SOLUTION NOTES

Concurrent Systems and Applications 2003 Paper 5 Question 4 (TLH)

This question is based on lectures 10-14 in the section *Communication within an address space*. The term *feeding philosophers* warrants some explanation. It is, of course, similar to the classical problem of the *dining philosophers* seated around a table with shared forks (or chopsticks and so on) between each pair. The placement of these feeding philosophers along a trough rather than at a table also draws on Roger Needham's categorisation of the quality of restaurants as *feeding*, *eating* or *dining*.

(*a*) *Mutual exclusion* refers to how at most one thread should be allowed to be executing within a critical region for a particular resource at any instant in time. In Java this is usually enforced by using the `synchronized` keyword on a method. This causes any thread invoking the method to have to acquire a mutual exclusion lock associated with the object on which the method is called, blocking until it is able to do so. Mutual exclusion is used to enforce safety properties.

(*b*) In Java, *deadlock* occurs when a group of threads is cyclically holding mutual exclusion locks required by others in the group. No thread can make progress and thereby release the locks that it is holding. The other "static" requirements for deadlock (holding resources while waiting, no revocation of resources, resource requests can be refused) all exist in Java. Deadlock relates to a lack of liveness.

(*c*) This can be implemented by a building a simple mutual-exclusion scheme, for instance:

```
class TroughPosition
{
  private boolean busy = false;

  synchronized void startEating()
           throws InterruptedException
  {
    while (busy) wait ();
    busy = true;
  }

  synchronized void finishEating()
  {
    busy = false;
    notify();
  }
}
```

Note the use of `synchronized` methods, the use of a `while` loop to guard the testing of `busy`, the use of the `private` modifier, the consideration of `InterruptedException` while waiting and the slight performance improvement of using `notify()` rather than `notifyAll()`.

(*d*) This solution cannot suffer from deadlock: each thread only ever needs one resource and so can never attempt to acquire a further resource while holding an existing one.

(*e*) This basic solution *can* suffer from starvation at the ends of the table. There is no attempt to ensure fair access to the `TroughPosition` and so, for instance, 1 and 2 could eat alternately while 9 starves. It could be avoided by enforcing FCFS access to the table, for instance by assigning philosophers sequentially numbered tickets when they `startEating()` and providing them with access in ticket order.