Lab 6: 实现 fork 机制

3210105488 刘扬

1 实验过程

1.1 准备工作

基于lab5，不赘述。

因为这次实验只初始化一个线程，需要修改`task_init()`

```
task[1] = (struct task_struct*)kalloc();
task[1]->state = TASK_RUNNING;
task[1]->counter = 0;
task[1]->priority = 1;
task[1]->pid = 1;

task[1]->thread.ra = (uint64)__dummy;
task[1]->thread.sp = (uint64)(task[1]) + PGSIZE;
task[1]->thread_info.kernel_sp = (unsigned long)(task[1])+PGSIZE;


pagetable_t pgtbl = (pagetable_t)kalloc();
for(int j = 0;j < PGSIZE / sizeof(uint64*);j++){
    pgtbl[j] = swapper_pg_dir[j];
}
task[1]->pgd = pgtbl;

load_program(task[1], pgtbl);
uint64 satp = (uint64)0x8 << 60;
satp |= ((uint64)(pgtbl) - PA2VA_OFFSET) >> 12;
task[1]->satp = satp;
printk("[S] Initialized: pid: 1, priority: 1, counter: 0\n");
```

同时要注意在`schedule()`时，因为**有些线程没有初始化**，需要额外判断`task[i] != NULL`

1.2 实现`sys_clone()`

1.2.1 参考 task_init 创建一个新的 task，将的 parent task 的整个页复制到新创建的 task_struct 页上。将 thread.ra 设置为__ret_from_fork，并正确设置 thread.sp

复制task_struct并设置thread.ra

```
struct task_struct *child = (struct task_struct*)kalloc();
for(int i = 0;i < PGSIZE;i++){
    ((char*)child)[i] = ((char*)current)[i];
}
child->thread.ra = (uint64)__ret_from_fork;
```

设置thread.sp：这里的sp是用于给`__switch_to`正确的栈指针，使`__ret_from_fork`的栈指针是正确的。

```
uint64 offset = (uint64)regs % PGSIZE;
struct pt_regs *child_regs = (struct pt_regs*)(child + offset);
//__switch_to->__ret_from_fork(in _traps)->user program
child->thread.sp = (uint64)child_regs; // for __switch_to
```

1.2.2 利用参数 regs 来计算出 child task 的对应的 pt_regs 的地址，并将其中的 a0, sp, sepc 设置成正确的值

通过parent task的regs相对于页起始地址的偏移量算出child task的pt_regs的地址

```
uint64 offset = (uint64)regs % PGSIZE;
struct pt_regs *child_regs = (struct pt_regs*)(child + offset);

for(int i = 0;i < 37 * 8;i++){
    ((char*)child_regs)[i] = ((char*)regs)[i];
}
child_regs->x[2] = (uint64)child_regs; // for _traps
child_regs->x[10] = 0; //return 0 (child thread)
child_regs->sepc = regs->sepc + 4;
```

1.2.3 为 child task 申请 user stack，并将 parent task 的 user stack数据复制到其中

```
uint64 child_stack = alloc_page();
for(int i = 0;i < PGSIZE;i++){
    ((char*)child_stack)[i] = ((char*)(USER_END - PGSIZE))[i];
}
```

1.2.4 同时将子 task 的 user stack 的地址保存在 thread_info->=user_sp 中

```
child->thread_info.kernel_sp = (uint64)(child)+PGSIZE;
child->thread_info.user_sp = (uint64)USER_END;
```

1.2.5 为 child task 分配一个根页表，并仿照 setup_vm_final 来创建内核空间的映射

```
child->pgd = (uint64*)alloc_page();
uint64 satp = (uint64)0x8 << 60;
satp |= ((uint64)(child->pgd) - PA2VA_OFFSET) >> 12;
child->satp = satp;
for(int i = 0;i < PGSIZE;i++){
    ((char*)child->pgd)[i] = ((char*)swapper_pg_dir)[i];
}
create_mapping(child->pgd, USER_END-PGSIZE, (uint64)child_stack-PA2VA_OFFSET, PGSIZE, 23);
```

1.2.6 根据 parent task 的页表和 vma 来分配并拷贝 child task 在用户态会用到的内存

遍历parnet task的vma，对于每个vma中的每一页，通过走页表来判断parent task是否有做映射，如果做了映射，child task的相应地址也需要做映射

```
void copy_vma(uint64* pgd){
    for (int i = 0; i < current->vma_cnt; i++) {
        struct vm_area_struct *cur_vma = &(current->vmas[i]);
        uint64 cur_addr = cur_vma->vm_start;
        while (cur_addr < cur_vma->vm_end) {
            walk_pgd(current->pgd, PGROUNDDOWN(cur_addr), pgd);
            cur_addr += PGSIZE;
        }
    }
}
```

walk_pgd的主要逻辑是判断parent task的每个page table entry的V位，如果V位都被置1，说明该页已经映射，相应的child task需要做拷贝

```
int walk_pgd(uint64* parent_pgd, uint64 va, uint64* child_pgd){
    uint64 cur_vpn;
    uint64* cur_pgtbl;
    uint64 cur_pte;
    cur_pgtbl = parent_pgd;
    cur_vpn = ((uint64)(va) >> 30) & 0x1ff;
    cur_pte = *(cur_pgtbl + cur_vpn);
    if (!(cur_pte & 1)) {
        return;
    }
    cur_pgtbl = (uint64*)(((cur_pte >> 10) << 12) + PA2VA_OFFSET);
    cur_vpn = ((uint64)(va) >> 21) & 0x1ff;
    cur_pte = *(cur_pgtbl + cur_vpn);
    if (!(cur_pte & 1)) {
        return;
    }
    cur_pgtbl = (uint64*)(((cur_pte >> 10) << 12) + PA2VA_OFFSET);
    cur_vpn = ((uint64)(va) >> 12) & 0x1ff;
    cur_pte = *(cur_pgtbl + cur_vpn);
    if (!(cur_pte & 1)) {
        return;
    }
    uint64 page = alloc_page();
    create_mapping((uint64)child_pgd, va, (uint64)page-PA2VA_OFFSET, PGSIZE, 31);
    for(int i = 0;i < PGSIZE;i++){
        ((char*)page)[i] = ((char*)va)[i];
    }
    return;
}
```

1.3 修改`sys_call`和`_traps`

`sys_call`需要额外处理SYS_CLONE逻辑

```
else if (regs->x[17] == SYS_CLONE) {
    regs->x[10] = sys_clone(regs);
}
```

`_traps`需要`__ret_from_fork`符号，让child task在做`schedule`时能从中断返回后开始执行

```
_traps:
  ...(save registers)
  call trap_handler
__ret_from_fork:
  ...(load registers)
  sret
```

2 编译及测试

给的三个测试点都通过

```
...setup_vm_final done.
[S] Initialized: pid: 1, priority: 1, counter: 0
2022 Hello RISC-V
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
00000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010158
[S] New task: 2
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011978, sepc: 0
0000000000101f0
[U-PARENT] pid: 1 is running!, global_variable: 0
[U-PARENT] pid: 1 is running!, global_variable: 1
switch to [PID = 2 PRIORITY = 1 COUNTER = 1]
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011978, sepc: 0
00000000001018c
[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
```

```
[S] Initialized: pid: 1, priority: 1, counter: 0
2022 Hello RISC-V
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
00000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010158
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011a00, sepc: 0
00000000001017c
[U] pid: 1 is running!, global_variable: 0
[U] pid: 1 is running!, global_variable: 1
[U] pid: 1 is running!, global_variable: 2
[S] New task: 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
switch to [PID = 2 PRIORITY = 1 COUNTER = 1]
[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
```

```
[S] Initialized: pid: 1, priority: 1, counter: 0
2022 Hello RISC-V
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
00000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
000000000010158
[S] Supervisor Page Fault, scause: 000000000000000d, stval: 0000000000011930, sepc: 0
000000000010174
[U] pid: 1 is running!, global_variable: 0
[S] New task: 2
[U] pid: 1 is running!, global_variable: 1
[S] New task: 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
switch to [PID = 2 PRIORITY = 1 COUNTER = 1]
[U] pid: 2 is running!, global_variable: 1
[S] New task: 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
switch to [PID = 3 PRIORITY = 1 COUNTER = 1]
[U] pid: 3 is running!, global_variable: 2
[U] pid: 3 is running!, global_variable: 3
[U] pid: 3 is running!, global_variable: 4
switch to [PID = 4 PRIORITY = 1 COUNTER = 1]
```

斐波那契数列也通过

```
SET [PID = 1 PRIORITY = 1 COUNTER = 1]
switch to [PID = 1 PRIORITY = 1 COUNTER = 1]
[S] Supervisor Page Fault, scause: 000000000000000c, stval: 00000000000100e8, sepc: 0
00000000000100e8
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0
00000000000101e0
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000000000011bd8, sepc: 0
000000000010214
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000000000012000, sepc: 0
000000000010214
[S] Supervisor Page Fault, scause: 000000000000000f, stval: 0000000000013000, sepc: 0
000000000010214
[S] New task: 2
[U] fork returns 2
[U-PARENT] pid: 1 is running! the 0th fibonacci number is 1 and the number @ 1000 in
the large array is 20
[U-PARENT] pid: 1 is running! the 1th fibonacci number is 1 and the number @ 999 in t
he large array is 999
[U-PARENT] pid: 1 is running! the 2th fibonacci number is 1 and the number @ 998 in t
he large array is 998
[U-PARENT] pid: 1 is running! the 3th fibonacci number is 2 and the number @ 997 in t
he large array is 997
[U-PARENT] pid: 1 is running! the 4th fibonacci number is 3 and the number @ 996 in t
he large array is 996
[U-PARENT] pid: 1 is running! the 5th fibonacci number is 5 and the number @ 995 in t
he large array is 995
[U-PARENT] pid: 1 is running! the 6th fibonacci number is 8 and the number @ 994 in t
he large array is 994
```

```
switch to [PID = 2 PRIORITY = 1 COUNTER = 1]
[U] fork returns 0
[U-CHILD] pid: 2 is running! the 0th fibonacci number is 1 and the number @ 1000 in t
he large array is 20
[U-CHILD] pid: 2 is running! the 1th fibonacci number is 1 and the number @ 999 in th
e large array is 999
[U-CHILD] pid: 2 is running! the 2th fibonacci number is 1 and the number @ 998 in th
e large array is 998
[U-CHILD] pid: 2 is running! the 3th fibonacci number is 2 and the number @ 997 in th
e large array is 997
[U-CHILD] pid: 2 is running! the 4th fibonacci number is 3 and the number @ 996 in th
e large array is 996
[U-CHILD] pid: 2 is running! the 5th fibonacci number is 5 and the number @ 995 in th
e large array is 995
[U-CHILD] pid: 2 is running! the 6th fibonacci number is 8 and the number @ 994 in th
e large array is 994
[U-CHILD] pid: 2 is running! the 7th fibonacci number is 13 and the number @ 993 in t
he large array is 993
[U-CHILD] pid: 2 is running! the 8th fibonacci number is 21 and the number @ 992 in t
he large array is 992
[U-CHILD] pid: 2 is running! the 9th fibonacci number is 34 and the number @ 991 in t
he large array is 991
[U-CHILD] pid: 2 is running! the 10th fibonacci number is 55 and the number @ 990 in
the large array is 990
[U-CHILD] pid: 2 is running! the 11th fibonacci number is 89 and the number @ 989 in
```

3 思考题

1.*参考 task_init 创建一个新的 task，将的 parent task 的整个页复制到新创建的 task_struct 页上*。这一步复制了哪些东西?

复制了将parent task的`task_struct`：

```
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;
    uint64 satp;
    pagetable_t pgd;
    uint64 vma_cnt;                       /* 下面这个数组里的元素的数量 */
    struct vm_area_struct vmas[0];        /* 为什么可以开大小为 0 的数组?
                                这个定义可以和前面的 vma_cnt 换个位置吗? */
};
```

此时`pid`,`thread.ra/sp`,`satp`,`pgd`还是parent task的，同时vma应该映射的部分也还没映射，需要做后续调整。

2.*将 thread.ra 设置为* `__ret_from_fork`，*并正确设置* `thread.sp`。仔细想想，这个应该设置成什么值？可以根据 child task 的返回路径来倒推。

应该设置成parent task此时的sp，即目前的栈顶:`pt_regs`。因为child task的返回路径是:**_switch_to->_ret_from_fork(in _traps)->user program**，因此`__switch_to`需要给`__ret_from_fork`正确的栈指针，具体还要看每个人`_traps`的设计，总之要给`__ret_from_fork`执行时正确的栈指针,一般而言是`pt_regs`。

3.*利用参数 regs 来计算出 child task 的对应的 pt_regs 的地址，并将其中的 a0, sp, sepc 设置成正确的值*。为什么还要设置 sp?

`thread.sp`是为了保证`__ret_from_fork`执行时有正确的栈指针，而设置栈内的sp(即pt_regs->sp)是为了在`__ret_from_fork`的出栈时能够ld到正确的sp。