

Remote Procedure Calls (RPC)

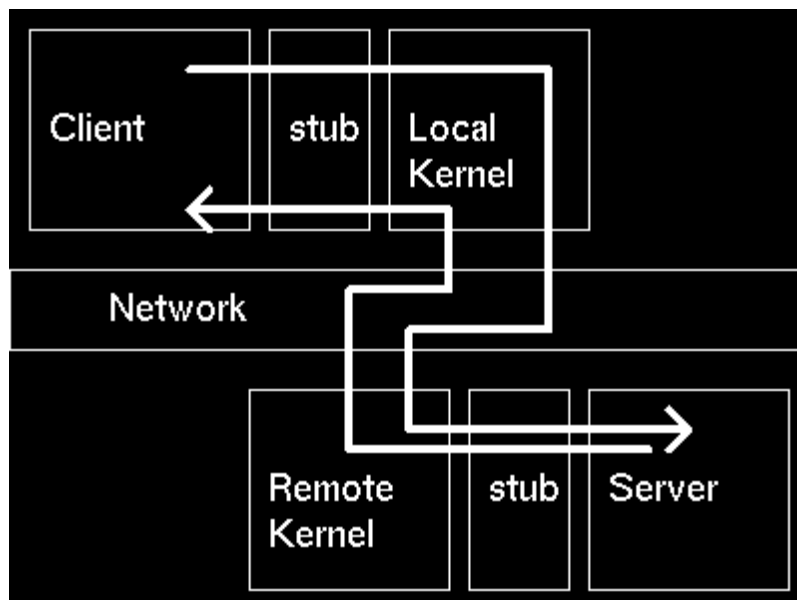
Birrell & Nelson 1984

Hide the message passing system so that it looks like a series of procedure calls.

Most requests for service wait until the request is fulfilled – semantically just like a procedure call.

Programmer doesn't have to package and unpackage data in the messages.

Also adds flexibility because sometimes the service might actually be local.



Client and Server stubs

- Client makes ordinary procedure call to the local stub.
- Stub marshals parameters (may need to locate server as well).
- Stub sends request via local kernel.
- Remote kernel passes request to Server stub.
- Server stub unpacks message request and parameters.
- Makes ordinary procedure call to Server.
- And then vice-versa.

The stubs at both ends need to be constructed from the same interface specification - to ensure consistency.

Care has to be taken about different versions of the service. A different version may take slightly different parameters or return different types etc.

RPC messages

Messages are highly structured

what procedure to execute

parameters

version number

service may survive a long time and code may be written to different versions

timestamp

could be used for synchronization purposes

source address

where to send the results

possibly the type of machine the request comes from

Finding the server

- include port numbers at compile time or
- have a binder or rendezvous (or matchmaker) service

Server usually registers with a binder or name server

client end sends the request to find the server

multiple servers – the binder can spread the load

binder can periodically check on servers - deregistering those that don't respond

binder could do the security work

otherwise servers have to check each call

Marshalling

Heterogeneous networks.

Data formats in the different machines may be different.

- ASCII, Unicode, EBCDIC for strings
- Big or little endian for integers
- Different floating point formats

Stubs need to know about this.

Different solutions

client stub converts to the server format

server stub converts from the client format

convert to and from a canonical format

no one needs to know other machines formats

(we don't want to have to convert from format A to canonical to A and then back again)

Reference parameter problem

The references no longer make sense.

- either disallow reference parameters
- or copy the parameter to and fro
- either in one chunk or whenever the server makes a change to it
(this is just like distributed shared memory)

Copy/restore semantics aren't quite the same.

- e.g. pointer as a reference parameter
- the same parameter passed twice in the parameter list
we can check for this

Sometimes we don't need to copy the data both ways
e.g. input buffer (only needs to be copied from the server to the client)

Our interface specification should be able to express this.

RMI & CORBA

Remote Method Invocation

- Java technology

- RPC with objects

- Solves object reference problem by sending serialized versions of the object. If the object passed as a parameter is itself actually on the remote site it is just sent by reference.

Common Object Request Broker Architecture

- Similar to RMI

- Works with a wide variety of languages – not just Java

- Needs a common description language to specify interfaces – IDL Interface Definition Language

Linda tuplespace

Another technique to share services over a network. Rather than explicitly sending messages we can share data in a tuplespace.

The tuplespace is a logically shared (sometimes distributed) memory consisting of tuples. A tuple is a list of parameters, some of which are empty, e.g, <"hi", 15, 12.5>

Tuples are put into the tuplespace by the "out" primitive and retrieved from the tuplespace by matching contents and types with the "in" primitive. The example above could be matched with <"hi", _, _>

JavaSpaces are an implementation of Linda tuplespaces. Rinda is the Ruby version of Linda. Linda originally worked with C and Fortran.

Tuplespace advantages

Processes don't communicate directly with each other but instead access the tuplespace. It doesn't matter if one process dies, as long as another that deals with the same tuple is available (and the tuple hasn't been removed yet).

Tuplespaces scale naturally. Adding more processes which handle particular requests is trivial.

The space itself deals with synchronization. Each tuple operation is atomic.

Process Migration

Moving a process from one site to another while it is running.

Why would we like to do process migration?

- To enable us to do proper load balancing.
- If the process can be subdivided we can increase performance by having different parts running simultaneously on different machines.
- To move the process closer to the resources it is currently accessing.
- To move the process closer to the user e.g. connected via a WAN but not on the user's default machine.
- To enable us to keep a process going when the site it is executing on has to be taken down.

What do we need?

We need location and migration transparency of
processes
resources used by the process

What defines a process?

- PCB
- Resources
 - files
 - communication channels
 - memory
 - devices - including windows, keyboard, mouse
- Threads

Need some compatible machine (or virtual machine) architecture.

Internal and external references

- References to resources within the program.
- References to the process from outside e.g. other processes communicating with it.

How can that be done?

Need a way of referring to all resources indirectly via global tables (like we did with our distributed file systems).

We can extend the ideas of a distributed file system to refer to other objects, including processes.

All process identifiers have no host information in the identifier.

e.g.

- A process table keeps track of which site each process is running on.
- When the process is moved the table is updated.
- Caching of information can be used for efficiency but we need ways to recover when the cache data is out of date.
- Not all processes need to be stored in this table.
 - Processes specific to a site which are not visible away from the site.

Doing the migration

Minimise the amount of down time

Process must be stopped at some stage

Stop, copy, notify

How much do we copy?

Only the working set

- get the remaining pages by demand paging

- Can't be used if the host is going down.

Everything, but don't stop the process

- then copy pages which were dirtied during the copy

Both approaches only stop the process while the working set is moving.

Current uses

In reality process migration is not used for load balancing.

It is too expensive.

- Most processes only run for a few seconds.
- Transferring a process can easily take a few seconds.

It is still useful when a machine needs to be closed down for maintenance and it has running processes which we don't want to kill.

Common use - idle workstations

Move processes when no longer idle.

- Generally load balancing is only done when a process starts
 - or when it has to move.
 - Where is the best place to run this process?
-

The textbook also talks about Computation Migration – this means sending messages (or RPCs) to get work done on another site.

New topic - Memory

If we define memory as a place where data is stored there are many levels of memory:

- Processor registers
- Primary (or main) memory
 - RAM
 - these days the stuff that comes on sticks
- Secondary memory
 - slower and more permanent
 - disks
- Tertiary memory
 - archival
 - removable
- Cache memory
 - one level of memory pretending to be another
 - different levels of cache
 - at the processor
 - at devices
 - and at hosts in distributed file systems

Main Memory

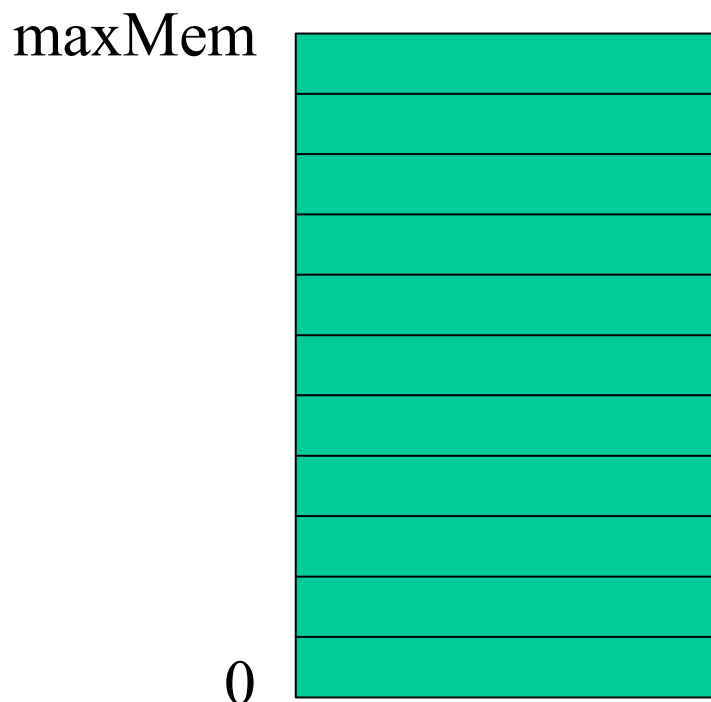
All processes need main memory.

Instructions and data being worked on are brought into the processor from memory and sent back to memory.

Traditional view:

Addresses are numbers: 0 – maxMem

An address goes out of the processor to the address bus of memory.



Address binding

We can make the connection between code and data and their addresses at different times:

- **Compile time**

Need to know exactly where the value will be stored.

Fixed addresses.

Not very flexible.

- **Load time**

Object modules need tables of offsets.

e.g. variable x is at 24 bytes from the start of the module

The mapping is done as the module is loaded.

Can also reference other modules – linking.

More flexible but can't be changed after loading.

- Dynamic loading – don't load unless called
- Dynamic linking similar – but can be at a different address (maybe a library brought in by another process) – needs OS help

- **Run time**

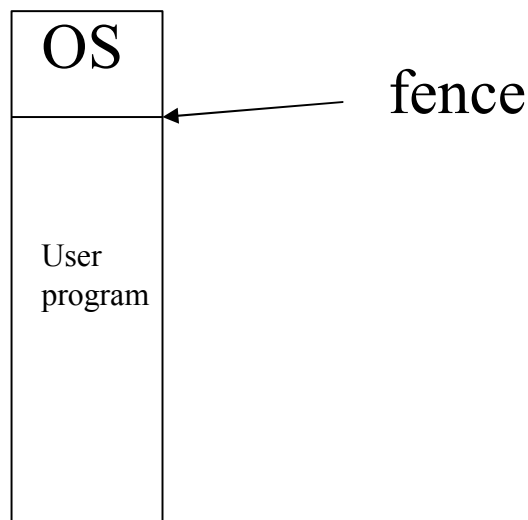
Mapping to final address maintained on the go by hardware.

Memory spaces

Even in simple systems we would like a way of protecting OS memory from our program and enabling programs larger than memory to run.

Split memory

Can protect with a single fence register.



Overlays

Load needed instructions over the top of ones not needed any more.

Dividing memory

With linking loaders we can have multiple programs in memory simultaneously.

We also want protection.

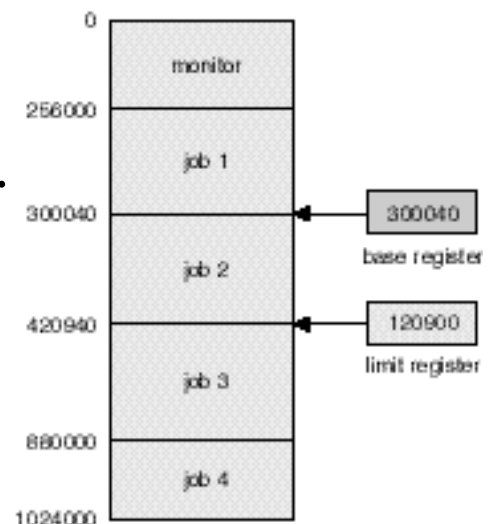
In the history section we saw separate partitions and base and limit registers.

Both require contiguous memory.

The algorithm for base and limit registers is simple.

If the address is less than the base or greater than the base + limit - 1 then we have an access violation.

The base and limit registers are loaded for each process.



Two different addresses

If we change our base and limit system to produce the address by adding the base register (now called a relocation register) to each address we can make all processes start at address 0.

This is a huge conceptual change.

We now have two types of addresses.

- The logical (virtual) address coming out of the process
- and the physical (real) address used on the address bus to memory.

We still have contiguous memory for each process but a process can now be positioned anywhere in physical memory.

We can even move a process around, just copy the memory to the new place and change the relocation register.

This is useful if we want to defragment memory to give one large free area.

Have to be careful if moving data (e.g. from a device) into the process' memory space.

Before next time

Read from the textbook

4.6.2 – Remote Procedure Calls

4.6.3 – Remote Method Invocation

9.1 – Memory Background