实验四 进程同步

161130118 尹浚宇
908664035@qq.com

实验目的

掌握系统内核对于信号量机制的实现
掌握基于信号量的进程同步机制

实验内容

实现信号量相关的系统调用 sem_init, sem_post, sem_wait, sem_destory.
实现进程相关的系统调用 getpid，并修改 pcb 和 gdt 使得系统支持 8 个进程.
基于上述系统调用实现一个特定的生产者/消费者的用户程序.

程序设计思路

信号量相关的系统调用中, sem_init 已经实现好了，我们只需要实现剩下三个即可.
首先我们观察信号量结构的定义:

```
struct Semaphore {
    int state;
    int value;
    struct ListHead pcb;
};
```

```
struct ListHead {
    struct ListHead *next;
    struct ListHead *prev;
};
```

其中 state 表示使用状态(0 为未使用, 1 为使用), value 表示信号量的值, pcb 表示在该信号量上挂起的进程队列，注意该队列使用一个双向循环链表结构来维护.
下面讲述各函数的实现思路:
sem_wait:
sem_wait 系统调用对应于信号量的 P 操作，其先使得 sem 指向的信号量的值减一，之后若 value 小于 0，则当前进程阻塞自己，否则继续执行.
首先我们需要得到执行该操作的信号量是哪一个，这个值在执行 syscall 时被存放在 sf->edx 中. 在函数中通过如下语句获得:

```
int sem_index = sf->edx;
```

得到该下标后，我们便可以去系统内核中维护的信号量数组中执行相关操作. 这里要把当前进程的 pcb 加入到该信号量的挂起队列里，这里需要注意对于双向循环链表的操作. 加入完成后便可以通过 int 20 调用相关系统函数切换到其他进程. 具体代码如下图所示.

```
// P operation
sem[sem_index].value--;
if (sem[sem_index].value < 0)
{
    pcb[current].blocked.next = sem[sem_index].pcb.next;
    pcb[current].blocked.prev = &(sem[sem_index].pcb);
    sem[sem_index].pcb.next = &(pcb[current].blocked);
    (pcb[current].blocked.next)->prev = &(pcb[current].blocked);

    pcb[current].state = STATE_BLOCKED;
    pcb[current].sleepTime = -1; // blocked on sem[sem_index]
    asm volatile("int $0x20");
}
pcb[current].regs.eax = 0; // set return value
```

sem_post:

sem_post 系统调用对应于信号量的 V 操作，其先使得 sem 指向的信号量的值加 1，之后若 value 小于等于 0，则说明有进程阻塞在该信号量上，于是需要释放一个挂起队列中的进程. 获得信号量数组下标的操作不在赘述，这里需要注意的难点是我们如何得到指向待释放进程的 pcb 的指针，以及释放一个进程后对于双向循环链表的维护.

获得指向 pcb 的指针的核心思想是通过强制类型转换得到 pcb 中 ListHead 相对于 pcb 起始地址的偏移量. 具体代码如下:

```
ProcessTable *pt = NULL;
```

```
pt = (ProcessTable*)((uint32_t)(sem[sem_index].pcb.prev) -
                     (uint32_t)&(((ProcessTable*)0)->blocked));
pt->state = STATE_RUNNABLE;
pt->sleepTime = 0;
```

双向链表的维护注意细节即可，这里不再赘述.

sem_destory:

该系统调用用于销毁 sem 指向的信号量，若尚有进程阻塞在该信号量上，会带来未知错误. 这里只需要逻辑上销毁该信号量，即只需要将 state 置为 0 即可.

对于 getpid 和对于 8 个进程的支持，实现思路如下:

getpid:

由于底层汇编会把函数的返回值放在 eax 寄存器中，所以这里将 pid 放入 eax 即可.

进程数量拓展:

这里每个用户进程都要有各自的数据段和代码段，即都需要占用 gdt 中的两项，gdt 的第 0 项不用，最后一项用于存放 TSS，故 8 个进程共需要 16+2 = 18 段. 观察 kvm.c 里的 initSeg 函数，对 gdt 的赋值是通过宏操作的，且为每个用户进程分配大小为 0x100000 的空间. 因为 qemu 的寻址空间有 4G，故这里顺序分配下去即可，内存空间是足够的. 所以这里只需要修改段数量的定义即可，具体代码如下:

```
#define NR_SEGMENTS      18          // GDT size
```

对于生产者/消费者问题，实现思路如下:
首先框架代码中已经存在一个对于信号量相关调用的测试样例，这里使用条件编译加入新的测试样例. 具体如下图所示.

```
#define TEST 2

#if TEST == 1
int uEntry(void) {…
}
#endif

#if TEST == 2
void producer(pid_t pid, sem_t mutex, sem_t full)
{…
}

void consumer(pid_t pid, sem_t mutex, sem_t full)
{…
}

int uEntry(void)
{…
}
#endif
```

这里 1 号进程为主进程, 2-3 号进程作为生产者, 4-7 号进程作为消费者.
因为任意两进程间是不共享数据的，那想要使得两个进程间接共享数据就需要通过信号量来完成，只要进程操作的是同一个下标的信号量，他们就共享数据了.
这里我模拟的 buffer 是一个无界的 buffer，所以只使用两个信号量，其中 mutex 用于互斥，初始值为 1, full 用于同步，初始值为 0.
生产者伪代码如下:

```
for i from 1 to 8
    produce();
    P(mutex);
    Critical_Section();
    V(mutex);
    V(full);
```

消费者伪代码如下:

```
for i from 1 to 4
    P(full);
    P(mutex);
    Critical_Section();
    V(mutex);
    consume();
```

主进程创建子进程执行任务的框架如下:

```
// fork 6 processes
for (i = 0; i < 6; i++)
{
    ret = fork();
    if (ret == 0 || ret == -1) break;
}

if (ret == -1) // error, should not h
{…
}

else if (ret == 0) // child processes
{
    pid_t pid = getpid();
    if ((pid > 1) && (pid < 4)) // pr
        producer(pid, mutex, full);
    else // process 4-7 used as consu
        consumer(pid, mutex, full);
    exit();
}

else // father processes
{…
}
```

因为 qemu 中输出滚动太快，且无法查看之前结果，这里在 syscallWriteStdOut 函数中调用了 putChar 将程序运行结果在终端里同步显示. 部分运行结果截图如下:

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

实验心得

通过某些方法可以将 qemu 中的输出同时输出到终端，这样更便于观察与调试程序.