Operating System Report - Assignment 2

Milin Narayan Bhade
Department of Computer Science and Automation, IISc, Bangalore
email : milinbhade@iisc.ac.in
Sr.No. - 21404

1 Problem Statement

Implementing an operating system abstraction (a system call) for user-space programs to save/restore their program state in memory. Implementation and comparision of context-based method as an alternative to the fork-based method in Linux. Adding a new system call (named "mmcontext") in Linux to enable fast operating system support for saving and restoring program state.

1.1 Design

A new system call (mmcontext) was introduced for saving and restoring context (anonymous memory) of a program/ application which can invoke this system call to save the program state before executing potentially unsafe code and to restore the saved program state after the unsafe code has finished.

System call accepts one argument either 0 for saving the context or 1 for restoring the saved context. Lazy copying of pages (same as COW) was used so that only those pages which has their content written/changed will be restored as other pages remains unchanged. To do this, system call makes all the pages corresponding to anonymous memory write protected. Whenever any changes to any of the write protected page is made, page fault is generated and the page to be written is saved in context in kernel heap memory. This way whenever we want to restore the context, it can be copied to the userspace from kernel memory.

1.2 Implementation

For implementing the functionality for saving and restoring the context, a new data structure called $struct\ mmcontext$ was added in $mmtypes.h$. This was used to hold the page content and address of the page when its context is saved. In $task\_struct$ a new entry was created called $struct\ list\_head\ context$ which is pointer to the linked list of context (pages) saved in kernel memory, and $context\_saved$ which was a boolean used to avoid saving context more than ones.

$mmcontext.c$ file contains the implementation of system call $mmcontext$. $save\_context$ and $restore\_context$ are the functions used for saving and restoring the context when $mmcontext$ is invoked with argument 0 or 1.

In $save\_context$, we iterate over VMA regions and check if it is a anonymous region of memory. An anonymous VMA is a memory region for a process with no backing storage. In other words, any private allocation pertaining to the process address space thats not file-backed can be termed as an anonymous page. Such pages are made write protect using $pte\_wrprotect$ function. This helps in achieving Copy on Write functionality. Whenever a write is attempted on such pages a page fault is generated and this pages needs to be copied to kernel memory for saving as context if $save\_context$ is invoked. This is done in wp_page_resuse funtion in $memory.c$. A new page is allocated in kernel memory using $alloc\_page(GFP\_KERNEL)$ function, here flag GFP_KERNEL indicates that allocation is in kernel memory. Page is copied from user space to kernel space using $copy\_from\_user$ and saved in $struct\ mmcontext$ allocated using $kmalloc$. So, only those pages which has modified data are saved in context to avoid unnecessary copying.

In $restore\_context$, pages stored in context of the process are restored back to the userspace by using $copy\_to\_user$ function. And also memory in kernel is freed using $\_\_free\_pages$ and $kfree$.

If no restore is called before process termination, then the kernel memory is freed in do_exit function in $kernel/exit.c$ to avoid memory leak.

2 Performance and Comparision

2.1 Performance

The primary difference between the two approaches of saving and restoring context is that fork relies on additional process to execute the unsafe code while the context-based method will save/restore program state without creating an additional process.

Following is the comparision between context-based and fork-based approach of the aforementioned goal:

Size of memory allocation in 2^n Bytes (n)	Fork-Based Throughput	Context-based Throughput	Speedup
5	14192	539896	38.04
10	14196	365425	25.74
15	8272	31245	3.69
20	513	791	1.54
25	17	20	1.18

It can be observed that as the size of memory allocation is increased speedup decreases. Fork based method relies on creating new process which takes time for setting up address space of new process and other related activities like context switch. But using context based approach this overhead can be reduced which results in getting high throughput. But as allocation size increases to large value, then the overhead of allocating large memory in kernel heap is far more than that of fork based approach and thus no significant speedup.

Different system configuration results in different values of speedups. Intel systems does TLB flush when changes are done in Page Table and thus has lower speedups than AMD systems.

2.2 Major Limitations

Following are the major limitations of the context based method using mmcontext system call:

1. As the size of the allocation is increased the speedup decreases, thus if process has large address space, then this method suffers w.r.t performance. Also, it creates copies of context which may not be used if restore is not called.
2. Memory limitations: Saving and restoring program state can require a significant amount of memory, especially if the program has a large working set or if the state contains a lot of data. This can be a concern in systems with limited memory resources.
3. While implementations, huge pages support was not considered. Thus page granularity copying is done.
4. Complexity: Implementing context-based methods can be more complex than using fork-based methods, especially if the application has multiple threads or complex control flow. This can make it more difficult to reason about and debug the application.
5. TLB thrashing: When the program state is saved and restored, it can cause TLB entries to be flushed and reloaded. This can lead to TLB thrashing, which is when the TLB is constantly being cleared and reloaded, resulting in a decrease in memory access performance.

2.3 Further Optimization

- As process size increases substantially, the main overhead is with copying data from user space to kernel memory. This involves memory overcommiting as we may not restore context and result in low memory availability. Thus to avoid this we can store context in disc space, but this may result in performance degradation.