Multiprocessor Operating Systems

CS239 Advanced Operating Systems

November 30, 1999
Dr. Richard Guy

Lecture Source Materials

Singhal, Ch. 16:
Multiprocessor System Architectures

Singhal, Ch. 17:
Multiprocessor Operating Systems

Outline

- Introduction/Motivation
- Multiprocessor System Architectures
- Multiprocessor Operating Systems
  - Process Parallelism
  - Process Synchronization/Scheduling
  - Memory Management
  - Reliability and Fault Tolerance
- Summary

Introduction

- Increased computing power
  - faster devices from denser components
  - but, approaching physical limits ~2005
- Use multiple components in parallel
- Try to provide single-system image
  - multiple processors
  - multiple memory modules
  - multiple interconnection components

Key questions

- What are common hardware architectures?
- How does hardware affect OS services?

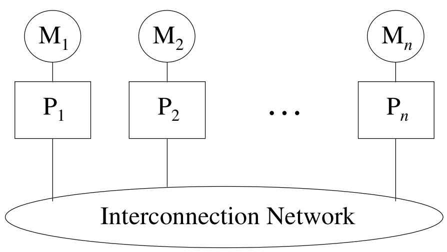
Multiprocessor System Architectures

- Basic architectures
- Interconnection networks
- Caching
- Hypercube architectures
- Architecture summary

Basic architectures

- Tightly- vs. Loosely-Coupled
- UMA vs. NUMA vs. NORMA

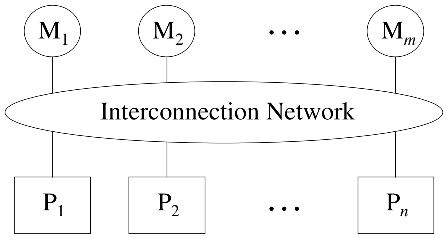
Loosely-coupled systems



Loosely-coupled details

- Main memory partitioned
- Processor-private address space
  - no direct access to remote memory
  - use message-passing
    - interprocessor communication
    - interprocessor synchronization
- Example: Intel Hypercube

Tightly-coupled systems



Tightly-coupled details

- Shared memory address space
- Direct access by all processors
- Use main memory
  - interprocessor communication
  - interprocessor synchronization
- Concurrent access to multiple modules OK
- Memory contention problem

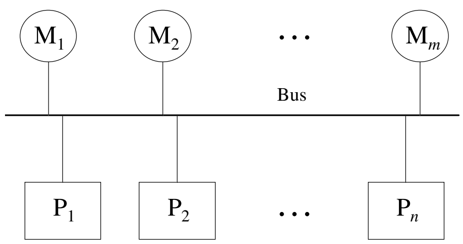
UMA vs NUMA vs NORMA

- UMA: Uniform Memory Access
  - central, “equidistant” memory location
- NUMA: Non-Uniform Memory Access
  - remote access slower than local access
- NORMA: No Remote Memory Access

Interconnection networks

- Bus
- Cross-bar switch
- Multistage interconnection network

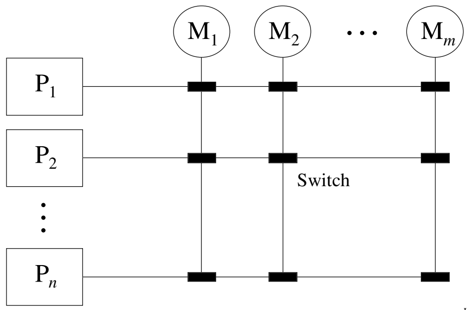
Bus



Bus details

- Simple; easy/cheap to implement
- Bus contention is major issue
- Scale limited by bus bandwidth
  – multiple buses to reduce problem

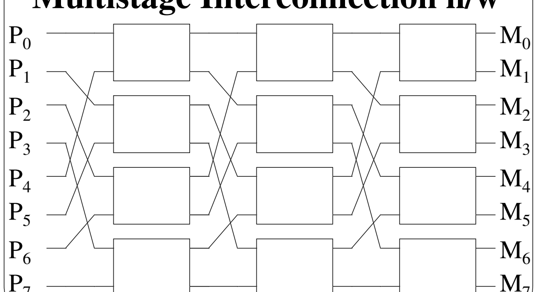
Cross-bar



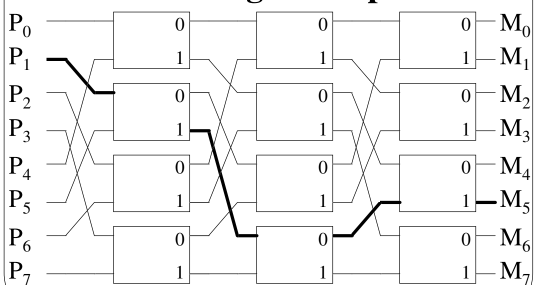
Cross-bar details

- exclusive connection between P_i and M_j
- if $n \leq m$, n concurrent accesses possible from n distinct processors to m distinct memory modules
- only memory module contention
- expensive
- poor scaling (quadratic |switch| growth)

Multistage Interconnection n/w



Multistage example



Multistage details

- concurrent P-M pairs
- cheaper than cross-bar, better than bus
- identical stages of cross-bar switches
- routing path is binary string
- $N \times N$ multistage interconnection network
  - $\log_2 N$ stages of $N/2$ 2×2 switches
  - $(N \log_2 N)/2$ switches vs N^2 in $N \times N$ cross-bar
- switch contention possible

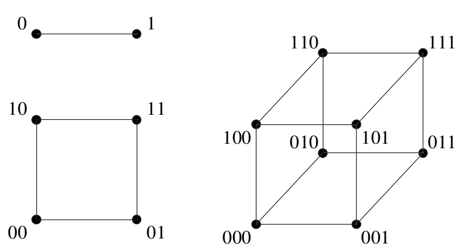
Caching

- Private memory to reduce access time
  - faster than (variable) remote memory access
  - reduces interconnection network traffic
  - reduces memory module contention
  - adds *cache coherence* problem (cf. DSM)
- Cache coherence strategies
  - write-update
  - write-invalidate

Hypercube architectures

- arrange 2^n nodes in n-dimensional cube
  - each node directly connected to n other nodes
  - called n-degree hypercube, or n-cube
  - maximum distance between nodes is n hops
  - scaleable: maximum h/w delay is n
  - uses store-and-forward communication
- node IDs are n-bit values from 0 to 2^n -1
  - assigned so adjacent IDs differ in one bit

Hypercube examples



Architecture Summary

- Basic architectures
  - Tightly- vs. Loosely-Coupled
  - UMA vs. NUMA vs. NORMA
- Interconnection networks
  - Bus, Cross-bar switch, Multistage Intercon. n/w
- Caching issues
- Hypercube architectures

Multiprocessor OSs

- Introduction
- Multiprocessor OS Structures
- Design Issues
- Multiprocessor OS summary

Introduction

- Similar to multitasking uniprocessor OSs
- Physical, not just virtual, concurrency gives added complexity
- Goal: efficiently and effectively exploit the power of multiple processors to support physical concurrent multiple task execution, and degrade gracefully in the face of failure.

Multiprocessor OS Structures

- Separate supervisor
- Master-slave
- Symmetric

Separate supervisor

- each processor has own copy of kernel, supervisor, data structures
- some shared data structures to protect
- very loosely coupled…
- difficult to schedule parallel subtasks
- graceful degradation when processors fail

Master-Slave

- (one) master processor monitors status and assigns work to all other processors
  - slaves only execute application programs
  - single copy of OS
    - easy to implement
    - efficient
- single point of failure: master
- bottleneck: master

Symmetric

- one copy of kernel shared by all processors
- controlled concurrent access to shared data
  - whole OS as critical section: *floating master*
    - execution can become bottleneck
  - partition OS into low-interaction pieces
    - what granularity of pieces???
- degrades gracefully under failure
- very hard to get right

Design Issues

- Process Parallelism
- Process Synchronization
- Processor Scheduling
- Memory Management
- Reliability and Fault Tolerance

Process Parallelism

- Issues
- User-level threads
- Kernel-level threads
- First-class threads

Process Issues

- Traditional process
  - single address space/single thread of control
  - state information
    - page tables
    - swap images
    - file descriptors
    - I/O requests
    - saved register values
- Process is costly to create/maintain/switch

Process Issues, cont.

- Idea: separate execution issues from rest of process state management→thread (lwp)
- Thread
  - private program counter
  - private activation stack
  - control block
  - all other process state shared with other threads
- How/where to support?

User-level threads

- run-time library provides thread managment
- thread mgt op is < 10*procedure call cost

\+ no modifications needed to existing kernel
\+ flexible: many languages, tuned to needs
\- "blocking" system calls stop all threads
\- no scheduling/synchronization coordination

Kernel-level threads

- kernel directly support/manages threads

\+ scheduling/synchronization coordination
\+ frequent kernel interaction is OK
\- more overhead than ULT for mgt operations
\- kernel must provide all (costly) features

- want ULT performance, but kernel involved

First-class threads

- most thread management ops are user-level
- kernel: preemptive scheduling, protection
- avoid kernel traps in ULT↔kernel comm.
- ULT package and kernel share duties/data
  - shared data segments, unidirectional read-only
  - kernel uses s/w interrupts to signal ULT
  - scheduler interfaces for blocking/unblocking

Process Synchronization

- mutual exclusion for shared data access
- busy-waiting can swamp network
- hardware can provide atomic
  - single element read/write
  - simple operation on single element
- non-atomic critical section needs locking
  - can provide with previous hardware
  - avoid busy-waiting with sleep-lock or queueing

Processor Scheduling

- Issues
- Co-scheduling
- Smart scheduling
- NYU Ultracomputer scheduling
- Affinity-based scheduling
- Mach scheduling

Issues

- Preemption inside spinlock-controlled critical sections
  - other waiting (spinning) tasks wait a long time
- Cache corruption
  - task switching to unrelated tasks
- Context switching overheads
  - expensive; causes cache corruption
- Smart scheduling to avoid problems?

Co-scheduling

- two-level scheduling
  - all tasks within application scheduled together for a fixed timeslice
  - on preemption, context switch to another application (prempting all related tasks)
  - avoids spin-lock preemption problem
  - doesn't help other two issues

Smart scheduling

- scheduler knows about critical sections and avoids preemption within critical section
- avoids rescheduling of busy-waiting tasks until critical section is released
- like co-scheduling, fixes spinlock problem, and doesn't help cache corruption or context switching problems

NYU Ultracomputer scheduling

- combines co-scheduling/smart scheduling
- choice of three policies for “task group”
  - normal scheduling
  - group scheduling/preemption
  - no preemption for group
- task can override group preemption policy
- helps spinlock; doesn’t address latter issues

Affinity-based scheduling

- addresses cache corruption
- schedule task where last executed
- limits load balancing

Mach scheduling

- processors divided into disjoint sets
- processor set: local/global priority queues
  - local has preference over global
  - preemption only if higher priority thread ready
  - variable timeslice
- Scheduler hints to improve synchronization
  - discouragement hints: mild, strong, absolute
  - handsoff hints: priority inversion solution

Memory Management

- Issues
- Mach kernel
- Task address space/memory protection
- Machine independence
- Memory sharing
- Efficiency
- Object sharing

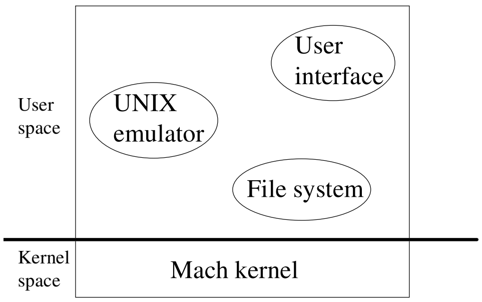
Memory Management Issues

- Portability to distinct architectures
  - MM performance often architecture-specific
- Data sharing(DSM)
- Protection
- Efficiency
  - address translation, page table lookup
  - parallel execution to exploit multiple processors

Mach goals

- for parallel and distributed environments
- runs on multiprocessors
- supports parallel application execution
- OS itself is considered parallel ‘application’
  - algorithms assume parallel execution
  - data structures tuned for highly parallel access
- Mach virtual memory designed to support high performance parallel applications

Mach kernel



CS 239, Fall 1999 Lecture 13 Page 49

Mach kernel details

- Five basic abstractions
  - Tasks and threads
  - Messages and ports
  - Memory objects
- Single paged address space per task
- Memory protection at page level

CS 239, Fall 1999 Lecture 13 Page 50

Machine independent VM

- assume minimal paged MMU
- separate machine dependent service from machine independent services
  - dependent: virtual-to-physical address map
  - independent: everything higher level
- VM implementation decoupled from “OS”
- Paging managed by user-level process

CS 239, Fall 1999 Lecture 13 Page 51

Memory sharing

- all threads in a task share memory objects
- memory objects shareable across tasks
- inheritance mechanism for new tasks

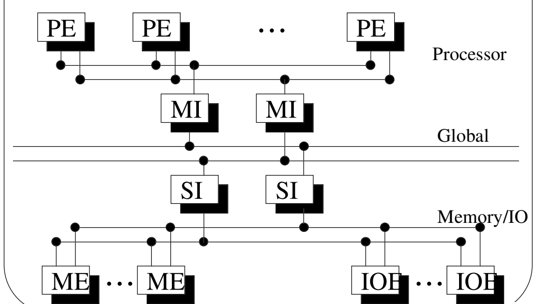
Efficiency

- Parallel implementation/execution
- Simplicity
- Lazy evaluation
  - copy-on-write
  - shadow memory objects

Reliability and Fault Tolerance

- Issues
  - Fault detection and isolation
  - Fault recovery
  - Efficiency
- Sequoia
  - Architecture
  - Fault detection
  - Fault recovery

Sequoia architecture



Sequoia fault detection

- Error detecting codes
  - ***all*** data protected at ***all*** times
- Comparison of duplicated operations
  - duplicate+comparator cheaper than low-level error-detecting code management
- Protocol monitoring
  - detect inter-component sequence/timing protocol violations

Sequoia fault recovery

- Recovery from processor failure
  - careful cache-to-memory flushing
  - shadow pages on distinct MEs
- Recovery from main memory failure
  - shadow pages
    - on separate ME for writable pages
    - on disk for executable/read-only data pages
- Recovery from I/O failure
  - dual ported mirrored disks on separate IOEs

Summary

- Variety of multiprocessor architectures
- Multiprocessor OS has new problems
  - scheduling
  - synchronization
  - memory management
  - reliability/fault tolerance
- Multiprocessor feasible, but complex
  - costly, depending on hardware design