lab5实验报告

练习0: 填写已有实验

本次代码需要对lab4中的一些函数进行相应的修改，然后才能进行这一lab的实验。

lab4的alloc_pages()函数中，由于对结构体新增了变量wait_state和孩子进程指针*cptr,兄弟进程指针* yptr和* optr，因此相应的需要对这些变量进行初始化。

```
static struct proc_struct *
alloc_proc(void)
{
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL)
    {
        // LAB5 YOUR CODE : (update LAB4 steps)
        /*
         * below fields(add in LAB5) in proc_struct need to be initialized
         *       uint32_t l;                         // waiting state
         *       struct proc_struct *cptr, *yptr, *optr;     // relations betwee
         */
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0; //设置等待状态
        proc->cptr = proc->optr = proc->yptr = NULL; //设置孩子进程指针和兄弟进程指
    }
    return proc;
}
```

在do_fork()中也需要进行一定的修改，fork时需要保证wait_state变量为0，同时还需要设置前面提到的几个指针变量。

```c
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;
    // LAB5 YOUR CODE : (update LAB4 steps)
    // TIPS: you should modify your written code in lab4(step1 and step5), not a
    /* Some Functions
     *    set_links:  set the relation links of process.  ALSO SEE: remove_links
     *    -------------------
     *    update step 1: set child proc's parent to current process, make sure c
     *    update step 5: insert proc_struct into hash_list && proc_list, set the
     */

    if ((proc = alloc_proc()) == NULL)
    {
        goto bad_fork_cleanup_proc;
    }
    proc->parent = current; // 设置fork进程的父进程为当前进程
    assert(current->wait_state == 0); //保证wait_state值为0
    if ((ret = setup_kstack(proc)) == -E_NO_MEM)
    {
        goto bad_fork_cleanup_kstack;
    }
    copy_mm(clone_flags, proc);
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc); // 设置进程关系
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;

fork_out:
    return ret;

```

```
48 bad_fork_cleanup_kstack:
49     put_kstack(proc);
50 bad_fork_cleanup_proc:
51     kfree(proc);
52     goto fork_out;
53 }
```

练习1: 加载应用程序并执行（需要编码）

补充编程部分

在 `load_icode` 的第6步，我们建立了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，从而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。

1. `tf->gpr.sp` ，用户栈指针，将其设置为用户栈的顶部地址 `USTACKTOP` 。这样，在用户程序执行时，栈将从栈顶向下增长。
2. `tf->epc` ，用户程序的入口点，将其设置为 ELF 文件的入口地址 `elf->e_entry` 。这样，在用户程序开始执行时，将从该地址开始执行。
3. `tf->status` ，用户程序的状态，将其设置为 `sstatus` 的值，并清除 `SSTATUS_SPP` 和 `SSTATUS_SPIE` 标志位。 `SSTATUS_SPP` 表示用户程序的特权级，清除该标志位将其设置为用户模式。 `SSTATUS_SPIE` 表示之前的 SPIE（Supervisor Previous Interrupt Enable）状态，清除该标志位将其设置为关闭。

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。

简略的来说，有以下过程：

- 创建一个用户态进程并且加载了可执行程序后,调度器schedule调用 proc_run();
- 在proc.c中的proc_run()，设置指针 current 为要切换的进程的 PCB， 并加载该进程的内核栈和页目录表;
- 调用 switch_to ，由于当前进程的 context 其中的 epc 被设置为 forkret, 因此 switch_to ret 后会跳转到 forkret 处, forkret 又会将栈设置为当前进程的trapframe 然后跳到 _trapret, 此时 _trapret 会根据当前进程的trapframe 恢复上下文
- 最后,退出中断 sret 从系统调用的函数调用路径, 返回切换到用户进程第一句语句 处开始执行

下面详细说明一下用户态进程被选择占用CPU执行进入RUNNING态到具体执行应用程序第一条指令的经过。

首先需要注意的是，ucore的RUNNING态没有进行切换，而是沿用RUNNABLE态，在RUNNING和RUNNABLE态进行切换时，不进行进程状态变量值的修改。

- 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；
- 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，这时会进一步转发给函数do_execve():

```
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);
}
```

在do_execve()函数中，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；

```
int do_execve(const char *name, size_t len, unsigned char *binary, size_t size)
  struct mm_struct *mm = current->mm; //获取当前进程的内存地址
  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);
  //为加载新的执行码做好用户态内存空间清空准备
  if (mm != NULL) {
    lcr3(boot_cr3); //设置页表为内核空间页表
    if (mm_count_dec(mm) == 0) { //如果没有进程再需要此进程所占用的内存空间
      exit_mmap(mm); //释放进程所占用户空间内存和进程页表本身所占空间
      put_pgdir(mm);
      mm_destroy(mm);
    }
    current->mm = NULL; //把当前进程的 mm 内存管理指针为空
  }
```

```
22    int ret;
23    // 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。这里涉及到读 ELF 格式的文件，
24    if ((ret = load_icode(binary, size)) != 0) { //调用load_icode函数
25      goto execve_exit;
26    }
27    set_proc_name(current, local_name); return 0;
28    execve_exit:
29    do_exit(ret);
30    panic("already exit: %e.\n", ret);
31  }
32
```

其中load_icode函数就是本练习需要我们编写的函数。

该函数主要完成的工作如下：

1. 调用 mm_create 函数来申请进程的内存管理数据结构 mm 所需内存空间,并对 mm 进行初始化；
2. 调用 setup_pgdir 来申请一个页目录表所需的一个页大小的内存空间，并把描述 ucore 内核虚空间映射的内核页表(boot_pgdir 所指)的内容拷贝到此新目录表中，最后让 mm->pgdir 指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间;
3. 根据可执行程序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 格式执行程序的各个段(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mm结构中，表明这些是用户进程的合法用户态虚拟地址空间;
4. 根据可执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；
5. 需要给用户进程设置用户栈，为此调用 mm_mmap 函数建立用户栈的 vma 结构,明确用户栈的位置在用户虚空间的顶端，大小为 256 个页，即1MB，并分配一定数量的物理内存且建立好栈的虚地址<–>物理地址映射关系；
6. 至此，进程内的内存管理 vma 和 mm 数据结构已经建立完成，于是把 mm->pgdir 赋值到 cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的 init 已经被 exit 的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；
7. 先清空进程的中断帧,再重新设置进程的中断帧，使得在执行中断返回指令 iret 后，能够让cpu转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。具体的设置过程我们已经在上面的补充编程部分提到。

在执行完 do_exec 函数之后，进行正常的中断返回的流程，同时跳转到应用程序的入口处，并具体执行第一条指令。

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。

这个函数的作用是将一个进程 A 的内存内容复制到另一个进程 B。它会先对输入参数进行一些断言检查，以确保输入的地址范围和对齐方式是有效的。然后，按页的单位进行内存内容的复制。

首先，通过从进程 A 的页目录表中获取页表项来找到进程 A 的物理页。然后，通过从进程 B 的页目录表中获取页表项来检查对应的页是否存在，并在必要时分配一个新的页表。如果进程 A 的物理页存在，它将复制该页的内容到一个新的物理页中，并使用 `page_insert` 函数将新的物理页与进程 B 的线性地址建立映射关系。重复上述步骤，直到复制范围结束或者遇到错误。

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（COW）是一种优化策略，通常用于处理进程之间的内存共享。以下是大致的 COW 设计步骤：

1. **内存映射：** 首先，操作系统会将同一文件映射到多个进程的虚拟地址空间，通过内存映射（`mmap`）来实现，确保这些进程可以共享相同的物理内存页。
2. **标记为只读：** 这些共享的内存页被标记为只读，这样，如果有进程尝试写入这些页，就会触发一个页错误（Page Fault）。
3. **写入触发：** 当某个进程尝试写入一个被标记为只读的内存页时，会触发页错误。
4. **异常处理：** 可以在trap.c中新增写入触发的页错误，执行异常处理程序。
5. **复制页面：** 在异常处理程序中，操作系统为触发写入的进程创建一个新的物理内存页的拷贝（copy），这样，写入进程将拥有自己的内存页，而其他进程仍然共享原始的只读页。
6. **更新页表：** 操作系统更新进程的页表，将原始的只读页替换为新创建的可写页。
7. **继续执行：** 操作系统返回到触发写入的进程，并重新执行触发写入的指令。

通过这种方式，每个进程都可以独立地修改它们自己的内存拷贝，而不会影响其他进程。这种策略在处理共享内存时，可以提高效率，避免不必要的复制，直到写入操作发生。

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）

系统调用的实现

在ucore中定义了一些系统调用的编号，用于标识和处理不同的系统调用请求。

```
#define SYS_exit            1
#define SYS_fork            2
#define SYS_wait            3
#define SYS_exec            4
#define SYS_clone           5
#define SYS_yield           10
#define SYS_sleep           11
#define SYS_kill            12
#define SYS_gettime         17
#define SYS_getpid          18
#define SYS_brk             19
#define SYS_mmap            20
#define SYS_munmap          21
#define SYS_shmem           22
#define SYS_putc            30
#define SYS_pgdir           31
```

当发生系统调用时，通过内联汇编进行 `ecall` 环境调用，并产生一个trap, 进入S mode进行异常处理。

```
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_ECA
          tf->epc += 4;
          syscall();
          break;
       }
 }
```

在这个异常处理函数 `exception_handle` 中，接受一个指向 `trapframe` 结构体的指针 `tf` 作为参数。函数的主要作用是根据 `tf->cause` （异常原因）的值进行处理。在这段代码中，只处理了 `CAUSE_USER_ECALL` （用户环境调用）的情况。通过 `tf->cause` 的值判断当前异常是来自用户模式的环境调用（ecall），如果是用户环境调用，将 `tf->epc` （产生异常的指令位置）加上 4，使 `sepc` 寄存器指向产生异常的指令的下一条指令。这样做是为了避免陷入无限循环，因为如果 `sepc` 寄存器不被更新，它将指向 ecall 指令，导致重复触发异常。调用 `syscall` 函数进行系统调用处理。该函数将根据系统调用号和参数在内核中执行相应的操作。

```
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,
    [SYS_pgdir]             sys_pgdir,
};

```

这里定义了一个函数指针数组 `syscalls`，用于存储系统调用的函数指针。每个系统调用的编号作为数组索引，并将对应的函数指针初始化为相应的系统调用函数。其中存储系统调用的函数指针。每个系统调用的编号作为数组索引，对应的函数指针被初始化为相应的系统调用函数。

`NUM_SYSCALLS` 宏：它表示系统调用函数指针数组 `syscalls` 的大小，即数组中元素的个数。通过计算 `sizeof(syscalls) / sizeof(syscalls[0])` 来获取数组的大小。

```
#define NUM_SYSCALLS        ((sizeof(syscalls)) / (sizeof(syscalls[0])))

void
syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0;
    if (num >= 0 && num < NUM_SYSCALLS) {
        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);
            return ;
        }
    }
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, current->pid, current->name);
}

```

`syscall` 函数是一个系统调用处理函数，用于根据传入的系统调用编号执行相应的系统调用函数。函数首先获取当前进程的 `trapframe` 结构体指针 `tf`。然后，根据 `tf->gpr.a0` （a0 寄存器保

存的系统调用编号）判断系统调用编号是否有效。如果有效，检查对应的函数指针是否非空，如果非空，则将参数从寄存器中取出并传递给对应的系统调用函数进行处理。最后，将系统调用函数的返回值存储在 `tf->gpr.a0` 中，并返回。

如果系统调用编号无效，或者对应的系统调用函数指针为空，则调用 `print_trapframe` 函数打印当前进程的 `trapframe` 信息，并通过 `panic` 函数抛出一个错误，指示传入的系统调用编号未定义。

fork

`fork` 的执行流程可以分为用户态和内核态两个阶段。

首先是用户态阶段，在父进程中，用户态代码调用 `fork` (ulib.c) 系统调用，触发从用户态切换到内核态，用户态程序的状态（包括寄存器值、栈指针等）被保存到当前进程的 `trapframe` 结构体中，此时用户态程序的执行权转交给内核态。

在进入内核态阶段后，操作系统内核根据系统调用号识别到 `fork` 系统调用，内核态代码执行 `sys_fork` 函数，负责为子进程创建并初始化一个新的 `proc_struct` 结构体，并且内核态代码调用 `do_fork` 函数进行实际的进程复制操作。

```
1 static int
2 sys_fork(uint64_t arg[]) {
3     struct trapframe *tf = current->tf;
4     uintptr_t stack = tf->gpr.sp;
5     return do_fork(0, stack, tf);
6 }
```

在 `do_fork` 函数中，内核态代码为子进程分配内核栈空间、复制/共享父进程的内存空间、设置子进程的 `trapframe` 和内核栈，分配唯一的进程 ID（PID），并将子进程设置为可运行状态。内核态代码返回到 `sys_fork` 函数，并将子进程的 PID 作为返回值传递给用户态。

用户态程序的执行权再次被恢复，可以根据 `fork` 的返回值判断当前是在父进程还是在子进程中。在父进程中，用户态代码继续执行后续的指令，在子进程中， `fork` 的返回值为 0，可以通过条件判断执行不同的逻辑。

exec

`exec` 的执行流程可以分为用户态和内核态两个阶段。

在用户态阶段，用户态程序调用 `exec` 系统调用，传递相应的参数，程序的执行权转交给内核态。

```
1 static int
2 sys_exec(uint64_t arg[]) {
3     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];
    return do_execve(name, len, binary, size);
}
```

在内核态，操作系统内核根据系统调用号识别到 `exec` 系统调用，内核态代码执行 `sys_exec` 函数，负责处理 `exec` 系统调用该函数首先从用户传递的参数中获取文件路径、参数列表等信息，并将它们传递给 `do_execve` 函数， `do_execve` 函数进行实际的执行操作，包括解析可执行文件、创建新的内存空间、加载程序代码和数据等。最后，内核态代码根据执行结果返回相应的错误码或成功标志给用户态。

wait

用户程序调用 `wait` 系统调用，传递相应的参数（如子进程的进程ID、存储退出码的指针等）之后执行权转交给内核态。

```
static int
sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
    return do_wait(pid, store);
}
```

在内核态，操作系统内核根据系统调用号识别到 `wait` 系统调用，内核态代码执行 `sys_wait` 函数，负责处理 `wait` 系统调用。 `sys_wait` 函数从用户传递的参数中获取子进程的进程ID和存储退出码的指针，并调用 `do_wait` 函数进行实际的等待操作， `do_wait` 函数在内核态执行等待子进程结束的操作，包括查找子进程、等待子进程结束、处理子进程的退出状态等，最后内核态代码根据执行结果返回相应的错误码或成功标志给用户态。

```
// do_wait - wait one OR any children with PROC_ZOMBIE state, and free memory sp
//         - 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;
        }
    }

```

```
13      struct proc_struct *proc;
14      bool intr_flag, haskid;
15  repeat:
16      haskid = 0;
17      if (pid != 0) {
18          proc = find_proc(pid);
19          if (proc != NULL && proc->parent == current) {
20              haskid = 1;
21              if (proc->state == PROC_ZOMBIE) {
22                  goto found;
23              }
24          }
25      }
26      else {
27          proc = current->cptr;
28          for (; proc != NULL; proc = proc->optr) {
29              haskid = 1;
30              if (proc->state == PROC_ZOMBIE) {
31                  goto found;
32              }
33          }
34      }
35      if (haskid) {
36          current->state = PROC_SLEEPING;
37          current->wait_state = WT_CHILD;
38          schedule();
39          if (current->flags & PF_EXITING) {
40              do_exit(-E_KILLED);
41          }
42          goto repeat;
43      }
44      return -E_BAD_PROC;
45
46  found:
47      if (proc == idleproc || proc == initproc) {
48          panic("wait idleproc or initproc.\n");
49      }
50      if (code_store != NULL) {
51          *code_store = proc->exit_code;
52      }
53      local_intr_save(intr_flag);
54      {
55          unhash_proc(proc);
56          remove_links(proc);
57      }
58      local_intr_restore(intr_flag);
59      put_kstack(proc);
```

```
60      kfree(proc);
61      return 0;
62  }
```

exit

`exit` 是一个系统调用，用于终止当前进程的执行。它的执行流程涉及用户态和内核态两个阶段。

用户态阶段，用户程序调用 `exit` 系统调用，传递相应的参数（如退出码），并将执行权转交给内核态。

```
1  static int
2  sys_exit(uint64_t arg[]) {
3      int error_code = (int)arg[0];
4      return do_exit(error_code);
5  }
```

在内核态，操作系统内核根据系统调用号识别到 `exit` 系统调用，内核态代码执行 `sys_exit` 函数，负责处理 `exit` 系统调用。 `sys_exit` 函数从用户传递的参数中获取退出码，并调用 `do_exit` 函数进行实际的退出操作。

`do_exit` 函数在内核态执行，负责实现 `exit` 的具体功能。首先，执行与进程退出相关的清理操作，如释放资源、关闭文件描述符等并将进程状态设置为 `PROC_ZOMBIE` ，表示进程已经终止。此时将退出码存储到进程的数据结构中，供其他进程查询。唤醒父进程，以便父进程能够处理子进程的退出状态并将当前进程的子进程移交给 `init` 进程。调用调度器，切换到其他可运行的进程执行。

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？

用户态程序与内核态程序的交错执行是通过系统调用实现的。当用户态程序调用 系统调用时，它触发了从用户态到内核态的切换。内核态代码执行与用户态代码分开，执行完相关的内核操作后，再切换回用户态，将执行权返回给用户态程序继续执行。

内核态执行结果是通过上下文切换和寄存器的保存与恢复来返回给用户程序的。在用户态与内核态的切换过程中，当前进程的状态（包括寄存器值、栈指针等）被保存到 `trapframe` 结构体中。当内核态完成相关操作后，再将保存的状态恢复到对应的进程上下文中，包括返回值等信息。当用户态程序再次被调度执行时，可以获取到之前保存的状态，包括 返回值，以继续执行相应的逻辑。

请给出ucore中一个用户态进程的执行状态生命周期图（包括执行状态，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）。

在lab代码的proc.c中，给出了一个进程的状态切换流程图，与小组成员总结的结果一致，因此此处以lab代码中的注释为准：

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

------------------------------
process state changing:

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

而有关系统调用时的状态转换等，在上文中已经描述过，此处不再赘述。