Development of Applications Independently from a Threading Model

Canvas Group Number	4
Student	Robert Bogart
Student	Sinan Bayati

Table of Contents

Development of Applications Independently from a Threading Model **1**

Table of Contents **2**

Abstract **3**

Introduction **3**

Background **3**

Approach/Methodology/Background **5**

- Task 5
- Priority Queue 6
- Worker Thread Pool 7
- Worker Thread Sequence Diagram 8
- Putting It Together 9

Results, Findings, Analysis **10**

- Stock Ticker 10
- Sudoku Validator 12
- Multithreaded Sorting Program 15

Conclusion and Recommendation **17**

References **18**

Appendix **19**

- Project source code 19
- DTrace script for aggregating execution times 19
- DTrace script for tracking application system calls 19

Abstract

With greater processing power and more sophisticated operating systems, multithreaded applications have become more prominent in the last 20 years. Depending on the nature of an application and assuming that it may perform multiple operations that are independent of each other, that is, the outcome of one operation does not affect the starting state of another, it may be desirable to multithread such operations. It is normal to see developers create threads by hand (i.e. a manual call to `pthread_create()`) and to do something more might be considered overkill for a single application that occasionally performs asynchronous operations. At the same time, there are a number of benefits to having a type of framework that handles the rigors of multithreading on behalf of the developer -- especially in a situation where a high number of threads are needed or one where a number of multithreaded applications are being developed. This project intends to explore the merits of an approach where a threadpool and priority queue are used over other approaches where threads are created and destroyed manually for specific tasks. Finally, we will implement this strategy for multithreading a number of different applications to demonstrate its convenience.

Introduction

The procedures involved in developing multithreaded applications can be laborious, tedious, cumbersome and error prone. With a lot of multithreaded applications requiring similar architectures, it is desirable to relieve developers of repeatedly implementing the scaffolding required to multithread an application by introducing a common library that implements most of the multithreading specific operations regardless of the application. There are several merits to this generic approach. First, it frees the developer from the burden of creating, joining, and terminating threads which can be repetitive across applications. Second, it will allow the developer to tune the number of threads used to perform a task without any invasive changes to the architecture of the application itself, reducing both risk and time required to change the level of concurrency. Third, it frees the developer to think about the problems that they intend to solve on a semantic level without spending time focusing on implementation details such as POSIX interfaces. Finally, because the thread implementation details are abstracted away from the application, highly portable multithreaded applications can be quickly developed so long as the underlying system has some notion of a thread or task as well as synchronization primitives similar to mutexes, semaphores and conditional variables. We will demonstrate how this model can be used to quickly and conveniently implement a number of different multithreaded applications, where some make use of task concurrency, some make use of data concurrency and others that are hybrid.

Background

Normally, when implementing a multithreaded application, a call to `pthread_create()` is performed, supplying a number of arguments including the function that we want to run and the arguments that it requires. Furthemore, when (or if) these threads complete, we must invoke `pthread_join()` supplying the identifier of the thread (that we obtained during its creation and hopefully retained), to wait on for completion, otherwise we have no idea whether the operations that we initiated have completed or are still in progress. The application is responsible for retaining knowledge of information such as the thread identifier to call `pthread_join()` on it. An excerpt of such a program might look like the following:

```
/* Create one thread for each command-line argument */
for (tnum = 0; tnum < num_threads; tnum++) {
       tinfo[tnum].thread_num = tnum + 1;
       tinfo[tnum].argv_string = argv[optind + tnum];
       /*
        * The pthread_create() call stores the thread ID into
        * corresponding element of tinfo[]
        */
       if ((s = pthread_create(&tinfo[tnum].thread_id, &attr,
           &thread_start, &tinfo[tnum])) != 0)
       handle_error_en(s, "pthread_create");
}

/*
 * Destroy the thread attributes object, since it is no
 * longer needed.
 */
if ((s = pthread_attr_destroy(&attr)) != 0)
       handle_error_en(s, "pthread_attr_destroy");

/* Now join with each thread, and obtain its returned value. */
for (tnum = 0; tnum < num_threads; tnum++)
       s = pthread_join(tinfo[tnum].thread_id, &res);
```

Figure 2.1 Excerpt of a basic multithreaded application.

In figure 2.1, the vast majority of the code is used for creating threads, specifying functions and arguments for them to run and finally, waiting for them to compete -- and practically none of this even has anything to do with the application! Not only that, but as you can see in the example above, if we would like to run the same functions again in the same asynchronous manner, the developer must perform the same ceremony of creating more threads, retaining their identifiers and joining them. On top of the tedium required to develop multithreaded applications this way, they can also be vulnerable to performance penalties due to the cost of repeated thread creation and destruction. In addition to simplifying the development of multithreaded applications, it would also be desirable to reuse existing threads, avoiding the overhead of repeated resource allocation and reclamation. In this project, we aim to create an application that demonstrates it's versatility by having multiple implementations that will be shown in the next chapters.

Approach/Methodology/Background

In order to implement a mechanism that allows for the execution of asynchronous operations there are several requirements along with considerations that must be made. Starting with the constructs that are required at the most fundamental level, some kind of queueing structure is necessary to serve as storage for operations which we intend to execute.

Task

In the context of this project, an operation is referred to as a task and comprises a few important data members in a C struct:

```
typedef struct task {
       uint64_t pri;              /* Task priority. */
       void (*fptr)(void *, int); /* Address of function to execute. */
       void *args;                /* Argument supplied to function. */
} task_t;
```

Within a `task_t` structure, each member has a certain significance. The first member, pri, denotes the priority of the task. The smaller the priority, the more important the task is considered. Tasks are fetched for execution by order of importance. The second member, `fptr`, is a pointer to a function which takes two arguments -- an address of some structure whose actual data type is only known to the application itself, and an integer which represents the index of the thread invoking the function. When the function in fptr is executes, it can cast the first argument (`void *`) to whatever data type it expects that argument to be. As for the thread index, its usage is usually optional, but it is supplied to the function in the event that the

application code invoked needs to perform some action which is predicated on the thread which is invoking it.

Priority Queue

The priority queue is a structure which holds outstanding tasks to be processed. As discussed earlier, the queue is drained by threads which remove elements from the queue and process them. The term “queue” is used generally here, but the underlying storage structure is actually a min-heap, defined as follows:

```
typedef struct heap_elem {
       int val;      /* Used as the key in min-heap operations. */
       void *meta;   /* Proprietary metadata attached to key. */
} heap_elem_t;

typedef struct heap {
       int capacity;       /* Heap capacity. */
       int total;          /* Total number of unprocessed elements. */
       heap_elem_t *data;  /* Contiguous allocation of heap_elem_t. */
} heap_t;
```

The relationship between a `task_t` and a `heap_elem_t` is best described by this illustration:

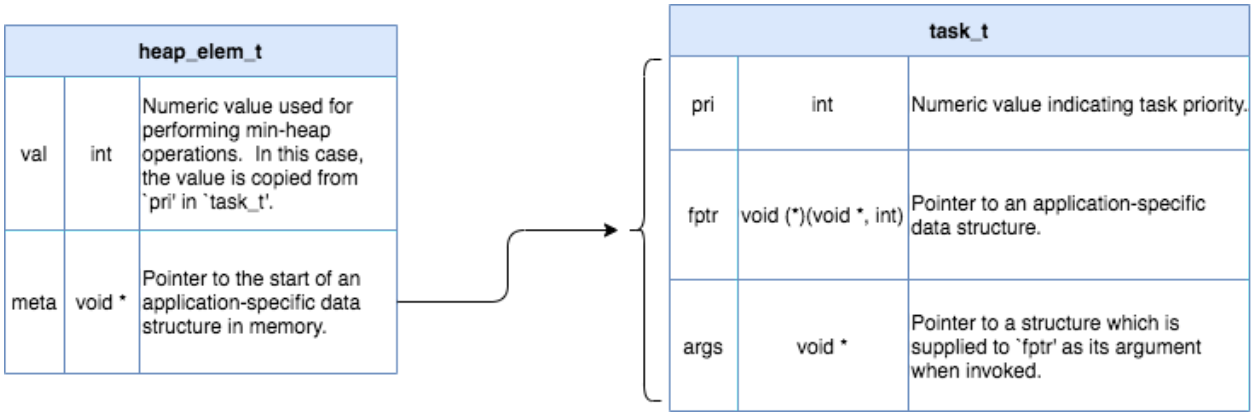

Figure 3.1 The mapping between a heap element and a task.

As you can see in figure 3.1, the address of the `task_t` structure is stored in `meta`, while the task priority itself is copied into the `val` field of the `heap_elem_t`. The reason why a copy of the priority is made is because the `heap_elem_t` structure itself is intended to be completely unaware of tasks and priorities but still requires that value when performing standard heap operations (insert/remove). Finally, the runtime complexity of the insertion and removal operations is O(log n). Because the queue of tasks will be modified by more than one thread, it is necessary to serialize access to it in order to prevent the possibility of data corruption.

Additionally, it is desirable to put worker threads in the pool to sleep when the queue is empty (e.g. there is no work to do), and signal them when the queue new tasks are ready for processing. For this reason, both a mutex and a condition variable are introduced. The final composition of the priority queue is:

```
typedef struct priority_queue {
        pthread_cond_t cv; /* For signalling sleeping workers. */
        pthread_mutex_t lock;     /* For serializing access to the queue. */
        heap_t *heap;             /* Min-heap of tasks. */
        int remaining_tasks;      /* Number of remaining tasks. */
} priority_queue_t;
```

Worker Thread Pool

Next, we need a group of threads which will consume operations (i.e. tasks) from the queue and execute them. Tasks are posted to the queue by the application. Some predetermined number of threads are created at initialization time. Their thread id is recorded in an array. It is necessary to retain the thread identifiers so that if/when a program wants to stop the scheduler and free its resources, `pthread_join()` can be called on each thread, ensuring that all threads have returned before reclaiming all resources consumed by the priority queue and its respective synchronization primitives. Internally, each thread in the thread pool runs an infinite loop which will only be broken upon observing that the scheduler state has changed to DONE which indicates that all workers should exit gracefully upon completely draining the queue. The life of a worker thread in the threadpool is depicted by the following flow chart.

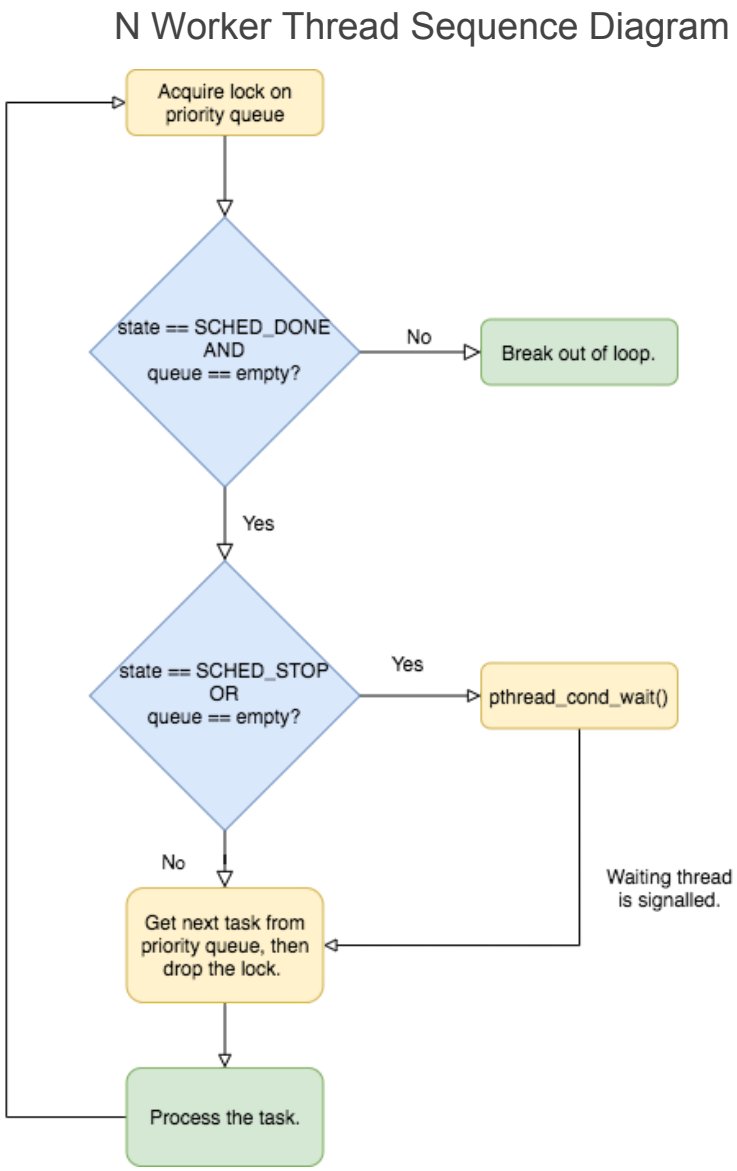

Figure 3.2 The worker thread flow chart.

As indicated in the sequence diagram above, each worker thread in the pool will loop indefinitely, collaboratively draining the queue until no tasks remain. At that point, they will wait on the condition variable to be signaled, indicating either that there is more work to be done, or the scheduler has changed state. A few important points:

1. Tan squares represent moments when a lock is either acquired or released.
2. The lock is always dropped before processing a task in order to keep the amount of time that we spend holding the lock to a minimum.
3. The only way for a thread to terminate is if the scheduler state is changed to SCHED_DONE.

Putting It Together

All of the elements discussed so far are the primary ingredients for the (perhaps naively named) scheduler. The implementation details have been mostly abstracted away -- the application does not need to have intimate knowledge of anything but the task_t structure and the interfaces exposed by the scheduler. Those APIs and their semantic meanings are:

Name	Arguments	Return type	Description
sched_init	1. Address of an uninitialized sched_t. 2. Number of worker threads. 3. Queue depth.	Bool: True on success, false upon failure.	Allocates memory needed for heap as well as all synchronization primitives and threads needed for processing tasks.
sched_post	1. Address of an initialized sched_t. 2. Address of a task_t to post. 3. A flag indicating whether to process immediately or when sched_execute() is invoked.	Bool: True on success, false upon failure.	Inserts a task into the priority queue and signals all workers who may be asleep, waiting for work. This function **does not block the caller** waiting for task completion.
sched_execute	Address of an initialized sched_t.	void	If the scheduler is not executing tasks already, this will jump start the processing of tasks. **This function blocks the caller until the queue has been fully drained.**
sched_fini	Address of an initialized sched_t.	void	Sets the state of the scheduler to DONE and wakes up all threads who might have been asleep. Upon observing the new state, worker threads will exit. All resources are freed.

Figure 3.3 Scheduler APIs

Results, Findings, Analysis

With the approach to the scheduler design ironed out, we put it to work, developing different applications and testing their respective performance with different degrees of multithreading.

Stock Ticker

The purpose of this application is to read in a file containing a list of stock symbols and for each symbol, issue a request to query1.finance.yahoo.com asking for information associated with this symbol, including full company name, net market cap, current stock price, and how much it has changed either way since the market opened. The payload of the response is a JSON object containing the aforementioned data. An array with the results is populated as the responses are received however, the array is not displayed to the screen until all requests have been completed. The figure below shows sample 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

Figure 4.1 Sample output from the stock ticker.

We used the utility TIME(1) for measuring latency. While some amount of time is spent initially, opening and reading a file, and at the end where results are printed to the screen, the vast majority of the time will be spent performing requests and as the number of requests that we perform increases, the more insignificant the time we spend reading from a file contents in comparison to the overall execution time. It's worth mentioning that the first time the stock file is accessed, it may be cold, leading to higher latency times, but repeated accesses during tests will likely find it cached by the file system, but that is beyond the scope of this project. When performing the requests for each symbol (in a file of 17) sequentially, the amount of time

required to display the results is in the neighborhood of 2 and a half seconds. Entire fortunes can be obtained or lost in that amount of time on Wall Street! Thanks to the ability to alter the degree of multithreading with the change of a single character to the second argument in `sched_init()`, we can change the way we initialize our scheduler

from:

```
(void) sched_init(&scheduler, 1, queue_capacity);
```

to:

```
(void) sched_init(&scheduler, 2, queue_capacity);
```

thereby doubling our level of multithreading. Running the test again with two threads instead of one results in a total run time of 1.2 seconds, approximately half of the execution time that it took when performed sequentially.

While we know that performance has improved, it would be desirable to see how many threads can be thrown at the problem until we reach the point of diminishing returns. Thanks to our ability to conveniently control the level of multithreading, we can quickly explore our options. The table below shows our results.

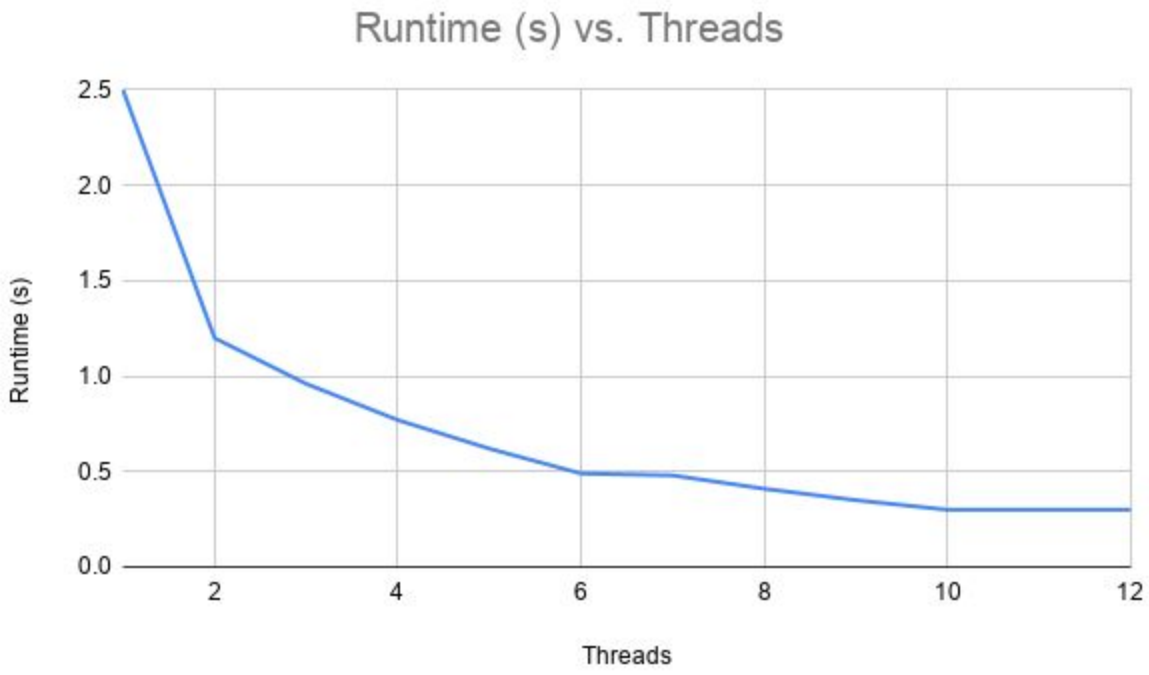

Figure 4.2 Graph showing the total runtime of the stock ticker as threads increase.

While we can see that performance begins to improve as the number of threads go up, we can also see something which is perhaps just as important -- namely how much faster the program runs with each additional thread. While the program runs at its fastest with around 12 threads, we can see that execution time improvements come in much smaller increments when adding more threads beyond 6. In the context of this problem, that realization might seem pedantic, but

in an environment where threading resources are more scarce, knowing where the diminishing returns are could make the difference between whether or not a feature ships with the product.

Sudoku Validator

A Sudoku puzzle uses a 9x9 grid in which each column and row, as well as each of the nine 3x3 subgrids, must contain all of the digits between 1-9 (inclusive). If we were to design an algorithm which validates the integrity of a fully populated Sudoku puzzle, it would require validating each column, each row and then each 3x3 submatrix within the 9x9 matrix.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.3 A solution to a 9x9 Sudoku puzzle.

There is a problem discussed in more detail in *Operating Systems Concepts by Silberschatz Ninth Edition, Galvin and Gagne*, where the author challenges the reader with an exercise to implement a program multithreaded program for validating a puzzle. In particular, the recommended strategy was:

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3x3 subgrids contains the digits 1 through 9

Before that was attempted, we thought it would be instructive to know just how much of a difference a total of eleven threads makes over one. Since the latency of this program will likely be measured in nanoseconds due to how cheap the operations are and how few of them there are, we used a DTrace so that we would not need to instrument any part of the application. A simple DTrace script which uses the pid provider to instrument the entry and return points of the function which kicks off the scheduler and blocks until the entire Sudoku map has been validated was used:

```
#!/usr/sbin/dtrace -s

pid$target::validate_sudoku_map:entry
{
        self->time = timestamp;
}

pid$target::validate_sudoku_map:return
{
        self->now = timestamp - self->time;
        printf("%lu\n", self->now);
}
```

In order to really accentuate the difference in performance between a sequential implementation and a multithreaded one, we tuned our scheduler during initialization to have a thread pool of one worker. When executing the Sudoku validator we observed a total execution time of 848,261 nanoseconds. Next, we increased the number of worker threads to a total of eleven, matching the total number of tasks that must be processed. To our surprise and dismay, a total execution time of 1,891,252 nanoseconds was observed. Clearly, throwing more threads at the problem did not help -- moreover, it made things worse. Given that tasks are not processed while holding the lock, contention for the task queue seemed to be an unlikely suspect as wait times would be nearly negligible. The other area to investigate was the function which processes tasks which is small and concise:

```
void
process_task(task_t *pt, int thread_num)
{
  assert(pt != NULL);

  pt->fptr(pt->args, thread_num);
}
```

Our next objective was to measure the latency incurred every time this function executes. Using a DTrace aggregation, the left column (value) represents execution time as a bucket and the right column (count) represents the number of elements (times the function was executed) in that bucket. With this approach, we can categorize every call made to `process_task()` by its execution time. Processing tasks sequentially, we observed the following results:

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

Figure 4.4 DTrace aggregation for `process_task()` execution times when tasks are processed sequentially.

Based on the results in figure 4.4, we can see that the majority of the tasks were processed in 8192 nanoseconds or less. When running the same test again with eleven threads, the following aggregation was observed:

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

Figure 4.5 Output of DTrace aggregation for `process_task()` execution times when tasks are processed sequentially.

Surprisingly, execution times for processing tasks degraded dramatically with the majority of them taking 1048576 nanoseconds or more to complete. With this in mind, the culprit was narrowed down further. Rather than guess, we once again turned to DTrace to tell us what is happening at the point when a task is being processed with an eye for anything that might involve putting a thread to sleep (e.g. a call to lwp_park). Using the syscall provider, we found a very frequently occurring stack during task processing:

```
4  63791                    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
```

For debugging purposes, part of each task printed a small amount of information to the screen and there was clearly contention due to attempting to perform I/O at the same time, as `lwp_park()` would suggest. There are three functions that can be associated with a task in the context of the sudoku problem:

- `void validate_rows_func()`
- `void validate_cols_func()`
- `void validate_3_by_3_func()`

Within the body of each of these functions, we identified the following line as the suspect:

```
printf("Thread num: %d.\n", thread_num);
```

Using a single threaded model, there would never be contention among multiple threads in this process, which would explain why a thread was never blocked midway through processing a task. This would also explain why execution times for processing a task were significantly faster in a sequential model than a multithreaded one. Eliminating the `printf()` statement from each function resulted in a total execution time of 459,446 nanoseconds -- half of the execution time of a single-threaded model. In particular, task processing times dramatically improved:

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

Figure 4.6 Task execution times when running with eleven threads where none of the worker threads print anything.

The moral of the story here is that while multithreading can greatly improve the performance of a program, it is highly dependent on what the threads are actually doing. While there may be no contention for a resource in user space, we must be cognizant of these kinds of issues in the kernel and make considerations for avoiding them in the design of our applications. As an added bonus, the fact that we were able to dynamically tune the level of multithreading for testing purposes, without introducing invasive changes to the code, aided in our efforts to quickly and conveniently track down the smoking gun.

Multithreaded Sorting Program

Our final use case for our project was another problem presented in *Operating Systems Concepts by Silberschatz Ninth Edition, Galvin and Gagne* where a list of unsorted integers is divided into two sublists of equal size and given to two separate sorting threads which each sort their respective sublists. Upon both threads completing their job, a final thread should merge the two sublists into one fully sorted list.

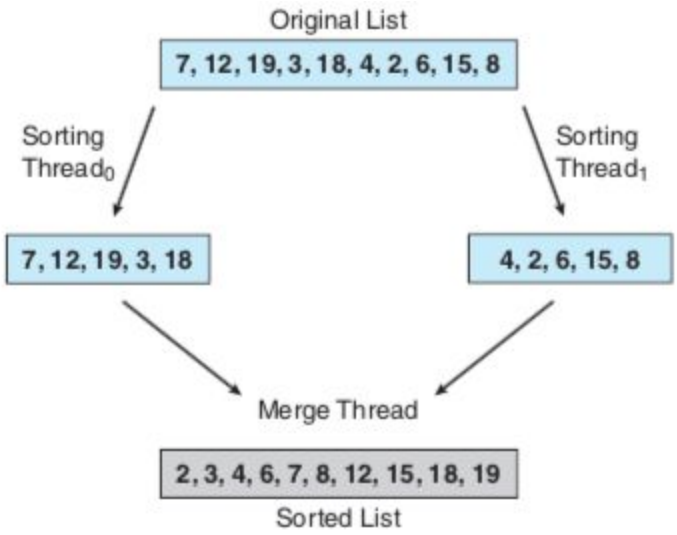

Figure 4.7 Multithreaded sorting.

Unlike the first two use cases, this one primarily demonstrates how convenient it can be to multithread a process like this. Two threads are needed to sort their respective sublists. While the book recommends creating a third thread for the merging, we can reuse one of our existing threads in our threadpool for that operation after they have both completed their initial sorting tasks.

Each sorting task must be given a “slice” of the original array. A structure is defined as follows:

```
typedef struct array_slice {
        int *array;
        int left;       /* Index of start. */
        int right; /* Index of end. */
} array_slice_t;
```

The sorting function used is a basic version of heapsort and can be found in *Data Structures & Algorithm Analysis in C by Mark Allen Weiss*. The reason that heapsort was selected was because it always has a worst/average/best case of O(n log n) runtime complexity and can be implemented iteratively pretty easily.

With these items defined, there is little more required beyond a small amount of boilerplate. We first kick off our sorting tasks and block until they have completed:

```
for (i = 0; i < 2; i++) {
        task_init(&tasks[i], 1, sort_func, (void *)&slices[i]);
        (void) sched_post(&sched_sort, &tasks[i], false);
}
sched_execute(&sched_sort);
```

Once `sched_exeute()` returns, the task queue is empty and we are free to add one final task -- the merge function. While not discussed above, it is a simple function which expects two sorted arrays and lengths and will merge them into a single array. This process is quite straightforward:

```
task_init(&tasks[0], 1, merge_func, (void *)slices);
(void) sched_post(&sched_sort, &tasks[0], false);
sched_execute(&sched_sort);
```

Not only were we able to conserve resources by reusing one of our original sorting threads, we also reused the storage for one of our tasks. At this point, the array is fully sorted. When comparing performance between single and multithreading models, the difference was relatively small at first. We know that the complexity of the sorting tasks would always be O(n log n), however, the merge task is O(n). As the size of the array increased however, the cost of the O(n) merge at the end became less significant in comparison to the cost of the sorting tasks. Arrays of 100,000 elements in size or larger exhibited this trend the most. With arrays that are sufficiently large (i.e. 100,000+ elements), the use of a second thread to help sort the array will reduce execution time by about 50% which is probably about as good as one can hope for by adding a second thread.

In order to reduce execution time further, it would be advisable to divide the array up into a greater number of parts and distribute those to more than two threads. This would also require a change to the merge function in order to perform an N-way merge as opposed to a standard two-way merge that is normally seen in mergesort. While we speculate this idea to be highly effective, it is beyond the intended scope of this test case.

Conclusion and Recommendation

In the three scenarios discussed above, we observed some major advantages offered by decoupling the threading model from the application. In the first instance, with our stock ticker it was clear that being able to quickly throw more threads at the problem so that requests could be performed in parallel made a substantial difference in execution time. It also enabled us to easily see where we reach the point of diminishing returns.

In the second case, with the Sudoku puzzle validator, we initially observed that additional threads increased execution time for a mysterious reason. Without the ability to easily tune the number of threads and iterate over different test cases however, that issue may not have ever been exposed in the first place. We also learned that just because two threads do not contend for a resource within the application does not mean that they won't need to contend for something in the kernel. We must be cognizant of this or else the hard work that we have done to multithread an application could be in vain.

In the final case, while we observed that adding another thread when sorting an array thereby splitting the work among two threads can reduce execution time by 50%, however it should be understood that in order to reap those benefits, the amount of data to be sorted must be of a certain quantity, otherwise, it will not be worth it.

Ultimately, designing a subsystem like the one in this study, which allows multiple applications in an ecosystem to take advantage of what it offers is highly encouraged. Subsequent applications can be developed, tuned and tested more rapidly, saving organizations hundreds (if not thousands) of hours of time as well as millions of dollars over solutions where each application continues to reimplement the same software for its own purposes, leading to longer hours, more source code maintenance points, greater risk and missed deadlines.

References

1. Silberschatz, A., Galvin, P. and Gagne, G. (2012). Operating System Concepts. 9th ed. John Wiley & Sons, Inc.
2. Weiss, M. (1992). Data Structures & Algorithm Analysis in C. Addison Wesley Longman, Inc.
3. Kernighan, B., Ritchie, D. (1988). The C Programming Language. 2nd ed. Prentice Hall
4. McDougall, R. Mauro, J. (2006). Solaris Internals. 2nd ed.Prentice Hall

Appendix

Project source code

https://github.com/rhb2/CMPE108C

DTrace script for aggregating execution times

```
#!/usr/sbin/dtrace -s

pid$target::process_task:entry
{
        self->start = timestamp;
}

pid$target::process_task:return
{
        @latencies = quantize(timestamp - self->start);
}
```

DTrace script for tracking application system calls

```
#!/usr/sbin/dtrace -s

syscall:::entry
/pid == $target/
{
        @syscalls[probefunc] = count();
}
```