Implementing Symmetric Multiprocessing in LispWorks

Making a multithreaded application more multithreaded

Martin Simmons, LispWorks Ltd

Outline

- Introduction
- Changes in LispWorks
- Application requirements
- Future work

Why SMP?

- Demand from customers

  and also
- Makes better use of modern hardware
- Multi-core hardware readily available
- Fun!

Roadmap

Multiprocessing models in LispWorks

Green threads	Native threads	SMP
1987	1997	2009
LispWorks 2 & 3	LispWorks 4 & 5	LispWorks 6
Lisp scheduler implements threads	Lisp scheduler chooses thread for native scheduler	Native scheduler guided by Lisp scheduler

Changes in LispWorks

- Runtime system changes
  - Change some global data to be per-thread
    - Bindings, catch tags, current thread
  - Compiler changes to access to per-thread data
  - Garbage collector (addition of locking)
  - FLI (removal of locking)
- Common Lisp implementation
- Extensions and libraries

Interaction of CL with SMP

- No formal specification for threads in CL
  - Some consensus between implementations
- Thread-safety
- Atomicity
- Specify some semantics
  - Goal is to remain the same as existing threading model

What does thread-safe mean?

- Safety in the implementation
  - Avoids breaking the implementation
  - Implicit locks
- Safety for applications
  - We need to specify some semantics that can be guaranteed

Safety in the CL implementation

- Access to all standard CL objects is thread-safe
  - Readers always return valid CL objects
  - Does not imply useful semantics overall
- Immutable objects
  - Numbers, characters, functions, pathnames and restarts
  - Can be freely shared between threads
- Mutable objects
  - Use with more than one thread needs to be controlled
  - Atomic access possible in some cases

Atomic access

- Scenario:
  - There is **one** object
  - Several threads are reading and writing **one** of its slots
- The value of each read operation looks like
  - Some write operations have finished
  - But all other write operations have not started yet
- Not specified for multiple reads
  - Same slot or different slots

Mutable objects: atomic access

- Access to conses, simple arrays, symbols and structures is atomic.
  - Does **not** apply to non-simple arrays (compound objects)
- Slot access in objects of type standard-object is atomic with respect to
  - modification of the slot
  - class redefinition, but MOP semantics are problematic
- vector-pop, vector-push, vector-push-extend, (setf fill-pointer) and adjust-array
  - atomic with respect to each other and with respect to other access to the array elements
- Hash tables operations are atomic with respect to each other
  - Making several calls to these functions will not be atomic overall
  - New: modify-hash to atomically read and write an entry and with-hash-table-locked for more complex operations
- Access to packages is atomic
  - Though some scenarios are nonsensical

Mutable objects: non-atomic access

- Access to lists (including alists and plists) is not atomic
  - Lists are made of multiple cons objects, so although access to the individual conses is atomic, the same does not hold for the list as a whole
- Sequence operations that access multiple elements are not atomic
  - E.g. delete, find
- Macros that expand to multiple read/write operations are not atomic
  - push, incf, rotatef etc
  - Atomic versions of some of these are available in LispWorks 6
- Stream operations are in general not atomic
  - Optional locking of streams at application granularity

New atomic operators

- Usable with a restricted set of Common Lisp *places*
- Primitives
  - atomic-exchange
  - compare-and-swap
  - atomic-fixnum-incf
- High level
  - atomic-push
  - atomic-pop
  - atomic-incf

Synchronization Objects

- Locks
  - Simple and exclusive/sharing
- Mailboxes
  - FIFO queues, use for communication between threads
- Barriers
  - Wait until fixed number of threads have synchronized
- Condition variables
  - Used with a lock for a complex Lisp condition to control the scheduler
- Counting semaphores
  - Traditional API to control number of concurrent uses of a resource

Native scheduler vs. Lisp scheduler

- Native scheduler uses synchronization objects
- Lisp scheduler uses an arbitrary predicate to control wake-up
- Syntax

  ```
  process-wait reason predicate &rest args
  ```

- process-wait is still supported
  - Using synchronization objects is usually better
  - process-wait has some problems

Problems with process-wait

- It is unspecified which thread calls the predicate
  - The dynamic environment is also unknown
- Thread-safety in the predicate is often assumed
- Lisp scheduler wake-up vs. native wake-up (timeout)
- Lifetime of the predicate
  - May have dynamic extent data in the predicate
  - But that will become invalid if native wake-up occurs
- Error handling and debugging is difficult
- Very easy for the scheduler to become a bottleneck

An alternative process-wait

- Retain the convenience of process-wait
  - Distribute the work of the Lisp scheduler
  - Same syntax and still has a Lisp predicate
- Comparison to process-wait
  - The waiting thread calls the predicate when needed
  - Call is triggered by calling **`process-poke`** ***`process`***
  - Or it can be called periodically
  - Predicate lifetime and environment is well defined
  - Errors and debugging no longer a problem

An alternative process-wait (cont)

- Working name is process-wait-local
- Syntax

  `process-wait-local` ***`reason predicate`*** **`&rest`** ***`args`***

- We don't like the name
  - Can you suggest a better one?
- Could instead rename process-wait as process-wait-using-scheduler
  - Not quite correct for backward compatibility

Native GUI threading

- Used by the LispWorks IDE and CAPI applications
- Windows
  - Threading is built-in
  - Per thread event processing
- GTK+
  - Threading via a global lock
  - Per thread event processing can be simulated
- Cocoa
  - One GUI thread
  - No good way to simulate per thread events

Changes for applications

- Remove use of macros like without-preemption etc
  - Works as an all-powerful lock, stopping the world
  - Avoid like the (plague) swine flu
  - Cannot be mixed reliably with other locks
- Use other threading primitives like atomic-push
- Atomic read-modify-write primitives like compare-and-swap
- More use of locks
  - need a design to avoid deadlocks
  - use sparingly to avoid contention
- Try to use process-wait-local rather than process-wait
- Use other synchronization objects

An application: the LispWorks IDE

- Already multithreaded
- Many changes to the editor
  - Original design was single threaded
  - Many types of interacting objects
    - Buffer, window etc
  - Programmatic and interactive
  - Streams

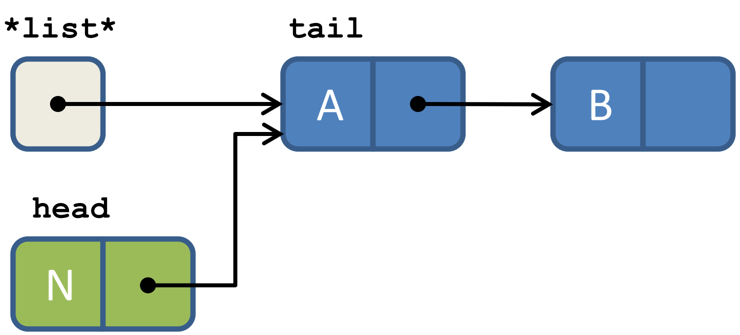
Common conversion pitfalls

- Overuse of locks
- Deadlocks
- Avoiding locks by sleepy waiting or busy waiting
- Misuse of new atomic operations

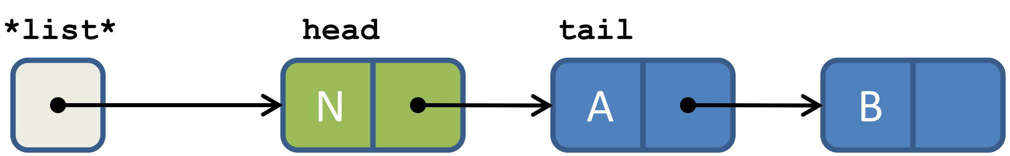
Entertaining bug

- Goal is a pop/push resource for conses
- Atomic push:

```
(atomic-push N *list*)
```



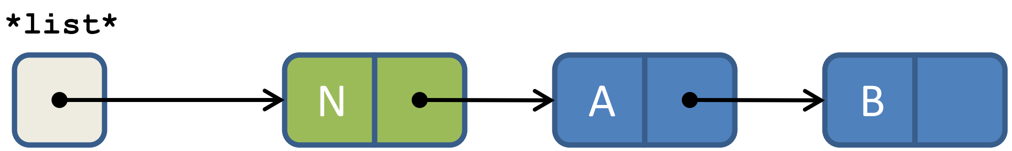

```
(loop
 (let* ((tail *list*)
        (head (cons N tail)))
   (when (compare-and-swap
          *list* tail head)
     (return))))
```

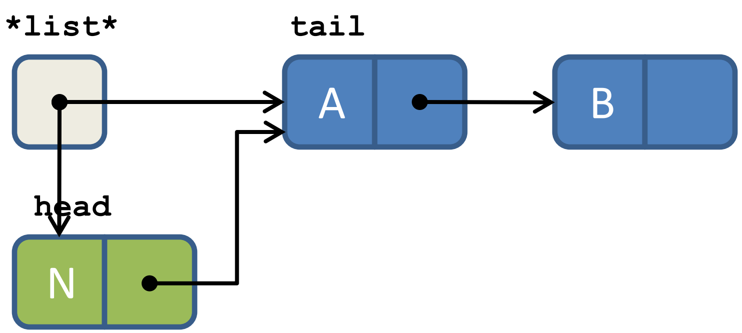


Entertaining bug (cont)

- Atomic pop:

```
(atomic-pop *list*) => N
```





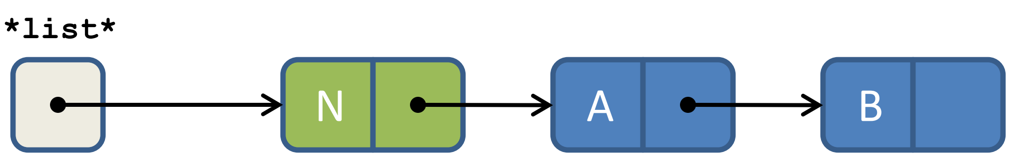

```
(loop
 (let* ((head *list*)
        (tail (cdr head)))
   (when (compare-and-swap
          *list* head tail)
     (return (car head)))))
```

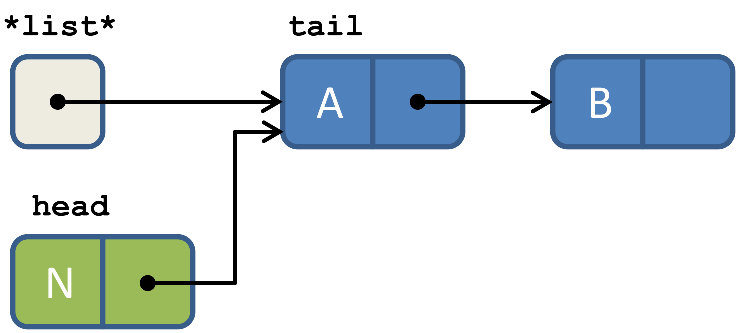
- Can we reuse the cons?

LispWorks LW www.lispworks.com

Entertaining bug (cont)

- Atomic pop cons: `(atomic-pop-cons *list*) => (N)`



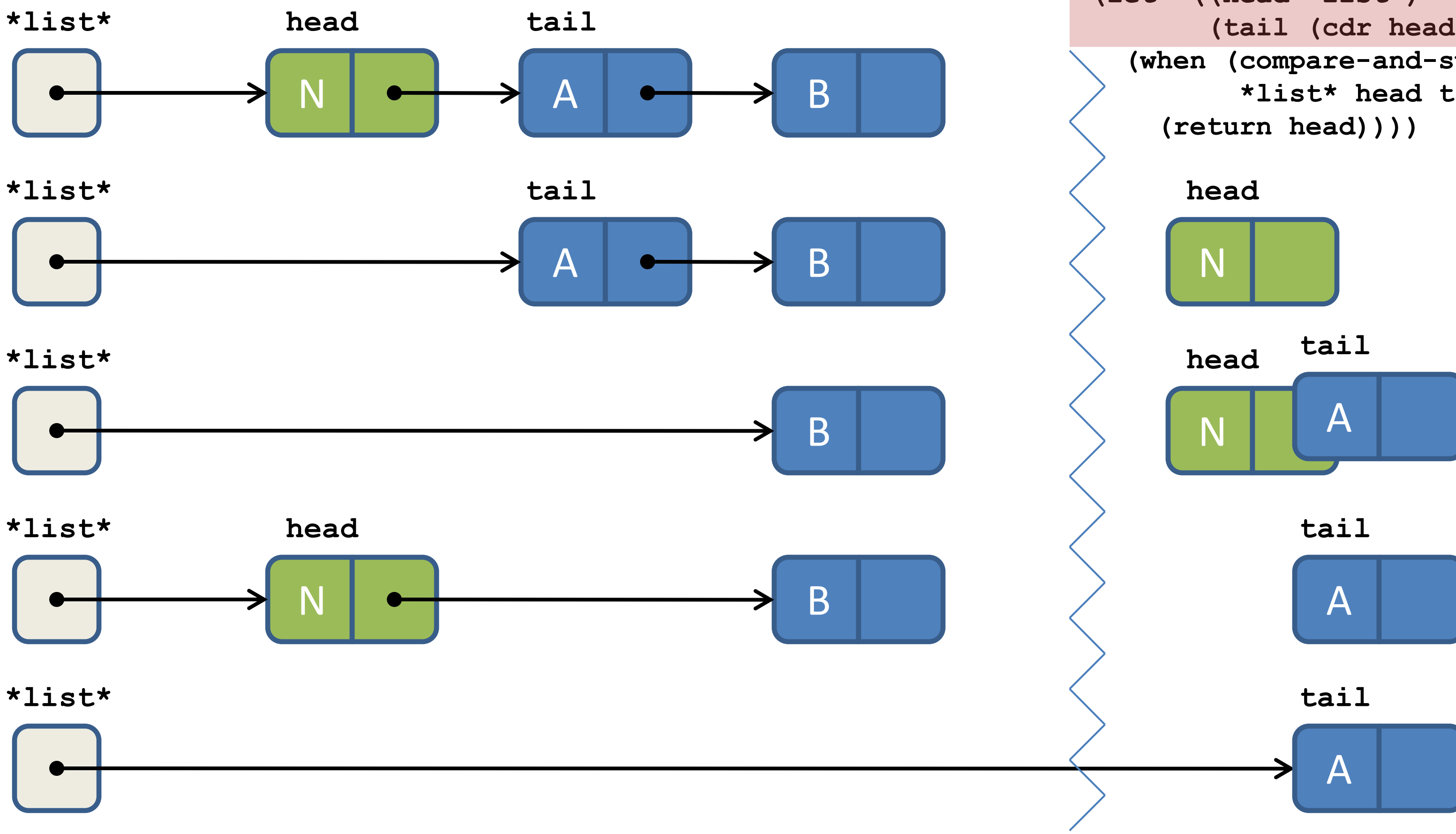



```
(loop
 (let* ((head *list*)
        (tail (cdr head)))
   (when (compare-and-swap
           *list* head tail)
     (return head))))
```

- Atomic push cons is similar

Entertaining bug (cont)

```
(loop
 (let* ((head *list*)
        (tail (cdr head)))
   (when (compare-and-swap
          *list* head tail)
     (return head))))
```



Most MP code can be ported easily

- Watch for code that was never thread-safe
  - Much more likely to break in a SMP Lisp
- Customers should contact us for advice

What comes next

- LispWorks 6 beta
- Possible future work
  - Multithreaded GC?
  - Other threading primitives?
  - Other paradigms such as transactional approaches

Summary

- Changes in LispWorks
  – New atomic access model
  – New primitives
  – Performance comparable to current stable release
- Application changes
  – Limited to interaction with threads
- Available in LispWorks 6