lab2 重要函数总结:

entry.S

```
#include <mmu.h>
#include <memlayout.h> #define REALLOC(x) (x - KERNBASE) .text
.globl kern_entry
kern_entry:
    # REALLOC是因为内核在构建时被设置在了高位(kernel.ld中设置了内核起始虚地址0xC0100000,使得虚地址整体增加了KERNBASE)
    # 因此需要REALLOC来对内核全局变量进行重定位，在开启分页模式前保证程序访问的物理地址的正确性     # load pa of boot pgdir
    # 此时还没有开启页机制，__boot_pgdir(entry.S中的符号)需要通过REALLOC转换成正确的物理地址
    movl $REALLOC(__boot_pgdir), %eax
    # 设置eax的值到页表基址寄存器cr3中
    movl %eax, %cr3      # enable paging 开启页模式
    movl %cr0, %eax
    # 通过or运算，修改cr0中的值
    orl $(CR0_PE | CR0_PG | CR0_AM | CR0_WP | CR0_NE | CR0_TS | CR0_EM | CR0_MP), %eax
    andl $~(CR0_TS | CR0_EM), %eax
    # 将cr0修改完成后的值，重新送至cr0中(此时第0位PE位已经为1，页机制已经开启，当前页表地址为刚刚构造的__boot_pgdir)
    movl %eax, %cr0      # update eip
    # now, eip = 0x1..... next是处于高位地址空间的
    leal next, %eax
    # set eip = KERNBASE + 0x1.....
    # 通过jmp至next处，使得内核的指令指针指向了高位。但由于巧妙的设计了高位映射的内核页表，使得依然能准确访问之前低位虚空间下的所有内容
    jmp *%eax
next:      # unmap va 0 ~ 4M, it is temporary mapping
    xorl %eax, %eax
    # 将__boot_pgdir的第一个页目录项清零，取消0~4M虚地址的映射
    movl %eax, __boot_pgdir      # 设置C的内核栈
    # set ebp, esp
    movl $0x0, %ebp
    # the kernel stack region is from bootstack -- bootstacktop,
    # the kernel stack size is KSTACKSIZE (8KB)defined in memlayout.h
    movl $bootstacktop, %esp
    # now kernel stack is ready , call the first C function
    # 调用init.c中的kern_init总控函数
    call kern_init # should never get here
# 自旋死循环(如果内核实现正确，kern_init函数将永远不会返回并执行至此。因为操作系统内核本身就是通过自旋循环常驻内存的)
spin:
    jmp spin .data
.align PGSIZE
    .globl bootstack
bootstack:
    .space KSTACKSIZE
    .globl bootstacktop
```

```
bootstacktop: # kernel builtin pgdir
# an initial page directory (Page Directory Table, PDT)
# These page directory table and page table can be reused!
.section .data.pgdir
.align PGSIZE
__boot_pgdir:
.globl __boot_pgdir
    # map va 0 ~ 4M to pa 0 ~ 4M (temporary)
    # 80386的每一个一级页表项能够映射4MB连续的虚拟内存至物理内存的关系
    # 第一个有效页表项，当访问0~4M虚拟内存时，虚拟地址的高10位为0，即找到该一级页表项(页目录项)，进而可以找到二级页表__boot_pt1
    # 进而可以进行虚拟地址的0~4M -> 物理地址 0~4M的等价映射
    .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)
    # space用于将指定范围大小内的空间全部设置为0(等价于P位为0，即不存在的、无效的页表项)
    # KERNBASE/一个物理页的大小(PGSHIFT 4KB即偏移12位)/一个二级页表内的页表项(2^10个) * 4(一个页表项32位，即4byte)
    # 偏移的距离 - (. - __boot_pgdir) 是为了对齐
    .space (KERNBASE >> PGSHIFT >> 10 << 2) - (. - __boot_pgdir) # pad to PDE of KERNBASE
    # map va KERNBASE + (0 ~ 4M) to pa 0 ~ 4M
    # 第二个有效页表项，前面通过.space偏移跳过特定的距离，当虚拟地址为KERNBASE~KERNBASE+4M时，能够查找到该项
    # 其对应的二级页表同样是__boot_pt1，而其中映射的物理地址为按照下标顺序排列的0~4M，
    # 因此其最终的效果便能将KERNBASE~KERNBASE+4M的虚拟内存空间映射至物理内存空间的0~4M
    .long REALLOC(__boot_pt1) + (PTE_P | PTE_U | PTE_W)
    .space PGSIZE - (. - __boot_pgdir) # pad to PGSIZE .set i, 0
# __boot_pt1是一个存在1024个32位long数据的数组，当将其作为页表时其中每一项都代表着一个物理地址映射项
# i为下标，每个页表项的内容为i*1024作为映射的物理页面基址并加上一些低位的属性位(PTE_P代表存在，PTE_W代表可写)
__boot_pt1:
.rept 1024
    .long i * PGSIZE + (PTE_P | PTE_W)
    .set i, i + 1
.endr
```

pmm_init函数:

```
//pmm_init - setup a pmm to manage physical memory, build PDT&PT to setup paging mechanism
//         - check the correctness of pmm & paging mechanism, print PDT&PT
void
pmm_init(void) {
    // We've already enabled paging
    // 此时已经开启了页机制，由于boot_pgdir是内核页表地址的虚拟地址。通过PADDR宏转化为boot_cr3物理地址，供后续使用
    boot_cr3 = PADDR(boot_pgdir);     //We need to alloc/free the physical memory (granularity is 4KB or other size).
    //So a framework of physical memory manager (struct pmm_manager)is defined in pmm.h
    //First we should init a physical memory manager(pmm) based on the framework.
    //Then pmm can alloc/free the physical memory.
```

```
    //Now the first_fit/best_fit/worst_fit/buddy_system pmm are available.     //
初始化物理内存管理器
    init_pmm_manager();     // detect physical memory space, reserve already
used memory,
    // then use pmm->init_memmap to create free page list     // 探测物理内存空间,
初始化可用的物理内存
    page_init();     //use pmm->check to verify the correctness of the
alloc/free function in a pmm
    check_alloc_page();     check_pgdir();     static_assert(KERNBASE % PTSIZE
== 0 && KERNTOP % PTSIZE == 0);     // recursively insert boot_pgdir in itself
    // to form a virtual page table at virtual address VPT
    // 将当前内核页表的物理地址设置进对应的页目录项中(内核页表的自映射)
    boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;     // map all
physical memory to linear memory with base linear addr KERNBASE
    // linear_addr KERNBASE ~ KERNBASE + KMEMSIZE = phy_addr 0 ~ KMEMSIZE
    // 将内核所占用的物理内存，进行页表<->物理页的映射
    // 令处于高位虚拟内存空间的内核，正确的映射到低位的物理内存空间
    // (映射关系(虚实映射): 内核起始虚拟地址(KERNBASE)~内核截止虚拟地址
(KERNBASE+KMEMSIZE) =  内核起始物理地址(0)~内核截止物理地址(KMEMSIZE))
    boot_map_segment(boot_pgdir, KERNBASE, KMEMSIZE, 0, PTE_W);     // Since we
are using bootloader's GDT,
    // we should reload gdt (second time, the last time) to get user segments and
the TSS
    // map virtual_addr 0 ~ 4G = linear_addr 0 ~ 4G
    // then set kernel stack (ss:esp) in TSS, setup TSS in gdt, load TSS
    // 重新设置GDT
    gdt_init();     //now the basic virtual memory map(see memalyout.h) is
established.
    //check the correctness of the basic virtual memory map.
    check_boot_pgdir();     print_pgdir();
}
```

init_pmm_manager函数:

```
//init_pmm_manager - initialize a pmm_manager instance
static void
init_pmm_manager(void) {
    // pmm_manager默认指向default_pmm_manager 使用第一次适配算法
    pmm_manager = &default_pmm_manager;
    cprintf("memory management: %s\n", pmm_manager->name);
    pmm_manager->init();
}
```

pmm_manager定义:

```
// pmm_manager is a physical memory management class. A special pmm manager -
XXX_pmm_manager
// only needs to implement the methods in pmm_manager class, then XXX_pmm_manager
can be used
// by ucore to manage the total physical memory space.
```

```
struct pmm_manager {
    const char *name;                                 // XXX_pmm_manager's name
                                                      // 管理器的名称      void
(*init)(void);                                 // initialize internal
description&management data structure
                                                      // (free block list, number
of free block) of XXX_pmm_manager
                                                      // 初始化管理器      void
(*init_memmap)(struct Page *base, size_t n); // setup description&management data
structcure according to
                                                      // the initial free
physical memory space
                                                      // 设置可管理的内存,初始化可分
配的物理内存空间      struct Page *(*alloc_pages)(size_t n);             // allocate
>=n pages, depend on the allocation algorithm
                                                      // 分配>=N个连续物理页,返回分配
块首地址指针      void (*free_pages)(struct Page *base, size_t n);  // free >=n
pages with "base" addr of Page descriptor structures(memlayout.h)
                                                      // 释放包括自Base基址在内的，起
始的>=N个连续物理内存页      size_t (*nr_free_pages)(void);                    //
return the number of free pages
                                                      // 返回全局的空闲物理页数量
void (*check)(void);                              // check the correctness of
XXX_pmm_manager
};
```

Page结构

```
/* *
 * struct Page - Page descriptor structures. Each Page describes one
 * physical page. In kern/mm/pmm.h, you can find lots of useful functions
 * that convert Page to other data types, such as phyical address.
 * */
struct Page {
    // 当前物理页被虚拟页面引用的次数(共享内存时，影响物理页面的回收)
    int ref;                        // page frame's reference counter
    // 标志位集合(目前只用到了第0和第1个bit位) bit 0表示是否被保留（可否用于物理内存分配: 0
未保留，1被保留）;bit 1表示对于可分配的物理页，当前是否是已被分配的
    uint32_t flags;                 // array of flags that describe the status
of the page frame
    // 在不同分配算法中意义不同(first fit算法中表示当前空闲块中总共所包含的空闲页个数 ，只有
位于空闲块头部的Page结构才拥有该属性，否则为0)
    unsigned int property;          // the num of free block, used in first fit
pm manager
    // 空闲链表free_area_t的链表节点引用
    list_entry_t page_link;         // free list link
};
```

page_init函数:

```
/* pmm_init - initialize the physical memory management */
static void
page_init(void) {
    // 通过e820map结构体指针，关联上在bootasm.S中通过e820中断探测出的硬件内存布局
    // 之所以加上KERNBASE是因为指针寻址时使用的是线性虚拟地址。按照最终的虚实地址关系(0x8000 + KERNBASE)虚拟地址 = 0x8000 物理地址
    struct e820map *memmap = (struct e820map *)(0x8000 + KERNBASE);
    uint64_t maxpa = 0;     cprintf("e820map:\n");
    int i;
    // 遍历memmap中的每一项(共nr_map项)
    for (i = 0; i < memmap->nr_map; i ++) {
        // 获取到每一个布局entry的起始地址、截止地址
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        cprintf("  memory: %08llx, [%08llx, %08llx], type = %d.\n",
                memmap->map[i].size, begin, end - 1, memmap->map[i].type);
        // 如果是E820_ARM类型的内存空间块
        if (memmap->map[i].type == E820_ARM) {
            if (maxpa < end && begin < KMEMSIZE) {
                // 最大可用的物理内存地址 = 当前项的end截止地址
                maxpa = end;
            }
        }
    }     // 迭代每一项完毕后，发现maxpa超过了定义约束的最大可用物理内存空间
    if (maxpa > KMEMSIZE) {
        // maxpa = 定义约束的最大可用物理内存空间
        maxpa = KMEMSIZE;
    }     // 此处定义的全局end数组指针，正好是ucore kernel加载后定义的第二个全局变量(kern_init处第一行定义的)
    // 其上的高位内存空间并没有被使用,因此以end为起点，存放用于管理物理内存页面的数据结构
    extern char end[];     // 需要管理的物理页数 = 最大物理地址/物理页大小
    npage = maxpa / PGSIZE;
    // pages指针指向->可用于分配的，物理内存页面Page数组起始地址
    // 因此其恰好位于内核空间之上(通过ROUNDUP PGSIZE取整，保证其位于一个新的物理页中)
    pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);     for (i = 0; i < npage; i ++) {
        // 遍历每一个可用的物理页，默认标记为被保留无法使用
        SetPageReserved(pages + i);
    }     // 计算出存放物理内存页面管理的Page数组所占用的截止地址
    // freemem = pages(管理数据的起始地址) + (Page结构体的大小 * 需要管理的页面数量)
    uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
  // freemem之上的高位物理空间都是可以用于分配的free空闲内存
    for (i = 0; i < memmap->nr_map; i ++) {
        // 遍历探测出的内存布局memmap
        uint64_t begin = memmap->map[i].addr, end = begin + memmap->map[i].size;
        if (memmap->map[i].type == E820_ARM) {
            if (begin < freemem) {
                // 限制空闲地址的最小值
                begin = freemem;
            }
            if (end > KMEMSIZE) {
                // 限制空闲地址的最大值
                end = KMEMSIZE;
            }
```

```
            if (begin < end) {
                // begin起始地址以PGSIZE为单位，向高位取整
                begin = ROUNDUP(begin, PGSIZE);
                // end截止地址以PGSIZE为单位，向低位取整
                end = ROUNDDOWN(end, PGSIZE);
                if (begin < end) {
                    // 进行空闲内存块的映射，将其纳入物理内存管理器中管理，用于后续的物理内存分配
                    // 这里的begin、end都是探测出来的物理地址
                    // 第一个参数：起始Page结构的虚拟地址base = pa2page(begin)
                    // 第二个参数：空闲页的个数 = (end - begin) / PGSIZE
                    init_memmap(pa2page(begin), (end - begin) / PGSIZE);
                }
            }
        }
    }
}
```

lab3 重要函数结构总结：

vma_struct结构：

```
// the virtual continuous memory area(vma)
// 连续虚拟内存区域
struct vma_struct {
    // 关联的上层内存管理器
    struct mm_struct *vm_mm; // the set of vma using the same PDT
    // 描述的虚拟内存的起始地址
    uintptr_t vm_start;      //    start addr of vma
    // 描述的虚拟内存的截止地址
    uintptr_t vm_end;        // end addr of vma
    // 当前虚拟内存块的属性flags
    // bit0 VM_READ标识是否可读; bit1 VM_WRITE标识是否可写; bit2 VM_EXEC标识是否可执行
    uint32_t vm_flags;       // flags of vma
    // 连续虚拟内存块链表节点 (mm_struct->mmap_list)
    list_entry_t list_link;  // linear list link which sorted by start addr of vma
};

#define le2vma(le, member)                  \
    to_struct((le), struct vma_struct, member)

#define VM_READ                 0x00000001
#define VM_WRITE                0x00000002
#define VM_EXEC                 0x00000004
```

mm_struct结构：

```
// the control struct for a set of vma using the same PDT
struct mm_struct {
    // 连续虚拟内存块链表 (内部节点虚拟内存块的起始、截止地址必须全局有序，且不能出现重叠)
    list_entry_t mmap_list;        // linear list link which sorted by start
addr of vma
    // 当前访问的mmap_list链表中的vma块(由于局部性原理，之前访问过的vma有更大可能会在后续继续访问，该缓存可以减少从mmap_list中进行遍历查找的次数，提高效率)
    struct vma_struct *mmap_cache; // current accessed vma, used for speed
purpose
    // 当前mm_struct关联的一级页表的指针
    pde_t *pgdir;                  // the PDT of these vma
    // 当前mm_struct->mmap_list中vma块的数量
    int map_count;                 // the count of these vma
    // 用于虚拟内存置换算法的属性，使用void*指针做到通用 (lab中默认的swap_fifo替换算法中，将其做为了一个先进先出链表队列)
    void *sm_priv;                   // the private data for swap manager
};
```

alloc_pages函数：

```
//alloc_pages - call pmm->alloc_pages to allocate a continuous n*PAGESIZE memory

struct Page *
alloc_pages(size_t n) {
    struct Page *page=NULL;
    bool intr_flag;

    while (1)
    {
        // 关闭中断，避免分配内存时，物理内存管理器内部的数据结构变动时被中断打断，导致数据错误
        local_intr_save(intr_flag);
        {
            // 分配n个物理页
            page = pmm_manager->alloc_pages(n);
        }
        // 恢复中断控制位
        local_intr_restore(intr_flag);

        // 满足下面之中的一个条件，就跳出while循环
        // page != null 表示分配成功
        // 如果n > 1 说明不是发生缺页异常来申请的(否则n=1)
        // 如果swap_init_ok == 0 说明没有开启分页模式
        if (page != NULL || n > 1 || swap_init_ok == 0) break;

        extern struct mm_struct *check_mm_struct;
        //cprintf("page %x, call swap_out in alloc_pages %d\n",page, n);
        // 尝试着将某一物理页置换到swap磁盘交换扇区中，以腾出一个新的物理页来
        // 如果交换成功，则理论上下一次循环，pmm_manager->alloc_pages(1)将有机会分配空闲物理页成功
        swap_out(check_mm_struct, n, 0);
```

```
    }
    //cprintf("n %d,get page %x, No %d in alloc_pages\n",n,page,(page-pages));
    return page;
}
```

捕获页异常逻辑：

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

    int ret;

    switch (tf->tf_trapno) {
    case T_PGFLT:  //page fault
        // T_PGFLT 14号中断 页异常处理
        if ((ret = pgfault_handler(tf)) != 0) {
            // 页异常处理失败，打印栈帧
            print_trapframe(tf);
            panic("handle pgfault failed. %e\n", ret);
        }
        break;

    // 不完全。。。。。。
}

static int
pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        // 传入check_mm_struct是为了配合check_pgfault检查函数的
        // 在未来的实验中同一进程是共用一个mm_struct内存管理器，而截止lab3只存在一个进程：内
核进程
        // rcr2页异常发生时，cr2页故障线性地址寄存器，保存最后一次出现页故障的32位线性地址
        return do_pgfault(check_mm_struct, tf->tf_err, rcr2());
    }
    panic("unhandled page fault.\n");
}
```

从磁盘换入到主存swap_in：

```
int
swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
{
     // 分配一个新的物理页
     struct Page *result = alloc_page();
     assert(result!=NULL);

     // 获得线性地址addr对应的二级页表项指针
```

```
     pte_t *ptep = get_pte(mm->pgdir, addr, 0);
     // cprintf("SWAP: load ptep %x swap entry %d to vaddr 0x%08x, page %x, No
%d\n", ptep, (*ptep)>>8, addr, result, (result-pages));

     int r;
     // 将磁盘中读入的一整个物理页数据，写入result(此时的ptep二级页表项中存放的是
swap_entry_t结构的数据)
     if ((r = swapfs_read((*ptep), result)) != 0)
     {
        assert(r!=0);
     }
     cprintf("swap_in: load disk swap entry %d with swap_page in vadr 0x%x\n",
(*ptep)>>8, addr);
     // 令参数ptr_result指向已被换入内存中的result Page结构
     *ptr_result=result;
     return 0;
}

int
swapfs_read(swap_entry_t entry, struct Page *page) {
    // swap_offset宏右移8位，截取前24位 = swap_entry_t的offset属性
    // swap_entry_t的offset * PAGE_NSECT(物理页与磁盘扇区大小比值) = 要读取的起始扇区号

    // 从设备号指定的磁盘中，读取自某一扇区起始的N个连续扇区，并将其写入指定起始地址的内存空间
中
    // SWAP_DEV_NO参数指定设备号，swap_offset(entry) * PAGE_NSECT指定起始扇区号
    // page2kva(page)指定所要写入的目的页面虚地址起始空间，PAGE_NSECT指定了需要顺序连续读取
的扇区数量
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

从主存换出到磁盘swap_out:

```
/**
 * 参数mm，指定对应的内存管理器
 * 参数n，指定需要换出到swap扇区的物理页个数
 * 参数in_tick，可以用于发生时钟中断时，定时进行主动的换出操作，腾出更多的物理空闲页
 * */
int
swap_out(struct mm_struct *mm, int n, int in_tick)
{
     int i;
     for (i = 0; i != n; ++ i)
     {
          uintptr_t v;
          //struct Page **ptr_page=NULL;
          struct Page *page;
          // cprintf("i %d, SWAP: call swap_out_victim\n",i);
          // 由swap置换管理器，挑选出需要被牺牲的(被置换到swap磁盘扇区)的page，令page指针变
量指向其指针
          int r = sm->swap_out_victim(mm, &page, in_tick);
          if (r != 0) {
```

```
                // 挑选失败
                cprintf("i %d, swap_out: call swap_out_victim failed\n",i);
                break;
            }

            // 获得挑选出来的物理页的虚拟地址
            v=page->pra_vaddr;
            // 获得page->pra_vaddr线性地址对应的二级页表项
            pte_t *ptep = get_pte(mm->pgdir, v, 0);
            assert((*ptep & PTE_P) != 0);

            // 将其写入swap磁盘
            // page->pra_vaddr/PGSIZE = 虚拟地址对应的二级页表项索引(前20位)；
            // (page->pra_vaddr/PGSIZE) + 1 （+1为了在页表项中区别 0 和 swap 分区的映
射）
            // ((page->pra_vaddr/PGSIZE) + 1) << 8，为了构成swap_entry_t的高24位
            // 举个例子：
            // 假设page->pra_vaddr = 0x0000100,则page->pra_vaddr/PGSIZE = 0x00000001
            // page->pra_vaddr/PGSIZE + 1 = 0x00000002
            // 对应的swap_entry_t = 0x00000002 << 8 = 0x00000200,高24位为0x000002
            if (swapfs_write( (page->pra_vaddr/PGSIZE+1)<<8, page) != 0) {
                cprintf("SWAP: failed to save\n");
                // 当前物理页写入swap，交换失败。重新令其加入swap管理器中
                sm->map_swappable(mm, v, page, 0);
                continue;
            }
            else {
                // 交换成功
                cprintf("swap_out: i %d, store page in vaddr 0x%x to disk swap
entry %d\n", i, v, page->pra_vaddr/PGSIZE+1);
                // 设置ptep二级页表项的值
                *ptep = (page->pra_vaddr/PGSIZE+1)<<8;
                // 释放、归还page物理页
                free_page(page);
            }
            // 由于对应二级页表项出现了变化，刷新TLB快表
            tlb_invalidate(mm->pgdir, v);
        }
        return i;
}

int
swapfs_write(swap_entry_t entry, struct Page *page) {
    // swap_offset宏右移8位，截取前24位 = swap_entry_t的offset属性
    // swap_entry_t的offset * PAGE_NSECT(物理页与磁盘扇区大小比值) = 要写入的起始扇区号

    // 从设备号指定的磁盘中，从指定起始地址的内存空间开始，将数据写入自某一扇区起始的N个连续扇
区内
    // SWAP_DEV_NO参数指定设备号，swap_offset(entry) * PAGE_NSECT指定起始扇区号
    // page2kva(page)指定所要读入的源数据页面虚地址起始空间，PAGE_NSECT指定了需要顺序连续写
入的扇区数量
    return ide_write_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT,
page2kva(page), PAGE_NSECT);
}
```

lab4 重要函数结构总结：

proc_struct结构：

```
// process's state in his life cycle
// 进程状态
enum proc_state {
    // 未初始化
    PROC_UNINIT = 0,  // uninitialized
    // 休眠、阻塞状态
    PROC_SLEEPING,    // sleeping
    // 可运行、就绪状态
    PROC_RUNNABLE,    // runnable(maybe running)
    // 僵尸状态(几乎已经终止，等待父进程回收其所占资源)
    PROC_ZOMBIE,      // almost dead, and wait parent proc to reclaim his
resource
};

/**
 * 进程控制块结构（ucore进程和线程都使用proc_struct进行管理）
 * */
struct proc_struct {
    // 进程状态
    enum proc_state state;                      // Process state
    // 进程id
    int pid;                                    // Process ID
    // 被调度执行的总次数
    int runs;                                   // the running times of Proces
    // 当前进程内核栈地址
    uintptr_t kstack;                           // Process kernel stack
    // 是否需要被重新调度，以使当前线程让出CPU
    volatile bool need_resched;                 // bool value: need to be
rescheduled to release CPU?
    // 当前进程的父进程
    struct proc_struct *parent;                 // the parent process
    // 当前进程关联的内存总管理器
    struct mm_struct *mm;                       // Process's memory management
field
    // 切换进程时保存的上下文快照
    struct context context;                     // Switch here to run process
    // 切换进程时的当前中断栈帧
    struct trapframe *tf;                       // Trap frame for current
interrupt
    // 当前进程页表基地址寄存器cr3(指向当前进程的页表物理地址)
    uintptr_t cr3;                              // CR3 register: the base addr
of Page Directroy Table(PDT)
    // 当前进程的状态标志位
    uint32_t flags;                             // Process flag
    // 进程名
    char name[PROC_NAME_LEN + 1];               // Process name
    // 进程控制块链表节点
```

```
    list_entry_t list_link;                     // Process link list
    // 进程控制块哈希表节点
    list_entry_t hash_link;                     // Process hash list
};
```

proc_init函数:

```
// proc_init - set up the first kernel thread idleproc "idle" by itself and
//           - create the second kernel thread init_main
// 初始化第一个内核线程 idle线程、第二个内核线程 init_main线程
void
proc_init(void) {
    int i;

    // 初始化全局的线程控制块双向链表
    list_init(&proc_list);
    // 初始化全局的线程控制块hash表
    for (i = 0; i < HASH_LIST_SIZE; i ++) {
        list_init(hash_list + i);
    }

    // 分配idle线程结构
    if ((idleproc = alloc_proc()) == NULL) {
        panic("cannot alloc idleproc.\n");
    }

    // 为idle线程进行初始化
    idleproc->pid = 0; // idle线程pid作为第一个内核线程，其不会被销毁，pid为0
    idleproc->state = PROC_RUNNABLE; // idle线程被初始化时是就绪状态的
    idleproc->kstack = (uintptr_t)bootstack; // idle线程是第一个线程，其内核栈指向
bootstack
    idleproc->need_resched = 1; // idle线程被初始化后，需要马上被调度
    // 设置idle线程的名称
    set_proc_name(idleproc, "idle");
    nr_process ++;

    // current当前执行线程指向idleproc
    current = idleproc;

    // 初始化第二个内核线程initproc， 用于执行init_main函数，参数为"Hello world!!"
    int pid = kernel_thread(init_main, "Hello world!!", 0);
    if (pid <= 0) {
        // 创建init_main线程失败
        panic("create init_main failed.\n");
    }

    // 获得initproc线程控制块
    initproc = find_proc(pid);
    // 设置initproc线程的名称
    set_proc_name(initproc, "init");

    assert(idleproc != NULL && idleproc->pid == 0);
    assert(initproc != NULL && initproc->pid == 1);
```

```
}
```

kern_init函数：

```
// kernel_thread - create a kernel thread using "fn" function
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
// 创建一个内核线程，并执行参数fn函数，arg作为fn的参数
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    // 构建一个临时的中断栈帧tf，用于do_fork中的copy_thread函数(因为线程的创建和切换是需要利用CPU中断返回机制的)
    memset(&tf, 0, sizeof(struct trapframe));
    // 设置tf的值
    tf.tf_cs = KERNEL_CS; // 内核线程，设置中断栈帧中的代码段寄存器CS指向内核代码段
    tf.tf_ds = tf.tf_es = tf.tf_ss = KERNEL_DS; // 内核线程，设置中断栈帧中的数据段寄存器指向内核数据段
    tf.tf_regs.reg_ebx = (uint32_t)fn; // 设置中断栈帧中的ebx指向fn的地址
    tf.tf_regs.reg_edx = (uint32_t)arg; // 设置中断栈帧中的edx指向arg的起始地址
    tf.tf_eip = (uint32_t)kernel_thread_entry; // 设置tf.eip指向kernel_thread_entry这一统一的初始化的内核线程入口地址
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

do_fork函数：

```
/* 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

    // 分配一个未初始化的线程控制块
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    // 其父进程属于current当前进程
    proc->parent = current;

    // 设置，分配新线程的内核栈
    if (setup_kstack(proc) != 0) {
        // 分配失败，回滚释放之前所分配的内存
        goto bad_fork_cleanup_proc;
    }
    // 由于是fork，因此fork的一瞬间父子线程的内存空间是一致的（clone_flags决定是否采用写时复
制）
    if (copy_mm(clone_flags, proc) != 0) {
        // 分配失败，回滚释放之前所分配的内存
        goto bad_fork_cleanup_kstack;
    }
    // 复制proc线程时，设置proc的上下文信息
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        // 生成并设置新的pid
        proc->pid = get_pid();
        // 加入全局线程控制块哈希表
        hash_proc(proc);
        // 加入全局线程控制块双向链表
        list_add(&proc_list, &(proc->list_link));
        nr_process ++;
    }
```

```
    local_intr_restore(intr_flag);
    // 唤醒proc，令其处于就绪态PROC_RUNNABLE
    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;
}
```

copy_thread函数：

```
// copy_thread - setup the trapframe on the  process's kernel stack top and
//             - setup the kernel entry point and stack of process
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    // 令proc-tf 指向proc内核栈顶向下偏移一个struct trapframe大小的位置
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    // 将参数tf中的结构体数据复制填入上述proc->tf指向的位置(正好是上面struct trapframe指针-1腾出来的那部分空间)
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    // 令proc上下文中的eip指向forkret,切换恢复上下文后，新线程proc便会跳转至forkret
    proc->context.eip = (uintptr_t)forkret;
    // 令proc上下文中的esp指向proc->tf，指向中断返回时的中断栈帧
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

cpu_idle函数：

```
// cpu_idle - at the end of kern_init, the first kernel thread idleproc will do below works
void
cpu_idle(void) {
    while (1) {
        // idle线程执行逻辑就是不断的自旋循环，当发现存在有其它线程可以被调度时
        // idle线程，即current.need_resched会被设置为真，之后便进行一次schedule线程调度
        if (current->need_resched) {
            schedule();
        }
    }
}
```

schedule函数中，会先关闭中断，避免调度的过程中被中断再度打断而出现并发问题。然后从ucore的就绪线程队列中，按照某种调度算法选择出下一个需要获得CPU的就绪线程。

通过proc_run函数，令就绪线程的状态从就绪态转变为运行态，并切换线程的上下文，保存current线程(例如：idle_proc)的上下文，并在CPU上恢复新调度线程(例如：init_proc)的上下文。

schedule函数：

```
/**
 * 进行CPU调度
 * */
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    // 暂时关闭中断，避免被中断打断，引起并发问题
    local_intr_save(intr_flag);
    {
        // 令current线程处于不需要调度的状态
        current->need_resched = 0;
        // lab4中暂时没有更多的线程，没有引入线程调度框架，而是直接先进先出的获取init_main线程进行调度
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                // 找到一个处于PROC_RUNNABLE就绪态的线程
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            // 没有找到，则next指向idleproc线程
            next = idleproc;
        }
        // 找到的需要被调度的next线程runs自增
        next->runs ++;
        if (next != current) {
            // next与current进行上下文切换，令next获得CPU资源
            proc_run(next);
        }
    }
    // 恢复中断
    local_intr_restore(intr_flag);
}
```

proc_run函数：

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
// 进行线程调度，令当前占有CPU的让出CPU，并令参数proc指向的线程获得CPU控制权
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // 只有当proc不是当前执行的线程时，才需要执行
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;

        // 切换时新线程任务时需要暂时关闭中断，避免出现嵌套中断
        local_intr_save(intr_flag);
        {
            current = proc;
            // 设置TSS任务状态段的esp0的值，令其指向新线程的栈顶
            // ucore参考Linux的实现，不使用80386提供的TSS任务状态段这一硬件机制实现任务上下文切换，ucore在启动时初始化TSS后(init_gdt)，便不再对其进行修改。
            // 但进行中断等操作时，依然会用到当前TSS内的esp0属性。发生用户态到内核态中断切换时，硬件会将中断栈帧压入TSS.esp0指向的内核栈中
            // 因此ucore中的每个线程，需要有自己的内核栈，在进行线程调度切换时，也需要及时的修改esp0的值，使之指向新线程的内核栈顶。
            load_esp0(next->kstack + KSTACKSIZE);
            // 设置cr3寄存器的值，令其指向新线程的页表
            lcr3(next->cr3);
            // switch_to用于完整的进程上下文切换，定义在统一目录下的switch.S中
            // 由于涉及到大量的寄存器的存取操作，因此使用汇编实现
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

struct context：

```
// Saved registers for kernel context switches.
// Don't need to save all the %fs etc. segment registers,
// because they are constant across kernel contexts.
// Save all the regular registers so we don't need to care
// which are caller save, but not the return register %eax.
// (Not saving %eax just simplifies the switching code.)
// The layout of context must match code in switch.S.
// 当进程切换时保存的当前寄存器上下文
struct context {
    uint32_t eip;
    uint32_t esp;
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
```

```
    uint32_t edi;
    uint32_t ebp;
};
```

switch_to函数定义：

```
void switch_to(struct context *from, struct context *to);
```

switch_to实现：

```
.text
.globl switch_to
switch_to:                      # switch_to(from, to)

    # save from registers
    # 令eax保存第一个参数from(context)的地址
    movl 4(%esp), %eax          # eax points to from
    # from.context 保存eip、esp等等寄存器的当前快照值
    popl 0(%eax)                # save eip !popl
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)

    # restore to registers
    # 令eax保存第二个参数next(context)的地址,因为之前popl了一次，所以4(%esp)目前指向第二个
参数
    movl 4(%esp), %eax          # not 8(%esp): popped return address already
                                # eax now points to to
    # 恢复next.context中的各个寄存器的值
    movl 28(%eax), %ebp
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp
    pushl 0(%eax)               # push eip

    # ret时栈上的eip为next(context)中设置的值(fork时，eip指向 forkret,esp指向分配好的
trap_frame)
    ret
```