

Thread Implementation

Processes or Threads

- A process or thread is a potentially-active execution context
- Processes/threads can come from
 - Multiple CPUs
 - Kernel-level multiplexing of single physical CPU (kernel-level threads or processes)
 - Language or library-level multiplexing of kernel-level abstraction (user-level threads)
- Threads can run
 - Truly in parallel (on multiple CPUs)
 - Unpredictably interleaved (on a single CPU)
 - Run-until-block (coroutine-style)

Processes Vs. Threads

- Process
 - Single address space
 - Single thread of control for executing program
 - State information
 - Page tables, swap images, file descriptors, queued I/O requests, saved registers
- Threads
 - Separate notion of execution from the rest of the definition of a process
 - Other parts potentially shared with other threads
 - Program counter, stack of activation records, control block (e.g., saved registers/state info for thread management)
 - Kernel-level (lightweight process) handled by the system scheduler
 - User-level handled in user mode

Thread Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

```
save all callee-saves registers on stack, including ra
and fp
*current := sp
current := other
sp := *current
pop all callee-saves registers (including ra, but NOT
sp!)
return (into different coroutine!)
```

Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)
- ```
reschedule:
 t : cb := dequeue(ready_list)
 transfer(t)
yield:
 enqueue(ready_list, current)
 reschedule
sleep_on(q):
 enqueue(q, current)
 reschedule
```

## Preemption

- Use timer interrupts or signals to trigger involuntary yields
  - Protect scheduler data structures by disabling/reenabling prior to/after rescheduling
- ```
yield:
  disable_signals
  enqueue(ready_list, current)
  reschedule
  re-enable_signals
```

Multiprocessor Scheduling

- Disabling signals not sufficient
- Acquire scheduler lock when accessing any scheduler data structure, e.g.,

yield:

```
disable_signals
acquire(scheduler_lock) // spin lock
enqueue(ready_list, current)
reschedule
release(scheduler_lock)
re-enable_signals
```

Anderson et al.

- Raises issues of
 - Locality (per-processor data structures)
 - Granularity of scheduling tasks
 - Lock overhead
 - Tradeoff between throughput and latency
 - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

Performance Measures

- Latency
 - Cost of thread management under the best case assumption of no contention for locks
- Throughput
 - Rate at which threads can be created, started, and finished when there is contention

Optimizations

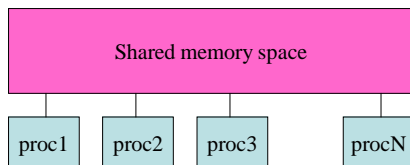
- Allocate stacks lazily
- Store deallocated control blocks and stacks in free lists
- Create per-processor ready lists
- Create local free lists for locality
- Queue of idle processors (in addition to queue of waiting threads)

Ready List Management

- Single lock for all data structures
- Multiple locks, one per data structure
- Local freelists for control blocks and stacks, single shared locked ready list
- Queue of idle processors with preallocated control block and stack waiting for work
- Local ready list per processor, each with its own lock

Shared Memory: Synchronization, Coherence, and Consistency

Shared Memory: A Look Underneath



Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
 - ensuring that modifications made by a processor propagate to all copies of the data
 - Program order preserved
 - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
 - defines when and in what order modifications are propagated to other processors

Synchronization

- Basic types
 - Mutual exclusion
 - Events
- Components of a synchronization operation
 - Acquire method (enter critical section, proceed past event)
 - Waiting algorithm (busy waiting, blocking)
 - Release method (enable others to proceed)

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Spinlock Algorithms

- Test&test&set (w , w/o exponential backoff)
- Ticket lock (w , w/o proportional backoff)
- Array based queue locks
- MCS linked-list based queue locks

Performance Goals

- Low latency, short critical path
- Low traffic
- Scalability
- Low storage cost
- Fairness

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```

Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4    ; mov exchange value
     ll  R2, 0(R1) ; load linked
     sc R3, 0(R1)  ; store conditional
     beqz R3, try  ; branch if store fails
     mov R4, R2   ; put load value in R4
```

MCS Lock Acquire

```
mcs_lock_acquire:
    st %g0, [%o1+4]
    mov %o1, %g3
    swap [%o0], %g3
    cmp %g3, 0
    be .LL4
    mov l, %g2
    st %g2, [%o1]
    st %o1, [%g3+4]
.LL9:
    ld [%o1], %g2
    cmp %g2, 0
    bne .LL9
    nop
.LL4:
    retl
    nop
```

MCS Lock Release

```
mcs_lock_release:
    ld [%o1+4], %g2
    cmp %g2, 0
    bne .LL11
    nop
    cas [%o0], %o1, %g2
    cmp %g2, %o1
    be .LL10
    nop
.LL17:
    ld [%o1+4], %g2
    cmp %g2, 0
    be .LL17
    nop
.LL11:
    st %g0, [%g2]
.LL10:
    retl
    nop
```

Properties

- Lock-free data structures
 - Operations defined on it do not require mutual exclusion over multiple instructions (use atomic primitives)
- Non-blocking algorithms
 - Operations guarantee that some process will complete its operation a finite amount of time, even if other processes halt
- Wait-free algorithms
 - Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

Barrier Algorithms

- Centralized sense-reversing barrier
- Software combining tree
- Tournament barrier
- Dissemination barrier
- Combining tree with improved locality

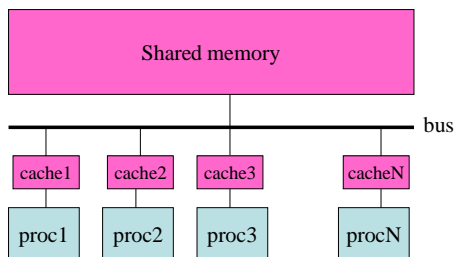
Shared Memory Implementation

- Coherence - defines the behavior of reads and writes to the same memory location
 - ensuring that modifications made by a processor propagate to all copies of the data
 - Program order preserved
 - Writes to the same location by different processors serialized
- Synchronization - coordination mechanism
- Consistency - defines the behavior of reads and writes with respect to access to other memory locations
 - defines when and in what order modifications are propagated to other processors

Coherence

- A multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the result of the execution and
- it ensures that modifications made by a processor propagate to all copies of the data
 - program order is preserved for each process in this hypothetical order
 - writes to the same location by different processors are serialized and the value returned by each read is the value written by the last write in the hypothetical order

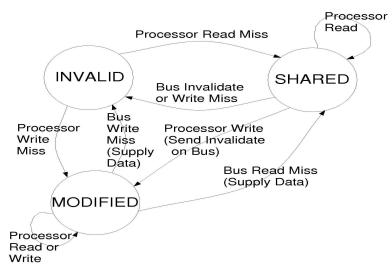
Physical Implementation



Snoop-Based Coherence

- Makes use of a shared broadcast medium to serialize events (all transactions visible to all controllers and in the same order)
 - Write update-based protocol
 - Write invalidate-based (e.g., basic MSI, MESI protocols)
- Cache controller uses a finite state machine (FSM) with a handful of stable states to track the status of each cache line
- Consists of a distributed algorithm represented by a collection of cooperating FSMs

A Simple Invalidate-Based Protocol - State Transition Diagram



Correctness Requirements

- Need to avoid
 - Deadlock – caused by a cycle of resource dependencies
 - Livelock – activity without forward progress
 - Starvation – extreme form of unfairness where one or more processes do not make forward progress while other do

Directory-Based Coherence

- Distribute memory, use point-to-point interconnect for scalability
- Need to manage coherence for each memory line – state stored in directory
 - Simple memory-based (e.g., DASH, FLASH, SGI Origin, MIT Alewife, HAL)
 - Cache-based (linked list (e.g., Sequent NUMA-Q, IEEE SCI)

Simple Memory-based Directory Coherence

- Advantage
 - Precise sharing information
- Disadvantage
 - Space/storage proportional to P x M
- Work-around for either width or height
 - Increase cache block size
 - 2-level protocol
 - Limited pointer scheme
 - Directory cache

Cache-Based Directory Coherence

- Home main memory contains a pointer to the first sharer + state bits
- Pointers at each cache line to maintain a doubly-linked list
- Advantage – reduced space overhead
- Disadvantage – serialized invalidates (latency and occupancy)

A Framework for Sharing Patterns

- Predictable vs. unpredictable
- Regular vs. irregular
- Coarse vs. fine-grain (contiguous vs. non-contiguous in the address space)
- Near-neighbor vs. long range in an interconnection topology
- In terms of invalidation patterns
 - Read-only
 - Producer-consumer
 - Broadcast/multicast
 - Migratory
 - Irregular read-write

Memory Consistency Models

When must a processor see a value that has been updated by another processor?

P1: A = 0;	P2: B = 0;
...	...
A = 1;	B = 1;
L1: if (B == 0) ...	L2: if (A == 0) ...

Sequential Consistency

- "A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lampport 79]
 - In practice, this means that every write must be seen on all processors before any succeeding read or write can be issued

Consistency Model Classification

- Models vary along the following dimensions
 - Local order - order of a processor's accesses as seen locally
 - Global order - order of a single processor's accesses as seen by each of the other processors
 - Interleaved order - order of interleaving of different processor's accesses on other processors