Swap Fairness for Thrashing Mitigation

François Goichon, Guillaume Salagnac, Stéphane Frénot

Swap Fairness for Thrashing Mitigation

François Goichon, Guillaume Salagnac, and Stéphane Frénot

University of Lyon
INSA-Lyon, CITI-INRIA
F-69621, Villeurbanne

Abstract. The swap mechanism allows an operating system to work with more memory than available RAM space, by temporarily flushing some data to disk. However, the system sometimes ends up spending more time swapping data in and out of disk than performing actual computation. This state is called thrashing. Classical strategies against thrashing rely on reducing system load, so as to decrease memory pressure and increase global throughput. Those approaches may however be counterproductive when tricked into advantaging malicious or long-standing processes. This is particularily true in the context of shared hosting or virtualization, where multiple users run uncoordinated and selfish workloads.

To address this challenge, we propose an accounting layer that forces swap fairness among processes competing for main memory. It ensures that a process cannot monopolize the swap subsystem by delaying the swap operations of abusive processes, reducing the number of system-wide page faults while maximizing memory utilization.

1 Introduction

When the operating system is under memory pressure, the virtual memory manager picks some arbitrary memory pages and temporarily moves them out of RAM, to free up some space. Whenever such a page is later requested, the hardware generates a *page fault*, i.e. an event informing the OS that the page needs to be reloaded –*swapped in*– again in RAM. While swapping gives applications the illusion of an "infinite" memory, it may turn into a severe performance bottleneck, as accessing secondary storage is several orders of magnitude slower than main memory. Under high memory pressure from multiple tasks, the system has to constantly swap pages in and out, yielding low CPU utilization. This state is called thrashing, and is a common issue on shared systems, where the activity of a unique user may have a significative impact on the system behavior [5].

Classical strategies [1, 3, 4, 11] to avoid thrashing either reduce global memory utilization or are very expensive to implement. More recently, Jiang et al.'s *token-ordered LRU* policy [9] proposes to protect the most memory-intensive task by keeping all its pages in main memory. This approach expects a quicker termination of that task and release of its memory pressure.

However, modern systems are typically running uncoordinated workloads, from multiple users. In this context, trying to favour the most memory-hungry task may obliterate lighter tasks performance, especially when heavier tasks are long-standing. An obvious solution to this issue would be to use memory quotas. However, figuring out suitable settings is challenging, and even impossible when the workloads are not known in advance. Moreover, statically pre-allocating space leads to poor memory utilization.

On the contrary, our approach tries to dynamically minimize the impact of deviant behaviors, while still maximizing memory utilization. Our idea is to intercept and manipulate page faults: if N users contend for memory, then we restrict each one to be causing no more than a $\frac{1}{N}$ fraction of the overall swapping time. Delaying swap-in requests of abusive users reduces the total number of page faults and allow other tasks to run more smoothly.

2 Related Work

This section first reviews existing approaches aiming at detecting or preventing thrashing. We then discuss past work on fairness for disk usage, which is an important topic in the real-time systems community.

Thrashing Mitigation. Past proposals mainly focus on reducing system-wide load. The idea is to first try and evaluate the needs of each task, and then to take appropriate actions when memory pressure arises, from task suspension to bin-packing and memory-aware scheduling [3, 4, 11]. Even though they were implemented [6, 12], such approaches increase overall throughput at the expense of increased latency for individual tasks, and cannot adapt to fluctuating memory demands [10].

To cope with dynamicity, Jiang and Zhang [8, 9] propose to temporary protect the most memory-intensive task: while protected, the task's pages cannot be swapped out. If the process finishes faster, its memory is freed, reducing system-wide memory pressure and early thrashing peaks. The Linux kernel implements this idea, starting with v2.6.9, with the name *swap token*. The main limitation of this approach is the hypothesis that memory-hungry tasks are transient. If several long-running tasks compete for memory, the swap token is of no help, and can even make things worse.

On the contrary, the local page replacement policy [1] aims at isolating tasks performance, as it restricts every process to only swap out its own pages. However, this idea requires specific memory allocations schemes [2, 7], which are difficult to fine-tune, and do not maximize memory space utilization.

Disk usage fairness. To improve performance, the OS typically interposes several software layers between user programs and hardware devices. Each layer reorders requests to increase overall throughput. Unfortunately, this enables adverse effects caused by one task to have a significant impact on other tasks. For instance, abusing filesystem locality may cause request starvation, as both the operating system's I/O scheduler and on-disk schedulers try to minimize disk head movement [15]. Many techniques have been proposed to address request starvation, such as draining the request queue [13], or dynamically adapting the number of best-effort requests allowed to be passed to the I/O scheduler, considering missed deadlines by real-time requests [14]. Whereas these ideas are designed for real-time environments, they apply the idea of sporadic scheduling as a fairness mechanism for device requests, focusing on delaying requests that can negatively impact the reponse time of others.

Discussion. While most approaches to mitigate thrashing ignore individual performance, others [1] bring up the idea of performance isolation, a requirement on shared platforms. On the other hand, researchers from the real-time community have developped dynamic approaches to reduce or bound the maximum duration of disk requests. In the next section, we propose an approach aiming at controlling the fairness of swap usage, to reduce page faults from memory heavy processes, thus bounding the impact of deviant workloads while still maximizing memory utilization.

3 Our Approach

Our approach to mitigate thrashing is to enforce fairness among the different users requesting memory. We refer to those users as *swapping domains.* A swapping domain may consist of one process, or of all the processes of a system user or system group. In this section, we present our approach with further detail and argue why fairness on swap operations can help to mitigate thrashing.

Fairness on swap operations to mitigate thrashing. The natural circumventions to domains monopolizing main memory - quotas or local page replacement [2] - do not use the system's space at its full potential and are hard to setup in practice. Our approach to deal with this problem is to disregard space usage, and instead to account for the amount of *work* that each domain induces on the swapping subsystem. When the system is thrashing, it means that one swapping domain must be preventing others to establish their working sets. This implies that this domain has to constantly produce page faults to keep its own pages in main memory. Therefore, the system spends more time in swapping operations on behalf of this particular domain than for other domains. Our approach aims at detecting this situation, and reacting by delaying requests from abusive domains until other domains have had their share of swapping time. This may increase the actual execution time of memory-heavy processes but reduces global page faults rate, while providing non-abusive domains with guaranteed periods of time where their pages will remain available. This strategy is almost the opposite of the swap-token approach, in which a process causing more page faults becomes less likely to have its pages swapped out.

Approach formalization. We consider a set of N swapping domains $\{D_i\}_{i \in \{1..N\}}$, and we write $S(D_i)$ to denote the cumulated duration of swapping operations caused by domain D_i. A domain D is said to be abusive if $S(D) > \frac{1}{N} \sum S(D_i)$. In other words, a domain is abusive when it induces more, or longer, swapping operations than other competing domains. Whenever a swapping domain is detected as being abusive, we delay all its future swap-in operations until $S(D) \leq \frac{1}{N} \sum S(D_i)$ holds again.

The main OS events required to calculate the $S(D_i)$ are the swap request issue dates and completion dates: by calculating the time difference between these two events we can deduce the precise duration of each swap-in operation. The sum of these durations reflects the pressure that a particular swapping domain is putting on the virtual memory subsystem. Please note however that neither the disk devices nor the software I/O layers have a FIFO behavior. As a result, multiple disk requests sometimes cost less than a single one. Therefore, computing the duration of the requests and not just counting them provides a sounder basis for the accounting. Implementing such an approach in a monolithic kernel brings up many efficiency concerns, that we detail and address in a companion technical report [5].

Prototype. To evaluate our idea, we implemented our accounting layer within the Linux kernel, and compared its performance and its fairness to the Linux swap-token implementation. As expected, when a memory-hungry (maybe malicious) process runs for a long enough time, then the swap token makes it harder for other legitimate processes to execute smoothly. With our swap accounting layer, legit workloads that are intensive in memory are allocated more swap time than with the swap token, and their performance is improved significantly as a result. Moreover, forcing fairness on the swap-in operations is equivalent to force a general fairness in terms of computation, and induces a better predictability of execution duration. More details on our experiments results can be found in [5].

4 Conclusion

The problem of physical memory shortage, with thrashing as its side effect, has been an open problem for more than 50 years. As a result, the virtual memory subsystem has been widely studied and many improvements over the existing page replacement policies have been presented to allow concurrent processes to run more smoothly. The most recent step is the introduction of the token-ordered LRU, or swap token, which selects processes for LRU evasion. This mechanism allows processes with important memory demands to keep their pages in main memory and hopefully finish quickly enough to reduce system pressure.

In this paper, we highlight the fact that the swap token may be counterproductive when in presence of malicious or uncoordinated workloads that do not end their execution quickly. As an alternative, we propose a lightweight accounting layer that delays swap requests from processes monopolizing the virtual memory subsystem without any preliminary configuration. Such a system allows processes with legit memory needs to have normal access to the swap space at the expense of abusive processes. Our first results show that the approach is promising in terms of performance and fairness, and is well adapted to shared hosting platforms.

References

1. A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, 1971.
2. A. Alderson. Thrashing in a multiprogrammed paging system. Technical report, University of Newcastle, 1972.
3. P. J. Denning. Thrashing: its causes and prevention. In *1968 Fall Joint Computer Conference*, pages 915–922. ACM.
4. P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.
5. F. Goichon, G. Salagnac, and S. Frénot. Swap fairness for thrashing mitigation. Technical report, INRIA, 2013.
6. Hewlett-Packard. *HP-UX 11i Version 3: serialize(1)*, 2010.
7. S. Iyer. *Advanced memory management and disk scheduling techniques for general-purpose operating systems*. PhD thesis, Rice University, 2005.
8. S. Jiang and X. Zhang. TPF: a dynamic system thrashing protection facility. *Software: Practice and Experience*, 32:295–318, 2002.
9. S. Jiang and X. Zhang. Token-ordered LRU: An effective page replacement policy and its implementation in Linux systems. *Perform. Eval.*, 60(1-4):5–29, 2005.
10. J. B. Morris. Demand paging through utilization of working sets on the MANIAC II. *Commun. ACM*, 15(10):867–872, 1972.
11. M. Reuven and Y. Wiseman. Medium-term scheduler as a solution for the thrashing effect. *Computer J.*, 49(3):297–309, 2006.
12. J. Rodriguez-Rosell and J.-P. Dupuy. The design, implementation, and evaluation of a working set dispatcher. *Commun. ACM*, 16(4):247–253, 1973.
13. M. J. Stanovich, T. P. Baker, and A. I. Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Real-Time and Embedded Technology and Applications Symposium, RTAS 2008*.
14. J. Wu and S. Brandt. Storage access support for soft real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, RTAS 2004*.
15. Y. J. Yu, H. Shin, D. I.and Eom, and H. Y. Yeom. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. *ACM Transactions Storage*, 6(1):2, 2010.