基于 Bochs 的操作系统内核实现

[fleuria] 2012

简介

Bochs 简介

Bochs(读音Box)是一个开源的模拟器(Emulator)，它可以完全模拟x86/x64的硬件以及一些外围设备。与VirtualBox / VMware等虚拟机(Virtual Machine)产品不同，它的设计目标在于模拟一台真正的硬件，并不追求执行速度的高效，而追求模拟环境的真实，同时带有强大的调试功能，比如观察寄存器、对实地址/虚拟地址下断点、装载符号表等等。对于操作系统内核的开发者而言，是一只不可多得的强力工具，通过简单的设置，即可大大地降低内核开发与调试的困难。

作为开源软件，我们可以很方便地获取它：

- 主页：http://bochs.sourceforge.net/getcurrent.html
- 参考文档：http://wiki.osdev.org/Bochs

安装

在Ubuntu操作系统下，可以通过apt-get来安装：

```
sudo apt-get install bochs
```

若要利用Bochs的调试功能，则需要自己编译安装：

```
wget http://sourceforge.net/projects/bochs/files/bochs/2.5.1/bochs-2.5.1.tar.gz/download -O bochs.tar.gz
tar -xvfz bochs.tar.gz
cd bochs-2.5.1
./configure --enable-debugger --enable-debugger-gui --enable-disasm --with-x --with-term
make
sudo cp ./bochs /usr/bin/bochs-dbg
```

配置

Bochs 提供了许多配置选项，在项目中，我们可以灵活的选择/设置自己所需的功能，比如模拟器的内存大小、软/硬盘镜像以及引导方式等等。而这些配置选项都统一在一个.bochsrc文件中，样例如下：

.bochsrc:

```
# BIOS与VGA镜像
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/bochs/VGABIOS-lgpl-latest
# 内存大小
megs: 128
# 软盘镜像
floppya: 1_44=bin/kernel.images, status=inserted
# 硬盘镜像
ata0-master: type=disk, path="bin/rootfs.images", mode=flat, cylinders=2, heads=16, spt=63
# 引导方式(软盘)
boot: a
# 日志输出
log: .bochsout
panic: action=ask
error: action=report
info: action=report
debug: action=ignore
# 杂项
vga_update_interval: 300000
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
mouse: enabled=0
private_colormap: enabled=0
fullscreen: enabled=0
screenmode: name="sample"
keyboard_mapping: enabled=0, map=
```

```
keyboard_type: at
# 符号表(调试用)
debug_symbols: file=main.sym
# 键盘类型
keyboard_type: at
```

在启动bochs时，使用命令：

```
bochs -q -f .bochsrc
```

内置调试器

bochs内置了强大且方便的调试功能。主要命令如下：

- `b`，`vb`，`lb` 分别为物理地址、虚拟地址、逻辑地址设置断点
- `c` 持续执行，直到遇到断点或者错误
- `n` 下一步执行
- `step` 单步执行
- `r` 显示当前寄存器的值
- `sreg` 显示当前的段寄存器的值
- `info gdt`， `info idt`， `info tss`， `info tab` 分别显示当前的GDT、IDT、TSS、页表信息
- `print-stack` 打印当前栈顶的值
- `help` 显示帮助

fleurix 简介

fleurix 是一个简单的单内核(Monolithic Kernel)操作系统实现，它的功能精简但不失完整，代码简短(七千行C，二百多行汇编)且易于阅读，可作为操作系统课程教学中的样例系统。在设计时选择采用了类UNIX的系统调用接口，因此在开发过程中可以获取丰富的文档供参考，也可以作为学习UNIX操作系统实现的一个参考材料。

fleurix 在编写时尽量使用最简单的方案。它假定CPU为单核心、内存固定为128mb，前者可以简化内核同步机制的实现，后者可以简化内存管理的实现。从技术角度来看，这些假定并不合理，但可以有效地降低刚开始开发时的复杂度。待开发进入轨道，也不难回头解决。 此外，你也可以在源码中发现许多穷举算法——在数据量较小的前提下，穷举并不是太糟的解决方案。

- 主页： http://github.com/fleurer/fleurix
- 开发环境： Ubuntu
- 平台：x86
- 依赖： `bochs`， `rake`， `binutils`， `nasm`， `mkfs.minix`

特性

- minix v1的文件系统。原理简单，而且可以利用linux下的mkfs.minix，fsck.minix等工具。
- `fork()` / `exec()` / `exit()` 等系统。可执行文件格式为a.out，实现了写时复制与请求调页。
- 信号。
- 一个纯分页的内存管理系统，每个进程4gb的地址空间，共享128mb的内核地址空间。至少比Linux0.11中的段页式内存管理方式更加灵活。
- 一个简单的 `kmalloc()` 。
- 一个简单的终端。

编译运行

```
git clone git@github.com:Fleurer/fleurix.git
cd fleurix
rake
```

调试

```
# 需要自行编译安装带调试功能的bochs-dbg，安装步骤参见前文。
cd fleurix
rake debug
```

设计与实现

编译与链接

fleurix的内核镜像为裸的二进制文件，结构大体如下：

(补图)

Rakefile

对于项目中的一些日常性质操作，比如：

- 编译bootloader，生成引导镜像
- 编译并链接内核，生成内核镜像
- 生成符号表
- 初始化根文件系统，生成硬盘镜像
- 编译整个项目，并运行bochs进行调试

它们需要的命令比较多，而且存在依赖关系，此任务必须在确保彼任务执行完毕并成功之后才可以执行。对此，比较通用的解决方案便是make，它可以自动分析任务之间的依赖关系再依次执行，从而简化日常操作的脚本编写。但是make的语法比较晦涩，对于没有任何基础的初学者来讲，上手起来并不容易。为此fleurix选择了rake，它相当于make的ruby实现，可以使用ruby语言的语法来编写make脚本，好处是易于上手，而代价是不如make的语法简洁。

fleurix中常用的rake命令有：

- `rake` 或者 `rake bochs` ，构建整个项目并运行bochs
- `rake build` ，构建整个项目到 `/bin` 目录
- `rake debug` ，构建整个项目并运行bochs的调试器
- `rake clean` ，将 `/bin` 目录清空
- `rake nm` ，生成符号表
- `rake todo` ，列出代码中遗留的待解决事项
- `rake werr` ，打开gcc的-Werror选项进行编译，方便排除代码中的warning
- `rake rootfs` ，构建根文件系统
- `rake fsck` ，对根文件系统执行fsck，检查结构是否正确

ldscript

内核开发与应用程序开发的不同之一便在于开发者需要对二进制镜像的结构有所了解，在必要时必须进行一些重定位。比如内核的入口为0x100000，为此需要将入口的代码(`bin/entry.o`)安排到内核镜像的最前方。而这便可以通过ldscript来完成，如下：

tool/main.ld:

```
ENTRY(kmain)
SECTIONS {
    __bios__ = 0xa0000; # 绑定BIOS保留内存的地址到__bios__
    vgamem = 0xb8000; # 绑定vga缓冲区的地址到符号vgamem
    .text 0x100000 : { # 内核二进制镜像中的.text段(Section)，从0x100000开始
        __kbegin__ = .; # 内核镜像的开始地址
        __code__ = .;
        bin/entry.o(.text) bin/main.o(.text) *(.text); # 将bin/entry.o中的.text段安排到内核镜像的最前方
        . = ALIGN(4096); # .text段按4kb对齐
    }
    .data : {
        __data__ = .;
        *(.rodata);
        *(.data);
        . = ALIGN(4096);
    }
    .bss : {
```

```
        __bss__ = .;
        *(.bss);
        . = ALIGN(4096);
    }
    __kend__ = .; # 内核镜像的结束地址
}
```

Rakefile中的相关命令如下，在链接时选择tool/main.ld作为链接脚本：

```
sh "ld #{ofiles * ' '} -o bin/main.elf -e c -T tool/main.ld"
```

bootloader

bootloader是一段小程序，负责执行一些初始化操作，并将内核装载到内存，是内核执行的入口，也是内核开发的第一步。

x86体系结构的CPU在设计中为了保持向前兼容，在PC机电源打开之后，x86平台的CPU会先进入实模式(Real Mode)，并从0xFFF0开始执行BIOS的一些初始化操作。随后，BIOS将依次检测启动设备(软盘或者硬盘)的第一个扇区(512字节)，如果它的第510字节处的值为0xAA55，则认为它是一个引导扇区，将它装载到物理地址0x7C00，并跳转到0x7C00处开始执行。这便是bootloader的入口地址。

实模式中默认可用的地址总线为20位，可以寻址1mb的内存，但寄存器只有16位。为此英特尔公司做出的设计是，在实模式的寻址模式中，令物理地址为16位段寄存器左移4位加16位逻辑地址的偏移所得的20位地址。若要访问1mb之后的内存，则必须开启A20 Line开关，将32位地址总线打开，并进入保护模式(Protect Mode)才可以。

在实模式中，0~4kb为中断向量表保留，640kb~1mb为显存与BIOS保留，实际可用的内存只有636kb。考虑到日后内核镜像的体积有超过1mb的可能，所以将其装载到物理地址1mb(0x100000)之后连续的一块内存中可能会更好。但实模式中并不可以访问1mb以后的内存，若要装载内核到物理地址1mb，一个解决方案便是在实模式中暂时将其装载到一个临时位置，待进入保护模式之后再移动它。

由上总结可知，bootloader所需要做的工作便依次为：

- 装载内核镜像到一个临时的地址；
- 进入保护模式；
- 移动内核镜像；
- 跳转到内核的入口。

相关代码可见于 `src/boot/boot.S` 。

保护模式与GDT

x86的保护模式是对段寻址的增强，除去可以访问32位的地址空间(4Gb)之外，更有了对保护级别(即ring0/ring1/ring2/ring3)的划分、对内存区域的限制、以及访问控制。为实现这些功能，x86的做法是引入了GDT(Global Descriptor Table)。将每个段(Segments)的属性对应为GDT中的一项段描述符(Segment Descriptor)，并通过段寄存器(如 `cs` 、 `ds` 、 `ss`)中指明的选择符进行选择。GDT是驻留于内存中的一个表，通过 `lgdt` 指令装载到CPU。

在bootloader中进入保护模式的目的仅仅是为了访问1mb以后的内存，而且bootloader在完成引导系统之后即被视为废弃，因此这里的GDT只能做临时使用。其中含有两个段描述符，它们的选择符分别为0x08与0x10，分别用于内核态代码与数据的访问。

进入内核之后，fleurix会在 `gdt_init()` 中重新设置GDT(见 `scr/kern/seg.c`)。

fleurix是一个纯分页的系统，虽然并不需要段式的内存管理，但依然需要一个GDT，只采用它的内存保护功能，而绕过它的分段功能。在fleurix最终的GDT中，将只保留四个段描述符，它们的内存区域皆为0~4Gb，选择符分别为 `KERN_CS` 、 `KERN_DS` 、 `USER_CS` 与 `USER_DS` ——前两者的权限为ring0，用于内核态代码与数据的访问；后两者的权限为ring3，分别用于用户态代码与数据的访问——从而实现内核态与用户态的分离，使后者受到更多限制，将系统“保护”起来。

需要留意的是，除四个段描述符之外，fleurix的GDT中也带有一个TSS描述符，其选择符为(`_TSS`)。英特尔公司引入TSS机制的动机为实现硬件的任务切换，每个任务拥有一个TSS，在进程切换时，将当前进程的所有上下文保存在TSS中。比起软件的任务切换，硬件任务切换的开销相对比较大，而且没有调试与优化的余地。fleurix采用了软件的任务切换机制，并无用到TSS的任务切换功能，但依然保留一个TSS是为了保存中断处理时ss0与esp0两个寄存器的值，在CPU通过中断门或者自陷门转移控制权时，据此获取内核栈的位置。

装载内核

在早期开发中为方便装载，fleurix内核的二进制镜像被放置在软盘镜像中，自第二个扇区开始，大约为50kb。

在实模式中，可以通过调用13h号中断来读取软盘扇区，将内核镜像临时读取到物理地址0x10000处。在设置临时的GDT之后，通过jmp指令进入保护模式，并将内核拷贝至物理地址0x100000(1mb)处。

内核初始化

待bootloader执行完毕之后，内核会首先进入 `kmain()` (见 `src/kern/main.c`)，执行一些初始化操作。这些操作依次为：

- 清理屏幕(`cls()` ，见 `src/chr/vga.c`)，初始化 `puts()` 与 `printk()` 等函数供调试与输出使用。

- 重新设置GDT(`gdt_init()` , 见 `src/kern/seg.c`)。
- 初始化IDT(`idt_init()` , 见 `src/kern/trap.c`)。
- 初始化内存管理(`mm_init()` , 见 `src/kern/pm.c`)。
- 初始化进程0(`proc0_init()` , 见 `src/kern/proc.c`)。
- 初始化高速缓冲(`buf_init()` , 见 `src/blk/buf.c`)。
- 初始化tty(`tty_init()` , 见 `src/chr/tty.c`)。
- 初始化硬盘驱动(`hd_init()` , 见 `src/blk/hd.c`)。
- 初始化内核定时器(`timer_init()` , 见 `src/kern/timer.c`)
- 初始化键盘驱动(`keybd_init()` , 见 `src/chr/keybd.c`)。
- 开启中断(`sti()` , 见 `src/inc/asm.h`)。
- 初始化进程1(`kspawn(&init)`), 通过 `do_exec()` (见 `src/kern/exec.c`)即进入用户态。

中断处理

中断是CPU中打断当前程序的控制流以处理外部事件、报告错误或者处理异常的一种机制。若详细分类，仍可将中断分为三种：

- 中断(Interrupt)：由CPU外部产生，CPU处于被动的位置，多用于CPU与外部设备的交互。
- 自陷(Trap)：在CPU本身的执行过程中产生。一般由专门的指令有意产生，比如 `int $0x80` ，因此又被称作”软件中断”。
- 异常(Exception)：因CPU执行某指令失败而产生，如除0、缺页等等。与自陷的不同在于，CPU会在处理例程结束之后重新执行产生异常的指令。

(注：即，自陷发生时，入栈的返回地址为下一条指令的地址；而异常发生时，入栈的返回地址为当前指令的地址)

在保护模式的x86平台中，中断通过中断门(Interrupt Gate)转移控制权，自陷与异常通过自陷门(Trap Gate)转移控制权。

每个中断对应一个中断号，系统开发者可以将自己的中断处理例程绑定到相应的中断号，表示中断号与中断处理例程之间映射关系的结构被称作中断向量表(Interupt Vector Table)。在保护模式中的x86平台，这一结构的实现为IDT(Interrupt Descriptor Table)。与GDT类似，IDT也是一个驻留于内存中的结构，通过 `lidt` 指令装载到CPU。每个中断处理例程对应一个门描述符(Gate Descriptor)。在fleurix中初始化IDT的代码位于 `idt_init()` (见 `src/trap.c`)。

在中断发生时，CPU会先执行一些权限检查，若正常，则依据特权级别从TSS中取出相应的ss与esp切换栈到内核栈，并将当前的eflags、cs、eip寄存器压栈(某些中断还会额外压一个error code入栈)，随后依据门描述符中指定的段选择符(Segment Selector)与目标地址跳转到中断处理例程。 保存当前程序的上下文则属于中断处理例程的工作。在fleurix中，保存中断上下文的操作由 `_hwint_common_stub` (见 `src/kern/entry.S.rb`)负责执行，它会将中断上下文保存到栈上的 `struct trap` 结构(见 `src/inc/idt.h`)。

在这里有三个地方值得留意：

- 只有部分中断会压入error code，这会导致栈结构的不一致。为了简化中断处理例程的接口，fleurix采用的方法是通过代码生成，在中断处理例程之初为不带有error code的中断统一压一个双字入栈，值为0，占据error code在 `struct trap` 中的位置。并将中断调用号压栈，以方便程序的编写与调试。
- fleurix中的中断处理例程都经过汇编例程 `_hwint_common_stub` ，它在保存中断上下文之后，会调用 `hwint_common()` (见 `src/kern/trap.c`)函数。 `hwint_common()` 函数将依据中断号，再查询 `hwint_routines` 数组找到并调用相应的处理例程。
- 中断的发生往往就意味着CPU特权级别的转换，因此，可以将陷入(或称”软件中断”)作为用户态进入内核态的入口，从而实现系统调用。在fleurix中系统调用对应的中断号为0x80，与linux相同。

I/O

外部设备一般分为机械部分与电路部分。电路部分又被称作控制器(Controller)或者适配器(Adapter)，负责设备的逻辑与接口。

CPU一般都是通过寄存器的形式来访问外部设备。外设的寄存器通常包括控制寄存器、状态寄存器与数据寄存器三类，分别用于发送命令、读取状态、读写数据。按照访问外设的寄存器的方式，CPU又主要分为两类：

- 将外设寄存器与内存统一编址(Memory-Mapped)：访问寄存器即一般的内存读写，没有专门用于I/O的指令。
- 将外设寄存器独立编址(I/O-Mapped)：每个寄存器对应一个端口号(port)，通过专门读/写的指令访问外设的寄存器，如 `in` 与 `out` 指令。

x86是后者：采用独立编址的方式，外设寄存器即I/O端口，并通过in、out等汇编指令进行读写。

在fleurix中，提供了如下的几个函数来读写端口：

- `inb()` 与 `outb()` ：按字节读写端口
- `inw()` 与 `outw()` ：按字读写端口

- `insb()` 与 `outsb()` ：对某端口读/写一个字节序列
- `insl()` 与 `outsl()` ：对某端口读/写一个双字的序列

以上函数都是对汇编指令的简单包装，定义于 `src/inc/asm.h` 。

留意它们的源代码，可以注意到它们都会在最后调用一个 `io_delay()` 函数。这是因为对于一些老式总线的外部设备，读写I/O端口的速度若过快就容易出现丢失数据的现象，为此在每次I/O操作之间插入几条指令作为延时，等待慢速外设。

PIT

fleurix通过Intel 8253 PIT(Programmable Interval Timer)定时器定时产生中断，用于计时与进程调度。

Intel 8253 PIT芯片拥有三个定时器：作为系统时钟，定时器1为历史遗留中用于定期刷新DRAM，定时器2用于扬声器。三个定时器分别对应三个数据寄存器0x40、0x41、0x42，以及一个命令寄存器0x43。这里只需要关心计时器0的功能，用到的寄存器只有0x40与0x43。

定时器0的默认频率为1193180HZ，可以通过如下的代码调整它的频率：

```
uint di = 1193180/HZ;
outb(0x43, 0x36);
outb(0x40, (uchar)(di&0xff));
outb(0x40, (uchar)(di>>8));
```

以上代码摘自 `src/kern/timer` 中的 `timer_init()` 。其中 `HZ` 常量的值为100，如此设置，可使PIT在每100毫秒产生一次中断，触发中断处理例程 `do_timer()` 。每次时钟中断被称作一个节拍(tick)。

VGA

VGA(Video Graphics Array，视频图形阵列)是使用模拟信号的一种视频传输标准，内核可以通过它来控制屏幕上字符或者图形的显示。

在默认的文本模式(Text-Mode)下，VGA控制器保留了一块内存(0x8b000~0x8bfa0)作为屏幕上字符显示的缓冲区，若要改变屏幕上字符的显示，只需要修改这块内存就好了。它可以被视作如下的一个二维数组，以表示屏幕上显示的25x80个字符：

```
/* VGA is a memory mapping interface, you may view it as an 80x25 array
 * which located at 0x8b000 (defined in main.ld).
 * */
extern struct vchar vgamem[25][80];
```

其中每项的结构如下：

```
struct vchar {
    char    vc_char:8;
    char    vc_color:4;
    char    vc_bgcolor:4;
};
```

其中， `vc_char` 表示要显示的字符内容， `vc_color` 与 `vc_bgcolor` 分别表示字符的颜色与字符的背景色。

除了字符的显示，我们也希望能够控制光标的位置，这里需要用到的是 `0x3D4` 与 `0x3D5` 两个端口，相关代码如下：

```
/* adjust the position of cursor */
void flush_csr(){
    uint pos = py * 80 + px;
    outb(0x3D4, 14);
    outb(0x3D5, pos >> 8);
    outb(0x3D4, 15);
    outb(0x3D5, pos);
}
```

VGA内部的寄存器多达300多个，显然无法一一映射到I/O端口的地址空间。对此VGA控制器的解决方案是，将一个端口作为内部寄存器的索引： `0x3D4` ，再通过 `0x3D5` 端口来设置相应寄存器的值。在这里用到的两个内部寄存器的编号为 `14` 与 `15` ，分别表示光标位置的高8位与低8位。

以上代码皆可见于 `src/chr/vga.c` 。

系统调用

系统调用(System Call)即应用程序访问内核的接口。

在fleurix中，每个系统调用对应一个系统调用号，在调用时，将系统调用号放置于 `eax` ，将可能的参数放置于 `ebx` 、 `ecx` 、 `edx` ，最后通过 `int 0x80` 从而进入内核并触发中断处理例程 `do_syscall()` ，继而依据系统调用号查询数组 `sys_routines[]` 找到对应的处理例程并执行。待执行结束之后，将返回值放置于中断上下文的 `eax` 寄存器中，并在出错时设置 `errno` 。

为方便在应用程序中调用系统调用，fleurix提供了四个宏 `_SYS0` 、 `_SYS1` 、 `_SYS2` 、 `_SYS4` 来生成系统调用的C接口。以 `_SYS3` 为例：

```
#define _SYS3(T0, FN, T1, T2, T3)               \
    T0 FN(T1 p1, T2 p2, T3 p3){                 \
        register int r;                         \
        asm volatile(                           \
            "int $0x80"                         \
            :"=a"(r)                            \
            :"a"(NR_##FN),                      \
            "b"((int)p1),                       \
            "c"((int)p2),                       \
            "d"((int)p3)                        \
        );                                      \
        if (r<0){                               \
            errno = -r;                         \
            return -1;                          \
        }                                       \
        return r;                               \
    }

...
static inline _SYS3(int, write, int, char*, int);
static inline _SYS3(int, read, int, char*, int);
static inline _SYS3(int, lseek, int, int, int);
...
```

更具体的实例，可见于 `usr/` 目录下的几个应用程序。

分页

fleurix应用x86平台的分页机制，实现了纯页式的内存管理。与段式内存管理(如DOS)或者段页式混合的内存管理(如linux0.11)相比，纯页式内存管理(以下简称”页式内存管理”)中可用的地址空间更大，也更加灵活。比如写时复制与请求调页这样的机制，在段式内存管理中则属于不可能实现的。

按照x86平台的分页机制，内存被划分为4kb或者4mb大小的物理页(又称”页框”)，由页表来表示虚拟页到物理页的映射关系。为节约页表本身所占用的内存，x86采用了二级页表。每个页表占4kb，含有1024条页表项，可以映射4mb的地址空间；页目录也同样4kb，含有1024项，可以映射4gb的地址空间。在进行地址翻译时，将先查询页目录，找到虚拟地址对应的页表，再在页表中查询得出相应的物理页，外加页内的偏移，最终得到物理地址。其中有个例外，便是4mb的大页，页目录中的表项可以不指向一个页表，而是仅仅表示一个4mb大页的地址映射，它的便利之处在于映射大块连续的地址空间，可以做到既方便又高效。

对于CPU来说，每次地址翻译都到内存中查询页表是不可容忍的高开销，为此，支持分页的CPU往往都提供了TLB(Translation Lookaside Buffer，俗称”快表”)作为页表的缓存。在这里开发者需要留意的是，只要更新了页表，便需要留意保持TLB的同步，不然就会有一些难于调试的问题出现。在bochs的内嵌调试器中，可以通过 `info tab` 命令来检查当前的页面映射。

(注: 因为内存局部性原理，TLB一般只需要很小(比如64项)即可达到不错的效果。)

页面可以被标记为只读(Readonly)或者不存在(Non-Present)，也可以设置页面的保护级别。这一来在读写内存时，如果发生不合法的内存读写，就会产生一个页面错误(Page Fault)，触发中断处理例程 `do_pgfault()` (见 `src/mm/pgfault.c`)。这时产生页面错误的地址，将被保存在cr2寄存器中，同时产生一个error code，表示页面错误的类型。待页面错误处理完成，被打断的程序可以恢复执行，也有可能因为严重的错误而中止(收到信号 `SIGSEGV`)。

在fleurix中，每个进程拥有一个独立的页目录，从而实现进程地址空间的隔离；通过4mb的大页，实现虚拟地址与物理地址的一对一映射直到128mb为止，作为内核地址空间；并过页表项的保护级别，限制用户态应用程序对内核地址空间的读写；通过将页面标记为只读或者不存在，实现写时复制(Copy On Write)与请求调页(Demand Paging)。 。

对于x86平台，值得留意的地方有：

- cr0寄存器中的Paging位表示分页机制的开关(`mmu_enable()` , 见 `src/inc/asm.h`)；
- cr4寄存器中的PSE位表示4mb大页的开关(在一些较旧的CPU上并没有PSE的支持)；
- 页目录的地址装载于cr3寄存器(`lpgd()` , 见 `src/inc/asm.h`)；
- 页面错误中的error code可能会有三种flag，即 `PFE_P` 、 `PFE_W` 与 `PFE_U` (定义于 `src/inc/mmu.h`)，分别表示页面不存在、页面只读及权限不足。
- 只要重新装载页目录，即为刷新TLB(`flmmu()` , 见 `src/mm/pte.c`)。

内存分配

fleurix假定用户的物理内存为128mb，并将内核永远地映射于每个地址空间的低端(0~128mb)，使得内核地址空间中的虚拟地址与物理地址做到一对一的映射。这一来只要分配了物理页面，内核就可以直接读写它的内容或将它映射到用户进程。需要留意的是，从技术角度这一假设并不合理：若用户的物理内存若小于128mb，内核就会崩溃；若用户的物理内存大于128mb，则无法利用128mb以上的内存。但它可以有效地降低项目开发之初的复杂度，待项目进入轨道，则应优先解决这一问题。

`pgalloc()` 与 `pgfree()` 为内核内存分配的基础例程，分别用于申请/释放一个物理页。一个物理页面可能会被多个进程映射到，因此一个引用计数是必须的；物理页面可能会比较多，使用穷举式的分配效率不高。对此，fleurix实现了一个 `struct page` 结构(定义于 `src/inc/page.h`)，并在内核初始化时，初始化一个数组 `struct page coremap[NPAGE]` 与一个队列 `struct page pgfreelist` (见于 `src/mm/pm.c` 中的 `pm_init()`)，前者作为物理页是否可用的标记，数组的每一项对应一个物理页，物理页面的地址就等于 数组下标 * 4kb ，若对应的 `struct page` 结构中的引用计数为0，则表示物理页是可用的；后者则将所有可用的物理页组织到一个链表之中，这一来即可将分配/释放物理页的操作的时间复杂度降到O(1)。

kmalloc()

fleurix使用了一个简单且高效的内存分配算法，它将 `pgalloc()` 作为后端，能够以O(1)的时间分配2次幂对齐的虚拟内存块，单次内存分配的上限为4kb。

`kmalloc()` 将固定大小的内存块(32b、64b、128b…4kb)分别组织为不同的链表。假如待分配的内存块大小为n，它会依据n来找到合适的链表(通过 `bkslot()` ，见于 `src/mm/malloc.c`)，其中内存块的大小为m(m为2次幂且 `n <= m <= 4096`)，然后检查链表中是否有可用的内存块。如果有，就将它取出链表，直接返回；如果没有，则通过 `pgalloc()` 分配一个物理页，将它划分为 `4096 / m` 个内存块并链到对应的链表中，重复尝试分配。

与C标准库函数 `free()` 的不同在于， `kfree()` 需要调用者记住内存块的大小，用以找到对应的链表。这是个不好的设计，使用者若将大小写错就会有bug产生。另外值得留意的地方是，除了4kb内存块的特殊情况， `kfree()` 不能将其它物理页返还给操作系统，而是留做以后内存分配的保留内存。这是一个不足之处，如果一次性分配比较多的临时对象，将会造成较大的内存浪费。

`kmalloc()` 与 `kfree()` 都不会进入睡眠，如果物理内存用尽则会产生一个 `panic()` ，这是编写代码时为方便调试而遗留的问题，也是亟需待改进的一个地方。

静态内存分配

对于内核中常用数据结构(比如 `struct inode` 、 `struct super` 等)的内存分配，fleurix采用的还是早期UNIX的解决方案：某类对象单独一个固定长度的数组，通过对象的一个标志判断是否可用，如果可用，则为修改标志并返回；如果不可用，则根据情况进入睡眠或者返回错误。

这一方案的优点在于几乎没有任何依赖，在内核开发之初即可在一定程度上满足内存分配的需求。不足在于每类对象的分配都是代码的重复，代码的复用率很低。

改进方案就是采用slab算法，将不同的对象组织在不同的缓存之中，而将内存分配的接口统一起来。

进程

进程即运行中的程序实体。每个进程拥有独立的地址空间以及一些资源，相互并发执行。在fleurix中，进程为代码执行与资源管理的基本单位。

终其一生，进程可能有五种状态：

- `SSLEEP` ： 睡眠且不可被信号唤醒，等待一个高优先级的事件；
- `SWAIT` ： 睡眠，可被信号唤醒，等待一个低优先级的事件；
- `SRUN` ： 正常执行；
- `SZOMB` ： 僵尸进程，是在进程因为某种原因退出执行(主动调用 `_exit()` 或者被信号杀死)、在被父进程回收之前的进程状态。
- `SSTOP` ： 停止中，在进程创建之初以及进程回收时的进程状态。

在fleurix中，表示进程的结构为 `struct proc` ，它含有进程的pid(`p_pid`)、状态(`p_stat`)、父进程id(`p_ppid`)、进程组(`p_pgid`)、用户id(`p_uid`)、组id(`p_gid`)、地址空间(`p_vm`)、上下文(`p_contxt`)、打开的文件(`p_ofile`)、信号处理例程(`p_sigact`)、可执行文件的inode(`p_inode`)等诸多信息，正是fleurix中最为复杂的结构。

`struct proc` 与这一进程的内核栈同处一个物理页，前者位于低端固定，后者位于高端向下增长。在这里不难发现，内核栈的可用空间非常小(小于4kb)，因此在内核开发中，应尤其注意不要在栈上放置较大的对象，抑或进行较深的递归，不然内核栈若溢出，绝不会像用户态中那样出现 `Segmentation Fault` 的提示，而会默默地搞乱内核中的数据结构，出现一些难于调试的问题。

为方便对进程结构的引用，fleurix设置了一个数组即 `struct proc *proc[NPROC]` ，数组的下标即进程的pid， `NPROC` 则为系统中进程数量的上限；以及一个指针 `struct proc *cu` ，永远指向当前的进程结构。

进程创建

进程只能通过 `fork()` 系统调用创建，它会复制当前进程的地址空间与资源，生成一个一模一样的子进程。不过， `fork()` 会在父进程中返回0，在子进程中则返回子进程的pid作为区别，如下：

```
int pid;
if ((pid = fork()) == 0) {
    printf("I'm the parent process\n");
}
else {
    printf("I'm the child process\n");
}
```

在 `fork()` 时，直接复制整个进程地址空间的操作是昂贵的，而且大多数子进程都会在执行之初调用 `exec()` 覆盖掉当前地址空间，之前的复制也就没有意义了。对此，类UNIX系统大多基于CPU的分页机制，提供了写时复制的实现：在复制进程地址空间时，并不直接拷贝地址空间中页面的内容，而是仅仅复制父进程的页表，使得父子进程共享相同的物理页，并将二者的虚拟页面皆设置为只读。随后若二者任一方试图修改内存，则申请一个新的物理页并复制旧页的内容。这里需要留意的是，为控制物理页的共享，每个物理页都需要维护一个引用计数，当 `fork()` 时引用计数增1，当进程杀死或者发生写时复制时减1，并在引用计数为0时释放这个物理页。

`fork()` 的主要行为大致如下：

- 申请pid与进程结构
- 设置ppid为父进程的pid
- 复制用户相关的字段，如 `p_pgrp` 、 `p_gid` 、 `p_ruid` 、 `p_euid` 、 `p_rgid` 、 `p_egid`
- 复制调度相关的字段，如 `p_cpu` 、 `p_nice` 、 `p_pri`
- 复制父进程的文件描述符(`p_ofile`)，并增加引用计数
- 复制父进程的信号处理例程(`p_sigact`)
- 通过 `vm_clone()` (见于 `src/mm/vm.c`)，复制父进程的地址空间(`p_vm`)
- 复制父进程的寄存器状态(`p_contxt`)
- 复制父进程的中断上下文，并设置 `tf->eax` 为 `0` ，使 `fork()` 在子进程中返回0。

fleurix在开始运行之初会初始化一个0号进程(`proc0_init()` ，见于 `src/kern/fork.c`)，其后的所有进程皆由它fork而来。

程序执行

`exec()` 是fleurix中行为最为复杂的系统调用之一。就表面的行为而言，它会取一个可执行文件的地址与相关参数(`argv`)，并执行它。然而在内部，它所做的工作却远比表面上复杂：

- 读取文件的第一个块，检查Magic Number(`NMAGIC`)是否正确
- 保存参数(`argv`)到临时分配的几个物理页，其中的每个字符串单独一页
- 清空旧的进程地址空间(`vm_clear()` ，见于 `src/mm/vm.c`)，并结合可执行文件的header，初始化新的进程地址空间(`vm_renew()` ，见于 `src/mm/vm.c`)
- 将 `argv` 与 `argc` 压入新地址空间中的栈
- 释放临时存放参数的几个物理页
- 关闭带有 `FD_CLOEXEC` 标识的文件描述符
- 清理信号处理例程
- 通过 `_retu()` 返回用户态

这里值得留意的是，之所以将 `argv` 保存到临时分配的几个页面，是因为 `argv` 中的字符串与这个数组本身都是来自旧的地址空间，而旧的地址空间会被销毁， `argv` 所指向的内存区域，自然也就无法访问了。

与写时复制的实现相似， `exec()` 在执行时，并不会立即将可执行文件完全读入内存。而是通过 `vm_renew()` ，将当前进程的虚拟页面统统设置为不存在，待进入用户态开始执行时，每发生一次页面不存在的错误，便读取一页可执行文件的内容并映射。这样的机制被称作请求调页(Demand Paging)，好处是可以加速程序的启动，不必等待可执行文件完全读入内存即可开始程序的执行，在某种意义上，也可以节约内存的使用。缺点是如果程序的体积较小，就不如一次性将可执行文件全部读入内存的方式高效。

为简单起见，fleurix只支持a.out格式作为可执行文件格式，对应可执行文件中不同的区段(section)，进程的地址空间也分为不同的内存区(VMA，Virutal Memory Area)，如正文区(`.text`)、数据区(`.data`)、bss区(.bss)、堆区(`.heap`)与栈区(`.stack`)。它们的性质各不相同：正文区与数据区内容都来自可执行文件，然而正文区是只读的，数据区可读可写；bss区、堆区与栈区的内存皆来自动态分配，都可读可写，不过bss区的内存都默认为0，堆区可以通过 `brk()` 系统调用来调整它的长度，而栈区可以自动向下增长。对于这些不同需求，fleurix提供了一个结构 `struct vma` ，它可以绑定一个inode，并在必要时依据相关的几个标志(即 `VMA_RDONLY` 、 `VMA_STACK` 、 `VMA_ZERO` 、 `VMA_MMAP` 、 `VMA_PRIVATE`)执行不同的操作。具体可见于 `src/mm/pgfault.c` 文件中 `do_no_page()` 的相关代码。

进程切换

负责进程切换的函数为 `swtch_to()` ，可见于 `src/kern/sched.c` 。内容如下：

```
void swtch_to(struct proc *to){
    struct proc *from;
    tss.esp0 = (uint)to + PAGE;
    from = cu;
    cu = to;
    lpgd(to->p_vm.vm_pgd);
    _do_swtch(&(from->p_contxt), &(to->p_contxt));
}
```

`_do_swtch()` 是一段汇编例程，它负责将当前的上下文保存到 `from->p_contxt` ，同时将 `to->p_contxt` 中保存的上下文恢复出来，也就是真正发生进程切换的地方。

fleurix采用软件的进程切换，一切进程切换都发生在内核态。结合 `swtch_to` 的源码，已知进程的上下文有：

- 内核栈的顶，供中断处理例程使用；
- 页目录，也就是地址空间；
- `eip` ；
- `esp` 与所有其它通用寄存器(`eax` 、 `ebx` 、 `ecx` 、 `edx` 、 `edi` 、 `esi` 、 `ebp`)。

需要留意的是，依据gcc的调用约定， `eax` 、 `ecx` 与 `edx` 为 `caller-saved registers` ，会在调用 `_swtch_to()` 时由调用者自动保存，不需要额外保存，因此内核只需要保存 `ebx` 、 `ebp` 、 `edi` 、 `esi` 、 `esp` 五个通用寄存器。 另外，因为进程切换都发生在内核态， `cs` 等段寄存器的内容皆等同于常量，也无需保存。

fleurix将上下文相关的寄存器保存在一个 `struct jmp_buf` 结构中，它与C标准库中的 `jmp_buf` 基本相同，甚至可以这样想：进程切换等价于在切换地址空间之后，为当前进程的上下文执行 `setjmp()` 记录下来，同时通过 `longjmp()` 跳转到目标进程的上下文。

进程调度

fleurix采用传统UNIX的优先级调度算法。

在 `src/kern/proc.h` 中可以见到几个默认的优先级： `PSWP` 、 `PINOD` 、 `PRIBIO` 、 `PPIPE` 、 `PRITTY` 、 `PWAIT` 、 `PSLEP` 、 `PUSER` 。其中除了优先级最小的 `PUSER` 专用于CPU调度的基数之外，皆表示某事件的特定优先级。

在进程结构中，调度相关的字段只有三个(取值范围皆为-126到127)：

- `p_cpu` ：已执行的时间片计数；
- `p_nice` ：用户通过 `nice()` 系统调用设置的微调；
- `p_pri` ：进程的优先值，优先值越小优先级越高。

内核会随着节拍(`tick`)增加当前进程的 `p_cpu` ，同时每隔一定时间便依据 `p_cpu` 与 `p_nice` 重新计算 `p_pri` (可见于 `src/kern/timer.c` 中的 `sched_cpu()`)。公式大致为：

```
p->p_pri = p->p_cpu/16 + PUSER + p->p_nice;
```

也会调整所有进程的 `p_cpu` ，使得进程不至饿死：

```
p->p_cpu /= 2;
```

随着时间的增加，当前进程的优先级会慢慢地低于其它的任何进程。在这时调用 `swtch()` ，便可以找出当前优先级最高的进程并切换。

值得留意的是，调用 `swtch()` 的时机有两种：

1. 从内核态返回用户态的那一刻，发生进程抢占；
2. 进程主动调用，自愿放弃控制权，一般是为了等待资源。

fleurix是非抢占的内核，一切进程抢占都发生在内核态返回用户态的那一刻。这样的考虑主要出于：

- 来自PIT的时钟中断会定时触发。
- 一些中断处理例程就在执行结束之后，一般都会唤醒一些等待资源的进程。这些进程的优先级都比较高(`PRIBIO` 、 `PINO`)，在这时切换进程，可以使得相应的资源在第一时间得到处理。

相关代码可见于 `src/kern/trap.c` 中 `hwint_common()的结尾处` ：

```
setpri(cu);
if ((tf->cs & 3)==RING3) {
    swtch();
}
```

它首先尝试调整当前进程的优先级，再通过中断上下文中保存的 `cs` 寄存器判断当前的中断上下文是否是来自用户态。只有确定是来自用户态，才尝试执行任务切换，这样可以保证内核态中不会发生抢占。

进程同步

fleurix的内核是非抢占的，但中断处理例程依然有可能打断内核代码的执行。要保证代码的一致性，可以通过 `cli()` 与 `sti()` 来关/开中断形成一个临界区。需要留意的是，开/关中断的方式只在单处理器环境中适用，若支持多处理器，则需要提供自选锁(spin lock)的实现。

此外一个常见的情景是，在申请资源时若这一资源不可用，就让这个进程进入睡眠(`sleep()`)以等待资源的释放，待资源恢复可用时，再唤醒(`wakeup()`)所有等待该资源的进程恢复执行。 `sleep()` 与 `wakeup()` 即为fleurix的基本同步原语。

`sleep()` 的代码如下，取自 `src/kern/sched.c` ：

```
/* mark a proccess SWAIT, commonly used on waiting a resource.
**/
void sleep(uint chan, int pri){
    if (pri < 0) {
        cli();
        cu->p_chan = chan;
        cu->p_pri  = pri;
        cu->p_stat = SSLEEP; // uninterruptible
        sti();
        swtch();
    }
    else {
        if (issig())
            psig();
        cli();
        cu->p_chan = chan;
        cu->p_pri = pri;
        cu->p_stat = SWAIT; // interruptible
        sti();
        if (issig())
            psig();
        swtch();
    }
}
```

`sleep()` 的第一个参数 `chan` 为”channel”的缩写，表示等待的事件的标志符；第二个参数 `pri` 表示进程在唤醒那一刻的优先级。依据优先级的分类，睡眠又分为可中断(interruptible)与不可中断(uniterruptible)两种，意指在睡眠中的进程若收到信号，是否中断睡眠恢复执行。

`wakeup()` 的代码如下，取自 `src/kern/sched.c` ：

```
void wakeup(uint chan){
    struct proc *p;
    int i;
    for(i=0; i<NPROC; i++){
        if ((p = proc[i]) == NULL)
            continue;
        if (p->p_chan == chan) {
            setrun(p);
        }
    }
}
```

它的内容就是依据参数 `chan` ，找到所有因等待此事件而进入睡眠的进程并唤醒。

设备

设备(Device)即外部设备在内核中的基本抽象，主要分为两种：

- 块设备：将数据储存在固定大小的块中，每个块都有自己的地址，可供驱动程序随机访问，如硬盘、光驱等；
- 字符设备：输入输出都是不可以随机访问数据流，如键盘、打字机等。

在fleurix中，块设备与字符设备分别对应 `struct bdevsw` 与 `struct cdevsw` 两个结构(定义于 `src/inc/conf.h`)，如下：

```
struct bdevsw {
    int             (*d_open)();
    int             (*d_close)();
    int             (*d_request)(struct buf *bp);
    struct devtab    *d_tab;
};

extern struct bdevsw    bdevsw[NBLKDEV];

struct cdevsw {
    int             (*d_open)   (ushort dev);
    int             (*d_close)  (ushort dev);
    int             (*d_read)   (ushort dev, char *buf, uint cnt);
    int             (*d_write)  (ushort dev, char *buf, uint cnt);
    int             (*d_sgtty)();
```

```
};

extern struct cdevsw    cdevsw[NCHRDEV];
```

可以看出，两个结构的主要部分都是函数指针，它们就是设备驱动程序的统一接口了。

类UNIX系统将设备文件(Device File)作为应用程序访问设备的接口，使得访问外部设备与访问一个普通的文件并无二致。设备文件本身并没有任何内容，真正发挥作用的只是设备类型与设备号——在读写设备文件时，内核会依据它们来调用对应的设备驱动程序(Device Driver)，执行真正的读写。

在linux下可以通过mknod命令来创建一个设备文件，比如：

```
mknod /dev/tty0 c 1 0
```

Buffer Cache

比起访问内存，访问外部设备的速度往往要慢许多。为此，fleurix为块设备实现了Buffer Cache，将最近读过的块缓存到内存中，从而加快对块设备的访问。另外，Buffer Cache也扮演着I/O请求队列的角色，从中断中读取的数据将直接写入Buffer Cache中。

Buffer Cache相关的结构主要为 `struct buf` 、 `struct devtab` ，以及 `char buffers[NBUF][BUF]` 与 `bfreelist` ，其中每个 `struct buf` 都对应着 `buffers[]` 中的一块内存，大小与文件系统的虚拟块相同(1024字节，可见于 `param.h` 中 `BLK` 的定义)。另外，它们主要构成了三个 `buf` 对象的链表：

- 空闲列表：表示当前系统中所有可用的 `buf` ，用于 `buf` 的分配。使用LRU(Last Recently Used)策略，分配一定是在链表的头部取出，释放则一般都是放回链表的尾部。链表的头部为 `bfreelist` ， `buf` 之间由 `av_prev` 和 `av_next` 连接；
- 缓存列表：每个设备拥有独立的缓存列表，盛放着设备所有的 `buf` ，用于 `buf` 缓存的查找。除非被标记为 `B_BUSY` ， `buf` 可以同时存在于空闲列表和缓存列表。链表的头部为 `struct devtab` ，由 `b_prev` 与 `b_next` 连接；
- 请求队列：同为每个设备独立，表示该设备的I/O请求队列。位于请求队列中的 `buf` 会存在于设备的缓存列表，但不会存在于空闲列表。链表的头部为 `struct devtab` ，由 `av_prev` 与 `av_next` 连接。

可以认为，Buffer Cache为块设备的读写提供了统一的接口，也为文件系统实现了所需的基础例程。这些例程主要有：

- `getblk(dev, blknum)` :分配 `buf` 对象，并标记为 `B_BUSY` ；
- `brelse(buf)` ：释放 `buf` 对象，将其放回空闲列表；
- `bread(dev, blknum)` ：读取设备的块，将 `buf` 对象插入设备的请求队列，随后进入睡眠等待读取完毕；
- `bwrite(dev, buf)` ：将 `buf` 对象中的内容写回设备。

以上例程皆可见于 `src/blk/buf.c` 。

需要留意的是， `getblk()` 这个名字很容易给人一个错误的印象，实际上 `getblk()` 函数并不会读取设备的块(读取设备块的函数为 `bread()`)，它用于分配 `buf` 结构，并将其标记为 `B_BUSY` ：依据设备号和块号，查找相应的 `buf` 是否存在于缓存中，若存在，就直接返回它；若不存在，则从空闲列表中取出一个可用的 `buf` 对象返回。 具体起来，有如下五种情景：

1. 在设备的缓存列表中找到了对应的 `buf` 对象，且正好可用，则返回这一 `buf` 并标记为 `B_BUSY` (通过 `notavail()` 函数)；
2. 在设备的缓存列表中找到了对应的 `buf` 对象，不过这个 `buf` 正忙（ `B_BUSY` ），则将这个 `buf` 标记为 `B_WANTED` ，令进程睡眠等待它被释放；
3. 没有在设备的缓存列表中找到对应的 `buf` 对象，且 `bfreelist` 为空，则将整个 `bfreelist` 标记为 `B_WANTED` ，令进程睡眠等待它获取可用的 `buf` ；
4. 没有在设备的缓存列表中找到对应的 `buf` 对象，且 `bfreelist` 不为空，则将头部的 `buf` 取出 `bfreelist` ；
5. 若得到的 `buf` 对象被标记为 `B_DIRTY` ，则表示它有内容发生变化，需要写回到设备中。

请求队列

设备在同一时刻一般只能处理一个请求，且时间较长。因此，合理的做法是将待处理的I/O操作排队，在发出一个请求之后使进程进入睡眠等待中断，待中断发生时，读取输入并再次尝试发送队列中的请求，如是循环。

在fleurix中，请求队列并无专门的数据结构，而是直接利用 `struct devtab` 为队列的头部， `struct buf` 作为队列的成员，通过 `av_prev` 与 `av_next` 连接起来。

以硬盘为例，发送I/O请求的例程为 `hd_request()` ，它取一个 `buf` 对象作为参数，只负责将其插入 `hdtab` 的请求队列。而真正依据请求队列发送I/O请求的例程则为 `hd_start()` ，它取出队列的头部，依据 `buf` 对象的标志(`B_READ` 或者 `B_WRITE`)来发送读请求或者写请求，待设备在读取/写入完毕之后，就会触发中断处理例程 `do_hd_intr()` ，在这里将 `buf` 对象取出请求队列，在读取数据之后，唤醒等待在这一 `buf` 对象上的所有进程，并再次调用 `hd_start()` ，尝试处理排队中的I/O请求。

以上例程皆定义于 `src/blk/hd.c` 。

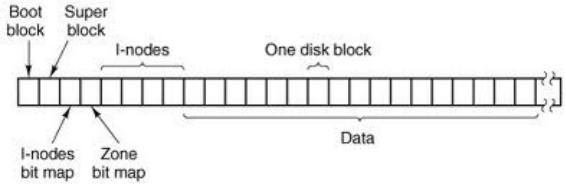
文件系统

文件是对I/O的抽象，而文件系统提供了文件的组织方式。fleurix实现了minix v1文件系统，它结构简单、易于实现，而且在开发环境中也有丰富的工具可供使用，如 `mkfs.minix` 、 `fsck.minix` 等。

超级块

超级块表示了文件系统的基本信息，也描述了文件系统的存储结构。在内核中，有时可以将超级块视作文件系统的同义词。

minix v1文件系统主要分为六个部分，如下图：



1. 引导块，总是位于设备的第一个虚拟块，为bootloader所保留；
2. 超级块，位于第二个虚拟块。它保存了一个文件系统的详细信息，比如inode的数量、zone的最大数量等；
3. inode位图，每个位对应一个磁盘上的inode，表示它是否空闲，大小与超级块中的 `s_nimap_blk` 字段相关；
4. zone位图，每个位对应一个zone，表示它是否空闲，大小与超级块中的 `s_nzmap_blk` 字段相关。zone是文件系统中虚拟块的别名，一个zone可能等于1个物理块，也可能等于2、4、8个物理块的大小，具体由超级块中 `s_log_bz` 字段指定。在fleurix中，zone等于两个物理块的大小(1024字节)。
5. inode区域，储存着文件系统中所有的inode，大小与超级块中的 `s_max_inode` 字段相关。
6. 数据区域，也就是文件系统中所有的虚拟块，供inode引用。

在fleurix中有两个数据结构与超级块相关： `struct d_super` 与 `struct super` ，皆定义于 `src/inc/super.h` ，分别对应磁盘上与内存中超级块的表示。后者多了几个字段来表示挂载信息。

与 `struct buf` 结构类似，内存中也有一个固定的 `struct super mnt[NMNT]` 数组(定义于 `src/fs/mount.c`)，用作 `super` 对象的分配与缓存。然而更重要的用途，则为内核中所有文件系统的挂载表。

在类UNIX系统中若要访问一个文件系统，必先将它挂载(Mount)。在fleurix中，对应的例程为 `do_mount(uint dev, struct inode *ip)` 。它取两个参数，第一个参数为文件系统的设备号，第二个参数指向挂载目标的inode。它会首先遍历 `mnt[]` 数组，查找设备号对应的 `super` 对象是否已存在，若存在，则直接跳转至 `_found` ；若不存在，则选出一个空闲的 `super` 对象，读取磁盘上的超级块并跳转至 `_found` 。随后，它会依据设备号判断是否为根文件系统，并增加挂载目标的引用计数。

与之相对，卸载一个文件系统的例程为 `do_umount(ushort dev)` 。它会将设备号对应的 `super` 对象写回到磁盘，随后释放它，并减少目标inode的引用计数。

除 `do_mount()` 与 `do_umount()` 之外，有关超级块的例程还有：

- `getsp()` ，依据设备号，获取一个已挂载的 `super` 对象并上锁；
- `unlk_sp()` ，释放一个 `super` 对象的锁；
- `spload()` ，读取磁盘中的超级块到 `super` 对象；
- `spupdate()` ，将 `super` 对象的改动写回磁盘。

以上例程皆定义于 `src/fs/super.c` 。

块分配

minix文件系统采用位图来表示文件系统中空闲的块，一个位对应着数据区域的一个逻辑块，1表示已占用，0表示可用。每个块对应着一个块号，最小为0，最大为数据区中块的数，在文件系统格式化时确定。表示块号的类型为 `unsigned short` (16位)。

在fleurix中，通过 `balloc()` (定义于 `src/fs/alloc.c`)来分配块。它取一个设备号做参数，通过查询位图找到并返回文件系统中第一个可用的块号。若没有可用的块，则报告一个错误。

与之相对，释放块的例程为 `bfree()` 。

inode

在类UNIX操作系统中，普通文件、目录或者文件系统中的其它对象皆由一个统一的数据结构inode来表示。它记录了文件的类型、用户信息、访问权限、修改日期等信息，也记录了文件中逻辑块的布局，但并不包含本文件的名字信息。

每个inode拥有一个唯一的编号，作为内核访问inode的凭据。编号从1开始，最大为文件系统中inode的数量，在文件系统格式化时确定。表示inode编号的类型为 `unsigned short` (16位)。

同超级块类似，fleurix中有两个数据结构与inode相关：`struct d_inode`与`struct inode`，皆定义于`src/inc/inode.h`，分别对应磁盘上与内存中inode的表示。后者增加了引用计数、设备号、inode编号与标志等信息。

在fleurix中，要访问一个inode对象，可以通过`iget()`例程(定义于`src/fs/inode.c`)，它取一个设备号与inode编号做参数，返回一个上锁的inode对象。大体行为如下：

1. 依据设备号与inode编号，遍历`inode[]`数组判断inode对象是否位于缓存；
2. 若位于缓存且无锁，则增加引用计数(使`i_count`增1)并上锁(通过`lock_ino()`，定义于`src/fs/inode.c`)后返回；
3. 若位于缓存但上锁，则进入睡眠等待inode对象释放，重复步骤1；
4. 若没有位于缓存，则分配一个空闲的inode对象，读取磁盘中的inode对象，重复步骤1；
5. 若没有位于缓存，且没有空闲的inode对象，则报告一个错误。

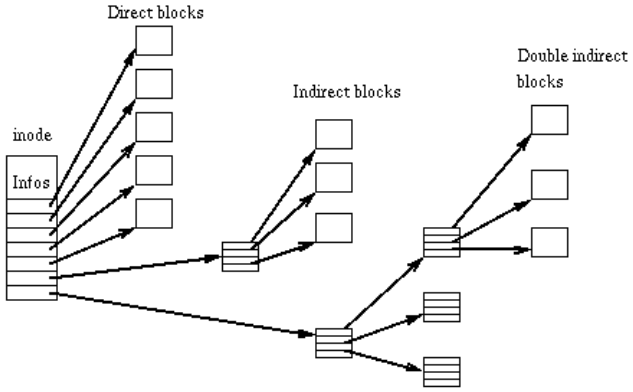
与之相对，释放`inode`对象的例程为`iput()`，它会将`i_count`减一，当`i_count`为0时释放inode对象。

除了`i_count`，inode结构还有一个字段`i_nlink`，用于 表示磁盘上的引用数，也就是硬连接的数量。当新建一个文件时，`i_nlink`的值为1，随后每增加一个硬连接时增1，删除时减1，当`i_nlink`为0时才真正删除磁盘上的inode。

此外值得注意的是，`iput()`与`unlk_ino()`虽同为”释放一个inode对象”，但含义有所不同。准确来讲，`unlk_ino()`的行为是释放一个inode对象的锁，`iput()`则是根据引用计数来释放inode对象本身。锁的目的是限制对象的控制权，保护对象的数据不被破坏，内核必须在系统调用的结束之前及时地释放锁，不然将导致死锁；而引用计数的目的是跟踪对象的所有权的变化，来管理对象的生存周期。

bmap()

文件是组织虚拟I/O的一种方式，每个文件都可以视作是独立的一段地址空间。fleurix通过`bmap()`(定义于`src/fs/bmap.c`)将文件中的偏移地址翻译为设备的物理块号，而inode在这里就扮演了翻译表的角色。



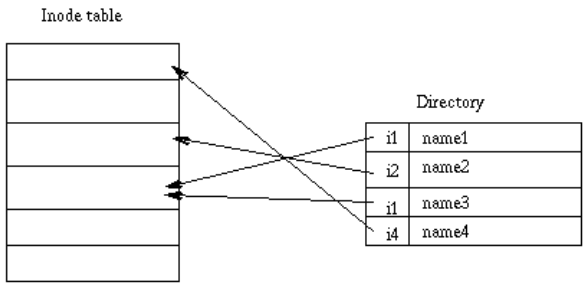
如上图，minix v1文件系统采用了传统UNIX文件系统的分组多级中间表，默认只提供7个逻辑块的映射，若文件增长超过7个块的大小，则分配一个块作为中间表，额外提供512个块(即`NINDBLK`，定义于`src/inc/param.h`，等于`BLK / sizeof(unsigned short)`)的映射。如果文件更长，就采用二级中间表，这样最大可以支持262663个块(`7+512+512*512`)的映射，也就是说，单个文件最大限制约为256mb(`MAX_FILESIZ`，定义于`src/inc/param.h`)。

`bmap(struct inode *ip, ushort nr, uchar creat)`取三个参数，第一个参数`ip`指向一个上锁的inode对象，第二个参数`nr`表示文件中虚拟块的偏移，第三个参数`creat`表示查找过程中是否申请新的块。当查找失败时若`creat`为0，会返回0表示映射不存在；若`creat`不为0，则申请一个块并继续查找。值得一提的是，文件系统中的一切块分配皆发生于设置`creat`标志时的`bmap()`。

`bmap()`主要用于`read()`、`write()`与`lseek()`等系统调用的实现，也在内核中读取文件时有所使用。

namei()

前面曾提到，inode结构并没有保存本文件的名字信息。所有文件的文件名，以及文件目录之间的层级关系，都保存在目录类型(`S_IFDIR`，定义于`src/inc/stat.h`)的inode中。每个文件系统的第1个inode都是目录类型。



目录的数据布局与普通文件一致，不同在于数据的内容。目录文件的格式可以视作是`struct dirent`结构的一个数组，表示了一个目录中文件名到inode编号的映射关系。`struect dirent`的声明为：

```
#define NAMELEN 12

struct dirent {
    ushort  d_ino;
    char    d_name[NAMELEN];
    char    __p[18]; /* a padding. each dirent is aligned with a 32 bytes boundary. */
};
```

每条 `dirent` (目录项)占据32字节，其中inode编号为2字节，文件名为12字节，保留16字节。可知在minix v1文件系统中，文件名的大小限制为12个字符。

而 `namei(char *path, uchar creat)` 所做的工作就是，根据路径依次查找目录文件，并在必要时新建inode，最后返回一个上锁的inode对象或在出错时返回NULL。此外，内核有时会对父目录的内容更感兴趣(比如通过 `unlink()` 来删除一个文件)，对此fleurix也提供了一个 `namei_parent(char *path, char **name)` 例程，它可以返回父目录的inode对象，同时找到指向目标文件名的指针。

在查找过程中需要小心地处理inode对象的锁和一些异常情况， `namei()` 与 `namei_parent()` 中有关遍历目录的代码会很复杂，所以将它们分开实现是没有意义的。为此，fleurix实现了 `_namei(char *path, uchar creat, uchar parent, char **name)` 作为 `namei()` 与 `namei_parent()` 所共有的基础例程。它的4个参数的意义分别为：

- `path` ：目标文件的路径，若 `path` 为绝对路径(以 `'/'` 开头)， `_namei()` 将从根目录开始查找，否则从当前的活动(`cu->p_wdir`)目录开始查找；
- `creat` ：若最后没有找到对应的文件，则新建一个文件；
- `parent` ：若不为0，则返回父目录的inode对象，同时将目标文件名的地址存入第四个参数 `name` ；
- `name` ：当 `parent` 不为0时，保存目标文件名的地址。

`_namei()` 主要用于 `link()` 、 `unlink()` 、 `open()` 、 `exec()` 等系统调用的实现，凡是需要访问文件路径的地方，就都会调用到它。

遇到的问题

总结

参考文献

- 《Linux内核完全注释》，赵炯 著
- 《Linux内核完全剖析》，赵炯 著
- 《莱昂氏UNIX源代码分析》，John Lions 著
- 《UNIX操作系统设计》， Maurice J.Bach 著
- 《操作系统设计与实现》，Andrew S. Tanenbaum、 Albert S. Woodhull 著
- 《现代操作系统》，Andrew S. Tanenbaum 著
- 《计算机的心智：操作系统之哲学原理》，邹恒明 著
- 《结构化计算机组成》，Andrew S.Tanenbaum 著
- 《链接器和加载器》，John R.Levine 著
- 《4.4 BSD操作系统的设计与实现》，Marshall Kirk McKusick、Keith Bostic、Michael J.Karels、John S.Quarterman 著
- 《UNIX Internals》，Uresh Vahalia 著
- 《Bran’s Kernel Development Toturial》，Brandon Friesen 著
- 《Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX† Operating System‡》，Ozalp Babao lug、William Joy、Juan Porcar 著
- 《Virtual Memory Architecture in SunOS》，Robert A. Gingell、Joseph P. Moran、 William A. Shannon 著
- 《Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference A-M》，英特尔公司 著
- 《Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2B: Instruction Set Reference N-Z》，英特尔公司 著
- 《Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1》，英特尔公司 著
- 《Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2》，英特尔公司 著
- 《80x86处理器和80x87协处理器大全》，Hummel，R.L. 编著
- 《UNIX环境高级编程》，W.Richard Stevens 著

- 《An Introduction to GCC》, Brian J. Gough 著