操作系统实验四说明文档

191850205 夏宇

1 添加系统调用实现进程休眠

在syscall.asm中添加如下代码

```
delay_milli_seconds:
    mov ebx, [esp + 4]                  ;ebx保存传过来的时间参数
    mov eax, _NR_delay_milli_seconds            ;eax保存在系统调用函数表中的偏移量
    int INT_VECTOR_SYS_CALL             ;调用系统调用
    ret
```

在global.c的系统调用表中添加相应函数

```
PUBLIC system_call sys_call_table[NR_SYS_CALL] = {sys_get_ticks,
sys_delay_milli_seconds, disp_str, sys_p, sys_v}; //系统调用的函数
```

然后修改const.h中定义的系统调用数量，然后在proc.c中实现该系统调用函数

```
PUBLIC void sys_delay_milli_seconds(int milli_second) //todo
{
    p_proc_ready->sleep_time = milli_second; //睡眠时间以tick为单位，一个tick就是对应的一个ms
    int index = find();                     //把该进程移出可调度队列
    if (index != -1)
    {
        remove(index);
    }
    schedule();
    restart();
}
```

为了实现指定时间的休眠函数，我在进程结构体定义中添加了新的变量 `int sleep_time;`，该变量记录了此进程需要休眠的时长，然后将此进程从就绪队列中移出，以保证此进程在休眠时间中不被调度。然后立即执行 `schedule()` 函数切换当前进程，接着执行 `restart()` 函数切换堆栈，实现进程的迁移。

为了实现进程的唤醒，我修改了clock.c中处理时钟中断的方法，在每次时钟中断发生时，检查所有进程，若有进程在休眠，则将其休眠时间片减一，如果休眠时间结束，则将其重新移入就绪队列

```
PUBLIC void clock_handler(int irq)
{
        ticks++;
        /* p_proc_ready->ticks--; */
```

```
        if (k_reenter != 0)
        {
                return;
        }

        for (int i = 0; i < NR_TASKS; i++)
        {
                if (proc_table[i].sleep_time > 0)
                {
                        proc_table[i].sleep_time--;
                        if (proc_table[i].sleep_time == 0)
                        {
                                //移入可调度队列
                                push(proc_table[i].pid);
                        }
                }
        }

        schedule();
}
```

2 添加系统调用打印字符串

与上述添加系统调用方法一致。

在syscall.asm中添加系统调用

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

然后在global.c中的sys_call_table中添加系统调用函数

3 添加系统调用执行PV操作，并模拟实现读者写者问题

3.1 添加系统调用执行PV操作

定义信号量，其成员包括信号量的值，信号量队列的大小，信号量队列中排队的进程，还有一个是该信号量的名称，此名称是为了方便我debug使用的，实际运行中并不需要。

```
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，与前面添加系统调用的方法一样，不再赘述

3.1.1 P、V操作的实现

下面讲一下P、V操作的具体实现。

由于P、V操作都是原子性的，所以我将系统调用的P、V操作再次封装，在进入时关中断，在退出时开中断，以实现原子性的要求。

```
void atomicP(SEMAPHORE *s)
{
    //disable_irq(CLOCK_IRQ);
    disable_int();
    P(s);
    enable_int();
    //enable_irq(CLOCK_IRQ);
}

void atomicV(SEMAPHORE *s)
{
    //disable_irq(CLOCK_IRQ);
    disable_int();
    V(s);
    enable_int();
    //enable_irq(CLOCK_IRQ);
}
```

P操作的具体实现。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(); //先找到当前进程在可调度队列中的下标，然后再删除当前进程
    if (index != -1)
    {
        remove(index); //todo
    }

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

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

V操作的具体实现。首先释放信号量，然后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);
    }
}

void 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--;
    }
}
```

3.2 模拟实现读者写者问题

3.2.1 添加进程

```
PUBLIC TASK task_table[NR_TASKS] = {{ReadA, STACK_SIZE_ReadA, "ReadA"},
                                    {ReadB, STACK_SIZE_ReadB, "ReadB"},
                                    {ReadC, STACK_SIZE_ReadC, "ReadC"},
                                    {WriteD, STACK_SIZE_WriteD, "WriteD"},
                                    {WriteE, STACK_SIZE_WriteE, "WriteE"},
                                    {F, STACK_SZIE_F, "F"}}; //用户进程
```

3.2.2 读优先的实现

读者优先饿死问题的解决在后续介绍 `schedule()` 函数时说明

读操作

```
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);
```

写操作

```
//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);
```

3.2.3 写者优先的实现

写者优先饿死问题的解决在后续介绍 `schedule()` 函数时说明

读操作

```
//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，那么写进程要排很久的队
        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);
```

写操作

```
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);
```

3.2.4 F进程

F进程是一个特殊的进程，它负责打印其余进程的运行，具体在 `schedule()` 函数中说明。

3.3 schedule()函数

```
PUBLIC void schedule()
{
    if (ticks - lastTicks >= 100)
    {
        lastTicks = ticks;
```

```
            p_proc_ready = proc_table + NR_TASKS - 1; //选中F进程
            isBlockedF = 0;
            return;
        }

        PROCESS *p;

        if (schedulable_queue_size == 0)
        {
            disp_str("no process is schedulable\n");
        }
        else
        {
            if (!numOfNotWorked())
            {
                reWork();
            }

            do
            {
                int process = schedulable_queue[0];
                p_proc_ready = proc_table + process;
                remove(0);      //删除
                push(process); //移到队末
            } while (p_proc_ready->hasWorked == 1);
        }
    }
```

首先是F进程，对于F进程的调度，我特殊处理，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F进程设为运行态，并更新F进程的调度时间。

其次是读写优先防止饿死，我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。