

操作系统项目文档

一：小组成员

| 学号 | 姓名 | 班级 |
|---------|-----|----------|
| 1352888 | 林悦锵 | 42028703 |
| 1352892 | 谭靖儒 | 42028703 |

二：项目概述

1. 配置相关

- 编写语言：C、汇编
- 开发环境：ubuntu linux下的 BOCHS
- 运行环境：mac(parallel desktop)、windows(VMWare)

2. 项目基础

- 该项目不是由自己一步步搭建出来，而是在一个已经完成的操作系统上进行了一定的修改。
- 该项目模拟dos系统，根据输入的命令进行不同的操作。
- 该项目主要研究的方向放在进程和输入输出(IO)两个模块上
- 主要完成的功能有：两个用户级应用，一个系统级应用，对内核部分的进程模块进行了一定的修改

三：开发过程

1. 系统工作流程

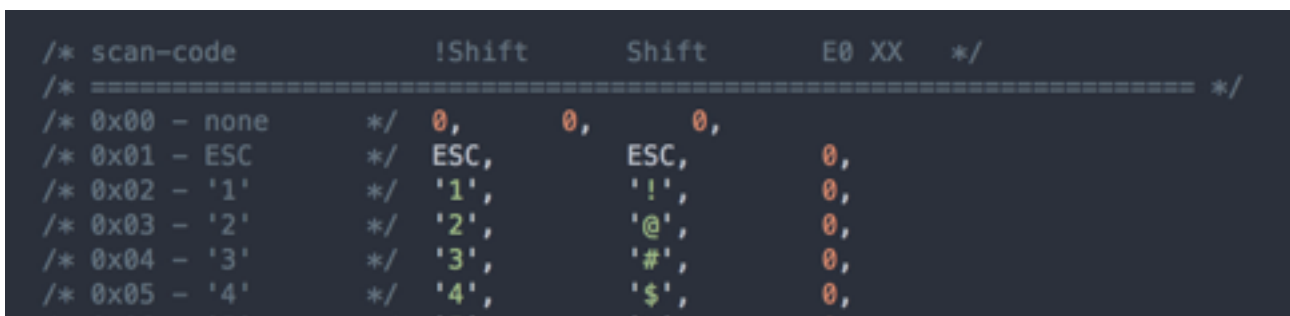
从软盘引导 → 在软盘中查找Loader.bin → 加载loader.bin → 跳转至loader.bin中的代码开始执行 → 在软盘中查找系统内核kernel.bin → 进入保护模式 → 加载kernel.bin → 跳转至kernel.bin中的代码开始执行 → 更新GDT → 初始化IDT → 初始化TSS → 跳入系统主函数 → 启动系统进程 → 开启时钟中断 → 开始进程调度 → （系统开始运转）

2. 输入输出研究

• 键盘响应(输入)

由书上的知识可知，要想让计算机响应用户的键盘事件，就必须建立一个键盘中断机制，所以我们在keyboard.c中定义了一个函数keyboard_handler放在init_keyboard()中，并在main.c的kernel_main()调用init_keyboard()；但是为了让计算机识别用户到底是按的什么键，就必须对键盘上的每一个按键都建立一个一对一的关系，又根据不同的按键有不同的Make和Break值，所以在Keymap.h里面定义了一个keymap数组来对应不同按键。另外，为了处理组合按键，例如Shift + A 我们建立了一个键盘输入缓冲区。

由图可以看出，对于每一个按键，都对应了3列，分别对应了单独按下已经两种组合按键。例如 Shift + 1 得到的是 ‘!’而不是’1’



| /* scan-code | !Shift | Shift | E0 XX | */ |
|----------------|--------|-------|-------|----|
| /* 0x00 - none | 0, | 0, | 0, | |
| /* 0x01 - ESC | ESC, | ESC, | 0, | |
| /* 0x02 - '1' | '1', | '!', | 0, | |
| /* 0x03 - '2' | '2', | '@', | 0, | |
| /* 0x04 - '3' | '3', | '#', | 0, | |
| /* 0x05 - '4' | '4', | '\$', | 0, | |

• 键盘响应(输出)

我们把实现输出的函数放在了tty.c的in_process里面，在keyboard.c里面

有一个函数`keyboard_read()`,每当发生了键盘中断,都会去执行这个函数,同时根据不同的`keymap`的映射,将参数传入`in_process`中去,最后在`in_process`函数中,将`key`的值打印出来。这样一来,我们就了解了整个键盘在输入输出过程中的各个步骤,所以,我们在响应自己的想要的键盘输入的时候,就可以在`keyboard_read`里面加入自己的`in_process`(暂且叫做`myIn_process()`)来完成自己想做的事情。

3. 进程研究

- 进程创建

课本知识里面讲到过,系统会创建一张静态的表来记录所有的进程,而表的大小是确定的,所以一个系统在运行过程中最大可同时运行进程的数量也是确定的(并发度),在这个操作系统中,也是这样做的。不难找到,在`global.h`文件中,有一个`proc_table`记录了所有的进程,可以发现他有几个成员,`stackframe`(进程栈,用来保存进程运行时各个寄存器的状态) `ticks`(进程调度时需要用到的时间片) `priority`(进程的优先级) `name`(进程的名称) `pid`(进程id) `run_count`(进程执行过的时间片,该属性为自己新加入的,用来实现后面的多级反馈队列) 等等,这里的`proc_table`是一个创建的一个宏,不是很方便的创建进程通过`dos`命令创建(其实可以不用宏),如果我们想要创建一个新的进程,首先我们需要修改`proc_table`宏,其次,在`kernel_main()`完成它的初始化,在`proc.h`中为其声明相关的函数。这样就能完成一个新进程的创建了。

- 进程的调度

调度基础:中断

简单的轮转调度:每次发生中断的时候,让`p_proc_ready`(指向`proc_table`某一项的一个指针) 自加一次(指向下一个进程),如果超过`proc_table`的大小,就让`p_proc_ready`指向第一个进程

```

p_proc_ready++;
if (p_proc_ready >= proc_table + NR_TASKS) {
    p_proc_ready = proc_table;
}

```

优先级调度:使用优先级调度的时候,引入了一个新的变量,ticks,简单的理解就是时间片,初始化的时候,让进程的ticks等于priority.且增加一个schedule(调度函数),每次调度发生的时候,选择优先级最高的一个进程为就绪进程,每次时钟中断的时候,就绪进程的ticks减1,如果ticks>0,那么函数直接返回(继续执行改就绪进程),当ticks为0时,调用schedule,如果所有进程的ticks都为0,为所有进程的ticks重新赋值为其priority的值.

clock.c

```

PUBLIC void clock_handler(int irq)
{
    ticks++;
    p_proc_ready->ticks--;

    if (k_reenter != 0) {
        return;
    }

    if (p_proc_ready->ticks > 0) {
        return;
    }

    schedule();
}

```

proc.c

```

PUBLIC void schedule()
{
    PROCESS* p;
    int greatest_ticks = 0;

    while (!greatest_ticks) {
        for (p = proc_table; p < proc_table+NR_TASKS; p++) {
            if (p->ticks > greatest_ticks) {
                greatest_ticks = p->ticks;
                p_proc_ready = p;
            }
        }

        if (!greatest_ticks) {
            for (p = proc_table; p < proc_table+NR_TASKS; p++) {
                p->ticks = p->priority;
            }
        }
    }
}

```

四：研究成果

| 项目 | 级别 | 类型 |
|--------|------|----|
| 贪吃蛇 | 用户级 | 游戏 |
| 逃离迷宫 | 用户级 | 游戏 |
| 进程管理器 | 系统级 | 管理 |
| 多级反馈队列 | 内核修改 | 结构 |

1.两个用户级应用

- 逃离迷宫

```
*****
*                                     *
*      * * *      A stupid OS      *
*      *  =  =  *      Welcome!    *
*      *    *    *      by 1352892 tanjingru  *
*      *  O  *    *      AND      *
*      * * *    *      1352888 linyueqiang  *
*                                     *
*****

[root@localhost ~]# help
=====
Command List      :
1. process        : A process manage,show you all process-info here
2. filemng        : Run the file manager
3. clear          : Clear the screen
4. help           : Show this help message
5. chat           : Do you want to talk to me?0.0
6. runttt         : Run a small game on this OS
6. labyrinth      : Run a labyrinth game
7. information     : Show students' information
=====
```

进入界面后，输入help可以查看到相关的指令操作，输入labyrinth 即可执行逃离迷宫游戏。

游戏简介:扮演一个小人,逃离一个迷宫

游戏说明:小人会一直运动,不能停下来,玩家只能改变小人运动的方向,小人碰到墙后游戏结束。

游戏操作:上w下s左a右d

游戏界面预览:



● 贪吃蛇

输入snake，进入贪吃蛇游戏。
这是游戏的初始界面。



游戏简介：就是一个普通的贪吃蛇游戏

游戏说明：就是一个普通的贪吃蛇游戏，随着吃到更多的目标，速度会加快。

游戏操作：一开始玩家处于静止状态，可以按a、w、s、d键来开始游戏，游戏中用这四个键来操控蛇的走向。

游戏界面预览：

#表示目标

@表示蛇头

0表示蛇身



2. 一个系统级应用

进程管理器

开启方式：进入主界面后，输入process

功能：暂停进程，恢复进程，结束进程，查看所有进程状态

命令：resume a/b/c pause a/b/c kill b/c up a/b/c

几点说明：

1) 进程B和进程C没有做太多的定义，只是不停打印字符b和字符c，我们可

以通过观察字符b和c是否不停的出现来判断进程的情况。

2) 一个进程被pause后还可以被resume，如果被kill之后，就不能被resume了,也就是被彻底摧毁了。

界面展示

```
=====Process Manager=====
=====  myID      |  name      |  spriority  |  running?  =====
=====  4         |  TestA     |  4          |  1         =====
=====  5         |  TestB     |  1          |  0         =====
=====  6         |  TestC     |  1          |  0         =====
=====
tips: use 'pause a/b/c' command -> you can pause one process
      use 'resume a/b/c' command -> you can resume one process
=====
[root@localhost ~]#
```

具体实现的细节：

resume 以及 pause 是通过使用 proc 结构体 p_flags 来实现进程的阻塞与畅通， kill则直接通过引用一个第三方变量，使得resume时不能生效来实现进程的永久阻塞(死亡)；

3.对内核的一些修改

- 进程调度的修改(将优先级调度改为多级队列反馈调度)

在参考书中，使用的调度方法是比较简单的优先级调度，意思是说，每当发生调度的时候，选择一个优先级最高的进程为就绪进程，然后一直执行只到这个进程执行完(本书有一点不同，他不是执行完，优先级越高的进程能够执行比较长的时间)，在书中，一般设定优先级为150和5能够很明显地看出两个进程运行的频率的差别，但是随之程序的不断扩张，如果一个系统的进程成百上千个，说不定有的进程就出现需要100万个时间片，那么就必须设定他的优先级为100万，这样一来，其余只要1个时间片的进程就会发生饥饿现象，为此我引入了一个新的变量run_count，用来记录进程运行过的时间片数量，每次时钟中断的时候，让run_count++，如果run_count

等于 $\text{priority}/2$ 的话(运行若干时间片后没能成功),则降低他的优先级为 $\text{priority}/2$,并把ticks设为0(不提供时间片让其运行).

代码如下:

调度实现部分:
(clock.c)

```
PUBLIC void clock_handler(int irq)
{
    if (++ticks >= MAX_TICKS)
        ticks = 0;

    if (p_proc_ready->ticks)
    {
        p_proc_ready->ticks--;
    }
    //*****

    //every time runs, count add one
    p_proc_ready->run_count++;
}

//if has run for 20 time,lowing the priority
if(p_proc_ready->run_count >= p_proc_ready->priority/2){
    p_proc_ready->ticks = 0;
    p_proc_ready->priority = p_proc_ready->priority/2;
    if(p_proc_ready->priority == 0){
        p_proc_ready->priority = 1;
    }
    p_proc_ready->run_count = 0;
}
//*****
```

p_proc->run_count = 0;

声明与初始化
(proc.h)
(main.c)

```
struct proc {
    struct stackframe regs;    /* process registers saved in stack frame */
    unsigned int ldt_sel;      /* gdt selector giving ldt base and limit */
    struct descriptor ldts[LDTS_SIZE]; /* local descriptors for code and data */

    int ticks;                /* remained ticks */
    int priority;

    int run_count;

    int pid;                  /* process id passed in from MM */
    char name[16];            /* name of the process */

    int run_state;

    int p_flags;              /**
                               * process flags.
                               * A proc is runnable iff p_flags==0
                               */
}
```