Lab4 Report

Yao Wei (wei.849@osu.edu)
Zuanxu Gong (gong.366@osu.edu)

1. Basic checkingpointing of memory

1) Design

①sys_cp_range(unsigned long *start_addr, unsigned long *end_addr)
Implement a system call which takes a range of virtual addresses as the input parameters. This system call will call function walk_through_VMAs() to walk through VM areas.

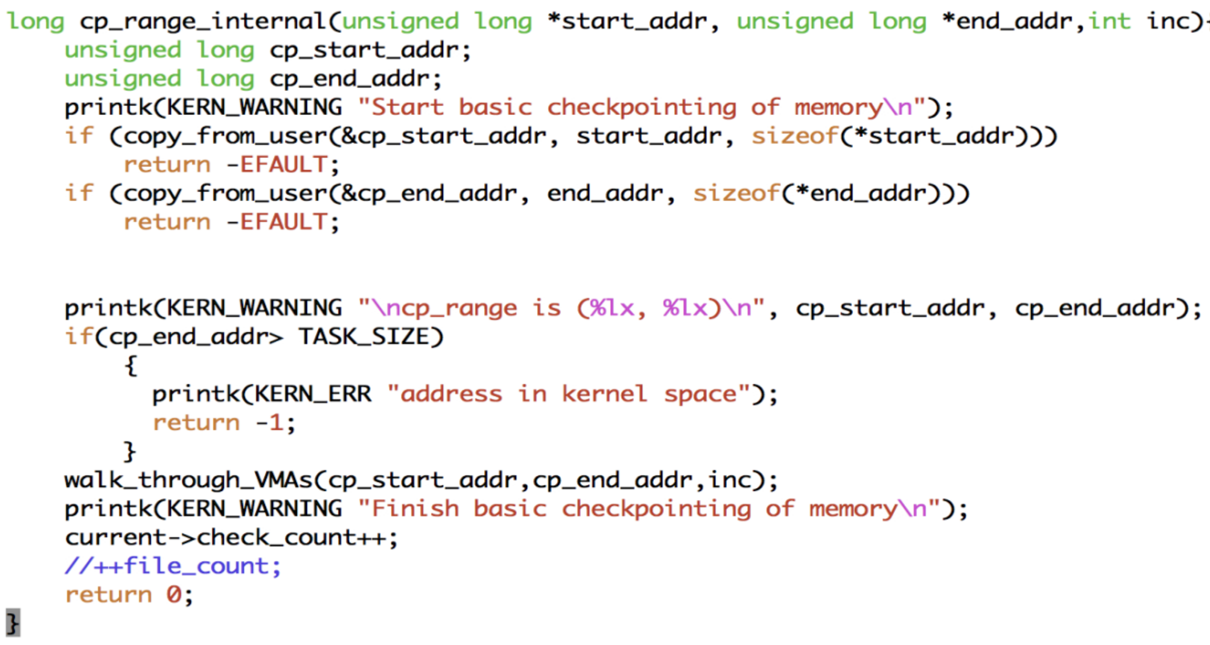
②walk_through_VMAs(unsigned long cp_start_addr,unsigned long cp_end_addr,int is_inc)
Implement the function walk_through_VMAs() to walk through each VMA within the range. For each area, we will go through page tables to check whether they are in the range of our input virtual addresses.

③lookup_address_page(struct mm_struct* mm, unsigned long addr)
For a page table whose addresses are completely within the range of input virtual addresses, we will use function lookup_address_page() to look up the Page Table Entry(PTE) of this page and use vfs_write() to save the content of this page to the local file. When looking up page table entry, we will check pte_none(pte), pte_present(pte) and zero pages for performance optimization.

④ open_file(char* filename,mm_segment_t *old_fsp) and close_file(mm_segment_t old_fs,struct file *filp)
Implement open_file() and close_file() functions to open and close the local file. The file name is designed as cp_P + Number.

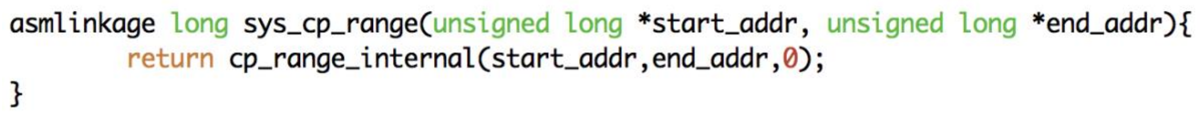
2) Implementation

Step1:
Implement a system call which takes a range of virtual addresses as the input parameters. This system call will call function walk_through_VMAs() to walk through VM areas.



```
long cp_range_internal(unsigned long *start_addr, unsigned long *end_addr,int inc){
    unsigned long cp_start_addr;
    unsigned long cp_end_addr;
    printk(KERN_WARNING "Start basic checkpointing of memory\n");
    if (copy_from_user(&cp_start_addr, start_addr, sizeof(*start_addr)))
        return -EFAULT;
    if (copy_from_user(&cp_end_addr, end_addr, sizeof(*end_addr)))
        return -EFAULT;


    printk(KERN_WARNING "\ncp_range is (%lx, %lx)\n", cp_start_addr, cp_end_addr);
    if(cp_end_addr> TASK_SIZE)
        {
          printk(KERN_ERR "address in kernel space");
          return -1;
        }
    walk_through_VMAs(cp_start_addr,cp_end_addr,inc);
    printk(KERN_WARNING "Finish basic checkpointing of memory\n");
    current->check_count++;
    //++file_count;
    return 0;
}
```

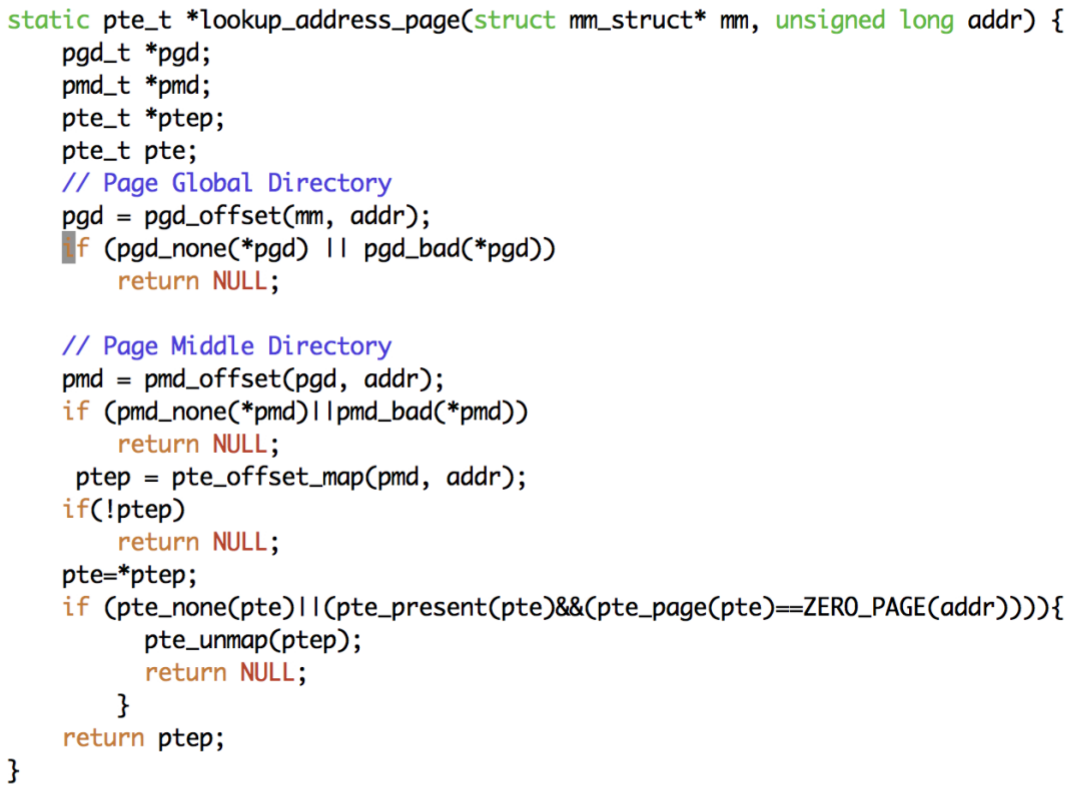


```
asmlinkage long sys_cp_range(unsigned long *start_addr, unsigned long *end_addr){
        return cp_range_internal(start_addr,end_addr,0);
}
```

Step 2:
Implement the function walk_through_VMAs() to walk through each VMA within the range.
Firstly, we use **find_vma()** to find the first VMA that may contains our start address. Then for each VMA starting from that VMA, we will go through page tables to check whether they are in the range of our input virtual addresses. For a page table whose addresses are completely within the range of input virtual addresses, we will use function lookup_address_page() to look up the Page Table Entry(PTE) of this page and use vfs_write() to save the content of this page to the local file. (After writing one page, we will write another page in the new line)

For each VMA, we check the VM_IO flag first, if VM_IO flag is set, then skip that VMA.

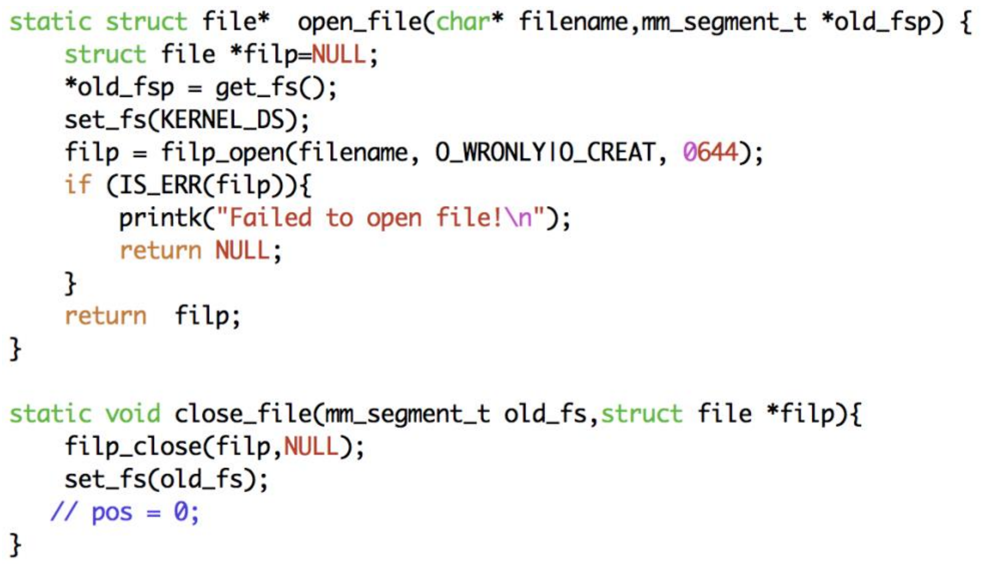
Step 3:
Implement the function lookup_address_page() to get the page table entry of the address. When looking up page table entry, we will check pte_none(pte), pte_present(pte) and zero pages for performance optimization.



```
static pte_t *lookup_address_page(struct mm_struct* mm, unsigned long addr) {
    pgd_t *pgd;
    pmd_t *pmd;
    pte_t *ptep;
    pte_t pte;
    // Page Global Directory
    pgd = pgd_offset(mm, addr);
    if (pgd_none(*pgd) || pgd_bad(*pgd))
        return NULL;

    // Page Middle Directory
    pmd = pmd_offset(pgd, addr);
    if (pmd_none(*pmd)||pmd_bad(*pmd))
        return NULL;
     ptep = pte_offset_map(pmd, addr);
    if(!ptep)
        return NULL;
    pte=*ptep;
    if (pte_none(pte)||(pte_present(pte)&&(pte_page(pte)==ZERO_PAGE(addr)))){
          pte_unmap(ptep);
          return NULL;
        }
    return ptep;
}
```

Step 4:
Implement open_file() and close_file() functions to open and close the local file. We add a **int check_count** member to the task_struct to track the current checkpoint file number. For normal checkpoint algorithm, file name would be cp_pid_checkcount



```
static struct file*  open_file(char* filename,mm_segment_t *old_fsp) {
    struct file *filp=NULL;
    *old_fsp = get_fs();
    set_fs(KERNEL_DS);
    filp = filp_open(filename, O_WRONLY|O_CREAT, 0644);
    if (IS_ERR(filp)){
        printk("Failed to open file!\n");
        return NULL;
    }
    return  filp;
}

static void close_file(mm_segment_t old_fs,struct file *filp){
    filp_close(filp,NULL);
    set_fs(old_fs);
   // pos = 0;
}
```

2. Incremental checkpointing scheme

1) Design

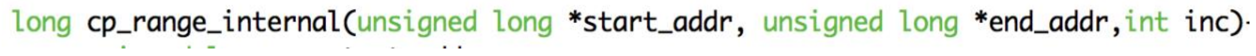
At the first time, we will take full checkpoint.

We can use **pte_dirty**() function to test whether the page has been modified since last page fault.

After each checkpoint, we have to clear the dirty bit so we won't store that page at next checkpoint if not modified.

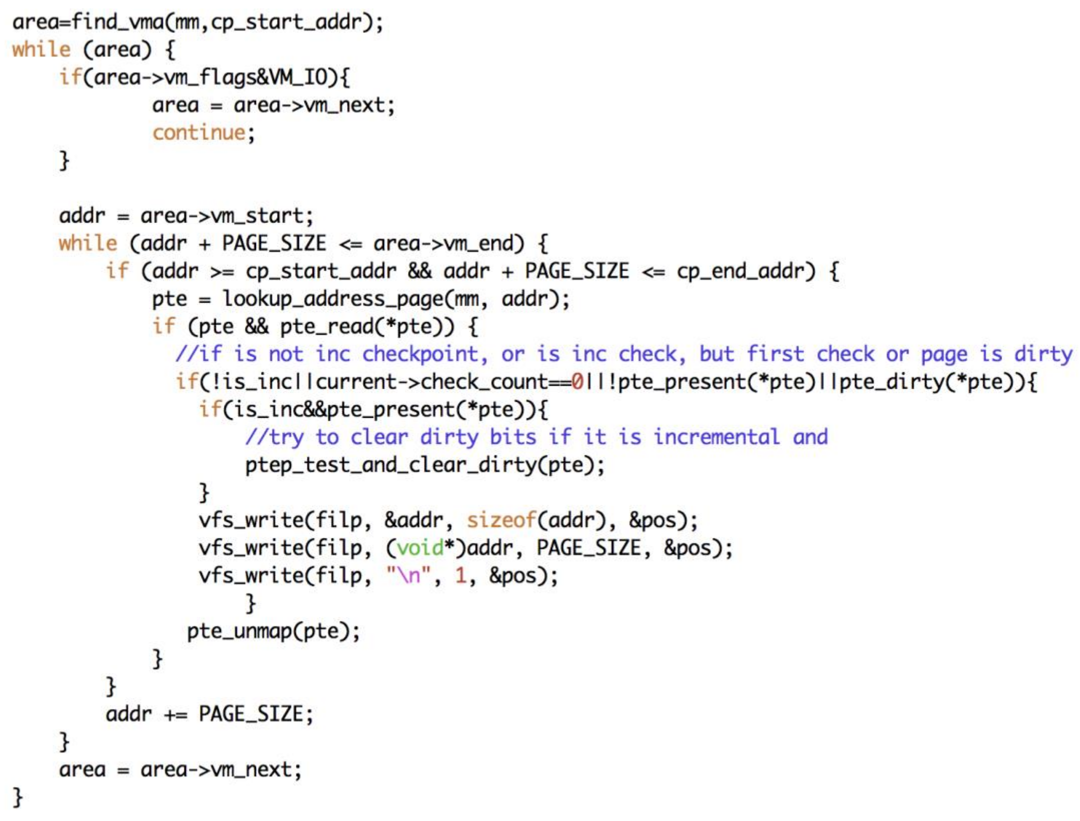
2) Implementation

Since the Incremental checkpoint scheme is based on the based checkpoint algorithm, we merge two algorithm into one function



```
long cp_range_internal(unsigned long *start_addr, unsigned long *end_addr,int inc)
```

we use **inc** to control whether its basic or incremental algorithm.



```
area=find_vma(mm,cp_start_addr);
while (area) {
    if(area->vm_flags&VM_IO){
            area = area->vm_next;
            continue;
    }

    addr = area->vm_start;
    while (addr + PAGE_SIZE <= area->vm_end) {
        if (addr >= cp_start_addr && addr + PAGE_SIZE <= cp_end_addr) {
            pte = lookup_address_page(mm, addr);
            if (pte && pte_read(*pte)) {
              //if is not inc checkpoint, or is inc check, but first check or page is dirty
              if(!is_inc||current->check_count==0||!pte_present(*pte)||pte_dirty(*pte)){
                if(is_inc&&pte_present(*pte)){
                    //try to clear dirty bits if it is incremental and
                    ptep_test_and_clear_dirty(pte);
                }
                vfs_write(filp, &addr, sizeof(addr), &pos);
                vfs_write(filp, (void*)addr, PAGE_SIZE, &pos);
                vfs_write(filp, "\n", 1, &pos);
                    }
               pte_unmap(pte);
            }
        }
        addr += PAGE_SIZE;
    }
    area = area->vm_next;
}
```

if the page table entry exists and page is dirty or it's first time we take checkpoint (check_count=0) we dump the page, clear the dirty bit if set.