Thread Scheduling Data Structures: An Empirical Study

Harry H. Porter III

HHPorter3@gmail.com

30 August 2021

Abstract

How much does a red-black tree cost, compared to a linked list? Where is the cross-over point, where one is preferred over the other?

There are a number of algorithms to schedule threads in an OS kernel, using different data structures, including these approaches:

- Round-robin, using a linked list
- Round-robin, using a red-black tree
- Linked list, sorted by scheduling priority
- Red-black tree, sorted by scheduling priority
- Multiple linked lists, with one per scheduling priority

What are the relative costs associated with these different approaches?

A number of empirical tests were performed. This paper reports the results and conclusions.

Context

This discussion focuses entirely on scheduling "threads". We will not discuss address spaces or any larger concepts. We will not make a distinction between kernel and user threads. We will assume a single core.

We assume the existence of a single “ready list” (equivalently called a “ready queue”). This will be some sort of a collection, and our goal is to evaluate various implementations of this collection, such as linked lists and red-black trees. The elements in the ready list are all thread “objects”.

We assume that each thread is given a time-slice. The thread executes for a while until a timer interrupt occurs. The timer interrupt signals the end of the time-slice, at which time the “thread scheduler” is invoked. The scheduler will return the previously running thread to the ready list and select another thread from the ready list. The new thread will then begin its time-slice.

We measure the time spent in the scheduling—that is, between the time-slices—looking at the performance of the following data structures:

(1) Round-robin, with a linked list

This is the simplest scheduling algorithm. All threads have equal priority. There is a single ready list of thread objects, kept as a linked list. There are two operations, which we shall call “**getNext**” and “**insert**”. The list functions as a first-in-first-out (FIFO) queue. The **getNext** operation removes and returns the thread at the head of the list. The **insert** operation adds a thread to the tail end of the list.

Of course the size of the ready list—i.e., the number of threads waiting to be scheduled—will not impact the time for the **insert** and **getNext** operations.

(2) Round-robin, with a red-black tree

With this approach, the threads are scheduled in the same order as with the above. The difference is that we use a red-black tree instead of a linked list. A red-black tree must be sorted on something and, to achieve the first-in-first-out behavior, we can use a simple counter. The counter acts as the key and, for each insertion, the counter is incremented by 1. The **getNext** operation simply removes the thread with the lowest key.

Of course it’s a terrible idea to use a red-black tree for a FIFO queue! The insertion is always on the right side of the tree and the removal is always on the left. To keep the tree balanced, a fair amount of shuffling is required. On top of this, the algorithm for **insert** and **getNext** for a linked list is trivial, while the **insert** and **getNext** algorithms for red-black trees are complex, to say the least. So there is quite a bit of overhead… but how much?

Another question is: What impact does the size of the ready queue make?

(3) Linked list, sorted by scheduling priority

You probably want a scheduler that accommodates some sort of priority scheme.

When it comes time to **insert** a thread into the ready list, we will compute a "priority", which is just an integer. When it is time to select a thread for scheduling, our **getNext** operation will need to choose the thread with the highest priority.

With linked lists, we will keep the list sorted on thread "priority". The **getNext** operation will remove the thread at the head of the list, which has the highest priority. The **insert** operation will walk the list and insert the thread into the correct place in the list.

For this approach, we will need a doubly-linked list. The **getNext** operation is fast and constant-time, but the **insert** operation is not. As you know, searching a linked list for the correct insertion spot is time-consuming and gets worse as the size of the list grows. To be more precise, the **getNext** operation is linear in the size of the list, since you are required to walk half the list length on average.

(4) Red-black tree, sorted by scheduling priority

In this approach, we will schedule the threads in the exact same order as (3). The difference is that we will use a red-black tree instead of a doubly-linked list to implement our ready queue.

This is where red-black trees shine. The **insert** and **getNext** operations are more-or-less constant-time operations.

With a small number of threads, we expect the linked list approach to be fastest. But with a large number of threads, we expect the red-black tree approach to be faster. The big question is where does the cross-over occur? At what number of threads is the linked list approach superior and at what number is a red-black tree preferred?

We can't say here how many threads a real computer might be actively scheduling, but we assume it's more than 10 and less than 5,000. At this moment, my MacBook Pro is running about 1,500 threads, although some of them may be sleeping and the number of threads actively waiting on a ready list on any given core is probably much less.

I would venture to suggest that an upper limit of 100 threads per core might be a reasonable quesstimate for a laptop with a typical usage pattern.

(5) Multiple linked lists, with one per scheduling priority

You'll probably want to have many different priority levels and there is another way to achieve it.

Instead of having one giant red-black tree (or one frighteningly long linked list), our next proposed solution is to maintain a number (e.g., 10) of individual lists. For example, we might choose to limit ourselves to 10 priority levels, with a single linked list at each priority level to contain all threads running at that level. Each list will be a simple linked-list with FIFO scheduling. On other words, we schedule all threads at a given priority level using round-robin, which is fast and simple.

Then we must somehow decide which list to choose from. For example, we might choose the highest priority list and run several threads from it, using the basic FIFO algorithm to perform round-robin scheduling. Then, we may switch to the next highest-priority list and run several threads from it. And so on.

The beauty of this is that we can use the FIFO **insert** and **getNext** operations, which are fast. But we still have some overhead in switching from one queue to the next.

Another disadvantage is that we have a limited number (10 in our example) of priority levels, but this is really not much of a limitation. I am unable to think of 10 different tasks, each of which truly requires a different priority than all the others.

The question is what is the overhead for this approach?

Methodology

We are operating within the Blitz-64 system, running on a virtual machine emulator. We are using and modifying the existing thread scheduler. The emulator makes it convenient to capture the number of instructions executed.

We will measure time as a count of instructions executed. We ignore any differences in the speed at which different instructions may execute, assuming that each instruction takes the same amount of time. This seems reasonable for any RISC architecture.

Of course, a cache can speed execution, and the cache may have different effects on different pieces of code. Perhaps all scheduler instructions are always resident in the I-Cache, in which case the cache will be a constant-time speed-up of all instructions. If so, we can ignore the I-cache. The D-Cache may contain some but not all thread objects. Walking a linked list may visit many more thread objects than a red-black tree operation, but the red-black operation involves a fair amount of variable accesses. It is not clear which data algorithm would benefit more from the D-Cache.

In any case, we will ignore the effects of any cache.

For our testing, we will be creating a number of threads and then running them for a while. To reduce start-up effects, we will run for 5 seconds, then stop and print the instruction counts.

Each individual thread will execute an infinite loop. This loop will do nothing but a “yield” operation. In other words, when given a time-slice, the thread will immediately sacrifice the remainder of it to other threads. Thus, the actual time-slices will be very small and the bulk of the 5 seconds will be spent in the scheduler itself.

Our goal is to compare different data structures used with the scheduler, in an attempt to reduce the overhead of thread switching. But the amount of overhead imposed on computation by a thread

scheduler is a different question. The overhead is dependent on the average time-slice size, and the time-slice size can be adjusted as desired to change the percentage of computation spent scheduling. The best choice for time-slice sizes is a question of the desired responsiveness of the computer, given a particular workload and processor speed. We do not address those questions here.

Whenever a time-slice ends, the scheduler is invoked. Roughly speaking, the scheduler performs an **insert** operation to put the previously executing thread back into the ready list. Then the scheduler executes a **getNext** operation to find and remove the next thread to be scheduled. So the computer executes a series of time-slices, each separated by an invocation of the scheduler. The scheduler is invoked once per time-slice.

To determine how much time is spent in the scheduler itself, the code will note the instruction count at the time the scheduler begins and again at the time the scheduler ends. In addition, we count the number of times the scheduler is invoked which, of course, will be once per time-slice. We add all these times together and divide by the number of time-slices to compute:

The average number of instructions per scheduling operation

In addition to the **insert** and **getNext** operations which we are interested in, the scheduler also has a fair amount of overhead, which includes interrupt processing, register and state saving, misc accounting, and loading and initiating a new time-slice. However, all this additional overhead is constant-time. In other words, of the reported number of instructions per time-slice, a certain amount of overhead will always be included, but the overhead is independent of the actual **insert** and **getNext** scheduling operations.

The Blitz scheduler also contains a facility for real-time threads. However, in these tests, there are no real-time threads. On each invocation, the scheduler will check about real-time scheduling but, since there are never any such threads, the tests will always fail. The effect will be to add a few instructions to the count but, like the other sources of overhead, this will be a constant time overhead. Therefore, we can be ignore the overhead associated with real-time thread scheduling, just as we ignore the other overhead.

To eliminate noise, all threads in the kernel have been disabled. Even the “idle thread” was modified so that it will not execute during the test. The idle thread will get exactly one time-slice for initialization and then will print an error if it should ever get a second time-slice, to verify that it is not running. All other kernel daemon threads have been completely disabled.

A “test” will begin with a single “master” thread, which will create and initiate a number of “target” threads. The master thread will then sleep for 5 seconds. When it wakes up, the master thread will collect and print the instruction counts.

Interrupts are disabled during the master thread, so that it can complete all kernel initialization and can start all the target threads without invoking the scheduler. After the target threads have been created and added to the ready list, the master thread sleeps for 5 seconds. This “sleep” operation terminates its time-

slice and allows the test to begin. When the master thread wakes up, it completes its reporting without any further time-slicing.

Timer interrupts will not happen, since each time-slice will be terminated prematurely when the target threads perform the “yield” operation. The computer will experience no interrupts or exceptions during our testing runs, which might introduce noise into our timing results.

I wrote all the code involved in these tests, including the code for the red-black tree operations and the linked-list operations which are to be compared. The compiler performs basic optimizations on the code and the same level of optimization is used for all tests. Obviously, increasing the level of compiler optimization will speed execution up, but I expect all additional performance improvements from increased optimization to be constant-time. In other words, I expect optimized code to run faster, in the same way the code would run faster on faster hardware. I would not expect compiler optimization to favor any particular algorithm, or alter the conclusions we will draw.

It is certainly possible to hand-code the linked list operations directly in assembly code and this might increase the performance of the linked-list **insert** and **getNext** operations, at least compared to minimally-optimized compiler-generated code. It is not reasonable to hand-code the red-black tree operations in assembly code, so we might be able to give the linked-list operations an increased benefit if we were to compare hand-coded assembly for linked list operations to a compiler-generated, minimally-optimized version of the red-black tree operations. But in the end, we expect all scheduler code to be fully optimized compiler-generated code, so any discussion of hand-coded assembly language is irrelevant.

All code for these tests is being emulated. The emulation is at the Instruction Set Architecture (ISA) level. Each instruction is considered to take an identical amount of time and there is no pipelining. Real hardware functions differently in order to obtain greater performance. But I do not expect such differences to impact our conclusions.

When we measure time (as in “This test was run for 5 seconds”), what we actually mean was that the test was run for 750,000,000 instructions, which takes about 5 seconds on the emulator.

Dynamic Research

I am writing this paper at the same time I am performing the experiments described herein. While I have the testing setup more-or-less completed, as of this moment, I have not collected the numbers. So at this point, I know as much as you about what the rest of this paper will say.

Test #1

We start with scheduling algorithm (2), which is a red-black tree being used for strict round-robin scheduling.

We start with 10 threads and vary the time of the test.

Test Length	Number of Time-Slices	Average # of Instructions per Time-slice
1 second	46,751	2740.529742679
5 seconds	233,754	2740.505736800
10 seconds	467,508	2740.502881234
15 seconds	701,263	2740.501922959

This gives us confidence that the test results are consistent. Under these conditions, we are observing about 2,740 instructions spent in the scheduler code, regardless of how long we run the test.

Henceforth, we will truncate our results to whole numbers. We will also perform future tests with a run of 5 seconds.

Henceforth, we will not report the number of time-slices either. The number of time-slices we can perform in 5 seconds will be a large number. For example, if the scheduler takes twice as long (2740 × 2 = 5480), then we would only be doing about half as many time slices (233,754 ÷ 2 = 116,877). This should still be enough to give us a good approximation to the time required for the average scheduling operation.

As mentioned above, a "5 second" test run is actually a run for 750,000,000 instructions on the emulator. Given that there were 233,754 time-slices in the 5 second test, we can see that the average time between time-slices is 750,000,000 ÷ 233,754 = 3,208.5 instructions. Above we show a "time per time-slice" of 2,740. The difference (i.e., 468.5 instructions) is consumed by state saving, context switching, and the computation actually done within the target thread. The number we report (2,740) is the number of instructions spent in the code that performs the scheduling. We are concerned with the **insert** and **getNext** operations, but the scheduling code does a number of additional tasks unrelated to these operations. This additional work acts as a fixed overhead, which consumes the same number of instructions for each time-slice. For example, there is code to give real-time threads processor time, but since there are no real-time threads, those checks always fail. There is also code to perform accounting and some additional activities. While these extra activities cannot be easily removed, they are a fixed cost per time-slice, just like the 468.5 instructions.

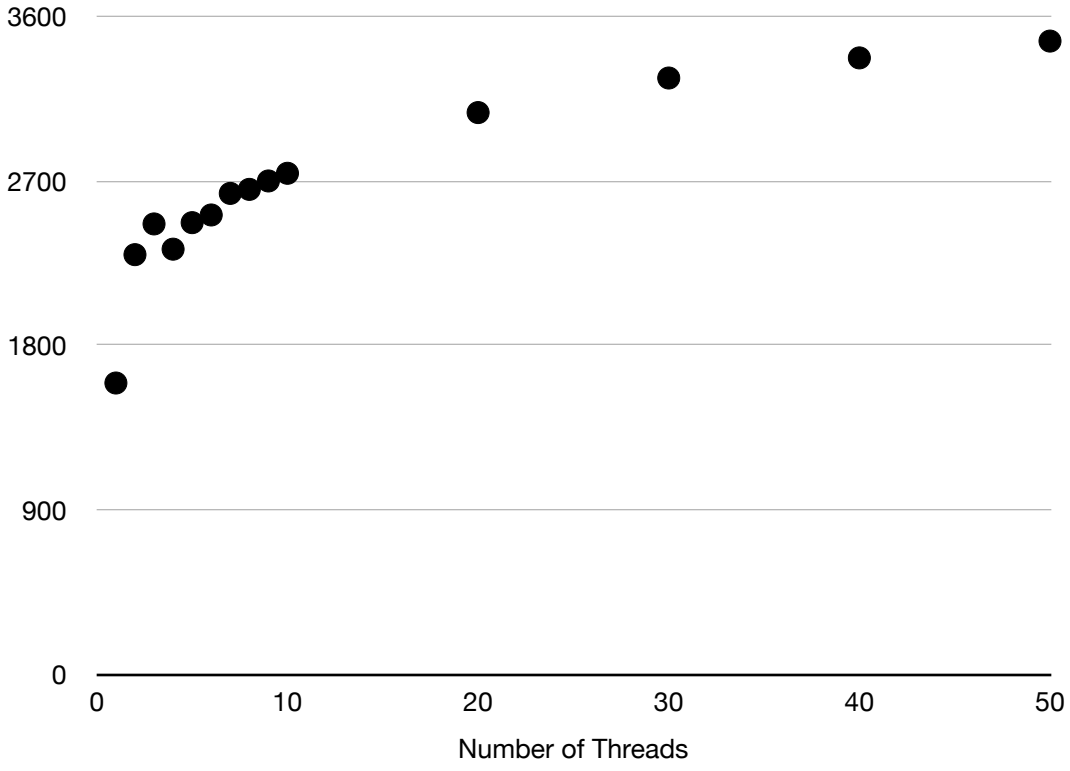
I do not know how to accurately measure the fixed cost per time-slice, but it doesn't matter, since it is constant and unchanging, regardless of which algorithms we use for **insert** and **getNext**. So in the following graphs, please remember the fixed cost is there. In other words, don't look at the zero axis; the actual height above the zero line is arbitrarily determined by the fixed cost.

Test #2

Sticking with algorithm (2), let's vary the number of threads.

Number of Threads	Average # of Instructions per Time-slice
1	1593
2	2295
3	2463
4	2325
5	2470
6	2512
7	2630
8	2652
9	2698
10	2740
20	3072
30	3261
40	3371
50	3463

These numbers form a nice graph, with the values asymptotically approaching about 3600, although the graph is a little chaotic and noisy with under 8 threads.



Test #3

Now let's see what happens with algorithm (4). In other words, we are continuing to use a red-black tree but we are no longer using it in FIFO order. We will be using random numbers to determine the priorities, which means that threads are not added to the tail end of the queue, but are inserted into different places in the red-black tree.

We expect that the results will be about the same, or perhaps slightly smaller.

First, let's evaluate our randomizing function with 30 threads. We'll call our various randomizing functions A, B, C, ...

Randomization	Average # of Instructions per Time-slice
A: 200-200	3261 (no randomization; strictly round-robin)
B: 200-250	2940
C: 200-300	2857
D: 200-400	2795
E: 200-500	2775
F: 200-1000	2740
G: 200-2000	2713
H: 1-10	2896
I: 1-100	2442
J: 1-200	2231
K: 1-500	2105
L: 1-1000	2040
M: 1-2000	2531 <<<
N: 1-3000	1999

It seems that the numbers seem to get smaller with increasing randomization. This makes sense: If the thread just finishing a time-slice is always inserted in round-robin fashion, at the very tail end of the ready list, then the entire tree must be shifted in order to maintain balance. On the other hand, if the thread is inserted in a random spot, it will tend to be inserted in the middle of the tree, which will only require a rebalancing of about half of the tree.

But this priority scheme not realistic. For example, "N" assigns each thread a priority number between 1 and 3,000. A thread with priority 3,000 is run 1/3000 as often as a thread with priority level 1. But does anyone really need such a great variation of priority levels? And with so few threads, the assignment of random priority levels is itself introducing a large degree of randomness.

We'll use a different approach. We will randomly assign each thread a priority between 1 and 5 and each thread will keep its priority unchanged throughout the test. The threads with priority level 1 are the highest and run most frequently. The threads with priority level 2 are run at half the speed of threads at level 1. The threads at level 3 are run 1/3 as often as threads at level 1. Threads at level 5 are run 1/5 as

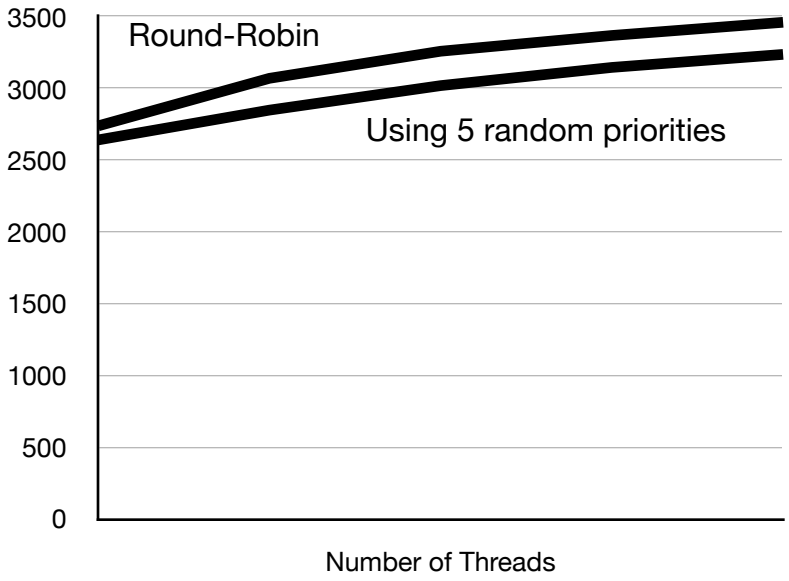
often as threads at level 1. Thus, threads at level 5 have only 1/5 as many time-slices. Threads at level 1 tend to be inserted near the low end of the tree—that is, near the front end of the ready list—and threads with a higher number tend to be inserted closer to the tail end of the tree.

All the threads with a given level are scheduled round-robin. For example, if you look only at the threads at level 3, you'd see they are scheduled in a strictly round-robin order. Thus, threads at level 5 are always inserted at the tail end of the ready list.

Let's vary the number of threads and see what happens with 5 priority levels. For comparison, we show the round-robin times—that is, where there is just a single priority level. We also show how much faster the random threads are ("difference").

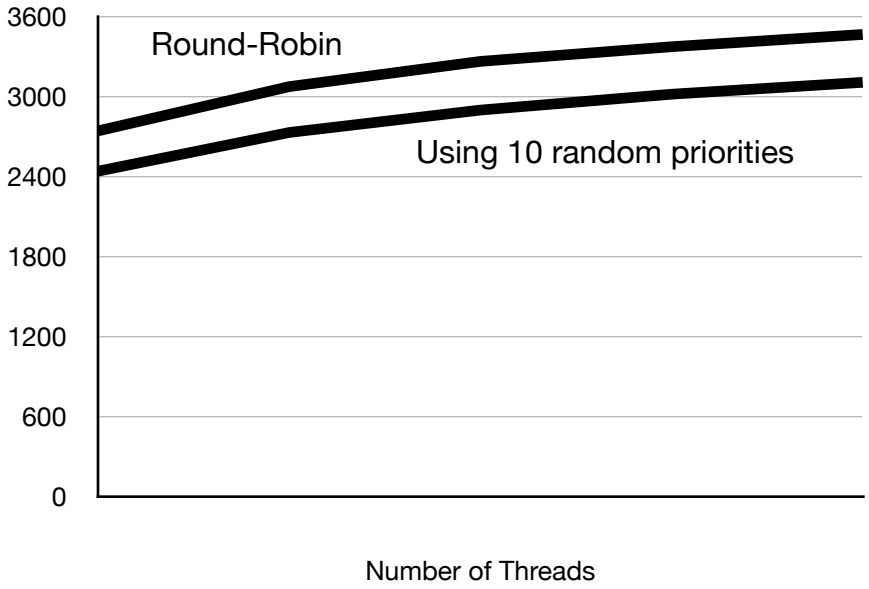
Number of Threads	Random Priorities (5)	Round-Robin (1)	Difference
10	2642	2740	98
20	2850	3072	222
30	3021	3261	240
40	3148	3371	223
50	3239	3463	224

That's a pretty consistent result: Using just a red-black tree in both cases, increasing the number of priorities tends to reduce the overhead spent by a small and roughly constant amount.



Let's try a larger number of priorities and see what happens. Next, we will randomly assign each thread with a priority between 1 and 10. Threads at level 10 will be scheduled 1/10 as often as threads at level 1.

Number of Threads	Random Priorities (10)	Round-Robin (1)	Difference
10	2438	2740	302
20	2728	3072	344
30	2896	3261	365
40	3015	3371	356
50	3104	3463	359

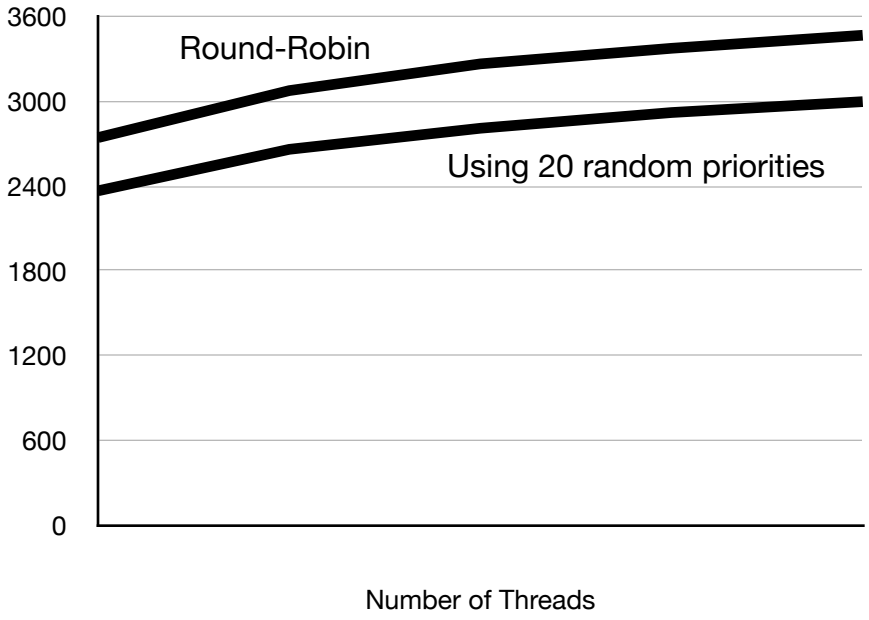


Again, we get a pretty consistent result. Randomizing the threads causes the time spent scheduling to decline, and the decline is by a roughly constant amount, independent of the number of threads being scheduled. Of course, we always spend more time scheduling when we have more threads.

And we find that a larger number of priority levels reduces the scheduling time a little, but the amount is modest. We can call it about 360-220 = 140, which is small when compared to the difference in times for scheduling say 20 threads and 50 threads, which is 3104-2728 = 376.

The bottom line—at least with red-black trees—is that there is no penalty for having a larger number of priority levels. Let's test this by looking at 20 different priority levels.

Number of Threads	Random Priorities (20)	Round-Robin (1)	Difference
10	2364	2740	376
20	2657	3072	415
30	2806	3261	455
40	2917	3371	454
50	2994	3463	469

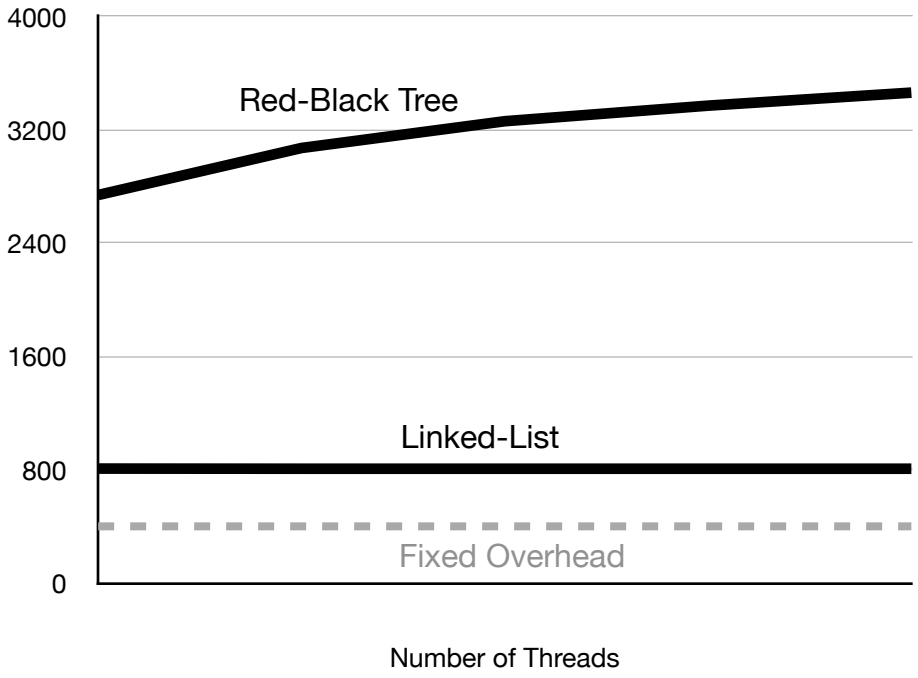


So the trend seems consistent.

Linked Lists

Next, we let's look at scheduling times for linked lists. We begin with straight round-robin scheduling. We compare it to the red-black implementation with only one priority level, which also implements round-robin scheduling.

Number of Threads	Linked List	Red-Black Tree
10	808	2740
20	807	3072
30	807	3261
40	807	3371
50	807	3463



We see that the linked-list implementation is dramatically faster for round-robin scheduling. We also see that the time spent in **getNext** and **insert** is always the same, regardless of the number of threads.

As mentioned before, we are not able to quantify the fixed, constant-time costs associated with each scheduling event. There is some amount of constant overhead is included in these numbers, which we have suggested with the dashed line, although we do not know exactly how much it is. Wherever the dashed line is actually located, it forms the baseline, on top of which the **getNext** and **insert** operations are added.

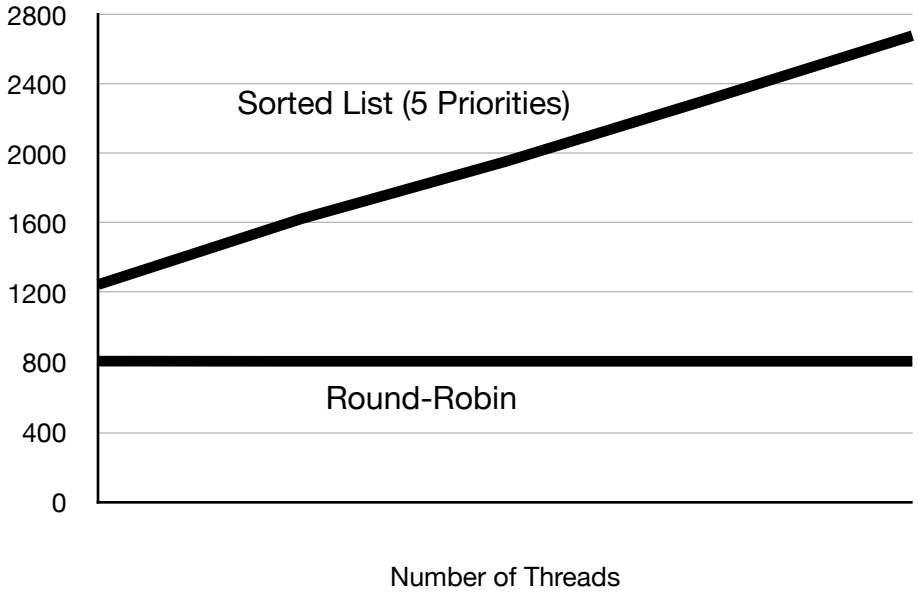
Regardless of how much overhead our measurements include, we conclude that the savings is dramatic. But of course, we always expected linked lists to be much faster whenever the scheduling is strictly round-robin.

Sorted Linked Lists

Next, let's look at priority scheduling with linked lists. This will require us to perform a sorted-insert when we reschedule.

Here, we compare linked-list against linked-list. In one case, there are 5 priority levels, and in the other there is just 1 priority.

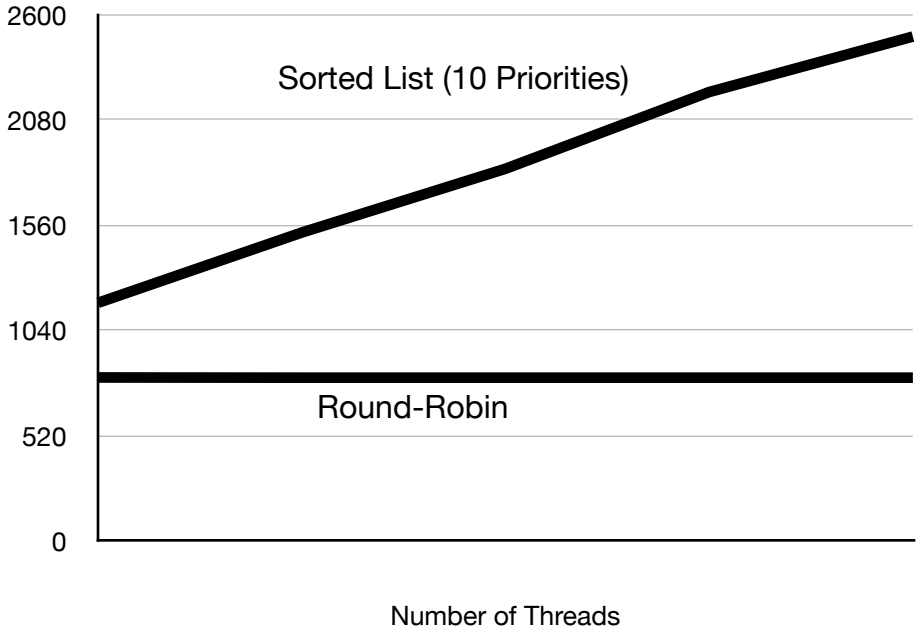
Number of Threads	Random Priorities (5)	Round-Robin (1)	Difference	Delta
10	1248	808	440	
20	1627	807	820	379
30	1955	807	1148	328
40	2313	807	1506	358
50	2679	807	1872	366



Just a we expect, the time spent grows with an increasing number of threads. The "delta" column shows the extra time that each scheduling event requires, when we add 10 more threads. For example, to schedule 30 threads with random priorities required an average of 1955 instructions, while to schedule 40 threads with random priorities requires 2313 instructions. That's a delta of 358.

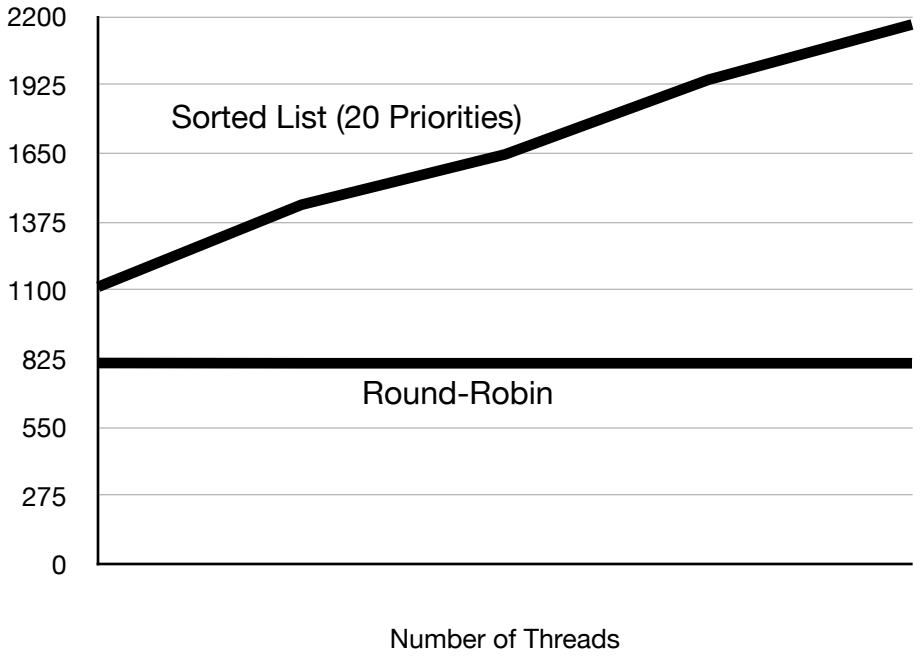
Next, let's look at 10 priority levels.

Number of Threads	Random Priorities (10)	Round-Robin (1)
10	1177	808
20	1524	807
30	1839	807
40	2218	807
50	2492	807



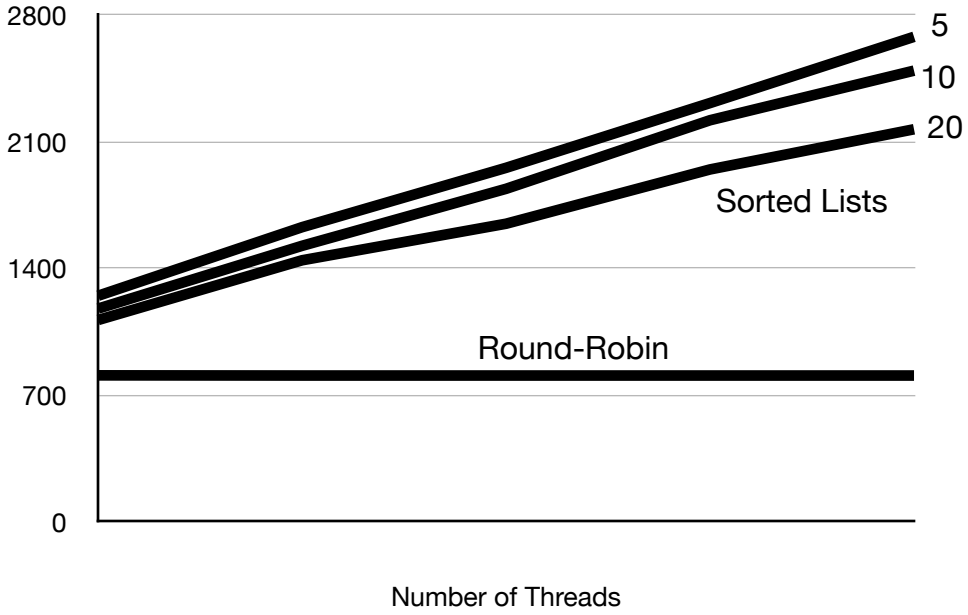
Next, let's look at 20 priority levels.

Number of Threads	Random Priorities (20)	Round-Robin (1)
10	1114	808
20	1444	807
30	1646	807
40	1946	807
50	2168	807



As before, we see that an increase in the number of priority levels seems to slightly reduce the cost of scheduling. For example, scheduling 50 threads with 5 priority levels requires 2679 instructions, but if we increase the number of priority levels to 20, the time goes down to 2168.

Combining the previous three graphs, we get this:



As expected, we see that increasing the number of threads causes the time spent in the scheduler to go up, more or less linearly.

The Big Question

So the question in comparing the linked list implementation to the red-black tree implementation is this:

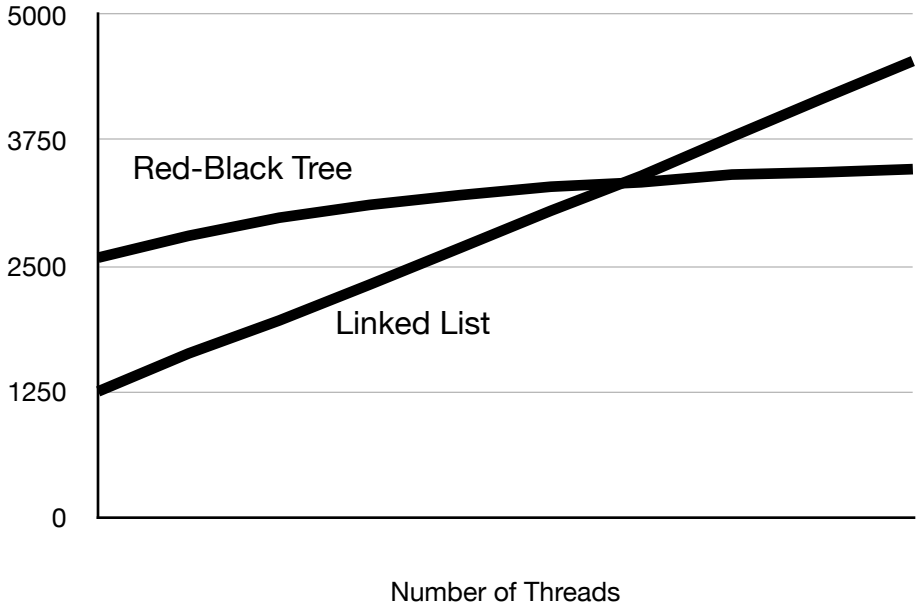
How many threads are required before the red-black approach is more efficient?

When I implemented the linked-list version, I dealt with the "idle thread" in a different way. I now realize that the previous numbers I got for the red-black tree all included the idle-thread, which was always in the tree, although it never came to the front of the queue. This makes the previous numbers for red-black trees incomparable to the numbers for the linked-list. So, treating the idle thread the same way, I have revised the timings for the red-black trees.

Also, in order to find the cross over point, we'll need to look at more threads.

Let's start with 5 priority levels:

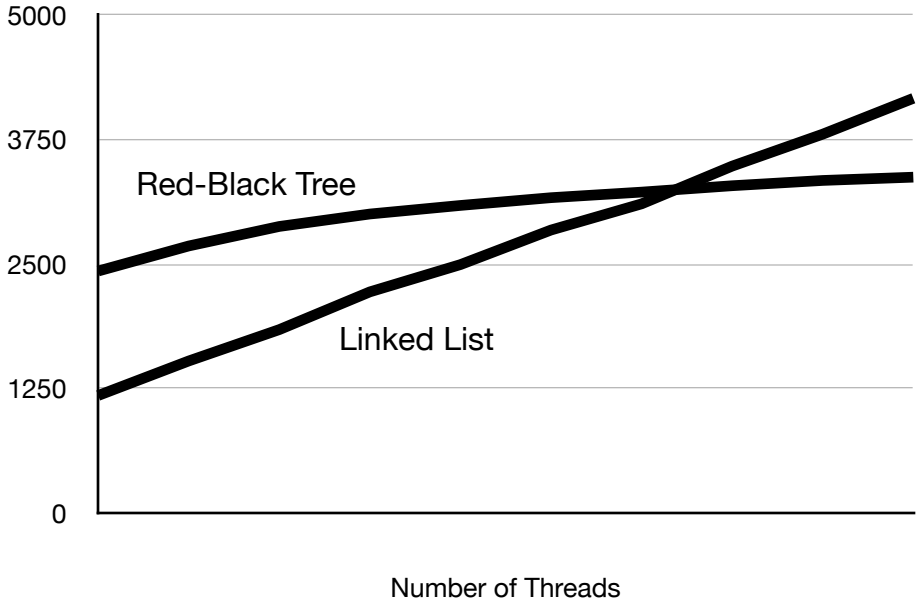
Number of Threads	Linked List	Red-Black Tree
10	1248	2581
20	1627	2799
30	1955	2979
40	2313	3107
50	2679	3204
60	3048	3288
70	3396	3332 <<<
80	3785	3409
90	4166	3431
100	4540	3463



The log curve for the red-black tree algorithm crosses the linear curve for the linked list algorithm. Above 60 threads, the red-black approach is faster.

Here is the data for 10 priority levels:

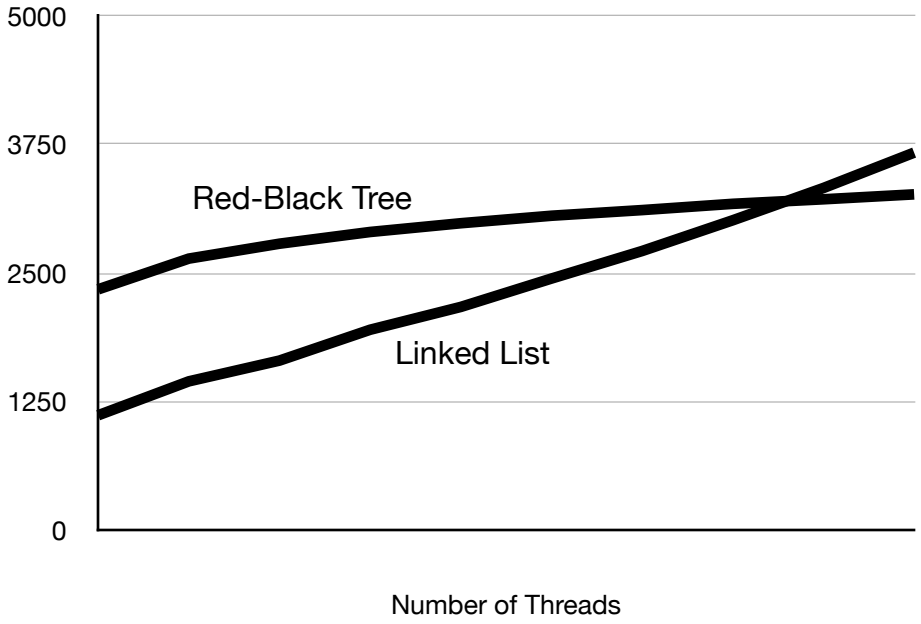
Number of Threads	Linked List	Red-Black Tree
10	1177	2427
20	1524	2679
30	1839	2875
40	2218	3000
50	2492	3086
60	2839	3164
70	3102	3220
80	3481	3284 <<<
90	3802	3337
100	4162	3370



The crossover point has moved to the right. Above about 70 threads, the red-black approach is superior.

Here is the data for 20 priority levels:

Number of Threads	Linked List	Red-Black Tree
10	1114	2339
20	1444	2638
30	1646	2783
40	1946	2897
50	2168	2982
60	2444	3055
70	2712	3110
80	3011	3171
90	3325	3215 <<<
100	3669	3263



As you can see, the curve for the red-black trees looks like a log plot and the curve for the linked-list looks linear, which is exactly what we expect. We see clear cross-over points, which seem to be between about 70 and 90 threads. The exact cross-over point is dependent on the number of priority levels we are using.

The threads in these simulations are assigned randomly. This may not be a good model for a real mix of threads. It may be that we have a large number of low priority threads and a small number of high priority threads. Or perhaps it the opposite mix is more normal; I really don't have a good intuition about this.

But one thing we can say is that the red-black curve is virtually flat at the cross over point, while the link-list curve remains linear. This means that very quickly after the cross-over point—wherever it occurs—the two diverge quickly and the linked-list becomes awfully slow.

I suppose lightly loaded systems tend to have a small number of threads and heavily loaded systems tend to have a larger number of threads, although this is by no means obvious. In any case, we are most concerned about efficiency in a heavily loaded system.

If the computer almost always has 50 or fewer active threads, then we conclude that the linked-list is the winner and is an acceptable choice.

But if the computer often has more than 100 active threads, then the red-black tree is the clear winner.

Between these two numbers, it is unclear. To answer that, we would need to look at the actual priority mix for a real system.

My guess is that my MacBook Pro, with its 1500 threads, has a lot of threads that are sleeping. I just can't imagine what background tasks 1500 threads might be actively engaged in computing. I'll bet many threads wake up periodically to check for some condition, such as incoming email. Maybe other threads only wake up when an event occurs, such as a particular button being pressed or some other thread requiring a service. But most of the time, my laptop is just doing nothing.

Of course, we are not concerned about efficiency when the laptop is doing nothing; we care about efficiency when we have a bunch of things happening. For example, what if I push a button at the same moment that the email daemon wakes up, and while a spell-checker is looking for misspellings, and while a video player is playing in a separate window? Maybe there is a virus scanner operating in the background and maybe the disk head algorithm is busy trying to reorder disk requests and maybe a couple of tasks have garbage collectors running in the background. That is about 10 activities. Still, I find it difficult to imagine as many as 20 simultaneously executing threads, each actively computing something. And even if we happen to have 20 active threads at one moment, I cannot imagine that level of congestion will occur often or will persist for very long.

However, in different environments, I can imagine many more active threads. Consider a robot with 50 joints. Each joint requires a thread which is reading a sensor, performing a computation, and adjusting a signal to a servo motor. Or consider an inflight media system, where each seat in the airplane involves an individual thread. Each thread is actively computing the data to display on the screen, showing the location, airspeed, and so on. Or consider a factory sensor system, where each thread is reading a sensor and computing results based on the current conditions (which are always in flux) to determine if the sensor reading is within parameters.

We want these threads to run as fast as possible, which means we need the scheduler to have the lowest possible overhead.

One thing to remember is that we can always cut the scheduler overhead in half by doubling the size of each time-slice. If the time-slices are twice as long, then the scheduler will only need to select a new thread half as often. Therefore, it will consume half as many cycles. While each thread will ultimately get the same amount of CPU cycles per minute (not accounting for the reduced overhead), each thread

will get a time-slice half as often. So the thread gets a time-slice half as often, but the time-slice is twice as large. This means that a thread may have to wait for twice as long between time-slices.

In the case of a thread that is monitoring a sensor, the long time-slice might not be as important as having frequent time-slices. Perhaps on each time slice, the thread takes a reading, checks it, and then yields the remainder of its time-slice. The checking might not take much time, but the thread needs to perform these checks at a high frequency, in order to catch and respond to a problem quickly.

Linked List Search Direction

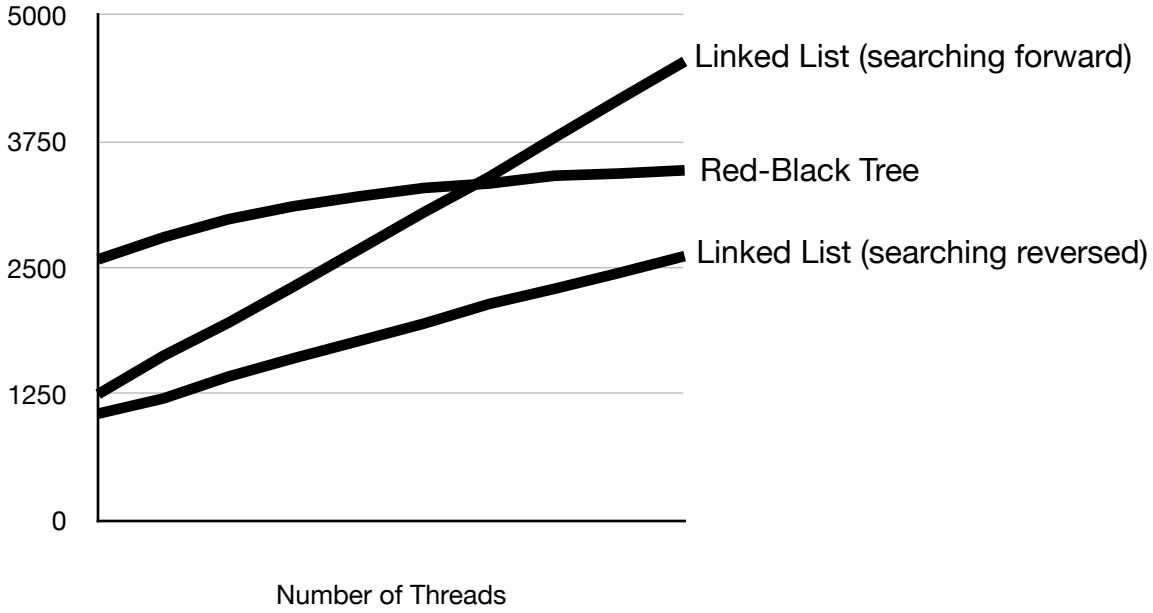
Previously, to perform an insert with a linked list, we searched from front to back. When the list contains a thread with the same key, we continued searching until we had searched past it, before performing the insert. What happens if we search in the reverse direction? In other words, what happens if we walk the list from the tail end? If there are several threads on the ready list with the same key then, by searching in the reverse order, we can avoid walking past all the equal entries. Will this save us any time?

In the next experiment, we compare the forward search with the reverse search. For comparison, we show the red-black tree numbers. This should make the biggest difference when we have a large number of threads with a small number of priority levels. For example, if we have 100 threads with only 5 different priority levels then, on average, there are 10 threads at each priority level. Thus, whenever we insert a thread into the ready queue, the queue already contains 9 threads with the same priority level.

We are curious whether the search direction makes a significant difference in the cross-over point, when we compare it to red-black trees. So we'll include the red-black timings for comparison. The cross-over points, where the red-black tree performance is superior, are marked.

Let's start with 5 priority levels:

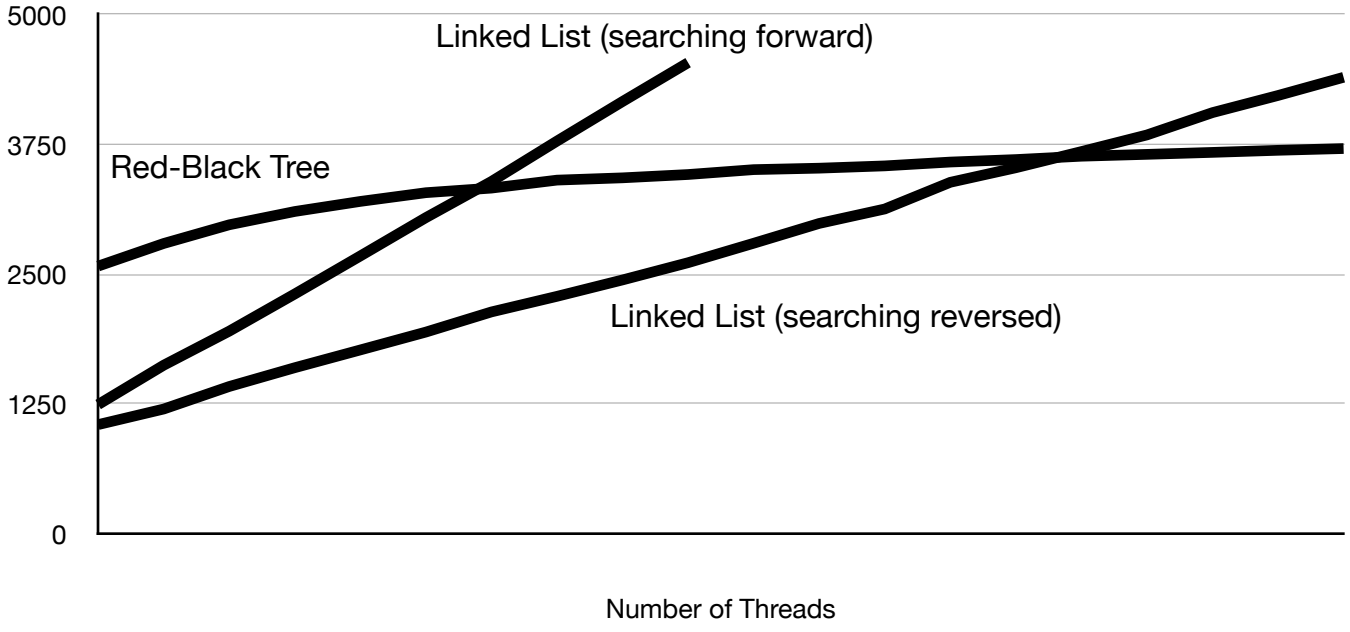
Number of Threads	Linked List (forward)	Linked List (reverse)	Red-Black Tree
10	1248	1055	2581
20	1627	1205	2799
30	1955	1422	2979
40	2313	1603	3107
50	2679	1774	3204
60	3048	1946	3288
70	3396 <<<	2140	3332
80	3785	2290	3409
90	4166	2448	3431
100	4540	2614	3463



Wow! The linked list—when searched in reverse order—is outperforming red-black trees even with 100 threads.

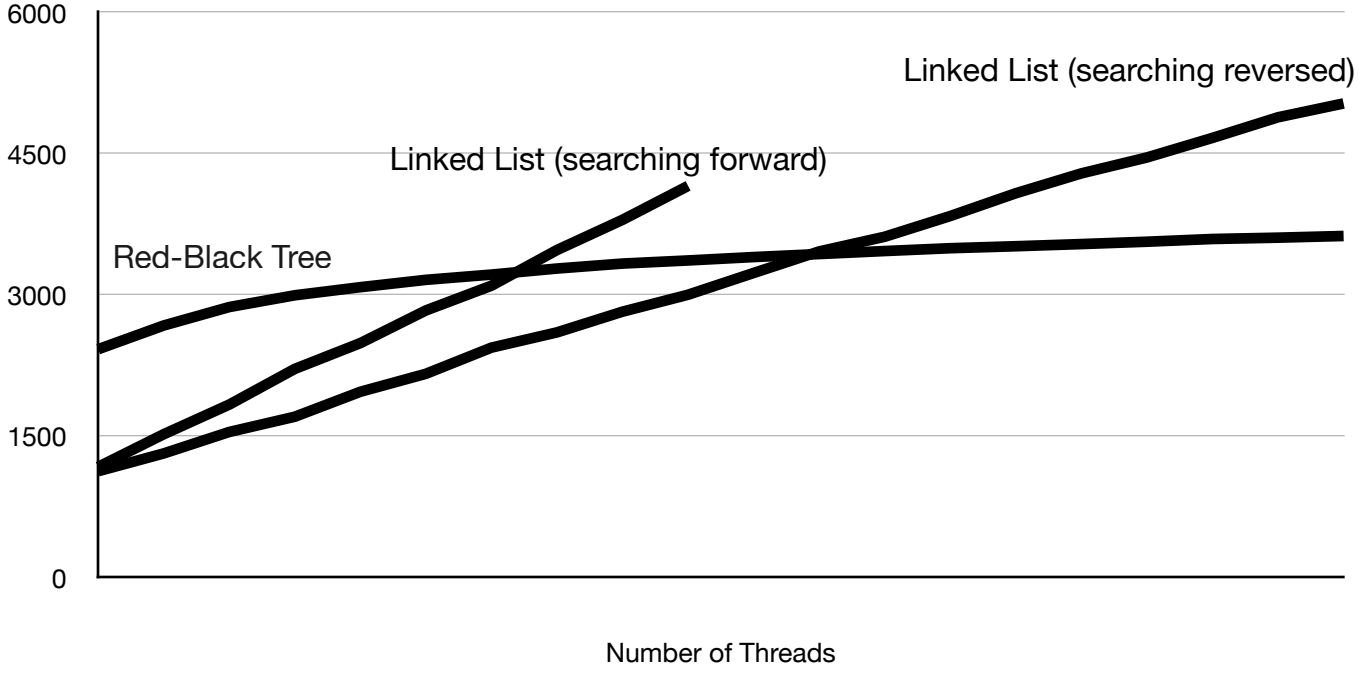
Let's keep going and test with more threads. I'll have to go back a run some more red-black tree tests to find the cross-over point.

Number of Threads	Linked List (forward)	Linked List (reverse)	Red-Black Tree
10	1248	1055	2581
20	1627	1205	2799
30	1955	1422	2979
40	2313	1603	3107
50	2679	1774	3204
60	3048	1946	3288
70	3396 <<<	2140	3332
80	3785	2290	3409
90	4166	2448	3431
100	4540	2614	3463
110		2801	3510
120		2992	3524
130		3131	3546
140		3388	3583
150		3528	3607
160		3684 <<<	3639
170		3844	3657
180		4059	3676
190		4224	3696
200		4399	3713



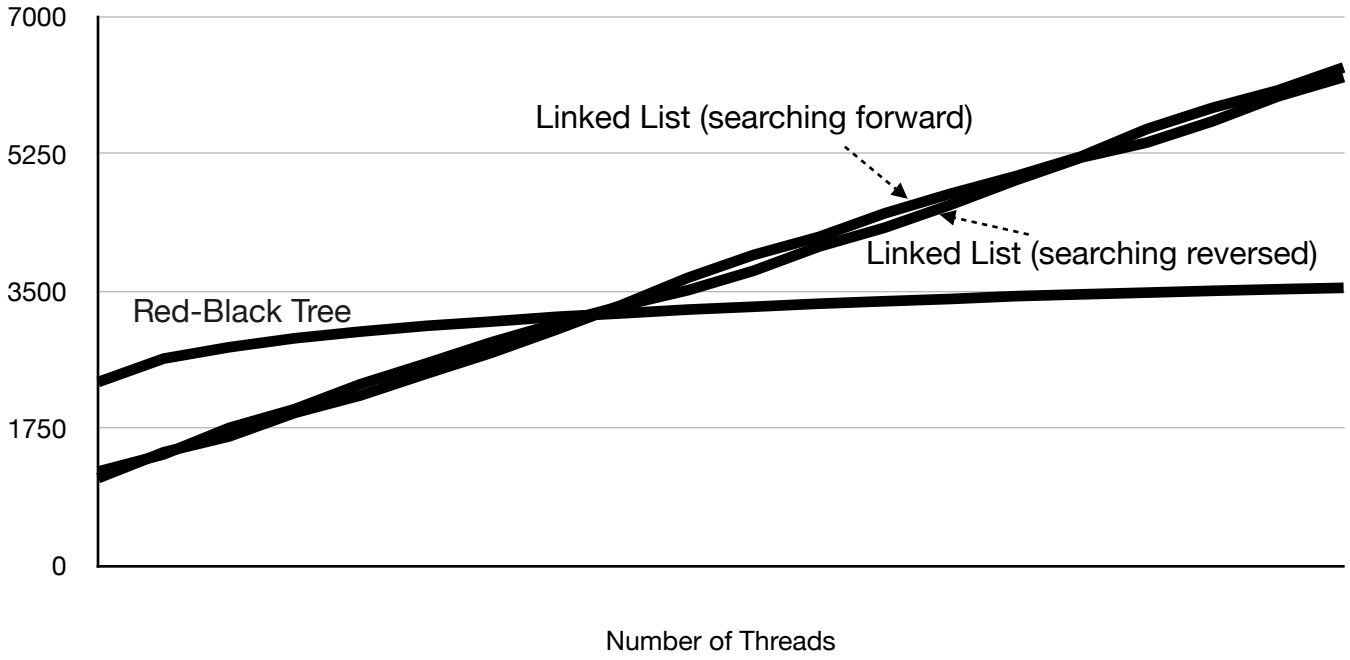
Here is the data for 10 priority levels:

Number of Threads	Linked List (forward)	Linked List (reverse)	Red-Black Tree
10	1177	1131	2427
20	1524	1320	2679
30	1839	1548	2875
40	2218	1710	3000
50	2492	1975	3086
60	2839	2166	3164
70	3102	2444	3220
80	3481 <<<	2606	3284
90	3802	2826	3337
100	4162	3006	3370
110		3237	3406
120		3466 <<<	3438
130		3622	3470
140		3842	3500
150		4084	3521
160		4295	3544
170		4464	3569
180		4672	3598
190		4891	3612
200		5038	3630



Here is the data for 20 priority levels:

Number of Threads	Linked List (forward)	Linked List (reverse)	Red-Black Tree
10	1114	1198	2339
20	1444	1417	2638
30	1646	1755	2783
40	1946	1994	2897
50	2168	2311	2982
60	2444	2574	3055
70	2712	2846	3110
80	3011	3087	3171
90	3325 <<<	3313 <<<	3215
100	3669	3510	3263
110	3960	3758	3299
120	4190	4067	3338
130	4490	4307	3369
140	4739	4598	3398
150	4964	4913	3434
160	5216	5202	3457
170	5565	5392	3479
180	5835	5663	3502
190	6058	5981	3522
200	6350	6227	3540



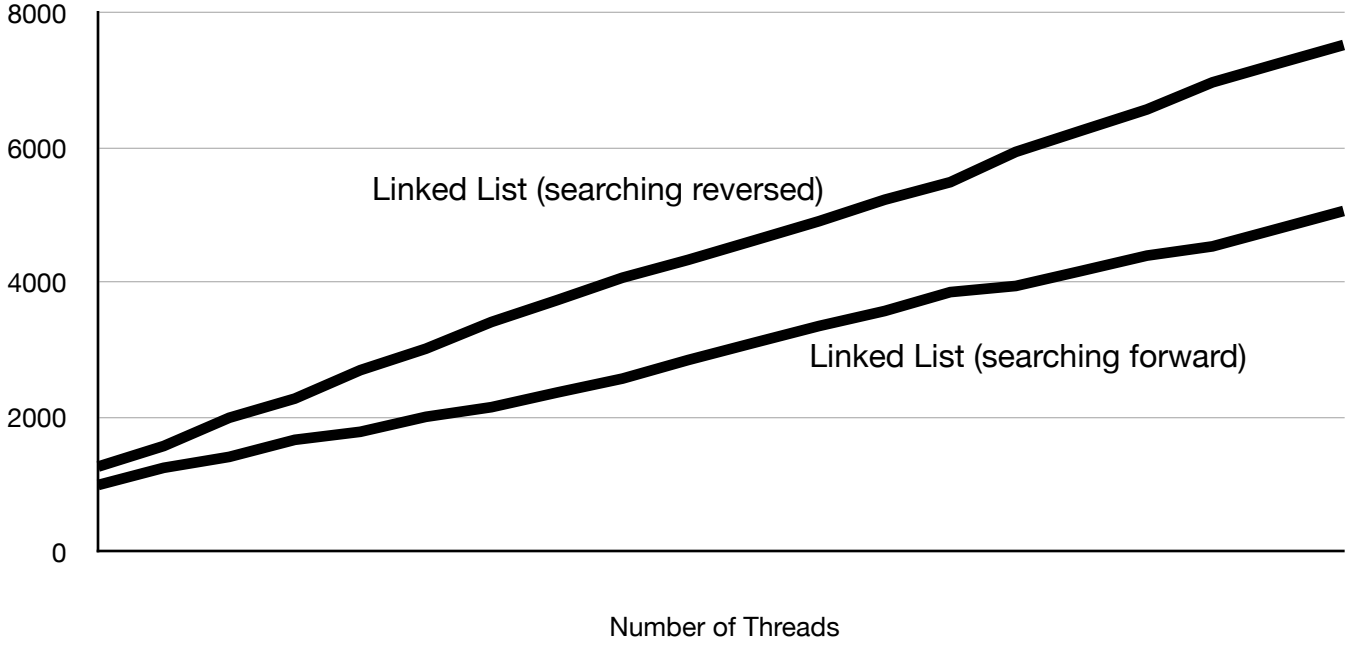
So we can conclude that with a lot of priority levels, if you are using the linked-list implementation, it doesn't matter whether you search forward or backward. But it you have a smaller number of priority levels, then it really pays to search from back-to-front.

With an even larger number of priority levels, I expect the search order not to matter. You can search the linked list front-to-back or back-to-front; the problem is that you have a lot of elements of different priorities that you have to skip over and these account for more of the time.

But just to make sure, I'll run the linked list tests for 50 priority levels. If there is some unexpected effect, that should reveal it.

Here is the data for 50 priority levels:

Number of Threads	Linked List (forward)	Linked List (reverse)
10	997	1270
20	1252	1575
30	1414	1994
40	1667	2278
50	1786	2700
60	2008	3018
70	2151	3414
80	2368	3737
90	2575	4071
100	2850	4335
110	3104	4621
120	3355	4909
130	3577	5227
140	3857	5487
150	3948	5936
160	4171	6252
170	4398	6565
180	4535	6967
190	4797	7246
200	5060	7521



OK, that's not what I expected. But that's why we collect data!

With a large number of priority levels, searching from front-to-back is clearly preferable to searching from back-to-front.

I assume the reason is that the threads that have high priority run a lot and require a lot of insertions. But since they are high-priority threads, they will tend to be inserted near the front of the ready queue, so it makes sense to search the linked list from front-to-back.

Keep in mind that our priority model is that a thread with a priority level of 50 is very low priority and runs at 1/50 the speed of a thread at level 1. With a large number of priorities, we will have a lot of threads that are running very rarely, like 1/50-th of the time, or 1/47-th of the time. So with 200 threads, we've got a lot of threads that are sitting on the ready queue that get selected almost never. Every time we search from back-to-front, we have to skip past all these more-or-less dead threads. So this explains why searching from back-to-front is a bad idea with this many priority levels.

The Initial Priority System

In the previous experiments, we were using a number of different priority levels. We asked about threads distributed over 5, 10, 20, and even 50 different priority levels.

Up to now, our priority scheme has worked as shown below.

Imagine that we have one thread running at each priority level. In this diagram, there are threads named A, B, C, …, each running at a different priority level. Each letter indicates a single time-slice for that thread.

Time moves left-to-right. For example, thread C runs 1/3 as often as thread A. Likewise, thread H runs at 1/8 the speed of thread A.

```
1  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
2   B B B B B B B B B B B B B B B B B B B B B B B B B
3    C  C  C  C  C  C  C  C  C  C  C  C  C  C  C  C  C
4     D   D   D   D   D   D   D   D   D   D   D   D
5      E    E    E    E    E    E    E    E    E    E
6       F     F     F     F     F     F     F     F
7        G       G       G       G       G       G       G
8         H         H         H         H         H         H
```

Time slices are executed one-after-the other, so the actual, linearized order of scheduling is:

```
A AB AC ABD AE ABCF AG ABDH AC ABE A ABCDF A ABG ACE ABDH ABCF ...
```

You can see that between each C there are 3 As. Likewise, between each H there are 8 As.

Of course we may have several threads at one priority level. For example, we may have 2 threads at priority 2, which we can call B_1 and B_2. Then our scheduling order is:

A AB_1B_2 AC AB_1B_2D AE AB_1B_2CF AG AB_1B_2DH AC AB_1B_2E A AB_1B_2CDF A ...

A Better Priority System

I feel this approach used so far gives the scheduler a much finer granularity of priority than we actually require: there are too many priority levels.

Clearly, there is a big difference between thread A and thread B which runs at 1/2 the speed of A, and this is useful. But it is unclear why you would need one thread to run at 1/21 the speed of thread A and another thread running at 1/22 the speed of A. Do we really need two distinct priority levels for these two threads? I think it is quite reasonable to simply group them into the same priority level.

So next we describe a different priority scheme.

Now we will have different priority levels, where a thread at one level will run at *half the speed* of a thread at the level above it. And let's start our level numbers at 0 instead of 1.

```
0   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
1    B B B B B B B B B B B B B B B B B B B B B B B B
2      C   C   C   C   C   C   C   C   C   C   C   C
3          D       D       D       D       D       D
4              E               E               E
```

When linearized, we get this order:

A AB A ABC A AB A ABCD A AB A ABC A AB A ABCDE ...

Here, thread B runs at half the speed of A. Thread C runs at half the speed of B and 1/4 the speed of A. Thread D runs at half the speed of C, 1/4 the speed of B, and 1/8 the speed of A. And so on.

(Of course our previous scheduling algorithm is more general and this new algorithm is a subset of it. To see this, just imagine the old scheduling algorithm, but restrict yourself to only using levels 1, 2, 4, 8, 16, ...)

With 10 levels (i.e., 0...9), we go from high-speed threads at level 0 like A, to threads at level 9 which are running at 1/512 the speed of A (since $2^9 = 512$).

Multiple Ready Lists

In the above discussion, we were assuming only a single thread at each priority level, but of course we must accommodate many threads at each level. So within each level, there may be many threads. We can give the threads at level 0 the following names:

$a_1, a_2, a_3, a_4, \ldots$

We can give the threads at level 1 the following names:

$b_1, b_2, b_3, b_4, \ldots$

And so on.

For each level, we will have a simple FIFO list, which we implement with a linked-list. When it is time to schedule a thread from some level, we select the thread at the front of the ready list for that level and give it a time-slice.

Repeating from above, here again is the scheduling order:

A AB A ABC A AB A ABCD A AB A ABC A AB A ABCDE…

which is:

AABAABCAABAABCDAABAABCAABAABCDE…

So we see that the number of time-slices we give to A threads is twice the number of time-slices we give to B threads. This means that A threads are run at twice the speed of threads at level B if and only if there are an equal number of threads at the A level and the B level. But there may not be.

For example, if there are 4 times as many threads at level A as are at level B, then the threads at level A will run twice as slow as the threads at level B.

$a_1\ a_2\ \underline{b}_1\ a_3\ a_4\ \underline{b}_1\ c_1\ a_1\ a_2\ \underline{b}_1\ a_3\ a_4\ \underline{b}_1\ c_1\ d_1\ a_1\ a_2\ \underline{b}_1\ a_3\ a_4\ \underline{b}_1\ \ldots$

In this example, you can see that thread b_1 is running twice as often as thread a_1. This is not what we want.

To remedy this, we will make the following modification. When it is time to run the A threads, we will give every thread on the A ready list a time-slice. In other words, we will go through the entire FIFO queue, running all threads exactly once. And since A runs twice before B runs (according to the schedule shown above), we will run through the FIFO queue for A twice before moving to the FIFO queue for B.

a_1 a_2 a_3 a_4 a_1 a_2 a_3 a_4 $\underline{b}_1$ a_1 a_2 a_3 a_4 a_1 a_2 a_3 a_4 $\underline{b}_1$ c_1…

Now you can see that thread b1 is running half as often as thread a1, which is what we want.

Notice that our schedule is repetitive. If we look at only 5 levels (0…4 or equivalently A…E), we can see that it repeats after 31 (i.e., 2^5-1). If we look at 10 levels (0…9 or A…J), we can see that it repeats after 1023 (i.e., 2^{10}-1).

We can express the schedule as follows. We can show which list is selected and how many times we have to go through it:

```
A-2 B-1 A-2 B-1 C-1 A-2 B-1 A-2 B-1 C-1 D-1 A-2 B-1 A-2 B-1 C-1 A-2
B-1 A-2 B-1 C-1 D-1 E-1 …
```

Also, to reduce the size of our schedule, we will consolidate. In the above schedule, list A is selected 16 times, B is selected 8 times, C is selected 4 times, D is selected 2 times, and E is selected 1 time.

We could rewrite the schedule as follows. Notice, that these numbers are preserved.

```
A-8 B-4 C-2 D-1 A-8 B-4 C-2 D-1 E-1 …
```

To execute this schedule, we begin by selecting the A ready FIFO list. We will give each thread on this list a time-slice and then we will repeat 7 additional times. Thus, every thread on the A list will get 8 time slices. Then we move to the next item in the schedule and give each thread in the B list 4 time-slices. If, when we come to a list, the list is empty, we immediately move to the next item in the schedule. For example, if there are not threads on the C list, we move on to the next entry, which is D-1.

The following schedule would also satisfy the overall numerical requirement for 5 priority levels:

```
A-16 B-8 C-4 D-2 E-1 …
```

but this runs A threads for a long time, somewhat starving the other threads. With 10 priority levels, the following is the shortest possible schedule:

```
A-512 B-256 C-128 D-64 E-32 F-16 G-8 H-4 I-2 J-1…
```

Assuming each time slice is 2 milliseconds, it means that a single thread on the A list will monopolize the CPU for 512 × 2 ms = 1.024 seconds, and all other threads will be completely locked out for that time. This is unacceptable for any interactive application. And it would be even worse if there were several threads on the A list. For this reason, we want a schedule that will execute the A threads at most 16 at once. A delay of 16 × 2 ms = 32 ms seems acceptable.

Here is the schedule we will use:

```
A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4

F-4 G-2 H-1

A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4

F-4 G-2 H-1 I-1

A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4

F-4 G-2 H-1

A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4
A-16 B-8 C-4
A-16 B-8 C-4 D-4 E-4

F-4 G-2 H-1 I-1 J-1
```

When we reach the end, we repeat from the beginning.

Although this schedule doesn't exactly replicate the pattern originally given, it respects it, in the sense that each thread is executed the same number of times as above.

Implementation This schedule will be preset and hardcoded into the algorithm. There are two arrays (**scheduleList** and **scheduleCount**) which will be initialized at compile-time. The **scheduleList** is an array where each element is a pointer to one of the 10 ready lists. The **scheduleCount** is an array where each element is an integer giving the number of times that the ready list is to be gone through. For example:

scheduleList [0] = ptr to readyListA	**scheduleCount** [0] = 16
scheduleList [1] = ptr to readyListB	**scheduleCount** [1] = 8
scheduleList [2] = ptr to readyListC	**scheduleCount** [2] = 4
scheduleList [3] = ptr to readyListA	**scheduleCount** [3] = 16
…	…

Below is the pseudo-code for the scheduling algorithm. The following variables are used to keep track of where we are:

`schedIndex` = index of element in **scheduleList** / **scheduleCount** currently being used.
`thisList` = ptr to the ready list being used
`remainingCount` = number of times left to go through the list
`listLast` = ptr to the thread at the tail end of the readyList we are running through, so we can detect how many times we've gone through the FIFO list.

`INITIALIZATION...`

```
schedIndex = SCHEDULE_SIZE - 1
thisList = readyList for "initThread"
remainingCount = 0
listLast = ptr to "initThread"
```

`AT END OF "thr"s TIME-SLICE...`

```
thisList.AddLast (thr)
if (thr == listLast)
  remainingCount = remainingCount - 1
  if (remainingCount <= 0)
    repeat
      schedIndex = (schedIndex + 1) % SCHEDULE_SIZE
      thisList = scheduleList [schedIndex]
    until (thisList is not empty)
    remainingCount = scheduleCount [schedIndex]
    listLast = thisList.last
  endIf
endIf
```

TO SELECT A NEW THREAD AND START THE NEXT TIME-SLICE...

```
thr = thisList.RemoveFirst ()
START TIME-SLICE...
```

This algorithm should execute very quickly to select a new thread for scheduling. In most cases, one of the first two "if" tests is bound to fail, so the amount of work is only slightly more than pure round-robin with a single linked list. Occasionally, both tests will succeed and the "repeat" loop will be executed. In heavily loaded systems where there are threads at every priority level, the body of the repeat loop will be executed only once. Even if the body is executed a couple of times, the body is quite short.

To evaluate this algorithm, I will take a number of threads (e.g., 50 threads) and randomly assign them priorities in the range 0…9 (i.e., A…J). Then, I will run this algorithm and see how long is spent in the scheduler.

For comparison, I will take the same number of threads (50 in this example) and assign each thread the same priority (i.e., 0…9). Then, to run the threads at the same rate, I will convert a priority of N into 2^N (i.e., into 1, 2, 4, 8, 16, …, 512) and use the red-black scheduling algorithm.

For example, a thread assigned priority 0 will be run most frequently. In this new algorithm, such a thread will be placed in ready list A. On the other hand, a thread assigned a priority of 3 will be placed in ready list D and will be run at 1/8 the frequency of a thread in ready list A. For comparison, this same thread will be assigned priority 2^3 and (i.e., 8) for the red-black algorithm and will therefore be run at 1/8 the speed of threads in the fastest priority.

Given a thread assigned some priority N, the thread should be run at the exactly the same rate in the two different algorithms. However we should note that, although the threads are run at the same overall rate, the exact ordering of individual time-slices will be quite different. For example, consider two threads A and B, where B runs at half the speed of A. In the red-black algorithm, the time-slices will be ordered like this:

```
AABAABAABAABAAB …
```

In the modified, multi-list algorithm, the time-slice ordering will be:

```
AAAAAAAAAAAAAAAA BBBBBBBB AAAAAAAAAAAAAAAA BBBBBBBB …
```

The difference is that a thread like B will have to wait 8 times longer before it gets a change to run, but when it does run, it will get 8 times as many time-slices. So the modified, multi-list algorithm introduces some lack of responsiveness to threads like B. In the schedule given above, B has to wait 8 times as long but it gets 8 time-slices when it is scheduled. Similarly, threads C, D, E, and F have to wait 4 times as long before they are scheduled, but they get 4 time-slices when they run. Thread G has to wait twice as long, but gets 2 time-slices when scheduled. There is no effect on threads H, I, and J, which are given a

single time-slice when they run and are only made to wait as long as necessary to achieve their desired frequency.

I hope to learn whether the modified, multi-list algorithm or the red-black algorithm performs better, and under what conditions. I fully expect the modified, multi-list algorithm to perform much better, but by how much? Is it significant?

It would also be possible to have a different schedule that eliminates the delays imposed on threads like B, C, D, E, F, and G, but this schedule would be longer:

```
A-2 B-1 A-2 B-1 C-1 A-2 B-1 A-2 B-1 C-1 D-1 …
```

We could also streamline the schedule an eliminate the array **scheduleCount** altogether, since only A ever runs through its FIFO list more that once. This would give a schedule like this:

```
A A B A A B C A A B A A B C D …
```

This schedule would be longer. In our example with 10 priority levels, it is a difference between 135 entries in arrays **scheduleList** and **scheduleCount** (as shown above) versus 1023 entries (= 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1) in array **scheduleList**. Perhaps a scheme with only 8 priority levels (and 255 entries in its **scheduleList** array) would be a good compromise.

So another question is which approach to the schedule arrays is best: either bunched (A-16, B-8, C-4, …) or not bunched (A A B A A B C …)? I will address the un-bunched schedule proposal after experimenting with the bunched schedule.

This is a “dynamic paper”. At this point, I must pause the writing to implement the algorithm I have just described before I can collect any performance numbers. My plan is to implement the scheduling algorithms described above…

There are some subtle points concerning the algorithm above. From time to time threads may become blocked or otherwise made un-runnable. In other words, a thread that is on a ready list might get removed. For example, when a task is killed, it forcies the termination of all its threads.

The algorithm as presented above saves a pointer to the last thread on the ready list, in variable **listLast**. In particular, **listLast** always points to a thread from **thisList**. We are using this pointer to check to see when we have just completed running though the entire ready list. We have several cases that must be handled.

First, the currently running thread may become blocked, in which case it will not be added back to the ready list (i.e., to **thisList**). This could leave the current ready list empty. When the algorithm is ready to select the next thread to run, blindly grabbing the first element of **thisList** will fail.

Second, the thread that is pointed to by **listLast** may be removed. Perhaps this is the currently running thread and it blocks itself. Or perhaps the thread is not running, but is removed from the ready list by some other thread, for example when a thread is "killed". Without fixing the code, the algorithm above will continue to cycle through **thisList** forever, never reaching a thread that satisfies the test

```
if (thr == listLast)
```

We can solve the first problem by inserting a check whenever we remove the next element from **thisList**. If there is none, then we will need to move on to the next schedule entry:

TO SELECT A NEW THREAD AND START THE NEXT TIME-SLICE...

```
thr = thisList.RemoveFirst ()
if (thr == null)
  repeat
    schedIndex = (schedIndex + 1) % SCHEDULE_SIZE
    thisList = scheduleList [schedIndex]
  until (thisList is not empty)
  remainingCount = scheduleCount [schedIndex]
  listLast = thisList.last
  thr = thisList.RemoveFirst ()
endIf
START TIME-SLICE...
```

We can solve the second problem by inserting a check every time a thread becomes blocked or is otherwise removed from the ready list and made un-runnable. We must find a new thread to use to detect the end of **thisList**. In other words, we must set **listLast** to some other thread.

AFTER "thr" IS MADE UN-RUNNABLE...

```
if (thr == listLast)
  if (thisList.last != null)
    listLast = thisList.last
  elseIf (currentThread came from thisList)
    listLast = currentThread
  else
    repeat
      schedIndex = (schedIndex + 1) % SCHEDULE_SIZE
      thisList = scheduleList [schedIndex]
    until (thisList is not empty)
    remainingCount = scheduleCount [schedIndex]
    listLast = thisList.last
  endIf
endIf
```

We need the test to see if **currentThread** came from **thisList** because the scheduler also supports real-time threads (whose scheduling is not discussed here) and the thread performing the "kill" action may actually be a real-time thread, so we cannot blindly use it.

Evalution of the Multi-List Algorithm

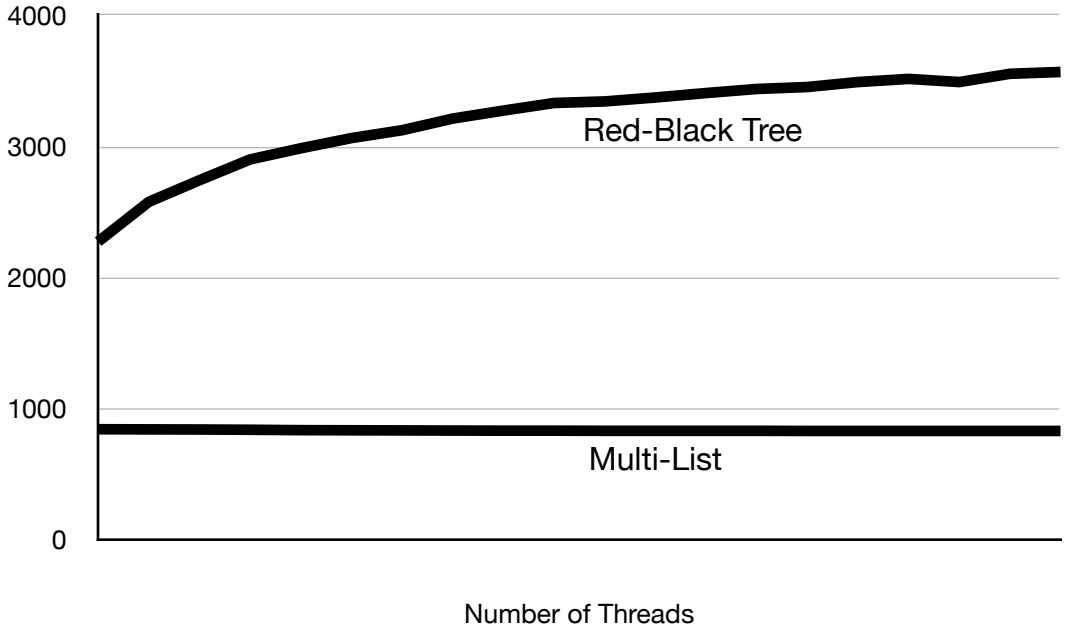
Okay, I'm back and ready to present some numbers. We will compare the the multi-list algorithm (as described above) to the red-black tree algorithms.

Since the multi-list algorithm accommodates 10 priority levels, we will randomly assign each thread a priority in the range (0…9). A thread at one level (e.g., 7) will run at half the speed of a thread at the next greater level (e.g., 8). The red-black algorithm sort the tree on a key based on "frequency", where frequency determines how often the thread runs. To compare "apples to apples", we will only use frequencies of 2^N (i.e., 1, 2, 4, 8, … 512) for the red-black algorithm.

We will use the same randomization for both the multi-list and the red-black algorithm. For example, if thread 17 happens to be assigned a priority level of 3, it will be at priority level 3 in both the multi-list and the red-black. In the multi-level algorithm, level 3 means it is placed in ready list D (since the priorities are 0=A, 1=B, 2=C, 3=D, … 9=J). In the red-black algorithm, level 3 means that it has a frequency value of $2^N = 8$ (since the priorities are 1, 2, 4, 8, 16, … 512). Thus, we are running the same mix of jobs against both algorithms.

Here is the result, using 10 priority levels:

Number of Threads	Red-Black Tree	Multi-List
10	2273	840
20	2577	839
30	2742	838
40	2901	836
50	2986	833
60	3064	832
70	3124	831
80	3213	830
90	3274	829
100	3332	829
110	3344	828
120	3374	828
130	3408	828
140	3439	828
150	3454	827
160	3492	827
170	3516	827
180	3492	827
190	3555	827
200	3569	827



This data indicates that the multi-list algorithm is vastly superior to the red-black tree algorithm. The multi-list algorithm is—in practice—a constant-time algorithm, and with a small constant.

Keep in mind that the numbers presented here include additional unrelated overhead, so the actual cost of the algorithms is even less than the numbers we collected. The gap between these two curves—when viewed as a percentage improvement—is bigger than it looks. We measured about 830 instructions for the multi-list algorithm, but the actual cost is a smaller number. (Perhaps 200…??? I really don't know.) In any case, the performance improvement of multi-list over red-black is greater than this graph suggests.

The take-away here seems unambiguous:

> *If you can live with a small, fixed number of priority levels (e.g., 10 different priority levels) where all threads within one level are scheduled in a round-robin fashion, then the multi-level algorithm blows the red-black algorithm out of the water.*

As a comparison, recall that straight round-robin (which doesn't accommodate any priority levels) was taking about 807 instructions. The multi-list algorithm is taking about 830 instructions, so it is almost as efficient as straight round-robin.

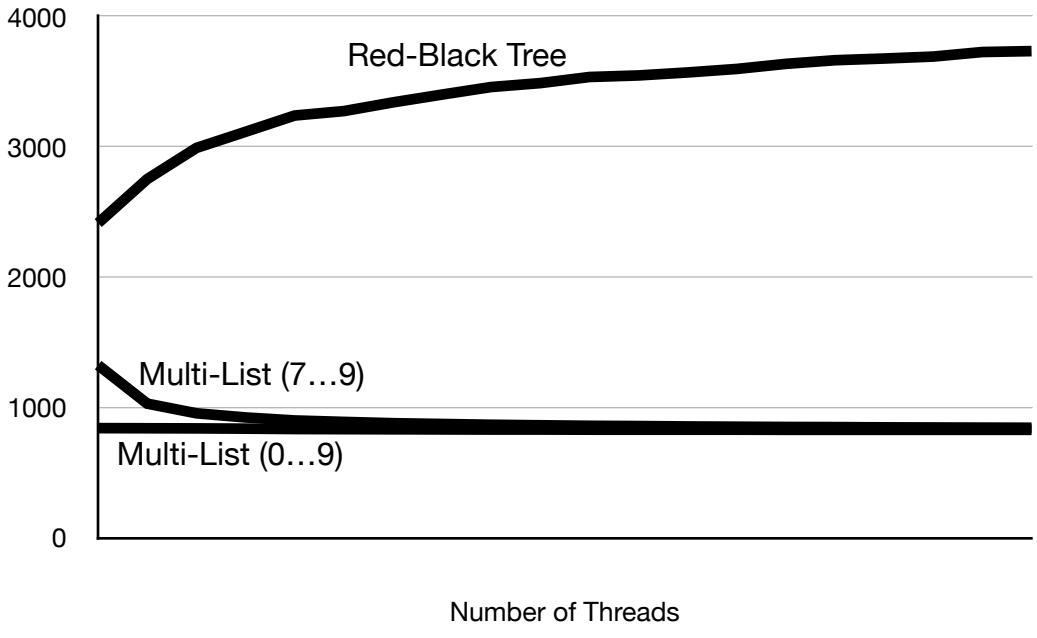
Note that the times for the multi-list algorithm actually decline a slight bit as we increase the number of threads, going from 840 down to 827. Imagine a job mix with 10 threads; since there are 10 different priority levels, each ready list has an average of 1 element. Thus, we spend a larger percentage of time switching from one ready list to the next. Now imagine a job mix of 150 threads; each ready list will have an average of 15 elements. For this job mix, we spend a larger percentage of the time just going through the ready-list in FIFO order, which is fast. I think this explains why the times actually fall a little bit with a larger number of threads.

Next, let's try a different job mix, to see if this has a significant effect on the performance of the algorithms.

Next, we compare the multi-list algorithm to itself under two different conditions. In the first, the job mix is distributed randomly over all 10 priority levels (0…9). This is just the same as the previous test. In the second, we still have 10 levels, but all threads are randomly assigned to one of the lowest 3 priority levels (i.e., 7…9). The algorithm spends a lot of time with the high-priority ready lists (ready list A, ready list B, etc.), but in this job mix, those lists are empty. This is expected to degrade performance, but the question is: How much?

For comparison, we will also run the red-black algorithm. To make the comparison fair, we'll restrict the job mix for the red-black algorithm to the lowest 3 priority levels.

Number of Threads	Multi-List (7…9)	Multi-List (0…9)	Red-Black (7…9)
10	1324	840	2416
20	1028	839	2754
30	954	838	2993
40	922	836	3117
50	899	833	3242
60	888	832	3276
70	878	831	3343
80	872	830	3403
90	866	829	3461
100	862	829	3491
110	858	828	3538
120	856	828	3550
130	853	828	3573
140	851	828	3600
150	849	827	3639
160	848	827	3667
170	846	827	3680
180	845	827	3695
190	844	827	3730
200	843	827	3737

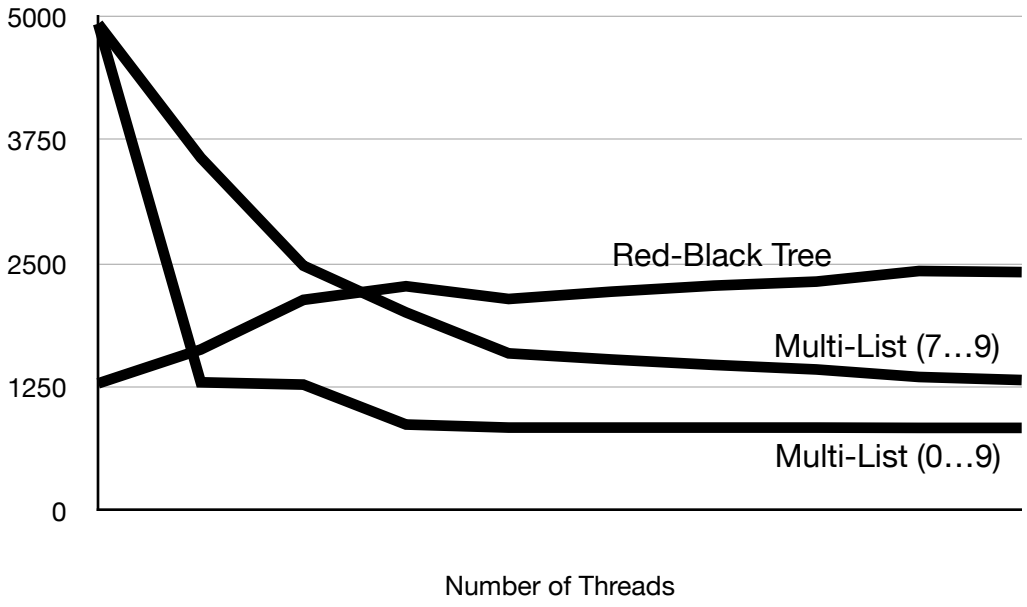


We see that with a small number of low-priority threads, the multi-list algorithm suffers from the increased overhead of having to constantly bypass the high-priority ready lists, which are all empty. And it cannot "get up to speed" because the ready lists that do have elements are not lengthy. However, with a larger number of threads, the ready lists are longer, and the round-robin approach within each priority level pays off, improving the performance. Above around 40 or 50 threads, it doesn't really matter how the threads are distributed between priority levels.

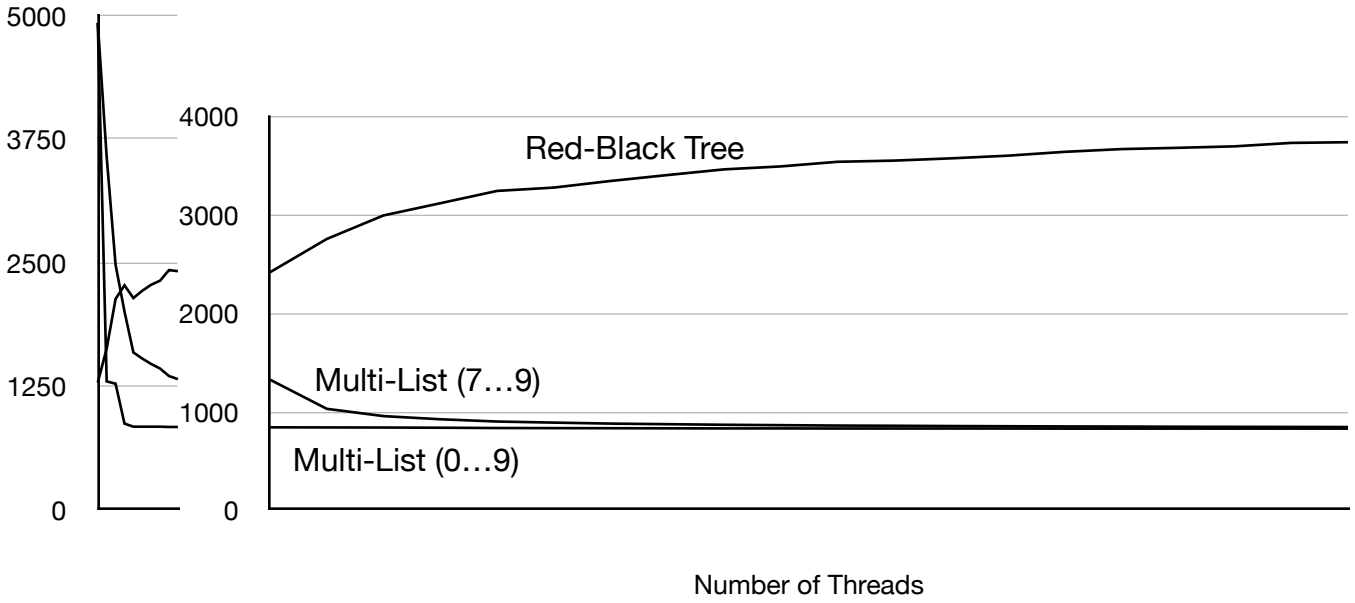
Whether you have 10 or 200 threads, and regardless of how those threads are distributed over the different priority levels, the multi-list algorithm is still clearly superior to the red-black algorithm.

It seems possible that with a very small number of threads, the red-black algorithm might be able to out-perform the multi-list algorithm. Let's look at job mixes with a very small number of threads. We are essentially asking what happens to these lines as we move further to the left.

Number of Threads	Multi-List (7…9)	Multi-List (0…9)	Red-Black (7…9)
1	4928	4928	1290
2	3573	1301	1633
3	2481	1278	2135
4	2008	874	2273
5	1594	843	2144
6	1532	843	2218
7	1477	843	2279
8	1431	843	2320
9	1355	840	2427
10	1324	840	2416



Perhaps we can make it clearer by rescaling and redrawing the above two graphs:



From this, we conclude that with 3 or fewer threads, the red-black algorithm might actually perform better. So if you are building a kernel in which you can be certain the maximum number of threads will always be small—say below 5—then there is no clear advantage to the multi-list algorithm. But with such a draconian limit on your job mix, you might as well go with straight round-robin, or do something more specific to your application area. In a typical kernel, you'll really need to be able to accommodate more than 10 threads, in which case, the multi-list approach seems to be overwhelmingly superior.

Increasing the Responsiveness

In the multi-list algorithm described above, we used a schedule that would run A threads for 16 times, before running the B threads. But when the B threads were run, they ran 8 times.

```
AAAAAAAAAAAAAAAA BBBBBBBB CCCC AAAAAAAAAAAAAAAA BBBBBBBB CCCC DDDD…
```

We would really like a schedule with more responsiveness, like the following:

```
AAB AAB C AAB AAB CD AAB AAB C AAB AAB CDE …
```

Next, we present an algorithm which we will call "**Multi-List-2**", to differentiate it from the previous algorithm, which we will now call "**Multi-List-1**".

We will make the following modifications:

We will eliminate the **remainingCount** variable. In other words, when we run through a ready list in the schedule we will run through it exactly one time.

The schedule for **Multi-List-1** was:

```
A-16 B-8 C-4
A-16 B-8 C-4 D-4
…
```

which we represented as:

scheduleList [0] = ptr to readyListA	**scheduleCount** [0] = 16
scheduleList [1] = ptr to readyListB	**scheduleCount** [1] = 8
scheduleList [2] = ptr to readyListC	**scheduleCount** [2] = 4
scheduleList [3] = ptr to readyListA	**scheduleCount** [3] = 16
…	…

The schedule for **Multi-List-2** will be:

```
A-2 B-1 A-2 B-1 C-1
A-2 B-1 A-2 B-1 C-1 D-1
A-2 B-1 A-2 B-1 C-1
A-2 B-1 A-2 B-1 C-1 D-1 E-1
…
```

or (equivalently):

```
A A B A A B C
A A B A A B C D
A A B A A B C
A A B A A B C D E
…
```

or (equivalently):

```
A
A B
A
A B C
A
A B
A
A B C D
A
A B
A
A B C
A
```

```
A B
A
A B C D E
…
```

which we will represent as:

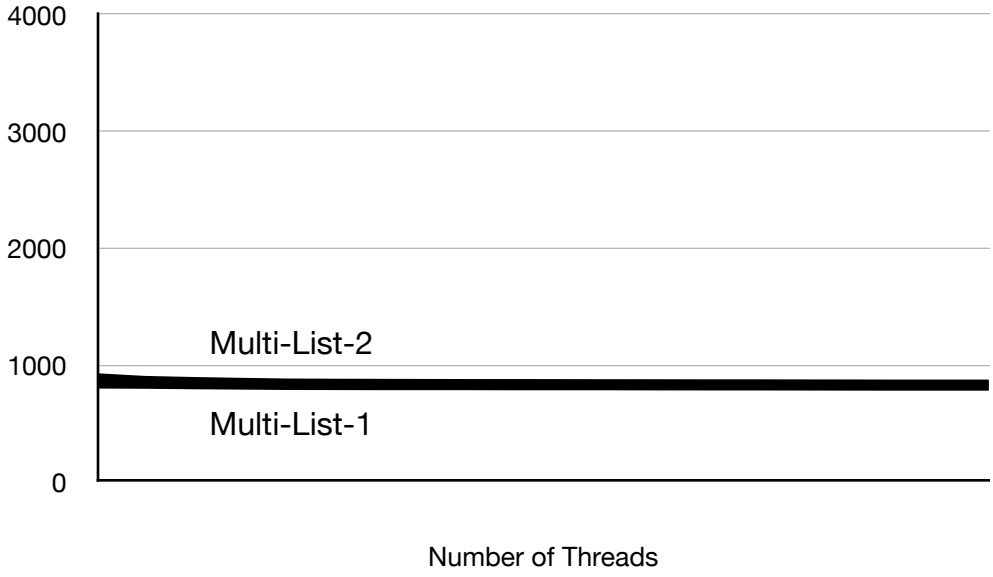
scheduleList [0] = ptr to readyListA
scheduleList [1] = ptr to readyListA
scheduleList [2] = ptr to readyListB
scheduleList [3] = ptr to readyListA
scheduleList [4] = ptr to readyListA
scheduleList [5] = ptr to readyListB
scheduleList [6] = ptr to readyListC
…

The **scheduleList** array will get pretty long, so we will reduce the number of priority levels from 10 to 8 (i.e., from 0…9 to 0…7). The length of the array will be 255 (i.e., 2^N-1, where N is the number of levels).

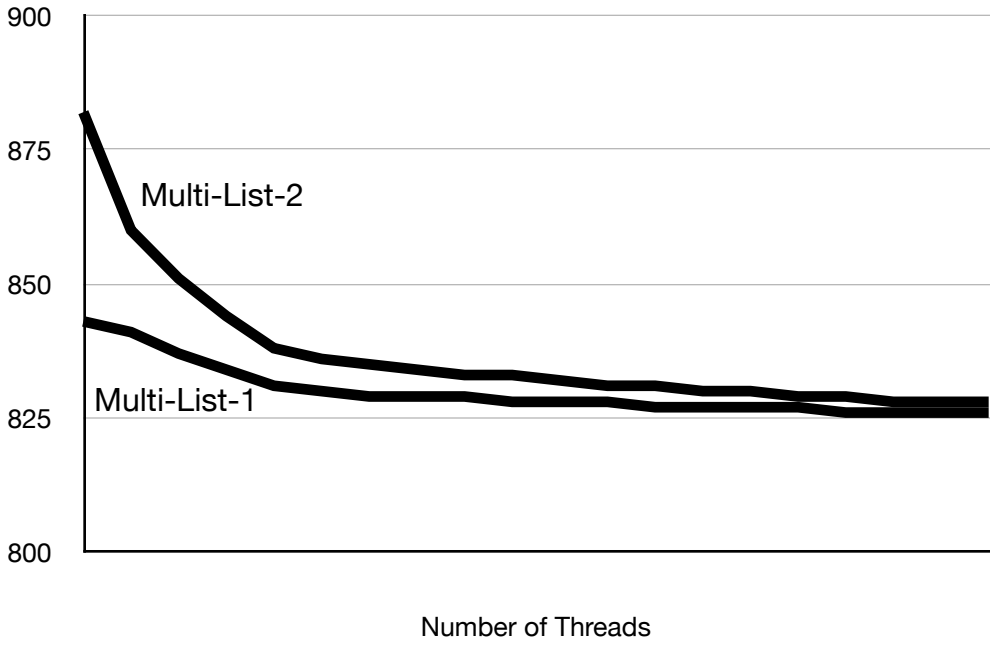
Here is the comparison of the performance between the **Multi-List-1** and **Multi-List-2** algorithms:

Number of Threads	Multi-List-1 (0…7)	Multi-List-2 (0…7)
10	843	882
20	841	860
30	837	851
40	834	844
50	831	838
60	830	836
70	829	835
80	829	834
90	829	833
100	828	833
110	828	832
120	828	831
130	827	831
140	827	830
150	827	830
160	827	829
170	826	829
180	826	828
190	826	828
200	826	828

First, we show this data using a scale similar to the previous graphs. This demonstrates that the difference is insignificant when compared to the red-black algorithm.



Next, we show the same data using a different scale, so we can see what is happening with a small number of threads.



When we have more than about 50 threads, the difference between the two algorithms is 6 instructions, going down to 2 instructions when we have 200 threads. This is essentially nothing. Even when we have only 10 threads, the **Multi-List-2** algorithm only adds an overhead of about 40 instructions. This seems a small price to pay for the increased responsiveness. In other words, for a small penalty, we make sure

that higher priority threads get time-slices much more regularly and are do not get “locked out” while threads in other priority levels get a huge amount of run-time.

Next, let’s look at job mixes where all threads are at the same priority level. We want to evaluate the **Multi-List-2** algorithm when all threads are at the same priority level. Does having to skip over a lot entries in the schedule that point to empty run lists degrade performance significantly?

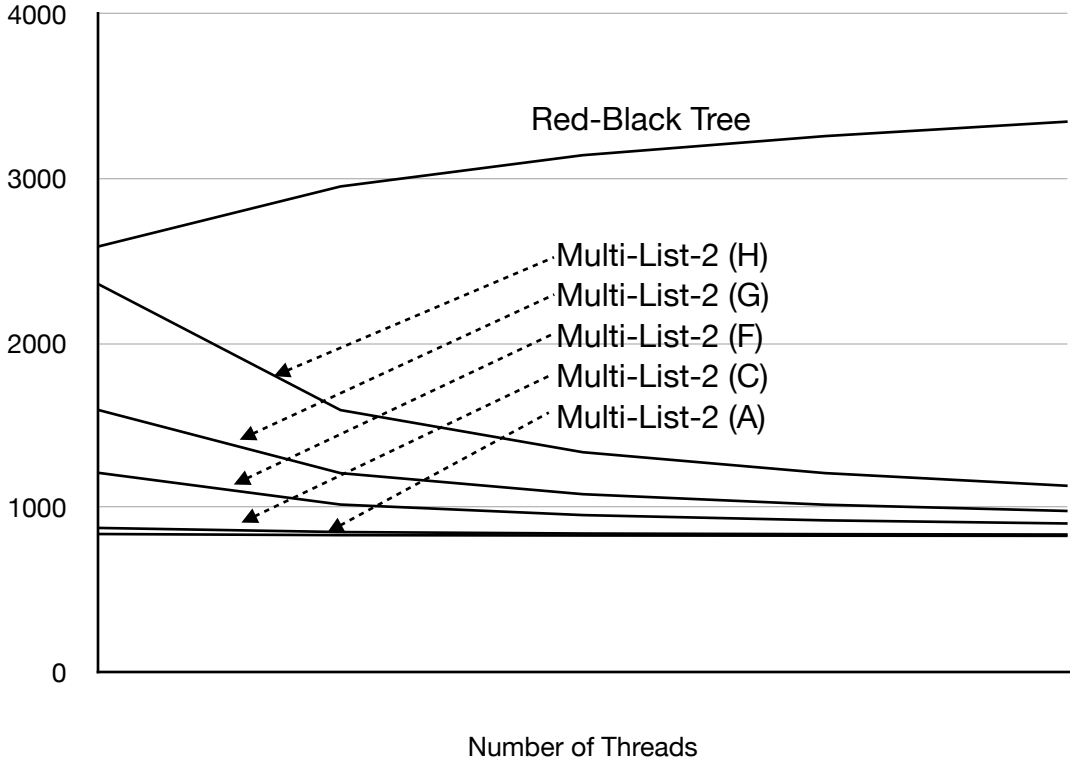
We’ll look at 5 different job mixes:

(1) All threads are at the highest priority 0=A
(2) All threads are at priority 2=C
(3) All threads are at priority 5=F
(4) All threads are at priority 6=G
(5) All threads are at the lowest priority 7=H

We will compare it to the red-black algorithm where all the jobs are at the same priority. With the red-black algorithm, it doesn’t matter whether all jobs are at level 1 or at level 128; the performance is identical as long as all jobs have the exact same frequency.

We expect **Multi-List-2** to perform especially poorly when all threads are at the lowest priority, since the algorithm must spend a lot of time looking through the schedule array, only to find that most priority levels must be skipped. The question is whether the Red-Black algorithm will outperform it in this worst case scenario.

	Multi-List-2					
Threads	**A**	**C**	**F**	**G**	**H**	**Red-Black**
10	838	875	1210	1592	2357	2585
20	831	850	1017	1208	1591	2950
30	829	841	953	1080	1335	3140
40	828	837	921	1016	1208	3256
50	827	835	902	978	1131	3344



We see that—at least with 10 or more threads, the **Multi-List-2** algorithm is superior to the Red-Black algorithm even in the worst case, i.e., with all threads at the lowest priority.

Perhaps a wise approach in OS design is to use a default the thread priority somewhere in the middle (e.g., 5=E) and only use the lower and higher priorities when there is good reason. This way we can avoid the worst case performance for **Multi-List-2**.

Conclusion

In summary, this experimentation suggests to me:

> *The **Multi-Level-2** algorithm is clearly the superior approach to thread scheduling*

for a kernel in which there can be expected to be several (e.g., 10+) threads, running at different priority levels. We have only looked at non-real-time threads. (Scheduling real-time threads—which is also done in the Blitz kernel—is a completely different beast.)

To summarize the **Multi-Level-2** algorithm, there are 8 priority levels which are numbered 0…7 (or equivalently, A…H), where 0=A is the highest priority level and 7=H is the lowest priority. For each priority level, there is a single run queue, implemented as a doubly-linked list which is processed in FIFO order. When a given priority is scheduled, each thread in the corresponding run list gets a single time-slice. The threads in the FIFO list are scheduled in order, removing them from the front of the list and returning them to the tail of the list. After each thread in the list has been given a single time slice, the scheduling algorithm moves on to the next priority level.

There is a separate “schedule array", which determines which priority is serviced when. This array is fairly large (255 entries) and the scheduler cycles through the array repeatedly. The array is initialized and never changes. It is initialized so that each priority level is given the appropriate number of turns. That is, priority level A appears a lot of times, giving each high-priority thread a lot of time-slices, while the lower priorities like F, G, and H appear only a few times.

The scheduling order (i.e., the array initialization) is:

```
AABAABCAABAABCDAABAABCAABAABCDE …
```

Adding spaces may make this order clearer:

```
AAB AAB C AAB AAB CD AAB AAB C AAB AAB CDE …
```

where A represents the FIFO run queue for the highest priority threads, B represents the FIFO run queue for the second highest priority threads, and so on.

The threads at level N run at half the speed—i.e., get time-slices at half the frequency—of threads at level N-1. For example, a thread at level D will run at half the speed of a thread at level C, 1/4 the speed of a thread at level B, but twice as fast as a thread at level E, and 4 times as fast as level F. A thread at the highest priority level (0=A) will be given a time-slice 128 times as often as a thread at the lowest priority level (7=H) is given a time-slice.

Moral

Some algorithms (such as red-black trees) are extremely clever and have superior asymptotic performance. But that doesn't always make them the best choice in a specific environment.

The red-black tree algorithm was a super-big pain to program, debug, and test. Initially, I considered avoiding any algorithm based on red-black trees, but I knew I would be plagued by the suspicion that I was just too lazy to use the best algorithm. By going through the hard work of implementation and then performing the above empirical testing, I am now able to discard the red-black algorithm, knowing that a much simpler algorithm is truly the better choice for the Blitz kernel's thread scheduler.