Lab 5

练习0：填写已有实验

> 本实验依赖实验2/3/4。请把你做的实验2/3/4的代码填入本实验中代码中有“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行lab5的测试应用程序，可能需对已完成的实验2/3/4的代码进行进一步改进。

本次实验相较于过去的填写有所不同——需要对原有代码进行进一步改进

首先，在 `alloc_proc` 函数中，我们新增了2个成员变量：

```
proc->wait_state = 0; // 初始化进程等待状态
proc->cptr = proc->optr = proc->yptr = NULL; // 设置进程指
```

`wait_state` 表示进程当前的等待状态，0表示进程不在等待状态；

```
proc->cptr = proc->optr = proc->yptr = NULL; // 设置进程指针
```

`proc->cptr`：指向当前进程的第一个子进程；

`proc->optr`：指向当前进程在父进程子进程链表中的上一个兄弟进程；

`proc->yptr`：指向当前进程在父进程子进程链表中的下一个兄弟进程。

在 `do_fork` 函数中，我们相较于Lab 4做出如下修改：

首先在创建进程（分配进程）后添加 assert(current->wait_state == 0); 确保进程处在等待状态；

然后再进行下面这些修改——将原先简单的进程计数和添加改为 set_links 函数，设置进程链接，即

```
// list_add(&proc_list, &proc->list_link); // 这才是正确的打开方式（bushi）
// nr_process++; // 更新进程数
set_links(proc); // 设置进程链接
```

练习1: 加载应用程序并执行（需要编码）

> **do_execv**函数调用`load_icode`（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序。你需要补充`load_icode`的第6步，建**立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好`proc_struct`结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。**需设置正确的trapframe内容。
>
> 请在实验报告中简要说明你的设计实现过程。
>
> - 请简要描述这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

- load_icode：被do_execve调用，完成加载放在内存中的执行程序到进程空间，这涉及到对页表等的修改，分配用户栈
- do_execve：先回收自身所占用户空间，然后调用load_icode，用新的程序覆盖内存空间，形成一个执行新程序的新进程

```
```

```
/* load_icode - load the content of binary program(ELF format) as the new content
of current process
 * @binary:  the memory addr of the content of binary program
 * @size:  the size of the content of binary program
 */
static int
load_icode(unsigned char *binary, size_t size) {
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    //(3) copy TEXT/DATA section, build BSS parts in binary to memory space of
process
    struct Page *page;
    //(3.1) get the file header of the bianry program (ELF format)
    struct elfhdr *elf = (struct elfhdr *)binary;
    //(3.2) get the entry of the program section headers of the bianry program
(ELF format)
    struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff);
    //(3.3) This program is valid?
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    uint32_t vm_flags, perm;
    struct proghdr *ph_end = ph + elf->e_phnum;
    for (; ph < ph_end; ph ++) {
    //(3.4) find every program section headers
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            // continue ;
        }
    //(3.5) call mm_map fun to setup the new vma ( ph->p_va, ph->p_memsz)
        vm_flags = 0, perm = PTE_U | PTE_V;
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        // modify the perm bits here for RISC-V
```

```
        if (vm_flags & VM_READ) perm |= PTE_R;
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        if (vm_flags & VM_EXEC) perm |= PTE_X;
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        unsigned char *from = binary + ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

        ret = -E_NO_MEM;

     //(3.6) alloc memory, and  copy the contents of every program section (from,
from+end) to process's memory (la, la+end)
        end = ph->p_va + ph->p_filesz;
     //(3.6.1) copy TEXT/DATA section of bianry program
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memcpy(page2kva(page) + off, from, size);
            start += size, from += size;
        }

      //(3.6.2) build BSS section of binary program
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    //(4) build user stack memory
```

```
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !=
0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    //(5) set current process's mm, sr3, and set CR3 reg = physical addr of Page
Directory
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    /* LAB5:EXERCISE1 2213787
     * should set tf->gpr.sp, tf->epc, tf->status
     * NOTICE: If we set trapframe correctly, then the user level process can
return to USER MODE from kernel. So
     *          tf->gpr.sp should be user stack top (the value of sp)
     *          tf->epc should be entry point of user program (the value of sepc)
     *          tf->status should be appropriate for user program (the value of
sstatus)
     *          hint: check meaning of SPP, SPIE in SSTATUS, use them by
SSTATUS_SPP, SSTATUS_SPIE(defined in risv.h)
     */
tf->gpr.sp = USTACKTOP; // 用户栈的顶
tf->epc = elf->e_entry; //设置用户程序的入口地址
tf->status = (read_csr(sstatus) & ~SSTATUS_SPP & ~SSTATUS_SPIE);

    ret = 0;
out:
    return ret;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

laod_icode()函数实现了将新程序加载到当前的进程中去。

它的主要任务为：

- 检查并创建新的内存管理结构mm和页目录pgdir。

- 解析ELF文件的头部和程序段，将程序段映射到进程的内存空间。
- 会为程序分配内存，并将程序段内容复制到进程内存。
- 处理BSS部分，将未初始化的内存区域填充为0.
- 建立用户栈，并为其分配内存、设置栈顶。
- 设置 `trapframe`，为进程的用户态执行做准备。

该函数的第六步主要是进行中断帧的设置：

1. `tf->gpr.sp：`将tf->gpr.sp设置为用户栈的顶部地址USTACKTOP，以便运行用户程序时可以正确地访问栈。
2. `tf->gpr.sp：`将tf->gpr.sp设置为用户程序的入口地址elf->e_entry，以便可以从正确位置开始。
3. `tf->status：`设置进程状态，以便正确切换到用户态。SPP表示处理器在发生异常或中断之前的特权级别，如果其为0，表示发生异常或者中断之前是用户模式（U mode），1则表示之前是S mode。SPIE表示发生异常或者中断之前的中断使能状态，0表示发生异常或者中断之前中断被禁用，1表示之前中断是可以使用的。

> 请简要描述这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

从用户态被创建到具体执行的关键步骤如下：

1. **进程创建**：加载程序的二进制文件，初始化内存和 `trapframe`。
2. **调度选择**：调度器选择目标进程切换为运行态。
3. **上下文切换**：恢复目标进程的上下文，包括页表和寄存器。
4. **进入用户态**：通过 `sret` 指令切换到用户模式并开始执行用户程序的第一条指令。

   该问题主要是关注上下文切换部分，接下来将讲述一下该过程：

我们在 `proc_init()` 函数里初始化进程的时候, 认为启动时运行的ucore程序, 是一个内核进程("第0个"内核进程), 并将其初始化为 `idleproc` 进程。然后我们新建了一个内核进程执行 `init_main()` 函数。

```
// kern/process/proc.c (lab5)
static int init_main(void *arg) {
    size_t nr_free_pages_store = nr_free_pages();
    size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    while (do_wait(0, NULL) == 0) {
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->optr ==
NULL);
    assert(nr_process == 2);
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));
```

```
    cprintf("init check memory pass.\n");
    return 0;
}
```

在`init_main()`中，使用`kernel_thread(user_main, NULL, 0)`创建了一个新的用户进程，并将`user_main`函数作为用户线程的入口。

```
// user_main - kernel thread used to exec a user program
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(exit);
#endif
    panic("user_main execve failed.\n");
}
```

在user_main()中执行了`kern_execve("exit", _binary_obj___user_exit_out_start,_binary_obj___user_exit_out_size)`这么一个函数，即加载了存储在这个位置的程序`exit`并在`user_main`这个进程里开始执行。这时`user_main`就从内核进程变成了用户进程。

```
// kernel_execve - do SYS_exec syscall to exec a user program called by user_main kernel_thread
static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int64_t ret=0, len = strlen(name);
 //   ret = do_execve(name, len, binary, size);
    asm volatile(
        "li a0, %1\n"
        "lw a1, %2\n"
        "lw a2, %3\n"
        "lw a3, %4\n"
        "lw a4, %5\n"
        "li a7, 10\n"
        "ebreak\n"    // 触发 `ebreak` 指令，触发系统调用
        "sw a0, %0\n"
        : "=m"(ret)
        : "i"(SYS_exec), "m"(name), "m"(len), "m"(binary), "m"(size)
        : "memory");
    cprintf("ret = %d\n", ret);
    return ret;
}
```

在`kernel_execve`函数中会触发`SYS_exec`系统调用来执行用户程序，它将程序名、程序长度、程序二进制数据以及大小传递给系统调用，并通过`ebreak`指令切换到内核模式。`SYS_exec`系统调用的目的是加载一个用户程序，并开始执行它。SYS_exec中会执行调用`do_execve`，`do_execve`中会调用`load_icode`来加载文件。

进程调度：（该问答题的答案）

当用户进行被创建后，它可能处于就绪队列中，当某个时刻调用了system选择了该用户进程时，说明它将从就绪状态变为执行状态，但是在具体执行应用程序第一条指令前，还需要做一些工作（主要是`do_execve`和`load_icode`函数）：

do_execve主要是负责将当前进程的内存空间清理并加载一个新的程序：

- 首先会检查用户程序的名称有效性，使用user_mem_check函数检查传入的`name`（程序名）是否可以被当前用户进程访问。
- 会限制程序名称的长度
- 存储程序名称，使用`memcpy`将`name`拷贝到`local_name`中。
- **清理当前进程的内存管理结构**：如果当前进程有内存管理结构（`mm != NULL`），则先切换到内核的页目录：`lcr3(boot_cr3)`；减少当前进程的内存引用计数，如果计数为 0，则表示没有其他进程使用该内存资源；退出当前进程的内存映射，释放虚拟内存的映射关系；释放进程的页表；销毁当前进程的内存管理结构`mm`；将当前进程的`mm`设置为`NULL`，表明当前进程不再持有内存管理结构。（在加载新程序之前，必须保证当前进程的内存管理结构`mm`是**空**的，因为`load_icode`会为当前进程分配新的内存空间。）
- 加载新程序的二进制数据：这部分主要是调用`load_icode`函数。
- 设置进程名称
- 如果`load_icode`加载新程序失败，则会进入`execve_exit`，执行错误处理：调用`do_exit`退出当前进程，传递错误代码`ret`。

load_icode主要是加载 ELF 格式的二进制程序到当前进程的内存空间中，并准备好程序的内存映射、栈、以及其他执行所需的环境：

- **检查当前进程的内存管理结构**：必须为空，因为在这个函数中会为其分配新的内存空间。
- **创建新的内存管理结构**：调用`mm_create()`为当前进程创建一个新的内存管理结构`mm`
- 创建新的页目录：调用`setup_pgdir(mm)`来为进程创建页目录，并将页目录的地址存储在`mm->pgdir`中。
- 解析 ELF 文件
- 遍历并加载程序段
- 设置内存映射权限
- 设置虚拟内存区域：使用`mm_map`将 ELF 程序段的虚拟地址（`p_va`）和内存大小（`p_memsz`）映射到进程的虚拟地址空间中。
- 构建用户栈
- **设置当前进程的内存管理结构**：首先增加内存管理结构`mm`的引用计数；将新的内存管理结构`mm`设置为当前进程的内存管理结构；设置`current->cr3`为页目录的物理地址，并通过`lcr3`切换到当前进程的页目录。
- **设置用户态的 trapframe**：`trapframe`是进程从内核态切换到用户态时的上下文保存结构，在里面设置了用户栈指针、用户程序的入口地址、进程状态。

以上工作就实现了构建用户程序运行的上下文，接下来要实现上下文切换：

我们在之前提到了“SYS_exec中会执行调用`do_execve`，`do_execve`中会调用`load_icode`来加载文件。”在系统调用`sys_exec`之后，在trap返回的时候调用了`sret`指令，这时只要`sstatus`寄存器的`SPP`二进制位为0，就会切换到U mode。

练习2：父进程复制自己的内存空间给子进程（需要编码）

> 创建子进程的函数`do_fork`在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过`copy_range`函数（位于kern/mm/pmm.c中）实现的，请补充`copy_range`的实现，确保能够正确执行。
>
> 请在实验报告中简要说明你的设计实现过程。
>
> - 如何设计实现`Copy on Write`机制？给出概要设计，鼓励给出详细设计。
>
> > Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

2.1问题1：

下面是调用`copy_range`的流程图：

```
do_fork:
    |--> alloc_proc: 分配进程控制块
    |--> setup_kstack: 分配内核栈
    |--> copy_mm: 复制或共享地址空间
          |--> 如果不共享：
          |    |--> mm_create: 创建新的地址空间
          |    |--> setup_pgdir: 分配新的页目录
          |    |--> dup_mmap: 复制地址空间映射
          |           |--> copy_range: 逐页复制页面内容，并建立页表映射
    |--> copy_thread: 设置子进程的上下文
    |--> wakeup_proc: 使子进程可调度
```

由上面的流程图可以看出，`copy_range`的调用过程是：`do_fork()` --> `copy_mm()` -->`dup_mmap()` -->`copy_range()`

在lab4中并没有实现`copy_mm`，而在LAB 5中，`copy_mm` d的定义为：

```
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    struct mm_struct *mm, *oldmm = current->mm;

    /* current is a kernel thread */
    if (oldmm == NULL) {
        return 0;
    }
    if (clone_flags & CLONE_VM) {
        mm = oldmm;
        goto good_mm;
    }
    int ret = -E_NO_MEM;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
```

```
    }
    lock_mm(oldmm);
    {
        ret = dup_mmap(mm, oldmm);
    }
    unlock_mm(oldmm);

    if (ret != 0) {
        goto bad_dup_cleanup_mmap;
    }

good_mm:
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
    return 0;
bad_dup_cleanup_mmap:
    exit_mmap(mm);
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    return ret;
}
```

在上述代码中，`copy_mm` 使用互斥锁（防止并发修改），用了`dup_mmap`：

```
lock_mm(oldmm);
{
    ret = dup_mmap(mm, oldmm);
}
unlock_mm(oldmm);
```

而`dup_mmap` 又进一步调用了`copy_range` 。而在`copy_range` 中实现了一个 **页表复制** 功能：将一段地址空间（页）从源进程的页表 `from` 复制到目标进程的页表 `to`。

首先调用 `get_pte` 获取源进程页表 `from` 中 `start` 地址的页表项，并检查其有效性；接着在目标进程的页表 `to` 中获取或新建 `start` 地址的页表项，并为页分配内存。最后，确保源页和新页都成功分配后，准备进行复制操作，也就是我们需要完成的代码，一共分为四步：

- 找到 `src_kvaddr`：页面的内核虚拟地址。将`page` 通过宏`page2kva` 转换为源虚拟内存地址。
- 找到 `dst_kvaddr`：`npage` 的内核虚拟地址。和第一步类似，`page2kva` 被用来获取目标页面 `npage` 的内核虚拟地址。
- 从 `src_kvaddr` 到 `dst_kvaddr` 进行内存拷贝，大小为 `PGSIZE`。通过`memcpy`将虚拟地址进行复制。
- 使用线性地址 `start` 构建 `npage` 的物理地址映射。通过 `page_insert` 将物理页面 `npage` 映射到目标进程的线性地址 `start`。

具体的实现代码见下：

```
void *src_kvaddr = page2kva(page);   // 1. 源页面的内核虚拟地址
void *dst_kvaddr = page2kva(npage);  // 2. 目标页面的内核虚拟地址
// 3. 复制页面内容
memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
// 4. 在目标页表中建立映射
ret = page_insert(to, npage, start, perm);
```

2.2问题2：

概要设计：

Copy-on-Write 是一种高效的资源共享机制，设计的目标是延迟内存页的实际复制，提高资源利用率，直到写操作发生时再进行复制。以下是关键设计点：

1. **共享资源**：父子进程共享相同的内存页，避免不必要的复制。
2. **写操作触发复制**：当进程尝试修改共享的页时，操作系统会拦截写操作，分配新的内存页并复制原数据。
3. **权限管理**：通过页表的权限标志，区分只读共享页和可写独占页。
4. **内存分配优化**：仅在写操作时分配新的物理页，减少内存开销。
5. **异常处理**：在写入共享页时触发缺页异常，通过缺页处理程序完成复制。

详细设计：

1. 页表初始化

在进程创建时（如 `fork` 操作），只复制父进程的内存描述符和页表结构，但不复制实际的物理页：

- 页表项指向父进程的物理页。
- 清除页表项中的写权限（`PTE_W` 标志），设置为只读。

2. 写操作的异常捕获

当进程尝试写入共享页时：

- 硬件检测到写权限不足，触发缺页异常。
- 操作系统的异常处理函数（如 `cow_pgfault`）被调用。

3. 写时复制流程

在异常处理函数中完成以下操作：

1. **检查异常原因**：判断缺页是否由写操作引起，以及是否需要 COW 处理。
2. **分配新页**：为当前进程分配一个新的物理页。
3. **数据复制**：将原页内容复制到新页。
4. **更新页表**：将页表项更新为指向新页，并设置写权限。
5. **同步 TLB**：刷新 TLB 确保新的权限生效。

4. 数据结构

- **页表项 (`pte_t`)**：记录物理页地址和权限标志，COW 实现中需特别关注 `PTE_W` 和 `PTE_V` 标志。
- **进程内存管理结构 (`mm_struct`)**：包含页表地址和虚拟内存区域信息。
- **虚拟内存区域 (`vma_struct`)**：描述进程虚拟内存的起始地址、结束地址及权限。

5. 内存管理优化

- **引用计数**：每个物理页维护引用计数，记录共享进程数。当引用计数减少至 0 时释放物理页。
- **延迟复制**：只有在实际发生写操作时才分配新页并复制，减少开销。

6. 实现细节

1. `setup_pgdir`：初始化页表，复制父进程的页表结构。
2. `cow_copy_mm`：在进程创建时复制内存管理结构，但不实际复制物理页。
3. `cow_pgfault`：在写时处理缺页异常，完成物理页分配、数据复制和页表更新。

7. 系统退出时的资源释放

- 在进程结束时，遍历其页表，释放独占页。
- 对于共享页，减少引用计数，当引用计数归零时释放物理页。

COW 机制的优势：

- **高效性**：减少不必要的内存分配与复制，提高性能。
- **灵活性**：在资源需求变化时动态调整。
- **隔离性**：保证进程间的数据独立性，写操作对其他进程不可见。

练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：

- > 请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？

```
// user/libs/ulib.c
int fork(void) { return sys_fork(); }
int wait(void) { return sys_wait(0, NULL); }
int waitpid(int pid, int *store) { return sys_wait(pid, store); }
void yield(void) { sys_yield();}
int kill(int pid) { return sys_kill(pid); }
int getpid(void) { return sys_getpid(); }
```

```
// kern/trap/trap.c
//这里把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数
static int sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}
static int sys_fork(uint64_t arg[]) {
    struct trapframe *tf = current->tf;
    uintptr_t stack = tf->gpr.sp;
    return do_fork(0, stack, tf);
}
static int sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
```

```
    return do_wait(pid, store);
}
static int sys_exec(uint64_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    //用户态调用的exec(), 归根结底是do_execve()
    return do_execve(name, len, binary, size);
}
```

1.fork：创建子进程

具体实现：fork —> SYS_fork—>sys_fork—> do_fork

当程序执行fork时，会使用系统调用SYS_fork ，然后转发给do_fork 函数：

把当前的进程复制一份，创建一个子进程，原先的进程是父进程。接下来子进程和父进程都会收到`sys_fork()`的返回值，如果返回0说明当前位于子进程中，返回一个非0的值（子进程的PID）说明当前位于父进程中。可以根据返回值的不同，在两个进程里进行不同的处理。

具体的流程为：

1. 分配并初始化进程控制块（`alloc_proc`函数）
2. 分配并初始化内核栈（`setup_stack`函数）
3. 根据`clone_flags`决定是复制还是共享内存管理系统（`copy_mm`函数）
4. 设置进程的中断帧和上下文（`copy_thread`函数）
5. 把设置好的进程加入链表
6. 激活子进程（`wakeup_proc`函数）
7. 将返回值设为线程id

2.exec：执行新进程

exec —> SYS_exec —> sys_exec—> do_execve：

1. 首先会检查用户程序的名称有效性，使用user_mem_check函数检查传入的`name`（程序名）是否可以被当前用户进程访问。
2. 会限制程序名称的长度
3. 存储程序名称，使用`memcpy`将`name`拷贝到`local_name`中。
4. 清理当前进程的内存管理结构：如果当前进程有内存管理结构（`mm != NULL`），则先切换到内核的页目录：`lcr3(boot_cr3)`；减少当前进程的内存引用计数，如果计数为 0，则表示没有其他进程使用该内存资源；退出当前进程的内存映射，释放虚拟内存的映射关系；释放进程的页表；销毁当前进程的内存管理结构`mm`；将当前进程的`mm`设置为`NULL`，表明当前进程不再持有内存管理结构。
5. 加载新程序的二进制数据：这部分主要是调用`load_icode`函数。
6. 设置进程名称
7. 如果`load_icode`加载新程序失败，则会进入`execve_exit`，执行错误处理：调用`do_exit`退出当前进程，传递错误代码`ret`。

3.wait：等待进程

wai—> SYS_wait —>sys_wait —> do_wait：目的是等待子进程退出，并清理子进程的资源。

1. 检查进程的状态，确保能合法地等待某个子进程。
2. 如果子进程尚未退出（状态不是 `PROC_ZOMBIE`），则当前进程进入睡眠状态，等待子进程退出。
3. 找到一个子进程退出后，清理该子进程的内存和进程资源。
4. 返回成功或者错误码，表示等待过程的结果。

4.exit：退出进程

exit—> SYS_exit—> sys_exit—> do_exit：退出当前的进程。

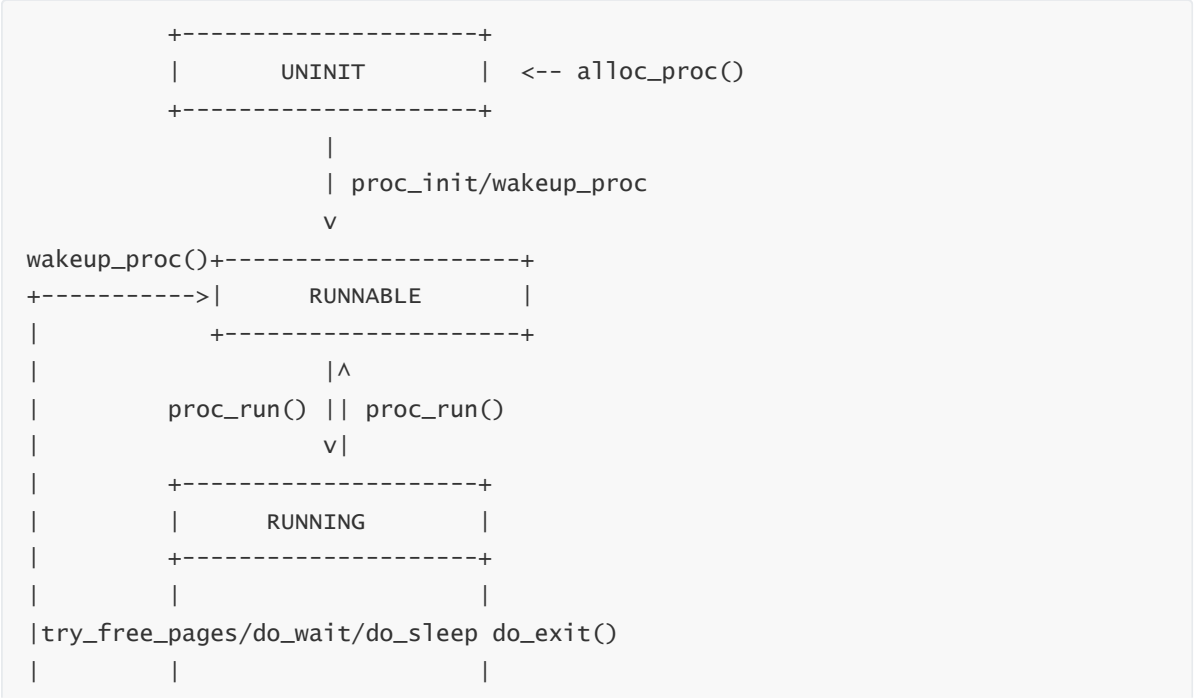
1. 检查进程是否是 `idleproc` 或 `initproc`，避免它们退出。
2. 释放当前进程的内存资源，包括内存映射、页表和内存管理结构。
3. 设置进程为 `PROC_ZOMBIE` 状态，并保存退出码。
4. 唤醒父进程，让它能够回收子进程。
5. 将当前进程的所有子进程的父进程设置为 `initproc`，并转移它们到 `initproc` 的子进程链表。
6. 切换到其他进程，完成进程退出。

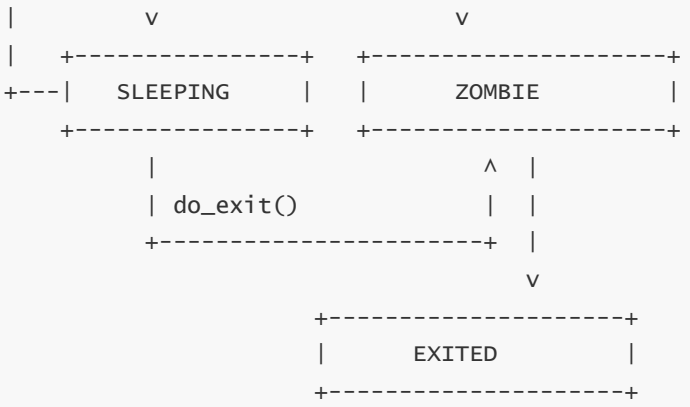
我们知道：用户程序在用户态运行(U mode), 系统调用在内核态执行(S mode)。

通过前面给出的fork/exec/wait/exit函数及其后续调用的函数的代码，知道在用户态的时候调用fork、exec、wait、exit函数（此时用户态），然后执行sys_fork等函数时，把系统调用进一步转发给proc.c的do_exit(), do_fork()等函数（内核态），内核执行相应的操作，然后再切换回用户态，将执行结果返回给用户程序。

> - 请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

3.2问题2：





扩展练习

COW实现

代码

创建失败时执行，这两项与`proc.c`中相同：

```
static int
setup_pgdir(struct mm_struct *mm) {
    struct Page *page;
    if ((page = alloc_page()) == NULL) {
        return -E_NO_MEM;
    }
    pde_t *pgdir = page2kva(page);
    memcpy(pgdir, boot_pgdir, PGSIZE);

    mm->pgdir = pgdir;
    return 0;
}

static void
put_pgdir(struct mm_struct *mm) {
    free_page(kva2page(mm->pgdir));
}
```

`do_pgfault`中添加判断页表项权限：

```
// 判断页表项权限，如果有效但是不可写，跳转到COW
if ((ptep = get_pte(mm->pgdir, addr, 0)) != NULL) {
    if((*ptep & PTE_V) & ~(*ptep & PTE_W)) {
        return cow_pgfault(mm, error_code, addr);
    }
}
```

将`do_fork`函数中的`copy_mm`改为`cow_copy_mm`：

```
// if(copy_mm(clone_flags, proc) != 0) {
//     goto bad_fork_cleanup_kstack;
// }
if(cow_copy_mm(proc) != 0) {
    goto bad_fork_cleanup_kstack;
}
```

复制虚拟内存空间：

```
int
cow_copy_mm(struct proc_struct *proc) {
    struct mm_struct *mm, *oldmm = current->mm;

    /* current is a kernel thread */
    if (oldmm == NULL) {
        return 0;
    }
    int ret = 0;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    lock_mm(oldmm);
    {
        ret = cow_copy_mmap(mm, oldmm);
    }
    unlock_mm(oldmm);

    if (ret != 0) {
        goto bad_dup_cleanup_mmap;
    }

good_mm:
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
    return 0;
bad_dup_cleanup_mmap:
    exit_mmap(mm);
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    return ret;
}
```

只复制`mm`与`vma`，将页表项均指向原来的页：

```
int
cow_copy_mmap(struct mm_struct *to, struct mm_struct *from) {
    assert(to != NULL && from != NULL);
    list_entry_t *list = &(from->mmap_list), *le = list;
```

```
    while ((le = list_prev(le)) != list) {
        struct vma_struct *vma, *nvma;
        vma = le2vma(le, list_link);
        nvma = vma_create(vma->vm_start, vma->vm_end, vma->vm_flags);
        if (nvma == NULL) {
            return -E_NO_MEM;
        }
        insert_vma_struct(to, nvma);
        if (cow_copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end) !=
0) {
            return -E_NO_MEM;
        }
    }
    return 0;
}
```

设置页表项指向：

```
int cow_copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    do {
        pte_t *ptep = get_pte(from, start, 0);
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            *ptep &= ~PTE_W;
            uint32_t perm = (*ptep & PTE_USER & ~PTE_W);
            struct Page *page = pte2page(*ptep);
            assert(page != NULL);
            int ret = 0;
            ret = page_insert(to, page, start, perm);
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

实现COW的缺页异常处理：

```
int
cow_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    int ret = 0;
    pte_t *ptep = NULL;
    ptep = get_pte(mm->pgdir, addr, 0);
    uint32_t perm = (*ptep & PTE_USER) | PTE_W;
    struct Page *page = pte2page(*ptep);
    struct Page *npage = alloc_page();
    assert(page != NULL);
    assert(npage != NULL);
    uintptr_t* src = page2kva(page);
```

```
    uintptr_t* dst = page2kva(npage);
    memcpy(dst, src, PGSIZE);
    uintptr_t start = ROUNDDOWN(addr, PGSIZE);
    *ptep = 0;
    ret = page_insert(mm->pgdir, npage, start, perm);
    ptep = get_pte(mm->pgdir, addr, 0);
    return ret;
}
```

至此，`make qemu`与`make grade`均能正常运行并通过。

错误复现

在`user/exit.c`文件中添加

```
uintptr_t* p = 0x800588;
cprintf("*p = 0x%x\n", *p);
*p = 0x222;
cprintf("*p = 0x%x\n", *p);
```

如果使用原本的策略，执行`make qemu`会提示：

```
*p = 0x5171101
Store/AMO page fault
kernel panic at kern/fs/swapfs.c:20:
    invalid swap_entry_t = 2013281b.
```

但是如果使用COW策略，则会正常运行：

```
*p = 0x5171101
Store/AMO page fault
COW page fault at 0x800000
Store/AMO page fault
COW page fault at 0x7ffff000
*p = 0x222
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:479:
    initproc exit.
```

这表明我们在`qemu`模拟出的磁盘上的`0x800588`处写入了数据`0x222`。

用户程序加载

该用户程序在操作系统加载时一起加载到内存里。我们平时使用的程序在操作系统启动时还位于磁盘中，只有当我们需要运行该程序时才会被加载到内存里。

原因是在`Makefile`里执行了`ld`命令，把执行程序的代码连接在了内核代码的末尾。