Interprocess Communications

CS 239 — Advanced Operating Systems

April 18, 1995

Outline

- IPC Fundamentals
- Unix Sockets
- Duality of Message/Procedure Models
- Remote Procedure Call
- Shared Memory and Large Memory Spaces

IPC Fundamentals

What is IPC?

Mechanisms to transfer data between processes

Why is it needed?

Intraprocess methods don’t work

Why Do Processes Communicate?

- To share resources
- Client/server paradigms
- Inherently distributed applications
- Reusable software components
- Other good software engineering reasons

Basic IPC Concept

A sending process needs to communicate data to a receiving process

Sender wants to avoid details of receiver’s condition

Receiver wants to get data in an organized way

Desirable Characteristics of IPC

- Fast
- Easy to use
- Well defined synchronization model
- Versatile
- Easy to implement
- Works remotely

IPC and Synchronization

Synchronization is a major concern for IPC

- Allowing sender to indicate when data is “transmitted”
- Allowing receiver to know when data is ready
- Allowing both to know when more IPC is possible

Standard synchronization primitives can be used

Basic IPC Mechanisms

- File system IPC
- Message-based IPC
- Procedure call IPC
- Shared memory IPC

IPC Through the File System

“Sender” writes a file

“Receiver” reads it

But when does the receiver perform the read?

Often through lock files or file locking

Special types of files can make file-based IPC easier

File System IPC Diagram

Message-Based IPC

Sender formats data into a formal message

With some form of address for receiver

OS delivers message to receiver’s message input queue

Receiver (when ready) reads a message from the queue

Sender may or may not block

Message-Based IPC

Procedure Call IPC

Interprocess communication using procedure call interface

- Data passed as parameters
- Information returned via return values

Complicated since destination procedure in a different address space

Generally calling process blocks waiting return

Procedure Call-Based IPC

Shared Memory IPC

Different processes share a common piece of memory

- Either physically or virtually

Communications via normal reads and writes

May need semaphores or locks in or associated with the shared memory

- Otherwise non-blocking

Shared Memory-Based IPC

Typical Unix IPC Mechanisms

Different versions of Unix introduced different mechanisms

- Pipes
- Message queues
- Semaphores
- Shared memory
- Sockets
- RPC

Pipes

Only IPC mechanism (other than files) in early Unix systems

Uni-directional, unformatted, uninterpreted, interprocess byte streams

- One process feeds it bytes
- Another consumes bytes from it

Accessed in file-like way

Pipe Details

Potentially blocking communication mechanism

A given pipe has a fixed capacity

Writing more bytes than can be held blocks the sender

Reading bytes when none are available blocks the receiver

Named pipes allow more flexibility

Message Queues

Introduced in System V Release 3 Unix

Like pipes, but—

Data organized into messages

Message components include:

- Type identifier
- Length
- Data

Semaphores

Also introduced in System V Release 3

Mostly synchronization only

Often used in conjunction with shared memory

Shared Memory for Unix

Also introduced in System V Release 3

Allows setting up shared memory segments

Some control over read/write permissions

Often used to implement threads packages for Unix

Sockets

Introduced in 4.3 BSD

A *socket* is an IPC channel with generated endpoints

Great flexibility in its characteristics

Intended as building block for communications

Endpoints established by the source and destination processes

More details shortly

Remote Procedure Calls

Procedure calls from one address space to another

On the same or different machines

Requires cooperation from both processes

In Unix, often built on sockets

Establishing Sockets

Using the socket() system call

Specify domain, type, and protocol

Sockets can be connected or connectionless

Each side responsible for proper setup/access

Socket Domains

Describes protocol family

Generally related to the address family

Domain can be:

- internal protocols
- Internet protocols
- IMP link layer protocols

Socket Types

Describes semantics of communication over the socket

Several types are defined—

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_SEQPACKET
- SOCK_RAW
- SOCK_RDM

Socket Protocols

Specifies particular protocol to be used

Must match other parameters

Generally, only one protocol per socket type available

Socket Streams

Of type SOCK_STREAM

Full-duplex reliable byte streams

Like 2-way pipes

Requires other side to connect

Socket Sequential Packets

Like streams, but for fixed-sized data packets

So reads always return a fixed number of bytes

Socket Datagrams

Like sequential packets, but non-reliable delivery

send, rather than write, used to send data

Socket Options

Many possible options on sockets, e.g.

- Connection or connectionless
- Blocking or non-blocking
- Out-of-band information
- Broadcast messages
- Buffer sizes (input and output)
- Routing options

Binding Sockets

Sockets are typically bound to local names

For IPC on a single machine

Often accessed using descriptors (like file descriptors)

Requires clean-up when done

Binding can be to Internet addresses, as well

Connecting to Sockets

Method for setting up a receiving end of a socket

In local domain, similar to opening a file

Multiple clients can connect to a socket

- Program establishing socket can limit connections

Duality in IPC Mechanisms

Many aspects of IPC mechanisms are duals of each other

First recognized in the context of messages vs procedures

At least, IPC mechanisms can be simulated by other IPC mechanisms

A Case Study

OS's characterized as message-oriented or procedure-oriented

Study demonstrates a duality mapping between these types

To the extent that mechanical translation possible

Message-Oriented OS's

Support easy, efficient message passing

Generally with bound connections

Processes and connections relatively static

Processes operate in static context

Real time and process control systems

Consequences of Message-Oriented OS

- Synchronization via message queues
- Shared data structures passed in messages
- Peripheral devices treated as processes
- Static process priority
- Processes handle messages serially
- Generally no procedural interface

Important Message OS Concepts

A *message* is data structure sent between processes

A *message identifier* is a message handle

A *message channel* identifies destination of message

A *message port* is a queue capable of holding messages for a process

Processes hold certain ports and send to certain channels

Processes creation is heavyweight

Message Transmission Ops

SendMessage[msgChannel,msgBody] returns [msgID]

AwaitReply[msgid] returns [msgBody]

WaitForMessage[set of msgPorts] returns[msgBody,msgID,msgPort]

SendReply[msgID,msgBody]

A Message-Oriented Resource Manager

```
begin m: messageBody;

do forever;
[m,i,p]<-WaitForMessage[s];
case p of
port1=> . . .;/*algorithm for port 1*/
port2=> . . .;/*algorithm for port 2*/
.
.
.
endcase;
endloop;
end
```

Procedure-Oriented Systems

Efficient, safe procedure-call oriented system

Locking mechanisms for synchronization of global data structures

- System resources controlled through such structures

Process creation cheap

Consequences of Procedure-Oriented Systems

- Synchronization through queues of processes waiting on locks
- Data shared directly among processes
- Peripheral devices manipulate locks and shared data
- Process priorities
- Global naming schemes important

Important Procedure OS Concepts

A *procedure* is a collection of data and algorithms

With an external identifier

A *module* is a related set of procedures and data

A *monitor* is a lockable module

Procedure Manipulation Primitives

Fork and **Join** for async procedure call

New and **Start** to instantiate modules

Wait and **Signal** to manipulate condition variables

Entry to describe external procedure entry point within module

A Procedure-Oriented Resource Manager

```
ResourceManager:MONITOR =

proc1:ENTRYPROCEDURE[. . .] =
begin
. . .
end; /* algorithm for proc1*/
proc2:ENTRYPROCEDURE[. . .]  =
begin
. . .
end/* algorithm for proc2 */
.
.
.
end;
```

Duality Mapping

Message-Oriented System	Procedure-Oriented System
Processes, CreateProcess	Monitors,New/Start
Message Channels	External Procedure Identifiers
Message Ports	Entry procedure identifiers
SendMessage;AwaitReply	simple procedure call
Delayed send/reply	Fork; . . . Join
SendReply	return from procedure

Consequences of Duality Mapping

Programs in either style easily convertable

Without affecting program logic

Performance may be little affected, too

So Which Style to Build In?

Similar program structure and performance

Computational complexity similar

But hardware may offer special support for one or the other

Limitations on This Result

Only absolutley true to the extent systems match their models

Modern systems do not often match these models

So what’s the value of the result?

Insight into the basic similarities of IPC mechanisms