ucore 实验报告

> git repo：https://github.com/kurodes/ucore_lab

Lab 1 系统软件启动过程

练习1：理解通过make生成执行文件的过程

问题1：

操作系统镜像文件ucore.img是如何一步一步生成的?

- 编译得到内核程序 `bin/kernel`
- 生成 `bin/bootblock` 引导程序
  - 通过编译链接，生成 `obj/bootblock.o`，`sign.o`
  - 生成 `bin/bootblock` 引导扇区
- 使用 `dd` 生成 `ucore.img` 虚拟磁盘
  - 初始化 `ucore.img` 为内容为0的文件
  - 拷贝 `bin/bootblock` 到 `ucore.img` 第一个扇区
  - 拷贝 `bin/kernel` 到 `ucore.img` 第二个扇区往后的空间

问题2：

一个被系统认为是符合规范的硬盘主引导扇区的特征是什么?

- `sign.c` 处理 `bootblock.o` 成一个符合规范的引导扇区
  - 大小 `512 byte`，用0填充
  - 文件内容 <= `510 bytes`
  - 最后 `2 bytes` 为 `0x55 0xAA`

练习2：使用qemu执行并调试lab1中的软件

- `make lab1-mon`
- `make debug`

练习3：分析bootloader进入保护模式的过程

问题1：

为何开启A20，以及如何开启A20

- `A20` 控制了 `CPU` 能够访问的内存空间大小
- 在实模式下, 我们只希望 `CPU` 访问 `1MB` 以内的内存空间
- 在实模式下，将第 `20bit` 的地址线限制为 `0`，这样 `CPU` 就不能访问超过 `1MB` 的内存空间
- 进入保护模式后，再解除对 `A20` 地址线的限制，这样 `CPU` 就能访问超过 `1MB` 的内存空间
- 默认情况下，`A20` 地址线是关闭的，因此在进入保护模式前，我们需要开启 `A20` 地址线

问题2：

如何初始化GDT表

- 保护模式的内存管理模式是：纯段模式和段页式；段模式是必不可少的
- 段描述符：[Base Address, Limit, Access]，`64bit`
  - 段模式通过Base Address, Limit寻址
  - 保护模式也为段模式提供了保护机制，因此需要通过段描述符来规定访问权限
- Intel 为了保持向后兼容，将段寄存器仍然规定为`16bit`，无法直接引用`64bit`的段描述符；
  - 进而将段描述符放入全局数组`GDT`中，通过下标索引来间接引用（段寄存器中的高`13bit`的内容为索引）；
  - 寄存器`GDTR`用来存放`GDT`的入口地址
  - 程序员通过 `LGDT` 指令将入口地址寄存器`GDTR`
- 实模式下, 逻辑地址=段选择子`16bit`+偏移量`32bit`；对应线性地址= (段选择子=>GDT=>段Base Address) + 偏移量

问题3：

如何使能和进入保护模式

- 开启A20
- 初始化gdt后
- 将控制寄存器`CR0`的`PE`位，置为`1`即可。

bootloader进入保护模式的过程

- bootloader开始运行在实模式，物理地址为0x7c00,且是16位模式
- bootloader关闭所有中断，方向标志位复位，ds，es，ss段寄存器清零
- 打开A20使之能够使用高位地址线
- 由实模式进入保护模式，使用lgdt指令把GDT描述符表的大小和起始地址存入gdt寄存器，修改寄存器CR0的最低位完成从实模式到保护模式的转换
- 使用ljmp指令跳转到32位指令模式
- 进入保护模式后，设置ds，es，fs，gs，ss段寄存器，堆栈指针，便可以进入c程序bootmain

练习4：分析bootloader加载ELF格式的OS的过程

问题1：

bootloader如何读取硬盘扇区的?

- bootmain中readsect函数完成读取磁盘扇区的工作
- 调用waitdisk函数等待磁盘准备好
- 调用insl函数把磁盘扇区数据读到指定内存

问题2：

bootloader是如何加载ELF格式的OS?

- 调用readseg函数从kernel头读取8个扇区得到elfher
- 通过elfher判断是符合格式的ELF文件，循环调用readseg函数加载每一个程序段
- 调用elfher的入口指针进入OS

练习5：实现函数调用堆栈跟踪函数

- `ebp`为基址指针寄存器，该寄存器中存储着栈中的一个地址（原`ebp`入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的`ebp`值

- `ss:[ebp+4]`处为返回地址，`ss:[ebp+8]`处为第一个参数值（最后一个入栈的参数值，此处假设其占用4字节内存），`ss:[ebp-4]`处为第一个局部变量，`ss:[ebp]`处为上一层ebp值
- `esp`为堆栈指针寄存器(指向栈顶)
- `ebp`为0时表明程序返回到了最开始初始化的函数，`ebp=0`为循环的退出条件

```
void print_stackframe(void){
    uint32_t ebp = read_ebp(), eip = read_eip();
    int i, j;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i ++) {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        // ebp向上移动4个字节为eip
        uint32_t *args = (uint32_t *)ebp + 2;
        // 再向上每4个字节都为输入的参数(这里只是假设4个参数，做实验)
        for (j = 0; j < 4; j ++) {
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);
        // ebp指针指向的位置向上一个地址为上一个函数的eip
        eip = ((uint32_t *)ebp)[1];
        // ebp指针指向的位置存储的上一个ebp的地址
        ebp = ((uint32_t *)ebp)[0];
    }
}
```

练习6：完善中断初始化和处理

问题1：

中断描述符表（也可简称为保护模式下的中断向量表）中一个表项占多少字节？其中哪几位代表中断处理代码的入口?

- 中断的类型
  - 外部中断/异步中断：由外部设备引起的外部事件，如I/O中断、时钟中断、控制台中断等
  - 内部中断/同步中断：在CPU执行指令期间检测到不正常的或非法的条件(如除零错、地址访问越界)所引起的内部事件
  - 陷入中断/软中断/系统调用：在程序中使用请求系统服务的系统调用而引发的事件
  - 中断会清空`IF`标志，不允许被打断
- 中断描述符表（IDT）
  - 当`CPU`收到中断时，会查找对应的`IDT`，确定对应的中断服务例程
  - `IDT`是一个8字节的描述符数组，可以位于内存的任意位置
  - CPU通过`IDT寄存器（IDTR）`的内容来寻址`IDT`的起始地址
  - 指令`LIDT`和`SIDT`用来操作`IDTR`
  - `IDT`的一个表项内容有：段选择子、offset、其他标志位；`64bit`

问题2：

请编程完善kern/trap/trap.c中对中断向量表进行初始化的函数idt_init。

- `SETGATE`生成一个中断描述表项
  - `gate`为中断描述符表项对应的数据结构
  - `istrap`标识是中断还是系统调用；区别在于，中断会清空`IF`标志，不允许被打断
  - `sel`为中断服务例程的代码段选择子

    - `off`为中断服务例程的偏移量
    - `dpl`为访问权限
- 中断号
    - 保护模式下有`256个中断号`
    - `0~31`是保留的, 用于处理异常和`NMI`(不可屏蔽中断)
    - `32~255`由用户定义, 可以是设备中断或系统调用
- 实现`idt_init`
    - 使用SETGATE宏设置每一个idt，为中断门描述符
    - 权限均为内核态权限，设置T_SYSCALL
    - 使用陷阱门描述符，权限为用户权限
    - 最后调用lidt函数

```
void idt_init(void){
    extern uintptr_t __vectors[];
    int i;
    for(i = 0 ; i < 256 ; i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    lidt(&idt_pd);
}
```

问题3：

请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”。

- 找到时钟中断 IRQ_TIMER 位置

```
case IRQ_OFFSET + IRQ_TIMER:
    ticks ++;
    if (ticks % TICK_NUM == 0) {
        print_ticks();
    }
    break;
```

Lab 2 物理内存管理

> 实现物理内存管理系统
>
> 物理内存管理中的连续空间分配算法
>
> 建立二级页表管理物理内存

练习1：实现 first-fit 连续物理内存分配算法

设计实现

- first-fit
    - 空闲页块链表，按照空闲页块起始地址来排序
    - first-fit 从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后按照作业的大小，从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中

```
struct Page {
    int ref;                        // page frame's reference counter
    uint32_t flags;                 // array of flags that describe the status
of the page frame
    unsigned int property;          // the num of free block, used in first fit
pm manager
    list_entry_t page_link;         // free list link
};
```

- 物理页四个成员变量
  - `ref` 表示物理页被页表的引用记数，即映射此物理页的虚拟页个数。一旦某页表中有一个页表项设置了虚拟页到这个 `Page` 管理的物理页的映射关系，就会把 `Page` 的 `ref` 加一。反之，若是解除，那就减一
  - `flags` 表示此物理页的状态标记，有两个标志位
    - 第一个表示是否被保留，如果被保留了则设为 `1`（比如内核代码占用的空间）。
    - 第二个表示此页是否是 `free` 的。如果设置为 `1` ，表示这页是 `free` 的，可以被分配；如果设置为 `0` ，表示这页已经被分配出去了，不能被再二次分配。
  - `property` 用来记录某连续内存空闲块的大小，这里需要注意的是用到此成员变量的这个 `Page` 一定是连续内存块的开始地址（第一页的地址）
  - `page_link` 把多个连续内存空闲块链接在一起的双向链表指针；连续内存空闲块利用这个页的成员变量 `page_link` 来链接比它地址小和大的其他连续内存空闲块。

```
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // # of free pages in this free list
} free_area_t;
```

- 空闲页块链表
  - free_list 是一个 list_entry 结构的双向链表指针
  - nr_free 记录当前空闲页的个数

1. `default_init` 初始化空闲页块链表

```
static void default_init(void) {
    list_init(&free_list);  //初始化链表
    nr_free = 0; //空闲页块一开始是0个
}
```

2. `default_init_memmap` 初始化空闲页块, 将全部的可分配物理页视为一大块空闲块加入空闲表

```
static void default_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));//确认本页是否为保留页
        //设置标志位
        p->flags = p->property = 0;
        set_page_ref(p, 0);//清空引用

    }
    base->property = n; //连续内存空闲块的大小为n，属于物理页管理链表，头一个空闲页块 要设置数量
```

```
    SetPageProperty(base);
    nr_free += n;  //说明连续有n个空闲块，属于空闲链表
    list_add_before(&free_list, &(p->page_link));//插入空闲页的链表里面，初始化完每个空闲页后，将其要插入到链表每次都插入到节点前面，因为是按地址排序
}
```

3. `default_alloc_memmap` 从空闲页块的链表中去遍历，找到第一块大小大于 `n` 的块，然后分配出来，把它从空闲页链表中除去；如有剩余，把分完剩下的部分再次加入会空闲页链表中。
    - 保证空闲表中空闲空间地址递增：分配的page块有剩余空间时，把剩余空闲块节点插入到当前节点的前一个节点的后面

```
if (page->property > n) {
    struct Page *p = page + n;
    p->property = page->property - n;
    // 将page的property改为n
    page->property = n;
    // 在le的前一个节点后面插入
    list_add(list_prev(le), &(p->page_link));
}
```

4. `default_free_pages` 将需要释放的空间标记为空之后，需要找到空闲表中合适的位置。邻接的空间合并。

```
static void
default_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;//修改标志位
        set_page_ref(p, 0);
    }
    base->property = n;//设置连续大小为n
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    // 合并到合适的页块中
    while (le != &free_list) {
        p = le2page(le, page_link);//获取链表对应的Page
        le = list_next(le);
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    le = list_next(&free_list);
    // 将合并好的合适的页块添加回空闲页块链表
    while (le != &free_list) {
```

```
            p = le2page(le, page_link);
            if (base + base->property <= p) {
                break;
            }
            le = list_next(le);
        }
        list_add_before(le, &(base->page_link));//将每一空闲块对应的链表插入空闲链表中
    }
```

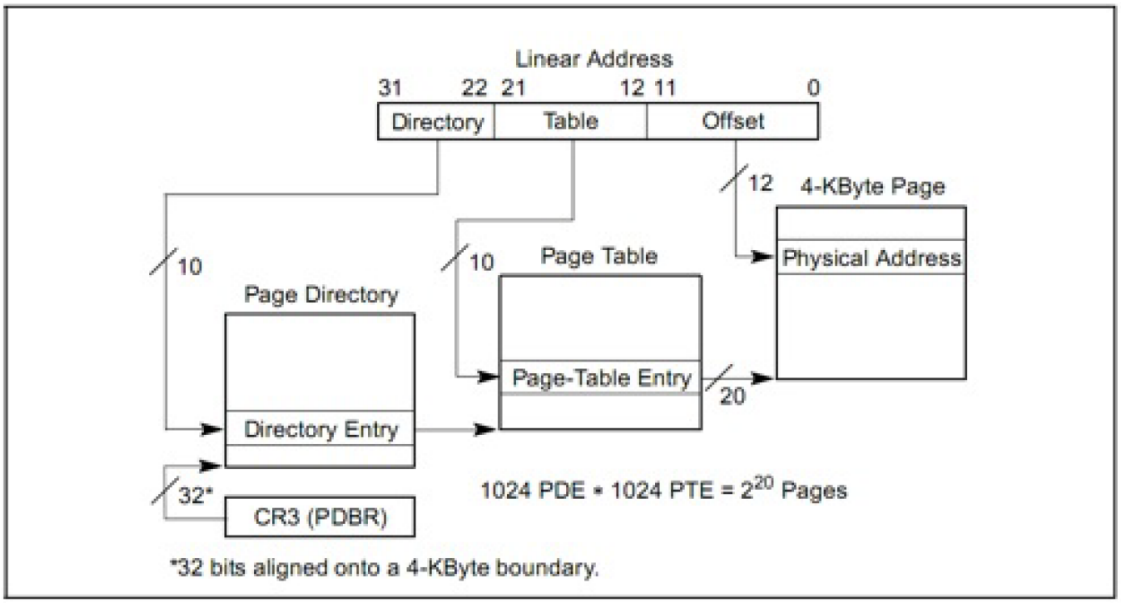
问题1：

你的first fit算法是否有进一步的改进空间

- 在上面的 `First Fit` 算法中，需要进行链表查找和有序链表插入。对于有序链表插入，在特殊情况下是可以优化的。就一个刚被释放的内存块来说，如果它的邻接空间都是空闲的，那么就不需要进行线性时间复杂度的链表插入操作，而是直接并入邻接空间，时间复杂度为常数。为了判断邻接空间是否为空闲状态，空闲块的信息除了保存在第一个页面之外，还需要在最后一页保存信息，这样新的空闲块只需要检查邻接的两个页面就能判断邻接空间块的状态。

练习2：实现寻找虚拟地址对应的页表项

- ucore 物理内存的页式管理通过一个二级的页表实现



  - 一个页表项管理一个物理页 4KB
  - 页目录表占一页
  - 根据LAZY，二级页表等到需要的时候再添加。

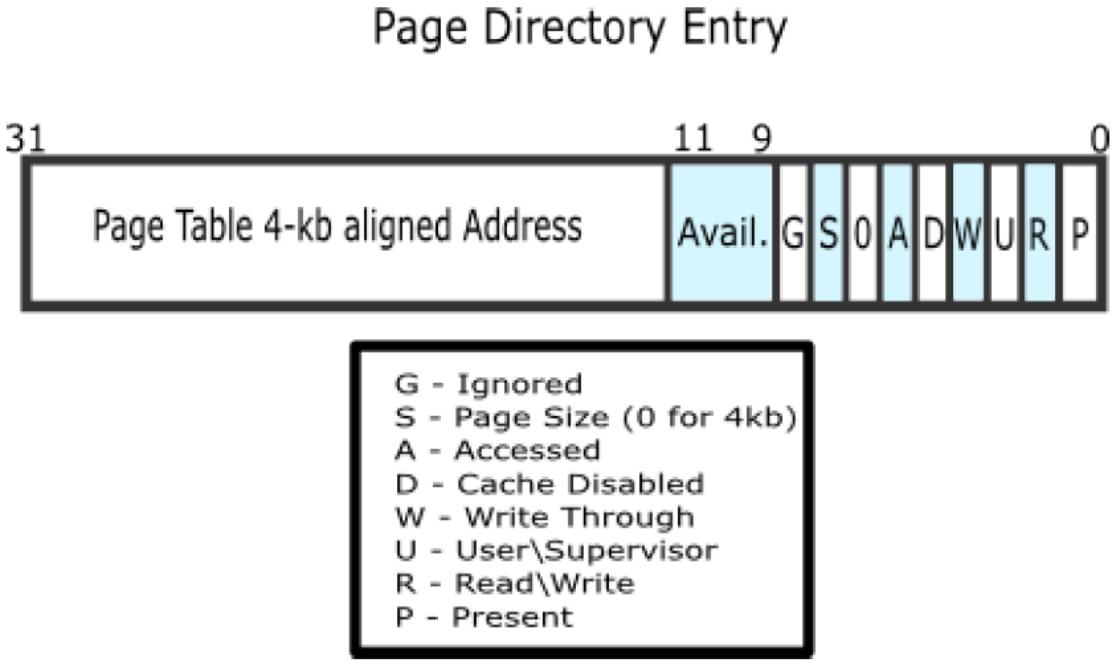
设计实现 `get_pte`

- 提供一个虚拟地址，然后根据这个虚拟地址的高 10 位，找到页目录表中的 PDE 项。前20位是页表项 (二级页表) 的线性地址，后 12 位为属性，然后判断一下 PTE 是否存在(就是判断 P 位)。不存在，则获取一个物理页，然后将这个物理页的线性地址写入到 PTE 中，最后返回 PTE 项。简而言之就是根据所给的虚拟地址，构造一个 PTE 项。

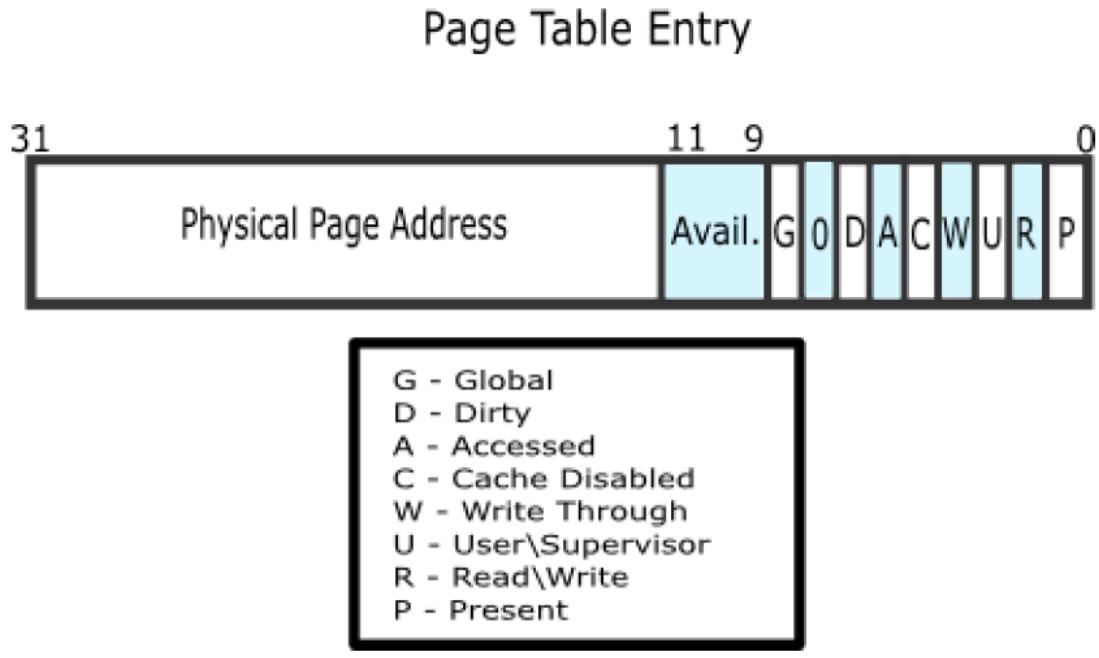
问题1：

请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中每个组成部分的含义以及对ucore而言的潜在用处

- Page Directory Entry



  - *P* (Present) 位：表示该页保存在物理内存中。
  - *R* (Read/Write) 位：表示该页可读可写。
  - *U* (User) 位：表示该页可以被任何权限用户访问。
  - *W* (Write Through) 位：表示 CPU 可以直写回内存。
  - *D* (Cache Disable) 位：表示不需要被 CPU 缓存。
  - *A* (Access) 位：表示该页被写过。
  - *S* (Size) 位：表示一个页 4MB 。
  - 9-11 位保留给 OS 使用。
  - 12-31 位指明 PTE 基质地址。
- Page Table Entry



  - 0-3 位同 PDE。
  - *C* (Cache Disable) 位：同 PDE *D* 位。
  - *A* (Access) 位：同 PDE 。
  - *D* (Dirty) 位：表示该页被写过。
  - *G* (Global) 位：表示在 CR3 寄存器更新时无需刷新 TLB 中关于该页的地址。
  - 9-11 位保留给 OS 使用。

- 12-31 位指明物理页基址。

问题2：

如果ucore执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

- 进行换页操作。
  - 首先 CPU 将产生页访问异常的线性地址放到 cr2 寄存器中
  - 和普通的中断一样，将寄存器的值压入栈中，设置错误代码 error_code，触发 Page Fault 异常
  - 进入 error_code 中断服务例程，将外存的数据换到内存中来
  - 最后退出中断，回到进入中断前的状态

练习3：释放某虚地址所在的页并取消对应二级页表项的映射

设计实现

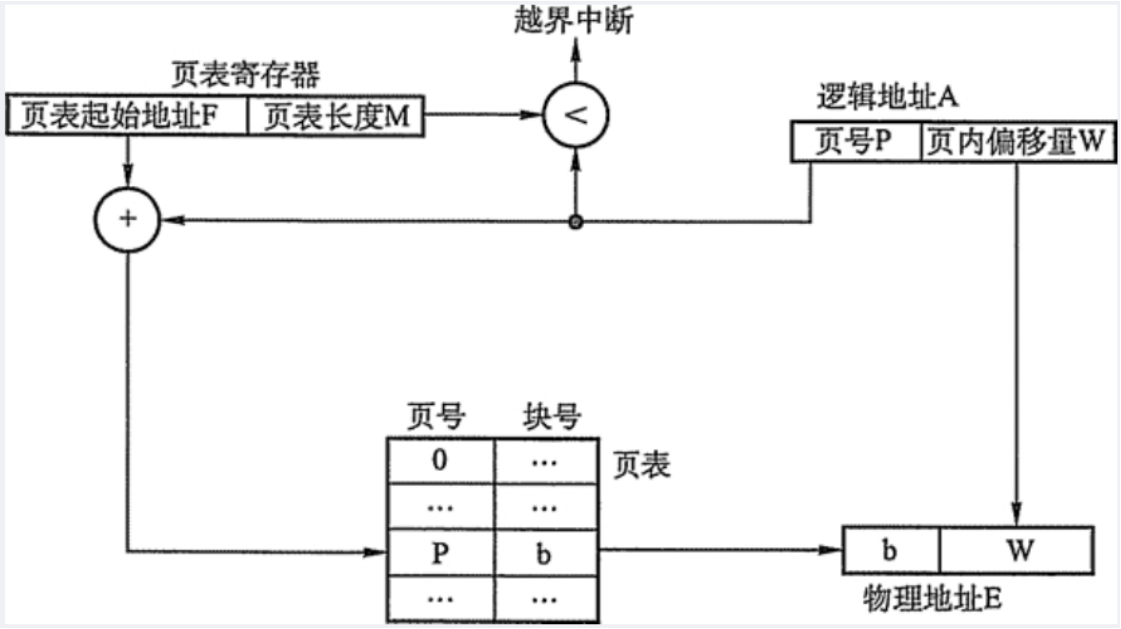
- 通过 `pte2page(**ptep*)` 获取相应页
- 将物理页的引用数目减一，如果变为零，那么释放页面；
  - 将页目录项清零；
  - 刷新TLB。

```
page_remove_pte(pde_t *pgdir, uintptr_t la, pte_t *ptep) {
    if ((*ptep & PTE_P)) { //判断页表中该表项是否存在
        struct Page *page = pte2page(*ptep);// 将页表项转换为页数据结构
        if (page_ref_dec(page) == 0) { // 判断是否只被引用了一次，若引用计数减一后为0，则释放该物理页
            free_page(page);
        }
        *ptep = 0; // //如果被多次引用，则不能释放此页，只用释放二级页表的表项，清空 PTE
        tlb_invalidate(pgdir, la); // 刷新 tlb
    }
}
```

问题1：

数据结构Page的全局变量（其实是一个数组）的每一项与页表中的页目录项和页表项有无对应关系?如果有，其对应关系是啥?

- 有



- pages 为 page 组成的数组
- PPN 为线性地址的高20位。PPN = PDX + PTX
- page即是 pages 全局数组中以 Page Directory Index 和 Page Table Index 的组合 PPN (Physical Page Number) 为索引的那一项；`&pages[PPN] = page`

问题2：

如果希望虚拟地址与物理地址相等，则需要如何修改lab2，完成此事?

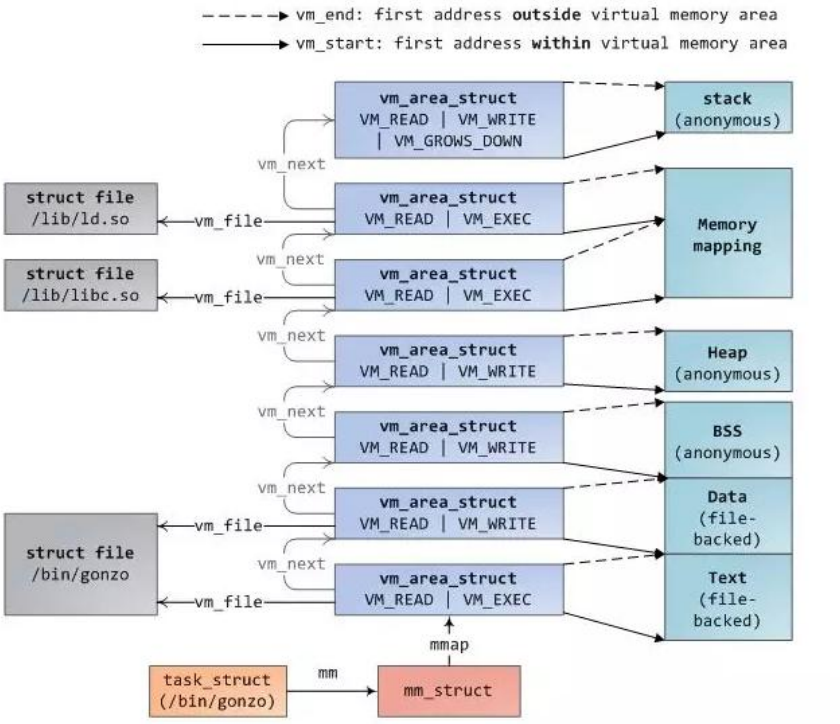
- 修改链接脚本，将内核起始虚拟地址修改为`0x100000`
- 修改虚拟内存空间起始地址为0
- 注释掉取消0~4M区域内存页映射的代码

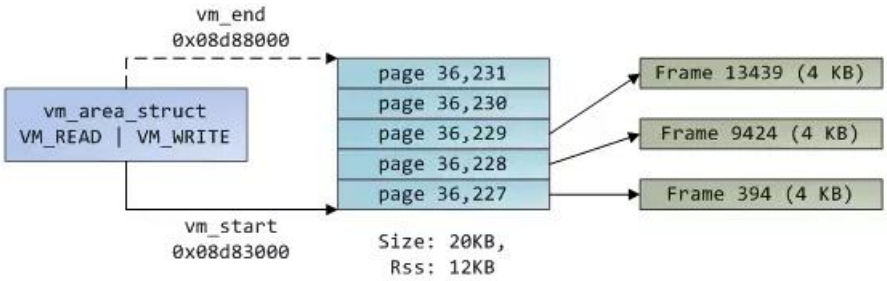
Lab 3 虚拟内存管理

> 借助于页表机制和实验一中涉及的中断异常处理机制，完成Page Fault异常处理和FIFO页替换算法的实现。
>
> 实验原理最大的区别是在设计了如何在磁盘上缓存内存页，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。

练习1：给未被映射的地址映射上物理页

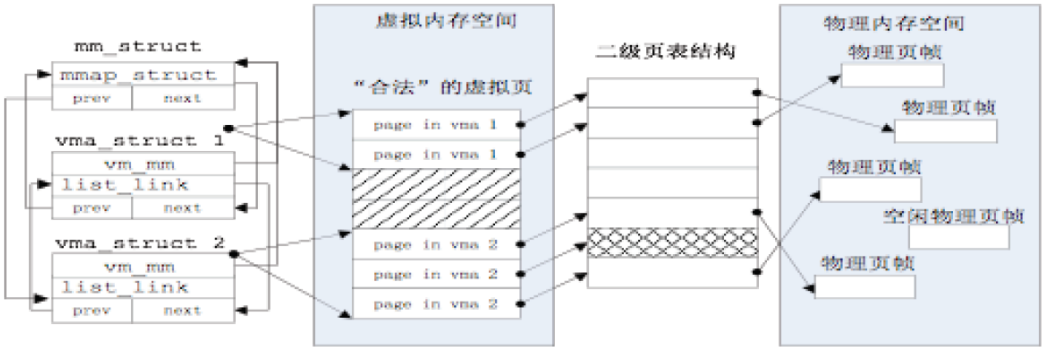
- 虚拟内存的作用和意义
  - 程序所能“看到”的内存不再是物理内存，实现了一层虚拟化，简称为**内存地址虚拟化**；通过这层虚拟化，我们可以实现：
  - 通过设置页表项来限定软件运行时的访问空间，确保软件运行不越界，完成**内存访问保护**的功能。
  - 可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术称为**按需分页**（demand paging）
  - 把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为**页换入换出**（page swap in/out）。可以让更多的程序在内存中并发运行。
- page fault

- 产生的原因
  - 目标页面不存在；页表项为空
  - 目标页面的物理页面不在内存中；页表项非空，`Present` 标志位为 `0`
  - 访问权限不符合；写只读页面
- page fault 产生时，页故障线性地址寄存器 `CR2` ：保存页错误异常的虚拟地址，中断栈中 `error code` ：保存页访问异常类型
  - error code：32bit；物理页不存在、写了只读页、权限异常等信息
- 判断引起异常的虚拟地址内存访问是否合法
  - “合法地址”访问：还未分配内存、数据被临时换出到磁盘上
  - “非法地址”访问：越界、写只读页
- `mm_struct` 和 `vma_struct` 描述应用程序运行所需的合法内存空间。
- 

- 

- 处理：`trap--> trap_dispatch-->pgfault_handler-->do_pgfault`

- `vma_struct` 描述应用程序对虚拟内存“需求”，包含一下字段：



- `vm_start`和`vm_end`：描述了一个连续地址的虚拟内存空间
- `vm_flags`：这个虚拟内存空间的属性，可读/可读写/可执行
- `list_link`：双向链表，按照从小到大的顺序把一系列用`vma_struct`表示的虚拟内存空间链接起来；vma之间
  的地址空间无交集
- `vm_mm`：指针，指向`mm_struct`
  - `mm_struct`包含所有虚拟内存空间的共同属性，包含字段：
    - `mmap_list`：双向链表头，链接了所有属于同一**页目录表(Page Directory Table)**的虚拟内存空间
    - `pgdir`：指向对应的**页目录表(Page Directory Table)**
    - `map_count`：记录`mmap_list`里面链接的`vma_struct`的个数
    - `mmap_cache`：指向当前正在使用的虚拟内存空间，利用局部性提高性能
    - `sm_priv`：private data for swap manager
- 操作函数：
  - `vma_create`：创建vma
  - `insert_vma_struct`：插入一个vma
  - `find_vma`：查询vma

设计实现

- `do_pgfault()`
  - 判断是否为“合法地址”访问
  - 若是，进行相应处理：分配空闲页/swap
  - 刷新TLB，然后调用iret中断，返回

```
    if ((ptep = get_pte(mm->pgdir, addr, 1)) == NULL) {
        cprintf("get_pte in do_pgfault failed\n");
        goto failed;
    }
    //如果页表为空，尝试分配一空闲页，匹配物理地址与逻辑地址，建立对应关系
    if (*ptep == 0) { //phy addr isn't exist, then alloc a page & map the phy
addr
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) { //失败内存不够退出
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    }
```

问题1：

请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处。

- 表项中 `PTE_A` 表示内存页是否被访问过，`PTE_D` 表示内存页是否被修改过，借助着两位标志位可以实现 Enhanced Clock 算法。

问题2：

如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

- 如果出现了页访问异常，那么硬件将引发页访问异常的地址将被保存在 `cr2` 寄存器中，设置错误代码，然后触发 `Page Fault` 异常。

练习2：补充完成基于FIFO的页面替换算法

设计思想

- 页面换入换出：
  - 页面换入： `do_pgfault()` 函数
    - 访问swap到硬盘的页面
  - 页面换出： `swap_out_vistim()` 函数
    - 消极换出：在调用 `alloc_pages` 函数获取空闲页时，此函数如果发现无法从物理内存页分配器（比如 `First Fit` ）获得空闲页，就会进一步调用 `swap_out` 函数 换出某页，实现一种消极的换出策略。
    - `FIFO` 替换算法会维护一个队列，队列按照页面调用的次序排列，越早被加载到内存的页面会越早被换出。
    - `map_swappable` 函数用于记录页访问情况相关属性
    - `swap_out_vistim` 函数依赖于 `map_swappable` 函数记录的页访问情况来挑选需要换出的页

```
_fifo_map_swappable(...)
{
    ...
    list_add(head, entry); //将最近用到的页面添加到次序的队尾
    return 0;
}
_fifo_swap_out_victim(...)
{
    ...
    list_entry_t *le = head->prev; //指出需要被换出的页
    assert(head!=le);
    //le2page 宏可以根据链表元素，获得对应 page 的指针p
    struct Page *p = le2page(le, pra_page_link);
    list_del(le); //将进来最早的页面从队列中删除
    assert(p !=NULL);
    *ptr_page = p; //将这一页的地址存储在ptr_page中
    return 0;
}
```

问题1：

如果要在ucore上实现"extended clock页替换算法"请给你的设计方案，现有的swap_manager框架是否足以支持在ucore中实现此算法？如果是，请给你的设计方案。如果不是，请给出你的新的扩展和基此扩展的设计方案。并回答：需要被换出的页的特征是什么？在ucore中如何判断具有这样特征的页？何时进行换入和换出操作？

对于每个页面都有两个标志位，分别为使用位和修改位，记为`<使用,修改>`。换出页的使用位必须为0，并且算法优先考虑换出修改位为零的页面。

当内存页被访问后，MMU 将在对应的页表项的 `PTE_A` 这一位设为1；

当内存页被修改后，MMU 将在对应的页表项的 `PTE_D` 这一位设为1。

当保存在磁盘中的内存需要被访问时，需要进行**换入**操作；

当位于物理页框中的内存被页面替换算法选择时，需要进行**换出**操作。

Lab 4 内核线程管理

> 内核线程的 创建、执行、调度

练习1：分配并初始化一个进程控制块

- 用户进程、内核进程（可把ucore看成一个内核进程）、用户线程、内核线程
  - 所有 内核线程 从属于同一个唯一的内核进程，即ucore内核本身，共享一个内核地址空间（内核虚拟内存地址空间）和其他资源；

```
struct proc_struct {                //进程控制块
    enum proc_state state;          //进程状态
    int pid;                        //进程ID
    int runs;                       //运行时间
    uintptr_t kstack;               //内核栈位置
    volatile bool need_resched;     //是否需要调度
    struct proc_struct *parent;     //父进程
    struct mm_struct *mm;           //进程的虚拟内存
    struct context context;         //进程上下文
    struct trapframe *tf;           //当前中断帧的指针
    uintptr_t cr3;                  //当前页表地址
    uint32_t flags;                 //进程
    char name[PROC_NAME_LEN + 1];//进程名字
    list_entry_t list_link;         //进程链表
    list_entry_t hash_link;         //进程哈希表
};
```

- 进程控制块 `PCB` 数据结构如上，字段含义为：
  - state：进程所处的状态。
    - PROC_UNINIT // 未初始状态
    - PROC_SLEEPING // 睡眠（阻塞）状态
    - PROC_RUNNABLE // 运行与就绪态
    - PROC_ZOMBIE // 僵死状态
  - pid：进程 id 号。
  - kstack：记录了分配给该进程/线程的内核栈的位置。
  - need_resched：是否需要调度

- parent：用户进程的父进程。
- mm：即实验三中的描述进程虚拟内存的结构体
- context：进程的上下文，用于进程切换。
- tf：中断帧的指针，总是指向内核栈的某个位置。中断帧记录了进程在被中断前的状态。
- cr3：记录了当前使用的页表的地址

设计实现：

- alloc_proc 函数
  - 分配一个新的 struct proc_struct 结构
  - 并进行初始化：
    - 指定的成员变量取特殊值：
      - `proc->state = PROC_UNINIT;`
      - `proc->pid = -1;`
      - `proc->cr3 = boot_cr3;` 由于是内核线程，共用一个虚拟内存空间
    - 其他成员变量均初始化为0

```
alloc_proc(void) {
...
        proc->state = PROC_UNINIT;  //设置进程为未初始化状态
        proc->pid = -1;             //未初始化的的进程id为-1
        proc->runs = 0;             //初始化时间片
        proc->kstack = 0;           //内存栈的地址
        proc->need_resched = 0;     //是否需要调度设为不需要
        proc->parent = NULL;        //父节点设为空
        proc->mm = NULL;            //虚拟内存设为空
        memset(&(proc->context), 0, sizeof(struct context));//上下文初始化
        proc->tf = NULL;            //中断帧指针置为空
        proc->cr3 = boot_cr3;       //页目录设为内核页目录表的基址
        proc->flags = 0;            //标志位
        memset(proc->name, 0, PROC_NAME_LEN);//进程名
...
}
```

问题1：

请说明proc_struct中 struct context context 和 struct trapframe *tf 成员变量含义和在本实验中的作用是啥？（提示通过看代码和编程调试可以判断出来）

- context 字段
  - 意义就在于内核线程之间进行切换的时候，将原先的线程运行的上下文保存下来这一作用。
- tf 字段
  - 作用在于在构造出了新的线程的时候，如果要将控制权交给这个线程，使用中断返回的方式进行的（跟lab1中切换特权级类似的技巧），因此需要构造出一个伪造的中断返回现场，也就是 trapframe，使得可以正确地将控制权转交给新的线程；具体切换到新的线程的做法为：
    - 调用switch_to函数。
    - 然后在该函数中进行函数返回，直接跳转到 forkret 函数。
    - 最终进行中断返回函数 __trapret，之后便可以根据 tf 中构造的中断返回地址，切换到新的线程了。
  - tf 是一个中断帧的指针，总是指向内核栈的某个位置：

- 当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。
- 当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。
- 除此之外，ucore 内核允许嵌套中断，因此为了保证嵌套中断发生时 tf 总是能够指向当前的 trapframe，ucore 在内核栈上维护了 tf 的链。

练习2：为新创建的内核线程分配资源

- do_fork函数
  - 作用：
    - ucore 一般通过 do_fork 实际创建新的内核线程。
    - 创建当前内核线程的一个**副本**，它们的执行**上下文、代码、数据都一样**，但是**存储位置不同**
    - 在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态
  - 执行步骤：
    - 1、分配并初始化进程控制块（ alloc_proc 函数）；
    - 2、分配并初始化内核栈，为内核进程（线程）建立栈空间（ setup_stack 函数）；
    - 3、根据 clone_flag 标志复制或共享进程内存管理结构（ copy_mm 函数）；
    - 4、设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（ copy_thread 函数）；
    - 5、为进程分配一个 PID（ get_pid() 函数）；
    - 6、把设置好的进程控制块放入 hash_list 和 proc_list 两个全局进程链表中;
    - 7、自此，进程已经准备好执行了，把进程状态设置为“就绪”态;
    - 8、设置返回码为子进程的 PID 号。

设计实现

```
do_fork() {
...
    if ((proc = alloc_proc()) == NULL) { //调用 alloc_proc() 函数申请内存块，如果失败，直接返回处理
        goto fork_out;//返回
    }
    proc->parent = current; //将子进程的父节点设置为当前进程
    if (setup_kstack(proc) != 0) { //调用 setup_stack() 函数为进程分配一个内核栈
        goto bad_fork_cleanup_proc; //返回
    }
    if (copy_mm(clone_flags, proc) != 0) { //调用 copy_mm() 函数复制父进程的内存信息到子进程
        goto bad_fork_cleanup_kstack; //返回
    }
    copy_thread(proc, stack, tf); //调用 copy_thread() 函数复制父进程的中断帧和上下文信息
    //将新进程添加到进程的 hash 列表中
    bool intr_flag;
    local_intr_save(intr_flag); //屏蔽中断，intr_flag 置为 1
    {
        proc->pid = get_pid(); //获取当前进程 PID
        hash_proc(proc); //建立 hash 映射
        list_add(&proc_list, &(proc->list_link)); //将进程加入到进程的链表中
        nr_process ++; //进程数加 1
    }
    local_intr_restore(intr_flag); //恢复中断
    wakeup_proc(proc); //一切就绪，唤醒子进程
    ret = proc->pid; //返回子进程的 pid
```

```
...
}
```

问题1：

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

是。PID 的确定过程中会检查所有进程的 PID，来确保 PID 是唯一的。

练习3：阅读代码，理解 proc_run 函数和它调用的函数如何完成进程切换的。

- 进程调度 `schedule()`
  - `schedule` 函数通过查找 proc_list 进程队列，在这里只能找到一个处于就绪态的 initproc 内核线程。于是通过 `proc_run` 和进一步的 switch_to 函数完成两个执行现场的切换。
  - 执行步骤：
    - 1、调度开始时，先屏蔽中断，设置当前内核线程 current->need_resched 为 0。
    - 2、在进程链表中，查找第一个可以被调度的程序，即在 proc_list 队列中查找下一个处于就绪态的线程或进程 next。
    - 3、找到这样的进程后，就调用 proc_run 函数，保存当前进程 current 的执行现场(进程上下文)，恢复新进程的执行现场，运行新进程，允许中断，完成进程切换。
  - `switch_to` 函数主要完成的是进程的上下文切换，先保存当前寄存器的值，然后再将下一进程的上下文信息保存到对于寄存器中。
  - `proc_run` 将当前的 CPU 的控制权交给指定的线程。
    - 保存 IF 位并且禁止中断；
    - 将 current 指针指向将要执行的进程；
    - 更新 TSS 中的栈顶指针；
    - 加载新的页表；
    - 调用 switch_to 进行上下文切换；
    - 当执行 proc_run 的进程恢复执行之后，需要恢复 IF 位。

问题1：

在本实验的执行过程中，创建且运行了几个内核线程?

- 从 `kern_init()` 开始：
  1. 通过`proc_init()` 函数调用完成了 `idleproc` 内核线程（pid=0）和 `initproc` 内核线程(pid=1)的初始化。
  2. 通过 `cpu_idle()` 唤醒了 0 号 idle 进程。
     - `idleproc` 内核线程：循环调用`schedule`函数
       - `schedule` 函数通过查找 proc_list 进程队列，在这里只能找到一个处于就绪态的 `initproc` 内核线程。于是通过 `proc_run` （调用`switch_to`）切换到 `initproc` 内核线程。
- 2个
  - idle_proc，为第 0 个内核线程，在完成新的内核线程的创建以及各种初始化工作之后，进入死循环，用于调度其他进程或线程；
  - init_proc，被创建用于打印 "Hello World" 的线程。本次实验的内核线程，只用来打印字符串。

问题2：

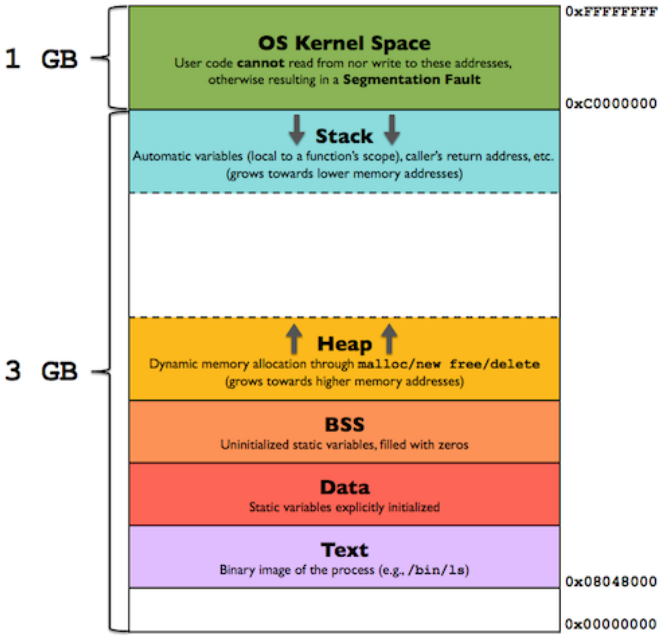
语句 `local_intr_save(intr_flag);....local_intr_restore(intr_flag);` 在这里有何作用?请说明理由

- 在进行进程切换的时候，需要避免出现中断干扰这个过程，所以需要在上下文切换期间清除 IF 位屏蔽中断，并且在进程恢复执行后恢复 IF 位。

Lab 5 用户进程管理

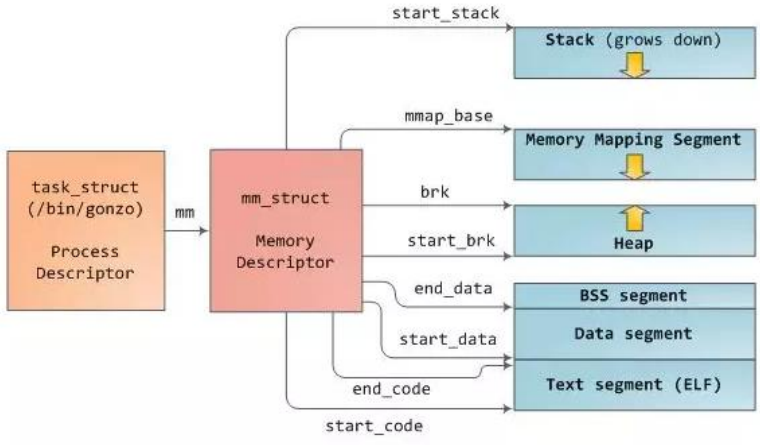
> 用户进程创建
>
> 系统调用的实现机制
>
> 通过系统调用 `sys_fork/sys_exec/sys_exit/sys_wait` 进行进程管理

- **每个进程/线程会有两个栈，一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。**
  - **进程执行系统调用陷入内核态时，要从用户栈转移到内核栈之中。**
  - **TSS记录了进程内核栈的栈指针。**
  - 内核栈/用户栈意义在于权限和系统安全。
  - 在创建进程时，把进程的trapframe放在给进程的内核栈分配的空间的顶部
- 内核线程
  - 共享内核虚拟内存地址空间
- 用户环境
  - 特点：
    - 一方面与存储空间相关
      - **限制用户进程可以访问的物理地址空间**
      - 让各个用户进程之间的物理内存空间访问不重叠
    - 另一方面与执行指令相关
      - 限制用户进程可执行的指令：不能让用户进程执行特权指令（比如修改页表起始地址），从而保证用户进程无法破坏系统。
      - 给用户进程一个 “服务窗口”
        - 用户进程可以通过这个“窗口”向操作系统提出服务请求，由操作系统来帮助用户进程完成需要特权指令才能做的各种服务。
      - 和一个“中断窗口”
        - 让用户进程不主动放弃使用CPU时，操作系统能够通过这个“中断窗口”强制让用户进程放弃使用CPU，从而让其他用户进程有机会执行。
  - 定义：
    - 建立**用户虚拟空间的页表** 和 支持页换入换出机制的用**户内存访存错误异常服务例程**
    - 应用程序执行的**用户态CPU特权级**：在用户态CPU特权级，应用程序只能执行一般指令，如果特权指令，结果不是无效就是产生“执行非法指令”异常；
    - 系统调用机制：给用户进程提供“服务窗口”；
    - 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被
      强制打断，CPU控制权将被操作系统的中断服务例程使用。
- 用户进程的运行状态，5种
  - 创建（new）态
    - **进程控制块**的创建
  - 就绪（ready）态
    - 进程执行所需要的**虚拟内存空间**，**执行代码**，**要处理的数据**等都准备好了，

        - 做好“执行准备”，放入放入到**就绪队列**中
    - 运行（running）态
    - 等待（blocked）态
    - 退出（exit）态
        - 回收用户进程占用的各种资源，回收进程控制块
- 用户进程的虚拟地址空间，分成了两部分
    - 一块映射到所有用户进程都共享的**内核虚拟地址空间**（0xC0100000），映射到**同样的物理内存空间**中，这样在物理内存中只需放置一份内核代码。
    - 另外一块是用户虚拟地址空间（0x800020），映射到不同且没有交集的物理内存空间中。
    - 

- 创建并执行用户进程
    - 由第二个内核线程initproc通过把hello应用程序执行码覆盖到initproc的用户虚拟内存空间来创建
    - do_execve 函数主要工作：
        - 首先做好用户态**内存空间清空**准备。
        - 然后**加载应用程序执行码**到当前进程的新创建的用户态虚拟空间中（load_icode）
            - load_icode：
                1. 创建（申请所需内存空间）并初始化内存管理数据结构mm；
                2. 创建进程新的页目录表，拷贝ucore内核虚空间映射的内核页表（boot_pgdir）的内容到此新目录表中，最后让`mm->pgdir`指向此页目录表
                3. 解析ELF格式的执行程序，根据ELF格式说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构

中



4. 分配物理内存空间；根据ELF中各个段的起始位置**确定虚拟地址**，在页表中建立好虚拟地址和物理地址的映射关系；把执行程序各个段的内容拷贝到虚拟地址对应的物理内存中
5. 给用户进程设置用户栈，建立用户栈的vma结构，栈的大小为256个页（1MB）
6. 至此，进程的内存管理vma和mm数据结构已经建立完成。于是把mm->pgdir赋值到cr3
   寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好
7. 重新设置进程的中断帧；使得在执行中断返回指令 iret 后，能够让 CPU 转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断;

- 此时 initproc 将按产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”后，将切换到用户进程hello的第一条语句位置_start处开始执行。

- 进程退出
  - 释放进程占用的资源，分为两步：
    - 首先由进程本身完成大部分资源的占用内存回收工作
      - 用户态的exit函数，会访问sys_exit系统调用，进行大部分资源回收，通过执行内核函数 `do_exit` 来完成对当前进程的退出处理。
    - 然后由此进程的**父进程**完成剩余资源占用内存的回收工作：进程控制块，进程的内核栈
      - 父进程完成对子进程的最终回收工作
- 系统调用 vs 用户态函数调用
  - 系统调用：通过“INT”指令发起调用；通过“IRET”指令完成调用返回；
  - 函数调用：通过“CALL”指令发起调用；通过“RET”指令完成调用返回；

练习1: 加载应用程序并执行

设计实现

- **进程切换总是在内核态中发生**，当内核选择一个进程执行的时候，首先切换内核态的上下文（EBX、ECX、EDX、ESI、EDI、ESP、EBP、EIP 八个寄存器）以及内核栈。完成内核态切换之后，内核需要使用 IRET 指令将 trapframe 中的用户态上下文恢复出来，返回到进程态，在用户态中执行进程。
- load_icode 函数需要填写的部分的实现思路：
  - 将 trapframe 的代码段设为 USER_CS；
  - 将 trapframe 的数据段、附加段、堆栈段设为 USER_DS；
  - 将 trapframe 的栈顶指针设为 USTACKTOP；
  - 将 trapframe 的代码段指针设为 ELF 的入口地址 elf->e_entry；
  - 将 trapframe 中 EFLAGS 的 IF 置为 1。

```
load_icode(...) {
...
    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = USTACKTOP; //0xB0000000
    tf->tf_eip = elf->e_entry;
    tf->tf_eflags = FL_IF; //FL_IF为中断打开状态
...
}
```

问题1：

见上。创建并执行用户进程。

练习2: 父进程复制自己的内存空间给子进程

- copy_range 函数的具体执行流程是遍历父进程指定的某一段内存空间中的每一个虚拟页，如果这个虚拟页是存在的话，为子进程对应的同一个地址（但是页目录表是不一样的，因此不是一个内存空间）也申请分配一个物理页，然后将前者中的所有内容复制到后者中去，然后为子进程的这个物理页和对应的虚拟地址（事实上是线性地址）建立映射关系；
- 由于在内核态进行数据拷贝，因此需要知道 src 物理页 和 des 物理页 对应的内核虚拟地址，然后通过memcpy 对内核虚拟地址进行数据拷贝。具体流程如下：
    1. 找到父进程指定的某一物理页对应的内核虚拟地址；
    2. 找到需要拷贝过去的子进程的对应物理页对应的内核虚拟地址；
    3. 通过 memcpy 将前者的内容拷贝到后者中去；
    4. 为子进程当前分配这一物理页映射上对应的在子进程虚拟地址空间里的一个虚拟页；

```
copy_range(...) {
...
    void * kva_src = page2kva(page);
    void * kva_dst = page2kva(npage);
    memcpy(kva_dst, kva_src, PGSIZE);
    ret = page_insert(to, npage, start, perm);
...
}
```

问题1：

如何设计实现”Copy on Write 机制“

“Copy on Write” 机制的主要思想为：进程执行 fork 系统调用进行复制的时候，父进程不会简单地将整个内存中的内容复制给子进程，而是暂时共享相同的物理内存页；而当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，然后在自己的内存页中进行修改。

- 首先，在进行 fork 操作的时候不直接复制内存，映射到同一个物理页面，并在各自的PTE上将这个页置成只读不可写。
- 然后，另外一个处理在于出现了内存页访问异常的时候，会将共享的内存页复制一份，然后在新的内存页进行修改。

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

- fork
  - 首先当程序执行 fork 时，fork 使用了系统调用 SYS_fork，而系统调用 SYS_fork 则主要是由 do_fork 和 wakeup_proc 来完成的。
- exec
  - 当应用程序执行的时候，会调用 SYS_exec 系统调用，而当 ucore 收到此系统调用的时候，则会使用 do_execve() 函数来实现，do_execve() 函数主要时完成用户进程的创建工作，同时使用户进程进入执行。
- wait
  - 当执行 wait 功能的时候，会调用系统调用 SYS_wait，而该系统调用的功能则主要由 do_wait 函数实现，主要工作就是父进程如何完成对子进程的最后回收工作。
    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 schdule() 函数选择的进程执行，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
  - 当执行 exit 功能的时候，会调用系统调用 SYS_exit，而该系统调用的功能主要是由 do_exit 函数实现。

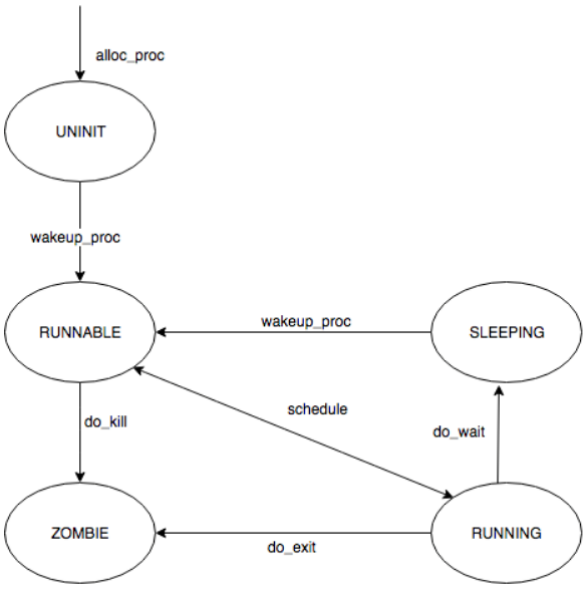
问题1：

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

- fork 执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID。
- fork 不会影响当前进程的执行状态，但是会将子进程的状态标记为 RUNNALB，使得可以在后续的调度中运行起来；
- exec 完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。
- exec 不会影响当前进程的执行状态，但是会修改当前进程中执行的程序；
- wait 是等待任意子进程的结束通知。wait_pid 函数等待进程 id 号为 pid 的子进程结束通知。这两个函数最终访问 sys_wait 系统调用接口让 ucore 来完成对子进程的最后回收工作。
- wait 系统调用取决于子进程的执行状态是不是PROC_ZOMBIE，如果是，状态不变，当前进程继续完成对子进程的最终回收工作；如果不是，会将当前进程置为 SLEEPING 态，等待执行了 exit 的子进程将其唤醒；
- exit 会把一个退出码 error_code 传递给 ucore，ucore 通过执行内核函数 do_exit 来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。
  - exit 会将当前进程的状态修改为 ZOMBIE 态，并且会将父进程唤醒（修改为RUNNABLE），然后主动让出 CPU 使用权；

问题2：

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）



- 进程首先在 cpu 初始化或者 sys_fork 的时候被创建，当为该进程分配了一个进程控制块之后，该进程进入 uninit态(在proc.c 中 alloc_proc)。
- 当进程完全完成初始化之后，该进程转为runnable态。
- 当到达调度点时，由调度器 sched_class 根据运行队列rq的内容来判断一个进程是否应该被运行，即把处于runnable态的进程转换成running状态，从而占用CPU执行。
- running态的进程通过wait等系统调用被阻塞，进入sleeping态。
- sleeping态的进程被wakeup变成runnable态的进程。
- running态的进程主动 exit 变成zombie态，然后由其父进程完成对其资源的最后释放，子进程的进程控制块成为unused。
- 所有从runnable态变成其他状态的进程都要出运行队列，反之，被放入某个运行队列中。

Lab 6 调度器

> 调度管理机制

练习1: 使用 Round Robin 调度算法

- idle进程：（区分 进程调度 和 idle进程）
  - 当cpu没有进程可以执行的时候，系统应该如何工作
  - ucore建立了一个单独的进程(kern/process/proc.c 中的 idleproc)作为 cpu 空闲时的 idle 进程，这个程序是通常一个死循环。
- 调用进程调度函数`schedule`的位置：
  - `proc.c::do_exit`：用户线程执行结束，主动放弃CPU控制权
  - `proc.c::do_wait`：用户线程等待子进程结束，主动放弃CPU控制权
  - `proc.c::cpu_idle`：idleproc 内核线程，循环调用schedule函数
  - `proc.c::init_main`：initproc内核线程等待所有用户进程结束，如果没有结束，就主动放弃CPU控制权;
  - `sync.h::lock`：在获取锁的过程中，如果无法得到锁，则主动放弃CPU控制权

    - `trap.c::trap`：如果在当前进程在用户态被打断去，且当前进程控制块的成员变量need_resched设置为1，则当前线程会放弃CPU控制权
- 进程切换过程（在只有两个进程的情况下）
    1. 首先在执行某进程A的用户代码时，出现了一个 trap (例如是一个外设产生的中断)，这个时候就会从进程A的用户态切换到内核态(过程(1))，并且保存好进程A的trapframe；
    2. 当内核态处理中断时发现需要进行进程切换时，ucore要通过schedule函数选择下一个将占用CPU执行的进程（即进程B），然后会调用proc_run函数，proc_run函数进一步调用switch_to函数，**切换到进程B的内核态**(过程(2))，继续进程B上一次在内核态的操作，并通过iret指令，最终将执行权转交给进程B的用户空间(过程(3))。
    3. 当进程B由于某种原因发生中断之后(过程(4))，会从进程B的用户态切换到内核态，并且保存好进程B的trapframe；
    4. 当内核态处理中断时发现需要进行进程切换时，即需要切换到进程A，ucore再次切换到进程A的内核态(过程(5))，会执行进程A上一次在内核调用schedule (具体还要跟踪到switch_to 函数)函数返回后的下一行代码，这行代码当然还是在进程A的上一次中断处理流程中。
    5. 最后当进程A的中断处理完毕的时候，执行权又会反交给进程A的用户代码(过程(6))。
    6. 注意：
        - 调度发生在应用进程的内核态
        - 内核启动第一个用户进程的过程，实际上是从进程启动时的内核状态切换到该用户进程的内核状态的过程
- **调度框架**：
    - （kern/schedule/sched.c 只实现调度器框架；调度算法在单独的文件（default_sched.[ch]；）
    - 三个基本操作：
        - 在就绪进程集合中选择 `sched_class_pick_next()`
        - 进入就绪进程集合 `sched_class_enqueue()`
        - 离开就绪进程集合 `sched_class_dequeue()`
    - 影响调度选择的重要因素：就绪进程的等待时间，执行进程的执行时间
        - 这种进程变化的情况就形成了调度器相关的一个变化感知操作：**timer时间事件感知操作。**
        - 随着时间推移，调度器可以调整进程控制块中与进程调度相关的属性值（比如消耗的时间片、进程优
          先级等），并最终导致调选择新的进程
        - 每次timer中断调用`sched_class_proc_tick`来金总timer时间事件感知操作
- 数据结构：
    - 进程池：调度器引入 run-queue（简称rq,即运行队列）的概念，通过链表结构管理进程。
        - 每个list_entry_t 通过宏 le2proc 对应到了 struct proc_struct *
- `Round Robin` 调度算法：
    - 让所有 runnable 态的进程分时轮流使用 CPU 时间。`Round Robin` 调度器维护当前 runnable 进程的有序运行队列。当前进程的时间片用完之后，调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。
    - 进程控制块proc_struct中增加了一个成员变量time_slice，用来记录进程当前剩余的可运行时间片段。

问题1：

请理解并分析sched_class中各个函数指针的用法，并结合Round Robin 调度算法描ucore的调度执行过程

- 在ucore中调用调度器的主体函数（不包括 init，proc_tick）的代码仅存在在 wakeup_proc 和 schedule，前者的作用在于将某一个指定进程放入可执行进程队列中，后者在于将当前执行的进程放入可执行队列中，然后将队列中选择的下一个执行的进程取出执行；
- 当需要将某一个进程加入就绪进程队列中，则需要将这个进程的能够使用的时间片进行初始化，然后将其插入到使用链表组织的队列的对尾；这就是具体的 Round-Robin enqueue 函数的实现；
- 当需要将某一个进程从就绪队列中取出的时候，只需要将其直接删除即可；
- 当需要取出执行的下一个进程的时候，只需要将就绪队列的队头取出即可；
- 每当出现一个时钟中断，则会将当前执行的进程的剩余可执行时间减 1，一旦减到了 0，则将其标记为可以被调度的，这样在 ISR 中的后续部分就会调用 schedule 函数将这个进程切换出去；

问题2：

请在实验报告中简要说明如何设计实现"多级反馈队列调度算法"，给出概要设计，鼓励给出详细设计

- 在 proc_struct 中添加总共 N 个多级反馈队列的入口，每个队列都有着各自的**优先级**，编号越大的队列优先级约低，并且优先级越低的队列上时间片的长度越大，为其上一个优先级队列的两倍；并且在 PCB 中记录当前进程所处的队列的优先级；
- 处理调度算法初始化的时候需要同时对 N 个队列进行初始化；
- 在处理将进程加入到就绪进程集合的时候，观察这个进程的时间片有没有使用完，如果使用完了，就将所在队列的优先级调低，加入到优先级低 1 级的队列中去，如果没有使用完时间片，则加入到当前优先级的队列中去；
- 在同一个优先级的队列内使用时间片轮转算法；
- 在选择下一个执行的进程的时候，有限考虑高优先级的队列中是否存在任务，如果不存在才转而寻找较低优先级的队列；（有可能导致饥饿）
- 从就绪进程集合中删除某一个进程就只需要在对应队列中删除即可；
- 处理时间中断的函数不需要改变；

练习2: 实现 Stride Scheduling 调度算法

- Stride Scheduling 调度算法
    - 1、为每个 runnable 的进程设置一个当前状态 stride，表示该进程当前的调度权。另外定义其对应的 pass 值，表示对应进程在调度后，stride 需要进行的累加值。
- 2、每次需要调度时，从当前 runnable 态的进程中选择 stride 最小的进程调度。对于获得调度的进程 P，将对应的 stride 加上其对应的步长 pass（只与进程的优先权有关系）。
    - 3、在一段固定的时间之后，回到步骤 2，重新调度当前 stride 最小的进程。
- *default_sched.c*
    - `proc_stride_comp_f`：优先队列的比较函数，主要思路就是通过步数相减，然后根据其正负比较大小关系
    - `stride_init`：进行调度算法初始化的函数，在本 stride 调度算法的实现中使用了斜堆来实现优先队列，因此需要对相应的成员变量进行初始化
    - `stride_enqeue`：在将指定进程加入就绪队列的时候，需要调用斜堆的插入函数将其插入到斜堆中，然后对时间片等信息进行更新
    - `stride_dequeue`：将指定进程从就绪队列中删除，只需要将该进程从斜堆中删除掉即可
    - `stride_pick_next`: 选择下一个要执行的进程，根据stride算法，只需要选择stride值最小的进程，即斜堆的根节点对应的进程即可
    - `stride_proc_tick`：每次时钟中断需要调用的函数，仅在进行时间中断的ISR中调用

Lab 7 同步互斥

> 理解操作系统的同步互斥的设计实现；

> 理解底层支撑技术：禁用中断、定时器、等待队列；
>
> 理解信号量（semaphore）机制的具体实现；
>
> 理解管程机制，在ucore内核中增加基于管程（monitor）的条件变量（condition variable）的支持；
>
> 使用同步机制解决进程同步问题。

- 同步互斥的底层支撑
  - 在硬件级保证 **定时器、屏蔽/使能中断**和**等待队列** 等机器指令的原子性。从而大大简化同步互斥原语。
  - **定时器（timer）**：
    - 实现基于时间长度的睡眠等待和唤醒机制
    - `timer_init (timer_t *timer, struct proc_struct *proc, int expires)`：对某定时器进行初始化，让它在 expires 时间片之后唤醒 proc 进程。
    - `add_timer (timer t *timer)`：向 `timer_list` 添加某个初始化过的timer_t，该定时器在指定时间后被激活，并将对应的进程唤醒至runnable（如果当前进程处在等待状态）。
    - `del_timer (timer_t *time)`：向 `timer_list` 删除（或者说取消）某一个定时器。
    - `run_timer_list (void)`：更新当前系统时间点，遍历当前所有处在系统管理内的定时
      器，找出所有应该激活的计数器，并激活它们。
      - 该过程在且只在每次定时器中断时被调用。在ucore 中，其还会调用调度器事件处理程序。
  - **屏蔽与使能中断**
    - 实现了对临界区的互斥操作（单核）
    - ```
      local_intr_save (intr_flag);
      {
      临界区代码
      }
      local_intr_restore (intr_flag);
      ```
  - **等待队列（wait_queue，wq）**：
    - 等待的事件可以是：睡眠结束、时钟到达、任务完成、资源可用等
    - ```
      //让wait与进程关联，且让当前进程关联的wait进入等待队列queue，当前进程睡眠
      void wait_current_set(wait_queue_t *queue, wait_t *wait, uint32_t
      wait_state);
      //把与当前进程关联的wait从等待队列queue中删除
      void wait_current_del(queue, wait);
      //唤醒与wait关联的进程
      void wakeup_wait(wait_queue_t *queue, wait_t *wait, uint32_t
      wakeup_flags, bool del);
      //唤醒等待队列上挂着的第一个wait所关联的进程
      void wakeup_first(wait_queue_t *queue, uint32_t wakeup_flags, bool
      del);
      //唤醒等待队列上所有的等待的进程
      void wakeup_queue(wait_queue_t *queue, uint32_t wakeup_flags, bool
      del);
      ```

练习1：理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题

- 信号量
  - 信号量为一个特殊变量；信号量的V操作采用进程可执行原语semSignal(s)，来通过信号量s传送信
    号；信号量的P操作采用进程可执行原语semWait(s)，来通过信号量s接收信号。
  - 数据结构:
    - ```
      typedef struct {
          int value; //信号量的当前值
          wait_queue_t wait_queue; //信号量对应的等待队列，一个等待的进程会挂在此等待队列上
      } semaphore_t;
      ```
  - 操作函数：
    - P操作函数 `__down(semaphore_t *sem, uint32_t wait_state)`：
      - 首先关掉中断，然后判断当前信号量的value是否大于0。
      - 如果value>0，则表明可以获得信号量，故让value减一，并打开中断返回即可；
      - 如果不是>0，则表明无法获得信号量，故需要将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。
    - V操作函数 `__up(semaphore_t *sem, uint32_t wait_state)`：
      - 首先关中断
      - 如果信号量对应的wait queue中没有进程在等待，直接把信号量的value加一，开中断返回；
      - 如果有进程在等待且进程等待的原因是semophore设置的，则调用wakeup_wait函数将waitqueue中等待的第一个wait删除，且把此wait关联的进程唤醒，开中断返回。
  - 信号量的计数器 value 具有有如下性质：
    - value>0，表示共享资源的空闲数
    - vlaue<0，表示该信号量的等待队列里的进程数
    - value=0，表示等待队列为空

问题1：

请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

将内核信号量机制迁移到用户态的最大麻烦在于，用于**保证操作原子性的禁用中断机制、以及 CPU 提供的 Test and Set 指令机制都只能在内核态下运行**，而使用软件方法的同步互斥又相当复杂，这就使得没法在用户态下直接实现信号量机制；于是，为了方便起见，可以将信号量机制的实现放在 OS 中来提供，然后使用**系统调用的方法**统一提供出若干个管理信号量的系统调用，分别如下所示：

- 申请创建一个信号量的系统调用，可以指定初始值，返回一个信号量描述符(类似文件描述符)；
- 将指定信号量执行 P 操作；
- 将指定信号量执行 V 操作；
- 将指定信号量释放掉；

给内核级线程提供信号量机制和给用户态进程/线程提供信号量机制的异同点在于：

- 相同点：
  - 提供信号量机制的代码实现逻辑是相同的；
      ```

- 不同点：
  - 由于实现原子操作的中断禁用、Test and Set 指令等均需要在内核态下运行，因此提供给用户态进程的信号量机制是通过系统调用来实现的，而内核级线程只需要直接调用相应的函数就可以了；

练习2：完成内核级条件变量和基于内核级条件变量的哲学家就餐问题

- 管程和条件变量
  - 管程把共享变量和对它进行操作集中并封装起来，作为一个隔离区。
  - 管程的定义：
    - 一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据
  - 管程由四部分组成：
    - 管程内部的共享变量；
    - 管程内部的条件变量；
    - 管程内部并发执行的进程；
    - 对局部于管程内部的共享数据设置初始值的语句。
  - 管程每次只允许一个进程进入管程，从而确保进程之间**互斥**。
  - 条件变量 CV 可理解为进程的等待队列。
  - 管程数据结构：
    - ```
      typedef struct monitor{
          semaphore_t mutex; //二值信号量
          int next_count; //由于发出 singal_cv 而睡眠的进程个数
          condvar_t *cv; //条件变量
      } monitor_t;
      ```
    - cv 条件变量：`wait_cv` 与 `signal_cv` 操作
      - 执行 `wait_cv` ，会使得等待某个条件Cond为真的进程能够离开管程并睡眠，且让其他进程进入管程继续执行
      - 进入管程的某进程设置条件Cond为真并执行 `signal_cv` 时，能够让等待某个条件Cond为真的睡眠进程被唤醒，从而继续进入管程中执行
    - mutex 二值信号量：实现每次只允许一个进程进入管程
    - next_count 由于发出 singal_cv 而睡眠的进程个数
      - 这是由于发出 signal_cv 的进程A会唤醒由于 wait_cv 而睡眠的进程B，由于管程中只允许一个进程运行，所以进程B执行会导致唤醒进程B的进程A睡眠，直到进程B离开管程，进程A才能继续执行，这个同步过程是通过信号量next完成的；而next_count表示了由于发出 singal_cv 而睡眠的进程个数。
  - 条件变量数据结构：
      ```

- ```c
  typedef struct condvar{
      semaphore_t sem;
      int count;
      monitor_t * owner;
  } condvar_t;
  ```

- sem：这个信号量用于让发出 wait_cv 操作的等待某个条件Cond为真的进程睡眠，而让发出 signal_cv 操作的进程通过这个sem来唤醒睡眠的进程

- count：等在这个条件变量上的睡眠进程的个数

- owner：此条件变量的宿主是哪个管程。

设计实现：

- phi_take_forks_condvar() 函数实现思路：（拿刀叉）
  1. 获取管程的锁
  2. 将自己设置为饥饿状态
  3. 判断当前叉子是否足够就餐，如不能，等待其他人释放资源
  4. 释放管程的锁
- phi_put_forks_condvar() 函数实现思路：（放刀叉）
  1. 获取管程的锁
  2. 将自己设置为思考状态
  3. 判断左右邻居的哲学家是否可以从等待就餐的状态中恢复过来

```
哲学家 -> 试试拿刀叉 -> 能拿 -> signal唤醒被wait阻塞的进程 -> 阻塞自己
                  |               |                          A
                  |               V                          |
                  ->不能拿 -> wait阻塞自己                    |
                                                             |
哲学家    ->    放刀叉    ->    让左右两边试试拿刀叉 ->    有哲学家睡在signal 唤醒他
```

- cond_signal() 函数实现思路：
  1. 判断条件变量的等待队列是否为空
  2. 修改 next 变量上等待进程计数，跟下一个语句不能交换位置，为了得到互斥访问的效果，关键在于访问共享变量的时候，管程中是否只有一个进程处于 RUNNABLE 的状态
  3. 唤醒等待队列中的某一个进程
  4. 把自己等待在 next 条件变量上
  5. 当前进程被唤醒，恢复 next 上的等待进程计数
- cond_wait() 函数实现思路：
  1. 修改等待在条件变量的等待队列上的进程计数
  2. 释放锁
  3. 将自己等待在条件变量上
  4. 被唤醒，修正等待队列上的进程计数

问题1：

请在实验报告中给出给用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级　　提供条件变量机制的异同。
```

管程的实现中互斥访问的保证是完全基于信号量的，也就是如果按照上文中的说明使用 syscall 实现了用户态的信号量的实现机制，那么就完全可以按照相同的逻辑在用户态实现管程机制和条件变量机制；
- 相同点：基本的实现逻辑相同；
- 不同点：最终在用户态下实现管程和条件变量机制，需要使用到操作系统使用系统调用提供一定的支持; 而在内核态下实现条件变量是不需要的；

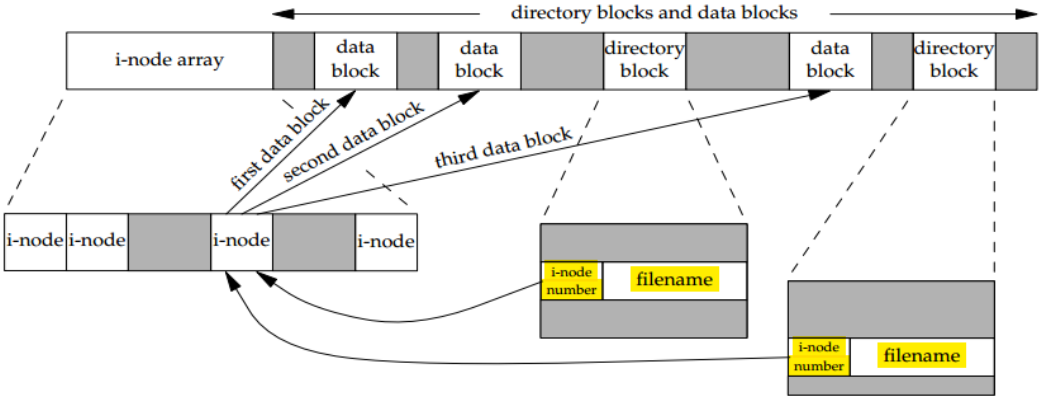
问题1：

能否不用基于信号量机制来完成条件变量？如果不能，请给出理由，如果能，请给出设计说明和具体实现。

不清楚

Lab 8 文件系统

> 基本的文件系统系统调用的实现方法
>
> 基于索引节点组织方式的Simple FS文件系统的设计与实现
>
> 文件系统抽象层-VFS的设计与实现

- ucore模仿了UNIX的文件系统设计，文件系统架构主要由四部分组成:
  - **通用文件系统访问接口层**：从用户空间到文件系统的标准访问接口
  - **文件系统抽象层**：屏蔽不同文件系统的实现细节
  - **Simple FS文件系统层**：文件系统抽象层各种抽象函数的具体实现。文件系统数据保存在磁盘上。
  - **外设接口层**：屏蔽不同硬件细节
- 四类主要的数据结构：
  - 超级块（SuperBlock）
  - 索引节点（inode）
  - 目录项（dentry）
  - 文件（file）
  - 

- **通用文件系统访问接口层**
  - 文件操作：
    - open：
      - 在读写一个文件之前，首先要用open系统调用将其打开。
      - 第一个参数指定文件的路径名，可使用绝对路径名；第二个参数指定打开的方式，可设置为O_RDONLY、O_WRONLY、O_RDWR，分别表示只读、只写、可读可写。
      - 在打开一个文件后，就可以使用它返回的**文件描述符fd**对文件进行相关操作。
    - close:

        - 在使用完一个文件后，还要用close系统调用把它关闭，其参数就是文件描述符fd。这样它的文件描述符就可以空出来，给别的文件使用。
    - read:
        - read系统调用有三个参数：一个指定所操作的文件描述符，一个指定读取数据的存放地址，最后一个指定读多少个字节。
        - 把实际读到的字节数返回
    - write
  - 目录操作：
    - 跳转到某个目录，对应的用户库函数是chdir
    - 读目录的内容了，即**列出目录中的文件或目录名**：
        - 通过opendir函数打开目录
        - 通过readdir来获取目录中的文件信息
            - readdir需要调用**获取目录内容**的**特殊系统调用**sys_getdirentry
        - 读完后还需通过closedir函数来关闭目录
  - 与文件相关的open、close、read、write用户库函数对应的是sys_open、sys_close、sys_read、sys_write四个系统调用接口。与目录相关的readdir用户库函数对应的是sys_getdirentry系统调用。
  - 通过这些系统调用函数接口，调用内核文件系统抽象层的file接口和dir接口。
- **文件系统抽象层 - VFS**
  - file接口：
  - 进程控制块 files_struct 字段；file数据结构；

```
struct files_struct {
  struct inode *pwd; //进程当前执行目录的内存inode指针
  struct file *fd_array; //进程打开文件的数组
  atomic_t files_count; //访问此文件的线程个数
  semaphore_t files_sem; //确保对进程控制块中fs_struct的互斥访问
};
struct file {
  enum {
      FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
  } status; //访问文件的执行状态
  bool readable; //文件是否可读
  bool writable; //文件是否可写
  int fd; //文件在filemap中的索引值
  off_t pos; //访问文件的当前位置
  struct inode *node; //该文件对应的内存inode指针
  int open_count; //打开此文件的次数
};
```

  - inode 接口：
    - inode 是VFS结构中的重要数据结构，因为它实际负责把不同文件系统的特定索引节点信息（甚至不能算是一个索引节点）统一封装起来，避免了进程直接访问具体文件系统。

- **Simple FS 文件系统**
  - 文件系统通常保存在磁盘上。SFS文件系统在磁盘上的布局为：
  -

- 文件系统mount时，硬盘上的SFS文件系统的超级块superblock和freemap被加载到内存中。
- 索引节点：记录**文件内容的存储位置**以及**文件名与文件内容的对应关系**
  - sfs_disk_inode记录了文件或目录的内容存储的索引信息，该数据结构在硬盘里储存，需要时读入内存
  - sfs_disk_entry表示一个目录中的一个文件或目录，包含该项所对应inode的位置和文件名，同样也在硬盘里储存，需要时读入内存。
- 内存中的索引节点：sfs_inode
  - 一个内存inode是在打开一个文件后才创建的，如果关机则相关信息都会消失；dirtyinode 数据能够被写回到磁盘。
  - Inode的文件操作函数：
    - sfs_openfile不用做什么事；
    - sfs_close需要把对文件的修改内容写回到硬盘上，这样确保硬盘上的文件内容数据是最新的；
    - sfs_read和sfs_write函数都调用了一个函数sfs_io，并最终通过访问硬盘驱动来完成对文件内容数据的读写。
  - Inode的目录操作函数：
    - 相对于sfs_open，sfs_opendir只是完成一些open函数传递的参数判断，没做其他更多的事情。
    - 目录的close操作与文件的close操作完全一致。
    - 由于目录的内容数据与文件的内容数据不同，所以**读出目录的内容数据**的函数是**sfs_getdirentry**，其主要工作是获取目录下的文件inode信息。

- 设备层文件 IO 层
  - stdin设备文件 对应 键盘
  - stdout设备文件 对应 CONSOLE（串口、并口和文本显示器）
  - disk0设备文件 对应 承载SFS文件系统的磁盘设备

练习1: 完成读文件操作的实现

- 打开文件
  - **通用文件访问接口层的处理流程：**
  - 1. 调用如下用户态函数：`open->sys_open->syscall`，从而引起系统调用进入到内核态。
    2. 到了内核态后，通过中断处理例程，会调用到sys_open内核函数，并进一步调用sysfile_open内核函数。
    3. 到了这里，需要把位于**用户空间的字符串**"sfs_filetest1"拷贝到**内核空间中的字符串**path中，并进入VFS层.
  - **文件系统抽象层（VFS）的处理流程**
    - 调用VFS的file_open函数：
      - 1. 给当前用户进程**分配了一个file数据结构**的变量
        1. 给这个即将打开的文件分配一个file数据结构的变量；
        2. 放入当前进程的打开文件数组 `current->fs_struct->filemap[]` 中的一个空闲元素；
        3. 索引值就是最终要返回到用户进程并赋值给变量fd1；

2. 进一步调用vfs_open函数，找到path所对应的**VFS索引节点inode。**
    1. 通过vfs_lookup找到path对应文件的inode；
        1. 调用vop_lookup函数：查找到根目录“/”下对应文件sfs_filetest1的索引节点，返回此索引节点
    2. 调用vop_open函数打开文件。
3. 把当前进程的current->fs_struct->filemap[fd]（即file所指变量）的成员变量node指针指向了代表sfs_filetest1文件的索引节点inode。

- **SFS文件系统层的处理流程**
    - `.vop_lookup=sfs_lookup`，`sfs_lookup` 调用 `sfs_dirent_search_nolock` 函数来查找与路径名匹配的目录项，根据目录项找到**SFS磁盘inode**，读入，创建**SFS内存inode。**

- **路径查询处理流程：**
- `sfs_lookup`：Parse path relative to the passed directory DIR, and hand back the inode content
    - `sfs_lookup_once`：find inode corresponding the file name in the DIR
        - `sfs_dirent_search_nolock`：find the file's inode number
            - **read every file entry** in the DIR, compare file name with each entry->name. If equal, then return the file's inode number
            - `sfs_dirent_read_nolock`：read the file entry from disk block which contains the entry with given index
                - `sfs_bmap_load_nolock`：according to DIR's simpleFS in memory inode + the slot index of file entry，find index of disk block which contains this file entry
                - `sfs_rbuf`：read the content of file entry in the disk block，to sfs_disk_entry
        - `sfs_load_inode`：load the content of inode with the the NO. of disk block

问题1：

请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案

- 在磁盘上生成一个特定的临时文件来作为 pipe 机制的缓冲区；
- 或者在 VFS 层，建立一个虚拟的 pipe 文件，把数据缓存在磁盘上；

练习2: 完成基于文件系统的执行程序机制的实现

- 初始化进程控制块中的 files_struct 字段 `proc->filesp = NULL;`
- load_icode 主要是将文件加载到内存中执行，从上面的注释可知分为了一共七个步骤：
    - 1、建立内存管理器
    - 2、建立页目录
    - 3、将文件逐个段加载到内存中，这里要注意设置虚拟地址与物理地址之间的映射
    - 4、建立相应的虚拟内存映射表
    - 5、建立并初始化用户堆栈
    - 6、处理用户栈中传入的参数
    - 7、最后很关键的一步是设置用户进程的中断帧
    - 8、发生错误还需要进行错误处理。
-

问题1：

请在实验报告中给出设计实现基于”UNIX的硬链接和软链接机制“的概要设方案，鼓励给出详细设计方案

- 保存在磁盘上的 inode 信息均存在一个 nlinks 变量用于表示当前文件的被链接的计数，可以依此实现硬链接和软链接机制；