Concurrency II

Staying in Sync

Contention

Resources are Limited

Concurrent applications frequently need to access the same resources across different processes

Concurrent Resource Usage is “okay”, but comes with a dizzying array of caveats

Concurrent Resource Usage sometimes just doesn’t make sense

Ready/Modify/Store cycle

Back to CPU diagram we go!

Mutual Exclusion

Guarantee that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

Mutual Exclusion is one mechanism that Concurrent Applications use to protect themselves from Contention

Code blocks under “protection” are called Critical Sections

Use of Mutual Exclusion in an application is called Synchronization

Atomic Operations

Atomicity is another mechanism that Concurrent Applications use to protect themselves from Contention

Guarantee that a process’ Read/Modify/Write cycle happens as “one operation”

Atomic Operations

CPUs provide the following:

Interlocked Read

Interlocked Increment

Interlocked Decrement

Interlocked Exchange

Interlocked CompareExchange

Atomic Operations

Further Atomic Operations are few and far between

Synchronization Primitives

Lock

Readers-Writer Lock

Semaphore

Condition Variable

Lock

Purest implementation of the Mutual Exclusion Guarantee

See example...

Readers-Writer Lock

Observation: Most accesses to memory are Reads, few are writes

Writers must wait until all Readers and all other Writers are done

Readers must wait on Writers, but do not have to wait for other Readers

Semaphore

Allows up to N processes to acquire a resource at once

As processes “acquire” the resource, the semaphore counts up

Once a predetermined capacity is reached, the semaphore makes processes wait to acquire

Releasing a resource makes the semaphore count down

Condition Variable

Indicates to waiting processes when a specific condition is true

The condition itself is not encapsulated in the Condition Variable, only its signal state

waitBriefly()?

Examples so far keep using the waitBriefly function. What?

The implementation is actually very complicated

Boils down to two possibilities (usually mixed together):

1) Do busy work (Spin Wait)
2) Relinquish execution cycles to other processes (Blocking Wait)

Not so Primitive

Monitor

Futures

Barrier

Lock Free Data Structures

Transactional Memory

Monitor

Locks by themselves are not enough

Some concurrent operations require both Mutual Exclusion and Condition Variables

E.g. a Thread waiting for a queue to be filled up before it can operate

Futures

A.k.a. Promises

Essentially a Condition Variable without a reset function

Barrier

Waits on multiple processes to reach a demarcated point of execution.

$.when(async1, async2, async3, …).then(function() {

})

Lock Free Data Structure

What if we can design data structures that have strong guarantees that they don’t need Mutual Exclusion?

Few and far between

Lock Free List, Stack, Queue, Set, Skiplist

Transactional Memory

Guarantee that Read/Modify/Write cycles to memory are all atomic (regardless of complexity)

Intel Haswell and Broadwell CPUs added TSX instructions, but they were broken on both tries

The Cutting Edge of Concurrency Control™