CS 370: Operating Systems
[Inter Process Communications]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- shmget(key_t key, size_t size, int shmflg)
  - Returns the identifier of the shared memory segment
- Shared memory:
  - How many processes can be involved?
  - Does each process get its own memory reference to it?
  - Is its size fixed?
- Modularity with processes and how does it help?
- Messages: what are they? How do they differ from processes?
- Mailboxes: How do you know when its been added to the mailbox?

Topics covered in this lecture

- Inter Process Communications
  - Messaging
  - Pipes

Message passing: Synchronization issues Options for implementing primitives

- Blocking send
  - Block *until* received by process or mailbox
- Nonblocking send
  - Send and *promptly resume* other operations
- Blocking receive
  - Block *until* message available
- Nonblocking receive
  - Retrieve *valid* message or *null*
- Producer-Consumer problem: Easy with blocking

Message Passing: Buffering

- Messages exchanged by communicating processes reside in a **temporary** queue
- Implementation schemes for queues
  - ZERO Capacity
  - Bounded
  - Unbounded

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***

MICROKERNELS

The Microkernel Approach [1/2]

- Mid 1980's at Carnegie Mellon University
  - **Mach**
- Structure OS by *removing non-essential components* from the kernel
  - Implement other things as system/user programs
- Provide minimal process and memory management
- Main function: Provide communication facility between client and services
  - **Message passing**

The Microkernel Approach [2/2]

- Traditionally 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 in 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 micro-kernel

- Client and service 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
- **Mac OS X kernel based on Mach microkernel**
  - XNU: 2.5 Mach, 4.3 BSD and Objective-C for device drivers

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

IPC communications: Mach

- Tasks are similar to processes
  - Multiple threads of control
- Most communications in Mach use **messages**
  - System calls
  - Inter-task information
  - Sent and received from mailboxes: *ports*

Mach: Task creation and mailboxes

- Task creation results in 2 more mailboxes
  1. Kernel mailbox: Used by kernel to communicate with task
  2. Notify mailbox: Notification of event occurrences
- System calls for communications
  - `msg_send()`, `msg_receive()` and `msg_rpc()`

Mach: Mailbox creation

- Done using the `port_allocate()`
  - Allocate space for message queue
    - MAX_SIZE default is 8 messages
- Creator is owner and can also receive
- Only task can own/receive from mailbox
  - BUT these **rights can be sent** to other tasks

Mach: Message queue ordering

- FIFO guarantees for messages from same sender
- Messages from multiple senders queued in any order

Mach: Send and receive operations

- If mailbox is not full, copy message
- If mailbox is FULL
  1. Wait indefinitely till there's room
  2. Wait at most **n** milliseconds
     - Don't wait, simply return
  3. Temporarily cache the message
     - **Only 1** message to a full mailbox can be *pending* for a *given* sending thread
- Receive can specify mailbox or mailbox set

Another idea related to microkernels

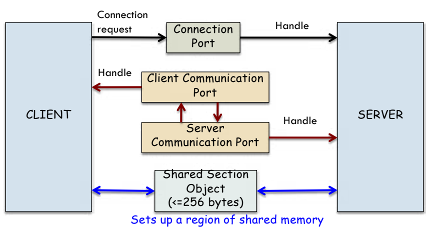
- Put **mechanisms** for doing something in the *kernel*
  - But not the policy
- Example: Scheduling
  - Policy of assigning priorities to processes can be done in the user-mode
  - The mechanism to look for the highest priority process and to schedule it is in the kernel

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 <u>small message</u>
  - 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

PIPES

Pipes

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

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`

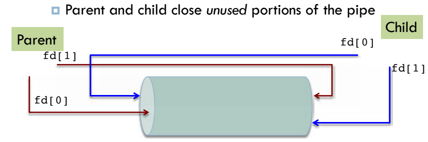
Output of **`ls`** delivered as input to **`more`**

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

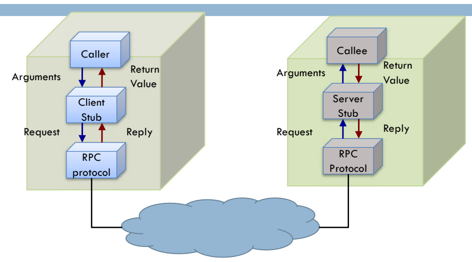
RPCs are slightly more complicated than local procedure calls

- 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

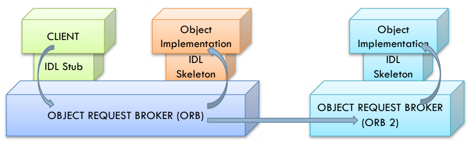
RPC mechanism



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

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]
- *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]