《操作系统原理》lab5实验报告

专业：计算机科学与技术

姓名：郝裕玮

学号：18329015

1. 实验目的

①了解第一个用户进程创建过程

②了解系统调用框架的实现机制

③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

2. 实验过程与结果

练习 0：填写已有实验

本实验依赖实验 1/2/3/4。请把你做的实验 1/2/3/4 的代码填入本实验中代码中有“LAB1”/“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 1/2/3/4 的代码进行进一步改进。

答：需要修改的有 debug/kdebug.c，trap/trap.c，mm/default_pmm.c，mm/pmm.c，kern/process/proc.c，mm/vmm.c，swap_fifo.c 这 7 个文件，使用 meld 进行修改即可。

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，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。

请在实验报告中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。

（1）请在实验报告中简要说明你的设计实现过程。

答：根据 load_icode 函数的注释，可知该函数的作用为：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，具体流程和代码逻辑如下：

①调用 mm_create( )，为当前进程创建一个新的 mm；

②创建一个新的页目录表，并调用 setup_pgdir( )使得 mm->pgdir 指向该页目录表的内核虚拟地址；

③复制 TEXT/DATA 部分，以二进制形式构建 BSS 部分到进程的内存空间：

（1）获取二进制程序的文件头（ELF 格式）；

（2）获取二进制程序的程序段头的入口（ELF 格式）；

（3）判断该程序是否有效；

（4）找到每个程序段的头部；

（5）调用 mm_map( )来设置新的 vma (ph->p_va，ph->p_memsz)；

（6）分配内存，并将每个程序段的内容复制到进程内存中；

（i）复制二进制程序的 TEXT/DATA 部分

（ii）构建二进制程序的 BSS 部分

④构建用户堆栈内存；

⑤设置当前进程的 mm，sr3，并将页目录的物理地址存在 CR3 寄存器中；

⑥为用户环境设置中断帧；

根据 LAB 5:EXERCISE1 YOUR CODE 部分注释可将代码补充完整（具体解析已放在代码注释中），补充部分代码如下所示：

```
    tf->tf_cs=USER_CS;
    //tf_cs should be USER_CS segment (see memlayout.h)
    tf->tf_ds=tf->tf_es=tf->tf_ss=USER_DS;
    //tf_ds=tf_es=tf_ss should be USER_DS segment
    tf->tf_esp=USTACKTOP;
    //tf_esp should be the top addr of user stack (USTACKTOP)
    tf->tf_eip=elf->e_entry;
    //tf_eip should be the entry point of this binary program (elf->e_e
ntry)
    tf->tf_eflags=FL_IF;//FL_IF 为中断打开状态
    //tf_eflags should be set to enable computer to produce Interrupt
```

（2）请在实验报告中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。

答：整体过程为：

①首先调用 schedule 函数，在调度器占用了 CPU 资源之后，用户态进程调用 exec 系统调用，从而转入系统调用的处理例程；

②之后操作系统开始执行中断处理例程，并将控制权转移到 syscall.c 中的 syscall 函数，之后根据系统调用号转移到 sys_exec 函数，最后在该函数中调用 do_execve( )来完成指定应用程序的加载；

③在 do_execv 函数中，若 mm 不为空，则设置其页表为内核空间页表，再调用自减函数 mm_count_dec( )，若 mm 的值 -1 后变为 0，则说明该进程所占的内存空间空闲，可释放该进程页表所占用的用户内存空间。再依次按顺序调用 exit_mmap( )，put_pgdir( )和 mm_destroy( )来回收自身所占的用户内存空间；

④之后 do_execv 函数再调用 load_icode( )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，并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；

⑤do_exec 函数执行完毕后，进行正常的中断返回的流程。因为中断处理例程的栈上面的 eip 已经被修改成应用程序的入口处，且 CS 上的 CPL 为用户态，所以 iret 进行中断返回时会将堆栈切换到用户栈，把特权级从内核级切换到用户级，并最终跳转到具体执行应用程序的第一条指令处开始执行。

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。

答：首先由练习 0 可知，我们需要对已完成的实验 1/2/3/4 的代码进行进一步改进。

①首先我们观察到 kern/process/proc.h 中关于 proc_struct 结构体的定义多出了两行内容：

```
    uint32_t wait_state;                        // waiting state
    struct proc_struct *cptr, *yptr, *optr;     // relations between pr
ocesses
```

所以修改后的 proc.c 中的 alloc_proc 函数如下所示（为了方便定位新增代码的位置，删去了在之前实验中对于该部分代码的注释）：

```
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+1);
    //以下 2 行为新增内容
        proc->wait_state=0;//进程控制块中新增的条目，初始化进程等待状态
        proc->cptr=proc->optr=proc->yptr=NULL;//进程相关指针初始化
    }
    return proc;
}
```

②同时由①可知，我们需要对 proc.c 中的 do_fork 函数也进行修改（为了方便定位新增代码的位置，删去了在之前实验中对于该部分代码的注释）：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;

    if((proc=alloc_proc())==NULL){
        goto fork_out;
    }
    proc->parent=current;
    assert(current->wait_state==0);//确保当前进程正在等待
    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();
        hash_proc(proc);
        set_links(proc);//单纯的计数加法并不能满足进程管理调度
        //所以调用 set_link 函数来设置进程之间的链接。
    }
    local_intr_restore(intr_flag);
    wakeup_proc(proc);
    ret=proc->pid;
```

```
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

③kern/trap/trap.c 中的 idt_init 函数（为了方便定位新增代码的位置，删去了在之前实验中对于该部分代码的注释）：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i=0;
    for(i=0;i<(sizeof(idt)/sizeof(struct gatedesc));i++){
            SETGATE(idt[i],0,GD_KTEXT,__vectors[i],DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL],1,GD_KTEXT,__vectors[T_SYSCALL],DPL_USER);
    //设置给用户态使用的中断门，为 T_SYSCALL 设置用户态权限，使得用户态能够进行系统调用
    lidt(&idt_pd);
}
```

④kern/trap/trap.c 中的 trap_dispatch 函数（为了方便定位新增代码的位置，删去了在之前实验中对于该部分代码的注释）：

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;
    int ret=0;

    switch (tf->tf_trapno) {
    case T_PGFLT:
        if ((ret = pgfault_handler(tf)) != 0) {
            print_trapframe(tf);
            if (current == NULL) {
                panic("handle pgfault failed. ret=%d\n", ret);
            }
            else {
                if (trap_in_kernel(tf)) {
                    panic("handle pgfault failed in kernel mode. ret=%d\n", ret);
```

```
                }
                cprintf("killed by kernel.\n");
                panic("handle user mode pgfault failed. ret=%d\n", ret)
;
                do_exit(-E_KILLED);
            }
        }
        break;
    case T_SYSCALL:
        syscall();
        break;
    case IRQ_OFFSET + IRQ_TIMER:
#if 0

#endif
        ticks ++;
        if(ticks==TICK_NUM){
            assert(current!=NULL); //确保当前进程存在
            current->need_resched=1;//每个时间片用完后需要对进程进行调度
        }
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        break;
    default:
        print_trapframe(tf);
        if (current != NULL) {
            cprintf("unhandled trap.\n");
            do_exit(-E_KILLED);
        }
        panic("unexpected trap in kernel.\n");
```

```
    }
}
```

接下来我们可以正式根据相关注释来补充 copy_range 函数（具体内容和解释已放在代码注释中），代码如下所示：

```
/* copy_range - copy content of memory (start, end) of one process A to
another process B
 * @to:    the addr of process B's Page Directory
 * @from:  the addr of process A's Page Directory
 * @share: flags to indicate to dup OR share. We just use dup method, s
o it didn't be used.
 *
 * CALL GRAPH: copy_mm-->dup_mmap-->copy_range
 */
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
 share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr st
art
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr st
art. If pte is NULL, just alloc a PT
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
        uint32_t perm = (*ptep & PTE_USER);
        //get page from ptep
        struct Page *page = pte2page(*ptep);
        // alloc a page for process B
        struct Page *npage=alloc_page();
        assert(page!=NULL);
        assert(npage!=NULL);
        int ret=0;
        /* LAB5:EXERCISE2 YOUR CODE
```

```
         * replicate content of page to npage, build the map of phy add
r of nage with the linear addr start
         *
         * Some Useful MACROs and DEFINEs, you can use them in below im
plementation.
         * MACROs or Functions:
         *    page2kva(struct Page *page): return the kernel vritual ad
dr of memory which page managed (SEE pmm.h)
         *    page_insert: build the map of phy addr of an Page with th
e 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 PGSIZ
E
         * (4) build the map of phy addr of  nage with the linear addr
start
         */
        void *src_kvaddr=page2kva(page); //获得父进程（源页面）的内核虚拟
页地址
        //find src_kvaddr: the kernel virtual address of page
        void *dst_kvaddr=page2kva(npage); //获得子进程（目标页面）的内核虚
拟页地址
        //find dst_kvaddr: the kernel virtual address of npage
        memcpy(dst_kvaddr,src_kvaddr,PGSIZE);// 将父进程数据复制到子进程
中，大小为 PGSIZE
        //memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
        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
        //build the map of phy addr of  nage with the linear addr start
        assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

补充完毕后，执行 make qemu，结果如下所示：

```
moocos-> make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002a (phys)
  etext  0xc010bddf (phys)
  edata  0xc019bf2a (phys)
  end    0xc019f0b8 (phys)
Kernel executable memory footprint: 637KB
ebp:0xc0129f38 eip:0xc0100adf args:0x00010094 0x00000000 0xc0129f68 0xc01000d3
    kern/debug/kdebug.c:339: print_stackframe+21
ebp:0xc0129f48 eip:0xc0100dce args:0x00000000 0x00000000 0x00000000 0xc0129fb8
    kern/debug/kmonitor.c:129: mon_backtrace+10
ebp:0xc0129f68 eip:0xc01000d3 args:0x00000000 0xc0129f90 0xffff0000 0xc0129f94
    kern/init/init.c:58: grade_backtrace2+33
ebp:0xc0129f88 eip:0xc01000fc args:0x00000000 0xffff0000 0xc0129fb4 0x0000002a
    kern/init/init.c:63: grade_backtrace1+38
ebp:0xc0129fa8 eip:0xc010011a args:0x00000000 0xc010002a 0xffff0000 0x0000001d
    kern/init/init.c:68: grade_backtrace0+23
ebp:0xc0129fc8 eip:0xc010013f args:0xc010bdfc 0xc010bde0 0x0000318e 0x00000000
    kern/init/init.c:73: grade_backtrace+34
ebp:0xc0129ff8 eip:0xc010007f args:0x00000000 0x00000000 0x0000ffff 0x40cf9a00
    kern/init/init.c:33: kern_init+84
memory management: default_pmm_manager
e820map:
```

```
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efe000, [00100000, 07ffdfff], type = 1.
  memory: 00002000, [07ffe000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ----------------------
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
```

```
BEGIN check_swap: count 31866, total 31866
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
page fault at 0x00002000: K/W [no page found].
page fault at 0x00003000: K/W [no page found].
page fault at 0x00004000: K/W [no page found].
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
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
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
```

```
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
count is 5, total is 5
check_swap() succeeded!
++ setup timer interrupts
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:429:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

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 在实现中是如何影响进程的执行状态的？

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）

对 fork/exec/wait/exit 函数的分析：

①对于 fork 函数：

（1）检查当前总进程数目是否超过限制，若超过则返回 E_NO_FREE_PROC；

（2）调用 alloc_proc 函数为当前进程分配内存；

（3）调用 setup_kstack 函数为内核进程（线程）分配栈空间；

（4）调用 copy_mm 函数来拷贝或共享内存空间；

（5）调用 copy_thread 函数来复制父进程的中断帧和上下文；

（6）调用 get_pid 函数来为进程分配一个 PID；

（7）将进程控制块加入全局进程链表（hash_list 和 proc_list）；

（8）调用 wakeup_proc 唤醒新进程，把进程状态设置为就绪态；

（9）返回子进程的 PID；

②对于 exec 函数：

（1）调用 user_mem_check 函数检查进程名称的地址和长度是否合法；

（2）如有需要则释放内存空间；

（3）调用 load_icode 函数将程序加载到相应的用户内存空间当中；

③对于 wait 函数：

（1）判断 pid，若 pid != 0，则寻找 id 为 pid 且状态为 ZOMBIE 态的子进程。反之则寻找任意一个状态为 ZOMBIE 态的子进程；

（2）若不存在 ZOMBIE 态的子进程（表明该进程尚未退出），则将当前进程状态设置为 SLEEPING 态，并调用 schedule 函数来继续执行新进程，直至有子进程被唤醒，此时跳回步骤（1）继续循环执行；

（3）若存在 ZOMBIE 态的子进程（表明该进程已退出），则让其父进程对该子进程进行资源回收和内存释放。

④对于 exit 函数：

（1）判断是否为用户级进程，是则回收其内存空间，反之则跳出；

（2）设置当前进程状态为 PROC_ZOMBIE，并设置返回码为 error_code。表明该进程无法再被调度，只能等待其父进程将其回收；

（3）若父进程处于等待态（wait_state == WT_CHILD），则唤醒父进程并使其对子进程进行资源回收和内存释放；

（4）若当前进程也有子进程，则将当前进程的所有子进程变为 init 的子进程，并由 init 对这些子进程进行资源回收和内存释放；

（5）调用 schedule 函数来继续执行新进程；

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

答：①fork：创建子进程，并将子进程的状态从 UNINIT 态改为 RUNNABLE 态，但是不改变父进程的状态；

②exec：只修改当前进程中执行的程序，不会影响当前进程的执行状态；

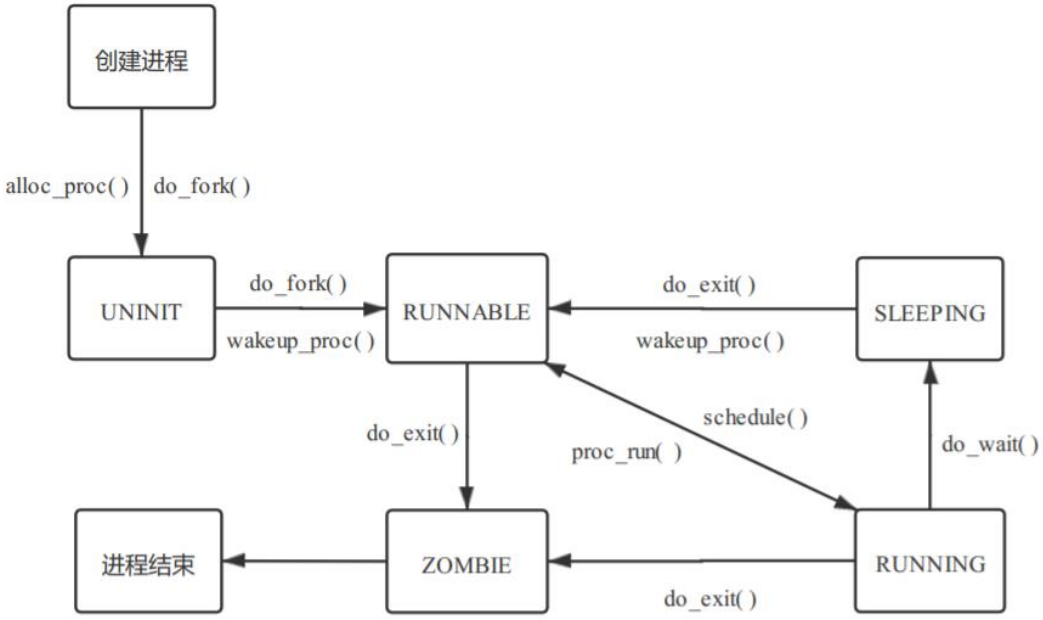
③wait：如果有已经结束的子进程或没有子进程，则调用结束，不影响进程状

态；反之因为进程需要等待子进程结束，所以会将当前进程置为 SLEEPING 态，并等待执行了 exit 的子进程将其唤醒；

④exit：将当前进程的状态从 RUNNING 态修改为 ZOMBIE 态，并唤醒父进程对其进行资源回收和内存释放。

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）

答：生命周期图如下所示：



3. 实验感想

本次实验难度相较于 lab4 又有所提升，主要体现在对各个函数作用及其之间的逻辑关系的分析上。进程的生成，切换，执行，销毁始终是我们操作系统的一大难点之一，所以想要完全看懂源码并给出解释还是需要一定的时间与功夫的。好在本次实验需要补充的代码不多，难度也不高，根据代码注释的提示即可完成。

本次实验的困难主要体现在 make grade 和 make qemu 上，一开始我的程序

总是在最后报错“nr_process == 2”

```
kernel panic at kern/process/proc.c:815:
    assertion failed: nr_process == 2
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
```

反复排查后才发现是我之前的代码中多写了一行 nr_process++，才导致后面的 assert 断言报错，将其删除后，make qemu 结果正确。

```
bool intr_flag;
local_intr_save(intr_flag);
{
    proc->pid=get_pid();
    hash_proc(proc);
    set_links(proc);//单纯的计数加法并不能满足进程管理调度
    //所以调用set_link函数来设置进程之间的链接。
    nr_process++;
}
```

但是此时 make grade 仍然没有满分（但是将 make qemu 结果与标准答案对比后并无差错），在参考了操统群里大佬的发言之后，我尝试将 proc.c 中的 assert 断言注释掉，发现成功得到满分（虽然这样感觉有点投机取巧，但是既然我 make qemu 的结果没有差错，我有理由相信我的代码没有问题，或许正是因为 lab5 的 make grade 标准有问题，黄聃老师才删去了 make grade 的要求）。

通过本次实验，我明白了用户态进程的创建和执行机制，同时明白了内核级进程和用户级进程的区别，懂得了用户态进程与内存管理和进程调度的关系及其应用方式，并掌握了用户态进程中不同状态的转换关系及其具体的函数调用方法。