Chalmers | GÖTEBORGS UNIVERSITET
Computer Science and Engineering

Principles of Concurrent Programming TDA384/DIT391

Saturday, 28 October 2017, 08:30–12:30

(including example solutions)

Teacher/examiner: K. V. S. Prasad (`prasad@chalmers.se`)

Exercise 1: Mutual exclusion *(19 points)*

Here is yet another algorithm to solve the critical section problem, built from atomic `await` commands (②, ⑤) that await either of two conditions, and atomic **`switch`**/**`case`** commands (①, ④, ③ and ⑥). In the **`switch`**/**`case`** commands, the test on `s`, and the subsequent assignment to it, take place without interruption. Note that the **`default`** fallback case in every **`switch`** does not change the value of any variable. The global variable `s` can take any of 5 possible values: `Z`, `P`, `Q`, `PQ` or `QP`.

```
enum s = Z;
```

thread p

```
while (true) {
      // non-critical section
  ①  switch (s) {
        case Z: s = P; break;
        case Q: s = QP; break;
        default: break;
      }
  ②  await(s == P || s == PQ);
      // critical section
  ③  switch (s) {
        case P: s = Z; break;
        case PQ: s = Q; break;
        default: break;
      }
}
```

thread q

```
while (true) {
    // non-critical section
   switch (s) {                      ④
     case Z: s = Q; break;
     case P: s = PQ; break;
     default: break;
   }
    await(s == Q || s == QP);      ⑤
    // critical section
  switch (s) {                       ⑥
    case Q: s = Z; break;
    case QP: s = P; break;
    default: break;
  }
}
```

A state transition table is a tabular version of a state/transition diagram. Below is part of the state transition table for this program, where we only keep track of the program locations marked by a circled number in the code (① through ⑥). In the table, the leftmost column lists the *current* state (where threads p and q are, and the value of `s`). The middle column gives the next state if p executes a step, and the last column gives the next state if q executes a step. In many states p or q

are free to execute the next step, and either may do so. But in some states, one or both threads may be blocked. There are 9 states in all. The notation $p \triangleright k$ means that thread p is at line $\textcircled{k}$ in the program (and similarly for thread q)

	current state ($p \triangleright j$, $q \triangleright k$, `s`)	next state if p moves	next state if q moves
1.	($p \triangleright 1$, $q \triangleright 4$, Z)	($p \triangleright 2$, $q \triangleright 4$, P)	($p \triangleright 1$, $q \triangleright 5$, Q)
2.	($p \triangleright 1$, $q \triangleright 5$, Q)	($p \triangleright 2$, $q \triangleright 5$, QP)	($p \triangleright 1$, $q \triangleright 6$, Q)
3.			
4.			
5.			
6.			
7.			
8.	($p \triangleright 3$, $q \triangleright 4$, P)	($p \triangleright 1$, $q \triangleright 4$, Z)	($p \triangleright 3$, $q \triangleright 5$, PQ)
9.			

Question 1.1 (6 points): Complete the state transition table (we have left 6 lines blank).

	current state ($p \triangleright j$, $q \triangleright k$, `s`)	next state if p moves	next state if q moves
1.	($p \triangleright 1$, $q \triangleright 4$, Z)	($p \triangleright 2$, $q \triangleright 4$, P)	($p \triangleright 1$, $q \triangleright 5$, Q)
2.	($p \triangleright 1$, $q \triangleright 5$, Q)	($p \triangleright 2$, $q \triangleright 5$, QP)	($p \triangleright 1$, $q \triangleright 6$, Q)
3.	($p \triangleright 1$, $q \triangleright 6$, Q)	($p \triangleright 2$, $q \triangleright 6$, QP)	($p \triangleright 1$, $q \triangleright 4$, Z)
4.	($p \triangleright 2$, $q \triangleright 4$, P)	($p \triangleright 3$, $q \triangleright 4$, P)	($p \triangleright 2$, $q \triangleright 5$, PQ)
5.	($p \triangleright 2$, $q \triangleright 5$, PQ)	($p \triangleright 3$, $q \triangleright 5$, PQ)	—
6.	($p \triangleright 2$, $q \triangleright 5$, QP)	—	($p \triangleright 2$, $q \triangleright 6$, QP)
7.	($p \triangleright 2$, $q \triangleright 6$, QP)	—	($p \triangleright 2$, $q \triangleright 4$, P)
8.	($p \triangleright 3$, $q \triangleright 4$, P)	($p \triangleright 1$, $q \triangleright 4$, Z)	($p \triangleright 3$, $q \triangleright 5$, PQ)
9.	($p \triangleright 3$, $q \triangleright 5$, PQ)	($p \triangleright 1$, $q \triangleright 5$, Q)	—

Question 1.2 (3 points): Using the notation $p \triangleright k$ to mean that thread p is at line $\textcircled{k}$ in the program (and similarly for thread q), express the following properties of the program using temporal logic:

a) The program guarantees mutual exclusion between p and q.
$\Box\neg(p \triangleright 3 \wedge q \triangleright 6)$

b) The program guarantees freedom from deadlock.
$\Box\,(p \triangleright 2 \vee q \triangleright 5 \;\rightarrow\; \Diamond(p \triangleright 3 \vee q \triangleright 6))$

c) The program guarantees freedom from starvation for p and for q.
$\Box(p \triangleright 2 \rightarrow \Diamond p \triangleright 3) \wedge \Box(q \triangleright 5 \rightarrow \Diamond q \triangleright 6)$

Question 1.3 (3 points): By analyzing the transition table, explain how it proves that the program ensures mutual exclusion.

There is no state with ($p \triangleright 3$, $q \triangleright 6$, `s`).

Question 1.4 (3 points): By analyzing the transition table, explain how it proves that the program does not deadlock (there are `await` statements, so it is possible for a thread to block).

Every state has at least one possible move (outgoing transition).

Question 1.5 (4 points): By analyzing the transition table, explain how it proves that even if q terminates in state $q \triangleright 4$, the program ensures the liveness of p (i.e., p will progress).

The only states where p is unable to make progress are those at lines 6 and 7 of the transition table. At line 6, q's current state is $q \triangleright 5$: q is waiting to enter the critical section, and it can proceed given that `s == QP`. At line 7, q's current state is $q \triangleright 6$: q is in its critical section, and we assume that it must eventually exit it. Thus, in both cases the system makes progress, eventually allowing p to proceed.

Exercise 2: Semaphores *(10 points)*

Consider $n + 1$ threads that execute in parallel; the threads share an integer variable `x`, whose value is initially 0, and synchronize by means of a shared semaphore instance `sem` initialized to 2 (the semaphore's *capacity*). The $n + 1$ threads are split in two groups: $n > 0$ threads $t_1, \ldots, t_n$ execute the code on the left in the following table, which increments `x`; the other thread u executes the code on the right, which decrements `x`. Each numbered line corresponds to one atomic statement; each thread executes its code once (there is no implicit loop).

```
int x = 0;
Semaphore sem = 2;
```

	thread t_k	thread u	
1	`sem.down();`	`sem.down();`	5
2	`int y = x;`	`int y = x;`	6
3	`x = y + 1;`	`x = y - 1;`	7
4	`sem.up();`	`sem.up();`	8

Question 2.1 (3 points): Describe an interleaving of the threads such that `x == -1` when all threads terminate.

Thread u acquires one permission of the semaphore, and writes `x == 0` to its local `y`. Then all t_k threads proceed sequentially, using the other semaphore permission, and increment `x` to n. Finally, u terminates execution by writing `-1` to `x` and releasing its permission.

Question 2.2 (5 points): For every integer a, $-1 \leq a < n$, describe an interleaving of the threads such that `x ==` a when all threads terminate.

$a + 1$ threads t execute sequentially, incrementing `x` to $a + 1$; then u acquires one permission of the semaphore, and writes `x` to its local `y`; then all remaining $n - (a + 1)$ threads t proceed sequentially using the other semaphore permission. Finally, u terminates execution by writing $a + 1 - 1 = a$ to `x` and releasing its permission.

Question 2.3 (2 points): Describe a change to the initialization of the global variables that ensures that `x` is always a nonnegative value when all threads terminate (without deadlocking).

If `sem`'s capacity is set to 1 instead of 2, the threads have to strictly alternate without interruptions. Thus, the final value of `x` will be $n - 1 \geq 0$ in every possible scenario.

Exercise 3: Locks *(12 points)*

Consider the following implementation of a lock using atomic reads and writes.

```java
class FLock implements Lock {
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = this.threadID();   // id of running thread
    do {
      do {
        turn = me;
      } while (busy);
      busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

In the following questions, assume multiple threads have access to a shared instance of class `FLock`. Also assume that the attributes of `FLock` can be read and written consistently by multiple threads (that is, as if they were declared `volatile` in Java).

Question 3.1 (6 points): Does the lock implementation of `FLock` guarantee freedom from **starvation** for two concurrent threads executing method `lock()`? If it does, explain how it does so; if it does not, describe a scenario where starvation occurs.

Starvation may occur because deadlock may occur: see solution to the next question.

Question 3.2 (6 points): Does the lock implementation of `FLock` guarantee freedom from **deadlock** for two concurrent threads executing method `lock()`? If it does, explain how it does so; if it does not, describe a scenario where deadlock occurs.

Deadlock may occur. Suppose two threads t and u exit the inner loop reading `busy == false`, and suppose that u wrote `turn` last. If t now proceeds, it sets `busy` to `true`, but has to repeat the outer loop because `turn` is set to u's id; t continues and sets `turn` to its ids, but finds `busy == true`. Now u writes `true` into `busy` again, and finds `turn` equal to t's id, so it also repeats the inner loop. At this point both threads are stuck forever in the inner loop with `busy == true`, trying to acquire the lock. Any other thread executing `lock()` would also get stuck in the inner loop.

Exercise 4: Message passing with Erlang *(15 points)*

In this exercise, you will implement a concurrent Erlang program that sorts *positive integers* sent to it by maintaining a multi-process ordered list.

The basic idea is that there are as many processes as positive integers in the list, plus two separate nodes marking the beginning (head) and end (tail) of the list. Each process runs an instance of a function `chain`, which behaves like a server event loop: it receives messages from other processes and interprets them as commands. We refer to such processes as *nodes*.

A node stores a positive integer number `N` and the PID `Next` of another node. The list of `Next` references is constructed starting from a *head* node, and ending in a *tail* node. If we follow the chain of nodes from the head to the tail we find positive integers stored in increasing order.

Each node (running function `chain`) recognizes two kinds of messages:

- a tuple `{num, X}`, which signals that positive integer `X` should be added to the chain as a new node in the correct position;
- a tuple `{list, From}`, which signals that process `From` requests a list with all the elements from the receiving process to the end of the chain in their sorted order.

We assume that the head stores the number `0`, the tail stores the atom `infinity`, and clients only send messages with positive, finite integers directly to the head node (which then forwards them to other nodes in the list as appropriate). Thus, this is the general structure of `chain`:

```
chain(N, Next) ->
  receive
    {num, X} ->      % Question 4.2
    {list, From} ->  % Question 4.3
  end.
```

Question 4.1 (3 points): Write a function `init()` that spawns two processes for the *head* and *tail* nodes, connects them, and returns the PID of the head. You will store `0` and `infinity` in the head and tail node respectively. Since we assume that *every number* that will be sent to `Head` will be a positive integer, every such number will be greater than `0` and smaller than `infinity` (in Erlang every number is by convention smaller than any atom such as `infinity`).

```
init() ->
                      % the value Tail's Next does not matter
  Tail = spawn(fun() -> chain(infinity, self()) end),
  Head = spawn(fun() -> chain(0, Tail) end),
  Head.
```

Question 4.2 (6 points): Write the code handling a message `{num, X}` in function `chain` outlined above. The message is a request to the receiving node to add positive integer `X` to the chain. Since the chain is kept in order, the receiving node behaves as follows:

- If `X > N`, `X` should be placed to the right of the receiving node. Thus, the receiving node forwards the message to the `Next` node, and keeps the same state.

- Otherwise, X should be placed to the immediate left of the receiving node, since all nodes to the left of the receiving node forwarded it to the receiving node (that's how it arrived here). Thus, the receiving node spawns a new process running a suitable instance of chain, and changes its state so that the whole chain of nodes remains connected and in the correct order.

Note that we cannot change the Next of the node to the left of the receiving node, thus the receiving node must switch its stored value to X < N and connect to the new process storing N.

```
{num, X} ->
  if
    X > N ->
      Next ! {num, X},
      chain(N, Next);
    true ->
      New = spawn(fun() -> chain(N, Next) end),  % the new process stores N
      chain(X, New)        % the process of the receiving node now stores X
  end;
```

Question 4.3 (6 points): Write the code handling a message {list, From} in function chain outlined above. The message is a request to the receiving node to construct a list of numbers in the chain from the receiving node to the tail, and to send it to From. Since the chain is kept in order, the node behaves as follows:

- If N < infinity, the receiving node is *not* the tail. Thus, the receiving node forwards the message to the Next node, waits for its reply, adds N to the list sent in the reply, and sends the resulting list to From.
- Otherwise, the receiving node is the tail node, and there are no further nodes after it. Thus, the receiving node sends a list with the only element N to From.

In both cases, after a request is served the node goes back to its previous state.

```
{list, From} ->
  if
    N < infinity ->
      Next ! {list, self()},
      receive L -> From ! [N | L] end;
    true -> From ! [N]
  end,
  chain(N, Next)
```

Exercise 5: Concurrent data structures *(14 points)*

In this exercise, you will evaluate different implementations of a simple list data structure in Java, analyzing whether they are *thread safe*, that is accessible concurrently by multiple threads without running into race conditions.

Let us first consider a basic implementation of a list of integers as class `FindList`:

```java
class FindList {
  protected ArrayList<Integer> data;

  FindList(ArrayList<Integer> data) {
    this.data = new ArrayList<>(data);
  }

  public int find(Integer el) {
    int k = -1;
    for (Integer e: data) {
      k += 1;
      if (e == el) break;
    }
    return k;
  }
}
```

`FindList` only supports one operation, implemented by method `find`: return the index of element `el`, if it exists in the list; return `-1`, if `el` is not in the list. The data is stored using a standard `java.util.ArrayList` as attribute `data`, which is initialized by `FindList`'s constructor.

Class `DropList` inherits from `FindList`, adding an operation implemented as method `remove`, which removes its argument `el` from the list if it is found, and otherwise it does not do anything. Class `DropList` also includes a lock object as attribute `lock`, but the lock is not used in `DropList`'s implementation.

```java
class DropList extends FindList {
  private Lock lock = new ReentrantLock();
  public void remove(Integer el) {
    int k = find(el);              // find position of el in the list
    if (k >= 0) data.remove(k);    // remove the element at position k
  }
}
```

Question 5.1 (2 points): Suppose there is an instance `fl` of class `FindList`, and multiple threads have access to `fl` – any number of threads, each executing any of the available operations of `FindList`. Is access to `fl` thread safe? If it is, explain why threads cannot interfere; if it is not, outline an example of threads interleaving that determines interference.

The only public operation is `find`, which does not modify the list content. Therefore, all operations are read-only, `fl` is immutable, and interference cannot occur.

Question 5.2 (2 points): Suppose there is an instance `dl` of class `DropList`, and multiple threads have access to `dl` – any number of threads, each executing any of the available operations of `DropList`. Is access to `dl` thread safe? If it is, explain why threads cannot interfere; if it is not, outline an example of threads interleaving that determines interference.

Now interference is possible, because `remove` modifies the list content. For example, threads t and u may be both trying to remove the same element `el`. If they interleave in a way that they both execute `find` first, both threads will then remove the element at the same position `k`, effectively removing two elements instead of just one.

Question 5.3 (3 points): Write an implementation of `remove` that is thread safe, using `lock` to avoid race conditions. Your implementation cannot change the signature of method `remove` (that is, `public void remove(Integer el)`) and can only use `lock` to synchronize threads.

A simple implementation just acquires a lock at the beginning, and releases it at the end:

```
public void remove(Integer el) {
  lock.lock();
  try {
    int k = find(el);              // find position of el in the list
    if (k >= 0) data.remove(k);    // remove the element at position k
  } finally { lock.unlock(); }
}
```

Question 5.4 (7 points): Class `DropList2` is a variant of `DropList` using a version of `remove` that is thread safe without using any locks. Unlike `DropList`, `DropList2` does not have a method `find`: `DropList2`'s only public operation is `remove`. Here is the implementation of `DropList2` with two parts replaced by comments, which you have to fill in. (The constructor is not shown for brevity.)

```
class DropList2 {
  protected /* ATTRIBUTE TYPE */ data;

  public void remove(Integer el) {
    ArrayList<Integer> curData, newData;
    do {
      // get current value of ArrayList data
      curData = data.get();
      // build a local copy of curData
      newData = new ArrayList<Integer>(curData);
      // look for position k of el in newData
      int k = -1;
      for (Integer e: newData) {
        k += 1;
        if (e == el) break;
      }
      // if el was found, remove it from newData
      if (k >= 0) newData.remove(k);
      // update data to refer to newData, if it has not changed
    } while(/* LOOP CONDITION */);
  }
}
```

a) (1 point) Choose a suitable **type** for attribute `data`, so that it can be updated thread-safely without using locks.

```
protected AtomicReference<ArrayList<Integer>> data;
```

b) (2 points) Choose a suitable **loop condition** for the `do ... while` loop of `remove`, so that the loop is exited when `data` is updated successfully.

```
do { /* ... */ } while(!data.compareAndSet(curData, newData));
```

c) (2 points) Explain how the implementation of `remove` makes `DropList2` thread safe without using locks.

If multiple threads are executing `remove`, they can proceed independently until they execute `compareAndSet`. Then, only the first thread will successfully execute `compareAndSet` thus updating `data`; the other threads will find `data` changed and thus repeat the loop trying again from scratch.

d) (2 points) What is a wait-free algorithm? Is your implementation of `remove` wait free?

A wait-free algorithm is one where every thread eventually makes progress in getting access to the shared data. The implementation of `swap` is lock-free but not wait-free, since one thread may loop forever trying to update `data` while other threads change it successfully.