# Interprocess Communications, Continued

**CS239**
**Advanced Operating Systems**
**Spring 1996**



## Outline

- Shared memory IPC
- Shared memory and large address spaces
- Windows NT IPC Mechanisms



## Shared Memory

A simple and powerful form of IPC

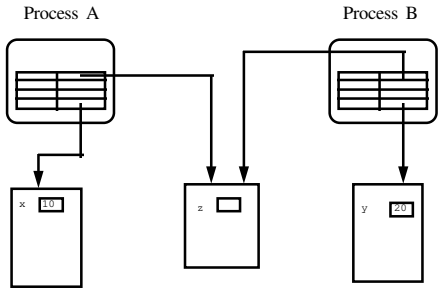
Most multiprogramming OS's use some form of shared memory

E.g., sharing executables

Not all OS's make shared memory available to applications



## Shared Memory Diagram






## Some Problems With Shared Memory

- Synchronization
- Protection
- Pointers



## Shared Memory and Synchronization

Shared memory itself does not provide synchronization of communications

Except at the single-word level

Typically, some other synchronization mechanism is used

- E.g., semaphores in Unix,
- Events, semaphores, or hardware locks in Windows NT

## Protection in Shared Memory

Who can access a segment? And in what ways?

Unix allows some read/write controls

Windows NT has general security monitoring based on the object-status of shared memory



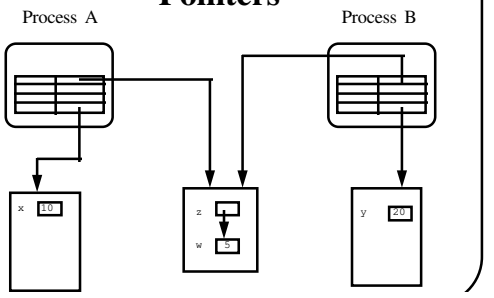
## Pointers in Shared Memory

Pointers in a shared memory segment can be troublesome

For that matter, pointers in any IPC can be troublesome

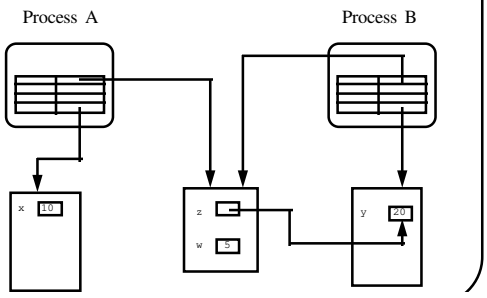


## Shared Memory Containing Pointers






## A Troublesome pointer






## So How Do You Share Pointers?

Several methods are in use

- Copy-time translation
- Reference time translation
- Pointer swizzling

All involve somehow translating pointers at some point before they are used



## Copy-Time Pointer Translation

When a process sends data containing pointers to another process,

Locate each pointer within old version of the data

Then translate pointers as required

Requires both sides to traverse entire structure

Not really feasible for shared memory

## Reference-Time Translation

- Encode pointers in shared memory segment as *pointer surrogates*
- Typically as offsets into some other segment understood within each separate contexts
- So each sharer can have its own copy of what is pointed to
- Slow, pointers in two formats



## Pointer Swizzling

- Like reference-time, but cache results in the memory location
- Only first reference is expensive
- But each sharer must have his own copy
- Must "unswizzle" pointers to transfer data outside of local context
- Stale swizzled pointers can cause problems



## Shared Memory in a Wide Virtual Address Space

- When virtual memory was created, 16 or 32 bit addresses spaces were available
- Reasonable size for one process
  - But maybe not for all processes on a machine
  - And certainly not for all processes ever on a machine



## Wide Address Space Architectures

- Computer architects can now give us 64 bit virtual addresses
- A 64 bit address space, consumed at 100 megabytes per second, lasts 5000 years
  - Orders of magnitude beyond any process' needs
  - 40 bits can address a terabyte



## Do We Care?

- Should OS designers care about wide address spaces?
- Well, what can we do with them?
- One possible answer---
  - Put all processes in the same address space
  - Maybe all processes for all time?



## Implications of Single Shared Address Space

- IPC is trivial
  - Shared memory, RPC
- Separation of concepts of address space and protection domain
- *Uniform-address space*

## Address Space and Protection Domain

A process has a protection domain

The data that cannot be touched by other processes

And an address space

The addresses it can generate and access

In standard systems, these concepts are merged



## Separating the Concepts

These concepts are potentially orthogonal

Just because you can issue an address doesn't mean you can access it

(Though clearly to access an address you must be able to issue it)

Existing hardware can support this separation



## Context-Independent Addressing

Addresses mean the same thing in any execution context

So a given address always refers to the same piece of data

Key concept of uniform-address systems

Allows many OS optimizations/improvements



## Uniform-Addressing Allows Easy Sharing

Any process can issue any address

So any data can be shared

All that's required is changing protection to permit desired sharing

Suggests programming methods that make wider use of sharing



## The Opal System

New OS using uniform-addressing

Developed at U. of Washington

Not intended as slight alteration to existing Unix system

Most of the rest of material specific to Opal



## Protection Mechanisms for Uniform-Addressing

Protection domains are assigned portions of the address space

They can allow other protection domains to access them

- Read-only
- Transferable access permissions
- System-enforced page-level locking

## Program Execution in Uniform-Access Memory

Executing a program creates a new protection domain

The new domain is assigned an unused portion of the address space

But it may also get access to used portions

E.g., a segment containing the required executable image



## Virtual Segments

Global address space is divided into segments

Each composed of variable number of contiguous virtual pages

Domains can only access segments they *attach* to

Attempting to access unattached segment causes a segment fault



## Persistent Memory in Opal

*Persistent* segments exist even when attached to no current domain

*Recoverable* segments are permanently stored

And can thus survive crashes

All Opal segments can be persistent and recoverable

Pointers can thus live forever on disk



## Code Modules in Opal

Executable code stored in *modules*

Independent of protection domains

*Pure modules* can be easily shared

Because they are essentially static

Can get benefit of dynamic loading without run-time linking



## Address Space Reclamation

Trivial in non-uniform-address systems

Tricky in uniform-address systems

Problem akin to reclaiming inodes in the presence of hard links

But even if segments improperly reclaimed, only trusting domains can be hurt



## Windows NT IPC

- Inter-thread communications
  Within a single process
- Local Procedure Calls
  Between processes on same machine
- Shared Memory

## Windows NT and Threads

Windows NT supports multiple threads of control in a single process address space

Threads share address space

So communication between them is through memory



## Windows NT and Client/Server Computing

Windows NT strongly supports the client/server model of computing

Various OS services are built as servers, rather than parts of the kernel

Windows NT needs facilities to support client/server operations

Which guide users to building client/server solutions



## Client/Server Computing and RPC

In client/server computing, clients request services from servers

Service can be requested in many ways

But RPC is a typical way

Windows NT uses a specialized service for single machine RPC



## Local Procedure Call (LPC)

Similar in many ways to RPC

But optimized to only work on a single machine

Primarily used to communicate with protected subsystems

Windows NT also provides a true RPC facility for genuinely distributed computing



## Basic Flow of Control in Windows NT LPC

Application calls routine in an application programming interface

Which is usually in a dynamically linked library

Which sends a message to the server through a messaging mechanism



## Windows NT LPC Messaging Mechanisms

- Messages between port objects
- Message pointers into shared memory
- Using dedicated shared memory segments

## Port Objects in Windows NT

Windows NT is generally object-oriented

Port objects support communications

Two types -

- Connection ports
- Communications ports



## Connection Ports

Used to establish connections between clients and servers

Named, so they can be located

Only used to set up communications ports



## Communications Ports

Used to actually pass data

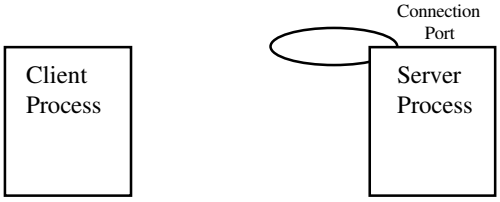
Created in pairs, between given client and given server

Private to those two processes

Destroyed when communications end

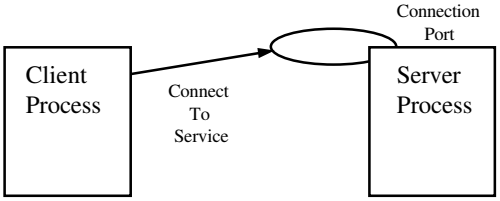


## Windows NT Port Example



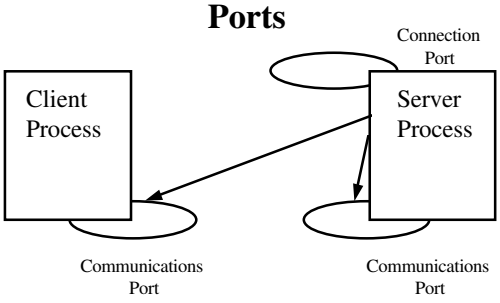



## The Connection Request

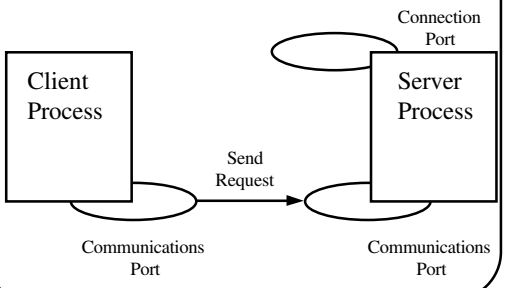





## Setting Up Communications Ports



## Sending a Request






## Message Passing Through Port Object Message Queues

One of three methods in Windows NT to pass messages

1. Client submits message to OS
2. OS copies to receiver's queue
3. Receiver copies from queue to its own address space



## Characteristics of Message Passing Via Queues

- Two message copies required
- Fixed-sized, fairly short messages
  - 256 bytes
- Port objects stored in system memory
  - So always accessible to OS
- Fixed number of entries in message queue



## Message Passing Through Shared Memory

Used for messages larger than 256 bytes

Client must create *section object*

Shared memory segment

Of arbitrary size

Message goes into the section

Pointer to message sent to receiver's queue



## Setting Up Section Objects

Pre-arranged through OS calls

Using virtual memory to map segment into both sender & receiver's address space

If replies are large, need another segment for the receiver to store responses

OS doesn't format section objects



## Characteristics of Message Passing Via Shared Memory

- Capable of handling arbitrarily large transfers
- Sender and receiver can share single copy of the data
  - i.e., data need be copied only once
- Requires pre-arrangement for section object

## Server Handling of Requests

Windows NT servers expect requests from multiple clients

Typically, they have multiple threads to handle requests

Must be sufficiently general to handle many different ports and section objects



## Message Passing Through Quick LPC

Third way to pass messages in Windows NT

Used exclusively with Win32 subsystem

Like shared memory, but with a key difference -

Dedicated resources



## Use of Dedicated Resources in Quick LPC

To avoid overhead of copying notification messages to port queue

And thread switching overhead

Client sets up dedicated server thread only for its use

Also dedicated 64Kb section object

And *event pair object*

For synchronization



## Characteristics of Message Passing Via Quick LPC

- Transfers of limited size
- Very quick
- Minimal copying of anything
- Wasteful of OS resources



## Shared Memory in Windows NT

Similar in most ways to other shared memory services

Windows NT runs on multiprocessors, which complicates things

Done through virtual memory



## Shared Memory Sections

Block of memory shared by two or more processes

Created with unique name

Can be very, very large

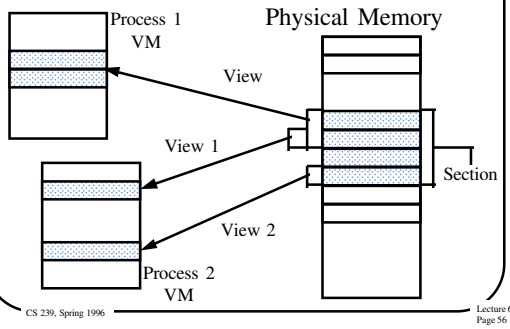
## Section Views

Given process might not want to waste lots of its address space on big sections

So a process can have a *view* into a shared memory section

Different processes can have different views of same section

Or multiple views for single process



## Shared Memory View Diagram