Concurrency and VMs

Introduction

Why? How?

- Concurrency seems inevitable, but federated single-thread VMs can be practical (and simpler), if there is limited communication between them, and fairly even resource requirements.
- VMs don’t scale to current hardware; certainly don’t scale across nodes.
    - Replication is needed anyway to deal with faults (software, hardware)
- But for fast communication, nothing beats shared memory (for now).

Concurrent models: choices

- Atomicity
- Locking
- Memory model

- Explicit concurrency:
  - Actors, monitors
- Implicit (introduced) concurrency:
  - Vectors, SIMD, auto-parallelization

Scheduling

- It is much easier to build a cooperatively scheduled VM supporting language-levels threads on a single hardware/OS thread than to make a multi-threaded VM.

- However, on a multi-processor, cooperative scheduling doesn't help; preemption is the current hardware/OS reality.

- It would be nice to have support for cooperation baked in to the hardware and OS; e.g., it could help with safe points (ask me).

Concurrency in interpretation

- Interpreters are slow so it's not a goal to parallelize them to make them faster.

- However, it would be nice if each interpreted thread wasn't *slower* because of concurrency.

- Problem: some interpreter techniques are not thread-safe.

    - Examples: bytecode rewriting (resolution, "quick" bytecodes, selective inlining).

- Case in point: Truffle ASTs are copied per thread (node rewriting is not thread-safe)

- Not much known about how threads interact with profile-feedback.

Object Synchronization

- The most widely used synchronization technique is *object locking*, which is derived from *monitors* (Brinch Hansen 1972, Hoare 1974).

- In Java, these are *synchronized methods* and the `synchronized` construct.

Implementing object locking

- In Java, if a thread owns the lock on an object, it can freely call any other synchronized method on that object; this happens a lot. The lock is released when the last such method is exited. This means we have to count how many times a lock has been acquired without being released.

- The simple way to implement object locking is to have a lock table, indexed by object ID. The locks themselves are implemented in a library or by the OS.

- Implemented naively, this is very slow. Hashing into the table for each monitor enter/exit takes forever.

Fast object locking

- Considerable effort has been expended to optimize locking.

- The best kind of locking is no locking — which can be achieved in some cases through escape analysis.

- But if you need to lock…

Some empirical observations

- Most locks are uncontended
- Once a thread has acquired a lock, it frequently reacquires it.
- Hence: we should make uncontended acquisition and reacquisition/release fast.

Fast locking in HotSpot

- Based on [Russell and Detlefs, 2006].
- The tag bits of an object's header word encode its lock state:

  xxxxx00 - unlocked
  xxxxx01 - lightweight lock
  xxxxx10 - heavyweight lock

Lock records

- When a monitorenter is executed on an unlocked object, the header word is copied to a slot in the current stack frame and a pointer to this slot is CAS'd into the header word. If successful, the lock has been acquired. If unsuccessful, a slow path is entered using heavyweight locking.

- If a subsequent monitorenter (by the same thread) is executed on the object, the pointer is compared to the stack's bounds to determine ownership. If the current thread owns the lock, the lock record is set to zero.

- The number of lock records on the stack indicates the locking depth.

Further improving uncontended locking

- On multi-processor systems, CAS is an expensive operation. To remove many of the CASes, HotSpot uses *biased* locks.
- If a lock is considered bias-able, the first lock acquisition attempts to CAS in a special biased pattern, together with its thread ID:

  xxxxx000 - unlocked
  xxxxx001 - lightweight lock
  xxxxx010 - heavyweight lock
  xxxxx101 - bias-able

Biased locking

- If the CAS is successful, the lock is now biased towards its owner.

- Once a lock has been biased, nested lock and unlock operations do nothing but look at the header word — no atomics are needed.

- If the CAS was unsuccessful, another thread has taken the lock and biased it towards itself. The bias has to be revoked. All threads are brought to a stop at a safepoint, and then the stack of the errant owner is walked and changed to make it look like it took a lightweight lock (i.e., the appropriate lock records are filled).

Bulk rebiasing and revocation

- Bias revocation is expensive. If it happens too often within a particular type, a bulk rebias is done: all biasable instance have their bias owner reset.
- If bias revocations persist for that type, a bulk revocation is performed. All header words of biasable instances are reset to use lightweight locking, and biasing is disabled for new instances of that type.

Compiler implementation of locking

- As we saw in the JVM bytecode overview, lock acquisitions and releases (monitorenter, monitorexit) might not be paired in the bytecode.
- Detecting this case in biased locking adds much complexity, so biased locking is only used in compiled code. Compiled code is generated only when the compiler can prove the monitorenters and -exits are balanced. Otherwise, the method is executed in the interpreter; only the interpreter has to deal with unpaired operations.

Code management

- Code patching in a concurrent environment is *tricky*.
    - Used for inline caches; class resolution; linking to new code; flushing old code.
        - Even on a uniprocessor, pipeline flushes have to be inserted in key places.
        - I-caches must be flushed of invalid code. Easy enough to flush your own I-cache, but what about another socket's? (coherent I$ is nice!)
        - The memory model might reorder patching operations.
- Most ISAs assume code patching is as rare as a unicorn, so don't make many guarantees about timeliness.

Concurrent compilation

- With abundant threads, it can make a big difference to run the compiler in parallel.

- Most high-performance VMs maintain a compilation queue; counter overflows, inline cache misses, etc., push items onto the queue; one or more compiler threads drain the queue and generate code.

- The compiler must get a consistent view of the “source” (e.g., bytecodes, types, etc.) — some operations (e.g., class loading, mutation) might be held up during a compile.

Foreign code

- Foreign code (such as pre-compiled libraries, either linked to the VM or dynamically loaded by the application) will not have safe points, may use its own calling convention, etc.

- When a call is made to such code from the VM, a barrier is crossed. Threads outside the barrier are not suspended for GC, locking, etc., unless they attempt to re-enter the VM over the barrier.

Next time…

- Memory management in a concurrent environment, including an introduction to concurrent GC.
- Tools and VMs, with Michael Van De Vanter.
- The results of the competition.