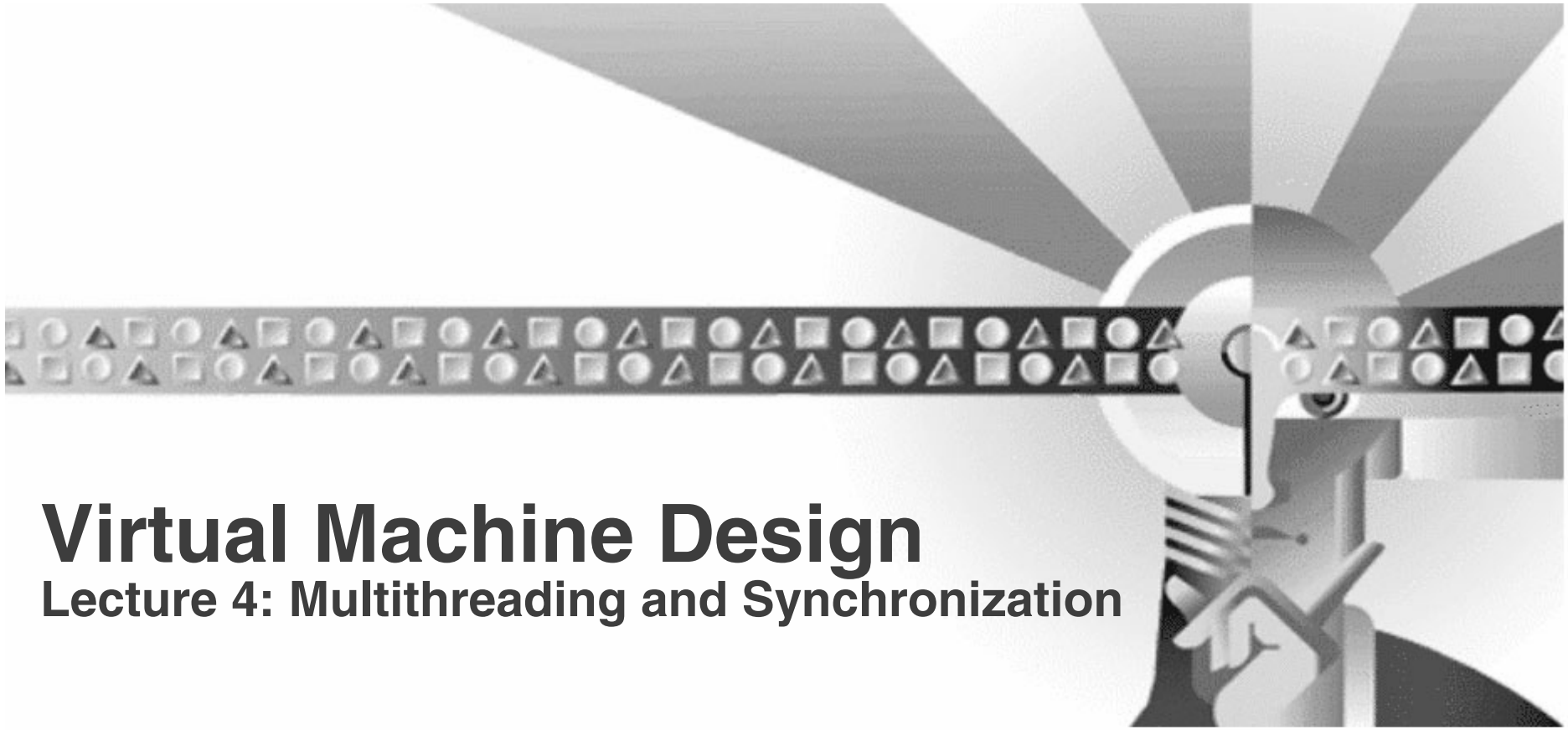


Virtual Machine Design

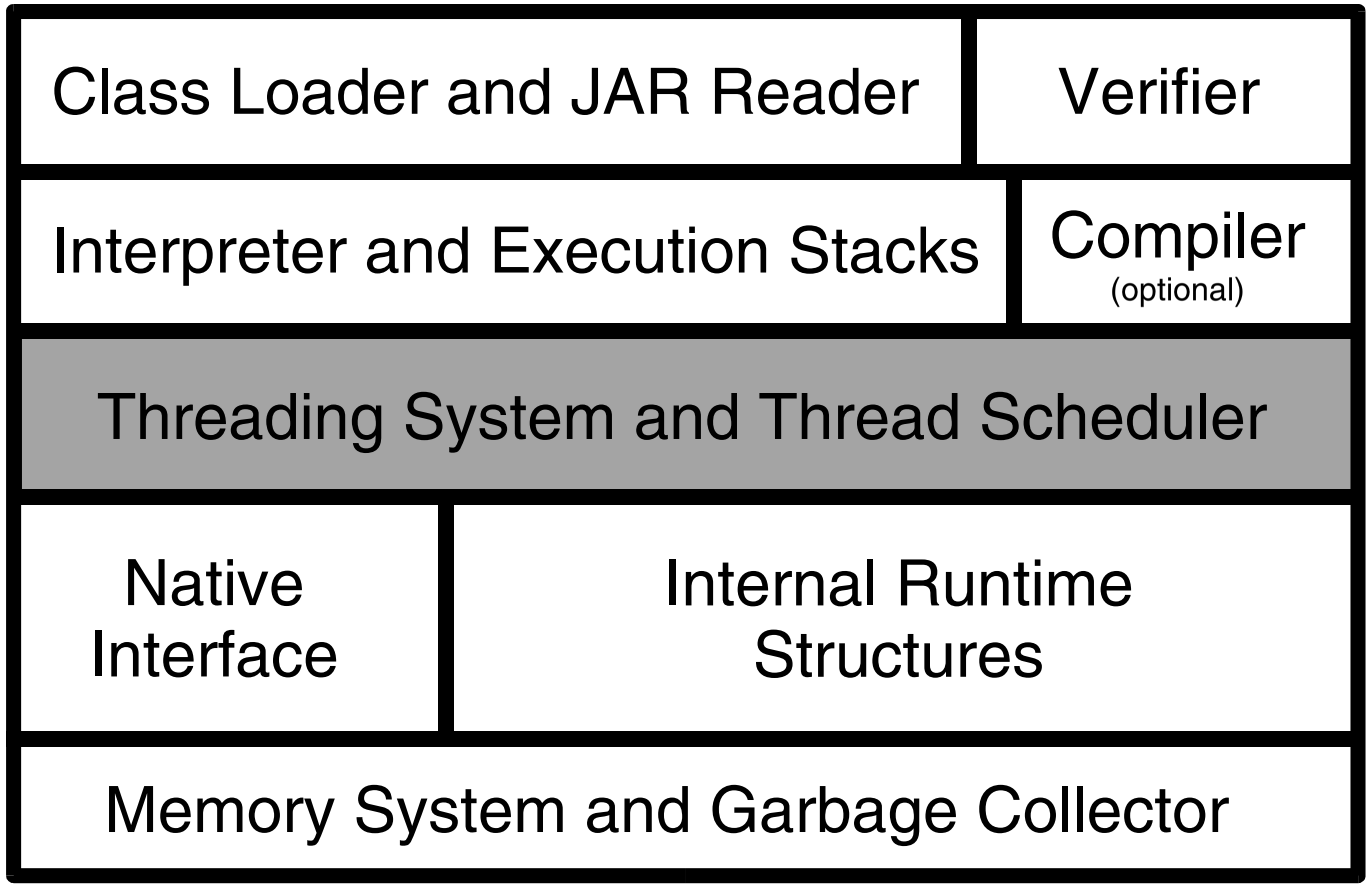
Lecture 4: Multithreading and Synchronization

Antero Taivalsaari
September 2003

Lecture Goals

Give an overview of multithreading support and the synchronization / locking mechanisms in virtual machines, focusing on the implementation aspects.

Essential Components of a (Java) Virtual Machine



Multithreading

Multithreading and Synchronization

- A key feature of many programming languages is *multithreading.*
  - *Multithreading*: the ability to create multiple concurrently running threads/programs.
  - Smalltalk, Forth, Ada, Self, Java, ...
- Each thread behaves as if it owns the entire virtual machine
  - ... except when the thread needs to access external resources such as storage, network, display, or perform I/O operations in general.
  - ...or when the thread needs to communicate with the other threads.
- *Synchronization/locking* mechanisms are needed to ensure controlled communication and controlled access to external resources.

Multithreading Support in Java

- Example:

```
public class MyClass implements Runnable {
  public void run() {
    ... do something ...
  }

  public static void main(String[] args) {
      for (int i = 0; i < 10; i++) {
          Runnable r = new MyClass();
          Thread t = new Thread(r);
          t.start();
      }
  }
}
```

Implementing Multithreading: Technical Challenges

- Each thread must have its own virtual registers and execution stacks.
- Critical places of the VM (and libraries) must use mutual exclusion to avoid deadlocks and resource conflicts.
- Access to external resources (e.g., storage, network) must be controlled so that two threads do not interfere with each other.
- I/O operations must be designed so that one thread's I/O operations do not block the I/O of other threads.
- Generally: All native function calls must be non-blocking.
- Locking / synchronization operations must be provided also at the application level.

Recap: Components of an Interpreter

- Interpreters usually have the following components:

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

Virtual registers

ip fp ...
sp lp

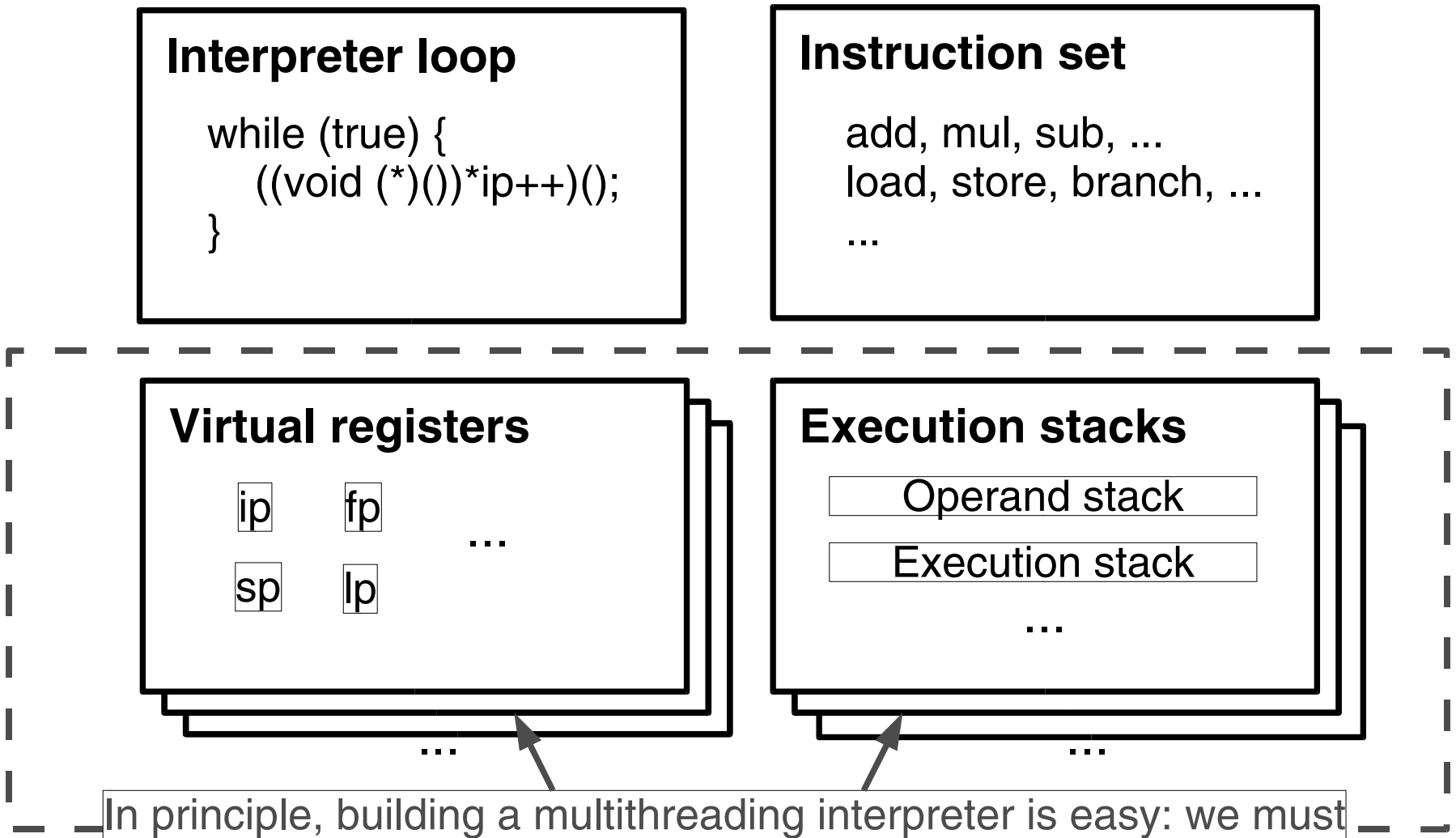
Execution stacks

Operand stack

Execution stack

...

Building a Multithreading VM



Building a Multithreading VM: Interrupts

- In addition, we must modify the system so that it allows the current thread to be *interrupted*.
- When a thread is interrupted, a *context switch* is performed. Example:

```
int* ip; /* instruction pointer */
while (true) {
        if (isInterrupted()) ContextSwitch();
        ((void (*)())*ip++)();
}
```

Building a Multithreading VM: Context Switching

- What happens during a context switch?
  - ❶ The virtual registers of the current thread are stored (context save).
  - ❷ Current thread pointer is changed to point to the new current thread.
  - ❸ Virtual registers are replaced with the saved virtual registers of the new current thread (context load).
- Context switching must be performed as an uninterrupted operation!
  - No further interrupts may be processed until the context switch operation has been performed to completion.
  - Operating systems commonly have a “supervisor mode” for running system-critical code.

Avoiding Atomicity Problems Using Safepoints

- In operating systems, threads can usually be interrupted at arbitrary locations.
  - Interrupts may be generated by hardware at any time.
  - The entire operating system must be designed to take into account mutual exclusion problems!
  - Must use monitors or semaphores to protect code that can be executed only by one thread at the time.
- In VMs, simpler solutions are often used.
  - Threads can only be interrupted in certain locations inside the VM source code.
  - These locations are known as "*safepoints*".
  - In the simplest case, thread switching is only allowed in one place inside the VM.
  - Makes VM design a lot simpler and more portable!

Using the “One Safepoint” Solution

- No separate interrupt handler routine.
- All the checking for interrupts happens inside the interpreter loop.
- Even if the actual interrupts are generated asynchronously, the actual context switching is not performed until the interpreter loop gets a chance to detect and process the interrupt:

```
int* ip; /* instruction pointer */

while (true) {
        if (isInterrupted()) ContextSwitch();
        ((void (*)())*ip++)();
}
```

Making Thread Switching 100% Portable

- You don't necessarily need an external clock to drive the interrupts!
- In the simplest case, you can count the number of executed instructions using a “timeslice”:

```
int* ip; /* instruction pointer */

while (true) {
        if (--TimeSlice <= 0) ContextSwitch();
        ((void (*)())*ip++)();
}
```

  - Force a context switch every 1000 bytecodes or so.
- You can also enforce a thread switch at each I/O request (used in Forth systems).

Thread Scheduling

- When you perform a context switch, which thread should run next?
- Various choices:
  - FCFS (First-Come-First-Serve)
  - Round robin approach
  - Priority-based scheduling
    - ... with fixed priorities or varying priorities
- In operating systems, thread scheduling algorithms can be rather complicated.
- There is no “ideal” scheduling algorithm.
  - The needs for interactive and non-interactive programs are different.

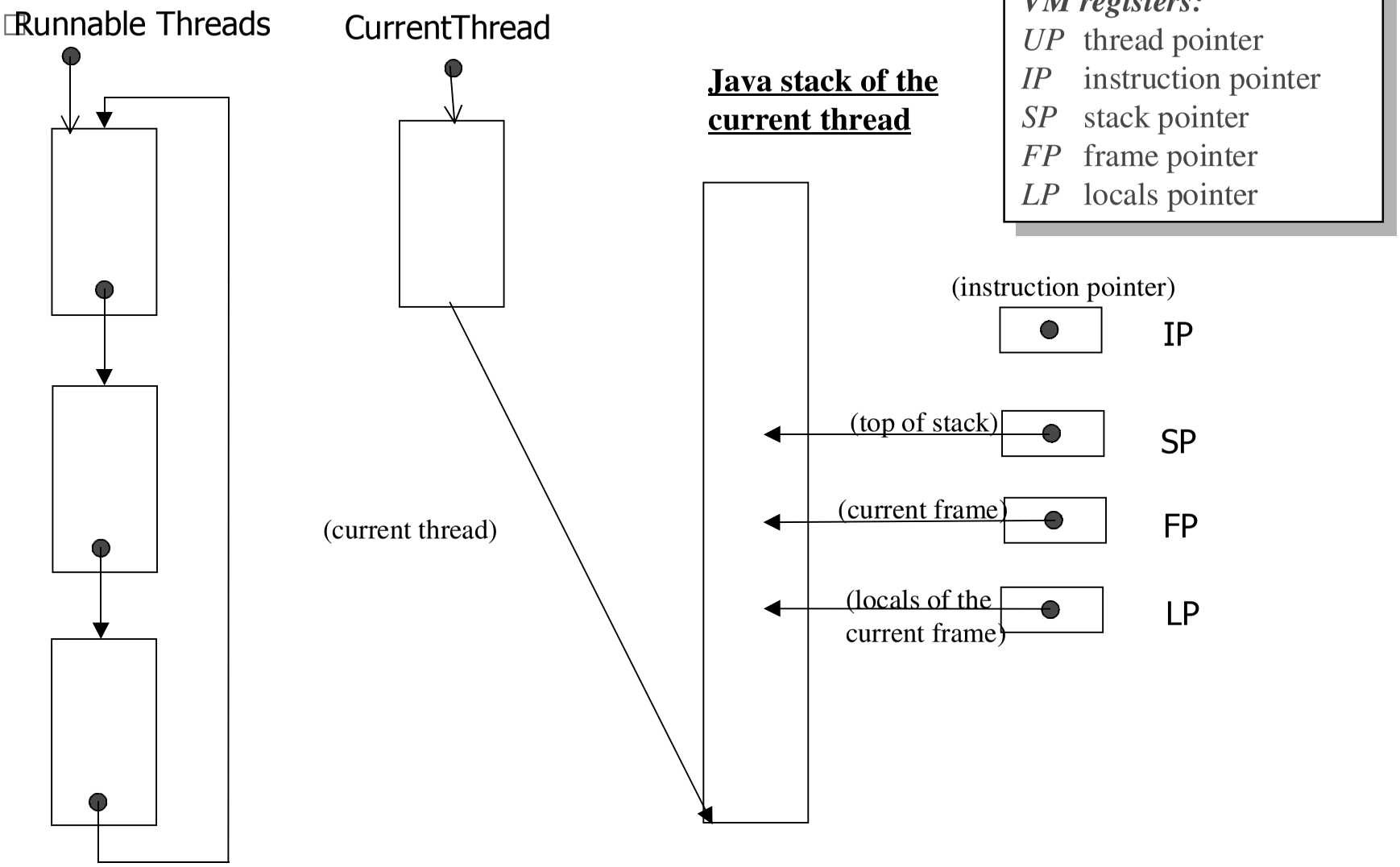
Example: Thread Scheduling in Java

- Thread scheduling is specified very loosely by the Java Virtual Machine Specification.
- Scheduling is assumed to be “fair”.
  - Generally: higher priority threads run before lower priority threads.
  - Priorities range from 1 to 10 (default priority: 5).
- In JVMs using native threads, scheduling is performed by the operating system.
- In JVMs using non-native threading systems, scheduling can be driven by:
  - The number of bytecodes executed (KVM)
  - Invocation of native I/O operations
  - External interrupts (processed at safepoints)

Case Study: KVM

- The original KVM (Spotless) implementation used a simple *round robin* scheduler.
  - Threads stored in a circular list; each thread got to execute a fixed number of bytecodes until interruption.
  - Thread priority only affected the number of bytecodes a thread may run before it gets interrupted.
- In the product version, thread scheduling is based on Java thread priority.
  - Higher-priority threads always run first.
- Fully portable thread implementation; interrupts driven by bytecode counting; thread switching handled inside the interpreter loop.

Example: KVM Threads



Straightforward bytecode interpreter with five VM registers:

UP thread pointer
IP instruction pointer
SP stack pointer
FP frame pointer
LP locals pointer

Example: Thread Switching in KVM

Code in the interpreter loop:

```
if (--Timeslice <= 0)
    do {
        ulong64 wakeupTime;

        /* Check if it is time to exit the VM */
        if (AliveThreadCount == 0) return;

        /* Check if it is time to wake up threads */
        /* that are waiting in the timer queue    */
        checkTimerQueue(&wakeupTime);

        /* Handle external events */
        InterpreterHandleEvent(wakeupTime);

    } while (!SwitchThread());
```

Actual thread switching code (a bit simplified):

```
bool_t SwitchThread() {
    StoreVirtualRegisters();
    CurrentThread = removeQueueStart(&RunnableThreads);
    if (CurrentThread == NULL) return FALSE;
    LoadVirtualRegisters(CurrentThread);
    Timeslice = CurrentThread->timeslice;
    return TRUE;
}
```

Native vs. Non-Native Multithreading

Different Multithreading Techniques

- Multithreading can be implemented in two basic ways:
  - *Non-native ("green") threads*: Multithreading is implemented internally by the VM; the threading system is independent of the multithreading capabilities provided by the underlying operating system.
  - *Native threads*: Multithreading is implementing using the capabilities provided by the underlying operating system.
- The efficiency, portability and complexity of the virtual machine varies considerably depending on the multithreading approach that has been chosen!

Non-Native Multithreading

- Multithreading implemented completely independently of native (OS) threads.
  - Only one physical native thread used by the VM.
- Thread scheduling is driven by the interpreter.
  - At scheduled intervals, the VM stores current virtual registers into the current thread, and loads the virtual registers of another thread, allowing the other thread to run.
  - Thread switching can occur only in predetermined locations inside the VM; this simplifies VM design.
- Used commonly in J2ME VM implementations, where portability and simplicity are more important than maximum performance.

Pros and Cons of Non-Native Multithreading

- Pros:
  - Simplifies VM design dramatically
    - No VM-internal synchronization / mutexes necessary
    - Java monitors can be simple counters
    - No porting layer needed for low-level synchronization operations
    - Garbage collector design much easier
  - Easier to control thread memory/resource usage; can support large number of threads in limited memory
  - Maximum portability
- Cons:
  - Some performance overhead on the interpreter and I/O
  - Cooperation from native methods required (native methods must not block)
  - I/O latency: threads cannot be pre-empted/interrupted arbitrarily by native code

Native Multithreading

- Multithreading implemented using the native thread system provided by the operating system.
  - Java threads mapped onto operating system threads.
  - Thread switching, scheduling and mutual exclusion operations are provided by the operating system.
  - Thread switching can occur at arbitrary locations inside the VM; mutual exclusion is needed to protect critical/shared code regions. This increases complexity.
- Native multithreading is typically used on those VMs that run on high-performance operating systems and/or that are generally performance-oriented.

Pros and Cons of Native Multithreading

- Pros:
  - Small I/O latency; threads can be interrupted by the OS
  - I/O mechanisms can be mapped to native I/O easier
  - Native function calls do not have to block the system
  - General: Maximizes benefits of the native thread system
- Cons:
  - Strong dependency between the VM and operating system -> poor portability
  - Native threading system must be available & efficient (this is not true on all embedded platforms)
  - Complexity
    - Asynchronous handling of I/O buffers can be complicated
    - All VM code must be written in a thread-safe manner
    - Porting interface needed for low-level mutual exclusion operations
  - Less or no control over thread memory/resource usage

Mixing Non-Native and Native Threading

- Sometimes, there are good reasons to use both non-native and native threads in the same VM.
- For instance, in high-end mobile devices native threads are fairly efficient, and can be used to simplify I/O calls.
- Solution: Use a pool of native threads to implement non-blocking I/O, while using only a single thread inside the core VM to keep the VM small, simple and easier to port.

Locking and Synchronization

Locking and Synchronization

- The Java programming language provides the concept of *synchronization* to support application-level concurrency control.
- Any Java method can be declared *synchronized*:

```
synchronized public static void myMethod();
```

- In addition, the programmer can declare *synchronized statements* (blocks) inside methods:

```
synchronized (obj) { ... Java code ... }
```

- Synchronized methods and blocks may be executed only by one thread at a time. VM is required to guarantee mutual exclusion.

Why Is Synchronization Needed?

- Synchronization is needed to provide controlled access to shared resources such as storage, network, display or I/O devices in general.
- Without synchronization, the I/O requests of threads could get garbled.
  - One thread must be able to finish its I/O request until another thread starts using the same medium.
- I/O requests of one thread must not interfere with the I/O requests of another thread.
- Generally, all I/O operations in multithreaded systems must be *thread-safe*.

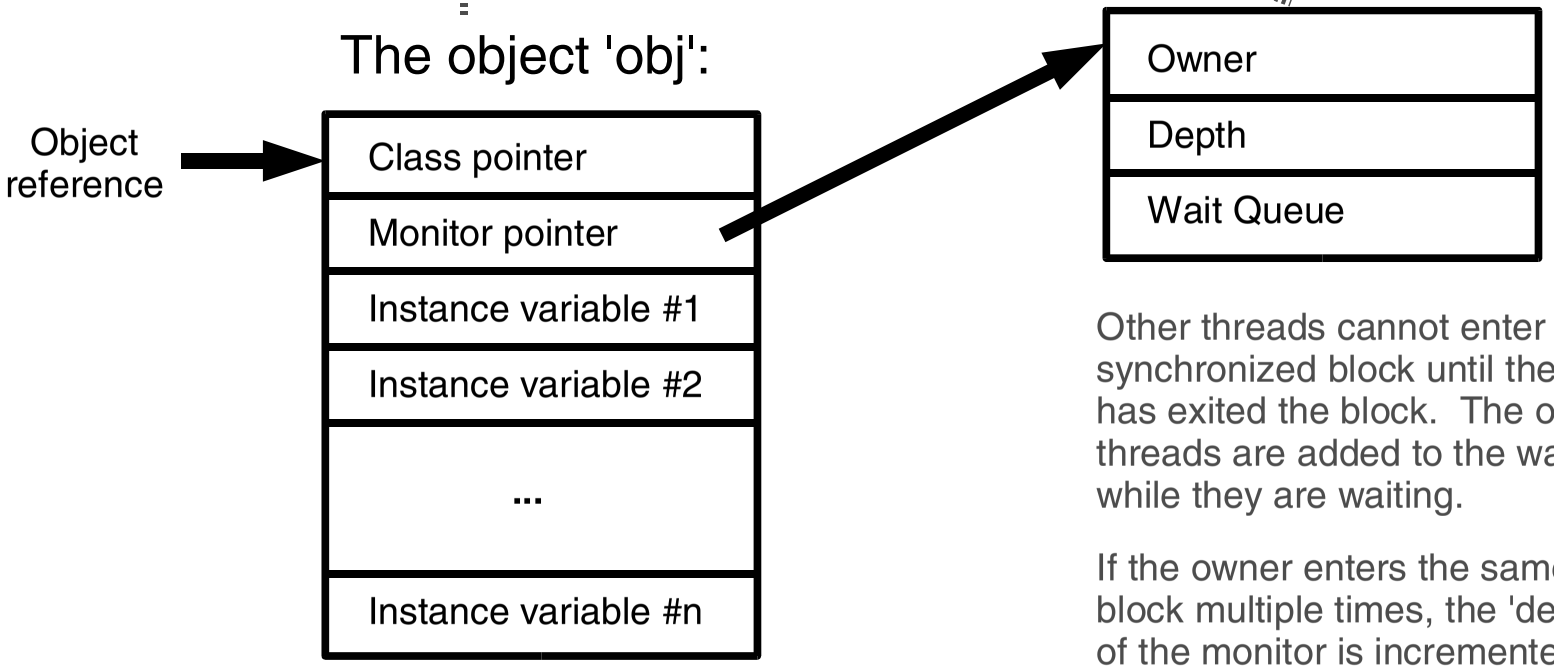
Implementing Synchronization

- To support synchronization, the VM may have a *lock* and a *wait queue* (monitor data structure) associated with each object.
- When a thread enters a synchronized method or statement, it tries to acquire the lock associated with the object.
  - If the lock is free, the thread acquires the lock and method execution can begin immediately.
  - If the lock is not free, the thread must wait until another thread has finished executing the code, and until the scheduler gives the waiting thread a chance to run again.
- Locking can be expensive to implement.
  - Monitors are usually short-lived but performance-critical.
  - The wait queue sizes vary dynamically.

Locking and Synchronization: General Case

```
synchronized (obj) {
   ... code protected by a monitor ...
}
```

The thread that first arrives in the synchronized block or method reserves the monitor (becomes the 'owner')



Other threads cannot enter the synchronized block until the owner has exited the block. The other threads are added to the wait queue while they are waiting.

If the owner enters the same block multiple times, the 'depth' of the monitor is incremented. The monitor cannot be released until 'depth' becomes 0 again (i.e., the number of enters and exits must match).

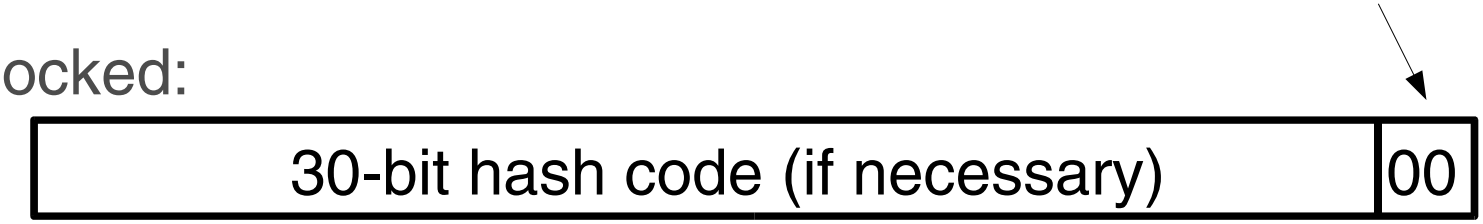
Locking and Synchronization: Performance Considerations

- Java libraries use synchronization extensively.
  - I/O, networking, graphics libraries must be thread-safe!
- Therefore, locking speed has a significant impact on the performance of a JVM.
- Observation: Most locking operations are *uncontended.*
  - *Uncontended:* there is only one thread who wants to acquire the lock.
  - Performance can be improved significantly if locking system is designed to optimize for this common case.
  - No need to instantiate “real” lock/monitor objects in most cases.

Example: Locking in KVM

- Each object in the KVM has a special MHC (monitorOrHashCode) field that is used in four different ways, depending on the low 2 bits.

Unlocked:



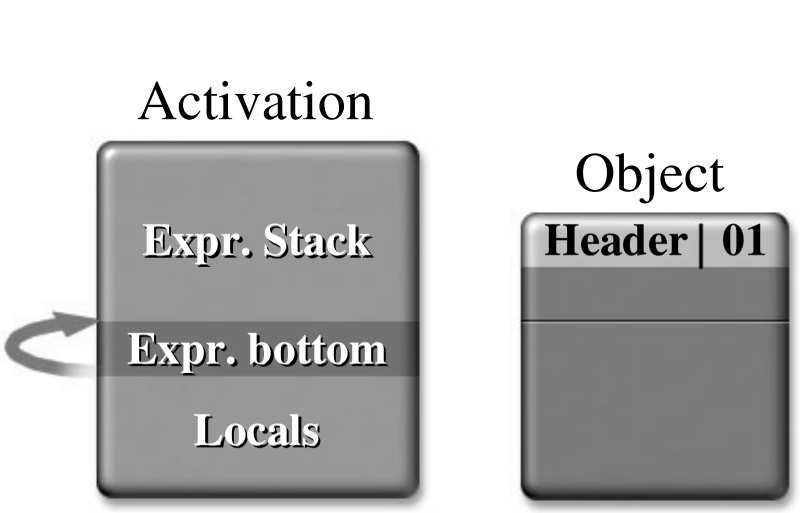
Uncontended locking:

Extended uncontended locking (lock depth > 1):

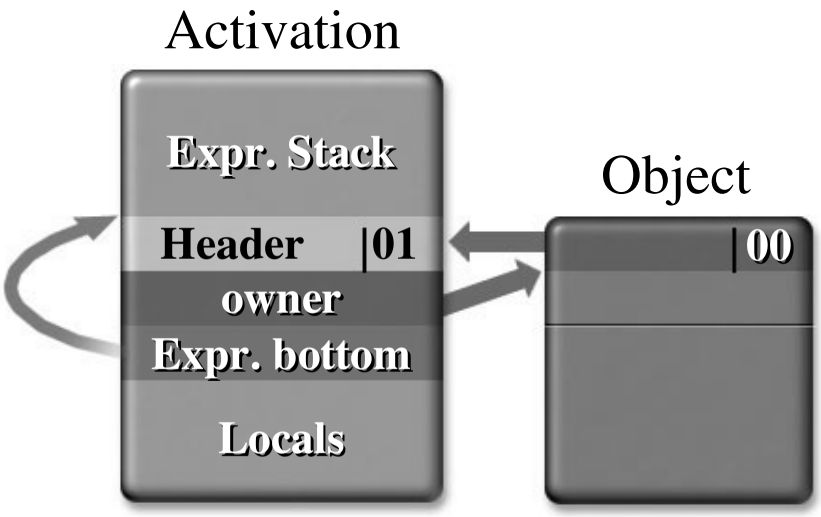
Contended locking (other threads waiting):

Example: Locking in the HotSpot Virtual Machine

Unlocked



Locked



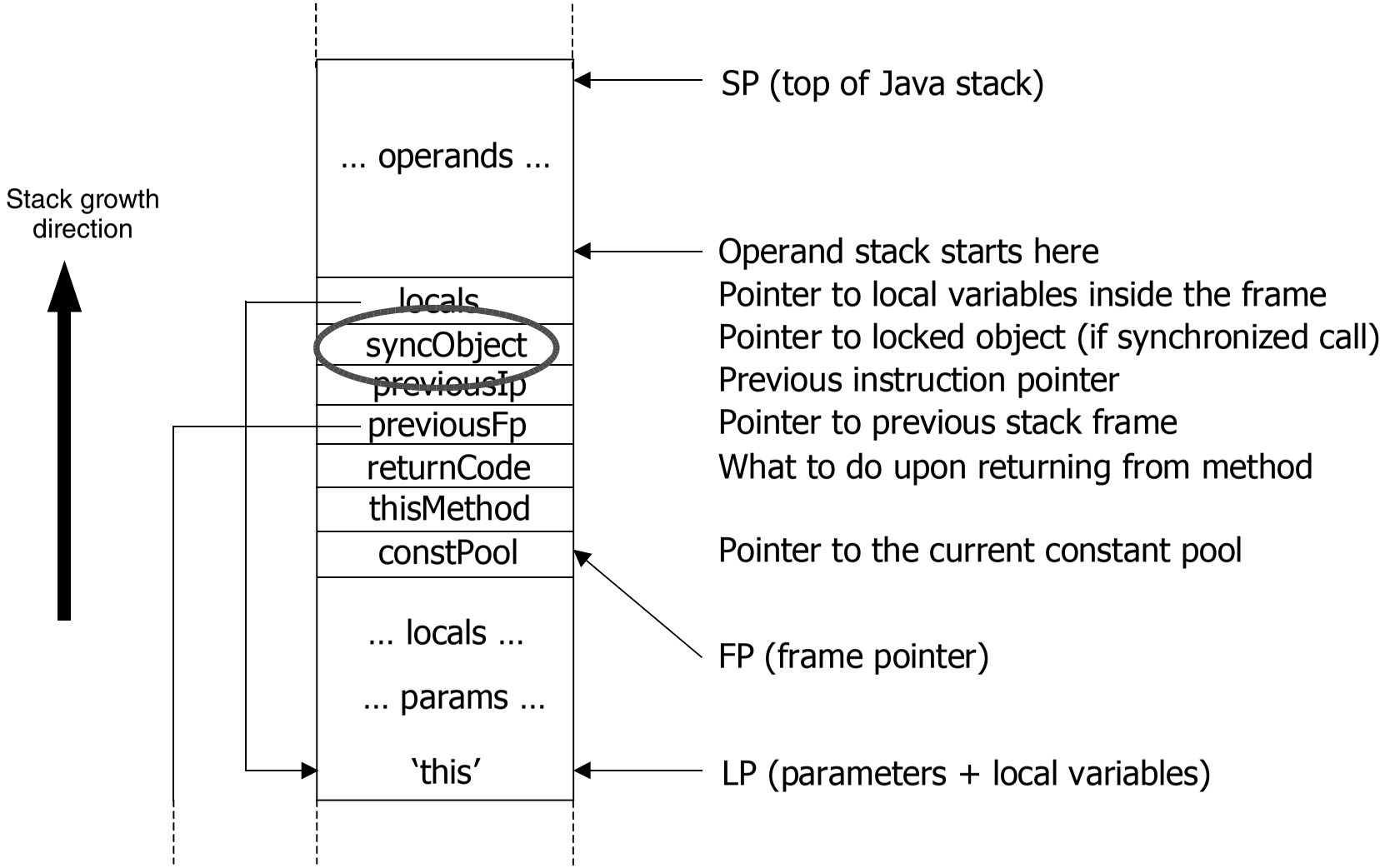
Key idea:
Locks are stored on the execution stack; no need to create lock objects in heap

Performing Synchronized Method Calls in Java

- When calling synchronized methods (vs. entering synchronized blocks), the monitor information is stored in the stack frame.
  - Synchronization information is needed in the stack frame so that locks can be released if an exception occurs!!
- If the monitor of the receiver of the method is already reserved, method invocation must wait until the previous owner has exited the method.

```
synchronized void foo() {
  ...only one thread can execute this code at the same time...
}
```

Example: Performing Synchronized Method Calls in Java (KVM)



Timers

Timer Support

- Many programming language, including Smalltalk and Java, allow the definition of *timers.*
- When a thread is associated with a timer, the execution of the thread is suspended until the specific time occurs.
- Allows “sleeping” and the definition of time-triggered events:

```
public void run() {
    Thread.sleep(5000); // Wait 5 seconds
    ... do something ...
}
```

Implementing Timers

- A naïve timer implementation uses *busy wait*.
  - Loop until the requested time occurs.
  - Not suitable for battery-operated devices!!
- A better implementation uses a *timer queue* that holds those threads that are waiting for a timer release.
- The interpreter loop or the interrupt handler must periodically check the threads in the timer queue.
  - Compare the requested event time in each waiting thread with the current time.
  - After enough time has passed, release and schedule the waiting thread(s) for execution.

Using Native Functions to Access External Resources

Problems with Native Function Calls

- A virtual machine commonly needs to call various native functions to perform I/O, access external resources etc.
  - For instance, a VM might have to call I/O routines that are specific to Win32, Linux or Symbian OS.
- In a VM that uses non-native threads, the rest of the VM is “blocked” when a native function is called.
  - There is only one physical thread of control.
  - If the native function does not return immediately, all the other threads in the VM are stuck indefinitely until the native function returns.
  - In the worst case, the entire system may deadlock if the current thread is waiting for something from another thread.

Possible Solutions

❶ All native functions must be designed so that they return immediately.
  – Might have to wrap some I/O calls with code that uses polling / busy waits to check if data is available.

❷ A pool of native threads might be used at the implementation level to perform I/O calls asynchronously.
  – Allows the VM to continue execution until data is present.

- (2) is a better choice but not fully portable.
  – KVM provides both choices (as a compile-time option).
- VMs that use native threads don't have these problems (but have many other thread-related problems instead).

Advanced Topics

Battery Conservation Issues

- In mobile virtual machines, battery conservation is critical.
- The VM must call system-specific battery conservation / hibernation features whenever the VM has nothing to do!
  - For instance, if the VM currently has only one thread that is sleeping or waiting for an I/O operation to complete, the VM should not simply wait in a busy loop.
  - The interpreter loop must be designed so that it calls system-specific battery conservation functions whenever there is only one thread that has nothing urgent to do.
- This is more complicated that it sounds.
  - There are conflicts with performance requirements, I/O latency requirements, etc.

Multithreading in High-Performance Virtual Machines

- In high-performance VMs, multithreading and synchronization are challenging to implement.
  - For instance, the machine code generated by an adaptive compiler might have to include code to perform thread switching.
  - Otherwise, tight loops would prevent thread switching!
  - In a VM with native threads, generated machine code needs to include the necessary mutexes to avoid machine code being pre-empted in wrong places.
  - In general, one should try to minimize thread-specific information in the generated code.
  - For instance, inline caching might not work properly, because method call information of one thread would be overwritten by another thread.
- We will hear more about these problems on Oct 29

Real-Time Support

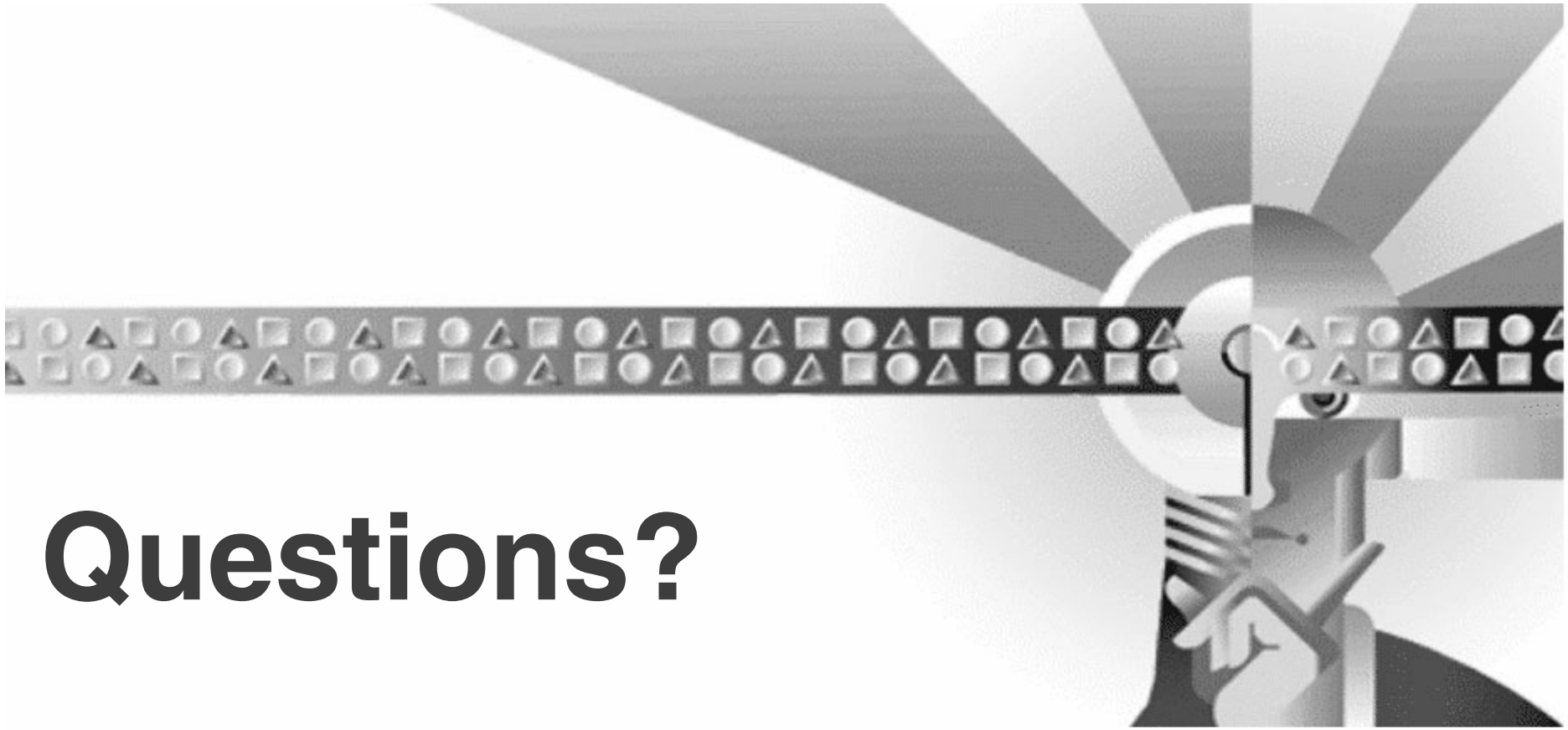
- Some systems have real-time (RT) constraints.
  - Predictable response times/latencies.
  - Tasks guaranteed to finish on time.
- Hard vs. soft real-time.
  - Hard RT: timing failures are potentially life-threatening.
- Hard real-time virtual machines are very difficult to implement.
  - Scheduling and thread behavior must be totally predictable.
  - Lots of implications for interrupt handling, memory management, exception handling, synchronization, ...
- See "*Real-Time Specification For Java*":
  - http://www.rtj.org/

From Threads to Isolates

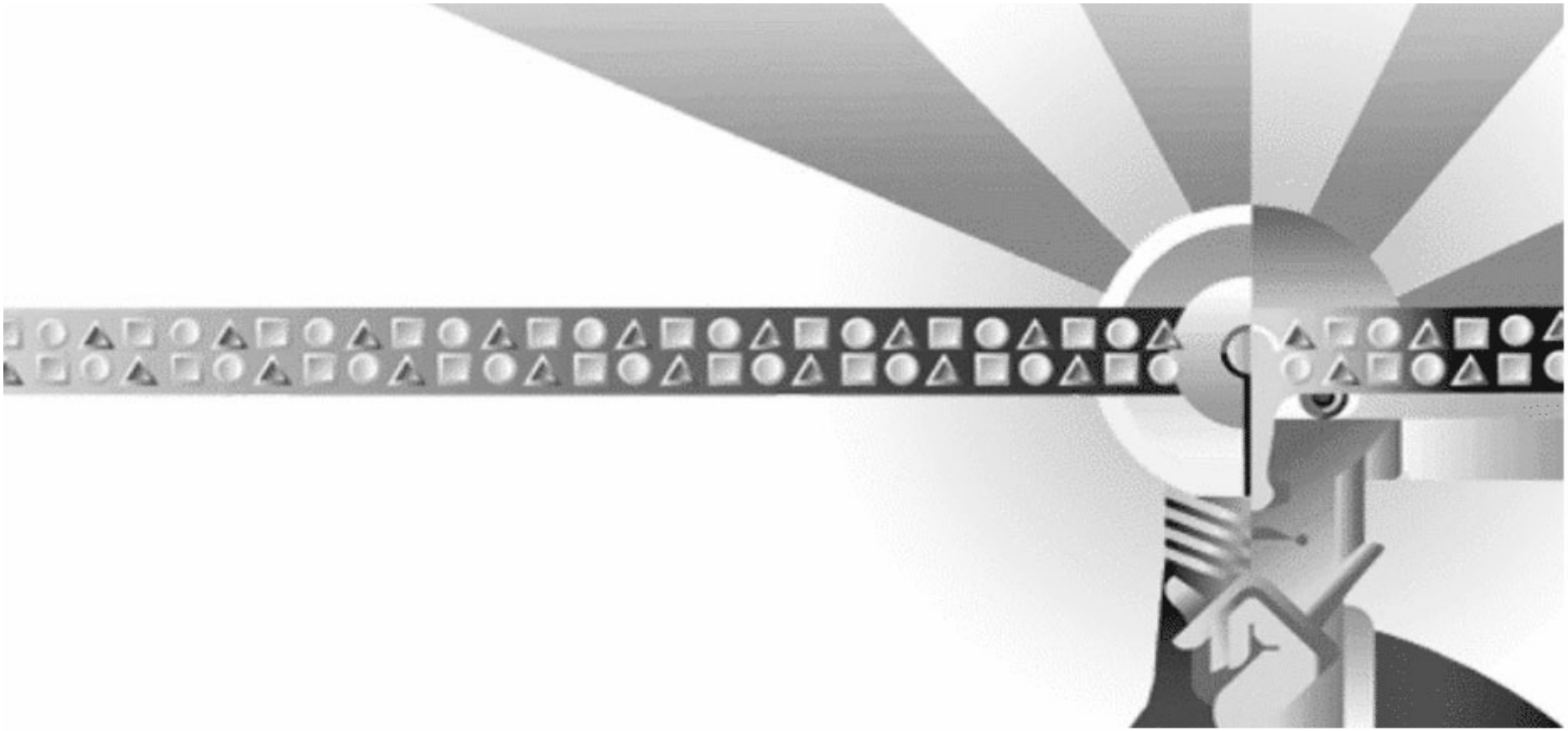
- In most virtual machines, all threads live in the same global name space.
  - All threads share the classes, methods, exceptions, as well as object instances and static variables that are currently in the system.
  - There is very little protection to prevent one thread from modifying the data of another thread.
  - For instance, in a Smalltalk VM one thread could easily rename, replace or remove the classes or data used by another thread.
- Some Java virtual machines support the concept of "*isolates*" to support multithreading in a more protected fashion.
  - See: http://jcp.org/en/jsr/detail?id=121

Isolates

- Each isolate can be viewed like a separate “logical” virtual machine.
  - No sharing of name spaces.
  - Actual class implementations may be shared, but all static variables are duplicated/reinitialized for each isolate.
  - All object instances are specific to a certain isolate; an instance created in one isolate cannot access the data of another isolate.
- Isolates are becoming popular also in the J2ME area.
  - Allow multiple MIDlets to be run safely inside one physical virtual machine; no need to launch multiple operating system processes for multiple VM instances.



Questions?



antero.taivalsaari@sun.com

Sun microsystems

We make the net work.