UNIVERSITY OF LIVERPOOL

COMP319 Actor Model Notes

Background

One of the problems with the standard way of handling threads is the issue of thread deadlock, this can happen in a number of ways but let us look at a specific example.

```
public void transfer(Account fromAccount, Account toAccount,    int amount) {
        synchronized (fromAccount) {
                synchronized (toAccount) {
                        if (fromAccount.hasSufficientBalance(amountToTransfer) {
                                fromAccount.debit(amountToTransfer);
                                toAccount.credit(amountToTransfer);
    }
}
```

This code attempts to lock two accounts before transferring money between them using the Java synchronized apart from this being relatively slow it is prone to a problem called deadlock.

If thread 1 calls the code like this transferMoney(account1,account2) and then thread 2 calls the code like this transferMoney(account2,account1). If then thread 1 is pre-empted after locking account1 and just before locking account 2 and thread 2 is allowed to proceed, thread 2 will lock account 2, thread 2 will then deadlock when trying to lock account 1. Thread 1 will also be deadlocked trying to lock acccount 2.

There are other ways that deadlock can occur but generally it always involves some type of circular locking dependency as explained before.

Testing for deadlock

For this error to be revealed the timing characteristics are particularly stringent, the second thread has to pre-empt the first thread at a very particular time (just between the 2 statements). This may happen under very heavy load conditions but may happen very rarely in practise.

One way of dealing with this deadlock is to modify the code to keep the order of locking always the same, here is some re-engineered code without this problem:

The code can be changed by locking the accounts always in the same order. So change to something like this:

```
if (fromAccount.getId()<toAccount.id()) {
    synchronized(fromAccount) {
        synchronized(toAccount) {
                // call transfer code
        }
    }
} else {
    synchronized(toAccount) {
        synchronized(fromAccount) {
// call transfer code
        }
    }
}
```

Thread starvation

One of the other issues with the standard thread model is something called thread starvation. If multiple threads are waiting on a monitor and the monitor is released, one of the waiting threads will be released, the problem however in Java™ this does not need to be the thread that has waited longest, this can lead to threads waiting and waiting for service and not getting any CPU time. This situation is call thread starvation.

Introduction to the Actor model

The Actor model is a way of modelling concurrency in which state data is not shared between different threads. In the Actor model each thread has some data which it is allowed to read and write, in this way there is not the need for the complex system of synchronisation that is usually seen with other concurrency models. It also eliminates the problems of deadlock and starvation which are present in classic threading model where data is shared.

If an actor wishes to read data from or write data to another actor, it does this be sending the other actor a message.

Actor example operation

For example if an actor representing a client wanted to get the account balance from another account representing a bank account it would do this by sending the actor a message like this.

Client Actor sends message Request Balance to account Actor

The Account actor has an inbox which stores all its messages, when it processes its incoming stream of messages it can then act on the incoming message and send a message back to the client actor containing the balance

Account Actor ---> BalanceUpdataMessage ---> Account Actor

Imagine a client Actor wants to transfer some money from Account1 to Account 2, the process would be like this:

Client makes up unique transaction Id.

Client Actor ---> TransferAmount(Account1,Account2,amount,transactionId) ---> Account 1 Actor

Account 1 when it gets the message, checks the balance if the balance is too small it sends a message back to the Client rejecting the transfer, if the balance is ok, it re-calculates the balance

Balance=Balance-Amount

It pushes the message onto Account2

Account 1 Actor ---> AddAmount(Client,amount,transactionId) --> Account 2 Actor

Notice in the message is the id of the original client, this is so when the process is finished Account 2 can send a confirmation back to the Client

Account 2 ---> ConfirmTransfer(transactionId) ---> Client Actor

Notice the transaction can be always checked by using the unique transaction id, this is usually a concatenation of the ClientID, Account 1 and Account 2 ids and the current time stamp.

Actor features
So Actors can do the following:

Create other Actors
Send messages to other Actors
Receive and act on incoming messages from other Actors

Messages between Actors should be immutable, this means the messages themselves cannot be modified after sending.

There are two possible ways a message can be sent, by reference (i.e. a pointer to the message is sent by the Actor), this is the fastest approach but you can also send the message by value, this means all the data must be copied to the other Actor. If two Actors are on physically separate systems, i.e. connected over a network, the need to use pass by value (pointers do not make sense between separate systems).

Features and benefits of the Actor model
Each Actor has a working thread which is used to process incoming messages.
Each Actor owns its own internal data set and cannot directly read or write any other data in the system.
Actors can be on totally separate systems, so you can have one actor running on a computer in London, sending messages to two other Actors one in New York and another in Shanghai. They can all be part of the same software system and communicate via the Internet.

No deadlock
Given that data is not shared between Actors it is not possible to get deadlock in the Actor model.

Scalable
With the Actor model it makes it easier to build massively scalable software systems since the processing can be spread across a large number of physical systems, if you need more power or capacity to your application you can just add in more computers which are running more actors.

Actor addressing
Actor message addressing is transparent, this means then you send a message to another actor you don't know if it is local (in the same computer) or remote on a separate computer. This communication is commonly done using a message or object broker.

Fair scheduling
When multiple actors are running in a given system the actor model ensures that each one is given a fair amount of processing time, this eliminates possible problems with thread starvation.

Actor mobility
Sometimes it is important in the actor model to be able to move actors between systems for the following reasons:

Locality of reference (Move two actors closer together possibly onto the same machine to speed up communications)
Load sharing (move an actor to a less busy machine)
Reliability (move an actor from a machine that is not working well to another machine)

There are two possible ways an actor can provide mobility, strong and weak mobility.

In strong mobility the Actor and all its state can be transfer to the new machine.

In weak mobility the Actor is transfer when its message queue is empty and then transfered to the new machine with some possible initialisation state.

To transfer an actor from 1 machine to another, you need to do the following, stop the actor, capture the state of the actor, creation of new actor, delete the old actor, restoring the state to the new actor, handling addressing (re-direct messages to new actor).