Aalto University
School of Science

CS-E4110 Concurrent Programming

Week 6 – Exercise session

2021-12-10

14:15 – 16:00

Jaakko Harjuhahto

Today

- Weak memory models
- Demo: C++ atomics
- Reactor Task-C tips
- Discussion: Reactor Task-B solution
- General Q&A

Weak Memory Models

Aalto-yliopisto
Teknillinen korkeakoulu

J.Harjuhahto
2021-12-10

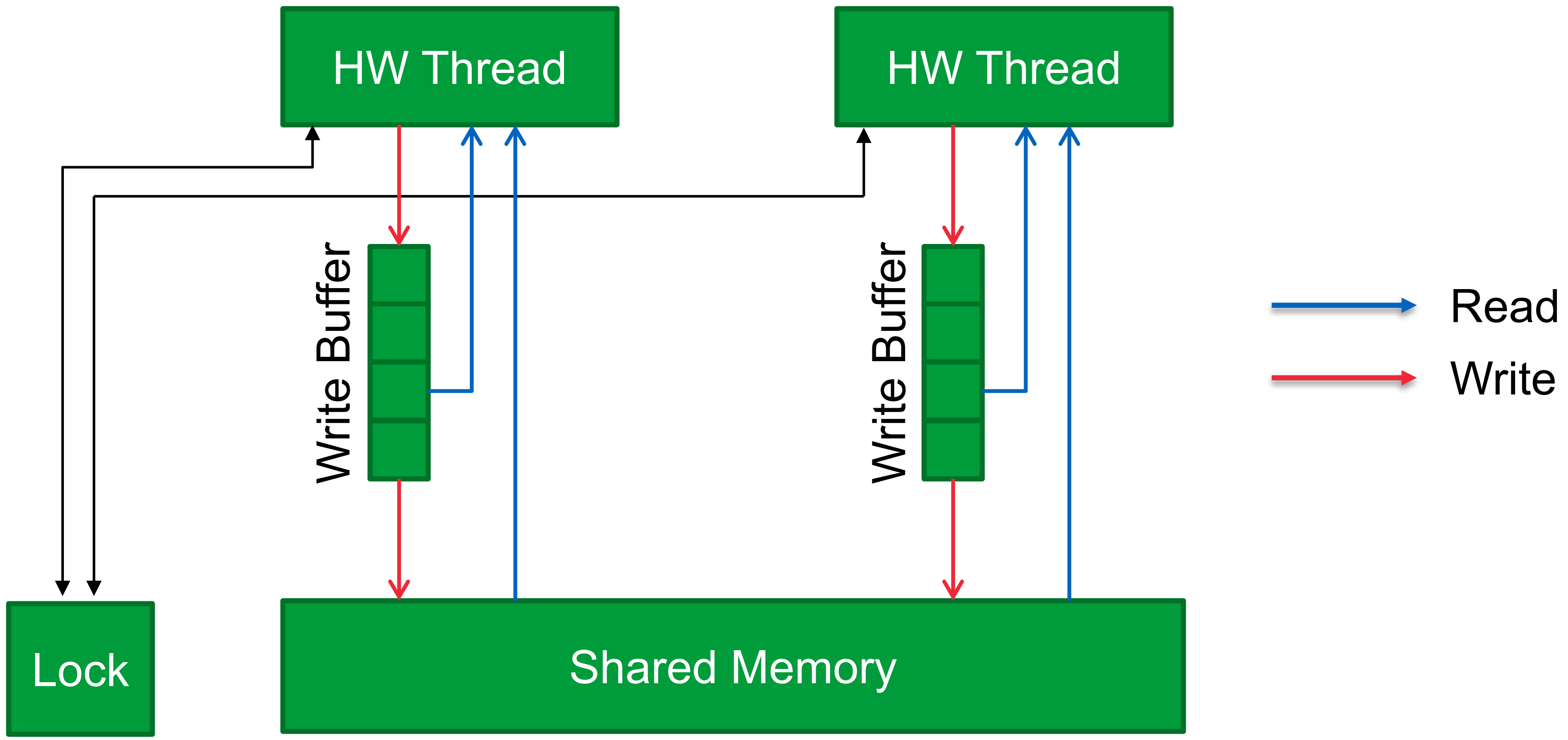
The Hardware Reality

- Shared memory multi-processors are the de factor norm
- Relaxing semantics allows greater utilization of system resources for performance
  - Weak memory behaviours
  - Instruction reordering
  - Speculative execution
  - And many more...
- Hardware increasingly optimized and weakly consistent
- Weak behaviours are an active area of research

x86-TSO

- TSO comes from ”Total Store Order”
  - Memory model developed by Sewell, Sarkar, Owens, Nardelli, and Myreen in 2010 to describe how x86 multi-core processors treat shared memory reads and writes
  - Model for an abstract machine, that is much easier to reason about compared to comprehensive architectural specifications
  - https://doi.org/10.1145/1785414.1785443
  - About 8 pages in length
- Intel’s manuals for x86 and x64 software developers
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - More than 4700 pages of technical data

The x86-TSO Model



Dekker's Algorithm for Mutual Exclusion

boolean wantp ← false, wantq ← false integer turn ← 1	
P	**Q**
loop forever	loop forever
p1: non-critical section	p1: non-critical section
p2: wantp ← true	p2: wantq ← true
p3: while wantq	p3: while wantp
p4: if turn = 2	p4: if turn = 1
p5: wantp ← false	p5: wantq ← false
p6: await turn = 1	p6: await turn = 2
p7: wantp ← true	p7: wantq ← true
p8: critical section	p8: critical section
p9: turn ← 2	p9: turn ← 1
p10: wantp ← false	p10: wantq ← false

See Ben-Ari sections 3.9 nad 4.5 for a proof of correctness

Dekker's Algorithm on x86 TSO

boolean wantp ← false, wantq ← false integer turn ← 1	
P	Q
loop forever p1: non-critical section p2: wantp ← true p3: while wantq p4: if turn = 2 p5: wantp ← false p6: await turn = 1 p7: wantp ← true p8: critical section p9: turn ← 2 p10: wantp ← false	loop forever p1: non-critical section p2: wantq ← true p3: while wantp p4: if turn = 1 p5: wantq ← false p6: await turn = 2 p7: wantq ← true p8: critical section p9: turn ← 1 p10: wantq ← false

Step	Thread P	Thread Q	wantp	wantq	turn	Write buffer P	Write buffer Q
1.	p_1	q_1	$false$	$false$	1		
2.	p_2	q_1	$false$	$false$	1		
3.	p_3	q_1	$false$	$false$	1	$wantp \leftarrow true$	
4.	p_3	q_2	$false$	$false$	1	$wantp \leftarrow true$	
5.	p_3	q_3	$false$	$false$	1	$wantp \leftarrow true$	$wantq \leftarrow true$
6.	p_8	q_3	$false$	$false$	1	$wantp \leftarrow true$	$wantq \leftarrow true$
7.	p_8	q_8	$false$	$false$	1	$wantp \leftarrow true$	$wantq \leftarrow true$

During steps 1-3, thread P initiates the pre-protocol for entering the critical section by writing `true` to `wantp`. The write is appended to the threads write buffer, but is not immediately flushed to main memory. Thread Q performs it's respective operation in steps 4-5.

Dekker's Algorithm on x86 TSO

boolean wantp ← false, wantq ← false integer turn ← 1	
P	Q
loop forever p1: non-critical section p2: wantp ← true p3: while wantq p4: if turn = 2 p5: wantp ← false p6: await turn = 1 p7: wantp ← true p8: critical section p9: turn ← 2 p10: wantp ← false	loop forever p1: non-critical section p2: wantq ← true p3: while wantp p4: if turn = 1 p5: wantq ← false p6: await turn = 2 p7: wantq ← true p8: critical section p9: turn ← 1 p10: wantq ← false

Write buffers have not yet flushed to shared memory

Both threads in the critical section

Step	Thread P	Thread Q	wantp	wantq	turn	Write buffer P	Write buffer Q
1.	p_1	q_1	*false*	*false*	1		
2.	p_2	q_1	*false*	*false*	1		
3.	p_3	q_1	*false*	*false*	1	$wantp \leftarrow true$	
4.	p_3	q_2	*false*	*false*	1	$wantp \leftarrow true$	
5.	p_3	q_3	*false*	*false*	1	$wantp \leftarrow true$	$wantq \leftarrow true$
6.	p_8	q_3	*false*	*false*	1	$wantp \leftarrow true$	$wantq \leftarrow true$
7.	p_8	q_8	*false*	*false*	1	$wantp \leftarrow true$	$wantq \leftarrow true$

During steps 1-3, thread P initiates the pre-protocol for entering the critical section by writing `true` to `wantp`. The write is appended to the threads write buffer, but is not immediately flushed to main memory. Thread Q performs it's respective operation in steps 4-5.

Exploiting Weak Memory Models

- Weak memory behaviour is also an opportunity
  - Exploit detailed control over memory behaviours
  - Increase opportunities for hardware level concurrency, improve performance, optimize energy consumption
- C++ offers more control over memory effects
  - Low level atomic variables and control over memory order
    - https://en.cppreference.com/w/cpp/atomic
    - https://en.cppreference.com/w/cpp/atomic/memory_order
  - Memory/thread fences to force synchronization order
    - https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence
  - Part of the standard library and will work for any architecture with a compliant compiler
    - I.e. Usable on all of: x86, ARM and POWER

Demo: C++ Atomics

Reactor Task-C Tips

Aalto-yliopisto
Teknillinen korkeakoulu

J.Harjuhahto
2021-12-10

Task C in a Nutshell

- Use the Dispatcher to implement a small network game
  - No threads, JVM monitors or synchronization needed... Why?
- There is no pre-defined API
  - The task is specified as game behaviour
- We want to see proper use of the Reactor pattern, i.e. the Dispatcher from Task B
  - Event based game application logic
  - Handling of all possible scenarios that can arise from processing events in arbitrary order

Task C in a Nutshell

- `HangmanGame` holds game state and logic
  - Current game state, initialized as:
    - `GameState(hiddenWord, initialGuessCount, Set())`
    - The empty set is the collection of player guesses, initially empty
  - A single instance of the Dispatcher
  - Handler for accepting incoming connections
    - The reference to the handler must be retained, so that it can be removed from the Dispatcher later on, when the game ends
  - Current players and references to event handlers for these player connections
    - Alternatively, you can think about the event handler for a specific player connection as an object representing the player

The Necessary Event Handlers

- Write event handler classes to match the two handles:
  - E.g. `AcceptHandler` for `AcceptHandle`
    - Adds a new player to the game
  - E.g. `PlayerHandler` for `TCPTextHandle`
    - First input assigns a name to the player
    - Progresses game state by making guesses
  - The names are suggestions, the instructions are agnostic about how you name the internals of the `HangmanGame`, as long the implementation is readable and clearly documented

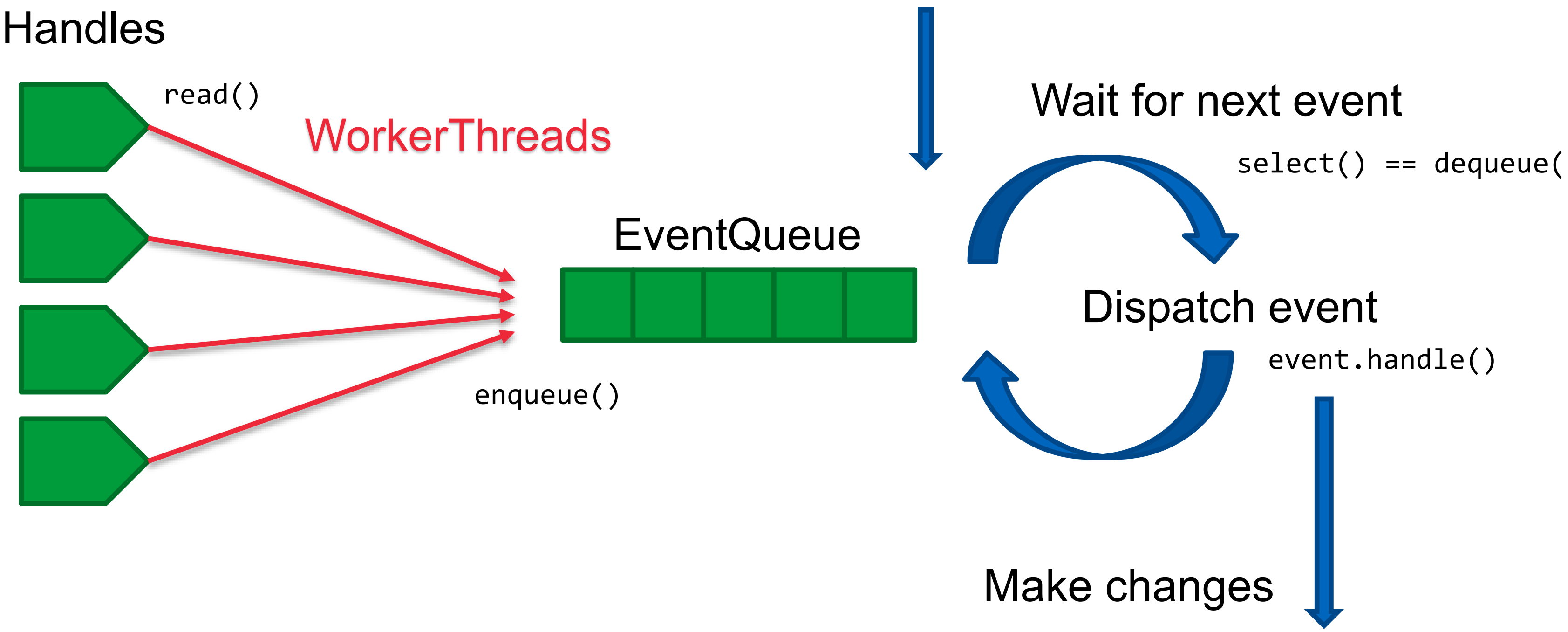
Starting the Game

- The game server is started by:
  - Creating a new `HangmanGame` with the command line arguments for the hidden word and the number of guesses
- Inside the `HangmanGame`
  - Creating an `AcceptHandler` and adding it to the Dispatcher
  - Call `handleEvents()` on the Dispatcher
  - The call to `handleEvents()` will block until the game ends
  - The entire application (= JVM) exits

Discussion: Reactor Task-B Solution

The Dispatcher

The ”main” thread will initialize the `Dispatcher` with one or more handlers, then call `handleEvents()`.



Changing application state can involve adding new handlers to the `Dispatcher` as well are removing existing handlers. Note that it is the ”main” thread that executes these changes.

Application State

Q&A