Synchronized queues

&

Sofware Transactional Memory

Terminology

- **Concurrency**: programming structuring technique involving multiple threads of control.
  - A mean to improve responsiveness / modularity
  - Does not necessarily aim at performance improvement
    (it is but one way to implement parallelism)

- **Synchronization**: coordination between threads of control to agree on a particular sequence of action.
  - Waiting for an asynchronous computation (future)
  - Ensuring shared mutable data consistency
  - Joining threads at program exit
  - Etc.

Purpose

- This presentation will focus on some synchronization means to achieve data consistency in a concurrent application.
- The goal is to introduce the following concepts:
  - Synchronized queues
  - Software transactional memory
- And to demonstrate them on two simple examples:
  - A concurrent logger
  - A concurrent bank account
- Note: all examples have been implemented in Haskell, for simplicity. But do not worry, there are not many snippets :)

Agenda

- Low level synchronization primitives
- Synchronized queues
  - Example: a thread-safe logger
- Limitations of lock-based synchronization
- Software Transactional Memory
  - Example: a thread-safe bank account
- Conclusion
- Sources

Low level synchronization primitives

- **MUTual EXclusion**: process of defining critical sections that only one thread of control can execute at a time
  - Mutex acquisition
  - Execution of the critical section
  - Mutex release

- Mutexes also **constraints compiler optimization**, especially regarding the re-ordering of instructions:
  - Code before an acquire may move into the critical section
  - Code after the release may move into the critical section
  - Code inside the critical section may not be move outside (of course)
  - Code outside the critical section may not cross it (less obvious)

Low level synchronization primitives

- **Mutexes** are very low level and error prone
  - Placed inside a recursive function, a mutex will lead to deadlock, even with one single thread
- **Re-entrant locks**: a higher level construct on top of mutexes
  - Can be acquired by at most one thread
  - But can be acquired several times by the same thread
- **Monitor**: class maintaining its own re-entrant lock
  - Synchronized methods are wrapped inside an acquire / release of the object lock
  - Somehow a syntactic sugar for a decorator / proxy

Synchronized queues

- If several threads try to access a critical section:
  - One of them will be granted access
  - The other threads are stalled until the task is completed
- Having computation intensive critical sections will lower the reactivity of your system
- **Synchronized queue** will help minimizing this blocking time:
  - Tasks are en-queued by callers
  - To be executed later by worker thread(s)
  - The critical section is limited to the queue access

  Explicitly playing with locks is error prone anyway

Example: a thread-safe logger

- Problem definition:
  - Implement a simple logger that writes strings into a file.
  - Several threads might concurrently access it.
- First design, based on the **monitor** concept:
  - Callers will wait for IO operations to complete
- Second design, based on **synchronized queues**:
  - Callers will simply queue log requests
  - It may increase the application memory usage
  - Do not forget to wait for the logger at application shutdown!
    (for e.g. send a shutdown request and wait for the answer)

Example: a thread-safe logger

- The monitor-based logger performs IO operations in the caller thread:

```
-- | Log a new message
logMsg :: SyncLogger -> Format -> String -> IO()
logMsg logger fmt str =
   RLock.with(_lock logger) $
      hPutStrLn (_file logger) =<< formatMsg fmt str
```

- Whereas the queue-based logger sends a message to be later processed by a single worker thread:

```
-- | Log a new message
logMsg :: ChanLogger -> Format -> String -> IO()
logMsg logger fmt str =
   send logger $ LogCommand fmt str
```

Example: a thread-safe logger

- Test description
  - 8 threads, logging 5000 strings prefixed with the logger UTC time
  - Wait for the caller threads completion and measure delay
  - Wait for the logger to finish and measure total time elapsed

- Test results (4 cores)
  - Monitor-based logger: 1.34s total time, 1.34s delay
  - Queue-based logger: 0.68s total time, 0.13s delay

- It is not a real latency measure, but still...
  - It makes you want to choose the queue-based logger

Limitations of lock-based primitives

- Let us start by solving a simple problem:
  - Design a thread safe bank account
  - Users can deposit / withdraw money
  - Users can access the balance

- Accessing and incrementing a number is not computer-intensive, hence the monitor-based approach.

- The account provides two synchronized methods:
  - Deposit, which takes an number (to add to the balance)
  - GetBalance, which returns the value of the balance

Limitations of lock-based primitives

- But a new user story comes, asking for:
  - Atomic money transfers
  - Atomic global observation of balances

- In order to be made atomic, both transfers and balance observations should lock all their target objects before starting their task.

- Locking the objects as they arrive will result in tricky deadlocks: consider concurrent transfers from A to B and B to A.
  - One solution is to have globally ordered locks
  - Or we could use a try-lock approach: attempt to acquire all locks, and rollback if not successful.

Limitations of lock-based primitives

- Whatever strategy you choose, it should be consistent all over the application (the order of locks should be the same everywhere)
  - Encapsulation is violated
  - Say hello to error-prone code bloat
- As you cannot reason locally, testing your application:
  - Becomes much harder (you have to mind your surroundings)
  - Does not ensure it works forever (new clients might forget to RTFM)
- Synchronized queues suffer many drawbacks: you end up with a single queue to guaranty atomicity.
- Conclusion: lock-based approaches are not composable.

Software transactional memory

- **STM** is based on **optimistic locks**. Instead of acquiring a lock to execute the task T:
  - Capture the initial "state of the world" S
  - Perform the task T on this copy of S
  - At the end of T, try to integrate the changes back into S
    - In case of conflicts, rollback and retry later
    - Otherwise, commit the changes

- To work properly, T should have **no irreversible side effect** and act exclusively on S.
  - No IO operations (logs, database, network, display, etc.)
  - No self-destruction sequence, etc...

Software transactional memory

- STM distinguishes:
  - **STM blocks** which hold the logic
  - From the **atomic transaction** executing them.

- The API of our concurrent bank account will provide two STM methods:
  - GetBalance
  - Deposit

- A **transfer** involves two Deposit calls (one for each account).

  A **global observation** involves one GetBalance call per account.

Example: a thread-safe account

- Composing STM methods is elementary:

```
deposit :: Account -> Int -> STM ()
getBalance :: Account -> STM Int

-- | Atomic transfer between two accounts
transfer :: Int -> Account -> Account -> IO ()
transfer amount source destination =
   atomically $ do
      deposit source (-amount)
      deposit destination amount

-- | Atomic observation of several accounts
observe :: [Account] -> IO [Int]
observe accs = atomically $ mapM getBalance accs
```

Example: a thread-safe account

- Test description:
  - 4 accounts, 8 threads, 5000 transfers per threads
  - 1 observer thread, doing 5000 observations of the 4 accounts, and counting the consistency errors

- Results for **monitor-based** account (4 cores):
  - No atomicity: 0.352s but 10322 inconsistencies!
  - With try locks: 3.138s (varies a lot)
  - With ordered locks: 0.767s

- Results for **STM based** account (4 cores):
  - Synchronous STM: 0.019s
  - STM queue: 0.106s

Software transactional memory

- With STM:
  - **Encapsulation** is preserved (no global order needed)
  - **Composition** is possible inside atomic transactions
  - **No code bloat** thanks to the keyword "atomically"
  - **Performance** is pretty damn good

- There must be a catch...
  - **Synchronous STM** performance decreases with the size of the transaction.
  - Prefer **STM queues** when dealing with computational-intensive transactions.
  - STM application scope is limited: transactions cannot have any side effects besides memory updates (no IO actions, etc...).

Conclusion

- We introduced two synchronization methods to ensure data consistency:
  - **Synchronized queues**
  - **Software Transactional Memory**

- These primitives are available in numerous languages, even sometimes as built-in functions.

- Regarding the performance of each approach, although we gave some guidelines here, do not forget the essential rules:
  - Do not trust guidelines too much.
  - Try out and always, **always measure.**

Sources

- Documentation
  - “Beautiful concurrency” by Simon Peyton Jones
  - “Parallel and Concurrent Programming in Haskell” by Simon Marlow
  - “Atomic<> weapons” by Herb Sutter
  - “http://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html”
- Source code examples
  - https://github.com/QuentinDuval/HConcurrentLogger
  - https://github.com/QuentinDuval/ConcurrencyAccount

Thank you

Any questions?