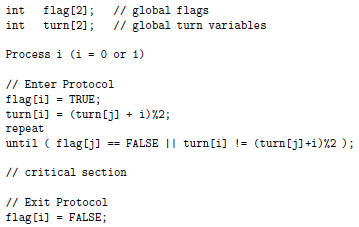
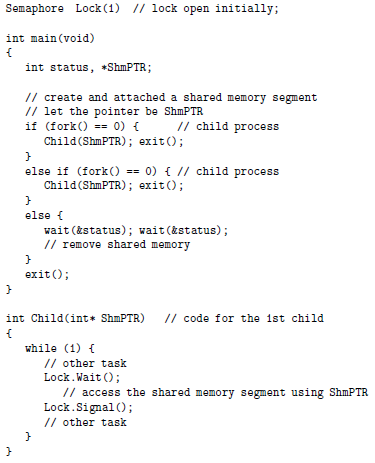
Exam 2 Fall 2014

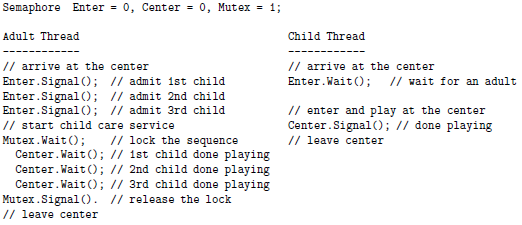
1. Consider the following solution to the mutual exclusion problem for two processes where flag is a Boolean array of two elements and turn is an int array, each of its two elements can only hold 0 or 1. Note that flag and turn are global variables shared by both processes. Prove that this solution satisfies mutual exclusion.



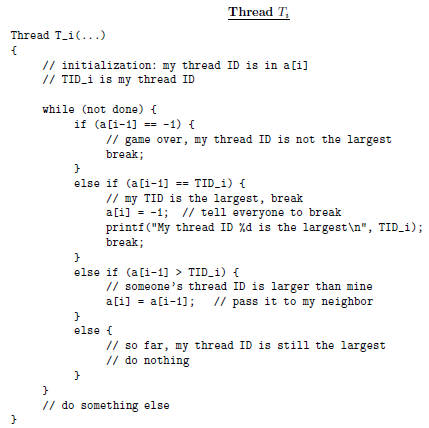
1. Define the meaning of a race condition. Use an execution sequence.
2. The semaphore methods wait() and signal() must be atomic to ensure a correct implementation of mutual exclusion. Use execution sequences to show that if wait() isn’t atomic then mutual exclusion can’t be maintained.
3. A programmer wrote a program in which child processes are created and communicate using a shared memory segment. This programmer uses the semaphore capability of ThreadMentor to avoid potential race conditions as follows. However, even though the initialization, process creation, and shred memory segments are correct, this program can never run properly. Identify the problem as clear as possible and provide a convince explanation. Use execution sequences if needed.



1. At a child care center, state regulation require there is always one adult present for every three children. When an adult comes to the center, a thread is created to simulate that adult. Similarly, when a child arrives at the center, a thread is created to simulate that child. A programmer suggested the following solution using three semaphores. The programmer insisted that the lock mutex can’t be eliminated, because a deadlock may occur when the child care center has a certain number of adults and children (i.e. 3 children and 2 adults). Find an explain this deadlock with an execution sequence.



1. Let T0, … , Tn-1 be n threads, and let a be a global int array. Thread Ti only have access to a[i-1] and a[i] if and thread T0 only has access to a[n-1] and a[0]. Each thread knows its thread ID, a positive integer which is only available to the thread. All thread IDs are distinct. Initially, a[i] contains the thread ID of thread Ti. We hope to find the largest thread ID of these threads. An algorithm is outlined below. Race conditions are everywhere, so declare and add semaphores to the code so that the task can be performed correctly. Use as many semaphores as you want, but thread Ti can only share its resources, semaphores included, with its left neighbor Ti-1 and right neighbor Ti+1. You may ignore T0. Explain why your implementation is correct in details.



1. Three kinds of threads share access to a singly-linked list: searchers, inserters, and deleters. Searchers examine the list, and can execute concurrently with each other. Inserters append new nodes to the end of the list and must be mutually exclusive. However, one insertion can proceed in parallel with any number of searches. Deleters remove nodes from anywhere in the list. At most one deleter can access the list at a time, and deletion must be mutually exclusive with searches and insertions. Searchers and deleters are the readers and writers in the reader-writer problem, with code below. Write the code for the inserter and add semaphores and variables as needed. Do not modify searcher and deleter. Provide an elaboration to show correctness.

