10. Performance Tuning

The End of Free Performance

- Moore's law
- multicore chips and hyperthreaded architectures (with own logic units)
- hyperthread - two or more threads operating in parallel in a single CPU
- parallel programming is generally for experts but it will gradually become more in demand
- hardware design affects software developers due to these issues
- writing efficient code is very important

Approaches to Performance

- use profiling tools (Instruments, Shark, others) early and often
- be aware of changes across architectures and os versions and profile on each
- do not optimize too early though
- be aware of differences in development build and release build and do not waste time optimizing a development build when the release build may have different machine code and code paths, etc

Major Causes of Performance Problems

- generally come from one or more of:
    - algorithms
    - memory
    - CPU
    - disk
    - graphics
- can use performance tools to look at each aspect separately

Memory

- even with lots of memory, RAM is still a scarce resource
- mac starts swapping memory pages to disk which will drastically effect performance
- iOS may kill a program that is a memory hog
- optimizing memory usage often increases execution performance (tough discipline)

Locality of Reference

- memory accesses that happen near each other

- processor retrieves a cache line or sequence of bytes around the requested memory
- program to operate on sequential memory as much as possible
- see locality.m here (shows examples for managing page related memory access)

Caches

- can often boost performance with caching but can cause issues when interacting with system paging because hand programmed custom caches that are not used for a long time will be swapped to disk
- iOS does not page data to disk
- prefer splitting cache data and metadata describing cache data to take advantage of locality of reference

Memory is the New I/O

- I/O from disk is extremely expensive
- RAM has become like I/O in that it is more expensive than programming patterns that take advantage of memory caches, vectorization, and other techniques
- manual programmed caching can be slower in some cases than these other techniques, even if they require jumping through hoops
- level-1 cache is only 32-64 kb per core
- certain optimizations like loop unrolling and 64-bit code in general can blow out the level-1 cache and cause slowdown from RAM accesses
- C semantics can also cause issues from this perspective
- when accessing a global data structure (especially in a local loop) an optimization is to use a local variable to reference the global data structure to facilitate compiler optimizations

CPU

- easy to discover and monitor high CPU usage in a program

Disk

- disk access is slow
- caching too much data can increase disk I/O
- bad locality of reference causes access of many different pages
- can avoid some disk I/O by not doing the work (only load what is needed including separating windows into different .nib files)
- accessing data from a database piecemeal can yield speedups

- using memory-mapped files can avoid disk activity

Graphics

- Quartz is system resource heavy
  - uses large graphic buffers for each window visible on the screen
  - compositing operations to render user's desktop
- some operations cannot be done by GPU so must be done by CPU
- to optimize avoid drawing when possible
- use Quartz Debug utility
  - Drawing Information
  - Autoflush drawing
  - displays identical screen updates and highlights drawing areas
- NSView
  - getRectsBeingDrawn
  - needsToDrawRect
- Quartz quark to note
  - overlapping lines in a single path is expensive
    - antialiasing
    - managing transparent color paints
- draw small paths instead

Before using any of the profiling tools

- programmers are notoriously bad about predicting where performance issues are actually occurring
- maintain notes of optimizations and discoveries to learn from for later
- always test optimization issues with large data sets because poorly performing algorithms will really stand out then
- when to optimize
  - choose the middle ground (not too early and not too late)
  - do not obfuscate code early in the development process

Command-Line Tools

- time
  - times command execution
  - C shell version gives more info
- dtruss
  - shows all system calls a program makes (truss, strace, ktrace, etc)
- fs_usage
  - shows filesystem related system calls
  - set the environment variable DYLD_IMAGE_SUFFIX to _debug to show Carbon calls

    - good for tracking I/O performance problems
- sc_usage
    - shows system call information
    - good for tracking I/O performance problems
- top
    - monitor all programs on the system
    - good to discover performance issues that have manifested as system-wide slowdowns and may not manifest in an easily observable program-specific manner
    - -e shows VM, network, disk activity and messaging stats
    - -d shows in delta mode (vs. cumulative mode) in 1 second intervals
    - -s set update interval

Stochastic profiling

- run program in debugger and interrupt occasionally to inspect the call stack
- look for excessive repetition
- sample
    - samples a process at 10 ms intervals and builds a snapshot of the program's activity

Precise Timing with mach_absolute_time()

- time command sometimes doesn't give enough granularity
- mach_absolute_time() reads the CPU time base register and reports the value
- this time base register serves as the basis for other time measurements in the OS

```
uint64_t mach_absolute_time (void);
```

- get the scaling of the returned values

```
kern_return_t mach_timebase_info (mach_timebase_info_t info);
// mach_timebase_info_t = pointer to
struct mach_timebase_info {
    uint32_t numer;
    uint32_t denom;
};
```

- see machtime.m here
- remember -> timing runs can be perturbed by too many variables to consider

GUI Tools

- Activity Monitor

- Instruments
  - in Finder or Xcode (or `instruments` command line tool)
  - Leaks
  - Allocations
  - Target
  - Inspection Range

More Memory: Heapshots

- Allocations instrument
  - Mark Heap -> snapshot current heap

Time Profiler

- measures CPU usage by program and optionally across entire system

Other Instruments

- Memory
  - peek inside the garbage collector, see the virtual memory consumption of a process, track shared memory creation and destruction, and detect zombies, which are over-released objects that are subsequently messaged
- Automation
  - can write test scripts against an application on a device
- Power
  - Energy Diagnostics
  - Energy Usage
  - CPU Usage
  - Display Brightness
- Graphics
  - Core Animation
  - Sampler
- OpenGLES Analyzer
- File System
  - File Activity
  - Reads/Writes
  - File Attributes
  - Directory I/O
  - Disk Montior
- System Details
  - Activity Monitor
  - Scheduling

    - System Calls
    - VM Operations
    - Dispatch
    - Network Activity
- Symbol Trace and DTrace
    - Symbol Trace (shortcut for building a DTrace instrument)

18. Multiprocessing

- OS time-slices among runnable programs (i.e. programs not blocked waiting for an event like I/O completion)

Process Scheduling

- decides which process gets CPU next
- often uses
    - process priority
    - CPU time for a process
    - recent I/O completion
- processes have a niceness (-20 - +20, 0 by default)
    - higher niceness = lower priority
    - nicer processes are less CPU time consumptive
- use `nice` command to start a process with a specific niceness
- use `renice` to change niceness of a running process
- can only increase niceness with superuser privileges
- scheduler maintains priority queue of processes
- gets process on top of queue and runs until it blocks or time slice expires then puts back on queue
- `uptime` command show average length of the run queue
- general rule of thumb is that a healthy machine has a load average at or less than twice the healthy number of processors
- unlike Linux, mac OS does not report time blocked in disk I/O

Convenience Functions

- easiest method is to use system() which passes control to a new shell (/bin/sh)

```
int system(const char *string);
```

- system return value is return value of function call
- otherwise -1 is the result and errno is set to err before call

- 127 means shell failed for some reason
- to read or write from the process, use popen

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- invokes a shell to run a command
- can specify read/write modes (but using for rw generally does not work in practice due to buffering and potential deadlock)
- best bet for read and write is to redirect to a file or create child process using two pipes, one for reading and one for writing

fork

- to create a new process in Unix, must first create a copy of an existing process which can continue with code of original program or start executing new code

```
pid_t fork (void);
```

- one of the few functions that can return twice (in parent process, returns child PID, in child process, returns 0)
- on error, no child process is created, returns -1 and sets errno
- child inherits the parent process' memory, open files, real and effective user and group IDs, current working directory, signal mask, file mode creation mask (umask), environment, resource limits, and any attached shared memory segments
- shares all of the previously mentioned data with parent until a write is made to new process (COW -> copy on write) by marking memory pages of parent process as read-only and when necessary a copy is made and each copy is marked read/write for the respective process
- system and popen call fork under the hood but use more resources than a careful use of fork would
- gotchas
  - race conditions can occur between parent and child because there are no guarantees which process will run first
  - parent and child share the same file table entry in the kernel so file offsets, etc can change when the programmer is unaware between the parent and child/vice versa
  - buffered I/O buffers in parent's address space get duplicated to child so both the parent and child could print out the same buffered data when the data is accessed
- _exit() behaves like exit() but it does not flush the file stream buffers

- good practice to have a parent sleep before exiting to ensure the child process cleans up and closes before parent exit

Parent and Child Lifetimes

- Unix guarantees that a child will always have a parent by re-parenting an orphaned child to process with PID 1 (i.e. init process in many other Unix flavors, launchd in mac OS/OS X)
- when a child exits, OS keeps exit code and system resource usage statistics
- child is allowed to die when this info is given to parent
- use wait system call to reap process

```
pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

- causes parent to collect result code and block until child exit
- use waitpid to wait for a specific process
- WNOHANG
    - do not block waiting for a child
    - return immediately if there are no exited children
- WUNTRACED
    - report job-control actions on children (like being stopped or backgrounded)
- wait3 is like wait and wait4 is like waitpid but both populate rusage struct (see /usr/include/sys/resource.h)
- use macros for more info on child status results
    - WIFEXITED()
        - returns true if the process terminated normally via a call to exit() or _exit()
    - WEXITSTATUS()
        - if WIFEXITED(status) is true, returns the low-order byte of the argument the child passed to exit() or _exit()
        - the return value from main() is used as the argument to exit()
    - WIFSIGNALED()
        - returns true if the process terminated due to receipt of a signal
    - WTERMSIG
        - if WIFSIGNALED(status) is true, returns the number of the signal that caused the termination of the process
    - WCOREDUMP
        - if WIFSIGNALED(status) is true, returns true if the termination of the process was accompanied by the creation of a core dump
    - WIFSTOPPED
        - returns true if the process has not terminated, but was just stopped, such as by job control in a shell (for example, Control-Z will suspend a process)

    - WSTOPSIG
        - if WIFSTOPPED(status) is true, returns the number of the signal that caused the process to stop
- see status.m for usage
- a child process that has _exit() -ed still has a process table entry and the wait() info but is dead (e.g. it is a zombie, displayed in parens in `ps` command)
- when child exits, sends a SIGCHLD signal to parent which is ignored by default but can set a handler and use it to set a flag indicating that a child needs to be wait() -ed on
- generally avoid this and wait() periodically or use a kqueue (ch 16)

exec

- replaces currently running process with a new one
- several variants depending on how program arguments and environment variables are specified

```
            find exe            program args                    environment
execl       specified path      null-terminate arg list         inherited
execlp      PATH search         null-terminate arg list         inherited
execle      specified path      list of args                    explicit (null-ter
execv       specified path      null-terminated string array    inherited
execvp      PATH search         null-terminated string array    inherited
execvP      search path arg     null-terminated string array    inherited
execve      specified path      null-terminated string array    explicit (null-ter
```

- p - if filename contains slash, interpret as path, otherwise use PATH for lookup
- P - a p, but uses given string as lookup path
- v - program args are an array (vector) of strings
- l - program args are a list of separate (varargs) args in exec command
- e - environment variables are an array of strings in form "VAR=val"
- (no e) - inherits environment as `environ` variable
- execve is the system call that all of the other commands are based on
- arg list, arg vector, and env vector must be terminated with a NULL ptr

```
char *envp[] = {"PATH=/usr/bin", "EDITOR=/usr/bin/vim", NULL};
char *argv[] = {"/usr/bin/true", NULL};
```

- attributes inherited from exec call
    - open files
    - process ID, parent process ID, process group ID
    - access groups, controlling terminal, resource usages
    - current working directory
    - umask, signal mask

Pipes

- open files are inherited with exec() unless close on exec is specified
- this is used to build pipelines between programs

```
int pipe(int fildes[2]);
```

- puts two fds in filedes array
  i. create pipe
  ii. fork()
  iii. child uses dup2() to move `filedes[1]` to stdout
  iv. child exec()-s a program
  v. parent reads program from `filedes[0]`
     - when child exits, fds are closed and any reads from `filedes[0]` will return EOF after pipe is drained
  vi. parent wait()-s on child
- can chain multiple parent-child programs using pipes
- see pipeline.m

fork() Gotchas

- Unix fds are inherited across fork()
- Mach ports are not
- Cocoa uses Mach ports for IPC (including IPC with window server)
- problems will occur when using Cocoa after a fork
- Cocoa and Core Foundation use threads implicitly
- only thread that calls fork() is running in the child
- all other mutexes and data structures still exist and locked mutexes may end up in an indeterminate state
- only safe functions to call in threads after fork() are exec() async-signal-safe functions (consider for GCD workqueue thread, Foundation URL system, Core Foundation, and more)
- prefer threading over multiprocessing for many mac applications due to these issues
- for single-threaded BSD-style program do not worry
- otherwise, only fork() in order to exec()

Summary

- mac is a multiprocessing platform
- can use system or popen to run pipelines in a shell
- can use fork and exec to create new child processes
- can use pipe to establish a communications channel between related processes

19. Using NSTask

- details
  - creating new processes with NSTask
  - sending data to and reading stdout and stderr with NSPipe and NSFileHandle
  - using NSProcessInfo to get program info

NSProcessInfo

- return a shared instance of NSProcessInfo for current process

```
+ (NSProcessInfo *)processInfo
```

- return a dictionary containing all the environment variables as keys and their values

```
- (NSDictionary *)environment
```

- return name of the computer upon which the program is running

```
- (NSString *)hostName
```

- return name of the program (used by the user defaults system)

```
- (NSString *)processName
```

- create a unique ID string using the host name, process ID, and a timestamp

```
- (NSString *)globallyUniqueString
```

NSTask

- used to create and control processes
- on end, posts an NSTaskDidTerminateNotification
- set attributes of new process before creating or launching
- set path to executable code to create process

```
- (void)setLaunchPath: (NSString *)path
```

- set program arguments as an array of strings

```
- (void)setArguments: (NSArray *)arguments
```

- set environment variables (uses parent's env vars if unset)

```
- (void)setEnvironment: (NSDictionary *)dict
```

- set cwd for process on startup (uses parent's cwd if unset)

```
- (void)setCurrentDirectoryPath: (NSString *)path
```

- set stdin/stdout/stderr with NSPipe or NSFileHandle object

```
- (void)setStandardInput: (id)input
- (void)setStandardOutput: (id)output
- (void)setStandardError: (id)error
```

- most commonly used methods on running process

```
// create the new process
- (void)launch

// kill the new process by sending it a SIGTERM signal
- (void)terminate

// get PID
- (int)processIdentifier

// return YES if the new process is running
- (BOOL)isRunning
```

NSFileHandle

- common in Cocoa to read entire file into an NSData or NSString or create one of the two to write to file
- to manage reads and writes incrementally, use NSFileHandle
- has blocking and non-blocking methods
- read data from the file handle

```
- (NSData *) readDataToEndOfFile
- (NSData *) readDataOfLength: (unsigned int) length
```

- write data to a file handle

```
- (void) writeData: (NSData *) data
```

- seeking in a file

```
- (unsigned long long) offsetInFile
- (void) seekToFileOffset: (unsigned long long) offset
```

- closes the file

```
- (void)closeFile
```

NSPipe

- NSPipe has two instances of NSFileHandle
- for read

```
- (NSFileHandle *) fileHandleForReading
```

- for write

```
- (NSFileHandle *) fileHandleForWriting
```

Creating an App that Creates a New Process

- see SortThem app

Non-blocking reads

- avoid waiting for data from a process that will return data asynchronously
- create file handle that posts notifications when data is available
- see TraceRoute app
- ultimately uses the readInBackgroundAndNotify along with a dataReady notification triggered by an NSFileHandleReadCompletionNotification

20. Multithreading

Posix Threads

- mac uses pthreads for native threading, which handles errors errors differently than the rest of the Unix API

Creating threads

```
int pthread_create (
    pthread_t *threadID,
    const pthread_attr_t *attr,
    void *(*startRoutine)(void *),
    void *arg
);
```

- threadID - pointer to a pthread_t
    - thread ID for the new thread will be written here
- attr - set of attributes
    - pass NULL to use the default attributes
    - specific attributes are not discussed (tend to confuse discussions about threaded programming)
- startRoutine - pointer to a function with a signature of void *someFunction (void *someArg)
    - where execution in the thread will start
    - thread will terminate when this function returns
    - someArg parameter is the value of the arg parameter passed to pthread_create()
    - return value is some pointer to return status
    - can pass whatever data structure you want for these two values
- arg - argument given to the startRoutine
- private stack for each thread
- shared heap
- use pthread_join() to bring a thread back with another

```
int pthread_join (pthread_t threadID, void **valuePtr)
```

- will block until thread exits and write return into valuePtr
- can detach with pthread_detach()

```
int pthread_detach (pthread_t threadID)
```

- to create a daemon thread

```
pthread_detach(pthread_self());
```

- see basics.m
- threads are at mercy of the OS scheduler
- main thread is different
- in all other threads, return is equivalent to pthread_exit()
- return in main is equivalent to exit()

Synchronization

- three cases to consider for thread start time within a loop
    - immediate execution
        - generally corresponds to expected behavior and should work OK
    - slightly delayed interleaving
        - may or may not result in slight corruption
    - heavily delayed interleaving
        - generally leads to corruption

Mutexes

- used to serialize access to critical sections of code
    - other threads will be blocked from running that section of code until running thread releases lock
    - an arbitrary thread will be selected to run critical section next
- minimize duration of mutex lock
- mutexes control access to code but it is important to realize this is just a mechanism for controlling data access
- mutex -> pthread_mutex_t
- one method for initialization

```
static pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
```

- another option for initialization is to get a chunk of memory the size of pthread_mutex_t and use pthread_mutex_init() on that memory

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

- use pthread_mutex_init() to create a mutex per data structure

- when finished with a mutex you initialized with pthread_mutex_init(), use pthread_mutex_destroy() to release its resources

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- to acquire a mutex use pthread_mutex_lock()

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- if unavailable, this call will block until it becomes free
- when execution resumes after this call (with a zero return value), you know you have sole possession of the mutex
- to release a mutex, use pthread_mutex_unlock()

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- to not block when acquiring a mutex, use pthread_mutex_trylock()

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- if pthread_mutex_trylock returns with zero, the mutex is locked
- if returns EBUSY, the mutex is locked by another party, and a retry is necessary
- see mutex.m here

Deadlocks

- with multiple mutexes, always acquire in preset global ordering each time
- otherwise a deadlock can be encountered
- simple example
    - thread one locks A and gets pre-empted
    - thread two locks B and gets pre-empted
    - thread one attempts to lock B and blocks
    - thread two attempts to lock A and blocks
- simple test

```
while (1) {
    pthread_mutex_lock (mutexA);
    if (pthread_mutex_trylock(mutexB) == EBUSY) {
        pthread_mutex_unlock (mutexA);
    }
}
```

Condition variables

- used to check if a some prerequisite is true before locking
- if using mutexes, creates a polling operation which will waste CPU time
- pthread_cond_t
- blocking will stop via a signal from another thread
- can initialize with PTHREAD_COND_INITIALIZER or

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);
```

- destroy with

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

- to wait use

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

- web server is classic example of concurrent producer/consumer problem
- for queue of requests with an "accept" thread blocking on the accept call for a connection request and a pool of threads to handle connections
  - accept thread (when space is available on queue)
    - put request on queue
    - signal a connection thread to wake up
  - when the queue is full
    - block on a condition variable until space is available
    - when condition variable causes wakeup, put request in queue
  - connection thread
    - queue has a request
      - get request from the queue
      - if the queue was previously completely full, signal the accept thread that space is available or free space in queue
    - the queue is empty
      - block on a condition variable until there is space in the queue
      - when it wakes up, get the request from the queue

```
pthread_mutex_lock(queueLock);
while (queue is full) {
    pthread_cond_wait(g_queueCond, queueLock);
}
// put item on the queue;
pthread_mutex_unlock(queueLock);
```

```
// signal a connection thread go back to accept()

...
pthread_mutex_lock(queueLock);
while (queue is empty) {
    pthread_cond_wait(g_queueCond, queueLock);
}
// get item from the queue;
pthread_mutex_unlock(queueLock);
// signal the accept thread process the request.
```

- can specify a timeout for blocking with pthread_cond_timedwait

```
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const st

// timespec struct
struct timespec {
    time_t  tv_sec; // seconds
    long tv_nsec; // nanoseconds
};
```

- see webserve-thread.m for a simple webserver using threads (use telnet to communicate)

Cocoa and Threading

NSThread

- abstracts threads
- create with class method

```
+ (void)detachNewThreadSelector: (SEL) aSelector toTarget: (id) aTarget withObject
// aSelector's signature
- (void) aSelector: (id) anArgument;
```

- creates a background thread so no need for waiting, joining, or detaching
- posts and NSWillBecomeMultiThreadedNotification and `[NSThread isMultiThreaded]`
- using threads when this returns NO will slow things down
- pthreads are not aware of this, so manually check when mix the two
- creation of NSThread also creates an NSRunLoop
- if doing any Cocoa calls in the thread, create an NSAutoReleasePool
- NSApplication has a convenience method for thread creation and autorelease pool creation in one (not limited to drawing in resulting thread)

```
+detachDrawingThread:toTarget:withObject:
```

- NSLock is very similar to pthread_mutex_t

- NSConditionLock manages loop for pthread_cond_t -> is a state machine that is programmer definable
- prior to 10.2 child threads could not draw into Cocoa views
- after 10.2 use -(BOOL)lockFocusIfCanDraw

```
if ([drawView lockFocusIfCanDraw]) {
    // set colors, use NSBezierPath and NSString drawing functions
    [[drawView window] flushWindow];
    [drawView unlockFocus];
}
```

Cocoa and thread safety

- parts of Foundation and AppKit are thread safe and some parts are not
- generally
  - immutable objects are thread safe and can be used by multiple threads
  - mutable objects are not thread safe
- there are more considerations
  - NSStrings are generally immutable but through inheritance a class like NSMutableString can be created and passed to an API that takes an NSString
- UI objects are not thread safe and should only be used on the main thread
- default to assuming an object is not thread safe

Objective-C @synchronized blocks

- the use of native Objective-C exceptions (with compiler directive `-fobjc-exceptions`) enables the Objective-C synchronization operator
- @synchronized() locks a section of code
- takes a single argument (any Objective-C object)
  - instance var
  - self
  - class object
- note -> @synchronized(self) inside a class object method refers to the class
- @synchronized is a recursive mutex so a thread can use the same object in different blocks that are synchronized with the same object
- exceptions thrown from within will automatically release the lock

For the More Curious: Thread Local Storage

- can make some code simpler to be sure that storage is private to a thread
- thread local storage enables variables like globals that are private to threads
- errno is thread local

- with pthreads, use pthread_key_create

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

- creates an abstract key and destructor function is called when thread exits to clean up local resources
- set and get

```
int pthread_setspecific(pthread_key_t key, const void *value);
void *pthread_getspecific (pthread_key_t key);
```

- when working with NSThread use

```
- (NSMutableDictionary *) threadDictionary
```

- to use, find current thread with `[NSThread currentThread]`

For the More Curious: Read/Write Locks

- Cocoa and pthreads provide read/write locks
- read/write locks allow a data structure to be read by a number of readers simultaneously but only allow one thread write access at a time
- pthread_rwlock_t works like pthread_mutext_t (but with extra calls)

```
pthread_rwlock_init (pthread_rwlock_t *lock, const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
// specify lock type
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
// unlock
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

- OS X does not define PTHREAD_RWLOCK_INITIALIZER (cannot create them statically)
- generally doubles number of lock operations and can be replaced by a mutex and two condition variables
- good for many reads and few writes

21. Operations

- increasing performance requires parallel programming which requires the use of threads
- threaded programming is error prone and difficult

- Mac OSX 10.5 simplified with NSOperation and NSOperationQueue which handles parallel operations without direct interaction with threads
- concurrent and non-concurrent
- non-concurrent run in own thread while many concurrent operations can be run in the same thread (if operation needs a thread to itself it becomes non-concurrent)
- simple-lifetime and complex-lifetime operations
- simple-lifetime operations override -main, do their work, and then run through bottom of the method
- when -main returns the operation queue assumes the operation has completed and will schedule the next one to run
- complex-lifetime operations provide lifetime control
- override -start
- must maintain KVO contracts for complex-lifetime

Simple-Lifetime Operations

- use simple-lifetime operations for synchronous resource-heavy computations
- subclass NSOperation and override -main

```
- (void) main;
```

- poll -isCancelled for cancellation

```
- (BOOL) isCancelled;
```

- cancellation can cause issues and functions differently with NSOperation than pthreads
- can add and remove dependencies between operations

```
- (void) addDependency: (NSOperation *) op;
- (void) removeDependency: (NSOperation *) op;
```

- circular dependencies are not supported and not detected (managed by NSOperationQueue)
- can inspect dependencies

```
- (NSArray *) dependencies;
```

- can manipulate priority of operations in queue

```
- (NSOperationQueuePriority) queuePriority;
- (void) setQueuePriority: (NSOperationQueuePriority) p;

// NSOperationQueuePriority enum
```

```
enum {
    NSOperationQueuePriorityVeryLow,
    NSOperationQueuePriorityLow,
    NSOperationQueuePriorityNormal,
    NSOperationQueuePriorityHigh,
    NSOperationQueuePriorityVeryHigh
};
```

- does not change OS priority

NSOperationQueue

- accepts NSOperation instances and runs them
- different for 10.5 vs 10.6 and later
- for 10.5
  - concurrent -> same thread
  - non-concurrent -> queue creates and manages threads automatically
- 10.6
  - queue creates and manages threads automatically
- to add an operation

```
- (void) addOperation: (NSOperation *) op;
```

- query for current operations

```
- (NSArray *) operations;
```

- number of concurrent threads is limited by hardware and "other factor"
- can manually tweak and query concurrency factor

```
- (NSInteger) maxConcurrentOperationCount;
- (void) setMaxConcurrentOperationCount: (NSInteger) count;
```

- cancel all operations

```
- (void) cancelAllOperations;
```

- synchronously wait on operations

```
- (void) waitUntilAllOperationsAreFinished;
```

Threading issues

- same issues as threading for data access with non-concurrent operations
- NSNotifications run in same thread as operation
- to post notifications in the main thread, use one of the -performSelectorOnMainThread: methods to trigger a notification on the main thread

MadelOpper

- see MandelOpper project

Bitmap

- describes creation of a bitmap class here

BitmapView

CalcOperations

MandelOpperAppDelegate

NSBlockOperations

- used to run block operations

```
NSBlockOperation *allDone = [NSBlockOperation blockOperationWithBlock: ^{
    // Most likely we are *not* the main thread when this block runs,
    // and we need to be on the main thread to update the status line
    // text field.
    [self performSelectorOnMainThread: @selector(updateStatus:)
                                   withObject: @"done!"
                                   waitUntilDone: NO];
    }
];
```

Complex-Lifetime Operations

- can create concurrent operations by overriding -isConcurrent and returning YES
- tells NSOperationQueue to start the operation in the current thread
- requires any prerequisite code to be setup
- need to override four methods

```
- (void) start;
- (BOOL) isConcurrent; // 10.5 only
- (BOOL) isExecuting;
- (BOOL) isFinished;
```

- do not `[super start];`

KVO properties

- override -isExecuting and -isFinished so that they return YES if your operation is currently executing or if it has finished executing
- make KVO compliant -> when execution state transitions from “not executing” to “executing” make sure that KVO notifications for the isFinished and isExecuting key occur
- NSOperation has many read-only properties that are KVO observable
  - isCancelled
    - YES if the operation has been cancelled
    - NO if it is still free to execute
  - isConcurrent
    - YES if the operation sets up its own execution environment
    - NO if NSOperationQueue should create a thread for it
  - isExecuting
    - YES if the operation is currently running
    - value starts off at NO, changes to YES when the operation runs, then changes back to NO when the operation stops running
  - isFinished
    - YES when the operation has finished
  - isReady
    - YES if the operation can be run
    - NSOperation looks at its dependencies to decide if it is ready
  - dependencies
    - an NSArray of all of the operations this operation is dependent on
  - queuePriority
    - one of the NSOperationQueuePriority* constants
    - mutable property

22. Grand Central Dispatch

- many drawbacks and difficulties to threading, locking, and parallel algorithms
- 10.6 introduced Grand Central Dispatch
- queueing mechanism that provides a way of serializing operations
- can use multiple serial threads combined with parallelism to get deterministic concurrent operations

- removes lock contention and opportunities for starvation
- GCD communicates with the kernel to manage system-wide resources that is not possible with other methods
- GCD will run work it is passed as soon as possible
- just public name for libdispatch
- include `<dispatch/dispatch.h>`
- integrates with Foundation and Core Foundation's runloop
- otherwise use dispatch_main()
- can use NSOperationQueue for pre-10.6 OSs

GCD Terminology

- uses tasks
  - tasks can have dependencies
  - can be broken into subtasks and ultimately work items
- work items are the individual chunklets of work (corollary to helper functions)
- tasks are built from work items
  - when it reaches the front of a queue it gets assigned to a thread
- GCD is a high-level abstraction above threads

Queues

- GCD's fundamental structure run as FIFOs
- work items can be submitted as a block or a function pointer/context pair
- each serial queue is backed by a global queue
- two kinds
  - serial
  - global
- serial will not have two different work items executing simultaneously
- global queue issues work has FIFO but does not guarantee order
- work is run from thread pool is balanced by kernel
- three global queues of different priority
  - high
  - default
  - low
- global queues have unlimited number of concurrent operations
- serial queues can only run 1 operation at a time
- serial queues have target queues where the work gets run (which eventually reaches a global queue)
- serial queue priority is determined by the priority of the target
- serial queues are lightweight and many can be created

- unless there is a good reason for serial work (ensuring single thread manipulates a resource at a time) use global queues

Object-Oriented Design

- libdispatch is written in C but is object-oriented using pointers to opaque data structures
- dispatch_object_t
    - dispatch_once_t
    - dispatch_time_t
- not objects but semi-opaque scalar types

Dispatch API

Queues

- get main queue (serial queue)

```
dispatch_queue_t dispatch_get_main_queue(void);
```

- for global queue (with DISPATCH_QUEUE_PRIORITY_LOW, DEFAULT, or HIGH priority constants and 0UL for flags)

```
dispatch_queue_t dispatch_get_global_queue(long priority, unsigned long flags);
```

- get current queue (queue that issued currently running piece of work) NOTE: Apple recommends only using this for identity tests

```
dispatch_queue_t dispatch_get_current_queue(void);
```

- create queue

```
dispatch_queue_t dispatch_queue_create (const char *label, dispatch_queue_attr_t a
```

- Apple recommends using the reverse-domain DNS-style naming scheme
- set a queue's target

```
void dispatch_set_target_queue(dispatch_object_t object, dispatch_queue_t target);
```

Dispatching

- dispatch work to queues