**CS 153**

**Design of Operating Systems**

**Ch. 5 Thread Safety and Coordination**

threads need to coordinate their access to shared servers

example: two threads simultaneously try to obtain the first buffer from a free-buffer list and get the same one

*thread safe* – servers that are coherently sharable among multiple concurrent threads

*scheduler/dispatcher/allocator* – server that delays the progress of client threads by making them wait

acquire – service that takes parameters about priorities and amounts of resources being requested; a handler of a service “checks in” by invoking this

release – the handler of the service invokes this to “check out”

ownership can be transferred between threads

high-level schedulers are usually monitors that have simple schedulers…*locks*

*critical section* – code segments where a lock is held; best to be kept short

**5.1 Mutual Exclusion**

**5.1.1 Locking**

*mutual-exclusion problem* – implement a class of schedulers, called *locks* that enforce a one-thread-at-a-time scheduling protocol

a class of servers having acquire and release services that meet the following specs

* lock promises that
  + no lock is ever held by more than one client at a time
  + any invocation of acquire on an unheld lock returns promptly
  + bound on the number of times any other client can acquire a given lock during a given invocation of acquire
* provided that the following are observered by all clients
  + not release a lock it does not hold
  + not acquire a lock it holds

**5.1.2 The blocking of preemption**

a thread must temporarily block its host processor from preemption by requests for a certain preemptive service of a surrounding server to enforce coordination protocols

requests that reach a given processor for any blocked service are enqueued to be handled upon unblocking the service

**Perspective notes**

* locks usually postpone requests on a server-wide basis
* locks are shared among threads; each thread has its own preemption mask
* on thread resume, its new host’s preemption mask gets set to the state that the thread’s previous host had when that thread last suspended
* event-blocking flag is never simply released; it is restore to its prior state

**Fundamental principles of preemption blocking**

Before acquiring a given lock

* to avoid undefined behavior from a recursive request, block all diverting services who might do so
* to avoid self-deadlock, block all other preemptee-blocking preemptive services who might do so
* to avoid performance degradations, if this lock is low-level, block all preemptive services that it is safe to block
* block all services whose handlers will surely attempt to acquire the lock...this diminishes concurrency very little

for high level locks, blocking preemptee-nonblocking services that are not sure to attempt to acquire WILL diminish concurrency

each thread keeps a blockage count for each preemptive service; acquire: +1; release: -1; note: do not want preemption to occur during counter shifting

**Critical sections**

sections of code requiring mutual exclusion

**5.2 Monitors**

*monitor* – a monothreaded server…handlers constitute the critical region controlled by a local server-specific lock

for C++ objects, mutual exclusion is enforced by acquiring the server’s lock at each handler’s entry and releasing it on return

*acquisition via initialization* – constructor of class handles recoding and blocking of preemptive services and locking of the surrounding monitor’s lock; destructor releases the lock and restores the preemptive services

**5.2.1 Conditions**

used to efficiently delay the handling of service requests

a scheduler having wait and signal services such that wait’s handler does not return until a subsequence signal request has been issued

**Waiting**

client thread requests the condition’s wait service; the handler puts the client’s descriptor onto the condition’s queue of suspended threads and suspends the client

**Postconditions on waits**

meanings of “condition”

1. condition variable
2. statement about the requesting client, server’s attributes etc.

pre/post-conditions: conditions that are expected to be true before and after some segment of code

use wait to slow down rechecking in predicate-rechecking

**Signaling**

when a client thread requests a signal service, the highest priority waiter becomes runnable

*signaler* – client that requests the signal service

*signalee* – the dispatched thread

three common protocols for dealing with how only one of them may hold the monitor’s lock:

* *Mesa semantics* – signalee defers to signaler by requesting the monitor’s lock
* *Hoare semantics* – signaler defers to signalee by passing waiting on a hidden condition, simultaneously passing the lock to the signalee
* *Brinch Hansen semantics* – signaler defers to signalee by passing the lock to the signalee and immediately leaving the monitor

*Mesa semantics*

submit an acquire request to the Monitor’s lock, competing with all other threads trying to acquire that lock

signaler finishes its job and exits the monitor or waits on a condition…this releases the lock

waits are all recheck, since a competing thread may beat the signalee in acquiring the lock…this could also cause the predicate to be false due to changes made by the competing thread

the signaler need not establish the predicate before signaling…it is more of an indication for rechecking

broadcast: like signal, but resumes all threads waiting on the condition

tail wait: the wait on a Mesa Condition is followed by a return…this allows the signalee to not bother acquiring the lock since it should be released immediately (use a suppress-release)

* if all data-manipulation code after a wait can be moved ahead of the corresponding signal invocation on the Condition, we can replace the wait with a tail wait
* the acquire and releases would have been serial without the optimization…each waiter would need to acquire the lock in turn, while the others wait…loss of concurrency
* a monitor’s data is no longer accessible to a signalee once it has returned from a service routine of that Monitor…cannot use predicate rechecking
* optimization does not work for Hoare or Brinch Hansen semantics

*Hoare semantics*

signaler defers to the signalee

signaler immediately attempts to acquire the Monitor’s lock after transferring it to and resuming the signalee

avoiding pseudo-deadlocks:

hidden condition: urgent

signaler waits on a hidden condition instead of the lock

give priority to waiting signalers by checking urgent

*Brinch Hansen (BH) semantics*

like Hoare semantics, but leaves the monitor’s service routine without releasing the lock

if all signal expressions involving or potentially involving a given Mesa or Hoare Condition are tail signals, that condition can be turned into a BH condition.

**Timeouts**

a waiting client can timeout after waiting for a certain period

client receives an artificial signal…but this does not guarantee that the Condition’s invariants/predicate that the thread is waiting for are true

spurious signals: when a timeout occurs at the time of a signal

tail waiting cannot be used where spurious signals are possible

**Notes on Conditions**