Implementing concurrency and communication

Kamal Lodaya
The Institute of Mathematical Sciences
Chennai

Lecture 3 $\boxed{\text{Concurrency}}$

Text: G.R.Andrews, *Foundations of multithreaded, parallel, and distributed programming*, Addison-Wesley, 2000

Threads

The simplest way to define concurrent threads in Java is to have a class extend the built-in class `Thread`. Suppose we have such a class, as follows, in which we define a function `run()` as shown:

```
public class Window extends Thread{
  private int id;
  public Window(int i){  // Constructor
    id = i;
  }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.print("Window "+id+":>");
      try{
        sleep(1000);
      }      // Go to sleep for 1000 ms
      catch(InterruptedException e){
        CommandHandler(e);
} } } }
```

Threads

Then, we can created objects of type `Window` and "start" them off in parallel. The call `w[i].start()` initiates the function `w[i].run()` in a separate thread.

```
public class TestWindows {
  public static void main(String[] args){
    Window w[] = new Window[5];

    for (int i = 0; i < 5; i++){
       w[i] = new Window(i);
       w[i].start(); // Start off w[i].run()
                     // in concurrent thread
}   }
```

If we write `w[i].run()` instead of `w[i].start()` `w[i].run()` is like any other function in the current thread—control is transferred to `w[i].run()` and when it finishes, passes back to the caller.

Threads within threads

Given Java's single inheritance mechanism, we cannot always extend `Thread` directly. An alternative is to implement the interface `Runnable`. A class that implements `Runnable` is defined in the same way as one that extends `Thread`—we have to define a function `public void run()...` However, to invoke this in parallel, we have to explicitly create a `Thread` from the `Runnable` object by passing the reference to the object to the `Thread` constructor. Here is how we would rework the earlier example.

First the `Runnable` class:

```
public class Window implements Runnable{
             // only this line has changed
  private int id;
  public Window(int i){ ... } // Constructor
  public void run(){ ... }
}
```

Threads within threads

Now, the class that uses the `Runnable` class.

```
public class TestWindows {
  public static void main(String[] args){
    Window w[] = new Window[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       w[i] = new Window(i);
       t[i] = new Thread(w[i]);
            // Make a thread t[i] from w[i]
       t[i].start(); // Start off w[i].run()
                     // in concurrent thread
}   }
```

Note: `t[i].start()`, not `w[i].start()`

Threads

To summarize: a Java thread is a lightweight "process": it has its own stack and execution context, and it has direct access to all variables in its scope. To program a thread:

- Define a new class that *extends* `Thread` or *implements* `Runnable`.
- Define a `run` method in the new class which contains the code of the thread.
- Create an instance of the new class with `new`.
- Start the thread using the `start` method.

Parallel programming

In most parallel iterative algorithms, each iteration typically depends upon the results of the previous iteration.

```
while (``not completed''){
  Process p[] = new Process[5];

  for (int i = 0; i < 5; i++){
     p[i] = new Process(i);
     p[i].start(); // p[i].run() will do
                   // the i'th job
} }
```

This is quite inefficient since every iteration spawns 5 threads. It is costly to create and destroy threads (even though they are "lightweight").

Parallel programming

A more efficient structure is:

```
 Process p[] = new Process[5];
 for (int i = 0; i < 5; i++){
      p[i] = new Process(i);
      p[i].start(); // p[i].run() will do
                    // the i'th job
 }

public class Process extends Thread{
  private int id;
  public Process(int i){  // Constructor
    id = i;
  }
  public void run(){
    while (``not completed''){
      ``code to do an iteration
        of the i'th job;''
      ``barrier synchronization''
} } }
```

Synchronization

To summarize: rather than creating and destroying processes, it is more efficient to create them once and for all, and to use a synchronization protocol.

- If you are familiar with regular expressions, and we use $||$ to separate parallel processes, the motto is that a program of the form $(a||b)^*$ can be more efficiently transformed into $(a; barrier)^*||(b; barrier)^*$.

Implementing synchronization

But how do we implement the barrier ?

Some programming languages even have such a construct. But then how do the languages themselves implement it ?

Barrier synchronization

A barrier can be a shared integer `count`, and synchronization is

```
        count++;
        while (count != N){ // do nothing
        }
```

But how is `count` reset? How can it be reset before any process again tries to increment it?

Make a barrier

Suppose we distribute `count` into an array `arrive[]`. The increment `count++` is replaced by `arrive[i] = 1`. Then the test will be:

```
await (arrive[1]+arrive[2]+...+arrive[N] == N);
```

This gives rise to a lot of memory contention, and it is inefficient to keep making the test in every process.

Centralized barrier synchronization

```
public class Process extends Thread{
  public Process(int i) ...
  public void run(){
    while (‘‘not completed’’){
      ‘‘code to do an iteration’’
      arrive[i] = 1;
      await (continue[i] == 1);
      continue[i] = 0;
  } }
public class Coordinator extends Thread{
  public Coordinator ...
  public void run(){
    while (true){
      for (int i = 0; i < N; i++){
        await (arrive[i] == 1);
        arrive[i] = 0;
      }
      for (int i = 0; i < N; i++){
        continue[i] = 1;
  } } }
```

Distributed barrier synchronization

Each worker process can also be made to act as a coordinator, for instance by arranging the processes in the form of a ring or a tree. The code for a process looks like:

```
// barrier code for leaf process p[l]
arrive[l] = 1;
await (continue[l] == 1); continue[l] = 0;

// barrier code for interior process p[i]
await (arrive[left] == 1); arrive[left] = 0;
await (arrive[right] == 1); arrive[right] = 0;
arrive[i] = 1;
await (continue[i] == 1); continue[i] = 0;
continue[left] = 1; continue[right] = 1;

// barrier code for root process p[r]
await (arrive[left] == 1); arrive[left] = 0;
await (arrive[right] == 1); arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

Communication

To summarize: it is not *concurrency* that makes parallel programming difficult.

- Lightweight implementations such as threads can be provided relatively easily in a programming language.
- But the processes need to *coordinate* with each other. In the simplest case they need to *synchronize* across threads.
- More generally, processes need to *communicate* information among each other.
- Implementing synchronization and communication is not easy.

Load balancing

How would you go about finding all primes upto 10^{10} using 10 threads ?

Load balancing

How would you go about finding all primes upto 10^{10} using 10 threads ?

Strategy 1: dividing up the input domain

```
int i = ThreadID.get(); // threads 0..9
long int blocksize = power(10,9);
for (long int j = (i * blocksize)+1;
                j <= (i+1)*blocksize; j++){
    if (testPrime(j))
        System.out.print(j);
}
```

Dividing up the input domain is not a good strategy for the primes problem. Why ?

Load balancing

- There are a lot more primes between 1 and 10^9 than between 9×10^9 and 10^{10}. The distribution of the primes is not uniform.
- Checking if a number is prime is easier when the number is small, for large numbers it is harder. The work load is not uniform.

Strategy 2: dynamic load balancing

```
Counter counter = new Counter(1); // shared
long int i = 0;
long int limit = power(10,10);
while (i < limit) {
    i = counter.Incr();
    if (testPrime(i))
        System.out.print(i);
}
```

This solves the problem of skewed distributions. But how do we implement the counter ?

Load balancing

```
public class Counter{
  private long int value;
  public Counter(int i){
      value = i;
  }
  public long int Incr(){
      return value++;
} }
```

On many machines, the last piece of code will be implemented as:

```
long int temp = value;
         value = temp+1;
         return temp;
```

The field `value` is part of the shared `Counter` object. But each thread has its own local copy of `temp`.

What can happen if two threads call `Incr` at about the same time ?

Race conditions

Consider a system in which we have two threads that update a shared variable `n`, as follows:

```
Thread 1            Thread 2


...                 ...
m = n;              k = n;
m++;                k++;
n = m;              n = k;
...                 ...
```

Under normal circumstances, after these two segments have executed, we would expect the value of `n` to have been incremented twice, once by each thread.

Race conditions

However, because of time-slicing, the order of execution of the statements may be as follows;

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;
          // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;
     // Same value as that set by Thread 1
```

In this sequence, the increments performed by the two threads overlap and the final value of `n` is only one more than its initial value. This kind of inconsistent update, whose effect depends on the exact order in which concurrent threads execute, is known as a *race condition*.

Race conditions

A more amusing example has a shared array `double accounts[100]` that holds the current balance for 100 bank accounts and two functions:

```
// transfer "amount" from source to target
boolean transfer
(double amount, int source, int target){
    if (accounts[source] < amount)
        {return false;}
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
}

// compute the total balance of the bank
double audit(){
    double bank = 0.00;
    for (int i = 0; i < 100; i++)
        {bank += accounts[i];}
    return bank;
}
```

Race conditions

Now, suppose we execute these functions in concurrent threads, as follows:

```
Thread 1                     Thread 2

...                          ...
transfer(500.00,7,8);        print (audit());
...                          ...
```

Suppose that Thread 2 gets access to `accounts[7]` and `accounts[8]` after Thread 1 has debited `accounts[7]` but before it has credited `accounts[8]`. The audit will then report that a sum of 500.00 has been “lost” from the accounts in the bank, because it uses the updated value of `accounts[7]` but the old value of `accounts[8]`.

Race conditions

```
Thread 1                  Thread 2
...                       ...
transfer(500.00,7,8);     print (audit());
...                       ...
```

The situation is even more complicated than this. Even if Thread 1 executes both the debit and the credit without interruption, it is possible that Thread 2 reads `accounts[7]` before the transfer and `accounts[8]` *after* the transfer, thereby recording an excess of 500.00 in the total bank balance.

All of these examples can be formulated as instances of the familiar problem of *mutual exclusion* to *critical regions* of program code.

Monitor synchronization

The idea of a “secretary” to manage synchronization was suggested by Edsger Dijkstra. Per Brinch Hansen and Tony Hoare proposed *monitors* to support mutual exclusion. Java uses *synchronized* methods.

A `wait()` suspends the currently executing thread. A `notifyAll()` wakes up suspended threads. The notifying process continues in the monitor until it completes its normal execution. A woken-up thread now gets a chance to run.

```
public class bank_account{
  double accounts[100];
  public synchronized boolean transfer
    (double amount, int source, int target){
        while (accounts[source] < amount)
          {wait();}
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll(); return true;
  }
  public synchronized double audit(){
        double bank = 0.0;
        for (int i = 0; i < 100; i++)
          {bank += accounts[i];}
        return bank;
  }
  // a non-synchronized method
  public double current_balance(int i){
        return accounts[i];
} }
```

Java's `wait-notify` mechanism is called *signal and continue.*

Readers and writers

We program a basic class which allows several reader processes to concurrently read a database or one writer process to exclusively write on it.

```
// concurrent read or exclusive write
class ReadersWriters {
   protected int data = 0; // the "database"
   private int nr = 0;
   private synchronized void startRead() {
         nr++;
   }
   private synchronized void endRead() {
         nr--;
         if (nr == 0)
           notifyAll(); // wake up writers
   }
   public void read(){
         startRead();
         System.out.println("read: "+data);
         endRead();
   }
   public synchronized void write() {
         while (nr > 0)
            // delay if readers are active
            try{wait();}
         catch (InterruptedException ex) {
            return;
         }
         data++;
         System.out.println("wrote: "+data);
         notifyAll();  // wake up writer
}  }
```

Reader and Writer processes can be written by extending the class Thread. For example:

```
class Reader extends Thread {
   int n; ReadersWriters RW; // ref to object
   public Reader(int n, ReadersWriters RW) {
         this.n = n; this.RW = RW;
   }
   public void run() {
      for (int i=0; i < n; i++) {RW.read();}
}  }
```

Synchronized objects

Actually, Java has a more flexible mechanism to provide mutual exclusion: Every object can be synchronized with a separate “internal queue”, so an arbitrary block of code can be synchronized for any object.

```
public class XYZ{
Object o = new Object();
public int f(){
  ...
  synchronized(o){ ...
     o.wait(); // in queue attached to "o"
     ...
} }
public double g(){
  ...
  synchronized(o){ ...
     o.notifyAll(); // wake up "o" queue
     ...
} }
```

A class of locks

Even more flexibly, Java provides the `Lock` class.

```
import java.util.concurrent.locks;
public interface Lock{
  public void lock(); //for entry to CS
  public void unlock(); //on exit from CS
}
```

These methods are to be used as shown below, ensuring that locks are always released in time.

```
mutex.lock();
try{ ... // body of critical section
}finally{mutex.unlock();}
```

In addition, a thread should be *well formed*, that is, each critical section should be associated with a unique `Lock` object. Threads which are not well formed can behave unpredictably.

Using the class

Our earlier shared counter example can be written using any of the synchronization techniques. We illustrate the use of locks.

```
public class Counter{
  private long int value;
  private Lock mutex;
  public Counter(int i){
      value = i;
  }
  public long int Incr(){
      mutex.lock();
      try{long int temp = value;
                   value = temp+1;
      }finally{mutex.unlock();}
      return temp;
} }
```

Java synchronization

To summarize: race conditions can be created when threads run in parallel using shared memory.

- Java provides `synchronized` methods, objects and locks to program exclusive access to shared memory.
- Each synchronized method, object or lock has an internal queue. The predefined `wait` method suspends the currently executing thread (and puts it on the object's queue), similarly the `lock` method will suspend if the lock is already acquired by another thread.
- `notifyAll` or `unlock` releases the lock and wakes up suspended threads.

But how are these locks (and hence synchronized methods and objects) implemented ?

Implementing concurrency and communication

Kamal Lodaya
The Institute of Mathematical Sciences
Chennai

Lecture 4 $\boxed{\text{Locks}}$

Text: M.P.Herlihy and N.Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008

Implementing locks for two threads: Strategy 1

```
public interface Lock{
  public void lock(); public void unlock();
}
class SeqLock implements Lock{
private volatile flag[]=new boolean[2];
public void lock(){
  int i = ThreadID.get(); int j = 1-i;
  flag[i] = true;
  while (flag[j]) {}  // wait
}
public void unlock(){
  flag[ThreadID.get()] = false;
}}
```

$A(flag[A] := true) \rightarrow A(\neg flag[B]) \rightarrow CS_A$
$B(flag[B] := true) \rightarrow B(\neg flag[A]) \rightarrow CS_B$
$A(\neg flag[B]) \rightarrow B(flag[B] := false)$.

So either $CS_A \rightarrow CS_B$ or $CS_B \rightarrow CS_A$. But $\{A(flag[A] := true), B(flag[B] := true)\} \rightarrow \{A(?flag[B]), B(?flag[A])\} \rightarrow$ deadlock !

Implementing locks for two threads: Strategy 2

```
public interface Lock{
  public void lock(); public void unlock();
}
class ConcLock implements Lock{
private volatile int victim;
public void lock(){
  int i = ThreadID.get();
  victim = i; // let the other go first
  while (victim == i) {} // wait
}
public void unlock() {} // do nothing
}
```

$A(victim := A) \to A(victim, B) \to CS_A$
$B(victim := B) \to B(victim, A) \to CS_B$
$A(victim := A) \to B(victim := B) \to A(victim, B)$.

So either $CS_A \to CS_B$ or $CS_B \to CS_A$. But if one thread completes before the other starts, we have deadlock !

Implementing locks for two threads

```
class Peterson implements Lock{
private volatile flag[]=new boolean[2];
private volatile int victim;
public void lock(){
  int i = ThreadID.get(); int j = 1-i;
  flag[i] = true; // I am interested
  victim = i;     // but you go first
  while (flag[j] && victim == i) {} // wait
}
public void unlock(){
  flag[ThreadID.get()] = false;
}}
```

$A(flag[A] := true) \rightarrow A(victim := A) \rightarrow A(?flag[B]) \rightarrow A(?victim) \rightarrow CS_A$ and $B(flag[B] := true) \rightarrow B(victim := B) \rightarrow B(?flag[A]) \rightarrow B(?victim) \rightarrow CS_B$.

Suppose that $B(victim := B) \rightarrow A(victim := A)$. This implies $A(victim := A) \rightarrow A(\neg flag[B])$. So $B(flag[B] := true) \rightarrow A(\neg flag[B])$, impossible!

Implementing locks for two threads

```
public void lock(){
  int i = ThreadID.get(); int j = 1-i;
  flag[i] = true; // I am interested
  victim = i;     // but you go first
  while (flag[j] && victim == i) {} // wait
}
public void unlock(){
  flag[ThreadID.get()] = false;
}
```

The previous two arguments showed that mutual exclusion is maintained. Suppose now that `A` starves, waiting for $\neg flag[B]$ or $victim == B$. What is `B` doing? As soon as it reenters its critical section, it sets $victim := B$. So `B` must also be waiting. But `victim` cannot be both `A` and `B`, a contradiction!

Starvation freedom implies deadlock freedom. So Peterson's algorithm implements mutual exclusion without any deadlock or starvation.

Implementing locks for n threads

Generalizing Peterson's algorithm, we have an n-element integer array `level[]`, with `level[A]` indicating the highest level that thread `A` is trying to enter. Each thread must pass through $n - 1$ levels of "exclusion" to enter its critical section, by "filtering out" at level l a distinct thread `victim[l]`, leaving one winner.

By induction we argue that this algorithm correctly provides mutual exclusion without deadlock or starvation.

But this is very inefficient !

Cannot we do something better ?

Implementing locks for n threads

```
class Bakery implements Lock{
boolean[] flag; Label[] token;
public Bakery (int n) {
  flag = new boolean[n]; token = new Label[n];
  for (int i = 0; i < n; i++){
    flag[i] = false; token[i] = 0;
}}
public void lock(){
  i = ThreadID.get(); flag[i] = true;
  token[i] = max(token[0],...,token[n-1])+1;
               // always increasing
  k = 0; while (k < n){
    if (flag[k] && (token[k],k)<(token[i],i))
        // lexicographic symmetry breaking
        k = 0; //stay in loop, FCFS
    else k++;
} }
public void unlock(){
  flag[ThreadID.get()] = false;
}}
```

Implementing locks using reads/writes

Leslie Lamport's bakery algorithm provides mutual exclusion, has no deadlocks, and satisfies a first-come-first-served property. Hence it is starvation-free as well. There are cleverer versions (one invented by Sibsankar Haldar and Paul Vitányi) which reuse tokens so that they do not grow unboundedly.

But it is not used in practice because it reads and writes as many memory locations (n) as there are concurrent threads, and that number is unbounded.

It was shown by Jim Burns and Nancy Lynch that any deadlock-free mutual exclusion algorithm which does not use any stronger method of synchronization than read and write operations must suffer the same disadvantage.

Welcome to the real world!

We implement, let us say, Peterson's algorithm for two threads and run it on a modern multiprocessor many times: let us say each critical section is entered exactly half a million times. So the value of the shared counter after all this should be exactly one million.

When you run this, you discover that it is slightly off the one million mark!

The error is small, but why is there an error at all?

Multiprocessor architecture

When programming our multiprocessor, we assume that our reads and writes to memory are atomic. Unfortunately modern multiprocessor architectures do not guarantee this. For example, a write operation might put the value in a cache, and only later is the value transferred from a cache to memory.

Parallelizing compilers also sometimes change the order in which operations are done, and this might have a different effect from what you intended when interleaved with another thread.

Test-and-set

The `java.util.concurrent` package provides an `AtomicBoolean` class which stores a Boolean value (similarly `AtomicInteger`). This has a `set(`b`)` method which stores the bit b and a `getAndSet(`b`)` method which replaces the current value with b, and returns the old value. These operations are guaranteeed to be atomic.

```
import java.util.concurrent;
public class TASLock implements Lock{
AtomicBoolean locked = new AtomicBoolean(false)
public void lock(){
  while (locked.getAndSet(true)) {}
}
public void unlock(){
  locked.set(false);
}}
```

Using this implementation you will not get the kind of errors we saw above. But as the number of threads using this class increases, performance drops sharply.

Test-and-test-and-set

```
import java.util.concurrent;
public class TTASLock implements Lock{
AtomicBoolean locked = new AtomicBoolean(false);
public void lock(){
  while (true){
    while (locked.get()) {};
    if (! locked.getAndSet(true))
        return;
} }
public void unlock(){
  locked.set(false);
}}
```

This implementation, which is called test-and-test-and-set, was developed by Clyde Kruskal, Larry Rudolph and Marc Snir in 1988. It is functionally the same as the previous one. It still slows down as the number of threads using the class increases but it performs better than the previous one.

But why is there a slowdown ?

Multiprocessor caches

The slowdown in both implementations is due to cache performance.

- During the processing cycle, the processor asks for the value at a memory location.
- If the required memory address is in cache (a cache hit), it is loaded immediately.
- Otherwise there is a cache miss. The processor broadcasts the required address on the bus.
- If another processor has that address in its cache, it broadcasts its value on the bus and that is picked up.
- Otherwise memory (which is the slowest) responds with the value.

Do we have to know all these details ?

The present state-of-the-art is that all these details matter for performance and expert parallel programmers have to be aware of them.

Spinning threads

```
public class TASLock implements Lock{
AtomicBoolean locked = new AtomicBoolean(false);
public void lock(){
  while (locked.getAndSet(true)) {}
}
public void unlock(){
  locked.set(false);
}}
```

- Every `getAndSet()` call has to be broadcast on the bus. All processors use the bus to communicate with memory, so these calls delay *all* threads, not just those waiting for the lock.
- Worse, the `getAndSet()` forces other processors to discard their own cached copies of the lock.
- So next time they get a cache miss and fetch the new, but unchanged value.

Spinning threads

```
public class TTASLock implements Lock{
AtomicBoolean locked = new AtomicBoolean(false)
public void lock(){
  while (true){
    while (locked.get()) {};
    if (! locked.getAndSet(true))
        return;
} }
public void unlock(){
  locked.set(false);
}}
```

If thread A holds the lock, then the first time thread B reads the lock it takes a cache miss. But now as long as A holds the lock, B reads, hitting the cache.

When the lock is released, all waiting threads take a cache miss. Then one of them succeeds . . .

Local spinning

```
public void lock(){
  while (true){
    while (locked.get()) {};
    if (! locked.getAndSet(true))
        return;
} }
```

As we saw on the previous slide, local spinning makes test-and-test-and-set more efficient than test-and-set.

Can we build on this idea ?

Once the `get` says that the bit is locked, should we immediately spin again ?

Anant Agarwal and Mathews Cherian had a better idea in 1989.

Exponential backoff

```
public class Backoff{
  final int minDelay, maxDelay;
  int limit;
  final Random random;
  public Backoff(int min, int max){
    minDelay = min; maxDelay = max;
    limit = minDelay;
    random = new Random();
  }
  public void backoff()
  throws InterruptedException{
    int delay = random.nextInt(limit);
    limit = Math.min(maxDelay, 2*limit);
    Thread.sleep(delay);
} }
```

In distributed computing, probabilistic algorithms often perform better than deterministic ones. The Ethernet protocol uses exponential backoff. The duration of backoff is random, and is doubled on failure upto a fixed maximum.

The backoff lock

```
public class BackoffLock implements Lock{
private AtomicBoolean locked
    = new AtomicBoolean(false);
private static final int MinD = ...;
private static final int MaxD = ...;
public void lock(){
  Backoff back = new Backoff(MinD,MaxD);
  while (true){
    while (locked.get()){};
    if (! locked.getAndSet(true)){
        return;
    }else{
        back.backoff();
} } }
public void unlock(){
  locked.set(false);
}}
```

Queue locks

There are two main problems with backoff locks:

- Threads delay longer than necessary, so the critical section is underutilized.
- All threads spin on the same shared location. So every successful lock access causes cache-coherence traffic. This is less than the test-and-set lock, but still a matter of inefficiency.

Having threads form a queue avoids both of these problems. Each thread now spins on a different location. There is no need to guess how long to wait, each thread is notified by its predecessor in the queue. Also the queue provides first-come-first-served fairness (as in Lamport's Bakery algorithm) and hence starvation-freedom.

Anderson's 1990 lock

```
public class ALock implements Lock{
ThreadLocal<Integer> myIndex
  = new ThreadLocal<Integer>(){
    protected Integer initialValue(){
        return 0;
  } }
AtomicInteger tail;
boolean[] flag; // shared
int size;
public ALock(int capacity){
  size = capacity;
  tail = new AtomicInteger(0);
  flag = new boolean[capacity];
  flag[0] = true;
}
```

The disadvantage of Tom Anderson's algorithm is that it requires a known bound on the maximum number of concurrent threads, since that is the capacity of the array it allocates per lock.

Anderson's 1990 lock

```
public void lock(){
  int slot = tail.getAndIncrement()%size;
  myIndex.set(slot);
  while (! flag[slot]) {};
}
public void unlock(){
  int slot = myIndex.get();
  flag[slot] = false;
  flag[(slot+1)%size] = true;
}}
```

To acquire the lock, a thread gets a "slot" by incrementing the array's "tail". If `flag[`j`]` is true, the thread with slot j has permission to acquire the lock, which it does by spinning until the flag at its slot becomes true. To release the lock, a thread sets its slot to false and the next slot to true.

The Craig;Landin&Hagersten'93 lock

The CLH lock uses a more usual pointer implementation of a queue, except that now it has to use a predefined `AtomicReference` type. A sophisticated implementation (not shown) recycles queue nodes. On cache-coherent architectures the CLH algorithm has the best performance.

```
public class CLHLock implements Lock{
AtomicReference<Q> tail
  = new AtomicReference<Q>(new Q());
ThreadLocal<Q> myPred; ThreadLocal<Q> myNode;
public CLHLock(){
  tail = new AtomicReference<Q>(new Q());
  myNode = new ThreadLocal<Q>(){
    protected Q initialValue(){
      return new Q();
    } }
  myPred = new ThreadLocal<Q>(){
    protected Q initialValue(){
      return null;
} } }
```

The Craig;Landin&Hagersten'93 lock

```
public void lock(){
  Q qnode = myNode.get();
  qnode.locked = true;
  Q pred = tail.getAndSet(qnode);
  myPred.set(pred);
  while (pred.locked) {}
}
public void unlock(){
  Q qnode = myNode.get();
  qnode.locked = false;
  myNode.set(myPred.get());
}}
```

Each thread's status is in a `Q` object. If its `locked` field is $true$, either the thread has acquired the lock, or is waiting for the lock. If the `locked` field is $false$, the thread has released the lock.

Each thread refers to its predecessor through a thread-local `pred` variable. The `tail` field is public and has the node most recently added to the queue.

The Mellor-Crummey&Scott'91 lock

```
public class MCSLock implements Lock{
AtomicReference<Q> tail;
ThreadLocal<Q> myNode;
public MCSLock(){
  tail = new AtomicReference<Q>(null);
  myNode = new ThreadLocal<Q>(){
    protected Q initialValue(){
        return new Q();
} } }
public void lock(){
Q qnode = myNode.get();
Q pred = tail.getAndSet(qnode); // I want it
if (pred != null){ // point predecessor to me
    qnode.locked = true;
    pred.next = qnode;
    // spin until predecessor gives up lock
    while (qnode.locked) {}
}}
```

To acquire the lock, a thread appends its own node at the tail of the list.

The Mellor-Crummey&Scott'91 lock

```
public void unlock(){
Q qnode = myNode.get();
if (qnode.next == null){ // no one waiting
  if (tail.compareAndSet(qnode,null))
      return;
  // spin until predecessor fills next field
  while (qnode.next == null){}
}
qnode.next.locked = false;
qnode.next = null;
}}
```

While unlocking, there may be a slow thread trying to acquire the lock. If the `compareAndSet` succeeds, no one else is trying to acquire the lock.

The Java Virtual Machine (JVM) uses enhancements of these lock algorithms of John Mellor-Crummey and Michael Scott. They work reasonably efficiently on a variety of architectures.

Implementing concurrency and communication

Kamal Lodaya
The Institute of Mathematical Sciences
Chennai

Lecture 5 $\boxed{\text{Message passing}}$

Text: G.R.Andrews, *Foundations of multithreaded, parallel, and distributed programming*, Addison-Wesley, 2000

Message passing

Asynchronous message passing is like communicating with a letter (or email): *sending* and *receiving* are two separate events. If you want to receive a message, you might have to wait until the message comes. But you can send a message without waiting.

Synchronous message passing is like communicating with a phone (or a webpage). There is no separate "sending" and "receiving", it is one *transaction* like a method call. Even if you only want to send a message, you have to wait until the recipient is ready to receive it.

Programmers like asynchronous message passing, but then someone else has to implement the buffers where messages are kept until they are delivered.

Spammers also like asynchronous message passing. If they had to wait until you were ready to receive their message, they would lose a lot of time.

The dining philosophers (Dijkstra 1968)

Five philosophers share a circular table, which has five forks, placed between each philosopher. Each philosopher spends life alternately thinking and eating. In the centre of the table is a large plate of tangled spaghetti, and a philosopher needs two forks to extract the spaghetti and eat it.

This is a classic resource-sharing problem. In our implementation of this problem, each fork is a thread and each philosopher is also a thread.

```
public class Fork extends Thread {
    private int id;
    public Fork(int i) {id = i;}
    public void PickUp() {} // no action
    public void PutDown() {} // no action
    public void run() {
        while (true) {
            receive PickUp(); // pseudocode
            receive PutDown(); // pseudocode
}   }   }
```

```
public class Phil extends Thread {
    private int id, first, second;
    public Phil(int i) { // Constructor
        id = i;
        first = i; second = i+1;
        if (i == 4) { // asymmetric
            first = 0; second = 4;
        }
    public void run() {
        while (true) { // more pseudocode
            think;
            send f[first].PickUp();
            send f[second].PickUp();
            eat;
            send f[first].PutDown();
            send f[second].PutDown();
}   }   }
```

The main program

The main program does bookkeeping.

- Creates an array `f[]` of fork threads
- Creates an array `p[]` of philosopher threads
- Starts all the threads off

```
public class Dining {
    public static void main (String[] args) {
        public Fork f[] = new Fork[5];
        public Phil p[] = new Phil[5];
        for (int i = 0; i < 5; i++) {
            f[i] = new Fork(i); f[i].start();
            p[i] = new Phil(i); p[i].start();
}   }   }
```

Is the message passing asynchronous or synchronous? How is it to be implemented in Java?

Asynchronous message passing between threads

Asynchronous message passing between threads is provided by the Java class `Selectable`.

```
class Port extends Selectable {
  public synchronized void send(Object v) {...}
  public synchronized Object receive()
         throws InterruptedException {...}
```

Threads can now use these methods.

```
class Producer implements Runnable {
  private Port port;
  Producer(Port p) {port = p;}

  public void run() {
    try{
        int n = 0;
        while (true) {
           port.send (new Integer(n));
           n = n+1;
    }  }catch (InterruptedException e){}
} } }
```

```
class Consumer implements Runnable {
  private Port port;
  Consumer(Port p) {port = p;}

  public void run() {
    try{
         Integer v=null;
         while (true) {
            v = (Integer) port.receive();
            System.out.println(v.toString());
    }  }catch (InterruptedException e){}
} } }
```

Synchronous message passing between threads

Synchronous message passing between threads is also provided by the Java class Selectable.

```
class Channel extends Selectable {
  public synchronized void send(Object v) {...}
          throws InterruptedException {...}
  public synchronized Object receive()
          throws InterruptedException {...}

class Sender implements Runnable {
  private Channel chan;
  Sender(Channel c) {chan = c;}

  public void run() {
    try{
          int n = 0;
          while (true) {
             chan.send (new Integer(n));
             n = n+1;
    }  }catch (InterruptedException e){}
} } }
```

```
class Receiver implements Runnable {
  private Channel chan;
  Receiver(Channel c) {chan = c;}

  public void run() {
    try {
          Integer v=null;
          while (true) {
             v = (Integer) chan.receive();
             System.out.println(v.toString());
    }  }catch (InterruptedException e){}
} } }
```

There is hardly any difference between synchronous and asynchronous message passing! Can it be all that easy?

Message passing between machines

The Java class `Selectable` implements message passing only between two *threads* on the *same* machine. Actually they are implemented using the monitor synchronization we saw earlier.

What we want is to have message passing between two processes on *different* machines which communicate through a network.

Programming internet and web applications is much harder. . . !

For asynchronous message passing, the `java.net` package contains a number of classes that support stream-based communication (for example, using TCP/IP).

A *connection* is a link (with *sockets* at either end) between two hosts that is established before communication occurs. A *stream* is an ordered sequence of messages sent over a connection.

Messages are not lost, so long as the connection itself does not fail.

Sockets

We program a web server maintaining files of clients.

```
import java.io.*; import java.net.*;
public class FileServer {
public static void main(String[] args){
  try{ ServerSocket s = new ServerSocket(9999);
    while (true){ //awaiting connection on port
      Socket c = s.accept(); //client connects
      // create streams to talk to client
      BufferedReader from = new BufferedReader(
      new InputStreamReader(c.getInputStream()))
      PrintWriter to = new PrintWriter(
        c.getOutputStream());
      // get filename from client
      String filename = from.readLine();
      File clientfile = new File(filename);
      if (!clientfile.exists()){ //no file
        to.println("cannot open "+filename);
        to.close(); from.close(); c.close();
        //close streams and socket
      }else{ //send file
        BufferedReader input = new BufferedReade
          new FileReader(clientfile));
        String line;
        //read and print each line
        while ((line=input.readLine())!= null)
          {to.println(line);}
        to.close(); from.close(); c.close();
        //close streams and socket
    } }
  }catch (Exception e){
      System.err.println(e);
}}}
```

Sockets

Here is the program on the client side.

```
import java.io.*; import java.net.*;
public class Client {
public static void main(String[] args){
  try{ String hostdomain = args[0];
    String filename = args[1];
    // open socket on agreed port
    Socket h = new Socket(hostdomain,9999);
    // open i/o streams on socket
    BufferedReader from = new BufferedReader(
      new InputStreamReader(h.getInputStream()))
    PrintWriter to = new PrintWriter(
      h.getOutputStream());
    // send filename
    to.println(filename); to.flush();
    // read and print lines
    String line;
    while ((line=from.readLine())!= null){
        System.out.println(line);
  } }
  // until server closes connection
  // or the client catches an exception
  catch (Exception e){
      System.err.println(e);
}}}
```

Each program is compiled on its host machine. The server is started by executing

```
java FileReader
```

The client is started on its host by executing

```
java Client hostname filename
```

where `filename` is the file the client wants to read. More clients can be started. The server continues running until it is killed or catches an exception.

What happens if the client is started before the server?

What happens if two clients try to connect to the server at about the same time?

Asynchronous message passing summary

To program the server:

- Create a new `ServerSocket` on a port.
- Wait for the `Socket` with which the client connects.
- Create input and output streams for using the socket.
- Use the streams just like any other stream.

To program the client:

- Open a `Socket` connection to a port on a host.
- Create input and output streams for using the socket.
- Use them just like any other stream.

Remote method invocation

The `java.rmi` and `java.rmi.server` packages contain classes which support synchronous message passing. In addition to the server and client, such applications must also have an interface. The interface class must extend `java.rmi.Remote`, and each method in it must throw `RemoteException`.

```
import java.rmi.*; import java.rmi.server.*;
public interface Database extends Remote {
  public int read()
    throws RemoteException;
  public void write(int value)
    throws RemoteException;
}
```

Remote server

The implementing server class must extend `java.rmi.server.UnicastRemoteObject` and implement the interface methods.

Each server object has to be "registered" with a *registry service*, which is a program maintaining a list of registered servers on a host. Before running the server on the host virtual machine using `java`, the registry must be run as a background job on the host (eg, using `rmiregistry 9999 &`).

```
class DatabaseServer
extends UnicastRemoteObject
implements Database {
  protected int data = 0; // the ``database''!
  public DatabaseServer() throws RemoteExceptio
    {super;} // Constructor because of throws
  public int read() throws RemoteException
    {return data;}
  public void write(int value)
  throws RemoteException
    {data = value;}
public static void main(String[] args){
  try{
    DatabaseServer s = new DatabaseServer();
    String name = // register name
      "rmi://srv.imsc.res.in:9999/path";
    Naming.bind(name,s);
    System.out.println(name+" is running");
  }catch (Exception e){
      System.err.println(e);
}}}
```

After compiling the server with `javac`, the special RMI compiler `rmic` has to be run. This enables a *skeleton stub* to be created for each method on the server side and a *proxy stub* for each method called on the client side. The Java virtual machine will run a synchronization protocol between the proxy and the skeleton.

Remote client

The client program needs to set up a *security manager* and use the *registry service* to look up the server. This example client program only does some trial reads and writes.

```
import java.rmi.*; import java.rmi.server.*;
class Client {
public static void main(String[] args){
  try{ // set security, then find database
    System.setSecurityManager(
        new RMISecurityManager());
    String s="rmi://srv.imsc.res.in:9999/path";
    Database db = (Database)Naming.lookup(s);
    int n = Integer.parseInt(args[0]);
    for (int i=0; i < n; i++) {
        int value = db.read();
        db.write(value+1);
  } }catch (Exception e){
      System.err.println(e);
}}}
```

Synchronous message passing summary

First create an interface class which extends `java.rmi.Remote`, with each method throwing `RemoteException`.

The server class must extend `java.rmi.server.UnicastRemoteObject` and implement the interface methods.

It has to be compiled with `javac` and then with `rmic DatabaseServer` to create stubs.

On the server side:

- Run a registry as a background job.
- Now run `java DatabaseServer`.

On the client side:

- Set up an RMI security manager.
- Run the client using `java Client n`.
- Client looks up server and makes connection.
- Now the server's (remote) methods can be called just like any other (local) method.

Synchronous from asynchronous

Synchronous message passing can be implemented using asynchronous messages by using three arrays of channels: `sourceReady`, `destReady` and `transmit`. The first two are "control" channels, the third one is used to communicate the data.

Synchronous send by source process `S`:

```
gather message into buffer b;
send sourceReady[R](S); //R, I am ready
receive destReady[S](); //await R ready
send transmit[R](b);    //send the message
flush buffer b;
```

Synchronous receive by target process `R`:

```
int source; byte buffer[BUFSIZE];
receive sourceReady[R](source); //await sender
send destReady[source]();     //I am ready
receive transmit[R](buffer); //get the message
unpack buffer;
```

Asynchronous from synchronous

Asynchronous message passing can be implemented using synchronous messages by using an intermediate process which maintains buffers. This is what we do when we use a mail server.

Usually we want messages to be received in the order they were sent, so the buffers are implemented as queues.

Should the buffers be on the sender's side or on the receiver's side?