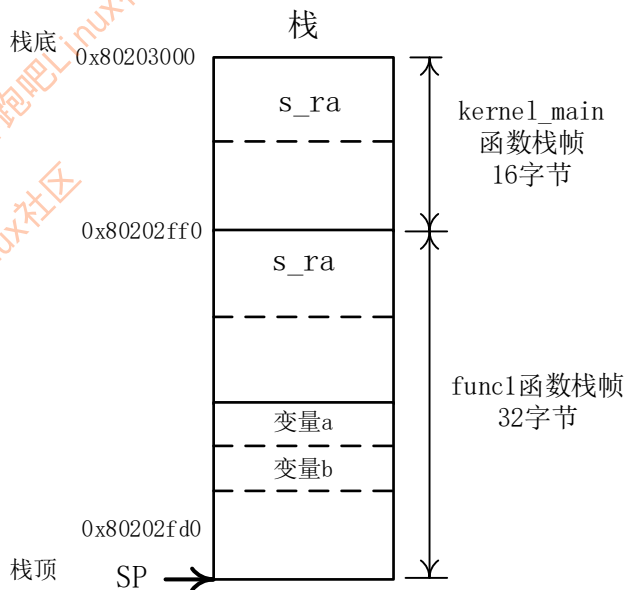
<func1()函数的反汇编>

```
(gdb) disassemble  
Dump of assembler code for function func1:  
0x0000000008020013e <+0>:      addi    sp,sp,-32  
0x00000000080200140 <+2>:      sd      ra,24(sp)  
=> 0x00000000080200142 <+4>:      li      a5,1  
0x00000000080200144 <+6>:      sw      a5,12(sp)  
0x00000000080200146 <+8>:      li      a5,2  
0x00000000080200148 <+10>:     sw      a5,8(sp)  
0x0000000008020014a <+12>:     lw      a4,8(sp)  
0x0000000008020014c <+14>:     lw      a5,12(sp)  
0x0000000008020014e <+16>:     mv      a1,a4  
0x00000000080200150 <+18>:     mv      a0,a5  
0x00000000080200152 <+20>:     jal     ra,0x80200124 <add_c>  
0x00000000080200156 <+24>:     mv      a5,a0  
0x00000000080200158 <+26>:     mv      a0,a5  
0x0000000008020015a <+28>:     ld      ra,24(sp)  
0x0000000008020015c <+30>:     addi    sp,32  
0x0000000008020015e <+32>:     ret
```

入栈操作



出栈操作



# 调用add\_c函数

<add\_c()函数的反汇编>

←

(gdb) disassemble ←

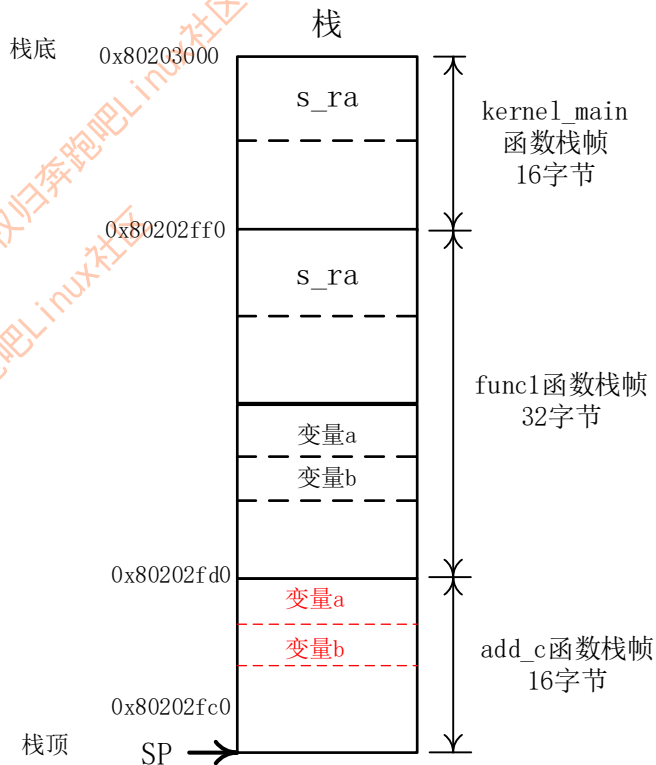
Dump of assembler code for function add\_c:←

```
0x00000000080200124 <+0>:      addi    sp,sp,-16←
0x00000000080200126 <+2>:      mv       a5,a0←
0x00000000080200128 <+4>:      mv       a4,a1←
0x0000000008020012a <+6>:      sw       a5,12(sp)←
0x0000000008020012c <+8>:      mv       a5,a4←
0x0000000008020012e <+10>:     sw       a5,8(sp)←
=> 0x00000000080200130 <+12>:    lw       a4,12(sp)←
0x00000000080200132 <+14>:    lw       a5,8(sp)←
0x00000000080200134 <+16>:    addw     a5,a5,a4←
0x00000000080200136 <+18>:    sext.w   a5,a5←
0x00000000080200138 <+20>:    mv       a0,a5←
0x0000000008020013a <+22>:    addi     sp,sp,16←
0x0000000008020013c <+24>:    ret←
```

入栈操作



↑  
出栈操作





# 总结 - 栈的布局 – 不使用FP的情况

RISC-V的函数栈布局的关键点如下。

- ✓ 所有的函数调用栈是从高地址向低地址扩展。
- ✓ 栈指针sp指向栈顶（栈的最低地址处）。
- ✓ 函数的返回地址（如果调用了子函数）需要保存到栈里，即s\_ra位置处。
- ✓ 栈的大小为16字节的倍数。
- ✓ 函数返回时需要先把返回地址从栈（s\_ra位置处）中恢复到ra寄存器，然后执行ret指令。

# 栈的布局 – 使用FP的情况

GCC使用了“-fno-omit-frame-pointer”编译选项

在BenOS中, kernel\_main()调用子函数func1(), 然后在func1()函数中继续调用add\_c()函数。

首先在boot.S文件中分配栈空间, 假设栈指针寄存器SP为0x80203000, 然后跳转到C语言的kernel\_main()函数中。

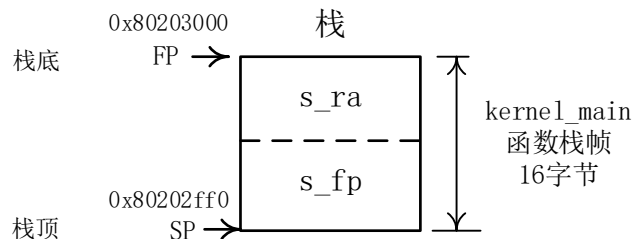
```
<boot.S>
1      .section ".text.boot"
2      .globl _start
3      _start:
4          /* 分配栈空间, 设置SP */
5          la sp, stacks_start
6          li t0, 4096
7          add sp, sp, t0
8
9          tail    kernel_main
10
11     .section .data
12     .align 12
13     .global stacks_start
14     stacks_start:
15         .skip 4096
```

```
<kernel.c>
1      int add_c(int a, int b)
2      {
3          return a + b;
4      }
5
6      int func1(void)
7      {
8          int a = 1;
9          int b = 2;
10
11          return add_c(a, b);
12      }
13
14     void kernel_main(void)
15     {
16         func1();
17     }
```

# 调用kernel\_main函数

入栈操作

```
(gdb) disassemble
Dump of assembler code for function kernel_main:
0x00000000802001a4 <+0>:   addi    sp,sp,-16
0x00000000802001a6 <+2>:   sd      ra,8(sp)
0x00000000802001a8 <+4>:   sd      s0,0(sp)
0x00000000802001aa <+6>:   addi    s0,sp,16
=> 0x00000000802001ac <+8>:   jal     ra,0x80200174 <func1>
0x00000000802001b0 <+12>:  nop
0x00000000802001b2 <+14>:  ld      ra,8(sp)
0x00000000802001b4 <+16>:  ld      s0,0(sp)
0x00000000802001b6 <+18>:  addi    sp,sp,16
0x00000000802001b8 <+20>:  ret
```



# 调用func1函数

<func1()函数的反汇编>

←

Dump of assembler code for function func1:

```
0x00000000080200174 <+0>: addi sp,sp,-32
0x00000000080200176 <+2>: sd ra,24(sp)
0x00000000080200178 <+4>: sd s0,16(sp)
0x0000000008020017a <+6>: addi s0,sp,32
=> 0x0000000008020017c <+8>: li a5,1
0x0000000008020017e <+10>: sw a5,-20(s0)
0x00000000080200182 <+14>: li a5,2
0x00000000080200184 <+16>: sw a5,-24(s0)
0x00000000080200188 <+20>: lw a4,-24(s0)
0x0000000008020018c <+24>: lw a5,-20(s0)
0x00000000080200190 <+28>: mv a1,a4
0x00000000080200192 <+30>: mv a0,a5
0x00000000080200194 <+32>: jal ra,0x8020014c <add_c>
0x00000000080200198 <+36>: mv a5,a0
0x0000000008020019a <+38>: mv a0,a5
0x0000000008020019c <+40>: ld ra,24(sp)
0x0000000008020019e <+42>: ld s0,16(sp)
0x000000000802001a0 <+44>: addi sp,sp,32
0x000000000802001a2 <+46>: ret
```

入栈操作

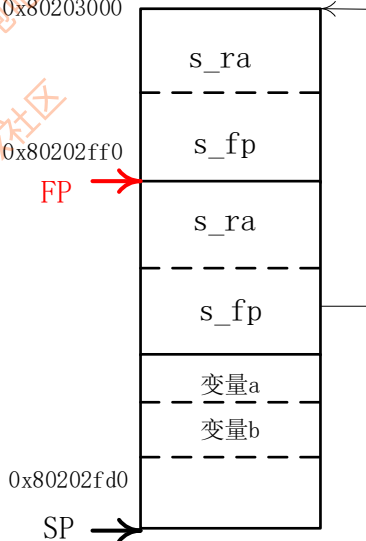


出栈操作



栈底 0x80203000

栈



kernel\_main  
函数栈帧  
16字节

func1函数栈帧  
32字节

# 调用add\_c函数

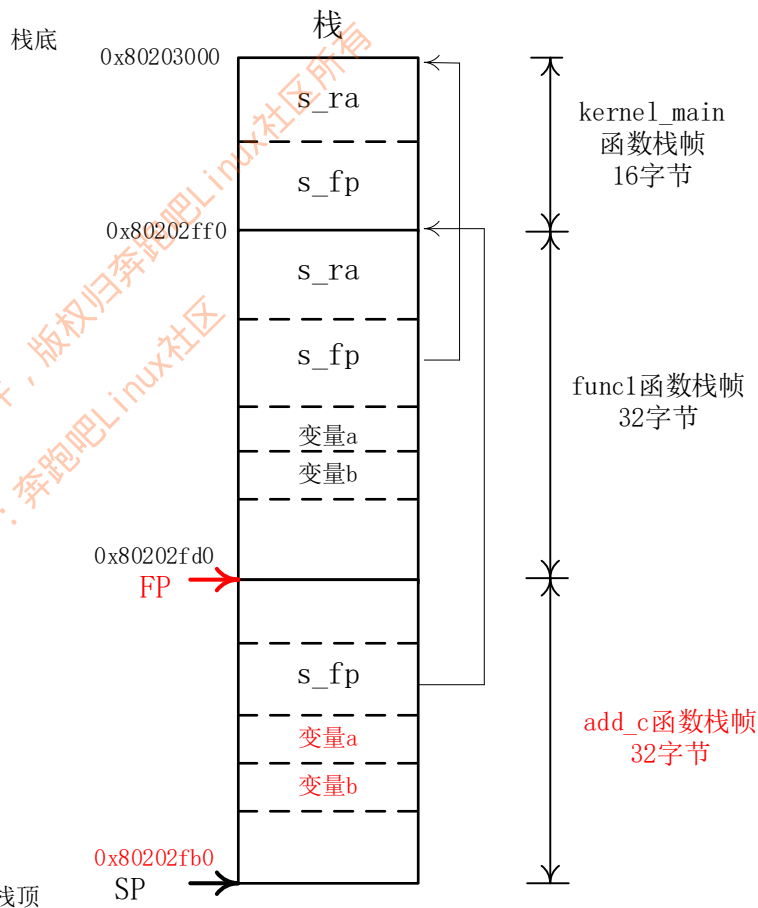
入栈操作

<add\_c()函数的反汇编>

Dump of assembler code for function add\_c:

```
0x0000000008020014c <+0>: addi    sp,sp,-32
0x0000000008020014e <+2>: sd     s0,24(sp)
0x00000000080200150 <+4>: addi    s0,sp,32
0x00000000080200152 <+6>: mv     a5,a0
0x00000000080200154 <+8>: mv     a4,a1
0x00000000080200156 <+10>: sw     a5,-24(s0)
0x0000000008020015a <+14>: mv     a5,a4
0x0000000008020015c <+16>: sw     a5,-24(s0)
=> 0x00000000080200160 <+20>: lw     a4,-20(s0)
0x00000000080200164 <+24>: lw     a5,-24(s0)
0x00000000080200168 <+28>: addw    a5,a5,a4
0x0000000008020016a <+30>: sext.w  a5,a5
0x0000000008020016c <+32>: mv     a0,a5
0x0000000008020016e <+34>: ld     s0,24(sp)
0x00000000080200170 <+36>: addi    sp,sp,32
0x00000000080200172 <+38>: ret
```

出栈操作



# 总结 – 使用FP的情况

- 所有的函数调用栈都会组成一个单链表。
- 每个栈由两个地址来构成这个链表，这两个地址都是64位宽的，并且它们都位于栈底。
  - ✓ `s_fp`的值指向上一个栈帧（父函数的栈帧）的栈底。
  - ✓ `s_ra`保存当前函数的返回地址，也就是父函数调用该函数时的地址。
- 函数返回时，RISC-V处理器先把返回地址从栈的`s_ra`位置处载入到当前`ra`寄存器，然后执行`ret`指令。

《RISC-V体系结构编程与实践》配套课程：奔跑吧Linux社区所有  
微信公众号：奔跑吧Linux社区

# 栈的回溯

操作系统常用的输出栈信息等技术手段是通过栈帧指针FP来回溯整个栈，例如

```
Oops - Store/AMO page fault
Call Trace:
[<0x0000000080202edc>] test_access_unmap_address+0x1c/0x42
[<0x0000000080202f12>] test_mmu+0x10/0x1a
[<0x000000008020329a>] kernel_main+0xb4/0xb6
..
```

# 例子

见《RISC-V体系结构编程与实践》例4-4.

输出每个栈的范围，以及调用该函数时候的PC值，如下面的日志信息。

```
Call Frame:↵  
[0x00000000080202fa0 - 0x00000000080202fb0] pc 0x00000000080200f32↵  
[0x00000000080202fb0 - 0x00000000080202fd0] pc 0x0000000008020114a↵  
[0x00000000080202fd0 - 0x00000000080202ff0] pc 0x00000000080201184↵  
[0x00000000080202ff0 - 0x00000000080203000] pc 0x000000000802011a4↵
```



```

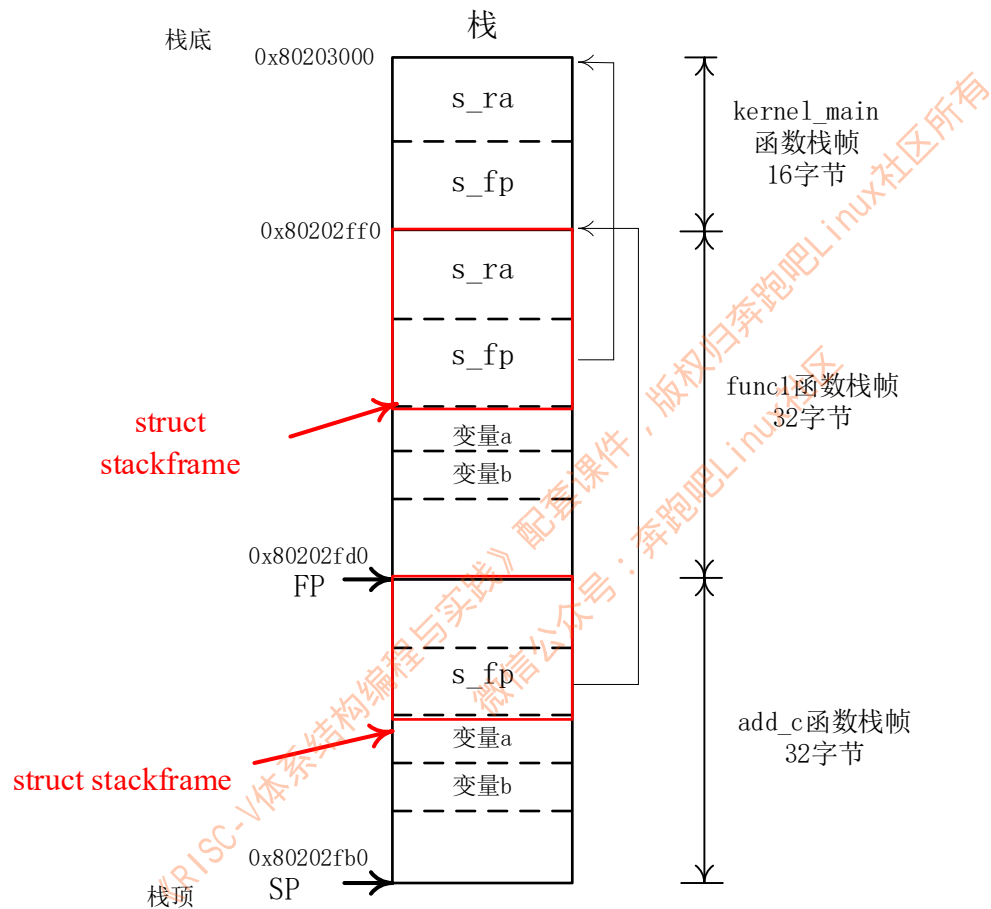
<stacktrace.c>
1 struct stackframe {
2     unsigned long s_fp;
3     unsigned long s_ra;
4 };
5
6 extern char _text[], _etext[];
7 static int kernel_text(unsigned long addr)
8 {
9     if (addr >= (unsigned long)_text &&
10        addr < (unsigned long)_etext)
11         return 1;
12
13     return 0;
14 }
15
16 static void walk_stackframe(void)
17 {
18     unsigned long sp, fp, pc;
19     struct stackframe *frame;
20     unsigned long low;
21
22     const register unsigned long current_sp __asm__ ("sp");
23     sp = current_sp;
24     pc = (unsigned long)walk_stackframe;
25     fp = (unsigned long)__builtin_frame_address(0);
26
27     while (1) {
28         if (!kernel_text(pc))
29             break;
30
31         /* 检查fp是否有效 */
32         low = sp + sizeof(struct stackframe);
33         if ((fp < low || fp & 0xf))
34             break;

```

```

35
36         /*
37          * fp 指向上一级函数的栈底
38          * 减去16个字节, 正好是struct stackframe
39          */
40         frame = (struct stackframe *) (fp - 16);
41         sp = fp;
42         fp = frame->s_fp;
43
44         pc = frame->s_ra - 4;
45
46         if (kernel_text(pc))
47             printk("[0x%016lx - 0x%016lx] pc 0x%016lx\n", sp, fp, pc);
48     }
49 }
50
51 void dump_stack(void)
52 {
53     printk("Call Frame:\n");
54     walk_stackframe();
55 }

```



# 实验1：观察栈布局

## 1. 实验目的

熟悉RISC-V的栈布局。

## 2. 实验目的

在BenOS里实现如下函数调用：`kernel_main()`→`func1()`→`func2()`。假设`func2()`的参数大于8个。然后使用GDB来观察栈的变化情况，并画出栈布局图。

# 实验2：观察栈回溯

## 1. 实验目的

熟悉RISC-V的栈布局。

## 2. 实验目的

在BenOS里实现如下函数调用：kernel\_main()→func1()→func2()，并实现一个栈回溯功能，打印栈的地址范围和大小，并通过GDB来观察栈是如何回溯的。

# 文字不如声音，声音不如视频



扫描订阅RISC-V视频课程



笨叔（老笨）  
邀请你一起学习

奔跑吧Linux社区  
RISC-V体系结构编程与实践

主讲：笨叔

第4季 RISC-V体系结构编程与实践  
¥299.00 999.00



长按扫码查看详情

小鹏通提供技术支持

第4季 奔跑吧Linux社区 视频课程

RISC-V体系结构编程与实践

主讲：笨叔

课程名称	进度	时长 (分钟)
第1课：课程介绍 (免费)	完成	20
第2课：RISC-V体系结构介绍 (免费)	完成	47
第3课：RISC-V处理器微架构 (免费)	完成	48
第4课：搭建RISC-V开发环境 (免费)	完成	30
第5课：RISC-V指令集 (免费)	完成	128
第6课：RISC-V函数调用规范	完成	40
第7课：RISC-V GNU AS汇编器	完成	42
第8课：RISC-V GNU 链接脚本	完成	90
第9课：RISC-V GNU 内嵌汇编	完成	52
第10课：RISC-V异常处理	完成	80
第11课：RISC-V中断处理	完成	52
第12课：RISC-V内存管理	完成	116
第13课：内存管理实验讲解	完成	36
第14课：cache基础知识	完成	78
第15课：缓存一致性	完成	96
第16课：RISC-V TLB管理	完成	54
第17课：RISC-V原子操作	未录制	
第18课：RISC-V内存屏障	未录制	
第19课：BenOS操作系统相关知识	未录制	
第20课：RVV可伸缩向量计算	未录制	
第21课：RISC-V压缩指令	未录制	
第22课：RISC-V虚拟化	未录制	
		总计17小时

更多精彩内容马上献上....

微信公众号：奔跑吧Linux社区

视频课程持续更新中...