

lab4

191220138 杨飞洋

实验进度

已基本完成所有内容，但是 `scanf()` 函数仍然有问题，难以解决，十分抱歉。

背景知识

信号是一种抽象数据类型，由一个整型（sem）变量和两个原子操作组成；

P()

```
sem--  
如sem<0，进入等待，否则继续
```

V()

```
sem++  
如sem<=0，唤醒一个等待进程
```

信号量的实现（伪代码）：

```
class Semaphore {  
    int sem;  
    waitQueue q;  
}  
Semaphore::P(){  
    sem--;  
    if(sem < 0){  
        Add this thread t to q;  
        block(t)  
    }  
}  
Semaphore::V(){  
    sem++;  
    if(sem <= 0){  
        Remove a thread t from q;  
        wakeup(t);  
    }  
}
```

数据结构和一些宏

syscall相关

```
#define SYS_WRITE 0
#define SYS_FORK 1
#define SYS_EXEC 2
#define SYS_SLEEP 3
#define SYS_EXIT 4

#define STD_OUT 0
#define STD_IN 1

#define MAX_BUFFER_SIZE 256
```

进程结构

```
#define MAX_STACK_SIZE 1024
#define MAX_PCB_NUM ((NR_SEGMENTS-2)/2)

#define STATE_RUNNABLE 0
#define STATE_RUNNING 1
#define STATE_BLOCKED 2
#define STATE_DEAD 3

#define MAX_TIME_COUNT 16

struct StackFrame {
    uint32_t gs, fs, es, ds;
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
    uint32_t irq, error;
    uint32_t eip, cs, eflags, esp, ss;
};

struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct StackFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
};

typedef struct ProcessTable ProcessTable;
```

信号量相关

```
#define MAX_SEM_NUM 8

struct Semaphore {
```

```

    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this semaphore
};
typedef struct Semaphore Semaphore;

#define MAX_DEV_NUM 4

struct Device {
    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this device
};
typedef struct Device Device;

Semaphore sem[MAX_SEM_NUM];
Device dev[MAX_DEV_NUM];

```

有用的全局变量

```

SegDesc gdt[NR_SEGMENTS]; // the new GDT, NR_SEGMENTS=10, defined in
x86/memory.h
TSS tss;

ProcessTable pcb[MAX_PCB_NUM]; // pcb
int current; // current process

```

实验目的

本实验要求实现操作系统的信号量及对应的系统调用，然后基于信号量解决哲学家就餐问题

实验过程

实现格式化输入函数scanf

框架代码已经实现的差不多了，我们需要完成的就是 `keyboardhandle()` 和 `syscallstdin()`

`keyboardHandle` 要做的事情就两件：

将读取到的keycode放入到keyBuffer中
唤醒阻塞在dev[STD_IN]上的一个进程

第一件事情框架代码已经做好了，我们只需要完成第二件。

将 `dev[STD_IN].value++;`，即活跃的进程数加一。

利用手册的指导来取出一个阻塞的进程。

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);
```

并且修改该进程的 `state`、`sleeptime` 来唤醒她

部分代码如下：

```
dev[STD_IN].value++;
ProcessTable *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;
```

接下来看 `syscallstdin()`

它要做的事情也就两件：

如果 `dev[STD_IN].value == 0`，将当前进程阻塞在 `dev[STD_IN]` 上
进程被唤醒，读 `keyBuffer` 中的所有数据

首先阻塞进程的代码手册中有给出，直接照搬即可，再设置 `state`、`sleeptime` 来阻塞他，之后加上 `int $0x80` 在此中断。

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);
```

等待进程被唤醒之后，读数据，手册中也给到了详尽的指导

和实验2中 `printf` 的处理例程类似，以下代码可以将读取的字符 `character` 传到用户进程

```
int sel = sf->ds;
char *str = (char *)sf->edx;
int i = 0;
asm volatile("movw %0, %%es:::m"(sel));
asm volatile("movb %0, %%es:(%1)":"r"(character), "r"(str + i));
```

在此基础上遍历 `beybuffer[]`，用 `getchar()` 函数将键盘码转化为 `char` 型。用 `bufferhead`、`buffertail` 来判断是否读完。

将字节数即 `i` 返回给 `eax`，当然如果 `value < 0` 直接返回-1即可。

代码如下：

```

int sel = sf->ds;
char *str = (char*)sf->edx;
int size = sf->ebx; // MAX_BUFFER_SIZE, 256, reverse last byte
int i = 0;
char character = 0;
asm volatile("movw %0, %%es"::"m"(sel));
//putChar(size+48);
//putInt(size);
while (bufferHead != bufferTail)
{
    if (i < size - 1)
    {
        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;
}
//updateCursor(displayRow, displayCol);
asm volatile("movb $0x00, %%es:(%0)"::"r"(str+i));
pcb[current].regs.eax = i;
return;
}
else if (dev[STD_IN].value < 0) { // with process blocked
    pcb[current].regs.eax = -1;
    return;
}
}

```

但是实际上在按照手册要求实现之后，`scanf()` 无法正常运行，更加让我无奈的是，当我想用 `putchar()`、`putInt()`、`putstr()` 等输出函数想要进行调试时，程序的行为会发生极大的变化，就只是加输出语句，就会改变行为，让我根本不知道问题何在，无奈之下，只得放弃，对此十分抱歉。

实现信号量相关系统调用

SEM_INIT

`sem_init()` 系统调用用于初始化信号量，其中参数 `value` 用于指定信号量的初始值，初始化成功则返回0，指针 `sem` 指向初始化成功的信号量，否则返回-1

可以遍历 `sem[]` 数组找到一个空闲信号量，将其 `state` 设置为1，根据 `syscall.c` 中 `*sem = syscall(SYS_SEM, SEM_INIT, value, 0, 0, 0);` 可知 `value` 变量通过 `edx` 传入，对于双向链表的初始化可以模仿 `initSem()` 函数来做，最后将通过 `eax` 返回。

如果找不到空闲的信号量，返回-1.

`irqhandle` 中代码如下：

```

int i = 0;
for (; i < MAX_SEM_NUM ; i++) {
    if (sem[i].state == 0){
        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;
        return;
    }
}
pcb[current].regs.eax = -1;

```

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

有两种情况不能操作成功：

- 1.传入的参数超出了数组的范围
- 2.指向的信号量不在工作，即 `state == 0`

如果 `state == 1`，`value++`，通过 `eax` 返回0。如果 `value <= 0`，则可以用手册上的代码取出一个进程，并且设置 `state`, `sleeptime` 来唤醒它。

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);

```

根据syscall.c中 `return syscall(SYS_SEM, SEM_POST, *sem, 0, 0, 0);`可知 `sem` 变量通过 `edx` 传入

`irqhandle` 中代码如下：

```

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) {
        ProcessTable *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);
    }
}

```

```

    }
}
else
    pcb[current].regs.eax = -1;
}

```

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

首先还是根据`syscall.c`中 `return syscall(SYS_SEM, SEM_WAIT, *sem, 0, 0, 0);`可知 `sem` 变量通过 `edx` 传入

操作不成功的情况和 `sem_post()` 相同，在此不多赘述。

在这里主要说一下如何阻塞自身。

在手册中给出了将`current`进程加到相应的阻塞列表的代码，可以复用一下。

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);

```

在加入列表之后，将`current`进程的 `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。

`irqhandle` 中代码如下：

```

int i = (uint32_t)sf->edx;
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) {
        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;

```

SEM_DESTROY

`sem_destroy()` 系统调用用于销毁 `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

`sem` 还是通过 `edx` 传入。

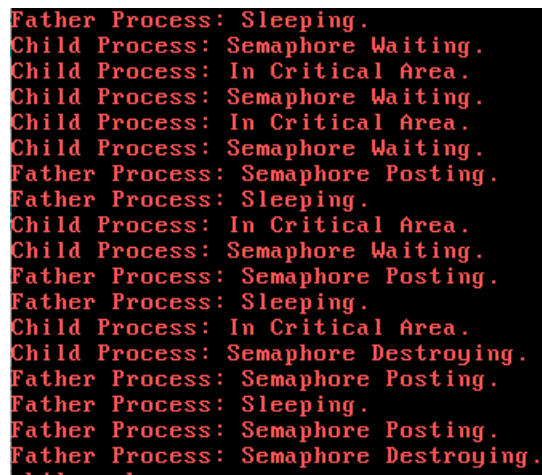
操作不成功的情况和 `sem_post()` 相同，在此不多赘述。

将对应信号量的 `state` 置为0，开启中断则销毁成功。

`irqhandle` 中代码如下：

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

测试结果



```
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.
```

分析打印结果：

父进程初始化信号量，`value = 2`，`fork` 出子进程,打印 `sleeping` 后便去睡眠。

由于信号量的值为 2，子进程可以进入两次critical area，在第三次时被阻塞，释放父进程，父进程苏醒，释放信号量,又去睡眠

被阻塞的子进程释放,再次进入关键区,又被阻塞，之后同上。

被阻塞的子进程释放,销毁信号量,退出。

只有父进程运行,最终销毁信号量。

基于信号量解决进程同步问题

哲学家就餐问题

问题描述:

5个哲学家围绕一张圆桌而坐
桌子上放着5支叉子
每两个哲学家之间放一支
哲学家的动作包括思考和进餐
进餐时需要同时拿到左右两边的叉子
思考时将两支叉子返回原处
如何保证哲学家们的动作有序进行? 如: 不出现有人永远拿不到叉子

首先实现 `getpid()` 功能, 获取当前进程的 `pid`

在 `syscall.c` 中将 `eax = 7` 传入 `syscall()` 函数

```
int getpid() {  
    return syscall(7, 0, 0, 0, 0, 0);  
}
```

在 `irqhandle` 中返回 `current` 值就行了。

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

再对宏做出一些改变, 本来是

```
#define MAX_SEM_NUM 4
```

```
#define MAX_DEV_NUM 4
```

```
#define MAX_PCB_NUM ((NR_SEGMENTS-2)/2)
```

但是哲学家五个人五个叉子, 需要将最大容量增加一下, 增加到5就行了。

利用手册中给出的伪码, 用 `printf()` 函数来表示 `think` 还是 `eat`, PV操作用 `sem_wait()` 和 `sem_post` 函数替代, 注意传入的是地址。

每次用 `getpid()` 获取当前进程, 在操作之间用 `sleep(128)` 间隔, 函数代码如下:

```
void philosopher(int i){  
    int id = getpid();  
    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)%N]);
            sleep(128);
        }
        else{
            sem_wait(&forks[(i+1)%N]);
            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)%N]);
        sleep(128);
    }
}

```

在main函数中的调用：

首先利用 `sem_init()` 来初始化进程。

然后调用 `fork()` 函数切换进程，执行对应的 `philosopher()` 函数。

最后利用 `sem_destroy()` 来销毁进程。

```

for (int i = 0; i < 5; i++){
    sem_init(&forks[i], 1);
}

for (int i = 0; i < 5; i++){
    if(fork() == 0){
        philosopher(i);
        exit();
    }
}
exit();
for (int i = 0; i < 5; i++){
    sem_destroy(&forks[i]);
}

```

部分结果截图：

```
Philosopher 2 : think  
Philosopher 3 : think  
Philosopher 4 : think  
Philosopher 5 : think  
Philosopher 6 : think  
Philosopher 2 : eat  
Philosopher 5 : eat  
Philosopher 2 : think  
Philosopher 3 : eat  
Philosopher 5 : think  
Philosopher 6 : eat  
Philosopher 3 : think  
Philosopher 4 : eat  
Philosopher 6 : think  
Philosopher 2 : eat  
Philosopher 4 : think  
Philosopher 5 : eat  
Philosopher 2 : think  
Philosopher 3 : eat
```

实验心得

学习了基于信号量的进程同步机制，并且利用代码初步实现。