

操作系统原理实验

实验六（Ucore Lab5） - 用户进程管理

实验准备

主机环境：macOS X 10.14.5 Mojave

虚拟机环境：Linux Ubuntu 14.04 LTS

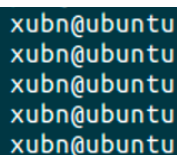
虚拟机搭载软件：Parallels Desktop.app

命令行终端：Linux 下 Terminal

终端shell：zsh

之前的第一次 ucore 实验中，实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好，卡顿严重，分辨率也不高等问题。故从这次实验开始，在Github上将原实验项目clone下来，自己按照指导书第一章讲的环境配置，安装所需支持，配置实用工具进行实验。

为满足实验要求，将命令行用户名字段临时指定为姓名。（需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果，故用我的中大NetID作为署名标记：许滨楠 - xubn。）



```
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~  
xubn@ubuntu ~
```

实验目的

- 了解第一个用户进程创建的过程；
- 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。

实验内容

- lab4 中完成了内核线程，但到目前为止所有的运行都在内核态执行。lab5 中将创建用户进程，让用户进程在用户态执行，并且在需要 ucore 支持时，可以通过系统调用，让 ucore 提供服务。
- 为此需要构造出第一个用户进程，并通过系统调用 `sys_fork/sys_exec/sys_exit/sys_wait` 来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。

练习

练习0

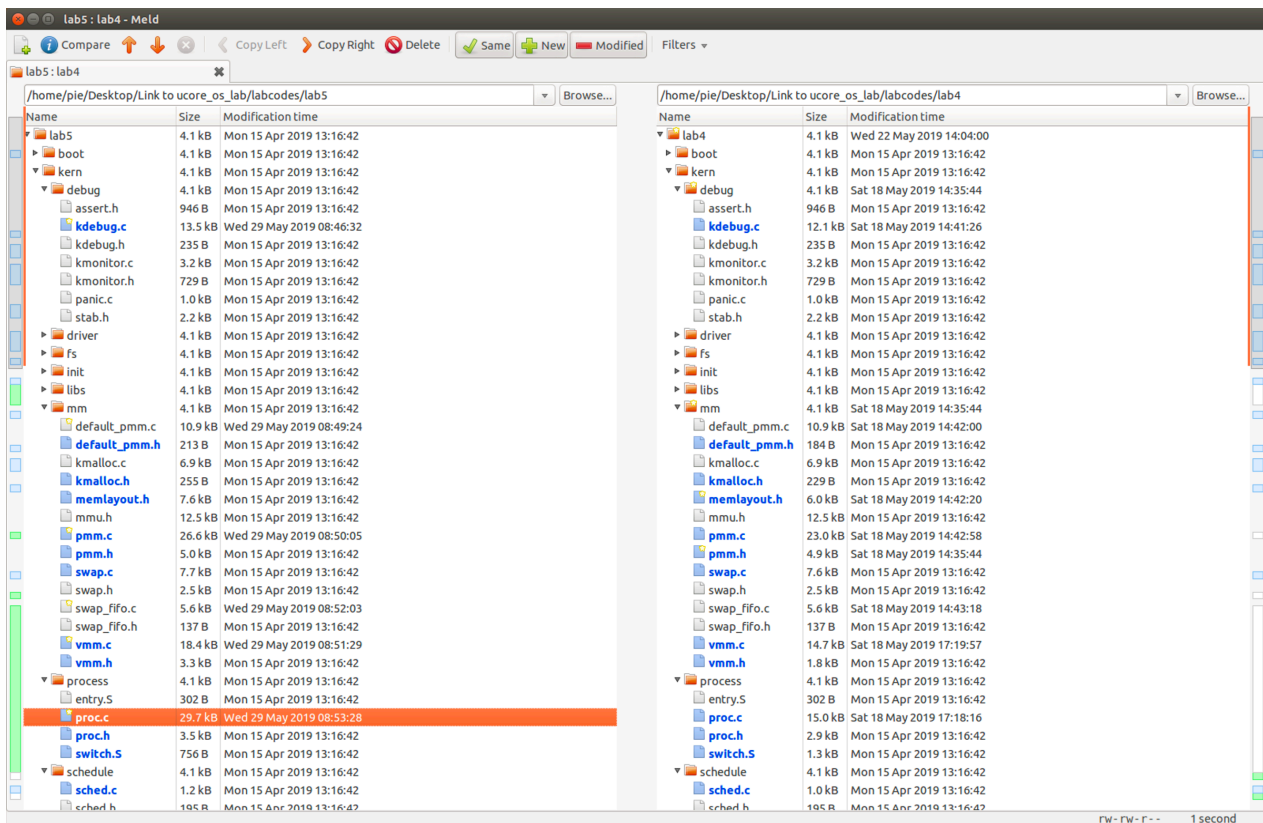
填写已有实验。

（为了能够正确执行 lab5 的测试应用程序，可能需要对已完成的 lab1 - lab4 的代码进行进一步改进）

利用 meld 工具，将之前完成的 lab1/2/3/4 代码 merge 到 lab5 的代码中。因为之前已经将 lab1/2/3 的代码整合到 lab4，所以这次直接将 lab4 和 lab5 的代码在 meld 中进行目录比较，合并已经完成的代码。

主要在之前编写过代码的文件中做比较，补充填入已完成的代码。由于之前在 merge 的时候错把一些新的 labcode 在与旧的 code merge 的时候去掉了，导致重要内容缺失，后续的测试总是出错，所以只需要把前面的 lab 练习中要求补充完整的函数 merge 到新的 lab 中。具体需要填写的文件和函数如下：

- Lab 1
 - `kdebug.c` : `print_stackframe`
 - `trap.c` : `idt_init`
 - `trap.c` : `trap_dispatch`
- Lab 2
 - `default_pmm.c` : `default_init`
 - `default_pmm.c` : `default_init_memmap`
 - `default_pmm.c` : `default_alloc_pages`
 - `default_pmm.c` : `default_free_pages`
 - `pmm.c` : `get_pte`
 - `pmm.c` : `page_remove_pte`
- Lab 3
 - `vmm.c` : `do_pgfault`
 - `swap_fifo.c` : `__fifo_map_swappable`
 - `swap_fifo.c` : `__fifo_swap_out_victim`
- Lab 4
 - `proc.c` : `alloc_proc`
 - `proc.c` : `do_fork`



先前操作的文件基本比对合并完毕，发现其中有许多之前已经完成过的函数中带有 update LAB4 steps 的注释，提示还需要对部分代码进行修改，以支持新的实验内容：

- **proc.c : alloc_proc**

- 这里需要将注释提示中的 `proc->wait_state` 置0，并将其中 `cptr`，`optr`，`yptr` 都赋值为空，完成初始化；

- **proc.c : do_fork**

- 注释提示：

```
1 update step 1: set child proc's parent to current process, make sure
  current process's wait_state is 0
2 update step 5: insert proc_struct into hash_list && proc_list, set
  the relation links of process
```

- 要求对原先代码过程中调用 `alloc_proc` 的步骤1和将取得的 `proc_struct` 插入链表的步骤5进行更新；

- 按照注释提示：

在步骤1处添加代码，将父进程指针指向当前进程后，用 `assert` 语句确保当前进程等待状态标记为0；

在步骤5处添加代码，将原来的简单技术改为调用 `set_link` 函数，实现进程相关连接的设置；

- **trap.c : idt_init**

- 注释提示添加一到两行代码，用于使用户应用可以使用系统中断，获得相关的系统服务，即需要设置系统中断门；

- 在之前已经完成的实验基础上，调用 SETGATE，将中断描述符 idt[T_SYSCALL] 的特权级设置为 DPL_USER，使其能在用户态下产生中断，中断向量处理地址位于 __vectors[T_SYSCALL]。建立好中断描述符之后，用户进程执行 INT T_SYSCALL 后，CPU 就会从用户态切换到内核态，保存相关寄存器，跳转到 __vectors[T_SYSCALL] 处开始执行。
- 在调用 lidt(&idt_pd) 之前添加对 SETGATE 的调用，完成代码补充：

```
1 SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
```

■ trap.c : trap_dispatch

- 根据注释提示，需要添加一到两行代码，实现每一个 TICK 循环中，进程的 need_resched 变量设置为1，表示该进程当前的时间片已经用完，需要进行调度；
- 补充代码如下：

```
1 ticks ++;
2 if (ticks % TICK_NUM == 0) {
3     assert(current != NULL);
4     current->need_resched = 1; //时间片用完设置为需要调度
5 }
6 break;
```

至此练习前的准备完成，将当前代码进行commit保存到版本管理仓库以备不时之需。接着进行后续练习。

```
xubn@ubuntu ~$ cd /labcodes/lab5
xubn@ubuntu ~$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       modified:   kern/debug/kdebug.c
       modified:   kern/mm/default_pmm.c
       modified:   kern/mm/pmm.c
       modified:   kern/mm/swap_fifo.c
       modified:   kern/mm/vmm.c
       modified:   kern/process/proc.c
       modified:   kern/trap/trap.c
       modified:   kern/trap/trap.h

no changes added to commit (use "git add" and/or "git commit -a")
xubn@ubuntu ~$ git add .
xubn@ubuntu ~$ git commit -m "2019/5/29 lab5 ini"
[master ede0f10] 2019/5/29 lab5 ini
8 files changed, 147 insertions(+), 12 deletions(-)
xubn@ubuntu ~$
```

练习1

加载应用程序并执行。

基本认识

依照实验指导书中的说明，`do_execv` 函数调用 `load_icode`（`kern/process/proc.c`）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防止应用程序的代码段、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。练习 1 中要求设置正确的 `trapframe` 内容。

`load_icode` 函数的作用在实验指导书中已经描述得很清楚，其主要工作是给用户进程创建一个能够让其正常运行的用户环境。

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

根据注释内容，lab5 的 exercise1 主要是要求完成对 `trapframe` 中的各成员变量的设置，且相关的提醒在实验源代码文件的备注中已经给出（见上图）。先看到 `trapframe` 数据结构的具体内容，位于 `/trap/trap.h`：

```
1 struct trapframe {
2     struct pushregs tf_regs;
3     uint16_t tf_gs;
4     uint16_t tf_padding0;
5     uint16_t tf_fs;
6     uint16_t tf_padding1;
7     uint16_t tf_es;
8     uint16_t tf_padding2;
9     uint16_t tf_ds;
10    uint16_t tf_padding3;
11    uint32_t tf_trapno;
12    /* below here defined by x86 hardware */
13    uint32_t tf_err;
14    uintptr_t tf_eip;
15    uint16_t tf_cs;
16    uint16_t tf_padding4;
17    uint32_t tf_eflags;
18    /* below here only when crossing rings, such as from user to kernel
19    */
19    uintptr_t tf_esp;
20    uint16_t tf_ss;
21    uint16_t tf_padding5;
22 } __attribute__((packed));
```

实现思路

结合 `trapframe` 数据结构的内容和注释提示，需要对中断帧进行设置，完成用户态的代码段、数据段和堆栈在 `tf` 中的设置，指令指针指向进程的第一条指令，使得中断 `iret` 返回时，能正确开始用户进程的执行，并确保在用户态能够响应中断。

tf_cs 代码段设置为 USER_CS 表示用户态代码段寄存器；将 tf 中的 tf_ds, tf_es, tf_ss 都设置为 USER_DS 表示用户态数据段寄存器；将 tf_esp 设置为用户态栈的顶部；tf_eip 设置为进程入口；tf_eflags 设置开中断。根据函数注释就可以进行相应设置了。

具体代码

```
1 static int
2 load_icode(unsigned char *binary, size_t size) {
3     ...
4
5     //(6) setup trapframe for user environment
6     struct trapframe *tf = current->tf;
7     memset(tf, 0, sizeof(struct trapframe));
8
9     tf->tf_cs = USER_CS;
10    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
11    tf->tf_esp = USTACKTOP;
12    tf->tf_eip = elf->e_entry;
13    tf->tf_eflags = FL_IF;
14
15    ...
16 }
```

问题解答

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。

当创建一个用户态进程并加载应用程序后，在这个练习中补充完成的 load_icode 会进行程序的运行，设置用户堆栈，建立虚拟地址与物理地址之间的映射，建立页表，进行 trapframe 的初始化设置，以使用户程序调用系统中断陷入内核态，然后执行完相应的中断处理程序，再回到用户态进程及继续执行，从而确保用户态下用户进程可以完成中断响应。这个基本设置下，当该进程被 ucore 选择占用 CPU 执行，执行 iret 中断返回，接着系统会从前面设置过的 tf 中恢复运行环境，进而执行应用程序的第一条指令。

练习2

父进程复制自己的内存空间给子进程。

基本认识

创建子进程的函数 do_fork 在执行中将复制当前进程（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（kern/mm/pmm.c）实现的，练习 2 中要求补充 copy_range 的实现，从而确保 do_fork 调用能正常执行。

copy_range 中提示代码如下：

```

/* 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 virtual 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
 */

```

函数补充部分总体的过程是需要找到父进程和子进程对应的地址然后利用函数进行复制并建立映射关系。

实现思路

需要补充的函数完成拷贝父进程中的内存内容到子进程的工作，可以分为拷贝和填页表两个部分：

- 调用 `page2kva(page)` 和 `page2kba(npage)` 分别得到父进程和子进程的内存虚拟页地址，然后调用 `memcpy` 函数进行拷贝；
- 调用 `page_insert` 完成子进程页表的映射。

具体代码

```

1  int
2  copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
   share){
3      ...
4
5      /* LAB5:EXERCISE2 YOUR CODE */
6      // (1) find src_kvaddr: the kernel virtual address of page
7      void* src_kva = page2kva(page);
8      // (2) find dst_kvaddr: the kernel virtual address of npage
9      void* dst_kva = page2kva(npage);
10     // (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
11     memcpy(dst_kva, src_kva, PGSIZE);
12     // (4) build the map of phy addr of nage with the linear addr start
13     ret = page_insert(to, npage, start, perm);
14
15     ...
16 }

```

问题解答

简要说明如何设计实现 Copy on Write 机制，给出概要设计。

Copy on Write 机制可以通过对这个练习中补充的 `copy_range` 进行修改实现。可以在进行父子进程信息复制的时候，不直接使用 `memcpy` 进行完全的“深拷贝”复制，而是只将子进程页目录项的指针也指向父进程的页目录项，然后将该页目录项的读写权限设置为只读，利用标志位标识各部分内存的复制情况（即记录表示该部分是否已经在写时完整复制过了）。当某个进程需要对页目录项进行修改的时候，因为权限不足会引发缺页异常，在异常处理中复制需要的内存，修改页面的标志位。

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。

首先根据实验指导书中所提供的信息，可以对上述几个执行命令的系统调用归纳如下：

系统调用名	含义	具体完成服务的函数
SYS_exit	process exit	do_exit
SYS_fork	create child process, dup mm	do_fork-->wakeup_proc
SYS_wait	wait child process	do_wait
SYS_exec	after fork, process execute a new program	load a program and refresh the mm

接下来对各个指令调用逐个理解：

fork

当程序执行 fork 时，从上表可知通过 SYS_fork 系统调用来调用 do_fork 函数以及 wakeup_proc 函数完成 fork 过程。至于 do_fork 函数的执行过程和功能，可以在 lab4 的分析中找到：

- 调用之前编写的 **alloc_proc** 函数，获得一块可用的已经初始化后的用户信息块；
- 调用 **setup_kstack** 函数，为进程分配一个内核堆栈；
- 调用 **copy_mm** 函数，将原进程的内存管理信息复制到新进程；
- 调用 **copy_thread** 函数，将原进程上下文复制到新进程；
- 将新建进程以前述提及的 **proc_struct** 形式添加到 **hash_list** 及 **proc_list** 中；
- 调用 **wakeup_proc** 函数，将新的子进程唤醒；
- 将新建进程的进程号 **pid** 作为函数的返回值。

在 do_fork 完成子进程的创建和资源分配后，由 wakeup_proc 通过 `proc->wait_state = 0`，将进程的状态设置为等待，将新建的子进程放进就绪队列。

exec

当程序执行 exec 时，通过 SYS_exec 系统调用来进一步调用 do_execve 函数。这个函数可以完成用户进程的创建工作，同时让用户进程开始执行。

- 为新的执行码准备好空间，需要时回收原内存空间：
 - 如果 mm 不为 NULL，则设置页表为内核空间页表；
 - 判断 mm 引用计数是否为1，若是则说明没有其他进程再需要该内存空间，可以释放，取消映射；
 - 把当前进程的内存管理指针设置为空；
- 加载应用程序执行码到当前进程新建的用户态虚拟空间中；
- 调用之前练习中的 load_icode，加载应用程序（具体过程已经于练习1中给出）。

wait

当程序执行 wait 时，通过 SYS_wait 系统调用来进一步调用 do_wait 函数。这个函数的主要功能为父进程等待子进程执行完毕，并负责释放子进程占用的资源：

- 对进程链表进行遍历，根据调用 wait 时传入的 pid 参数寻找对应子进程，如果 pid 遍历之后的结果不为0，则找到了对应子进程，否则返回错误信息；
- 如果根据 pid 判断得子进程的状态不为 PROC_ZOMBIE，说明子进程还没有退出，则要将当前进程的执行状态设置为 PROC_SLEEPING，wait_state 设置为 WT_CHILD，表示当前进程挂起等待其子进程完成执行，调用 schedule 函数完成调度等待，等待下一次唤醒重新进行上述判断；
- 若该子进程状态为 PROC_ZOMBIE，表示进程已经处于退出状态，则释放子进程占用的资源，返回。

exit

当程序执行 exit，将通过 SYS_exit 调用 do_exit 函数实现，完成对进程占用的资源的释放和进程的结束运行。具体的过程为：

- 判断当前进程是否为用户进程，是则开始回收该用户进程占用的用户态虚拟内存空间；
- 释放页表项中所记录的物理内存空间、mm、vma、页目录占用的内存空间；
- 将自身 state 设置为 PROC_ZOMBIE，退出码为 error_code，用以标识该进程无法再被调度；
- 自身子进程判断：
 - 如果当前进程有子进程，则需要将子进程挂载到内核线程 init；
 - 如果当前进程有处于退出态的子进程，则需要唤醒 init 来对其进行回收；
- 唤醒父进程进行回收：
 - 如果当前进程的父进程处于前面提到的等待状态且是 WT_CHILD，应该唤醒等待子进程结束中的父进程来进行子进程资源回收；
- 执行 schedule 函数进行调度，调度新的进程占用 CPU 执行。

SYSCALL

进行系统调用时，执行 INT T_SYSCALL（syscall 接口在 user/libs/syscall.c 中封装）指令之后，CPU 根据系统建立的系统调用中断描述符（在 idt_init 中完成初始化）转入内核态，然后开始执行系统调用。在内核函数执行之前要先保存系统调用前的 CPU 现场以便执行完毕后恢复原态运行，将其保存在前述的 tf 结构体中，之后便可以开始进行系统调用（相应的处理函数在 syscall.c 中用函数数组进行了存储），完成调用之后再调用 IRET 返回到原态，恢复现场。

问题解答

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

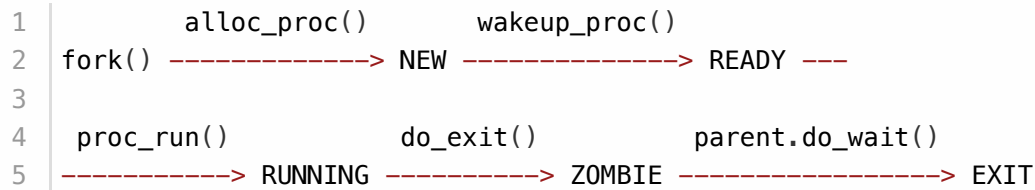
fork: 在完成内存拷贝之后，将子进程设置为就绪态的操作中改变了进程的执行状态；

exec: 将（原来另一进程的）空间回收后，让新的进程得到资源并运行；

wait: 判断子进程的运行状态并进行相应处理，如果需要则改变父进程的执行状态 - 对其进行挂起等待子进程执行完毕，再唤醒回收；

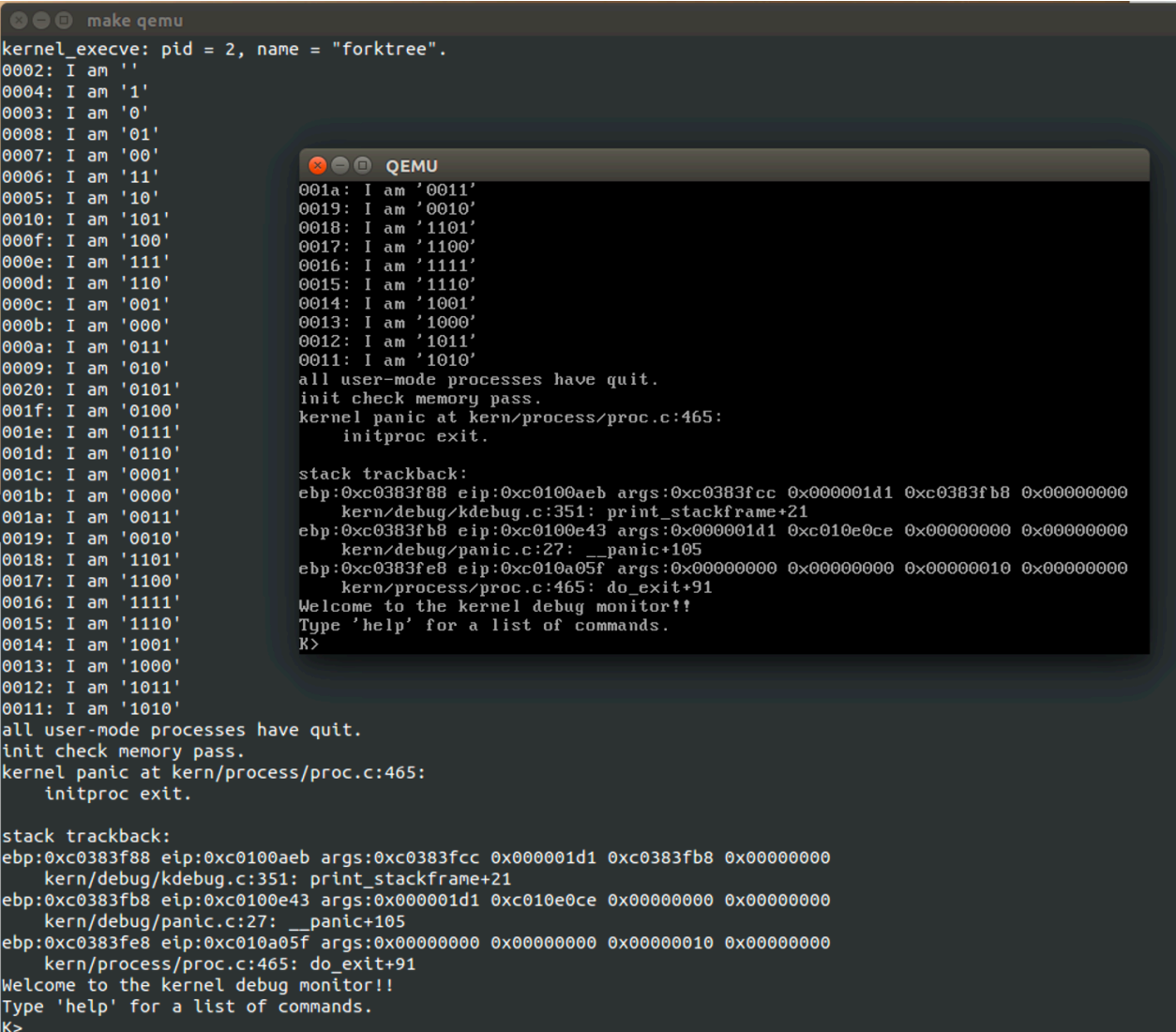
exit: 将自身子进程重新挂载到 init 内核线程，若有已退出进程则回收；将进程自身的运行状态设置为 PROC_ZOMBIE 的退出状态，唤醒父进程。

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、其间的变换关系、产生变换的事件或函数调用）。



实验结果

完成实验中各部分代码的编写后，命令行 cd 进入实验目录 lab5，随后执行 **make qemu** 命令，查看结果：



```
make qemu
kernel_exe: pid = 2, name = "forktree".
0002: I am ''
0004: I am '1'
0003: I am '0'
0008: I am '01'
0007: I am '00'
0006: I am '11'
0005: I am '10'
0010: I am '101'
000f: I am '100'
000e: I am '111'
000d: I am '110'
000c: I am '001'
000b: I am '000'
000a: I am '011'
0009: I am '010'
0020: I am '0101'
001f: I am '0100'
001e: I am '0111'
001d: I am '0110'
001c: I am '0001'
001b: I am '0000'
001a: I am '0011'
0019: I am '0010'
0018: I am '1101'
0017: I am '1100'
0016: I am '1111'
0015: I am '1110'
0014: I am '1001'
0013: I am '1000'
0012: I am '1011'
0011: I am '1010'
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:465:
initproc exit.

stack traceback:
ebp:0xc0383f88 eip:0xc0100aeb args:0xc0383fcc 0x000001d1 0xc0383fb8 0x00000000
kern/debug/kdebug.c:351: print_stackframe+21
ebp:0xc0383fb8 eip:0xc0100e43 args:0x000001d1 0xc010e0ce 0x00000000 0x00000000
kern/debug/panic.c:27: __panic+105
ebp:0xc0383fe8 eip:0xc010a05f args:0x00000000 0x00000000 0x00000010 0x00000000
kern/process/proc.c:465: do_exit+91
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

可以看到显示结果基本如实验要求中，内容基本一致。

执行 `make grade` 命令，查看利用 shell 脚本进行检查的检查结果：

```
xubn@ubuntu labcodes/lab5 master ● make grade
badsegment: (1.7s)
-check result: OK
-check output: OK
divzero: (1.5s)
-check result: OK
-check output: OK
softint: (1.5s)
-check result: OK
-check output: OK
faultread: (1.5s)
-check result: OK
-check output: OK
faultreadkernel: (1.5s)
-check result: OK
-check output: OK
hello: (1.5s)
-check result: OK
-check output: OK
testbss: (1.9s)
-check result: OK
-check output: OK
pgdir: (1.5s)
-check result: OK
-check output: OK
yield: (1.5s)
-check result: OK
-check output: OK
badarg: (4.2s)
-check result: OK
-check output: OK
exit: (1.8s)
-check result: OK
-check output: OK
spin: (4.6s)
-check result: OK
-check output: OK
waitkill: (13.6s)
-check result: OK
-check output: OK
forktest: (1.5s)
-check result: OK
-check output: OK
forktree: (1.5s)
-check result: OK
-check output: OK
Total Score: 150/150
```

基本满足实验要求。

实验总结

分析与区别

因为源代码中的注释内容已经非常具体清晰，每一个变量的设置和函数的调用都已经有了详尽的提示，因此根据注释进行代码补充，结果与答案基本一致。

重要知识点

- 系统调用的初始化设置，用户态的权限设置；

- 中断帧的具体内容设置和初始化；
- 父子进程内存空间的拷贝；

补充知识点

因为本次的实验内容（如之前的许多内容一样）比较细节化具体化，而且主要是要求完成某个功能片段的补充实现，较为基础细节的实现和练习难免会带来全局认识的缺失，比如因为只完成了父子进程的内存空间管理，所以对整个父子进程的创建执行过程实现就比较不清晰。

体会与反思

已经是第五次 ucore 实验了，这次的实验跟前两次一样，因为注释非常详尽，所以即使还要对原来的内容进行补充，完成下来的过程也还是比较顺利的，过程中还总是担心遗漏了需要补充的内容所以全局搜索了补充关键字，直到最后能正常调试运行的时候才放下心，调试的时候也没有遇到太多bug。并且基本可以理解各个函数完成的功能以及其调用链的情况。对整个用户进程的管理和系统中断的使用有了进一步的实践和认识。

需要反思的是，会否在这样的过程中对提示性的信息产生了过重的依赖，从而导致自己的思考探索能力没有得到锻炼或者没有发现自己想法和认知上的漏洞。所以在实验中也要多注意自己对知识的理解和认识才是。