Kernel Programming Lab 7: Under Pressure

Erwin
Tom

October 27, 2017

1 Introduction

1.1 Goals

The goal if this lab was to implement memory pressure management in the form of disk-swapping and OOM killing. The main goal for these memory management modules it to free up memory by swapping the least accessed pages to disk and thus free up memory in main memory. If one of the core modules of the kernel requests a page and the call fails, it will be the task of the OOM killer to free up memory as fast as possible by killing the application that occupies the most pages.

1.2 Architecture

- Physical swapping of pages is done by "swappy" swapping engine.
- Periodic page reclaiming is done by "kswapd" service, which uses swappy.
- Direct reclaiming is performed in critical situations, by having the OOM killer murders big environments to free up memory.
- Only user env memory (ELF regions and anonymous memory) marked as swappable.
- Static kernel memory not swappable, to prevent critical sections having to be swapped in.

2 Implementation

2.1 Swapping engine

The swappy engine is comprised of two service modules and direct functions. The service modules, swapout and swapin are both kernel threads and thus run every so often due to being scheduled. These modules simply handle all requests that are inserted in their respective queue. Each queue, for swapin and swapout, has one page allocated and thus they can store 512 and 1024 items respectively. The queue sizes should keep the swapper busy while the other deamons decide what to swap. When the queue is full, by using one of the 'static' functions to fill them, the queue inserter will return a error value and the request is simply dropped. By using flags one can wait for the inserter to finish successfully. The flag for a direct method is also available for swapping out and in and thus can bypass the queue and get a fast reclaim. Most direct functions, just regular functions to be called, are locked as they can be called from anywhere in the hierarchy.

The swapper also features a test case module that test the funcionality on startup to prevent accidental bugs that can trigger in the long run.

2.2 kswapd

kswapd is the service that determines what pages are to be swapped. It does this by iterating over all pages in the pages array and finding all referencing page table entries, setting the access bits to zero and continuing. It uses CLOCK iteration, i.e. by moving a head pointer over the entire array cyclically.

If the set memory threshold (default is 0.8) of in-use physical pages has been reached, kswapd will start swapping out pages who's pte accessbits have not been set since the last time it handled those pages. It offloads the actual swapping to swappy by adding a request to its queue.

This thus might fill up the swapout queue. Any failure to insert a page into the swapout queue is simply ignored, it will be tried again later anyway.

2.3 OOM killer

The OOM killer is a routine that counts for each user env how many non-shared pages it references. If a page is shared (e.g. page refcounter > 1) it ignores it as these pages will not be freed when the enviroment is killed. Killing is determined based on the hypothesised amount of pages freed. The env with with the highest score wins a ride with the env_free().

2.4 Critical page allocation

Some allocations in the kernel may not fail due to their critical function, such as inserting a page table or allocating a non-user enviroment. We've created a special function for this, called page_alloc_crit(). This method calls page_alloc(). If that fails, a direct (i.e. blocking) page reclaim is attempted by calling kswapd_direct_reclaim() and attempting another allocation up until the point where kswapd returns that no more pages can be swapped. At this point, the OOM killer is called and the allocation is tried one last time. If this is not successful, a kernel panic is thrown.

3 Discussion

3.1 Limitations

3.1.1 Kernel space swapping

Swapping of pages is limited to user envs only and thus kernel pages are not swapped. This is a potentially impactful limitation but comes forth out of the non-robust workings of kernel fault handling.

3.1.2 PTE lookup

The PTE lookup function (revsere_pte_lookup()) is a naive implementation that iterates over all page table entries in the page directory of each enviroment. This approach is slow and costly but easy to implement and robust and does not cost memory. The reverse lookup function uses a external iterator and each call returns a found pte until no pte is found and zero is returned. To support multi-cpu action, the iterator is allocated by the caller and is not a static internal state.

3.1.3 Reclaiming problem

One of the problems in the implementation is that the function for reading pages does not work via non-blocking the swappy swapin service. The ide_start_read() and ide_read_sector() calls are called in exactly the same way in both the direct (blocking) swapin and the queued swapin. Unfortunately, when reading a sector form a kernel service the buffer or memory target for the read does not seem to be affected at all.