OSlab5

组员：

2110729 张弛
2111451 朱景博
2111566 米奕霖

练习0: 经典合并代码

事实上之前的实验也基本都涉及到了合并代码，但之所以没有在报告里提及，是因为之前只是单纯的 copy 即可，但此处需对之前填充的部分代码进行更新（当然也十分简单...），具体涉及到如下代码块：

```
proc.c/alloc_proc
proc.c/proc_run
proc.c/do_fork
vmm.c/do_pgfault
```

下面仅对需要更新的代码部分进行说明：

- `alloc_proc`函数

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    proc->state = PROC_UNINIT;
    proc->pid = -1;
    proc->runs = 0;
    proc->kstack = 0;
    proc->need_resched = 0;
    proc->parent = NULL;
    proc->mm = NULL;
    memset(&(proc->context), 0, sizeof(struct context));
    proc->tf = NULL;
    proc->cr3 = boot_cr3;
    proc->flags = 0;
    memset(proc->name, 0, PROC_NAME_LEN);

    proc->wait_state = 0; //PCB新增的条目，初始化进程等待状态
    proc->cptr = proc->yptr = proc->optr = NULL; //设置指针
    }
    return proc;
}
```

新增初始化字段的具体含义阐述如下：

- `proc->wait_state = 0`：

  此处的`proc->wait_state`用于表示进程的等待状态。等待状态是一个进程可能处于的状态，例如等待某个事件发生或等待某个资源可用。初始化为 0 可以表示初始时进程处于没有等待的状态。

- `proc->cptr = proc->yptr = proc->optr = NULL`：

  此处的 `proc->cptr`、`proc->yptr` 和 `proc->optr` 是指向其他 `struct proc_struct` 结构体的指针。它们通常用于建立进程之间的关系，比如父子关系、兄弟关系等。将它们初始化为 NULL 表示在初始时，新分配的进程还没有建立与其他进程的关联

  - `proc->cptr（Child Pointer）`：指向当前进程的子进程。
  - `proc->yptr（Younger Sibling Pointer）`：指向当前进程的下一个兄弟进程。
  - `proc->optr（Older Sibling Pointer）`：指向当前进程的前一个兄弟进程。

- `do_fork` 函数

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    ...
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    assert(current->wait_state == 0); //确保父进程不处于等待状态

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        set_links(proc); //设置进程链接
        hash_proc(proc);
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);
    ret = proc->pid;
    ...
}
```

  - `assert(current->wait_state == 0);`：
    此处使用了 `assert` 宏，它的作用是确保当前进程处于没有等待的状态，具体而言就是确保在执行 `fork` 时，父进程不处于等待状态。
  - `set_links(proc);`：
    此处对 `set_links` 的调用，是用于设置新创建的进程的链接关系。下面是对该函数的主要解释：

```
static void set_links(struct proc_struct *proc) {
    list_add(&proc_list, &(proc->list_link));
    proc->yptr = NULL;
    if ((proc->optr = proc->parent->cptr) != NULL) {
        proc->optr->yptr = proc;
    }
    proc->parent->cptr = proc;
    nr_process++;
}
```

- `list_add(&proc_list, &(proc->list_link));`：将新创建的进程 `proc` 加入到一个全局的进程列表 `proc_list` 中。这个列表可能用于系统中对所有进程的管理。
- `proc->yptr = NULL;`：将新进程的 `yptr` 设置为 `NULL`，表示当前进程在兄弟链表中没有下一个兄弟。
- `if ((proc->optr = proc->parent->cptr) != NULL) { proc->optr->yptr = proc; }`：设置新进程的 `optr` 指向当前进程的子进程链表的头部，如果有的话，并将原来的兄弟的 `yptr` 指向新进程。这样就建立了当前进程和新进程之间的兄弟关系。
- `proc->parent->cptr = proc;`：将新进程设置为当前进程的子进程。
- `nr_process++;`：增加系统中进程的数量计数。

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。请在实验报告中简要说明你的设计实现过程。

首先 `load_icode` 函数本身的逻辑是比较繁琐的，但此处需要填充的仅是其中关于设置 `trapframe` 的语句，再配合保姆级的注释提示，填起来还是十分轻松的。

```
static int
load_icode(unsigned char *binary, size_t size) {
    ...
    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    // Set user stack pointer (sp)
    tf->gpr.sp = USTACKTOP;
    // Set entry point of user program (epc)
    tf->epc = elf->e_entry;
    // Set status register
    tf->status = (read_csr(sstatus) | SSTATUS_SPIE ) & ~SSTATUS_SPP;
    ...
}
```

具体对`load_icode`函数及填充部分的解析如下：

1. **检查当前进程的内存管理结构（`current->mm`）：**

```
if (current->mm != NULL) {
    panic("load_icode: current->mm must be empty.\n");
}
```

检查当前进程的内存管理结构是否为空。如果不为空，说明当前进程的内存已经被分配，可能是一个错误状态，因此触发内核 panic。

2. **创建新的内存管理结构 (`mm`):**

```
if ((mm = mm_create()) == NULL) {
    goto bad_mm;
}
```

创建一个新的内存管理结构 `mm`，并将其设置为当前进程的 `mm`。

3. **设置页表目录 (`pgdir`):**

```
if (setup_pgdir(mm) != 0) {
    goto bad_pgdir_cleanup_mm;
}
```

为新的内存管理结构创建页表目录，并将其设置为内存管理结构的 `pgdir`。

4. **复制 TEXT/DATA 段到内存空间:**

```
// 获取 ELF 文件头和程序头表
struct elfhdr *elf = (struct elfhdr *)binary;
struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff);

// 遍历程序头表，处理每个可加载段
for (; ph < ph_end; ph ++) {
    if (ph->p_type != ELF_PT_LOAD) {
        continue;
    }
    // 处理每个可加载段，包括 TEXT/DATA 段和 BSS 段
    // 调用 mm_map 函数将段映射到进程的地址空间
    if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
        goto bad_cleanup_mmap;
    }
    // 处理 TEXT/DATA 段，将内容从 ELF 文件复制到进程内存
    // 处理 BSS 段，分配内存并清零
    // ...
}
```

5. **设置用户栈:**

```
vm_flags = VM_READ | VM_WRITE | VM_STACK;
if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {
    goto bad_cleanup_mmap;
}
// 分配用户栈的页面并设置权限
// ...
```

6. **设置当前进程的内存管理结构等信息：**

```
mm_count_inc(mm);
current->mm = mm;
current->cr3 = PADDR(mm->pgdir);
lcr3(PADDR(mm->pgdir));
```

7. **设置用户程序的入口地址、用户栈指针和状态寄存器等信息：**

```
struct trapframe *tf = current->tf;
tf->gpr.sp = USTACKTOP;
tf->epc = elf->e_entry;
tf->status = (read_csr(sstatus) | SSTATUS_SPIE ) & ~SSTATUS_SPP;
```

   - `tf->gpr.sp`：设置用户栈指针为用户栈的顶部。
   - `tf->epc`：设置用户程序的入口地址。
   - `tf->status`：设置状态寄存器，确保用户程序在用户态运行，并开启中断。
     - 将 `SSTATUS_SPIE` 位设置为 1，表示在异常返回时重新启用中断。
     - 将 `SSTATUS_SPP` 位清零，表示特权级别返回到用户态。

总之，`load_icode` 函数的作用是加载用户程序的代码和数据段到当前进程的内存空间，并设置好用户栈和进程控制块等信息，以便将控制权转移到用户程序。特别是在设置用户程序入口地址、用户栈指针和状态寄存器时，确保用户程序能够正确运行在用户态。

> 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。

用户态进程被 ucore 选择占用 CPU 执行到具体执行应用程序第一条指令的过程，可概括如下：

- 使用 `mm_create` 来申请一个新的mm并初始化
- 使用 `setup_pgdir` 来申请一个页目录表所需的一个页大小，并且把 ucore 内核的虚拟空间所映射的内核页表 `boot_pgdir` 拷贝过来，然后 `mm->pgdir` 指向这个新的页目录表
- 根据程序的起始位置来解析此程序，使用`mm_map`为可执行程序的代码段，数据段，BSS 段等建立对应的 vma 结构，插入到 mm 中，把这些作为用户进程的合法的虚拟地址空间
- 根据各个段大小来分配物理内存，确定虚拟地址，在页表中建立起虚实的映射。然后把内容拷贝到内核虚拟地址中
- 为用户进程设置用户栈，建立用户栈的 vma 结构。并且要求用户栈在分配给用户虚空间的顶端，占据256个页，再为此分配物理内存和建立映射
- 将 `mm->pgdir` 赋值给 cr3 以更新用户进程的虚拟内存空间。
- 清空进程中断帧后，重新设置进程中断帧以使得在执行中断返回指令 `iret` 后让CPU 跳转到 Ring3，回到用户态内存空间，并跳到用户进程的第一条指令。

值得一提的是，在第六步的时候，`init` 已经被 `exit` 所覆盖，构成了第一个用户进程的雏形。在之后才建立这个用户进程的执行现场。

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。

此处同样按照注释提示直接填充即可，具体如下：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
               bool share) {
    ...
    /* LAB5:EXERCISE2 YOUR CODE
     * replicate content of page to npage, build the map of phy addr of
     * nage with the linear addr start
     *
     * Some Useful MACROs and DEFINEs, you can use them in below
     * implementation.
     * MACROs or Functions:
     *    page2kva(struct Page *page): return the kernel vritual addr of
     * memory which page managed (SEE pmm.h)
     *    page_insert: build the map of phy addr of an Page with the
     * linear addr la
     *    memcpy: typical memory copy function
     *
     * (1) find src_kvaddr: the kernel virtual address of page
     * (2) find dst_kvaddr: the kernel virtual address of npage
     * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
     * (4) build the map of phy addr of  nage with the linear addr start
     */
    void * kva_src = page2kva(page);
    void * kva_dst = page2kva(npage);
    memcpy(kva_dst, kva_src, PGSIZE);
    ret = page_insert(to, npage, start, perm);
    ...
}
```

填充部分的具体解析如下：

- `void * kva_src = page2kva(page);`:
  - 使用 `page2kva` 宏将源页面 `page` 的物理地址映射到内核的虚拟地址。
- `void * kva_dst = page2kva(npage);`:
  - 使用 `page2kva` 宏将目标页面 `npage` 的物理地址映射到内核的虚拟地址。得到目标页面在内核虚拟地址空间中的起始地址。
- `memcpy(kva_dst, kva_src, PGSIZE);`:
  - 使用 `memcpy` 函数将源页面的内容复制到目标页面。
- `ret = page_insert(to, npage, start, perm);`:
  - 使用 `page_insert` 函数建立目标地址空间 `to` 中目标页面 `npage` 与指定线性地址 `start` 之间的映射关系。
  - `start` 是当前复制的线性地址，`perm` 是权限标志，包括用户/内核态、读/写/执行等。

整体而言，填充代码块的作用是将源地址空间的一页内容复制到目标地址空间的一页，并在目标地址空间中建立相应的映射关系。这通常用于实现进程间的共享或复制内存。

> 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。

此处仅基于 *Copy on Write* 机制给出相应的概要设计：

- **页面管理：**
  - 每个页面都附加一个引用计数（reference count）。引用计数跟踪有多少个进程共享相同的物理页面。
  - 在页表项中，增加一个标志位，用于表示该页面是否是只读的。
- **写时复制操作：**
  - 当一个进程试图写入一个只读页面时，操作系统会进行写时复制。
  - 操作系统会检查引用计数，如果引用计数为1（表示只有一个进程拥有该页面），那么直接将页面标记为可写。如果引用计数大于1，那么进行页面复制。
  - 复制操作涉及到为新的页面分配物理内存，并将原始页面的内容复制到新页面。
  - 新页面的引用计数设为1，原始页面的引用计数减1。
- **页表更新：**
  - 更新原始页面的页表项，将其指向新的物理页面。
  - 更新新页面的页表项，将其标记为可写。
- **引用计数维护：**
  - 当一个进程终止或释放对页面的引用时，引用计数减1。
  - 当引用计数减为0时，释放相应的物理内存。

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。

截止到本次实验，ucore 中涉及的系统调用及其大致函数调用展示如下：

```
SYS_exit        : process exit,                           -->do_exit
SYS_fork        : create child process, dup mm            -->do_fork-->wakeup_proc
SYS_wait        : wait process                            -->do_wait
SYS_exec        : after fork, process execute a program   -->load a program and refresh the mm
SYS_clone       : create child thread                     -->do_fork-->wakeup_proc
SYS_yield       : process flag itself need resecheduling, -->proc->need_sched=1, then scheduler will rescheule this process
SYS_sleep       : process sleep                           -->do_sleep
SYS_kill        : kill process                            -->do_kill-->proc->flags |= PF_EXITING
                                                             -->wakeup_proc-->do_wait-->do_exit
SYS_getpid      : get the process's pid
```

一般来说，用户进程只能执行一般的指令，无法执行特权指令。采用系统调用机制为用户进程提供一个获得操作系统服务的统一接口层，简化用户进程的实现。

根据之前的分析，应用程序调用的 `exit/fork/wait/getpid` 等库函数最终都会调用 `syscall` 函数，区别仅在于传入的参数不同（分别是 `SYS_exit / SYS_fork / SYS_wait / SYS_getid` ）

当应用程序调用系统函数时，一般执行 `INT T_SYSTEMCALL` 指令后，CPU 根据操作系统建立的系统调用中断描述符，转入内核态，然后开始了操作系统系统调用的执行过程，在内核函数执行之前，会保留软件执行系统调用前的执行现场，然后保存当前进程的 `tf` 结构体中，之后操作系统就可以开始完成具体的系统调用服务，完成服务后，调用 `IRET` 返回用户态，并恢复现场。这样整个系统调用就执行完毕了。

fork

涉及到的函数调用过程：

```
fork->SYS_fork->do_fork + wakeup_proc
```

其中 `wakeup_proc` 函数主要是将进程的状态设置为等待。

`do_fork` 函数的执行流程如下：

```
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_flag 标志复制或共享进程内存管理结构(copy_mm 函数);
4、设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy_thread 函数);
5、把设置好的进程控制块放入hash_list 和 proc_list 两个全局进程链表中;
6、自此,进程已经准备好执行了,把进程状态设置为“就绪”态;
7、设置返回码为子进程的 id 号。
```

exec

涉及到的函数调用过程：

```
SYS_exec->do_execve
```

`do_execve` 函数的执行流程如下：

```
1、首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。
2、接下来是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。之后就是调用load_icode从而使之准备好执行。
```

wait

涉及到的函数调用过程：

```
SYS_wait->do_wait
```

`do_wait` 函数的执行流程如下：

1、如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程；
2、如果此子进程的执行状态不为 PROC_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC_SLEEPING（睡眠），睡眠原因为WT_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行；
3、如果此子进程的执行状态为 PROC_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc_list和hash_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。

exit

涉及到的函数调用过程：

```
SYS_exit->exit
```

`exit` 函数的执行流程如下：

1、先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间；（具体的回收过程不作详细说明）
2、设置当前进程的中hi性状态为PROC_ZOMBIE，然后设置当前进程的退出码为error_code。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块）
3、如果当前父进程已经处于等待子进程的状态，即父进程的wait_state被置为WT_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。
4、如果当前进程还有子进程,则需要把这些子进程的父进程指针设置为内核线程init,且各个子进程指针需要插入到init的子进程链表中。如果某个子进程的执行状态是 PROC_ZOMBIE,则需要唤醒 init来完成对此子进程的最后回收工作。
5、执行schedule()调度函数，选择新的进程执行。

> 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？

执行流程见上述分析

内核态与用户态交错执行

在 `fork/exec/wait/exit` 等系统调用的过程中，用户态进程调用了系统调用，触发了从用户态到内核态的转换。
内核态执行相应的系统调用服务，完成相应的工作，包括进程的创建、加载新程序、等待子进程退出、进程资源的清理等。
在系统调用完成后，会再次切换回用户态，继续执行用户程序。

内核态执行结果返回给用户程序

在系统调用的执行过程中，内核通过相关的数据结构（如进程控制块、中断帧等）保存了用户程序的执行现场。
执行完系统调用后，通过中断返回（IRET）将控制权重新交还给用户程序，并在用户程序中恢复之前保存的执行现场，使用户程序继续执行。

> 请给出 ucore 中一个用户态进程的执行状态生命周期图（包括执行状态，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）

```
  (alloc_proc)          (wakeup_proc)
    ---------------> NEW ----------------> READY
                                             |
                                             |
                                             | (proc_run)
                                             |
         (do_wait)             (do_exit)     V
   EXIT <---------- ZOMBIE <-------------- RUNNING
```

效果验证

完成上述各代码模块的补充后，即可进行`make qemu`验证了，验证结果如下：

```
Load page fault
page fault at 0x00001000: K/R
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "forktest".
Breakpoint
I am child 31
I am child 30
I am child 29
I am child 28
I am child 27
I am child 26
I am child 25
I am child 24
I am child 23
I am child 22
I am child 21
I am child 20
I am child 19
I am child 18
I am child 17
I am child 16
I am child 15
I am child 14
I am child 13
I am child 12
I am child 11
I am child 10
I am child 9
I am child 8
I am child 7
I am child 6
I am child 5
I am child 4
I am child 3
I am child 2
I am child 1
I am child 0
forktest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:478:
    initproc exit.
```

之后再进行`make grade`即可完成验证：

```
badsegment:              (1.1s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (1.0s)
  -check result:                             OK
  -check output:                             OK
softint:                 (1.1s)
  -check result:                             OK
  -check output:                             OK
faultread:               (1.0s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:         (1.0s)
  -check result:                             OK
  -check output:                             OK
hello:                   (1.0s)
  -check result:                             OK
  -check output:                             OK
testbss:                 (1.1s)
  -check result:                             OK
  -check output:                             OK
pgdir:                   (1.1s)
  -check result:                             OK
  -check output:                             OK
yield:                   (1.0s)
  -check result:                             OK
  -check output:                             OK
badarg:                  (1.0s)
  -check result:                             OK
  -check output:                             OK
exit:                    (1.0s)
  -check result:                             OK
  -check output:                             OK
spin:                    (4.2s)
  -check result:                             OK
  -check output:                             OK
forktest:                (1.1s)
  -check result:                             OK
  -check output:                             OK
Total Score: 130/130
```

扩展练习 Challenge 2

> 说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

在本次实验中，用户程序在编译时被链接到内核中，并定义好了起始位置和大小，然后在 `user_main()` 函数 `KERNEL_EXECVE` 宏调用 `kernel_execve()` 函数，从而调用 `load_icode()` 函数将用户程序加载到内存中。 而在我们常用的操作系统中，用户程序通常是存储在外部存储设备上的独立文件。当需要执行某个程序时，操作系统会从磁盘等存储介质上动态地加载这个程序到内存中。 原因是 ucore 没实现硬盘，出于简化和教学性质的考虑，将用户程序编译到内核中减少了实现的复杂度。