实验项目：用户进程管理

姓名：张伟焜　学号：17343155　邮箱：zhangwk8@mail2.sysu.edu.cn
院系：数据科学与计算机学院　专业：17 级软件工程　指导教师：张永东

【实验题目】

用户进程管理

【实验目的】

了解第一个用户进程创建过程

了解系统调用框架的实现机制

了解 ucore 如何实现系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来进行进程管理

【实验要求】

根据指导，完成练习 0~3。

【实验方案】

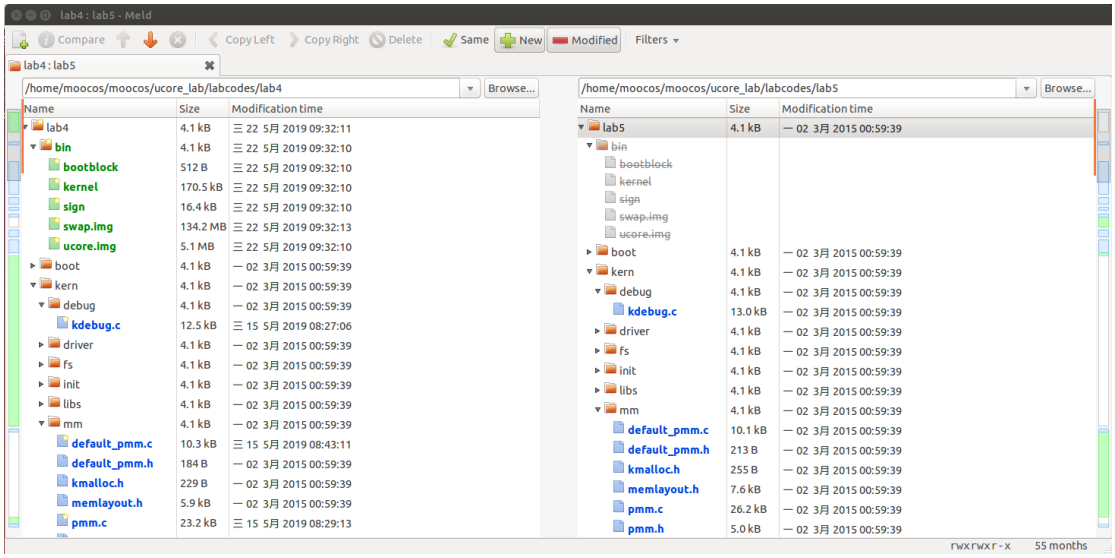
实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求

实验思路：根据实验指导，先了解理论知识，再进行实验

【实验过程】

练习 0：填写已有实验。

使用 meld 软件将 ucore 启动实验的代码导入。



注意要点击标星文件进行对比，将上次实验完成的函数复制过来，不要将整个文件进行覆盖。之前实验修改的内容主要在 kdebug.c trap.c pmm.c default_pmm.c vmm.c swap_fifo.c proc.c。

其中 proc.c 中有两处对之前代码的更新

1.对一些变量的初始化。初始化等待状态并设置指针

```
106      //LAB5 YOUR CODE : (update LAB4 steps)
107     /*
108      * below fields(add in LAB5) in proc_struct need to be initialized
109      *       uint32_t wait_state;                        // waiting state
110      *       struct proc_struct *cptr, *yptr, *optr;     // relations between processes
111          */
112      proc->state = PROC_UNINIT;
113      proc->pid = -1;
114      proc->runs = 0;
115      proc->kstack = 0;
116      proc->need_resched = 0;
117      proc->parent = NULL;
118      proc->mm = NULL;
119      memset(&(proc->context), 0, sizeof(struct context));
120      proc->tf = NULL;
121      proc->cr3 = boot_cr3;
122      proc->flags = 0;
123      proc->wait_state=0;
124      proc->cptr = NULL;
125      proc->yptr = NULL;
126      proc->optr = NULL;
127      memset(&(proc->name), 0, PROC_NAME_LEN);
128     }
129     return proc;
130 }
```

2.使用 set_links 函数来完成将 fork 的线程添加到线程链表中的过程。

由于该函数中就包括了对进程总数加 1 这一操作，因此需要将原先+1 操作给删除掉

```
assert(current->wait_state == 0); //保证子进程在等待态
set_links(proc); //设置进程链接
```

```
415         //LAB5 YOUR CODE : (update LAB4 steps)
416    /* Some Functions
417     *    set_links:  set the relation links of process.  ALSO SEE: remove_links:  lean the relation links
   of process
418     *    ------------------
419         *    update step 1: set child proc's parent to current process, make sure current process's
   wait_state is 0
420         *    update step 5: insert proc_struct into hash_list && proc_list, set the relation links of
   process
421     */
422     if(!(proc = alloc_proc()))
423         goto fork_out;
424     proc->parent = current;
425     assert(proc->wait_state==0);//保证子进程在等待态
426     if(setup_kstack(proc))
427         goto bad_fork_cleanup_kstack;
428     if(copy_mm(clone_flags,proc))
429         goto bad_fork_cleanup_proc;
430     copy_thread(proc,stack,tf);
431     proc->pid = get_pid();
432     hash_proc(proc);
433     set_links(proc);//设置进程链接
434     wakeup_proc(proc);
435     ret = proc->pid;
436 fork_out:
437     return ret;
```

trap.c 中也有两处代码更新

1.设置系统调用对应的中断描述符，使其能够在用户态下被调用，并且设置为 trap 类型。

```
SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);//设置相应的中断门
```

```
56      /* LAB5 YOUR CODE */
57      //you should update your lab1 code (just add ONE or TWO lines of code), let user app to use syscall
  to get the service of ucore
58      //so you should setup the syscall interrupt gate in here
59      extern uintptr_t __vectors[];
60      int i;
61      for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
62          SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
63      }
64      SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
65      lidt(&idt_pd);
66  }
```

2.每过 TICK_NUM 个中断，就将当前的进程设置为可以被重新调度的，这样使得当前的线程可以被换出，从而实现多个线程的并发执行。将时间片设置为需要调度，说明当前进程的时间片已经用完。

```
229         /* LAB5 YOUR CODE */
230         /* you should upate you lab1 code (just add ONE or TWO lines of code):
231          *    Every TICK_NUM cycle, you should set current process's current->need_resched = 1
232          */
233         ticks ++;
234         if (ticks % TICK_NUM == 0) {
235             assert(current != NULL);
236             current->need_resched = 1;
237         }
238         break;
```

练习 1：加载应用程序并执行。

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，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。

分析及设计过程：

本练习涉及的主要函数为 load_icode 函数，由 do_execve 函数调用。do_execve 函数是 exec 系统调用的最终处理的函数，它会将某一个指定 ELF 可执行二进制文件加载到当前内存中来，之后当前进程就执行该文件，而 load_icode 函数的功能是为执行新的程序初始化好内存空间。在调用该函数之前，do_execve 中已经退出了当前进程的内存空间，改为使用内核的内存空间，这样使得对原先用户态的内存空间的操作成为可能。

load_icode 函数中，我们需要完成的是伪造中断返回现场，使得系统调用返回之后可以正确跳转到需要运行的程序入口，并正常运行。

实验给出的注释如下：

```
625     /* LAB5:EXERCISE1 YOUR CODE
626      * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
627      * NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from
   kernel. So
628      *          tf_cs should be USER_CS segment (see memlayout.h)
629      *          tf_ds=tf_es=tf_ss should be USER_DS segment
630      *          tf_esp should be the top addr of user stack (USTACKTOP)
631      *          tf_eip should be the entry point of this binary program (elf->e_entry)
632      *          tf_eflags should be set to enable computer to produce Interrupt
633      */
```

初始化 tf 中的变量：

1.为了保证在用户态运行，将段寄存器初始化为用户态的代码段、数据段、堆栈段；

2.esp 指向先前步骤中创建的用户栈的栈顶；

3.eip 指向 ELF 可执行文件加载到内存之后的入口；

4.eflags 初始化为中断使能，其第 1 位恒为 1；

5.ret 为 0，表示正常返回；

结合分析与注释，得到代码：

```
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;
ret = 0;
```

总的来说，load_icode 函数的主要功能就是给用户进程建立一个能够让用户进程正常运行的用户环境。

描述当创建一个用户态进程并加载了应用程序后，CPU 是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被 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。

在经过调度器占用了 CPU 的资源之后，用户态进程调用了 exec 系统调用，进而转入系统调用的处理例程。

经过正常的中断处理例程之后，最终控制权给了 syscall.c 中的 syscall 函数，然后根据系统调用号转移给 sys_exec 函数，在该函数中调用了 do_execve 函数来完成指定应用程序的加载。

do_execve 首先检查用户态虚拟内存空间是否合法，如果合法且目前只有当前进程占用，则释放虚拟内存空间，包括取消虚拟内存到物理内存的映射，释放 vma，mm 及页目录表占用的物理页等。

换用 kernel 的 PDT 之后，使用 load_icode 函数，来完成对整个用户线程内存空间的初始化，包括堆栈的设置以及将 ELF 可执行文件的加载，之后通过 current->tf 指针修改了当前系统调用的 trapframe（保证最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处）。

do_exec 函数之后，进行正常的中断返回。由于中断处理例程栈上面的 eip 已经被修改为应用程序的入口处，而 cs 上的 CPL 是用户态，因此 iret 中断返回时会将堆栈切换到用户的栈，并完成特权级的切换，再跳转到要求的应用程序的入口处，接着具体执行应用程序第一条指令。

练习 2：父进程复制自己的内存空间给子进程。

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。

分析：

copy_range 函数执行过程为：

遍历父进程指定的某一段内存空间中的每一个虚拟页，如果这个虚拟页存在，为子进程对应的同一个地址申请分配一个物理页，然后将前者的所有内容复制到后者中去，然后为子进程的这个物理页和对应的虚拟地址建立映射关系；

实验给出的注释如下:

```
531         /* LAB5:EXERCISE2 YOUR CODE
532          * replicate content of page to npage, build the map of phy addr of nage with the linear addr start
533          *
534          * Some Useful MACROs and DEFINEs, you can use them in below implementation.
535          * MACROs or Functions:
536          *    page2kva(struct Page *page): return the kernel vritual addr of memory which page managed
    (SEE pmm.h)
537          *    page_insert: build the map of phy addr of an Page with the linear addr la
538          *    memcpy: typical memory copy function
539          *
540          * (1) find src_kvaddr: the kernel virtual address of page
541          * (2) find dst_kvaddr: the kernel virtual address of npage
542          * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
543          * (4) build the map of phy addr of  nage with the linear addr start
544          */
```

内存的复制和映射的建立，具体步骤为:

找到父进程指定的某一物理页对应的内核虚拟地址;

找到需要拷贝的子进程的对应物理页对应的内核虚拟地址;

将前者的内容拷贝到后者中去;

建立子进程的物理页与虚拟页的映射关系

结合分析与注释，得到代码:

```
char *src_kvaddr = page2kva(page); // 找到父进程需要复制的物理页在内核地址空间中的虚拟地址，这是由于这个函数执行的时候使用的时内核的地址空间
char *dst_kvaddr = page2kva(npage); // 找到子进程需要被填充的物理页的内核虚拟地址
memcpy(dst_kvaddr, src_kvaddr, PGSIZE); // 将父进程的物理页的内容复制到子进程中去
ret = page_insert(to, npage, start, perm);// 建立子进程的物理页与虚拟页的映射关系
assert(ret == 0);
```

请在实验报告中简要说明如何设计实现"Copy on Write 机制"，给出概要设计，鼓励给出详细设计。

概要设计:

"Copy on Write"机制是进程 fork 进行复制的时候，父进程不是直接将整个内存中的内容复制给子进程，而是子进程和父进程暂时共享相同的物理内存页;当其中一个进程需要对内存进行修改的时候，额外创建一个私有的物理内存页，将共享的内容复制过去后，在自己的内存页中进行修改。

根据上述分析，需要两部分。一个部分使进行 fork 操作的时候不直接复制内存，另一部分处理出现内存页访问异常时，将共享的内存页复制一份，然后在新的内存页进行修改。

详细设计:

do fork 部分: 在 copy_range 函数内部，不实际进行内存的复制，将子进程和父进程的虚拟页映射上同一个物理页面。之后分别将这个页设为不可写，利用 PTE 中的保留位将这个页设置成共享的页面。

page fault 部分: 在 ISR 部分，增加判断是否由于尝试写某个共享页面引起异常。如果是，额外申请分配一个物理页面，然后将当前的共享页的内容复制过去，建立出错的线性地址与新创建的物理页面的映射关系，将 PTE 设置设置成非共享的;然后查询原先共享的物理页面是否还是由多个其他进程共享使用的，如果不是的话，就将对应的虚地址的 PTE 进行修改，删掉共享标记，恢复写标记;这样 page fault 返回之后就可以正常完成对虚拟内存的写操作。

练习 3：分析代码: fork/exec/wait/exit 函数，以及系统调用的实现。

请分析 fork/exec/wait/exit 在实现中是如何影响进程的执行状态的?

fork:

创建一个新进程所需的控制信息。do_fork 函数, 完成新的进程的进程控制块的初始化、设置、以及将父进程内存中的内容到子进程的内存的复制工作，然后将新创建的进程放入可执行队列（runnable）。

fork 不会影响当前进程的执行状态，但是会将子进程的状态标记为 RUNNALB，使得可以在后续的调度中运行起来。

exec:

exec 的功能是在已经存在的进程的上下文中运行新的可执行文件，替换先前的可执行文件。执行 do_execve 函数, 对内存空间进行清空, 然后将新的要执行的程序加载到内存中，并设置好中断帧，使得最终中断返回之后可以跳转到指定的应用程序的入口处。

exec 不会影响当前进程的执行状态，但是会修改当前进程中执行的程序。

wait:

wait 的功能是等待子进程结束，从而释放子进程占用的资源。do_wait 函数如果找到子进程，但状态不为 ZOMBIE，则将当前进程的 state 设置为 SLEEPING、wait_state 设置为 WT_CHILD，然后调用 schedule 函数，从而进入等待状态。等再次被唤醒后，重复寻找状态为 ZOMBIE 的子进程。

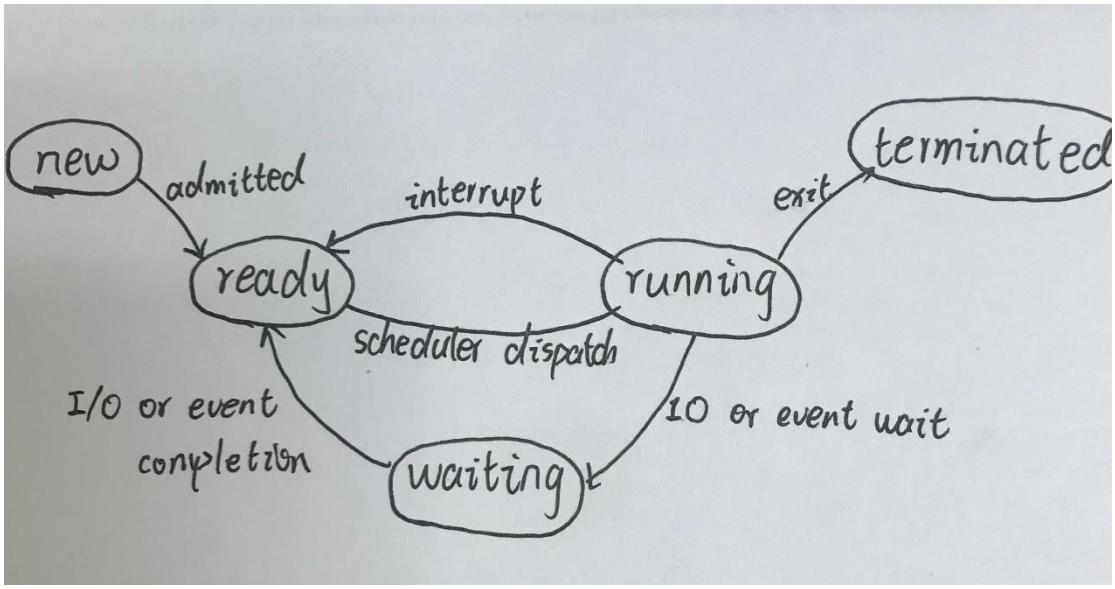
wait 系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为 SLEEPING 态，等待执行了 exit 的子进程将其唤醒

exit:

exit 的功能是释放进程占用的资源并结束运行进程。释放页表项记录的物理内存，以及 mm 结构、vma 结构、页目录表占用的内存, 将其标记为 ZOMBIE 态, 然后调用 wakeup_proc 函数将其父进程唤醒（如果父进程执行了 wait 进入 SLEEPING 态的话），然后调用 schedule 函数，让出 CPU 资源，等待父进程进一步完成其所有资源的回收。

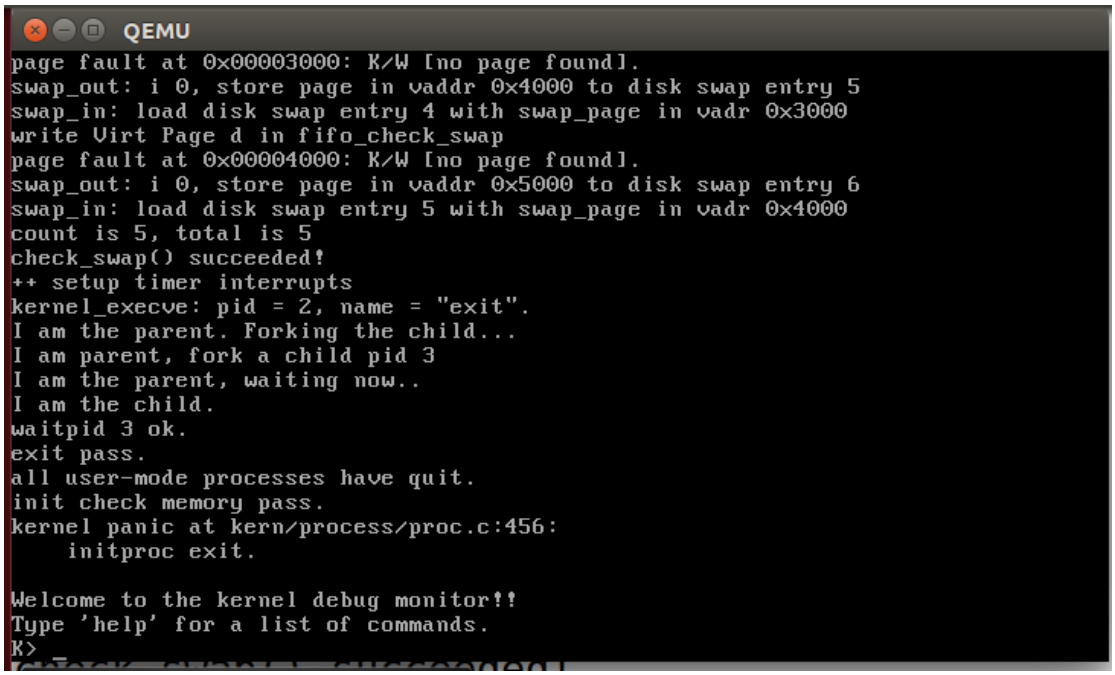
exit 会将当前进程的状态修改为 ZOMBIE 态, 并且会将父进程唤醒(修改为 RUNNABLE)，然后主动让出 CPU 使用权。

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）



运行结果：

make qemu:



make grade:

```
zhangweikun$>cd moocos
zhangweikun$>cd ucore_lab
zhangweikun$>cd labcodes
zhangweikun$>cd lab5
zhangweikun$>make grade
badsegment:              (2.8s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (1.6s)
  -check result:                             OK
  -check output:                             OK
softint:                 (1.6s)
  -check result:                             OK
  -check output:                             OK
faultread:               (1.8s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:         (1.7s)
  -check result:                             OK
  -check output:                             OK
hello:                   (1.6s)
  -check result:                             OK
  -check output:                             OK
testbss:                 (1.8s)
  -check result:                             OK
  -check output:                             OK
pgdir:                   (1.7s)
  -check result:                             OK
  -check output:                             OK

  yield:                 (1.7s)
    -check result:                           OK
    -check output:                           OK
  badarg:                (1.8s)
    -check result:                           OK
    -check output:                           OK
  exit:                  (2.1s)
    -check result:                           OK
    -check output:                           OK
  spin:                  (4.7s)
    -check result:                           OK
    -check output:                           OK
  waitkill:              (13.8s)
    -check result:                           OK
    -check output:                           OK
  forktest:              (1.7s)
    -check result:                           OK
    -check output:                           OK
  forktree:              (1.7s)
    -check result:                           OK
    -check output:                           OK
  Total Score: 150/150
  zhangweikun$>
```

【实验总结】

完成实验后，请分析 ucore_lab 中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别

结合注释打代码，注释中每一步都很详细，按照注释打下来几乎和答案一模一样，并且最终运行效果是一样的。

最后在写实验报告的时候给代码加上了注释。

列出你认为本实验中重要的知识点，以及与对应的 OS 原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

1.实验涉及到从内核态切换到用户态的方法；

2.用到了 ELF 可执行文件的格式的知识；

3.实验涉及到用户进程的创建和管理；

4.涉及到简单的进程调度；

5.涉及了系统调用的实现；

对应了 OS 原理中的：

创建、管理、切换到用户态进程的具体实现；

加载 ELF 可执行文件的具体实现；

对系统调用机制的具体实现；

实验知识是理论知识的实际应用与实践。

列出你认为 OS 原理中很重要，但在实验中没有对应上的知识点

1.内核线程与用户线程的多对多模型。

2.线程池的概念

3.对死锁的多种处理方法

心得体会

通过本次实验，我对用户进程管理有了更深入的认识。本次实验中涉及到之前代码的更新，往往是牵一发而动全身。刚开始，即使答案正确，make grade 仍拿不到 150 就是因为之前的代码没有更新正确，导致检测未通过。这提示我要更深入、周全地考虑历史代码的遗留问题，使得前后代码不冲突地完成实验。

本次实验涉及的编程任务不多，主要是分析问题多。每个分析问题都帮助我更深入地了解了知识的应用。分析问题也涉及到之前实验的知识点，做起来也比较费劲，也有些不懂的问题及时请教了同学。

【参考文献】

《操作系统实验指导(清华大学)陈渝、向勇编著》