**CS 153**

**Design of Operating Systems**

**Ch. 6 Implementing High-Level Coordination**

Three fundamental issues in passive waiting:

* implementing and handling primitive locks
* queue handling
* passing a processor from one thread to another

**6.1 Low-Level Locks**

* block locks
* spin locks
* number locks

**6.1.1 Block Locks**

hoard the processor cycles; after a lock has been acquired, no thread other than the lock’s holder gets any processor cycles or request the lock

no assurance of other threads not acquiring the lock when it is released

while holding this lock, a thread may not invoke any function and/or allow preemptive invocation of any service routine that attempts to acquire the lock

only one BlockLock; attempts to acquire another is a recursive-acquisition error

can use one of the bits of Monitor’s mask…mask[0] to control blocking of other processors

use diminishes processor utilization and extends response time for preemptive service…a reason for keeping the critical sections of code short

in mono-processor systems, not really a problem since the BlockLock does exactly what is needed without degrading performance

**6.1.2 Spin Locks**

A *busy-waiting* implementation; use a flag to indicate that the lock is held

need to make sure the flag is checked and set within one cycle…use the testAndSet instruction

improve by using held || testAndSet(held) to prevent constant overwriting of held when unnecessary, which invalidates cache, causing memory bus congestion

possibility of *indefinite postponement* – some thread waits indefinitely while others repeatedly acquire and release the lock

Use a circularly linked list to organize the threads. On release, the release passes the lock to the next thread on the list

**6.1.3 Number Locks**

Locks without using special instructions such as testAndSet and BlockOtherProcessors

Bakery algorithm: inspired by the “take-a-number” system in bakeries

Precedence of a thread is based on its number. Smallest first, then thread addresses if there is a tie.

**6.1.4 Notes on Low-Level Locks**

* BlockLocks are not individually releaseable; SpinLocks and NumberLocks are.
* BlockLocks stop all other processors; SpinLocks and NumberLocks block only those processors that host threads trying to acquire a held lock…good for concurrency on multiprocessor systems
* Recursive requesting of low-level locks
  + SpinLock: self-deadlock; busy-wait on itself forever
  + NumberLock: lock releases and cause the thread to wait its turn to reacquire the lock while another thread acquires it
  + BlockLock: first subsequent invocation of release resumes other processors, partially releasing the lock. No problem in improved version using the mask bit
* Locking the bank of memory containing a sensitive data structure is another way to guarantee exclusive access. Caution with systems with processors having instruction/data caches
* compiler optimizations may change the order of instructions…possibly leaking code out of their critical sections

**Taking turns.**

If we know this, we can just use flags to coordinate the threads

**6.2 Priority Queues**

* insert(Item &x, int priority)
* extract()

In our case, the items are thread descriptors

A pitfall: when a new “node” is allocated on an insert, a recursive-lock request occurs when a scheduler using the queue is used the schedule a thread-safe allocator

**6.3 Passing a Processor**

store the running thread’s host state into the running thread’s descriptor and restore that processor’s state form data in the descriptor of the suspended thread