UNIVERSITY OF PORTSMOUTH

Operating Systems and Internetworking

OSI – M30233 (OS Theme)

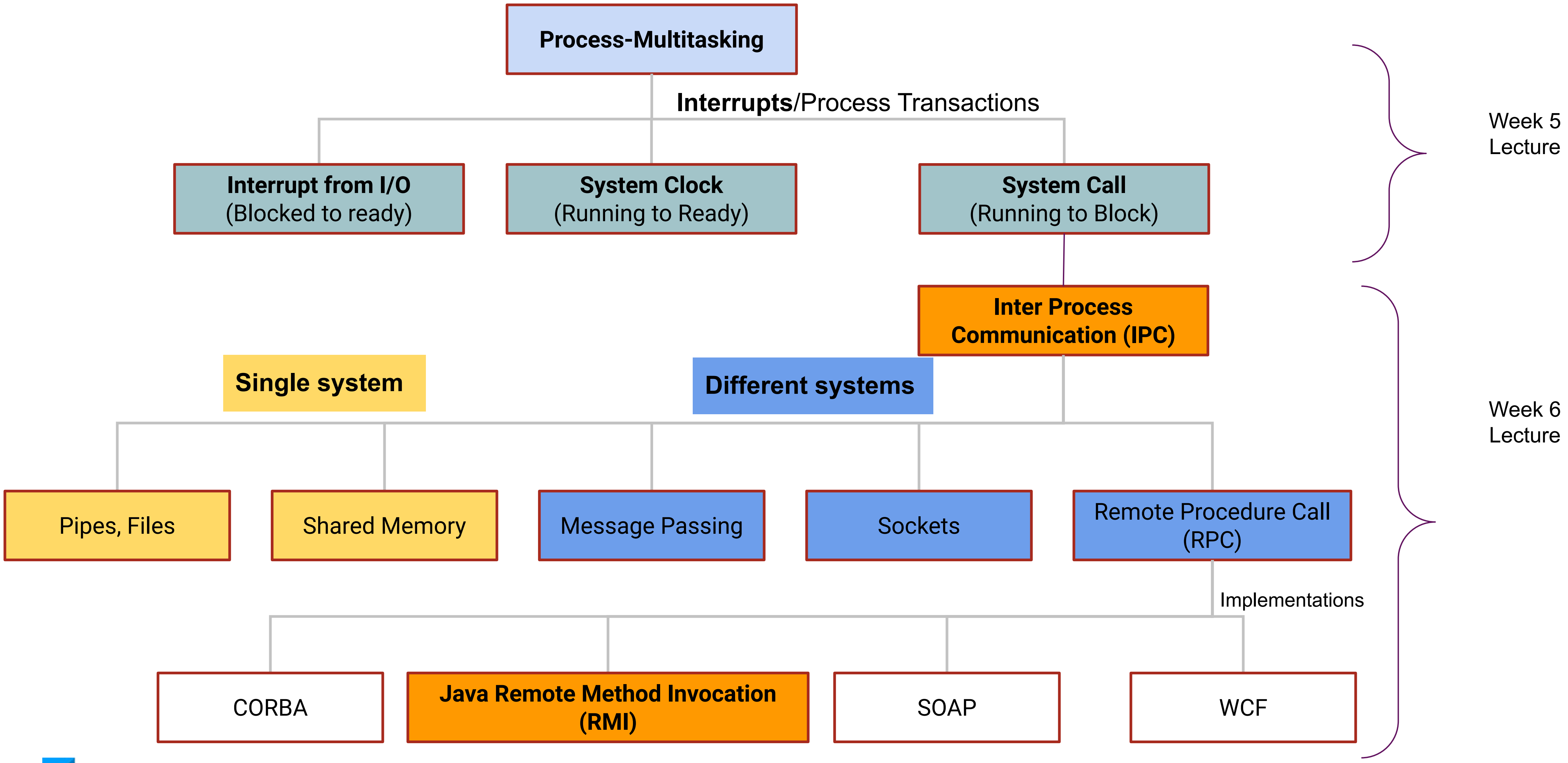
Week 6- Inter-Process Communication

Dr Tamer Elboghdadly

Interprocess Communication

- **Plan:**
  - Interprocess Communication (IPC) overview
  - IPC for communication between processes in "**single systems**"
  - IPC for communication between processes running on **different systems.**

UNIVERSITY OF PORTSMOUTH



UNIVERSITY OF PORTSMOUTH

Inter-process Communication (IPC)

- In earlier lectures we considered interaction between *threads* in general, using access to shared variables, and semaphores, etc.
- But *processes* generally have their own private address space, and <u>don't have any program variables in common</u>.
- Traditionally, OS-supported mechanisms by which processes running in the same computer cooperate are called *Inter-process Communication* (IPC)
- We will take a broader view of IPC, and include mechanisms through which processes on *different* computers can communicate.

UNIVERSITY OF PORTSMOUTH

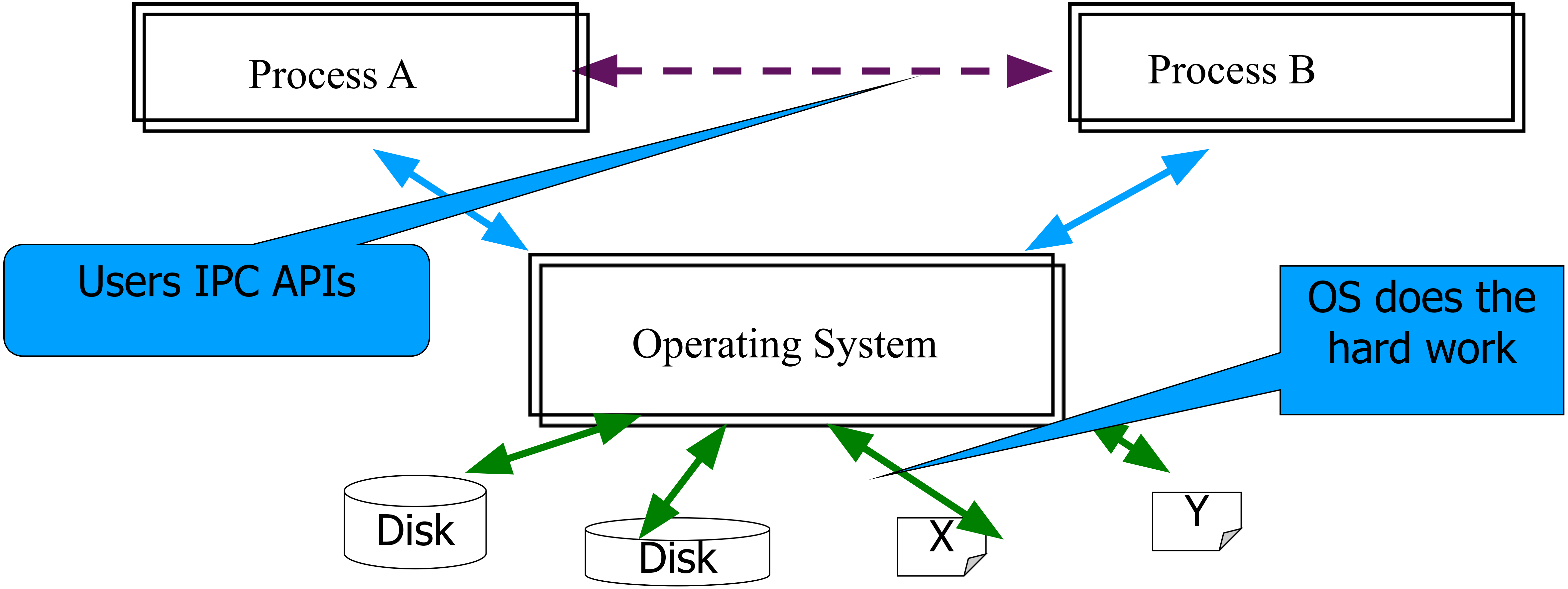
Benefits and Problems

- Advantages of process cooperation:
  - Information sharing,
  - Computation speed-up (parallel processing),
  - Modularity.
- Questions:
  - How do the processes “share” or “communicate” data?
  - What are the problems/issues of sharing data?

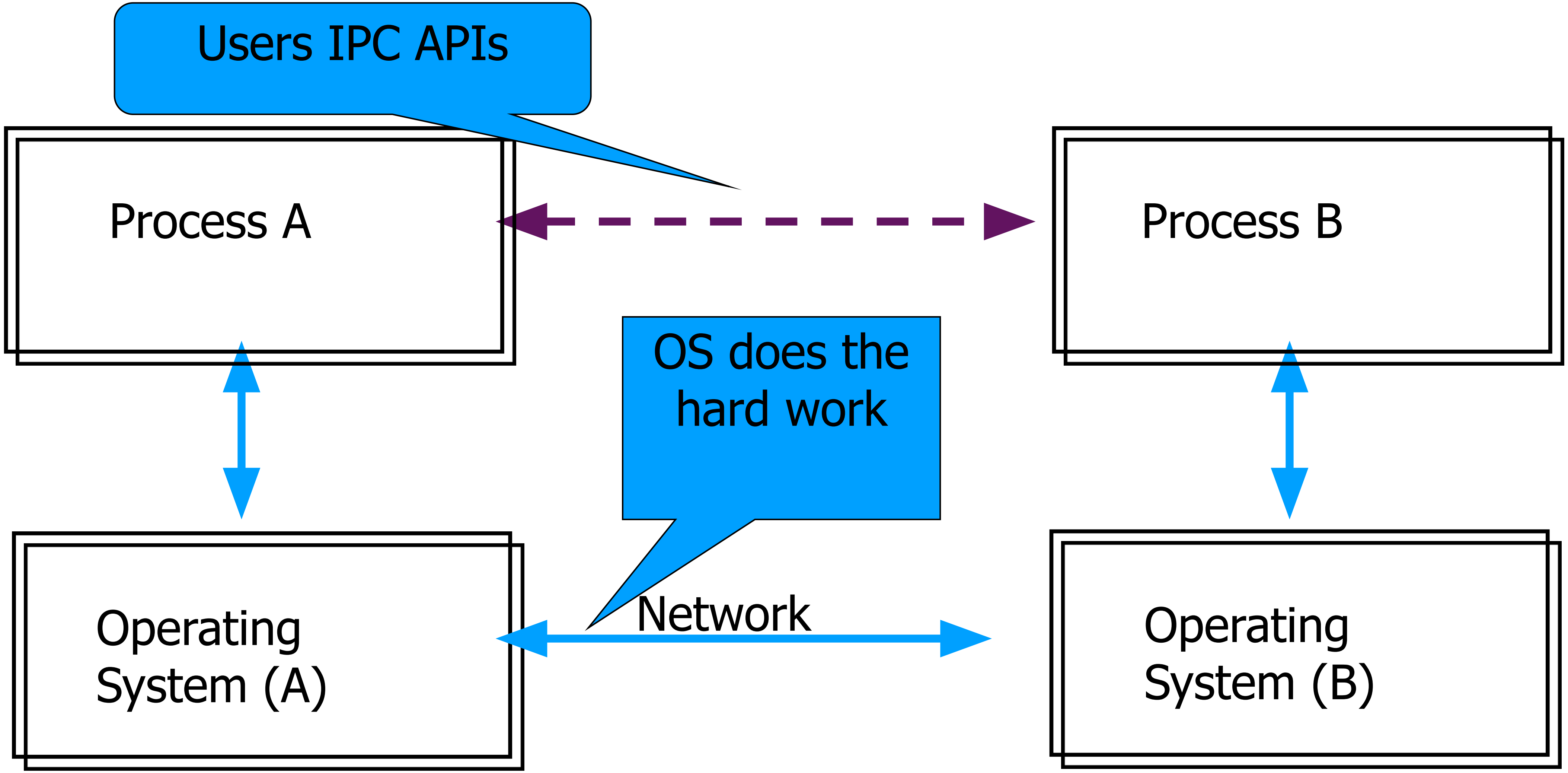
How Can Processes communicate?

- IPC mechanisms include:
  - Pipes and sockets (stream oriented).
    - Processes communicate in continuous *streams of bytes* sent over persistent connections.
  - Shared memory (memory oriented).
    - Processes communicate with each other through shared variables in *memory.*
  - Message passing (message oriented).
    - Processes communicate by sending each other discrete *chunks of data - messages.*
  - Remote procedure call (procedure oriented).
    - Processes communicate using *procedure call*

Single System IPC



IPC Across Different Systems



UNIVERSITY OF PORTSMOUTH

Towards Common APIs

- *Application Programming Interface* (API) is the generic name for an interface to some library of software functions. It is a connection between computers or between computer programs to allow communications.
- We first review some "traditional" APIs for IPC - specifically applicable to the "single-system" case.
- *But*, has become increasingly common to adopt the more general "different-systems" APIs, *even* when the communicating processes happen to run on the same system
  - In recent years much work has been put into developing these APIs for *distributed systems*.

SINGLE SYSTEM IPC

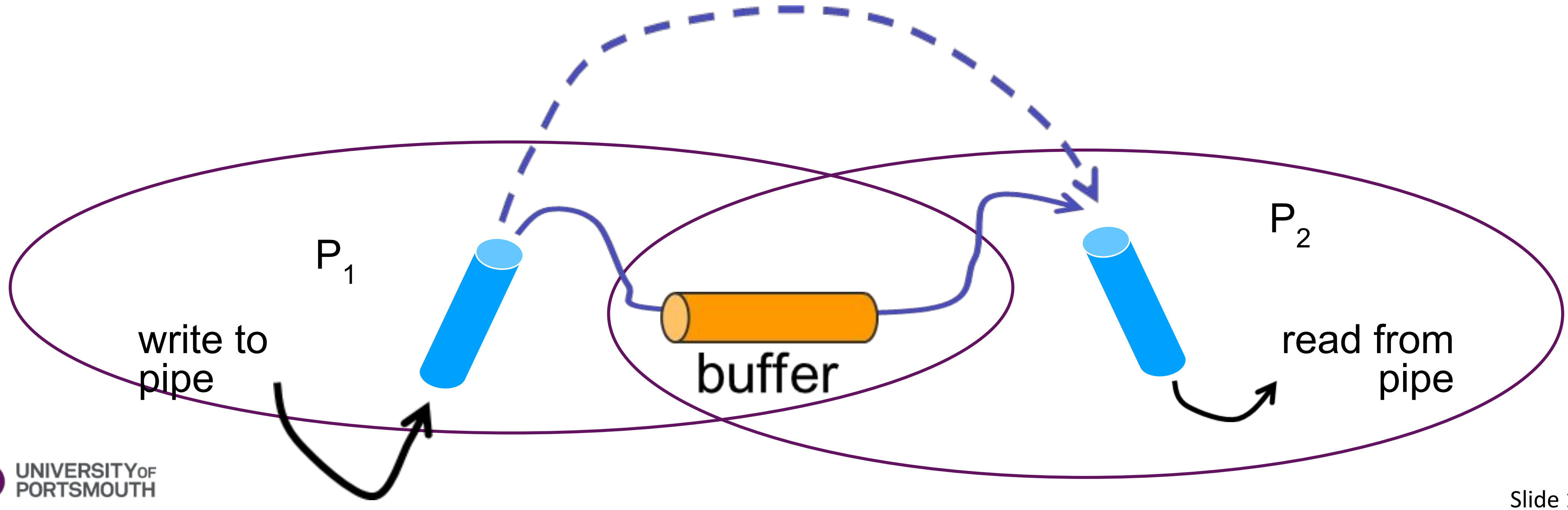
Traditional IPC: Pipes

- One of the simplest forms of single-system IPC
  - Still ubiquitous in UNIX-like systems (e.g. Linux).
- A UNIX *pipe* has an input and an output.
  - A stream of bytes is written to the output; the same stream of bytes is read from the input.
  - The inputting process will *block* if there is no data currently in the pipe.
- A pipe is created by some *parent process*, then used to communicate between (typically) two *child processes*.

UNIVERSITY OF PORTSMOUTH

Stream-based IPC Using Pipes

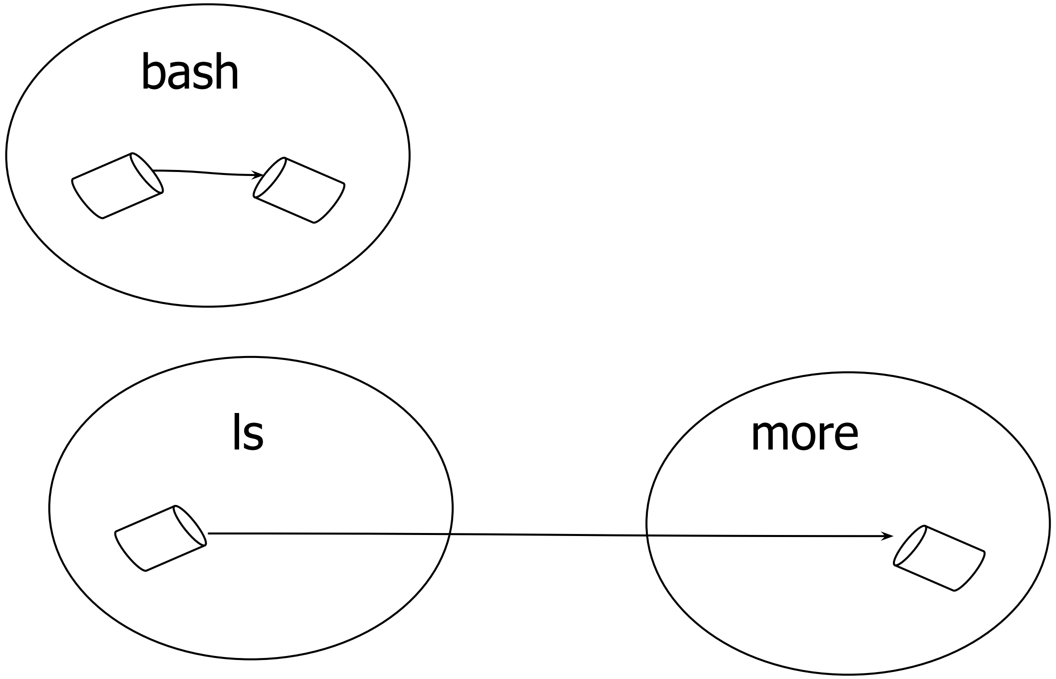
- write and read like a file, but more efficient.
  - One way byte stream.
  - Kernel buffers the data.
  - Operates as a *bounded buffer* with blocking (buffer is 64KB in Linux).



Scenario

- Most familiar use is in UNIX shell, e.g.:

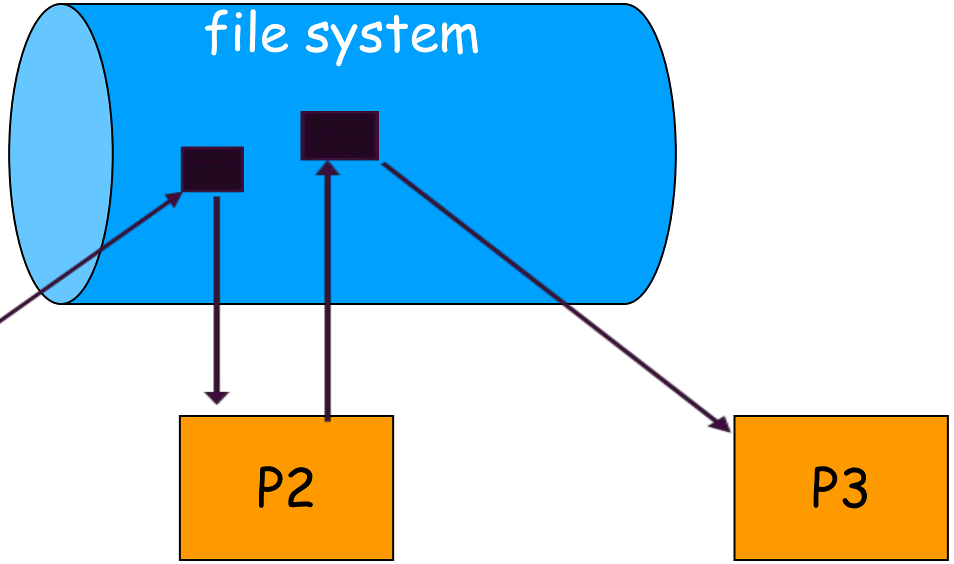
```
$ ls | more
```



1. Shell creates pipe with two ends (`pipe` system call)
2. Shell creates two child processes (`fork` system calls), passing one end of pipe to each.
3. Child processes exec `ls` and `more` commands respectively.
4. They communicate through the inherited pipe.

Traditional IPC: Shared Files

- Require file locking or record locking to allow cooperating processes to share a resource safely.
  - File – locks the whole file;
  - Record – locks portion of a file.



Traditional IPC: System V

- *System V* was a dialect of UNIX developed in the 1980s.
- Many features adopted into "Portable Operating System Interface" (*POSIX)* standards, and still available today in Linux, etc.
- It incorporated *APIs* for single-system IPC supporting, for example:
  - Shared memory segments
  - Semaphores
  - Message queues

Shared Memory

- We have emphasized in earlier lectures that processes (in contrast to threads) have *private* address spaces, and in general *don’t* share memory.
- This is the general rule, but specific *system calls* can be used to *create* memory areas that can be accessed by multiple processes.

System V Shared Memory API

- Creating shared memory segments is generally done in systems languages like *C*, e.g.:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

- Creates and attaches to a shared memory segment of size 1024 bytes, identified by `key`.

† Example taken from:
http://beej.us/guide/bgipc/html/single/bgipc.html#shmat

Memory Mapped Files

- On modern UNIX-like systems it is more common to implement shared memory segments by using the *virtual memory system* explicitly.
- A process can explicitly map a specified file into their *memory space* using the POSIX function **`mmap`**.
- If two or more processes map *the same file*, this effectively creates a shared memory region.

Shared Memory: Issues

- Once we have shared memory, the same issues of *data consistency* arise as with threads
  - How to prevent processes getting in each others way when performing critical updates?
  - How to make sure processes access shared data in some orderly patterns?
- The same kinds of solutions apply, and System V and POSIX provide *semaphores* that can be accessed from multiple processes.

IPC ACROSS COMPUTERS

IPC via Message Passing

- Processes interact by sending and receiving messages
  - Similar to pipes, but messages are isolated data chunks of specified size, rather than *unstructured streams* of bytes, as in pipes.
  - Sometimes say message passing is *connectionless*.
- As with pipes, problems of interference that arise when accessing shared data are avoided, *and the model works for communication between processes on different computers.*

Message Passing: Issues

- Problem: how many messages should be buffered temporarily during communication?
- Solutions:
  - Zero-capacity queues: 0 messages,
    - Sender *always waits for receiver* (synchronization is called *rendezvous*).
    - Sounds restrictive, but simplifies formal analysis of distributed programs (*CSP*, *CCS*, *Pi Calculus*, etc)
  - Bounded capacity queues: **finite** length of n messages,
    - Sender waits *if* link buffer is full (e.g. *MPI*).
  - Unbounded capacity queues: **infinite** queue length,
    - Sender *never waits*
    - Message queue in week 4 appendix, for example.

Message Passing: Implementations

- For general messaging there is the *Java Message Service* (JMS) API, implemented by various projects and vendors, e.g.:
  - Sun (Oracle) Java System Message Queue
  - BEA Weblogic
  - IBM WebSphere
- For *parallel computing* there is the *Message Passing Interface* (MPI), implemented by open source projects and hardware vendors.

UNIVERSITY OF PORTSMOUTH

Sockets

- *Sockets* provide a *programming model* with some features of message passing, though they are most commonly used for *stream-oriented* communication.
- In this respect they are similar to pipes. But unlike pipes, sockets can connect *unrelated* processes – including processes on *different* computers.

Socket API

- *Berkeley Sockets* API† implemented by *system calls* in Linux or Windows:

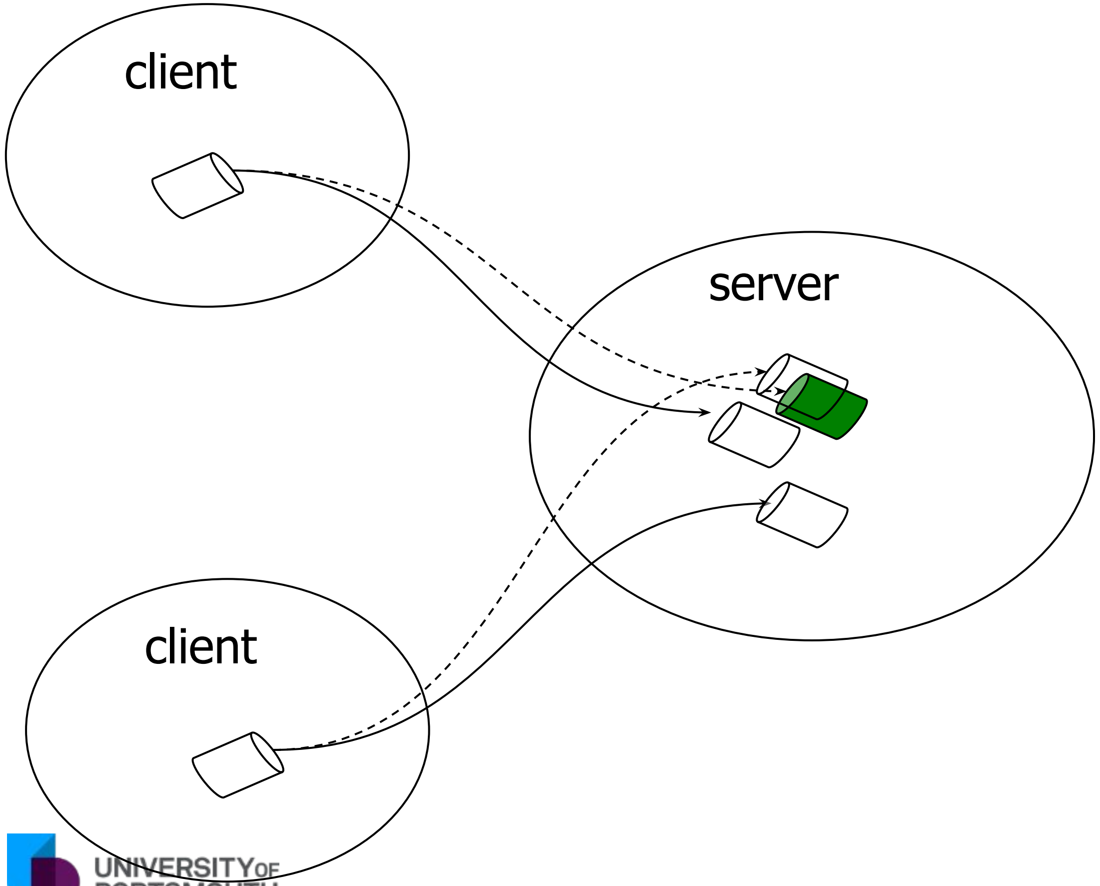
Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

†Tanenbaum, CN, Fig 6-5

Using Sockets

- After initializing a socket by `socket()`, `bind()`, `listen()`, “server” waits for connections on specified port by calling `accept()`.
- “Client” calls `connect()`, passing in a local socket and address of server socket, (IP address plus port number).
  - If connection succeeds, a new socket is returned by `accept()`.
  - Client and server exchange byte arrays of data over the socket pair using `send()` and `recv()` calls.

Scenario



1. Server creates socket (**socket** system call), binds it to an IP address and port (**bind**), and marks it as a server socket (**listen**)
2. Server does **accept**, to wait for connection from client.
3. Client creates socket (**socket**)...
4. ... and connects to IP address and port for server socket (**connect**).
5. A new socket is created on server connected to client socket; hosts communicate.
6. Server may do **accept** again, waiting for other connections...

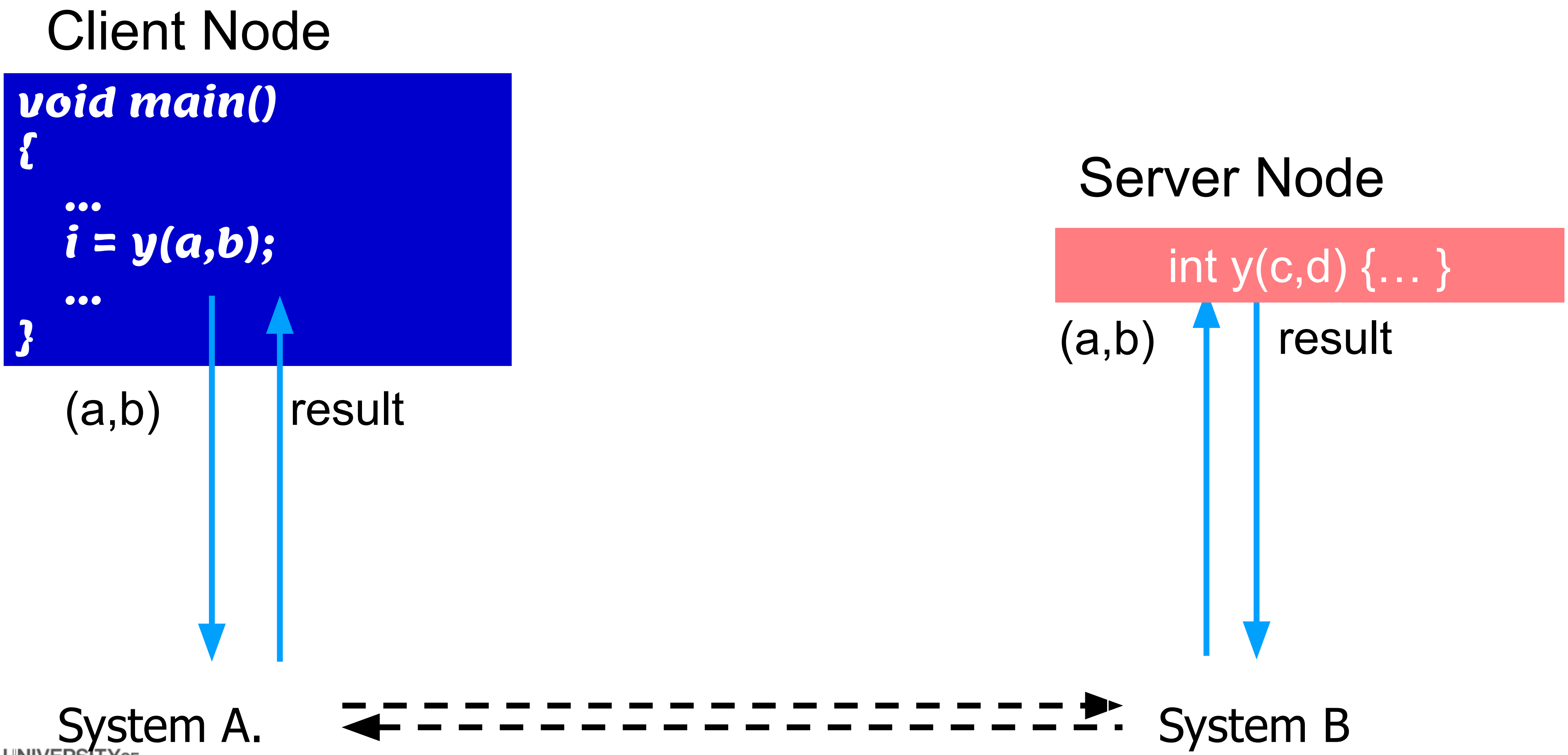
UNIVERSITY OF PORTSMOUTH

Remote Procedure Call (RPC)

- Suggested by Birell and Nelson in 1984.
- Goals:
  - Access-transparent call semantics.
  - Remote calls *look like* local procedure calls.
    - Require to convert calls into network messages.
- Server:
  - Exports modules of procedures (somehow).
- Clients:
  - Call these procedures (somehow).
  - Look as close as possible to local-procedure call, *but* really network messages.

UNIVERSITY OF PORTSMOUTH

How RPC Works?



Why RPC ?

- Extends the conventional procedure call to the client/server model.
- Remote procedures accept arguments and return results.
- Makes it easy to design and understand programs.
- Helps programmer to focus on the application instead of the communication protocol.
- Allows a client to execute procedures on other computers.
- Simplifies the task of writing client/server programs.

 RPC *forms the foundation for many distributed utilities* used today, like “Network File System” NFS and “Network Information Service” NIS in UNIX-based systems.

UNIVERSITY OF PORTSMOUTH

RPC: Issues

- Transparency:
  - Syntactic transparency: RPC should have *same syntax* as local procedure call.
  - Semantic transparency: RPC *semantics should be identical* to local procedure call.
- Standard representation: Need for external data representation (XDR) for all data types.
  - One machine may be *little-endian* and the other machine may be *big-endian*.
  - One machine may be using ASCII and the other UTF-16.
  - Representation of floating point numbers on the two machines may be different.

RPC: Implementations

- *Common Object Request Broker Architecture* (**CORBA**).
- Java *Remote Method Invocation* (**RMI**).
  - See next week’s lab
- *SOAP* (formerly *Simple Object Access Protocol*) Web services.
- *Windows Communication Foundation* (**WCF**).

Summary

- IPC mechanisms:
  - Shared Memory (memory based).
  - Pipe and File (stream based).
  - Message Passing (message based).
  - Sockets (commonly stream based).
  - RPC (procedure based).
- We’ve only scratched the surface of IPC.
- *Next Lecture – File Systems*

Single system

Different systems

Further Reading

- Andrew S. Tanenbaum and David J. Wetherall, "*Computer Networks*", 5th Edition, Pearson, 2013 (CN)
  - Sockets discussed section 6.1.3.

UNIVERSITY OF PORTSMOUTH

Questions?