CS 370: OPERATING SYSTEMS

[INTER PROCESS COMMUNICATIONS]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALICKARA
Topics covered in this lecture

- Briefly, microkernels
- Inter Process Communications
  - Messaging
  - Pipes
- Threads
MICROKERNELS
The Microkernel Approach

- 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
  - One of these, 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 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

- 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 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 (anonymous) 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

```
Parent
fd[0]
fd[1]

Child
fd[0]
fd[1]
```

- `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

- **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, manipulated like a file
  - Created with `mkfifo()`
  - Manipulated with `open()`, `read()`, `write()` etc

- **FIFO**: **half-duplex** transmissions on Linux
  - If data must go both ways: use 2 FIFOs
  - Sockets can be 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
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: eXternal Data Representation
  - Client side parameter marshalling
    - Convert machine-dependent data to XDR
  - Server side
    - Convert XDR data to machine dependent representation
RPC mechanism

- Caller
  - Client Stub
  - RPC protocol
  - Arguments
  - Request
  - Reply
  - Return Value

- Callee
  - Server Stub
  - RPC Protocol
  - Arguments
  - Request
  - Reply
  - Return Value
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 early 1990s, RMI mid-late 90s
Distributed Objects in CORBA defined using the Interface Definition Language

GIOP/IIOP
General Inter-ORB Protocol/Internet Inter-Orb Protocol
THREADS
Some background on threading

- Exploited to make programs easier to write
  - Split programs into separate tasks

- Took off when GUls became standard
  - User perceives better performance
    - Programs did not run faster: this was an illusion
    - Dedicated thread to service input OR display output

- Growing trend to exploit available processors on a machine
What are threads?

- Miniproceses or lightweight processes
- Why would anyone want to have a kind of process within a process?
The main reason for using threads

- In many applications *multiple activities* are going on at once
  - Some of these may block from time to time

- Decompose application into multiple sequential threads
  - Running in quasi-parallel
Isn’t this precisely the argument for processes?

- Yes, but there is a new dimension …

- Threads have the ability to share the address space (and all of its data) among themselves

- For several applications
  - Processes (with their separate address spaces) don’t work
Threads are also lighter weight than processes

- **Faster** to create and destroy than processes
- In many systems thread creation is 10-100 times faster
- When number of threads needed changes dynamically and rapidly?
  - Lightweight property is very useful
Threads:
The performance argument

- When all threads are CPU bound all the time?
  - Additional threads would likely yield no performance gain

- But when there is substantial computing and substantial I/O
  - Having threads allows activities to overlap
  - Speeds up the application possibly
AN EXAMPLE APPLICATION
WORD PROCESSOR
Our Word Processor

- Displays document being created on the screen
- Document formatted exactly as it will appear on a printed page
Let’s take a look at someone editing a 800-page document

- User deletes one sentence from Page-1 of a 800-page document

- Now user wants to make a change on page 600
  - Either go to that page or search for term that only appears there
- Word processor *does not know* what’s the first line on page 600
- Word processor has to **reformat** entire book up to page 600
- Threads could help here …
Suppose the word processor is written as a 2-threaded program

- One thread **interacts** with the user
- The second thread handles **formatting** in the background
- As soon as the sentence is deleted
  - Interactive thread tells formatter thread to format the book
While we are at it, why not add a third thread?

- Automatically save file every few minutes
- Handle disk backups *without interfering* with the other 2 threads
What if the program were single threaded?

-Whenever disk backup started
  - Commands from keyboard/mouse would be **ignored** till backup was finished
  - User perceives sluggish performance

-Alternatively, keyboard/mouse events could **interrupt** the disk backup
  - Good performance
  - Complex, interrupt-driven programming
With 3 threads the programming model is simpler

- First thread **interacts** with the user
- Second thread **reformats** when told to
- Third thread **writes** contents of RAM on to disk periodically
Three separate processes WOULD work here

- All three threads need to operate on document

- By having 3 threads instead of 3 processes
  1. The threads share a common memory
  2. Have access to document being edited

- Using processes would require setting up shared memory space, synchronizations, IPC etc. Doable, but much more tedious
  - Tend to use threads when working on the same data within the process
Applications are typically implemented as a process with multiple threads of control

- Perform different tasks in the application
  - Web browser
    - Thread A: Render images and text
    - Thread B: Fetch network data

- Assist in the performance of several similar tasks
  - Web Server: Manages requests for web content
    - Single threaded model: One client at a time
      - Poor response times
    - Multithreaded model: Multiple clients served concurrently
The contents of this slide-set are based on the following references

