# 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

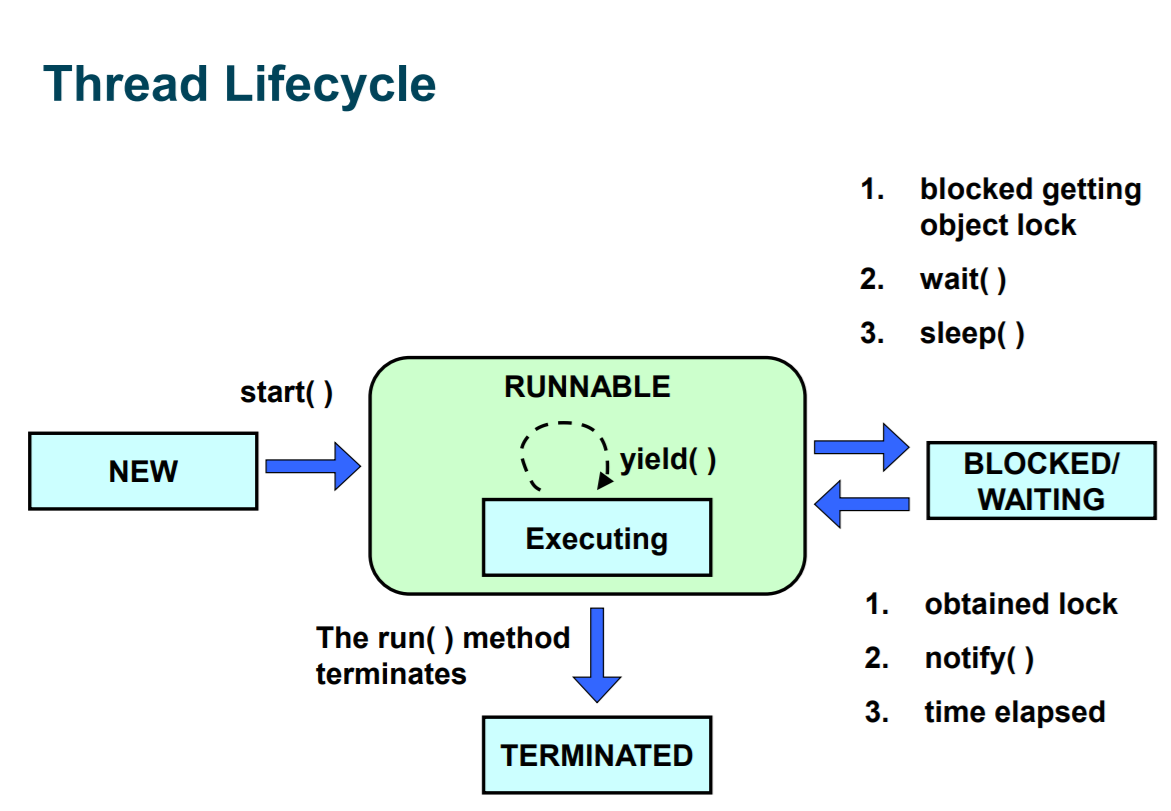
• Each with its own set of registers, in particular

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