Interprocess Communication: OS/2 2.1 and UnixWare

Multi-tasking OSes like Unix and OS/2 bring interprocess communication, IPC, explicitly into the realm of applications programming. As a Unix or OS/2 developer you can design a single system with dozens of independent processes (or threads) if that's what you want. However, in order to make a coherent system, these processes usually have to pass information back and forth, and that's where IPC comes in.

As an applications programmer looking at the APIs of the two systems, you would be hard-pressed to see any similarities. However, it turns out that the models of IPC available are very similar, though the implementations differ greatly. Here, we'll look at IPC under OS/2 2.1 and UnixWare 1.1, and see what common ground we can find.

When you're designing a multi-process (or thread) system, there are a number of IPC issues you need to think about. The first one is, how are the processes going to identify the shared resource? Since related processes (child processes and threads) share a common set of open file handles with their parent, they don't have to worry about resource naming. The parent can create any number of anonymous resources (pipes, shared memory ...) and both parent and child can refer to them simply by handle.

Unrelated processes don't enjoy the luxury of sharing open handles, and thus, need to have another way to identify shared resources. Under OS/2, resources have names that must be unique, fully qualified filenames under OS/2's rules for file naming. Each resource type uses a common base filename. For semaphores this is "\SEM32", queues "\QUEUES", shared memory "\SHAREMEM", and pipes "\PIPE".

Under Unix, queues, semaphores and shared memory share a single naming scheme. These resources have an integer id that is typically created by a call to **ftok** which uses an agreed-upon filename and a char seed to create a consistent integer id. Listing 1 shows a typical use of **ftok** to create a resource ID.

With the resource named, the processes then have to agree on who owns the resource, that is, who's responsible for creating and destroying it. System-wide resources, resources that unrelated processes can access, generally have to be explicitly destroyed. That means that if your process gets disappeared without destroying the resource, the next time it comes up, the resource will already exist and the create call will fail. Take it from me, processes disappear mysteriously a lot more often than anyone is willing to admit. Local resources, like anonymous pipes, live and die with the process that created them, but in general, you have to put some thought into what you need to do if a process fails to properly dispose of an unneeded resource.

Another issue to pay attention to in any IPC situation is atomicity. I write atomic code all the time; blows up spectacularly. In this case, though, atomicity refers to whether or not a call can be task switched. For our purposes, a call is said to be atomic if you are guaranteed not to be task-switched in the middle of it. Consider the case of a bunch of OS/2 processes trying to gain ownership of a resource protected by a single, mutual exclusion semaphore. Each process makes a call to **DosRequestMutexSem** and blocks waiting for the semaphore to clear. The current owner clears the semaphore. Only one of the waiting processes can now become the new owner. If **DosRequestMutexSem** were not atomic, you could get two or more processes waking up and thinking they owned the semaphore.

The last, and perhaps most important consideration is what mode you're going to run the IPC channel in, blocking or non-blocking. In blocking mode, your process sleeps until the call (Read or Write) can be satisfied. In non-blocking mode, the call returns an error if it can't be satisfied. If, like the window procedure in a Windows program, a process is solely concerned with processing the message coming in over the IPC channel, it's simple, safe and very effective to use blocking IO. Most real-world programs, though, can't afford to block waiting for IO. Usually, you'll have to split a program into multiple related processes in order to use blocking IO. A common OS/2 technique is to spawn a thread for each blocking IO channel, and have each thread use non-blocking IO to get the information back to the main thread.

Shared Memory and Semaphores

IPC divides into roughly four types: shared memory, semaphores, pipes and queues. Shared memory is the simplest, most intuitive type of IPC. With shared memory, multiple processes have access to a single block of memory. Shared memory is particularly effective where the size of the data being communicated is small and fixed. The difference between shared memory and the message-style IPC APIs (pipes and queues) is the difference between a whiteboard and a pad of post-it notes.

- You can only fit so much on the whiteboard, while post-its go on forever.

- The information on the whiteboard also has no particular order, while the post-its are always stuck to your terminal in either chronological or priority order.

Consider the case of an uninterruptible power supply. A typical smart UPS keeps track of a couple of dozen variables, like line voltage and load, which it transmits to the system every couple of seconds over the serial port. The UPS control program might consist of two processes; a daemon which reads the serial port and updates the variables in the shared memory block and a user interface which displays the variables it reads in the shared memory block. The user is interested in only the current value of any variable, so the user interface doesn't need to try to display every change of value and thus has no need to synchronize with the daemon. In this case, the shared memory block is used like the outfield scoreboard at a baseball game, simply transmitting the scores to anyone who wants to read them.

A simple example, but it presents two potentially nasty atomicity problems. The first is in initialization. The daemon process creates the shared memory block, and the user interface process attaches to it. There is a small window of time after the daemon makes the block when the values in the block will be uninitialized. In that window, the daemon could be task-switched and the user interface could attach to the block and read the bogus values. Figure 1 illustrates the problem. The second situation occurs whenever the user interface reads, or the daemon writes the block. Assume, for arguments sake that one of our UPS' values is a string. The daemon could be **strcpy**'ing our string to the shared memory block, when the OS task switches to the user interface, which then tries to use the half completed string. Not a good situation. The same thing can be said in reverse; if the user interface is halfway through **strcpy**'ing the string to video memory and the daemon overwrites the string you also get inconsistent data.

This is where semaphores come in. With our UPS example, a single semaphore could guard the block. On startup, the daemon would create the semaphore and set it. Then it would create and initialize the shared memory block. Only after initialization would the daemon clear the semaphore. The user interface would attach to the semaphore and always block waiting for it to clear before reading the shared memory block. Since the daemon would always set the semaphore before beginning to write the block, the user interface is guaranteed never to read a half-complete value. Figure 2 shows how semaphore protection might work in this case.

Tables 1 and 2 show the OS/2 and UnixWare shared memory and semaphore APIs. As usual, under OS/2 there's a separate call for every step, while Unix lumps a bunch of functionality into a catch-all, **ioctl**-style, **shmctl** call. OS/2 goes even further, by dividing semaphores into two types: mutual exclusion like the one in our example above, and event such as one might use to indicate that a print job has finished. Unix also operates by default on a set of semaphores, making you jump through hoops to implement a single semaphore, while OS/2 assumes a single semaphore and makes you use different calls to operate on sets. I tend to agree with R.W. Stevens who calls the SVR4 semaphore implementation unnecessarily complicated. Stevens goes so far as to suggest using file record locking for simple semaphores. Listings 2 and 3 show OS/2 and UnixWare implementations of a semaphore-protected, shared memory initialization sequence.

Anonymous Pipes

But everyone knows that real comm programmers send messages, and if it's messages you're after, you have only two choices; pipes and queues. A pipe is a FIFO and it comes in two flavors, named and anonymous, with

two sizes, full and half duplex. The original pipe was a Unix invention and it came in only one flavor and size; anonymous, half-duplex. If you wanted full-duplex you had to buy two half-duplexes. A call to **pipe** yielded one read and one write handle. The reading process used the read handle, the writing process the write handle. If Process A wrote a message using the write handle, then read a message using the read handle, it would get back the message it just wrote. Figure 3 shows an example of a half-duplex pipe.

OS/2 anonymous pipes are always half-duplex. For Process A to talk to Process B via an anonymous pipe, Process A creates a pipe, getting a read and a write handle, then spawns process (or thread) B which inherits both of those handles. Process A then closes the read handle, while process B closes the write handle. This leaves A with a valid write handle and B with a valid read handle. Closing the useless handle ensures that the process gets notified when the process on the other end of the pipe goes away. If B wants to talk to A, they have to open another pipe.

A full-duplex pipe is a two way channel which makes the name pipe a bit of a misnomer, because in a real-world pipe, the water always flows in one direction at a time; half-duplex. In the Unix world, half-duplex pipes were judged to be less than completely elegant, so as of SVR3.2, all pipes are full duplex (also known as **anonymous stream pipes**). A call to **pipe** gives a handle to each end of a full-duplex pipe. Process A reads from and writes to the first handle, while Process B reads from and writes to the second handle. So a pipe shared by processes A and B is more properly thought of as two pipes, one from A to B and one from B to A. Figure 4 shows an example of a full-duplex pipe.

In service, a pipe is sort of a cross between a serial port and a file. Like a file, you can generally choose how you want to read the data, binary or translated, whether you want a byte at a time or CR-LF delimited lines. Like a serial port, you can never read what you write (unless you're abusing a half-duplex pipe), and read data always comes at you in FIFO order. You can open it as a byte-stream (raw mode), or as a message stream (cooked mode). You can also end up blocking on the open, if there's no process on the other end, on the read, if there's no data available, on the write, if the pipe's full and on the close if there's still data flowing.

Named Pipes

Named pipes are exactly what the name implies: IPC channels which are referred to by name, and can thus have unrelated processes on either end. OS/2 has a single, fairly simple, named pipe API which provides full-duplex IPC channels.

On the other hand, if you plow into the UnixWare doc looking for named pipes you will be sorely disappointed. UnixWare actually has two forms of named-pipe; **FIFO**s and **named stream pipes**. **FIFO**s are an interesting case. They are actually **FIFO**s, in that data always gets read in the order it was written in. They are also files though, in that they actually exist within the file-system and can be seen with a simple directory listing. The file IO functions (open, close, read, write ...) work on them. You create a **FIFO** by calling mkfifo or using the **creat** system call with the **FIFO_MODE** bit set. **FIFO**s have one big drawback; they are only half-duplex.

If you want full-duplex named pipes, you need to use **named stream pipes**. **Named stream pipes** are even more difficult to find in the doc than **FIFO**s (try ***streams(7)*** or just read Stevens). This is mostly because they're implemented by combining two other facilities: **FIFO**s and **anonymous stream pipes**. On the server end, you create a **FIFO** and an **anonymous stream pipe**, push the streams module **connld** onto the stream, then use **fattach** to attach the **FIFO** name to the **anonymous stream pipe**. Voila, the **anonymous stream pipe** now has a name. The tricky part is that the file descriptor the server uses for reading and writing is neither the original **anonymous stream pipe** descriptor, nor one gotten from opening the **FIFO** but one received via the **ioctl** call **I_RECVFD**. The client end is easy; simply open the **FIFO** and read and write it as if it were an **anonymous stream pipe**. If it all sounds a little convoluted, well, it's not that bad. Listing 4 shows a complete program for a client and server exchanging messages over a **named stream pipe**. Tables 3 and 4 list the calls in the OS/2 and UnixWare pipe APIs.

If you need to send a byte stream, pipes, named or anonymous, are the only IPC choice you have. This is why the classic pipe examples always involve dup'ing standard input and ouput. In fact, a good application for named

pipes is in a multi-user game, where the users all sit there bashing the arrow keys to move their game pieces around and shoot at each other. There's a single, global game process and then a user-interface process for each user. Each user-interface opens a pipe to the game process. The user-interface sends the user's keystrokes to the game process over the pipe, and receives all the other user's moves from the game process over the same pipe. Someone at Microsoft actually wrote a great multi-user Tank game for LAN Manager that used networked, named-pipes in this fashion.

The Other End

Unlike all the other forms of IPC, pipes are true process-to-process connections, so they pay close attention to the status of the processes on both ends of the conversation. Depending on what the processes are doing, something happening on one end of the pipe can result in a signal (SIG_PIPE), an error return, or blocking on the other end.

Queues

Now that Windows has conquered the world, message queues are a painfully familiar topic. The Windows message queue is actually a very good example of when to use a queue.

- All the events have to be received (forget about mouse moves for now). Wouldn't do to miss any mouse clicks. In our UPS example, all the data value changes didn't have to be processed by the user interface.

- The order of events must be maintained. When you double-click on an icon, the double-click message has to come after the mouse-move that took the mouse to the icon. In our UPS example, the user doesn't care if the value for line voltage was received before the value for load.

- The event can be described with a small amount of data of fixed size. Most Windows messages try to put the event data into the fixed-size message. When the data won't fit, Windows uses a pointer.

There is an important difference between OS/2 and Unix queues. Unix queues can hold a configurable maximum total number of bytes in the queue, so you can copy variable sized messages directly into the queue, while OS/2 queues operate strictly on 32 bit data values. Big deal, you say; just use pointers. Pointers to what, though? Say Processes A and B are unrelated and trying to communicate 10 byte messages via a queue. A passes B a pointer to a 10 byte bit of memory. Unless that pointer points to shared memory, B will GP fault trying to read it. Listing 5 shows an example of an OS/2 program that uses queues and shared memory to pass > 32 byte messages.

OS/2 queues allow you to choose either FIFO, LIFO or priority ordering of elements. You set the read order when you create the queue and it can't be changed. Unix approaches read ordering a little differently. You create the queue without specifying a read order. You specify (and can change) the read order with each call to read. You can read either the first message, or the first message that has a particular integer id attached to it. By using the target process's pid as the integer id, you can then use the queue as a two way, full-duplex FIFO channel. Tables 5 and 6 show the Unix and OS/2 queue APIs.

Multiplexing

In many cases, you'll want to set up a single process that serves a number of blocking IPC channels. UnixWare includes the ability to operate on multiple IO handles (not just pipes) via the **poll** and **select** calls. Under OS/2, you have to attach a semaphore to each pipe or queue, then create a multiplexed semaphore containing all the individual semaphores. Listing 5 uses OS/2 multiplexed semaphores to implement a single process servicing multiple queues.

Permissions

Unix strives to preserve a consistent approach to all system resources, whether they be files, devices, or IPC channels. Thus, with all the four forms of UnixWare IPC, you have to pay careful attention to the read/write permissions (execute mostly doesn't apply) for owner, group and public the same way you do for files. Along with permissions you also have to be aware of the effective user id of the processes. In OS/2, permissions are generally binary: other processes can access the resource or they can't, with no gray areas.

Signals

I've ignored signals to this point. Under UnixWare, they provide a significantly different IPC model whereby you can register a function to be called when an event occurs, then "send" that event from within an unrelated process. They suffer from serious restrictions, however. No information is passed with the signal, and the number of user-defined signals is severely limited. As well, if a process is in a blocking system call when a signal arrives, the system call will generally return an error which must be explicitly ignored. OS/2 uses the signaling model for handling exceptions like CTL-C, but does not provide for user-defined signals.

Other IPC

Under UnixWare and OS/2, several other forms of IPC are also available. These include things like, OLE, and DDE (which is built on shared memory anyway) and the messaging APIs of the Motif and Presentation Manager GUIs. Both also support network socket APIs and UnixWare, of course, comes bundled with the NetWare networking APIs.

Conclusion

Like most everything else, IPC is not portable between OS/2 and UnixWare. However, though the implementation details differ greatly, the two systems do share certain ways of thinking about IPC. They try to cover the same functionality, and almost any style of IPC you implement in one can usually be replicated in the other.

John Rodley is a writer and independent consultant.

Bibliography

R.W. Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley

Robert Orfali and Dan Harkey, Client/Server Programming with OS/2 2.0, Van Nostrand Reinhold.

OS/2 2.0 Technical Library, Programming Guide Volume 1, IBM

OS/2 Shared Memory API	Notes
DosAllocSharedMem	Create shared memory.
DosFreeMem	Destroy shared memory.
DosGetNamedSharedMem	Attach to named shared memory.
DosGetSharedMem	Attach to unnamed shared memory.
DosGiveSharedMem	Give access to unnamed shared memory.

OS/2 Semaphore API	Notes
DosCloseEventSem	Destroy event semaphore.
DosCreateEventSem	Create event semaphore.
DosOpenEventSem	Attach to event semaphore.
DosPostEventSem	Trigger an event.
DosQueryEventSem	Query event sem status.
DosResetEventSem	Clear an event from semaphore.
DosWaitEventSem	Block waiting for event trigger.
DosCloseMutexSem	Destroy a mutual exclusion semaphore.
DosCreateMutexSem	Create mutex semaphore.
DosOpenMutexSem	Attach to mutex semaphore.
DosQueryMutexSem	Check status of mutex semaphore.
DosReleaseMutexSem	Clear mutex semaphore.
DosRequestMutexSem	Wait for semaphore to clear, then set it.
DosAddMuxWaitSem	Add a semaphore to semaphore set on the fly.

DosCloseMuxWaitSem	Destroy the semaphore set.
DosCreateMuxWaitSem	Create semaphore set.
DosDeleteMuxWaitSem	Delete a semaphore from semaphore set on the fly.
DosOpenMuxWaitSem	Attach to existing semaphore set.
DosQueryMuxWaitSem	Check semaphore's status.

Table 1: *OS/2 shared memory and semaphore API*

UnixWare Shared Memory API	**Notes**
shmget	Create shared memory.
shmctl	Destroy shared memory.
shmat	Attach to existing shared memory.
shmdt	Detach from shared memory.
UnixWare Semaphore API	**Notes**
semget	Create a semaphore set.
semctl	Destroy a semaphore set, get and set semaphore values.

Table 2: *UnixWare shared memory and semaphore API*

OS/2 Pipe API	**Notes**
DosCallNPipe	Does a transact and a close. See DosTransact and DosClose.
DosConnectNPipe	Put server end of pipe into listen mode. Blocks until client end calls DosOpen.
DosCreateNPipe	Make a named pipe.
DosDisConnectNPipe	Server acknowledges client calling DosClose.
DosPeekNPipe	See if anything's in the pipe.
DosQueryNPipeHState	See if pipe's listening, connected, closed ...

DosQueryNPipeInfo	Check pipe parameters.
DosQueryNPipeSemState	Check on the semaphore attached to this pipe.
DosSetNPHState	Change blocking and read modes.
DosSetNPipeSem	Attach a semaphore to a pipe.
DosTransactNPipe	Write a message and read a response.
DosWaitNPipe	Wait for named pipe to be created.
DosClose	Close the pipe.
DosCreatePipe	Create anonymous pipe.
DosOpen	Open a pipe.
DosRead	Read a pipe.
DosWrite	Write a pipe.

Table 3: *OS/2 pipe API.*

UnixWare Pipe API	**Notes**
pipe	Create anonymous stream pipe.
popen	Exec a process and create a new anonymous stream pipe and attach it to process's stdin and stdout.
pclose	Close a pipe.
ioctl	Used to push connld onto named pipe stream and to receive named pipe file descriptor. See I_PUSH and I_RCVFD commands.
fattach	Attach file system name to stream.
creat	Create a FIFO (set FIFO_MODE).
mkfifo	Create a FIFO.

Table 4: *UnixWare pipe API.*

OS/2 Queue API	**Notes**
DosCloseQueue	Destroy a queue.

DosCreateQueue	Create a queue.
DosOpenQueue	Attach to an existing queue.
DosPeekQueue	See what's in the queue.
DosPurgeQueue	Clean out the queue.
DosReadQueue	Take a message off the queue.
DosWriteQueue	Put a message into the queue.

Table 5: *OS/2 queue API.*

UnixWare Queue API	**Notes**
msgget	Create a new queue or attach to an existing queue.
msgctl	Destroy the queue.
msgsnd	Send a message.
msgrcv	Take a message off the queue.

Table 6: *UnixWare queue API.*

Time slice	Daemon Process	User Interface Process
1		Loop waiting for shared memory to appear.
2	Create uninitialized shared memory.	
3		Read shared memory.
4	Initialize shared memory.	
5		Display bogus values from uninitialized shared memory.

Figure 1: *Unprotected shared memory initialization.*

Time slice	Daemon Process	User Interface Process
1		Loop waiting for semaphore to appear.
2	Create and set semaphore	
3		Block waiting for semaphore to clear.
4	Create uninitialized shared memory.	
5		Block ...
6	Initialize shared memory.	
7		Block ...
8	Clear semaphore	
9		Unblock now that semaphore is clear read shared memory block, and display initialized values.

Figure 2 : *Shared memory initialization protected by a semaphore.*

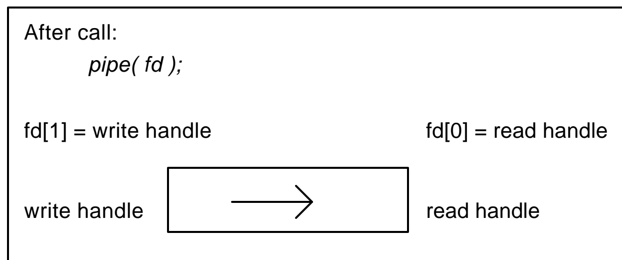

Figure 3: *A half-duplex pipe.*

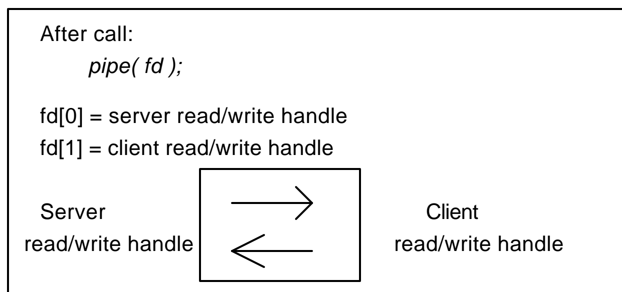

Figure 4: *A full-duplex pipe.*