浙江大学实验报告

一、实验内容

准备工程

根据实验指导，从git仓库中将需要的代码拷贝至本地，并在需要改动的文件内增加相应的代码。

创建用户态进程

为了创建用户态进程，我们将三个记录用户态进程信息的变量加入线程状态结构体中。由于要保证用户态进程之间的隔离，每个用户态进程都要有各自的页表。向进程信息结构中加入一个记录页表首地址的变量。

```
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 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;

    pagetable_t pgd;
};
```

下面修改进程初始化函数`task_init()`。首先将每个用户态进程的`sepc`设置为进程空间的首地址；开启`sstatus`的`SPIE`和`SUM`，使得U-mode下可以开启中断且S-mode可以访问用户页面；将`sscratch`设置为用户空间的尾地址，作为栈指针。

其次，为每个用户态进程都创建自己的页表，在这里需要多次调用`create_mapping()`来为每个段创建映射。和上次实验一样，要注意首地址的对齐问题。在传入物理地址的时候，将虚拟地址减去线性映射的偏移来得到物理地址。

```
void task_init()
{
    unsigned long u_stack = 0;
    idle = (struct task_struct *)kalloc(PGSIZE);

    // set idle thread
```

```
    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(PGSIZE);
        task[i]->state = TASK_RUNNING;
        task[i]->counter = 0;
        task[i]->priority = rand() % PRIORITY_MAX + PRIORITY_MIN;
        task[i]->pid = i;
        task[i]->thread.ra = (uint64)__dummy;
        task[i]->thread.sp = (uint64)task[i] + PGSIZE;
        task[i]->thread.sepc = USER_START;
        task[i]->thread.sstatus = SUM_MASK | SPIE_MASK;
        // sp of U-mode
        task[i]->thread.sscratch = USER_END;
        // the address of the copy of swapper_pg_tbl
        task[i]->pgd = (pagetable_t)kalloc(PGSIZE);
        u_stack = (unsigned long)kalloc(PGSIZE);
        printk("task[%d]->pgd = 0x%lx\n",i,task[i]->pgd);
        // recreate swapper_pg_tbl since there is no memcpy()
        memset(task[i]->pgd,0x0,PGSIZE);
        create_mapping(task[i]->pgd,(unsigned long)_stext,
                       (unsigned long)(_stext - PA2VA_OFFSET),
                       (unsigned long)PGROUNDUP(_etext - _stext),
                       X_MASK | R_MASK | V_MASK);
        create_mapping(task[i]->pgd,(unsigned long)_srodata,
                       (unsigned long)(_srodata - PA2VA_OFFSET),
                       (unsigned long)PGROUNDUP(_erodata - _srodata),
                       R_MASK | V_MASK);
        create_mapping(task[i]->pgd,(unsigned long)_sdata,
                       (unsigned long)(_sdata - PA2VA_OFFSET),
                       (unsigned long)PGROUNDUP(_edata - _sdata),
                       W_MASK | R_MASK | V_MASK);
        create_mapping(task[i]->pgd,(unsigned long)_sbss,
                       (unsigned long)(_sbss - PA2VA_OFFSET),
                       (unsigned long)PGROUNDUP(_ebss - _sbss),
                       W_MASK | R_MASK | V_MASK);

        create_mapping(task[i]->pgd,(unsigned long)PGROUNDUP(_ekernel),
                       (unsigned long)PGROUNDUP(_ekernel - PA2VA_OFFSET),
                       (unsigned long)( PHY_END -  PGROUNDUP(_ekernel -
PA2VA_OFFSET)),
                       W_MASK | R_MASK | V_MASK);

        // mapping UAPP
        create_mapping(task[i]->pgd,
                       (unsigned long)USER_START,
                       (unsigned long)(uapp_start - PA2VA_OFFSET),
                       (unsigned long)PGROUNDUP(uapp_end - uapp_start),
                       U_MASK | X_MASK  | R_MASK | V_MASK | W_MASK);
        // mapping the stack of U-mode
        create_mapping(task[i]->pgd,
                       (unsigned long)(USER_END - PGSIZE),
```

```
                        (unsigned long)(u_stack - PA2VA_OFFSET - PGSIZE),
                        (unsigned long)PGSIZE,
                        U_MASK  | R_MASK | V_MASK | W_MASK);
    }

    printk("...proc_init done!%d\n", NR_TASKS);
}
```

修改进程切换函数`__switch_to`。加入保存与恢复`sepc`、`sstatus`和`sscratch`以及切换页表的代码。需要注意的是，`satp`中保存的应该是页表的物理地址，因此在保存与恢复的时候要注意加减线性映射的偏移。

```
__switch_to:
    ...
    # save sepc, sscratch, satp
    csrr t0, sepc
    sd t0, 152(a0)
    csrr t0, sstatus
    sd t0, 160(a0)
    csrr t0, sscratch
    sd t0, 168(a0)
    csrr t0, satp
    slli t0,t0,12
    li t1, 0xffffffdf80000000
    add t0, t0, t1
    sd t0,176(a0)


    ...
    # set sepc
    ld t0, 152(a1)
    csrw sepc, t0
    # set sstatus
    ld t0, 160(a1)
    csrw sstatus, t0
    # set sscratch
    ld t0, 168(a1)
    csrw sscratch, t0
    # set satp
    ld t0, 176(a1)
    li t1, 0xffffffdf80000000
    sub t0, t0, t1
    srli t0, t0, 12
    li t1, 0x8
    slli t1, t1, 60
    add t0, t0, t1
    csrw satp, t0
    sfence.vma zero, zero

    ret
```

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

首先处理和堆栈有关的问题。初始化进程的时候，`sp`中保存的是S-mode的堆栈，而`sscratch`中保存的是U-mode的堆栈。进程初始化进入`__dummy`时，进程由S-mode进入U-mode，因此在返回之前要先交换`sscratch`和`sp`中的值。

在异常处理时同理。首先交换`sscratch`和`sp`中的值，再判断`sp`的值是否为0。如果是0，则说明`sscratch`原本的值就是0，这是一个内核线程触发的异常。此时再次交换`sscratch`和`sp`中的值，则实现了"不需要切换堆栈"的效果。在进入异常处理函数之前保存通用寄存器和`sepc`、`scause`的值。从异常处理函数返回之后，恢复通用寄存器和`sepc`、`scause`的值，并再次检查是否需要切换堆栈，最后退出`_trap`。

```
__dummy:
    csrrw sp, sscratch, sp
    sret
_traps:
    # obviously, the context should be saved on the kernel stack instead of the
user stack
    # judge whether sscratch == 0
    # the entry is kernel thread, not swap the sscratch and sp
    csrrw sp, sscratch, sp
    bne sp, zero, __is_kernel
    csrrw sp, sscratch, sp
    # -----------
    # 1. save 32 registers and sepc to stack
__is_kernel:
    addi sp, sp, -272
    sd x0,0(sp)
    ...
    csrr a1, sepc
    sd a1, 256(sp)
    csrr a0, scause
    sd a0, 264(sp)
    addi a2, sp, 0
    call trap_handler  # trap_handler

    # -----------
    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from
stack
    ld t0, 256(sp)
    csrw sepc, t0
    ld t0, 264(sp)
    csrw scause, t0
    ld x1,8(sp)
    ...
    addi sp,sp,272
    # judge whether sscratch == 0
    # the entry is kernel thread, not swap the sscratch and sp
    csrrw sp, sscratch, sp
    bne sp, zero, __exit
    csrrw sp, sscratch, sp
__exit:
    sret
```

根据保存寄存器的顺序，不难写出`pt_regs`结构体的定义。在异常处理函数中，判断触发异常的原因，如果是来自U-mode的系统调用，通过结构体向系统函数传参并保存返回值。根据要求，这里不可以直接修改寄存器，需要修改堆栈中保存的U-mode的寄存器值。需要注意的是，处理`ECALL_FROM_U_MODE`的异常时，需要手动将返回地址+4，确保该类异常处理完成后程序能继续运行。

```
#define ECALL_FROM_U_MODE 0x8
#define ECALL_FROM_S_MODE 0x9

struct pt_regs{
    uint64 reg[32];
    uint64 sepc;
    uint64 sstatus;
};

void trap_handler(unsigned long long scause, unsigned long long sepc, struct
pt_regs*regs) {
    if (scause == 0x8000000000000005){
        // printk("[%d] Supervisor Mode Timer Interrupt ", i=(i+1)%3600);
        // printk("Next time: ");
        clock_set_next_event();
        do_timer();

    }
    else if(scause == ECALL_FROM_U_MODE){
        switch (regs->reg[17])
        {
        case SYS_GETPID:
            regs->reg[10] = getpid();
            break;
        case SYS_WRITE:
            regs->reg[10] = write(regs->reg[10],(char *)regs->reg[11],regs-
>reg[12]);
            break;
        default:
            break;
        }
        regs->sepc = regs->sepc + 4;
    }
    else {
        // printk("123\n\n\n\n\n");
    }
    return ;
}
```

添加系统调用

新建`syscall.h`和`syscall.c`两个文件，包含系统调用函数。如果是输出字符串，获取需要打印的字符串之后，调用`sbi_ecall`逐个打印字符即可。如果是获取pid，直接返回当前进程的pid即可，由`trap_handler`将该返回值保存到堆栈中。

```
extern struct task_struct *current;
uint64 write(unsigned int fd, const char* buf, size_t count)
{
    for(uint64 i=0 ; i<count ; i++)
```

```
    {
        // system call to output a char at a time
        sbi_ecall(SBI_PUTCHAR,0,buf[i],0,0,0,0,0);
    }
    return count;
}
uint64 getpid()
{
    // simply return the value
    return current->pid;
}
```

修改 `head.S` 以及 `start_kernel`

在`head.S`中注释掉设置`sstatus`的SIE位的代码，确保`schedule()`不会被中断。

```
_start:
    ...
    # set sstatus[sie]
    la a0, _traps
    csrw stvec, a0
    # csrsi sstatus,2
    # set the first time interrupt
    ...
```

将`schedule()`放在`test()`前面，手动进行即时的调度而不是等待时钟中断。

```
int start_kernel() {
    printk("%d",2021);
    printk(" Hello RISC-V 3190103058 3190102214\n");
    // call schedule() since the first timer interrupt is disabled
    schedule();
    test(); // DO NOT DELETE !!!

    return 0;
}
```

编译及测试

运行程序，可以看到正常输出。可以看到，每个进程都有各自的页表，程序的运行结果也与实验指导的一致。

二、思考题

1. 如果我们想让 `sstatus[SUM]` 位置0，那么我们应该如何访问用户传入的内容呢，例如，`sys_write()` 中的buf指针指向的内容?

在Linux中有两个特殊的系统调用函数 `copy_from_user()` 和 `copy_to_user()`，前者用于从用户空间往内核空间中拷贝内容，后者用于从内核空间往用户空间中拷贝内容。在调用这两个函数时，系统会将 `sstatus[SUM]` 短暂置1，待拷贝的动作完成后又将 `sstatus[SUM]` 位置0。 通过这两个函数，buf指针指向的内容即是内核空间中用户传入内容的拷贝。

三、讨论心得

本次实验比较复杂，因为牵涉的权限设置、上下文切换、创建映射内容繁多。这次实验让我们感受到了，实际的os设计之复杂、精巧。

实验过程中尤其需要注意每个权限位的意义，比如 `sstatus` 的SPP位，如果其为0代表着进入S-mode之前，系统处于U-mode，否则代表着系统一直处在S-mode中。即，触发异常的是一个S-mode下的进程。如果不清楚每个权限位的意义，实验的进行会有很多困难。这一次实验要求具有完整的知识储备。

而这一次的思考题我们依旧求助了linux源代码，有点没想到最后的解决方案还是需要把 `sstatus[SUM]` 置1。