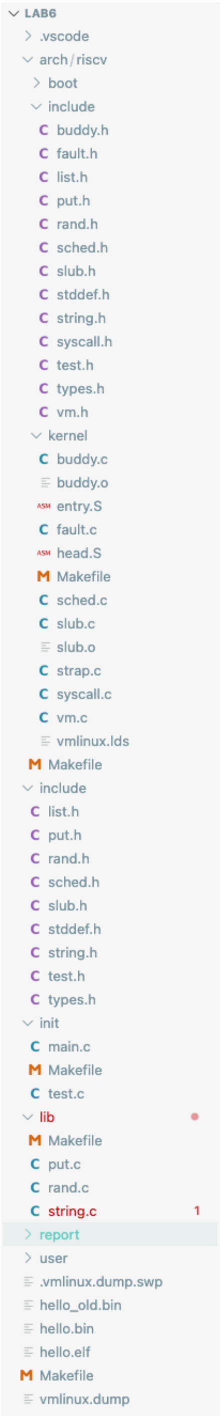
OS Lab6

> 郭子扬 3180105514

实验步骤

环境搭建



管理空闲内存空间

环境变量

```
#define P_PAGE 0x1000
#define parent(x) (((x + 1) >> 1) - 1)
#define lson(x) (((x + 1) << 1) - 1)
#define rson(x) ((x + 1) << 1)

struct buddy buddy;
unsigned ins_bitmap[2 * P_PAGE - 1];
unsigned long page_offset = 0x80000000ul;

extern unsigned long *_end;
```

P_PAGE为总页数，16M/4K = 0x1000

parent、lson、rson分别为计算父亲、左儿子和右儿子的标号

ins_bitmap是定义在程序字段的bitmap，buddy.bitmap指向这个数组的开头

page_offset是决定alloc_pages返回的地址的偏移量，起始置为0x80000000，alloc_pages返回物理地址，当paging_init结束后置page_offset为0xffffffe000000000，alloc_pages返回虚拟地址。

_end为从lds中引入的标签

init_buddy_system

```
void init_buddy_system(void) {
    buddy.size = P_PAGE;
    buddy.bitmap = ins_bitmap;
    unsigned long x, y, i;
    for (x = buddy.size, y = 1, i = 0; x > 0; x >>= 1, i += y, y <<= 1) {
        int j;
        for (j = 0; j < y; ++j) {
            ins_bitmap[i + j] = x;
        }
    }
    alloc_pages((((unsigned long)&_end - 0x80000000ul) >> 12));
}
```

alloc_pages

```
void *alloc_pages(int n) {
    int i;
    for (i = 0; ; ++i) {
        if ((1 << i) >= n) {
            n = (1 << i);
            break;
```

```
        }
    }
    if (ins_bitmap[0] < n) {
        return 0;
    }
    i = 0;
    int full_size = buddy.size;
    unsigned t_bitmap;
    while (i < 2 * P_PAGE - 1) {
        if (ins_bitmap[i] == n && ins_bitmap[i] == full_size) {
            t_bitmap = ins_bitmap[i];
            ins_bitmap[i] = 0;
            int ii = i;
            while (ii > 0) {
                ii = parent(ii);
                ins_bitmap[ii] = ins_bitmap[rson(ii)] > ins_bitmap[lson(ii)] ?
                        ins_bitmap[rson(ii)] : ins_bitmap[lson(ii)];
            }
            break;
        } else {
            if (ins_bitmap[lson(i)] >= n) {
                full_size >>= 1;
                i = lson(i);
            } else if (ins_bitmap[rson(i)] >= n) {
                full_size >>= 1;
                i = rson(i);
            }
        }
    }
    unsigned long addr = (((i + 1) * t_bitmap - buddy.size) << 12) +
page_offset;
    if (addr >= 0xffffffe000000000) {
        puts("[S] Buddy allocate addr: ");
        puti64(addr);
        puts("\n");
    }
    return addr;
}
```

注意使用page_offset来控制返回的地址是物理地址还是虚拟地址。

free_pages

```
void free_pages(void* x) {
    unsigned long index = x;
    index -= page_offset;
    index >>= 12;
    index += buddy.size;
```

```
    int i = 0;
    int full_size = buddy.size;
    while (i < 2 * P_PAGE - 1) {
        if ((i + 1) * ins_bitmap[i] == index) {
            if (ins_bitmap[i] == 0) {
                ins_bitmap[i] = full_size;
                int ii = i;
                int t_full_size = full_size;
                while (ii > 0) {
                    ii = parent(ii);
                    t_full_size <<= 1;
                    if (ins_bitmap[lson(ii)] == (t_full_size >> 1)
                        && ins_bitmap[rson(ii)] == (t_full_size >> 1)) {
                        ins_bitmap[ii] = t_full_size;
                    } else {
                        ins_bitmap[ii] = ins_bitmap[rson(ii)] >
ins_bitmap[lson(ii)] ?
                                ins_bitmap[rson(ii)] : ins_bitmap[lson(ii)];
                    }
                }
            } else {
                full_size >>= 1;
                i = lson(i);
            }
        } else {
            full_size >>= 1;
            i = rson(i);
        }
    }
}
```

实现统一内存分配接口

```
void *kmalloc(size_t size)
{
  int objindex;
  void *p;

  if(size == 0)
    return NULL;

  // size 若在 kmem_cache_objsize 所提供的范围之内，则使用 slub allocator 来分配内存
  for(objindex = 0; objindex < NR_PARTIAL; objindex ++){
    if (size <= kmem_cache_objsize[objindex]) {
      p = kmem_cache_alloc(slub_allocator[objindex]);
      break;
    }
      // YOUR CODE HERE
```

```
  }

  // size 若不在 kmem_cache_objsize 范围之内，则使用 buddy system 来分配内存
  if(objindex >= NR_PARTIAL){
    // YOUR CODE HERE
    p = alloc_pages(size >> 12);

    set_page_attr(p, (size-1) / PAGE_SIZE, PAGE_BUDDY);
  }

  return p;
}
```

```
void kfree(const void *addr)
{
    struct page *page;
    if(addr == NULL)
        return;
    // 获得地址所在页的属性
    // YOUR CODE HERE
    page = ADDR_TO_PAGE(addr);
    // 判断当前页面属性
    if(page->flags == PAGE_BUDDY){
        // YOUR CODE HERE
        free_pages(addr);
        clear_page_attr(ADDR_TO_PAGE(addr)->header);
    } else if(page->flags == PAGE_SLUB){
        // YOUR CODE HERE
        kmem_cache_free(addr);
    }
    return;
}
```

为 mm_struct 添加 vm_area_struct 数据结构

```
typedef struct { unsigned long pgprot; } pgprot_t;
struct vm_area_struct {
    /* Our start address within vm_area. */
    unsigned long vm_start;
    /* The first byte after our end address within vm_area. */
    unsigned long vm_end;
    /* linked list of VM areas per task, sorted by address. */
    struct vm_area_struct *vm_next, *vm_prev;
    /* The address space we belong to. */
    struct mm_struct *vm_mm;
    /* Access permissions of this VMA. */
    pgprot_t vm_page_prot;
```

```c
    /* Flags*/
    unsigned long vm_flags;
};

struct mm_struct {
    unsigned long *pgtbl;
    unsigned long text_start;
    unsigned long text_end;
    unsigned long stack_top;
    unsigned long pa_for_stack;
    struct vm_area_struct *vm_area_head;
};
```

实现mmap/munmap/mprotect的SysCall

实现mmap函数

```c

#define __off_t unsigned long

unsigned long get_unmapped_area(size_t length) {
    unsigned long i = 0;
    for (i = 0; ; ++i) {
        struct vm_area_struct *ptr;
        int flag = 0;
        for (ptr = current->thread.mm->vm_area_head; ptr != NULL; ptr = ptr-
>vm_next) {
            if (ptr->vm_start < i + length && ptr->vm_end > i) {
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            return i;
        }
    }

}

void *do_mmap(struct mm_struct *mm, void *start, size_t length, int prot) {
    struct vm_area_struct *ptr;
    for (ptr = mm->vm_area_head; ptr != NULL; ptr = ptr->vm_next) {
        if (ptr->vm_start < (unsigned long)start + length && ptr->vm_end >
(unsigned long)start) {
            start = get_unmapped_area(length);
            break;
        }
    }
```

```
    struct vm_area_struct *new_vm_area = kmalloc(sizeof(struct
vm_area_struct));
    new_vm_area->vm_start = (unsigned long)start;
    new_vm_area->vm_end = start + length;
    new_vm_area->vm_flags = prot;
    new_vm_area->vm_mm = mm;
    new_vm_area->vm_page_prot.pgprot = ((prot & PROT_READ ? 1 : 0) | (prot &
PROT_WRITE ? 2 : 0) | (prot & PROT_EXEC ? 4 : 0)) << 1;
    if (mm->vm_area_head == NULL) {
        mm->vm_area_head = new_vm_area;
    } else {
        for (ptr = mm->vm_area_head; ptr->vm_next != NULL; ptr = ptr->vm_next)
;
        ptr->vm_next = new_vm_area;
        new_vm_area->vm_prev = ptr;
    }
    return start;
}

void *mmap (void *__addr, size_t __len, int __prot,
            int __flags, int __fd, __off_t __offset) {
    return do_mmap(current->thread.mm, __addr, __len, __prot);
}
```

修改task_init函数代码，更改为需求分页机制

```
sscratch_top = kmalloc(PAGE_SIZE) + PAGE_SIZE;
task[1]->state = TASK_RUNNING;
#ifdef SJF
task[1]->counter = rand();
#endif
#ifdef PRIORITY
task[1]->counter = 8 - 1;
#endif
task[1]->priority = 5;
task[1]->blocked = 0;
task[1]->pid = 1;
task[1]->thread.sp = USER_END;

task[1]->thread.sepc = 0;
task[1]->thread.ra = (unsigned long)task_first_ret;
task[1]->thread.sscratch = sscratch_top;
task[1]->thread.mm = kmalloc(sizeof(struct mm_struct));
task[1]->thread.mm->pgtbl = (unsigned long)kmalloc(PAGE_SIZE) - page_offset +
0x80000000ul;
task[1]->thread.mm->vm_area_head = NULL;
task[1]->thread.mm->pa_for_stack = 0;
unsigned long pgtbl_va = (unsigned long)task[1]->thread.mm->pgtbl;
```

```
initial_pgtbl(pgtbl_va);

do_mmap(task[1]->thread.mm, 0, PAGE_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC);
do_mmap(task[1]->thread.mm, USER_END - PAGE_SIZE, PAGE_SIZE, PROT_READ |
PROT_WRITE);
```

实现munmap函数

```
int munmap(void *start, size_t length) {
    struct vm_area_struct *found = NULL;
    if (current->thread.mm->vm_area_head == NULL) {
        return -1;
    } else {
        struct vm_area_struct *ptr;
        for (ptr = current->thread.mm->vm_area_head; ptr != NULL; ptr = ptr-
>vm_next) {
            if (ptr->vm_start == (unsigned long)start && ptr->vm_end ==
(unsigned long)start + length) {
                found = ptr;
                break;
            }
        }
    }
    if (found == NULL) {
        return -1;
    }
    free_page_tables(current->thread.mm->pgtbl, start, length, 1);
    if (found->vm_prev == NULL && found->vm_next == NULL) {
        current->thread.mm->vm_area_head = NULL;
    } else if (found->vm_prev == NULL) {
        found->vm_next->vm_prev = NULL;
        current->thread.mm->vm_area_head = found->vm_next;
    } else if (found->vm_next == NULL) {
        found->vm_prev->vm_next = NULL;
    } else {
        found->vm_prev->vm_next = found->vm_next;
        found->vm_next->vm_prev = found->vm_prev;
    }
    kfree(found);
    return 0;
}
```

实现free_page_tables函数

```
unsigned long free_table_dir(unsigned long *tblptr, unsigned long va, unsigned
long right) {
    tblptr = (unsigned long)tblptr - 0x80000000ul + page_offset;
```

```
    unsigned long tbl_index = (va >> (unsigned long)right) & (unsigned
long)0x1FF;

    if ((tblptr[tbl_index]) & 1) {
        if (((tblptr[tbl_index] >> 1) & 0x7) == 0) {
            return free_table_dir(((((tblptr[tbl_index]) >> 10) & (unsigned
long)0xFFFFFFFFFFF) << 12), va, right - 9);
        } else {
            tblptr[tbl_index] ^= 1;
            return (tblptr[tbl_index] >> 10) << 12;
        }
    } else {
        return NULL;
    }
}

void free_page_tables(unsigned long pgtbl, unsigned long va, unsigned long sz,
int free_frame) {
    unsigned long i = 0;
    unsigned long va_aligned = va;// >> 12) << 12;
    for (i = 0; i < sz; i += 0x1000) {
        unsigned long pa = free_table_dir(pgtbl, va_aligned + i, 30);
        if (pa != NULL && free_frame) {
            kfree(pa + kmem_struct.virtual_offset);
        }
    }
}
```

实现mprotect函数

```
int mprotect (void *__addr, size_t __len, int __prot) {
    struct vm_area_struct *found = NULL;
    if (current->thread.mm->vm_area_head == NULL) {
        return -1;
    } else {
        struct vm_area_struct *ptr;
        for (ptr = current->thread.mm->vm_area_head; ptr != NULL; ptr = ptr-
>vm_next) {
            if (ptr->vm_start == (unsigned long)__addr && ptr->vm_end ==
(unsigned long)__addr + __len) {
                found = ptr;
                break;
            }
        }
    }
    if (found == NULL) {
        return -1;
    }
```

```
    protect_page_tables(current->thread.mm->pgtbl, __addr, __len, __prot);
    found->vm_flags = __prot;
    found->vm_page_prot.pgprot = __prot << 1;
    return 0;
}
```

实现 Page Fault 的检测与处理

```
void do_page_fault() {

    unsigned long scause;
    __asm__ __volatile__ ("csrr %0, scause\n\t" : "=r" (scause));

    unsigned long sepc;
    __asm__ __volatile__ ("csrr %0, sepc\n\t" : "=r" (sepc));

    unsigned long stval;
    __asm__ __volatile__ ("csrr %0, stval\n\t" : "=r" (stval));
    puts("[S] PAGE_FAULT: PID = ");
    puti(current->pid);
    puts("\n[S] PAGE_FAULT: scause: ");
    puti(scause);
    puts(", sepc: 0x");
    puti64(sepc);
    puts(", badaddr: 0x");
    puti64(stval);
    puts("\n");

    struct vm_area_struct *found = NULL;
    if (current->thread.mm->vm_area_head == NULL) {
        puts("Invalid vm area in page fault");
        return;
    } else {
        struct vm_area_struct *ptr;
        for (ptr = current->thread.mm->vm_area_head; ptr != NULL; ptr = ptr-
>vm_next) {
            if (ptr->vm_start <= stval && stval < ptr->vm_end) {
                found = ptr;
                break;
            }
        }
    }
    if (found == NULL) {
        puts("Invalid vm area in page fault\n");
        return;
    }
    if (scause == 12 && (found->vm_flags & PROT_READ) == 0 && (found->vm_flags
& PROT_EXEC) == 0 && (found->vm_flags & PROT_WRITE) == 0) {
```

```c
        puts("Invalid vm area in page fault\n");
        return;
    } else if (scause == 13 && (found->vm_flags & (PROT_READ)) == 0) {
        puts("Invalid vm area in page fault\n");
        return;
    } else if (scause == 15 && (found->vm_flags & PROT_READ) == 0 && (found->vm_flags & PROT_WRITE) == 0) {
        puts("Invalid vm area in page fault\n");
        return;
    }
    unsigned pa;
    unsigned long sz = (found->vm_end - found->vm_start + PAGE_SIZE - 1) / PAGE_SIZE * PAGE_SIZE;
    if (current->thread.mm->pa_for_stack != 0 && found->vm_end == USER_END) {
        pa = current->thread.mm->pa_for_stack;
        create_mapping(current->thread.mm->pgtbl, found->vm_start, current->thread.mm->pa_for_stack, sz, found->vm_page_prot.pgprot | (1 << 4));
    } else if (found->vm_start == 0) {
        pa = 0x84000000ul;
        create_mapping(current->thread.mm->pgtbl, found->vm_start, 0x84000000ul, sz, found->vm_page_prot.pgprot | (1 << 4));
    } else {
        pa = kmalloc(sz) - page_offset + 0x80000000ul;
        create_mapping(current->thread.mm->pgtbl, found->vm_start / PAGE_SIZE * PAGE_SIZE, pa, sz, found->vm_page_prot.pgprot | (1 << 4));
    }

    puts("[S] mapped PA: ");
    puti64(pa);
    puts(" to VA: ");
    puti64(found->vm_start);
    puts(" with size: ");
    puti64(sz);
    puts("\n");
}
```

注意需要特判fork子进程的栈页和进程的代码段，这时不需要用kmalloc来获取物理页，前者通过映射到记录的已复制的子进程物理页，后者映射到0x84000000

实现`fork`系统调用

本次fork是实验的一大难点，增加了很多实验手册中所没有说明的代码。

fork

```c
int fork() {
    ++task_num_top;
    task[task_num_top] = kmalloc(sizeof(struct task_struct));
    task[task_num_top]->state = TASK_RUNNING;
```

```c
    unsigned long sscratch_top = kmalloc(PAGE_SIZE) + PAGE_SIZE;
    task[task_num_top]->state = TASK_RUNNING;
    #ifdef SJF
    task[task_num_top]->counter = rand();
    #endif
    #ifdef PRIORITY
    task[task_num_top]->counter = 8 - task_num_top;
    #endif
    task[task_num_top]->priority = 5;
    task[task_num_top]->blocked = 0;
    task[task_num_top]->pid = task_num_top;
    task[task_num_top]->thread.sp = USER_END;

    puts("[PID = ");
    puti(task[task_num_top]->pid);
    puts("] Process fork from [PID = ");
    puti(current->pid);
    puts("] Successfully! counter = ");
    puti(task[task_num_top]->counter);
    puts("\n");

    // initial_kernel_stack(sscratch_top, sscratch_top - 280);
    task[task_num_top]->thread.sepc = current->thread.sepc;
    task[task_num_top]->thread.ra = (unsigned long)forkret;
    task[task_num_top]->thread.sscratch = sscratch_top;
    task[task_num_top]->thread.sp = sscratch_top - 280;
    task[task_num_top]->thread.mm = kmalloc(sizeof(struct mm_struct));
    task[task_num_top]->thread.mm->pgtbl = (unsigned long)kmalloc(PAGE_SIZE) -
page_offset + 0x80000000ul;
    task[task_num_top]->thread.mm->vm_area_head = NULL;
    task[task_num_top]->thread.mm->pa_for_stack = 0;
    unsigned long pgtbl_va = (unsigned long)task[task_num_top]->thread.mm-
>pgtbl;
    initial_pgtbl(pgtbl_va);

    if (current->thread.mm->vm_area_head != NULL) {
        struct vm_area_struct *tail = NULL;
        for (struct vm_area_struct *ptr = current->thread.mm->vm_area_head; ptr
!= NULL; ptr = ptr->vm_next) {
            struct vm_area_struct *t = kmalloc(sizeof(struct vm_area_struct));
            t->vm_start = ptr->vm_start;
            t->vm_end = ptr->vm_end;
            t->vm_flags = ptr->vm_flags;
            t->vm_page_prot = ptr->vm_page_prot;
            t->vm_mm = task[task_num_top]->thread.mm;
            t->vm_prev = tail;
            t->vm_next = NULL;
            if (tail == NULL) {
                task[task_num_top]->thread.mm->vm_area_head = t;
```

```c
            } else {
                tail->vm_next = t;
            }
            tail = t;
        }
    }

    task[task_num_top]->thread.stack = task[task_num_top]->thread.sp;

    for (int i = 0; i < 280 / 8; ++i) {
        task[task_num_top]->thread.stack[i] = current->thread.stack[i];
    }
    task[task_num_top]->thread.stack[9] = 0;
    char *new_stack = kmalloc(PAGE_SIZE);
    task[task_num_top]->thread.mm->pa_for_stack = (unsigned long)new_stack - 
page_offset + 0x80000000ul;
    for (int i = 0; i < PAGE_SIZE; ++i) {
        new_stack[i] = ((char*)(USER_END - PAGE_SIZE))[i];
    }

    return task_num_top;
}
```

这里的一大难点是，要在这里记录复制好的子进程的栈页表的物理地址。用于后续page fault的时候直接对复制好的物理地址做映射。

```c
void forkret() {
    asm volatile ("csrrw x0, sscratch, %0" : : "r"(current->thread.sscratch));
    ret_from_fork(current->thread.stack);
}
```

子进程退出handler_s时候的返回函数，主要为调用ret_from_fork

```
ret_from_fork:
    ld t0, 248(a0)
    csrw sepc, t0
    ld t0, 256(a0)
    csrw sstatus, t0
    ld t0, 264(a0)
    csrw stval, t0
    ld t0, 272(a0)
    csrw scause, t0
    ld x31, 240(a0)
    ld x30, 232(a0)
    ld x29, 224(a0)
    ld x28, 216(a0)
    ld x27, 208(a0)
    ld x26, 200(a0)
```

```
    ld x25, 192(a0)
    ld x24, 184(a0)
    ld x23, 176(a0)
    ld x22, 168(a0)
    ld x21, 160(a0)
    ld x20, 152(a0)
    ld x19, 144(a0)
    ld x18, 136(a0)
    ld x17, 128(a0)
    ld x16, 120(a0)
    ld x15, 112(a0)
    ld x14, 104(a0)
    ld x13, 96(a0)
    ld x12, 88(a0)
    ld x11, 80(a0)
    ld x9, 64(a0)
    ld x8, 56(a0)
    ld x7, 48(a0)
    ld x6, 40(a0)
    ld x5, 32(a0)
    ld x4, 24(a0)
    ld x3, 16(a0)
    ld x2, 8(a0)
    ld x1, 0(a0)
    ld x10, 72(a0)
    sret
```

这里从传入的stack参数中恢复寄存器。

编译及测试

遇到的问题

- create_mapping的传入地址是虚拟地址还是物理地址

  因为页表项的内容是物理地址，而我们kmalloc返回的地址均设为虚拟地址。这本来是很棘手的问题，但因为我们kmalloc返回的虚拟地址是内核地址，所以可以做简单的减法变换到物理变量。我通过在buddy.c中设置page_offset，可以将alloc_pages的返回值这个page_offset相减并加上0x80000000来获得物理地址。

- fork的时候会发生嵌套异常，这需要特别处理sscratch来判断是否发生嵌套异常。（该思路来自于linux的内核中关于嵌套处理的代码，由陈昱文同学提供思路）

运行结果

```
oslab@32e1b0aeeaf1:~/lab5$ make run-hello
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -initrd hello.bin
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
[S] Buddy allocate addr: FFFFFFE000040000
[S] Buddy allocate addr: FFFFFFE000080000
[S] Buddy allocate addr: FFFFFFE000038000
[S] Buddy allocate addr: FFFFFFE00003C000
[S] Buddy allocate addr: FFFFFFE000090000
[S] Buddy allocate addr: FFFFFFE000094000
[S] Buddy allocate addr: FFFFFFE000098000
[S] Buddy allocate addr: FFFFFFE00009C000
[S] Buddy allocate addr: FFFFFFE0000A0000
[S] Buddy allocate addr: FFFFFFE0000A4000
[S] Buddy allocate addr: FFFFFFE0000A8000
ZJU OS LAB              Guo Ziyang
task init ...
[S] Buddy allocate addr: FFFFFFE000035000
[S] Buddy allocate addr: FFFFFFE000036000
[S] Buddy allocate addr: FFFFFFE000037000
[S] Buddy allocate addr: FFFFFFE0000AC000
[S] Buddy allocate addr: FFFFFFE0000AD000
[S] Buddy allocate addr: FFFFFFE0000AE000
[S] Buddy allocate addr: FFFFFFE0000AF000
[S] Buddy allocate addr: FFFFFFE0000B0000
[S] Buddy allocate addr: FFFFFFE0000B1000
[S] Buddy allocate addr: FFFFFFE0000B2000
[S] Buddy allocate addr: FFFFFFE0000B3000
[S] Buddy allocate addr: FFFFFFE0000B4000
[S] Buddy allocate addr: FFFFFFE0000B5000
[S] Buddy allocate addr: FFFFFFE0000B6000
[S] Buddy allocate addr: FFFFFFE0000B7000
[ PID = 1] Process Create Successfully! counter = 1
[ PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 to task 1, prio: 5, counter: 1
[S] PAGE_FAULT: PID = 1
[S] PAGE_FAULT: scause: 12, sepc: 0x0000000000000000, badaddr: 0x0000000000000000
[S] Buddy allocate addr: FFFFFFE0000B8000
[S] Buddy allocate addr: FFFFFFE0000B9000
[S] mapped PA: 000084000000 to VA: 000084000000 with size: 000084001000
```

```
[S] Buddy allocate addr: FFFFFFE0000B5000
[S] Buddy allocate addr: FFFFFFE0000B6000
[S] Buddy allocate addr: FFFFFFE0000B7000
[ PID = 1] Process Create Successfully! counter = 1
[ PID = 0] Context Calculation: counter = 0
[!] Switch from task 0 to task 1, prio: 5, counter: 1
[S] PAGE_FAULT: PID = 1
[S] PAGE_FAULT: scause: 12, sepc: 0x0000000000000000, badaddr: 0x0000000000000000
[S] Buddy allocate addr: FFFFFFE0000B8000
[S] Buddy allocate addr: FFFFFFE0000B9000
[S] mapped PA: 000084000000 to VA: 000084000000 with size: 000084001000
[S] PAGE_FAULT: PID = 1
[S] PAGE_FAULT: scause: 15, sepc: 0x000084001070, badaddr: 0xFFFFFFDF7FFFFFF8
[S] Buddy allocate addr: FFFFFFE0000BA000
[S] Buddy allocate addr: FFFFFFE0000BB000
[S] Buddy allocate addr: FFFFFFE0000BC000
[S] mapped PA: 00800BA000 to VA: FFFFFFDF7FFFF000 with size: FFFFFFDF7FFF1000
[S] Buddy allocate addr: FFFFFFE0000BD000
[PID = 2] Process fork from [PID = 1] Successfully! counter = 4
[S] Buddy allocate addr: FFFFFFE0000BE000
[S] Buddy allocate addr: FFFFFFE0000BF000
[S] Buddy allocate addr: FFFFFFE0000C0000
[S] Buddy allocate addr: FFFFFFE0000C1000
[S] Buddy allocate addr: FFFFFFE0000C2000
[S] Buddy allocate addr: FFFFFFE0000C3000
[S] Buddy allocate addr: FFFFFFE0000C4000
[S] Buddy allocate addr: FFFFFFE0000C5000
[S] Buddy allocate addr: FFFFFFE0000C6000
[S] Buddy allocate addr: FFFFFFE0000C7000
[S] Buddy allocate addr: FFFFFFE0000C8000
[S] Buddy allocate addr: FFFFFFE0000C9000
[S] Buddy allocate addr: FFFFFFE0000CA000
[S] Buddy allocate addr: FFFFFFE0000CB000
[PID = 3] Process fork from [PID = 1] Successfully! counter = 5
[S] Buddy allocate addr: FFFFFFE0000CC000
[S] Buddy allocate addr: FFFFFFE0000CD000
[S] Buddy allocate addr: FFFFFFE0000CE000
[S] Buddy allocate addr: FFFFFFE0000CF000
[S] Buddy allocate addr: FFFFFFE0000D0000
[S] Buddy allocate addr: FFFFFFE0000D1000
[S] Buddy allocate addr: FFFFFFE0000D2000
[S] Buddy allocate addr: FFFFFFE0000D3000
[S] Buddy allocate addr: FFFFFFE0000D4000
[S] Buddy allocate addr: FFFFFFE0000D5000
[S] Buddy allocate addr: FFFFFFE0000D6000
```

实验心得

本次实验，调的时候真的超级超级超级自闭，create_mapping的bug卡了好久。最后也没找出是啥原因，重构了一遍代码后才解决bug。

好难好难好难，可能是因为工作量太大，我又拖到考试事周前才开始做，比较着急。

但是做下来真的学到了很多，感谢周亚金老师，感谢助教。

对实验指导的建议

fork提供的提示有些少，好多要自己摸索。可以提一下嵌套映射的问题，以及画一张图说明一下fork父进程和子进程调用时的栈信息。