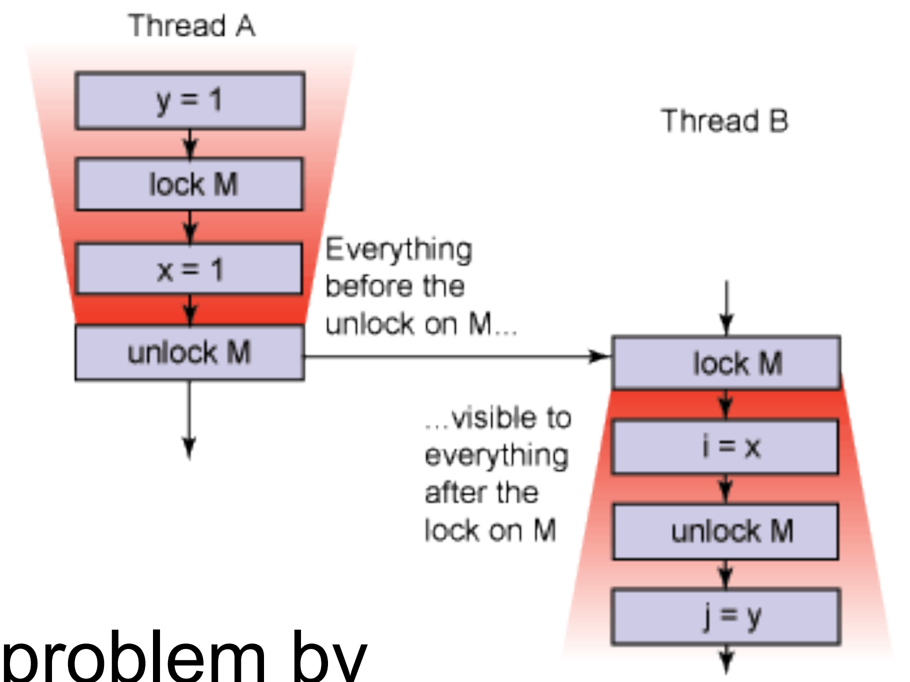
UNIVERSITY *of* STIRLING

Concurrent and Distributed Systems

Java Synchronisation

Contents



- Background
- Bounded Buffer problem
- Critical Sections
- Solving the critical section problem by synchronising processes / threads
- Java Synchronisation

Background

- Cooperating sequential threads/processes run asynchronously and share data
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Illustrate the problem with the Bounded Buffer problem
- Shared-memory solution to bounded-buffer problem has a race condition on the class data ***count***

Bounded Buffer Problem

- Producer

```
while(count == BUFFER_SIZE)
    ; // no-op
// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

- Consumer

```
while (count == 0)
    ; // no-op
// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE
```

Bounded Buffer Problem

- Both consumer and producer work well separately, however, they may not function in combination
- Variable count is shared
- Assume ++count and --count happen concurrently!
- Is the result 4,5, or 6 (due to processor operations) **?**
- **(how? Hint: which starting value is possible?)**
- Race Condition!

```
register1 = count;
register1 = register1 +1;
count = register1;
```

```
register2 = count;
register2 = register2 – 1;
count = register2;
```

Race Conditions: The problem

- Shared variables are written to/read from
- Transfer from one consistent state to another takes several separate operations
- Context switch can happen any time, and operations be interrupted
- Concurrent threads (multi-processor) may share data, also.
- This leads to corrupted data
- Approach: “Critical Section”

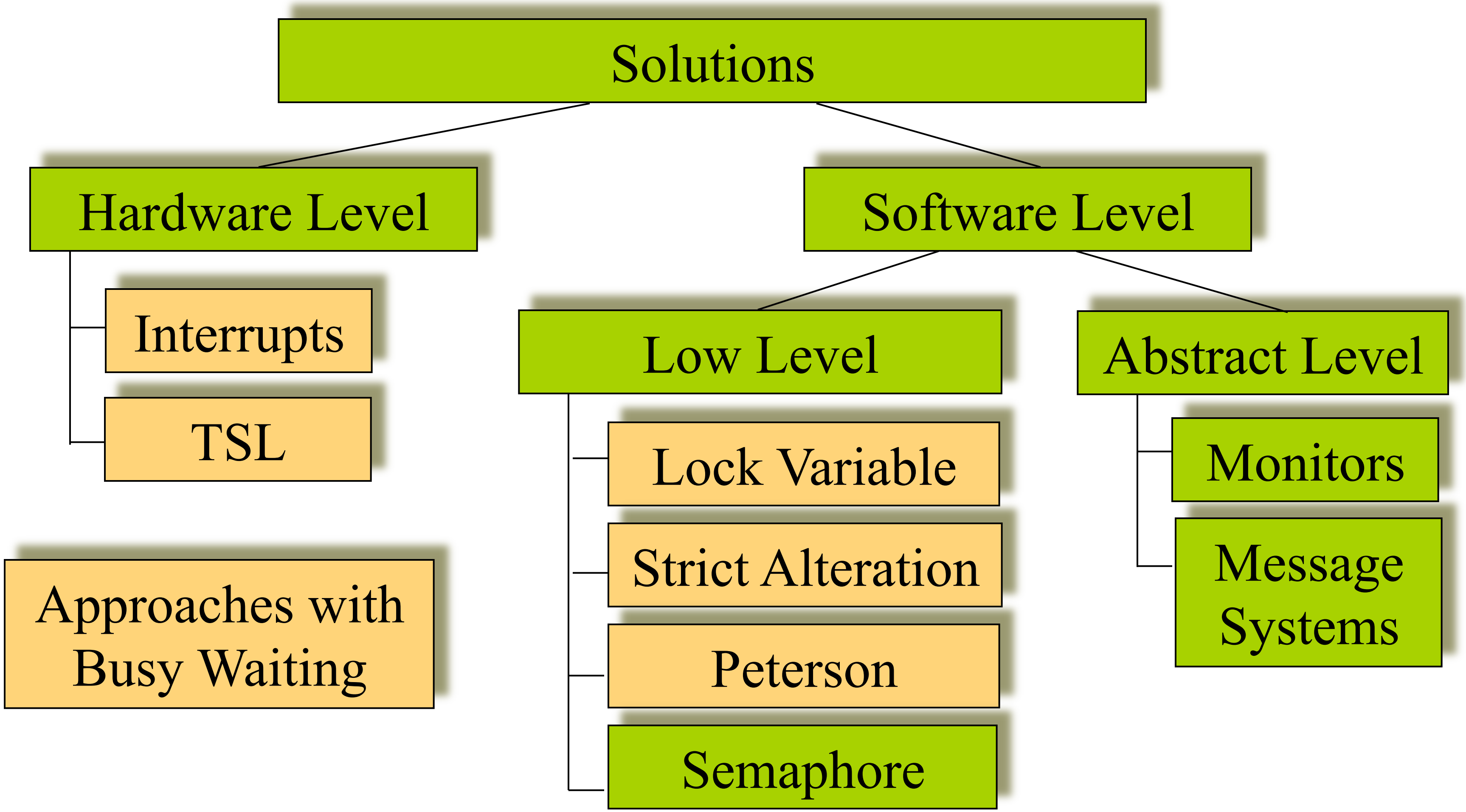
Critical Section

- To prevent race conditions → only one thread at the time manipulates the variable count
- Synchronisation of threads is required!
- Sections of code are declared critical
  - Changing common variables
  - Updating a shared table
  - Writing to a file
- Access to critical sections is regulated
  - If one thread executes in a critical section no other thread may enter their critical sections
  - Mutual exclusion in time

Solving the Critical Section Problem

- A solution must satisfy four requirements
  - No two processes may be simultaneously inside their critical sections
  - No assumptions may be made concerning speeds or numbers of processors
  - No process running outside its critical region may block other processes
  - No process should have to wait forever to enter its critical region (fairness/starvation)

Possible Solutions



Java Synchronisation

- Enforcing mutual exclusion between threads → Thread Safe
- Alternative to Busy Waiting
- Solving Race Conditions
  - Synchronized
  - Wait(), notify(), notifyAll()

Busy Waiting

- Remember Bounded Buffer problem
- → wait until Buffer is not empty / not full
- Alternative: Thread.yield()
  - Thread stays in runnable state
  - Allows JVM to select another thread for execution (equal priority), if any!
  - Problem: Potential deadlock!!!

Deadlock scenario

- Deadlock (informal): a series of processes/thread is waiting on conditions (resources) depending on the other processes/thread in the set and no one can run. Permanent condition.

Necessary conditions for deadlock:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock scenario

- JVM uses priorities, thread with highest priority of all threads in runnable state is run before threads with lower priority
  - `myThread.setPriority(8);`

  almost ever a good solution !

- Producer has higher priority than consumer
  - If buffer is full, producer will execute yield()
  - Consumer still cannot run because of lower priority
  - Deadlock!

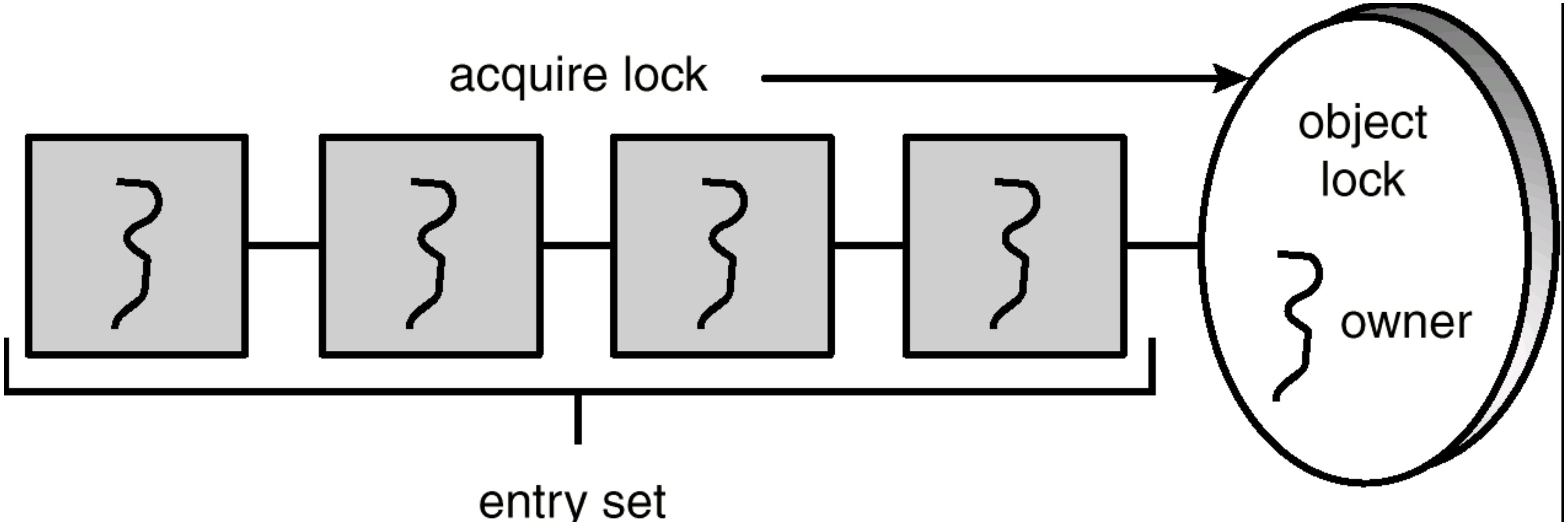
Fixing Race Conditions

- Java introduces keyword **synchronized**
- Every java *Object* has an associated lock
- Object associated with Bounded Buffer class also has a lock associated
- Normally, when a method is invoked, the lock is ignored
- However, using **synchronized** requires owning the lock for the object

Synchronized mechanism

- If the lock is not available (owned by another thread) the thread blocks
- The blocked thread is put into a queue called *entry set*
- Entry set represents all threads waiting for the lock to become available
- The lock is released when the owning thread exists a synchronized method
- One thread from the entry set gets the lock

Entry Set



Code example

```
public synchronized void
            enter(Object item) {

    while (count == BUFFER_SIZE)
          Thread.yield();
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
public synchronized Object remove() {
    Object  item;
    while (count == 0)
          Thread.yield();
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}
```

- **Still danger of Deadlock!**

Wait() and Notify()

- Every lock is also equipped with a *wait set*
- If a thread determines it cannot proceed inside a synchronized method it calls wait()
  - Thread releases the lock for the object
  - The state of the thread is set to blocked
  - The thread is placed in the wait set
- Other threads may acquire the lock
- Deadlock is prevented!

Notify()

- Normally when a thread exits a **synchronized** method, it just releases the lock (perhaps removing one thread from the *entry set*)
- Notify()
  - Picks an arbitrary thread T from the *wait set* and puts it into the *entry set*
  - Moves the state of the thread from blocked to runnable
  - T now competes for the lock with all threads in the entry set
  - Once it owns the lock, the wait() call returns

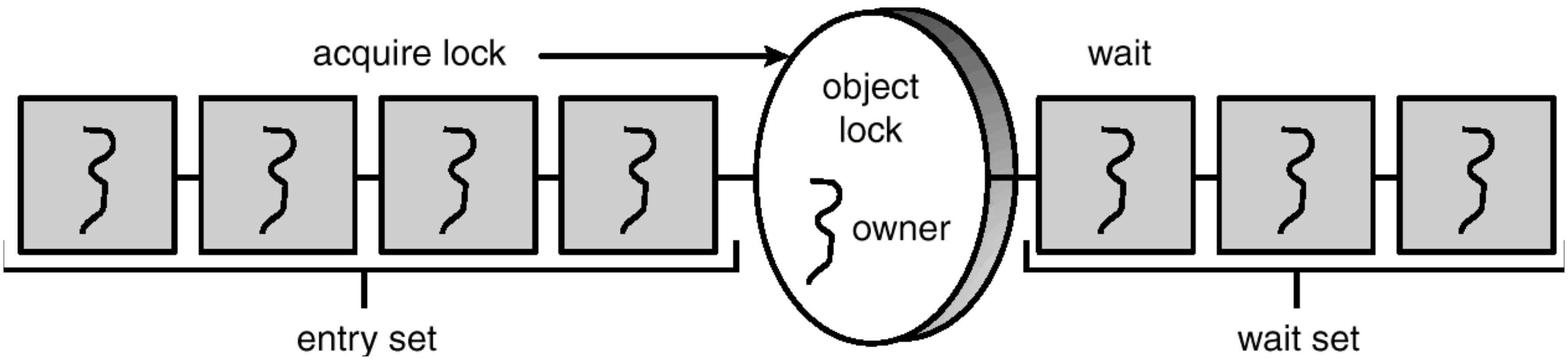
Notify() cont.

- Wait() and Notify() are a synchronisation but even more a communication mechanism

- Wait() and Notify() are independent of the conditions they are used for!

- Wait() and Notify() need to be called from within a synchronized block – otherwise race condition!
  - `IllegalMonitorState` exception

Entry and wait sets



Code example

```
public synchronized
                void enter(Object item) {
    while (count == BUFFER_SIZE)
          try {
                wait();
          }
          catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    notify();
}
```

```
public synchronized Object remove()
    { Object  item;
    while (count == 0)
          try {
                wait();
          }
          catch (InterruptedException e)
    { }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    notify();
    return item;
}
```

Watch the while loop around wait()!

Multiple Notifications

- notify() selects an arbitrary thread from the wait set. This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected.
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- Useful if threads may wait for several conditions
- However, a thread may be woken up for an entirely different condition! - Put wait() into a while loop!
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set
- Inefficient, since all threads need to re-acquire the lock!

Block Synchronisation

- Blocks of code – rather than entire methods – may be declared as synchronized.
- This yields a lock scope that is typically smaller than a synchronized method.
- Uses a java object to perform the synchronisation
- Used for larger methods where only a small part is a critical section
- Use of wait() and notify() possible (use the same object)
- Useful for static methods

Code example

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    // non-critical section
    synchronized(mutexLock) {
        // critical  section
    }
    // non-critical section
}
```

```
Object mutexLock = new Object();
    . . .
public void someMethod() {
    // non-critical section
    synchronized(mutexLock) {
        try{
                mutexLock.wait();
        }catch (InterruptedException ie() {}
    }
    // non-critical section
}
public void someOtherMethod() {
    synchronized(mutexLock) {
        mutexLock.notify();}
}
```

Some rules on synchronisation

- A threads that owns the lock for an object may enter another **synchronized** method of the same object
- A thread can nest synchronized method invocations for different objects. Thus a thread can own the lock for several objects
- If a method is not declared synchronized it can be invoked regardless of lock ownership, even while a synchronized method of the same object is being executed
- If the wait set for an object is empty, then a call to notify() or notifyAll() has no effect
- If an exception occurs while a thread holds a lock, the lock is freed → possible inconsistent state in the object