School of Electrical Engineering and Computing

SENG2200/6220 Programming Languages & Paradigms

Topic 8

Concurrency and Java Threads

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic Overview

Introduction to Concurrency and Resource Sharing

Problems with Concurrency, Cooperation, and Access to Resources

Java Synchronization and Threads

Topic 8 - Part 1

Introduction to Concurrency, Resource Sharing, Processes and Threads

Introduction

Concurrency means “at the same time”, so concurrent processing means processing different things “at the same time”.

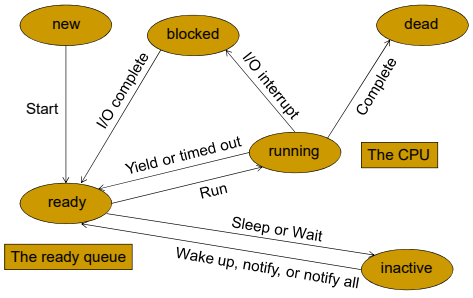
Process
- an active entity with a stack and a point of control (PC position)

Categories of concurrency
- physical – more than one processor is available
- logical – the system looks like it is processing things at the same time
- multithreaded – one program with several points of control
- statement-level – fine-grained parallelism, e.g. for loop iterations.

Processes
- heavyweight – each has its own address space
- lightweight – all share the one address space (common name is a thread)

Languages with Design for Concurrency
- Include Ada95, Java, C#, Python, and Ruby.
- Their ultimate predecessors are PL/1 (1960s) & Concurrent-Pascal (1972)

Processes and Threads



States in the Lifetime of a Thread

Classes of Synchronization

Competition Synchronization
- 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 Synchronization
- simplest is 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.
- Simple synchronization 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).

Topic 8 - Part 2

Problems with Concurrency, Cooperation, and Access to Resources

Part 2 – Outline

The problems inherent in processes sharing concurrent access to the computer's resources,

The basic requirements of processes sharing concurrent access to the computer's resources,

What policy issues must be addressed when sharing resources,

Solutions to classic resource sharing problems.

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.

First to the 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:

- We can reduce the problem to one of sharing a single piece of memory coherently between multiple processes. If we can do this, then we can share any resource. (Why?)

Sharing the CPU

Interrupts work 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 other resources

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 (or even x++ if you like)

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
```

Having a special "increment" instruction doesn't help, the problem just reappears in a different form later.

Now look at what can happen if two processes are sharing the memory location x.

A Problem Manifest

PROCESS A:

Load x into R1 // a CPU register

interrupt

Add 1 onto R1

interrupt

interrupt

Store R1 into x

interrupt

interrupt

Remember that x is shared but R1 is not, the x is the same in each process, but each process has its own version of R1.

If x starts off at 0, what value does it hold once both processes have tried to increment it?

PROCESS B:

Load x into R1

Add 1 onto R1

Store R1 into x

The interaction does not need to be this complicated. Once process A has loaded x then any interaction that results in process B loading the value of x before process A has stored into it, will have the same final result.

Answer: 1

The solution: we need a way to make the x = x + 1 atomic for each process.

13

A further example

Initially: *shared integer x = 0;*

PROCESS C:

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

PROCESS D:

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

What value does x hold after both processes have completed?

Machine code for the loop body:

```
START: if (R2>=10) Go to END:
       Add 1 to R2   // i ⟶ R2
       Load x into R1
       Add 1 to R1
       Store R1 into x
       Go to START
END:   continue ……
```

Mostly it would be 20, but the result is non-deterministic.

When execution of the processes overlaps then other values are possible.

What is the range of final x values possible after both C & D complete the loop?

14

Implications for Resource Sharing

If we can find a way to (safely and deterministically) share a single piece of memory then we can provide coherent access to scarce resources by many competing processes.

Eg. A laser printer - obviously we only want one process writing to the printer at any one time.

The solution is to have a piece of memory shared between processes that will tell whether someone else is currently using the printer (a simple 0/1 variable).

A process checks the "laser printer status", if it's not in use (ie status is 1) then the process sets it to be in use (sets status to 0), and begins using the laser printer.

Once finished printing, the process resets the status to 1.

15

Two processes in competition

Our counter problem occurs again but in slightly different form:

Process A:

Load status into R1

if (R1!=1) try again later

Store 0 into status

… A uses the resource …

Store 1 into status

Process B:

Load status into R1

if (R1!=1) try again later

Store 0 into status

… B uses the resource …

Store 1 into status

… use the resource … is referred to as a ***critical section***

The classical name for this problem is ***mutual exclusion***

16

What is needed?

What is the minimum solution to this problem?

We need to be able to *Test and Set* the status variable in an *atomic* operation. Once the operation has begun, it cannot be interrupted until it has completed.

Given the way that interrupts get checked within the fetch-execute cycle of the computer, the only way to do this is to have a single machine instruction which does *TestAndSet*. There are many forms of this instruction.

```
TestAndSet  status, R1    // sets status to 0 and puts the value
if (R1==0) try again later //    that status used to have into R1
… use the resource …
Store 1 into status
```

You need to satisfy yourself that this will solve the mutual exclusion problem.

17

Try again later ???

We need a way for a process to check whether a resource is available, and if the resource is not available, for it to wait *off-line* until the resource is available.

Also, we need a way to ensure that competition between the processes for any resource is *fair*.

Firstly, we need to define the terms *off-line* and *fair*.

Off-line: Not using the CPU while waiting for the resource.

Fair: It's up to the particular system, we'll choose First-Come-First-Served. *(Why is this a good choice?)*

We already have operating system support to switch processes onto and off the CPU via interrupts.

The solution to "try again later" is to use TestAndSet to build a more powerful abstraction for resource sharing.

18

The Semaphore

Notice the problem becoming more general.

We now have the operating system keep note of which resource the process is waiting for -> a process queue (ie a queue of processes) for each set of resources.

We add to this a shared counter variable that will show how many resources are available. For the mutual exclusion problem, this will be initialised to 1.

This counter and queue will be protected by way of a TestAndSet status variable (initially 1), similar to before.

This becomes an object which we will call a semaphore s, and it will have 2 distinct behaviours:

s.wait( ) **wait until a resource is available.**

s.signal( ) **release the resource.**

What will these methods do?

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

19

s.wait( )

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

```
BEGIN: TestAndSet s.status, R1
       if (R1==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.

20

s.signal( )

```
BEGIN: TestAndSet s.status, R1
       if (R1==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?

21

Mutual Exclusion with Semaphores

Semaphore s, with s.counter initialised to 1.

note: the key to making semaphores work is the initial value of the counter.

n processes competing for access to their critical sections:

P_1:	s.wait( )	…	P_n: s.wait( )
	critical section$_1$	…	*critical section*$_n$
	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.

22

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.

23

non panaceae sunt

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 A and B both require resources 1 & 2, controlled by semaphores s1 & s2 (counters initially 1).

Process A		**Process B**
s1.wait( )	both interrupted here?	s2.wait( )
s2.wait( )		s1.wait( )
use the resources		*use the resources*
s2.signal( )	Processes wait for each other → **Deadlock**	s1.signal( )
s1.signal( )		s2.signal( )

24

Deadlock and Starvation

Note that semaphores don't cause deadlock - they simply solve other problems to the point where the problem of deadlock can be exposed easily.

Another problem to be exposed is **starvation** - when a group of processes are able to interact in such a way that some other process is prevented from obtaining the resources it needs to continue.

Note that this was the reason that we said that FCFS was a good choice 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.

non panaceae sunt (et cetera)

25

But deadlock is not the only problem with semaphores

- They solve the problem of resource competition and cooperation provided they are used properly
- BUT … Semaphores can be misused - it is up to the programmer use always the wait( ) and signal( ) operations at the correct time
- not doing so will compromise the correctness of the semaphore and therefore EVERY process that tries to access the resource(s) controlled by it.
- E.g.
  - one process not releasing a resource
  - one process releasing a resource "twice"
  - Etc.
- A more complicated structure (in our case a Monitor) partially solves this problem by turning the semaphore into an Abstract Data Type --- covered in Part 3

Classic Resource Sharing Problems

26

Mutual Exclusion: *done.*
- 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

27

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.

Bounded Buffer Problem

28

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

29

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 Classic Problem

30

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:

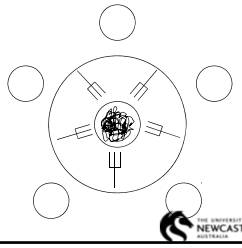
repeat forever do
think
pick up my 2 forks
eat
put down forks

Why is this **THE** classic problem?

Rules:

Philosophers, forks, etc are homogeneous.

Solution must not allow either deadlock or starvation.



Topic 8 - Part 3

From Semaphores to Monitors,

Java Synchronization and Threads

Monitors

Semaphores - initial value is the resource count - 0 for a synch event, 1 for Mutual Exclusion, n for n buffers (-N if waiting for N events).

Misuse of Semaphores - it is up to the programmer always use wait and signal at the correct time - not doing so will compromise the correctness of the semaphore and therefore EVERY process that tries to access the resource controlled by it.

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 synchronized access - allowing only one access at a time
- A monitor is effectively a Mutual Exclusion Abstract Data Type
- Can be implemented in Java with the keyword synchronized

Java Threads – The *Thread* Class

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 – significant step fwd

Java Threads – The *Thread* Class (2)

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.

Java Threads – Creating a Thread using *Thread*

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( ).

Java Threads – Using the *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 ( ) { … implements run( ) method … }
}
…
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 Synchronization in Java

37

Uses the Java keyword synchronized - Sebesta example (ch13, p610)

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 (cont)

38

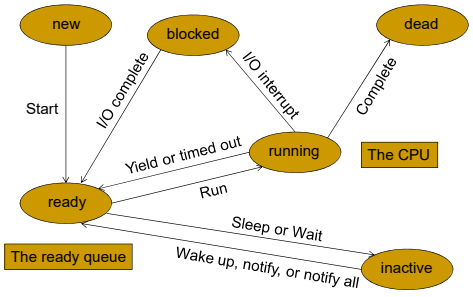
2. **synchronized** statements
   - Uses some other object reference as a **lock**
   - A **block** of statement that cannot be run concurrently with the same object lock.

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

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

Processes and Threads

39



States in the Lifetime of a Thread

Where to (concurrently) from here?

40

Remember that this is only an 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/Arbitration in Operating Systems – don't tell your lecturer that you have done it all before (this is a very simplistic introduction).

Java Example – The Producer/Consumer Problem

41

```
// Sebesta example (10e, Ch13, pp609-11)
// We begin with the shared data structure
class Queue {          // note this is a circular Array Queue
   private int[ ] que;    // and so has a fixed capacity
   private int nextIn, nextOut, filled, queSize;
   public Queue (int size) {
       que = new int [size];
       filled = 0;
       nextIn = 0;    // see notes later – text initialises to 1
       nextOut = 0;  // as it mis-translates from ADA
       queSize = size;
   } // end of constructor for Queue                    …
   /over
```

(cont)

42

```
public synchronized void deposit (int item) {
    try {
        while (filled == queSize) wait( );
        que[nextIn] = item;
        nextIn = (nextIn + 1) % queSize;   // the text is
        filled++;                  // in error here, (ADA code)
        notifyAll( );              // (nextIn % queSize + 1) ?
    } catch (InterruptedException e) {
        // ignore for this example
    }
}                                  // … /over
```

(cont)

```
public synchronized int fetch ( ) {
    int item = 0;
    try {
        while (filled == 0) wait( );
        item = que[nextOut];
        nextOut = (nextOut + 1) % queSize;
        filled--;                    // text has ADA again
        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
        }
   }
}
```


Now to instantiate the objects and run/run (or start/start)

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* …

```
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.