Comp 319 Tutorial Actor

Actor model and concurrency

Question 1 Define the following terms:

Monitor, task, thread and deadlock.

A thread of execution is a sequence of running instructions within a computer program that is managed by the operating system scheduler, with a JVM multiple threads can be running through 1 program. Because many threads can be sharing the same data, there can be problems when 1 thread is interrupted and another thread takes over. For this reason when operating on shared data it is common practise to use mechanisms to ensure that only 1 thread is allowed to access to the data at one time.

A set of tasks are different from a set of threads, in that each task has its own database and does not usually directly share memory with other tasks apart from with carefully designed mechanisms.

One of the mechanisms used to keep access to data exclusive is called a monitor. The monitor only allows 1 thread to execute at a time and locks out all other threads. If the 1 thread is currently using a monitor and another second thread requests to use it, the second thread is blocked until the first thread has finished.

If a thread requires access to 2 different sources of shared data it may well need a monitor for both lots of data, i.e. 2 monitors. This in certain cases can to deadlock situation. For example if we have 2 monitors and 1 thread locks on 1 monitor and a 2^{nd} thread executes on the 2^{nd} monitor, if they then both try and lock on the other monitor, they will both a locked up for ever, as they cannot proceed and will be waiting for the other 2 release the others monitor.

Thread starvation

More than 1 thread can be waiting on a monitor at once. When the monitor becomes free, 1 of the threads is woken up. It is possible for 1 of the threads to be left waiting and the other woken up, because there is no defined protocol to make sure that threads are treated fairly then it is possible for a thread to be starved of execution time and end up not running.

Question 2 Why can deadlock be difficult to determine by code testing?

In practise deadlock faults are sensitive to timing, this means that only in very particular timing conditions will the deadlock occur, this may be when the system is under great load. Even so in general relying on testing to reveal deadlocks in code can be hit and miss. For this reason deadlock should be eliminated by careful design decisions (for example locking all monitors in a fixed sequential order).

Question 3 What is the lost update problem?

The lost update problem can occur in the case where data is read from memory, altered and then written back to memory. For example look at the following line of code

sum = sum + balance

One thread could read the balance and add it to the sum, a second thread could then pre-empt the first thread read the balance and add it to the sum and write the result back in the sum, when the first thread is re-started it will store the sum back and the 2nd update will be lost.

Question 4 Describe the following features of the Actor model

With the actor model, data is not shared between each thread. Each thread acts on its own data set. Actors communicate with each other by sending messages, the messages are stored in mailboxes and the actor acts on each message in the mailbox in turn. Each message is sent asynchronously i.e. without waiting for a reply. Because each actor doesn't share data the problems with lost updates and deadlock do not happen.

The actor model defines a mechanism called fair scheduling, with fair scheduling each non blocked actor is guaranteed at least some execution time.

Actors can be executing on different machines at different geographical locations and still communicate. This is one of the important aspects of the actor model called location independence, i.e. the messaging between actors is the same if they are local or remote.

In a distributed architecture the use of the actor model allows the application to easily use many processing cores spread across many machine, this is very important since the ability to efficiently use multi-processor architectures in a distributed environment has always been a serious software engineering challenge.