Unix 中多程序间共享内存

Unix 中多程序间共享内存

北京网擎科技 许杨春

共享内存 (shared memory)是 Unix 下的多进程之间的通信方法,这种方法通常用于一个程序的多进程间通信，实际上多个程序间也可以通过共享内存来传递信息。本文介绍如何在 Client/Server 方式下实现多个程序间共享内存。

问题分析

多个程序之间共享内存,首先要解决的问题是怎样让各个程序能够访问同一块内存和相同的信号量。共享内存的 id 可以通过调用 shmget(key_t key, size_t size, intshmflg)函数取得；信号量的 id 可以通过调用 semget(key_t key, intnsems, intsemflg)函数取得。实际上，只要在调用这两个函数时使用相同的 key 值，各程序之间就能达到共享内存的目的。 Unix 通过调用 key_tftok(const char＊ path, int id)函数来产生 key 值,如果各程序都用同样的参数来调用此函数，自然也就得到相同的 key 值了。例子中各个程序都使用 key=ftok(" /", 0)得到相同的 key 值,再进而由 key 值得到相同的共享内存 id 和信号量 id。

第二个要解决的问题是如何控制多个程序并发访问共享内存。本文的例子模拟在 Client/Server 方式下，由一个 Server 产生数据，多个 Client 去读取数据的操作。常规的方法是设一个信号量，将访问共享内存的程序作为临界区来处理。程序进入时用 p()操作取得锁，退出时用 v()操作释放锁。但这样做有两个问题：一是这样各个程序就处于平等的地位，而实际中往往 Server 的优先级应该比 Client 更高。比如，在股票行情应用程序中,共享内存里存放行情信息， Server 负责定时更新； Client 是 CGI 程序，负责按客户要求读取共享内存中的数据，然后再反馈给客户。在这种情况下，Server 就不能等所有 Client 进程都读完了才开始写，因为这样 Client 取得的数据反而是过时的。二是各个 Client 之间由于都是读操作，所以没有必要互斥。

本文对这两个问题的解决方案是：只有 Server 进行 p()、v()操作，信号量初始值设为 0， p()操作将它加一， v()操作将它减一；Client 读共享内存之前要先等待信号量的值为 0，这样 Server 的 p()操作总是成功，而 Server 的 p()操作后，尚未进入临界区的 Client 只能等到 Server 执行 v()操作后才能读。这样 Server 比 Client 优先， Client 之间不互斥。但这样又产生另一个问题：一个 Server 开始写时，部分 Client 可能已经进入临界区，有可能出现读不完整的问题。因此，例子基于这样一个前提：Client 程序比较简单，不会被阻塞，并且能够在一个时间片内执行完读取操作。本例中处于临界区中的 Client 数目是有限的，如果 Server 等待一个时间片 (例子中是等待一分钟)后， Client 就能全部退出临界区，这个问题就能排除。很多 CGI 程序能够满足这个假设条件，如果 Client 确实不满足条件，可以生成访问共享内存的子进程，它的执行时间应该满足上述要求。

应用实例

下面给出实现多程序间共享内存的例子程序的部分代码：

1.Server 端程序

```
＃ define SEGSIZE 1024
＃ define READTIME 1
union semun {
int val;
struct semid_ds＊ buf;
ushort_t＊ array;
} ;
//生成信号量
int sem(key_t key){
union semun sem ;
```

```
int semid;
sem.val=0;
semid=semget(key,1,IPC_CREAT|0666);
if (semid ==－ 1){
printf(" create semaphore error\n" );
exit(－ 1);
}
//初始化信号量
semctl(semid,0,SETVAL,sem);
return semid;
}
//删除信号量
void d_sem(int semid){
union semun sem ;
sem.val=0;
semctl(semid,0,IPC_RMID,0);
}
int p(int semid){
struct sembuf sops={0,＋ 1,IPC_NOWAIT};
return(semop(semid,＆ sops,1));
}
int v(int semid){
struct sembuf sops={0,－ 1,IPC_NOWAIT};
return(semop(semid,＆ sops,1));
}
int main(){
key_t key;
int shmid,semid;
char＊ shm;
char msg[7]=" data " ;
char i;
struct shmid_ds buf;
key=ftok(" /" , 0);
shmid=shmget(key,SEGSIZE,IPC_CREAT|0604);
if(shmid ==－ 1){
printf(" create shared momery error\n" );
return－ 1;
}
shm=(char＊ )shmat(shmid,0,0);
if((int)shm ==－ 1){
printf(" attach shared momery error\n" );
return－ 1;
```

```
}
semid=sem(key);
for(i=0;i<=3;i＋＋ ){
sleep(1);
p(semid);
sleep(READTIME);
msg[5]=' 0'＋ i;
memcpy(shm,msg,sizeof(msg));
sleep(58);
v(semid);
}
shmdt(shm);
shmctl(shmid,IPC_RMID,＆ buf);
d_sem(semid);
return 0;
}
2.Client 端程序＃ define SEGSIZE 1024
union semun {
int val;
struct semid_ds＊ buf;
ushort_t＊ array;
} ;
//打印程序执行时间
void secondpass(){
static long start=0;
time_t timer;
if (start == 0){
timer=time(NULL);
start=(long)timer;
printf(" now start \n" );
}
printf(" second:％ ld \n" ,(long)(time(NULL))－start);
}
int sem(key_t key){
union semun sem ;
int semid;
sem.val=0;
semid=semget(key,0,0);
if (semid ==－ 1){
printf(" get semaphore error\n");
exit(－ 1);
}
```

```
return semid;
}
//等待信号量变成 0
void waitv(int semid){
struct sembuf sops={0,0,0};
semop(semid,＆ sops,1);
}
int main(){
key_t key;
int shmid,semid;
char＊ shm;
char msg[100];
int i;
key=ftok(" /" , 0);
shmid=shmget(key,SEGSIZE,0);
if(shmid ==－ 1){
printf(" get shared momery error\n" );
return－ 1;
}
shm=(char＊ )shmat(shmid,0,0);
if((int)shm ==－ 1){
printf(" attach shared momery error\n" );
return－ 1;
}
semid=sem(key);
for(i=0;i< 3;i＋＋ ){
sleep(2);
waitv(semid);
printf(" the msg get is \n％ s\n" ,shm＋ 1);
secondpass();
}
shmdt(shm);
return 0;
}
```