

# 实验4

## 个人信息

---

- 数据科学与计算机学院
- 2018级 软工3班
- 18342075
- 米家龙

## 目录

---

### 实验4

个人信息

目录

实验名称

实验目的

实验要求

实验内容

实验环境

2. WSL

实验过程

练习0: 填写已有实验

练习1: 分配并初始化一个进程控制块

查看 `proc.c` / `proc.h` 的注释

完成 `alloc_proc()`

回答问题

练习2: 为新创建的内核线程分配资源

查看 `do_fork()` 函数的注释

完成 `do_fork()` 函数

回答问题

练习3: 分析代码: `proc_run` 函数

进程切换

创建了几个内核进程

语句的作用

实验结果

实验总结

对比 `ucore_lab` 中提供的参考答案, 描述区别

重要并且对应的知识点

实验中没有对应上的知识点

## 实验名称

---

实验5 内核线程管理

## 实验目的

---

- 了解内核线程创建/执行的管理过程
- 了解内核线程的切换和基本调度过程

## 实验要求

---

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

- 内核线程值运行在内核态
- 用户进程会在用户态和内核态交替运行
- 所有内核线程公用 ucore 内核内存空间，不需要为每个内核线程维护单独的内存空间
- 用户进程需要维护各自的用户内存空间

## 实验内容

---

- 练习0：填写已有实验
- 练习1：分配并初始化一个进程控制块（需要编程）
- 练习2：为新创建的内核线程分配资源（需要编程）
- 练习3：阅读代码，理解 `proc_run` 函数和它调用的函数如何完成进程切换的

## 实验环境

---

使用老师提供的 `mooc-os-2015.vdi`，在虚拟机中创建 64 位的 Ubuntu 虚拟机并加载该 vdi，获得了版本为：

```
1 Linux moocos-VirtualBox 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC
  2014 x86_64 x86_64 x86_64 GNU/Linux
```

的虚拟机操作系统

并且使用 vscode 配合 Remote SSH 插件，实现通过远程终端在 windows 环境的对文件的编辑和运行

### 2. WSL

WSL 配置如下：

```
1 root@LAPTOP-QTCGESHO:/mnt/d/blog/work/matrix/step1/001# uname -a
2 Linux LAPTOP-QTCGESHO 4.4.0-19041-Microsoft #1-Microsoft Fri Dec 06 14:06:00 PST
  2019 x86_64 x86_64 x86_64 GNU/Linux
```

## 实验过程

---

### 练习0：填写已有实验

本实验依赖ucore实验1/2/3。请把你做的ucore实验1/2/3的代码填入本实验中代码中有“LAB1”,“LAB2”,“LAB3”的注释相应部分。

需要修改的文件如下：

- kern/debug/kdebug.c

```

kdbg> k!dd? <--> default.pname <--> pframe <--> ttemp_hlcc <--> vmmle <--> ttemp_e
tboodex? t!dd? <--> k!em? debug? <--> k!ddbg?
305 * NOTICE: the calling function's return addr ebp = ss:[ebp+4]
306 * the calling function's return addr = ss:[ebp]
307 */
308 uint32_t ebp = read_ebp(); // 读取 ebp
309 uint32_t ebp = read_ebp(); // 读取 ebp
310 int i = 0;
311 for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++)
312 {
313     printf("ebp = %08X (ebp = %08X)\n", ebp, ebp);
314     uint32_t *arguments = (uint32_t *) ebp + 2; // 得到参数
315     printf("(\"%s\", %08X (\"%s\" = %08X)\n", *arguments, *arguments);
316     printf("    (%08X (%08X + 1), (%08X (%08X + 2), (%08X (%08X + 3));
317     print_debugInfo(ebp - 3);
318     ebp = ((uint32_t *)ebp)[1]; // 将 ebp 指向下一个参数的起始地址
319     ebp = ((uint32_t *)ebp)[0]; // 将 ebp 指向下一个参数的起始地址
320 }
321
322
323
324
325

```

- kern/mm/default\_pmm.c

- kern/mm/pmm.c

```
438: pte_t *  
439: {  
440:     get_pte(pte_t **pgdir, uintptr_t la, bool create) {  
441:     if (0  
442:     }  
443:     // 上述的序言没有动  
444: }  
445:   
446: //get_page - get related Page struct for linear address la using PDT pgdir  
447: struct Page *  
448: get_page(pte_t **pgdir, uintptr_t la, pte_t **ptep_store) {  
449: }  
450:   
451: //page_remove_pte - free an Page struct which is related linear address la  
452: // and clean(invalidate) pte which is related linear address la  
453: //note: PT is changed, so the tlb need to be invalidate  
454: static inline void  
455: page_remove_pte(pte_t **pgdir, uintptr_t la, pte_t **ptep) {  
456:     if (0  
457:     }  
458:     }  
459:     }  
460:     }  
461:     }  
462:     }  
463:     }  
464:     }  
465:     }  
466:     }  
467:     }  
468:     }  
469:     }  
470:     }  
471:     }  
472:     }  
473:     }  
474:     }  
475:     }  
476:     }  
477:     }  
478:     }  
479:     }  
480:     }  
481:     }  
482:     }  
483:     }  
484:     }  
485:     }  
486:     }  
487:     }  
488:     }  
489:     }  
490:     }  
491:     }  
492:     }  
493:     }  
494:     }  
495:     }  
496:     }  
497:     }  
498:     }  
499:     }  
500:     }  
501:     }  
502:     }  
503:     }  
504:     }  
505:     }  
506:     }  
507:     }  
508:     }  
509:     }  
510:     }  
511:     }  
512:     }  
513:     }  
514:     }  
515:     }  
516:     }  
517:     }  
518:     }  
519:     }  
520:     }  
521:     }  
522:     }  
523:     }  
524:     }  
525:     }  
526:     }  
527:     }  
528:     }  
529:     }  
530:     }  
531:     }  
532:     }  
533:     }  
534:     }  
535:     }  
536:     }  
537:     }  
538:     }  
539:     }  
540:     }  
541:     }  
542:     }  
543:     }  
544:     }  
545:     }  
546:     }  
547:     }  
548:     }  
549:     }  
550:     }  
551:     }  
552:     }  
553:     }  
554:     }  
555:     }  
556:     }  
557:     }  
558:     }  
559:     }  
560:     }  
561:     }  
562:     }  
563:     }  
564:     }  
565:     }  
566:     }  
567:     }  
568:     }  
569:     }  
570:     }  
571:     }  
572:     }  
573:     }  
574:     }  
575:     }  
576:     }  
577:     }  
578:     }  
579:     }  
580:     }  
581:     }  
582:     }  
583:     }  
584:     }  
585:     }  
586:     }  
587:     }  
588:     }  
589:     }  
590:     }  
591:     }  
592:     }  
593:     }  
594:     }  
595:     }  
596:     }  
597:     }  
598:     }  
599:     }  
600:     }  
601:     }  
602:     }  
603:     }  
604:     }  
605:     }  
606:     }  
607:     }  
608:     }  
609:     }  
610:     }  
611:     }  
612:     }  
613:     }  
614:     }  
615:     }  
616:     }  
617:     }  
618:     }  
619:     }  
620:     }  
621:     }  
622:     }  
623:     }  
624:     }  
625:     }  
626:     }  
627:     }  
628:     }  
629:     }  
630:     }  
631:     }  
632:     }  
633:     }  
634:     }  
635:     }  
636:     }  
637:     }  
638:     }  
639:     }  
640:     }  
641:     }  
642:     }  
643:     }  
644:     }  
645:     }  
646:     }  
647:     }  
648:     }  
649:     }  
650:     }  
651:     }  
652:     }  
653:     }  
654:     }  
655:     }  
656:     }  
657:     }  
658:     }  
659:     }  
660:     }  
661:     }  
662:     }  
663:     }  
664:     }  
665:     }  
666:     }  
667:     }  
668:     }  
669:     }  
670:     }  
671:     }  
672:     }  
673:     }  
674:     }  
675:     }  
676:     }  
677:     }  
678:     }  
679:     }  
680:     }  
681:     }  
682:     }  
683:     }  
684:     }  
685:     }  
686:     }  
687:     }  
688:     }  
689:     }  
690:     }  
691:     }  
692:     }  
693:     }  
694:     }  
695:     }  
696:     }  
697:     }  
698:     }  
699:     }  
700:     }  
701:     }  
702:     }  
703:     }  
704:     }  
705:     }  
706:     }  
707:     }  
708:     }  
709:     }  
710:     }  
711:     }  
712:     }  
713:     }  
714:     }  
715:     }  
716:     }  
717:     }  
718:     }  
719:     }  
720:     }  
721:     }  
722:     }  
723:     }  
724:     }  
725:     }  
726:     }  
727:     }  
728:     }  
729:     }  
730:     }  
731:     }  
732:     }  
733:     }  
734:     }  
735:     }  
736:     }  
737:     }  
738:     }  
739:     }  
740:     }  
741:     }  
742:     }  
743:     }  
744:     }  
745:     }  
746:     }  
747:     }  
748:     }  
749:     }  
750:     }  
751:     }  
752:     }  
753:     }  
754:     }  
755:     }  
756:     }  
757:     }  
758:     }  
759:     }  
760:     }  
761:     }  
762:     }  
763:     }  
764:     }  
765:     }  
766:     }  
767:     }  
768:     }  
769:     }  
770:     }  
771:     }  
772:     }  
773:     }  
774:     }  
775:     }  
776:     }  
777:     }  
778:     }  
779:     }  
780:     }  
781:     }  
782:     }  
783:     }  
784:     }  
785:     }  
786:     }  
787:     }  
788:     }  
789:     }  
790:     }  
791:     }  
792:     }  
793:     }  
794:     }  
795:     }  
796:     }  
797:     }  
798:     }  
799:     }  
800:     }  
801:     }  
802:     }  
803:     }  
804:     }  
805:     }  
806:     }  
807:     }  
808:     }  
809:     }  
810:     }  
811:     }  
812:     }  
813:     }  
814:     }  
815:     }  
816:     }  
817:     }  
818:     }  
819:     }  
820:     }  
821:     }  
822:     }  
823:     }  
824:     }  
825:     }  
826:     }  
827:     }  
828:     }  
829:     }  
830:     }  
831:     }  
832:     }  
833:     }  
834:     }  
835:     }  
836:     }  
837:     }  
838:     }  
839:     }  
840:     }  
841:     }  
842:     }  
843:     }  
844:     }  
845:     }  
846:     }  
847:     }  
848:     }  
849:     }  
850:     }  
851:     }  
852:     }  
853:     }  
854:     }  
855:     }  
856:     }  
857:     }  
858:     }  
859:     }  
860:     }  
861:     }  
862:     }  
863:     }  
864:     }  
865:     }  
866:     }  
867:     }  
868:     }  
869:     }  
870:     }  
871:     }  
872:     }  
873:     }  
874:     }  
875:     }  
876:     }  
8
```

- kern/trap/trap.c

```

1  #include <default_pmmic>
2  #include <trap.h>
3  static struct gatedesc idt[256] = {0};
4
5  static struct pseudosd idt_pd = {
6      sizeof(idt) - 1, (uintptr_t)idt
7  };
8
9  /* lidt_init - Initialize IDT to each of the entry points in kern/trap/vectors.c */
10 void
11 lidt_init(void)
12 {
13     /* LAH3 VMOR CODE - STIP 2 */
14     /* (1) where are the entry address of each interrupt service routine (ISR)?
15      * All ISRs entry address are stored in _vectors, where is uintptr_t _vectors[]
16      * _vectors[] is in kern/trap/vectors.c which is produced by tool/vector.c
17      * (try "make" command in lab4, then you will find vector.5 in kern/trap DIR)
18      * You can use "extern uintptr_t _vectors[]" to define this extern variable which will
19      * (2) Now you should setup the entries of IDT in Interrupt Description Table (IDT).
20      * Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE macro to setup
21      * (3) After setup the contents of IDT, you will let CPU know where is the IDT by using "lidt"
22      * You don't know the meaning of this instruction? just google it! and check the lib4/sd
23      * notice: the argument of lidt is lidt_pd, try to find it!
24     */
25
26     extern uintptr_t _vectors[]; // 声明 vector
27     int i = 0;
28     for (i = 0; i < 256; i++)
29         // 对于每一个条目进行设置
30         SETGATE(idt[i], 0, GD_TEXT, _vectors[i], DPL_KERNEL);
31     SETGATE(idt[T_SWITCH_TOK], 0, GD_TEXT, _vectors[T_SWITCH_TOK], DPL_KERNEL);
32     lidt(idt_pd); // 加载 idt_pd
33 }

```

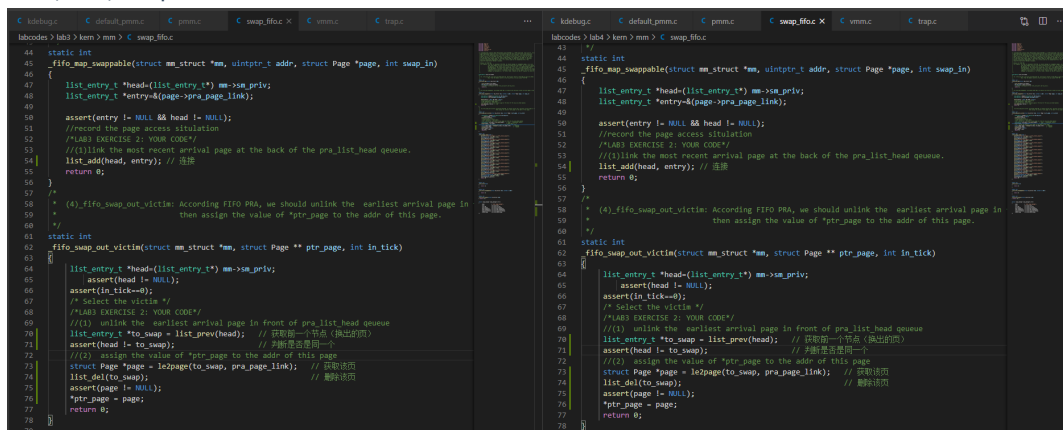
- kern/mm/vmm.c

```

lab0ides % lab2 > kern > mm > > vmnuc
399 #endif
400 // 以下重练习代码
401 ptep = get_pte(m->pgdir, addr, 1); // 获取 ptep
402 if (ptep == NULL) // 如果 ptep 的 pt 不存在
403 {
404     cprintf("ptep isn't existed, get_pte() in do_psfault() failed\n");
405     goto failed;
406 }
407
408 if ("ptep == 0" // 如果 pa 不存在
409     {
410         if (pgdir_alloc_page(m->pgdir, addr, perm) == NULL) // 尝试分配页面且映射
411             cprintf("pa isn't existed, and alloc page failed in do_psfault() failed\n");
412         goto failed;
413     }
414
415 // 以下重练习代码
416 else // 如果 pa 不为空，将页表项清空，尝试填入页面
417 {
418     if (swap_init_ok) // 代表初始化成功
419     {
420         struct Page *page = NULL;
421         ret = swap_in(m, addr, &page); // 换页
422         if (ret != 0) // 如果换页失败
423             cprintf("swap_in in do_psfault failed\n");
424         goto failed;
425     }
426
427     page_insert(m->pgdir, page, addr, perm); // 建立映射，同时通过 perm 设置物理页权限
428     swap_map_swapable(m, addr, page, 1); // 设置页为可换
429     page->prva_vaddr = addr; // 设置页对应的虚拟地址
430 }
431 else // 初始化失败
432 {
433     cprintf("no swap_init_ok but ptep is %x, failed\n", ptep);
434 }

```

- kern/mm/swap\_fifo.c



## 练习1：分配并初始化一个进程控制块

- alloc\_proc函数（位于 kern/process/proc.c 中）负责分配并返回一个新的 struct proc\_struct 结构，用于存储新建的内核线程的管理信息。ucore 需要对这个结构进行最基本的初始化，你需要完成这个初始化过程。
  - 【提示】在 alloc\_proc 函数的实现中，需要初始化的 proc\_struct 结构中的成员变量至少包括：  
state / pid / runs / kstack / need\_resched / parent / mm / context / tf / cr3 / flags / name。
- 请在实验报告中简要说明你的设计实现过程。请回答如下问题：
  - 请说明 proc\_struct 中 struct context context 和 struct trapframe \*tf 成员变量含义和在本实验中的作用是啥？（提示通过看代码和编程调试可以判断出来）

## 查看 proc.c / proc.h 的注释

了解到进程的状态：

状态	含义	相关函数
PROC_UNINIT	未初始化	alloc_proc
PROC_SLEEPING	休眠状态	try_free_pages, do_wait, do_sleep
PROC_RUNNABLE	可运行/正在运行	proc_init, wakeup_proc
PROC_ZOMBIE	将要死亡	do_exit

找到对应的枚举类型

```

1 // process's state in his life cycle
2 enum proc_state {
3     PROC_UNINIT = 0, // 未初始化
4     PROC_SLEEPING, // 休眠
5     PROC_RUNNABLE, // 可运行/正在运行
6     PROC_ZOMBIE, // 将近死亡，等待父进程回收资源
7 };

```

查看结构体 proc\_struct 的代码，了解相关成员变量的作用

```

1 #define PROC_NAME_LEN 15 // 进程名长度
2 #define MAX_PROCESS 4096 // 最大进程数量

```

```

3  #define MAX_PID (MAX_PROCESS * 2) // 最大 pid 数量
4
5  extern list_entry_t proc_list; // 储存变量的列表
6
7  struct proc_struct {
8      enum proc_state state; // 进程状态
9      int pid; // 进程 id
10     int runs; // 运行时间
11     uintptr_t kstack; // 内核栈位置
12     volatile bool need_resched; // 是否需要 CPU 调度
13     struct proc_struct *parent; // 父进程
14     struct mm_struct *mm; // 内存管理字段
15     struct context context; // 用于进程切换
16     struct trapframe *tf; // 当前中断帧
17     uintptr_t cr3; // CR3寄存器: CR3保存页表的物理地址
18     uint32_t flags; // 进程标志
19     char name[PROC_NAME_LEN + 1]; // 进程名
20     list_entry_t list_link; // 进程链表
21     list_entry_t hash_link; // 进程哈希表
22 };

```

查看结构体 `context` 的代码:

```

1  // 为内核的上下文切换保存寄存器
2  // 由于段寄存器是跨内核的上下文常量, 因此不需要保存 %fs 等所有的段寄存器
3  // 保存所有常规寄存器, 从而不需要关心哪些是调用者保存的, 而不是返回 %eax (简化代码)
4  // 上下文布局需和 switch.S 中的代码匹配
5  struct context {
6      uint32_t eip;
7      uint32_t esp;
8      uint32_t ebx;
9      uint32_t ecx;
10     uint32_t edx;
11     uint32_t esi;
12     uint32_t edi;
13     uint32_t ebp;
14 };

```

## 完成 `alloc_proc()`

如果要初始化, 那么需要将大部分成员变量设置为0或者空, 除了初始的 cr3 寄存器需要初始化为页目录表的基址, 完成该函数如下:

```

1  // alloc_proc - alloc a proc_struct and init all fields of proc_struct
2  static struct proc_struct *
3  alloc_proc(void) {
4      struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
5      if (proc != NULL) {
6          proc->state = PROC_UNINIT; // 状态
7          proc->pid = -1; // 给一个无
8          proc->runs = 0; // 未运行过
9          proc->kstack = 0; // 没有内核栈
10         proc->need_resched = false; // 不需要被 CPU 调
11         度
12         proc->parent = NULL; // 没有父进程
13         proc->mm = NULL; // 没有内存管理字段
14         memset(&(proc->context), 0, sizeof(struct context)); // 设置上下文
15         proc->tf = NULL; // 无中断帧

```

```

15         proc->cr3 = boot_cr3;                                // 设置为内核页目录
    表的基址
16         proc->flag = 0;                                        // 设置进程标志
17         memset(proc->name, 0, PROC_NAME_LEN + 1);           // 设置进程名为空
18     }
19     return proc;
20 }

```

## 回答问题

请说明 `proc_struct` 中 `struct context context` 和 `struct trapframe *tf` 成员变量含义和在本实验中的作用是啥？

- `context` 保存前一个进程各个寄存器的状态，用于上下文切换。
  - 当该进程变为 `init` 进程时，保存寄存器状态
  - 当该进程变为 `idle` 进程是，根据 `context` 恢复现场从而继续执行
- `*tf` 是中断帧。
  - 当进程从用户空间跳入**内核空间**时，中断帧记录被中断前的状态。
  - 当该进程跳回内核空间后，需要调整中断帧来恢复对应的寄存器值，从而使得进程继续执行。
  - 和 `context` 相比，中断帧包含了 `context` 的信息，以及**段寄存器、中断号和 err 等信息**。
  - 中断帧一般在**系统调用或中断**时，因为发生了**特权级的转换**。

## 练习2：为新创建的内核线程分配资源

创建一个内核线程需要分配和设置好很多资源。`kernel_thread` 函数通过调用 `do_fork` 函数完成具体内核线程的创建工作。`do_kernel` 函数会调用 `alloc_proc` 函数来分配并初始化一个进程控制块，但 `alloc_proc` 只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。ucore 一般通过 `do_fork` 实际创建新的内核线程。`do_fork` 的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。你需要完成在 `kern/process/proc.c` 中的 `do_fork` 函数中的处理过程。

`do_fork` 的大致步骤包括：

- 调用 `alloc_proc`，首先获得一块用户信息块
- 为进程分配一个内核栈
- 复制原进程的内存管理信息到新进程（但内核线程不必做此事）
- 复制原进程上下文到新进程
- 将新进程添加到进程列表
- 唤醒新进程
- 返回新进程号

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

- 请说明 ucore 是否做到给每个新 fork 的线程一个**唯一的 id**？请说明你的分析和理由。

## 查看 `do_fork()` 函数的注释

获取到部分函数和宏定义和变量：

- 宏/函数
  - `alloc_proc`：创建并初始化一个 `proc` 结构体

- `setup_kstack` : 申请一块大小为 `KSTACKPAGE` 的页作为进程内核堆栈
- `copy_mm` : 进程 `proc` 还是共享当前进程 `current` , 由 `clone_flags` 决定, 如果 `clone_flags & CLONE_VM` 为真, 则共享, 否则复制
- `copy_thread` : 在进程内和堆栈顶设置中断帧, 并设置该进程的内核入口和堆栈
- `hash_proc` : 将进程添加到进程哈希列表
- `get_pid` : 为进程申请一个独一的 `pid`
- `wakeup_proc` : 将进程的状态设置为 `PROC_RUNNABLE` , 唤醒进程
- 变量
  - `proc_list` : 进程集合的列表
  - `nr_process` : 进程集合的数量

分别查看并了解上述定义的作用

## 完成 `do_fork()` 函数

根据注释, 可以得到具体步骤为:

- 使用 `alloc_proc` 初始化进程控制模块
- 使用 `setup_stack` 为子进程分配并初始化内核栈
- 使用 `copy_mm` 根据 `clone_flag` 复制/共享进程内存管理结构
- 使用 `copy_thread` 设置进程在内核正常运行和调度所需的中断帧和上下文
- 将设置好的控制进程模块插入到 `hash_list` 和 `proc_list` 中
- 使用 `wakeup_proc` 将进程设置为就绪状态
- 将返回结果设置为子进程 `pid`

完成代码如下:

```

1  /* do_fork -      parent process for a new child process
2  * @clone_flags: used to guide how to clone the child process
3  * @stack:       the parent's user stack pointer. if stack==0, It means to fork
   a kernel thread.
4  * @tf:          the trapframe info, which will be copied to child process's
   proc->tf
5  */
6  int
7  do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
8      int ret = -E_NO_FREE_PROC;
9      struct proc_struct *proc;
10     if (nr_process >= MAX_PROCESS) {
11         goto fork_out;
12     }
13     ret = -E_NO_MEM;
14     //LAB4:EXERCISE2 YOUR CODE
15
16     //    1. 使用 `alloc_proc` 初始化进程控制模块
17     if ((proc = alloc_proc()) == NULL) // 申请内存失败
18         goto fork_out;
19     //    2. 使用 `setup_stack` 为子进程分配并初始化内核栈
20     proc->parent = current; // 将父进程设置为当前进程
21     if (setup_kstack(proc) != 0) // 如果分配内核栈失败
22         goto bad_fork_cleanup_kstack;
23     //    3. 使用 `copy_mm` 根据 `clone_flag` 复制/共享进程内存管理结构
24     if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
25         goto bad_fork_cleanup_proc;
26     //    4. 使用 `copy_thread` 设置进程在内核正常运行和调度所需的中断帧和上下文
27     copy_thread(proc, stack, tf);

```

```

28     //    5. 将设置好的控制进程模块插入到 `hash_list` 和 `proc_list` 中
29     bool intr_flag;           // 标志
30     local_intr_save(intr_flag); // 屏蔽中断, 并将标志设置为1
31     {
32         proc->pid = get_pid();           // 获取 pid
33         hash_proc(proc);                 // 建立映射
34         nr_processes++;                   // 记录数量增加
35         list_add(&proc_list, &(proc->list_link)); // 进程加入到进程链表中
36     }
37     local_intr_restore(proc); // 恢复中断
38     //    6. 使用 `wakeup_proc` 将进程设置为**就绪**状态
39     wakeup_proc(proc); // 唤醒进程
40     //    7. 将返回结果设置为**子进程 pid**
41     ret = proc->pid;
42
43 fork_out:
44     return ret;
45
46 bad_fork_cleanup_kstack:
47     put_kstack(proc);
48 bad_fork_cleanup_proc:
49     kfree(proc);
50     goto fork_out;
51 }

```

## 回答问题

请说明 ucore 是否做到给每个新 fork 的线程一个**唯一的 id**？请说明你的分析和理由。

查看 `get_pid` 函数的代码

```

1  // get_pid - alloc a unique pid for process
2  static int
3  get_pid(void) {
4      static_assert(MAX_PID > MAX_PROCESS);
5      struct proc_struct *proc;
6      list_entry_t *list = &proc_list, *le;
7      static int next_safe = MAX_PID, last_pid = MAX_PID;
8      if (++ last_pid >= MAX_PID) {
9          last_pid = 1;
10         goto inside;
11     }
12     if (last_pid >= next_safe) {
13     inside:
14         next_safe = MAX_PID;
15     repeat:
16         le = list;
17         while ((le = list_next(le)) != list) {
18             proc = le2proc(le, list_link);
19             if (proc->pid == last_pid) {
20                 if (++ last_pid >= next_safe) {
21                     if (last_pid >= MAX_PID) {
22                         last_pid = 1;
23                     }
24                     next_safe = MAX_PID;
25                     goto repeat;
26                 }
27             }

```



```

28         else if (proc->pid > last_pid && next_safe > proc->pid) {
29             next_safe = proc->pid;
30         }
31     }
32 }
33 return last_pid;
34 }

```

该函数运行的过程为：

- 判断 `MAX_PID` 是否小于 `MAX_PROCESS`，避免文件改动带来的错误
- 声明静态局部变量 `next_safe` 和 `last_pid`，并初始化为 `MAX_PID`
- 将 `last_pid` 置为1，遍历 `[1, MAX_PID]`
  - 当 `last_pid` 未和已知 pid 冲突，则缩小 `next_safe`，此时 `[last_pid, MAX_PID]` 之间是未被使用的 pid
  - 当出现冲突，`last_pid + 1`，检查 `[last_pid, MAX_PID]` 是否合法
    - 如果合法，继续遍历
    - 不合法则表示小于 `last_pid` 已经消耗完，则重新开始扫描 `[last_pid, MAX_PID]`
- 遍历结束时，`last_pid` 为未被使用过的 pid，可以使用

因此可以判断能够给每个 fork 的新线程一个唯一的 id

### 练习3：分析代码： `proc_run` 函数

- 阅读代码，理解 `proc_run` 函数和它调用的函数如何完成进程切换的。
- 请在实验报告中简要说明你对 `proc_run` 函数的分析。并回答如下问题：
  - 在本实验的执行过程中，创建且运行了几个内核线程
  - 语句 `local_intr_save(intr_flag);...local_intr_restore(intr_flag);` 在这里有何作用？请说明理由。

### 进程切换

查看 `proc_run` 函数的代码：

```

1  // proc_run - make process "proc" running on cpu
2  // NOTE: before call switch_to, should load base addr of "proc"'s new PDT
3  void proc_run(struct proc_struct *proc)
4  {
5      if (proc != current)
6      {
7          bool intr_flag;
8          struct proc_struct *prev = current, *next = proc;
9          local_intr_save(intr_flag);
10         {
11             current = proc;
12             load_esp0(next->kstack + KSTACKSIZE);
13             lcr3(next->cr3);
14             switch_to(&(prev->context), &(next->context));
15         }
16         local_intr_restore(intr_flag);
17     }
18 }

```

过程分为三个步骤:

1. 屏蔽中断
2. 修改 `exp0` 和页表项, 并且进行上下文切换
3. 恢复中断

其中调用了三个函数

- `load_esp0()` 修改 `exp0`, 以便在使用中间帧从用户到内核后能够使用不同的内核堆栈
- `lcr3()` 使得进程能够在 CPU 中运行
- `switch_to()` 切换进程

查看 `switch_to()` 的代码:

```
1  .text
2  .globl switch_to
3  switch_to:                # switch_to(from, to)
4
5      # save from's registers
6      movl 4(%esp), %eax     # eax points to from
7      popl 0(%eax)          # save eip !popl
8      movl %esp, 4(%eax)    # save esp::context of from
9      movl %ebx, 8(%eax)    # save ebx::context of from
10     movl %ecx, 12(%eax)   # save ecx::context of from
11     movl %edx, 16(%eax)   # save edx::context of from
12     movl %esi, 20(%eax)   # save esi::context of from
13     movl %edi, 24(%eax)   # save edi::context of from
14     movl %ebp, 28(%eax)   # save ebp::context of from
15
16     # restore to's registers
17     movl 4(%esp), %eax     # not 8(%esp): popped return address already
18                             # eax now points to to
19     movl 28(%eax), %ebp    # restore ebp::context of to
20     movl 24(%eax), %edi    # restore edi::context of to
21     movl 20(%eax), %esi    # restore esi::context of to
22     movl 16(%eax), %edx    # restore edx::context of to
23     movl 12(%eax), %ecx    # restore ecx::context of to
24     movl 8(%eax), %ebx     # restore ebx::context of to
25     movl 4(%eax), %esp     # restore esp::context of to
26
27     pushl 0(%eax)         # push eip
28
29     ret
```

该函数的作用:

- 储存前一个进程的7个寄存器值到 `context`
- 将 `context` 中的值恢复到寄存器

从而实现进程的上下文切换

## 创建了几个内核进程

两个

```
1  // idle proc
2  struct proc_struct *idleproc = NULL;
3  // init proc
4  struct proc_struct *initproc = NULL;
```

并且在 `proc_init` 函数中也能够看到对上述两个线程的初始化

1. `idleproc`，用于完成内核中各个子系统的初始化，然后调度、运行其他进程
2. `initproc`，上一个进程完成后就运行该进程，输出一段字符串

## 语句的作用

用于屏蔽中断和恢复中断，以免在进程切换中出现中断，导致其他进程进行调度

## 实验结果

运行 `make grade` 命令，结果如下

```
mijialong$>make grade
Check VMM: (3.4s)
-check pmm: no $qemu_out
-check page table: no $qemu_out
-check vmm: no $qemu_out
-check swap page fault: no $qemu_out
-check ticks: no $qemu_out
-check initproc: no $qemu_out
Total Score: 0/90
make: *** [grade] Error 1
mijialong$>pwd
/home/moocos/ucore_os_lab/labcodes/lab4
```

然后对比答案，发现自己和答案思路一致，并没有其他有问题的地方

于是尝试在 WSL 上尝试运行，结果如下图

```
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab4# make grade
Check VMM: (2.3s)
-check pmm: OK
-check page table: OK
-check vmm: OK
-check swap page fault: OK
-check ticks: OK
-check initproc: OK
Total Score: 90/90
root@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab4#
```

由于之前出现过使用 VSCode 的 Remote SSH 插件连接虚拟机时因为出现了储存空间不足导致连接失败的情况，因此猜测是虚拟机能够运行 ucore 的空间不足，在对之前的 lab 中执行 `make clean` 命令删除对应的文件和文件夹，再在 lab4 文件夹中执行 `make grade` 命令，能够通过，结果如下图：

```

mijialong$>make grade
Check VMM: (3.2s)
-check pmm: no $qemu_out
-check page table: no $qemu_out
-check vmm: no $qemu_out
-check swap page fault: no $qemu_out
-check ticks: no $qemu_out
-check initproc: no $qemu_out
Total Score: 0/90
make: *** [grade] Error 1
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../lab1
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../lab2
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../lab3
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../../labcodes_answer/
mijialong$>cd lab1_result/
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../lab2_result/
mijialong$>make clean
mijialong$>cd ../lab3_result/
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../lab4_result/
mijialong$>make clean
rm -f -r obj bin
mijialong$>cd ../../labcodes/lab4
mijialong$>make grade
Check VMM: (2.8s)
-check pmm: OK
-check page table: OK
-check vmm: OK
-check swap page fault: OK
-check ticks: OK
-check initproc: OK
Total Score: 90/90
mijialong$>

```

确实是储存空间的问题

执行 `make qemu-nox` , 观察输出, 结果如下, 基本符合预期

```

swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/R [no page found].
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
count is 5, total is 5
check_swap() succeeded!

```

```

++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:348:
    process exit!!.

```

```

stack traceback:
ebp = 0xc030cf98      eip = 0xc01009f4
    arg0 = 0xc010a174      arg1 = 0xc030cfdc
    arg2 = 0x0000015c      arg3 = 0xc030cfcc
    kern/debug/kdebug.c:310: print_stackframe+22
ebp = 0xc030cfc8      eip = 0xc0100d58
    arg0 = 0xc010be09      arg1 = 0x0000015c
    arg2 = 0xc010be1d      arg3 = 0xc012a064
    kern/debug/panic.c:27: __panic+105
ebp = 0xc030cfe8      eip = 0xc0108e5c
    arg0 = 0x00000000      arg1 = 0xc010be9c
    arg2 = 0x00000000      arg3 = 0x00000010
    kern/process/proc.c:348: do_exit+33
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

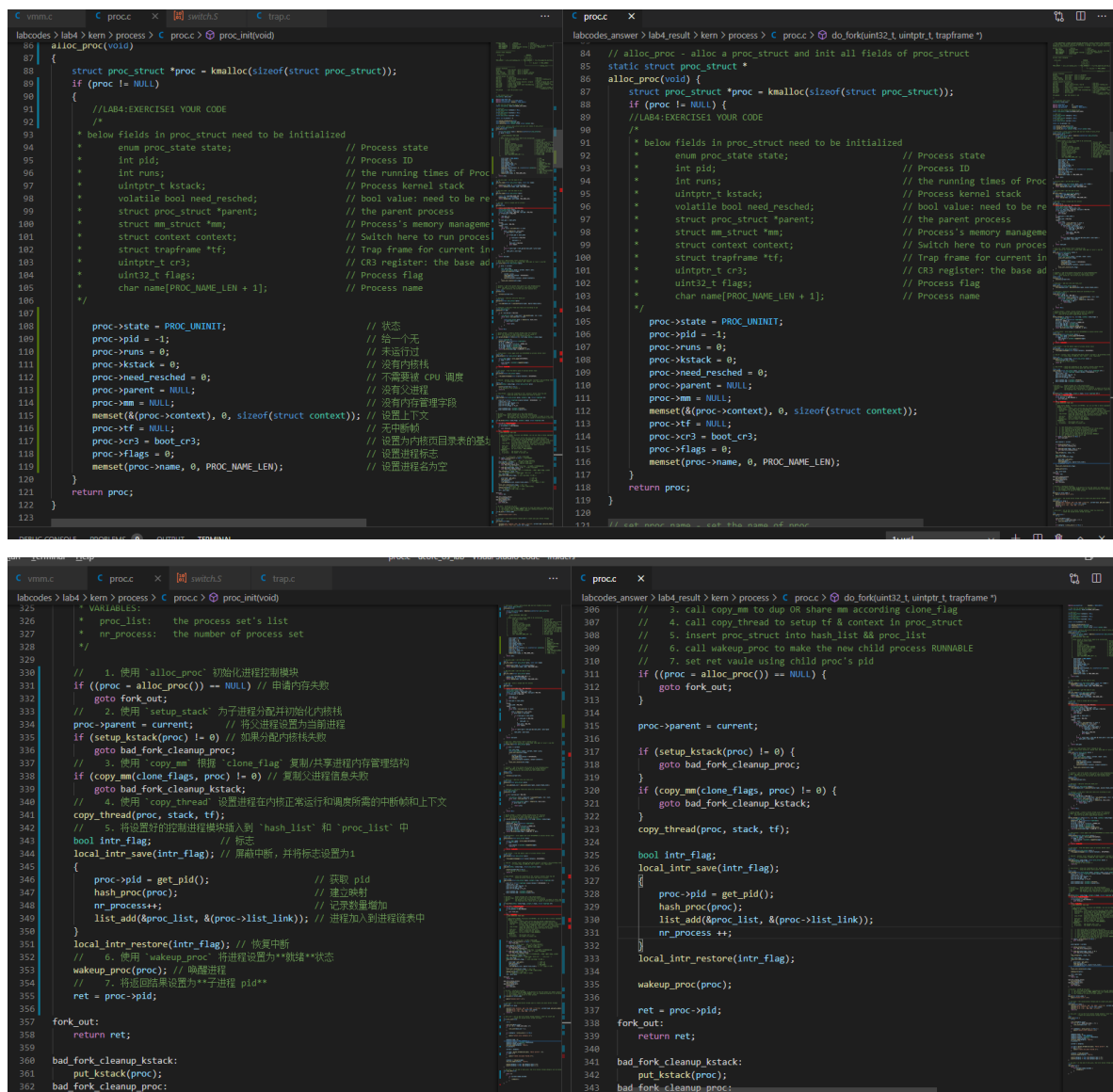
```

## 实验总结

比较简单地了解了进程创建和初始化等相关步骤的直接操作，对进程的调度和生存周期有了更深入地了解；在编程过程中也发现了编程环境的一些问题，并且较为顺利地解决，并且使用了 WSL 来进行验证。

## 对比 ucore\_lab 中提供的参考答案，描述区别

两个函数都没有区别



## 重要并且对应的知识点

### 实验

- 线程控制块的概念以及组成
- 切换不同线程的方法

### 原理：

- 对内核线程的管理
- 对内核线程之间的切换

这两者之间的关系为，前者为后者在OS中的具体实现提供了基础；

## 实验中没有对应上的知识点

- OS中对用户进程的管理
- OS中对线程/进程的调度