Professional Linux® Kernel Architecture

Wolfgang Mauerer

WILEY

Wiley Publishing, Inc.

18.7 The Swap Token

One of the methods to avoid page thrashing is the swap token, as briefly discussed in Section 18.1.2. The method is simple but effective. When multiple processes swap pages concurrently, it can occur that most of the time is spent writing the pages to disk and reading them in again, only to swap them out again after a short interval. This way much of the available time is spent writing pages back and forth to disk, but little progress is achieved. Clearly, this is a rare situation, but nevertheless a very frustrating one if an interactive user sits on his chair and watches the activity on the hard disk, while nothing is actually being achieved.

To prevent this situation, the kernel makes one and only one of the processes that currently swap in pages the owner of the so-called swap token. The benefit of having the swap token is that pages of the holder will not be reclaimed — or will, at least, be exempted from reclaim as well as possible. This allows any swapped-in pages to remain in memory, and increases the chance that work is going to be finished.

Essentially, the swap token implements a sort of ''superordinate scheduling'' for processes that swap in pages. (However, the results of the CPU scheduler are not modified at all!) As with every scheduler, fairness between processes must be ensured, so the kernel guarantees that the swap token will be taken away from one process after some time and passed on to another one. The original swap token proposal (see Appendix F) uses a time-out after which the token is passed to the next process, and this strategy was employed in kernel 2.6.9 when the swap token approach was first integrated. During the development of kernel 2.6.20, a new scheme to preempt the swap token was introduced; how this works is discussed below. *It is interesting that the swap token implementation is very simple and consists of only roughly* 100 lines — this proves once more that good ideas need not be complicated.

The swap token is implemented by a global pointer to the `mm_struct` of the process that is currently owning the token[14]:

mm/thrash.h
```
struct mm_struct *swap_token_mm;
static unsigned int global_faults;
```

The global variable `global_faults` counts the number of calls to `do_swap_page`. Every time a page is swapped in, this function is called (more about this in the next section), and the counter is increased. This provides a possibility for deciding how often a process has tried to grab the swap token in contrast to other processes in the system. Three fields in `struct mm_struct` are used to answer this question:

<mm_types.h>
```
struct mm_struct {
...
        unsigned int faultstamp;
        unsigned int token_priority;
        unsigned int last_interval;
...
}
```

[14]Actually, the memory region could be shared among several processes, and the swap token is associated with a specific memory region, not a specific process. *The swap token could therefore belong to more than one process at a time in this sense. In reality,* it belongs to the specific memory region. To simplify matters, however, assume that just one single process is associated with the memory region of the swap token.

faultstamp contains the value of global_faults when the kernel tried to grab the token last. token_priority is a swap-token-related scheduling priority that regulates access to the swap token, and last_interval denotes the length of the interval (again in units of global_faults) during which the process was waiting for the swap token.

The swap token is grabbed by calling grab_swap_token, and the meaning of the aforementioned values will become clearer by inspecting the source code:

mm/thrash.c
```
void grab_swap_token(void)
{
        int current_interval;
        global_faults++;
        current_interval = global_faults - current->mm->faultstamp;
...
        /* First come first served */
        if (swap_token_mm == NULL) {
                current->mm->token_priority = current->mm->token_priority + 2;
                swap_token_mm = current->mm;
                goto out;
        }
...
```

If the swap token is not assigned to any process yet, it can be grabbed without problems. Jumping to the label out will just update the settings for faultstamp and last_interval as you will see below.

Naturally, things are slightly more involved if the swap token is currently held by some process. In this case, the kernel has to decide if the new process should preempt the old one:

mm/thrash.c
```
        if (current->mm != swap_token_mm) {
                if (current_interval < current->mm->last_interval)
                        current->mm->token_priority++;
                else {
                        if (likely(current->mm->token_priority > 0))
                                current->mm->token_priority--;
                }
                /* Check if we deserve the token */
                if (current->mm->token_priority >
                                swap_token_mm->token_priority) {
                        current->mm->token_priority += 2;
                        swap_token_mm = current->mm;
                }
        } else {
                /* Token holder came in again! */
                current->mm->token_priority += 2;
        }
...
```

Consider the simple case first: If the process requesting the swap token already *has* the token (the second else branch), this means that it swaps in a lot of pages. Accordingly, the token priority is increased because it is badly required.

If a different process holds the token, then the current task's token priority is increased if it has been waiting longer for the token than the holder had to, or decreased otherwise. Should the current token priority exceed the priority of the holder, then the token is taken from the holder and given to the requesting process.

Finally, the token time stamps of the current process need to be updated:

mm/thrash.c
```
out:
        current->mm->faultstamp = global_faults;
        current->mm->last_interval = current_interval;
        return;
}
```

Notice that if a process *cannot* obtain the swap token, it still can swap in pages as required but will *not* be protected from memory reclaim.

`grab_swap_token` is only called from a single place, namely, at the beginning of `do_swap_page`, which is responsible for swapping-in pages. The token is grabbed if the requested page cannot be found in the swap cache and needs to be read in from the swap area:

mm/memory.c
```
static int do_swap_page(struct mm_struct *mm, struct vm_area_struct *vma,
unsigned long address, pte_t *page_table, pmd_t *pmd,
int write_access, pte_t orig_pte)
{
...
        page = lookup_swap_cache(entry);
        if (!page) {
                grab_swap_token(); /* Contend for token _before_ read-in */
...
                /* Read the page in */
        }
...
}
```

`put_swap_token` must be employed to release the swap token for the current process when the `mm_struct` of the current swap token is not required anymore. `disable_token` takes the token away forcefully. This is necessary when swapping out is really necessary, and you will encounter the corresponding cases below.

The key to the swap token implementation lies in the places where the kernel checks if the current process is the owner of the swap token, and the consequences for the process if it has the swap token. `has_swap_token` tests if a process has the swap token. The check is, however, only performed at a single place in the kernel: when it checks if a page has been referenced (recall that this is one of the essential ingredients to decide if a page is going to be reclaimed, and that `page_referenced_one` is a subfunction of `page_referenced`, which is only called from there):

mm/rmap.c
```
static int page_referenced_one(struct page *page,
        struct vm_area_struct *vma, unsigned int *mapcount)
{
...
        /* Pretend the page is referenced if the task has the
           swap token and is in the middle of a page fault. */
        if (mm != current->mm && has_swap_token(mm) &&
                        rwsem_is_locked(&mm->mmap_sem))
                referenced++;

...
}
```

Two situations must be distinguished:

1. The memory region in which the page in question is located belongs to the task on whose behalf the kernel is currently operating, and this task holds the swap token. Since the owner of the swap token is allowed to do what it wants with its pages, `page_referenced_one` ignores the effect of the swap token.

   This means that the current holder of the swap token is not prevented from reclaiming pages — if it wants to do so, then the page is really not necessary and can be reclaimed without hindering its work.

2. The kernel operates on behalf of a process that does not hold the swap token, but operates on a page that belongs to the address space of the swap token holder. In this case, the page is marked as referenced and is therefore protected from being moved to the inactive list from being reclaimed, respectively.

   However, one more thing needs to be considered: While the swap token has a beneficial effect on highly loaded systems, it affects loads with little swapping adversely. The kernel therefore adds another check before it marks the page referenced, namely, if a certain semaphore is held. The original swap token proposal requires enforcing the effect of the swap token at the moment when a page fault is handled. Since this is not so easy to detect in the kernel, the behavior is approximated by checking if the `mmap_sem` semaphore is held. While this can happen for several reasons, it also happens in the page fault code, and this is good enough as an approximation.

   The probability is low that a page fault happens when the system requires only little or no swapping. However, the probability increases if the corresponding swap pressure gets higher. All in all, this means that the swap token mechanism is gradually being enforced the more page faults there are in the system. This removes the negative impact of the swap token on systems with little swapping activity but retains the positive effect on highly loaded systems.

18.8 Handling Swap-Page Faults

While swapping pages out of RAM memory is a relatively complicated undertaking, swapping them in is much simpler. As discussed in Chapter 4, the processor triggers a page fault when an attempt is made to access a page that is registered in the virtual address space of the process but is not mapped into RAM memory. This does not necessarily mean that a swapped-out page has been accessed. It is also possible, for example, that an application has tried to access an address that is not reserved for it, or that a nonlinear mapping is involved. The kernel must therefore first find out whether it is really necessary to swap in a page; it invokes the architecture-specific function `handle_pte_fault`, as discussed in Section 4.11 to do this by examining the memory management data structures.