CSE 5433 OS Laboratory
Lab4 Report

Group 8
Sixiang Ma, Xiaokuan Zhang

11/01/2017

1 Objectives

In lab4, our goal is to implement partial functions of process checkpointing at the system level. In particular, this lab focuses on checkpointing memory data for a process. It includes three components:

(a) implement the basic function of checkpointing a memory range specified by a user program;

(b) design an incremental checkpointing scheme;

(c) implement the incremental checkpointing scheme.

2 Basic Checkpointing Scheme

2.1 High-level Design

On a high level, we need to implement a system call like previous labs, which will be called by a userspace process for checkpointing its memory within certain range (begin, end). In that system call, there are mainly two steps we should do:

(a) Traverse the vitual memory area of the process, page by page. If a page is fully within the range, then we need to output it to a file.

(b) Output corresponding pages to a file.

2.2 Implementation

2.2.1 Creating A System Call

This step is similar to what we did in previous labs. we defined two global variables cp_start, cp_end to store the requested range for checkpointing. Also, we defined a $file\_count$ variable to store the current file

counter, and a *filename* char array to store the current filename. After updating the *cp_start*,*cp_end*, we call the *traverse*() function to do the real checkpointing.

```
unsigned long cp_start;
unsigned long cp_end;
int file_count = 0;
char filename[16];

asmlinkage long sys_cp_range(unsigned long *start_addr, unsigned long *end_addr){
    unsigned long startnow;
    unsigned long endnow;
    printk("enter cp_range\n");
    if (copy_from_user(&startnow, start_addr, sizeof(*start_addr)))
        return -EFAULT;
    if (copy_from_user(&endnow, end_addr, sizeof(*end_addr)))
        return -EFAULT;
    printk("\ncp_range: %lx, %lx\n\n", startnow, endnow);
    cp_start = startnow;
    cp_end = endnow;
    traverse();
    printk("exit cp_range\n");
    ++file_count;
    return 0;
}
```

2.2.2 Traversing the Memory Area of a Process

We can use `area = current->mm->mmap` to get a `vm_area_struct*`, which points to the start of the vm area of the current process. Since it is a linked list, we can use `area = area->vm_next` to traverse this list to visit each vm area.

For each vm area, we visit each page to determine if it is fully within the requested range. If it is, we use *vfs_write*() to write this page to a file. To ensure the safety, we also do a page walk the retrieve the `pte` of an virtual address. If `pte && pte_read(*pte)`, we write this page to a file. The page walk is performed in *lookup_address_userspace*() function.

```
pte_t *lookup_address_userspace(struct mm_struct* mm, unsigned long address)
{
    pgd_t *pgd;
    //pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    pgd = pgd_offset(mm, address);

    if (pgd_none(*pgd) || pgd_bad(*pgd))
        return NULL;

    pmd = pmd_offset(pgd, address);
    if (pmd_none(*pmd)||pmd_bad(*pmd)||!pmd_present(*pmd))
        return NULL;

    pte = pte_offset_kernel(pmd, address);
```

```
    if (pte_none(*pte)||!pte_present(*pte))
        return NULL;
    return pte;
}
```

```
void traverse(void){
    struct task_struct *tsk;
    struct vm_area_struct *area;
    struct mm_struct *mm;
    int ret;
    unsigned long addr;
    pte_t *pte;
    printk("enter traverse\n");

    ret = open_file();
    if (ret > -1) {
        tsk = current;
        if (tsk->mm) {
            mm = tsk->mm;
            area = tsk->mm->mmap;
            while (area) {
                printk("vm_start: %lx, vm_end: %lx\n", area->vm_start, area->vm_end);
                addr = area->vm_start;
                while (addr +PAGE_SIZE <= area->vm_end) {
                    if (addr>=cp_start && addr+PAGE_SIZE <=cp_end) {
                        pte = lookup_address_userspace(mm, addr);

                        if (pte && pte_read(*pte)) {
                            vfs_write(filp, (void*)addr, PAGE_SIZE, &pos);
                            vfs_write(filp, "\n", 1, &pos);
                            printk("addr written: %lx\n", addr);
                        }
                    }
                    addr += PAGE_SIZE;
                }
                area = area->vm_next;
            }
        }
        printk("exit traverse\n");
    }
    close_file();
}
```

2.2.3 Writing to a File

We use $vfs\_write()$ function to write to a file. We defined three global variables $pos, old_fs, filp$ and two helper functions, $open\_file()$ and $close\_file()$, which are called in $traverse()$.

```
loff_t pos = 0;
mm_segment_t old_fs;
struct file *filp = NULL;
int open_file(void){
    printk("enter open_file\n");
    snprintf(filename,sizeof(filename),"%s%d","cp_out_",file_count);
```

```
7
8       old_fs = get_fs();
9       set_fs(KERNEL_DS);
10      filp = filp_open(filename, O_WRONLY|O_CREAT, 0644);
11      if (IS_ERR(filp))
12      {
13          printk("OPEN ERROR!\n");
14          return -1;
15      }
16      printk("exit open_file\n");
17      return 0;
18
19  }
20
21  void close_file(void){
22      printk("enter close_file\n");
23      filp_close(filp,NULL);
24      set_fs(old_fs);
25      pos = 0;
26      printk("exit close_file\n");
27  }
```

3 Incremental Checkpointing Scheme

3.1 High-level Design

In this section, the following three questions are answerd at conceptual design level:

(1) How does your scheme achieve the incremental checkpointing?

(2) What are the special cases your scheme need to handle?

(3) How does your scheme handle these special cases?

3.1.1 The Scheme of Incremental Checkpointing

We design our incremental checkpointing by borrowing the idea of Copy-on-Write(COW). In the incremental checkpointing, after finishing checkpointing for a certain page, we mark it as read-only by setting the corresponding bit of its page table entry to zero. If there are no write operations to this page afterwards, during the next checkpoint, we can skip this page by checking whether its state is still read-only. On the other hand, if writes to that page indeed happens after the first checkpointing, the fore-mentioned bit in the page table entry will be changed to one (writable) during the page fault handling procedure.

3.1.2 Special Cases

The method described in the previous section suffices to achieve our incremental checkpointing except the special cases which will change the state of table page entry as wirte protect as well.

For example, if fork system call is invoked in the process where we are going to perform checkpointing, the writable bit in the page table entry can be ”overrided” to zero even though writes really happened before. As a result, we will falsely miss to perform checkpointing for those pages.

3.1.3 How to Handle Special Cases

To address this issue, an unused bit in the page table entry will be used to serve as a flag bit (we call it override bit here) to record the situations where writable bit of a page table entry is modified to zero by other sources besides our increamental memory checkpointing mechanism. Then, during the next checkpointing, we will perform the memory checkpointing for the pages whose state is either writable or overrided.

3.2 Implementation

In this section, we describe the implementation details of our incremental checkpointing mechanism.

3.2.1 Modification to the Kernel

In the header file of $include/asm-i386/pgtable.h$, we added two utility functions to facilitate the use of our newly introduced override bit.

Function $pte\_override()$ returns the override bit and function $pte\_clearoverride\_and\_wrprotect()$ sets both the writable and override bit to zero.

```
1 static inline int pte_override(pte_t pte)
2 { return (pte).pte_low & _PAGE_UNUSED1; }
3
4 static inline pte_t pte_clearoverride_and_wrprotect(pte_t pte)
5 { (pte).pte_low &= ~_PAGE_RW; (pte).pte_low &= ~_PAGE_UNUSED1; return pte; }
```

Also, we modified the existing fucntions which will set pte state as write protect.

```
1 static inline pte_t pte_wrprotect(pte_t pte)
2 { (pte).pte_low &= ~_PAGE_RW; (pte).pte_low |= _PAGE_UNUSED1; return pte; }
3
4 static inline void ptep_set_wrprotect(pte_t *ptep)
5 { clear_bit(_PAGE_BIT_RW, &ptep->pte_low); set_bit(_PAGE_BIT_UNUSED1, &ptep->pte_low)
    ;}
```

3.2.2 Modification to the Basic Checkpointing Mechanism

We extended our basic checkpointing mechanism to support incremental checkpointing. The modifications to the basic mechanism consist of two parts.

First, as shown in the line 2, we only perform memory checkpointing for the pages whose state is writable or overdid, which means either these pages have been written or its writable bit has been overrided to zero (e.g., due to COW) before.

Second, after checkpointing, we change the state of pages to write protect and not overrided by flipping the corresponding flag bits in the page table entry, which is achieved by simply invoking the newly introduced *pte_clearoverride_and_wrprotect*() function. After that, we also need to update the entry in the page table as well as flush cache. By doing this, we can skip the memory checkpointing for the pages the state of which are not written and not overrided.

```
if ( pte && pte_read(*pte)) {
    if (pte_write(*pte) || pte_override(*pte)) {
        vfs_write(filp, (void*)addr, PAGE_SIZE, &pos);
        vfs_write(filp, "\n", 1, &pos);
        printk("addr written: %lx\n", addr);
    }
    spin_lock(&mm->page_table_lock);
    wrprotect_notoverride_pte = pte_clearoverride_and_wrprotect(*pte);
    set_pte(pte, wrprotect_notoverride_pte);
    flush_cache_page(area, area->vm_start);
    ptep_establish(area, area->vm_start, pte, wrprotect_notoverride_pte);
    update_mmu_cache(area, area->vm_start, wrprotect_notoverride_pte);
    spin_unlock(&mm->page_table_lock);
}
```