进程同步

李杨 161220071

实验目标:

1. 实现一个简单的生产者消费者程序，掌握基于信号量的进程同步机制。

2. 了解 sem_init(), sem_post(), sem_wait(), sem_destroy()这四个系统调用函数的作用。

实验背景:

SEM_INIT 系统调用:

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。

SEM_POST 系统调用:

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

SEM_WAIT 系统调用:

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

SEM_DESTROY 系统调用:

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

具体内容

定义信号量结构体 semaphore

```
Struct semaphore{
        Int value;
        Int cnt;
        Struct processblock* queue[10]
};
```

表项名称	含义
Value	信号量的值，由此进行 pv 原子操作
Count	阻塞队列的信号计数
queue[10]	阻塞的信号进程队列

在用户程序中，用 sem_t sem 来表示信号量。定位为 unsigned int 类型。将信号量作为参数传递时，传递的是 sem 的地址，将这个地址中的值保存到 semlist 中。

函数实现

关于信号量的原子操作主要有两类，P 操作和 V 操作

P 操作对信号量的值减一，如果信号量的值小于 0，则将

该进程阻塞，放入该信号量下的等待队列中等待，直到条件满足再次被唤醒之后再进入等待队列。

V 操作将信号量的值加一，当加一之后的信号量的值小于等于 0 时，就将阻塞在该信号量下的一个进程解除阻塞，将其加入到等待队列中。

P 操作:

```
void P(struct Semaphore* sem)
{
        disableInterrupt();
        sem->value--;
        if(sem->value <0)
                W(sem);
        enableInterrupt();
}
```

当信号量小于 0，则阻塞进程，调用 w

W 函数:

```
void W(struct Semaphore* sem){
        pcb_cur->state = BLOCKED;
        sem->queue[sem->cnt] = pcb_cur;
        sem->cnt++;
        scheduler();
}
```

V 操作:

```
void V(struct Semaphore* sem)
{
        disableInterrupt();
        sem->value++;
        if(sem->value <= 0)
                R(sem);
        enableInterrupt();
}
```

当信号量小于等于 0，则解除阻塞，调用 R

R 函数:

```
void R(struct Semaphore* sem)
{
        sem->queue[0]->state = RUNNABLE;
}
```

系统调用函数实现按照伪代码进行，对于地址进行判断是否合法，合法进行对应原子操作，并返回 0，否则返回-1

代码如下：

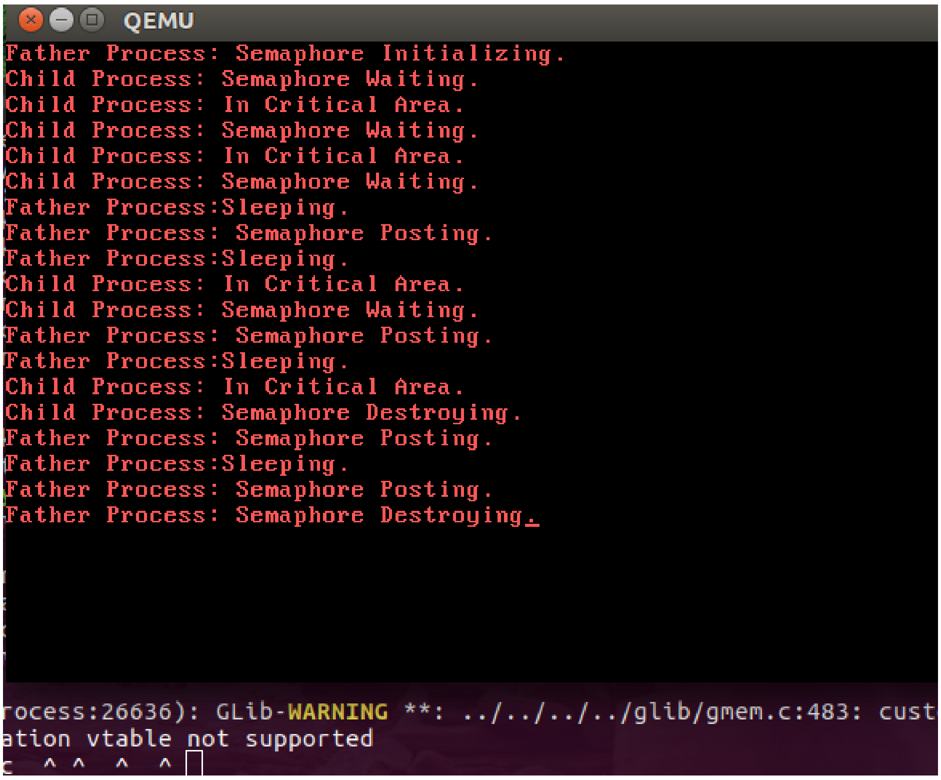
```
void sem_init(struct TrapFrame *tf){
	struct Semaphore* cursem = (struct Semaphore*)tf->ebx;
	begin_sem_addr = (void*)cursem;
	end_sem_addr = begin_sem_addr + sizeof(struct Semaphore
	semlist[sem_cnt] = *cursem;
	cursem->value = tf->ecx;
	cursem->cnt = 0;
	sem_cnt++;
	tf->eax = 0;
}

void sem_post(struct TrapFrame *tf){
	struct Semaphore* cursem = (struct Semaphore*)tf->ebx;
	void *addr =(void*)cursem;
	if(addr >= begin_sem_addr && addr <= end_sem_addr)
	{
		V(cursem);
		tf->eax = 0;
	}
	else
		tf->eax = -1;
}

void sem_wait(struct TrapFrame *tf){
	struct Semaphore* cursem = (struct Semaphore*)tf->ebx;
	void* addr =(void*)cursem;
	if(addr >= begin_sem_addr && addr <= end_sem_addr)
	{
		P(cursem);
		tf->eax = 0;
	}
	else
		tf->eax = -1;
}

void sem_destroy(struct TrapFrame *tf){
	struct Semaphore* cursem = (struct Semaphore*)tf->ebx;
	void* addr =(void*)cursem;
	if(addr >= begin_sem_addr && addr <= end_sem_addr)
	{
		tf->eax = 0;
	}
	else
		tf->eax = -1;
}
```

实验结果：


QEMU
Father Process: Semaphore Initializing.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process:Sleeping.
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.
rocess:26636): GLib-WARNING **: ../../../../glib/gmem.c:483: cust
ation vtable not supported