

Lab4 进程同步

俞星凯 171830635

2651904866@qq.com

实验目的

1. 实现一个简单的生产者消费者程序。
2. 介绍基于信号量的进程同步机制。

实验内容

1. 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`。
2. 库：对上述系统调用进行封装。
3. 用户：对上述库函数进行测试。

背景知识

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，其中 W 用于阻塞进程自身在该信号量上，R 用于释放一个阻塞在该信号量上的进程，其伪代码如下

```
struct Semaphore {
    int value;
    struct ListHead;
};
typedef struct Semaphore Semaphore;
void P(Semaphore *s) {
    s->value --;
    if (s->value < 0)
        W(s);
}
void V(Semaphore *s) {
    s->value ++;
    if (s->value <= 0)
        R(s);
}
```

实验过程

1. 启动阶段

大体与 Lab3 类似，不同之处在于加入两个数组 `sem` 和 `dev` 用来实现进程同步，两个数组的元素类型是类似的，其中，`state` 表示该元素是否有效，`value` 表示信号量的值，为正时表示在封锁前对 `s` 可施行的 P 操作，为负时其绝对值表示在 `s` 队列中等待的进程个数，`pcb` 则是一个双向链表，用于记录执行 P 操作时被阻塞的进程。

```

struct Semaphore {
    int state;
    int value;
    struct ListHead pcb;
    // link to all pcb ListHead blocked on this semaphore
};
typedef struct Semaphore Semaphore;

```

```

struct Device {
    int state;
    int value;
    struct ListHead pcb;
    // link to all pcb ListHead blocked on this device
};
typedef struct Device Device;

```

2. 系统调用

(1) GETPID 系统调用

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

`getpid` 的实现较为简单，添加宏定义 `#define SYS_PID 7`，并在 `irqhandle.c` 添加一个系统调用处理函数，利用 `eax` 返回 `pid` 即可。

```

void syscallPid(struct StackFrame *sf) {
    pcb[current].regs.eax = current;
    return;
}

```

(2) 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_init` 的实现流程是在数组 `sem` 中查找一个未使用的信号量，根据参数初始化信号量的值 `value`，并设置好双向链表 `pcb`。在框架代码中已经给出。

(3) 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。

```

void syscallSemPost(struct StackFrame *sf) {
    int i = sf->edx;
    ProcessTable *pt = NULL;
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].value++;
        if (sem[i].value <= 0) {
            pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) - (uint32_t)&(((ProcessTable*)0)->blocked));
            pt->state = STATE_RUNNABLE;
            pt->sleepTime = 0;
            sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
            (sem[i].pcb.prev)->next = &(sem[i].pcb);
        }
    }
    else
        pcb[current].regs.eax = -1;
    return;
}

```

`sem_post` 首先检查参数对应的信号量 `state` 是否为 1，若为 0 则报错，否则将 `value` 加一，若 `value <= 0`，说明有进程被阻塞，应该释放。因为 `pcb` 的 `prev` 是最先被阻塞的进程，所以将它从双向链表取出。

(4) `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。

```
void syscallSemWait(struct StackFrame *sf) {
    int i = sf->edx;
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].value--;
        if (sem[i].value < 0) {
            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);
            pcb[current].state = STATE_BLOCKED;
            pcb[current].sleepTime = -1;
            asm volatile("int $0x20");
        }
    }
    else
        pcb[current].regs.eax = -1;
    return;
}
```

`sem_wait` 同样检查 `state`，然后将 `value` 减一，若 `value < 0`，说明进程应该被阻塞，将其加入双向链表 `pcb` 的 `next`，并设置 `pcb[current]` 的 `state` 和 `sleepTime`，最后激发时钟中断重新调度。

(5) `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。

```
void syscallSemDestroy(struct StackFrame *sf) {
    int i = sf->edx;
    if (sem[i].state == 1) {
        pcb[current].regs.eax = 0;
        sem[i].state = 0;
        asm volatile("int $0x20");
    }
    else
        pcb[current].regs.eax = -1;
    return;
}
```

`sem_destroy` 直接将信号量的 `state` 清零，并触发时钟中断重新调度即可。

3. 信号量解决生产者消费者问题

(1) `main` 函数

`main` 函数需要准备两个信号量 `mutex`(用于互斥)和 `buffer`(用于生产者消费者同

步), value 分别为 1 和 0, 然后循环 fork 出 6 个子进程, 根据 fork 的返回值, 若为 0 说明是子进程, 调用相应的生产者消费者函数, 并跳出循环, 否则是父进程, 继续循环。

```
sem_init(&mutex, 1);
sem_init(&buffer, 0);
for (int i = 0; i < 6; ++i) {
    if (fork() == 0) {
        if (i < 2)
            producer(i + 2, &mutex, &buffer);
        else
            consumer(i - 1, &mutex, &buffer);
        break;
    }
}
```

(2) 生产者进程

生产者循环生产, 生产过程用 sleep(64)模拟, 生产完成打印 produce 信息, 接着尝试获取 mutex 实现互斥访问, 最后对 buffer 执行 V 操作表示 buffer 内已经有产品了。

```
void producer(int arg, sem_t* mutex, sem_t* buffer) {
    int pid = getpid();
    for (int k = 1; k <= 8; ++k) {
        sleep(64);
        printf("pid %d, producer %d, produce, product %d\n", pid, arg, k);
        printf("pid %d, producer %d, try lock, product %d\n", pid, arg, k);
        sem_wait(mutex);
        printf("pid %d, producer %d, locked\n", pid, arg);
        sem_post(mutex);
        printf("pid %d, producer %d, unlock\n", pid, arg);
        sem_post(buffer);
    }
}
```

(3) 消费者进程

消费者循环消费, 打印一条 try consume 信息, 并对 buffer 执行 P 操作, 此时若 buffer 为空则被阻塞, 接着获取 mutex 访问临界区, 最后用 sleep(64)模拟消费过程, 在完成消费后打印 consumed 信息。

```
void consumer(int arg, sem_t* mutex, sem_t* buffer) {
    int pid = getpid();
    for (int k = 1; k <= 4; ++k) {
        printf("pid %d, consumer %d, try consume, product %d\n", pid, arg, k);
        sem_wait(buffer);
        printf("pid %d, consumer %d, try lock, product %d\n", pid, arg, k);
        sem_wait(mutex);
        printf("pid %d, consumer %d, locked\n", pid, arg);
        sem_post(mutex);
        printf("pid %d, consumer %d, unlock\n", pid, arg);
        sleep(64);
        printf("pid %d, consumer %d, consumed, product %d\n", pid, arg, k);
    }
}
```

实验结果

(1) 用户程序测试

由框架代码给出的对系统调用和库函数的测试部分的实验结果如下

```

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.

```

对执行过程的分析如下

1. 父进程初始化信号量，fork 出子进程，打印 sleeping 后便去睡眠。
2. 由于信号量的值为 2，子进程可以进入两次关键区，在第三次时被阻塞。
3. 父进程苏醒，释放信号量，又去睡眠
4. 被阻塞的子进程释放，再次进入关键区，又被阻塞。
5. 同 4。
6. 被阻塞的子进程释放，销毁信号量，退出。
7. 只有父进程运行，最终销毁信号量。

(2) 生产者消费者

生产者消费者进程的执行结果较长，截取部分

```

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

```

对执行结果分析如下

1. 生产者 1、2 均执行 `sleep` 模拟生产过程，消费者 1、2、3、4 执行 `try consume` 但由于 `buffer` 为空被阻塞。
2. 生产者 1、2 各完成生产一个产品。
3. 消费者 1、2 各开始消费一个产品。
4. 生产者 1、2 各完成生产一个产品。
5. 消费者 1、2 各完成消费第一个产品，尝试消费第二个产品被阻塞。
6. 消费者 3、4 各开始消费一个产品。

实验收获

1. 了解了基于信号量的进程同步机制。
2. 提高了编写进程同步代码的能力。