浙江大学实验报告

课程名称：	操作系统	**实验类型：**	综合型
实验项目名称：	实验 5：RV64 用户态、系统调用、缺页异常处理以及 fork 机制		
学生姓名：	张云轩	**专业：**	信息工程
学号：	3200105087	**手机：**	13583875373
邮件地址：	zhangyunxuan@zju.edu.cn	**实验日期：**	2022 年 12 月 26 日

一、实验目的

1.1. 通过 vm_area_struct 数据结构实现对进程多区域虚拟内存的管理

1.2. 在 lab4 实现页式存储管理的基础上，添加缺页异常处理 Page Fault Handler

1.3. 为进程加入 fork 机制，能够支持通过 fork 创建新的进程

二、主要仪器设备（必填）

硬件平台：Dell R720, Intel Xeon E5-2650v2 16 Cores CPU @2.60GHz; 24GB Memory

宿主机系统：VMware ESXi 7.0 U2

虚拟机系统：Ubuntu/Linux 20.04 LTS Server，14C16G，Kernel version 5.4.0-125

三、实验原理

3.1. 虚拟内存的用户内存空间和内核内存空间

在 linux 开启了虚拟内存之后，虽然操作系统和用户进程都运行在虚拟地址上，但是操作系统的虚拟地址和用户进程的虚拟地址是不同的。操作系统的虚拟地址是 0xffffffe000000000-0xffffff000000000 这一段地址，而用户程序的虚拟地址是 0x0000000000-0x4000000000 这一段。如果有多个用户进程，则每个用户进程都能使用 0x0000000000-0x4000000000 这一段虚拟内存空间，互不干扰。实现这一功能的方式是每个用户进程都有自己独立的页表，这个页表里映射了操作系统内存段和自己用户进程内存段。在进程切换时，操作系统要更换页表。

3.2. PCB

操作系统中对每一个用户进程都有一个控制数据结构，称作 PCB。PCB 不是实现用户程序功能，而是实现用户进程的管理和调度。在本实验中，PCB 占用一个内存页，低地址处保存进程的状态等信息，用于调度算法决策；高地址处是内核栈，用于进程切换时保存和载入寄存器状态上下文。

3.3. 内核态和用户态的切换

在本实验中，当用户程序进行系统调用请求、发生缺页异常时，都会使 CPU 自陷（trap）而进入内核态；在内核态调用 sret 命令时则会回到用户态。在内核态与用户态之间切换时，要交换内核栈指针与用户栈指针。

3.4. demand page

在加载程序的时候，不是一股脑地把全部代码、数据全部一次性加载到内存，而且分配足够多的栈空间。程序加载时并不真正在内存中开辟空间，而是在 PCB 中 vmas 数据结构中记录这个程序要开辟哪些内存空间。当程序执行时，发现本应可以访问的内存空间不能访问（寻址时报 page fault），说明访问的这个地址还没有真正被分配，由 page fault 处理程序分配空间后再重新执行。

只有在真正用到时（page fault）才分配内存空间，不仅可以起到节省内存的作用，也可以实现内存交换，即将一部分内存页换出到硬盘等外部存储设备，缓解内存不足的压力。

3.5. fork 机制

fork 操作创建一个新的 PCB，复制已实际分配的内存，并深拷贝页表（前面提到每一个进程都有自己的页表树）。子进程从父进程停顿的地方继续执行，而不是从头执行。fork 机制的操作最好以代码+注释的形式清晰给出。

四、实验过程

4.1. 用户进程的初始化

主要是设置新的 PCB，并准备该进程的页表。每个用户进程的页表都包含内核代码段的映射。

```
for(int i=1; i<=1; i++) // 一开始只初始化一个进程
    {
        task[i]=(struct task_struct *)kalloc();
        task[i]->state=TASK_RUNNING;
        task[i]->counter=0;
        task[i]->priority=rand();
        task[i]->pid=i;

        task[i]->pgd=(pagetable_t)alloc_page(); // 页表首地址，虚拟地址
        // 拷贝内核态页表
        memcpy((void *)(task[i]->pgd), (void *)(&swapper_pg_dir), PGSIZE);
        // for(int i=0; i<512; i++)
        //     printk("%u ", swapper_pg_dir[i]);
        // 页表的入口地址是物理地址
        task[i]->pgd=(unsigned long)(task[i]->pgd)-(unsigned long)PA2VA_OFFSET;

        load_binary_program(task[i]);

        // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 `thread_struct` 中的 `ra` 和 `sp`
        // 4. 其中 `ra` 设置为 __dummy （见 4.3.2）的地址， `sp` 设置为 该线程申请的物理页的高地址
        (task[i]->thread).ra=__dummy;
        (task[i]->thread).sp=(uint64)task[i]+PGSIZE-1;

    }
```

在 load_binary_program 将程序载入内存过程中，不是真正将程序读入内存，而是在 vmas 上记录下这些内容应该在内存里，当他们被真正访问时，会触发缺页中断，那时候再载入内存。

```
void load_binary_program(struct task_struct *task)
{
    printk("%u\n", (uint64_t)uapp_end-(uint64_t)uapp_start);
    // 记录代码
    do_mmap(task, USER_START, (uint64_t)(uapp_end)-(uint64_t)(uapp_start)+1, VM_X_MASK |
VM_R_MASK | VM_W_MASK, uapp_start);

    // 记录用户栈
    do_mmap(task, (uint64_t)USER_END-(uint64_t)PGSIZE, PGSIZE, VM_R_MASK | VM_W_MASK | VM_ANONYM,
0);
    printk("(load_binary_program) user stack: %u\n", task->vmas[1].vm_start);

    task->thread.sepc=USER_START;
    task->thread.sscratch=USER_END; // 指向用户栈栈顶
    task->thread.sstatus=(1<<18) | (1<<5);  // 权限

    return;
}
```

4.2. 中断入口_trap 的设计

由于在内核态和用户态都会发生中断，因此中断入口函数中首先要检查现在是内核态还是用户态。如果是用户态要切换到内核态，那么要交换内核栈指针和用户栈指针，并在退出内核态时再次交换。如果是内核态中断那就无需交换。此外，中断入口中还要尽量传

递足够多的信息给中断处理程序，以便探查中断原因，并对中断作适当处理。

```
_trap:
    # 是用户态则需要交换栈指针
    csrrw t0, sscratch, t0
    beq t0, x0, _smode_trap_1
    csrrw t0, sscratch, t0
    csrrw sp, sscratch, sp
    jal x0, _umode_trap_1
_smode_trap_1:
    csrrw t0, sscratch, t0
_umode_trap_1:
    addi sp, sp, -272
    sd x1, 0(sp)
    ...
    sd x31, 240(sp)

    csrr t0, sepc
    sd t0, 248(sp)
    csrr t0, sstatus
    sd t0, 256(sp)
    # sscratch 也需要保存，因为每个进程的用户栈指针都是不一样的
    csrr t0, sscratch
    sd t0, 264(sp)

    csrr a0, scause
    csrr a1, sepc
    add a2, sp, x0
    jal x1, trap_handler

__ret_from_fork:
    ld t0, 256(sp)
    csrw sstatus, t0
    ld t0, 248(sp)
    csrw sepc, t0
    ld t0, 264(sp)
    csrw sscratch, t0

    ld x1, 0(sp)
    ld x3, 16(sp)   # x2(sp)要放到最后 load
    ...
    ld x31, 240(sp)
    ld x2, 8(sp)
    addi sp, sp, 272

    csrrw t0, sscratch, t0
    beq t0, x0, _smode_trap_2
    csrrw t0, sscratch, t0
    csrrw sp, sscratch, sp
    jal x0, _umode_trap_2

_smode_trap_2:
    csrrw t0, sscratch, t0

_umode_trap_2:
    sret
```

4.3. 用户进程的切换__switch_to 的设计

为了使用同样的用户地址空间，每个用户进程都有独立的页表，所以在进程切换时不但要保存、载入 **CPU** 寄存器上下文，还要要切换页表。同时为了保证不同用户程序用户态和内核态的正常切换，**sepc**、**sstatush** 和 **sscratch** 寄存器也要保存/载入。

```
__switch_to:
    # 保存当前线程的 ra、sp 和 s0 ~ s11，到当前线程的 thread_struct 中
    # C 代码中传入的是 current 和 next 结构体的指针。利用偏移量访问结构体中的元素。
    # ra 偏移量 40，sp 偏移量 48，非临时寄存器的保存地址顺延。
    sd ra, 40(a0)    # ra 存到内存中
    sd sp, 48(a0)    # sp 存到内存中
    sd s0, 56(a0)
    ...
    sd s11, 144(a0)
    csrr t0, sepc
    sd t0, 152(a0)
    csrr t0, sstatus
    sd t0, 160(a0)
    csrr t0, sscratch
    sd t0, 168(a0)

    # 将下一个线程的 thread_struct 中的相关数据载入到 ra、sp 和 s0 ~ s11 中。
    ld sp, 48(a1)   # sp 在内存中取出
    ld s0, 56(a1)
    ...
    ld s11, 144(a1)
    ld ra, 40(a1)   # ra 在内存中取出
    ld t0, 152(a1)
    csrw sepc, t0
    ld t0, 160(a1)
    csrw sstatus, t0
    ld t0, 168(a1)
    csrw sscratch, t0

    # 切换页表
    ld t0, 176(a1)
    # satp 寄存器只有低 44 位存储页表入口地址，要先做处理
    srli t0, t0, 12
    li t1, 0x8000000000000000

    or t0, t0, t1
    csrw satp, t0

    # 重置 TLB
    sfence.vma zero, zero
    # 清空指令缓存
    fence.i

    ret
```
当一个用户进程第一次运行时，不需要恢复上文，因此在**__switch_to** 对 **PCB** 交换完成胡不是 **return** 到**_trap** 的 **trap_handler** 之后继续恢复用户上下文，而是跳到**__dummy** 函数，不恢复新进程的上下文（没有可恢复的）而是交换用户栈和内核栈后就直接跳转到用户代码运行了。
```
__dummy:        # 跳过初次上下文切换，直接跳到用户代码，并切换到用户栈
    csrr t0, sscratch
    csrw sscratch, sp
    add sp, t0, x0

    sret
```

4.4. demand page

在创建进程时不真正分配内存，而仅仅记录要分配那些内存，当真正访问这些内存时再真正分配。
```
struct vm_area_struct *find_vma(struct task_struct *task, uint64_t addr)
{
    for(int i=0; i<task->vma_cnt; i++)
    {
        if(addr>=task->vmas[i].vm_start && addr<=task->vmas[i].vm_end)
            return &(task->vmas[i]);
    }

    return NULL;
}

uint64_t do_mmap(struct task_struct *task, uint64_t addr, uint64_t length, int prot, uint64_t
vm_content_offset)
{
    struct vm_area_struct *new_vma=&(task->vmas[task->vma_cnt++]);

    new_vma->vm_start=addr;
    new_vma->vm_end=addr+length-1;
    new_vma->vm_flags=prot;
    new_vma->vm_content_offset=vm_content_offset;
    new_vma->have_allocated=0;

    return addr;
}
```

4.5. 缺页中断处理

当程序访问未被分配的内存时，会触发缺页中断。首先要解析中断类型：

```
void handler_exception(uint64 scause, uint64 sepc, struct pt_regs *regs)
{
    if(scause==ECALL_FROM_U_MODE)   // 用户态系统调用
    {
        ...
    }
    else if(   // 12、13、15 号中断分别为指令缺页异常、读缺页和写缺页
        scause==INSTRUCTION_PAGE_FAULT
        || scause==STORE_AMO_PAGE_FAULT
        || scause==LOAD_PAGE_FAULT
    )
    {
        uint64_t stval=csr_read(stval);
        page_fault_handler(stval);
    }

    return;
}
```

然后对缺页中断作对应处理，也就是分配物理页并映射页表：

```
void page_fault_handler(uint64_t stval)
{
    struct vm_area_struct *vma=find_vma(current, stval);    // 首先查找访问的地址在 vma 中是否有记
录
    if(!vma)    // 如果没有记录则是意外情况
        printk("Fatal Error: stval dose not exist in vmas! stval: %u\n", stval);
    else    // 有记录的话就分配一个物理页
    {
        uint64_t new_page=kalloc();
        create_mapping(
            (uint64_t *)((uint64_t)(current->pgd)+(uint64_t)PA2VA_OFFSET),
            stval,
            (uint64_t)(new_page)-(uint64_t)PA2VA_OFFSET,
            PGSIZE,
            PTE_R | PTE_W | PTE_X | PTE_V | PTE_U
        );
        // 如果非匿名则是磁盘映射，要拷贝
        if(!(vma->vm_flags&VM_ANONYM))
            memcpy((void *)new_page, (void *)vma->vm_content_offset,
vma->vm_end-vma->vm_start+1);

        vma->have_allocated=1;  // 标记“已分配”
    }
}
```

4.6. fork 机制的实现

首先 fork 是系统调用，因此首先要在 trap_handler 中解析它：

```
void handler_exception(uint64 scause, uint64 sepc, struct pt_regs *regs)
{
    if(scause==ECALL_FROM_U_MODE)   // 用户态系统调用
    {
        switch(regs->reg[16])     // a7 寄存器，保存了系统调用号
        {
            ...
            case SYS_CLONE: // fork()
                regs->reg[9]=sys_clone(regs);    // a0 寄存器，保存 fork 返回值（pid）
                break;
            default:
                break;
        }
        regs->sepc+=4;   // 下一条继续执行
    }
}
```

fork 主要有三点：PCB 创建、微调和页表的深拷贝

```
extern struct task_struct *task[];
extern uint64_t tasks_cnt;
extern uint64_t __ret_from_fork;
extern uint64_t swapper_pg_dir[];
uint64_t sys_clone(struct pt_regs *regs)
{
    struct task_struct *new_task_struct=(struct task_struct *)kalloc();
    task[tasks_cnt]=new_task_struct;
    tasks_cnt++;
    // 复制父进程的 PCB 给子进程
    memcpy((void *)new_task_struct, (void *)current, PGSIZE);

    // 子进程在父进程的 fork 断点处继续执行
    // PCB 中内核栈中记录了调度时的寄存器状态。为了让子进程返回时 pid 等值不同于父进程，要对子进程 PCB 的
内核栈进行修改
    new_task_struct->pid=tasks_cnt-1;
    new_task_struct->thread.ra=(uint64_t)&__ret_from_fork;
    struct pt_regs *new_regs=(struct pt_regs
*)((uint64_t)new_task_struct+(uint64_t)PGOFFSET((uint64_t)regs));
    new_task_struct->thread.sp=(uint64_t)new_regs;
    new_regs->reg[9]=0; // 返回 pid=0 代表这是子进程的返回
    new_regs->reg[1]=(uint64_t)new_regs;     // PCB 的栈顶指针也变了
    // 每个用户程序都运行在自己的 0x0-0x4000000000 这一段虚拟地址上，所以 fork 之后用户栈指针不用变
    // 但是 PCB 内核栈虽然也在虚拟地址，但是每个进程的地址是不一样的

    new_regs->sepc=regs->sepc+4;  // 父进程在中断处理返回时 pc+=4，子进程没有额外处理。所以在这里+=4

    // 拷页表
    new_task_struct->pgd=(pagetable_t)kalloc();
    memcpy((char *)(new_task_struct->pgd), (char *)(swapper_pg_dir), PGSIZE);
    new_task_struct->pgd=(pagetable_t)((unsigned long)(new_task_struct->pgd)-(unsigned
long)PA2VA_OFFSET);
    for(int i=0; i<current->vma_cnt; i++)
    {
        if(current->vmas[i].have_allocated==1)  // 已经被映射的页
        {
            uint64_t new_physics_page=kalloc();
            create_mapping(
                (uint64_t *)((uint64_t)(new_task_struct->pgd)+(uint64_t)PA2VA_OFFSET),
                current->vmas[i].vm_start,
                (uint64_t)(new_physics_page)-(uint64_t)PA2VA_OFFSET,
                PGSIZE,
                PTE_R | PTE_W | PTE_X | PTE_V | PTE_U
            );
            memcpy((void *)new_physics_page, (void *)current->vmas[i].vm_start, PGSIZE);
            printk("15\n");
        }
    }

    // 对于父进程，会在这里返回，返回 pid 是自己的 pid，子进程的 pid 在 PCB 中被修改了寄存器，返回时是 0
    return tasks_cnt-2;
}
```

五、实验结果记录

对 getpid.c 中四个用户程序都进行了测试：

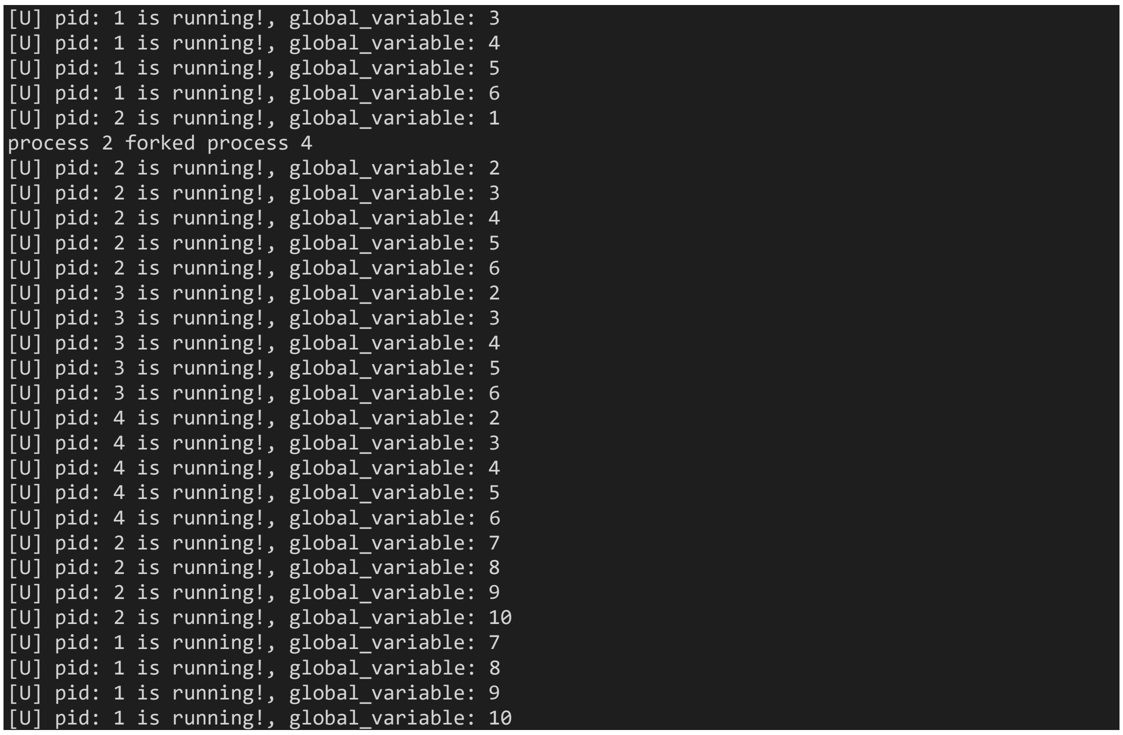
```
process 1 forked process 2
[PID = 1] is running, variable: 0
[PID = 1] is running, variable: 1
[PID = 1] is running, variable: 2
[PID = 1] is running, variable: 3
[PID = 1] is running, variable: 4
[PID = 2] is running, variable: 0
[PID = 2] is running, variable: 1
[PID = 2] is running, variable: 2
```

```
[PID = 2] is running, variable: 3
[PID = 2] is running, variable: 4
[PID = 2] is running, variable: 5
[PID = 2] is running, variable: 6
[PID = 2] is running, variable: 7
[PID = 2] is running, variable: 8
[PID = 1] is running, variable: 5
[PID = 1] is running, variable: 6
[PID = 1] is running, variable: 7
[PID = 1] is running, variable: 8
[PID = 1] is running, variable: 9
[PID = 1] is running, variable: 10
[PID = 1] is running, variable: 11
[PID = 1] is running, variable: 12
[PID = 1] is running, variable: 13
[PID = 1] is running, variable: 14
[PID = 1] is running, variable: 15
[PID = 1] is running, variable: 16
[PID = 1] is running, variable: 17
[PID = 1] is running, variable: 18
```

```
process 1 forked process 2
[U-PARENT] pid: 1 is running!, global_variable: 0
[U-PARENT] pid: 1 is running!, global_variable: 1
[U-PARENT] pid: 1 is running!, global_variable: 2
[U-PARENT] pid: 1 is running!, global_variable: 3
[U-PARENT] pid: 1 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 0
[U-CHILD] pid: 2 is running!, global_variable: 1
[U-CHILD] pid: 2 is running!, global_variable: 2
[U-CHILD] pid: 2 is running!, global_variable: 3
[U-CHILD] pid: 2 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 5
[U-CHILD] pid: 2 is running!, global_variable: 6
[U-CHILD] pid: 2 is running!, global_variable: 7
[U-CHILD] pid: 2 is running!, global_variable: 8
[U-PARENT] pid: 1 is running!, global_variable: 5
[U-PARENT] pid: 1 is running!, global_variable: 6
[U-PARENT] pid: 1 is running!, global_variable: 7
[U-PARENT] pid: 1 is running!, global_variable: 8
[U-PARENT] pid: 1 is running!, global_variable: 9
[U-PARENT] pid: 1 is running!, global_variable: 10
[U-PARENT] pid: 1 is running!, global_variable: 11
```

```
[U] pid: 1 is running!, global_variable: 0
[U] pid: 1 is running!, global_variable: 1
[U] pid: 1 is running!, global_variable: 2
process 1 forked process 2
[U-PARENT] pid: 1 is running!, global_variable: 3
[U-PARENT] pid: 1 is running!, global_variable: 4
[U-PARENT] pid: 1 is running!, global_variable: 5
[U-PARENT] pid: 1 is running!, global_variable: 6
[U-PARENT] pid: 1 is running!, global_variable: 7
[U-CHILD] pid: 2 is running!, global_variable: 3
[U-CHILD] pid: 2 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 5
[U-CHILD] pid: 2 is running!, global_variable: 6
[U-CHILD] pid: 2 is running!, global_variable: 7
[U-CHILD] pid: 2 is running!, global_variable: 8
[U-CHILD] pid: 2 is running!, global_variable: 9
[U-CHILD] pid: 2 is running!, global_variable: 10
[U-CHILD] pid: 2 is running!, global_variable: 11
[U-PARENT] pid: 1 is running!, global_variable: 8
[U-PARENT] pid: 1 is running!, global_variable: 9
[U-PARENT] pid: 1 is running!, global_variable: 10
[U-PARENT] pid: 1 is running!, global_variable: 11
```

```
[U] pid: 1 is running!, global_variable: 0
process 1 forked process 2
[U] pid: 1 is running!, global_variable: 1
process 1 forked process 3
[U] pid: 1 is running!, global_variable: 2
```



```
[U] pid: 1 is running!, global_variable: 3
[U] pid: 1 is running!, global_variable: 4
[U] pid: 1 is running!, global_variable: 5
[U] pid: 1 is running!, global_variable: 6
[U] pid: 2 is running!, global_variable: 1
process 2 forked process 4
[U] pid: 2 is running!, global_variable: 2
[U] pid: 2 is running!, global_variable: 3
[U] pid: 2 is running!, global_variable: 4
[U] pid: 2 is running!, global_variable: 5
[U] pid: 2 is running!, global_variable: 6
[U] pid: 3 is running!, global_variable: 2
[U] pid: 3 is running!, global_variable: 3
[U] pid: 3 is running!, global_variable: 4
[U] pid: 3 is running!, global_variable: 5
[U] pid: 3 is running!, global_variable: 6
[U] pid: 4 is running!, global_variable: 2
[U] pid: 4 is running!, global_variable: 3
[U] pid: 4 is running!, global_variable: 4
[U] pid: 4 is running!, global_variable: 5
[U] pid: 4 is running!, global_variable: 6
[U] pid: 2 is running!, global_variable: 7
[U] pid: 2 is running!, global_variable: 8
[U] pid: 2 is running!, global_variable: 9
[U] pid: 2 is running!, global_variable: 10
[U] pid: 1 is running!, global_variable: 7
[U] pid: 1 is running!, global_variable: 8
[U] pid: 1 is running!, global_variable: 9
[U] pid: 1 is running!, global_variable: 10
```

六、心得体会

在这次实验中我体会最深的是用户程序的载入是通过读入二进制文件，而不是代码，这意味着操作系统是在不知道程序功能的情况下载入任意功能的用户程序，并统一管理、调度他们的运行，这反映了操作系统的通用性。