November 2020

ENABLING CRITICAL APLs TO SHARE RESOURCE POOLS ACROSS SERVICE BULKHEADS

HP INC

Enabling Critical APIs to Share Resource Pools Across Service Bulkheads

ABSTRACT:

It is common for services to bulkhead sets of APIs so that they are ensured a certain number of resources. This is done to support availability and performance service level agreements for the APIs. When bulkheading is performed, the resources available to the service are segregated between the sets of APIs. While this division addresses the availability and performance needs, it restricts critical APIs to a subset of the total resources available to the service when those resources are not in use by the other APIs.

What is desired is the bulkheading of resources between critical and non-critical APIs so that the non-critical set of APIs is limited in its use of resources and to support the availability and performance of the critical set of APIs by using resources from other pools.

This solution is to create resource pools for each set of APIs; in this discussion, there are two pools (critical and non-critical). The non-critical APIs use their resource pool exclusively. The critical APIs call the non-critical resource pool with a timeout. If the resource is returned before timing out, that resource is used. If the request times out, the critical resource pool is called with an infinite timeout, and the call waits for a free resource.

DESCRIPTION:

This disclosure relates to software. Specifically, it outlines the prioritization of requests for resources whose pools have been bulkheaded to address performance and application availability.

A method is disclosed that bulkheads resources between critical and non-critical APIs so that the non-critical set of APIs can only use resources from the non-critical pool but also supports the availability and performance of the critical set of APIs by using resources from both pools.

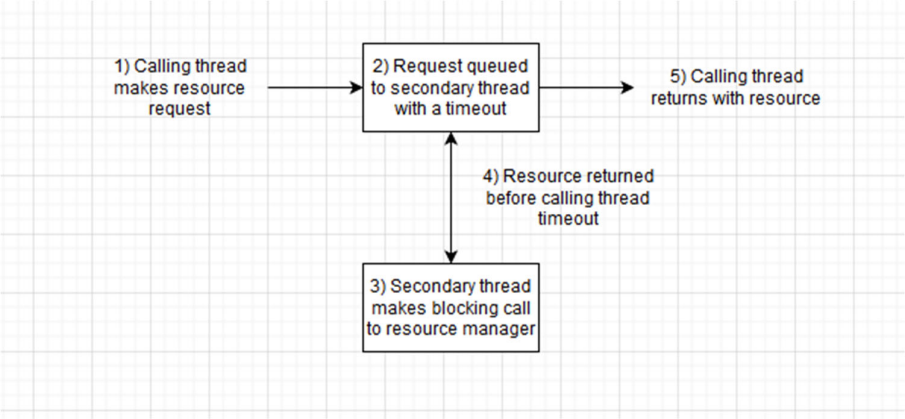
This method applies, generally, to a pair of critical and non-critical pools. While a service could have multiple pairs of critical and non-critical pools (and even reference the same pool across pairs), this discussion will cover the case of having a single critical pool and a single non-critical pool.

To use this method, the developer creates two resource pools: one for critical APIs and one for non-critical APIs. There is a resource manager that the critical APIs use to try to retrieve a resource from the non-critical pool in a non-blocking manner and, if the request times out, makes a blocking request from the critical resource pool. The resource manager has a timeout and a thread pool to handle blocking resource requests. The resource manager also has a deque of non-critical resource requests that have been satisfied after the waiting thread timed out, a thread to release the non-critical resources, and references to the critical and non-critical resource pools.

The resource manager's member function to acquire resources for the critical APIs first acquires a thread and makes a blocking request for a resource from the non-critical resource pool. The calling thread waits for a limited amount of time for the blocking call to complete.

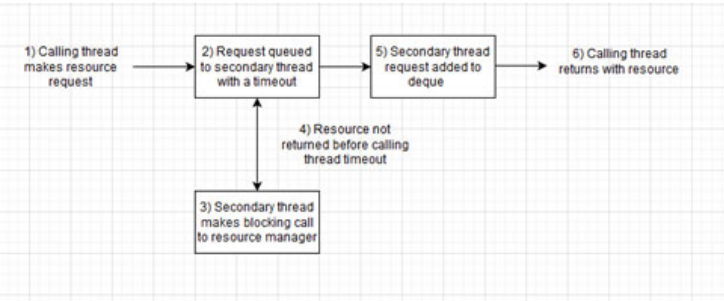
If the blocking thread returns with a resource from the non-critical pool, that resource is returned from the member function.

Figure 1: Resource returned before timeout



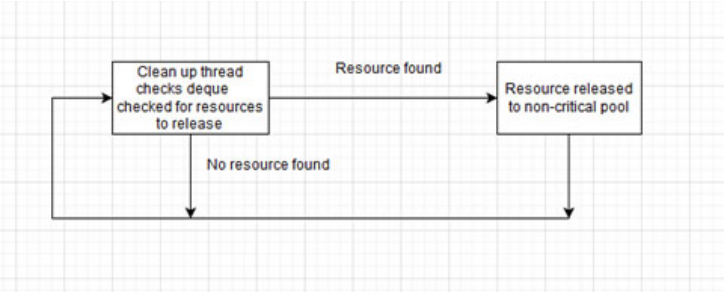
If the blocking thread does not return a resource before the timeout, the request being blocked on by the acquired thread is queued to the clean-up deque. The calling thread then blocks on the critical resource pool to acquire the resource.

Figure 2: Resource returned after timeout and added to clean-up thread



The clean-up thread checks the deque for non-critical resources that have been returned after their requests have timed out. If a non-critical resource is found in this deque, the clean-up thread releases that resource back to its pool.

Figure 3: Deque and clean-up thread



It is common for services to bulkhead sets of APIs so that each set is ensured a certain number of resources. This division is done to support availability and performance service level agreements for the APIs. When bulkheading is performed, the resources available to the service are segregated between the sets of APIs. While this bulkheading addresses availability and performance needs, it also restricts critical APIs to a subset of the total resources available to the service, even when non-critical resources are not in use by the other APIs. This restriction constrains critical APIs' performance, call rates, and ability to meet service level agreements.

Many resource pool managers block threads when retrieving a resource. This means that critical threads block on the critical pool, even when there might be free resources available in the non-critical pool. This solution allows critical APIs to use both pools without blocking by wrapping those standard blocking pool managers.

This solution is to create resource pools for each set of APIs – in this discussion, there are two pools (critical and non-critical). The non-critical APIs use their resource pool exclusively. The critical APIs call the non-critical resource pool with a timeout. If the resource is returned before timing out, that resource is used. If the request times out, the critical resource pool is called with an infinite timeout, and the call waits for a free resource.

Instead of limiting APIs to their own resource pools, this solution bulkheads resources so that the non-critical set of APIs is limited in its use of resources but also supports the availability and performance of the critical set of APIs by using resources from other pools.

For a developer to use this method, for each pair of critical and non-critical pools, they should create three things: a non-blocking resource manager, a clean-up thread, and a deque of resources to be released. The resource manager requests resources from the non-critical pool with a timeout. If a request is not processed before the timeout, the resource manager makes a blocking request for a resource from the critical pool. Once the original request from the non-critical pool returns, that returned resource should be added to the clean-up deque so that the clean-up thread can return it to its pool.

Disclosed by Donald Abel, Kendra Swafford and Jon K Lewis, HP Inc.