Lab4实验报告

201300086 史浩男 1306189897@qq.com

一、实验进度和版本差异

我完成了所有必做内容

版本信息

- `Ubuntu 21.10`
- `gcc (Ubuntu 11.2.0-7ubuntu2) 11.2.0`

修改bootmain

- 需要将框架代码中三行注释掉，才能boot成功

```
5
6   void bootMain(void) {
7       int i = 0;
8       //int phoff = 0x34;
9       int offset = 0x1000;
10      unsigned int elf = 0x100000;
11      void (*kMainEntry)(void);
12      kMainEntry = (void(*)(void))0x100000;
13
14      for (i = 0; i < 200; i++) {
15          readSect( dst: (void*)(elf + i*512),  offset: 1+i);
16      }
17
18      kMainEntry = (void(*)(void))((struct ELFHeader *)elf)->entry;
19      // phoff = ((struct ELFHeader *)elf)->phoff;
20      // offset = ((struct ProgramHeader *)(elf + phoff))->off;
21
22      for (i = 0; i < 200 * 512; i++) {
23          *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
24      }
25
26      kMainEntry();
27  }
```

修改../bootloader/Makefile

- 添加如图所示四行，即增加权限、运行脚本、缩小扇区
- 修改之后才能成功一键make

```
bootloader.bin: $(BOBJS)
    $(LD) $(LDFLAGS) -e start -Ttext 0x7c00 -o bootloader.elf $(BOBJS)
    #objcopy -O binary bootloader.elf bootloader.bin
    #@../utils/genBoot.pl bootloader.bin
    objcopy -S -j .text -O binary bootloader.elf bootloader.bin
    chmod +x ../utils/genBoot.pl
    chmod +x ../utils/genKernel.pl
    ../utils/genBoot.pl bootloader.bin
```

二、实验思路

4.1、scanf

4.1.1 keyboardHandle

- 本段代码唤醒阻塞在dev[STD_IN]上的一个进程
- 将 `dev[STD_IN].value` ++即活跃的进程数加一，修改该进程的 `state`、`sleeptime` 来唤醒

```
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));
    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);
}
```

4.1.2 syscallReadStdIn

- 如果 `dev[STD_IN].value == 0`，将进程阻塞在 `stdin` 上
- 进程被唤醒后，将输入缓冲区拷贝到传入的 buffer。

```
int sel = sf->ds;
char *str = (char*)sf->edx;
int size = sf->ebx; // MAX_BUFFER_SIZE, reverse last byte
int i = 0;
char character = 0;
asm volatile("movw %0, %%es"::"m"(sel));
while(i < size-1) {
    if(bufferHead != bufferTail){ // what if keyBuffer is overflow
        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));
            i++;
        }
    }
    else
        break;
}
asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
pcb[current].regs.eax = i;
return;
```

- 如果已经有进程在读，`value < 0`，直接返回-1

```
else if (dev[STD_IN].value < 0) { // with process blocked
    pcb[current].regs.eax = -1;
    return;
}
```

4.2 sem

4.2.1 SEM_INIT

`init`系统调用用于初始化信号量

- 参数 `value` 指定了信号量的初始值，初始化成功返回0，`value`变量通过 `edx` 传入
- 指针 `sem` 指向初始化成功的信号量，否则返回-1
- 双向链表的初始化模仿了 `initSem()`函数，最后将i通过 `eax` 返回
- 如果找不到空闲的信号量，返回-1

```
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 = (int32_t)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;
}
```

4.2.2 SEM_WAIT

对应信号量的P操作，使 `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

- `sem` 变量通过 `edx` 传入，操作不成功的情况和 `sem_post()` 相同
- 加入列表后将 `current` 进程的的 `state` 设为阻塞状态， `sleeptime` 设为-1，执行中断 `int 0x80` 完成了对自身的阻塞。

```
void syscallSemWait(struct StackFrame *sf)
{
    int i = (int)sf->edx;
    if (i < 0 || i >= MAX_SEM_NUM){
        pcb[current].regs.eax = -1;
        return;
    }
    if (--sem[i].value < 0){
        push_sem(current , i);
        pcb[current].state = STATE_BLOCKED;
        asm volatile("int $0x20");
    }
    sf->eax = 0;
}
```

4.2.3 SEM_POST

对应信号量的V操作，使 `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

- 但当传入的参数超出了数组的范围，或 `state == 0` 即指向的信号量不在工作不能操作成功

- 如果 `state == 1` ， `value++` ，通过 `eax` 返回0。如果 `value <= 0` ，则取出一个进程，并且设置 `state` 和 `sleeptime` 来唤醒它

```
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;
    }
    if (sem[i].value >= 0) {
        sem[i].value ++;
        pcb[current].regs.eax = 0;
        return;
    }
    if (sem[i].state == 0) {
        pcb[current].regs.eax = -1;
        return;
    }
    if (sem[i].value < 0) { // release process blocked on this sem
        sem[i].value ++;
        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);
        pcb[current].regs.eax = 0;
        return;
```

4.2.4 SEM_DESTROY

用于销毁 `sem` 指向的信号量，销毁成功则返回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

相比之前三个较为简单，不再赘述。

4.3基于信号量解决进程同步问题

哲学家就餐问题

1、增加了 `get_pid()` 功能，用于获取当前进程的 `pid` 。只需在syscall.c中将 `eax = 7` 传入 syscall() 函数

2、之后需要在 `irqhandle` 中返回 `current` 值

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

3、另外我将 `sleep(128)` 改为了每次 `sleep(rand() % 128)` ，这里的 `rand` 采用了固定的 `seed` 。结果证明这样确实可以避免每次固定的间隔导致有些内容测试不到，测试更加全面。

```
    while (1)
    {
        if (i % 2 == 0){
            sem_wait(&forks[i]);
            sleep(rand() % 128);
            sem_wait(&forks[(i + 1) % N]);
        }
        else{
            sem_wait(&forks[(i + 1) % N]);
            sleep(rand() % 128);
            sem_wait(&forks[i]);
        }
        printf("Philosopher %d: eat\n", i);
        sleep(rand() % 128 + 64); // eat
        printf("Philosopher %d: think\n", i);
        sem_post(&forks[i]);
        sleep(rand() % 128);
        sem_post(&forks[(i + 1) % N]);
        sleep(rand() % 128 + 64); // think
    }
}
void testphilosopher()
{
    sem_t forks[N];
    int id;
    for (int i = 0; i < N; ++i)
        sem_init(&forks[i], 1);
    for (int i = 0; i < N - 1; ++i)
    {
        int ret = fork();
        //printf("ret=%d",ret);
        if (ret == -1)
        {
            printf("fork error: %d\n", i);
            exit();
        }
        if (ret == 0)
        {
            id = get_pid();
            printf("id=%d  ",id);
            philosopher(id, forks);
        }
    }
    id = get_pid();
    printf("id=%d  ",id);
    philosopher(id, forks);
}
```

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

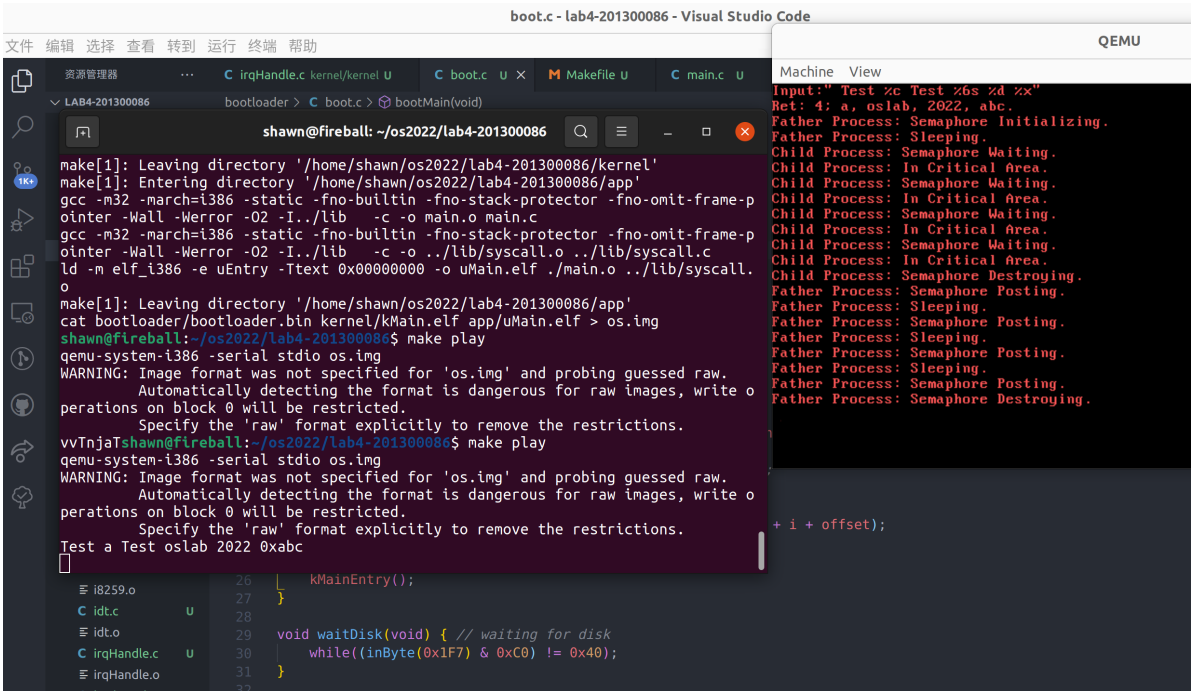
三、实验结果

app/main.c修改

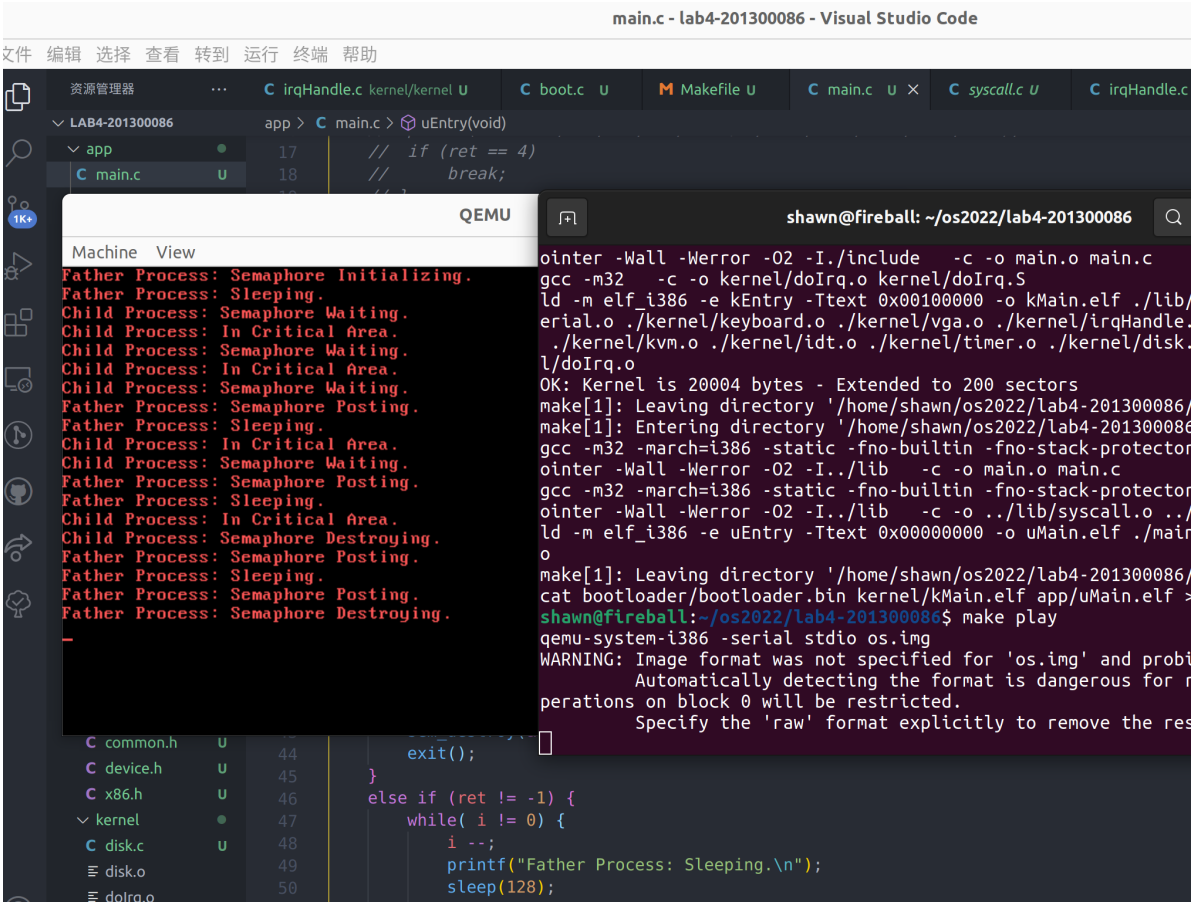
- 分别将Lab4的三部分封装起来，测试时注意把其他注释掉

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

    // For lab4.2
    // Test 'Semaphore'
    testsemaphore();

    // For lab4.3
    // TODO: You need to design and test the philosopher problem.
    // Producer-Consumer problem and Reader& Writer Problem are optional.
    // Note that you can create your own functions.
    // Requirements are demonstrated in the guide.
    testphilosopher();
    return 0;
}
```

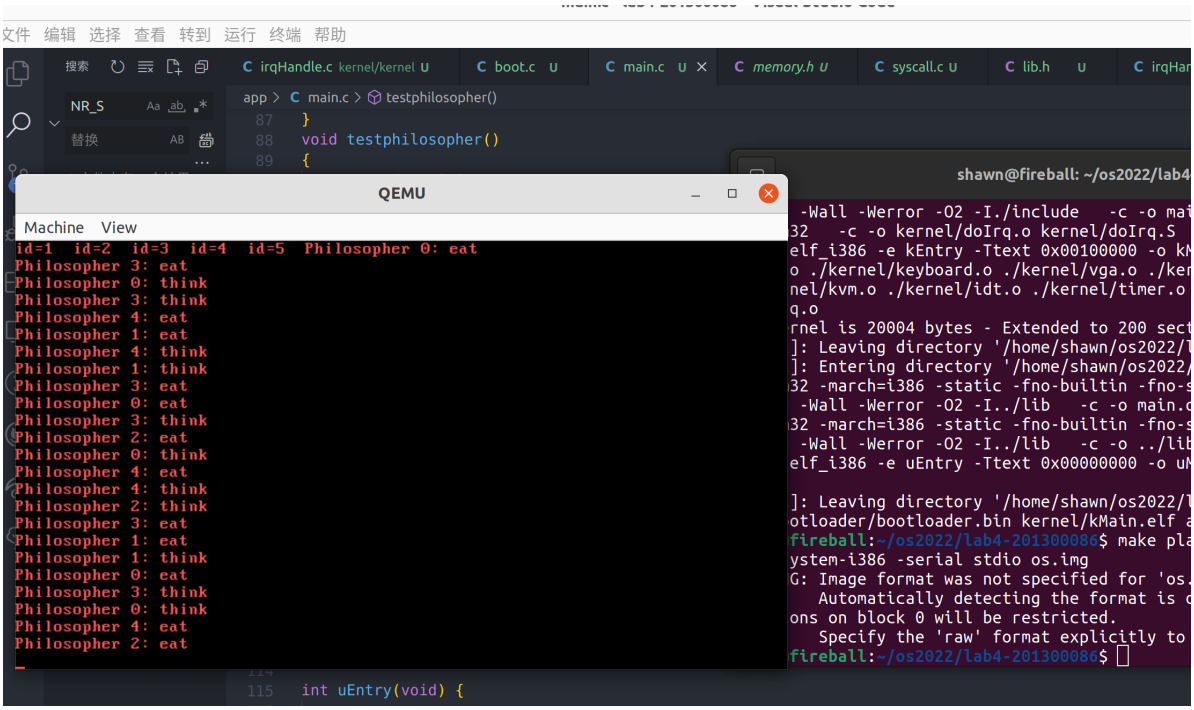
4.1scanf测试



4.2sem测试



4.3哲学家问题



四、实验感想

深刻体会到了信号量相关操作的实现

为了解决处理进程同步的问题，充分阅读了框架代码

了解了scanf的实现原理以及其他相关的部分代码

修改makefile可以快速解决框架版本问题