JVM Implementation Ideas

This is a list of various useful design ideas and implementation tricks that may be desirable to include in Harmony.

Per-class loader memory areas

- Idea: class loaders and all their defined classes are unloaded all at once. Memory is allocated and not freed until that happens, so allocation behavior is stack-like. Instead of using the more general and therefore costly malloc() or the Java heap for class loader specific memory, use a lighter weight, stack like memory subsystem. In addition, all java.lang.Class objects can be stored in per-class loader memory instead of in the normal heap, reducing work for the GC subsystem.
- Advantages: more efficient, makes class loader unloading easier
- Disadvantages: if java.lang.Class objects are stored in per-loader memory, their references must be tracked manually.
- Comments: this one is pretty much a no brainer
- Origin: SableVM

Bi-directional object layout

- Idea: let objects' primitive fields grow upward from the object head and let reference fields grow downward from the object head. Object pointers pointing to objects themselves containing object references therefore point into the middle of the object's memory area.
- Advantages: all object references are contiguous, which makes following these references during garbage collection, etc. easier and simpler.
- Disadvantages: object head (lockword, vtable pointer, etc) is not necessarily at the beginning of an object's memory area, which can mean extra work to find the head when iterating through the heap e.g. during mark/sweep GC. Etiene notes that there is a performance degradation, mainly due to poor locality of this layout.
- Origin: SableVM

Spinless thin-locks

- Idea: compare-and-swap instruction used to grab an object lock in the common case of no contention, otherwise fall back to mutexes.
- Advantages: very efficient
- Disadvantages: some assembly required
- Comments: using some form of thin locks is a no-brainer
- Origin: lots of people. SableVM has a nice version of this algorithm.

SableVM thread state tracking

- Idea: use compare-and-swap to transition threads from running in Java mode vs. running in native mode, and to detect a "stop the world" operation. When a Java thread goes into "native mode", i.e., it invokes a JNI function or some other blocking system call like pthread_cond_wait(), it has to detach itself in some sense from the JVM so that the JVM doesn't get stuck waiting indefinitely for it to return.
- Advantages: very efficient implementation for Java locking
- Disadvantages: none known
- Origin: SableVM

Signals for thread notification

- Idea: threads must periodically check whether they need to do something. Typically this is done at backward branches. To make this check efficient, have the thread read a byte from a well-known page of mmap()'d memory. When another thread wants the thread to check-in, it simply maps that page unreadable. The target thread gets a signal, it does whatever check required, the page is re-mapped readable, and the target thread returns from the signal and continues on its merry way.
- Advantages: efficient way to enforce checking in
- Disadvantages: requires mmap() and signals, signal latency can be an issue if the notification delay is critical, eg gc safepoints (see the paper from some Sun guys about this)
- Origin: unknown/multiple; implemented in lots of places.

Inline threaded interpreter

- Idea: similar to a direct threaded interpreter, but in the preparation step copy the code segment of some opcodes instead of pointer + data.
- Advantages: faster interpreter.
- Disadvantages: deciding with code segments can be inlined is a non-trivial, non-portable issue. A conservative approach will result in worse performance.
- Origin: SableVM