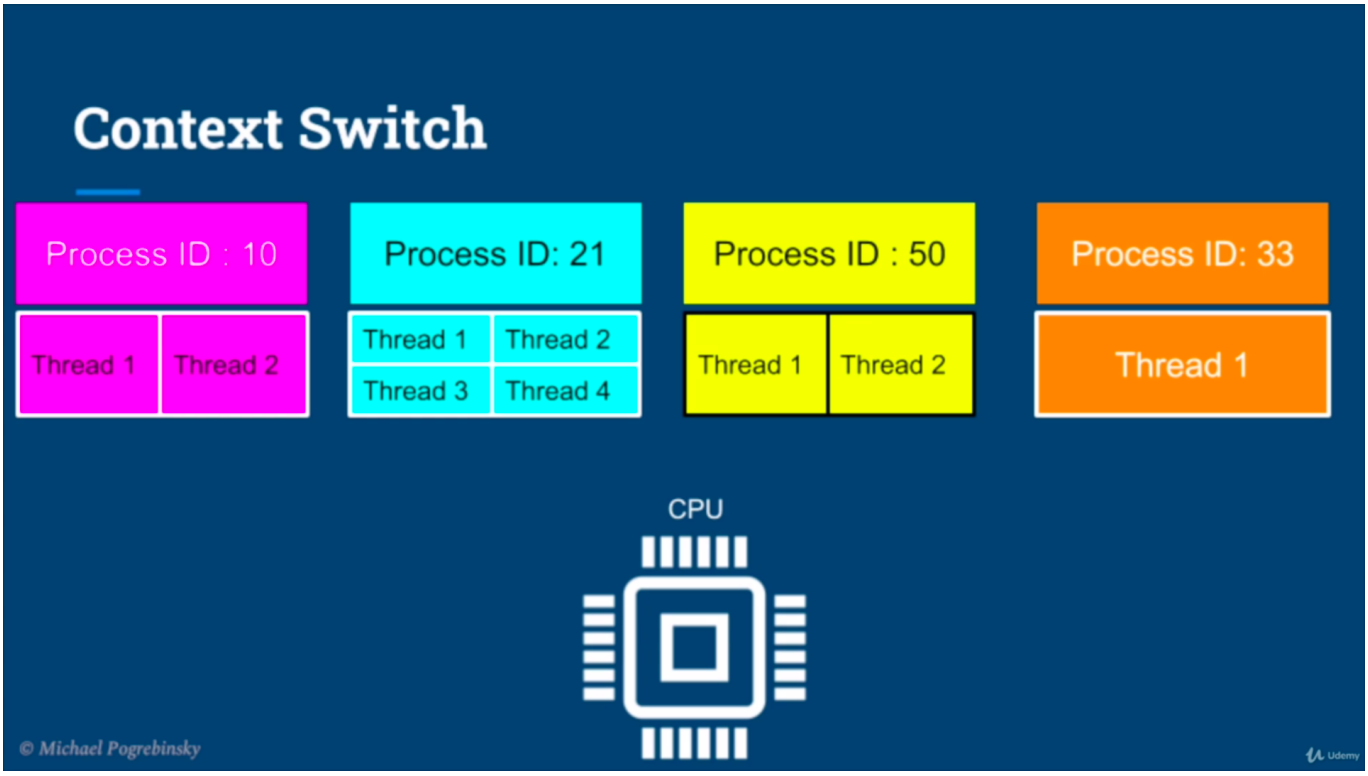
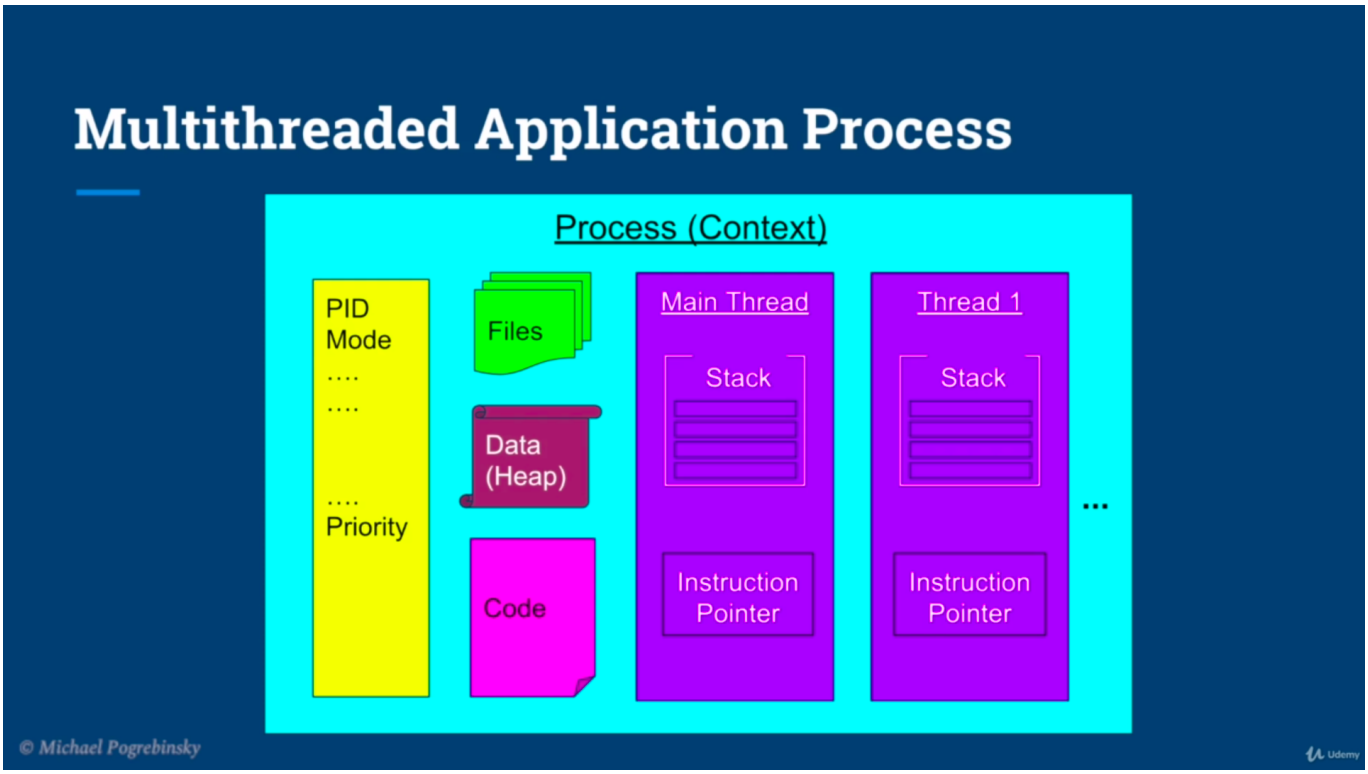
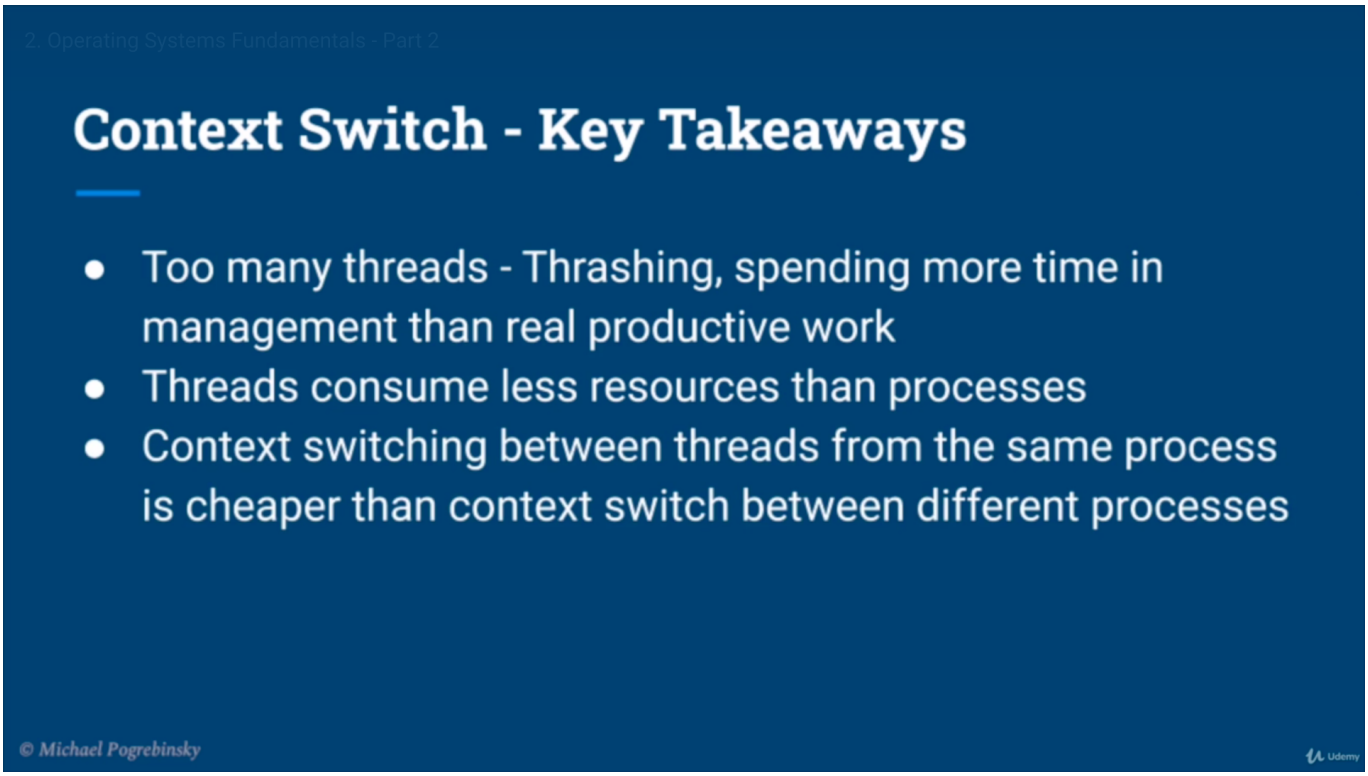
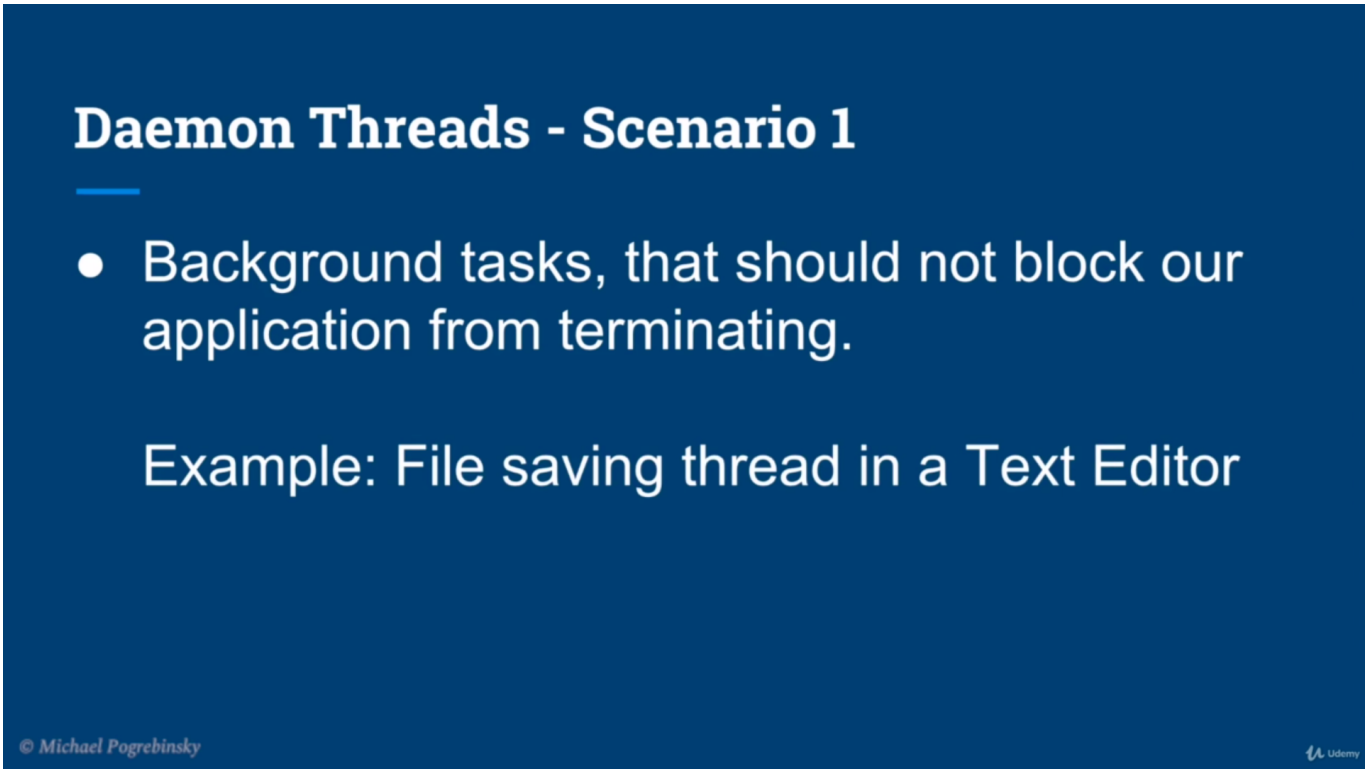
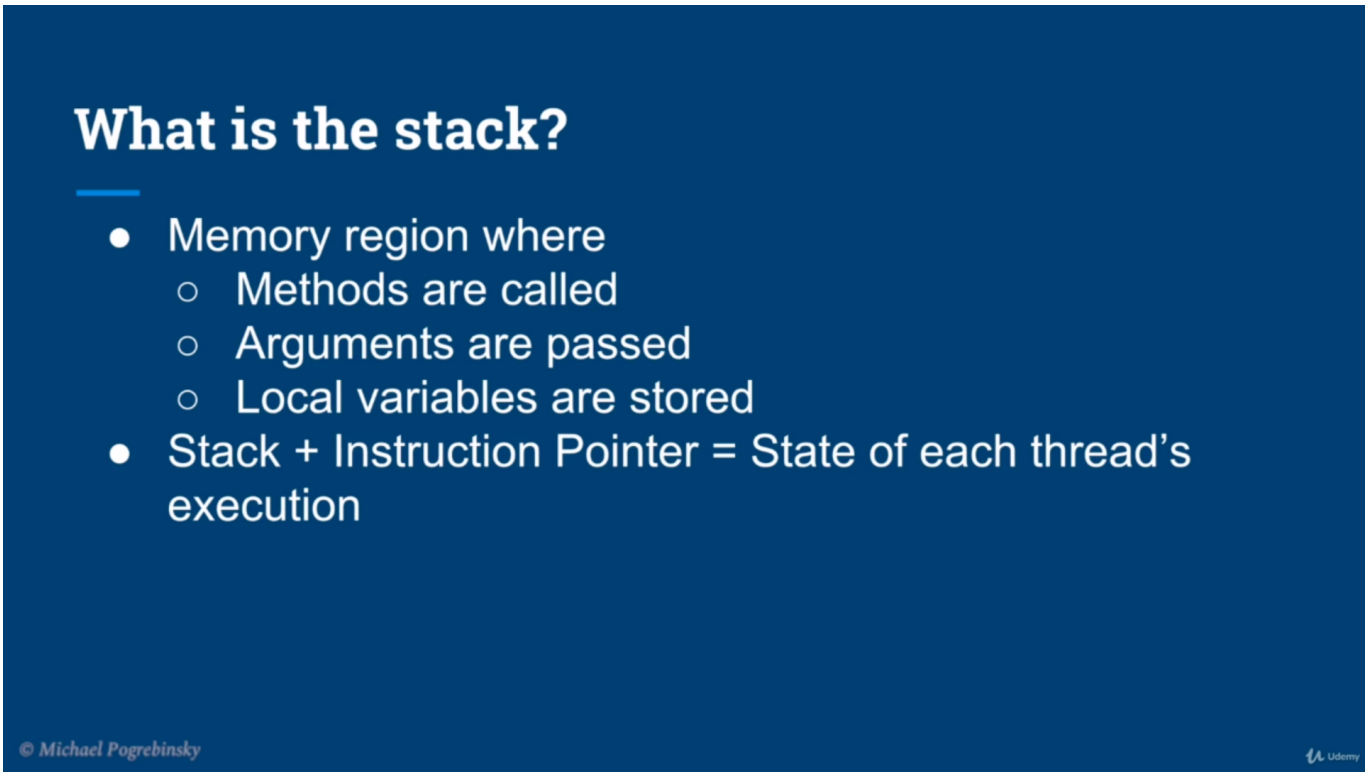
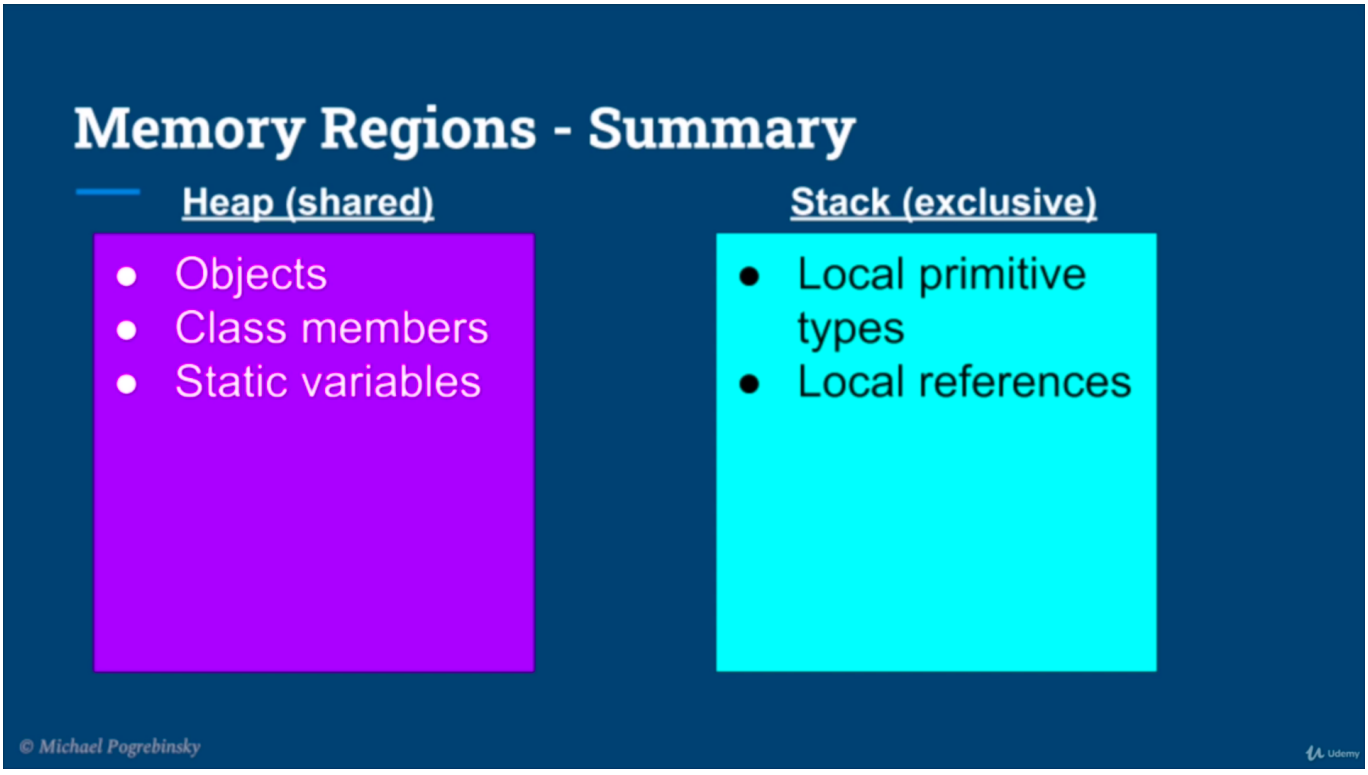

Context Switch
Process ID : 10
Thread 1
Thread 2
Process ID: 21
Thread 1
Thread 2
Thread 3
Thread 4
Process ID : 50
Thread 1
Thread 2
Process ID: 33
Thread 1
CPU
© Michael Pogrebinsky

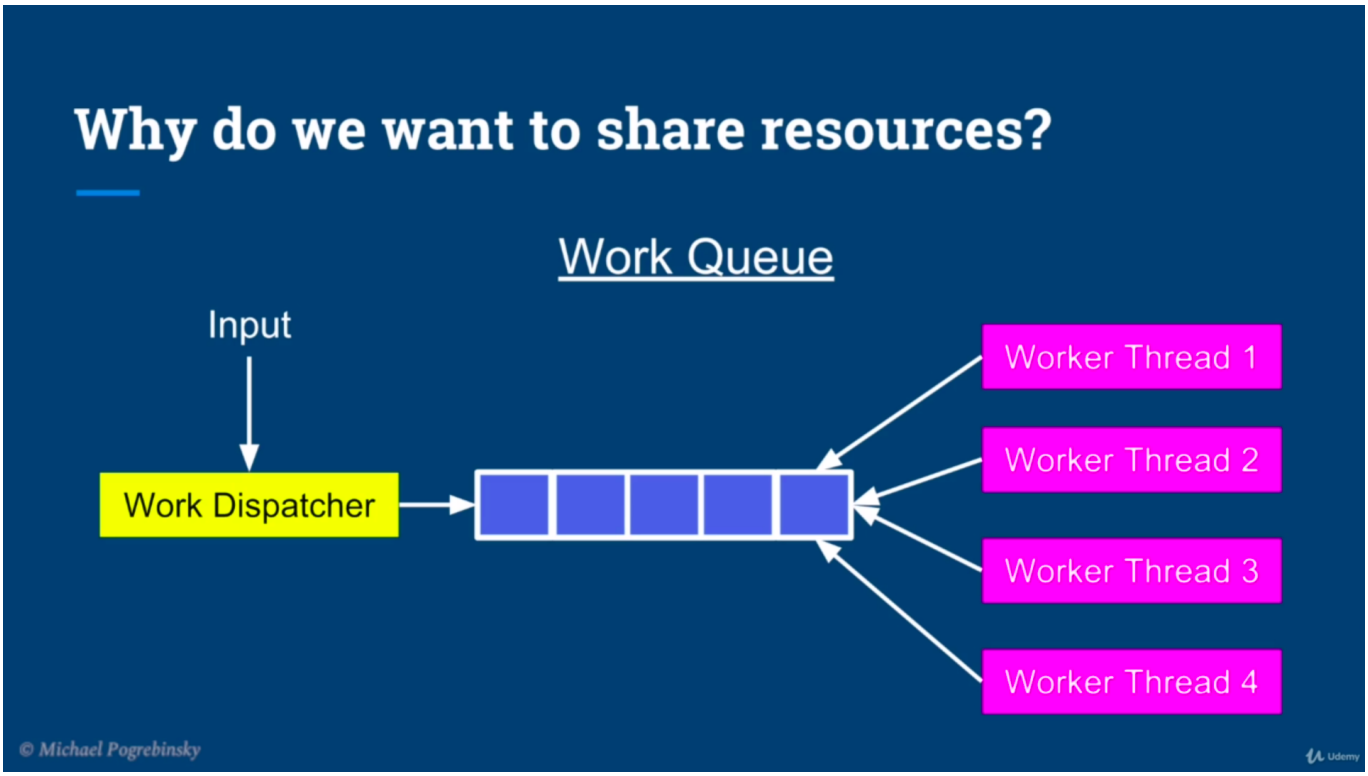

Multithreaded Application Process
Process (Context)
PID
Mode
....
....
....
Priority
Files
Data
(Heap)
Code
Main Thread
Stack
Instruction
Pointer
Thread 1
Stack
Instruction
Pointer
...
© Michael Pogrebinsky

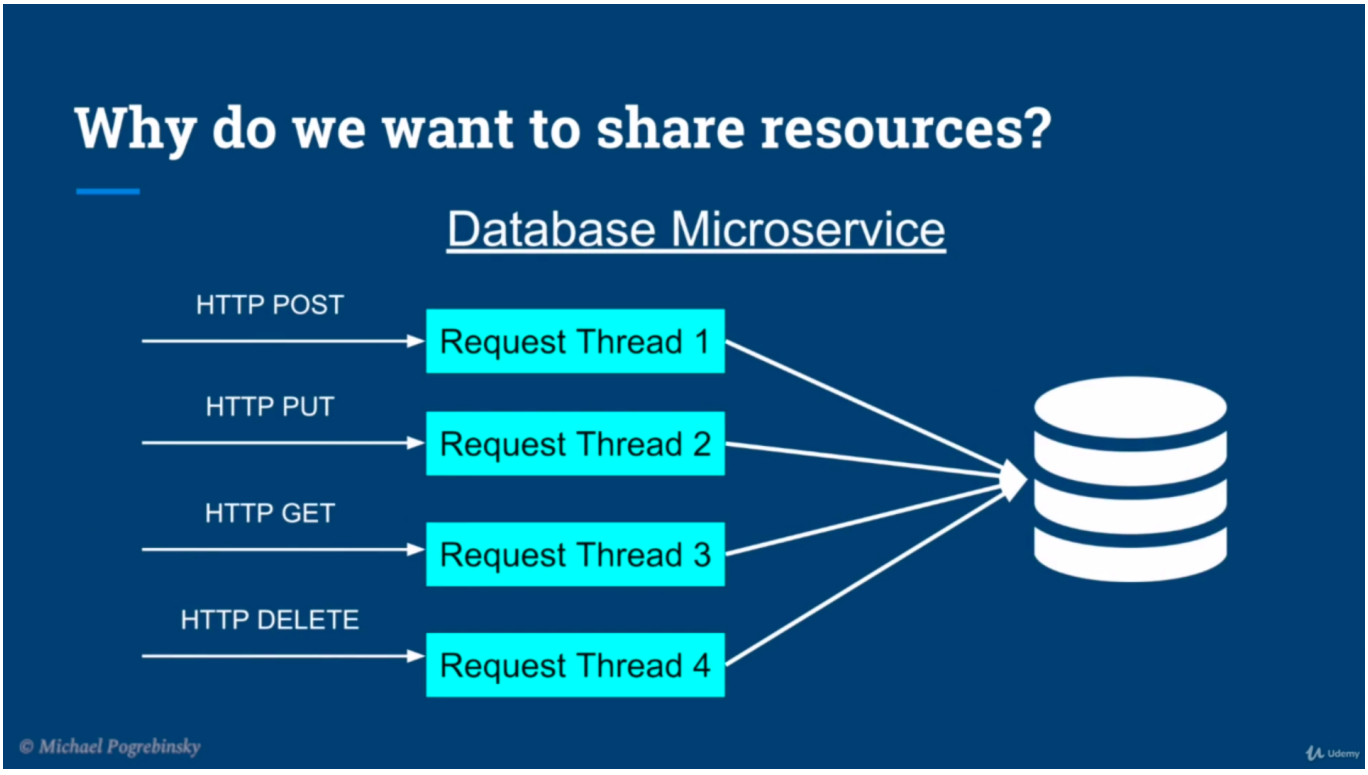

2. Operating Systems Fundamentals - Part 2
Context Switch - Key Takeaways
Too many threads - Thrashing, spending more time in management than real productive work
Threads consume less resources than processes
Context switching between threads from the same process is cheaper than context switch between different processes
© Michael Pogrebinsky
Udemy


Daemon Threads - Scenario 1
Background tasks, that should not block our application from terminating.
Example: File saving thread in a Text Editor
© Michael Pogrebinsky
Udemy


What is the stack?
Memory region where
Methods are called
Arguments are passed
Local variables are stored
Stack + Instruction Pointer = State of each thread's execution
© Michael Pogrebinsky
Udemy


Memory Regions - Summary
Heap (shared)
Objects
Class members
Static variables
Stack (exclusive)
Local primitive types
Local references
© Michael Pogrebinsky
Udemy


Why do we want to share resources?
Work Queue
Input
Work Dispatcher
Worker Thread 1
Worker Thread 2
Worker Thread 3
Worker Thread 4
© Michael Pogrebinsky

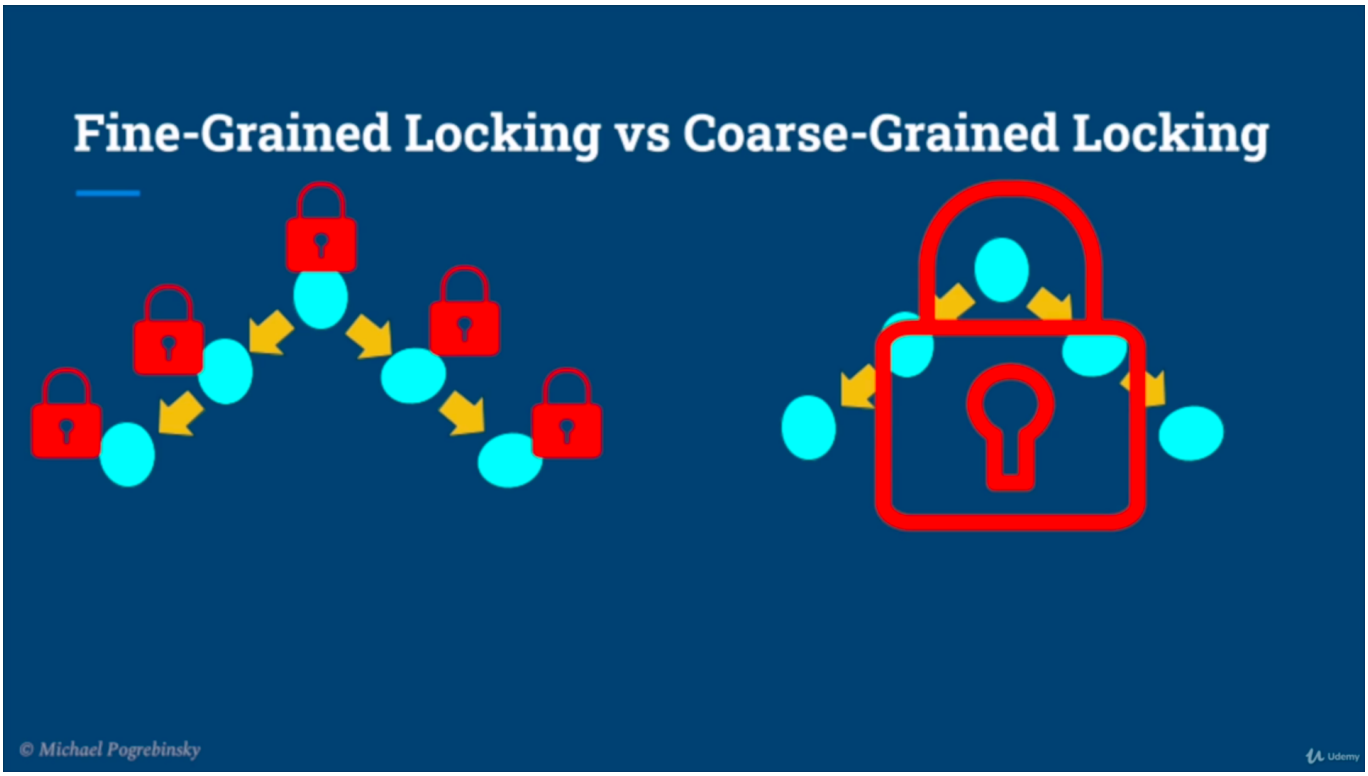

Why do we want to share resources?
Database Microservice
HTTP POST
HTTP PUT
HTTP GET
HTTP DELETE
Request Thread 1
Request Thread 2
Request Thread 3
Request Thread 4
© Michael Pogrebinsky

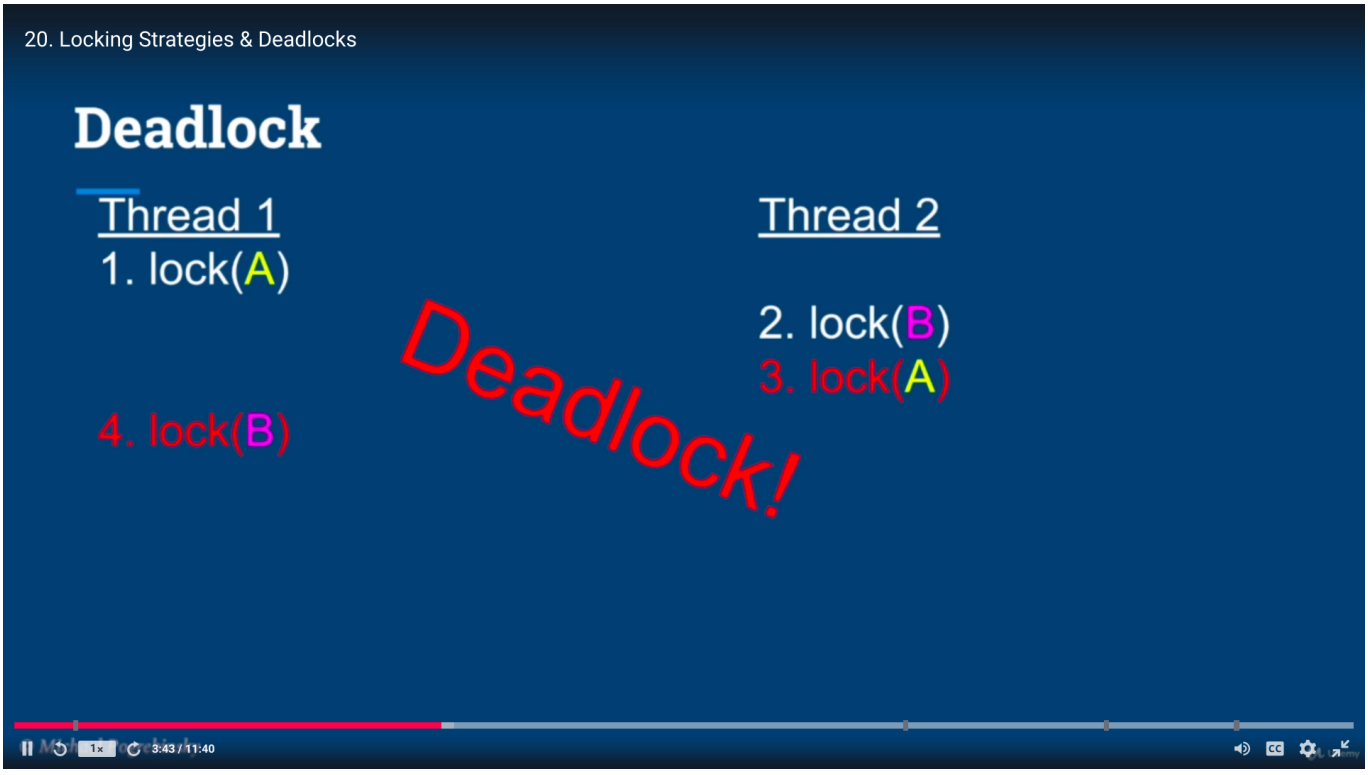
Race Condition

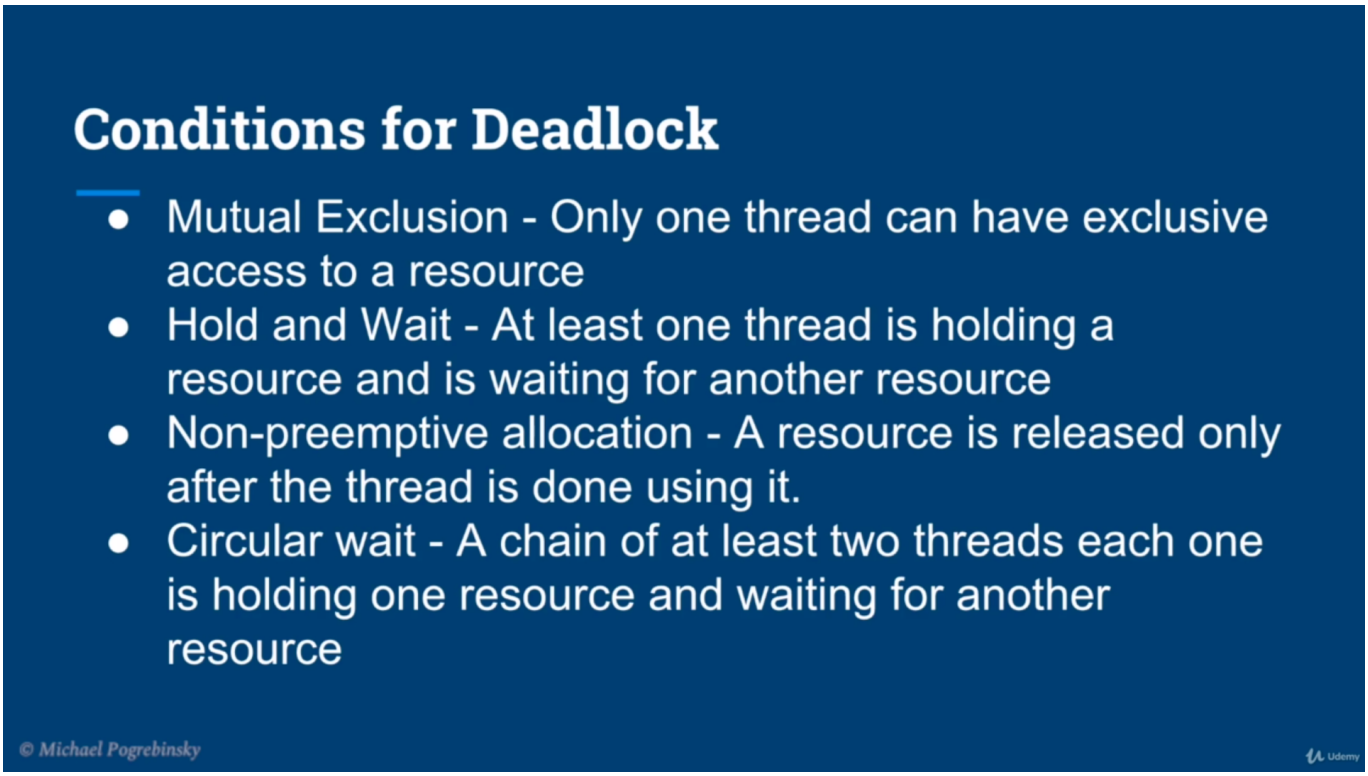
- Condition when multiple threads are accessing a shared resource.
- At least one thread is modifying the resource
- The timing of threads' scheduling may cause incorrect results
- The core of the problem is non atomic operations performed on the shared resource

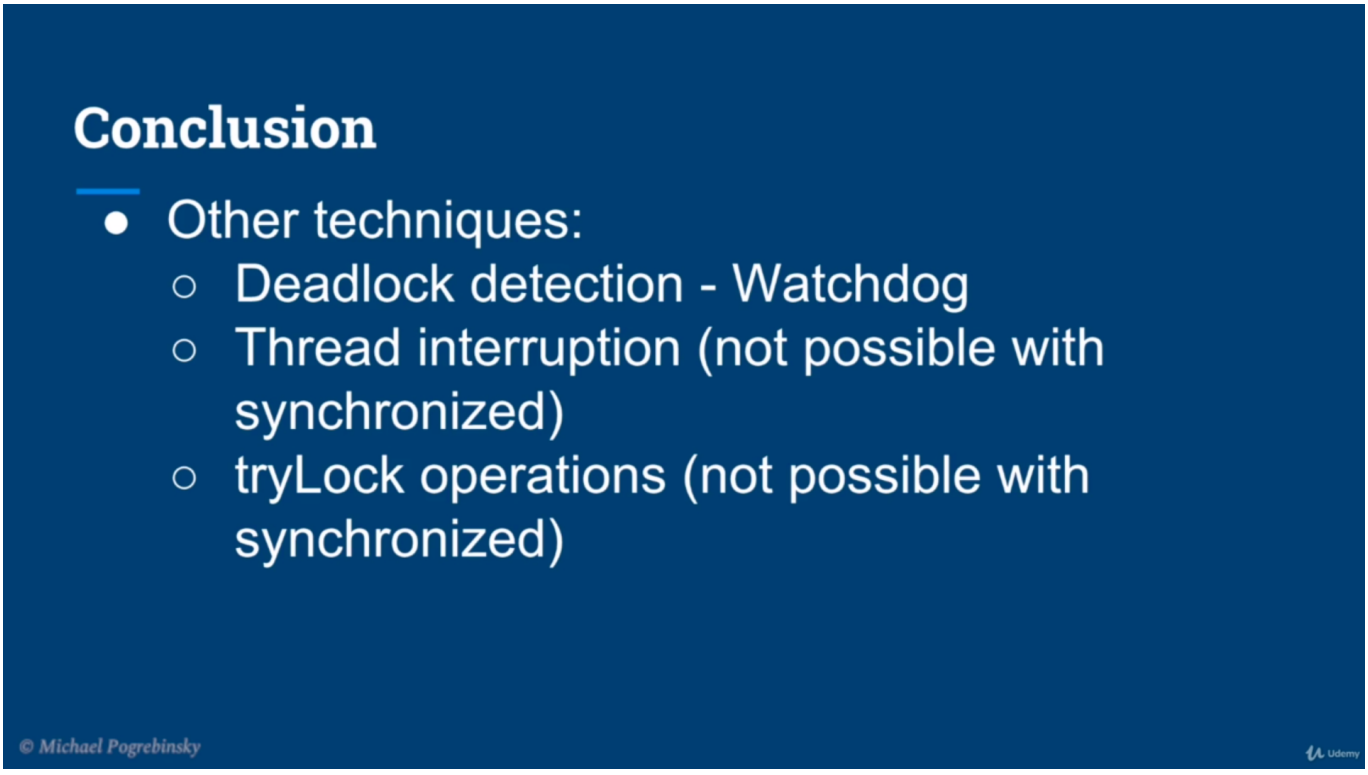
Data Race - Solutions

```
volatile int sharedVar;
public void method() {
   …. // All instructions will be executed before
   read/write(sharedVar);
   …. // All instructions will be executed after
}
```


Fine-Grained Locking vs Coarse-Grained Locking
© Michael Pogrebinsky


20. Locking Strategies & Deadlocks
Deadlock
Thread 1
1. lock(A)
4. lock(B)
Thread 2
2. lock(B)
3. lock(A)
Deadlock!
1×
3:43 / 11:40


Conditions for Deadlock
Mutual Exclusion - Only one thread can have exclusive access to a resource
Hold and Wait - At least one thread is holding a resource and is waiting for another resource
Non-preemptive allocation - A resource is released only after the thread is done using it.
Circular wait - A chain of at least two threads each one is holding one resource and waiting for another resource
© Michael Pogrebinsky
Udemy


Conclusion
Other techniques:
Deadlock detection - Watchdog
Thread interruption (not possible with synchronized)
tryLock operations (not possible with synchronized)
© Michael Pogrebinsky
Udemy

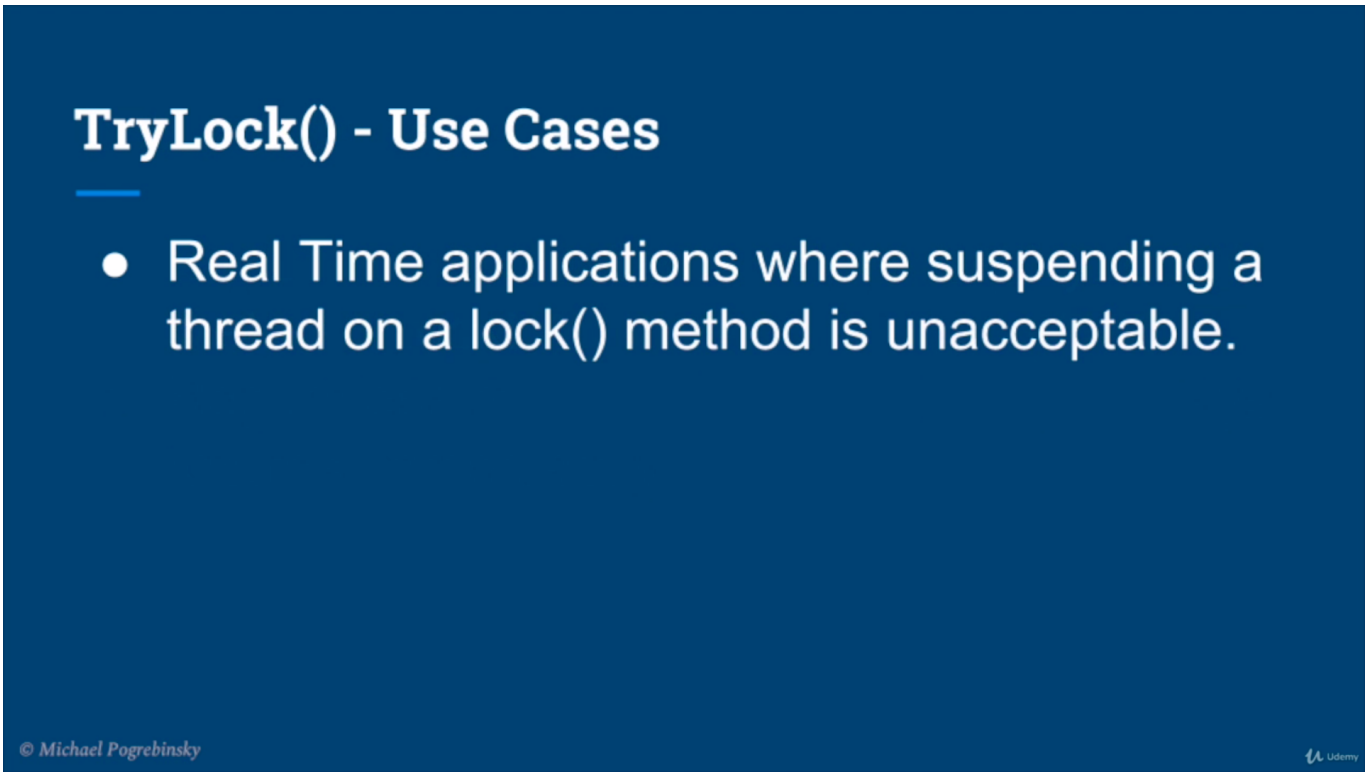
ReentrantLock

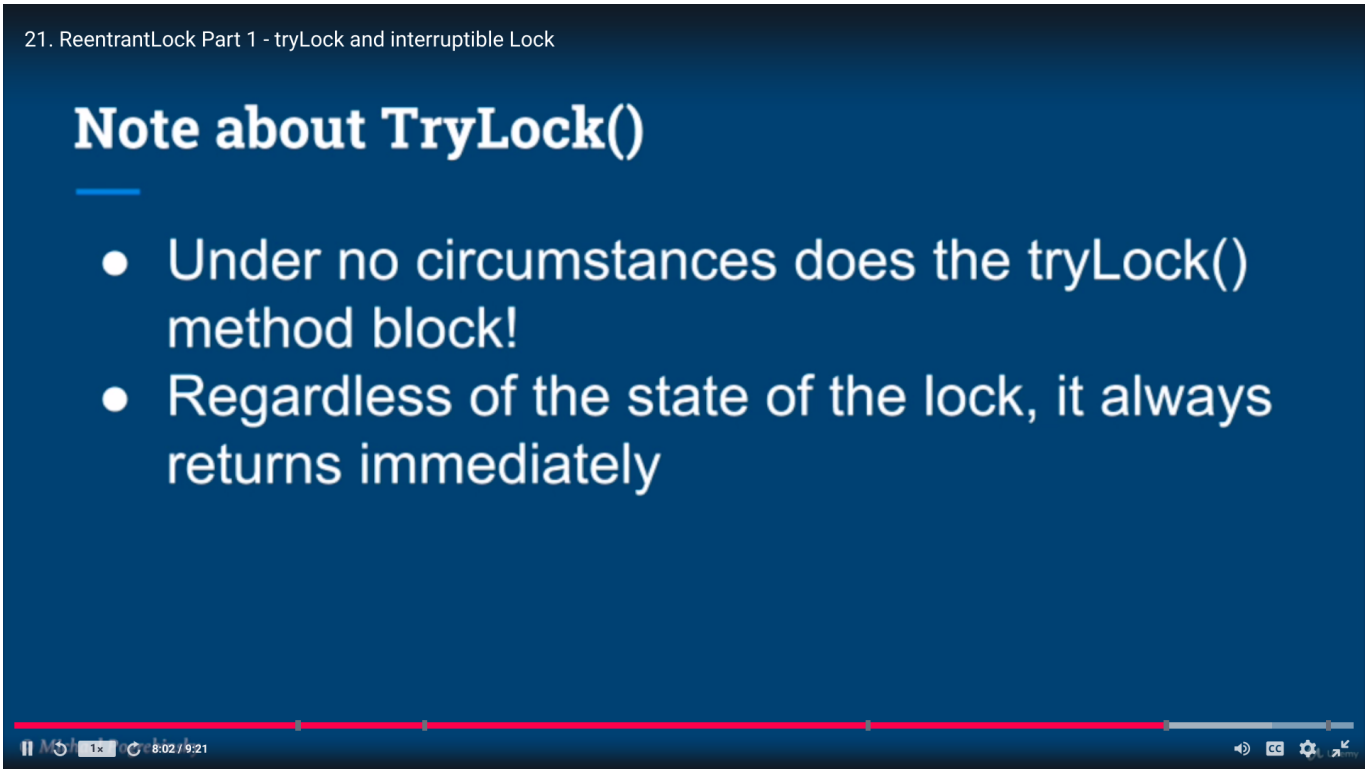
```
Object lockObject = new Object();
Resource resource = new Resource();
…
…
public void method() {
   synchronized(lockObject) {
      …
      use(resource);
   }

}
```

```
Lock lockObject  = new ReentrantLock();
Resource resource = new Resource();
…
…
public void method() {
   lockObject.lock();
   …
   use(resource);

   lockObject.unlock();
}
```


ReentrantLock - Solution

```
Lock lockObject  = new ReentrantLock();

public int use() throws SomeException {
   lockObject.lock();
   try {
        someOperations();
        return value;
   }
   finally {
      lockObject.unlock();
   }
}
```


TryLock() - Use Cases
Real Time applications where suspending a thread on a lock() method is unacceptable.
© Michael Pogrebinsky


21. ReentrantLock Part 1 - tryLock and interruptible Lock
Note about TryLock()
Under no circumstances does the tryLock() method block!
Regardless of the state of the lock, it always returns immediately
1×
8:02 / 9:21

TryLock() - Use Cases

- Real Time applications where suspending a thread on a lock() method is unacceptable.
- Examples:
  - Video/Image processing
  - High Speed/Low latency trading systems
  - User Interface applications

Semaphore - Producer Consumer with Queue

```
Semaphore full = new Semaphore(0);
Semaphore empty = new Semaphore(CAPACITY);
Queue queue = new ArrayDeque();
Lock lock = new ReentrantLock();
```

Producer:

```
while(true) {
    empty.acquire();
    item = produceNewItem();
    full.release();
}
```

Consumer:

```
while(true) {
    full.acquire();
    consume(item);
    empty.release();
}
```

Semaphore - Producer Consumer with Queue

```
Semaphore full = new Semaphore(0);
Semaphore empty = new Semaphore(CAPACITY);
Queue queue = new ArrayDeque();
Lock lock = new ReentrantLock();
```

Producer:

```
while(true) {
    Item item = produce();
    empty.acquire();
    lock.lock();
    queue.offer(item);
    lock.unlock();
    full.release();
}
```

Consumer:

```
while(true) {
    full.acquire();
    lock.lock();
    Item item = queue.poll();
    lock.unlock();
    consume(item);
    empty.release();
}
```


Semaphore - Producer Consumer

```
Semaphore full = new Semaphore(0);
Semaphore empty = new Semaphore(1);
Item item = null;
```

Producer:

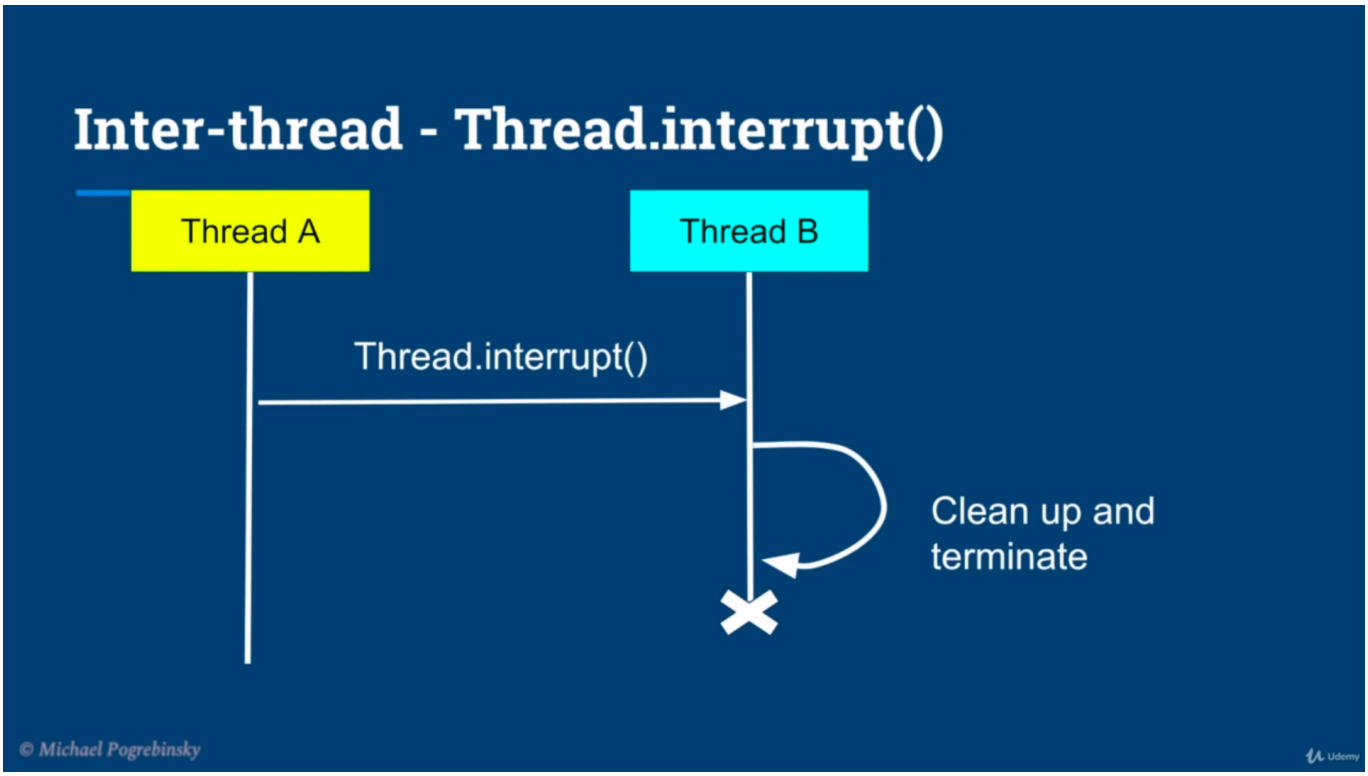
```
while(true) {
    empty.acquire();
    item = produceNewItem();
    full.release();
}
```

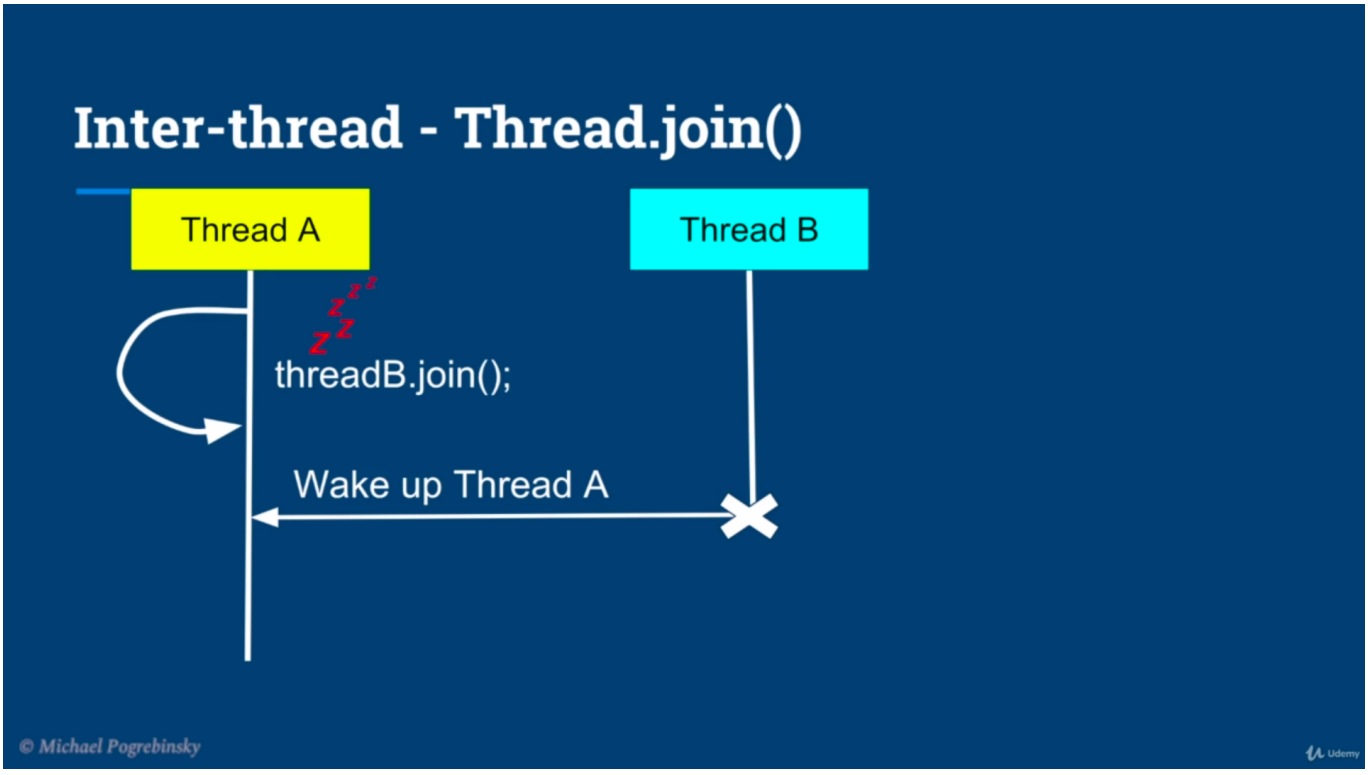
Producer

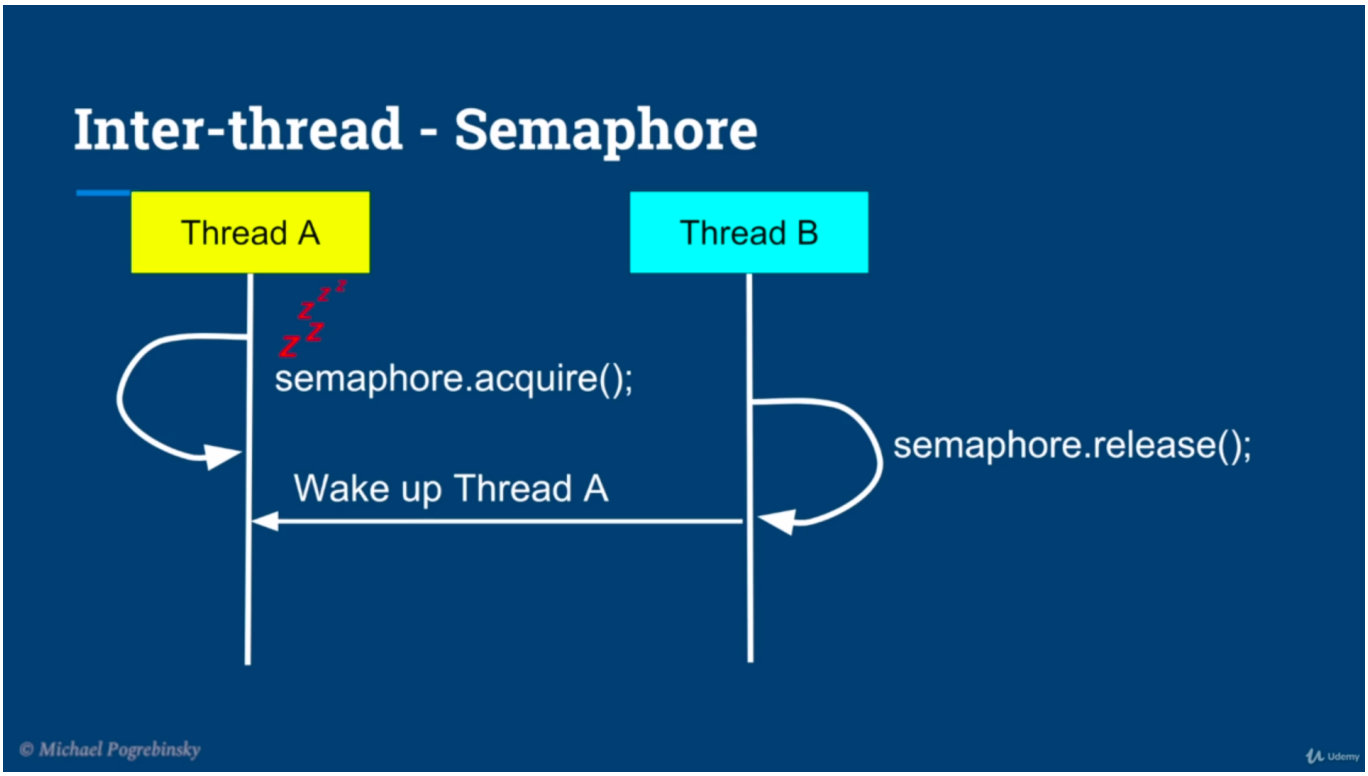
Consumer:

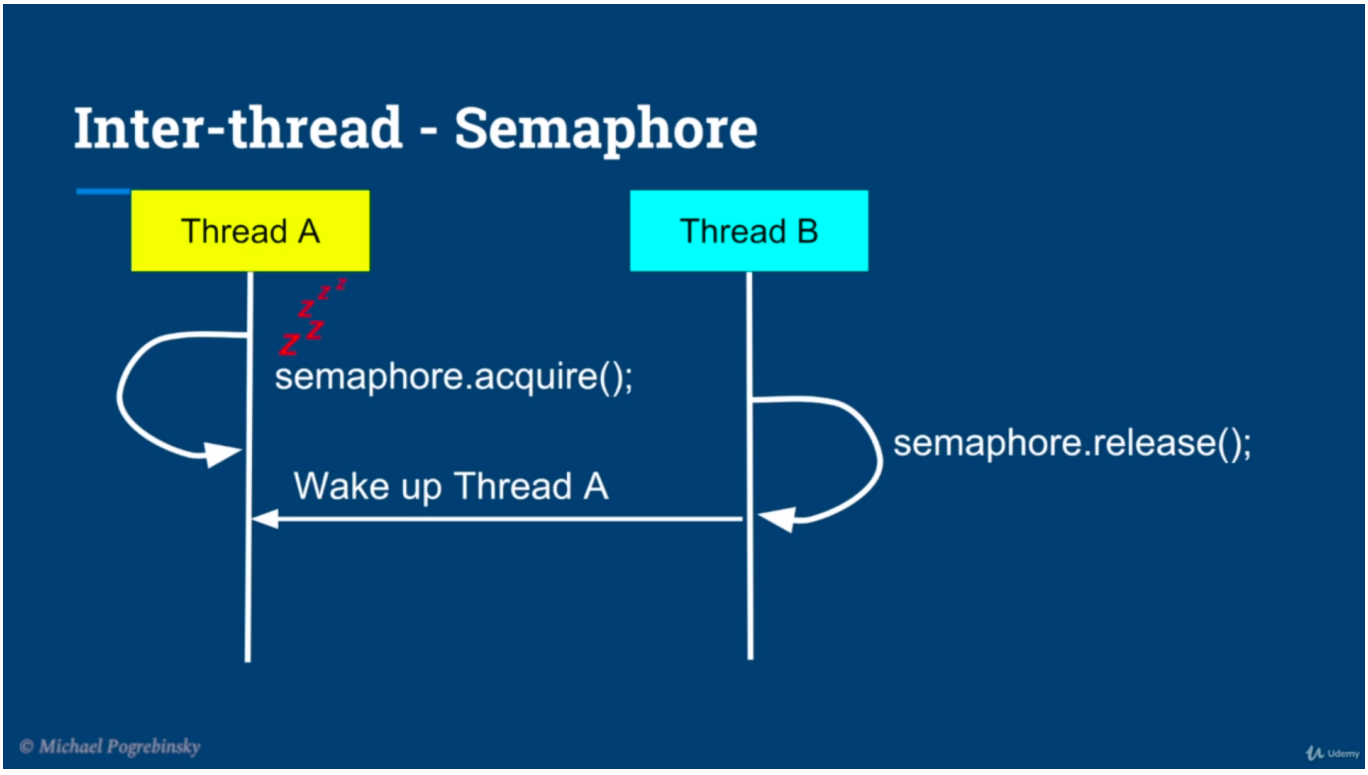
```
while(true) {
    full.acquire();
    consume(item);
    empty.release();
}
```

Consumer


Inter-thread - Thread.interrupt()
Thread A
Thread B
Thread.interrupt()
Clean up and terminate
© Michael Pogrebinsky


Inter-thread - Thread.join()
Thread A
Thread B
threadB.join();
Wake up Thread A
© Michael Pogrebinsky


Inter-thread - Semaphore
Thread A
Thread B
semaphore.acquire();
semaphore.release();
Wake up Thread A
© Michael Pogrebinsky


Inter-thread - Semaphore
Thread A
Thread B
semaphore.acquire();
semaphore.release();
Wake up Thread A
© Michael Pogrebinsky

Inter-thread - Creation

- Condition Variable is always associated with a lock
- The lock ensures atomic check and modification of the shared variables, involved in the condition

Inter-thread - Condition.await()

- ***void await()*** - unlock lock, wait until signalled
- ***long awaitNanos(long nanosTimeout)*** - wait no longer than nanosTimeout
- ***boolean await(long time, TimeUnit unit)*** - wait no longer than time, in given time units
- ***boolean awaitUntil(Date deadline)*** - wake up before the deadline date

Inter-thread - Condition.signal()

- ***void signal()*** **-** wakes up a single thread, waiting on the *condition variable*
- A thread that wakes up has to reacquire the lock associated with the *condition variable*
- If currently no thread is waiting on the *condition variable*, the signal method doesn't do anything

Object Signalling vs Condition Variables

Object Signalling	Condition Variable
synchronized(object) {	lock.lock()
}	lock.unlock()
object.wait()	condition.await()
object.notify()	condition.signal()
object.notifyAll()	condition.signalAll()