Lab 4: RV64 用户态程序

张志心 `3210106357`

1 实验目的

- 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。

2 实验环境

- Same as previous labs.

3 实验步骤

3.1 准备工程

- 将用户态程序 `uapp` 加载至 `.data` 段。按如下修改：

```
...

.data : ALIGN(0x1000){
        _sdata = .;

        *(.sdata .sdata*)
        *(.data .data.*)

        _edata = .;

        . = ALIGN(0x1000);
        _sramdisk = .;
        *(.uapp .uapp*)
        _eramdisk = .;
        . = ALIGN(0x1000);

    } >ramv AT>ram

...
```

- 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
```

- 将 `user` 纳入工程管理。

3.2 创建用户态进程

- 创建用户态进程要对 `sepc` `sstatus` `sscratch` 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：

```c
// proc.h
/* 线程状态段数据结构 */
struct thread_struct {
    uint64 ra;
    uint64 sp;
    uint64 s[12];

    uint64 sepc, sstatus, sscratch;
};

/* 线程数据结构 */
struct task_struct {
    struct thread_info thread_info;
    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

    struct thread_struct thread;

    uint64 satp;
    pagetable_t pgd;
};
```

- 修改 task_init
  - 对于每个进程，初始化：
    - 将 `sepc` 设置为 `USER_START`。
    - 配置 `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）。
    - 将 `sscratch` 设置为 `U-Mode` 的 sp，其值为 `USER_END` （即，用户态栈被放置在 `user space` 的最后一个页面）。
  - 对于每个进程，创建属于它自己的页表。
  - 为了避免 `U-Mode` 和 `S-Mode` 切换的时候切换页表，我们将内核页表 （`swapper_pg_dir` ） 复制到每个进程的页表中。
  - 将 `uapp` 所在的页面映射到每个进行的页表中。注意，在程序运行过程中，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，造成预期外的进程间相互影响。
  - 设置用户态栈。对每个用户态进程，其拥有两个栈： 用户态栈和内核态栈；其中，内核态栈在 `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用户态栈，并映射到进程的页表中。

```c
// proc.c
// for all task[i] :
task[i]->thread.sepc = USER_START;
task[i]->thread.sstatus &= ~(1 << 8);
task[i]->thread.sstatus |= (1 << 5);
task[i]->thread.sstatus |= (1 << 18);
task[i]->thread.sscratch = USER_END;
task[i]->pgd = (unsigned long*)kalloc();
memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
task[i]->thread.sp = (uint64_t)task[i] + PGSIZE;
task[i]->thread_info.kernel_sp = (uint64_t)task[i] + PGSIZE;
task[i]->thread_info.user_sp = (uint64_t)kalloc();

#define STACK_SIZE (PGSIZE << 4)
#define UAPP_SIZE (1 << 24) // 16 MiB

uint64_t va = USER_END - STACK_SIZE;
uint64_t pa = (uint64)(task[i]->thread_info.user_sp) - PA2VA_OFFSET;
create_mapping(task[i]->pgd, va, pa, STACK_SIZE, 0x17);

// #define ELF
#ifndef ELF

#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define CEIL(sza, szb) ((((sza) + ((szb) - 1)) & (~((szb) - 1))) / PGSIZE)
va = USER_START;
int n_pages = 32; // default 32 pages (can be modified)
void *uapp_pt = _sramdisk;
while (n_pages--) {
    uint64_t npa = kalloc();
    uint64_t pa = npa - PA2VA_OFFSET;
    create_mapping(task[i]->pgd, va, pa, PGSIZE, 0x1f);
    memcpy((void *) npa, uapp_pt, PGSIZE);
    uapp_pt += PGSIZE;
    va += PGSIZE;
}
```

修改 `satp`：

```c
uint64 satp = csr_read(satp);
satp = (satp >> 44) << 44;
satp |= ((uint64)(task[i]->pgd) - PA2VA_OFFSET) >> 12;
task[i]->satp = satp;
```

- 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**。

```asm
__switch_to:
    mv t0, a0
    sd ra, 48(t0)
    # ...
    sd s11, 152(t0)
    csrr t1, sepc
```

```
    sd t1,160(a0)
    csrr t1, sstatus
    sd t1,168(a0)
    csrr t1, sscratch
    sd t1,176(a0)
    csrr t1, satp
    sd t1,184(a0)

    mv t1, a1
    ld ra, 48(t1)
    # ...
    ld s11, 152(t1)
    ld t1,160(a1)
    csrw sepc, t1
    ld t1,168(a1)
    csrw sstatus, t1
    ld t1,176(a1)
    csrw sscratch, t1
    ld t1,184(a1)
    csrw satp, t1

    sfence.vma zero, zero
    fence.i
```

3.3 修改中断入口/返回逻辑以及中断处理函数

- 修改 `__dummy`。

```
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0;
    csrwi sepc, 0
    sret
```

- 修改 `_trap` 。

```
_traps:

    # swap sp and sscratch
    csrrw sp, sscratch, sp
    beqz sp, _ignore_switch

    csrrw t0, sscratch, t0
    addi sp, sp, -context_size
    sd t0, 16(sp)
    csrrw t0, sscratch, t0

    j _start_save_context

_ignore_switch:

    # just read back current sp
    csrr sp, sscratch
    addi sp, sp, -context_size
```

```
    sd x0, 16(sp)

_start_save_context:

    csrw sscratch, x0

    sd x1, 8(sp)
    # ...
    sd x31, 248(sp)

    # save sepc
    csrr t0, sepc
    sd t0, 0(sp)
    sd t0, 256(sp)

    csrr t0, scause
    mv a0, t0
    csrr t0, sepc
    mv a1, t0
    mv a2, sp
    call trap_handler

    ld t0, 0(sp)

    # temporarily add 4 to sepc if it's not clock trap
    li t1, 0x8000000000000005
    csrr a0, scause
    beq a0, t1, _csrwrite
    addi t0, t0, 4

_csrwrite:
    csrw sepc, t0

    ld t0, 16(sp)
    addi t1, sp, context_size
    sd t1, 16(sp)
    beqz t0, _restore_cont

    # restore sscratch
    addi sp, sp, context_size
    csrw sscratch, sp
    addi sp, sp, -context_size
    sd t0, 16(sp)

_restore_cont:

    ld x1, 8(sp)
    # ...
    ld x31, 248(sp)

_traps_sret:

    # resume sp
    ld sp, 16(sp)
    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。修改 `trap_handler` 如下：

```c
struct pt_regs {
    uint64 x[32];
    uint64 sepc;
};

void trap_handler(uint64 scause, uint64 sepc, struct pt_regs* regs) {
    if ((long) scause < 0 && (scause & ((1ul << 63) - 1)) == 5) {
        // printk("%s", "Get STI!\n");
        clock_set_next_event();
        do_timer();
        return ;
        // printk("[S] Supervisor Mode Timer Interrupt!\n");
    } else if (scause== 8) {
        syscall(regs);
        return;
    }
    uint64 stval = csr_read(stval);
    printk("Unhandled trap: scause = %lx, sepc = %llx, stval = %llx\n",
scause, sepc, stval);
}
```

3.4 添加系统调用

```c
#define SYS_WRITE   64
#define SYS_GETPID  172

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

3.5 修改 head.S 和 start_kernel

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

```asm
# set sstatus[SIE] = 1
# csrr t0, sstatus
# ori t1, t0, 0x00000002
# csrw sstatus, t1
```

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

```
int start_kernel() {
    printk("2022");
    printk(" Hello RISC-V\n");

    schedule();
    test(); // DO NOT DELETE !!!

    return 0;
}
```

3.6 ELF 文件

```
Elf_Ehdr *header = (void *) _sramdisk;
//  fs_lseek(target_fd, 0, SEEK_SET);
Elf_Half phnum = header->e_phnum;
Elf_Off phoff = header->e_phoff;
while (phnum--) {
    Elf_Phdr *segment = (void *) _sramdisk + phoff;
    phoff += header->e_phentsize;
    printk("Segment at 0x%08x with memory size 0x%06x\n", segment->p_vaddr, segment-
>p_memsz);
    if (!(segment->p_type == PT_LOAD)) {
        printk("Not a PT_LOAD segment, ignoring...\n");
        continue;
    }

    void *file_pt = _sramdisk + segment->p_offset;
    #define MIN(a, b) ((a) < (b) ? (a) : (b))
    #define CEIL(sza, szb) ((((sza) + ((szb) - 1)) & (~((szb) - 1))) / PGSIZE)
    #define ROUNDUP(a, sz)      ((((uint64_t)a) + (sz) - 1) & ~((sz) - 1))
    #define ROUNDDOWN(a, sz)    ((((uint64_t)a)) & ~((sz) - 1))

    // printk("%d\n", CEIL(segment->p_memsz, PGSIZE));
    if ((segment->p_vaddr & (PGSIZE - 1))) {
        printk("Not a page aligned segment! Trying to fix...");
        uint64_t pg_off = segment->p_vaddr & (PGSIZE - 1);
        printk("offset at 0x%08x\n", pg_off);
        uint64_t pg_start = ROUNDDOWN(segment->p_vaddr, PGSIZE);
        uint64_t n_pgpa = kalloc();
        create_mapping(task[i]->pgd, pg_start, n_pgpa - PA2VA_OFFSET, PGSIZE, 0x1f);
        memset((void *) (n_pgpa + pg_off), 0, MIN(PGSIZE - pg_off, segment-
>p_memsz));
        memcpy((void *) (n_pgpa + pg_off), file_pt, MIN(PGSIZE - pg_off, segment-
>p_filesz));
        file_pt += MIN(PGSIZE - pg_off, segment->p_filesz);
        segment->p_vaddr = ROUNDUP(segment->p_vaddr, PGSIZE);
        segment->p_memsz -= MIN(PGSIZE - pg_off, segment->p_memsz);
        segment->p_filesz -= MIN(PGSIZE - pg_off, segment->p_filesz);
        printk("Rest at %x with size 0x%06x\n", segment->p_vaddr, segment->p_memsz);
    }
```

```
    int j = i;
    for (int i = 0; i < CEIL(segment->p_memsz, PGSIZE); i++) {
        // printk("%d\n", i);
        uint64_t n_pgpa = kalloc();
        create_mapping(task[j]->pgd, segment->p_vaddr + i * PGSIZE, n_pgpa -
PA2VA_OFFSET, PGSIZE, 0x1f);

        printk("Memset: 0x%08x bytes\n", MIN(PGSIZE, segment->p_memsz - PGSIZE *
i));
        memset((void *) n_pgpa, 0, MIN(PGSIZE, segment->p_memsz - PGSIZE * i));
        //Avoid negative number!!!
        printk("Read from elf: 0x%08x bytes\n", MIN(PGSIZE, segment->p_filesz -
MIN(PGSIZE * i, segment->p_filesz)));
        memcpy((void *) n_pgpa, file_pt, MIN(PGSIZE, segment->p_filesz - MIN(PGSIZE
* i, segment->p_filesz)));
        file_pt += MIN(PGSIZE, segment->p_filesz - MIN(PGSIZE * i, segment-
>p_filesz));
    }
    printk("Load finished\n");
}
```

4 运行结果

4.1 普通二进制文件

在 `proc.c` 中删除 `#define ELF` ，并且在 `uapp.S` 中写入 `.incbin "uapp.bin"`。

运行 `make run` 得到结果如下：

```
...setup_vm done!
...buddy_init done!
Entering task init
...proc_init done!
2022 Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
switch to [PID = 4 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
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
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
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]
SET [PID = 4 COUNTER = 4]
switch to [PID = 1 COUNTER = 1]
switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3
switch to [PID = 4 COUNTER = 4]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 3 COUNTER = 10]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.4
SET [PID = 1 COUNTER = 10]
SET [PID = 2 COUNTER = 10]
SET [PID = 3 COUNTER = 5]
SET [PID = 4 COUNTER = 2]
switch to [PID = 4 COUNTER = 2]
switch to [PID = 3 COUNTER = 5]
```

4.2 ELF 文件

在 `proc.c` 中写入 `#define ELF` ，并且在 `uapp.S` 中写入 `.incbin "uapp.elf"`。

运行 `make run` 得到结果如下：

```
Entering task init
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000b68
Memset: 0x00000b68 bytes
Read from elf: 0x00000774 bytes
Load finished
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000b68
Memset: 0x00000b68 bytes
Read from elf: 0x00000774 bytes
Load finished
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000b68
Memset: 0x00000b68 bytes
Read from elf: 0x00000774 bytes
Load finished
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
Segment at 0x00000000 with memory size 0x00000b68
Memset: 0x00000b68 bytes
Read from elf: 0x00000774 bytes
Load finished
Segment at 0x00000000 with memory size 0x00000000
Not a PT_LOAD segment, ignoring...
```

```
...proc_init done!
2022 Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003fffffffe0, this is print No.1
switch to [PID = 4 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.1
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
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
SET [PID = 1 COUNTER = 1]
SET [PID = 2 COUNTER = 4]
SET [PID = 3 COUNTER = 10]
SET [PID = 4 COUNTER = 4]
switch to [PID = 1 COUNTER = 1]
switch to [PID = 2 COUNTER = 4]
[U-MODE] pid: 2, sp is 0000003fffffffe0, this is print No.3
switch to [PID = 4 COUNTER = 4]
[U-MODE] pid: 4, sp is 0000003fffffffe0, this is print No.2
switch to [PID = 3 COUNTER = 10]
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.3
[U-MODE] pid: 3, sp is 0000003fffffffe0, this is print No.4
```

5 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。

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

   两者大小不同主要是因为未初始化的数据（bss段）不需要被实际保存在可执行文件中，所以 `p_filesz` 会小于 `p_memsz`。

   还有可能是因为在可执行文件中需要对段段大小进行对齐填充，所以和实际数据大小不同。

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?

因为每一个物理内存通过映射到虚拟内存空间，每个用户现成都可以看成是独享了一个独立的地址空间，所以可以用同样的虚拟内存地址，但它们映射到的物理内存是不一样的。

用户可以查看 `proc/$pid/pagemap` 得知栈所在的物理地址。例如如下例子（**arm Ubuntu**）：

pac@ubuntu: ~/tmp

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdint.h>
int main(){
    int pid = getpid();
    char pathname[1000];
    sprintf(pathname, "/proc/%d/pagemap", pid);
    int pgmap_fd = open(pathname, O_RDONLY);
    uintptr_t sp;
    __asm__ ("mov %[sp], sp" : [sp] "=r"(sp));
    printf("0x%016lx\n", sp);
    lseek(pgmap_fd, (sp >> 12) << 3, SEEK_SET);
    uint64_t pte;
    read(pgmap_fd, &pte, 8);
    printf("%lx\n", (pte & ((1ul << 55) - 1)) | (sp & ((1ul << 12) - 1)));
}
```