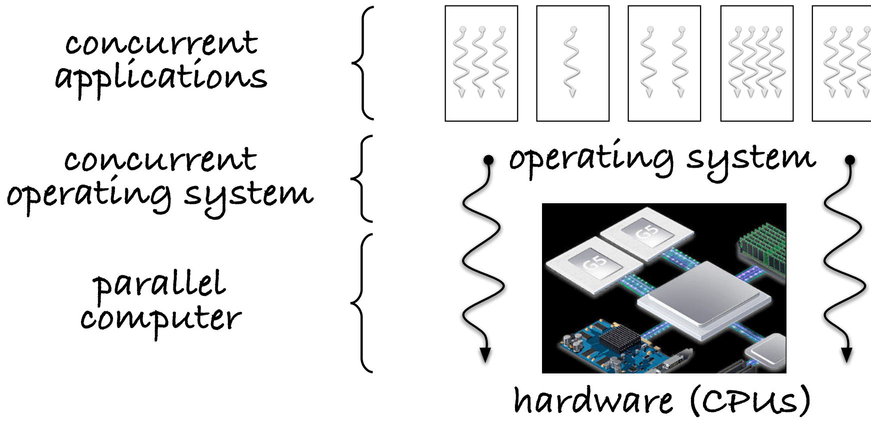
Concurrent Programming

Benoît Garbinato
distributed object programming lab

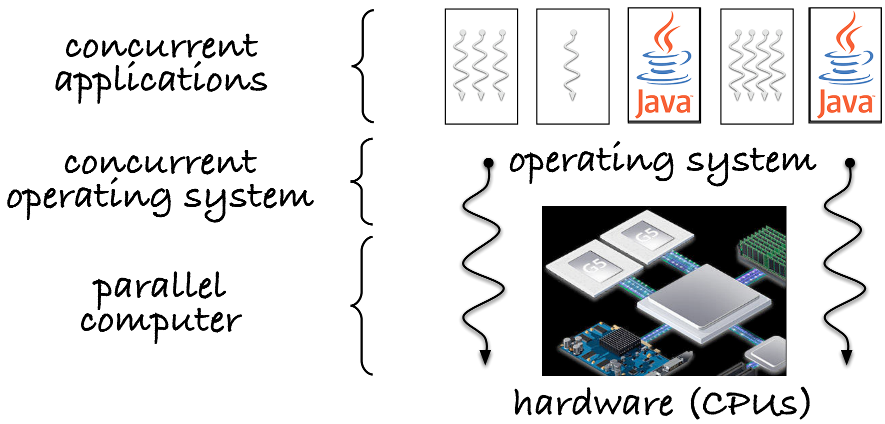
Processes & threads

- A process is a unit of execution managed at the level of the operating system
- Each process has its own address space, i.e., no other process can access it
- A thread is a sequential flow of control executing within a process
- All threads within a process share the same address space, i.e., they share memory

Concurrency & parallelism



Concurrency & parallelism



Pseudo vs. Quasi Parallelism

- With pseudo-parallelism, a thread can be interrupted by the system at any time (we say that the system is preemptive)
- With quasi-parallelism, a thread can only be interrupted voluntarily, either explicitly or when it performs a input/output system call

Liveness & safety

Safety : nothing bad ever happens

In object-oriented terms, this means that the state of no object will ever be corrupted

Liveness : something good eventually happens

In object-oriented terms, this means that the state of some object will eventually change

Creating threads in java

Extending `Thread`

```
public class PrimeThread extends Thread {
    long minPrime;

    public PrimeThread(long minPrime) {
        this.minPrime= minPrime;
    }

    public void run() {
        // Compute prime larger than minPrime
        ...
    }

    public static void main(String[] args) {
        PrimeThread pt= new PrimeThread(7);
        pt.start();
    }
}
```

Implementing `Runnable`

```
public class PrimeRun implements Runnable  {
    long minPrime;

    public PrimeRun(long minPrime) {
        this.minPrime= minPrime;
    }

    public void run() {
        // Compute prime larger than minPrime
        ...
    }

    public static void main(String[] args) {
        PrimeRun pr= new PrimeRun(7);
        new Thread(pr).start();
    }
}
```


Mutual exclusion

- The readers/writers problem is a typical mutual exclusion problem:
  - no reader should be allowed to read while a writer is writing
  - no writer should be allowed to write while either another writer is writing or a reader is reading

Readers/Writers

```
public class Data {
    private String name;
    private String phone;        } critical resources
    public void setName(String name) { this.name= name; }
    public String getName() { return name;}
    public void setPhone(String phone) { this.phone= phone; }
    public String getPhone() { return phone; }
}
```

```
data= new Data();
w1= new Writer(data, "James", "007");
w2= new Writer(data, "Devil", "666");
r1= new Reader(data);
r1.start(); w1.start(); w2.start();
```

```
public class Reader extends Thread {
    public Data data;

    public Reader(Data data) {
        this.data= data;
    }
    public void run() {
        while (true) {
critical {      System.out.print(data.getName());
section  {      System.out.println(data.getPhone());
        }
    }
}
```

```
public class Writer extends Thread {
    private Data data;
    private String name;
    private String phone;

    public Writer(Data data, String name,
                  String phone) {
        this.data= data;
        this.name= name; this.phone= phone;
    }
    public void run() {
        while (true) {
            data.setName(name);        } critical
            data.setPhone(phone);      } section
        }
    }
}
```


The concept of monitor

- A monitor is associated with an object to...
  - ...ensure mutual exclusion of its methods
  - ...explicitly suspend or wake up threads using that object
- In Java, each object has an associated monitor
- You have two ways to express mutual exclusion in Java:

At the method level

```
synchronized
public void setData(String name, String phone) {
    this.name= name;
    this.phone= phone;
}
```

At the object level

```
synchronized (data) {
    name= data.getName();
    phone= data.getPhone();
}
```

Readers/Writers revisited

```
public class Data {
    private String name;
    private String phone;
    public String getName() { return name;}
    public String getPhone() { return phone; }
    synchronized
    public void setData(String name, String phone)
        { this.name= name; this.phone= phone;}
}
```

```
public class Reader extends Thread {
    public Data data;

    public Reader(Data data) {
        this.data= data;
    }
    public void run() {
        while (true) {
            synchronized (data) {
                System.out.print(data.getName());
                System.out.print(data.getPhone());
            }
        }
    }
}
```

```
data= new Data();
w1= new Writer(data, "James", "007");
w2= new Writer(data, "Devil", "666");
r1= new Reader(data);
r1.start(); w1.start(); w2.start();
```

```
public class Writer extends Thread {
    private Data data;
    private String name;
    private String phone;

    public Writer(Data data, String name,
                  String phone) {
        this.data= data; this.name= name;
        this.phone= phone;
    }
    public void run() {
        while (true)
            data.setData(name,phone);
    }
}
```


Waiting & notifying

A monitor is associated to an object to explicitly suspend or wake up threads using that object.

```
public class Object {
    ...
    public final void wait() {...}
    public final void notify() {...}
    public final void notifyAll() {...}
    ...
}
```

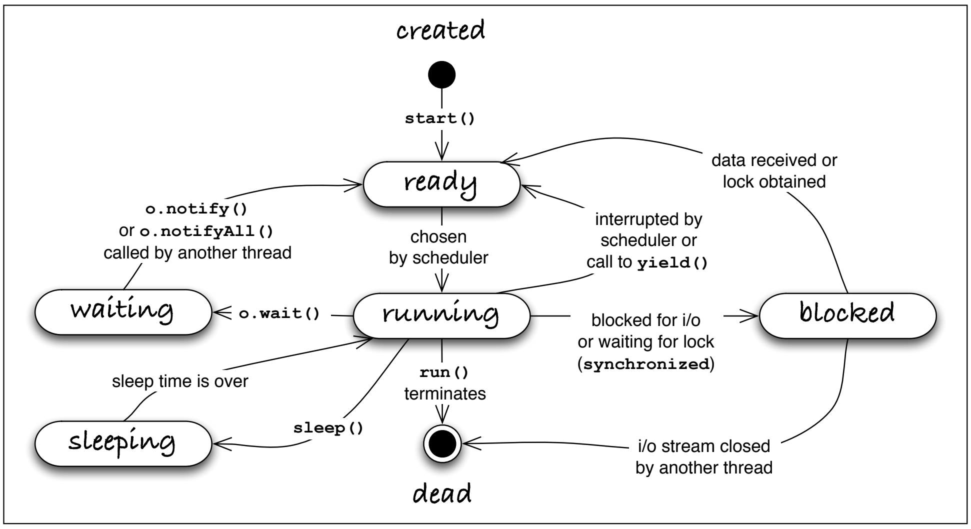
Using wait() and notify()

```
public static void main(String[] args) {
    final Object wham= new Object();
    Runnable singer= new Runnable() {
        public void run() {
            try {
                for (int i= 1; i<=100; i++)
                    synchronized (wham) { wham.wait(); }
            } catch (InterruptedException e) {}
        }
    };
    Thread george= new Thread(singer, "George");
    Thread andrew= new Thread(singer, "Andrew");
    george.start(); andrew.start();

    int i= 0;
    while (george.isAlive() || andrew.isAlive()) {
        synchronized (wham) {wham.notify();}
        i++;
    }
    System.out.println("\nI had to send " + i + " notifications.");
}
```

<u>Question</u>: how many times was notify() called ?

Lifecycle of a thread



Synchronization

- The producers/consumers problem is a typical synchronization problem:
  - Let S be a bounded buffer
  - A producer should only be allowed to produce as long as S is not full
  - A consumer should only be allowed to consume as long as S is not empty

Producer/Consumer

```
public class Producer extends Thread {
    private Stack shared;
   public Producer(Stack shared)
    { this.shared= shared;}

    public void run() {
        while (true)  {
            int d= ...;
            shared.push(d);
        }
    }
}
```

```
public class Consumer extends Thread {
    private Stack shared;
    public Consumer(Stack shared)
    { this.shared= shared; }

    public void run() {
        while (true) {
            int d= shared.pop();
            ...
        }
    }
}
```

```
public class Stack {
    private int[] data;
    private int i;
    public Stack(int size) {
        data= new int[size];
        i= 0;
    }
    synchronized
    public  void push(int d) {
        if (i == data.length) wait();
        data[i]= d;
        i++;
        notify();
    }
    synchronized
    public int pop() {
        if (i == 0 ) wait();
        i--;
        int d= data[i];
        notify();
        return d;
    }
}
```

Producers/Consumers

```
synchronized
public void push(int d) {
    while (i == data.length) wait();
    data[i]= d;
    i++;
    notifyAll();
}
```

```
synchronized
public int pop() {
    while (i == 0) wait();
    i--;
    int d= data[i];
    notifyAll();
    return d;
}
```

if → while transformation ensures safety

notify → notifyAll transformation ensures liveness

General pattern

ensures safety → while (condition not true) wait();

ensures liveness → notifyAll();

```
synchronized public void doSomething(...) {
    ....
    while ( condition not true ) wait();
    ...
    notifyAll();
}
```

A problematic scenario with while and notify (not notifyAll)

[S.size() == 1]

		1	2	3	4	5
Stack	S	empty	full	empty	full	full
Consumer	C1	state: ready [1] do: wait	state: ready [2] do: empty S do: notify P1 do: wait	state: wait	state: wait	state: wait
Producer	P1	state: ready [2] do: fill S do: notify C1 do: wait	state: wait	state: ready do: fill S do: notify P2 do: wait	state: wait	state: wait
Producer	P2	state: blocked [outside push]	state: ready [1] do: wait	state: wait	state: ready do: wait	state: wait

Limitations of monitors (1)

The monitor abstraction is somewhat too high-level. In particular, it is impossible to:

- acquire mutual exclusion if it is already granted, or give it up after a timeout or an interrupt
- acquire mutual exclusion in one method and release it in another method
- alter the semantics of mutual exclusion, e.g., with respect to reentrancy, reads vs. writes, or fairness

Limitations of monitors (2)

The monitor abstraction is somewhat too low-level. In particular, it provides no direct support for:

- Atomic variables (thread-safe single variables)
- Reader/writer and producer/consumer
- Highly concurrent collection classes

Concurrency utilities (JSE5)

- Design goals
  - Reduced programming effort
  - Increased performance & reliability
  - Improved maintainability & productivity
- Features
  - Task scheduling framework
  - Concurrent collections & atomic variables
  - Synchronizers & locks
  - Nanosecond-granularity
- Packages
  - `java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks`

Locks & condition variables

```
public class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull  = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x)
    throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length)
               putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
    ...
```

```
    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

- The await() call is equivalent to the wait() call
- The signal() call is equivalent to the notify() call

Atomic variables

Atomic variables lock-free & thread-safe programming on single variables

```
public class Sequencer {
    private long unsafeSequenceNumber = 0;
    private AtomicLong safeSequenceNumber = new AtomicLong(0);
    public long unsafeNext() { return unsafeSequenceNumber++; }
    synchronized public long blockingNext() { return unsafeSequenceNumber++; }
    public long safeLockFreeNext() { return safeSequenceNumber.getAndIncrement(); }
}
```


Reader/Writer support

Interface **ReadWriteLock** & class **ReentrantReadWriteLock** support reader/writer solutions with the following properties:

- multiple threads can read simultaneously
- fairness policy can be enforced (arrival-order)

```
public class ReaderWriterDictionary {
    private final Map<String, String>  m = new TreeMap<String, String>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock(true);
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public String get(String key) {
        r.lock(); try { return m.get(key); } finally { r.unlock(); }
    }
    public String put(String key, String value) {
        w.lock(); try { return m.put(key, value); } finally { w.unlock(); }
    }
}
```

Producer/Consumer support

Interface **BlockingQueue** & various implementation classes support producers/consumers solutions in a direct manner

```
class Producer implements Runnable {
    private final BlockingQueue queue;
    Producer(BlockingQueue q) { queue = q; }
    public void run() {
        try {
            while(true) { queue.put(produce()); }
        } catch (InterruptedException ex) { ... }
    }
    Object produce() { ... }
}
```

```
class Consumer implements Runnable {
    private final BlockingQueue queue;
    Consumer(BlockingQueue q) { queue = q; }
    public void run() {
        try {
            while(true) { consume(queue.take()); }
        } catch (InterruptedException ex) { ... }
    }
    void consume(Object x) { ... }
}
```

```
void main() {
    BlockingQueue q = new ArrayBlockingQueue(1000,true);
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
}
```


Concurrent collections

- The **Hashtable** is already thread-safe, so why define a new class **ConcurrentHashMap**?
- With a **Hashtable**, every method is synchronized, so no two threads can access it concurrently
- With a **ConcurrentHashMap**, multiple operations can overlap each other without waiting, i.e.,
  - unbounded number of reads can overlap each other
  - up to 16 writes can overlap each other
  - reads can overlap writes

Questions?