IPC

Dave Eckhardt
de0u@andrew.cmu.edu

P3 interlude

- Don't forget about Chapter 4
- What if a multi-threaded process calls fork()?
  - You do ***not*** need to “copy all the threads”
    - (What could make this impossible?)
  - Copy the whole memory image
  - Copy the fork()ing thread

P3 interlude

- What if a multi-threaded process calls exec()?
  - Several reasonable answers
  - What is ***not*** a reasonable answer?
  - Design & document “something reasonable”

Outline

Scope of “IPC”

- Communicating process on one machine
- Multiple machines?
  - Virtualize single-machine IPC
  - Switch to a “network” model
    - Failures happen
    - Administrative domain switch
    - ...

IPC parts

- Naming
- Synchronization/buffering
- Copy/reference/size

Naming

- Message sent to ***process*** or to ***mailbox***?
- Process model
  - send(P, msg)
  - receive(Q, &msg) or receive(&id, &msg)
  - No need to set up “communication link”
    - But you need to know process id's
    - You get only one “link” per process pair

Naming

- Mailbox model
  - send(box1, msg)
  - receive(box1, &msg) or receive(&box, &msg)
- Where do mailbox id's come from?
  - “name server” approach
    - box = createmailbox();
    - register(box1, “Terry's process”);
    - boxT = lookup(“Terry's process”);
- File system approach

Multiple Senders

- Problem
  - Receiver needs to know who sent request
- Typical solution
  - “Message” not just a byte array
  - OS imposes structure
    - sender id (maybe process id and mailbox id)
    - maybe: type, priority, ...

Multiple Receivers

- Problem
  - Service may be “multi-threaded”
  - Multiple receives posted to one mailbox
- Typical solution
  - OS “arbitrarily” chooses receiver per message
    - (Can you guess how?)

Synchronization

- Issue
  - Does communication imply synchronization?
- Blocking send()?
  - Ok for request/response pattern
  - Provides assurance of message delivery
  - Bad for producer/consumer pattern
- Non-blocking send()?
  - Raises buffering issue (below)

Synchronization

- Blocking receive()?
  - Ok/good for “server thread”
    - Remember, de-scheduling is a kernel ***service***
  - Ok/good for request/response pattern
  - Awkward for some servers
    - Abort connection when client is “too idle”
- Pure-non-blocking receive?
  - Ok for polling
  - Polling is costly

Synchronization

- Receive-with-timeout
  - Wait for message
  - Abort if timeout expires
  - Can be good for real-time systems
  - What timeout value is appropriate?

Synchronization

- Meta-receive
  - Specify a group of mailboxes
  - Wake up on first message
- Receive-scan
  - Specify list of mailboxes, timeout
  - OS indicates which mailbox(es) are “ready”
  - Unix: select(), poll()

Buffering

- Issue
  - How much “free space” does OS provide?
  - “Kernel memory” limited
- Options
  - No buffering
    - implies blocking send
  - Fixed size, undefined size
    - send may or may not block

A buffering problem

- P1
  - send(P2, p1-my-status)
  - receive(P2, &p1-peer-status)
- P2
  - send(P1, p2-my-status)
  - receive(P1, &p2-peer-status)
- What's the problem?

Copy/reference/size

- Issue
  - Ok to copy small messages sender -> receiver
  - Bad to copy 1-megabyte messages
  - “Chop up large messages” evades the issue
- “Out of line” message part
  - Page-aligned, multiple-page memory regions
  - Can ***transfer ownership*** to receiver
  - Can share the physical memory
    - Mooooo!

Rendezvous

- Concept
  - Blocking send
  - Blocking receive

Example: Mach IPC

- Why study Mach?
  - “Pure” “clean” capability/message-passing system
  - Low abstraction count
  - This ***is*** CMU...
- Why not?
  - Failed to reach market
  - Performance problems with multi-server approach?
- Verdict: hmm... (GNU Hurd?)

Mach IPC – ports

- Port: Mach “mailbox” object
  - One receiver
  - One “backup” receiver
  - Potentially many senders
- Ports identify system objects
  - Each task identified/controlled by a port
  - Each ***thread*** identified/controlled by a port
  - Kernel exceptions delivered to “exception port”

Mach IPC – port rights

- Receive rights
  - “Receive end” of a port
  - Held by one task
  - Capability typically unpublished
    - receive rights imply ownership
- Send rights
  - “Send end” - ability to transmit message to mailbox
  - Frequently published via “name server” task
  - Confer no rights (beyond “denial of service”)

Mach IPC – message

- Memory region
  - In-line for “small” messages (copied)
  - Out-of-line for “large” messages
    - Sender may de-allocate on send
    - Otherwise, copy-on-write
- Port rights
  - Sender specifies task-local port #
  - OS translates to internal port-id while queued
  - Receiver observes task-local port #

Mach IPC – operations

- send
  - block, block(n milliseconds), don't-block
  - “send just one”
    - when destination full, queue 1 message in ***sender thread***
    - sender notified when transfer completes
- receive
  - receive from port
  - receive from ***port set***
  - block, block(n milliseconds), don't-block

Mach IPC – RPC

- Common pattern: “Remote” Procedure Call
- Client synchronization/message flow
  - Blocking send, blocking receive
- Client must allow server to respond
  - Transfer “send rights” in message
    - “Send-once rights” speed hack
- Server message flow (N threads)
  - Blocking receive, non-blocking send

Mach IPC – naming

- Port send rights are OS-managed capabilities
  - unguessable, unforgeable
- How to contact a server?
  - Ask the name server task
    - ***Trusted*** – source of all capabilities
- How to contact the name server?
  - Task creator specifies name server for new task
    - Can create custom environment for task tree

Summary

- Naming
  - Name server?
  - File system?
- Queueing/blocking
- Copy/share/transfer
- A Unix surprise
  - sendmsg()/recvmsg() pass file descriptors!