操作系统———— Lab5

> **学号：2111033**
> **姓名：艾明旭**
> **专业：信息安全**

> 实验目的
>
> • 了解第一个用户进程创建过程
>
> • 了解系统调用框架的实现机制
>
> • 了解 ucore 如何实现系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来进行进程管理

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。你需要补充 load_icode 的第 6 步，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。

> 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:

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。

```
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 | PTE_V;
        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;
        // modify the perm bits here for RISC-V
        if (vm_flags & VM_READ) perm |= PTE_R;
```

```
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        if (vm_flags & VM_EXEC) perm |= PTE_X;
        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;
    // Keep sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    /* LAB5:EXERCISE1 YOUR CODE
     * should set tf->gpr.sp, tf->epc, tf->status
     * NOTICE: If we set trapframe correctly, then the user level process can return
to USER MODE from kernel. So
     *          tf->gpr.sp should be user stack top (the value of sp)
     *          tf->epc should be entry point of user program (the value of sepc)
     *          tf->status should be appropriate for user program (the value of
sstatus)
     *          hint: check meaning of SPP, SPIE in SSTATUS, use them by SSTATUS_SPP,
SSTATUS_SPIE(defined in risv.h)
     */
    tf->gpr.sp = USTACKTOP;
    tf->epc = elf->e_entry;
    tf->status = read_csr(sstatus) & ~SSTATUS_SPP & ~SSTATUS_SPIE;
    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;
}
```

代码实现思路

- 设置中断帧中通用寄存器sp的值为用户栈的栈顶。
- 设置epc寄存器的值为用户需要运行的可执行文件的入口点。
- 设置status为用户状态，即在原有状态的基础上，将SPP设置为0（必须设置），即用户模式，并允许中断

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

- `do_execve`函数部分执行用户进程的创建工作
- `load_icode`函数来给用户进程建立一个能够让用户进程正常运行的用户程序
- 用户进程的用户环境已经搭建完毕。此时initproc将按产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”后，将切换到用户进程的第一条语句位置_start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。
>
> 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。

- `do_fork`：进行内存复制时，如在copy_range中，不进行复制，而是将父进程和子进程的虚拟页映射上同一个物理页，然后将父进程的PDE直接赋值给子进程，将PTE_W置为0(不可写入)，但应用程序试图写入某个共享页就会产生页访问异常

```
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_V) {
            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
             */
            void *src_kvaddr = page2kva(page);
            void *dst_kvaddr = page2kva(npage);
            memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
            ret = page_insert(to, npage, start, perm);

            assert(ret == 0);
```

```
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

- 实现思路
  - 找到要复制的某一页pade的内核虚拟地址
  - 找到要被复制到的某一页npage的内核虚拟地址
  - 调用memcpy从src_kvaddr向dst_kvaddr进行内存复制，大小为PGSIZE
  - 将新复制出的npage插入进页表指定位置
- Copy on write概要设计
  - 该机制的主要思想为使得进程执行fork系统调用进行复制的时候，父进程不会简单地将整个内存中的内容复制给子进程，而是暂时共享相同的物理内存页；而当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，然后在自己的内存页中进行修改；
  - 根据上述分析，主要对实验框架的修改应当主要有两个部分，一个部分在于进行fork操作的时候不直接复制内存，另外一个处理在于出现了内存页访问异常的时候，会将共享的内存页复制一份，然后在新的内存页进行修改，具体如下。
  - 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，这样的话如果应用程序试图写某一个共享页就会产生页访问异常，从而可以将控制权交给操作系统进行处理；
  - 在page_fault的中断处理程序部分，增加对这种异常的处理，处理方法为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。
- `page_fault`：在page_fault的ISR部分，增加对这种异常的处理，处理方法为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）**

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
   - `fork`：不会影响当前进程的执行状态，但是会将子进程的状态标记为RUNNALB，使其可以在后续的调度中运行
   - `exit`：不会影响当前进程的执行状态，但是会修改当前进程中执行的程序
   - `execve`：系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为SLEEPING态，等待执行了exit的子进程将其唤醒
   - `wait`：将当前进程的状态修改为ZOMBIE态，并且会将父进程唤醒，对该进程的资源进行回收

问题1：fork/exec/wait/exit

调用过程为

1. SYS_fork调用syscall（用户态）

```
// user/libs/syscall.c
int sys_fork(void) { return syscall(SYS_fork); }
```

2. syscall内联汇编进行ecall环境调用，这将产生一个trap, 进入S mode进行异常处理。（用户态）

```c
// user/libs/syscall.c
static inline int syscall(int num, ...) {
  //va_list, va_start, va_arg都是C语言处理参数个数不定的函数的宏
  //在stdarg.h里定义
  va_list ap; //ap: 参数列表(此时未初始化)
  va_start(ap, num); //初始化参数列表, 从num开始
  //First, va_start initializes the list of variable arguments as a va_list.
  uint64_t a[MAX_ARGS];
  int i, ret;
  for (i = 0; i < MAX_ARGS; i ++) { //把参数依次取出
      /*Subsequent executions of va_arg yield the values of the additional
arguments
      in the same order as passed to the function.*/
      a[i] = va_arg(ap, uint64_t);
  }
  va_end(ap); //Finally, va_end shall be executed before the function returns.
  asm volatile (
      "ld a0, %1\n"
      "ld a1, %2\n"
      "ld a2, %3\n"
      "ld a3, %4\n"
      "ld a4, %5\n"
      "ld a5, %6\n"
      "ecall\n"
      "sd a0, %0"
      : "=m" (ret)
      : "m"(num), "m"(a[0]), "m"(a[1]), "m"(a[2]), "m"(a[3]), "m"(a[4])
      :"memory");
  //num存到a0寄存器， a[0]存到a1寄存器
  //ecall的返回值存到ret，这也就是内核态执行结果是如何返回给用户程序的
  return ret;
}
```

3. trap.c转发这个系统调用，调用syscall函数（和用户态定义的那个不一样）（内核态）

```
// kern/trap/trap.c
void exception_handler(struct trapframe *tf) {
  int ret;
  switch (tf->cause) { //通过中断帧里 scause寄存器的数值，判断出当前是来自USER_ECALL
的异常
      case CAUSE_USER_ECALL:
          //cprintf("Environment call from U-mode\n");
          tf->epc += 4;
          //sepc寄存器是产生异常的指令的位置，在异常处理结束后，会回到sepc的位置继续执
行
          //对于ecall，我们希望sepc寄存器要指向产生异常的指令(ecall)的下一条指令
          //否则就会回到ecall执行再执行一次ecall，无限循环
          syscall();// 进行系统调用处理
          break;
      /*other cases .... */
  }
}
```

4. syscall根据不同的编号，依据函数指针的数组，再转发到相应函数执行（内核态）

```
// kern/syscall/syscall.c
void syscall(void) {
  struct trapframe *tf = current->tf;
  uint64_t arg[5];
  int num = tf->gpr.a0;//a0寄存器保存了系统调用编号
  if (num >= 0 && num < NUM_SYSCALLS) {//防止syscalls[num]下标越界
      if (syscalls[num] != NULL) {
          arg[0] = tf->gpr.a1;
          arg[1] = tf->gpr.a2;
          arg[2] = tf->gpr.a3;
          arg[3] = tf->gpr.a4;
          arg[4] = tf->gpr.a5;
          tf->gpr.a0 = syscalls[num](arg);
          //把寄存器里的参数取出来，转发给系统调用编号对应的函数进行处理
          return ;
      }
  }
  //如果执行到这里，说明传入的系统调用编号还没有被实现，就崩掉了。
  print_trapframe(tf);
  panic("undefined syscall %d, pid = %d, name = %s.\n",
          num, current->pid, current->name);
}
//这里定义了函数指针的数组syscalls，把每个系统调用编号的下标上初始化为对应的函数指针
static int (*syscalls[])(uint64_t arg[]) = {
  [SYS_exit]              sys_exit,
  [SYS_fork]              sys_fork,
  [SYS_wait]              sys_wait,
  [SYS_exec]              sys_exec,
  [SYS_yield]             sys_yield,
  [SYS_kill]              sys_kill,
  [SYS_getpid]            sys_getpid,
  [SYS_putc]              sys_putc,
};
```

5. 相应函数（形如sys_exit、sys_fork），会再调用相应函数（形如do_exit()、do_execve()）。（内核态）

```c
// kern/syscall/syscall.c
static int sys_exit(uint64_t arg[]) {
  int error_code = (int)arg[0];
  return do_exit(error_code);
}
static int sys_fork(uint64_t arg[]) {
  struct trapframe *tf = current->tf;
  uintptr_t stack = tf->gpr.sp;
  return do_fork(0, stack, tf);
}
static int sys_wait(uint64_t arg[]) {
  int pid = (int)arg[0];
  int *store = (int *)arg[1];
  return do_wait(pid, store);
}
static int sys_exec(uint64_t arg[]) {
  const char *name = (const char *)arg[0];
  size_t len = (size_t)arg[1];
  unsigned char *binary = (unsigned char *)arg[2];
  size_t size = (size_t)arg[3];
  //用户态调用的exec(), 归根结底是do_execve()
  return do_execve(name, len, binary, size);
}
static int sys_yield(uint64_t arg[]) {
  return do_yield();
}
static int sys_kill(uint64_t arg[]) {
  int pid = (int)arg[0];
  return do_kill(pid);
}
static int sys_getpid(uint64_t arg[]) {
  return current->pid;
}
static int sys_putc(uint64_t arg[]) {
  int c = (int)arg[0];
  cputchar(c);
  return 0;
}
```

```
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;


fork_out:
    return ret;

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

上述代码是一个名为`do_fork`的函数，其功能是创建一个新的进程（子进程）。该函数接受三个参数：`clone_flags`表示进程创建的标志，`stack`表示子进程的内核栈地址，`tf`表示陷阱帧（trapframe）的指针。

代码的主要逻辑如下：

1. 首先，定义一个整型变量`ret`并初始化为`-E_NO_FREE_PROC`，表示没有可用的进程资源。
2. 检查当前进程数量是否已达到最大限制`MAX_PROCESS`，如果是，则跳转到`fork_out`标签处，将`ret`设置为`-E_NO_FREE_PROC`。
3. 如果当前进程数量未达到最大限制，则将`ret`设置为`-E_NO_MEM`，表示内存不足。
4. 根据注释提示，下面的代码是需要我们自己实现的。
5. 在注释中提供了一些宏、函数和定义，可以在下面的实现中使用。
6. 在实现中，我们需要完成以下步骤：
    - 调用`alloc_proc`函数分配一个`proc_struct`结构体，并对其初始化。
    - 调用`setup_kstack`函数为子进程分配一个内核栈。
    - 根据`clone_flags`参数的值，调用`copy_mm`函数将当前进程的内存空间（mm）复制或共享给子进程。
    - 调用`copy_thread`函数设置子进程的陷阱帧（trapframe）、内核入口点和堆栈。
    - 将子进程的`proc_struct`插入到进程哈希列表（hash_list）和进程列表（proc_list）中。
    - 调用`wakeup_proc`函数将子进程的状态设置为可运行（PROC_RUNNABLE）。
    - 使用子进程的pid设置`ret`的值。
7. 在代码的末尾，通过`fork_out`标签返回`ret`的值。

此外，在代码的后续部分还有错误处理的代码。如果在创建子进程的过程中发生错误，则会跳转到相应的错误处理标签，并释放相应的资源。

需要注意的是，代码中还有一个注释提到了在Lab5中需要更新的步骤，包括设置进程之间的关联链接（relation links）以及更新步骤1和步骤5的操作。然而，这部分的具体实现没有在提供的代码中给出，需要在Lab5中完成。

总之，上述代码的功能是创建一个新的子进程，并将其插入到进程列表中，使其处于可运行状态。

```
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);
        }
        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);
}
```

上述代码是一个名为`do_exit`的函数，其功能是使当前进程退出。该函数接受一个错误代码`error_code`作为参数。

代码的主要逻辑如下：

1. 首先，检查当前进程是否是空闲进程（idleproc）或初始化进程（initproc），如果是，则触发内核恐慌（panic）并输出相应的错误信息。
2. 获取当前进程的内存管理结构体（mm_struct）指针`mm`。
3. 如果`mm`不为NULL，则执行以下操作：
    - 使用`boot_cr3`重新加载CR3寄存器，切换到内核页表。
    - 调用`mm_count_dec`函数减少`mm`的引用计数，并判断引用计数是否减少到0。
        - 如果引用计数为0，则执行以下操作：
            - 调用`exit_mmap`函数清理与进程相关的内存映射。
            - 调用`put_pgdir`函数释放进程的页目录表。
            - 调用`mm_destroy`函数销毁`mm`结构体及其相关资源。
    - 将当前进程的`mm`指针设置为NULL。
4. 将当前进程的状态设置为`PROC_ZOMBIE`，表示进程已经退出。
5. 将错误代码`error_code`赋值给当前进程的`exit_code`字段。
6. 使用`local_intr_save`函数保存中断状态，并进入临界区。
    - 在临界区内，执行以下操作：
        - 将当前进程的父进程指针赋值给`proc`。
        - 如果`proc`的等待状态为`WT_CHILD`，则调用`wakeup_proc`函数唤醒`proc`。
        - 循环遍历当前进程的子进程链表（cptr）：
            - 将当前子进程指针赋值给`proc`。
            - 更新当前进程的子进程链表指针（cptr）为下一个子进程（optr）。
            - 将当前子进程的父指针（parent）设置为初始化进程（initproc）。
            - 将当前子进程插入到初始化进程的子进程链表中。
            - 如果当前子进程的状态为`PROC_ZOMBIE`，并且初始化进程的等待状态为`WT_CHILD`，则调用`wakeup_proc`函数唤醒初始化进程。
    - 离开临界区前，使用`local_intr_restore`函数恢复中断状态。
7. 调用`schedule`函数进行进程调度，选择下一个要运行的进程。
8. 最后，通过触发内核恐慌（panic）输出错误信息，表示`do_exit`函数不会返回。

综上所述，上述代码的功能是使当前进程退出，并进行相应的资源清理和状态更新，同时唤醒父进程和初始化进程，最后进行进程调度。

```
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) {
        cputs("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);
}
```

上述代码是一个名为 `do_execve` 的函数，其功能是执行一个新的可执行文件。该函数接受四个参数： `name` 表示可执行文件名的指针， `len` 表示可执行文件名的长度， `binary` 表示可执行文件的二进制数据指针， `size` 表示可执行文件的大小。

代码的主要逻辑如下：

1. 首先，获取当前进程的内存管理结构体（mm_struct）指针 `mm` 。
2. 使用 `user_mem_check` 函数检查 `name` 指针和长度是否合法，即检查可执行文件名是否位于用户空间。
    - 如果不合法，则返回错误码 `-E_INVAL` 表示参数无效。
3. 如果可执行文件名的长度超过了 `PROC_NAME_LEN` ，则将长度截断为 `PROC_NAME_LEN` 。

4. 创建一个字符数组`local_name`，并使用`memset`函数将其初始化为全0。
5. 使用`memcpy`函数将可执行文件名拷贝到`local_name`数组中。
6. 如果`mm`不为NULL，则执行以下操作：
    - 输出调试信息（"mm != NULL"）。
    - 使用`boot_cr3`重新加载CR3寄存器，切换到内核页表。
    - 调用`mm_count_dec`函数减少`mm`的引用计数，并判断引用计数是否减少到0。
        - 如果引用计数为0，则执行以下操作：
            - 调用`exit_mmap`函数清理与进程相关的内存映射。
            - 调用`put_pgdir`函数释放进程的页目录表。
            - 调用`mm_destroy`函数销毁`mm`结构体及其相关资源。
    - 将当前进程的`mm`指针设置为NULL。
7. 定义一个整型变量`ret`用于保存返回值。
8. 调用`load_icode`函数加载可执行文件的二进制数据，并将返回值保存到`ret`中。
    - 如果加载失败，则跳转到`execve_exit`标签处。
9. 调用`set_proc_name`函数设置当前进程的名称为`local_name`。
10. 返回0，表示执行成功。

在`execve_exit`标签处，调用`do_exit`函数以返回错误码`ret`并使当前进程退出。然后触发内核恐慌（panic）输出错误信息，表示进程已经退出但仍然执行到了此处。

综上所述，上述代码的功能是执行一个新的可执行文件，包括参数检查、资源清理、加载可执行文件、设置进程名称等操作，并返回相应的结果。

```
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;
}
```

上述代码是一个名为 `do_wait` 的函数，其功能是等待一个子进程退出并获取其退出状态。该函数接受两个参数： `pid` 表示要等待的子进程的ID， `code_store` 为存储子进程退出状态的指针。

代码的主要逻辑如下：

1. 首先，获取当前进程的内存管理结构体（mm_struct）指针 `mm` 。
2. 如果 `code_store` 不为NULL，则执行以下操作：
    - 使用 `user_mem_check` 函数检查 `code_store` 指针和大小是否合法，即检查存储子进程退出状态的内存是否位于用户空间。
        - 如果不合法，则返回错误码 `-E_INVAL` 表示参数无效。
3. 定义一个 `struct proc_struct` 类型的指针 `proc` 和布尔类型的变量 `intr_flag` 和 `haskid` 。
4. 进入一个重复的循环（ `repeat` 标签处），重复以下操作：
    - 将 `haskid` 设置为0。
    - 如果 `pid` 不等于0，则执行以下操作：
        - 调用 `find_proc` 函数根据 `pid` 查找子进程的进程控制块（proc_struct）指针，并将结果保存到 `proc` 中。
        - 如果 `proc` 不为NULL，并且 `proc` 的父进程是当前进程，则将 `haskid` 设置为1。
        - 如果 `proc` 的状态为 `PROC_ZOMBIE` ，则跳转到 `found` 标签处。
    - 否则，如果 `pid` 等于0，则执行以下操作：
        - 将 `proc` 指针设置为当前进程的子进程链表指针（cptr）。
        - 进行一个循环遍历子进程链表，从 `proc` 开始，直到 `proc` 为NULL，每次迭代将 `proc` 指针更新为下一个子进程（optr）。
            - 将 `haskid` 设置为1。
            - 如果 `proc` 的状态为 `PROC_ZOMBIE` ，则跳转到 `found` 标签处。
    - 如果 `haskid` 为真，则执行以下操作：
        - 将当前进程的状态设置为 `PROC_SLEEPING` ，表示当前进程正在等待。
        - 将当前进程的等待状态设置为 `WT_CHILD` ，表示当前进程正在等待子进程的退出。
        - 调用 `schedule` 函数进行进程调度，选择下一个要运行的进程。
        - 如果当前进程的标志中包含 `PF_EXITING` ，表示当前进程正在退出，则调用 `do_exit` 函数以返回错误码 `-E_KILLED` 并使当前进程退出。

        - 跳转到`repeat`标签处，重新开始循环。
5. 如果没有找到符合条件的子进程，则返回错误码`-E_BAD_PROC`表示没有有效的子进程等待。
6. 在`found`标签处，执行以下操作：
    - 检查`proc`是否为空闲进程（idleproc）或初始化进程（initproc），如果是，则触发内核恐慌（panic）并输出相应的错误信息。
    - 如果`code_store`不为NULL，则将子进程的退出状态（exit_code）保存到`code_store`指针指向的内存中。
    - 使用`local_intr_save`函数保存中断状态，并进入临界区。
        - 在临界区内，执行以下操作：
            - 从进程哈希表中移除`proc`。
            - 从进程链表中移除`proc`。
        - 离开临界区前，使用`local_intr_restore`函数恢复中断状态。
7. 使用`local_intr_restore`函数恢复中断状态。
8. 返回。

综上所述，上述代码的功能是等待指定的子进程退出，并获取其退出状态。它会进行参数检查、查找子进程、等待子进程退出、保存退出状态以及相应的资源清理等操作。

在给定的代码中，有多个函数涉及到用户态和内核态之间的交错执行。

1. `do_fork`函数：该函数在内核态执行。它创建一个新的进程，并在内核态进行一系列的初始化操作，如分配进程控制块（`proc_struct`）、内存分配等。在创建完成后，它将返回到用户态，并将创建的进程ID返回给用户程序。

2. `do_exit`函数：该函数在内核态执行。它用于终止当前进程，并执行一系列的清理操作，比如释放进程的内存资源、更新进程状态等。最后，它调用调度器选择下一个要执行的进程，并不会返回到用户程序。

3. `do_execve`函数：该函数在内核态执行。它用于加载并执行一个新的程序。它首先进行一些检查，如验证传递的参数是否有效，然后在内核态进行一系列的操作，如销毁当前进程的内存空间、加载新的程序等。最后，它将设置新程序的名称并返回到用户态。

4. `do_wait`函数：该函数在用户态和内核态之间交错执行。它用于等待子进程的退出。在函数的执行过程中，它会先在内核态进行一些检查，然后在用户态执行一些操作，如检查子进程的状态是否为`PROC_ZOMBIE`，如果是则获取子进程的退出码，并在内核态进行一些清理操作，如从进程链表中移除、释放相关资源等。最后，它将返回到用户程序并返回相应的退出码。

在用户态和内核态之间的交错执行是通过系统调用实现的。当用户程序需要访问内核的功能时，它会通过系统调用的方式切换到内核态执行相应的内核函数。执行完成后，内核将结果返回给用户程序，并再次切换到用户态继续执行。

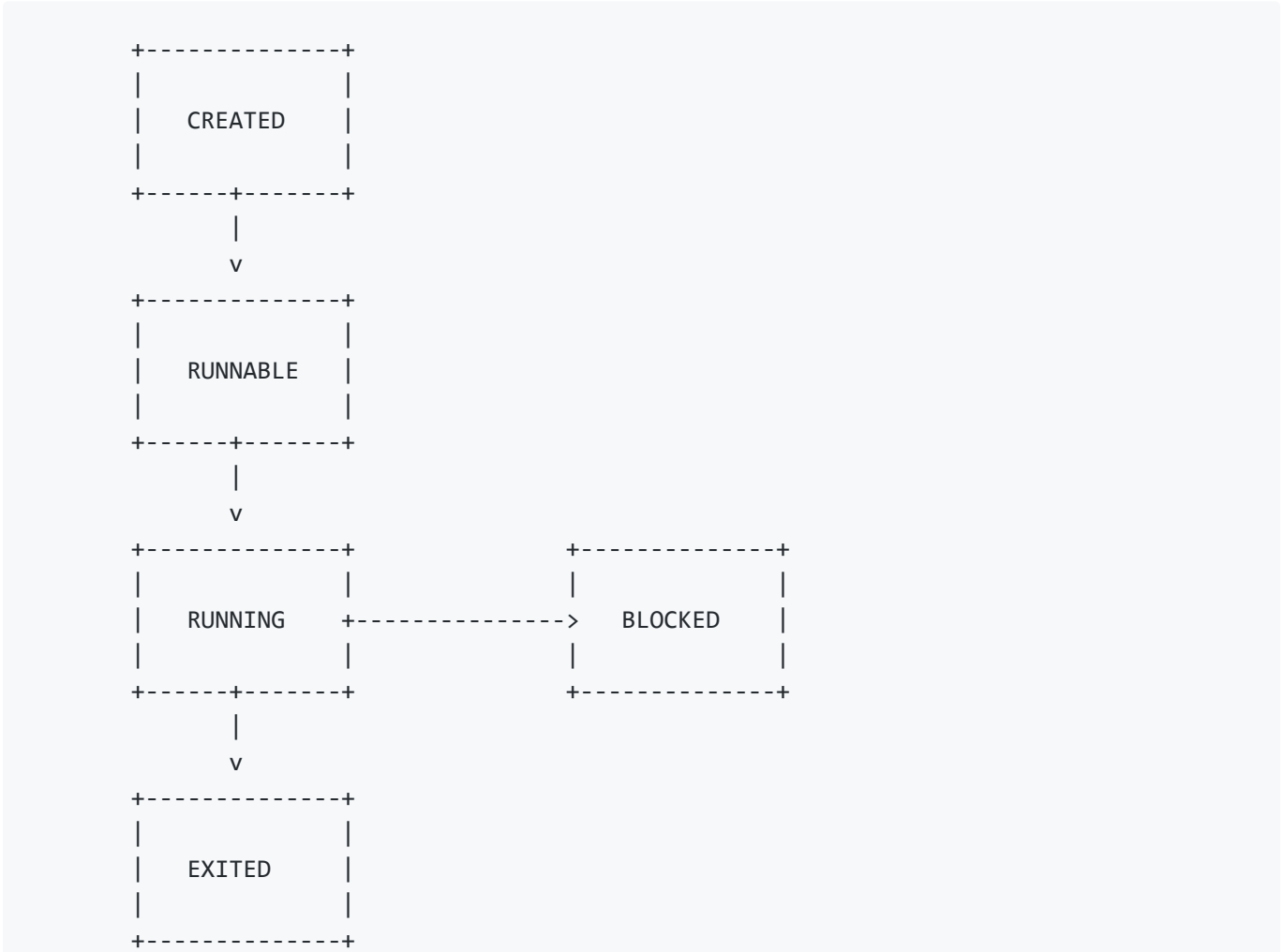
内核态执行的结果通过返回值或者传递给用户程序的参数来返回。在给定的代码中，通过函数的返回值来表示操作的结果。例如，在`do_fork`函数中，返回的`ret`变量用于表示创建新进程的结果，负数表示失败，正数表示成功，并返回给用户程序进行处理。在`do_wait`函数中，通过传递给函数的指针`code_store`来返回子进程的退出码，用户程序可以通过读取该指针指向的内存位置获取退出码。

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）

```
process state changing:

  alloc_proc                                  RUNNING
      +                                   +--<----<--+
      +                                   + proc_run +
      V                                   +-->---->--+
PROC_UNINIT -- proc_init/wakeup_proc --> PROC_RUNNABLE -- do_wait/do_sleep -->
PROC_SLEEPING --
                                            ^      +
+
                                            |      +--- do_exit --> PROC_ZOMBIE
+
                                            +
+
                                            -------------------wakeup_proc-------------
----------
```

process state	meaning	reason
PROC_UNINIT	uninitialized	alloc_proc
PROC_SLEEPING	sleeping	do_wait, do_sleep
PROC_RUNNABLE	runnable(maybe running)	proc_init, wakeup_proc
PROC_ZOMBIE	almost dead	do_exit



1. **CREATED**: 进程创建阶段，进程正在被创建，但尚未执行。在这个阶段，操作系统会为进程分配资源，并初始化进程的数据结构。
    - 产生变换的事件或函数调用：进程创建函数调用（如`fork()`）。
2. **RUNNABLE**: 进程就绪阶段，进程已经准备好执行，但还没有被调度执行。在这个阶段，进程等待系统调度器选择它来执行。
    - 产生变换的事件或函数调用：进程调度器选择该进程执行。
3. **RUNNING**: 进程运行阶段，进程正在被CPU执行。在这个阶段，进程的指令被CPU执行，完成实际的计算和操作。
    - 产生变换的事件或函数调用：CPU分配时间片给该进程，使其开始运行。
4. **BLOCKED**: 进程阻塞阶段，进程暂时不能执行，等待某个事件的发生才能继续执行。在这个阶段，进程无法继续执行，直到等待的事件发生。
    - 产生变换的事件或函数调用：进程等待某个事件（如I/O操作、信号等）。

5. **EXITED**: 进程退出阶段，进程已经执行完毕或被终止。在这个阶段，进程释放占用的资源，并等待操作系统回收其进程控制块及其他相关数据结构。
    - 产生变换的事件或函数调用：进程执行完毕或被强制终止（如调用`exit()`函数）。

进程在不同状态之间的转换关系如下：

- 进程从**CREATED**状态开始，通过进程创建函数（如`fork()`）被创建。
- 进程从**CREATED**状态转换到**RUNNABLE**状态，表示进程已经准备好执行，等待系统调度器选择它来运行。
- 进程从**RUNNABLE**状态转换到**RUNNING**状态，表示进程被CPU调度执行，开始运行。
- 进程在**RUNNING**状态下，可能发生以下转换：
    - 如果进程需要等待某个事件（如I/O操作、信号等），则进程从**RUNNING**状态转换到**BLOCKED**状态，等待事件发生。
    - 如果进程的时间片用完，或者被更高优先级的进程抢占，那么进程从**RUNNING**状态转换到**RUNNABLE**状态，等待下一次被调度执行。
- 当进程在**BLOCKED**状态等待的事件发生时，进程从**BLOCKED**状态转换到**RUNNABLE**状态，等待系统调度器选择它来运行。
- 当进程执行完毕或被强制终止时，进程从**RUNNING**或**BLOCKED**状态转换到**EXITED**状态，表示进程已经退出。

需要注意的是，状态之间的转换可以由不同的事件或函数调用触发，如进程创建、进程调度、事件发生、进程退出等。具体的转换过程和触发条件会受操作系统的具体实现和调度策略的影响。上述描述仅为一般情况下的状态转换示意，并不涵盖所有可能的细节和情况。

扩展练习 Challenge

1. 实现 Copy on Write （COW）机制

给出实现源码, 测试用例和设计报告（包括在 cow 情况下的各种状态转换（类似有限状态自动机）的说明）。

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。

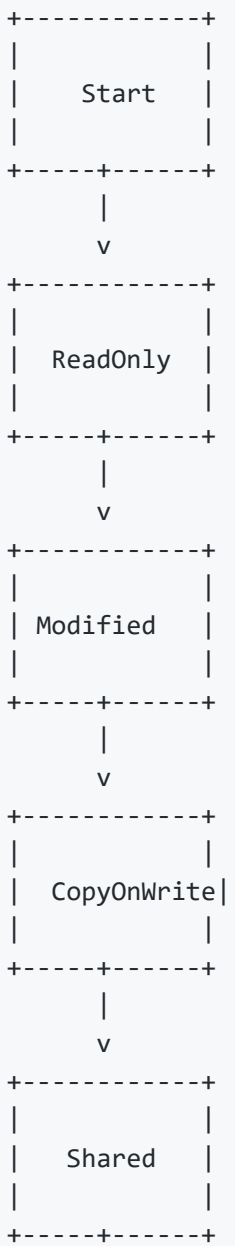
由于 COW 实现比较复杂，容易引入 bug，请参考 https://dirtycow.ninja/ 看看能否在 ucore 的 COW 实现中模拟这个错误和解决方案。需要有解释。

说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

- 是在调用load_icode函数的时候被加载到内存中的。而这个函数又是在调用do_execve的时候被调用的，根据传入的binary找到用户程序所在的地址。

2. 说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

   COW 在不同状态之间的状态转换示意图：



关于用户程序何时被预先加载到内存中，与常用操作系统的加载方式有所不同。在常用操作系统中，用户程序通常在执行之前会被完全加载到内存中，以便CPU直接执行其中的指令。而在 COW

机制下，用户程序的加载是延迟进行的。具体而言，当一个进程执行一个用户程序时，操作系统会将程序的代码段（只读）加载到内存中，但其他数据段（如堆和栈）不会立即加载。这样可以节省内存空间，并且实现了写时复制的机制。

原因是，许多用户程序在执行过程中并不需要修改代码段以外的数据，因此延迟加载这些数据段可以节省内存开销。只有在进程进行写操作时，才会触发对数据段的加载和复制操作，以确保每个进程都具有独立的修改副本。这种延迟加载和写时复制的机制有效地支持了多个进程之间的内存共享，同时提供了高效的内存使用和保护机制。

2. 总结

本次实验主要涉及进程的一些进程的知识，比如创建，管理，切换到用户态进程的具体实现；加载ELF可执行文件的具体实现；对系统调用机制的具体实现

理论课老师就讲过进程管理是操作系统很重要的一个知识点，也特别提到了进程五状态，这一点也在实验中有体现。challenge实现的COW，在Linux上也有体现所以相关资料很多，讲的十分详细，对challenge的实现有很大帮助，而且challenge也有注释提示

做完这次实验，对进程管理有了更深的认识，同时对理论课上学到的知识有了更进一步的理解，还能查漏补缺，发现自己遗忘或者没学好的知识点。期待自己未来更好的发展，`万事胜意、心想事成、未来可期`