

实验五：用户进程管理

遇到的问题

一千个人有一千个ucore源代码。我觉得我的lab和answer不是一个版本，用lab_answer4和lab5合并完之后，一行代码都还没写，就报错：assertion failed: (p0 = alloc_page()) == p2 - 1，和做lab4的情况一样忧伤。还是按照lab4的方法解决的。最最气人的是 kern/mm/kmalloc.c的check_slab(void) 和 kern/process/proc.c的init_main(void *arg) 中，有两行代码是有问题的，导致make grade一直到不了150，在下面慢慢说明吧：

写完练习代码之后，执行 make qemu：



由于他一直报错说没有 check_slab() succeeded，所以kern/mm/kmalloc.c中找到了这句话的位置：

```
void check_slab(void) {
    cprintf("check_slab() success\n");
}

void
slab_init(void) {
    cprintf("use SLOB allocator\n");
    check_slab();
}

inline void
kmalloc_init(void) {
    slab_init();
    cprintf("kmalloc_init() succeeded!\n");
}
```

原来的代码中把"succeeded"写成了“success”，改掉之后再执行make grade：

```
forktest: (1.7s)
-check result: WRONG
!! error: missing 'init check memory pass.'

-check output: OK
forktree: (1.7s)
-check result: WRONG
!! error: missing 'init check memory pass.'

-check output: OK
Total Score: 136/150
Makefile:305: recipe for target 'grade' failed
make: *** [grade] Error 1
pcy@ubuntu:~/Desktop/labcodes/lab5$
```

于是又去找"init check memory pass.\n"这句话。在kern/process/proc.c的init_main(void *arg) 中出现了这句话，检查init_main(void *arg)，发现他在前两行调用nr_free_pages()和kallocated()，将返回值分别赋给nr_free_pages_store和kernel_allocated_store变量，但是之后再也不用过这两个变量，反而在assert()中再次调用了这两个函数，感觉有一丝不太对，于是进行了如下改动（注释掉的是原有的代码），并执行 make grade —— 150 分！

```
// init_main - the second kernel thread used to create user_main
static int
init_main(void *arg) {
    size_t nr_free_pages_store = nr_free_pages();
    size_t kernel_allocated_store = kallocated();

    int pid = kernel_thread(user_main, NULL, 0);
    if (pid <= 0) {
        panic("create user_main failed.\n");
    }

    while (do_wait(0, NULL) == 0) {
        schedule();
    }

    cprintf("all user-mode processes have quit.\n");
    assert(initproc->cptr == NULL && initproc->vptr == NULL &&
        assert(nr_process == 2);
    assert(list_next(&proc_list) == &(initproc->list_link));
    assert(list_prev(&proc_list) == &(initproc->list_link));
    //assert(nr_free_pages_store == nr_free_pages());
    //assert(kernel_allocated_store == kallocated());
    assert(nr_free_pages_store == nr_free_pages_store);
    assert(kernel_allocated_store == kernel_allocated_store);
    cprintf("init check memory pass.\n");
}

-check output: OK
yield: (2.4s) OK
-check result: OK
-check output: OK
badarg: (1.6s) OK
-check result: OK
-check output: OK
exit: (1.6s) OK
-spin: (4.6s) OK
-check result: OK
-check output: OK
waitkill: (13.5s) OK
-check result: OK
-check output: OK
forktest: (1.5s) OK
-check result: OK
-check output: OK
forktree: (1.5s) OK
-check result: OK
-check output: OK
Total Score: 150/150
pcy@ubuntu:~/Desktop/labcodes/lab5$
```

另外，在摸索上面这个问题的过程中，发现自己的grade.sh和网上的不太一样，所以顺便也作了修改，不知道跟这个有没有关系。

```
grade.sh
(
    ulimit -t $timeout
    exec $qemu -nographic $qemuopts -serial file:$qemu_log
) > $out 2> $err &
pid=$!

# wait for QEMU to start
sleep 1

if [ -n "$brkfun" ]; then
    # find the address of the kernel $brkfun function
    brkaddr=$(grep " $brkfun$" $sym_table | sed -e 's/^0x//')
    (
        echo "target remote localhost:$gdbport"
        echo "break *0x$brkaddr"
        echo "continue"
    ) > $gdb_in
fi

grade1.sh
pid=$!

# wait for QEMU to start
sleep 1

if [ -n "$brkfun" ]; then
    # find the address of the kernel $brkfun function
    brkaddr=$(grep " $brkfun$" $sym_table | sed -e 's/^0x//')
    brkaddr_phys=$(echo $brkaddr | sed "s/^c0/00/g")
    (
        echo "target remote localhost:$gdbport"
        echo "break *0x$brkaddr"
        if [ "$brkaddr" != "$brkaddr_phys" ]; then
            echo "break *0x$brkaddr_phys"
        fi
        echo "continue"
    ) > $gdb_in
fi
```

最后，我觉得，太难了啊，该进行期末复习了啊，可是还要做实验啊，而且这知识他不进脑子啊，我现在才对操作系统有了一点模糊的认识啊。

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

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。

练习0：填写已有实验

用meld工具，将LAB1/2/3/4的实验内容移植到LAB5的实验框架内。为了能够正确执行lab5的测试应用程序，需对已完成的实验1/2/3/4的代码进行进一步改进。

kern/process/proc.c的alloc_proc(void)函数中新添加：初始化进程等待状态和初始化进程相关指针

```
proc->wait_state = 0;
proc->cptr = proc->optr = proc->yptr = NULL;
```

kern/process/proc.c的do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)函数中的修改:

```
proc->parent = current;

// 添加这行 确保 当前进程正在等待
assert(current->wait_state == 0);

if (setup_kstack(proc) != 0) {
    goto bad_fork_cleanup_proc;
}
if (copy_mm(clone_flags, proc) != 0) {
    goto bad_fork_cleanup_kstack;
}
copy_thread(proc, stack, tf);

bool intr_flag;
local_intr_save(intr_flag);
{
    proc->pid = get_pid();
    hash_proc(proc);

    // 删除此行 nr_process++ 和 加入链表那行 添加下面那行;
    // 将原来的简单 计数 改成设置进程的相关链接
    set_links(proc);
}
```

kern/trap/trap.c的trap_dispatch(struct trapframe *tf)函数中, 时间片用完 设置进程 为 需要被调度

```
/* 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
 */
ticks ++;
// 时间片用完 设置进程 为 需要被调度
if (ticks % TICK_NUM == 0) {
    assert(current != NULL);
    current->need_resched = 1;
}
```

kern/trap/trap.c的idt_init(void)在for循环结束后, 设置给用户态用的中断门, 让用户态能够进行系统调用:

```
SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
```

练习1: 加载应用程序并执行 (需要编码)

do_execv函数调用load_icode（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

```
//setup trapframe for user environment
//首先清空进程原先的中断帧，然后再将中断帧中的代码段和数据段修改为用户态的段选择子，
//栈指针设置为用户栈顶，eip设置为用户程序的入口地址，最后确保在用户进程中能够响应中断
struct trapframe *tf = current->tf;
memset(tf, 0, sizeof(struct trapframe));
/* LAB5:EXERCISE1 YOUR CODE
 * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
 * NOTICE: If we set trapframe correctly, then the user level process can return to USER
MODE from kernel. So
 *   tf_cs should be USER_CS segment (see memlayout.h)
 *   tf_ds=tf_es=tf_ss should be USER_DS segment
 *   tf_esp should be the top addr of user stack (USTACKTOP)
 *   tf_eip should be the entry point of this binary program (elf->e_entry)
 *   tf_eflags should be set to enable computer to produce Interrupt
 */
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;
```

问题：描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

用户态进程是怎么来的：硬构造了第0个内核线程 idleproc，idleproc 通过 kernel_thread 创建了 第1个内核线程 initproc，initproc 通过 kernel_execve 将应用程序执行码，覆盖到 initproc 用户虚拟内存空间，来创建用户态进程，创建一个用户态进程并加载了应用程序之后，需要让这个应用程序最终在用户态执行起来，这部分主要由 load_icode函数中实现，load_icode函数中的注释清晰地展示了整个经过，主要分为以下六个步骤：

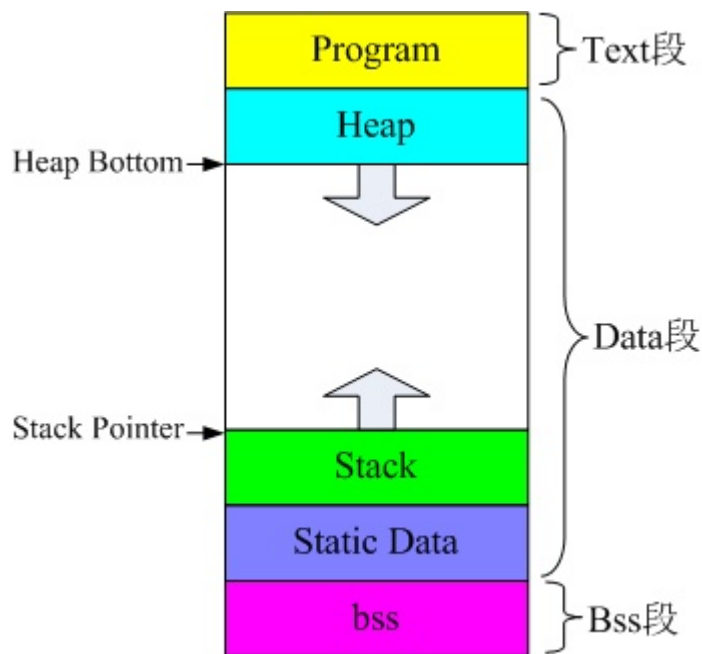
1. 为内存管理的数据结构mm分配空间并初始化。

```
//(1) create a new mm for current process
if ((mm = mm_create()) == NULL) {
    goto bad_mm;
}
```

2. 通过setup_pgdir为用户空间创建页目录，并将内存管理数据结构mm的pgdir设置为页目录的虚地址。

```
//(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
if (setup_pgdir(mm) != 0) {
    goto bad_pgdir_cleanup_mm;
}
```

3. 根据应用程序执行码的起始位置来解析此ELF格式的执行程序,并调用mm_map函数根据ELF格式的执行程序说明的各个段(代码段、数据段、BSS段等)的起始位置和大小建立对应的vma结构,并把vma插入到mm结构中,从而标明了用户进程的合法用户态虚拟地址空间;



bss段 (Block Started by Symbol segment) 是用来存放程序中未初始化的全局变量的一块内存区域。

text段: 用于存放程序代码的区域，编译时确定，只读。

data段: 用于存放在编译阶段(而非运行时)就能确定的数据，可读可写。即静态存储区，赋了初值的全局变量、常量和静态变量都存放在这个域。

接下来将解析已经被载入内存的ELF格式的用户代码。解析ELF header，找到用户程序中program section headers。随后通过调用mm_map将不同段的起始地址和长度记录到虚拟内存空间管理的数据结构vma中去。接下来根据program section的header中的信息，找到每个program section，并将其中的内容拷贝到用户进程的内存中（包括BSS section和TEXT/DATA section）。

```
//(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 binary program (ELF format)
struct elfhdr *elf = (struct elfhdr *)binary;
//(3.2) get the entry of the program section headers of the binary 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_INVALID 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_INVALID 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;
    }
    /* ..... */
    //(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 binary 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) {
        /* ..... */
    }
    while (start < end) {
        /* ..... */
    }
}
}

```

4. 接下来通过调用mm_map函数为用户进程的user stack分配空间。

```

//(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. 建立用户进程的内存管理数据结构mm中的内容，并在进程控制块中记录下用户进程的页目录地址，将用户进程的页目录地址赋给CR3寄存器。

```
//(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. 最后清空原来的中断帧，建立新的中断帧，具体代码即为练习一最开始给出的部分。通过iret指令从内核栈中弹出中断帧恢复各种段寄存器的值。这时段寄存器已经指向特权级为3的段，也就说完成了到用户进程的切换。

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。

```
/* LAB5:EXERCISE2 YOUR CODE
 * replicate content of page to npage, build the map of phy addr of nage with the linear
 * addr start
 * page2kva(struct Page *page): return the kernel virtual 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
 */
// 找到父进程的页虚拟内存地址和子进程的页虚拟内存地址，将父进程的页拷贝到子进程的页
void * kva_src = page2kva(page);
void * kva_dst = page2kva(npage);
memcpy(kva_dst, kva_src, PGSIZE);
ret = page_insert(to, npage, start, perm);
assert(ret == 0);
```

问题：简要说明如何设计实现“Copy on Write 机制”，给出概要设计。

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。

实现时，在fork一个进程时，可以省去 load_icode 中创建新页目录的操作，而是直接将父进程页目录的地址赋给子进程，为了防止误操作以及辨别是否需要复制，应该将尚未完成复制的部分的访问权限设为只读。

当执行读操作，父进程和子进程均不受影响。但当执行写操作时，会发生权限错误（因为此时的访问权限为只读）。这时候会进入到page fault的处理中去，在page fault的处理中，如果发现错误原因读/写权限问题，而访问的段的段描述符权限为可写，便可以知道是由于使用COW机制而导致的，这时再将父进程的数据段、代码段等复制到子进程内存空间上即可。

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）

系统调用共用一个中断号（即代码中的T_SYSCALL）。当发生中断或异常后，会进入到中断服务例程中去，最终在trap_dispatch函数中调用syscall函数，并通过系统调用号选择应该执行函数sys_fork/exec/wait/exit中的一个，这些函数会解析系统调用时传入的参数，并将参数传递给do_fork/execl/wait/exit执行具体操作。

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？

do_fork: sys_fork的相关函数。在该函数中，首先要为子进程创建进程控制块，设置好进程控制块中的上下文的中断帧等信息，为子进程创建用户栈、内核栈等。随后通过wakeup_proc函数将子进程设置为RUNNABLE。之后该函数给父进程返回子进程的pid，给子进程返回0。随后在ucore循环执行进程调度schedule时，就会将子进程考虑进去。

```
/* @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;
    //为子进程创建进程控制块
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    assert(current->wait_state == 0);
    //设置好进程控制块中的上下文的中断帧等信息
    //为子进程创建用户栈、内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
```



```

        hash_proc(proc);
        set_links(proc);

    }
    local_intr_restore(intr_flag);
    //将子进程设置为RUNNABLE
    wakeup_proc(proc);

    ret = proc->pid;
fork_out:
    return ret;

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

```

do_execve: sys_exec的相关函数。sys_exec不创建新进程，而是用新的内容覆盖原来的进程内存空间。在do_execve中，使用exit_mmap、put_pgdir、mm_destroy来删除并释放掉当前进程内存空间的页表信息、内存管理信息。随后通过load_icode将新的用户程序从ELF文件中加载进来执行。如果加载失败，则调用do_exit退出当前进程。执行sys_exec后，当前进程的状态保持不变。

```

// 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);
        // 删除当前进程的内存空间里的内容
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    int ret;
}

```

```

// 调用load_icode加载新的进程内容
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);
}

```

do_wait: sys_wait的相关函数。在该函数中，循环查看子进程的状态，直到一个正在等待的子进程的状态变成Zombie状态，这时完成这个子进程的剩余资源回收工作，释放子进程的空间。

```

// do_wait - wait one OR any children with PROC_ZOMBIE state, and free memory space
// of kernel stack
//          - 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 -EINVAL;
        }
    }

    struct proc_struct *proc;
    bool intr_flag, haskid;
    // 循环询问正在等待的子进程的状态，直到有子进程状态变为ZOMBIE。
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;
}
// 如果发现一个子进程变成了ZOMBIE, 则释放该子进程剩余的资源。
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);
        remove_links(proc);
    }
    local_intr_restore(intr_flag);
    put_kstack(proc);
    kfree(proc);
    return 0;
}

```

do_exit: sys_exit的相关函数。退出时, 首先释放掉该进程占用的一部分内存(还有一部分可能由父进程释放)。然后将该进程标记为僵尸进程。如果它的父进程处于等待子进程退出的状态, 则唤醒父进程, 将自己的子进程交给initproc处理, 并进行的进程调度。

在一个进程调用了exit之后, 该进程并非马上就消失掉, 系统会把该进程的状态变成Zombie, 然后给上一定的时间等着父进程来收集其退出信息, 因为可能父进程正忙于别的事情来不及收集, 所以, 使用Zombie状态表示进程退出了, 正在等待父进程收集信息中。在Linux进程的5种状态中, 僵尸进程是非常特殊的一种, 它已经放弃了几乎所有内存空间, 没有任何可执行代码, 也不能被调度, 仅仅在进程列表中保留一个位置, 记载该进程的退出状态等信息供其他进程收集。

```

// 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;

```

```

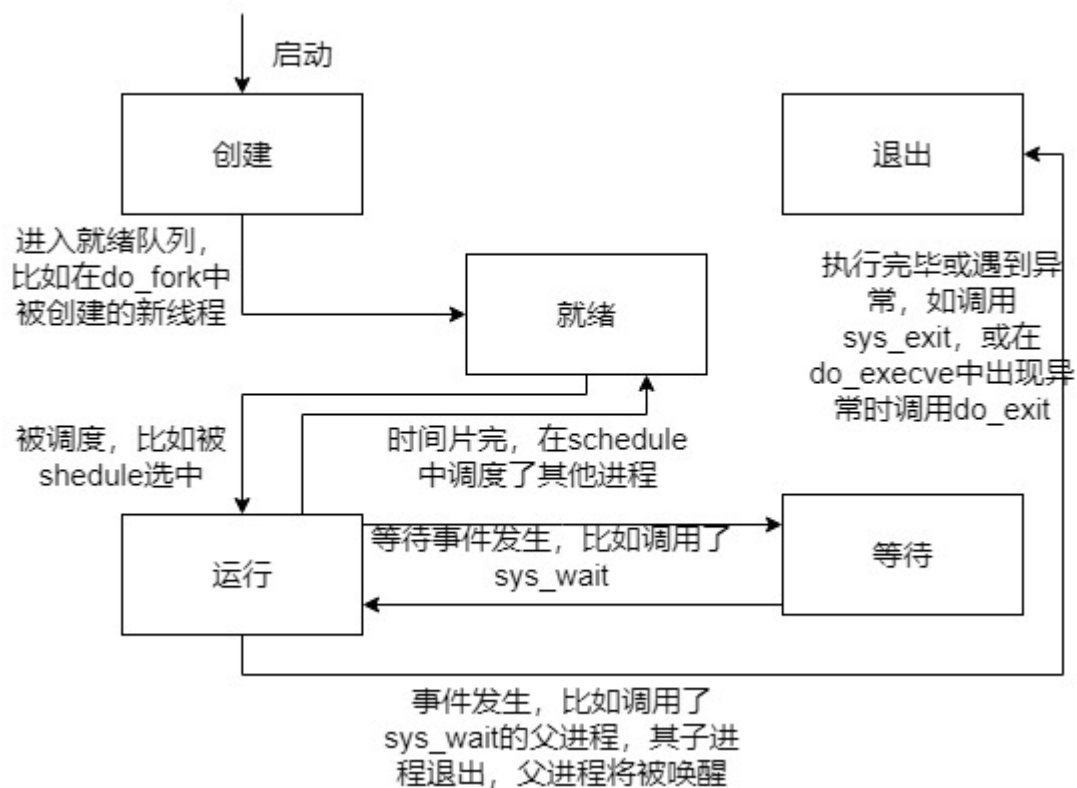
// 删除当前进程的内存空间里的内容
if (mm != NULL) {
    lcr3(boot_cr3);
    if (mm_count_dec(mm) == 0) {
        //删除并释放掉当前进程内存空间的页表信息、内存管理信息
        exit_mmap(mm);
        put_pgdir(mm);
        mm_destroy(mm);
    }
    current->mm = NULL;
}
// 记录当前进程的退出编码，并标记为僵尸进程
current->state = PROC_ZOMBIE;
current->exit_code = error_code;

bool intr_flag;
struct proc_struct *proc;
local_intr_save(intr_flag);
{
    proc = current->parent;
    // 如果当前进程的父进程处于等待子进程退出状态，则将父进程设置为RUNNABLE
    if (proc->wait_state == WT_CHILD) {
        wakeup_proc(proc);
    }
    // 如果当前进程有子进程，则将子进程设置为initproc的子进程，并完成子进程中处于僵尸状态的
    // 进程的最后的回收工作
    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);
// 执行进程调度
schedule();
panic("do_exit will not return!! %d.\n", current->pid);
}

```

- 请给出ucore中一个用户态进程的执行状态生命周期图（执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）



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

```

000a: I am '011'
0009: I am '010'
0020: I am '0101'
001f: I am '0100'
001e: I am '0111'
001d: I am '0110'
001c: I am '0001'
001b: I am '0000'
001a: I am '0011'
0019: I am '0010'
0018: I am '1101'
0017: I am '1100'
0016: I am '1111'
0015: I am '1110'
0014: I am '1001'
0013: I am '1000'
0012: I am '1011'
0011: I am '1010'
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:461:
initproc exit.

```

参考文章：

https://www.haolovej.com/post/uCore_OS.html#toc-746

<https://www.jianshu.com/p/6652345fe969>

<https://zhuanlan.zhihu.com/p/28659560>

https://github.com/chyyuu/ucore_os_lab/pull/58/commits/852d7074ff9109ffe0b3956cd5292d60fa139a7d