Interprocess Communications, Con’t

CS 239 - Advanced Operating Systems

April 20, 1995

Interprocess Communications, Con’t

- Remote procedure call
- Shared memory and large address spaces

Remote Procedure Call

Method of calling procedures in other address spaces

Either on the same machine

Or other machines

Attempts to provide interface just like local procedure call

Request/Reply communications model

Motivations for RPC

- Procedure calls are well understood
- Clear semantics
- Useful for programming in procedural languages
- Natural fit for client/server computing

RPC Case Studies

RPC in Cedar

Unix RPC

Semantics of RPC

Similar to regular procedure call

1. Calling procedure blocks
2. Set of parameters transferred to called procedure
3. Called procedure computes till it returns
4. Return value delivered to calling procedure
5. Calling procedure continues

High-Level RPC Mechanics

Hide details from applications

Clients pass requests to stub programs

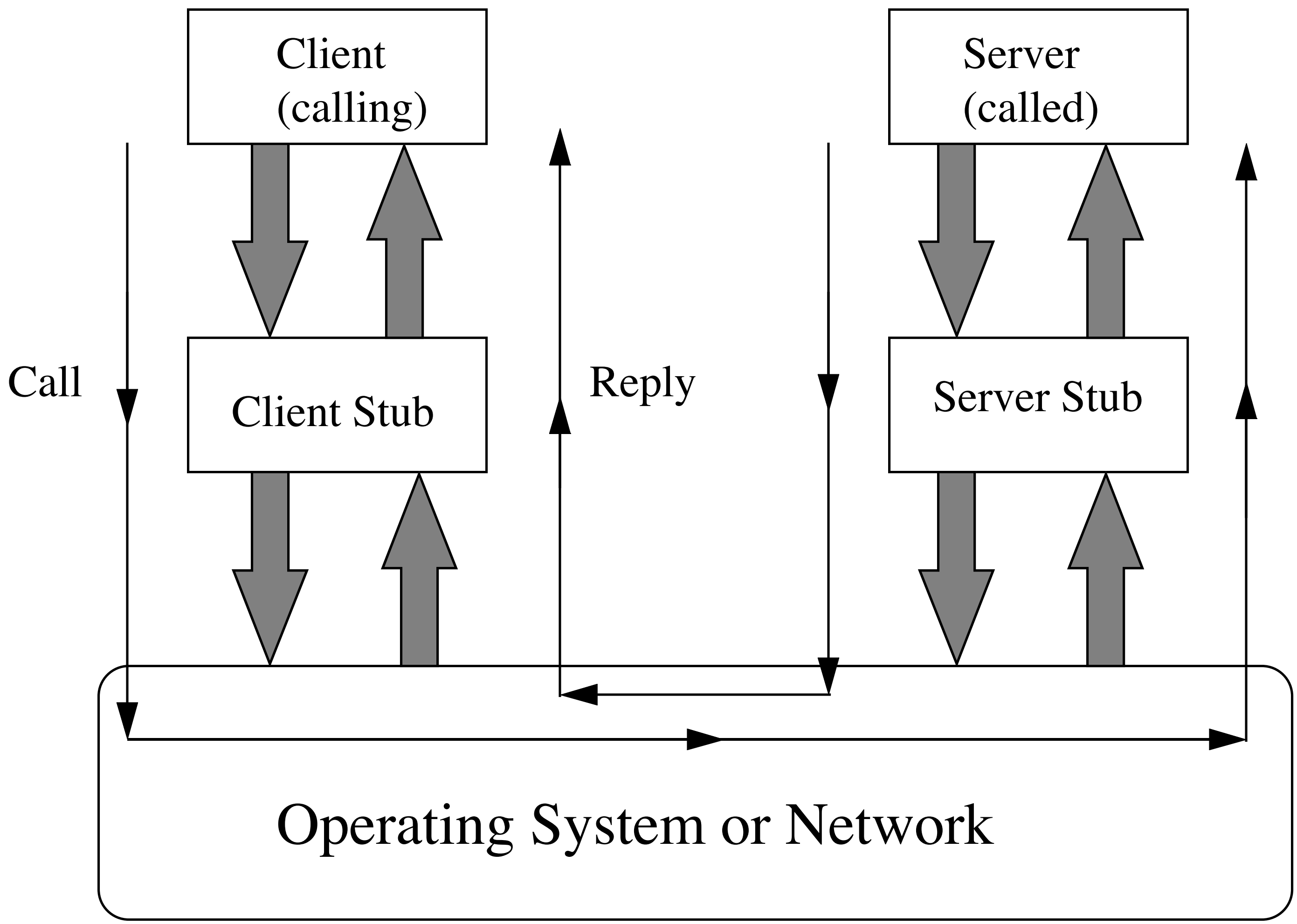
Client-end stub sends request to server stub

Server-end stub calls user-level server

Results travel in reverse direction

Network transport or OS actually moves data

Diagram of RPC in Action



What Do the Stubs Do?

Stubs handle complex details like:

- Marshalling arguments
- Message construction
- Data format translation
- Finding the server process

Setting Up RPC

Caller must have a way to find the called procedure

But it can’t be found at link time

- Unlike local procedures

Potential servers must make their presence known

Registering Servers

Either register the server at a “well-known” port

Or register it with a name server

- Which in turn is at a “well-known” port

Calling procedure “addresses” RPC to that port

Binding To a Service

A client process *binds* to the service it wants

Two major sub-problems:

- Naming
- Location

Binding: The Naming Problem

How does a caller name the server program?

Depends on the RPC mechanism

  And perhaps the protocol or type within it

Important naming question:

- Do you name the server explicitly?
- Or do you name the service and let some other authority choose which server?

Binding: The Location Problem

Where is the remote server?

Some naming schemes make it explicit

- Some don't

If it's not explicit, system must convert symbolic names to physical locations

Binding in Cedar RPC

Client applications bind to a symbolic name

- Composed of:
  - *Type*
  - *Instance*

Type is which kind of service

Instance is which particular implementor

Locating Cedar RPC Service

Names do not contain physical location

So Cedar consults a database of services

Services register with database

Binding call automatically looks up location in database

Binding in Unix RPC

bind() system call used by servers

- Allows servers to bind naming/location information to particular socket

connect() system call used by clients

- Using information similar to that bound by the server

Typically, all hidden from RPC programmer by automatically generated code

Unix Binding Information

Like most socket operations, it's flexible

Fill in all or part of a socket address data structure

Create an appropriate socket

Then call **bind** to link socket to address info

connect works similarly

Unix Binding Example

On server side,

```
struct sockaddr_un sin;
int sd;

strcpy(sin.sun_path,"./socket");
sd = socket(AF_UNIX,SOCK_STREAM,0);
bind(sd,&sin,sizeof(sin));
```

Unix Binding Example, Con’t

For client side,

```
struct sockaddr_un sin;
int sd;

strcpy(sin.sun_path,”./socket”);
sd=socket(AF_UNIX,SOCK_STREAM,0);
connect(sd,&sin,sizeof(sin));
```

Locating Remote Services in Unix RPC

- Similar to Cedar methods
- Register services with the portmapper
- The portmapper is typically called through automatically generated code
- The portmapper runs on each machine
- Another service (e.g., NIS) deals with intermachine requests

Using RPC

Once it's bound, how does the client use RPC?

Just call the routines

As if they were local

And the results come back

What's Happening Under the Covers?

When a client calls a remote routine, he really calls a local stub program

Stub program packages up request to remote server

And sends it to local transport code

When reply arrives, local transport code returns results to stub

Which returns to client program

What Happens at the Server Side?

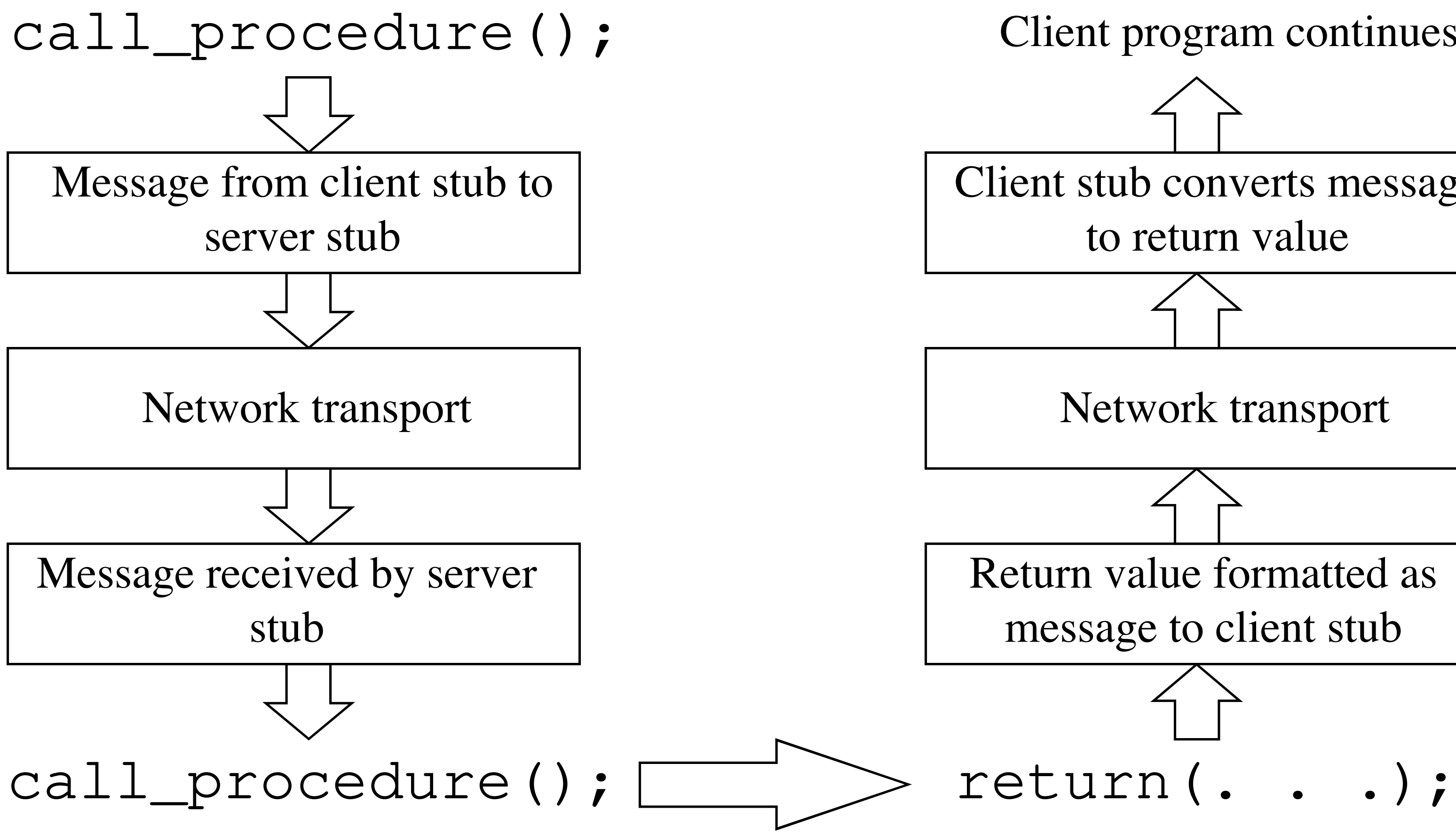
A request comes in to the RPC transport code

It routes it to the appropriate server stub

Which converts it into a local procedure call

Which is made within the context of the server

Conceptual Diagram of RPC



Transport For RPC

In Cedar, special-purpose RPC transport protocol

In Unix RPC, can use either UDP or TCP for transport

- Typically protocol is chosen automatically by stub generator program

Other RPC Issues

Mostly related to intermachine RPC

- Data format conversions
- Security and authentication
- Locating remote services
- Multipacket RPC

Shared Memory

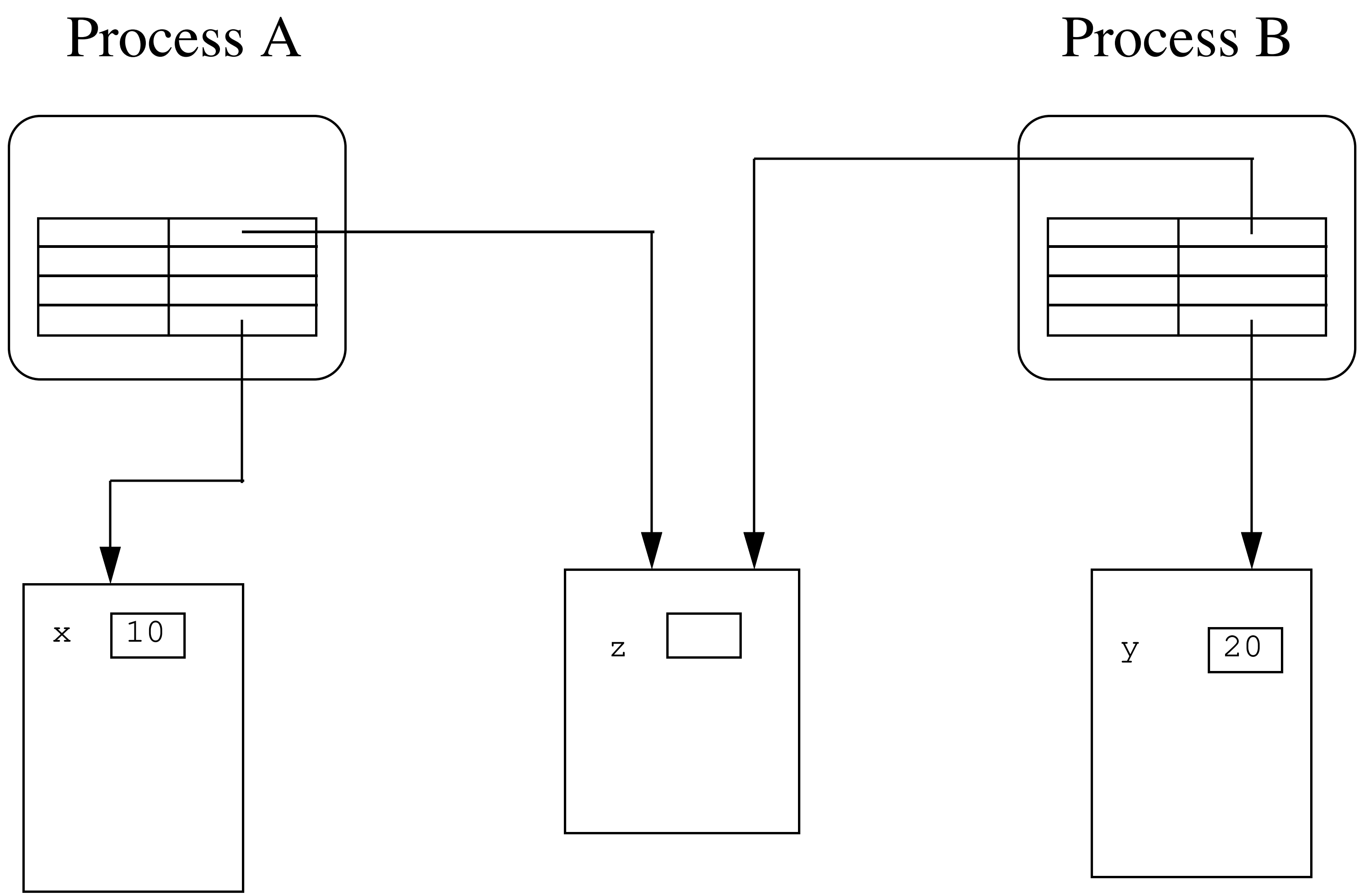
A simple and powerful form of IPC

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

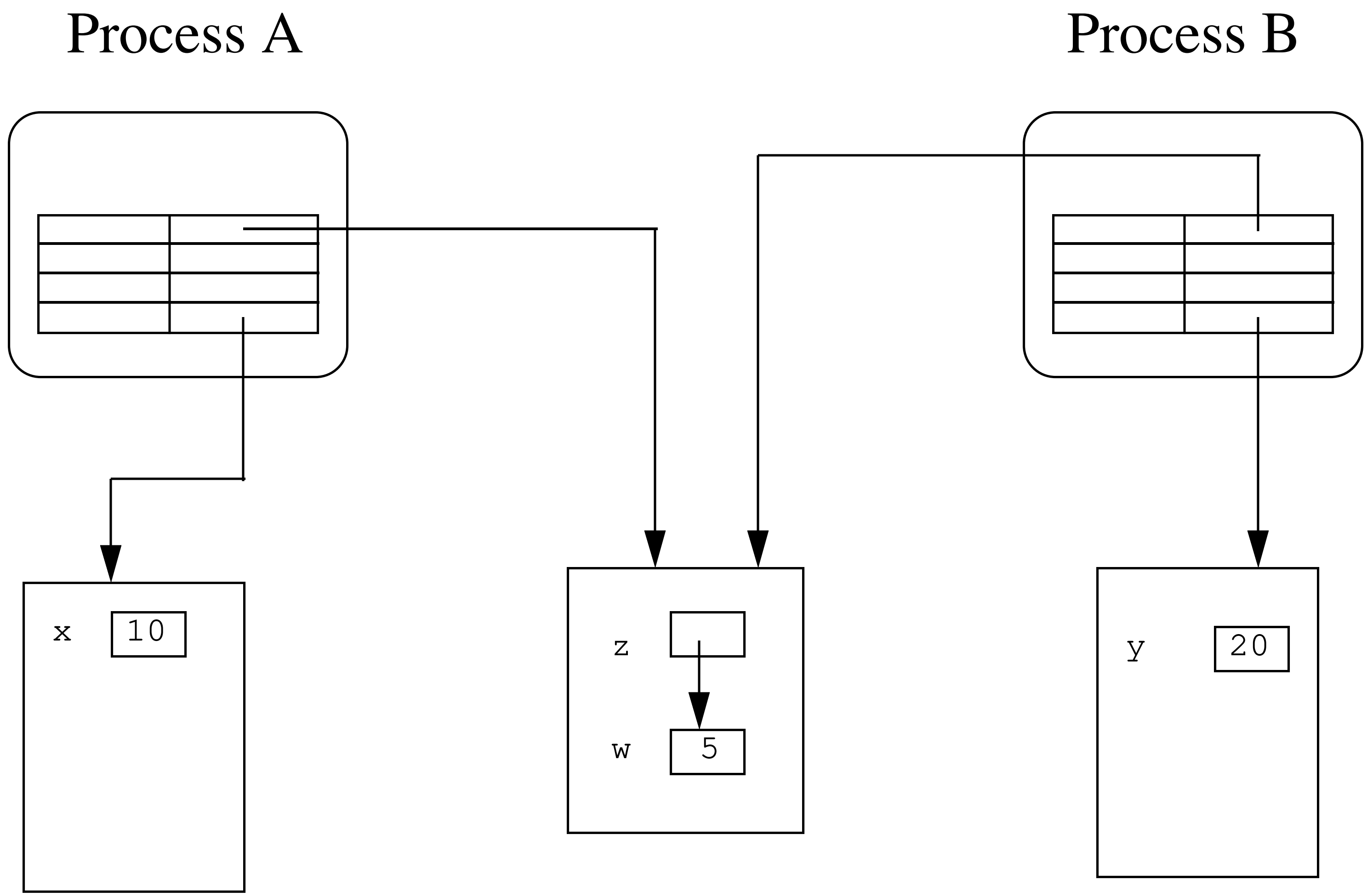
Windows NT has general security monitoring based on the object-status of shared memory

Pointers in Shared Memory

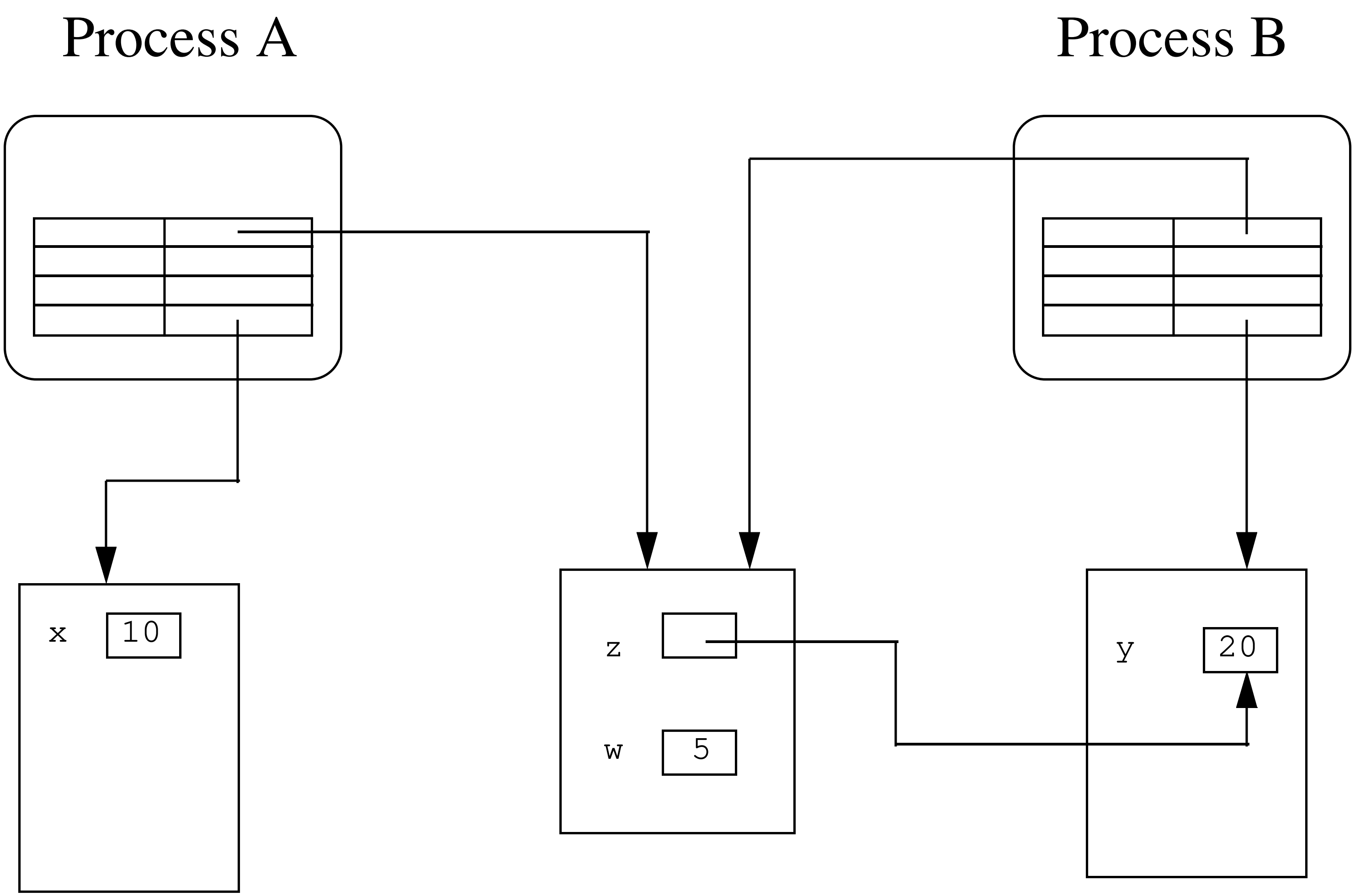
Pointers in a shared memory segment can be troublesome

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 bi addresses spaces were available

Reasonable size for one process

- But maybe not for all processes on a machine
- And certainly not for all processes <u>ever</u> 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
- Transferrable 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