G52CON: Concepts of Concurrency

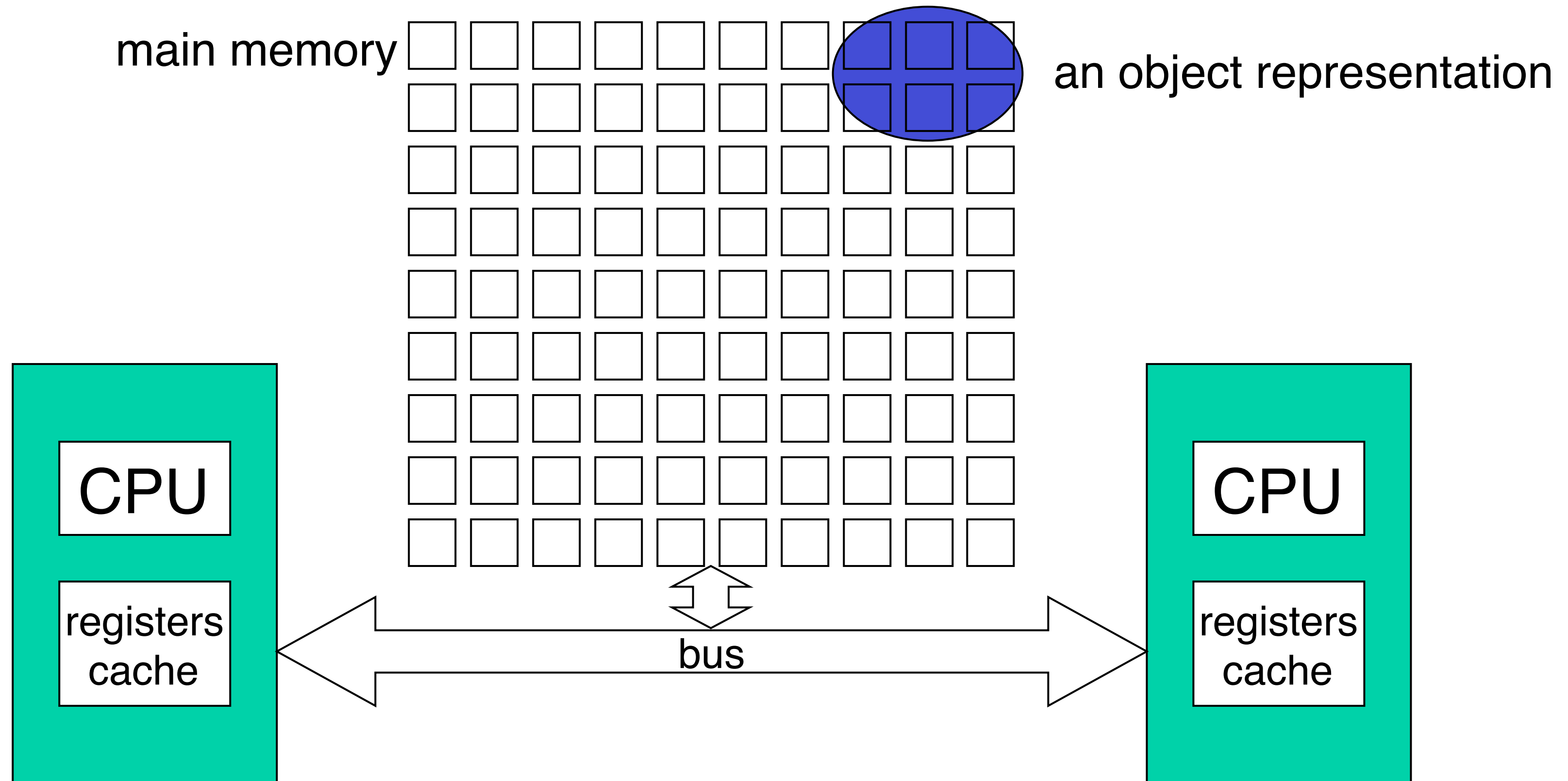
Lecture 11 Synchronisation in Java

Brian Logan
School of Computer Science
bsl@cs.nott.ac.uk

Outline of this lecture

- mutual exclusion in Java
- condition synchronisation in Java
- monitors
  - example: BoundedBuffer monitor and Java
- exercise: semaphores in Java

Java Memory Model



Working memory

Java allows threads that access shared variables to keep private working copies of variables:

- each thread is defined to have a *working memory* (an abstraction of caches and registers) in which to store values;

- this allows a more efficient implementation of multiple threads

Model properties

The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*:

- **atomicity**: which instructions must have indivisible effects
- **visibility**: under what conditions are the effects of one thread visible to another; and
- **ordering**: under what conditions the effects of operations can appear out of order to any given thread.

Unsynchronized code

- **atomicity**: reads and writes to memory cells corresponding to fields of any type *except* `long` or `double` are guaranteed to be atomic
- **visibility**: changes to fields made by one thread are not guaranteed to be visible to other threads
- **ordering**: from the point of view of other threads, instructions may appear to be executed out of order

`volatile` fields

If a field is declared `volatile`, a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

- reads and writes to a `volatile` field are guaranteed to be atomic (even for `long`s and `double`s);
- new values are immediately propagated to other threads; and
- from the point of view of other threads, the relative ordering of operations on `volatile` fields are preserved.

However the ordering and visibility effects surround only the single read or write to the `volatile` field itself, e.g, '++' on a `volatile` field is not atomic.

Mutual exclusion in Java

Java provides built-in support for mutual exclusion with the `synchronized` keyword:

- both methods and blocks can be `synchronized`
- each object has a *lock* (inherited from class `Object`)
- when a `synchronized` method (or block) is called, it waits to obtain the lock, executes the body of the method (or block) and then releases the lock
- allows the implementation of coarse grained atomic actions

Invoking `synchronized` methods

When a thread invokes a `synchronized` method `foo()` on an object `x`, it tries to obtain the lock on `x`

- if another thread already holds the lock on `x`, the thread invoking `foo()` blocks
- when it obtains the lock, it executes the body of the method and then releases the lock, even if the exit occurs due to an exception
- when one thread releases a lock, another thread may acquire it (perhaps the same thread, if it invokes another `synchronized` method)—there are no guarantees about which thread will acquire a lock next or if a thread will ever acquire a lock
- locks are *reentrant*, i.e., per thread, not per invocation—a `synchronized` method can invoke another `synchronized` method on the *same* object without deadlocking

`synchronized` methods

- the `synchronized` keyword is not part of a method's signature, and is not automatically inherited when subclasses override superclass methods
- methods in interfaces and class constructors cannot be declared `synchronized`
- `synchronized` instance methods in subclasses use the same lock as methods in their superclasses

`synchronized` blocks

Block synchronization is lower-level than method synchronization:

- synchronized methods synchronize on an instance of the method's class (or the `Class` object for `static` methods)
- block synchronization allows synchronization on *any* object
- this allows us to narrow the scope of a lock to only part of the code in a method
- also allows us to use a different object to implement the lock

Synchronized code

- **atomicity**: changes made in one `synchronized` method (or block) are atomic with respect to other `synchronized` methods (blocks) on the *same* object

- **visibility**: changes made in one `synchronized` method (or block) are visible with respect to other `synchronized` methods (blocks) on the *same* object

- **ordering**: order of `synchronized` calls is preserved from the point of view of other threads

A Simple Example: `ParticleApplet`

`ParticleApplet` creates *n* `Particle` objects, sets each particle in autonomous 'continuous' motion, and periodically updates the display to show their current positions:

- each `Particle` runs in its own Java `Thread` which computes the position of the particle; and
- an additional `ParticleCanvas` `Thread` periodically checks the positions of the particles and draws them on the screen.
- in this example there are at least 12 threads and possibly more, depending on how the browser handles applets.

ParticleApplet

There are three classes:

- `Particle`: represents the position and behaviour of a particle and can draw the particle at its current position;

- `ParticleCanvas`: provides a drawing area for the `Particles`, and periodically asks the `Particles` to draw themselves; and

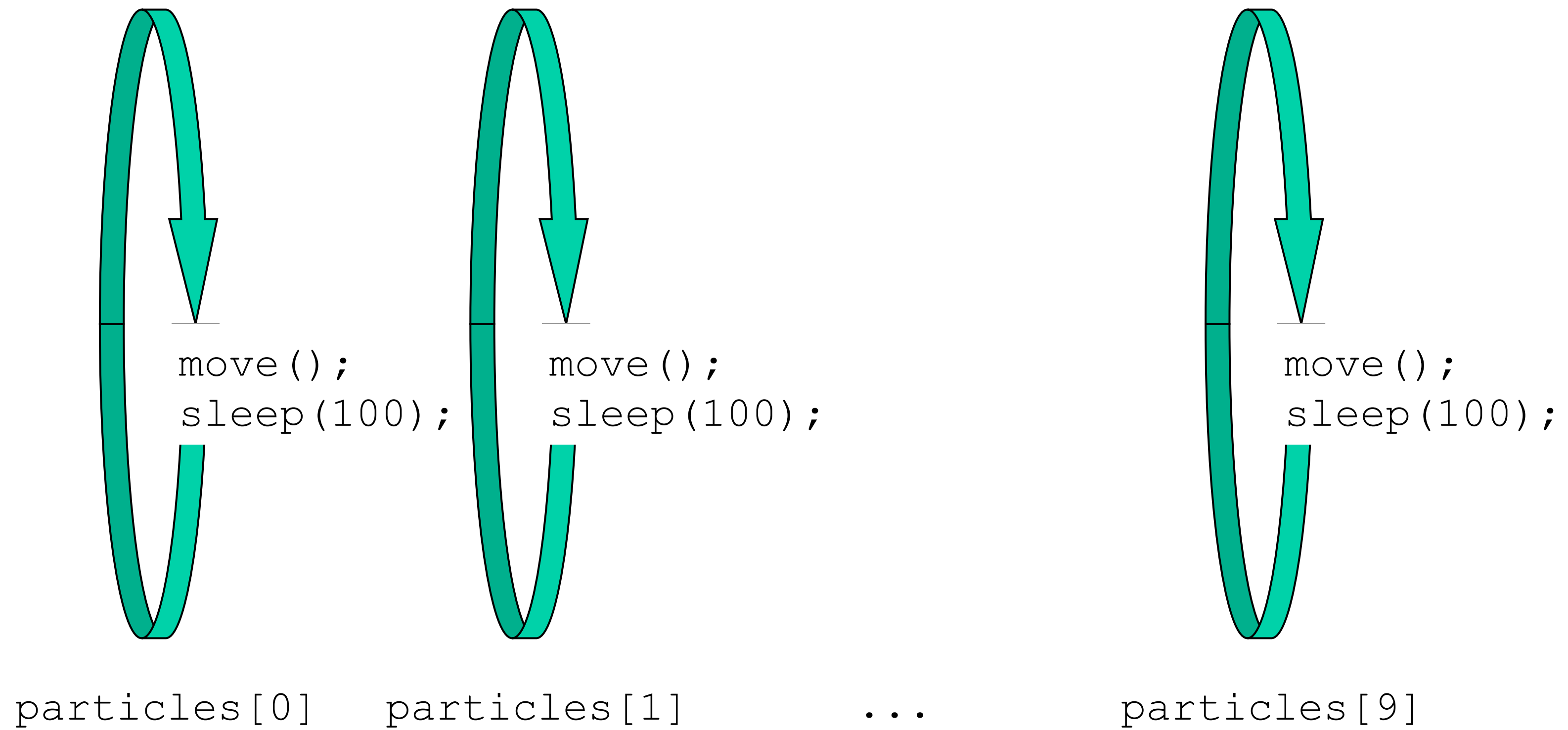
- `ParticleApplet`: creates the `Particles` and the canvas and sets the `Particles` in motion.

See also Lea (2000), chapter 1 for an alternative implementation.

Particle.run()

```
class Particle extends Thread {
    // fields, constructor etc…
    public void run() {
        try {
            for(;;) {
                move();
                sleep(100);
            }
        }
        catch (InterruptedException e) { return; }
    }
    // other methods …
}
```

`Particle` threads



ParticleCanvas.run()

```
class ParticleCanvas extends Canvas implements Runnable {
    // fields, constructor etc …
    public void run() {
        try {
            for(;;) {
                repaint();
                Thread.sleep(100);
            }
        }
        catch (InterruptedException e) { return; }
    }
    // other methods …
}
```

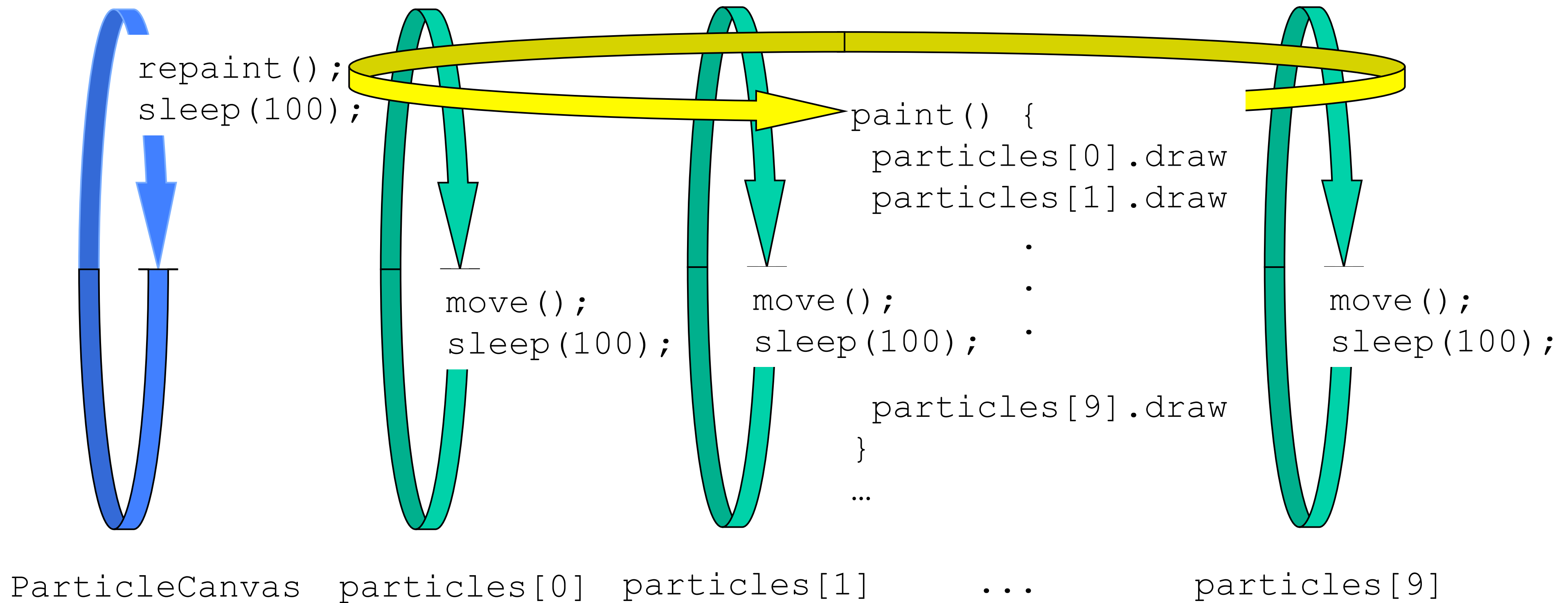
`ParticleCanvas` class continued

```
    protected synchronized void getParticles() {
        return particles;
    }

    // called by Canvas.repaint();
    public void paint(Graphics g) {
        Particle[] ps = getParticles();

        for (int i = 0; i < ps.length(); i++)
            ps[i].draw(g);
    }
}
```

ParticleCanvas & AWT event threads



Interference in the `ParticleApplet`

- need to ensure that `draw()` doesn't see an incompletely updated `x`, `y` position of a particle, e.g.
  - JVM runs a `Particle` thread which invokes `move()` which increments `x`
  - JVM then switches to running the `ParicleCanvas` thread which invokes `draw()` which sees the updated `x` position but the *old* `y` position
  - JVM switches back to running the `Particle` thread—`move()` completes, updating the `y` position of the particle
- `Particle` object is drawn in a position it never occupied

Synchronising `Particle.move`

- we can avoid interference by making access to the `x`, `y` position of a particle a critical section
- we use `synchoronized` to enforce mutual exclusion

```
// Particle move method
public synchronized void move() {
     x += (rng.nextInt() % 10);
     y += (rng.nextInt() % 10);
}
```

Synchronising `Particle.draw`

```
// Particle draw method
public void draw(Graphics g) {
    int lx, ly;
    synchronized (this) {
        lx = x;
        ly = y;
    }
    g.drawRect(lx, ly, 10, 10);
}
```

Condition synchronisation in Java

Java provides built-in support for condition synchronisation with the methods `wait()`, `notify()` and `notifyAll()`:

- to delay a thread until some condition is true, write a loop that causes the thread to `wait()` (block) if the delay condition is false
- ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.
- Java uses a Signal and Continue signalling discipline

`wait()`, `notify()` and `notifyAll()`

- `wait()` , `notify()` and `notifyAll()` must be executed within `synchronized` methods or blocks
- `wait()` releases the lock on the object held by the calling thread–the thread blocks and is added to the *wait set* for the object
- `notify()` wakes up *a thread* in the wait set (if any); `notifyAll()` wakes up *all threads* in the wait set (if any)
  - the thread that invoked `notify()` / `notifyAll()` continues to hold the object's lock
  - the awakened thread(s) remain blocked and execute at some future time when they can reacquire the lock

`notify()` **vs** `notifyAll()`

- `notify()` can be used to increase performance when only *one* thread needs to be woken:
  - all threads in the wait set are waiting on the *same* delay condition and
  - each notification enables at most one thread to continue and
  - the possibility of an `interrupt()` during `notify()` is handled (pre Java 5)

- `notifyAll()` is required when:
  - the threads in the wait set are waiting on different conditions; or
  - a notification can satisfy multiple waiting threads

Waking the right process

```
class TicketCounter {
    long ticket = 0;
    long turn = 0;

    synchronised takeTicket() throws InterruptedException {
        long myTurn = ticket++;
        while (myTurn != turn)
            wait();
    }

    synchronised nextTurn() {
        turn++;
        notifyAll();  // wakes all processes
    }
}
```

Loss of notification

```
class TicketCounter {  // broken - do not use ...
    long ticket = 0;
    long turn = 0;

    synchronised takeTicket() throws InterruptedException {
        long myTurn = ticket++;
        while (myTurn != turn)
            wait();
    }

    synchronised nextTurn() {
        turn++;
        notify();  // wakes an arbitrary process
    }
}
```

Interrupting a `Thread`

Each `Thread` object has an associated boolean interruption status:

- `interrupt()`: sets the thread's interrupted status to *true*
- `isInterrupted()`: returns *true* if the thread has been interrupted by `interrupt()`

A thread can periodically check its interrupted status, and if it is *true*, clean up and exit.

Handling interrupts

Threads which are blocked in calls `wait()`, `sleep()` and `join()` aren't runnable, and can't check the value of the interrupted flag

- interrupting a thread which is not runnable aborts the thread, throws an `InterruptedException` and sets the thread's interrupted status to *false*
- calls to `wait()`, `sleep()`, or `join()` are often enclosed in a `try catch` block:

```
synchronized <method or block>
    try {
        wait()|sleep(millis)|join(millis)
    } catch (InterruptedException e) {
        // clean up and return
    }
```

Approaches to mutual exclusion

- **Mutual exclusion algorithms:** pre- and post-protocols are implemented using special machine instructions or atomic memory accesses (e.g., Test-and-Set, Peterson's algorithm)
- **Semaphores:** pre- and post-protocols can be implemented using atomic ***P*** and ***V*** operations
- **Monitors:** mutual exclusion is *implicit*---pre- and post-protocols are executed automatically on entering and leaving the monitor to ensure that monitor procedures are not executed concurrently.

Implementing mutual exclusion in Java

- it is difficult to implement mutual exclusion algorithms in Java, due to problems of visibility, ordering, scheduling and efficiency
- we can implement semaphores as a Java class with methods which implement the P and V operations (see java.util.concurrent)
- monitors are the basis of Java's synchronisation primitives–there is a straightforward mapping from designs based on monitors to solutions using synchronized classes

Approaches to condition synchronisation

- **Busy-waiting:** the process sits in a loop until the condition is true
- **Semaphores:** ***P*** and ***V*** operations can be used to wait for a condition and to signal that it has occurred
- **Monitors:** condition synchronisation is explicitly programmed using *condition variables* and monitor operations, e.g., wait and signal.

Implementing condition synchronisation in Java

- it is difficult to implement busy waiting in Java, due to problems of visibility, scheduling and efficiency

- we can implement semaphores as a Java class with methods which implement the P and V operations (see `java.util.concurrent`)

- while monitors are the basis of Java's synchronisation primitives, each object in Java has a only a single condition variable and delay queue—monitor operations can be implemented using `wait()` and `notify()`, and `notifyAll()`

© Brian Logan 2014

Example: monitors

A *monitor* is an abstract data type representing a shared resource.

Monitors have four components:

- a set of *private variables* which represent the state of the resource;
- a set of *monitor procedures* which provide the public interface to the resource;
- a set of *condition variables* and associated *monitor operations* (**`wait`**`()`, **`signal`**`()`, **`signal_all`**`()`) used to implement condition synchronisation; and
- *initialisation code* which initialises the private variables.

Monitors in Java

Monitors can be implemented as Java classes:

- the monitor's private variables are `private` fields in a class
- the monitor procedures are implemented using `synchronized` methods — `synchronized` methods are executed under mutual exclusion with all other `synchronized` methods on the same object
- condition synchronisation is implemented using `wait()`, `notify()`, `notifyAll()`
  - each object has a single (implicit) condition variable and delay queue, the *wait set*
  - Java uses a *signal and continue* signalling discipline

Bounded buffer with monitors

```
monitor BoundedBuffer {
    // Private variables …
    Object buf = new Object[n];
    integer out = 0,        // index of first full slot
            in = 0,         // index of first empty slot
            count = 0;      // number of full slots

    // Condition variables ...
    condvar not_full,       // signalled when count < n
            not_empty;      // signalled when count > 0

    // continued ...
```

Bounded buffer with monitors 2

```
// Monitor procedures ...
//(signal & continue signalling discipline)
procedure append(Object data) {
    while(count == n) {
        wait(not_full);
}
buf[in] = data;
in = (in + 1) % n;
count++;
signal(not_empty);
}

// continued ...
```

Bounded buffer with monitors 3

```
    procedure remove(Object &item) {
        while(count == 0) {
            wait(not_empty);
        }
        item = buf[out];
        out = (out + 1) %n;
        count--;
        signal(not_full);
    }
}
```

Bounded buffer in Java

```
class BoundedBuffer {
    // Private variables …
    private Object buf;
    private int out = 0,   // index of first full slot
    private int in = 0,    // index of first empty slot
    private int count = 0; // number of full slots

    public BoundedBuffer(int n) {
        buf = new Object[n];
    }

    // continued ...
```

Bounded buffer in Java 2

```
// Monitor procedures …
public synchronized void append(Object data) {
    try {
        while(count == n) {
            wait();
        }
    catch (InterruptedException e) {
        return;
    }
    buf[in] = data;
    in = (in + 1) % n;
    count++;
    notifyAll();
    }
```

Bounded buffer in Java 3

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

Example: semaphores

A semaphore s is an integer variable which can take only non-negative values. Once it has been given its initial value, the only permissible operations on s are the atomic actions:

P(s) : if $s > 0$ then $s = s - 1$, else suspend execution of the process that called ***P***(s)

V(s) : if some process p is suspended by a previous ***P***(s) on this semaphore then resume p, else $s = s + 1$

A *general semaphore* can have any non-negative value; a *binary semaphore* is one whose value is always 0 or 1.

Exercise: semaphores in Java

```
class GeneralSemaphore {
    private long resource;

    public GeneralSemaphore (long r) {
        resource = r;
    }


    // method to implement the P operation



    // method to implement the V operation



}
```

© Brian Logan 2014

The Next Lecture

Synchronisation in Java II

Suggested reading:

- Lea (2000), chapter 3.