Operating System Lab Project
Theme B
Virtual Memory for the Geek OS
Report for this week

Rohit Kumar
120050028
rohit@cse.iitb.ac.in

Deependra Patel
120050032
deependra@cse.iitb.ac.in

Jayesh Bageriya
120050022
jayeshbageriya@cse.iitb.ac.in

Abhishek Thakur
120050008
abhishekthakur@cse.iitb.ac.in

April 4, 2015

1 Modification for using page table (Multilevel Page Table)

The first step is to modify the code to use page tables rather than just segments to provide memory protection. Now every region of the memory that is being accessed should have an entry in the page table.

Now, there will be a single page table for all kernel only threads, and a page table for each user process. In addition, the page tables for user mode processes will also contain entries to address the kernel mode memory (memory that can be accessed only by kernel). The kernel memory should be a one to one mapping of all of the physical memory in the processor. The page table entries for this memory have to be marked so that this memory is only accessible from kernel mode.

This is achieved by creating a page directory and page table entries for the kernel threads by writing code which will initialize the page tables and enables paging mode in the processor.

```
1
/*
3 * Kernel Page Directory initialization
 */
5 pde_t *g_kernel_pde = {0};
```

Now, the page tables are set. A new page directory is allocated using Alloc_Page() and then page tables for the entire region that will be mapped into this memory context are allocated (done in the for loop in below code). Appropriate fields in the page tables and page directories, (represented by the pde_t and pde_t data-types) are filled.

```
/*
 * Page directory entry datatype.
 * If marked as present, it specifies the physical address
 * and permissions of a page table.
 */
typedef struct {
    uint_t present:1;
    uint_t flags:4;
    uint_t accessed:1;
    uint_t reserved:1;
    uint_t largePages:1;
    uint_t globalPage:1;
    uint_t kernelInfo:3;
    uint_t pageTableBaseAddr:20;
} pde_t;

/*
 * Page table entry datatype.
 * If marked as present, it specifies the physical address
 * and permissions of a page of memory.
 */
typedef struct {
    uint_t present:1;
    uint_t flags:4;
    uint_t accessed:1;
    uint_t dirty:1;
    uint_t pteAttribute:1;
    uint_t globalPage:1;
    uint_t kernelInfo:3;
    uint_t pageBaseAddr:20;
} pte_t;
```

```

/*
 * Initialize virtual memory by building page tables
 * for the kernel and physical memory.
 */
pde_t *g_kernel_pde = {0};

void Init_VM(struct Boot_Info *bootInfo) {
    int  kernel_pde_entries;
    int  whole_pages;
    int  i,j;
    uint_t mem_addr;
    pte_t * cur_pte;
    whole_pages = bootInfo->memSizeKB/4;
    Print("whole pages are %d\n",whole_pages);
    kernel_pde_entries = whole_pages/NUM_PAGE_DIR_ENTRIES + (whole_pages %
    NUM_PAGE_DIR_ENTRIES == 0 ? 0:1);
    Print("kdep %d\n",kernel_pde_entries);
    Print("the kernel_pde_entries is %d\n",kernel_pde_entries);
    //KASSERT(0);
```

```
22      g_kernel_pde=(pde_t *)Alloc_Page();
        Print("KERNALADD: %d\n", (int)g_kernel_pde);
24
        memset(g_kernel_pde,'\0',PAGE_SIZE);
26
        pde_t * cur_pde_entry;
28      cur_pde_entry=g_kernel_pde;
        mem_addr=0;
30      for(i=0;i<kernel_pde_entries;i++){
            cur_pde_entry->present=1;
32          cur_pde_entry->flags=VM_WRITE | VM_USER;

34          Print("I: %d\n", i);

36          cur_pte=(pte_t *)Alloc_Page();
            KASSERT(cur_pte!=NULL);
38
            memset(cur_pte,'\0',PAGE_SIZE);
40          cur_pde_entry->pageTableBaseAddr=(uint_t)cur_pte>>12;
            mem_addr = i*NUM_PAGE_TABLE_ENTRIES*PAGE_SIZE;
42          int last_pagetable_num;
            last_pagetable_num=whole_pages%NUM_PAGE_TABLE_ENTRIES;
44          if(last_pagetable_num==0 || i!=(kernel_pde_entries-1)){
              last_pagetable_num=NUM_PAGE_TABLE_ENTRIES;
46          }

48          for(j=0;j<last_pagetable_num;j++){
                cur_pte->present=1;
50              cur_pte->flags=VM_WRITE|VM_USER;
                cur_pte->pageBaseAddr=mem_addr>>12;
52
                cur_pte++;
54              mem_addr+=PAGE_SIZE;
            }
56          cur_pde_entry++;
        }
58      Enable_Paging(g_kernel_pde);
        Install_Interrupt_Handler(14,Page_Fault_Handler);
60  }
```

Finally, to enable paging for the first time, an assembly routine, Enable Paging() is called that takes the base address of the page directory as a parameter and then load the passed page directory address into register cr3, and then set the paging bit in cr0.

```
1
;
3   ; Start paging
    ; load crt3 with the passed page directory pointer
5   ; enable paging bit in cr2
    align 8
7   Enable_Paging:
      mov eax, [esp+4]
9     mov cr3, eax
      mov eax, cr3
11    mov cr3, eax
      mov ebx, cr0
```

```
13    or   ebx, 0x80000000
      mov cr0, ebx
15    ret
```

2 Handling Page Faults

For working with paging, the operating system needs to handle page faults. To do this a page fault interrupt handler should be present.

The first thing the page fault handler will need to do is to determine the address of the page fault which can be found out by calling the Get Page Fault Address() function (present in geekos/paging.h. Also, the errorCode field of the Interrupt State data structure passed to the page fault interrupt handler contains information about the faulting access. This information is defined in the faultcode_t data type (defined in geekos/paging.h).

```
/*
2  * Datatype representing the hardware error code
   * pushed onto the stack by the processor on a page fault.
4  * The error code is stored in the "errorCode" field
   * of the Interrupt_State struct.
6  */
typedef struct {
8      uint_t protectionViolation:1;
       uint_t writeFault:1;
10     uint_t userModeFault:1;
       uint_t reservedBitFault:1;
12     uint_t reserved:28;
} faultcode_t;
```

Given a virtual address at which page fault arises, we get the index of the page directory corresponding to that address (10 MSB's) and the index of the page level of that page directory (next 10 MSB's). The address of the page directory entry is found by adding the base pointer of the first level page table (directory level) (userContext-¿pageDir) and the page directoy index obtained as above. This consists of 20 bits.

If the directory entry is present in the directory level page table, then the exact address of the page entry is obtained by adding page_addr of the virtual address(obtained above) to left shifted address of page directory entry(obtained above).

```
/*
 * Handler for page faults.
 * You should call the Install_Interrupt_Handler() function to
 * register this function as the handler for interrupt 14.
 */
/*static*/ void Page_Fault_Handler(struct Interrupt_State *state) {
    ulong_t address;
    faultcode_t faultCode;

    KASSERT(!Interrupts_Enabled());

    /* Get the address that caused the page fault */
    address = Get_Page_Fault_Address();
    Debug("Page fault @%lx\n", address);

    if (address < 0xfec01000 && address > 0xf0000000) {
        KASSERT0(0, "page fault address in APIC/IOAPIC range\n");
    }

    /* Get the fault code */
    faultCode = *((faultcode_t *) & (state->errorCode));

    /* rest of your handling code here */
   // TODO_P(PROJECT_VIRTUAL_MEMORY_B, "handle page faults");

    ulong_t page_dir_addr= PAGE_DIRECTORY_INDEX(address);
    ulong_t page_addr= PAGE_TABLE_INDEX(address);
    pde_t * page_dir_entry=(pde_t*)userContext->pageDir+page_dir_addr;
    pte_t * page_entry= NULL;

    if(page_dir_entry->present)
    {
        page_entry=(pte_t*)((page_dir_entry->pageTableBaseAddr) << 12);
        page_entry+=page_addr;
    }
    else
    {
    // Error occured. Not implemented for this part
    }
```

3 Things left to be done

- Finish Page Fault Handler and Page Replacement
- Implement Pinned Pages
- Creating swap space for a process in a file
- Complete user level virtual memory implementation
- TLB

4 Work Distribution

- Deependra and Rohit : Multilevel Page Table Implementation (Init_VM)
- Jayesh and Abhishek : Page Fault Handler