School of Electrical Engineering and Computing

SENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS

(S1, 2020)

Concurrency and Java Threads

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

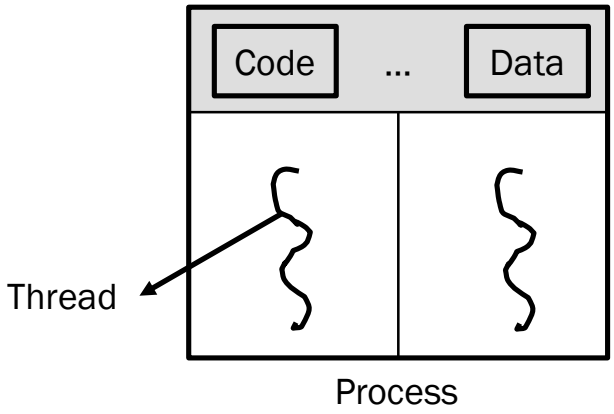
THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

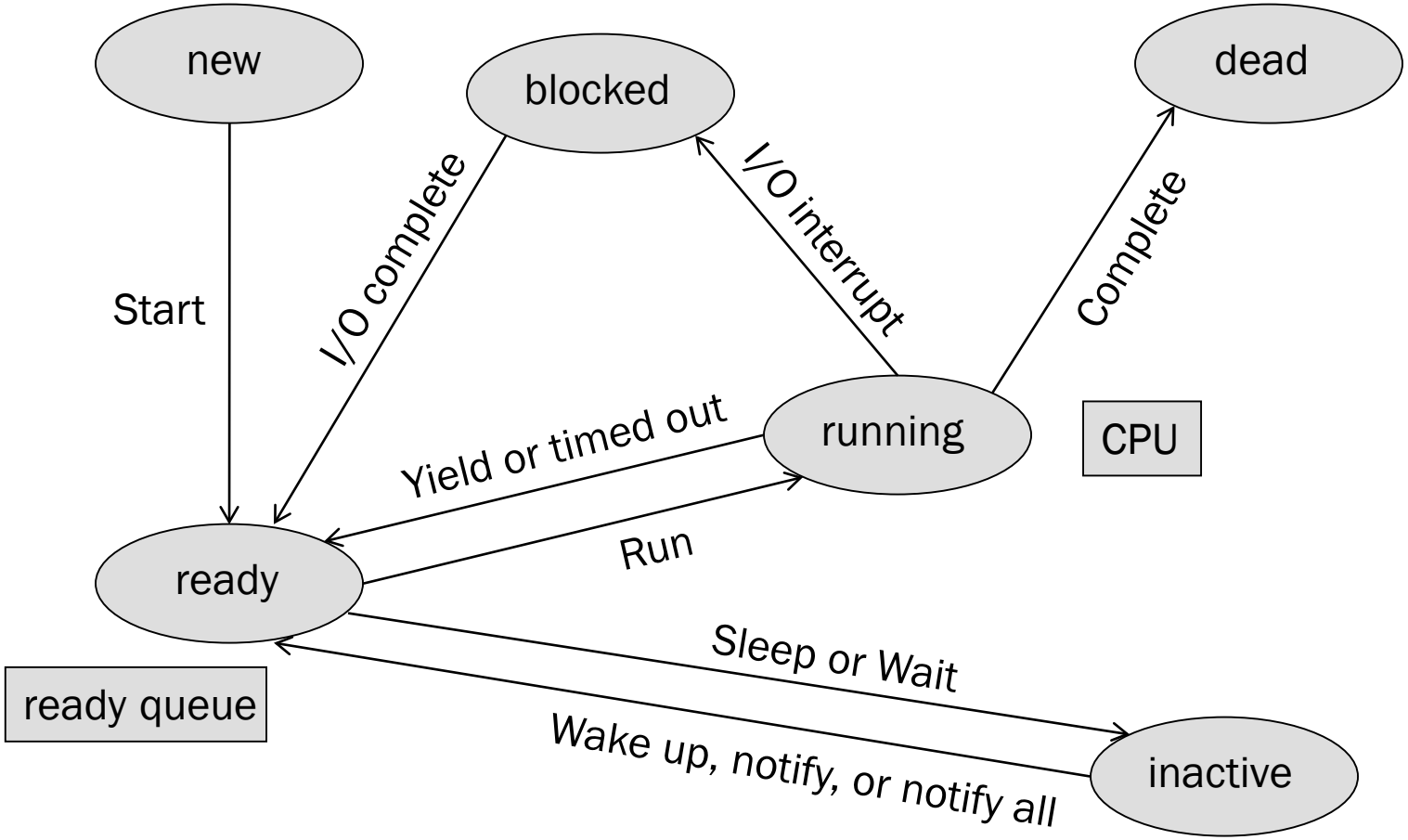
- Introduction to Concurrency and Resource Sharing
- Problems with Concurrency, Cooperation, and Access to Resources
- Java Synchronization and Threads

Introduction

- Concurrency: more than one process/thread operating at the same time.
- Process
  - *A program in execution*
  - *Process control block*
- Thread
  - *A segment of a process (as a lightweight process)*
  - *Share memory*
  - *States: ready, running, blocked.*



Processes and Threads Lifetime



Sharing Resources

- It happens all the time:
  - *Programs running in windows that share access to the monitor*
  - *Output being spooled to laser printers*
  - *Even the computer's CPU is a resource that needs to be shared between competing processes, otherwise the overall efficiency of the system is reduced dramatically.*
- For CPU:
  - *We need hardware support by way of interrupts, which transfer execution to an operating system that will load another process onto the CPU for it to execute.*
- For other resources, e.g., memory:
  - *We can reduce the problem to one of sharing a single piece of memory coherently between multiple processes.*

Sharing the CPU

- **Interruption** works by having an external event trigger a status bit within the CPU, which is checked as part of the fetch-execute cycle of the CPU.
- If the interrupt status is set, then the CPU executes code that will look after the interrupt. One such interrupt can be from the clock on a regular basis, preventing a CPU-intensive program from making other programs look like they are not making any progress at all.

```
while not halt do
        check interrupt status
            if set then handle the interrupt
        fetch next instruction
        decode instruction
        fetch operands
        execute instruction
```

Sharing Memory

- The main implication of this is that as a process executes a program written in a high-level language, we can no longer look upon even the smallest statement as atomic.
- Eg: x = x + 1;
  - *In assembly code this could look something like:*

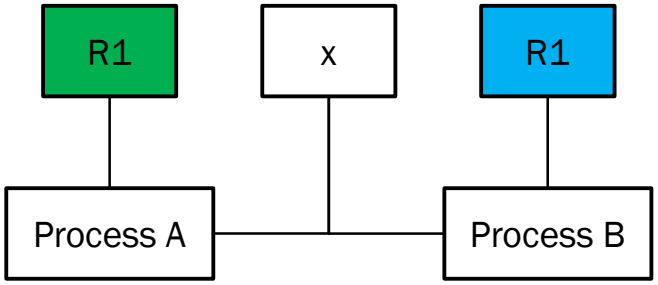
```
Load x into R1   // R1 is a CPU register
Add 1 to R1
Store R1 back into x
```

- What could happen if two processes are sharing the memory location x?

Sharing Memory

x = x + 1

If *x* starts off at *0*, what is the result of *x* following the execution of Process A and B below?



Process A

```
Load x into R1
Add 1 onto R1
Store R1 into x
```

interrupt →

← interrupt

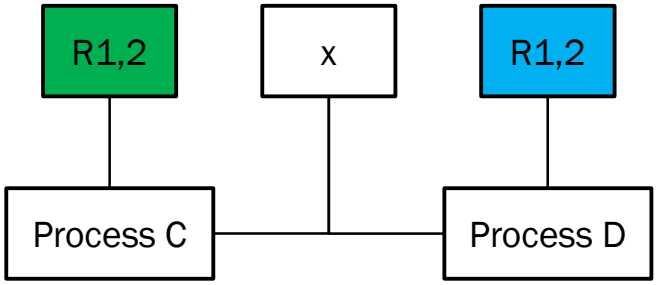
Process B

```
Load x into R1
Add 1 onto R1
Store R1 into x
```

x=1

Sharing Memory

If *x* starts off at *0*, what is the result of *x* following the execution of Process C and D below?



Process C

```
for(i=0; i<10; i++)
  x = x + 1;
```

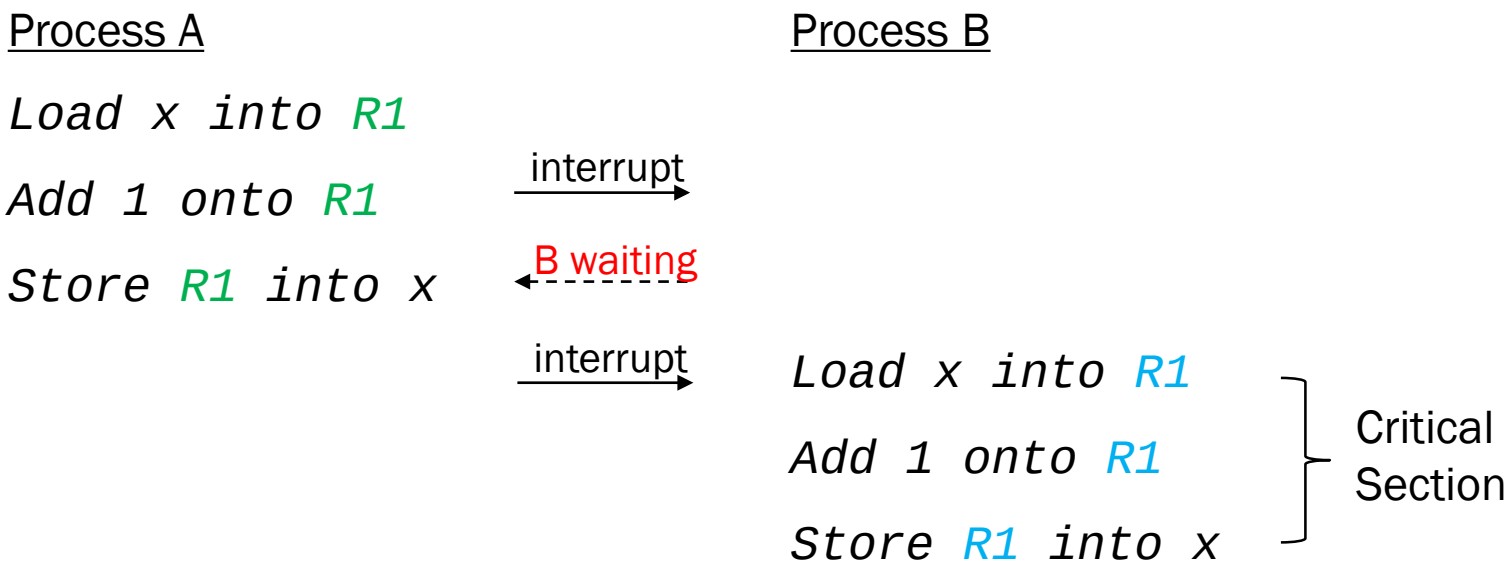
Process D

```
for(i=0; i<10; i++)
  x = x + 1;
```

Mostly *x* would be *20*, but the result is non-deterministic. When execution of the processes overlaps then other values are possible.

Mutual Exclusion

- **Critical section**: a section of code that uses the shared resource which must not be used by another process at the same time.
- **Mutual exclusion**: when one process is running a **critical section**, no other process can be in the same critical section that accesses the shared resources.



Mutual Exclusion

- To support the mutual exclusion mechanism, we need operations at least:
  - *Test: whether the shared resource is available.*
  - *Set: change status of the shared resource (critical section)*
    - lock/unlock
- Have a single machine instruction which does *TestAndSet*. There are many forms of this instruction.

```
TestAndSet T0          // check if T0 is available
  if (T0==0)           // resource T0 not available
     try again later
  else
     … use the resource …
…
```

Deadlock and Starvation

- **Competition**: many processes compete for shared resource access.
  - *Only one process can safely access the shared resource at a time.*
  - *A process is unaware of the others.*
  - *A process should not be affected by others.*
  - *E.g., Mutual exclusion, deadlock, and starvation problems.*
- **Deadlock:** two or more processes are unable to proceed because each is waiting for one of the others.
- **Starvation:** a process is overlooked indefinitely that it is unable to access the resources.
- Scheduling algorithms: assign resources to a process.
  - *E.g., First-Come-First-Serve (FCFS)*

Semaphores

- **Semaphore** supports simple signals to solve the resource sharing problems for concurrent processes.
- Keep a note of which resource(s) the process is waiting for.
- Use a counter to know how many of those resources are available.
  - *For the mutual exclusion problem, this counter will be initialised to 1.*

Semaphores

- A semaphore has two essential operations
  - *wait: decreases the semaphore (counter) value. If the value becomes negative, then the process becomes blocked.*
    - It “locks” the resource.
  - *signal: increases the semaphore (counter) value. If the value becomes non-positive, then the blocked process becomes unblocked.*
    - It “releases” the resource.

Semaphore s
int counter *q queue* *int status*
void wait() *void signal()*

wait()

- Let us first assume that s.counter is initially 1, ie that we are solving a mutual exclusion problem:

```
BEGIN: TestAndSet s.status,T0
            if (T0==0) go to BEGIN
            Subtract 1 from s.counter
            if (s.counter < 0) then
                  add this process onto s.queue
                  store 1 into s.status
                  suspend this process
            else
                  store 1 into s.status
            return to caller
```

- Control will not return to the point beyond the suspend until the resource is available for the process to use.

signal()

```
BEGIN: TestAndSet s.status,T0
          if (T0==0) go to BEGIN
          Add 1 to s.counter
          if(s.counter <= 0) then
             remove next process from head of
             s.queue and make it ready to run
          store 1 into s.status
          return to caller
```

- The counter can only remain negative or zero after it has been incremented if there is at least one process waiting for access to the resource.
- Why is the busy waiting in wait() and signal() not a worry?

Mutual Exclusion with Semaphores

- Semaphore s, with s.counter initialised to 1.
  - *The initial value of counter is important for a semaphore works correctly.*
- *n* processes competing for access to their critical sections.

```
P1: s.wait()                …   Pn: s.wait()
      critical section1     …         critical sectionn
    s.signal()              …       s.signal()
```

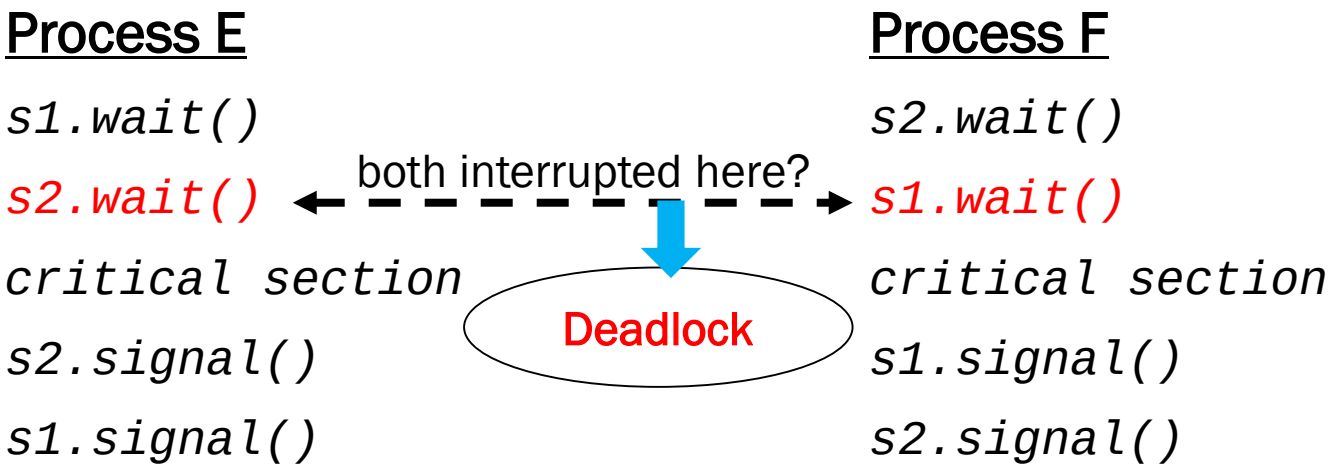
- Not only do we have mutual exclusion, we have first-come-first-served access, which guarantees that each process will eventually gain access to its critical section.

Resource Sharing in General

- What if we initialise s.counter to a number other than 1?
- For any positive number *n*, the first *n* processes to do s.wait() will be allowed through and subsequent ones will wait on s.queue.
- So if we have *n* resources to share, all we do is initialise s.counter to *n*.
- What if we initialise s.counter to 0?
- Some process must do s.signal() *BEFORE* any process will be allowed through s.wait(). We can use this to indicate events, and to synchronise processes. s.counter is just a resource count and if it is initially 0, then the resource (ie the event) just hasn't been created yet.

Deadlock with Semaphores

- Semaphores can solve lots of concurrency problems, but they also expose a new set of problems, especially when processes are competing for multiple resources.
- Suppose processes E and F both require resources 1 & 2, controlled by semaphores s1 & s2 (counters initially 1).



Deadlock with Semaphores

- Note that semaphores don't cause deadlock - they simply solve other problems to the point where the problem of deadlock can be exposed easily.
- **Starvation** would occur with semaphores similarly as the deadlock problem.
- Semaphore usually uses FCFS for the semaphore queue.
- Also note that we are not talking about a process just being slowed down by others, but about processes continually "getting in front of" a process to the point where it will never get the resources it needs.

Some Other Problems

- Semaphores can be misused - it is up to the programmer use always the wait( ) and signal( ) operations at the correct time
- Otherwise, it will compromise the correctness of the semaphore and therefore EVERY process that tries to access the resource(s) controlled by it, for example,
  - *one process not releasing a resource*
  - *one process releasing a resource "twice"*

Monitors

- Semaphores is somehow difficult to produce a correct program since the wait and signal operation are scattered throughout a program.
- This gives rise to the idea of a Monitor which provides an abstract data type which will do Mutual Exclusion - making it safe.
  - *Shared data is resident within the Monitor rather than in any of the client units*
  - *Access mechanisms are part of the monitor*
  - *Guarantees synchronised access - allowing only one access at a time*
  - *A monitor is effectively a Mutual Exclusion Abstract Data Type*

Classic Resource Sharing Problems

- Mutual Exclusion
  - *Only one process is allowed to use a resource at any one time*
- Synchronization
  - *One process is not allowed to proceed beyond a particular point until another process gets to a particular point in its execution.*
- Bounded Buffers
  - *Many processes sharing mutually exclusive but temporary access to a suite of N buffers.*
- Producer / Consumer
  - *Two or more cooperating processes continually produce intermediate results and then consume them.*
- Multiple Readers / Single Writer
  - *Any number of processes are allowed to concurrently read a file (or memory area) but only one process is allowed to write to it at a time, and only then when no-one is reading.*

Synchronisation

Process G:

```
pre-processing
wait for Process H
continue processing
```

Process H:

```
pre-processing
allow Process G to go
continue processing
```

Semaphore `s` (counter initially `0`)

Process G:

```
pre-processing
s.wait()
continue processing
```

Process H:

```
pre-processing
s.signal()
continue processing
```

It is possible to implement any synchronisation graph in this way.

Classes of Synchronisation

- Competition Synchronisation
  - *Mutual Exclusion - All competition for exclusive access to resources can be controlled via access to a single location in memory as that location can be seen as "permission" to use the resource, 0 for locked, 1 for available.*
  - *If you are the one to lock the controlling memory location, you have use of the resource.*
- Cooperation Synchronisation
  - *Simple synchronisation of processes can be seen as one process creating a resource (an event) and the other waiting for that resource to become available (waiting for the event).*
  - *E.g., producer consumer - one creates resource, other consumes the resource - events are resources in their own right. This often uses Mutual Exclusion for coherent access to the data structures which allow cooperation.*

Bounded Buffer Problem

- A system has N buffers to be used for whatever processes may wish.
- A process requests a buffer and then has sole access to that buffer until it releases the buffer.
- The buffers are homogeneous, any free buffer is able to satisfy any request.
- Competition for the buffers should be FCFS.

Solution:

An array that shows which buffers are free and which are in use.

Mutual exclusion on access and update of this array.

A semaphore which will allow up to N requests for a buffer to be satisfied.

```
Semaphore mutex (init 1)
Semaphore freebuffer (init N)
Array [1..N] of integer freelist
```

Bounded Buffer Solution

Semaphore mutex (init 1)

Semaphore freebuffer (init N)

int freelist[N] (initially 1...1)

int i // local variable or param

Request a Buffer (return buffer#)

```
int_buffer#   i
freebuffer.wait()
mutex.wait()
i = firstfree(freelist)
freelist[i] = 0
mutex.signal()
return i to caller
```

Release a Buffer (int_buffer# i)

```
mutex.wait()
freelist[i] = 1
mutex.signal()
freebuffer.signal()
return to caller
```

What if these lines were reversed?

mutex, freebuffer and freelist are shared, as are the buffers themselves.

Within the array freelist, 1 shows that the buffer with that id is not in use, while 0 shows it is being used by some process.

The Dining Table

The Dining Philosophers

5 philosophers are seated around a table, with a bowl of noodles in the middle of the table and one fork between each pair of philosophers (5 forks in total).

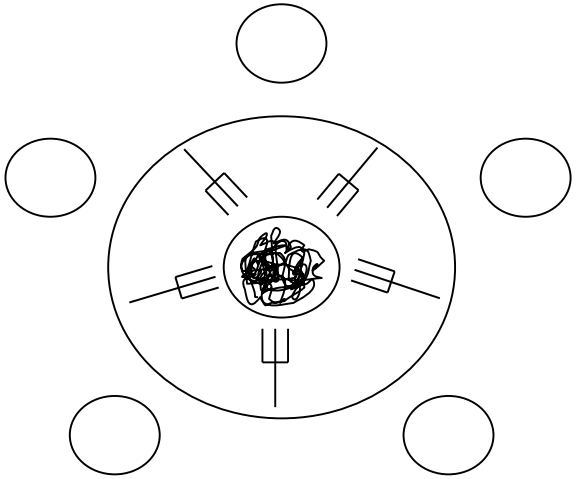
A philosopher process will:

repeat forever do
think
pick up my 2 forks
eat
put down forks

Rules:

Philosophers, forks, etc are homogeneous.

Solution must not allow either deadlock or starvation.



Java Threads

- Thread is a standard Java class
  - *It is not the natural parent of any other class*
  - *The only mechanism available in Java for creating concurrent programs*
- When a Java program begins, a thread is created to run the main method.
- Subsequent threads effectively run in competition with main and with each other
  - *They all compete for the same CPU(s) under the control of the JVM scheduler.*
  - *Since they share the same address space, they need to do their processing in a way that does not affect other threads and give non-deterministic results.*
- java.util.concurrent package was released in 2004

Java Threads

- The main operations on a Thread are
  - *start()* *command a thread to run*
  - *run()* *the algorithm the thread computes*
  - *join()* *wait until this thread finishes then continue*
  - *yield()* *give up the processor and got to ready queue*
  - *sleep(int)* *thread goes "off-line" for "int" millseconds*
- Cooperation between threads can be done using the methods.
  - *wait()* *suspend execution (must use try … catch)*
  - *notify()* *resume (make ready) the longest waiting thread*
  - *notifyAll()* *sends all waiting threads to the ready queue*
- A thread stops by reaching the end of its run() method
- The Java keyword synchronized allows an object to obtain a lock that will prevent a method or statement from being interrupted by another synchronized method – effectively creating a Monitor for that object.

Create Thread Using Thread Class

- Extend class Thread and instantiate it as an object that runs in/with/under its own thread of control

```
class MyThreadClass extends Thread {
 public void run()
 {//overrides Thread's run method}
}
…
Thread aThread = new MyThreadClass();
aThread.start();
```

- *The MyThreadClass object referred to by the Thread reference aThread goes onto the ready queue calling aThread.run().*

Create Thread Using Runnable Interface

- If the class to run the thread has a natural parent (extends it) then it cannot extend Thread, so the standard interface Runnable is implemented and the class "wrapped" in a thread by an explicit call to the Thread constructor.

```
class MyRunnableClass extends MyParent implements
Runnable {
        public void run() {//implementation}
}
…
Thread bThread = new Thread(new MyRunnableClass());
bThread.start();
```

- The MyRunnableClass object referred to by bThread calls bThread.run() and is placed on the ready queue.
- Both the aThread object (that is, the object referred to by the aThread reference) and the bThread object (ditto) are Java threads that are executing concurrently, and will be scheduled to run by the JVM scheduler.

Competition Synchronisation in Java

- Uses the Java keyword synchronized
- Java provides two basic synchronization mechanisms:

1. **synchronized** methods
   - *Thread must obtain the object's lock before invoking this method.*
   - *Multiple **concurrent** invocations of **synchronized** methods are not allowed on the same object.*

```
class ManageBuffer {
   private int [100] buf;
   …
   public synchronized void deposit(int) item {}
   public synchronized int fetch() {}
   …
} // shows synchronized methods
```

Competition Synchronization in Java

2. **synchronized** statements
   - *Uses some other object reference as a **lock***
   - *A **block** of statement that cannot be run concurrently with the same object lock.*

```
void method() {
        …
    synchronized (expression) {
      //  expression must evaluate to an object
            reference
    }
        …
}
```

- Queues needed to hold waiting threads are implicit within the JVM.

Where to (concurrently) from here?

- Remember that this is only a simple introduction.
- You probably don't know enough about threads in Java to sit down and start using them straight away – you will need more study of them to use them properly.
- The java.util.concurrent package significantly increased the capabilities for controlling the execution of threads.
- You will study a lot more to do with Synchronization, Concurrency, and Resource Sharing in Operating Systems

Example - The Producer/Consumer Problem

```
// We begin with the shared data structure.
// This is a circular Array Queue so has a fixed
// capacity.
class Queue {
     private int[] que;
     private int nextIn, nextOut, filled, queSize;
     public Queue (int size) {
          que = new int [size];
          filled = 0;
          nextIn = 0;
          nextOut = 0;
          queSize = size;
     }  // end of constructor for Queue
```

Example - The Producer/Consumer Problem

```
public synchronized void deposit (int item) {
      try {
            while (filled == queSize) {wait();}
            que[nextIn] = item;
            nextIn = (nextIn + 1) % queSize;
            filled++;
            notifyAll();
      } catch (InterruptedException e) {
            // ignore for this example}
}
```

Example - The Producer/Consumer Problem

```
public synchronized int fetch () {
     int item = 0;
     try {
          while (filled == 0) wait();
          item = que[nextOut];
          nextOut = (nextOut + 1) % queSize;
          filled--;
          notifyAll();
     } catch (InterruptedException e) {
          // ignore for this example}
     return item;
  }
} // end of class Queue
```

The Producer Class

```
class Producer extends Thread{// places items into Queue
      private Queue buffer;
      public Producer (Queue que) { buffer = que; }
      public void run () {
        int newItem;
        while (true) {  // loops forever producing items
            // -- code to Create a newItem
          buffer.deposit(newItem); //synchronized method
        }
      }
}
```

The Consumer Class

```
class Consumer extends Thread {// fetch items from Queue
      private Queue buffer;
      public Producer (Queue que) { buffer = que; }
      public void run () {
        int storedItem;
        while (true) {  // loops forever consuming items
          storedItem = buffer.fetch();// synchronized method
                // -- code to consume the storedItem
        }
      }
}
```

Instantiate the objects and Run

- We can create a Queue
- Create a Producer Thread object that references the Queue
- Create a Consumer Thread object that also references the Queue
- Set them both running

```
Queue buff1 = new Queue(100);
Producer p1 = new Producer(buff1);// or Thread p1 = …
Consumer c1 = new Consumer(buff1);// or Thread c1 = …
p1.start();
c1.start();
```

Using Runnable Interface

```
class Producer implements Runnable { … }
```

```
Producer pNotThreadYet = new
Producer(buff1);
Thread pThread = new Thread(pNotThreadYet);
pThread.start();
```

This will run in exactly the same way as the previous example of the Producer class.