Lab 5

操作系统课程实验报告

实验名称：Lab5 用户程序

小组成员： 2113388 高涵 2112849唐静蕾 2113999 陈翊炀

一、实验目的

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。

二、实验内容

练习1：加载应用程序并执行

> **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。
>
> - 请简要描述这个用户态进程被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

补充`load_icode`的第6步，设置正确的trapframe内容：

```
     * should set tf->gpr.sp, tf->epc, tf->status
     * NOTICE: If we set trapframe correctly, then the user level process can re
     *          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
     *          hint: check meaning of SPP, SPIE in SSTATUS, use them by SSTATUS
     */
     tf->gpr.sp=USTACKTOP;
     tf->epc=elf->e_entry;
     tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

1. 在经过调度器占用了 CPU 的资源之后，用户态进程调用了 `exec` 系统调用，从而转入到了系统调用的处理例程；
2. 在经过了正常的中断处理例程之后，最终控制权转移到了 `syscall.c` 中的 `syscall` 函数，然后根据系统调用号转移给了 `sys_exec` 函数，在该函数中调用了上文中提及的 `do_execve` 函数来完成指定应用程序的加载；
3. 在 `do_execve` 中进行了若干设置，包括推出当前进程的页表，换用 `kernel` 的 PDT 之后，使用 `load_icode` 函数，完成了对整个用户线程内存空间的初始化，包括堆栈的设置以及将 `ELF` 可执行文件的加载，之后通过 `current->tf` 指针修改了当前系统调用的 `trapframe` ，使得最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处；
4. 在完成了 `do_exec` 函数之后，进行正常的中断返回的流程，由于中断处理例程的栈上面的 `eip` 已经被修改成了应用程序的入口处，而 `CS` 上的 `CPL` 是用户态，因此 `iret` 进行中断返回的时候会将堆栈切换到用户的栈，并且完成特权级的切换，并且跳转到要求的应用程序的入口处；
5. 接下来开始具体执行应用程序的第一条指令；

练习2：父进程复制自己的内存空间给子进程

> 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。
>
> - 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

此处同样按照注释提示直接填充即可，具体如下：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
               bool share) {
    ...
    /* 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
```

```
18          * (2) find dst_kvaddr: the kernel virtual address of npage
19          * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
20          * (4) build the map of phy addr of  nage with the linear addr start
21          */
22         void * kva_src = page2kva(page);
23         void * kva_dst = page2kva(npage);
24         memcpy(kva_dst, kva_src, PGSIZE);
25         ret = page_insert(to, npage, start, perm);
26         ...
27     }
```

填充部分的具体解析如下：

- `void * kva_src = page2kva(page);`：
  - 使用 `page2kva` 宏将源页面 `page` 的物理地址映射到内核的虚拟地址。
- `void * kva_dst = page2kva(npage);`：
  - 使用 `page2kva` 宏将目标页面 `npage` 的物理地址映射到内核的虚拟地址。得到目标页面在内核虚拟地址空间中的起始地址。
- `memcpy(kva_dst, kva_src, PGSIZE);`：
  - 使用 `memcpy` 函数将源页面的内容复制到目标页面。
- `ret = page_insert(to, npage, start, perm);`：
  - 使用 `page_insert` 函数建立目标地址空间 `to` 中目标页面 `npage` 与指定线性地址 `start` 之间的映射关系。
  - `start` 是当前复制的线性地址， `perm` 是权限标志，包括用户/内核态、读/写/执行等。

整体而言，填充代码块的作用是将源地址空间的一页内容复制到目标地址空间的一页，并在目标地址空间中建立相应的映射关系。这通常用于实现进程间的共享或复制内存。

> 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。

此处仅基于 *Copy on Write* 机制给出相应的概要设计：

- **页面管理：**
  - 每个页面都附加一个引用计数（reference count）。引用计数跟踪有多少个进程共享相同的物理页面。
  - 在页表项中，增加一个标志位，用于表示该页面是否是只读的。
- **写时复制操作：**
  - 当一个进程试图写入一个只读页面时，操作系统会进行写时复制。
  - 操作系统会检查引用计数，如果引用计数为1（表示只有一个进程拥有该页面），那么直接将页面标记为可写。如果引用计数大于1，那么进行页面复制。

    - 复制操作涉及到为新的页面分配物理内存，并将原始页面的内容复制到新页面。
    - 新页面的引用计数设为1，原始页面的引用计数减1。
- **页表更新：**
    - 更新原始页面的页表项，将其指向新的物理页面。
    - 更新新页面的页表项，将其标记为可写。
- **引用计数维护：**
    - 当一个进程终止或释放对页面的引用时，引用计数减1。
    - 当引用计数减为0时，释放相应的物理内存。

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

> 请简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：
>
> - 请分析fork/exec/wait/exit的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？
> - 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）

首先我们可以罗列下目前 ucore 所有的系统调用，如下表所示：

系统调用名	含义	具体完成功能的函数
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
	get the process's pid	

SYS_getpid		

fork:

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

```
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;

     //    1. call alloc_proc to allocate a proc_struct
    if ((proc = alloc_proc()) == NULL) { //分配内存失败
        goto fork_out; //返回
    }
    proc->parent = current;
    assert(proc->wait_state == 0);//! ! 新增加
    //    2. call setup_kstack to allocate a kernel stack for child process
     if (setup_kstack(proc) != 0)              //调用setup_kstack为进程分配一个内核
                goto bad_fork_cleanup_proc;

    //    3. call copy_mm to dup OR share mm according clone_flag
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc,stack,tf);
    //    5. insert proc_struct into hash_list && proc_list

     bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc);//新增加! !
        //list_add(&proc_list, &(proc->list_link));
        //nr_process ++;
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);
```

```
40
41      ret=proc->pid;
42
43
44  fork_out:
45      return ret;
46
47  bad_fork_cleanup_kstack:
48      put_kstack(proc);
49  bad_fork_cleanup_proc:
50      kfree(proc);
51      goto fork_out;
52  }
53
```

首先当程序执行 fork 时，fork 使用了系统调用 SYS_fork，而系统调用 SYS_fork 则主要是由 do_fork 和 wakeup_proc 来完成的。

主要完成了以下工作：

1. 调用alloc_proc来分配proc_struct;
2. 调用setup_kstack为子进程分配内核堆栈;
3. 调用copy_mm来复制或根据clone_flag共享mm;
4. 调用copy_thread在proc_struct中设置tf&context;
5. 将proc_struct插入到散列列表和proc_list中;
6. 调用wakeup_proc使新的子进程可运行;
7. 自此，进程已经准备好执行了，把进程状态设置为“就绪”态;
8. 使用子进程的pid设置ret值。

而 wakeup_proc 函数主要是将进程的状态设置为等待，即 proc->wait_state = 0。

代码中涉及到的操作主要是在**内核态**完成的，因为代码中涉及到了进程的创建、内存管理、进程调度等操作，这些都是操作系统的核心功能，需要在内核态进行。

当用户程序需要创建一个新的进程时，它会发起系统调用，并陷入内核态，然后调用 do_fork 函数。在 do_fork 函数中，会进行一系列的内核态操作，包括分配 proc_struct 结构体、分配内核栈、拷贝内存、设置 trapframe 和 context 以及插入进程到相应的列表中。最后，执行 wakeup_proc 函数唤醒新创建的子进程。

在内核态执行完成后，会将创建的子进程的 pid 返回给用户程序，作为执行结果。用户程序可以通过系统调用的返回值获取到这个 pid，并继续在用户态执行。

exec：

调用过程：`SYS_exec->do_execve`

```
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);
// 释放内存
    if (mm != NULL) {
        cputs("mm != NULL");
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            // 删除该内存管理所对应的PDT
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
     // 加载可执行文件代码，重设mm_struct，以及重置trapframe
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    // 设置进程名称
    set_proc_name(current, local_name);
    return 0;

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

当应用程序执行的时候，会调用 SYS_exec 系统调用，而当 ucore 收到此系统调用的时候，则会使用 do_execve() 函数来实现，因此这里我们主要介绍 do_execve() 函数的功能，函数主要时完成用户进程的创建工作，同时使用户进程进入执行。

主要工作如下：

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。

在这段代码中，do_execve 函数是在**内核态**完成的，当调用这个函数时，它首先进行了内存检查，然后根据传入的可执行文件的二进制码加载代码并重设 mm_struct 和 trapframe 结构体，最后设置进程名称并返回。如果出现错误，则会跳转到 execve_exit 标签处调用 do_exit 函数退出进程，并输出错误信息。由于进程已经处于退出状态，不再需要返回执行权给用户程序，因此不需要将执行结果返回给用户程序。

wait：

调用过程为：`SYS_wait->do_wait`

```
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) {//如果pid! =0，则找到进程id为pid的处于退出状态的子进程
        proc = find_proc(pid);
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;//找到进程
            }
        }
    }
    else {//如果pid==0，则随意找一个处于退出状态的子进程
        proc = current->cptr;
```

```
        for (; proc != NULL; proc = proc->optr) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    if (haskid) {//如果没找到，则父进程重新进入睡眠，并重复寻找的过程
        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;
//释放子进程的所有资源
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);//将子进程从hash_list中删除
        remove_links(proc);//将子进程从proc_list中删除
    }
    local_intr_restore(intr_flag);
    put_kstack(proc); //释放子进程的内核堆栈
    kfree(proc); //释放子进程的进程控制块
    return 0;
}
```

当执行 wait 功能的时候，会调用系统调用 SYS_wait，而该系统调用的功能则主要由 do_wait 函数实现，主要工作就是父进程如何完成对子进程的最后回收工作，具体的功能实现如下：

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。

用户态完成：

- user_mem_check函数的调用，用于检查用户空间内存的合法性。

内核态完成：

- find_proc函数的调用，用于查找指定pid的进程。
- unhash_proc函数的调用，用于将进程从进程哈希表中移除。
- remove_links函数的调用，用于移除进程与父子进程之间的链接。
- put_kstack函数的调用，用于释放进程的内核栈。
- kfree函数的调用，用于释放进程所占用的内核内存。

内核态与用户态程序的交错执行是通过系统调用实现的。当用户程序需要进行系统调用时，会触发从用户态切换到内核态。在内核态执行相应的系统调用处理程序，完成相应的操作后，再切换回用户态继续执行用户程序

在这段代码中，函数do_wait的返回值表示操作的成功或失败，通过返回值告知用户程序执行的结果。同时，如果code_store参数不为NULL，那么通过指针传递将proc->exit_code的值返回给用户程序。

exit：

调用过程为：`SYS_exit->exit`

首先我们来看看 do_exit 函数的实现过程：

```
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) {//如果该进程是用户进程
        lcr3(boot_cr3);//切换到内核态的页表
```

```
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
 /*如果没有其他进程共享这个内存释放current->mm->vma链表中每个vma描述的进程合法空间中实际

            put_pgdir(mm);//释放页目录占用的内存
            mm_destroy(mm); //释放mm占用的内存
        }
        current->mm = NULL;
    }
    current->state = PROC_ZOMBIE;//僵死状态
    current->exit_code = error_code;//等待父进程做最后的回收
    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);//如果父进程在等待子进程，则唤醒
        }
        while (current->cptr != NULL) {
  /*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initproc，且各个子
            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(); //选择新的进程执行
    panic("do_exit will not return!! %d.\n", current->pid);
}

            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
```

```
59                 initproc->cptr->yptr = proc;
60             }
61             proc->parent = initproc;
62             initproc->cptr = proc;
63             if (proc->state == PROC_ZOMBIE) {
64                 if (initproc->wait_state == WT_CHILD) {
65                     wakeup_proc(initproc);
66                 }
67             }
68         }
69     }
70     local_intr_restore(intr_flag);
71     schedule();
72     panic("do_exit will not return!! %d.\n", current->pid);
73 }
```

当执行 exit 功能的时候，会调用系统调用 SYS_exit，而该系统调用的功能主要是由 do_exit 函数实现。具体过程如下：

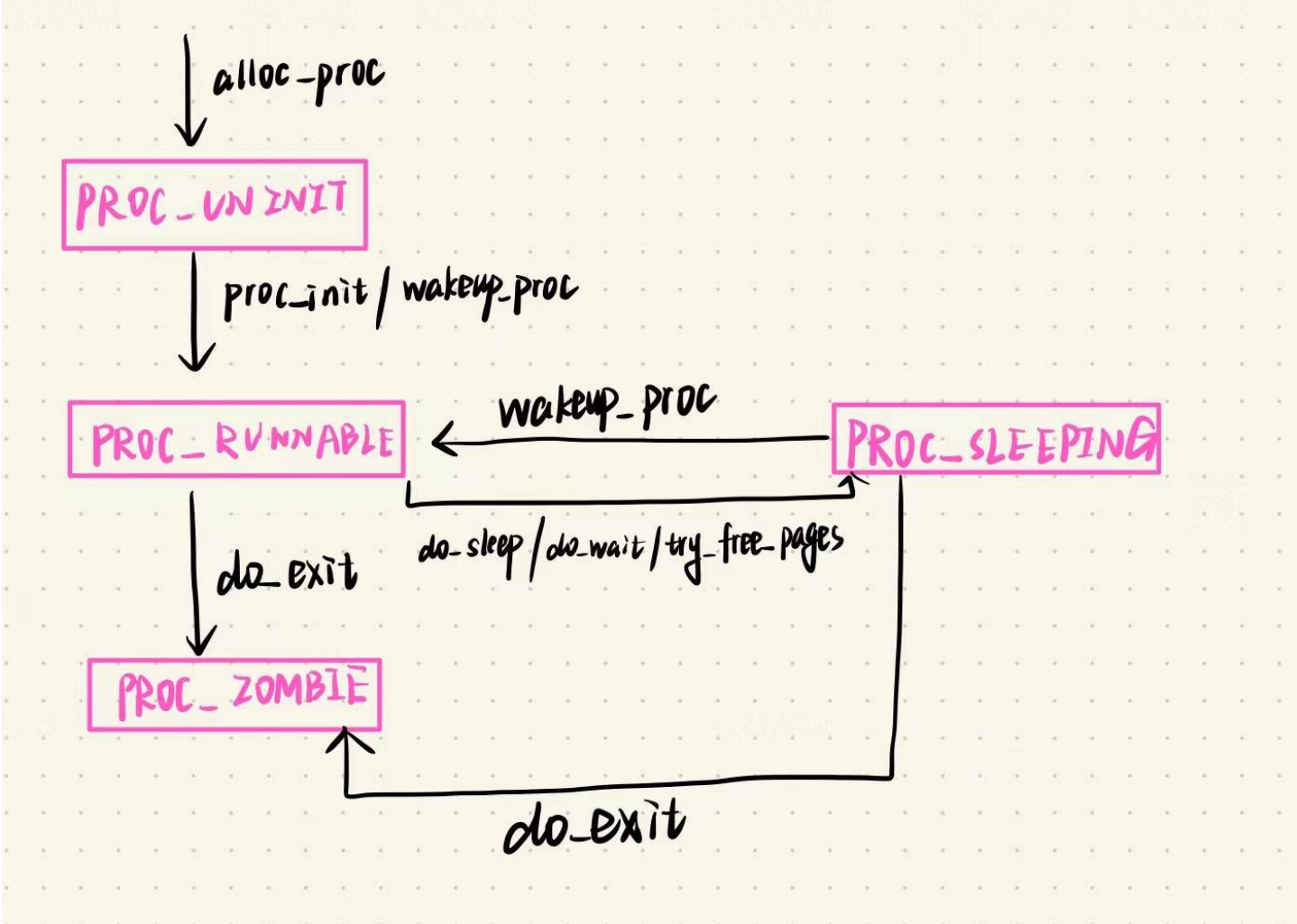
1. 先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间;（具体的回收过程不作详细说明）
2. 设置当前进程的中hi性状态为 PROC_ZOMBIE，然后设置当前进程的退出码为 error_code。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块
3. 如果当前父进程已经处于等待子进程的状态，即父进程的 wait_state 被置为 WT_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。
4. 如果当前进程还有子进程,则需要把这些子进程的父进程指针设置为内核线程 init，且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。

所以说该函数的功能简单的说就是，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。

这个函数中涉及的操作主要是在内核态完成的，因为代码中涉及到了对进程的管理和调度，以及对内存管理的操作，这些都是操作系统的核心功能，需要在**内核态**进行。

在这段代码中，do_exit 函数是在内核态完成的，当调用这个函数时，当前进程会被标记为 `PROC_ZOMBIE`，并进行一系列的清理和处理操作。最后通过调用 panic 函数输出错误信息，并且不会返回给调用者，这是因为该进程已经处于退出状态，不再需要返回执行权给用户程序。

一个用户态进程的执行状态生命周期图:



最后make grade结果：

```
  -check output:                          OK
divzero:                  (1.0s)
  -check result:                          OK
  -check output:                          OK
softint:                  (1.0s)
  -check result:                          OK
  -check output:                          OK
faultread:                (1.0s)
  -check result:                          OK
  -check output:                          OK
faultreadkernel:          (1.0s)
  -check result:                          OK
  -check output:                          OK
hello:                    (1.0s)
  -check result:                          OK
  -check output:                          OK
testbss:                  (1.0s)
  -check result:                          OK
  -check output:                          OK
pgdir:                    (1.0s)
  -check result:                          OK
  -check output:                          OK
yield:                    (1.0s)
  -check result:                          OK
  -check output:                          OK
badarg:                   (1.0s)
  -check result:                          OK
  -check output:                          OK
exit:                     (1.0s)
  -check result:                          OK
  -check output:                          OK
spin:                     (4.1s)
  -check result:                          OK
  -check output:                          OK
forktest:                 (1.0s)
  -check result:                          OK
  -check output:                          OK
Total Score: 130/130
gaohan@gaohan-virtual-machine:~/riscv64-ucore-labcodes/lab5$
```