LAB4 RV64 用户态程序

3210105488 刘扬

1 实验过程

1.1 准备工作

修改`vmlinux.lds`,`defs.h`,`MakeFile`，从`repo`同步文件。不赘述。

1.2 创建用户态进程

1.2.1 修改`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;
};
```

这里直接在`task_struct`中存储了`satp`，方便后续`__switch_to`实现切换页表时不再做`satp`的运算。

1.2.2 修改`task_init`

- 对于每个进程，初始化我们刚刚在`task_init`中添加的三个变量。具体而言：
  - 将 `sepc` 设置为 `USER_START`。

    ```
    task[i]->thread.sepc = 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）。

```
unsigned long sstatus = csr_read(sstatus);
sstatus &= ~(1 << 8); // sstatus[SPP] = 0
sstatus |= 1 << 5; // sstatus[SPIE] = 1
sstatus |= 1 << 18; // sstatus[SUM] = 1
```

  - 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）。

```
task[i]->thread.sscratch = USER_END;
```

- 对于每个进程，创建属于它自己的页表。

```
pagetable_t pgtbl = (pagetable_t)kalloc();
```

- 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。

```
for(int j = 0;j < PGSIZE / sizeof(uint64*);j++){
            pgtbl[j] = swapper_pg_dir[j];
}
```

- 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。

```
uint64 num_pages = ((uint64)_eramdisk - (uint64)_sramdisk) / PGSIZE;
if (num_pages * PGSIZE < ((uint64)_eramdisk - (uint64)_sramdisk)){
    num_pages++;
}
char *_uapp = (char *)alloc_pages(num_pages);
for(int j = 0;j < ((uint64)_eramdisk - (uint64)_sramdisk);j++){
    _uapp[j] = _sramdisk[j];
}
unsigned long va = USER_START;
unsigned long pa = (unsigned long)(_uapp) - PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, num_pages * PGSIZE, 31);
```

  这里发现之前的`create_mapping`有点小bug，因为这里的`num_pages*PGSIZE`刚好是一个页的大小，之前我在`create_mapping`中计算循环次数时始终加1:`int i = sz / PGSIZE + 1;`，会导致映射出错，修改一下这段逻辑即可。

- 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。

```
va = USER_END - PGSIZE;
pa = task[i]->thread_info.user_sp - PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, PGSIZE, 23);
```

- 最后算出`satp`

```
uint64 satp = (uint64)0x8 << 60;
satp |= ((uint64)(pgtbl) - PA2VA_OFFSET) >> 12;
task[i]->satp = satp;
```

1.2.3 修改`__switch_to`

需要加入：保存/恢复 `sepc` `sstatus` `sscratch` 以及 切换页表的逻辑。

切换页表的逻辑直接通过压栈和出栈`task_struct`的`satp`得到，因为`satp`刚好在`thread_struct`的后面，因此在地址上是连续的，不用再额外计算偏移量。

```
__switch_to:
    sd ra, 48(a0) # Offset of thread_struct = 48
    ...
    sd s11, 152(a0)
    csrr t0, sepc
    sd t0, 160(a0)
    csrr t0, sstatus
    sd t0, 168(a0)
    csrr t0, sscratch
    sd t0, 176(a0)
    csrr t0, satp
    sd t0, 184(a0)
    # load next ra,sp,s0-s11
    ld ra, 48(a1)
    ...
    ld t0, 160(a1)
    csrw sepc, t0
    ld t0, 168(a1)
    csrw sstatus, t0
    ld t0, 176(a1)
    csrw sscratch, t0
    ld t0, 184(a1)
    csrw satp, t0
```

1.3 修改中断入口/返回逻辑 (_trap) 以及中断处理函数 （ trap_handler)

1.3.1 修改`__dummy`

```
#__dummy
    .globl __dummy
__dummy:
    csrr t0, sscratch
    csrw sscratch, sp
    mv sp, t0
    sret
```

通过`sscratch`读取将要切换的用户栈指针，并将当前`sp`放入`sscratch`中。通过`sret`返回至用户态进程。

1.3.2 修改`_traps`

和修改`__dummy`同理，在`_trap`的首尾通过`sscratch`读取将要切换的用户栈指针，并将当前`sp`放入`sscratch`中。**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。**

```
_traps:
    csrr t0, sscratch
    beq t0, x0, _traps_save
    csrw sscratch, sp
    mv sp, t0
_traps_save:
    # 1. save 32 registers and sepc to stack
    csrr t0, sstatus
    sd t0, -8(sp)
    csrr t0, sepc
    ...
    sd x0, -272(sp)
    addi sp, sp, -272
    # 2. call trap_handler
    csrr a0, scause
    csrr a1, sepc
    mv a2, sp
    call trap_handler
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld x0, 0(sp)
    ...
    ld t0, 256(sp)
    csrw sepc, t0
    ld t0, 264(sp)
    csrw sstatus, t0
    ld x2, 16(sp)
    # 4. return from trap
    csrr t0, sscratch
    beq t0, x0, _traps_sret
    csrw sscratch, sp
    mv sp, t0
_traps_sret:
    sret
```

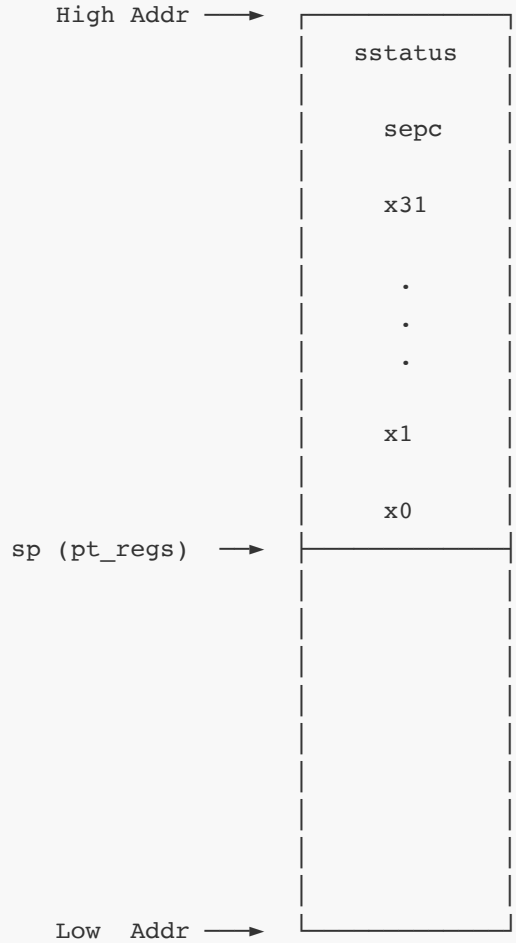
1.3.3 修改`trap_handler`

- `trap_handler`中需要额外传入`pt_regs`做`syscall`的处理。

```
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs)
```

`pt_regs`结构体是按照压栈顺序定义的。

我的栈空间如下：



因此`pt_regs`定义如下：

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

- 在`trap_handler`中加入处理syscall的逻辑

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Available 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	*Reserved*
0	11	Environment call from M-mode
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	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

查表可知`Environment call from U-mode`是8，因此`trap_handler`如下：

```
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs) {
    if(scause == 0x8000000000000005){
        //printk("[S] Supervisor Mode Timer Interrupt\n");
        clock_set_next_event();
        do_timer();
        return;
    }
    else if(scause == 8){
        syscall(regs);
        return;
    }
    return;
}
```

1.3.4 增加`syscall.c`

要注意针对系统调用这一类异常， 我们需要手动将 `sepc + 4`

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

1.3.5 测试纯二进制文件

```
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.1
switch to [PID = 4 PRIORITY = 66 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.2
switch to [PID = 3 PRIORITY = 52 COUNTER = 8]
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.2
switch to [PID = 2 PRIORITY = 88 COUNTER = 9]
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.2
```

1.4 添加ELF支持

1.4.1 准备工作

首先将 `uapp.S` 中的 payload 给换成ELF 文件。不赘述。

1.4.2 编写 `load_program`

load_program声明如下：

```
static uint64 load_program(struct task_struct* task, pagetable_t pgtbl);
```

实验文档中给出框架如下：

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

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

    Elf64_Phdr* phdr;
    int load_phdr_cnt = 0;
    for (int i = 0; i < phdr_cnt; i++) {
        phdr = (Elf64_Phdr*)(phdr_start + sizeof(Elf64_Phdr) * i);
        if (phdr->p_type == PT_LOAD) {
            // alloc space and copy content
            // do mapping
            // code...
        }
    }

    // allocate user stack and do mapping
    // code...

    // following code has been written for you
    // set user stack
    ...;
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    // sstatus bits set
    task->thread.sstatus = ...;
    // user stack for user program
    task->thread.sscratch = ...;
}
```

因此我们需要做的工作是在框架的基础上：

- 分配 `uapp` 的物理内存空间，并做页表映射。

  这里的逻辑和加载纯二进制文件比较相似，**需要注意的点是要做地址对齐**，否则会有page fault。同时记得要将 `[p_vaddr + p_filesz, p_vaddr + p_memsz)` 对应的物理区间清零。

```
if (phdr->p_type == PT_LOAD) {
    // alloc space and copy content
    uint64 num_pages = (phdr->p_memsz) / PGSIZE;
    if (num_pages * PGSIZE < phdr->p_memsz){
        num_pages++;
    }
    char *start_addr = (char *)(_sramdisk + phdr->p_offset);
    char *alloc_mem = (char *)alloc_pages(num_pages);
    for(int j = 0;j < phdr->p_memsz;j++){
        alloc_mem[j] = start_addr[j];
    }
    // clear
    memset((char *)((uint64)alloc_mem + phdr->p_filesz), 0, phdr->p_memsz - phdr-
>p_filesz);
    // allign
    uint64 va = phdr->p_vaddr;
    uint64 offset = va % PGSIZE;
    for(int i = num_pages * PGSIZE - offset - 1;i >= 0;i--){
        alloc_mem[i + offset] = alloc_mem[i];
    }
    memset(alloc_mem, 0, offset);
    va = va - offset;
    uint64 pa = (uint64)alloc_mem - PA2VA_OFFSET;
    create_mapping(pgtbl, va, pa, num_pages * PGSIZE, 31);
}
```

- 设置`user_sp`,`sepc`,`sstatus`,`sscratch`，对用户栈做页表映射。

  和加载纯二进制文件的做法相同，区别在于`sepc`是从`ehdr->e_entry`中得到。

```
// set user stack
task->thread_info.user_sp = kalloc();
// pc for the user program
task->thread.sepc = ehdr->e_entry;
// sstatus bits set
unsigned long sstatus = csr_read(sstatus);
sstatus &= ~(1 << 8); // sstatus[SPP] = 0
sstatus |= 1 << 5; // sstatus[SPIE] = 1
sstatus |= 1 << 18; // sstatus[SUM] = 1
task->thread.sstatus = sstatus;
// set sscratch
task->thread.sscratch = USER_END;

// user stack for user program
uint64 va = USER_END - PGSIZE;
uint64 pa = task->thread_info.user_sp - PA2VA_OFFSET;
create_mapping(pgtbl, va, pa, PGSIZE, 23);
```

1.4.3 加载ELF文件结果

```
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.1
switch to [PID = 4 PRIORITY = 66 COUNTER = 5]
[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 4, sp is 0000003ffffffe0, this is print No.2
switch to [PID = 3 PRIORITY = 52 COUNTER = 8]
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.2
switch to [PID = 2 PRIORITY = 88 COUNTER = 9]
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.1
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.2
SET [PID = 4 PRIORITY = 66 COUNTER = 1]
SET [PID = 3 PRIORITY = 52 COUNTER = 4]
SET [PID = 2 PRIORITY = 88 COUNTER = 10]
SET [PID = 1 PRIORITY = 37 COUNTER = 4]
switch to [PID = 4 PRIORITY = 66 COUNTER = 1]
switch to [PID = 1 PRIORITY = 37 COUNTER = 4]
[U-MODE] pid: 1, sp is 0000003ffffffe0, this is print No.2
switch to [PID = 3 PRIORITY = 52 COUNTER = 4]
[U-MODE] pid: 3, sp is 0000003ffffffe0, this is print No.3
switch to [PID = 2 PRIORITY = 88 COUNTER = 10]
[U-MODE] pid: 2, sp is 0000003ffffffe0, this is print No.3
```

2 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）

多对一。我们在实验中创建了一个内核态线程用于做系统启动，页表映射，处理中断异常和调度用户态进程。而用户态线程被创建了多个，独立执行。

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

参考elf - Difference between p_filesz and p_memsz of Elf32_Phdr - Stack Overflow

`p_filesz`：此字段（8 字节）给出本段内容在文件中的大小，单位是字节，可以是 0。

`p_memsz`：此字段（8 字节）给出本段内容在内容镜像中的大小，单位是字节，可以是 0。

二者的区别在于段的文件映像是指存储在文件中的段的部分。段的内存映像是指在程序执行期间加载到内存中的段的部分。`p_memsz`大于`p_filesz`的原因是，可加载段可能包含`.bss`节，该节包含未初始化的数据。将此数据存储在磁盘上会很浪费，因此，仅在ELF文件加载到内存后才占用空间。

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？

因为每个线程都有自己的页表，就算是相同的虚拟地址，经过页表映射后对应的物理地址也是不同的。

用户态没有权限去walk page table，所以无法得到物理地址。但是内核给我们提供了一个接口，在`/proc/pid/`下面有个文件叫pagemap，它给每个page生成了一个64bit的描述符，来描述虚拟地址这一页对应的物理页帧。

参考Documentation/admin-guide/mm/pagemap.rst · 9bf14b09c69d69b241709c63b2f6ed82f0db6718 · Kali Linux / Packages / linux · GitLab

`/proc/pid/pagemap`. This file lets a userspace process find out which physical frame each virtual page is mapped to. It contains one 64-bit value for each virtual page, containing the following data (from `fs/proc/task_mmu.c`, above pagemap_read):

> - Bits 0-54 page frame number (PFN) if present
> - Bits 0-4 swap type if swapped
> - Bits 5-54 swap offset if swapped
> - Bit 55 pte is soft-dirty (see :ref:`Documentation/admin-guide/mm/soft-dirty.rst <soft_dirty>`)
> - Bit 56 page exclusively mapped (since 4.2)
> - Bits 57-60 zero
> - Bit 61 page is file-page or shared-anon (since 3.5)
> - Bit 62 page swapped
> - Bit 63 page present

在计算物理地址时，只需要找到虚拟地址在`pagemap`的对应项，再通过对应项中的Bit 63判断此物理页是否在内存中，若在内存中则对应项中的物理页号(Bits 0 - 54)加上偏移地址，就能得到物理地址。