A Scheme for native threads

R. Kent Dybvig
Indiana University
Mitchfest / August 2009

What are Native Threads?

Each thread is a separate O/S thread

Each may run on own processor or core

All threads share same heap, file descriptors, etc.

Why Native Threads?

Express multithreaded computations

Handle blocking operations

Take advantage of multiple processors and cores

Interoperate with existing threaded applications

Outline

Features / Examples

Thread safety

Suspending and killing threads

Implementation

Conclusion

Model

Based on Posix Threads

Fork threads dynamically

Use mutexes for mutual exclusion

Use conditions to avoid polling

Thread Creation

(fork-thread *thunk*)

Mutexes

`(make-mutex)`

`(mutex-acquire` *mutex*`)`

`(mutex-release` *mutex*`)`

`(with-mutex` *expr*$_m$ *body*`)`

Conditions

`(make-condition)`

`(condition-wait` *condition mutex*`)`

`(condition-signal` *condition*`)`

`(condition-broadcast` *condition*`)`

Example

```
(let ([m (make-mutex)] [c (make-condition)])
  (with-mutex m
    (fork-thread
      (lambda ()
        (with-mutex m
          (display "hello\n"))
          (condition-signal c)))
    (condition-wait c m)
    (display "goodbye\n")))
```

prints:

```
hello
goodbye
```

Example: pcall

```
(define-syntax pcall
  (syntax-rules ()
    [(_ e0 e1 ...)
     (let ([m (make-mutex)]
           [c (make-condition)]
           [ls (list (lambda () e0) (lambda () e1) ...)]
           [n (length '(e0 e1 ...))])
       (with-mutex m
         (do ([ls ls (cdr ls)])
             ((null? ls))
           (fork-thread
             (lambda ()
               (set-car! ls ((car ls)))
               (with-mutex m
                 (set! n (- n 1))
                 (when (= n 0) (condition-signal c))))))
         (condition-wait c m))
       (apply (car ls) (cdr ls)))]))
```

Thread Activation

Thread activated by `make-thread`

Thread can also be activated in C

Should deactivate thread before blocking

Deactivation implicit for I/O operations

Thread Parameters

Thread parameters $\equiv$ thread-local storage

```
(make-thread-parameter value)
(make-thread-parameter value filter)
```

```
(parameterize ([expr_p expr_v] ...) body)
```

Setting/parameterizing affects only current thread

Child inherits parameter *values*, not locations

Can still create global parameters:

```
(make-parameter value)
(make-parameter value filter)
```

Thread Parameters

Example thread parameters:

- `current-input-port`
- `current-output-port`
- printer controls
- compiler controls

Example global parameters:

- `current-directory`
- `command-line`
- collector controls

Continuations

Continuations are per-thread

Can invoke continuation created in another thread

Should not be used to exit from thread

Thread Safety

Most primitives are thread-safe . . .

- `car`, `cons`, `+`, `=`, `list-sort`, `map`, etc.

. . . including many destructive operations

- `set-car!`, `vector-set!`

Thread Safety

Some primitives are “relatively” unsafe . . .

- `put-char`, `get-char`
- `hashtable-set!`, `hashtable-ref`

. . . relative, that is, to a given object

- okay to read/write different ports/hashtables concurrently

Improper synchronization is bad:

- lost data
- corrupted heap

Caveat emptor

Thread Safety

Remainder are “safety preserving”

- `apply`, `map`, `for-each`
- `eval`, `compile-file` (!)

Suspendable Threads

Threads cannot be suspended, interrupted, or killed

Justification:

- asynchronous interrupts $\equiv$ indeterminate state
- keep the mechanism simple

Suspendable Threads

Threads cannot be suspended, interrupted, or killed

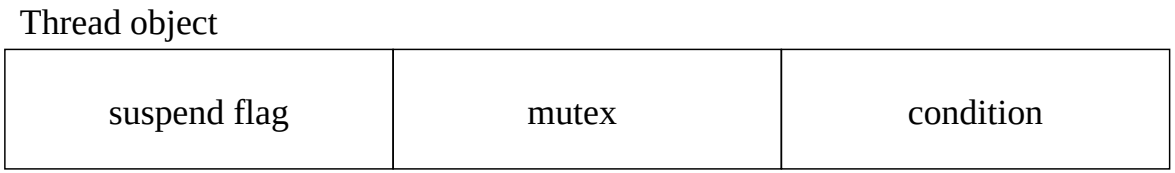
Solution: ask thread politely

- please suspend yourself
- please interrupt yourself
- please die

But, will it listen?

Suspendable Threads

Represent “suspendable thread” as a record



Set suspend flag `#t` to request suspension

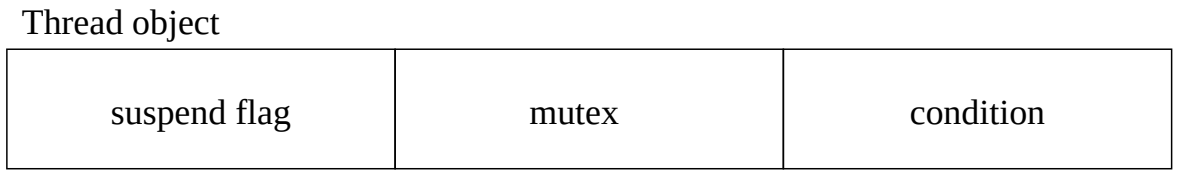
Thread polls flag via lightweight threads, timer interrupts, etc.

- if suspension requested, waits on condition

Set flag `#f` and signal condition to resume

Suspendable Threads

```
(define-record-type sthread
  (fields (mutable suspend)
          (immutable mutex)
          (immutable cond))
  (protocol
    (lambda (new)
      (lambda ()
        (new #f (make-mutex) (make-condition))))))
```



Suspendable Threads

```
(define fork-sthread
  (lambda (thunk)
    (define ticks 1000)
    (let ([sthr (make-sthread)])
      (fork-thread
        (lambda ()
          (timer-interrupt-handler
            (lambda ()
              (with-mutex (sthread-mutex sthr)
                (when (sthread-suspend sthr)
                  (condition-wait
                    (sthread-cond sthr)
                    (sthread-mutex sthr))))
              (set-timer ticks)))
          (set-timer ticks)
          (thunk)))
      sthr)))
```

Suspendable Threads

```
(define suspend-sthread
  (lambda (sthr)
    (with-mutex (sthread-mutex sthr)
      (set-sthread-suspend! sthr #t))))

(define restart-sthread
  (lambda (sthr)
    (with-mutex (sthread-mutex sthr)
      (set-sthread-suspend! sthr #f)
      (condition-signal (sthread-cond sthr)))))
```

Thread Implementation

Thread context

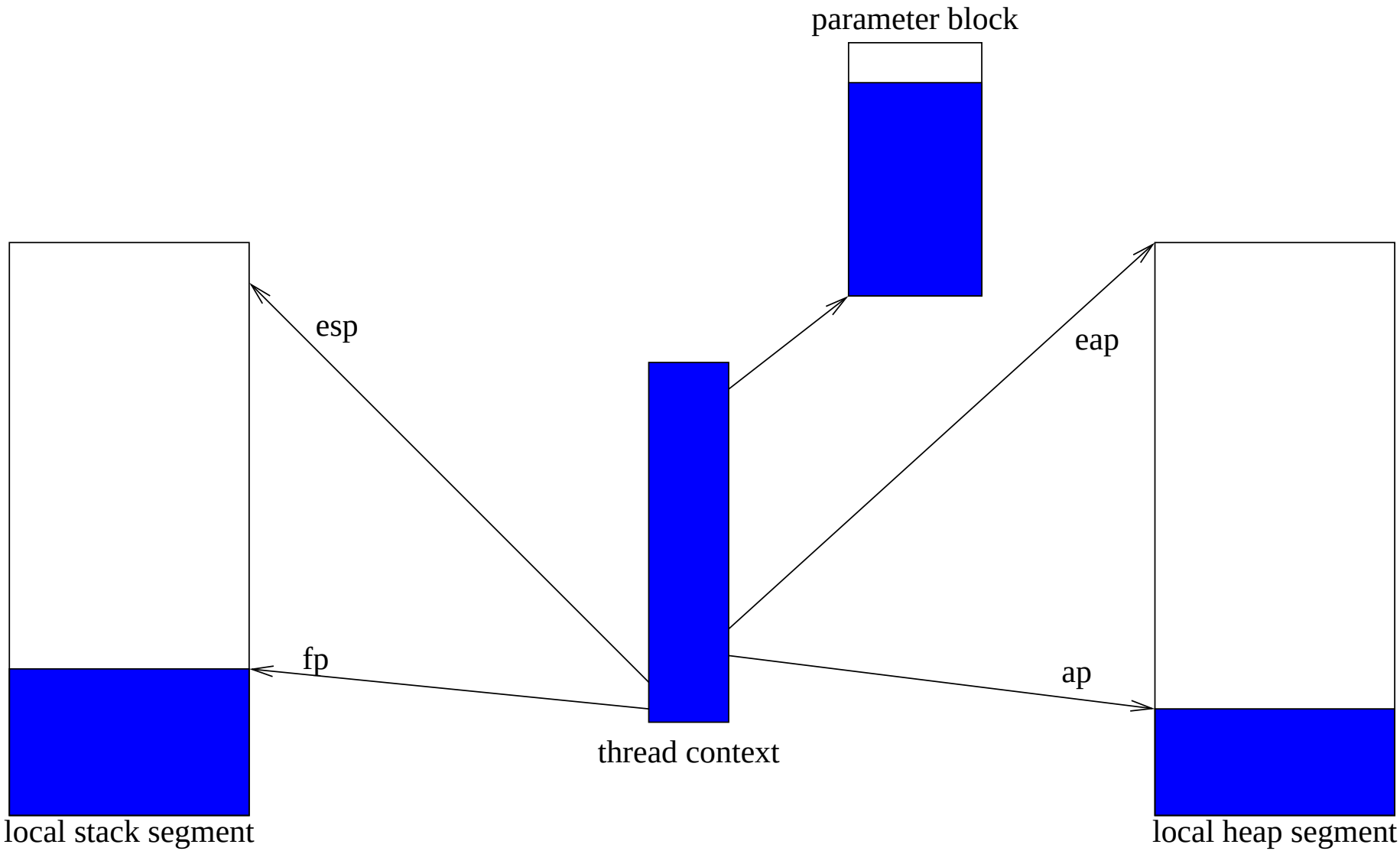
Stack management

Heap management

Parameters

Challenges

Thread Implementation



Garbage Collection

Collection is presently single-threaded

All active threads must rendezvous

Last thread collects, then releases others . . .

. . . or before deactivating, last thread wakes proxy

Challenges

Surprisingly few . . .

- segmented stack, heap essential
- mostly stateless compiler helped

. . . but there were some:

- setting dirty bits
- finding uses of state
- dealing with state
- bugs

Dealing with State

Some state is inherently per-thread

- e.g., compiler's variable records
- no worries

Other state can be made per-thread

- e.g., random seed, bignum temps
- embed in thread context . . .
- . . . or make into thread parameters

Other state requires synchronization

- e.g., system symbol table
- try to minimize synchronization window

The Bug from Hell

Some bad bugs:

- thread bugs (nonreproducible)
- collector bugs (indirect effects)
- multiple bugs (conflicting effects)

Worst case senario:

- multiple thread bugs involving the collector

Required eight (!) days to track

Dazzit!

Specialized database engine

- serves queries through web interface
- run-time code provided as shared library
- adaptors for Apache, FastCGI, IIS, .NET, Java
- each supports native threads
- Linux, Mac, Windows, Solaris

To reduce synchronization overhead:

- shared data is read-only
- modifications in per-thread caches

Interesting features:

- each query is compiled on the fly
- html generated to communicate results

Future Work

Safer ports and hashtables

Parallel garbage collection

Parallel garbage collection

Lighter weight synchronization

Higher-level abstractions

Conclusions

Not for the faint of heart

A good starting point is helpful

- segmented stacks and heaps
- few uses of global state

Posix Threads also helpful

- reduced implementation time
- simplified portability
- worth additional overhead

Acknowledments

Oscar Waddell

Al Reich, Robert Boone at Motorola SPS

Michael Lenaghan at Dazzit!