Lab4 实验报告

实验进度

本次实验我完成了所有必做内容。

实验步骤以及结果

1. 实现格式化输入函数
   完善syscallReadStdIn函数和 keyboardHandle函数。

```
void keyboardHandle(struct StackFrame *sf) {
	ProcessTable *pt = NULL;
	uint32_t keyCode = getKeyCode();
	if (keyCode == 0) // illegal keyCode
		return;
	//putChar(getChar(keyCode));
	keyBuffer[bufferTail] = keyCode;
	bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;

	if (dev[STD_IN].value < 0) { // with process blocked
		// TODO: deal with blocked situation
		// 唤醒阻塞在 dev[STD_IN] 上的一个进程
		dev[STD_IN].value++;
		pt = (ProcessTable*)((uint32_t)(dev[STD_IN].pcb.prev)-
	(uint32_t)(&((ProcessTable*)0)->blocked));
		pt->state = STATE_RUNNABLE;
		pt->sleepTime = 0;
		dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
		(dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);
	}
	return;
}
```

关于syscallReadStdIn，我们注意到dev数组内容其实相当于信号量。
那么stdin操作应该也类型信号量操作。
讲义中有如下描述：
将current线程加到信号量i的阻塞列表可以通过以下代码实现：

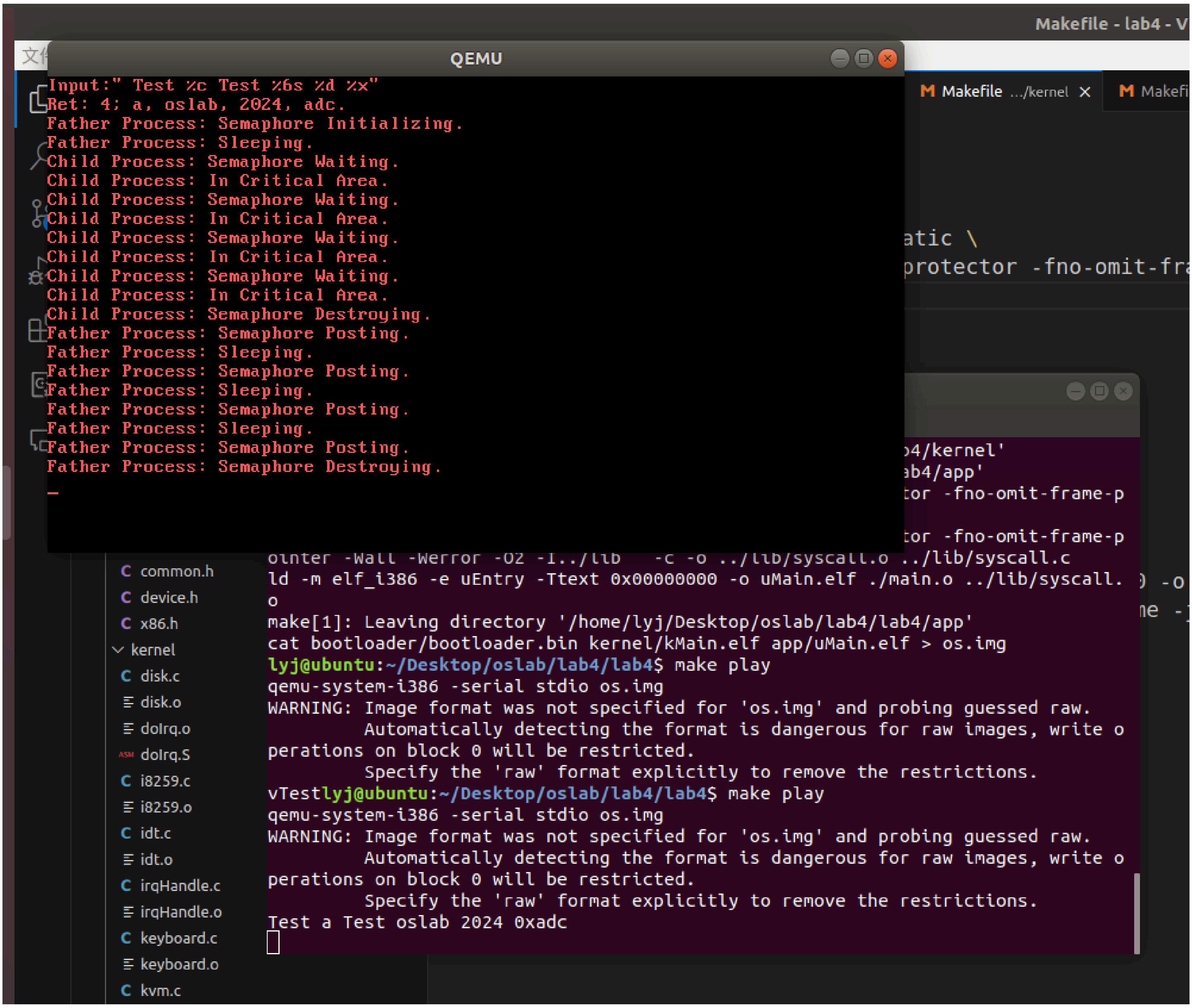
```
pcb[current].blocked.next = sem[i].pcb.next;
pcb[current].blocked.prev = &(sem[i].pcb);
sem[i].pcb.next = &(pcb[current].blocked);
(pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

以下代码可以从信号量i上阻塞的进程列表取出一个进程：

```
pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
(uint32_t)&(((ProcessTable*)0)->blocked));
sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
(sem[i].pcb.prev)->next = &(sem[i].pcb);
```

```
void syscallReadStdIn(struct StackFrame *sf) {
        // TODO: complete `stdin`
        if(dev[STD_IN].value == 0){
                dev[STD_IN].value--;
                //样将current线程加到信号量的阻塞列表可以通过以下代码实现
                pcb[current].blocked.next = dev[STD_IN].pcb.next;
                pcb[current].blocked.prev = &(dev[STD_IN].pcb);
                dev[STD_IN].pcb.next = &(pcb[current].blocked);
                (pcb[current].blocked.next)->prev = &(pcb[current].blocked);

                //阻塞
                pcb[current].state = STATE_BLOCKED;
                pcb[current].sleepTime = -1;
                asm volatile("int $0x20");
                //字符传入
                int sel = sf->ds;
                char *str = (char *)sf->edx;
                int i = 0;
                char character = 0;
                int len = sf->ebx;
                asm volatile("movw %0, %%es"::"m"(sel));
                while(i < len - 1){
                        if(bufferHead != bufferTail){
                                character = getChar(keyBuffer[bufferHead]);
                                bufferHead = (bufferHead + 1)%MAX_KEYBUFFER_SIZE;
                                putChar(character);
                                if(character != 0){
                                        asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(
                                        i++;
                                }
                        }
                        else break;//缓冲区溢出
                }
                asm volatile("movb $0x0, %%es:(%0)"::"r"(str+i));
                pcb[current].regs.eax = i;//scanf 返回实际读取的字节数
                return;
        }
        else if(dev[STD_IN].value < 0){
                pcb[current].regs.eax = -1;//后来的进程会返回 -1
        }

}
```

测试结果:



2. 实现信号量

关于信号量与当前进程的操作，前面已经提过了。

首先是信号量的初始化操作。

```c
void syscallSemInit(struct StackFrame *sf) {
	// TODO: complete `SemInit`
	int i = 0;
	for(; i < MAX_SEM_NUM; ++i){
		if(sem[i].state == 0) break;//空闲的信号量
	}
	if(i >= MAX_SEM_NUM){//初始化失败
		pcb[current].regs.eax = -1;
		return;
	}
	sem[i].state = 1;
	sem[i].value = (int)sf->edx;
	sem[i].pcb.next = &(sem[i].pcb);
	sem[i].pcb.prev = &(sem[i].pcb);
	pcb[current].regs.eax = i;
	return;
}
```

信号量P操作

```
void syscallSemWait(struct StackFrame *sf) {//P操作
	// TODO: complete `SemWait` and note that you need to consider some special situations
	int id = sf->edx;
	if(id < 0 || id >= MAX_SEM_NUM){
		pcb[current].regs.eax = -1;//操作失败
		return;
	}
	if(sem[id].state != 1){
		pcb[current].regs.eax = -1;//操作失败
		return;
	}
	sem[id].value--;
	pcb[current].regs.eax = 0;//操作成功返回0
	if(sem[id].value < 0){//阻塞自身
		//样将current线程加到信号量id的阻塞列表可以通过以下代码实现
		pcb[current].blocked.next = sem[id].pcb.next;
		pcb[current].blocked.prev = &(sem[id].pcb);
		sem[id].pcb.next = &(pcb[current].blocked);
		(pcb[current].blocked.next)->prev = &(pcb[current].blocked);
		//阻塞
		pcb[current].state = STATE_BLOCKED;
		pcb[current].sleepTime = -1;
		asm volatile("int $0x20");
	}
}
```

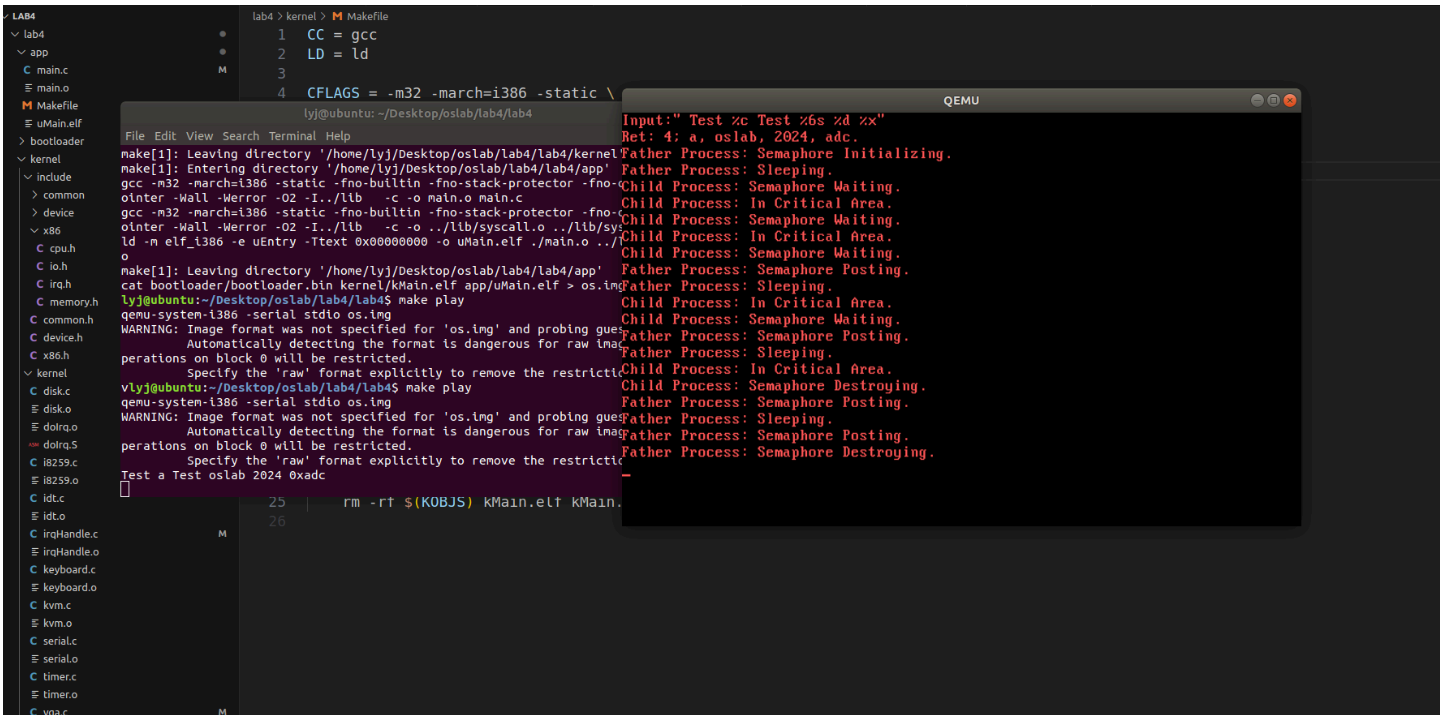
信号量V操作

```
void syscallSemPost(struct StackFrame *sf) {//V操作
        int i = (int)sf->edx;
        ProcessTable *pt = NULL;
        if (i < 0 || i >= MAX_SEM_NUM) {
                pcb[current].regs.eax = -1;//操作失败
                return;
        }
        // TODO: complete other situations
        if(sem[i].state != 1){
                pcb[current].regs.eax = -1;//操作失败
                return;
        }
        sem[i].value++;
        pcb[current].regs.eax = 0;//操作成功返回0
        if(sem[i].value <= 0){
                pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
                        (uint32_t)&(((ProcessTable*)0)->blocked));
                sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
                (sem[i].pcb.prev)->next = &(sem[i].pcb);
                pt->state = STATE_RUNNABLE;
                pt->sleepTime = 0;
        }
}
```

销毁信号量

```
void syscallSemDestroy(struct StackFrame *sf) {
        // TODO: complete `SemDestroy`
        int i = sf->edx;
        if(sem[i].state == 1){
                sem[i].state = 0;
                pcb[current].regs.eax = 0;
                asm volatile("int $0x20");
        }
        else {
                pcb[current].regs.eax = -1;
        }
        return;
}
```

下面是效果展示:



首先我们可以发现信号量sem初值为2.故子进程前两次在P操作后里面就可以立马进入互斥访问区。后面则是需要父进程对sem执行V操作，方可进入互斥访问区。最终是4次P，4次V。

3. 解决进程同步问题

这里解决生产者消费者问题

代码:

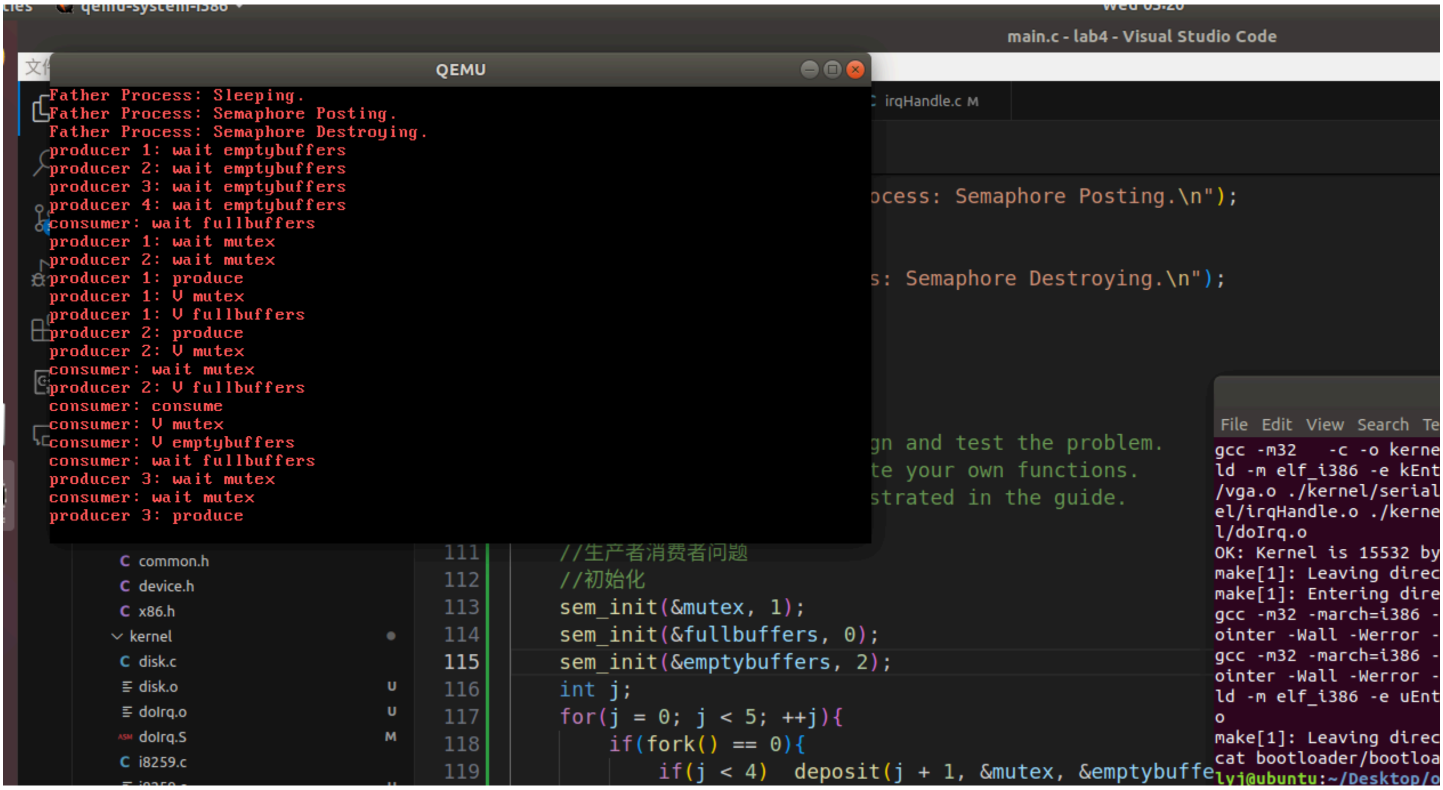
```
#include "lib.h"
#include "types.h"

void deposit(int id, sem_t* mutex, sem_t* emptybuffers, sem_t* fullbuffers)
{
	sleep(128);
	printf("producer %d: wait emptybuffers\n", id);
	sem_wait(emptybuffers);
	sleep(128);
	printf("producer %d: wait mutex\n", id);
	sem_wait(mutex);
	sleep(128);
	printf("producer %d: produce\n", id);
	sleep(128);
	printf("producer %d: V mutex\n", id);
	sem_post(mutex);
	sleep(128);
	printf("producer %d: V fullbuffers\n", id);
	sem_post(fullbuffers);
}

void remove(sem_t* mutex, sem_t* emptybuffers, sem_t* fullbuffers)
{
	int k;
	for(k = 1; k <= 4; ++k){
		sleep(128);
		printf("consumer: wait fullbuffers\n");
		sem_wait(fullbuffers);
		sleep(128);
		printf("consumer: wait mutex\n");
		sem_wait(mutex);
		sleep(128);
		printf("consumer: consume\n");
		sleep(128);
		printf("consumer: V mutex\n");
		sem_post(mutex);
		sleep(128);
		printf("consumer: V emptybuffers\n");
		sem_post(emptybuffers);
	}
}

int uEntry(void)
{
	// For lab4.1
	// Test 'scanf'
```

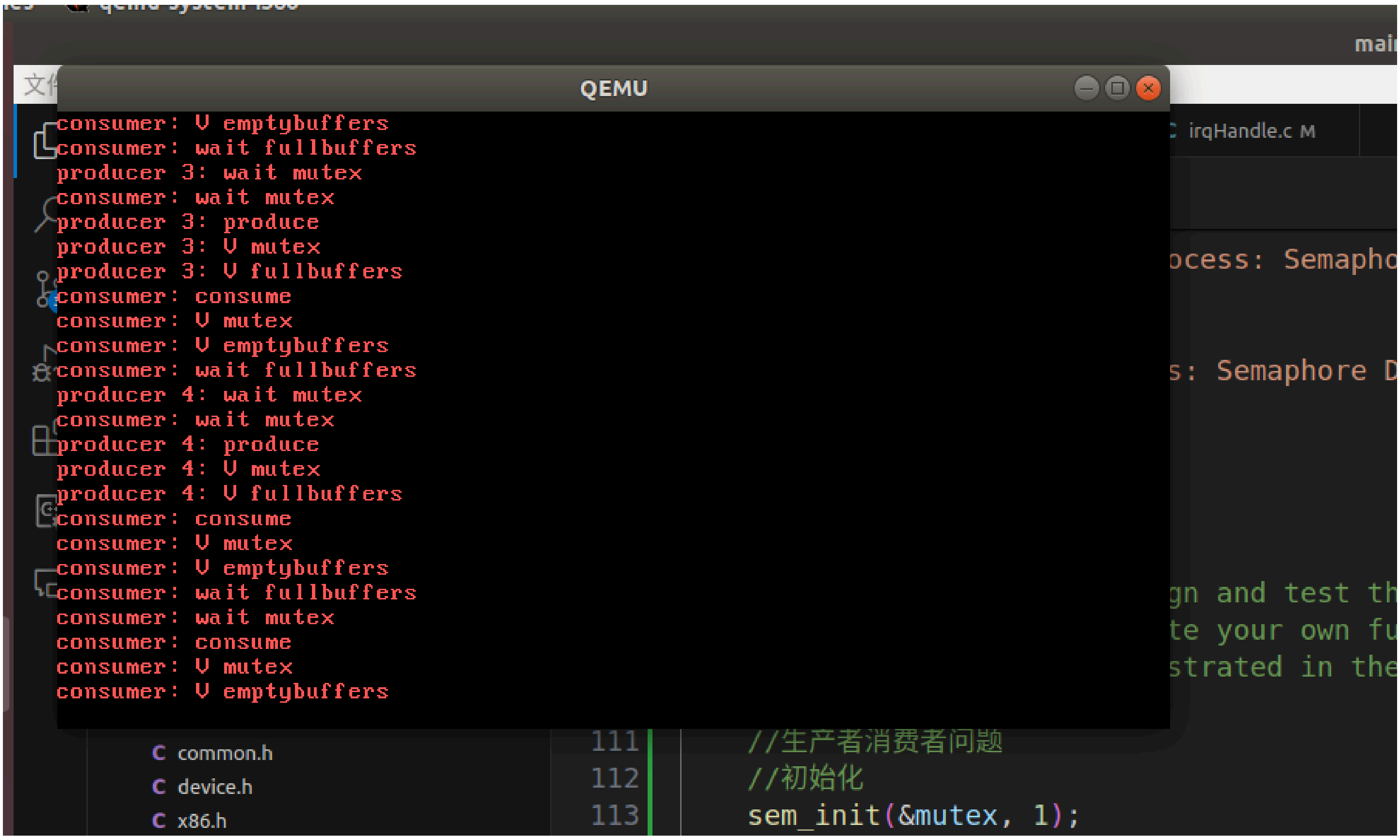
```
int dec = 0;
int hex = 0;
char str[6];
char cha = 0;
int ret = 0;
while (1)
{
        printf("Input:\" Test %%c Test %%6s %%d %%x\"\n");
        ret = scanf(" Test %c Test %6s %d %x", &cha, str, &dec, &hex);
        printf("Ret: %d; %c, %s, %d, %x.\n", ret, cha, str, dec, hex);
        if (ret == 4)
                break;
}

// For lab4.2
// Test 'Semaphore'
int i = 4;

sem_t sem;
sem_t mutex;
sem_t fullbuffers;
sem_t emptybuffers;
printf("Father Process: Semaphore Initializing.\n");
ret = sem_init(&sem, 2);
if (ret == -1)
{
        printf("Father Process: Semaphore Initializing Failed.\n");
        exit();
}

ret = fork();
if (ret == 0)
{
        while (i != 0)
        {
                i--;
                printf("Child Process: Semaphore Waiting.\n");
                sem_wait(&sem);
                printf("Child Process: In Critical Area.\n");
        }
        printf("Child Process: Semaphore Destroying.\n");
        sem_destroy(&sem);
        exit();
}
else if (ret != -1)
{
```

```
            while (i != 0)
            {
                    i--;
                    printf("Father Process: Sleeping.\n");
                    sleep(128);
                    printf("Father Process: Semaphore Posting.\n");
                    sem_post(&sem);
            }
            printf("Father Process: Semaphore Destroying.\n");
            sem_destroy(&sem);
            //exit();
    }

    // For lab4.3
    // TODO: You need to design and test the problem.
    // Note that you can create your own functions.
    // Requirements are demonstrated in the guide.

    //生产者消费者问题
    //初始化
    sem_init(&mutex, 1);
    sem_init(&fullbuffers, 0);
    sem_init(&emptybuffers, 2);
    int j;
    for(j = 0; j < 5; ++j){
            if(fork() == 0){
                    if(j < 4)  deposit(j + 1, &mutex, &emptybuffers, &fullbuffers);
                    else remove(&mutex, &emptybuffers, &fullbuffers);
                    break;
            }
    }
    //sem_destroy(&mutex);
    //sem_destroy(&fullbuffers);
    //sem_destroy(&emptybuffers);
    while(1);
    return 0;
}
```

总共4个生产者，1个消费者。这里我采用了mutex,fullbuffers,emptybuffers3个信号量。mutex用于临界区资源互斥访问。fullbuffers用于指示缓冲区是否满，若满，则消费者进行消费。emptybuffers用于指示缓冲区是否有空，若有则生产者进行生产。

下面是效果展示：





我初始设置emptybuffers=2，即缓冲区大小为2。故可以发现在前2个生产者依次进行生产操作后，后面的生产者P操作后会阻塞，需要等待消费者消费，V(emptybuffers)后方可继续进行后续操作。

自由报告

此次实验让我对信号量P，V操作有了更加深入的了解。