lab2 report

OS的感应与反应

> 任课教师：叶保留　　助教：尹熙喆，水兵

学院	计算机科学与技术系	专业	计算机科学与技术
学号	201220154	姓名	王紫萁
Email	201220154@smail.nju.edu.cn	开始/完成时间	2022. 3.17/2022.3.25

1. Exercieses

exercise 1

> exercise1：既然说确定磁头更快（电信号），那么为什么不把连续的信息存在同一柱面号同一扇区号的连续的盘面上呢？（Hint：别忘了在读取的过程中盘面是转动的）

如果按盘面优先存储，由于磁盘是转动的，而且数据只存在于柱面的一个部分中，所以读完前一盘面的数据，要想读下一盘面的数据需要等待磁盘转动几乎一圈才能继续由下一个磁头读取，速度反而没有提升。

exercise 2

> exercise2：假设CHS表示法中柱面号是C，磁头号是H，扇区号是S；那么请求一下对应LBA表示法的编号（块地址）是多少（注意：柱面号，磁头号都是从0开始的，扇区号是从1开始的）。

$$LBA = C \times 磁头总数 \times 每个磁道的扇区数 + H \times 每个磁道的扇区数 + S - 1$$

其中第一项表示当前柱面号前的所有柱面上的所有扇区的总数，直观上是该柱面内的圆柱体的体积；第二项是在该柱面上，目标磁道之前的所有扇区的总数，直观上是目标磁道之前的圆柱面的面积；定位到该磁道后，S-1就是该磁道上的扇区位置-1

exercise 3

> exercise3：请自行查阅读取程序头表的指令，然后自行找一个ELF可执行文件，读出程序头表并进行适当解释（简单说明即可）。

```
程序头：
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  INTERP         0x000154 0x00000154 0x00000154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00720 0x00720 R E 0x1000
  LOAD           0x000ed8 0x00001ed8 0x00001ed8 0x00130 0x00134 RW  0x1000
```

```
  DYNAMIC        0x000ee0 0x00001ee0 0x00001ee0 0x000f8 0x000f8 RW  0x4
  NOTE           0x000168 0x00000168 0x00000168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0005e8 0x000005e8 0x000005e8 0x0003c 0x0003c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000ed8 0x00001ed8 0x00001ed8 0x00128 0x00128 R   0x1

 Section to Segment mapping:
  段节...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym
.dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got
.text .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got

```

如上是在32位下编译得到的ELF文件程序头表，除了 `LOAD` 段，还有动态连接需要的 `DYNAMIC` 段，程序解释器 `INTERP` 段。对于我们重点关注的 `LOAD` 段，如第7行表示将磁盘上相对ELF文件起 `offset = 0xed8` 的地方读取 `FileSiz = 0x130` 个字节到逻辑地址 `VirtAddr = 0x1ed8` 的地方，对应的物理地址为 `PhysAddr = 0x1ed8` 。且内存大小 `MemSiz = 0x134` 说明 `.bss` 节有4字节大小，并采取4KB对齐的存储方式

exercise 4

> exercise4：上面的三个步骤都可以在框架代码里面找得到，请阅读框架代码，说说每步分别在哪个文件的什么部分（即这些函数在哪里）？

1. 初始化串口输出（ `initSerial` ）--> kernel/kernel/serial.c
2. 初始化中断向量表（ `initIdt` ） --> kernel/kernel/idt.c
3. 初始化8259a中断控制器（ `initIntr` ）--> kernel/kernel/i8259.c
4. 初始化 GDT 表、配置 TSS 段（ `initSeg` ）--> kernel/kernel/kvm.c
5. 初始化VGA设备（ `initVga` ） -->kernel/kernel/vga.c
6. 配置好键盘映射表（ `initKeyTable` ） --> kernel/kernel/keyboard.c
7. 从磁盘加载用户程序到内存相应地址（ `loadUMain` ）-->kernel/kernel/kvm.c
8. 进入用户空间（ `enterUserSpace` ） --> kernel/kernel/kvm.c

exercise 5

> exercise5：是不是思路有点乱？请梳理思路，请说说“可屏蔽中断”，“不可屏蔽中断”，“软中断”，“外部中断”，“异常”这几个概念之间的区别和关系。（防止混淆）

可屏蔽中断和不可屏蔽中断是对立的概念，表示当前的中断是否可以被忽视，如果忽视会导致系统panic的中断就是不可屏蔽中断，由NMI线请求，否则由于系统自身运行的需要必须屏蔽相应的中断为可屏蔽中断，由IRQ线请求。而软中断是软件可以发出中断信号告诉CPU要为他提供服务的中断，通常是可屏蔽中断。而外部中断是外部设备请求的服务，通常也是可屏蔽的（否则可能会发生逻辑上的错误或者和预期不符的结果，当然也偶不可屏蔽的如电源等）。异常是相对中断来说的，是操作系统不得不处理的事件，通常为不可屏蔽中断。

exercise 6

> exercise6：这里又出现一个概念性的问题，如果你没有弄懂，对上面的文字可能会有疑惑。请问：IRQ和中断号是一个东西吗？它们分别是什么？（如果现在不懂可以做完实验再来回答。）

不是，IRQ是中断请求号，需要外部设备送到8259A处理。而中断向量号是用来查找IDT表项的索引，且有中断向量号=起始向量号 + IRQ（如本次实验的键盘就是0x20 + 1），会将中断向量号送入CPU，由CPU负责根据中断向量号查找门描述符

exercise 7

> ...由于一些特殊的原因，这三个寄存器的内容必须由硬件来保存。
>
> exercise7：请问上面用斜体标出的"一些特殊的原因"是什么？（Hint：为啥不能用软件保存呢？注：这里的软件是指一些函数或者指令序列，不是gcc这种软件。）

虽然软件可以使用 `pushf` 将 `EFLAGS` 压栈，使用 `push %eip` 将eip压栈， `push %cs` 将cs压栈，但是需要注意的是这些指令只能写死在OS中，而OS并不知道外部中断和异常何时到来，而硬件才可能在运行的过程中检测到他们，于是保存这三个寄存器的任务就由 `scheduler` 动态地完成了。转入 `do_irq` 时软件才确定此时中断已经到来，需要手动 `pusha` 保存其余的寄存器上下文信息。

exercise 8

> 如果选择把信息保存在一个固定的地方，发生中断嵌套的时候，第一次中断保存的状态信息将会被优先级高的中断处理过程所覆盖！
>
> exercise8：请简单举个例子，什么情况下会被覆盖？（即稍微解释一下上面那句话）

比如vga的中断到来，需要在屏幕显示显存上的内容，此时键盘中断到来，如果采用在固定地方保存状态信息，那么此时键盘就会把vga的中断服务例程的状态暂存到其栈帧中，并执行键盘的中断程序，直到返回vga中断服务例程，可以正确恢复现场。然而需要从vga中断服务例程返回到用户进程时才发现原先保存的状态已经被vga中断服务例程的状态覆盖了，导致运行错误。

exercise 9

```
if(GDT[old_CS].DPL < GDT[CS].DPL)    //???
 pop ESP
 pop SS
```

> exercise9：请解释我在伪代码里用"???"注释的那一部分，为什么要pop ESP和SS?

`iret` 对应从特权等级高的内核态返回特权等级更低的用户态，如果状态发生了转换就需要切换到用户堆栈，需要将ESP, SS弹出使用户堆栈还原。否则说明还在内核态执行，不需要栈帧的转换。（对应的在处理中断时也只有在特权等级发生变化时将ss和esp顺序压栈了）

exercise 10

> exercise10：我们在使用eax, ecx, edx, ebx, esi, edi前将寄存器的值保存到了栈中（注意第五行声明了6个局部变量），如果去掉保存和恢复的步骤，从内核返回之后会不会产生不可恢复的错误?

可能会，暂存的步骤类似于保存现场的过程，在陷入内核处理之前，需要将用户的状态保存，因为内核执行的过程中可能会（几乎一定）用到这些寄存器。所以必须保存这些状态才能确保在返回后用户状态恢复正确

exercise 11

> exercise11：我们会发现软中断的过程和硬件中断的过程类似，他们最终都会去查找IDT，然后找到对应的中断处理程序。为什么会这样设计？（可以做完实验再回答这个问题）

方便软件和硬件可以调用公用的中断服务程序，比如可以由软件通过软中断读取键盘缓冲区，键盘也可以通过硬件中断向缓冲区中放入字符。

exercise 12

> exercise12：为什么要设置esp？不设置会有什么后果？我是否可以把esp设置成别的呢？请举一个例子，并且解释一下。（你可以在写完实验确定正确之后，把此处的esp设置成别的实验一下，看看哪些可以哪些不可以）

不设置会使esp指向的位置为0，而高位地址是内核地址空间，在栈增长时可能将内核代码覆盖。如果将esp设置在 `gdt` 的首地址，那用户栈就会覆盖 `gdt` 表项，使得系统崩溃，设置到 `0x1fffff` 正好在 `kernel` 之下不会发生内容覆盖。

exercise 13

```
1  // TODO: 填写kMainEntry、phoff、offset
2   ELFHeader* eh=(ELFHeader*)elf;
3   kMainEntry=(void(*)(void))(eh->entry);
4   phoff=eh->phoff;
5   ProgramHeader* ph=(ProgramHeader*)(elf+phoff);
6   offset=ph->off;
7
8   for (i = 0; i < 200 * 512; i++) {
9       *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
10  }
11  kMainEntry();
```

> exercise13：上面的bootloader为什么错了？

在第4行之前的寻找程序头表的过程是没问题的，问题出装载 `elf` 中可 `load` 的表项上这里并没有按照程序头表中type为 `load` 的表项指示待装载段的首地址和大小装载该段到指定物理地址中去，而是简单粗暴地将整个 `elf` 替换成了elf中程序头表开始的位置往后的内容（甚至复制的大小也是elf文件的大小，简直离谱）。

exercise 14

> exercise14：请查看Kernel的Makefile里面的链接指令，结合代码说明kMainEntry函数和Kernel的main.c里的kEntry有什么关系。kMainEntry函数究竟是啥？

```
1  $(LD) $(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf $(KOBJS)
```

实际上就是猫叫了个咪咪，将 `kEntry` 函数指定为程序的入口，对应于ELF头中的 `Entry point address` ，我们在 `boot loader` 中已经将这个入口地址赋值给了 `kMainEntry` ，调用 `kMainEntry` 就是调用 `kEntry` 。

exercise 15

> exercise15：到这里，我们对中断处理程序是没什么概念的，所以请查看doirq.S，你就会发现idt.c里面的中断处理程序，请回答：所有的函数最后都会跳转到哪个函数？请思考一下，为什么要这样做呢？

跳转到一个很重要的函数（NEMU里就着重理解了一下）

```
asmDoIrq:
    pushal // push process state into kernel stack
    pushl %esp //esp is treated as a parameter
    call irqHandle
    addl $4, %esp //esp is on top of kernel stack
    popal
    addl $4, %esp //interrupt number is on top of kernel stack
    addl $4, %esp //error code is on top of kernel stack
    iret
```

该函数相当于系统负责中断处理的主函数，执行了以下几个重要操作：

- 保存现场： `pusha` ，在此之前是用户态进程传递给中断服务例程的参数（包括系统调用号等）
- 中断执行： `irqHandle` ，负责通过 `eax` 中的系统调用号查找IDT表转到中断服务程序。
- 弹出参数指针： `TrapFrame *`
- 恢复现场： `popa` ，恢复用户程序的状态
- 弹出 `irq number` 和 `error code`

这样做正是能够在用户态和内核态之间无缝衔接。

exercise 16

> exercise16：请问doirq.S里面asmDoirq函数里面为什么要push esp？这是在做什么？（注意在push esp之前还有个pusha，在pusha之前……）

在exercise15中我们提到， `esp` 作为 `TrapFrame` 结构体的首地址，方便 `irqHandle` 函数访问到栈顶之上push过的错误码，中断请求号，以及服务例程需要的参数。

exercise 17

> exercise17：请说说如果keyboard中断出现嵌套，会发生什么问题？（Hint：屏幕会显示出什么？堆栈会怎么样？）

比如说以惊人的手速按下A 再按下B，那么在处理A的键盘中断时，又会去处理B的键盘中断，于是B会先进入键盘缓冲区，而A后进入，结果从我们预期的A → B变成了B → A。

exercise 18

> exercise18：阅读代码后回答，用户程序在内存中占多少空间？

用户程序最大不超过100KB，从0x200000开始存放。

exercise 19

> exercise19：请看syscallPrint函数的第一行：
>
> int sel = USEL(SEG_UDATA);
>
> 请说说sel变量有什么用。（请自行搜索）

`sel` 存放了用户数据段选择子，负责与当前屏幕位置坐标（加上0xb8000起始地址）构成逻辑地址，该逻辑地址对应VGA的内存映射。

exercise 20

> exercise20：paraList是printf的参数，为什么初始值设置为&format？假设我调用printf("%d = %d + %d", 3, 2, 1);，那么数字2的地址应该是多少？所以当我解析format遇到%的时候，需要怎么做？

变长参数的地址通过访问栈顶得到，我们可以定位到format的地址&format，作为函数的第一个参数，拥有所有参数中最低的地址，其余参数存放在其上，采用4字节对齐。示例的栈帧结构为（假设format = 0x55483dc）：

```
                                +------------+  H
                                |00 00 00 01 |
                                +------------+
                                |00 00 00 02 |
                                +------------+
                                |00 00 00 03 |
                                +------------+
                                |05 54 83 dc |
                                +------------+
                                |return addr |
                                +------------+
                                |last %ebp   |
                           ebp-->+------------+  L
```

于是访问参数就转换成了访问堆栈。但是如何直到参数在栈中的长度呢，答案就在format中，每次遇到%*转换到处理占位符的状态时，根据%后的字符确定变量类型，来移动变量列表的指针。

exercise 21

> exercise21：关于系统调用号，我们在printf的实现里给出了样例，请找到阅读这一行代码
>
> syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
>
> 说一说这些参数都是什么（比如SYS_WRITE在什么时候会用到，又是怎么传递到需要的地方呢？）。

`SYS_WRITE` 为系统调用号，根据 `SYS_WRITE` 确定系统调用的类型； `STD_OUT` 为输出设备，在这里表示VGA显示器。buffer为打印字符串的首地址，count为需要打印处理字符串的长度。后两个参数暂时用不到。

exercise 22

> exercise22：记得前面关于串口输出的地方也有putChar和putStr吗？这里的两个函数和串口那里的两个函数有什么区别？请详细描述它们分别在什么时候能用。

串口输出的 `putChar` 和 `putStr` 属于外部中断，通过串口将字符输出到外部，是硬件级别的调用。而这里的属于系统级别的调用，显存映射将字符打印到屏幕上。

exercise 23

> exercise23：请结合gdt初始化函数，说明为什么链接时用"-Ttext 0x00000000"参数，而不是"-Ttext 0x00200000"。

进入用户程序后，地址空间就从内核态转移到用户态，由用户代码段的全局描述符可知代码段和数据段的起始位置（`base`）为0x200000，因此实际的物理地址为`base + offset`，链接时需要把逻辑地址放到0x00000000的地方，才能保证地址映射正确。

2. Tasks

方便起见，在这里简要说明实现的大体流程以及两个需要注意的点

2.1. kernel和用户程序的加载。

在前面的练习中我们解释了错误的实现方法，在此给出正确的方案。加载器通过解析程序头表得到`elf`文件的可装载段和入口地址，因此我们只需要遍历其中的表项即可。需要注意的是，为了防止有效代码被覆盖，我们不妨将`elf`从硬盘读取到一个较远的地方（本次实验中将`kernel`的`elf`读取到了0xe0000，用户程序加载到了0xf0000），之后通过`memcpy`和`memset`函数将程序加载到0x100000处即可（实地址模式）对加载完成了`elf`所占的空间就可以释放了。需要注意的是，`memcpy`和`memset`涉及到的地址应该是程序头表表项中的物理地址而非逻辑地址。装载用户程序核心代码如下

```
void loadUMain(void) {
    int i = 0;
    int phoff = 0x0;
    uint32_t elf = 0xe00000;
    uint32_t uMainEntry = 0;//read to 0x200000

    for (i = 0; i < 200; i++)
    {
        readSect((void *)(elf + i * 512), 201 + i); //用户程序在磁盘上紧跟kernel之后
    }
    ELFHeader *eh = (ELFHeader *)elf;
    uMainEntry = (uint32_t)eh->entry;
    phoff = eh->phoff;
    // unsigned int phentsize = eh->phentsize;
    ProgramHeader *phEntry = (ProgramHeader *)(elf + phoff);
    ProgramHeader *ph = phEntry;
    // putNum((uint32_t)phEntry);
    for (int idx_ph = 0; idx_ph < eh->phnum; idx_ph++)
    {
        if (ph->type == PT_LOAD)
        {
            putNum(ph->paddr);
            putChar('\n');
            mmemcpy((void *)(ph->paddr+0x200000), (void *)(elf + ph->off), ph->filesz);
            mmemset((void *)(ph->paddr + ph->filesz+0x200000), 0, ph->memsz - ph->filesz);
        }
        ph++;
```

```
    }
    enterUserSpace(uMainEntry);
}
```

在实验时有一个很小的点导致了卡顿。即进入 `enterUserSpace` 函数后不能调用仍和与 `debug` 有关的串口输出函数，会导致 `iret` 时栈结构改变从而使得触发保护错的中断。

2.2. printf

与 `idt` 和 `irq` 有关的内容只需要结合结构体的定义填写即可，注册irq时需要注意只有0x80号中断的 `dpl` 为3

我们重点说明一下 `printf` 的实现，之后会介绍在challenge 3中实现的 `scanf`

在exercise 20中我们已经介绍了对不定形参的查找。在不定形参之前一定有一个参数来确定补丁参数的个数和位置。在linux2.8的标准实现中有如下几个宏和类型定义很重要

```
//形参列表指针
typedef char *m_va_list;
//机器字长
typedef unsigned int acpi_native_uint;
//采用的对齐方式
#define m_AUPBND (sizeof (acpi_native_uint) - 1)
#define m_ADNBND (sizeof (acpi_native_uint) - 1)
//其作用是--倘若sizeof(X)不是4(机器字长为双字32位)的整数倍，去余加4。
#define m_bnd(X, bnd) (((sizeof (X)) + (bnd)) & (~(bnd)))
//讲参数链表指针向高地址跨越T长度（对齐后）找到下一个参数的首地址并取出该参数
#define m_va_arg(ap, T) (*(T *)(((ap) += (m_bnd (T, m_AUPBND))) - (m_bnd
(T,m_ADNBND))))
//其中A为第一个参数format，该宏帮我们讲参数列表指针指向第二个参数
#define m_va_start(ap, A) (m_va_list)((ap) = (((char *) &(A)) + (m_bnd
(A,m_AUPBND))))
//结束搜索
#define m_va_end(ap) ( (ap) = (m_va_list)0 )
```

于是，在初始化后要想获取下一个变量类型为T的参数就很容易了 `T arg = m_va_arg(ap, T);`

格式串的状态有三：0. 普通字符和%，1. 占位符， 2. 无效占位符，状态机主要在0和1之间状态转化。

将参数从上面的宏中读出来后转换成字符串写入 `buffer` ，调用 `syscall` 转到 `SYS_WRITE` 向屏幕发送写入好的 `buffer` 即可。

3. challenges

challenge 1

> challenge1：既然错了(错误的elf加载)，为什么不影响实验代码运行呢?
>
> 这个问题设置为challenge说明它比较难，回答这个问题需要对ELF加载有一定掌握，并且愿意动手去探索。Hint：可以在写完所有内容并保证正确后，改成这段错误代码，进行探索，并回答该问题。（做出来加分，做不出来不扣分）

```
oslab@oslab-VirtualBox:~/oslab/lab2/kernel$ readelf -l kMain.elf

Elf 文件类型为 EXEC (可执行文件)
Entry point 0x1000a8
There are 3 program headers, starting at offset 52

程序头:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x016a0 0x016a0 R E 0x1000
  LOAD           0x003000 0x00103000 0x00103000 0x00120 0x01f00 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10

 Section to Segment mapping:
  段节...
   00     .text .rodata .eh_frame
   01     .got.plt .data .bss
   02
```

我们可以查看 `kMain.elf` 的程序头表，发现其中的两个装入段的 `pysical address` 正是从0x100000开始顺序存放的（4KB对齐），第一个可装入段正是代码段的内容，而在elf头中我们发现程序入口点是0x1000a8，正是代码段中 `kEntry` 的首地址。按照错误代码的装载方法，直接将elf从磁盘上原封不动地复制到物理地址0x100000开始处，程序入口以及代码段和静态数据段在物理内存中的位置是正确的，进而可以正确执行。

```
ELF 头:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF32
  数据:                              2 补码，小端序 (little endian)
  版本:                              1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Intel 80386
  版本:                              0x1
  入口点地址:               0x1000a8
  程序头起点:          52 (bytes into file)
  Start of section headers:          39104 (bytes into file)
  标志:             0x0
  本头的大小:       52 (字节)
  程序头大小:       32 (字节)
  Number of program headers:         3
  节头大小:         40 (字节)
  节头数量:         18
  字符串表索引节头: 17
```

challenge 2

> challenge2：比较关键的几个函数
>
> KeyboardHandle函数是处理键盘中断的函数
> syscallPrint函数是对应于“写”系统调用
> syscallGetChar和syscallGetStr对应于“read”系统调用
>
> 请解释框架代码在干什么。

`KeyboardHandle` 维护了一个循环队列拥作为键盘缓存，同时对于退格符和回车符会操作屏幕光标的位置。对于正常字符，会根据显示大小调整光标位置（displayRow, displayCol）。并将结果放在显存中

`syscallPrint` 将给定地址上的字符串和长度对字符串中的字符遍历处理打印到显存中（和键盘中打印过程类似也设计到繁琐的光标移动操作）

`syscallGetChar` 和 `syscallGetStr` 用于获取键盘缓冲区中的字符或字符串前者通过eax返回一个字符，后者通过edx返回字符串的首地址。

对于 `KeyboardHandle` 和 `syscallPrint` 函数可以重构（耦合度太高，但逻辑没变），本应属于vga显示的部分在键盘控制中不应出现。因此可以再抽象出来一个函数 `void inline_putchar(const char ch)` 函数负责将仍和到来的字符处理到屏幕上（包括特殊字符等），`keyboardHandle` 函数只负责将字符放到缓冲区里，再将该字符传入到抽象出来的函数中即可。

- 并且框架代码对于退格的处理没有删去键盘缓冲区的字符，在本次实验中修改了这个bug

challenge 3

`scanf` 和 `printf` 的实现方法类似，不过需要对两个字符串进行解析，一个是 `format` 另一个是从键盘读取的输入字符串。在我的实现中有这样几个规定：对于 `%d，%x，%c` 可以解析分隔符(`,/\等`)而字符串以空格或回车做分隔符。解析之后的字符串需要转换成对应类型存储到参数列表给出的地址空间中。

字符串转相应类型的函数都是通过头尾两个index扫描键入的字符串实现的，实现过程较为简单。

同时还有 `tokenize` 和 `delSpace` 函数分别将 `right index` 移动到 `splitter` 和 `space` 之上。最后我们为scanf函数简单写了一个测试用例，更复杂的用例将在后续实验中不断完善。

```
printf("Now let's test scanf, input a decimal, a hexadecimal, a string and a character\n");
int decimal;
uint32_t hexadecimal;
char string[20];
char ch;
scanf(" decimal=%d, hexadecimal=%x, string=%s character=%c", &decimal, &hexadecimal, &string, &ch);
printf("your input is %d, %x, %s, %c, am I right?\n", decimal, hexadecimal, string, ch);
```

challenge 4

> 水平有限，暂且不表...

conclusions

conclusion 1

> conclusion1：请回答以下问题
>
> 请**结合代码**，**详细描述**用户程序app调用printf时，从**lib/syscall.c中syscall函数开始执行**到**lib/syscall.c中syscall返回**的全过程，硬件和软件分别做了什么？（实际上就是中断执行和返回的全过程，syscallPrint的执行过程不用描述）

软件将系统调用类型等必要参数存入 `%ecx，%edx` 等寄存器中，执行 `int $0x80` 查找0x80号门描述符，穿越门描述符，此时转为内核态执行（将 `cs : offset` 设置成门描述符中的相应值，在此之前硬件检查了权限是否正确，如若数值上cpl >dpl会再执行 `0xd` 中断）。转到 `syscallHandle` 根据 `SYS_WRITE` 进入 `syscallPrint` 函数执行（涉及了大量VGA有关的操作）一路返回到 `asmDoIrq` 的下一条指令，将 `esp` 弹出，恢复现场，从栈中释放 `intrNum` 和 `Errorcode` ，通过 `iret` 返回到用户态执行 `printf` 的下一条指令

conclusion 2

> conclusion2：请回答下面问题
>
> 请**结合代码**，**详细描述**当产生保护异常错误时，硬件和软件进行配合处理的全过程。

如 `conclusion 1` 中的一段所述，硬件会检查特权等级是否正确，如若产生错误CPU会发送一个保护异常错中断给 `i8259` ，后者将中断请求号进行优先级排队后再将其（错误码）发送给CPU来查询 `irq` 表项，之后的步骤与系统调用的步骤几乎如出一辙(idt-->irqGProtectFault-->asmDoIrq-->irqHandle-->GProtectFaultHandle)，以0xd号中断（GProtectFault）为例，我们的系统会直接 `assert(0)` 杀死自己（在我们的实验中会 `hlt` ）。缺页异常等可处理的会调用中断服务程序（像 `printf` 一样）处理异常。

conclusion3

zjgg们很有耐心，实验理解的很彻底。

附录

```
文件树
            lab2-201220154王紫萁
            |
            +--lab2
            |
            +--report
                |   oslab2.mp4  //效果展示，包括scanf
                |   challenge
                |       |   scanf.c //原函数在syscall.c中，此为展示并不能编译
                |       +   irqHandle.c //challenge2中的重构键盘和VGA的中断处理函数
                |   lab2_report.pdf
                +

```