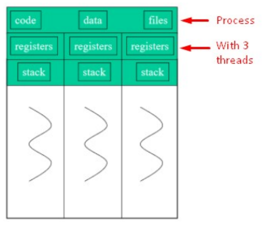
1. Operating Systems

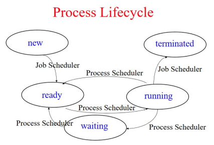
Created: *17/06/2018 6:41 PM*
Updated: *18/06/2018 10:08 AM*

Threads v Processes

There is a lot of overhead in creating a new process. Rather than creating a process, which inherit similar features to its parent process, a thread is more efficient. Processes are ideal when they perform a lot differently from their parent process.

A process contains a set of threads, or just a single thread. Threads run in parallel when there is more than one core. This is called multiprocessor architecture. Each core is considered a separate processor.





Each process is represented by a **Process Control Block**, by the OS. It contains,

- The state of the process
- Current situation (running, waiting...)
- Pointer to its memory location.
- Links to other *PCBs* to form a queue.

Spawning processes
Nested process: has a parent.
Flat: at the same level as the *spawnee.*

When creating a thread,
It allocates memory, initializes a new thread in JVM and invokes the run() method.

Concurrency

Two **types** of concurrency:

- Inherently: real time, activities that can happen simultaneously.
- Potentially: possible to parallelise, massive computations.

Flynn's types of concurrency
Single Instruction, Single Data Stream (SISD)
Single Instruction, Multiple Data Streams (SIMD, ideally GPUs)
Multiple Instructions, Single Data Streams (MISD, rarely used)
Multiple Instructions, Multiple Data Streams (MIMD, most flexible)

Embedded Systems: small dedicated processors, all communicate with main controller, work in real time.

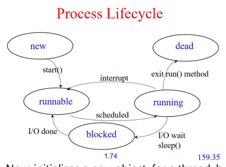
Context-switch: switching between processes, saving states between them and then going back to the waiting process.

An **atomic action** is a single instruction that cannot be interrupted.

A program is deterministic if there is only one path, that is the same output at all times. For concurrent systems, threads can execute at different times thus changing the output.

Concurrency is different from parallelism.
Concurrency refers to performing two tasks at the same time. But, one can be paused while another one is being worked on. Parallelism refers to performing tasks in simultaneously.

The Life-cycle



New: initializes a new object, for a thread, but does not start. It then becomes, runnable.
The thread is not started until start() is invoked.
The thread dies, when it is interrupted or the run method terminates.

Process Scheduling

Refers to the operating system, or JVM deciding what process (thread) to run next. Goal is to give every thread a fair share, minimize the time of running each process, use the processor as efficiently as possible. The **run-time support system** takes this job. JVM doesn't have this, OS manages it. Default = unfair.

- Giving every process an equal time slice, but increasing context switches (fair).
- Or, in sequence (unfair).

Timing

sleep(milli), a thread is then runnable again but doesn't start running immediately. yield() puts the running process back into ready queue.

2. Thread Interaction (v2)

Created: *17/06/2018 10:30 PM*
Updated: *18/06/2018 9:41 AM*

Questions:

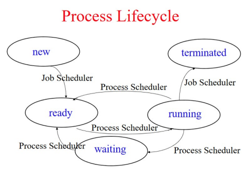
- What is condition synchronization?

Definitions

Throughput: keep completing processes.
Turnaround: Execute processes quickly.

Threads

Provides less cost towards context switching.
Threads share memory & resources of spawning process (shared-memory).
OS handles process scheduling



Job scheduler: keeps a good mix of job types, so everything is busy. Monitors queue of incoming jobs, checks I/O requirements, amount of computation. Selects ONE for *process queue*.

Process scheduler: monitors ready *process queue*. Chooses next job, and decides on time slice. Follows *scheduling policy*.

Process Control Blocks (PCB)

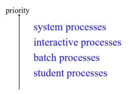
Pointer
Process status
Process number
Process counter
Registers
Memory limits
List of open files
…

```
repeat
    $P_i$ is running
    $P_j$ is selected to run
    State of $P_i$ is saved into $PCB_i$
    State of $P_j$ is loaded from $PCB_j$
    $P_j$ runs
forever
```

Left: PCB, right: context switch algorithm.

Scheduling Policies

1. First in, first out (FIFO or FCFS: first come first serve)...a time consuming job can clog.
2. Shortest Job First... process must give this information.
3. Priority scheduling... assign numbers to processes. Introduces starvation.
4. Round Robin: time slices (context-switching). Good for multiple users.
5. Shortest remaining time (SRT).
6. Multiple level queues (combination) -- see diagram.



Fair means that every eligible instruction will eventually be chosen (e.g random, FIFO).

Algorithm	Policy Type	Best For	Disadvantages	Advantages
FCFS	Non-preemptive	Batch	Unpredictable turnaround times	Easy to implement
SJN	Non-preemptive	Batch	Indefinite postponement of some jobs	Minimises average waiting time
Priority scheduling	Non-preemptive	Batch	Indefinite postponement of some jobs	Ensures fast completion of important jobs
SRT	Preemptive	Batch	Overhead occurred by context switching	Ensures fast completion of short jobs
Round robin	Preemptive	Interactive	Requires selection of good time quantum	Reasonable response times to interactive users; Fair CPU allocation
Multiple-level queues	Preemptive/Non-preemptive	Batch/ Interactive	Overhead occurred by monitoring of queues	Flexible; various techniques to favour CPU- or I/O-bound jobs, to prevent indefinite delay

Thread Interference

Each thread has its own register. This contains a version of something stored in shared memory. However, when multiple threads update the memory location in shared memory at the same time, the versions differ and are overwritten, or corrupted.

Problems

LIVELOCK: busy-wait, ***DEADLOCK***: blocked-waiting (sleeping).

Conditions for Deadlock

- ➢Shared resources must be used under mutual exclusion
- ➢Threads hold resources while waiting for new ones
- ➢The system cannot pre-empt resources
- ➢There is a circular chain of requests and allocations

Detecting & Preventing Deadlock

The OS -- detects & tells the user it has happened, letting them sort it out. Or, avoids -- overcome it; take avoiding action. Prevent by insisting it is logically impossible.

Detection
Graphs: look for cycles.
The OS can respond by aborting all threads, abort threads one at a time, or pre-empt resources one at a time.

Threads chosen for termination can be: random, by priority, length of operation or number & nature of resources required.

Avoidance
The Banker's Algorithm: threads state their maximum resource claim when they first request resources.

Each time, resources are requested, they are only allocated if:

- The thread didn't ask for more than its max.
- If the request was granted, the program could still complete if all the threads asked for their full claim.

- ❖Avoid mutual exclusion - spooling (virtual printer)
- ❖Avoid busy-waiting - request all resources at once
- ❖Introduce pre-emption of resources (can lose work)
- ❖Avoid circular waiting - request resources only in a pre-defined order

2. Concurrent Programming Correctness (NOTES)

Created: *15/06/2018 5:44 PM*
Updated: *17/06/2018 10:27 PM*

Concurrent Programming Correctness

Safety: Synchronization & mutual exclusion should be enforced, to avoid *race conditions*.

Liveness: Terminating before finishing. Caused either by livelock, deadlock or starvation. For example, if a thread dies, the other threads shouldn't too, or lock.

Race condition: when thread A races, and always wins over thread B. Removing the race condition := condition synchronization.
Deadlock: When all threads are locked waiting for some condition.
Livelock: threads are still running, but no useful work is being done by the threads.

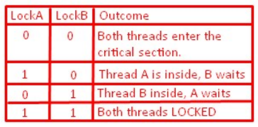
Examples introduced to...

1.
```
boolean lockA, lockB = false;

while (lockB) {}
lockA = true;
// critical section code
lockA = false;
```

Mutual exclusion isn't enforced above. Lock B corresponds to Thread B, and therefore Thread A will be blocked until the lock on Thread B has been released. However, Thread B can still enter the critical section while Thread A unblocks. Therefore, thread A should tell thread B it is waiting to access the critical section, then thread B will lock as soon as it exits the critical section. Below, mutual exclusion is enforced.

2.
```
lockA = true;
while (lockB) {}
// critical section code
lock A = false;
```

However, what if: lockA = true && lockB = true, then both threads busy wait. *Livelock*.



LockA	LockB	Outcome
0	0	Both threads enter the critical section.
1	0	Thread A is inside, B waits
0	1	Thread B inside, A waits
1	1	Both threads LOCKED

1. (0,0), (1,0) and (0,1) are possible. (1,1) is possible too, but disregarded since has no effect on locking.
2. (1,0), (0,1), (1,1) are possible. (0,0) is impossible.

Peterson's Algorithm overcomes (1,1) in **2.** assuring that both locks aren't set to true. **Dekker** came up with a similar algorithm although is less efficient.

```
boolean lockA, lockB = false;
int turn = 0;

lockA = true;
turn = 1;  // remember, Thread A has 0, Thread B has 1
while (lockB && turn != 0) { }
// critical section code
lockA = false;
```

Turn identifies that it is currently thread A's turn, and if both locks are enabled, turn blocks the other thread entering. This removes (1,1).

Bakery Algorithm (correctness for n threads)

Questions

1. The difference between livelock & deadlock.

3. Synchronisation

Created: *16/06/2018 6:02 PM*
Updated: *17/06/2018 10:28 PM*

Ultimate goal: to reduce time in critical sections. By all means, only access it if absolutely required, and only perform what is absolutely required.

Implicit Locks

Synchronisation is a mutex (mutually exclusive) lock. **This lock is per Java object.**
The JVM manages synchronization, therefore there is more information in thread dumps: making debugging easier.

Method

public synchronized void function() { ... } which contains some critical section. A thread must grasp onto the object's lock. When synchronising a static method, the class object is used instead. Otherwise the object which called the method, is used.

public synchronized void run(), means threads will have to wait until the thread terminates. The length of time that a lock is held, is called its **scope**.

Block

Mainly used to reduce the scope. Since sometimes, an entire synchronized method isn't required for a single thread: since it includes steps that are irrelevant. ***this***, is the same object as in method level.

```
public void function() {
  // Some code
  synchronized(this) {
        // Code that requires the lock
  }
  // More code
}
```

Explicit Locks

Includes ReentrantLock, read & write locks.

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class DoSomething {
  private Lock myLock = new ReentrantLock();

  public void concurrentCode() {
        try {
              myLock.lock();
              // Code
        } finally {
              myLock.unlock();
  }
}
```

The **try...finally** construct assures that if the method dies, the lock is always unlocked.
tryLock() will try obtain the lock, possible to then do other things if the lock is not obtainable.
ReentrantLock is similar to a semaphore. Locks are not unlocked, until the counter is zero, allowing nested locks?

Read/Write

```
void accessData() {
    rwl.readLock().lock();
    // What if you want to change it?
    if (wantChange) {
      rwl.readLock().unlock();   // must unlock first to obtain writelock
            rwl.writeLock().lock();
            // code to change data
            // downgrade lock
            rwl.readLock().lock();  // reacquire read without giving up write lock
            rwl.writeLock().unlock(); // unlock write, but retain read lock
```

---

4. Semaphores & Monitors

Created: *16/06/2018 7:10 PM*
Updated: *17/06/2018 10:28 PM*

Semaphores

Access control (e.g limited resources).

S.acquire() requests a 'ticket', and if it exceeds the maximum number, it waits till it is able to get a ticket.
S.release() frees another ticket, and makes it available. It also checks if any are waiting for a ticket, and gives it to one of them.

A *binary semaphore* (max ticket size = 1), is equal to a mutex lock. A *counting semaphore* := max ticket size > 1.
A thread can take many tickets.

Getting order of release() wrong can lead to deadlock, acquire() : not so much.

Atomic Classes
Semaphores make use of atomic integers. Atomic classes are used when only simple interactions are needed on a variable (as the critical section). This reduces the size of critical sections. However, in cases there can be overhead when continuously requesting locks. COMPARE & SWAP is faster than synchronized blocks or locks.

```
public class PseudoRandomUsingAtomic implements PseudoRandom {
    private final AtomicInteger seed;

    public PseudoRandomUsingAtomic(int s) {
        seed = new AtomicInteger(s);
    }

    public int nextInt(int n) {
        for (;;) {
            int s = seed.get();
            int nexts = Util.calculateNext(s);
            if (seed.compareAndSet(s, nexts))
                return s % n;
        }
    }
}
```

Need to loop, in the event a thread is awakened.

Monitors

Condition variables (event variables[Windows]) := access control.
Monitors are mutex. A thread wait()s, until a condition exists and then it is notify[ied]().
Built into the Object class.

The current thread may not own the monitor, hence race conditions possible and there it must be encapsulated by a synchronized block.

If sleep() is used instead of wait(), other threads cannot access the lock since sleep() doesn't know anything about locks, so it cannot free the lock.

Conditions should be enclosed by an infinite loop, in the event the thread is accidentally woken up, or the condition changes very quickly.

More efficient in circumstances that threads are waiting for extended periods, keeping threads off the CPU.

Map-Reduce

Created: *22/05/2018 6:09 PM*
Updated: *18/06/2018 12:00 PM*

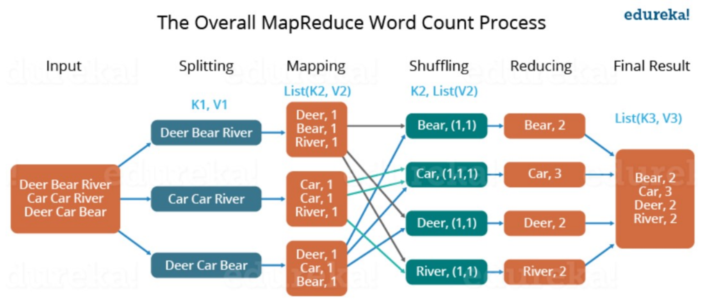
Hadoop is Apache's implementation. Map-Reduce is an abstraction layer on top of MPI.

Map

- Extract something you care about from each record.
- Shuffle and sort.

Reduce

- Aggregate, summarize, filter or transform.
- Write the results.



For example, this can be used to calculate the frequency of words in some array (shown above).

MapReduce can be scaled to clusters of servers to speed up the computation, with easy configuration.

Shuffle

Input to the `Reducer` is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

Sort

The framework groups `Reducer` inputs by keys (since different mappers may have output the same key) in this stage.

The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

Secondary Sort

If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction, then one may specify a `Comparator` via Job.setSortComparatorClass(Class). Since Job.setGroupingComparatorClass(Class) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.

Reduce

In this phase the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each `<key, (list of values)>` pair in the grouped inputs.

The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable).

Applications can use the `Counter` to report its statistics.

As mapped data is inbound over the network, they are *inserted*.

What is the purpose of mapping?

- To transform input records (key, pair) into intermediate records (key, pair).
- Similar to putting them into a HashMap.
- Often 10-100 maps per-node.

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line); Seps a line, by spaces.
       while (tokenizer.hasMoreTokens()) {
           word.set(tokenizer.nextToken()); One word = sep. by spaces.
           context.write(word, one);  write(Key, Value)
       }
   }
}
```

What is the purpose of reducing?

- Includes: Shuffling, sorting, reducing
- What happens if the keys are the same, what should the system do? Keys are grouped automatically, before they reach the reduce method. Car, (1,1,1)

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values, Context context)
     throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
           sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
```

Initializing a job

```
public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();

       Job job = new Job(conf, "wordcount");

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
}
```

Building

```
cd hadoop-3.1.0
mkdir -p build
javac -cp $( bin/hadoop classpath ) -d build -Xlint WordCount.java
```

- Create a JAR file for the WordCount application

```
jar -cvf wordcount.jar -C build .
```

Running

```
bin/hadoop jar wordcount.jar WordCount input output
```

OpenCL

Created: *31/05/2018 10:43 AM*
Updated: *18/06/2018 12:39 PM*

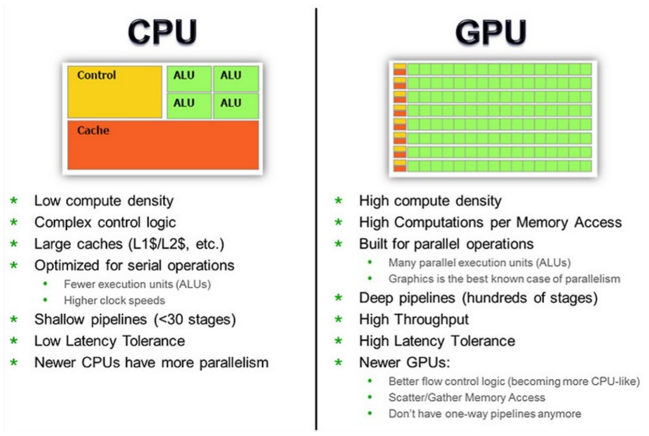
Open Computing Language
Introduction

- It is a framework that allows you to write programs that execute across heterogeneous platforms.
- It provides a standard interface for parallel computing using task-based & data-based parallelism.
- OpenCL can communicate with a large range of devices.

Let the Host be the desktop system.
Let the Compute Device be: CPU or GPU

Generally good for LOTS of data.
Use cases -- when to use GPU for computation

- When devices move memory faster than host.
- Changing from one data format to another.
- Devices calculator faster than the Host, big chunks of data. That is, more computation, using the ALU.



CPU

- Low compute density
- Complex control logic
- Large caches (L1$/L2$, etc.)
- Optimized for serial operations
  - Fewer execution units (ALUs)
  - Higher clock speeds
- Shallow pipelines (<30 stages)
- Low Latency Tolerance
- Newer CPUs have more parallelism

GPU

- High compute density
- High Computations per Memory Access
- Built for parallel operations
  - Many parallel execution units (ALUs)
  - Graphics is the best known case of parallelism
- Deep pipelines (hundreds of stages)
- High Throughput
- High Latency Tolerance
- Newer GPUs:
  - Better flow control logic (becoming more CPU-like)
  - Scatter/Gather Memory Access
  - Don't have one-way pipelines anymore

Double precision maths is slower on a GPU, by *2x*. It's because it requires more data, it doubles the amount of required data.

WHEN TO USE GPU
GPUs have less computation power per thread. While it has more threads it is able to provide better parallelism. It has on-board memory making it ideal f or large data sets. GPUs are often quick, since they do render screens in real time, just like how one would speak through the microphone, and someone over the network would almost receive that in real time. CPUs are more ideal for lower amount of threads, or for tasks which are computationally heavy.

Thread Interaction (Bakery, Mutual Exclusion)

Created: *29/04/2018 1:42 PM*
Updated: *29/04/2018 2:01 PM*

Week 3: Thread Interaction
Two properties in concurrent programs that need to be correct:

1. Safety: means that synchronization & mutual exclusion are enforced.
2. Liveliness: should not die before it is meant to. Live-lock, deadlock and starvation shouldn't be enforced.

Mutual Exclusion: One thread should be in a critical section at a time.
Four methods introduced in lectures.
Spinlocks

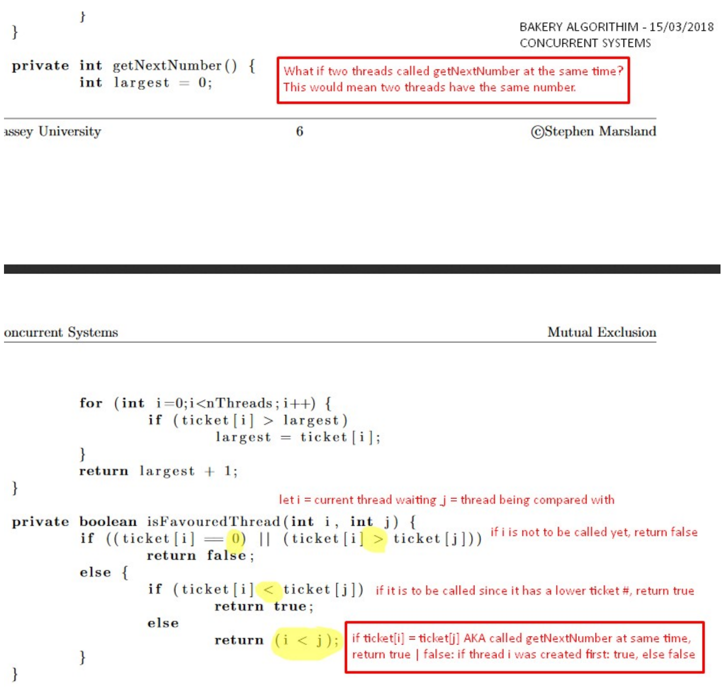
```
global boolean lock
thread A {
   while(lock)
   //--/ Critical Section
}
thread B {
   lock = true;
   //--/ Critical section
   lock = false;
}
```

- A thread may still be executing the critical section before it is locked.
- Busy-waits: the thread stays on the professor in an infinite loop.

In addition to mutual exclusion, assure:

- Each thread must eventually be able to enter the critical section.
- Any thread may halt in any noncritical section.

Bakery Algorithm



BAKERY ALGORITHIM - 15/03/2018
CONCURRENT SYSTEMS

```
        }
}

private int getNextNumber() {
        int largest = 0;
```

What if two threads called getNextNumber at the same time? This would mean two threads have the same number.

assey University 6 ©Stephen Marsland

oncurrent Systems Mutual Exclusion

```
        for (int i=0;i<nThreads;i++) {
                if (ticket[i] > largest)
                        largest = ticket[i];
        }
        return largest + 1;
}

private boolean isFavouredThread(int i, int j) {
        if ((ticket[i] == 0) || (ticket[i] > ticket[j]))
                return false;
        else {
                if (ticket[i] < ticket[j])
                        return true;
                else
                        return (i < j);
        }
}
```

let i = current thread waiting, j = thread being compared with

if i is not to be called yet, return false

if it is to be called since it has a lower ticket #, return true

if ticket[i] = ticket[j] AKA called getNextNumber at same time, return true | false: if thread i was created first: true, else false

Additional keywords:
Race condition: When a thread always wins the race to the critical section.
Starvation: When a thread waits for access to a critical section but never gets it.

Deadlock: All threads are blocked waiting for something that'll never happen.
Live-lock: When both threads are running, but are stuck in loops waiting for x condition.

Reentrant Locks

Created: *29/04/2018 2:49 PM*
Updated: *29/04/2018 2:57 PM*

Reentrant Locks

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    //../ Critical section
}
finally {
    lock.unlock();
}
```

For **fair (FIFO: first in first out)**, that is: give threads access in the order that they request it,

```
new ReetrantLock(true);
```

Fairness rarely required, since JVM guarantees that threads will not be starved.

Reentrant Read/Write Locks
Allows multiple simultaneous readers. However, writing must be mutually exclusive.

ReentrantReadWriteLock has two locks:
readLock: used to avoid anyone writing.
writeLock: prevents any other access.

Synchronization

Created: *29/04/2018 2:07 PM*
Updated: *29/04/2018 3:10 PM*

Synchronization
Is a mutex (mutually exclusive) lock. It permits only one thread in its critical section. Focuses on the *object*, rather than the thread itself.
Synchronization solves the following issues:

- **Thread interference:** when there are two operations running in different threads but acting on the same data, interleave. For example, two threads write to the variable *a* at the same time: they both have their own version of *a*, and only one wins. Likewise with reading, may read at same time of read: but what'll the variable contain at that time?

- **Memory consistency:** No guarantee of variables between threads, of the actual value.

Happens-Before Relationship: when a thread exits a synchronized block, it updates the modified variables so all threads have the same version.

Two methods of implementing synchronization:
Method level:

```
synchronized void function() {
    //../ critical section
}
```

Block level:

```
synchronized(this) { //this: the object that should be locked
    //../ critical section
}
```

A thread, when calling a synchronized block:

- if lock is free, go
- otherwise go to sleep & awaken when lock is free

Implicit (synchronized) -

- cannot interrupt threads waiting
- can't check if free before requesting access
- can't stop waiting
- they're not fair

Refer to Lab Three

Explicit Lock VS Implicit Lock

Created: *29/04/2018 3:00 PM*
Updated: *29/04/2018 3:07 PM*

Which lock is ideal?

- Use synchronous if limitations don't matter since it's easy to use.
- If need more functionality, use reentrant locks such as having simultaneous readers, or checking if a lock is in lock state before accessing it.

Ultimate goal: Avoid critical sections wherever possible otherwise, keep size minimal.

Volatile

- These variables are placed into main memory. This makes writing to the variable *atomic*.
- However, there may still be memory consistency problems where threads try to access the same variable at the same time when writing: only one will win access to the atomic operation.
- Implement compare & swap (see Atomic Classes for more) to fix this.

Semaphores

Created: *17/04/2018 7:26 PM*
Updated: *20/04/2018 6:45 PM*

Semaphores

- Semaphores were founded by Edsger W. Dijkstra.
- Two methods: acquire() and release()
- A semaphore is basically an atomic integer.
- Acquire() checks if the integer is greater than zero. If it is, it's decreased by one and the thread is allowed in the critical section.
- Release() checks if there are any suspended process waiting, and if there is it wakes only one up. Otherwise, increases the semaphore.
- Generic (or counting) semaphores allow multiple threads into the critical section. Often good for controlling access to a limited amount of resources: such as items in a buffer being added/removed.

Refer to BarberShop for implementation (src folder).

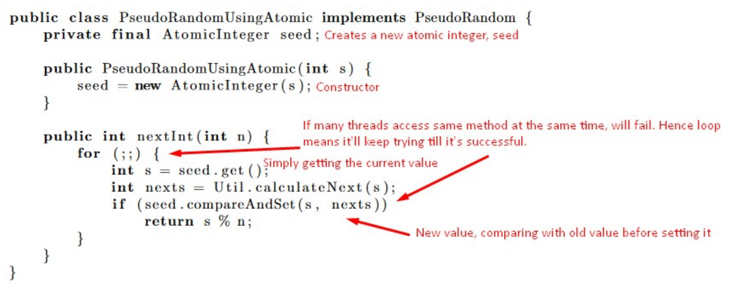
Atomic Classes

Created: *17/04/2018 7:34 PM*
Updated: *20/04/2018 6:45 PM*

Atomic Classes

Compare & Swap
Is a machine instruction which means it's atomic (cannot be interrupted). It's used to check if a variable's old value is still the same before assigning a new value: checking if another thread is accessing it.

java.util.concurrent.atomic
Provides the atomic classes. They have methods: get() & set() which is equivalent to volatile variables.



```
public class PseudoRandomUsingAtomic implements PseudoRandom {
    private final AtomicInteger seed; Creates a new atomic integer, seed

    public PseudoRandomUsingAtomic(int s) {
        seed = new AtomicInteger(s); Constructor
    }

    public int nextInt(int n) {
        for (;;) {
            int s = seed.get();
            int nexts = Util.calculateNext(s);
            if (seed.compareAndSet(s, nexts))
                return s % n;
        }
    }
}
```

If many threads access same method at the same time, will fail. Hence loop means it'll keep trying till it's successful.
Simply getting the current value
New value, comparing with old value before setting it

Monitors

Created: *19/04/2018 6:48 PM*
Updated: *2/05/2018 12:24 PM*

Monitors

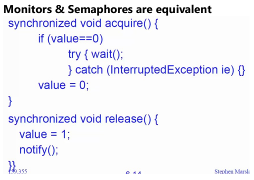
- Monitors use a 'wait and notify' method to communicate between other threads.
- That is, wait() and notify() methods corresponding to sleep and wake up.
- wait() is a must, sleep() does not enforce locks. Wait() is implemented with locking features.
- A monitor '*is essentially a **shared class** with **explicit queues***'.
- **A shared class** refers to a toolkit accessible by multiple threads, while all methods are synchronized but attributes remain encapsulated.
- Using a monitor to protect a class (a data structure?) and assure it is mutex. *Java has implemented this in the root Object class, so all objects have it.*

Methods in Java.lang.Object
notify() - wakes up a single chosen thread waiting on this object's monitor.
notifyAll() - wakes up all threads waiting on this object's monitors.
Threads awakened in the above method will not be able to proceed until the current thread holding the object's monitor relinquishes the lock on the object.
wait() - waits until it has been notified.

wait() should be guarded, in the event a thread is awakened when it shouldn't be. For example,
```
synchronized(obj) {
   while(<condition does not hold>)
      obj.wait();
}
```
This makes the thread currently accessing the object, wait until some condition is met.

Refer to ProducerProblem in monitors package.

Monitors & Semaphores are equivalent



```
synchronized void acquire() {
        if (value==0)
                try { wait();
                } catch (InterruptedException ie) {}
        value = 0;
}
synchronized void release() {
   value = 1;
   notify();
}}
159.355                    6.14                    Stephen Marsl
```

Thread Pools

Created: *2/05/2018 12:31 PM*
Updated: *3/05/2018 1:10 PM*

Thread Pools

1. Threads are created.
2. Threads wait until they are given tasks.
3. Once finished, they return to the thread pool.

- Pools can be created using *ThreadPoolExecutor*
- *LinkedBlockingQueue* is used to store any tasks that are waiting to run.
- The *ThreadPoolExecutor.execute()* method is used to call a run() method of a thread implementation.
- Thread pools are terminated using *ThreadPoolExecutor.shutdown()*. All tasks will be completed, but no new tasks will be accepted. *shutdownNow*() will attempt to stop all current tasks as well. Returns list of tasks waiting for execution.
- The pool shrinks to its specified minimum if threads go unused.

Different types of blocking queues
There are different types of blocking queues (which store the awaiting tasks).

- *ArrayBlockingQueue* (FIFO, bounded queue)
- *LinkedBlockingQueue*: linked list (FIFO, unbounded queue)
- *PriorityBlockingQueue*: orders tasks by priority
- *SynchronousQueue*: doesn't maintain a list, but forwards directly to a thread.

Thread Factory
Creates threads on demand.

The Rejected Execution Handler
Decides how to deal with tasks that are rejected by the execute() method.

- *AbortPolicy*: throws a *RejectedExecutionException*
- *CallerRunsPolicy*: executes the new task outside the thread pool.
- *DiscordPolicy* discards the task.
- *DiscardOldestPolicy*: discards the oldest task in the queue.

Executors
There are classes which makes the job easier. *newCachedThreadPool*() - reuses cached threads, *newFixedThreadPool()* - reuses a fixed number of threads. Also, *newScheduledThreadPool*().

```
import java.util.concurrent.*;

public class ThreadPoolTest {

        public static void main(String[] args) {
                int nTasks = Integer.parseInt(args[0]);
                long n = Long.parseLong(args[1]);
                int tpSize = Integer.parseInt(args[2]);

                ExecutorService executor = Executors.newFixedThreadPool(tpSize);
                Task[] tasks = new Task[nTasks];
                for (int i=0; i<nTasks; i++) {
                        tasks[i] = new Task(n, "Task_" + i);
                        executor.execute(tasks[i]);
                }
                executor.shutdown();
        }
}
```

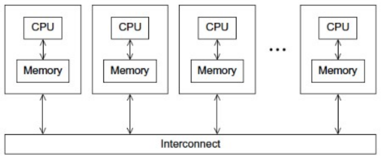
Message Passing Interface (MPI)

Created: *8/05/2018 1:17 PM*
Updated: *14/05/2018 5:50 PM*

Message Passing

Shared memory space is fine, until you have processes or threads across the network. This requires some sort of communication protocol.

Such system is called **Distributed Memory**. Since each thread or process has its own memory, there is no worry regarding race conditions.



Two methods for communication

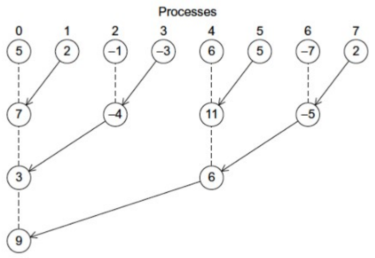
- *Synchronous*: threads sending & receiving messages are blocked until the message is sent.
- *Asynchronous:* threads will not block.
- *Rendezvous*: implements Request-Reply. Asynchronous receiving and synchronous reply. So a thread waits until the other thread replies.

MPI in Java using MPJ

MPJ offers the following methods:

1. *Send*: blocks while sending the message to the process.
2. *Recv*: blocks until it receives.
3. *Isend*: non-blocking send.
4. *Irecv*: non-blocking receiving.
5. *Bcast*: sends a global message.
6. MPI.Wtime() returns the current time stamp.

Collective Communication



Processes can combine messages to create a structure. **The load is furthermore balanced** across processes speeding up the process since the system is using parallelism.

There are functions such as *MPI.SUM* used to combine data from many processes. *MPI.Allreduce* transmits messages to all processes. *MPI.reduce* passes the content onto a single process.

Safety
Deadlock is possible. If unsure if process can deadlock, use *synchronized*. *MPI.Ssend* enforces *synchronized*.

Message Passing Java (MPJ) ...more

Created: *22/05/2018 2:59 PM*
Updated: *22/05/2018 6:02 PM*

Some more on Message Passing
The rank is the current process' ID. Each process has its own ID which uniquely identifies it. The size is the total number of processes specified when running with ***-np {NUMBER OF PROCESSES}*** as a parameter.

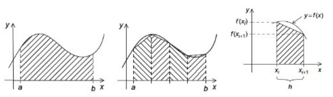
```
import mpi.MPI;

public class HelloWorld {
                public static void main(String[] args) throws Exception {

                        MPI.Init(args) ;

                        int rank = MPI.COMM_WORLD.Rank();
                        int size = MPI.COMM_WORLD.Size();
                        System.out.println("I am process <"+rank+"> of total
                                        <"+ size+"> processes.");
                        MPI.Finalize();
                }
}
```

The MPI.COMM_WORLD class is the communicator which handles the parallelism behind the communication between the processes. This class contains the methods *Send, Recv, Isend, Irecv, Bcast, Reduce, Gather, Scatter, and Barrier. I at beginning is non-blocking.*

Example usage in terms of scientific programming
The trapezoidal rule for example approximates the area underneath a curve. The number, n, is how many times the area underneath the curve is divided. This is also the number of processes for example, that will calculate their own individual areas.

$$\text{Area of 1 trapezoid} = \frac{h}{2}(f(x_i) + f(x_{i+1}))$$

$$h = \frac{b - a}{n}$$



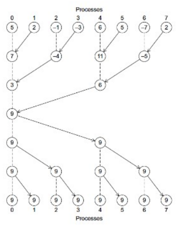
1. partition the solution into tasks.
    1. find the area of an individual trapezoid.
    2. sum the areas.
2. identify the communication channels between the tasks: pass the final areas to the process which sums the lot.
3. aggregate the tasks into composite tasks.
    1. how many processes are needed? n
    2. split the range (b - a) by n.

Each process will run the same implemented function. Therefore, the rank can be used to determine the value of x for performing the calculation. While rank == 0, is the root process summing, the other process send to the root process. Note MPI.DOUBLE is simply the data-type declaration.

```
if (rank != 0) {
MPI.COMM_WORLD.Send(local_int,0,1,MPI.DOUBLE,0,99);
} else {
total_int = local_int[0];
for (source = 1; source < size; source++) {
MPI.COMM_WORLD.Recv(local_int,0,1,MPI.DOUBLE,source,99
total_int += local_int[0];
}
}
```

WARNING (Deadlock)
Deadlock can occur with the BLOCKING version of COMM_WORLD.Send, so use ISend for non-blocking. There are also **synchronized** versions: MPI.COMM_WORLD.Ssend

Efficiency (Scatter, Gather, Collective Communication, Reduce, Allreduce)
In these example, process rank = 0 deals with all the messages. However, this isn't exactly efficient. This is when *Collective Communication* plays a role. It's a tree-based structure, where messages are passed along as they are finished. The load is somewhat more balanced between processes. This implies that there is more work done in parallel, which makes the program more efficient.

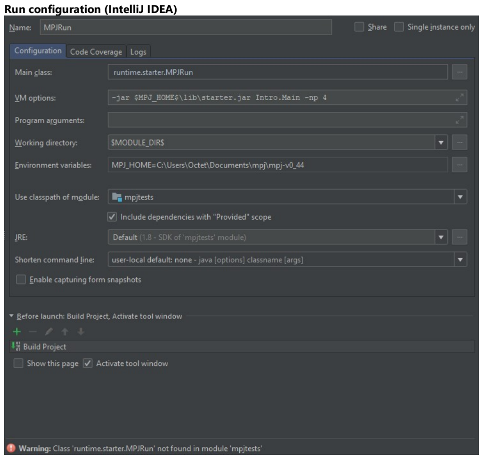


There is a method in MPI.COMM_WORLD, Reduce (the following is for C).

```
//Function prototype
MPI_Reduce(
    void* send_data,     //An array of data to send
    void* recv_data,     //Needed for Root process, to recv data. size: sizeof(datatype) * count
    int count,           //Total # of processes
    MPI_Datatype datatype,    //The data type of the data (i.e MPI.DOUBLE)
    MPI_Op op,            //The operation to perform, i.e MPI.SUM
    int root,            //The Rank of the root process
    MPI_Comm communicator
)
```

Refer to ./MPI/mpj_reduce.pdf
MPI.COMM_WORLD.Allreduce transmits to all processes, rather than just root.
MPI.COMM_WORLD.Scatter and MPI.COMM_WORLD.Gather scatter and gather data arrays from multiple processes. Scatter to give, gather to collect the results.

Run configuration (IntelliJ IDEA)



Name: MPJRun
Share: ☐ Single instance only: ☐

Configuration | Code Coverage | Logs

Main class: runtime.starter.MPJRun
VM options: `-jar $MPJ_HOME$\lib\starter.jar Intro.Main -np 4`
Program arguments:
Working directory: $MODULE_DIR$
Environment variables: MPJ_HOME=C:\Users\Octet\Documents\mpj\mpj-v0_44
Use classpath of module: mpjtests
☑ Include dependencies with "Provided" scope
JRE: Default (1.8 - SDK of 'mpjtests' module)
Shorten command line: user-local default: none - java [options] classname [args]
☐ Enable capturing form snapshots

Before launch: Build Project, Activate tool window
Build Project
☐ Show this page ☑ Activate tool window

Warning: Class 'runtime.starter.MPJRun' not found in module 'mpjtests'

MPI v MapReduce v OpenCL

Created: *30/05/2018 1:18 PM*
Updated: *30/05/2018 1:18 PM*

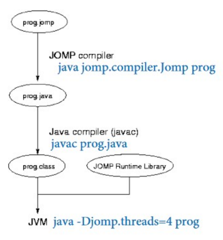
OpenMP

Created: *31/05/2018 11:12 AM*
Updated: *18/06/2018 12:28 PM*

Open Multi Processing

A master thread can spawn a team of threads. These teams form clutters called parallel regions. In Java, there is a port for OpenMP called Jomp (since OpenMP is designed for C/C++ & Fortran).

Rather than using #pragma, like in C to tell the compiler to do something... Java & Fortran uses the keyword, *omp*. Jomp has it's own compiler, which then after being compiled, compiles down to Java.



```
import jomp.runtime.*;

public class HelloWorld {

      public static void main(String [] args ) {

      int myid;

            //omp parallel private(myid) {
                  myid = OMP.getThreadNum();
                  System.out.println("Hello from " + myid);

            }
      }
}
```

//omp parallel, defines a portion of code that should be executed in parallel.
While, ***//omp barrier***, each thread waits at the barrier until all the threads arrive.
For synchronization & critical sections, ***//omp critical***.

```
//omp parallel {
            int id, i, nThrds, start, end;
            id = OMP.getThreadNum();
            nThrds = OMP.getNumThreads();
            start = id*N / nThrds;
            end = (id+1)*N / nThrds;
            if(id == nThrds-1)
                  end = N;
            for(i = start; i<end; i++){
                  a[i] = a[i] + b[i];
            }

            //omp critical{
                  sum += a[i];
            }
}
```

Parallel Loops
OpenMP allows for *for loops* to be executed in parallel. Only works for for loops, and only if the number of iterations is known. At the end of parallel constructs, there is an implicit barrier.

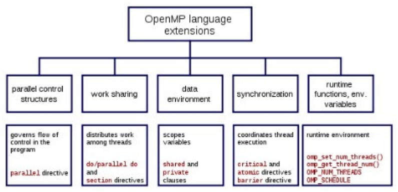
```
//omp parallel
// omp for
for(i=0;i<N;i++) {
    a[i] = a[i] + b[i];
}
```

WARNING: eliminate any dependency. All threads need access to a[0], so set it prior to the for loop.
A race condition exists in the following example, since the iterations are performed in a range of threads, local variables differ between threads inside the loop.

```
long sum = 0;
      //omp parallel for
            for (int i=1; i<1000000; i++)
                  for (int j=1; j<100; j++)
                        sum += i % j;
```

Therefore, a reduction variable must be used: basically declares a global variable that is shared across threads. Results from each thread are combined. A reduction variable can be declared using ***//omp parallel reduction(+: varname)*** where + is the operation being performed on the variable.

```
//omp parallel for reduction(+:sum)
            for (int i=1; i<1000000; i++)
                  for (int j=1; j<100; j++)
                        sum += i % j;
```


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Compiling & Runtime
To compile, must use jomp.compiler.Jomp someFile (with .jomp extension). This will produce a Java file, which can then be compiled. Javac someFile.java. Then running, the number of threads must be specified, java -Djomp.threads=n someFile. jomp1.Ob.jar must be in the CLASSPATH.

```
it042659:Jomp srmarsla$ export CLASSPATH=$CLASSPATH:.:./jomp1.0b.jar
it042659:Jomp srmarsla$ java jomp.compiler.Jomp HelloWorld
Jomp Version 1.0.beta.
Compiling class HelloWorld....
Parallel Directive Encountered
it042659:Jomp srmarsla$ javac HelloWorld.java
it042659:Jomp srmarsla$ java -Djomp.threads=2 HelloWorld
Hello from 0
```

Notes
In the notes, it uses an array, result[] to fetch the results. The reason it cannot return a result, is because it's being performed in parallel. Likewise, if it was a simple integer, it would not correspond to the same memory address. Therefore, an array is used. ***//omp sections { ... }*** followed by ***//omp section*** for specifying what should go on in the first & second threads, is useful.

```
//omp sections {
        //omp section {
                // code for first thread here
        }
        //omp section {
                \\ code for second thread here
        }
        //etc.
}
```

//omp for schedule(static, chunk)

1. chunk is some integer which defines how many iterations per block or thread. Default is total_iter/thread_count

---

Assignment

Created: *31/05/2018 11:34 PM*
Updated: *9/06/2018 12:44 PM*

What will need to be done initially?

1. Load entire Iris data set (*training* set).
2. Define y, := test instance.
3. Define k, the number of neighbors (or threads).

What will need to be done, in parallel?

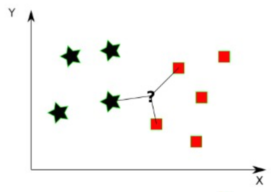
1. Calculate distance between test & training vectors.

2. Update list of k closest neighbors.

Barrier

1. At the end, the class with majority of k neighbors is selected/output.

In other words, finding the closest k neighbors, then the test data point will belong to the class which has the most neighbors.



So, the class with the most neighbors is the 🟥 (2 v 1). **?** then belongs to 🟥

1. Let k, be the number of processes, but also the number of closest neighbors to compute.
2. Each thread finds the closest neighbor for... problem: there may be a second point in another data set closer than a point in another data set.

DESIRE: Minimize communication, MPI is the choice since for larger data sets, they require more memory while there isn't as much computation being carried out. This enables the ability to use many nodes to have their individual memory (i.e servers), rather than having it on a single node. This makes the system expandable.

There is a testing set consisting of fifteen vectors.
Now, there is a training set which contains all the data minus the testing set. This will be 135 vectors.