Multi-threaded programming

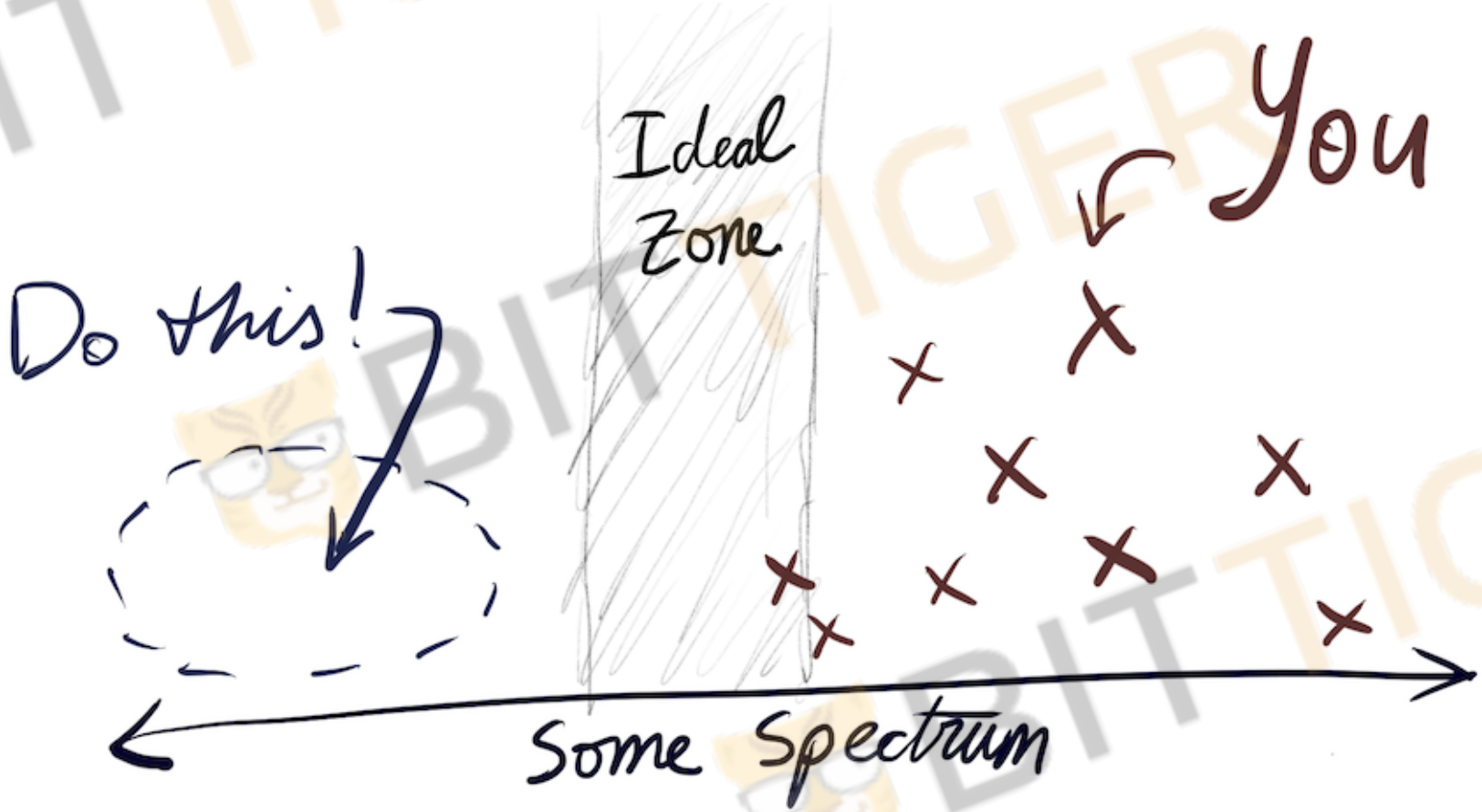
Chris@Confluent

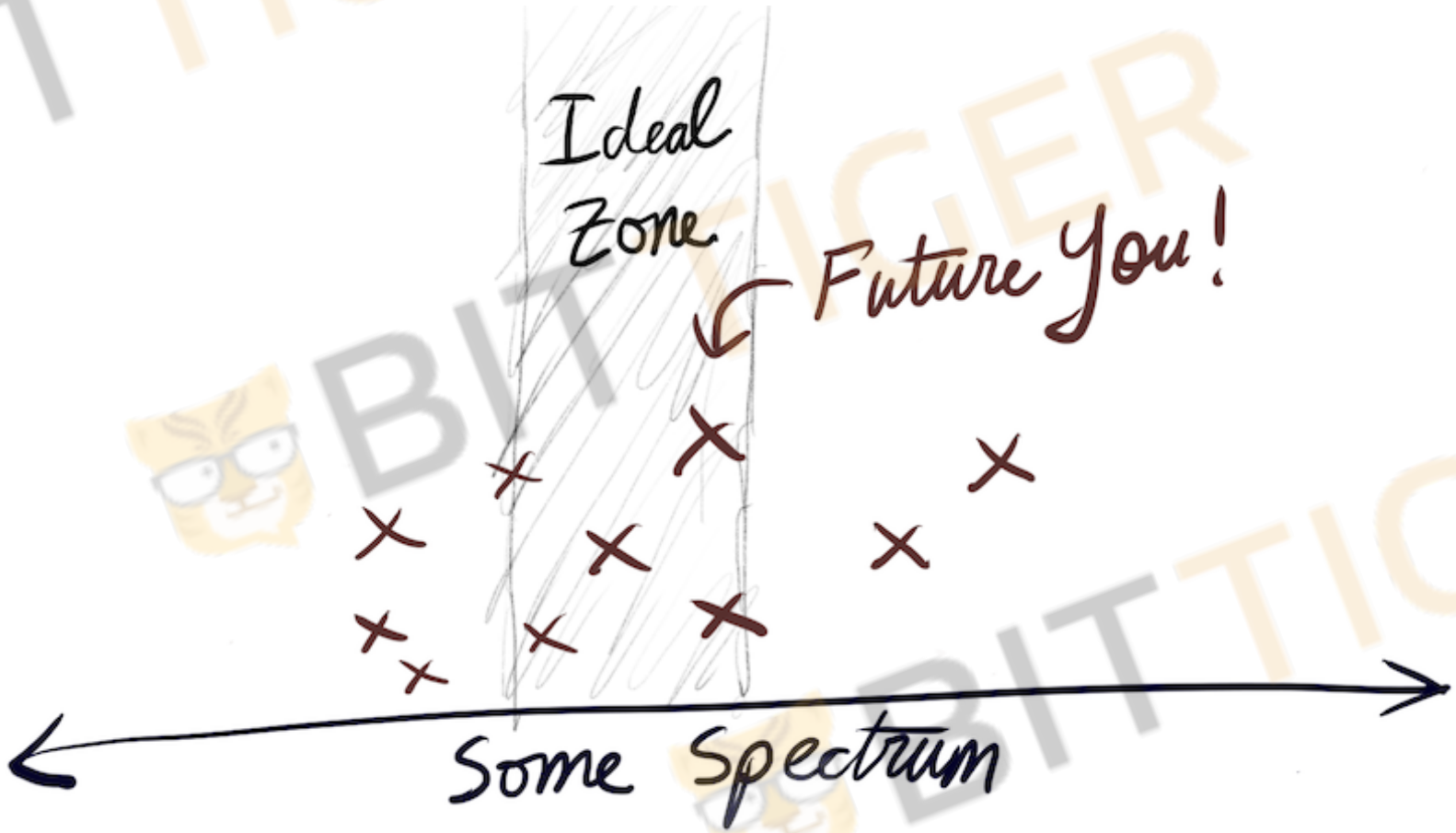
Today

- Motivation
- Multi-threaded programming use cases
- Computer architecture
- Operating systems basics
- Java threads
- Java memory model
- Thread synchronization

Next lecture

- Thread safe singleton
- Lock implementation
- Readers writers lock implementation
- Delayed scheduler implementation
- Multi-threaded programming in big data systems

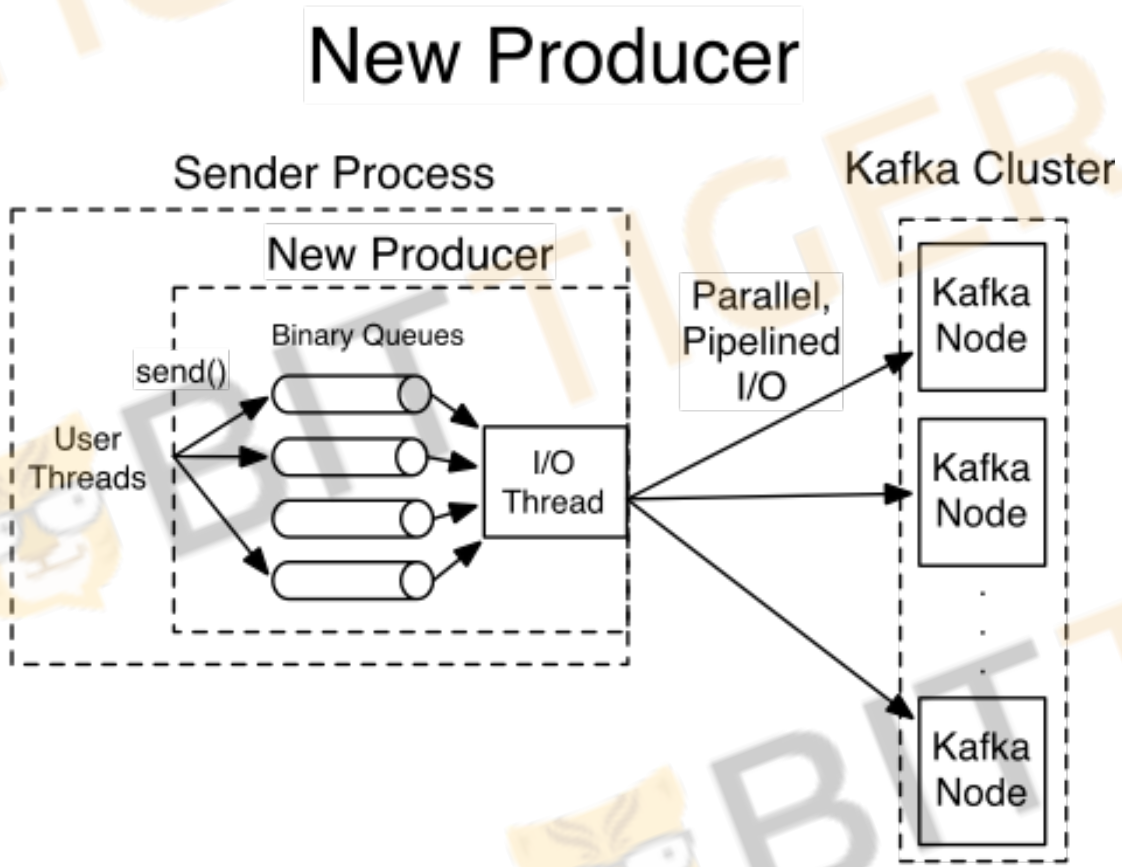

Ideal Zone
You
Do this!
Some Spectrum


Ideal
Zone
Future You!
Some Spectrum

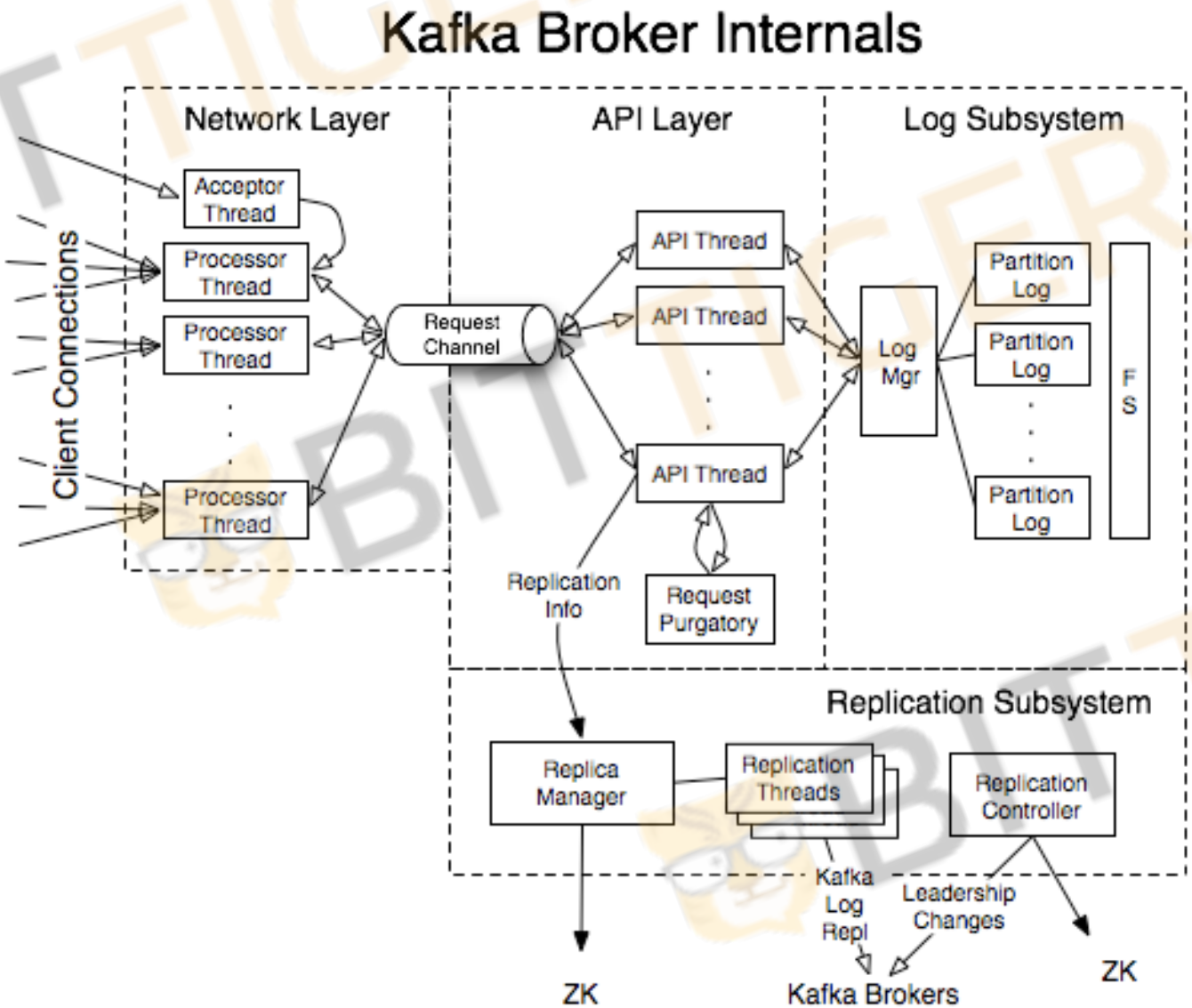
Motivation

- Multi-threaded programming is widely used in big data systems
  - Kafka
  - Spark
  - Twitter Heron
  - Schema Registry
- Multi-threaded programming is essential for Infrastructure engineers
  - Job duty
  - Interview

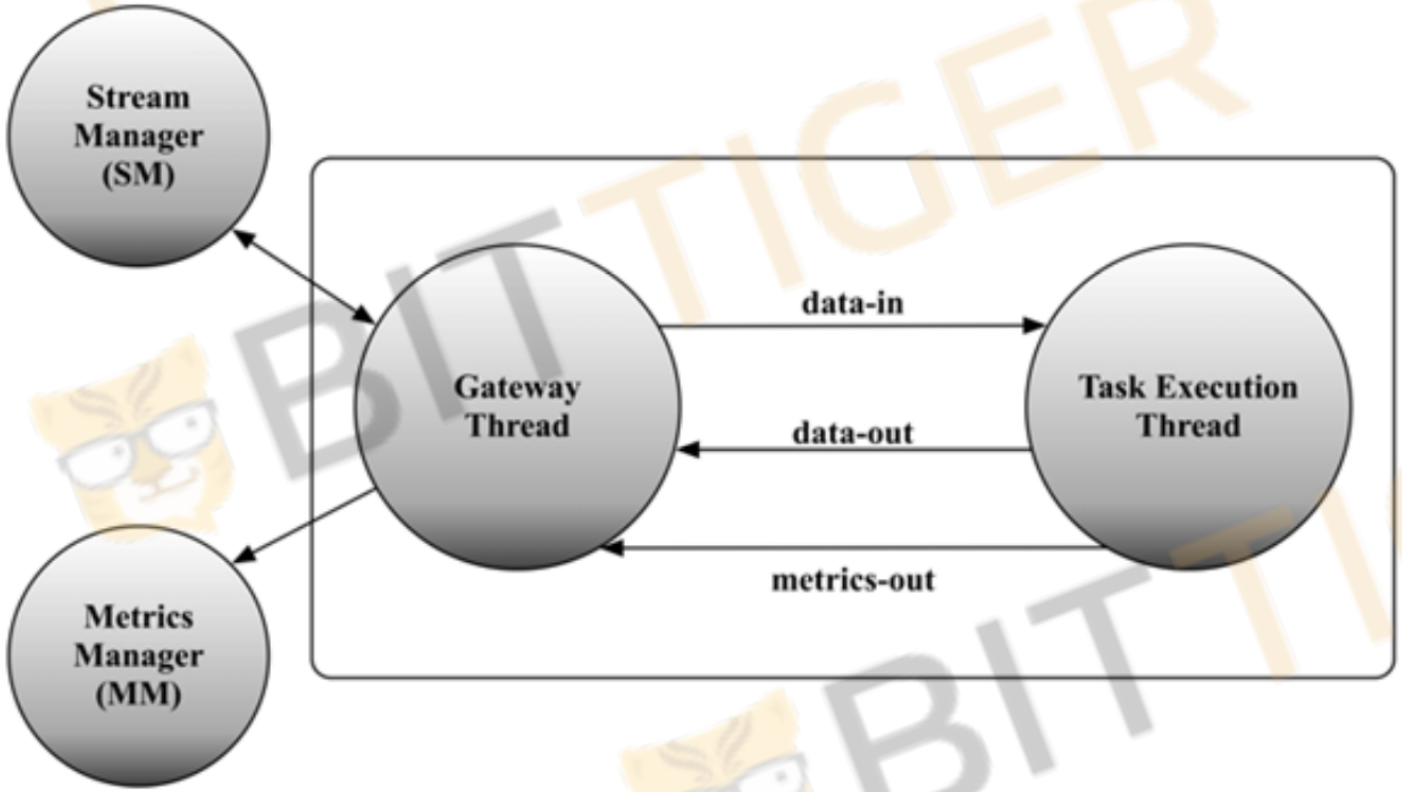
Kafka Producer



Kafka Broker



Twitter Heron



Example projects for Infrastructure engineers

- Schema registry
  - Thread safe schema registry client.
  - Thread safe serializers.
- Kafka connect Elasticsearch connector
  - Used multi-threading to boost performance.
- Python wrapper for Spark operators.
- Kafka stream join optimization.

Multi-threaded programming

Use cases

- Accessing slow I/O devices.
- Interacting with humans.
- Reducing latency by deferring work.
- Servicing multiple network clients.
- Computing in parallel on multi-core machines.

Computer architecture

Computer architecture summary

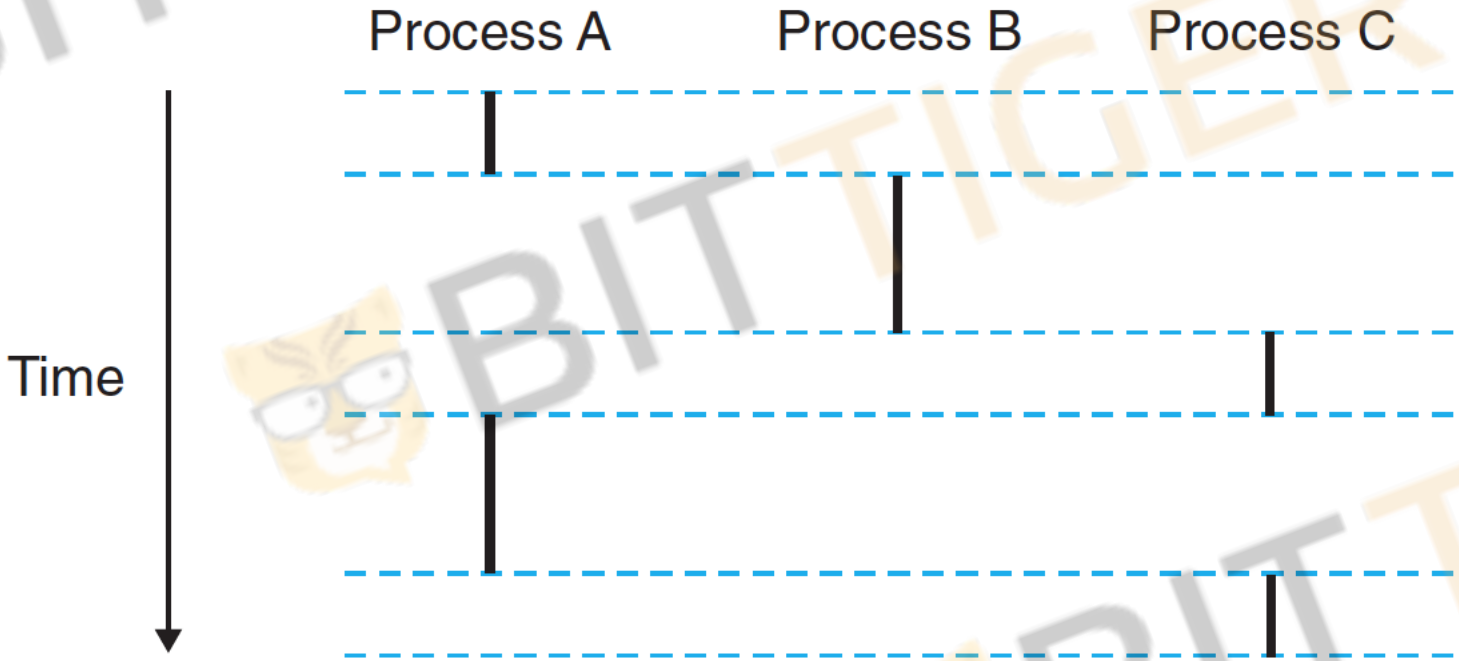
- Hardware architecture: CPU, memory and interconnect.
- Cache is used to bridge the latency gap between CPU and memory.
- The effective of cache is based on the locality principle.
- The unit of cache data eviction is cache line. A cache line is typically consists of multiple words.
- In multicore architecture, cache coherence is needed to ensure program correctness.
- False sharing degrades cache effectiveness.
- Out-of-order execution can alleviate pipeline hazards and reduce program execution time.

Operating systems basics

Process

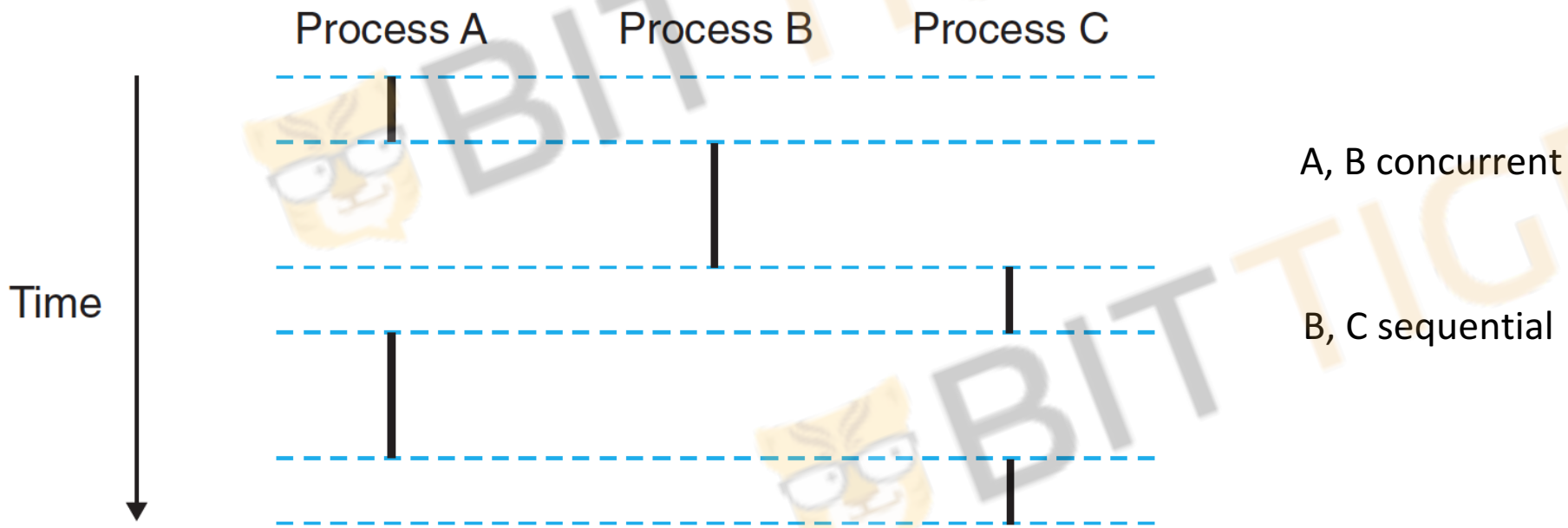
- A process is an instance of a running program.
- Process provides each program with two key abstractions
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these illusions maintained?
  - Process executions interleaved.
  - Address spaces managed by virtual memory system.

Logical control flows



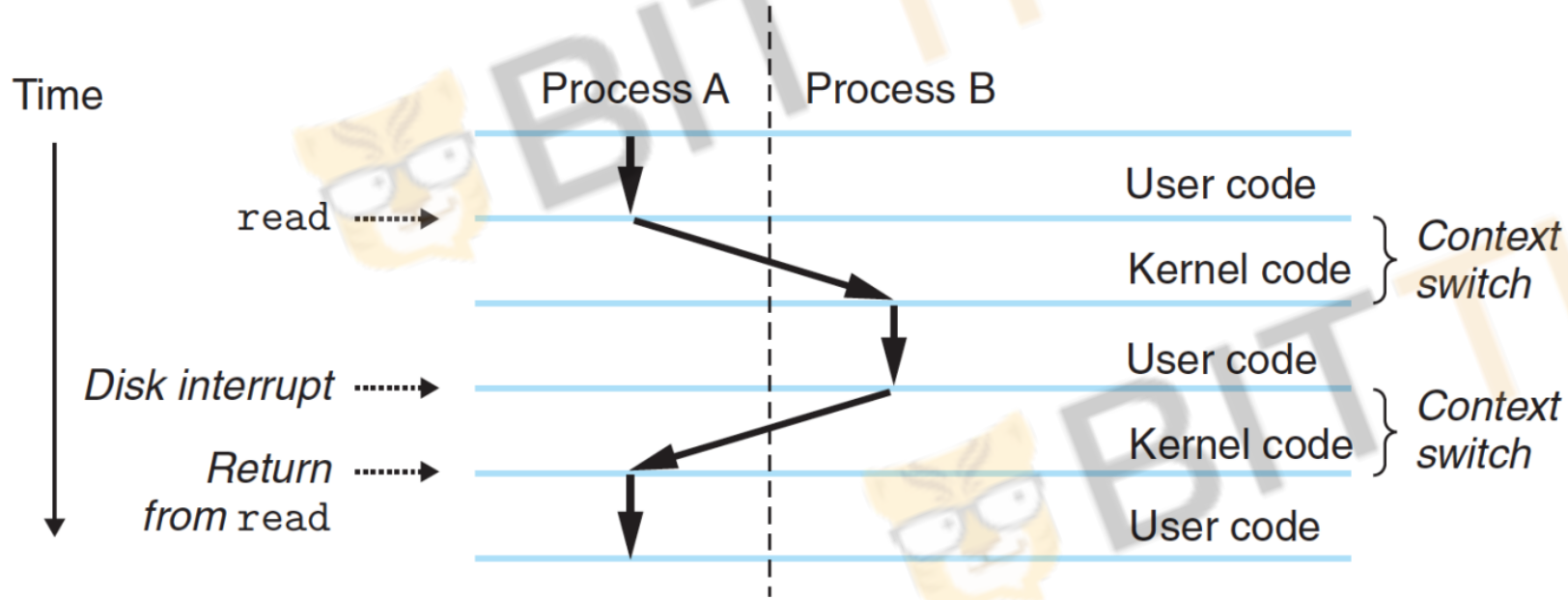
Concurrent processes

- Two processes are concurrent if their flows overlap in time.
- Otherwise, they are sequential.



Context switching

- Processes are managed by a shared chunk of OS code called the the kernel.
- Control flow passes from one process to another via context switch.



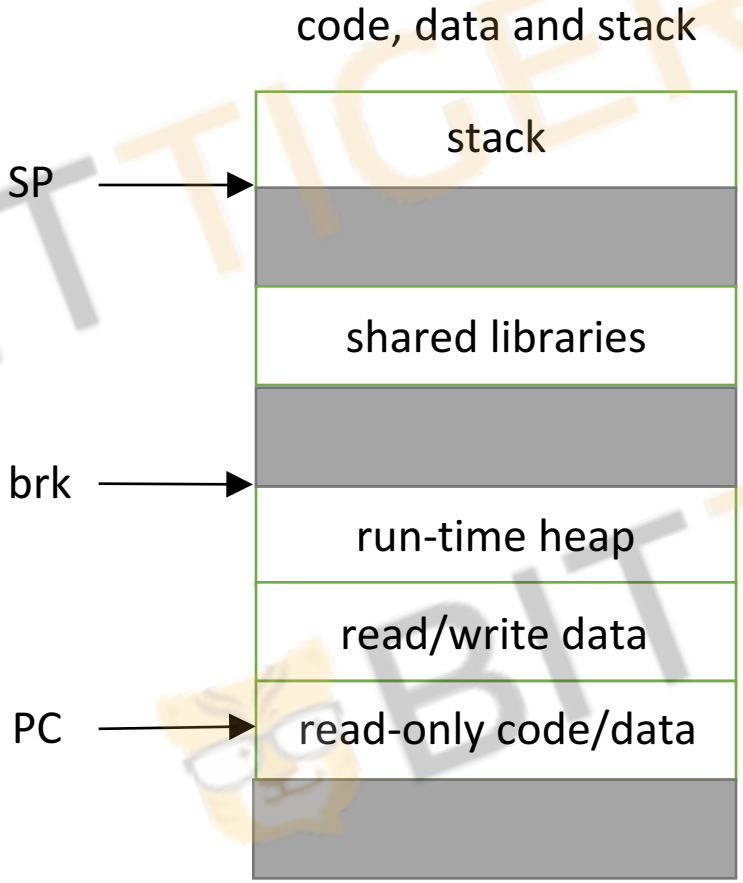
Tradition view of a process

Process = process context + code, data and stack

process context

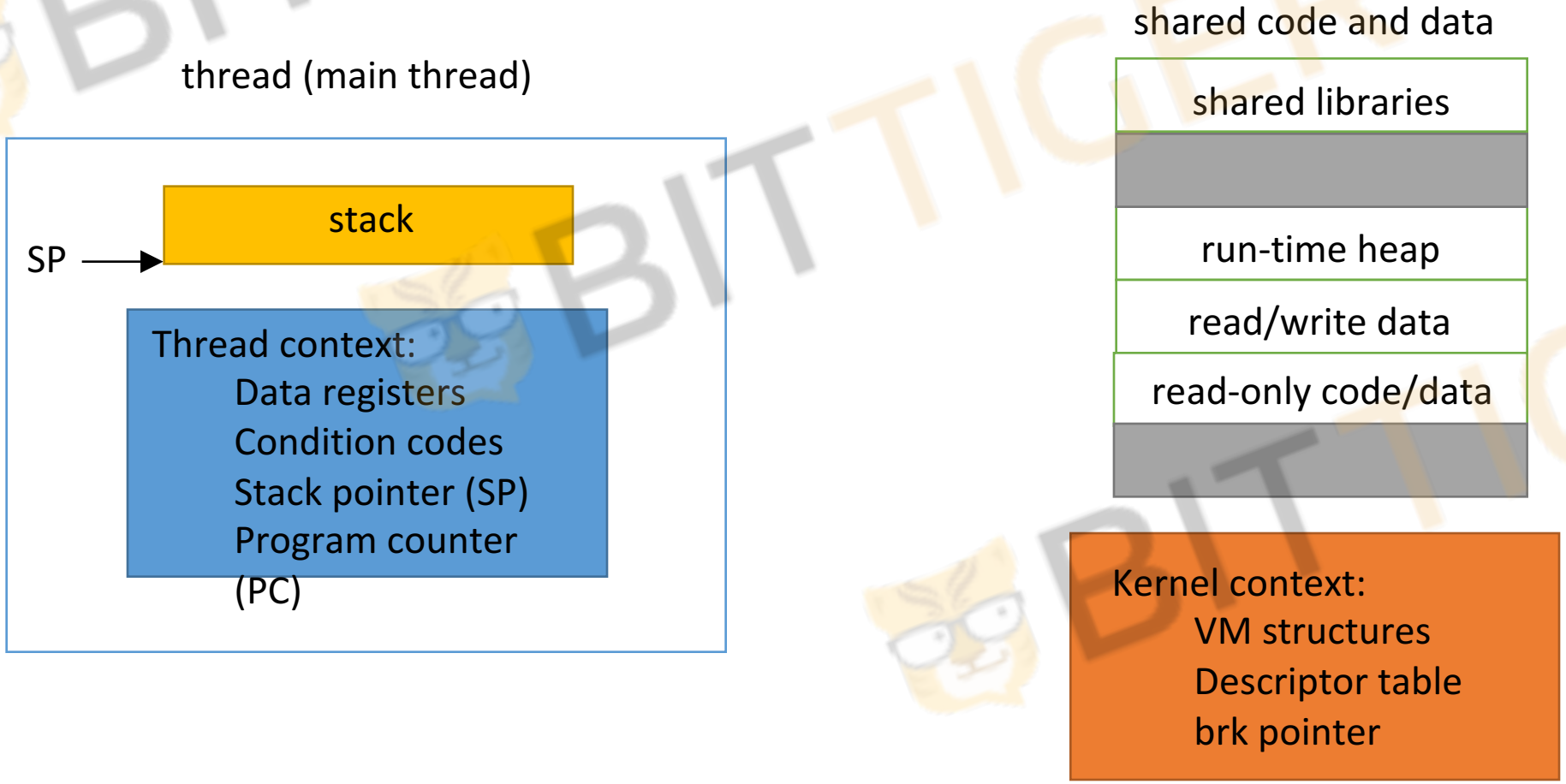
Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures
- Descriptor table
- brk pointer



Alternate view of a process

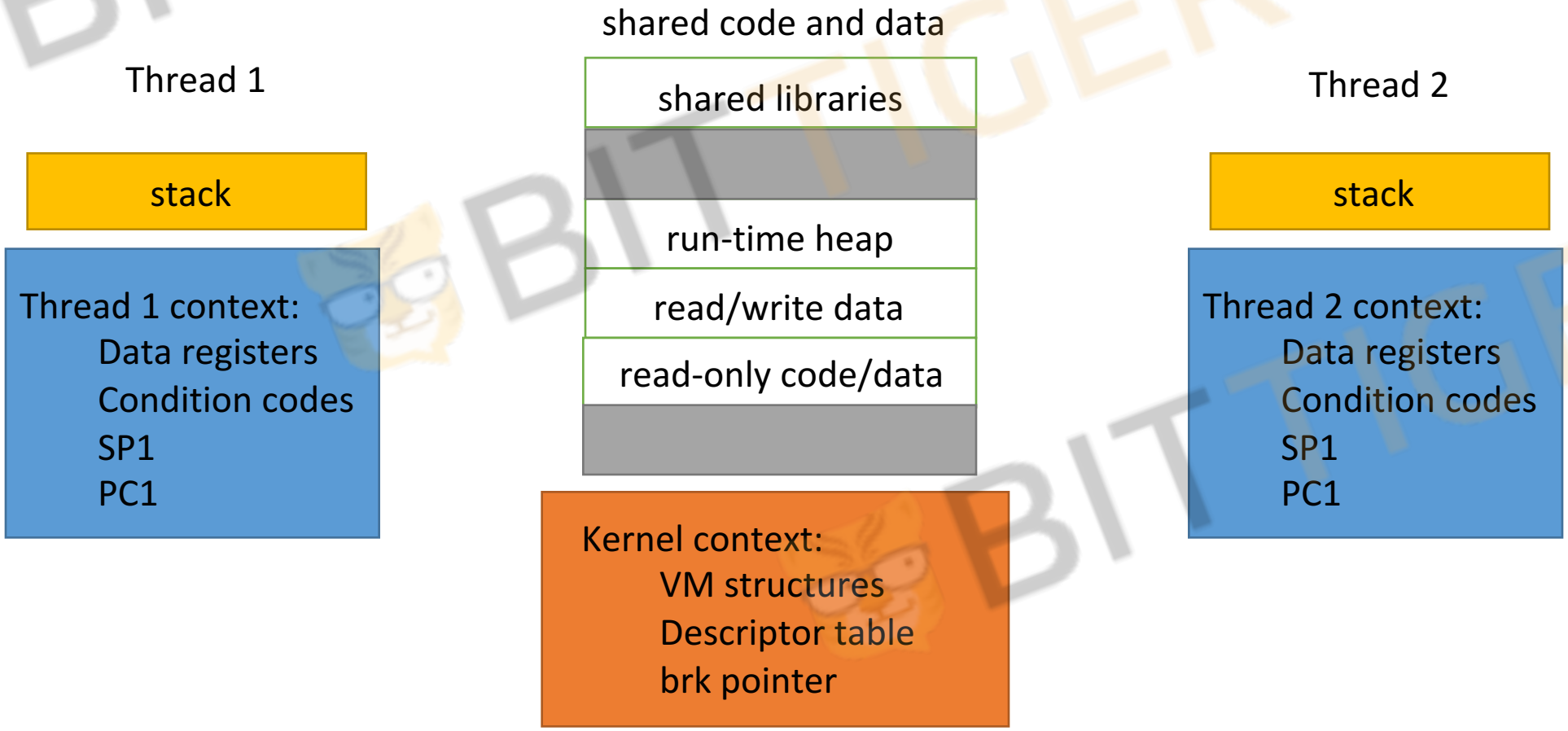
Process = thread + code, data and kernel context



A process with multiple threads

Multiple threads can be associated with a process

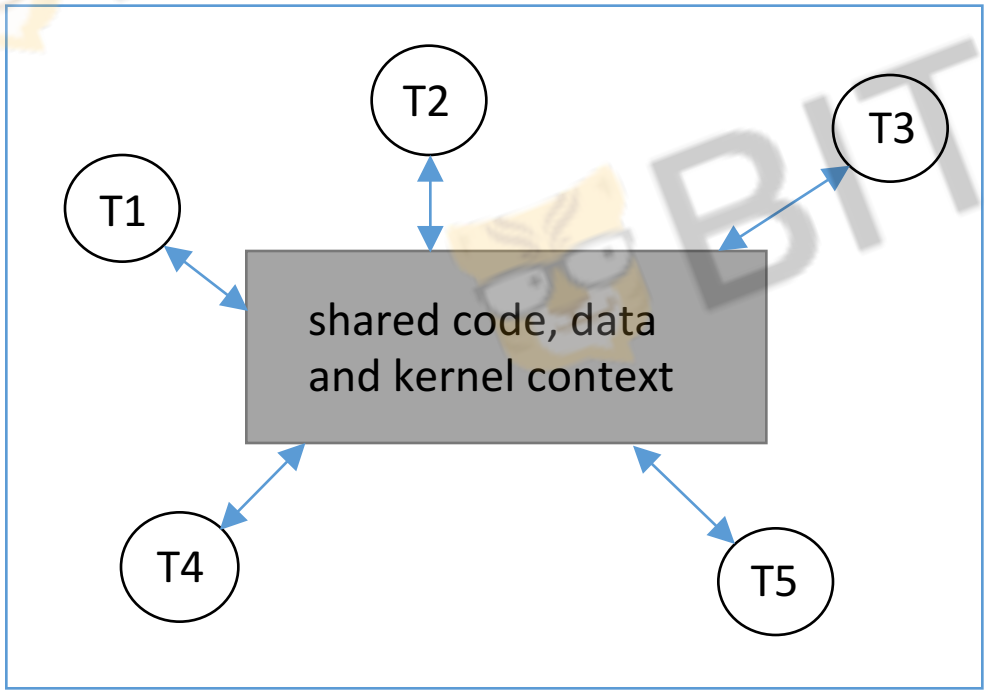
- Each thread has its own logical control flow (sequence of PC values)
- Each thread shares the same code, data and kernel context

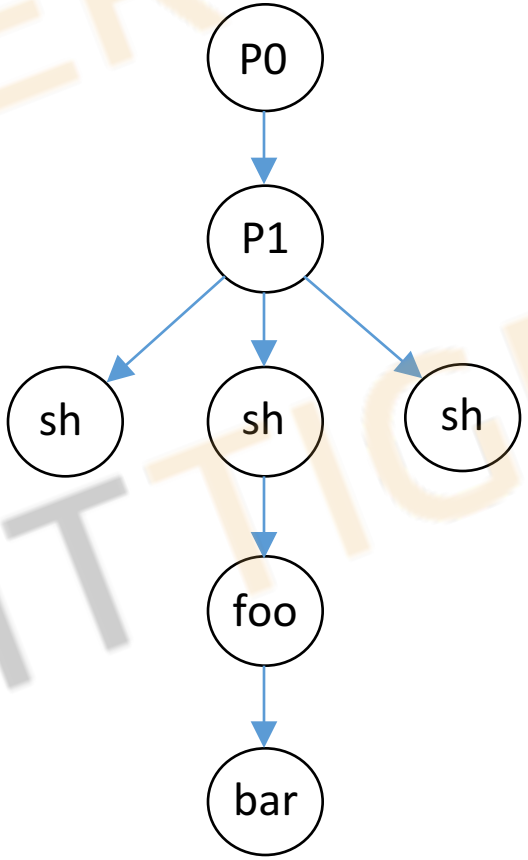


Logical view of threads

Threads associated with a process forms a pool of peers

Processes form a tree hierarchy



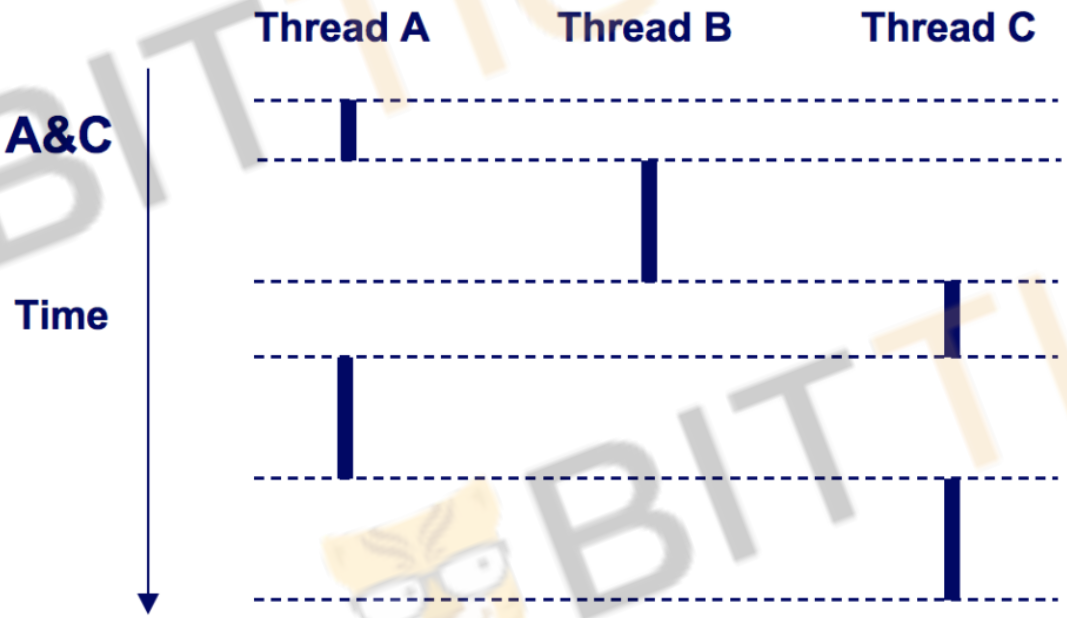


Concurrent thread execution

- Two threads are concurrent if their logical flows overlap in time.
- Otherwise, they are sequential.

Examples:

- **Concurrent: A & B, A&C**
- **Sequential: B & C**



Threads vs Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently
  - Each is context switched
- How threads and processes are different
  - Threads share code and data, processes do not
  - Threads are somewhat less expensive than processes

Operating system basics summary

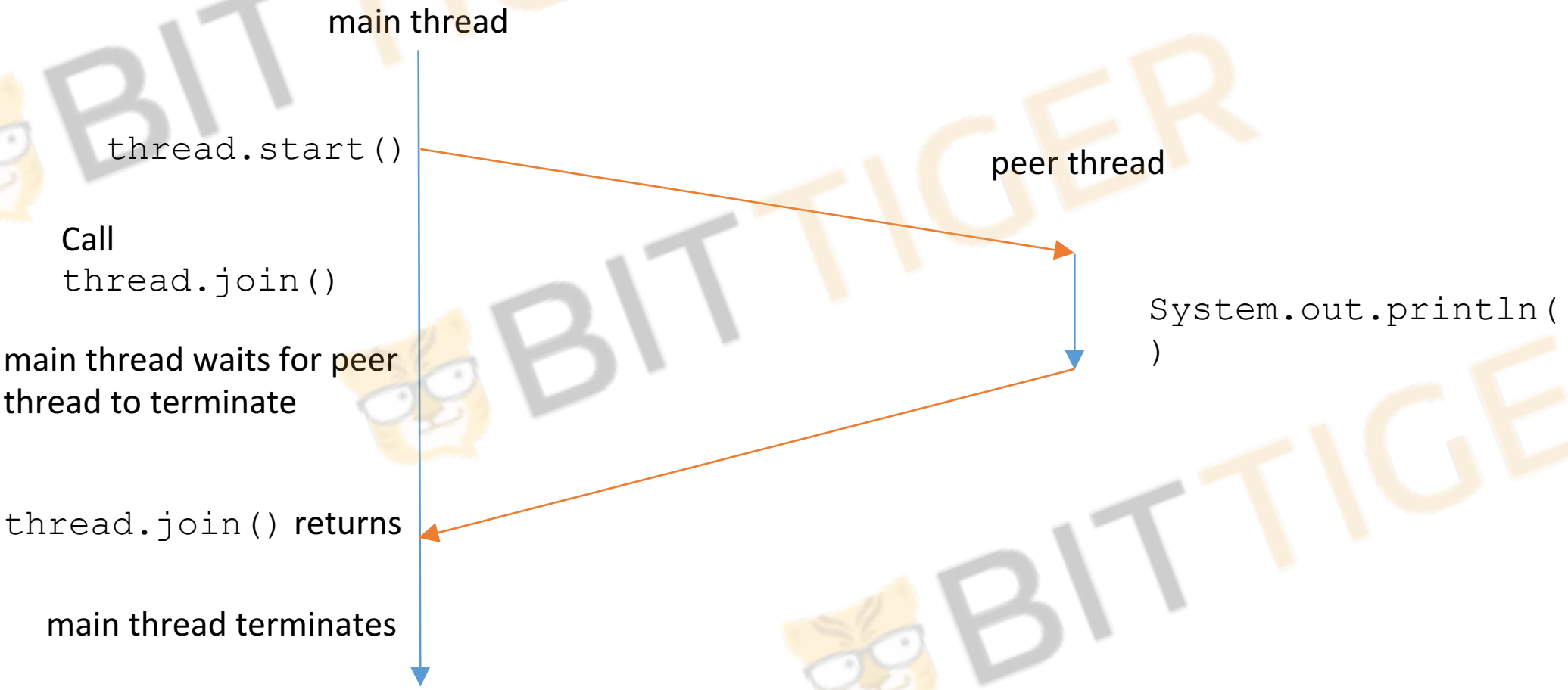
- Processes provides two abstractions: logical control flow and private and private address space.
- Two processes are concurrent if they overlap in time.
- A process can have multiple threads.
- Threads share code, heap and kernel context. Each thread has its own stack and context.
- Threads in a process are peers where processes are hierarchical.
- A thread is blocked when performing IO operations.

Java threads

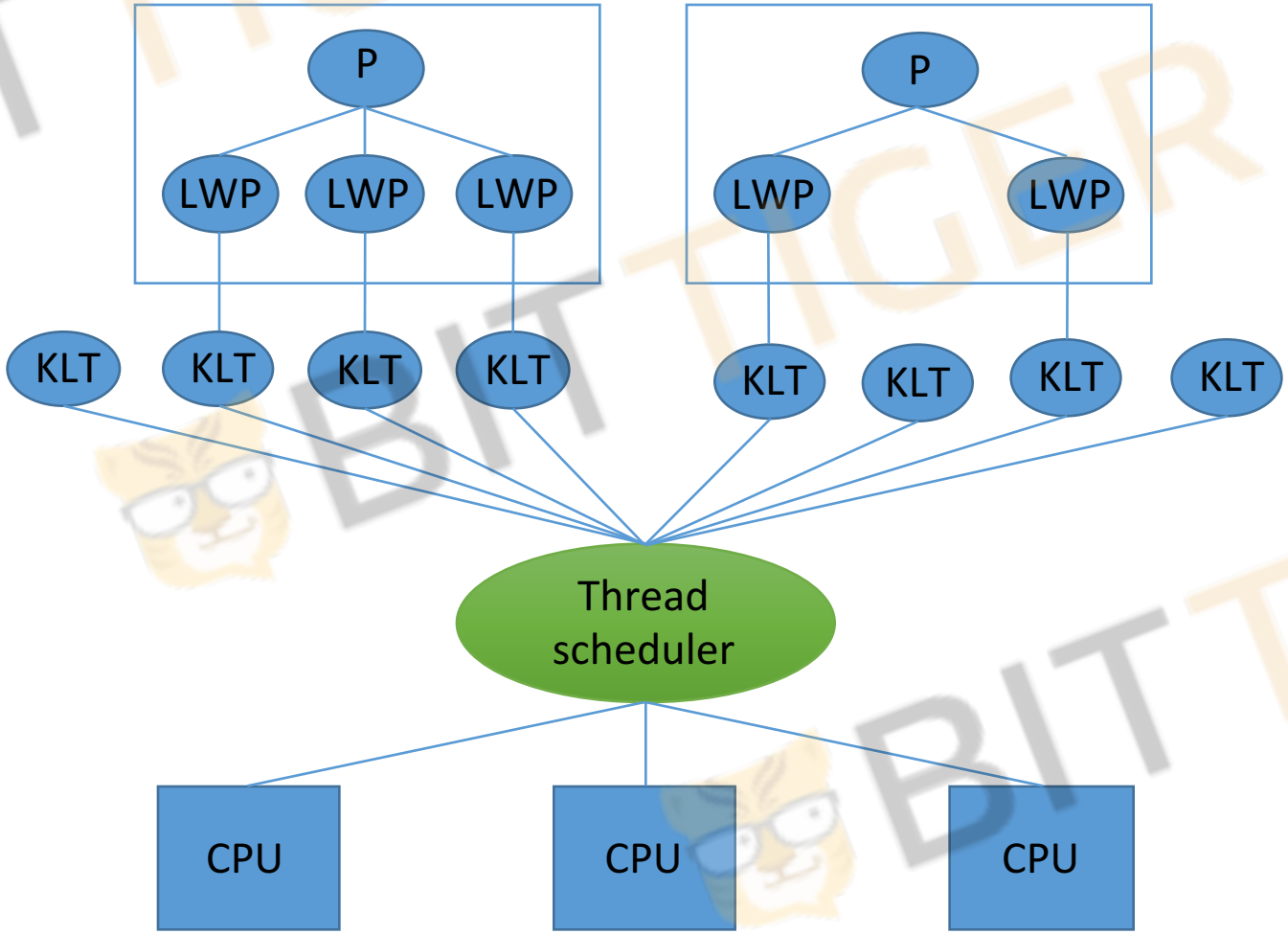
Java Threads

- Java Threads
  - Thread class
  - Runnable interface
- Thread creation
  - Override Thread class
  - Implement Runnable interface
- Hello World

Execution of threaded “Hello world”



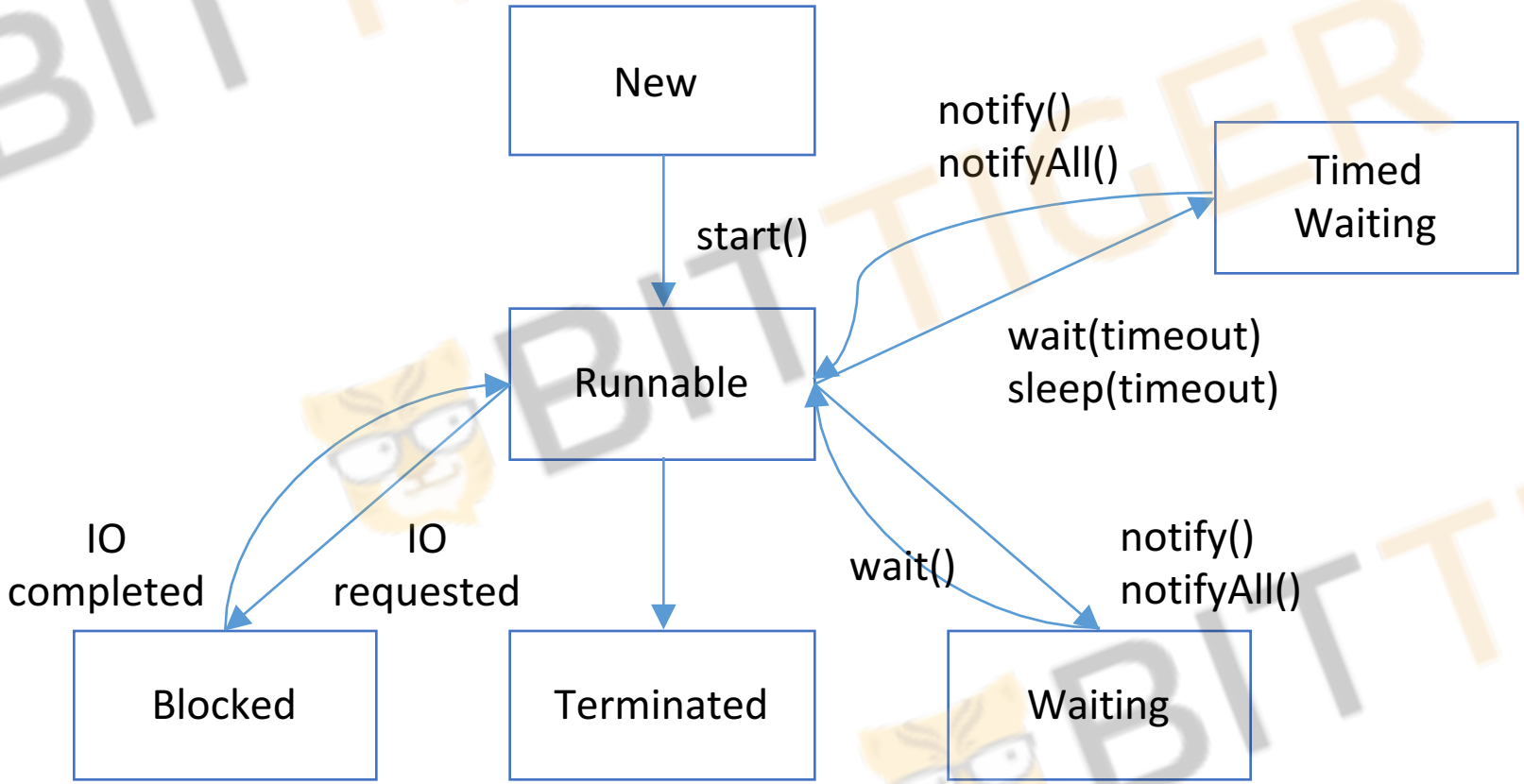
Threads implementation



Java thread states

- Java threads have 6 states, at any point of time, each thread can only be in one of the states
  - New
  - Runnable
  - Waiting
  - Timed waiting
  - Blocked
  - Terminated

Java thread lifecycle



Java thread lifecycle

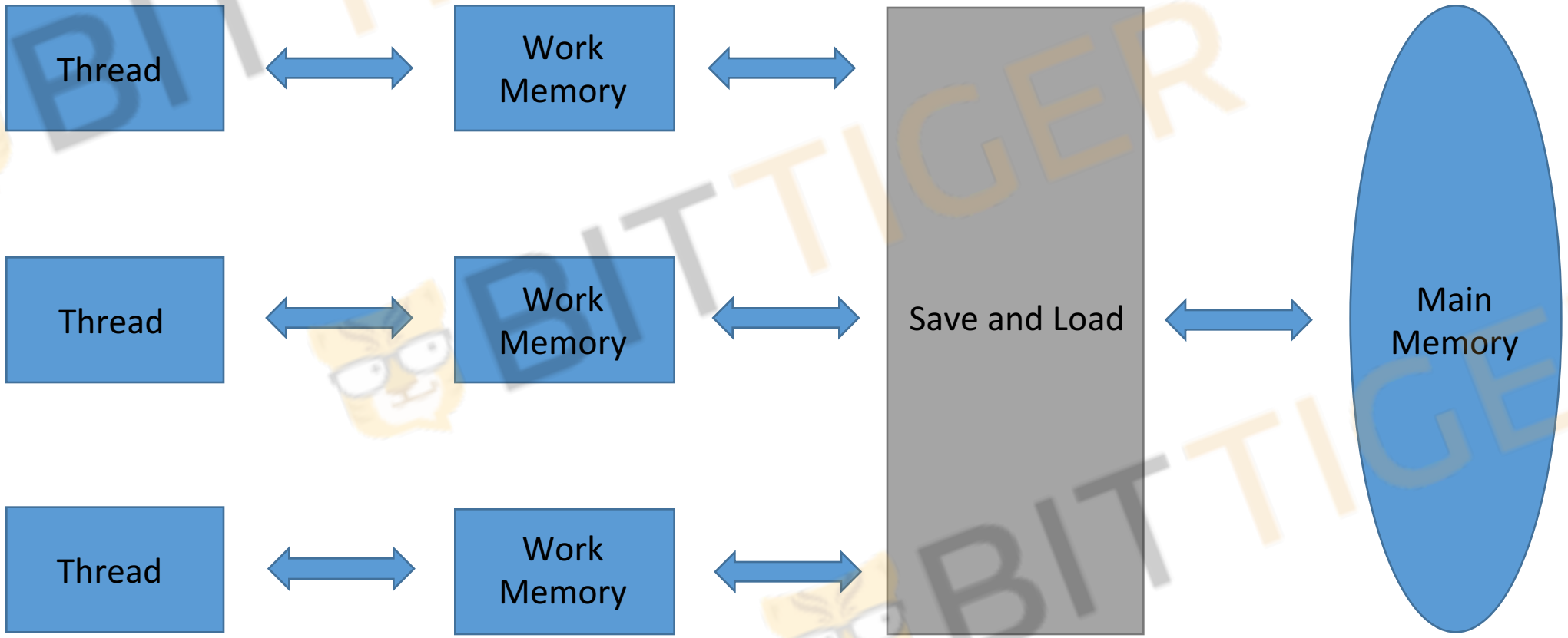
```
Thread t = new Thread(); // New
t.start(); // Runnable
//Become running if scheduled by OS
Thread.sleep(1000); // Timed waiting
wait(); //Waiting
System.out.println(“Hello world”); //Blocked
```

Java threads summary

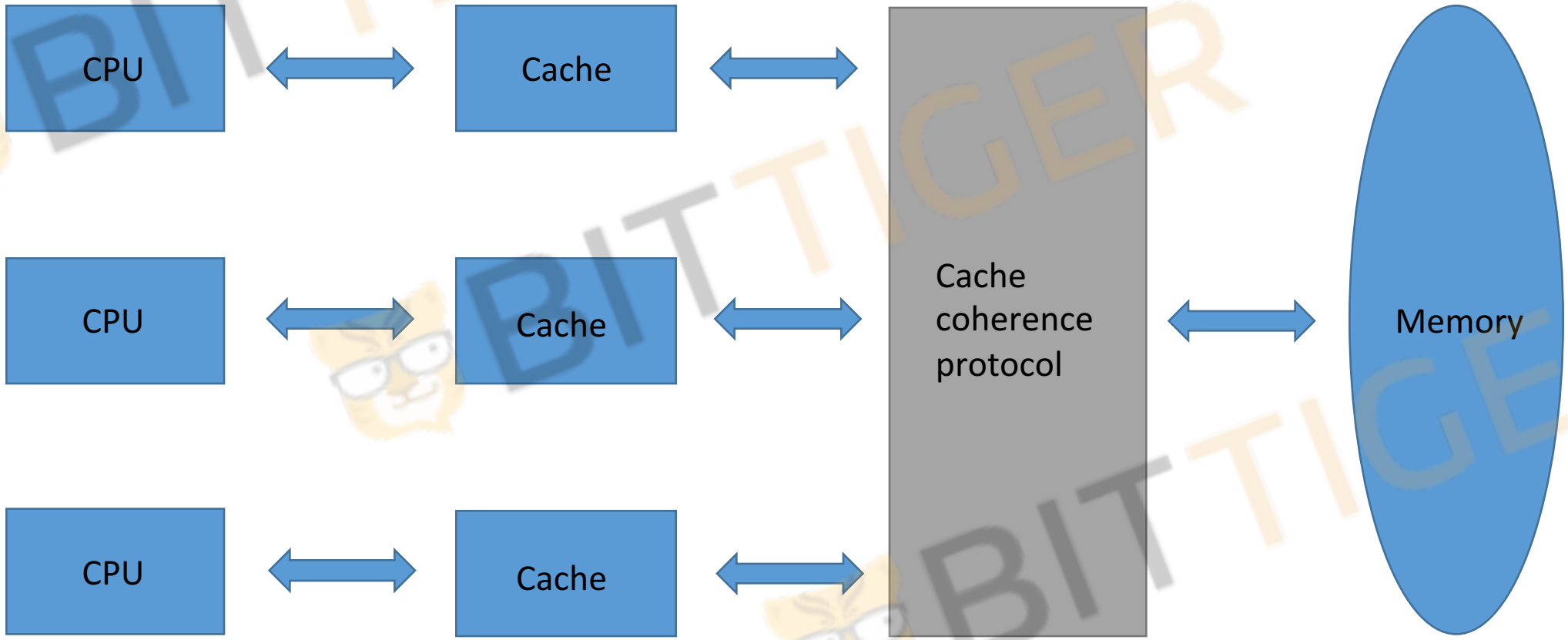
- Thread class and Runnable interface.
- Can use anonymous class to simplify thread creation.
- Java thread states: New, Runnable, Blocked, Waiting, Timed waiting, Terminated.
- A thread is in blocked state when performing IO. When blocked, the thread gives the CPU to other threads.
- In Linux, Java threads are mapped to kernel level threads (KLT) and scheduled by OS.

Java memory model

Java memory model



Cache coherence



Java memory model summary

- All threads share the main memory.
- Each thread uses a local working memory.
- Refreshing local memory to/from main memory must comply to Java memory model rules.

Thread synchronization

Synchronization

- Safety
  - Nothing bad happens ever
- Liveness
  - Something good happens eventually

Mutex

- Synchronized method

```
public class A {
    public synchronized void foo() {…}
    public synchronized void bar() {…}
    public void foobar() {…}
}
A a1 = new a1();
T1: a1.foo();
T2: a1.bar(); // T2 is blocked
T3: a1.foobar(); T3 is not blocked
```

Mutex

- Synchronized code block
  - `synchronized`(this) {..};
  - Object lock = new Object(); `synchronized`(lock) {…};
  - class A {..}; `synchronized`(A.class) {…};
- Only one thread can access the synchronized block, other threads will be blocked.

Mutex

- Synchronized method/block is reentrant to the same thread.
- Other threads are blocked.
- Java threads maps to kernel threads, so the kernel is in charge of thread state transition.
- user mode -> kernel mode, so synchronized is heavy weight lock.
- Changes in a synchronized method/block are made visible to other threads when exiting the synchronized block.

Mutex

- ```
  class A {
      public void foo();
    }
  ```
- `A a1 = new A();`
- `A a2 = new A();`
- `a1.foo(); // T1`
- `a2.foo(); // T2 blocked`
- We want to make sure that that only one thread can access foo() all all instances of A.
  ```

Reentrant lock

- `synchronized` is equivalent to reentrant lock

```
Lock lock = new ReentrantLock();
try {
   lock.lock();
   do something
} finally {
   lock.unlock();
}
```

Condition variable

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition).
- some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue.
- `wait()`
- `wait(long timeout)`
- `notify()`
- `notifyAll()`

Condition variable

- Conditional variable should be used inside a synchronized block.
- In Java, each object is associated with one and only one condition variable.
- This is a restriction, you may need to use explicit locking to get around this.

Condition variable

- Multiple condition variables can be associated with a reentrant lock.
- This is useful for implementing blocking queue.
- More about this in the next lecture.

Volatile variables

- Volatile semantics
  - Changes to volatile variables are visible to other threads immediately.
  - Volatile eliminate instruction reordering.
- Volatile use cases
  - Single writer.
  - The result doesn’t depend on the current value.

Atomic variables

- For an atomic operation, all threads either see the value of before assignment, or the value after assignment.
- Compare and swap (CAS)

Atomic variables

- ```
  public final int incrementAndGet() {
      for(;;) {
          int current = get();
          int next = current + 1;
          if (compareAndSet(current, next))
              return next;
      }
  }
  ```
- ABA problem
  - A variable is value A when when first read, and stays at A when assigning.
  - A -> B -> A
  ```

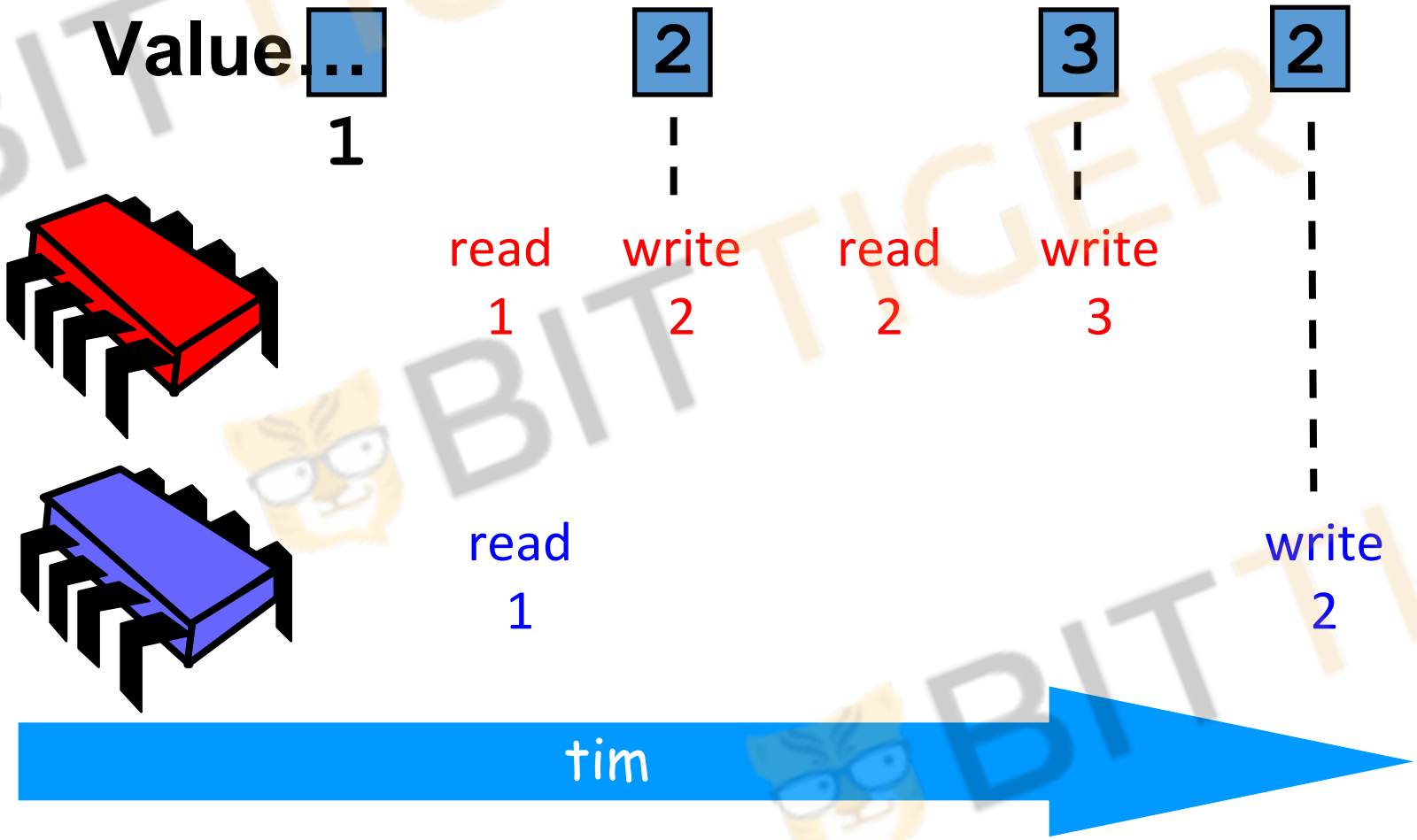
Example: Counter

- Implement a `Counter` class:
  - `void increment();`
  - `int getCount();`
- Make the `Counter` thread safe.

Example: Counter

- Synchronized on both methods.
- Synchronized on the code block.
- Reentrant lock.
- Atomic operations.
- Readers writers lock.

Example: Counter race condition



Thread safety

- A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Reasoning thread safety

- Identify the shared state.
- Find where the shared state is read or write.
- Think in the context that multiple threads are reading or writing the shared state.
- Think when the thread local copy is written back to main memory.
- Find an execution sequence that violates safety.

Thread synchronization summary

- Safety and liveness
- Synchronized use cases
  - Method
  - Code block
  - Static variable
- Synchronized is reentrant
- Reentrant lock
  - Release lock in the finally block
- Volatile variables
  - Changes to a variable are visible to other threads immediately.
- Atomic variables
  - all threads either see the value of before assignment, or the value after assignment.

Thread synchronization summary

- A condition variable is a queue to put threads on when some condition is not satisfied.
- Condition variables should be used in synchronized block/method.
- In Java, each object is associated with one and only one condition variable.
- Reentrant lock can have multiple condition variables. Useful for implementing blocking queue.

Summary

- Threads vs Processes
- Java thread lifecycle
- Java memory model
- Thread synchronization
  - Safety and liveness
  - Mutex
  - Volatile
  - Atomic variables
  - Condition variable
- Reasoning thread safety

Next lecture

- Thread safe singleton
- Lock implementation
- Readers writers lock implementation
- Delayed scheduler implementation
- Multi-threaded programming in big data systems