练习0：填写已有实验

以下操作都是将**内容**复制到 lab5 里（都可以从 lab4 复制）**不要复制整个文件！！**

将 lab1 的 `kern/debug/kdebug.c`、`kern/init/init.c` 以及 `kern/trap/trap.c` 复制到 lab5 里

将 lab2 的 `kern/mm/pmm.c` 和 `kern/mm/default_pmm.c` 复制到 lab5 里

将 lab3 的 `kern/mm/vmm.c` 和 `kern/mm/swap_fifo.c` 复制到 lab5 里

最后将 lab4 的 `kern/process/proc.c` 复制到 lab5 里，之后还要改进代码

alloc_proc 函数源码

写于：**kern/process/proc.c**

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
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
     */
     //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
     */
    }
    return proc;
}
```

可以看出来要多加两行初始化的代码

proc_struct 结构体

写于：**kern/process/proc.h**

先来看新的 proc_struct 结构体，里面在最后多加了两行内容

```c
struct proc_struct {
    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
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
    int exit_code;                              // exit code (be sent to parent proc)
    uint32_t wait_state;                        // waiting state
    struct proc_struct *cptr, *yptr, *optr;     // relations between processes
};
```

alloc_proc 函数答案

写于：**kern/process/proc.h**

```c
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->pid = -1;                                         // 进程ID
        memset(&(proc->name), 0, PROC_NAME_LEN);                // 进程名
        proc->state = PROC_UNINIT;                              // 进程状态
        proc->runs = 0;                                         // 进程时间片
        proc->need_resched = 0;                                 // 进程是否能被调度
        proc->flags = 0;                                        // 标志位
        proc->kstack = 0;                                       // 进程所使用的内存栈地址
        proc->cr3 = boot_cr3;                                   // 将页目录表地址设为内核页目录表基址
        proc->mm = NULL;                                        // 进程所用的虚拟内存
```

```
        memset(&(proc->context), 0, sizeof(struct context));  // 进程的上下文
        proc->tf = NULL;                                      // 中断帧指针
        proc->parent = NULL;                                  // 该进程的父进程
        proc->wait_state = 0;                                 // Lab5：等待状态的标志位
        proc->cptr = NULL;                                    // Lab5：该进程的子进程
        proc->yptr = NULL;                                    // Lab5：该进程的弟进程
        proc->optr = NULL;                                    // Lab5：该进程的兄进程
    }
    return proc;
}
```

do_fork 函数源码

写于：**kern/process/proc.c**

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's proc->tf
 */
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;
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm according clone_flags
     *                 if clone_flags & CLONE_VM, then "share" ; else "duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
```

```
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid

    //LAB5 YOUR CODE : (update LAB4 steps)
   /* Some Functions
    *    set_links:  set the relation links of process.  ALSO SEE: remove_links:  lean the relation links of process
    *    -------------------
    *    update step 1: set child proc's parent to current process, make sure current process's wait_state is 0
    *    update step 5: insert proc_struct into hash_list && proc_list, set the relation links of process
    */

fork_out:
    return ret;

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

set_links 函数

写于：**kern/process/proc.c**

```
// set_links - set the relation links of process
static void
set_links(struct proc_struct *proc) {
    list_add(&proc_list, &(proc->list_link));         // 将 proc 进程添加到进程列表里
    proc->yptr = NULL;                                // 初始化 proc 进程的弟进程
    if ((proc->optr = proc->parent->cptr) != NULL) {  // 如果 proc 进程的兄进程等于 proc 进程的父进程的子进程
        proc->optr->yptr = proc;                      // 就设置 proc 进程的兄进程的弟进程为 proc 进程
    }
    proc->parent->cptr = proc;                        // 使 proc 进程的父进程的弟进程为 proc 进程
    nr_process ++;                                    // 进程控制块总数加一
}
```

所以这个进程的主要目的就是将 proc 进程控制块放到进程列表里

do_fork 函数答案

写于：**kern/process/proc.c**

因为 `set_links 函数` 里有 `nr_process ++`，所以 `do_fork 函数` 里的 `nr_process ++` 就可以删掉了

```c
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;                          // 设置返回值为 -5
    struct proc_struct *proc;                           // 定义 proc_struct 结构体指针变量 proc（子进程）
    if (nr_process >= MAX_PROCESS) {                    // 进程总数（全局变量）如果 >= 0x1000
        goto fork_out;                                  // 就跳转到 fork_out 地址，此时返回值是 -5
    }
    ret = -E_NO_MEM;                                    // 更换返回值为 -4
    if ((proc = alloc_proc()) == NULL) {                // 申请一个 proc_struct 结构体，作为子进程的进程控制块
        goto fork_out;                                  // 若是申请失败就跳转到 fork_out 地址，此时返回值是 -4
    }

    proc->parent = current;                             // 设置当前进程为上面新申请的子进程的父进程
    assert(current->wait_state == 0);                   // Lab5：设置当前进程为等待进程

    if (setup_kstack(proc) != 0) {                      // 分配并初始化子进程的内核栈，返回值是 0 则成功分配
        goto bad_fork_cleanup_proc;                     // 创建失败，即内存不足，则跳转到 bad_fork_cleanup_proc
    }
    if (copy_mm(clone_flags, proc) != 0) {              // 将当前进程的内存信息复制到子进程
        goto bad_fork_cleanup_kstack;                   // 出错就跳转到 bad_fork_cleanup_kstack
    }
    copy_thread(proc, stack, tf);                       // 复制当前进程的中断帧和上下文到子进程中

    bool intr_flag;
    local_intr_save(intr_flag);                         // 禁止中断发生，保护代码运行
    {
        proc->pid = get_pid();                          // 获取子进程的 pid
        hash_proc(proc);                                // 将子进程添加进哈希列表
        set_links(proc);                                // Lab5：将子进程的进程控制块放到进程列表里
    }
    local_intr_restore(intr_flag);                      // 如果 intr_flag 不为 0，则允许中断发生

    wakeup_proc(proc);                                  // 唤醒该进程

    ret = proc->pid;                                    // 更新返回值为子进程的 pid
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);                                   // 释放子进程的内核栈
```

```
bad_fork_cleanup_proc:
    kfree(proc);                                    // 释放申请的物理页，跳转到fork_out
    goto fork_out;
}
```

idt_init 函数源码

写于：**kern/trap/trap.c**

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */
void
idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
      *     All ISR's entry addrs are stored in __vectors. where is uintptr_t __vectors[] ?
      *     __vectors[] is in kern/trap/vector.S which is produced by tools/vector.c
      *     (try "make" command in lab1, then you will find vector.S in kern/trap DIR)
      *     You can use  "extern uintptr_t __vectors[];" to define this extern variable which will be used later.
      * (2) Now you should setup the entries of ISR in Interrupt Description Table (IDT).
      *     Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup each item of IDT
      * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using 'lidt' instruction.
      *     You don't know the meaning of this instruction? just google it! and check the libs/x86.h to know more.
      *     Notice: the argument of lidt is idt_pd. try to find it!
      */
     /* LAB5 YOUR CODE */
     //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
     //so you should setup the syscall interrupt gate in here
}
```

idt_init 函数答案

写于：**kern/trap/trap.c**

```c
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); // Lab5
    lidt(&idt_pd);
}
```

trap_dispatch 函数源码

写于：**kern/trap/trap.c**

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

    switch (tf->tf_trapno) {
    case T_PGFLT:  // page fault
        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
    LAB3 : If some page replacement algorithm(such as CLOCK PRA) need tick to change the priority of pages,
    then you can add code here.
#endif
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
```

```
        /* LAB5 YOUR CODE */
        /* you should upate you lab1 code (just add ONE or TWO lines of code):
         *    Every TICK_NUM cycle, you should set current process's current->need_resched = 1
         */

        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;
    // LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        print_trapframe(tf);
        if (current != NULL) {
            cprintf("unhandled trap.\n");
            do_exit(-E_KILLED);
        }
        // in kernel, it must be a mistake
        panic("unexpected trap in kernel.\n");

    }
}
```

trap_dispatch 函数答案

写于：**kern/trap/trap.c**

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

    switch (tf->tf_trapno) {
    case T_PGFLT:  // page fault
        if ((ret = pgfault_handler(tf)) != 0) {
            print_trapframe(tf);
            panic("handle pgfault failed. %e\n", ret);
        }
        break;
    case IRQ_OFFSET + IRQ_TIMER:
        ticks ++;
        if (ticks % TICK_NUM == 0) {    // 当时间片用完
            assert(current != NULL);    // Lab5: 判断 current 变量是否有值
```

```
                current->need_resched = 1;  // Lab5：设置当前进程需要被调度
            }
            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;
        // LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
        case T_SWITCH_TOU:
        case T_SWITCH_TOK:
            panic("T_SWITCH_** ??\n");
            break;
        case IRQ_OFFSET + IRQ_IDE1:
        case IRQ_OFFSET + IRQ_IDE2:
            /* do nothing */
            break;
        default:
            // in kernel, it must be a mistake
            if ((tf->tf_cs & 3) == 0) {
                print_trapframe(tf);
                panic("unexpected trap in kernel.\n");
            }
        }
}
```

练习1: 加载应用程序并执行

mm_struct 结构体

```
// the control struct for a set of vma using the same PDT
struct mm_struct {                   // 描述一个进程的虚拟地址空间，每个进程的 pcb 中会有一个指针指向本结构体
    list_entry_t mmap_list;          // 链接同一页目录表的虚拟内存空间中双向链表的头节点
    struct vma_struct *mmap_cache;   // 当前正在使用的虚拟内存空间
    pde_t *pgdir;                    // mm_struct 所维护的页表地址(用来找 PTE)
    int map_count;                   // 虚拟内存块的数目
    void *sm_priv;                   // 记录访问情况链表头地址(用于置换算法)
    int mm_count;                    // 共享 mm 的进程数，或者说对使用该结构体的某一结构体变量的引用次数
    lock_t mm_lock;                  // 互斥锁，用于使用 dup_mmap 复制 mm
};
```

这个结构体新加了 `mm_count` 和 `mm_lock` 变量

load_icode 函数源码

写于：**kern/process/proc.c**

```
/* load_icode - load the content of binary program(ELF format) as the new content of current process
```

```
 * @binary:  the memory addr of the content of binary program
 * @size:  the size of the content of binary program
 */
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
     */
    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;
}
```

mm_create 函数

写于：**kern/mm/vmm.c**

```
// mm_create -  alloc a mm_struct & initialize it.
struct mm_struct *
mm_create(void) {
    // 申请一块内存地址空间
    struct mm_struct *mm = kmalloc(sizeof(struct mm_struct));

    if (mm != NULL) {                               // 如果申请成功，返回了内存地址空间的地址
        list_init(&(mm->mmap_list));           // 将该内存地址空间中的链表节点设置为头节点
        mm->mmap_cache = NULL;                 // 初始化虚拟内存空间
```

```
        mm->pgdir = NULL;                      // 初始化 mm_struct 所维护的页表地址
        mm->map_count = 0;                     // 初始化虚拟内存块的数目

        if (swap_init_ok) swap_init_mm(mm);   // 如果可以换入页面，那么就初始化用于置换算法的链表
        else mm->sm_priv = NULL;               // 否则就将记录访问情况的链表头地址设置为 NULL

        set_mm_count(mm, 0);                   // 设置该虚拟内存块的数目为 0
        lock_init(&(mm->mm_lock));             // 初始化互斥锁
    }
    return mm;                                 // 返回 mm 内存地址空间的地址
}
```

该函数的作用是申请并初始化一块内存地址空间

swap_init_mm 函数

写于：**kern/mm/swap.c**

```
int
swap_init_mm(struct mm_struct *mm)
{
     return sm->init_mm(mm);
}
```

- **swap_manager_fifo 结构体变量**

  写于：**kern/mm/swap_fifo.c**

  ```
  struct swap_manager swap_manager_fifo =
  {
       .name            = "fifo swap manager",
       .init            = &_fifo_init,
       .init_mm         = &_fifo_init_mm,
       .tick_event      = &_fifo_tick_event,
       .map_swappable   = &_fifo_map_swappable,
       .set_unswappable = &_fifo_set_unswappable,
       .swap_out_victim = &_fifo_swap_out_victim,
       .check_swap      = &_fifo_check_swap,
  };
  ```

  所以 **init_mm 函数**就是 **_fifo_init_mm 函数**

_fifo_init_mm 函数

写于：**kern/mm/swap_fifo.c**

```
/*
 * (2) _fifo_init_mm: init pra_list_head and let  mm->sm_priv point to the addr of pra_list_head.
 *              Now, From the memory control struct mm_struct, we can access FIFO PRA
 */
static int
_fifo_init_mm(struct mm_struct *mm)
{
     list_init(&pra_list_head);
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

将 pra_list_head 的地址作为链表的头地址

将记录访问情况链表的头地址设置为 pra_list_head 的地址

set_mm_count 函数

写于：**kern/mm/swap_fifo.c**

```
static inline void
set_mm_count(struct mm_struct *mm, int val) {
    mm->mm_count = val;
}
```

设置共享 mm 的进程数为 val

lock_init 函数

写于：**kern/sync/sync.h**

```
static inline void
lock_init(lock_t *lock) {
    *lock = 0;
}
```

初始化互斥锁，将其设置为 0

setup_pgdir 函数

写于：**kern/proc/proc.c**

```
// setup_pgdir - alloc one page as PDT
static int
setup_pgdir(struct mm_struct *mm) {
    struct Page *page;
    if ((page = alloc_page()) == NULL) {  // 申请一个物理页
        return -E_NO_MEM;                 // 申请失败就返回 -4
    }
    pde_t *pgdir = page2kva(page);        // 通过该物理页获取其内核虚拟地址（页目录表）
    memcpy(pgdir, boot_pgdir, PGSIZE);    // 将启动时页目录的虚拟地址中的内容复制到 pgdir 中
```

```
    // 将 pgdir 中对应关于 VPT 的元素的内容设置为 pgdir 的地址转为物理地址后再加上标志位的值
    pgdir[PDX(VPT)] = PADDR(pgdir) | PTE_P | PTE_W;
    mm->pgdir = pgdir;                    // 设置 mm_struct 所维护的页表地址为 pgdir 的地址（虚拟地址）
    return 0;
}
```

该函数的作用就是创建一个页目录表，并准备好相应的初始化工作

VPT 宏

写于：**kern/mm/memlayout.c**

```
/* *
 * Virtual page table. Entry PDX[VPT] in the PD (Page Directory) contains
 * a pointer to the page directory itself, thereby turning the PD into a page
 * table, which maps all the PTEs (Page Table Entry) containing the page mappings
 * for the entire virtual address space into that 4 Meg region starting at VPT.
 * */
#define VPT                 0xFAC00000
```

虚拟页表的地址

ELF_MAGIC 宏

写于：**libs/elf.h**

```
#define ELF_MAGIC    0x464C457FU         // "\x7FELF" in little endian
```

E_INVAL_ELF 宏

写于：**libs/error.h**

```
#define E_INVAL_ELF         8    // Invalid elf file
```

ELF_PT_LOAD 宏

写于：**libs/elf.h**

```
/* values for Proghdr::p_type */
#define ELF_PT_LOAD                     1
```

表明该段可被加载

ELF_PF_X 宏

写于：**libs/elf.h**

```
/* flag bits for Proghdr::p_flags */
#define ELF_PF_X                        1
```

表明有执行权限

ELF_PF_W 宏

写于：**libs/elf.h**

```
#define ELF_PF_W                    2
```

ELF_PF_R 宏

写于：**libs/elf.h**

```
#define ELF_PF_R                    4
```

mm_map 函数

写于：**kern/mm/vmm.c**

```
int
mm_map(struct mm_struct *mm, uintptr_t addr, size_t len, uint32_t vm_flags,
       struct vma_struct **vma_store) {
    // 规定页的起始地址和结束地址
    uintptr_t start = ROUNDDOWN(addr, PGSIZE), end = ROUNDUP(addr + len, PGSIZE);
    if (!USER_ACCESS(start, end)) {  // 判断当前地址是否能分配给用户使用
        return -E_INVAL;             // 返回 -3
    }

    assert(mm != NULL);              // 要求传入的 mm 必须有地址，而不是申请失败后的空值

    int ret = -E_INVAL;              // 设置返回值为 -3

    struct vma_struct *vma;
    // 更新 mm 里的 vma 并取得它的值，判断页的结束地址是否大于 vma 的起始地址
    if ((vma = find_vma(mm, start)) != NULL && end > vma->vm_start) {
        goto out;                    // 大于就有问题了，直接跳到 out
    }
    ret = -E_NO_MEM;                 // 更新返回值为 -4

    // 初始化 vma，其实经过上面的判断 vma 已经不可能是 NULL 了
    if ((vma = vma_create(start, end, vm_flags)) == NULL) {
        goto out;                    // 跳转到 out
    }
    insert_vma_struct(mm, vma);      // 将该 vma 对应的链表插入到 vma->list_link 链表中
    if (vma_store != NULL) {         // vma_store 如果不为空
        *vma_store = vma;            // 更新 *vma_store 为 vma
    }
    ret = 0;                         // 更新返回值为 0

out:
    return ret;
}
```

该函数的大致意思就是找到一个合法的 vma 内存，并更新 `mm->mmap_cache` 为这个新 vma

ROUNDUP 宏

写于：**libs/defs.h**

```
/* Round up to the nearest multiple of n */
#define ROUNDUP(a, n) ({                                        \
            size_t __n = (size_t)(n);                           \
            (typeof(a))(ROUNDDOWN((size_t)(a) + __n - 1, __n));     \
        })
```

注释意思：四舍五入操作，四舍五入到 n + 1 的倍数

其实应该只要 n 不是 0，都可以进行对于 a 的倍数的四舍五入

只是在这个 ucore 的代码里用的都是 2 的倍数（都用的 PGSIZE == 4096）

如果 a 是 15 的话，`ROUNDUP(15, 4) == 16`

USER_ACCESS 宏

写于：**kern/mm/memlayout.h**

```
#define USER_ACCESS(start, end)                             \
(USERBASE <= (start) && (start) < (end) && (end) <= USERTOP)
```

判断当前地址是否能分配给用户使用

USERBASE 宏

写于：**kern/mm/memlayout.h**

```
#define USERBASE            0x00200000
```

用户地址的起始地址

USERTOP 宏

写于：**kern/mm/memlayout.h**

```
#define USERTOP             0xB0000000
```

用户地址的结束地址

find_vma 函数

写于：**kern/mm/vmm.c**

```
// find_vma - find a vma  (vma->vm_start <= addr <= vma_vm_end)
struct vma_struct *
find_vma(struct mm_struct *mm, uintptr_t addr) {
    struct vma_struct *vma = NULL;
    if (mm != NULL) {
        vma = mm->mmap_cache;                          // 设置 vma 为该内存管理空间的虚拟内存空间
        // 如果 vma 不为空且 addr 属于合法的地址
        if (!(vma != NULL && vma->vm_start <= addr && vma->vm_end > addr)) {
                bool found = 0;                        // 初始化标记位为 0
```

```
                list_entry_t *list = &(mm->mmap_list), *le = list;    // 获取头节点的信息
                while ((le = list_next(le)) != list) {  // 遍历双向链表
                    vma = le2vma(le, list_link);        // 依靠列表元素得到对应的 vma_struct 结构体地址
                    if (vma->vm_start<=addr && addr < vma->vm_end) {  // 如果 addr 为合法地址
                        found = 1;                      // 改变标记位为 1，确认找到了符合的地址
                        break;                          // 退出循环
                    }
                }
                if (!found) {                           // 标记位为 0，说明没找到合法地址
                    vma = NULL;                         // 因为没找到合法地址，所以 vma 为 NULL
                }
        }
        if (vma != NULL) {                              // 如果 vma 找到了合法地址
            mm->mmap_cache = vma;                       // 更新该内存管理空间的虚拟内存空间为 vma
        }
    }
    return vma;                                         // 返回 vma
}
```

貌似就是去虚拟内存空间里找一块合法的空间来替代原来使用的空间

vma_create 函数

写于：**kern/mm/vmm.c**

```
// vma_create - alloc a vma_struct & initialize it. (addr range: vm_start~vm_end)
struct vma_struct *
vma_create(uintptr_t vm_start, uintptr_t vm_end, uint32_t vm_flags) {
    struct vma_struct *vma = kmalloc(sizeof(struct vma_struct));

    if (vma != NULL) {
        vma->vm_start = vm_start;
        vma->vm_end = vm_end;
        vma->vm_flags = vm_flags;
    }
    return vma;
}
```

用来初始化 vma 的函数

insert_vma_struct 函数

写于：**kern/mm/vmm.c**

```
// insert_vma_struct -insert vma in mm's list link
void
insert_vma_struct(struct mm_struct *mm, struct vma_struct *vma) {
```

```
    assert(vma->vm_start < vma->vm_end);                    // 规定起始地址必须小于结束地址
    list_entry_t *list = &(mm->mmap_list);                  // 获得虚拟内存空间的头节点
    list_entry_t *le_prev = list, *le_next;                 // 初始化前驱节点和后继节点

        list_entry_t *le = list;                            // 初始化链表节点元素
        while ((le = list_next(le)) != list) {              // 遍历双向链表
            // 依靠列表元素得到对应的 vma_struct 结构体地址
            struct vma_struct *mmap_prev = le2vma(le, list_link);
            if (mmap_prev->vm_start > vma->vm_start) {      // 如果该新虚拟内存空间的起始地址大于原始的起始地址
                break;                                      // 退出循环
            }
            le_prev = le;                                   // 设置上一个列表元素为 le
        }

    le_next = list_next(le_prev);                           // le 的下一个元素为 le_next

    /* check overlap */
    if (le_prev != list) {                                  // 如果 le 不等于 list
        check_vma_overlap(le2vma(le_prev, list_link), vma); // 检查是否出现重叠的内存区域
    }
    if (le_next != list) {                                  // 如果 le_next 不等于 list
        check_vma_overlap(vma, le2vma(le_next, list_link)); // 检查是否出现重叠的内存区域
    }

    vma->vm_mm = mm;                                        // 更新虚拟内存空间属于的内存管理区域
    list_add_after(le_prev, &(vma->list_link));             // 将 le 元素添加到 vma->list_link 链表中

    mm->map_count ++;                                       // 虚拟内存块的数目自增 1
}
```

这个函数的作用就是将该 vma 对应的链表插入到 vma->list_link 链表中

check_vma_overlap 函数

写于：**kern/mm/vmm.c**

```
// check_vma_overlap - check if vma1 overlaps vma2 ?
static inline void
check_vma_overlap(struct vma_struct *prev, struct vma_struct *next) {
    assert(prev->vm_start < prev->vm_end);
    assert(prev->vm_end <= next->vm_start);
    assert(next->vm_start < next->vm_end);
}
```

检查分配的内存区域是否出现重叠情况，出现就报错中止程序

正常情况下，起始地址都要小于相应的结束地址，且 prev 的结束地址要小于 next 的起始地址

load_icode 函数答案

写于：**kern/process/proc.c**

```
static int
load_icode(unsigned char *binary, size_t size) {
    if (current->mm != NULL) {                                  // 判断当前进程的内存地址空间是否为空
        panic("load_icode: current->mm must be empty.\n");  // 不为空就报错退出程序
    }

    int ret = -E_NO_MEM;                                        // 设置返回值为-4
    struct mm_struct *mm;                                       // 创建内存地址空间变量
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL) {                           // 申请并初始化一块内存地址空间
        goto bad_mm;                                            // 申请失败就跳到 bad_mm
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0) {                                 // 创建一个页目录表
        goto bad_pgdir_cleanup_mm;                              // 创建失败就跳到 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) {                            // 检查程序文件头是否合法
        ret = -E_INVAL_ELF;                                     // 更改返回值为-8
        goto bad_elf_cleanup_pgdir;                             // 跳转到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) {                        // 如果该段不是可加载的段
            continue ;                                          // 就跳过
        }
```

```
        if (ph->p_filesz > ph->p_memsz) {                      // 如果段的大小大于段的内存大小
            ret = -E_INVAL_ELF;                                 // 更改返回值为-8
            goto bad_cleanup_mmap;                              // 跳转到bad_cleanup_mmap
        }
        if (ph->p_filesz == 0) {                               // 如果段的大小等于 0
            continue ;                                          // 就跳过
        }
    //(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;     // 如果文件有执行权限，将虚拟内存空间页设置为有执行权限
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;    // 如果文件有可写权限，将虚拟内存空间页设置为有可写权限
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;     // 如果文件有可读权限，将虚拟内存空间页设置为有可读权限
        if (vm_flags & VM_WRITE) perm |= PTE_W;               // 如果虚拟内存空间页有可写权限，将页目录表设置为有可写权限
        // 找到一个合法的 vma 内存，并更新 mm->mmap_cache 为这个新 vma
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;                              // 没找到合法的vma 就跳转到 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;                                        // 更新返回值为-4

     //(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) {                                   // 如果 start 小于 end 就一直遍历
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) { // 申请一个页表
                goto bad_cleanup_mmap;                          // 跳转到bad_cleanup_mmap
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {                                     // 如果 end 小于 la
                size -= la - end;                               // 用 size 减去(la + end)
            }
            memcpy(page2kva(page) + off, from, size);          // 将 from 的内容复制到 page 对应的虚拟地址中
            start += size, from += size;                        // 更新 start 和 from 的地址
        }

      //(3.6.2) build BSS section of binary program
```

```c
        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
    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;
}
```

分析不下去了，反正这部分就差个布置用户态的栈信息的代码，加上就好了

练习2: 父进程复制自己的内存空间给子进程

copy_range 函数源码

写于：**kern/mm/pmm.c**

```
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 start
        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 start. 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 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
             */
            assert(ret == 0);
        }
        start += PGSIZE;
    } while(start != 0 && start < end);
    return 0;
}
```

copy_range 函数答案

写于：**kern/mm/pmm.c**

```
int
copy_range(pde_t* to, pde_t* from, uintptr_t start, uintptr_t end, bool share){
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);     // 确保 start 和 end 的大小都是页对齐的
    assert(USER_ACCESS(start, end));                      // 确保 start 和 end 是在用户态空间范围内的
    // copy content by page unit.
    do{
        //call get_pte to find process A's pte according to the addr start
        pte_t* ptep = get_pte(from, start, 0), * nptep;   // 获得一个有存在位标志的 ptep
        if(ptep == NULL){                                 // 如果没有成功获得有存在位标志的 ptep
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);    // 将 start 设置为 start + PTSIZE 的大小
            continue;                                     // 继续下一个循环
        }
    //call get_pte to find process B's pte according to the addr start. If pte is NULL, just alloc a PT
        if(*ptep & PTE_P){                                // 如果得到的 ptep 有存在位标志
            if((nptep = get_pte(to, start, 1)) == NULL){  // 新分配一个页表
                return -E_NO_MEM;                         // 分配失败就返回 -4
            }
            uint32_t perm = (*ptep & PTE_USER);           // 设置用户标志位(用户位、可写位、存在位)
            //get page from ptep
            struct Page* page = pte2page(*ptep);          // 从页表得到对应的物理页
            // alloc a page for process B
            struct Page* npage = alloc_page();            // 申请一块物理页
```

```
            assert(page != NULL);                       // 没得到对应的物理页，中止程序
            assert(npage != NULL);                      // 申请物理页不成功，中止程序
            int ret = 0;                                // 设置返回值为 0

            void* kva_src = page2kva(page);    // Lab5：得到 page 的虚拟地址
            void* kva_dst = page2kva(npage);   // Lab5：得到 npage 的虚拟地址
            memcpy(kva_dst, kva_src, PGSIZE);  // Lab5：将 page 虚拟地址里的内容复制到 npage 的虚拟地址里
            ret = page_insert(to, npage, start, perm); // Lab5：释放原先的二级页表映射并建立和 npage 映射的页表
            assert(ret == 0);                           // 返回值为 0 代表函数运行正常，返回 -4 就中止程序
        }
        start += PGSIZE;                                // 将起始地址加上一个页的大小
    } while(start != 0 && start < end);                 // 如果 start 不大于等于 end 且不等于 0 就一直循环
    return 0;
}
```

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

do_fork 函数分析

写于：**kern/process/proc.c**

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC; // 设置返回值为 -5
    struct proc_struct *proc;  // 定义 proc_struct 结构体指针变量 proc（子进程）
    if (nr_process >= MAX_PROCESS) {  // 进程总数（全局变量）如果 >= 0x1000
        goto fork_out;         // 就跳转到 fork_out 地址，此时返回值是 -5
    }
    ret = -E_NO_MEM;           // 更换返回值为 -4
    if ((proc = alloc_proc()) == NULL) {  // 申请一个 proc_struct 结构体
        goto fork_out;  // 若是申请失败就跳转到 fork_out 地址，此时返回值是 -4
    }

    proc->parent = current;    // 设置当前进程的父进程地址

    if (setup_kstack(proc) != 0) {  // 分配并初始化内核栈，返回值是 0 则成功分配
        // 创建失败，即内存不足，则跳转到 bad_fork_cleanup_proc
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {  // 将父进程的内存信息复制到子进程
        // 出错就跳转到 bad_fork_cleanup_kstack
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);    // 复制父进程的中断帧和上下文

    bool intr_flag;
    local_intr_save(intr_flag);      // 禁止中断发生，保护代码运行
    {
```

```c
        proc->pid = get_pid();      // 获取该程序的 pid
        hash_proc(proc);            // 将该节点添加进哈希列表
        list_add(&proc_list, &(proc->list_link));  // 将该节点添加进双向链表
        nr_process ++;              // 记录总进程数的变量自增 1
    }
    local_intr_restore(intr_flag);  // 如果 intr_flag 不为 0，则允许中断发生

    wakeup_proc(proc);              // 唤醒该进程

    ret = proc->pid;                // 更新返回值为该进程的 pid
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);               // 释放内核栈
bad_fork_cleanup_proc:
    kfree(proc);                    // 释放申请的物理页，跳转到 fork_out
    goto fork_out;
}
```

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

do_exit 函数分析

写于：**kern/process/proc.c**

```c
// do_exit - called by sys_exit
//   1. call exit_mmap & put_pgdir & mm_destroy to free the almost all memory space of process
//   2. set process' state as PROC_ZOMBIE, then call wakeup_proc(parent) to ask parent reclaim itself.
//   3. call scheduler to switch to other process
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }

    struct mm_struct *mm = current->mm;
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
```

```c
        current->mm = NULL;
    }
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;

    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);

    schedule();
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

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

do_execve 函数分析

写于：**kern/process/proc.c**

```c
// do_execve - call exit_mmap(mm)&put_pgdir(mm) to reclaim memory space of current process
//           - call load_icode to setup new memory space accroding binary prog.
int
do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) {
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) {
        len = PROC_NAME_LEN;
    }

    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);

    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

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

do_wait 函数分析

写于：**kern/process/proc.c**

```c
// do_wait - wait one OR any children with PROC_ZOMBIE state, and free memory space of kernel stack
```

```c
//          - proc struct of this child.
// NOTE: only after do_wait function, all resources of the child proces are free.
int
do_wait(int pid, int *code_store) {
    struct mm_struct *mm = current->mm;
    if (code_store != NULL) {
        if (!user_mem_check(mm, (uintptr_t)code_store, sizeof(int), 1)) {
            return -E_INVAL;
        }
    }

    struct proc_struct *proc;
    bool intr_flag, haskid;
repeat:
    haskid = 0;
    if (pid != 0) {
        proc = find_proc(pid);
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    else {
        proc = current->cptr;
        for (; proc != NULL; proc = proc->optr) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    if (haskid) {
        current->state = PROC_SLEEPING;
        current->wait_state = WT_CHILD;
        schedule();
        if (current->flags & PF_EXITING) {
            do_exit(-E_KILLED);
        }
        goto repeat;
    }
    return -E_BAD_PROC;

found:
    if (proc == idleproc || proc == initproc) {
        panic("wait idleproc or initproc.\n");
    }
    if (code_store != NULL) {
        *code_store = proc->exit_code;
    }
    local_intr_save(intr_flag);
    {
        unhash_proc(proc);
        remove_links(proc);
    }
    local_intr_restore(intr_flag);
```

```
    put_kstack(proc);
    kfree(proc);
    return 0;
}
```

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。

扩展练习 Challenge： 实现 Copy on Write （COW）机制

咕咕咕