linux信号量-应用编程-专题讲座

written by 王保明

# 1信号量基本概念

|  |
| --- |
| **信号量**   * 信号量和P、V原语由Dijkstra（迪杰斯特拉）提出 * 信号量 * 互斥：P、V在同一个进程中 * 同步：P、V在不同进程中 * 信号量值含义 * S>0：S表示可用资源的个数 * S=0：表示无可用资源，无等待进程 * S<0：|S|表示等待队列中进程个数   struct semaphore  {  int value;  pointer\_PCB queue;  }  若 value 为 1 相当于一个竞争锁 |
| **P原语**  P(s)  {  s.value = s.value--;  if (s.value < 0)  {  该进程状态置为等待状状态  将该进程的PCB插入相应的等待队列s.queue末尾  }  } |
| **V原语**  V(s)  {  s.value = s.value++;  if (s.value < =0)  {  唤醒相应等待队列s.queue中等待的一个进程  改变其状态为就绪态  并将其插入就绪队列  }  } |
|  |

# 2信号量API

## 信号量集

|  |
| --- |
| struct semid\_ds {  struct ipc\_perm sem\_perm; /\* Ownership and permissions \*/  time\_t sem\_otime; /\* Last semop time \*/  time\_t sem\_ctime; /\* Last change time \*/  unsigned short sem\_nsems; /\* No. of semaphores in set \*/  }; |
| **信号量集函数**   * #include <sys/types.h> * #include <sys/ipc.h> * #include <sys/sem.h> * int semget(key\_t key, int nsems, int semflg); * int semctl(int semid, int semnum, int cmd, ...); * int semop(int semid, struct sembuf \*sops, unsigned nsops); |
| **semget函数**   * 功能：用来创建和访问一个信号量集 * 原型 * int semget(key\_t key, int nsems, int semflg); * 参数 * key: 信号集的名字 * nsems:信号集中信号量的个数 * semflg: 由九个权限标志构成，它们的用法和创建文件时使用的mode模式标志是一样的 * 返回值：成功返回一个非负整数，即该信号集的标识码；失败返回-1 |
| **shmctl函数**   * 功能：用于控制信号量集 * 原型 * int semctl(int semid, int semnum, int cmd, ...); * 参数 * semid:由semget返回的信号集标识码 * semnum:信号集中信号量的序号 * cmd:将要采取的动作（有三个可取值） * 最后一个参数根据命令不同而不同 * 返回值：成功返回0；失败返回-1 |
| **shmctl函数续** |
| **semop函数**   * 功能：用来创建和访问一个信号量集 * 原型 * int semop(int semid, struct sembuf \*sops, unsigned nsops); * 参数 * semid:是该信号量的标识码，也就是semget函数的返回值 * sops:是个指向一个结构数值的指针 * nsops:信号量的个数 * 返回值：成功返回0；失败返回-1 |
| **semop函数续**   * sembuf结构体：   struct sembuf {  short sem\_num;  short sem\_op;  short sem\_flg;  };   * sem\_num是信号量的编号。 * sem\_op是信号量一次PV操作时加减的数值，一般只会用到两个值，一个是“-1”，也就是P操作，等待信号量变得可用；另一个是“+1”，也就是我们的V操作，发出信号量已经变得可用 * sem\_flag的两个取值是IPC\_NOWAIT或SEM\_UNDO（当进程退出时，自动恢复信号量，即自动执行进程中此前的对应p的v操作） |
|  |

# 3信号量API示例及工具

|  |
| --- |
| //创建信号量  int sem\_create(key\_t key)  {  /\* 注意是 IPC\_CREAT | IPC\_EXCL不是 O\_CREAT|O\_EXCL);  int semid = 0;  semid = semget(key, 1, 0666 | O\_CREAT|O\_EXCL);  if (semid == -1)  ERR\_EXIT("semget");    return semid;  \*/      int semid = semget(key, 1, 0666 | IPC\_CREAT | IPC\_EXCL);  if (semid == -1)  ERR\_EXIT("semget");  return semid;  }  int sem\_open(key\_t key)  {  int semid = semget(key, 0, 0);  if (semid == -1)  ERR\_EXIT("semget");  return semid;  }  int sem\_setval(int semid, int val)  {  union semun su;  su.val = val;  int ret;  ret = semctl(semid, 0, SETVAL, su);  if (ret == -1)  ERR\_EXIT("sem\_setval");  return 0;  }  int sem\_getval(int semid, int val)  {  union semun su;  su.val = val;  int ret;  ret = semctl(semid, 0, GETVAL, su);  if (ret == -1)  ERR\_EXIT("sem\_setval");  return 0;  }  int sem\_d(int semid)  {  int ret;  ret = semctl(semid, 0, IPC\_RMID, 0);  if (ret == -1)  ERR\_EXIT("semctl");  return 0;  }  int sem\_p(int semid)  {  int ret = 0;    struct sembuf sp = {0, -1, 0};  ret = semop(semid, &sp, 1); //第三个参数是信号量的参数  if (ret == -1)  ERR\_EXIT("semctl");  return ret;  }  int sem\_v(int semid)  {    struct sembuf sp = {0, 1, 0};  int ret = semop(semid, &sp, 1); //第三个参数是信号量的参数  if (ret == -1)  ERR\_EXIT("semctl");  return ret;  }      int main(int argc, char \*argv[])  {  int semid;  //semid = sem\_create(0x3234);  semid = sem\_open(0x3234);  printf("semid:%d \n", semid);  sleep(5);  sem\_d(semid);    return 0;  } |

eg：

1. 客户端给服务端发报文
2. 服务端把接收的服务次数，写共享内存

共享内存中读数据和写数据

3. 服务器初始化时创建共享内存和信号量，服务器的各个子进程通过进程互斥的方式去操作共享内存。