Impact of memory allocation on the performance of a delegation synchronization algorithm

Written by Riyane SID-LAKHDAR (M1 MoSIG)
Supervised by Thomas ROPARS (LIG, team ERODS)

June 20, 2016

Contents

1 Intro / Motivation

1.1 Mem alloc = botleneck for multithreading

Most modern operating systems allow applications to run many execution flows concurrently. This is basicaly what allows you to listen to music while you send your emails and organize your schedul (and do all this at the same time).

Theoretically, we could expect that this approach divides the execution time of any task by n, the number of processors or threads involved. However, as you probably know, this threashold objective is way of the mark.

The main problem face by multithreading comes from the complexity of the processor architecture: The processor communication may involve some on-ship networking. Thus, a memory access time may be multiplied by a factor up to 100.

Among the parts of an OS that limit the gain of parallelism, the memory allocation is one that is often disregarded but that can have a deep impact on performance.

1.2 Delegation algo

In this context, delegation is an example of multithreaded algorithms that may be impacted by the used allocation. Basically, delegation represents an approach to implement mutual exclusion. Let consider a set of threads that try concurrently to access a given critical section. The basic approach consists in synchronizing the threads to let them access the cs sequentially. Using the delegation approach, the execution of the CS will be delegated to a central thread (or a server thread). This thread will execute the CS instead of the others. Such an approach has allowed to implement the most efficient known queue for multithreading. And Different algorithms have been designed based on this approach. But the problem is that in literature, most of the studies that evaluate and asses this algorithm do not consider the impact of memory allocation on the performance that they measure.

1.3 Objective

Thus, in our study, we show the impact of the used allocation strategy on two custom delegation algorithms. Our objective is, (silence...) starting from a property which has already been stated about this algorithm. We want to prove that memory allocation may influence or even infirm such a known propert of multithreaded algorithms.

1.4 Presenting the outline:

To do so, we will present in a first part the main principle of the delegation approach. We also show the specification of the two custom algorithms that we consider.

In a second part, we focus on the memory allocation principle: we give an overview of the challenges of memory allocation, and a of the most efficient proposed solutions. Basically we show how the complexity of memory allocation makes us wonder how it could affect any multthreaded algorithm, and specifically the delegation ones that we consider.

In a final step, we present the evaluation that we made in order to compare and to choose the most efficient allocator for our workbench. We also present our experiments which show how the usage of the best chosen allocator infirms some of the previously stated properties about the delegation algorithms

2 Formal definition of the delegation algos

2.1 Dlegation approach

One of the main performance advantage of the delegation approach is the cache locality. Let me explain. As the critical section is executed by the same thread, the data used for this cs stay in the same cache. Thus it dramatically reduces the number of cache misses, and of Remote Memory References, which, as we saw, are extremely costly.

2.2 The benchmark algorithm: combiner

One of the most efficient state of the art algorithm that uses this approach is known as the combiner. The basis of this algorithm is that no dedicated thread is used to execute the critical section. When different threads try to access it, one of them is elected to execute its own CS as well as the one of a given number of others.

2.3 Proposed optimization of the delegation algo

To the delegation principle, two optimizations attempts have been proposed. Basically, this second algo tries to improve the communication between the server and the clients.

The first one consists in using a non temporal memory access to write the server answers. Thanks to this "streaming store", the server does not need to wait for his answer to be available for the client.

The second improvement of the algorithm consists ************

2.4 Limits of the evaluation of the improvements

An experimental evaluation of this two algorithms has been conducted in former researches. It considered some ubiquitous shared objects (stack, queues, ...) submited to contention. This study showed that the backoff and streaming optimization allows this shared objects to outperform compared to the combiner algorithm. However, this study has been lead using an unrealistic memory allocator. It's basically an allocator that uses a block of free memory mapped to each thread to allocate memory. More details about the principles of this allocator may be found in the paper that we handout. But for now, the important point is that such an allocator

- Outperforms within the specific environment of the test
- But, and this is the problem, it can definitely not be used as a general purpose one (manly because it can not scale to a large number of threads).

Knowing the complexity and the challenges faced by a memory allocator, we could wonder how the use of a general purpose allocator could change the former experimental results. And this is the point of our next section:

So in this second section, let focus on the challenges faced by any general purpose allocator

3 Allocation challenge and proposed solutions

3.1 Alloctor = bottleneck for multithreading

The main issue of a memory allocator is that it relays on sequential algorithms. This means that when different threads try to allocate memory concurrently, they have to wait for the allocator to answer to each request one by one: Thus it becomes a botleneck.

3.2 How to lighten the contention: core local buffer

...

Now to feed all this core local buffer, a shared structure is used. And the problem of the contention will appear again. But once again.

3.3 False sharing

Let consider .... The main problem is that at each local modification, the modification needs to be transmited to all the foreign caches, involving costly remote references. And this overhead is totally idle, as the addresses used by the different threads are independent.

So a way to face this challenge is the usage of a virtual span. A virtual span is a block of fixed size of free memory. Different ranges of sizes exists. But the interesting part is that all this sizes are multiples of a cache line size. Which means that at any time, no two different objects will share the same cache line. And this will collapse the impact of false sharing.

An other very interesting advantage of the virtual span is that no matter the size of object asked by the user, all the request will be satisfied using the same block of memory (virtual span). Which makes the allocation time constant, and simplifies the data structures used to manage this blocks.

At this point, you may think that a lot of extra memory may be used for small objects. And you're right. But, by simply aligning this extra memory to a page size, it will be stored into independent virtual pages. Hence swapped out of the RAM by the virtual page system. As the user never call this addresses, they will trigger no page fault. So to sum up all the extra memory used will have no space imoact and no time impact.

4 Results

4.1 Testing environment

Basically all this rules have been designed in order to test the allocators and the delegation algorithms without the inference of other parameter which are out of the scope of our study. For example we pin each thread on an independent core in order to get ride of the overhead linked to scheduling.

What we present next is the throughput reached by a shared queue (a Michael Scott queue), using two different hardware architecture: namely an Intel and an AMD architecture.

In the next experiments, we run an experimental comparison of the two delegation algorithms that we have presented

And we compare different memory allocator. First the custom allocator that has been used in the previous evaluation of the delegation algorithms. We also consider ptmalloc, which the allocator used by the C standard library. And different open source allocator that implement on or sevral strategies that we just described.

4.2 Choosing the best allocator

4.3 Evaluation of the delegation algo

5 Conclusion

During this study, we have shown that even using the best state of the art allocator, the behaviour of a multithreaded algorithm may be opposed to its theoretical or expected behaviour. This means that from now on, the dynamic memory allocation needs to be considered as an influente parameter when ever you design or assess any shared data-structures.

So our study has shown experimentally the impact of the chosen memory allocator on a custom delegation algorithm. In order to validate and to better understand this result, one could wonder which design of the used allocator is responsible of this behaviour. What is the influence of the hardware architecture: For example how the hardware processor architecture. What is the influence of the kernel allocation policies. ***** What is the influence of the workload that we used on our results ********

This three points are the basis of our following research. But for now ...