SCC.211 Operating Systems

Locks & Race Conditions

Dr Amit Chopra
School of Computing & Communications, Lancaster University, UK

Objectives

- Race condition
- Atomic and Critical Section
- Locks
- Mutual exclusion in Java

Synchronization in Shared Memory

Mechanism to ensure multiple concurrent processes or threads **do not simultaneously access** shared resources, e.g., a bank balance.

In a program, the bank account would be represented by a shared variable for the bank balance. This means that threads who want to access it must do so in a coordinated manner. They must ``synchronize'' on the variable.

We will study two widely-used synchronization mechanisms: locks and semaphores

Shared Memory Synchronization

Coordinate how threads access the critical section

Critical Section: Region of code that accesses variables that are shared between threads.

Shared Resource can be anything of interest: bank balance, data structure, device, network connection. Always represented in memory, so we can talk equivalently in terms of **Shared Variables**.

Example: Banking

Banking system operates an account balance

Two threads called update on the same bank account. The code inside update is the critical region, the shared variable being bal.

```
public class Bank_account {
   private int bal = 0;

   public void Bank_account(int start_balance) {
          bal = start_balance;
   }
   public void update(int amount) {
          bal = bal + amount;
   }
}
...
Bank_account b = new Bank_account(0);
```

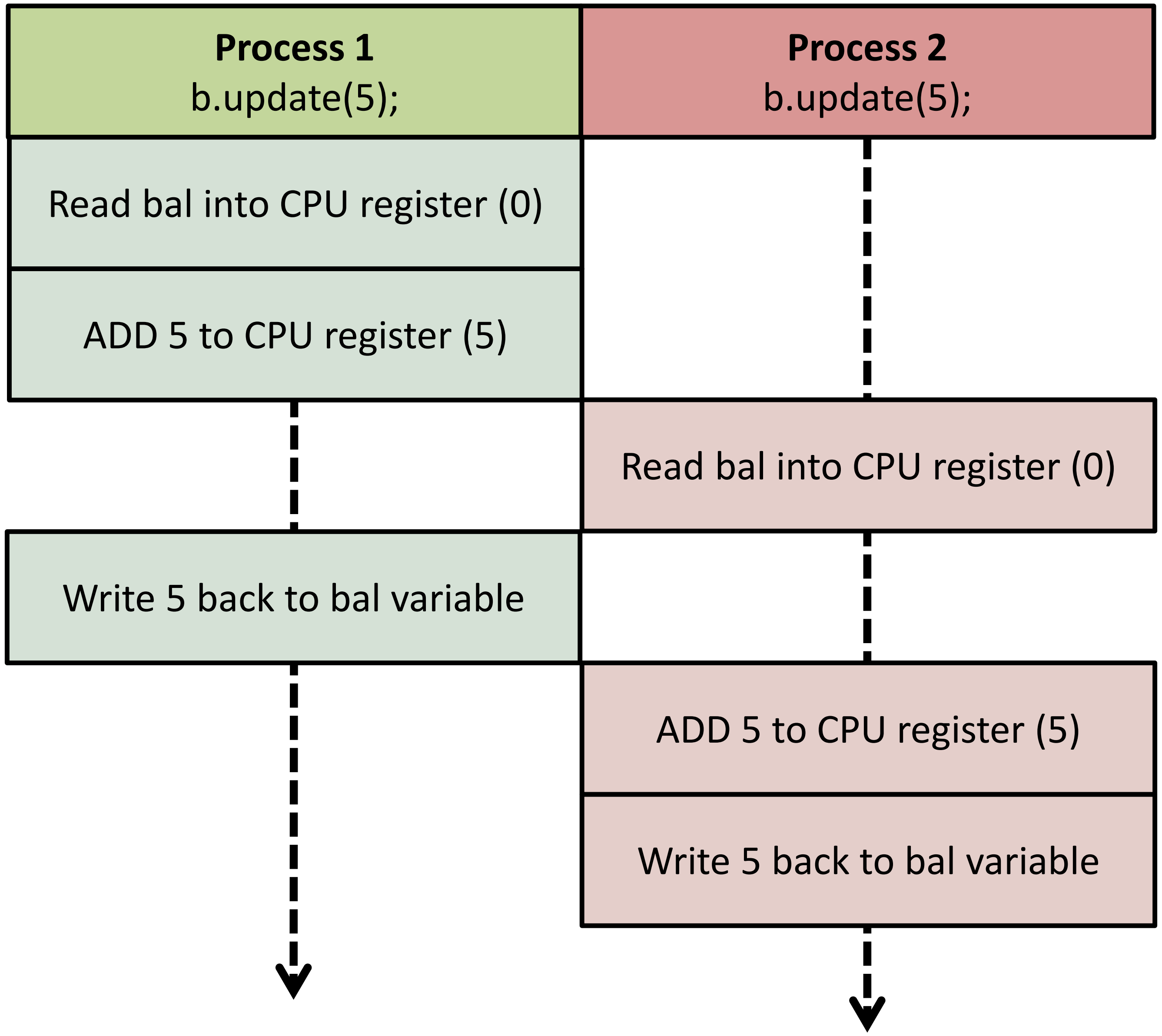
Process 1
b.update(5);

Process 2
b.update(5);

Problem: Banking

Final result **may be** that the balance is 5 instead of 10

This is a **lost update problem.**



Process 1 b.update(5);	**Process 2** b.update(5);
Read bal into CPU register (0)	
ADD 5 to CPU register (5)	
	Read bal into CPU register (0)
Write 5 back to bal variable	
	ADD 5 to CPU register (5)
	Write 5 back to bal variable

```
public class Bank_account {
  private int bal = 0;

  public void Bank_account(int start_balance) {
       bal = start_balance;
  }
  public void update(int amount) {
       bal = bal + amount;
  }
}
…
Bank_account b = new Bank_account(0);
```

Problem Source

update of bank account is not **atomic,** meaning that when one thread is in the middle of updating a bank account, another thread may start updating the same account

```
public class Bank_account {
  private int bal = 0;

  public void Bank_account(int start_balance) {
      bal = start_balance;
  }
  public void update(int amount) {
      bal = bal + amount;
  }
}
...
Bank_account b = new Bank_account(0);
```

Variable update requires

Read
Manipulate
Write

Problem Nature: Race Condition

The condition where an incorrect program output may be generated depending on the relative order in which instructions from multiple threads are interleaved (executed).

Process 1 b.update(5);	**Process 2** b.update(5);
Read balinto CPU register (0)	
ADD 5 to CPU register (5)	
	Read bal into CPU register (0)
Write 5 back to bal variable	
	ADD 5 to CPU register (5)
	Write 5 back to bal variable

Incorrect

£5

Process 1 b.update(5);	**Process 2** b.update(5);
Read balinto CPU register (0)	
ADD 5 to CPU register (5)	
Write 5 back to bal variable	
	Read bal into CPU register (5)
	ADD 5 to CPU register (10)
	Write 10 back to bal variable

Correct

£10

Threads Introduce Nondeterminism

Sequential programs are *deterministic*:
for the same input, the same output

Threads make a program *nondeterministic*:
For the same input, output depends on the *interleaving* of instructions from different threads. Race condition if output can be incorrect.

Aim of synchronization: Eliminate race conditions

Correct output? One influential idea is that any output from interleaving multiple threads is the same as an output from executing the threads sequentially.

Atomicity of critical section means that only one thread may be in it and the thread must finish the section before another thread may enter, thus ensuring "sequentiality."

Locks: Solving Banking Problem via Mutual Exclusion

Lancaster University

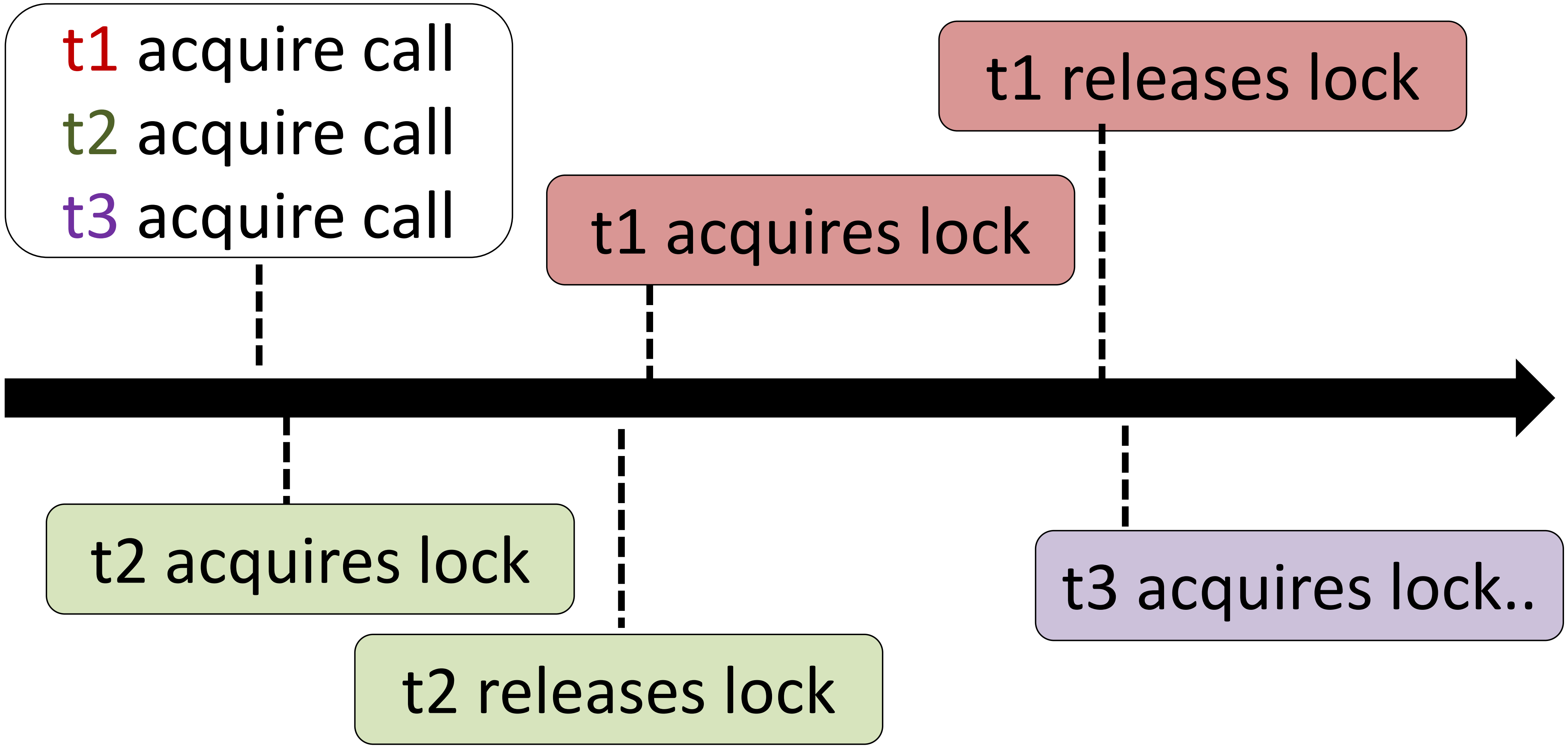
Mutual exclusion for a resource: limits resource access to one thread at a time. Guarantees atomicity.

Lock can be in one of two states

- Held: A thread is in the critical section (no other thread can enter)
- Not held: No thread in the critical section (any thread can enter)

Two operations to move lock between states

- Acquire: mark lock as held, or wait until release
- Release: mark as not held



t1 acquire call
t2 acquire call
t3 acquire call

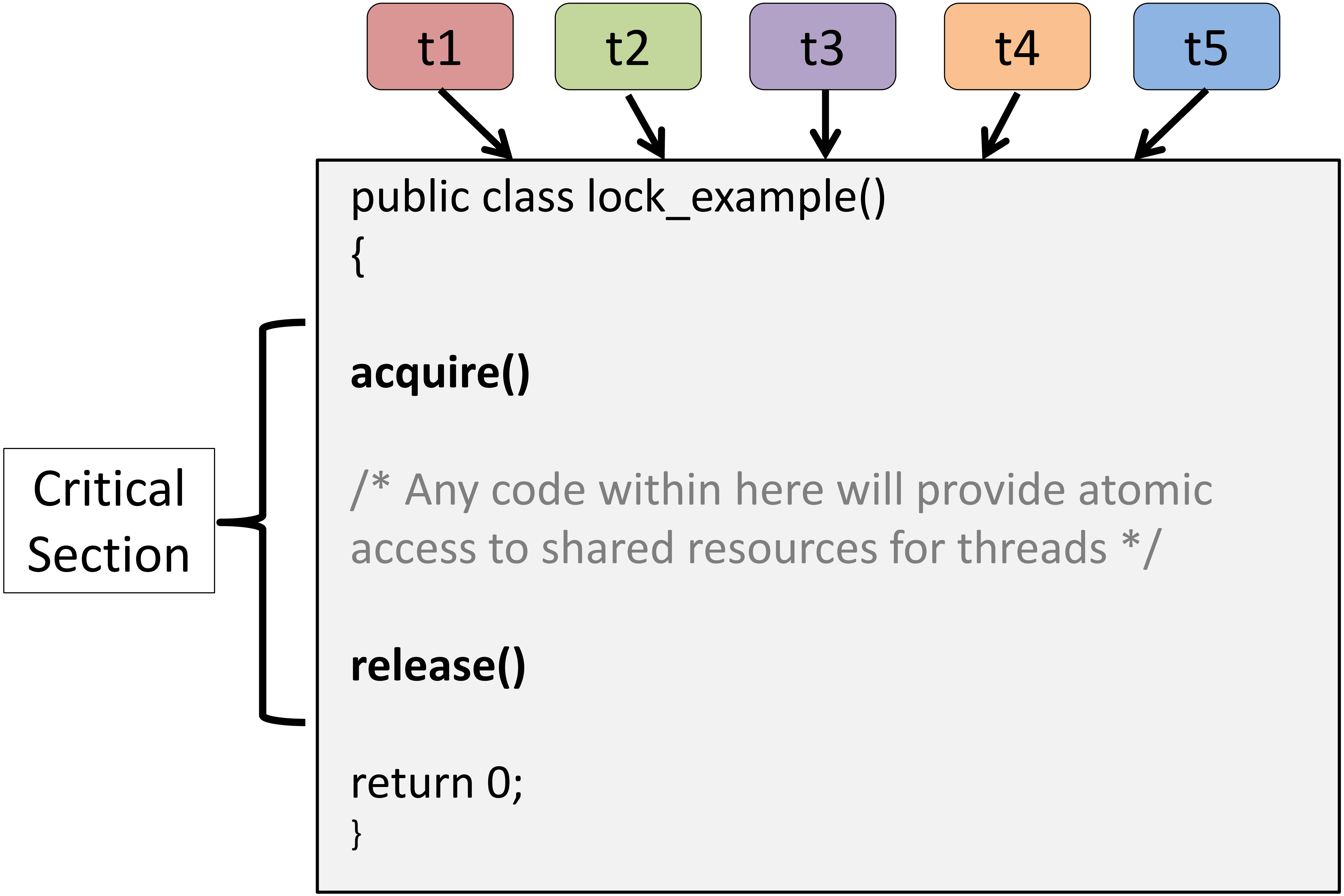
t2 acquires lock

t1 acquires lock

t2 releases lock

t1 releases lock

t3 acquires lock..



t1 t2 t3 t4 t5

Critical Section

```
public class lock_example()
{

acquire()

/* Any code within here will provide atomic
access to shared resources for threads */

release()

return 0;
}
```

Lock Declaration

Declared like a variable

- Lock L;

Programs can have multiple locks

- Lock L, P;

- Call **acquire** at <u>start of critical section</u>
- Call **release** at <u>end of critical section</u>

- Achieves mutual exclusion and progress

Acquisition of a lock blocks only threads attempting to acquire the same lock.

<u>Must use same lock for all critical sections accessing the same resource</u>

Mutual Exclusion in Java

Every Java object has an implicit (i.e. invisible) lock

Lock is achieved via the **synchronized** block

- Prevents thread entry to block until lock acquisition
- Lock release on block completion so other threads enter

synchronized(<object whose lock is needed>)

{<Arbitrary code block to be executed within held lock>}

----

synchronized **methodName (args) {body}**

is shorthand for

methodName(args) {synchronized(this) {body}}

Locks in Java - Method

```
public class Bank_account {

    private int bal = 0;

    public void Bank_account(int start_balance)
      {
      bal = start_balance;
     }

    public synchronized void update(int amount)
    {
      bal = balance + amount;
    }

}
```

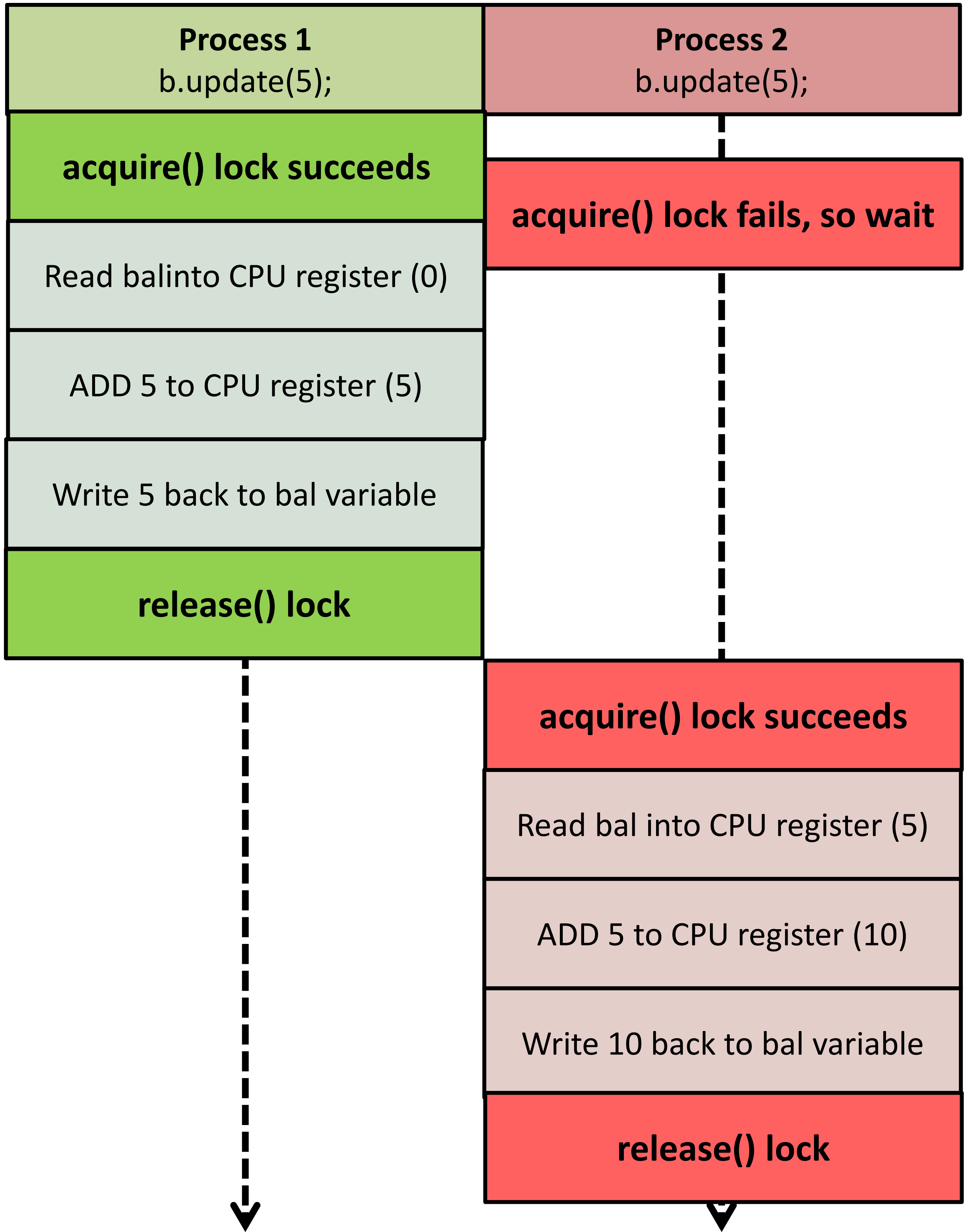
Locks in Java – Code Block

```
public class Bank_account {

      Private Object lockA = new Object();
      private int bal = 0;

      public void update(int amount)
     {
        synchronized (lockA)
         {
          try {
                Thread.sleep(1);              //Should we have this?
                }
           catch(InterruptedException e) {e.printStackTrace();
           }
          bal = balance + amount;
         }
}
```

Bank_account in action



Process 1
b.update(5);

acquire() lock succeeds

Read balinto CPU register (0)

ADD 5 to CPU register (5)

Write 5 back to bal variable

release() lock

Process 2
b.update(5);

acquire() lock fails, so wait

acquire() lock succeeds

Read bal into CPU register (5)

ADD 5 to CPU register (10)

Write 10 back to bal variable

release() lock

JVM calls acquire() at start of synchronized method

Calls release() at the end of synchronized method

Notice the sequentiality!

Terminology Recap

Critical section

- Logically, all the code that accesses a shared variable (resource).

Race condition

- Condition where incorrect program output may be generated depending on the interleaving of instructions from multiple threads.

Atomic code

- Code that is executed by a thread indivisibly from the point of view of other threads trying to execute the same code.
- Idea: To avoid race conditions, make critical sections atomic.

Mutual exclusion

- Only one thread may access a shared resource at a time.
- Ensures atomicity

Locks

- A mechanism for guaranteeing mutual exclusion.

Programming Coursework

- Coursework relies upon using Java threads and *synchronized* code
- You have all the concepts needed to implement the coursework
- Questions?