Lab 6: fork机制

3210105944 黄锦骏

1. 实验目的

- 为 task 加入 **fork** 机制，能够支持通过 **fork** 创建新的用户态 task 。

2. 实验环境

- Environment in previous labs.

3 实验步骤

3.1 准备工作

- 此次实验基于 lab5 同学所实现的代码进行。
- 从 repo 同步以下文件夹: user 并按照以下步骤将这些文件正确放置。

```
.
└── user
    ├── Makefile
    ├── getpid.c
    ├── link.lds
    ├── printf.c
    ├── start.S
    ├── stddef.h
    ├── stdio.h
    ├── syscall.h
    └── uapp.S
```

- 修改 `task_init` 函数中修改为仅初始化一个 task ， 之后其余的 task 均通过 `fork` 创建。

`task_init`函数修改如下:

```
void task_init() {
    ...
    // 1. 参考 idle 的设置, 为 task[1]进行初始化，其余赋值成NULL
    for(int i = 1; i < NR_TASKS; i++){
        if(i != 1){
            task[i] = NULL;
            continue;
        }
        task[i] = (struct task_struct *)kalloc();
        task[i]->state = TASK_RUNNING;
        task[i]->pid = i;
        task[i]->counter = task_test_counter[i];
        task[i]->priority = task_test_priority[i];
        task[i]->thread.ra = (uint64)__dummy;
        task[i]->thread.sp = (uint64)task[i] + 4096;

```

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

        load_program(task[i]);
        printk("[S] Initialized: pid: %d, priority: %d, counter: %d\n",
task[i]->pid, task[i]->priority, task[i]->counter);
    }

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

3.1 实现 fork()

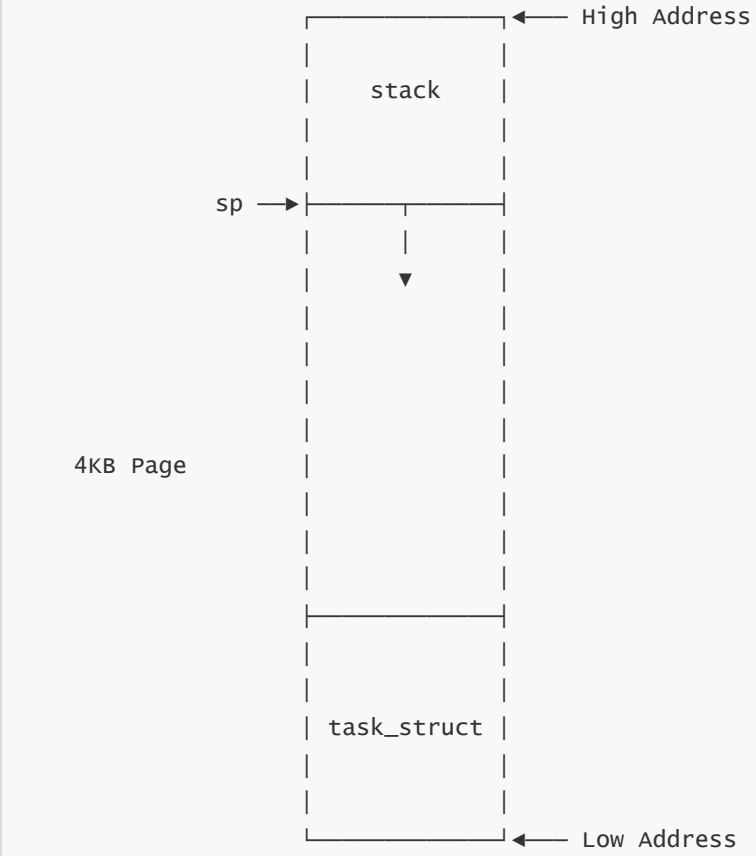
3.1.1 sys_clone

Fork 在早期的 Linux 中就被指定了名字，叫做 `clone`，

```
#define SYS_CLONE 220
```

我们在实验原理中说到，fork 的关键在于状态和内存的复制。我们不仅需要完整地**深拷贝**一份页表以及 VMA 中记录的用户态的内存，还需要复制内核态的寄存器状态和内核态的内存。并且在最后，需要将 task “伪装”成是因为调度而进入了 Ready Queue。

回忆一下我们是怎样使用 `task_struct` 的，我们并不是分配了一块刚好大小的空间，而是分配了一整个页，并将页的高处作为了 task 的内核态栈。



也就是说，内核态的所有数据，包括了栈、陷入内核态时的寄存器，还有上一次发生调度时，调用 `switch_to` 时的 `thread_struct` 信息，都被存在了这短短的 4K 内存中。这给我们的实现带来了很大的方便，这 4K 空间里的数据就是我们需要的所有所有内核态数据了！（当然如果你没有进行步骤 4.0，那还需要开一个页并复制一份 `thread_info` 信息）

除了内核态之外，你还需要**深拷贝**一份页表，并遍历页表中映射到 parent task 用户地址空间的页表项（为了减小开销，你需要根据 parent task 的 vmas 来 walk page table），这些应该由 parent task 专有的页，如果已经分配并且映射到 parent task 的地址空间中了，就需要你另外分配空间，并从原来的内存中拷贝数据到新开辟的空间，然后将新开辟的页映射到 child task 的地址空间中。想想为什么只要拷贝那些已经分配并映射的页，那些本来应该被分配并映射，但是暂时还没有因为 Page Fault 而被分配并映射的页怎么办?

对于这些页，由于已经做过了`do_mmap`，所以`task_struct`中的vmas保存有相关的虚拟内存分配信息，需要访问那些页的时候，就可以交给Page Fault进行分配。

3.1.2 __ret_from_fork

让 fork 出来的 task 被正常调度是本实验**最重要**的部分。我们在 Lab3 中有一道思考题

> 2. 当线程第一次调用时, 其 `ra` 所代表的返回点是 `__dummy`。那么在之后的线程调用中 `context_switch` 中, `ra` 保存/恢复的函数返回点是什么呢? 请同学用 gdb 尝试追踪一次完整的线程切换流程, 并关注每一次 `ra` 的变换 (需要截图)。

经过了对这个问题的思考，我们可以认识到，一个程序第一次被调度时，其实是可以选择返回到执行哪一个位置指令的。例如我们当时执行的 `__dummy`，就替代了正常从 `switch_to` 返回的执行流。这次我们同样使用这个 trick，通过修改 `task_struct->thread.ra`，让程序 `ret` 时，直接跳转到我们设置的 symbol `__ret_from_fork`。

我们在 `_traps` 中的 `jal x1, trap_handler` 后面插入一个符号：

`__ret_from_fork` 实现如下：

- 具体而言利用 `sp` 寄存器，从内核态的栈上恢复出我们在 `sys_clone` 时拷贝到新的 task 的栈上的，原本在 context switch 时被压入父 task 的寄存器值
- 然后切换到用户栈

```
    .global _traps
_traps:
    ...
   jal x1, trap_handler
    .global __ret_from_fork
__ret_from_fork:
    ld t0, 16(x2)
    csrw sepc, t0

    ld  x0,  24(x2)
    ld  x1,  32(x2)
    ld  x3,  48(x2)
    ld  x4,  56(x2)
    ld  x5,  64(x2)
    ld  x6,  72(x2)
    ld  x7,  80(x2)
    ld  x8,  88(x2)
    ld  x9,  96(x2)
    ld x10,  104(x2)
```

```
    ld x11, 112(x2)
    ld x12, 120(x2)
    ld x13, 128(x2)
    ld x14, 136(x2)
    ld x15, 144(x2)
    ld x16, 152(x2)
    ld x17, 160(x2)
    ld x18, 168(x2)
    ld x19, 176(x2)
    ld x20, 184(x2)
    ld x21, 192(x2)
    ld x22, 200(x2)
    ld x23, 208(x2)
    ld x24, 216(x2)
    ld x25, 224(x2)
    ld x26, 232(x2)
    ld x27, 240(x2)
    ld x28, 248(x2)
    ld x29, 256(x2)
    ld x30, 264(x2)
    ld x31, 272(x2)
    ld  x2,  40(x2)

    csrr t1, sscratch
    csrw sscratch,sp
    add sp, t1, x0
    sret
```

继续回忆，我们的 `__switch_to` 逻辑的后半段，就是从 `task_struct->thread` 中恢复 callee-saved registers 的值，其中正包括了我们恢复寄存器值所需要的 sp。

自此我们知道，我们可以利用这两个寄存器，完成一个类似于 ROP(return oriented programming) 的操作。也就是说，我们通过控制 `ra` 寄存器，来控制程序的执行流，让它跳转到 context switch 的后半段；通过控制 `sp` 寄存器，从内核态的栈上恢复出我们在 `sys_clone` 时拷贝到新的 task 的栈上的，原本在 context switch 时被压入父 task 的寄存器值，然后通过 sret 直接跳回用户态执行用户态程序。

于是，父 task 的返回路径是这样的：`sys_clone->trap_handler->_traps->user program`，而我们新 `fork` 出来的 task，要以这样的路径返回: `__switch_to->__ret_from_fork(in _traps)->user program`.

3.1.3 Code Skeleton

某知名体系结构课程老师说过，skeleton 是给大家参考用的，不是给大家直接抄的。接下来我们给大家的代码框架**理论上**可以直接运行，因为在写作实验文档前某助教刚刚自己完整实现了一次。但是我们的当前的框架是最 Lean 的，也就是说虽然一定能跑，但是同学们照着这个来写可能会有一些不方便，同学可以自行修改框架，来更好地贴合自己的实现。

我们要在存储所有 task 的数组 `task` 中寻找一个空闲的位置。我们用最简单的管理方式，将原本的 `task` 数组的大小开辟成 16, IDLE task 和 初始化时新建的 task 各占用一个，剩余 14 个全部赋值成 NULL。 如果 `task[pid] == NULL`，说明这个 pid 还没有被占用，可以作为新 task 的 pid，并将 `task[pid]` 赋值为新的 `struct task_struct*`。由于我们的实验中不涉及 task 的销毁，所以这里的逻辑可以只管填充，不管擦除。

在实现中，你需要始终思考的问题是，怎么才能够**让新创建的 task 获得调度后，正确地跳转到 `__ref_from_fork`，并且利用 `sp` 正确地从内存中取值。**为了简单起见，`sys_clone` 只接受一个参数 `pt_regs *`，下面是代码框架：

`sys_clone` 具体实现如下：

- 遍历task[l]找到空余的结构体来分配新进程
- 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, 根据current task的sp来设置新task的 `thread.sp = (uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp);`
- 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，这里设置sp是因为在context_switch时，需要从pt_regs中拿到正确的sp地址
- 为 child task 申请 user stack, 并将 parent task 的 user stack 数据复制到其中。
- 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。这里仅仅只用将之前已经分配过的内核态页表复制到该新进程的页表即可。
- 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，此时不用再次分配用户栈
- 返回新分配进程的pid

```
uint64_t sys_clone(struct pt_regs *regs) {
   ...
   if(func == 220){
        int free_index = -1;
        for(int i = 2; i < NR_TASKS; i++){
            if(task[i] == NULL){
                free_index = i;
                break;
            }
        }
        if(free_index == -1){
            printk("Task is full, sys_clone can't find a empty task\n");
            while(1);
        }
        /*
        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?）。将 thread.ra 设置为
        __ret_from_fork, 并正确设置 thread.sp
        */
        task[free_index] = (struct task_struct *)kalloc();
        uint64 *src = (uint64 *)current;
        uint64 *dst = (uint64 *)task[free_index];
        for(int i = 0; i < PGSIZE / 8; i++)
            dst[i] = src[i];

        task[free_index]->state = current->state;
        task[free_index]->pid = free_index;
        task[free_index]->counter = current->counter;
        task[free_index]->priority = current->priority;
        task[free_index]->thread_info = current->thread_info;

        uint64 current_sp = (uint64)regs;
```

```
        uint64 sscratch = csr_read(sscratch);
        uint64 sepc = csr_read(sepc);
        // printk("parent_sepc: %lx\n", sepc);
        // printk("parent_sscratch: %lx\n", sscratch);
        // printk("used_sepc: %lx\n", regs->sepc);
        // printk("used_sscratch: %lx\n", current->thread.sscratch);

        task[free_index]->thread.ra = __ret_from_fork;
        task[free_index]->thread.sp = (uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp);
        for(int i = 0; i < 12; i++)
            task[free_index]->thread.s[i] = current->thread.s[i];
        task[free_index]->thread.sepc = regs->sepc;
        task[free_index]->thread.sstatus = current->thread.sstatus;
        task[free_index]->thread.sscratch = sscratch;

        /*
         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(为什么还要设置 sp?)
        */
        struct pt_regs* child_regs = (uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp);
        child_regs->x[10] = 0;
        child_regs->x[2] = (uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp);
        child_regs->sepc = regs->sepc;

        // printk("child_Regs->x[10] addr: %lx\n", (uint64)&child_regs->x[10]);

        /*、
        3. 为 child task 申请 user stack，并将 parent task 的 user stack
        数据复制到其中。
        */
        uint64 *user_st = (uint64 *)kalloc();

        src = (uint64 *)(USER_END - (uint64)PGSIZE);
        dst = (uint64 *)user_st;
        for(int i = 0; i < PGSIZE / 8; i++){
            dst[i] = src[i];
        }
        // printk("pgd: %lx\n", (uint64)task[free_index]->pgd);
        // printk("satp: %lx\n", (((uint64)task[free_index]->pgd - PA2VA_OFFSET) >> 12) | ((uint64)0x8 << 60));
        // printk("user_stack_start: %lx\n",  (uint64)USER_END - PGSIZE);
        // printk("user_stack_end: %lx\n",  (uint64)USER_END);
        // printk("user_stack_pm: %lx\n", (uint64)user_st - PA2VA_OFFSET);


        /*4. 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*/
        task[free_index]->pgd = (pagetable_t)kalloc();
        memset(task[free_index]->pgd, 0, PGSIZE);
         for(int j = 0; j < PGSIZE / 8; j++){
            task[free_index]->pgd[j]=swapper_pg_dir[j];
```

```
        }

        create_mapping(task[free_index]->pgd, (uint64)USER_END - PGSIZE, (uint64)user_st - PA2VA_OFFSET, PGSIZE, 23);

        /*5. 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*/

        task[free_index]->vma_cnt = current->vma_cnt;
        for(int i = 0; i < current->vma_cnt; i++){
            task[free_index]->vmas[i].vm_start = current->vmas[i].vm_start;
            task[free_index]->vmas[i].vm_end = current->vmas[i].vm_end;
            task[free_index]->vmas[i].vm_flags = current->vmas[i].vm_flags;
            task[free_index]->vmas[i].vm_content_offset_in_file = current->vmas[i].vm_content_offset_in_file;
            task[free_index]->vmas[i].vm_content_size_in_file = current->vmas[i].vm_content_size_in_file;

            uint64 page_num = (current->vmas[i].vm_start - PGROUNDDOWN((uint64)current->vmas[i].vm_start) + current->vmas[i].vm_content_size_in_file + PGSIZE - 1) / PGSIZE;
            // printk("pid: %lx, current_pgd: %lx\n", current->pid, current->pgd);

            uint64 uapp = alloc_pages(page_num);
            memset(uapp, 0, page_num*PGSIZE);
            int annom_flag = current->vmas[i].vm_flags & VM_ANONYM;

            int *global_variable = (int *)(0x0000000000011944);
            // printk("global_variable[parent]: %d\n", *global_variable);
            if(!annom_flag){
                for(int t = 0; t < page_num; t++){
                    uint64* src = (uint64*)(PGROUNDDOWN((uint64)current->vmas[i].vm_start) + t * PGSIZE);
                    uint64* src2 = (uint64*)(ramdisk_start + t * PGSIZE + current->vmas[i].vm_content_offset_in_file);
                    uint64* dst = (uint64*)(uapp + t * PGSIZE);
                    for (int k = 0; k < PGSIZE / 8; k++) {
                        dst[k] = src[k];
                    }
                }
                create_mapping(task[free_index]->pgd, PGROUNDDOWN((uint64)current->vmas[i].vm_start), uapp - PA2VA_OFFSET, PGSIZE*page_num, current->vmas[i].vm_flags | 0b10001);
                // printk("uapp: %lx\n", (uint64)PGROUNDDOWN((uint64)current->vmas[i].vm_start));
                // printk("uapp_end: %lx\n", (uint64)(PGROUNDDOWN((uint64)current->vmas[i].vm_start) + PGSIZE * page_num));
                // printk("page_num: %ld\n", page_num);
            }
        }

        /*6. 返回子 task 的 pi*/
        regs->x[10] = free_index;
        printk("[S] New task: %ld\n", free_index);
```

```
    }
}
```

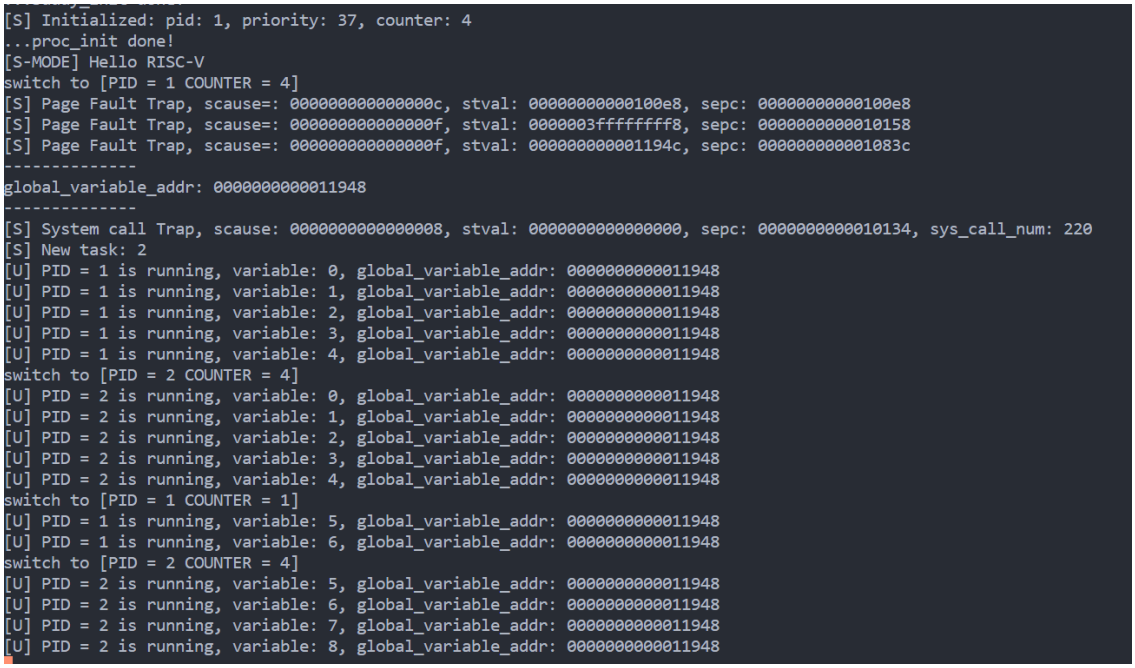
3.2 编译及测试

在测试时，由于大家电脑性能都不一样，如果出现了时钟中断频率比用户打印频率高很多的情况，可以减少用户程序里的 while 循环的次数来加快打印。这里的实例仅供参考，只要 OS 和用户态程序运行符合你的预期，那就是正确的。

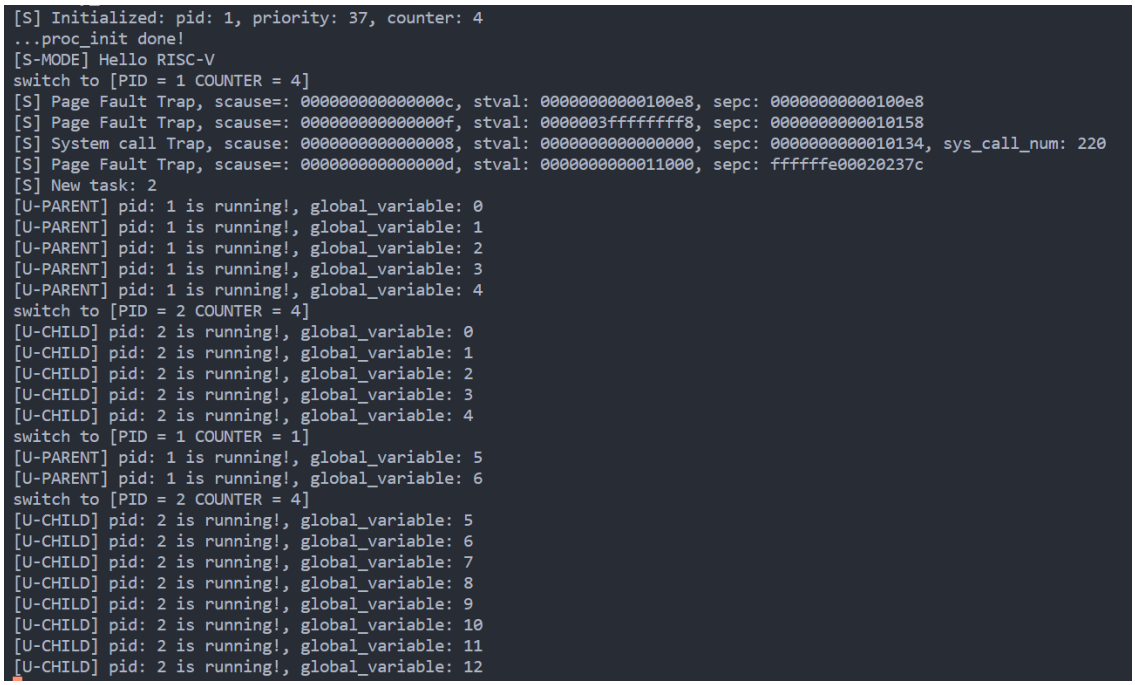
可以看到子进程继承了父进程在fork前的`global_variable`，`something_large_here`数组以及其他变量。

测试结果：

- main-1：一共发生了三次page fault



- main-2：一共发生了三次page faul



- main-3：一共发生了三次page fault

```
..buddy_init done!
[S] Initialized: pid: 1, priority: 37, counter: 4
...proc_init done!
[S-MODE] Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffff8, sepc: 0000000000010158
[S] Page Fault Trap, scause=: 000000000000000d, stval: 0000000000011a00, sepc: 000000000001017c
[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] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[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
[U-PARENT] pid: 1 is running!, global_variable: 6
[U-PARENT] pid: 1 is running!, global_variable: 7
switch to [PID = 2 COUNTER = 4]
[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
switch to [PID = 1 COUNTER = 1]
switch to [PID = 2 COUNTER = 4]
[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-CHILD] pid: 2 is running!, global_variable: 12
[U-CHILD] pid: 2 is running!, global_variable: 13
[U-CHILD] pid: 2 is running!, global_variable: 14
[U-CHILD] pid: 2 is running!, global_variable: 15
[U-CHILD] pid: 2 is running!, global_variable: 16
switch to [PID = 1 COUNTER = 5]
[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-PARENT] pid: 1 is running!, global_variable: 12
switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 17
[U-CHILD] pid: 2 is running!, global_variable: 18
[U-CHILD] pid: 2 is running!, global_variable: 19
[U-CHILD] pid: 2 is running!, global_variable: 20
switch to [PID = 1 COUNTER = 9]
[U-PARENT] pid: 1 is running!, global_variable: 13
```

- main-4：一共发生了三次page fault

```
[S] Initialized: pid: 1, priority: 37, counter: 4
..proc_init done!
[S-MODE] Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffff8, sepc: 0000000000010158
[S] Page Fault Trap, scause=: 000000000000000d, stval: 0000000000011930, sepc: 0000000000010174
[U] pid: 1 is running!, global_variable: 0
[S] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[S] New task: 2
[U] pid: 1 is running!, global_variable: 1
[S] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[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
[U] pid: 1 is running!, global_variable: 5
switch to [PID = 2 COUNTER = 4]
[U] pid: 2 is running!, global_variable: 1
[S] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[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
[U] pid: 2 is running!, global_variable: 5
switch to [PID = 3 COUNTER = 4]
[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
switch to [PID = 4 COUNTER = 4]
[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
switch to [PID = 1 COUNTER = 1]
[U] pid: 1 is running!, global_variable: 6
switch to [PID = 2 COUNTER = 4]
[U] pid: 2 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
switch to [PID = 4 COUNTER = 4]
[U] pid: 4 is running!, global_variable: 6
[U] pid: 4 is running!, global_variable: 7
[U] pid: 4 is running!, global_variable: 8
[U] pid: 4 is running!, global_variable: 9
```

- 附加测试：一共发生了五次page fault

```
[S] Initialized: pid: 1, priority: 37, counter: 4
...proc_init done!
[S-MODE] Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 00000000000101e0
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000011bd8, sepc: 0000000000010214
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000012000, sepc: 0000000000010214
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000013000, sepc: 0000000000010214
[S] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[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 the large array is 999
[U-PARENT] pid: 1 is running! the 2th fibonacci number is 1 and the number @ 998 in the large array is 998
[U-PARENT] pid: 1 is running! the 3th fibonacci number is 2 and the number @ 997 in the large array is 997
[U-PARENT] pid: 1 is running! the 4th fibonacci number is 3 and the number @ 996 in the large array is 996
switch to [PID = 2 COUNTER = 4]
[U] fork returns 0
[U-CHILD] pid: 2 is running! the 0th fibonacci number is 1 and the number @ 1000 in the large array is 20
[U-CHILD] pid: 2 is running! the 1th fibonacci number is 1 and the number @ 999 in the large array is 999
[U-CHILD] pid: 2 is running! the 2th fibonacci number is 1 and the number @ 998 in the large array is 998
[U-CHILD] pid: 2 is running! the 3th fibonacci number is 2 and the number @ 997 in the large array is 997
[U-CHILD] pid: 2 is running! the 4th fibonacci number is 3 and the number @ 996 in the large array is 996
[U-CHILD] pid: 2 is running! the 5th fibonacci number is 5 and the number @ 995 in the large array is 995
switch to [PID = 1 COUNTER = 1]
[U-PARENT] pid: 1 is running! the 5th fibonacci number is 5 and the number @ 995 in the large array is 995
[U-PARENT] pid: 1 is running! the 6th fibonacci number is 8 and the number @ 994 in the large array is 994
switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running! the 6th fibonacci number is 8 and the number @ 994 in the large array is 994
[U-CHILD] pid: 2 is running! the 7th fibonacci number is 13 and the number @ 993 in the large array is 993
[U-CHILD] pid: 2 is running! the 8th fibonacci number is 21 and the number @ 992 in the large array is 992
[U-CHILD] pid: 2 is running! the 9th fibonacci number is 34 and the number @ 991 in the 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 the large array is 989
[U-CHILD] pid: 2 is running! the 12th fibonacci number is 144 and the number @ 988 in the large array is 988
[U-CHILD] pid: 2 is running! the 13th fibonacci number is 233 and the number @ 987 in the large array is 987
[U-CHILD] pid: 2 is running! the 14th fibonacci number is 377 and the number @ 986 in the large array is 986
[U-CHILD] pid: 2 is running! the 15th fibonacci number is 610 and the number @ 985 in the large array is 985
[U-CHILD] pid: 2 is running! the 16th fibonacci number is 987 and the number @ 984 in the large array is 984
[U-CHILD] pid: 2 is running! the 17th fibonacci number is 1597 and the number @ 983 in the large array is 983
```

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?

   这一步复制了`task_struct`的state, counter, priority, thread_info, thread，其中thread中的数据不是完全复制，而应该如下图所示：

   ```
   uint64 current_sp = (uint64)regs;
   uint64 sscratch = csr_read(sscratch);
   uint64 sepc = csr_read(sepc);

   task[free_index]->thread.ra = __ret_from_fork;
   task[free_index]->thread.sp = (uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp);
   for(int i = 0; i < 12; i++)
       task[free_index]->thread.s[i] = current->thread.s[i];
   task[free_index]->thread.sepc = regs->sepc;
   task[free_index]->thread.sstatus = current->thread.sstatus;
   task[free_index]->thread.sscratch = sscratch;
   ```

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。

   `thread.sp`应该设置为`(uint64)task[free_index] + PGSIZE - ((uint64)current + PGSIZE - (uint64)current_sp)`, 即利用原进程的内核sp计算新进程的内核sp

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?

   因为我们在`__switch_to`的后半部分时，需要从`pt_regs`中加载sp，从而在`__ret_from_fork`中利用sp加载`sys_clone`复制到新进程栈上的寄存器值

更多测试样例

下面是同学提供的测试样例，不强制要求大家都运行一遍。但是如果想增强一下对自己写的代码的信心，可以尝试替换`main`并运行。如果你有其他适合用来测试的代码，欢迎为仓库做出贡献。

lhjgg 给出的样例：

```c
#define LARGE 1000

unsigned long something_large_here[LARGE] = {0};

int fib(int times) {
  if (times <= 2) {
    return 1;
  } else {
    return fib(times - 1) + fib(times - 2);
  }
}

int main() {
  for (int i = 0; i < LARGE; i++) {
    something_large_here[i] = i;
  }
  int pid = fork();
  printf("[U] fork returns %d\n", pid);

  if (pid == 0) {
    while(1) {
      printf("[U-CHILD] pid: %ld is running! the %dth fibonacci number is %d and the number @ %d in the large array is %d\n", getpid(), global_variable, fib(global_variable), LARGE - global_variable, something_large_here[LARGE - global_variable]);
      global_variable++;
      for (int i = 0; i < 0xFFFFFF; i++);
    }
  } else {
    while (1) {
      printf("[U-PARENT] pid: %ld is running! the %dth fibonacci number is %d and the number @ %d in the large array is %d\n", getpid(), global_variable, fib(global_variable), LARGE - global_variable, something_large_here[LARGE - global_variable]);
      global_variable++;
      for (int i = 0; i < 0xFFFFFF; i++);
    }
  }
}
```

测试结果：符合预期，子进程继承了父进程的数组

```
[S] Initialized: pid: 1, priority: 37, counter: 4
...proc_init done!
[S-MODE] Hello RISC-V
switch to [PID = 1 COUNTER = 4]
[S] Page Fault Trap, scause=: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000003fffffffe8, sepc: 00000000000101e0
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000011bd8, sepc: 0000000000010214
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000012000, sepc: 0000000000010214
[S] Page Fault Trap, scause=: 000000000000000f, stval: 0000000000013000, sepc: 0000000000010214
[S] System call Trap, scause: 0000000000000008, stval: 0000000000000000, sepc: 0000000000010134, sys_call_num: 220
[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 the large array is 999
[U-PARENT] pid: 1 is running! the 2th fibonacci number is 1 and the number @ 998 in the large array is 998
[U-PARENT] pid: 1 is running! the 3th fibonacci number is 2 and the number @ 997 in the large array is 997
[U-PARENT] pid: 1 is running! the 4th fibonacci number is 3 and the number @ 996 in the large array is 996
switch to [PID = 2 COUNTER = 4]
[U] fork returns 0
[U-CHILD] pid: 2 is running! the 0th fibonacci number is 1 and the number @ 1000 in the large array is 20
[U-CHILD] pid: 2 is running! the 1th fibonacci number is 1 and the number @ 999 in the large array is 999
[U-CHILD] pid: 2 is running! the 2th fibonacci number is 1 and the number @ 998 in the large array is 998
[U-CHILD] pid: 2 is running! the 3th fibonacci number is 2 and the number @ 997 in the large array is 997
[U-CHILD] pid: 2 is running! the 4th fibonacci number is 3 and the number @ 996 in the large array is 996
[U-CHILD] pid: 2 is running! the 5th fibonacci number is 5 and the number @ 995 in the large array is 995
switch to [PID = 1 COUNTER = 1]
[U-PARENT] pid: 1 is running! the 5th fibonacci number is 5 and the number @ 995 in the large array is 995
[U-PARENT] pid: 1 is running! the 6th fibonacci number is 8 and the number @ 994 in the large array is 994
switch to [PID = 2 COUNTER = 4]
[U-CHILD] pid: 2 is running! the 6th fibonacci number is 8 and the number @ 994 in the large array is 994
[U-CHILD] pid: 2 is running! the 7th fibonacci number is 13 and the number @ 993 in the large array is 993
[U-CHILD] pid: 2 is running! the 8th fibonacci number is 21 and the number @ 992 in the large array is 992
[U-CHILD] pid: 2 is running! the 9th fibonacci number is 34 and the number @ 991 in the 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 the large array is 989
[U-CHILD] pid: 2 is running! the 12th fibonacci number is 144 and the number @ 988 in the large array is 988
[U-CHILD] pid: 2 is running! the 13th fibonacci number is 233 and the number @ 987 in the large array is 987
[U-CHILD] pid: 2 is running! the 14th fibonacci number is 377 and the number @ 986 in the large array is 986
[U-CHILD] pid: 2 is running! the 15th fibonacci number is 610 and the number @ 985 in the large array is 985
[U-CHILD] pid: 2 is running! the 16th fibonacci number is 987 and the number @ 984 in the large array is 984
[U-CHILD] pid: 2 is running! the 17th fibonacci number is 1597 and the number @ 983 in the large array is 983
```