

Threads, Part II

Thread Synchronisation
CS3524 Distributed Systems
Lecture 05

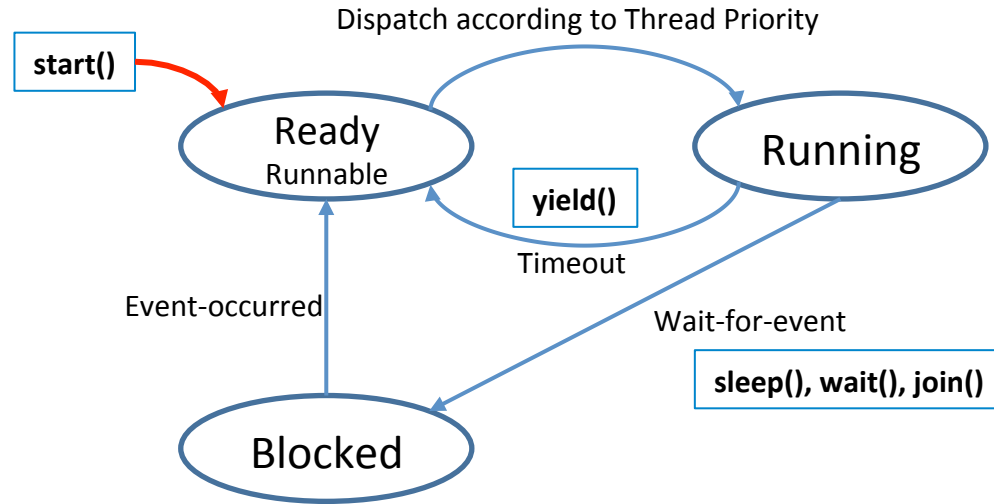
Threads

- Multi-threaded execution is an essential feature of Java
- A thread is a single sequential flow of control within a program
- Threads in Java exist within a process - the Java Virtual Machine
- Many threads may execute concurrently
- All threads share resources owned by this execution environment

Java Thread Life-Cycle

- The Java thread objects provide a number of methods to control the life-cycle of threads:
 - **run()** – implements the code to be executed by the thread
 - **start()** – starts a thread
 - **sleep()** – a thread will wait for the specified amount of time
 - **join()** – can be used to synchronize execution between threads, one thread will wait for another thread to terminate before continuing execution
 - **yield()** – causes the thread to give up the processor and allow other threads to execute

Thread Scheduling



- Calling `start()` will make thread “**runnable**”, thread is put into Ready queue
- Threads are dispatched according to priority
- Thread can give up processor with `yield()`
- Thread can put itself into Blocked state
 - If thread calls `sleep()`, `wait()`, `join()` – it will be transferred to the Blocked queue

Race Conditions and the Need for Mutual Exclusion

Race Conditions

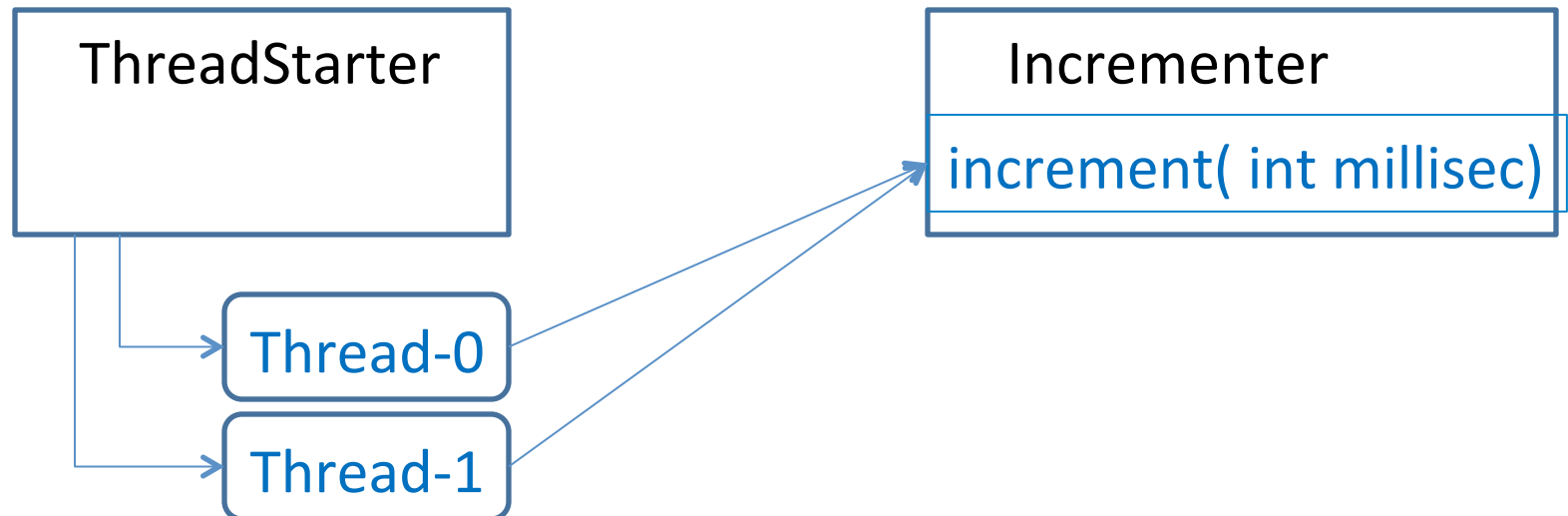
- Multiple threads run within the context of one process
 - Remember: they are called lightweight processes
 - They share a common *memory space*, but
 - They have their own *local variables* (“working memory”)
- **Danger of race conditions**: they have to compete for resources within their execution context
 - e.g. access to global variables
- Therefore: *synchronisation* of thread activity becomes necessary

Critical Sections and Mutual Exclusion

- Threads (in most cases) interfere with each other
 - they access shared resources concurrently
- **Critical Section**: a particular section of code, specified in a Java class, that may be executed by multiple threads concurrently
 - this can be a method called by multiple threads concurrently
- **Goal: Mutual Exclusion** between threads
 - only one thread at a time should be able to execute such a section **in its completeness**

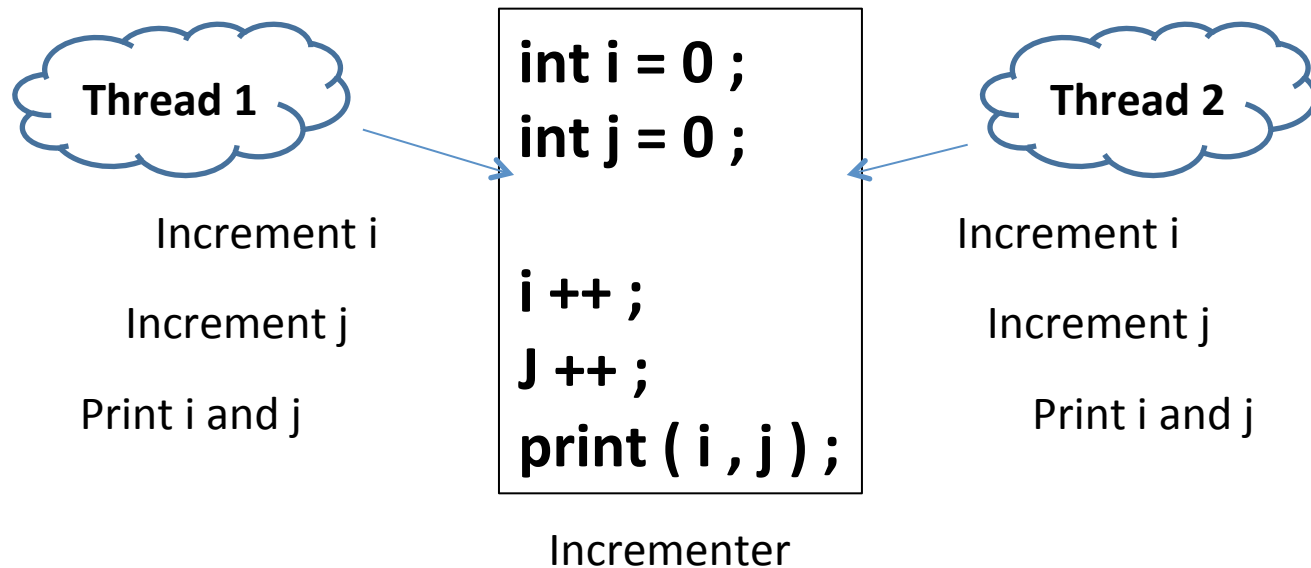
Example Scenario

- Two classes
 - ThreadStarter: we instantiate two threads from this class
 - Incrementer: provides a method `increment()` that is called by both threads



Example:

- What is the result of this computation in thread 1?
- What is the result of this computation in thread 2?



Thread Concurrency

```
public class Incrementer
{
    private int i = 0 ;
    private int j = 0 ;
    public void increment ( int milliseconds )
    {
        i ++ ;
        pause ( milliseconds ) ;
        j ++ ;
        System.out.println ( "Method increment(): " +
                               ((i == j) ? "Same" : "Different") ) ;
    }

    private void pause( int p ) {
        try { Thread.currentThread().sleep( p ) ;
        } catch ( Exception e ) {}
    }
}
```

- What happens to the respective values of i and j ?

Thread Concurrency

```
public class ThreadStarter extends Thread
{
    // NOTE: Incrementer is a shared resource
    static private Incrementer incrementer = new Incrementer() ;

    public ThreadStarter ( ) {}

    public void run () {
        incrementer.increment( 1000 ) ;
    }

    static public void main ( String args[] ) {
        ThreadStarter t1 = new ThreadStarter() ;
        ThreadStarter t2 = new ThreadStarter() ;
        t1.start() ;
        t2.start() ;
    }
}
```

- The incrementer is known to both threads

Thread Concurrency

- One particular run:
 - We cannot predict, which thread is scheduled first
 - Both interfere with each other in the incrementing of the variables *i* and *j*

After incrementing *i*, Current thread: Thread-1, *i* = 1

After incrementing *i*, Current thread: Thread-0, *i* = 2

After incrementing *j*, Current thread: Thread-1, *j* = 1

Method increment(), printed from thread Thread-1: Different

After incrementing *j*, Current thread: Thread-0, *j* = 2

Method increment(), printed from thread Thread-0: Same

Race Condition

- A ***race condition*** is an undesirable situation that occurs when two or more threads attempt to manipulate a shared resource concurrently
 - Two or more threads are able to access shared data and they try to make changes at the same time
- The result of such a computation depends on the sequence of how these threads are scheduled over time
 - A different schedule may result in a completely different interleaving of actions of the different threads and in different results
- “the threads are ‘racing’ against each other to win access to the shared resource”
 - E.g.: multiple read and write operations by different threads may result in one thread overwriting / removing a computation of another thread.

Race Condition

- Often occurs when one thread does a “check-then-act”
 - Check: First, read the shared data and check a condition of the read data
 - Act: if the checked condition hold, manipulate the shared data
- Between these two actions, another thread may be scheduled and perform the same sequence of actions, manipulating the shared data first

Atomicity and Mutual Exclusion

- We have to safeguard the performance of these actions by one thread
 - “Atomicity”: only one thread is allowed to execute a section of code (“critical section”) in its completeness (“all-or-nothing”)
- How can we guarantee this “mutual exclusive” access:
 - “Serialisation of access”: a thread first has to acquire a “lock”, before it can enter a critical section of code and execute it
 - Only one thread may proceed, all other threads have to wait

Monitor in Java

Mutual Exclusive Access to Shared
Resources

Monitor

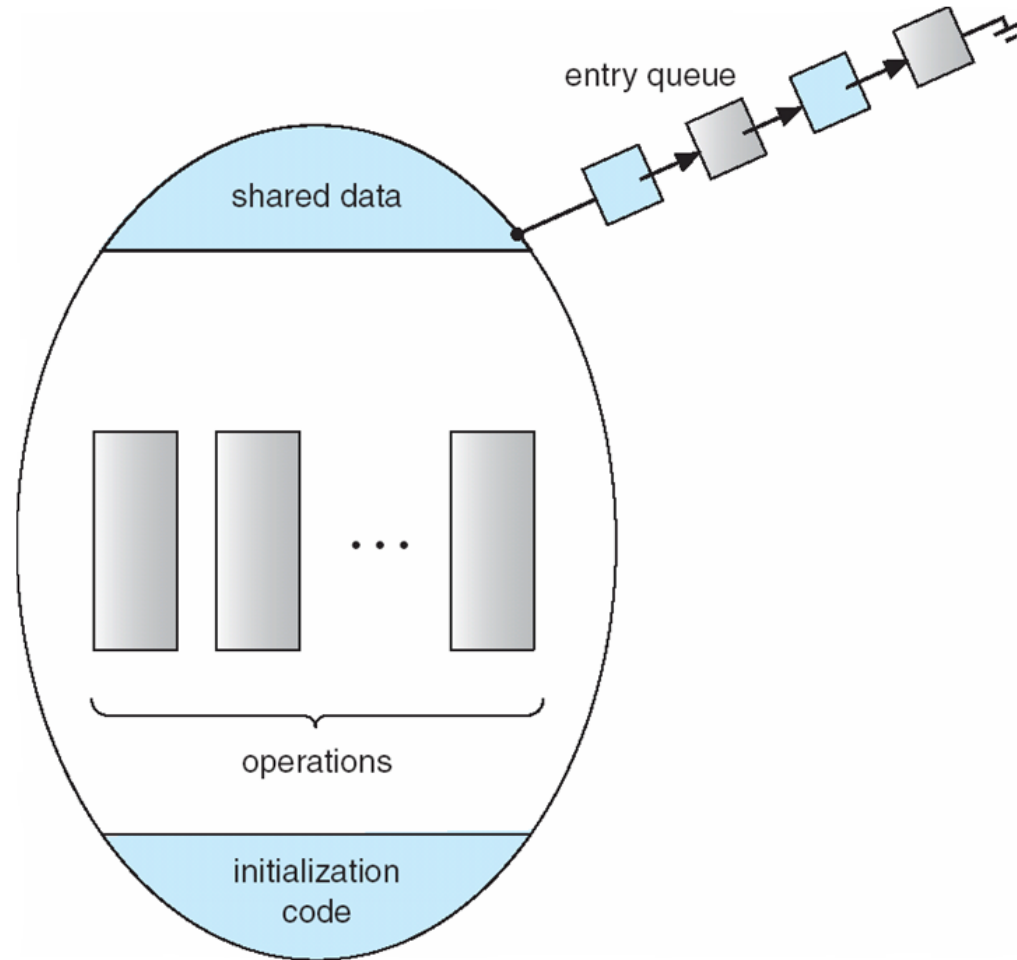
- We need a mechanism to *coordinate and synchronise* thread activity. Java provides simple synchronisation mechanisms based on C. A. R. Hoare's *monitor* concept
- A monitor can be thought of as a *barrier* securing a *shared resource* with a **lock** :
 - If the resource is not used by another thread, a thread can *acquire* the lock and access the resource – the thread is regarded as “entering the monitor”
 - Other threads wanting to access the resource must wait (in a queue) until the lock is *released*

Monitor

- A monitor is a software construct that serves two purposes:
 - enforces mutual exclusion of concurrent access to shared data objects
 - Processes have to acquire a lock to access such a shared resource
 - Support conditional synchronisation between processes accessing shared data
 - Multiple processes may use monitor-specific wait()/signal() mechanisms to wait for particular conditions to hold

Monitor: Entry

- A monitor has an entry queue
- Processes calling monitor procedures may be added to waiting queue and suspended



Mutual Exclusion

- When a process calls one of these procedures, it “enters” the monitor
 - The process has to acquire a monitor lock first
- The monitor guarantees that
 - Only one process at a time may call one of these procedures and “enter” the monitor
 - All other processes have to wait

```
monitor SharedBuffer  
  buffer b[N] ;  
  int in, out ;  
  
  procedure append()  
  ...  
  end;  
  
  procedure take()  
  ...  
  end;  
end monitor;
```

Monitor in Java

- In Java, a monitor is represented simply by a Java class specification, if the keyword **synchronized** is used
- The keyword **synchronized** declares code within the Java class as critical sections, this can be
 - a **synchronized** method or
 - a **synchronized** block or
 - a **static synchronized** block
- If the keyword **synchronized** is used, the Java runtime environment will add functionality to manage the monitor locking mechanisms

Critical Section in Java

- The synchronized keyword indicates where a thread must acquire the object's lock
 - When a thread is scheduled for execution and enters a section of code that is enclosed with a **synchronized** construct, it has to acquire first a monitor lock on this object
 - If another thread already holds this lock, our currently scheduled thread has to wait
- Please note: the whole object is locked for a particular thread!

The Java **synchronized** construct

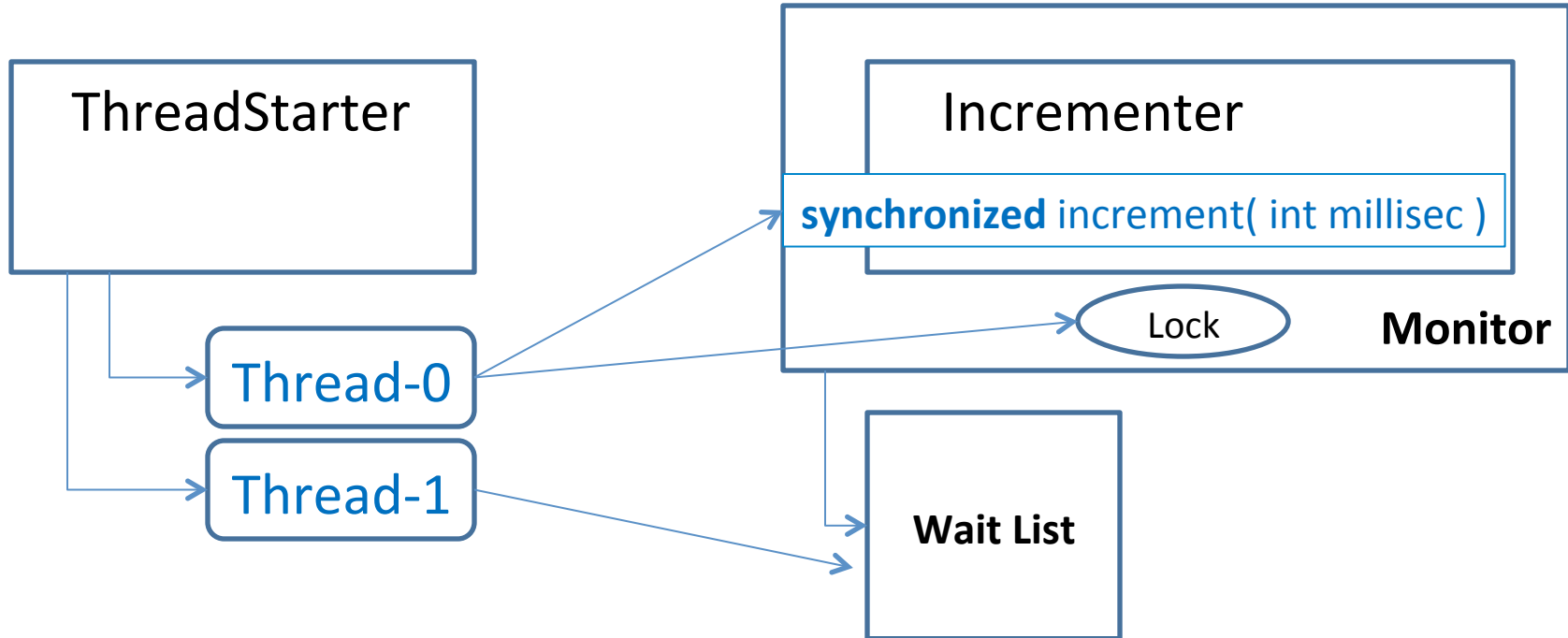
- Synchronized **statement/block**
 - Attempts to perform a lock action on the given object, the body of the synchronized statement is executed, when this lock is acquired

```
synchronized ( myObject )
{
    // critical section, myObject must be a reference to
    // an object
    incrementer.increment( 1000 ) ;
}
```

- Synchronized **methods**
 - Automatically performs a lock action on the method's object when it is invoked
 - Allows to declare methods themselves as critical sections

```
public synchronized increment ( int milliseconds )
{
    // critical section
}
```

Monitor



- A thread gains access to the shared resource by acquiring its lock
- Other threads wait in the wait list for the lock to become available


Synchronized Blocks

```
public class ThreadStarter extends Thread
{
    static private Incrementer incrementer = new Incrementer() ;

    public ThreadStarter ( ) {}

    public void run () {
        // we either use a synchronized block . . .
        { synchronized ( incrementer ) {
            incrementer.increment( 1000 ) ;
        }
    }

    static public void main ( String args[] ) {
        ThreadStarter t1 = new ThreadStarter() ;
        ThreadStarter t2 = new ThreadStarter() ;
        t1.start() ;
        t2.start() ;
    }
}
```



Synchronized Methods

```
public class Incrementer
{
    private int i = 0 ;
    private int j = 0 ;
    // . . . or we use a synchronized method
    public void synchronized increment ( int milliseconds )
    {
        i ++ ;
        pause ( milliseconds ) ;
        j ++ ;
        System.out.println ( "Method increment(): " +
                               ((i == j) ? "Same" : "Different") ) ;
    }

    private void pause( int p ) {
        try { Thread.currentThread().sleep( p ) ;
        } catch ( Exception e ) {}
    }
}
```

Thread Concurrency

Using synchronized Method

- One particular run:
 - Due to synchronization, there is no interference between the threads during the incrementation of the variables *i* and *j*

```
After incrementing i, Current thread: Thread-0, i = 1
After incrementing j, Current thread: Thread-0, j = 1
Method increment(), printed from thread Thread-0: Same
After incrementing i, Current thread: Thread-1, i = 2
After incrementing j, Current thread: Thread-1, j = 2
Method increment(), printed from thread Thread-1: Same
```

Thread Synchronisation

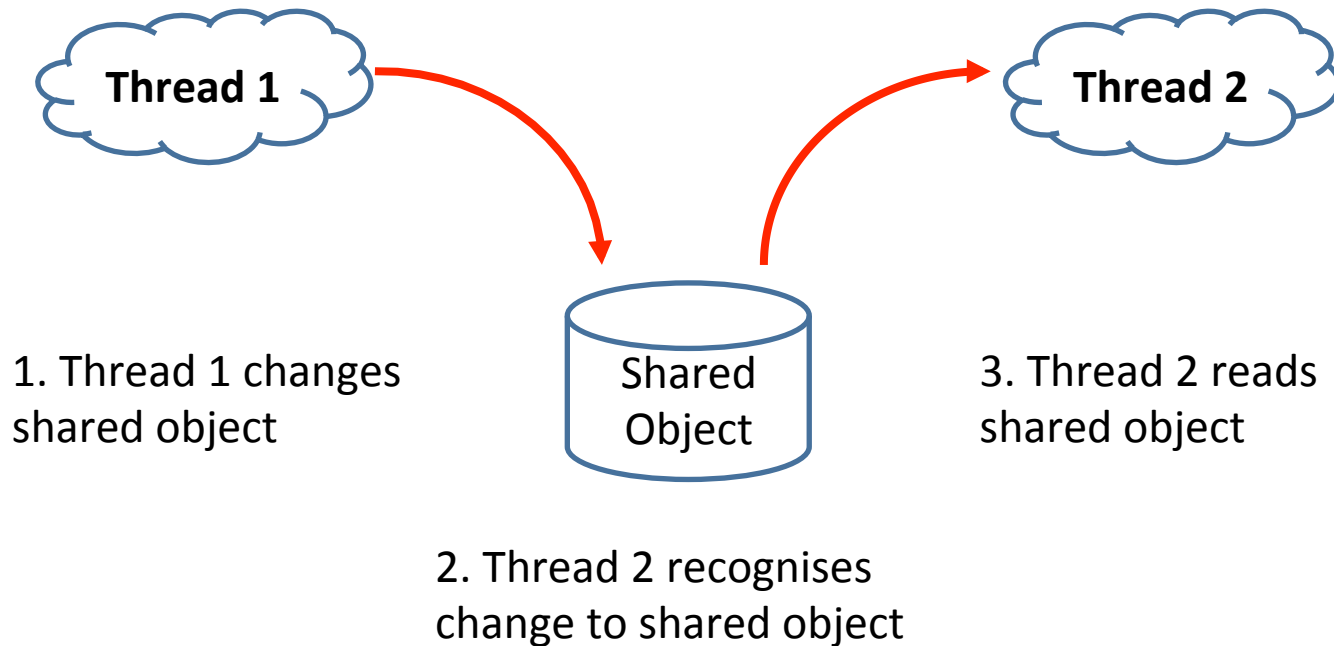
Observer Pattern

Synchronisation between Producer and Consumer

- There are cases where one thread produces data that should be picked up by another thread – they are in a producer-consumer relationship:
 - E.g.: a producer writes information into a shared buffer, if it is empty
 - When the buffer is full, the producer will indicate this fact to a consumer
 - The consumer waits for the shared buffer to be filled
 - it waits for the producer to indicate that the buffer is full
 - If this buffer is full, then the consumer reads the buffer and empties it, it will indicate to the producer that the buffer is empty

Threads as Producer and Consumer

- We want to use shared data objects for communication

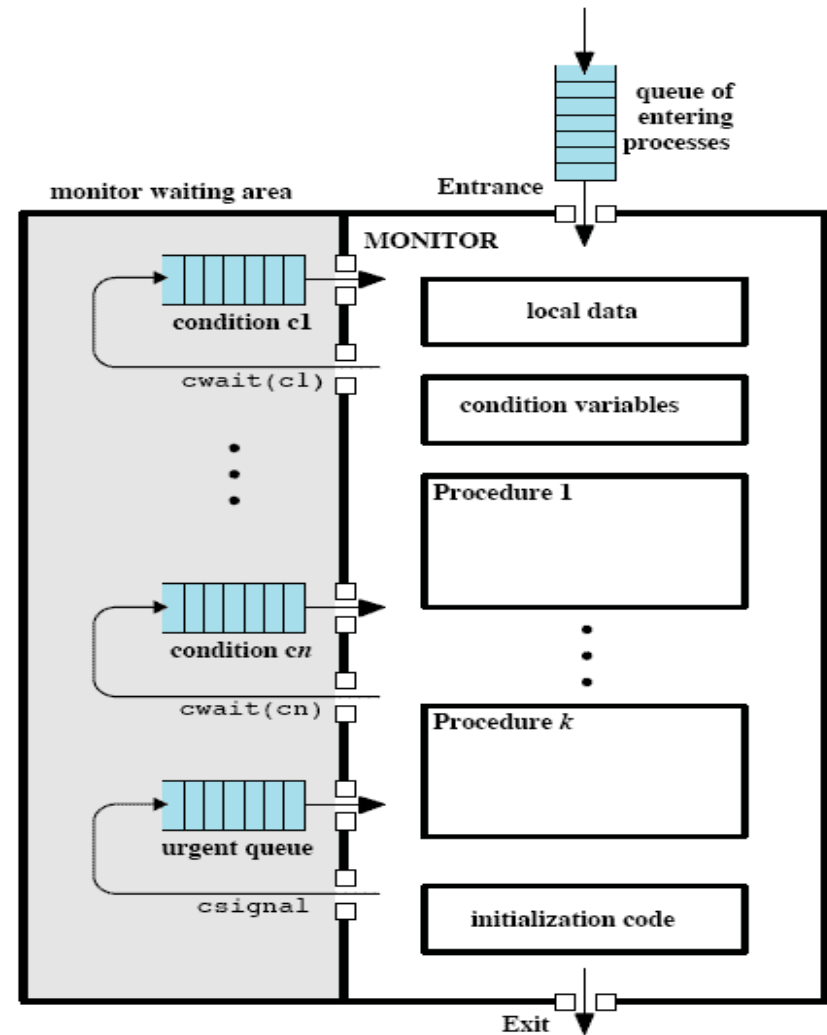


Communication between Threads

- Questions:
 - How does thread 1 know that it can change the shared object without interference with thread 2?
 - Is thread 2 / any other thread currently manipulating the shared data object?
 - How can thread 1 reserve exclusive access to the shared data object to avoid this interference?
 - How does thread 2 realise that the shared data object has changed?
 - How can thread 1 inform thread 2 that such a change occurred?

Monitor: Process Synchronisation with Condition Variables

- Signalling mechanism:
 - Process may call *wait(condition variable)*
 - made to wait for a condition in a condition queue
 - Process may call *signal(condition variable)*
 - This resumes one of the processes waiting for this conditional signal



Monitor: Process Synchronisation with Condition Variables, use Notify()

- The signal() is replaced by a notify()
 - When a signalling process issues the notify(c) for condition queue c, it continues to execute
 - It notifies the queue because a particular condition currently holds, e.g. “buffer is not empty any more”
 - The next process to be executed from the condition queue c will be rescheduled when the monitor becomes available
 - Rescheduled process has to **recheck condition**
 - E.g. “is buffer still not empty?”
 - This is necessary, because another process could be scheduled before and interfere
- Works without atomicity of notify() and resumption of processes

Monitors with notify() and Broadcast

- Note the “while” loop:
 - Rechecking the condition after wakeup
- Allows for non-atomicity between notify() and wakeup of other thread

```
monitor boundedBuffer
{
    char b[N]; int count, in, out ;
    condition notfull, notempty;

    void append(char item) {
        while (count == N) wait(notfull);
        b[in] = item;
        in = (in+1) % N;
        count++;
        notify(notempty);
    }

    char take() {
        while (count == 0) wait(notempty);
        item = b[out];
        out = (out+1) % N;
        count--;
        notify(notfull);
    }
}
```

Java Implementation of Condition Synchronisation: Observer Pattern

Java implementation

Observer Pattern

- Class `java.lang.Object` provides the following methods (from the Observer Pattern):
 - `wait()`
 - `notify()`
 - `notifyAll()`
- These methods support an efficient transfer of control from one thread to other threads
- These methods can be used to handle the competition of threads for entering a monitor / shared object with a lock

Wait Sets and Notification

The Observer Pattern

- Wait Set:
 - Each shared object with synchronized code elements, beside having an associated lock, also has a “wait set”, which is a set of threads waiting for the object to be unlocked
- When the object is first created, its wait set is empty
- Notification:
 - The methods `wait()`, `notify()` and `notifyAll()` should only be invoked on a particular object when the invoking thread (the “current thread”) has already locked this object
 - `notify()` and `notifyAll()` are used to notify waiting threads that they can enter a critical section

Wait Sets and Notification

The Observer Pattern

- Method `notify()`
 - We assume there is a “current thread” using a shared object, which issues a “`notify()`”
 - If the wait set of the locked object contains waiting threads then one of them (arbitrarily chosen) is taken out of the wait set and re-enabled for thread scheduling
 - The re-enabled thread will now wait until the “current thread” gives up the lock on the object
- Method `notifyAll()`
 - If the wait set of the locked object contains waiting threads then all of them will be taken out of the wait set and re-enabled for thread scheduling
 - The re-enabled threads will now wait until the “current thread” gives up the lock on the object

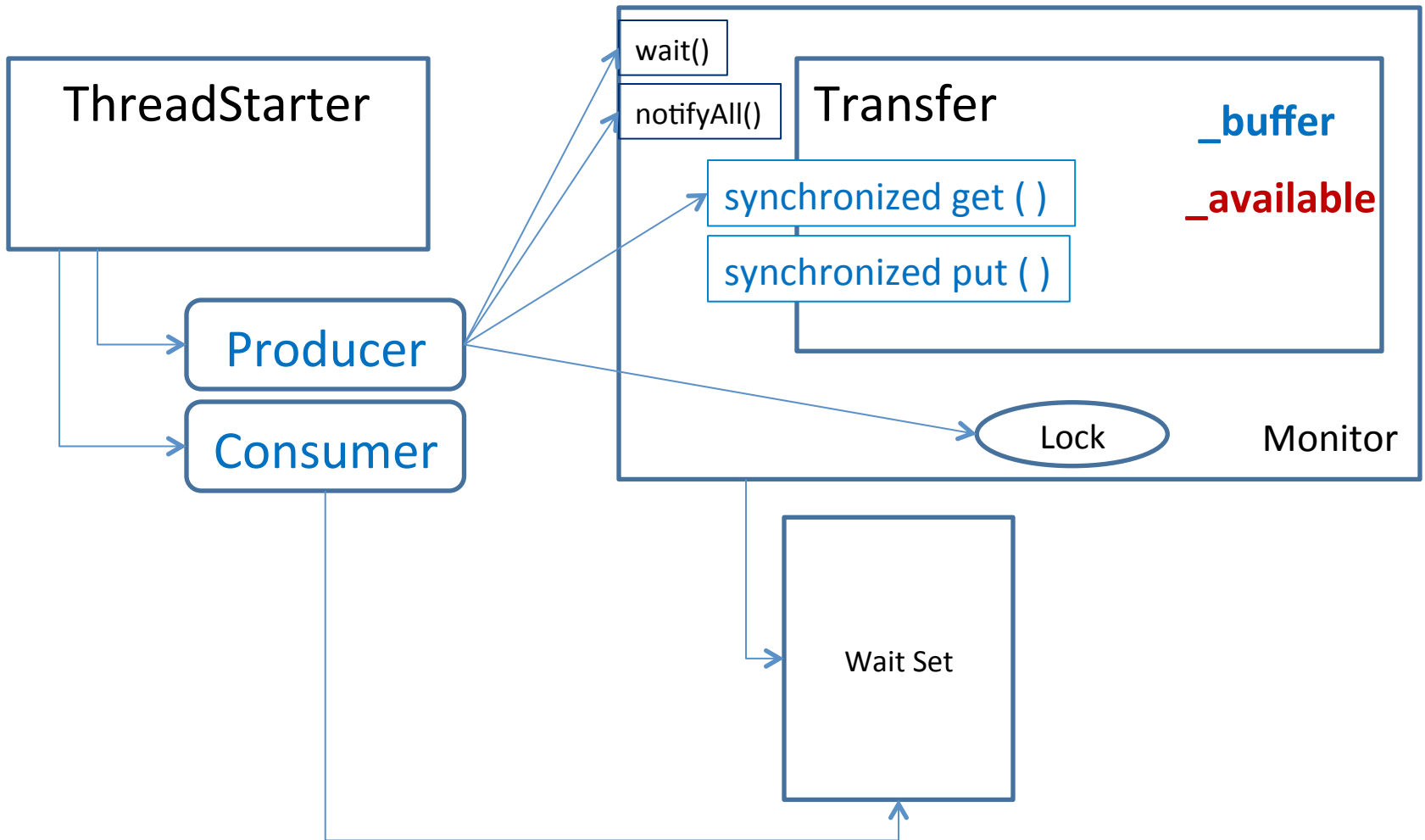
Wait Sets and Notification

The Observer Pattern

- Method wait()
 - Adds the “current thread” (the one who invokes the wait() on the locked object) to the wait set of the locked object, disables the thread for being scheduled, and removes all locks held by the “current thread”
 - The “current thread” will wait now in the wait set until one of the following happens:
 - Some other thread, which now has locked the object, invokes the notify() method and our (previously the “current”) now waiting thread is chosen by chance (and re-enabled for scheduling)
 - Some other thread, which now has locked the object, invokes the notifyAll() method, which empties the complete wait set

Producer and Consumer

Using the Observer Pattern



Synchronisation Example

Producer & Consumer

```
public class Transfer
{
    private Object _buffer;
    private boolean _available = false;


    public synchronized Object get()
    {
        // Consumer acquires lock, we want the consumer
        // to wait until the producer puts something into
        // buffer, producer must notify when done
        while ( _available == false ) {
            try {
                wait(); // give up lock to give producer access
            }
            catch ( InterruptedException e ) { }
        }
        _available = false; // Consumer indicates that buffer is
                           // consumed (empty)
        notifyAll();        // Consumer notifies Producer
        return _buffer;
    } // Consumer gives up lock
}
```

critical section

Tells us that this is a blocking method
(puts thread in blocked queue) that
waits for an external event

Synchronisation Example:

Producer & Consumer



```
public synchronized void put ( Object element )
{
    // Producer acquires lock, must wait until
    // consumer takes a value, consumer must notify

    while ( _available == true )
    {
        try
        {
            wait(); // give up lock to give consumer access
        }
        catch ( InterruptedException e ) { }
    }

    _buffer = element; // Producer fills buffer
    _available = true; // Producer indicates that buffer
                        // is full
    notifyAll(); // Producer notifies Consumer

} // Producer releases lock
}
```

Synchronisation Example:

Producer & Consumer

- There are two synchronized methods: `get()` and `put()`
- The thread reading the buffer should not attempt to get elements from an empty buffer
- Therefore, the Consumer thread is told to wait in the `get()` method (it invokes `wait()`), until the Producer sends a notification that an element is available
- The `notifyAll()` method wakes up all threads waiting on the the object in question
- Awakened threads compete for the lock on this object
- One thread gets the lock, all the others (if there are any) go back to waiting

Synchronisation Example:

Guarded Blocks

- These are guarded blocks – a guard condition is tested in a while loop by the thread until it becomes satisfied and the thread proceeds
- In addition, they contain a call to wait() so that the thread goes into a wait state (otherwise we would loop until the guard condition holds and consume CPU cycles)

```
while ( _available == false ) {  
    try {  
        wait(); // give up lock to give producer access  
    }  
    catch ( InterruptedException e ) { }  
}
```

Consumer

```
while ( _available == true ) {  
    try {  
        wait(); // give up lock to give consumer access  
    }  
    catch ( InterruptedException e ) { }  
}
```

Producer

Guarded Blocks

- The wait() method always has to be called within a while loop because:
 - A thread in a waiting state may be interrupted / notified by another thread either intentionally or accidentally
 - It therefore cannot be guaranteed that the guard condition holds, which is required for the thread to proceed and the thread has to check again the guard condition when it comes out of the wait() ;
 - If the guard condition holds the thread will leave the guarded block and proceed, otherwise it will call wait() again immediately and be put back into the wait set of the shared object

Deadlock

Mutual Exclusion Problem

- We put in place mechanisms for coordinating threads:
 - Threads gain exclusive access to a shared resource
 - Avoids that updates on the object produce wrong results
 - Avoids that threads overwrite each other's updates, which results in updates being lost
 - We serialise access to critical sections, using locks, monitoring, queuing and scheduling mechanisms
- This may cause problems

Deadlock

- Deadlock: two threads are blocking each other
- A deadlock between two threads occurs, if each of them holds a lock on a resource, and one thread will release its own lock as soon as the other thread releases its lock

Thread A holds resource X

Thread B holds resource Y

Thread A will release resource X, if it can access resource Y

Thread B will release resource Y, if it can access resource X

Deadlock

- Necessary conditions for a deadlock to occur (“Coffman Conditions”):
 - Mutual exclusion:
 - At least one resource must be non-shareable – only one thread / process at a time may use this resource
 - “Hold and wait”:
 - A thread / process is currently holding at least one non-shareable resource and is waiting for another non-shareable resource held by another thread / process
 - No preemption:
 - The operating system cannot simply free the resource, it must be given up by the holding thread / process voluntarily
 - Circular wait:
 - Two threads / processes: one thread waits for the release of a resource that is held by a second thread, which, in turn, waits for the release of a resource held by the first thread
 - N threads / processes: given n threads, the first thread waits for the second thread to release a resource, the second waits for the third to release a resource, etc., and the nth thread waits for the first thread to release a resource – circular wait.

Further Problems

- Livelock:
 - If thread A acts in response to the actions of thread B and thread B immediately reacts to thread A's action, they may be locked into an endless loop of mutually reacting to each others' actions, always performing the same set of actions
- Starvation:
 - can occur when threads can have different priorities
 - A thread is never scheduled for execution, because the scheduler always schedules threads with higher priority
 - Scheduling algorithm has to take this problem into account