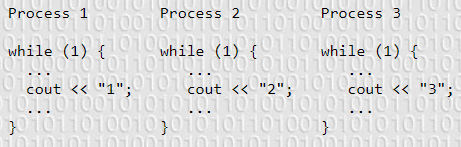
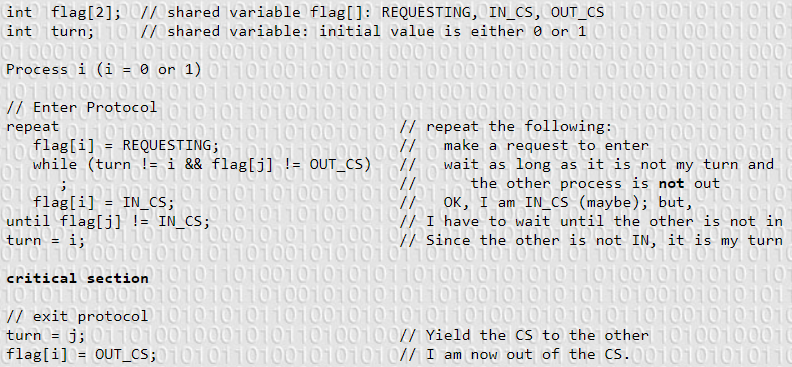
Exam 2 Questions

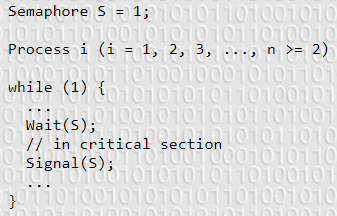
1. Suppose a public member function GetCounter() is added to a semaphore and it is atomic. If the counter value is negative, its absolute value is the number of processes in the waiting list. Discuss the accuracy of the returned value by member function GetCounter(). In order words, is it always the case that the returned value provides an accurate count of waiting processes?
2. Show by an execution sequence that if the method wait() is not atomic but signal() is then mutual exclusion can’t be guaranteed using a semaphore.
3. Show by an execution sequence that if the method signal() is not atomic but wait() is then mutual exclusion can’t be guaranteed using a semaphore.
4. The simple solution using the atomic TS instruction satisfies mutual exclusion but not bounded waiting. Does it satisfy the progress condition?
5. Suppose we have three processes and P­­I only prints the value of i as follows. Add semaphores and cout if necessary to the processes so that the three processes will print the sequence 1 2 3 2 1 2 3 2 1…



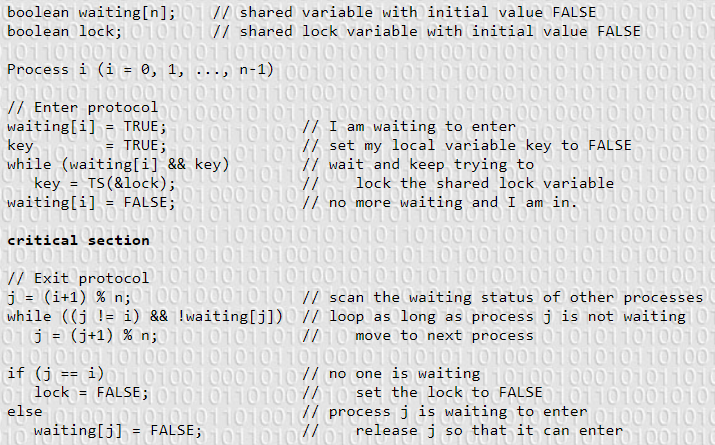
1. Modify the above template so that it will print out the following sequence 1 2 3 1 2 3…
2. We know that a semaphore with an appropriate initial value can guarantee mutual exclusion. Does the semaphore solution satisfy the progress and bounded waiting conditions? Use execution sequences to answer this question.
3. Use the Test and Set instruction to implement semaphore wait and signal.
4. It was shown that deadlocks can occur in a naïve solution to the dining philosopher’s problem. It was also shown to avoid deadlocks a righty philosopher may be introduced or add a restriction that at most four philosophers may sit down and eat. Do the following
   1. Prove that the righty solution doesn’t cause deadlock
   2. Prove that the four-chair solution doesn’t cause deadlock
   3. Show with execution sequences that the naïve, righty, and four-chair solutions can have starvation.
5. For the dining philosopher’s problem, add a single chopstick at the center of the table. A philosopher picks up his left chopstick and then competes to grab the chopstick at the center. If he can get both, then he eats.
   1. Does this approach avoid any possible deadlock?
   2. Are there any bad consequences that this attempt could have such as maximum parallelism?
   3. With semaphores, is it possible to implement the following protocol? Why? Discuss the efficiency and deadlock issues of this “suggested” solution.
      1. Each philosopher picks up his left chopstick
      2. If he fails o do so, then try to get the center one. Otherwise, he waits until his left chopstick becomes available.
      3. Do the same for his right chopstick.
6. For the dining philosopher’s problem, instead of having the chopsticks arranged as discussed, we make all five chopsticks in a tray next to the table so that any philosopher can pick any chopstick. In this attempt, each philosopher picks up his first chopstick then his second. Is deadlock possible. How about parallelism?
7. Consider the following solution to the critical section problem. The flag[] variable may be REQUESTING, IN\_CS, and OUT\_CS and the turn variable is initialized to 0 or 1. Show that the solution does satisfy all three conditions.



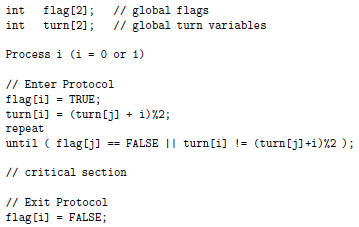
1. Show that if wait() and signal() are atomic then the following implementation of mutual exclusion is correct. More precisely, prove rigorously that the following code guarantees mutual exclusion among all processes.



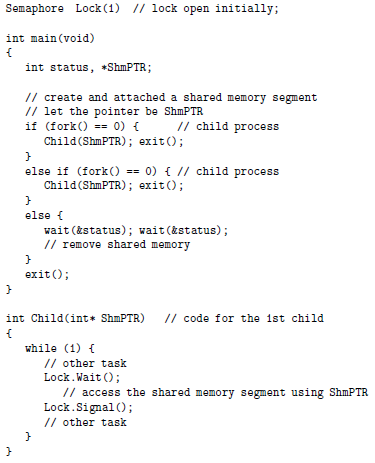
1. Consider the following solution using the atomic TS instruction. The shared variable waiting[] and lock are initialized to false. Show that the solution satisfies mutual exclusion, progress, and bounded waiting. If we have such a good solution that satisfies all three conditions, why isn’t the solution to the critical section problem implemented this way?



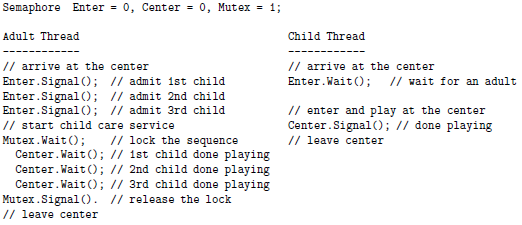
1. Consider the following solution to the mutual exclusion problem for two processes where flag is a Boolean array of two elements and turn is an int array, each of its two elements can only hold 0 or 1. Note that flag and turn are global variables shared by both processes. Prove that this solution satisfies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



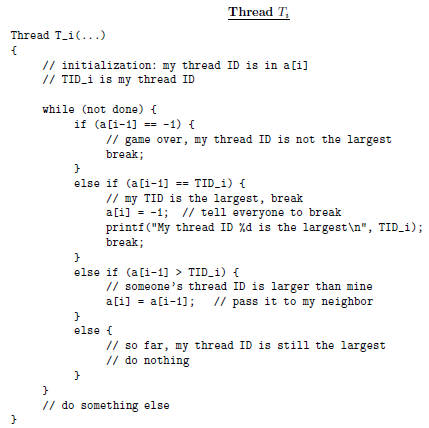
1. Define the meaning of a race condition. Use an execution sequence.
2. A programmer wrote a program in which child processes are created and communicate using a shared memory segment. This programmer uses the semaphore capability of ThreadMentor to avoid potential race conditions as follows. However, 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.



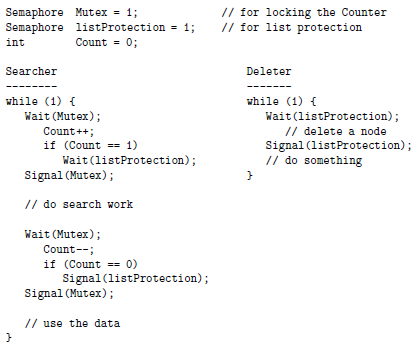
1. At a child care center, state regulation require there is always one adult present for every three children. When an adult comes to the center, a thread is created to simulate that adult. Similarly, when a child arrives at the center, a thread is created to simulate that child. A programmer suggested 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.



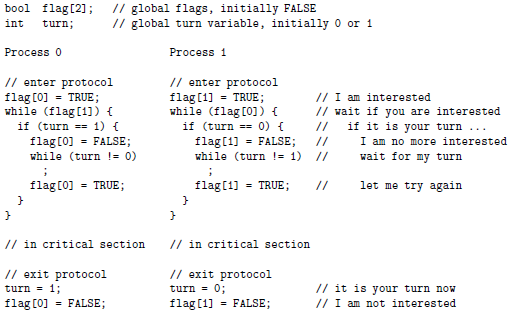
1. Let T0, … , Tn-1 be n threads, and let a be a global int array. Thread Ti only have access to a[i-1] and a[i] if and thread T0 only has access to a[n-1] and a[0]. Each thread knows its thread ID, a positive integer which is only available to the thread. All thread IDs are distinct. Initially, a[i] 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.



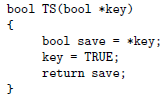
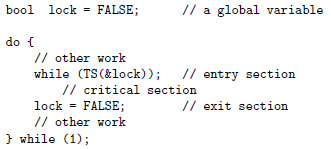
1. Three kinds of threads share access to a singly-linked list: searchers, inserters, and deleters. Searchers examine the list, and can execute concurrently with each other. Inserters append new nodes to the end of the list and must be mutually exclusive. However, one insertion can proceed in parallel 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.



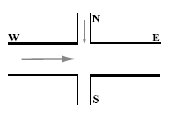
1. Consider the following solution to the critical section problem for two processes. Show this solution satisfies mutual exclusion.



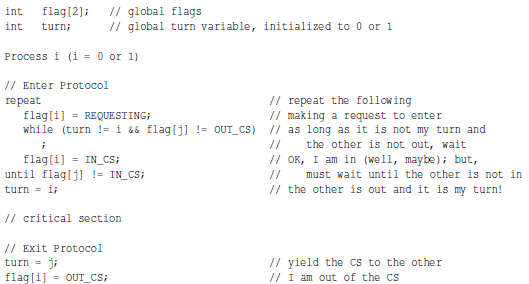
1. The TS instruction is atomic and has the following form. Consider the implementation of mutual exclusion by TS. Show the implementation satisfies mutual exclusion.

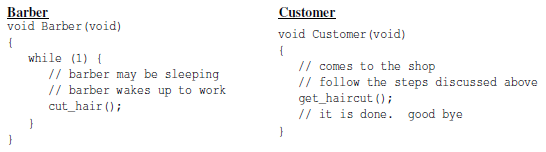
1. Show that the 1 weirdo solution to the dining philosophers problem will not cause circular waiting and is deadlock free.
2. A restaurant has n tables, each of which can only sit one customer according to the following rules. Design a customer thread with semaphore to simulate this activity.
   1. Initially, the restaurant has no customers and all tables are free
   2. When a customer arrives, if there is a free table and no one is waiting, he could sit down and order
   3. When a customer arrives, if all tables are occupied or there are waiting customers, he must wait until all eating customers finish and leave
   4. After finishing, a customer leaves
3. A main highway cuts through a rural road as shown below. East-bound cars are on the highway, while south-bound cars are on the rural road. To avoid delays on the highway, the following traffic regulations are implemented. Write code for the east-bound and south-bound and add semaphores and variables as needed.
   1. As long as there are east-bound cars, they don’t have to stop. In this case, south-bound cars must stop.
   2. If there is a south-bound car crossing, all east-bound vehicles must stop.
   3. To prevent south-bound cars from blocking the highway, only one south-bound car can enter the intersection. However, multiple east-bound cars may cross the intersection at the same time.
   4. If east-bound cars and south-bound cars approach the intersection at the same time, only one can proceed and it can be either one.



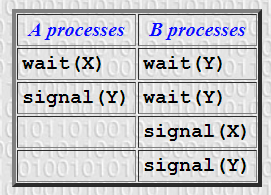
1. Consider the following solution to the mutual exclusion problem for two processes. Prove the solution satisfies mutual exclusion.



1. For the dining philosophers problem, suppose each chopstick is assigned a number. Each philosopher must pick up a chopstick of lower number, then of higher number after he has successfully obtained one. For example, philosopher 3 picks up chopstick C3 and C4. Is deadlock possible?
2. For the dining philosophers problem, suppose an additional chopstick is placed at the center of the table. A philosopher picks up his left chopstick and then competes to grab the chopstick at the center. If the chopstick at the center of the table has been taken, the philosopher tries to grab his right chopstick. Is deadlock possible?
3. A barber shop has one barber, one chair, and n chairs for waiting customers. The barber and customer activities are as follows. The barber is simulated by a thread. When he comes to work, he calls Barber(). The customers are simulated by dynamically created threads. When they need a haircut, they call Customer(). Write the code for these functions are add semaphores and variables as needed.
   1. The barber waits for customers
   2. If there is a customer, the barber brings a waiting customer to the chair and cuts his hair
   3. After serving a customer, the barber sleeps
   4. If there are no waiting customers, the barber sleeps
   5. When a customer arrives, he looks if there is a free chair in the waiting room
   6. If there is a free chair, the customer tells the barber there is a new customer, and sits and waits
   7. If there isn’t a free chair, the customer leaves

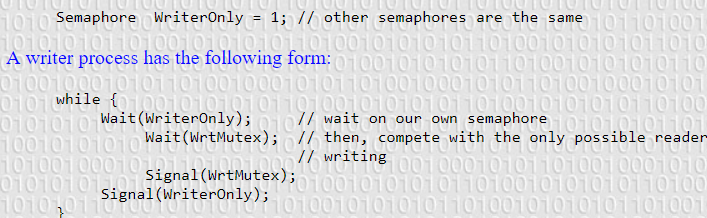


1. Suppose a system has two types of threads, type A processes and type B processes. All type A processes execute the same code and all type B processes execute the same code. There are two semaphores, X = 2 and Y = 0. The code of each process type is shown below. Answer the following questions. If it is possible, show the execution sequence and if not, provide an argument.

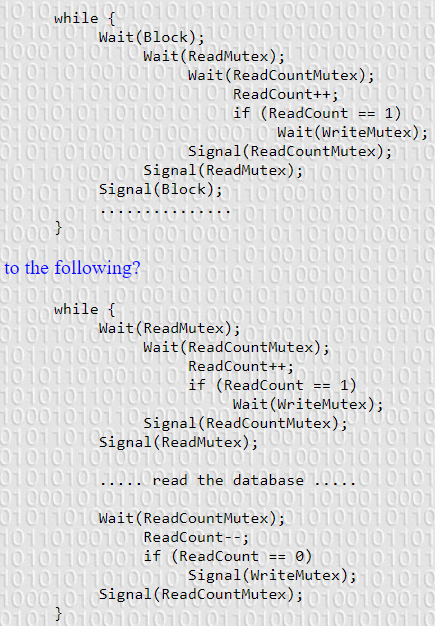


* 1. Is it possible for processes to finish in the order AABAB?
  2. Is it possible for processes to finish in the order AABBA?
  3. Is it possible for processes to finish in the order AAABB?
  4. Can the system deadlock?
  5. Modify the code so that processes must finish in the order AABAB.

1. Suppose the smokers problem is changed to the following version. There are three semaphores: one for tobacco, one for matches, and one for paper. The smoker who has tobacco waits on semaphores matches and paper, the smoker who has matches waits on semaphores tobacco and paper, and the smoker who has paper waits on semaphores tobacco and matches. The agent randomly picks up two ingredients and signals the corresponding semaphores. There is a table still. Does this version work properly without problems?
2. The above more general smoker version is due to waiting on multiple semaphores sequentially. Suppose there is a mechanism to collect multiple semaphores into a single unit so that a thread can wait on and signal multiple semaphores simultaneously in an atomic way. Would this mechanism solve this general smokers problem?
3. In the Reader/Writer problem, the wtrMutex is shared by the readers and writers. It blocks no more than one reader and multiple writers (prove this). As a result, when an exiting reader or writer signals the semaphore, a reader or writer could be released. Some argue that this doesn’t satisfy the reader priority requirement. Below is another reader priority solution with an added semaphore writerOnly with initial value 1. Is this a better version? In order words, does the only waiting reader on the wrtMutex semaphore have a better chance to take priority?



1. Below is a writer priority version to the Reader/Writer problem. The initial values of ReadCount and WriteCount are zero.
   1. What is the purpose of using wait(Block) and wait(ReadMutex)?
   2. You would say “semaphores Block and ReadMutex are used to lock the database” but if this were true, the database would by accessed by no more than one reader or one writer. Is this a correct observation?
   3. Why can readers access the database simultaneously?
   4. The initial value of Block is 1. Why don’t we change



Removing Block is the most natural suggestion, since it is used in the reader process and used exactly once. Explain why you can’t do this. If Block is removed, what happens to the solution?