

浙江大学



操作系统  
设计报告

陈翰逸 李仁钟 萧芷晴  
3160100052 3160104920 3160105360

January 15, 2019

# Contents

<b>1 功能介绍</b>	<b>4</b>
1.1 内存管理 . . . . .	4
1.2 进程管理 . . . . .	5
1.3 文件系统 . . . . .	6
<b>2 详细设计</b>	<b>7</b>
2.1 内存管理 . . . . .	7
2.1.1 物理内存管理 . . . . .	7
2.1.2 虚拟内存设计 . . . . .	9
2.1.3 总结 . . . . .	11
2.2 进程管理: 多级反馈队列 . . . . .	12
2.2.1 进程数据结构 . . . . .	12
2.2.2 进程创建 . . . . .	14
2.2.3 进程退出 . . . . .	15
2.2.4 进程调度 . . . . .	16
2.2.5 进程等待机制 . . . . .	17
2.2.6 kill 进程 . . . . .	17
2.3 进程管理 : UC/OS II . . . . .	18
2.3.1 进程数据结构 . . . . .	18
2.3.2 任务管理操作 . . . . .	20
2.3.3 进程调度 . . . . .	21

2.4 文件系统 . . . . .	22
2.4.1 初始化文件系统 . . . . .	22
2.4.2 FAT32 结构 . . . . .	22
2.4.3 FAT32 数据结构 . . . . .	23
2.4.4 FAT32 文件系统实现思路 . . . . .	25
2.4.5 源码结构 . . . . .	28
<b>3 测试过程与结果 . . . . .</b>	<b>29</b>
3.1 内存测试 . . . . .	29
3.1.1 slab 分配 . . . . .	30
3.1.2 slab 批量分配以及释放 . . . . .	30
3.1.3 malloc/free 虚拟内存测试 . . . . .	31
3.1.4 malloc/free 的增加测试 . . . . .	32
3.2 进程管理: 多级反馈队列 . . . . .	33
3.2.1 进程创建与退出测试 . . . . .	33
3.2.2 进程调度测试 . . . . .	34
3.2.3 kill 进程测试 . . . . .	35
3.3 进程管理 : UC/OS II . . . . .	36
3.3.1 进程创建与退出测试 . . . . .	36
3.3.2 进程调度 . . . . .	37
3.4 文件系统 . . . . .	39
3.4.1 模拟测试环境 . . . . .	39
3.4.2 实际测试 . . . . .	44
<b>4 讨论及体会 . . . . .</b>	<b>49</b>
<b>5 组内成员分工 . . . . .</b>	<b>51</b>
<b>6 附注 . . . . .</b>	<b>52</b>

7 参考文献	53
--------	----

# Chapter 1

## 功能介绍

### 1.1 内存管理

本操作系统的内存管理结构设计包括物理内存的管理、用户进程的虚拟地址段管理和相应的申请内存系统调用。

对比助教组，此操作系统重构了 slab，设计了进程下的红黑树管理 VMA，完成了 malloc 和 free 设计。

对于物理内存，其由 bootmem, buddy 和 slab 三个子模块组成。直接与进程部分和虚拟内存交互的是 slab 模块。其通过缓存的方式提高申请以及释放内存的速度。所以对于助教的代码进行了重构，优化了代码结构。便于在其他部分，尤其是 VMA 的实现中调用。

对于 VMA 管理，使用了红黑树的结构，优化了使用速度，使得 LRU 算法可以在此基础上拓展。

为了配合 VMA 的实现，加入了对于用户虚拟内存的申请和释放函数。用户程序在执行的过程中可以利用 malloc/free 来申请和释放内存。

提供的用户指令：

Table 1.1: 用户指令

<code>slabone</code>	申请 1000 字节的内存块，不释放。 输出申请前后的 Buddy 信息
<code>slabck</code>	申请 99 个 100 字节的内存块，释放。 输出申请和释放前后的 Buddy 信息
<code>memc</code>	malloc 10 个 10 大小的虚拟内存，然后逐个 free。 输出 VMA 在 <code>mm_struct</code> 的红黑树中的颜色
<code>mem_alltree</code>	malloc 7 个 10 大小的虚拟内存，然后从小地址到大地址 free。 输出每步中红黑树的 BFS 下的结构。

## 1.2 进程管理

进程模块总的可以分为进程创建与结束、进程调度、进程等待机制、杀死进程等四个方面的功能。

进程创建用于在系统中创建新的进程，而新创建的进程会自动设定为当前进程子进程。进程结束为在进程结束之后将退出系统时进行的操作。可支持按照指定优先级创建新进程。

进程调度用于对系统中所同时存在的多个进程进行调度，合理分配硬件资源。

进程等待机制用于协调进程之间的同步，通过进程等待机制可以使得将一个进程挂起，而等到另一个指定进程完成之后才将挂起进程恢复执行。但在此机制中挂起进程与被等待进程之间必须满足父子关系。

杀死进程使得用户或者程序可以强制结束特定进程的执行，以灵活管理进程。

其中提供接口供用户使用的指令有：

Table 1.2: 进程模块提供的用户指令

execk
创建新的进程
execkwait
创建新的进程并且使当前进程挂起以等待新创建进程结束
execprio
创建新的进程并且指定新进程的初始优先级
kill
根据进程的 pid 号杀死进程
ps
显示系统中的全部进程信息

## 1.3 文件系统

对于用户来说，文件系统能够帮助用户在 FAT32 文件系统中实现以下指令：其中底层文件系统应该具备：

- 读文件
- 写文件
- 删 除文件
- 创建文件
- 移动文件
- 创建目录
- 读目录
- 写目录

实现这些功能可以使得以下指令得以运行

Table 1.3: 提供的用户指令

指令	說明
ls	列举目录中的内容
cd	改变当前的工作目录
cat	输出文件内容
touch	创建文件
mkdir	创建目录
rm	删除文件
write	改变文本文件的内容
mv	移动文件

# Chapter 2

## 详细设计

### 2.1 内存管理

#### 2.1.1 物理内存管理

##### 概述

物理内存管理含三个模块：Bootmem、Buddy、Slab。Bootmem 机制是内核在启动时对内存的一种简单的页面管理方式。它为建立页表管理代码中的数据结构提供动态分配内存的支持，这种机制仅仅用在系统引导时，它为整个物理内存建立起一个页面位图。Buddy 系统以页为单位对内存进行粗粒度的管理，以 Buddy 算法来记录现存的空闲连续页框块的情况，以尽量避免为满足对小块的请求而把大块的空闲块进行分割。以此减少外碎片。Slab 则是以字节为单位的细粒度内存管理，将内存离散成多个级别的大小，为每个级别维护空闲列表，减小内碎片。

##### Bootmem、Buddy

这两个模块全部沿用自教的系统，在此只简单介绍。

Bootmem 只在系统初始化时用到。它以一个位图来表示物理内存的使用情况，每一个物理页都映射到位图上的一个字节。同时，使用 `bootmm_info` 结构对于连续的物理内存进行管理。

在进行内存分配时，因为我们只进行连续的申请而不释放，所以从上次分配的内存末尾地址开始分配即可。分配时插入对应的 `bootmm_info`，使得内存段信息便于查找。

内存释放时，讲对应的 `bootmm_info` 从数组中移除即可。

Buddy 系统的关键在于伙伴块的 index 和物理地址的确认。而关键的数据结

构是一个 freelist 数组，每个阶数对应一个空闲链表，初始化时，将所有空闲页加入空闲列表中，此后不必再进行处理。每次分配时，从对应阶数的空闲列表向高阶列表遍历。直到找到空闲块。如果此空闲块比需求更大则进行拆分，同时将剩余的部分加入相应阶数的空闲列表。在释放内存时，还需要检查伙伴块是否可以合并，同时向上合并，直到无法合并。

Listing 2.1: Buddy 系统核心数据结构

```

1 struct freelist {
2     unsigned int nr_free;           // Number of free blocks
3     struct list_head free_head;    // Linked list
4 };
5
6 struct buddy_sys {
7     unsigned int buddy_start_pfn, buddy_end_pfn; // Physical frame number of buddy
8     struct page *start_page;          // Address of page array
9     struct lock_t lock;              // Exclusive clock
10    struct freelist freelist[MAX_BUDDY_ORDER + 1]; // Freelists for each order
11 };

```

## Slab

Slab 是针对一些经常分配并释放的对象，如进程信息等的内存申请系统，这些对象的大小一般比较小，如果直接采用 Buddy 系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。而 slab 分配器是基于对象进行管理的，大小相近的对象归为一类，每当要申请这样一个对象，slab 分配器就从一个 slab 列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab 分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。

其中的核心数据结构是 `kmem_cache` 和 `kmem_cache_cpu`。前者包括这个 slab 的所有信息，包括大小，分配完的页和等待分配的页。后者则指向当前可用的页。

Listing 2.2: Slab 系统核心结构

```

1 // 当前页
2 struct kmem_cache_cpu {
3     struct page *page;
4 };
5
6 struct kmem_cache {
7     unsigned int size;           // 大小
8     unsigned int objsize;        // 分配大小
9     struct kmem_cache_node node; // partial和full的页
10    struct kmem_cache_cpu cpu;   // 当前页
11 };

```

Listing 2.3: 对外接口

```

1 extern void *kmalloc(unsigned int size); // 申请内存
2 extern void kfree(void *obj);           // 释放内存

```

Kmalloc 是内核代码最常用的分配函数，无论是在进程还是虚拟内存中都会被用到。它读入所需的内存字节数。返回值是内核直接映射段的地址表示。假如返回值是 0 则说明分配失败。

Kfree 是与之对应的内存释放函数，传入参数为待释放的内存的首地址。

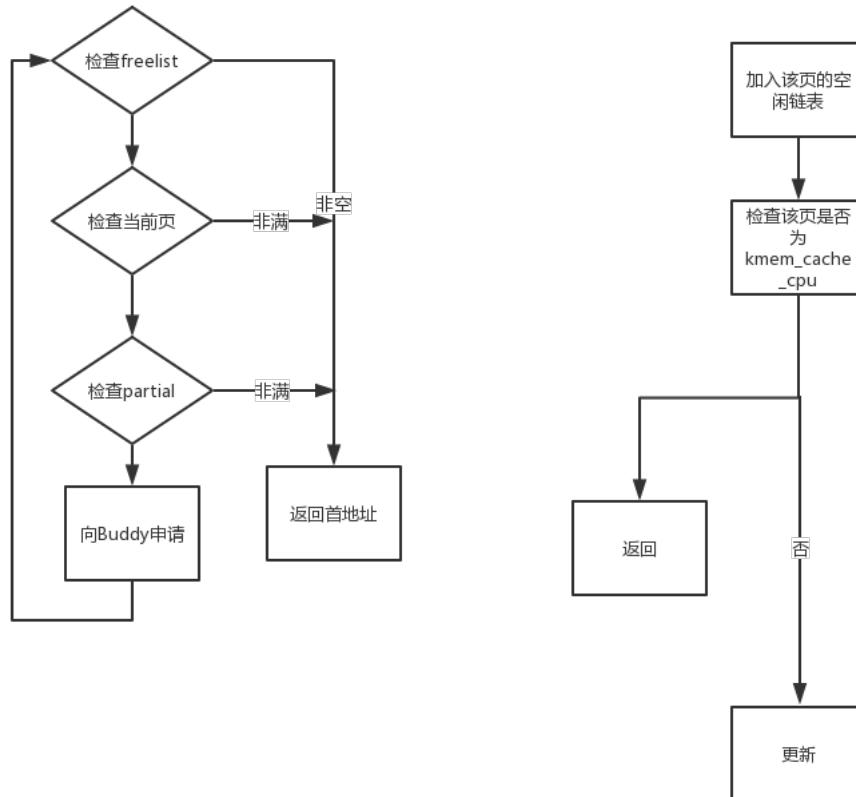


Figure 2.1: kmalloc/kfree 分配逻辑

### 2.1.2 虚拟内存设计

虚拟地址空间其中所有实现全部为基于红黑树的实现。

#### **mm\_struct**

进程控制块中会维护 **mm\_struct** 指针，来管理该进程在虚拟地址空间的信息。**mm\_struct** 结构中主要存储 VMA 的头，红黑树的根和进程页表的地址。

Listing 2.4: **mm\_struct**

```

1 struct mm_struct {
2     struct vma_struct *root;           // VMA root
  
```

```

3   struct vma_struct *mmap_cache; // Latest used VMA
4   struct rb_node mm_rb;        // 红黑树
5   int map_count;              // VMA count
6   pgd_t *pgd;                // Page table entry
7   unsigned int start_code, end_code;
8   unsigned int start_data, end_data;
9   unsigned int start_brk, brk;
10  unsigned int start_stack;
11 };

```

`mm_struct` 指向的红黑树的根可以使得每个进程都有自己可以高效访问的虚拟内存结构。这样可以更好地支持对虚拟内存性能要求较高的一些拓展。

VMA 的各项操作都对应红黑树的各项操作。因为没有与文件映射相关的操作，所以把 VMA 的合并接口留了出来，以后也可以予以拓展。

Listing 2.5: `vma_struct`

```

1 struct vma_struct {
2   struct mm_struct *vm_mm;
3   unsigned long vm_start, vm_end;      // 起止
4   struct vma_struct *vm_next;          // 链表结构
5   struct vma_struct *vma_left, *vma_right;
6   struct rb_node vm_rb;
7 };
8
9 struct vma_struct *find_vma(struct mm_struct *mm, unsigned long addr);
10 int insert_vm_struct(struct mm_struct *mm, struct vma_struct *vma);
11 void del_vm_struct(struct vma_struct *vma);

```

VMA 通过下文自主设计的 malloc 和 free 进行申请和释放。在介绍 malloc/free 之前先简要介绍继承自助教代码结构的 TLB。

## TLB

TLB 设计继承自助教组代码，在这里简单提及。

TLB 的设计是为了减少虚拟内存到物理内存的转换。而 TLB\_refill 则是在页表中无法查询到合法虚拟地址值时所作的操作。

遗憾的是此部分是无法体现 VMA 红黑树结构的正确性的，只能体现虚拟内存的访问，所以另外设计了 malloc 和 free 来申请和释放内存从而进行测试。

## malloc/free

Listing 2.6: malloc/free 接口

```

1 int *malloc(unsigned int size);
2 void free(unsigned int addr);
3
4 unsigned long do_map(unsigned long addr, unsigned long len, unsigned long flags);
5 int do_unmap(unsigned long addr, unsigned long len);
6 int is_in_vma(unsigned long addr);

```

malloc 使用 `get_unmapped_area` 通过链表循序访问找到下一个没有被 map, 同时又大小足够的内存将这块内存初始化为了 VMA 加入当前进程 `mm_struct` 中的红黑树中。

free 将这块 VMA 取出检查前后是否为被占用的空间若未被占用，则合并到空闲链表中删除其在红黑树中的节点。

### malloc/free 导致的内存泄漏

广义的内存泄漏会在 free 多个间隔的小内存时发生。尚未对这些泄漏的内存进行整合。从设计角度来说，当内存不够时，可以整合这些小片段，从而提供更多的内存。

### 红黑树结构

红黑树结构与 Linux 相近，但是删除了 Linux 中出于 vma 要映射文件而加入的部分。并重构手写得到。

红黑树关键定义以及部分宏如下。其性质不再赘述。

Listing 2.7: `struct st_rb_node`

```

1 typedef struct st_rb_node {
2
3     unsigned long parent_color;
4     #define RB_RED    1
5     #define RB_BLACK  0
6     struct st_rb_node *left, *right;
7 } __attribute__((aligned(sizeof(long)))) rb_node;
8 //对齐是为了把红黑树的颜色挤进节点中，节约空间
9
10 #define rb_parent(r)   ((rb_node *)((r)->parent_color & ~3))
11 #define rb_color(r)    ((r)->parent_color & 1)
12 #define rb_is_red(r)   (!rb_color(r))
13 #define rb_is_black(r) rb_color(r)
14 #define rb_set_red(r)  do { (r)->parent_color &= ~1; } while (0)
15 #define rb_set_black(r) do { (r)->parent_color |= 1; } while (0)

```

### 2.1.3 总结

此内存系统在助教组代码的基础上，重构了 slab，加入了虚拟内存的红黑树管理，以及设计了 malloc 和 free 函数。达到了实现原有设计，同时加以创新的要求。也将 VMA 合并等接口留好，具有向用户程序加载、页置换等方面的可拓展性。

## 2.2 进程管理: 多级反馈队列

### 2.2.1 进程数据结构

#### task\_struct 结构

task\_struct 结构即为进程控制块，是进程模块中最主要的结构。本系统中的 task\_struct 设计参考 Linux 系统中的进程控制块结构。结合本操作系统的实际需要对 Linux 的进程控制块结构进行了删减和修改，使之更符合本系统的需要。

task\_struct 结构具体信息见下图。

Listing 2.8: task\_struct 结构

```
1 //进程PCB结构, 保存进程状态等相关信息
2 struct task_struct {
3     pid_t pid;                                // 进程pid号
4     unsigned char name[TASK_NAME_LEN];        // 进程名
5     pid_t parent;                             // 父进程pid号
6     int ASID;                                // 进程地址空间id号
7     int state;                               // 进程状态
8
9     unsigned int time_cnt;                   // 进程所拥有的时间片
10    unsigned int priority;                  // 进程的优先级
11
12    struct regs_context context;            // 进程寄存器信息
13    struct mm_struct *mm;                  // 进程地址空间结构指针
14    FILE *task_files;                     // 进程打开文件指针
15
16    struct list_head sched_list;           // 用于进程调度
17    struct list_head task_list;            // 用于进程链表
18};
```

## task\_union 结构

此结构利用 C 语言中的 union 关键字，巧妙地将进程的内核栈和进程控制块合并在一起进行存储，设计思想同样是源于 Linux 系统。

task\_union 结构具体信息见下图。

Listing 2.9: task\_union 数据结构

```

1 union task_union {
2     struct task_struct task;           //进程控制块
3     unsigned char kernel_stack[KERNEL_STACK_SIZE]; //进程的内核栈
4 };

```

## regs\_context 结构

regs\_context 结构用于存储进程的寄存器信息，属于进程模块的辅助结构。主要用于当系统中出现进程调度、进程退出等需要进行进程切换操作的情况下时保存进程的寄存器信息，其实也就是在保存进程的状态信息，以便在进程再次恢复执行时能够从一个正确的状态启动。

而在该结构中，不只是保存通用寄存器内容，也会对 CP0 协处理器中的 EPC 寄存器内容进行保存，而 EPC 寄存器中所存储的正是进程重新开始执行的指令地址。

regs\_context 结构具体信息见下图。

Listing 2.10: regs\_context 结构

```

1 //寄存器信息结构，主要用于进程调度时的进程切换
2 struct regs_context {
3     unsigned int epc;
4     unsigned int at;
5     unsigned int v0, v1;
6     unsigned int a0, a1, a2, a3;
7     unsigned int t0, t1, t2, t3, t4, t5, t6, t7;
8     unsigned int s0, s1, s2, s3, s4, s5, s6, s7;
9     unsigned int t8, t9;
10    unsigned int hi, lo;
11    unsigned int gp, sp, fp, ra;
12 };

```

## ready\_list 结构

ready\_list 结构用于存储进程的多级反馈就绪队列，属于进程模块的辅助结构。主要用于管理进程的就绪队列，在进程调度、进程退出等需要进行进程切换操作的情况下时存储处于就绪队列的进程，不同的队列代表不同的优先级的就绪队列，按照数组地址顺序匹配优先级大小排列，以便于查找在进程调度遍历多级就绪队列查找高优先级的进程。

`ready_list` 结构具体信息见下图。

Listing 2.11: `ready_list` 结构

```

1 struct ready_list {
2     int size;                                //记录当前队列长度
3     struct list_head head;                  //记录队列的头部节点
4 };

```

## 其他与进程有关的全局链表

本操作系统中以下的全局链表采用 Linux 中的 `struct list_head` 结构。

Table 2.1: 进程有关全局链表

链表名	用途
<code>total_tasks</code>	系统中的全部进程
<code>total_wait</code>	系统中的所有等待进程
<code>total_exited</code>	系统中的所有结束或未分配进程

### 2.2.2 进程创建

用户输入的 `execprio`, `execk`, `execwait` 等不同的进程创建指令, 都统一调用 `exec_from_kernel` 函数。在 `exec_from_kernel` 函数中调用 `task_create` 函数创建新的进程, 并且提供给 `task_create` 函数一个函数指针, 作为进程开始执行的入口地址, 默认设置新创建进程的入口地址为内核函数 `runprog`。

而在 `task_create` 函数中, 首先我们为新创建进程分配一个新的 pid 号, 创建相应的 `task_struct` 结构, 以及初始化结构内部必要的信息。为新建进程分配资源, 建立对应的虚拟地址空间即 `mm_struct` 结构, 初始化进程打开文件。其次, 我们设置进程的状态信息, 比如堆栈、进程入口、进程状态, 根据优先级设置默认分配的时间片, 并设置完毕后将新创建的进程加入就绪队列。

在 `init_kernel` 函数中会初始化进程相关的部分, `idle` 进程总是第一个被创建的, pid 为 0 的进程。`kernel shell` 进程是在 `init_kernel()` 中调用 `task_create()` 第二个被创建的, pid 为 1 的进程。

进程创建过程的层次结构如下。

Table 2.2: 创建进程过程的层次结构

<b>exec_from_kernel()</b> 创建进程过程开始的函数	
	<b>task_create()</b> 进程创建函数
	如果是创建内核线程
	设置新创建进程的入口函数为 runprog
	在 runprog 函数中加载外部程序作为内核线程代码
	是否等待子进程
	在 wait_pid 函数中唤醒进程，如果被唤醒，从此处继续执行

注：蓝色部分表示在新创建进程运行的部分

Table 2.3: **task\_create** 函数处理层次结构

<b>task_create()</b> 进程退出函数	
	<b>pid_alloc()</b> 分配 pid 号
	<b>mm_create()</b> 是否地址空间结构
	<b>cur-&gt;state = TASK_READY</b>
	<b>INIT_LIST_HEAD()</b> 初始化调度链表和进程链表
	<b>context</b> 寄存器上下文相关设置
	<b>add_task()</b> 将新进程加入所有进程链表中
	<b>add_ready()</b> 将新进程加入多级就绪队列中

### 2.2.3 进程退出

首先会调用唤醒父进程函数 `wakeup_parent` 函数，检查要退出进程的父进程是否在等待它，如果是，则唤醒父进程，使父进程恢复到可执行状态。然后将要退出进程的状态更改为终止状态，并且释放分配给进程的一些资源、如关闭已打开文件、释放 `mm_struct` 结构（描述进程地址空间的结构）等，最后将对应进程移出多级就绪队列并放入进程结束链表中，并且通过调度算法寻找下一个可执行的进程。在每次进程调度发生时，会对进程结束链表进行清理，释放资源。

进程退出过程的层次结构如下。

Table 2.4: pc\_exit 函数处理层次结构

pc_exit() 进程退出函数	
	file_close() 关闭打开文件
	mm_delete() 是否地址空间结构
	cur->state = TASK_EXITED
	wakeup_parent() 检查是否有父进程在等待要退出进程，如果有，唤醒父进程
	next = find_next_task() 调度算法寻址下一个执行进程
	pid_free() 释放进程 pid 号
	将退出进程放入结束链表，加载下一个执行的进程上下文信息

#### 2.2.4 进程调度

在本操作系统中采用的调度算法是以多级反馈队列调度算法为基础进行修改的。

初始化后，我们的多级就绪队列中有初始的 idle 进程、和 kernel shell 进程，idle 进程和 kernel shell 进程也有默认的时间片，如果不创建其他进程时，kernel shell 进程将会不断地和 idle 进程进行调度切换。

多级就绪队列的级数可以使用宏任意设定，不影响调度算法的核心部分。调度算法的核心是在当前进程时间片未用完时，可以被更高优先级的进程实时抢占，如果当前进程的时间片已用完，将会在多级就绪队列中找到下一个不为当前进程的可执行的进程执行，如果没有比当前进程更高优先级的进程，则会寻找比当前进程优先级更低的进程，如果比当前优先级更低的进程只有 idle 进程（默认设为最低优先级），那么将会执行 idle 进程。当然，如果仍存在比 idle 进程优先级更高的进程（至少有 kernel shell 进程），idle 进程执行时将会重新被抢占。

在找到下一个要执行的进程后，就只需要通过进程切换操作将寄存器中的信息更新为新的要执行的进程的信息即可开始新的进程的执行。

Table 2.5: 进程调度的层次结构

pc_schedule() 时钟中断触发的进程调度函数	
	clear_exited() 清理结束链表
	cur ->time_cnt 更新时间片
	cur->priority 更新优先级
	如果时间没有用完
	next = find_prio_task() 寻找一个更高优先级的进程
	如果时间片已用完
	next = find_next_task() 寻找一个可执行的进程
如果 next 不是当前进程, pc_switch()	

## 2.2.5 进程等待机制

本操作系统中会引入进程等待机制，主要是用于实现进程之间的同步，使得父进程可以挂起以等待子进程执行完毕之后再继续执行。进程等待机制的实现主要依赖于两个函数 `wait_pid()` 和 `wakeup_parent()`。

`wait_pid()` 函数由父进程系统调用，参数为父进程所要等待的子进程的 pid 号。该函数会将父进程的状态修改为等待状态，并将其从调度链表中移除而添加到进程等待链表中。然后通过调用进程调度算法选取下一个要执行的进程，并将下一个要执行的进程的上下文信息加载到寄存器中，即完成等待子进程操作。

而 `wakeup_parent()` 函数由子进程在进程退出时系统调用的 `pc_exit()` 函数中调用。该函数会首先检查当前进程的父进程是否在等待它，如果是的话，则会将父进程的状态修改为可执行状态，并将其从进程等待链表中移除而加入到多级就绪队列中，将其放入后台多级队列中相应队列的开头位置，这样使得父进程可以在进程调度时相对于同级队列中的进程优先被调度。

## 2.2.6 kill 进程

本操作系统实现 `kill` 进程系统调用，主要是为了保证系统/用户对进程管理拥有比较大的自由度，使得系统/用户可以通过提供进程 pid 号对进程进行杀死操作。

当调用 `kill` 进程系统调用的 `pc_kill()` 时，首先会判断要杀死的进程是否是不能杀死的进程以及所提供的进程 pid 号是否合法，如果合法，则将进程的状态更改为进程结束状态，释放分配给进程的资源，并将进程从调度链表或等待链表中移除而放入进程结束链表中，而进程结束链表会在进程调度发生时进行清理操作，释放相关资源。

## 2.3 进程管理：UC/OS II

由于进程部分总体工作量相对较少，又因为在实验课上提及的 uC/OS II(Micro Control Operation System Two) 系统非常有特点，因此在进程部分也尝试移植了该系统的任务管理部分。

采用的调度算法参考于 uC/OS II(Micro Control Operation System Two) 系统。它是一个可以基于 ROM 运行的、可裁减的、抢占式、实时多任务内核，具有高可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的实时操作系统。它可以简单地视为是一个多任务调度器，在这个任务调度器之上完善并添加了和多任务操作系统相关的系统服务，比如信号量等。uC/OS II 绝大部分代码使用 C 语言编写，CPU 硬件相关部分是使用汇编语言编写的，其源代码公开。

在其任务调度部分，有许多有意思的特点，区别于我们之前在课堂上学习到的调度算法，比如使用优先级而不是进程 pid 来唯一标示进程，存在优先级继承机制，任务调度和事件机制的结合等。在这里，我们只移植了较为基础的纯粹的任务调度部分。

同时，进程创建和进程退出部分，一些数据结构和我们上述调度算法大同小异，接下来我们只重点阐述不同的部分。

### 2.3.1 进程数据结构

#### `task_struct` 结构

区别于之前的调度算法，uC/OS II 中不是通过 pid 来唯一标示一个进程，而是使用优先级，每一个任务处于一个不同的优先级，并有一张任务就绪表来管理所有就绪的任务，为了提高效率，很多的操作都是通过位运算来进行的，优先级使用 8 位数字保存，也因此在该系统中最多含有 64 个任务 (64 个不同的优先级)。为了位运算的方便，参考该系统，我们也将优先级拆分为高位和低位分开保存。

`task_struct` 结构中最重点的部分如下图。

Listing 2.12: `task_struct` 数据结构

```

1 unsigned char prio; /* task priority (highest = 0) */
2 unsigned char x; /* Bit position in group corresponding to task priority */
3 unsigned char y; /* Index in ready table corresponding to task priority */
4 unsigned char bit_x; /* Bit mask to access bit position in ready table */
5 unsigned char bit_y; /* Bit mask to access bit position in ready group */

```

## 任务就绪表

不同于我们之前的调度算法，或者是 Linux/Windows 的调度算法，该系统使用数组来管理所有的就绪任务。此任务就绪表用于管理就绪任务。

任务就绪表的定义如下图。

Listing 2.13: 任务就绪表定义

```
1 unsigned char rdy_group; /* Ready list group */
2 unsigned char rdy_table[OS_RDY_TBL_SIZE];
3 /* Table of tasks which are ready to run */
```

## 任务控制块和就绪表的关联

8 位优先级按照小端保存时，分别为 D7, D6, D5, D4, D3, D2, D1, D0。在 uC/OS II 中最多 64 个优先级，即优先级的最大数值为 **00111111**，仅使用了 6 位数字表示（最大数值的优先级是最低优先级，0u 是最高优先级）。所以任务控制块中的 y 表示任务优先级的高 3 位 (D5, D4, D3)，指明 **rdy\_table** 的下标，即 **rdy\_group** 的数据位。任务控制块中的 x 表示任务优先级的低三位 (D2, D1, D0)，指明 **rdy\_table[y]** 中的哪一位数据位。**bit\_y** 和 **bit\_x** 主要是为了寻找最高优先级时匹配使用。

相应变量的计算方式如下图。

Listing 2.14: 相关位运算

```
1 ptcb->y = (unsigned char)(prio >> 3);
2 ptcb->x = (unsigned char)(prio & 0x07);
3 ptcb->bit_y = (unsigned char)(1 << ptcb->y);
4 ptcb->bit_x = (unsigned char)(1 << ptcb->x);
```

## map\_table 表

用于寻找最高优先级任务时进行匹配，找出目标任务的核心算法在于确定某数值为 1 的最低位（每个任务具有不同的优先级），uC/OS II 中的具体实现是，借助 **map\_table**，通过高效的位运算完成。

**map\_table** 表的结构如下图。

Listing 2.15: **map\_table** 表

```
1 unsigned char const map_table[256] = {
2     0u, 0u, 1u, 0u, 2u, 0u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
3     /* 0x00 to 0x0F */
4     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
5     /* 0x10 to 0x1F */
```

```

6      5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
7      /* 0x20 to 0x2F */
8      4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
9      /* 0x30 to 0x3F */
10     6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
11     /* 0x40 to 0x4F */
12     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
13     /* 0x50 to 0x5F */
14     5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
15     /* 0x60 to 0x6F */
16     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
17     /* 0x70 to 0x7F */
18     7u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
19     /* 0x80 to 0x8F */
20     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
21     /* 0x90 to 0x9F */
22     5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
23     /* 0xA0 to 0xAF */
24     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
25     /* 0xB0 to 0xBF */
26     6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
27     /* 0xC0 to 0xCF */
28     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
29     /* 0xD0 to 0xDF */
30     5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
31     /* 0xE0 to 0xEF */
32     4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u
33     /* 0xF0 to 0xFF */
34 };

```

### 2.3.2 任务管理操作

#### 登记就绪任务

当某个任务处于就绪状态时，系统就将该任务登记在任务就绪表中，即在该任务优先级对应的任务就绪表中的位置设置值为 1。对应的位操作如下图。

Listing 2.16: 登记就绪任务操作

```

1 rdy_group |= ptcb->bit_y;
2 rdy_table[y] |= ptcb->bit_x;

```

#### 注销就绪任务

当某个任务处于就绪状态时，系统就将该任务在任务就绪表中注销，即在该任务优先级对应的任务就绪表中的位置重置值为 0。对应的位操作如下图。

Listing 2.17: 注销就绪任务操作

```

1 unsigned char y;
2 y = current_task->y;
3 rdy_table[y] &= ~current_task->bit_x;    // set the bit in rdy_table[y] zero
4 if (rdy_table[y] == 0x00) {                  // no ready task
5     rdy_group &= ~current_task->bit_y;        // set the bit in rdy_group zero
6 }

```

## 寻找最高优先级任务

用于寻找最高优先级任务时进行匹配，找出目标任务的核心算法在于确定某数值为 1 的最低位 (每个任务具有不同的优先级)。本质上就是哈希算法。对应的位操作如下图。

Listing 2.18: 寻找最高优先级任务操作

```

1 unsigned char y;
2 y = map_table[rdy_group]; // group the highest task in
3 prio_highest_rdy = (unsigned char)((y << 3) + map_table[rdy_table[y]]);
4 // which bit in group

```

### 2.3.3 进程调度

为了简化系统设计，在移植部分没有修改原有系统的中断触发的底层部分，进程调度发生的情况如下：

一方面是进程级的上下文切换，具有高优先级的进程因为需要某种临界资源，主动请求挂起，让出处理器，此时发生进程调度，执行处于就绪状态的低优先级进程。

另一方面是中断级的上下文切换，具有高优先级的进程时间片用完，在时钟中断的处理程序中，内核发现高优先级的进程重新获得了执行条件，则从中断态直接切换到高优先级的进行执行。

我们使用一个调度链表管理所有任务，初始化后含有 idle 进程和 kernel shell 进程。调度的工作主要分为两部分：最高优先级进程的寻找和进程切换。最高优先级进程的寻找我们通过建立就绪进程表，使用数组管理实现。在进程切换时，先使用变量记录当前最高就绪任务的结点，然后调用相关函数进行切换，将寄存器中的信息更新为新的要执行进程的信息即可开始新的进程的执行。同时更新任务就绪表中的操作。

## 2.4 文件系统

本次工程实现的文件系统是 MBR 上的 FAT32。

### 2.4.1 初始化文件系统

硬盘分割主要有两种分割表，GPT 与 MBR，MBR 虽然比较旧而且最多只能由 4 个主要分区，但因为设计简单，本文件系统设计只支持 MBR 分割表。

Table 2.6: 标准 MBR 结构

地址	描述	长度(字节)
0	代码区	440 (最大 446)
440	选用磁盘标志	4
444	一般为空值; 0x0000	2
446	标准 MBR 分区表规划 (四个 16 byte 的主分区表入口)	64
510	0x55	1
511	0xAA	1

如上表，MBR 在硬盘中的第一个扇区，启动时必须先通过读 MBR 知道 FAT32 分区的位置，并且再读 FAT32 的 BIOS Parameter Block(BPB)，初始化 FAT32 分区的基本信息

Table 2.7: FAT32 分区的基本信息

变量名	意义
<code>total_sectors</code>	总共有多少扇区
<code>sectors_per_fat</code>	一个 FAT 中有多少个扇区
<code>first_data_sector</code>	第一个数据扇区位置
<code>total_data_sectors</code>	数据扇区数量
<code>total_data_clusters</code>	数据簇数量

### 2.4.2 FAT32 结构

FAT32 文件系统中，文件的分配方式是链表分配，文件拥有的所有磁盘块将在文件分配表中以链表的形式记录下来。如下简图所示：

Table 2.8: FAT32 文件系统磁盘组织简图

结构	保留区			文件分配表				数据区
	DBR (512 字节)	FSINFO (512 字节)	...	0	-	1	-	
				2	3	3	7	备份的 FAT 表
				4	x	5	x	
				6	x	7	x	存放实际数据

FAT32 的短文件名都是固定长度为 32 字节的元数据，记录文件的名称、扩展名、属性、大小、位置、时间等信息。

Table 2.9: FAT32 文件结构

偏移量	长度(字节)	描述
0x00	8	文件名
0x08	3	扩展名
0x0B	1	文件属性 本系统主要用到的有 0x10 代表子目录, 0x20 代表文件
0x0C	1	保留
0x0D	1	创建时间
0x0E	2	创建时间
0x10	2	创建日期
0x12	2	最近访问日期
0x14	2	FAT32 中第一个簇的两个高字节
0x16	2	最后更改时间
0x18	2	最后更改日期
0x1A	2	FAT32 中第一个簇的两个低字节。
0x1C	4	文件大小

### 2.4.3 FAT32 数据结构

C 语言中，用 `struct __attribute__((__packed__))` 定义结构可以使二进制数据按照定义顺序存储，搭配 `union` 定义结构，可以让读数据变得方便许多。

Listing 2.19: 短文件名元数据

```

1 struct __attribute__((__packed__)) dir_entry_attr {
2     u8 name[8];           /* Name */
3     u8 ext[3];           /* Extension */
4     u8 attr;              /* attribute bits */
5     u8 lcase;             /* Case for base and extension */
6     u8 ctime_cs;          /* Creation time, centiseconds (0-199) */
7     u16 ctime;             /* Creation time */
8     u16 cdate;             /* Creation date */
9     u16 adate;             /* Last access date */

```

```

10     u16 starthi;           /* Start cluster (Hight 16 bits) */
11     u16 time;             /* Last modify time */
12     u16 date;             /* Last modify date */
13     u16 startlow;          /* Start cluster (Low 16 bits) */
14     u32 size;              /* file size (in bytes) */
15 };
16
17 union dir_entry {
18     u8 data[32];
19     struct dir_entry_attr attr;
20 };

```

Listing 2.20: 文件结构

```

1 struct fat_file {
2     u8 path[256];
3     /* Current file pointer */
4     u32 loc;
5     /* Current directory entry position */
6     u32 dir_entry_pos;
7     u32 dir_entry_sector;
8     /* current directory entry */
9     union dir_entry entry;
10 };

```

Listing 2.21: FAT 分区的 BPB 结构

```

1 struct __attribute__((__packed__)) BPB_attr {
2     u8 BS_jmpBoot[3];      /* jump_code[3] */
3     u8 BS_OEMName[8];      /* oem_name[8] */
4     u16 BPB_BytsPerSec;   /* sector_size */
5     u8 BPB_SecPerClus;    /* sectors_per_cluster */
6     u16 BPB_RsvdSecCnt;   /* reserved_sectors */
7     u8 BPB_NumFATs;       /* number_of_copies_of_fat */
8     u16 BPB_RootEntCnt;   /* max_root_dir_entries */
9     u16 BPB_TotSec16;     /* num_of_small_sectors */
10    u8 BPB_Media;         /* media_descriptor */
11    u16 BPB_FATSz16;      /* sectors_per_fat */
12    u16 BPB_SecPerTrk;    /* sectors_per_track */
13    u16 BPB_NumHeads;     /* num_of_heads */
14    u32 BPB_HiddSec;      /* num_of_hidden_sectors */
15    u32 BPB_TotSec32;     /* num_of_sectors */
16    u32 BPB_FATSz32;      /* num_of_sectors_per_fat */
17    u16 BPB_ExtFlags;      /* flags */
18    u16 BPB_FSVer;        /* version */
19    u32 BPB_RootClus;     /* cluster_number_of_root_dir */
20    u16 BPB_FSInfo;        /* sector_number_of_fs_info */
21    u16 BPB_BkBootSec;     /* sector_number_of_backup_boot */
22    u8 BPB_Reserved[12];   /* reserved_data[12] */
23    u8 BS_DrvNum;          /* logical_drive_number */
24    u8 BS_Reserved1;        /* unused */
25    u8 BS_BootSig;          /* extended_signature */
26    u32 BS_VolID;          /* serial_number */
27    u8 BS_VolLab[11];       /* volume_name[11] */
28    u8 BS_FilSysType[8];    /* fat_name[8] */
29    u8 exec_code[420];       /* exec_code[420] */
30    u8 Signature_word[2];   /* boot_record_signature[2] */
31 };
32
33 union BPB_info {
34     u8 data[512];
35     struct BPB_attr attr;
36 };

```

Listing 2.22: FAT 信息

```

1 struct fs_info {
2     u32 base_addr;
3     u32 sectors_per_fat;
4     u32 total_sectors;
5     u32 total_data_clusters;
6     u32 total_data_sectors;
7     u32 first_data_sector;
8     union BPB_info BPB;
9     u8 fat_fs_info[SECTOR_SIZE];
10 };

```

## 2.4.4 FAT32 文件系统实现思路

### 查找文件

查找文件 (文件项, file entry) 是文件系统中最重要的功能, 因爲每个文件都必須要先被开啓, 知道该文件的实际位置后才能对文件做其他操作 (读、写等)。实现思路如下:

---

#### Algorithm 1 查找文件

---

```

1: function 查找文件 (路径)
2:     解析路径
3:     for 每一层目录, 由左往右遍历 do
4:         查找在当前目录中下一层目录的文件项
5:         继续循环, 将扇区指向下一层目录, 直到最後一层
6:     end for
7: end function

```

---



---

#### Algorithm 2 在目录中查找一个文件项

---

```

1: function 在目录中查找一个文件项 (目录)
2:     将扇区指向当前簇的第一个扇区
3:     for 簇中的 8 个扇区, 依序读入 do
4:         查找是否有文件名与欲查找的名称相同
5:         若有, 则离开循环, 以这个文件项作为返回值
6:         到 FAT 中查找下一个簇号, 如果没有, 则离开, 并返回查找失败
7:         将下一个簇号转换为扇区号
8:     end for
9: end function

```

---

### 读文件

对於一个已经开啓的文件, 可以从偏移量 0x14, 0x1A 知道文件实际所在的簇号, 然後进入文件数据区读取文件, 过程如下:

---

**Algorithm 3** 读文件

---

```

1: function 读文件 (目录)
2:   开啓文件
3:   取得文件所在簇号
4:   依照要读入的大小、开始位置, 计算开始、结束的簇、字节数
5:   for 每一个簇 do
6:     若还没到开始的簇, 则跳过
7:     如果在开始、结束的簇之中, 需要考虑字节偏移量
8:     一般的情况则读满 8 个扇区
9:     到 FAT 中查找下一个簇号, 如果没有, 则离开, 并返回失败
10:    end for
11:  end function

```

---

**写文件**

写文件与读文件基本相同, 只有最後一步不同, 如果到 FAT 中查找下一个簇号失败, 表示文件原本的空间不够用, 应该要在 FAT 中查找可用空间, 然後修改 FAT 的信息, 将文件原来的最後一个簇指向新获取的簇。

**读目录**

读目录与查找文件非常相近, 可以先利用开啓文件将目录开啓 (目录与文件的元数据并没有本质区别, 都是 32 字节的文件项), 然後以 32 字节为单位读该文件中的内容, 若是遇到合法的文件属性就输出。

**创建空文件**

创建空文件的方法很简单, 在目录中读到第一个空白 (或者是被擦除) 的 32 字节就能将其作为放新文件的位置, 然後更新所在目录的扇区。

**创建目录**

创建目录首先要创建一个空文件, 然後获取一个新的簇给这个文件, 在将. 与.. 写入簇中的前 0 到 31 、32 到 64 字节。

前 64 字节的内容直接写成固定的形式如下:

Listing 2.23: . 与..

```

1 /* entry of '.' */
2 const u8 dot[32] = {
3   0x2E, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
4   0x20, 0x20, 0x20, 0x10, 0x00, 0x00, 0x00,

```

```

5     0x00, 0x00, 0x00, 0x00, dc0, dc1, 0x00, 0x00,
6     0x00, 0x00, dc2, dc3, 0x00, 0x00, 0x00, 0x00};
7
8 /* entry of '..' */
9 const u8 double_dot[32] = {
10    0x2E, 0x2E, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
11    0x20, 0x20, 0x20, 0x10, 0x00, 0x00, 0x00, 0x00,
12    0x00, 0x00, 0x00, 0x00, dd0, dd1, 0x00, 0x00,
13    0x00, 0x00, dd2, dd3, 0x00, 0x00, 0x00, 0x00};

```

## 删除

删除分成两个部分，一是删除在目录中的文件项，二是删除文件实际内容，并且释放文件所占的空间，过程如下：

---

### Algorithm 4 删除文件

---

```

1: function 删除文件(文件)
2:   开啓文件
3:   取得文件所在簇号
4:   for 每一个簇 do
5:     到 FAT 中查找下一个簇号，如果没有，则离开。
6:     修改 FAT，把当前簇的下一个簇改成 0(代表空闲簇)
7:     进入下一个簇
8:   end for
9:   读文件所在的目录
10:  修改该文件的元数据为 0，第一个字节为 0xE5(代表被擦除)
11:  写回目录
12: end function

```

---

## 移动

文件的移动并不涉及文件数据的移动，只是修改在目录中的信息而已，如同链表一般，修改某个节点只需要  $O(1)$  的时间，过程如下：

1. 找到来源路径的文件项
2. 复制文件项 32 字节的数据
3. 在目标路径创建文件项，使用复制的数据
4. 删删除来源路径的文件项

## 相对路径

前面的创建目录已经实现了。与..，因此剩下的部分只有实现维护 `pwd`，`pwd` 是一个文件项的全局变量，在文件系统初始化时，多加了一行开啓根目录为 `pwd`。

### 2.4.5 源码结构

#### 文件概述

Table 2.10: 头文件概述

文件名	描述
<b>fat.h</b>	FAT32 底层文件系统的对象的定义，对象专用的方法、文件系统的接口，文件系统内部函数等函数的声明。
<b>usr.h</b>	实现提供用户操作接口，比如 <code>cat,ls,cd</code> 等函数的声明。
<b>utils.h</b>	一些工具函数的声明。
<b>include/fat.h</b>	将能提供给其他程序的系统调用声明於此，相较于 <code>fat.h</code> ， <code>include/fat.h</code> 对外是公用的。

Table 2.11: 源文件概述

文件名	描述
<b>fat.c</b>	FAT32 底层文件系统的对象专用的方法、文件系统的接口，文件系统内部函数等函数的定义。
<b>usr.c</b>	实现提供用户操作接口，比如 <code>cat,ls,cd</code> 等函数的定义。
<b>utils.c</b>	一些工具函数的定义。

#### 代码量统计

Table 2.12: 统计

行数	文件名
1370	<b>fat.c</b>
337	<b>usr.c</b>
95	<b>utils.c</b>
144	<b>fat.h</b>
7	<b>usr.h</b>
29	<b>utils.h</b>
219	<b>include/fat.h</b>
2201	全部

# Chapter 3

## 测试过程与结果

### 3.1 内存测试

内存测试的四条指令

- slabone
- slabck
- memc
- mem\_alltree

Listing 3.1: memc

```
1   for (i = 0; i < total; i++) adArr[i] = malloc(size);
2   for (i = 0; i < total; i++) free(adArr[i]);
3
4 slabck:
5     buddy_info();
6     for (i = 0; i<total; i++) addrArr[i] = kmalloc(size);
7     kernel_printf("Allocate %d blocks sized %d byte\n", total, size);
8     buddy_info();
9     for (i = 0; i<total; i++) kfree(addrArr[i]);
10    kernel_printf("Deallocate\n");
11    buddy_info();
12
13 slabone:
14     buddy_info();
15     addrArr[1] = kmalloc(size);
16     kernel_printf("sized %d byte\n", size);
17
18 buddy_info();
```

`mem_alltree` 为验收后增加的模块，  
增加了在删除和插入时执行 `void BFS(mm_struct *mm)`，  
通过 `mm_struct` 找到当前进程块的红黑树的根节点，然后进行 BFS 访问。  
其它代码与 `memck` 相同，为了便于计算验证，把 10 个节点换成了 7 个。

### 3.1.1 slab 分配

申请了 1000 字节的内存块。观察到 Buddy 系统信息：共使用一页。这是正确的。此指令没有进行 free，在此基础上，我们计算批量分配和释放的页占用是否合理。

The screenshot shows a terminal window with the following text output:

```
PS :>slabone
Buddy-system :
    start page-frame number : 10e0
    end page-frame number : 8000
    (0) #: 2 frees
    (1) #: 1 frees
    (2) #: 1 frees
    (3) #: 0 frees
    (4) #: 6f1 frees
sized 1000 byte
Buddy-system :
    start page-frame number : 10e0
    end page-frame number : 8000
    (0) #: 1 frees
    (1) #: 1 frees
    (2) #: 1 frees
    (3) #: 0 frees
    (4) #: 6f1 frees
```

Figure 3.1: slab 分配

### 3.1.2 slab 批量分配以及释放

申请了 99 个 100 字节的内存块。计算得到应该占用 3 页。考虑到上一条指令占用了一页的一部分。此时 4 页的数字是比较合理的。

```
end page-frame number : 8888
(0) : 1 frees
(1) : 0 frees
(2) : 0 frees
(3) : 1 frees
(4) : 6f1 frees
Allocate 99 blocks sized 100 byte
Buddy-system :
start page-frame number : 18e8
end page-frame number : 8888
(0) : 1 frees
(1) : 0 frees
(2) : 1 frees
(3) : 0 frees
(4) : 6f1 frees
Deallocate
Buddy-system :
start page-frame number : 18e8
end page-frame number : 8888
(0) : 2 frees
(1) : 1 frees
(2) : 1 frees
(3) : 0 frees
(4) : 6f1 frees
Successfully slab check 0
PS:>
```

Figure 3.2: slab 批量分配以及释放

### 3.1.3 malloc/free 虚拟内存测试

因为 malloc 和 free 都会对 vma 对应的红黑树做操作。所以直接在红黑树节点插入和删除的时候输出节点颜色。通过 malloc 和 free 对应地址的计算，可以得到 vma 是否正常被分配，以及是否成功插入和删除红黑树节点。

从低地址到高地址申请了 10 个节点，同时从高地址到地地址释放了所有节点。可以观察到，第一个是 0，后面是 1。删除的时候，最后一个是 0，前面是 1。(未拍下最后一个 0，验收时助教已看到)。这是因为，在红黑树从 1 到 10 插入，从 10 到 1 删除时，就是这样的。但是在助教的解释下，我也觉得这样不能很好的反映 VMA 的结构。所以设计了下面的指令。

```
PS:>memc
one 0 insertone 1 insertone 1 insertone 1 insertone 1 insertone 1
sertone 1 insertone 1 insertone 1 insertone 1 deleteone 1 deleteone 1 deleteone
1 deleteone 1 deleteone 1 deleteone 1 deleteone 1 deleteone 1 deleteone 1 del
VMA check return with 0
```

Figure 3.3: malloc/free 虚拟内存测试

### 3.1.4 malloc/free 的增加测试

在助教的建议下，每次插入和删除都对红黑树进行广度优先遍历，同时输出树节点的颜色。此时可以通过观察红黑树是否符合定义来判断 vma 是否正常分配。

为了显示版本的区别，我在输出时输出的颜色与之前版本相反。

此为广度优先 (BFS) 遍历树的颜色输出。通过手工计算模拟红黑树的插入和删除，得到这是符合从低地址到高地址插入 7 个点，再从低地址到高地址删除 7 个点的红黑树结构的。故而 malloc 和 free 的结构没有问题，可以正常使用，红黑树结构也使得虚拟内存可以更好地支持以后对性能要求较高的拓展。

```
PS>mem_alltree
1:1
1:1 2:0
2:1 1:0 3:0
2:1 1:1 3:1 4:0
2:1 1:1 4:1 3:0 5:0
2:1 1:1 4:0 3:1 5:1 6:0
2:1 1:1 4:0 3:1 6:1 5:0 7:0
4:1 2:1 6:1 3:0 5:0 7:0
4:1 3:1 6:1 5:0 7:0
6:1 4:1 7:1 5:0
6:1 5:1 7:1
6:1 7:0
7:1PS>
```

Figure 3.4: malloc/free 的增加测试

## 3.2 进程管理: 多级反馈队列

### 3.2.1 进程创建与退出测试

`execk` 进程名

为了测试进程创建功能，我们提供了 `execk` 进程名测试指令。该指令的执行会在系统中新建一个进程，并且进程名即为输入指令后面的参数。测试结果照片如下。

```

PS :>execk test
Enter execk
exec_from_kernel(): begin
!is user
task_create(): begin
task_create(): end
exec_from_kernel(): end
execk return with 0
PS :>runprog(): begin
current_task: 2
...execute task (pid = 2)...
runprog(): end
pc_exit(): begin
wakeup_parent(): begin
wakeup_parent(): end
pc_exit(): add_exited()
pc_exit(): pid_free
print_task(): begin
[task list]:
name:idle pid:0 parent:0 priority:0
time_cnt:2 state:TASK_READY
name:kernel_shell pid:1 parent:0 priority:1
time_cnt:2 state:TASK_READY
name:test pid:2 parent:1 priority:10
time_cnt:3 state:TASK_EXITED
print_task(): end

```

Figure 3.5: 进程创建、退出测试

在上图中，我们可以看到在我们输入 `execk test` 命令后，第一个模块起始为 `exec_from_kernel():begin` 和 `exec_from_kernel():end`，在这之中完成了进程的创建，我们可以看到 `execk return with 0` 的返回值，这表明进程成功创建了。第二个模块起始为 `runprog()`，进入了内核进程的执行函数，我们可以看到 `excute task pid = 2` 说明进程现在正在执行，此后，我们可以看到一系列有关 `pc_exited()` 的操作，说明当前内核进程已经执行完毕，正在退出。在最后我设置打印出了所有进程链表中的信息，我们可以看到存在一个状态为 `TASK_EXITED` 的进程，表示进程已退出完毕。(在打印操作后会将进程移出所有进程链表)。

`execprio` 进程优先级

```

name:kernel_shell pid:1 parent:0 priority:10
time_cnt:0 state:TASK_RUNNING
name:Process8 pid:2 parent:1 priority:8
time_cnt:100 state:TASK_READY
print_task(): end
task_create(): end
exec_from_kernel(): end
PS:>runprog(): begin
current_task: 2

current_task: 2
current_task: 2
current_task: 2

current_task: 2 End.....
runprog(): end
pc_exit(): begin
wakeup_parent(): begin
wakeup_parent(): end
pc_exit(): find_next_task()
pc_exit(): remove...
pc_exit(): add_exited()
remove_exited(): begin
remove_exited(): end

PS:>
PS:>

```

Figure 3.6: 进程创建、退出测试

为了测试进程创建功能，我们提供了 `execprio` 进程优先级测试指令。该指令的执行会在系统中新建一个进程，并且进程名即为输入指令后面的参数。测试结果照片如上。我们可以看到在我们输入 `execprio 20` 命令后，第一个模块以结束 `exec_from_kernel():end`，在这之中完成了进程的创建。第二个模块起始为 `runprog()`，进入了内核进程的执行函数，我们可以看到 `current_task: 2` 的多次输出说明进程现在正在执行，此后，我们可以看到一系列有关 `pc_exited()` 的操作，说明当前内核进程已经执行完毕，正在退出。(这是第一个版本的图片，因而打印输出和前图有所不同)

### 3.2.2 进程调度测试

为了测试多级就绪队列，看出它的动态更新优先级时间片，我们连续创建两个优先级相同的进程，查看测试结果。测试结果图片如下。

```

current_task_id
time_cnt:97 state:TASK_READY
print_task(): end
task_create(): end
exec_from_kernel(): end
pc_exit(): begin
wakeup_parent(): begin
wakeup_parent(): end
pc_exit(): find_next_task()
pc_exit(): remove...
pc_exit(): add_exited()
remove_exited(): begin
remove_exited(): end

PS:>ps
print_task(): begin
task_list:
name:idle pid:8 parent:0 priority:8
time_cnt:99 state:TASK_READY
name:kernel_shell pid:1 parent:0 priority:10
name:Process4 pid:2 parent:1 priority:4
time_cnt:100 state:TASK_EXITED
name:Process4 pid:3 parent:1 priority:7
time_cnt:97 state:TASK_EXITED
print_task(): end
ps return with 0
PS:>

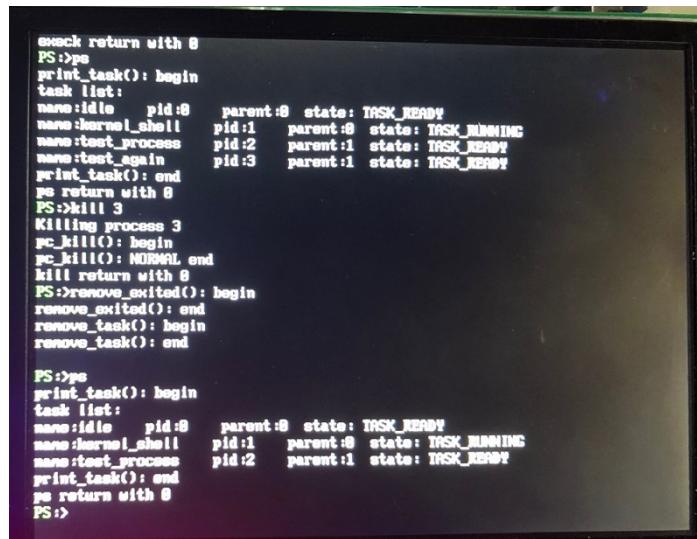
```

Figure 3.7: 进程调度测试

首先，我们使用 `execprior 4` 两次，以创建两个 name 为 `Process4` 的进程。在图中第一部分是第二次优先级为 4 的进程执行结束，我们立即输入 `ps` 来打印所有进程链表信息，可以看到两个初始优先级都为 4 的进程都已完成执行，并退出。但是两个进程现在呈现的优先级已经不相同，并且具有不同的时间片。说明在调度执行的过程中，优先级和时间片确在动态更新。

### 3.2.3 kill 进程测试

为了测试进程创建功能，我们提供了 `kill` 进程 `pid` 测试指令。该指令的执行会在系统中新建一个进程，并且进程名即为输入指令后面的参数。测试结果照片如下。



```

scheck return with 0
PS:>ps
print_task(): begin
task list:
name:idle pid:0 parent:0 state: TASK_READY
name:kernel_shell pid:1 parent:0 state: TASK_RUNNING
name:test_process pid:2 parent:1 state: TASK_READY
name:test_again pid:3 parent:1 state: TASK_READY
print_task(): end
ps return with 0
PS:>kill 3
Killing process 3
pc_kill(): begin
pc_kill(): NORMAL end
kill return with 0
PS:>remove_exited(): begin
remove_exited(): end
remove_task(): begin
remove_task(): end

PS:>ps
print_task(): begin
task list:
name:idle pid:0 parent:0 state: TASK_READY
name:kernel_shell pid:1 parent:0 state: TASK_RUNNING
name:test_process pid:2 parent:1 state: TASK_READY
print_task(): end
ps return with 0
PS:>

```

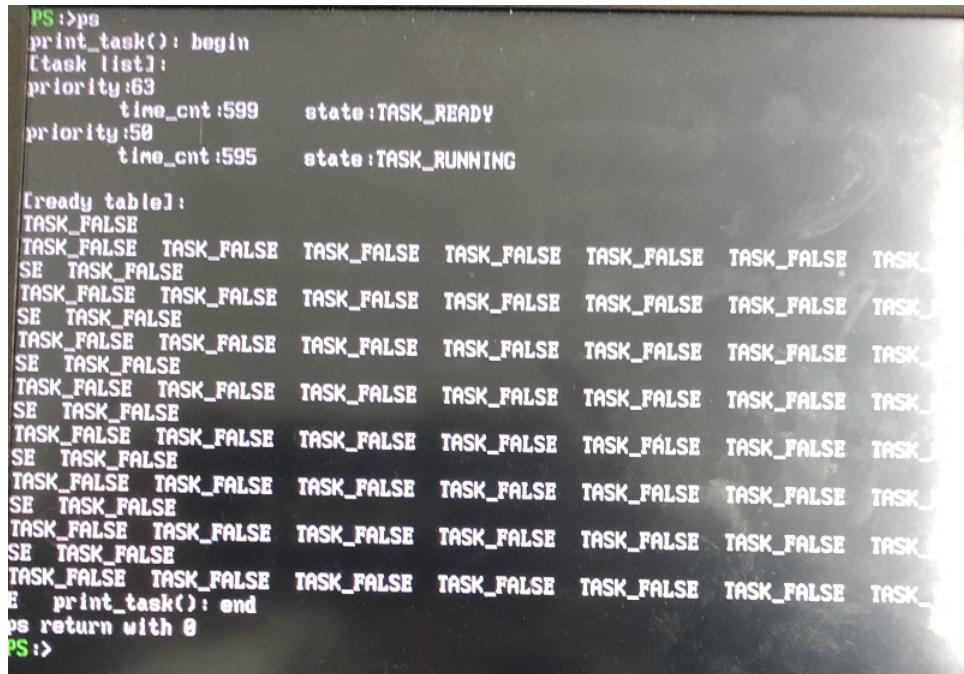
Figure 3.8: kill 进程测试

首先我们输入 `ps` 指令，以确保 `pid = 3` 的进程仍然在就绪队列中。如上图所示，此时该进程确实仍存在于系统之中，且处于就绪状态。然后我们输入 `kill 3` 指令，之后再次输入 `ps` 指令，从上图中可以看到，此时该进程已经不再存在于系统之中，这表明该进程已经成功被 `kill` 掉了。

### 3.3 进程管理：UC/OS II

#### 3.3.1 进程创建与退出测试

execk 进程名



```
PS:>ps
print_task(): begin
[task list]:
priority:63
    time_cnt:599    state:TASK_READY
priority:50
    time_cnt:595    state:TASK_RUNNING

[ready table]:
TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE TASK_FALSE
E print_task(): end
ps return with 0
PS:>
```

Figure 3.9: 进程创建测试

在初始化后，我们可以使用 `ps` 命令，展示当前所有链表信息和任务就绪表的全部内容。我们可以看到仅 IDLE 进程（优先级数值为 63）在任务就绪表中为 `TASKTURE`。而其他优先级都为 `TASK_FALSE`。

### 3.3.2 进程调度

```
print_task(): begin
[task list]:
priority:63
    time_cnt:599 state:TASK_READY
priority:58
    time_cnt:334 state:TASK_RUNNING
priority:53
    time_cnt:8 state:TASK_READY

[ready table]:
TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
SE_TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_TRUE TASK_FALSE TASK_
SE_TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_
E print_task(): end
PS return with 0
PS:>
```

Figure 3.10: 进程调度测试

首先，我们使用了 `exec test` 创建了一个进程，在系统中分配了一个空闲的优先级 53 给新进程。进程创建后，我们立即打印了当前所有链表信息，和任务就绪表的全部内容。可以看到处于就绪表中的进程 53 和 63 在表中的状态是 `TASKTURE`，如图 3.10。

```
print_task(): begin
[task list]:
priority:63
    time_cnt:599    state:TASK_READY
priority:58
    time_cnt:0     state:TASK_READY
priority:52
    time_cnt:600    state:TASK_EXITED

[ready table]:
TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE
SE_TASK_FALSE
TASK_FALSE TASK_TRUE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE
SE_TASK_FALSE
TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE TASK_FALSE
E print_task(): end

PS:>■
```

Figure 3.11: 进程调度测试

此时进程的优先级数值上高于 kernel 进程，但实际优先级意义上低于 kernel 进程，当 kernel 进程的时间片用完时，该进程被立即执行，再一次打印输出任务就绪表的全部内容，则显示该进程已不在就绪表中，如图 3.11。

## 3.4 文件系统

本文件系统基本实现了在 FAT32 上对数据增、删、查、改 4 项数据基本操作，并且支持绝对路径与相对路径，经过在 PC 上模拟测试以及实际测试，功能都正常运行。

Table 3.1: 实现的功能

功能	系统调用	用户接口
读文件	<code>fs_read</code>	<code>cat</code>
写文件	<code>fs_write</code>	<code>write</code>
读目录	<code>fs_read_dir</code>	<code>ls, cd</code>
删除文件	<code>fs_remove_file</code>	<code>rm</code>
新建文件	<code>fs_create</code>	<code>touch</code>
新建目录	<code>fs_mkdir</code>	<code>mkdir</code>

### 3.4.1 模拟测试环境

由於在文件系统底层涉及许多二进制操作，每一个操作如直接到实体机上测试，难以验证其正确性（比如先读、然後写，虽然在实体机上的输出都是正确，但也有可能是读写一起发生错误，如此会非常难以检查错误发生的位置），因此开发过程中，可以现在 PC 上用虚拟硬盘、二进制浏览器、文件浏览器以及自己模拟实现的用户界面测试。

#### 模拟 `ps.c`

在 ZJUNIX 中，用户接口实现在 `ps.c` 之中，而开发过程中可以在 PC 上模拟这个功能，实现如下：

Listing 3.2: FAT 信息

```

1 int main() {
2     init_fs();
3
4     char buf[512];
5     char arg0[128];
6     char arg1[128];
7     char arg2[128];
8
9     while (1) {
10        printf("%s\u2020\n", pwd.path);
11        gets(buf);
12        sscanf(buf, "%s%s%s", arg0, arg1, arg2);
13
14        if (strcmp(arg0, "cat") == 0) {
15            if (fs_cat(arg1) != 0) {
16                puts("fail");
17            }
18        } else if (strcmp(arg0, "touch") == 0) {
19            if (fs_touch(arg1) != 0) {

```

```

20         puts("fail");
21     }
22 } else if (strcmp(arg0, "rm") == 0) {
23     /* skip */
24 }
25 puts("");
26 }
```

另外还需要模拟硬件接口，具体需要如下几个函数：

Listing 3.3: FAT 信息

```

1 void * kernel_memset (void * ptr, int value, size_t num);
2 void* kmalloc (size_t size);
3 int kernel_printf(const char * format, ...);
4 void kfree(void* ptr);
5 int read_block(unsigned char *buf, unsigned int addr, unsigned int count);
6 int write_block(unsigned char *buf, unsigned int addr, unsigned int count);
```

这些函数的实现都可以直接调用 `stdio.h` 以及 `stdlib.h`。

## 虚拟硬盘

虚拟硬盘可以用 Windows 的管理工具建立。

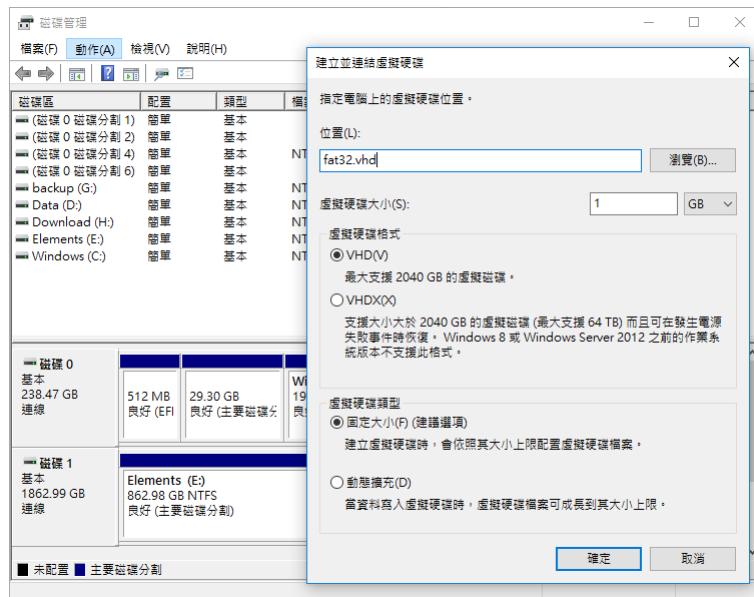


Figure 3.12: FAT32 文件系统磁盘组织简图

## ls, cd, mkdir



```

hanyi@DESKTOP-S1FB9J8:/mnt/c/Users/chanh/Desktop/filesystem/src$ ./main
$ 
ls / 

$ 
mkdir /dir

$ 
cd /dir

/dir $ 
mkdir ..../dir2

/dir $ 
mkdir dir3

/dir $ 
^C
hanyi@DESKTOP-S1FB9J8:/mnt/c/Users/chanh/Desktop/filesystem/src$ 

```

Figure 3.13: ls, cd, mkdir 测试

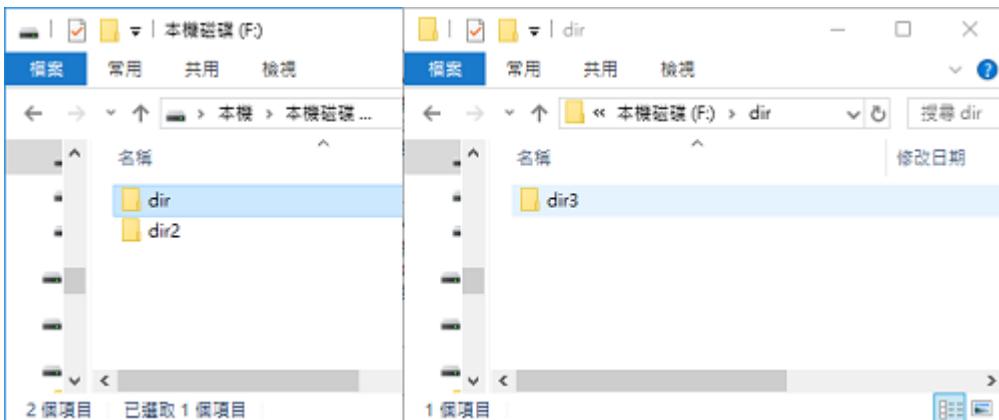
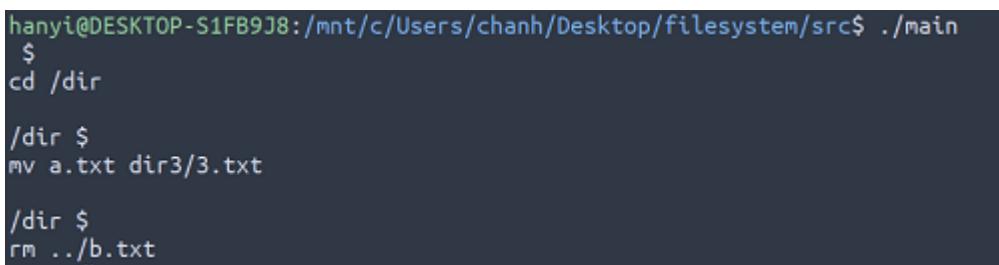


Figure 3.14: ls, cd, mkdir 结果

## rm, mv



```

hanyi@DESKTOP-S1FB9J8:/mnt/c/Users/chanh/Desktop/filesystem/src$ ./main
$ 
cd /dir

/dir $ 
mv a.txt dir3/3.txt

/dir $ 
rm ..../b.txt

```

Figure 3.15: rm, mv 测试

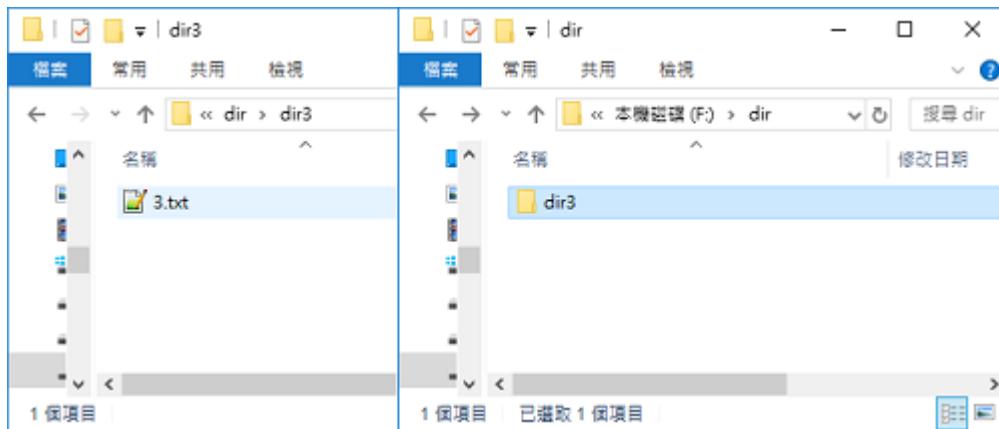


Figure 3.16: rm, mv 结果

### touch, write

```
hanyi@DESKTOP-S1FB9J8:/mnt/c/Users/chanh/Desktop/filesystem/src$ ./main
$ cd /dir
/directory $ touch a.txt
/directory $ touch /b.txt
/directory $ touch ../dir2/c.txt
/directory $ write a.txt
/directory $ write /b.txt
/directory $ write ../dir2/c.txt
/directory $
```

Figure 3.17: touch, write 测试

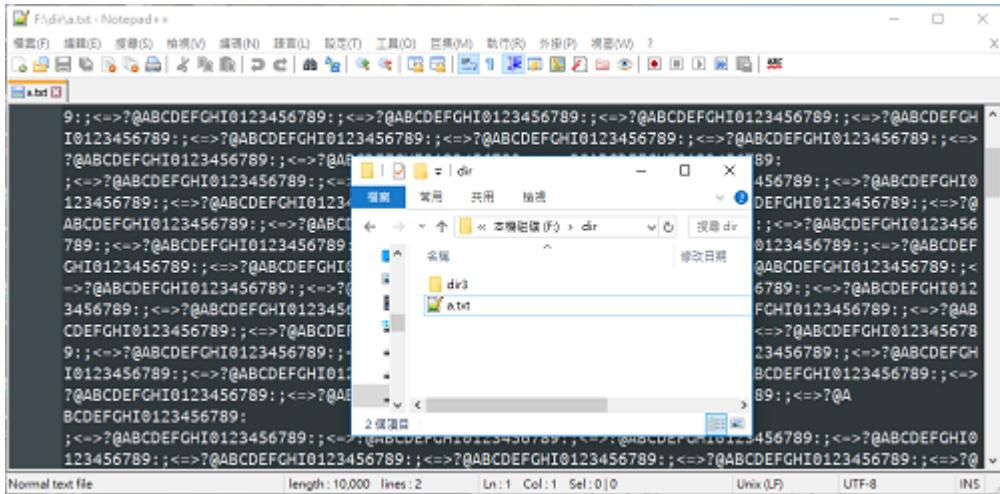


Figure 3.18: touch, write 结果

cat

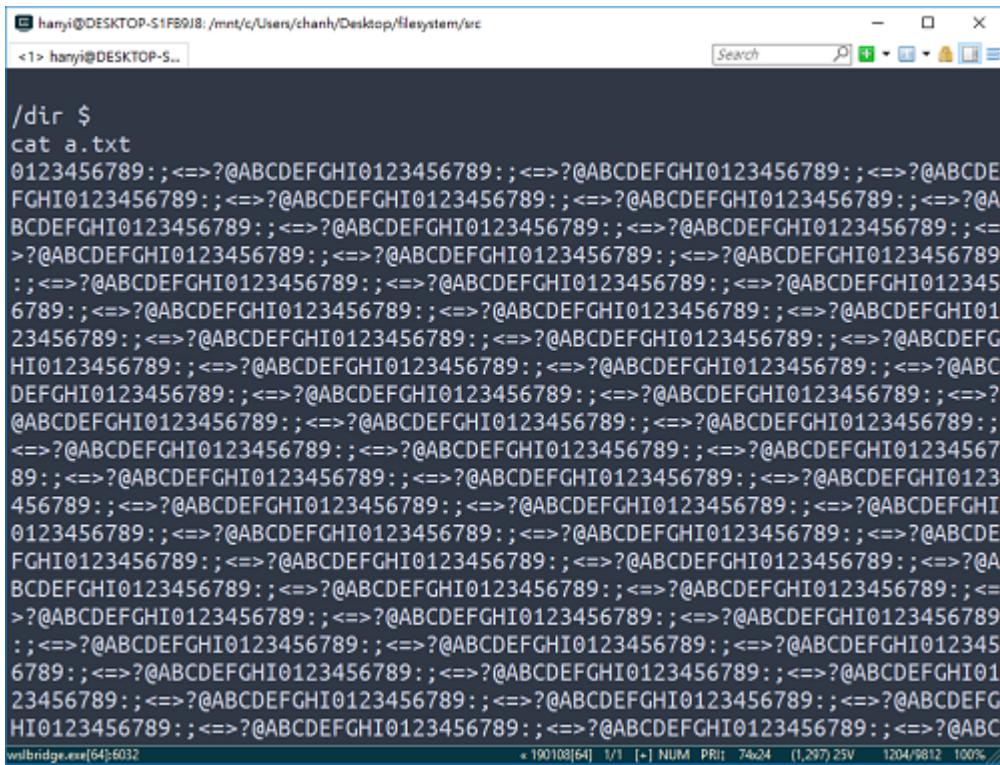


Figure 3.19: cat 测试

### 3.4.2 实际测试

ls, cd, mkdir

```
PS:>ls /
KERNEL.BIN
PS:>mkdir /test
PS:>ls /
KERNEL.BIN  TEST/
PS:>cd test
PS:>ls .
./ ..
PS:>touch b.txt
PS:>ls .
./ ..  B.TXT
PS:>ls ..
KERNEL.BIN  TEST/  A.TXT
```

Figure 3.20: ls, cd, mkdir

```
PS:>ls /
KERNEL.BIN  SYSCALL.BIN  SEC.BIN  TDIR/  B.TXT  C.TXT
PS:>ls /abcd
File /abcd open failed
PS:>ls /tdir
./ ..
PS:>cd /tdir
PS:>ls .
./ ..
PS:>ls ..
KERNEL.BIN  SYSCALL.BIN  SEC.BIN  TDIR/  B.TXT  C.TXT
PS:>ls abc
File abc open failed
PS:>mkdir abc
PS:>ls .
./ ..  ABC/
PS:>ls abc
./ ..
PS:>
PS:>
```

Figure 3.21: ls, cd, mkdir 测试

touch

```
PS:>touch /a.txt  
PS:>ls /  
KERNEL.BIN  TEST/  A.TXT
```

Figure 3.22: touch 测试

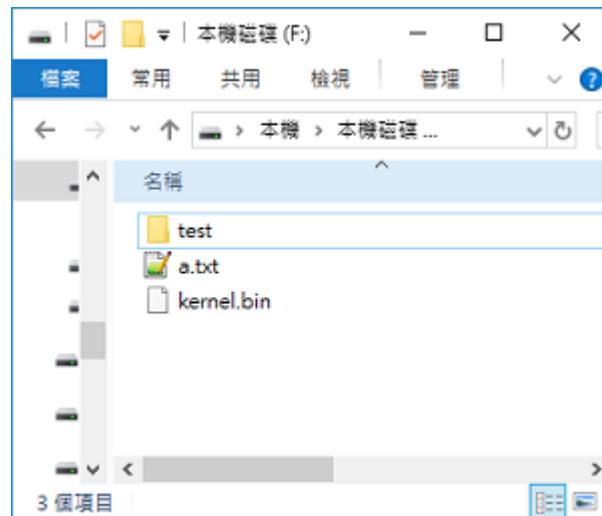


Figure 3.23: touch 结果

cat

Figure 3.24: cat 测试

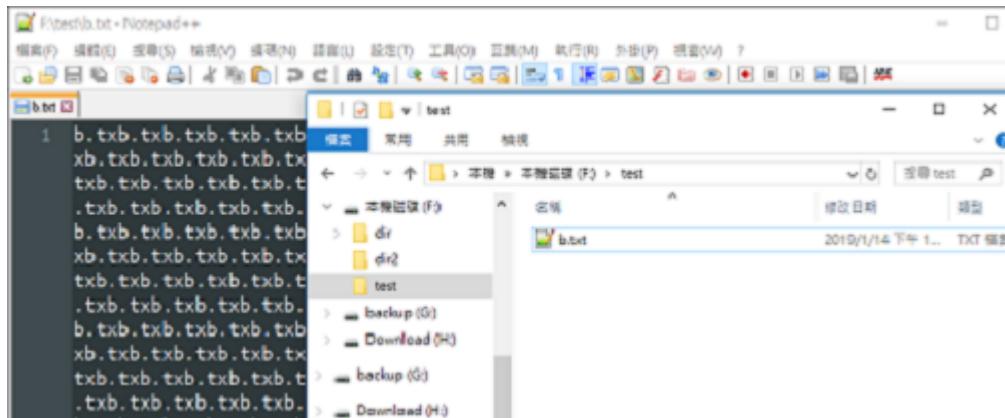


Figure 3.25: cat 结果

**write**

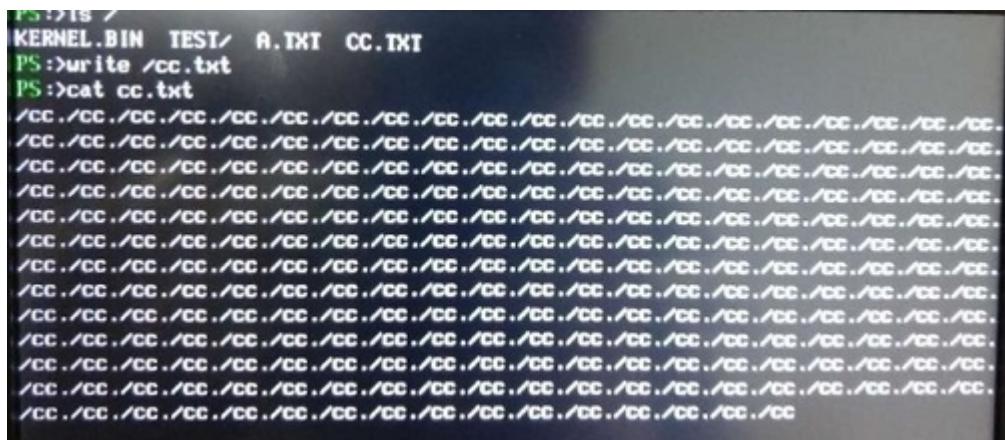


Figure 3.26: write 测试

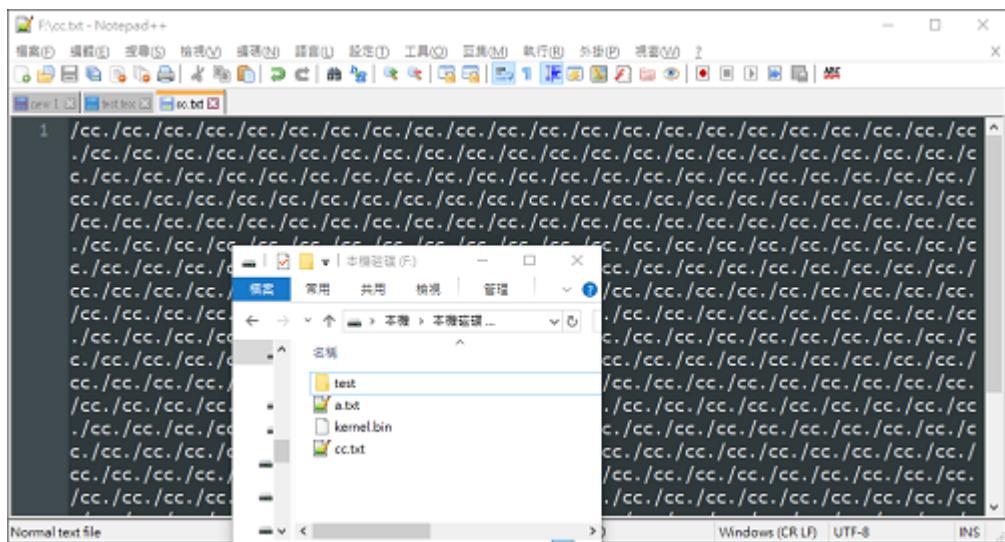


Figure 3.27: write 结果

mv

```
PS:>mv b.txt ../cc.txt  
PS:>cd ..  
PS:>  
PS:>cat cc.txt
```

Figure 3.28: mv 测试

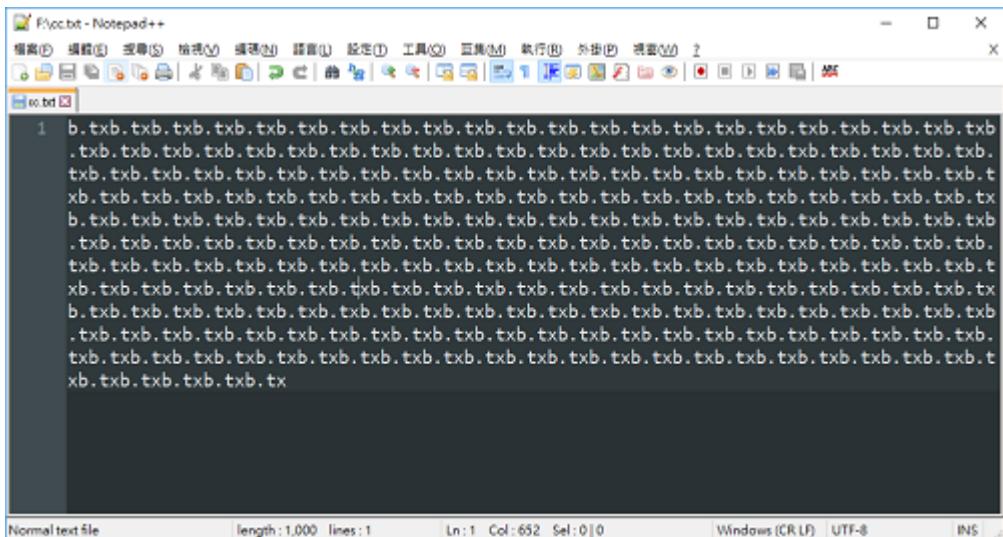


Figure 3.29: mv 结果

rm

```
PS:>rm cc.txt  
PS:>ls .  
File . open failed  
PS:>  
PS:>ls /  
KERNEL.BIN TEST/ A.TXT
```

Figure 3.30: rm 测试

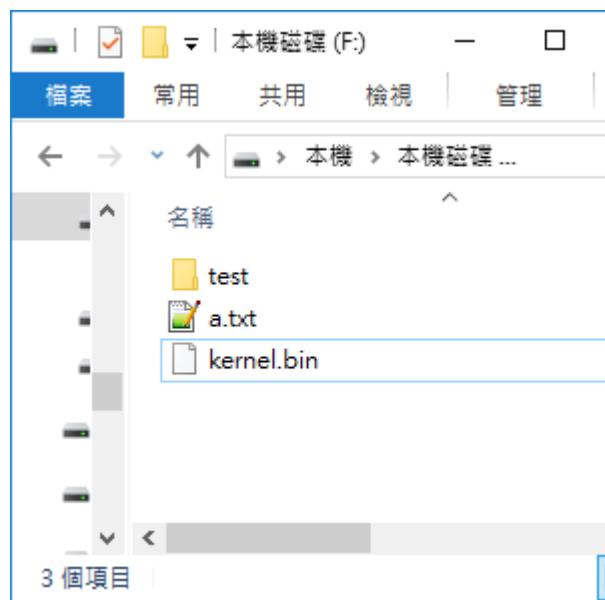


Figure 3.31: rm 結果

# Chapter 4

## 讨论及体会

我实现的是内存部分。一开始看助教的代码，觉得内存部分已经实现的比较好。尤其是虚拟内存部分，我觉得我在助教的基础上拓展一下，发挥一下，站在巨人的肩膀上总是好行事的。

所以把目标定在了页置换算法和内存池上。写好内存池之后才细看助教的代码，然后发现助教的 TLB 是有问题的。问题具体体现在无法真正映射虚拟内存。那内存池和页置换之前就要修复助教的 BUG。

在修复助教的 BUG 的时候，发现一个底层的 BUG，当系统处理异常的时候，他有时会跳转到 `TLB_refilling` 异常。我觉得是硬件底层的问题。询问了另一个硬件小组的同学，他也这么觉得。

事情变得神秘了起来。我就把目标定在了使用红黑树管理 `mm_struct` 中的 `vma_struct` 和绕过 TLB，写一个直接对 vma 层面操作的 `malloc` 和 `free` 函数。之所以写 `malloc/free` 也是为了验收的时候可以展示。同时这个 `malloc/free` 也给了之后版本的可拓展性。

说到可拓展性，VMA 里面的合并接口，我都留了出来，假如之后可以再写页置换的话，还可以再写一写。

其他方面，比如像时间安排啥的。我都没什么问题，慢慢写的，也不紧张。但是没有地方可以调试是一个问题。实验室总是不开门，写到一定程度是一定要去实验室测试代码的。所以还是希望之后实验室能多开门。

最后，这个操作系统还是极大的提高了我的视野，我才知道硬件小组的人都这么强。可能只有强者才能学硬件。

李仁钟

实现一个操作系统工作量非常大，用一个学期来完成这个项目虽然足够，但依照学生的惯性一般都要拖到最後才写，我也不例外，我是到了验收前一个月才开始大部分工作，尽管开学初前几周助教就已经在周五下午的课讲解完 FAT32 与 EXT2 的实现。

实现的过程其实并没有遇到很多问题，主要花的时间都在 PC 上调试或上网查资料，也可能是因为我做的是文件系统，开始开发之前我第一个实现的功能是 PC 上的模拟器，如此一来就不需要经常去实验室调试，非常方便（事实上我只去了 3 次，验证的结果跟在 PC 上模拟完全相同）。

然而我的队友以及其他同学们似乎不是如此，他们到最后一周工作量非常大、看他们好像很痛苦的样子，虽然没有控制好进度是我们的问题，但如果老师能用其他方式更精确的检查进度应该会好一些。

原本我并不知道我选的操作系统班是教改班，然而当我知道期末要做这个作业时已经没办法换了（因为其他课时间都跟着排好），因此只好努力完成要求，虽然我自己觉得我做的工作没有很突出，但经由自己实现操作系统的过称确实使我对操作系统的原理有非常深刻的理解，在复习期末考试的期间，看到课件上的内容都可以联想到内核中的代码，我觉得这是我在这个班最大的收获。毕业後大部分课上讲过的内容可能最终都会忘记，然而自己做过操作系统会一直留在我的记忆中。

陈翰逸

# Chapter 5

## 组内成员分工

本次课程设计分工情况安排如下

Table 5.1: 分工情况

角色	姓名	分工
组长	陈翰逸	文件系统
组员	李仁钟	内存部分
组员	萧芷晴	进程管理 多级反馈进程调度 uc/osII 任务调度
	共同参与	中期报告的参与部分的撰写 定期的组内讨论与交流

# Chapter 6

## 附注

# Chapter 7

## 参考文献

???