Advanced Programming

Concurrent programming

Fritz Henglein

Datalogisk Institut, Københavns Universitet (DIKU)

Goals

Week 4 (last week):

- monads revisited;
- free monads;
- monads for programming with side effects;
- monadic input/output;
- monadic exceptions;
- updatable references;
- free monads for factoring out implementations of computations.

Week 5 (this week):

- concurrency;
- channels and threads;
- client-server programming.

Concurrent programming: Basic concepts

- A *buffer* is a data structure for adding and removing data items.
- A *thread* is a sequential computation to be *executed (run)*.
- A *message* is a data item intended for being sent from one thread to another thread.
- A *channel* is a message buffer.
- A thread can *fork (spawn)* a new *concurrent thread*.
- A thread can *create* a new channel.
- A thread can *write (add)* a message to a channel.
- A thread can *read (remove)* a message from a channel.

Concurrent programming: Basic concepts

- Message buffers: Many different types.
  - Unbounded-size FIFO message queue with atomic read and write. (Haskell: `Chan a`.)
  - Unbounded-size FIFO message queue with composable atomic access operations. (Haskell: `TChan a` and `TQueue a`.)
  - Bounded-size FIFO message queue; e.g. Unix pipes. (Haskell: `TBQueue a` and `BChan a`.)
  - One-element buffer. (Haskell: `MVar a` and `TMVar a`.)
  - Single-assignment buffer: One-element buffer with peek instead of remove. (Haskell: `IVar a` and `Future a`.)
  - Priority queue: reads (removes) the highest-priority message.
- We study only the standard Haskell unbounded FIFO message queues in AP.

Thread states

- A thread can be in one of the following states:

Running:	Currently executing.
Runnable:	Not running, but ready to run.
Blocked:	Blocked on a blocking operation.
Finished:	Terminated (no operations left to execute).

Processes and threads

- A *process* is a unit of computation and private storage to be executed.
- For each process the operating or run-time system maintains a *thread set*, a set of running, runnable and blocked threads.
  - Initially, the thread set contains a single runnable thread, the *entry point* of the process.
  - When a running thread spawns (forked) a new thread, the new thread is added to the thread set as a runnable thread.
  - When a running thread terminates normally, it is removed from the thread set.
  - When a running thread *blocks*, its status changes to blocked, and the scheduler changes the state of a runnable thread to running.
- A process *terminates normally* when the thread set becomes empty.
- A process *deadlocks* (is deadlocked) when the thread set is nonempty, but all threads are permanently blocked (cannot become unblocked).

Blocking and nonblocking operations

- An operation is either *blocking* or *nonblocking*.
- It is *nonblocking* if will unconditionally complete execution; e.g. adding two numbers.
- It is *blocking* if its completion depends on a condition that does not (yet) hold; e.g. a nonempty message queue (channel).
  - The thread containing it is stopped with status "blocked".
  - The blocked operation is *unblocked* when the condition holds.
  - The thread containing changes status from "blocked" to runnable.
- *Single-threaded programming*: There is always just one thread. The process stops on a blocking operation until it becomes unblocked.
- *Multi-threaded programming*: There are multiple threads. Switch to another runnable thread when the running thread becomes blocked.
  - Use cases: I/O (especially reads), pipe-based programming (programming with finite-sized FIFO message buffers).
  - Nonuse cases: Code with no blocking operations.
    - Reasoning about (correctness of) concurrent thread executions is *very hard*. Don't use multi-threaded programming unless you *need* it.

Channel-based programming in Haskell

```haskell
import Control.Concurrent
  ( Chan,         -- :: Type -> Type
                  -- FIFO message queues (channels)
    ThreadId,     -- :: Type
                  -- thread ids
    forkIO,       -- :: IO () -> IO ThreadId
                  -- nonblocking fork of new thread
    newChan,      -- :: IO (Chan a)
                  -- generate new message queue with elements of type a
    readChan,     -- :: Chan a -> IO a
                  -- read from message queue, blocks on empty queue
    writeChan,    -- :: Chan a -> a -> IO ()
                  -- nonblocking write to message queue
    killThread,   -- :: ThreadId -> IO ()
                  -- kill thread with given thread id
    threadDelay   -- :: Int -> IO ()
                  -- delay thread for given number of microseconds
  )
```

Example: Concurrent logging

```
0: logger3 = do
1:   c <- newChan
     let readPrintLoop = do
2:         r <- readChan c
3:         putStr r
4:         readPrintLoop
5:   forkIO readPrintLoop
6:   writeChan c "1"
7:   writeChan c "2"
8:   writeChan c "3"
```

Example: Concurrent logging

- Execution of `logger3` process:

```
Channels                    Thread set             Output stream
                            t1 |-> 1
c1 |-> []                   t1 |-> 5
c1 |-> []                   t1 |-> 6, t2 |-> 2
c1 |-> ["1"]                t1 |-> 7, t2 |-> 2
c1 |-> ["1", "2"]           t1 |-> 8, t2 |-> 2
c1 |-> ["1", "2", "3"]      t2 |-> 2
c1 |-> ["2", "3"]           t2 |-> 2                1
c1 |-> ["3"]                t2 |-> 2                12
c1 |-> []                   t2 |-> 2                123
```

- Observe:
  - The thread set has at most one running thread at any time: Execution is *sequential*.
  - There is a permanently blocked thread, but not a runnable thread at the end. The process is *deadlocked*.

Concurrency versus parallelism

- *Concurrency*: Multiple threads may exist simultaneously.
  - Typical scenario: Threads trying to get access (read from/write to) to a shared resource are blocked, e.g. a file. Other threads are executed until they get unblocked.
- *Parallelism*: Executing multiple operations at the same time ("in parallel").
  - SIMD: Same instruction applied to multiple data elements at the same time, e.g. adding +1 to all elements of a sequence.
  - MIMD: Different instructions applied to multiple data elements at the same time.
- Concurrent computation is a priori *sequential*.
  - At any given point at most one thread is running.
  - The *scheduler* switches between runnable threads.
- Concurrent threads may be *implemented* using MIMD parallelism: Multiple runnable threads execute in parallel.
  - Parallel execution *should* preserve sequential semantics to ensure program is hardware independent.
  - But often it doesn't. Programmer needs to craft code carefully.

Summary

- Concurrent programming: Programming with threads.
  - Channel-based concurrent programming: Use message buffers for communication between threads.
- Concurrency versus parallelism: Different concepts.