# Why is Concurrent Programming required?

* Performance gain from multiprocessing hardware

– Parallelism

* Increased application throughput

– A blocking I/O call only blocks one thread

* Increased application responsiveness

– High priority thread for user requests

• More appropriate structure

– For programs which control multiple activities and handle multiple events

• Embedded systems for driving hardware (like Therac-25)

– Equipment might naturally consist of multiple sensors/activators

• Distributed systems

# Concurrency abstraction

The concurrency abstraction ignores all aspects of time –

each atomic action takes an arbitrary 1 unit of time.

It allows us to think about what possible scenarios may result within a concurrent system.

It allows us to not worry about whether things are really happening in parallel or other particular aspects of real time.

It models concurrent systems as interleaving of atomic actions.

# Relationship between OS Process and Threads

OS Process consists of

– One address space containing memory segments

– Shared I/O resources

– Multiple threads

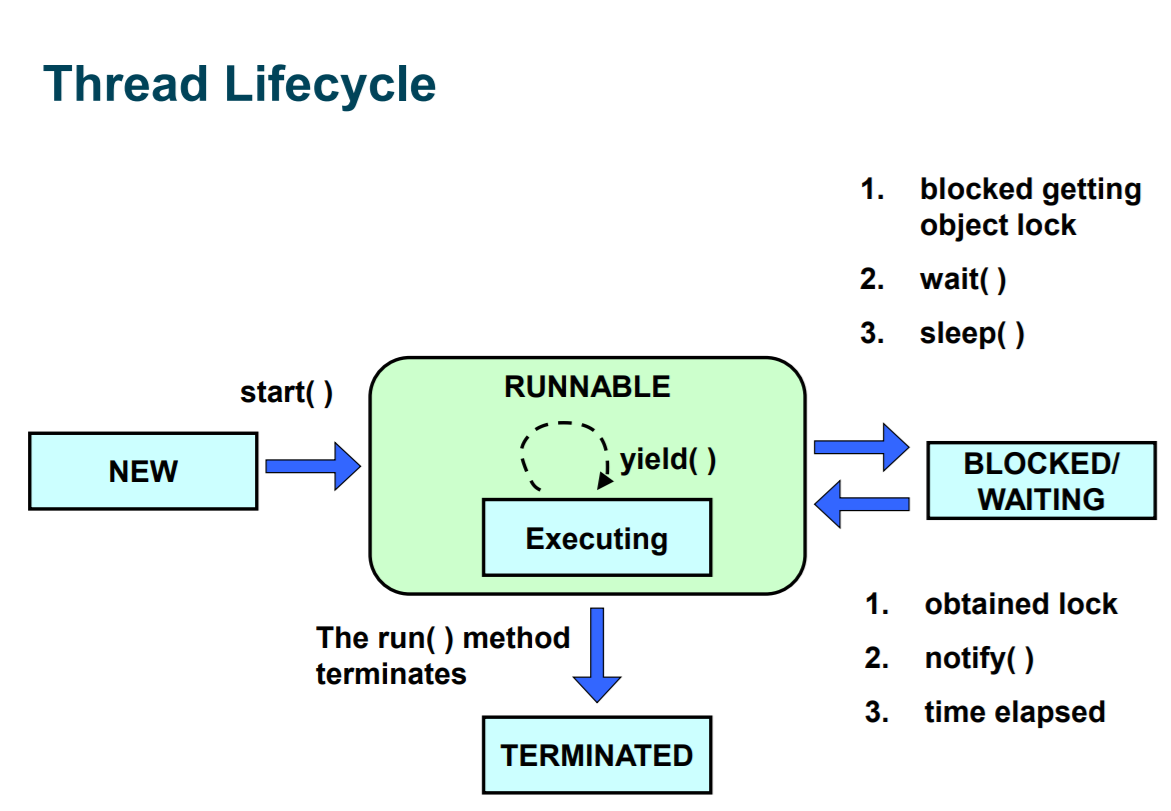
• Each with its own set of registers, in particular

PC, stack pointer and general purpose registers.

• Each has its own stack in the address space

• Code and data is shared.

• The OS schedules schedules threads to processor cores to keep them busy.



**When a Thread is ‘RUNNABLE’ … it may not be**

**executing on the processor ...**

The Thread may be waiting for other resources from the

operating system such as a physical processor (CPU cores

are shared between different threads by the OS scheduler).

Or it may be waiting on I/O and hence cannot execute.

**Threads can enter the BLOCKED/WAITING state by:**

1. blocked getting object lock

2. wait( )

3. sleep( )

**Exist by:**

1. obtained lock

2. notify( )

3. time elapsed

# Interference and Mutual Exclusion

Destructive update, caused by the arbitrary interleaving of read and write actions, is termed interference. This is a type of race condition which are bugs caused when threads have unfortunate timings against each other.

Interference bugs are extremely difficult to locate. The general solution is to give methods mutually exclusive access to shared objects.

Concurrent execution of methods in Java can be made mutually exclusive by prefixing the method with the keyword **synchronized**.

# What does the “volatile” keyword do ?

• Marking a variable with the volatile keyword is a more

efficient way to guarantee all threads will read and write this variable directly from main memory so this variable is visible between all threads.

• Using a volatile for variable visibility is more efficient

that locking since threads will not be blocked while accessing it.

• BUT it does not make modifications mutually exclusive ... so “x++;” will still interfere even when x is marked as volatile (it needs full synchronization to stop this interference ... and then **volatile is not needed since the synchronization guarantees visibility**).

All assignment of primitive types in Java are specified (“The Java Language Specification” by Gosling et al.) to be atomic EXCEPT …

• When you assign either longs or doubles.

• WHY ?

• A hardware consideration … these are 64-bit

values and a lot of 16/32-bit hardware do not have

atomic store instructions for 64-bit values and so

Java permits long/double assignments to be

non-atomic

Even within a shared memory system – a variable

written by one thread may not be visible to other

threads in a timely manner without proper synchronization. Other threads may see old stale data.

• Why ?

• The compiler may store a variable in a processor register so changes cannot be seen by other threads !

• Even if a processor (core) writes the variable to main memory ... it may just change its local cache and not be written to the main memory (or other caches) in a timely manner ... so again changes cannot be seen by other threads

# The key “Happens-Before” Rules of the JMM

• Program order rule: Each action in a thread happen-before every action in that thread that comes later in the program order.

• Monitor lock rule: An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.

• Volatile variable rule: A write to a volatile field happens-before every subsequent read of that same field.

• Transitivity: If A happens-before B and B happens-before C then A happens-before C

# Ways to “safely published” objects …

1. Initializing an object reference from a static initializer.

2. Storing a reference to it into a volatile field (or Atomic Reference).

3. Storing a reference to it into a final field of a properly constructed object.

4. Storing a reference to it into a field that is properly guarded by a lock.

Safely published: Both the reference to the object and the objects latest state will be made visible to other threads at the same time.

An object is immutable if

– Its state cannot be modified after construction.

– All its fields are marked final.

– It is properly constructed (the this reference does not

‘escape’ during construction so that threads could see a

partially constructed object).

• Immutable objects are always thread-safe.

• Immutable objects can be published through

any mechanism.

An object is effectively immutable if

– Its state does not change after publication.

– It doesn’t need to obey all the conditions given earlier of

a proper immutable object.(So our previous Holder object is actually effectively immutable ...)

• These objects can be safely used between threads if they are “safely published”.

• All threads will then be guaranteed to see the “as-published” version of the object as soon as the object reference is visible.

An object is mutable if

– Different threads change it state after publication.

• Mutable objects must be “safely published” (so

threads see them as their “as-published” state).

• Also they must either be thread-safe or

guarded by a lock so that changes that occur

to the state of the object are visible.

• We will be looking at the “monitor design” in

the next lecture for such a situation.

• Two main classes of properties that exist for

concurrent programs:

– Safety Properties: assert that nothing ‘bad’ will

ever happen during any execution (the program

will never enter a ‘bad’ state)

- Interference

- Deadlock

– Liveness Properties: assert that something ‘good’

will eventually happen during every execution

Definition: In live systems, every activity progress towards completion, every method invoked eventually executes.

**Types of liveness issues**

• Deadlock with threads not progressing since they are waiting for each others’ resources.

– Only solution in Java is to eliminate the cyclic dependency in the actual code (allocating resources to different threads in same order)

• Livelock which is similar to deadlock but instead of being blocked ... threads constantly respond to each other and simply do not make progress.

• Starvation where threads do not ever get a particular resource (often CPU cycles) to make progress due to the structure of the program (or overloading of system resources).

# Deadlock Conditions

• Deadlock exists if all 4 of these conditions hold in a

system:

– Shared resources with mutual exclusion: if a

resource is being used, other processes need to wait.

– Hold-and-wait: Processes hold only resources while

waiting to acquire additional resources

– No pre-emption: Resources cannot be pre-empted

(forcefully withdrawn) - only released voluntarily by a

process.

– Wait-for Cycle: A cycle of processes exists such that

each holds a resource requested by next process in the

cycle (and refused).

# Synchronized Methods

• Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called.

• Other threads cannot enter any other synchronized method on the same object until the object is unlocked. They have to wait (they are in the blocked state).

• The acquisition and release of a lock is done automatically and atomically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity.

# What do we mean by a Monitor and Conditional Synchronization?

• The term Monitor is used inconsistently within different

threading contexts & languages …

• We will employ the Hansen and Hoare's idea of a Monitor

as first and foremost a data abstraction mechanism for

concurrent systems.

• A Monitor encapsulates the internal state representing an

object and provides a set of operations that are the only

means to manipulate the internal state.

• A Monitor only allows one Thread to modify the

representation at a time (mutual exclusive access)

• Monitors can support conditional synchronization by

only allowing methods to be run when the internal state of

the object is appropriate (guarded actions).

# Condition Synchronization

We refer to a thread entering a monitor when it acquires the mutual exclusion lock associated with the monitor and exiting the monitor when it releases the lock.

wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.

**Java 5 concurrency classes**

• Safety

– Lock, ReentrantLock, Condition provide features similar

to synchronized methods & wait/notify mechanisms.

(All these are in the java.util.concurrent.locks package.)

– A large number of pre-written thread-safe container

classes to handle data in a thread-safe manner such as

atomic datatype containers, different types of arrays,

lists, queues, hashmaps, etc.

• Control of path of execution of multiple threads.

– A number of classes that implement higher-level

concurrency control mechanisms such as counting

semaphores, cyclic barriers, blocking queues and

thread pools.

**Semaphore Class**

• A Semaphore object is constructed with a given number

of ‘permits’.

• The key methods are acquire() and release(), although

other methods allow you to obtain a number of permits

atomically and also ‘polling’ to see if permits are

available (tryAcquire, etc.)

• One of the constructors can take the boolean value true

which turns the semaphore into a ‘fair’ semaphore

**CyclicBarrier** and **CountDownLatch** Classes

• These act as barriers to threads – it causes them to wait

until a condition is true.

• The key method is await() – causing the Thread to wait

until the required condition. The CountDownLatch also

has a countDown() method.

**ReadWriteLock** Class

• Sometimes we have the situation where a number of

Threads want to read the values of an object, and only

infrequently will a Thread want to write to the object.

• In this case it is more efficient to allow multiple Threads

to simultaneously read the state of the object, while only

allowing one Thread mutual exclusive access to write to

the object.

• The key methods are readLock() and writeLock() which

obtain, respectively, read and write locks (which

implement the Lock interface). These can then be used

to lock the object for reading and writing.