Operating Systems **CS355** Spring 2007

Lab 3: Concurrent Programming

In this lab, you will will be asked to develop a multithreaded system that is free of both deadlock and starvation, and in which the concurrency control does not dramatically decrease throughput.

Consider a system in which a ladder is stretched horizontally between two trees on opposite sides of an otherwise uncrossable river, and there are apes who wish to cross the river by brachiating along the ladder. The ladder is too narrow to allow apes to pass, so apes can't either pass each other or get by an ape going the other way on the ladder. Thus, the apes will need some sort of strategy for coordinating their access to the ladder. Your job is to develop such a strategy and use whatever Java mechanism(s) you wish to implement it.

Start by checking out the "ApesAndLadders" project. This will include the version of the problem shown in lecture, in which apes successively lock each rung of the ladder as they cross, but nothing but luck and careful timing of the creation of apes prevents deadlock.

1. Replace the body of the `main` method in class `Jungle` with code to create two threads, one of which generates (and `start`s) eastbound apes, and the other of which generates (and `start`s) westbound apes. Both threads should start right away, so if you do this correctly, the system will almost certainly deadlock.

   This is not conceptually difficult, but you will have to get good with techniques for creation of jobs in Java. You may want to use the current Ape class as a model or reference, as well as the course textbooks or Sun's Java web site.

2. Modify the program as necessary so that it is free of both deadlock and starvation. You should not degrade throughput in any way that changes the "O( )" complexity, e.g. by changing it from O(brachiating speed) to O(brachiating speed * number of rungs), as would happen if you successfully avoided deadlock by allowing only one ape to use the ladder at a time. If you make any assumptions about fairness of Java mechanisms such as `Semaphore`s or `wait` and `signal`, document them carefully with comments in your source code.

3. Create a file README.txt in which you give

   a) an overview of your approach, including an explanation of why it avoids deadlock and starvation, and

   b) a discussion of the impact of concurrency control on throughput, in which you discuss both the fundamental limits or conflicts inherent in the problem, and the strengths and weaknesses of your approach.

Remember to use "Team->Share" to submit your work when you are done, and at other points during development when you have completed a significant part of the work.