

## 实验3 进程控制 2-信号量

### 一、 信号量（信号灯）

#### 1. 信号量概述

在多任务操作系统环境下，多个进程会同时运行，并且一些进程之间可能存在一定的关联。多个进程可能为了完成同一个任务会**相互协作**，这样形成进程之间的**同步关系**。在不同进程之间，为了争夺有限的系统资源（硬件或软件资源）会进入**竞争状态**，这就是进程之间的**互斥关系**。

进程之间的互斥与同步关系存在的根源在于**临界资源**。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器以及其他外围设备等）和软件资源（共享代码段，共享结构和变量等）。访问临界资源的代码叫做**临界区**，临界区本身也会成为临界资源。

信号量也称之为信号灯，是用来解决进程之间的同步与互斥问题的一种进程之间通信机制，包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作——**PV操作**（P为荷兰语的“通过”，V为荷兰语的“释放”）。其中**信号量**对应于某一种资源。**信号量值**指的是当前可用的该资源的数量，若它等于0则意味着目前没有可用的资源；当它的值大于0时，表示当前可用资源的数量；当它的值小于0时，其绝对值表示等待使用该资源的进程个数。

PV操作的具体定义如下：

- R表示资源，S表示该资源的信号量值，即可用资源的数量；
- 执行一次P操作意味着请求分配一个单位资源，因此信号量值S减1；当S=0时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去；
- 而执行一个V操作意味着释放一个单位资源，因此信号量值S加1。

1) 可以这样理解PV操作：

临界区（烧烤店）门前有门卫（服务员）发放通行证（资源R，其数量为S）；

进程（吃货）先从门卫（服务员）那里领取一张通行证才有资格进CPU（烧烤店）的门（ $S=S-1$ ）；

如果通行证用完了（ $S \leq 0$ ），门卫（服务员）说欠你一张通行证（ $S < 0$ ），您老人家排队等待吧；

进程（吃货）没辙只好在门外边排队等待；

得到通行证的进程（吃货）继续运行（吃），运行（吃）完了要出门，马上还回一张通行证（ $S=S+1$ ）

若有进程（吃货）在等待（ $S \leq 0$ ），就唤醒一个进程（吃货），把通行证交给该进程（吃货），进程可以进CPU（烧烤店）的门开始运行（吃）。

2) P操作顺序执行下述两个动作：

- ① 信号量的值减1，即 $S=S-1$ ；
- ② 如果 $S \geq 0$ ，则该进程开始使用临界资源；如果 $S < 0$ ，则把该进程的状态置为阻塞态，把相应的PCB（进程管理块）连入该信号量队列的末尾，并放弃处理机，进行等待（直至其它进程在S上执行V操作，把它释放出来为止）。

3) V操作顺序执行下述两个动作：

- ① S值加1，即 $S=S+1$ ；
- ② 如果 $S > 0$ ，则该进程释放临界资源后该干嘛干嘛；如果 $S \leq 0$ ，则释放信号量

队列上的第一个PCB（即信号量指向指针项所指向的PCB）所对应的进程（把阻塞态改为就绪态），执行V操作的进程继续运行。

最简单的信号量是只取0和1两种值，这种信号量被叫做二维信号量。这里主要讨论二维信号量，二维信号量的应用比较容易扩展到使用多维信号量的情况。

2. 信号量的应用

1) 函数说明

在 Linux 系统中，使用信号量通常分为以下几个步骤。

- ① 创建**信号量集**或获得在系统已存在的**信号量集**，此时需要调用semget()函数。不同进程通过使用同一个**信号量键值**来获得同一个信号量集。
- ② 初始化信号量，此时使用semctl()函数，设置指定信号量集中指定信号量的初值。
- ③ 设置信号量的PV操作，此时调用semop()函数，设置指定信号量集中指定信号量的PV操作。这一步是实现进程之间的同步和互斥的核心工作部分。注意，除了初始化，信号量值只能通过PV操作改变。
- ④ 如果不需要信号量，则从系统中删除它，使用semctl()函数的 IPC\_RMID 操作。此时需要注意，在程序中不应该出现对已经被删除的信号量的操作。

2) 函数格式

表1列举了semget()函数的语法要点。

表1 semget () 函数语法要点	
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semget(key_t key, int nsems, int semflg)
函数传入值	key: 信号量集的键值，可通过ftok()函数得到
	nsems: 需要创建的信号量数目，通常取值为1
	semflg: 同open()函数的权限位，如使用IPC_CREAT 标志创建新的信号量集，即使该信号量集已经存在（具有同一个键值的信号量集已在系统中存在），也不会出错。如果同时使用IPC_EXCL 标志可以创建一个新的唯一的信号量集，此时如果该信号量集已经存在，该函数会返回出错等
返回值	成功: 信号量集IPC标识符（作用类似于文件标识符，不同在于文件标识符只对当前进程有效，而信号量IPC标识符对所有进程有效，其中IPC为“进程间通信”），在信号量的其他函数中都会使用该值
	出错: -1（errno被设置为相应的值）

【注意】所谓的键值，实质上就是在指定的路径下一个指定的ID号，只不过semget函数不能直接使用这个路径和ID号，所以我们必须通过ftok函数把这个路径和ID号生成一个32位的二进制数，即键值，才能被semget函数使用。另外，类似于我们在系统中创建一个文件，进程用open函数打开文件时，会产生文件标识符，进程用semget函数打开键值时，也会产生相应的信号量集IPC标识符。

下面例子简单创建了一个简单的键值，打印输出可以看到，键值长度为4字节。

```
/* key.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int main()
```

```
{

    int x;
    x=ftok("/",1);    /* 在根目录下创建一个ID为1的键值 */
    printf("x = %p\n",x);

}
```

输出后可以通过“`ipcs -s`”命令查看是否生成了信号量，`ipcs`是用于报告进程间通信机制状态的命令。它可以查看共享内存（-m）、消息队列（-q）、信号量（-s）等各种进程间通信机制的情况。

查看后看到还没有生成信号量，因为此时还没有调用`semget`，仅仅是生成了键值。下面添加代码，生成信号量。

```
/* semget.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int main()
{

    int x,y;
    x=ftok("/",1);
    printf("x = %p\n",x);
    y=semget(x,1,0666|IPC_CREAT); /* 用ftok生成的键值，创建信号量集，信号量集中只有1个信号量，读写权限为0666 */
    if(y <0 )
        perror("semget error");/* perror可以根据系统errno值自动获取当前的错误原因 */
    printf("y = %d\n",y);

}
```

再次运行“`ipcs -s`”目录，可以看到生成了信号量，信号量的key值就是`ftok`生成的键值，信号量的`semid`值就是`semget`生成的信号量集IPC标识符。

```
[root@localhost between_process3]# ./a.out
x = 0x2010002
y = 131075
[root@localhost between_process3]# ipcs -s

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0x00000000   0          root       600        1
0x00000000   32769       root       600        1
0x01010002   98306       root       666        1
0x02010002   131075      root       666        1
```

表2列举了`semctl()`函数的语法要点。

表2 semctl() 函数语法要点	
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>

函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)
函数传入值	semid: semget()函数返回的信号量集IPC标识符
	semnum: 操作信号量在信号量集中的编号, 第一个信号量的编号为0, 如果只有一个信号量, 该值就取0
	cmd: 指定对信号量的各种操作, 常用的有以下几种: IPC_STAT: 获得该信号量 (或者信号量集合) 的semid_ds结构, 并存放在由第4 个参数arg的buf指向的semid_ds结构中。semid_ds是在系统中描述信号量的数据结构, 在/usr/include/linux/shm.h中定义; IPC_RMID: 从系统中删除信号量集; SETVAL: 将信号量集中某一信号量值设置为arg的val值; SETALL: 设置信号量集中所有信号量的值 GETVAL: 返回信号量集中一个信号量的当前值 GETALL: 读取信号量集中所有信号量的值
	arg: 是union semun结构, 该结构可能在某些系统中并不给出定义, 此时必须由程序员自己定义 <pre>union semun {     int val;          //信号量的初值, 二维信号量可设置为0或1, 当cmd为SETVAL时使用     struct semid_ds *buf;    //当cmd为STAT时使用     unsigned short *array;   //当cmd为GETALL或SETALL时使用 }</pre>
返回值	成功: 根据cmd值的不同而返回不同的值 IPC_STAT、IPC_SETVAL、IPC_RMID: 返回0 IPC_GETVAL: 返回信号量的当前值
	出错: -1 (errno被设置为相应的值)

表3列举了semop()函数的语法要点。

表3 semop() 函数语法要点	
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop(int semid, struct sembuf *sops, size_t nsops)
函数传入值	semid: semget()函数返回的信号量集IPC标识符
	sops: 指向信号量操作数组, 一个数组包括以下成员: <pre>struct sembuf {     short sem_num; /* 信号量编号, 第一个信号量的编号为0, 如果只有一个信号量, 该值就取0/     short sem_op; /* 信号量操作: 取值为-1 则表示P 操作, 取值为+1 则表示V 操作*/     short sem_flg; /* 通常设置为SEM_UNDO, 这样在进程没释放信号量而退出时, 系统自动释放该进程中未释放的信号量*/ }</pre>

	<b>nsops:</b> 操作数组sops 中的操作个数，即要进行操作的信号量的个数，如果只有1个信号量，则取值为1
返回值	成功: 0
	出错: -1 (errno被设置为相应的值)

3) 使用实例

本实例说明信号量的概念以及基本用法。在实例程序中，首先创建一个子进程，接下来使用信号量来控制两个进程（父子进程）之间的执行顺序。要求子进程先运行，然后父进程再运行。此时可以将信号量初值设置为0，子进程不执行P操作直接运行，运行结束后执行V操作；父进程先执行P操作，再运行，结束后不执行V操作。

因为信号量相关的函数调用接口比较复杂，我们可以将它们封装成二维单个信号量的几个基本函数。它们分别为信号量初始化函数（或者信号量赋值函数）init\_sem()、 P操作函数sem\_p()、 V操作函数sem\_v()以及删除信号量的函数del\_sem()等，具体实现如下所示：

```
/* sem_com.c */
/* 信号量初始化（赋值）函数*/
int init_sem(int sem_id, int init_value)
{
    union semun{
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    };
    union semun sem_union;
    sem_union.val = init_value; /* init_value 为初始值 */
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1)
    {
        perror("Initialize semaphore");
        return -1;
    }
    return 0;
}

/* 从系统中删除信号量的函数 */
int del_sem(int sem_id)
{
    union semun{
        int val;
        struct semid_ds *buf;
        unsignedshort *array;
    };
    union semun sem_union;
    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
    {
        perror("Delete semaphore");
        return -1;
    }
}
```

```

}
}

/* P 操作函数 */
int sem_p(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0; /* 单个信号量的编号应该为0，即信号量集中的第一个信号量 */
    sem_b.sem_op = -1; /* 表示P 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量*/
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("P operation");
        return -1;
    }
    return 0;
}

/* V 操作函数*/
int sem_v(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0; /* 单个信号量的编号应该为0 */
    sem_b.sem_op = 1; /* 表示V 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量*/
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("V operation");
        return -1;
    }
    return 0;
}

```

现在我们调用这些简单易用的接口，可以轻松解决控制两个进程之间的执行顺序的同步问题。实现代码如下所示：

```

/* fork.c */
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include"sem_com.c"
#define DELAY_TIME 3 /* 为了突出演示效果，等待几秒钟*/

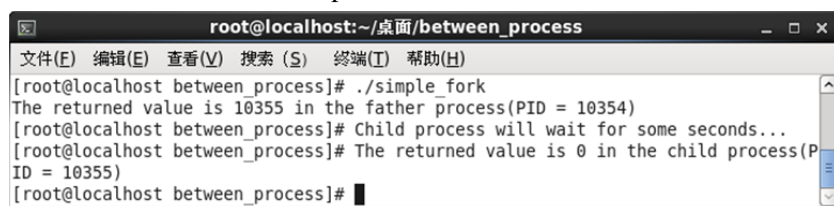
```

```

int main(void)
{
    pid_t result;
    int sem_id; /* 定义信号量集ID号*/
    sem_id = semget(ftok("/", 1), 1, 0666|IPC_CREAT); /* 创建一个信号量集*/
    init_sem(sem_id, 0); /* 初始化信号量*/
    /*调用fork()函数创建子进程*/
    result = fork();
    if(result == -1)
    {
        perror("Fork\n");
    }
    elseif(result == 0) /*返回值为0 代表子进程, 在子进程中直接运行程序, 最后调用sem_v
    释放资源*/
    {
        printf("Child process will wait for some seconds...\n");
        sleep(DELAY_TIME);
        printf("The returned value is %d, in the child process(PID = %d)\n", result,
        getpid());
        sem_v(sem_id);
    }
    else /*返回值大于0 代表父进程, 在父进程中先调用sem_p消耗资源, 因为信号量初值设置
    为0, 即没有资源, 如果子进程没有释放出资源, 则父进程sem_p操作不能进行, 所以父进程
    一直等待子进程释放出资源后才能继续sem_p(sem_id)后面的程序。*/
    {
        sem_p(sem_id);
        printf("The returned value is %d, in the father process(PID = %d)\n", result,
        getpid());
        del_sem(sem_id);
    }
    exit(0);
}

```

可以先从该程序中删除掉信号量相关的代码部分并观察运行结果,发现每次运行结果不一定相同,因为父子进程执行顺序不确定。当然可以在父进程中添加sleep语句,让父进程休眠时间比子进程长,但是用这种方法并不可靠,因为系统中还有其他进程需要占用CPU,如果其他进程占用CPU的时间超过sleep的时间,则父子进程的执行顺序就又无法确定了。



```

root@localhost:~/桌面/between_process
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost between_process]# ./simple fork
The returned value is 10355 in the father process(PID = 10354)
[root@localhost between_process]# Child process will wait for some seconds...
[root@localhost between_process]# The returned value is 0 in the child process(P
ID = 10355)
[root@localhost between_process]# █

```

```
root@localhost:~/桌面/between_process
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost between_process]# ./simple_fork
Child process will wait for some seconds...
The returned value is 10425 in the father process(PID = 10424)
[root@localhost between_process]# The returned value is 0 in the child process(PID = 10425)
[root@localhost between_process]#
```

再添加信号量的控制部分并运行结果。

```
root@localhost:~/桌面/between_process
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[root@localhost between_process]# ./a.out
Child process will wait for some seconds...
The returned value is 0 in the child process(PID = 10304)
The returned value is 10304 in the father process(PID = 10303)
[root@localhost between_process]#
```

先运行子进程，  
再运行父进程

本实例说明使用信号量怎么解决多进程之间存在的同步问题。我们将在后面讲述的共享内存和消息队列的实例中，看到使用信号量实现多进程之间的互斥。