Grand Central Dispatch
FreeBSD Devsummit

Robert Watson rwatson@FreeBSD.org

18-Sep-2009

libdispatch

- runtime library for Grand Central Dispatch feature of Mac OS X version 10.6 Snow Leopard
- available as open source
  - http://libdispatch.macosforge.org
  - Apache License, Version 2.0
- **also runs on FreeBSD 9-CURRENT**

libdispatch

- used in combination with sibling technology “blocks”
  - closures for C
  - not required for libdispatch but very useful
- compiler and runtime also available as open source under BSD-style license
  - http://clang.llvm.org
  - http://compiler-rt.llvm.org

hello world

```
#include <dispatch/dispatch.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
   dispatch_queue_t queue = dispatch_queue_get_main();

   dispatch_async(queue, ^{
      printf(“%s: Hello World\n”, argv[0]);
   });

   // begin main event loop
   dispatch_main();
}
```

topics

- goals
- data-types (objects, queues, event sources)
- examples
- conclusion

goals

- convenience
- efficiency
- task-level parallelism
- synchronization
- scaleability

convenience

- provide a callback-based event loop
- unify “event code” with “application code”
  - historically, these were separate domains
    - select, poll, kqueue, etc.
    - sem_wait, pthread_cond_wait
  - processes must bridge application events to an OS primitive (e.g. pipe) to wake event loop

efficiency

- perform as much as possible in user space
  - use of pipes in previous example requires context switching to kernel space for send and receive; consumes finite resource (fd table)
  - libdispatch prefers to use a small number of atomic operations for inter-thread communication

task-level parallelism

- individual event loops expressed as queues
- multiple queues may be processed in parallel
- mapping logical subsystems onto separate queues allows implicit task-level parallelism

synchronization

- queues execute events in FIFO order and wait for completion
- can be used to protect access to shared resources

“Islands of serialization in a sea of concurrency.”

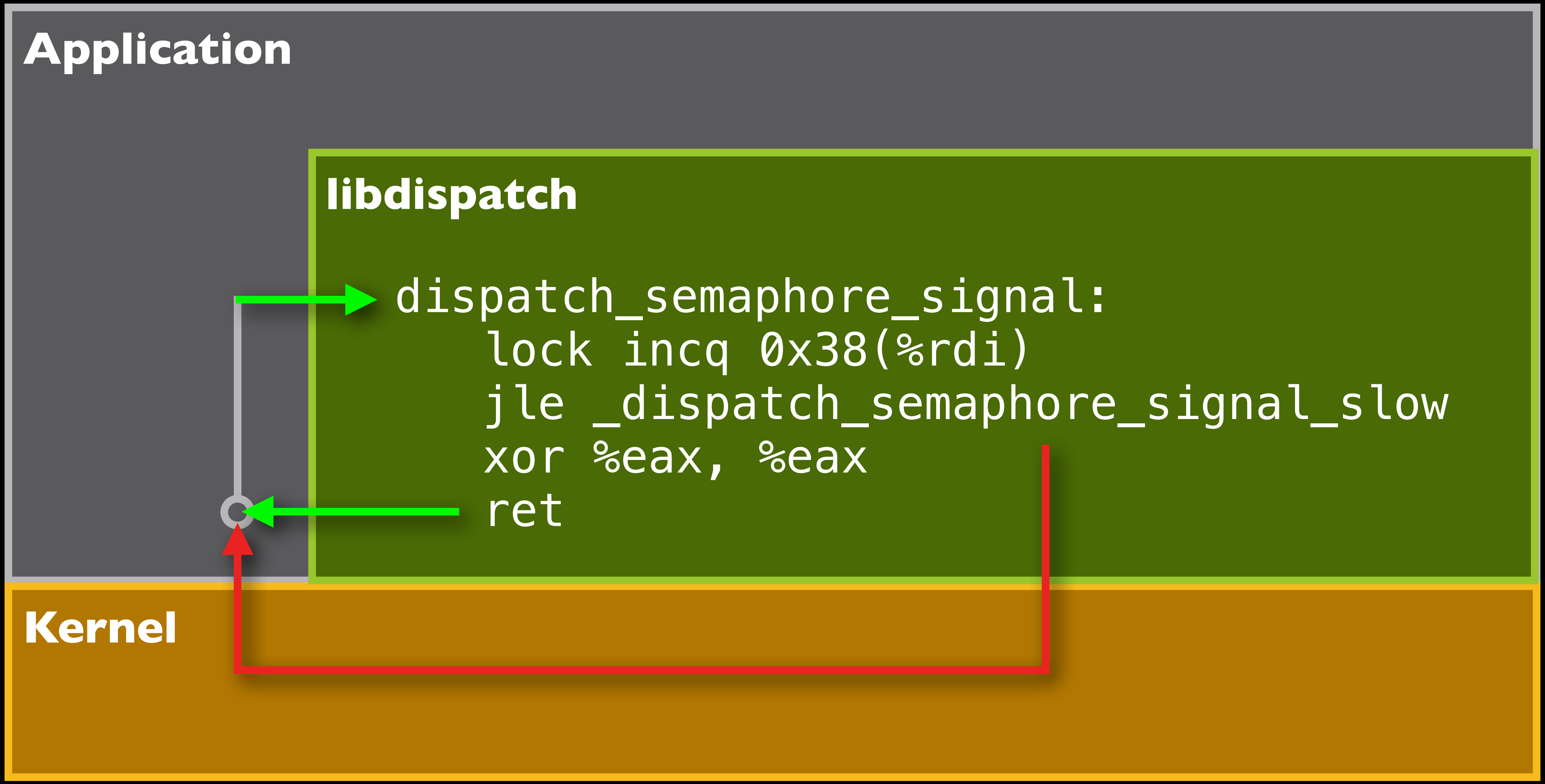
scaleability

- peak performance under load
- “pump-priming” costs payed in transition from idle to busy
  - ex., thread creation is a side-effect of first enqueue operation
- once busy, additional work can be taken on efficiently
  - ex., subsequent enqueue operations require only a single atomic instruction

fast-path / slow-path

- the implementation is organized into fast-paths and slow-paths
  - slow-path represents the idle→busy transition
    - allocations and kernel traps may be required
  - fast-path represents the busy→busy transition
    - implemented with the fewest atomic operations possible

fast-path / slow-path



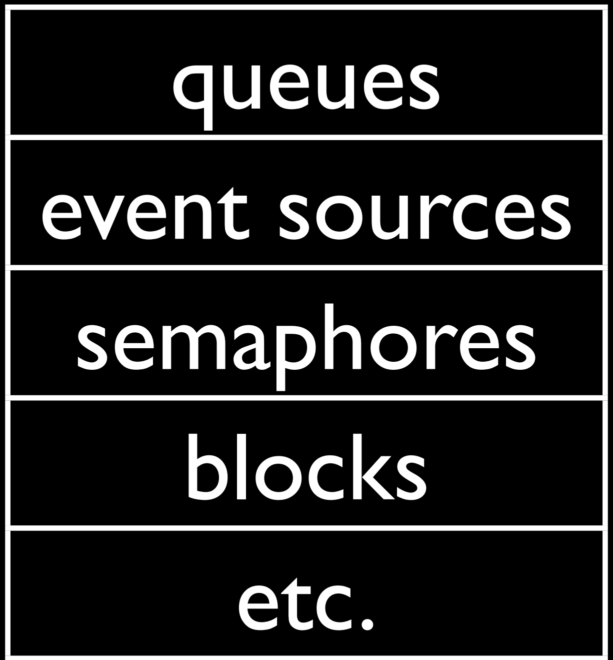
```
dispatch_semaphore_signal:
    lock incq 0x38(%rdi)
    jle _dispatch_semaphore_signal_slow
    xor %eax, %eax
    ret
```

fast-path / slow-path

- taking the fast-path assumes the object is already enqueued for further processing
- taking the slow-path means a “wake up” must be performed on the object to continue processing (more on that later)

data types

- libdispatch data types are known as dispatch objects
  - reference counted memory management
  - polymorphic (especially in internal usage)


queues

event sources

semaphores

blocks

etc.

dispatch objects

- each dispatch object is a reference-counted linked-list element that may be enqueued
- includes queues of queues!

dispatch object
vtable
reference count
next pointer
target queue
…

dispatch queues

- dispatch queues are FIFO queues of dispatch objects (including other dispatch queues) waiting to be “invoked”
- queues may be processed on separate threads; invoke elements in FIFO order
- wait-free atomic enqueue from any thread

dispatch queues

dispatch queue
vtable
reference count
next pointer
target queue
…
head pointer
tail pointer

dispatch_async

- dispatch_async is the workhorse of libdispatch
- schedule objects to be invoked
- wait-free atomic enqueue algorithm in pseudocode:

```
prev = atomic_xchg(queue->tail, obj);
if (prev) {       // fast-path
   prev->next = obj;
} else {          // slow-path
   queue->head = obj;
   wakeup(queue);
}
```

wake up

- “wake up” function schedules an object for further processing:
  - typically enqueues the object on its target queue
  - alternatively performs an action specific to the object type

invoke

- dispatch object vtable entry to process an object

queues	recursively invoke elements in FIFO order
event sources	invoke event handler; register/unregister with manager queue
blocks	invoke the block

pthread_workqueue

- pthread_workqueue interfaces are an extension to pthreads on Darwin
- libdispatch registers *interest* in additional threads via pthread_workqueue_additem_np
- xnu creates threads based on feedback from scheduler
  - generally N threads for N available CPUs
  - additional threads may be provided if existing threads are blocked in I/O, on a mutex, etc.

pthread_workqueue

- threads created by xnu run a libdispatch-provided function with a libdispatch provided context
- grabs temporary ownership of a global concurrent queue, pops first element, and begins recursive invocation
- **FreeBSD note: not implemented**

event sources

- user-defined event handler blocks submitted to a target queue in response to system activity
- follows kqueue conceptual model of events
  - register for specific event types
  - events described by type and a word of data
    - counter or bit-mask of flags
    - data may be coalesced

event types

event type	event data
fd state	readable, writeable
vnode attributes	rename, delete, etc.
process events	fork, exec, exist, etx.
~~mach port~~	~~readable, dead name, etc.~~
signal	SIGHUP, etc.
timer	500ms, etc.
custom	user-defined

event coalescing

event type	coalescing	interpretation
fd state	XCHG	bytes available
vnode attributes	OR	flags
process events	OR	flags
mach port	OR	flags
signal	ADD	count (# times)
timer	ADD	count (# times)
custom	ADD / OR	user-defined

event reflector

- event reflector thread ("dispatch manager queue") monitors a kqueue
  - dequeues events in bulk
  - atomically merges data into event source object
  - performs "wake up" on event source — enqueues event source object on target queue for invocation of event handler block

event reflector

- adding/removing event sources done by enqueuing event source object to manager queue
- manager queue has custom wake up routine
  - uses EVFILT_USER / NOTE_TRIGGER

event performance

- no allocations required after event source creation
    - event source object contains the queue linkage
- data is coalesced while event source object is pending on a queue

examples

- Running a task in the background and returning the result to the main event loop (main thread)

```
dispatch_async(my_queue, ^{
  result = my_task(my_data);
  free(my_data);
  dispatch_async(dispatch_get_main_queue(), ^{
    show_result(result);
    free(result);
  });
});
```

examples

- creating a file descriptor read source

```
FILE *file = fopen(“/usr/share/dict/words”, “r”);

source = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
   fileno(file), 0, my_queue);

dispatch_source_set_event_handler(source, ^{
   my_read(file);
   if (feof(file) || ferror(file)) {
     dispatch_source_cancel(source);
   }
});

dispatch_source_set_cancel_handler(source, ^{
   fclose(file);
});

dispatch_resume(source);
```

the FreeBSD port

- port to FreeBSD relatively straight forward
    - autoconf/automake/libtool (pain)
    - ifdef Mach port event handling
    - Mach semaphores ➙ POSIX semaphores
    - Mach time ➙ POSIX time/timespec
    - new kqueue features: EVFILT_USER, EV_DISPATCH, EV_RECEIPT
    - thread pool rather than pthread workqueues

FreeBSD questions

- we already support this technology out of the box in 9.x, soon 7.x / 8.x
    - do we want to implement pthread_workqueues?
    - should we explore OS-specific optimizations
    - what is required to get C Blocks working -- libgcc, compiler-rt, etc
    - should we consider adopting this technology for our own OS components

conclusion

- libdispatch is a significant departure from threaded programming model
- offers simplicity and convenience for "multithreaded" event loops
- efficient and scaleable design
- available as open source under Apache 2.0 license to encourage widespread adoption