Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Saturday, 26 October 2019

Exam supervisor: K. V. S. Prasad (prasad@chalmers.se, 0736 30 28 22)

(Exam set by K. V. S. Prasad, Sandro Stucki and Nir Pieterman,
based on the course given Sep-Oct 2019)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will not receive any points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; six questions, numbered Q1 through Q6. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (8p). Below is the pseudo-code of a program with two threads, `p` and `q`. The variables `n` and `flag` are shared between `p` and `q`.

int n = 0; **boolean** flag = false;	
p	q

p_1: **while**(flag==false) {
p_2: n = 1 - n;
}
	q_1: **while**(flag==false) {
	q_2: **if**(n==0) {
	q_3: flag = true
	}
	}

The labels p_1, p_2, q_1, q_2 and q_3 are given only for ease of reference.

(Part a) Construct a scenario for which the program terminates. *(2p)*

(Part b) What are the possible values of `n` when the program terminates? *(2p)*

(Part c) Does the program terminate for all scenarios? *(2p)*

(Part d) Does the program terminate for all fair scenarios? *(2p)*

Answers:

(Part a) *q1, q2, q3, p1, q1.*

(Part b) *n can be 0 or 1. (Only p2 changes it).*

Part(a) shows termination with 0.

Termination with 1: p1, q1, q2, p2, q3, p1, q1.

(Part c) *The program does not terminate for all scenarios. Consider p1, p2, (now n=1), then follow by any number of (q1, p1, p2, p1, p2) so that q1 executes exactly when n=1.*

(Part d) *The previous infinite loop was fair (both p and q run infinitely often).*

Q2 (18p). The pseudo-code below tries to solve the critical section (CS) problem with two threads, `p` and `q`. The keyword **atomic** encloses a pair of assignments that happen in one step, i.e., no instruction by the other thread can occur between those two assignments. Remember that CS must exit after a finite time, but NCS may loop.

The label p_i can mean the command that follows p_i, or the proposition that thread `p` is at p_i, and *the next command* `p` will execute is p_i.

	int `tg`= 1;		
	`p`		`q`
	int `tp`= 0;		**int** `tq`= 0;
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
	do{		**do**{
p_2:	**atomic**{`tp`= `tg`; `tg`= 0};	q_2:	**atomic**{`tq`= `tg`; `tg`= 0};
p_3:	} **while**(`tp`!= 1);	q_3:	} **while**(`tq`!= 1);
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	**atomic**{`tp`= 0; `tg`= 1};	q_5:	**atomic**{`tq`= 0; `tg`= 1};
	}		}

A full state description would be a quintuple, $(p_i, q_j,$ `tp, tg, tq`$)$. For a refresher on logical notation, see Appendix B.

(Part a) Show that each of the three variables `tg, tp, tq` can take only the values 0 and 1. *(2p)*

Answer: True after `p` *and* `q` *are initialised. Only* p_2, p_5, q_2, q_5 *change* `tg, tp, tq` *and they introduce only 0 or 1.*

(Part b) Let `I=tp+tg+tq`. Prove `I=1` is invariant (always true). *(3p)*

Answer: `I=1` *at init. The values only change at* p_2, $p5$, q_2 *and* q_5. *Of these,* p_5 *and* q_5 *each yield* `I=1` *when executed. For* p_2*: Either* `tg`*=1 at* p_2 *and* `tp`*+*`tg`*=1 before and after* p_2*, or* `tg`*=0 at* p_2 *and* `tp`*+*`tg`*=0 before and after* p_2.

Notation: Let `tv` stand for any one of `tp`, `tg`, `tq`.
Then from **(Part a)**, $\neg(\texttt{tv} = 1)$ iff $\texttt{tv} = 0$.
From **(Part b)**, in any state, exactly one of `tp`, `tg`, `tq` is 1.
A state description need therefore only be a triple $(p_i, q_j, \texttt{tv})$, where `tv` is the one out of `tp`, `tg`, `tq` that has value 1.

(Part c) Show that $p_5 \rightarrow (\texttt{tp} = 1)$. *(2p).*

Answer: `p` *can get to* p_5 *only if* $\texttt{tp} = 1$ *at* p_3. *Also, q can't affect* `tp`.

(Part d) Show that $p_2 \rightarrow ((\texttt{tp} = 0) \wedge (\texttt{tg} = 1))$. *(2p).*

Answer: There was an error in this question. It is true that p_2 *implies* $(\texttt{tp} = 0)$*, but not also that* $(\texttt{tg} = 1)$. *Everyone who attempted Q2 was given 2 points for this part.*

(Part e) Show that $\neg(p_5 \wedge q_5)$ is an invariant, i.e., that the program guarantees mutual exclusion. *(3p)*

Answer: At $p_5 \wedge q_5$, we have from **(Part d)**, $\texttt{tp} = \texttt{tq} = 1$, *so* `tp+tg+tq`$\geq 2$, *which breaks the invariant* `I` *of* **Part(b)**.

(Part f) Is there a loop of states from $(p_2, q_2, \texttt{tg})$ that includes neither p_5 nor q_5? I.e., can the system livelock, with `p` repeating p_2 and p_3, while `q` repeats q_2 and q_3? *(3p)*

Answer: No. $(p_2, q_2, \texttt{tg}) \rightarrow (\texttt{tg} = 1 \wedge \texttt{tp} = \texttt{tq} = 0)$, so if $(p_3, q_2, \texttt{tv})$ is the next state, it will have $\texttt{tp} = 1 \wedge \texttt{tg} = \texttt{tq} = 0$, and `p` *will move on to p_5. So no livelock. But an unfair scheduler can let $\texttt{tp} = 1 \wedge \texttt{tg} = \texttt{tq} = 0$ after p_2, and then starve p. Then q will loop, and neither p_5 nor q_5 will ever occur.*

(Part g) Thread `q` can starve thread `p` if a certain command by `p` never executes at a certain time. What command at what time? *(3p)*

Answer: If p_2 is never allowed to run between the completion of q_5 and the completion of q_2.

Q3 (10p). The program from Q2 is repeated below for convenience. Read through Q2, including the clarifications and questions, before trying Q3.

	int tg= 1;		
	p		q
	int tp= 0;		**int** tq= 0;
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
	do{		**do**{
p_2:	**atomic**{tp= tg; tg= 0};	q_2:	**atomic**{tq= tg; tg= 0};
p_3:	} **while**(tp!= 1);	q_3:	} **while**(tq!= 1);
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	**atomic**{tp= 0; tg= 1};	q_5:	**atomic**{tq= 0; tg= 1};
	}		}

At the start, we expected each state to be a quintuple, (tp, tq, tp, tg, tq) where tp and tq can take on 3 values each, and each of tp, tg, tq take on two values each, yielding 72 possible states. But **Q2** tells us each state can be writen (tp, tq, tv), where each of tp, tq and tv can take 3 values, so we need only deal with 27 of those 72 program states.

Notation: We now also abbreviate our state notation by writing p_iq_j to mean $(p_i, q_j, \texttt{tg})$, p'_iq_j to mean $(p_i, q_j, \texttt{tp})$, and $p_iq'_j$ to mean $(p_i, q_j, \texttt{tq})$. **Q(2)b** tells us we won't need $p'_iq'_j$.

Here is a partial state transition table for the program above. You may assume that only 11 states are reachable from the start state p_2q_2.

state = p_iq_j perhaps primed		new state if p moves	new state if q moves
s1	p_2q_2	p'_3q_2 = s8	
s2		p'_3q_3 = s9	
s3	$p_2q'_3$		$p_2q'_5$ = s4
s4	$p_2q'_5$		
s5			$p_3q'_3$ = s6
s6	$p_3q'_3$		
s7	$p_3q'_5$		
s8	p'_3q_2		
s9	p'_3q_3		
s10	p'_5q_2		
s11	p'_5q_3		

(Part a) Fill in the blank entries in the table. *(8p)*

(Part b) Pick an unreachable state out of the 27 the abbreviated notation allows, and say why it is unreachable. *(2p)*

Answer:

state = $p_i q_j$ *perhaps primed*		*new state if p moves*	*new state if q moves*
s1	$p_2 q_2$	$p'_3 q_2 = s8$	$p_2 q'_3 = s3$
s2	$p_2 q_3$	$p'_3 q_3 = s9$	$p_2 q_2 = s1$
s3	$p_2 q'_3$	$p_3 q'_3 = s6$	$p_2 q'_5 = s4$
s4	$p_2 q'_5$	$p_3 q'_5 = s7$	$p_2 q_2 = s1$
s5	$p_3 q_2$	$p_2 q_2 = s1$	$p_3 q'_3 = s6$
s6	$p_3 q'_3$	$p_2 q'_3 = s3$	$p_3 q'_5 = s7$
s7	$p_3 q'_5$	$p_2 q'_5 = s4$	$p_3 q_2 = s5$
s8	$p'_3 q_2$	$p'_5 q_2 = s10$	$p'_3 q_3 = s9$
s9	$p'_3 q_3$	$p'_5 q_3 = s11$	$p'_3 q_2 = s8$
s10	$p'_5 q_2$	$p_2 q_2 = s1$	$p'_5 q_3 = s11$
s11	$p'_5 q_3$	$p_2 q_3 = s2$	$p'_5 q_2 = s10$

Q4 (12p). Consider the following event loop of a "compute server" in Erlang.

```
1 server_event_loop() ->
2   receive
3     {request, Fun, Args, From} ->
4       spawn(
5         fun() ->
6           Ans = apply(Fun, Args),
7           From ! {response, Ans}
8         end)
9   end,
10 server_event_loop().
```

(Part a). Give a sequence of messages sent by the server in response to the following sequence of requests sent from *a single client*.

```
1 Me = self(),
2 server ! {request, fun exp/2, [10, 5], Me},
3 server ! {request, fun lists:append/2, [[1, 2], [3]], Me},
4 server ! {request, fun(X) -> X + X end, [8], Me},
```

where

```
1 exp(_, 0) -> 1;
2 exp(X, 1) -> X;
3 exp(X, Y) -> X * exp(X, Y - 1).
```

(3p)

Answer: Possible Solution:

```
{response,100000}
{response,[1,2,3]}
{response,16}
```

(Part b). Is your answer to Part a the only possible sequence that could have been observed? *(2p)*

Answer: No, the `spawn` *in the server loop introduces non-determinism.*

(Part c). What is the point of spawning new (sub-)processes for each computation? *(2p)*

Answer: Parallelism. Without the `spawn`*, the server would have to compute the answers to requests sequentially, even if there are many processor cores available to parallelize the different computations.*

(Part d). Consider the following client code for sending a sequence of requests to the server and collecting responses

```
%Send the messages to the server.
[server ! {request, fun exp/2, [Num, Exp], self()} || Num <- Nums],

%Collect answers and print them.
Ans = [receive {response, Ans} -> Ans end || _ <- Nums],
io:fwrite("~p ^ ~p = ~p~n", [Nums, Exp, Ans]).
```

Can this code go wrong? Will you get the expected output? *(2p)*

Answer: Responses may arrive in the wrong order.

(Part e). Sketch a fix for this problem (no coding necessary, just suggest a fix in words). *(3p)*

Answer: Use references. Here is an example implementation:

```
server_event_loop2 ->
  receive
    {request, Fun, Args, From, Ref} ->
      spawn(
        fun() ->
          Ans = apply(Fun, Args),
          From ! {response, Ans, Ref}
        end)
  end,
  server_event_loop2().

powers(Nums, Exp) ->
  % We generate a new reference for each number in Nums
  RefsNums = [{make_ref(), Num} || Num <- Nums],
  [server ! {request, fun exp/2, [Num, Exp], self(), Ref} ||
    {Ref, Num} <- RefsNums],
  % Receive messages in the correct order
  Ans = [receive {response, Ans, Ref} -> Ans end ||
    {Ref, _} <- RefsNums],
  io:fwrite("~p ^ ~p = ~p~n", [Nums, Exp, Ans]).
```

Q5 (10p). The following recursive Java method `add()` implements vector addition for two arrays `v1` and `v2`.

```
1  void add(int[] v1, int[] v2, int start, int end) {
2    if (end - start < 1)       return;
3    else if (end - start == 1)
4           { v1[start] = v1[start] + v2[start]; }
5    else {
6      int mid = (start + end) / 2;
7      add(v1, v2, start, mid);
8      add(v1, v2, mid, end);
9    }
10 }
```

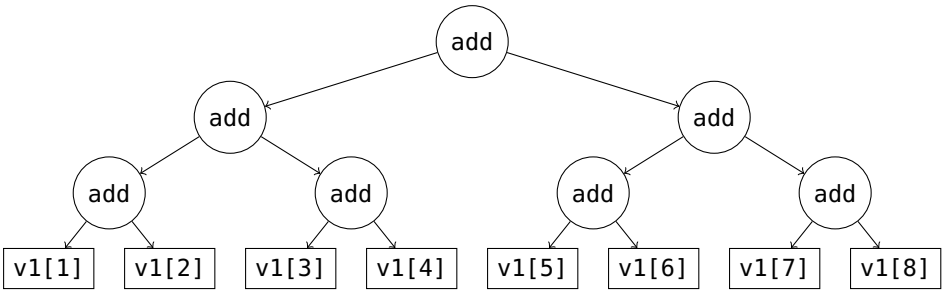
The method adds the entries of `v1` and `v2` from index `start` to index `end` − 1 (excluding `v1[end]` and `v2[end]`), and writes the result back into `v1`.

The class `AddTask` below implements a fork/join parallelization of `add()`. The constructor of `AddTask` has the same arguments as the method `add()`; the `compute()` method computes the result and stores it in the array `v1`. See Appendix A.2 for a full code listing.

```
1  class AddTask extends RecursiveAction {
2    private int[] v1, v2;
3    private int start, end;
4
5    AddTask(int[] v1, int[] v2, int start, int end) {
6    this.v1 = v1; this.v2 = v2;
7    this.start = start; this.end = end;
8    }
9
10   public void compute() {
11     if (end - start < 1) return;
12     else if (end - start == 1) {
13              v1[start] = v1[start] + v2[start];}
14     else {
15       int mid = (start + end) / 2;
16       AddTask t1 = new AddTask(v1, v2, start, mid);
17       AddTask t2 = new AddTask(v1, v2, mid, end);
18       t1.fork(); t2.fork();
19       t1.join(); t2.join();
20     }
21   }
22 }
```

(Part a). Draw the dependency graph for a run of `AddTask(v1, v2, 0, 8)` assuming both `v1` and `v2` have exactly 8 elements. Each node should represent a task instance, with leaf nodes representing the base cases (where `end-start <= 1`). Edges should represent parent-child relationships between tasks. How many nodes does the graph have? *(2p)*

Answer: there are $8 + 4 + 2 + 1 = 15$ *nodes.*



(Part b). What is the approximate runtime of `SumTask(arr, 0, 8)` assuming there are at least 8 CPU cores and each task takes about one unit of time to perform its computation? How can this be inferred from the dependency graph? *(2p)*

Answer: 4 time units – the depth of the dependency graph.

Note. There is an error in this question: `SumTask(arr, 0, 8)` *should be* `AddTask(v1, v2, 0, 8)`*.*

(Part c). What is the maximum number of tasks that can be executed in parallel in this implementation (excluding parent tasks waiting for a child task to finish)? How can this be inferred from the dependency graph? *(2p)*

Answer: 8 calls – the width of the dependency graph.

(Part d). From your dependency graph, deduce the total number of tasks started by `AddTask` when running an instance of `AddTask(v1, v2, 0, 8)`. How does this compare to the maximum number of tasks running in parallel that you computed in **Part(b)**? What does this difference correspond to? *(2p)*

Answer: There are 15 tasks – the same number as there are nodes in the dependency graph – and 7 more than the maximum number of calls that can execute in parallel. This means that there may be up to 7 blocked tasks when the "leaves" are being computed.

Note. There is an error in this question: the result should be compared to ***Part(c)****, not* ***Part(b)****. Correct answers received full points irrespective of whether they compared to* ***Part(b)*** *or* ***(c)****.*

(Part e). Suppose an instance of `AddTask(v1, v2, 0, 1000000)` is run on a machine with a 4-core processor, and that tasks are scheduled using an optimally configured `ForkJoinPool`. Will this result in an efficient parallel computation? Name one source of inefficiency. *(2p)*

Answer: The computation will be reasonably efficient, but not optimal. Since every "recursive call" involves the creation of a new task object even for very small computations (near the "leaves"), spawning overhead will be large. In addition, most tasks are created unnecessarily since the number of tasks ($\simeq$1M tasks) far exceeds the physical capacity for parallel execution (4 cores) and most tasks are just waiting idly for their sub-tasks to complete. Although using fork-join tasks and a thread pool (instead of OS threads) reduces the spawning overhead, it remains significant for small sub-tasks (at the leaves). Here are two ways to reduce spawning overhead:

- *compute small instances of* `AddTask` *(fewer than 1000 elements) using a sequential* `add()` *function;*
- *remove the call to* `t1.fork()` *and replace the call to* `t1.join()` *with a call to* `t1.compute()`*. This reduces the total number of tasks and avoids blocked tasks since parents now work in parallel with their children.*

Q6 (14p). The code on page 9 is a modified sequential singly-linked list implementation. The list stores a sorted set of integers, and the method `addIncrease` adds elements in increasing order. We tried to convert it to a lock-free thread-safe implementation using Compare and Swap.

Besides the code shown on page 9, the interface of AtomicNode contains a getter and setter for val, and a getter for next.

In the `List` class on page 9, notice that the list does not use the value field of the head node (lines 4, 11, and 18). This is intentional. The first element in the list is held in the next field of the head. The method `addIncrease` works as follows. The list is sorted in increasing order. So, if the next element in the list is larger than the value to add, then the value is added after the current element (lines 11–18). If the next element in the list is the same as the value to add, then the search stops (and the value is not added again; lines 18–20). Otherwise, the method continues to scan the list (lines 21–23, and back to line 10). Finally, if there is no next element, the value is added at the end (lines 24–26 just after the while-loop (lines 10–23) terminates).

But there are bugs in this implementation with multiple threads: the list might not be sorted, or the same value may appear multiple times.

(Part a). Explain why the implementation has this issue. *(4p)*

Answer: The compare and swap should resume the search if a new element was added directly after current. This means that there is a new element that is larger than current but it could be also smaller than the value that is begin added or the same.

The same could happen with the two CAS operations: in the middle and in the end of the list.

(Part b). Update the code of `addIncrease` to fix the implementation. You should use a CAS (compare-and-set) synchronization primitive but no semaphores or locks. Use either Java or pseudo code. *(6p)*

```
1 public class AtomicNode {
2   private int val;
3   private AtomicNode next;
4   public boolean compareAndSetNext(AtomicNode existing,
5                                    AtomicNode updated) {
6     ...
7   }
8 }
```

```
1 class List {
2   AtomicNode head;
3   public List() { // value of head not used
4     head = new AtomicNode(0,null);
5   }
6
7   public void addIncrease(int val) {
8     AtomicNode current = this.head;
9     AtomicNode next = current.getNext();
10    while (next != null) {
11      if (next.getVal() > val) {
12        AtomicNode temp;
13        do {next = current.getNext();
14            temp = new AtomicNode(val,next);
15        } while (!current.compareAndSetNext(next,temp));
16        return;
17      }
18      if (next.getVal() == val) {
19        return;
20      }
21      current = next;
22      next = current.getNext();
23    }
24    do {next = current.getNext();
25        temp = new AtomicNode(val,next);
26    } while (!current.compareAndSetNext(next,temp));
27  }
28 }
```

Figure 1: The interface AtomicNode snd the class List.

Answer: The failure of the CAS should lead to continued search. This is the case also for CAS failure at the end.

```
1 public void addIncrease(int val) {
2   AtomicNode current = this.head;
3   while (true) {
4     AtomicNode next = current.getNext();
5     if (next == null || next.getVal() > val) {
6       Node temp = new AtomicNode(val,next);
7       if (current.compareAndSetNext(next,temp)) {
8           return;
9       } else {
10          continue;
11      }
12    }
13    if (next.getVal() == val) {
14      return;
15    }
16    current = next;
17  }
18 }
```

You benchmark the list implementation versus the usage of the original sequential list protected by a lock. You create threads that use the new (thread-safe) list and run the following code:

```
1 for (int j = 0 ; j<100 ; ++j) {
2   l.addIncrease(j);
3 }
```

You create threads that use the old sequential implementation and one additional lock and run the following code:

```
1 for (int j=0 ; j<100 ; ++j) {
2   lock.lock();
3   l.addIncrease(j-1,j);
4   lock.unlock();
5 }
```

(Part c). What happens when you run the thread-safe version vs the sequential version with the lock in the following scenarios: (i) there are 32 CPU cores and 32 threads, and (ii) there is 1 CPU core and 32 threads. *(4p)*

Answer:

(i) With 32 cores, the lock-free version works better than the lock version. The lock version allows only one CPU to work at a time.

So effectively not using the different cores. In the lock-free version all the threads work at the same time. Sometimes interfering with each other but overall, using all the available cores.

(ii) With 1 core, the lock version works better than the lock-free one. In the lock-free version, all the different threads are going to constantly compete for the single CPU obstructing each other and forcing each other to repetitively do the same operations. In the lock version, the locked-out threads just go to sleep and do not interfere with each other.

A Full code listings

A.1 Code for Q4

```
1  -module(q4_appendix).
2  -export([main/0]).
3
4  server_event_loop() ->                    % Server loop
5    receive
6      {request, Fun, Args, From} ->
7        spawn(
8          fun() ->
9            Ans = apply(Fun, Args),
10           From ! {response, Ans}
11         end)
12   end,
13   server_event_loop().
14
15 exp(_, 0) -> 1;
16 exp(X, 1) -> X;
17 exp(X, Y) -> X * exp(X, Y - 1).
18
19 main() ->                           % Start and register the server
20   Pid = spawn(fun server_event_loop/0),
21   register(server, Pid),
22
23   Me = self(),                                    % For Part a:
24   server ! {request, fun exp/2, [10, 5], Me},
25   server ! {request, fun lists:append/2, [[1, 2], [3]], Me},
26   server ! {request, fun(X) -> X + X end, [8], Me},
27
28   % Receive and print the responses.
29   [ receive R -> io:fwrite("~p~n", [R]) end || _ <- [1, 2, 3]],
```

```
30
31   Nums = [7,0,5,2],                                    % For Part d:
32   Exp = 3,
33
34   % Send the messages to the server.
35   [server ! {request, fun exp/2, [Num, Exp], self()} || Num <- Nums],
36
37   % Collect answers and print them.
38   Ans = [receive {response, Ans} -> Ans end || _ <- Nums],
39   io:fwrite("~p ^ ~p = ~p~n", [Nums, Exp, Ans]).
```

A.2 Code for Q5

```
1  import java.util.concurrent.*;
2
3  class AddTask extends RecursiveAction {
4    private int[] v1, v2;
5    private int start, end;
6
7    AddTask(int[] v1, int[] v2, int start, int end) {
8      this.v1 = v1; this.v2 = v2;
9      this.start = start; this.end = end;
10   }
11
12   @Override public void compute() {
13     if (end - start < 1) return;
14     else if (end - start == 1) {
15               v1[start] = v1[start] + v2[start];}
16     else {
17       int mid = (start + end) / 2;
18       AddTask t1 = new AddTask(v1, v2, start, mid);
19       AddTask t2 = new AddTask(v1, v2, mid, end);
20       t1.fork(); t2.fork();
21       t1.join(); t2.join();
22     }
23   }
24 }
25
26   public static void main(String[] args) {
27     final int N = 20;
28     int[] numbers = new int[N];
29     for (int i = 0; i < N; ++i) { numbers[i] = i; }
30
31     AddTask m = new AddTask(numbers, numbers, 0, N);
```

```
32      ForkJoinPool.commonPool().invoke(m);
33
34      for (int i = 0; i < N; ++i) {
35        System.out.println("numbers[" + i + "] = " + numbers[i]);
36      }
37    }
38  }
```

B Linear Temporal Logic (LTL) notation

1. An atomic proposition such as q_2 (process q is at label q_2) *holds for* a state s if and only if process q is at q_2 in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\Box$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$