Concurrent Systems and Applications

Introduction to concurrency

CST Part 1B

Nov 19 – Dec 1 2008

Tim Harris

tharris@microsoft.com

Concurrency (1)

The next section of the course concerns different ways of structuring systems in which concurrency is present and, in particular, co-ordinating multiple threads, processes and machines accessing shared resources and data.

Two main scenarios...

- Tasks operating with a shared address space—e.g. multiple threads created within a Java application.
- Tasks communicating between address spaces—e.g. different processes, whether on the same or separate machines.

In each case we must consider...

- How shared resources and data are named and referred to by the participants.
- Conventions for representing shared data.
- How access to resources and data is controlled.
- What kinds of system failure are possible.

Concurrency (2)

- The focus in this introduction is on the practical foundations of concurrent programming
- Previous examples have been implemented using a single thread that runs the `main` method of a program.
- Java supports lightweight concurrency within an application—multiple threads can be running at the same time.
- Can simplify code structuring and aid interactive response—e.g. one thread deals with user interaction, another thread deals with computation.
- Can benefit from multi-processor and multi-core hardware

Concurrency (3)

Most OS introduce a distinction between processes (as discussed in Part 1A) and threads.

Processes are the unit of protection and resource allocation. Each process has a process control block (PCB) holding:

- identification (e.g. PID, UID, GID);
- memory management information;
- accounting information; and
- (references to) one or more TCB...

Threads are the entities considered by the scheduler. Each thread has a thread control block (TCB) holding:

- thread state;
- saved context information;
- references to user (and kernel?) stack; and
- scheduling parameters (e.g. priority).

Concurrency (4)

Structure of this section:

- Managing threads in Java.
- Simple shared objects—shared counters, shared hashtables, etc.
  - Mutual exclusion locks (mutexes)
- Shared objects in Java with internal blocking—queues, multi-reader single-writer (MRSW) locks.
  - Condition variables (condvars)
- Implementation of mutexes and condvars.
  - Direct scheduler support
  - Semaphores
  - Event counters / sequences
- Alternative abstractions
  - Monitors
  - Active objects

Creating threads in Java (1)

- There are two ways of creating a new thread. The simplest is to define a sub-class of `java.lang.Thread` and to override the `run()` method.
  - `run()` provides the code that the thread will execute while it is on the CPU.
  - The thread terminates when `run()` returns.
  - It is common to have daemonic threads in a loop: `while (!done) <code>`

```
class MyThread extends Thread {
  public void run() {
    while (true) {
      System.out.println("Hello from " +
                         this);
      Thread.yield();
    }
  }

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

Creating threads in Java (2)

- The `run` method of the class `MyThread` defines the code that the new thread(s) will execute. Just defining such a class does not create any threads.
- The two calls to `new` instantiate the class to create two objects representing the two threads that will be executed.
- The calls to `start()` actually start the two threads executing.
- The program continues to execute until all ordinary threads have finished, even after the `main` method has completed.

```
$ java MyThread
Hello from Thread[Thread-0,5,main]
Hello from Thread[Thread-1,5,main]
Hello from Thread[Thread-0,5,main]
Hello from Thread[Thread-1,5,main]
...
```

- A daemon thread will not prevent the application from exiting.

```
t1.setDaemon(true);
```

Creating threads in Java (3)

- The second way of creating a new thread is to define a class that `implements` the `java.lang.Runnable` interface.

```
class MyCode implements Runnable {
  public void run() {
    while (true) {
      System.out.println("Hello from " +
                  Thread.currentThread());
      Thread.yield();
    }
  }

  public static void main(String [] args) {
    MyCode mt = new MyCode();
    Thread t_a = new Thread(mt);
    Thread t_b = new Thread(mt);
    t_a.start();
    t_b.start();
  }
}
```

Creating threads in Java (4)

- As before, the `MyCode` class defines the code that the new threads will execute.
- The two calls to `new` instantiate two `Thread` objects, passing a reference to an instance of `MyCode` to them as their targets.
- The two calls to `start()` set the two threads executing.
- Note that here the `run()` methods of the two threads are being executed on the same `MyCode` object, whereas two separate `MyThread` objects were required.
- The second way of creating threads is more complex, but also more flexible.
  - It doesn't consume the single opportunity to sub-class a parent class.
- Generally, the fields in the class containing the `run()` method will hold per-thread state—e.g. which part of a problem a particular thread is tackling.

Creating threads in Java (5)

- In some situations a thread is interrupted immediately if it is blocked—e.g. `sleep` may throw `InterruptedException`. For example:

```
class Example {
  public static void main(String [] args) {
    Thread t = new Thread() {
      public void run() {
        try {
          do {
            Thread.sleep(1000); // 1s sleep
          } while (true);
        } catch (InterruptedException ie) {
          // Interrupted: better exit
        }
      }
    };
    t.start();
    t.interrupt();
  }
}
```

- If the thread didn't block then the `while (true)` could perhaps be

```
while (!isInterrupted());
```

Join

- The `join` method on `java.lang.Thread` causes the currently running thread to wait until the target thread dies.

```
class Example {
  public void startThread(void)
    throws InterruptedException
  {
    Thread t = new Thread() {
      public void run() {
        System.out.println("Hello world!");
      }
    };

    t.start();
    t.join(0);
  }
}
```

- The call to `join` waits for the thread started on the previous line to finish. The parameter specifies a time in milliseconds (0 $\longrightarrow$ wait forever).
- The `throws` clause on `startThread` is required: the call to `join` may be interrupted.

Priority controls

- Methods `setPriority` and `getPriority` on `java.lang.Thread` allow the priority to be controlled.
- A number of standard priority levels are defined: `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY`.
- The programmer can also try to influence thread scheduling using the `yield` method on `java.lang.Thread`. This is a hint to the system that it should try switching to a different thread—note how it was used in the previous examples.
  - In a non-preemptive system even low priority threads may continue to run unless they periodically `yield`.
- Selecting priorities becomes complex when there are many threads or when multiple programmers are working together.
- Although it may work on some systems, the variation in behaviour between different JVMs means that it is never correct to use thread priorities to control access to shared data in portable code.

Thread scheduling

- The choice of exactly which thread(s) execute at any given time can depend both on the operating system and on the JVM.
- Some systems are preemptive—i.e. they switch between the threads that are eligible to run. Typically these are systems in which the OS supports threads directly, i.e. maintaining separate PCBs and TCBs.
- Other systems are non-preemptive—i.e. they only switch when the running thread yields, becomes blocked, or exits. Typically these systems implement threads within the JVM.
- The Java language specification says that, in general, threads with higher priorities will run in preference to those with lower priorities.
- To write correct, portable code it is therefore important to think about what the JVM is guaranteed to do—not just what it does on one system. Different behaviour might occur at different nodes within a distributed system!

The `volatile` modifier (1)

```
static boolean signal = false;

public void run() {
  while (!signal) {
    doSomething();
  }
}
```

If some other thread sets the `signal` field to `true` then what will happen?

- The thread running the code above might keep executing the `while` loop.
- This might happen if the JVM produces machine code that loads the value of `signal` into a processor register and just tests that register value each time around the loop.
  - This is common when the body of the loop is short—no need for compiler to re-use the register containing `signal`.
  - Commonly seen in embedded C/Java systems.
- Such behaviour is valid and might help performance.

The `volatile` modifier (2)

`volatile` is a modifier that can be applied to fields, e g.

```
static volatile boolean signal = false;
```

When a thread reads or writes a `volatile` field it must actually access the memory location in which that field's value is held.

The precise rules about when it is permitted for the JVM to re-use a value that is held in a register are still being formulated. However, in general, if a shared field is being accessed then either:

- the thread updating the field must release a mutual exclusion lock that the thread reading from the field acquires; or
- the field should be `volatile`.

As we will see, the first condition is satisfied by the usual use of `synchronized` methods (or classes) $\longrightarrow$ `volatile` is rarely seen in practice.

For more details, see Section 2.2 of Doug Lea's book (online at `http://gee.cs.oswego.edu/dl/cpj/jmm.html`).

Exercises

1. Describe the facilities in Java for creating multiple threads of execution.
2. What is the difference between a preemptive and a non-preemptive scheduler? Write a Java class containing a method

   ```
   boolean probablyPreemptive();
   ```

   which returns `true` if the JVM running it appears to be preemptive and returns `false` otherwise. (Hint: your solution will probably need to start multiple threads that perform some kind of experiment.)
3. A Java-based file server is to use a separate thread for each user granted access. Discuss the merits of this approach from the point of view of security, possible performance, and likely ease of implementation.
4. Examine the behaviour that one or more JVMs provide for the following aspects of thread management:

   (i) whether scheduling is preemptive;

   (ii) whether the highest-priority runnable-thread is guaranteed to run; and

   (iii) the impact on performance of making a frequently-accessed field `volatile`.

Mutual exclusion

Previous lecture

- Creating and terminating threads
- `volatile`

Overview of this lecture

- Shared data structures
- Mutual exclusion locks

Safety

In concurrent environments we must ensure that the system remains safe no matter what the thread scheduler does—i.e. that ‘nothing bad happens’.

- Unlike type-soundness, it is usually the case that this cannot be checked automatically by compilers or tools (although some exist to help).
- It is often useful to think of safety in terms of invariants—things that must remain true, no matter how different parts of the system evolve during execution.
  - e.g. a ‘transfer’ operation between bank accounts preserves the total amount.
- We can identify consistent object states in which all invariants are satisfied.
- ...and aim that each of the operations available on the system keeps it in a consistent state.
- Therefore many of the problems that we will see come down to deciding when different threads can be allowed access to objects in various ways.

Liveness

As well as safety, we would also like liveness—i.e. ‘something good eventually happens’. We often distinguish per-thread and system-wide liveness.

Standard problems include:

- Deadlock—a circular dependency between processes holding resources and processes requiring them. Typically the ‘resources’ are exclusive access to locks.
- Livelock—a thread keeps executing instructions but makes no useful progress, e.g. busy-waiting on a condition that will never become true.
- Missed wake-up (wake-up waiting)—a thread misses a notification that it should continue with some operation and instead remains blocked.
- Starvation—a thread is waiting for some resource but never receives it—e.g. a thread with a very low scheduling priority may never receive the CPU.
- Distribution failures—of nodes or network connections in a distributed system.

Shared data (1)

- Most useful multi-threaded applications will share data between threads.
- Sometimes this is straightforward, e.g. data passed to a thread through fields in the object containing the `run()` method.
- More generally, threads may share state through...
  - `static` fields in mutually-accessible classes, e.g. `System.out`.
  - objects to which multiple threads have references.
- What happens to field `o.x`:

```
Thread A        Thread B
o.x = 17;       o.x = 42;
```

- Most field accesses are atomic in Java (and many other languages)—the value read from `o.x` after those updates will be either 17 or 42.
- The only exceptions are numeric fields of type `double` or type `long`—some third value may be read in those cases.

Shared data (2)

- This is an example of a race condition: the result depends on the uncontrolled interleaving of the threads' executions.
- We need some way of controlling how threads are executed when accessing shared data.
- The basic notion is of critical regions: parts of the program during which a thread should have exclusive access to some data structures while making a number of operations on them.
- Careful programming is rarely sufficient, e.g.

```
boolean busy;
int x;

...

while (busy) { /* nothing */ }
busy = true;
x = x + 1;
busy = false;
```

- Using `x++` would be no better.

Locks in Java (1)

- Simple shared data structures can be managed using mutual exclusion locks ('mutexes') and the `synchronized` keyword to delimit critical regions.
- The JVM associates a separate mutex with each object. Each acts like the 'busy' flag on the previous slide except:
  - There is no need to spin while waiting for it—the thread is blocked.
  - The race condition between the `while` loop and `busy=true;` is avoided.
- The `synchronized` keyword can be used in two ways—either applied to a method or applied to a block of code.
- For example, suppose we want to maintain an invariant between multiple fields:

```
class BankAccounts {
  private int balanceA;
  private int balanceB;

  synchronized void transferToB(int v) {
    balanceA = balanceA - v;
    balanceB = balanceB + v;
  }
}
```

Locks in Java (2)

- When a synchronized method is called, the thread must lock the mutex associated with the object.
- If the lock is already held by another thread then the called thread is blocked until the lock becomes available.
- Locks therefore operate on a per-object basis—that is, only one synchronized method can be called on a particular object at any time.
  - ...and similarly, it is OK for multiple threads to be calling the same method, so long as they do so on different objects.
- Locks are re-entrant, meaning that the thread may call one synchronized method from another.
- If a `static synchronized` method is called then the thread must acquire a lock associated with the class rather than with an individual object.
- The `synchronized` modifier cannot be used directly on classes or on fields.

Locks in Java (3)

- The second form of the `synchronized` keyword allows it to be used within methods, e.g.

```
void methodA(Object x) {
  synchronized (x) {
    System.out.println("1");
  }

  ...

  synchronized (x) {
    System.out.println("2");
  }
}
```

- The first `synchronized` region locks the mutex associated with the object to which `x` refers, performs the `println` operation, and then releases the lock.
- Before entering the second region, the mutex must be re-acquired.

  This kind of usage is good if an intervening operation, not requiring the mutual exclusion, may take a long time to execute: other threads may acquire the lock while the computation proceeds.

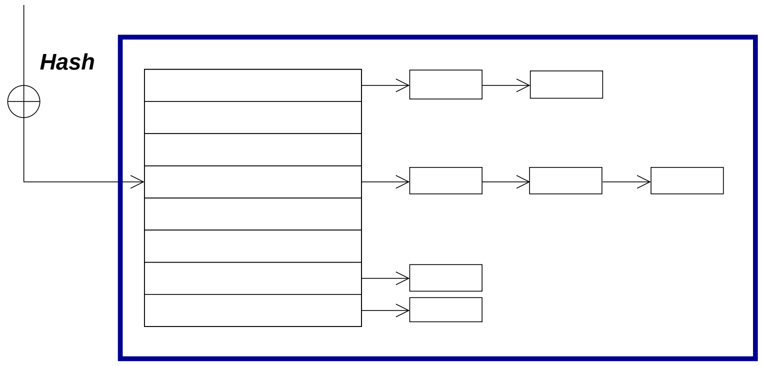
What about exceptions and errors?

- What if an exception is thrown inside a `synchronized` region or a `synchronized` method?
- If the exception is not caught inside the region/method, then the flow of execution leaves the synchronized region.
- The JVM will release the mutex automatically before executing the `catch` block.
  - This helps prevent deadlock caused by accidentally not releasing a mutex.
  - But sometimes we need to exercise a little caution...

```
void screwItUp(Bank college) {
  try {
    synchronized (college) {
      college.credit(fees);
    }
  } catch (OutOfMoneyException oome) {
    System.out.println("Credit to "+
      college+" failed!");
    // oops... that access on ’college’
    //         wasn’t thread-safe!
  }
}
```

Compound data structures (1)

- How can we use locks on a data structure built from multiple objects, e.g. a hashtable?
- One "big lock" associated with the hashtable object itself:



Advantages

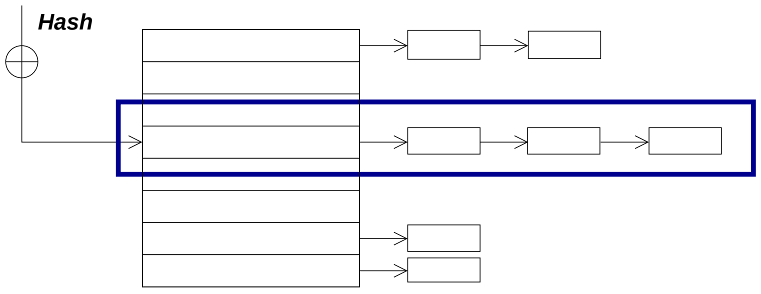
- Easy to implement
- "Obviously correct"
- Good performance under light load: only one lock to acquire/release per operation.

Disadvantage

- Poor performance in most other cases—only one operation can proceed at a time.

Compound data structures (2)

- Separate "small locks", e.g. associated with each bucket of the hashtable:



Advantage

- Operations using different buckets can proceed concurrently.

Disadvantage

- Harder to implement—consider resizing the hashtable...

In general designing an effective fine-grained locking scheme is hard:

- A poor scheme may leave the program spending its time juggling locks rather than doing useful work.
- Having many locks does not automatically imply better concurrency.
- Deadlock problems...

Exercises

1. Describe how the mutual-exclusion locks provided by the `synchronized` keyword can be used to control access to shared data structures.
2. Describe what a race condition is, with the aid of example code.

   "A Java class is safe for use by multiple threads if all of its methods are synchronized."

   To what extent do you agree with this statement?
3. Suppose that, instead of using mutual exclusion locks, a programmer attempts to support critical regions by manipulating the running thread's scheduling priority in a class extending `java.lang.Thread`:

   ```
   void enterCriticalRegion() {
     oldPriority = getPriority();
     setPriority(Thread.MAX_PRIORITY);
   }

   void exitCriticalRegion() {
     setPriority(oldPriority);
   }
   ```

   What assumptions are needed to guarantee this works? Does your JVM guarantee them?

Deadlock

Previous lecture

- Safety and liveness requirements
- Mutual exclusion locks

Overview of this lecture

- Deadlock
- Automatic detection
- Avoidance

Deadlock (1)

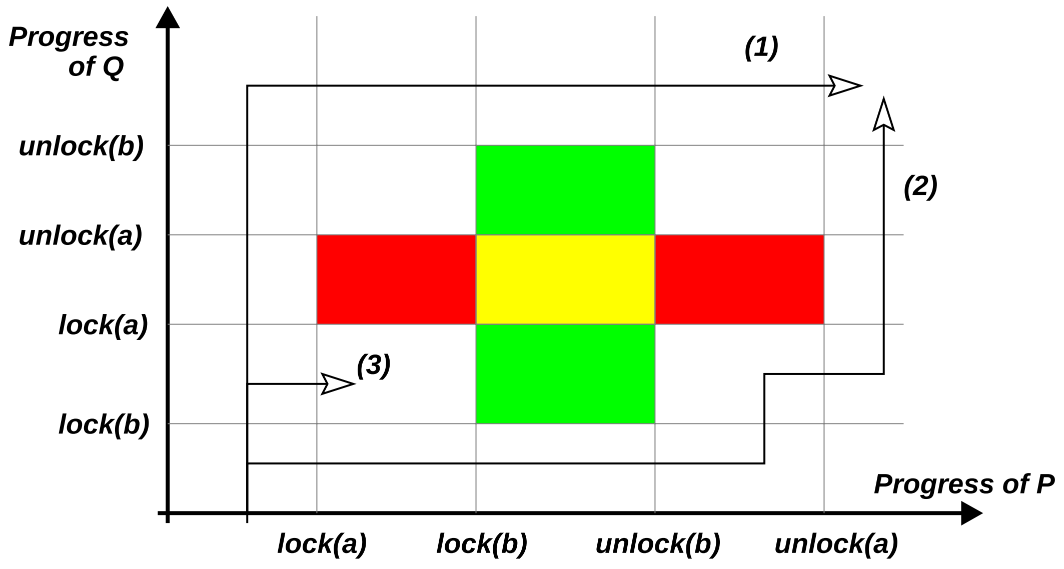
Suppose that `a` and `b` refer to two different shared objects,

```
Thread P            Thread Q
synchronized (a)    synchronized (b)
  synchronized        synchronized
(b)                 (a)
  {                   {
    ...                 ...
  }                   }
```

- If P locks both `a` and `b` then it can complete its operation and release both locks, thereby allowing Q to acquire them.
- Similarly, Q might acquire both locks, then release them and thus allow P to continue.
- But, if P locks `a` and Q locks `b` then neither thread can continue: they are each deadlocked waiting for the resources that the other has.

Deadlock (2)

Whether this deadlock actually occurs depends on the dynamic behaviour of the applications. We can show this graphically in terms of the threads' progress:



- In the horizontal area one thread is blocked by the other waiting to lock `a`. In the vertical area it is lock `b`.
- Paths (`1`) and (`2`) show how these threads may be scheduled without reaching deadlock.
- Deadlock is inevitable on path (`3`) (but hasn't yet occurred in the position indicated).

Requirements for deadlock

If all of the following conditions are true then deadlock exists:

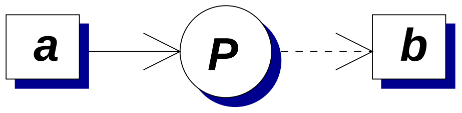
1. A resource request can be refused—e.g. a thread cannot acquire a mutual-exclusion lock because it is already held by another thread.
2. Resources are held while waiting—e.g. while a thread blocks waiting for a lock it does not have to release any others that it holds.
3. No preemption of resources is permitted—e.g. once a thread acquires a lock then it is up to that thread to choose when to release it, it cannot be taken away from the thread.
4. Circular wait—a cycle of threads exists such that each holds a lock requested by the next process in the cycle, and that request has been refused.

In the case of mutual exclusion locks in Java, 1–3 are always true (they are static properties of the language), and so the existence of a circular wait leads to deadlock.

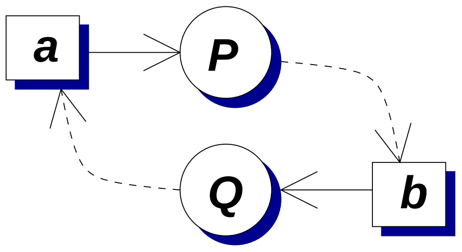
Object allocation graphs

An object allocation graph shows the various tasks in a system and the resources that they have acquired and are requesting. We will use a simplified form in which resources are considered to be individual objects.

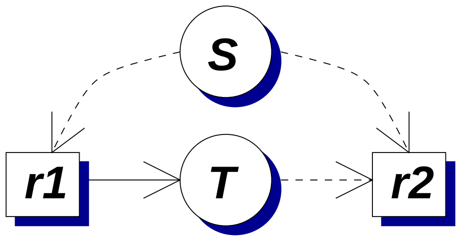
`a` is held by thread `P` and `P` is requesting object `b`:



`a` is held by `P`, `b` is held by `Q`:



Should `r2` be allocated to `S` or `T`?



Deadlock detection (1)

Deadlock can be detected by looking for cycles in object allocation graphs (as in the second example on the previous slide).

Let A be the object allocation matrix, with one thread per row and one column per object. A_{ij} indicates whether thread i holds a lock on object j.

Let R be the object request matrix. R_{ij} indicates whether thread i is waiting to lock object j.

We proceed by marking rows of A indicating threads that are not part of a deadlocked set. Initially no rows are marked. A working vector W indicates which objects are available.

1. Select an unmarked row i such that $R_i \leq W$—i.e. a thread whose requests can be met. Terminate if no such row exists.
2. Set $W = W + A_i$, mark row i, and repeat.

This identifies when deadlock has occurred—we might be interested in other properties such as whether deadlock is:

- inevitable (must happen in all possible execution paths); or
- possible (might happen in some paths).

Deadlock detection (2)

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

1. $W = (0, 0, 0, 1, 1)$
2. Thread 3's requests can be met $\Rightarrow$ it is not deadlocked so can continue and might release object 1.
3. $W = (1, 0, 0, 1, 1)$
4. Thread 4's requests can now be met $\Rightarrow$ it is not deadlocked.
5. $W = (1, 0, 0, 1, 1)$

➤ Nothing more can be done: threads 1 and 2 are deadlocked.

Deadlock avoidance (1)

A conservative approach:

- Require that each process identifies the maximum set of resources that it might ever lock, C_{ij}.
- When a thread i requests a resource then construct a hypothetical allocation matrix A' in which it has been made and a hypothetical request matrix B' in which every other process makes its maximum request.
- If A' and B' do not indicate deadlock then the allocation is safe.

Advantage

- This does avoid deadlock—might be preferable to deadlock recovery.

Disadvantages

- Need to know maximum requests.
- Run-time overhead.
- What if there are no safe states?
- Objects are usually instantiated dynamically...

Deadlock avoidance (2)

It is often more practical to prevent deadlock by careful design. How else can we tackle the four requirements for deadlock?

- Use locking schemes that allow greater concurrency—e.g. multiple-readers, single-writer in preference to mutual exclusion.
- Do not hold resources while waiting—e.g. acquire all necessary locks at the same time.
- Allow preemption of locks and roll-back (not a primitive in Java if using built-in locks).
  - Hardware/software transactional memories...

Two practical schemes that are widely applicable:

- Coalesce locks so that only one ever needs to be held—e.g. have one lock protecting all bank accounts.
- Enforce a lock acquisition order, making it impossible for circular waits to arise, e.g. lock two 'bank account' objects in order of increasing account number.

...trade-off between simplicity of implementation and possible concurrency.

Priority inversion (1)

Another liveness problem in priority-based systems:

- Consider low, medium, and high priority threads called P_{low}, P_{med}, and P_{high} respectively.
  1. First P_{low} starts, and acquires a lock on object a.
  2. Then the other two processes start.
  3. P_{high} runs since it has the highest scheduling priority, tries to lock a, and blocks.
  4. Then P_{med} gets to run, thus preventing P_{low} from releasing a, and hence P_{high} from running.
- Usual solution is priority inheritance:
  - associate with every lock the priority p of the highest priority process waiting for it; then
  - temporarily boost the priority of the holder of the lock up to p.
  - We can use handoff scheduling to implement this.
- Windows 2000 "solution": priority boosts
  - checks if $\exists$ a thread in the ready-to-run state but not run for $\geq$ 300 ticks.
  - if so, double the on-CPU time quantum and boost priority to 15.
- What happens in Java?

Priority inversion (2)

- With basic priority inheritance we can distinguish (assuming a uni-processor with strict-priority scheduling)
  - direct blocking of a thread waiting for a lock; and
  - push-through blocking of a thread at one priority by an originally-lower-priority thread that has inherited a higher priority.
- A thread P_{high} can be blocked by each lower priority thread P_{low} for at most one of P_{low}'s critical sections.
- A thread P_{high} can experience push-through blocking for any lock accessed by a lower-priority thread and by a job which has (or can inherit) a priority $\geq P_{high}$.

  This can give an upper bound on the total blocking delay that a thread encounters, but
- chains of blocking may limit the bounded and practical performance: the former is a particular problem for real-time systems; and
- remember: does not prevent deadlock.

Exercises (1)

1. In the dining philosophers problem, five philosophers spend their time alternately thinking and eating. They each have a chair around a common, circular table. In the centre of the table is a bowl of spaghetti and the table is set with five forks, one between each pair of adjacent chairs. From time to time philosophers might get hungry and try to pick up the two closest forks. A philosopher may only pick up one fork at a time. It is a common axiom of philosophic thought that one is only allowed to eat with the aid of two forks and that, of course, both forks are put down while thinking.

   Model this problem in Java using a separate thread for each philosopher.

   Does your simulation illustrate either deadlock or livelock? If so, then what changes could you make to avoid it?

2. Write a Java class that attempts to cause priority inversion with a medium-priority thread preventing a high-priority thread from making progress. Do you observe priority inversion in practise?

Exercises (2)

3. Show how the deadlock detection algorithm can be extended to manage locks that support a separate write mode (in which it can be held by at most one thread at a time) and a read mode (in which it can be held by multiple threads at once). The lock cannot be held in both modes at the same time.

4. One way to avoid deadlock is for a thread to simultaneously acquire all of the locks that it needs for an operation. However, Java's `synchronized` keyword can only acquire or release a single lock at a time.

   Sketch the design of a class `LockManager` that implements the `LockManagerIfc` interface (below) so that the `doWithLocks` operation:

   1. appears to atomically acquire locks on all of the objects in the array `o`;
   2. invokes `op.doOp(arg)` keeping the result of that method as the eventual result of `doWithLocks()`; and
   3. releases all of the locks initially acquired.

Exercises (3)

```
interface Operation {
  Object doOp(Object arg);
}

interface LockManagerIfc {
  Object doWithLocks(Object o[],
                     Operation op,
                     Object arg);
}
```

Hint: one approach is to assume initially some mechanism for mapping each object to a unique integer value and later to examine how to provide that mechanism.

Condition synchronization

Previous lecture

- Deadlock
- Ordered acquisition
- Priority inversion and inheritance

Overview of this lecture

- Condition synchronization
- `wait`, `notify`, `notifyAll`

Limitations of mutexes (1)

- Suppose we want a one-cell buffer with a `putValue` operation (store something if the cell is empty) and a `removeValue` operation (read something if there is anything in the cell).

```
class Cell {
  private int value;
  private boolean full;

  public synchronized int removeValue() {
    if (full) {
      full = false;
      return value;
    } else {
      /* ??? */
    }
  }

  ...
}
```

- What can we write in place of "`/* ??? */`" to finish the code?

Limitations of mutexes (2)

- We could keep testing `full`—i.e. implement a ‘spin lock’...

```
 1    class Cell {  /* Incorrect */
 2      private int value;
 3      private boolean full;
 4
 5      public int removeValue() {
 6        while (!full) {/* nothing */}
 7        synchronized (this) {
 8          full = false;
 9          return value;
10        }
11      }
12    }
```

But this is...

1. incorrect: if multiple threads try to remove values then they might each see `full` false at Line 6 and independently execute 7–10;
2. inefficient: threads consume CPU time while waiting $\longrightarrow$ this might impede a thread about to put a value into the cell; and
3. incorrect: `full` needs to be `volatile` anyway!

Limitations of mutexes (3)

- Another problem: what if we want to enforce some other kind of concurrency control?
- e.g. if we identify read-only operations which can be executed safely by multiple threads at once.
- e.g. if we want to control which thread gets next access to the shared data structure.
  - perhaps to give preference to threads performing update operations,
  - or to enforce a first-come first-served regime,
  - or to choose on the basis of the threads' scheduling priorities?
- All that mutexes are able to do is to prevent more than one thread from running the code on a particular object at the same time.

Condition synchronization

- What we might like to write:

```
 1    class Cell {  /* Not valid Java */
 2      private int value;
 3      private boolean full;
 4
 5      public synchronized int removeValue() {
 6        wait_until (full);
 7        full = false;
 8        return value;
 9      }
10    }
```

- Line 6 would have the effect of
  - if `full` is false, blocking the caller atomically with doing the test and releasing the lock on the cell—the method is `synchronized`—to allow another thread to put items into it; and
  - unblocking the thread when `full` becomes true and the lock can be re-acquired (so the lock prevents multiple 'removes' of the same value).
- We can't directly implement `wait_until` in Java...
  - call-by-value semantics mean that `full` would be evaluated only once!
  - we would need some way of releasing the lock on the `Cell`.

Condition variables

- Condition variables provide one solution.
- In general, condition variables support two kinds of operation:
  - a cv.CVWait(m) operation causing the current thread to atomically release a lock on mutex `m` and to block itself on condition variable `cv`, re-acquiring the lock on `m` before it completes; and
  - a cv.CVNotify(m) operation that wakes up (one? all?) threads blocked on `cv`.
- Such operations would be more cumbersome in this simple example than a general `wait_until` primitive:

```
class Cell {  /* Not valid Java */
  private int value;
  private boolean full;
  private ConditionVariable cv =
    new ConditionVariable();

  public synchronized int removeValue() {
    while (!full) cv.CVWait(this);
    full = false;
    cv.CVNotify();
    return value;
  }
}
```

Condition variables in Java (1)

- Java doesn't (currently) provide individual condition variables in this way.
- Instead, each object `o` has an associated condition variable which is accessed by:
  - `o.wait()`
  - `o.notify()`
  - `o.notifyAll()`
- Calling `o.wait()` acts as the equivalent of `cv.CVWait(o)` on the condition variable associated with `o`.
- This means that `o.wait()` always releases the mutual exclusion lock held on `o`...
- ...and therefore the caller may only use `o.wait()` when holding that lock (otherwise an `IllegalMonitorStateException` is thrown).
- `o.notify()` unblocks exactly one thread (if any are waiting), otherwise it does nothing—no wake-up waiting is left.
- `o.notifyAll()` unblocks all waiting threads.

Condition variables in Java (2)

```java
class Cell {
  private int value;
  private boolean full = false;

  public synchronized int removeValue()
    throws InterruptedException
  {
    while (!full) wait();

    full = false;
    notifyAll();
    return value;
  }

  public synchronized void putValue(int v)
    throws InterruptedException
  {
    while (full) wait();

    full = true;
    value = v;
    notifyAll();
  }
}
```

Condition variables in Java (3)

- Line 8 causes a thread executing `removeValue()` to block on the condition variable until the cell is full.
  - Think about whether I really need the `while` loop around `wait()` (answer in 4 slides' time...).
- Line 10 updates the object to mark it empty.
- Line 11 notifies all threads currently blocked on the condition variable.
- Similarly, Line 18 causes a thread executing `putValue()` to block on the condition variable until the cell is empty.
- Lines 20–21 update the fields to mark the cell full and store the value in it, Line 22 notifies waiting threads.

An `InterruptedException` will be thrown if the thread is interrupted while waiting. In general it should be propagated until it can be handled. Be wary of writing:

```
try {
  while (full) wait();
} catch (InterruptedException ie) {
  /* nothing */
}
/* did we get here because wait() succeeded
   or were we interrupted?    */
```

Condition variables in Java (4.1)

Is this code cunning or broken?

```
class Cell {
  private int value;
  private boolean full = false;

  public synchronized int removeValue()
    throws InterruptedException
  {
    while (!full) {
      wait();
      /* Should a put’er or a remove’er have
         been woken up?  */
      if (!full) {
        // pass on the nofitication,
        // hopefully to a put’er.
        notify();
      }
    }

    full = false;
    notify();  // surely waking one thread
               // is better than waking all?
    return value;
  }
```

Condition variables in Java (4.2)

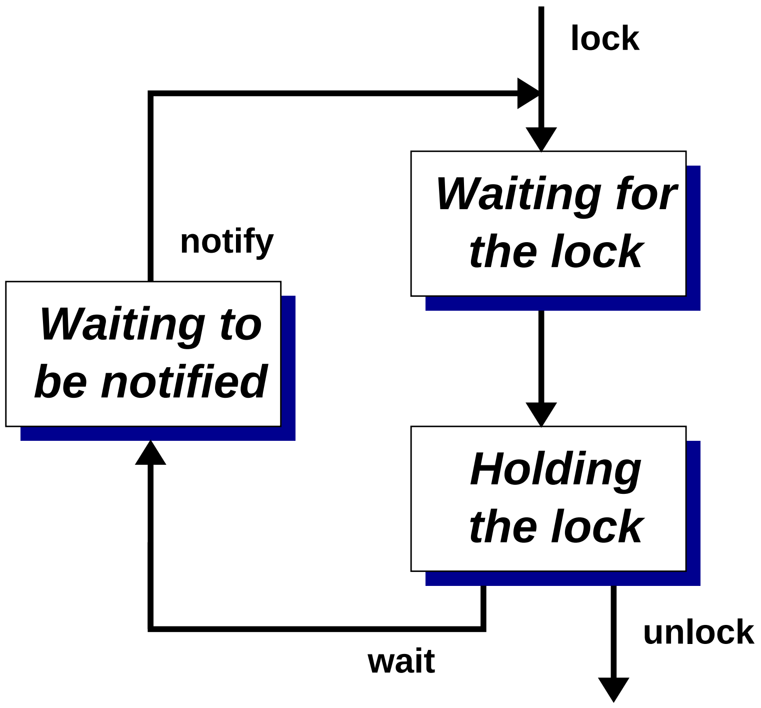
Is this code cunning or broken (continued)?

```
  public synchronized void putValue(int v)
    throws InterruptedException
  {
    while (full) {
      wait();
      /* Should a put'er or a remove'er have
         been woken up?  */
      if (full) {
        // pass on the nofitication,
        // hopefully to a remove'er.
        notify();
      }
    }

    full = true;
    value = v;
    notify();  // surely waking one thread
               // is better than waking all?
  }
}
```

Condition variables in Java (5)

- Note how there are now two different ways that a thread may be blocked:



- It might have entered a `synchronized` region for an object and found that the associated mutual exclusion lock is already held.
- It might have called `wait()` on an object and blocked until the associated condition variable is notified.
- When notified, the thread must compete for the lock once more.

Condition variables in Java (5)

- When should `notify()` be used and when should `notifyAll()` be used?
- With `notifyAll()` the programmer must ensure that every thread blocked on the condition variable can continue safely:
  - e.g. Line 8 in the example surrounds the invocation of `wait()` with a `while` loop;
  - if a 'removing' thread is notified when there is no work for it, it just waits again.
- `notify()` selects arbitrarily between the waiting threads: the programmer must therefore be sure that the exact choice does not matter.
- In the `Cell` example, we can't use `notify()` because although only one thread is to be woken a successful `removeValue()` must allow a call blocked in `putValue()` to proceed rather than another thread that is blocked in `removeValue`—we cannot control which thread will be notified by the `notify()` call.

`notify()` does not guarantee to wake the longest waiting thread.

Suspending threads

- The `suspend()` and `resume()` methods defined on `java.lang.Thread` allow one thread to temporarily stop and start the execution of another (or to suspend itself).

  ```
  Thread t = new MyThread();
  t.suspend();
  t.resume();
  ```

- As with `stop()`, the `suspend()` and `resume()` methods are deprecated.
- This is because the use of `suspend()` can lead to deadlocks if the target thread is holding locks. It also risks missed wake-up problems:

  ```
  1    public int removeValue() {
  2      if (!full) {
  3        Thread.suspend(Thread.currentThread());
  4      }
  ```

  The status might change between executing Lines 2 and 3 $\longrightarrow$ a lost wake-up problem!

  `suspend()` should never be used: even if the program does not explicitly take out locks the JVM might use locks in its implementation.

Exercises

1. Describe the facilities in Java for restricting concurrent access to critical regions. Explain how shared data can be protected through the use of shared objects.
2. Consider the following class definition:

```
class Example implements Runnable {
  public static Object o = new Object();
  int count = 0;
  public void run() {
    while (true) {synchronized(o) {++count;}}
  }
}
```

(i) Show how to start two threads, each executing this `run()` method on separate instances of `Example`.

(ii) When this program runs, only one of the `count` fields is found to increment even though threads are scheduled preemptively. Why might this be?

(iii) If two threads are run on the same instance of class `Example`, would you expect the value of `count` to increase more rapidly or less rapidly than a single thread running `while (true) ++count;`? Why? Does it make any difference if the machine being used has several processors instead of just one?

(iv) Compared to a uni-processor, approximately how rapidly would you expect `count` to increase on a dual-processor machine running the code if the synchronization on `o` was removed from the `while` loop entirely?

Previous lecture

- Condition synchronization
- `wait`, `notify`, `notifyAll` in Java

Overview of this lecture

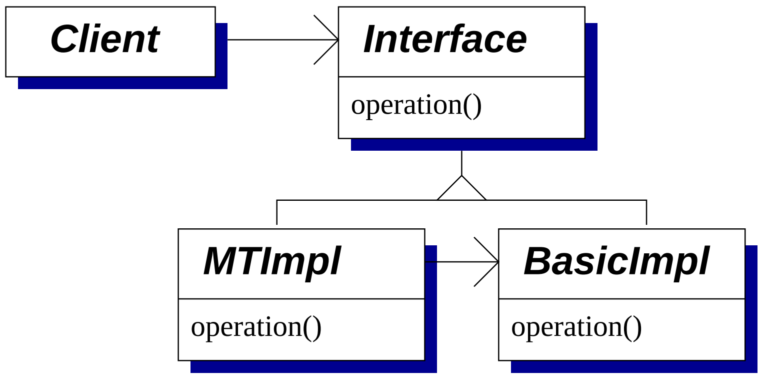
- Further examples of how to use these facilities
- Common design features

Design (1)

Suppose that we wish to have a shared data structure on which multiple threads may make read-only access, or a single thread may make updates ⇒ the multiple-reader, single-writer problem.

- How can this be implemented using the facilities of Java:
  - In terms of a well-designed OO structure?
  - In terms of the concurrency-control features?

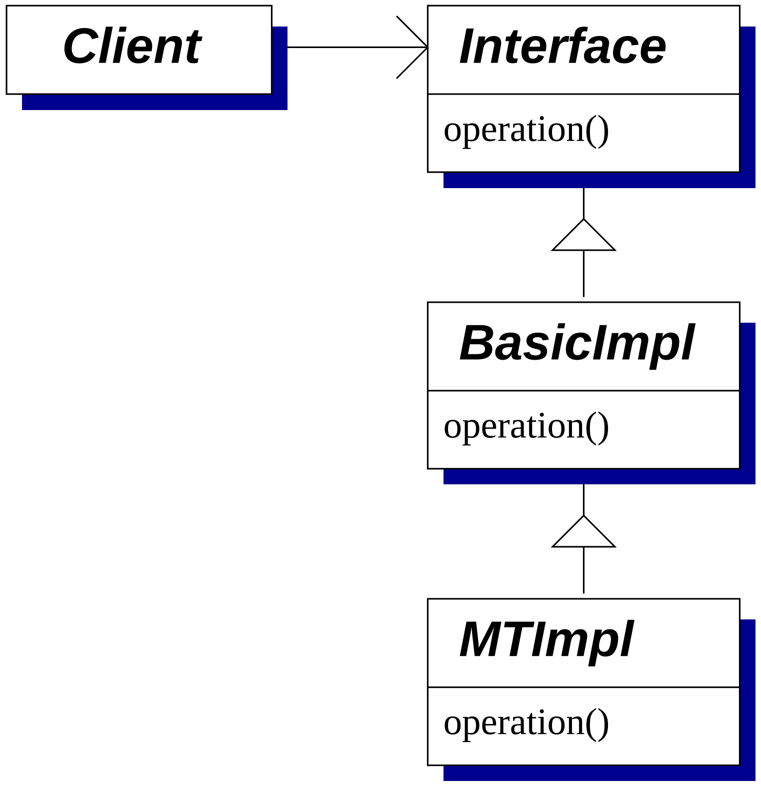
One option is based on delegation and the Adapter pattern:



- `BasicImpl` provides the actual data structure implementation, conforming to `Interface`. The class `MTImpl` wraps each operation with appropriate code for its use in a multi-threaded application, delegating calls to an instance of `BasicImpl`.

Design (2)

➤ How does that compare with:



Advantages

➤ Sub-classes enforce encapsulation and mean that only one instance is needed.

➤ Delegation may be easier; just use `super.operation()`.

Disadvantages

➤ Separate sub-classes are needed for each implementation of `Interface`.

➤ Composition of wrappers is fixed at compile time.

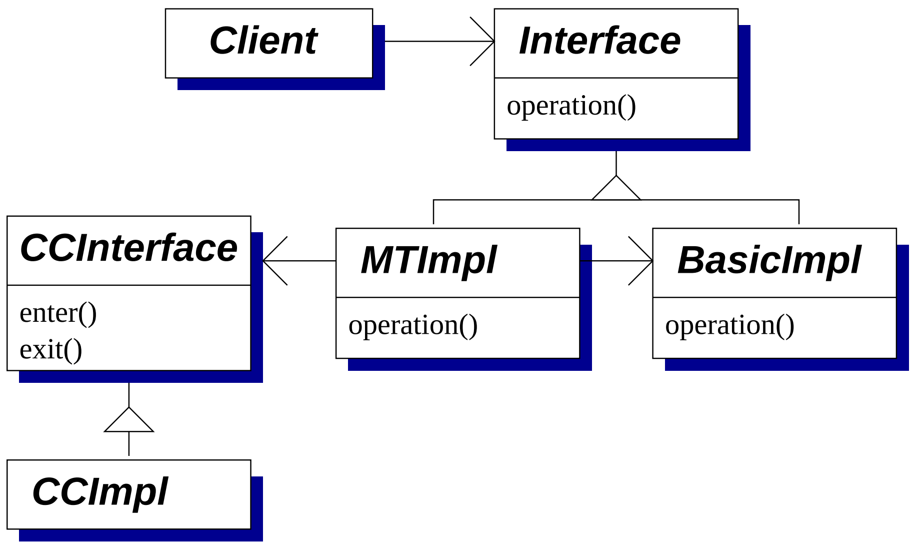
Design (3)

In each of these cases the class `MTImpl` will define methods that can be split into three sections.

1. An entry protocol responsible for concurrency control—usually waiting until it is safe for the operation to continue.
2. Delegation to the underlying data structure implementation (either by an ordinay method invocation on an instance of `BasicImpl` or a call using the `super` keyword).
3. An exit protocol—generally selecting the next thread(s) to perform operations on the structure.

This common structure often motivates further separation of concurrency control protocols from the data structure.

Design (4)



`MTImpl` now just deals with delegation, wrapping each invocation on `Interface` with appropriate calls to `enter()` and `exit()` on a general concurrency-control interface (`CCInterface`).

Sub-classes, e.g. `CCImpl`, provide specific entry/exit protocols. A factory class might be used to instantiate and assemble these objects.

Advantages

- Concurrency-control protocols can be shared.
- Only a single `MTImpl` class is needed per data structure interface.

Multiple readers, single writer (1)

- As a more concrete example:

```
interface MRSW {
  public void enterReader()
    throws InterruptedException;
  public void enterWriter()
    throws InterruptedException;
  public void exitReader();
  public void exitWriter();
}
```

- This could be used as:

```
class MTHashtable implements Dictionary {
  ...
  Object get(Object key) {
    Object result;
    cc.enterReader();
    try {
      result = ht.get(key);
    } finally {
      cc.exitReader();
    }
  }
}
```

- Why is `try...finally` used like this? How should `InterruptedException` be managed?

Multiple readers, single writer (2)

- We'll now look at implementing an example protocol, MRSW.

```
class MRSWImpl1 implements MRSW {
  private int numReaders = 0;
  private int numWriters = 0;
  ...
```

- A reader must wait until `numWriters` is zero. A writer must wait until both fields are zero.

```
  synchronized void enterReader()
    throws InterruptedException
  {
    while (numWriters > 0) wait();
    numReaders++;
  }

  synchronized void enterWriter()
    throws InterruptedException
  {
    while ((numWriters > 0) ||
           (numReaders > 0)) wait();
    numWriters++;
  }
```

Multiple readers, single writer (3)

The exit protocols are more straightforward:

```
  synchronized void exitReader() {
    numReaders--;
    notifyAll();
  }

  synchronized void exitWriter() {
    numWriters--;
    notifyAll();
  }
}
```

Advantage

- Simple design: (1) create a class containing the necessary fields; (2) write entry protocols that keep checking these fields and waiting; (3) write exit protocols that cause any waiting threads to assess whether they can continue.

Disadvantage

- `notifyAll()` may cause too many threads to be woken—the code is safe but might be inefficient.

Is that inefficiency likely to be a problem?

Could `notify()` be used instead?

Giving writers priority

- ...how else could `MRSW` be implemented?

```
 1  class PrioritizedWriters implements MRSW {
 2    private int numReaders = 0;
 3    private int numWriters = 0;
 4    private int waitingWriters = 0;
 5
 6    synchronized void enterReader()
 7    throws InterruptedException {
 8      while ((numWriters>0)||(waitingWriters>0))
 9        wait();
10      numReaders++;
11    }
12
13    synchronized void enterWriter()
14    throws InterruptedException {
15      waitingWriters++;
16      while ((numWriters>0)||(numReaders>0))
17        wait();
18      waitingWriters--;
19      numWriters++;
20    }
21    ...
22  }
```

- What happens to instances of `PrioritizedWriter` if the code is interrupted at line 17?

First-come first-served ordering (1)

- Suppose now we want an ordinary lock that provides FCFS semantics—the longest waiting thread is given access next.

```
class FCFSImpl implements CCInterface {
  private int currentTurn = 0;
  private int nextTicket = 0;
```

- Threads take a ticket and wait until it becomes their turn:

```
  synchronized void enter()
    throws InterruptedException
  {
    int myTicket = nextTicket++;
    while (currentTurn < myTicket)
      wait();
  }

  synchronized void exit() {
    currentTurn++;
    notifyAll();
  }
}
```

First-come first-served ordering (2)

Advantages

- The implementation is simple!

Disadvantages

- If a thread is interrupted during `wait()` then its ticket is lost.
- `notifyAll()` will wake all threads waiting in `enter()` on this object—in this case we know that only one can continue.
- What happens if the program runs for a long time and `nextTicket` overflows?

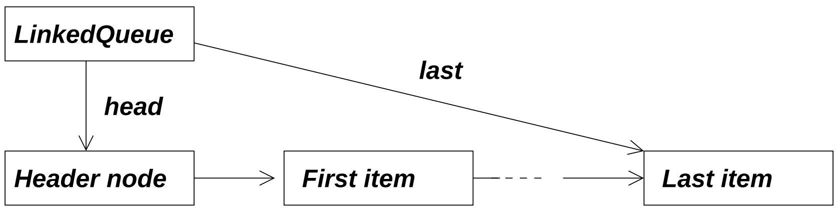
Resolving these issues in an effective way depends on the context in which the class is being used, e.g.

- lots of waiting threads and frequent contention: have an explicit queue of per-thread objects and use `notify()` on the object at the head of the queue;
- safe with arbitrary interruption: allow the `enter()` method to manage aborted waiters, e.g. using a queue as above with an `abandoned` field in each entry;
- no undetected failures: would `long`s ever overflow here?

Splitting locks (1)

Our last example shows how instances of `java.lang.Object` can be used to create separate locks to protect different parts of a data structure.

- This technique is generally useful to obtain more concurrency:
  - e.g. different locks to protect different operations that are safe concurrently,
  - e.g. in a list, 'push on tail' is usually safe to run in parallel with 'pop from head'.



We will use one lock to protect the `head` field, one for the `last` field, and then separate locks for each entry in the list protecting its `next` field.

- Do this after seeing that a lack of concurrency is a problem; the code is rarely as clear and often is wrong :-)
- More of this in the Part II course "Advanced Systems Topics"

Splitting locks (2)

```
// Based on 2.4.2.4 in Doug Lea’s book
class LinkedQueue {
  Node headNode = new Node(null);
  Node lastNode = headNode;
  Object headField = new Object();
  Object lastField = new Object();

  public void pushTail(Object x) {
    Node n = new Node(x);
    synchronized (lastField) {
      synchronized (lastNode) {
        // insert after last
        // and update last
      }
    }
  }

  public Object popHead() {
    synchronized (headField) {
      synchronized (headNode) {
        // read value from the
        // node after head and
        // make that node the
        // new head
      }
    }
  }
}
```

Exercises

1. In the `PrioritizedWriters` example the `waitingWriters` field is supposed to be a count of the number of threads executing in the body of the `enterWriter` method. How can this invariant be broken? Correct the code.
2. Update the `FCFSImpl` class so that it
   - (i) allows threads to safely be interrupted during `wait()`;
   - (ii) uses `notify()` instead of `notifyAll()`;
   - (iii) will not suffer from the ticket counter overflowing;

   How does the performance of your new implementation compare with that of the basic version?
3. Update the `FCFSImpl` class so that the lock is re-entrant—a thread already holding the lock can make subsequent calls to `enter()` without blocking. The lock is released only when a matched number of (properly-nested) calls to `exit()` have been made.
4. Complete the implementation of the `LinkedQueue` class by giving a suitable definition for `Node` and filling in the missing code in `pushTail` and `popHead`. Can you write `pushHead`? What about `popTail`?

Low-level synchronization

Previous lecture

- Integrating concurrency control
- Several examples: MRSW, FCFS.
- General design methods for other cases

Overview of this lecture

- Implementing mutexes and condition variables
- Direct scheduler support
- Semaphores
- Event counts / sequencers
- Alternative language features

Implementing mutexes and condvars

- Nowadays mutexes and condvars are usually implemented using a combination of:
  - operations provided by the scheduler to suspend and resume threads;
  - atomic assembly language instructions, e.g. compare-and-swap

  ```
  seen=CAS(addr,old,new)
  ```

  Read from address addr, if it matches `old` then store `new` at that address. Return the value `seen`.
- Care is needed to avoid the problems seen with Java's `Thread.suspend()` and `Thread.resume()` methods.
- Some implementations provide "lower-level" primitives and build mutexes and condvars over these:
  - semaphores
  - event counts and sequencers

  This layering is no longer typical, although we will still briefly look at these other primitives.

Implementing mutexes (1)

- Using `CAS` we can build a simple spin-lock:

```
class Mutex {
  int lockField = 0;

  void lock() {
    while (CAS(&lockfield,0,1) != 0) {
      /* someone else has the lock */
    }
  }

  void unlock() {
    lockField = 0;
  }
}
```

- Many performance problems: most importantly the `lock` operation consumes CPU time while waiting.
- Also, if multiple threads are waiting, then the data-cache line holding `lockField` will bounce between different CPUs on a multi-processor machine.
  - More about cache implications in the Part II course "Advanced Systems Topics".

Implementing mutexes (2)

- To avoid spinning, each mutex usually has a queue of blocked threads associated with it.
- A thread attempts to acquire the lock directly (e.g. using `CAS`), if it succeeds then it is done.
- If it doesn't succeed then it adds itself to the queue and invokes a **`suspend`** operation on the scheduler.
- After releasing the lock, a thread checks whether the queue is empty.
- If the queue is non-empty the thread selects an entry and **`resume`**s it. To avoid lost wake-up problems, either outstanding **`resume`** operations must be remembered:

```
1    Thread A, LOCK():        Thread B, UNLOCK():
2    see that lock is held
3    add A to queue
4                             release lock
5                             take A from queue
6                             resume A
7    suspend A
```

...or the scheduler should support a "disable thread switches" operation.

Implementing condvars (1)

- Condition variables are more intricate but can build on very similar techniques.
- Recall that a condition variable in general supports two operations:
  - a cv.CVWait(m) operation causes the current thread to atomically release a lock on mutex `m` and to block itself on condition variable `cv`, re-acquiring the lock on `m` when it is woken; and
  - a cv.CVNotify() operation that causes threads blocked on `cv` to continue.
- Internally, in a typical implementation, each condvar has private fields that hold:
  - a queue of threads that are waiting on the condition variable; and
  - an additional mutex `cvLock` that is used to give the atomicity required by `CVWait()`.

Implementing condvars (2)

- `cv.CVWait(m)` proceeds by:

```
Acquire mutex cv.cvLock
  Add the current thread to cv.queue
  Release mutex m
Release mutex cv.cvLock
Suspend current thread
Re-acquire mutex m
```

- `cv.CVNotify()` proceeds by:

```
Acquire mutex cv.cvLock
  Remove one thread from cv.queue
  Resume that thread
Release mutex cv.cvLock
```

- Again, it is important to avoid lost wake-up problems—typically by remembering resumptions.
- A real implementation is more complex—e.g. in Java it is necessary to deal with threads being interrupted.
  - See `linuxthreads/condvar.c` for a Linux implementation.

Semaphores

- These examples have used the language-level `mutexes` and condition variables.
- Semaphores provide basic operations on which the language-level features could be built. In Java-style pseudo-code:

```
class CountingSemaphore {
  CountingSemaphore (int x) {
    ...
  }

  native void P();
  native void V();
}
```

- `P` (sometimes called wait) decrements the value and then blocks if the result is less than zero.
- `V` (sometimes called signal) increments the value and then, if the result is zero or less, selects a blocked thread and unblocks it.
- Using semaphores directly is intricate—the programmer must ensure `P()` / `V()` are paired correctly.

Programming with semaphores (1)

- Typically the integer value of a counting semaphore is used to represent the number of instances of some resource that are available, e.g.

```
class Mutex {
  CountingSemaphore sem;

  Mutex() {
    sem = new CountingSemaphore(1);
  }

  void acquire() {
    sem.P();
  }

  void release () {
    sem.V();
  }
}
```

- The mutex is considered unlocked when the value is 1 (it is initialized unlocked)...
- ...and is locked when the value is 0 or less.
- How does this mutex differ from a Java-style one?

Programming with semaphores (2)

```
class CondVar {
  int numWaiters = 0;
  Mutex cv_lock = new Mutex();
  CountingSemaphore cv_sleep =
    new CountingSemaphore (0);

  void CVWait(Mutex m) {
    cv_lock.acquire();
    numWaiters++;
    m.release();
    cv_lock.release();
    cv_sleep.P();
    m.acquire();
  }

  void CVNotify() {
    cv_lock.acquire();
    if (numWaiters > 0) {
      cv_sleep.V();
      numWaiters--;
    }
    cv_lock.release();
  }
}
```

Event counts and sequencers

A further style of concurrency control is presented by event count and sequencer primitives.

- An event count is represented by a positive integer, initialized to zero, supporting the following atomic operations:
  - `advance()`—increment the value by one, returning the new value;
  - `read()`—return the current value; and
  - `await(i)`—wait until the value is greater than or equal to `i`.
- A sequencer is again represented by a positive integer, initialized to zero, supporting a single atomic operation:
  - `ticket()`—increment the value by one, returning the old value;
- Mutual exclusion is easy: a thread takes a ticket entering a critical region and invokes `await()` to receive its turn (c.f. `FCFSImpl`).
- The values returned by `await()` can be used directly in implementing a single-producer single-consumer N-slot buffer: they give the modulo-N indices to read/write.

Mutexes without hardware support

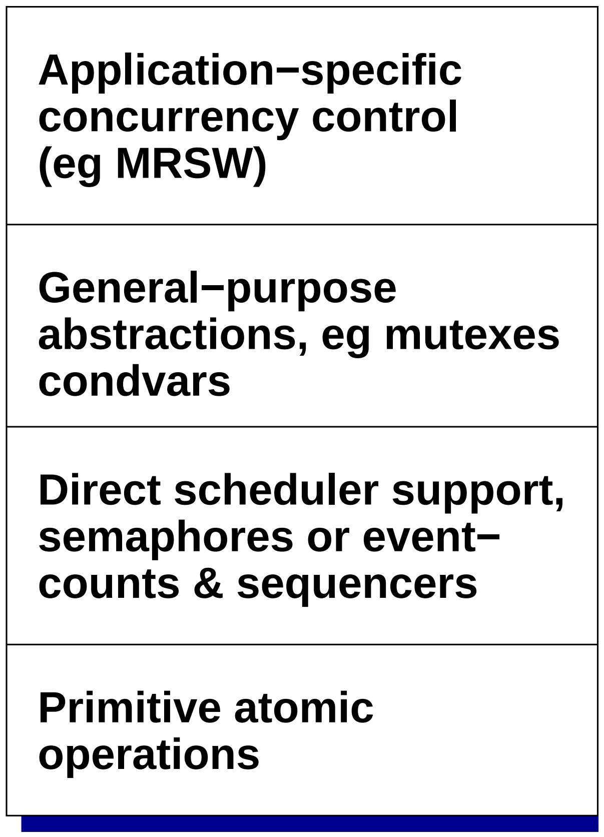
- What can we do if there isn't a `CAS` or `TAS` instruction, just atomic read and write? (e.g. the ARM7 only has a `swap` operation)
- The 'Bakery' algorithm due to Lamport (1974)—this algorithm is now an example: not for practical use!

```
enter()     taking[i] = true;
            ticket[i]=max(ticket[0],..., ticket[n−1])+1
            taking[i] = false;
            for (j=0; j<i; j++) {                          1
              while (taking[j]) { }
              while ((ticket[j] != 0) &&
                        (ticket[j] <= ticket[i])) { }
            }
            for (j=i; j<n; j++) {                          2
              while (taking[j]) { }
              while ((ticket[j] != 0) &&
                        (ticket[j] < ticket[i])) { }
            }

exit()      ticket[i] = 0;
```

- Threads enter the critical region in ticket order, using their IDs (`i`) as a tie-break.

Recap


Application−specific concurrency control (eg MRSW)
General−purpose abstractions, eg mutexes condvars
Direct scheduler support, semaphores or event−counts & sequencers
Primitive atomic operations

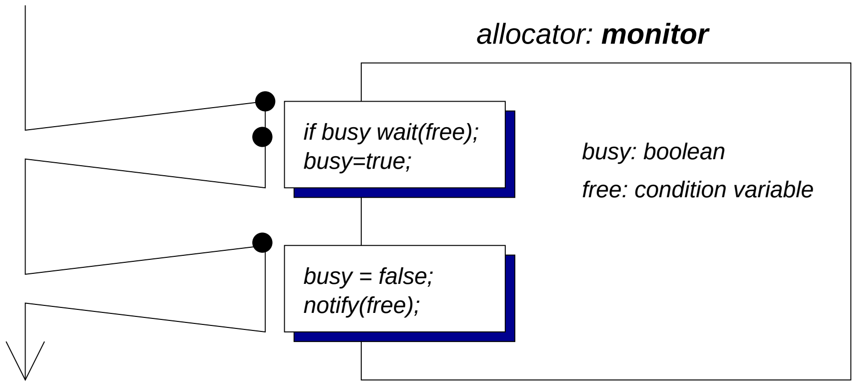
The details of exactly what is implemented where vary greatly between systems, e.g.

- whether the thread scheduler is implemented in user-space or in the kernel,
- which synchronization primitives can be used between address spaces.

Similarly, unless the application builds it, FCFS semantics and fairness are rarely guaranteed.

Alternative language features (1)

A monitor is an abstract data type in which mutual exclusion is enforced between invocations of its operations. Often depicted graphically showing the internal state and external interfaces, e.g. in pseudo-code



When looking at a definition such as this, independent of a specific programming language, it is important to be clear on what semantics are required of `wait()` and `notify()`:

- Does `notify` wake at most one, exactly one, or more than one waiting thread?
- Does `notify` cause the resumed thread to continue immediately (if so, must the notifier exit the monitor)?

Alternative language features (2)

An active object achieves mutual exclusion between operations by (at least conceptually) having a dedicated thread that performs them on behalf of external callers, e.g.

```
loop
  SELECT
    when count < buffer-size
      ACCEPT insert(param) do
        [insert item into buffer]
      end;
    increment count;
    [manage ref to next slot for insertion]

    or when count > 0
      ACCEPT remove(param) do
        [remove item from buffer]
      end;
    decrement count;
    [manage ref to next slot for removal]
  end SELECT
end LOOP
```

- Guarded `ACCEPT` statements provide operations and pre-conditions that must hold for their execution.
- Management code occurs outside the `ACCEPT` statements.

Exercises (1)

1. Using the `CountingSemaphore` class (and not the `synchronized` keyword) implement a sequencer. The sequencer should hold a single positive number, initialized to zero, and support an atomic operation `ticket()` which increments the value by one and returns the old value.
2. Using the example `EventCount` and `Sequencer` classes, implement a single-cell buffer supporting an arbitrary number of producers and consumers, but holding only a single value at once.
3. A binary semaphore is a simplified version of the counting semaphore from the slides. Rather than an integer count value it has a binary flag. P_b blocks (if necessary) until the flag is set and then atomically clears it. V_b sets the flag (atomically unblocking one thread, if any blocked in P_b on that semaphore).
   - (i) In pseudo-code, show how a binary semaphore can be built using atomic compare-and-swap (`CAS`) or test-and-set (`TAS`) machine instructions.
   - (ii) In pseudo-code, show how a counting semaphore can be built using binary semaphores. Your solution might need more than one binary semaphore and another field to hold the count value.

Exercises (2)

4. Some data structures can be implemented directly using the `CAS` primitive without needing mutual exclusion locks. Suppose that a Java-like language supports a `CAS` operation on fields. Show how a single-ended queue could be defined (implemented using a singly-linked list) supporting `push` and `pop` operations at the head of the queue.