CS471- Parallel Processing

Dr. Ahmed Hesham Mostafa

Lecture 7 – Concurrency and Synchronization

Introduction

- A system typically consists of several (perhaps hundreds or even thousands) of threads running either in parallel. Threads often share user data.
- Meanwhile, the operating system continuously updates various data structures to support multiple threads.
- A race condition exists when access to shared data is not controlled, possibly resulting in corrupt data values.
- Process synchronization involves using tools that control access to shared data to avoid race conditions.
- These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

Introduction

- On the basis of synchronization, processes are categorized as one of the following two types:
- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.
- Process synchronization problem arises in the case of Cooperative process

Introduction

- Cooperating processes can be allowed to share data only through shared memory.
- Concurrent access to shared data may result in data inconsistency
- In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Race condition

- Race condition
  - Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count.
  - To make such a guarantee, we require that the processes be synchronized in some way.
- There are two types of race conditions:

1. Read-modify-write
2. Check-then-act

Race Condition Example (Train ticket booking)
Check-then-act

- Let's say there is only 1 ticket available on the train, and two passengers are trying to book that ticket at the same time without synchronization.
- It might happen that both might end up booking up tickets, though the only ticket was available, which is, of course, going to create a problem.

Race Condition Example (Train ticket booking)-

```
class TicketBooking imzplements Runnable{
    int ticketsAvailable=1;
    public void run(){
        System.out.println("Waiting to book ticket for : "+Thread.currentThread().getName());
        if(ticketsAvailable>0){
            System.out.println("Booking ticket for : "+Thread.currentThread().getName());

    //Let's say system takes some time in booking ticket (here we have taken 1 second time)
            try{
                Thread.sleep(900);
            }catch(Exception e){}
            ticketsAvailable--;
            System.out.println("Ticket BOOKED for : "+ Thread.currentThread().getName());
            System.out.println("currently ticketsAvailable = "+ticketsAvailable);
        }
        else{
            System.out.println("Ticket NOT BOOKED for : "+ Thread.currentThread().getName());
        }   }}
```

Race Condition Example (Train ticket booking)-

```
public class Main {
    public static void main(String[] args) {
        TicketBooking obj=new TicketBooking();

        Thread thread1=new Thread(obj,"Passenger1 Thread");
        Thread thread2=new Thread(obj,"Passenger2 Thread");

        thread1.start();
        thread2.start();
    }}
```

Race Condition Example (Train ticket booking)-

```
/*OUTPUT
Waiting to book ticket for : Passenger1 Thread
Waiting to book ticket for : Passenger2 Thread
Booking ticket for : Passenger1 Thread
Booking ticket for : Passenger2 Thread
Ticket BOOKED for : Passenger1 Thread
currently ticketsAvailable = 0
Ticket BOOKED for : Passenger2 Thread
currently ticketsAvailable = -1
*/
```

Race Condition Example (Train ticket booking)

- If we note the above program, first **Passenger1 Thread** and **Passenger2 Thread** waited to book tickets.
- Then, both threads tried to check the available ticket count and it was 1.
- Both threads were able to book tickets.
- And ultimately available ticket was reduced to **-1,** which is **practically impossible**, tickets count can never dip below 0.
- **RACE CONDITION PROBLEM: 1 ticket was booked by 2 passengers.**

Counter Race Condition Example-

```
class task implements Runnable{
    private int count=0;
    @Override
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            count++;
        }
        System.out.println(Thread.currentThread().getName()+"Counter =" + count);
    }}
```

Counter Race Condition Example-

```
public class Main {
    public static void main(String[] args) {
        task mytask=new task();
        Thread t1=new Thread(mytask, "threadA ");
        Thread t2=new Thread(mytask, "threadB ");
        t1.start();
        t2.start();

    }
}
```

Counter Race Condition Example-

- When running the program three times there are different outputs?
- threadA Counter =1030992
- threadB Counter =1904891
- threadB Counter =1839097
- threadA Counter =1054236
- threadB Counter =1035076
- threadA Counter =1910447
- All outputs are wrong as one of two thread must reach to 2000000 to exit from for loop

Critical Section Problem

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

General structure of a typical process P_i

Reorder instruction problems in Multithreading applications

- **Two threads share the data:**

```
boolean flag = false;
int x = 0;
```

- **Thread 1 performs**

```
while (!flag);
    ;
print x
```

- **Thread 2 performs**

```
x = 100;
flag = true
```

- **What is the expected output?**

Reorder instruction problems in Multithreading applications

- The expected behavior is, of course, that Thread 1 outputs the value 100 for variable x.
- However, as there are no data dependencies between the variables flag and x, it is possible that a processor may reorder the instructions for Thread 2 so that flag is assigned true before the assignment of x = 100.
- In this situation, it is possible that Thread 1 would output 0 for variable x.

- **Two threads share the data:**

```
boolean flag = false;
int x = 0;
```

- **Thread 1**

```
while (!flag);

print x
```

- **Thread 2**

```
flag = true
x = 100;
```

Reorder instruction problems Counter Example

```java
public class Main {
    public static   boolean stopcount=false;
    public static void counter(){
        int x=0;
        System.out.println("inside counter");
        while(!stopcount){
            x++;
        }
        System.out.println("Count = "+x);
    }
```

```
// Compiler optimizes to this:
x = 0;
if (!s_stopcount)
   while (true) x++; // Faster!
System.out.println("Count = "+x);
```

```java
public static void main(String[] args) {
   Thread thread1 = new Thread(() -> { counter();});
    thread1.start();
    try {
        Thread.sleep(100);
        System.out.println("thread sleeping");
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    stopcount=true;
}
```

//output will be
inside counter
thread sleeping

//The Correct Output must be
inside counter
thread sleeping
Count = 105820573

Reorder instruction problem Counter Example

- In this code, the Main method creates a new thread that executes the counter method.
- This counter method counts as high as it can before being told to stop.
- The Main method allows the counter thread to run for 100 milseconds before telling it to stop by setting the static **boolean** field to **true**.
- At this point, the counter thread should display what it counted up to, and then the thread will terminate.

Reorder instruction problem Counter Example

- Looks simple enough, right? Well, the program has a potential problem due to all the optimizations that could happen to it.
- You see, when the Counter method is compiled, the compiler sees that stopcount is either true or false, and it also sees that this value never changes inside the counter method itself.
- So the compiler could produce code that checks stopcount first.

Reorder instruction problem Counter Example

- If stopcountis true, then counter: stopped when x=0 will be displayed.
- If stopcount is false, then the compiler produces code that enters an infinite loop that increments x forever.
- You see, the optimizations cause the loop to run very fast because checking stopcount only occurs once before the loop; it does not get checked with each iteration of the loop.

Hardware Solutions for Critical Section Problem

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

  1. Memory barriers
  2. Hardware instructions
  3. Atomic variables

Memory Model

- How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model.**
- Memory models may be either:
- **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
- **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.

Memory Model

- Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor.
- To address this issue, computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors.
- Such instructions are known as **memory barriers or memory fences.**

Memory Model

- When a **memory barrier** instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Java Memory Model (volatile)

- **Volatile key word in java solve two problems:**
  - **Visibility of Shared Objects (This problem in Multicore Processors)**
  - **Reordering Instructions**

Java Memory Model (volatile) Visibility of Shared Objects

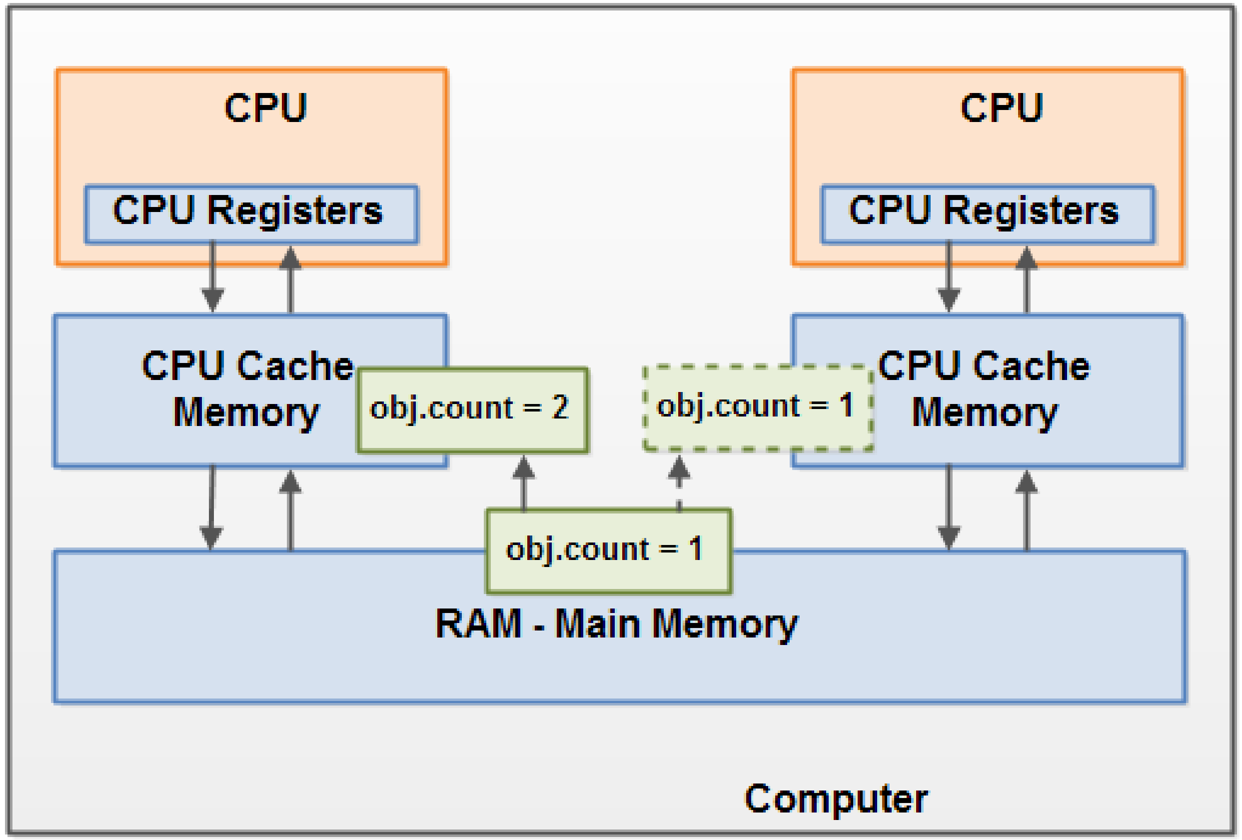
- **Visibility of Shared Objects**
- When you write to a Java volatile variable the value is guaranteed to be written directly to main memory. Additionally, all variables visible to the thread writing to the volatile variable will also get synchronized to main memory.
- When you read the value of a Java volatile the value is guaranteed to be read directly from memory. Furthermore, all the variables visible to the thread reading the volatile variable will also have their values refreshed from main memory.
-

Java Memory Model (volatile) Visibility of Shared Objects

- If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.
- Imagine that the shared object is initially stored in main memory.
- A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object.
- As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs.
- This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

Java Memory Model (volatile) Visibility of Shared Objects

- The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2.
- When thread 2 read value of count will read 1
- This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



Java Memory Model (volatile) Reordering

- The reading and writing instructions of volatile variables cannot be reordered by the JVM
- **Happens Before Guarantee for Writes to volatile Variables**
  - A write to a non-volatile or volatile variable that happens before a write to a volatile variable is guaranteed to happen before the write to that volatile variable. (any instructions before volatile instruction will happen before volatile instructions)
- **Happens Before Guarantee for Reads of volatile Variables**
  - A read of a volatile variable will happen before any subsequent reads of volatile and non-volatile variables(all reads *after* the volatile read will remain after the volatile read).

Solve Reorder instruction problems in Peterson Using Memory Barriers

```
class Main {
    static    int val=0;
    static     volatile boolean[] flag = {false, false};
    static volatile int turn=0;
    static Thread process(int i) {
        return new Thread(() -> {
            int j = 1 - i;
            for (int n=0; n<1000000; n++) {
                flag[i] = true; // 1
                turn=j;
                while (flag[j]&&turn==j) ;
                val++;   //critical section
                flag[i] = false; // // UNLOCK
            }
        });
    }
```

Solve Reorder instruction problems in Peterson Using Memory Barriers

```
public static void main(String[] args) {
    try {
        Thread p0 = process(0);
        Thread p1 = process(1);
        p0.start();
        p1.start();
        p0.join();
        p1.join();
        System.out.println("val = "+val);
    }
    catch (InterruptedException e) {}
}
```

val= 2000000

Solve Reorder instruction problems Counter Example

```java
public class Main {
    public static volatile boolean stopcount=false;
    public static void counter(){
        int x=0;
        System.out.println("inside counter");
        while(!stopcount){
            x++;
        }
        System.out.println("Count = "+x);
    }
```

```java
public static void main(String[] args) {
   Thread thread1 = new Thread(() -> { counter();});
    thread1.start();
    try {
        Thread.sleep(100);
        System.out.println("thread sleeping");
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    stopcount=true;
}
```

Solve Reorder instruction problems Counter Example

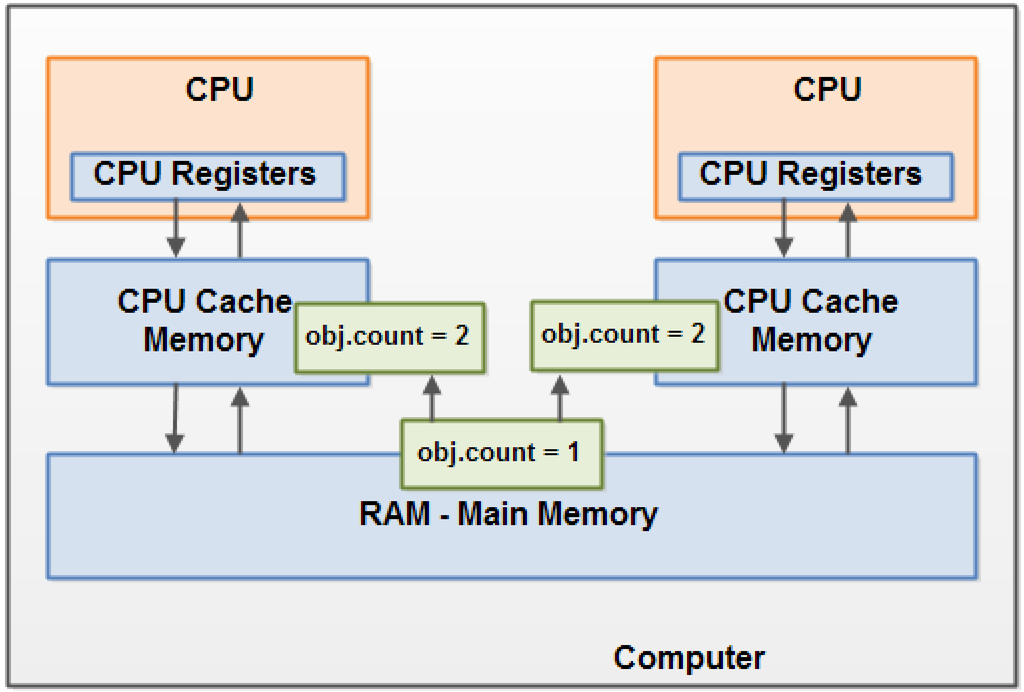
```
//Run Output
inside counter
thread sleeping
Count = 105820573
```

Volatile Notes:

- Volatile solve the problem of reordering and the problem of variable visibility in multicore processors but not it doesn't guarantee that it will solve the synchronization problem for race condition.

- **Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.**

- **To solve this problem, you can use a Java synchronized block. A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time.**



Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Atomic Variables

The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
      int temp;

      do {
            temp = *v;
      }
      while (temp!=(compare_and_swap(v,temp,temp+1));
}
```

```
int compare _and_swap(int *value, int expected, int new_value) {
      int temp = *value;

      if (*value == expected)
         *value = new_value;
    return temp;
   }
```

Counter Example

```
class Main extends Thread {
      static int val=0;
   public void run()
   {
      for (int i = 0; i < 1_000000; i++) {
         val++;
      }
   }
```

```
//output run
1550896
```

```
public static void main(String[] args)
        throws InterruptedException
   {
      Main c = new Main();
      Thread first = new Thread(c, "First");
      Thread second = new Thread(c, "Second");
      Thread third = new Thread(c, "Third");
      Thread fourth = new Thread(c, "Fourth");
       first.start();
      second.start();
      third.start();
      fourth.start();
      // main thread will wait for child threads to complete execution
      first.join();
      second.join();
      third.join();
      fourth.join();
      System.out.println(val);
   }
}
```

Counter Example with Atomic method getAndSet

```java
import java.util.concurrent.atomic.AtomicBoolean;
class Main extends Thread {
   AtomicBoolean lock=new AtomicBoolean(false);
   static int val=0;
   public void run()
   {
      for (int i = 0; i < 1_000000; i++) {
         while(lock.getAndSet(true));
         val++;
         lock.set(false);
      }
   }
```

TestAndSet

//output run
4000000

```java
public static void main(String[] args)
        throws InterruptedException
  {
     Main c = new Main();
     Thread first = new Thread(c, "First");
     Thread second = new Thread(c, "Second");
     Thread third = new Thread(c, "Third");
     Thread fourth = new Thread(c, "Fourth");
      first.start();
     second.start();
     third.start();
     fourth.start();
     // main thread will wait for child threads to complete execution
     first.join();
     second.join();
     third.join();
     fourth.join();
     System.out.println(val);
  }
}
```

Counter Example with Atomic method addAndGet

```java
import java.util.concurrent.atomic.AtomicInteger;
class Main extends Thread {
    AtomicInteger count=new AtomicInteger();
     public void run()
    {
       for (int i = 0; i < 1_000000; i++) {
                count.addAndGet(1);
       }
    }
```

increment

```
//output run
4000000
```

```java
public static void main(String[] args)
        throws InterruptedException
   {
      Main c = new Main();
      Thread first = new Thread(c, "First");
      Thread second = new Thread(c, "Second");
      Thread third = new Thread(c, "Third");
      Thread fourth = new Thread(c, "Fourth");
       first.start();
      second.start();
      third.start();
      fourth.start();
      // main thread will wait for child threads to complete execution
      first.join();
      second.join();
      third.join();
      fourth.join();
      System.out.println(val);
   }
}
```

Counter Example with Atomic method getAndSet

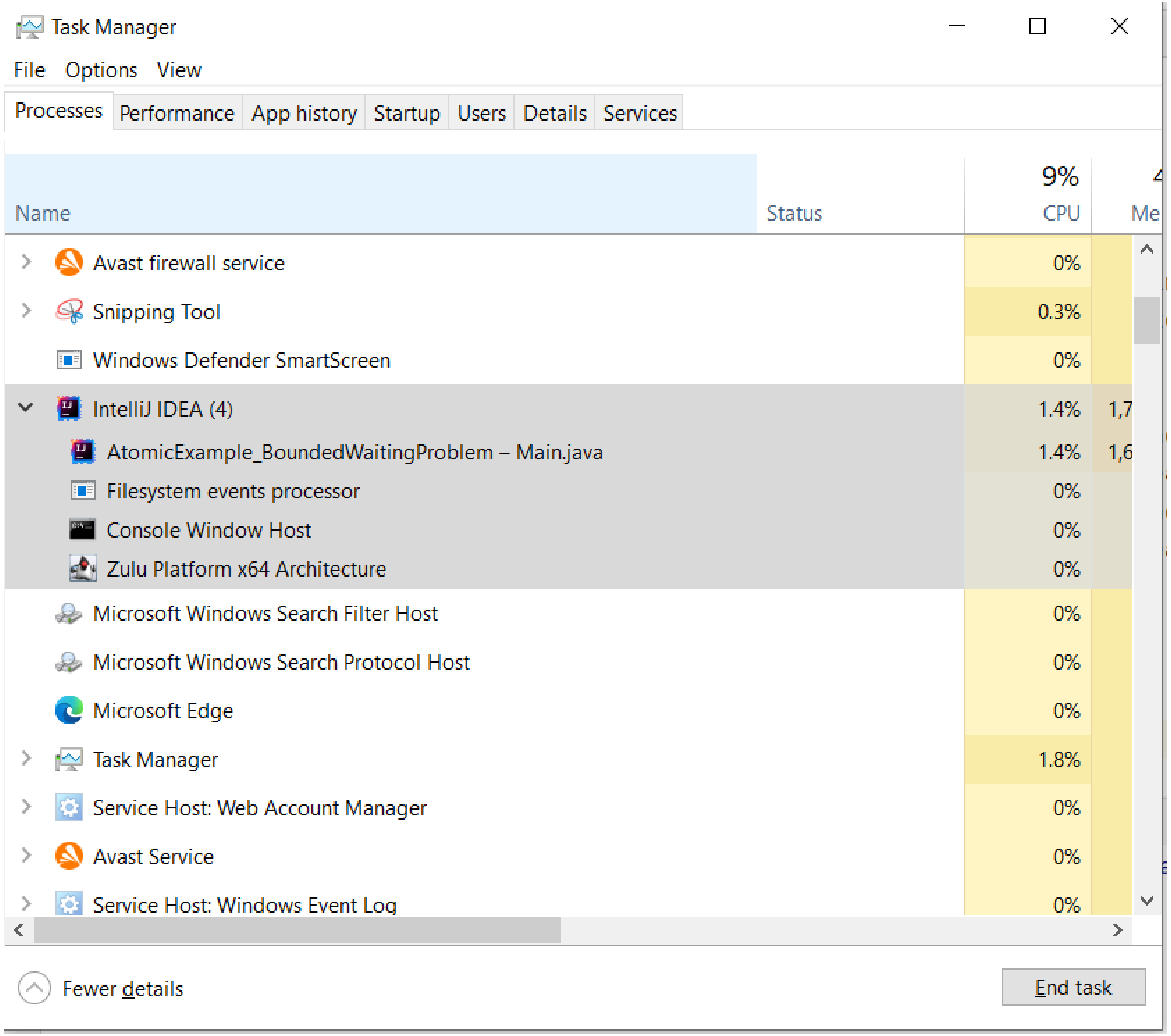
```java
import java.util.concurrent.atomic.AtomicInteger;
class Main extends Thread {
  AtomicInteger count=new AtomicInteger();
   static int val=0;
   public void run()
   {
       for (int i = 0; i < 1_000000; i++) {
          while(lock. compareAndExchange(0,1)!=0);
          val++;
          lock.set(0);
       }
   }
```

CompareAndSwap

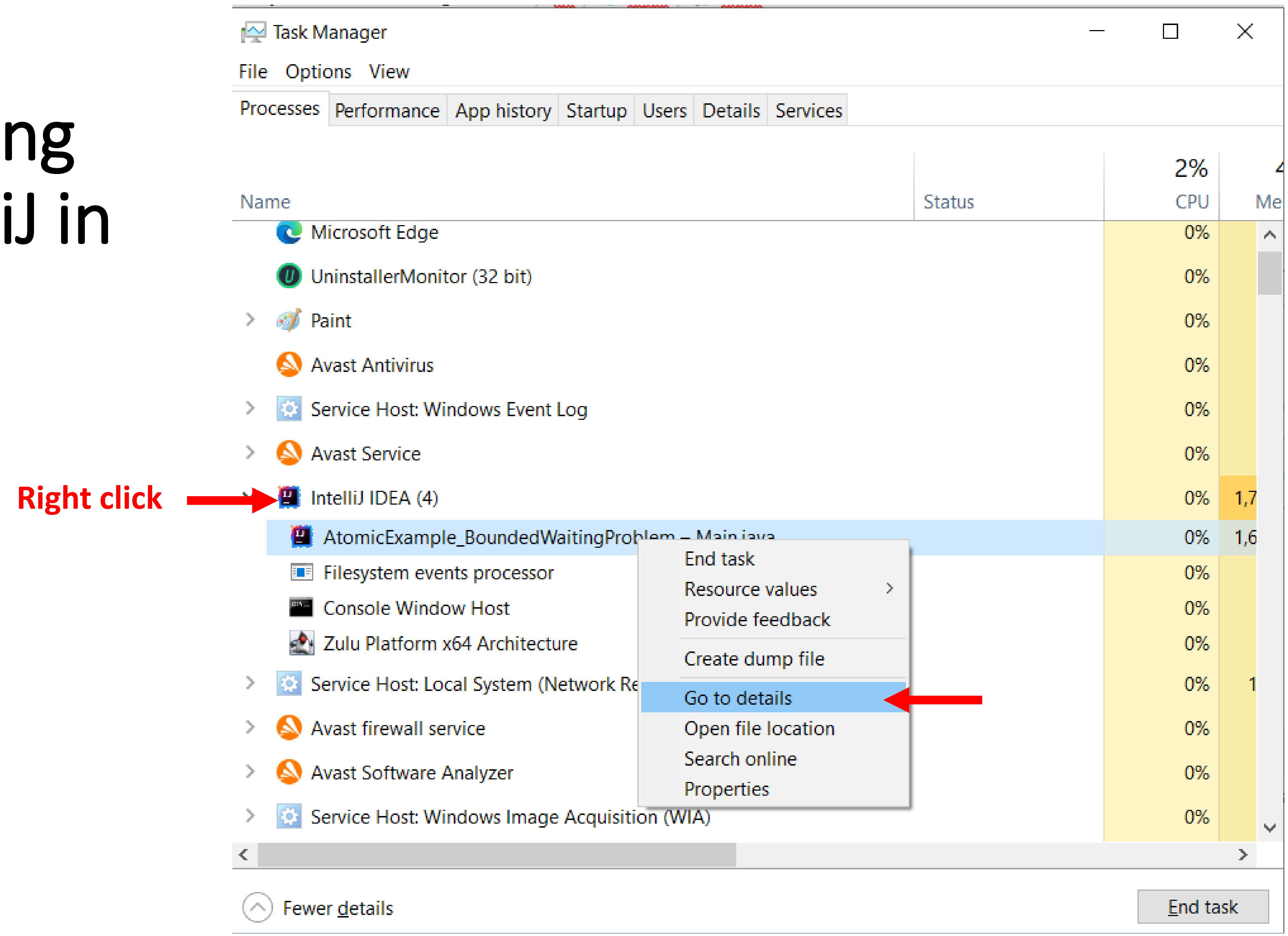
//output run
4000000

```java
public static void main(String[] args)
        throws InterruptedException
   {
      Main c = new Main();
      Thread first = new Thread(c, "First");
      Thread second = new Thread(c, "Second");
      Thread third = new Thread(c, "Third");
      Thread fourth = new Thread(c, "Fourth");
       first.start();
      second.start();
      third.start();
      fourth.start();
      // main thread will wait for child threads to complete execution
      first.join();
      second.join();
      third.join();
      fourth.join();
      System.out.println(val);
   }
}
```

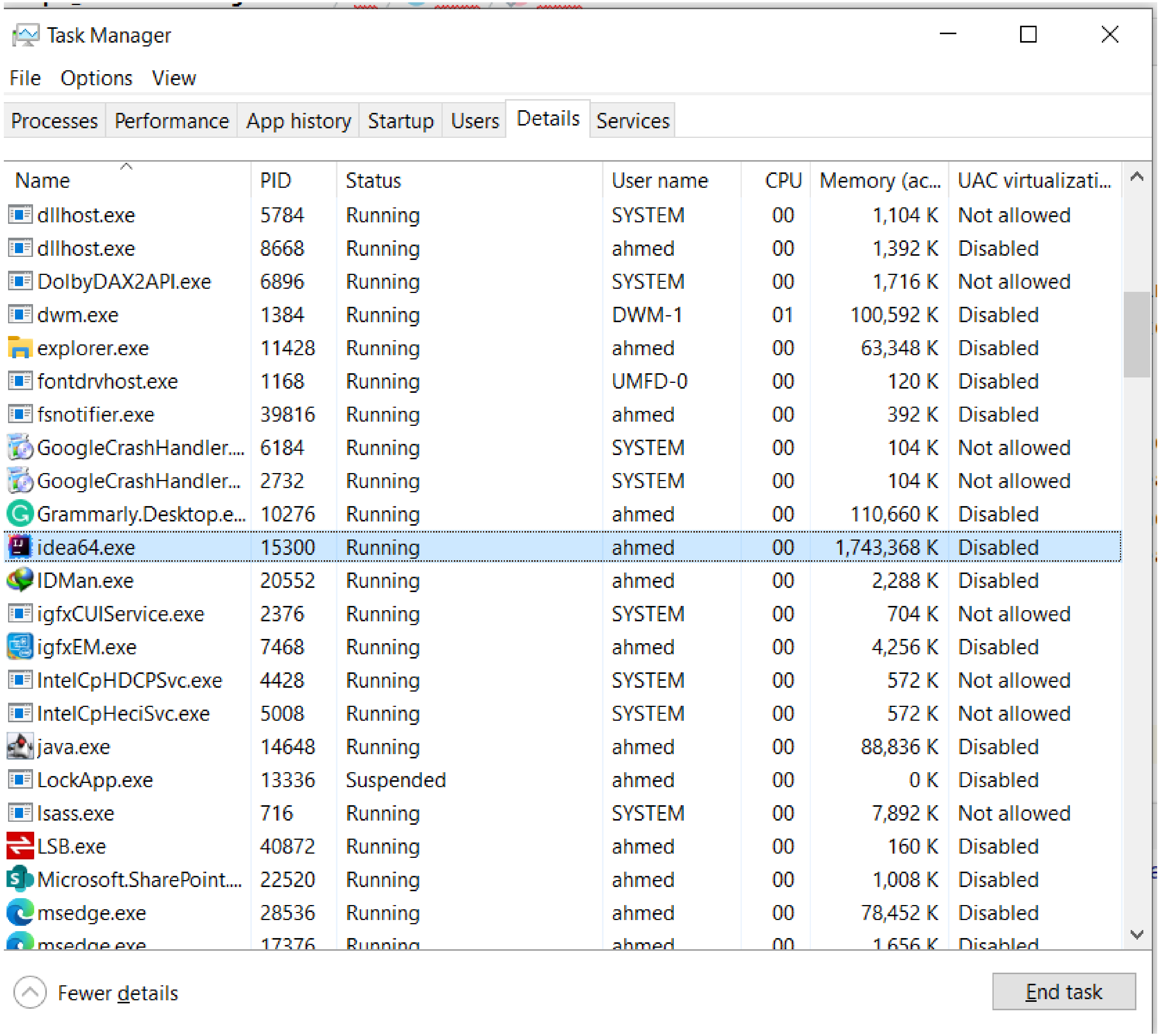
Run Boundedwaiting project in IntelliJ in one cpu



Run Boundedwaiting project in IntelliJ in one cpu



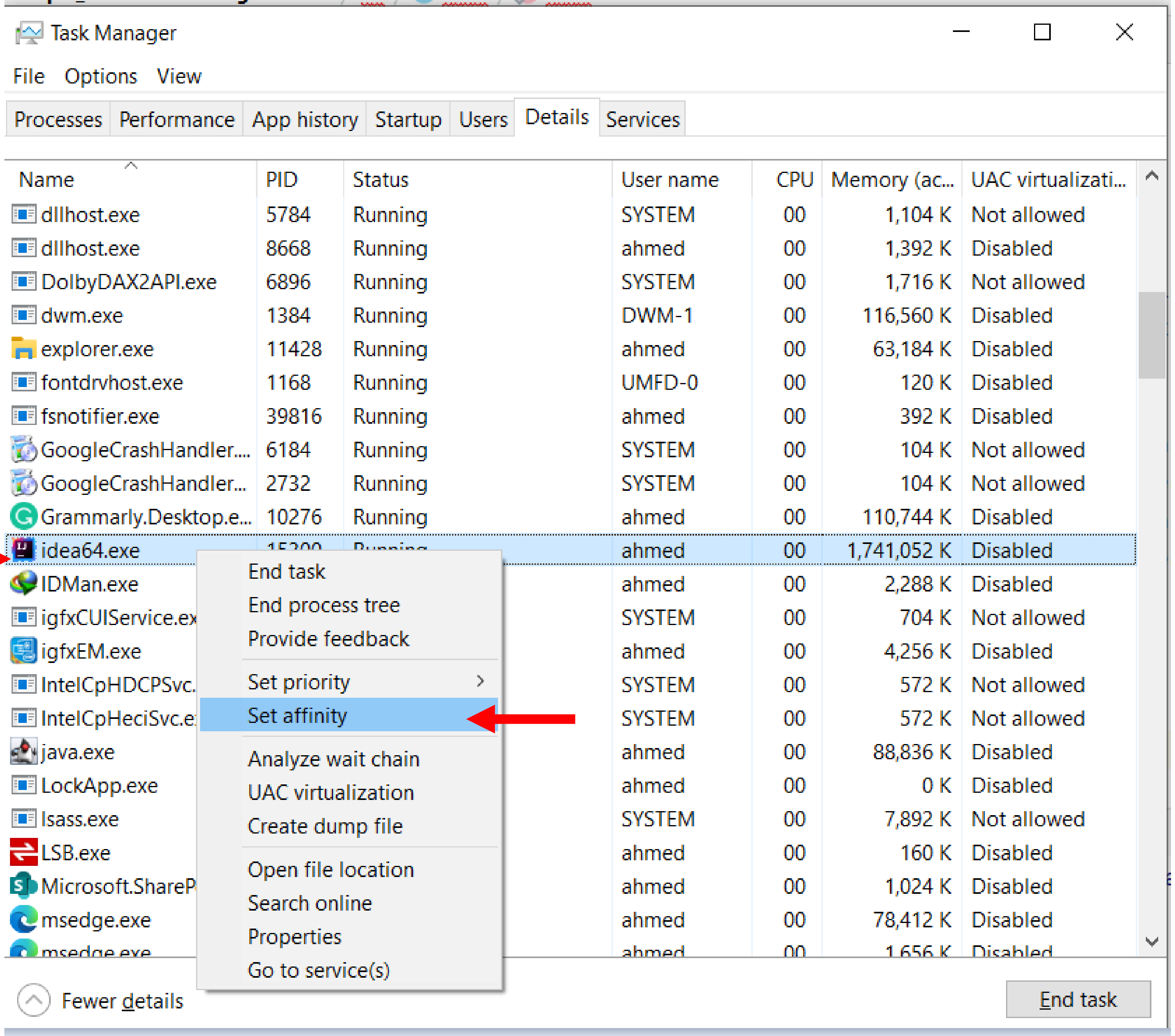
Run Boundedwaiting project in IntelliJ in one cpu



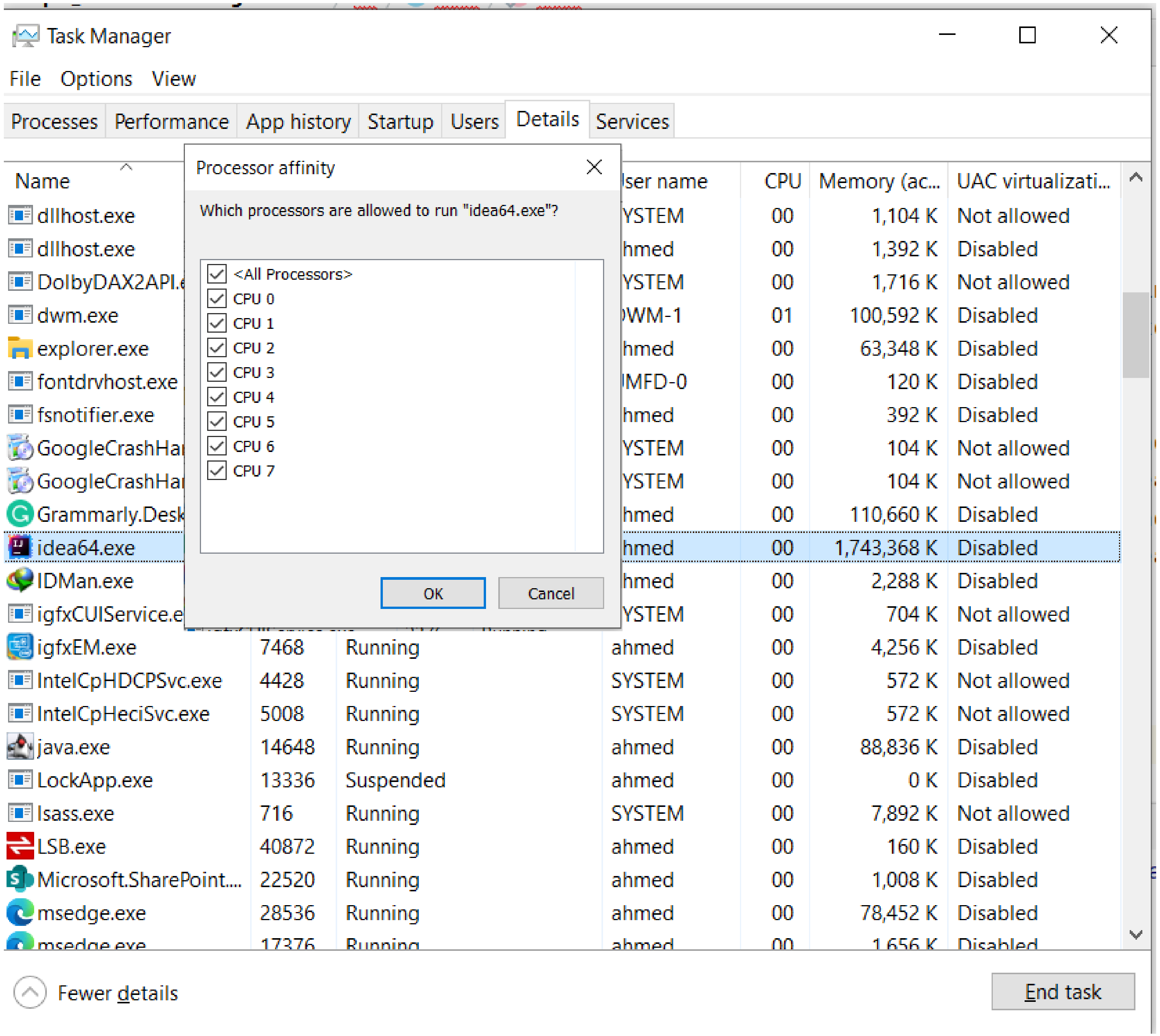
Name	PID	Status	User name	CPU	Memory (ac...	UAC virtualizati...
dllhost.exe	5784	Running	SYSTEM	00	1,104 K	Not allowed
dllhost.exe	8668	Running	ahmed	00	1,392 K	Disabled
DolbyDAX2API.exe	6896	Running	SYSTEM	00	1,716 K	Not allowed
dwm.exe	1384	Running	DWM-1	01	100,592 K	Disabled
explorer.exe	11428	Running	ahmed	00	63,348 K	Disabled
fontdrvhost.exe	1168	Running	UMFD-0	00	120 K	Disabled
fsnotifier.exe	39816	Running	ahmed	00	392 K	Disabled
GoogleCrashHandler....	6184	Running	SYSTEM	00	104 K	Not allowed
GoogleCrashHandler...	2732	Running	SYSTEM	00	104 K	Not allowed
Grammarly.Desktop.e...	10276	Running	ahmed	00	110,660 K	Disabled
idea64.exe	15300	Running	ahmed	00	1,743,368 K	Disabled
IDMan.exe	20552	Running	ahmed	00	2,288 K	Disabled
igfxCUIService.exe	2376	Running	SYSTEM	00	704 K	Not allowed
igfxEM.exe	7468	Running	ahmed	00	4,256 K	Disabled
IntelCpHDCPSvc.exe	4428	Running	SYSTEM	00	572 K	Not allowed
IntelCpHeciSvc.exe	5008	Running	SYSTEM	00	572 K	Not allowed
java.exe	14648	Running	ahmed	00	88,836 K	Disabled
LockApp.exe	13336	Suspended	ahmed	00	0 K	Disabled
lsass.exe	716	Running	SYSTEM	00	7,892 K	Not allowed
LSB.exe	40872	Running	ahmed	00	160 K	Disabled
Microsoft.SharePoint....	22520	Running	ahmed	00	1,008 K	Disabled
msedge.exe	28536	Running	ahmed	00	78,452 K	Disabled
msedge.exe	17376	Running	ahmed	00	1,656 K	Disabled

Run Boundedwaiting project in IntelliJ in one cpu

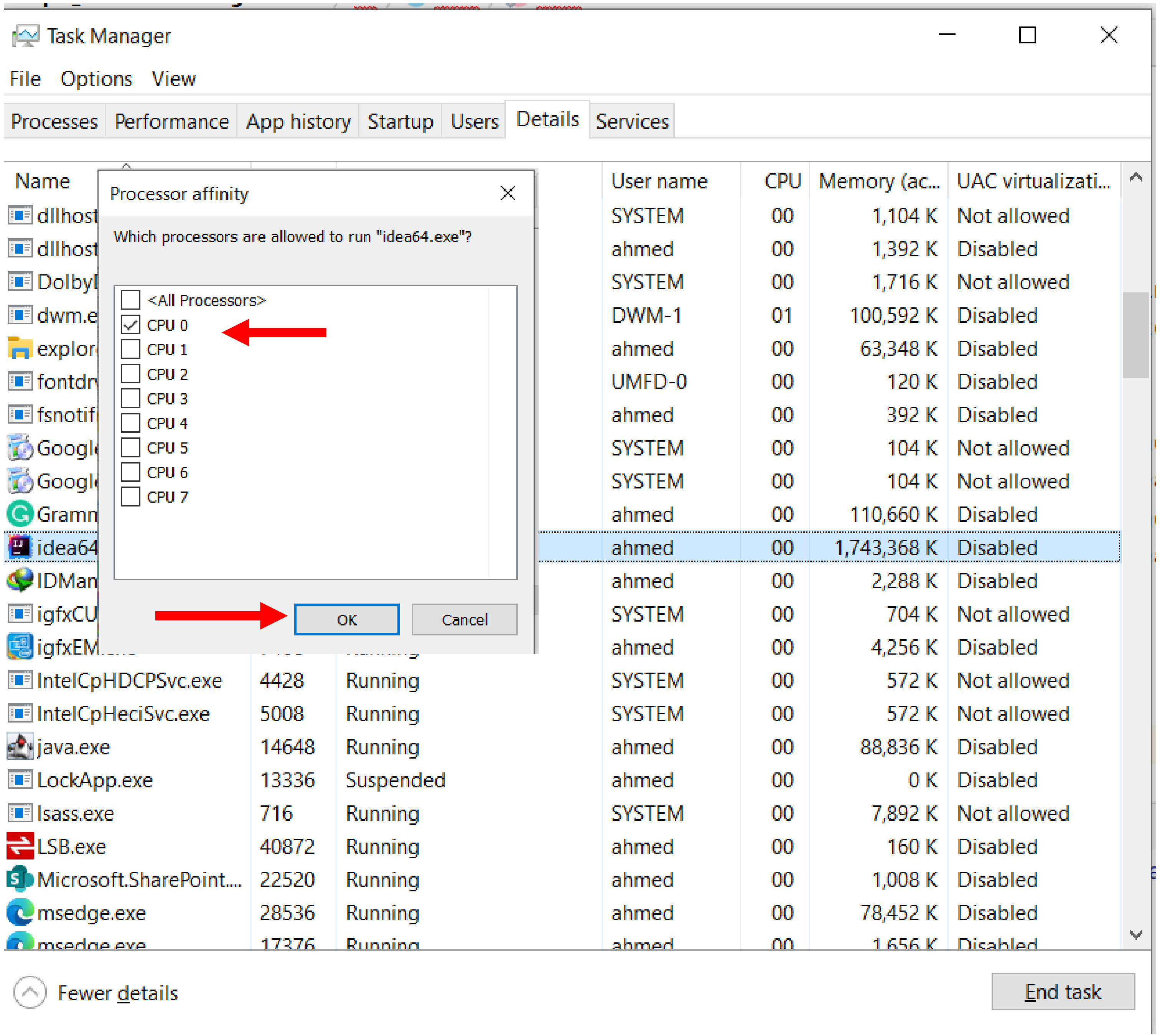
Right click



Run Boundedwaiting project in IntelliJ in one cpu



Run Boundedwaiting project in IntelliJ in one cpu



Semaphore

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem (APIs)
- Synchronization tool provided by the OS that does not require busy waiting
- One of this sycnonization tools is **Semaphore**

Semaphore

- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - (Originally called **`P()`** and **`V()`**)

Semaphore

- Definition of the `wait() operation`

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the `signal() operation`

```
signal(S) {
    S++;
}
```

- When one process modifies the semaphore value, no other process can modify the semaphore value.

- Also in *wait(S)*, the testing of the integer value of *S* (*S* <= 0), as well as possible modifications (*S=S-1*), must be executed without interrruption

Semaphore

- Shared data:

  semaphore mutex = 1;

- Process *Pi*

```
do {
    wait(mutex);

        critical section

    signal(mutex);

        remainder section

} while (true);
```

- There are counting and binary semaphores in the operating systems.
- Counting semaphores value can range over an unrestricted domain. Used to control access to a given resource consisting a number of instances. The semaphore is initialized to the number of resource available.
- Binary semaphores value can range only between 0 and 1.

Semaphore Implementation

- Must guarantee that no two processes can execute the wait() and signal() **on the same semaphore** at the same time
- The semaphore definition above and all other mutual exclusion solutions given so far require "busy waiting" (continuo looping in entry code). Busy waiting wastes CPU cycles so must be dealt with.
- To avoid busy waiting: When a process has to wait, it will be put in a blocked queue of processes waiting for the same event.
- Assume two simple operations:
  - **block()** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Semaphore Implementation

- **Semaphore operations on variable S are now defined as,**

```
wait(S)    {

    S.value--;
    if (S.value < 0) {
            add this process to S.List;
            block(); //suspends, waiting state
        }
    }

signal(S)  {
    S.value++;
    if (S.value <= 0) {
            remove a process P from S.List;
            wakeup(P); //resumes, ready state
        }
    }
```

```
struct Semaphore
{
        int value;
        int List[n];
};
Semaphore S;
```

Here, semaphore value may be negative. If it is, its magnitude is the number processes waiting on that semphore. abs(value) shows the number of blocked processes. value < = 0 means that there is a process to wakeup.
For Example, if S.value = 5 and there 10 processes so 5 processes will enter critical section and 5 processes will wait
so s.value = -5

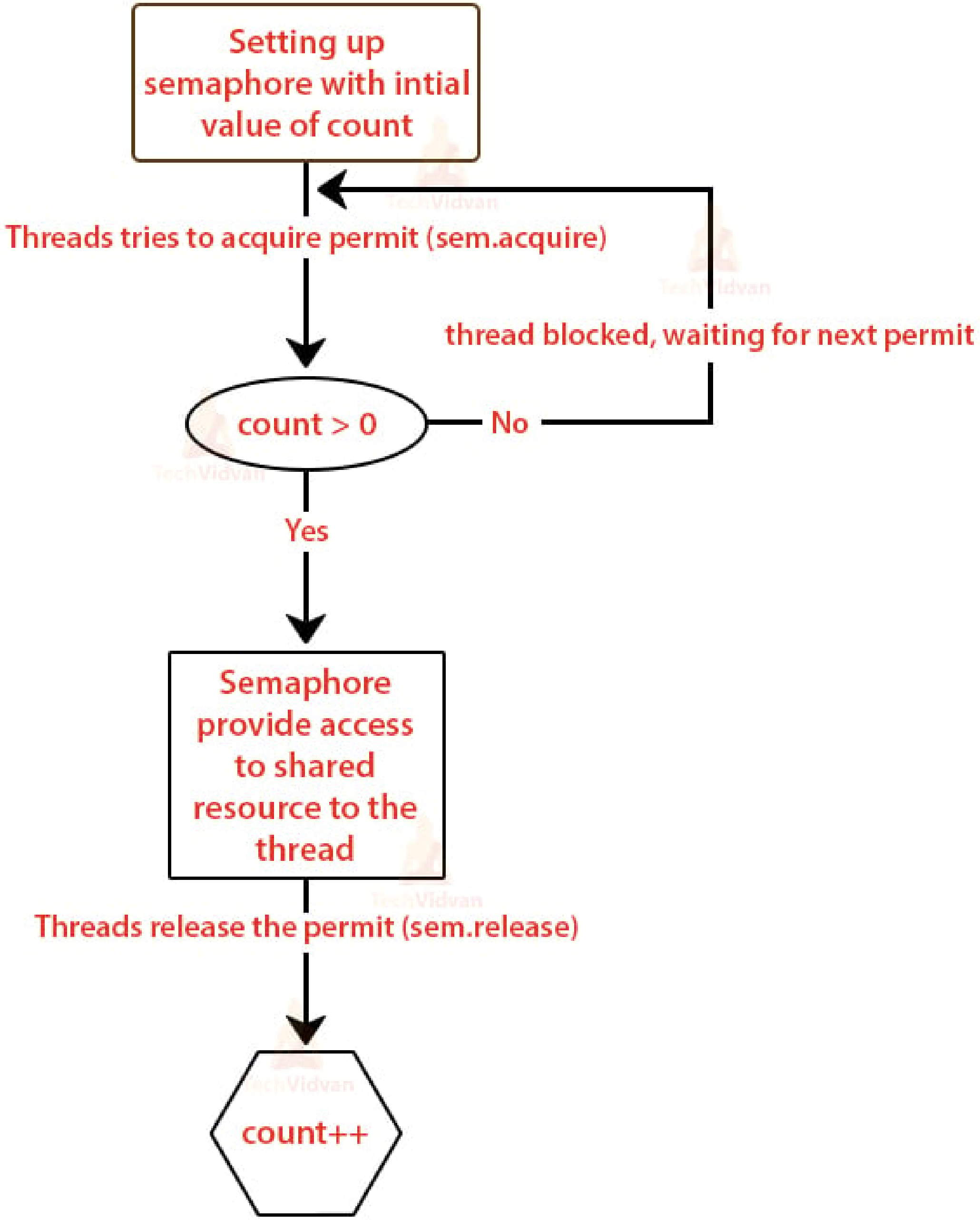
Problems with Semaphores

- Incorrect use of semaphore operations:
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*);	*signal*(*S*);

- P_0 executes wait(S) and P_1 executes wait(Q), when P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait (S), it must wait until P_0 executes signal (S). Since these signal() operations cannot be executed, P_1 and P_1 are deadlocked.

Working of Semaphore in Java



Counter Example Using Semaphore

```java
import java.util.concurrent.Semaphore;
class Main extends Thread {
    Semaphore sem = new Semaphore(3);
    static int val=0;
    public void run()
    {
        try {
            sem.acquire();
            System.out.println(Thread.currentThread().getName() + " Enter in critical section" );
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + " val=" + (++val));
            } }
        catch(InterruptedException e){
            throw new RuntimeException(e);
        }
        System.out.println(Thread.currentThread().getName() + " exit critical section" );
        sem.release();
    }
```

```java
public static void main(String[] args)
        throws InterruptedException
  {
     Main c = new Main();
     Thread first = new Thread(c, "First");
     Thread second = new Thread(c, "Second");
     Thread third = new Thread(c, "Third");
     Thread fourth = new Thread(c, "Fourth");
     first.start();
     second.start();
     third.start();
     fourth.start();
     first.join();
     second.join();
     third.join();
     fourth.join();
     System.out.println(val);
  }
}
```

```
Third Enter in critical section
Second Enter in critical section
First Enter in critical section
Second val=2
Second val=4
Second val=5
Second val=6
Second val=7
First val=3
Third val=1
Third val=9
Third val=10
Third val=11
First val=8
Third val=12
Third exit critical section
Second exit critical section
Fourth Enter in critical section
Fourth val=14
Fourth val=15
Fourth val=16
Fourth val=17
Fourth val=18
Fourth exit critical section
First val=13
First val=19
First val=20
First exit critical section
20
```

As shown in output only three processes entered the critical section at the same time while the fourth process waited until any process exit the critical section

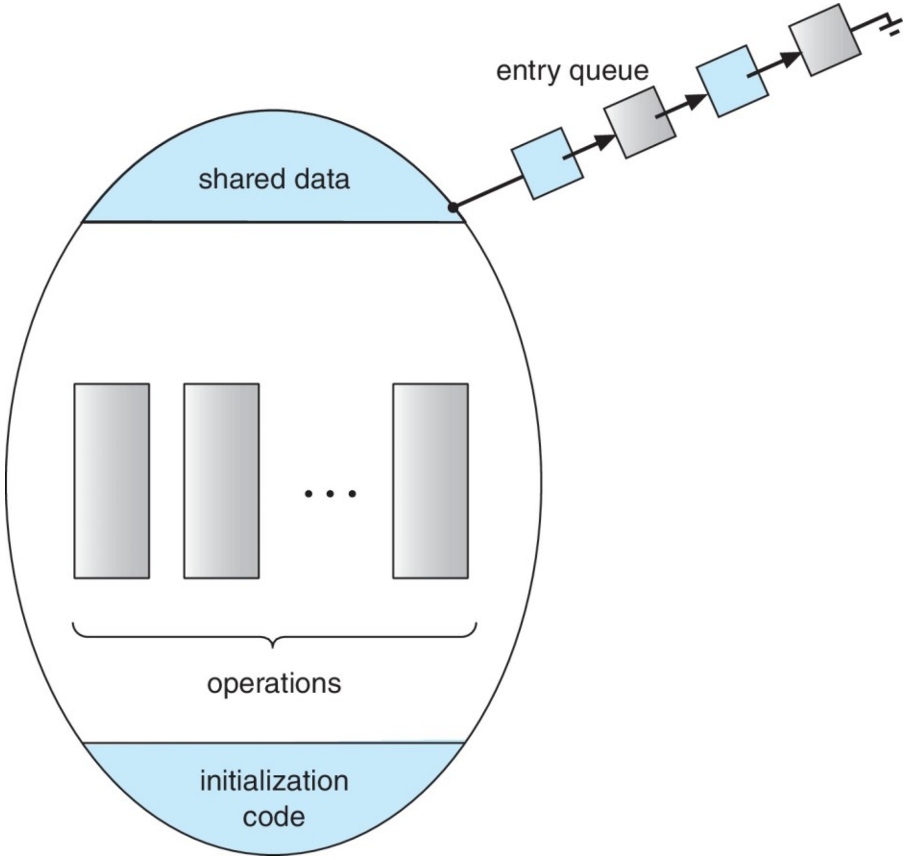
Counter Example Using Semaphore

Monitors

- A high-level abstraction that provides an effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

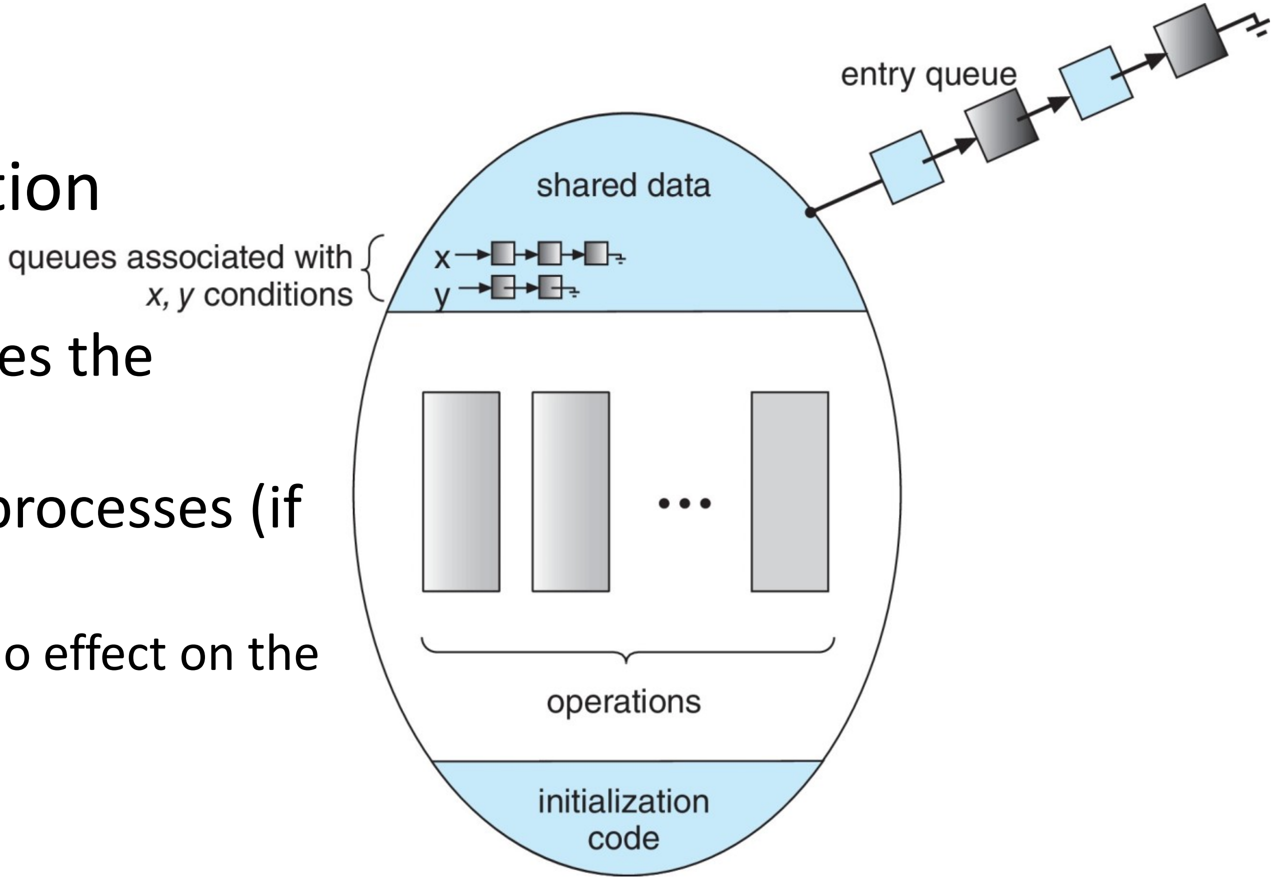
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

   initialization code (…) { … }
}
```

Schematic view of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** **(Block)**– a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** (**Wakeup**)– resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

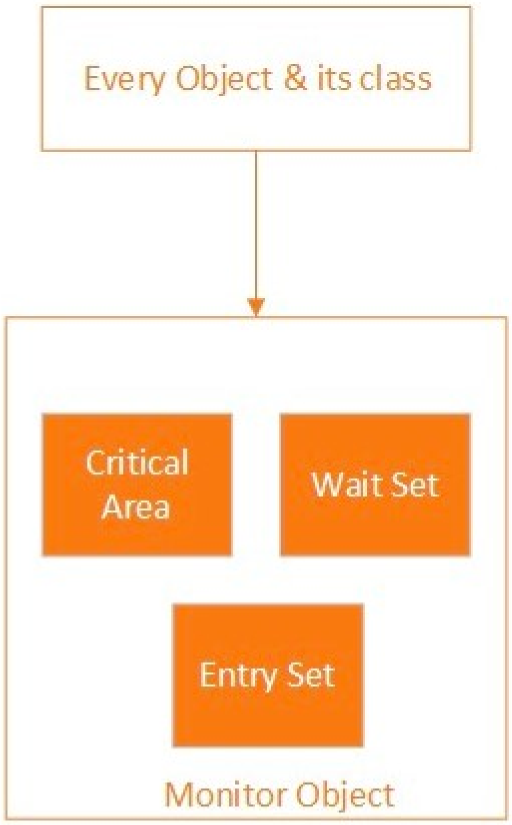


Monitor in Java

- Monitors basically ‘monitor’ the access control of shared resources and objects among threads.
- Using this construct only one thread at a time gets access control over the critical section at the resource while other threads are blocked and made to wait until certain conditions.
- In Java, monitors are implemented using synchronized keywords (synchronized blocks, synchronized methods, or classes).
- Java's monitor supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*.
- Whereas mutual exclusion helps keep threads from interfering with one another while sharing data, cooperation helps threads to work together towards some common goal.

Monitor in Java

- In java, every object has a single lock (monitor) associated with it.
- Instance variables of objects that need to be protected from concurrent access including a critical area for a monitor that is associated with the object and Instance variables of classes /static variables of a class that needs to be protected from concurrent access included in the critical area for the monitor which is associated with the class.



Monitor in Java

- This critical area is protected with a lock and this lock ensures mutual exclusion.
- A Wait set is also associated with a monitor that is used to provide coordination between threads.
- An entry set is used to hold the threads that are already requested for the lock and the lock is not acquired by them yet.*

Monitor in Java - mutual exclusion

- When a thread arrives at the beginning of a monitor region, it is placed into an *entry set* for the associated monitor.
- If no other thread is waiting in the entry set and no other thread currently owns the monitor, the thread acquires the monitor and continues executing the monitor region.
- When the thread finishes executing the monitor region, it exits (and releases) the monitor.
- If a thread arrives at the beginning of a monitor region that is protected by a monitor already owned by another thread, the newly arrived thread must wait in the entry set.

Monitor in Java - mutual exclusion

- When the current owner exits the monitor, the newly arrived thread must compete with any other threads also waiting in the entry set.
- Only one thread will win the competition and acquire the monitor.
- The first kind of synchronization listed above, is mutual exclusion.

Monitor in Java - Cooperation

- Cooperation is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state.
- For example, one thread, a "read thread," may be reading data from a buffer that another thread, a "write thread," is filling. The read thread needs the buffer to be in a "not empty" state before it can read any data out of the buffer.
- If the read thread discovers that the buffer is empty, it must wait.

Monitor in Java - Cooperation

- The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading.
- The form of monitor used by the Java virtual machine is called a "Wait and Notify" monitor. (It is also sometimes called a "Signal and Continue" monitor.) In this kind of monitor, a thread that currently owns the monitor can suspend itself inside the monitor by executing a *wait command*.
- When a thread executes a wait, it releases the monitor and enters a *wait set*. The thread will stay suspended in the wait set until some time after another thread executes a *notify command* inside the monitor. When a thread executes a notify, it continues to own the monitor until it releases the monitor of its own accord, either by executing a wait or by completing the monitor region.
- Note: the waiting thread may be suspended itself because the data protected by the monitor wasn't in a state that would allow the thread to continue doing useful work.

Monitor in Java - mutual exclusion

Synchronization in Java

- The thread which is entering into a synchronized method or synchronized block will get that lock, all other threads which are remaining to use the shared resources have to wait for the first thread to complete and the release of the lock.
- Synchronization can be achieved in the following ways:
  - Synchronized Method
  - Synchronized block
  - Static Synchronization

Java Synchronized Method

- If we use the Synchronized keywords in any method then that method is Synchronized Method.
- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won’t be released until the thread completes its function.

Java Synchronized Method

- **Syntax:**

Acess_modifiers synchronized return_type method_name (Parameters)
{
// Code of the Method.
}

Example: public **synchronized int count (**int pid**)**
{
// Code of the Method.
}

Example –Violate Synchronization

```
class counter{
private int count;
public counter(int count){
   this.count= count;
}
public  void increment(){
   this.count++;
}
public  void decrement(){
   this.count--;
}
public int getCount(){
   return count;
}
}
```

Example –Violate Synchronization

```java
class task1 extends Thread{
    counter obj;
    String name;
    task1(counter obj,String name){
        this.obj=obj;
        this.name=name;
        this.setName(name);
    }
    public void run(){
        for(int i=0;i<1000000;i++){
            obj.increment();
        }
    }
}
```

```java
class task2 extends Thread{
    counter obj;
    String name;
    task2(counter obj,String name){
        this.obj=obj;
        this.name=name;
        this.setName(name);
    }
    public void run(){
        for(int i=0;i<1000000;i++){
            obj.increment();
        }
    }
}
```

Example –Violate Synchronization

```
class Main {
   public static void main(String[] args)
         throws InterruptedException
   {
      // Instance of Counter Class
      counter c = new counter(0);
      // Defining Two different threads
      task1 first = new task1(c,"task1");
      task2 second = new task2(c, "task2");
      // Threads start executing
      first.start();
      second.start();
      // main thread will wait for child threads to complete execution
      first.join();
      second.join();
      System.out.println(c.getCount());
   }
}
```

//output run
1171846

//correct run must be
2000000

Example –with Synchronization method

```
class counter{
private int count;
public counter(int count){
    this.count= count;
}
public synchronized void increment(){
    this.count++;
}
public synchronized void decrement(){
    this.count--;
}
public int getCount(){
    return count;
}
}
```

```
//when re-run the program
2000000
```

Example –with Synchronization method with multiple objects

```java
class counter{
private int count;
public counter(int count){
   this.count= count;
}
public Synchronized void increment(){
                    this.count++;
          }
public Synchronized  void decrement(){
                    this.count--;
          public int getCount(){
   return count;
}
}
```

```
//Output run
2000000
2000000
```

```java
class Main {
  public static void main(String[] args)
       throws InterruptedException
  {
    counter obj1 = new counter(0);
    counter obj2 = new counter(0);
    task1 first = new task1(obj1,"task1f");
    task2 second = new task2(obj1, "task2s");
    task1 third = new task1(obj2,"task1td");
    task2 fourth = new task2(obj2, "task2fth");
    first.start();
    second.start();
    third.start();
    fourth.start();
    first.join();
    second.join();
    third.join();
    fourth.join();
    System.out.println(obj1.getCount());
    System.out.println(obj2.getCount());
  }
}
```

Synchronized Block

- Suppose you don’t want to synchronize the entire method, you want to synchronize few lines of code in the method, then a synchronized block helps to synchronize those few lines of code.
- It will take the object as a parameter. It will work the same as Synchronized Method.
- In the case of synchronized method lock accessed is on the method but in the case of synchronized block lock accessed is on the object.
- Syntax:

```
synchronized (object)
{ //code of the block. }
```

Example –with Synchronization Block

```
class counter{
private int count;
public counter(int count){
    this.count= count;
}
public void increment(){
            Synchronized(this){
             this.count++;
            }
}
public void decrement(){
             Synchronized(this){
             this.count--;
            }
public int getCount(){
    return count;
}
}
```

//when re-run the program
2000000

```java
class counter{
private int count;
public counter(int count){
   this.count= count;
}
public void increment(){
         Synchronized(this){
          this.count++;
         }
}
public void decrement(){
          Synchronized(this){
          this.count--;
         }
public int getCount(){
   return count;
}
}
```

Example –with Synchronization method with multiple objects

```
//Output run
2000000
2000000
```

```java
class Main {
  public static void main(String[] args)
      throws InterruptedException
  {
    counter obj1 = new counter(0);
    counter obj2 = new counter(0);
    task1 first = new task1(obj1,"task1f");
    task2 second = new task2(obj1, "task2s");
    task1 third = new task1(obj2,"task1td");
    task2 fourth = new task2(obj2, "task2fth");
    first.start();
    second.start();
    third.start();
    fourth.start();
    first.join();
    second.join();
    third.join();
    fourth.join();
    System.out.println(obj1.getCount());
    System.out.println(obj2.getCount());
  }
}
```

Static Synchronization

- Suppose in the case of where we have more than one object, in this case, two separate threads will acquire the locks and enter into a synchronized block or synchronized method with a separate lock for each object at the same time. To avoid this, we will use static synchronization.
- In this, we will place synchronized keywords before the static method. In static synchronization, lock access is on the class not on object and Method.
- Synchronized blocks can also be used inside of static methods.

Example –with Synchronizatior method with static members

```
class counter{
private static int count;
public counter(int count){
   this.count= count;
}
public Synchronized  void increment(){
                    this.count++;
          }
public Synchronized void decrement(){
                    this.count--;
          public int getCount(){
   return count;
}
}
```

```
//output run
3593665
3593665

//correct output must be
4000000
4000000
```

```
class Main {
  public static void main(String[] args)
       throws InterruptedException
  {
    counter obj1 = new counter(0);
    counter obj2 = new counter(0);
    task1 first = new task1(obj1,"task1f");
    task2 second = new task2(obj1, "task2s");
    task1 third = new task1(obj2,"task1td");
    task2 fourth = new task2(obj2, "task2fth");
    first.start();
    second.start();
    third.start();
    fourth.start();
    first.join();
    second.join();
    third.join();
    fourth.join();
    System.out.println(obj1.getCount());
    System.out.println(obj2.getCount());
  }
}
```

Example –with Synchronizatior method with static members

```
class counter{
private static int count;
public counter(int count){
   this.count= count;
}
public static Synchronized  void increment(){
                     this.count++;
          }
public static Synchronized void decrement(){
                     this.count--;
          public int getCount(){
   return count;
}
}
```

```
//running output
4000000
4000000
```

```
class Main {
   public static void main(String[] args)
        throws InterruptedException
   {
      counter obj1 = new counter(0);
      counter obj2 = new counter(0);
      task1 first = new task1(obj1,"task1f");
      task2 second = new task2(obj1, "task2s");
      task1 third = new task1(obj2,"task1td");
      task2 fourth = new task2(obj2, "task2fth");
      first.start();
      second.start();
      third.start();
      fourth.start();
      first.join();
      second.join();
      third.join();
      fourth.join();
      System.out.println(obj1.getCount());
      System.out.println(obj2.getCount());
   }
}
```

Synchronized and Data Visibility

- The synchronized keyword changes that. When a thread enters a synchronized block it will refresh the values of all variables visible to the thread.
- When a thread exits a synchronized block all changes to variables visible to the thread will be committed to main memory.
- This is similar to how the volatile keyword works.

Synchronized and Instruction Reordering

- the Java synchronized keyword places some restrictions on the reordering of instructions before, inside, and after synchronized blocks.
- This is similar to the restrictions placed by the **volatile keyword**.

Synchronized Block Limitations and Alternatives

- What if you want to allow N threads to enter a synchronized block, and not just one? You could use a **Semaphore** to achieve that behavior.
- Synchronized blocks do not guarantee in what order threads waiting to enter them are granted access to the synchronized block.
- What if you need to guarantee that threads trying to enter a synchronized block get access in the exact sequence they requested access to it? You need to implement **Fairness** yourself.

Monitor in Java - Cooperation

Monitor in Java - Cooperation

- **wait():**
- Call to this method causes the current thread to wait until another thread invokes the notify() or the notifyAll() method for this object. The current thread must own this object's monitor(lock). The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify or the notifyAll method.
  This method should only be called by a thread that is the owner of this object's monitor.

Monitor in Java - Cooperation

- **notify():**
- Call to this method wakes up a single thread that is waiting on this object’s monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
- *This method should only be called by a thread that is the owner of this object’s monitor*.

.

Monitor in Java - Cooperation

- **notifyAll():**
- Call to this method wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The awakened threads will not be able to proceed until the current thread relinquishes(voluntarily releases) the lock on this object.
  This method should only be called by a thread that is the owner of this object's monitor

ReaderWriter problem Without Monitor Cooperation

```
class Message {
    String message;
    boolean empty = true;
    //Method used by reader
    public synchronized String read() {
        while (empty) ;//if message is empty then keep looping.
        empty = true;//Reader reads the message and marks empty as true.
        return message;//Reader reads the message.
    }
    //Method used by writer
    public synchronized void write(String message) {
        while (!empty) ;//if message is not empty then keep looping.
        this.message = message;//Writer writes the message.
        empty = false;//Now make empty as false.
    }
}
```

ReaderWriter problem Without Monitor Cooperation

```
class Reader implements Runnable {
    private Message message;
    public Reader(Message message) {
        this.message = message;
    }
    @Override
    public void run() {
        String latestMessage;
        do{
            latestMessage= message.read();
            System.out.println(latestMessage);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Reader Thread Interrupted!!!");
            }
        }while(!"Finished!".equals(latestMessage));
    }
}
```

ReaderWriter problem Without Monitor Cooperation

```java
class Writer implements Runnable {
    private Message message;
    public Writer(Message message) {
        this.message = message;
    }
    @Override
    public void run() {
        String messages[] = {"hello fcih", "mid next week", "MCQ question","study well" };
        for (int i = 0; i < messages.length; i++) {
            message.write(messages[i]);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println("Writer Thread Interrupted!!!");
            }
        }
        message.write("Finished!");
    }
}
```

ReaderWriter problem Without Monitor Cooperation

```java
public class Main {
    public static void main(String[] args) {
        //Shared message object between Reader and Writer threads.
        Message message = new Message();

        Thread writerThread = new Thread(new Writer(message));
        Thread readerThread = new Thread(new Reader(message));

        writerThread.start();
        readerThread.start();
    }
}
```

This scenario is called a **Deadlock**.

When we started the threads from the Main class, both the threads called the **run()** method.
Note that both threads are sharing a common message object. Now the Reader thread called the synchronized **read()** method and hence acquired the lock of the message object.
As initially the boolean **empty** flag was set to **true**, the Reader thread keeps executing while loop infinitely.
Also, the Writer thread won't be able to execute the **write()** method as the lock of the **message** object is already acquired by the Reader thread.

//another output run
hello fcih
mid next week
//loop forever

//another output run
hello fcih
mid next week
//loop forever

//another output run
//loop for ever

Solution for ReaderWriter problem with Monitor Cooperation

- Simply when the Writer thread writing a message add the wait() to read method until the writer finishes writing and release the lock and wake up the Reader thread using notify() or notifyall()
- Also, when the reader has the lock and reads the message, simply add the wait() to write method until the Reader finishes reading and releases the lock and notifies the writer thread using notify() or notifyall()

```java
class Message{
  String message;
  boolean empty = true;
  public synchronized String read() {
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {
        System.out.println(Thread.currentThread().getName() + "Interrupted.");
      }
    }
    empty = true;
    notifyAll();
    return message;
  }
 public synchronized void write(String message) {
    while (!empty) {
      try {
        wait();
      } catch (InterruptedException e) {
        System.out.println(Thread.currentThread().getName() + "Interrupted.");
      }
    }
    this.message = message;
    empty = false;
    notifyAll();
  }}
```

Reader thread waits until Writer invokes the notify()method or the notifyAll() method for 'message' object. Reader thread releases ownership of lock and waits until Writer thread notifies Reader thread waiting on this object's lock to wake up either through a call to the notify method or the notifyAll method.

Wakes up all threads that are waiting on 'message' object's monitor(lock). This thread(Reader) releases the lock for 'message' object.

Writer thread waits until Reader invokes the notifyAll() method for 'message' object. Writer thread releases ownership of lock and waits until Reader thread notifies Writer thread waiting on his object's lock to wake up

Wakes up all threads that are waiting on 'message' object's monitor(lock). This thread(Writer) releases the lock for 'message' object.

What Is Deadlock

- A deadlock occurs when two or more threads wait forever for a lock or resource held by another of the threads.
- An application may stall or fail as the deadlocked threads cannot progress.

Deadlock Example

- First, let's take a look at a simple Java example to understand deadlock.
- In this example, we'll create two threads, thread_one and thread_two.
- Thread thread_one calls operation1, thread and thread_two calls operation2.
- To complete their operations, thread thread_one needs to acquire first_mutex first and then second_mutex, whereas thread thread_two needs to acquire second_mutex first and then first_mutex.
- So, basically, both threads are trying to acquire the locks in the opposite order.

Deadlock Example

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Main {
    private Lock first_mutex = new ReentrantLock(true);
    private Lock second_mutex = new ReentrantLock(true);
    public static void main(String[] args) throws InterruptedException {
        Main deadlock = new Main();
        Thread thread_one=new Thread(deadlock::operation1, "thread_one");
        Thread thread_two=new Thread(deadlock::operation2, "thread_two");
        thread_one.start();
        thread_two.start();
        thread_one.join();
        thread_two.join();
    }
```

The double colon (::) operator, also known as method reference operator in Java, is used to call a method by referring to it with the help of its class directly. They behave exactly as lambda expressions.

```
//using lambda expression
//Note method operation1() must be static as it is called inside main()
Thread thread_one2=new Thread(()->{operation1();}, "thread_one");
```

Deadlock Example

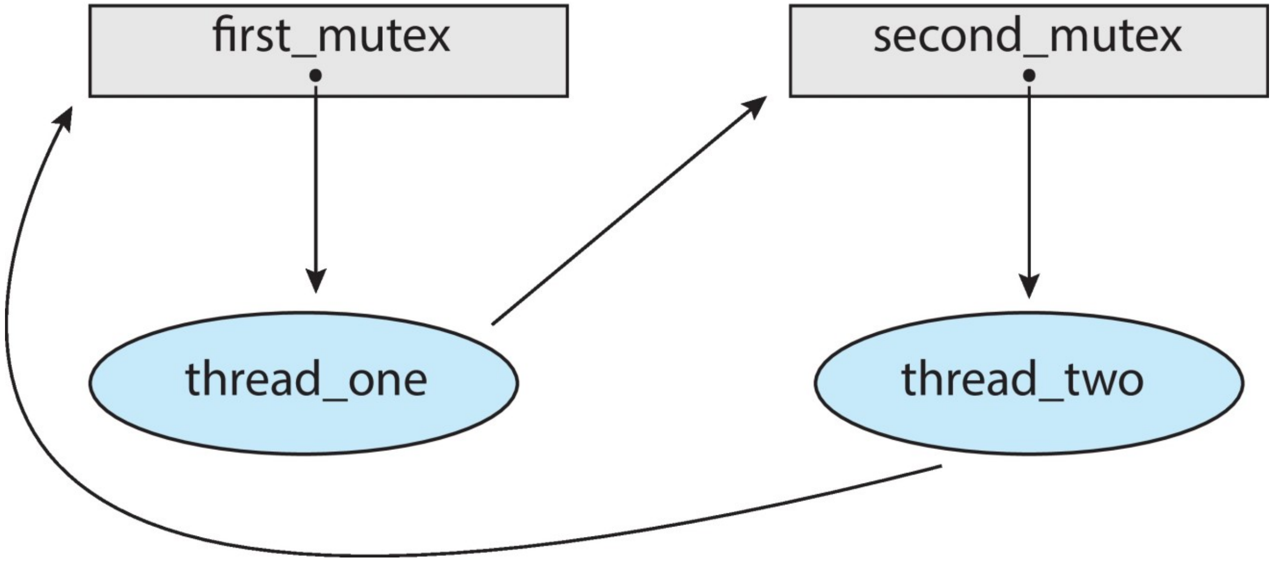
```
public void operation1() {
    first_mutex.lock();
    System.out.println("first_mutex acquired by Thread:"+Thread.currentThread().getName()+"
                        , waiting to acquire second_mutex.");
    second_mutex.lock();
    System.out.println("second_mutex acquired");

    try {
        System.out.println("executing first operation.");
        Thread.sleep(50);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    second_mutex.unlock();
    first_mutex.unlock();
}
```

Deadlock Example

```
public void operation2() {
        second_mutex.lock();
        System.out.println("second_mutex acquired by Thread: "+Thread.currentThread().getName()+
                                        " , waiting to acquire first_mutex.");
        first_mutex.lock();
        System.out.println("first_mutex acquired");
        try {
            Thread.sleep(50);
            System.out.println("executing second operation.");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        first_mutex.unlock();
        second_mutex.unlock();
    }
}
```

Deadlock in Multithreaded Application

- Deadlock is possible if thread_one acquires **`first_mutex`** and thread_two acquires **`second_mutex`**.
- Thread_one then waits for **`second_mutex`** and thread_two waits for **`first_mutex`**.
- Can be shown with a **resource allocation graph**:



// running output
second_mutex acquired by Thread: thread_two , waiting to acquire first_mutex.
first_mutex acquired by Thread:thread_one , waiting to acquire second_mutex.

Notes about **`ReentrantLock`**

- As per Javadoc, `ReentrantLock` is a mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended features such as fairness
- ***synchronized*** Does not provide any fair locks in java.
- In ***synchronized the*** acquiring lock and releasing is implicit.
- **`ReentrantLock`** acquiring lock and releasing is explicit

ReentrantLock

- **provides fair locks**, when lock is fair - first lock is obtained by longest-waiting thread in java.
- Constructor to provide fairness - ***ReentrantLock****(boolean* ***fair****)*
- Creates an instance of ReentrantLock.
- When ***fair*** is set true, first lock is obtained by longest-waiting thread.
- If ***fair*** is set false, any waiting thread could get lock

ReentrantLock

- **Provide tryLock() method. If lock is held by another thread then method return false in java.**
  boolean tryLock()
- Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.
- If current thread already holds lock then method returns true. And increments **lock hold count** by 1.
- If lock is held by another thread then method return false.

ReentrantLock

- provide *int getQueueLength()* method to return number of threads that may be waiting to acquire this lock in java.
- *isHeldByCurrentThread()* method is **used to find out whether lock is held by current thread or not**. If current thread holds lock method returns true in java.
- provides *newCondition()* method.
- Method returns a Condition instance to be used with this Lock instance.
- Condition instance are similar to using **Wait(), notify() and notifyAll()** methods on object.

What Is Livelock

- Livelock is another concurrency problem and is similar to deadlock.
- In livelock, two or more threads keep on transferring states between one another instead of waiting infinitely as we saw in the deadlock example.
- Whereas deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails.

Livelock Example

- Now, to demonstrate the livelock condition, we'll take the same deadlock example we've discussed earlier
- However, we'll change the logic of these operations slightly.
- Both threads need two locks to complete their work. Each thread acquires its first lock but finds that the second lock is not available. So, in order to let the other thread complete first, each thread releases its first lock and tries to acquire both the locks again.
- the tryLock method in the Lock interface, to make sure that a thread does not block infinitely if it is unable to acquire a lock.

Livelock Example

```
public void operation1() {
    while (true) {
        try {
            first_mutex.tryLock(50, MILLISECONDS);
            System.out.println("first_mutex acquired by Thread:" + Thread.currentThread().getName() +
                               " , waiting to acquire second_mutex.");
            Thread.sleep(50);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        if (second_mutex.tryLock())
            System.out.println("second_mutex acquired");
        else {
            System.out.println("can not acquire second_mutex, releasing first_lock");
            first_mutex.unlock();
            continue;
        }
        System.out.println("executing first operation.");
        break;
    }
        second_mutex.unlock();
        first_mutex.unlock();
    }
```

Livelock Example

```
public void operation2() {
      while (true) {
         try {
            second_mutex.tryLock(50, MILLISECONDS);
            System.out.println("second_mutex acquired by Thread:" + Thread.currentThread().getName() +\
                              " , waiting to acquire first_mutex.");
            Thread.sleep(50);
         } catch (InterruptedException e) {
            throw new RuntimeException(e);
         }
         if (first_mutex.tryLock())
            System.out.println("first_mutex acquired");
         else {
            System.out.println("can not acquire first_mutex, releasing first_mutex");
            second_mutex.unlock();
            continue;
         }
         System.out.println("executing second operation.");
         break;
      }
      first_mutex.unlock();
      second_mutex.unlock();
   }
}
```

Livelock Example

As we can see in the output, both threads are repeating the acquiring and releasing locks. Because of this, none of the threads are able to complete the operation.

```
first_mutex acquired by Thread:thread_one , waiting to acquire second_mutex.
second_mutex acquired by Thread:thread_two , waiting to acquire second_mutex.
can not acquire second_mutex, releasing first_lock
can not acquire first_mutex, releasing second_mutex
first_mutex acquired by Thread:thread_one , waiting to acquire second_mutex.
//will infinitely repeating printing the message
```