Development of Applications Independently from a Threading Model

Robert Bogart, Sinan Bayati

Overview

- Introduction
- Objectives
- Architecture
- Applications
    - Stock Ticker
    - Sudoku Validator
    - Multithreaded Sorting
- Summary
- Final Thoughts

Introduction

- A lot of applications stand to gain an improvement in performance by having the ability to perform operations asynchronously and/or concurrently.
- Regardless of the application, the methods employed to execute concurrent operations are similar.
- Opportunity to create a common framework that can be used by most of these applications to execute code concurrently.
- Not a novel idea. Apple offers a subsystem known as Grand Central Dispatch (GCD) which is a collection of language features, runtime libraries and system enhancements to support concurrent code execution.

Objectives: What do we hope to achieve?

- Rapid development of a number of applications which can perform asynchronous/concurrent operations.
    - Expose a set of semantics-based interfaces for performing these operations which is easy to understand.
- Ability to allow applications to adjust their degree of concurrency with little change to the code even after implementation is complete.
- Reduce the likelihood of failure due to out-of-resource issues by allocating up front.
- Reduce overhead incurred by thread creation/destruction by reusing existing threads.
- Abstract away the platform-specific implementation details of threading so that applications using this model can easily be ported across a number of systems with little to no change to the code.

Architecture: Overview

Application

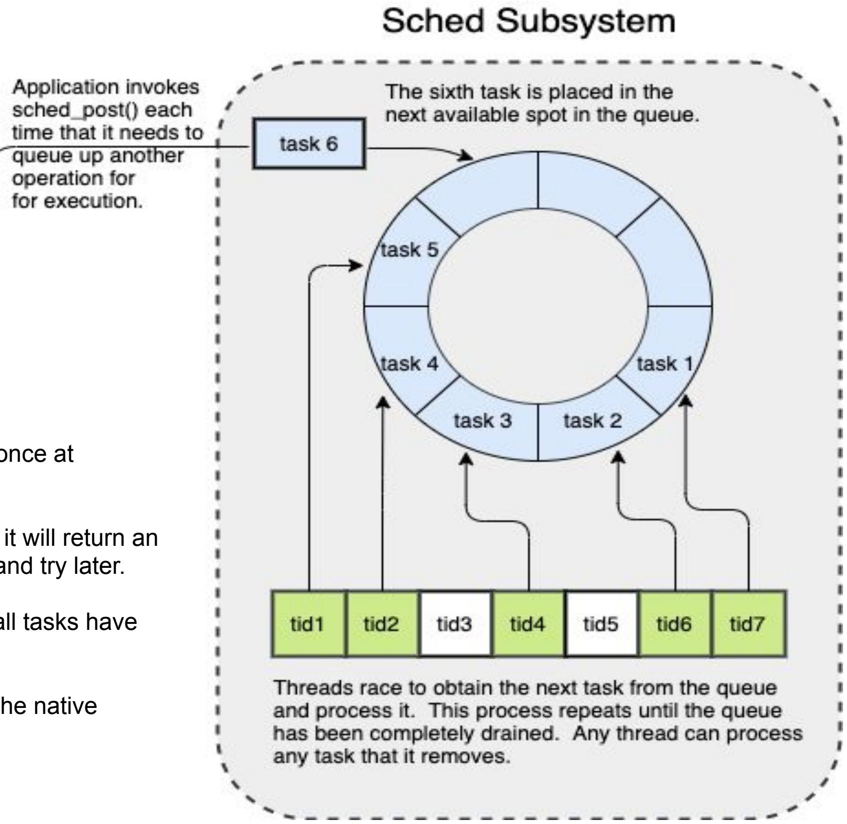
```
#include <sched.h>

#define THREADS      7
#define QUEUE_DEPTH 8

void main(int argc, char **argv)
{
    sched_init(sp, THREADS, QUEUE_DEPTH);
    (void) sched_post(..);
    [...]
    (void) sched_execute(..);
    (void) sched_fini(..);
}
```

- Number of workers and task queue capacity is specified once at initialization time.

- If the task queue is full when `sched_post()` is invoked, it will return an error giving the caller an opportunity to wait, or move on and try later.

- The call to `sched_execute()` will block the caller until all tasks have been completed.

- The order in which the threads run is at the discretion of the native scheduler.



Sched Subsystem

Application invokes sched_post() each time that it needs to queue up another operation for for execution.

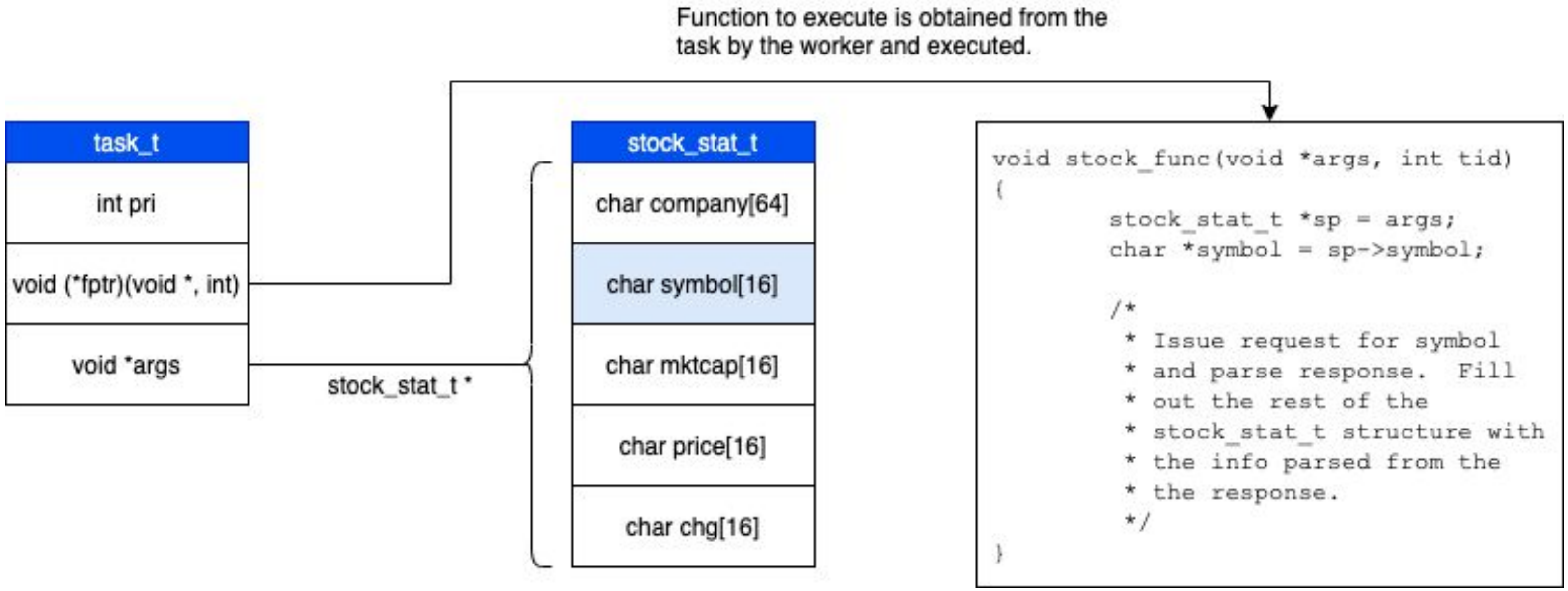
The sixth task is placed in the next available spot in the queue.

Threads race to obtain the next task from the queue and process it. This process repeats until the queue has been completely drained. Any thread can process any task that it removes.

Architecture: Task Creation

- A task is a special data structure which contains information:
  - Address of the function to execute.
  - Arguments that the function needs.
  - Priority.
  - Results of the operation which can be written to a dedicated field(s) supplied in the function arguments.
- Once a task has been posted to the scheduler, the caller should not attempt to access the task until has been fully processed.
- A given task structure can be reused for later operations **IFF** you are positive that the scheduler isn’t using it anymore.
- Best practice is to create as many tasks as there are operations that you want to run.

Stock Ticker: Task Initialization



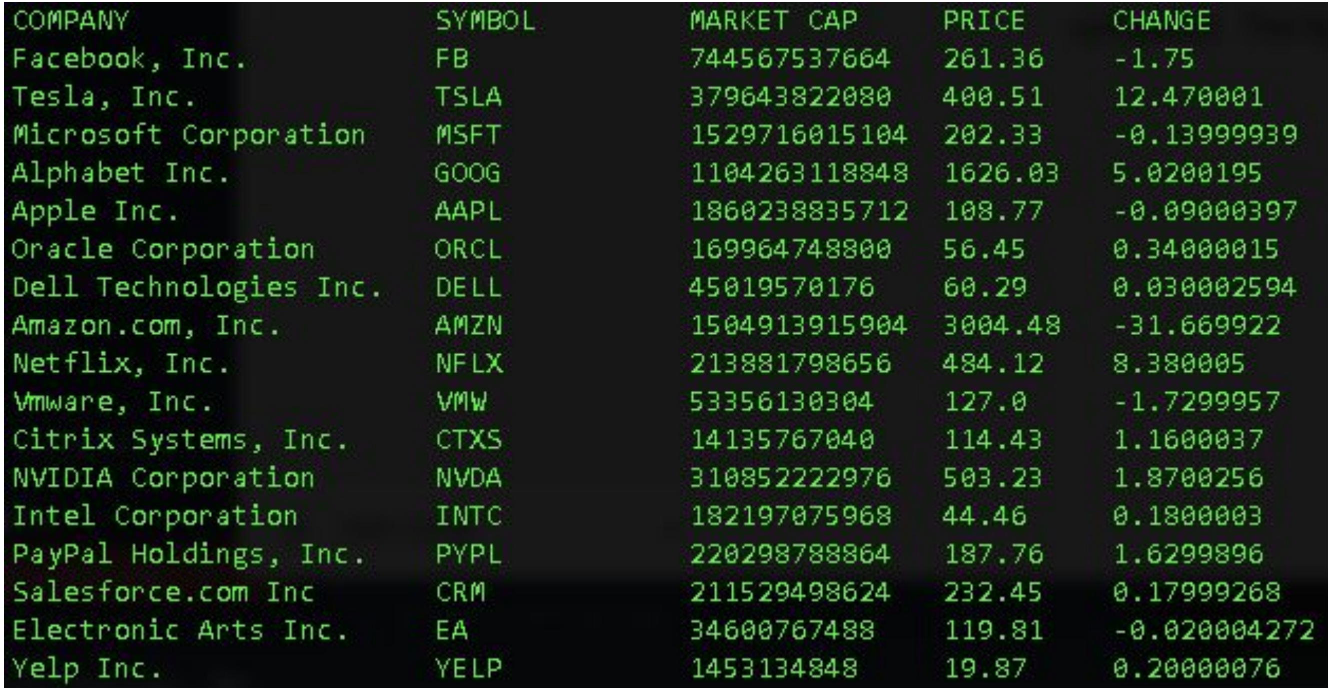
Function to execute is obtained from the task by the worker and executed.

```
void stock_func(void *args, int tid)
{
        stock_stat_t *sp = args;
        char *symbol = sp->symbol;

        /*
         * Issue request for symbol
         * and parse response.  Fill
         * out the rest of the
         * stock_stat_t structure with
         * the info parsed from the
         * the response.
         */
}
```

Stock Ticker: Setup

- For *N* different stock symbols, create *N* tasks.
- Populate each task with:
  - The address of `stock_func()` which is called by the scheduler on our behalf.
  - Address of a `stock_stat_t` structure which has the `symbol` field initialized to the value of the stock symbol of interest.
- Post the task(s) to the scheduler.
- Call `sched_execute()`.
- As `stock_func()` is called by the scheduler for each task, it fills out the remaining fields in the `stock_stat_t` structure for us.
- When `sched_execute()` returns, all of our `stock_stat_t` structures have been fully populated with the desired information and it can be printed!

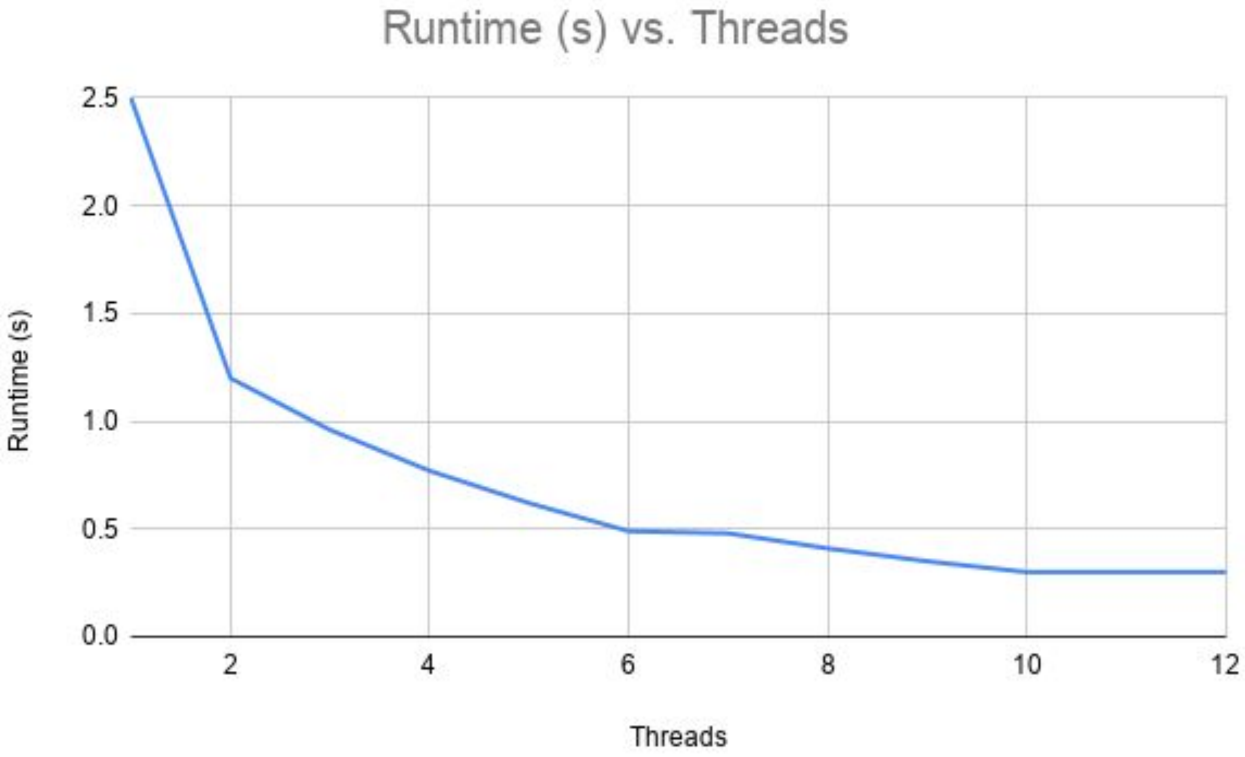
Stock Ticker: Output



```
COMPANY                  SYMBOL         MARKET CAP     PRICE     CHANGE
Facebook, Inc.           FB             744567537664   261.36    -1.75
Tesla, Inc.              TSLA           379643822080   400.51    12.470001
Microsoft Corporation    MSFT           1529716015104  202.33    -0.13999939
Alphabet Inc.            GOOG           1104263118848  1626.03   5.0200195
Apple Inc.               AAPL           1860238835712  108.77    -0.09000397
Oracle Corporation       ORCL           169964748800   56.45     0.34000015
Dell Technologies Inc.   DELL           45019570176    60.29     0.030002594
Amazon.com, Inc.         AMZN           1504913915904  3004.48   -31.669922
Netflix, Inc.            NFLX           213881798656   484.12    8.380005
Vmware, Inc.             VMW            53356130304    127.0     -1.7299957
Citrix Systems, Inc.     CTXS           14135767040    114.43    1.1600037
NVIDIA Corporation       NVDA           310852222976   503.23    1.8700256
Intel Corporation        INTC           182197075968   44.46     0.1800003
PayPal Holdings, Inc.    PYPL           220298788864   187.76    1.6299896
Salesforce.com Inc       CRM            211529498624   232.45    0.17999268
Electronic Arts Inc.     EA             34600767488    119.81    -0.020004272
Yelp Inc.                YELP           1453134848     19.87     0.20000076
```

Stock Ticker

- More threads improve performance.
- Throughput only improves marginally beyond 6 threads.
- The power of adjustable concurrency made this easy to discover.



Sudoku Validator

- Initially we saw results:
  - 1 thread: 848,261ns
  - 11 threads: 1,891,252ns(?!)
- How could this be?
- DTrace helped us find the source of the performance pathology.
- First place to look: `process_task()` execution times.
- Using the pid DTrace provider, we observed that `process_task()` had much longer execution times when running in multithreaded mode than it did when using single thread.

Sudoku Validator: Eleven Threads

```
  value  ------------- Distribution ------------- count
 131072 |                                         0
 262144 |@@@@                                     1
 524288 |@@@@@@@@@@@@@@@                          4
1048576 |@@@@@@@@@@@@@@@@@@@@@@                   6
2097152 |                                         0
```

Of the eleven times that `process_task()` was invoked, we can see that the vast majority of them were taking between 524288 and 1048676 ns.

Sudoku Validator: Single Threaded

```
 value  ------------- Distribution ------------- count
  4096 |                                         0
  8192 |@@@@@@@@@@@@@@@@@@@@@@                   6
 16384 |@@@@@@@@@@@                              3
 32768 |@@@@                                     1
 65536 |                                         0
131072 |                                         0
262144 |@@@@                                     1
524288 |                                         0
```

Of the eleven times that `process_task()` was invoked, we can see that the vast majority of them were taking between 8192 and 16384 ns.

Sudoku Validator: process_task()

- The `process_task()` function appears to be the primary source of latency.
- What’s going on in there?
- Using the syscall DTrace provider, we saw the following instruction stack:

```
lwp_park:entry
libc.so.1`__lwp_unpark+0x14
libc.so.1`cancel_safe_mutex_unlock+0x1e
libc.so.1`printf+0x12e
sudoku`validate_3_by_3_func+0x4b
sudoku`process_task+0x49
sudoku`worker_func+0x125
libc.so.1`_thrp_setup+0x8a
Libc.so.1`_lwp_start
```

- It looks like **`printf()`** is the culprit!

Sudoku Validator: Conclusion

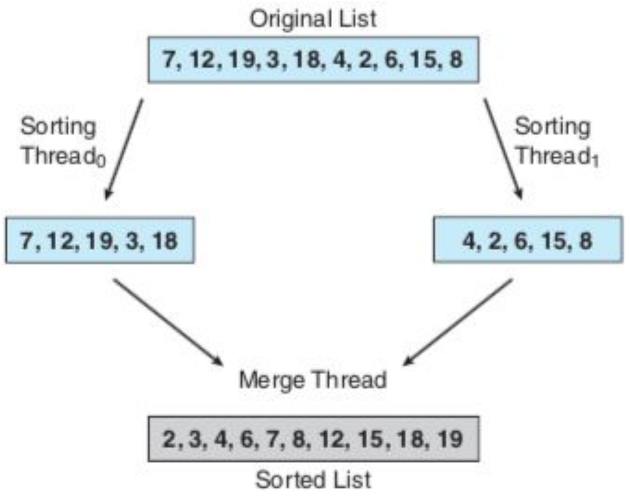
- Eliminating the calls to `printf()` markedly improved performance.
- Execution times for `process_task()` when running with 11 threads now:

```
value  ------------- Distribution ------------- count
 1024 |                                          0
 2048 |@@@@                                      1
 4096 |@@@@@@@@@@@                               3
 8192 |@@@@@@@@@@@@@@@@@@@@@@                    6
16384 |@@@@                                      1
32768 |                                          0
```

- No more contention for performing I/O.
- Over-gratuitous logging is OK for debug builds, but not for production.
- DTrace allows you to instrument production code dynamically! (www.dtrace.org)

Multithreaded Sorting Program

- Original array divided into two halves.
- After each half is sorted, merge them together by a third thread.
- Able to reuse one of our sorting threads to do the merge, thus only requiring two threads.
- Sort operation O(n log n).
- Merge O(n).

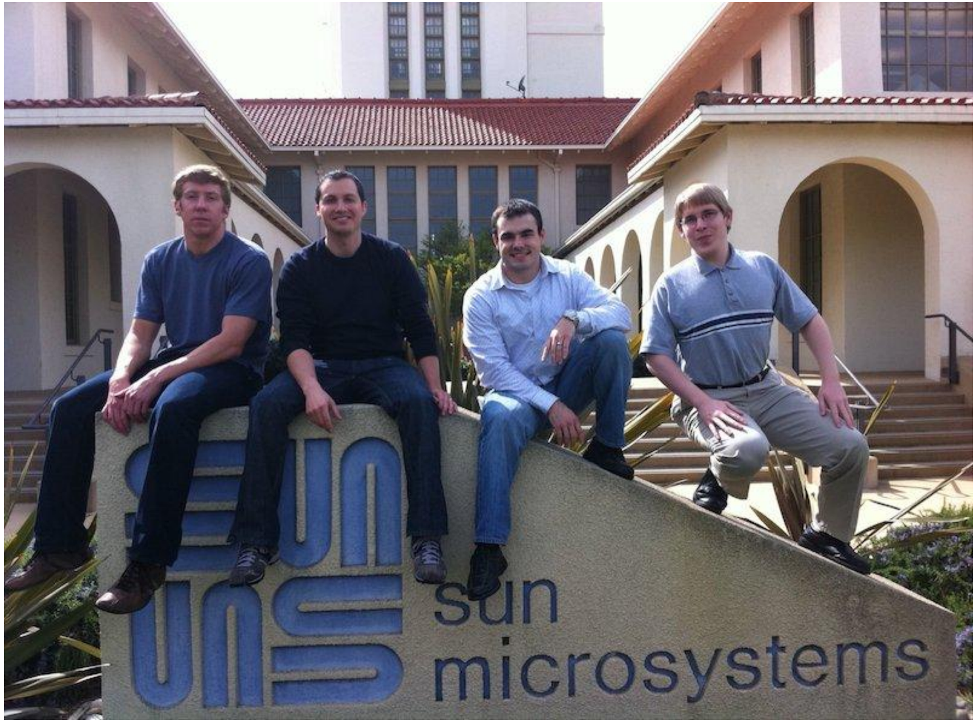


Summary

Decoupling the threading model from the application offers major benefits. In particular:

- Allow developers to focus more on the objective that matters: the features they want to offer.
- Develop highly portable applications.
- Quickly characterize application performance pathologies.
- Tune applications without substantial change.
- Shorten the duration of future development and test cycles, reducing time to market.
- Mitigate risk by reusing common, mature, battle tested code.

Final Thoughts



“Kick butt and have fun” --Scott McNealy

“Simple can be harder than complex: you have to work hard to get your thinking clean to make it simple. But it’s worth it in the end because once you get there, you can move mountains.” --Steve Jobs