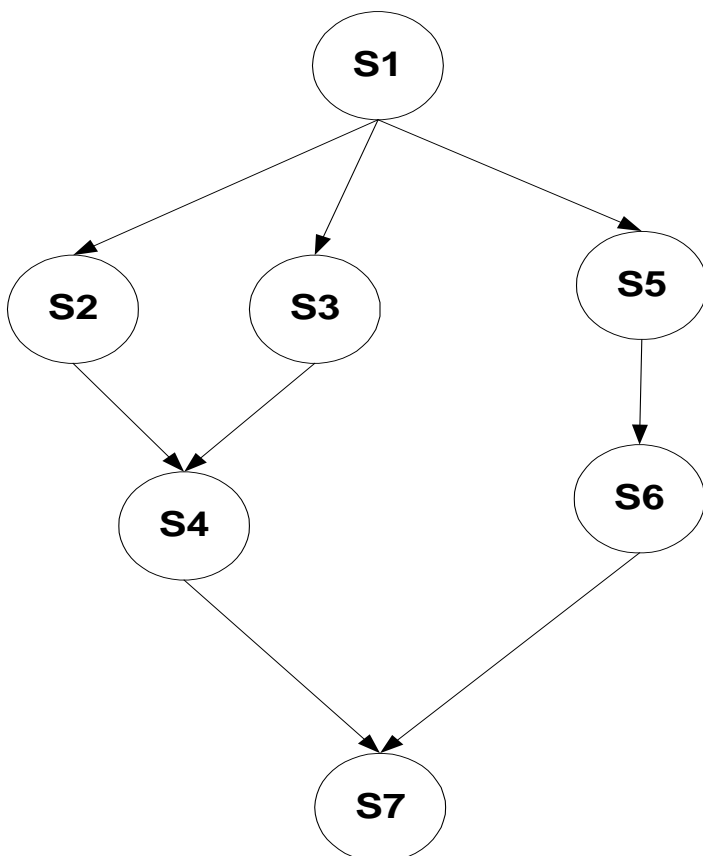


Multithreading and Java

Concurrent Programming

Intro on concurrency



fork/join

begin

```
S1;  
count1 = 2;  
count2 = 2;  
fork L3;  
fork L5;  
S2;  
goto L4;
```

L3: S3;

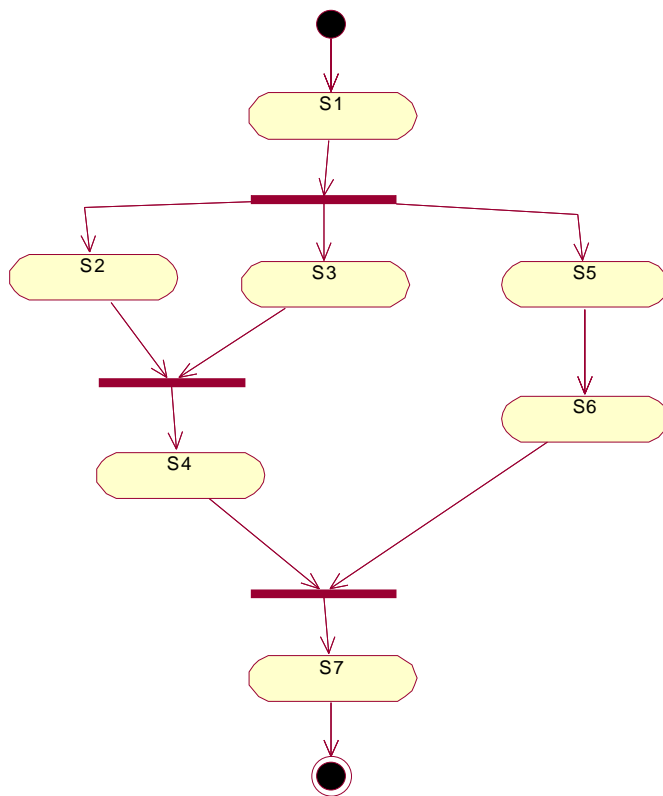
```
L4: join count1;  
S4;  
goto L7;
```

L5: S5;

S6;

```
L7: join count2;  
S7;
```

end



parbegin/parend

```
begin main
  parbegin
    lhs
    rhs
  parend
  S7
end main
```

```
begin lhs
  parbegin
    S2
    S3
  parend
  S4
end lhs
```

```
begin rhs
  S5
  S6
end rhs
```

Critical Sections and Mutual Exclusion

When two processes share access to a memory location, an object or a file, its data can become corrupt or inconsistent if both processes attempt to modify it simultaneously.

A **Critical Section** is a section of code where a process accesses a shared resource.

Example – no overlap

Process 1	shared variable X	Process 2
.	42	.
.	42	.
X.increment()	42	.
.	43	.
.	43	X.increment()
.	44	.
.	44	.
X.increment()	44	.
.	45	.
.	45	.
.	45	X.increment()
.	46	.
.	46	.

Example – access overlaps

Process 1	shared variable X	Process 2
.	42	.
.	42	.
X.increment()	42	X.increment()
.	43	.
.	43	X.increment()
.	44	.
.	44	.
X.increment()	44	X.increment()
.	45	.
.	45	.
.	45	.

Value of X is inconsistent with no of increment operations.

We need to ensure **mutual exclusion**, i.e. only one process can access a shared resource at a given time.

Suppose, we associate a condition with entry into the critical section. The condition will be true if the critical section is not occupied and will be false otherwise. Will it ensure mutual exclusion?

for example:

```

Process1
{
    do something;
    while(true)
    {
        ...
        ...
        while(available == false)
            ;           // loop repeatedly doing nothing

        available = false; // enter the critical section
        X.increment();

        available = true;  // exit the critical section
        ...
        ...
    }
}

```

Process2 will be similar

What happens if process1 tests *available*, finds it free and is about to enter the critical section when it is preempted and the CPU is passed to Process2? Process2 will also find *available* set to true as process1 hadn't set it to false before being preempted. Process2 will then duly proceed into the critical section. If the CPU is returned to process1 while process2 is in the critical section, it will also enter the critical section.

Clearly this method won't guarantee mutual exclusion.

We need to be able to test and set the variable *available* in one operation without the possibility of the operation being suspended while still incomplete (operation needs to be **atomic**).

Even, if this approach worked, one is left with a *busy-waiting* while loop.

Semaphores

Synchronisation primitive for ensuring mutual exclusion

Used to synchronise access to a critical section.

Due to Dijkstra

A semaphore uses a non negative integer variable (s say) which apart from initialisation can only be accessed by two **atomic** operations. It is used as a *lock*

```

wait(s)    P
signal(s)  V

```

```

wait(s)
{
    if( s <= 0) then
        s = s - 1
    else
        add process to the semaphore queue
        suspend process
}

```

```

signal(s)
{
    if (semaphore queue is empty)
        s = s + 1
    else
        remove process from queue and make runnable
}

```

Semaphore has:
 an integer variable
 2 atomic operations
 a queue for block processes waiting on the semaphore

It is implemented partly in hardware and partly in software in the OS kernel

Process1 from above could now be written as:

```

Process1
{
    do something;
    while(true)
    {
        ...
        ...
        wait(mutex);           // try to enter the critical section
                               // mutex is a semaphore
        X.increment();
        signal(mutex);        // exit the critical section
        ...
        ...
    }
}

```

Monitors

The correct use of semaphores was left to the programmer.

Programmer could easily forget a wait() or signal() operation and could get different semaphores confused.

Semaphore was too low level a construct to be used reliably by programmers

Tony Hoare (1974) proposed a programming language construct called a monitor which would remove the responsibility for wait() and signal() calls for mutual exclusion from the programmer.

A monitor is an abstract data type, with both internal data structure and associated operations on the data structure. i.e. its internal data is encapsulated and hidden from the outside world being only accessible through the operations.

Also, the monitor has built in mutual exclusion, so that only one of its operations may be run at a given time

Thus, the responsibility for mutual exclusion shifts to the compiler.

However, responsibility for other types of synchronisation remain with the programmer. So there are still two operations similar to wait() and signal() available to the programmer – but not for mutual exclusion - see consumer producer example below.

Java and Synchronisation

Java uses the concept of the monitor as a synchronisation mechanism.

In Java a monitor is a class whose methods can only be executed by one thread at a time.

```
class Counter {
    private long count = 42;

    public synchronized void increment() {
        count++;
    }

    public synchronized void decrement() {
        count--;
    }

    public synchronized long value() {
        return count;
    }
}
```

Example – access synchronised

Process 1	shared variable X	Process 2
.	42	.
.	42	.

```
X.increment() 42    X.increment() /blocked
.    43    .
.    44    .
.    44    X.increment()
.    45    .
.    45    .
X.increment() 45    X.increment()/blocked
.    46    .
.    47    .
.    47    .
```

Support for Multithreading in Java

Two ways to set up your own threads in Java

- 1) extend the Thread class
- 2) define a class which implements the Runnable interface

Class java.lang.Thread

```
java.lang.Object
|
+----java.lang.Thread
```

```
public class Thread
extends Object
implements Runnable
```

Example 1

MyThread.java

```
public class MyThread extends Thread {

    int n = 10, id;

    public MyThread( int ident) {
        id = ident;
    }

    public void run() {
        for(int i=0; i<n; ++i)
            System.out.println("Hello from thread " + id);
    }
}
```

MyThreadApplic1.java

```
public class MyThreadApplic1 {

    public static void main( String argv[] ) {
        MyThread myThread1, myThread2;

        myThread1 = new MyThread( 1);
        myThread2 = new MyThread( 2);

        myThread1.start();
        myThread2.start();
    }
}
```

```

        System.out.println("Hello from main thread ");
    }
}

```

```
D:\My Documents\Concurrency Seminar>javac MyThreadApplic1.java
```

```
D:\My Documents\Concurrency Seminar>java MyThreadApplic1
```

```
Hello from main thread
```

```
Hello from thread 1
```

```
Hello from thread 1
```

```
Hello from thread 1
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 1
```

```
Hello from thread 1
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 1
```

```
Hello from thread 1
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 1
```

```
Hello from thread 1
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 2
```

```
Hello from thread 1
```

```
D:\My Documents\Concurrency Seminar>
```

Interface `java.lang.Runnable`

Public interface `Runnable`

This provides another way to create threads in Java.

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

An object which implements the `run()` method may be passed as an argument to the thread constructor - see example below.

```
RunnableClass.java
```

```
public class RunnableClass implements Runnable {
```

```
int n = 10, id;  
  
public RunnableClass( int ident) {  
    id = ident;  
}  
  
public void run() {  
    for(int i=0; i<n; ++i)  
        System.out.println("Hello from thread " + id);  
}  
}
```

MyThreadApplic2.java

```
public class MyThreadApplic2 {

    public static void main( String argv[] ) {
        RunnableClass ro1, ro2;
        Thread MyThread1, MyThread2;

        ro1 = new RunnableClass( 1);
        ro2 = new RunnableClass( 2);

        MyThread1 = new Thread( ro1);
        MyThread2 = new Thread( ro2);

        MyThread1.start();
        MyThread2.start();

        System.out.println("Hello from main thread ");
    }
}
```

Thread Methods

`start()` causes a thread to call its `run` method as an independent activity. None of the synchronisation locks held by the caller thread are automatically retained by the new thread. Unless a special control method (such as `stop`) is called on the thread, it terminates when the `run` method returns.

`isAlive()` returns true if a thread has been started but has not terminated. (It will return true if the thread is merely suspended in some way.)

`stop()` irrevocably terminates a thread. This is the most common way of terminating threads. Stopping a thread does *not* in any sense kill the Thread object, it just stops the activity. However, stopped methods cannot be restarted. When terminated and no longer referenced, Thread objects may be garbage collected.

A more extreme variant, `destroy()`, stops and kills a thread without giving it or the Java run-time system any chance to intervene. It is not recommended for routine use.

`suspend()` temporarily halts a thread in a way that will continue normally after a (non-suspended) thread calls `resume` on that thread.

`sleep()` causes the thread to suspend for a given time (specified in milliseconds) and then automatically resume. The thread might not continue immediately after the given time if there are other active threads.

`join()` suspends the *caller* until the target thread completes (that is, it returns when `isAlive` is false). A version with a (millisecond) time argument returns control even if the thread has not completed within the specified time limit.

Monitor Methods

The equivalent of `wait()` and `signal()` in Java are `wait()` and `notify()`.

The methods `wait()`, `notify()`, and `notifyAll()` may be invoked only when the synchronisation lock is held on their targets. This is normally ensured by using them only within methods or code blocks synchronised on their targets. Compliance cannot usually be verified at compile time. Failure to comply results in an `IllegalMonitorStateException` at run time.

A `wait()` invocation results in the following actions:

The current thread is suspended.

The Java run-time system places the thread in an internal and otherwise inaccessible *wait queue* associated with the target object.

The synchronisation lock for the target object is released, but all other locks held by the thread are retained. (In contrast, suspended threads retain *all* their locks.)

A `notify()` invocation results in the following actions:

If one exists, an arbitrarily chosen thread, say *T*, is removed by the Java runtime system from the internal wait queue associated with the target object.

T must re-obtain the synchronisation lock for the target object, which will *always* cause it to block at least until the thread calling `notify()` releases the lock. It will continue to block if some other thread obtains the lock first.

T is then resumed at the point of its `wait()`.

A `notifyAll()` invocation works in the same way as `notify` except that the steps occur for all threads waiting in the wait queue for the target object.

Consumer Producer Example

CPBuffer.java

```
public class CPBuffer {
    protected int[] buf;
    protected int hi, lo, size;

    public CPBuffer() {
        hi = lo = size = 0;
        buf = new int[10];
    }

    public synchronized void write( int n) {
        while( size == 10)
            try {
                wait();
            } catch( InterruptedException e) {return;}

        buf[hi] = n;
        hi = (hi + 1) % 10;
        ++size;

        notify();
    }

    public synchronized int read() {
        while( size == 0)
            try {
                wait();
            } catch( InterruptedException e) { return 0;}

        int save;
        save = buf[lo];
        --size;
        lo = (lo +1) % 10;

        notify();
        return save;
    }
}
```

ProducerThread.java

```
public class ProducerThread extends Thread {
    protected CPBuffer cpbuf;
    private int id;

    public ProducerThread( CPBuffer cpbuff, int ident) {
        cpbuf = cpbuff;
        id = ident;
    }

    public void run() {
        int n;
        while(true) {
            n = (int) (Math.random() * 100);
            cpbuf.write(n);
            System.out.println( "Producer " + id + " wrote " + n);
        }
    }
}
```

ConsumerThread.java

```
public class ConsumerThread extends Thread {
    protected CPBuffer cpbuf;
    private int id;

    public ConsumerThread( CPBuffer cpbuff, int ident) {
        cpbuf = cpbuff;
        id = ident;
    }

    public void run() {
        int n, i;
        for(i=0; i<10; ++i) {
            n = cpbuf.read();
            System.out.println( "Consumer " + id + " read " + n);
            try { sleep(100); } catch (InterruptedException e)
                {return;}
        }
    }
}
```

ConProApplic.java

```
public class ConProApplic {  
  
    public static void main( String arg[])  
    {  
        CPBuffer cpb;  
        ProducerThread pT;  
        ConsumerThread cT1, cT2, cT3;  
  
        cpb = new CPBuffer();  
  
        pT = new ProducerThread( cpb, 1);  
        cT1 = new ConsumerThread( cpb, 1 );  
        cT2 = new ConsumerThread( cpb, 2 );  
        cT3 = new ConsumerThread( cpb, 3 );  
  
        cT1.start();  
        cT2.start();  
        cT3.start();  
        pT.start();  
    }  
}
```

Communicating Processes

Communication between processes(threads) can be achieved through
shared memory
message passing
input/output streams
remote links - RMI, CORBA
channels

However, shared memory requires explicit synchronisation implemented by the programmer.

Situation can easily get out of hand due to strong coupling between methods/threads via the synchronisation locks and wait()/notify() calls. Difficult to reason about behaviour of an individual thread. One would have to consider the simultaneous behaviour of all other threads to which it is coupled.

Thus can be very difficult to rule out
deadlock
livelock
starvation
(race hazard is prevented by the monitor synchronisation/mutual exclusion)

A CSP type channel offers a more reliable way for processes to communicate

A channel is an intermediate object between 2 processes which allows 1-way communication

A channel encapsulates
synchronisation
scheduling
data transfer

Once channels are in place, the programmer is the need for explicit synchronisation between processes.

It is much easier to ensure correct behaviour of communicating processes which are built around channels and other CSP constructs as CSP
has a solid mathematical foundation
is supported by a number of CASE tools for detecting deadlock, livelock, starvation etc.