

# 《操作系统》实验报告

## LAB4 进程同步

姓名：陈攀岭

学号：171860516

邮箱：171860516@sma i l . n j u . e d u . c n

## 一、实验要求

本实验通过实现一个简单的生产者消费者程序，介绍基于信号量的进程同步机制。

内核：提供基于信号量的进程同步机制，并提供系统调用 `sem_init`、`sem_post`、`sem_wait`、`sem_destroy`。

库：对上述系统调用进行封装。

用户：对上述库函数进行测试。

### 1. 实现 SEM\_INIT、SEM\_POST、SEM\_WAIT、SEM\_DESTROY 系统调用

实现 `SEM_INIT`、`SEM_POST`、`SEM_WAIT`、`SEM_DESTROY` 系统调用，使用用户程序测试，并在实验报告中说明实验结果。

### 2. 多进程进阶

调整框架代码中的 `gdt`、`pcb` 等进程相关的数据和代码，使得你实现的操作系统最多支持到 8 个进程（包括内核 `idle` 进程），注意进程分配内存空间大小，不要越界。

### 3. 信号量进程同步进阶

理解 1. 中的测试程序，实现两个生产者、四个消费者的生产者消费者问题，不需要考虑进程调度，公平调度就行。

- 以函数形式实现生产者和消费者，生产者和消费者的第一个参数为 `int` 类型，用于区分不同的生产者和消费者；
- 每个生产者生产 8 个产品，每个消费者消费 4 个产品；
- 先 `fork` 出 6 个进程，通过 `getpid` 获取当前进程的 `pid`，简单粗暴的以 `pid` 小的进程为生产者；
- 信号量模拟 `buffer` 中的产品数量；
- 保证 `buffer` 的互斥访问；
- 每一行输出需要表达 `pid i, producer/consumer j, operation(, product k)`；
- `operation` 包括 `try lock`、`locked`、`unlock`、`produce`、`try consume`、`consumed`；
- 除了 `lock`、`unlock` 的其他操作需要包含括号中的内容；
- `i` 表示进程号，`j` 表示生产者/消费者编号，`k` 表示生产者/消费者的生产/消费的第几个产品；

## 二、实验原理

### 1. 信号量机制

内核维护 `Semaphore` 这一数据结构，并提供 `P`、`V` 这一对原子操作。

### 2. GETPID 系统调用

为了方便区分当前正在运行的进程，实验 4 要求实现一个 `getpid` 系统调用用于返回当前进程标识 `ProcessTable.pid`，不允许调用失败。

### 3. 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。

#### 4. 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。

#### 5. 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。

#### 6. 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。

### 三、实验步骤

#### 1. 实验任务

- (1) 完善 kernel/kernel/irqHandle.c 中的 syscallSemWait、syscallSemPost 和 syscallDestroy 函数；
- (2) 修改 app/main.c 中函数，实现一个简单的生产者消费者程序。

#### 2. 代码实现

##### (1) syscallSemWait 函数：

通过查看 lib/syscall.c 中 syscall 函数以及 sem\_wait 函数可知，函数 sem\_wait(sem\_t \*sem) 中参数 \*sem 保存于 EDX 寄存器中，通过栈帧保存的寄存器得到对应信号量表项。设置返回值为 0。将信号量表项 value 减 1，若 value 值小于零，阻塞自身，即将当前进程状态设置为阻塞（STATE\_BLOCKED），将当前进程的阻塞链表（blocked）添加至当前信号量的阻塞链表（pcb）上。内联汇编代码（“int \$0x20”）陷入时间中断以进行进程切换。

```
void syscallSemWait(struct StackFrame *sf) {
    int32_t i = (int32_t)sf->edx;
    //putChar('0'+i);
    sem[i].value--;
    sf->eax = 0;
    if(sem[i].value < 0) {
        pcb[current].state = STATE_BLOCKED;
        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);
        asm volatile("int $0x20");
    }
    return;
}
```

##### (2) syscallSemPost 函数：

通过查看 lib/syscall.c 中 syscall 函数以及 sem\_post 函数可知，函数 sem\_post(sem\_t \*sem) 中参数 \*sem 保存于 EDX 寄存器中，通过栈帧保存的寄存器得到对应信号量表项。设置返回值为 0。将对应信号量 value 值加 1，如果 value 值不大于零，释放一个阻塞在该信号量上的进程。通过信号量上的阻塞链表（pcb）得到需要释放的进程表项指针，将进程状态设置为等待（STATE\_RUNNABLE）。在信号量上的阻塞链表将该释放的进程链表项删去。陷入时间中断进行进程切换。

```

void syscallSemPost(struct StackFrame *sf) {
    int32_t i = (int32_t)sf->edx;
    sem[i].value++;
    sf->eax = 0;
    if(sem[i].value <= 0) {
        ProcessTable *pt = NULL;
        pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
                               (uint32_t)&((ProcessTable*)0)->blocked));
        pt->state = STATE_RUNNABLE;
        sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
        (sem[i].pcb.prev)->next = &(sem[i].pcb);
        asm volatile("int $0x20");
    }
    return;
}

```

### (3) syscallSemDestroy 函数:

通过查看 lib/syscall.c 中 syscall 函数以及 sem\_destroy 函数可知, 函数 sem\_destroy(sem\_t \*sem) 中参数\*sem 保存于 EDX 寄存器中, 通过栈帧保存的寄存器得到对应信号量表项。销毁对应信号量, 即将信号量状态设置为 1, 设置返回值为 0。

```

void syscallSemDestroy(struct StackFrame *sf) {
    int32_t i = (int32_t)sf->edx;
    sem[i].state = 0;
    sf->eax = 0;
    return;
}

```

### (4) 多进程进阶:

观察 kernel/include/x86/memory.h 中宏定义代码可知, 对于 GDT 表项, 出去空的第 0 项及指向 TSS 的最后一项外, 需要支持 8 个进程 (包括内核 idle 进程), 单个进程代码及数据段各虚一个表项, 因此 8 个进程需要  $2 \times 8 = 16$  个表项, 因此只需将 GDT 的大小更改为 18 即可实现对 8 个进程的支持。

```

// GDT entries
#define NR_SEGMENTS    18 //10          // GDT size

```

### (5) getpid 函数:

首先查看 lib/lib.h, 需要自行增加 GETPID 系统调用。观察系统调用号, 使用未使用的数字 7 表示 GETPID 的系统调用号。

```

#define SYS_WRITE 0
#define SYS_READ 1
#define SYS_FORK 2
#define SYS_EXEC 3
#define SYS_SLEEP 4
#define SYS_EXIT 5
#define SYS_SEM 6
#define SYS_GETPID 7

```

并增加 getpid 函数的声明。

```
int getpid(void);
```

在 lib/syscall.c 中增加 getpid 函数的实现。

```

int getpid(void) {
    return syscall(SYS_GETPID, 0, 0, 0, 0, 0);
}

```

在 kernel/kernel/irqHandle.c 中增加对系统调用号 SYS\_GETPID 的宏定义, 增加函数 syscallgetpid 函数实现。

```
#define SYS_WRITE 0
#define SYS_READ 1
#define SYS_FORK 2
#define SYS_EXEC 3
#define SYS_SLEEP 4
#define SYS_EXIT 5
#define SYS_SEM 6
#define SYS_GETPID 7
```

```
void syscallgetpid(struct StackFrame *sf) {
    sf->eax = pcb[current].pid;
}
```

更改 kernel/kernel/syscallHandle 函数，增加对系统调用好 SYS\_GETPID 的处理。

```
case SYS_GETPID:
    syscallgetpid(sf);
    break; // for SYS_GETPID
default:break;
```

### (6) producer 生产者函数:

在 app/main.c 增加 producer 函数，以进程号 pid，互斥信号量 mutex，缓冲区信号量 empty 和 full 为参数。对于生产者号，因为父进程进程号为 1，两个较小子进程号 2，3 为生产者，其生产者号为 1，2，即 pid-1，每个生产者生产 8 个产品，即单个生产者进行 8 次生产，每次生产中：首先观察缓冲区是否有空间，对 empty 信号量进行 P 操作（sem\_wait），输出生产者生产，延时处理（sleep(128)），然后准备锁定缓冲区，输出“try lock”，对互斥信号量 mutex 进行 P 操作（sem\_wait），输出已锁定“locked”，完成后对互斥信号量 mutex 进行 V 操作（sem\_post），输出“unlock”，最后对缓冲区信号量 full 进行 V 操作。

```
void producer(int pid, sem_t mutex, sem_t empty, sem_t full) {
    int i = pid - 1;
    for(int k = 0; k < 8; k++) {
        sem_wait(&empty);
        printf("pid %d, producer %d, produce, product %d\n", pid, i, k + 1);
        sleep(128);
        printf("pid %d, producer %d, try lock, product %d\n", pid, i, k + 1);
        sem_wait(&mutex);
        printf("pid %d, producer %d, locked\n", pid, i);
        sem_post(&mutex);
        printf("pid %d, producer %d, unlock\n", pid, i);
        sem_post(&full);
    }
}
```

### (7) consumer 消费者函数:

在 app/main.c 增加 consumer 函数，以进程号 pid，互斥信号量 mutex，缓冲区信号量 empty 和 full 为参数。对于消费者号，对应进程号 4、5、6、7 的消费者为 1、2、3、4，即 pid-3，每个消费者消费 4 个产品，即进行 4 次消费，在每次消费中：首先输出准备消费（try consume），观察缓冲区是否存在产品，即对 full 信号量进行 P 操作（sem\_wait），然后准备锁定缓冲区，输出“try lock”，对互斥信号量 mutex 进行 P 操作（sem\_wait），输出已锁定，完成后对互斥信号量 mutex 进行 V 操作（sem\_post），输出“unlock”，然后对缓冲区信号量 empty 进行 V 操作（sem\_post），最后经过一段延时后完成消费（sleep(128)），输出“consumed”。

```

void consumer(int pid, sem_t mutex, sem_t empty, sem_t full) {
    int i = pid - 3;
    for(int k = 0; k < 4; k++) {
        printf("pid %d, consumer %d, try consume, product %d\n", pid, i, k + 1);
        sem_wait(&full);
        printf("pid %d, consumer %d, try lock, product %d\n", pid, i, k + 1);
        sem_wait(&mutex);
        printf("pid %d, consumer %d, locked\n", pid, i);
        sem_post(&mutex);
        printf("pid %d, consumer %d, unlock\n", pid, i);
        sem_post(&empty);
        sleep(128);
        printf("pid %d, consumer %d, consumed, product %d\n", pid, i, k + 1);
    }
}

```

#### (8) 创建子进程完成生产者消费者函数：

更改 app/main.c 中 uEntry 函数，初始化信号量 mutex 为 1，因为缓冲区为无限制，即设定为可能存在最大产品数量  $2 \times 8 = 16$ ，于是 empty 初始化为 16，full 为 0。循环产生子进程 6 次，对于进程号 4、5、6、7 进程，为消费者进程，2、3 进程号为生产者进程。

```

int uEntry(void) {
    sem_t mutex, empty, full;
    sem_init(&mutex, 1);
    sem_init(&empty, 16);
    sem_init(&full, 0);
    pid_t pid;
    for(int i = 0; i < 6; i++) {
        pid = fork();
        if(pid == 0 || pid == -1)
            break;
    }
    if(pid != -1) {
        pid = getpid();
        if(pid > 3) {
            consumer(pid, mutex, empty, full);
        }
        else if(pid > 1) {
            producer(pid, mutex, empty, full);
        }
        exit();
    }
}

```

### 3. 实验运行

#### (1) 给 utils 文件夹下两个.pl 文件权限

```

cpl@debian:~/OSlab/lab4-171860516陈攀岭/lab$ cd utils/
cpl@debian:~/OSlab/lab4-171860516陈攀岭/lab/utils$ ls
genBoot.pl  genKernel.pl
cpl@debian:~/OSlab/lab4-171860516陈攀岭/lab/utils$ chmod 777 genBoot.pl
cpl@debian:~/OSlab/lab4-171860516陈攀岭/lab/utils$ chmod 777 genKernel.pl

```

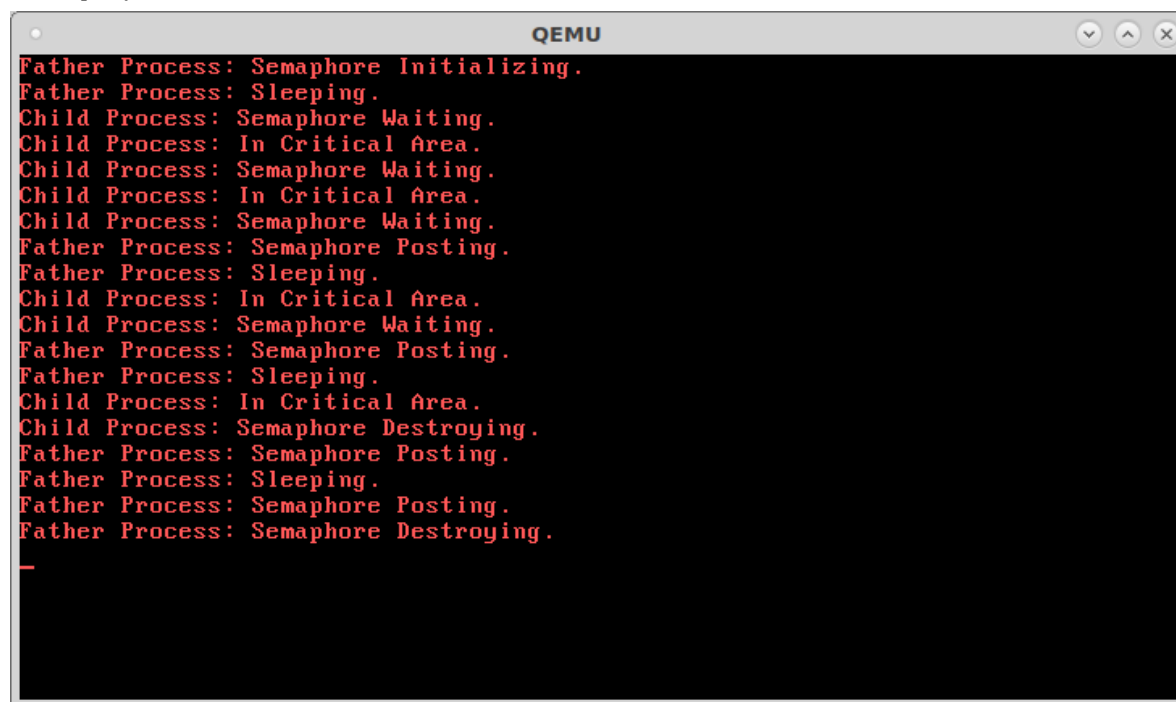
## (2) 信号量系统调用测试

```
int i = 4;
int ret = 0;
sem_t sem;
printf("Father Process: Semaphore Initializing.\n");
ret = sem_init(&sem, 2);
if (ret == -1) {
    printf("Father Process: Semaphore Initializing Failed.\n");
    exit();
}

ret = fork();
if (ret == 0) {
    while( i != 0) {
        i--;
        printf("Child Process: Semaphore Waiting.\n");
        sem_wait(&sem);
        printf("Child Process: In Critical Area.\n");
    }
    printf("Child Process: Semaphore Destroying.\n");
    sem_destroy(&sem);
    exit();
}
else if (ret != -1) {
    while( i != 0) {
        i--;
        printf("Father Process: Sleeping.\n");
        sleep(128);
        printf("Father Process: Semaphore Posting.\n");
        sem_post(&sem);
    }
    printf("Father Process: Semaphore Destroying.\n");
    sem_destroy(&sem);
    exit();
}

return 0;
```

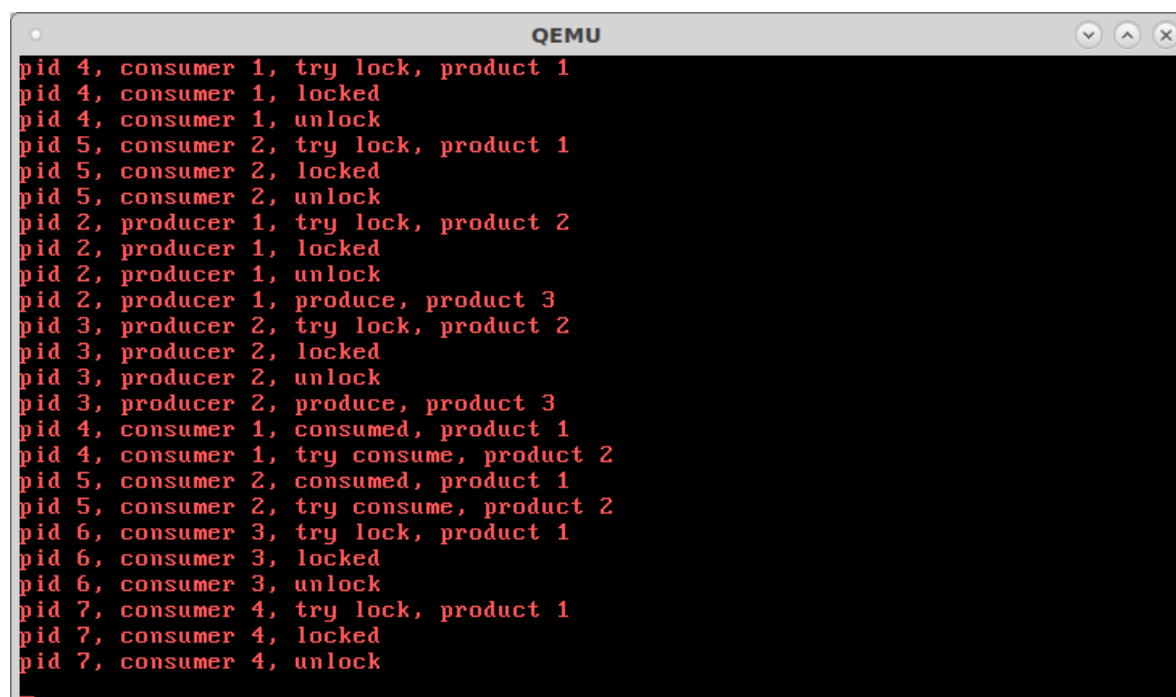
make play



```
QEMU
Father Process: Semaphore Initializing.
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.
```

### (3) 生产者消费者程序测试

make play



```
QEMU
pid 4, consumer 1, try lock, product 1
pid 4, consumer 1, locked
pid 4, consumer 1, unlock
pid 5, consumer 2, try lock, product 1
pid 5, consumer 2, locked
pid 5, consumer 2, unlock
pid 2, producer 1, try lock, product 2
pid 2, producer 1, locked
pid 2, producer 1, unlock
pid 2, producer 1, produce, product 3
pid 3, producer 2, try lock, product 2
pid 3, producer 2, locked
pid 3, producer 2, unlock
pid 3, producer 2, produce, product 3
pid 4, consumer 1, consumed, product 1
pid 4, consumer 1, try consume, product 2
pid 5, consumer 2, consumed, product 1
pid 5, consumer 2, try consume, product 2
pid 6, consumer 3, try lock, product 1
pid 6, consumer 3, locked
pid 6, consumer 3, unlock
pid 7, consumer 4, try lock, product 1
pid 7, consumer 4, locked
pid 7, consumer 4, unlock
```



```
QEMU
pid 4, consumer 1, unlock
pid 5, consumer 2, try lock, product 4
pid 5, consumer 2, locked
pid 5, consumer 2, unlock
pid 6, consumer 3, consumed, product 3
pid 6, consumer 3, try consume, product 4
pid 7, consumer 4, consumed, product 3
pid 7, consumer 4, try consume, product 4
pid 2, producer 1, try lock, product 8
pid 2, producer 1, locked
pid 2, producer 1, unlock
pid 3, producer 2, try lock, product 8
pid 3, producer 2, locked
pid 3, producer 2, unlock
pid 4, consumer 1, consumed, product 4
pid 5, consumer 2, consumed, product 4
pid 6, consumer 3, try lock, product 4
pid 6, consumer 3, locked
pid 6, consumer 3, unlock
pid 7, consumer 4, try lock, product 4
pid 7, consumer 4, locked
pid 7, consumer 4, unlock
pid 6, consumer 3, consumed, product 4
pid 7, consumer 4, consumed, product 4
```