OS-lab4

211275009 陈铭浩
email：211275009@smail.nju.edu.cn
实验进度：我完成了所有内容

个人感悟

- 本次实验设计到semaphore的数据结构，虽然没有让自己实现，但是我注意到助教给出的结构十分精妙，采用了双向链表来保存阻塞在某一信号量上的进程队列，从头部插入，从尾部取出，既方便使用，还保证了基本上先进先出的性质。

1. 实现格式化输入函数(irqHandle.c)

- 完成keyboardHandle()函数：将按键的扫描码存储在键盘缓冲区中，并处理阻塞在标准输入上的进程，使其变为可运行状态。

```c
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
        dev[STD_IN].value++;
        pt = (ProcessTable *)((uint32_t)dev[STD_IN].pcb.prev - (uint32_t)&(((ProcessTable *)0)->blocked));
        dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
        (dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);

        pt->state = STATE_RUNNABLE;
        pt->sleepTime = 0;
    }

    return;
}
```

- 完成syscallReadStdIn()函数：首先判断STD_IN对应的信号量的进程队列中是否有进程阻塞（value？0），如果没有，则将当前进程阻塞在dev[STD_IN]上（这里sleepTime设为-1表示无限期阻塞），然后唤醒进程，将键盘缓冲区的内容读入到用户缓冲区，读取数据；如果已有进程阻塞在标准输入上，则返回-1

```c
void syscallReadStdIn(struct StackFrame *sf) {
    // TODO: complete `stdin`
    if(dev[STD_IN].value == 0){
        // 将当前进程阻塞在dev[STD_IN]上
        dev[STD_IN].value--;

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

        // 唤醒进程，将键盘缓冲区的内容读入到用户缓冲区
        int sel = sf->ds;
        char *str = (char *)sf->edx;
        int size = sf->ebx;
        int i = 0;
        char character = 0;
        asm volatile("movw %0, %%es"::"m"(sel));
        for(i = 0; i < size-1; i++){
            if(bufferHead == bufferTail){
                break;
            }
            character = getChar(keyBuffer[bufferHead]);
            bufferHead = (bufferHead + 1) % MAX_KEYBUFFER_SIZE;
            putChar(character);
            if(character != 0){
                asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str + i));
            }
```

2. 实现信号量（irqHandle.c）

- 完成syscallSemInit()函数：初始化信号量，首先寻找是否有可用的空闲信号量，若没有，直接返回-1，若有，将对应信号量state设为1占用，value设为传入值，返回相应信号量的索引

```c
void syscallSemInit(struct StackFrame *sf) {
    // TODO: complete `SemInit`
    int i;
    for (i = 0; i < MAX_SEM_NUM; i++) {
        if (sem[i].state == 0)
            break;
    }
    if(i != MAX_SEM_NUM){
        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;
    }
    else{
        pcb[current].regs.eax = -1;
    }
    return;
}
```

- 完成syscallSemWait()函数：P操作，首先考虑传入的信号量索引值是否合法（0~MAX_SEM_NUM），然后再检查对应信号量的state是否为1，为1说明对应信号量存在，value自减，然后检查当前value是否小于0，如果小于，则将当前进程置为阻塞态，加入对应信号量的进程队列，最后进行进程切换

```c
void syscallSemInit(struct StackFrame *sf) {
    // TODO: complete `SemInit`
    int i;
    for (i = 0; i < MAX_SEM_NUM; i++) {
        if (sem[i].state == 0)
            break;
    }
    if(i != MAX_SEM_NUM){
        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;
    }
    else{
        pcb[current].regs.eax = -1;
    }
    return;
}
```

- 完成syscallSemPost()函数：V操作，首先考虑传入的信号量索引值是否合法（0~MAX_SEM_NUM），然后再检查对应信号量的state是否为1，为1说明对应信号量存在，value自增，然后检查当前value是否小于等于0，如果是，唤醒信号量进程队列中的一个进程，将其置为就绪态

```
void syscallSemPost(struct StackFrame *sf) {
    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
    if(sem[i].state == 0){
        pcb[current].regs.eax = -1;
    }
    else{
        // sem[i].state == 1
        sem[i].value++;
        pcb[current].regs.eax = 0;
        if(sem[i].value <= 0){
            pt = (ProcessTable*)((uint32_t)(sem[i].pcb.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);
        }
    }
}
```

- 完成syscallSemDestroy()函数：首先检查信号量是否存在（state==1），若存在才可以销毁信号量，将state置为0，并显式进程切换

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

3. 解决进程同步问题（main.c）

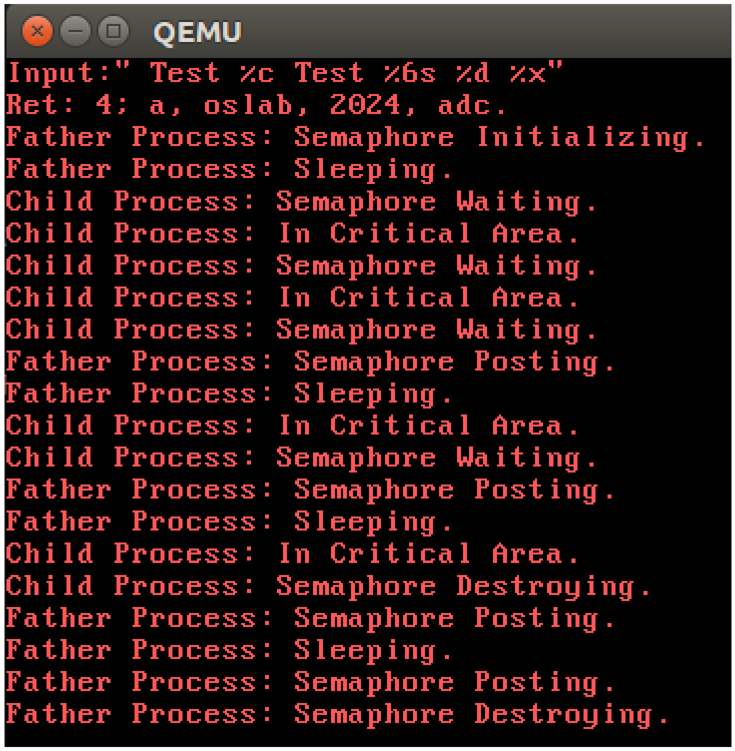
- 实现getpid()函数：获得当前进程的pid
- 解决生产者消费者问题：通过创建四个生产者进程和一个消费者进程，利用信号量 empty、full 和 mutex 来协调它们之间的操作，验证了生产者-消费者问题的解决方案，确保生产者在缓冲区有空槽时生产，消费者在有产品时消费，并通过互斥信号量避免竞态条件，实现正确同步。具体的，设置缓冲区大小为2，每个生产者生产两个产品就停止。

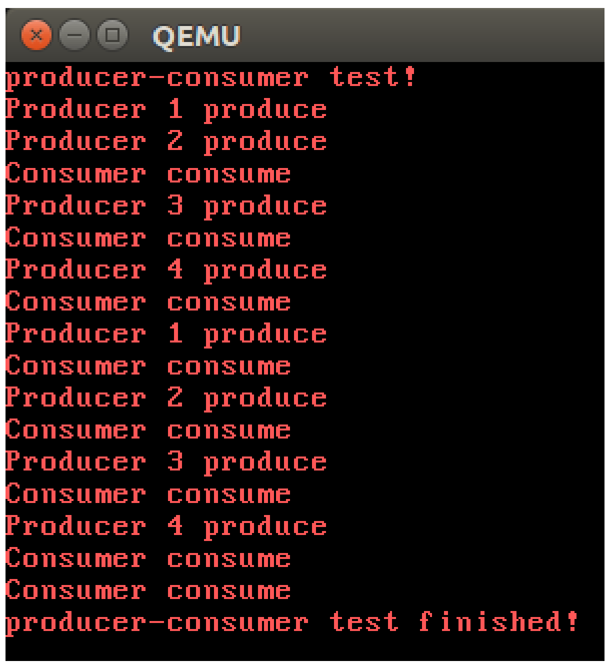
```c
int id = getpid();
// producer
if(id < 5){
    for(int i=0;i<num_to_produce;i++){
        sleep(128);
        sem_wait(&empty);
        sleep(128);
        sem_wait(&mutex);
        sleep(128);
        printf("Producer %d produce\n", id);
        //if(i==1)printf("Producer %d finished\n", id);
        sleep(128);
        sem_post(&mutex);
        sleep(128);
        sem_post(&full);
        sleep(128);
    }
}
// consumer
else if(id == 5){
    for(int i=0;i<4*num_to_produce;i++){
        sleep(128);
        sem_wait(&full);
        sleep(128);
        sem_wait(&mutex);
        sleep(128);
        printf("Consumer consume\n");
        sleep(128);
        sleep(128);
        sem_post(&mutex);
        sleep(128);
        sem_post(&empty);
        sleep(128);
    }
}
```

4. 实验结果

在lab4文件夹下执行如下命令：

```
chmod +x utils/genBoot.pl # 需要首先将genBoot.pl文件提权
chmod +x utils/genKernel.pl # 需要首先将genKernel.pl文件提权
make
make play
```


QEMU
Input:" Test %c Test %6s %d %x"
Ret: 4; a, oslab, 2024, adc.
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.


QEMU
producer-consumer test!
Producer 1 produce
Producer 2 produce
Consumer consume
Producer 3 produce
Consumer consume
Producer 4 produce
Consumer consume
Producer 1 produce
Consumer consume
Producer 2 produce
Consumer consume
Producer 3 produce
Consumer consume
Producer 4 produce
Consumer consume
Consumer consume
producer-consumer test finished!