本科生实验报告

实验课程:_____操作系统原理实验_____

实验名称:_____实验 9 Course Projects_____

专业名称:_____计算机科学与技术(超级计算方向)_____

学生姓名:_____林天皓_____

学生学号:_____18324034_____

实验地点:_____

实验成绩:_____

报告时间:_____2021.7.16_____

1 对 COW 的尝试

一开始，尝试实现 copy-on-write 方法，但是在实现过程中遇到的诸多问题。

1.初步测试，直接将子进程的页目录表中储存的物理地址写入为父进程相同的物理地址。在 fork 中

```
            int pte = pageTableVaddr[j];
            brk();
            asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间
                pageTableVaddr[j] = pte ;//直接复制父进程的页表
```

运行测试，发现父进程运行正常，而切换到子进程后模拟器闪退出错。

进入 debug 查看，现象为由于父进程在返回 fork 函数的过程中，会改变栈地址，而这部分栈地址子进程同样在使用，显然切换到子进程时候，该栈的状态已经被父进程所破坏，子进程在执行 ret 指令时候就会跳转到错误的地址上去。造成异常地址访问的错误。

2.修改该部分函数变为只读，加入缺页处理函数。

```
            int pte = pageTableVaddr[j];
            brk();
            asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间
                pageTableVaddr[j] = pte & 0xfffffffd ;//直接复制父进程的
页表，只读
```

在中断中添加缺页处理函数

```
asm_get_cr2:
    mov eax,cr2
    ret
```

```
extern "C" void c_pagefault_interrupt_handler()
{
    int t=asm_get_cr2();
    printf("pagefault at phyaddress: %x\n",t);
    memoryManager.copy_on_write(t);
    //asm_halt();
}
```

这部分通过 cr2 寄存器获取缺页发生的物理地址，传入 copy_on_write 函数进行

写时拷贝操作。写时拷贝的函数如下

```
int MemoryManager::copy_on_write(const int vaddr){
    int nvaddr=vaddr&0xfffff000;
    int buf = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::US
ER, 1);
    if (!paddr)
    {
        return false;
    }
    memcpy((void * )buf, (void *)nvaddr, PAGE_SIZE);
    connectPhysicalVirtualPage(nvaddr,paddr);
    memcpy((void * )nvaddr, (void *)buf, PAGE_SIZE);
    printf("v %x p %x",nvaddr,paddr);
    return true;
}
```

其中先计算缺页发生的物理地址对应的虚拟地址，申请一页新的空间，然后将物理地址中的内容拷贝进入新申请的页中，最后通过 connectPhysicalVirtualPage 将该虚拟地址与物理地址相连，并同时改变页表访问权限。

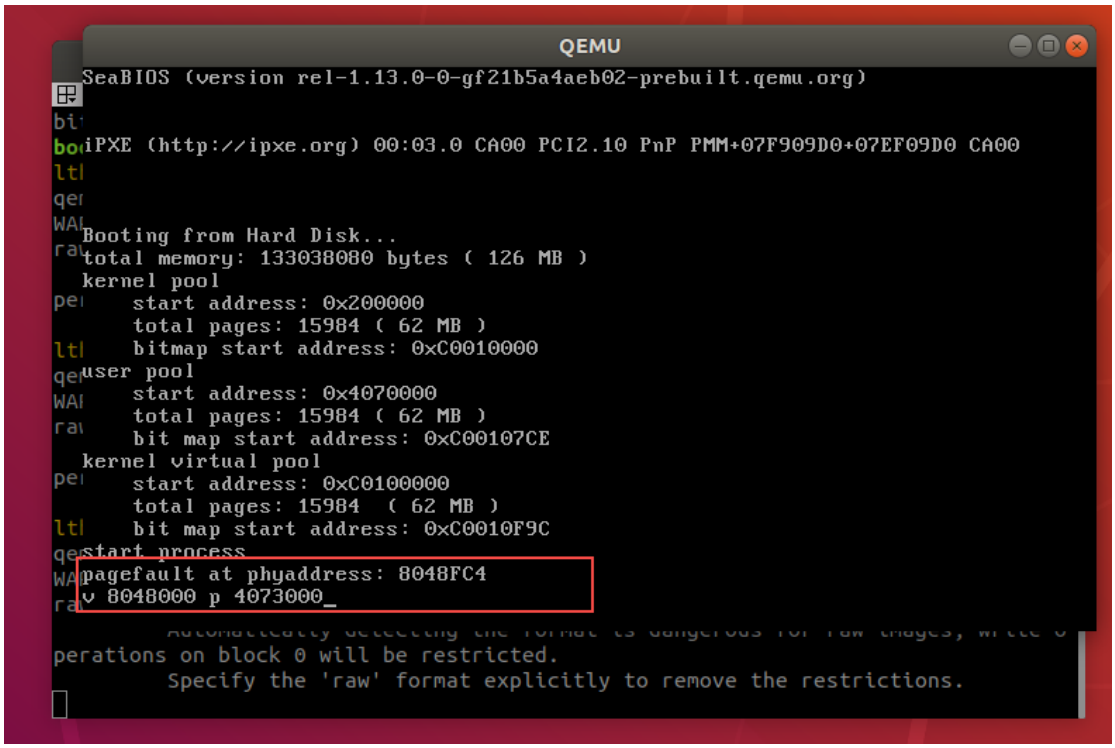

图 1-出现缺页的地址

可见在运行过程中我们检测到了缺页错误，并输出了产生缺页的物理地址和对应

的虚拟地址。但是拷贝过程不能正常执行。

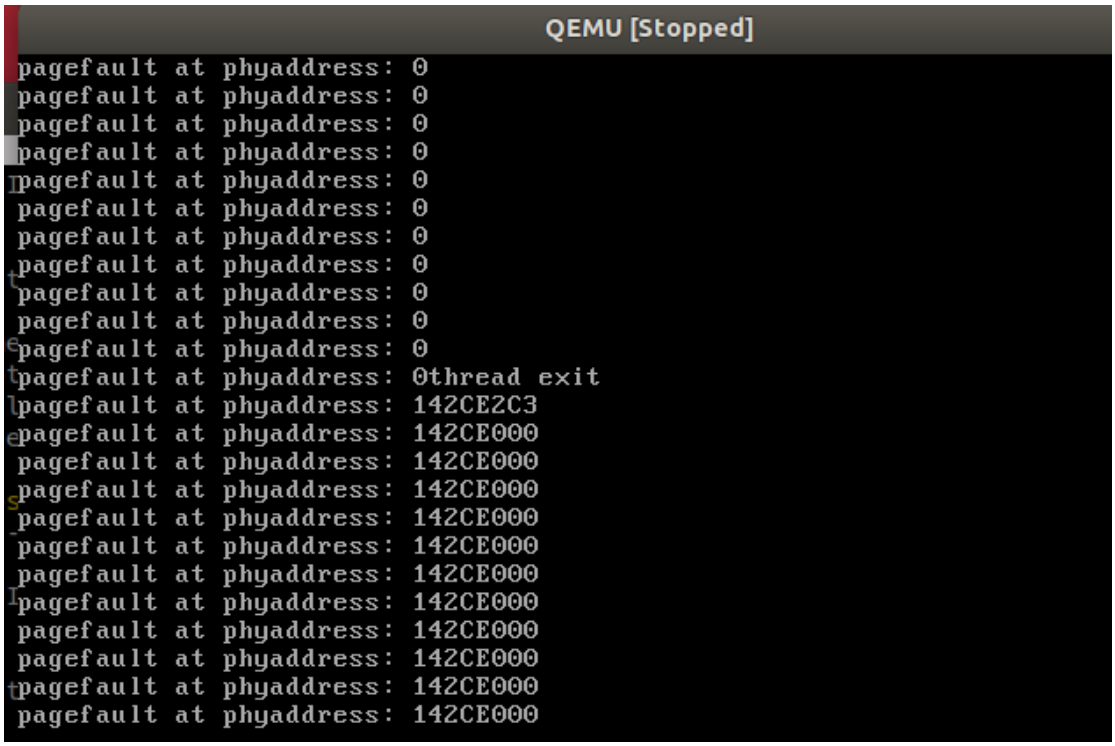

图 2-缺页发生嵌套错误

其中有几点问题，一是该处已经启用了页表，不能直接将虚拟地址对应的物理地址中的内容写入另一个物理地址。这样 memcpy 是错误的，造成的结果为每一的 copy 过程会发生一次页错误，逐层嵌套，最终层数过多栈错误，模拟器出现闪退。

3.进一步的考虑，放弃使用写时拷贝

我们只要解决了上面 2 中提到的问题就可以了吗？不是这样的，经过再次检查 fork 的运行流程，其实父进程和子进程所对应的栈的内容都是不一致的。父进程在 fork 返回后会一步步返回到原来执行 fork 的地方继续执行程序，而子进程我们通过修改子进程中的栈空间，使得子进程运行时候，先执行 asm_start_process，然后才和父进程一样继续从 fork 处运行，所以直接拷贝父进程的栈空间，子进程仍然不能正常运行。因此放弃使用写时拷贝方法完成 lab9。

2 对 shell 的实现

1.键盘输入的实现

在 interrupt 设置键盘中断

```
void InterruptManager::setKeyBoardInterrupt(void *keyboard_handler)
{
    setInterruptDescriptor(IRQ0_8259A_MASTER + 1, (uint32)keyboard_hand
ler, 0);
}
```

键盘中断的中断号为时钟中断的中断号+1

在 setup 中建立键盘中断与键盘中断处理函数的对应关系

```
extern "C" void setup_kernel()
{

    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handle
r);
    interruptManager.setKeyBoardInterrupt((void *)asm_keyboard_interrup
t_handler);


```

建立键盘中断处理函数

```
asm_keyboard_interrupt_handler:
    pushad

    cli ;应禁止中断

    mov al,  0x61
    out 0x20, al  ;PIC0_OCW2
    in  al,  0x60  ;从键盘读入按键扫描码

    call c_keyboard_interrupt_handler

    mov               al , 0x20  ;告诉硬件,中断处理完毕,即发送 EOI 消息
    out               0x20 , al
    out               0xa0 , al

```

```
    sti
    popad
    iret
```

需要注意需要修改 stdio‘实现退格键的处理。’

```
        case '\b':
            row = getCursor() / 80;
            col = getCursor() % 80;

            if (col > 0)
            {
                moveCursor(row * 80+col-1);
                print(' ');
                moveCursor(row * 80+col-1);
            }
            break;
```

c 语言读取中断码并写入键盘缓冲区

```
extern "C" void c_keyboard_interrupt_handler()
{
    char t = asm_return_eax();
    for (int i = 0; i < 45; i++)
    {
        if (t == key_labels[i].value)
        {
            if (key_labels[i].label_base != '\b')
            {
                //printf("get key:%x  char:%c\n",t,key_labels[i].label_
base);
                key_buf[key_buf_size++] = key_labels[i].label_base;
                printf("%c", key_labels[i].label_base);
            }
            else
            {
                if (key_buf_size > 0)
                {
                    key_buf_size -= 1;
                    printf("%c", key_labels[i].label_base);

                }
            }
        }
    }
```

```
}
```

其中使用一个键盘码表储存中断产生的触发码对应的字符

```
#ifndef KEY_H
#define KEY_H
struct key_label_map {
  char value;
  char label_base;
};
struct key_label_map key_labels[] = {
{0x2,'1'},
{0x3,'2'},
{0x4,'3'},
{0x5,'4'},
{0x6,'5'},
{0x7,'6'},
{0x8,'7'},
{0x9,'8'},
{0xa,'9'},
{0xb,'0'},
以下省略
```

以上关于键盘的部分处理完毕，下面实现 shell 的部分。

2.实现 shell 的部分

定义 shell 相关全局变量

```
char key_buf[200];
int key_buf_size = 0;
char dir[60] = "/";
char subscript[30] = "";
char username[30] = "avarpow";
char PC[30] = "";
```

创建 shell 进程，无限循环，轮询键盘缓冲区，若读取到最后一个字符为回车键代表用户输入了一条指令，开始解析。

实现了指令为 ls，su，cd，help，shutdown。

其中与目录与文件相关的通过直接定义文件系统结构体模拟，分别对应路径与文件名和文件名长度。

```
struct FILE
{
    char filename[20];
    int namesize;
};
FILE f[10] = {
    {"/etc/", 5},
    {"/var/", 5},
    {"/lib/", 5},
    {"/root/", 6},
    {"/home/", 6},
    {"/home/avarpow/", 14},
    {"/home/avarpow/1.txt", 19},
    {"/home/avarpow/2.txt", 19},
    {"/home/avarpow/3.txt", 19},
    {"/home/avarpow/4.txt", 19}};
```

1.ls

通过get_subscript获取用户输入的路径,对比文件目录中与之相同的目录并输出,

特别判断用户输入为空的情况。

```
if (strcmp(key_buf, "ls", 2))
{
    int k = get_subscript(2);
    printf("shell ls\n");
    if (k <= 0)
    {
        printf("NO argument\n");
        key_buf_size = 0;
        printf("%s @ %s> ", username, dir);
        continue;
    }
    for (int i = 0; i < 10; i++)
    {
        if (k <= f[i].namesize && strcmp(subscript, f[i].filename, k))
        {
            printf("%s    ", f[i].filename);
        }
    }
    printf("\n");
}
```

2.su

通过 get_subscript 获取用户输入的用户名，将当前用户名更改为目标用户名，特别判断用户输入为空的情况。

```
if (strcmp(key_buf, "su", 2))
{
    int k = get_subscript(2);
    if (k <= 0)
    {
        printf("NO argument\n");
        key_buf_size = 0;
        printf("%s @ %s> ", username, dir);
        continue;
    }
    strcpy(username, subscript, 30);
}
```

3.cd

通过 get_subscript 获取用户输入的文件名，判断文件目录中是否存在，若存在还需要判断是否是一个文件而不是目录，特别判断用户输入为空的情况。

```
else if (strcmp(key_buf, "cd", 2))
{
    int k = get_subscript(2);
    printf("shell cd\n");
    if (k <= 0)
    {
        printf("NO argument\n");
        key_buf_size = 0;
        printf("%s @ %s> ", username, dir);
        continue;
    }
    int flag = 0;
    for (int i = 0; i < 10; i++)
    {
        if (k <= f[i].namesize && strcmp(subscript, f[i].filename, k))
        {
            if (f[i].filename[f[i].namesize - 1] == '/')
            {
                strcpy(dir, f[i].filename, f[i].namesize);
                key_buf_size = 0;
                printf("%s @ %s> ", username, dir);
                continue;
```

```
            }
            else
            {
                printf("%s is a file, not a directory");
            }
        }
    }
    printf("\n");
}
```

4.shutdown

经过一段假装的延时关机过程后，禁止键盘中断，并且程序进入死循环，不在接受响应。

```
if (key_buf_size > 8)
{
    if (strcmp(key_buf, "shutdown", 8))
    {
        printf("shutdown");
        for (int i = 0; i < 5; i++)
        {
            printf(".");
            int t = 99999999;
            while (t--)
                ;
        }
        printf("OK\n");
        interruptManager.disableKeyboardInterrupt();
        asm_halt();
    }
}
```

Shell 测试

```
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x107CE
start shell thread
avarpow @ /> help
=======help=====
1.ls
2.cd
3.su
4.shutdown
5.help
avarpow @ />
```

图 3-Help 显示支持的命令

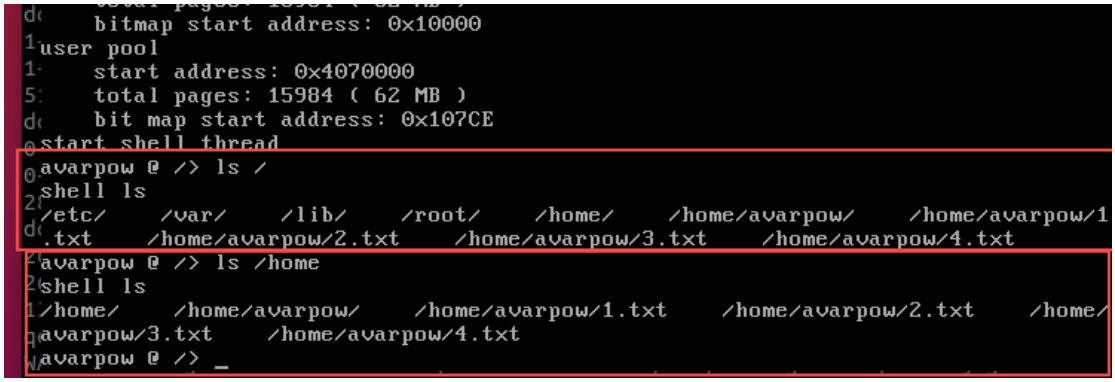

图 4-ls 测试

第一个 ls /命令

现象为显示所有文件与目录

第二个 ls /home 命令

现象为显示所有文件名前缀为/home 的文件共 6 个

cd 测试

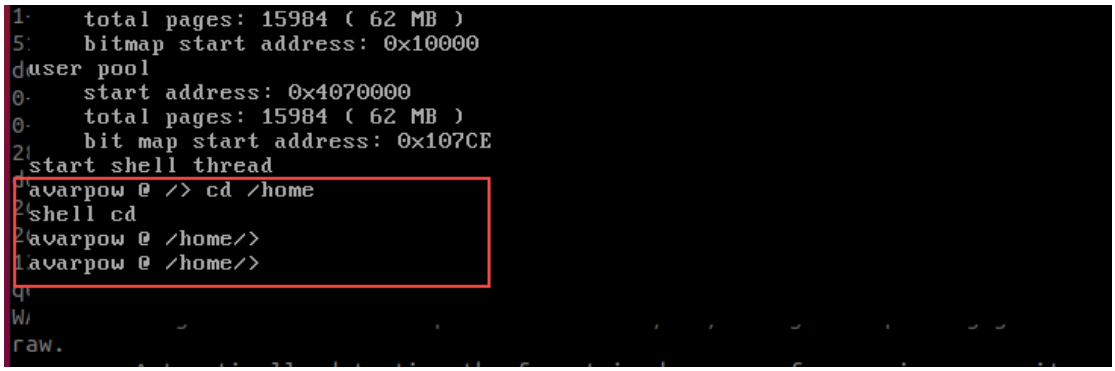

图 5- cd 测试 1，正常切换路径

cd home

将路径切换为 home 路径

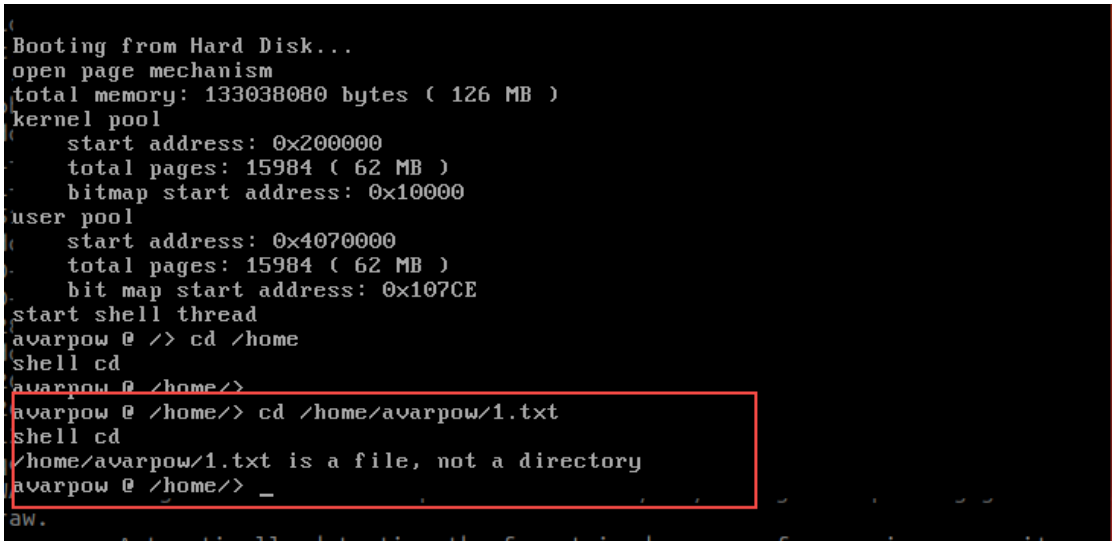

图 6-cd 测试，路径为文件

cd /home/avarpow/1.txt

现象为显示该路径表示一个文件，不是一个路径。

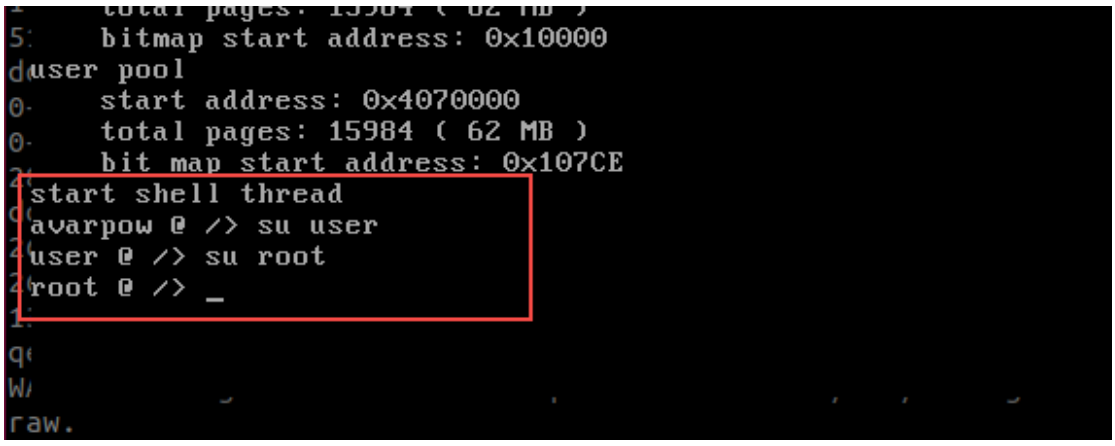

图 7-su 测试切换用户

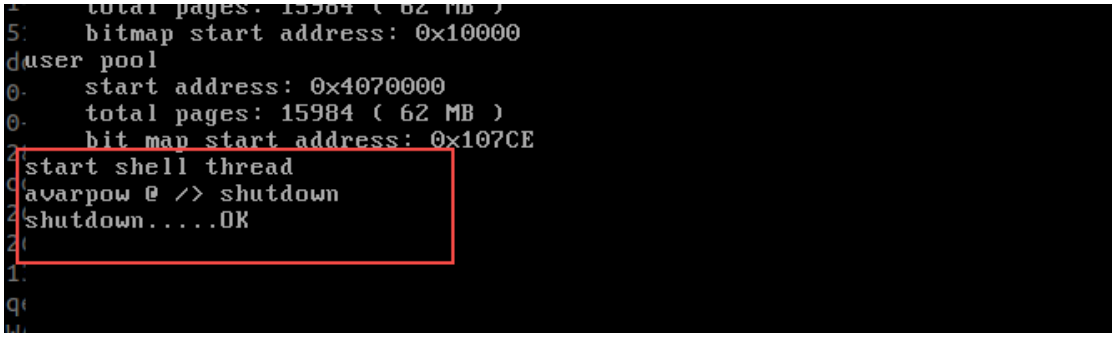

图 8-shutdown 测试

使用 shutdown 命令，经过 3 秒钟显示 OK，之后任意点击键盘均无效果。

实验感想

本次实验是最后一次实验。首先尝试对写时拷贝做一个实现，在实现的过程中遇到了很多的困难，且这些错误都是内存错误，会导致模拟器的直接闪退，不能通过 debug 的方式查找错误。后来通过对 fork 过程中父进程与子进程的运行栈的步步跟踪，与手动更改进程调度器切换进程返回顺序，发现了第二个执行的线程就会卡死，是因为共用同一片物理地址时，第二个线程需要使用的栈已经被第一个线程 pop 掉，返回到了无效的地址空间导致闪退。关键在于子进程本来就不是和课本中所描述的那样“一致”，而是有一些我们在 fork 过程中做的 trick 使得第二个进程需要经过 asm_start_progress 才回到和父进程一样的运行状态。因此就算把这部分按照原样拷贝父进程也会导致运行错误。于是放弃了写时拷贝的实现。

在 shell 的实现中，由于时间仓促，没能实现本想做一个支持分屏的多窗口 shell，文件系统也仅使用结构体模拟，主要在于对键盘扫描码的理解，通过开启键盘中断完成键盘的输入与键盘处理缓冲区的处理工作。