CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003
Lecture Notes: Multiprocessors (updated version)

Announcement

- Colloq by Dennis Ritchie
  - "UNIX and Beyond: Themes of Operating Systems Research at Bell Labs,"
  - 4:30 pm, Wednesday, November 12
  - Wu-Chen Auditorium
- Written Assignment will be post later today

Systems with Multiple CPUs

- Collection of independent CPUs (or computers) that appears to the users/applications as a single system
- Technology trends
  - Powerful, yet cheap, microprocessors
  - Advances in communications
  - Physical limits on computing power of a single CPU
- Examples
  - Network of workstations
  - Servers with multiple processors
  - Network of computers of a company
  - Microcontrollers inside a car

Advantages

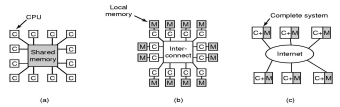
- Data sharing: allows many users to share a common data base
- Resource sharing: expensive devices such as a color printer
- Parallelism and speed-up: multiprocessor system can have more computing power than a mainframe
- Better price/performance ratio than mainframes
- Reliability: Fault-tolerance can be provided against crashes of individual machines
- Flexibility: spread the workload over available machines
- Modular expandability: Computing power can be added in small increments (upgrading CPUs like memory)

Design Issues

- Transparency: How to achieve a single-system image
  - How to hide distribution of memory from applications?
  - How to maintain consistency of data?
- Performance
  - How to exploit parallelism?
  - How to reduce communication delays?
- Scalability: As more components (say, processors) are added, performance should not degrade
  - Centralized schemes (e.g. broadcast messages) don't work
- Security

Classification

- Multiprocessors
  - Multiple CPUs with shared memory
  - Memory access delays about 10 – 50 nsec
- Multicomputers
  - Multiple computers, each with own CPU and memory, connected by a high-speed interconnect
  - Tightly coupled with delays in micro-seconds
- Distributed Systems
  - Loosely coupled systems connected over Local Area Network (LAN), or even long-haul networks such as Internet
  - Delays can be seconds, and unpredictable



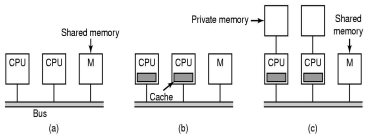
Mutiprocessors

Multiprocessor Systems

- Multiple CPUs with a shared memory
- From an application's perspective, difference with single-processor system need not be visible
  - Virtual memory where pages may reside in memories associated with other CPUs
  - Applications can exploit parallelism for speed-up
- Topics to cover
  1. Multiprocessor architectures (Section 8.1.1)
  2. Cache coherence
  3. OS organization (Section 8.1.2)
  4. Synchronization (Section 8.1.3)
  5. Scheduling (Section 8.1.4)

Multiprocessor Architecture

- UMA (Uniform Memory Access)
  - Time to access each memory word is the same
  - Bus-based UMA
  - CPUs connected to memory modules through switches
- NUMA (Non-uniform memory access)
  - Memory distributed (partitioned among processors)
  - Different access times for local and remote accesses



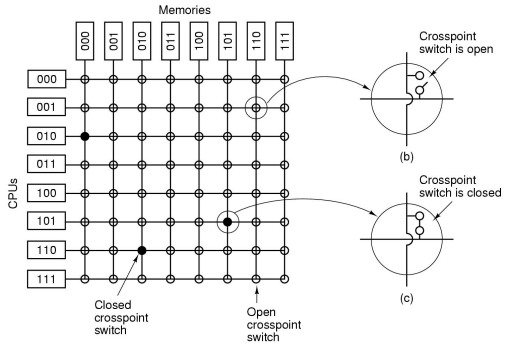
Bus-based UMA

- All CPUs and memory module connected over a shared bus
- To reduce traffic, each CPU also has a cache
- Key design issue: how to maintain coherency of data that appears in multiple places?
- Each CPU can have a local memory module also that is not shared with others
- Compilers can be designed to exploit the memory structure
- Typically, such an architecture can support 16 or 32 CPUs as a common bus is a bottleneck (memory access not parallelized)

Switched UMA

- Goal: To reduce traffic on bus, provide multiple connections between CPUs and memory units so that many accesses can be concurrent
- Crossbar Switch: Grid with horizontal lines from CPUs and vertical lines from memory modules
- Crossbar at (i,j) can connect i-th CPU with j-th memory module
- As long as different processors are accessing different modules, all requests can be in parallel
- Non-blocking: waiting caused only by contention for memory, but not for bus
- Disadvantage: Too many connections (quadratic)
- Many other networks: omega, counting, …

Crossbar Switch



Cache Coherence

- Many processors can have locally cached copies of the same object
  - Level of granularity can be an object or a block of 64 bytes
- We want to maximize concurrency
  - If many processors just want to read, then each one can have a local copy, and reads won't generate any bus traffic
- We want to ensure coherence
  - If a processor writes a value, then all subsequent reads by other processors should return the latest value
- Coherence refers to a logically consistent global ordering of reads and writes of multiple processors
- Modern multiprocessors support intricate schemes

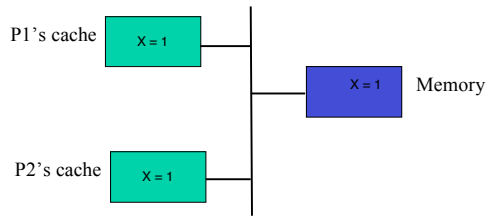
13

Consistency and replication

- Need to replicate (cache) to improve performance
  - How updates are propagated between cached replicas
  - How to keep them consistent
- How to keep them consistency (much more complicated than sequential processor)
  - When a processor change the vale value of its copy of a variable,
    - the other copies are invalidated (invalidate protocol), or
    - the other copies are updated (update protocol).

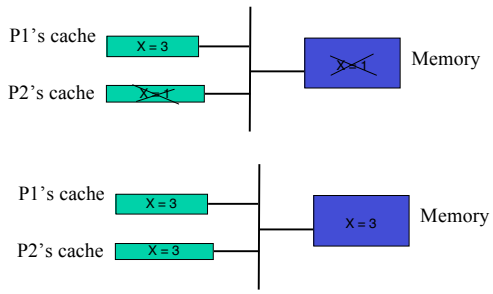
14

Example



15

Invalidate vs. update protocols

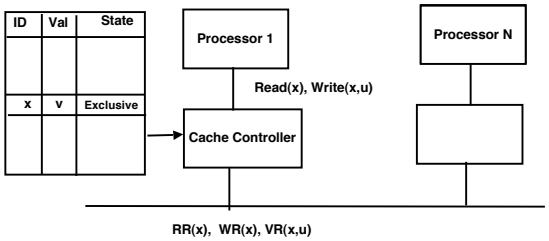


16

Snoopy Protocol

- Each processor, for every cached object, keeps a state that can be **Invalid**, **Exclusive** or **Read-only**
- Goal: If one has **Exclusive** copy then all others must be **Invalid**
- Each processor issues three types of messages on bus
  - Read-request (**RR**), Write-request (**WR**), and Value-response (**VR**)
  - Each message identifies object, and VR has a tagged value
- Assumption:
  - If there is contention for bus then only one succeeds
  - No split transactions (**RR** will have a response by **VR**)
- Protocol is called Snoopy, because everyone is listening to the bus all the time, and updates state in response to messages **RR** and **WR**
- Each cache controller responds to 4 types of events
  - Read or write operation issued by its processor
  - Messages (**RR**, **WR**, or **VR**) observed on the bus
- Caution: This is a simplified version

Snoopy Cache Coherence



Snoopy Protocol

- If state is **Read-only**
  - Read operation: return local value
  - Write operation: Broadcast **WR** message on bus, update state to **Exclusive**, and update local value
  - **WR** message on bus: update state to **Invalid**
  - **RR** message on bus: broadcast **VR(v)** on bus
- If state is **Exclusive**
  - Read operation: return local value
  - Write operation: update local value
  - **RR** message on bus: Broadcast **VR(v),** and change state to **Read-only**
  - **WR** message on bus: update state to Invalid
- If state is **Invalid**
  - Read operation: Broadcast **RR**, Receive **VR(v),** update state to **Read-only**, and local value to v
  - Write operation: As in first case
  - **VR(v)** message on bus: Update state to **Read-only**, and local copy to v
  - **WR** message on the bus: do nothing

Sample Scenario for Snoopy

- Assume 3 processors P1, P2, P3. One object *x : int*
- Initially, P1's entry for x is invalid, P2's entry is Exclusive with value 3, and P3's entry is invalid
- A process running on P3 issues Read(x)
- P3 sends the message RR(x) on the bus
- P2 updates its entry to Read-only, and sends the message VR(x,3) on the bus
- P3 updates its entry to Read-only, records the value 3 in the cache, and returns the value 3 to Read(x)
- P1 also updates the x-entry to (Read-Only, 3)
- Now, if Read(x) is issued on any of the processors, no messages will be exchanged, and the corresponding processor will just return value 3 by a local look-up

- P1: x=(inv,-) … x=(ro,3)
- P2: x=(exc,3) … X=(ro,3); VR(x,3);
- P3: x=(inv,-) … Read(x); RR(x); … x=(ro,3),return(x,3)

Snoopy Scenario (Continued)

- Suppose a process running on P1 issues Write(x,0)
- At the same time, a process running on P2 issues Write(x,2)
- P1 will try to send WR on the bus, as well as P2 will try to send WR on the bus
- Only one of them succeeds, say, P1 succeeds
- P1 will update cache-entry to (Exclusive,0)
- P3 will update cache-entry to Invalid
- P2 will update cache-entry to Invalid
- Now, Read / Write operations by processes on P1 will use local copy, and won't generate any messages

- P1: Write(x,0); WR(x); x=(ex,0)
- P2: Write(x,2); WR(x); x=(inv,-)
- P3: … x=(inv,-)

Notions of consistency

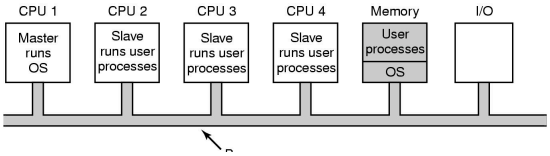
- Strict consistency: any read on a data item x returns a value corresponding to the result of the most recent write on x (need absolute global time)
  - P1: w(x)a P1: w(x)a
  - P2: r(x)a P2: r(x)NIL r(x)a
- Sequential consistency: the result of any execution is the same as if the R/W operations by all processes were executed in some sequential order and the operations of each process appear in this sequence in the order specified by its program
  - P1: w(x)a P1: w(x)a
  - P2: w(x)b P2: w(x)b
  - P3: r(x)b r(x)a P3: r(x)b r(x)a
  - P4: r(x,b) r(x,a) P4: r(x)a r(x)b

Multiprocessor OS

- How should OS software be organized?
- OS should handle allocation of processes to processors. Challenge due to shared data structures such as process tables and ready queues
- OS should handle disk I/O for the system as a whole
- Two standard architectures
  - Master-slave
  - Symmetric multiprocessors (SMP)

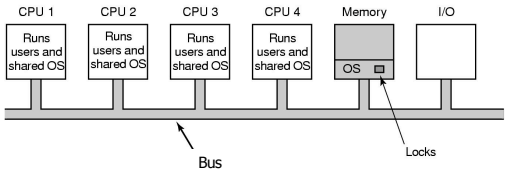
Master-Slave Organization

- Master CPU runs kernel, all others run user processes
- Only one copy of all OS data structures
- All system calls handled by master CPU
- Problem: Master CPU can be a bottleneck



Symmetric Multiprocessing (SMP)

- Only one kernel space, but OS can run on any CPU
- Whenever a user process makes a system call, the same CPU runs OS to process it
- Key issue: Multiple system calls can run in parallel on different CPUs
  - Need locks on all OS data structures to ensure mutual exclusion for critical updates
- Design issue: OS routines should have independence so that level of granularity for locking gives good performance



Synchronization

- Recall: Mutual exclusion solutions to protect critical regions involving updates to shared data structures
- Classical single-processor solutions
  - Disable interrupts
  - Powerful instructions such as Test&Set (TSL)
  - Software solution such as Peterson's algorithm
- In multiprocessor setting, competing processes can all be OS routines (e.g., to update process table)
- Disabling interrupts is not relevant as there are multiple CPUs
- TSL can be used, but requires modification

Original Solution using TSL

```
Shared variable: lock :{0,1}
lock==1 means some process is in CS
Initially lock is 0
Code for process P0 as well as P1:
while (TRUE) {
  try: TSL X, lock  /* test-and-set lock */
  if (X!=0) goto try; /*retry if lock set*/
  CS();
  lock = 0; /* reset the lock */
  Non_CS();
 }
```


TSL solution for multi-processors

- TSL involves testing and setting memory, this can require 2 memory accesses
  - Not a problem to implement this in single-processor system
- Now, bus must be locked to avoid split transaction
  - Bus provides a special line for locking
- A process that fails to acquire lock checks repeatedly issuing more TSL instructions
  - Requires Exclusive access to memory block
  - Cache coherence protocol would generate lots of traffic
- Goal: To reduce number of checks
  1. Exponential back-off: instead of constant polling, check only after delaying (1, 2, 4, 8 instructions)
  2. Maintain a list of processes waiting to acquire lock.

Busy-Waiting vs Process switch

- In single-processors, if a process is waiting to acquire lock, OS schedules another ready process
- This may not be optimal for multiprocessor systems
  - If OS itself is waiting to acquire ready list, then switching impossible
  - Switching may be possible, but involves acquiring locks, and thus, is expensive
- OS must decide whether to switch (choice between spinning and switching)
  - spinning wastes CPU cycles
  - switching uses up CPU cycles also
  - possible to make separate decision each time locked mutex encountered

29

Multiprocessors: Summary

- Set of processors connected over a bus with shared memory modules
- Architecture of bus and switches important for efficient memory access
- Caching essential; to manage multiple caches, cache coherence protocol necessary (e.g. Snoopy)
- Symmetric Multiprocessing (SMP) allows OS to run on different CPUs concurrently
- Synchronization issues: OS components work on shared data structures
  - TSL based solution to ensure mutual exclusion
  - Spin locks (i.e. busy waiting) with exponential backoff to reduce bus traffic

30

Scheduling

- Recall: Standard scheme for single-processor scheduling
  - Make a scheduling decision when a process blocks/exits or when a clock interrupt happens indicating end of time quantum
  - Scheduling policy needed to pick among ready processes, e.g. multi-level priority (queues for each priority level)
- In multiprocessor system, scheduler must pick among ready processes and also a CPU
- Natural scheme: when a process executing on CPU k finishes or blocks or exceeds its time quantum, then pick a ready process according to scheduling policy and assign it to CPU k. But this ignores many issues…

31

Issues for Multiprocessor Scheduling

- If a process is holding a lock, it is unwise to switch it even if time quantum expires
- Locality issues
  - If a process p is assigned to CPU k, then CPU k may hold memory blocks relevant to p in its cache, so p should be assigned to CPU k whenever possible
  - If a set of threads/processes communicate with one another then it is advantageous to schedule them together
- Solutions
  - Space sharing by allocating CPUs in partitions
  - Gang scheduling: scheduling related threads in same time slots

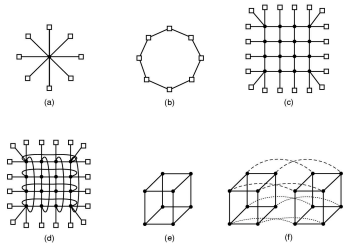
32

Multicomputers

Multicomputers

- Definition:
  Tightly-coupled CPUs that do not share memory
- Communication by high-speed interconnect via messages
- Also known as
  - cluster computers
  - clusters of workstations (COWs)

Clusters



- Interconnection topologies

(a) single switch
(b) ring
(c) grid
(d) double torus
(e) cube
(f) Hypercube (2**d, d is dimeter)

Switching Schemes

- Messages are transferred in chunks called packets
- Store and forward packet switching
  - Each switch collects bits on input line, assembles the packet, and forwards it towards destination
  - Each switch has a buffer to store packets
  - Delays can be long
- Hot-potato routing: No buffering
  - Necessary for optical communication links
- Circuit switching
  - First establish a path from source to destination
  - Pump bits on the reserved path at a high rate
- Wormhole routing
  - Split packet into subpackets to optimize circuit switching

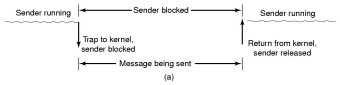
Interprocess Communication

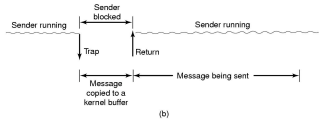
- How can processes talk to each other on multi-computers?
  - User-level considerations: ease of use etc
  - OS level consideration: efficient implementation
- Message passing
- Remote procedure calls (RPC)
- Distributed shared memory (DSM)

Message-based Communication

- Minimum services provided
  - send and receive commands
- These are blocking (synchronous) calls

(a) Blocking send call





(b) Nonblocking send call

User-level Communication Primitives

- Library Routines
  - Send (destination address, buffer containing message)
  - Receive (optional source address, buffer to store message)
- Design issues
  - Blocking vs non-blocking calls
  - Should buffers be copied into kernel space?

Blocking vs Non-blocking

- Blocking send: Sender process waits until the message is sent
  - Disadvantage: Process has to wait
- Non-blocking send: Call returns control to sender immediately
  - Buffer must be protected
- Possible ways of handling non-blocking send
  - Copy into kernel buffer
  - Interrupt sender upon completion of transmission`
  - Mark the buffer as read-only (at least a page long), copy on write
- Similar issues for handling receive calls

Buffers and Copying

- Network interface card has its own buffers
  - Copy from RAM to sender's card
  - Store-and-forward switches may involve copying
  - Copy from receiver's card to RAM
- Copying slows down end-to-end communication
  - Copying not an issue in disk I/O due to slow speed
- Additional problem: should message be copied from sender process buffer to kernel space?
  - User pages can be swapped out
- Typical solutions
  - Programmed I/O for small packets
  - DMA for large messages with disabling of page replacement

41

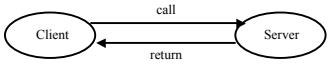
The Problem with Messages

- Messages are flexible, but
- They are not a natural programming model
  - Programmers have to worry about message formats
  - messages must be packed and unpacked
  - messages have to be decoded by server to figure out what is requested
  - messages are often asynchronous
  - they may require special error handling functions

42

Remote Procedure Call

- Procedure call is a more natural way to communicate
  - every language supports it
  - semantics are well defined and understood
  - natural for programmers to use
- Basic idea of RPC (Remote Procedure Call)
  - define a server as a module that *exports* a set of procedures that can be called by client programs.



43

A brief history of RPC

- Birrell and Nelson in 1980, based on work done at Xerox PARC.
- Similar idea used in RMI, CORBA or COM standards
- Core of many client-server systems
- Transparency is to goal!

44

Remote Procedure Call

- Use procedure call as a model for distributed communication
- RPCs can offer a good programming abstraction to hide low-level communication details
- Goal - make RPC look as much like local PC as possible
- Many issues:
  - how do we make this invisible to the programmer?
  - what are the semantics of parameter passing?
  - how is binding done (locating the server)?
  - how do we support heterogeneity (OS, arch., language)?
  - how to deal with failures?
  - etc.

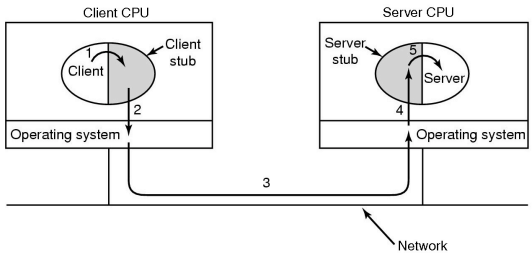
45

Steps in Remote Procedure Calls

- There are 3 components on each side:
  - a user program (client or server)
  - a set of *stub* procedures
  - RPC runtime support
- Steps in RPC
  - Client invokes a library routine called client stub, possibly with parameters
  - Client stub generates a message to be sent: parameter marshaling
  - Kernels on client and server handle communication
  - Receiver kernel calls server stub
  - Server stub unpacks parameters and invokes server routine

46

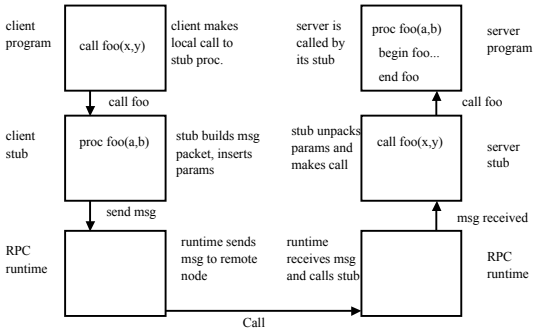
Remote Procedure Call



- Steps in making a remote procedure call
  - the stubs are shaded gray

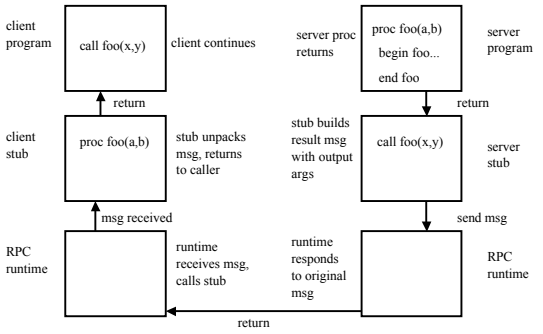
47

RPC Call Structure



48

RPC Return Structure



RPC Stubs

- A client-side stub is a procedure that looks to the client as if it were a callable server procedure.
- A server-side stub looks to the server as if it's a calling client.
- The stubs send messages to each other to make the RPC happen.
- Server program defines the server's interface using an *interface definition language* (IDL)
  - Define names, parameters, and types
- A *stub compiler* reads the IDL and produces two stub procedures for each server procedure
- The server program links it with the server-side stubs; the client program links with the client-side stubs.

RPC Parameter Marshalling

- The packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshall (or unmarshall) all of the parameters to the call.
- Representation needs to deal with byte ordering issues
  - Different data representatin (ASCII, UNICODE, EBCDIC)
  - big-endian (bytes from left to rigth, Intel) versus little-endian (bytes from right to left, SPARC)
  - strings (some CPUs require padding)
  - alignment, etc.
- Parameter passing
  - By value
  - By reference
  - Size limit?

RPC failure semantics

- A remote procedure call makes a call to a remote service look like a local call
  - RPC makes transparent whether server is local or remote
  - RPC allows applications to become distributed transparently
  - RPC makes architecture of remote machine transparent
- What if there is a failure?
- Goal: Make RPC behave like local procedure call

Types of failure

- Cannot locate the server
  - server down
  - version mismatch
  - raise an exception
- Request message is lost
- Reply message is lost
- Server crashes after receiving a request
- Client crashes after sending a request

53

Handling message failure

- request msg is lost
  - use timer and resend request msg
- reply msg is lost
  - use timer and resend another request
  - server need to tell whether the request is duplicate unless the request is idempotent
    - make all request idempotent
      - redefine read (fd, buf, n) to read (fd, buf, pos, n)
      - deposit (money) -- not possible to make it idempotent
    - assign request numbers and keep track

54

Possible semantics to deal with crashes

- Do nothing and leave it up to the user
- At least once
  - Successful return
    - Executed at lease once.
  - Only for idempotent functions
- At most once
  - Suppress duplicated requests
  - Client
    - Each request has an unique id
  - Server
    - Saves request results
- Exactly once (not possible to implement)

55

Shared memory vs. message passing

- Message passing
  - better performance
  - know when and what msgs sent: control, knowledge
- Shared memory
  - familiar
  - hides details of communication
  - no need to name receivers or senders, just write to specific memory address and read later
  - caching for "free"
  - porting from centralized system (the original "write once run anywhere")
  - no need to rewrite when adding processs, scales because adds memory for each node
  - Initial implementation correct (agreement is reached at the memory system level), all changes are just optimizations

56

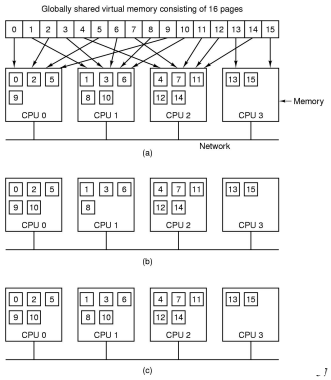
Distributed Shared Memory (DSM)

Replication

(a) Pages distributed on 4 machines

(b) CPU 0 reads page 10

(c) CPU 1 reads page 10



_ 7

Distributed Shared Memory (DSM)

- data in shared address space accessed as in traditional VM.
- mapping manager -- maps the shared address space to the physical address space.
- Advantage of DSM
  - no explicit comm. primitives, send and receive, needed in program. It is believed to be easier to design and write parallel alg's using DSM
  - complex data structure can be passed by reference.
  - moving page containing the data take advantage of locality and reduce comm. overhead.
  - cheaper to build DSM system than tightly coupled multiprocessor system.
  - scalable -- improved portability of programs written for multiprocessors.

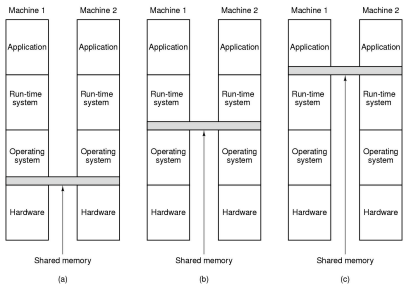
58

DSM Implementation Issues

- Recall: In virtual memory, OS hides the fact that pages may reside in main memory or on disk
- Recall: In multiprocessors, there is a single shared memory (possibly virtual) accessed by multiple CPUs. There may be multiple caches, but cache coherency protocols hide this from applications
  - how to make shared data concurrently accessible
- DSM: Each machine has its own physical memory, but virtual memory is shared, so pages can reside in any memory or on disk
  - how to keep track of the location of shared data
- On page fault, OS can fetch the page from remote memory
  - how to overcome comm. delays and protocol overhead when accessing remote data

59

Distributed Shared Memory



- Note layers where it can be implemented
  - hardware
  - operating system
  - user-level software

60

Some Implementation Details

- Every computer has its own page-table
- If accessed page is not in memory, then message sent to lookup where it resides, and the page is fetched
  - The client-server algorithm
  - The migration algorithm
- Replication used to reduce the traffic
  - The read-replication algorithm
  - The full-replication algorithm
- As in cache coherence for multiple caches in a multiprocessor system, a page can reside on multiple computers with read-only flag set
- To write a page other copies must be invalidated
- False sharing: No variables actually shared, but they may reside on the same page
  - Compiler should make an effort to avoid this

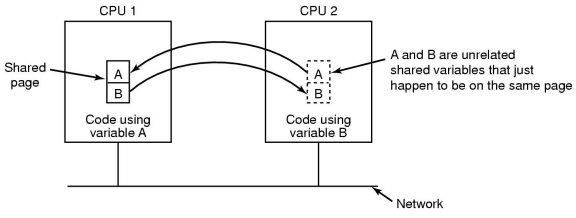
61

Cache/Memory Coherence and Consistency

- Coherence: every cache/CPU must have a coherent view of memory
  - If P writes X to A, then reads A, if no other proc writes A, then P reads X
  - If P1 writes X to A, and no other processor writes to A, then P2 will eventually read X from A.
  - If P1 writes X to A, and P2 writes Y to A, then *every* processor will either read X then Y, or Y then X, but all will see the writes in the same order.
- Consistency: memory consistency model tells us when writes to *different* locations will be seen by readers.

62

False sharing in DSM



- False Sharing
- Must also achieve sequential consistency

63

Load Balancing

- In a multicomputer setting, system must determine assignment of processes to machines
- Formulation as an optimization problem:
  - Each process has estimated CPU and memory requirements
  - For every pair of processes, there is an estimated traffic
- Goal: Given k machines, cluster the processes into k clusters such that
  - Traffic between clusters is minimized
  - Aggregate memory/CPU requirements of processes within each cluster are evenly balanced (or are below specified limits)

64

Algorithms for Load Balancing

- Finding optimal allocation is computationally expensive
  - NP-hard (must try all possible combinations in the worst case)
  - Must settle for greedy heuristics that perform well
- Dynamic adaptation schemes
  - Sender Initiated Schemes
    - Assign processes by default choices
    - If one machine senses overload, probe others for help
    - If load is low, respond to probes, and accept process migration
  - Receiver Initiated Schemes
    - When load is low, probe other machines if they are loaded with processes
    - Probe traffic does not degrade performance during overload

65