CHAPTER 14

Using Task Priorities

The Threading Building Blocks scheduler is not a real-time scheduler, and therefore it is not suitable for use in *hard* real-time systems. In real-time systems, a task can be given a deadline by which it must complete, and the usefulness of the task degrades if it misses its deadline. In hard real-time systems, a missed deadline can lead to a total system failure. In soft real-time systems, a missed deadline is not catastrophic but leads to a decrease in quality of service. The TBB library has no support for assigning deadlines to tasks, but it does have support for task priorities. These priorities might be of use in applications that have soft real-time requirements. Whether or not they are sufficient requires understanding both the soft real-time demands of the application and the properties of TBB tasks and task priorities.

Beyond soft real-time use, task priorities can have other uses as well. For example, we may want to prioritize some tasks over others because doing so will improve performance or responsiveness. Perhaps we want to prioritize tasks that free memory over tasks that allocate memory, so that we reduce the memory footprint of our application. Or perhaps, we want to prioritize tasks that touch data already in our caches over tasks that will load new data into our caches.

In this chapter, we describe task priorities as supported by TBB tasks and the TBB task scheduler. Readers that are considering TBB for soft real-time applications can use this information to determine if TBB is sufficient for their requirements. Other readers might find this information useful if needed to implement performance optimizations that benefit from task priorities.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_14

Support for Non-Preemptive Priorities in the TBB Task Class

Just like with the support for task affinities described in Chapter 13, TBB's support for priorities is enabled by functions in the low-level task class. The TBB library defines three priority levels: `priority_normal`, `priority_low`, and `priority_high` as shown in Figure 14-1.

```
namespace tbb {
  enum priority_t {
    priority_normal = implementation-defined,
    priority_low = implementation-defined,
    priority_high = implementation-defined
  };

  class task {
    // . . .
    static void enqueue( task&, priority_t );
    void set_group_priority ( priority_t );
    priority_t group_priority () const;
    // . . .
  };
}
```

Figure 14-1. *The types and functions in class task that support priorities*

In general, TBB executes tasks that are higher priority before tasks that are lower priority. But there are caveats.

The most important caveat is that TBB tasks are executed non-preemptively by TBB threads. Once a task has started to execute, it will execute to completion – even if a higher priority task has been spawned or enqueued. While this behavior may seem like a weakness, since it may delay the application's switch to higher priority tasks, it is also a strength because it helps us avoid some dangerous situations. Imagine if, for example, a task t_0 holds a lock on a shared resource and then higher priority tasks are spawned. If TBB doesn't allow t_0 to finish and release its lock, the higher priority tasks can deadlock if they block on the acquisition of a lock on this same resource. A more complicated but similar issue, *priority inversion*, was famously the cause of problems with the Mars Pathfinder rover in the late 1990s. In "What Happened on Mars?", Mike Jones suggests priority inheritance as a way to address these situations. With priority inheritance, a task

that blocks higher priority tasks inherits the priority of the highest task it blocks. The TBB library does not implement priority inheritance or other complicated approaches since it avoids many of these issues due to its use of non-preemptive priorities.

The TBB library ***does not*** provide any high-level abstraction for setting ***thread priorities***. Because there is no high-level support in TBB for thread priorities, if we want to set thread priorities, we need to use OS-specific code to manage them – just as we did for thread-to-core affinity in Chapter 13. And just as with thread-to-core affinity, we can use `task_scheduler_observer` objects and invoke these OS-specific interfaces in the callbacks as threads enter and exit the TBB task scheduler, or a specific task arena. However, we warn developers to ***use extreme caution when using thread priorities***. If we introduce thread priorities, which are preemptive, we also invite back in all of the known pathologies that come with preemptive priorities, such as priority inversion.

Critical Rule of Thumb Do not set different priorities for threads operating in the same arena. Weird things can and will happen because TBB treats threads within an arena as equals.

Beyond the non-preemptive nature of TBB task execution, there are a few other important limitations to mention about its support for task priorities. First, changes may not come into effect immediately on all threads. It's possible that some lower priority tasks may start executing even if there are higher priority tasks present. Second, worker threads may need to migrate to another arena to gain access to the highest priority tasks, and as we've noted before in Chapter 12, this can take time. Once workers have migrated, this may leave some arenas (that do not have high priority tasks) without worker threads. But, because master threads cannot migrate, the master threads will remain in those arenas, and they themselves are not stalled – they can continue to execute tasks from their own task arena even if they are of a lower priority.

Task priorities are not hints like TBB's support for task-to-thread affinity described in Chapter 13. Still, there are enough caveats to make task priorities weaker in practice than we may desire. In addition, the support for only three priority levels, low, normal, and high, can be quite limiting in complex applications. Even so, we will continue in the next section by describing the mechanics of using TBB task priorities.

Setting Static and Dynamic Priorities

Static priorities can be assigned to individual tasks that are enqueued to the shared queue (see enqueued tasks in Chapter 10). And *dynamic priorities* can be assigned to groups of tasks, through either the `set_group_priority` function or through a `task_group_context` object's `set_priority` function (see the `task_group_context` sidebar).

TASK_GROUP_CONTEXT: EVERY TASK BELONGS TO A GROUP

A `task_group_context` represents a group of tasks that can be canceled, or have their priority level set, together. All tasks belong to some group, and a task can be a member of only one of these groups at a time.

In Chapter 10, we allocated TBB tasks using special functions such as `allocate_root()`. There is an overload of this function that lets us assign a `task_group_context` to a newly allocated root task:

```
static proxy2 allocate_root( task_group_context& );
```

A `task_group_context` is also an optional argument to TBB high-level algorithms and to the TBB flow graph, for example:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index_type last, const Func& f,
                  partitioner[, task_group_context& group]] );

graph([task_group_context& context]);
```

We can assign groups at the task level during allocation but also through the higher-level interfaces, such as TBB algorithms and flow graphs. There are other abstractions, such as `task_group`, that let us group tasks for execution purposes. The purpose of `task_group_context` groups is to support cancellation, exception handling, and priorities.

When we use the `task::enqueue` function to provide a priority, the priority affects only that single task and cannot be changed afterward. When we assign a priority to a group of tasks, the priority affects all of the tasks in the group and the priority can be changed at any time by subsequent calls to `task::set_group_priority` or `task_group_context::set_priority`.

The TBB scheduler tracks the highest priority of ready tasks, including both enqueued and spawned tasks, and postpones (the earlier caveats aside) execution of lower priority tasks until all higher priority tasks are executed. By default, all tasks and groups of tasks are created with `priority_normal`.

Two Small Examples

Figure 14-2 shows an example that enqueues 25 tasks on a platform with P logical cores. Each task actively spins for a given duration. The first task in the `task_priority` function is enqueued with normal priority and is set to spin for roughly 500 milliseconds. The for-loop in the function then creates P low priority, P normal priority, and P high priority tasks, each of which will actively spin for roughly 10 ms. When each task executes, it records a message into a thread-local buffer. The high-priority task `id`s are prefixed with `H`, the normal task `id`s with `N` and the low priority tasks `id`s with `L`. At the end of the function, all of the thread local buffers are printed, providing an accounting of the order in which tasks were executed by the participating threads. The complete implementation of this example is available in the Github repository.

```
#include <iostream>
#include <string>

#include <tbb/tbb.h>

void doWork(double sec);

class Spinner : public tbb::task {
public:
  Spinner(const char *m, int id, double len);
  tbb::task *execute() override;

private:
  std::string msg;
  int messageId;
  double length;
};

void enqueueTask(const char *msg,
                 int id,
                 double len,
                 tbb::priority_t
priority=tbb::priority_normal) {
  tbb::task::enqueue(*new( tbb::task::allocate_root())
                       Spinner(msg, id, len),
                     priority);
}

void fig_14_02() {
  int P = tbb::task_scheduler_init::default_num_threads();

  enqueueTask("N", 0, 0.5);
  doWork(0.01);

  for (int i = 0; i < P; ++i) {
    enqueueTask("L", i, 0.01, tbb::priority_low);
    enqueueTask("N", i+1, 0.01, tbb::priority_normal);
    enqueueTask("H", i, 0.01, tbb::priority_high);
  }
  doWork(1.0);
}
```

Figure 14-2. *Enqueuing tasks with different priorities*

Executing this example on a system with eight logical cores, we see the following output:

```
N:0              ← thread 1
H:7 H:5 N:3 L:7  ← thread 2
H:2 H:1 N:8 L:5  ← thread 3
H:6 N:1 L:3 L:2  ← thread 4
H:0 N:2 L:6 L:4  ← thread 5
H:3 N:4 N:5 L:0  ← thread 6
H:4 N:7 N:6 L:1  ← thread 8
```

In this output, each row represents a different TBB worker thread. For each thread, the tasks it executes are ordered from left to right. The master thread never participates in the execution of these tasks at all, since it doesn't call `wait_for_all`, and so we only see seven rows. The first thread executes only the first long, normal priority task that executed for 500 milliseconds. Because TBB tasks are non-preemptive, this thread cannot abandon this task once it starts, so it continues to execute this task even when higher priority tasks become available. Otherwise though, we see that even though the `for`-loop mixes together the high, normal, and low priority task enqueues, the high priority tasks are executed first by the set of worker threads, then the normal tasks and finally the low priority tasks.

Figure 14-3 shows code that executes two `parallel_for` algorithms in parallel using two native threads, `t0` and `t1`. Each `parallel_for` has 16 iterations and uses a `simple_partitioner`. As described in more detail in Chapter 16, a `simple_partitioner` divides the iteration space until a fixed grainsize is reached, the default being a grainsize of 1. In our example, each `parallel_for` will result in 16 tasks, each of which will spin for 10 milliseconds. The loop executed by thread `t0` first creates a `task_group_context` and sets its priority to `priority_high`. The loop executed by the other thread, `t1`, uses a default `task_group_context` that has a `priority_normal`.

```
#include <iostream>
#include <thread>

#include <tbb/tbb.h>

void doWork(double sec);

void fig_14_03() {
  std::thread t0([]() {
    tbb::task_group_context tcg;
    tcg.set_priority(tbb::priority_high);
    tbb::parallel_for( 0, 16, [] (int) {
      // do high priority work
      doWork(0.01);
      std::cout << "High\n";
    }, tbb::simple_partitioner(), tcg );
  });

  std::thread t1( []() {
    tbb::parallel_for( 0, 16, [] (int) {
      // do normal priority work
      doWork(0.01);
      std::cout << "Normal\n";
    }, tbb::simple_partitioner());
  });

  t0.join();
  t1.join();
}
```

Figure 14-3. *Executing algorithms with different priorities*

An example output from the sample when executed on a platform with eight logical cores follows:

```
Normal
High
High
High
High
High
High
```

```
Normal
High
High
High
High
High
High
High
High
Normal
High
High
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
```

Initially, there are seven "`High`" tasks executed for every one "`Normal`" task. This is because thread `t1`, which started the `parallel_for` with normal priority, cannot migrate away from its implicit task arena. It can only execute the "`Normal`" tasks. The other seven threads however, execute only the "`High`" tasks until they are all completed. Once the high priority tasks are completed, the worker threads can migrate to thread `t1`'s arena and help out.

Implementing Priorities Without Using TBB Task Support

What if low, normal, and high are not enough? One workaround is to spawn generic wrapper tasks that look to a priority queue, or other data structure, to find the work they should do. With this approach, we rely on the TBB scheduler to distribute these generic wrapper tasks across the cores, but the tasks themselves enforce priorities through a shared data structure.

Figure 14-4 shows an example that uses a `task_group` and a `concurrent_priority_queue`. When a piece of work needs to be done, two actions are taken: (1) a description of the work is pushed into the shared queue and (2) a wrapper task is spawned in the `task_group` that will pop and execute an item from the shared queue. The result is that there is exactly one task spawned per work item – but the specific work item that a task will process is not determined until the task executes.

```
#include <iostream>
#include <tbb/tbb.h>

class WorkItem {
public:
  WorkItem() { }
  WorkItem(int p) : priority(p) { }

  bool operator<(const WorkItem &b) const {
    if (priority < b.priority) return true;
    else return false;
  }

  void doWork();

private:
  int priority;
};

void fig_14_4() {
  tbb::concurrent_priority_queue<WorkItem> q;
  tbb::task_group g;

  const std::string prefix = "w";
  for (int i = 0; i < 16; ++i) {
    q.push(WorkItem(i));
    g.run([&q]() {
      WorkItem w;
      if (q.try_pop(w))
        w.doWork();
    });
  }
  g.wait();
}
```

Figure 14-4. *Using a concurrent priority queue to feed work to wrapper tasks*

A `concurrent_priority_queue` by default relies on `operator<` to determine ordering and so when we define `work_item::operator<` as shown in Figure 14-4, we will see an output that shows the items executing in decreasing order, from 15 down to 0:

```
WorkItem: 15
WorkItem: 14
WorkItem: 13
WorkItem: 12
```

```
WorkItem: 11
WorkItem: 10
WorkItem: 9
WorkItem: 8
WorkItem: 7
WorkItem: 6
WorkItem: 5
WorkItem: 4
WorkItem: 3
WorkItem: 2
WorkItem: 1
WorkItem: 0
```

If we change the operator to return true if (`priority > b.priority`), then we will see the tasks execute in increasing order from 0 to 15.

Using the generic-wrapper-task approach provides increased flexibility because we have complete control over how priorities are defined. But, at least in Figure 14-4, it introduces a potential bottleneck – the shared data structure accessed concurrently by the threads. Even so, when TBB task priorities are insufficient we might use this approach as a backup plan.

Summary

In this chapter, we provided an overview of task priority support in TBB. Using mechanisms provided by `class task`, we can assign low, normal, and high priorities to tasks. We showed that we can assign static priorities to tasks that are enqueued and dynamic priorities to groups of tasks using `task_group_context` objects. Since TBB tasks are executed non-preemptively by the TBB worker threads, the priorities in TBB are also non-preemptive. We briefly discussed the benefits and drawbacks of non-preemptive priorities, and also highlighted some of the other caveats we need to be aware of when using this support. We then provided a few simple examples that demonstrated how task priorities can be applied to TBB tasks and to algorithms.

Since there are many limitations to the task priority support in the library, we concluded our discussion with an alternative that used wrapper tasks and a priority queue.

The TBB scheduler is not a hard real-time scheduler. We see in this chapter though that there is some limited support for prioritizing tasks and algorithms. Whether these features are useful or not for soft real-time applications, or to apply performance optimizations, needs to be considered by developers on a case-by-case basis.

For More Information

Mike Jones, "What Happened on Mars?" a note sent on December 5, 1997. www.cs.cmu.edu/afs/cs/user/raj/www/mars.html.

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.