Overview

- Introduction
- Communication in monolithic systems
  - UNIX
- Communication in multi-server systems
  - Mach
  - L4
  - VMs
Why communicate?

- Synchronization
- Data Transfer
- Control Transfer

I’ll be home at 6!

Ok, I’ll leave at 5!

If you go shopping, could you please bring:

- Food for dinner
- Soft drinks
- Pampers

Can you cook the kids’ dinner tonight, please?

Got it!

Can you cook the kids’ dinner tonight, please?

Synchronization

Data Transfer

Control Transfer
How communicate?

- Data transfer
  - Shared data
    - Shared memory, storage, registers, …
    - Implicit, unsynchronized communication
  - Messages
    - Send, receive
    - Explicit communication

- Control transfer
  - Procedure call: call/ret
  - Safe procedure call: sysenter/sysexit, int 0x80/iret
  - Remote Procedure call: RPC
  - Based on
    - Shared code segments
    - Special primitives
Communication principles

- **Addressing modes:**
  - Addressing senders/receivers:
    - Semaphores, sockets
  - Addressing communication facility
    - Pipes, mailboxes, …
  - Unicast, Anycast, Multicast, Broadcast

- **Design Considerations:**
  - Complexity
  - Communication model
    - Dispatcher/worker
    - Mailing list
    - Telephone
Communication principles

- Message transfer
  - Copying
  - By reference
    - Shared memory / storage
    - Using memory mapping techniques

- Design considerations
  - Pass-by-reference requires shared storage
  - Copying incurs overhead (time and space)
  - Memory mapping as well (modifying pagetable mappings)
  - Copying may be required to ensure integrity
    - Kernel can’t checksum network packets in user-accessible memory
    - Can’t pass references to untrusted clients
Communication principles

- Synchronous or not?
  - Synchronous
    - All parties are involved *at the same time*
    - Telephone, RPC
  - Asynchronous
    - Senders submit messages, receiver processes them some other time
    - Mail, message queues

- Design considerations:
  - Complexity:
    - Asynchronous communication may require buffering
  - Availability of communication partners:
    - No need for e-mail, I’m always reachable via mobile
Communication principles

- Blocking vs. non-blocking communication:
  - Blocking
    - Sender may block until message has been delivered / received / processed
    - Receiver may block until message has arrived
    - Enable synchronous communication if partners not always ready
    - Use timeouts to signal errors
Communication principles

- Blocking vs. non-blocking communication:
  - Non-blocking
    - Sender returns after submission
    - Receiver returns independent of message status
    - Can use interrupts for signaling completion/message availability

- Design considerations:
  - Importance of temporal synchronization with partners
    - Can I continue without a message having been processed?
  - Communication latency
    - How long does sending a message take?
    - Can I overlap I/O processing with other computation?
Communication principles

- Buffered vs. unbuffered:
  - Communication facility provides buffers for
    - Asynchronous, non-blocking communication
    - Lossy communication channels

- Design considerations
  - Importance of preserving (temporarily) undeliverable messages
    - I don’t care if you’re not online to see the video broadcast
  - Availability of buffer space
  - Buffer management
    - E.g.: Underflow/overflow handling -- Block sender/receiver? Return an error?
Communication in Monolithic Operating Systems
Communication in Monolithic OSes

- Vertical communication
  - User/kernel
    - Safety requirements
      - No arbitrary control transfer
      - No arbitrary data sharing
      - Need well-defined semantics
    - Partially shared data
      - Kernel hidden from user, but not vice-versa
    - Safe procedure call
      - System call/return
        - kernel→user, user→kernel
        - Shared argument data, but no shared code
      - Asynchronous control transfer
        - Signals, callbacks, ...
        - kernel→user only
        - shared code
    - Implicit addressing of kernel
Communication in Monolithic OSes

- **Vertical communication**
  - **Kernel/devices**
    - Safety requirements ignored, kernel is trusted
    - (Partially) shared data
      - I/O registers, DMA, ...
    - Control transfer
      - in, out, ...
      - kernel-*device*
  - **Interrupts**
    - Device-*kernel*
Communication in Monolithic OSes

- Horizontal communication
  - User/user
    - Safety requirements
    - No arbitrary data sharing
    - Kernel-provided abstractions
      - Messages
        - pipes, fifos, ipc, sockets, ...
      - Shared data
        - (Explicitly) shared memory
  - Kernel/kernel
    - No safety requirements, kernel is trusted
    - Shared data (global state and knowledge)
    - Direct control transfer (call/ret)
Communication primitives in UNIX

- **Example: Pipe**
  - Linear, byte-oriented, unidirectional data stream
  - Asynchronous, buffered communication
    - Ring-buffer of fixed size
    - Atomic write for data < buffer size
  - Blocking and non-blocking variants
    - Can block on full/empty buffer
  - Addressing: process-local handle
    - local file handle
    - passed to child/parent processes
Communication primitives in UNIX

- FIFO
  - Same semantics as pipe
  - Global and persistent communication channel
  - Addressed via global handle
  - Mapped to file system
  - open()/close() for creating and removing FIFOs
  - Manage access rights via file system permissions
Communication primitives in UNIX

- **Reading from a pipe/FIFO**

<table>
<thead>
<tr>
<th>pipe size $p$</th>
<th>At least one writing process</th>
<th>no writing process</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>blocking read</td>
<td>non-blocking read</td>
</tr>
<tr>
<td></td>
<td>sleeping writer</td>
<td>no sleeping writer</td>
</tr>
</tbody>
</table>

- $p = 0$
  - Copy $n$ bytes, return $n$; wait on empty buffer
  - Wait for some data, copy it, return its size
  - Return -EAGAIN
  - Return 0

- $0 < p < n$
  - Copy $p$ bytes, return $p$

- $p \geq n$
  - Copy $n$ bytes, return $n$; ($p-n$ bytes will be left in the buffer)

- **Writing to a pipe/FIFO**

<table>
<thead>
<tr>
<th>available buffer space $u$</th>
<th>at least one reading process</th>
<th>no reading process</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>blocking write</td>
<td>nonblocking write</td>
</tr>
<tr>
<td>$u &lt; n \leq$ bufsize</td>
<td>Wait until $n-u$ bytes freed, copy $n$ bytes, return $n$.</td>
<td>Return -EAGAIN</td>
</tr>
<tr>
<td>$n &gt;$ bufsize</td>
<td>if $u &gt; 0$, copy $u$ bytes, return; else return -EAGAIN</td>
<td>Send SIGPIPE signal and return -EPIPE</td>
</tr>
<tr>
<td>$u \geq n$</td>
<td>copy $n$ bytes and return $n$.</td>
<td></td>
</tr>
</tbody>
</table>

Communication primitives in UNIX

- System V IPC
  - Set of communication mechanisms for
    - Synchronization: *semaphores*
    - Message passing: *message queues*
    - Sharing memory: *IPC shared memory*
  - Addressed via global and local handles
    - 32-bit ipc key
      - Chosen by programmer
      - User-defined namespace
      - Passed around
    - 32-bit ipc reference identifier
      - Chosen and used by kernel
      - Unique in the system
      - (Can be passed around as well)
Communication primitives in UNIX

- IPC Semaphores
  - Synchronization via “atomic counting”
    - Consists of one or more primitive semaphores
    - Atomically incremented/decremented variables
  - Operations
    - semget(key): creates/resolves semaphore
    - semctl(id): initialize/force/destroy semaphore
    - semop(id, sem_op[]): operations on primitive semaphores
      - All operations performed atomically and simultaneously
      - atomic decrement (enter)
      - atomic increment (leave)
      - wait for zero
      - non-blocking and blocking variants
  - Undoable semaphores
    - Undo reservations of dying processes
    - Process can mark operations as undoable
    - Kernel tracks operations
    - Requires per-process list of semaphores
Communication primitives in UNIX

- IPC Message queues
  - Asynchronous, buffered messages
    - Requires user-kernel copying
  - Typed, FIFO-ordered message queues
    - Maximum number of queues
    - Maximum size of queue (16K)
    - Maximum size of messages (8K)

- Operations
  - `msgget(key)`: create/resolve message queue
  - `msgsend(id)`
  - `msgreceive(id)`
    - Blocking and non-blocking variants
      (on full/empty queues)

- Implementation
  - Linked list of messages
  - Messages are broken into sub-pages
Communication primitives in UNIX

- IPC Shared Memory
  - Memory segment shared between processes
  - May be used for asynchronous, implicit communication

- Kernel-provided memory buffers
  - Maximum number (4096)
  - Maximum segment size (8 MB)
  - Maximum total size (8 GB)

- Operations
  - shmget(key): create/resolve shared segment
  - shmat(id, [address]): attach segment
  - shmdt(id): detach segment from process

- Implementation
  - Via virtual memory subsystem
    (upcoming lecture)
Communication in Multi-Server Systems
Communication in Multi-Server Systems

- Kernel split into
  - Multiple kernel subsystems (servers) at user-level
    - Pager, Fileserver, Network, Drivers, …
  - Small, privileged μ-kernel

- Vertical communication
  - User/μ-kernel
  - Server/μ-kernel
  - Server/Device

- Horizontal communication
  - User/User
  - Server/User
  - Server/Server
Communication in Multi-Server Systems

- Analysis
  - Complex safety and isolation requirements
    - Need safe communication facilities
    - \(\mu\)-kernel approach: kernel-provided IPC
  - Modularization results in increased communication
    - Need flexible, low-overhead communication
    - E.g. L4 approach: synchronous IPC
  - Need abstraction for safe communication with devices
    - Driver subsystems are not implicitly trusted anymore
    - Need safe transaction with drivers
    - Ultimately limited by hardware (e.g., DMA problems)
Communication in Multi-Server Systems

- Case study: Mach
  - Kernel servers at user-level
    - Memory managers
    - Network Proxies
    - ...
  - Device drivers reside within the kernel
    - No user-level I/O subsystems
    - No I/O hardware abstractions
  - Asynchronous, unidirectional port system
    - Port: mailbox with only one receiver
    - Like TCP port, but per-process (rather than per-node)
    - Associated rights to send receive
  - Used for user/user and user/µ-kernel communication
    - Kernel-provided ports for system calls/services
    - kernel-provided startup ports for bootstrapping
Communication in Multi-Server Systems

- Case study: Mach
  - Messages are buffered by the kernel
    - Enables asynchronous non-blocking send
    - Requires copying data to and from kernel
    - Blocks on full buffers
  - In-line data:
    - Copied indirectly between sender and receiver
    - Guaranteed in-order copy
  - Out-of-line data:
    - Copy-on-write mechanism
    - Avoid copying large buffers
  - Operations
    - port_allocate(self, &port)
    - port_insert_send()
    - port_receive()
  - Port sets
    - Groups of ports
    - UNIX select() semantics
Communication in Multi-Server Systems

- **Case study: L4**
  - Kernel servers at user-level
    - Memory managers
    - User-level device drivers
  - **Synchronous, rendezvous-based IPC**
    - Communication endpoints: threads
    - Receiver can address multiple threads
    - Blocking and non-blocking variants (via timeouts)
  - **IPC types**
    - Register IPC
      - Fast register transfer
      - Use, e.g., for synchronization
    - String IPC
      - Transfer memory content
      - Copied during IPC
    - Map IPC
      - Send pages (or ports) of own address space
      - Potentially restrict access rights
      - Kernel updates MMU hardware during IPC
Communication in Multi-Server Systems

- **Case study: L4**
  - No buffering by the kernel
    - Pure synchronous send
    - Direct transfer from sender to receiver
  - Implement asynchronous IPC at user-level
    - Proxy threads
    - Buffer management at user level

- **Operations**
  - send to
  - receive from
  - combinations

- **IPC abstracts hardware**
  - Paging IPC
    - Can send page fault IPCs
    - Can send/receive memory/IO
  - Interrupts/Exceptions
    - Can send exception IPCs
    - Devices can send interrupt IPCs
    - Synchronous waiting for (actually asynchronous) interrupts
Communication in Multi-Server Systems

- Case study: Xen
  - Special case: virtual machine hypervisor
  - Multiple virtual machines at user-level
  - Client virtual machines
    - Concurrent workload
  - Control virtual machines
    - Management software
  - Hypervisor-aware client virtual machines (originally)

Source: Paul Barham et al. *Xen and the Art of Virtualization.*
Proceedings of the nineteenth ACM symposium on Operating systems principles p.164-177
Communication in Multi-Server Systems

- Case study: Xen
  - Two types of VM/hypervisor communication:
    - Control transfer
      - Hypercall:
        VM requires privileged services
        - Same as system call in traditional OSes
        - Hardware-provided system call mechanisms
          (int, syscall, ...)
      - Asynchronous notification:
        VM receives virtual interrupt/events
        - User-specified callback handler
        - User flag temporarily disables events (virtual interrupt mask)

Source: Paul Barham et al. *Xen and the Art of Virtualization.* Proceedings of the nineteenth ACM symposium on Operating systems principles p.164-177
Communication in Multi-Server Systems

- Case study: Xen
  - Two types of VM/hypervisor
  - Data transfer
    - Used for transferring data blocks to/from devices
    - E.g., NIC packets, disk blocks,
    - Device virtualization requires hardware interpositioning
    - Extra data transfer layer between client VM and real device
    - Xen-aware device drivers
  - No explicit concept for inter-VM communication
    - Use virtual network abstraction

Source: Paul Barham et al. *Xen and the Art of Virtualization.* Proceedings of the nineteenth ACM symposium on Operating systems principles p.164-177
Communication in Multi-Server Systems

- Case study: Xen
  - Data transfer: I/O descriptor rings
    - Circular queue of requests and responses
      - Contains descriptors, not data
      - References I/O buffers within a client VM
    - Producer/consumer semantics
      - Client VM puts descriptor into ring
      - Xen removes descriptor
      - Use asynchronous event notification for responses
    - No in-order processing
      - Descriptors/Responses are tagged
    - No data copying
      - Reference physical memory
      - Sending: Use scatter-gather DMA
      - Receiving: Use memory mapping (exchange page frame)

Source: Paul Barham et al. *Xen and the Art of Virtualization*. Proceedings of the nineteenth ACM symposium on Operating systems principles p.164-177
Thursday

- L4 API crash course – Part II