Lab5

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0：填写已有实验

LAB4中部分代码修改如下：

1. （proc.c）alloc_proc函数

```
proc->flags = 0;
memset(proc->name,0,PROC_NAME_LEN);
proc->wait_state = 0; //初始化进程等待状态
proc->cptr = proc->optr = proc->yptr = NULL;//进程相关指针初始化
```

进程块增加了子进程指针，young兄弟进程和old兄弟进程指针等信息，以及进程是否处于等待状态，因此还要对这两个成员变量进行初始化。

2. （proc.c）do_fork函数

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

由于本实验给进程添加了子进程指针，young兄弟进程和old兄弟进程指针，因此在set_links函数中对相应进程块进行关系设置，并且把原来的list_add(&proc_list,&(proc->list_link));和nr_process++;也放入该函数实现。

练习1：加载应用程序并执行（需要编码）

- do_execv函数调用load_icode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。补充load_icode的第6步如下：

```
tf->gpr.sp = USTACKTOP;
tf->epc = elf->e_entry;
tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

tf->gpr.sp设置为用户栈顶，即为USTACKTOP；
tf->epc设置为elf->e_entry，是用户应用程序的入口；
tf->status中，与SSTATUS_SPP、SSTATUS_SPIE对应的使能位应当置零；
SSTATUS_SPP：Supervisor Previous Privilege（设置为 supervisor 模式，因为这是一个内核线程）
SSTATUS_SPIE：Supervisor Previous Interrupt Enable（设置为启用中断，因为这是一个内核线程）

- 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:
  首先调用load_icode函数来加载应用程序：为用户进程创建新的mm结构，创建页目录表，创建虚拟内存空间，即往mm结构体添加vma结构，分配内存，并拷贝ELF文件的各个program section到新申请的内存上，为BSS section分配内存，并初始化为全0，分配用户栈内存空间，设置当前用户进程的mm结构、页目录表的地址及加载页目录表地址到cr3寄存器，设置当前用户进程的tf结构。
  load_icode返回到do_exevce，do_execve设置完当前用户进程的名字为“exit”后也返回了。这样一直原路返回到__alltraps函数时，接下来进入__trapret函数。__trapret函数先将栈上保存的tf的内容pop给相应的寄存器，然后跳转应用程序的入口（exit.c文件中的main函数），特权级也由内核态跳转到用户态（sret），接下来就开始执行用户程序。

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，补充copy_range的实现如下：

```
// 获取源页面所在的虚拟地址（注意，此时的PDT是内核状态下的页目录表）
void * kva_src = page2kva(page);
// 获取目标页面所在的虚拟地址
void * kva_dst = page2kva(npage);
// 页面数据复制
memcpy(kva_dst, kva_src, PGSIZE);
// 将该页面设置至对应的PTE中
ret = page_insert(to, npage, start, perm);
```

设计实现Copy on Write机制

- 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。
- 当进行内存访问时，CPU会根据PTE上的读写位PTE_P、PTE_W来确定当前内存操作是否允许，如果不允许，则缺页中断。我们可以在copy_range函数中，将父进程中所有PTE中的PTE_W置为0，这样便可以将父进程中所有空间都设置为只读。然后使子进程的PTE全部指向父进程中PTE存放的物理地址，这样便可以达到内存共享的目的。

- 当某个进程想写入一个共享内存时，由于PTE上的PTE_W为0，所以会触发缺页中断处理程序。此时进程需要在缺页中断处理程序中复制该页内存，并设置该页内存所对应的PTE_W为1。

前两步的设计比较简单，因此只对do_pgfault进行了详细修改，以下是该函数代码：

```
int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
   // 查找当前虚拟地址所对应的页表项
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }
    // 如果这个页表项所对应的物理页不存在，则
    if (*ptep == 0) {
        // 分配一块物理页，并设置页表项
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else {
        struct Page *page=NULL;
        // 如果当前页错误的原因是写入了只读页面
        if (*ptep & PTE_P) {
            // 写时复制：复制一块内存给当前进程
            cprintf("\n\nCOW: ptep 0x%x, pte 0x%x\n",ptep, *ptep);
            // 原先所使用的只读物理页
            page = pte2page(*ptep);
            // 如果该物理页面被多个进程引用
            if(page_ref(page) > 1)
            {
                // 释放当前PTE的引用并分配一个新物理页
                struct Page* newPage = pgdir_alloc_page(mm->pgdir, addr, perm);
                void * kva_src = page2kva(page);
                void * kva_dst = page2kva(newPage);
                // 拷贝数据
                memcpy(kva_dst, kva_src, PGSIZE);
            }
            // 如果该物理页面只被当前进程所引用,即page_ref等1
            else
                // 则可以直接执行page_insert，保留当前物理页并重设其PTE权限。
                page_insert(mm->pgdir, page, addr, perm);
        }
        else
        {
            // 如果swap已经初始化完成
            if(swap_init_ok) {
                // 将目标数据加载到某块新的物理页中。
                // 该物理页可能是尚未分配的物理页，也可能是从别的已分配物理页中取的
                if ((ret = swap_in(mm, addr, &page)) != 0) {
                    cprintf("swap_in in do_pgfault failed\n");
                    goto failed;
                }
```

```
                // 将该物理页与对应的虚拟地址关联，同时设置页表。
                page_insert(mm->pgdir, page, addr, perm);
            }
            else {
                cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
                goto failed;
            }
        }
        // 当前缺失的页已经加载回内存中，所以设置当前页为可swap。
        swap_map_swappable(mm, addr, page, 1);
        page->pra_vaddr = addr;
    }
    ret = 0;
failed:
    return ret;
}
```

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）

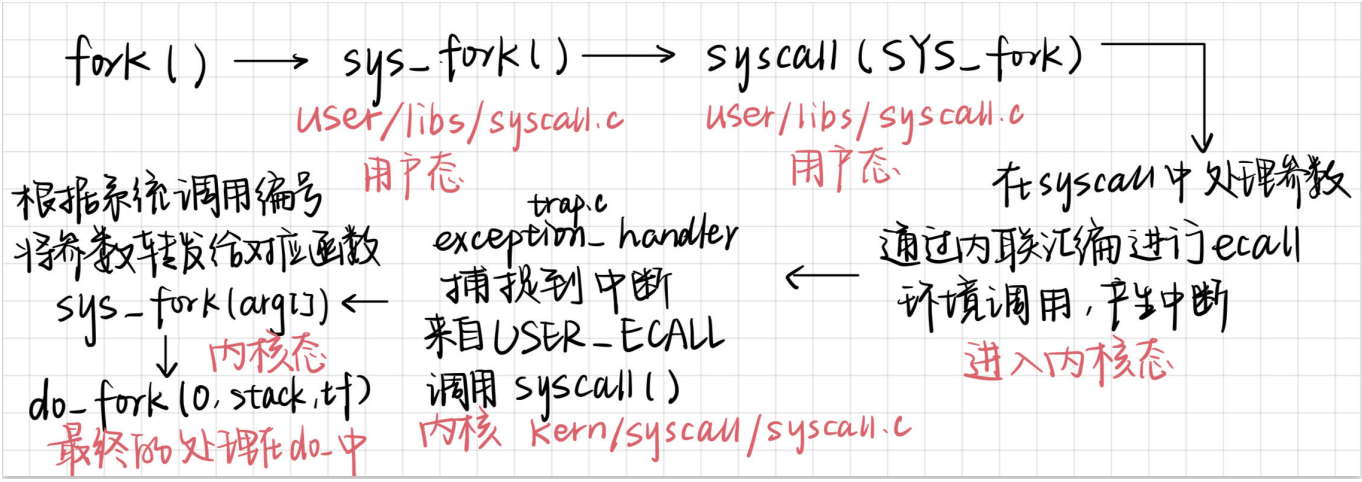
init_main函数创建了一个新的内核线程，用于执行user_main函数。user_main在内核状态下通过kernel_execve函数调用了sys_execve，最终do_execve函数加载了存储在相应位置的用户程序exit.c并执行。

```
//exit.c
if ((pid = fork()) == 0) {
    cprintf("I am the child.\n");
    yield();
    yield();
    yield();
    yield();
    yield();
    yield();
    yield();
    exit(magic);
}
```

exit中执行了fork、yield、exit系统调用。在fork完毕的子进程通过sys_exit成功退出后，waitpid的父进程也随之被唤醒进行对应子进程的回收工作。

```
// 父进程通过wait库函数执行do_wait系统调用，等待子进程退出并回收(waitpid返回值=0代表回收成功)
assert(waitpid(pid, &code) == 0 && code == magic);
```

以fork函数为例，以下是系统调用的执行流程分析：



exec/wait/exit/yield/kill/getpid/putc的执行流程与fork基本一致。 用户态通过内联汇编进行ecall环境调用，产生一个中断，进入S mode进行异常处理。ecall的返回值，即系统调用编号，存在ret中，作为用户态的syscall函数的返回值，继续向前作为sys_fork的返回值返回给fork()函数。

接下来是对各个函数的分析。

- **fork函数**

  fork系统调用最终的处理逻辑在内核中的syscall.c中定义的sys_fork处。sys_fork中调用do_fork函数用于fork复制一个子线程。
  do_fork函数的步骤为：

  1. 调用alloc_proc函数分配一个proc_struct结构体
  2. 调用setup_kstack为子进程分配一个内核栈
  3. 根据clone_flag调用copy_mm来复制或共享mm（内存映射）
  4. 调用copy_thread来设置tf和proc_struct中的context上下文
  5. 将proc_struct插入到hash_list和proc_list中
  6. 调用wakeup_proc使新的子进程变为可运行状态
  7. 使用子进程的pid设置返回值。父进程fork时返回值是子进程的pid，而子进程fork的返回值为0。可以通过判断fork函数的返回值是否为0，编写对应父子进程的不同处理逻辑。

- **exec函数**

  user_main通过kernel_execve函数，在内核态发起了sys_exec的系统调用。sys_exec接受四个参数，arg[0]和arg[1]用于指定所要创建、执行的进程名及其字符串长度；arg[2]和arg[3]用于指定对应程序在内存中的地址以及程序的大小。sys_exec的核心逻辑位于其调用的do_execve中。
  do_execve函数并没有创建新进程，而是用新的内容覆盖原来进程的内存空间。在进行了user_mem_check检查内存空间合法之后，通过exit_mmap、put_pgdir、mm_destroy函数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。

  - 如果set_proc_name的实现不变, 为什么不能直接set_proc_name(current, name)?
    传进来的参数name是一个指向字符串的指针char*，而set_proc_name()函数期望的参数类型是一个字符数组（字符串），不是指针。直接将name传递给set_proc_name()会导致类型不匹配的错误。

- **wait函数**

  sys_wait的核心逻辑在do_wait函数中，循环查看子进程是否处于僵尸态，如果有，则回收其内核栈和进程控制块，释放子进程的空间。
  根据传入参数的不同，执行不同操作。如果pid不为0，代表回收pid对应的僵尸态线程；pid为0，代表回收当前线程的任意一个僵尸态子线程，则对所有子线程进行遍历。

  1. 如果当前线程不存在任何可以回收的子线程，sys_wait会直接返回一个错误码。
  2. 如果调用sys_wait时当前线程能够找到一个符合要求的，且可以立即回收的僵尸态子线程，那么就会将其回收(一次sys_wait只会回收一个子线程)。
  3. 如果调用sys_wait时当前线程存在可以回收的子线程，但是还不处于僵尸态。那么当前线程将进入阻塞态，等待可以回收的子线程退出，成为僵尸态。

- **exit函数**

  当前需要退出的进程会尽可能的将自己持有的包括占用的内存空间的等各种资源释放。

```
struct mm_struct *mm = current->mm;
  if (mm != NULL) {
      lcr3(boot_cr3);
      // 由于mm是当前进程内所有线程共享的，当最后一个线程退出时(mm->mm_count ==
0),需要彻底释放整个mm管理的内存空间
      if (mm_count_dec(mm) == 0) {
          // 解除mm对应一级页表、二级页表的所有虚实映射关系
          exit_mmap(mm);
          // 释放一级页表(页目录表)所占用的内存空间
          put_pgdir(mm);
          // 将mm中的vma链表清空(释放所占用内存)，并回收mm所占物理内存
          mm_destroy(mm);
      }
      current->mm = NULL;
  }
```

  但是由于要执行指令，需要退出的进程的内核栈是无法自己回收的；当前退出进程的进程控制块由于调度的需要，也是无法自己回收掉的。进程退出最终的回收工作需要其父进程来完成。因此，在do_exit函数的最后，当前进程会将自己的状态设置为僵尸态，并且唤醒可能在等待子进程退出而被阻塞的父进程。

```
// 设置当前线程状态为僵尸态，等待父线程回收
current->state = PROC_ZOMBIE;
// 设置退出线程的原因(exit_code)
current->exit_code = error_code;
```

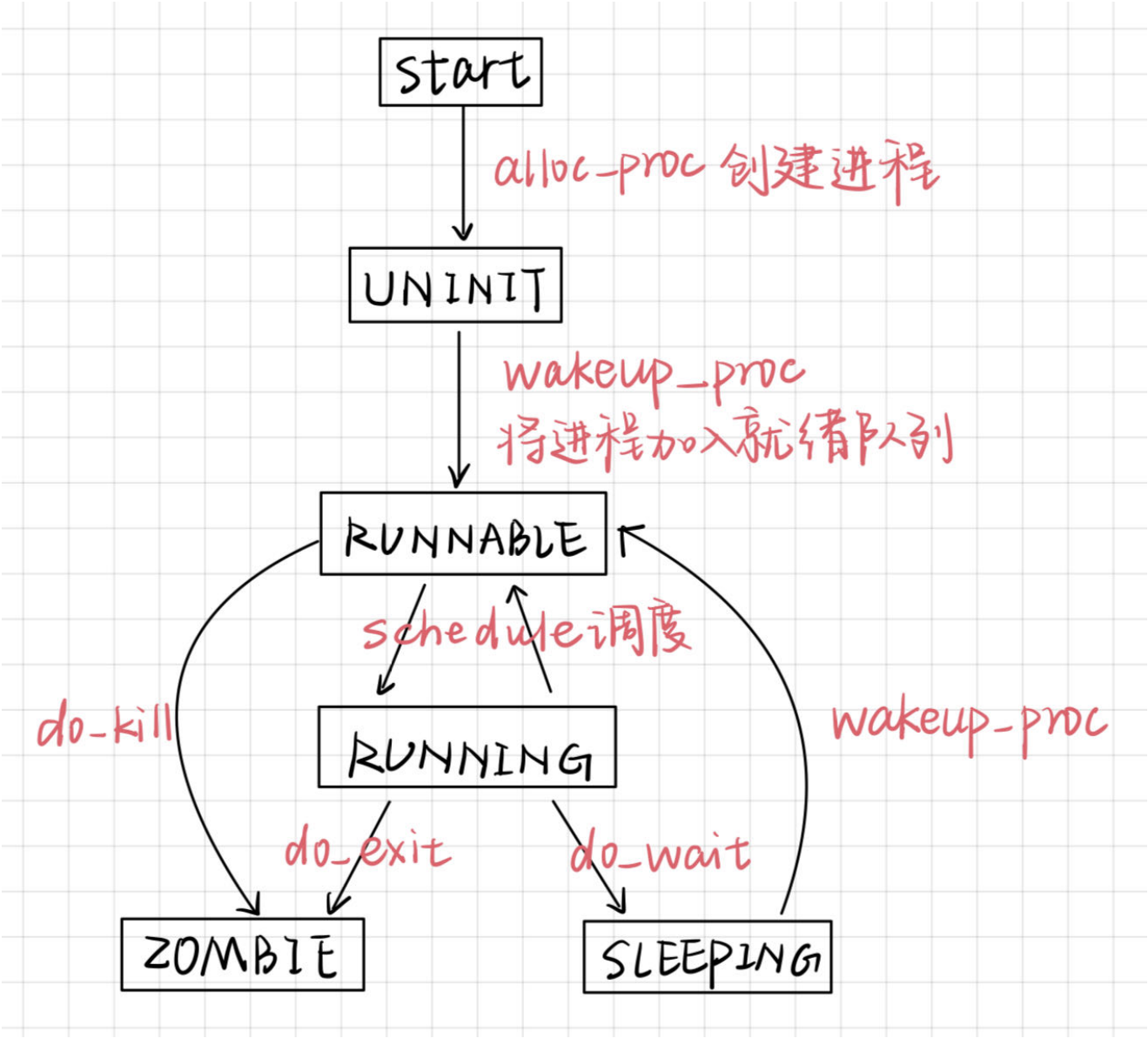
如果当前退出的子进程还有自己的子进程，那么还需要遍历所有的子进程，将它们 移交给内核的第一个进程initproc，使其成为这些子进程的父进程，确保系统在父进程退出时能够正确处理子进程的状态，并维护进程管理结构的完整性，以完成后续的子进程回收工作。

```
// 获取当前进程的父进程
proc = current->parent;

// 如果父进程处于等待子进程状态，则唤醒父进程
if (proc->wait_state == WT_CHILD) {
    wakeup_proc(proc);
}
// 遍历当前进程的所有子进程
while (current->cptr != NULL) {
    proc = current->cptr;
    current->cptr = proc->optr;

    // 设置子进程的父进程为initproc，并加入initproc的子进程链表
    proc->yptr = NULL;
    if ((proc->optr = initproc->cptr) != NULL) {
        initproc->cptr->yptr = proc;
    }
    proc->parent = initproc;
    initproc->cptr = proc;
    // 如果子进程也处于退出状态，唤醒initproc
    if (proc->state == PROC_ZOMBIE) {
        if (initproc->wait_state == WT_CHILD) {
            wakeup_proc(initproc);
        }
    }
}
```

一个用户态进程的执行状态生命周期图如下：



- 内核态执行结果是如何返回给用户程序的？ 用户程序可以通过使用软中断或陷阱指令触发系统调用，将需要执行的操作和参数传递给内核。内核在执行完相应的操作后，将结果存储在指定的寄存器或内存地址中，然后用户程序可以通过读取这些位置来获取执行结果。

重要知识点

1. 僵尸态（Zombie）进程：当一个子进程先于父进程退出时，它会变成僵尸态进程。僵尸态进程的进程控制块（PCB）仍然存在，但占用系统资源较少。僵尸态进程的存在是为了让父进程可以查询子进程的退出状态。

2. 回收子进程：当父进程需要获取子进程的退出状态时，它可以调用等待子进程退出的函数（如本例中的 do_wait 函数）来回收子进程。回收子进程会释放僵尸态进程所占用的资源，并将子进程的退出状态返回给父进程。

3. 父进程与子进程关系：在本例中，父进程可以通过指定 PID 来回收特定的子线程，或者通过将 PID 设置为 0 来回收当前线程的任意一个子线程。在进行回收之前，需要确保指定的进程是当前进程的子进程。

4. 休眠态（Sleeping）：当父进程需要回收子进程，但当前没有可回收的子进程时，父进程会进入休眠态，等待子进程退出。进入休眠态后，父进程会让出 CPU，并进行一次线程调度，直到有子进程退出并唤醒了父进程。

5. 临界区保护：在回收子进程的过程中，需要保护关键代码段，避免并发访问导致的数据竞争问题。本例中使用了关中断的方式来保护关键代码段，确保在执行关键代码时不会被中断打断。

6. 资源释放：回收子进程时，需要释放子进程占用的资源，包括从进程控制块 hash 表和链表中移除子进程的记录，释放子进程的内核栈以及释放子进程的线程控制块结构。

7. 错误处理：在回收子进程的过程中，需要对一些异常情况进行处理，例如，尝试回收不存在的子进程、回收了空闲进程或系统初始化进程等非法操作。在这些情况下，可以选择触发错误，中止程序执行。