Inter-Process Communication

Introduction

We can divide process interactions into a two broad categories:

1. the coordination of operations with other processes:
    - synchronization (e.g. mutexes and condition variables)
    - the exchange of signals (e.g. *kill(2)*)
    - control operations (e.g. *fork(2)*, *wait(2)*, *ptrace(2)*)
2. the exchange of data between processes:
    - uni-directional data processing pipelines
    - bi-directional interactions

The first of these are discussed in other readings and lectures. This is an introduction to the exchange of data between processes.

Simple Uni-Directional Byte Streams

These are easy to create and trivial to use. A pipe can be opened by a parent and inherited by a child, who simply reads standard input and writes standard output. Such pipelines can be used as part of a standard processing model:

```
macro-processor | compiler | assembler > output
```

or as custom constructions for one-off tasks:

```
find . -newer $TIMESTAMP | grep -v '*.o' | tar cfz archive.tgz -T -
```

All such uses have a few key characteristics in common:

- Each program accepts a byte-stream input, and produces a byte-stream output that is a well defined function of the input.
- Each program in the pipeline operates independently, and is unaware that the others exist or what they might do.
- Byte streams are inherently unstructured. If there is any structure to the data they carry (e.g. newline-delimited lines or comma-separated-values) these conventions are implemented by parsers in the affected applications.
- Ensuring that the output of one program is suitable input to the next is the responsibility of the agent who creates the pipeline.
- Similar results could be obtained by writing the output of each program into a (temporary) file, and then using that file as input to the next program.

Pipes are temporary files with a few special features, that recognize the difference between a file (whose contents are relatively static) and an inter-process data stream:

- If the reader exhausts all of the data in the pipe, but there is still an open write file descriptor, the reader does not get an *End of File*(EOF). Rather the reader is blocked until more data becomes available, or the write side is closed.
- The available buffering capacity of the pipe may be limited. If the writer gets too far ahead of the reader, the operating system may block the writer until the reader catches up. This is called *flow control*.
- Writing to a pipe that no longer has an open read file descriptor is illegal, and the writer will be sent an exception signal (SIGPIPE).

- When the read and write file descriptors are both closed, the file is automatically deleted.

Because a pipeline (in principle) represents a closed system, the only data privacy mechanisms tend to be the protections on the initial input files and final output files. There is generally no authentication or encryption of the data being passed between successive processes in the pipeline.

Named Pipes and Mailboxes

A named-pipe (*fifo(7)*) is a baby-step towards explicit connections. It can be thought of as a persistent pipe, whose reader and writer(s) can open it by name, rather than inheriting it from a *pipe(2)* system call. A normal pipe is custom-plumbed to interconnect processes started by a single user. A named pipe can be used as a rendezvous point for unrelated processes. Named pipes are almost as simple to use as ordinary pipes, but ...

- Readers and writers have no way of authenticating one-another's identities.
- Writes from multiple writers may be interspersed, with no indications of which bytes came from whom.
- They do not enable clean fail-overs from a failed reader to its successor.
- All readers and writers must be running on the same node.

Recognizing these limitations, some operating systems have crated more general inter-process communication mechanisms, often called *mailboxes*. While implementations differ, common features include:

- Data is not a byte-stream. Rather each write is stored and delivered as a distinct message.
- Each write is accompanied by authenticated identification information about its sender.
- Unprocessed messages remain in the mailbox after the death of a reader and can be retrieved by the next reader.

But mailboxes still subject to single node/single operating system restrictions, and most distributed applications are now based on general and widely standardized network protocols.

General Network Connections

Most operating systems now provide a fairly standard set of network communications APIs. The associated Linux APIs are:

- *socket(2)* ... create an inter-process communication end-point with an associated protocol and data model.
- *bind(2)* ... associate a *socket* with a local network address.
- *connect(2)* ... establish a connection to a remote network address.
- *listen(2)* ... await an incoming connection request.
- *accept(2)* ... accept an incoming connection request.
- *send(2)* ... send a message over a socket.
- *recv(2)* ... receive a message from a socket.

These APIs directly provide a range of different communications options:

- byte streams over reliable connections (e.g. TCP)
- best effort datagrams (e.g. UDP)

But they also form a foundation for higher level communication/service models. A few examples include:

- Remote Procedure Calls ... distributed request/response APIs.
- RESTful service models ... layered on top of HTTP GETs and PUTs.
- Publish/Subscribe services ... content based information flow.

Using more general networking models enables processes to interact with services all over the world, but this adds considerable complexity:

- Ensuring interoperability with software running under different operating systems on computers with different instruction set architectures.
- Dealing with the security issues associated with exchanging data and services with unknown systems over public networks.
- Discovering the addresses of (a constantly changing set of) servers.
- Detecting and recovering from (relatively common) connection and node failures.

Applications are forced to choose between a simple but strictly local model (pipes) or a general but highly complex model (network communications). But there is yet another issue: performance. Protocol stacks may be many layers deep, and data may be processed and copied many times. Network communication may have limited throughput, and high latencies.

Shared Memory

Sometimes performance is more important than generality.

- The network drivers, MPEG decoders, and video rendering in a set top box are guaranteed to be local. Making these operations more efficient can greatly reduce the required processing power, resulting in a smaller form-factor, reduced heat dissipation, and a lower product cost.
- The network drivers, protocol interpreters, write-back cache, RAID implementation, and back-end drivers in a storage array are guaranteed to be local. Significantly reducing the execution time per write operation is the difference between an industry leader and road-kill.

High performance for Inter-Process Communication means generally means:

- efficiency ... low cost (instructions, nano-seconds, watts) per byte transferred.
- throughput ... maximum number of bytes (or messages) per second that can be transferred.
- latency ... minimum delay between sender write and receiver read.

If we want ultra high performance Inter-Process Communication between two local processes, buffering the data through the operating system and/or protocol stacks is not the way to get it. The fastest and most efficient way to move data between processes is through shared memory:

- create a file for communication.
- each process maps that file into its virtual address space.
- the shared segment might be locked-down, so that it is never paged out.
- the communicating processes agree on a set of data structures (e.g. polled lock-free circular buffers) in the shared segment.
- anything written into the shared memory segment will be immediately visible to all of the processes that have it mapped in to their address spaces.

Once the shared segment has been created and mapped into the participating process' address spaces, the operating system plays no role in the subsequent data exchanges. Moving data in this way is extremely efficient and blindingly fast ... but (like all good things) this performance comes at a price:

- This can only be used between processes on the same memory bus.
- A bug in one of the processes can easily destroy the communications data structures.
- There is no authentication (beyond access control on the shared file) of which data came from which process.

Network Connections and Out-of-Band Signals

In most cases, event completions can be reported simply by sending a message (announcing the completion) to the waiter. But what if there are megabytes of queued requests, and we want to send a message to abort those queued requests? Data sent down a network connection is FIFO ... and one of the problems with FIFO scheduling is the delays waiting for the processing of earlier but longer messages. Occasionally, we would like to make it possible for an important message to go directly to front of the line.

If the recipient was local, we might consider sending a signal that could invoke a registered handler, and flush (without processing) all of the buffered data. This works because the signals travel over a different channel than the buffered data. Such communication is often called *out-of-band*, because it does not travel over the normal data path.

We can achieve a similar effect with network based services by opening multiple communications channels:

- one heavily used channel for normal requests
- another reserved for out-of-band requests

The server on the far end periodically polls the out-of-band channel before taking requests from the normal communications channel. This adds a little overhead to the processing, but makes it possible to preempt queued operations. The chosen polling interval represents a trade-off between added overhead (to check for out-of-band messages) and how long we might go (how much wasted work we might do) before noticing an out-of-band message.