CS 370: Operating Systems
[Inter Process Communications]

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Producer-consumer with bounded buffer
  - How is the size of the buffer enforced?
  - Do the variables `out` and `in` have to been in sync?
- `init` process: What if I try to kill it? What if that process crashes?
- Shared memory between `A` and `B`: Are the numeric values of the addresses the same?
  - Could you pick the answer as too high?
- Background processes: Are they somehow connected to the shell?
- Do you have to configure processes as independent or cooperative?
- Difference between terminal and shell

2

Topics covered in this lecture

- Inter Process Communications
  - Messaging
  - Pipes
- Monolithic Kernels and Micro kernels

Message Passing Buffer: Consumer always has to wait for message

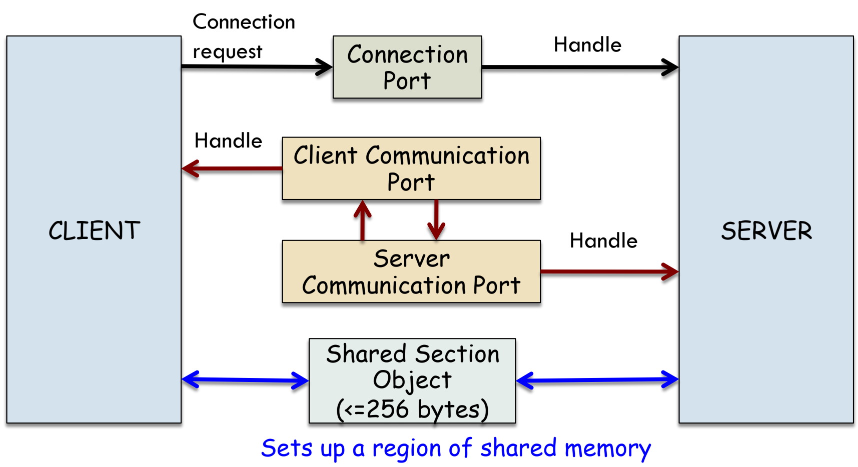
- ZERO capacity: No messages can reside in queue
  - Sender ***must block*** till recipient receives
- BOUNDED: At most **n** messages can reside in queue
  - Sender **blocks *only if* queue is *full***
- UNBOUNDED: Queue length potentially infinite
  - Sender ***never blocks***

MESSAGE PASSING IN WINDOWS XP

Message passing in Windows XP

- Called the local procedure call (LPC) facility
- Communications provided by **port** objects
  - Give applications a way to set up communication channels
- Uses two types of message passing
  - Small messages (max 256 bytes)
  - Large messages

Connection ports are named objects visible to all processes [LPC in XP]



Windows XP message passing
Small messages

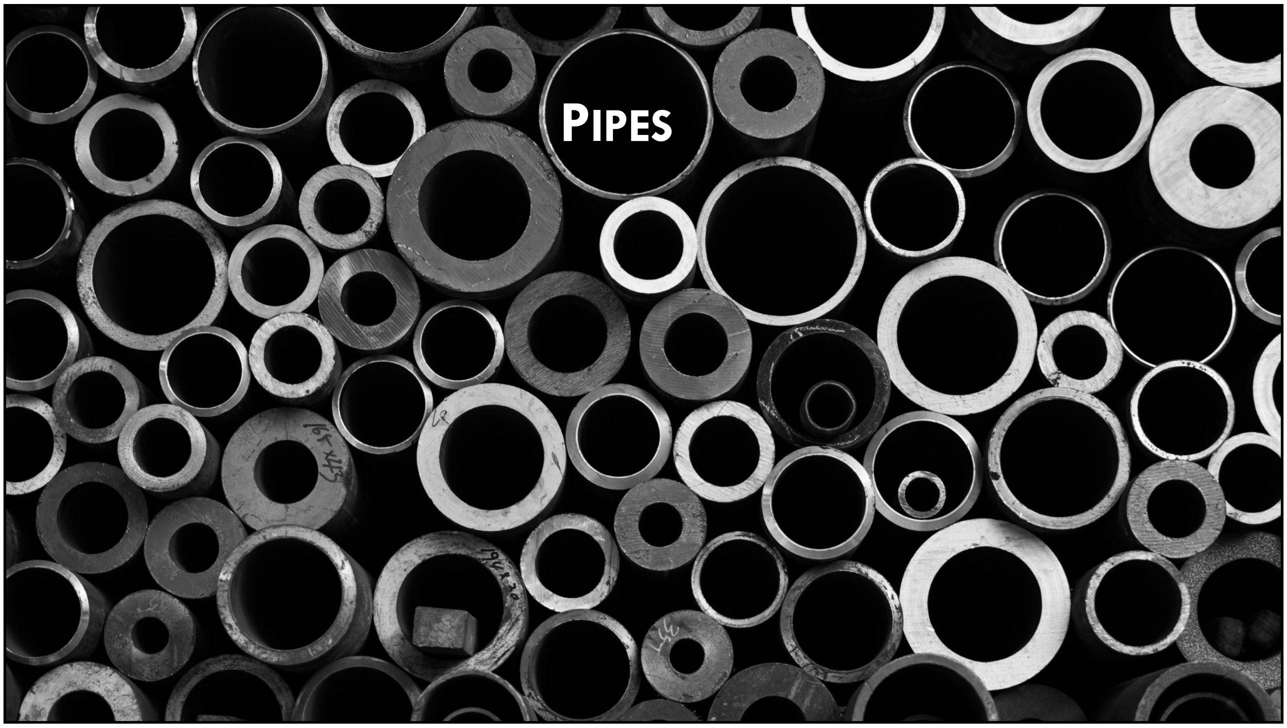
- Use port's internal message queue as intermediate storage
- Copy messages from one process to another

Windows XP message passing: Large messages [1/2]

- Send message through **section object**
  - Sets up shared memory
- Section object info sent as a small message
  - Contains pointer + size information about section object

Windows XP message passing: Large messages [2/2]

- 2 ends of communications set up section objects if the request or reply is large
- Complicated, but **avoids data copying**
- **Callbacks** used if the endpoints are busy
  - Allows delayed responses
  - Allows asynchronous message handling



11

Pipes

- Pipes serve as a **conduit** for communications between processes
- One of the first IPC implementation mechanisms

12

Issues to consider when implementing a pipe

- Unidirectional or bidirectional
- If it is bidirectional
  - **Half duplex**: Data can travel one way at a time
  - **Full duplex**: Data traversal in both directions *simultaneously*
- Must a relationship exist between the endpoints?
  - e.g., parent-child
- Range of communications
  - Intra-machine or Over the network

Pipes in practice

- Set up pipe between commands

```
ls | more
```

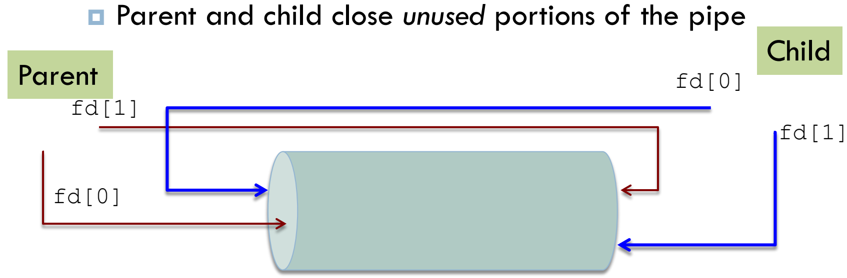
Output of `ls` delivered as input to **`more`**

Ordinary pipes

- Producer writes to one end of the pipe
- Consumer reads from the other end
- In UNIX: `pipe(int fd[])` to create pipe
  - `fd[0]` is the read-end
  - `fd[1]` is the write-end
  - Treats a pipe as a **special type of file**
    - Access with `read()` and `write()` system calls

A child inherits open files from its parent

- Since a pipe is a special type of file, the pipe is also inherited
  - Parent and child close *unused* portions of the pipe



`fd[0]` is the read-end
`fd[1]` is the write-end

Pipes: Example

```
if (pipe(fd) == -1) {
    /* creation failed */
}
pid = fork();

if (pid > 0) {
   close(fd[READ_END]);
   write(fd[WRITE_END], write_msg,…);
}

if (pid == 0) {
   close(fd[WRITE_END]);
   read(fd[READ_END], …);
}
```


Windows Ordinary Pipes: These are unidirectional

- Anonymous Pipes
- Child **does not** automatically inherit pipe
  - Programmer *specifies* attributes a child will inherit
  - Initialize `SECURITY_ATTRIBUTES` to allow handles to be inherited
  - Redirect child's standard I/O handles to read/write handle of pipe
  - Pipes are half duplex

Some other things about ordinary pipes on UNIX and Windows

- Requires **parent-child** relationship
  - MUST be on same machine
- **Exist** only when processes communicate with one another
  - Upon termination, pipe ceases to exist

Named Pipes

- Can be bidirectional
- **NO** parent-child relationship needed
- Once named pipe is established
  - *Several* processes can use it for communications
- Continues to exist after communicating processes have finished

Named Pipes on UNIX/Windows

- Referred to as **FIFO** on UNIX systems
  - Created with `mkfifo()`
  - Manipulated with `open()`, `read()`, `write()` etc.
- FIFO: Bidirectional but **half-duplex** transmissions
  - If data must go both ways: use 2 FIFOs
  - Sockets used for inter-machine communications
- Windows: Full duplex communications

COMMUNICATIONS IN CLIENT-SERVER SYSTEMS

Remote Procedure Calls

- Abstracts procedure call mechanisms for use with network endpoints
- Based on the **request/reply** model
- Message is addressed to the **RPC daemon** listening to a port for incoming traffic
  - Contains identifiers of function to execute
  - Parameters to pass to the function
  - TCP/UDP port number: 530
    - Other example ports: DNS(53), HTTP(80), NTP(123), etc.

Remote Procedure Calls

- Application makes CALL into a procedure
  - May be local or remote **and**
  - BLOCKS until call returns
- Origins:
  - **RFC 707** (1976)
  - First use by Xerox 1981 (Courier)
  - 1984 paper by Birell and Nelson

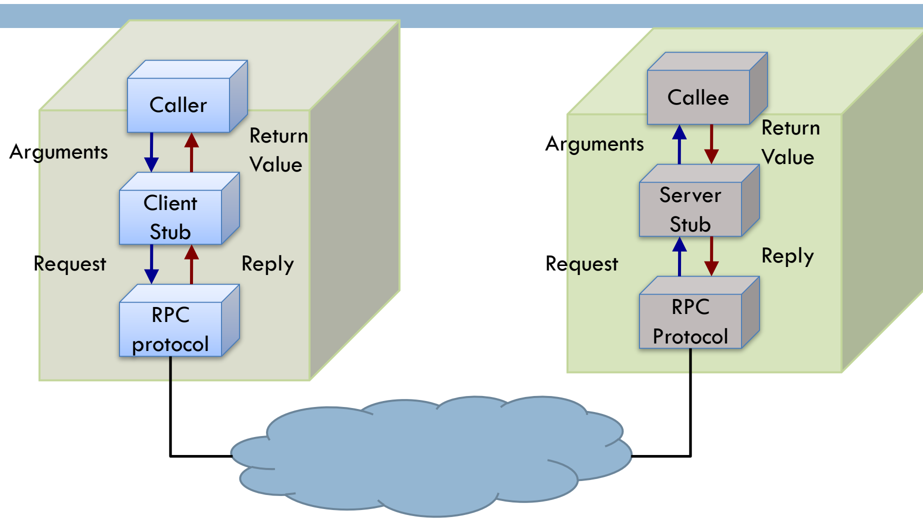
RPCs are slightly more complicated than local procedure calls

- Network between the *Calling* process and *Called* process can
  - **Limit** message sizes,
  - **Reorder** them or
  - **Lose** them
- Computers hosting processes may differ
  - Architectures and data representation formats

Resolving big-endian/little endian issues

- Big endian: Store **MSB** first
- Little endian: Store **LSB** first
- Machine independent data representation
  - XDR: e**X**ternal **D**ata **R**epresentation
  - Client side parameter marshalling
    - Convert machine-dependent data to XDR
  - Server side
    - Convert XDR data to machine dependent representation

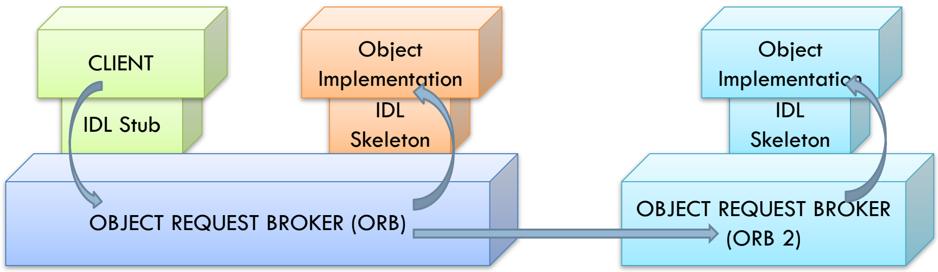
RPC mechanism



Distributed Objects

- RPC based on distributed objects with an **inheritance** mechanism
- *Create*, *invoke* or *destroy* remote objects, and interact as if they are local objects
- Data sent over network:
  - **References**: class, object and method
  - Method *arguments*
- CORBA early1990s, RMI mid-late 90s

Distributed Objects in CORBA defined using the Interface Definition Language



GIOP/IIOP
General Inter-ORB Protocol/Internet Inter-Orb Protocol

29

30

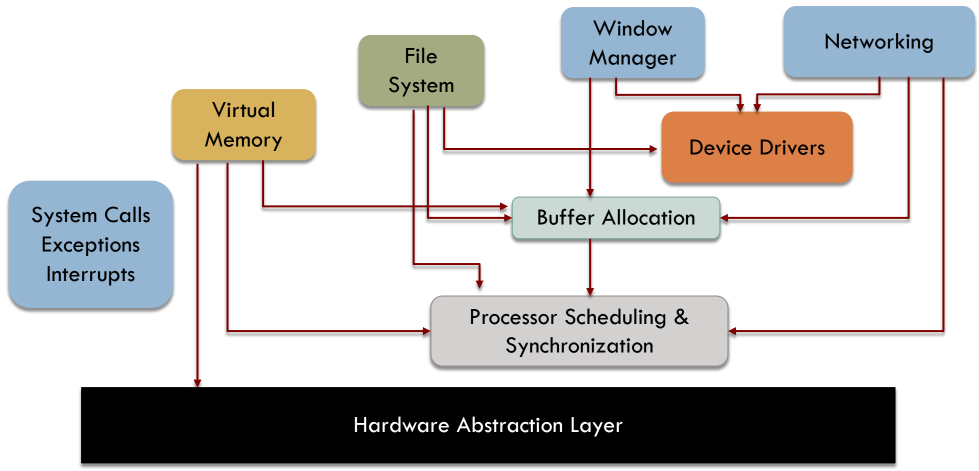
There are many dependencies (and interactions) among the modules inside the OS [1/2]

- Several modules depend on **synchronization primitives** for coordinating access to shared data structures with the kernel
- The virtual memory system depends on low-level hardware support for address translation
  - Support that is specific to a particular processor architecture

There are many dependencies (and interactions) among the modules inside the OS [2/2]

- Both the **file system** and the **virtual memory** system share a common pool of blocks of physical memory
  - They also both depend on the disk device driver
- The **file system** can depend on the **network protocol stack** if the disk is physically located on a different machine

OS Kernel Structure



This has led operating system designers to wrestle with a fundamental tradeoff

- By **centralizing functionality** in the kernel, performance is improved
  - It makes it easier to arrange tight integration between kernel modules
- However, the resulting systems are less flexible, less easy to change, and less adaptive to user or application needs

? Prevalence of monolithic kernels?

Where is the monolithic kernel used?

- **Almost all** widely used commercial operating systems, such as Windows, MacOS, and Linux
- Monolithic is a bit of a misnomer, there are often large segments of what users consider the OS that runs outside the kernel
  - Either as utilities like the shell, or in system libraries, such as libraries to manage the user interface

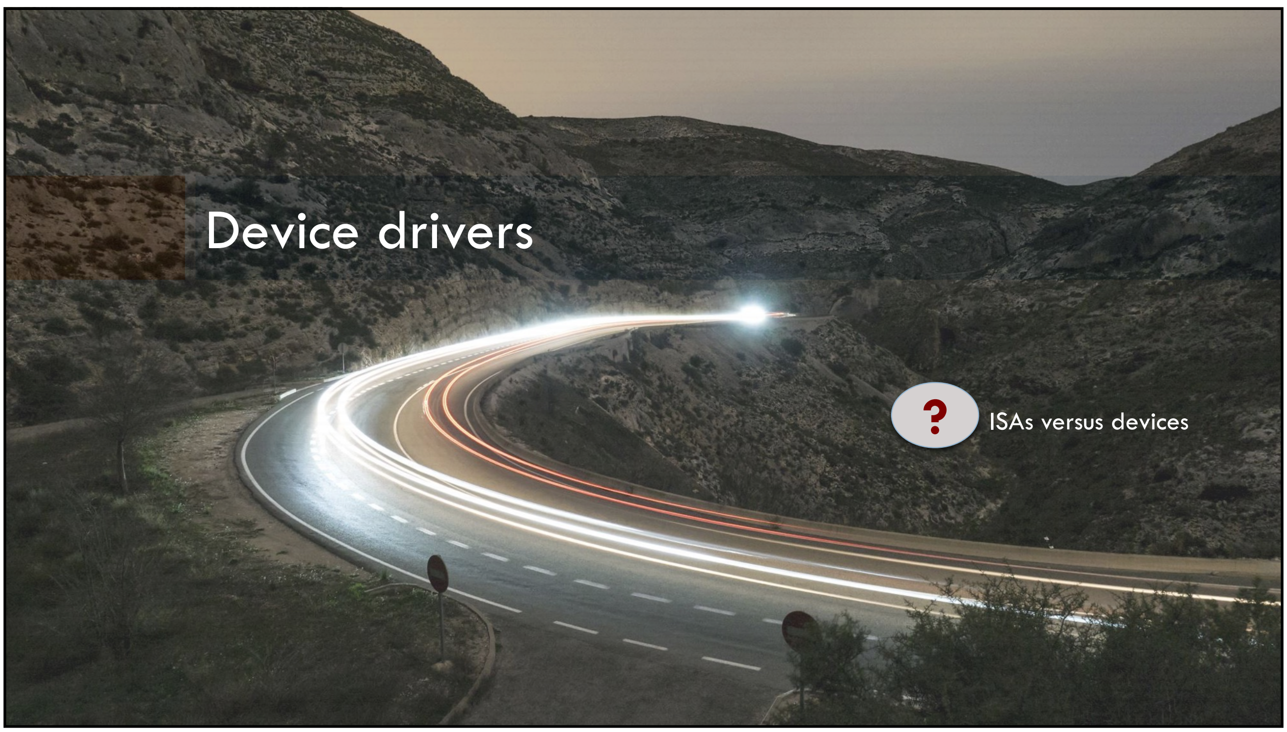
A key goal of operating systems is to be portable across a wide variety of hardware platforms [1/2]

- This requires careful design of the hardware abstraction layer
- **Portable interface** to machine configuration and processor-specific operations within the kernel

A key goal of operating systems is to be portable across a wide variety of hardware platforms [2/2]

- Within the same processor family, such as an Intel x86
  - Manufacturers use different machine-specific code to configure and manage interrupts and hardware timers
  - x86-based Mac vs x86-based Windows
- Across processor families, will need processor-specific code for process and thread context switches
  - Between an ARM and an x86 or
  - Between a 32-bit and a 64-bit x86

37



38

Operating Systems try to accommodate a wide variety of physical I/O devices

- There are **only a handful** of different instruction set architectures (ISA) in wide use today:
  - But there are a huge number of different types of physical I/O devices, manufactured by a large number of companies
  - There is diversity in the hardware interfaces to devices as well as in the hardware chip sets for managing the devices

What is a dynamically installed device driver?

- Software to manage a specific device, interface, or chipset, that is added to the kernel **after it starts running**
  - To handle the devices that are present on a particular machine
- The device manufacturer typically provides the **driver code**, using a standard interface supported by the kernel
  - The kernel calls into the driver whenever it needs to read or write data to the device

Dynamically Installed Device Drivers

- A recent survey found that approximately 70% of the code in the Linux kernel was in device-specific software
- However, 90% of all system crashes were due to bugs in device drivers, rather than in the operating system itself

41

42

The Microkernel Approach [1/2]

- An alternative to the monolithic kernel approach is to run as much of the OS as possible in one or more user-level servers/services
- Structure OS by *removing non-essential components* from the kernel
  - Implement other things as system/user programs
  - **Mach**
- Provide minimal process and memory management
- Main function: Provide communication facility between client and services
  - **Message passing**

The Microkernel Approach [2/2]

- In monolithic kernels all the layers went in the kernel
  - But this is not really necessary
- In fact, it may be best to *put as little as possible* in the kernel
  - Bugs in the kernel can bring down the system instantly
- Contrast this with setting up user processes to have less power
  - A bug may not be fatal

Getting there …

- Achieve high reliability by **splitting OS into small, well-defined modules**
  - The microkernel runs in the kernel mode
  - The rest as relatively powerless ordinary user processes
- Running each device driver as a separate process?
  - Bugs cannot crash the entire system

Communications in the microkernel

- Client and service/server never interact directly
- Indirect communications by exchanging messages with the microkernel
- Advantages
  - Easier to port to different hardware
  - More security and reliability
    - Most services run as user, rather than kernel

Some examples

- The window manager on most operating systems works this way:
  - Individual applications draw items on their portion of the screen by sending requests to the window manager
  - The **window manager adjudicates** which application window is in front or in back for each pixel on the screen, and then renders the result
  - If the system has a hardware graphics accelerator present, the window manager can use it to render items more quickly
- Some systems have moved other parts of the operating system into user-level servers: the network stack, the file system, device drivers, and so forth

Another idea related to microkernels

- Put **mechanisms** for doing something in the *kernel*
  - But <u>not the policy</u>
- Example: Scheduling
  - Policy of assigning priorities to processes can be done in the user-mode
  - The mechanism to look for the highest priority process and to schedule it is in the kernel

The difference between monolithic/microkernel design is often transparent to the programmer

- The location of the service can be hidden in a user-level library
  - Calls go to the library, which casts the requests either as
    - System calls or
    - Reads and writes to the server through a pipe
- The location of the service can also be hidden inside the kernel
  - The application calls the kernel as if the kernel implements the service
    - But instead, the kernel reformats the request into a pipe that the service can read

Increased system function overhead can degrade microkernel performance

- A microkernel design offers considerable benefit to the operating system developer
  - Easier to modularize and debug user-level services than kernel code
- Aside from a potential reliability improvement, however, microkernels offer little in the way of visible benefit to end users and can **slow down overall performance considerably**
  - By inserting extra steps between the application and the services it needs

A hybrid model

- Some operating system services are run at user-level and some are in the kernel
  - Depending on the specific tradeoff between code complexity and performance

Increased system function overhead can degrade microkernel performance

- Windows NT: First release, layered microkernel
  - Lower performance than Windows 95
- Windows NT 4.0 solution
  - Move layers from user space to kernel space
- By the time Windows XP came around
  - More monolithic than microkernel

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 3]
- *Thomas Anderson and Michael Dahlin. Operating Systems: Principles and Practice, 2nd Edition. Recursive Books. ISBN: 0985673524/978-0985673529. [Chapters 2-3]*
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2]
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 3, 4]