OS-Lab4说明文档

- 基于chapter6-o修改

1 添加系统调用

1.1 进程休眠

在 `syscall.asm` 中添加如下代码：

```
; 添加系统调用
delay_milli_seconds:
    mov ebx, [esp + 4]
    mov eax, _NR_delay_milli_seconds
    int INT_VECTOR_SYS_CALL
    ret
```

在proc.c中实现系统调用函数再注册到 `global.c:PUBLIC system_call sys_call_table[NR_SYS_CALL]`
在const.h中修改 `NR_SYS_CALL`
syscall中的函数名通 `proto.h` 引入

```
PUBLIC void sys_delay_milli_seconds(int milli_second)
{
    p_proc_ready->sleep_time = milli_second; //新增的记录进程休眠时间的成员变量
    int index = find(); // 就绪队列中寻找， -1代表不在或阻塞
    if (index != -1) remove(index); // 找到就移除
    schedule(); //切换当前进程
    restart();  //切换堆栈实现进程的迁移
}
```

同时为了唤醒进程对 `clock.c:PUBLIC void clock_handler(int irq)` 进行了修改

1.2 字符串打印

与1.1一样， `syscall.asm` 中的global标签名为：

```
print_str:
    mov ebx, [esp + 4]
    mov eax, _NT_print_str
    int INT_VECTOR_SYS_CALL
    ret
```

2 添加两个系统调用执行信号量PV操作，在此基础上模拟读者写者问题

2.1 添加两个系统调用执行信号量PV操作

定义信号量:

```
typedef struct semaphore
{
    int value;
    int size;
    char name[32];
    PROCESS *list[MAX_WAIT_PROCESS];
} SEMAPHORE;
```

添加P、V操作的系统调用方式与1.1相同

2.2 PV操作实现

封装系统调用，进入时关中断，退出时开中断，以实现PV操作原子性的要求

```
// main.c
void atomicP(SEMAPHORE *s)
{
    disable_int();
    P(s);
    enable_int();
}

void atomicV(SEMAPHORE *s)
{
    disable_int();
    V(s);
    enable_int();
}
```

2.2.1 P操作具体实现

如果信号量足够则允许申请，否则执行 sleep() 函数，将当前进程从就绪队列中移出以保证当前进程不再被调度，然后执行 schedule() 和 start() 方法切换进程

```
PUBLIC void sys_p(SEMAPHORE *s)
{
    s->value--;
    if (s->value < 0) sleep(s);
}

void sleep(SEMAPHORE *s)
{
    //需要将当前的进程从可调度队列中移除
    int index = find();
    if (index != -1) remove(index);

    p_proc_ready->isWaiting = 1;
    s->list[s->size] = p_proc_ready;
    s->size++;

    schedule();
    restart();
}
```

2.2.2 V操作具体实现

释放信号量，唤醒等待此信号量的队首进程

```
PUBLIC void sys_v(SEMAPHORE *s)
{
    s->value++;
    if (s->value <= 0) wakeup(s);
}

wakeup(SEMAPHORE *s)
{
    if (s->size > 0)
    {
        push(s->list[0]->pid); //移入可调度队列

        s->list[0]->isWaiting = 0; //从当前信号量的等待队列中移出队首的等待进程
        for (int i = 0; i < s->size - 1; i++)
        {
            s->list[i] = s->list[i + 1];
        }
        s->size--;
    }
}
```

2.3 模拟读者写者问题

2.3.1 添加读者写者进程

```
PUBLIC TASK task_table[NR_TASKS] = {
    {ReadB, STACK_SIZE_ReadB, "ReadB"},
    {ReadC, STACK_SIZE_ReadC, "ReadC"},
    {ReadD, STACK_SIZE_ReadD, "ReadD"},
    {WriteE, STACK_SIZE_WriteE, "WriteE"},
    {WriteF, STACK_SIZE_WriteF, "WriteF"},
    {A, STACK_SZIE_A, "A"}
};
```

修改同时读要修改 `global.c` 中的： `nr_readers` 第一个元素
修改每个进程休息时间要在 `main.c` 中修改每个进程读写完后的休眠时间

2.3.2 READ FIRST

```
void read(int slices)
{
    if (mode == 0)
    { //读者优先
        atomicP(&rmutex);
        if (readCount == 0)
            atomicP(&rw_mutex); //有进程在读的时候不让其它进程写
        readCount++;
        atomicV(&rmutex);

        atomicP(&nr_readers);
        disp_read_start();

        //读操作消耗的时间片
        //milli_delay(slices * TIMESLICE);
        for (int i = 0; i < slices; i++)
        {
            disp_reading();
            milli_delay(TIMESLICE);
        }

        disp_read_end();
        atomicV(&nr_readers);

        atomicP(&rmutex);
        readCount--;
        if (readCount == 0)
            atomicV(&rw_mutex);
        atomicV(&rmutex);
    } else if (mode == 1) //write first
}

void write(int slices)
{
    if (mode == 0)
    {
        //读者优先
        atomicP(&rw_mutex);
        writeCount++;
        disp_write_start();
        //milli_delay(slices * TIMESLICE);
        for (int i = 0; i < slices; i++)
        {
            disp_writing();
            milli_delay(TIMESLICE);
        }
        disp_write_end();
        writeCount--;
        atomicV(&rw_mutex);
    } else if (mode == 1) //write first
}
```

2.3.3 WRITE FIRST

```
void read(int slices)
{
    if (mode == 0)
    { /* 读者优先 */ }
    else if (mode == 1)
    {
        //P(&queue); //增加queue信号量是为了防止r上有长队列，因为如果r上有长队列的话，如果有写进程，
        P(&r);
        P(&rmutex);
        if (readCount == 0)
            P(&w);
        readCount++;
        V(&rmutex);
        V(&r);
        //V(&queue);

        atomicP(&nr_readers);
        disp_read_start();

        //读操作消耗的时间片
        //milli_delay(slices * TIMESLICE);
        for (int i = 0; i < slices; i++)
        {
            disp_reading();
            milli_delay(TIMESLICE);
        }

        disp_read_end();
        atomicV(&nr_readers);

        atomicP(&rmutex);
        readCount--;
        if (readCount == 0)
        {
            atomicV(&w);
        }
        atomicV(&rmutex);
    }
}

void write(int slices)
{
    if (mode == 0)
    { /* 读者优先 */ }
    else if (mode == 1)
    {
        P(&wmutex);
        if (writeCount == 0)
            P(&r); //申请r锁
        writeCount++;
        V(&wmutex);

        P(&w);
        disp_write_start();
```

```
        //milli_delay(slices * TIMESLICE);
        for (int i = 0; i < slices; i++)
        {
            disp_writing();
            milli_delay(TIMESLICE);
        }
        disp_write_end();
        V(&w);

        P(&wmutex);
        writeCount--;
        if (writeCount == 0)
            V(&r);
        V(&wmutex);
    }
}
```

2.3.4 A PROCESS

只负责每个时间片打印其他进程的消息

2.3.5 解决饿死问题

对于A进程的调度特殊处理，检查上次调度F进程的时间与当前时间的差值，如果超过预设值，则强制将A进程设为运行态，并更新A进程的调度时间。

为了防止饿死，为每一个进程添加了一个变量 int hasWorked; 来标识此进程是否在本轮调度中执行过，如果已经执行过，则不再给此进程分配时间片，一轮结束后，清空此变量，重新恢复调度。因为读写饿死问题是因为读者反复请求读，写者反复请求写，导致读优先写进程无法进入，写优先读进程无法进入，保证每个进程每轮调度一次即可解决该问题。