Operating system

Lab 2

Author: 吴杭 李克成

Date: 2024/12/13

Autumn-Fall 2024-2025 Semester

Table of Contents

Chapter 1: 实验流程

1.1) 结构体更新

按照文档上的更新即可，同时为了后续更换进程方便，我们在 `thread_struct` 中增加了 `satp` 寄存器的值。

1.2) 修改 `task_init`

对于每个用户态线程，先通过 load elf 文件的方法（在报告的后续章节介绍）将其从 `_sramdisk` 到 `_eramdisk` 的数据拷贝到一块新的内存中，从而保证每个线程的栈是分离的。然后设置他们的 satp 寄存器，并且分配好他们的栈空间，建立虚拟地址到物理地址的映射，写入到页表中。大致代码如下：

```
for (int i = 1; i < NR_TASKS; i++) {
  struct task_struct *new_task = (struct task_struct *)kalloc();
  new_task->state = TASK_RUNNING;
  new_task->pid = i;

  new_task->counter = 0;
  new_task->priority =
      rand() % (PRIORITY_MAX - PRIORITY_MIN + 1) + PRIORITY_MIN;

  new_task->thread.ra = (uint64_t)__dummy;
  new_task->thread.sp =
      (uint64_t)new_task + PGSIZE; // notice new_task is also an address of
                                   // the struct (the bottom of the PAGE)

  Elf64_Ehdr *elf_header = (Elf64_Ehdr *)_sramdisk;
  new_task->thread.sepc = elf_header->e_entry;

  new_task->thread.sstatus = SUM_BIT;
  new_task->thread.sscratch = USER_END;

  new_task->pgd = alloc_page();

  memcpy(new_task->pgd, swapper_pg_dir, PGSIZE);

  uint64_t ppn = VA2PA((uint64_t)(new_task->pgd)) >> 12;
  new_task->thread.satp = ppn | (SATP_MODE_SV39 << 60);

  load_elf(new_task);

  char *stack_umode = alloc_page();

  create_mapping(new_task->pgd, PGROUNDDOWN(USER_END - 1),
                 VA2PA((uint64_t)stack_umode), PGSIZE,
                 PRIV_U | PRIV_W | PRIV_R | PRIV_V);

  task[i] = new_task;
}
```

1.3) 修改__switch_to

因为我们在线程的数据结构里面额外增加了一些 csr 寄存器，所以在 switch 的时候需要在原来的基础上再 store 和 load 这些寄存器，并且因为重新设置了 satp，所以需要刷新 TLB。增加的代码如下：

```
    ...
    csrr t0, sepc
    sd t0, 144(a0)
    csrr t0, sstatus
    sd t0, 152(a0)
    csrr t0, sscratch
    sd t0, 160(a0)
    csrr t0, satp
    sd t0, 168(a0)

    ...
    csrw sepc, t0
    ld t0, 152(a1)
    csrw sstatus, t0
    ld t0, 160(a1)
    csrw sscratch, t0
    ld t0, 168(a1)
    csrw satp, t0
    # flush TLB
    sfence.vma zero, zero
    ret
```

1.4) 更新中断处理逻辑

1.4.1) 修改 __dummy

在切换 `thread_struct.sp` 和 `thread_struct.sscratch` 的时候使用 `csrrw sp, sscratch, sp` 指令即可。

```
__dummy:
    csrrw sp, sscratch, sp
    sret # the program will return to User-mode program
```

1.4.2) 修改__traps

在切换 `thread_struct.sp` 和 `thread_struct.sscratch` 的时候使用 `csrrw sp, sscratch, sp` 指令即可，然后比较 `sp` 的值是否等于 0，如果为 0，说明此时就是在内核态，不应当切换，所以重新用 `csrrw sp, sscratch, sp` 将 sp 切换回来。如果不等于 0，说明此时在用户态，就直接跳转到后续的逻辑即可。

```
_traps:
    csrrw sp, sscratch, sp
    bne sp, zero, _traps_real
    csrrw sp, sscratch, sp
_traps_real:
    addi sp, sp, -36*8 # x0 is not saved
    sd ra, 0(sp)
```

```
    sd sp, 8(sp)
    sd gp, 16(sp)
    ...
    ld gp, 16(sp)
    ld ra, 0(sp)
    ld sp, 8(sp)
    addi sp, sp, 36*8
    csrrw sp, sscratch, sp
    bne sp, zero, __return
    csrrw sp, sscratch, sp
```

1.4.3) 修改 `trap_handler`

这里主要是要增加获取到 environment call from U-mode 的时候，调用 syscall 的逻辑。首先通过判断 `scause` 的值是否为 8，判断是否为 environment call from U-mode ，如果是的话，再通过 traps 中存下的用户态的 `a7` 寄存器的值判断是哪种系统调用（从用户态的代码可以看出调用 `ecall` 之前是把 SYS_CALL 的类型填入到 `a7` 中的，而这个值在我们的`_traps` 处理中，被存在了栈的某个位置，所以可以根据 `sp` 的位置去访问，在代码中被转换成了一个 `general_regs` 数组）。然后调用相关的函数即可。最后需要把 `sepc` 的值加上 4，因为此时 `sepc` 的值是 `ecall` 的值，但是和其他由于时钟中断等类型不同，我们可以认为 `ecall` 这个指令已经执行完了，所以需要把 `sepc` 加上 4，否则就会再次调用 `ecall`，触发了相同的中断。

```
    if (scause == 8) {                    // environment call from U-mode
      if (regs->general_regs[16] == 172) { // a7 == SYS_GETPID
        regs->general_regs[9] = getpid();
      } else if (regs->general_regs[16] == 64) { // a7 == 64
        regs->general_regs[9] =
            write(regs->general_regs[9], (char *)regs->general_regs[10],
                  regs->general_regs[11]);
      }
      regs->sepc += 4;
      return;
    }
```

1.5) 增加系统调用

这小节的部分内容在前面修改 `trap_handler` 中已经提及，这里补充对 `getpid()`和 `write()` 的介绍。

1.5.1) `getpid`

这个函数直接返回 current 线程的 pid 值即可，我们在初始化的时候就把 pid 的值存在了 `task_struct` 中。

```
uint64_t getpid(){
    return current->pid;
}
```

1.5.2) write

这个函数需要调用 printk 函数，具体而言，我们根据得到的 buf（被打印字符串的起始地址），以及 count（被打印字符串的长度），逐个输出字符即可。

```
uint64_t write(uint64_t fd, const char* buf, uint64_t count){
    // print buf
    if (fd == 1){
        for (int i = 0; i < count; i++){
            printk("%c", buf[i]);
        }
    }
    else {
        printk("fd != 1");
    }
    // return the number of characters that have been printed
    return count;
}
```

1.6) 调整时钟中断

按照实验文档上说的，在 start_kernel()中，tests()之前调用 schedual()，然后在 head.S 中注释掉 sstatus.SIE 的逻辑。

1.7) 添加 ELF 解析与加载

在程序的_sramdisk 到_eramdisk 这段内容中存储的是我们的 ELF 文件格式 load 进来的 uapp 程序。我们首先把_sramdisk 这个地址转化成 Elf64_Ehdr *类型，也就是我们的 elf 文件头结构，其中的 e_phoff 表示的是 elf 文件头结构到第一个程序头结构（Elf64_Phdr *）的偏移量。我们可以根据这个得到第一个程序头 program_header_start，然后我们根据 elf 文件头中储存的整个程序被分成的段数，也就是 e_phnum，来遍历所有的段。对于每一个段，我们再根据这个段的 p_type 来判断是否为 PT_LOAD 来判断是否需要把他 load 到分配给用户态的内存中。如果需要的话，我们再根据 program_header 中记录的 p_vaddr（也就是这个段的起始虚拟地址）以及 p_mem（也就是这个段的内存占用大小）来计算需要分配给他的 pages 数量（这里需要注意 p_vaddr 和页边界不一定是对齐的，所以要处理 shift，也就是在复制内容的时候需要从页边界加上一个 shift 的位置开始）。然后根据 program_header 中的 p_flags 来判断来赋予这个页的权限即可。

```
static void load_elf(struct task_struct *new_task) {
  Elf64_Ehdr *elf_header = (Elf64_Ehdr *)_sramdisk;
  // find the program header
  Elf64_Phdr *program_header_start =
      (Elf64_Phdr *)(_sramdisk + elf_header->e_phoff);
  for (int i = 0; i < elf_header->e_phnum; ++i) {
    // enumerate all the program headers
    Elf64_Phdr *program_header = program_header_start + i;
    if (program_header->p_type == PT_LOAD) { // loadable
      // align the vaddr to the page size
      uint64_t shift = program_header->p_vaddr % PGSIZE;
      uint64_t pages = (shift + program_header->p_memsz + PGSIZE - 1) / PGSIZE;
```

```
            // allocate pages for the uapp
            char *uapp_space = alloc_pages(pages);
            memcpy(uapp_space + shift, _sramdisk + program_header->p_offset,
                   program_header->p_memsz);
            uint64_t priv = program_header->p_flags;
            uint64_t priv_r = priv & PF_R ? PRIV_R : 0;
            uint64_t priv_w = priv & PF_W ? PRIV_W : 0;
            uint64_t priv_x = priv & PF_X ? PRIV_X : 0;
            // create the address mapping for uapp
            create_mapping(new_task->pgd, PGROUNDDOWN(program_header->p_vaddr),
                           VA2PA((uint64_t)uapp_space), pages << 12,
                           PRIV_U | priv_w | priv_x | priv_r | PRIV_V);
        }
    }
}
```

Chapter 2: 思考题

2.1) 1.

我们的用户态线程和内核态线程是一对一的关系，因为我们每创建一个用户态的线程，都会给他记录两个栈（分别记录在 thread_struct 中的 sp 和 sscratch 中，在 switch 和中断处理的时候会切换），一个用户态栈的，一个内核态栈，在处理中断的时候，就会切换到这个用户态线程对应的内核态线程进行处理。

2.2) 2.

系统调用的时候所用的通用寄存器是此时内核态下的通用寄存器。在处理完中断之后会把用户态的寄存器从栈中重新加载出来，之前的通用寄存器的值会被覆盖，所以不能直接写到通用寄存器中，而是需要保存到将要被 load 出来的用户态寄存器中。

2.3) 3.

因为此时 `sepc` 的值是 `ecall` 的值，但是和其他由于时钟中断等类型不同，我们可以认为 `ecall` 这个指令已经执行完了，所以需要把 `sepc` 加上 4，这样在 `sret` 的时候会返回到 `ecall` 的下一条指令。否则就会再次调用 `ecall`，触发了相同的中断。

2.4) 4.

`p_filesz` 表示的是 segment 在文件中占的大小，而 `p_memsz` 表示的是 segment 在内存中占的大小。对于 `.bss` 段的数据，在文件中实际上并不真实存在，因为他只包含未初始化或者初始化为零的全局变量等，但是加载到内存中的时候需要给他们分配真实的大小，所以 `p_memsz` 会比 `p_filesz` 要来的大。参考 https://ctf-wiki.org/executable/elf/structure/basic-info/

2.5) 5.

多个进程的栈虚拟地址是相同的不会有任何问题，因为首先他们对应的物理地址不同，其次每个进程都有专属的页表，所以不会出现不同进程相同的虚拟地址映射到相同物理地址的情况。

用户态不应当有常规的方法获取自己的栈的物理地址的位置，否则就是违反了虚拟地址的设计初衷，也许会存在一些安全性上的问题。

Chpater 3: 心得体会

3.1) 遇到的问题

一开始在加载二进制程序的时候认为`_sramdisk`到`_eramdisk`中的内容完全是uapp的代码段，所以`create_mapping`的时候没有加上写入的权限，但是通过gdb调试的时候发现，在用户态程序的时候会发生写入的操作，如果不把权限设置为可写的话就会发生中断了。

```
Dump of assembler code from 0x0 to 0x100:
   0x0000000000000000:  j       0x38
   0x0000000000000004:  addi    sp,sp,-32
   0x0000000000000008:  sd      s0,24(sp)
   0x000000000000000c:  addi    s0,sp,32
   0x0000000000000010:  ld      a5,-24(s0)
   0x0000000000000014:  li      a7,172
   0x0000000000000018:  ecall
   0x000000000000001c:  mv      a5,a0
   0x0000000000000020:  sd      a5,-24(s0)
   0x0000000000000024:  ld      a5,-24(s0)
   0x0000000000000028:  mv      a0,a5
   0x000000000000002c:  ld      s0,24(sp)
   0x0000000000000030:  addi    sp,sp,32
   0x0000000000000034:  ret
   0x0000000000000038:  addi    sp,sp,-32
   0x000000000000003c:  sd      ra,24(sp)
   0x0000000000000040:  sd      s0,16(sp)
   0x0000000000000044:  addi    s0,sp,32
   0x0000000000000048:  auipc   ra,0x0
   0x000000000000004c:  jalr    -68(ra) # 0x4
   0x0000000000000050:  mv      a1,a0
   0x0000000000000054:  mv      a2,sp
=> 0x0000000000000058:  auipc   a5,0x1
   0x000000000000005c:  addi    a5,a5,564 # 0x128c
   0x0000000000000060:  lw      a5,0(a5)
   0x0000000000000064:  addiw   a5,a5,1
   0x0000000000000068:  sext.w  a4,a5
   0x000000000000006c:  auipc   a5,0x1
   0x0000000000000070:  addi    a5,a5,544 # 0x128c
   0x0000000000000074:  sw      a4,0(a5)
   0x0000000000000078:  auipc   a5,0x1
   0x000000000000007c:  addi    a5,a5,532 # 0x128c
   0x0000000000000080:  lw      a5,0(a5)
   0x0000000000000084:  mv      a3,a5
   0x0000000000000088:  auipc   a0,0x1
   0x000000000000008c:  addi    a0,a0,384 # 0x1208
   0x0000000000000090:  auipc   ra,0x1
   0x0000000000000094:  jalr    -256(ra) # 0xf90
   0x0000000000000098:  sw      zero,-20(s0)
   0x000000000000009c:  j       0xac
   0x00000000000000a0:  lw      a5,-20(s0)
   0x00000000000000a4:  addiw   a5,a5,1
   0x00000000000000a8:  sw      a5,-20(s0)
   0x00000000000000ac:  lw      a5,-20(s0)
   0x00000000000000b0:  sext.w  a4,a5
   0x00000000000000b4:  lui     a5,0x50000
   0x00000000000000b8:  addi    a5,a5,-2 # 0x4ffffffe
   0x00000000000000bc:  bgeu    a5,a4,0xa0
   0x00000000000000c0:  j       0x48
```

后面通过和同学交流得知，这段内容还应当包括用户态程序的数据段，所以写入其实是发生在数据段的，所以确实应该加上写入的权限（虽然为了方便我把代码段的写入权限也加上了，没有判断哪里是数据段的起始位置）。

另外我由于没有认真阅读文档，导致我以为 SPP 这个 bit 需要设置为 1 才能在 sret 的时候返回用户态，实际上应当设置为 0。这一点在后续的 debug 中解决了，设置为 1 的时候一旦进入用户态程序就报 instruction page fault 的 exception。

3.2) 心得体会

这次的实验让我更加深入理解了用户态程序和内核态程序之间的隔离，尤其是解决了我一个之前的疑惑（“为什么用户态程序都有相同的虚拟地址而不会冲突”，答案在思考题中）。同时我觉得这次的实验文档写的比 lab3 的更加详细，实验的体验感更好。

Declaration

We hereby declare that all the work done in this lab 3 is of our independent effort.