实验4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。实验5将创建用户进程，让用户进程在用户态执行，且在需要ucore支持时，可通过系统调用来让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。相关原理介绍可看附录B。

练习0：经典合并代码

代码补充部分

```
proc.c
default_pmm.c
pmm.c
swap_fifo.c
vmm.c
trap.c
```

1. 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->optr = proc->yptr = NULL;//设置指针
    }
    return proc;
}
```

注解：

```
// 新增的两行
proc->wait_state = 0; //初始化进程等待状态
proc->cptr = proc->optr = proc->yptr = NULL; //指针初始化
```

```
// 指针注解
process relations
parent:           proc->parent  (proc is children)
children:         proc->cptr    (proc is parent)
older sibling:    proc->optr    (proc is younger sibling)
younger sibling:  proc->yptr    (proc is older sibling)
```

2. do_fork函数

```
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); //确保进程在等待
    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); //设置进程链接
    }
    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;
}
```

注解:

```
// 新增
assert(current->wait_state == 0); //确保进程在等待
set_links(proc); //设置进程链接

// set_links函数详解
// set_links函数的作用就是设置当前进程的process relations
static void
set_links(struct proc_struct *proc) {
    list_add(&proc_list,&(proc->list_link));//进程加入进程链表
    proc->yptr = NULL; //当前进程的younger sibling为空
    if ((proc->optr = proc->parent->cptr) != NULL) {
        proc->optr->yptr = proc; //当前进程的older sibling为当前进程
    }
    proc->parent->cptr = proc; //父进程的子进程为当前进程
    nr_process ++; //进程数加一
}
```

3. idt_init函数

```
void idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    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);
    lidt(&idt_pd);
}
```

注解：

```
// 新增
SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
// 这里主要是设置相应的中断门
```

4. trap_dispatch函数

```
ticks ++;
if (ticks % TICK_NUM == 0) {
    assert(current != NULL);
    current->need_resched = 1;
}
break;
```

注解：

```
// 新增
current->need_resched = 1;
```

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。

首先对于alloc_proc根据注释添加代码即可#~~注释写这么清楚不会有人看不懂吧~~

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
    /*
     * below fields in proc_struct need to be initialized
     *       enum proc_state state;                      // Process state
     *       int pid;                                    // Process ID
     *       int runs;                                   // the running
times of Proces
     *       uintptr_t kstack;                           // Process kernel
stack
     *       volatile bool need_resched;                 // bool value:
need to be rescheduled to release CPU?
     *       struct proc_struct *parent;                 // the parent
process
     *       struct mm_struct *mm;                       // Process's
memory management field
     *       struct context context;                     // Switch here to
run process
     *       struct trapframe *tf;                       // Trap frame for
current interrupt
     *       uintptr_t cr3;                              // CR3 register:
the base addr of Page Directroy Table(PDT)
     *       uint32_t flags;                             // Process flag
     *       char name[PROC_NAME_LEN + 1];               // Process name
     */
        proc->state = PROC_UNINIT; //进程为初始化状态
        proc->pid = -1;            //进程PID为-1
        proc->runs = 0;            //初始化时间片
        proc->kstack = 0;          //内核栈地址
```

```
        proc->need_resched = 0;     //不需要调度
        proc->parent = NULL;        //父进程为空
        proc->mm = NULL;            //虚拟内存为空
        memset(&(proc->context), 0, sizeof(struct context));//初始化上下文
        proc->tf = NULL;            //中断帧指针为空
        proc->cr3 = boot_cr3;       //页目录为内核页目录表的基址
        proc->flags = 0;            //标志位为0
        memset(proc->name, 0, PROC_NAME_LEN);//进程名为0
     //LAB5 YOUR CODE : (update LAB4 steps)
    /*
     * below fields(add in LAB5) in proc_struct need to be initialized
     *       uint32_t wait_state;                        // waiting state
     *       struct proc_struct *cptr, *yptr, *optr;     // relations
between processes
     */
        proc->wait_state = 0;
        proc->cptr = NULL;
        proc->optr = NULL;
        proc->yptr = NULL;
    }
    return proc;
}
```

新增的四个量含义如下

```
proc->wait_state = 0; //初始化进程等待状态
proc->cptr = NULL; //chlidren置为空
proc->optr = NULL; //older sibling置为空
proc->yptr = NULL; //young sibling置为空
//加上之前初始化的父进程，这个进程的相关指针就初始化完毕了
proc->parent = NULL;//父进程为空
```

之后我们来看load_icode。这个看这个函数的时候我的心情是无比刺激的。刚进去映入眼帘的就是什么(1),(2),(3),(3.1)，这些东西长得实在太像让我填代码的注释了，我直呼吾命休矣。然后翻到底下，哦没事了，原来上面都不是要填的，我们要填的是设置tf。而要填的tf给了可以说是保姆级别的注释，直接填就完事了#~~注释写这么清楚不会有人看不懂吧~~。填完的代码如下

```
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;
        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;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        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;
    memset(tf, 0, sizeof(struct trapframe));
    /* 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
     */
    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;
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;
}
```

然后就是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过程。这个过程可以看之前吓我一跳的那些注释，分为以下步骤

1. 使用mm_create来申请一个新的mm并初始化
2. 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
3. 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
4. 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
5. 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

6. 将mm->pgdir赋值给cr3以更新用户进程的虚拟内存空间。
7. 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。

需要注意的是，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。

```
 /*        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) //获取page结构的内核虚拟地址
         *    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
         */
```

按照提示，在循环当中，首先找到待拷贝的源地址和目的地址，然后使用memcpy函数复制一个页（每次一个页）的内容至目的地址，最后建立虚拟地址到物理地址的映射。

写出代码如下：

```
    void * kva_src = page2kva(page);
        void * kva_dst = page2kva(npage);

        memcpy(kva_dst, kva_src, PGSIZE);
```

```
        ret = page_insert(to, npage, start, perm);
```

- copy-on-write机制

在父进程执行do_fork函数创建子进程时进行浅拷贝：在进行内存复制的部分，比如copy_range函数内部，不实际进行内存的复制，而是将子进程和父进程的虚拟页映射上同一个物理页面，然后在分别在这两个进程的虚拟页对应的PTE部分将这个页置成是不可写的，同时利用PTE中的保留位将这个页设置成共享的页面； 在子进程产生page fault时进行深拷贝：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。

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）
>
> 执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。

目前ucore的系统调用为：

```
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,只是调用的参数不同而已（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。

1. fork

```
// 调用过程: fork->SYS_fork->do_fork + wakeup_proc

// wakeup_proc 函数主要是将进程的状态设置为等待。

// do_fork()
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。
```

2. exec

```
// 调用过程:  SYS_exec->do_execve

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。
```

3. wait

```
// 调用过程:  SYS_wait->do_wait

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。
```

4. exit

```
// 调用过程： SYS_exit->exit

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。
```

Q&A

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

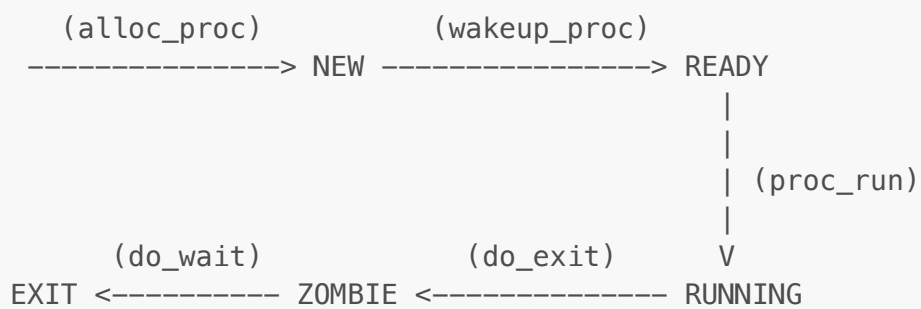
①fork：执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程

②exit：会把一个退出码error_code传递给ucore，ucore通过执行内核函数do_exit来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。

③execve：完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步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。

④wait：等待任意子进程的结束通知。wait_pid函数等待进程id号为pid的子进程结束通知。这两个函数最终访问sys_wait系统调用接口让ucore来完成对子进程的最后回收工作。

2. 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）。



Challenge：实现 Copy on Write 机制

> 这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在ucore操作系统中，当一个用户 父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进 程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用 户内存空间中的某页面时，ucore会通过page fault异常获知该操作，并完成拷贝内存页面，使 得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请 在ucore中实现这样的COW机制。

如果要实现“Copy on Write机制”，可以在现有代码的基础上稍作修改。修改内容：

- 在执行do_fork时，子进程的页目录表直接拷贝父进程的页目录表，而不是拷贝内核页目录表；在dup_mmap，只需保留拷贝vma链表的部分，取消调用copy_range来为子进程分配物理内存。
- 将父进程的内存空间对应的所有Page结构的ref均加1，表示子进程也在使用这些内存
- 将父子进程的页目录表的写权限取消，这样一旦父子进程执行写操作时，就会发生页面访问异常，进入页面访问异常处理函数中，再进行内存拷贝操作，并恢复页目录表的写权限。