Synchronization Constructs

Synchronization Constructs Overview

1. Limitation of mutexes and condition variables
    - Error prone/correctness/ease of use
        - Unlock wrong mutex, signal wrong condition variable
    - Lack of expressive power
        - Helper variables for access or priority control
    - Requires low level support
        - Hardware provides atomic instructions
2. May repeatedly check to continue
    - Achieving synchonization through spinlocks
3. Processes may wait for a signal to continue
    - Synchronization through mutexes and condition variables
4. Waiting hurts performance
    - CPUs waste cycles for checking; cache effects

Spinlocks

1. Spinlock is like a mutex
    - Provide mutual exclusion for critical section
    - Lock and unlock (free)
2. With spinlocks, the process doesn't give up the CPU (burns cycles)
    - Mutexes relinquish the CPU
3. Basic synchronization primitive used to implement more complex constructs

Semaphores

1. Like a traffic light; STOP and GO
2. Similar to a mutex, but more general
3. Represented as an integer value
    - Initialized as some maximum value (positive int)
    - When attempted to acquire (wait)...
        - If non-zero, decrement and proceed -> counting semaphore
    - If initialized with 1 == mutex (binary semaphore)
    - On exit (post), increment
4. POSIX semaphores
    - sem_t sem;
    - sem_init(sem_t* sem, int pshared, int count);
    - sem_wait(sem_t* sem);
    - sem_post(sem_t* sem);

Reader/Writer Locks

1. Synchronizing different types of accesses
    - Reading (never modify)
        - Shared access
    - Write (always modify)
        - Exclusive access
2. RW Locks
    - Specify the type of access; underneath, the lock behaves accordingly
    - read/write == shared/exclusive
3. API:
    - rwlock_t m;

- read_lock(m);
    - read_unlock(m);
    - write_lock(m);
    - write_unlock(m);
4. Semantic differences
    - Recursive read_lock -> what happens on read_unlock?
    - Upgrade/downgrade priority?
    - Interaction with scheduling policy
        - Block if higher priority writer waiting

Monitors

1. Monitors were developed for MESA by XEROX PARC
    - In Java, synchronized methods generate monitor code, but must call notify explicitly
    - Monitor also refers to the programming style of using mutexes to enter/exit a critical section
2. Monitors are a high-level synchronization construct that specify…
    - Shared resource
    - Entry procedure
    - Possible condition variables
3. On entry…
    - lock, check, …
4. On exit
    - unlock, check signal, …

Other Synchronization Constructs

1. Serializers: Define priorities and hide need for signalling and condition variables
2. Path expressions: Specify regular expression that captures the correct synchronization behavior ("many reads or a single write")
3. Barriers: Opposite of semaphore; wait for n threads to arrive before proceeding
4. Rendezvous points: Wait for multiple threads to reach a particular point
5. Optimistic wait-free sync: No conflict due to concurrent writes and safe to allow reads to proceed concurrently (read-copy-update (RCU))
6. All of these require hardware support for atomic access to a memory location

Spinlock as a Building Block

1. Spinlock: Basic synchronization construct
2. The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors; Anderson
    - Alternative implementation of spinlocks
    - Generalize techniques to other constructs
3. Not possible to implement solely in software; requires hardware support
    - Concurrent check/update on different CPUs can overlap

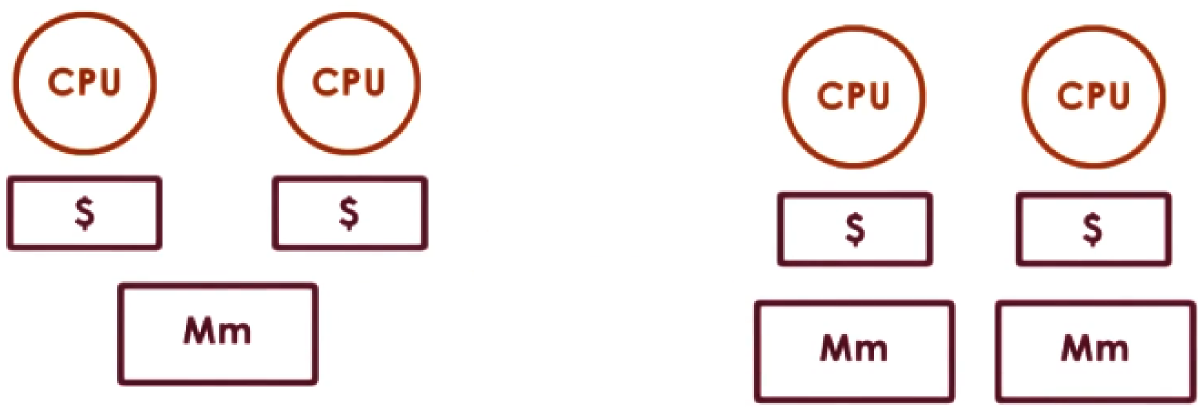
Atomic Instructions

1. Hardware-specific
    - test_and_set
    - read_and_increment
    - compare_and_swap
2. Guarantees
    - Atomicity
    - Mutual exclusion
    - Queue all concurrent instructions but one
3. Atomic instructions: Critical section with hardware-supported synchronization

    - Software using atomics must be ported between hardware
        - Efficiency varies among atomic operations, so need to ensure that implementation is optimal for particular hardware
4. spinlock_lock(lock):
    - while(test_and_set(lock) == busy);
    - test_and_set(lock) atomically returns (tests) original value and sets new value = 1 (busy)
    - First thread: test_and_set(lock) -> 0 (free)
    - Next thread(s): test_and_set(lock) -> 1 (busy)
        - Will set the lock to 1 each time, but that's okay

Shared Memory Multiprocessors

1. Multiple CPUs with block of shared memory
    - Can be single block or multiple with interconnection between them
2. Also referred to as symmetric multiprocessors or SMPs
3. Caches high memory latency; memory "further away" due to contention
    - No write, write-through (back to cache), write-back (waits to write back to memory until cache line is evicted)



Shared Memory Multiprocessors

Cache Coherence

1. Non-cache-coherent (NCC) vs cache-coherent (CC)
    - NCC means a change in the cache of one CPU won't be reflected elsewhere and must be handled in software
    - CC means the hardware handles this
2. Write-invalidate (WI): Mark values as invalid when changed; future references will result in a cache miss
    - Pros: Lower bandwidth, amortize cost (don't need to invalidate twice)
3. Write-update (WU): Values in other caches will be updated when a change occurs in a different CPU
    - Pros: Update available immediately
4. WI vs WU is determined by hardware (programmer doesn't have a choice)
5. Difficult to determine if an atomic operation on one CPU has also been applied to another CPU
    - Important to maintain atomicity though
    - Solution: Atomics always issued to the memory controller
    - Pros: -Can be ordered and synchronized
    - Cons:
        - Takes much longer (must go to memory, no caching)
        - Generates coherence traffic regardless of change

- Atomics and SMP
  - Expensive because of bus or interconnection (I/C) contention
  - Expensive because of cache bypass and coherence traffic

Spinlock Performance Metrics

1. Reduce Latency
   - "Time to acquire a free lock" - Ideally immediately
2. Reduce Waiting Time (delay)
   - "Time to stop spinning and acquire a lock that has been freed" - Also ideally immediately
3. Reduce Contention (bus/network interconnection traffic)
   - Ideally zero

Test and Set Spinlock

1. spinlock_lock(lock):
   - while(test_and_set(lock) == busy);
2. Latency is minimal (just the atomic operation) Good
3. Delay is potentially minimal (spinning continuously on atomic) Good
4. Contention (processors go to memory on each spin) Bad
   - Spinning on an atomic is expensive

Test and Test and Set Spinlock

1. Test the cached value; only try to execute and test_and_set if the cached value indicates that the lock is free
2. spinlock_lock(lock):
   - while((lock == busy) OR (test_and_set(lock) == busy))
3. Also referred to as spin on read and spin on cached value
4. Latency is good, but not as good as test and set
5. Delay is good, but not as good as test and set
6. Contention is better, but...
   - NCC - No difference
   - CC-WU - Improved
   - CC-WI - Worse (continually invalidates the cache every spin)
     - Contention due to atomic + invalid caches == more contention
     - Everyone sees lock is free at same time
     - Everyone tries to acquire the lock at the same time

Spinlock "Delay" Alternatives

1. Delay after lock release
   - Everyone sees lock is free, but not everyone attempts to acquire it

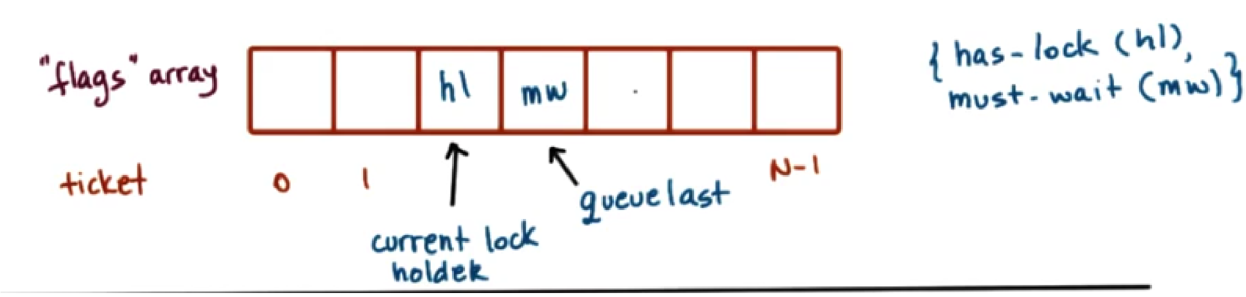
```
spinlock_lock(lock):
    while((lock == busy) || (test_and_set(lock) == busy)) {
        while(lock == busy());
        delay();
    }
```

2. Latency is decent
3. Delay is much worse
4. Contention is improved
5. Alternative: Delay after each lock reference
   - Doesn't spin constantly
   - Works on NCC architectures

    - Can make delay even worse
6. What's the correct delay?
    - Static delay (based on fixed value, e.g. CPU ID)
        - Pros: Simple approach
        - Cons: Unnecessary delay under low contention
    - Dynamic delay (backoff-based; random delay in a range that increases with "perceived" contention
        - Perceived == failed test_and_set(); delay after each reference will keep growing based on contention or length of critical section

Queueing Lock

1. Common problems in spinlock implementations:
    - Everyone tries to acquire a lock at the same time once lock is freed
        - Delay alternatives
    - Everyone sees the lock is free at the same time
        - Anderson's Queueing Lock
2. "Flags" array where each value is has-lock (HL) or must-wait (MW)
3. Hold pointers to current lock holder and last process in queue
4. Set unique ticket for arriving thread
5. Assigned queue[ticket] is private lock
6. Enter critical section when you have the lock:
    - queue[ticket] == must_wait -> spin
    - queue[ticket] == has_lock -> enter critical section
7. Signal/set next lock holder on exit
    - queue[ticket+1] = has_lock
8. Downsides:
    - Assumes read_and_increment atomic (not as common as test_and_set)
    - O(N) size



Anderson's Queueing Lock

Queueing Lock Implementation

```
init:
    flags[0] = has-lock
    flags[1:p-1] = must-wait
    queuelast = 0; // global variable

lock:
    myplace = read_and_increment(queuelast); // get ticket
    // spin
    while(flags[myplace mod p] == must-wait)
    // now in critical section
    flags[myplace mod p] = must-wait;
```

```
unlock:
    flags[myplace+1 mod p] = has-lock;
```

1. Latency: Uses more costly read_and_increment (Bad)
2. Delay: Directly signal next CPU/thread to run (Good)
3. Contention: Better than alternatives but requires cache coherence and cacheline aligned elements (Good)
4. Only 1 CPU/thread sees the lock is free and tries to acquire!
5. Each array element must be in a separate cacheline; total size required is (number of CPUs) * (size of cacheline)

Spinlock Performance Comparisons

1. Setup:
    - N processes running critical section 1M times
    - N varied based on system
2. Metrics:
    - Overhead compared to ideal performance
    - Theoretical limit based on number of critical sections to be run
3. Under high loads:
    - Queue best (most scalable), test_and_test_and_set worst
    - Static better than dynamic, delaying after each reference is better than delaying after each release (avoids extra invalidations)
4. Under light loads:
    - test_and_test_and_set performs well (low latency)
    - Dynamic better than static (lower delay)
    - Queueing lock worst (high latency due to read_and_increment)

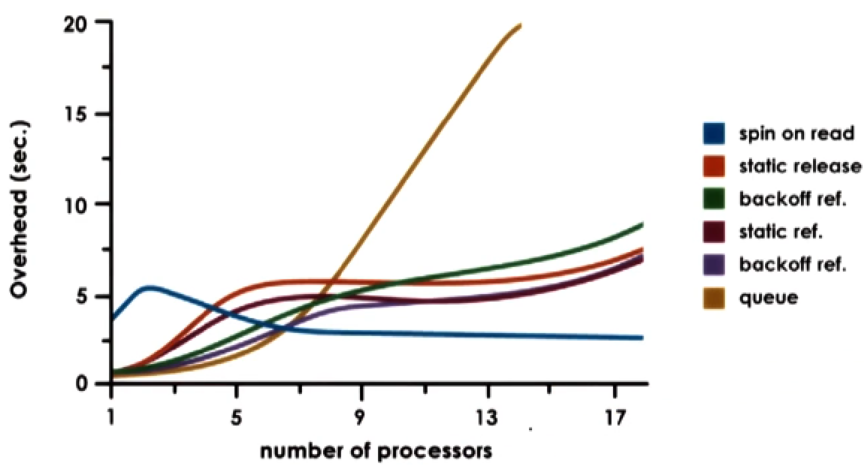


Fig. 3 - Principle performance comparison: spin-waiting overhead (seconds) in executing benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

Spinlock Performance Comparisons