Programmation Concurrente (SE205)

CM8 - Transactional Memory

Florian Brandner & Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay

Outline

Course Outline

- CM1-2: Introduction and shared-memory model (me)
- CM2: The *shared-memory* model (me)
- CM3-5: Concurrent programming POSIX/Java (L. Pautet)
- CM6: Actor-based programming (me)
- CM7: Patterns and Algorithms (L. Pautet)
- **CM8: Transactional memory**
  - Modify shared data structures trough *transactions*
  - Simple to program and *composable*
  - Avoid locks, mutexes, semaphores

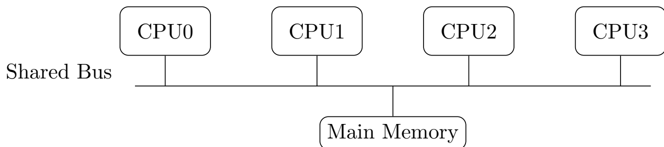
TELECOM ParisTech

Recapture

Shared-Memory Model

- All processors/threads share the same main memory
  - Data is exchanged through that memory
  - Data is shared through that memory
  - Threads synchronize through that memory
- Concurrent accesses
  - Coherency: how to threads agree on the *latest* value?
  - Consistency: in which order do updates appear *globally*

  $\Rightarrow$ Memory models cover both aspects

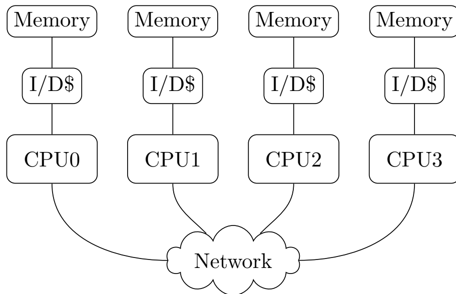


TELECOM ParisTech

Message Passing between Computers

General concept, with many different implementations:

- Several processors in potentially many computers
- No globally shared memory accessible to all processors
- Information exchange is based on *messages*



TELECOM ParisTech

Actors

Basic unit of computation:

- Actors can **communicate** among each other
- Actors can **compute** in response to a message
- Actors can **create** other actors
- Actors can designate how to handle the next message

- Guiding principles
    - Encapsulation and atomicity
    - Fairness
    - Location transparency

- Many different implementations (Akka, Erlang, Scala, . . .)

TELECOM ParisTech

Shared-Memory vs. Message Passing

Shared Memory

- A single memory
- Concurrent data accesses
- Race conditions
- Solution:
  Atomic accesses/mutex-es/semaphores/. . .

Message Passing

- No shared memory
- Exchange data by messages
- Deadlocks
- Solution:
  Partition/distribute data, raise abstraction level

TELECOM ParisTech

Shared-Memory vs. Message Passing

Shared Memory

- A single memory
- Concurrent data accesses
- Race conditions
- Solution:
  Atomic accesses/mutex-es/semaphores/. . .

Avoid conflicts by mutual exclusion

Message Passing

- No shared memory
- Exchange data by messages
- Deadlocks
- Solution:
  Partition/distribute data, raise abstraction level

Avoid conflicts by partitioning data

TELECOM ParisTech

Transactional Memory

Alternative to handle conflicts

Assume again a shared memory:

- Replace atomic accesses/mutexes/semaphores/...
  - Which ensure mutual exclusion before conflicts occur
  - Block before entering critical section
- Assumes by default that no conflicts occur:
  - Immediately enter critical section (no blocking)
  - Check for conflicts before leaving the critical section
  - Revert changes and retry in case of conflicts

TELECOM ParisTech

Example: Simple Transactions

```
int a = 4; x = 0; y = 0; z = 0;
```

Thread 1

```
x = 6;
y = a;
z = 5;
```

Thread 2

```
if (x + y > z) {
  a = 7;
}
```

Thread 2 either sees `x = y = z = 0` or `x = 6`, `y = 4`, and `z = 5`. Nothing in between.

TELECOM ParisTech

Transactional Memory: Concepts

- Transactions are indicated by *scopes*
  (For instance: **`atomic`** `{...}`)
- Transactions keep track of all memory accesses
  - Read set R: Memory locations read during a transaction
  - Write set W: Memory locations written during a transaction
- Memory writes do not actually write to memory
  - Instead, write into a buffer
  - We will see how this works out later

TELECOM ParisTech

Group Exercise: Read and Write Sets

Form groups of three:

- What are the read/write sets of the two threads?
- Is there an overlap?
- What might this indicate?

```
int a = 4; x = 0; y = 0; z = 0;
```

Thread 1

```
x = 6;
y = a;
z = 5;
```

Thread 2

```
if (x + y > z) {
  a = 7;
}
```

TELECOM ParisTech

Group Exercise: Read and Write Sets

Form groups of three:

- What are the read/write sets of the two threads?
- Is there an overlap?
- What might this indicate?

```
int a = 4; x = 0; y = 0; z = 0;
```

Thread 1

```
x = 6;
y = a;
z = 5;
```

$R_{\text{Thread 1}} = \{@a\}$

$W_{\text{Thread 1}} = \{@x, @y, @z\}$

Thread 2

```
if (x + y > z) {
  a = 7;
}
```

$R_{\text{Thread 2}} = \{@x, @y, @z\}$

$W_{\text{Thread 2}} = \{@a\}$ or $\emptyset$

TELECOM ParisTech

Conflict Detection

Transaction may *conflict*:

- Check for overlap in the read/write sets:
  - R_A/W_A: read/write sets of transaction A
  - R_B/W_B: read/write sets of another transaction B
  - Conflict if: $(R_a \cup W_a) \cap W_b \neq \emptyset$
- Conflicts are resolved by restarting one of the two transactions
- When are conflicts detected?
  - Eager: Check for conflicts as read/write sets evolve
  - Lazy: When some transaction completes (*commit*)

TELECOM ParisTech

Group Exercise: Conflict Detection

Form groups of three:

- Assume the following interleaving: $\alpha_1, \alpha_2, \beta_1, \alpha_3, \ldots$
- What happens after α_3 (β_2 or β_3, or something else)?
- Which transaction finishes first?
- What happens with eager conflict detection?
- What happens with lazy conflict detection?

```
int a = 4; x = 0; y = 0; z = 0;
```

Thread 1

```
α1  x = 6;
α2  y = a;
α3  z = 5;
```

Thread 2

```
β1  if (x + y > z) {
β2      a = 7;
β3  }
```

TELECOM ParisTech

Example: Eager Conflict Detection

Statement	Thread 1 R	Thread 1 W	Thread 2 R	Thread 2 W
α_1	$\emptyset$	{ @x }	$\emptyset$	$\emptyset$
α_2	{ @a }	{ @x, @y }	$\emptyset$	$\emptyset$
β_1	{ @a }	{ @x, @y }	{ @x, @y, @z }	$\emptyset$
↯ Conflict!! – β restarts				
α_3	{ @a }	{ @x, @y, @z }	–	–
α commits transaction				
β_1	–	–	{ @x, @y, @z }	$\emptyset$
β_2	–	–	{ @x, @y, @z }	{ @a }
β_3	–	–	{ @x, @y, @z }	{ @a }
β commits transaction				

TELECOM ParisTech

Example: Lazy Conflict Detection

Statement	Thread 1 R	Thread 1 W	Thread 2 R	Thread 2 W
α_1	$\emptyset$	{ @x }	$\emptyset$	$\emptyset$
α_2	{ @a }	{ @x, @y }	$\emptyset$	$\emptyset$
β_1	{ @a }	{ @x, @y }	{ @x, @y, @z }	$\emptyset$
no conflict detected yet				
α_3	{ @a }	{ @x, @y, @z }	{ @x, @y, @z }	$\emptyset$
α commits transaction ↯ Conflict!! β restarts				
β_1	–	–	{ @x, @y, @z }	$\emptyset$
β_2	–	–	{ @x, @y, @z }	{ @a }
β_3	–	–	{ @x, @y, @z }	{ @a }
β commits transaction				

TELECOM ParisTech

Advantages of Transactional Memories

- Transactions may improve performance
  - Atomic operations lock the bus/network to memory
  - Locks/mutexes/semaphores need to be acquired
  - This overhead can be avoided, if conflicts are rare
  - Non-conflicting transactions may also execute in parallel
- Transactions are composable
  - Taking two/more locks may easily lead to deadlocks
  - Taking two/more locks often is not sufficient
    $\Longrightarrow$ additional locks needed
  - Transactions may overlap without any problem
  - Much simpler programing model

TELECOM ParisTech

Group Exercise: Locks and Composability

Form groups of three:

- Assume a thread-safe list implementation:
- Adding (`add`) and removing (`remove`) are protected by a lock
- The data structure is safe and cannot be corrupted
- What is the problem with the code below?
- How could the problem be fixed?

```
boolean move(List<T> s, List<T> d, T item) {
  s.remove(item);
  d.add(item);
}
```

TELECOM ParisTech

Example: Locks and Composability

The operation `move` is not atomic:

- Another thread might search for some `item` *x*
- If it does so after `remove`, but before `add`
- It would appear as if *x* would neither exist in `s` nor in `d`

```
boolean move(List<T> s, List<T> d, T item) {
  s.remove(item); // lock/unlock s
  // other thread executes - item is "nowhere"
  d.add(item);    // lock/unlock d
}
```


TELECOM ParisTech

Example: Locks and Composability (2)

Making `move` atomic:

- Requires an additional lock `m`
- Any other thread also needs to acquire `m`
- Clearly, locks cannot be composed easily

```
boolean move(List<T> s, List<T> d, T item) {
  lock(m)  // other threads too have to acquire m
  s.remove(item); // lock/unlock s
  d.add(item);    // lock/unlock d
  unlock(m)
}
```

TELECOM ParisTech

Transactional Memory and Composability

How does transactional memory help?

- Assume two threads execute `move` on same lists
- Nested transactions inside `add` and `remove`
- Simply unify read/write sets with surrounding transaction
- Commit only at end of outer-most transaction
- Transactions compose nicely!

```
boolean move(List<T> s, List<T> d, T item) {
  atomic {
    s.remove(item); // nested atomic
    d.add(item);    // nested atomic
  }
}
```


TELECOM ParisTech

Downsides of Transactional Memory

Transactional memory seems nice, but:

- Is expensive to implement
  - In software:
    intercept/rewrite all memory accesses, explicitly manage buffer and read/write sets $\Longrightarrow$ expensive
  - In hardware (SPARC, POWER8, x86 TSX):
    More efficient, but limited buffer sizes
- Retries are costly:
  - Depends on frequency of conflicts
- Limits to composability:
  - Operations with side-effects (I/O, thread creation, ...)
  - Incompatible with usage of locks

TELECOM ParisTech

Summary

Transactional memory:

- Alternative to atomic accesses/locks/mutexes/...
- Optimistically assume that no conflicts will occur
  - Directly enter the critical section
  - Track read/write sets
  - Buffer all memory writes
- Conflict detection
  - Check for overlapping read/write sets of concurrent transactions
  - Can be continuously (eager) or at commit (lazy)
- Successful transactions
  - When no conflict was detected
  - Transfer buffered memory writes to main memory
  - Other transactions: restart

TELECOM ParisTech

Questions?