E0253 - OPERATING SYSTEMS: ASSIGNMENT 3

NABARUN SARKAR[1]

CONTENTS

1 Introduction 2
1.1 Background 2
1.2 Objective of this assignment 2
1.3 Submission requirements 2
2 PART B 3
2.1 Design 3
2.2 Implementation `hello/hello.c` 3
3 Part C 7
3.1 User-space program used to measure the timing of system calls [2] 7
4 Part D 8
4.1 Performance comparison 8
4.2 Limitations 8

LIST OF TABLES

Table 1 The output of iterations 8

[1] *Department of Computer Science and Automation, IIsc, Bangalore, India*

1 INTRODUCTION

1.1 Background

The unsafe nature of C programming language does not mandate if a memory access is valid, e.g. if any arbitrary value is being used as a pointer. Hence any program can manipulate arbitrary locations in heap memory. Hence, it becomes the task of the user to make sure of correctness of code.

1.2 Objective of this assignment

In this project, we have to continue the system call implementation on the Linux kernel sources of version 5.0 and perform following operations:

1.2.1 *Building an operating system abstraction (a system call) for user-space programs to save/restore their program state in memory*

1.2.2 *Saving the program state (if the argument is 0). Any user-space program can invoke the system call to save its program state before executing potentially unsafe code*

1.2.3 *Restoring the program state (if the argument is 1). This will restore memory content to the same state as it were just before saving the context.*

1.2.4 *Performance comparison with the fork based method.*

1.3 Submission requirements

Description: We have to submit a tarball consisting of two things:

1.3.1 *A single kernel patch using git, including all changes in this patch-including the code submitted in assignment-0.*

1.3.2 *Report with a brief description of design, implementation and performance comparison with the fork-based method. Also discussion of the major limitations of your implementation and possible way to solve.*

---

2 PART B

2.1 Design

2.1.1 *Modifying `syscall_64.tbl` file*

Adding the following line at the end of the list.

```
335     common  my_precious             __x64_sys_my_precious
```

2.1.2 *Defining a new system call*

```
SYSCALL_DEFINE1(my_precious, int, a)
{
    /* Main code here */
}
```

2.1.3 *Adding a pointer of type `pte_t` in `vm_area_struct` in file `include/linux/mm_types.h`*

```
struct vm_area_struct {
    /*
     * Other code here
     */
    atomic_long_t swap_readahead_info;

    pte_t *vm_old_ptes;
    /*
     * Other code here
     */
}
```

2.1.4 *Adding a syscall definition in file `include/linux/syscalls.h`*

```
asmlinkage long sys_my_precious(int);
```

2.2 Implementation `hello/hello.c`

2.2.1 *Get the syscall argument into "a", find current task mm_struct and loop through all VMAs in that mm_struct and select only the anonymous VMAs*

```
SYSCALL_DEFINE1(my_precious, int, a)
{
  struct mm_struct *mm;
  struct vm_area_struct *vma;
  struct task_struct *task = current;
  unsigned long addr, vm_flags;
  pte_t *ptep, pte, old_pte;
  size_t count = 0, page_count;
  int retval = -EINVAL;
  struct page *page;
  spinlock_t *ptl;

  if (a == 0 || a == 1) {
    /*
     * the task list is protected by RCU
     */
    rcu_read_lock();
```

```
18    {
19      mm = get_task_mm(task); // take a reference to mm
20    }
21    rcu_read_unlock();
22
23    if (mm) {
24      /* iterate over all VMAs with the mmap semaphor held */
25      down_read(&mm->mmap_sem);
26      for (vma = mm->mmap; vma; vma = vma->vm_next) {
27        if(vma_is_anonymous(vma) && !vma_is_stack_for_current(vma)) {
28          vm_flags = vma->vm_flags;
29          count = 0;
30          page_count = (vma->vm_end - vma->vm_start) / PAGE_SIZE;
```

19. Get the current task mm struct
26. Loop through all the VMAs
27. Check if the VMA is anonymous or not and not for stack
29. Get the page count in that VMA

2.2.2 *If the argument is 0, the context is to be saved. Allocate memory to save the Page Table Entries (PTE)*

```
1           if (a == 0) {
2             /* Check if context is already available */
3             if (vma->vm_old_ptes != NULL)
4               return retval;
5
6             vma->vm_old_ptes = vmalloc(page_count * sizeof(pte_t));
7
8             /* Check if vmalloc is successful */
9             if (!vma->vm_old_ptes)
10              return retval;
11          }
```

1. Check for SAVE context argument
3. If context is already saved, then just return error
6. Vmalloc a size of number of pages times PTE size to the custom pointer that we added to vm_area_struct
10. Return if no memory is allocated

2.2.3 *Loop through all the address in every VMA, hopping PAGE_SIZE. We write protect the PTEs for each page in anonymous VMA. [1]*

```
1           for (addr = vma->vm_start; addr < vma->vm_end; addr += PAGE_SIZE) {
2             ptep = get_locked_pte(mm, addr, &ptl);
3             if (a == 0) {
4               pte = *ptep;
5
6               page = pte_page(pte);
7               get_page(page);
8               vma->vm_old_ptes[count] = *ptep;
9               ptep_set_wrprotect(mm, addr, ptep);
10              pte = pte_wrprotect(*ptep);
11
12              set_pte_at(vma->vm_mm, addr, ptep, pte);
13              atomic_inc(&page->_mapcount);
14
15              /* no need to invalidate: a not-present page won't be cached */
16              update_mmu_cache(vma, addr, ptep);
```

```
17              flush_tlb_page(vma, addr);
```

1. Loop through all the pages in the VMA
2. Get the corresponding PTE of the page with a lock
3. For saving context, proceed
6. Get the page struct with the PTE
8. Store the PTE values in out structure for later to restore
9. MAIN: set the PTE to be write protected for implementing COW during fault to write
12. Now set the PTE with the modified write protect bit set PTE
13. Increase the map count so that the page doesn't get freed
16-17. Update the cache and flush TLB for old values

2.2.4 *If the argument is 1, then restoring the old page must be done, if a context is already saved.*

```
1             } else {
2               /* When argument is 0: Restore */
3               if (vma->vm_old_ptes != NULL) {
4                 /* Restore available */
5                 old_pte = vma->vm_old_ptes[count];
6                 page = pte_page(old_pte);
7                 atomic_dec(&page->_mapcount);
8
9                 page = pte_page(*ptep);
10
11                zap_vma_ptes(vma, addr, PAGE_SIZE);
12
13                set_pte_at(mm, addr, ptep, old_pte);
14                update_mmu_cache(vma, addr, ptep);
15                flush_tlb_page(vma, addr);
16              } else {
17                /* Restore not available */
18                return retval;
19              }
20            }
```

1. Check for RESTORE context argument
3. Check if some context is already saved, if yes then proceed
5. Get the old PTE values from the custom pointer
6-7. Get the page and decrease the map count
11. Release the COWed page that was modified, safely
13. MAIN: Set the PTE pointer to old values to restore
14-15. Update the cache and flush TLB for old values
13. Increase the map count so that the page doesn't get freed
16-18. If the checkpoint is not available, just return back

2.2.5 *Finally, unlock the PTE pointer, and free the VMALLOC'd area.*

```
1             pte_unmap_unlock(ptep, ptl);
2             count++;
3           }
4           if (a == 1 && vma->vm_old_ptes != NULL) {
5             vfree(vma->vm_old_ptes);
6             vma->vm_old_ptes = NULL;
7           }
8         }
9       }
```

```
10        retval = 0;
11        up_read(&mm->mmap_sem);
12      }
13    } else
14      printk("Sorry, invalid argument.\n");
15
16    pte_unmap(ptep);
17
18    return retval;
19  }
```

1. release the lock on PTE pointer
3. Check if some context is already saved, if yes then proceed
4-6. Free the custom pointer
13-14. If argument id neither 0 nor 1

3 PART C

3.1 User-space program used to measure the timing of system calls [2]

Description: For this part of the assignment, a C code was given for performance measurement. We are required to measure the execution time, in terms of number of successful restorations done in given time, of the system call as compared to fork.

```
#include "header.h"

int main(int argc, char *argv[])
{
  int perf_fork, perf_context;

  init_params(argc, argv);
  perf_fork = run_fork();
  printf("Baseline(fork) Throughput:%15d\n", perf_fork);
  perf_context = run_context();
  printf("Optimized(context) Throughput:%12d\n", perf_context);
  printf("Speedup:%33.2f\n", (double) perf_context / perf_fork);
  return 0;
}
```

Algorithm 1: The user space code to measure performance

```
do_good(buff, ACTION_INIT);
while (*continue_work) {
    nr_calls++;
    /***** save context *****/
    ret = syscall(syscall_num, 0);
    assert(ret == 0);
    /* suspicious code */
    do_evil(buff);
    /***** recover context *****/
    ret = syscall(syscall_num, 1);
    assert(ret == 0);
}
do_good(buff, ACTION_VERIFY);
free(buff);
return nr_calls / timeout;
```

Algorithm 2: run_context() in header.h

3.1.1 *Output like:*

```
Baseline(fork) Throughput:             3911
Optimized(context) Throughput:         13651
Speedup:                               3.49
```

4 PART D

4.1 Performance comparison

Description: Reporting the measurements based on multiple runs (minimum 10) along with the standard deviation.

The output is listed in the Table 1.

Table 1: The output of iterations

Iteration	Baseline (fork) Throughput	Optimized (context) Throughput	Speed up
1	5508	14420	2.62
2	4378	11731	2.68
3	3872	13003	3.36
4	3911	13651	3.49
5	5835	14019	2.40
6	5100	13507	2.65
7	5048	14269	2.83
8	5112	13819	2.70
9	4089	12950	3.17
10	5616	14372	2.56
Mean	**4846**	**13574**	**2.85**
Median	**5074**	**13735**	**2.69**
Std. Dev.	**730.32**	**831.068**	**0.365**

As we can see from the Table, we do see that the cycles taken by system call my_precious is very low as compared to that of fork to complete the given task.

Speed up of mean 2.85 is achieved and maximum went up to 3.49 times that of fork based program.

4.2 Limitations

The following possible limitations may exist in this implementation:

- The use of `vmalloc` might be replaced by `kmalloc`.
- The variation in size of program is not tested in this, and hence future scope of the size of memory load may be tested.
- The thread level protection is not taken care in this assignment, hence possibility of lock acquisition for atomic execution may be pondered over.
- The issue of huge pages are not included in this section.

REFERENCES

[1] Bootlin. Linux source code (v5.0) - bootlin, 2019. URL https://elixir.bootlin.com/linux/v5.0/source. [Online].

[2] Ashish Panwar. Sample test cases for the programming assignment, 2019. URL https://github.com/apanwariisc/e0253-os-2020. [Online].