浙江大学

本科实验报告

课程名称：操作系统

姓　　名：黄文杰

学　　院：计算机科学与技术学院

系：软件工程系

专　　业：软件工程

学　　号：3210103379

指导教师：夏莹杰

2023 年 11 月 15 日

浙江大学操作系统实验报告

实验名称： Lab 4: RV64 用户态程序

电子邮件地址： s2530555829@outlook.com 手机： 19857139861

实验地点： 浙江大学玉泉校区 实验日期： 2023 年 11 月 24 日

一、实验目的和要求

- 创建用户态进程，并设置 sstatus 来完成内核态转换至用户态。
- 正确设置用户进程的用户态栈和内核态栈， 并在异常处理时正确切换。
- 补充异常处理逻辑，完成指定的系统调用（SYS_WRITE, SYS_GETPID）功能。

二、实验过程

实验环境为 Ubuntu 22.04.2 LTS Windows Subsystem for Linux 2

1. 准备工程

修改 vmlinux.lds.S，将用户态程序 uapp 加载至 .data 段。

需要修改 defs.h，在 defs.h 添加如下内容。

#define USER_START (0x0000000000000000) // user space start virtual address

#define USER_END (0x0000004000000000) // user space end virtual address

从 repo 同步要求的文件和文件夹并正确放置。

修改根目录下的 Makefile，将 user 纳入工程管理。

2. 创建用户态进程

2.1 修改 proc.h

本次实验只需要创建 4 个用户态进程，修改 proc.h 中的 NR_TASKS 如下。

```
#define NR_TASKS  (1 + 4)
```

由于创建用户态进程要对 sepc, sstatus, sscratc 做设置，我们将其加入 thread_struct 中。由于多个用户态进程需要保证相对隔离，因此不可以共用页表。我们为每个用户态进程都创建一个页表。修改 task_struct 如下。

```
typedef unsigned long* pagetable_t;
/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];
    uint64_t sepc, sstatus, sscratch;
};

/* 线程数据结构 */
struct task_struct {
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1 最低 10 最高
    uint64 pid;      // 线程 id
    uint64 kernel_sp;
```

```
    uint64 user_sp;
    struct thread_struct thread;
    pagetable_t pgd;
};
```

由于修改了 task_struct，还需要更改 switch_to 中各个变量的偏移量，使我们的 OS 正常运行。

2.2 修改 task_init

对每个用户态进程，其拥有两个 stack： U-Mode Stack 以及 S-Mode Stack， 其中 S-Mode Stack 在 lab3 中我们已经设置好了。我们可以通过 alloc_page 接口申请一个空的页面来作为 U-Mode Stack，实现如下：

```
task[i]->kernel_sp = (unsigned long)(task[i])+PGSIZE;
task[i]->user_sp = kalloc();
```

为每个用户态进程创建自己的页表并将 uapp 所在页面，以及 U-Mode Stack 做相应的映射，同时为了避免 U-Mode 和 S-Mode 切换的时候切换页表，我们也将内核页表（swapper_pg_dir）复制到每个进程的页表中，实现如下：

```
pagetable_t pgtbl = kalloc();
memcpy(pgtbl, swapper_pg_dir, PGSIZE);
```

注意程序运行过程中，有部分数据不在栈上，而在初始化的过程中就已经被分配了空间（比如我们的 uapp 中的 counter 变量），所以二进制文件需要先被拷贝到一块某个进程专用的内存之后再进行映射，防止所有的进程共享数据，造成期望外的进程间相互影响，实现如下：

```
unsigned long va = USER_START;
unsigned long pa = (unsigned long)(uapp_start)-PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, (unsigned long)(uapp_end)-(unsigned
long)(uapp_start), 31);
va = USER_END-PGSIZE;
pa = task[i]->user_sp-PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, PGSIZE, 23);
unsigned long satp = csr_read(satp);
satp = (satp >> 44) << 44;
satp |= ((unsigned long)pgtbl-PA2VA_OFFSET) >> 12;
task[i]->pgd = satp;
```

对每个用户态进程我们需要将 sepc 修改为 USER_START，配置修改好 sstatus 中的 SPP（使得 sret 返回至 U-Mode）， SPIE （sret 之后开启中断）， SUM （S-Mode 可以访问 User 页面）， sscratch 设置为 U-Mode 的 sp，其值为 USER_END（即 U-Mode Stack 被放置在 user space 的最后一个页面）。

查看文档可以得到 sstatus 寄存器的结构如下：

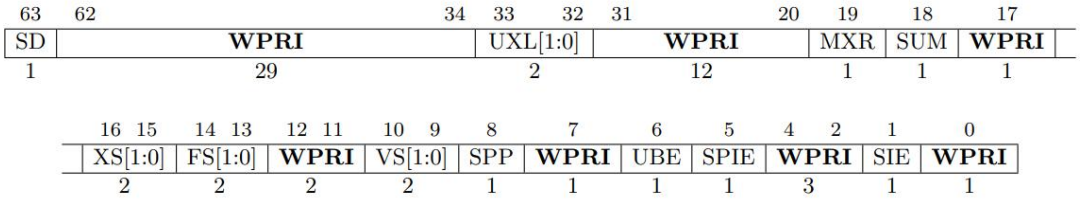

Figure 4.2: Supervisor-mode status register (sstatus) when SXLEN=64.

实现如下：

```
task[i]->thread.sepc = USER_START;
unsigned long sstatus = csr_read(sstatus);
sstatus &= ~(1<<8); // set sstatus[SPP] = 0
sstatus |= 1<<5; // set sstatus[SPIE] = 1
sstatus |= 1<<18; // set sstatus[SUM] = 1
task[i]->thread.sstatus = sstatus;
task[i]->thread.sscratch = USER_END;
```

修改__switch_to， 需要加入 保存/恢复 sepc, sstatus, sscratch 以及切换页表的逻辑。在切换了页表之后，需要通过 fence.i 和 vma.fence 来刷新 TLB 和 ICache。

实现如下：

```
_switch_to:
    # save state to prev process
    addi t0, a0, 40
    sd ra, 0(t0)
    sd sp, 8(t0)

    addi t0, a0, 56
    sd s0,0(t0)
    sd s1,8(t0)
    sd s2,2*8(t0)
    sd s3,3*8(t0)
    sd s4,4*8(t0)
    sd s5,5*8(t0)
    sd s6,6*8(t0)
    sd s7,7*8(t0)
```

```
sd s8,8*8(t0)
sd s9,9*8(t0)
sd s10,10*8(t0)
sd s11,11*8(t0)

addi t0, a0, 152
csrr t1, sepc
sd t1, 0(t0)
csrr t1, sstatus
sd t1, 8(t0)
csrr t1, sscratch
sd t1, 16(t0)
csrr t1, satp
sd t1, 24(t0)

# restore state from next process
addi t0, a1, 48
ld ra, 0(t0)
ld sp, 8(t0)

addi t0, a1, 56
ld s0,0(t0)
ld s1,8(t0)
ld s2,2*8(t0)
ld s3,3*8(t0)
ld s4,4*8(t0)
ld s5,5*8(t0)
ld s6,6*8(t0)
ld s7,7*8(t0)
ld s8,8*8(t0)
ld s9,9*8(t0)
ld s10,10*8(t0)
ld s11,11*8(t0)

addi t0, a1, 152
ld t1, 0(t0)
csrw sepc, t1
ld t1, 8(t0)
csrw sstatus, t1
ld t1, 16(t0)
csrw sscratch, t1
ld t1, 24(t0)
csrw satp, t1
```

```
    # flush tlb
    sfence.vma zero, zero
    # flush icache
    fence.i

    ret
```

3. 修改中断入口/返回逻辑以及中断处理函数

与 ARM 架构不同的是，RISC-V 中只有一个栈指针寄存器(sp)，因此需要我们来完成用户栈与内核栈的切换。

由于我们的用户态进程运行在 U-Mode 下， 使用的运行栈也是 U-Mode Stack，因此当触发异常时， 我们首先要对栈进行切换。同理 让我们完成了异常处理，从 S-Mode 返回至 U-Mode， 也需要进行栈切换。

修改 __dummy。在 2 中我们初始化时， thread_struct.sp 保存了 S-Mode sp，thread_struct.sscratch 保存了 U-Mode sp， 因此在 S-Mode 切换为 U->Mode 的时候，我们只需要交换对应的寄存器的值即可。

实现如下：

```
_dummy:
    la t0,dummy
    csrw sepc,t0
    csrr t1,sp
    csrr t2,sscratch
    csrw sp,t2
    csrw sp,t1
    sret
```

修改 _trap 。同理 在 _trap 的首尾我们都需要做类似的操作。注意如果是内核线程(没有 U-Mode Stack) 触发了异常，则不需要进行切换。（内核线程的 sp 永远指向的 S-Mode Stack， sscratch 为 0）

实现如下：

```
_traps:
    csrr t1,sscratch
    beq t1, x0, _traps_switch
    csrw sscratch,sp
    csrw sp,t1
_traps_switch:
    ...
```

```
    csrr t1,sscratch
    beq t1, x0, _traps_end
    csrw sscratch,sp
    csrw sp,t1
_traps_end:
    sret
```

uapp 使用 ecall 会产生 ECALL_FROM_U_MODE exception。因此我们需要在 trap_handler 里面进行捕获。首先修改 trap_handler 如下：

```
struct pt_regs {
  unsigned long x[32];
  unsigned long sepc;
  unsigned long sstatus;
}
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs)
{
    ...
}
```

查阅 RISCV 文档对 scause 的介绍

4.1.8 Supervisor Cause Register (scause)

The scause register is an SXLEN-bit read-write register formatted as shown in Figure 4.11. When a trap is taken into S-mode, scause is written with a code indicating the event that caused the trap. Otherwise, scause is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the scause register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Table 4.2 lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field. It is required to hold the values 0–31 (i.e., bits 4–0 must be implemented), but otherwise it is only guaranteed to hold supported exception codes.

SXLEN-1	SXLEN-2 … 0
Interrupt	Exception Code (**WLRL**)
1	SXLEN-1

Figure 4.11: Supervisor Cause register scause.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

通过分析发现 Interrupt 为 0 且 Exception Code 为 8 时代表 ECALL_FROM_U_MODE 异常，因此捕获 ECALL_FROM_U_MODE 异常的具体实现如下：

```
void trap_handler(uint64_t scause, uint64_t sepc, struct pt_regs *regs)
{
    //判断是否是 Interrupt
    if ((scause & 0x1ULL<<63) == 0x1ULL<<63)
    {

        //判断是否是 timer interrupt
        unsigned long long exception = scause & ~(0x1ULL<<63);
        if (exception == 5)
        {

            clock_set_next_event();
            do_timer();
```

```
        }
        else if(exception == 8)
        {
            syscall(regs);
        }
    }
    return;
}
```

4. 添加系统调用

本次实验要求实现的的系统调用函数原型以及具体功能如下：

64 号系统调用 sys_write(unsigned int fd, const char* buf, size_t count) 该调用将用户态传递的字符串打印到屏幕上，此处 fd 为标准输出（1），buf 为用户需要打印的起始地址，count 为字符串长度，返回打印的字符数。(具体见 user/printf.c)

172 号系统调用 sys_getpid()该调用从 current 中获取当前的 pid 放入 a0 中返回，无参数。（ 具体见 user/getpid.c ）

在 trap.c 中实现 getpid 以及 write 逻辑。针对系统调用这一类异常， 我们还需要手动将 sepc + 4。

```
void syscall(struct pt_regs* regs) {
     if (regs->x[17] == SYS_WRITE) {
        if (regs->x[10] == 1) {
            int i;
            char* buf = (char*)regs->x[11];
            for ( i = 0; i < regs->x[12]; i++) {
                printk("%c", buf[i]);
            }
            regs->x[10] = i;
        } else {
            regs->x[10] = -1;
        }
     }
    else if (regs->x[17] == SYS_GETPID) {
        regs->x[10] = current->pid;
        // printk("SYS_GETPID:%ld\n",regs->x[10]);
    }
    regs->sepc+=4;
}
```

5. 修改 head.S 以及 start_kernel

在之前的 lab 中， 在 OS boot 之后，我们需要等待一个时间片，才会进行调度。我们现在更改为 OS boot 完成之后立即调度 uapp 运行。

在 start_kernel 中调用 schedule() 注意放置在 test() 之前。

将 head.S 中 enable interrupt sstatus.SIE 逻辑注释，确保 schedule 过程不受中断影响。

6. 测试二进制文件

```
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
setup_vm is done !
...buddy_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
 [S-MODE] Hello RISC-V

switch to [PID = 1 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2

switch to [PID = 4 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2

switch to [PID = 3 COUNTER = 8]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3

switch to [PID = 2 COUNTER = 9]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3

SET [PID = 1 COUNTER = 2]
SET [PID = 2 COUNTER = 1]
SET [PID = 3 COUNTER = 1]
SET [PID = 4 COUNTER = 6]

switch to [PID = 3 COUNTER = 1]

switch to [PID = 1 COUNTER = 2]

switch to [PID = 4 COUNTER = 6]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.3
```

7. 添加 ELF 支持

首先我们需要将 uapp.S 中的 payload 换成我们的 ELF 文件。具体实现如下：

```
.section .uapp

.incbin "uapp"
```

对 task_init 中的初始化步骤进行修改,以通过 ELF 载入用户程序。具体实现如下：

```
void task_init()
{
    ...
    for (int i = 1; i < NR_TASKS; i++){
        task[i] = (struct task_struct *)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->counter = task_test_counter[i];
        task[i]->priority = task_test_priority[i];
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)&_dummy;
        task[i]->thread.sp = (uint64)(task[i]) + PGSIZE;
        //load_bin(task[i]);
        load_program(task[i]);
    }
    ...
}


static uint64_t load_program(struct task_struct* task) {
    Elf64_Ehdr* ehdr = (Elf64_Ehdr*)uapp_start;

    uint64_t phdr_start = (uint64_t)ehdr + ehdr->e_phoff;
    int phdr_cnt = ehdr->e_phnum;

    Elf64_Phdr* phdr;
    int load_phdr_cnt = 0;
    uint64* pgtbl = (uint64 *)alloc_page();
    memcpy(pgtbl, swapper_pg_dir, PGSIZE);
    uint64 va,pa;
    for (int i = 0; i < phdr_cnt; i++) {
        phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD) {
        // 计算所需页面数量并分配空间
        uint64 phdr_num = PGROUNDUP((uint64)phdr->p_memsz + phdr->p_vaddr
- PGROUNDDOWN(phdr->p_vaddr)) / PGSIZE;
```

```
        uint64 *phdr_temp = (uint64 *)alloc_pages(phdr_num);
        //从 elf 文件拷贝大小为 p_filesz 的内容
        memcpy((void *)((uint64)phdr_temp + phdr->p_vaddr -
PGROUNDDOWN(phdr->p_vaddr)),
        (void*)((uint64)&uapp_start + (uint64)phdr->p_offset),
(uint64)phdr->p_filesz);
        //将 [p_vaddr + p_filesz, p_vaddr + p_memsz)对应的物理区间清零
         memset((void *)((uint64)phdr_temp + phdr->p_vaddr -
PGROUNDDOWN(phdr->p_vaddr)+(uint64)phdr->p_filesz),
        0, (uint64)phdr->p_memsz-(uint64)phdr->p_filesz);
         // do mapping
        va = (uint64)PGROUNDDOWN(phdr->p_vaddr);
        pa = (uint64)phdr_temp-PA2VA_OFFSET;
        create_mapping(pgtbl, va, pa, phdr_num*PGSIZE, (phdr->p_flags << 1)
| 17);
        }
    }

    // allocate user stack and do mapping
    // code...
    // following code has been written for you
    // set user stack
    task->kernel_sp = (uint64)(task)+PGSIZE;
    task->user_sp = alloc_page();
    va = USER_END-PGSIZE;
    pa = task->user_sp-PA2VA_OFFSET;
    create_mapping(pgtbl, va, pa, PGSIZE, 23);
    uint64 satp = csr_read(satp);
    satp = (satp >> 44) << 44;
    satp = satp|(((uint64)pgtbl-PA2VA_OFFSET) >> 12);
    task->satp = (uint64*)satp;
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    // sstatus bits set
    uint64 sstatus = csr_read(sstatus);
    sstatus = sstatus&(~(1<<8)); // set sstatus[SPP] = 0
    sstatus = sstatus|(1<<5); // set sstatus[SPIE] = 1
    sstatus = sstatus|(1<<18); // set sstatus[SUM] = 1
    task->thread.sstatus = sstatus;
    // user stack for user program
     task->thread.sscratch = USER_END;
}
```

8. 测试 ELF 文件

```
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
setup_vm is done !
...buddy_init done!
mapping kernel text !
mapping kernel rodata !
mapping other memory !
...proc_init done!
 [S-MODE] Hello RISC-V

switch to [PID = 1 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.2

switch to [PID = 4 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2

switch to [PID = 3 COUNTER = 8]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3

switch to [PID = 2 COUNTER = 9]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.2
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3

SET [PID = 1 COUNTER = 2]
SET [PID = 2 COUNTER = 1]
SET [PID = 3 COUNTER = 1]
SET [PID = 4 COUNTER = 6]

switch to [PID = 3 COUNTER = 1]

switch to [PID = 1 COUNTER = 2]
```

三、讨论和心得

遇到的问题

在修改了 task_struct 的结构之后，会影响到进程切换_switch_to 函数中的保存和回复进程相关信息的逻辑。

解决方案

在修改 task_struct 的结构后要注意同步修改_switch_to 函数中的相关逻辑，以保证和当前 task_struct 结构的一致性。

遇到的问题

在进行用户地址空间映射时，发生 Insruction access Fault。

解决方案

检查发现是设置虚拟内存权限时存在问题，需要深入了解设置权限时各个字段并正确设置权限。

心得体会

在实验过程中，我们主要遇到了三个难点。首先，我们需要设计用户态进程 PCB 并进行适当的初始化；其次，我们需要设计切换用户态进程和内核态进程的逻辑；最后，我们需要使用 ELF 文件载入我们的用户态程序，这需要我们对 ELF 文件有一定的了解

由于我们的中断处理程序只对一小部分我们所需的中断类型进行了处理，当发生其他中断时我们无法了解发生的具体中断类型，只会表现为程序不正常运行。**因此需要在中断处理程序中输出与中断类型相关的调试信息，便于排查错误和调试程序**。

总的来说，通过这次实验，我们对操作系统内核的用户程序运行机制有了更深入的理解。这将对我们今后的学习和实践中有很大的帮助。

四、思考题

1. 我们在实验中使用的用户态线程和内核态线程的对应关系是怎样的？（一对一，一对多，多对一还是多对多）

一对一。

2. 为什么 Phdr 中，p_filesz 和 p_memsz 是不一样大的？

p_filesz（File Size）： 表示该段在可执行文件中的大小，即占用的文件空间大小。这个字段告诉操作系统在加载这个段时需要从可执行文件中读取多少字节的数；p_memsz（Memory Size）： 表示该段在内存中的大小，即在程序执行时实际占用的内存空间大小。这个字段告诉操作系统在运行时为这个段分配多少内存空间。在实际情况下，p_filesz 几乎总是小于等于 p_memsz，原因有以下三种：1.在可执行文件中，某些段可能包含未初始化的数据，这些数据在文件中并不占用实际空间，但在内存中需要被初始化为零。2.在内存中，数据通常需要按照一定的边界对齐，这可能导致 p_memsz 大于 p_filesz。3.可能有一些元数据或附加信息存储在内存中，而这部分在文件中并没有对应的存储

3. 为什么多个进程的栈虚拟地址可以是相同的？用户有没有常规的方法知道自己栈所在的物理地址？

因为每个进程都有自己的页表。在不同的页表中，虽然虚拟地址相同，但它们映射到不同的物理内存地址，因此多个进程可以具有相同的栈虚拟地址。
用户若要知道自己栈的物理地址，必须知道虚拟地址映射的方式，从虚拟地址计算出物理地址。否则没有其它常规方法直接知道物理地址。

五、附录

1. 参考资料

- RISC-V Assembly Programmer's Manual
- RISC-V Unprivileged Spec
- RISC-V Privileged Spec
- RISC-V 手册（中文）
- 操作系统-linux0.11 进程调度函数分析 - zju_cxl - 博客园 (cnblogs.com)

2. RISC V 通用寄存器列表

Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0, ignores writes	*n/a*
x1	ra	return address for jumps	no
x2	sp	stack pointer	yes
x3	gp	global pointer	*n/a*
x4	tp	thread pointer	*n/a*
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 *or* fp	saved register 0 *or* frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value *or* function argument 0	no
x11	a1	return value *or* function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no
x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 7	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	*(none)*	program counter	*n/a*