Outline

• Interprocess communication
  – shared memory (procedure-oriented)
  – message passing (message-oriented)
  – dual relationship of the two models
  – lightweight RPC

  [Laue78]
  Hugh C. Lauer and Roger M. Needham
  *On the Duality of Operating System Structures*

  [Bers90]
  Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy
  *Lightweight Remote Procedure Call*

Shared Memory (Procedure-oriented System)

• All execution is in one address space
  – processes move from one context to another (across module boundaries)
  – shared data structures protected by synchronization mechanisms (locks)
  – examples: POSIX threads, Java, Mesa, small multiprocessors

• Components (more about these in Lectures 7-11)
  – procedures
  – procedure-call facilities
    • synchronous
    • asynchronous (fork/join)
  – modules and monitors
    • monitors limit access to only one process at a time
  – module instantiation
  – condition variables
    • provides flexible synchronization

On the Duality of Operating System Structures
Message Passing (Message-oriented System)

- Execution is in separate address spaces
  - synchronization using message channels
  - examples: UNIX processes, large multiprocessors, etc.
  - other names: event-driven, process-based

Components

- messages and message identifiers
- message channels and ports
  - channels must be bound to ports
  - queues associated with ports
- message transmission operations
  - SendMessage[channel, body] returns id
  - AwaitReply[id]
  - WaitForMessage[set of ports]
  - SendReply[id, body]
- process declarations

Message Passing in Windows NT

- NT supports multiple subsystems: Win32, POSIX, etc.
  - application programs can be considered clients of the subsystem server

Local Procedure Call Facility (LPC)

- Port objects
  - connection ports establish/maintain connection between two processes
  - communication ports provide each client with its own communication channel to the subsystem server
    - a pair of private port objects (handles for all communication)
  
- 3 types of message-passing techniques:
  - small messages: write/read from the message queue associated with port
  - larger messages: written to sections (a shared-memory segment)
  - quick LPC (shares some ideas with LRPC)
    - server sets up a dedicated server thread which handles all requests from client
    - eliminates copying, overhead of using the port object, and dispatch

Characteristics of the Models

Shared Memory

- synchronization of processes and queuing for congested resources is centered around locks associated with data structures
- data is shared directly among processes, which lock portions of it for (relatively) short periods of time
- interaction with peripheral devices is also through shared memory
- processes inherit priorities dynamically from their contexts
- contexts represent an import form of protection
- global naming schemes help optimize context switching

Message Passing

- synchronization of processes and queuing for congested resources is implemented in message queues attached to processes associated with those resources
- data sharing is by argument passing: a process does not continue to manipulate data it has passed in a message to another process
- peripheral devices are treated as processes: interaction uses messages
- priorities tend to be statically assigned to processes
- because of static process contexts, global naming is less useful

Duality Mapping
Performance Similarity

- 3 components
  - execution times of the programs themselves
  - computational overhead of the primitive system operations
  - resource queuing and waiting times

- Arguments
  - no fundamental reason why overheads of primitive operations in the two models should be any different
    - sending a message (allocating a message block) vs. forking to a procedure
    - entering/leaving a monitor same overhead as enqueuing/dispatching messages
    - context-switching in both models (issue is when it happens)
  - similar argument for resource queuing and waiting times
    - can force use of same underlying scheduling policies

- Currently: Performance bias is towards shared memory
  - easier to support primitive operations in hardware

Duality of OS Structures: Discussion

- Original motivation
  - big controversy in OS design at that time (late 1970s)
  - one of the original flame wars!

- Still a controversy with the ultimate choice based on several factors
  - unprocessors
    - programming ease:
      - explicitly managing program state (message passing) vs.
      - process synchronization and deadlock detection (shared memory)
    - performance: context-switch overhead vs. locking overhead
    - other constraints:
      - can a subsystem be made thread-safe?
      - portability of application interfaces (local and remote placement of servers)
  - distributed/parallel operating systems
    - raw hardware support is the same
      - support shared memory on message-passing architecture

Remote Procedure Call (RPC)

- Steps:
  - binding: authenticate client request and bind it to a server
    - result is a message channel that can be used to communicate with the server
  - invocation and reply:
    - client-side stub injects request into the channel
    - server-side stub has a thread that is always running (waiting for requests)
    - this thread receives the request, does the invocation, replies with the result
    - client-side stub returns to the user program
  - termination:
    - channel becomes unavailable
    - error condition propagated to the user program
LRPC: Rationale

1. Most use of RPCs is cross-domain (as compared to cross-machine)
   - applications retain same interface (with similar protection concerns)
   - less than 6% of the calls in study comprising three Oses
     • V, Taos, Sun UNIX+NFS
     - caveat: results are very dependent upon workload
     • applications rely on more remote services these days

2. Most use of RPCs involves small messages with fixed-size arguments
   - 4 out of 5 arguments are fixed size (65% were 4 bytes or fewer)
   - 60% of RPCs involved message sizes < 32 bytes
   - marshalling usually done by other system procedures (not RPC code)
   - makes sense to provide a faster mechanism that does just byte copying

- Overheads of cross-machine RPC mechanisms dominate!

RPC: Sources of Overhead

### Table II. Cross-Domain Performance (times are in microseconds)

<table>
<thead>
<tr>
<th>System</th>
<th>Processor</th>
<th>Null (theoretical minimum)</th>
<th>Null (actual)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accent</td>
<td>PERQ</td>
<td>444</td>
<td>2,300</td>
<td>1,856</td>
</tr>
<tr>
<td>Taos</td>
<td>Finefly C-VAX</td>
<td>100</td>
<td>664</td>
<td>664</td>
</tr>
<tr>
<td>Mach</td>
<td>C-VAX</td>
<td>90</td>
<td>754</td>
<td>664</td>
</tr>
<tr>
<td>V</td>
<td>68020</td>
<td>170</td>
<td>730</td>
<td>560</td>
</tr>
<tr>
<td>Amoeba</td>
<td>68020</td>
<td>170</td>
<td>800</td>
<td>630</td>
</tr>
<tr>
<td>DASH</td>
<td>68020</td>
<td>170</td>
<td>1,590</td>
<td>1,120</td>
</tr>
</tbody>
</table>

- **Stub overheads**
- **Message overheads**: buffering, access validation, transfer (queues)
  - four copy operations are required on each RPC (client-to-kernel, kernel-to-server, and two more on the return)
- **Thread overheads**: scheduling, context switch, and dispatch

Design of LRPC

- Servers
  - exist in their own protection domains
  - export a specific set of procedures
  - are passive

- Clients call the server by way of a **kernel trap**
  - the kernel
    - validates the caller
    - creates a call linkage
    - dispatches the client’s concrete thread directly to the server domain
  - the client
    - provides the server with an argument stack
    - its own concrete thread of execution
  - upon completion
    - control and results return through the kernel to the caller

LRPC: Binding

- Conceptually: Servers *export* interfaces and clients *bind* to them
- Steps
  1. server exports interface
  2. clerk (in the LRPC runtime library) registers interface with the kernel, and waits for import requests
  3. client imports interface via kernel
  4. clerk replies with **Procedure Descriptor List (PDL)**
     - one Procedure Descriptor (PD) per exported procedure
       - entry address
       - size of the procedure’s argument stack (A-stack)
  5. kernel allocates memory
     - **A-stacks** for each PD
       - equal to the number of simultaneous calls allowed
       - mapped read-write and shared by both domains
     - **linkage record** for each A-stack
  6. kernel returns **Binding Object** and A-stack list to client
LRPC: Calling

- **Client**
  - client stub dequeues A-stack
  - arguments are copied onto A-Stack
  - registers are loaded with address of A-Stack, Binding Object, and procedure ID
  - kernel is trapped
- **Kernel**
  - verifies Binding Object, procedure ID, and the A-stack
  - locates the correct PD and linkage
  - ensures that no other thread is using that A-Stack/linkage pair
  - records the caller’s return address and current SP in linkage
  - pushes linkage onto the top of a stack kept in thread’s control block
  - locates execution stack (E-stack) in the server’s domain
  - updates the thread’s user SP to run off new E-stack
  - reloads processor’s VM registers with those of the server domain.
  - performs upcall into the server stub to address specified in the PD
- **Server**
  - server procedure executes and can directly access parameters via A-Stack
  - procedure returns through its own stub and traps kernel
  - kernel switches the thread back to client

LRPC: Issues in Argument Copying

- A-stacks reduce number of times arguments/results are copied
  - cross-domain RPC requires 4 copies in each direction
    - client stub -> RPC message -> kernel buffer -> RPC message -> server stub
  - arguments/results are written only once in LRPC (on the A-stack)
- Issues
  - protection from third-party domains
    - kernel ensures this by pairwise allocation of A-stacks
  - asynchronous update (by either client or server) of values on stack
    - no protection!
  - type safety (server gets the arguments it expects)
    - in RPC: check done explicitly in the server stub
    - in LRPC: check can be folded in with the copying onto the A-stack in the client-side stub
    - LRPC does this by relying on conformity of Modula 2 argument types
    - an example of language-based protection (more details in Lecture 19)

LRPC: Additional Issues

- Multiple processors can be conveniently used
  - to improve throughput: more simultaneous RPCs
  - to lower call latency: caching of server domain contexts
    - reduces overhead of switching VM contexts
- Transparency is preserved
  - A bit in the Binding Object indicates whether call is remote or local
- Domain termination
  - issue: call from a terminated server domain must return to the caller, and a call should not be allowed to return to a terminated client (caller)
  - basic idea:
    - revoke Binding Object (to prevent additional calls)
    - hang on to the client- and server-domain threads (those that are involved in active LRPC calls) till the call completes

LRPC: Performance

- Arguments/results are only copied once (onto A-Stack)
  - as opposed to 4 times
    - client stub -> message -> kernel buffer -> kernel buffer -> message -> server stub
- Domain switching is roughly 3 times faster than RPC
  - client domain’s concrete thread continues to run
- TLB misses are minimized in LRPC
  - not doing complete domain switching + careful allocation of A-stacks
- No apparent throughput limiting factor on multiprocessor systems
LRPC: Discussion

- LRPC trades off memory-management costs for reducing overhead
  - kernel needs to allocate/manage more storage

- Higher risk of mutual interference between clients and servers?
  - shared A-stacks
  - shared E-stacks
  - language- (type safety) and capability-based (BindingObject) protection
    - more of this in Lecture 19

- What happens to a server that migrates from a remote machine to the local machine (and vice versa)?
  - difficulty is with handling active bindings ...

Lecture Summary

- Interprocess communication
  - shared memory (procedure-oriented)
  - message passing (message-oriented)
  - dual relationship of these models
  - lightweight RPC
    - control transfer and communication model of capability systems
    - programming semantics and protection model of RPC

Next Lecture

- CPU scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
  - multiple-processor scheduling

Reading

- Silberschatz/Galvin: Sections 5.1-5.4