

Kylix Shared Memory

by Jeremy Blythe

How to use Kylix to create and control semaphore synchronised shared memory.

The Kylix article in the last edition showed how named pipes can be used for communication between separate processes. While this is a very good way to communicate, there are more efficient ways to pass data between processes. In this article, we see how to use shared memory for Inter Process Communication.

When using pipes the communication is through a stream of data. One application puts data in the pipe and another takes it out. With shared memory, the data is stationary and any application using the memory can change it.

Shared memory is a very quick communication method. Each application involved accesses it like a normal variable. As soon as a value is changed in that address range, it is available to other processes. As its name suggests, it is a great way for multiple processes to share a block of information.

Sharing

The following LibC functions are used to create, control and destroy the shared memory.

```
function shmctl(__shmid: Integer; __cmd: Integer;
__buf: PSharedMemIdDescriptor): Integer;
function shmget(__key: key_t; __size: size_t;
__shmflg: Integer): Integer;
function shmat(__shmid: Integer; __shaddr: Pointer;
__shmflg: Integer): Pointer;
function shmdt(__shaddr: Pointer): Integer;
```

When setting up a shared memory system, one application creates the memory and other processes attach that memory into their address space. The shmget function is used to create the memory segment. It uses an integer key to identify it; we'll be using 1234 in the example. All the processes sharing the memory require this key. Shmget returns an ID that is used in the other shared memory functions. Following is a snippet of code that sets up the shared memory. We call shmget to create the memory segment of the specified size. Shmat is then used to attach the memory to the address space of our process. Shmat returns a pointer to the first byte of shared memory. We then assign the shared memory to our record.

```
const
  MEM_SZ    = 4096;
  BUFFER_SZ = 255;

type
  TSharedRec = record
    SomeText : array[0..BUFFER_SZ] of char;
  end;

var
  shared_mem : pointer;
  shared_stuff : ^TSharedRec;
  shmid : integer;

function TfrmMain.InitSM : boolean;
begin
  shmid := shmget(1234, MEM_SZ, 0666 or
                 IPC_CREAT);

  if shmid = -1 then
    result := false
  else
```

```
begin
  shared_mem := shmat(shmid, nil, 0);
  if integer(shared_mem) = -1 then
    result := false
  else
    begin
      shared_stuff := shared_mem;
      result := true;
    end;
  end;
end;
```

We can now use this record as if it was declared like a normal variable. It's as simple as that.

```
procedure TfrmMain.WriteSM(const s : string);
begin
  strcpy(shared_stuff^.SomeText, s);
end;

function TfrmMain.ReadSM : string;
begin
  result := shared_stuff^.SomeText;
end;
```

When we've finished with the shared memory we should detach it and then delete it. To detach the memory we use shmdt and to delete it we use shmctl with the IPC_RMID command.

```
function TfrmMain.DetachSM : boolean;
begin
  result := shmdt(shared_mem) <> -1;
end;

function TfrmMain.FreeSM : boolean;
begin
  result := shmctl(shmid, IPC_RMID, nil) <> -1;
end;
```

Semaphores

The shared memory system we have just developed has a major problem. What if two processes access the memory at the same time? This may be OK if one process writes and the rest read, but if we want all processes to have read and write access then we have to develop some kind of locking system. The Linux IPC system includes semaphores that can be used in a very simple way to provide the locking we require. All we need is a way of showing that the shared memory is locked or not. This is known as a binary semaphore. When the process enters the critical section (accessing the shared memory) it can set the lock. When it has finished it releases the lock. While the lock is set all other processes have to wait. Following are the LibC functions for semaphores.

```
function semctl(__semid: Integer; __semnum: Integer;
__cmd: Integer): Integer; varargs;
function semget(__key: key_t; __nsems: Integer;
__semflg: Integer): Integer;
function semop(__semid: Integer; __sops: PSemaphoreBuffer;
__nsops: size_t): Integer;
```

Don't confuse these IPC semaphore routines with the thread based semaphores also declared in the LibC unit.

As with the shared memory, the semaphore is created by one process and the other processes use the integer key to get it. Semget is used to create the semaphore or get the id if it has already been created.

```
function TfrmMain.GetSemId : boolean;
begin
  SemId := semget(1234, 1, 0666 or IPC_CREAT);
  result := SemId <> -1;
end;
```

One of the processes must then initialise the semaphore using Semctl. This is so we can set the initial value of the semaphore to 1. We use Semctl with the SETVAL command and pass the initial value as the val member of a SemRec record. SemRec has to be defined by our program as a translation of the semun C union. The definition can be found as a comment in LibC or in the man pages.

```
type
  {SEMUN UNION DEFINITION
  int val; <= value for SETVAL
  struct semid_ds *buf; <= buffer for
  IPC_STAT & IPC_SET
  unsigned short int *array; <= array for
  GETALL & SETALL
  struct seminfo *__buf; <= buffer for
  IPC_INFO}

  SemRec = record
    val : integer;
    buf : PSemaphoreIdDescriptor;
    arr : ^byte;
    __buf : PSemaphoreInfo;
  end;

function TfrmMain.InitSem : boolean;
var
  sr : SemRec;
begin
  sr.val := 1;
  result := semctl(SemId, 0, SETVAL, sr) <> -1;
end;
```

Now we need to define the EnterCriticalSection and LeaveCriticalSection functions. When we enter the critical section, we want to decrement the value of the semaphore to mark it as unavailable. When we leave the critical section, we mark it as available by incrementing the semaphore value. The semop function is used to change the value of the semaphore. We pass it a record that has three members. The first member, sem_num, is the semaphore number. In this case, it's 0 since we are only using one semaphore. Sem_op is the value that we want to change the semaphore by, -1 to decrement and +1 to increment. We set the final member, sem_flg, to SEM_UNDO this lets the operating system take care of releasing the semaphore if it was held by a process that terminated without releasing it.

```
function TfrmMain.EnterCriticalSection :
boolean;
var
  sb : TSemaphoreBuffer;
begin
  sb.sem_num := 0;
  sb.sem_op := -1;
  sb.sem_flg := SEM_UNDO;
  result := semop(SemId, @sb, 1) <> -1;
end;

function TfrmMain.LeaveCriticalSection :
boolean;
var
  sb : TSemaphoreBuffer;
begin
  sb.sem_num := 0;
  sb.sem_op := 1;
  sb.sem_flg := SEM_UNDO;
  result := semop(SemId, @sb, 1) <> -1;
end;
```

When we've finished with the semaphore we delete it using semctl with the IPC_RMID command.

```
function TfrmMain.DeleteSem : boolean;
var
  sr : SemRec;
begin
  result := semctl(SemId, 0, IPC_RMID, sr) <> -1;
end;
```

Synchronising

We can take the functions for shared memory and semaphores and bring them together to create a synchronised communication system. The critical section routines can be used to guard any access to the shared memory. Whether just writes, or reads and writes should be guarded is dependent on the application. It might be vital that processes can never read partly written data, in which case reads need to be in critical sections as well. However, the data could be arranged such that reading or some write operations don't need to go in a critical section. So, when using this communication system, we have to weigh up safety against speed.

Following is an example of a critical section guarding a write operation. A sleep command is included to artificially extend the lock time. This is purely to demonstrate the semaphores in action. If we run this procedure on two processes we'll find that one will be forced to wait for the other to release the lock before it continues.

```
procedure TfrmMain.btnWriteClick(Sender:
TObject);
begin
  if EnterCriticalSection then
    try
      WriteSM(mData.Text);
      Sleep(10000);
    finally
      LeaveCriticalSection;
    end;
end;
```

On the command line

We can use the ipcs command to show all the shared memory segments and semaphores which are currently in use. While running the example program the command shows the following output. (The 1234 key is given in hex)

```
----- Shared Memory Segments -----
key      shmid  owner  perms  bytes  nattch  status
0x000004d2  851970  root   232    4096    2

----- Semaphore Arrays -----
key      semid   owner  perms  nsems  status
0x000004d2  0       root   232    1
```

Comprehensive information on shared memory and semaphore functions is available in the man pages.

Jeremy Blythe has been programming in Delphi since version 1 and Turbo Pascal before that. His Unix skills are from University, and a lot of late nights with Linux! Jeremy is currently a senior software developer for Pharos Communications in Reading. His email address is: jeremy@pharos-comms.tele2.co.uk if you wish to contact him.