CS 344-OS_LAB Assignment-3 Report

Group Name: G7
Submission Date:13-11-2020

Team Members:

Ashish Kumar Baranwal (180123006) Dept:MnC
Bhargab Gautom (180123008) Dept:MnC
Harsh Yadav (180123015) Dept:MnC
Karan Gupta (180123064) Dept:Mnc

Part A) Implementing Lazy Memory Allocation

After applying the patch file, and running echo hi, the following output was obtained, as expected.

```
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$
```

In sbrk, we just increase proc->sz, but do not actually allocate the page in physical memory to the process. As physical memory is not allocated, when that location is tried to be accessed then page fault occurs.

In trap.c, void trap(struct trapframe *tf), in the switch condition we add the following code snippet.

```
  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
              tf->trapno, cpuid(), tf->eip, rcr2());
      panic("trap");
    }
    // In user space, assume process misbehaved.


//Lazy Allocation ----------------------------------------------

    //Page Fault due to physical memory not being allocated beforehand
    if (rcr2() <= myproc()->sz) {
      if (tf->trapno == T_PGFLT) {
        char *new_page;
        uint page_start;
        page_start = PGROUNDDOWN(rcr2());
        new_page = kalloc();    //Assign a page(4096 B) of physical memory to new_page
        if (new_page == 0) {
          cprintf("allocuvm out of memory\n");
          myproc()->killed = 1;
          return;
        }
        memset(new_page, 0, PGSIZE);
        mappages(myproc()->pgdir, (char*)page_start, PGSIZE, V2P(new_page), PTE_W|PTE_U);
        break;
      }
    }
    else
        cprintf("Page Fault cannot be handled\n");
//-------------------------------------------------------------

    cprintf("pid %d %s: trap %d err %d on cpu %d "
            "eip 0x%x addr 0x%x--kill proc\n",
            myproc()->pid, myproc()->name, tf->trapno,
```

- First we check whether the fault was due to physical memory not being allocated, we use rcr2() which reads the CR2 register storing the address at which page fault occurred. Then we compare the trap number to T_PGFLT (trap 14) which would mean that the page fault is due to our implementation of lazy memory allocation.

- If the conditions are met then we try to assign a new page using kalloc(), which allocates a page of 4096 bytes to the user program. If unsuccessful, then that process is killed.

- Memset is used to fill the new allocated memory with zeros. Then we run mappages to update the page table for the process.

- After adding the above code to trap.c, simple shell commands start executing properly.

In **vm.c**, the keyword static was removed from mappages() and it was called as extern by **trap.c**.

Part B) Implementing Paging mechanism

Task 1: Kernel Processes

- Kernel processes always run in kernel mode and can change its parent process to init process. We used kernel processes to implement the paging mechanism.
- The function *create_kernel_process(const char *name, void (*entrypoint)())* was created in proc.c.
- Its implementation is similar to allocproc() or fork() where a new process is created but all the data is initialized manually for the new process.
- The trapframe and various information of the proc struct has been set as shown in the code snippet below:

```
void
create_kernel_process(const char *name, void (*entrypoint) ()){
  struct proc *np;

  // Allocate process
  if ((np = allocproc()) == 0)
    panic("Failed to allocate.");


  if((np->pgdir = setupkvm()) == 0){
    //Copied from fork()
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    panic("Failed to setup pgdir for kernel process.");
  }
  np->sz = PGSIZE;
  np->parent = initproc; // Mentioned in the assignment
  memset(np->tf, 0, sizeof(*np->tf));
  np->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  np->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  np->tf->es = np->tf->ds;
  np->tf->ss = np->tf->ds;
  np->tf->eflags = FL_IF;
  np->tf->esp = PGSIZE;
  np->tf->eip = 0;  // beginning of initcode.S

  // Clear %eax so that fork return 0 in the child
  np->tf->eax = 0;
  np->cwd = namei("/");

  safestrcpy(np->name, name, sizeof(name));

  // lock to force the compiler to emit the np-state write last.
  acquire(&ptable.lock);
  np->context->eip = (uint)entrypoint;
  np->state = RUNNABLE;
  release(&ptable.lock);
}
```

- The arguments name and entrypoint are used as follows:
    - Name of the new process is set as name using *safestrcpy(np->name, name, sizeof(name));*
    - context->eip register is set to the function pointer of entrypoint, so the process will start running at the function entrypoint on starting.
- This function is called twice in main.c for creating the swapping out and swapping in processes, which are explained in the next tasks(2 & 3).
- *create_kernel_process("swapin", swapin);*
  create_kernel_process("swapout", swapout);
- These function calls create processes swapin and swapout which run handle the memory management required for implementing paging in the xv6 OS.

Task 2: Swapping out mechanism

- First implemented a function handle_pg_default() which handles the trap “T_PGFLT”
- After this select a victim page to swap it to the disk using approximate LRU implementation.

- For all pages in the user virtual space if mapping exists and if not dirty, is present in main memory and access bit not set, these conditions needs to be checked through flags defined in mmu.h i.e. PTE_A, PTE_P.
- After selecting the victim page the, write this page to disk clear present bit as the page needs to be swapped to the disk then swap victim page to disk.
- Store block id and swapped flag in the pte entry whose page was swapped to the disk. So, when next time this pte is dereferenced, we know that the page has been swapped to the disk and we can bring this page again to memory.

Task 3: Swapping in mechanism

- Once the page fault is handled and the victim page is swapped out so now swapping in a page can be done.
- Now a physical page has been swapped to disk and free, so this time we will get a physical page for sure.
- So run kalloc() to allocate a physical page, map this physical page to virtual page (addr) and set the access bit of the page.
- In another case if the page was swapped earlier and is not present in the main memory which can be checked by the flag i.e. PTE_SWAPPED, then read from the disk using the same block id which was earlier saved and remap the page to the virtual page (addr).

Task 4: Sanity Test

- We have created a file memtest.c which is a user-program to test the swapping mechanism for paging.
- Forked 20 child processes from the original parent process memtest.
- In each forked process 10 iterations were run which allocated 4096 bytes to new_mem pointer.
- If this memory allocation fails, then “memtest failed” is printed and the process exits.
- Otherwise the function “func_mem()” is called which is created to add values to the 4096 bytes of memory that was allocated.
- func_mem() takes a pointer as an argument and fills 1024 integers of size 4 bytes into the space of 4096 bytes and the set values inserted is the Fibonacci series modulo 10000.
- In the same function, the values stored in the page is validated by comparing its each cell to the actual value of the element in the Fibonacci sequence. If all values match then the function returns 1 else 0.
- The flag variable is cleared if func_mem returns 0, then “memtest failed” is printed. If func_mem returns 1, then “memtest OK” is printed.
- When all the child processes have run, then it prints out “memtest Done”.

In case of a successful memest run the following output is obtained on the shell: