OS-lab4-report

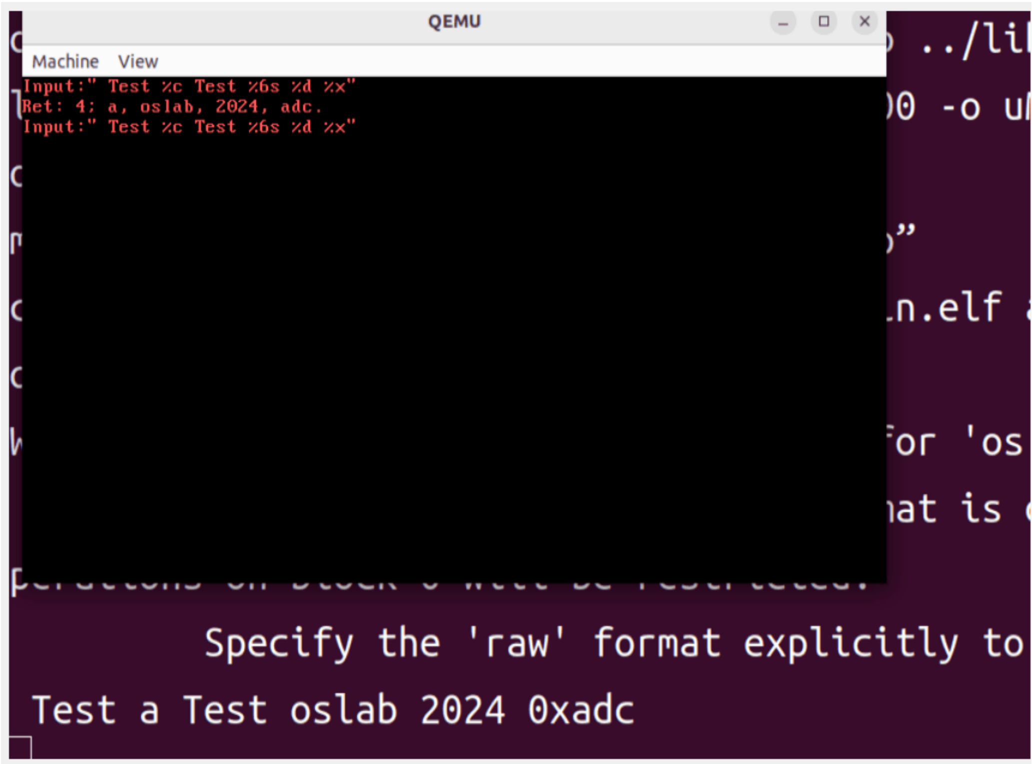
姓名	时昌军
学号	221220085
邮箱	221220085@smail.nju.edu.cn

一、实验进度

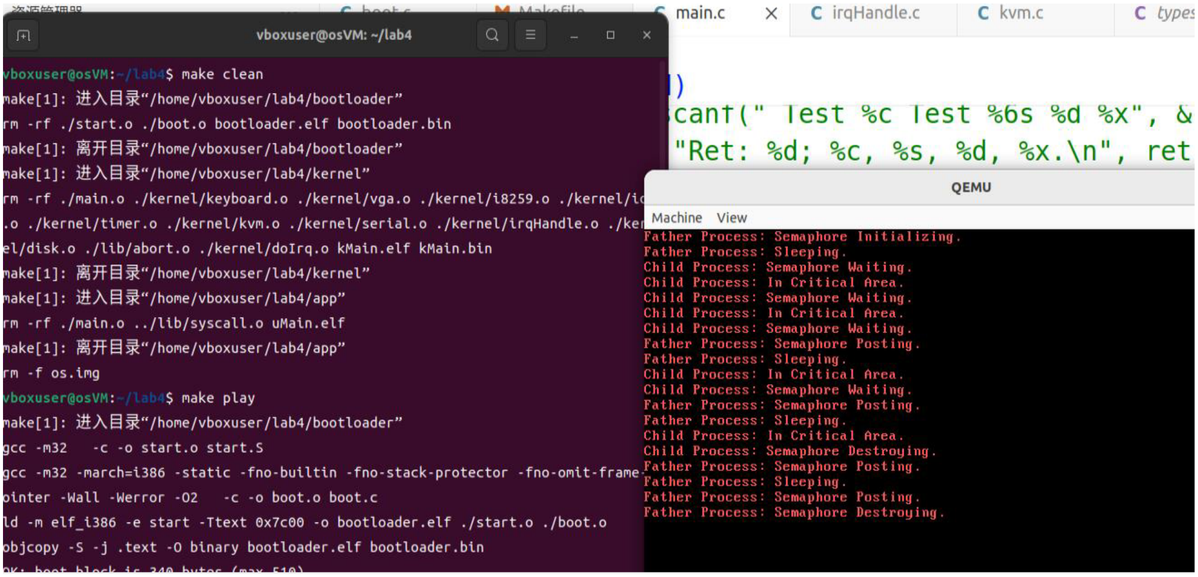
实验要求全部完成，选做部分完成了哲学家进餐问题。

二、实验结果

1.1

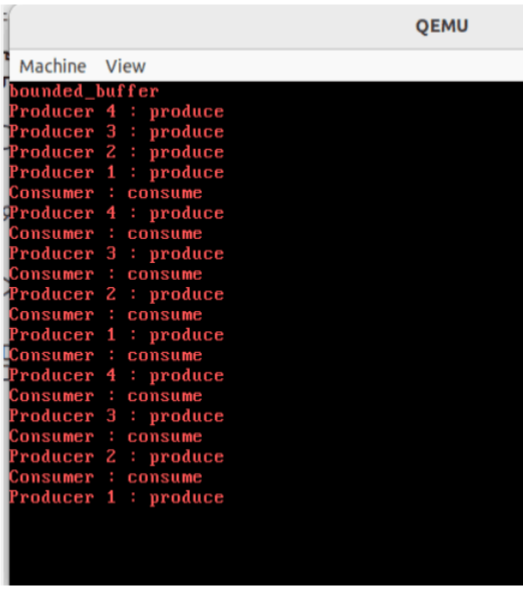


1.2

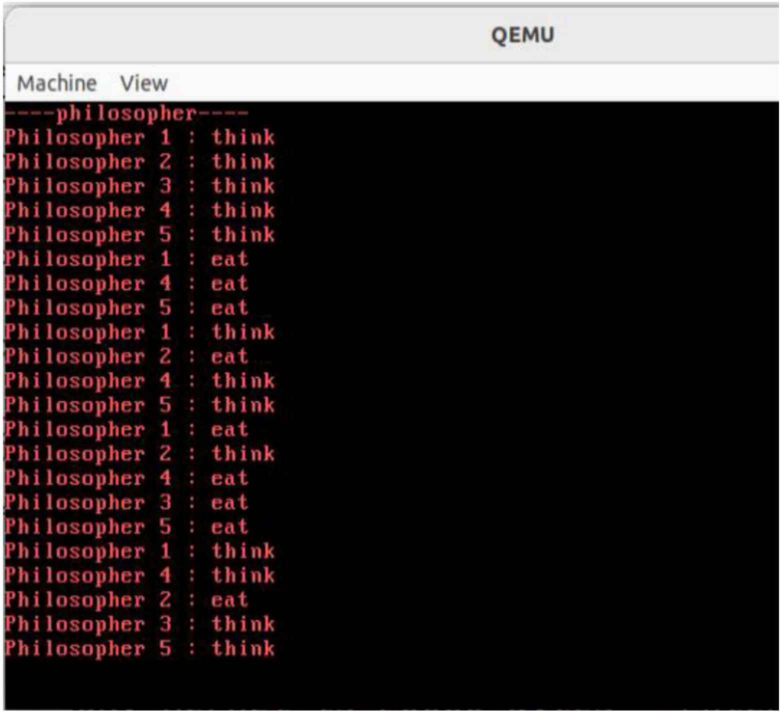


1.3

生产者消费者：



哲学家进餐：



三、实验修改的代码

1.1. 实现格式化输入函数

`keyboardHandle` 要做的事情就两件:

1. 将读取到的 `keyCode` 放入到 `keyBuffer` 中

```
uint32_t keyCode = getKeyCode();
if (keyCode == 0) // illegal keyCode
    return;
keyBuffer[bufferTail] = keyCode;
bufferTail=(bufferTail+1)%MAX_KEYBUFFER_SIZE;
```

2. 唤醒阻塞在 `dev[STD_IN]` 上的一个进程

```
if (dev[STD_IN].value < 0) { // with process blocked
    // TODO: deal with blocked situation
    uint32_t prev=(uint32_t)(dev[STD_IN].pcb.prev);
    uint32_t blocked=(uint32_t)&(((ProcessTable*)0)->blocked);
    pt = (ProcessTable*)(prev-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);
    dev[STD_IN].value = 0;
}
```

接下来安排 `syscallReadStdIn` ，它要做的事情也就两件:

1. 如果 `dev[STD_IN].value == 0`，将当前进程阻塞在 `dev[STD_IN]` 上

```
if(dev[STD_IN].value == 0){
    pcb[current].blocked.next = dev[STD_IN].pcb.next;
```

```
    pcb[current].blocked.prev = &(dev[STD_IN].pcb);
    (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
    pcb[current].state = STATE_BLOCKED;
    pcb[current].sleepTime = -1;

    dev[STD_IN].pcb.next = &(pcb[current].blocked);
    dev[STD_IN].value = -1;
}
else if(dev[STD_IN].value < 0){
    sf->eax = -1;
    return;
}
```

成功阻塞后中断 `asm volatile("int $0x20")`;，切换进程同时监听键盘输入。

2. 进程被唤醒，读 `keyBuffer` 中的所有数据 (参考实验手册上的实现)

```
int sfds = sf->ds;
char *sfedx = (char*)sf->edx;
char character = 0;
int cnt = 0;
int size = (bufferTail - bufferHead + MAX_KEYBUFFER_SIZE) %
MAX_KEYBUFFER_SIZE;
asm volatile("movw %0, %%es"::"m"(sfds));
for(int i=0;i<size;++i){
    character = getChar(keyBuffer[bufferHead+i]);
    if(character>0){
        putChar(character);
        asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(sfedx+cnt));
        cnt+=1;
    }
}
character = 0;
asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(sfedx+cnt));
bufferTail = bufferHead;
sf->eax = cnt;
```

1.2 实现信号量相关系统调用

参考手册中的以下代码：

这样将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);
```

1. `sem_init`

`sem_init` 系统调用用于初始化信号量，其中参数 0 ，指针`sem`指向初始化成功的信号量，否则返回-1

```
int i;
for (i = 0; i < MAX_SEM_NUM ; i++) {
    if (sem[i].state == 0) // do not use
        break;
}
if (i != MAX_SEM_NUM) {
    sem[i].state = 1;
    sem[i].value = (int32_t)sf->edx;
    sem[i].pcb.next = &(sem[i].pcb); // 用自己的位置作为指针，本质上是一个无效的位置
    sem[i].pcb.prev = &(sem[i].pcb);
    pcb[current].regs.eax = i;
}
else
    pcb[current].regs.eax = -1;
```

2. `sem_post`

`sem_post` 系统调用对应信号量的V操作，其使得 `sem` 指向的信号量的 `value` 增一，若 `value` 取值不大于0，则释放一个阻塞在该信号量上进程（即将该进程设置为就绪态），若操作成功则返回0，否则返回-1

```
int i = (int)sf->edx;
//ProcessTable *pt = NULL;
if (i < 0 || i >= MAX_SEM_NUM) {
    pcb[current].regs.eax = -1;
    return;
}
// TODO: complete other situations
else if (sem[i].state == 1) {
    pcb[current].regs.eax = 0;
    sem[i].value++;
    if (sem[i].value <= 0) {
        //以从信号量i上阻塞的进程列表取出一个进程
        uint32_t prev=(uint32_t)(sem[i].pcb.prev) ;
        ProcessTable *pt = (ProcessTable*)(prev - (uint32_t)&(((ProcessTable*)0)->blocked));
        pt->state = STATE_RUNNABLE;
        pt->sleepTime = 0;
        sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
        (sem[i].pcb.prev)->next = &(sem[i].pcb);
    }
}
else
    pcb[current].regs.eax = -1;
```

3. `sem_wait`

`sem_wait` 系统调用对应信号量的P操作，其使得`sem`指向的信号量的 `value` 减一，若 `value` 取值小于0，则阻塞自身，否则进程继续执行，若操作成功则返回0，否则返回-1。

```
int i = (int)sf->edx;
//ProcessTable *pt = NULL;
if (i < 0 || i >= MAX_SEM_NUM) {
    pcb[current].regs.eax = -1;
    return;
}
else if (sem[i].state == 1) {
    pcb[current].regs.eax = 0;
    sem[i].value--;
    if (sem[i].value < 0) {
        //将current线程加到信号量i的阻塞列表
        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);

        pcb[current].state = STATE_BLOCKED;
        pcb[current].sleepTime = -1;
        asm volatile("int $0x20");
    }
}
else
    pcb[current].regs.eax = -1;
```

4. `sem_destroy`

`sem_destroy` 系统调用用于销毁 `sem` 指向的信号量，销毁成功返回0，否则返回-1，若尚有进程阻塞在该信号量上，可带来未知错误。

```
int i = sf->edx;
if (sem[i].state == 1)
{
    pcb[current].regs.eax = 0;
    sem[i].state = 0;
    asm volatile("int $0x20");
}
else
    pcb[current].regs.eax = -1;
```

1.3. 解决进程同步问题

实现 `getpid` 系统调用：

```c
void syscallGetPid(struct StackFrame *sf) {
    pcb[current].regs.eax = current;
    return;
}
```

1.3.1. 生产者-消费者问题

4个生产者，1个消费者同时运行

生产者生产， `printf("Producer %d: produce\n", id);`

消费者消费， `printf("Consumer : consume\n");`

任意P、V及生产、消费动作之间添加 `sleep(128)`;

生产者---->缓冲区---->消费者

多个生产者在生产数据后放在一个缓冲区里，单个消费者从缓冲区取出数据处理，任何时刻只能有一个生产者或消费者可访问缓冲区。 任何时刻只能有一个线程操作缓冲区（互斥访问）； 缓冲区空时，消费者必须等待生产者（条件同步）； 缓冲区满时，生产者必须等待消费者（条件同步）

用信号量描述每个约束： 二进制信号量 `mutex` ，资源信号量 `fullBuffers` ，资源信号量 `emptyBuffers`

手册中提供了伪代码：

> 伪代码描述一下：
>
> ```
> class BoundedBuffer {
>     mutex = new Semaphore(1);
>     fullBuffers = new Semaphore(0);
>     emptyBuffers = new Semaphore(n);
> }
> ```
>
> ```
> BoundedBuffer::Deposit(c){
>   emptyBuffers->P();
>   mutex->P();
>   Add c to the buffer;
>   mutex->V();
>   fullBuffers->V();
> }
> ```
>
> ```
> BoundedBuffer::Remove(c){
>   fullBuffers->P();
>   mutex->P();
>   Remove c from buffer;
>   mutex->V();
>   emptyBuffers->V();
> }
> ```

代码如下：

```c
void deposit(sem_t* mutex, sem_t* fullBuffers, sem_t* emptyBuffers){
    int i = 2;
    while (i > 0){
        sem_wait(emptyBuffers);
        sleep(128);
        sem_wait(mutex);
        sleep(128);
        int id=getpid()-1;
        printf("Producer %d : produce\n",id);
        sleep(128);
        sem_post(mutex);
        sleep(128);
        sem_post(fullBuffers);
```

```
            sleep(128);
            i--;
        }
}
void remove(sem_t* mutex, sem_t* fullBuffers, sem_t* emptyBuffers){
        int i = 8;
        while (i > 0){
            sem_wait(fullBuffers);
            sleep(128);
            sem_wait(mutex);
            sleep(128);
            printf("Consumer : consume\n");
            sleep(128);
            sem_post(mutex);
            sleep(128);
            sem_post(emptyBuffers);
            sleep(128);
            i--;
        }
}
int boundedBuffer(void){
        int n = 4;      // buffer size
        int producer = 4;
        int consumer = 1;
        sem_t mutex,fullBuffers,emptyBuffers;
        sem_init(&mutex,1);
        sem_init(&fullBuffers,0);
        sem_init(&emptyBuffers,n);
        int ret;
        while(producer >0){
            ret = fork();
            if(ret == 0){
                deposit(&mutex,&fullBuffers,&emptyBuffers);
                exit();
            }
            producer-=1;
        }
        while(consumer >0){
            ret = fork();
            if(ret == 0){
                remove(&mutex,&fullBuffers,&emptyBuffers);
                exit();
            }
            consumer-=1;
        }
        exit();
        return 0;
}
```

1.3.2. 哲学家就餐问题

5个哲学家同时运行

哲学家思考, `printf("Philosopher %d: think\n", id);`

哲学家就餐, `printf("Philosopher %d: eat\n", id);`

在P、V及思考、就餐动作之间添加 `sleep(128)`;

参照手册上的方案3:

方案3:

```
#define N 5                 // 哲学家个数
semaphore fork[5];          // 信号量初值为1
void philosopher(int i){    // 哲学家编号：0-4
  while(TRUE){
    think();                // 哲学家在思考
    if(i%2==0){
      P(fork[i]);           // 去拿左边的叉子
      P(fork[(i+1)%N]);     // 去拿右边的叉子
    } else {
      P(fork[(i+1)%N]);     // 去拿右边的叉子
      P(fork[i]);           // 去拿左边的叉子
    }
    eat();                  // 吃面条
    V(fork[i]);             // 放下左边的叉子
    V(fork[(i+1)%N]);       // 放下右边的叉子
  }
}
```

没有死锁，可以实现多人同时就餐

代码如下:

```
sem_t forks[5];
for (int i = 0; i < 5; i++)
    sem_init(&forks[i], 1);
for(int i=0,ret=0;i<5;++i){
    ret = fork();
    if(ret == 0){
        int id = getpid()-1;
        while(1){
            printf("Philosopher %d : think\n",id);
            sleep(128);
            if(i%2 == 0){
                sem_wait(&forks[i]);
                sleep(128);
                sem_wait(&forks[(i+1)%5]);
                sleep(128);
            }
            else{
                sem_wait(&forks[(i+1)%5]);
                sleep(128);
```

```
                    sem_wait(&forks[i]);
                    sleep(128);
                }
                printf("Philosopher %d : eat\n",id);
                sleep(128);
                sem_post(&forks[i]);
                sleep(128);
                sem_post(&forks[(i+1)%5]);
                sleep(128);
            }
            exit();
        }
}
```

四、实验心得

本次实验让我进一步理解了进程之间的同步机制，对信号量的实现有了更深的理解。