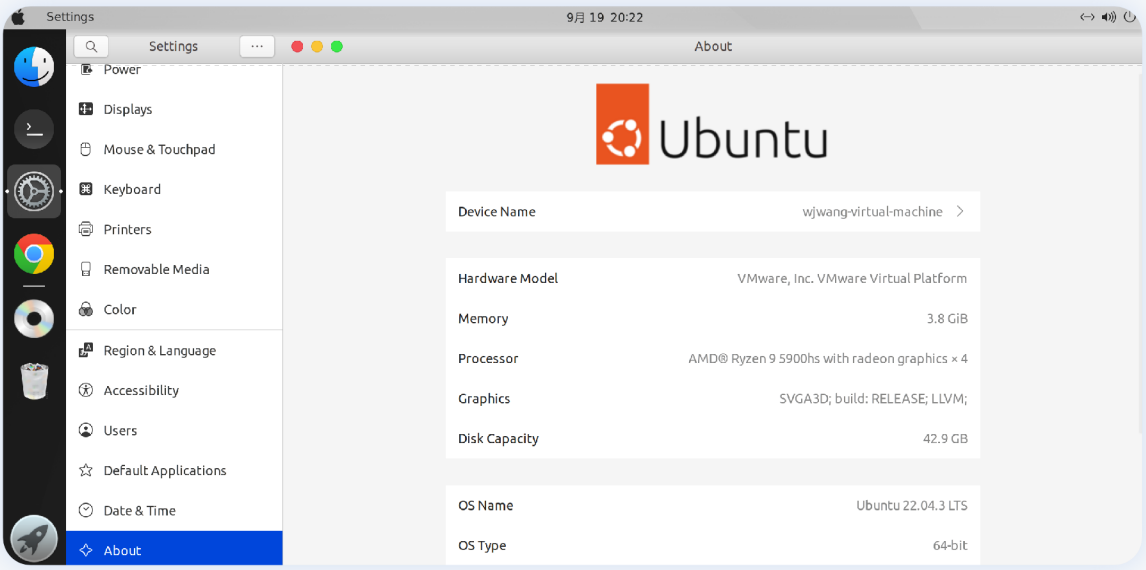
Lab 4 RV64 用户态程序

学号：3210106034
姓名：王伟杰

实验环境：

```
uname -a
Linux wjwang-virtual-machine 6.2.0-33-generic #33~22.04.1-Ubuntu SMP
PREEMPT_DYNAMIC Thu Sep  7 10:33:52 UTC 2 x86_64 x86_64 x86_64
GNU/Linux
```



准备工程

`git clone` 仓库中的代码，合并到之前完成的框架中。修改 `vmlinux.lds` 如下：

```
...

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

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

        _edata = .;

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

    } >ramv AT>ram

...
```

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
```

修改底层 `Makefile` 文件，将 `user` 纳入工程管理

```
all: clean
    ${MAKE} -C lib all
    ${MAKE} -C test all
    ${MAKE} -C init all
    ${MAKE} -C user all # 纳入管理
    ${MAKE} -C arch/riscv all
    @echo -e '\n'Build Finished OK
```

创建用户态进程

准备

修改 `proc.h` 中的 `NR_TASKS=(1+3)`

由于创建用户态进程要对 `sepc` `sstatus` `sscratch` 做设置，我们将其加入 `thread_struct` 中。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。根据提示，由于 `thread_struct` 可以删除，这里我们删掉这个数据结构，并修改 `entry.S` 中的 `__switch_to` ，将跳过的位数从48(6个uint64)改为32(4个uint64)，修改 `task_struct` 如下

```
// proc.h

typedef unsigned long* pagetable_t;

struct thread_struct {
    uint64_t ra;
    uint64_t sp;
    uint64_t s[12];

    uint64_t sepc, sstatus, sscratch;
};

struct task_struct {
    // struct thread_info* thread_info;
    uint64_t state;
    uint64_t counter;
    uint64_t priority;
    uint64_t pid;

    struct thread_struct thread;

    pagetable_t pgd;
};
```

`entry.S` :

```
__switch_to:
    # ...
    add t0, a0, 32

    # ...
    add t0, a1, 32

    # ...
    ret
```

修改 task_init

对于每个进程，初始化我们刚刚在 thread_struct 中添加的三个变量。具体而言：

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），所以spp = 0, spie = 1, 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）。

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。

```
1  extern char _sramdisk[], _eramdisk[];
2
3  void task_init() {
4      test_init(NR_TASKS);
5
6      idle = (struct task_struct*)kalloc();
7      idle->state = TASK_RUNNING;
8      idle->counter = 0;
9      idle->priority = 0;
10     idle->pid = 0;
11     current = idle;
12     task[0] = idle;
13
14     for(int i = 1; i < NR_TASKS; ++i){
15         task[i] = (struct task_struct*)kalloc();
16         task[i]->state = TASK_RUNNING;
17         task[i]->counter = task_test_counter[i];
18         task[i]->priority = task_test_priority[i];
19         task[i]->pid = i;
20         task[i]->thread.ra = (uint64)__dummy;
21         task[i]->thread.sp = (uint64)task[i] + PGSIZE;
22         // 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)
        // spp = 0, spie = 1, sum = 1
        task[i]->thread.sstatus = (1 << 18) | (1 << 5);
        // 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;

        task[i]->pgd = (pagetable_t)alloc_page();
        // memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
        for (int j = 0; j < PGSIZE / 8; ++j) {
            task[i]->pgd[j] = swapper_pg_dir[j];
        }

        uint64 user_stack = (uint64)alloc_page();
        create_mapping(task[i]->pgd, USER_END - PGSIZE, user_stack - PA2VA_OFFSET, PGSIZE, 0x17);

        uint64 uapp_size = _eramdisk - _sramdisk;
        char *uapp = (char *)alloc_pages((uapp_size + PGSIZE - 1) / PGSIZE);
        // memcpy(uapp, _sramdisk, uapp_size);
        for (int j = 0; j < uapp_size; ++j) {
            uapp[j] = _sramdisk[j];
        }
        create_mapping(task[i]->pgd, USER_START, (uint64)uapp - PA2VA_OFFSET, uapp_size, 0x1F);

    }

    printk("...proc_init done!\n");
}
```

- 修改 __switch_to，需要加入 保存/恢复 `sepc` `sstatus` `sscratch` 以及 切换页表的逻辑。

切换页表时，先加载 `pagetabled` 的地址，然后使用与 `relocate` 函数相同的逻辑转换成 `PPN`，并加上 `MODE` 的字段，写入satp寄存器。

```
__switch_to:
    # ...
    csrr t1, sepc
    sd t1, 112(t0)
```

```
    csrr t1, sstatus
    sd t1, 120(t0)
    csrr t1, sscratch
    sd t1, 128(t0)

    # ...
    ld t1, 112(t0)
    csrw sepc, t1
    ld t1, 120(t0)
    csrw sstatus, t1
    ld t1, 128(t0)
    csrw sscratch, t1

    # page table
    addi a1, a1, 168 # thread struct
    ld t0, 0(a1)
    li t1, 0xffffffdf80000000
    sub t0, t0, t1
    srl t0, t0, 12
    # sv39
    li t1, 0x8000000000000000
    or t0, t0, t1
    csrw satp, t0

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

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

首先修改 `__dummy`，只需要用 `csrrw` 实现第一次发生切换时用户栈与内核栈的切换即可

```
__dummy:
    csrrw sp, sscratch, sp
    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）。由于Linux源码中在_traps没有存tp寄存器，这里是可以用tp寄存器来存sscratch的值进行判断的，但是我们这里的实现方式存了tp寄存器，所以需要将sscratch的值与sp先做交换，如果交换过来的值为0，就是内核线程，这时需要再交换回来再进行处理；如果不是零，则为用户线程，直接处理即可。

`uapp` 使用 `ecall` 会产生 `ECALL_FROM_U_MODE` exception，查阅手册得值为8。因此我们需要在 `trap_handler` 里面进行捕获，这里我们增加一个参数

```
_traps:
    csrrw sp, sscratch, sp
    bnez sp, 2f
    csrrw sp, sscratch, sp
2:
    addi sp, sp, -256
    # ...

    add a2, sp, 0 # regs
    jal trap_handler

    # ...
    sret
```

对于 `trap_handler` 我们修改如下：

```
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs
*regs) {
    unsigned long temp = 1;
    if(scause & (temp << 63)) { // interrupt
        switch (scause & ~(temp << 63))
        {
        case 0x5:
            printk("[INTERRUPT] S mode timer interrupt!\n");
            clock_set_next_event();
            do_timer();
            break;

        default:
            break;
        }
```

```
    } else {
        switch (scause & ~(temp << 63))
        {
        case 0x8: // ECALL_FROM_U_MODE
            // 系统调用参数使用 a0 - a5 , 系统调用号使用 a7 ,  系统调用的返回值会被保存到 a0, a1 中
            uint64 syscall_id = regs->a7;
            switch (syscall_id)
            {
            case 64: // sys_write
                regs->a0 = sys_write(regs->a0, regs->a1, regs->a2);
                break;
            case 172: // sys_getpid
                regs->a0 = sys_getpid();
                break;
            }
            // 针对系统调用这一类异常,  我们需要手动将 sepc + 4
            regs->sepc += 4;
            break;

        default:
            break;
        }
    } // exception
}
```

其中 `pt_regs` 结构体的定义与 `_traps` 函数中传来的参数一致:

```
struct pt_regs
{
    uint64 ra, gp, tp;
    uint64 t0, t1, t2;
    uint64 s0, s1;
    uint64 a0, a1, a2, a3, a4, a5, a6, a7;
    uint64 s2, s3, s4, s5, s6, s7, s8, s9, s10, s11;
    uint64 t3, t4, t5, t6;
    uint64 sepc;
};
```

添加系统调用

增加 `syscall.c` `syscall.h` 文件，并在其中实现 `getpid` 以及 `write` 逻辑。

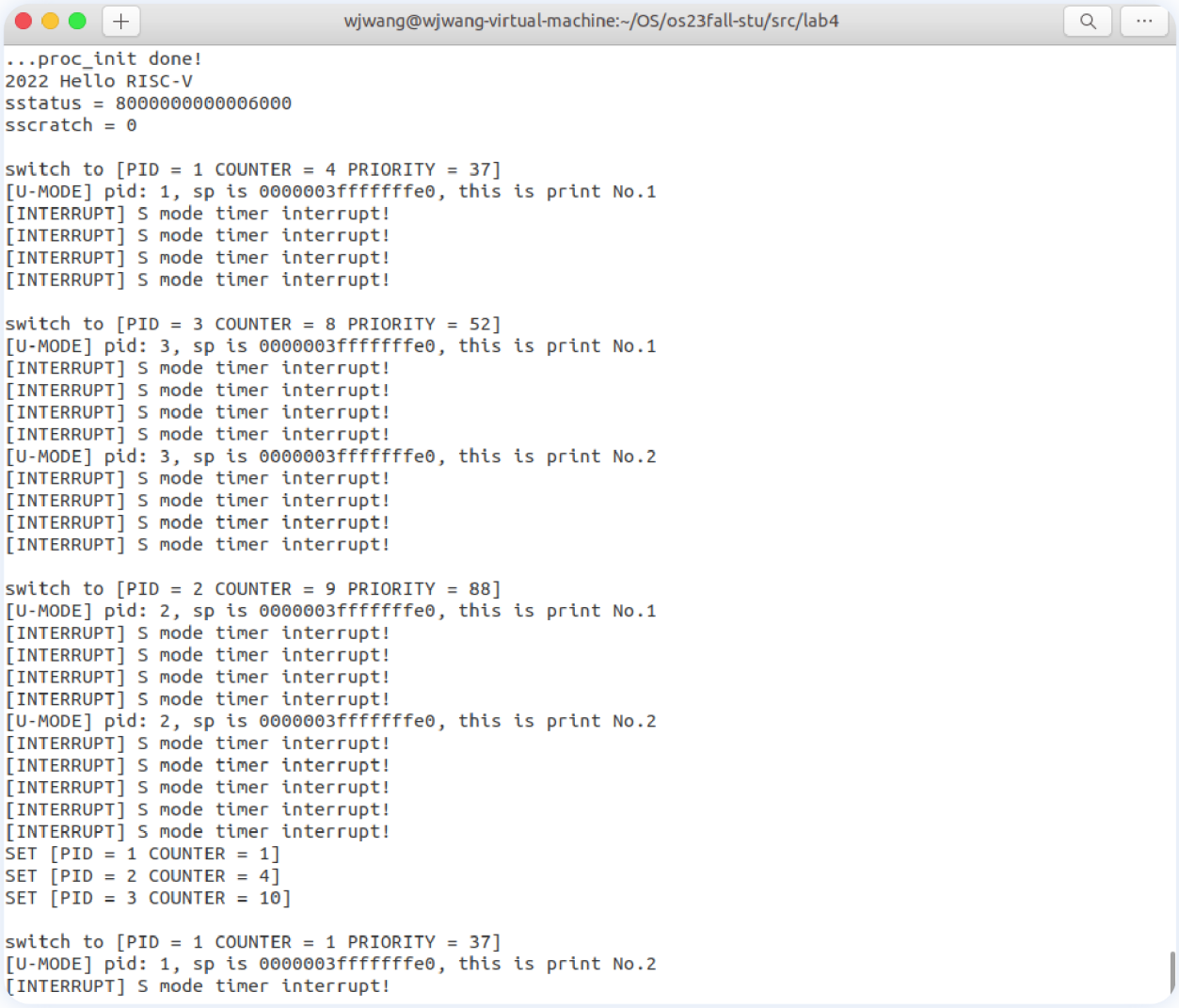
```h
// syscall.h
#ifndef _SYS_SYSCALL_H
#define _SYS_SYSCALL_H

#include "types.h"
#include "printk.h"
#include "proc.h"

#define STDOUT 1

uint64 sys_write(unsigned int fd, const char* buf, size_t count);
uint64 sys_getpid();

#endif
```

```c
// syscall.c
#include "syscall.h"

extern struct task_struct *current;

uint64 sys_write(unsigned int fd, const char* buf, size_t count) {
    if (fd != STDOUT) {
        printk("sys_write: fd != STDOUT\n");
        return -1;
    }
    if (count <= 0) {
        printk("sys_write: count <= 0\n");
        return -1;
    }
    uint64 cnt = printk("%s", buf);
    return cnt;
}

uint64 sys_getpid() {
    return current->pid;
}
```

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。

测试纯二进制文件



添加 ELF 支持

首先我们将 `uapp.S` 中的 payload 给换成我们的 ELF 文件，这时候从 `_sramdisk` 开始的数据就变成了名为 `uapp` 的 ELF 文件。

```
/* user/uapp.S */
.section .uapp

.incbin "uapp"
```

修改 `task_init` 函数，使之支持ELF，实际上就是将之前从 `_ramdisk` 开始的程序变成ELF的格式，处理逻辑基本与二进制文件相同。

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

    // check ELF header
    // printk("ehdr->e_ident = %lx\n", ehdr->e_ident[0]);
    if (ehdr->e_ident[0] != 0x7f || ehdr->e_ident[1] != 'E' ||
        ehdr->e_ident[2] != 'L' || ehdr->e_ident[3] != 'F') {
        printk("not elf format\n");
        return -1;
    }

    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
            uint64 offset = phdr->p_vaddr & 0xfff;
            uint64 size = phdr->p_memsz + offset;
            char* va = (char*)alloc_pages((size + PGSIZE - 1) / PGSIZE);
            // memcpy(va + offset, (char*)ehdr + phdr->p_offset, phdr-
>p_filesz);
            for (int j = 0; j < phdr->p_filesz; ++j) {
                va[offset + j] = ((char*)ehdr)[phdr->p_offset + j];
            }
            memset(va + offset + phdr->p_filesz, 0, phdr->p_memsz - phdr-
>p_filesz);
```

```
            create_mapping(task->pgd, phdr->p_vaddr, (uint64)va - PA2VA_OFFSET, size, 1 << 4 | 1 << 0 | (phdr->p_flags & 0x4) >> 1 | (phdr->p_flags & 0x2) << 1 | (phdr->p_flags & 0x1) << 3);
        }
    }

    // allocate user stack and do mapping

    uint64 user_stack = (uint64)alloc_page();
    create_mapping(task->pgd, USER_END - PGSIZE, user_stack - PA2VA_OFFSET, PGSIZE, 0x17);
    // pc for the user program
    task->thread.sepc = ehdr->e_entry;
    return 0;
}
```

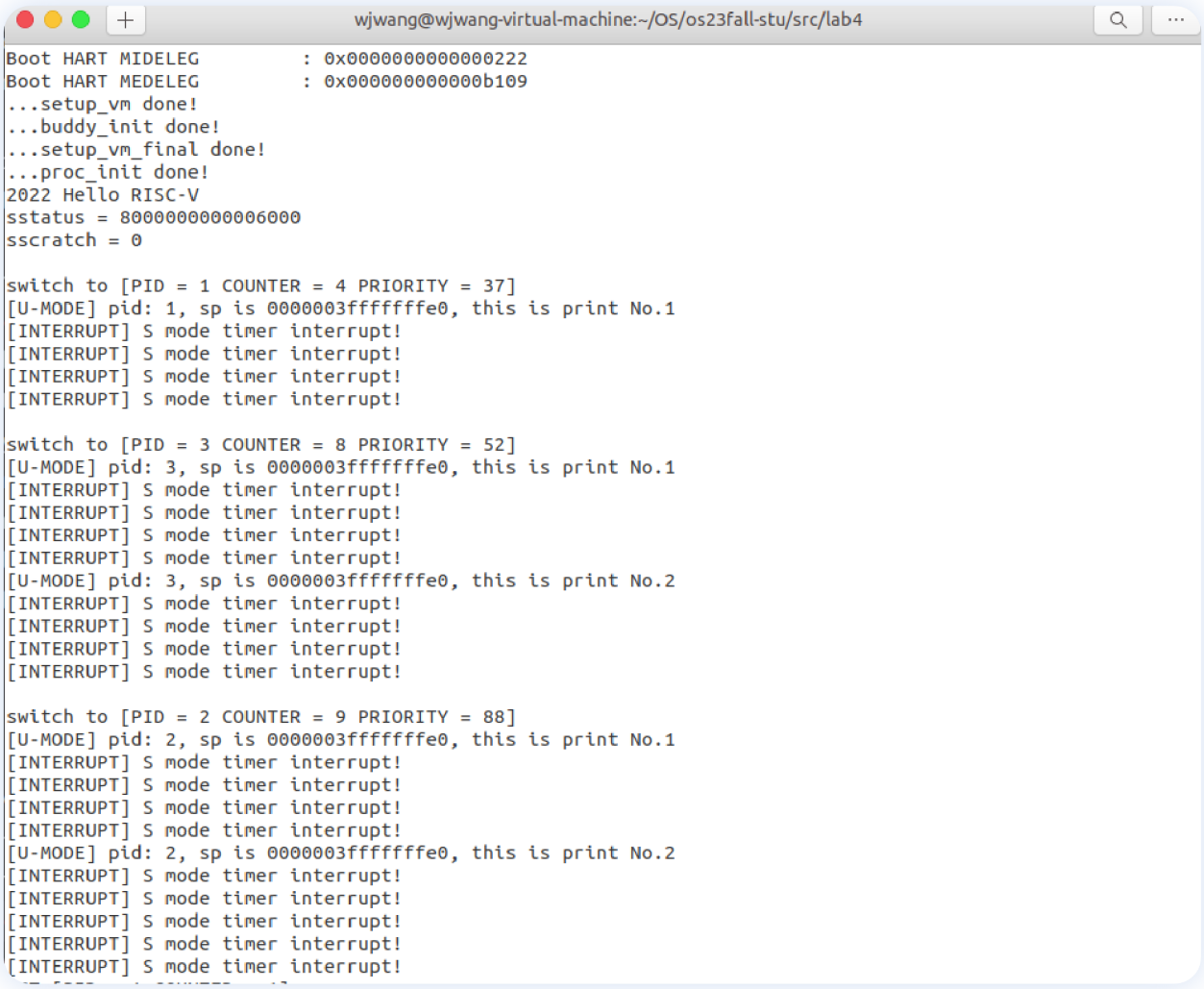
```
void task_init() {
    test_init(NR_TASKS);

    idle = (struct task_struct*)kalloc();
    idle->state = TASK_RUNNING;
    idle->counter = 0;
    idle->priority = 0;
    idle->pid = 0;
    current = idle;
    task[0] = idle;

    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;
        // 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）
        // spp = 0, spie = 1, sum = 1
        task[i]->thread.sstatus = (1 << 18) | (1 << 5);
        // 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;
```

```
25
26          task[i]->pgd = (pagetable_t)alloc_page();
27          // // memcpy(task[i]->pgd, swapper_pg_dir, PGSIZE);
28          for (int j = 0; j < PGSIZE / 8; ++j) {
29              task[i]->pgd[j] = swapper_pg_dir[j];
30          }
31
32          // select how to load the program
33          load_program(task[i]);
34          // load_binary(task[i]);
35
36      }
37
38      printk("...proc_init done!\n");
39  }
```

我们顺便修改了 `task_init` 以适应函数结构变化，将之前的二进制部分也写成函数：

```
1   static uint64_t load_binary(struct task_struct* task) {
2       uint64 user_stack = (uint64)alloc_page();
3       create_mapping(task->pgd, USER_END - PGSIZE, user_stack - PA2VA_OFFSET,
    PGSIZE, 0x17);
4
5       uint64 uapp_size = _eramdisk - _sramdisk;
6       char *uapp = (char *)alloc_pages((uapp_size + PGSIZE - 1) / PGSIZE);
7       // memcpy(uapp, _sramdisk, uapp_size);
8       for (int j = 0; j < uapp_size; ++j) {
9           uapp[j] = _sramdisk[j];
10      }
11      create_mapping(task->pgd, USER_START, (uint64)uapp - PA2VA_OFFSET,
    uapp_size, 0x1F);
12      task->thread.sepc = USER_START;
13      return 0;
14  }
```

测试ELF支持



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。因为我们创建用户态线程的时候，每个线程都有单独的内核栈和用户栈。这种安排会使用户态线程执行系统调用时，不会因为其他线程的系统调用而被阻塞。这样的设计提高了线程的运行效率，因为每个线程都可以独立地进行系统调用，而不会相互干扰。

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

   `p_filesz` 指定了segment在文件中的大小，也就是在磁盘上实际占用的空间大小。这个大小仅包括该段中需要从文件中读取的数据。`p_memsz` 指定了段在内存中占用的空间大小，某些段在运行时需要额外的空间。`.bss` 段（未初始化数据段）在文件中就不需要占用空间，这些数据只需要在当ELF文件加载到内存中占用空间即可，因此 `p_memsz` 包含 `.bss` 段，而 `p_filesz` 不包含。

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?

   每个进程的虚拟地址空间是独立的，每一个进程都有自己的页表，不同进程可能会使用相同的虚拟地址，但是各自的页表会将其映射到不同的物理地址上面。在实验中，用户进程的栈所在的虚拟地址都是相同的，因为我们使用了相同的方式来分配它们。

   栈所在的物理地址通常来说用户是没办法知道的，页表是将虚拟地址映射到物理地址的桥梁，但是页表是由os管理的，用户无法访问，所以只能使用虚拟地址。