# 练习0

修改alloc_proc函数 初始化proc->wait_state cptr optr 和 yptr

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 YOUR CODE
    /*
     * below fields in proc_struct need to be initialized
     *       enum proc_state state;                      // Process state
     *       int pid;                                    // Process ID
     *       int runs;                                   // the running times of
Proces
     *       uintptr_t kstack;                           // Process kernel stack
     *       volatile bool need_resched;                 // bool value: need to
be rescheduled to release CPU?
     *       struct proc_struct *parent;                 // the parent process
     *       struct mm_struct *mm;                       // Process's memory
management field
     *       struct context context;                     // Switch here to run
process
     *       struct trapframe *tf;                       // Trap frame for
current interrupt
     *       uintptr_t cr3;                              // CR3 register: the
base addr of Page Directroy Table(PDT)
     *       uint32_t flags;                             // Process flag
     *       char name[PROC_NAME_LEN + 1];               // Process name
     */
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;
    }
    return proc;
}
```

修改trap.c中的idt_init函数 let user app to use syscall to get the service of ucore

```
/* idt_init - initialize IDT to each of the entry points in kern/trap/vectors.S
*/
```

```
void
idt_init(void) {
     /* LAB1 YOUR CODE : STEP 2 */
     /* (1) Where are the entry addrs of each Interrupt Service Routine (ISR)?
      *     All ISR's entry addrs are stored in __vectors. where is uintptr_t
__vectors[] ?
      *     __vectors[] is in kern/trap/vector.S which is produced by
tools/vector.c
      *     (try "make" command in lab1, then you will find vector.S in kern/trap
DIR)
      *     You can use  "extern uintptr_t __vectors[];" to define this extern
variable which will be used later.
      * (2) Now you should setup the entries of ISR in Interrupt Description
Table (IDT).
      *     Can you see idt[256] in this file? Yes, it's IDT! you can use SETGATE
macro to setup each item of IDT
      * (3) After setup the contents of IDT, you will let CPU know where is the
IDT by using 'lidt' instruction.
      *     You don't know the meaning of this instruction? just google it! and
check the libs/x86.h to know more.
      *     Notice: the argument of lidt is idt_pd. try to find it!
      */
     /* LAB5 YOUR CODE */
     //you should update your lab1 code (just add ONE or TWO lines of code), let
user app to use syscall to get the service of ucore
     //so you should setup the syscall interrupt gate in here
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}
```

修改trap.c中的dispatch函数 每隔一百次计数就设置进程需要被调度

```
static void
trap_dispatch(struct trapframe *tf) {
    char c;

    int ret=0;

    switch (tf->tf_trapno) {
    case T_PGFLT:  //page fault
        if ((ret = pgfault_handler(tf)) != 0) {
            print_trapframe(tf);
            if (current == NULL) {
                panic("handle pgfault failed. ret=%d\n", ret);
            }
            else {
                if (trap_in_kernel(tf)) {
                    panic("handle pgfault failed in kernel mode. ret=%d\n", ret);
                }
                cprintf("killed by kernel.\n");
                panic("handle user mode pgfault failed. ret=%d\n", ret);
                do_exit(-E_KILLED);
```

```
            }
        }
        break;
    case T_SYSCALL:
        syscall();
        break;
    case IRQ_OFFSET + IRQ_TIMER:
#if 0
    LAB3 : If some page replacement algorithm(such as CLOCK PRA) need tick to change the priority of pages,
    then you can add code here.
#endif
        /* LAB1 YOUR CODE : STEP 3 */
        /* handle the timer interrupt */
        /* (1) After a timer interrupt, you should record this event using a global variable (increase it), such as ticks in kern/driver/clock.c
         * (2) Every TICK_NUM cycle, you can print some info using a funciton, such as print_ticks().
         * (3) Too Simple? Yes, I think so!
         */
        /* LAB5 YOUR CODE */
        /* you should upate you lab1 code (just add ONE or TWO lines of code):
         *    Every TICK_NUM cycle, you should set current process's current->need_resched = 1
         */
        ticks ++;
        if (ticks % TICK_NUM == 0) {
            assert(current != NULL);
            current->need_resched = 1;
        }
        break;
    case IRQ_OFFSET + IRQ_COM1:
        c = cons_getc();
        cprintf("serial [%03d] %c\n", c, c);
        break;
    case IRQ_OFFSET + IRQ_KBD:
        c = cons_getc();
        cprintf("kbd [%03d] %c\n", c, c);
        break;
    //LAB1 CHALLENGE 1 : YOUR CODE you should modify below codes.
    case T_SWITCH_TOU:
    case T_SWITCH_TOK:
        panic("T_SWITCH_** ??\n");
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        print_trapframe(tf);
        if (current != NULL) {
            cprintf("unhandled trap.\n");
            do_exit(-E_KILLED);
        }
        // in kernel, it must be a mistake
        panic("unexpected trap in kernel.\n");

    }
```

```
}
```

修改do_fork函数

增加使用set_links设置进程之间的关系

```
// set_links - set the relation links of process
static void
set_links(struct proc_struct *proc) {
    list_add(&proc_list, &(proc->list_link));
    proc->yptr = NULL;
    if ((proc->optr = proc->parent->cptr) != NULL) {
        proc->optr->yptr = proc;
    }
    proc->parent->cptr = proc;
    nr_process ++;
}
```

修改后函数如下：

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a
kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's
proc->tf
 */
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * Some Useful MACROs, Functions and DEFINEs, you can use them in below
implementation.
     * MACROs or Functions:
     *   alloc_proc:   create a proc struct and init fields (lab4:exercise1)
     *   setup_kstack: alloc pages with size KSTACKPAGE as process kernel stack
     *   copy_mm:      process "proc" duplicate OR share process "current"'s mm
according clone_flags
     *                 if clone_flags & CLONE_VM, then "share" ; else "duplicate"
     *   copy_thread:  setup the trapframe on the  process's kernel stack top and
     *                 setup the kernel entry point and stack of process
     *   hash_proc:    add proc into proc hash_list
     *   get_pid:      alloc a unique pid for process
     *   wakeup_proc:  set proc->state = PROC_RUNNABLE
     * VARIABLES:
     *   proc_list:    the process set's list
     *   nr_process:   the number of process set
     */

    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
```

```
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;
    assert(current->wait_state == 0);

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

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

    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

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

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

# 练习1 加载应用程序并执行（需要编码）

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

补全proc.c的load_icode函数如下

```
/* load_icode - load the content of binary program(ELF format) as the new content
of current process
```

```
 * @binary:  the memory addr of the content of binary program
 * @size:  the size of the content of binary program
 */
static int
load_icode(unsigned char *binary, size_t size) {
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    //(3) copy TEXT/DATA section, build BSS parts in binary to memory space of
process
    struct Page *page;
    //(3.1) get the file header of the bianry program (ELF format)
    struct elfhdr *elf = (struct elfhdr *)binary;
    //(3.2) get the entry of the program section headers of the bianry program
(ELF format)
    struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff);
    //(3.3) This program is valid?
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    uint32_t vm_flags, perm;
    struct proghdr *ph_end = ph + elf->e_phnum;
    for (; ph < ph_end; ph ++) {
    //(3.4) find every program section headers
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            continue ;
        }
    //(3.5) call mm_map fun to setup the new vma ( ph->p_va, ph->p_memsz)
        vm_flags = 0, perm = PTE_U;
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        unsigned char *from = binary + ph->p_offset;
        size_t off, size;
```

```
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

        ret = -E_NO_MEM;

     //(3.6) alloc memory, and  copy the contents of every program section (from,
from+end) to process's memory (la, la+end)
        end = ph->p_va + ph->p_filesz;
     //(3.6.1) copy TEXT/DATA section of bianry program
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memcpy(page2kva(page) + off, from, size);
            start += size, from += size;
        }

      //(3.6.2) build BSS section of binary program
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    //(4) build user stack memory
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !=
0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);
```

```
    //(5) set current process's mm, sr3, and set CR3 reg = physical addr of Page
Directory
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    /* LAB5:EXERCISE1 YOUR CODE
     * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
     * NOTICE: If we set trapframe correctly, then the user level process can
return to USER MODE from kernel. So
     *          tf_cs should be USER_CS segment (see memlayout.h)
     *          tf_ds=tf_es=tf_ss should be USER_DS segment
     *          tf_esp should be the top addr of user stack (USTACKTOP)
     *          tf_eip should be the entry point of this binary program (elf-
>e_entry)
     *          tf_eflags should be set to enable computer to produce Interrupt
     */
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = USTACKTOP;
    tf->tf_eip = elf->e_entry;
    tf->tf_eflags = FL_IF;
    ret = 0;
out:
    return ret;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

## 代码设计过程：

设置为用户态用户段

将栈指针执行用户栈顶

将程序计数器eip执行二进制程序的入口点

设置允许中断

## 请在实验报告中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。

1. 用户态进程 exec 系统调用，转入系统调用的处理；

2. 在经过了正常的中断处理例程之后，最终控制权转移到了 syscall.c 中的 syscall 函数，然后根据系统调用号转移给了 sys_exec 函数，在该函数中调用了上文中提及的 do_execve 函数来完成指定应用程序的加载；
3. 在do_execve中推出当前进程的页表，换用 kernel 的 PDT 之后，使用 load_icode 函数，完成了对整个用户线程内存空间的初始化，包括堆栈的设置以及将 ELF 可执行文件的加载，之后通过修改当前系统调用的 trapframe，使得最终中断返回的时候能够切换到用户态，从新程序的入口点开始继续执行指令

# 练习2 父进程复制自己的内存空间给子进程（需要编码）

创建子进程的函数do_fork在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。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。

补全pmm.c中的copy_range函数

```
/* copy_range - copy content of memory (start, end) of one process A to another process B
 * @to:    the addr of process B's Page Directory
 * @from:  the addr of process A's Page Directory
 * @share: flags to indicate to dup OR share. We just use dup method, so it didn't be used.
 *
 * CALL GRAPH: copy_mm-->dup_mmap-->copy_range
 */
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr start. If pte is NULL, just alloc a PT
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
        uint32_t perm = (*ptep & PTE_USER);
        //get page from ptep
        struct Page *page = pte2page(*ptep);
        // alloc a page for process B
        struct Page *npage=alloc_page();
        assert(page!=NULL);
        assert(npage!=NULL);
        int ret=0;
        /* LAB5:EXERCISE2 YOUR CODE
         * replicate content of page to npage, build the map of phy addr of nage with the linear addr start
```

```
         *
         * Some Useful MACROs and DEFINEs, you can use them in below
implementation.
         * MACROs or Functions:
         *    page2kva(struct Page *page): return the kernel vritual addr of
memory which page managed (SEE pmm.h)
         *    page_insert: build the map of phy addr of an Page with the linear
addr la
         *    memcpy: typical memory copy function
         *
         * (1) find src_kvaddr: the kernel virtual address of page
         * (2) find dst_kvaddr: the kernel virtual address of npage
         * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
         * (4) build the map of phy addr of  nage with the linear addr start
         */
        void * kva_src = page2kva(page);
        void * kva_dst = page2kva(npage);

        memcpy(kva_dst, kva_src, PGSIZE);

        ret = page_insert(to, npage, start, perm);
        assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

说明：

首先获取page和npage的内核虚拟地址

然后调用memcpy将内存从src复制到dst

最后建立地址映射

# 请在实验报告中简要说明如何设计实现"Copy on Write 机制"，给出概要设计

- fork创建出的子进程，**与父进程共享内存空间**。也就是说，如果子进程**不对内存空间进行写入操作的话，内存空间中的数据并不会复制给子进程**，这样创建子进程的速度就很快了！(不用复制，直接引用父进程的物理空间)。
- 并且如果在fork函数返回之后，子进程**第一时间**exec一个新的可执行映像，那么也不会浪费时间和内存空间了。

> 在fork之后exec之前两个进程**用的是相同的物理空间**（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，两者的虚拟空间不同，但其对应的**物理空间是同一个**。

> 当父子进程中**有更改相应段的行为发生时**，再**为子进程相应的段分配物理空间**。

> 如果不是因为exec，内核会给子进程的数据段、堆栈段分配相应的物理空间（至此两者有各自的进程空间，互不影响），而代码段继续共享父进程的物理空间（两者的代码完全相同）。

> 而如果是因为exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

fork()之后，kernel把父进程中所有的内存页的权限都设为read-only，然后子进程的地址空间指向父进程。当父子进程都只读内存时，相安无事。当其中某个进程写内存时，CPU硬件检测到内存页是read-only的，于是触发页异常中断（page-fault），陷入kernel的一个中断例程。中断例程中，kernel就会**把触发的异常的页复制一份**，于是父子进程各自持有独立的一份。

# 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）

## fork

```
1.调用alloc_proc以分配进程的结构体
2.调用setup_kstack为子进程分配内核堆栈
3.根据clone_flag调用copy_mm以复制或共享内存管理
4.调用copy_thread以在proc_struct中设置 trapframe 和上下文
5.将proc_struct插入hash_list和proc_list
6.调用wakeup_proc以使新的子进程可运行
7.使用子进程的pid设置返回值
```

## exec

```
1.调用exit_mmap和put_pgdir为当前进程重新分配内存
2.调用load_icode根据二进制程序重新设置新的内存空间
```

## wait

```
1、 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;
2、 如果此子进程的执行状态不为PROC_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC_SLEEPING（睡眠），睡眠原因为WT_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;
3、 如果此子进程的执行状态为 PROC_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc_list和hash_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。
```

## exit

```
1.调用exit_mmap & put_pgdir & mm_destroy 以释放进程的所有内存空间
2.将进程的状态设置为PROC_ZOMBIE，然后调用wakeup_proc（parent） 要求父级收回自身。
3.调用调度程序以切换到其他进程
```

# 测试结果

```
-> % make grade
badsegment:              (s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (s)
```

```
  -check result:                            OK
  -check output:                            OK
softint:                 (s)
  -check result:                            OK
  -check output:                            OK
faultread:               (s)
  -check result:                            OK
  -check output:                            OK
faultreadkernel:         (s)
  -check result:                            OK
  -check output:                            OK
hello:                   (s)
  -check result:                            OK
  -check output:                            OK
testbss:                 (s)
  -check result:                            OK
  -check output:                            OK
pgdir:                   (s)
  -check result:                            OK
  -check output:                            OK
yield:                   (s)
  -check result:                            OK
  -check output:                            OK
badarg:                  (s)
  -check result:                            OK
  -check output:                            OK
exit:                    (s)
  -check result:                            OK
  -check output:                            OK
spin:                    (s)
  -check result:                            OK
  -check output:                            OK
waitkill:                (s)
  -check result:                            OK
  -check output:                            OK
forktest:                (s)
  -check result:                            OK
  -check output:                            OK
forktree:                (s)
  -check result:                            OK
  -check output:                            OK
Total Score: 150/150
```

# 扩展练习 Challenge ：实现 Copy on Write 机制

首先学习什么是copy on write机制：

**写时复制**（**Copy-on-write**，简称**COW**）其核心思想是，如果有多个调用者（callers）同时请求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的。此作法主要的优点是如果调用者没有修改该资源，就不会有副本（private copy）被创建，因此多个调用者只是读取操作时可以共享同一份资源。

也就是说 在fork调用创建新进程时 不直接复制一块新的内存空间 而共享父进程的内存空间 为了保证父进程正常运行 需要新进程对这段内存权限设置为只读

如果新进程需要在这段共享内存内写入新数据 则在操作系统内部就会由于写只读内存引发页错误 在页错误处理函数中 可以进行判断 如果是这种情况导致的页错误 那么再采取拷贝原有内存的方式为这个新进程写入新的数据

这样的好处就是如果新进程不需要在原有内存中写入新数据 那么使用共享内存就会大大提高性能 因为不需要复制内存的开销 并且可以节省内存占用 因为多个进程使用了同一段共享内存

实现方式如下

首先需要修改copy_range函数 在share为1的情况下 执行写时复制 而不是分配新的内存空间并完全拷贝原有进程的数据

```
/* copy_range - copy content of memory (start, end) of one process A to another
process B
 * @to:    the addr of process B's Page Directory
 * @from:  the addr of process A's Page Directory
 * @share: flags to indicate to dup OR share. We just use dup method, so it
didn't be used.
 *
 * CALL GRAPH: copy_mm-->dup_mmap-->copy_range
 */
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr start. If
pte is NULL, just alloc a PT
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            //get page from ptep
            struct Page *page = pte2page(*ptep);
            int ret=0;
            if(share){
                // lab5 challenge
                // if use COW
                cprintf("Sharing the page 0x%x\n", page2kva(page));
                // 物理页面共享，并设置两个PTE上的标志位为只读
                page_insert(from, page, start, perm & ~PTE_W);
                ret = page_insert(to, page, start, perm & ~PTE_W);
            }else{
                // alloc a page for process B
                struct Page *npage=alloc_page();
```

```
                assert(page!=NULL);
                assert(npage!=NULL);

                /* LAB5:EXERCISE2 YOUR CODE
                * replicate content of page to npage, build the map of phy addr
of nage with the linear addr start
                *
                * Some Useful MACROs and DEFINEs, you can use them in below
implementation.
                * MACROs or Functions:
                *    page2kva(struct Page *page): return the kernel vritual addr
of memory which page managed (SEE pmm.h)
                *    page_insert: build the map of phy addr of an Page with the
linear addr la
                *    memcpy: typical memory copy function
                *
                * (1) find src_kvaddr: the kernel virtual address of page
                * (2) find dst_kvaddr: the kernel virtual address of npage
                * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
                * (4) build the map of phy addr of  nage with the linear addr
start
                */
                void * kva_src = page2kva(page);
                void * kva_dst = page2kva(npage);

                memcpy(kva_dst, kva_src, PGSIZE);

                ret = page_insert(to, npage, start, perm);
            }
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

修改do_pgfalut函数 处理发生写入只读页面造成的页错误

```
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;
    //try to find a vma which include addr
    struct vma_struct *vma = find_vma(mm, addr);

    pgfault_num++;
    //If the addr is in the range of a mm's vma?
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }
    //check the error_code
    switch (error_code & 3) {
    default:
            /* error code flag : default is 3 ( W/R=1, P=1): write, present */
    case 2: /* error code flag : (W/R=1, P=0): write, not present */
        if (!(vma->vm_flags & VM_WRITE)) {
```

```
            cprintf("do_pgfault failed: error code flag = write AND not present,
but the addr's vma cannot write\n");
            goto failed;
        }
        break;
    case 1: /* error code flag : (W/R=0, P=1): read, present */
        cprintf("do_pgfault failed: error code flag = read AND present\n");
        goto failed;
    case 0: /* error code flag : (W/R=0, P=0): read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            cprintf("do_pgfault failed: error code flag = read AND not present,
but the addr's vma cannot read or exec\n");
            goto failed;
        }
    }
    /* IF (write an existed addr ) OR
     *    (write an non_existed addr && addr is writable) OR
     *    (read  an non_existed addr && addr is readable)
     * THEN
     *    continue process
     */
    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;
    /*LAB3 EXERCISE 1: YOUR CODE
    * Maybe you want help comment, BELOW comments can help you finish the code
    *
    * Some Useful MACROs and DEFINEs, you can use them in below implementation.
    * MACROs or Functions:
    *   get_pte : get an pte and return the kernel virtual address of this pte
for la
    *             if the PT contians this pte didn't exist, alloc a page for PT
(notice the 3th parameter '1')
    *   pgdir_alloc_page : call alloc_page & page_insert functions to allocate a
page size memory & setup
    *             an addr map pa<--->la with linear address la and the PDT pgdir
    * DEFINES:
    *   VM_WRITE  : If vma->vm_flags & VM_WRITE == 1/0, then the vma is
writable/non writable
    *   PTE_W           0x002                   // page table/directory entry
flags bit : Writeable
    *   PTE_U           0x004                   // page table/directory entry
flags bit : User can access
    * VARIABLES:
    *   mm->pgdir : the PDT of these vma
    *
    */
#if 0
    /*LAB3 EXERCISE 1: YOUR CODE*/
    ptep = ???              //(1) try to find a pte, if pte's PT(Page Table)
isn't existed, then create a PT.
    if (*ptep == 0) {
```

```
                            //(2) if the phy addr isn't exist, then alloc a page
& map the phy addr with logical addr

    }
    else {
    /*LAB3 EXERCISE 2: YOUR CODE
    * Now we think this pte is a  swap entry, we should load data from disk to a
page with phy addr,
    * and map the phy addr with logical addr, trigger swap manager to record the
access situation of this page.
    *
    *  Some Useful MACROs and DEFINEs, you can use them in below implementation.
    *  MACROs or Functions:
    *    swap_in(mm, addr, &page) : alloc a memory page, then according to the
swap entry in PTE for addr,
    *                               find the addr of disk page, read the content
of disk page into this memroy page
    *    page_insert : build the map of phy addr of an Page with the linear addr
la
    *    swap_map_swappable : set the page swappable
    */
        if(swap_init_ok) {
            struct Page *page=NULL;
                                    //(1)According to the mm AND addr, try to
load the content of right disk page
                                    //    into the memory which page managed.
                                    //(2) According to the mm, addr AND page,
setup the map of phy addr <---> logical addr
                                    //(3) make the page swappable.
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
            goto failed;
        }
   }
#endif
    // try to find a pte, if pte's PT(Page Table) isn't existed, then create a
PT.
    // (notice the 3th parameter '1')
    ptep = get_pte(mm->pgdir, addr, 1);
    if (ptep == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }

    if (*ptep == 0) {
        // if the phy addr isn't exist, then alloc a page & map the phy addr with
logical addr
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
    else {
        struct Page *page=NULL;
        if(*ptep & PTE_P){
            // 如果当前页错误的原因是写入了只读页面
            // 写时复制：复制一块内存给当前进程
```

```
            cprintf("\n\nCOW: ptep 0x%x, pte 0x%x\n",ptep, *ptep);
            // 原先所使用的只读物理页
            page = pte2page(*ptep);
            // 如果该物理页面被多个进程引用
            if(page_ref(page) > 1)
            {
                // 释放当前PTE的引用并分配一个新物理页
                struct Page* newPage = pgdir_alloc_page(mm->pgdir, addr, perm);
                void * kva_src = page2kva(page);
                void * kva_dst = page2kva(newPage);
                // 拷贝数据
                memcpy(kva_dst, kva_src, PGSIZE);
            }
            // 如果该物理页面只被当前进程所引用,即page_ref等1
            else
                // 则可以直接执行page_insert，保留当前物理页并重设其PTE权限。
                page_insert(mm->pgdir, page, addr, perm);

        }else{
            // if this pte is a swap entry, then load data from disk to a page
with phy addr
            // and call page_insert to map the phy addr with logical addr
            if(swap_init_ok) {

                ret = swap_in(mm, addr, &page);
                if (ret != 0) {
                    cprintf("swap_in in do_pgfault failed\n");
                    goto failed;
                }
                page_insert(mm->pgdir, page, addr, perm);
                swap_map_swappable(mm, addr, page, 1);
                page->pra_vaddr = addr;
            }
            else {
                cprintf("no swap_init_ok but ptep is %x, failed\n",*ptep);
                goto failed;
            }
        }

    }
    ret = 0;
failed:
    return ret;
}
```

测试：

```
-> % make grade
badsegment:              (s)
  -check result:                    OK
  -check output:                    OK
divzero:                 (s)
  -check result:                    OK
  -check output:                    OK
softint:                 (s)
```

```
  -check result:                                   OK
  -check output:                                   OK
faultread:                  (s)
  -check result:                                   OK
  -check output:                                   OK
faultreadkernel:            (s)
  -check result:                                   OK
  -check output:                                   OK
hello:                      (s)
  -check result:                                   OK
  -check output:                                   OK
testbss:                    (s)
  -check result:                                   OK
  -check output:                                   OK
pgdir:                      (s)
  -check result:                                   OK
  -check output:                                   OK
yield:                      (s)
  -check result:                                   OK
  -check output:                                   OK
badarg:                     (s)
  -check result:                                   OK
  -check output:                                   OK
exit:                       (s)
  -check result:                                   OK
  -check output:                                   OK
spin:                       (s)
  -check result:                                   OK
  -check output:                                   OK
waitkill:                   (s)
  -check result:                                   OK
  -check output:                                   OK
forktest:                   (s)
  -check result:                                   OK
  -check output:                                   OK
forktree:                   (s)
  -check result:                                   OK
  -check output:                                   OK
Total Score: 150/150
```

没有出现问题 说明实验成功