INVENTEURS DU MONDE NUMÉRIQUE

Swap Fairness for Thrashing Mitigation

François Goichon, Guillaume Salagnac, Stéphane Frénot

ISSN 0249-6399 ISRN INRIA/RR--8388--FR+ENG

RESEARCH REPORT
N° 8388
June 2013
Project-Team socrate

Swap Fairness for Thrashing Mitigation

François Goichon[*], Guillaume Salagnac, Stéphane Frénot

Project-Team socrate

Research Report n° 8388 — June 2013 — 14 pages

Abstract:
The swap mechanism allows operating systems to manage more memory than the available RAM space, by temporarily storing unused memory pages on disk. However, disk transfers are way slower than normal RAM operations, and under memory pressure, the system may spend more time in retrieving and storing swapped pages than performing actual computation: this state is called memory thrashing.
To reduce thrashing, several ideas were brought up to optimize page replacement algorithms and system-wide load. For instance, Linux currently implements the swap-token, a mechanism designed to immunize the memory pages of the heaviest process against swapping. Such a mechanism eliminates early thrashing peaks and improves general system performance if the process is to finish quickly. The swap-token may however be counterproductive when it is 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.
In this paper, we present an accounting layer that forces swap fairness among processes competing for main memory. It ensures that a process cannot perform longer swap operations than others, and delays the swap operations of processes abusing the swapping mechanism. With such a layer, we are able to significantly reduce the dispersion of execution times under memory pressure, and generally improve the performance of legit, memory-heavy processes running concurrently with abusive ones.

Key-words: Operating Systems, Resource Control, Virtual Memory, Swap

[*] E-mail: francois.goichon@insa-lyon.fr

Mémoire virtuelle équitable pour réduction de l'écroulement

Résumé : Le mécanisme de swap permet à un système d'exploitation de travailler avec plus de mémoire que la quantité de RAM disponible, en déchargeant temporairement sur le disque les pages inutilitées. Cependant, les communications avec le disque sont beaucoup plus lentes que les opérations mémoires classiques. En cas de forte contention mémoire, le système peut donc se retrouver à passer l'essentiel de son temps à transférer des données depuis et vers le disque plutôt qu'à exécuter les programmes. On parle alors d'écroulement (thrashing).

Pour atténuer ce problème, les approches classiques proposent des algorithmes optimisés de remplacement de page, ou des techniques de réduction de la charge globale du système. Par exemple, Linux emploie une technique appelée jeton de swap (swap-token), conçue pour protéger l'un des processus du système contre tout déchargement de page. Ce mécanisme vise à permettre à ce processus de terminer son exécution le plus rapidement possible, de façon à faire redescendre la pression sur le swap. Cependant, le swap-token se révèle inefficace, voire contre-productif lorsqu'il se trompe et qu'il favorise un processus malveillant, ou simplement durable. Ce problème est particulièrement sévère dans le contexte de l'hébergement mutualisé ou de la virtualisation, où plusieurs utilisateurs doivent se partager une machine et y exécuter plusieurs charges de travail simulatnées et égoïste.

Dans ce rapport, nous présentons une couche de facturation qui force l'équité entre tous les processus en compétition pour la mémoire. De cette façon, un processus ne peut pas s'accaparer le mécanisme de swap, ni retarder déraisonnablement les requêtes de swap des autres processus. Cette couche permet de réduire significativement la dispersion des temps d'exécution de programmes en situation de contention mémoire, et de manière générale, d'améliorer la performance des application légitimes lorsqu'elles s'exécutent en concurrence avec des applications abusives.

Mots-clés : Système d'exploitation, Gestion de ressources, Mémoire virtuelle, Swap

1 Introduction

When the operating system is under memory pressure, the role of the virtual memory manager is to choose arbitrary pages to be moved out of main memory, so as to free up some RAM space. The content of these pages is temporarily stored on another storage device (e.g. the hard disk) called the *swap space*. Later on, when the program actually needs its data, the page is transparently reloaded from disk and *swapped in* again in RAM. This swapping mechanism allows the operating system to work with more memory than the available RAM space, but it may also turn into a severe performance bottleneck. Indeed, accessing an external storage device is generally several orders of magnitude slower than accessing main memory. Under high memory pressure from multiple tasks, the operating system may have to constantly swap pages in and out, yielding low CPU utilization. This effect is called thrashing.

Whereas efficient page replacement algorithms have been widely studied with the intent to optimize the process of swaping out the least used pages, little work has been done to mitigate or protect against thrashing. Past work, such as local page replacement [2], load control [5] or the working set model [6], either minimize the concurrency or memory usage on the system, or are very expensive to implement properly. Jiang et al. [10] propose a so-called *token-ordered LRU* policy meant to avoid early thrashing stages. The idea is to help the most memory-intensive program by keeping its pages in main memory. This approach relies on the assumption that this task will terminate quickly and thus will soon release its resources. In the meantime, other tasks may have to swap out pages more frequently, as the total number of pages contributing to the LRU is reduced. If the elected task does indeed terminate quickly enough, the overhead imposed on other tasks may be tolerable. However, modern systems have multiple users, running multiple tasks, and even multiple virtualized operating systems. In this context, trying to favour the most memory-hungry tasks may have the worst side effects, in particular when those tasks are long-standing.

Motivating example As an illustration of this situation, we consider a shared web hosting platform. Such services may provide web hosting for hundreds of users on the same machine. Each website runs with its own privileges to ensure proper isolation between each users's data.

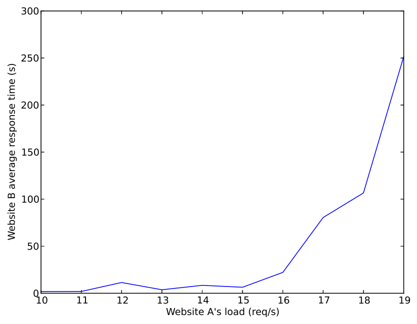


Figure 1: Lack of performance isolation on a shared hosting webserver

In this scenario, if one of the users decides to setup a memory heavy website, requiring high memory usage for each request, other lightweight websites may significantly suffer from memory contention. In the example below, we use a basic Linux webserver, with 512MB of RAM and only 2 users. User A runs a fresh install of Drupal 7.15, a memory-intensive PHP framework, and user B runs a more lightweight PHPNuke 8.1 website. Figure 1 represents the average response time of website B, with respect to the frequency of incoming requests for website A.

As expected, website B may be heavily impacted by the actual workload of website A, with HTTP requests taking up to 250 seconds. Moreover, from 15 requests per second, some requests either time out or return an erroneous HTTP code. As thrashing is a problem in real-life scenarii, it may be reinforced by potential malicious behaviors - external flooding, malloc bombs.

Obviously, enforcing memory quotas would prevent this situation from happening. But figuring out suitable settings is at least challenging, or even impossible when the workloads are unknown in advance. Moreover, each workload's needs may vary over time, thus space reserved in advance is wasted when not at peak demand. And the actual bottleneck which causes thrashing is not space usage per se, but swap request service time. Our approach tries to minimize the impact of potentially deviant behaviors, while still maximizing memory utilization and disk activity. We define *accounting domains* - process or user IDs - for swap usage and enforce the fact that, if N domains contend for memory, then each domain is restricted to causing no more than an $\frac{1}{N}$ fraction of the swapping time. Other approaches [10] help memory-heavy processes to establish and keep their working set in memory. On the contrary, we argue that lighter workloads should not be further impacted by heavier ones. Enforcing fairness, i.e. delaying the swap-in requests of heavier tasks, not only reduces the global number of page faults but also allows other tasks to run more smoothly.

We implemented an accounting layer within the Linux 3.2.5 kernel to enforce swap fairness among processes. The prototype is able to significantly improve the performance of legit processes running concurrently with abusive workloads. It also provides more constant results in terms of execution time, thus bounding the impact of inappropriate workloads on the system.

The remainder of this paper is organized as follows. Section 2 reviews past work on thrashing prevention and mitigation, as well as existing approaches aiming at improving fairness for disk requests. Section 3 presents our approach to mitigate thrashing for legit processes, while maximizing memory usage. It uses a kernel-level accounting layer that delays swap-in requests for abusive processes. Finally, section 4 details our prototype implementation within the Linux 3.2.5 kernel, as well as experimental results evaluating its impact on performance and fairness of legit applications under memory pressure.

2 Related Work

This section reviews existing approaches proposed to detect or prevent thrashing. As fairness is also a critical topic in the real-time community, we discuss past work on fairness for disk usage, which brings up different, dynamic methods to enforce fairness between disk users.

2.1 Thrashing Mitigation

Different ideas have long been proposed at the operating system level to mitigate thrashing effects. As early as 1968, Denning [6] established a model for program behavior, which splits a task activity in two distinct parts: its processor and memory demand. The *working set*, i.e. the current set of memory pages needed to perform most of the task's computation, has to remain in main memory as much as possible, for the task to finish quickly. Whenever too many working sets compete for main memory, thrashing may occur. The same author also propose a load control approach [5], to suspend or swap out tasks when thrashing is detected. The idea is therefore to reduce the general multiprogramming level (MPL). Several operating systems have adopted this method to some extent [7, 13]. However, building precise working sets in real time for each task is very expensive [11].

In the same vein of reducing the MPL, Reuven and Wiseman [12] propose to add another layer of scheduling to the Linux operating system: they group processes such that each group requires as close as possible to the actual physical memory amount. A so-called *medium-term scheduler* is introduced, which schedules these groups for execution in a round-robin fashion. Processes inside a group are scheduled by the classical (short-term) Linux scheduler. Swapping therefore occurs only at the beginning of a medium-term timeslice. Overall throughput is increased, at the expense of increased latency for each individual task. This approach cannot however solve the problem of abusive workloads requesting more memory than the available RAM space, nor adapt to short peaks of memory needs that compilation or compression applications typically require. Moreover, modern systems have multiple users, running multiple tasks, and even multiple virtualized operating systems. In this context, trying to favour the most memory-hungry tasks may have the worst side effects, in particular when those tasks are long-standing.

Another early proposal is the local page replacement policy [2], which forces a process to swap out one of its own pages when it requires more memory. Several operating systems implement this idea [8] as a way to isolate tasks performance. Local page replacement requires specific memory allocation policies [3], which intrinsicically do not maximize memory space utilization and are difficult to tune. However, we think that the local page replacement policy brings up a strong idea, which is the fact that tasks should have some sort of isolation, regarding to other tasks' memory needs.

Probably the most relevant and most widely deployed work regarding thrashing prevention is the *swap token*, proposed by Jiang and Zhang [10]. Their work is based on the observation that when the system is thrashing, pages may be marked inactive simply because the task endures significant delay from the virtual memory subsystem. Under memory pressure, the authors propose to choose a certain task and forbid this task's pages from being replaced for a certain period of time. This way, the elected process is given a chance to establish its working set in memory and hopefully finish quickly enough for the thrashing effect to be negated. Their approach significantly improves the behaviour of memory intensive programs and eliminates early thrashing peaks. The swap token thus has been implemented in Linux, starting with kernel version 2.6.9. The first implementation assigned the token randomly, and for a very short period of time [4, Ch. 17]. The current version implements a priority counter, which increments with the number of swap out events for a specific task. A task may then "preempt" the swap token if its priority counter gets higher than the current holder's. The main limitation of the swap token approach is the underlying hypothesis, i.e. assuming that all memory-hungry tasks are transient. If several long-running tasks compete for memory, the swap token is of no help in improving the behaviour of the system.

2.2 Disk usage fairness

In order to increase overall performance, the OS typically interposes several software layers between user applications and the physical storage device. However, adverse effects caused by one task may still have a significant impact on other tasks using the device. For instance, abusing filesystem locality is a way to monopolize disk time, as both the operating system's I/O scheduler and on-disk schedulers will try to minimize disk head movement [15].

Disk starvation is a serious problem in the real-time community, and many techniques have been developped to bound disk requests time or force disk usage fairness. Stanovich et al. [14] propose to drain the disk request queue, delaying further requests, whenever a request is detected as spending too much time in queue, providing a priori response time guarantees. Wu and

Brandt [1] work in a model where two distinct request types, best effort and soft real time, compete for disk usage. They dynamically adapt the number of best effort requests allowed to be passed to the I/O scheduler, considering the history of missed deadlines by real-time requests. 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 putting on hold requests that can negatively impact the reponse time of others.

Past work in thrashing mitigation cover different approaches: some aim at system-wide best effort performance and others at tasks isolation. System-wide approaches such as the swap token are more efficient in the general case and are deployed in common operating systems. We believe that general throughput should not be improved at the expense of performance isolation, as malicious or uncoordinated workloads can have a huge impact on other processes. This is especially true on shared servers or hypervisors. 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 force fairness among different users requesting memory. We refer to those users as swapping domains. A swapping domain may consist of one process, or of all processes owned by the same 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, and show how such an approach can be implemented.

Fairness on swap operations to mitigate thrashing. Whenever a particular swapping domain monopolizes the main memory, it forces memory pages from other legitimate domains to be swapped out. Processes from legitimate domains always have to pay for the swap-in operations at the beginning of their time slice and therefore achieve very low computation throughput. The two natural circumventions to this problem are memory quotas and local page replacement [3]. However, both of these solutions 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 per se, and instead to account for the amount of *work* that each domain induces on the swapping subsystem.

When the system is in a thrashing state, it means that a swapping domain prevents others to establish their working sets in memory. It implies that this domain constantly produces page faults and its pages are abusively overwhelming main memory. As this swapping domain has to produce lots of page fault, it triggers as much swap-in operations, to bring its pages in main memory whenever they were swapped out. Therefore, the system spends more time in swapping operations on behalf of this particular domain than for other domains. Our approach aims at detecting whenever such a scenario occurs, and reacting by delaying requests from abusive domains until other domains have required as much swapping time. This increases the actual execution time of memory heavy processes but reduces global system page faults and swapping operations, while providing non-abusive domains with guaranteed periods of time where their pages are not being swapped out.

This approach is almost the opposite of current swap-token implementations, where a process is less susceptible to have its pages swapped out as it provokes more page faults.

Approach formalization. Let N be the number of swapping domains having produced at least one page fault over an arbitrary period of time. Let D_i refer to the swapping domain $i \in 1..N$, and $S(D_i)$ the cumulated time of swapping operations on behalf of domain D_i over the same period of time.

A domain D is said to be abusive if $S(D) > \frac{\sum S(D_i)}{N}$. It means that a domain is not abusive as long as it does not induce more, or longer swapping operations than other domains. One may note that if there is only one swapping domain, it cannot be considered abusive, as other processes do not require memory.

Whenever a swapping domain is detected as being abusive, we delay its future swap-in operations until $S(D) < \frac{\sum S(D_i)}{N}$ again.

The main operating system events required to account for swapping operations time are the swap requests and swap requests completion events. Those two events are sufficient to efficiently account for swapping operations and delay abusive swapping domains. Algorithms 1 and 2 further detail the intuition of our approach, based on those two entry points.

Algorithm 1 executed for each swap-in request R (of domain D)

```
if $abusive(D) = True$ then
    $t_0(R) \leftarrow now()$
    forward $R$ to lower layer
else
    put $R$ in delayed queue
end if
```

Algorithm 2 executed when swap-in request R is completed

```
$S(D) \leftarrow S(D) + now() - t_0(R)$
for $i \in 0..N$ do
    $abusive(D_i) \leftarrow True$
    if $S(D_i) < \frac{\sum S(D_i)}{N}$ then
        $abusive(D_i) \leftarrow False$
        if $D_i$ has delayed requests then
            Requeue delayed requests for $D_i$
        end if
    end if
end for
```

By calculating the difference between the request completion and the swap request, one can deduce the time during which the requester was waiting for a swap-in request to complete its memory operation. The sum of those durations expresses the pressure that a particular swapping domain puts on the virtual memory subsystem as a whole. One may note that disk devices do not have a linear behavior. As a result, multiple requests may actually cost less than a single disk request. Therefore, processing the duration of those requests and not their count provides a better, sound basis for the accounting.

Efficient accounting. With any fairness enforcement mechanism comes the problem of hysteresis. Indeed, a direct implementation of algorithm 2 would force an abusive domain to stumble on the acceptable limit, and its status would be changed with almost each incoming request.

Moreover, it requires non-trivial computation for each request completed and could significantly impact the virtual memory subsystem's performance.

To overcome these weaknesses, the different $S(D_i)$ have to be grouped for a short period of time, and the sums processed at once. This way, the swapping domains status remain unchanged until the global sums are processed, reducing the stumble effect and removing any computation else than the actual value of $S(D_i)$ from request completion events.

Finally, the swap-in request and request completion events have to be measured as close to the hardware events as possible. Indeed, measuring at too high a level would also include the operating system's I/O scheduling and can have significant effects on the actual accounting, as explained in [15].

4 Experimental Evaluation

This section exposes our implementation of swap accounting within the Linux kernel, as well as experimental results studying the impact of our accounting layer on concurrent, memory-heavy processes. We show that such a layer significantly reduces the dispersions of execution times under memory pressure, and generally improves the performance of legit, memory-heavy processes running concurrently with abusive ones.

4.1 Linux Implementation

Our approach would be ideally suited to be implemented within a fully modular operating system architecture, such as L4. In such an OS, it would be convenient to add a wrapper between two components so as to transparently intercept every *swap-in* request. However, due to the the current lack of driver implementations and swap subsystems for most L4 instances, we implemented our prototype within the Linux kernel 3.2.5.

Implementation overview. Figure 2 provides an overview of the swap in mechanism in the Linux kernel. Our implementation mainly hooks three kernel events : swap-in request creation, start of disk request processing and swap-in request completion.

1. Swap-in request creation - *swap_readpage()*. The accounting layer marks the block request as a swap-in operation. If the request's swapping domain is abusive, the request is appended to the swapping domain's delayed requests queue. Otherwise, the request is transmitted to the block layer.

2. Start of disk processing - *blk_start_request()*. If the request is marked as a swap-in operation, the accounting layer fills its start time.

3. Swap-in request completion - *end_swap_bio_read()*. If the request is marked as a swap-in operation, the accounting layer saves fills its end time and adds it to the list of processed requests.

The actual update of swapping domains status is deferred to a new kernel thread, kswpacc. kswpacc removes every request from the list of processed requests and updates the swapping domain's swap duration and abusive status. For every swapping domain that is not abusive anymore, delayed swap-in requests are transmitted to the block layer.

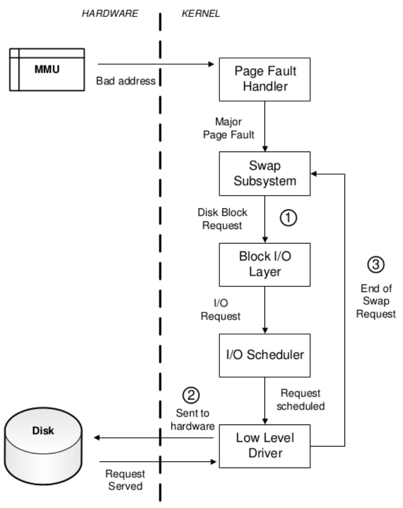


Figure 2: Hooks in the Linux kernel swap in process

Efficient accounting. To address the efficiency concerns brought up in 3, times are indeed measured as close to the hardware as possible: the start time is measured within the last function called by the low level driver before hardware processing, and the end time is measured within the first generic function called after a swap-in request has been processed. Moreover, the fact that non-trivial computations are performed on a group of requests by a kernel thread reduces the stumble effect, as multiple requests are processed at once when the kernel has spare time to do so.

Limitations. In this prototype, a swapping domain is defined as a single process. Therefore, fairness cannot be enforced yet on other types of domains such as users, user groups or cgroups. Moreover, the prototype accounts for swap durations from the beginning of the life of a process to its end only. While this is sufficient to study the effects of our approach on most workloads, time windows should be implemented to make accounting possible in more complex scenarios such as periodic workloads.

This prototype is available as a patch against the 3.2.5 Linux kernel[1]. It has been tested on Intel x86 platforms only.

4.2 Experimental Results

Benchmark details. We used several real-life and synthetic benchmarks to evaluate the actual fairness of our system, as well as its avantages and drawbacks in terms of performance.

In order to focus on the swap susbsystem behaviour, we try to avoid side effects by writing a first set of simple programs we called *swappers*. Each of these programs allocates a large, fixed portion of the available RAM, and then performs a simple memory operation sequentially on every memory page. The idea is to try and minimize the effects of caches and other features of the system, while maximizing swap activity. In our evaluation, we distinguish between *infinite swappers* which perform those operations indefinitly, and *timed swappers* which perform those operations during a defined period of time, and record the duration of each round, or cycle on the allocated memory.

[1]Patch available at http://perso.citi-lab.fr/fgoichon/linux-3.2.5_swpacc-1.2.patch

The second set of applications is taken from the SPEC CPU2006 suite, which contains programs designed to provide performance measurements and are usually intensive for both CPU and memory. More specifically, we used 401.bzip2 and 403.gcc, as they both represent common memory-intensive workloads that still have very different memory and CPU usage profiles.

Each SPEC program comes with several input files of various sizes. Table 1 summarizes all the benchmarks we use, as well as memory consumption and nominal execution times (i.e. completion time when run alone).

Name	Input	Average Duration (s)	Maximum VM size (MB)
401.bzip2	chicken.jpg	53	100
401.bzip2	liberty.jpg	66	100
401.bzip2	input.combined	106	610
403.gcc	166.in	66	200
403.gcc	c-typeck.in	85	430
403.gcc	g23.in	287	840
Timed swapper	-	10,000	200
Infinite swapper	-	∞	400

Table 1: Benchmark information

We performed two set of experiments, both allowing us to evaluate the fairness and the impact on performance of our prototype.

The first set of experiments studies the behavior of the prototype under high memory pressure from multiple applications : we launch several timed swappers on large periods of time and study the regularity of their memory cycles, as well as the number of cycles processed.

The second set of experiments studies the impact on real-life, intensive applications taken from the SPEC CPU2006 suite: those applications are executed in parallel of a single infinite swapper, allowing us to evaluate the impact on performance as well as the deviation of those measures over multiple runs.

Each experiment is executed on our modified Linux 3.2.5 kernel enforcing swap fairness as well as on a basic 3.2.5 kernel without any modification. The system used for those experiments is a Dell Latitude 1.66 GHz with 512 MB of RAM and 1 GB of swap space, containing a fresh Debian Squeeze installation.

Performance. In figure 3a, we run 5 concurrent timed swappers with the unmodified Linux kernel, and plot the duration of each iteration of each program. Figure 3b presents the results of the same experiment with our prototype.

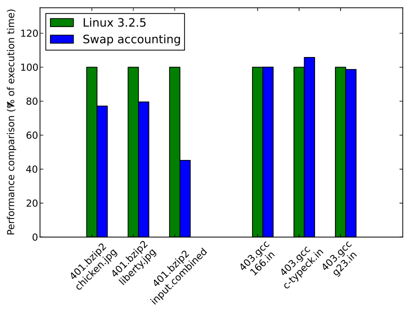
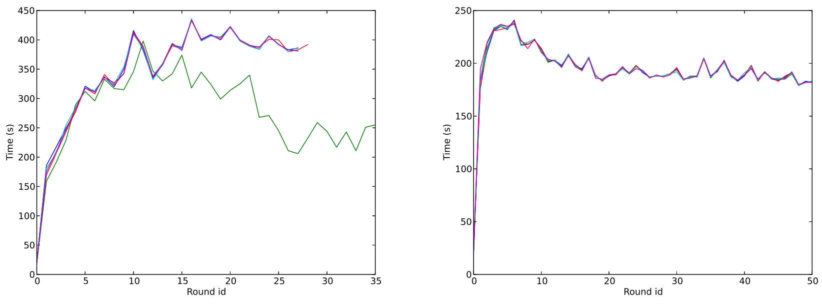


Figure 4: Performance comparison of SPEC CPU 2006 workloads in parallel with an infinite swapper



(a) Linux 3.2.5 Vanilla (b) Linux 3.2.5 + Swap accounting

Figure 3: 5 parallel swappers - Round duration

The first result is that with our prototype, the timed swappers perform 50 rounds during the experiment, whereas the best performer on the vanilla kernel achieves only 35 rounds. The difference is clearly visible: the average round durations are respectively 193 and 328 seconds. The process holding the swap token in the basic kernel case is clearly visible, yet it does not achieve a better performance than any process on our prototype, except for the first few rounds. One must note that swappers are the best case for swap accounting, as they access memory every few instruction.

Figure 4 shows the performance differences between 20 runs of SPEC CPU 2006 applications running in parralel with an infinite swapper on a basic kernel and on our prototype.

The bzip application is very intensive in large memory accesses, as it needs to scan all the data multiple times to perform the compression. On the other hand, gcc performs successive CPU-intensive computations on more localized areas. Therefore, the swap accounting works very well in the bzip case, as performance is very significantly increased, whereas gcc sometimes cannot perform its computations whenever it is marked as abusive, even if it does not need more memory pages. This drawback should be reduced or eliminated by not accounting for the full life span of a process, but for more reduced time windows.

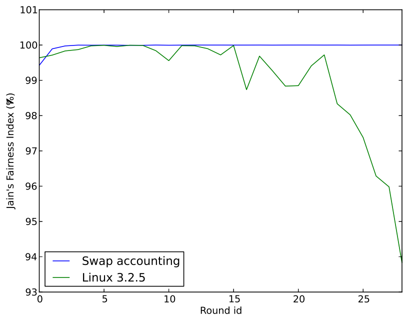


Figure 5: Jain's fairness index (computed on round durations) for 5 parallel swappers

Name	Input	σ (s)	σ with swap accounting (s)
401.bzip2	chicken.jpg	5.00	3.61
401.bzip2	liberty.jpg	4.00	2.65
401.bzip2	input.combined	46.38	5.92
403.gcc	166.in	28.74	11.96
403.gcc	c-typeck.in	121.68	29.80
403.gcc	g23.in	474.08	57.43

Table 2: Dispersion of SPEC CPU 2006 workloads execution time in parallel with an infinite swapper

Fairness. Figure 3 already shows that 5 timed swappers are very well synchronized in our prototype case, compared to the basic kernel case. To further demonstrate this point, Figure 5 compares the fairness of each round, using Jain's fairness index [9]. Only the first 28 rounds were compared, as only the swap token holder was able to achieve more rounds in the basic kernel case.

It shows that our prototype struggles for the first few rounds to establish a fair balance between each process, but then stays asymptotic to 100%, whereas the fairness is not as good in the swap token case, and becomes worse as the experiment goes on.

Table 2 shows the dispersion of execution times for 20 runs of SPEC CPU 2006 applications running in parallel with an infinite swapper, with or without our prototype.

The table clearly shows that execution times are more regular with swap accounting. The dispersion is always lower, up to 87% less in the case of the bzip2 compression of input.combined. An application may not reach its best performance with swap accounting, as it is the case for the gcc benchmarks, but it still bounds the impact of deviant workloads on legit ones.

Discussion. The swap token is designed to give an edge to one arbitrary process with the hope that it will finish quickly enough to reduce the MPL of the system. As expected, in cases where this process never ends or runs for a prolongated amount of time, 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. 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. The prototype should however be improved to deal with applications that are more intensive in CPU cycles that in memory, as they currently suffer from unwarranted idle times.

5 Conclusion

The problem of physical memory shortage, with thrashing as its side effect, has been an open problem for almost 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 to the swap token, we propose a lightweight accounting layer that delays swap requests from processes monopolizing the virtual memory subsystem more than others, 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.

We implemented such a mechanism on the Linux 3.2.5 kernel and were able to significantly speed up legit processes that otherwise struggle under memory pressure. Moreover, the accounting layer bounds the impact of abusive workloads on legit ones, as the execution times of tested workloads have low dispersion compared to their counterparts with the classic swap-token kernel.

In the future, the accounting layer should be completed, to deal with process groups and more complex workloads such as periodic applications or virtualized systems. Accounting over time windows should allow such work, and would open more evolved perspectives such as automated feedback. For instance, the accounting layer could adapt priorities or time windows depending on the recent behavior of the accounted applications, or switch to or from swap-token mode when needed.

References

[1] Storage access support for soft real-time applications. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '04, pages 164–, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, January 1971.

[3] Albert Alderson. Thrashing in a multiprogrammed paging system. Technical report, University of Newcastle, 1972.

[4] Daniel P. Bover and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly, 2005.

[5] Peter J. Denning. Thrashing: its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

[6] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[7] Hewlett-Packard. *HP-UX 11i Version 3: serialize(1)*, 2010.

[8] Sitaram Iyer. *Advanced memory management and disk scheduling techniques for general-purpose operating systems.* PhD thesis, Rice University, 2005.

[9] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. Technical report, Digital Equipment Corporation, 1984.

[10] Song Jiang and Xiaodong Zhang. Token-ordered lru: an effective page replacement policy and its implementation in linux systems. *Perform. Eval.*, 60(1-4):5–29, May 2005.

[11] James B. Morris. Demand paging through utilization of working sets on the MANIAC II. *Commun. ACM*, 15(10):867–872, October 1972.

[12] Moses Reuven and Yair Wiseman. Medium-term scheduler as a solution for the thrashing effect. *Comput. J.*, 49(3):297–309, May 2006.

[13] Juan Rodriguez-Rosell and Jean-Pierre Dupuy. The design, implementation, and evaluation of a working set dispatcher. *Commun. ACM*, 16(4):247–253, April 1973.

[14] Mark J. Stanovich, Theodore P. Baker, and An-I Andy Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 331–341, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. Ncq vs. i/o scheduler: Preventing unexpected misbehaviors. *Trans. Storage*, 6(1):2:1–2:37, April 2010.

INVENTEURS DU MONDE NUMÉRIQUE

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399