



Design and Implementation of Object Oriented Virtual Machines

by Lars Bak



Object-based locking in Java

- Libraries are designed for multi-threaded use
- High frequency of synchronization
- Ubiquity: all objects are subject to synchronization
- Example javac compiling javac
 - > 20.000.000 of synchronization operations
 - Single threaded program and therefore no contention



Fast Synchronization In Java

- Design Goals
 - Minimal space overhead per object
 - Minimal execution time overhead in the common case



Synchronization in Java

- when invoking a synchronized method
`synchronized void foo() { ... }`
- when executing a synchronized statement
`synchronized(obj) { ... }`
 - `monitor_enter` & `monitor_exit` byte codes

Synchronization in Java is block structured



Entering a Synchronized Operation

- If the object is:
 - not locked, the current thread acquires the lock
 - locked by the current thread, increment the lock counter
 - locked by another thread, the current thread waits for the object to be unlocked, then acquires the lock



Leaving a Synchronized Operation

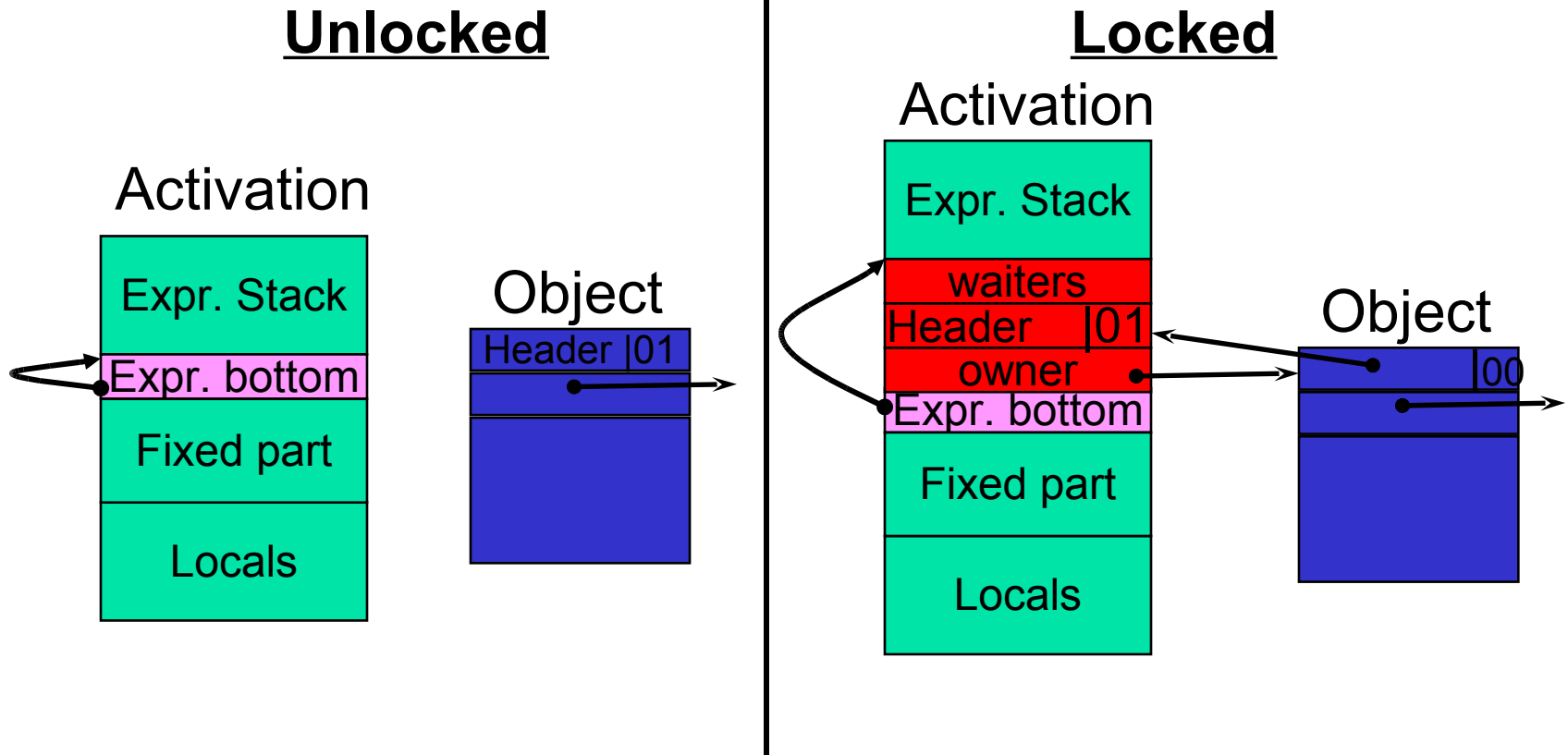
- Decrement the lock counter
- If the counter of the object lock is 0, release the lock



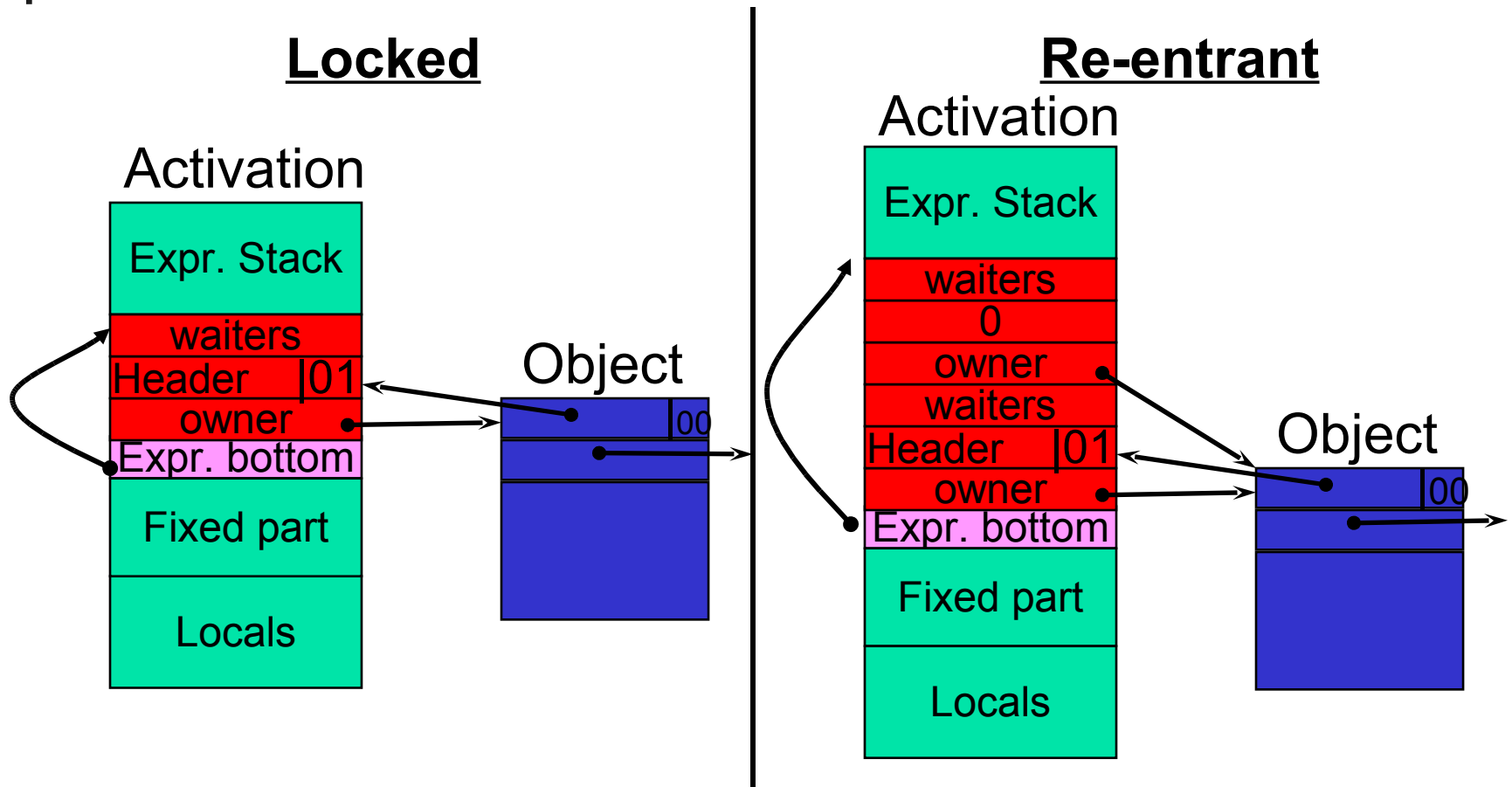
The Idea

- Allocate the monitor structure in the activation executing the synchronized operation
- Make a forward pointer from the object to the monitor structure in the activation
- Assumes the a stack pointer uniquely identifies the thread
- When leaving the activation the interpreter checks whether all monitor structures have been released
- Lock counter is implicit

Locking an Object



Relocking an Object





Interpreter and Monitors

- Interpreter expands the monitor area as needed
- When leaving the activation the interpreter checks whether all monitor structures have been released
- Lock counter is implicit



Compiled code and Monitors

- Compiler uses abstract interpretation to compute the number of monitors needed and pre-allocates them in the activation
- Result is fast code



Locking Code

Object header field

```
value := thread_specific_sentinel
swap(obj->header_addr(), &value)
if (value->is_unlocked()) {
    monitor->set_header(value)
    obj->set_header(monitor->header_addr())
} else if (value->is_sentinel()) {
    ... spin_lock and priority boosting
} else {
    if (in_same_thread(value)) {
        obj->set_header(value)
        monitor->set_header(NULL)
    } else
        ... queue myself and wait for notification
}
```

H		01	->	unlocked
SP		00	->	locked
SP		10	->	busy



Unlocking Code

```
if (monitor->header() != NULL) {
    value := thread_specific_sentinel;
    swap(obj->header_addr(), &value)
    if(value->is_locked()) {
        obj->set_header(monitor->header() | 01)
    } else {
        ... spin_lock and priority boosting
    }
    if (monitor->header() has clear lsb) {
        ... notify waiters
    }
}
```



Generated Code for i486

```
_method_prologue:          ; Synchronization prologue
    ; ecx points to the object
    lea    ebx, 2[esp]          ; value = sentinel
    swap   ebx, [ecx]          ; swap(value, object header)
    testl  ebx, 0x1            ; if (!object_is_unlocked)
    jne    _locked_in_method_prologue ; goto _locked_in_method_prologue;
    pushl  ebx                  ; stack->push(value);
    movl   [ecx], esp          ; obj->set_header(stack->top_address())
_end_method_prologue:

_method_epilogue:          ; Synchronization epilogue
    ; ecx points to the object
    popl   ebx                  ; value = stack->pop();
    testl  ebx, ebx            ; if (value == 0)
    je     _end_method_epilogue ; goto _end_method_epilogue;
    ;     ...recursive case, so no
    ;     microlock is needed

    lea    edx, 2[esp]          ; value = sentinel
    swap   edx, [ecx]          ; swap(value, object header)
    testl  edx, 0x1            ; if (!object_is_unlocked)
    jne    _locked_method_epilogue ; goto _locked_method_epilogue;
    testl  ebx, 0x1            ; if (header has clear lsb)
    je     _threads_waiting    ; goto _threads_waiting;
    movl   [ecx], ebx          ; obj->set_header(stack->top_address())
_end_method_epilogue:
```



Contended Case

- If native threading is used:
 - The light-weight monitor is inflated to OS level monitor and the execution continues
- Scales as well as the underlying OS



Alternative Implementations

- Off-stack allocation of monitor structures
- Create map from object to monitor
 - hash table, thread local
 - extra word in object



Fast Synchronization

- Software only
- Minimal space overhead per object
 - 2 bit in the object header
- Use stack pointer as thread identifier
- Minimal execution speed overhead in the common case
- Supports native threads, SMP



Subtype Tests

- Common operation in object-oriented systems
- Used for safe down casting:
`X x;`
`Y y = (Y) x;`
- Array store check
- Subtype tests
`if (x instance of Y) { ... }`



Implementations

X is a subtype of Y?

- Class hierarchy traversal
- Table-based
- Caching

Please note interface types and multiple inheritance makes it complicated



Class Hierarchy Traversal

- Search inheritance hierarchy for validating the X matches Y or one of its super classes
- Pros:
 - Space efficient, no extra space is needed
- Cons:
 - Very slow, linear is the number of super classes for Y



Table-based Subtype Tests

- Two-dimensional table indexed by class numbers
 $T[i, j]$ returns whether C_i is a subtype of C_j
- Pro:
 - Simple implementation and fast access
- Cons:
 - Space bloat: quadratic in number of classes
 - Resizing is hard when having dynamic class loading



Cache-based Subtype Tests

X is a subtype of Y?

- Where should the cache be kept?
 - In X?
 - In Y?
 - Save positive results, negative result or both?
- How many cache elements?



Cache-based Subtype Tests

- Empirical in Java tests shows:
 - Negative tests are very infrequent
 - Cache should be kept in X
 - Two element cache is enough
- Pros:
 - >98% hit rate and very fast
 - Takes only two words per class
- Cons:
 - Worst case is still like hierarchy traversal



Code for Cache-based Test

```
class Klass {
    // is_a cache variable
    Klass* _subtype_cache_1;
    Klass* _subtype_cache_2;
    // is_a cache accessors
    Klass* subtype_cache_1() const { return _subtype_cache_1; }
    Klass* subtype_cache_2() const { return _subtype_cache_2; }
    // returns whether this is a subtype of k
    bool is_subtype_of(Klass* k) {
        return subtype_cache_1() == k
            || subtype_cache_2() == k
            || compute_and_set_is_subtype_of(k);
    }
    // computes whether this is a subtype of k and updates cache
    bool compute_and_set_is_subtype_of(Klass* k);
}
```



The End

- Q & A



Project For Next Quarter

- Written report
 - ~10 pages report
 - with benchmark numbers
 - with footprint numbers
- Source code
- Project presentation



What Project

- Design and implement you own virtual machine
 - Examples Python, Smalltalk, Ruby
- Implement an interpreter for V8
 - Design and optimize bytecode set
- Implement a profiler that collects call graphs



What Project

- Implement a real time profiler
- Design and implement an incremental old space garbage collector
- Implement machine migration of a running V8 system



What is Important?

- Make it run early and make incremental changes
- I REALLY don't like code that does not run
- Ask us if you are stuck