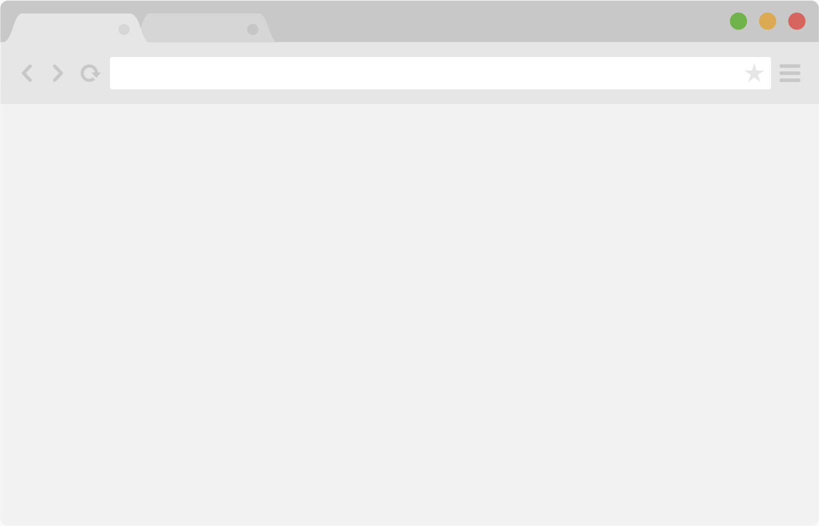
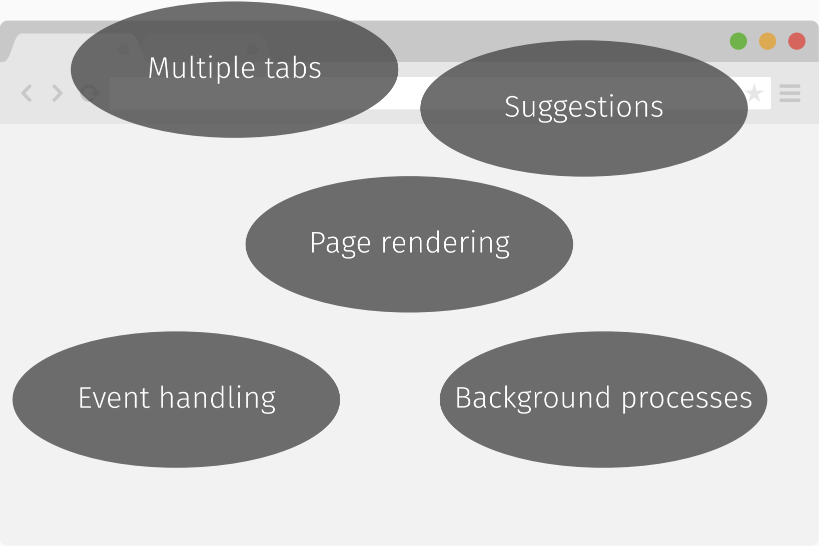
Composable Concurrency Models

Dan Stelljes

November 19, 2016





Multiple tabs

Suggestions

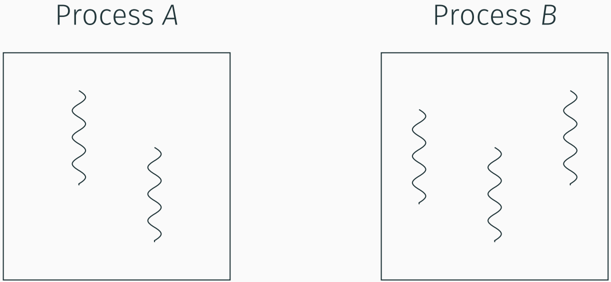
Page rendering

Event handling

Background processes

1. Background
    - Concurrency
    - Complications
    - Consistency models
2. Concurrency models
    - Atomic variables
    - Software transactional memory
    - Communicating threads
3. Composability
    - Correctness criteria
    - Possible conflicts
    - Ongoing work

Threads and processes



Process *A*

Process *B*

- **Threads** are independent sequences of operations.
- **Processes** are instances of programs made up of one or more threads.

Concurrency

The “happens before” ($\rightarrow$) relation[1]

$A \rightarrow B$ if one of the following is true:

1. A and B are operations in the same thread and A occurs before B.
2. A is the sending of a message by one thread and B is the receipt of the same message by another thread.

A and B are said to be concurrent if $A \nrightarrow B$ and $B \nrightarrow A$.

[1] Lamport, “Proving the Correctness of Multiprocess Programs.”

Complications

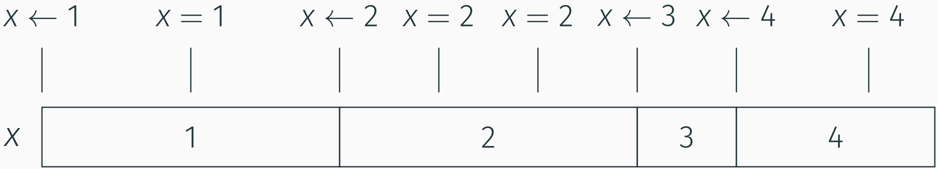
- **Sequential program:** Does the order of operations yield a correct result?

Complications

- **Sequential program:** Does the order of operations yield a correct result?
- **Concurrent program:** Does *every possible* order of operations yield a correct result?

Complications

Single thread:

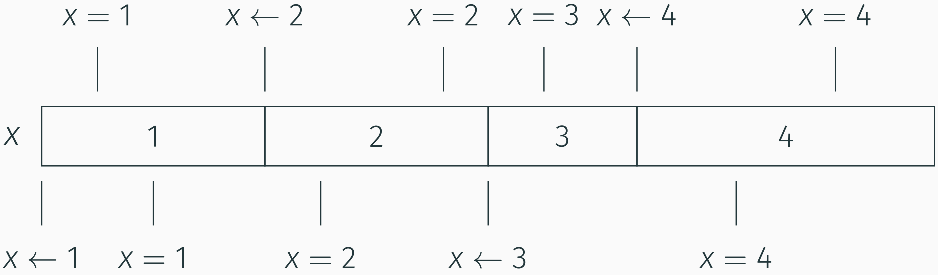


$x \leftarrow 1$ $x = 1$ $x \leftarrow 2$ $x = 2$ $x = 2$ $x \leftarrow 3$ $x \leftarrow 4$ $x = 4$

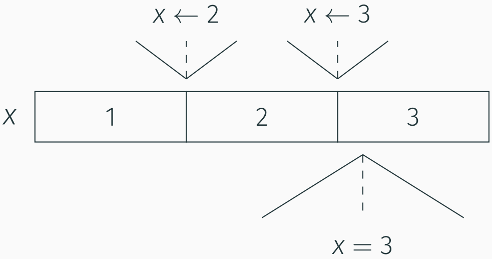
X | 1 | 2 | 3 | 4

Complications

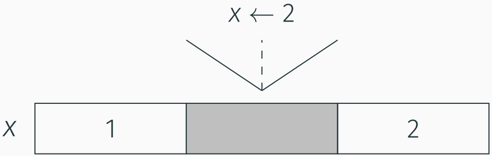
Multiple threads:



Complications

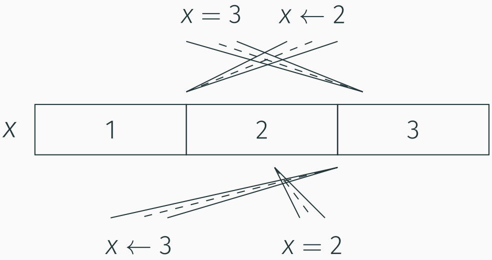


Linearizability



- Linearizability guarantees that the completion of an operation on a single object will appear to be instantaneous.
- The results of a linearizable operation will be visible as soon as the operation is complete.

Serializability



- Serializability guarantees that operations can occur in any order as long as an equivalent sequential ordering exists.
- While a serializable set of operations is being executed, it appears to be the only set of operations being executed.

Strict serializability

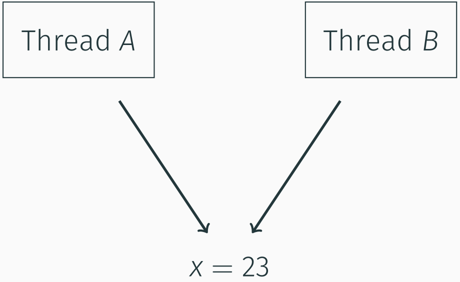
- Linearizability *and* serializability yield strict serializability, which guarantees both consistency and isolation.

Strict serializability[2]

An ordering of operations is equivalent to some sequential ordering and that ordering corresponds to the order of execution in real time.

[2] Herlihy and Wing, “Linearizability: A Correctness Condition for Concurrent Objects.”

Atomic variables

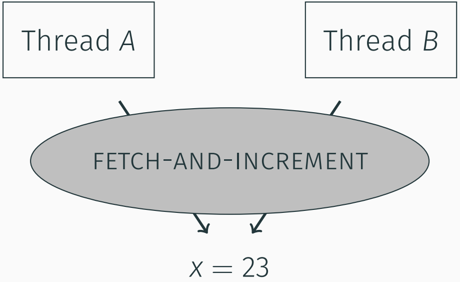


$x = 23$

1. A reads x
2. B reads x
3. A increments value
4. A writes incremented value to x
5. B increments value
6. B writes incremented value to x

$x = 24$

Atomic variables

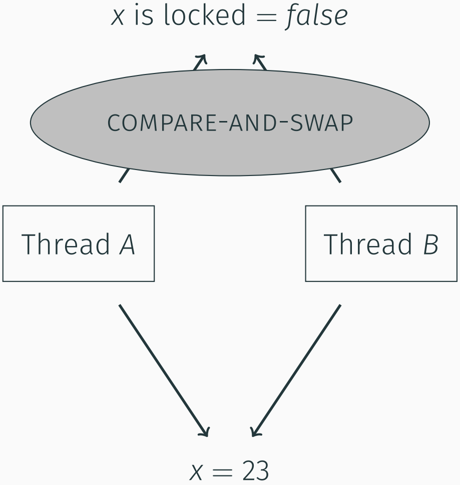


$x = 23$

1. A calls FETCH-AND-INCREMENT on x
2. B calls FETCH-AND-INCREMENT on x

$x = 25$

Atomic variables



x is locked $=$ *false*

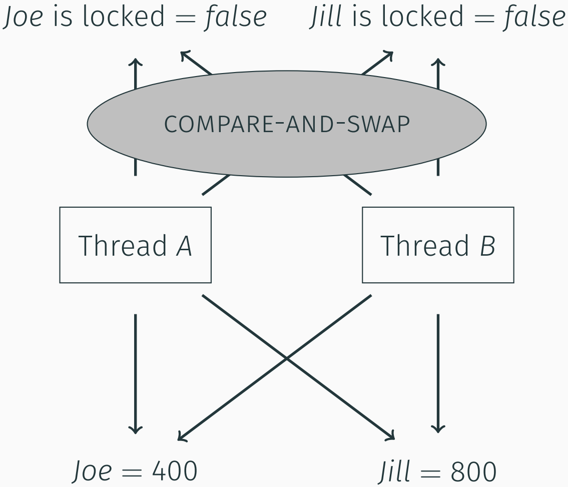
COMPARE-AND-SWAP

Thread A

Thread B

$x = 23$

Atomic variables



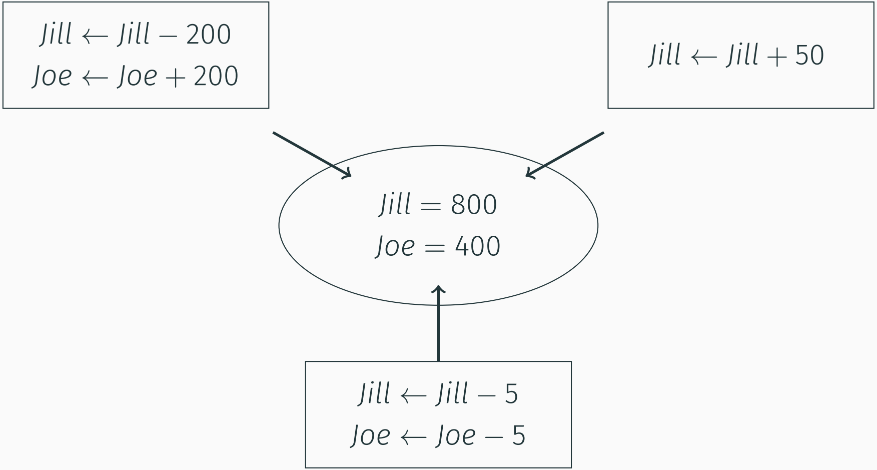
Software transactional memory

Software transactional memory (STM) is an optimistic approach to working with shared memory:[3]

1. A thread writes to a shared memory location, keeping track of the transaction in a log.
2. If there are conflicting changes at the end of the transaction, the transaction is aborted and retried.
3. If there are no conflicts, the changes are committed and become visible.

[3] Shavit and Touitou, "Software Transactional Memory."

Software transactional memory



$Jill \leftarrow Jill - 200$

$Joe \leftarrow Joe + 200$

$Jill \leftarrow Jill + 50$

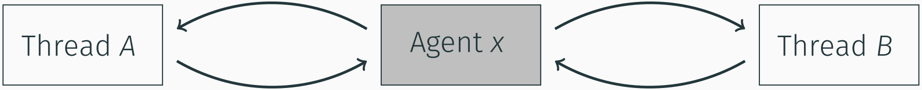
$Jill = 800$

$Joe = 400$

$Jill \leftarrow Jill - 5$

$Joe \leftarrow Joe - 5$

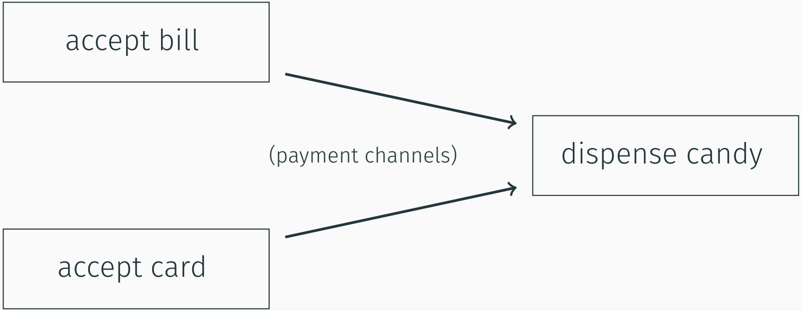
Communicating threads



Agents: An isolated thread wraps an object.[4]

[4] Swalens et al., “Towards Composable Concurrency Abstractions.”

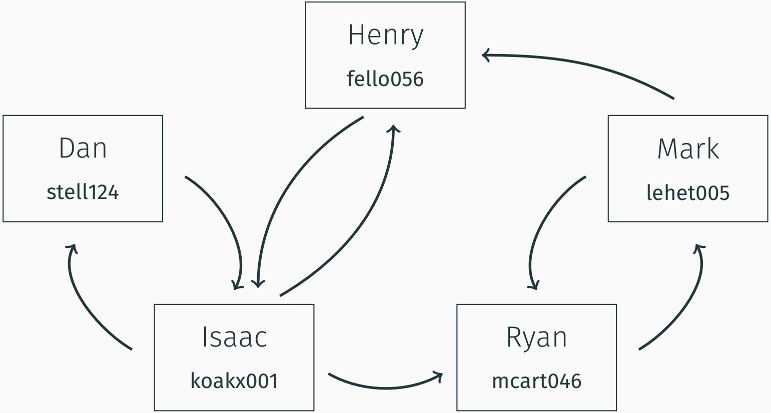
Communicating threads



Communicating sequential processes (CSP): Independent threads communicate synchronously through predefined channels.[5]

[5] Hoare, “Communicating Sequential Processes.”

Communicating threads



The actor model: Independent threads send messages to known addresses.[6]

[6] Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.*

Correctness criteria

How do we know that models are composable?[7]

- Safety: “Nothing bad will happen!” (The output of a program or algorithm will not be incorrect.)
- Liveness: “Something will eventually happen!” (The program or algorithm will terminate.)

Two models are composable if using them within each other doesn’t compromise safety or liveness.

[7] Swalens et al., “Towards Composable Concurrency Abstractions.”

Possible conflicts

using within	Safety				Liveness			
	atoms	refs	agents	channels	atoms	refs	agents	channels
atoms	✗	✗	✗	✗	✓	✓	✓	✗
refs	✗	✓	✓	✗	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✗
channels	✓	✓	✓	✓	✓	✓	✗	✗

Possible conflicts

	Safety				Liveness			
using within	atoms	refs	agents	channels	atoms	refs	agents	channels
atoms	✗	✗	✗	✗	✓	✓	✓	✗
refs	✗	✓	✓	✗	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✗
channels	✓	✓	✓	✓	✓	✓	✗	✗

- A model reexecutes code that performs an irrevocable action.

Possible conflicts

	Safety				Liveness			
using within	atoms	refs	agents	channels	atoms	refs	agents	channels
atoms	✗	✗	✗	✗	✓	✓	✓	✗
refs	✗	✓	✓	✗	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✗
channels	✓	✓	✓	✓	✓	✓	✗	✗

- A model reexecutes code that performs an irrevocable action.
- A model reexecutes code that causes the reexecution to continually happen.

Possible conflicts

	Safety				Liveness			
using within	atoms	refs	agents	channels	atoms	refs	agents	channels
atoms	✗	✗	✗	✗	✓	✓	✓	✗
refs	✗	✓	✓	✗	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✗
channels	✓	✓	✓	✓	✓	✓	✗	✗

- A model reexecutes code that performs an irrevocable action.
- A model reexecutes code that causes the reexecution to continually happen.
- A model that supports blocking operations is used within a model that doesn’t.

Possible conflicts

using within	Safety				Liveness			
	atoms	refs	agents	channels	atoms	refs	agents	channels
atoms	✗	✗	✗	✗	✓	✓	✓	✗
refs	✗	✓	✓	✗	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✗
channels	✓	✓	✓	✓	✓	✓	✗	✗

- A model reexecutes code that performs an irrevocable action.
- A model reexecutes code that causes the reexecution to continually happen.
- A model that supports blocking operations is used within a model that doesn’t.
- A model does not guarantee safety or liveness by design.

Ongoing work

- Composable "building blocks" (thread creation, message passing, etc.) that could be used to build common concurrency models[8]

[8] Swalens et al., "Towards Composable Concurrency Abstractions."

[9] Marr and D'Hondt, "Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines."

[10] Ziv et al., "Composing Concurrency Control."

Ongoing work

- Composable "building blocks" (thread creation, message passing, etc.) that could be used to build common concurrency models[8]
- Unifying abstractions for high-level language virtual machines[9]

[8] Swalens et al., "Towards Composable Concurrency Abstractions."

[9] Marr and D'Hondt, "Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines."

[10] Ziv et al., "Composing Concurrency Control."

Ongoing work

- Composable “building blocks” (thread creation, message passing, etc.) that could be used to build common concurrency models[8]

- Unifying abstractions for high-level language virtual machines[9]

- Formal theories for safely composing concurrency control[10]

[8] Swalens et al., “Towards Composable Concurrency Abstractions.”

[9] Marr and D’Hondt, “Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines.”

[10] Ziv et al., “Composing Concurrency Control.”

Thanks to Elena Machkasova, K.K. Lamberty, and Matthew Justin for their guidance and suggestions.

github.com/dstelljes/senior-sem

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[2] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects.” In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492. ISSN: 0164-0925. DOI: `10.1145/78969.78972`.

[3] C. A. R. Hoare. “Communicating Sequential Processes.” In: *Communications of the ACM* 21.8 (1978), pp. 666–677. ISSN: 0001-0782. DOI: `10.1145/359576.359585`.

[4] Leslie Lamport. “Proving the Correctness of Multiprocess Programs.” In: *IEEE Transactions on Software Engineering* 3.2 (1977), pp. 125–143. ISSN: 0098-5589. DOI: `10.1109/TSE.1977.229904`.

References

[5] Stefan Marr and Theo D'Hondt. "Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines." In: *Objects, Models, Components, Patterns: 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*. Ed. by Carlo A. Furia and Sebastian Nanz. Berlin, Heidelberg, Germany: Springer-Verlag, 2012, pp. 171–186. ISBN: 978-3-642-30561-0. DOI: `10.1007/978-3-642-30561-0_13`.

[6] Nir Shavit and Dan Touitou. "Software Transactional Memory." In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: `10.1145/224964.224987`.

References

[7] Janwillem Swalens et al. "Towards Composable Concurrency Abstractions." In: *Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*. PLACES '14. Grenoble, France: EPTCS 155, 2014, pp. 54–60. DOI: `10.4204/EPTCS.155.8`.

[8] Ofri Ziv et al. "Composing Concurrency Control." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 240–249. ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737970`.