



Lock-free programming

Lecture 12 of TDA384/DIT391
(Principles of Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg
SP1 2017/2018

Software transactional memory

Transactions

The notion of transaction, which comes from database research, supports a general approach to lock-free programming:

A **transaction** is a **sequence** of steps executed by a single thread, which are executed **atomically**.

A transaction may:

- **succeed**: all changes made by the transaction are **committed** to shared memory; they appear as if they happened instantaneously
- **fail**: the partial changes are **rolled back**, and the shared memory is in the same state it would be if the transaction had never executed

Therefore, a transaction either executes completely and successfully, or it does not have any effect at all.

Programming with transactions

The notion of transaction supports a **general approach** to **lock-free** programming:

- define a transaction for every access to shared memory
- if the transaction succeeds, there was no interference
- if the transaction failed, **retry** until it succeeds

Imagine we have a syntactic means of defining **transaction code**:

```
atomic {                                % execute Function(Arguments)  
  // transaction code                  % as a transaction (retry until success)  
}                                          atomic(Function, Arguments)  
// retry until success
```

Transactions may also support invoking **retry** and **rollback** explicitly.

(Note that **atomic** is not a valid keyword in Java or Erlang: we use it for illustration purposes, and later we sketch how it could be implemented as a function in Erlang.)

Transactions are better than locks

Transactional atomic blocks look superficially similar to monitor's methods with implicit locking, but they are in fact much **more flexible**:

- since transactions do not lock, there is **no** locking **overhead**
- **parallelism** is achieved without risks of race conditions
- since no locks are acquired, there is **no** problem of **deadlocks** (although starvation may still occur if there is a lot of contention)
- transactions **compose** easily

```
class Account {  
    void deposit(int amount)  
        { atomic {  
            balance += amount; }}  
    void withdraw(int amount)  
        { atomic {  
            balance -= amount; }}  
}
```

```
class TransferAccount extends Account {  
    // transfer from 'this' to 'other'  
    void transfer(int amount,  
                  Account other)  
        { atomic {  
            this.withdraw(amount);  
            other.deposit(amount); }}  
}
```

no locking, so no deadlock is possible!

Transactional memory

A **transactional memory** is a shared memory storage that supports atomic updates of **multiple memory locations**.

Implementations of transactional memory can be based on hardware or software:

- **hardware** transactional memory relies on support at the level of instruction sets (Herlihy & Moss, 1993)
- **software** transactional memory is implemented as a library or language extension (Shavit & Touitou, 1995)

Software transactional memory implementations are available for several mainstream languages (including Java, Haskell, and Erlang). This is still an active research topic — quality varies!

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and **version**:

```
-record(var, {name, version = 0, value = undefined}).
```

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and **version**:

```
-record(var, {name, version = 0, value = undefined}).
```

Clients use an STM as follows:

- at the beginning of a transaction, **check out** a copy of all variables involved in the transaction
- execute the transaction, which modifies the **values** of the **local** copies of the variables
- at the end of a transaction, try to **commit** all local copies of the variables

Implementing software transactional memory

We outline an implementation of software transactional memory (STM) in Erlang.

Each variable in an STM is identified by a name, value, and **version**:

```
-record(var, {name, version = 0, value = undefined}).
```

The STM's **commit** operation ensures atomicity:

- if all committed variables have the **same version number** as the corresponding variables in the STM, there were no changes to the memory during the transaction: the transaction **succeeds**
- if some committed variable has a **different version number** from the corresponding variable in the STM, there was some change to the memory during the transaction: the transaction **fails**

The counter example — with software transactional memory

```
int cnt;
```

thread t

```
int c;  
atomic {  
  c = cnt;  
  cnt = c + 1;  
}
```

thread u

```
int c;  
atomic {  
  c = cnt;  
  cnt = c + 1;  
}
```

The `atomic` translates into a `loop` that repeats `until` the transaction `succeeds`:

1. check out (`pull`) the current value of `cnt`
2. increment the local variable `c`
3. try to commit (`push`) the new value of `cnt`
4. if `cnt` has changed version when trying to commit, repeat the loop

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt • c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: \perp$	$c_u: \perp$	cnt: 0_3

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 0_3$	$c_u: \perp$	$\text{cnt}: 0_3$

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); ● c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 1_3$	$c_u: \perp$	$\text{cnt}: 0_3$

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); ● c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	$c_u: \perp$	cnt: 1 ₄

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; ● } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: 1_4$	cnt: 1_4

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); ● // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: 2_4$	cnt: 1_4

The counter example: a successful run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

thread u

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	success	cnt: 2 ₅

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt • c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: \perp$	$c_u: \perp$	$\text{cnt}: 0_3$

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

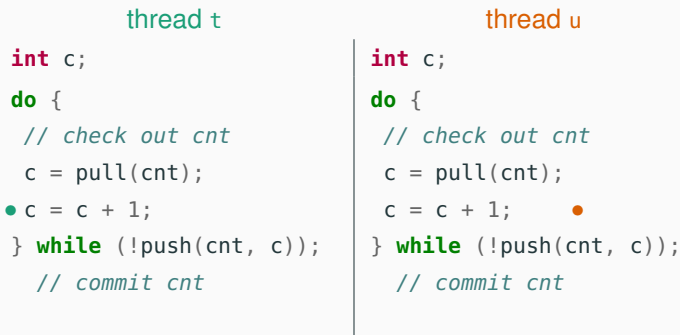
thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); • c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 0_3$	$c_u: \perp$	$\text{cnt}: 0_3$

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

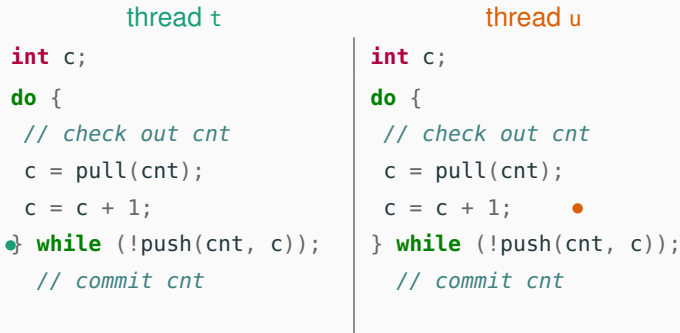


The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 0_3$	$c_u: 0_3$	$\text{cnt}: 0_3$

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$



The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 1_3$	$c_u: 0_3$	$\text{cnt}: 0_3$

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); ● // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
$c_t: 1_3$	$c_u: 1_3$	$\text{cnt}: 0_3$

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); ● // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
success	$c_u: 1_3$	cnt: 1_4

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

thread u

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	fail	cnt: 1 ₄

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); ● c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	retry	cnt: 1 ₄

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); ● c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: \perp$	cnt: 1 ₄

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

thread u

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1; ●  
} while (!push(cnt, c));  
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: 1_4$	cnt: 1_4

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); ● // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	$c_u: 2_4$	cnt: 1_4

The counter example: a retry run

$\langle \text{name: cnt, version: } x, \text{value: } y \rangle$

thread t

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

thread u

```
int c;  
do {  
  // check out cnt  
  c = pull(cnt);  
  c = c + 1;  
} while (!push(cnt, c));  
  // commit cnt
```

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	success	cnt: 2 ₅

The counter example: a retry run

`<name: cnt, version: X, value: y>`

thread t	thread u
<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>	<pre>int c; do { // check out cnt c = pull(cnt); c = c + 1; } while (!push(cnt, c)); // commit cnt</pre>

The subscript in a variable's value indicates its version:

t'S LOCAL	u'S LOCAL	STM
done	done	cnt: 2 ₅

STM in Erlang

An STM is a **server** that provides the following main operations:

- `pull(Name)`: check out a copy of variable with name `Name`
- `push(Name, Vars)`: commit all variables in `Vars`; return `fail` if unsuccessful

Clients read and write **local copies** of variables using:

- `read(Var)`: get value of variable `Var`
- `write(Var, Value)`: set value of variable `Var` to `Value`

We base the STM implementation on the `gserver` generic server implementation we presented in a previous class.

STM: operations

```
create(Tm, Name, Value) ->
  gserver:request(Tm, {create, Name, Value}).
drop(Tm, Name) ->
  gserver:request(Tm, {drop, Name}).
pull(Tm, Name) ->
  gserver:request(Tm, {pull, Name}).
push(Tm, Vars) when is_list(Vars) ->
  gserver:request(Tm, {push, Vars});
read(#var{value = Value}) ->
  Value.
write(Var = #var{}, Value) ->
  Var#var{value = Value}.
```


STM: server handlers

The storage is a **dictionary** associating variable names to variables; it is the essential part of the server state.

```
stm(Storage, {pull, Name}) ->
  case dict:is_key(Name, Storage) of
    true ->
      {reply, Storage,
       dict:fetch(Name, Storage)};
    false ->
      {reply, Storage, not_found}
  end;

stm(Storage, {push, Vars}) ->
  case try_push(Vars, Storage) of
    {success, NewStorage} ->
      {reply, NewStorage, success};
    fail ->
      {reply, Storage, fail}
  end.
```

STM: try to push

Helper function `try_push` determines if any variable to be committed has a different version from the corresponding one in the STM.

```
try_push([], Storage) ->
  {success, Storage};
try_push([Var = #var{name = Name, version = Version} | Vars],
         Storage) ->
  case dict:find(Name, Storage) of
  {ok, #var{version = Version}} ->
    try_push(Vars,
             dict:store(Name,
                        Var#var{version = Version + 1},
                        Storage));
  _ -> fail
  end.
```

Using the Erlang STM

Using the STM to create atomic functions is quite straightforward. For example, here are **pop** and **push** atomic operations for a list:

```
% pop head element from 'Name'
qpop(Tm, Name) ->
  Queue = pull(Tm, Name),
  [H|T] = read(Queue),
  NewQueue = write(Queue, T),
case push(Tm, NewQueue) of
  % push failed: retry!
  fail -> qpop(Tm, Name);
  % push successful: return head
  _ -> H
end.
```

```
% push 'Value' to back of 'Name'
qpush(Tm, Name, Value) ->
  Queue = pull(Tm, Name),
  Vals = read(Queue),
  NewQueue = write(Queue,
                    Vals ++ [Value]),
case push(Tm, NewQueue) of
  % push failed: retry!
  fail -> qpush(Tm, Name, Value);
  % push successful: return ok
  _ -> ok
end.
```

Composable transactions?

The simple implementation of STM we have outlined does not support easily **composing** transactions:

```
% pop from Queue1 and push to Queue2  
qtransfer(Tm, Queue1, Queue2) ->  
  Value = qpop(Tm, Queue1), % another process may interleave!  
  qpush(Tm, Queue2, Value).
```

To implement composability, we need to keep track of **pending transactions** and defer commits until all nested transactions have completed.

See the course's website for an example implementation:

```
% atomically execute Function on arguments Args  
atomic(Tm, Function, Args) -> todo.
```