计算机体系结构基础

胡伟武、苏孟豪

第04章 软硬件协同

- 应用程序二进制接口
- 中断的生命周期
- 系统调用过程
- 同步与通讯

应用程序二进制接口（ABI）

为什么要有ABI

- 计算机是个复杂系统
  - 多层次的软件：引导程序、系统内核、动态链接库、应用...
  - 各种开发工具：汇编、C、C++
  - 对于如何在指令系统基础上构建软件系统，必须要有一些规范
- Application Binary Interface (ABI)
- 规范些什么？
  - 寄存器使用
  - 函数调用
  - 栈布局

MIPS ABI

- O32
  - 传统的MIPS约定，仍广泛用于嵌入式工具链和32位Linux中
- N64
  - 在64位处理器编程使用的新的正式ABI，改变了指针和long型整数的宽度，并改变了寄存器使用的约定和参数传递的方式
- N32
  - 在64位处理器上执行的32位程序；与N64的区别在于指针和long型整数的宽度为32位；与O32相比，可以使用64位寄存器，浮点寄存器数量从16个变为32个

MIPS ABI

```
{
    printf("%d %d %d %d",
            sizeof(int),
            sizeof(long),
            sizeof(long long),
            sizeof(int*));
}
```

- O32

  4 4 8 4
- N64

  4 8 8 8
- N32

  4 4 8 4

X86 ABI

- i386
  - 1985年Intel 80386开始使用的32位ABI，又称IA-32 (Intel Architecture 32bit)
- x86-64
  - 64位版本
- x32
  - 在64位处理器上执行的32位程序，类似MIPS n32
  - 多数SPEC定点测试提升6%

Feature	i386	x32	x86-64
Pointers	4 bytes		8 bytes
Max. memory per process	4 GiB		128 TiB
Integer registers	6 (PIC)	15	
FP registers	8	16	
64-bit arithmetic	No	Yes	
Floating point arithmetic	x87	SSE	
Calling convention	Memory	Registers	
PIC prologue	2 - 3 instructions	None	

MIPS ABI寄存器使用

- MIPS ABI有以下约定
  - a*：函数参数
  - v*：函数返回值
  - t*：临时变量，子函数可改
  - s*：子函数不改的变量
  - k*：内核保留
  - gp：全局指针
  - sp：栈顶指针
  - fp：栈帧指针
  - ra：返回地址

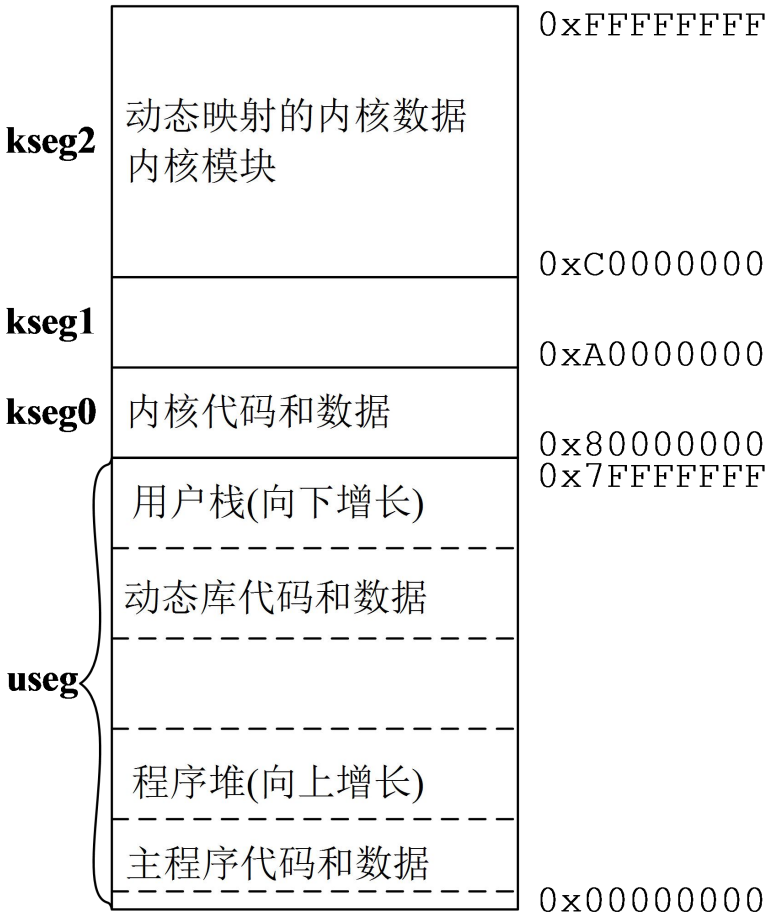
寄存器号	o32	n32	Saver
0	zero		-
1	at		er
2-3	v0,v1		er
4-7	a0-a3		er
8-11	t0-t3	a4-a7	er
12-15	t4-t7	t0-t3	er
16-23	s0-s7		ee
24-25	t8,t9/jp		er
26-27	k0,k1		-
28	gp		er
29	sp		ee
30	s8/fp		ee
31	ra		ee

MIPS ABI函数调用(N32)

- 定点参数经扩展后放到a0-a7，超过8个的通过栈传递
- 浮点参数通过浮点寄存器传递，定点+浮点共8个
- 结构体视为双字序列，通过寄存器传递
- 超过两个双字的返回值，用指针(a0)通过栈传递
- 在栈中预留8个参数槽，对应8个参数，在必要的时候用于参数存储，其宽度固定为64位(N32)
- 栈16字节对齐，指针增量为16的倍数

MIPS虚存空间

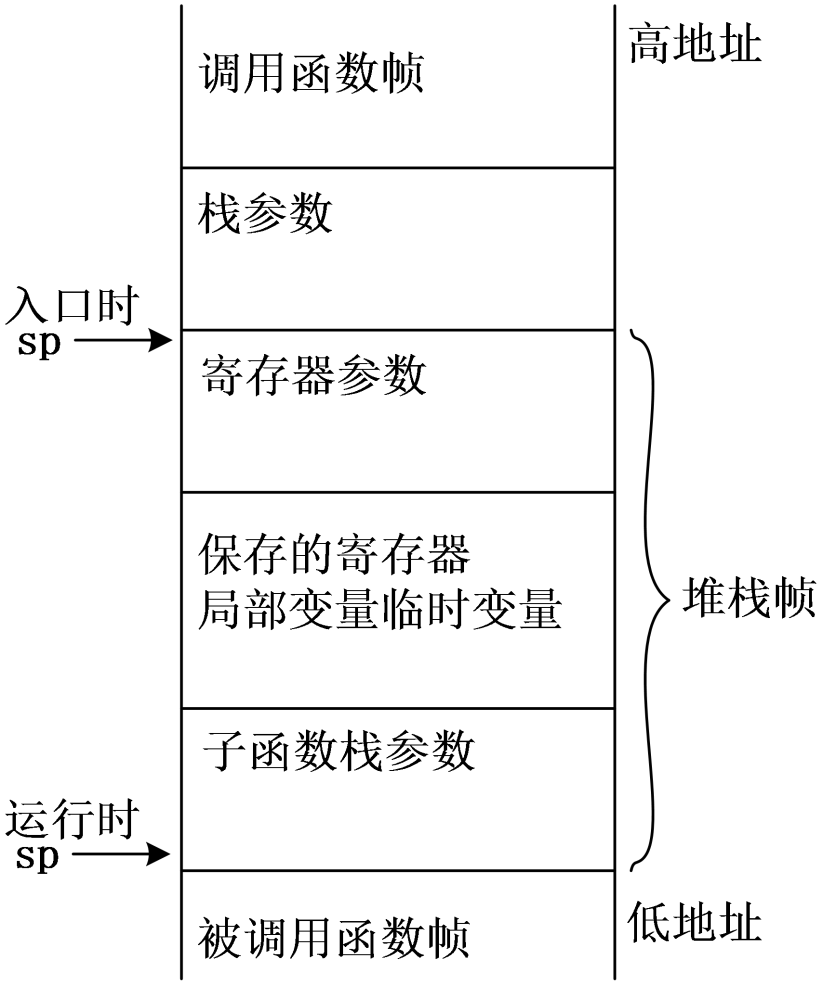
- Linux线程空间划分(32位)
  - 线程可用空间2GB
  - 主程序和动态库的代码与数据位于底端
  - 堆空间自底向上增长，存放动态分配的数据
  - 用户栈自顶向下增长，存放函数中的临时变量和数据
  - 动态链接库放在栈的下方



思考：不用地址0有什么好处？

MIPS ABI栈布局

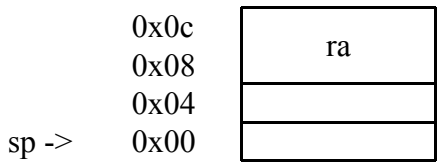
- 栈——代码的例化
  - 保存上下文
  - 传递参数
  - 保存临时变量，非静态局部变量
- 使用分类
  - 简单叶子函数：可以不用栈
  - 编译时可确定栈大小的：只用sp
  - 运行时确定栈大小的：fp(s8)+sp
- 栈帧构成
  - 参数区+临时变量区+子函数参数
  - 编译器优化后不一定保留参数区



MIPS ABI示例

----3个定点参数(-O1)

```
int add_3(int in0, int in1, int in2)
{
    return in0 + in1 + in2;
}
//参数个数小于8，使用寄存器a0~a2传递
//参数用法简单，不需要保存到栈
//叶子函数无需保存返回地址
//返回值用v0传递

int call_add_3()
{
    return add_3(1, 2, 3);
}
//栈16字节对齐，只用 8 字节也留16字节
//64位寄存器，ra指针当作8字节保存
```




```
00000000 <add_3>://叶子函数
   0:        00851021    addu   v0,a0,a1
   4:        03e00008    jr     ra
   8:        00461021    addu   v0,v0,a2


0000000c <call_add_3>://非叶子函数
   c:        27bdfff0    addiu  sp,sp,-16
  10:        ffbf0008    sd     ra,8(sp)
  14:        24040001    li     a0,1
  18:        24050002    li     a1,2
  1c:        0c000000    jal    0 <add_3>
  20:        24060003    li     a2,3
  24:        dfbf0008    ld     ra,8(sp)
  28:        03e00008    jr     ra
  2c:        27bd0010    addiu  sp,sp,16
```

gcc (GCC) 4.9.3 20150626 (Red Hat 4.9.3-7)

gcc -mabi=n32 -O1 -fno-inline -fno-pic -mno-abicalls -c call.c

objdump -d call.o

MIPS ABI示例

----3个定点参数(-O0)

```
int add_3(int in0, int in1, int in2)
{
    return in0 + in1 + in2;
}
//参数个数小于8，使用寄存器传递
//未作优化，先在栈中保存参数，后读出使用（非必要）
//未作优化，叶子函数也保存返回地址到栈（非必要）
```

```
00000000 <add_3>:
   0:   27bdffe0        addiu   sp,sp,-32
   4:   ffbe0018        sd      s8,24(sp)
   8:   03a0f02d        move    s8,sp
   c:   afc40000        sw      a0,0(s8)
  10:   afc50004        sw      a1,4(s8)
  14:   afc60008        sw      a2,8(s8)
  18:   8fc30000        lw      v1,0(s8)
  1c:   8fc20004        lw      v0,4(s8)
  20:   00621821        addu    v1,v1,v0
  24:   8fc20008        lw      v0,8(s8)
  28:   00621021        addu    v0,v1,v0
  2c:   03c0e82d        move    sp,s8
  30:   dfbe0018        ld      s8,24(sp)
  34:   27bd0020        addiu   sp,sp,32
  38:   03e00008        jr      ra
  3c:   00000000        nop
```

	0x1c	s8
	0x18	
	0x14	-
	0x10	-
	0x0c	-
	0x08	a2
	0x04	a1
sp/s8->	0x00	a0

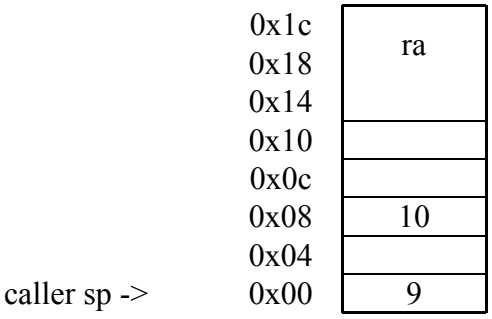
gcc (GCC) 4.9.3 20150626 (Red Hat 4.9.3-7)

gcc -mabi=n32 -O0 -fno-inline -fno-pic -mno-abicalls -c call.c

objdump -d call.o

MIPS ABI示例

```
int add_10(int i0, int i1, int i2, int i3,
           int i4, int i5, int i6, int i7,
           int i8, int i9)
{
    return i0 + i1 + i2 + i3 + i4 +
           i5 + i6 + i7 + i8 + i9;
}

int call_add_10()
{
    return add_10(1,2,3,4,5,6,7,8,9,10);
}

//第 8 个参数后使用栈传递
//
```



----10个定点参数

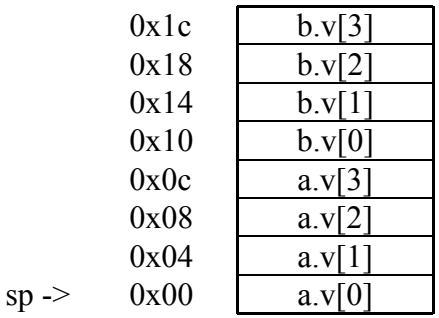
```
000003a4 <add_10>:
 3a4:   00852021        addu    a0,a0,a1
 3a8:   00863021        addu    a2,a0,a2
 3ac:   00c73821        addu    a3,a2,a3
 3b0:   00e84021        addu    a4,a3,a4
 3b4:   01094821        addu    a5,a4,a5
 3b8:   012a5021        addu    a6,a5,a6
 3bc:   014b5821        addu    a7,a6,a7
 3c0:   8fa20000        lw      v0,0(sp)
 3c4:   01623021        addu    a2,a7,v0
 3c8:   8fa20008        lw      v0,8(sp)
 3cc:   03e00008        jr      ra
 3d0:   00c21021        addu    v0,a2,v0
000003d4 <call_add_10>:
 3d4:   27bdffe0        addiu   sp,sp,-32
 3d8:   ffbf0018        sd      ra,24(sp)
 3dc:   24020009        li      v0,9
 3e0:   afa20000        sw      v0,0(sp)
 3e4:   2402000a        li      v0,10
 3e8:   afa20008        sw      v0,8(sp)
 3ec:   24040001        li      a0,1
 ...    ...             ...     ...
 404:   240a0007        li      a6,7
 408:   0c000000        jal     <add_10>
 40c:   240b0008        li      a7,8
 410:   dfbf0018        ld      ra,24(sp)
 414:   03e00008        jr      ra
 418:   27bd0020        addiu   sp,sp,32
```

MIPS ABI示例

----参数为结构体

```
typedef struct vint4 { int v[4];} vint4_t;

void add_vint4_2_s(vint4_t a, vint4_t b,
                   vint4_t* s)
{
    int i;
    for (i=0; i<4; i++) {
        s->v[i] = a.v[i] + b.v[i];
    }
}
//a0,a1打包存放参数vint4_t a
//a2,a3打包存放参数vint4_t b
//a4为参数s，指向返回结构体
//循环变量i没有存储，用指针判断结束
```

```
000000f0 <add_vint4_2_s>:
  f0:   27bdffe0        addiu   sp,sp,-32
  f4:   ffa40000        sd      a0,0(sp)
  f8:   ffa50008        sd      a1,8(sp)
  fc:   ffa60010        sd      a2,16(sp)
 100:   ffa70018        sd      a3,24(sp)
 104:   03a0102d        move    v0,sp
 108:   27a40010        addiu   a0,sp,16
 10c:   0080302d        move    a2,a0

 110:   8c430000        lw      v1,0(v0)
 114:   8c850000        lw      a1,0(a0)
 118:   00651821        addu    v1,v1,a1
 11c:   ad030000        sw      v1,0(a4)
 120:   24420004        addiu   v0,v0,4
 124:   24840004        addiu   a0,a0,4
 128:   1446fff9        bne     v0,a2,110
 12c:   25080004        addiu   a4,a4,4

 130:   03e00008        jr      ra
 134:   27bd0020        addiu   sp,sp,32
```



MIPS ABI示例

----小结构体返回

```
typedef struct vint4 { int v[4];} vint4_t;

vint4_t get_vint4(int v0, int v1,
                  int v2, int v3)
{
    vint4_t s;
    s.v[0] = v0;
    s.v[1] = v1;
    s.v[2] = v2;
    s.v[3] = v3;
    return s;
}
//v0打包存放返回值的v[0],v[1]
//v1打包存放返回值的v[2],v[3]

//dext rt, rs, pos, size
// rt[size-1:0] := rs[pos+size-1 : pos]

//dins rt, rs, pos, size
// rt[size+pos-1 : pos] := rs[size-1:0]
```

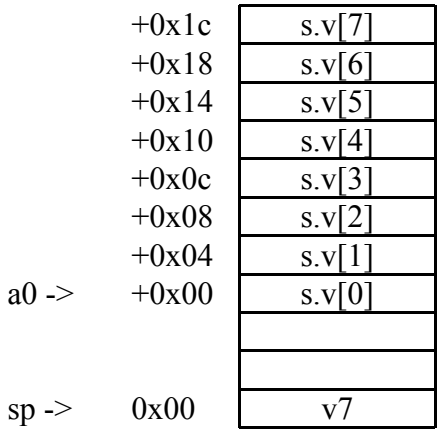
```
0000006c <get_vint4>:
  6c:   27bdfff0        addiu   sp,sp,-16

  70:   0000102d        move    v0,zero
  74:   7c84f803        dext    a0,a0,0x0,0x20
  78:   7c82f807        dins    v0,a0,0x0,0x20
  7c:   7ca5f803        dext    a1,a1,0x0,0x20
  80:   7ca2f806        dins    v0,a1,0x20,0x20

  84:   0000182d        move    v1,zero
  88:   7cc6f803        dext    a2,a2,0x0,0x20
  8c:   7cc3f807        dins    v1,a2,0x0,0x20
  90:   7ce7f803        dext    a3,a3,0x0,0x20
  94:   7ce3f806        dins    v1,a3,0x20,0x20

  98:   03e00008        jr      ra
  9c:   27bd0010        addiu   sp,sp,16
```

MIPS ABI示例

----大结构体返回(1)

```
typedef struct vint8 { int v[8];} vint8_t;

vint8_t get_vint8(int v0, int v1, int v2,
                  int v3, int v4, int v5,
                  int v6, int v7)
{
    vint8_t s;
    s.v[0] = v0;
    s.v[1] = v1;
    s.v[2] = v2;
    s.v[3] = v3;
    s.v[4] = v4;
    s.v[5] = v5;
    s.v[6] = v6;
    s.v[7] = v7;
    return s;
}
//寄存器a0存放指向在栈中的返回结构
//寄存器a1~a7放参数v0~v6
//参数v7通过栈传递
```

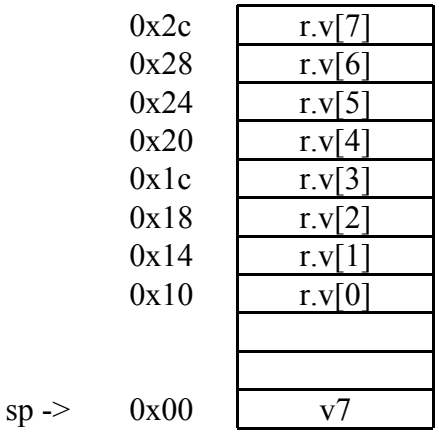
```
000001f0 <get_vint8>:
 1f0:   0080102d        move    v0,a0
 1f4:   ac850000        sw      a1,0(a0)
 1f8:   ac860004        sw      a2,4(a0)
 1fc:   ac870008        sw      a3,8(a0)
 200:   ac88000c        sw      a4,12(a0)
 204:   ac890010        sw      a5,16(a0)
 208:   ac8a0014        sw      a6,20(a0)
 20c:   ac8b0018        sw      a7,24(a0)
 210:   8fa30000        lw      v1,0(sp)
 214:   03e00008        jr      ra
 218:   ac83001c        sw      v1,28(a0)
```



MIPS ABI示例

----大结构体返回(2)

```
typedef struct vint8 { int v[8];} vint8_t;

int sum_vint8(int v)
{
    vint8_t r;
    int     sum = 0;
    r = get_vint8(v, v, v, v, v, v, v, v);
    for (int i=0; i<8; i++) {
        sum += r.v[i];
    }
    return sum;
}

//寄存器a0存放指向在栈中的返回结构，此例为局部变量r
//如果返回值要写到全局变量，则先写到栈中，再搬
```




```
0000021c <sum_vint8>:
 21c:   27bdffc0        addiu   sp,sp,-64
 220:   ffbf0038        sd      ra,56(sp)
 224:   0080582d        move    a7,a0
 228:   afa40000        sw      a0,0(sp)
 22c:   27a40010        addiu   a0,sp,16
 230:   0160282d        move    a1,a7
 234:   0160302d        move    a2,a7
 238:   0160382d        move    a3,a7
 23c:   0160402d        move    a4,a7
 240:   0160482d        move    a5,a7
 244:   0c000000        jal     <get_vint8>
 248:   0160502d        move    a6,a7
 24c:   27a30010        addiu   v1,sp,16
 250:   27a50030        addiu   a1,sp,48
 254:   0000102d        move    v0,zero

 258:   8c640000        lw      a0,0(v1)
 25c:   24630004        addiu   v1,v1,4
 260:   1465fffd        bne     v1,a1,258
 264:   00441021        addu    v0,v0,a0

 268:   dfbf0038        ld      ra,56(sp)
 26c:   03e00008        jr      ra
 270:   27bd0040        addiu   sp,sp,64
```

MIPS ABI示例

----浮点参数(1)

```
float fadd_3(float in0, float in1, float in2)
{
    return in0 + in1 + in2;
}


float fadd_10(int i0, float i1, float i2,
              int i3, float i4, float i5,
              float i6, float i7, float i8,
              int i9)
{
    return i0 + i1 + i2 + i3 + i4 +
           i5 + i6 + i7 + i8 + i9;
}

//8个浮点参数寄存器（$f12 ~ $f19）
//与8个定点参数寄存器对应
//用了定点参数则相应浮点参数跳过不用，反之亦然
//  例如fadd_10中不用$f12和$f15，而是用了a0和a3

//第 8 个参数后使用栈传递
//$f0为返回值
```

```
10000c2c <fadd_3>:
10000c2c:       460d6000        add.s   $f0,$f12,$f13
10000c30:       03e00008        jr      ra
10000c34:       460e0000        add.s   $f0,$f0,$f14

10000c5c <fadd_10>:
10000c5c:       44840800        mtc1    a0,$f1
10000c60:       46800820        cvt.s.w $f0,$f1
10000c64:       460d0340        add.s   $f13,$f0,$f13
10000c68:       460e6b40        add.s   $f13,$f13,$f14
10000c6c:       44870000        mtc1    a3,$f0
10000c70:       468003a0        cvt.s.w $f14,$f0
10000c74:       460e6b40        add.s   $f13,$f13,$f14
10000c78:       46106c00        add.s   $f16,$f13,$f16
10000c7c:       46118440        add.s   $f17,$f16,$f17
10000c80:       46128c80        add.s   $f18,$f17,$f18
10000c84:       461394c0        add.s   $f19,$f18,$f19
10000c88:       c7a00000        lwc1    $f0,0(sp)
10000c8c:       46009b40        add.s   $f13,$f19,$f0
10000c90:       c7a10008        lwc1    $f1,8(sp)
10000c94:       46800820        cvt.s.w $f0,$f1
10000c98:       03e00008        jr      ra
10000c9c:       46006800        add.s   $f0,$f13,$f0
```

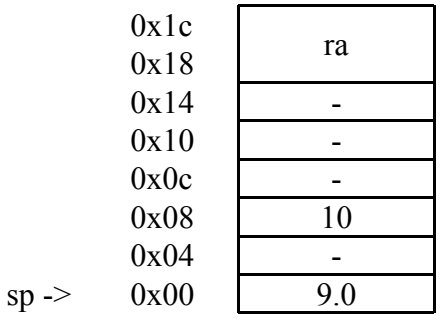
此处反汇编的是最终的可执行文件

objdump -d a.out

MIPS ABI示例

----浮点参数(2)

```
float call_fadd_10()
{
    return fadd_10(1,
                   2.0,3.0,
                   4,
                   5.0,
                   6.0,7.0,8.0,9.0,
                   10);
}
//定点常量用立即数指令初始化
//浮点常量放在只读数据段中，用lwc1初始化
```

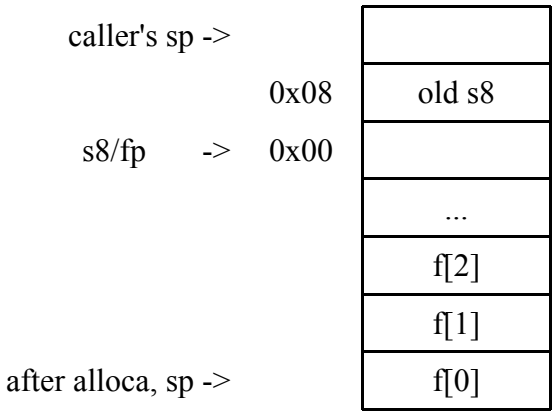



```
10000ca0 <call_fadd_10>:
10000ca0:       27bdffe0        addiu   sp,sp,-32
10000ca4:       ffbf0018        sd      ra,24(sp)
10000ca8:       3c021000        lui     v0,0x1000
10000cac:       c4400f60        lwc1    $f0,3936(v0)
10000cb0:       e7a00000        swc1    $f0,0(sp)
10000cb4:       2402000a        li      v0,10
10000cb8:       afa20008        sw      v0,8(sp)
10000cbc:       24040001        li      a0,1
10000cc0:       3c021000        lui     v0,0x1000
10000cc4:       c44d0f48        lwc1    $f13,3912(v0)
10000cc8:       3c021000        lui     v0,0x1000
10000ccc:       c44e0f4c        lwc1    $f14,3916(v0)
10000cd0:       24070004        li      a3,4
10000cd4:       3c021000        lui     v0,0x1000
10000cd8:       c4500f50        lwc1    $f16,3920(v0)
10000cdc:       3c021000        lui     v0,0x1000
10000ce0:       c4510f54        lwc1    $f17,3924(v0)
10000ce4:       3c021000        lui     v0,0x1000
10000ce8:       c4520f58        lwc1    $f18,3928(v0)
10000cec:       3c021000        lui     v0,0x1000
10000cf0:       0c000317        jal     10000c5c <fadd
10000cf4:       c4530f5c        lwc1    $f19,3932(v0)
10000cf8:       dfbf0018        ld      ra,24(sp)
10000cfc:       03e00008        jr      ra
10000d00:       27bd0020        addiu   sp,sp,32
```

MIPS ABI示例

----栈中分配空间

```
//求阶乘 N!
#include<alloca.h>
int factorial(int n)
{
    int* f = alloca((n+1)*sizeof(int));
    f[0] = 1;
    for (int i=1; i<=n; i++) {
        f[i] = f[i-1] * i;
    }
    return f[n];
}
//栈帧指针s8保存静态分配后的sp指针
//直接把sp减去所要分配的空间大小
// sp -= ((((a0 + 1) << 2) + 30)>>4)<<4
```



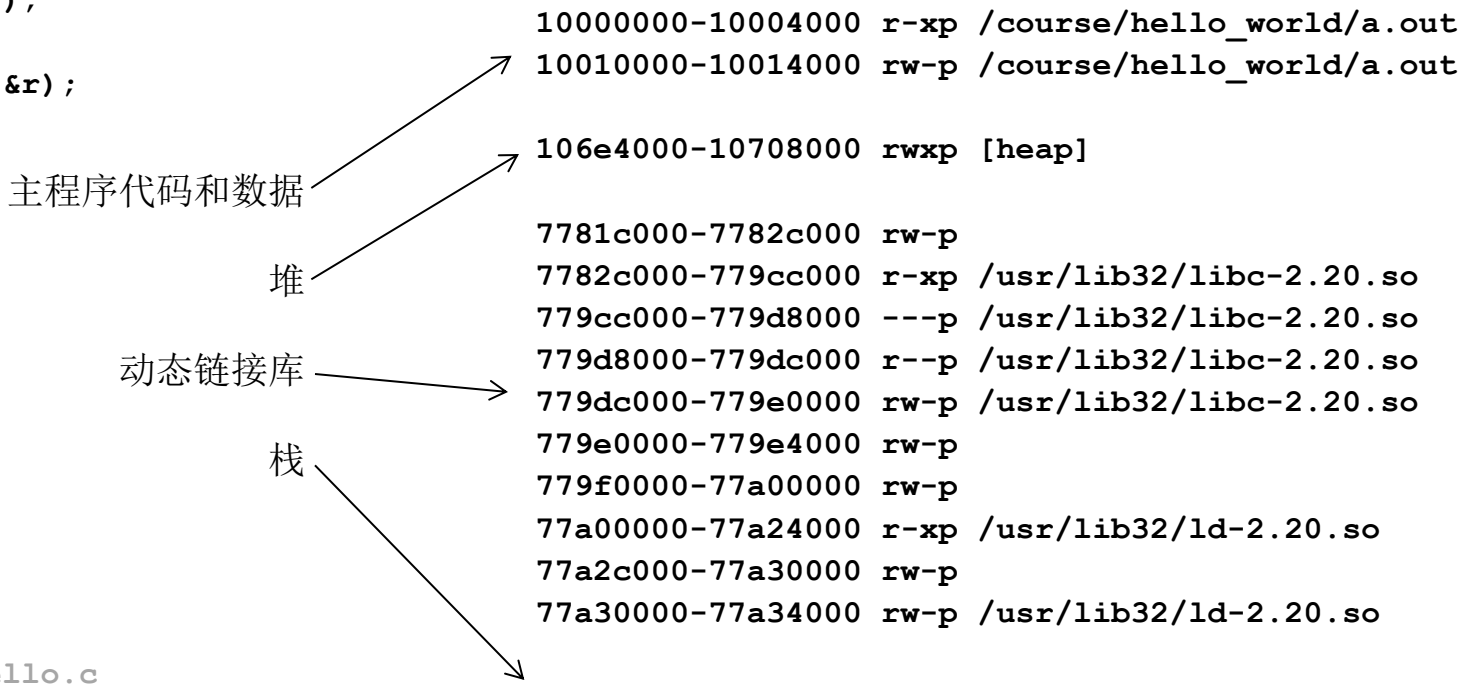

```
00000000 <factorial>:
   0:   27bdfff0        addiu   sp,sp,-16
   4:   ffbe0008        sd      s8,8(sp)
   8:   03a0f02d        move    s8,sp
   c:   24860001        addiu   a2,a0,1
  10:   00063880        sll     a3,a2,0x2
  14:   24e2001e        addiu   v0,a3,30
  18:   00021102        srl     v0,v0,0x4
  1c:   00021100        sll     v0,v0,0x4
  20:   03a2e823        subu    sp,sp,v0
  24:   03a0302d        move    a2,sp
  28:   24020001        li      v0,1
  2c:   18800009        blez    a0,54 <factorial+0x54>
  30:   afa20000        sw      v0,0(sp)//f[0] = 1
  34:   03a0182d        move    v1,sp
  38:   8c650000        lw      a1,0(v1)
  3c:   70a22802        mul     a1,a1,v0
  40:   ac650004        sw      a1,4(v1)//f[i]=f[i-1]*i
  44:   24420001        addiu   v0,v0,1 //i++
  48:   0082282a        slt     a1,a0,v0//loop if!(n<i)
  4c:   10a0fffa        beqz    a1,38 <factorial+0x38>
  50:   24630004        addiu   v1,v1,4
  54:   00c73021        addu    a2,a2,a3
  58:   8cc2fffc        lw      v0,-4(a2)//load f[n]
  5c:   03c0e82d        move    sp,s8
  60:   dfbe0008        ld      s8,8(sp)
  64:   03e00008        jr      ra
  68:   27bd0010        addiu   sp,sp,16
```

gcc -std=c99 -mabi=n32 -O1 -mips64r2 -c fact.c

MIPS ABI示例

----虚存空间分配

```
#include<stdio.h>

int main()
{
    printf("Hello World!\n");
    malloc(10000);
    int r;
    scanf("%d", &r);
    return r;
}
```




```
gcc -mabi=n32 hello.c
./a.out &
cat /proc/$(jobs -p)/maps |
 awk '{printf("%s %s %s\n",$1, $2, $6)}'
```

MIPS ABI示例

----跨函数返回

```
#include<setjmp.h>
#include<stdio.h>
#include<stdlib.h>
jmp_buf env;
void do_something(int level)
{
    printf("in %d\n", level);
    if (level==0) longjmp(env, 2);
    do_something(level-1);
    printf("out %d\n", level);
}
int main()
{
    int i = setjmp(env);
    printf("setjmp return %d\n", i);

    if (i == 0) {
        // first time here
        printf("before do...\n");
        do_something(10);
        printf("after do...\n");
    } else {
        // after longjmp
        printf("longjmp backed...\n");
    }
    return 0;
}
```

执行效果：

```
setjmp return 0
before do...
in 10
in 9
in 8
in 7
in 6
in 5
in 4
in 3
in 2
in 1
in 0
setjmp return 2
longjmp backed...
```

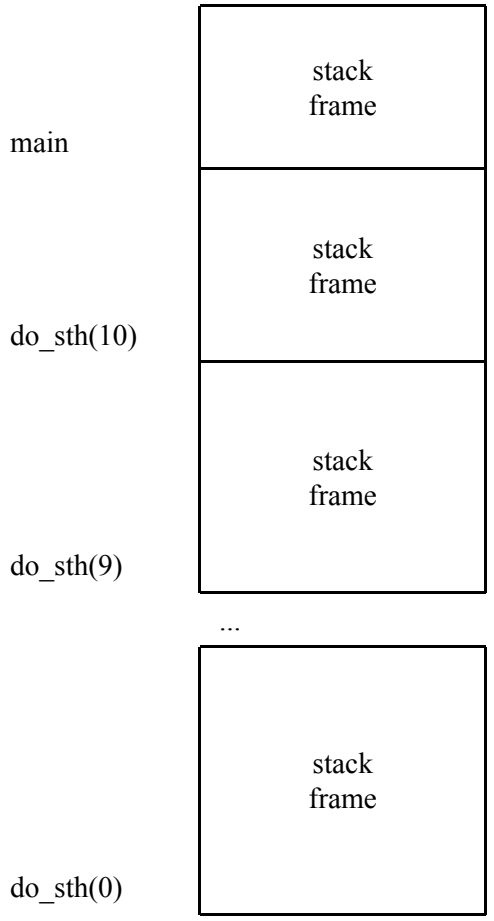
递归到第0层后直接返回到main
执行longjmp后就像直接从setjmp返回一样
如何实现？

MIPS ABI示例

----跨函数返回

在jmpbuf中保存用于返回的上下文包括哪些？

```
typedef struct __jmp_buf_internal_tag
  {
    /* Program counter.  */
    __extension__ long long __pc;

    /* Stack pointer.  */

    __extension__ long long __sp;

    /* Callee-saved registers s0 through s7.  */
    __extension__ long long __regs[8];

    /* The frame pointer.  */
    __extension__ long long __fp;

    /* The global pointer.  */
    __extension__ long long __gp;
    ...

    /* Callee-saved floating point registers.  */
    ...
  } __jmp_buf[1];
```



中断的生命周期

Linux的中断处理过程

- 中断触发
  - 硬件相关过程：外设→中断控制器→处理器
- 硬中断处理
  - 关中断状态下快速的处理过程，“上半部”
- 软中断处理
  - 开中断状态下，完成庞杂的“下半部”
- 中断返回

以龙芯1号Linux系统中的网络收包为例
linux-3.10.84

以网络收包为例—中断触发

- 网卡
  - 收到一个网络包，根据接收描述符的指示，DMA写内存
  - 接收描述符中有中断位，在更新完描述符状态后发出中断
- 中断控制器
  - 收到中断信号，存到状态寄存器中
  - 根据中断使能情况，往处理器传递中断信号
- 处理器
  - 到达中断输入引脚，送到CP0_STATUS.IP域
  - 根据MASK、IE以及EXL/ERL，确定是否处理该中断
  - 如是，则将某条指令带上外部中断的信息，在提交时触发异常
  - 保存PC到EPC，切换到EXL状态，跳到0x80000180

以网络收包为例—中断初始化

- 内核在通用异常入口放置处理代码
- 外部中断入口ebase+0x180、例外号0

arch/mips/kernel/traps.c：

```
void __init trap_init(void)
{
    ...
    //Copy the generic exception handlers to their final destination. ...
    set_handler(0x180, &except_vec3_generic, 0x80);
    ...
    set_except_vector(0, handle_int);
    set_except_vector(1, handle_tlbm);
    set_except_vector(2, handle_tlbl);
    set_except_vector(3, handle_tlbs);
    ...
    set_except_vector(8, handle_sys);
    ...
}
void __cpuinit set_handler(unsigned long offset, void *addr, unsigned long size)
{
    memcpy((void *)(ebase + offset), addr, size);
    local_flush_icache_range(ebase + offset, ebase + offset + size);
}
```

//arch/mips/kernel/genex.S:

```
NESTED(except_vec3_generic, 0, sp)
    .set    push
    .set    noat
    mfc0    k1, CP0_CAUSE
    andi    k1, k1, 0x7c
    PTR_L   k0, exception_handlers(k1)
    jr      k0
    .set    pop
    END(except_vec3_generic)
NESTED(handle_int, PT_SIZE, sp)
    SAVE_ALL
    CLI

    LONG_L  s0, TI_REGS($28)
    LONG_S  sp, TI_REGS($28)
    PTR_LA  ra, ret_from_irq
    PTR_LA  v0, plat_irq_dispatch
    jr      v0
    END(handle_int)
```

以网络收包为例—硬中断(1)

- 通用例外入口(180)
  - 取出Cause，从例外处理函数列表加载处理函数的指针
  - 直接跳过去
- 外部中断入口
  - 保存进程上下文到内核栈
    - SAVE_SOME(sp/$0/v*/a*/jp/gp/ra cp0_status/cause/epc)
    - SAVE_AT(at)
    - SAVE_TEMP(t*/hi/lo)
    - SAVE_STATIC(s*)
  - 关IE，清EXL，切换到内核模式
    - CU0=1
    - KSU=kernel_mode
    - EXL/ERL=0
  - 设置返回函数
  - 转到平台相关中断分发

```
//arch/mips/kernel/genex.S:
NESTED(except_vec3_generic, 0, sp)
        .set    push
        .set    noat
        mfc0    k1, CP0_CAUSE
        andi    k1, k1, 0x7c
        PTR_L   k0, exception_handlers(k1)
        jr      k0
        .set    pop
        END(except_vec3_generic)
NESTED(handle_int, PT_SIZE, sp)
        SAVE_ALL
        CLI

        LONG_L  s0, TI_REGS($28)
        LONG_S  sp, TI_REGS($28)
        PTR_LA  ra, ret_from_irq
        PTR_LA  v0, plat_irq_dispatch
        jr      v0
        END(handle_int)
```

```
arch/mips/include/asm/stackframe.h:
        .macro  SAVE_ALL
        SAVE_SOME
        SAVE_AT
        SAVE_TEMP
        SAVE_STATIC
        .endm
        .macro  CLI
        mfc0    t0, CP0_STATUS
        li      t1, ST0_CU0 | STATMASK
        or      t0, t1
        xori    t0, STATMASK
        mtc0    t0, CP0_STATUS
        irq_disable_hazard // ehb
        .endm
//STATMASK = 0x1f
```

以网络收包为例—硬中断(2)

- 平台相关中断分发
  - 读中断状态
  - 算中断号
  - 调用do_IRQ

```
//arch/mips/loongson1/common/irq.c:
asmlinkage void plat_irq_dispatch(void)
{
        unsigned int pending;
        pending = read_c0_cause() & read_c0_status() & ST0_IM;

        if (pending & CAUSEF_IP7)
                do_IRQ(TIMER_IRQ);
        else if (pending & CAUSEF_IP2)
                ls1x_irq_dispatch(0); /* INT0 */
        else if (pending & CAUSEF_IP3)
                ls1x_irq_dispatch(1); /* INT1 */
        else if (pending & CAUSEF_IP4)
                ls1x_irq_dispatch(2); /* INT2 */
        else if (pending & CAUSEF_IP5)
                ls1x_irq_dispatch(3); /* INT3 */
        else if (pending & CAUSEF_IP6)
                ls1x_irq_dispatch(4); /* INT4 */
        else
                spurious_interrupt();

}
static void ls1x_irq_dispatch(int n)
{
        u32 int_status, irq;
        /* Get pending sources, masked by current enables */
        int_status = __raw_readl(LS1X_INTC_INTISR(n)) &
                     __raw_readl(LS1X_INTC_INTIEN(n));
        if (int_status) {
                irq = LS1X_IRQ(n, __ffs(int_status));
                do_IRQ(irq);
        }
}
```

```
#define LS1X_IRQ(n, x)              ((n << 5) + (x))
#define LS1X_INTC_REG(n, x) \
        ((void __iomem *)KSEG1ADDR(LS1X_INTC_BASE + (n * 0x18) + (x)))

#define LS1X_INTC_INTISR(n)         LS1X_INTC_REG(n, 0x0)
#define LS1X_INTC_INTIEN(n)         LS1X_INTC_REG(n, 0x4)
#define LS1X_INTC_INTSET(n)         LS1X_INTC_REG(n, 0x8)
#define LS1X_INTC_INTCLR(n)         LS1X_INTC_REG(n, 0xc)
#define LS1X_INTC_INTPOL(n)         LS1X_INTC_REG(n, 0x10)
#define LS1X_INTC_INTEDGE(n)        LS1X_INTC_REG(n, 0x14)
```

如果有向量中断会怎样处理？

以网络收包为例—硬中断(3)

- do_IRQ
  - 设置硬中断标记
  - 调用挂载在对应中断号的处理程序
  - 退出硬中断，检查并处理软中断

//arch/mips/kernel/irq.c：

```
/*
 * do_IRQ handles all normal device IRQ's (the special
 * SMP cross-CPU interrupts have their own specific
 * handlers).
 */
void __irq_entry do_IRQ(unsigned int irq)
{
        irq_enter();
        check_stack_overflow();
        if (!smtc_handle_on_other_cpu(irq))
                generic_handle_irq(irq);
        irq_exit();   //invoke_softirq();
}
```

//drivers/net/ethernet/stmicro/stmmac/stmmac_main.c：

```
static int stmmac_open(struct net_device *dev)
{
        ...
        /* Request the IRQ lines */
        ret = request_irq(dev->irq, stmmac_interrupt,  IRQF_SHARED, dev->name, dev);
        ...
}
```

以网络收包为例—硬中断(4)

- 网卡的中断处理
  - 读网卡寄存器，确定需要处理的事件
  - 关闭网卡中断使能，避免大量网络包引起频繁中断
  - 通知NAPI有新任务，但不立即执行

```
//drivers/net/ethernet/stmicro/stmmac/stmmac_main.c
static void stmmac_dma_interrupt(struct stmmac_priv *priv)
{
        int status;
        status = priv->hw->dma->dma_interrupt(priv->ioaddr, &priv->xstats);

        if (likely((status & handle_rx)) || (status & handle_tx)) {
                if (likely(napi_schedule_prep(&priv->napi))) {
                        stmmac_disable_dma_irq(priv);
                        __napi_schedule(&priv->napi);
                }
        }
        ...
}
```

```
//drivers/net/ethernet/stmicro/stmmac/dwmac_lib.c:
int dwmac_dma_interrupt(void __iomem *ioaddr,
                        struct stmmac_extra_stats *x)
{
        int ret = 0;
        /* read the status register (CSR5) */
        u32 intr_status = stmmac_readl(ioaddr + DMA_STATUS);
        ...
        /* Clear the interrupt by writing a logic 1 to the CSR5[15-0] */
        stmmac_writel((intr_status & 0x1ffff), ioaddr + DMA_STATUS);
        return ret;
}
```

如果用消息中断会怎样处理？

以网络收包为例—软中断

- 通过软件优化，提高系统响应速度
  - 在开中断状态下运行
  - 处理硬中断期间所设置的处理任务
  - 内核中预定义了10种软中断，一般驱动只能用基于软中断实现的tasklet机制

```
void irq_exit(void) {
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();
}
static int stmmac_poll(struct napi_struct *napi, int budget)
{
    ...
    stmmac_tx_clean(priv); //发送
    work_done = stmmac_rx(priv, budget); //接收
    if (work_done < budget) { //接收的包个数少于允许的上限，则收完了
        napi_complete(napi);
        stmmac_enable_dma_irq(priv);
    }
    return work_done;
}
```

中断返回

- 恢复上下文，执行eret指令
- ret_from_irq
  - resume_kernel
    - 如果可抢占，则尝试调度
    - 否则restore_all...eret
  - resume_userspace
    - 检查是否有待办事项

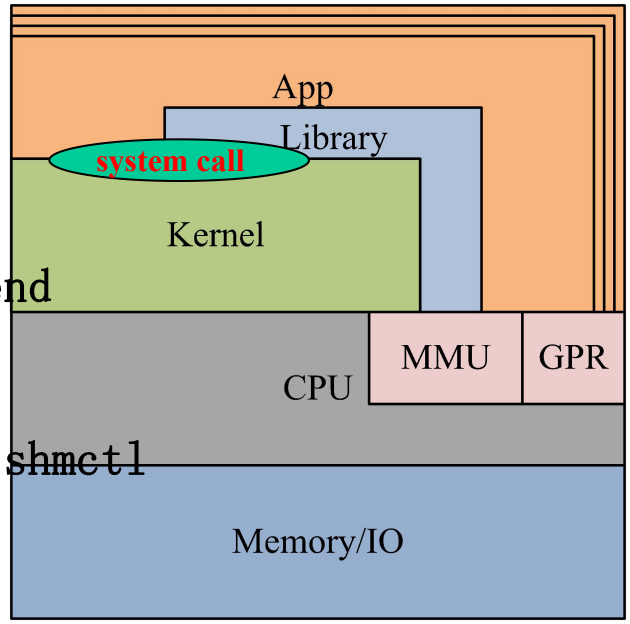
    （有抢占需求、有回调函数等）
    - 否则restore_all...eret

```
//arch/mips/kernel/entry.S:
FEXPORT(ret_from_irq)
    LONG_S    s0, TI_REGS($28)
resume_userspace_check:
    LONG_L    t0, PT_STATUS(sp)          # returning to kernel mode?
    andi      t0, t0, KU_USER
    beqz      t0, resume_kernel
resume_userspace:
    local_irq_disable
    LONG_L    a2, TI_FLAGS($28)      # current->work
    andi      t0, a2, _TIF_WORK_MASK  # (ignoring syscall_trace)
    bnez      t0, work_pending
    j         restore_all
resume_kernel:
    local_irq_disable
    lw        t0, TI_PRE_COUNT($28)
    bnez      t0, restore_all
need_resched:
    LONG_L    t0, TI_FLAGS($28)
    andi      t1, t0, _TIF_NEED_RESCHED
    beqz      t1, restore_all
    LONG_L    t0, PT_STATUS(sp)          # Interrupts off?
    andi      t0, 1
    beqz      t0, restore_all
    jal       preempt_schedule_irq
    b         need_resched
restore_all:                    # restore full frame
    .set    noat
    RESTORE_TEMP
    RESTORE_AT
    RESTORE_STATIC
restore_partial:          # restore partial frame
    RESTORE_SOME
    RESTORE_SP_AND_RET
```

系统调用过程

系统调用简介

- 内核给用户程序提供的子程序
  - 进程控制：fork, clone, execve, exit, getpid, pause, wait
  - 文件操作：create, open, close, read, write, access, chdir, chmod
  - 系统控制：ioctl, ioperm, gettimeofday
  - 内存管理：brk, mmap, sync
  - 网络管理：gethostname, socket, connect, send
  - 用户管理：getuid, getgid
  - 进程间通信：signal, msgsnd, pipe, semctl, shmctl
  - 用strace命令可以跟踪一个进程的系统调用



- 两个基本要求
  - 安全性：核心态下运行，所有参数都应检查，避免受攻击
  - 兼容性：避免改应用，保护生态

系统调用简介

- 与函数调用类似，Linux同时支持多种ABI
  - Syscall ID: O32(4000~4999), N32(5000~5999), N64(6000~6999)

Register	use on input	use on output	Note
at	—	(caller saved)	
v0	syscall number	return value	
v1	—	2nd fd only for pipe(2)	
a0~a2	syscall arguments	returned unmodified	O32
a0~a2, a4~a7	syscall arguments	returned unmodified	N32 and 64
a3	4th syscall argument	a3 set to 0/1 for success/error	
t0~t9	—	(caller saved)	
s0~s8	—	(callee saved)	
hi, lo	—	(caller saved)	

系统调用例子

纯汇编的HelloWorld

- 将字符串写到标准输出
  - a0：文件描述符
  - a1：写的内容
  - a2：写的字节数
  - v0：write系统调用号
  - 调用完v0返回实现写的数量
- 退出进程
  - a0：返回值
  - v0：exit系统调用

```
//hello.S:
#include<regdef.h>
#include<syscall.h>

        .global main
        .ent    main
        .set    noreorder
main:
        li      a0, 1           // 1: stdout
        la      a1, hello_str
        li      a2, hello_end - hello_str
        li      v0, SYS_write
        syscall

        li      a0, 3           // return value
        li      v0, SYS_exit
        syscall
        .end    main

        .data
hello_str:
        .ascii "Hello World!\n"
hello_end:
```

gcc -g -fno-pic -mno-abicalls -mabi=n32 -c hello.S
ld -melf32ltsmipn32 -emain hello.o

系统调用过程

- syscall指令
- 通用例外入口(0x80000180)
- excode=0x08, handle_sys
- SAVE_SOME
  - 切换至线程的内核栈
  - 保存sp/$0/v*/a*/jp/gp/ra cp0_status/cause/epc)
- STI
  - 切换成内核模式
  - 开中断
- 根据系统调用号跳到相应的服务函数

```
//arch/mips/kernel/scall64-n32.S:
        .align  5
NESTED(handle_sysn32, PT_SIZE, sp)
        .set    noat
        SAVE_SOME
        TRACE_IRQS_ON_RELOAD
        STI
        .set    at

        dsubu   t0, v0, __NR_N32_Linux  # check syscall number
        sltiu   t0, t0, __NR_N32_Linux_syscalls + 1

        ld      t1, PT_EPC(sp)          # skip syscall on return
        daddiu  t1, 4                   # skip to next instruction
        sd      t1, PT_EPC(sp)
        beqz    t0, not_n32_scall

        dsll    t0, v0, 3               # offset into table
        ld      t2, (sysn32_call_table - (__NR_N32_Linux * 8))(t0)

        ...

        jalr    t2                      # Do The Real Thing (TM)

        li      t0, -EMAXERRNO - 1      # error?
        sltu    t0, t0, v0
        sd      t0, PT_R7(sp)           # set error flag
        beqz    t0, 1f

        ld      t1, PT_R2(sp)           # syscall number
        dnegu   v0                      # error
        sd      t1, PT_R0(sp)           # save it for syscall rest
1:      sd      v0, PT_R2(sp)           # result

        j       syscall_exit_partial
```

系统调用过程

- 调用结束返回
- 如果没有额外要做的，则返回

  retore_partial
- 否则保存更多的上下文，进一步处理

  resume_userspace中含restore_all

```
//arch/mips/kernel/entry.S:
FEXPORT(syscall_exit_partial)
        local_irq_disable               # make sure need_resched d
                                        # change between and retur
        LONG_L  a2, TI_FLAGS($28)       # current->work
        li      t0, _TIF_ALLWORK_MASK
        and     t0, a2
        beqz    t0, restore_partial
        SAVE_STATIC
syscall_exit_work:
        LONG_L  t0, PT_STATUS(sp)               # returning to ker
        andi    t0, t0, KU_USER
        beqz    t0, resume_kernel
        li      t0, _TIF_WORK_SYSCALL_EXIT
        and     t0, a2                  # a2 is preloaded with TI_
        beqz    t0, work_pending        # trace bit set?
        local_irq_enable                # could let syscall_trace_
                                        # call schedule() instead
        move    a0, sp
        jal     syscall_trace_leave
        b       resume_userspace

restore_partial:                # restore partial frame
        RESTORE_SOME
        RESTORE_SP_AND_RET
```

同步与通讯

同步与通讯

- 现代操作系统的关键特性：多任务
- 当同一块数据被多段代码并发访问时需要同步
- 并发访问可能来自
  - 多处理器系统中同时运行的多个代码块
  - 由于中断引起并发代码段
  - 由于线程调度引起的多线程并发执行
- 多数代码隐含原子性实现的假定
  - 读—改—写
  - 队列插入、删除
  - 红黑树插入、删除

基于锁的同步

- 将可能并发访问同一块数据的代码定义为临界区
- 进入临界区前先申请锁（并加锁）
- 申请失败者被阻塞
- 持有锁的线程离开临界区后释放锁
- 锁的类型
  - 自旋锁：循环等待直至得到锁
  - 互斥锁：如果加锁失败则休眠等待
  - 读写锁：允许有多个读或者一个写

  ...

锁的类型

- 自旋锁(spinlock)
  - 循环等待直至得到锁
  - 在多处理器环境下使用
  - 传统的自旋锁不够公平

    不体现先后次序

    可能出现饿死的情况
  - 排队自旋锁

    增加ticket和serving_now

    原子性地取号

    根据当前服务号判断是否抢到

    解锁时将服务号加1

```
selfspin:
    ll   t0, lock
    bnez t0, selfspin //wait till unlocked
    nop
    li   t1, 0x1
    sc   t1, lock
    beqz t1, selfspin //loop if failed
    nop

    <Critical section>

unlock:
    sw   zero, lock
```

锁的类型

- 互斥锁(mutex)
  - 由count决定锁的状态

    =1 ： 可用

    =0 ： 已锁定

    <0 ： 已锁定，多个加锁请求
  - 没取到锁则睡眠等待
  - 解锁时检查等待队列

与IO的中断模式类似
spinlock则对应轮循

```
//include/linux/mutex.h
struct mutex {
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
        struct task_struct      *owner;
        ...
};


//include/asm-generic/mutex-dec.h
static inline void
__mutex_fastpath_lock(atomic_t *count,
                      void (*fail_fn)(atomic_t *))
{
        if (atomic_dec_return(count) < 0)
                fail_fn(count);
}
static inline void
__mutex_fastpath_unlock(atomic_t *count,
                      void (*fail_fn)(atomic_t *))
{
        if (atomic_inc_return(count) <= 0)
                fail_fn(count);
}
```

锁的类型

- 读写锁(rwlock)
  - 允许一个写线程或多个读线程访问共享数据，但读写之间互斥。可允许多个读同时访问，使读数据的效率更高。
  - 适用于共享数据频繁被多个线程同时读，但很少修改的情况。
- 顺序锁(seqlock)
  - 写优先（spinlock不分读写、rwlock读优先）
  - 写进入和退出时对锁变量作原子性递增
  - 读开始和结束时取锁变量作判断，相等且不为奇数时读不需要重试
- RCU锁(Read Copy Update)
  - 共享数据通过指针引用，每个写都要创建一个副本
  - 旧的副本没有在引用后删除
  - 相比顺序锁，不需要让读进程重试

非阻塞的同步

- 锁的问题
  - 若持有锁的线程死亡、阻塞或死循环
  - 加解锁的代价
  - 并发度受限，细粒度锁设计难度大
- 事务内存的思想
  - 把临界区代码定义为事务
  - 尝试性地执行事务代码
  - 执行过程中动态检测冲突
  - 根据冲突检测结果提交或取消事务
  - *LL/SC可以理解为大小仅为一个word的事务操作*

非阻塞的同步

- SUN Rock增加两条指令
  - chkpt <fail_pc>：事务开始，取消时跳转到<fail_pc>
  - commit：提交事务
- X86 TSX两套机制
  - Hardware Lock Elision
    - 通过在锁获取和锁释放指令前添加前缀来实现
    - 总是假定成功获取锁，但不修改真正的信号量
    - 软件可以与老机型兼容
  - Restricted Transactional Memory
    - XBEGIN：开始事务，给出用于事务中断的emergency函数
    - XEND ：用于事务提交
    - XABORT：用于显式中止transaction

作业