西北工业大学 操作系统实验 实验报告

班号：10012006　　姓名：夏卓　　学号：2020303245

实验日期：2022/11/6　　实验名称：创建 GeekOS 内核线程

一、实验目的

1.熟悉 GeekOS 实验环境，了解操作系统引导过程。

2.分析 GeekOS 中内核代码，掌握内核线程的实现原理和中断管理方法。

二、实验要求

1.创建一个线程，实现从键盘接收一个按键，并在屏幕上显示。

2.创建两个线程，分别打印输出不同的字符串信息。观察并分析结果。

3.自行设计个性化的内核线程，给出运行结果。

4.回答课后思考题。第一章第 1 题、第三章第 1 题，第四章第 1 题，第五章 1、2 题，第六章 1、4 题。

三、实验过程及结果

1.创建一个线程，实现从键盘接收一个按键，并在屏幕上显示。

在 main 函数前实现一个名为 print_key 的函数，首先我们选用了 Wait_For_Key 接口获取键盘输入，由于按下和松开按键都会产生扫描码，因此我们需要屏蔽掉其中一个，接着判断 keycode 是否为 Ctrl+d，若是则直接退出，否则需要输出对应字符。这里由于 keycode 是 16 位无符号数，且低 8 位是按键对应的建码，因此直接按字符输出其低 8 位即可，另外这里增加了判断输入是否为换行符的语句，否则无法进行换行操作。

具体代码如下：

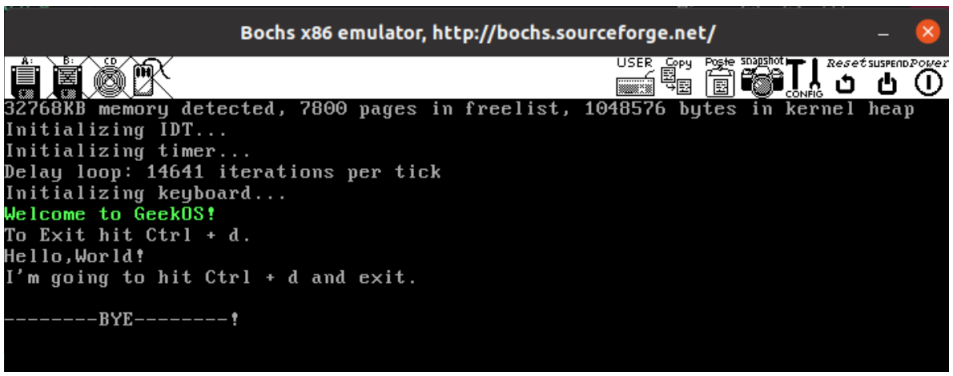
```
void print_key(ulong_t arg){
    Print("To Exit hit Ctrl + d.\n");
    Keycode keycode;
    while(1){
        keycode = Wait_For_Key();
        if(!(keycode & KEY_RELEASE_FLAG)){
            if(keycode == ('d'|KEY_CTRL_FLAG)){
                Print("\n--------BYE--------!\n");
                Exit(1);
            }else{
                char ascii = keycode & 0xff;
                Print("%c",ascii == '\r' ? '\n' : ascii);
            }
        }
    }
}
```

接着在 main 函数中把宏 TODO 注释掉，并使用 Start_Kernel_Thread 增加一个线程，该线程即是 print_key 函数，优先级设置为 PRIORITY_NORMAL。

```
    // TODO("Start a kernel thread to echo pressed keys and print counts");

    Start_Kernel_Thread(print_key,0,PRIORITY_NORMAL,true);
```

编译运行，可以发现成功实现了按键中断，并可以在屏幕中显示字符，结果如下图所示：



2.创建两个线程，分别打印输出不同的字符串信息。观察并分析结果。

在 main 函数前添加两个函数分别打印字符 A 与 B:

```
void print_A(ulong_t arg){
    while(1){
        Print("A");
    }
}

void print_B(ulong_t arg){
    while(1){
        Print("B");
    }
}
```

然后在 main 函数中使用相同方法增加这两个线程：

```
Start_Kernel_Thread(print_A,0,PRIORITY_NORMAL,true);
Start_Kernel_Thread(print_B,0,PRIORITY_NORMAL,true);
```

运行结果如下，可以看到屏幕上一会儿输出 A，一会儿输出 B：

3.自行设计个性化的内核线程，给出运行结果。

结合键盘中断和背景颜色设置，我设计了一个切换背景颜色的内核线程。

与实验 1 类似，通过 Wait_For_Key 监听键盘输入，当检测到关键词时设置不同的背景颜色。另外又增加了多线程抢占的测试，同时设置多个线程设置不同的背景颜色并进行输出，惊奇的发现产生了类似波浪般的效果，同时使用了延迟以增强视觉效果

具体实现如下：

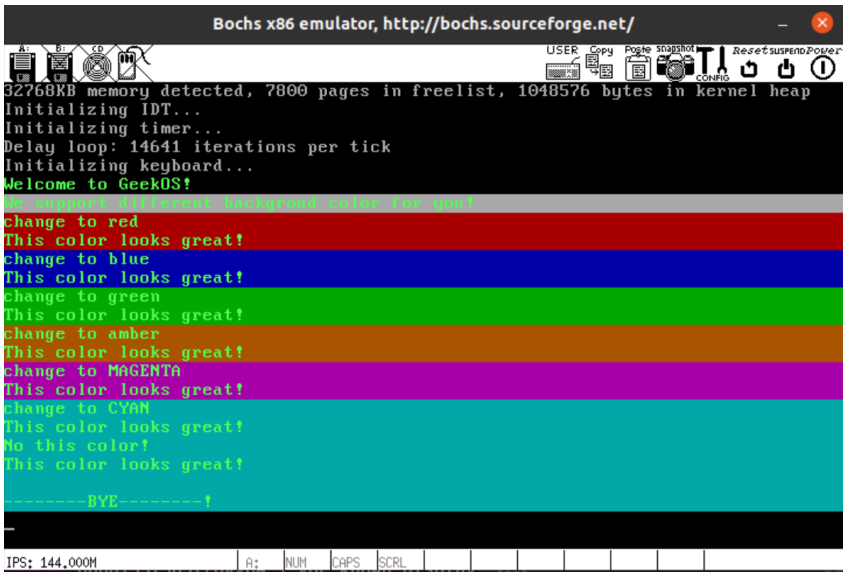
```
void showColor(){
    Print("This color looks great!\n");
}

void changeColor(ulong_t arg){
    Set_Current_Attr(ATTRIB(GRAY, GREEN|BRIGHT));
    Print("We support different backgroud color for you!\n");
    Keycode keycode;
    while(1){
        keycode = Wait_For_Key();
        if(!(keycode & KEY_RELEASE_FLAG)){
            if(keycode == ('d'|KEY_CTRL_FLAG)){
                Print("\n--------BYE--------!\n");
                Exit(1);
            }else{
                char ascii = keycode & 0xff;
                switch (ascii)…
                showColor();
            }
        }
    }
}
```

Switch 部分就是针对不同输入切换不同背景颜色，只展示部分，其他类似：

```
switch (ascii)
{
case 'r': Set_Current_Attr(ATTRIB(RED, GREEN|BRIGHT));
          Print("change to red\n");
    break;
case 'b': Set_Current_Attr(ATTRIB(BLUE, GREEN|BRIGHT));
          Print("change to blue\n");
    break;
```

实现效果如下：



当输入其他字符时将产生”waves”颜色波浪：

```
default:
        Print("The waves is comming!!!\n");
        random();
    break;
}
```

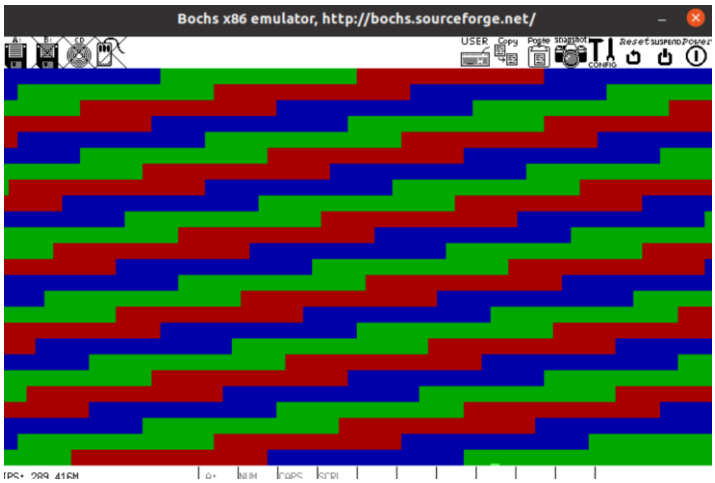
调用的 random 函数的作用就是创建三个抢占式线程：

```
void green(ulong_t arg){
    while (1)
    {
        int delay = 10000;
        while(delay--);
        Set_Current_Attr(ATTRIB(GREEN, GREEN|BRIGHT));
        Print(" ");
    }

}

void random(){
    Start_Kernel_Thread(blue,0,PRIORITY_NORMAL,true);
    Start_Kernel_Thread(green,0,PRIORITY_NORMAL,true);
    Start_Kernel_Thread(red,0,PRIORITY_NORMAL,true);
}
```

实现效果如下：



四、实验分析

实验 1 涉及键盘中断处理，以及内核线程的创建问题。经过前期的准备，系统已经向我们提供了 Start_Kernel_Thread 方法来创建内核级线程，只需自行创建一个函数然后将函数指针传递给它即可。另外 GeekOS 也向我们提供了 Read_Key 和 Wait_For_Key 两个键盘接口，分析可知后者性能更佳，因为当没有键盘输入时，线程会自动进入等待队列，直到被唤起。最后根据 screen.h 文件可以得知特殊键的定义，如此便好判断输入是否为 Ctrl+d，最后，为了不重复输出，我们需要屏蔽掉松开时产生的扫描码。

实验 2 涉及多线程抢占问题，通过实验可以看出两个 print 线程在进行抢占式调度，这里发生的中断是时钟中断，当时间片耗费完时，内核将会选择另一个线程运行。

实验 3 中我结合了实验 1 和实验 2 的内容，完成了键盘中断处理和多线程抢占的功能，并且也惊喜的发现产生了类似波浪的效果，还挺炫酷。

五、所遇问题及解决方法

在实验 1 中一开始并不清楚如何判断输入字符以及如何消除重复输出的问题，后来在仔细阅读实验书知道了键值都定义在了 screen.h 文件中，其中包含了 KEY_RELEASE_FLAG，也顺带解决了如何屏蔽松开键盘时的扫描码。实验 3 中对于个性化设计这种自由命题，感觉没什么思路，最后决定结合前两个实验的考察内容进行编写，在编写过程中也惊奇的发现抢占式背景设置带来了一个颜色波浪视觉冲击，也算是意外收获。

六、思考与练习

1. 引导扇区的标志是 0x55AA，为何代码中写入的却是 dw0xAA55?

因为 x86 系统是小端字节序系统，即低字节数据存放在内存低地址处，高字节数据存放在内存高地址处。

2. fd_boot 主要完成了哪些工作？

fd_boot 运行时，首先将自身复制到 0x90000 处，然后跳转到 0x90000 处执行。它完成的主要功能是把 setup 代码和 kernel 代码载入到内存 0x90200 和 0x10000 处，最后跳转到内存中的 0x90200 处执行 setup 代码

3. 系统是如何进入 main 函数开始执行的？

setup 模块的最后一条指令是 jmp KERNEL_CS:ENTRY_POINT，而 ENTRY_POINT 正是 kernel 中的 main 函数，因此系统在执行完 setup 模块后会跳转到 main 函数开始执行。另外，setup 还将机器系统数据放入了堆栈中，作为参数（boot_info）传给了 main 函数，并为系统进入保护模式做了一系列工作。

4. 中断入口表 g_entryPointTableStart 与中断向量表 g_interruptTable 是否相同？各自的作用分别是什么？

中断入口表和中断向量表不同，前者是中断号与该中断类型相对应的中断服务程序入口之间的连接表，用于根据中断号跳转到相应的中断处理函数去（首先会跳转到通用中断处理函数）；后者是存储各种中断处理函数的表，用于处理系统运行时可能产生的各种中断。

5. 中断处理前后为何要保护、恢复寄存器？

因为中断处理结束后可能会跳转到原来的程序继续执行，而中断处理过程中可能会改变寄存器中的值，因此为了使返回原程序后能正常运行，需要保护原程序的上下文，其中就包括了寄存器。

6. 系统的第一个线程是什么？初始化它的时候为何不调用 Setup_Kernel_Thread?

系统的第一个线程是 main 函数对应的线程。Setup_Kernel_Thread 的作用是将创建的线程的上下文压入到堆栈中，而对于 main 函数来说，由于其是 kernel 的入口函数，这一步已经由 setup 模块完成了，故不需要再调用 Setup_Kernel_Thread。

7. Switch_To_Thread 中调整堆栈的目的是什么？

由于将来系统会在其他的线程上下文中使用 iret 切换回现在的线程，而 iret 会还原寄存器 cs 和 eflag，这是与 ret 不同的地方，因此需要调整堆栈以将 eflags 和 cs 压栈。