Lab 5： 动态内存分配与缺页异常处理

1 . 实验目标

- 通过vm_area_srtuct数据结构实现对进程多区域虚拟内存的管理
- 在Lab4 实现用户态程序的基础上，添加缺页异常处理 Page Fault Handler。
- 为进程加入**fork** 机制，能够支持创建新的用户态进程，并测试运行。

2 . 实验环境

- Docker

3 . 背景知识

此处略过背景知识部分

4 . 实验步骤

为 mm_struct 添加 vm_area_struct 数据结构

定义`vm_area_struct`数据结构：

```
typedef struct
{
    unsigned long pgprot;
} pgprot_t;

struct vm_area_struct
{
    /* Our start address within vm_area. */
    unsigned long vm_start;
    /* The first byte after our end address within vm_area. */
    unsigned long vm_end;
    /* linked list of VM areas per task, sorted by address. */
    struct vm_area_struct *vm_next, *vm_prev;
    /* The address space we belong to. */
    struct mm_struct *vm_mm;
    /* Access permissions of this VMA. */
    pgprot_t vm_page_prot;
    /* Flags*/
    unsigned long vm_flags;
};
```

- `vm_page_prot`和`vm_flags`的区别

我们可以注意到这两个字段的C语言类型。`vm_page_prot`的类型是`pgprot_t`，这是arch级别的数据类型，这意味着它可以直接应用于底层架构的Page Table Entries。在RISC-V 64位上，这个字段直接存储了vma的pte中保护位的内容。而`vm_flags`是一个与arch无关的字段，它的位是参照linux/mm.h中定义的。可简单地将`vm_flags`看作是`vm_page_prot`的翻译结果，方便在操作系统其它地方的代码中判断每个`vm_area`的具体权限。

- 其中vm_flags的取值需要采用与这里【link】一致的定义方式。
- 其余结构体内具体的成员变量含义已在`4.2节`中描述。

建立上述 `vm_area_struct` 数据结构后，需配合 `mmap` 等系统调用为进程初始化一个合理的 `vm_area_struct` 链表，使得后续进程在运行时可同时管理多个 `vm_area` 。

实现 Page Fault 的检测与处理

- 修改 `strap.c` 文件, 添加对于Page Fault异常的检测

  RISC-V下的中断异常可以分为两类，一类是interrupt，另一类是exception，具体的类别信息可以通过解析scause[XLEN-1]获得。在前面的实验中，我们已经实现在interrupt中添加了时钟中断处理，在exception中添加了用户态程序的系统调用请求处理。在本次实验中，我们还将继续添加对于Page Fault的处理，即当Page Fault发生时，能够根据scause寄存器的值检测出该异常，并调用针对该类异常的处理函数 `do_page_fault`。

  当Page Fault发生时scause寄存器可能的取值声明如下：

  ```
  # define CAUSE_FETCH_PAGE_FAULT     12
  # define CAUSE_LOAD_PAGE_FAULT      13
  # define CAUSE_STORE_PAGE_FAULT     15
  ```

  注：可在 `strap.c` 中实现使得所有类型的Page Fault处理都最终能够调用到同一个 `do_page_fault`，然后再在该处理函数中继续判断不同Page Fault的类型，并针对该类Page Fault做进一步的处理。

- 在 `fault.c` 中实现 Page Fault 处理函数: `do_page_fault`
  - 根据前文 `5.6节` 中所描述的 `mmap` 映射虚拟地址区域所采用的需求分页（Demand Paging）机制。等进程调度结束，首次切换到某一进程的时候，由于访问了未映射的虚拟内存地址，即会触发产生Page Fault异常。
  - `do_page_fault` 处理函数中具体实现的功能如下：
  - 1. 读取`csr STVAL`寄存器，获得访问出错的虚拟内存地址（`Bad Address`），并打印出该地址。
    2. 检查访问出错的虚拟内存地址（ `Bad Address` ）是否落在该进程所定义的某一`vm_area`中，即遍历当前进程的`vm_area_struct` 链表，找到匹配的`vm_area` 。若找不到合适的`vm_area`，则退出Page Fault处理函数，同时打印输出`Invalid vm area in page fault`。

3. 根据`csr SCAUSE`判断Page Fault类型。
4. 根据Page Fault类型，检查权限是否与当前`vm_area`相匹配。
    - 当发生`Page Fault caused by an instruction fetch`时，Page Fault映射的`vm_area`权限需要为可执行；
    - 当发生`Page Fault caused by a read`时，Page Fault映射的`vm_area`权限需要为可读；
    - 当发生`Page Fault caused by a write`时，Page Fault映射的`vm_area`权限需要为可写；
    - 若发生上述权限不匹配的情况，也需要退出Page Fault处理函数，同时打印输出`Invalid vm area in page fault`。
    - 只有匹配到合法的`vma_area`后，才可进行相应的物理内存分配以及页表的映射。
5. 最后根据访问出错的`Bad Address`，调用Lab4中实现过的`create_mapping`实现新的页表映射关系。注意此时的Bad Address不一定恰好落在内存的4K对齐处，因此需要稍加处理，取得合适的虚拟内存和物理内存的映射地址以及大小。

- 整个do_page_fault函数如下所示

```
void do_page_fault(int scause)
{

    uint64 stval = csr_read(stval);
    struct vm_area_struct *begin = current->mm.vm_area;
    printk("[S] PAGE_FAULT: PID: %d, scause: %d, sepc: %lx, badaddr: %lx\n",
current->pid, scause, current->stack->sepc, stval);
    while (begin != NULL)
    {
        stval = csr_read(stval);

        if (stval >= begin->vm_start && stval < begin->vm_end)
        {

            switch (scause)
            {
            case CAUSE_FETCH_PAGE_FAULT:
                if (!(begin->vm_flags & VM_EXEC))
                {
                    puts("[S] Invalid vm area in page fault[exec]\n");
                    return;
                }
                break;

            case CAUSE_LOAD_PAGE_FAULT:
                if (!(begin->vm_flags & VM_READ))
                {
                    puts("[S] Invalid vm area in page fault[load]\n");
                    return;
                }
                break;
            case CAUSE_STORE_PAGE_FAULT:
                if (!(begin->vm_flags & VM_WRITE))
                {
                    puts("[S] Invalid vm area in page fault [write]\n");
                    return;
                }
                break;
```

```
            default:
                puts("[S] Invalid fault type!\n");
                return;
            }

            uint64 va = PGROUNDDOWN(stval), pa;
            printk("va is %lx\n", va);
            if (va < USER_SIZE)
            {
                pa = PHY_USER_START + va;
            }
            else if (va >= VM_USER_STACK_TOP - PGSIZE && va <
VM_USER_STACK_TOP)
            {
                pa = (uint64)current->mm.user_stack_begin - PA2VA_OFFSET;
            }
            else
            {
                printk("[S] A weird address in page fault %lx\n", va);
                pa = (uint64)alloc_pages(1) - PA2VA_OFFSET;
            }

            printk("[S] mapped PA :%lx to VA :%lx with size :%d,perm:%d \n",
                   pa, va, PGSIZE, begin->vm_flags);
            create_mapping((uint64 *)current->mm.pgtbl, va, pa, PGSIZE,
USER_PERM | (begin->vm_flags << 1) | 1);

            return;
        }
        begin = begin->vm_next;
    }
    puts("[S] Invalid vm area in page fault [not exist]\n");
    return;
}
```

实现 fork 系统调用

修改task_struct

修改task_struct，添加成员`struct pt_regs *stack`：

```
struct pt_regs
{
    uint64 ra, sp, gp, tp;
    uint64 t0, t1, t2, 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;
};

struct task_struct
{
    //.................
    struct pt_regs *stack; // fork使用
```

```
};
```

`stack`成员用于保存异常发生时的寄存器状态，会在fork函数中使用。可以在每次异常处理`handler_s`触发时都将当前的寄存器状态保存到`stack`成员中。具体拷贝代码如下所示

```
void copy_stack(struct pt_regs *src, struct pt_regs *dst)
{
    uint64 *s = (uint64 *)src;
    uint64 *d = (uint64 *)dst;
    for (int i = 0; i < sizeof(struct pt_regs) / sizeof(uint64); i++)
    {
        *(d + i) = *(s + i);
    }
}
```

实现fork函数

`fork`函数的函数原型如下：

```
int fork(void)
{
    return create_fork_task();
}
```

- 父进程`fork`成功时返回：子进程的`pid`，子进程返回：`0`。`fork`失败则父进程返回：`-1`。

`fork`具体的操作如下：

- 创建子进程，仿照lab5的方式，设置好`task_struct`中的`sp`,`sepc`,`sstatus`和`sscratch`成员。根据lab5设置的不同，`sscratch`成员可能设置为陷入异常处理的用户态sp或其他值。
- 为子进程`task_struct`中的`stack`成员分配空间，并将父进程的`stack`内容拷贝进去，注意为了让子进程的返回值为0，需要修改寄存器a0状态。
- 为子进程分配并拷贝父进程的`mm_struct`。
- 为子进程新建页表，为了适应page_fault，这里同样只映射内核的映射。
- 拷贝用户栈。我们需要分配一个页的空间并将父进程的用户栈即`[USER_END - PAGE_SIZE, USER_END)`拷贝进去，作为子进程的用户栈。子进程的用户栈起始地址需要保存到`task_struct`中，等待page_fault为其建立映射（5.8.5节）。
- 设置`task_struct`中的`ra`成员为5.8.3中的`forkret`函数。
- 返回子进程的`pid`，结束。
- 仿照lab5的做法，将`fork`系统调用添加到异常处理函数`handler_s`中，系统调用号为220。

```
int create_fork_task()
{
    puts("create fork task...\n");
    struct task_struct *new_task = (struct task_struct *)(kmalloc(sizeof(struct
task_struct)));
    new_task->state = TASK_RUNNING;
    new_task->counter = COUNTER_INIT_COUNTER[task_num];
    new_task->priority = PRIORITY_INIT_COUNTER[task_num];
    new_task->blocked = 0;
    new_task->pid = task_num;
```

```
    new_task->sscratch = (unsigned long long)alloc_pages(1) + PGSIZE;
    new_task->sepc = current->sepc;

    //初始化stack
    new_task->stack = (struct pt_regs *)kmalloc(sizeof(struct pt_regs));
    uint64 *s = (uint64 *)current->stack;
    for (int i = 0; i < sizeof(struct pt_regs) / sizeof(uint64); i++)
    {
        *(d + i) = *(s + i);
    }
    new_task->stack->a0 = 0;
    new_task->stack->sp = csr_read(sscratch);
    // stack初始结束

    new_task->mm.pgtbl =user_paging_init();
    new_task->mm.user_size = USER_SIZE;
    new_task->mm.vm_area = NULL;
    //这时候直接分配用户栈的物理内存，page fault中只需要建立映射即可
    new_task->mm.user_stack_begin = (uint64 *)alloc_pages(1);
    //拷贝
    for (int i = 0; i < PGSIZE / (sizeof(uint64)); i++)
    {
        *(new_task->mm.user_stack_begin + i) = *(current->mm.user_stack_begin +
i);
    }

    do_mmap(&new_task->mm, 0, USER_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC);
    do_mmap(&new_task->mm, (void *)(VM_USER_STACK_TOP - PGSIZE), PGSIZE,
PROT_READ | PROT_WRITE);

    new_task->thread.ra = (unsigned long long)forkret;

    printk("[PID = %d ] Process fork from [PID = %d]!\n", new_task->pid,
current->pid);

    task[task_num] = new_task;
    task_num++;
    return new_task->pid;
}
```

实际上fork函数想清楚了其实不难写，最重要的是要记住一定要注意a0和sp寄存器的值，a0在子进程中我们设置为0，sp设置为此时的`sscratch`寄存器，因为此时的该寄存器记录着用户栈的指针，其他的直接拷贝current中的stack即可

实现forkret函数

这是完成fork后，context switch到达子进程时该进程返回到的c语言函数，它的作用主要是调用5.8.4中的汇编函数`ret_form_fork`使fork结束的子进程返回到用户模式。

```
void forkret()
{
    return ret_from_fork((uint64 *)current->stack);
}
```

实现ret_from_fork函数

这是一个汇编函数，函数原型如下：

```
ret_from_fork:
    ld t3,248(a0)
    csrw sepc,t3

    #切换权限
    li t1, 0x100            #t1=0001 0000 0000
    csrc sstatus, t1         #将mstatus第8位清0


    ld ra,0(a0)
    ld sp,8(a0)
    ld gp,16(a0)
    ld tp,24(a0)
    ld t0,32(a0)
    ld t1,40(a0)
    ld t2,48(a0)
    ld s0,56(a0)
    ld s1,64(a0)
    ld a1,80(a0)
    ld a2,88(a0)
    ld a3,96(a0)
    ld a4,104(a0)
    ld a5,112(a0)
    ld a6,120(a0)
    ld a7,128(a0)
    ld s2,136(a0)
    ld s3,144(a0)
    ld s4,152(a0)
    ld s5,160(a0)
    ld s6,168(a0)
    ld s7,176(a0)
    ld s8,184(a0)
    ld s9,192(a0)
    ld s10,200(a0)
    ld s11,208(a0)
    ld t3,216(a0)
    ld t4,224(a0)
    ld t5,232(a0)
    ld t6,240(a0)
    ld a0,72(a0)

    sret
```

- 该函数主要是读取`stack`中的所有寄存器状态并返回到用户模式，调用时`stack`是当前`task_struct`中的成员。

- 读取寄存器时需要注意读取的顺序，sscratch已经在__switch_to的时候修改过了，因此这里不需要进行修改
- 这里一定要注意load的顺序，a0一定要是最后一个load自己，其他使用的寄存器应该先使用，后load覆盖

修改 Page Fault 处理

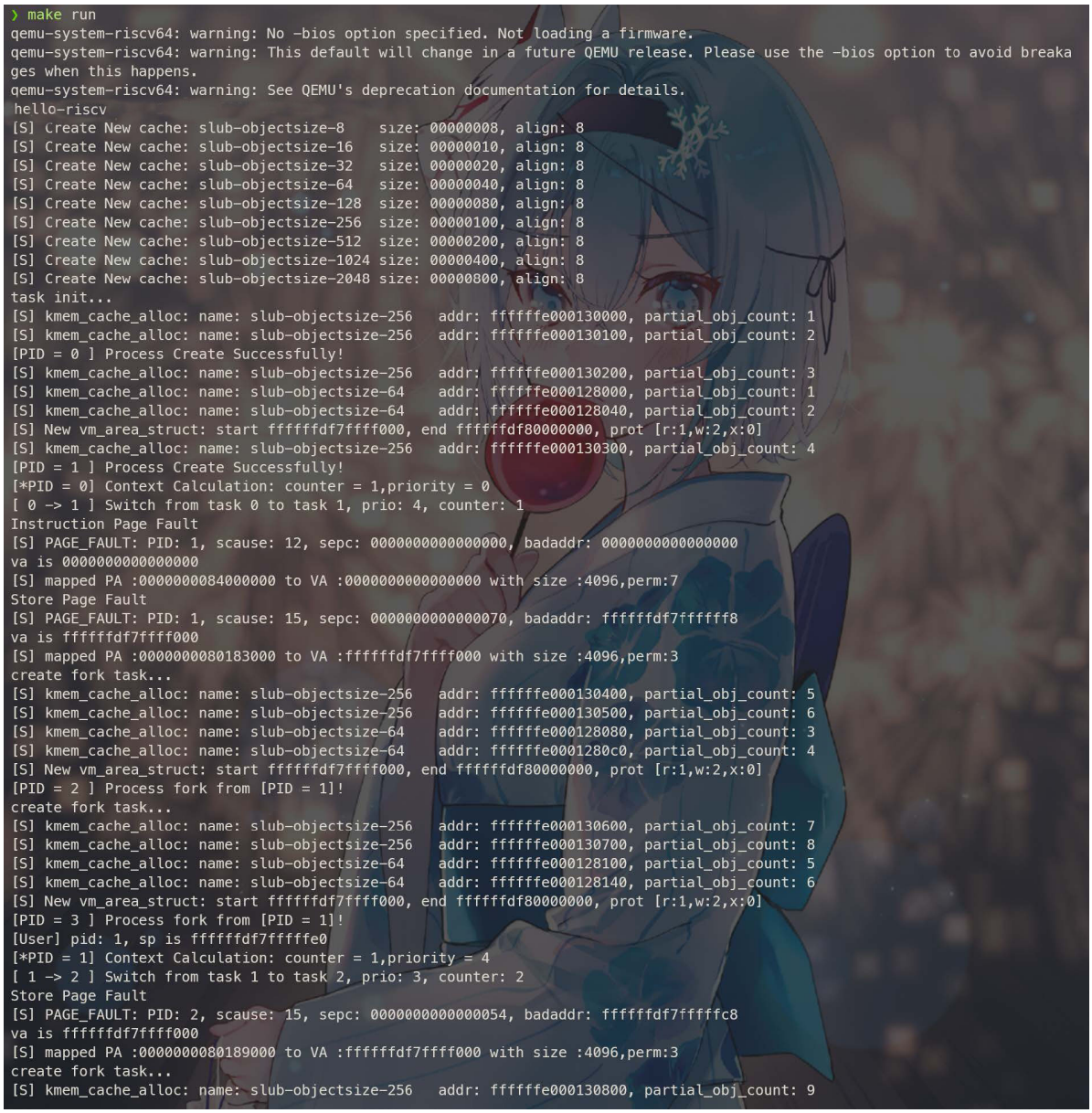
在之前的Page Fault处理中，我们对用户栈Page Fault处理方法是自由分配一页作为用户栈并映射到`[USER_END - PAGE_SIZE, USER_END)`的虚拟地址。但由fork创建的进程，它的用户栈已经拷贝完毕，因此Page Fault处理时直接为该页建立映射即可。

修改task_init函数代码，使fork正常运行

上述代码已经修改过，创建task0 和 task1两个进程，具体操作请参考上文。

5 编译及测试

本次实验的输出结果参考如下：



```
[*PID = 3] Context Calculation: counter = 1,priority = 2

---------------New loop---------------
[PID] 1 get a new counter 5
[PID] 2 get a new counter 5
[PID] 3 get a new counter 5
[PID] 4 get a new counter 2

----------------------------------

[ 3 -> 4 ] Switch from task 3 to task 4, prio: 1, counter: 2
[*PID = 4] Context Calculation: counter = 2,priority = 1
[*PID = 4] Context Calculation: counter = 1,priority = 1
[ 4 -> 3 ] Switch from task 4 to task 3, prio: 2, counter: 5
[*PID = 3] Context Calculation: counter = 5,priority = 2
[*PID = 3] Context Calculation: counter = 4,priority = 2
[*PID = 3] Context Calculation: counter = 3,priority = 2
[*PID = 3] Context Calculation: counter = 2,priority = 2
[*PID = 3] Context Calculation: counter = 1,priority = 2
[ 3 -> 2 ] Switch from task 3 to task 2, prio: 3, counter: 5
[*PID = 2] Context Calculation: counter = 5,priority = 3
[*PID = 2] Context Calculation: counter = 4,priority = 3
[*PID = 2] Context Calculation: counter = 3,priority = 3
[*PID = 2] Context Calculation: counter = 2,priority = 3
[*PID = 2] Context Calculation: counter = 1,priority = 3
[ 2 -> 1 ] Switch from task 2 to task 1, prio: 4, counter: 5
[*PID = 1] Context Calculation: counter = 5,priority = 4
[*PID = 1] Context Calculation: counter = 4,priority = 4
[*PID = 1] Context Calculation: counter = 3,priority = 4
[*PID = 1] Context Calculation: counter = 2,priority = 4
[*PID = 1] Context Calculation: counter = 1,priority = 4

---------------New loop---------------
[PID] 1 get a new counter 4
[PID] 2 get a new counter 4
[PID] 3 get a new counter 4
[PID] 4 get a new counter 5

----------------------------------

[ 1 -> 3 ] Switch from task 1 to task 3, prio: 2, counter: 4
[*PID = 3] Context Calculation: counter = 4,priority = 2
[User] pid: 3, sp is ffffffdf7fffffe0
[*PID = 3] Context Calculation: counter = 3,priority = 2
[*PID = 3] Context Calculation: counter = 2,priority = 2
[*PID = 3] Context Calculation: counter = 1,priority = 2
[ 3 -> 2 ] Switch from task 3 to task 2, prio: 3, counter: 4
[*PID = 2] Context Calculation: counter = 4,priority = 3
[*PID = 2] Context Calculation: counter = 3,priority = 3
[*PID = 2] Context Calculation: counter = 2,priority = 3
[User] pid: 2, sp is ffffffdf7fffffe0
[*PID = 2] Context Calculation: counter = 1,priority = 3
[ 2 -> 1 ] Switch from task 2 to task 1, prio: 4, counter: 4
[*PID = 1] Context Calculation: counter = 4,priority = 4
[*PID = 1] Context Calculation: counter = 3,priority = 4
[*PID = 1] Context Calculation: counter = 2,priority = 4
[*PID = 1] Context Calculation: counter = 1,priority = 4
[ 1 -> 4 ] Switch from task 1 to task 4, prio: 1, counter: 5
[*PID = 4] Context Calculation: counter = 5,priority = 1
```

6思考题

根据同学们的实现，分析父进程在用户态执行 fork 至子进程被调度并在用户态执行的过程，最好能够将寄存器状态的变化过程清晰说明。

内核中的 fork 的系统调用对应的就是 sys_fork 函数，最终会执行fork（）来生成一个以当前进程为父进程，并将当前进程的内存地址空间复制到新产生的进程之中。之后的子进程和父进程同时执行的代码段指向相同。