Programming project 1 — due on Tuesday 24 October 2000

You are to simulate the workings of an operating system. The "Warm-up exercise" handout outlined the core requirements; here we add resource management. The goal is to understand the details of implementing a deadlock detection algorithm, and to explore the tradeoff between the overhead of the algorithm and the frequency of execution of the algorithm.

On reusable-resource graphs. A reusable-resource graph is a directed graph, with two types of nodes, process nodes and resource nodes, and two types of edges, request edges and assignment edges. A request edge goes from a process node to a resource node to indicate that the process has requested one unit of the resource. An assignment edge goes from a resource node to a process node to indicate that one unit of the resource is assigned to the process. For each kind of resource, its total number of units is known.

We say that we *reduce the graph by a process* (which is neither blocked nor isolated) by removing all edges to and from the process. This makes that process isolated, and it can't be part of any deadlock. We say that a graph is completely reducible if there exists a sequence of reductions that deletes all edges of the graph. All reduction sequences of a given graph lead to the same irreducible graph. Deadlock exists iff the graph is not completely reducible. *Note: we will discuss this further in class.*

Processes. You are to implement resource management. Processes will perform *Request* and *Release* operations. These system calls take one time unit.

Occasionally the OS decides to check for deadlock. You are to implement deadlock detection using reduction of the reusable-resource graph. If deadlock is detected, the OS will choose one of the deadlocked processes and destroy it.

What should the simulator report? Report important events, such as process creation, completion, results of running the deadlock detection algorithm, process destruction events, etc.

You may also produce a more detailed action log, listing each system call in the simulator, or, if you are simulating at the architecture level, you can log individual instructions.

Required experiments. Simulate various processes making *Request* and *Release* calls in various orders, chosen so as to create deadlock, or not to create deadlock. You may need to use large numbers of processes.

1. Invoke the deadlock detection algorithm every 100 units of time. Measure the execution time of the deadlock detection algorithm. How you do this depends on the level of detail of your simulator. (If it's architecture-level, then you will count the number of instructions, or cycles, taken up. Otherwise, measure how long the *simulator* takes, but don't charge any time in the simulated OS.) Print the execution time for each invocation as well as a total for all invocations of deadlock detection over the entire simulation run.

2. Double the frequency of invocation, to once every 50 units of time; and, in general, vary the frequency.

Documentation. You should write a detailed document on the design of your simulator. Describe (1) what functions of the operating system it is simulating, at what level, and (2) how these functions are represented and mapped into your implementation programming language.

Grading. Full credit for this project is 100 points, as follows: Design document: 20 points; Implementation structure and code documentation: 20 points; Completeness of OS simulation: 20 points; Experimental evaluation and presentation of results: 20 points; Overall impression: 20 points; Extra credit: 20 points for experiments in the warm-up exercise, and 20 points to extend the deadlock detection mechanism to be able to tell how long a deadlock has existed once it is discovered and to evaluate experimentally the cost and the benefits of this extension.

Start early; focus on process management, system calls, and interrupts before you work on resources and deadlock detection. We shall review progress in class on Tuesday 10 October.