CSE 513
Introduction to Operating Systems

Class 9 - Distributed and Multiprocessor Operating Systems

Jonathan Walpole
Dept. of Comp. Sci. and Eng.
Oregon Health and Science University
Why use parallel or distributed systems?

- Speed - reduce time to answer
- Scale - increase size of problem
- Reliability - increase resilience to errors
- Communication - span geographical distance
Overview

- Multiprocessor systems
- Multi-computer systems
- Distributed systems
Multiprocessor, multi-computer and distributed architectures

- shared memory multiprocessor
- message passing multi-computer (cluster)
- wide area distributed system
Multiprocessor Systems
Multiprocessor systems

- **Definition:**
  - A computer system in which two or more CPUs share full access to a common RAM

- **Hardware implements shared memory among CPUs**

- **Architecture determines whether access times to different memory regions are the same**
  - UMA - uniform memory access
  - NUMA - non-uniform memory access
Bus-based UMA and NUMA architectures

Bus becomes the bottleneck as number of CPUs increases
Crossbar switch-based UMA architecture

Interconnect cost increases as square of number of CPUs
Multiprocessors with 2x2 switches

(a) 

(b) 

Module | Address | Opcode | Value
Omega switching network from 2x2 switches

Interconnect suffers contention, but costs less
NUMA multiprocessors

- Single address space visible to all CPUs
- Access to remote memory via commands
  - LOAD
  - STORE
- Access to remote memory slower than to local memory
- Compilers and OS need to be careful about data placement
Directory-based NUMA multiprocessors

(a) 256-node directory based multiprocessor
(b) Fields of 32-bit memory address
(c) Directory at node 36
Operating systems for multiprocessors

- **OS structuring approaches**
  - Private OS per CPU
  - Master-slave architecture
  - Symmetric multiprocessing architecture

- **New problems**
  - multiprocessor synchronization
  - multiprocessor scheduling
The private OS approach

- Implications of private OS approach
  - shared I/O devices
  - static memory allocation
  - no data sharing
  - no parallel applications
The master-slave approach

- **OS only runs on master CPU**
  - Single kernel lock protects OS data structures
  - Slaves trap system calls and place process on scheduling queue for master
- **Parallel applications supported**
  - Memory shared among all CPUs
- **Single CPU for all OS calls becomes a bottleneck**
Symmetric multiprocessing (SMP)

- **OS runs on all CPUs**
  - Multiple CPUs can be executing the OS simultaneously
  - Access to OS data structures requires synchronization
  - Fine grain critical sections lead to more locks and more parallelism ... and more potential for deadlock
Multiprocessor synchronization

- Why is it different compared to single processor synchronization?
  - Disabling interrupts does not prevent memory accesses since it only affects “this” CPU
  - Multiple copies of the same data exist in caches of different CPUs
    - atomic lock instructions do CPU-CPU communication
  - Spinning to wait for a lock is not always a bad idea
Synchronization problems in SMPs

1. CPU 1 reads a 0
2. CPU 2 reads a 0
3. CPU 1 writes a 1
4. CPU 2 writes a 1

Word 1000 is initially 0

TSL instruction is non-trivial on SMPs
Avoiding cache thrashing during spinning

Multiple locks used to avoid cache thrashing
**Spinning versus switching**

- **In some cases CPU “must” wait**
  - scheduling critical section may be held
- **In other cases spinning may be more efficient than blocking**
  - spinning wastes CPU cycles
  - switching uses up CPU cycles also
  - if critical sections are short spinning may be better than blocking
  - static analysis of critical section duration can determine whether to spin or block
  - dynamic analysis can improve performance
Multiprocessor scheduling

- Two dimensional scheduling decision
  - time (which process to run next)
  - space (which processor to run it on)

- Time sharing approach
  - single scheduling queue shared across all CPUs

- Space sharing approach
  - partition machine into sub-clusters
Time sharing

- Single data structure used for scheduling
- Problem - scheduling frequency influences inter-thread communication time
Interplay between scheduling and IPC

Problem with communication between two threads
- both belong to process A
- both running out of phase
Space sharing

- Groups of cooperating threads can communicate at the same time
  - fast inter-thread communication time
Gang scheduling

- **Problem with pure space sharing**
  - Some partitions are idle while others are overloaded

- **Can we combine time sharing and space sharing and avoid introducing scheduling delay into IPC?**

- **Solution: Gang Scheduling**
  - Groups of related threads scheduled as a unit (gang)
  - All members of gang run simultaneously on different timeshared CPUs
  - All gang members start and end time slices together
## Gang scheduling

<table>
<thead>
<tr>
<th>Time slot</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B₀</td>
<td>B₁</td>
<td>B₂</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
</tr>
<tr>
<td>2</td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>D₄</td>
<td>E₀</td>
</tr>
<tr>
<td>3</td>
<td>E₁</td>
<td>E₂</td>
<td>E₃</td>
<td>E₄</td>
<td>E₅</td>
<td>E₆</td>
</tr>
<tr>
<td>4</td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
</tr>
<tr>
<td>5</td>
<td>B₀</td>
<td>B₁</td>
<td>B₂</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
</tr>
<tr>
<td>6</td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>D₄</td>
<td>E₀</td>
</tr>
<tr>
<td>7</td>
<td>E₁</td>
<td>E₂</td>
<td>E₃</td>
<td>E₄</td>
<td>E₅</td>
<td>E₆</td>
</tr>
</tbody>
</table>
Multi-computer Systems
Multi-computers

- Also known as
  - cluster computers
  - clusters of workstations (COWs)

- Definition: Tightly-coupled CPUs that do not share memory
Multi-computer interconnection topologies

(a) single switch  (b) ring  (c) grid
(d) double torus  (e) cube  (f) hypercube
Store & forward packet switching

(a) Entire packet
(b) Entire packet
(c) Entire packet
Network interfaces in a multi-computer

- Network co-processors may off-load communication processing from the main CPU
OS issues for multi-computers

- Message passing performance

- Programming model
  - synchronous vs asynchronous message passing
  - distributed virtual memory

- Load balancing and coordinated scheduling
Optimizing message passing performance

- Parallel application performance is dominated by communication costs
  - interrupt handling, context switching, message copying ...

- Solution - get the OS out of the loop
  - map interface board to all processes that need it
  - active messages - give interrupt handler address of user-buffer
  - sacrifice protection for performance?
CPU / network card coordination

- How to maximize independence between CPU and network card while sending/receiving messages?
  - Use send & receive rings and bit-maps
  - One always sets bits, one always clears bits
Blocking vs non-blocking send calls

- **Minimum services provided**
  - send and receive commands

- These can be blocking (synchronous) or non-blocking (asynchronous) calls

(a) Blocking send call

(b) Non-blocking send call
Blocking vs non-blocking calls

- **Advantages of non-blocking calls**
  - ability to overlap computation and communication
    improves performance

- **Advantages of blocking calls**
  - simpler programming model
Remote procedure call (RPC)

- **Goal**
  - support execution of remote procedures
  - make remote procedure execution indistinguishable from local procedure execution
  - allow distributed programming without changing the programming model
Remote procedure call (RPC)

- Steps in making a remote procedure call
  - client and server stubs are proxies
RPC implementation issues

- **Cannot pass pointers**
  - call by reference becomes copy-restore (at best)

- **Weakly typed languages**
  - Client stub cannot determine size of reference parameters
  - Not always possible to determine parameter types

- **Cannot use global variables**
  - may get moved (replicated) to remote machine

- **Basic problem - local procedure call relies on shared memory**
Distributed shared memory (DSM)

- **Goal**
  - use software to create the illusion of shared memory on top of message passing hardware
  - leverage virtual memory hardware to page fault on non-resident pages
  - service page faults from remote memories instead of from local disk
Distributed shared memory (DSM)

- DSM at the hardware, OS or middleware layer
Page replication in DSM systems

Replication

(a) Pages distributed on 4 machines

(b) CPU 0 reads page 10

(c) CPU 1 reads page 10
Consistency and false sharing in DSM

A and B are unrelated shared variables that just happen to be on the same page.
Strong memory consistency

- Total order enforces sequential consistency
  - intuitively simple for programmers, but very costly to implement
  - not even implemented in non-distributed machines!
Scheduling in multi-computer systems

- Each computer has its own OS
  - local scheduling applies

- Which computer should we allocate a task to initially?
  - Decision can be based on load (load balancing)
  - load balancing can be static or dynamic
Graph-theoretic load balancing approach

- Two ways of allocating 9 processes to 3 nodes
- Total network traffic is sum of arcs cut by node boundaries
- The second partitioning is better
Sender-initiated load balancing

- Overloaded nodes (senders) off-load work to underloaded nodes (receivers)
Receiver-initiated load balancing

- Underloaded nodes (receivers) request work from overloaded nodes (senders)
Distributed Systems
Distributed systems

- **Definition:** Loosely-coupled CPUs that do not share memory
  - where is the boundary between tightly-coupled and loosely-coupled systems?

- **Other differences**
  - single vs multiple administrative domains
  - geographic distribution
  - homogeneity vs heterogeneity of hardware and software
Comparing multiprocessors, multi-computers and distributed systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Multiprocessor</th>
<th>Multicomputer</th>
<th>Distributed System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node configuration</td>
<td>CPU</td>
<td>CPU, RAM, net interface</td>
<td>Complete computer</td>
</tr>
<tr>
<td>Node peripherals</td>
<td>All shared</td>
<td>Shared exc. maybe disk</td>
<td>Full set per node</td>
</tr>
<tr>
<td>Location</td>
<td>Same rack</td>
<td>Same room</td>
<td>Possibly worldwide</td>
</tr>
<tr>
<td>Internode communication</td>
<td>Shared RAM</td>
<td>Dedicated interconnect</td>
<td>Traditional network</td>
</tr>
<tr>
<td>Operating systems</td>
<td>One, shared</td>
<td>Multiple, same</td>
<td>Possibly all different</td>
</tr>
<tr>
<td>File systems</td>
<td>One, shared</td>
<td>One, shared</td>
<td>Each node has own</td>
</tr>
<tr>
<td>Administration</td>
<td>One organization</td>
<td>One organization</td>
<td>Many organizations</td>
</tr>
</tbody>
</table>
Ethernet as an interconnect

- Bus-based vs switched Ethernet
The Internet as an interconnect
OS issues for distributed systems

- Common interfaces above heterogeneous systems
  - Communication protocols
  - Distributed system middleware

- Choosing suitable abstractions for distributed system interfaces
  - distributed document-based systems
  - distributed file systems
  - distributed object systems
Network service and protocol types

<table>
<thead>
<tr>
<th>Service</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable message stream</td>
<td>Sequence of pages of a book</td>
</tr>
<tr>
<td>Reliable byte stream</td>
<td>Remote login</td>
</tr>
<tr>
<td>Unreliable connection</td>
<td>Digitized voice</td>
</tr>
<tr>
<td>Unreliable datagram</td>
<td>Network test packets</td>
</tr>
<tr>
<td>Acknowledged datagram</td>
<td>Registered mail</td>
</tr>
<tr>
<td>Request-reply</td>
<td>Database query</td>
</tr>
</tbody>
</table>
Protocol interaction and layering
Homogeneity via middleware

Common base for applications

Application
Middleware
Windows
Pentium

Application
Middleware
Linux
Pentium

Application
Middleware
Solaris
SPARC

Application
Middleware
Mac OS
Macintosh
Distributed system middleware models

- Document-based systems
- File-based systems
- Object-based systems
Document-based middleware - WWW
Document-based middleware

How the browser gets a page
- Asks DNS for IP address
- DNS replies with IP address
- Browser makes connection
- Sends request for specified page
- Server sends file
- TCP connection released
- Browser displays text
- Browser fetches, displays images
File-based middleware

- Design issues
  - Naming and name resolution
  - Architecture and interfaces
  - Caching strategies and cache consistency
  - File sharing semantics
  - Disconnected operation and fault tolerance
(b) Clients with the same view of name space
(c) Clients with different views of name space
Naming and transparency issues

- Can clients distinguish between local and remote files?

- Location transparency
  - file name does not reveal the file's physical storage location.

- Location independence
  - the file name does not need to be changed when the file's physical storage location changes.
Global vs local name spaces

- **Global name space**
  - file names are globally unique
  - any file can be named from any node

- **Local name spaces**
  - remote files must be inserted in the local name space
  - file names are only meaningful within the calling node
  - but how do you refer to remote files in order to insert them?
    - globally unique file handles can be used to map remote files to local names
Building a name space with super-root

- Super-root / machine name approach
  - concatenate the host name to the names of files stored on that host
  - system-wide uniqueness guaranteed
  - simple to locate a file
  - not location transparent or location independent
Building a name space using mounting

- Mounting remote file systems
  - exported remote directory is imported and mounted onto local directory
  - accesses require a globally unique file handle for the remote directory
  - once mounted, file names are location-transparent
    - location can be captured via naming conventions
  - are they location independent?
    - location of file vs location of client?
    - files have different names from different places
Local name spaces with mounting

Client A

remote

bin

vu

mbox

Exported directory mounted by client

Server

users

steen

mbox

Exported directory mounted by client

Client B

work

me

bin

mbox

Network
Nested mounting on multiple servers

Client

Exported directory contains imported subdirectory

bin

draw

install

Server A

Client imports directory from server A

packages

draw

install

Server B

Server A imports directory from server B

install

Client needs to explicitly import subdirectory from server B

Network
NSF name space

- Server exports a directory
- mountd: provides a unique file handle for the exported directory
- Client uses RPC to issue nfs_mount request to server
- mountd receives the request and checks whether
  - the pathname is a directory?
  - the directory is exported to this client?
**NFS file handles**

- **V-node contains**
  - reference to a file handle for mounted remote files
  - reference to an i-node for local files

- **File handle uniquely names a remote directory**
  - file system identifier: unique number for each file system (in UNIX super block)
  - i-node and i-node generation number
Mounting on-demand

- Need to decide where and when to mount remote directories
- Where? - Can be based on conventions to standardize local name spaces (ie., /home/username for user home directories)
- When? - boot time, login time, access time, ...?
- What to mount when?
  - How long does it take to mount everything?
  - Do we know what everything is?
  - Can we do mounting on-demand?
- An automounter is a client-side process that handles on-demand mounting
  - it intercepts requests and acts like a local NFS server
Distributed file system architectures

- **Server side**
  - how do servers export files
  - how do servers handle requests from clients?

- **Client side**
  - how do applications access a remote file in the same way as a local file?

- **Communication layer**
  - how do clients and servers communicate?
Local access architectures

- **Local access approach**
  - move file to client
  - local access on client
  - return file to server
  - data shipping approach
Remote access architectures

- Remote access
  - leave file on server
  - send read/write operations to server
  - return results to client
  - function shipping approach
File-level interface

- Accesses can be supported at either the file granularity or block granularity

- File-level client-server interface
  - local access model with whole file movement and caching
  - remote access model client-server interface at system call level
  - client performs remote open, read, write, close calls
Block-level interface

- **Block-level client-server interface**
  - client-server interface at file system or disk block level
  - server offers virtual disk interface
  - client file accesses generate block access requests to server
  - block-level caching of parts of files on client
NFS architecture

The basic NFS architecture for UNIX systems.
NFS server side

- **Mountd**
  - server exports directory via mountd
  - mountd provides the initial file handle for the exported directory
  - client issues `nfs_mount` request via RPC to mountd
  - mountd checks if the pathname is a directory and if the directory is exported to the client

- **nfsd**: services NFS RPC calls, gets the data from its local file system, and replies to the RPC
  - Usually listening at port 2049

- **Both mountd and nfsd use RPC**
## Communication layer: NFS RPC Calls

<table>
<thead>
<tr>
<th>Proc.</th>
<th>Input args</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup</td>
<td>dirfh, name</td>
<td>status, fhandle, fattr</td>
</tr>
<tr>
<td>read</td>
<td>fhandle, offset, count</td>
<td>status, fattr, data</td>
</tr>
<tr>
<td>create</td>
<td>dirfh, name, fattr</td>
<td>status, fhandle, fattr</td>
</tr>
<tr>
<td>write</td>
<td>fhandle, offset, count, data</td>
<td>status, fattr</td>
</tr>
</tbody>
</table>

- NFS / RPC uses XDR and TCP/IP
- fhandle: 64-byte opaque data (in NFS v3)
  - what's in the file handle?
NFS file handles

- **V-node contains**
  - reference to a file handle for mounted remote files
  - reference to an i-node for local files

- **File handle uniquely names a remote directory**
  - file system identifier: unique number for each file system (in UNIX super block)
  - i-node and i-node generation number
NFS client side

- Accessing remote files in the same way as accessing local files requires kernel support
  - Vnode interface

```c
struct file
{
    Mode
    Vnode
    offset
}

struct vnode
{
    V_data
    fs_op
    {int (*open)();
    int (*close)();
    int (*read)();
    int (*write)();
    int (*lookup)();
    ...
    }
```

read(fd,..)
Caching vs pure remote service

- **Network traffic?**
  - caching reduces remote accesses ⇒ reduces network traffic
  - caching generates fewer, larger, data transfers

- **Server load?**
  - caching reduces remote accesses ⇒ reduces server load

- **Server disk throughput?**
  - optimized better for large requests than random disk blocks

- **Data integrity?**
  - cache-consistency problem due to frequent writes

- **Operating system complexity?**
  - simpler for remote service.
Four places to cache files

- **Server's disk**: slow performance
- **Server's memory**
  - cache management, how much to cache, replacement strategy
  - still slow due to network delay
- **Client's disk**
  - access speed vs server memory?
  - large files can be cached
  - supports disconnected operation
- **Client's memory**
  - fastest access
  - can be used by diskless workstations
  - competes with the VM system for physical memory space
Cache consistency

- Reflecting changes to local cache to master copy
- Reflecting changes to master copy to local caches

Copy 1

write

Master copy

Copy 2

update/invalidate
Common update algorithms for client caching

- **Write-through**: all writes are carried out immediately
  - Reliable: little information is lost in the event of a client crash
  - Slow: cache not useful for writes

- **Delayed-write**: writes do not immediately propagate to server
  - batching writes amortizes overhead
  - wait for blocks to fill
  - if data is written and then deleted immediately, data need not be written at all (20-30% of new data is deleted with 30 secs)

- **Write-on-close**: delay writing until the file is closed at the client
  - semantically meaningful delayed-write policy
  - if file is open for short duration, works fine
  - if file is open for long, susceptible to losing data in the event of client crash
Cache coherence

- How to keep locally cached data up to date / consistent?
  - **Client-initiated approach**
    - check validity on every access: too much overhead
    - first access to a file (e.g., file open)
    - every fixed time interval
  - **Server-initiated approach**
    - server records, for each client, the (parts of) files it caches
    - server responds to updates by propagation or invalidation
- **Disallow caching during concurrent-write or read/write sharing**
  - allow multiple clients to cache file for read only access
  - flush all client caches when the file is opened for writing
NFS – server caching

- **Reads**
  - use the local file system cache
  - prefetching in UNIX using read-ahead

- **Writes**
  - write-through (synchronously, no cache)
  - commit on close (standard behaviour in v4)
NFS – client caching (reads)

- Clients are responsible for validating cache entries (stateless server)

- Validation by checking last modification time
  - time stamps issues by server
  - automatic validation on open (with server?)

- A cache entry is considered valid if one of the following are true:
  - cache entry is less than t seconds old (3-30 s for files, 30-60 s for directories)
  - modified time at server is the same as modified time on client
NFS – client caching (writes)

- **Delayed writes**
  - modified files are marked dirty and flushed to server on close (or sync)

- **Bio-daemons** (*block* input-output)
  - read-ahead requests are done asynchronously
  - write requests are submitted when a block is filled
Semantics of File sharing

- (a) single processor gives sequential consistency
- (b) distributed system may return obsolete value
Consistency semantics for file sharing

- **What value do reads see after writes?**
  - **UNIX semantics**
    - value read is the value stored by last write
    - writes to an open file are visible immediately to others with the file open
    - easy to implement with one server and no cache
  - **Session semantics**
    - writes to an open file are not visible immediately to others with the file opened already
    - changes become visible on close to sessions started later

- **Immutable-Shared-Files semantics - simple to implement**
  - A sharable file cannot be modified
  - File names cannot be reused and its contents may not be altered

- **Transactions**
  - All changes have all-or-nothing property
  - W1,R1,R2,W2 not allowed where P1 = W1;W2 and P2 = R1;R2
NFS - file sharing semantics

- Not UNIX semantics!
- Unspecified in NFS standard
- Not clear because of timing dependencies
- Consistency issues can arise
  - Example: Jack and Jill have a file cached. Jack opens the file and modifies it, then he closes the file. Jill then opens the file (before t seconds have elapsed) and modifies it as well. Then she closes the file. Are both Jack’s and Jill’s modifications present in the file? What if Jack closes the file after Jill opens it?
- Locking part of v4 (byte range, leasing)