Lab3 实验报告

实验步骤以及结果

1. 完成库函数
   在./lib/syscall.c里完成系统调用

```
pid_t fork()
{
        // TODO:call syscall
        return syscall(SYS_FORK, 0, 0, 0, 0, 0);
        //return 0;
}

int sleep(uint32_t time)
{
        // TODO:call syscall
        return syscall(SYS_SLEEP, time, 0, 0, 0, 0);
        //return 0;
}

int exit()
{
        // TODO:call syscall
        return syscall(SYS_EXIT, 0, 0, 0, 0, 0);
        //return 0;
}
```

2. 时钟中断处理
   (1) 首先 遍历pcb，将状态为STATE_BLOCKED的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为STATE_RUNNABLE。
   (2) 将当前进程的timeCount加一，如果时间片用完（timeCount==MAX_TIME_COUNT）且有其它状态为STATE_RUNNABLE的进程，切换，否则继续执行当前进程

关于进程切换，即是在P1时间片用完后（timeCount==MAX_TIME_COUNT）,切换至就绪状态的进程P2，并从当前P1的内核堆栈切换至P2的内核堆栈。然后 从进程P2的内核堆栈中弹出P2的现场信息，切换至P2的用户态堆栈，从时间中断处理程序返回执行P2(也就是那一段内敛汇编代码)。

```
void timerHandle(struct StackFrame *sf)
{
	// TODO
	int i = (current + 1) % MAX_PCB_NUM;
	for(; i != current; i = (i + 1) % MAX_PCB_NUM){
		if(pcb[i].state == STATE_BLOCKED){//sleep状态
			if(pcb[i].sleepTime > 0){
				pcb[i].sleepTime--;
				if(pcb[i].sleepTime == 0) pcb[i].state = STATE_RUNNABLE;
			}
		}
	}
	if(pcb[current].state == STATE_RUNNING && pcb[current].timeCount < MAX_TIME_COUNT){
		pcb[current].timeCount++;
		return;
	}
	if(pcb[current].state == STATE_RUNNING && pcb[current].timeCount >= MAX_TIME_COUNT){
		pcb[current].timeCount = 0;
		pcb[current].state = STATE_RUNNABLE;
	}
	for(i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM){
		if(pcb[i].state == STATE_RUNNABLE && i!=0){
			current = i;
			break;
		}
	}
	if(pcb[i].state != STATE_RUNNABLE) current = 0;
	pcb[current].state = STATE_RUNNING;
	pcb[current].timeCount = 0;
	uint32_t tmpStackTop = pcb[current].stackTop;
	pcb[current].stackTop = pcb[current].prevStackTop;
	tss.esp0 = (uint32_t)&(pcb[current].stackTop);
	asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
	asm volatile("popl %gs");
	asm volatile("popl %fs");
	asm volatile("popl %es");
	asm volatile("popl %ds");
	asm volatile("popal");
	asm volatile("addl $8, %esp");
	asm volatile("iret");
}
```

3. 系统调用例程

3.1 syscallFork

寻找一个空闲的pcb(pcb[i].state == STATE_DEAD)做为子进程的进程控制块，将父进程的资源复

制给子进程。如果没有空闲pcb，则fork失败，父进程返回-1，成功则子进程返回0，父进程返回子进程pid(返回值存放在pcb[pos].regs.eax和pcb[current].regs.eax里)
注意复制进程控制块，是复制0x100000大小的内存空间。

```
void syscallFork(struct StackFrame* sf){
        //查找空闲pcb
        int pos = -1;
        int j;
        for(j = 0; j < MAX_PCB_NUM; ++j){
                if(pcb[j].state == STATE_DEAD){
                        pos = j;
                        break;
                }
        }
        if(pos == -1){//没有空闲pcb，父进程返回-1,
                //sf->eax = -1;
                pcb[current].regs.eax = -1;
                return;
        }
        //资源复制
        //memcpy((void* )((pos + 1) * 0x100000), (void* )((current + 1) * 0x100000), 0x100000);
        enableInterrupt();//开中断
        int i;
        for(i = 0; i < 0x100000; ++i){
                *(unsigned char *)(i + (pos + 1) * 0x100000) = *(unsigned char *)(i + (current
        }
        disableInterrupt();//关中断
        pcb[pos].stackTop = pcb[current].stackTop
    + (uint32_t)(&pcb[pos].stackTop) - (uint32_t)(&pcb[current].stackTop);
        pcb[pos].prevStackTop = pcb[current].prevStackTop
    + (uint32_t)(&pcb[pos].stackTop) - (uint32_t)(&pcb[current].stackTop);
        pcb[pos].regs.edi = pcb[current].regs.edi;
        pcb[pos].regs.esi = pcb[current].regs.esi;
        pcb[pos].regs.ebp = pcb[current].regs.ebp;
        pcb[pos].regs.xxx = pcb[current].regs.xxx;
        pcb[pos].regs.ebx = pcb[current].regs.ebx;
        pcb[pos].regs.edx = pcb[current].regs.edx;
        pcb[pos].regs.ecx = pcb[current].regs.ecx;
        pcb[pos].regs.eax = pcb[current].regs.eax;
        pcb[pos].regs.irq = pcb[current].regs.irq;
        pcb[pos].regs.error = pcb[current].regs.error;
        pcb[pos].regs.eip = pcb[current].regs.eip;
        pcb[pos].regs.eflags = pcb[current].regs.eflags;
        pcb[pos].regs.esp = pcb[current].regs.esp;

        pcb[pos].regs.cs = USEL(1 + 2 * pos);
        pcb[pos].regs.ss = USEL(2 + 2 * pos);
        pcb[pos].regs.ds = USEL(2 + 2 * pos);
        pcb[pos].regs.es = USEL(2 + 2 * pos);
        pcb[pos].regs.fs = USEL(2 + 2 * pos);
```

```
        pcb[pos].regs.gs = USEL(2 + 2 * pos);
        pcb[pos].state = STATE_RUNNABLE;
        pcb[pos].timeCount = pcb[current].timeCount;
        pcb[pos].sleepTime = pcb[current].sleepTime;
        pcb[pos].pid = pos;
        pcb[pos].regs.eax = 0;//子进程返回0
        pcb[current].regs.eax = pos;//父进程返回pid
}
```

3.2 syscallSleep

将当前的进程的sleepTime设置为传入的参数(ecx)，将当前进程的状态设置为STATE_BLOCKED，然后利用 `asm volatile("int $0x20");` 模拟时钟中断，利用 timerHandle 进行进程切换。
传入的参数应当不小于0，否则非法， `assert(0);` 。等于0的情况直接 `return;` 就行。

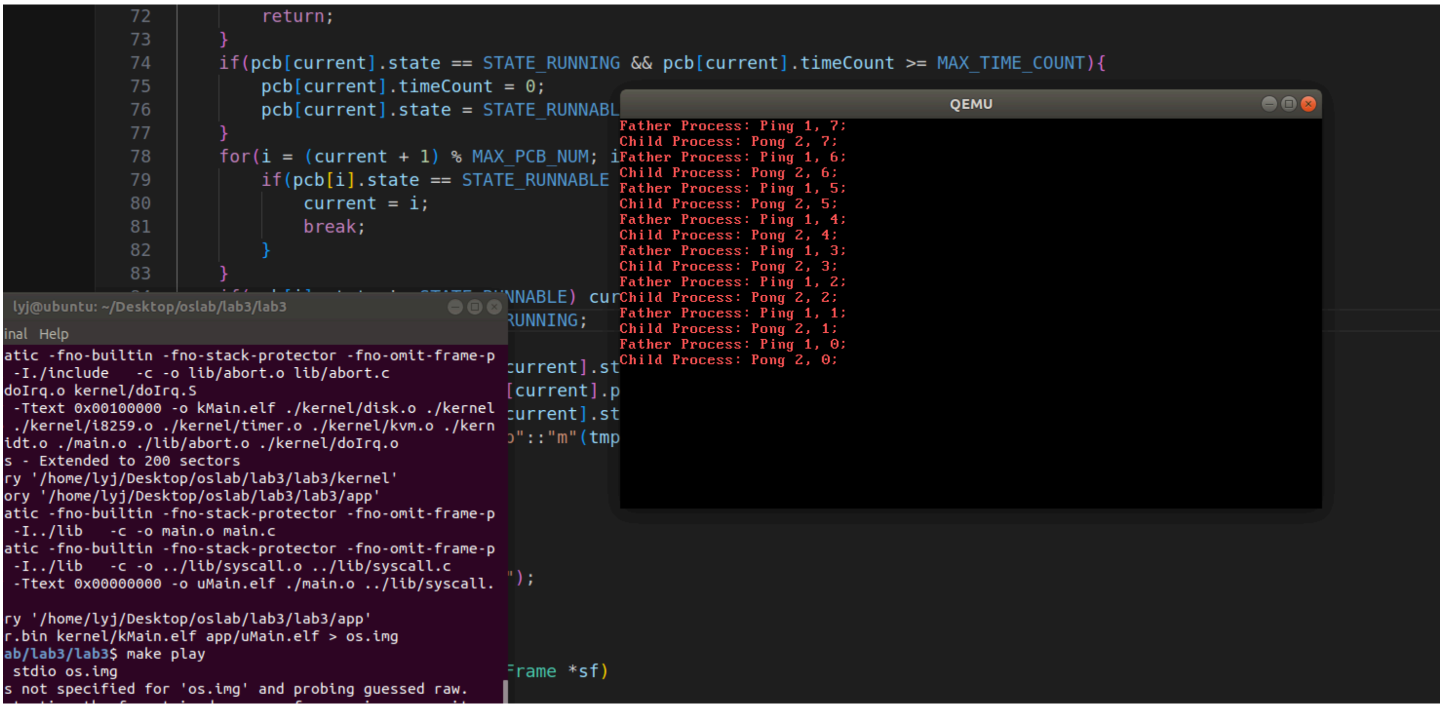
```
void syscallSleep(struct  StackFrame* sf)
{
        if(sf->ecx < 0) assert(0);//时间不能小于0
        if(sf->ecx == 0) return;
        pcb[current].sleepTime = sf->ecx;
        pcb[current].state = STATE_BLOCKED;//阻塞
        asm volatile("int $0x20");
}
```

3.3 syscallExit

将当前进程的状态设置为STATE_DEAD，然后模拟时钟中断进行进程切换

```
void syscallExit(struct StackFrame* sf)
{
        pcb[current].state = STATE_DEAD;
        asm volatile("int $0x20");
}
```

最终测试效果



自由报告

此次实验让我对进程管理，任务调度有了更深入的理解。