

实验四实验报告

161220182 周宇航

一、实验目的

本实验通过实现一个简单的生产者消费者程序，介绍基于信号量的进程同步机制

具体要求：

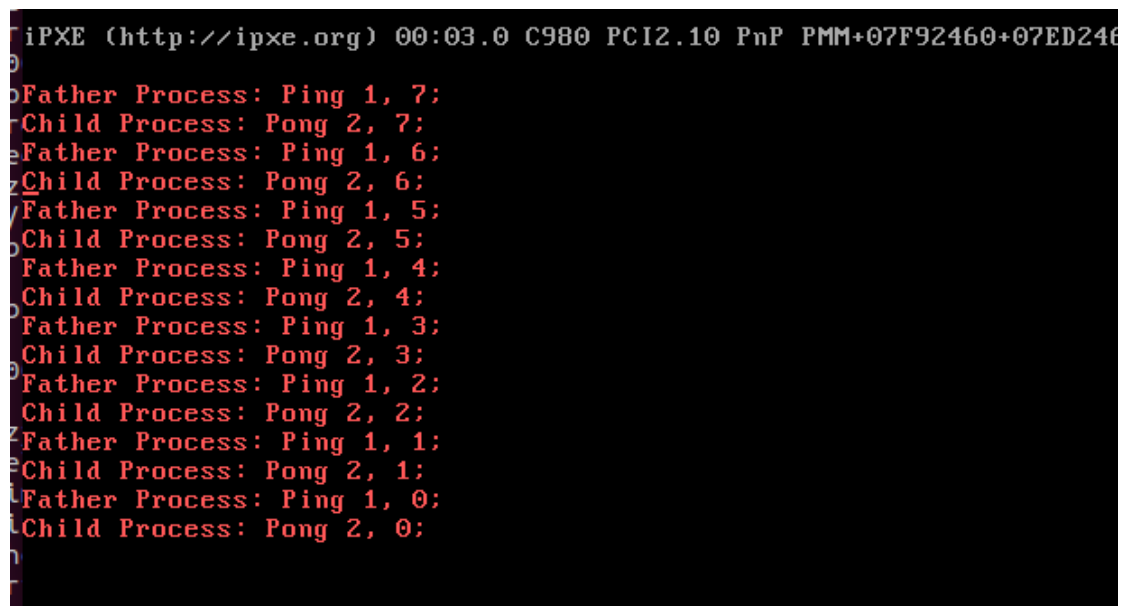
1.1. 实现 SEM_INIT、SEM_POST、SEM_WAIT、SEM_DESTROY 系统调用

1.2. 实现键盘驱动（选做）实现一个键盘驱动，并实现 GETCHAR、GETS、SCANF 等系统调用

结果：两个要求均实现，具体结果见下

二、实验过程

首先在上次实验 lab3 基础上我们完善了上次未能完成的进程切换的功能实现（完成补交），如下图：



```
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460
Father Process: Ping 1, 7;
Child Process: Pong 2, 7;
Father Process: Ping 1, 6;
Child Process: Pong 2, 6;
Father Process: Ping 1, 5;
Child Process: Pong 2, 5;
Father Process: Ping 1, 4;
Child Process: Pong 2, 4;
Father Process: Ping 1, 3;
Child Process: Pong 2, 3;
Father Process: Ping 1, 2;
Child Process: Pong 2, 2;
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
```

并且完成了这个之后我们再开始设计基于信号量的进程同步机制。

基于给出的信号量结构体完成函数实现：

```
struct Semaphore {
    int value;
    struct ProcessTable* proclst;
};
Value 作为信号量值；
proclst 信号量队列指针

typedef struct Semaphore Semaphore;

void P(Semaphore *s) {
    s->value --;
    if (s->value < 0)
        W(s);
}
```

```

void V(Semaphore *s) {
    s->value ++;
    if (s->value <= 0)
        R(s);
}

```

以上机制基本完整，同时我们需要通过完成 W(), R()函数。

具体实现如下：

根据 P、V 操作的相关定义：

P 操作：执行表示把调用函数的进程置成等待信号量 s 的状态，并移入 s 信号量队列，同时释放 CPU，转向进程调度程序

V 操作：执行表示可能释放一个等待信号量 s 的进程，从 s 信号量队列中移除，修改成就绪态并移入就绪队列。，执行 V 操作的进程继续执行。

我做出以下实现：

```

void W(Semaphore *s)
{
    s->proclist++;
    s->proclist = curproc;
    s->proclist->state = BLOCKED;
    schedule();
}

```

将现执行进程放入指针指向的区域，并且将其状态置为阻塞态，然后调用 schedule()函数进行进程调度。此时会执行处于就绪态的其他进程。

```

void R(Semaphore *s)
{
    s->proclist->state = RUNNABLE;
    s->proclist--;
}

```

将指针指向区域中的进程状态置为就绪态，然后将指针减一，即完成队列的出队操作。其余不变，继续执行。

在我的实现中，并没有采用队列的数据结构，而是直接通过指针指向和加减来确定信号量中挂起的进程，这种方法简单易使用。可以很好地完成任务。

信号量机制实现后，我们将测试用例设置好，开始分析其运行规律。

其中

```
sem_t sem;
```

sem_t 类型需要自己先定义，然后通过 ret = sem_init(&sem, value);进行初始化。

那么我们开始实现 SEM_INIT 系统调用

```

int sem_init(sem_t *sem, uint32_t value)
{
    int semret = syscall(5, value, (uint32_t)sem, 0, 0, 0);
    if(semret == 1)

```

```

        return 0;
    else
        return -1;
}

```

我们定义 sem_init 函数系统调用号为 5，然后将 sem 和信号量初始值 value 传入系统调用中，若返回值为 1 则创建成功返回 0，否则返回-1 那么在此 sem 即为信号量变量指针表示信号量变量的地址。用于在内核中创建一个信号量结构体，同时将其地址写进 sem 所对应地址里。

系统调用后，进入内核。处理过后进入 sys_semininit 函数：

```

void sys_semininit(struct TrapFrame *tf)
{
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x1000000;
    uint32_t* temp = (void *)tf->ecx;
    *temp = (uint32_t)sem;
    sem->value = tf->ebx;
    tf->eax = 1;
}

```

同样由于进程地址偏移原因，对于进程 2 要加上偏移，每次涉及地址时，都应转换。然后将传入的地址转换为指针指向系统内核创建的信号量结构体指针，即完成目的操作。然后对信号量进行处理，赋值为 tf->ebx，完成初始化。

随后我们完成 fork 进入父进程，但是父进程 sleep 系统调用后，进程调度转向子进程，开始执行 sem_wait(&sem)即 P 操作。

```

int sem_wait(sem_t *sem)
{
    int semret = syscall(6, 0, (uint32_t)sem, 0, 0, 0);
    if(semret == 1)
        return 0;
    else
        return -1;
}

```

P 操作同样，系统调用号为 6，将 sem 传入如果系统调用返回值为 1，即为成功。

进入内核，调用 sys_semwait：

```

void sys_semwait(struct TrapFrame *tf)
{
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x1000000;
    uint32_t* temp = (void *)tf->ecx;
    *temp = (uint32_t)sem;
    P(sem);
    tf->eax = 1;
}

```

同样的将地址转化为指针指向信号量指针，对信号量进行操作，执行 P 操作，由于 value 初

始为 2，所以第一次调用正常执行。但到 while 循环执行第三次时，P 操作使 value 值小于 0，此时进程被阻塞，系统进程调度，切换到已经从 sleep 中醒来的父进程。

在父进程中继续执行，随后即执行 sem_post(&sem)函数即 V 操作：

```
int sem_post(sem_t *sem)
{
    int semret = syscall(7, 0, (uint32_t)sem, 0, 0, 0);
    if(semret == 1)
        return 0;
    else
        return -1;
}
```

与上面 P 操作基本类似，仅系统调用为 7，随后进入内核执行 sys_semopost

```
void sys_semopost(struct TrapFrame *tf)
{
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    uint32_t* temp = (void *)tf->ecx;
    *temp = (uint32_t)sem;
    V(sem);
    tf->eax = 1;
}
```

依旧与 P 操作类似执行 V 操作，执行后 value 加一为 0，此时将子进程加入就绪态。回到用户进程，随后执行 sleep。

父进程 sleep 后进行进程调度，则将子进程继续执行，随后重复以上操作，子进程 P 操作，父进程 V 操作，交替执行，直至子进程循环结束，实行 sem_destroy(&sem)：

```
int sem_destroy(sem_t *sem)
{
    int semret = syscall(8, 0, (uint32_t)sem, 0, 0, 0);
    if(semret == 1)
        return 0;
    else
        return -1;
}
```

与以上类似，调用号为 8，其余不变，进入内核执行 sys_semdestroy：

```
void sys_semdestroy(struct TrapFrame *tf)
{
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    uint32_t* temp = (void *)tf->ecx;
    *temp = (uint32_t)sem;
    sem->value = 0;
    sem->proclist = NULL;
    tf->eax = 1;
}
```

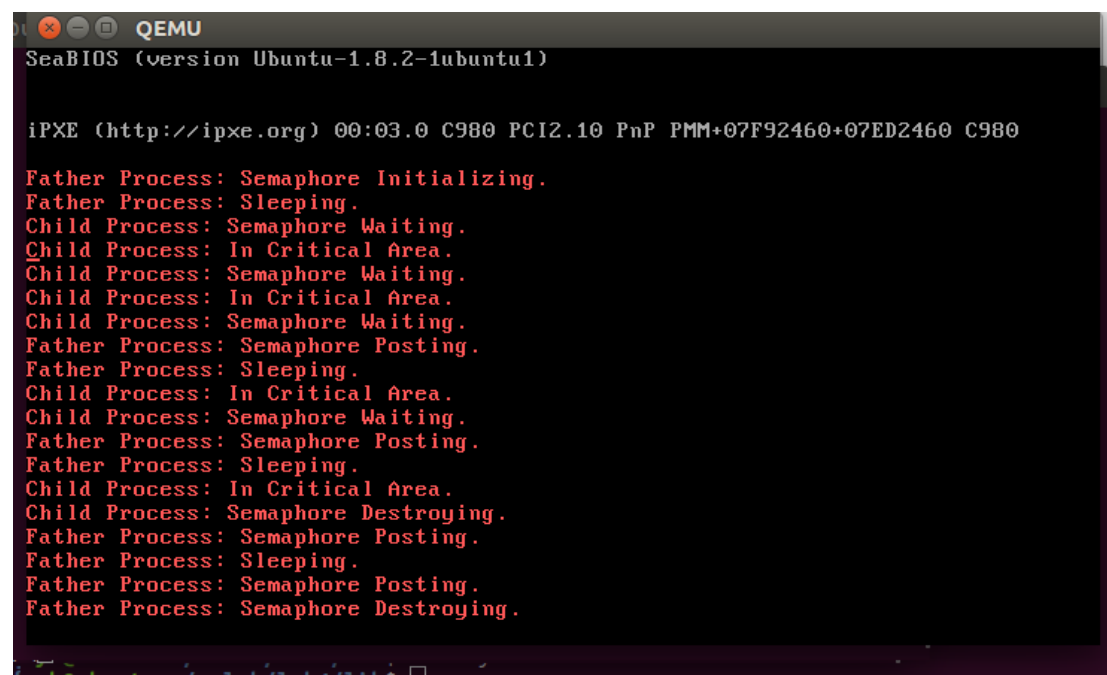
```
}
```

此处我并未释放内存，简便起见，我仅将信号量结构体 value 置 0、队列指针置空，即完成信号量销毁。

完成后，子进程撤销，父进程继续执行，完成 sleep 和 V 操作后，同样销毁信号量、撤销进程，然后程序执行结束。

具体运行结果如下：

三、实验结果



```
QEMU
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

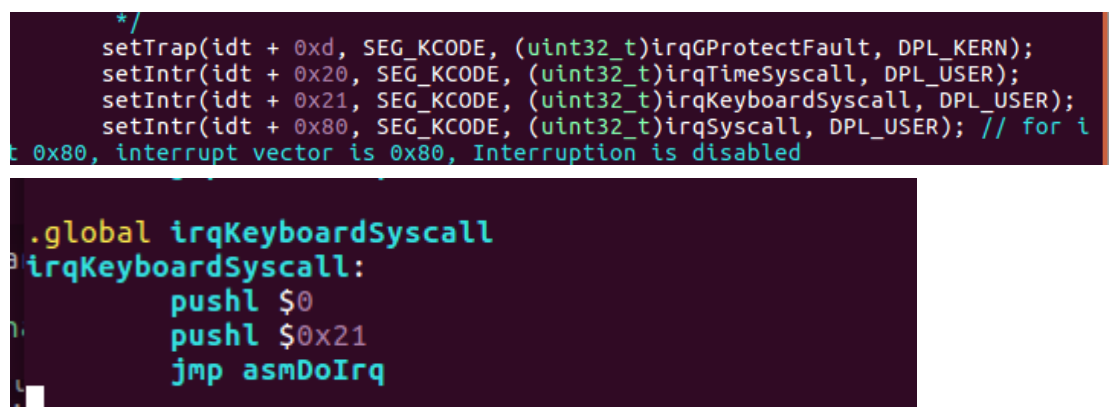
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F92460+07ED2460 C980

Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
```

四、拓展功能（选做）

即实验要求 1.2：实现键盘驱动（选做）实现一个键盘驱动，并实现 GETCHAR、GETS、SCANF 等系统调用

首先，需要实现键盘中断，与时钟中断类似，在 idt 表中增加表项为 0x21，然后再 doIrq.S 中增加 irqKeyboardSyscall 函数



```
*/
setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);
setIntr(idt + 0x20, SEG_KCODE, (uint32_t)irqTimeSyscall, DPL_USER);
setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboardSyscall, DPL_USER);
setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER); // for i
0x80, interrupt vector is 0x80, Interruption is disabled

.global irqKeyboardSyscall
irqKeyboardSyscall:
    pushl $0
    pushl $0x21
    jmp asmDoIrq
```

然后再在系统调用处理部分完成键盘中断处理函数：

而在处理键盘中断时，我们所使用的 getKeyCode()获取的键盘扫描码，必须要经过转化

为 ascii 码后，才能将其读入缓冲区。而在此我定义了相应的扫描码、ascii 码数组来对应的转化过程。其中包括常用的按键的 shift 时 ascii 码和 unshift 时的 ASCII 码，还有功能键，在中断函数中每次采集到扫描码后，就对其进行处理，具体过程不再赘述。最终将键盘输入存入缓冲区数组。

在这里，即介绍缓冲区的设置：

包括 32 位的字符数组，next_w 是将输入字符放入缓冲区数组时的下标、next_r 为读取缓冲区的下标，len 为缓冲区数组字符串长度。

```
struct KEYBUF {
    char key_buf[32];
    int next_r, next_w, len;
};

struct KEYBUF keybuf;
```

```
int breakcode[47] = {0x9e, 0xb0, 0xae, 0xa0, 0x92, 0xa1, 0xa2, 0xa3, 0x97, 0xa4,
0xa5, 0xa6, 0xb2, 0xb1, 0x98, 0x99, 0x90, 0x93, 0x9f, 0x94, 0x96, 0xaf, 0x91, 0
xad, 0x95, 0xac, 0xa9, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8
b, 0x8c, 0x8d, 0x9a, 0x9b, 0xa7, 0xa8, 0xab, 0xb3, 0xb4, 0xb5};
```

```
int usascii[47] = {0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0
x6b, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x7
8, 0x79, 0x7a, 0x60, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x30,
0x2d, 0x3d, 0x5b, 0x5d, 0x3b, 0x27, 0x5c, 0x2c, 0x2e, 0x2f};
```

```
int sasascii[47] = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4a, 0x
4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58
, 0x59, 0x5a, 0x71, 0x21, 0x40, 0x23, 0x24, 0x25, 0x5e, 0x26, 0x2a, 0x38, 0x29,
0x5f, 0x2b, 0x7b, 0x7d, 0x3a, 0x22, 0x7c, 0x3c, 0x3e, 0x3f};
```

```
int funkeys[11][2] = {{0x0e, 0x8e}, {0x0f, 0x8f}, {0x1c, 0x9c}, {0x1d, 0x9d}, {0
x2a, 0xaa}, {0x36, 0xb6}, {0x38, 0xb8}, {0x39, 0xb9}, {0x3a, 0xba}, {0xe01d, 0xe
09d}, {0xe038, 0xe0b8}};
```

```
void keyboardHandle(struct TrapFrame *tf){
    //Log("jian pan\n");
    uint32_t code = getKeyCode();
    int ascii = 0;
    int i;
    if(code == 0x3a || code == 0x2a || code == 0x36 || code == 0xaa || code
== 0xb6)
    {
        if(flag == 0)
            flag = 1;
        else
            flag = 0;
    }
    else
    {
        if(code == 0x0e)
        {
            keybuf.len--;
            keybuf.next_w--;
        }
    }
}
```

443 1 0 0.6%

```

{
    keybuf.len--;
    keybuf.next_w--;
}
else if(code == 0x0f)
{
    for(int m = 0; m < 8; m++)
    {
        keybuf.key_buf[keybuf.next_w] = ' ';
        keybuf.len++;
        keybuf.next_w = (keybuf.next_w + 1) % 32;
    }
}
else if(code == 0x1c)
{
    V(sem);
    keybuf.key_buf[keybuf.next_w] = (char)0x0d;
    Log("%c\n", (char)0x0d);
    keybuf.len = 0;
    keybuf.next_w = 0;
    keybuf.next_r = 0;
}
else if(code == 0x39)

```

459,1-8

98%

```

keybuf.key_buf[keybuf.next_w] = ' ';
keybuf.len++;
keybuf.next_w = (keybuf.next_w + 1) % 32;
}
else
{
    for(i = 0; i < 47; i++)
    {
        if(flag == 0)
        {
            if(code == scancode[i])
            {
                ascii = usascii[i];
                break;
            }
        }
        else
        {
            if(code == scancode[i])
            {
                ascii = sascii[i];
                break;
            }
        }
    }
}

```

474,1-8

9%

```

        if(i != 47)
        {
            if (keybuf.len < 32)
            {
                keybuf.key_buf[keybuf.next_w] = (char)as
                keybuf.len++;
                keybuf.next_w = (keybuf.next_w + 1) % 32
            }
            else
            {
                keybuf.len = 0;
                keybuf.key_buf[keybuf.next_w] = (char)as
                keybuf.len++;
                keybuf.next_w = (keybuf.next_w + 1) % 32
            }
        }
    }
}

```

504.1-8 Bot

完成这些后我们实现了键盘输入和存到缓冲区的操作，然而要实现 getchar 函数我们要做一下工作：

首先定义它：

通过系统调用 3 号，传入字符地址和字符串长度 1，进入内核，返回值为字符 ascii 码

```

int getchar(void)
{
    char c;
    return (syscall(3, 0, (uint32_t)&c, 1, 0, 0) == 1)?(unsigned char)c:EOF;
}

```

进入内核调用 sys_read，同样注意地址转化是否加上偏移量，随后将长度和地址传入 fs_read 函数，进行缓冲区读取。

```

void sys_read(struct TrapFrame *tf) {
    if(curproc->pid == 2)
        tf->ecx = tf->ecx + 0x100000;
    tf->eax = fs_read(tf->ebx, (char*)tf->ecx);
}

```

fs_read 函数：每次从缓冲区读取一个字符，赋给传入的字符地址，next_r 加一，用于下次读取。到此完成 getchar 读入。

```

int fs_read(int fd, char *buf) {
    char code = '\0';
    code = keybuf.key_buf[keybuf.next_r];
    keybuf.next_r++;
    buf[0] = (char)code;
    buf[1] = '\0';
    //Log("%x\n", code);
    //Log("read:%c\n", (char)code);
    return 1;
}

```

随后返回用户进程。

那么在实现了 getchar 函数的基础上，我们实现 gets 函数，通过调用 getchar 实现。事实上，我们 gets 即要在输入缓冲区的字符输入回车之前的字符全部存入字符串数组中，所以我们仅需循环调用 getchar，当输入回车时停止。最后即可得到我们需要的字符串数组。实现如下：

```
char * gets(char *buf)
{
    int i = 0;
    int temp = getchar();
    while(temp != 0xd)
    {
        buf[i] = (char)temp;
        i++;
        temp = getchar();
    }
    buf[i] = '\0';
    //syscall(3, 0, (uint32_t)buf, 0, 0, 0);
    return buf;
}
```

其中 temp == 0xd 时，即为输入了回车。

同样的我们在实现了 getchar 函数和 gets 函数的基础上实现 scanf 函数,同时借鉴 printf 的实现：

采用 printf 相同的可变参数实现：

同时对%c、%d、%s 三种不同实现：

%c 一个字符直接调用 getchar 实现

%d 一个整型数首先采用 gets 函数的方式将缓冲区中取出来作为 char 数组，随后使用小函数 str2int 将其转化为 int 型整数。

%s 一个字符串通过用 gets 函数的方式将缓冲区中取出来作为 char 数组，即可完成。

同时在我的设计中，对于 scanf 多参数输入实现，中间使用输入空格分割，最后输入回车结束。

最后还有一点，在我的设计中，我们通过信号量 P(sem)来阻塞 getchar 等函数的系统调用，直至键盘中断输入回车时 V(sem)，这样就可以实现 getchar 直至缓冲区输入结束才会开始 getchar 系统调用读取缓冲区。

这种方法目前缺陷较大，即每次不同输入操作，都要对信号量赋值改变，但确实对于中断等待不是十分熟悉，有待改进，当前这样实现，考虑后面改进。

```

int scanf(const char *format,...)
{
    va_list ap;
    int retval = 0;
    char *valstr;
    int dec;
    int *valint;
    char buf[100];
    va_start(ap, format);
    //retval = _doscan(stdin, format, ap);
    while (*format)
    {
        if (*format == '%')
        {
            retval++;
            format++;
            switch (*format)
            {
                case 'c':
                {
                    valstr = va_arg(ap, char*);
                    valstr[0] = (char)getchar();
                    getchar();
                }break;
                case 'd':
                {
                    int i = 0;
                    int temp = getchar();
                    while(temp != 32 && temp != 0xd)
                    {
                        buf[i] = (char)temp;
                        i++;

```

```

                        temp = getchar();
                    }
                    buf[i] = '\0';
                    dec = str2int(buf);
                    valint = va_arg(ap, int*);
                    valint[0] = dec;
                }break;
                case 's':
                {
                    valstr = va_arg(ap, char*);
                    int i = 0;
                    int temp = getchar();
                    while(temp != 0xd && temp != 32)
                    {
                        valstr[i] = (char)temp;
                        i++;
                        temp = getchar();
                    }
                    valstr[i] = '\0';
                }break;
                default: break;
            }
        }
        format++;
    }
    va_end(ap);
    return retval;
}

```

选做实验结果如下：

Getchar 演示如下：

```
int uEntry(void)
{
    //char code;
    char buf;
    //char buf[100];
    //char code[100];
    //int num;
    int value = 0;
    sem_t sem;
    sem_init(&sem, value);
    while(1)
    {
        sem_wait(&sem);
        buf = getchar();
        //gets(code);
        //scanf("%c\n", &buf);
        printf("%c\n", buf);
        //num = scanf("%d%s", &buf, &code);
        //printf("%d%s%d\n", buf, code, num);
    }
    return 0;
}
```

```
SeaBIOS (version Ubuntu-1.8.2-1ubu
iPXE (http://ipxe.org) 00:03.0 C98

s
s
aooting from Hard Disk...
w
r
w
q
```

Gets 函数演示实现输入：

```

int uEntry(void)
{
    //char code;
    //char buf;
    //char buf[100];
    char code[100];
    //int num;
    int value = 0;
    sem_t sem;
    sem_init(&sem, value);
    while(1)
    {
        sem_wait(&sem);
        //buf = getchar();
        gets(code);
        //scanf("%c\n", &buf);
        printf("%s\n", code);
        //num = scanf("%d%s", &buf, &code);
        //printf("%d%s%d\n", buf, code, num);
    }
}

```

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10

```

a.jgkg
aquitoqtt
aboting from Hard Disk...
gut
m,
a
MJKahf

```

Scanf 演示%c 输入：

```

int uEntry(void)
{
    //char code;
    char buf;
    //char buf[100];
    //char code[100];
    //int num;
    int value = 0;
    sem_t sem;
    sem_init(&sem, value);
    while(1)
    {
        sem_wait(&sem);
        //buf = getchar();
        //gets(code);
        scanf("%c\n", &buf);
        printf("%c\n", buf);
        //num = scanf("%d%s", &buf, &code);
        //printf("%d%s%d\n", buf, code, num);
    }
    return 0;
}

```

```

SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 P

C
W
Booting from Hard Disk...
S
W
C

```

scanf 实现%d%s 两个变量输入

```

int uEntry(void)
{
    //char code;
    //char buf;
    //char buf[100];
    char code[100];
    int num;
    int value = 0;
    sem_t sem;
    sem_init(&sem, value);
    while(1)
    {
        sem_wait(&sem);
        //buf = getchar();
        //gets(code);
        //scanf("%c\n", &buf);
        //printf("%s\n", code);
        scanf("%d%s", &num, &code);
        printf("%d%s\n", num, code);
    }
}

```

```

n
12jakghgKL
d37957650mihua87fH
S123mi09 from Hard Disk...
t

```

以上测试用例均在 app/main.c 文件中

选做总结：实现了键盘输入，包括大小写、字符 Unshift 和 shift 不同输入区分，部分功能键 tab、空格、回车等。实现 getchar、gets、scanf 基本功能，在以后可能会涉及到功能补充和修改。