# Operating Systems for Multiprocessors

CS 239, Spring 1995

June 5, 1995

## Outline

- Introductory material
- Process scheduling
- Fault tolerance issues
- Process migration
- I/O and file system issues

## Multiprocessors Vs. Distributed Systems

Why are they any different?

At some level, they aren't

But in practice, they are

Different system solutions are used in multiprocessor systems

## Parallel & Distributed Systems Differences

- Clients served
- Type of jobs
  - And associated data
- Special hardware
- Orders of magnitude difference in interconnect speed/bandwidth

## An Introduction to Parallel Machines

- Motivations
- Architecture types
- Interconnection network types
- Basic problems in parallel systems software

## Why Build Parallel Machines?

- Speed
- Access to greater resources
- Fault tolerance

## Typical Parallel Processor Applications

- Scientific supercomputing
- Simulations
- Cryptographic applications
- Anything requiring lots of processing power

## Parallel Processor Architectures

- SIMD
- MIMD
  - UMA
  - NUMA
  - NORMA

## SIMD Architectures

Single Instruction, Multiple Data

All processors execute the same instructions

But each on a different piece of data

Typical SIMD machines have vast numbers of tiny processors

Useful for certain types of problems

CM-2 is one such machine

## MIMD Machines

Multiple Instructions, Multiple Data

Each processor executes instructions independently

On its own data

Typically composed of several (to hundreds) of commodity processors

And some spiffy way of connecting them

## Connecting MIMD Processors

Tightly coupled versus loosely coupled

Tightly coupled - processors share a memory address space

Loosely coupled - processors do not share an address space

## UMA Parallel Machines

Uniform Memory Access

Any processor can reach any word of memory in the same amount of time

Practically, means memory kept separately from processors

E.g., Encore Multimax, Sequent Balance

## NUMA Parallel Architectures

Non-Uniform Memory Access

For each processor, some memory is “close” and some is “far away”

Typically, each processor has local memory

And can access other processors’ memories

E.g., CMU Cm* and BBN Butterfly

## NORMA Parallel Architectures

NO Remote Memory Access

In other words, loosely coupled

Either build software virtual memory on top

Or use message-passing/RPC paradigm

E.g., Intel Hypercube

## Parallel Interconnection Technologies

Parallel machines are expected to have their nodes communicate a lot

So a fast, high bandwidth interconnect is key

In fact, people build parallel processors out of workstations fitted with a fast interconnect

Key point -

**All parallel architectures provide hardware “fast-path” for bits**

## Parallel Interconnects for NORMA Machines

Typically treat the machine as a special-purpose computer network

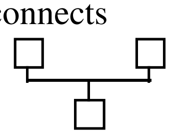
Use very fast routing, forwarding techniques to get messages from one place to another

Speed, congestion are the major issues

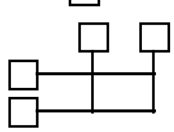
Considerable research in this area

## NUMA Interconnects

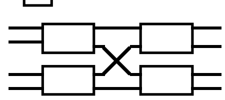
- Bus



- Crossbar switch



- Multistage interconnect



## Caching in Multiprocessors

Caching of data from other processors used aggressively

Since applications are coupled

Cache coherence is a common problem

Write-update and write-invalidate approaches are used to handle coherence

## OS Architecture for Parallel Machines

Optimized for handling tasks that expect to need parallel machine's special resources

Must avoid slowing down data communications

Must avoid wasteful use of each node's resources

## Parallel OS Organization Choices

- Separate supervisor configuration
- Master-slave configuration
- Symmetric configuration

## Separate Supervisor Configurations

Each node has its own copy of the OS

OS copies cooperate to run tasks in parallel

+ Easy model

+ Quick access to system services

+ Degrades gracefully

– Wastes memory

– Hard for tasks to cooperate

## Master/Slave Configurations

One master processor contains most of the system; slaves act on its orders

+ Simple

+ Easy to cooperate on tasks

– Poor failure characteristics

– Master can be a bottleneck

– System services require cross-node communications

## Symmetric Configurations

All processors are equal, and share one copy of the OS

Most common in shared memory systems

Concurrent access to OS a major issue

- Must be correct
- But must avoid bottlenecks
- Typically, divide OS into separately lockable parts

## Symmetric Configurations, Con't

+ Flexible and versatile

+ Permits easy parallel task execution

+ Degrades gracefully

– Complex

– Not suited for NORMA machines

– Dangers of bottlenecks in OS

## Processor Scheduling for Parallel Machines

- More on threads
- Synchronization issues
- Problems in scheduling
- How some systems schedule

## Threads for Parallel Processors

Threads are very important for parallel processors

Particularly those using shared memory

Users must structure their tasks as a collection of cooperating threads

But they don't know how many processors will run them

## User-Level Threads

Implemented (typically) by a library at the user level

Allow users to create and schedule threads within their tasks

Different thread packages can provide different effects

## Pros/Cons of User Level Threads

+ No kernel modification/support required
+ Flexible
+ Low overhead for scheduling
– Requires non-blocking OS traps
– Can't use kernel knowledge to improve scheduling

## Kernel Threads

Threads directly supported in the kernel

Kernel schedules threads

+ Kernel knowledge available
+ Works well when little OS interaction
+ Lower overhead than processes
– Higher overhead than user-level threads
– Tends to lack flexibility

## First-Class Threads

Threads supported as a kernel object

But allows user-space code to do most of their handling (through separate packages)

Typically, kernel and user space share data structures associated with threads

Kernel sends some interrupts to user space

Must support interaction between different user-level thread packages

## Threads and Scheduler Activations

Basically, a method of allowing the user-space to watch kernel calls

And also to provide kernel space for saving context of stopped thread

Kernel signals owning *scheduler activation* when a thread blocks

- Allowing, for example, scheduling of another thread

## Synchronization Issues for Parallel Systems

How to synchronize true parallel programs?

Several choices available -

- Spin locks
- Blocking
- Switch-spinning
- Switch-blocking
- Two-phase algorithms

## Spin Lock Synchronization

Read a memory location till it changes

A feasible operation if -

- It's a local location, or
- It's a shared memory machine

and, in any case, if the wait isn't long

Spin locks burn cycles

## Blocking Synchronization

Block a thread until its synchronization condition is achieved

Blocking is a heavyweight operation

- Even for threads

Suitable if it's a long wait

- But not if it's a short one

## Switch-Spinning Synchronization

Like spin locking, but instead of just testing continuously, switch to other resident threads

Periodically, the waiting thread gets to check again

Context switching cheaper than blocking

- But not that cheap

## Switch-Blocking Synchronization

Thread blocks itself (but does not unload itself)

Other resident threads then start executing

Blocked thread skipped over until it can be unblocked

Blocking without unloading/reloading significantly cheaper

## Two-Phase Synchronization Algorithms

Generally don't know how long a thread must wait for synchronization

Spin locks better for short waits

Blocking better for long waits

Two-phase methods spin lock for a while, then switch to blocking

Trick is to know when to switch

## Scheduling for Parallel Processors

- NORMA scheduling
- UMA/NUMA scheduling

## NORMA Scheduling

At task/thread creation, assign it to a node

Each node runs its own scheduling

Scheduling something outside the node is a heavyweight operation

Load balancing and process migration are important here

## UMA/NUMA Scheduling

Central or distributed scheduler assigns a thread to a node

When it blocks, get another thread

Since all nodes can talk to memory, more flexibility in what threads get scheduled

## Performance Problems in Scheduling for UMA/NUMA

- Preemption during critical sections
- Cache corruption
- Context switching overheads

## Preemption Performance Problems

If a task is preempted while its in a critical section, other tasks waiting for the critical section are unnecessarily delayed

If critical sections small and infrequently used, may not happen often

But cost is big when it does happen

## Cache Corruption Performance Problems

A given node may run many different threads

Unless care is taken, node's cache will not have any useful contents for its next thread

Leading to high startup costs for a thread as it gets lots of early cache misses

At worst, just when it's filled up the cache, the thread is swapped out

## Context Switching Times Performance Problems

Node has full set of internal registers, virtual memory tables, etc.

All of which must be changed when a new thread is scheduled

## Approaches to Parallel Processor Scheduling Problems

- Co-Scheduling
- Smart Scheduling
- Affinity Based Scheduling

## Co-Scheduling

Developed for Medusa on Cm*

Schedule all tasks of an application as a unit

On as many processors as available

Preemption of one task preempts them all

Tends to prevent tasks waiting on critical section

Doesn't do anything for cache corruption, context switch overheads

## Smart Scheduling

Don't preempt tasks when in critical sections

If a preempted task was busy-waiting on a critical section, don't reschedule it

Wait till task holding section releases it

Does nothing for cache corruption, context switch overhead

## Affinity Based Scheduling

Schedule tasks on the processor that last executed them

That processor's cache will probably still have much of the task's data

Saving time, bandwidth when it restarts

Limits flexibility of scheduling tasks to processors

May also need process migration

## Single-Level Scheduling Versus Two-Level Scheduling

In single-level scheduling, kernel schedules all threads

In two-level scheduling, kernel assigns set of processors to an application, which assigns its threads to processors

Two-level scheduling currently viewed as superior

## Scheduling in Mach for Parallel Processors

Two-level priority scheme

Divide processors into *processor sets*

Assign set of threads to processor set

Several priority queues shared by the processor set

And also a local queue of local threads

## Selecting a Thread to Run in Mach

Check the local queue first

If no ready threads there, choose the first ready thread on the highest priority queue

When a thread of given priority uses up its timeslice, only relinquish processor if higher priority thread is ready

If no ready threads available, run idle thread until one is

## Scheduling Multiple Jobs in a Parallel Processor

A parallel machine of a given size may need to run multiple jobs with multiple threads

Who gets how many processors?

Does a job keep all its processors till completion?

Can it get more processors?

How do you take processors away?

## Allocating Processors

Many parallel programs have varying amounts of effective parallelism

Sensible scheduling policy is to give them more processors when they have more parallelism

And fewer when they have less

## Reallocating Processors

Studies show that dynamic scheduling that reallocates processors performs much better

Also improves fairness

But having applications assist in moving preempted processors is critical

- What if you grab a processor holding a thread that's spinning on a lock?

## Fault Tolerance in Parallel Processors

Parallel processors have potential for high fault tolerance

What needs to be done to achieve potential?

- Fault detection and isolation
- Fault recovery
- Efficiency

## Sequoia - A Fault Tolerant Parallel Processor

Sequoia achieves fault tolerance through a combined hardware/software approach

Redundant buses connect processor and memory elements

Processor elements contain two identical processors

Memory elements contain memory and locking hardware

## Sequoia Error Detection Hardware

Single error detecting codes on all data in all locations

Memory, processor cache, bus, etc.

Hardware organized so single component failure can produce only single bit per byte error

Half of each word protected by odd parity, half by even parity

## Sequoia Hardware to Compare Duplicate Operations

Error detection too expensive for some components (e.g., processors)

Such components are duplicated

All operations performed at both duplicates

The results checked against each other

## Sequoia Protocol Monitoring

Used to detect problems related to timing and communications

Hardware knows what results of certain operations should be and monitors them

## What if the Fault Detection Hardware Fails?

Sequoia's OS checks the fault detection circuitry periodically

Memory and I/O probed

Processors required to periodically update a special block

Master processor checks blocks of other processors

Other processors watch master processor

## What if a Fault Happens?

- Disable the outputs of the faulty component
- Notify the operating system of the faulty component
- Initiate fault recovery

## Fault Recovery in Sequoia

- Sequoia must ensure that any fault can be overcome
- Meaning affected processes must be returned to a consistent state
- For processor, memory, I/O component failures

## Recovering From Processor Failure

- Any important changes in a process' state are stored in its cache (plus registers)
- Dirty blocks are flushed periodically
- All dirty blocks (plus registers) flushed at once
- What if it fails during a flush?
  - Shadow paging of flushed blocks

## Recovering From Main Memory Failure

- Writable memory blocks are <u>always</u> duplicated
- Read/execute only memory blocks backed up on disk
- So failure of one memory component cannot destroy any memory blocks

## Recovering From I/O Failures

- Disk failures handled by using dual-ported mirrored disks
  - Writes go to both, reads are load balanced between them
- If other component in path fails, try to find alternate path
- Otherwise, use the other mirrored disk

## So What Can Sequoia Do?

- Transparently continue execution of a parallel program even if a processor fails
- Guarantee that all data will be accessible regardless of single component failure