MODIS Hand-In 4 Mandatory Exercises

Dennis Thinh Tan Nguyen, Jakob Holm, Jacob Mullit Mniche,
Pernille Lous, Thor Valentin Aakjr Olesen, William Diedrichsen Marstrand

30. november 2015

Indhold

1 Assignment 15.1

An unreliable failure detector may produce one or two values when given the identity of a process, which can either be Unsuspected or Suspected.

- Unsuspected The detector has recently received evidence suggesting that the process has not failed. By way of example, a message may recently have been received from it. The process may have failed since.
- Suspected The detector may have some indication that the process has failed. By way of example, a process may not have received any message for a prolonged timeframe that exceeds a nominal maximum length of silence. The reason may be that the network has been slow while the process is still functioning.

Both of these may or may not accurately reflect whether the process has actually failed

2 Assignment 15.3

One can express the relation between maximum throughput and syncronization delay in a mutual exclusion system with the following formula below. It expresses the maximum throughput of critical-section-entries per second.

$$Assume : \tag{1}$$

$$s = synchronization\ delay \tag{2}$$

$$m = writing\ time \tag{3}$$

$$t_m = maximum\ througput \tag{4}$$

$$\tag{5}$$

$$t_m = 1 \div (s + m) \tag{6}$$

3 Assignment 15.4

- Process A sends a request rA for entry then sends a message m to B
- On receipt of m, B sends request rB for entry. To satisfy happened-before order, rA should be granted before rB .
- Due to the changes of message transmission delay, rB arrives at the server before rA , and they are serviced in the opposite order.

4 Assignment 15.6

A scenario where process X wait indefinitely if new processes XY keep occurring or using the requested resource repeatedly. By way of example, a simple scheduling algorithm could cause starvation in a multi-tasking system. The system might always switch between the first two tasks while a third never gets to run, consequently starving the third task of CPU time.

Another example: A server that uses optimistic concurrency control but doesnt verify that a client has its transaction aborted repeatedly will lead to starvation of the client.

5 Assignment 15.22

We assume correct processes can reach different conclusions due to a use of different algorithms. This implies we cannot reach a conclusion in one round of messages due to the fact that there may be multiple correct answers provided by loyal lieutenants. This could be solved by letting each processor send three signed messages. One for the commander, one for a lieutenant and thereafter forward the message the lieutenant has received. This enables the commander to find disloyal lieutenant and thus cast a faithful vote. The solution assumes the commander cannot be disloyal, which is the proposed integrity solution.

6 Assignment 15.23

Assume the system has 3 generals. One of these is treacherous. Also, assume a signature cannot be forged and everyone can validate the authenticity of a signature.

- All generals send their conclusion to all other lieutenants.
- Each lieutenant forwards the received orders to the other lieutenants.

It is now possible to determine which lieutenant was faulty based on the signature since, by the assumption, one is not able to forge signatures when forwarding messages.

7 Assignment 16.2

Give three serially equivalent interleaveings of Transaction T and U:

Transaction T	Transaction U
	x:=read(k)
x:=read(j)	
y:=read(i)	
write(j, 44)	
write(i, 33)	
	write(i, 55)
	y:=read(j)
	write(k, 66)

Transaction T	Transaction U
x:=read(j)	
y:=read(i)	
write(j, 44)	
write(i, 33)	
	x:=read(k)
	write(i, 55)
	y:=read(j)
	write(k, 66)

Transaction T	Transaction U
x:=read(j)	
y:=read(i)	
	x:=read(k)
write(j, 44)	
write(i 33)	
	write(i, 55)
	y:=read(j)
	write(k, 66)

8 Assignment 16.3

(I) Strict:
Before (U) accessing a resource, a transaction must wait for all previous transactions (T), accessing the same resource, to either commit or abort.
Transaction T will lock j and i. This is Strict and Transaction T will not unlock the locked resources before all reads and writes are done.

Transaction U will commit after (write(k, 66).

Transaction T	Transaction U
x:=read(j)	
y:=read(i)	
	x:=read(k)
write(j, 44)	
write(i, 33)	
COMMIT	
	write(i, 55)
	y:=read(j)
	write(k, 66)

(II) Not strict, no cascading aborts:
Not strict since U access i before T commits, but theres no cascading abort since there can be no dirty reads.

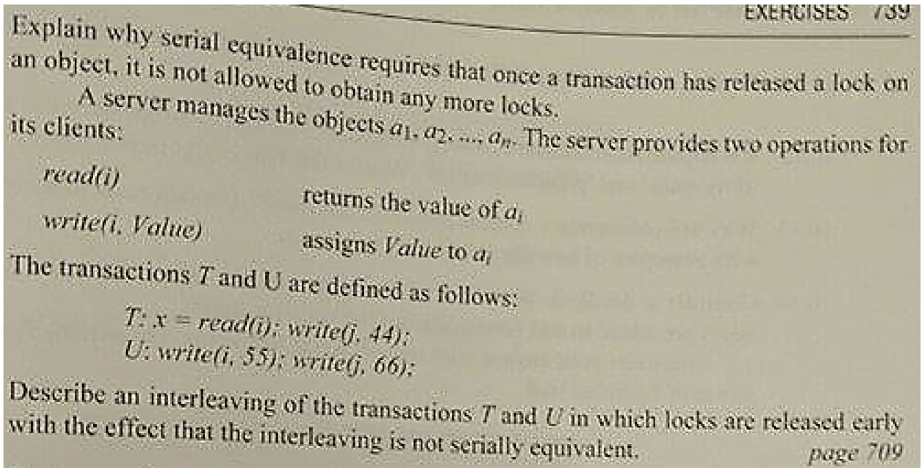
Transaction T	Transaction U
x:=read(j)	
y:=read(i)	
	x:=read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
COMMIT	
	y:=read(j)
	write(k, 66)

(III) With Cascading aborts:
U is reading before T commits. Therefore, a cascading abort can happen after U tries to read j it is not committed yet.

Transaction T	Transaction U
x:=read(j)	
y:=read(i)	
	x:=read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
	y:=read(j)
COMMIT	
	write(k, 66)

9 Assignment 16.8

Assignment Description



EXERCISES 739

Explain why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks.

A server manages the objects $a_1, a_2, \ldots, a_n$. The server provides two operations for its clients:

read(i)	returns the value of a_i
write(i, Value)	assigns *Value* to a_i

The transactions *T* and U are defined as follows:

T: x = read(i); write(j, 44);
U: write(i, 55); write(j, 66);

Describe an interleaving of the transactions *T* and *U* in which locks are released early with the effect that the interleaving is not serially equivalent. *page 709*

The reason why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks is the following:
If a transaction locks an object after already having released it once, other transactions could potentially try to access and manipulate the object. This could result in the transaction ending up with a wrong result e.g. if a bank transaction is not serial equivalent, there could be to much money or to little in an account after the transaction has ended, because the transactions have been working on different versions of an object.

A non serial equivalent interleaving of the transactions T and U could be:
T: x = read(i) U: write(i,55) U: write(j,66) T: write(j,44)

10 Assignment 16.9

Assignment Description

The transactions T and U at the server in Exercise 16.8 are defined as follows:

T: x = read(i); write(j, 44);
U: write(i, 55); write(j, 66);

Initial values of a_i and a_j are 10 and 20, respectively. Which of the following interleavings are serially equivalent, and which could occur with two-phase locking?

(a)

T	U
x= read (i);	
	write(i, 55);
write(j, 44);	
	write(j, 66);

(b)

T	U
x= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(c)

T	U
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	

(d)

T	U
	write(i, 55);
x= read (i);	
	write(j, 66);
write(j, 44);	

page 709

Both a), b), c), and d) are serially equivalent. a) is serially equivalent to T before U, b) is serially equivalent to T before U, c) is serially equivalent to U before T, d) is serially equivalent to U before T.

The 2-phase-locking protocol states that a transaction must handle its locks in two distinct, consecutive phases during the transaction's execution:

1. **Expanding phase**' (aka Growing phase): locks are acquired and no locks are released (the number of locks can only increase).
2. **Shrinking phase**: locks are released and no locks are acquired.

You can do this in all the interleavings, and so they could all happen with 2PL. b) and c) are the most obvious choices because T comes before U and U comes before T, which clearly makes it possible to acquire locks and release them. But also a) and)d could be done with 2PL. For a) you could just lock both i and j for T in the beginning and then release them one at a time, then lock them one at a time for U and release all of them in the end.

For d) you could use the same strategy just starting with the locks for U and then for T.

11 Assignment 16.16

Tabel 1: T commits first with backward validation

T	U
read(i)	
write(j,44)	
	write(i,55)
	write(j,66)
Validate	
Commit	
	Validate
	Commit

Table 1 shows that T's transactions commits while U's transaction will also comit because by the definition of backward validation, one is not allowed to read something written by another one. However, in this case U will not read anything written by T.

Tabel 2: U commits first with backward validation

T	U
read(i)	
write(j,44)	
	write(i,55)
	write(j,66)
	Validate
	Commit
Validate	
Abort	

Table 2 shows that U's transactions commits first while T's transaction will abort because by the definition of backward validation, one is not allowed to read something written by someone else concurrently. In this case T will read something written by U and should thus abort.

Tabel 3: T commits first with forward validation

T	U
read(i)	
write(j,44)	
	write(i,55)
	write(j,66)
Validate	
Commit	
	Validate
	Commit

Table 3 shows that T's transactions commits first while U's transaction will also commit because by the definition of forward validation, one is not allowed to write something read by someone else concurrently. In this case, U does not read anything and thus no conflicts are occuring.

Tabel 4: U commits first with forward validation

T	U
read(i)	
write(j,44)	
	write(i,55)
	write(j,66)
	Validate
	Abort/Commit
Validate	
Abort/Commit	

Table 4 shows that U's transactions can commit or abort annd T's transaction can also commit or abort by choice, since by the definition of forward validation, one is not allowed to write something read by someone else concurrently.. In this case U will write something that is being read by T.

So U can choose to abort its own transaction or T's transaction. *(See Lecture06.pdf slide 55 on learnit.itu.dk for definition)* However, as defined in *(Lecture06.pdf slide 44 on learnit.itu.dk for definition)*, U's transaction must abort.

12 Assignment 16.18

For two-phase locking all of the transactions (T and U) will always be committed without any conflicts because locks prevent transactions from accessing resources that are being used. On the other hand, Optimistic concurrency control is called optimistic for a reason, since it is forced to abort some transactions in various transaction sequences in order to preserve serializability,