Implementing Thread Priorities in a HTTP Server

Soumith Chintala, Yuri Soussov, Randolph Chiu Tan
Courant Institute of Mathematical Sciences, New York University

Abstract-- We explore different ideas to implement thread priorities in general and in the process, implement three approaches and test them on an existing HTTP Server framework described in [2]. The first approach explores simple strict priorities while optimizing lock overheads by using a hierarchical check of queue sizes. The second approach uses biased probabilities to schedule the threads. The third approach uses Stride scheduling [1], exploring different stride sizes to evaluate performance changes.

1 Introduction
The goal of our project was to implement task priorities in a thread pool [2]. Our thread pool was designed to process HTTP requests. An acceptor parses requests and then creates tasks to fulfill the requests and puts them into a queue. Within our thread pool, worker threads wait for new tasks to arrive and execute them. A Diagram of how the initial thread pool worked can be seen in Fig.1

![Diagram of thread pool](image)

**Fig.1 Working of the unmodified thread-pool**

When implementing priorities on top of this standard thread pool model you have to consider the following aspects

- Tasks never being processed (Starvation [3])
- Importance of priorities (How frequent should a high priority task be chosen)
- Number of priorities
- Overhead of priority scheduling and management
- The scheduling algorithm

A very basic way to deal with priorities is by inserting tasks into the queue based on their priority. Highest priority tasks would be put at the head of the queue while lowest priority tasks would be put at the tail. A major issue with this approach is that it suffers from the starvation problem[3] where, if there are enough high priority tasks, turnaround time for low priority tasks would be very large. This may be acceptable in some systems but an HTTP server isn’t one of them. This brings up a highly probable
assumption that there is no single best way to implement priorities and scheduling performance varies by application. In order to devise an effective algorithm, you need to know the problem at hand and adjust your approach to the systems requirements. Most systems need pools that combat starvation and hence, we concentrate on priority scheduling which handles starvation.

One idea for combating starvation is to incorporate a dispatcher thread into your thread pool. The basic task of the dispatcher thread is to assign tasks to threads and to maintain the queue in such a way that starvation does not occur. A major issue with implementing this dispatching thread is the processing overhead that is created within the thread pool. Most implementations will attempt to have the dispatcher thread save its book-keeping work until it is otherwise idle to minimize this overhead. The basic idea for avoiding starvation is to stamp a thread’s waiting units and then increase its priority once a certain threshold is reached. Two possible metrics to use for the decision of increasing priority can be obtained by either time stamping the tasks or incrementing a counter every time the dispatcher access the queue.

2 Algorithms

Three implementations that we explored to handle priorities and combat starvation are described in detail in the section.

2.1 Strict Priorities with a flexible starvation-handling mechanism

This is the most basic approach to priorities. We evaluated this approach before evolving our algorithm to more efficient methods. It is extremely simple and optionally handles starvation. The algorithm is roughly described in Fig.2 and is as follows:
For each priority, we assign a separate queue of tasks. There are two types of workers which grab tasks and execute them. The first type are “special workers” which are dedicated to each priority and are used to handle starvation. The second type of workers are “general workers” which pop a task from the task queue in order of priority, i.e they pop the first available task by going through each task queue in decreasing order of priority. By modifying the number of workers, we see some special cases of the server.

- if there are 0 “special workers”, this algorithm becomes a strict priority implementation.
- If there are 0 “general workers”, it becomes a priority-less implementation.

We would ideally want number of general workers >> number of special workers for good performance. For optimization of performance and to avoid the overhead of locks, we have a two-stage mechanism in reading a task queue. In the first stage, we do an unreliable check on queue sizes which we call “dirty read” because it involves lockless reading of a boolean array. This gives us information on the queue size and hence acts as a fast check. Once this stage gives us a priority to start with, we perform a reliable lock-based check and pop a task from the queue to be executed by a general worker.
2.2 Probabilistic Selection

This is an idea derived from working with genetic algorithms, but similar approaches are mentioned elsewhere in literature. Not wanting to create overhead by adding a dispatcher thread, multiple queues are used to hold tasks of different priorities as shown in Fig 3. When choosing which queue to extract a task from, queues with a higher priority have a greater chance of being picked. The choice is still determined through the use of random numbers, so even the queue with lowest priority will still have a chance of being picked as the queue from which to extract the next task from.

One issue of using this thread pool model is deciding the probabilistic bias of a higher priority queue over a low priority one. There is also no way to guarantee that a high priority task will execute in a certain amount of time. However there are certain benefits to using this approach. It is easy to implement and reason about. There is very little overhead added to the thread pool. It also guarantees that starvation is impossible.
2.3 Stride Scheduling

Stride scheduling [1] is a deterministic scheduling algorithm that supports proportional allocation of resources. The basic concept of this technique is that each consumer (i.e. process) of the resource (i.e. process time) is assigned a ticket. The ticket is the a fix throughout the lifetime and it determines how big of a share the consumer will get. The stride is just the inverse of the ticket. The original paper has other sophisticated techniques that can dynamically scale the stride value, but for this project, the purpose of stride and ticket can be treated as the same thing. Every time a consumer gets allocated, it’s pass value will get incremented by its stride value, and the scheduler would always pick the one with the lowest pass value.

Fig. 4 shows a sample run of the stride scheduler. Each shape represents different processes. The triangle has the smallest stride value and gets selected by the scheduler the most while the square has the largest stride value and gets selected the least.

Fig. 5 shows how the stride scheduler was incorporated to the thread pool to handle different task priorities. Similar to the other implementations, this also uses a separate queue for each priority. The signature of the addTask method is modified to include a priority argument. Once the client calls this method, the task will be inserted to the queue corresponding to its priority. It will then signal the dispatcher thread that there is a new task available. The dispatcher will then get a lock on the task queues and consult the stride scheduler on which priority queue to get the next task from, if the queue is empty, it will keep on consulting the scheduler for the next one. Once the dispatcher is able to get a task, it will then unlock the task queue and wait for a free worker to be available and pass the task to it. The worker then dequeues itself from the free worker list and executes the task and enqueues itself again to the free worker list when done.
Fig. 4 Sample session of the stride scheduler. A straight line between two shapes means that it has been chosen by the scheduler and is running at that period of time (the x axis). The figure is borrowed from the original paper with added legend and coloring for clarity.

Fig. 5 Overview of the thread pool implementation using stride scheduler.

The motivation of using a separate dispatcher is to make the flow less chaotic. Instead of having several worker threads banging on the task queue, only one dispatcher gets to touch the task queue. And thus, the many to one relationship (many worker threads -> task queues) becomes one to one to one (one task
queues <-> one dispatcher <-> one free worker list).

3 Experiments

3.1 Machine Setup

We have one machine acting as the server and two clients connected via a 100 MBit switch requesting different files through httpperf. We also alternate the clients between files between runs to take care of any biases and to make the results reproducible.

3.2 Server Setup

We changed a couple of things in the server in order to test our algorithms. First, we converted the socketWrite() call inside Connection::doWrite() into a callback task so we can assign different priorities to different writes depending on priority. As a consequence, we had to move the entire block of code after it inside socketWrite() since socketWrite() is now an asynchronous task and we still have to preserve the original sequence of operations. Fortunately, doWrite was the only method calling socketWrite.

We also assigned 3 priorities for all tasks in the server - low, mid and high. All tasks have high priority by default - that includes reading the socket, collecting stats, serving files not explicitly labeled as low or mid. We then assigned two files that the test clients are going to request via httpperf as low and mid respectively.

Even after we purposely picked large files to fetch in the tests, we discovered that the write operations where performed so fast that the ThreadPool’s task queue gets emptied at a rate faster than the rate at which tasks are being added. This is bad for evaluation since that means that tasks that comes will be served immediately and hence, we won’t be able to test the effectiveness of our task selection through priority. Therefore, just for the purpose of the experiment, we inserted an artificial delay (a very long CPU spin) on the write task, to purposely line up jobs in the ThreadPool’s task queue.

To get statistics for the tests, we reused the HttpStats class[2]. To get a separate statistics for each priority, we created 3 corresponding instances of this class.

4 Results

This section shows the results we have obtained from performing the experiments described in the previous section. Each graph below shows the average requests per second per priority. The graph only includes the low and mid priorities for brevity. These are the tasks serving the files requested by the test machines.

As show in the figures, every algorithm was able to serve the higher priority task more than the lower priority task.

The graph for stride scheduling as shown in Fig.7 shows a very uniform saw tooth pattern, as we could expect from a deterministic algorithm.
Probabilistic Scheduling seen in Fig.8, as expected, has a very chaotic and unpredictable variation in performance.

*Fig.6 Graph for a run using Strict priorities.*

*Fig.7 Graph for a run using Probabilistic Scheduling.*
5 References