

实验6

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6

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: 加载应用程序并执行

回答问题

练习2: 父进程复制自己的内存空间给子进程

更新代码

1. alloc_proc 函数

2. do_fork 函数

3. idt_init 函数 (位于 kern/trap/trap.c) 中

4. trap_dispatch 函数 (位于 kern/trap/trap.c) 中

完善 copy_range 函数

简要设计“COW 机制”

练习3: 阅读分析源代码

分析

1. fork 函数

2. exec 函数

3. wait 函数

4. exit 函数

回答问题

生命周期图

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6 用户进程管理

实验目的

- 了解第一个用户进程创建机制
- 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

实验要求

实验5（ucore lab4）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。

本实验6（ucore lab5）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。

实验内容

- 练习0：填写已有实验
- 练习1：加载应用程序并执行（需要编码）
- 练习2：父进程复制自己的内存空间给子进程（需要编码）
- 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）

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  mijialong@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab5$ 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
```

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：

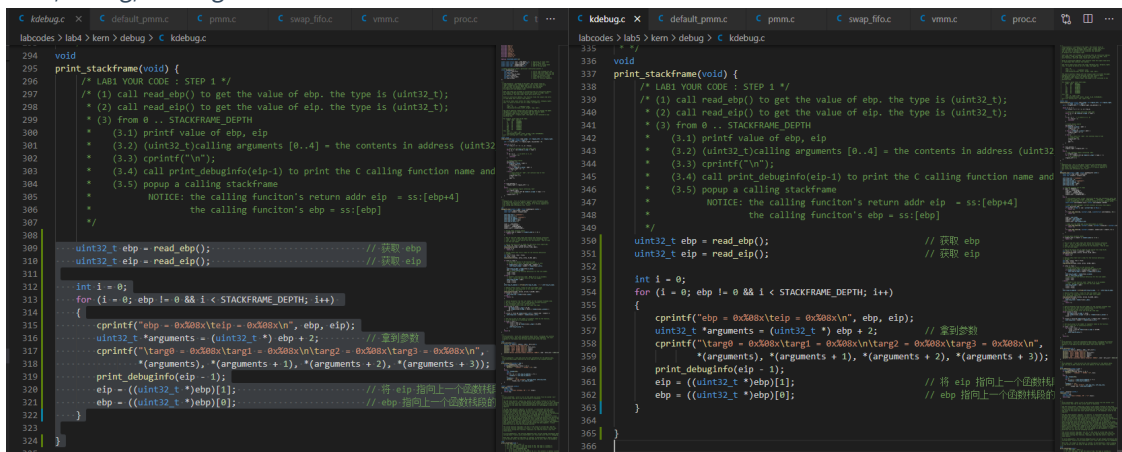
实验过程

练习0：填写已有实验

本实验依赖ucore实验1/2/3/4。请把你做的ucore实验1/2/3/4的代码填入本实验中代码中有“LAB1”、“LAB2”、“LAB3”，“LAB4”的注释相应部分。

需要修改的文件如下：

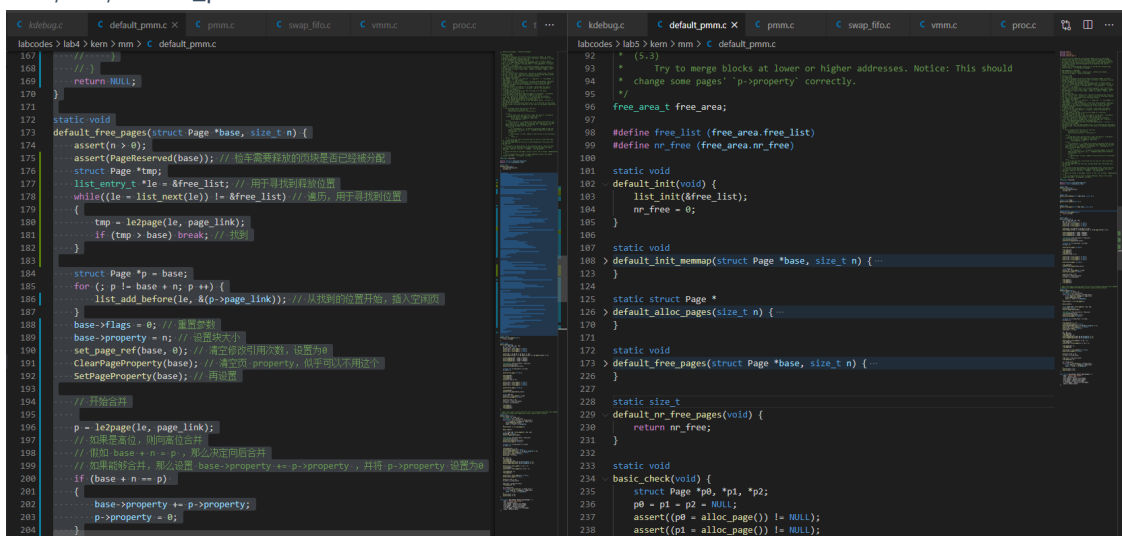
- kern/debug/kdebug.c



```
void print_stackframe(void) {
    /* LAB1 YOUR CODE : STEP 1 */
    /* (1) call read_ebp() to get the value of ebp. the type is (uint32_t);
    * (2) call read_eip() to get the value of eip. the type is (uint32_t);
    * (3) from 0 .. STACKFRAME_DEPTH
    * (3.1) printf value of ebp, eip
    * (3.2) (uint32_t)calling arguments [0..4] = the contents in address (uint32_t)
    * (3.3) cprintf("\n");
    * (3.4) call print_debuginfo(eip-1) to print the C calling function name and
    * (3.5) popup a calling stackframe
    * NOTICE: the calling function's return addr eip = ss:[ebp+4]
    * the calling function's ebp = ss:[ebp]
    */
    uint32_t ebp = read_ebp(); // 获取 ebp
    uint32_t eip = read_eip(); // 获取 eip

    int i = 0;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++) {
        cprintf("ebp = 0x%08x\teip = 0x%08x\n", ebp, eip);
        uint32_t *arguments = (uint32_t *) ebp + 2; // 拿到参数
        cprintf("\targ0 = 0x%08x\targ1 = 0x%08x\targ2 = 0x%08x\targ3 = 0x%08x\n",
            *arguments, *(arguments + 1), *(arguments + 2), *(arguments + 3));
        print_debuginfo(eip - 1);
        ebp = ((uint32_t *)ebp)[1]; // 将 eip 指向上一个函数帧
        eip = ((uint32_t *)ebp)[0]; // ebp 指向上一个函数帧的
    }
}
```

- kern/mm/default_pmm.c



```
static void default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    assert(PageReserved(base)); // 检查需要释放的块是否已经被分配
    struct Page *tmp;
    list_entry_t *le = &free_list; // 用于寻找初始位置
    while((le = list_next(le)) != &free_list) // 遍历，用于寻找初始位置
    {
        tmp = le2page(le, page_link);
        if (tmp > base) break; // 找到
    }
    struct Page *p = base;
    for (; p != base + n; p++) {
        list_add_before(le, &(p->page_link)); // 从找到的位置开始，插入空闲块
    }
    base->flags = 0; // 重置参数
    base->property = n; // 设置大小
    set_page_ref(base, 0); // 清空被引用次数，设置为0
    ClearPageProperty(base); // 清空 p->property，防止可以引用这个
    SetPageProperty(base); // 再设置

    // 开始合并
    p = le2page(le, page_link);
    // 如果是相邻，则向高位合并
    // 假如 base + n == p，那么从定位后合并
    // 如果相等合并，那么设置 base->property += p->property，并让 p->property 设置为0
    if (base + n == p) {
        base->property += p->property;
        p->property = 0;
    }
}
```

- kern/mm/pmm.c

```

labcodes> lab5> kern> mm> C pmm.c
339 // pte_t: the kernel virtual address used to store pte
340 // la: the linear address need to map
341 // create: a logical value to decide if alloc a page for PT
342 // return value: the kernel virtual address of this pte
343 pte_t *
344 get_pte(pte_t *pgdir, uintptr_t la, bool create) { ...
345 // if 0 ...
346 //endif
347 // 上面的东西没有动 ...
348 //endif
349 }
350
351 //get_page - get related Page struct for linear address la using PDT pgdir
352 struct Page *
353 get_page(pte_t *pgdir, uintptr_t la, pte_t **ptep_store) { ...
354 }
355
356 //page_remove_pte - free an Page struct which is related linear address la
357 //note: PT is changed, so the TLB need to be invalidate
358 static inline void
359 page_remove_pte(pte_t *pgdir, uintptr_t la, pte_t *ptep) { ...
360 //endif
361 }
362
363 //page_remove - free an Page which is related linear address la and has an validated
364 void
365 page_remove(pte_t *pgdir, uintptr_t la) {
366     pte_t *ptep = get_pte(pgdir, la, 0);
367     if (ptep != NULL) {
368         page_remove_pte(pgdir, la, ptep);
369     }
370 }
371
372 //page_insert - build the map of phy addr of an Page with the linear addr la
373 //parameters:
374 // pgdir: the kernel virtual base address of PDT
375 // page: the Page which need to map
376
labcodes> lab5> kern> mm> C vmm.c
390 set_page_ref(p, 1); // 如果设置该页表, 则需设置该页表引用次数
391 uintptr_t pa = page2pa(p); // 得到物理地址
392 // 但是还没有找到对应的页表, 只能先通过获取物理
393 memset(KADDR(pa), 0, PGSIZE); // 将物理地址初始化为虚拟地址, 因此需要初始
394 // *pdir = pa | PTE_P; // 设置存在
395 // *pdir = *pdir | PTE_U; // 设置可读
396 // *pdir = *pdir | PTE_M; // 设置可写
397 // *pdir = pa | PTE_P | PTE_U | PTE_M; // 统一设置更加方便
398 *pdir = pa | PTE_USER;
399 }
400
401 return ((pte_t *)KADDR(POE_ADDR(*pdir)))[PTX(la)]; // 得到二级页表对应的线性地址
402
403 //get_page - get related Page struct for linear address la using PDT pgdir
404 struct Page *
405 get_page(pte_t *pgdir, uintptr_t la, pte_t **ptep_store) { ...
406 }
407
408 //page_remove_pte - free an Page struct which is related linear address la
409 //note: PT is changed, so the TLB need to be invalidate
410 static inline void
411 page_remove_pte(pte_t *pgdir, uintptr_t la, pte_t *ptep) { ...
412 //endif
413 }
414
415 //page_remove - free an Page which is related linear address la and has an validated
416 void
417 page_remove(pte_t *pgdir, uintptr_t la) {
418     pte_t *ptep = get_pte(pgdir, la, 0);
419     if (ptep != NULL) {
420         page_remove_pte(pgdir, la, ptep);
421     }
422 }
423
424 //page_insert - build the map of phy addr of an Page with the linear addr la
425 //parameters:
426 // pgdir: the kernel virtual base address of PDT
427 // page: the Page which need to map
428
labcodes> lab5> kern> mm> C vmm.c
498 ptep = get_pte(mm->pgdir, addr, 1); // 获取 ptep
499 if (ptep == NULL) // 如果 ptep 不存在
500 {
501     printf("ptep isn't existed, get_pte() in do_pfault() failed\n");
502     goto failed;
503 }
504
505 if (*ptep == 0) // 如果 pa 不存在
506 {
507     if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) // 尝试分配页并映射
508     {
509         printf("pa isn't existed, and alloc page failed in do_pfault() failed\n");
510         goto failed;
511     }
512 }
513
514 // 以下是练习代码
515 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
516 if (swap_init_ok) // 代表初始化成功
517 {
518     struct Page *page = NULL;
519     ret = swap_in(mm, addr, &page); // 换页
520     if (ret != 0) // 如果换页失败
521     {
522         printf("swap_in in do_pfault failed\n");
523         goto failed;
524     }
525 }
526
527 // 以下是练习代码
528 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
529 if (swap_init_ok) // 代表初始化成功
530 {
531     struct Page *page = NULL;
532     ret = swap_in(mm, addr, &page); // 换页
533     if (ret != 0) // 如果换页失败
534     {
535         printf("swap_in in do_pfault failed\n");
536         goto failed;
537     }
538 }
539
540 page_insert(mm->pgdir, page, addr, perm); // 建立映射, 同时通过 perm 设置
541 swap_map_swappable(mm, addr, page, 1); // 设置为可交换
542
543 page->pra_vaddr = addr; // 设置页对应的虚拟地址
544
545 // 初始化失败
546 else // 初始化失败
547 {
548     printf("no swap_init_ok but ptep is %x, failed\n", *ptep);
549     goto failed;
550 }
551
552 return page;
553 }
554
555 // 初始化失败
556 else // 初始化失败
557 {
558     printf("no swap_init_ok but ptep is %x, failed\n", *ptep);
559     goto failed;
560 }
561
562 return page;
563 }

```

- kern/mm/vmm.c

```

labcodes> lab5> kern> mm> C vmm.c
481 goto failed;
482
483 // 如果 ptep == 0 // 如果 pa 不存在
484 {
485     if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) // 尝试分配页并映射
486     {
487         printf("pa isn't existed, and alloc page failed in do_pfault() failed\n");
488         goto failed;
489     }
490 }
491
492 // 以下是练习代码
493 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
494 if (swap_init_ok) // 代表初始化成功
495 {
496     struct Page *page = NULL;
497     ret = swap_in(mm, addr, &page); // 换页
498     if (ret != 0) // 如果换页失败
499     {
500         printf("swap_in in do_pfault failed\n");
501         goto failed;
502     }
503 }
504
505 // 以下是练习代码
506 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
507 if (swap_init_ok) // 代表初始化成功
508 {
509     struct Page *page = NULL;
510     ret = swap_in(mm, addr, &page); // 换页
511     if (ret != 0) // 如果换页失败
512     {
513         printf("swap_in in do_pfault failed\n");
514         goto failed;
515     }
516 }
517
518 // 以下是练习代码
519 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
520 if (swap_init_ok) // 代表初始化成功
521 {
522     struct Page *page = NULL;
523     ret = swap_in(mm, addr, &page); // 换页
524     if (ret != 0) // 如果换页失败
525     {
526         printf("swap_in in do_pfault failed\n");
527         goto failed;
528     }
529 }
530
531 // 以下是练习代码
532 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
533 if (swap_init_ok) // 代表初始化成功
534 {
535     struct Page *page = NULL;
536     ret = swap_in(mm, addr, &page); // 换页
537     if (ret != 0) // 如果换页失败
538     {
539         printf("swap_in in do_pfault failed\n");
540         goto failed;
541     }
542 }
543
544 // 以下是练习代码
545 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
546 if (swap_init_ok) // 代表初始化成功
547 {
548     struct Page *page = NULL;
549     ret = swap_in(mm, addr, &page); // 换页
550     if (ret != 0) // 如果换页失败
551     {
552         printf("swap_in in do_pfault failed\n");
553         goto failed;
554     }
555 }
556
557 // 以下是练习代码
558 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
559 if (swap_init_ok) // 代表初始化成功
560 {
561     struct Page *page = NULL;
562     ret = swap_in(mm, addr, &page); // 换页
563     if (ret != 0) // 如果换页失败
564     {
565         printf("swap_in in do_pfault failed\n");
566         goto failed;
567     }
568 }
569
570 // 以下是练习代码
571 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
572 if (swap_init_ok) // 代表初始化成功
573 {
574     struct Page *page = NULL;
575     ret = swap_in(mm, addr, &page); // 换页
576     if (ret != 0) // 如果换页失败
577     {
578         printf("swap_in in do_pfault failed\n");
579         goto failed;
580     }
581 }
582
583 // 以下是练习代码
584 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
585 if (swap_init_ok) // 代表初始化成功
586 {
587     struct Page *page = NULL;
588     ret = swap_in(mm, addr, &page); // 换页
589     if (ret != 0) // 如果换页失败
590     {
591         printf("swap_in in do_pfault failed\n");
592         goto failed;
593     }
594 }
595
596 // 以下是练习代码
597 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
598 if (swap_init_ok) // 代表初始化成功
599 {
600     struct Page *page = NULL;
601     ret = swap_in(mm, addr, &page); // 换页
602     if (ret != 0) // 如果换页失败
603     {
604         printf("swap_in in do_pfault failed\n");
605         goto failed;
606     }
607 }
608
609 // 以下是练习代码
610 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
611 if (swap_init_ok) // 代表初始化成功
612 {
613     struct Page *page = NULL;
614     ret = swap_in(mm, addr, &page); // 换页
615     if (ret != 0) // 如果换页失败
616     {
617         printf("swap_in in do_pfault failed\n");
618         goto failed;
619     }
620 }
621
622 // 以下是练习代码
623 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
624 if (swap_init_ok) // 代表初始化成功
625 {
626     struct Page *page = NULL;
627     ret = swap_in(mm, addr, &page); // 换页
628     if (ret != 0) // 如果换页失败
629     {
630         printf("swap_in in do_pfault failed\n");
631         goto failed;
632     }
633 }
634
635 // 以下是练习代码
636 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
637 if (swap_init_ok) // 代表初始化成功
638 {
639     struct Page *page = NULL;
640     ret = swap_in(mm, addr, &page); // 换页
641     if (ret != 0) // 如果换页失败
642     {
643         printf("swap_in in do_pfault failed\n");
644         goto failed;
645     }
646 }
647
648 // 以下是练习代码
649 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
650 if (swap_init_ok) // 代表初始化成功
651 {
652     struct Page *page = NULL;
653     ret = swap_in(mm, addr, &page); // 换页
654     if (ret != 0) // 如果换页失败
655     {
656         printf("swap_in in do_pfault failed\n");
657         goto failed;
658     }
659 }
660
661 // 以下是练习代码
662 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
663 if (swap_init_ok) // 代表初始化成功
664 {
665     struct Page *page = NULL;
666     ret = swap_in(mm, addr, &page); // 换页
667     if (ret != 0) // 如果换页失败
668     {
669         printf("swap_in in do_pfault failed\n");
670         goto failed;
671     }
672 }
673
674 // 以下是练习代码
675 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
676 if (swap_init_ok) // 代表初始化成功
677 {
678     struct Page *page = NULL;
679     ret = swap_in(mm, addr, &page); // 换页
680     if (ret != 0) // 如果换页失败
681     {
682         printf("swap_in in do_pfault failed\n");
683         goto failed;
684     }
685 }
686
687 // 以下是练习代码
688 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
689 if (swap_init_ok) // 代表初始化成功
690 {
691     struct Page *page = NULL;
692     ret = swap_in(mm, addr, &page); // 换页
693     if (ret != 0) // 如果换页失败
694     {
695         printf("swap_in in do_pfault failed\n");
696         goto failed;
697     }
698 }
699
700 // 以下是练习代码
701 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
702 if (swap_init_ok) // 代表初始化成功
703 {
704     struct Page *page = NULL;
705     ret = swap_in(mm, addr, &page); // 换页
706     if (ret != 0) // 如果换页失败
707     {
708         printf("swap_in in do_pfault failed\n");
709         goto failed;
710     }
711 }
712
713 // 以下是练习代码
714 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
715 if (swap_init_ok) // 代表初始化成功
716 {
717     struct Page *page = NULL;
718     ret = swap_in(mm, addr, &page); // 换页
719     if (ret != 0) // 如果换页失败
720     {
721         printf("swap_in in do_pfault failed\n");
722         goto failed;
723     }
724 }
725
726 // 以下是练习代码
727 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
728 if (swap_init_ok) // 代表初始化成功
729 {
730     struct Page *page = NULL;
731     ret = swap_in(mm, addr, &page); // 换页
732     if (ret != 0) // 如果换页失败
733     {
734         printf("swap_in in do_pfault failed\n");
735         goto failed;
736     }
737 }
738
739 // 以下是练习代码
740 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
741 if (swap_init_ok) // 代表初始化成功
742 {
743     struct Page *page = NULL;
744     ret = swap_in(mm, addr, &page); // 换页
745     if (ret != 0) // 如果换页失败
746     {
747         printf("swap_in in do_pfault failed\n");
748         goto failed;
749     }
750 }
751
752 // 以下是练习代码
753 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
754 if (swap_init_ok) // 代表初始化成功
755 {
756     struct Page *page = NULL;
757     ret = swap_in(mm, addr, &page); // 换页
758     if (ret != 0) // 如果换页失败
759     {
760         printf("swap_in in do_pfault failed\n");
761         goto failed;
762     }
763 }
764
765 // 以下是练习代码
766 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
767 if (swap_init_ok) // 代表初始化成功
768 {
769     struct Page *page = NULL;
770     ret = swap_in(mm, addr, &page); // 换页
771     if (ret != 0) // 如果换页失败
772     {
773         printf("swap_in in do_pfault failed\n");
774         goto failed;
775     }
776 }
777
778 // 以下是练习代码
779 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
780 if (swap_init_ok) // 代表初始化成功
781 {
782     struct Page *page = NULL;
783     ret = swap_in(mm, addr, &page); // 换页
784     if (ret != 0) // 如果换页失败
785     {
786         printf("swap_in in do_pfault failed\n");
787         goto failed;
788     }
789 }
790
791 // 以下是练习代码
792 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
793 if (swap_init_ok) // 代表初始化成功
794 {
795     struct Page *page = NULL;
796     ret = swap_in(mm, addr, &page); // 换页
797     if (ret != 0) // 如果换页失败
798     {
799         printf("swap_in in do_pfault failed\n");
800         goto failed;
801     }
802 }
803
804 // 以下是练习代码
805 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
806 if (swap_init_ok) // 代表初始化成功
807 {
808     struct Page *page = NULL;
809     ret = swap_in(mm, addr, &page); // 换页
810     if (ret != 0) // 如果换页失败
811     {
812         printf("swap_in in do_pfault failed\n");
813         goto failed;
814     }
815 }
816
817 // 以下是练习代码
818 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
819 if (swap_init_ok) // 代表初始化成功
820 {
821     struct Page *page = NULL;
822     ret = swap_in(mm, addr, &page); // 换页
823     if (ret != 0) // 如果换页失败
824     {
825         printf("swap_in in do_pfault failed\n");
826         goto failed;
827     }
828 }
829
830 // 以下是练习代码
831 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
832 if (swap_init_ok) // 代表初始化成功
833 {
834     struct Page *page = NULL;
835     ret = swap_in(mm, addr, &page); // 换页
836     if (ret != 0) // 如果换页失败
837     {
838         printf("swap_in in do_pfault failed\n");
839         goto failed;
840     }
841 }
842
843 // 以下是练习代码
844 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
845 if (swap_init_ok) // 代表初始化成功
846 {
847     struct Page *page = NULL;
848     ret = swap_in(mm, addr, &page); // 换页
849     if (ret != 0) // 如果换页失败
850     {
851         printf("swap_in in do_pfault failed\n");
852         goto failed;
853     }
854 }
855
856 // 以下是练习代码
857 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
858 if (swap_init_ok) // 代表初始化成功
859 {
860     struct Page *page = NULL;
861     ret = swap_in(mm, addr, &page); // 换页
862     if (ret != 0) // 如果换页失败
863     {
864         printf("swap_in in do_pfault failed\n");
865         goto failed;
866     }
867 }
868
869 // 以下是练习代码
870 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
871 if (swap_init_ok) // 代表初始化成功
872 {
873     struct Page *page = NULL;
874     ret = swap_in(mm, addr, &page); // 换页
875     if (ret != 0) // 如果换页失败
876     {
877         printf("swap_in in do_pfault failed\n");
878         goto failed;
879     }
880 }
881
882 // 以下是练习代码
883 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
884 if (swap_init_ok) // 代表初始化成功
885 {
886     struct Page *page = NULL;
887     ret = swap_in(mm, addr, &page); // 换页
888     if (ret != 0) // 如果换页失败
889     {
890         printf("swap_in in do_pfault failed\n");
891         goto failed;
892     }
893 }
894
895 // 以下是练习代码
896 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
897 if (swap_init_ok) // 代表初始化成功
898 {
899     struct Page *page = NULL;
900     ret = swap_in(mm, addr, &page); // 换页
901     if (ret != 0) // 如果换页失败
902     {
903         printf("swap_in in do_pfault failed\n");
904         goto failed;
905     }
906 }
907
908 // 以下是练习代码
909 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
910 if (swap_init_ok) // 代表初始化成功
911 {
912     struct Page *page = NULL;
913     ret = swap_in(mm, addr, &page); // 换页
914     if (ret != 0) // 如果换页失败
915     {
916         printf("swap_in in do_pfault failed\n");
917         goto failed;
918     }
919 }
920
921 // 以下是练习代码
922 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
923 if (swap_init_ok) // 代表初始化成功
924 {
925     struct Page *page = NULL;
926     ret = swap_in(mm, addr, &page); // 换页
927     if (ret != 0) // 如果换页失败
928     {
929         printf("swap_in in do_pfault failed\n");
930         goto failed;
931     }
932 }
933
934 // 以下是练习代码
935 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
936 if (swap_init_ok) // 代表初始化成功
937 {
938     struct Page *page = NULL;
939     ret = swap_in(mm, addr, &page); // 换页
940     if (ret != 0) // 如果换页失败
941     {
942         printf("swap_in in do_pfault failed\n");
943         goto failed;
944     }
945 }
946
947 // 以下是练习代码
948 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
949 if (swap_init_ok) // 代表初始化成功
950 {
951     struct Page *page = NULL;
952     ret = swap_in(mm, addr, &page); // 换页
953     if (ret != 0) // 如果换页失败
954     {
955         printf("swap_in in do_pfault failed\n");
956         goto failed;
957     }
958 }
959
960 // 以下是练习代码
961 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
962 if (swap_init_ok) // 代表初始化成功
963 {
964     struct Page *page = NULL;
965     ret = swap_in(mm, addr, &page); // 换页
966     if (ret != 0) // 如果换页失败
967     {
968         printf("swap_in in do_pfault failed\n");
969         goto failed;
970     }
971 }
972
973 // 以下是练习代码
974 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
975 if (swap_init_ok) // 代表初始化成功
976 {
977     struct Page *page = NULL;
978     ret = swap_in(mm, addr, &page); // 换页
979     if (ret != 0) // 如果换页失败
980     {
981         printf("swap_in in do_pfault failed\n");
982         goto failed;
983     }
984 }
985
986 // 以下是练习代码
987 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
988 if (swap_init_ok) // 代表初始化成功
989 {
990     struct Page *page = NULL;
991     ret = swap_in(mm, addr, &page); // 换页
992     if (ret != 0) // 如果换页失败
993     {
994         printf("swap_in in do_pfault failed\n");
995         goto failed;
996     }
997 }
998
999 // 以下是练习代码
1000 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1001 if (swap_init_ok) // 代表初始化成功
1002 {
1003     struct Page *page = NULL;
1004     ret = swap_in(mm, addr, &page); // 换页
1005     if (ret != 0) // 如果换页失败
1006     {
1007         printf("swap_in in do_pfault failed\n");
1008         goto failed;
1009     }
1010 }
1011
1012 // 以下是练习代码
1013 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1014 if (swap_init_ok) // 代表初始化成功
1015 {
1016     struct Page *page = NULL;
1017     ret = swap_in(mm, addr, &page); // 换页
1018     if (ret != 0) // 如果换页失败
1019     {
1020         printf("swap_in in do_pfault failed\n");
1021         goto failed;
1022     }
1023 }
1024
1025 // 以下是练习代码
1026 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1027 if (swap_init_ok) // 代表初始化成功
1028 {
1029     struct Page *page = NULL;
1030     ret = swap_in(mm, addr, &page); // 换页
1031     if (ret != 0) // 如果换页失败
1032     {
1033         printf("swap_in in do_pfault failed\n");
1034         goto failed;
1035     }
1036 }
1037
1038 // 以下是练习代码
1039 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1040 if (swap_init_ok) // 代表初始化成功
1041 {
1042     struct Page *page = NULL;
1043     ret = swap_in(mm, addr, &page); // 换页
1044     if (ret != 0) // 如果换页失败
1045     {
1046         printf("swap_in in do_pfault failed\n");
1047         goto failed;
1048     }
1049 }
1050
1051 // 以下是练习代码
1052 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1053 if (swap_init_ok) // 代表初始化成功
1054 {
1055     struct Page *page = NULL;
1056     ret = swap_in(mm, addr, &page); // 换页
1057     if (ret != 0) // 如果换页失败
1058     {
1059         printf("swap_in in do_pfault failed\n");
1060         goto failed;
1061     }
1062 }
1063
1064 // 以下是练习代码
1065 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1066 if (swap_init_ok) // 代表初始化成功
1067 {
1068     struct Page *page = NULL;
1069     ret = swap_in(mm, addr, &page); // 换页
1070     if (ret != 0) // 如果换页失败
1071     {
1072         printf("swap_in in do_pfault failed\n");
1073         goto failed;
1074     }
1075 }
1076
1077 // 以下是练习代码
1078 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1079 if (swap_init_ok) // 代表初始化成功
1080 {
1081     struct Page *page = NULL;
1082     ret = swap_in(mm, addr, &page); // 换页
1083     if (ret != 0) // 如果换页失败
1084     {
1085         printf("swap_in in do_pfault failed\n");
1086         goto failed;
1087     }
1088 }
1089
1090 // 以下是练习代码
1091 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1092 if (swap_init_ok) // 代表初始化成功
1093 {
1094     struct Page *page = NULL;
1095     ret = swap_in(mm, addr, &page); // 换页
1096     if (ret != 0) // 如果换页失败
1097     {
1098         printf("swap_in in do_pfault failed\n");
1099         goto failed;
1100     }
1101 }
1102
1103 // 以下是练习代码
1104 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1105 if (swap_init_ok) // 代表初始化成功
1106 {
1107     struct Page *page = NULL;
1108     ret = swap_in(mm, addr, &page); // 换页
1109     if (ret != 0) // 如果换页失败
1110     {
1111         printf("swap_in in do_pfault failed\n");
1112         goto failed;
1113     }
1114 }
1115
1116 // 以下是练习代码
1117 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1118 if (swap_init_ok) // 代表初始化成功
1119 {
1120     struct Page *page = NULL;
1121     ret = swap_in(mm, addr, &page); // 换页
1122     if (ret != 0) // 如果换页失败
1123     {
1124         printf("swap_in in do_pfault failed\n");
1125         goto failed;
1126     }
1127 }
1128
1129 // 以下是练习代码
1130 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1131 if (swap_init_ok) // 代表初始化成功
1132 {
1133     struct Page *page = NULL;
1134     ret = swap_in(mm, addr, &page); // 换页
1135     if (ret != 0) // 如果换页失败
1136     {
1137         printf("swap_in in do_pfault failed\n");
1138         goto failed;
1139     }
1140 }
1141
1142 // 以下是练习代码
1143 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1144 if (swap_init_ok) // 代表初始化成功
1145 {
1146     struct Page *page = NULL;
1147     ret = swap_in(mm, addr, &page); // 换页
1148     if (ret != 0) // 如果换页失败
1149     {
1150         printf("swap_in in do_pfault failed\n");
1151         goto failed;
1152     }
1153 }
1154
1155 // 以下是练习代码
1156 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1157 if (swap_init_ok) // 代表初始化成功
1158 {
1159     struct Page *page = NULL;
1160     ret = swap_in(mm, addr, &page); // 换页
1161     if (ret != 0) // 如果换页失败
1162     {
1163         printf("swap_in in do_pfault failed\n");
1164         goto failed;
1165     }
1166 }
1167
1168 // 以下是练习代码
1169 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1170 if (swap_init_ok) // 代表初始化成功
1171 {
1172     struct Page *page = NULL;
1173     ret = swap_in(mm, addr, &page); // 换页
1174     if (ret != 0) // 如果换页失败
1175     {
1176         printf("swap_in in do_pfault failed\n");
1177         goto failed;
1178     }
1179 }
1180
1181 // 以下是练习代码
1182 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1183 if (swap_init_ok) // 代表初始化成功
1184 {
1185     struct Page *page = NULL;
1186     ret = swap_in(mm, addr, &page); // 换页
1187     if (ret != 0) // 如果换页失败
1188     {
1189         printf("swap_in in do_pfault failed\n");
1190         goto failed;
1191     }
1192 }
1193
1194 // 以下是练习代码
1195 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1196 if (swap_init_ok) // 代表初始化成功
1197 {
1198     struct Page *page = NULL;
1199     ret = swap_in(mm, addr, &page); // 换页
1200     if (ret != 0) // 如果换页失败
1201     {
1202         printf("swap_in in do_pfault failed\n");
1203         goto failed;
1204     }
1205 }
1206
1207 // 以下是练习代码
1208 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1209 if (swap_init_ok) // 代表初始化成功
1210 {
1211     struct Page *page = NULL;
1212     ret = swap_in(mm, addr, &page); // 换页
1213     if (ret != 0) // 如果换页失败
1214     {
1215         printf("swap_in in do_pfault failed\n");
1216         goto failed;
1217     }
1218 }
1219
1220 // 以下是练习代码
1221 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1222 if (swap_init_ok) // 代表初始化成功
1223 {
1224     struct Page *page = NULL;
1225     ret = swap_in(mm, addr, &page); // 换页
1226     if (ret != 0) // 如果换页失败
1227     {
1228         printf("swap_in in do_pfault failed\n");
1229         goto failed;
1230     }
1231 }
1232
1233 // 以下是练习代码
1234 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1235 if (swap_init_ok) // 代表初始化成功
1236 {
1237     struct Page *page = NULL;
1238     ret = swap_in(mm, addr, &page); // 换页
1239     if (ret != 0) // 如果换页失败
1240     {
1241         printf("swap_in in do_pfault failed\n");
1242         goto failed;
1243     }
1244 }
1245
1246 // 以下是练习代码
1247 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1248 if (swap_init_ok) // 代表初始化成功
1249 {
1250     struct Page *page = NULL;
1251     ret = swap_in(mm, addr, &page); // 换页
1252     if (ret != 0) // 如果换页失败
1253     {
1254         printf("swap_in in do_pfault failed\n");
1255         goto failed;
1256     }
1257 }
1258
1259 // 以下是练习代码
1260 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1261 if (swap_init_ok) // 代表初始化成功
1262 {
1263     struct Page *page = NULL;
1264     ret = swap_in(mm, addr, &page); // 换页
1265     if (ret != 0) // 如果换页失败
1266     {
1267         printf("swap_in in do_pfault failed\n");
1268         goto failed;
1269     }
1270 }
1271
1272 // 以下是练习代码
1273 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1274 if (swap_init_ok) // 代表初始化成功
1275 {
1276     struct Page *page = NULL;
1277     ret = swap_in(mm, addr, &page); // 换页
1278     if (ret != 0) // 如果换页失败
1279     {
1280         printf("swap_in in do_pfault failed\n");
1281         goto failed;
1282     }
1283 }
1284
1285 // 以下是练习代码
1286 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1287 if (swap_init_ok) // 代表初始化成功
1288 {
1289     struct Page *page = NULL;
1290     ret = swap_in(mm, addr, &page); // 换页
1291     if (ret != 0) // 如果换页失败
1292     {
1293         printf("swap_in in do_pfault failed\n");
1294         goto failed;
1295     }
1296 }
1297
1298 // 以下是练习代码
1299 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1300 if (swap_init_ok) // 代表初始化成功
1301 {
1302     struct Page *page = NULL;
1303     ret = swap_in(mm, addr, &page); // 换页
1304     if (ret != 0) // 如果换页失败
1305     {
1306         printf("swap_in in do_pfault failed\n");
1307         goto failed;
1308     }
1309 }
1310
1311 // 以下是练习代码
1312 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1313 if (swap_init_ok) // 代表初始化成功
1314 {
1315     struct Page *page = NULL;
1316     ret = swap_in(mm, addr, &page); // 换页
1317     if (ret != 0) // 如果换页失败
1318     {
1319         printf("swap_in in do_pfault failed\n");
1320         goto failed;
1321     }
1322 }
1323
1324 // 以下是练习代码
1325 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1326 if (swap_init_ok) // 代表初始化成功
1327 {
1328     struct Page *page = NULL;
1329     ret = swap_in(mm, addr, &page); // 换页
1330     if (ret != 0) // 如果换页失败
1331     {
1332         printf("swap_in in do_pfault failed\n");
1333         goto failed;
1334     }
1335 }
1336
1337 // 以下是练习代码
1338 // 如果 pa 不为空, 则页表项非空, 尝试填入该页面
1339 if (swap_init_ok) // 代表初始化成功
1340 {
1341     struct Page *page = NULL;
1342     ret = swap_in(mm, addr, &page); // 换页

```

- kern/trap/trap.c

```
x vmime x trap.x
```

```
labcodes > lab5 > kern > trap > C trap.x  
26 /* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S */  
27 void  
28 idt_init(void) {  
29     /* LAB5 YOUR CODE : STEP 2 */  
30     /* (1) where are the entry address of each Interrupt Service Routine (ISR)?  
31      * All ISR's entry address are stored in __vectors, which is uintptr_t vector.  
32      * __vectors[] is in kern/trap/vectors.S which is produced by tools/vector.c  
33      * (try "make" command in lab1, then you will find vector.S in kern/trap DIR  
34      * You can use "extern uintptr_t __vectors[];" to define this extern variab  
35      * (2) Now you should setup the entries of ISR in Interrupt Description Table (I  
36      * You can see idt[256] in this file! Yes, it's IDT! you can use SETGATE mac  
37      * After setup the contents of IDT, you will let CPU know where is the IDT b  
38      * You don't know the meaning of this instruction? Just google it! and check  
39      * Notice: the argument of lidt is idt_pd. try to find it!  
40     */  
41  
42     extern uintptr_t __vectors[];           // 声明 vector  
43     int i = 0;  
44     for (i = 0; i < 256; i++)              // 对于每一个表项进  
45     {  
46         SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);  
47     }  
48     SETGATE(idt[T_SWITCH_TOK], 0, GD_KTEXT, __vectors[T_SWITCH_TOK], DPL_KERNEL);  
49     lidt(&idt_pd);                        // 加载 idt  
50 }  
51  
52 static const char *  
53 trname(int trapno) {  
54  
labcodes > lab5 > kern > trap > C trap.x  
198 /*  
199 (3) Too Simple? Yes, I think so!  
200 */  
201 ticks++;  
202 if (ticks == TICK_NUM)  
203 {  
204     ticks = 0;  
205     print_ticks();  
206 }  
207 break;  
208 case IRQ_OFFSET + IRQ_COM1:  
209     c = cons_getc();  
210     cprintf("serial [%03d] %c\n", c, c);  
211     break;  
212 case IRQ_OFFSET + IRQ_KBD:
```

- kern/process/proc.c

```

lab0odes> lab4> km> process> < proc.c
91 * below fields in proc_struct need to be initialized
92 * enum proc_state; // Process state
93 * int pid; // Process ID
94 * int rums; // the running times of Proc
95 * uintptr_t kstack; // Process kernel stack
96 * volatile bool need_resched; // bool value: need to be re
97 * struct proc_struct *parent; // the parent process
98 * struct mm_struct *mm; // Process's memory manage
99 * struct context context; // Switch here to run proces
100 * struct trapframe *tf; // Trap frame for current in
101 * uintptr_t cr3; // CR3 register: the base ad
102 * uint32_t flags; // Process flag
103 * char name[PROC_NAME_LEN + 1]; // Process name
104 */
105
106 proc->state = PROC_UNINIT; // 状态
107 proc->pid = -1; // 给一个无
108 proc->rums = 0; // 未运行过
109 proc->kstack = 0; // 没有内核栈
110 proc->need_resched = 0; // 不需要被 CPU 调度
111 proc->parent = NULL; // 没有父进程
112 proc->mm = NULL; // 没有内存管理字段
113 memset(&(proc->context), 0, sizeof(struct context)); // 设置上下文
114 proc->tf = NULL; // 无中断帧
115 proc->cr3 = boot_cr3; // 设置内核页目录表的
116 proc->flags = 0; // 设置进程标志
117 memset(&(proc->name), 0, PROC_NAME_LEN); // 设置进程名空为
118
119 return proc;
120 }
121
122 // set proc_name - set the name of proc
123 char *
124 set_proc_name(struct proc_struct *proc, const char *name) {
125     memset(&(proc->name), 0, sizeof(proc->name));
126     memcpy(&(proc->name), name, strlen(name));
127 }
128
129 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
303 */
304
305 // 1. 使用 "alloc_proc" 初始化进程控制块
306 if ((proc = alloc_proc()) == NULL) // 申请内存失败
307     goto fork_out;
308
309 // 2. 使用 "setup_states" 为子进程分配并初始化内核栈
310 proc->parent = current; // 将父进程设置为当前进程
311 if (setup_kstack(proc) != 0) // 如果分配内核栈失败
312     goto bad_fork_cleanup_proc;
313
314 // 3. 使用 "copy_mm" 根据 "clone_flag" 复制/共享进程的内存管理结构
315 if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
316     goto bad_fork_cleanup_proc;
317
318 // 4. 使用 "copy_thread" 设置进程在内核正常运行和调度所需的断点和上下文
319 copy_thread(proc, stack, td);
320
321 // 5. 将设置好的控制进程模块插入到 "hash_list" 和 "proc_list" 中
322 bool intr_flag; // 标志
323 local_intr_save(intr_flag); // 屏蔽中断，并将标志设置为1
324 {
325     proc->pid = get_pid(); // 获取 pid
326     hash_proc(proc); // 建立映射
327     nr_process++; // 记录数量增加
328     list_add(&(proc->list), &(proc->list_link)); // 进程加入入到进程表中
329 }
330 local_intr_restore(intr_flag); // 恢复中断
331
332 // 6. 使用 "wakeup_proc" 将进程设置为"就绪"状态
333 wakeup_proc(proc); // 唤醒进程
334
335 // 7. 将返回结果设置为"子进程 pid"
336 ret = proc->pid;
337
338 fork_out:
339 return ret;
340
341 bad_fork_cleanup_kstack:
342 put_kstack(proc);
343 bad_fork_cleanup_proc:
344 kfree(proc);
345
346 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
93 * enum proc_state; // Process state
94 * int pid; // Process ID
95 * int rums; // the running times of Proc
96 * uintptr_t kstack; // Process kernel stack
97 * volatile bool need_resched; // bool value: need to be re
98 * struct proc_struct *parent; // the parent process
99 * struct mm_struct *mm; // Process's memory manage
100 * struct context context; // Switch here to run proces
101 * struct trapframe *tf; // Trap frame for current in
102 * uintptr_t cr3; // CR3 register: the base ad
103 * uint32_t flags; // Process flag
104 * char name[PROC_NAME_LEN + 1]; // Process name
105 */
106
107 proc->state = PROC_UNINIT; // 状态
108 proc->pid = -1; // 给一个无
109 proc->rums = 0; // 未运行过
110 proc->kstack = 0; // 没有内核栈
111 proc->need_resched = 0; // 不需要被 CPU 调度
112 proc->parent = NULL; // 没有父进程
113 memset(&(proc->context), 0, sizeof(struct context)); // 设置上下文
114 proc->tf = NULL; // 无中断帧
115 proc->cr3 = boot_cr3; // 设置内核页目录表的
116 proc->flags = 0; // 设置进程标志
117 memset(&(proc->name), 0, PROC_NAME_LEN); // 设置进程名空为
118
119 //LABS YOUR CODE : (update LAB4 steps)
120
121 * below fields(add in LAB5) in proc_struct need to be initialized
122 * uint32_t wait_state; // waiting state
123 * struct proc_struct *cptr, *pptr, *optr; // relations between process
124 */
125
126 return proc;
127
128 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
403 */
404
405 // 1. 使用 "alloc_proc" 初始化进程控制块
406 if ((proc = alloc_proc()) == NULL) // 申请内存失败
407     goto fork_out;
408
409 // 2. 使用 "setup_states" 为子进程分配并初始化内核栈
410 proc->parent = current; // 将父进程设置为当前进程
411 if (setup_kstack(proc) != 0) // 如果分配内核栈失败
412     goto bad_fork_cleanup_proc;
413
414 // 3. 使用 "copy_mm" 根据 "clone_flag" 复制/共享进程的内存管理结构
415 if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
416     goto bad_fork_cleanup_proc;
417
418 // 4. 使用 "copy_thread" 设置进程在内核正常运行和调度所需的断点和上下文
419 copy_thread(proc, stack, td);
420
421 // 5. 将设置好的控制进程模块插入到 "hash_list" 和 "proc_list" 中
422 bool intr_flag; // 标志
423 local_intr_save(intr_flag); // 屏蔽中断，并将标志设置为1
424 {
425     proc->pid = get_pid(); // 获取 pid
426     hash_proc(proc); // 建立映射
427     nr_process++; // 记录数量增加
428     list_add(&(proc->list), &(proc->list_link)); // 进程加入入到进程表中
429 }
430 local_intr_restore(intr_flag); // 恢复中断
431
432 // 6. 使用 "wakeup_proc" 将进程设置为"就绪"状态
433 wakeup_proc(proc); // 唤醒进程
434
435 // 7. 将返回结果设置为"子进程 pid"
436 ret = proc->pid;
437
438 //LABS YOUR CODE : (update LAB4 steps)
439
440 * Some Functions
441 * set_links: set the relation links of process. ALSO SEE: remove_links; lea
442 * -----
443 * update step 1: set child proc's parent to current process, make sure current
444 * update step 5: Insert proc_struct into hash_list and proc_list, set the relat
445
446 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
403 */
404
405 // 1. 使用 "alloc_proc" 初始化进程控制块
406 if ((proc = alloc_proc()) == NULL) // 申请内存失败
407     goto fork_out;
408
409 // 2. 使用 "setup_states" 为子进程分配并初始化内核栈
410 proc->parent = current; // 将父进程设置为当前进程
411 if (setup_kstack(proc) != 0) // 如果分配内核栈失败
412     goto bad_fork_cleanup_proc;
413
414 // 3. 使用 "copy_mm" 根据 "clone_flag" 复制/共享进程的内存管理结构
415 if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
416     goto bad_fork_cleanup_proc;
417
418 // 4. 使用 "copy_thread" 设置进程在内核正常运行和调度所需的断点和上下文
419 copy_thread(proc, stack, td);
420
421 // 5. 将设置好的控制进程模块插入到 "hash_list" 和 "proc_list" 中
422 bool intr_flag; // 标志
423 local_intr_save(intr_flag); // 屏蔽中断，并将标志设置为1
424 {
425     proc->pid = get_pid(); // 获取 pid
426     hash_proc(proc); // 建立映射
427     nr_process++; // 记录数量增加
428     list_add(&(proc->list), &(proc->list_link)); // 进程加入入到进程表中
429 }
430 local_intr_restore(intr_flag); // 恢复中断
431
432 // 6. 使用 "wakeup_proc" 将进程设置为"就绪"状态
433 wakeup_proc(proc); // 唤醒进程
434
435 // 7. 将返回结果设置为"子进程 pid"
436 ret = proc->pid;
437
438 //LABS YOUR CODE : (update LAB4 steps)
439
440 * Some Functions
441 * set_links: set the relation links of process. ALSO SEE: remove_links; lea
442 * -----
443 * update step 1: set child proc's parent to current process, make sure current
444 * update step 5: Insert proc_struct into hash_list and proc_list, set the relat
445
446 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
403 */
404
405 // 1. 使用 "alloc_proc" 初始化进程控制块
406 if ((proc = alloc_proc()) == NULL) // 申请内存失败
407     goto fork_out;
408
409 // 2. 使用 "setup_states" 为子进程分配并初始化内核栈
410 proc->parent = current; // 将父进程设置为当前进程
411 if (setup_kstack(proc) != 0) // 如果分配内核栈失败
412     goto bad_fork_cleanup_proc;
413
414 // 3. 使用 "copy_mm" 根据 "clone_flag" 复制/共享进程的内存管理结构
415 if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
416     goto bad_fork_cleanup_proc;
417
418 // 4. 使用 "copy_thread" 设置进程在内核正常运行和调度所需的断点和上下文
419 copy_thread(proc, stack, td);
420
421 // 5. 将设置好的控制进程模块插入到 "hash_list" 和 "proc_list" 中
422 bool intr_flag; // 标志
423 local_intr_save(intr_flag); // 屏蔽中断，并将标志设置为1
424 {
425     proc->pid = get_pid(); // 获取 pid
426     hash_proc(proc); // 建立映射
427     nr_process++; // 记录数量增加
428     list_add(&(proc->list), &(proc->list_link)); // 进程加入入到进程表中
429 }
430 local_intr_restore(intr_flag); // 恢复中断
431
432 // 6. 使用 "wakeup_proc" 将进程设置为"就绪"状态
433 wakeup_proc(proc); // 唤醒进程
434
435 // 7. 将返回结果设置为"子进程 pid"
436 ret = proc->pid;
437
438 //LABS YOUR CODE : (update LAB4 steps)
439
440 * Some Functions
441 * set_links: set the relation links of process. ALSO SEE: remove_links; lea
442 * -----
443 * update step 1: set child proc's parent to current process, make sure current
444 * update step 5: Insert proc_struct into hash_list and proc_list, set the relat
445
446 vmcc < proc.c >
lab0odes> lab4> km> process> < proc.c
403 */
404
405 // 1. 使用 "alloc_proc" 初始化进程控制块
406 if ((proc = alloc_proc()) == NULL) // 申请内存失败
407     goto fork_out;
408
409 // 2. 使用 "setup_states" 为子进程分配并初始化内核栈
410 proc->parent = current; // 将父进程设置为当前进程
411 if (setup_kstack(proc) != 0) // 如果分配内核栈失败
412     goto bad_fork_cleanup_proc;
413
414 // 3. 使用 "copy_mm" 根据 "clone_flag" 复制/共享进程的内存管理结构
415 if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
416     goto bad_fork_cleanup_proc;
417
418 // 4. 使用 "copy_thread" 设置进程在内核正常运行和调度所需的断点和上下文
419 copy_thread(proc, stack, td);
420
421 // 5. 将设置好的控制进程模块插入到 "hash_list" 和 "proc_list" 中
422 bool intr_flag; // 标志
423 local_intr_save(intr_flag); // 屏蔽中断，并将标志设置为1
424 {
425     proc->pid = get_pid
```

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，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。

查看 `load_icode` 函数，了解到其目的是加载 EOF 格式的二进制内容作为最近进程的新内容，主要工作是建立一个让用户进程正常运行的环境，具体完成的工作为：

1. 为进程创建一个新的 `mm` （调用 `mm_create` 函数）
2. 创建一个新的页表目录，并且让 `mm->pgdir` 指向该页表目录的内核虚拟地址 （调用 `setup_pgdir` 函数）
3. 复制 `TEXT/DATA` 段，在进程内存空间中建立 `BSS` 段，具体过程
 1. 得到 ELF 格式二进制程序的文件头
 2. 获取该程序的程序段头的进入点
 3. 判断该程序是否有效
 4. 寻找到程序段头
 5. 调用 `mm_map` 函数设置新的 `vma`
 6. 申请内存，然后复制每个程序段的内容到进程的内存中，具体过程：
 1. 复制二进制文件的 `TEXT/DATA` 段
 2. 建立二进制程序的 `BSS` 段
4. 创建用户栈内存
5. 设置当前进程的 `mm`，`sr3`，并且将 **CR3 寄存器** 设置为 **页目录的物理地址**
6. 设置用户环境的中断帧

阅读第6步的注释：

```
1  /* LAB5:EXERCISE1 YOUR CODE
2    * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
3    * NOTICE: If we set trapframe correctly, then the user level process can return
   to USER MODE from kernel. So
4    *      tf_cs should be USER_CS segment (see memlayout.h)
5    *      tf_ds=tf_es=tf_ss should be USER_DS segment
6    *      tf_esp should be the top addr of user stack (USTACKTOP)
7    *      tf_eip should be the entry point of this binary program (elf-
   >e_entry)
8    *      tf_eflags should be set to enable computer to produce Interrupt
9    */
```

完成完整的 `load_icode` 函数：

```
1  static int
2  load_icode(unsigned char *binary, size_t size)
3  {
4      if (current->mm != NULL)
5      {
6          panic("load_icode: current->mm must be empty.\n");
7      }
8
9      int ret = -E_NO_MEM;
10     struct mm_struct *mm;
11     //(1) create a new mm for current process
12     if ((mm = mm_create()) == NULL)
13     {
14         goto bad_mm;
15     }
16     //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
17     if (setup_pgdir(mm) != 0)
18     {
```

```

19         goto bad_pgdir_cleanup_mm;
20     }
21     //(3) copy TEXT/DATA section, build BSS parts in binary to memory space of
process
22     struct Page *page;
23     //(3.1) get the file header of the binary program (ELF format)
24     struct elfhdr *elf = (struct elfhdr *)binary;
25     //(3.2) get the entry of the program section headers of the binary program
(ELF format)
26     struct proghdr *ph = (struct proghdr *) (binary + elf->e_phoff);
27     //(3.3) This program is valid?
28     if (elf->e_magic != ELF_MAGIC)
29     {
30         ret = -E_INVALID_ELF;
31         goto bad_elf_cleanup_pgdir;
32     }
33
34     uint32_t vm_flags, perm;
35     struct proghdr *ph_end = ph + elf->e_phnum;
36     for (; ph < ph_end; ph++)
37     {
38         //(3.4) find every program section headers
39         if (ph->p_type != ELF_PT_LOAD)
40         {
41             continue;
42         }
43         if (ph->p_filesz > ph->p_memsz)
44         {
45             ret = -E_INVALID_ELF;
46             goto bad_cleanup_mmap;
47         }
48         if (ph->p_filesz == 0)
49         {
50             continue;
51         }
52         //(3.5) call mm_map fun to setup the new vma ( ph->p_va, ph->p_memsz)
53         vm_flags = 0, perm = PTE_U;
54         if (ph->p_flags & ELF_PF_X)
55             vm_flags |= VM_EXEC;
56         if (ph->p_flags & ELF_PF_W)
57             vm_flags |= VM_WRITE;
58         if (ph->p_flags & ELF_PF_R)
59             vm_flags |= VM_READ;
60         if (vm_flags & VM_WRITE)
61             perm |= PTE_W;
62         if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0)
63         {
64             goto bad_cleanup_mmap;
65         }
66         unsigned char *from = binary + ph->p_offset;
67         size_t off, size;
68         uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
69
70         ret = -E_NO_MEM;
71
72         //(3.6) alloc memory, and copy the contents of every program section
(from, from+end) to process's memory (la, la+end)
73         end = ph->p_va + ph->p_filesz;

```



```

74      //(3.6.1) copy TEXT/DATA section of binary program
75      while (start < end)
76      {
77          if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
78          {
79              goto bad_cleanup_mmap;
80          }
81          off = start - la, size = PGSIZE - off, la += PGSIZE;
82          if (end < la)
83          {
84              size -= la - end;
85          }
86          memcpy(page2kva(page) + off, from, size);
87          start += size, from += size;
88      }
89
90      //(3.6.2) build BSS section of binary program
91      end = ph->p_va + ph->p_memsz;
92      if (start < la)
93      {
94          /* ph->p_memsz == ph->p_filesz */
95          if (start == end)
96          {
97              continue;
98          }
99          off = start + PGSIZE - la, size = PGSIZE - off;
100         if (end < la)
101         {
102             size -= la - end;
103         }
104         memset(page2kva(page) + off, 0, size);
105         start += size;
106         assert((end < la && start == end) || (end >= la && start == la));
107     }
108     while (start < end)
109     {
110         if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
111         {
112             goto bad_cleanup_mmap;
113         }
114         off = start - la, size = PGSIZE - off, la += PGSIZE;
115         if (end < la)
116         {
117             size -= la - end;
118         }
119         memset(page2kva(page) + off, 0, size);
120         start += size;
121     }
122 }
123 //(4) build user stack memory
124 vm_flags = VM_READ | VM_WRITE | VM_STACK;
125 if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0)
126 {
127     goto bad_cleanup_mmap;
128 }
129 assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE, PTE_USER) != NULL);

```



```

130     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 2 * PGSIZE, PTE_USER) !=
        NULL);
131     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 3 * PGSIZE, PTE_USER) !=
        NULL);
132     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - 4 * PGSIZE, PTE_USER) !=
        NULL);
133
134     //(5) set current process's mm, sr3, and set CR3 reg = physical addr of
        Page Directory
135     mm_count_inc(mm);
136     current->mm = mm;
137     current->cr3 = PADDR(mm->pgdir);
138     lcr3(PADDR(mm->pgdir));
139
140     //(6) setup trapframe for user environment
141     struct trapframe *tf = current->tf;
142     memset(tf, 0, sizeof(struct trapframe));
143     /* LAB5:EXERCISE1 YOUR CODE
144      * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
145      * NOTICE: If we set trapframe correctly, then the user level process can
        return to USER MODE from kernel. So
146      *      tf_cs should be USER_CS segment (see memlayout.h)
147      *      tf_ds=tf_es=tf_ss should be USER_DS segment
148      *      tf_esp should be the top addr of user stack (USTACKTOP)
149      *      tf_eip should be the entry point of this binary program (elf-
        >e_entry)
150      *      tf_eflags should be set to enable computer to produce Interrupt
151      */
152     tf->tf_cs = USER_CS;
153     tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
154     tf->tf_esp = USTACKTOP;
155     tf->tf_eip = elf->e_entry;
156     tf->tf_eflags = FL_IF;
157     ret = 0;
158 out:
159     return ret;
160 bad_cleanup_mmap:
161     exit_mmap(mm);
162 bad_elf_cleanup_pgdir:
163     put_pgdir(mm);
164 bad_pgdir_cleanup_mm:
165     mm_destroy(mm);
166 bad_mm:
167     goto out;
168 }

```

回答问题

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。

查看 `do_execv` 函数的代码，其主要工作调用 `exit_mmap` 函数和 `put_pgdir` 函数，**回收**自身所占的内存空间；调用 `load_icode` 函数，使用新的程序覆盖内存空间，最终形成一个执行新程序的新进程，流程为：

1. 清理空间：

1. 如果 `mm` 不为 `NULL`，则设置页表为**内核空间页表**
2. 如果 `mm` 的引用计数 - 1 后变为 0，则说明该进程所占的内存空间空闲，分页释放进程页表所占用户空间
3. 将当前进程的 `mm` 内存管理指针设置为空
2. 加载代码：调用 `load_icode` 函数，完成 ELF 文件的读取、内存空间的申请、用户虚存空间的建立、加载并执行代码等操作

因此，答案的具体流程为：

1. `do_execv` 函数完成部分用户进程的创建，具体过程如上
2. `load_icode` 建立能够让用户进程正常运行的用户程序
3. `initproc` 按照产生系统调用的函数原路径返回，并在执行中断返回指令 `iret` 后，切换到用户进程的第一条语句位置 `_start` 处运行

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

根据上一个练习，并且查阅资料，可以了解到调用过程为

1. `do_fork`
2. `copy_mm`
3. `dup_mmap`
4. `copy_range`

查看代码，发现部分函数的代码在 LAB5 中存在需要更新的情况，于是分别进行修改

更新代码

1. `alloc_proc` 函数

更新后代码为：

```

1 // alloc_proc - alloc a proc_struct and init all fields of proc_struct
2 static struct proc_struct *
3 alloc_proc(void)
4 {
5     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
6     if (proc != NULL)
7     {
8         //LAB4:EXERCISE1 YOUR CODE
9         /*
10          * below fields in proc_struct need to be initialized
11          *      enum proc_state state;           // Process state
12          *      int pid;                          // Process ID
13          *      int runs;                        // the running times of
14          *      Proce
15          *      uintptr_t kstack;                // Process kernel stack
16          *      volatile bool need_resched;      // bool value: need to
17          *      be rescheduled to release CPU?
18          *      struct proc_struct *parent;      // the parent process
19          *      struct mm_struct *mm;            // Process's memory
20          *      management field

```

```

18     *      struct context context;                // Switch here to run
process
19     *      struct trapframe *tf;                  // Trap frame for
current interrupt
20     *      uintptr_t cr3;                          // CR3 register: the
base addr of Page Directroy Table(PDT)
21     *      uint32_t flags;                          // Process flag
22     *      char name[PROC_NAME_LEN + 1];           // Process name
23     */
24
25     proc->state = PROC_UNINIT;                      // 状态
26     proc->pid = -1;                                  // 给一个无
27     proc->runs = 0;                                  // 未运行过
28     proc->kstack = 0;                                // 没有内核栈
29     proc->need_resched = 0;                          // 不需要被 CPU 调度
30     proc->parent = NULL;                             // 没有父进程
31     proc->mm = NULL;                                 // 没有内存管理字段
32     memset(&(proc->context), 0, sizeof(struct context)); // 设置上下文
33     proc->tf = NULL;                                  // 无中断帧
34     proc->cr3 = boot_cr3;                            // 设置为内核页目录表的基址
35     proc->flags = 0;                                  // 设置进程标志
36     memset(proc->name, 0, PROC_NAME_LEN);            // 设置进程名为空
37
38     //LAB5 YOUR CODE : (update LAB4 steps)
39     /*
40     * below fields(add in LAB5) in proc_struct need to be initialized
41     *      uint32_t wait_state;                      // waiting state
42     *      struct proc_struct *cptr, *yptr, *optr;    // relations between
processes
43     */
44     // 以下是新增代码
45     proc->wait_state = 0;
46     proc->cptr = proc->yptr = proc->optr = NULL;
47 }
48 return proc;
49 }

```

2. do_fork 函数

更新后代码为：

```

1  int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
2  {
3      int ret = -E_NO_FREE_PROC;
4      struct proc_struct *proc;
5      if (nr_process >= MAX_PROCESS)
6      {
7          goto fork_out;
8      }
9      ret = -E_NO_MEM;
10     //LAB4:EXERCISE2 YOUR CODE
11     /*
12     * Some Useful MACROs, Functions and DEFINES, you can use them in below
implementation.
13     * MACROs or Functions:
14     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
15     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack

```

```

16      *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
17      *
18      *   copy_thread:  setup the trapframe on the process's kernel stack top and
19      *
20      *   hash_proc:    add proc into proc hash_list
21      *   get_pid:      alloc a unique pid for process
22      *   wakeup_proc:  set proc->state = PROC_RUNNABLE
23      * VARIABLES:
24      *   proc_list:    the process set's list
25      *   nr_process:   the number of process set
26      */
27
28      //    1. 使用 `alloc_proc` 初始化进程控制模块
29      if ((proc = alloc_proc()) == NULL) // 申请内存失败
30          goto fork_out;
31      //    2. 使用 `setup_stack` 为子进程分配并初始化内核栈
32      proc->parent = current;           // 将父进程设置为当前进程
33                                          // set child proc's parent to current
process
34      assert(current->wait_state == 0); // make sure current process's wait_state
is 0
35      if (setup_kstack(proc) != 0)      // 如果分配内核栈失败
36          goto bad_fork_cleanup_proc;
37      //    3. 使用 `copy_mm` 根据 `clone_flag` 复制/共享进程内存管理结构
38      if (copy_mm(clone_flags, proc) != 0) // 复制父进程信息失败
39          goto bad_fork_cleanup_kstack;
40      //    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
41      copy_thread(proc, stack, tf);
42      //    5. 将设置好的控制进程模块插入到 `hash_list` 和 `proc_list` 中
43      bool intr_flag;           // 标志
44      local_intr_save(intr_flag); // 屏蔽中断, 并将标志设置为1
45      {
46          proc->pid = get_pid(); // 获取 pid
47          hash_proc(proc);       // 建立映射
48          set_links(proc);       // set the relation links of process
49      }
50      local_intr_restore(intr_flag); // 恢复中断
51      //    6. 使用 `wakeup_proc` 将进程设置为**就绪**状态
52      wakeup_proc(proc); // 唤醒进程
53      //    7. 将返回结果设置为**子进程 pid**
54      ret = proc->pid;
55
56      //LAB5 YOUR CODE : (update LAB4 steps)
57      /* Some Functions
58      *   set_links: set the relation links of process. ALSO SEE: remove_links:
lean the relation links of process
59      *   -----
60      *   update step 1: set child proc's parent to current process, make sure
current process's wait_state is 0
61      *   update step 5: insert proc_struct into hash_list && proc_list, set the
relation links of process
62      */
63
64      fork_out:
65          return ret;
66
67      bad_fork_cleanup_kstack:

```

```

68     put_kstack(proc);
69     bad_fork_cleanup_proc:
70     kfree(proc);
71     goto fork_out;
72 }

```

3. idt_init 函数 (位于 kern/trap/trap.c) 中

需要设置调用中断门, 代码更新如下:

```

1  /* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S
   */
2  void idt_init(void)
3  {
4      /* LAB1 YOUR CODE : STEP 2 */
5      /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
6       *      All ISR's entry addrs are stored in __vectors. where is uintptr_t
7       *      __vectors[] ?
8       *      __vectors[] is in kern/trap/vector.S which is produced by
9       *      tools/vector.c
10      *      (try "make" command in lab1, then you will find vector.S in kern/trap
11      *      DIR)
12      *      You can use "extern uintptr_t __vectors[];" to define this extern
13      *      variable which will be used later.
14      * (2) Now you should setup the entries of ISR in Interrupt Description
15      *      Table (IDT).
16      *      Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE
17      *      macro to setup each item of IDT
18      * (3) After setup the contents of IDT, you will let CPU know where is the
19      *      IDT by using 'lidt' instruction.
20      *      You don't know the meaning of this instruction? just google it! and
21      *      check the libs/x86.h to know more.
22      *      Notice: the argument of lidt is idt_pd. try to find it!
23      */
24
25      extern uintptr_t __vectors[]; // 声明 vector
26      int i = 0;
27      for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i++) // 对于每一个表项进
28      行设置
29      {
30          SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
31      }
32      SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
33      lidt(&idt_pd); // 加载 idt
34
35      /* LAB5 YOUR CODE */
36      //you should update your lab1 code (just add ONE or TWO lines of code), let
37      //user app to use syscall to get the service of ucore
38      //so you should setup the syscall interrupt gate in here
39  }

```

4. trap_dispatch 函数 (位于 kern/trap/trap.c) 中

更新后的代码部分:

```

1  .....
2  #endif
3      /* LAB1 YOUR CODE : STEP 3 */

```

```

4         /* handle the timer interrupt */
5         /* (1) After a timer interrupt, you should record this event using a
global variable (increase it), such as ticks in kern/driver/clock.c
6         * (2) Every TICK_NUM cycle, you can print some info using a function,
such as print_ticks().
7         * (3) Too Simple? Yes, I think so!
8         */
9         {
10             ticks++;
11             if (ticks == TICK_NUM)
12             {
13                 assert(current != NULL);
14                 current->need_resched = 1;
15             }
16         }
17         break;
18         /* LAB5 YOUR CODE */
19         /* you should update your lab1 code (just add ONE or TWO lines of code):
20         * Every TICK_NUM cycle, you should set current process's current-
>need_resched = 1
21         */
22
23         case IRQ_OFFSET + IRQ_COM1:
24         .....

```

完善 copy_range 函数

查看注释，了解到具体过程：

1. 寻找 src_kvaddr (页的内核虚拟地址)
2. 寻找 dst_kvaddr (n页的内核虚拟地址)
3. dst_kvaddr 的内存复制到 src_kvaddr , 大小为 PGSIZE
4. 创建子进程页地址起始位置与物理地址的映射

完成代码：

```

1  /* copy_range - copy content of memory (start, end) of one process A to another
process B
2  * @to:    the addr of process B's Page Directory
3  * @from:  the addr of process A's Page Directory
4  * @share: flags to indicate to dup OR share. We just use dup method, so it
didn't be used.
5  *
6  * CALL GRAPH: copy_mm-->dup_mmap-->copy_range
7  */
8  int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool
share)
9  {
10     assert(start % PGSIZE == 0 && end % PGSIZE == 0);
11     assert(USER_ACCESS(start, end));
12     // copy content by page unit.
13     do
14     {
15         //call get_pte to find process A's pte according to the addr start
16         pte_t *ptep = get_pte(from, start, 0), *nptep;
17         if (ptep == NULL)
18         {
19             start = ROUNDDOWN(start + PTSIZE, PTSIZE);

```

```

20     continue;
21 }
22 //call get_pte to find process B's pte according to the addr start. If pte
is NULL, just alloc a PT
23 if (*ptep & PTE_P)
24 {
25     if ((nppte = get_pte(to, start, 1)) == NULL)
26     {
27         return -E_NO_MEM;
28     }
29     uint32_t perm = (*ptep & PTE_USER);
30     //get page from ptep
31     struct Page *page = pte2page(*ptep);
32     // alloc a page for process B
33     struct Page *npage = alloc_page();
34     assert(page != NULL);
35     assert(npage != NULL);
36     int ret = 0;
37     /* LAB5:EXERCISE2 YOUR CODE
38         * replicate content of page to npage, build the map of phy addr of nage
with the linear addr start
39         *
40         * Some Useful MACROs and DEFINES, you can use them in below
implementation.
41         * MACROs or Functions:
42         *   page2kva(struct Page *page): return the kernel virtual addr of
memory which page managed (SEE pmm.h)
43         *   page_insert: build the map of phy addr of an Page with the linear
addr la
44         *   memcpy: typical memory copy function
45         *
46         * (1) find src_kvaddr: the kernel virtual address of page
47         * (2) find dst_kvaddr: the kernel virtual address of npage
48         * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
49         * (4) build the map of phy addr of nage with the linear addr start
50         */
51     void *src_kvaddr = page2kva(page);          // 寻找旧页表的地址
52     void *dst_kvaddr = page2kva(npage);         // 获取新页表的地址
53     memcpy(dst_kvaddr, src_kvaddr, PGSIZE);     // 内存复制
54     ret = page_insert(to, npage, start, perm);  // 建立映射
55
56     assert(ret == 0);
57 }
58 start += PGSIZE;
59 } while (start != 0 && start < end);
60 return 0;
61 }

```

环境选择了 WSL

运行命令 `make qemu-nox` , 得到结果如下图


```

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
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:506:
    initproc exit.

stack traceback:
ebp = 0xc0384f88      eip = 0xc0100ba9
      arg0 = 0xc010bea8      arg1 = 0xc0384fcc      arg2 = 0x000001fa
rg3 = 0xc0384fb8
    kern/debug/kdebug.c:385: print_stackframe+22
ebp = 0xc0384fb8      eip = 0xc0100468
      arg0 = 0xc010def0      arg1 = 0x000001fa      arg2 = 0xc010df42
rg3 = 0x00000000
    kern/debug/panic.c:27: __panic+103
ebp = 0xc0384fe8      eip = 0xc0109f1f
      arg0 = 0x00000000      arg1 = 0x00000000      arg2 = 0x00000000
rg3 = 0x00000010
    kern/process/proc.c:506: do_exit+91
Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>

```

简要设计“COW 机制”

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。

根据上述概念，思路为：

- fork 时，父子进程暂时共享共同的物理内存页
- 其中一个进程需要修改内存时，额外创建一个私有物理内存页，复制共享内容，并在这上面进行修改

设计为：

- do_fork: 进行内容复制时，先不进行内容的复制，而是将父子进程的虚拟页**映射**同一个物理页，同时将父进程的 *PDE* 赋予子进程，将子进程对其的权限改为**不可写入**（*PTE_W* = 0）
- page_fault: 由于上面函数的修改，使得子进程城市修改共享页面时会出现**页访问异常**，因此需要增加对该现象的异常处理：
 - **创建**一个新的页面，将当前共享页的内容**复制**
 - 建立出错的线性地址和新创建的物理页面的**映射**，设置 *PTE* 为**非共享**
 - 查询之前的物理页面是否还有其他进程共享

- 若不是，则修改对应虚地址的 PTE，设为**非共享**，同时权限变为**可写** (PTE_W = 1)
- 若是，则不进行其他操作

练习3：阅读分析源代码

请在实验报告中简要说明你对 `fork` / `exec` / `wait` / `exit` 函数的分析。并回答如下问题：

- 请分析 `fork` / `exec` / `wait` / `exit` 在实现中是如何影响进程的执行状态的？
- 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）

分析

1. `fork` 函数

1. 检查总进程数目
2. 申请内存（调用 `alloc_proc` 函数）
3. 将父进程设置为当前进程
4. 分配**内核栈**（调用 `setup_kstack` 函数）
5. 将父进程信息复制到子进程（调用 `copy_mm` 函数）
6. 复制父进程的**中断帧**和**上下文**（调用 `copy_thread` 函数）
7. 为进程分配 `pid`（调用 `get_pid` 函数）
8. 在全局进程链表（`hash_list` 和 `proc_list`）中加入设置好的进程控制块
9. 返回获取到的 `pid`

2. `exec` 函数

1. 检查进程名称（调用 `user_mem_check` 函数）和长度
2. 清理内存空间
3. 加载要执行的程序到内存中（调用 `load_icode` 函数）

3. `wait` 函数

1. 判断 `pid`
 - 如果不为0，寻找 id 为 `pid` 的状态为 **ZOMBIE** 态的子进程
 - 如果为0，则寻找任意一个处于 **ZOMBIE** 态的子进程
2. 寻找对应状态的子进程
 - 如果不存在对应状态的子进程，则将当前状态置为 **SLEEPING** 态，标记为等待 **ZOMBIE**，调用 `schedule` 函数执行新进程，知道有对应的子进程唤醒
 - 如果存在对应状态的子进程，父进程对其进行回收，释放资源

4. `exit` 函数

1. 判断是否为用户进程
 - 是，回收内存空间
 - 不是，跳出
2. 将该子进程置为 **ZOMBIE** 态，并设置退出码为 `error_code`
3. 若父进程处于等待的状态，则使父进程回收该子进程
4. 若该子进程存在子进程，则将这些子进程的父进程设为 `init`，并且使用 `init` 回收子进程
5. 执行新进程（调用 `schedule` 函数）

回答问题

- `fork` 不影响执行状态，标记子进程为 **RUNNABLE**

- exec 不影响执行状态，会修改当前进程中的执行程序
- wait 取决于是否存在 ZOMBIE 态的子进程
 - 如果存在，不改变状态
 - 不存在，当前进程置为 SLEEPING 态，等待执行 exit 的子进程唤醒
- exit 将当前进程置为 ZOMBIE 态，唤醒父进程进行资源回收

生命周期图

```

1  process state changing:
2
3  alloc_proc
4      +
5      +
6      V
7  PROC_UNINIT -- proc_init/wakeup_proc
8
9  +-----+
10 |          RUNNING
11 |  +---<---<---+
12 |  + proc_run +
13 |  +--->--->---+
14 +--> PROC_RUNNABLE -- try_free_pages/do_wait/do_sleep --> PROC_SLEEPING --
15     A      +
16     |      +--- do_exit --> PROC_ZOMBIE
17     +
18     -----wakeup_proc-----

```

实验结果

运行 `make grade` 命令，结果如下

```

mijialong@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab5$ make grade
badsegment:                (2.6s)
  -check result:             OK
  -check output:             OK
divzero:                    (1.4s)
  -check result:             OK
  -check output:             OK
softint:                    (1.4s)
  -check result:             OK
  -check output:             OK
faultread:                  (1.4s)
  -check result:             OK
  -check output:             OK
faultreadkernel:           (1.4s)
  -check result:             OK
  -check output:             OK
hello:                      (1.4s)
  -check result:             OK
  -check output:             OK
testbss:                    (1.5s)
  -check result:             OK
  -check output:             OK
pgdir:                      (1.4s)
  -check result:             OK
  -check output:             OK
yield:                      (1.4s)
  -check result:             OK
  -check output:             OK
badarg:                     (1.4s)
  -check result:             OK
  -check output:             OK
exit:                       (1.4s)
  -check result:             OK
  -check output:             OK
spin:                       (4.4s)
  -check result:             OK
  -check output:             OK
waitkill:                   (13.5s)
  -check result:             OK
  -check output:             OK
forktest:                   (1.4s)
  -check result:             OK
  -check output:             OK
forktree:                   (1.4s)
  -check result:             OK
  -check output:             OK
Total Score: 150/150
mijialong@LAPTOP-QTCGESHO:/mnt/d/ucore_os_lab/labcodes/lab5$

```

实验总结

对比 ucore_lab 中提供的参考答案，描述区别

无区别

重要并且对应的知识点

实验：

- 从内核态切换到用户态的方法
- ELF可执行文件的格式
- 用户进程的创建和管理
- 简单的进程调度
- 系统调用的实现

理论：

- 创建、管理、切换到用户态进程的具体实现
- 加载ELF可执行文件的具体实现
- 对系统调用机制的具体实现

关系：前者的知识点为后者具体在操作系统中实现具体的功能提供了基础知识

实验中没有对应上的知识点

- 操作系统的启动
- 操作系统对内存的管理
- 进程间的共享、互斥、同步问题
- 文件系统的实现