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

explain the high-level idea

3.2 Implementation

detailed implementation