EXAMINATION

NWEN 303 (2009)
Concurrent Programming
MARKING GUIDE

Time Allowed: 3 Hours

Instructions: Paper foreign language dictionaries are permitted.

Non-programmable calculators without full alphabetic keys are permitted.

Electronic dictionaries and programmable calculators are not permitted.

An appendix at the end of the exam lists some useful algorithms.

Answer all of the following **six** questions:

1. Synchronisation.
2. Sharing resources.
3. Shared data structures.
4. Channels and Data Spaces.
5. Mutual Exclusion and Consensus.
6. Global Properties.

Each question is worth 30 marks.

The exam is worth 180 marks in total.

continued...

Question 1. Synchronisation. [30 marks]

Suppose you wish to find some natural number, k, such that $f(k) = 0$, for a given function f, assuming that at least one such k exists.

The following is a rather poor attempt at constructing a concurrent algorithm to solve this problem using N processes, where the intention is that process i checks $f(i)$, $f(i+N)$, $f(i+2N)$, $\cdots$, and *one* process which finds a k such that $f(k) = 0$ prints that value of k:

```
Algorithm: Concurrent search (N processes)
integer k
boolean found := false
p(i), for i = 0, ..., N-1
k := i
while not found
    if f(k) = 0 then
        found := true
        print k
    else k := k + N
```

(a) [6 marks] Explain, with reference to the above algorithm, what is meant by a *data race*, and identify the data races that exist in this algorithm.

A data race is a situation where different execution orders, arising from different scheduling decisions, can lead to different results. (2 marks)

There is a data race on k, because all of the processes use the same variable. This means that the program may not check all indexes of f. (2 marks)

There is a data race on $found$, because after process p checks $found$ and line 2, it may be changed by another process before p tests $f(k)$ at line 3. This means that more than one value of k may be printed. (2 marks)

(b) [14 marks] Describe how you would modify the above algorithm to avoid data races, explaining carefully (using invariants where appropriate) how you would ensure that the algorithm behaves correctly without suffering from either data races or deadlock.

- k should be made local to each process (adding synchronisation while leaving k global does not help). Now each counter is only accessed by one process, so there is no possibility of data race or deadlock. (7 marks)

- found should be tested and updated under mutual exclusion; doing this properly requires found to be test inside the loop body. Ensuring mutual exclusion in this guarantees there is no data race, and there will only be one lock/semaphore so there can be no deadlock. (7 marks)

In each case, 3 marks for describing the correction and 4 for explaining why it works and that it is free from data races and deadlock.

(c) [5 marks] Explain what is meant by *fairness*, and how the correctness of your algorithm from part **(b)** depends on a fairness assumption.

An implementation is fair if in every execution, any process that is waiting to execute will eventually get to execute. (2 marks)

In strong fairness this only applies if the process is continuously enabled; in weak fairness if applies if the process is enabled infinitely often. (1 mark)

The given algorithm depends on a fairness assumption, since if there is only one occurrence of 0 and the process that should find it (process m mod N if f(m) = 0) is starved by the scheduler, the occurrence of 0 will never be found. (2 marks)

(d) [5 marks] Briefly describe one way in which your algorithm from part **(b)** could be modified so that it would work correctly in the absence of fairness. Discuss the impact that this approach has on the performance of the algorithm.

The simplest approach is to make k a shared variable, and have each process increment k by one, so that the order in which indexes of f are examined is determined by the order in which processes execute. This will impair the performance if f is cheap to evaluate, since processes will have to wait to increment --- this effect will be lessened if f takes longer to evaluate. (3 marks for approach; 2 marks for discussing impact on performance).

Question 2. Sharing resources. [30 marks]

A *gate control system* at a sport ground is used to monitor the number of people in the sports ground at any time, and to ensure that it does not exceed the maximum allowed (*Max*). The system keeps a count of the total number of people currently in the sports ground, as well as a separate count of the number of people who have passed through each gate, which is used for administrative purposes.

The ground has *N* entrance gates and *M* exit gates. Each time a person goes through an entrance gate, the count for that gate and the total count are both increased by one. If the total count reaches *Max*, no additional people are able to enter until the total count falls below *Max*. Each time a person goes through an exit gate, the count for that gate is increased by one and the total count is decreased by one.

(a) [10 marks] Design a solution to this problem, using either semaphores or monitors. You should ensure that the counters for different gates can be updated concurrently (simulating the fact that, at any point in time, there can be up to *N* people passing through entrance gates and up to *M* people passing through exit gates). You should also ensure that only one process at a time can modify the total count. Your solution should be free from both deadlock and starvation.

This problem can be solved easily using semaphores, as follows. Firstly, use a counting semaphore, initialised to Max, on which an $Enter$ operation does a $wait$, decrementing the count as long as it is positive, and an $Exit$ operation does a $signal$, incrementing the count. The $wait$ operation ensures that no more that Max people may be in the ground at a time.

Alternatively, we can use a binary semaphore to control updates to a shared counter, and use an $await$ statement in the $Enter$ operation to ensure that an $Enter$ can only be completed if there are fewer that Max people in the ground. (5 marks).

Secondly, a separate semaphore should be used to control access to each individual gate counter. If a binary semaphore is used to control updates on the ground counter, the individual gate counters must not be updated within the critical region controlled by that sempahore, since that would prevent different gate counters from being upated concurrently. It is not correct to make the individual gate counters local variables, since then they cannot be accessed for the administrative purposes for which they are intended. (5 marks)

Using monitors, we would need to use a separate monitor for each counter, and the solution would end up being very similar to that using semaphores.

(b) [2 marks] Explain why you chose semaphores, or monitors, to express your solution.

Semaphores (especially a counting semaphore) lead to a simple solution. Naïve use of monitors does not give a satisfactory solution. (Any answer showing some understanding of the merits of the approach chosen gets the marks.)

(c) [10 marks] Give an invariant relating the values of the counters, and use it to argue that your algorithm updates the counts correctly.

Let total be a counting semaphore, initialised to Max, and In[N] and Out[M] be arrays of individual counters for the entrance and exit gates, respectively, all initialised to 0. Then, a suitable invariant is that when no operation is in progress:

0 <= total <= Max and
Max - total = $\sum^{N}_{i=1} In[i] + \sum^{M}_{j=1} Out[j]$

The invariant holds initially, since initially total = Max and In[i] = Out[j] = 0 for all i and j in the appropriate ranges.

The invariant is preserved because:

- *An Enter operation decrements total and increments one element of In, which preserves the second conjunct.*
- *An Enter may only be performed when total > 0, which preserves the first conjunct.*
- *An Exit operation increments total and one element of Out, which preserves the second conjunct.*
- *To show that Exit preserves the first conjunct, we have to assume that people can only pass through an exit gate if they have previously passed through an entrance gate, which means that $\sum^{N}_{i=1} In[i] => \sum^{M}_{j=1} Out[j]$ is also invariant.*

(4 marks for the invariant, 2 for each of the above arguments).

It is possible to give a more detailed invariant that holds during the execution of the Enter and Exit operations, but that is more difficult than is expected of this course.

(d) [4 marks] Give an argument showing that your algorithm is free from deadlock.

No process every holds more than one lock/semaphore at a time, so there is not possibility of two processes waiting on each other.

(e) [4 marks] Give an argument showing that your algorithm is free from starvation.

To ensure freedom from starvation, we have to introduce a fairness assumption. Under this assumption, and process attempting to perform an Enter or Exit operation is guaranteed to be scheduled and thus able to complete its operation.

(We do not need to distinguish between strong and weak fairness here, since a process attempting to perform an Enter or Exit operation remains ready to perform that operation until it succeeds in doing so.)

Question 3. Shared data structures. [30 marks]

Suppose you are required to implement a *concurrent priority queue*, to be shared by an arbitrary number of processes.

From your first year data structures course, you recall two possible ways of implementing a priority queue:

Ordered list: Items are stored in decreasing order of priority (where a small number means high priority), so enqueuing an item requires a search to find the appropriate position for insertion, while a dequeue just removes the first item from the list.

Multiple lists: The priority queue is represented as an array of lists, one for each possible priority value, each of which is treated as an ordinary queue. Thus, an enqueue adds an element to the end of the list for the given priority, while a dequeue finds the first non-empty list (i.e. the one with highest priority), if there is one, and removes the element at the front of that list.

(a) [6 marks] Either of the these implementations can be adapted to allow the priority queue to be shared by multiple processes by enclosing it in a *monitor*. Explain briefly how using a monitor ensures correct concurrent behaviour, and describe the disadvantages of this approach.

A monitor ensures that operations on a shared data structure are executed atomically, so there is no possibility of interference between operations being executed by different processes. The disadvantage of this approach is that there is no possibility of improving performance by having operations that do not interfere with each other executed concurrently - in the case of a priority queue, using either of the implementations described, there is a reasonable prospect that different operations will work on different parts of the data structure and therefore be able to safely execute concurrently. (3 for how it works, 3 for the disadvantage)

(b) [12 marks] Explain how you could adapt the ordered list implementation to allow the priority queue to be shared by multiple processes, while providing better concurrent behaviour that using a monitor.

With this implementation, every dequeue operation will have to access the head of the list, but enqueue operations may perform insertions at different places in the list and thus be able to execute concurrently. Thus, we implement a concurrent priority queue by keeping a lock for the head of the list, which is taken by any process wanting to perform a dequeue. We can then implement the enqueue operation based on the insertion operation for any of the fine-grained locking schemes for linked lists studied in class. For example, we can use the hand-over-hand locking technique, where we traverse the list always keeping a lock on the current node and the previous node, until we find the required insertion point. (6 for description of enqueue, 6 for dequeue).

(c) [12 marks] Explain how you could adapt the multiple lists implementation to allow the priority queue to be shared by multiple processes, while providing better concurrent behaviour that using a monitor.

NOTE: In parts **(b)** and **(c)**, you are only required to explain how concurrency is controlled, **not** to give detailed algorithms. You should explain how your solution ensures that the priority queue is updated correctly and why it is free from deadlock. You are not required

to consider starvation.

This implementation can be made concurrent very simply by just associating a single lock with each list in the array, so a process will take the lock before performing an operation that list. This will allow operations associated with different priority values to execute concurrently. We can do better still by noting that each of the lists is itself acting as a queue, and using Michael and Scott's two-lock implementation on each of the lists. (8 for describing the implementation, 4 for explaining how it gives better concurrency)

Question 4. Channels and Data Spaces. [30 marks]

(a) [5 marks] Explain the operation of the following pseudo-code (note that we assume two channels ch1 and ch2 have already been defined as well as a variable x):

```
either
  ch1 ⇒ x
or
  ch2 ⇒ x
```

Only one alternative can succeed. If communication can take both on multiple channels then one is chosen non-deterministically and succeeds (either ch1 or ch2). (3 marks)

Otherwise it is the alternative branch with the communication statement that could succeed. (1 mark)

If neither can succeed the process blocks. (1 mark)

(b) [5 marks] What are benefits and drawbacks of using a *process array* using synchronous channels to implement matrix multiplication?

Possible to do a direct mapping of the problem of matrix multiplication to processes, very fast as easily implemented in hardware. However, if change the size of matrices you need to change the size of the process array making it a highly inflexible approach. Also implementing in software requires considerable effort in configuring the channels.

(c) [5 marks] Briefly outline what happens when a *Rendezvous* between a client and server takes place.

Synchronise on the entry point. Client blocks until entry point can accept call. Client only returns when server returns a value. If multiple entry points can be called, one is chosen non deterministically. (5 marks).

(d) [5 marks] Contrast the style of coupling between processes using synchronous channels and processes using a Linda model of communication.

Channels are tightly coupled in time and space. Sender and receiver must exist at same time and identity of channel must be available to both. With Linda, the processes are decoupled. Sender and receiver can be present at different times, do not need to agree upon identities. (5 marks).

(e) [5 marks] Write a Linda program that implements the equivalent of a one-way, first-in first-out, synchronous channel shared by two processes.

The program should have the following features, do not be too strict on notation. Define a tag to represent the channel. Process at one end maintains a counter. Posts with a counter. Blocks on a semaphore. Process at other end reads using a counter to ensure reads in right order. When it reads it posts the required semaphore to unblock the source process. (5 marks).

(e) [5 marks] Why is a master-worker architecture well suited to distributing work over multiple home computers connected via the Internet?

Loosely coupled. Only need to know master. Can easily adapt if new machines become available.

Just pull down work, no need to preallocate work. Can easily adapt to different speeds and faster machines will simply pull down more work. (5 marks, must relate the characteristics of the Internet to the characteristics of the architecture).

Question 5. Mutual Exclusion and Consensus. [30 marks]

(a) [5 marks] Why is it generally believed to be impossible to implement a perfectly synchronised global clock in a distributed system? Explain how the permission-based Ricart- Agrawala algorithm (Algorithm 10.2 in the Appendix) for mutual exclusion could be simplified if such a clock was available.

The unpredictable communication delay makes it impossible to synchronise the clocks. Synchronisation could be achieved by tagging each request with a timestamp. The request with the earliest timestamp wins. (5 marks).

(b) [5 marks] Consider the full permission-based Ricart-Algrawala algorithm (Algorithm 10.2 in the Appendix). Explain what happens in the following scenarios:

(i) Node A and Node B choose the same ticket number.

Make the algorithm asymmetric by using NodeID (each node is assumed to have a unique ID) as a tie break. This ensures that cannot be tied on the same ticket number. Handled in line p3. Blocks at p6. (2 marks).

(ii) Node A chooses a lower ticket number than Node B and Node A requests entry to the critical section.

The node holding the lower ticket number does not send a reply message. Because the requesting node requires replies from all other nodes it is blocked until this node (and the others) reply. Handled in line p3 as well. Blocks at p6. (3 marks).

(c) [5 marks] Why can a node receive multiple requests for entry without a matching reply in the token-based version of the Ricart-Agrawala algorithm (Algorithm 10.3 in the Appendix) but not in the permission-based version?

In the token-based algorithm, reply only sent to the node that successfully entered. In the permission-based, request sent to everyone and everyone must send a matching reply. (5 marks).

(d) [5 marks] Consider the one-round algorithm for Consensus (Algorithm 12.1 in the Appendix). Explain under what circumstances the loyal generals may fail to achieve consensus.

Should one general crash and send a vote to one but not the other, a tie may arise that could result in a loyal general deciding differently to another general. (5 marks).

(e) [5 marks] Consider the Byzantine Generals algorithm (Algorithm 12.2 in the Appendix). Assume a system with four nodes: Basil, John, Zoe and Leo. Assume that Basil is the traitor, that John and Zoe vote attack and Leo votes retreat. Draw the full knowledge tree about the loyal general Leo.

The knowledge tree must from about Leo. This means the root is the truth about Leo, the next level is what Leo has told Basil, John and Zoe about himself. The next level is what Basil, John and Zoe have told each other about Leo. As Basil is a traitor we assume that he can say anything. (5 marks).

(f) [5 marks] Calculate the total number of messages sent by each general in the Byzantine Generals algorithm (Algorithm 12.2 in the Appendix) when there are *n* generals.

The total number of messages sent by each general is (n-1)+(n-1).(n-2)+(n-2).(n-3)+ ... because it sends it plan to every other general (n-1), a second-round report of each of the (n-1) generals to (n-2) generals, and so on. (5 marks).

Question 6. Global Properties. [30 marks]

(a) [5 marks] Consider a variant of the Dijkstra-Scholten algorithm (Algorithms 11.2 and 11.3 in the Appendix) where nodes simply inform the environment node when they wish to terminate, and the environment node declares system termination when all nodes have reported their willingness to terminate. Explain why this version of the algorithm is not correct.

A node might declare termination but in the meantime a message might be in transit to it. This would wake it up again. The node that sent the message may have declared termination after sending the message. More problematically, just because a node declares termination does not mean that other active nodes may not activate it. (5 marks).

(b) [5 marks] The Dijkstra-Scholten algorithm (Algorithms 11.2 and 11.3 in the Appendix) maintains a spanning tree containing all the active nodes in the system. With reference to the algorithms, explain how the spanning tree is maintained.

Only nodes with parents can be active and send messages - see p1. The first message from anyone defines who the parent is, p5-6. After informing the parent of its termination, the node disconnects itself from the tree. (5 marks).

(c) [5 marks] Consider a scenario with three nodes: A (environment node), B and C. Over a period of time, A sends three messages to B and B sends four messages to C. Assume that C terminates, followed by B. How many signals are sent using the Dijkstra-Scholten algorithm (Algorithms 11.2 and 11.3 in the Appendix) and how many signals would be sent if signal bundling was used?

Normal system, there are fourteen messages sent. Bundling: seven application messages, C sends three signals to B and one final one. B sends two signals to A and one final one.

(d) [5 marks] Consider the Chandy-Lamport algorithm (Algorithm 11.6 in the Appendix). Explain why a node may receive more than one marker message and what happens when it receives more than one marker.

If a node has multiple incoming edges it will receive multiple markers, one along each edge as each node floods its outgoing edges with markers when it receives one itself. The difference between the last message received before the node recorded is state (in messageAtRecord and the last message received before the marker was received on this edge (in messageAtMarker) is recorded as the messages on the edge that were sent but not received when the snapshot took place.

The edge will be considered empty, as all messages received before the marker are considered to be part of the state of the receiving node.

(e) [5 marks] Explain why we make the FIFO assumption in the Chandy-Lamport algorithm (Algorithm 11.6 in the Appendix) and explain how you could modify the marker message to allow us to remove this assumption.

The marker bounds the messages that are considered in and out of the sending node's state. If the marker could be reordered arbitrarily there would be no way for the receiving node to know what was in and what was not. This could be rectified if the marker carried with it a counter indicating where the boundary lay.

(f) [5 marks] Consider a system with three nodes where all nodes are one hop away from each other node. One node (playing the role of the environment node) initiates the distributed snapshot. How many marker messages are sent in total within the system? Now generalise this result for *n* nodes given the same assumption about connectivity.

Node 1 sends two marker messages. Node 2 receives marker and sends two. Node 3 receives marker and sends two. Six in total. Each node sends a marker to every other node in a mesh system. If there are n nodes, each one sends n-1 marker messages. This makes n . (n-1).

****

Appendix.

Algorithm 10.2: Ricart-Agrawala algorithm	
	integer myNum ← 0
	set of node IDs deferred ← empty set
	integer highestNum ← 0
	boolean requestCS ← false
	Main
	loop forever
p1:	non-critical section
p2:	requestCS ← true
p3:	myNum ← highestNum + 1
p4:	for all other nodes N
p5:	send(request, N, myID, myNum)
p6:	await reply's from all other nodes
p7:	critical section
p8:	requestCS ← false
p9:	for all nodes N in deferred
p10:	remove N from deferred
p11:	send(reply, N, myID)

Algorithm 10.2: Ricart-Agrawala algorithm (continued)	
	Receive
	integer source, requestedNum
	loop forever
p1:	receive(request, source, requestedNum)
p2:	highestNum ← max(highestNum, requestedNum)
p3:	if not requestCS or requestedNum << myNum
p4:	send(reply, source, myID)
p5:	else add source to deferred

Algorithm 10.3: Ricart-Agrawala token-passing algorithm

```
        boolean haveToken ← true in node 0, false in others
        integer array[NODES] requested ← [0,...,0]
        integer array[NODES] granted ← [0,...,0]
        integer myNum ← 0
        boolean inCS ← false

sendToken
    if exists N such that requested[N] > granted[N]
        for some such N
            send(token, N, granted)
            haveToken ← false
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

```
    Main
    loop forever
p1:     non-critical section
p2:     if not haveToken
p3:         myNum ← myNum + 1
p4:         for all other nodes N
p5:             send(request, N, myID, myNum)
p6:         receive(token, granted)
p7:         haveToken ← true
p8:     inCS ← true
p9:     critical section
p10:    granted[myID] ← myNum
p11:    inCS ← false
p12:    sendToken
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

```
        Receive
        integer source, reqNum
        loop forever
p13:        receive(request, source, reqNum)
p14:        requested[source] ← max(requested[source], reqNum)
p15:        if haveToken and not inCS
p16:            sendToken
```

Algorithm 11.2: Dijkstra-Scholten algorithm (env., preliminary)

```
                integer outDeficit ← 0
        computation
p1:     for all outgoing edges E
p2:         send(message, E, myID)
p3:         increment outDeficit
p4:     await outDeficit = 0
p5:     announce system termination
        receive signal
p6:     receive(signal, source)
p7:     decrement outDeficit
```

Algorithm 11.3: Dijkstra-Scholten algorithm

```
                integer array[incoming] inDeficit ← [0,...,0]
                integer inDeficit ← 0
                integer outDeficit ← 0
                integer parent ← −1
    send message
p1: when parent = −1          // Only active nodes send messages
p2:     send(message, destination, myID)
p3:     increment outDeficit
    receive message
p4: receive(message,source)
p5: if parent = −1
p6:     parent ← source
p7: increment inDeficit[source] and inDeficit
```

Algorithm 11.3: Dijkstra-Scholten algorithm (continued)

```
    send signal
p8: when inDeficit > 1
p9:     E ← some edge E for which
            (inDeficit[E] > 1) or (inDeficit[E] = 1 and E = parent)
p10:    send(signal, E, myID)
p11:    decrement inDeficit[E] and inDeficit
p12: or when inDeficit = 1 and isTerminated and outDeficit = 0
p13:    send(signal, parent, myID)
p14:    inDeficit[parent] ← 0
p15:    inDeficit ← 0
p16:    parent ← −1
    receive signal
p17: receive(signal, _)
p18: decrement outDeficit
```

Algorithm 11.4: Credit-recovery algorithm (environment node)
float weight ← 1.0
computation
p1: for all outgoing edges E
p2: weight ← weight / 2.0
p3: send(message, E, weight)
p4: await weight = 1.0
p5: announce system termination
receive signal
p6: receive(signal, w)
p7: weight ← weight + w

Algorithm 11.5: Credit-recovery algorithm (non-environment node)
constant integer parent ← 0 // Environment node boolean active ← false float weight ← 0.0
send message
p1: if active // Only active nodes send messages
p2: weight ← weight / 2.0
p3: send(message, destination, myID, weight)
receive message
p4: receive(message, source, w)
p5: active ← true
p6: weight ← weight + w
send signal
p7: when terminated
p8: send(signal, parent, weight)
p9: weight ← 0.0
p10: active ← false

Algorithm 11.6: Chandy-Lamport algorithm for global snapshots
integer array[outgoing] lastSent ← [0, . . . , 0] integer array[incoming] lastReceived ← [0, . . . , 0] integer array[outgoing] stateAtRecord ← [−1, . . . , −1] integer array[incoming] messageAtRecord ← [−1, . . . , −1] integer array[incoming] messageAtMarker ← [−1, . . . , −1]
send message
p1: send(message, destination, myID) p2: lastSent[destination] ← message
receive message
p3: receive(message,source) p4: lastReceived[source] ← message

Algorithm 11.6: Chandy-Lamport algorithm for global snapshots (continued)
receive marker
p6: receive(marker, source) p7: messageAtMarker[source] ← lastReceived[source] p8: if stateAtRecord = [−1,. . . ,−1] // Not yet recorded p9: stateAtRecord ← lastSent p10: messageAtRecord ← lastReceived p11: for all outgoing edges E p12: send(marker, E, myID)
record state
p13: await markers received on all incoming edges p14: recordState

Algorithm 12.1: Consensus - one-round algorithm

```
                    planType finalPlan
                    planType array[generals] plan
p1:  plan[myID] ← chooseAttackOrRetreat
p2:  for all other generals G
p3:      send(G, myID, plan[myID])
p4:  for all other generals G
p5:      receive(G, plan[G])
p6:  finalPlan ← majority(plan)
```

Algorithm 12.2: Consensus - Byzantine Generals algorithm

```
                    planType finalPlan
                    planType array[generals] plan, majorityPlan
                    planType array[generals, generals] reportedPlan
p1:  plan[myID] ← chooseAttackOrRetreat
p2:  for all other generals G                    // First round
p3:      send(G, myID, plan[myID])
p4:  for all other generals G
p5:      receive(G, plan[G])
p6:  for all other generals G                    // Second round
p7:      for all other  generals G' except G
p8:          send(G', myID, G, plan[G])
p9:  for all other generals G
p10:     for all other generals G' except G
p11:         receive(G, G', reportedPlan[G, G'])
p12: for all other generals G                    // First vote
p13:     majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])
p14: majorityPlan[myID] ← plan[myID]             // Second vote
p15: finalPlan ← majority(majorityPlan)
```

Algorithm 10.2: Ricart-Agrawala algorithm

```
                integer myNum ← 0
                set of node IDs deferred ← empty set
                integer highestNum ← 0
                boolean requestCS ← false
    Main
    loop forever
p1:     non-critical section
p2:     requestCS ← true
p3:     myNum ← highestNum + 1
p4:     for all other nodes N
p5:         send(request, N, myID, myNum)
p6:     await reply's from all other nodes
p7:     critical section
p8:     requestCS ← false
p9:     for all nodes N in deferred
p10:        remove N from deferred
p11:        send(reply, N, myID)
```

Algorithm 10.2: Ricart-Agrawala algorithm (continued)

```
    Receive
    integer source, requestedNum
    loop forever
p1:     receive(request, source, requestedNum)
p2:     highestNum ← max(highestNum, requestedNum)
p3:     if not requestCS or requestedNum << myNum
p4:         send(reply, source, myID)
p5:     else add source to deferred
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm

```
        boolean haveToken ← true in node 0, false in others
        integer array[NODES] requested ← [0,...,0]
        integer array[NODES] granted ← [0,...,0]
        integer myNum ← 0
        boolean inCS ← false
sendToken
    if exists N such that requested[N] > granted[N]
        for some such N
            send(token, N, granted)
            haveToken ← false
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

```
    Main
    loop forever
p1:     non-critical section
p2:     if not haveToken
p3:         myNum ← myNum + 1
p4:         for all other nodes N
p5:             send(request, N, myID, myNum)
p6:         receive(token, granted)
p7:         haveToken ← true
p8:     inCS ← true
p9:     critical section
p10:    granted[myID] ← myNum
p11:    inCS ← false
p12:    sendToken
```

Algorithm 10.3: Ricart-Agrawala token-passing algorithm (continued)

Receive

```
      integer source, reqNum
      loop forever
p13:    receive(request, source, reqNum)
p14:    requested[source] ← max(requested[source], reqNum)
p15:    if haveToken and not inCS
p16:      sendToken
```

Algorithm 11.2: Dijkstra-Scholten algorithm (env., preliminary)

integer outDeficit ← 0

computation

```
p1: for all outgoing edges E
p2:   send(message, E, myID)
p3:   increment outDeficit
p4: await outDeficit = 0
p5: announce system termination
```

receive signal

```
p6: receive(signal, source)
p7: decrement outDeficit
```

Algorithm 11.3: Dijkstra-Scholten algorithm	
	integer array[incoming] inDeficit ← [0,...,0]
	integer inDeficit ← 0
	integer outDeficit ← 0
	integer parent ← −1
	send message
p1:	when parent = −1 // Only active nodes send messages
p2:	send(message, destination, myID)
p3:	increment outDeficit
	receive message
p4:	receive(message,source)
p5:	if parent = −1
p6:	parent ← source
p7:	increment inDeficit[source] and inDeficit

Algorithm 11.3: Dijkstra-Scholten algorithm (continued)	
	send signal
p8:	when inDeficit > 1
p9:	E ← some edge E for which
	(inDeficit[E] > 1) or (inDeficit[E] = 1 and E = parent)
p10:	send(signal, E, myID)
p11:	decrement inDeficit[E] and inDeficit
p12:	or when inDeficit = 1 and isTerminated and outDeficit = 0
p13:	send(signal, parent, myID)
p14:	inDeficit[parent] ← 0
p15:	inDeficit ← 0
p16:	parent ← −1
	receive signal
p17:	receive(signal, _)
p18:	decrement outDeficit

Algorithm 11.4: Credit-recovery algorithm (environment node)
float weight ← 1.0
computation
p1: for all outgoing edges E
p2: weight ← weight / 2.0
p3: send(message, E, weight)
p4: await weight = 1.0
p5: announce system termination
receive signal
p6: receive(signal, w)
p7: weight ← weight + w

Algorithm 11.5: Credit-recovery algorithm (non-environment node)
constant integer parent ← 0 // Environment node boolean active ← false float weight ← 0.0
send message
p1: if active // Only active nodes send messages
p2: weight ← weight / 2.0
p3: send(message, destination, myID, weight)
receive message
p4: receive(message, source, w)
p5: active ← true
p6: weight ← weight + w
send signal
p7: when terminated
p8: send(signal, parent, weight)
p9: weight ← 0.0
p10: active ← false

Algorithm 11.6: Chandy-Lamport algorithm for global snapshots
integer array[outgoing] lastSent ← [0, . . . , 0] integer array[incoming] lastReceived ← [0, . . . , 0] integer array[outgoing] stateAtRecord ← [−1, . . . , −1] integer array[incoming] messageAtRecord ← [−1, . . . , −1] integer array[incoming] messageAtMarker ← [−1, . . . , −1]
send message

```
p1:  send(message, destination, myID)
p2:  lastSent[destination] ← message
```

```
     receive message
```

```
p3:  receive(message,source)
p4:  lastReceived[source] ← message
```

Algorithm 11.6: Chandy-Lamport algorithm for global snapshots (continued)

```
     receive marker
```

```
p6:  receive(marker, source)
p7:  messageAtMarker[source] ← lastReceived[source]
p8:  if stateAtRecord = [−1,. . .,−1]      // Not yet recorded
p9:      stateAtRecord ← lastSent
p10:     messageAtRecord ← lastReceived
p11:     for all outgoing edges E
p12:         send(marker, E, myID)
```

```
     record state
```

```
p13: await markers received on all incoming edges
p14: recordState
```

Algorithm 12.1: Consensus - one-round algorithm

```
                    planType finalPlan
                    planType array[generals] plan
p1:  plan[myID] ← chooseAttackOrRetreat
p2:  for all other generals G
p3:      send(G, myID, plan[myID])
p4:  for all other generals G
p5:      receive(G, plan[G])
p6:  finalPlan ← majority(plan)
```

Algorithm 12.2: Consensus - Byzantine Generals algorithm

```
                    planType finalPlan
                    planType array[generals] plan, majorityPlan
                    planType array[generals, generals] reportedPlan
p1:  plan[myID] ← chooseAttackOrRetreat
p2:  for all other generals G                    // First round
p3:      send(G, myID, plan[myID])
p4:  for all other generals G
p5:      receive(G, plan[G])
p6:  for all other generals G                    // Second round
p7:      for all other  generals G' except G
p8:          send(G', myID, G, plan[G])
p9:  for all other generals G
p10:     for all other generals G' except G
p11:         receive(G, G', reportedPlan[G, G'])
p12: for all other generals G                    // First vote
p13:     majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])
p14: majorityPlan[myID] ← plan[myID]             // Second vote
p15: finalPlan ← majority(majorityPlan)
```